Nancy Ann Lynch

OSZTOTT ALGORITMUSOK

KISKAPU

A könyv az Oktatási Minisztérium támogatásával, a Felsőoktatási Pályázatok Irodája által lebonyolított felsőoktatási tankönyvtámogatási program keretében jelent meg.

Az eredeti mű:
Nancy Ann Lynch: Distributed Algorithms
Morgan Kaufmann Publishers, Inc., San Francisco, 2000

A magyar nyelvű kiadás munkatársai:

Alkotó szerkesztő:
Iványi Antal

Lektorok:
Csirik János, Horváth Zoltán, Kiss Attila, Kormos János, Márkus Tibor

ISBN: 963-9301-03-5

Kiadja a Kiskapu Kft.
1081 Budapest, Népszínház utca 31.
Telefon: (+36-1) 303-9119, (+36-1) 477-0443 Fax: (+36-1) 303-1619
Honlap: http://www.kiskapu/hu
Elektronikus cím: kiskapu@kiskapu.hu
Felelős szerkesztő: Szy György
Műszaki szerkesztő: Csutak Hoffmann Levente

Nyomás és kötés: Debreceni Kinizsi Nyomda
Felelős vezető: Bördős János igazgató

Tartalomjegyzék

Előszó a magyar nyelvű kiadáshoz

A mai számítógépek jelentős része hálózatba van kapcsolva, időnként meghibásodik, jellemző adataik – például számítási és üzenetküldési sebességük, memóriájuk nagysága, megbízhatóságuk – lényegesen különböznek az adott hálózat többi számítógépének jellemző adataitól.

Ezért értékes és sikeres része az informatikai szakirodalomnak **Nancy Ann Lynch** *Distributed Algorithms* című könyve, amelyet 1996-os megjelenése óta évente újra kiadnak.

A magyar nyelvű kiadás megjelenéséhez nagy segítséget jelentett az **Oktatási Minisztérium** támogatása. A támogatás lehetővé teszi, hogy az Olvasók az előállítási költségnél olcsóbban jussanak hozzá ehhez a tankönyvhöz.

A könyvet kiegészítettük a szerző által ajánlott új gyakorlatokkal. Az irodalomjegyzéket friss hazai és magyar nyelvű hivatkozásokkal bővítettük és a szövegben alkalmazott jelölésekkel összhangban lévő új ábrákat rajzoltunk. Az irodalomjegyzékből hiperszöveget készítettünk (az élő elemeket a nyomtatott változatban aláhúzás jelzi). Az anyag könnyebb megértését a tárgymutató elején részletezett tipográfiai eszközökkel segítjük.

A szöveg fordítását és lektorálását a Babeş-Bolyai Egyetem, a Debreceni Egyetem, az Eötvös Loránd Tudományegyetem és a Szegedi Tudományegyetem oktatói végezték. Az ábrákat Locher Kornél rajzolta, a LATEX-problémákat Belényesi Viktor és Csörnyei Zoltán oldották meg. A hiperszöveget Iványi Anna készítette. A könyv nyelvi stílusát Rézműves László javította.

A könyv gyors megjelenéséhez szükség volt a **Kiskapu Kiadó** és a **Debreceni Kinizsi Nyomda** megkülönböztetett figyelmére.

A könyv irodalomjegyzékét, tárgymutatóját folyamatosan bővítjük, javítjuk. A `http://people.inf.elte.hu/tony/books/osztalg` címről letölthetők a friss változatok.

Arra számítunk, hogy rövidesen újabb magyar kiadásra lesz igény. Ebben szeretnénk az első kiadás hibáit javítani. Ezért kérjük az Olvasókat, hogy javaslataikat (lehetőleg pontosan megjelölve a hiba helyét és megadva a javasolt új szöveget) küldjék el a kolofon-oldalon felsorolt szakemberek bármelyikéhez (a fenti honlapon a kolofon-oldal élő változata is megtalálható).

Iványi Antal
alkotó szerkesztő
tony@inf.elte.hu

Budapest, 2004. november 4.

Előszó az angol nyelvű kiadáshoz

Az osztott algoritmusok olyan algoritmusok, melyeket olyan hardveren való alkalmazásra terveztek, amely sok, egymással összekapcsolt processzorból áll. Az osztott algoritmusok egyes részei egyidejűleg és egymástól függetlenül hajtódnak végre, mindegyik rész csak korlátozott mennyiségű információval rendelkezik. Feltételezzük, hogy ezek az algoritmusok akkor is helyesen működnek, ha az egyes processzorok és kommunikációs csatornák különböző sebességűek és a rendszer bizonyos elemei meghibásodnak.

Az osztott algoritmusokat széles körben alkalmazzák, például a telekommunikáció, osztott információfeldolgozás, tudományos számítások és a valósidejű folyamatirányítás területén. A mai telefonhálózatok, repülőjegy-foglalási, banki, globális információs, időjárás-előrejelző, repülőgépeket és atomerőműveket irányító rendszerek mindegyikében alapvető szerepet játszanak. Nyilvánvaló annak fontossága, hogy ezek az algoritmusok helyesen és gyorsan hajtódjanak végre. Mivel ezek az algoritmusok bonyolult hardveren futnak, tervezésük rendkívül nehéz feladat.

A könyv átfogó bevezetést tartalmaz az osztott algoritmusok elméletébe – a legjellemzőbb algoritmusok és megoldhatatlansági eredmények összefoglalását adja egyszerű automataelméleti modellek segítségével. Az algoritmusokat pontosan meghatározott bonyolultsági mértékek segítségével elemezzük. Összegezve, a könyv kitűnő alapokat kínál az osztott algoritmusok mély megértéséhez.

A könyv úgy íródott, hogy különböző olvasórétegek számára is hasznos legyen. Elsősorban úgy van felépítve, hogy a felsőbbéves informatikai előadásokhoz tankönyvként szolgáljon – különösen azon hallgatók számára, akiket a számítógéprendszerek gyakorlata és elmélete egyaránt érdekel. A könyv az osztott rendszerek tervezői számára tartott rövid továbbképző tanfolyamokhoz jegyzetként használható. Végül az is cél volt, hogy a könyvet kézikönyvként használhassák a tervezők, hallgatók, kutatók és bárki, akit a témakör érdekel.

A könyv számos gyakran előforduló probléma – például a megegyezés, a kommunikáció, az erőforrás-hozzárendelés és a szinkronizáció – megoldására tartalmaz algoritmusokat, különböző rendszereket feltételezve. Az algoritmusokat és a rájuk vonatkozó eredményeket elsősorban az osztott rendszerre vonatkozó alapvető feltételezések alapján csoportosítottuk. A csoportosítás alapja elsősorban az *időzítési modell* – eszerint szinkron, aszinkron és részben szinkron modelleket vizsgálunk. A csoportosítás második szempontja a *folyamatok közötti kommunikáció módja* – eszerint közös memóriát és üzenetküldést vizsgálunk. A rendszermodellek mindegyikével több fejezetben foglalkozunk; minden fejezetcsoport első fejezete

az adott csoportban alkalmazott formális modellt mutatja be, míg a további fejezetekben az algoritmusokat és megoldhatatlansági eredményeket tartalmazza. Bár a tárgyalásmód szigorú, mégis jelentősen támaszkodik az intuícióra.

Mivel ez a terület nagy és gyorsan fejlődik, a könyv meg sem próbál minden témával foglalkozni. A legfontosabb eredményeket választottuk. Ezek a bonyolultsági mértékek szempontjából nem mindig a legerősebb eredmények; általában olyan eredményeket tárgyalunk, amelyek egyszerűek és mégis bemutatják a tervezés és a gondolkodás általános módszereit.

Ez a könyv közel hozza az osztott számítások területének számos problémáját, algoritmusát és megoldhatatlansági eredményét. Képesek leszünk a problémáknak a gyakorlati környezetben való felismerésére, a könyvben tárgyalt algoritmusokhoz hasonló algoritmusok alkalmazására és arra, hogy a megoldhatatlansági eredményekre támaszkodva adott esetben amellett érveljünk, hogy a vizsgált probléma megoldhatatlan. A könyv segít a különböző rendszermodellek és lehetőségeik megértésében, ezért önállóan is tervezhetünk majd algoritmusokat, sőt új megoldhatatlansági eredményeket bizonyíthatunk. Végül a könyv talán meggyőz minket arról, hogy az osztott algoritmusok és rendszerek gondos tárgyalása megoldható: formálisan modellezhetők, megkívánt tulajdonságaik pontosan meghatározhatók, szigorúan bebizonyítható megadott követelményeknek való megfelelésük, közelítő bonyolultsági mértékekkel jellemezhetők és ezek szerint a mértékek szerint elemezhetők.

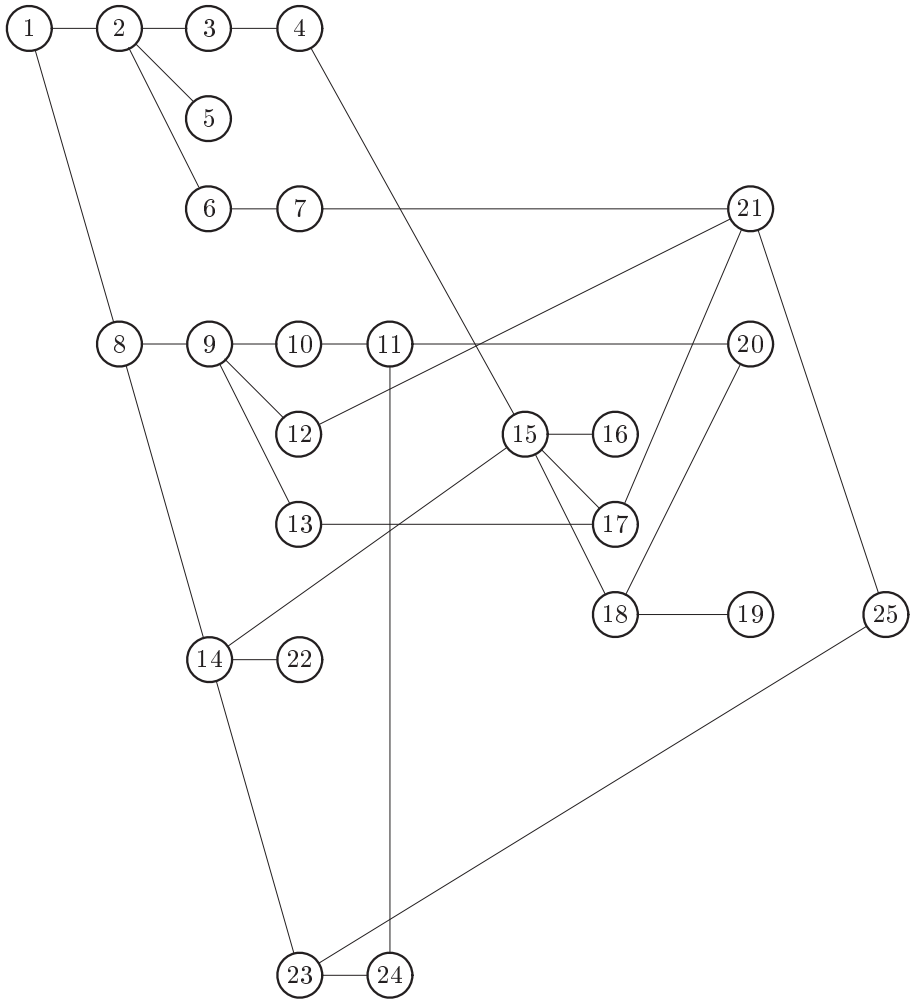
A könyv használata

Előismeretek.. A könyv megértéséhez csak a diszkrét matematika elemeire (beleértve az indukciós bizonyításokat és az aszimptotikus elemzést), bizonyos programozási készségre és a számítógépes rendszerekkel kapcsolatos tájékozottságra van szükség. A véletlenített algoritmusokra vonatkozó alfejezetekhez a valószínűségszámítás alapfogalmaira is szükség van. A soros algoritmusokkal és elemzésükkel foglalkozó, alsóbb évfolyamokon szokásos előadások hasznosak lehetnek, de nem szükségesek a megértéshez.

A fejezetek egymásra épülése.. A könyv felépítése olyan, hogy a különböző modelleket tárgyaló részek egymástól függetlenül is jól olvashatók. A fejezetek egymásra épülését az 1. ábra mutatja. Például ha gyorsan el akarunk jutni az aszinkron hálózatokról szóló anyaghoz, az 5–7. fejezeteket átugorhatjuk. A könyv jelentős részét elolvashatjuk és megérthetjük azok nélkül a modellezési fejezetek nélkül, amelyektől formailag függ.

Csillagos alfejezetek.. A könyv több olyan alfejezetet tartalmaz, melyek címe csillaggal végződik. Ezek az alfejezetek olyan anyagot tartalmaznak, amelyek a többi alfejezetnél nehezebbek és kevésbé alapvetők. Ezeket az alfejezeteket első olvasáskor különösebb kár nélkül átugorhatjuk.

Egyetemi tantárgyak.. A könyv korábbi változatait hosszú éveken át használtuk az MIT bevezető graduális előadásaihoz és három éven át rendszerszoftvert és alkalmazói szofvert gyártó cégek rendszertervezői számára tartott nyári



1.. ábra. A fejezetek tartalmának kapcsolata.

tanfolyamokon. A könyv elég anyagot tartalmaz egy egyéves tantárgyhoz, ezért könnyen kiválasztható belőle egy rövidebb kurzus anyaga (figyelembe véve a fejezetek összefüggéseit).

Például az aszinkron rendszerekkel foglalkozó féléves tantárgyhoz a 3., 4., 6., 7.2., 12. és 14–21. fejezeteket érdemes választani, hivatkozva a modellező fejezetekre (2., 8., 9.) és szükség szerint átvéve néhány definíciót a 10., 11. és 13. fejezetekből. Az osztott megegyezéssel foglalkozó féléves tantárgyhoz a 2–9., 12., 13.1., 15., 17., 21., 23. és 25. fejezeteket célszerű választani. Sok más lehetőség is van. Ha ezt a területet kutatjuk, kiegészíthetjük a könyv anyagát kedvenc területünk kutatási irodalmának friss és részletes eredményeivel.

Rendszertervezőknek tartott egy- vagy kéthetes tanfolyamon minden fejezettel foglakozhatunk, magas szinten elemezve az alapvető eredményeket és bizonyítási ötleteket, közben sok részletet elhagyva.

Hibák.. A könyvben talált hibákra vonatkozó észrevételeket és konstruktív javaslatokat köszönettel fogadom. Különösen hasznosak az új problémákra vonatkozó javaslatok. Kérem, hogy észrevételeiket a

`distalg@theory.lcs.mit.edu`
címre küldjék.

Köszönetnyilvánítás

Lehetetlen mindazoknak köszönetet mondani, akik ennek a könyvnek a megjelenéséhez hozzájárultak, mivel a könyv sokéves oktatási és kutatási munka eredménye. A hosszú munka során nagyon sok egyetemi hallgatóval és kutató kollegával voltam kapcsolatban. Mégis megpróbálom.

Ez a könyv az MIT 6852-es számú felsőéves tantárgyának végleges előadásjegyzete. A hallgatók sok éven át segítettek az anyag szervezésével kapcsolatos erőfeszítéseimben. Különösen sokat szenvedtek 1990-ben és 1992-ben, amikor elkészítették az előadások anyagának elektronikus változatát. Ken Goldman, Isaac Saias és Boaz Patt-Shamir gyakorlatvezetők a jegyzetek kidolgozását segítették. A tárgy oktatásának más szakaszaiban aktív oktató szerepet játszott Jennifer Welch és Rainer Gawlick.

Sok hallgató és kolléga járult hozzá ahhoz, hogy jobban megértsem a tárgyalt anyagot, miközben az ebben a könyvben megjelent eredményeken dolgoztunk vagy más könyvek anyagát elemeztük. Közéjük tartozik Yehuda Afek, Eshrat Arjomandi, Hagit Attiya, Baruch Awerbuch, Bard Bloom, Alan Borodin, James Burns, Soma Chaudri, Brian Coan, Haris Devarajan, Danny Dolev, Cynthia Dwork, Alan Fekete, Michael Fisher, Greg Frederickson, Eli Gafni, Rainer Gawlick, Ken Goldman, Art Harvey, Maurice Herlihy, Paul Jackson, Jon Kleinberg, Leslie Lamport, Butler Lampson, Victor Luchangco, Yishay Mansour, Michael Merritt, Michael Paterson, Boaz Patt-Shamir, Gary Peterson, Shlomit Pinter, Stephen Ponzio, Isaac Saias, Russel Schaffer, Roberto Segala, Nir Shavit, Liuba Shrira, Jorgen Søgaard-Andersen, Eugene Stark, Larry Stockmeyer, Mark Tuttle, Frits Vaardranger, George Varghese, Bill Weihl, Jennifer Welch és Lenore Zuck. Ketten közülük külön köszönetet érdemelnek: tanítómesterem, Michael Fisher, akivel 1978-ban kezdtünk dolgozni, amikor a témakör még kis, bár sokat ígérő kutatási téma volt, és tanítványom, Mark Tuttle, aki diplomamunkájában leírta és fejlesztette a b/k automata modellt.

Szeretném megköszönni Ajoy Datta, Roberto De Prisco, Alan Fekete, Faith Fich, Rainer Gawlick, Shai Halevi, Jon Kleinberg, Richard Ladner, John Leo, Victor Luchangco, Michael Melliar-Smith, Michael Merritt, Daniele Micciancio, Boaz Patt-Shamir, Anya Pogosyants, Stephen Ponzio, Sergio Rajsbaum, Roberto Segala, Nir Shavit, Mark Smith, Larry Stockmeyer, Mark Tuttle, George Varghese, Jennifer Welch és Lenore Zuck segítségét, akik a könyv kéziratának egy

részét elolvasták és sok hasznos észrevételt tettek. Ajoy, Faith és George különösen támogatta a könyv korai változatának használatára vonatkozó tanítási erőfeszítéseimet és hasznos javaslatokat tettek. Megköszönöm Joanne Talbotnak az anyag leírása, az ábrák rajzolása, az irodalomjegyzék elkészítése és a megszámlálhatatlan sok másolat készítése során végzett hatalmas munkáját. David Jones is segített az anyag végső kialakításában. Köszönöm John Guttagnak, Paul Penfieldnek és az MIT Villamosmérnöki és számítástudományi tanszéke más tagjainak, hogy biztosították számomra a könyv megírásához szükséges szabadidőt. Bruce Spatz, a Morgan Kaufman Kiadó munkatársa bátorított és segített ebben a nagy vállalkozásban és úgy tűnt, mindig a legmegfelelőbb javaslatokat tette. Julie Pabst és Diane Cerra, a Morgan Kaufman Könyvkiadó részéről intenzíven segítettek a könyv előkészítésének végső fázisában. Ugyancsak köszönetet mondok a Babel Press munkatársának, Ed Sznyternek LaTeX-szakértelméért.

Végezetül és legfőképpen hálás vagyok végtelenül türelmes családomnak, Dennis, Patrick és Mary Lynchnek, amiért tényleg minden gondot levettek a vállamról a könyv írása idején. Külön köszönet Dennisnek a kitűnő halételekért (nem is említve a fürdőszoba és a mosókonyha felújítását), mialatt én minden időmet a számítógép mellett töltöttem.

Nancy Lynch

Cambridge, Massachusetts

1. fejezet

Bevezetés

1.1.. Miről szól a könyv?

Az **osztott algoritmusok** kifejezés sokféle, széles körben alkalmazott párhuzamos algoritmust takar. Eredetileg olyan algoritmusok azonosítására használták, amelyeket nagy földrajzi területen **szétosztott** processzorokon való futásra terveztek. Az évek során azonban a kifejezés jelentése bővült, így ma azokat az algoritmusokat is tartalmazza, amelyek helyi hálózaton, sőt közös memóriájú többprocesszoros rendszereken futnak. Ez azért történt így, mert felismerték, hogy ezeknek a – különböző körülmények között futó – algoritmusoknak sok a közös vonása.

Az osztott algoritmusokat számos célra használják, például a telekommunikáció, az osztott információfeldolgozás, a tudományos számítások és a valósidejű folyamatirányítás területén. Ezen alkalmazások bármelyikénél a rendszerek létrehozása során a munka fontos része az osztott algoritmusok tervezése, megvalósítása és elemzése. Könyvünk célja az ilyen alkalmazások során felmerülő problémák és a megoldásukra szolgáló algoritmusok összefoglaló tárgyalása.

Az osztott algoritmusoknak számos különböző típusa van. Felsorolunk néhány tulajdonságot, amelyben különbözhetnek.

- *A folyamatok kommunikációs modellje* (IPC, interprocess communication). Az osztott algoritmusok egyszerre több processzoron futnak, melyeknek valahogy kommunikálniuk kell. A kommunikációs módszerek tartalmazzák a közös memóriához való hozzáférést, az üzenetek processzortól processzorig való vagy szóró továbbítását (mind helyi hálózatban, mind nagy távolságra) és a távoli eljáráshívások végrehajtását.

- *Időzítési modell.* A rendszerben bekövetkező események időzítésére nézve többféle feltevés lehetséges, amelyek az algoritmus által felhasználható időzítési eljárásra vonatkoznak. Szélsőséges esetben elképzelhető, hogy a processzorok teljes összhangban dolgoznak, az üzenetküldést és a számításokat is így végzik. A másik szélsőséges eset az, amikor a lépéseket a processzorok egymástól teljesen függetlenül (aszinkron módon), különböző sebességgel és tetszőleges sorrendben hajtják végre. A szélsőségek között van a lehet-

séges feltevéseknek az a nagy tartománya, amit *részben szinkron* esetnek nevezünk: ilyenkor a processzorok részleges információval rendelkeznek az események időzítéséről. Például a processzorok egymáshoz viszonyított sebességére nézve korlátokat ismerünk vagy a processzorok közelítőleg összehangolt (szinkronizált) órákhoz férhetnek hozzá.

- *Hibamodell.* Feltételezhetjük, hogy a hardver, amin az algoritmus fut, megbízható – de azt is, hogy az algoritmusnak a hibás viselkedést bizonyos mértékben tűrnie kell. A hibás viselkedés jelenthet processzorhibákat: a processzorok leállhatnak (figyelmeztetés nélkül vagy figyelmeztetés után); lehetnek átmenetileg hibásak vagy elkövethetnek ún. *bizánci hibát*, ami azt jelenti, hogy a hibás processzor tetszőlegesen viselkedhet. A hibás viselkedés lehet kapcsolati (kommunikációs) hiba is, például az üzenetek elvesztése vagy többszöri elküldése.
- *A megoldandó problémák köre.* Természetesen az algoritmusok abban is különbözhetnek, hogy milyen problémák megoldására szolgálnak. Azokat a jellemző problémákat vizsgáljuk, amelyek a korábban említett alkalmazások során fordulnak elő. Ezek közé tartoznak az erőforrások hozzárendelésével, a kommunikációval, az osztott processzorok közötti megegyezéssel, az adatbázisok párhuzamos vezérlésével, a holtpont felismerésével, a globális fényképekkel, a szinkronizációval és a különböző típusú objektumok kezelésével kapcsolatos problémák.

Bizonyos párhuzamos algoritmusokat, mint például a párhuzamos közvetlen hozzáférésű gépen (PRAM, Parallel Random Access Machine) és a rögzített-kapcsolású hálózatokon (tömb, fa, hiperkocka) futó algoritmusokat nem tárgyalunk ebben a könyvben. Az itt bemutatott algoritmusokat a párhuzamos algoritmusok bővebb osztályán belül a nagyfokú *bizonytalanság* és a *tevékenységek függetlensége* jellemzi. A könyvben tárgyalt algoritmusoknál a bizonytalanságnak és függetlenségnek például a következő formái fordulhatnak elő:

- ismeretlen számú processzor;
- ismeretlen hálózati topológia;
- különböző helyekre egymástól független bemenő adatok töltődnek;
- egyszerre több program hajtódik végre, amelyek különböző időpontban indultak el és különböző sebességgel hajtódnak végre;
- a processzorok nem determinisztikusak;
- bizonytalan az üzenetek szállítási ideje;
- ismeretlen az üzenetek sorrendje;
- a processzorok és adatátviteli vonalak meghibásodhatnak.

Szerencsére nem minden algoritmusnak kell mindezekkel a bizonytalanságokkal egyszerre megküzdenie!

Ezen bizonytalanságok miatt gyakran nagyon nehéz az osztott algoritmusok működését megérteni. Még akkor is, ha egy algoritmus kódja rövid, az a tény, hogy a kódot sok processzor – egymást meghatározatlan módon átfedő lépésekben, párhuzamosan – hajtja végre, azt vonja maga után, hogy ezek az algoritmusok ugyanazon bemenő adatok esetén is sokféleképpen viselkedhetnek. Ezért általában lehetetlen az algoritmusokat úgy megérteni, hogy megjósoljuk, pontosan

hogyan fognak végrehajtódni. Ez a viselkedés szembeállítható az olyan párhuzamos algoritmusokkal, mint a PRAM algoritmusok, melyeknél általában pontosan tudjuk, mit fog az algoritmus tenni egyes időpontokban. Az osztott algoritmusoknál a legjobb, ha a működés teljes megértése helyett arra törekszünk, hogy az algoritmus viselkedésének bizonyos kiválasztott *tulajdonságait* értsük meg.

Az osztott algoritmusokra vonatkozó ismeretanyag az elmúlt 20 év alatt meglehetősen rendszerezett területté fejlődött. Az ezen a területen folyó munka többékevésbé általános jellemzői a következők voltak. Először a gyakorlatban előforduló osztott számítások jellemző problémáit azonosították és leírták ezek matematikai tanulmányozásra alkalmas változatait. Ezután algoritmusokat dolgoztak ki ezeknek a problémáknak a megoldására. Az algoritmusokat pontosan leírták, majd bebizonyították, hogy a felvetett problémát megoldják. Bonyolultságukat különböző mértékek szerint elemezték. Az ilyen algoritmusok tervezői rendszerint a bonyolultság csökkentésére törekednek. Bebizonyították bizonyos problémák megoldhatatlanságát és jelentős erőforrásigényét, jellemezve ezzel a megoldható problémák körére és a megoldási költségekre vonatkozó korlátokat. Mindennek a munkának az alapját a matematikai modellek alkották.

Ezek az eredmények együtt egy nagyon érdekes matematikai elméletet alkotnak, sőt annál is többet. A problémák megfogalmazása valódi rendszerek részeinek pontos leírására is alkalmazható, az algoritmusok (sok esetben) a gyakorlatban is felhasználhatók, a megoldhatatlanságra vonatkozó eredmények pedig azzal segítik a tervezőket, hogy megmondják, mikor kell felhagyni az adott rendszer felépítésére vonatkozó erőfeszítésekkel. Mindezek az eredmények, valamint a velük kapcsolatos matematikai modellek hozzásegíthetik a tervezőket ahhoz, hogy megértsék azokat a rendszereket, amelyeket terveznek.

1.2.. Nézőpontunk

Ez a könyv osztott algoritmusokkal foglalkozik. Mivel a terület nagy és gyorsan fejlődik, nem tudunk minden témával foglalkozni. Mivel válogatnunk kellett, a témakörnek az elmélet és a gyakorlat szempontjából legfontosabb eredményeit foglaltuk össze. Ezek a bonyolultsági jellemzőket tekintve nem mindig optimális eredmények; inkább azokat részesítettük előnyben, amelyek tükrözik a tervezés és gondolkodás általános módszereit. Az általunk bemutatott eredmények néhány olyan problémához kapcsolódnak, amelyek ezen a területen jellemzőek: ilyenek a vezető folyamat választása, keresés a hálózatban, feszítőfa építése, osztott megegyezés, kölcsönös kizárás, erőforrások hozzárendelése, objektumok létrehozása, szinkronizáció, globális fényképek és megbízható kommunikáció. Ezek a problémák sok különböző alkalmazásban fellépnek, ezért több különböző rendszermodellben is megvizsgáljuk őket.

A könyv egyik tulajdonsága, hogy az összes algoritmust, megoldhatatlansági eredményt és alsó korlátot többé-kevésbé egységesített formális keretben mutatjuk be. Ez a keret a különböző típusú osztott rendszerek automataelméleti modelljeiből és a rendszerek ilyen modellek segítségével történő elemzésére vonatkozó szabványos módszerekből áll. Inkább automataelméleten, mint valamelyik

speciális formális nyelven vagy bizonyítási módszeren alapul, ami lehetővé teszi, hogy az eredményeket alapjában véve halmazelméleti terminológiával adjuk meg, és hogy a nyelvi részletek ne okozzanak gondot. Ugyancsak lehetővé teszi azt a rugalmasságot, hogy ugyanabban a keretben különböző nyelveket és logikai megközelítéseket alkalmazzunk az algoritmusok leírására és elemzésére. A formális modellek alkalmazásának köszönhető az összes eredmény szigorú kezelése.

Nézzük meg, mit értünk szigorúságon, amely az osztott algoritmusok területén különösen fontos, mivel sok bonyodalom léphet fel. Pontos fogalmak nélkül nehéz elkerülni a hibákat. Az azonban nem világos, hogyan tudjuk az anyagot pontosan tárgyalni úgy, hogy az elég rövid és könnyen érthető legyen. Ebben a könyvben kompromisszumot kötünk: az intuíció és a szigorú következtetések keverékét alkalmazzuk. Nevezetesen a megfelelő formális modellek pontos leírását adjuk. Az algoritmusokat helyenként a formális modellek segítségével határozzuk meg, másutt szavakkal írjuk le azokat, és esetenként mindkét módszert alkalmazzuk. Az algoritmusok helyességének bizonyításában a szigorúság foka erősen változik: helyenként formális bizonyítást adunk, másutt csak vázlatot. Azt reméljük, hogy elegendő eszközt mutatunk be ahhoz, hogy az olvasók az intuitív vázlatokat szükség esetén formális bizonyítássá alakítsák. A megoldhatatlansági eredményeket általában meglehetősen szigorúan bizonyítjuk a formális modellek segítségével.

Mivel sok különböző feltételt és problémát kell tárgyalnunk, nem egyértelmű, hogyan célszerű az anyagot bemutatni. Azt választottuk, hogy mondanivalónkat elsősorban a formális modellek szerint csoportosítjuk – méghozzá elsősorban a modellek azon tulajdonságai szerint, amelyek a legnagyobb befolyással vannak az eredményekre, és másodsorban a problémák elvont megfogalmazása szerint. A modellek közötti legmélyebb különbség az időzítési feltételekben van, de a kommunikáció módja és a hibákra vonatkozó feltételek szintén fontos tényezők.

A következő időzítési modelleket alkalmazzuk.

- *Szinkron modell.* Ez a legegyszerűbben leírható, programozható és elemezhető modell. Feltesszük, hogy a modell elemei egyszerre végzik a műveleteket, és a végrehajtás szinkronizált menetekben történik. Természetesen az osztott rendszerekben ez nem mindig így történik, de a szinkron modell mindenképpen hasznos. Annak megértése, hogyan oldunk meg egy feladatot a szinkron modellben, gyakran hasznos lépés annak megértése felé, hogyan oldhatjuk meg ugyanazt a feladatot reálisabb modellekben. Például esetenként lehetőség van arra, hogy egy valódi osztott rendszert szinkron rendszerrel szimuláljunk. Továbbá a szinkron rendszerre kapott megoldhatatlansági eredmények közvetlenül átvihetők a kevésbé meghatározott modellekre. Másrészt az osztott rendszerek számos típusára igaz, hogy azokat nem lehet vagy csak nagyon kis hatékonysággal lehet szinkron rendszerekkel megvalósítani.
- *Aszinkron modell.* Itt azt tételezzük fel, hogy a különböző összetevők a lépéseket tetszőleges sorrendben és tetszőleges sebességgel végzik el. Ezt a modellt szintén viszonylag egyszerűen le lehet írni, vannak azonban olyan részletek, amelyek *élénkségi* meggondolásokat tartalmazhatnak. Ezt a modellt nehezebb programozni, mint a szinkron modellt, mivel az események

sorrendje bizonytalan. Az aszinkron modell azonban lehetővé teszi, hogy a programozó figyelmen kívül hagyja az időzítési megfontolásokat. Mivel az aszinkron modell kevesebbet tesz fel az időzítésről, mint ami más osztott rendszerekre jellemző, az aszinkron rendszerekhez tervezett algoritmusok általánosak és átvihetők más rendszerekbe: biztos, hogy bármilyen időzítési mód használata esetén helyesen fognak működni. Másrészt az aszinkron modell időnként nem alkalmas arra, hogy a problémát hatékonyan oldjuk meg, sőt esetenként arra sem alkalmas, hogy a problémát egyáltalán meg tudjuk oldani.

- *részben (az időzítést illetően) szinkron modell.* Itt az események bekövetkezésének relatív időpontjára nézve teszünk bizonyos megszorításokat, de a végrehajtás nem teljesen azonos ütemre történik (azaz nem pontosan úgy, mint a szinkron modellben). Ezek a modellek közelebb állnak a valósághoz, de ezeket a legnehezebb programozni. Az időzítési információ ismeretére támaszkodó algoritmusok nagyon hatékonyak lehetnek, ugyanakkor nagyon érzékenyek az időzítési feltételek megsértésére.

A következő szempont, amit az osztályozásra felhasználunk, a folyamatok közötti kommunikáció (IPC) módja. Ebben a könyvben a közös memóriát és az üzenetküldést is tárgyaljuk. Először a közös memóriájú modellt mutatjuk be, mivel az erősebb és könnyebb megérteni, és mivel számos, a közös memóriára vonatkozó módszer és eredmény alkalmazható a hálózatokra is. Ezután az anyagot a tárgyalt problémáknak megfelelően szervezzük. Végül sok problémát tárgyalunk különböző, a hibákra vonatkozó feltevések mellett. Látni fogjuk, hogy ha a problémákat különböző modellekben fogalmazzuk meg, előfordul, hogy a feltételekben látszólag kicsi a különbség, az eredményekben azonban nagy. Igyekeztünk ezeket a különbségeket azonosítani és értelmezni.

Arra törekedtünk, hogy a tárgyalás annyira *moduláris* legyen, amennyire csak lehet: egyes algoritmusokat más algoritmusokból állítottunk össze, másokhoz általánosítással vagy egy modellből egy másikba való átalakítással jutottunk. Ez sokat segített az anyag egyszerűsítésében és lehetővé tette, hogy kevesebb munkával többet végezzünk. A modularitás a gyakorlatban előforduló osztott rendszerekben hasonló célt szolgálhat.

1.3.. A 2–25. fejezetek tartalmának áttekintése

A könyv a következő területekkel foglalkozik.

Modellek és bizonyítási módszerek.. A formális modelleket és a bizonyítási módszereket a szinkron hálózati algoritmusokról, aszinkron közös memóriájú algoritmusokról, aszinkron hálózati algoritmusokról és a részben szinkron algoritmusokról szóló fejezetek (2., 8., 9., 14. és 23.) első alfejezetében ismertetjük. Az anyagot a könnyebb hivatkozás érdekében tagoltuk több fejezetre. A könyv első olvasásakor bizonyos modellező részeket át lehet ugrani és a későbbi – algoritmusokkal foglalkozó – fejezetek olvasásakor vissza lehet térni az átugrott anyaghoz. Igyekeztünk a könyvet úgy megírni, hogy az algoritmusok anélkül legyenek olvashatók és nagyrészt érthetők, hogy túl sok energiát igényelne a formális modellek

megértése.

Minden alkalmazott modell olyan gépeken alapul, amelyeknek állapotai vannak (ezek száma gyakran végtelen) és az állapotátmeneteknek rendszerint neve van. Egy ilyen gép ezért mind a rendszer valamely elemének, mind pedig az egész osztott rendszernek a modellezésére alkalmas. A gép minden állapota az elem vagy az egész rendszer pillanatnyi állapotát tükrözi, beleértve az olyan információkat is, mint az egyes processzorok memóriájának tartalma, a futó programok utasításszámlálójának tartalma és a kommunikációs rendszerben lévő átmenő üzenetek. Az átmenetek az olyan, a rendszerben végbemenő változásokat írják le, mint az üzenetek küldése vagy fogadása, vagy a helyi számítások okozta változások. Különböző gépeket javaslunk a szinkron hálózatokra, aszinkron rendszerekre és az időzítésalapú rendszerekre.

Az osztott rendszerek formális modelljeinek egyik fontos alkalmazása a megoldandó feladatok pontos megfogalmazása és a megoldási algoritmusok helyességének bizonyítása. Az ilyen megfogalmazási és helyességbizonyítási feladatokat általános vagy egyedi módszerrel lehet megoldani, bizonyos módszerek azonban olyan gyakran előfordulnak, hogy a modellező fejezetekben pontosan leírjuk őket. Ezek a módszerek magukban foglalják az *invariáns állítások* és a *szimuláció* módszerét. Az invariáns állítás a rendszer olyan tulajdonsága, amely minden elérhető állapotra jellemző. Az invariáns állításokat rendszerint a rendszer végrehajtási sorozatainak lépéseire vonatkozó indukcióval bizonyítjuk. A szimuláció két rendszer közötti formális kapcsolat, ahol az egyik rendszer a megoldandó problémának, a másik pedig a megoldásnak felel meg, vagy pedig az egyik egy magas szintű, elvont megoldásnak, a másik pedig egy részletes megoldásnak. A szimulációs kapcsolatokat általában ugyancsak indukcióval bizonyítjuk be.

A 2. fejezet tartalmazza az első modellt, a szinkron hálózatokét. Ez nagyon egyszerű modell, amely az üzenetek küldésének és a számítások elvégzésének szinkronizált módját írja le. A 8. fejezet az aszinkron rendszerek általános modelljét, a beviteli/kiviteli automata (b/k automata) modelljét tartalmazza. A modell neve arra utal, hogy világosan megkülönbözteti a beviteli és kiviteli átmeneteket, azaz a rendszer által a környezetnek és a környezet által a rendszernek küldött üzeneteket. Egy b/k automatánál bármely adott állapotban több átmenet lehetséges; például a különböző processzorokkal kapcsolatos átmenetek tetszőleges sorrendben hajtódhatnak végre. Mivel a modell az átmenetek sorrendjét illetően nagy rugalmasságot tesz lehetővé, tartalmazza az élénkség fogalmát, ami lehetővé teszi, hogy kifejezzük azt, hogy valamilyen átmenet biztosan megvalósul. Ennek a modellnek hasznos tulajdonsága, hogy tartalmaz párhuzamos összeállítási műveletet, ami lehetővé teszi, hogy egy – b/k automataként modellezett elemekből összeállított – rendszert is b/k automataként modellezzünk. Egy összetett automata helyességének bizonyítása gyakran moduláris módon történik, azaz elemei helyességének bizonyításán alapul.

A 8. fejezetben tárgyalt modell elég általános ahhoz, hogy az aszinkron közös memóriájú rendszereket és az aszinkron hálózatokat is leírja (valamint az aszinkron rendszerek számos más típusát is); a 9. és 14. fejezetek pedig azt a kiegészítő szerkezetet tartalmazzák, ami ahhoz szükséges, hogy a modell a közös memóriájú és az üzenettovábbító rendszerek leírására is alkalmas legyen.

Végül, a 23. fejezetben az időzítés alapú rendszerek modelljeit mutatjuk be. Ezek a modellek állapotgépek, melyekben az állapotok az időzítésre vonatkozó adatokat is tartalmazzák, például az aktuális időt vagy a különböző események ütemezett időpontját. Ezek a modellek lehetővé teszik, hogy leírjuk az időzítés alapú rendszerek jellemző szerkezeteit, így a helyi órákat és időkorlátokat.

Szinkron hálózatok algoritmusai.. A vizsgált modellek közül a legegyszerűbb (és a legkevésbé bizonytalan) a szinkron hálózati modell, amelyben a processzorok szinkronizált lépésekben érintkeznek egymással, és ugyanígy végzik a számításokat. Nem foglalkozunk a szinkron közös memóriájú algoritmusokkal, mivel azok önálló tanulmányozást igénylő, nagy területet alkotnak (lásd a fejezet végén lévő megjegyzéseket). A hálózati modellekben feltesszük, hogy a processzorok egy G irányítatlan vagy irányított gráf csúcsaiban helyezkednek el, és a szomszédaikkal a G gráf élei mentén küldött üzenetekkel érintkeznek.

A 3–7. fejezetekben több, a szinkron hálózatokban fellépő, jellegzetes osztott problémát tárgyalunk. A 3. fejezetben egy egyszerű problémával kezdünk, amely a gyűrűs hálózatokban való számításokkal kapcsolatos. A probléma *egyetlen vezető folyamat választása* egy gyűrű szerkezetű hálózatban, feltételezve, hogy a csúcsok az *egyedi azonosítóktól* (UID) eltekintve azonosak. A fő bizonytalanság itt az, hogy a processzorok egyedi azonosítóinak halmaza nem ismert (csak az, hogy az azonosítók különböznek egymástól); továbbá rendszerint a hálózat mérete is ismeretlen. Ennek a problémának a legfontosabb előfordulása a helyi gyűrűs hálózat, amelyben egyetlen jel halad körbe, és mindig csak a jel tulajdonosa jogosult a kommunikáció kezdeményezésére. Időnként azonban a jel elvész, és ilyenkor szükség van egy olyan algoritmusra, amely újra előállítja a hiányzó jelet. Ez a helyreállítási eljárás a vezető választása. Bemutatunk néhány, a vezető választásával kapcsolatos bonyolultságelméleti eredményt, valamint a kommunikációhoz szükséges időre és üzenetek számára vonatkozó korlátokat bizonyítunk be.

Ezután a 4. fejezetben rövid áttekintést adunk az általánosabb hálózatokban használt alapvető algoritmusokról. Például algoritmusokat ismertetünk a *vezető-választás*, a *szélességi keresés*, a *legrövidebb utak*, *minimális feszítőfa*, *maximális független halmaz* meghatározására. A bizonytalanság jellemző formái itt az ismeretlen UID-ek és az ismeretlen hálózati gráfok.

Ezután az 5. és 6. fejezetekben a *megegyezés* elérésének problémáit vizsgáljuk osztott hálózatokban. Ezek olyan problémák, melyekben osztott processzoroknak közös véleményre kell jutniuk, annak ellenére, hogy kezdetben különböző a véleményük a helyes döntésről. A gyakorlatban számos különböző megegyezési probléma lép fel: például egy repülőgép fedélzetén lévő processzorok különböző magasságmérőt figyelnek, és megpróbálnak megegyezni egy közös magasságban. Egy másik példa, amikor a processzorok különböző hibavizsgáló eljárásokat hajtanak végre a rendszer egy másik összetevőjével kapcsolatban, és megpróbálják egyéni diagnózisukat arra vonatkozó közös döntéssé egyesíteni, hogy cserélni kell-e az adott összetevőt.

Az itt előforduló bizonytalanság nem csak a kezdeti vélemények különbözőségéből adódik, hanem a *hibákból* is, amelyek az adatátviteli vonalakban vagy a processzorokban előfordulhatnak. Az 5. fejezetben azt az esetet vizsgáljuk, ahol

az adatátviteli vonalakon üzenetek veszhetnek el. A 6. fejezetben két különböző típusú processzorhibát vizsgálunk: a megállási hibát, amikor egy processzor egy adott időpontban befejezheti saját helyi protokolljának végrehajtását, és a bizánci hibát, amikor a hibás processzor tetszőleges viselkedést mutathat (figyelembe véve azt a korlátozást, hogy nem teheti tönkre a rendszernek azt a részét, amelyhez nincs hozzáférése). Korlátokat adunk meg a tűrhető hibák számára, a kommunikáció idejére és a szükséges üzenetek számára nézve.

Végül a 7. fejezetben az alapvető megegyezési probléma bizonyos kiterjesztéseit és változatait vizsgáljuk, mint a *megegyezés értékek egy kis halmazára nézve* (egyetlen érték helyett), *közelítő megegyezés* egy valós értékre nézve és *véglegesítés osztott adatbázisban*.

Aszinkron közös memóriájú algoritmusok.. Miután bemelegítettünk a szinkron algoritmusokkal (amelyekben kevés a bizonytalanság), elkezdjük a jóval bonyolultabb aszinkron algoritmusok tanulmányozását. Itt már nem azt tesszük fel, hogy a processzorok egy ütemre működnek, hanem inkább azt, hogy lépéseik tetszőleges módon átfedhetik egymást, és nincs korlát az egyes processzorok sebességére nézve. Itt jellemzően inkább a külvilággal kapcsolatos információcsere zajlik (bemenő és kimenő adatok segítségével), például kezdeti adatbevitel és záró adatkivitel. Ilyen feltételek mellett az eredmények a szinkron modellekhez képest lényegesen eltérő jelleget mutatnak.

A 10–13. fejezetek aszinkron közös memóriájú algoritmusokat tartalmaznak. Az első probléma, amit a 10. fejezetben vizsgálunk, a *kölcsönös kizárás*. Ez az osztott algoritmusok világának legfontosabb problémája, és az első olyan, amely komoly elméleti vizsgálatok tárgyát képezte. A probléma lényege az, hogyan kezeljünk egy olyan oszthatatlan erőforrást, amelyhez egyidejűleg legfeljebb egy felhasználó férhet hozzá. A probléma úgy is megfogalmazható, hogy a programkódok bizonyos részei *kritikus szakaszok*, és nincs megengedve az, hogy két program egyszerre legyen a saját kritikus szakaszában. Ez a probléma osztott és nem osztott operációs rendszerekben is fellép. A lépések sorrendjére vonatkozó, alapvető bizonytalanság mellett itt az is bizonytalan, hogy mely felhasználók és mikor igénylik az adott erőforráshoz való hozzáférést.

Bemutatunk több olyan algoritmust, amely megoldja a kölcsönös kizárást közös memória esetén. Ezt a Dijkstra által 1965-ben javasolt algoritmussal kezdjük, és egyre jobb helyességi garanciákkal rendelkező algoritmusokkal folytatjuk. Az eredmények többsége olyan közös memórián alapul, amely csak író és olvasó műveletekkel érhető el; ezekre az írási-olvasási műveleteket tartalmazó közös memória-modellekre vonatkozóan a szükséges közös változók számára vonatkozó alsó korlátot is adunk. A közös memória erősebb változatára – az olvasható/módosítható/írható memóriára – vonatkozó problémát is vizsgáljuk: ebben az esetben alsó és felső korlátot adunk a szükséges közös memória méretére. Az algoritmusok és alsó korlátok bemutatása mellett a kölcsönös kizárási problémát arra is felhasználjuk, hogy az aszinkron osztott algoritmusok számos általános jelentőségű fogalmát bemutassuk. Ezek olyan általános modellezési fogalmak, mint az atomiság, pártatlanság, haladás és hibatűrés, invariáns állítások és szimulációs bizonyítások, valamint az időelemzési módszerek.

A 11. fejezetben a kölcsönös kizárási problémának a bonyolultabb *erőforrás-*

hozzárendelési problémákká való általánosítását vizsgáljuk; ezek a problémák többféle erőforrást és a felhasználásukra vonatkozó bonyolult követelményeket tartalmaznak. Például elemezzük az *étkező filozófusok* problémáját, amely egy olyan jellemző erőforrás-hozzárendelési probléma, melyben egy processzorgyűrűben páronként kölcsönösen kizárt erőforrásokat kell elosztani.

A 12. fejezetben újra a *megegyezés* problémáját vizsgáljuk, ezúttal az aszinkron közös memóriájú modellben. Ebben a fejezetben a fő eredmény az az alapvető tény, hogy hibák feltételezése mellett a kiemelt fontosságú megegyezési probléma nem oldható meg, ha a közös memória csak olvasási és írási műveleteket támogat. Ezzel ellentétben a közös memória erősebb típusainál – amilyen például az olvasható/módosítható/írható memória – egyszerű megoldást javaslunk.

A 13. fejezetben *atomi objektumokkal* foglalkozunk. A könyvben eddig a pontig feltételezzük, hogy a processzoroknak a közös memóriához való hozzáférése azonnali. Az atomi objektumok egyedi kérési és válaszoló reakciókat mutatnak, de egyébként nagyon hasonlítanak az azonnali hozzáférésű közös változókhoz. Meghatározzuk az atomi objektumokat, és olyan alapvető eredményeket bizonyítunk, melyek megmutatják, hogyan alkalmazhatók az atomi objektumok rendszerek felépítésére; például, hogyan alkalmazhatók a közös változók helyett. Több olyan algoritmust is vizsgálunk, amelyek hatásos atomi objektumokat hoznak létre gyengébb atomi utasítások – gyengébb típusú atomi objektumok vagy közös változók – segítségével. Ezeknek az algoritmusoknak egy érdekes tulajdonsága a *várakozási szabadság*, ami azt jelenti, hogy az objektumon végzett bármely műveletnek be kell fejeződnie – függetlenül a párhuzamosan végzett művelet hibájától.

Megmutatjuk, hogyan lehet létrehozni egy *pillanatnyi atomi objektumot* az olvasható/írható közös memória segítségével. A pillanatnyi atomi objektum *pillanatnyi műveletekre* képes, amelyek minden memóriahelyre egyszerre írnak be értéket. Azt is megmutatjuk, hogyan lehet egyszerű-olvasó/egyszerű-író közös memória segítségével létrehozni *többszörös-olvasó/többszörös-író atomi objektumot.*

Aszinkron hálózati algoritmusok.. A 15–22. fejezetekben azokat az algoritmusokat tárgyaljuk, amelyek az aszinkron hálózatokban működnek. A rendszert a szinkron hálózatokhoz hasonlóan gráfokkal vagy irányított gráfokkal modellezzük, amelyekben a processzorok a csúcsok és az adatátviteli vonalak az élek, de most a rendszer nem szinkronizált lépésekben működik. Most az üzenetek tetszőleges időpontban érkezhetnek, és a processzorok lépéseiket tetszőleges sebességgel végezhetik el. Azt mondhatjuk, hogy a rendszer összetevői kevésbé függnek egymástól, mint akár a szinkron, akár az aszinkron közös memóriájú rendszerekben. Tehát a modellben a bizonytalanság mértéke tovább nő.

A 15. fejezet azzal kezdődik, hogy újra megvizsgáljuk az aszinkron hálózatok problémáit és algoritmusait. Például elemezzük a vezetőválasztás, szélességi keresés, legrövidebb utak, üzenetszórás és konvergens üzenetszórás, minimális feszítőfa problémáit. Bár az algoritmusok egy része kis változtatásokkal alkalmas az új problémák megoldására, legtöbbjük lényeges módosítást igényel. Például meglehetősen nehéz a 4. fejezet egyszerű minimális feszítőfa algoritmusát kiterjeszteni az aszinkron probléma megoldására.

A 15. fejezet meggyőzheti az olvasót arról, hogy az aszinkron hálózatok pro-

gramozása bonyolult. Ez a bonyolultság motiválja a következő négy fejezetet (16–19. fejezetek), ahol a feladat egyszerűsítése érdekében négy módszert mutatunk be. Ezeket a módszereket mint *algoritmus-átalakításokat* írjuk le, amelyek lehetővé teszik, hogy az egyszerűbb vagy hatásosabb modellekre tervezett algoritmusokat a bonyolultabb aszinkron hálózati modellen futtassuk.

Az első módszer, amelyet a 16. fejezetben írunk le, a *szinkronizátor* bevezetése. A szinkronizátor a rendszerek olyan eleme, amely lehetővé teszi, hogy a hiba nélküli aszinkron hálózat utánozza a 2–4. fejezetekben leírt (ugyancsak hiba nélküli) szinkron hálózatokat. Hatékony megvalósításokat mutatunk be, és szembeállítjuk ezeket olyan alsó korlátokkal, amelyek azt sugallják, hogy bármely ilyen szimulációt szükségképpen alacsony hatékonyság jellemez.

A 17. fejezetben leírt második módszer az aszinkron közös memóriájú modell utánzása aszinkron hálózati modell segítségével. Ez a módszer lehetővé teszi, hogy a 10–13. fejezetben leírt aszinkron közös memóriájú algoritmusokat alkalmazzuk az aszinkron hálózatokban.

A 18. fejezetben leírt harmadik módszer az ellentmondásmentes *logikai idők* hozzárendelése az aszinkron osztott hálózatokban bekövetkező eseményekhez. Ez a módszer lehetővé teszi, hogy egy aszinkron hálózat egy olyan hálózatot utánozzon, amelyben a csúcsok tökéletesen szinkronizált valósidejű órákhoz férnek hozzá. Ennek a lehetőségnek egy fontos alkalmazása, hogy az aszinkron hálózattal utánozható egy központosított (nem osztott) állapotgép.

A 19. fejezet tartalmazza a negyedik módszert, az aszinkron hálózati algoritmusok futásidejű megfigyelését. Ez használható például tesztelés, visszatöltési változat előállítása vagy az algoritmus *stabil tulajdonságainak* megismerése céljából. A stabil tulajdonság olyan, hogy ha fellép, akkor örökre megmarad; ilyen például a *befejeződés* vagy a *holtpont*. Kiderül, hogy az alapvető atomi utasítás, amely segít a stabil tulajdonságok felismerésében, az *ellentmondásmentes globális fénykép* elkészítése az osztott algoritmus állapotáról. Több módot is bemutatunk ilyen fénykép készítésére, és leírjuk, hogyan alkalmazhatók ezek a stabil tulajdonságok felismerésére. Miután ismertettünk bizonyos hatékony módszereket, visszatérünk az aszinkron rendszerekben történő számításokhoz. A 20. fejezetben újra megvizsgáljuk az erőforrás-hozzárendelés problémáját. Több módszert mutatunk be az *étkező filozófusok* és a *kölcsönös kizárás* problémájának a szinkron hálózatokban való megoldására.

A 21. fejezetben az aszinkron hálózatokban hibák jelenléte esetén való számítások problémáját vizsgáljuk. Először a 17. fejezetben bemutatott átalakítást alkalmazva megmutatjuk, hogy a közös memóriájú megoldhatatlansági eredmény átvihető a hálózatokra. Ezután ennek az elkerülhetetlen korlátozásnak a környezetében vizsgálunk meg néhány témát: bemutatunk egy *véletlenített algoritmust* és egy *hibafelismerőket* alkalmazó módszert a megegyezés megoldására, és megmutatjuk, hogyan érhető el *közelítő megegyezés* a pontos megegyezés helyett.

A 22. fejezetben az *adatátviteli vonalak* problémáit vizsgáljuk. Az adatátviteli protokollok feladata, hogy megbízható kapcsolati vonalakat hozzanak létre a megbízhatatlan adatátviteli vonalakból. A BitVáltó protokollal kezdjük, amely önmagában is érdekes és mintapélda a párhuzamos algoritmusok helyességének bizonyítására. Ezzel a problémával kapcsolatban további algoritmusokat és meg-

oldhatatlansági eredményeket mutatunk be különböző – az adatátviteli csatornák meghibásodására vonatkozó – feltételek mellett.

Részben szinkron algoritmusok.. A részben szinkron modellek átmenetet jelentenek a szinkron és aszinkron modellek között. A részben szinkron modellekben feltesszük, hogy a processzoroknak van bizonyos információjuk az időről, például hozzáférnek egy valós idejű vagy

közelítő idejű órához, vagy valamilyen időkorlátozó tulajdonságuk van. Feltehetjük azt is, hogy a processzorok lépési ideje és/vagy az üzenettovábbítási ideje ismert alsó és felső korlátok közé esik. Mivel a részben szinkron rendszerekben kevesebb a bizonytalanság, mint az aszinkron rendszerekben, azt hihetnénk, hogy könnyebb őket programozni. Az időzítésből azonban további bonyodalmak származnak – például az algoritmusok helyessége gyakran lényegesen függ az időzítési feltételektől. Ezért az algoritmusok helyességének bizonyítása és elemzése a részben szinkron feltételek mellett gyakran bonyolultabb, mint az aszinkron feltételek mellett.

A 24. fejezetben a kölcsönös kizárási probléma megoldásának időigényére adunk alsó és felső korlátokat, míg a 25. fejezetben megegyezéssel kapcsolatos alsó és felső korlátokat adunk meg. Mivel a részben szinkronizált rendszerekkel kapcsolatos kutatások még folynak, a bemutatott eredmények szükségképpen előzetesek.

1.4.. Megjegyzések a fejezethez

A könyv anyagának fő forrása a kutatási szakirodalom, különösen az ACM PODC (*Principles of Distributed Computing*) című rendszeres évi szimpóziuma. Jelentős számú anyagot tartalmaznak az FOCS (*Foundations of Computer Science*), STOC (*Theory of Computing*), az SPAA (*Parallel Algorithms and Architectures*) és az évente megrendezett WDAG (*Workshop on Distributed Algorithms*) szimpóziumok. Az anyag nagy része megjelent az olyan informatikai folyóiratokban, mint a *Journal of ACM*, a *Distributed Computing*, a *Information and Computation*, a *SIAM Journal on Computing*, az *Acta Informatica* vagy az *Information Processing Letters*. Ezek a cikkek nagyon sokféle modellre vonatkoznak és a pontosságot tekintve nagyon különböző szintűek.

Több korábbi kísérlet történt ezen terület eredményeinek összegyűjtésére és rendszerezésére. A *Handbook of Computer Science* [185] osztott számításokról szóló, Lamport és Lynch által írt fejezete bizonyos modellezési és algoritmikus gondolatok vázlatos áttekintése. Raynal két könyve [249, 250] tartalmazza a kölcsönös kizárás és az aszinkron hálózatok algoritmusait. Raynal és Helary könyve [251] a hálózati szinkronizátorokról tartalmaz eredményeket. Chandy és Misra [69] könyve, amely a Unity programozási modellt alkalmazza, az osztott algoritmusok tartalmas gyűjteménye. Tel [276] egy másik nézőpontból vizsgálja a témát.

A szinkron közös memóriájú rendszerek PRAM modelljére vonatkozó eredményeket Karp és Ramachandran [166] gyűjtötte össze. A rögzített kapcsolatú hálózat szinkron párhuzamos algoritmusai Leighton könyvében [193] találhatók

meg. Lynch, Merritt, Weihl és Fekete [207], valamint Bernstein, Hadzilacos és Goodman [50] a párhuzamos vezérlés és az osztott adatfeldolgozó rendszerek számos algoritmusát mutatják be. Hadzilacos és Toueg [143] az atomi üzenetszórót tartalmazó kommunikációs rendszereken alapuló osztott rendszerek megvalósítására vonatkozó eredményeket tárgyalják.

A könyvben sok gráfelméleti fogalom szerepel. Meghatározásuk megtalálható Harary klasszikus művében [147], illetve Cormen, Leiserson, Rivest és Stein könyvében [83], amely magyarul *Algoritmusok* címen jelent meg.

1.5.. Jelölések

Itt összegyűjtöttük a könyvben leggyakrabban előforduló jelöléseket.

$(i, j) \leq_\gamma (k, l)$(folyamatindex, időpont) párok reflexív parciális rendezése

$\mathbb{N}$ a természetes számok $\{0,\ 1,\ 2,\ \ldots\}$ halmaza;

$\mathbb{N}^+$ a pozitív egész számok $\{1,\ 2,\ 3,\ \ldots\}$ halmaza;

P_i és i az i-edik folyamat;

$\mathbb{R}^+$ a pozitív valós számok halmaza; ;

$\mathbb{R}_0^+$ a nemnegatív valós számok halmaza; ;

$U \xrightarrow[p]{t} U'$ azt jelenti, hogy ha egy $\mathcal{A}$ tanácsadó az U állapotból indul, akkor legfeljebb t időn belül legfeljebb p valószínűséggel jut el az U' állapotba;

$\alpha \cdot \beta$ az α és β végrehajtási szeletek összeláncoltját jelenti α utolsó állapota nélkül;

$\alpha \overset{i}{\sim} \beta$ azt jelenti, hogy az α és β sorozatok *megkülönböztethetetlenek* a P_i folyamat számára;

$\alpha \sim \beta$ azt jelenti, hogy az α és β sorozatok *megkülönböztethetetlenek* a mindkét sorozatban hibátlanul működő P_i folyamat számára;

$\beta \mid S$ a β sorozat azon *részsorozata*, amely a β sorozatnak az S halmazhoz tartozó elemeit tartalmazza;

λ az üres karakterlánc;

$\varphi \rightarrow_\beta \rho$ azt jelenti, hogy a β eseménysorozatban a ρ esemény *függ* a φ eseménytől;

$\rho \approx \rho'$ a $\sim$ reláció tranzitív lezártja.

I. SZINKRON HÁLÓZATI ALGORITMUSOK

A könyv első része a 2. fejezettől a 7. fejezetig tart. A fejezetekben algoritmusok és alsó korlátok találhatók, ezek olyan *szinkron hálózati modellre* vonatkoznak, amelyben a hálózati processzorok szinkron körökben lépnek és cserélnek üzeneteket.

Ennek a résznek az első fejezete, a 2. fejezet, a szinkron hálózatokra megadott formális modellünket írja le. Az olvasó átugorhatja ezt a fejezetet, és ha szükséges, visszatérhet ide akkor, amikor az algoritmusokkal foglalkozó 3–7. fejezeteket olvassa. A 3. fejezet azzal az egyszerű problémával foglalkozik, hogyan történik *egyetlen vezető kiválasztása egy gyűrűs hálózatban*. A 4. fejezet a tetszőleges gráfú szinkron hálózatokban használt alapalgoritmusokat tekinti át. Az 5. és 6. fejezet azokkal az alapproblémákkal foglalkozik, hogyan lehet vonalhibák és processzorhibák esetén a szinkron hálózatokban *megegyezést elérni*. Végül, a 7. fejezet az alapvető megegyezési problémák kiterjesztéseit és variációit tartalmazza.

2. fejezet

Modellezés/I. Szinkron hálózati modell

Ez a könyv legrövidebb fejezete. Mindössze egy egyszerű kiszámítási modellt ad a szinkron hálózati algoritmusokra. A modellt elkülönítve adjuk meg, úgy, hogy ezt a fejezetet könnyen használhassuk majd a 3–7. fejezet olvasásakor is.

2.1.. Szinkron hálózati rendszerek

A *szinkron hálózati rendszer* egy irányított hálózati gráf csúcsaiban elhelyezett számítási elemekből áll. Az 1. fejezetben ezek a számítási elemek „processzorok" voltak, amiből arra lehet következtetni, hogy ezek valamilyen hardver berendezések. Ezeket azonban gyakran hasznosabb olyan logikai „szoftverfolyamatoknak" tekinteni, amelyek a processzorokon futnak (de nem azonosak a processzorokkal). A bemutatandó eredmények mindkét értelmezésre érvényesek. A könyvben mostantól kezdve a számítási elemeket „folyamatoknak" fogjuk nevezni.

A szinkron hálózati rendszer formális meghatározását egy $G = (V, E)$ irányított gráffal kezdjük. Jelöljük n-nel $|V|$-t, a hálózati irányított gráf csúcsainak számát. G minden P_i csúcsára jelöljük $ki\_szomszédok_i$-vel P_i „kimenő szomszédait", azaz azokat a csúcsokat, amelyekbe a P_i-ből irányított élek vezetnek, és $be\_szomszédok_i$-vel P_i „bemenő szomszédait", azaz azokat a csúcsokat, amelyekből él vezet P_i-be. Jelöljük $távolság(i, j)$-vel a P_i-ből a P_j-be vezető G-beli legrövidebb út hosszát; ha ilyen út nem létezik, akkor legyen $távolság(i, j) = \infty$. Határozzuk meg *átm*-ot, az *átmérőt* úgy, hogy minden (i, j) párra meghatározzuk a $távolság(i, j)$-k maximumát.

Azt is feltesszük, hogy adott egy rögzített M ábécé, az üzenetek ábécéje, és *null* jelöli azt, hogy nincs üzenet (*null nem eleme M-nek*).

Minden $i \in V$ csúcshoz hozzárendelünk egy olyan *folyamatot*, amely formálisan a következő összetevőkből áll:

- $állapotok_i$, az *állapotok* (nem feltétlenül véges) halmaza;
- $kezdő_i$, az $állapotok_i$ nemüres részhalmaza, a *kezdőállapotok* vagy *kezdőállapotok* halmaza;

- *üzenetek*$_i$, az *üzenetgeneráló függvény*, ami az *állapotok*$_i \times$ *ki_ szomszédok*$_i$-t leképezi $M \cup \{null\}$-ra;
- *átmenetek*$_i$, *állapotátmeneti függvény*, ami az *állapotok*$_i$-t és az $M \cup \{null\}$ elemeinek vektorait (amelyeket a *be_ szomszédok*$_i$-vel indexelünk) leképezi *állapotok*$_i$-re.

Tehát minden folyamatnak van egy állapothalmaza, az állapothalmaz egy részhalmazának pedig kitüntetett szerepe van, ez a kezdőállapotok részhalmaza. Az állapotok halmaza nem feltétlenül véges. Ez fontos, mivel lehetővé teszi azt, hogy olyan rendszereket modellezzünk, amelyekben nemkorlátos adatszerkezetek, például számlálók is vannak. Az üzenetgeneráló függvény minden állapothoz és kimenő szomszédhoz megadja az üzenetet (vagy nem ad üzenetet), ezt a P_i folyamat elküldi az adott állapotból induló megadott szomszédhoz. Az állapotátmeneti függvény minden állapotra és minden bemenő szomszédtól jövő üzenethalmazra megadja azt az új állapotot, amelyikbe a P_i folyamat megy.

A G minden (i, j) éléhez adott egy *csatorna*, amit *vonalnak* is nevezhetünk, ez lényegében egy olyan tároló, amely minden időpillanatban legfeljebb egy M-beli üzenetet tud tárolni.

Az egész rendszer végrehajtása úgy indul, hogy minden folyamat egy tetszőleges kezdőállapotban van és a csatornák üresek. Ezután a folyamatok óraütemenként a következő két lépés végrehajtását ismétlik.

1. A pillanatnyi állapotra alkalmazni kell az üzenetgeneráló függvényt, ez hozza létre az össszes kimenő szomszédhoz küldendő üzenetet. Az üzeneteket a megfelelő csatornákra kell helyezni.
2. Az állapotátmeneti függvényt a pillanatnyi állapotra és a bejövő üzenetekre kell alkalmazni, a függvény megadja az új állapotot. A csatornákról el kell távolítani minden üzenetet.

A két lépés kombinációját *menetnek* nevezzük. Megjegyezzük, hogy – általában – nem teszünk semmilyen korlátozást azoknak a számításoknak a méretére, amiket a folyamat végez akkor, amikor meghatározza az üzenetgeneráló és az állapotátmeneti függvények értékeit. Azt is megjegyezzük, hogy az itt bemutatott modell determinisztikus abban az értelemben, hogy az üzenetgeneráló függvény és az állapotátmeneti függvény (egyértékű) függvények. Így, ha adott a kezdőállapotok egy halmaza, akkor a végrehajtás csak egy úton haladhat.

Megállás.. Eddig még egyáltalán nem beszéltünk a *folyamatok megállásáról*. Könnyű elkülöníteni néhány folyamat-állapotot, mint *megállító állapotot*, és azt mondani, hogy ezekből az állapotokból semmilyen tevékenység nem indul, azaz semmilyen üzenet nem jön létre és az állapotátmenet eredménye is az eredeti állapot. Megjegyezzük, hogy ezek a megállító állapotok nem azt a szerepet játsszák, mint a hagyományos véges állapotú automaták ilyen állapotai. Azok általában mint *elfogadó állapotok* szerepelnek, azaz olyan állapotok, amelyek meghatározzák, hogy egy nyelv mely jelsorozatait számította ki a gép. Itt ezek az állapotok csak arra szolgálnak, hogy a folyamatokat megállítsák; azt, hogy egy folyamat mit számolt ki, más szabályoknak megfelelően kell meghatározni. Az elfogadó állapotok fogalmát az osztott algoritmusok rendszerint nem használják.

Különböző kezdési időpontok.. Időnként olyan szinkron rendszerekkel is foglalkozni akarunk, amelyben a folyamatok különböző menetekben kezdik a működésüket. Ezt az esetet úgy modellezhetjük, hogy a hálózati gráfot kiegészítjük egy olyan *környezeti csúccsal*, amelyikből él indul minden más csúcsba. A csúcshoz tartozó *környezeti folyamat* speciális *ébresztő* üzeneteket küld a többi folyamatnak. A többi folyamat minden kezdőállapota kezdetben *csendes*, ami azt jelenti, hogy semmilyen üzenetet nem hoz létre, és ezt a viselkedést csak akkor változtatja meg, ha a környezeti folyamattól *ébresztő* üzenetet kap, vagy ha valamelyik másik folyamat nem-*null* üzenetet küld neki. Tehát egy folyamat elindulhat közvetlenül, a környezeti folyamattól kapott *ébresztő* üzenet miatt, vagy közvetetten, egy már korábban elindult folyamat nem-*null* üzenete által.

Irányítatlan gráfok.. Néha azokkal az esetekkel is foglalkozni akarunk, ahol a hálózati gráf irányítatlan. Ezt az esetet az irányított gráfokra korábban már megadott modellel úgy írhatjuk le, hogy egy olyan irányított gráfhálózatot adunk meg, ahol bármely két szomszédos csúcs között kétirányú él van. Ebben az esetben P_i gráfbeli szomszédjainak jelölésére a $szomszédok_i$ jelölést használjuk.

2.2.. Hibák

Most a szinkron rendszerek különböző típusú hibáit fogjuk áttekinteni; foglalkozunk mind a *folyamathibákkal*, mind a *vonal- (csatorna-) hibákkal*.

A folyamat *megállási hibát* mutat akkor, ha egyszerűen megáll valahol a működése közben. Modellünk fogalmainak megfelelően a folyamat hibás lehet a fenti első vagy második lépés valamelyik végrehajtása előtt vagy után; sőt azt is megengedjük, hogy valahol az első lépés közepén is hibás lehet. Ez azt jelenti, hogy a folyamat csak az üzenetek

egy részhalmazát tudja az üzenetcsatornákra helyezni. Feltesszük, hogy ez *tetszőleges* részhalmaz lehet – és nem gondoljuk azt, hogy a folyamat az üzeneteket sorban állítja elő, és hogy valahol a sor közepén hibásodik meg.

A folyamat *bizánci hibát* is mutathat: ekkor a folyamat önkényes módon hozza létre a következő üzeneteket és a következő állapotot, anélkül tehát, hogy a megadott üzenetgeneráló és állapotátmeneti függvényeket használná.

A vonal hibás lesz, ha az üzenetek elvesznek. Modellünk kifejezéseivel: a folyamat megpróbál egy üzenetet a csatornára helyezni az első lépés alatt, de a hibás vonal nem tudja rögzíteni az üzenetet.

2.3.. Bemenetek és kimenetek

Eddig még semmit sem szóltunk a bemenetek és kimenetek modellezéséről. Az állapotokban szereplő bemenetek és kimenetek kódolására egy egyszerű módszert használunk, mégpedig azt, hogy a bemenetek a kezdő állapotokban levő kijelölt bemeneti változókba kerülnek. Az, hogy egy folyamatnak több kezdőállapota lehet, itt most azért fontos, hogy összegyűjthessük a különböző

lehetséges bemeneteket. Valóban, feltesszük, hogy a több kezdőállapot *csak*

azért szükséges, hogy lehetséges legyen az, hogy különböző bemenetértékek legyenek a bemeneti változókban. A kimenő értékek a kijelölt kimeneti változókban vannak; ezek a változók csak a rájuk vonatkozó első írási művelet eredményét tárolják (azaz *kizárólagosan írható* változók). Ugyanakkor a kimeneti változókat tetszőlegesen sokszor lehet kiolvasni.

2.4.. Végrehajtási sorozatok

A szinkron hálózati rendszer viselkedésének bemutatásához meg kell adnunk a rendszer „végrehajtásának" formális leírását.

A rendszer egy *állapot-hozzárendelését* úgy határozzuk meg, hogy a rendszer minden folyamatához egy állapotot rendelünk. Az *üzenethozzárendelés* azt jelenti, hogy minden csatornához hozzárendelünk egy üzenetet (esetleg a *null* üzenetet). Egy rendszer *végrehajtási sorozatát* a végtelen

$$C_0, M_1, N_1, C_1, M_2, N_2, C_2, \ldots$$

sorozattal adjuk meg, ahol mindegyik C_r egy állapot-hozzárendelés és mindegyik M_r és N_r egy üzenet-hozzárendelés. M_r, N_r és C_r $(r = 1, 2, \ldots)$ együtt alkotják a végrehajtási sorozat r-edik menetét: C_r a rendszer állapotát jelenti az r-edik menet után, míg M_r és N_r jelenti az r-edik menet alatt elküldött és kapott üzeneteket. (Ezek különbözők lehetnek, mivel a csatornákon az üzenet elveszhet.) A C_r-re gyakran úgy hivatkozunk, mint az r *időpontban* jelentkező állapothozzárendelésre, az r időpont az r-edik menet végrehajtása utáni időpontot jelzi.

Ha α és α' egy rendszer két végrehajtási sorozata, akkor azt mondjuk, hogy a P_i folyamat szempontjából az α *megkülönböztethetetlen* α'-től és ezt $\alpha \overset{i}{\sim} \alpha'$-vel jelöljük, ha α-ban és α'-ben P_i-nek ugyanaz az állapotsorozata, ugyanaz a kimenő üzeneteinek a sorozata, és ugyanaz a bemenő üzeneteinek a sorozata. Hasonlóan, azt mondjuk, hogy *a P_i folyamat szempontjából α és α' megkülönböztethetetlen az r-edik menetig*, ha α-ban és α'-ben P_i-nek az r-edik menet befejezéséig ugyanaz az állapotsorozata, ugyanaz a kimenő üzeneteinek a sorozata, és ugyanaz a bemenő üzeneteinek a sorozata. Ezeket a meghatározásokat kiterjesztjük azokra az esetekre is, ahol két különböző szinkron rendszer végrehajtási sorozatait hasonlítjuk össze.

2.5.. Bizonyítási módszerek

A szinkron rendszereknél használt legfontosabb bizonyítási módszer az *invariáns állítások* bizonyítása. Az invariáns állítás a rendszer állapotának (minden folyamat állapotának) olyan tulajdonsága, amely igaz minden végrehajtás, minden menet után. Megengedjük, hogy a befejezett menetek száma szerepeljen egy állításban, azért, hogy minden r-edik menet utáni állapotról állításokat mondhassunk. A szinkron rendszerekre vonatkozó invariáns állítások általában r szerinti teljes indukcióval bizonyíthatók, ahol r a befejezett menetek száma, és r kezdőértéke nulla.

Egy másik fontos módszer a *szimuláció*. Vázlatosan, a cél az, hogy megmutassuk, egy A szinkron algoritmus utánoz egy másik B szinkron algoritmust abban az értelemben, hogy ugyanazt a bemenő/kimenő működést mutatja. Az A és B közötti kapcsolatot olyan, az A és B állapotaira vonatkozó állítások fejezik ki, ahol a két algoritmus azonos bemeneti adatokkal indul és az azonos sorszámú menetekre azonos hibajelenségeket ad. Az ilyen állítást *szimulációs relációnak* nevezzük. Az invariáns állításokhoz hasonlóan a szimulációs relációkat is a befejezett menetek száma szerinti teljes indukcióval bizonyítjuk.

2.6.. Bonyolultsági mértékek

A szinkron osztott algoritmusokra két bonyolultsági mértéket szokás használni, az időbonyolultságot és a kommunikációs bonyolultságot.

A szinkron rendszerek *időbonyolultságát* a végrehajtott meneteknek az a száma adja, ameddig a kívánt kimenetek megjelennek vagy amikor minden folyamat megáll. Ha a rendszer lehetővé teszi a különböző kezdési időpontokat, a bonyolultság attól az első menettől kezdve számítódik, amelyben minden folyamat *felébredt*.

A *kommunikációs bonyolultság* jellemzően az elküldött nem-*null* üzenetek számát jelenti. Esetenként az

elküldött üzenetek bitszámát is figyelembe fogjuk venni.

A gyakorlatban az időmérték a fontosabb mérték, nem csak a szinkron osztott algoritmusok esetében, hanem minden osztott algoritmusnál. A kommunikációs bonyolultság főleg akkor fontos, ha olyan forgalomtorlódást okoz, ami lelassítja a végrehajtást. Emiatt mi nem foglalkozunk ezzel, csak az időbonyolultságot vizsgáljuk. A kommunikációs terhelésnek az időbonyolultságra való hatása azonban nem csak egyetlen osztott algoritmustól függ. A tipikus hálózatokban sok osztott algoritmus fut egyidejűleg, megosztva ugyanazt a hálózati sávszélességet. Egy algoritmus által adott vonalra küldött üzenet megnöveli ezen a vonalon a teljes üzenetterhelést, így hozzájárul ahhoz a forgalomtorlódáshoz, amit mindegyik algoritmus lát. Mivel nehéz kifejezni azt a hatást, amit az egyes algoritmusok üzenetei egy másik algoritmus időteljesítményére gyakorolnak, egyszerűen az egyes algoritmusok által létrehozott üzenetek darabszámát elemezzük (és megkíséreljük a darabszám minimalizálását).

2.7.. Véletlenítés

Ahelyett, hogy a folyamatok determinisztikusságát követelnénk meg, néha hasznos, ha a folyamatoknak lehetővé tesszük a véletlenszerű választást, ami bizonyos adott valószínűségi eloszlásokon alapul. Mivel az alap szinkron rendszer modellje ezt nem teszi lehetővé, kiegészítjük a modellt, az üzenetgeneráló és átmeneti függvényekhez hozzáveszünk egy új *véletlen függvényt*, ami a véletlenszerű választás lépéseit valósítja meg. Formálisan, az automata leírását a minden P_i csúcshoz megadott $véletlen_i$ összetevővel bővítjük; $véletlen_i(s)$ minden s állapotra egy, az $állapotok_i$ egy részhalmazán értelmezett valószínűségi eloszlást ad. A végrehajtási

sorozat minden menetében először a $véletlen_i$ véletlen függvényt alkalmazzuk. Ez meghatározza az új állapotokat, és ezután az $üzenetek_i$ és $átmenetek_i$ függvényeket a szokásos módon használhatjuk.

A véletlenszerű algoritmusra a végrehajtási sorozat formális leírása nem csak az állapot-hozzárendeléseket és üzenet-hozzárendeléseket, hanem a véletlen függvényekre vonatkozó információt is tartalmazza. A rendszer egy *végrehajtási sorozata* egy végtelen

$$C_0, D_1, M_1, N_1, C_1, D_2, M_2, N_2, C_2, \ldots$$

sorozat, ahol mindegyik C_r és D_r egy állapot-hozzárendelés és mindegyik M_r és N_r egy üzenet-hozzárendelés. D_r ábrázolja az r-edik menet utáni véletlen választással kapott új folyamatállapotokat.

Azok az állítások, melyeket a véletlenszerű rendszerekkel számítunk ki, rendszerint valószínűségeket tartalmaznak és azt állapítják meg, hogy bizonyos eredmények legalább mekkora valószínűséggel érhetők el. Amikor egy ilyen állítást megfogalmazunk, a cél általában az, hogy igaz legyen minden bemenetre, és a hibákat is tartalmazó rendszerek esetében fennálljon bármilyen hiba esetén is. A bemeneteket és a hibaeseteket modellezve rendszerint feltételezünk egy fiktív, *ellenfélnek* nevezett elemet is, ez szabályozza a bemenetek és a hibaelőfordulások véletlenszerűségét, és egy valószínűségi állítás azt mondja, hogy a rendszer helyesen viselkedik minden megengedett ellenféllel versenyezve. Ennek a kérdésnek az általános vizsgálata kívül esik e könyv témakörén, csak azokkal a speciális esetekkel foglalkozunk, amelyek szükségesek lesznek.

2.8.. Megjegyzések a fejezethez

Az állapotokkal rendelkező gépek modelljének általános fogalma a hagyományos véges állapotú automata modellekből származik. A véges állapotú gépekre vonatkozó alapvető tananyagok sok egyetemi tankönyvben megtalálhatók, ilyen például Lewisés Papadimitriou[195] és Martin[221] könyve. Az itt leírt állapotokkal rendelkező gépmodell speciális esetei az osztott számítások elméletének számos cikkéből származtathatók, például Fischer és Lynch [119]bizánci megegyezésre vonatkozó cikkéből.

Az invariáns állításokgondolatát először Floyd[124] javasolta soros feldolgozású programokra, majd Ashcroft [15] és Lamport [175] általánosította párhuzamos programokra. Hasonló gondolat számos helyen található. A szimuláció gondolatának is sok forrása van. Az egyik legfontosabb a soros feldolgozású programokban, például Liskov CLU programnyelvében [198], Milner [228] és Hoare [158] munkáiban megvalósított adatabsztrakciókra vonatkozó korai munka. Ilyen későbbi munkák, amelyek kiterjesztették a fogalmat a párhuzamos programokra, Park [236], Lamport [177], Lynch [203], Lynch és Tuttle [218] és Jonsson [165] cikkei.

3. fejezet

Vezető folyamat kiválasztása szinkron gyűrűben

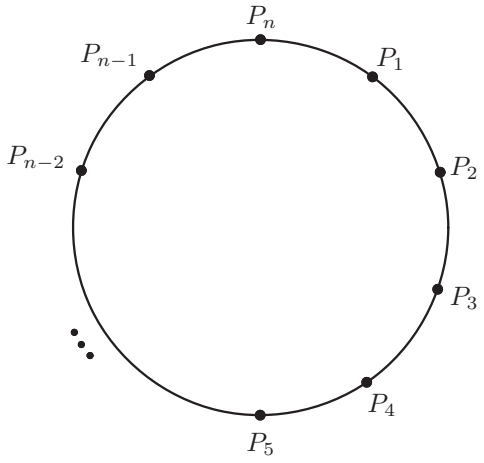
Ebben a fejezetben az első megoldandó problémaként a második fejezet szinkron modelljét használó alábbi feladatot mutatjuk be: egy hálózat folyamatai közül egy *egyértelműen meghatározott vezető folyamat kiválasztását*. Először vegyük azt az egyszerű esetet, amikor a hálózat topológiája gyűrű.

A probléma eredetileg a *vezérlőjeles gyűrű* (token ring) alapú helyi hálózatok vizsgálatainál vetődött fel. Egy ilyen hálózatban egyetlen ún. „vezérlőjel" (token) kering körbe-körbe, az éppen aktuális tulajdonosának biztosítva kizárólagos jogot kapcsolat kezdeményezésére. Ha a hálózat két csomópontja egyidőben kísérelne meg kommunikálni, az üzenetek zavarnák egymást. Néha azonban a vezérlőjel elveszhet. Ilyen esetekben a folymatoknak egy olyan algoritmust kell végrehajtaniuk, ami újra képezi az elveszett vezérlőjelet. Ezt a regeneráló eljárást hívjuk a vezető folyamat kiválasztásának.

3.1.. A feladat

Tételezzük fel, hogy a G hálózatunk egy n csúcsból álló gyűrű, az óramutató járásával egyező irányban számozva 1-től n-ig (lásd a 3.1. ábrát). Számításainkat általában modulo n végezzük, megengedve ezzel azt, hogy 0-val azonosítsuk az n-edik folyamatot, $(n+1)$-gyel az elsőt, és így tovább. A G gráf csúcsaihoz rendelt folyamatok nem ismerik a saját azonosítójukat, sem pedig szomszédaikét. Feltételezzük továbbá, hogy az üzeneteket létrehozó és továbbító függvények helyi kifejezésekkel adottak, relatív nevekkel utalva a szomszédokra. Azt is feltételezhetjük, hogy minden folyamat képes az óramutató járásával megegyező irányba eső szomszédját megkülönböztetni a másiktól. A követelmény az, hogy végül pontosan egy folyamat döntsön úgy, hogy ő a vezető folyamat, és ekkor rendeljen *vezető* értéket állapotának egy speciális *státus* (állapot) összetevőjéhez. A problémának különböző változatai vannak.

1. Az is megkövetelhető lenne, hogy a vezető folyamaton kívüli összes folyamat előbb-utóbb azt a kimenetet eredményezze, hogy ő nem vezető, mondjuk



3.1.. ábra. Folyamatok egy gyűrűje.

nem_ vezető értékre változtatva *státus* összetevőjét.

2. A gyűrűt leíró gráf lehet *csak az egyik irányban irányított* vagy *mindkét irányban irányított*. Ha csak az egyik irányban irányított, akkor ez az irány az általánosság megszorítása nélkül lehet az óramutató járásával megegyező irány, vagyis üzenetek csak ebben az irányban küldhetők a szomszéd csúcs felé.

3. A gyűrűt alkotó csúcsok n számát a folyamatok vagy ismerik, vagy nem. Ha ez ismert, a folyamatoknak csak az n méretű gyűrűben kell pontosan működniük, és ekkor n értékét a folyamatok használhatják programjaikban. Ha n nem ismert, feltételezhető, hogy a folyamatok különböző méretű gyűrűkben futnak. Ekkor a gyűrű méretéről semmilyen információ nem használható fel.

4. A folyamatok lehetnek azonosak, vagy lehetnek különbözők. Utóbbi esetben kezdődhetnek egy *egyedi azonosítóval* (*UID*), amelyek egy megfelelően nagy teljesen rendezett halmazból, például a természetes számok $\mathbb{N}$ halmazából választhatók. Feltételezhetjük, hogy a gyűrűben a folyamatazonosítók különböznek egymástól, de nincs megszorítás arra, hogy milyen UID-ek szerepelhetnek ténylegesen a gyűrűben. (Például nem szükséges, hogy egymás utáni egészek legyenek.) Megadható továbbá, hogy az azonosítókat kizárólagosan mely műveletek használhatják (például összehasonlítások), vagy nem teszünk ilyen jellegű megszorításokat.

3.2.. Megoldhatatlansági eredmény azonos folyamatokra

Először is könnyű észrevenni, hogy ha az összes folyamat azonos, akkor a problémának az adott modellben nem létezik megoldása. Még akkor is ez a helyzet, ha a gyűrű mindkét irányban irányított és a folyamatok ismerik a gyűrű méretét.

3.1. tétel. . *Legyen A egy n folyamatból ($n > 1$) álló rendszer, egy mindkét irányban irányított gyűrűben elrendezve. Ha A összes folyamata azonos, akkor A nem oldja meg a vezető folyamat kiválasztásának problémáját.*

Bizonyítás. Indirekt módon, tegyük fel, hogy létezik egy olyan A rendszer, ami megoldja a vezető folyamat kiválasztásának problémáját. Az általánosság megszorítása nélkül feltehető, hogy A minden egyes folyamatának pontosan egy kezdőállapota van. Ez azért igaz, mert ha minden folyamatnak több kezdőállapota is létezik, tetszőlegesen kiválaszthatunk közülük egyet, így egy olyan új megoldást kapunk, ahol minden folyamatnak már csak egy kezdőállapota van. Ezzel a feltételezéssel élve A-nak pontosan egy végrehajtási sorozata létezik.

Tekintsük tehát A-nak az egyetlen végrehajtási sorozatát. A végrehajtott menetek r száma szerinti indukcióval nem nehéz igazolni, hogy az r-edik menet után minden folyamat ugyanabban az állapotban van. Vagyis, ha valamely folyamat valaha is eléri azt az állapotot, amikor a *státus* állapota *vezető*, akkor A összes folyamata ugyanebben az időben szintén ugyanebben az állapotban lesz. Ez ellentmond az egyértelműségre vonatkozó követelménynek. □

A 3.1. tételből következik, hogy a vezető folyamat kiválasztásának problémáját csakis úgy lehet megoldani, ha valahogy megtörjük a szimmetriát. A gyakorlatban rendszerint azzal az ésszerű feltételezéssel élnek, hogy a folyamatok azonosak, kivéve az UID-et. A fejezet hátralévő részében mi is ezt a feltételt fogjuk használni.

3.3.. Egy alapvető algoritmus

Az első bemutatandó megoldás nagyon egyszerű. Le Lann, Chang és Roberts tiszteletére, akiknek a munkáiból az algoritmust vettük, LCR algoritmusnak fogjuk nevezni. Az algoritmus csak egyirányú kapcsolatot használ és nem igényli a gyűrű méretének ismeretét. Ezenkívül csak a vezető folyamatnak lesz kimenete. Az algoritmus az UID-ek között csak az összehasonlítás műveletét használja. Az alábbiakban az LCR algoritmus vázlatos leírását adjuk meg.

LCR algoritmus (vázlatosan)

Minden folyamat körbeküldi az azonosítóját a gyűrűben. Amikor valamely folyamathoz egy azonosító érkezik, összehasonlítja azt a sajátjával. Ha az érkezett azonosító nagyobb, mint a sajátja, a folyamat továbbítja, ha kisebb, eldobja. Amennyiben az érkezett azonosító megegyezik a sajátjával, a folyamat önmagát nevezi meg vezetőnek.

Ebben az algoritmusban a legnagyobb UID-del rendelkező folyamat az egyetlen, amelyik a *vezető* kimenetet adja. Intuíciónkat világosabbá téve a második fejezet modelljének segítségével most algoritmusunknak egy sokkal pontosabb leírását adjuk meg.

LCR algoritmus (formálisan)
Az üzenetek M ábécéje pontosan az UID-ek halmaza.
Az *állapotok*$_i$-ben lévő állapotok minden P_i folyamatra az alábbi összetevőket tartalmazzák:

> *u*, egy UID, kezdetben a P_i folyamat UID-je,
> *küld*, egy UID vagy *null*, kezdetben a P_i folyamat UID-je,
> *státus*, az {*ismeretlen, vezető*} értékek valamelyike, kezdetben *ismeretlen*.

A kezdőállapotok *kezdő*$_i$ halmaza az imént meghatározott kezdőértékekkel egyetlen állapotból áll.
Az *üzenetek*$_i$ üzenetgeneráló függvényt minden P_i folyamatra az alábbi módon adjuk meg:

> **send** *küld* aktuális értékét a P_{i+1} folyamatnak

Látjuk, hogy a P_i folyamatban relatív hivatkozás történik a P_{i+1} folyamatra. Például az „óramutató járásával megegyező irányba eső szomszéd" helyett egyszerűen P_{i+1}-et írunk. Emlékezzünk vissza, hogy a második fejezetben a *null* értéket használtuk az üzenet hiányának jelölésére. Vagyis ha a *küld* összetevő értéke *null*, a megfelelő *üzenetek*$_i$ függvény ténylegesen nem eredményez semmilyen üzenetet.
Az *átmenetek*$_i$ állapotátmeneti függvényt minden P_i folyamatra az alábbi pszeudokóddal adjuk meg:

```
küld := null
if az érkező üzenet v (egy UID) then
    case
        v > u: küld := v
        v = u: státus := vezető
        v < u: do semmi
    endcase
```

Az állapotátmeneti függvény definíciójának első sora törli a korábban érkezett üzenet hatását (ha volt ilyen), *null*-ra állítva a *küld* állapotösszetevőt. A kód többi része az igazán érdekes – ez tartalmazza a döntést arról, hogy a függvény továbbítsa vagy eldobja a beérkező UID-et, esetleg elfogadja azt, hogy önmagát kiálthassa ki vezető folyamatnak.

Az iménti leírás egy könnyen olvasható programozási nyelven íródott, de azért jegyezzük meg, hogy szükség esetében közvetlenül lefordítható lenne a második

fejezetben látott állapotokkal rendelkező gépmodellre. Ebben az értelmezésben az összes folyamat minden változójának van értéke, és az átmenetek a változók értékeinek megváltozását jelentik. Jegyezzük meg, hogy az *átmenetek*$_i$ függvény kódjának teljes blokkja egyetlen menetben, oszthatatlanul hajtódik végre.

Hogyan adjunk formális bizonyítást algoritmusunk helyességére? A helyesség most azt jelenti, hogy pontosan egy folyamat fogja előbb-utóbb a *vezető* kimenetet adni. Jelölje i_{max} a maximális UID-ű folyamat indexét és jelölje u_{max} ennek UID-jét. Elég lenne azt bebizonyítani, hogy (1) az n-edik menet végén az i_{max} folyamat adja a *vezető* kimenetet, és (2) egyetlen más folyamat sem ad ilyen kimenetet. A 3.2. és a 3.3. lemmákban ezen tulajdonságokat fogjuk bebizonyítani.

Itt és a könyv sok más részén is az i alsó indexet írjuk azon állapotösszetevők nevéhez, amelyeknél jelezni akarjuk, hogy a megfelelő állapotösszetevő példánya a P_i folyamathoz tartozik. Például az u_i jelölést használjuk a P_i folyamat u állapotösszetevője értékének jelölésére. A folyamat kódjának írásakor azonban általában nem használjuk az alsó indexes jelölést.

3.2. lemma. . *Az n-edik menet végén az i_{max} folyamat a „vezető” kimenetet adja.*

Bizonyítás. Vegyük észre, hogy a kezdőérték-adás miatt u_{max} az $u_{i_{max}}$ változó kezdeti értéke, vagyis az i_{max} folyamat u változójáé. Azt is észrevehetjük, hogy az u változók értékei sohasem változnak meg (a kód szerint), mind különbözők (a feltétel miatt), valamint hogy i_{max}-nak van a legnagyobb u értéke (i_{max} definíciója miatt). A kód szerint elegendő megmutatni, hogy az alábbi invariáns állítás teljesül:

3.3.1. állítás. *Az n-edik menet után státus$_i$ = „vezető”.*

Az invariáns bizonyításának legkézenfekvőbb módja az, ha a menetek száma szerinti indukciót használunk.

Hogy ezt megtehessük, szükségünk lesz egy előzetes invariánsra, ami alacsonyabb menetszámok esetére mond valamit. Ezért az alábbi állítással egészítjük ki a korábbikat.

3.3.2. állítás. *Az r-edik menet után minden $0 \leq r \leq n-1$ esetében küld$_{i_{max}+r} = u_{max}$.*

(Ne felejtsük el, hogy az összeadás modulo n értendő.) Az állítás azt mondja ki, hogy a *küld* összetevőben lévő maximális érték pozíciója a gyűrűben r távolságra van i_{max}-tól.

Elég nyilvánvaló, hogy a 3.3.2. állítást r szerinti indukcióval bizonyítjuk. Az $r = 0$ esetben a kezdőértékek miatt a nulladik menet után küld$_{i_{max}} = u_{max}$, ami pontosan az, amire szükségünk van. Az indukciós lépés azon a tényen alapul, hogy az i_{max}-tól különböző minden más csúcs felveszi a maximum értéket és ez tárolódik is a *küld* összetevőben, mivel u_{max} nagyobb minden más értéknél.

A következő feladat a 3.3.1. állítás bizonyítása. A 3.3.2. állítás eredményét fogjuk felhasználni az $r = n-1$ speciális esetben, továbbá még egy gondolatot

arról, hogy mi is történik egyetlen menet alatt. A kulcsészrevétel az, hogy az i_{max} folyamat elfogadja az u_{max} jelet, ami *vezető*-re állítja *státus* állapotösszetevőjét. □

3.3. lemma. . *Az i_{max}-on kívül egyetlen más folyamat sem eredményezi a „vezető" kimenetet.*

Bizonyítás. Elég azt megmutatni, hogy minden más folyamatra a *státus = ismeretlen* teljesül. Ahogy korábban, egy erősebb invariáns segíthet. Legyen P_i és P_j a gyűrű két tetszőleges folyamata, $i \neq j$, és jelölje $[i, j)$ indexek egy $\{i, i+1, \ldots, j-1\}$ halmazát, ahol az összeadás modulo n értendő. Vagyis a gyűrűben az óramutató járásával megegyező irányba haladva $[i, j)$ a P_i-vel kezdődő folyamattól ellentétes körüljárással a P_j szomszédjáig tartó folyamatokból álló halmazt jelöli. Az alábbi invariáns azt állítja, hogy semmilyen v UID nem adódik értékül egyetlen i_{max} és v eredeti i pozíciója közötti folyamat *küld* változójának sem.

> **3.3.3. állítás.** *Az r-edik menet után, ha $i \neq i_{max}$ és $j \in [i_{max}, i)$, akkor küld$_j \neq u_i$ tetszőleges r, valamint i, j esetében.*

Ismét indukciót használunk. A bizonyítás kulcsa most az, hogy egy nemmaximális UID-ű folyamathoz sohasem érkezhet ugyanaz az UID-érték. Ez azért van így, mert i_{max} összehasonlítja a bejövő értéket u_{max}-szal és u_{max} nagyobb, mint az összes többi UID.

Végezetül a 3.3.3. állítás használható annak bizonyítására, hogy csak az i_{max} folyamat kaphatja vissza a saját UID-jét és így csak i_{max} tudja a *vezető* kimenetet előállítani. □

A 3.2. és 3.3. lemmákból az alábbi tétel következik.

3.4. tétel. . *Az* LCR *algoritmus megoldja a vezető folyamat kiválasztásának problémáját.*

Megállás és *nem_vezető* kimenetek.. Amint már említettük, az LCR algoritmus sohasem ér véget abban az értelemben, hogy az összes folyamat elérné a megállás állapotát. Természetesen minden folyamatot kibővíthetnénk egy megállás állapottal, ahogy azt a 2.1. alfejezetben leírtuk. Ekkor algoritmusunkat úgy módosíthatnánk, hogy a kiválasztott vezető folyamat egy speciális *értesítő* üzenetet küldjön körbe a gyűrűn. Minden folyamat, amelyik megkapta az *értesítő* üzenetet, álljon meg, miután továbbította azt. Ez a stratégia nemcsak hogy engedélyezi a folyamatoknak a megállást, de arra is használható, hogy a nemvezető folyamatok a *nem_vezető* kimenetet adják. Továbbá hozzácsatolva az *értesítő* üzenethez a vezető indexét, a stratégia az összes részt vevő folyamatnak lehetőséget ad a vezető azonosítójának közzétételére. Jegyezzük meg, hogy minden nemvezető folyamat eredményezheti a *nem_vezető* kimenetet közvetlenül azután is, amint a beérkező UID-ből látja, hogy az nagyobb, mint a sajátja. De ez nem mondaná meg a nemvezető folyamatoknak azt, hogy mikor álljanak le.

Bonyolultságelemzés.. Az alap LCR algoritmus időbonyolultsága a kiválaszott elem bejelentkezéséig n menet. A legrosszabb eset kommunikációs bonyolultsága $\Theta(n^2)$ üzenet. Az algoritmus megállásos változatában az időbonyolultság $2n$, a kommunikációs bonyolultság továbbra is $\Theta(n^2)$. A megálláshoz és a *nemvezető* bejelentéshez szükséges többletidő csak n menet és a többlet kommunikáció is csak n üzenet.

Transzformáció.. Az előző két bekezdés egy általános transzformációt ír le és elemez, kezdve egy tetszőleges vezető folyamatot kiválasztó algoritmustól, ahol csak a vezető ad kimenetet és a többi folyamat sohasem áll le, egészen addig, amikor a vezető és nemvezető folyamatok mindegyike ad kimenetet és minden folyamat megáll. A kimenet többletköltsége a megállással együtt is csak n menet és n üzenet volt. Ez a transzformáció egyéb feltevéseink tetszőleges kombinációjában működik.

Különböző kezdési időpontok.. Jegyezzük meg, hogy az LCR algoritmus a szinkron modellben különböző kezdési időpontok esetében is módosítás nélkül működik. A modell ezen változatának leírása a 2.1. alfejezetben található.

A szimmetria megtörése.. A vezető folyamat kiválasztásának gyűrűbeli problémájában az igazi nehézség a szimmetria megtörése. A szimmetria megtörése más osztott rendszerben megoldandó problémáknak is fontos részét képezik, ilyenek például az *erőforrás-hozzárendelési* és *megegyezési* problémák. (Lásd az 5–7., 10–12., 20–21. és 25. fejezeteket).

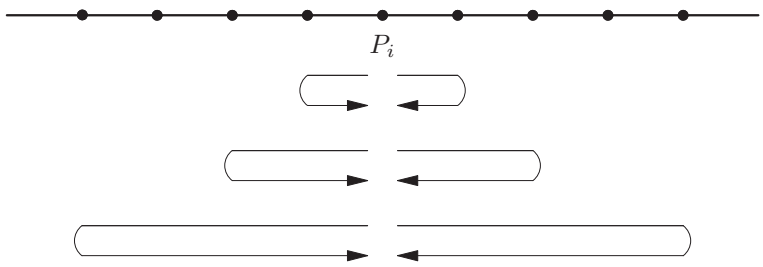
3.4.. Egy $\mathcal{O}(n \log n)$ kommunikációs bonyolultságú algoritmus

Ámbár az LCR algoritmus időbonyolultsága alacsony, az algoritmusban előforduló üzenetek száma meglehetősen magas.[1] Ez kevéssé jelentősnek tűnhet, mivel egy csatornán egyidőben soha sincs több egy üzenetnél. A második fejezetben azonban elemeztük, miért is olyan érdekes az üzenetek számának minimalizálása: sok egyidőben futó osztott algoritmus teljes kommunikációs terhelése a hálózat forgalmi torlódását eredményezheti. Ebben az alfejezetben egy olyan algoritmust mutatunk be, amely a kommunikációs bonyolultságot $\mathcal{O}(n \log n)$-re csökkenti. Az első olyan algoritmust, amelynél a legrosszabb eset bonyolultsága $\mathcal{O}(n \log n)$, Hirschberg és Sinclair tette közzé. Tiszteletükre az algoritmust HS algoritmusnak fogjuk nevezni. Ahogy korábban, ismét feltételezzük, hogy csak a vezetőnek szükséges kimenetet eredményeznie, bár a 3.3. alfejezet végén leírt transzformációból következik, hogy ez a megszorítás nem lényeges. Ahogy korábban, most is feltételezzük, hogy a gyűrű mérete nem ismert, de most mindkét irányú kommunikációt megengedjük.

Ahogy az LCR algoritmus tette, a HS algoritmus is a maximális UID-del rendelkező folyamatot választja ki. Most azonban az LCR algoritmustól eltérően, ahol minden folyamat az UID-jét elejétől végig körbeküldi a gyűrűn, a HS

[1] Minden esetben $\Theta(n^2)$. *Az alkotó szerkesztő.*

algoritmusban minden egyes folyamat UID-je bizonyos távolság megtétele után megfordul és visszatér az eredeti folyamathoz. Ezután ez ismétlődik egyre nagyobb távolságokkal. A HS algoritmus az alábbi módon jár el.



3.2.. ábra. A HS algoritmus P_i folyamatából kiinduló egymás utáni üzenetek pályája.

HS algoritmus (vázlatosan)
Minden P_i folyamat a $0, 1, 2, \ldots$ szakaszokban (fázisokban) működik. Minden l szakaszban a P_i folyamat az u_i UID-jét tartalmazó jeleket küld mindkét szomszédja felé. Ezek az üzenetek 2^l távolságot tesznek meg, azután visszatérnek a P_i kiindulási folyamathoz (lásd a 3.2. ábrát). Ha mindkét jelsorozat biztonságban visszaérkezik, a P_i folyamat a következő szakasszal folytatódik. Nem biztos azonban, hogy a jelek biztonságban vissza is érnek. Amíg az u_i jelsorozat P_i-től távolodik, minden, u_i útjába eső P_j folyamat összehasonlítja u_i-t a saját UID-jével, u_j-vel. Ha $u_i < u_j$, P_j egyszerűen eldobja az üzenetet, míg ha $u_i > u_j$, P_j továbbítja u_i-t. Ha $u_i = u_j$, akkor ez azt jelenti, hogy a P_j folyamathoz még visszafordulás előtt a saját UID-je érkezett, vagyis a P_j folyamat önmagát választja vezetőnek.
Ha a P_i folyamat által küldött üzenet közeledik, azt minden folyamat mindig továbbítja.

Most az algoritmust formálisan is megadjuk. A formalizálás némi könyvelést is igényel, hiszen biztosítani kell, hogy a jelsorozatok a megfelelő irányokba tartsanak. Például a jelsorozatoknak tartalmazniuk kell azt az információt, hogy távolodnak-e a kiindulási helyüktől vagy közelednek. Hasonlóképpen, a jelsorozatoknak tartalmazniuk kell a közben érintett folyamatok számát is, hogy távolodáskor a távolságot számon lehessen tartani. Ez lehetőséget ad a folyamatoknak arra, hogy eldöntsék, mikor kell az üzenetet visszafordítani. Az algoritmus formalizálása után az LCR algoritmusnál látott módon indokoljuk annak helyességét.

HS algoritmus (formálisan)
Az üzenetek M ábécéje olyan hármasokból álló halmaz, ahol az elemek összetevői az UID-ek, az üzenetek mozgási irányát jelző $\{$*megy, jön*$\}$ értékek valamelyike és egy pozitív egész szám (*ugrás_ számláló*), ami a közben érintett folyamatok számát jelöli.

Az *állapotok*$_i$-ben lévő állapotok minden P_i folyamatra az alábbi összetevőket tartalmazzák:

u, egy UID, kezdetben a P_i folyamat UID-je,
küld+, ami vagy M egy elemét tartalmazza, vagy *null*,
kezdetben a hármas tartalma a P_i folyamat UID-je, *megy* és 1,
küld−, ami vagy az M egy elemét tartalmazza, vagy *null*,
kezdetben a hármas tartalma a P_i folyamat UID-je, *megy* és 1,
státus, az {*ismeretlen, vezető*} értékek valamelyike, kezdetben *ismeretlen*,
fázis, nemnegatív egész, kezdetben 0.

A kezdőállapotok *kezdő*$_i$ halmaza az imént megadott kezdeti értékekkel egyetlen állapotból áll.

Az *üzenetek*$_i$ üzenetgeneráló függvényt minden P_i folyamatra az alábbi módon adjuk meg:

a *küld+* aktuális értékét továbbítsd a P_{i+1} folyamatnak,
a *küld−* aktuális értékét továbbítsd a P_{i-1} folyamatnak.

A *átmenetek*$_i$ állapotátmeneti függvény minden P_i folyamatra az alábbi pszeudokóddal adott:

küld+ := *null*
küld− := *null*
if a P_{i-1}-ből érkező üzenet (*v, megy, h*) **then**
  case
    $v > u$ és $h > 1$: *küld+* := (*v, megy,* $h-1$)
    $v > u$ és $h = 1$: *küld−* := (*v, jön,* 1)
    $v = u$: *státus* := *vezető*
  endcase
if a P_{i+1}-ből érkező üzenet (*v, megy, h*) **then**
  case
    $v > u$ és $h > 1$: *küld−* := (*v, megy,* $h-1$)
    $v > u$ és $h = 1$: *küld+* := (*v, jön,* 1)
    $v = u$: *státus* := *vezető*
  endcase
if a P_{i-1}-ből érkező üzenet (*v, jön,* 1) és $v \neq u$ **then**
  küld+ := (*v, jön,* 1)
if a P_{i+1}-ből érkező üzenet (*v, jön,* 1) és $v \neq u$ **then**
  küld− := (*v, jön,* 1)
if a P_{i-1}-ből és a P_{i+1}-ből érkező üzenetek mindegyike (*u, jön,* 1) **then**
  fázis := *fázis*+1
  küld+ := (*u, megy,* $2^{fázis}$)
  küld− := (*u, megy,* $2^{fázis}$)

Ahogy a korábbi esetben, az első két sor indítja el a folyamatot. A kód következő két részlete a távolodó üzenetek kezelését írja le: azok a jelek, amelyek UID-része nagyobb, mint u_i, vagy továbbítódnak, vagy visszafordítódnak, a közben érintett folyamatok számától függően. Amennyiben éppen az u_i érkezett, a P_i folyamat önmagát választja vezetőnek. A kód következő két részlete a közeledő üzenetek kezelésére ad választ: ezek egyszerűen továbbítódnak. (A közeledő üzenetek esetében az *ugrás_számláló* értéke 1.) Ha a P_i folyamat mindkét saját üzenetét visszakapta, a folyamat a következő szakaszba lép.

Bonyolultságelemzés.. Először a kommunikációs bonyolultságot elemezzük. A 0-dik szakaszban minden folyamat elküld egy üzenetet. Ez összesen $4n$, mindkét szomszéd felé egy elküldött, egy érkezett üzenet. Minden $l > 0$ esetében az l-edik szakaszban egy folyamat pontosan akkor küld jeleket, ha az $(l-1)$-edik szakaszban mindkét üzenet visszaérkezett. Márpedig ez a helyzet akkor, ha semmilyen más, a gyűrű bármely irányában 2^{l-1} távolságon belüli folyamat nem hiúsítja meg a továbbhaladást. Ebből az következik, hogy bármely $2^{l-1}+1$ darab egymás melletti folyamat közül legfeljebb egy olyan lesz, amelyik az l-edik szakaszban üzenetet küld. Ezen gondolat segítségével megmutatható, hogy összesen legfeljebb

$$\left\lfloor \frac{n}{2^{l-1}+1} \right\rfloor$$

folyamat küld üzenetet az l szakaszban. Az l szakaszban elküldött összes üzenetek számára így az alábbi felső becslést kapjuk:

$$4\left(2^l \cdot \left\lfloor \frac{n}{2^{l-1}+1} \right\rfloor\right) \leq 8n.$$

Azért ennyi, mert az l-edik szakaszban a jelsorozatok 2^l távolságot tesznek meg. Mint korábban, a 4-es szorzó abból adódik, hogy az üzenetek két irányban távolodnak és minden távolodó üzenet valamikor megfordul és visszaérkezik.

A vezető folyamat kiválasztása és az összes kommunikáció befejezése előtt végrehajtott szakaszok száma (a 0-dik szakasszal együtt) legfeljebb $1 + \lceil \log n \rceil$, így az üzenetek teljes száma legfeljebb $8n(1 + \lceil \log n \rceil)$, ami $\mathcal{O}(n \log n)$.

Az algoritmus időbonyolultsága $\mathcal{O}(n)$. Ezt könnyű belátni, hiszen minden l szakasz $2 \cdot 2^l = 2^{l+1}$ ideig tart (az üzenetek először távolodnak, aztán közelednek). Az utolsó szakasz n ideig tart – ez nem teljes szakasz, mert az üzenetek csak távolodnak. Az utolsó előtti szakasz az $l = (\lceil \log n \rceil - 1)$-edik, ennek időigénye legalább annyi, mint az összes ezt megelőző szakaszé összesen. Vagyis az utolsó szakasz kivételével a teljes időigény legfeljebb

$$2 \cdot 2^{\lceil \log n \rceil}.$$

Összegezve, ha n kettőhatvány, akkor a teljes időigény legfeljebb $3n$, egyébként legfeljebb $5n$. A hiányzó részletek belátását az Olvasóra bízzuk (lásd 3-7. gyakorlat).

Különböző kezdési időpontok.. A HS algoritmus a szinkron modellben különböző kezdési időpontok esetében is módosítás nélkül működik.

3.5.. Nem összehasonlítás-alapú algoritmusok

A következőkben azt a kérdést feszegetjük, hogy lehetséges-e a vezető folyamat kiválasztása kevesebb, mint $\Omega(n \log n)$ üzenettel. A válasz erre a kérdésre nemleges, a megoldhatatlansági eredményt meg is mutatjuk. Ez az eredmény azonban csak azokra az algoritmusokra igaz, amelyek kizárólag az UID-ek összehasonlításán alapulnak. (Az *összehasonlítás-alapú algoritmusokat* a 3.6. alfejezetben írjuk le.)

Ebben a fejezetben feltételezzük, hogy az UID-ek pozitív egészek, amelyeken általános aritmetikai műveleteket hajthatunk végre. Két algoritmust is adunk erre az esetre, az egyik az IdőSzelet algoritmus, a másik a VáltozóSebességek algoritmus, mindkettő $\mathcal{O}(n)$ kommunikációs bonyolultsággal. Ezen algoritmusok létezéséből az is következik, hogy az általános esetben $\Omega(n \log n)$ alsó korlát nem adható meg.

3.5.1.. Az IdőSzelet algoritmus

A most bemutatandó algoritmus azt a feltevést használja, hogy a gyűrű n méretét minden folyamat ismeri és az üzenettovábbítás egyirányú. Ezekkel a feltevésekkel az alábbi egyszerű algoritmus – amit IdőSzelet algoritmusnak fogunk nevezni – megfelelően működik. Az algoritmus a minimális UID-ű folyamatot választja ki.

Jegyezzük meg, hogy ez az algoritmus a szinkronitást mélyebb értelemben használja, mint az LCR algoritmus vagy a HS algoritmus. Az IdőSzelet algoritmus bizonyos menetekben az üzenetek nemlétét is felhasználja (egészen pontosan az érkező *null-üzenet*eket), mint információt.

> **IdőSzelet algoritmus (vázlatosan)**
>
> A számítások az $1, 2, \ldots$ szakaszokban (fázisokban) hajtódnak végre, ahol minden szakasz n egymás utáni menetből áll. Minden szakasz olyan lehetséges forgalmat jelent, amihez bizonyos UID-ű üzenetek tartoznak és ezek körbemennek a gyűrűn. Egészen pontosan a v szakaszban, ami a $(v-1)n+1, \ldots, vn$ menetekből áll, csak a v UID-ű üzenetek forgalma engedélyezett.
>
> Ha valamely időpillanatban P_i a v UID-del rendelkezik és a $(v-1)n+1$ menet anélkül ért véget, hogy a P_i folyamat előzőekben egyetlen *nem-null* üzenetet kapott volna, akkor P_i önmagát választja vezetőnek és a saját UID-jét körbeküldi a gyűrűn. Amikor ez az üzenet megy körbe a gyűrűn, az összes érintett folyamat feljegyzi annak megérkeztét. Ez a feljegyzés a későbbiekben megakadályozza őket abban, hogy önmagukat választhassák vezető folyamatnak vagy bármely későbbi fázisban üzenetküldést kezdeményezzenek.

Az algoritmusban a minimális UID (u_{min}) előbb-utóbb körbemegy a gyűrűn, aminek következtében annak eredeti tulajdonosa önmagát választja kitüntetett folyamatnak. Az $(u_{min}-1)n+1$ menet előtt nincs üzenetküldés, ahogy az $u_{min} \cdot n$ menet után sincs. Az elküldött üzenetek száma így n. Ha a minimális UID-ű folyamat helyett mi a maximális UID-ű folyamatot szeretnénk meghatározni, akkor a minimális UID-ű folyamat megkeresése után az egyszerűen körbeküld egy speciális üzenetet, aminek a segítségével u_{max} könnyen meghatározható. A kommunikációs bonyolultság így is $\mathcal{O}(n)$.

Az IdőSzelet algoritmus jó tulajdonsága, hogy az üzenetek száma n. Sajnos az időbonyolultság körülbelül $n \cdot u_{min}$, ami még rögzített méretű gyűrűben is tetszőlegesen nagy lehet. Ezért az időbonyolultság erősen korlátozza az algoritmus gyakorlati használhatóságát: csak kis elemszámú gyűrűs hálózatban és kis pozitív egész UID-ű folyamatok esetében praktikus.

3.5.2.. A VáltozóSebességek algoritmus

Az IdőSzelet algoritmus szerint abban az esetben, ha a folyamatok ismerik a gyűrű n méretét, $\mathcal{O}(n)$ üzenet elegendő a vezető folyamat kiválasztásához. De mi a helyzet akkor, ha n ismeretlen? Megmutatjuk, hogy ebben az esetben is létezik $\mathcal{O}(n)$ kommunikációs bonyolultságú algoritmus. Az általunk bemutatott algoritmust VáltozóSebességek algoritmusnak fogjuk nevezni. Az elnevezés rövid időn belül világos lesz. A VáltozóSebességek algoritmus is csak egyirányú kommunikációt használ.

Sajnos a VáltozóSebességek algoritmus időbonyolultsága még az IdőSzelet algoritmusénál is rosszabb, $\mathcal{O}(n \cdot 2^{u_{min}})$.

Világos, hogy senkinek sem jutna eszébe ezt az algoritmust a gyakorlatban alkalmazni. A VáltozóSebességek algoritmus az, amit mi *ellenpélda- algoritmusnak* hívunk. Az *ellenpélda algoritmus* egy olyan algoritmus, amelynek a fő célja annak megmutatása, hogy a megsejtett megoldhatatlansági eredmény hamis. Az ilyen algoritmus önmagában sem gyakorlati, sem matematikai szempontból nem érdekes és nem is különösebben elegáns. Annak megmutatására viszont kiválóan alkalmas, hogy a megoldhatatlansági eredményt nem lehet bebizonyítani. Íme az algoritmus:

> **VáltozóSebességek algoritmus (vázlatosan)**
>
> Minden P_i folyamat egy olyan jelsorozatot küld körbe a gyűrűn, ami a kiindulási folyamat u_i UID-jét tartalmazza. A különböző üzenetek különböző sebességgel haladnak. Egészen pontosan a v UID-et tartalmazó üzenet minden 2^v menetben egyetlen lépést halad előre. Vagyis minden folyamat, miután üzenetet kapott, vár 2^v menetet, és csak azután küld valamit tovább.
>
> Időközben minden folyamat figyelemmel követi az általa addig látott legkisebb UID-et és egyszerűen nem küldi tovább azokat az üzeneteket, amelyekben az UID nagyobb ennél az értéknél.
>
> Amikor egy üzenet visszaérkezik a feladójához, a folyamat önmagát vá-

lasztja vezetőnek.

Ahogy az IdőSzelet algoritmus, a VáltozóSebességek algoritmus is garantálja a minimális UID-ű folyamat kiválasztását.

Bonyolultságelemzés.. A VáltozóSebességek algoritmus garantálja, hogy a legkisebb UID-et tartalmazó üzenet (u_{min}) teljesen körbemegy a gyűrűn, a második legkisebb azonosítót tartalmazó üzenet legfeljebb a teljes út felét, a harmadik legkisebb UID legfeljebb a teljes út negyedét, általában pedig a k-adik legkisebb azonosítót tartalmazó üzenet legfeljebb a teljes út $1/(2^{k-1})$ részét teheti meg. Ezért a vezető kiválasztásának pillanatáig az u_{min} azonosítót tartalmazó üzenetek száma az összes többi üzenet együttes számánál is nagyobb. Mivel az u_{min} körbeérése pontosan n üzenetet jelent, a vezető folyamat kiválasztásáig az összes üzenetek száma kevesebb, mint $2n$.

Vegyük észre, hogy mialatt az u_{min} azonosítót tartalmazó üzenet körbemegy a gyűrűn, az összes érintett folyamat tudomást szerez erről az értékről, vagyis ők a későbbiekben már semmilyen üzenetet nem fognak küldeni. Ebből az következik, hogy $2n$ a *valaha elküldött üzenetek* számának egy felső becslése (magában foglalva a *vezető* kimenet megjelenése utáni időt is).

Ahogy korábban említettük, az algoritmus időbonyolultsága $n \cdot 2^{u_{min}}$, mert minden folyamat $2^{u_{min}}$ időegységgel késlelteti az u_{min} UID-et tartalmazó üzenet továbbküldését.

Különböző kezdési időpontok.. Az LCR algoritmustól és a HS algoritmustól eltérően az IdőSzelet algoritmus nem használható minden további nélkül a szinkron modell változatban különböző kezdési időpontok mellett. Az algoritmus egy módosított változata azonban már igen.

MódVáltozóSebességek algoritmus (vázlatosan)

Egy folyamatot nevezzünk *induló* folyamatnak, ha pontosan egy menettel korábban kapott egy *ébresztő* üzenetet, vagyis bármilyen közönséges (nemnull) üzenetet.

Minden P_i *induló* folyamat a saját u_i UID-jét tartalmazó üzenetet küldi körbe a gyűrűn. A nem *induló* folyamatok sohasem küldenek jelet. Kezdetben ez a jel „gyorsan" halad, egy átvitel/menet sebességgel. Az üzenetet fogadó „nem-induló" folyamat „felébred", ha pedig már *induló* folyamat volt (lehetett egy másik *induló* folyamat, vagy P_i önmaga), akkor az üzenet lelassulva folytatja útját, minden 2^{u_i} menetben egy átvitel sebességgel.

Időközben minden folyamat feljegyzi az *induló* folyamatok közül a minimális UID-űt és eldob minden olyan üzenetet, amiben az UID nagyobb, mint ez a legkisebb érték. Amikor egy üzenet visszaérkezik a feladójához, a folyamat önmagát választja vezetőnek.

A MódVáltozóSebességek algoritmus biztosítja a minimális UID értékkel rendelkező *induló* folyamat vezetővé választását. Jelölje ezen folyamat indexét i_{minind}.

Bonyolultságelemzés.. Az üzeneteket három csoportba oszthatjuk.

1. A kezdeti gyorsan továbbításra kerülő üzenetek. Ezekből pontosan n darab van.
2. A lassan továbbítandó üzenetek addig a pillanatig, amíg az i_{minind} üzenet először ér el valamely *induló* folyamatot. Ez az első felébredő folyamat idejétől kezdve legfeljebb n menet. Ezen idő alatt a v UID-del rendelkező üzenetek legfeljebb $n/2^v$-szer fordulhatnak elő, az üzenetek teljes összege így legfeljebb $\sum_{v=1}^{n} n/2^v < n$.
3. A lassan továbbítandó üzenetek attól a pillanattól kezdve, amikor az i_{minind} üzenet először ér el valamely *induló* folyamatot. Innentől elemzésünk hasonló a VÁLTOZÓSEBESSÉGEK algoritmusnál látottakhoz. Amíg a győztes jelsorozat végighalad a gyűrűn, a k-adik legkisebb *induló* folyamat azonosítóját tartalmazó üzenet legfeljebb a teljes út $1/(2^{k-1})$ részét teheti meg. Ezért az üzenetek számának teljes összege a vezető folyamat kiválasztásának pillanatáig kevesebb, mint $2n$. De amíg a győztes jelsorozat végighalad a gyűrűn, minden folyamat megismervén annak minimális UID értékét, a továbbiakban semmilyen üzenetet nem küld. Vagyis ebben az esetben az üzenetek számának egy felső korlátja $2n$.

Azt kaptuk, hogy a teljes kommunikációs bonyolultság legfeljebb $4n$.

Az időbonyolultság $n + n \cdot 2^{u_{minind}}$.

3.6.. Alsó korlát az összehasonlítás-alapú algoritmusokra

Az eddigiekben különféle algoritmusokat láttunk vezető folyamat kiválasztására szinkron gyűrűben. Az LCR algoritmus és a HS algoritmus összehasonlítás-alapú algoritmusok voltak, az utóbbi $\mathcal{O}(n \log n)$ kommunikációs bonyolultsággal és $\mathcal{O}(n)$ időbonyolultsággal. Ezzel szemben az IDŐSZELET algoritmus és a VÁLTOZÓSEBESSÉGEK algoritmusok nem összehasonlítás-alapú algoritmusok voltak $\mathcal{O}(n)$ kommunikációs és nagyon nagy időbonyolultsággal. Ebben az alfejezetben az üzenetek számának egy $\Omega(n \log n)$ alsó korlátját bizonyítjuk az összehasonlítás-alapú algoritmusokra. Ez az alsó korlát akkor is érvényben marad, ha a gyűrűben kétirányú kommunikációt engedélyezünk és ha a folyamatok ismerik a gyűrű n méretét. A következő alfejezetben hasonló alsó korlátot mutatunk a nem összehasonlítás-alapú algoritmusokra abban az esetben, ha az időbonyolultság korlátos.

Ezen alfejezet eredménye a *szimmetria megtörésének* nehézségén alapszik. Emlékezzünk vissza 3.1. tétel megoldhatatlansági eredményére: a szimmetria, a megkülönböztetett információ (mint például az UID-ek) hiánya miatt nem lehetséges a vezető folyamat kiválasztása. Az alábbi érvelés legfontosabb eleme az, hogy bizonyos mennyiségű szimmetria még az (egyedi) UID-ek jelenléte mellett is előfordulhat. Ebben az esetben az UID-ek lehetővé teszik a szimmetria megtörését, de ez hatalmas mennyiségű kommunikációt vehet igénybe.

Idézzük fel az egész fejezetre vonatkozólag tett korábbi megjegyzésünket, mi-

szerint a gyűrű folyamatai azonosak, kivéve az UID-eket. Vagyis a folyamatok kezdőállapotai azonosak, kivéve azon összetevőket, amelyek az UID-eket tartalmazzák. Általában nem tettünk semmilyen megszorítást arra nézve, hogy az üzenetgeneráló és állapotátmeneti függvények az UID-eket hogyan használhatják.

A fejezet hátralévő részében (ezen alfejezet és a következő) feltételezzük, hogy csak egyetlen esetében létezik, ami minden egyes UID-et tartalmaz. (Ahogy a 3.1. tétel bizonyításánál, a feltevés most sem csorbítja az általánosságot.) A feltételezés előnye, hogy következményeként (az UID-ek rögzített értékei mellett) a rendszernek pontosan egy végrehajtása létezik.

Az összehasonlítás-alapú algoritmusok megengednek bizonyos megszorításokat, amelyek az alábbi, kissé vázlatos meghatározás segítségével is kifejezhetők. Egy UID alapú gyűrű algoritmus *összehasonlítás alapú*, ha az UID-ek kezelésének kizárólagos módjai a másolás, az üzenetekben való részvétel (küldés, fogadás) és az összehasonlítás $(<, >, =)$ lehetnek.

A definíció megengedi a folyamatoknak az algoritmus végrehajtása során korábban előfordult UID-ek bármelyikének tárolását. Megengedi továbbá, hogy esetleg más információval együtt a folyamatok továbbküldjék őket. A folyamatok összehasonlíthatják a tárolt UID-eket és az összehasonlítások eredményeit felhasználhatják az üzenetgeneráló és állapotátmeneti függvények választásainál. Ezek a választások magukban foglalhatnak olyanokat is, mint például küldjön vagy ne küldjön üzenetet a szomszédainak, válassza vagy ne önmagát vezetőnek, tárolja vagy ne tárolja tovább az UID-ek valamelyikét stb. A lényeg az, hogy a folyamatok bármely tevékenysége csak az előforduló UID-ek relatív rangsorától függ, nem pedig a pontos értékeitől.

Az alábbi formális fogalom az esetleg fennálló, UID-eket is tartalmazó szimmetriát írja le. Legyen $U = (u_1, u_2, \ldots, u_k)$ és $V = (v_1, v_2, \ldots, v_k)$ UID-ek két k hosszúságú sorozata. Azt mondjuk, hogy az U *sorrendekvivalens* a V-vel, ha minden i, j esetében $(1 \leq i, j \leq k)$ $u_i \leq u_j$ akkor és csak akkor teljesül, ha $v_i \leq v_j$.

3.6.1. példa. Sorrendekvivalencia

> Az $(5, 3, 7, 0), (4, 2, 6, 1)$ és az $(5, 3, 6, 1)$ sorozatok sorrendekvivalensek, amennyiben az UID-ek halmaza a nemnegatív egész számok halmaza a szokásos rendezéssel.

Vegyük észre, hogy UID-ek két sorozata akkor és csak akkor sorrendekvivalens, ha a sorozatokban az UID-ek relatív sorrendje megegyezik. Most még két definíció következik. Azt mondjuk, hogy egy kiértékelés egy menete *aktív*, ha benne legalább egy (nem-null) üzenetet küldünk. Az n méretű R gyűrű valamely P_i folyamatának *k-szomszédságán* alatt a $P_{i-k}, \ldots, P_{i+k}$ $(2k+1$ darab) folyamatokat értjük $(0 \leq k < \lfloor n/2 \rfloor)$. Ezek pontosan azok a folyamatok, amelyek a P_i folyamattól legfeljebb k távolságra vannak és magát a P_i folyamatot is beleértjük.

Végezetül szükségünk lesz egy olyan meghatározásra, ami a folyamatok állapotainak azonosságát írja le (megengedve a folyamatok UID-jeinek különbözőségét). Azt mondjuk, hogy az s és t folyamatállapotok az $U = (u_1, u_2, \ldots, u_k)$

és $V = (v_1, v_2, \ldots, v_k)$ sorozatora vonatkozóan *összhangban vannak*, ha az alábbiak teljesülnek: s minden UID-je az U sorozatból való, t minden UID-je a V sorozatból való, valamint s és t megegyeznek, kivéve, hogy az s-beli u_i minden előfordulása helyett a t-beli v_i kerül minden $1 \leq i \leq k$-ra. Az *összhangban lévő üzenetek* ugyanígy adhatók meg.

Most már készen állunk az alsó korlát bizonyítására. Az alábbi lemma azt állítja, hogy a sorrendekvivalens k-szomszédságban lévő folyamatok lényegében ugyanúgy viselkednek, és ezt a viselkedésbeli azonosságot csak valamilyen, a k-szomszédságon kívülről érkező üzenet képes megtörni.

3.5. lemma. . *Legyen A egy n méretű R gyűrűben működő összehasonlítás-alapú algoritmus és legyen k egy egész szám, $0 \leq k < \lfloor n/2 \rfloor$. Legyen továbbá P_i és P_j két olyan folyamat A-ban, amelyekre a k-szomszédságaik UID-sorozatai sorrendekvivalensek. Ekkor legfeljebb k aktív menet utáni tetszőleges pillanatban a P_i és P_j folyamatok állapotai k-szomszédságaik UID-sorozataira vonatkozóan összhangban vannak.*

3.6.2. példa. Összhangban lévő állapotok

Legyen a P_i folyamat 3-szomszédságú UID-sorozata $(1, 6, 3, 8, 4, 10, 7)$ (ahol a P_i folyamat UID-ja 8), a P_j folyamat 3-szomszédságú UID-sorozata pedig $(4, 10, 7, 12, 9, 13, 11)$ (ahol a P_j folyamat UID-je 12). Mivel a két sorozat sorrendekvivalens, az iménti lemmából következik, hogy a P_i és P_j folyamatok állapotai legfeljebb három aktív meneten keresztül 3-szomszédságaik UID sorozataira vonatkozóan biztosan összhangban maradnak. Durván fogalmazva az indok az, hogy amennyiben csak három aktív menet történik, a sorrendekvivalens 3-szomszédságon kívüli üzeneteknek nincs lehetőségük P_i és P_j elérésére.

3.5. lemma bizonyítása. Az általánosság megszorítása nélkül feltehető, hogy $i \neq j$. A bizonyítás a menetek r száma szerinti indukcióval történik. Minden egyes r esetére a lemmát az összes lehetséges k-ra bebizonyítjuk.

Induló eset. $r = 0$. Az összehasonlítás-alapú algoritmusok definíciójának megfelelően P_i és P_j az UID-jeiktől eltekintve megegyeznek, ezért k-szomszédságaikra vonatkozóan minden k-ra összhangban vannak.

Indukciós feltevés. Tegyük fel, hogy a lemma állítása minden $r' < r$ menetre teljesül. Rögzítsük k-t oly módon, hogy P_i és P_j k-szomszédságai sorrendekvivalensek legyenek és tegyük fel, hogy az első r menet legfeljebb k aktív menetet tartalmaz.

Ha az r menetben sem P_i, sem P_j nem kap üzenetet, az indukciós feltevés miatt P_i és P_j állapotai k-szomszédságaikra vonatkozóan összhangban lesznek, hiszen az $r-1$ menetben is abban voltak, és új üzenet híján az állapotátmenetek összhangban maradnak.

Vagyis feltehető, hogy az r menetben P_i vagy P_j új üzenetet kap. Ekkor az r menet aktív, tehát az első $r-1$ menet legfeljebb $k-1$ aktív menetet tartalmazott.

Figyeljük meg, hogy P_i és P_j állapotainak $(k-1)$-szomszédságai sorrendekvivalensek, ami P_{i-1} és P_{j-1}, valamint P_{i+1} és P_{j+1} állapotaira is igaz. Ezért az $r-1$ menet után az indukciós feltevés miatt ($r-1$-re és $k-1$-re) P_i és P_j állapotai k-szomszédságaikra vonatkozóan szintén összhangban lesznek. Hasonló állítás igazolható P_{i-1} és P_{j-1}, valamint P_{i+1} és P_{j+1} állapotaira.

A bizonyítást az alábbi esetek vizsgálataival folytatjuk.

1. Az r menetben sem P_{i-1}, sem P_{i+1} nem küld P_i-nek üzenetet.
   Ekkor, mivel P_{i-1} és P_{j-1} állapotai $r-1$ menet után összhangban vannak és ugyanez igaz P_{i+1} és P_{j+1} állapotaira, azt kapjuk, hogy az r-edik menetben sem P_{j-1}, sem P_{j+1} nem küld P_j-nek üzenetet. De ez ellentmond azon feltételezésünknek, hogy P_i vagy P_j az r-edik menetben új üzenetet kap.
2. Az r menetben P_{i-1} üzenetet küld P_i-nek, de P_{i+1} nem.
   Ekkor, mivel P_{i-1} és P_{j-1} állapotai $r-1$ menet után összhangban voltak, az r-edik menetben P_{j-1} szintén küld üzenetet P_j-nek, és ez az üzenet megfelel a P_{i-1} és P_{j-1} folyamatok állapotai $(k-1)$-szomszédságainak, ezért P_i és P_j állapotai k-szomszédságaira vonatkozóan is összhangban lesznek azzal az üzenettel, amit P_{i-1} küld P_i-nek. Hasonló indoklással az r-edik menetben P_{j+1} sem küld üzenetet P_j-nek. Mivel P_i és P_j állapotai $r-1$ menet után összhangban voltak és összhangban lévő üzeneteket kaptak, állapotaik k-szomszédságaira vonatkozóan is összhangban maradnak.
3. Az r menetben P_{i+1} üzenetet küld P_i-nek, de P_{i-1} nem.
   Ez az eset az előzőhöz hasonló módon látható be.
4. Az r menetben P_{i-1} is és P_{i+1} is küld üzenetet P_i-nek.
   Az indoklás itt is a korábbihoz hasonló.

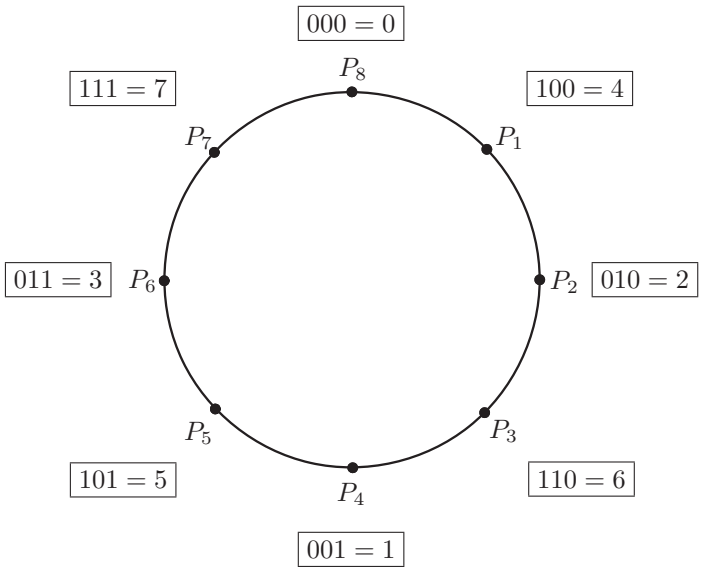
□

A 3.5. lemma azt mutatja, hogy elég sok aktív menet szükséges a szimmetria megtöréséhez, ha nagy sorrendekvivalens szomszédságok léteznek. Most egy olyan különleges tulajdonságokkal rendelkező gyűrűt írunk le, amelyben sok, különböző méretű sorrendekvivalens szomszédság található. Legyen c egy adott konstans $(0 \leq c \leq 1)$ és legyen az R gyűrű mérete n. Azt mondjuk, hogy az R gyűrű *c-szimmetrikus*, ha minden l-re $(\sqrt{n} \leq l \leq n)$ és R minden l hosszú S szegmenséhez létezik legalább $\lfloor cn/l \rfloor$ olyan szegmens R-ben, amely sorrendekvivalens S-sel (önmagát S-et is beleszámolva). Megjegyezzük, hogy az alsó korlátnál a négyzetgyökös feltétel elhagyható.

Ha n kettőhatvány, akkor egy $\frac{1}{2}$-szimmetrikus gyűrűt könnyű létrehozni. A következőkben az n méretű *bitfordító gyűrűt* határozzuk meg. Legyen $n = 2^k$. Ekkor minden P_i folyamathoz azt a $[0, n-1]$ intervallumból való egészet rendeljük hozzá, amelynek a k-bites bináris ábrázolása a fordítottja az i indexű folyamatban az i érték k-bites bináris ábrázolásának (ahol 0^k jelöli a 0 folyamat k-bites bináris ábrázolását).

3.6.3. példa. Bitfordító gyűrű.

$n = 8$ esetében $k = 3$ és a megfeleltetések a 3.3. ábrán láthatók.



3.3.. ábra. Egy 8 méretű bitfordító gyűrű.

3.6. lemma. . *Minden bitfordító gyűrű $\frac{1}{2}$-szimmetrikus.*

Bizonyítás. Azzal a megjegyzéssel, hogy a bitfordító gyűrűk esetében nincs szükség a négyzetgyököt tartalmazó alsó korlátra, a bizonyítást az Olvasóra bízzuk. □

A nem kettőhatvány n értékekre is léteznek c-szimmetrikus gyűrűk, de az általános eset kisebb c konstanst igényel.

3.7. tétel. . *Létezik olyan c konstans, hogy minden $n \in \mathbb{N}$ esetben megadható egy n méretű c-szimmetrikus gyűrű.*

A 3.7. tétel bizonyítása egy meglehetősen bonyolult rekurzión alapul (itt használnánk ki maradéktalanul a négyzetgyökös alsó korlátot). A keresett gyűrű előállítása nem könnyű feladat. Nem mondhatjuk egyszerűen azt, hogy az n-nél kisebb legnagyobb kettőhatvány méretű bitfordító gyűrűhöz hozzáveszünk néhány folyamatot. A hozzávett folyamatok ugyanis elrontják a szimmetriát.

Vagyis feltételezzük, hogy minden n-re létezik egy n méretű c-szimmetrikus R gyűrű. A következő lemma azt állítja, hogy ha egy ilyen gyűrű kiválasztja a vezető folyamatot, akkor az eljárásban sok aktív menet hajtódik végre.

3.8. lemma. . *Legyen A egy öszehasonlításon alapú algoritmus valamely n méretű c-szimmetrikus gyűrűben és tegyük fel, hogy A kiválasztja a vezető folyamatot. Tegyük fel továbbá, hogy k egy olyan egész, amelyre $\sqrt{n} \leq 2k+1$ és $\lfloor cn/(2k+1) \rfloor \geq 2$. Ekkor A több mint k aktív menetet hajt végre.*

Bizonyítás. Indirekt módon bizonyítunk. Tegyük fel, hogy A legfeljebb k aktív menetben választja ki a vezető P_i folyamatot. Legyen S a P_i folyamat k-szomszédsága. Ekkor S egy $2k+1$ hosszúságú szegmens. Mivel a gyűrű c-szimetrikus, a gyűrűben léteznie kell legalább $\lfloor cn/(2k+1) \rfloor \geq 2$ darab S-sel sorrendekvivalens szegmensnek, S-et önmagát is beleértve. Vagyis létezik legalább még egy S-sel sorrendekvivalens szegmens. Legyen ennek a szegmensnek a közepe P_j. De ekkor a 3.5. lemma miatt a végrehajtás alatt P_i és P_j a vezető folyamat kiválasztásának pillanatáig ekvivalens állapotokban maradnak. Azt kapjuk, hogy a P_j folyamat szintén vezető folyamat lesz. Ellentmondásra jutottunk. □

Most bebizonyítjuk az alsó korlátot.

3.9. tétel. . *Legyen A egy olyan öszehasonlításon alapú algoritmus, amely bármely n méretű gyűrűben kiválasztja a vezető folyamatot. Ekkor A-nak van olyan végrehajtási sorozata, amelyben a vezető folyamat kiválasztásának pillanatáig $\Omega(n \log n)$ üzenetetküldés történik.*

Megjegyezzük, hogy az $\Omega(n \log n)$ kifejezésben egy n-től független konstans szorzó is szerepel.

Bizonyítás. Legyen c egy olyan rögzített konstans, amelynek létezését a 3.7. tétel garantálja és vegyük az ennek megfelelő n méretű c-szimmetrikus R gyűrűt. Most az A algoritmus R gyűrűbeli végrehajtási sorozatait fogjuk vizsgálni.

Legyen $k = \lfloor (cn-2)/4 \rfloor$. Ekkor $\sqrt{n} \leq 2k+1$ (feltéve, hogy n elég nagy) és $\lfloor cn/(2k+1) \rfloor \geq 2$. A 3.8. lemmából következik, hogy az algoritmus több, mint k (vagyis legalább $k+1$) aktív menetet hajt végre.

Vizsgáljuk meg az r-edik aktív menetet, ahol $\sqrt{n}+1 \leq r \leq k+1$. Mivel a menet aktív, létezik olyan P_i folyamat, ami az r-edik menetben üzenetet küld. Legyen S a P_i folyamat $(r-1)$-szomszédsága. Az R gyűrű c-szimmetrikus volta miatt létezik legalább $\lfloor cn/(2r-1) \rfloor$ darab S-sel ekvivalens szegmens R-ben. De a 3.5. lemma miatt az r-edik aktív menet előtti pillanatig ezen szegmensek középpontjai állapotai összhangban voltak, vagyis mindegyikük küldött üzenetet.

Legyen most $r_1 = \lceil \sqrt{n} \rceil + 1$ és $r_2 = \lfloor (cn-2)/4 \rfloor + 1$. Az iménti gondolatmenetből következik, hogy az üzenetek összes száma legalább

$$\sum_{r=r_1}^{r_2} \left\lfloor \frac{cn}{2r-1} \right\rfloor \geq \sum_{r=r_1}^{r_2} \frac{cn}{2r-1} - r_2.$$

Az összeg második tagja $\mathcal{O}(n)$ nagyságrendű, vagyis elegendő azt belátni, hogy az első tag nagyságrendje $\Omega(n \log n)$. A kifejezést átalakítva azt kapjuk, hogy

$$\sum_{r=r_1}^{r_2} \frac{cn}{2r-1} = \Omega\left(n \sum_{r=r_1}^{r_2} \frac{1}{r} \right) = \Omega(n(\ln r_2 - \ln r_1)) =$$

$$= \Omega\left(n \left(\ln \left(\left\lfloor \frac{cn-2}{4} \right\rfloor + 1 \right) - \ln(\lceil \sqrt{n} \rceil + 1) \right) \right) = \Omega(n \log n).$$

A második sorban az összeg integrálközelítését használtuk. A bizonyítást befejeztük. □

3.7.. Alsó korlát a nem összehasonlítás-alapú algoritmusok üzeneteinek számára*

Vajon a nem összehasonlítás-alapú algoritmusok esetében az üzenetek számára vonatkozóan milyen alsó korlát adható? Bár az $\Omega(n \log n)$ korlát ebben az esetben nem érvényes, megmutatható, hogy ez a korlát csak nagy időbonyolultság árán léphető át. Tegyük fel, hogy a vezető folyamat kiválasztásáig eltelt idő t-vel felülről korlátos. Ha az azonosítók állapotterében az UID-ek összes száma elég nagy – mondjuk nagyobb, mint valamilyen gyorsan növő $f(n,t)$ függvény –, akkor az azonosítóknak létezik egy olyan U részhalmaza, amelyen meg lehet mutatni, hogy az algoritmus legalább t meneten keresztül „összehasonlítás-alapú algoritmusként" viselkedik. Ebből az következik, hogy az összehasonlítások alsó korlátja átvihető az U-beli azonosítókat használó időkorlátos algoritmusokra.

Gondolatunkat ennél részletesebben is kifejtjük, de leírásunk így is nagyon vázlatos marad. A *Ramsey-tétel* segítségével – ami a skatulya-elv egyfajta általánosítása – meg fogjuk határozni az imént említett gyorsan növő $f(n,t)$ függvényt. A tétel állításában n-részhalmazon egy n elemű részhalmazt értünk, színezésen pedig minden halmazhoz valamilyen szín hozzárendelését.

3.10. tétel. . (Ramsey-tétel) *Minden m, n és c egészhez található egy $g(n,m,c)$ egész az alábbi tulajdonsággal. Minden legalább $g(n,m,c)$ elemű S halmazhoz és az S halmaz n-részhalmazainak bármilyen legfeljebb c színt tartalmazó színezéséhez létezik S-nek olyan m elemű C részhalmaza, amelynek az összes n-részhalmaza ugyanazzal a színnel színezett.*

Először az algoritmusokat olyan *normálformára* hozzuk, amelyben minden állapot a LISP nyelv S-kifejezéseinek formátumában rögzíti a kezdeti UID-eket, az összes valaha érkezett üzenetet és minden *nem-null* üzenet tartalmazza küldőjének teljes állapotleírását. Ezen S-kifejezések közül bizonyosak *kiválasztott* állapotokat jelölnek, amelyekben a folyamat vezető folyamatként azonosítható. Ha az eredeti algoritmus egy megfelelő vezetőfolyamat-kiválasztó algoritmus volt, az új (a módosított kimenetre vonatkozó megállapodásokkal) úgyszintén az lesz, és a kommunikációs bonyolultságaik megegyeznek.

Az alsó korlátra vonatkozó tételünk az alábbit állítja.

3.11. tétel. . *Minden n és t egészhez létezik egy $f(n,t)$ egész az alábbi tulajdonsággal. Legyen A egy tetszőleges (nem szükségképpen összehasonlítás-alapú) algoritmus, ami az n méretű gyűrűben t időn belül kiválasztja a vezető folyamatot úgy, hogy az UID-ek tere legalább $f(n,t)$ méretű. Ekkor A-nak létezik olyan végrehajtási sorozata, amelyben a vezető folyamat kiválasztásáig $\Omega(n \log n)$ üzenetküldés történik.*

Bizonyításvázlat. Legyen n és t rögzített. Az általánosság megszorítása nélkül feltehető, hogy az algoritmusok normálformában vannak. Mivel az algoritmusok n darab folyamaton futnak és t menetidőt használnak, minden előforduló S-kifejezés legfeljebb n különböző argumentumot és legfeljebb t mélységű egymásba ágyazást tartalmaz.

Az UID-ek n-halmazain (n elemű halmazokon) minden A algoritmusra megadunk egy $\equiv_A$ ekvivalenciarelációt. Két n-halmaz akkor lesz ekvivalens, ha az A algoritmus ugyanúgy viselkedik rajtuk. Formálisan, legyen V és V' UID-ek két n-halmaza. Azt mondjuk, hogy $V \equiv_A V'$, ha az A algoritmus minden V feletti legfeljebb t mélységű egymásba ágyazást tartalmazó S-kifejezés és a neki megfelelő V' feletti S-kifejezés esetében (amit úgy kapunk, hogy V minden elemét a sorrendben ugyanannyiadik V'-belire cserélünk) ugyanarra a döntésre jut: küldjön vagy ne küldjön üzenetet, a folyamat válassza vagy ne önmagát vezetőnek.

Mivel az ekvivalenciareláció definíciójában az S-kifejezéseknek legfeljebb n argumentumuk van és legfeljebb t mélységű egymásba ágyazást tartalmaznak, csak véges sok ekvivalenciaosztály létezik. Valójában az osztályok számára létezik egy, az A algoritmustól független, csak n-től és t-től függő felső korlát. Jelöljük ezt a korlátot $c(n,t)$-vel.

Rögzítsük az A algoritmust. Mivel alkalmazni szeretnénk a Ramsey-tételt, először egy módszert kell adnunk az UID-ek n-halmazainak színezésére. Feleltessünk meg egy színt az n-halmazok minden $\equiv_A$ ekvivalenciaosztályának és egy osztályon belül minden n-halmazt színezzünk ugyanazzal a színnel.

Legyen $f(n,t) = g(n, 2n, c(n,t))$, ahol g a 3.10. tételben szereplő függvény és vegyük az UID-ek egy tetszőleges, legalább $f(n,t)$ azonosítót tartalmazó terét. Ekkor a 3.10. tétel miatt létezik az UID-ek terének egy olyan, legalább $2n$ elemet tartalmazó C részhalmaza, amelynek minden n-részhalmaza ugyanazzal a színnel színezett. A C halmaz n legkisebb elemeiből álló halmazt jelöljük U-val.

Most azt állítjuk, hogy ha az UID-eket az U halmazból választjuk, az algoritmus t meneten keresztül ugyanúgy viselkedik, mint egy összehasonlítás-alapú algoritmus. Vagyis bármely folyamat bármilyen döntést is hoz arról, hogy küldjön-e üzenetet valamely irányba vagy a folyamat vezetőnek titulálja-e önmagát, a döntés csak az aktuális állapot argumentumainak relatív sorrendjétől függ. Ennek belátásához rögzítsük U-nak bármely két, mondjuk m elemű W és W' részhalmazát. Tegyük fel, hogy S egy legfeljebb t egymásba ágyazást és W-beli UID-eket tartalmazó S-kifejezés és S' a neki megfelelő W' feletti S-kifejezés (amit úgy kapunk, hogy W minden elemét a sorrendben ugyanannyiadik W'-belire cserélünk). Ekkor W és W' az n elemű V és V' halmazokká bővíthető úgy, hogy mindegyikükhöz hozzávesszük a C halmaz legnagyobb $n-m$ elemét. Mivel V és V' színezése megegyezik, a két S-kifejezés ugyanolyan döntést eredményez arról, hogy küldjenek-e üzenetet a szomszédaik felé vagy a folyamatok vezetőnek választják-e magukat.

Mivel az algoritmus t meneten keresztül pontosan úgy viselkedik, mint egy összehasonlítás-alapú algoritmus (amennyiben az UID-eket az U halmazból választjuk), a 3.9. tételben bizonyított alsó korlát érvényben marad. □

3.8.. Megjegyzések a fejezethez

A 3.2. fejezet megoldhatatlansági eredménye a terület vizsgálatának legelejére nyúlik vissza. Az eredmény egy más modellre alkalmazott változata Angluin [13] cikkében található meg. Az LCR algoritmus Le Lann [191] egy fejlesztéséből

származik, amit Chang és Roberts optimalizált [71]. A HS algoritmus Hirschberg és Sinclair munkája [156].

Az $\mathcal{O}(n \log n)$ felső korlátjának konstans értékét folyamatosan javították, a jelenlegi érték $1,271n \log n + O(n)$ Higham és Przytycka [155] eredménye. A korlát egyirányú gyűrűkre is igaz. Peterson [239], valamint Dolev, Klawe és Rodeh [97] $\mathcal{O}(n \log n)$-es algoritmust adtak az egyirányú esetre.

Az IdőSzelet algoritmus szintén ősi darab, amelynél a vezető folyamat kiválasztásának stratégiája hasonlít az MIT vezérlőjeles gyűrűs hálózatban használatoshoz. A VáltozóSebességek algoritmus Frederickson és Lynch [127], valamint tőlük függetlenül Vitanyi [282] fejlesztése.

Az összehasonlítás-alapú és a nem összehasonlítás-alapú algoritmusokra vonatkozó alsó korlátok Frederickson és Lynch [127] eredményei. Attiya, Snir és Warmuth [27] a c-szimmetrikus gyűrűk másféle előállítását adták. A Ramsey-tétel a kombinatorika jól ismert eredménye, megtalálható például Berge [47] gráfelmélet könyvében.

Attiya, Snir és Warmuth [27] cikke más eredményeket is tartalmaz a szinkron gyűrűkben való számítások korlátairól. Az általuk használt bizonyítási módszerek a 3.6. fejezetben látottakhoz hasonlatosak.

3.9.. Gyakorlatok

3-1. Finomítsuk az LCR algoritmus helyességét bizonyító induktív bizonyítás részleteit.

3-2. Tekintsük az LCR algoritmust.

(a) Adjunk olyan UID-felsorolást, ahol az elküldött üzenetek száma $\Omega(n^2)$.

(b) Adjunk olyan UID-felsorolást, ahol az elküldött üzenetek száma $\mathcal{O}(n)$.

(c) Mutassuk meg, hogy az elküldött üzenetek átlagos száma $\mathcal{O}(n \log n)$, ahol az átlagot a gyűrű folyamatainak összes lehetséges egyformán valószínű elrendezésén való működésből számoljuk.

3-3. Módosítsuk az LCR algoritmust úgy, hogy az összes nemvezető folyamat a *nem_vezető* kimenetet eredményezze, vagyis az összes folyamat végül is álljon meg. Adjuk meg a módosított algoritmust az LCR algoritmusnál látott kódolási stílus segítségével.

3-4. Mutassuk meg, hogy az LCR algoritmus a szinkron modellben különböző induló időpontok mellett is helyesen működik. (Ehhez egy kicsit módosítsuk a kódot.)

3-5. Az LCR algoritmusnál látott invariáns állítások módszerével bizonyítsuk be a HS algoritmus helyességét.

3-6. Mutassuk meg, hogy a HS algoritmus a szinkron modellben különböző induló időpontok mellett is helyesen működik. (Ehhez egy kicsit módosítsuk a kódot.)

3-7. Tegyük fel, hogy a HS algoritmust úgy módosítjuk, hogy kettőhatványok helyett egymás utáni k-hatványokat használunk az utak hosszára ($k > 2$). Elemezzük a módosított algoritmus idő és kommunikációs bonyolultságát úgy, ahogy tettük ezt az eredeti HS algoritmusnál. Hasonlítsuk össze az eredményeket.

3-8. Vegyük a HS algoritmus olyan módosított változatát, ahol a folyamatok mindkét irány helyett csak az egyik irányba küldhetnek üzeneteket.

(a) Mutassuk meg, hogy a könyvben megadott algoritmus legkézenfekvőbb módosítása nem eredményez $\mathcal{O}(n \log n)$ kommunikációs bonyolultságot. Mi lesz a kommunikációs bonyolultság egy felső korlátja?

(b) Egy kis ravaszkodással módosítsuk úgy az algoritmust, hogy kommunikációs bonyolultsága ismét $\mathcal{O}(n \log n)$ legyen.

3-9. Tervezzünk egyirányú gyűrűben olyan vezetőfolyamat-választásos algoritmust, ami nem ismeri a gyűrű méretét és legrosszabb esetben is csak $\mathcal{O}(n \log n)$ számú üzenetet használ. Az algoritmus az UID-ekre kizárólag az összehasonlítás műveletet használhatja.

3-10. Kódoljuk az IdőSzelet algoritmust az állapotgép segítségével.

3-11. Módosítsuk úgy az IdőSzelet algoritmust, hogy hozzávett üzenetek árán fázisonként egyetlen UID helyett k darab UID továbbküldésének engedélyezésével csökkenjen a futási idő. Bizonyítsuk be az algoritmus helyességét és elemezzük bonyolultságát.

3-12. Kódoljuk a VáltozóSebességek algoritmust az állapotgép segítségével.

3-13. Mutassuk meg, hogy ha a folyamatok különböző időpontokban ébredhetnek fel, a VáltozóSebességek algoritmus kommunikációs bonyolultsága nem szükségszerűen $\mathcal{O}(n)$.

3-14. Adjunk a menetek számára vonatkozó minél jobb *alsó* korlátot valamely n méretű gyűrű vezetőfolyamat-választásos algoritmusának legrosszabb esetére. A feltevéseket körültekintően fogalmazzuk meg.

3-15. Adjuk meg az $n = 16$ csomópontú bitfordító gyűrű pontos leírását.

3-16. Bizonyítsuk be, hogy az $n = 2^k$ méretű bitfordító gyűrű minden $k \in \mathbb{N}$ esetében $\frac{1}{2}$-szimmetrikus.

3-17. Tervezzünk c-szimmetrikus gyűrűt nem kettőhatvány számú csomópont esetében valamilyen $c > 0$ értékre.

3-18. Valamely szinkron gyűrű esetében vegyük a vezető folyamat kiválasztásának problémáját, ahol minden folyamat ismeri a gyűrű n méretét és a folyamatoknak nincs UID-jük. Adjunk a probléma megoldására *véletlenített algoritmust*, vagyis olyat, ahol a folyamatok kódjuk determinisztikus végrehajtási sorozatán kívül véletlen választással is élhetnek. A helyes működést kielégítő tulajdonságokat óvatosan fogalmazzuk meg. Például az egyedi vezető folyamat kiválasztása biztosan garantált-e vagy valamilyen kis valószínűséggel elképzelhető, hogy ez nem történik meg? Mennyi lesz az algoritmus időbonyolultsága és kommunikációs bonyolultsága?

3-19. Vegyünk valamilyen ismeretlen n méretű szinkron, mindkét irányban irányított gyűrűt, ahol a folyamatoknak van UID-jük. Adjunk az üzenetek számára vonatkozó alsó és felső korlátot olyan összehasonlítás-alapú algoritmus esetében, ahol minden folyamat mod 2 számolja ki n-et.

4. fejezet

Általános szinkron hálózatok algoritmusai

A 3. fejezetben nagyon egyszerű szinkron hálózatokban – egyirányú és kétirányú gyűrűkben – oldottuk meg a vezetőválasztás leegyszerűsített problémáit. Ebben a fejezetben a szinkron hálózatok egy bővebb osztályában egy nagyobb problémakörrel foglalkozunk. Nevezetesen, olyan algoritmusokról lesz szó, amelyek tetszőleges gráfokból vagy irányított gráfokból épített hálózatokban *vezető folyamatot* választanak, *szélességi keresést (SZK)* végeznek, *legrövidebb utakat*, *minimális feszítőfát (MFF)*, és *maximális független halmazt (MFH)* keresnek.

A vezetőválasztás feladata akkor vetődik fel, amikor egy hálózati számítás „vezérlését" kell rábízni egy folyamatra. A szélességi keresést, a legrövidebb utak és a minimális feszítőfa keresését az motiválja, hogy megfelelő kommunikációs szerkezeteket tudjunk építeni egy hatékony üzenetváltás biztosítása érdekében. A maximális független halmaz keresésének igénye a hálózati erőforrások hozzárendelésének feladatából adódik. (Később, a 15. fejezetben, az aszinkron hálózatok kapcsán újra találkozunk ezen feladatok és algoritmusok többségével.)

Ebben a fejezetben egy tetszőleges, erősen összefüggő, n darab csúcsot tartalmazó $G = (V, E)$ irányított hálózati gráffal dolgozunk majd. (Néha arra az esetre szorítkozunk, amikor az élek kétirányúak, azaz ilyenkor a gráf lényegében irányítatlan.) Most is feltesszük – mint ahogy azt a szinkron hálózatoknál megszokhattuk –, hogy a folyamatok csak a gráf irányított élei mentén érintkezhetnek egymással. A gráf csúcsaihoz az $1, \ldots, n$ indexeket rendeljük, de a gyűrűkkel ellentétben ez az indexelés egyáltalán nem utal a csúcsoknak a gráfban való elhelyezkedésére. A folyamatok nem ismerik sem saját, sem szomszédjaik indexét, de szomszédjaikra helyi nevekkel hivatkozhatnak. Feltesszük továbbá, hogy ha a P_i folyamatnak a P_j folyamat egyszerre kimenő és a bejövő szomszédja is, akkor ezt az egybeesést P_i felismeri.

4.1.. Vezetőválasztás általános hálózatokban

Idézzük fel először a vezetőválasztás problémáját, de azt most egy tetszőleges, erősen összefüggő irányított gráfban fogalmazzuk meg.

4.1.1.. A feladat

Tegyük fel, hogy minden folyamat rendelkezik egy egyedi azonosítóval, amit az azonosítók teljesen rendezett halmazából választunk ki; minden egyes folyamat azonosítója különbözik a hálózat más folyamatának azonosítójától, és semmiféle megkötés sincs arra nézve, hogy egy folyamat melyik azonosítóval rendelkezzen. Mint azt a 3. fejezetben is megköveteltük, végül pontosan egy folyamat választhatja magát vezetőnek azáltal, hogy az állapotának a *státus* összetevőjét *vezető* értékre állítja. Akárcsak a 3. fejezetben, itt is több változata van a feladatnak.

1. Megkövetelhetjük, hogy az összes nemvezető folyamat *nem_vezető* értékre állítsa be a *státus*-át.
2. A folyamatok ismerhetik a csúcsok számát (n) és a gráf átmérőjét ($átm$), vagy ezen értékeknek egy felső korlátját.

4.1.2.. Egy egyszerű terjedő algoritmus

Most egy olyan egyszerű algoritmust mutatunk be, amelynek segítségével mind a vezető, mind a nemvezető folyamatok azonosíthatják magukat. Ennek az algoritmusnak előfeltétele, hogy a folyamatok ismerjék az *átm* értékét. Az algoritmus működése során a maximális egyedi azonosító terjed el a hálózatban, ezért nevezzük MaxTerjed algoritmusnak.

MaxTerjed algoritmus (vázlatosan)
Minden folyamat azt a maximális egyedi azonosítót tartja nyilván, amely a végrehajtás adott pillanatáig eljutott hozzá (kezdetben ez a saját egyedi azonosítója). Mindegyik menetben minden egyes folyamat továbbítja ezt a maximális egyedi azonosítót a kimenő szomszédjainak. Ha *átm* darab menet után egy folyamat a saját egyedi azonosítójával azonos értékű maximális egyedi azonosítót tárol, vezetőnek választja magát; egyébként nemvezető lesz.

A P_i folyamat kódja tehát a következő.

MaxTerjed algoritmus (formálisan)
Az üzenet ábécéje az egyedi azonosítók halmaza.

állapotok$_i$ összetevői:
u egy azonosító, kezdeti értéke a P_i folyamat egyedi azonosítója
max_azon egy azonosító, kezdeti értéke a P_i folyamat egyedi azonosítója
státus $\in$ {*ismeretlen, vezető, nem_vezető*}, kezdetben az értéke *ismeretlen*
menetek egy egész szám, kezdetben az értéke 0

üzenetek$_i$:
if *menetek* $<$ *átm* **then**
  send *max_azon* az összes $j \in$ *ki_szom* folyamatnak

átmenet$_i$:
menetek := *menetek* + 1
legyen U a *be_szom*-beli folyamatoktól érkező egyedi azonosítók halmaza
max_azon := $\max(\{max\_azon\} \cup U)$
if *menetek* = *átm* **then**
  if *max_azon* = *u*
    then *státus* := *vezető*
    else *státus* := *nem_vezető*

Könnyű belátni, hogy a MaxTerjed algoritmus a maximális egyedi azonosítójú folyamatot választja vezetőnek. Jelöljük a maximális egyedi azonosítójú folyamat indexét i_{max}-szal, az egyedi azonosítóját u_{max}-szal. Be fogjuk bizonyítani az alábbi tételt.

4.1. tétel. . *A* MaxTerjed *algoritmus átm darab menetben az i_{max}-adik folyamatot vezetőnek, a többi folyamatot nemvezetőnek jelöli ki.*

Bizonyítás. Elég belátni a következő állítást.

4.1.1. állítás. *A átm darab menet után státus$_{i_{max}}$ = vezető, és minden $j \neq i_{max}$ indexre a státus$_j$ = nem_vezet.*

A 4.1.1. állítás bizonyításának kulcsa az, hogy r menet után a maximális egyedi azonosító minden olyan folyamathoz eljut, amelynek az i_{max}-adik folyamattól a G gráfbeli irányított utak mentén vett távolsága legfeljebb r. Ezt fogalmazza meg az alábbi invariáns.

4.1.2. állítás. *Tetszőleges $0 \leq r \leq$ átm egész szám és tetszőleges P_j folyamat esetén teljesül, hogy ha a P_j folyamat i_{max}-tól vett távolsága legfeljebb r, akkor max_azon$_j$ = u_{max} fennáll az r-edik menet után.*

A gráf átmérőjének meghatározása miatt a 4.1.2. állításból következik, hogy a maximális egyedi azonosító az *átm*-adik menet végére minden folyamathoz eljut. A 4.1.2. állítás bizonyításához hasznos segítséget adnak a következő segédinvariánsok.

4.1.3. állítás. *Tetszőleges r egész szám és tetszőleges P_j folyamat esetén r menet után menetek$_j$ = r.*

4.1.4. állítás. *Tetszőleges r egész szám és tetszőleges P_j folyamat esetén r menet után $max\_azon_j \leq u_{max}$.*

A 4.1.2., a 4.1.3. állításokból, és $r = átm-1$ esetén a 4.1.4. állításból, valamint annak megfontolásából, hogy mi történik az *átm*-adik menetben, következik a 4.1.1. állítás, és ennek megfelelően a 4.1. tétel is. □

A MaxTerjed algoritmus a 3.3. alfejezetben tárgyalt LCR algoritmusnak egyfajta általánosítása, hiszen az LCR algoritmus szintén a maximális értéket terjesztette el a (gyűrűs) hálózatban. Meg kell azonban jegyeznünk, hogy az LCR algoritmus nem követelte meg az átmérőnek, mint speciális hálózati tulajdonságnak az ismeretét. Az LCR algoritmusban akkor vált egy folyamat vezetővé, amikor a saját egyedi azonosítóját kapta meg üzenetként, nem pedig egy meghatározott számú menet után, mint ahogy azt a MaxTerjed algoritmusban láttuk. Az LCR stratégiája gyűrűs hálózatokban megfelelő volt, de nem működik általános irányított gráfokban.

Bonyolultságelemzés.. Könnyű belátni, hogy *átm* darab menet összideje szükséges ahhoz, hogy a vezető folyamatot kiválasszuk (és a többi folyamat megtanulja, hogy ő nem vezető). Az üzenetek száma $átm \cdot |E|$, ahol az $|E|$ az irányított gráf irányított éleinek száma, ugyanis minden menetben minden irányított él mentén el kell küldeni egy üzenetet.

Az átmérő felső korlátja.. Megjegyezzük, hogy az algoritmus akkor is helyesen működik, ha a folyamatok az átmérő helyett annak csak egy d felső korlátját ismerik. A futási idő ekkor nő, mivel az *átm* helyett a d-től függ.

4.1.3.. A kommunikációs bonyolultság csökkentése

Az egyszerű optimalizálás[1], amelyet most alkalmazni fogunk, számos esetben képes a kommunikációs bonyolultság javítására, habár a legrosszabb eset nagyságrendjét nem csökkenti. Nevezetesen arról van szó, hogy a folyamatok csak akkor küldik el a saját maximális egyedi azonosítójuk értékét, amikor annak először birtokába jutnak, és nem minden menetben. A MaxTerjed algoritmus ezen módosítását OptMaxTerjed algoritmusnak hívjuk, a kódja pedig a következő.

OptMaxTerjed algoritmus (formálisan)

állapotok$_i$ új összetevője:
új_info logikai értékű változó, kezdetben *igaz*

üzenetek$_i$:
if *menetek* $<$ *átm* és *új_info* $=$ *igaz* **then**
  send *max_azon* az összes $j \in ki\_szom$ folyamatnak

[1] Az „optimalizálás” nem igazán megfelelő szó erre. A „javítás” jobb lenne, de a szakirodalomban az „optimalizálás” használata a megszokott.

átmenet$_i$:
menetek := *menetek* + 1
legyen U a *be_szom*-beli folyamatoktól érkező egyedi azonosítók halmaza
if $max(U) > max\_azon$
  then *új_info* := *igaz*
  else *új_info* := *hamis*
$max\_azon := \max(\{max\_azon\} \cup U)$
if *menetek* = *átm* **then**
  if $max\_azon = u$
    then *státus* := *vezető*
    else *státus* := *nem_vezető*

Érezhető, hogy ez a módosítás helyes algoritmust eredményez. De hogyan bizonyíthatjuk be ezt formálisan? Az egyik módja ennek az, hogy a MaxTerjed algoritmus bizonyításánál alkalmazott utat követjük, csak más invariáns állításokra támaszkodunk. Ez azonban sok olyan munkát is magában foglal, amit a korábbi bizonyításokban már egyszer elvégeztünk. Ahelyett, hogy mindezt újra végigcsinálnánk, kínálkozik egy másik lehetőség is. Egy olyan bizonyítást mutatunk, amely az OptMaxTerjed algoritmus és a MaxTerjed algoritmus formális kapcsolatára támaszkodik. Ez egy egyszerű példája az osztott algoritmusok helyességét ellenőrző úgynevezett *szimulációs* módszernek.

4.2. tétel. . *Az* OptMaxTerjed *algoritmus átm darab menetben az* i_{max}*-adik folyamatot vezetőnek, minden más folyamatot nemvezetőnek jelöl ki.*

Bizonyítás. Elég a 4.1.1. állításhoz hasonlóan bebizonyítani az alábbit.

4.1.5. állítás. *átm darab menet után* $státus_{i_{max}}$ = *vezető, és minden* $j \neq i_{max}$ *indexre* $státus_j$ = *nem_vezető.*

Kezdjük egy olyan invariáns bizonyításával, amely azt mondja ki, hogy egy folyamat *új_info* jelzője mindig *igaz* értékre áll be, valahányszor van olyan új információ, amit a folyamatnak a következő menetben tovább kell küldenie. Még pontosabban, ha a P_i folyamatnak valamelyik kivezető szomszédja a P_i folyamat maximális egyedi azonosítójánál kisebb azonosítót ismer, a P_i folyamat *új_info* jelzőjét *igaz*-ra kell állítani.

4.1.6. állítás. *Minden olyan* i, j *indexű folyamatra, ahol* $j \in ki\_szom_i$, $r(0 \leq r \leq átm)$ *menet után fennáll, hogy ha* $max\_azon_j < max\_azon_i$, *akkor* $új\_info_i$ = *igaz.*

A 4.1.6. állítás r szerinti indukcióval igazolható. Az állítás $r = 0$ esetben teljesül, mert kezdetben mindegyik *új_info* jelző *igaz*. Az indukciós lépésben veszünk egy tetszőleges olyan P_i és P_j folyamatot, ahol $j \in ki\_szom_i$. Ha max_azon_i értéke nő az r-edik menetben, az $új\_info_i$-t mindenképpen *igaz*-ra kell állítani; ekkor tehát teljesül az állítás. Ha max_azon_i értéke nem nő,

akkor két eset képzelhető el. Az egyik esetben már az r-edik menet előtt fennállt a $max\_azon_j \geq max\_azon_i$. Mivel a max_azon_j érték soha nem csökken, a $max\_azon_j \geq max\_azon_i$ igaz az r-edik menet után is. A másik esetben az r-edik menet előtt $max\_azon_j < max\_azon_i$ teljesül. Az indukciós feltevés miatt ilyenkor $új\_info_i = igaz$, amelynek következtében az r-edik menetben a max_azon_i új információ eljut a P_i folyamattól a P_j folyamathoz, így $max\_azon_j \geq max\_azon_i$ megvalósul. Egyik esetben sem kell tehát az $új\_info_i$-t *igaz*-ra állítani.

Most ahhoz, hogy az OptMaxTerjed helyességét belássuk, gondolatban futtassuk egymás mellett ezt és a MaxTerjed algoritmust, úgy, hogy a kezdőértékbeállítás mindkét esetben azonos legyen. A bizonyítás lelke az az invariáns állításként megfogalmazott *szimulációs kapcsolat*, amely azt fejezi ki, hogy ugyanannyi menet után mindkét algoritmus ugyanabba az állapotba kerül.

4.1.7. állítás. *Tetszőleges r-re, ahol $0 \leq r \leq átm$, mindkét algoritmus r-edik menet utáni állapotában az u, a max_azon, a státus és a menetek értékei megegyeznek.*

A 4.1.7. szimulációs állítás bizonyítása az r szerinti indukcióval történik, ugyanúgy, mintha csak egyetlen algoritmus esetében kellene egy állítást igazolni. Az indukciós lépés érdekes része az, amikor rámutatunk arra, hogy egy folyamatnak a max_azon értéke mindkét algoritmus működése során minden menetben megegyezik.

Vegyünk egy tetszőleges P_i és P_j folyamatot, amelyekre $j \in ki\_szom_i$. Ha az r-edik menet előtt az $új\_info_i$=*igaz*, akkor a P_i folyamat ugyanazt az információt küldi el j-ediknek az OptMaxTerjed algoritmusban, mint amit a MaxTerjed algoritmusban. Másrészről, ha az r-edik menet előtt az $új\_info_i$ = *hamis*, az OptMaxTerjed algoritmusban a P_i folyamat semmit sem küld a P_j-nek, ugyanakkor a MaxTerjed algoritmusban a P_i folyamat elküldi a P_j-nek a max_azon_i-t. Ilyenkor azonban a 4.1.6. állítás következtében az r-edik menet előtt $max\_azon_j \geq max\_azon_i$ áll fenn, így az üzenetnek nincs hatása a MaxTerjed algoritmusban. Ebből következik, hogy a P_i folyamat ugyanolyan hatással van a max_azon_j-re mindkét algoritmusban. Mivel ez minden i és j indexre igaz, bármely folyamatnak a max_azon értéke a két algoritmus működése során minden menetben megegyezik.

Hátra van még a 4.1.5. állítás, amely viszont következik a 4.1.7. és 4.1.1 állításokból. □

Ez a módszer, amit az OptMaxTerjed algoritmus helyességének bizonyítására alkalmaztunk, gyakran használható egy osztott algoritmus „optimalizált" változata helyességbizonyítására. Ilyenkor először az algoritmus egy kevésbé hatékony, egyszerűbb változatának helyességét bizonyítjuk, majd a hatékonyabb, de bonyolultabb változat helyességét az egyszerűbb változattal való formális kapcsolata alapján mutatjuk meg. Ez a kapcsolat szinkron hálózatokra megfogalmazott algoritmusoknál a fenti formában – mindkét algoritmus ugyanazon menetszám utáni állapotára megfogalmazott invariáns állítással – írható fel általánosan.

Másik javítás.. A MaxTerjed algoritmus üzeneteinek száma kisebb mértékben még tovább is csökkenthető. Nevezetesen, ha P_i folyamat egy új maximumot

fogad attól a P_j folyamattól, amely egyszerre bejövő és kimenő szomszédja is, azaz amellyel kétirányú kommunikációt folytathat, akkor a P_i-nek nem kell a P_j irányába üzenetet küldenie a következő menetben.

A vezetőválasztás olyan egyedi azonosítókkal ellátott irányított gráfbeli hálózatban is megvalósítható, amelynek sem átmérőjét, sem csúcsainak számát nem ismerjük. Azt javasoljuk az Olvasónak, hogy álljon meg itt, és próbáljon elkészíteni egy ilyen algoritmust. Erre az egyik lehetőség az, hogy olyan kisegítő protokollt vezet be, amely megengedi mindegyik folyamatnak, hogy a hálózat átmérőjét kiszámolja. Felhasználhatja az ebben a fejezetben később található ötleteket is.

4.2.. Szélességi keresés

Most egy olyan szélességi keresést (SZK) mutatunk be, amely egy kitüntetett kezdőcsúcsú, erősen összefüggő irányított gráfra épül. Pontosabban, azt vizsgáljuk meg, hogyan lehet irányított gráfokban *szélességi kereső fát* (SZK fa) felépíteni. Az ilyen fák létrehozását az motiválja, hogy a kommunikáció számára alkalmas szerkezetet találjunk. Az SZK fa minimalizálja a kitüntetett kezdőcsúcsnak megfeleltetett folyamattól a többi folyamat irányába tartó maximális üzenetközvetítési időket (feltételezve, hogy minden egyes él mentén ugyanannyi ideig halad egy-egy üzenet).

Az SZK feladat és annak megoldásai egyszerűbbek azokban az esetekben, ahol a szomszédos csúcsok kétirányú kommunikációval rendelkeznek, azaz ahol a hálózati gráf irányítatlan. Később ezeket az egyszerűsítéseket is be fogjuk mutatni.

4.2.1.. A feladat

A $G = (V, E)$ irányított gráf egy *irányított feszítőfája* olyan gyökeres fa, amely kizárólag a szülőcsúcsból a gyerekcsúcs felé irányuló E-beli irányított élekből áll, és a G összes csúcsát tartalmazza. A G-nek egy i indexű gyökércsúcsú irányított feszítőfája akkor SZK fa, ha az i indexű gyökércsúcstól d távolságra levő minden egyes csúcs a fa d-edik szintjén helyezkedik el (azaz a fában i indexű csúcstól d távolságra). Minden erősen összefüggő irányított gráfnak van egy SZK fája.

Az SZK problémánál feltételezzük, hogy a hálózat erősen összefüggő, és hogy van egy kitüntetett i_0 indexű kezdőcsúcs. Az algoritmus célja, hogy felderítse a hálózati gráf i_0 gyökerű SZK fájának szerkezetét. Ezt osztott módon tartjuk nyilván, úgy, hogy mindegyik P_{i_0}-tól különböző folyamat rendelkezik majd egy *szülő* összetevővel, amely kijelöli azt a csúcsot, amelyik az ő fabeli szülője.

Mint rendszerint, a folyamatok most is csak az irányított élek mentén küldhetnek egymásnak üzeneteket. A folyamatokhoz egyedi azonosítókat rendelünk, és a folyamatok nem rendelkeznek semmiféle ismerettel a hálózat méretéről vagy átmérőjéről.

4.2.2.. Egy alapvető szélességi kereső algoritmus

Az itt tárgyalt úgynevezett SzinkSZK algoritmus alapötlete megegyezik a szokásos szekvenciális szélességi keresés algoritmusáéval.

> **SzinkSZK algoritmus**
> A végrehajtás egy adott pillanatában a folyamatoknak egy bizonyos része „megjelölt", kezdetben azonban egyedül csak az i_0 indexű. Az i_0-adik folyamat egy `keres` üzenetet küld az 1. menetben az összes kimenő szomszédjának. Bármelyik menetben, ha egy megjelöletlen folyamat fogad egy `keres` üzenetet, megjelöli magát és szülőnek választ egy folyamatot azok közül, amelyektől a `keres` üzenetet kapta. Közvetlenül azután, hogy a folyamat megjelölt lett, a következő menetben tovább küldi a `keres` üzenetet a kimenő szomszédjainak.

Könnyű megmutatni, hogy a SzinkSZK algoritmus egy SZK fát állít elő. Ahhoz, hogy ezt formálisan is igazoljuk, be kell bizonyítani azt az állítást, amely szerint minden olyan csúcs, amelyik az i_0-tól d ($1 \leq d \leq r$) távolságra van, r menet után már rendelkezik szülő mutatóval; továbbá minden ilyen mutató egy olyan csúcsra mutat, amelynek az i_0-tól vett távolsága $d-1$. Ez az invariáns a szokásos módon, a menetek száma szerinti indukcióval látható be.

Bonyolultságelemzés.. A futási időt meghatározó menetek száma legfeljebb *átm* darab lehet. (A valóságban ezt finomíthatjuk egy kicsit az i_0 indexű csúcsnak a többi csúcstól vett távolságainak maximumára.) Az üzenetek száma éppen $|E|$, hiszen egy `keres` üzenet pontosan egyszer halad át minden irányított élen.

A kommunikációs bonyolultság csökkentése.. Akárcsak a MaxTerjed algoritmusnál, némileg itt is csökkenthetjük az üzenetek számát. Egy újonnan megjelölt folyamatnak ugyanis nem szükséges a `keres` üzenetet azon folyamatokhoz elküldenie, amelyektől már kapott ilyen üzenetet.

Üzenetszórás.. A SzinkSZK-t könnyű úgy kiegészíteni, hogy egy üzenet szétküldését valósítsa meg. Ha egy folyamatnak van egy olyan m üzenete, amelyet közölni akar a hálózat összes folyamatával, magát kezdőcsúcsnak választva végrehajtja a SzinkSZK algoritmust, úgy, hogy az 1. menetben elküldött `keres` üzenet mellé csatolja az m üzenetet is. A többi folyamat ugyancsak hozzákapcsolja az általa elküldött üzenethez az m üzenetet. Mivel a fa az összes csúcsot tartalmazza, az m üzenet minden folyamathoz eljut.

Gyerek mutató.. Az SZK probléma egy fontos változata az, amikor megköveteljük, hogy a folyamatok ne csak a fabeli szülőjüket, hanem a gyerekeiket is ismerjék meg. Ebben az esetben arra van szükség, hogy amikor egy folyamat egy `keres` üzenetet fogad, akkor egy `szülő` vagy `nem_szülő` üzenettel feleljen a küldő folyamatnak, hogy ezzel közölje, őt választotta-e szülőjének. Ha mindegyik szomszédos pár közt kétirányú kommunikációt engedünk meg, azaz a hálózati gráf irányítatlan, akkor ennek a többletkommunikációnak a megvalósítása nem okoz – egy kis költség növekedésen kívül más – problémát. Mivel azonban olyan szomszédos párokat is megengedünk, amelyek között csak egyirányú kapcsolat

van, a `szülő` vagy `nem_szülő` üzenetek közül néhányat közvetett úton kell elküldenünk. Ezt megtehetjük például úgy, hogy újra végrehajtjuk a SzinkSZK algoritmust, és a korábban látott módon hozzácsatoljuk a `szülő` vagy `nem_szülő` üzenetet. Ahhoz, hogy egy ilyen üzenetet a megfelelő fogadó folyamat felismerjen, a `keres`-hez csatolt üzenetnek tartalmaznia kell a fogadó egyedi azonosítóját (és egy helyi nevet, amely alapján a fogadó felismeri a küldőt). Megjegyezzük, hogy a SzinkSZK algoritmus ezen végrehajtásainak többsége párhuzamosan is futtatható. Ahhoz, hogy a formális modellünkhöz – amely szerint menetenként legfeljebb egy üzenet küldhető egy kapcsolaton keresztül – illeszkedjünk, több üzenet egyidejű küldése esetén azokat egyetlen üzenetbe kell belefoglalnunk.

A nem kizárólag kétirányú kommunikációt folytató irányított gráfoknál hasznos lehet a gyerekből a szülőbe vezető legrövidebb utat is kiszámoltatni a folyamatokkal a szülő és gyerek mutatókon kívül. Ilyen információ például a SzinkSZK algoritmus újbóli végrehajtásával állítható elő.

Bonyolultságelemzés.. Ha a gráf irányítatlan, az SZK fa meghatározásának futási ideje, beleértve a gyerek mutatók kiszámolását is, $\mathcal{O}(\acute{a}tm)$, a kommunikáció bonyolultsága pedig $\mathcal{O}(|E|)$.

Még ha valamely szomszédos pár között egyirányú kapcsolat van is, a fának a gyerek mutatókkal együtt való meghatározása $\mathcal{O}(\acute{a}tm)$ ideig tart csak, mert a további SZK-kat párhuzamosan is végre lehet hajtani. Ebben az esetben az üzenetek teljes száma $\mathcal{O}(\acute{a}tm|E|)$, mert legfeljebb $|E|$ üzenetet lehet menetenként küldeni, a menetek száma pedig $\mathcal{O}(\acute{a}tm)$. De mivel egy üzenet akár $|E|$ darab egyidejű SZK végrehajtásáról is tartalmazhat információt, egy üzenetben $|E|b$ bit lehet, ahol a b egy egyszerű egyedi azonosító ábrázolásához szükséges bitek számát jelöli. Ez vezet a kommunikáció bitjeinek teljes $\mathcal{O}(\acute{a}tm|E|^2b)$ számához. A bitek teljes számára egy kisebb korlát is kapható, ha észrevesszük, hogy a (legfeljebb $|E|$) darab egyidejű SZK végrehajtása legfeljebb $|E|$ olyan üzenetet használ, amelyek legfeljebb b bitből állnak. Így a kommunikációs bitek teljes száma $\mathcal{O}(|E|^2b)$.

Befejeződés.. Hogyan ismerheti fel a P_{i_0} folyamat, hogy a fa építése befejeződött? Ha minden egyes `keres` üzenetre `szülő` vagy `nem_szülő` üzenet a válasz, akkor azután már, hogy egy folyamat minden `keres` üzenetére választ kapott, ismerni fogja az összes fabeli gyerekét, és tudja, hogy azok már mind megjelölődtek. Ezeket az SZK fa leveleitől induló „befejezés-jelzéseket" össze kell gyűjtenünk a kezdőcsúcsban: minden egyes folyamat küld a fabeli szülőjének egy befejezést jelző üzenetet, azután, hogy (a) mindenkitől, akinek a `keres` üzenetet elküldte, választ kapott (így tudja, hogy kik a gyerekei, és hogy azok meg vannak jelölve), és (b) minden gyerekétől megkapta a befejezést jelző üzenetet. Az ilyen típusú eljárást *„üzenetgyűjtés"*-nek nevezzük.

Ha a gráf irányítatlan, az SZK fa előállításának ideje, beleértve a gyerek mutatók beállítását és a befejezést jelző üzeneteknek a kezdőcsúcshoz való visszajuttatását is, $\mathcal{O}(\acute{a}tm)$; a kommunikációs bonyolultság pedig csak $\mathcal{O}(|E|)$. Ha csak egyirányú kommunikációt engedünk meg, a teljes futási idő, beleszámítva a befejezést jelző üzenetek küldését is, $\mathcal{O}(\acute{a}tm^2)$. Ennek a négyzetes viselkedésnek az az oka, hogy befejezés-jelzéseket szintenként egymás után kell beállítani. Az üze-

netek teljes száma $\mathcal{O}(\acute{a}tm^2|E|)$ és a kommunikációs bitek teljes száma legfeljebb $\mathcal{O}(|E|^2 b)$.

4.2.3.. Alkalmazások

A szélességi keresés egy alapvető építőeleme az osztott algoritmusoknak. Most néhány példát mutatunk arra, hogyan használhatjuk fel a SzinkSZK algoritmust különféle feladatok megoldására.

Üzenetszórás.. Mint azt korábban már említettük, egy üzenet szétküldését egy SZK fa létesítésével valósíthatjuk meg. Ennek egy másik módja az, hogy először egy gyerek mutatókkal ellátott SZK fát hozzunk létre, úgy, mint azt fent leírtuk, és azután ezt a fát használjuk az üzenetek továbbítására. Csak az üzenetet kell végig terjeszteni a szülőktől azok gyerekeihez. Ez lehetővé teszi, hogy az SZK fa előállítására fordított munkát újra és újra felhasználhassuk, hiszen ugyanabban a fában több üzenet is elküldhető. Az SZK fát egyszer kell csak létrehozni, az egyes üzenetek közvetítésére felhasznált további idő mindössze $\mathcal{O}(\acute{a}tm)$, a továbbított üzenetek száma pedig csak $\mathcal{O}(n)$.

Globális számítás.. Az SZK fáknak egy másik alkalmazása a hálózaton keresztül történő információgyűjtés, vagy még általánosabban, egy osztott bemenetekre támaszkodó függvény értékének kiszámítása. Vegyük például azt a problémát, ahol minden folyamatnak egy nemnegatív egész bemenő értéke van, és meg akarjuk találni a hálózat bemeneteinek összegét. Ezt egy SZK fa felhasználásával könnyedén (és hatékonyan) megtehetjük, az alábbi módon. A levelektől elindulva összegyűjtjük az eredményeket az alábbi „üzenetgyűjtő" eljárással. Mindegyik levél elküldi az értékét a szülőjének; mindegyik szülő vár, amíg minden gyerekétől megkapja az értékeket, hozzáadja azokat saját bemenő értékéhez, és elküldi az összeget a saját szülőjéhez. Az SZK fa gyökerében kiszámolt összeg a válasz.

Feltételezve, hogy az SZK fát már korábban létrehoztuk és hogy a fa minden élén kétirányú kommunikációt engedünk meg, ez a séma $\mathcal{O}(\acute{a}tm)$ idő alatt $\mathcal{O}(n)$ üzenetet továbbít. Ugyanez a séma használható sok más függvény, például egész számok maximumának vagy minimumának kiszámolására is. (Az ilyen függvényeknek asszociatívnak és kommutatívnak kell lenniük.)

Vezetőválasztás.. A SzinkSZK-t felhasználva olyan algoritmust is tervezhetünk, amely egy egyedi azonosítókat tartalmazó hálózatban, ahol a folyamatok nem ismerik sem az n, sem az $\acute{a}tm$ értékét, vezető folyamatot választ. Minden folyamat párhuzamosan elindít egy szélességi keresést. A folyamatok az így felépített fa segítségével az előbb bemutatott globális számítási eljárás keretében határozzák meg a maximális egyedi azonosítót. Az a folyamat, amelyik a maximális egyedi azonosítóval rendelkezik, vezetőnek nyilvánítja magát, a többi pedig nemvezetőnek. Ha a gráf irányítatlan, a futási idő $\mathcal{O}(\acute{a}tm)$, az üzenetek száma pedig $\mathcal{O}(\acute{a}tm|E|)$, aminek oka ismét az, hogy legfeljebb $|E|$ üzenet küldhető az $\acute{a}tm$ darab menet mindegyikében. A bitek száma legfeljebb $\mathcal{O}(n|E|b)$, ahol b az egyedi azonosító ábrázolásához felhasznált bitek maximális száma.

Az átmérő kiszámítása.. A hálózat átmérője úgy számolható ki, hogy minden folyamat párhuzamosan elindít egy szélességi keresést. A P_i folyamat az így előállított fát arra használja, hogy meghatározza a $max\_távolság_i$ értéket, amely a P_i és a hálózat valamelyik másik folyamata közötti távolság legnagyobb értéke. Majd a folyamat újra felhasználja ezt a fát egy olyan globális számításban, amely előállítja a *max_ táv* értékek maximumát. Ha a gráf irányítatlan, a futási idő $\mathcal{O}(átm)$, az üzenetek száma $\mathcal{O}(átm|E|)$, a bitek száma legfeljebb $\mathcal{O}(n|E|b)$. Az így kiszámolt átmérőt felhasználhatjuk például a vezetőválasztó MaxTerjed algoritmusban.

4.3.. Legrövidebb utak

Vizsgáljuk meg most az SZK probléma egy általánosítását. Vegyünk ismét egy olyan erősen összefüggő irányított gráfot, ahol a szomszédok között csak egyirányú kapcsolat van. Tegyük fel, hogy mindegyik $e = (i, j)$ irányított él rendelkezik egy nemnegatív súllyal, amelyet *súly(e)*-vel vagy $súly_{i,j}$-vel jelölünk. A feladat az, hogy legrövidebb utat találjunk az irányított gráf egy kitüntetett i_0 indexű kezdőcsúcsából az irányított gráf másik csúcsába. Legrövidebb úton a minimális összsúlyú utat értjük.[2] Az i_0 indexű kezdőcsúcsból az összes többi csúcsba vezető legrövidebb utak együttese egy olyan részfát alkot az irányított gráfban, amelynek minden éle szülőtől gyerek felé irányul.

Egy ilyen fa előállítását – akárcsak a szélességi keresésnél – az indokolja, hogy megfelelő kommunikációs szerkezetet adjunk az üzenetszóráshoz. A súlyok az élek bejárásának költségét, például a kommunikáció idejét vagy árát fejezik ki. A legrövidebb utak fája minimalizálja a kitüntetett folyamatnak a hálózat bármely másik folyamatával történő kommunikációjának költségét.

Feltesszük, hogy kezdetben minden folyamat ismeri mindegyik hozzá kapcsolódó élnek a súlyát, és ezt speciális *súly* változókban tároljuk az él végpontjait adó mindkét folyamatnál. Ugyancsak feltesszük, hogy az egyes folyamatok ismerik az irányított gráf csúcsainak n számát. Megköveteljük azt is, hogy mindegyik folyamat egy részleges legrövidebb utak fájában is meg tudja határozni a szülőjét, és annak az i_0-tól való távolságát (azaz az odavezető legrövidebb út teljes költségét) is.

Ha mindegyik él súlya azonos, az SZK fa is egy legrövidebb utak fája. Ebben az esetben az egyszerű SzinkSZK egy nyilvánvaló módosításával elérhető, hogy előállítsuk a távolságokat és a szülő mutatókat.

A különböző súlyú élek esete már érdekesebb. Ezt a problémát például megoldhatjuk a Bellman-Ford legrövidebb utak szekvenciális algoritmusának egy osztott változatával.

BellmanFord algoritmus (vázlatosan)

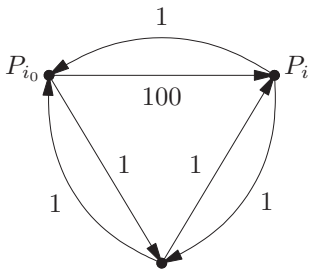
Minden P_i folyamat nyilvántartja a *táv* összetevőjében az adott pillanatig megismert P_{i_0}-tól vett legrövidebb távolságot és az ahhoz tartozó *szülő*-t,

[2] A súly és a távolság fogalma – a hagyománynak megfelelően – szerencsétlen módon keveredik.

azaz azt a bejövő szomszédos folyamatot, amely a $P-i$-t közvetlenül megelőzi a *táv* távolságú úton. Kezdetben $távᵢ₀ = 0$, minden $i \neq i_0$ folyamatra $távᵢ = \infty$, a *szülő* összetevő értéke pedig nem ismert. Az egyes menetekben mindegyik folyamat elküldi a *táv* értékét az összes kimenő szomszédjának. Azután „egy enyhítő lépésben" egy P_i folyamat megváltoztatja a *táv* értékét annak korábbi értéke és az összes a $távⱼ + súlyⱼ,ᵢ$ érték (ahol j index egy bejövő szomszédot jelöl) közül a legkisebbre. Ha a *táv* változik, a *szülő* összetevőt is frissíteni kell. Az $(n-1)$-edik menet után a *táv* tartalmazni fogja a legrövidebb távolságot, a *szülő* összetevő pedig a legrövidebb utak fájában szereplő szülőt.

Könnyű belátni, hogy $n-1$ menet után a *táv* értékek a helyes távolságokhoz közelítenek. A BellmanFord algoritmus helyességét például r szerinti indukcióval igazolhatjuk. Belátható ugyanis, hogy az r-edik menet után minden P_i folyamat *táv* és *szülő* összetevője megfelel egy olyan legrövidebb útnak, amely P_{i_0}-ból P_i-be vezet és legfeljebb r darab élt tartalmaz. (Ha nincs ilyen út, a $táv{=}\infty$ és a *szülő* nem meghatározott.) A részletek kidolgozását meghagyjuk gyakorlatnak (lásd 4-14. gyakorlat).

Bonyolultságelemzés.. A BellmanFord algoritmus futási ideje $n-1$, az üzenetek száma $(n-1)|E|$.



4.1.. ábra. A legrövidebb utak csak 2 menet után ismerhetők fel, pedig az $átm = 1$.

4.3.1. példa. A BellmanFord futási ideje

A SzinkSZK algoritmus futási idejét alapul véve azt gyaníthatnánk, hogy a BellmanFord algoritmus futási ideje valójában *átm*. A 4.1. ábrán látható példa azt mutatja, hogy ez nincs így. Ebben a példában ugyanis 2 menet kell ahhoz, hogy a P_{i_0}-ból P_i-be vezető út helyes távolsága, a 2 beállítódjon, mivel az út, amely mentén ez a távolság kialakul, két élből áll. Az átmérő ellenben csak 1.

A BellmanFord algoritmus az n helyett annak felső korlátjával is dolgozhat. Ha nem ismerünk ilyen korlátot, a 4.2. alfejezetben bemutatott módszert használhatjuk.

4.4.. Minimális feszítőfa

Most egy élsúlyozott irányított gráfban fogunk minimális (súlyú) feszítőfát (MFF) keresni. Egy ilyen fát az üzenetszórásnál használhatunk fel. Egy minimális súlyú feszítőfa minimalizálja annak a kommunikációnak a teljes költségét, amelyet egy tetszőleges forrásfolyamat a hálózat összes többi folyamatával folytat.

4.4.1.. A feladat

Egy $G = (V, E)$ irányítatlan gráf egy *feszítőerdeje* olyan erdő (azaz körmentes, de nem feltétlenül összefüggő gráf), amely kizárólag E-beli élekből áll és minden G-beli csúcsot tartalmaz. A G irányítatlan gráf egy *feszítőfája* G egy összefüggő feszítőerdeje. Ha az irányítatlan E-beli élekhez súlyokat rendelünk, akkor G bármely részgráfjának (feszítőfájának vagy feszítő erdejének) súlyán a benne levő élek súlyának összegét értjük.

Emlékezzünk arra, hogy modellünkben az irányítatlan gráfokat olyan irányított gráfoknak tekintjük, ahol a szomszédos csúcsokat kétirányú élek kötik össze. A 4.3. alfejezethez hasonlóan feltesszük, hogy mindegyik $e = (i, j)$ irányított él rendelkezik egy nemnegatív valós értékű súllyal, $súly(e) = súly_{i,j}$, és tegyük fel azt is, hogy minden i és j indexű csúcsra $súly_{i,j} = súly_{j,i}$. Feltesszük továbbá, hogy kezdetben minden folyamat ismeri mindegyik hozzá kapcsolódó élnek a súlyát, és ezt speciális *súly* változókban tároljuk az él végpontjait adó mindkét folyamatnál. Feltesszük, hogy a folyamatok rendelkeznek egyedi azonosítókkal, és ismert a csúcsok n száma is. A feladat az, hogy találjunk egy minimális súlyú (irányítatlan) feszítőfát a teljes hálózatra; konkrétabban, minden folyamatnak el kell tudnia dönteni, hogy a bevezető élei közül melyik tartozik a minimális feszítőfához, és melyik nem.

4.4.2.. Elméleti háttér

Minden ismert MFF algoritmus, a szekvenciálisak éppúgy, mint a párhuzamosak, ugyanarra a most bemutatandó ötletre épülnek. A minimális feszítőfák felépítésének alapmódszere az n darab, egyetlen csúcsból álló triviális feszítőerdőből indul ki, majd újra és újra összeköti az erdő két összetevőjét egy megfelelő éllel egészen addig, amíg egy feszítőfát nem hoz létre. Ahhoz, hogy a végén minimális feszítőfát kapjunk, fontos, hogy mindig azt az élt válasszuk ki az összekötéshez, amely egy összetevő kivezető élei közül a minimális súlyú. Ennek a kiválasztásnak az igazolását a következő lemma adja.

4.3. lemma. . *Legyen $G = (V, T)$ egy élsúlyozott irányítatlan gráf, és legyen $(V_i, E_i) : 1 \leq i \leq k$ a G egy feszítőerdeje, ahol $k > 1$. Rögzítsünk egy tetszőleges i-t $(1 \leq i \leq k)$. Legyen e az*

$$\{e' : e'\text{-nek pontosan egyik végpontja van } V_i\text{-ben}\}$$

halmaz legkisebb súlyú éle.

Ekkor van a G-nek egy olyan feszítőfája, amely a $\bigcup_j E_j$ mellett magában foglalja az e-t is, továbbá ez a fa minimális súlyú azon fák között, amelyek az $\bigcup_j E_j$-t tartalmazzák.

Bizonyítás. Bizonyítsunk indirekt módon. Tegyük fel, hogy létezik olyan T feszítőfa, amely tartalmazza $\bigcup_j E_j$-t, de nem tartalmazza e-t, ugyanakkor szigorúan kisebb súlyú, mint bármelyik más $\bigcup_j E_j$-t és az e-t tartalmazó feszítőfa. Jelöljük T'-vel azt a gráfot, amelyet úgy kapunk, hogy hozzávesszük T-hez az e-t. Világos, hogy T' tartalmaz egy olyan kört, amelynek van olyan V_i-ből kivezető e' éle, amely az e-től különbözik.

Az e választása miatt $súly(e') \geq súly(e)$. Készítsük most el a T'' gráfot, úgy, hogy a T'-ből töröljük az e'-t. Ekkor a T'' a G-nek egy olyan feszítőfája, amely tartalmazza $\bigcup_j E_j$-t és az e-t, de a súlya nem nagyobb, mint a T súlya. Ez azonban ellentmond az indirekt feltevésnek. □

A 4.3. lemma igazolja az alábbi MFF-t építő általános módszer helyességét.

Általános módszer az MFF-hez

Induljunk ki az n darab, egyetlen csúcsot tartalmazó triviális feszítőerdőből. Ezután ciklikusan hajtsuk végre a következőt: válasszunk az erdőből egy tetszőleges C összetevőt, és egy minimális súlyú, C összetevőből kivezető e élt. Vonjuk össze a C összetevőt, az e élt, és az e él végpontján levő másik összetevőt egy új összetevőbe. Akkor álljunk le, ha már csak egyetlen összetevőnk maradt.

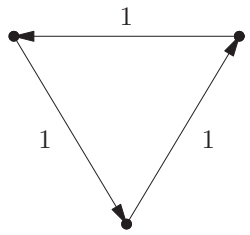
A 4.3. lemma felhasználható annak induktív bizonyításához, hogy a fenti eljárás minden lépésében a nyilvántartott erdő egy részgráfja valamelyik MFF-nek. Több jól ismert szekvenciális MFF algoritmus is ennek az általános stratégiának az egyedi esete. Például a *Prim–Dijkstra-algoritmus* kiválaszt a kiinduló (egyetlen csúcsból álló) összetevők közül egyet, és ezt bővíti újra és újra az abból kiinduló legkisebb súlyú éllel, amíg egy teljes feszítőfát nem kap. Másik példa a *Kruskal-algoritmus*, amely minden lépésben az aktuális feszítőerdő összetevőit összekötő legkisebb súlyú élt választja ki, ennek segítségével egyesít két összetevőt egészen addig, amíg egyetlen összetevő nem marad, amely már a végleges feszítőfa lesz.

Ahhoz, hogy ezt az általános stratégiát osztott módon alkalmazzuk, az lenne jó, ha az erdőt párhuzamosan egyszerre több éllel is bővíthetnénk. Azaz az összetevők mindegyike egymástól függetlenül meghatározhatná a saját minimális súlyú kimenő élét, amelyet aztán hozzáadhatnánk az erdőhöz, és ez így egyidejűleg több összetevőpár egyesítését foglalná magába. A 4.3. lemma azonban nem garantálja az ilyen párhuzamos stratégia helyességét. Sőt általában ez a stratégia nem is helyes.

4.4.1. példa. Kör keletkezése egy párhuzamos MFF algoritmusban

Vegyük a 4.2. példa gráfját. A csúcsok a feszítőfa összetevőit ábrázolják. A három 1 súlyú él az összes kimenő él. Ha az összetevők kivá-

lasztanák a nyilak által megjelenített legkisebb súlyú kimenő élüket, kör keletkezne.



4.2.. ábra. Minimális súlyú kivezető élek párhuzamos kiválasztása kört eredményez.

Abban a különleges esetben elkerülhetők a körök, amikor minden élnek eltérő súlya van. Ez az alábbi lemma miatt van így.

4.4. lemma. . *Ha egy G gráf minden éle különböző súllyal rendelkezik, akkor G-nek pontosan egy MFF-je van.*

Bizonyítás. A bizonyítás a 4.3. lemma bizonyításához hasonlóan történik. Feltesszük, hogy van két különböző minimális súlyú feszítőfa, T és T', és legyen e az a minimális súlyú él, amely a két fa közül csak az egyikben jelenik meg. Tegyük fel (az általánosság megsértése nélkül), hogy $e \in T$. Ekkor az e-nek T'-höz adásával egy T'' kört tartalmazó gráfot kapunk, és ebben a körben legalább egy olyan e' él is van, amely nem szerepel a T-ben. Mivel az élsúlyok mind különböznek és az e' a két fa közül csak az egyikben szerepel, az e választása miatt biztosan fennáll, hogy $súly(e') > súly(e)$. Az e'-nek T''-ből való eltávolítása egy olyan feszítőfát eredményez, amely kisebb súlyú, mint a T', és ez ellentmond az indirekt feltevésnek. □

Most vizsgáljuk felül az általános stratégiát abban az esetben, amikor a gráf különböző élsúlyokat tartalmaz, és így a 4.4. lemma szerint egyetlen kizárólagos MFF-je van. Ilyenkor a feszítőfa építése során az erdő bármelyik összetevőjének pontosan egy minimális súlyú kimenő éle van (amelyet a kiejthetetlen MSKÉ-vel rövidítünk). A 4.3. lemmából következik, hogy ha egy olyan erdővel van dolgunk, amelynek minden éle a kizárólagos MFF-ben van, az összes összetevő MSKÉ-je is a kizárólagos MFF-hez tartozik. Így mindet azonnal hozzáadhatjuk annak a veszélye nélkül, hogy kört hoznánk létre.

4.4.3.. Algoritmus

Most a fent leírt általános módszert követve bemutatunk egy tetszőleges élsúlyozott irányítatlan gráfban MFF-et építő osztott algoritmust. Mivel az összetevőket párhuzamosan egyesíthetjük, feltesszük, hogy az élek különböző súlyúak; majd

ennek a pontnak a végén megmutatjuk azt is, hogyan lehet ezt a feltételt elhagyni. Az algoritmust SzinkGHS-nek hívjuk, mert ez a Gallager, Humblet, és Spira által kifejlesztett aszinkron algoritmusra épül. (Majd a 15.5. alfejezetben fogjuk bemutatni az egyszerűen GHS-nek nevezett aszinkron algoritmust.)

SzinkGHS algoritmus

Az algoritmus "szintről szintre" építi a feszítőerdőnek (az MFF részfáiként megjelenő) összetevőit. Minden k-ra, a k szint összetevőinek legalább 2^k csúcsa van. Az egyes összetevők minden szinten rendelkeznek egy kitüntetett vezető csúccsal. A folyamatok minden szintet rögzített számú, $\mathcal{O}(n)$ menetben fejeznek be.

Az algoritmus egyetlen csúcsból álló, éleket nem tartalmazó 0 szintű összetevőkkel indul. Tegyük fel, hogy a k szintű összetevőket (a vezető csúcsaikkal együtt) már meghatároztuk. Részletesebben, tegyük fel, hogy minden folyamat ismeri saját összetevője vezetőjének egyedi azonosítóját; és ezek az egyedi azonosítók az egyes összetevőket is azonosítják. Minden folyamat tudja, hogy a hozzá kapcsolódó élek közül melyik tartozik a folyamatot tartalmazó összetevőbe.

A $k+1$ szintű összetevőket úgy kapjuk meg, hogy minden k szintű összetevő megkeresi a saját MSKÉ-jét. A vezető folyamat a 4.2. alfejezetben leírt üzenetszóró stratégia alapján indítja el ezt a keresést. Minden folyamat kiválasztja azt a legkisebb súlyú élt, amelyik kivezet a folyamatot tartalmazó összetevőből (ha van egyáltalán ilyen); ezt úgy érik el, hogy egy `teszt` üzenetet küldenek minden nem az összetevőhöz tartozó él mentén, hogy megtudják, annak másik végpontja ugyanabban az összetevőben található-e. (Ez az összetevők azonosítóinak összehasonlításával eldönthető.) Ezután a folyamatok az üzenetgyűjtés elvén továbbítják az egyes csúcsokból kivezető minimális súlyú élről származó információt, úgy, hogy összevetik annak súlyát a hozzájuk érkező többi élsúllyal, és mindig a minimális súlyút küldik tovább a vezető folyamat felé. A vezető folyamat által előállított minimum a szóban forgó összetevő MSKÉ-je.

Amikor minden k szintű összetevő megtalálta a saját MSKÉ-jét, az összetevőket ezen MSKÉ-k mentén egyesítjük, ezáltal megkapjuk a $k+1$ szintű összetevőket. Ez magában foglalja, hogy minden k szintű összetevő vezető folyamata kapcsolatba lép az MSKÉ végpontjához tartozó folyamattal, hogy az az új összetevőhöz tartozóként jelölje meg az élt.

Ezután a $k+1$ szintű összetevőkben új vezető folyamatot kell választani a következő módon: megmutatható, hogy a k szintű öszetevők minden $k+1$ szintű összetevőbe egyesítendő csoportjában egyetlen olyan e él van, amely közös MSKÉ-je a csoportba tartozó két k szintű összetevőnek. (Ezt később látjuk be.) Legyen az új vezető folyamat ennek az e élnek a nagyobb egyedi azonosítójú végpontja. Megjegyezzük, hogy ez az új vezető folyamat a helyi információkra támaszkodva azonosíthatja magát.

Végül az új vezető folyamat üzenetszórás útján elterjeszti az egyedi azonosítóját az új összetevőben.

Néhány szint után a feszítőerdő a hálózat összes csúcsát tartalmazó egyetlen összetevőből áll majd. Ezután már nem lehet MSKÉ-t találni, mert egyik folyamatnak sincs már az összetevőből kivezető éle. Amikor a vezető folyamat megtudja ezt, szétsugároz egy üzenetet, amely elmondja, hogy az algoritmus befejeződött.

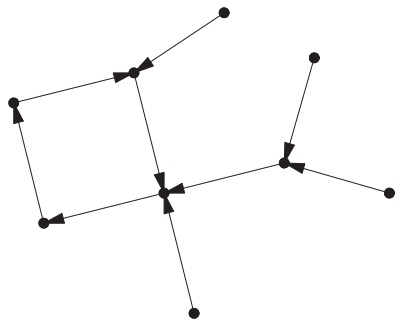
Az algoritmus kulcsa az, hogy a k szintű összetevők egyesítendő csoportjában egyetlen olyan (irányítatlan) él található, amely a végpontjait tartalmazó összetevőknek közös MSKÉ-je. Ahhoz, hogy lássuk, miért van ez így, bevezetjük az *összetevők irányított gráfját*, a G'-t, amely csúcsai azok a k szintű összetevők, amelyeket egy $k+1$ szintű összetevő előállításához használunk fel, élei pedig az MSKÉ-k. G' egy gyengén összefüggő irányított gráf, amelyben minden csúcsnak pontosan egy kivezető éle van. (Egy irányított gráf *gyengén összefüggő*, ha az irányítatlan változata, amelyet úgy kapunk, hogy figyelmen kívül hagyjuk az éleinek irányítását, összefüggő.) Felhasználhatjuk az alábbi tulajdonságot.

4.5. lemma. . *Legyen G egy gyengén összefüggő irányított gráf, amelyben minden csúcsnak pontosan egy kivezető éle van. Ekkor a G pontosan egy kört tartalmaz.*

Bizonyítás. A bizonyítást meghagyjuk gyakorlatnak (lásd 4-16. gyakorlat). □

4.4.2. példa. Csúcsonként egy kivezető élt tartalmazó gráf

A 4.3. ábra egy olyan gráfot mutat, amelynek minden éle pontosan egy kivezető élt tartalmaz.



4.3.. ábra. Egy gráf, amelyben minden egyes csúcs pontosan egy kivezető élt tartalmaz. Megfigyelhetjük az egyetlen kört is.

Alkalmazzuk a 4.5. lemmát az összetevők irányított G' gráfjára, hogy megkapjuk az összetevők egyetlen körét. A G' előállítása miatt a kör egymást követő

éleinek súlya nem növekszik; ezért az ilyen körnek a hossza nem lehet 2-nél nagyobb. Így az egyetlen kör pontosan 2 hosszú. Ez azonban egy élnek felel meg, amely mindkét szomszédos összetevőnek a közös MSKÉ-je.

A SzinkGHS algoritmus kritikus pontja a szintek szinkronizálása. Ehhez azt kell biztosítanunk, hogy amikor a P_i folyamat megpróbálja meghatározni, hogy a P_j folyamattal közös összetevőben vannak-e, akkor mindkét folyamat már az aktuális összetevő-azonosítóval rendelkezzen. Ha a P_j folyamat azonosítója különbözik a P_i azonosítójától, biztosak szeretnénk lenni abban, hogy P_i-edik és P_j-edik tényleg különböző összetevőben van, és nem csak arról van szó, hogy még nem kapták meg a vezetőjüktől az összetevőjük azonosítóját. Ahhoz, hogy a szinteket szinkronizáltan hajtsuk végre, a folyamatok az egyes szinteken előre meghatározott számú menetet engednek meg. Ahhoz, hogy a menetek minden számítása biztosan befejeződjön, ez a szám legyen $\mathcal{O}(n)$; megjegyezzük, hogy a $\mathcal{O}(\mathit{átm})$ nem mindig elegendő, és kizárólag emiatt kell a csúcsoknak ismerni az n-t. (A 15.5. alfejezetben, amikor az aszinkron hálózatokban újra megvizsgáljuk ezt az algoritmust, az összetevők szinkronizálására ettől eltérő módszert fogunk használni.)

Bonyolultságelemzés.. Figyeljük meg először, hogy minden k szintű összetevő legalább 2^k csúcsot tartalmaz. Ezt indukcióval lehet belátni, úgy, hogy észrevesszük, egy összetevőt minden szinten ugyanazon szint legfeljebb egy másik összetevőjével egyesíthetünk. Így a szintek száma legfeljebb $\log\ n$. Mivel mindegyik szint $\mathcal{O}(n)$ ideig tart, ebből az következik, hogy a SzinkGSH futási ideje $\mathcal{O}(n\ \log\ n)$. A kommunikáció bonyolultsága $\mathcal{O}((n+|E|)\cdot\log\ n)$, mert mindegyik szinten $\mathcal{O}(n)$ üzenetet kell a fa összes élén végig küldeni, és $\mathcal{O}(|E|)$ további üzenet szükséges az adott csúcsból kivezető legkisebb súlyú élek megtalálásához.

Üzenetek számának csökkentése.. Az üzenetek számát az adott csúcsból kivezető minimális súlyú élek óvatosabb stratégiájú keresésével $\mathcal{O}(n\ \log\ n+|E|)$-re csökkenthetjük. Ez a javítás a futási idő növekedéséhez vezet, bár annak nagyságrendje nem nő. Az ötlet az alábbi:

Minden egyes folyamat jelölje meg a hozzá tartozó élt az „elutasítva" címkével, amikor kiderül, hogy ugyanazon összetevőhöz tartozó csúcshoz vezet; így nincs szükség arra, hogy ezeket később ismét megvizsgáljuk. Az egyes szinteken csak a többi élt teszteljük egyenként a súlyuk szerint növekedő sorrendben, amíg az első olyat meg nem találjuk, amely kivezet az összetevőből (vagy amíg az élek el nem fogynak).

E javítás mellett a fa élein küldött üzenetek száma továbbra is $\mathcal{O}(n\ \log\ n)$. Nézzük most meg az adott csúcsokból kivezető minimális súlyú élek megtalálásának amortizációs elemzését. Minden él legfeljebb egyszer válik teszteltté és elutasítottá, ez összesen $\mathcal{O}(|E|)$. Egy adott csúcsból kivezető, már tesztelt minimális súlyú élt, amely nem a szóban forgó összetevő MSKÉ-je, újra lehet tesztelni, de minden szint minden csúcsánál legfeljebb egy ilyen vizsgálatra kerülhet sor, ami $\mathcal{O}(n\ \log\ n)$. A teljes kommunikációs bonyolultság így $\mathcal{O}(n\ \log\ n+|E|)$.

A most vázolt módszernek van egy másik előnye. Mivel az egyes csúcsok megjelölik mind a hozzájuk tartozó MFF-be kerülő éleket, mind azokat, amelyek nincsenek a fában, nincs szükség arra, hogy a végső fázisban a vezető folyamat

jelezze mindenkinek az algoritmus befejezését. Mindegyik csúcs egyszerűen tartalmazza a hozzá tartozó élekkel kapcsolatos információt.

Nem feltétlenük különböző élsúlyok.. Vizsgáljuk most meg az MFF problémát egy olyan gráfban, ahol az élsúlyok nem feltétlenül különbözők. Ebben az esetben egy kis módosítással alkalmazhatjuk a SzinkGHS algoritmust. Vegyük először észre, hogy a SzinkGHS algoritmus a súlyokat csak a $\{<, >, =\}$ összehasonlításoknak veti alá.

Tetszőleges élsúlyok esetén könnyen származtathatunk a csúcsok egyedi azonosítójából „élazonosítókat". Egy (i, j) élazonosítója a $(súly_{i,j}, v, v')$ hármas lesz, ahol v és v' az i és j indexű csúcsok egyedi azonosítói, és $v < v'$. (Így (i, j) és (j, i) ugyanazzal az azonosítóval bír.) A hármasok lexikografikus rendezése teljes rendezést határoz meg az élazonosítók között.

Mivel a SzinkGHS algoritmus csak összehasonlításra használja a súlyokat, így azt a valódi élsúlyok helyett az élazonosítókkal is lefuttathatjuk; ez ugyanazt a végrehajtást eredményezi, mint ha a SzinkGHS algoritmust az ugyanúgy rendezett egyedi élsúlyok halmazára futtatnánk le. Ennek eredményeként egy fát kapunk. Egy gyakorlatra hagyjuk annak az igazolását, hogy ez a fa valóban az eredeti gráf egy MFF-je (lásd 4-18. gyakorlat).

Vezető folyamat választása.. Ha egyszer egy irányítatlan gráfra épülő hálózatban egy MFF-t (vagy bármilyen feszítőfát) már megismertünk, könnyű egy egyedi vezető folyamatot választani a rendelkezésre álló egyedi azonosítók segítségével. Nevezetesen a feszítőfa levelei egy üzenetet indítanak el a fa útjain a gyökér felé; mindegyik belső csúcs legalább az egyik szomszédjára vár, mielőtt üzenetet küldene a többi szomszédjához. Ha egy csúcs az összes szomszédjától anélkül kap üzenetet, hogy üzenetet küldött volna, vezetőnek nevezi ki magát. Ha két szomszédos csúcs ugyanabban a menetben egymástól kap üzenetet, akkor az egyikük, mondjuk a nagyobb egyedi azonosítójú, nyilvánítja magát vezetőnek. A vezetőfolyamat-választás teljes többlet futási ideje (az MFF felépítés után) csak $\mathcal{O}(n)$, és ehhez $\mathcal{O}(n)$ üzenet küldése szükséges.

Összesítve ezt az MFF bonyolultságelemzésével, azt látjuk, hogy egy élsúlyozott irányítatlan gráffal indulva, amelyben a csúcsok ismerik az n-t (de az *átm* értékét nem), $\mathcal{O}(n \log n)$ idő alatt lehet vezetőt választani $\mathcal{O}(n \log n + |E|)$ üzenet küldése mellett.

4.5.. Maximális független halmaz

Az utolsó feladat, amivel ebben a fejezetben foglalkozunk, az irányítatlan gráfok csúcsai közül történő *maximális független halmaznak (MFH)* a kiválasztása. Csúcsoknak egy halmazát akkor nevezzük *független halmaznak*, ha nem tartalmaz szomszédos csúcsokat; egy független halmazt *maximálisnak* mondunk, ha nem lehet újabb csúcs hozzáadásával nagyobb független halmazt készíteni belőle. Megjegyezzük, hogy az irányítatlan gráfoknak számos különböző maximális független halmazuk lehet. Nincs szükségünk a lehető legnagyobb maximális független halmazra, bármelyik megteszi.

Az MFH feladat aktualitását az a feladat adja, hogyan foglaljuk le osztott erőforrásainkat a hálózatbeli folyamatok számára. A G gráf szomszédai képviselik azokat a folyamatokat, amelyek nem tudják egyidőben végrehajtani valamelyik osztott erőforrást igénylő tevékenységüket (ez lehet például adatbázis elérése vagy rádiós műsorszórás). Ki szeretnénk választani azon folyamatok halmazát, amelyek egyidejű végrehajtása megengedett; ahhoz, hogy az ütközéseket elkerüljük, ezeket a folyamatokat a G egy független halmazába gyűjtjük össze. Nyilván nem kívánatos kizárni ebből egyik folyamatot sem, amelynek egyik szomszédja sem aktív; ezért a kiválasztott halmaznak maximálisnak kellene lennie.

4.5.1.. A feladat

Legyen $G = (V, E)$ egy irányítatlan gráf. A csúcsok egy $I \subseteq V$ halmazát *függetlennek* mondjuk, ha minden $i, j \in I$-re $(i, j) \notin E$. Egy független I halmaz *maximális*, ha bármely olyan I' halmaz, amelynek valódi része az I, nem független. A cél, hogy meghatározzuk G egy maximális független halmazát. Minden folyamatnak, amelynek indexe az I-ben van, végül *győztessé* kell válnia, és mindegyik I-n kívüli indexű folyamatnak *vesztessé*.

Feltesszük, hogy a csúcsok számát, az n-t, az összes folyamat ismeri. (Használhatnánk helyette az n egy felső korlátját is.) A csúcsok most nem rendelkeznek egyedi azonosítókkal.

4.5.2.. Véletlenített algoritmus

Könnyű megmutatni, hogy azon gráfokban, ahol a csúcsokhoz rendelt folyamatok determinisztikusak, az MFH problémát nem lehet megoldani. Ennek igazolása a 3.1. tétel mintájára történik. Ebben az alfejezetben egy olyan egyszerű megoldást mutatunk, amely a véletlenszerűséget használja fel ahhoz, hogy leküzdje ezt a determinisztikus rendszerekben rejlő akadályt. Legyünk azonban korrektek, és ne felejtsük el megemlíteni azt sem, hogy van (nulla) valószínűsége annak, hogy a véletlenített algoritmus nem fog befejeződni. Ezt az algoritmust feltalálója, Luby után, LubyMFH algoritmusnak nevezzük.

A LubyMFH azon az iterációs sémán alapul, amely az adott G gráfból egy tetszőleges nemüres független halmazt választ ki, majd ennek a halmaznak a csúcsait és azok minden szomszédját eltávolítja a gráfból, és ezt a folyamatot ciklikusan ismételi. Ha W egy gráf csúcsainak egy részhalmaza, akkor a $szom(W)$ a W-beli csúcsok szomszédainak halmazát jelöli.

> Legyen a *gráf* a *csúcsok* és az *élek* mezők alkotta rekord, amelyet az eredeti G gráf alapján töltünk fel. Legyen I egy csúcsokat tartalmazó halmaz, amely kezdetben üres.
>
> **while** *gráf.csúcsok* $\neq \emptyset$ **do**
> kiválaszt egy független nemüres $I' \subseteq$ *gráf* halmazt a *gráf*-ban
> $I := I \cup I'$
> *gráf* := a *gráf.csúcsok* $- I' -$ *gráf_szom(I')* által indukált *gráf*-beli részgráf
> **end while**

Egy G gráf csúcsainak egy W részhalmaza által indukált részgráfján azt a gráfot értjük, amelynek csúcshalmaza a W, élei pedig a G gráf W-beli csúcsait összekötő élek.

Könnyű igazolni, hogy ez a séma egy maximális független halmazt állít elő. Ez a halmaz független, mert a kiválasztott I' halmaz minden szakaszban független, és ténylegesen kivesszük a maradék gráfból az összes olyan csúcs szomszédját, amely az I halmazba esik. Ez a halmaz maximális, hiszen csak olyan csúcsokat zárunk ki a további vizsgálatokból, amelyek az I halmazba került csúcsok szomszédjai.

Ennek az általános sémának egy osztott hálózatban történő megvalósításánál az a kulcskérdés, hogyan választható ki az egyes iterációkban az I' halmaz. Ehhez használjuk a véletlenített módszert. Minden szakaszban minden P_i folyamat kiválaszt egy egyenletes eloszlású véletlen *érték*$_i$ egész számot az $\{1, ..., n^4\}$ készletből. Az n^4 használatát az indokolja, hogy elég nagy ahhoz, hogy a gráf összes folyamata nagy valószínűséggel különböző számot válasszon. (Ebben a könyvben nem térünk ki e valószínűség kiszámolására, helyette felhívjuk a figyelmet Luby cikkére.) Miután a folyamatok kiválasztják ezeket az értékeket, az I'-be, definíció szerint, azok az i indexű csúcsok kerülnek, amelyek helyi győztesek, azaz a i csúcs minden j indexű szomszédos csúcsára fennáll, hogy *érték*$_i$ > *érték*$_j$. Ez nyilvánvalóan egy független halmazhoz vezet, mivel két szomszédos csúcs egyszerre nem győzheti le egymást.

Ebben a megvalósításban előfordulhat, hogy a véletlen választások olyan szerencsétlenül alakulnak, hogy az I' halmaz néhány szakaszon keresztül üres lesz; ezek a szakaszok tehát haszontalanok, hiszen semmit sem csinálnak. Feltéve, hogy az algoritmus nem ér el egy olyan ponthoz, amelytől kezdve csupa ilyen haszontalan szakaszokat hajt végre, egyszerűen figyelmen kívül hagyjuk a haszontalan szakaszokat és kikötjük, hogy a LubyMFH algoritmus helyesen kövesse az általános sémát. Az elemzésbe azonban bele fogjuk számolni a haszontalan szakaszokat is. Az algoritmus az alábbi.

LubyMFH algoritmus (vázlatosan)
Az algoritmus működése szakaszokra bontható, minden szakasz három menetből áll.
1. menet. Egy szakasz első menetében a folyamatok kiválasztanak maguknak egy saját *érték*-et, és elküldik azt minden szomszédjuknak. Az 1. menet végére, amikor az összes *érték* célba ér, a (szomszédjaiknál nagyobb értéket választó) győztesek – azaz az I'-beli folyamatok – ismerni fogják magukat.
2. menet. A második menetben a győztesek megismerik a szomszédjaikat. A 2. menet végére a vesztesek – azaz azok a folyamatok, amelyeknek van I'-beli szomszédjuk – meghatározzák magukat.
3. menet. A harmadik menetben mindegyik vesztes megismeri a szomszédjait. Aztán minden érintett folyamat – győztesek, vesztesek, és a vesztesek szomszédjai – törli a megfelelő csúcsot és az ahhoz tartozó éleket a gráfból. Pontosabban, a győztesek és vesztesek a további szakaszokban már nem vesznek részt, és a vesztesek szomszédjai törölnek minden olyan élt, amely a most törölt csúcsokhoz vezet.

Most modellünk segítségével formalizáljuk az algoritmust. Mint azt a 2.7.

alfejezetben már leírtuk, minden folyamat egy speciális $véletlen_i$ véletlen függvényt használ, amelyet minden menetben az $üzenetek_i$ és a $átmenet_i$ függvények végrehajtását megelőzően futtat le. Itt a *random* az $\{1, \ldots, n^4\}$-ból véletlenül, egyenletes eloszlással kiválasztott értéket jelöli.

LubyMFH

állapotok$_i$:
menet$\in \{1, 2, 3\}$, kezdetben 1
érték $\in \{1, \ldots, n^4\}$, kezdetben tetszőleges
éber egy logikai változó, kezdetben *igaz*
marad_szom csúcsok egy halmaza,
kezdetben az eredeti G gráfbeli összes szomszédja
státus $\in \{$*ismeretlen, győztes, vesztes*$\}$, kezdetben *ismeretlen*

véletlen$_i$:
if *éber* és $menet = 1$ **then** *érték* := *random*

üzenetek$_i$:
if *éber* **then**
  case
    $menet = 1$:
      send *érték* a *marad_szom* minden csúcsához
    $menet = 2$:
      if *státus* = *győztes* **then**
        send *győztes* a *marad_szom* minden csúcsához
    $menet = 3$:
      if *státus* = *vesztes* **then**
        send *vesztes* a *marad_szom* minden csúcsához
  endcase

Az alábbi kódban a 3 azonos a 0-val modulo 3.

átmenet$_i$:
if *éber* **then**
  case
    $menet = 1$:
      if *érték*$> v$ minden bejövő v értékre **then** *státus*:=*győztes*
    $menet = 2$:
      if egy *győztes* üzenet érkezett **then** *státus* := *vesztes*
        send *győztes* a *marad_szom* minden csúcsához
    $menet = 3$:
      if *státus* $\in$ {*győztes, vesztes*} **then**
        éber := *hamis*
        marad_szom :=
          marad_szom $-$ {j : j-től egy *vesztes* üzenet érkezett}
  endcase
$menet := (menet + 1)$ modulo 3

Megjegyezzük, hogy a LubyMFH algoritmus akkor is helyesen működik, ha egy-két szakaszban néhány szomszédos folyamat ugyanazon véletlen értékeket választja.

4.5.3.. Elemzés*

Azt már korábban megvitattuk, hogy a LubyMFH algoritmus, feltéve, hogy nem akad el a haszontalan szakaszok újra és újra történő végrehajtásával, egy MFH-t állít elő. Azt állítjuk, hogy az algoritmus egy valószínűséggel nem akad el. Pontosabban, az algoritmus egy tetszőleges szakaszában a megmaradt gráfból eltávolított élek számának várható értéke legalább akkora, mint a megmaradt élek számának egy állandó hányada. Ez maga után vonja azt is, hogy egy szakaszban az élek legalább egy rögzített hányadának eltávolítására egy állandó valószínűséggel kerül sor. Az is következik, hogy a befejeződésig végrehajtott menetek számának várható értéke $\mathcal{O}(\log\ n)$, továbbá, hogy az algoritmus egy valószínűséggel befejeződik.

A LubyMFH algoritmus teljes elemzése Luby eredeti cikkében található. Most bizonyítás nélkül mutatjuk be a legfontosabb lemmákat, és jelezzük, hogyan lehet azokat a szükséges eredmények bizonyításához felhasználni. A következő három lemmához rögzítünk egy $G = (V, E)$ gráfot, amelynek tetszőleges $i \in V$ csúcsára határozzuk meg az alábbi összeget, ahol $d(j)$ a G gráf j-edik csúcsának fokszáma.

$$sum(i) = \sum_{j \in szom_i} \frac{1}{d(j)}$$

4.6. lemma. . *Legyen az I' a* LubyMFH *algoritmus egyik szakaszában szereplő halmaz. Ekkor a szakasz előtt közvetlenül a gráf minden i indexű csúcsára fennáll, hogy*

$$Pr[i \in szom(I')] \geq \frac{1}{4} \min\left(\frac{sum(i)}{2}, 1\right).$$

A gráfból eltávolított élek számának várható értékére a 4.6. lemmát felhasználva kapunk korlátot.

4.7. lemma. . *A* LubyMFH *algoritmus egyetlen szakaszában a G gráfból eltávolított élek számának várható értéke legalább $\frac{|E|}{8}$.*

Bizonyítás. Az algoritmus biztosan elhagy minden olyan élt, amelynek legalább egyik végpontja $szom(I')$-beli. Ebből az következik, hogy az elhagyott élek számának várható értéke legalább

$$\frac{1}{2} \sum_{i \in V} d(i) \cdot Pr[i \in szom(I')].$$

Ez azért van így, mert minden i indexű csúcs bizonyos valószínűséggel rendelkezik egy I'-beli szomszéddal; ha ez az eset fennáll, akkor a i indexű csúcsot eltávolítjuk, ami maga után vonja az összes, $d(i)$ darab hozzá tartozó él törlését is. Az $\frac{1}{2}$ szorzó azt ellensúlyozza, hogy a törölt éleket kétszer számoljuk, hiszen minden élnek két olyan végpontja van, amely a törlését előidézheti.

Most a 4.6. lemmából származó korlátot használjuk fel, és azt kapjuk, hogy az elhagyott élek számának várható értéke legalább

$$\frac{1}{8} \sum_{i \in V} d(i) \cdot \min\left(\frac{sum(i)}{2}, 1\right).$$

Két részre bontva ezt az összeget aszerint, hogy a $sum(i)$ értéke mely i-kre lesz kisebb, mint 2 (és ennek megfelelően a minimalizálandó kifejezés milyen értéket vesz fel), azt kapjuk, hogy

$$\frac{1}{8}\left(\frac{1}{2}\sum_{i:sum(i)<2} d(i)\cdot sum(i) + \sum_{i:sum(i)\geq 2} d(i)\right).$$

Most ide beírjuk a $sum(i)$ definícióját, a $d(i)$-t pedig egy magától értetődő összeg alakjában fejezzük ki.

$$\frac{1}{8}\left(\frac{1}{2}\sum_{i:sum(i)<2}\sum_{j\in szom_i}\frac{d(i)}{d(j)} + \sum_{i:sum(i)<2}\sum_{j\in szom_i} 1\right).$$

Vegyük észre, hogy minden (i,j) irányítatlan él a zárójelen belüli kifejezés két (irányonként egy-egy) tagjában jelenik meg, és ezeknek a tagoknak az összege minden esetben nagyobb, mint 1. Így a teljes kifejezés legalább $\frac{|E|}{8}$.

□

A 4.7. lemmából következik az alábbi állítás:

4.8. lemma. . *A* LubyMFH *algoritmus egyetlen szakaszában a G-ből eltávolított élek száma legalább $\frac{1}{16}$ valószínűséggel legalább $\frac{|E|}{16}$.*

A 4.7. és 4.8. lemmákat felhasználva az alábbi tételhez jutunk:

4.9. tétel. . *A* LubyMFH *algoritmus egy valószínűséggel befejeződik. Továbbá, a befejeződésig végrehajtott menetek száma $\mathcal{O}(\log\ n)$.*

Véletlenített algoritmusok.. Véletlenített algoritmusokkal gyakran találkozhatunk az osztott algoritmusok között. Ezek fő szerepe a szimmetria megtörésében van. Például a vezetőválasztás és az MFH feladatokat nem lehet determinisztikus folyamatok általános gráfjában egyedi azonosítók nélkül megoldani, mivel a szimmetriát lehetetlen megtörni, véletlenített algoritmussal viszont megoldhatók. Ráadásul, ha a folyamatok rendelkeznek is egyedi azonosítókkal, a véletlenített algoritmus gyorsabban szünteti meg a szimmetriát.

A véletlenített algoritmusoknak azonban van egy hibájuk, nevezetesen az, hogy működésük helyességét és/vagy befejeződését csak nagy valószínűséggel tudják garantálni, de nem bizonyosan. Az ilyen algoritmusok tervezésében fontos meggyőződni arról, hogy az algoritmus kritikus tulajdonságait feltétlenül kell-e garantálnunk, vagy csak adott valószínűség mellett. Például a LubyMFH algoritmus bármelyik végrehajtása garantáltan egy független halmazt állít elő, tekintet nélkül a véletlen választások kimenetelére. A befejeződés azonban a szerencsés véletlen választásoktól függ. Lehetséges az is (nulla valószínűségű), hogy mindegyik folyamat újra és újra ugyanazt az értéket választja, így az algoritmus megakad. Az, hogy ezek a jelenségek komoly akadályát képezik-e az algoritmus alkalmazásának, a megoldandó problémától függ.

4.6.. Megjegyzések a fejezethez

A MaxTerjed algoritmust és az OptMaxTerjed algoritmust széles körben ismerik. Teljes szinkron hálózatokban Afek és Gafni [6] találta meg a vezetőfolyamat-választás bonyolultsági korlátjait. A SzinkSZK algoritmus, melyet megtalálhatunk például a [83]-ban, a jól ismert szekvenciális szélességi keresésre épül. A Bellman-Ford algoritmus egy osztott változata a Bellman és Ford által egymástól függetlenül felfedezett szekvenciális algoritmusnak [43, 125].

A SzinkGHS algoritmus egy szinkronizált (és ezért lényegesen egyszerűbb) változata a jól ismert aszinkron MFF algoritmusnak, amelyet Gallager, Humblet, és Spira tervezett. A LubyMFH algoritmus és annak elemzése Luby cikkében [200] található.

A [271]-ben példát találunk egy véletlenített algoritmus (nulla valószínűség mellett bekövetkező) olyan végrehajtására, amelynél a folyamatok rendre ugyanazt az értékválasztást végzik.

4.7.. Gyakorlatok

4-1. Dolgozzuk ki a MaxTerjed algoritmus helyességbizonyításának részleteit.

4-2. Könnyű belátni, hogy a MaxTerjed algoritmusban használt üzenetek $\acute{a}tm|E|$ száma $\mathcal{O}(n^3)$. Állítsunk elő irányított gráfoknak egy osztályát, amelyben az $\acute{a}tm|E|$ szorzat $\Omega(n^3)$, vagy mutassuk meg, hogy nincs az irányított gráfoknak ilyen osztálya.

4-3. Az OptMaxTerjed algoritmus által elküldött üzenetek számára adjunk a $\mathcal{O}(n^3)$-nál kisebb felső korlátot vagy mutassunk egy olyan irányítottgráf-osztályt és megfelelő egyedi azonosítókat, amelyben az üzenetek száma $\Omega(n^3)$.

4-4. Vegyük az OptMaxTerjed algoritmusnak a 4.1.3. szakasz végén azt a „tovább optimalizált" változatát, amely megakadályozza a folyamatokat abban, hogy újra elküldjék a *max_azon* információt azokhoz a folyamatokhoz, amelyektől előzőleg megkapták.

(a) Adjuk meg ennek az algoritmusnak a kódját ugyanabban a stílusban, amellyel ebben a fejezetben a többi kódot is megadtuk.

(b) Bizonyítsuk be ennek az algoritmusnak a helyességét az OptMaxTerjed algoritmus mintájára, felhasználva ugyanazt a fajta egyszerűsítő stratégiát, amit az OptMaxTerjed algoritmus helyességbizonyításánál (azaz a 4.2. tételben) alkalmaztunk.

4-5.

(a) Írjuk meg a SzinkSZK algoritmus kódját.

(b) Bizonyítsuk be invariáns állítások használatával ennek az algoritmusnak a helyességét.

(c) Ismételjük meg az (a) és (b) részeket a gyerek mutatókat használó SzinkSZK algoritmusra.

(d) Ismételjük meg az (a) és (b) részeket a gyerek mutatókat és befejezésjelzéseket használó SzinkSZK algoritmusra.

4-6. Vegyük a SzinkSZK algoritmusnak azt a 4.2.2. szakaszban vázolt optimalizált változatát, amely megakadályozza a folyamatokat abban, hogy elküldjék a `keres` üzenetet azoknak a folyamatoknak, amelyektől előzőleg már kaptak ilyen üzenetet.

(a) Adjuk meg ennek az algoritmusnak a kódját.

(b) Bizonyítsuk be ennek az algoritmusnak a helyességét a SzinkSZK algoritmus mintájára, felhasználva ugyanazt a fajta egyszerűsítő stratégiát, amit az OptMaxTerjed algoritmus helyességbizonyításánál (azaz a 4.2. tételben) alkalmaztunk.

4-7. Részletesen írjunk le egy algoritmust a SzinkSZK algoritmus kiegészítéseként, amely nemcsak a gyerek mutatókat állítja elő, hanem tájékoztat a gyerekekből a szülőkbe vezető legrövidebb útról is. Ezt az információt meg kellene osztani ezen utak mentén, úgy, hogy minden folyamat ismerje az út mentén elhelyezkedő következő folyamatot. Elemezzük a futási időt és a kommunikációs bonyolultságot.

4-8. Részletesen írjunk le egy algoritmust a SzinkSZK algoritmus kiegészítéseként, amely megengedi, hogy a forráscsúcs egy üzenetet küldjön minden más csúcsnak, és értesüljön arról, ha minden csúcs megkapta az üzenetet. Ez az algoritmus $\mathcal{O}(|E|)$ üzenetet küldhet, és $\mathcal{O}(\mathit{átm})$ futási időt használhat. Feltételezhetjük, hogy a hálózat irányítatlan.

4-9. Elemezzük a globális számítási séma, a vezetőválasztás és a 4.2. alfejezet végén szereplő átmérő számítási séma futási idejét és kommunikációs bonyolultságát, feltéve, hogy néhány szomszédos csúcs között kétirányú kommunikációt is megengedünk.

4-10. Tervezzünk egy minél hatékonyabb vezetőválasztó algoritmust egy olyan erősen összefüggő irányított hálózatban, amelyben a folyamatoknak van egyedi azonosítójuk.

(a) Tegyük fel, hogy a kommunikáció minden szomszédos csúcs közt kétirányú, azaz a hálózat irányítatlan.

(b) Ne alkalmazzuk az előző feltevést.

4-11. Adjunk egy minél hatékonyabb algoritmust a csúcsok teljes számának megállapítására egy olyan erősen összefüggő irányított hálózatban, amelyben a folyamatoknak van egyedi azonosítójuk.

(a) Tegyük fel, hogy a kommunikáció minden szomszédos csúcs közt kétirányú, azaz a hálózat irányítatlan.

(b) Ne alkalmazzuk az előző feltevést.

4-12. Adjunk minél hatékonyabb algoritmust az élek számának megállapítására egy olyan erősen összefüggő irányított hálózatban, amelyben a folyamatoknak van egyedi azonosítójuk.

(a) Tegyük fel, hogy a kommunikáció minden szomszédos csúcs közt kétirányú, azaz a hálózat irányítatlan.

(b) Ne alkalmazzuk az előző feltevést.

4-13. Adjunk minél hatékonyabb algoritmust minimális magasságú gyökeres feszítőfa előállítására. Feltételezhetjük, hogy a folyamatoknak van egyedi azonosítójuk, de nincs közöttük kitüntetett csúcs.

4-14.
(a) Adjuk meg a BellmanFord legrövidebb utak algoritmus kódját.
(b) Bizonyítsuk be invariáns állítások használatával ennek helyességét.

4-15. Adjuk meg a SzinkGHS algoritmus kódját.

4-16. Bizonyítsuk be a 4.5. lemmát.

4-17. Mutassuk meg, hogy a SzinkGHS algoritmusban $\mathcal{O}(diam)$ menet nem mindig elég ahhoz, hogy a számítás összes szintjét befejezzük.

4-18. Mutassuk meg, hogy a SzinkGHS algoritmus azon változata (4.4. alfejezet vége), amely élazonosítókat használ az élsúlyok helyett, tényleg egy MFF-t állít elő.

4-19. *Kutatási kérdés.* Fejlesszünk ki a SzinkGHS algoritmusnál – a futási idő vagy a kommunikációs bonyolultság, vagy mindkettő szempontjából – jobb szinkronizált minimális feszítőfa-kereső algoritmust.

4-20. Adjuk meg a kódját annak a 4.4. alfejezet végén vázolt üzenetgyűjtő algoritmusnak, amely egy irányítatlan gráf tetszőleges feszítőfájában vezető folyamatot választ.

4-21. Adjunk minél jobb alsó és felső korlátot egy tetszőleges feszítőfa előállításának feladatára irányítatlan gráfban. Feltételezhetjük, hogy a csúcsok egyedi azonosítóval rendelkeznek és súlyokat nem használunk. Körültekintően állapítsuk meg, hogy a folyamatok milyen ismeretekkel rendelkezzenek a gráfról.

4-22. Vegyünk egy vonalhálózatot, azaz az $1, \ldots, n$ sorszámú folyamatoknak egy olyan lineáris alakzatát, ahol a folyamatok kétirányú kapcsolatban állnak szomszédjaikkal. Tegyük fel, hogy minden egyes P_i folyamat meg tudja különböztetni a bal oldalát a jobb oldalától, és ismeri azt is, hogy ő maga végpont-e vagy sem. Tegyük fel, hogy minden folyamat kezdetben egy nagyon nagy v_i egész értékkel rendelkezik, és azt, hogy az ilyen értékekből egy adott időpillanatban csak adott számút tarthatunk nyilván a memóriában. Tervezzük meg azt az ezen értékeket sorba rendező algoritmust, amelyben az egyes P_i folyamatok által előállított o_i kimeneti értékek összeszorzott halmaza megegyezik a v_i bemeneti értékek összeszorzott halmazával, és $o_1 \leq \ldots \leq o_n$. Próbáljuk meg előállítani mind az üzenetek, mind a menetek száma tekintetében a leghatékonyabb algoritmust. Állításainkat indokoljuk.

4-23. Bizonyítsuk be a 4.5. alfejezet feltételeit alapul véve, de véletlenszerűek helyett determinisztikus folyamatokra, hogy van olyan gráf, amelyben lehetetlen az MFH problémát megoldani. Keressük meg a lehető legnagyobb olyan osztályát a gráfoknak, amelyre ez fennáll.

4-24. Tételezzük fel, hogy a LubyMFH algoritmust egy n méretű gyűrűben hajtjuk végre. Becsüljük meg, mekkora valószínűséggel távolít el az algoritmus egy tetszőleges élt egy ismétlésben (iterációban).

5. fejezet

Megegyezés osztott hálózatokban, vonalhibák esetében

Ebben és a következő két fejezetben a *megegyezés elérésének* problémájával foglalkozunk osztott hálózat esetében. Az effajta problémáknál a hálózati folyamatok egy bizonyos típusú kezdőértékkel indulnak és feltételezhető, hogy végül a kimenetként adott értékük ugyanabba a típusba tartozó érték lesz. A kimenetektől elvárjuk, hogy azonosak legyenek – a folyamatoknak meg kell *egyezniük* –, jóllehet a bemenetek tetszőlegesek lehetnek. Rendszerint adott egy *érvényességi feltétel*, mely leírja a megengedett kimeneti értékeket minden egyes bemeneti mintára.

Ha a rendszer alkotóelemei hibamentesek, akkor a megegyezési problémák általában könnyen megoldhatók egyszerű üzenetcserékkel. Izgalmasabbá válik a témánk, ha a feladatokat hibákat tartalmazó környezetben tanulmányozzuk. Ebben a fejezetben az alapvető megegyezési problémát kommunikációs hibák jelenlétében fogjuk vizsgálni, míg a hatodik fejezet a folyamatok hibáival foglalkozik. A hetedik fejezet az alapfeladat néhány olyan változatát tartalmazza, amelyekben folyamathibák állnak a középpontban.

Megegyezési problémák számos osztott számítógépes alkalmazás során felmerülnek. Például a folyamatok megpróbálhatnak megegyezni abban, hogy egy osztott adatbázisbeli tranzakció eredményét véglegesítsék vagy elvessék; vagy a folyamatok megpróbálhatnak egyetérteni egy repülőgép repülési magasságának megbecslése során, több magasságmérő adataira alapozva döntésüket; vagy megpróbálhatnak egyetérteni abban, hogy különböző folyamatok által végzett vizsgálatok alapján vajon hibásnak mondható-e a rendszer egy alkotóeleme.

Azt a sajátságos megegyezési problémát, amit ebben a fejezetben mutatunk be, *összehangolt támadási problémának* nevezzük; ez a megegyezés elérésének egy alapvető problémája arra az esetre, amikor az üzenetek elveszhetnek. Kezdetnek bemutatunk egy alapvető megoldhatatlansági eredményt determinisztikus rendszerekre vonatkozólag, majd megvizsgáljuk, mire juthatunk a véletlenített megoldás esetében. Megmutatjuk, hogy a feladat megoldható egy véletlenített

algoritmussal, egy bizonyos (bár számottevő) hibaszázalék mellett. Továbbá belátjuk, hogy ez a bizonyos hibaszázalék elkerülhetetlen.

5.1.. Az összehangolt támadási feladat – determinisztikus változat

A feladat vázlatos (így kissé pontatlan) megfogalmazásával kezdjük, egy harctéri ütközet előtti párbeszéd forgatókönyvén keresztül.

> Adottak tábornokok, akik különböző irányokból összehangolt támadást terveznek egy közös célpont ellen. Tudják, hogy a támadás csak akkor lehet sikeres, ha mindannyian támadnak; ha csak néhány tábornok támad, seregeik megsemmisülnek. Minden tábornoknak van egy kezdeti véleménye arról, hogy vajon a serege kész-e a támadásra.

> A tábornokok különböző helyeken tartózkodnak. A szomszédos tábornokok képesek üzenetet váltani, de csak gyalog közlekedő hírnökökön keresztül. A hírnökök azonban eleshetnek vagy fogságba eshetnek, és ennek következtében üzeneteik is elveszhetnek. A kommunikációnak csak ezt a megbízhatatlan formáját használva a tábornokoknak egyeztetniük kell, hogy támadjanak-e vagy sem. Ezenfelül a tábornokoknak támadniuk kell, amennyiben ez lehetséges.
>
> (Feltételezzük egyrészt, hogy a tábornokok „kommunikációs gráfja" irányítatlan és összefüggő gráf, valamint, hogy az összes tábornok ismeri a gráfot. Feltételezzük továbbá azt is, hogy ismert egy felső korlát arra az időre, amennyi idő alatt egy sikeres hírnök kézbesíti az üzenetet.)

Ha mindegyik hírnök megbízható, minden tábornok küldhet üzenetet az összes többi tábornoknak (lehetséges, hogy több lépésben), megüzenve, hogy akar-e támadni. A „kommunikációs gráf" átmérőjével megegyező számú „menet" után minden tábornok ismerni fogja az összes információt. Ezután alkalmazhatják a közösen megállapított szabályt, hogy ugyanazt a döntést hozzák a támadásról. Kizárólag akkor dönthetnek úgy, hogy támadnak, ha az összes többi tábornok is támadni szándékozik.

Egy olyan a modellben, amelyben a hírnökök eltűnhetnek, ez az egyszerű algoritmus nem működik. Kiderül, hogy nem csak az algoritmussal van probléma: bebizonyítjuk, hogy nincs olyan algoritmus, amely mindig hibátlanul megoldja ezt a feladatot.

A leírás mögött álló valóságos számítástudományi probléma az osztott adatbázisok véglegesítési problémája. Ez a feladat folyamatoknak egy csoportját tartalmazza, melyek részt vesznek egy adatbázis-tranzakció feldolgozásában. A feldolgozás után minden folyamat elérkezik egy kezdeti „véleményhez", arról, hogy a tranzakció *véglegessé* váljon-e (azaz, az eredmények tartóssá váljanak-e és szabadon használhatók legyenek-e a többi tranzakció által) vagy *elvessék* (azaz, az eredményeit eldobják). Egy folyamat általában a tranzakció véglegesítését támogatja, ha a tranzakció szerinti összes helyi számítást sikeresen végrehajtotta, máskülönben a tranzakció elvetését javasolja. Feltételezzük, hogy a folyamatok kommunikálnak és végül egyetértenek az eredmények egyikével, a *véglegesítéssel* vagy az *elvetéssel*. Amennyiben lehetséges az eredménynek a *véglegesítésnek* kell lennie.

Mielőtt bebizonyítanánk, hogy nem lehet eredményre jutni, fogalmazzuk meg a problémát formálisan is, megszüntetve a többértelműségeket. Vegyünk n folyamatot, indexeik legyenek rendre $1, \ldots, n$, amelyek egy tetszőleges irányítatlan gráf szerinti hálózatba rendezettek, és minden folyamat ismeri a teljes gráfot, beleértve a folyamatok indexeit is. Mindegyik folyamat egy kiválasztott állapot összetevőjének $\{0, 1\}$-beli bemeneti értékével kezd. Az 1 jelenti a „támadást" vagy *véglegesítést*, a 0 pedig a „nem támadást" vagy *elvetést*. Ugyanazt a szinkron modellt használjuk, mint amivel eddig dolgoztunk, kivéve, hogy most megengedjük, hogy az üzenetek közül néhány elvesszen egy végrehajtás alatt. (Meghatározást lásd a 2.2. alfejezetben.) A cél, hogy végül mindegyik folyamat kiadjon egy dön-

tést a $\{0,1\}$ halmazból, a *döntési* állapot összetevőjét 0 vagy 1 értékre állítva. A folyamatok által hozott döntéssel szemben a következő három elvárásunk van.

Megegyezés. Nem lehet két olyan folyamat, mely különböző érték mellett dönt.

Érvényesség.

1. Ha minden folyamat 0-val kezd, akkor csak a 0 a lehetséges döntési érték.
2. Ha minden folyamat 1-gyel kezd és az összes üzenetet kézbesítették, akkor csak az 1 a lehetséges döntési érték.

Befejeződés. Végül minden folyamatnak döntésre kell jutnia.

A megegyezés és a befejezés követelményei természetesen adódnak. Az érvényesség feltételei csak egy lehetőséget írnak le – itt több használható alternatíva is létezik. Az érvényesség feltételei általában azt az elképzelést fejezik ki, hogy a kialakított döntési érték „ésszerű"; ebben az esetben például az érvényességi feltételek második része kizárja azt a triviális protokollt, hogy a döntés mindig 0. A fentebb megadott érvényességi feltétel meglehetősen gyenge: például ha minden folyamat 1-gyel kezd, akkor elvárható, hogy az algoritmus 1-es döntést hozzon, de ha minden folyamat 1-gyel kezd és akár csak egyetlen üzenet is elveszett, akkor megengedett, hogy az algoritmus döntése 0 legyen. Ez a gyenge szabály is megfelelő, mivel ebben a fejezetben egy megoldhatatlansági eredmény bizonyítására összpontosítunk. Látni fogjuk, hogy a problémának még ezt az egyszerű változatát sem lehet megoldani tetszőleges, legalább két csúcspontot tartalmazó gráf esetében.

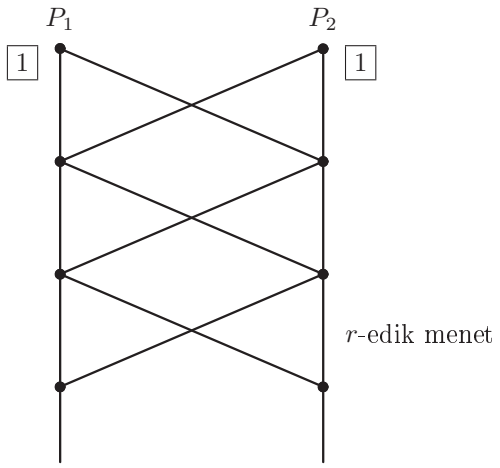
Bebizonyítjuk a megoldhatatlansági eredményt arra az egyedi esetre, amikor két csúcspont egy éllel van összekötve. Az olvasóra bízzuk annak belátását, hogy az erre az esetre vonatkozó megoldhatatlanság magában foglalja a megoldhatatlanságot tetszőleges, legalább két csúcsot tartalmazó gráf esetében. A bizonyításban a végrehajtások megkülönböztethetetlenségének 2. fejezetben megadott formális meghatározását ($\sim$) fogjuk használni.

5.1. tétel. . *Legyen G egy olyan gráf, amelynek csúcsai P_1 és P_2, egyetlen éllel összekötve. Ekkor nincs olyan algoritmus, amely megoldja az összehangolt támadási problémát G-ben.*

Bizonyítás. Indirekt módon bizonyítunk feltételezzük, hogy létezik egy megoldás; legyen ez az A algoritmus. Az általánosság megsértése nélkül feltételezhetjük, hogy minden folyamat esetében egy adott bemeneti értékhez pontosan egy esetében tartozik; ebből következik, hogy a rendszerben pontosan egy végrehajtási sorozat tartozik

a bemenetek egy rögzített elosztásához és az üzenetek egy rögzített mintájához. Szintén az általánosság megszorítása nélkül feltételezhetjük, hogy a folyamatok minden menetben küldenek üzenetet A-ban, mivel rákényszeríthetjük őket arra, hogy álüzenetek küldjenek.

Legyen α az a végrehajtási sorozat, mely akkor valósul meg, amikor mindkét folyamat 1-es értékkel kezd és az összes üzenetet kézbesítették. A befejeződés követelménye szerint végül mindkét folyamat dönt és az érvényességi feltétel második részének megfelelően mindkettő az 1-es értéket választja. Tegyük fel, hogy mindkettő r meneten belül dönt. Most legyen α_1 azonos α-val, kivéve, hogy az első r menet utáni üzenetek elvesznek. α_1-ben szintén mindkét folyamat az 1-et választja r meneten belül. Az α_1 kommunikációs mintája az 5.1. ábrán látható. Az élek olyan üzeneteket jelentenek, amelyeket kézbesítettek; az elküldött, de nem kézbesített üzeneteket egyszerűen nem rajzoljuk meg.

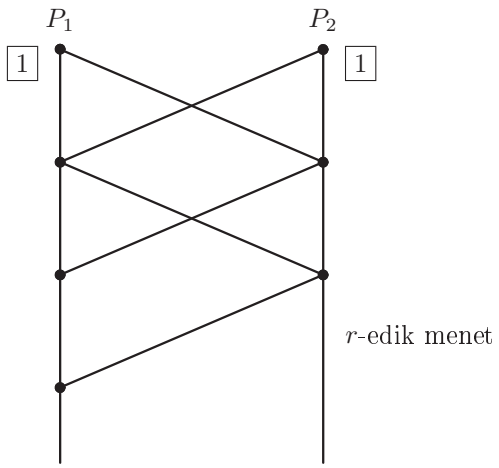


5.1.. ábra. Az üzenetváltások mintája α_1 végrehajtási sorozatban.

α_1-gyel kezdve képezzük a végrehajtási sorozatok egy sorozatát, melynek minden eleme megkülönböztethetetlen a sorban előtte lévőtől az egyes folyamatok szempontjából: ebből az következik, hogy mindezen végrehajtási sorozatok eredménye ugyanaz a döntési érték lesz.

Legyen α_2 ugyanaz a végrehajtási sorozat, mint α_1, kivéve, hogy az utolsó (r-edik) menetben a P_1 folyamatnak a P_2-höz küldött üzenete nem érkezik meg (lásd az 5.2. ábrát). Bár emiatt P_2 az r-edik menet után eltérő állapotba kerülhet α_2 végrehajtási sorozatban, mint az α_1-ben, ezt az eltérést nem tudatja P_1-gyel; ezért $\alpha_1 \overset{1}{\sim} \alpha_2$. Minthogy P_1 döntése α_1-ben 1 volt, 1 lesz α_2-ben is. A befejeződési és megegyezési tulajdonságokból következik, hogy P_2 folyamat is 1-es döntést hoz α_2-ben. Következő lépésben legyen α_3 ugyanaz mint α_2, kivéve, hogy P_2 utolsó P_1-hez küldött üzenete elvész. Minthogy $\alpha_2 \overset{2}{\sim} \alpha_3$, P_2 folyamat 1-est dönt α_3-ban, és a befejeződési és megegyezési tulajdonságok miatt P_1 is így tesz.

Ezt folytatva felváltva eltávolítjuk hol a P_1, hol a P_2 folyamat utolsó üzenetét, végül egy α' végrehajtási sorozathoz jutunk, melyben mindkét folyamat 1-essel kezd és egyetlen üzenet sincs kézbesítve. A fenti okok miatt mindkét folyamat arra kényszerül, hogy ebben az esetben is 1-es döntést hozzon.



5.2.. ábra. Az üzenetváltások mintája α_2 végrehajtási sorozatban.

Vegyünk azonban egy α'' végrehajtási sorozatot, amelyben P_1 folyamat 1-essel kezd, de P_2 folyamat 0-val és nincs kézbesített üzenet. Ekkor $\alpha'' \overset{1}{\sim} \alpha'$ fennáll, következésképp a P_1 folyamat még mindig 1-es mellett dönt és P_2 is így tesz a befejeződési és megegyezési tulajdonságok következtében. Minthogy $\alpha'' \overset{2}{\sim} \alpha'''$, ahol α''' az a végrehajtási sorozat, melyben mindkét folyamat 0-val kezd és nincs kézbesített üzenet, P_2 folyamat 1-est dönt α'''-ban. Ez azonban ellentmondás, mivel az érvényességi feltétel első része szerint α'''-ban mindkét folyamatnak 0-ás döntést kell hoznia.

□

Az 5.1. tétel az osztott hálózatokban rejlő képességek egy alapvető korlátját mutatja be. Azt sugallja, hogy kevés az esélyünk, hogy megoldást találjunk az alapvető megegyezési problémára, csakúgy mint az osztott adatbázisok véglegesítési problémájára, megbízhatatlan kommunikáció esetében. Mindezeken túl a feladat néhány változatát valós rendszerekben meg kell oldani. Az 5.1. tételben kimondott korláton két módon kerekedhetünk felül: vagy megerősítjük a modellt, vagy gyengítünk a feladat követelményein.

Az egyik megközelítés, hogy megtartva a folyamatok determinisztikusságát valószínűségi feltevéseket állítunk az üzenetek elvesztésével kapcsolatosan, majd engedélyeznünk kell a megegyezési és/vagy érvényességi szabályok megsérülését néhány esetben. Egy ilyen környezetben működő algoritmus kifejlesztését meghagyjuk gyakorlatnak(lásd 5-3. gyakorlat). Egy másik megközelítésben megengedjük a folyamatoknak a véletlenszerűség használatát, valamint ismét megengedjük a megegyezési és/vagy érvényességi szabályok megsértését néhány esetben. Ezt a megközelítést tárgyaljuk az 5.2. alfejezetben.

5.2.. Az összehangolt támadási feladat – véletlenített változat

Ebben a részben az összehangolt támadási problémát olyan körülmények között tekintjük át, ahol a folyamatok véletlenszerűen viselkedhetnek. Az előző részhez hasonlóan, vegyünk n folyamatot egy tetszőleges, irányítatlan, ismert gráf szerkezetű hálózatban. Mindegyik folyamat egy kijelölt rendszerösszetevőben beállított $\{0,1\}$-beli bemenő értékkel kezd; feltesszük, hogy mindegyik folyamat egy adott bemeneti értékhez pontosan egy kezdőállapottal rendelkezik. Feltesszük továbbá, hogy a protokoll rögzített számú $r \geq 1$ meneten belül befejeződik, még pontosabban azt, hogy az r-edik menet végére mindegyik folyamat kimenete egy $\{0,1\}$ halmazbeli döntés, amely a folyamat *döntési* változóját a 0 vagy 1 értékre állítja. Minden k, $1 \leq k \leq r$ menetben és mindegyik él mentén történik egy üzenetküldés, ezek közül néhány elveszhet.

A cél lényegében ugyanaz, mint az előbb volt, kivéve, hogy most gyengítünk a problémát leíró állításokon, bizonyos valószínűséggel megengedjük a hiba előfordulását. Nevezetesen, ugyanazt az érvényességi feltételt használjuk, mint az előbb, de gyengítünk a megegyezés feltételén (kis ϵ valószínűséggel megengedjük azt az esetet, amelyben nem jön létre megegyezés). Alsó és felső korlátot adunk ϵ elérhető értékeire az r függvényében, ahol r a menetek száma. Ahogy látni fogjuk, ϵ elérhető értékei nem kicsik.

5.2.1.. Formális modellezés

A feladat formális megfogalmazásához tisztázni kell a valószínűségi állításokat – a helyzet bonyolultabb, mint az MFH problémánál volt a 4.5. alfejezetben. A bonyolultságot az okozza, hogy a megvalósuló végrehajtási sorozat nem csak a véletlenszerű választásoktól függ, hanem attól is, mely üzenetek vesznek el. Nem azt fogjuk feltételezni, hogy az üzenetek elvesztését a véletlen határozza meg. Inkább úgy képzeljük, hogy azt egy „ellenfél" irányítja, amely megpróbálja olyan nehézzé tenni a dolgokat az algoritmus számára, amennyire csak lehet; az algoritmust úgy értékeljük ki, hogy feltesszük a lehetséges legrosszabb viselkedést a szóba jöhető ellenfelek osztályán.

Formálisan, meghatározunk egy *kommunikációs mintát*, amely az alábbi halmaz tetszőleges részhalmaza:

$$\{(i,j,k) : (i,j) \text{ egy éle a gráfnak, és } 1 \leq k\}.$$

A γ-val jelölt kommunikációs minta definíció szerint *jó* lesz, ha $k \leq r$ esetében minden $(i,j,k) \in \gamma$ (ez csak erre a fejezetre vonatkozik – a „jóság" egy másik fogalmát használjuk a 6. fejezetben). Egy jó kommunikációs minta olyan üzenetek halmazát ábrázolja, amelyeket kézbesítettek a végrehajtási sorozat során: ha (i,j,k) eleme a kommunikációs mintának, az azt jelenti, hogy P_i-nek a k-adik menetben P_j számára küldött üzenetét sikeresen átadták.

Az *ellenfél* fogalma, amelyet itt használunk, egy tetszőleges választás az alábbiakból:

1. bemeneti adatok hozzárendelése az összes folyamathoz;
2. egy jó kommunikációs minta.

Egy adott ellenfél esetében a folyamatok véletlenszerű választásai egy-egy egyedi végrehajtási sorozatot határoznak meg. Így minden egyes ellenfél esetében a folyamatok véletlenszerű választása egy valószínűségi eloszlást alkot a végrehajtási sorozatok halmazán. Ezt a valószínűségi eloszlást használva kifejezhetjük a valószínűségét az olyan eseményeknek, mint például: az összes folyamat egyetért. Az ellenfél szerepét hangsúlyozva, a Pr^B jelölést használjuk egy adott B ellenfél valószínűségi függvényének jelölésére.

Nézzük elölről az összehangolt támadási problémát ebben a véletlenített környezetben. Az állításnak paramétere ϵ, $0 \leq \epsilon \leq 1$.

Megegyezés. Minden B ellenfél esetében,

$$Pr^B[\text{a folyamatok közül néhány 0, néhány 1 mellett dönt}] \leq \epsilon \ .$$

Érvényesség. Az előbbivel azonos.

A befejeződésre nem írunk elő feltételt, mivel feltettük, hogy a folyamatok r meneten belül döntést hoznak. Célunk az, hogy találjunk egy algoritmust, a lehető legkisebb ϵ mellett, és bebizonyítsuk, hogy ennél kisebb ϵ érték nem érhető el.

5.2.2.. Egy algoritmus

Az egyszerűség kedvéért ebben és a következő fejezetben figyelmünket az n-csúcspontú teljes gráfok egy egyedi esetére korlátozzuk. Az általános gráfokra való kiterjesztést gyakorlatnak hagyjuk meg. Erre az egyedi esetre bemutatunk egy egyszerű algoritmust, mely eléri az $\epsilon = \frac{1}{r}$ értéket.

Az algoritmus azon alapul, hogy a folyamatok mit tudnak egymás kezdeti értékéről, valamint mit tudnak arról, hogy a többi folyamat mit tud a kezdeti értékekről, és így tovább. Szükségünk van néhány Meghatározásra, hogy a tudás effajta fogalmát megfoghassuk.

Először tetszőleges γ kommunikációs mintára meghatározunk egy $\leq_\gamma$ reflexív parciális rendezést az (i,k) párokra, ahol i egy folyamat indexe, k egy időtényező, $0 \leq k$. (Emlékezzünk vissza a 2. fejezetre, „k időpont" a végrehajtási sorozat azon pontjára vonatkozik, amikor k menet már lezajlott.) Ez a rendezés a különféle folyamatok, különféle időpontjai közötti információáramlást ábrázolja. A reláció definíciója az alábbi:

1. $(i,k) \leq_\gamma (i,k')$ igaz minden i, $1 \leq i \leq n$, és minden k, k', $0 \leq k \leq k'$ esetében;
2. ha $(i,j,k)\,\epsilon\,\gamma$, akkor $(i,k-1) \leq_\gamma (j,k)$;
3. ha $(i,k) \leq_\gamma (i',k')$ és $(i',k') \leq_\gamma (i'',k'')$, akkor $(i,k) \leq_\gamma (i'',k'')$.

Az első pont az azonos folyamaton belüli információáramlást ábrázolja. A második pont a küldőtől az elfogadóhoz áramló információt írja le. A harmadik pont a tranzitivitást definiálja. Az információáramlás hasonló elgondolása jelenik majd meg a könyv későbbi fejezeteiben, például a 14., 16., 18. és 19. fejezetekben.

Most rekurzívan definiáljuk tetszőleges γ jó kommunikációs mintára az *információ szintjét*, legyen ez a $szint_\gamma(i,k)$, ahol i tetszőleges P_i folyamat és k, $0 \leq k \leq r$ tetszőleges időpont. Három eset lehetséges:

1. $k = 0$ esetében
   $szint_\gamma(i,k)$ legyen 0;
2. $k > 0$ és van olyan $j \neq i$, hogy $(j,0) \not\leq_\gamma (i,k)$ esetében
   $szint_\gamma(i,k)$ legyen 0;
3. $k > 0$ és $(j,0) \leq_\gamma (i,k)$ minden $j \neq i$ esetében
   minden $j \neq i$ értékre jelölje l_j a $\max\{szint_\gamma(j,k') : (j,k') \leq_\gamma (i,k)\}$ értéket. (Ez az a legmagasabb szint, ahol elérjük, hogy P_i ismeri P_j-t.) Vegyük észre, hogy $0 \leq l_j \leq k-1$ igaz minden j értékre. Ezután $szint_\gamma(i,k)$ definíció szerint legyen $1 + \min\{l_j : j \neq i\}$.

Más szavakkal, minden egyes folyamat a 0. szinten kezd; amikor értesül az összes többi folyamatról, előre lép az 1. szintre. Amikor értesül arról, hogy az összes többi folyamat elérte az 1. szintet, tovább lép a 2. szintre, és így tovább. Előfordul néhányszor, hogy a $szint_B(i,k)$ jelölést használjuk egy γ kommunikációs mintával megadott B ellenfélre, de ekkor ezen a $szint_\gamma(i,k)$ információs szintet értjük.

5.2.1. példa. Az információ szintje

Tegyük fel, hogy $n = 2$ és $r = 6$. Legyen γ egy jó kommunikációs minta, mely pontosan a következő hármasokból áll:

$$(1,2,1),(1,2,2),(2,1,2),(1,2,3),(2,1,4),(1,2,5),(2,1,5),(1,2,6)$$

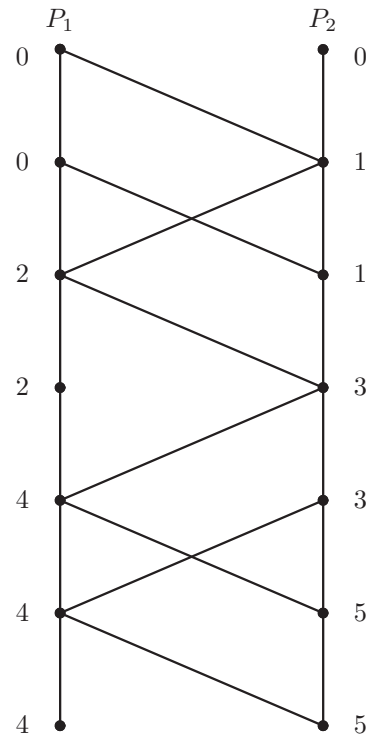
Az 5.3. ábra mutatja a γ kommunikációs mintát. A P_1 és P_2 folyamatok információ szintjét a k, $0 \leq k \leq 6$ időpontokban a címkék jelzik.

A következő lemma kimondja, hogy a különböző folyamatok információ szintje közötti eltérés nem lehet egynél nagyobb.

5.2. lemma. . *Bármely γ jó kommunikációs minta és bármely k, $0 \leq k \leq r$ és bármely i és j esetében $|szint_\gamma(i,k) - szint_\gamma(j,k)| \leq 1$.*

Bizonyítás. A bizonyítást meghagyjuk gyakorlatnak. □

A következő lemma azt mondja ki, hogy abban az esetben, amikor minden üzenet sikeresen eljut a címzetthez, az információ szintje a menetek számával egyenlő.



5.3.. ábra. A γ jó kommunikációs minta.

5.3. lemma. . *Amennyiben γ egy „teljes" kommunikációs minta, mely az összes (i, j, k) hármast tartalmazza $1 \leq k \leq r$ esetében, akkor $szint_\gamma(i, k) = k$ minden i és k értékre.*

Bizonyítás. A bizonyítást meghagyjuk gyakorlatnak (lásd 5-6. gyakorlat). □

A VÉLETLENÍTETTTÁMADÁS nevű algoritmus alapgondolata a következő.

VÉLETLENÍTETTTÁMADÁS algoritmus (vázlatosan)

A *szint* változó tárolja, hogy az egyes folyamatok (jelölésük P_i) a végrehajtási sorozatnak megfelelő kommunikációs minta szerint melyik szintnél tartanak. P_1 folyamat választ egy véletlen értéket, nevezzük ezt *kulcsnak*, ami egy egész szám az $[1, r]$ intervallumból. Ezt az értéket az algoritmus végigcipeli az összes üzeneten. Emellett a folyamatok kezdeti értékét is végigcipeljük az összes üzeneten.

Az r-edik menet után az egyes folyamatok döntése pontosan akkor lesz 1, ha a folyamat szintje legalább akkora, mint a *kulcs* értéke és a folyamat tudomására jutott, hogy az összes többi folyamat az 1 kezdeti értékkel indult.

VÉLETLENÍTETTTÁMADÁS algoritmus (formálisan)

Az üzenet ábécéjét az (S, E, k) alakú hármasok alkotják, ahol S vektor minden egyes eleme egy $[0, r]$ intervallumba tartozó egész értéket rendel a megfelelő indexű folyamathoz, hasonlóan E vektor minden egyes eleme a $\{0, 1, nem\_meghatározott\}$ halmazba tartozó értéket rendeli a megfelelő indexű folyamathoz, k értéke pedig vagy egy egész érték az $[1, r]$ intervallumból, vagy lehet *nem_meghatározott*.

állapotok$_i$:
menetek $\in \mathbb{N}$, kezdetben 0
döntés $\in \{ismeretlen, 0, 1\}$, kezdetben *ismeretlen*
kulcs $\in [1, r] \cup$ *nem_meghatározott*, kezdetben *nem_meghatározott*
minden j, $1 \leq j \leq n$ esetében:
    érték$(j) \in \{0, 1, nem\_meghatározott\}$; kezdetben *érték*$(i)$ P_i kezdeti értéke és *érték*(j) = *nem_meghatározott* minden $j \neq i$ esetében
    szint$(j) \in [-1, r]$; kezdetben *szint*$(i) = 0$ és *szint*$(j) = -1$ minden $j \neq i$ esetében

A *szint*(j) változó feladata, hogy nyomon kövesse P_j folyamat azon legmagasabb szintjét, amiről - az üzenetváltásokon keresztül - P_i folyamat már tudomást szerzett. Mielőtt P_i bármit is hall P_j felől minden $j \neq i$ esetében *szint*(j) alapértéke -1. A *véletlen*$_i$ függvényben használt *véletlen* egy $[1, r]$ intervallumba eső, egyenletes eloszlású véletlen értéket jelöl.

véletlen$_i$:
if $i = 1$ **and** *menetek* $= 0$ **then** *kulcs* := *véletlen*

üzenetek$_i$:
küldd el $(S, E,$ *kulcs*$)$ hármast minden P_j folyamatnak, ahol S a *szint* vektor és E az *érték* vektor

átmenet$_i$:
menetek := *menetek* $+ 1$
legyen (S_j, E_j, k_j) a P_j folyamattól érkező üzenet az összes olyan P_j esetében, amelyiktől üzenet érkezett
if van olyan j, amire $k_j \neq$ *nem_meghatározott* **then** *kulcs* := k_j
for minden $j \neq i$ **do**
    if van olyan i', hogy $E_{i'}(j) \neq$ *nem_meghatározott* **then** *érték*$(j) := E_{i'}(j)$
    if van olyan i', hogy $S_{i'}(j) >$ *szint*(j) **then** *szint*$(j) := \max_{i'}\{S_{i'}(j)\}$
szint$(i) := 1 + \min\{$*szint*$(j) : j \neq i\}$
if *menetek* $= r$ **then**
    if *kulcs* $\neq$ *nem_meghatározott* **and** *szint*$(i) \geq$ *kulcs* **and** *érték*$(j) = 1$ minden j esetében
        then *döntés* := 1
        else *döntés* := 0

A kód harmadik sora beállítja a *kulcs* komponenst; az nem okoz hibát, hogy ezt az értékadást többször is végrehajtja a kód, hisz ugyanaz az érték jár körbe *kulcs* értékként. Az ötödik sor a $j \neq i$ indexű folyamatok *érték* komponensét állítja be, ismét az egymásnak ellentmondó hozzárendelés veszélye nélkül. A hatodik sor a $j \neq i$ indexű folyamatok *szint* komponensét frissíti, ezekben tároljuk az egyes folyamatok P_i által már ismert szint értékei közül az éppen aktuális legnagyobbat. Majd P_i frissíti a saját *szint* komponensét, beállítva, hogy eggyel nagyobb legyen, mint a többi folyamatról megtudott érték minimuma. Végül, ha az utolsó menetben vagyunk ($menetek = r$) P_i döntést hoz az előzőleg megadott szabály szerint.

5.4. tétel. . *A* VÉLETLENÍTETTTÁMADÁS *algoritmus megoldja az összehangolt támadási feladat véletlenített változatát* $\epsilon = \frac{1}{r}$ *értékkel.*

Bizonyítás. A bizonyítás kulcsa az a segédállítás, hogy az algoritmus helyesen számítja ki a szint értékeket. Azaz, a VÉLETLENÍTETTTÁMADÁS bármely végrehajtási sorozata esetében, tetszőleges γ jó kommunikációs mintára és tetszőleges k, $0 \leq k \leq r$ értékekre igaz, hogy k menet után bármely i-re $szint(i)_i$ értéke egyenlő $szint_\gamma(i,k)$ értékkel. Úgyszintén, k menet után, ha $szint(i)_i \geq 1$, akkor $kulcs_i$ érték és $érték(j)_i$ érték – az utóbbi minden j esetében – meghatározott, továbbá az előbbi egyenlő a P_1 által választott *kulcs* értékkel, az utóbbiak pedig egyenlőek az egyes folyamatok kezdeti értékével.

Nyilvánvaló, hogy a VÉLETLENÍTETTTÁMADÁS algoritmus mindig befejeződik. Az érvényességet vizsgálva, ha minden folyamatnak 0 a kezdeti értéke, akkor nyilvánvalóan az egyetlen lehetséges döntés a 0 érték. Most tételezzük fel, hogy az összes folyamat az 1 értékkel kezd és minden üzenet megérkezik. Ekkor az 5.3. lemmából, valamint abból a tényből, hogy az algoritmus a szint értékeket helyesen számítja ki, az következik, hogy az r-edik menet azon pontján, ahol a döntés megszületik, minden i esetében $szint(i)_i = r$. Abból, hogy $szint(i)_i = r \geq 1$ az adott pontban, az következik, hogy $kulcs_i$ érték és $érték(j)_i$ érték is meghatározott, az utóbbi minden j esetében. Mivel a *kulcs* lehetséges értékei kisebb vagy egyenlők mint r, az egyetlen lehetséges döntési érték az 1.

Végül tekintsük a megegyezést. Legyen B tetszőleges ellenfél. Megmutatjuk, hogy

$$Pr^B[\text{néhány folyamat döntése 0, és néhány folyamat döntése 1}] \leq \epsilon.$$

Jelölje l_i tetszőleges i esetében a $szint(i)_i$ pillanatnyi értékét akkor, amikor P_i meghozza a döntését (az r-edik menetben). Az 5.2. lemmából következik, hogy l_i értékek egymástól legfeljebb eggyel térhetnek el. Ha a választott *kulcs* érték szigorúan nagyobb, mint $\max\{l_i\}$, vagy volt olyan folyamat, amely a 0 kezdeti értékkel indult, akkor mindegyik folyamat a 0 döntést hozza meg. Másrészről viszont, ha $kulcs \leq \min\{l_i\}$, és mindegyik folyamat kezdeti értéke 1, az összes folyamat az 1 döntést hozza meg. Ezért a „sikertelen megegyezés" csak akkor fordulhat elő, ha $kulcs = \max\{l_i\}$. Ennek az eseménynek a valószínűsége $\frac{1}{r} = \epsilon$, minthogy $\max\{l_i\}$ értéket a B ellenfél határozza meg, és a *kulcs* értéke egyenletes eloszlású a $[0, r]$ intervallumban. □

5.2.2. példa. A VÉLETLENÍTETTTÁMADÁS algoritmus viselkedése

Tekintsük azt az esetet, amikor $n = 2$ és $r = 6$. Tegyük fel, hogy a B ellenfél az 5.2.1. példában megadott γ jó kommunikációs mintát és a folyamatok bemeneteként az 1 értéket szolgáltatja. Legyen $\epsilon = \frac{1}{6}$. Az 5.4. tétel állítása szerint a megegyezés hiányának valószínűsége legfeljebb ϵ. Valójában ennek valószínűsége pontosan ϵ: ha a P_1 által választott *kulcs* értéke 5, akkor P_1 döntése 0, P_2 döntése 1 lesz; ha $kulcs \leq 4$, akkor mindkettőjük döntése 1 lesz; és ha $kulcs = 6$, akkor mindketten a 0 mellett döntenek.

Másrészről viszont, ha az ellenfél a γ kommunikációs mintával együtt a bemenetek egy bármilyen másik kombinációját nyújtja, akkor a sikertelen megegyezés valószínűsége 0, mivel mindkét folyamat döntése 0 lesz.

Az 5.4. tétel bizonyításához használt gondolatainkra támaszkodva beláthatjuk, hogy a VÉLETLENÍTETTTÁMADÁS algoritmus erősebb érvényességi feltételt is kielégít, mint ahogy azt állítottuk. Nevezetesen megmutathatjuk, az alábbiakat.

Érvényesség.

1. Ha van olyan folyamat, amelyik 0-val kezd, a lehetséges döntési érték csak a 0.
2. Tetszőleges olyan B ellenfél esetében, amelyben az összes bemenet értéke 1,
$$Pr^B[\text{minden folyamat 1 mellett dönt}] \geq l\epsilon,$$
ahol l a folyamatok szintjei közül a legkisebb az r időpontban, B ellenfél mellett.

A második tulajdonság hasznos lehet számos alkalmazásnál, például a hadviselés, vagy az osztott adatbázisokban a véglegesítés, ahol kívánatos, hogy előnyben részesüljön a pozitív kimenetel. Ha például csak egy üzenet vész el, az összehangolt támadás valószínűsége garantáltan magas, legalább $\frac{r-1}{r}$. Annak bizonyítását, hogy a VÉLETLENÍTETTTÁMADÁS algoritmus kielégíti az erősebb érvényességi feltételeket, egyszerű gyakorlatnak hagyjuk meg (lásd 5-8. gyakorlat).

5.2.3.. Alsó korlát a megegyezés hiányának valószínűségére

Most belátjuk, hogy az 5.4. tételben adott korlátnál jobbat nem érhetünk el. (Emlékezzünk vissza az előző alfejezetre, hogy csak az n-csúcsú teljes gráfokat vizsgáljuk.)

5.5. tétel. . *Az összehangolt támadási feladat véletlenített változatának tetszőleges r-menetes algoritmusára igaz, hogy a megegyezés hiányának valószínűsége legalább* $\frac{1}{r+1}$.

A fejezet további részében feltesszük, hogy egy bizonyos A algoritmus egy n-csúcsú teljes gráfban megoldja az összehangolt támadási problémát, úgy hogy a megegyezés hiányának valószínűsége ϵ; bebizonyítjuk, hogy $\epsilon \geq \frac{1}{r+1}$.

A tétel bizonyításához egy további meghatározásra van szükségünk. Vegyünk egy B ellenfelet, γ legyen az ellenfél kommunikációs mintája, P_i pedig egy tetszőleges folyamat, definiáljunk egy másik ellenfelet is, nevezzük *ritkít*(B,i)-nek. A *ritkít*(B,i) a B ellenfelet egyszerűen „megritkítja", azaz eltávolítja az olyan információkat, amelyek nem jutnak el P_i folyamathoz B-ben. $B' = $ *ritkít*(B,i) meghatározása így szól:

1. ha $(j,0) \leq_\gamma (i,r)$, akkor P_j bemenete B'-ben ugyanaz, mint B-ben, egyébként pedig 0;
2. a (j,j',k) hármas pontosan akkor eleme a B' kommunikációs mintának, ha eleme a B kommunikációs mintának, és $(j',k) \leq_\gamma (i,r)$.

Azaz B' ellenfél mindazon üzeneteket tartalmazza, amelyekről P_i a B-ben értesült, de a többit nem, és B' azon bemeneti értékeket, amelyekről P_i nem értesült, B-ben 0-nak határozza meg. A következő lemma szerint az ellenfél megritkított változata elegendő, hogy meghatározzuk a kimenetek valószínűségi eloszlását.

5.6. lemma. . *Ha B és B' mindketten ellenfelek, P_i egy folyamat, és* *ritkít*$(B,i) =$ *ritkít*(B',i), *akkor* $Pr^B[P_i$ *döntése* $1] = Pr^{B'}[P_i$ *döntése* $1]$.

Bizonyítás. A bizonyítást meghagyjuk gyakorlatnak (lásd 5-11. gyakorlat). □

Az 5.5. tétel bizonyítása a következő lemmán alapul.

5.7. lemma. . *Legyen B egy tetszőleges ellenfél, amelyre minden bemeneti érték 1, és legyen P_i egy tetszőleges folyamat, akkor*

$$Pr^B[P_i \text{ döntése } 1] \leq \epsilon(szint_B(i,r)+1).$$

Bizonyítás. $szint_B(i,r)$ szerinti indukcióval.

Alapeset: tegyük fel, hogy $szint_B(i,r) = 0$. Legyen $B' = $ *ritkít*(B,i). Ekkor *ritkít*$(B',i) = B' =$ *ritkít*(B,i), ezért az 5.6. lemma szerint

$$Pr^B[P_i \text{ döntése } 1] = Pr^{B'}[P_i \text{ döntése } 1]. \tag{5.1}$$

Mivel $szint_B(i,r) = 0$, van olyan P_j folyamat, $j \neq i$, melyre $(j,0) \not\leq_\gamma (i,r)$, ahol γ a B kommunikációs mintája. Ekkor B' ellenfél a 0 kezdeti értéket rendeli P_j folyamathoz, és kommunikációs mintája nem tartalmaz P_j-nek címzett üzenetet. Ebből az következik, hogy *ritkít*(B',j) az a triviális ellenfél, amelyben minden kezdeti érték 0 és a kommunikációs mintája nem tartalmaz üzeneteket. Jelölje B'' ezt a triviális ellenfelet. Innen *ritkít*$(B'',j) = B'' =$ *ritkít*(B',j), ezért az 5.6. lemma szerint

$$Pr^{B'}[P_j \text{ döntése } 1] = Pr^{B''}[P_j \text{ döntése } 1].$$

Az érvényességi feltételből következik, hogy

$$Pr^{B''}[P_j \text{ döntése } 1] = 0,$$

ezért

$$Pr^{B'}[P_j \text{ döntése } 1] = 0.$$

Mivel a megegyezés hiányának valószínűsége legfeljebb ϵ, azt kapjuk, hogy

$$|Pr^{B'}[P_i \text{ döntése } 1] - Pr^{B'}[P_j \text{ döntése } 1]| \leq \epsilon.$$

Ekkor

$$Pr^{B'}[P_i \text{ döntése } 1] \leq \epsilon,$$

az 5.1. egyenletből pedig az következik, hogy

$$Pr^{B}[P_i \text{ döntése } 1] \leq \epsilon,$$

amire szükségünk volt.

Indukciós lépés: tegyük fel, hogy $szint_B(i,r) = l > 0$, és tegyük fel, hogy a lemma igaz minden olyan szintre, amely kisebb, mint l. Legyen $B' = ritkít(B,i)$. Ekkor az 5.6. lemmából következik, hogy

$$Pr^{B}[P_i \text{ döntése } 1] = Pr^{B'}[P_i \text{ döntése } 1]. \tag{5.2}$$

Minthogy $szint_B(i,r) = l$, a *szint* definiciójából következik, hogy van olyan P_j folyamat, amelyre $szint_{B'}(j,r) \leq l-1$. Az indukciós feltevésből adódik, hogy

$$\begin{aligned} Pr^{B'}[P_j \text{ döntése } 1] &\leq \epsilon(szint_{B'}(j,r)+1) \\ &\leq \epsilon l. \end{aligned}$$

De mivel a megegyezés hiányának a valószínűsége legfeljebb ϵ, így fennáll, hogy

$$|Pr^{B'}[P_i \text{ döntése } 1] - Pr^{B'}[P_j \text{ döntése } 1]| \leq \epsilon.$$

Ekkor

$$Pr^{B'}[P_i \text{ döntése } 1] \leq \epsilon(l+1),$$

az 5.2. egyenletből pedig az következik, hogy

$$Pr^{B}[P_i \text{ döntése } 1] \leq \epsilon(l+1),$$

amire szükségünk volt. □

Most már bebizonyíthatjuk a tételt.

5.5. tétel bizonytása. Legyen B az az ellenfél, amelyre minden bemenet értéke 1, és nem vesznek el üzenetek. Annak valószínűsége, hogy mindegyik folyamat döntése 1 lesz, legfeljebb annyi lehet, mint annak a valószínűsége, hogy a folyamatok egyikének a döntése 1, ami az 5.7. lemma szerint legfeljebb $\epsilon(szint_B(i,r)+1) \leq \epsilon(r+1)$. De az érvényességi feltétel szerint az összes folyamatnak az 1 mellett kell döntenie valamennyi olyan végrehajtási sorozatban, melyet ez az ellenfél állít elő; ennélfogva annak valószínűsége, hogy mindannyiuk döntése 1 lesz, pontosan 1. Ebből következik, hogy $\epsilon(r+1) \geq 1$, amiből viszont adódik, hogy $\epsilon \geq \frac{1}{r+1}$, amit bizonyítani akartunk. □

5.3.. Megjegyzések a fejezethez

Az összehangolt támadási feladat megfogalmazása Graytől származik [142], aki ezt az osztott adatbázisokban előforduló véglegesítési feladat modellezéséhez használja. A determinisztikus modell megoldhatatlansági eredménye is Graynek köszönhető [142].

Az összehangolt támadási feladat véletlenített változatának eredményeit Varghese és Lynch munkájában találhatjuk meg [281].

5.4.. Gyakorlatok

5-1. Mutassuk meg, hogy az összehangolt támadási feladat (determinisztikus változatának) megoldása bármely nem triviális, összefüggő gráf esetében magában foglalja a feladat megoldását arra az egyszerű, két pontból álló gráfra, mely egy éllel van összekötve. (Ebből következik, hogy a feladat megoldhatatlan tetszőleges, nem triviális gráf

esetében.)

5-2. Tekintsük a (determinisztikus) összehangolt támadási feladat következő változatát. Tegyük fel, hogy a hálózat $n > 2$ résztvevőből álló teljes gráf. A befejeződési és érvényességi feltételek az 5.1. alfejezetben leírtakkal azonosak. Azonban a megegyezési feltételt gyengítjük: „Ha van olyan a folyamatok között, amelyik döntése 1, akkor legalább kettőnek 1-est kell döntenie." (Azaz szeretnénk kizárni azt az esetet, amikor egy tábornok magányosan támad, de megengedjük azt, hogy két vagy több tábornok együtt támadjon.) Vajon ez a feladat megoldható, vagy nem? Bizonyítsuk.

5-3. Tekintsük az összehangolt támadási problémát vonalhibák esetében arra az egyszerű esetre, amikor két folyamat egy éllel van összekötve. Tegyük fel, hogy a folyamatok determinisztikusak, de az üzenetrendszer véletlenített, abban az értelemben, hogy mindegyik üzenetnek van egy független p valószínűségi értéke $0 < p < 1$, ami annak a valószínűségét adja meg, hogy az üzenet sikeresen megérkezik. (Ahogy általában, most is megengedjük, hogy a folyamatok menetenként csak egy üzenetet küldjenek.) Tervezzünk ezekkel a beállításokkal olyan algoritmust, mely rögzített r számú meneten belül befejeződik, a megegyezés hiányának valószínűsége legfeljebb ϵ, és ehhez hasonlóan az érvényességi feltétel megsértésének valószínűsége is legfeljebb ϵ. A lehető legkisebb ϵ érték elérésére törekedjünk.

5-4. Az előző gyakorlat kikötései szerinti modellben adjunk alsó korlátot az ϵ értékére, bizonyítsuk be, hogy ez az elérhető legalacsonyabb érték.

5-5. Bizonyítsuk be az 5.2. lemmát.

5-6. Bizonyítsuk be az 5.3. lemmát.

5-7. Bizonyítsuk be nagyon körültekintően az 5.4. tétel bizonyításának első segédtételét, mely szerint a VÉLETLENÍTETTTÁMADÁS algoritmus helyesen számolja ki a *szint* értékeket, és helyesen szállítja a kezdeti értékeket és a kulcsot.

5-8. Bizonyítsuk be a VÉLETLENÍTETTTÁMADÁS algoritmussal kapcsolatos erősebb érvényességi feltételt – melyet az 5.2.2. szakasz végén adtunk meg, azaz bizonyítsuk be a következőket:

(a) ha van egy olyan folyamat, amelyik 0-val kezd, csak 0 lehet a végleges döntés;

(b) tetszőleges olyan B ellenfél esetében, amelyben az összes bemenet értéke 1,

$$Pr^B[\text{minden folyamat 1 mellett dönt}] \geq l\epsilon,$$

ahol l a folyamatok szintjei közül a legkisebb az r időpontban, B ellenfél mellett.

5-9. Általánosítsuk az összehangolt támadási feladat véletlenített változatát úgy, hogy megengedjük ϵ valószínűséggel mind az érvényességi, mind a megegyezési szabályok megsértését. Írjuk át a VÉLETLENÍTETTTÁMADÁS algoritmust úgy, hogy a módosított feltételek mellett elérje a lehető legkisebb ϵ értéket. Végezzünk elemzést.

5-10. Általánosítsuk a VÉLETLENÍTETTTÁMADÁS algoritmust, és az elemzését az általános (nem szükségszerűen teljes), irányítatlan gráfokra.

5-11. Bizonyítsuk be az 5.6. lemmát.

5-12. Általánosítsuk az 5.5. tételben kapott alsó korlátot az általános (nem szükségszerűen teljes), irányítatlan gráfokra.

5-13. Mi történne a fejezetben tárgyalt, véletlenített környezettel kapcsolatos eredményekkel, ha az ellenfél kommunikációs mintája nem lenne előre rögzítve, mint ahogy eddig feltettük, hanem az ellenfél közvetlen irányítással határozhatná meg azt. Pontosabban szólva, tegyük fel, hogy az ellenfél képes arra, hogy megvizsgálja a végrehajtási sorozatot bármely k-adik menettől visszafelé a kezdetig, mielőtt döntene, hogy a k-adik menetbeli üzenetek közül melyek legyenek kézbesítve.

(a) Milyen ϵ korlát garantálható a VÉLETLENÍTETTTÁMADÁS algoritmus esetében a megegyezés hiányára, ilyen közvetlen irányításra képes ellenfelek esetében?

(b) Adhatunk-e valamilyen érdekes alsó korlátot az elérhető ϵ értékekre?

6. fejezet

Egyetértés osztott hálózatokban processzorhibák esetében

Ebben a fejezetben folytatjuk a szinkron modellre vonatkozó egyetértési feladat tárgyalását, melyet az 5. fejezetben kezdtünk el. Most azzal az esettel foglalkozunk, amelyben a folyamatok, és nem a vonalak hibázhatnak. Természetesen értelmesebb lenne fizikai „processzor" hibákról beszélni, mint logikai „folyamat" hibákról, de hogy következetesek maradjunk a könyv többi részében használt szaknyelvhez, a *folyamatok* kifejezést fogjuk használni. Kétfajta hiba modellt vizsgálunk meg: a *megállási hiba* modellt, melyben a folyamat minden előzetes értesítés nélkül megállhat, és a *bizánci hiba* modellt, melyben a folyamat teljesen tetszőlegesen viselkedhet. A megállási hibák a megjósolhatatlan processzor összeomlásokat hivatottak modellezni. A bizánci hibák a processzor egyfajta önkényes, hibás viselkedését modellezik, ide sorolható például, ha a processzoron belül valamelyik önálló egység meghibásodik.

A *bizánci* kifejezést az ilyenfajta hibákra először Lamport, Pease és Shostak használják egy irányt mutató cikkükben, melyben az egyetértési problémát *bizánci tábornokok* példáján keresztül vezetik be. Az 5. fejezet összehangolt támadási problémájához hasonlóan, a bizánci tábornokok megpróbálnak megegyezni, hogy támadjanak-e. Most viszont a tábornokoknak nincs félnivalójuk amiatt, hogy a hírnökök elesnek, hanem a többi áruló tábornok viselkedésétől kell tartaniuk. A *bizánci* jelző egy szójáték eredménye – a történet az ókori Bizáncban játszódik, így lett a néhány áruló módon viselkedő tábornok jelzője „bizánci".

Azokban az egyetértési feladatokban – egyszerűen *megegyezési feladatnak* nevezzük őket –, melyeket ebben a fejezetben tárgyalunk, a folyamatok egyedi kezdő értéküket egy V halmazból veszik. Az összes hibamentesen működő folyamat kimeneti értéke ugyanezen V halmazba tartozik a megegyezési és érvényességi feltételek értelmében. (Az érvényesség feltétele, hogy amikor minden folyamat ugyanazzal a v értékkel kezd, az egyetlen megengedett döntési érték v lehet.)

A megegyezési feladat egy egyszerűsített változata annak a problémának, mely eredetileg a repülőgépek fedélzeti ellenőrző rendszerével kapcsolatban merült fel. Ott processzorok egy csoportjának kell megegyeznie a repülési magasság tekintetében, úgy, hogy mindegyikük egy különálló magasságmérővel áll kapcso-

latban, amely magasságmérők között lehet hibás is. Bizánci megegyezési algoritmusokat alkalmaznak a hibatűrő többprocesszoros hardverek esetében is, amelyekben a processzorok egy kis csoportja ugyanazt a számítást végzi el, és lépésenként megegyezésre kell jutniuk. E redundanciának köszönhetően a processzorok eltűrik az egyik processzor bizánci hibáját. A bizánci algoritmusok hasznosak a hibadiagnózisban is, lehetővé teszik, hogy a processzorok megegyezzenek, melyik hibás közülük (ezért azt vagy helyettesíteni kell, vagy figyelmen kívül kell hagyni).

Mindkét hiba modellünk esetében egy határt kell szabnunk a processzorhibák előfordulásának gyakoriságára. Hogyan írhatók le ezek a korlátok? Egyéb olyan munkákban, ahol processzorhibákat tartalmazó rendszerek elemzésével foglalkoznak, a korlátok valószínűségi eloszlásokon alapulnak, mely eloszlások a hibák előfordulásait határozzák meg. Itt, valószínűség használata helyett, egyszerűen azt feltételezzük, hogy a hibák számát egy előre megadott, rögzített f érték korlátozza. Ez egy egyszerűen kezelhető megkötés, elkerülhetjük vele azt a bonyolultságot, melyet a hiba előfordulások valószínűségi leírása okozna. A gyakorlatban ez a megközelítés élethű lehet olyan értelemben, hogy nem tartjuk valószínűnek azt, hogy több mint f hiba lép fel. Mindamellett ne feledkezzünk meg róla, hogy e feltevésünk egy kissé vitatható: a legtöbb valóságos helyzetben, ha már nagy számú hiba lépett fel, valószínű, hogy még több hiba fog előfordulni. Feltevésünkből, hogy a hibák száma korlátos, az következik, hogy a hibák *korrelációja negatív*, ugyanakkor a valóságban a hibák vagy függetlenek, vagy korrelációjuk pozitív.

A megegyezési feladat definiálása után bemutatunk egy sor algoritmust a megállási hiba és a bizánci hiba kezelésére. Majd bebizonyítunk két alsó korlátot, egyet a processzorok számára, melyek képesek a bizánci hiba problémájának megoldására, egy másikat pedig azon menetek számára, melyek – mindkét hibatípus esetében – szükségesek a megoldáshoz.

6.1.. A feladat

Feltesszük, hogy a hálózat egy n-csúcsú összefüggő, irányítatlan gráf, $P_1, \ldots, P_n$ folyamatokkal, amelyben minden egyes folyamat ismeri a teljes gráfot. Feltesszük, hogy minden egyes folyamat egy adott rendszerösszetevőben tárolt, és egy rögzített V halmazba tartozó bemeneti értékkel kezd; feltesszük továbbá azt, hogy a folyamatok egy adott bemeneti értékéhez egy kezdőállapot tartozik. A cél, hogy a folyamatok – a *döntési* állapot összetevőjüket beállítva – olyan döntést hozzanak, mely a V halmazba tartozó érték. Itt is a 3–5. fejezetben tárgyalt szinkron modellt használjuk, csak most megengedjük, hogy korlátos számú (legfeljebb f) folyamat hibázhat. Ebben a fejezetben feltesszük, hogy a kapcsolatok teljesen megbízhatók – az összes elküldött üzenet megérkezik. Kétfajta folyamathibával foglalkozunk: a megállási hibával és a bizánci hibával.

A megállási hiba modellben a folyamat az algoritmus végrehajtásának bármely pontján egyszerűen leállhat és a továbbiakban már nem tart lépést a többiekkel. Az is előfordulhat, hogy a folyamat az *üzenetküldő lépés közepén* áll le; így abban a menetben, amikor a folyamat leállt, azoknak az üzeneteknek csak

egy *részhalmaza* indul el ténylegesen, amelyeket a folyamat eredetileg el akart küldeni. Előfordulhat processzor-megállás abban a pillanatban is, amikor a folyamat egy adott menetben az üzeneteit már elküldte, de épp az adott menetnek megfelelő átmenet előtt áll.

A megállási hiba modellben a megegyezési feladat helyességi feltételei az alábbiak.

Megegyezés. Nincs két olyan folyamat, mely eltérő döntést hoz.

Érvényesség. Ha mindegyik folyamat ugyanazzal a $v \in V$ kezdeti értékkel kezd, akkor egyedül v a lehetséges döntési érték.

Befejeződés. Végül minden hibamentes folyamat döntést hoz.

A bizánci hiba modellben a folyamat hibájának nemcsak a megállás lehet az oka, hanem az önkényes viselkedés is. Ez azt jelenti, hogy a folyamat induláskor tetszőleges állapotban lehet, nemcsak a kezdőállapotainak egyikében; tetszőleges üzenetet küldhet, nemcsak a saját *üzenet* függvényének megfelelőt, és tetszőleges *állapot átmenetet* hajthat végre, nemcsak az *átmenet* függvényében meghatározott átmeneteket. (Gyakorlatilag – kényelmi okokból – azt az esetet is megengedjük, hogy egy bizánci folyamat teljesen hibátlanul működik.) Az egyetlen megkötés a hibás folyamat viselkedésére, hogy csak azon rendszeralkotókra lehet hatással, amelyek felett irányítási joga van, nevezetesen a saját kimenő üzeneteire és a saját állapotaira. Nem ronthatja el például a többi folyamat állapotát, nem módosíthatja vagy helyettesítheti mással azok üzeneteit.

A bizánci hiba modell megegyezési és érvényességi feltételei kissé eltérnek a megállási hiba modellnél megadottaktól.

Megegyezés. Nincs két olyan hibamentes folyamat, mely eltérő döntést hoz.

Érvényesség. Ha mindegyik hibamentes folyamat ugyanazzal a $v \in V$ kezdeti értékkel kezd, akkor egyedül v a lehetséges döntési érték a hibamentes folyamatoknál.

Befejeződés. A befejeződés feltétele ugyanaz.

A módosított feltételek azt a tényt tükrözik, hogy a bizánci modellben nem adhatunk korlátot a hibás folyamat kezdeti beállításaira és végső döntésére. A bizánci modell szerinti megegyezési problémára *bizánci megegyezési feladat* néven fogunk hivatkozni.

A megállási és a bizánci megegyezési feladat közötti összefüggés.. Az nem igaz, hogy egy olyan algoritmus, amelyik megoldja a bizánci megegyezési problémát, automatikusan megoldja a megegyezési problémát megállási hibák esetében; az a különbség, hogy a megállási hiba modellben megkívánjuk, hogy az összes döntésre jutott folyamat megegyezzen, *még azok is, amelyek utólag meghibásodtak*. Ha kicseréljük a megállási hiba megegyezési feltételét a bizánci hibánál megadottal, akkor már helytálló a fenti következtetés. Egy másik lehetőség az, ha a bizánci algoritmusban az összes hibamentes folyamat mindig egyazon menetben hoz döntést, akkor megállási hibák esetére is működik az algoritmus. A bizonyításokat a 6-1. és 6-2. gyakorlatra hagyjuk.

Erősebb érvényességi feltétel megállási hibára.. Egy másik érvényességi feltétel, melyet a megállási hiba modellben használnak, így szól.

Érvényesség. Tetszőleges folyamat tetszőleges döntési értékére igaz, hogy az kezdeti értéke volt valamelyik folyamatnak.

Könnyen belátható, hogy ez a feltétel magában foglalja az általunk korábban megadott érvényességi feltételt. Ezt az erősebb érvényességi feltételt fogjuk használni a megegyezési feladat általánosításánál, a 7. fejezetben tárgyalt k-megegyezési problémánál. Ebben a fejezetben a korábban megadott gyengébb feltételt használjuk; ez kissé gyengíti az algoritmussal kapcsolatos állításainkat, és kissé erősíti a megoldhatatlansági eredményeinket. E fejezet algoritmusainál külön kitérünk arra, hogy kielégítik-e ezt a szigorúbb érvényességi feltételt.

Bonyolultsági mértékek.. Az időbonyolultság mértékéhez megszámoljuk, hány menet szükséges ahhoz, hogy az összes hibamentes folyamat döntésre jusson. A kommunikációs bonyolultság mértékéhez az elküldött üzenetek számát és az üzenetek bitszámát határozzuk meg; a megállási hiba modellben számításba vesszük az összes folyamat által küldött üzenetet, míg a bizánci hiba modellben csak a hibamentes folyamatokra alapozzuk a számítást. Ennek az az oka, hogy a bizánci modellben a hibás folyamatok kommunikációjára nem tudunk egyszerű korlátot adni.

6.2.. Algoritmusok megállási hibák kezelésére

Ebben az alfejezetben a megállási hiba modell megegyezési problémájára mutatunk be algoritmusokat, speciálisan az n-csúcsú teljes gráfok esetére. Egy alap algoritmussal kezdjük, melyben a folyamatok egyszerűen újra meg újra közzéteszik az összes általuk valaha megkapott értéket. Folytatjuk az alap algoritmus néhány csökkentett bonyolultságú változatával, és végül bemutatunk olyan algoritmusokat, melyek az *exponenciális információgyűjtés* ($EIGY$) néven ismert stratégiát használják. Bár az exponenciális információgyűjtő algoritmusok költségesebbek és némileg bonyolultabbak, a kevésbé jólviselkedő hibamodellekre is kiterjeszthetők.

Megállapodások.. Ebben és a következő részben a v_0 jelölés egy előre meghatározott alapértéket jelöl, mely eleme a V bemeneti halmaznak. Úgyszintén használjuk a b jelölést azon bitek számának felső korlátjára, mely egy V-beli érték ábrázolásához szükséges.

6.2.1.. Egy alapvető algoritmus

A megállási hiba modell megegyezési problémájára van egy igen egyszerű algoritmus, melynek neve HalmazTerjed. A folyamatok egyre csak terjesztik az összes olyan V-beli értéket, melyről valaha tudomást szereztek, és végül egy egyszerű döntési szabály szerint döntenek.

> **HalmazTerjed algoritmus (vázlatosan)**
> Mindegyik folyamat egy W változót használ, melynek értéke a V halmaz egy részhalmaza. Kezdetben a P_i folyamat W változója csak P_i kezdeti értékét tartalmazza. Minden egyes menetben az $(f+1)$-edik menetig bezárólag, a folyamatok elküldik W-t, majd hozzáadják W-hez a kapott üzenet összes elemét.
>
> $f+1$ menet után P_i a következő döntési szabályt alkalmazza. Ha W egy egyelemű halmaz, akkor P_i döntése az az érték, ami a W halmaz eleme; különben P_i a v_0 alapérték mellett dönt.

Az algoritmus kódja a következő.

> **HalmazTerjed algoritmus (vázlatosan)**
> Az üzenet ábécé V halmaz részhalmazaiból áll.
>
> ***állapotok***$_i$:
> *menetek* $\in \mathbb{N}$, kezdetben 0
> *döntés* $\in V \cup \{$*ismeretlen*$\}$, kezdetben *ismeretlen*
> $W \subseteq V$, kezdetben egyelemű halmaz, P_i kezdeti értékével
>
> ***üzenetek***$_i$:
> **if** *menetek* $\leq f$ **then** küldd el W-t az összes többi folyamatnak
>
> ***átmenet***$_i$:
> *menetek* $:=$ *menetek* $+ 1$
> legyen X_j a P_j-től érkezett üzenet minden olyan j esetében,
> amelyre P_j-től érkezett üzenet
> $W := W \cup \bigcup_j X_j$
> **if** *menetek* $= f + 1$ **then**
> **if** $|W| = 1$ **then** *döntés* $:= v$, ahol $W = \{v\}$
> **else** *döntés* $:= v_0$

A HalmazTerjed algoritmus helyességének tárgyalása során $W_i(r)$ a P_i folyamat W változójának r menet utáni értékét fogja jelölni. Ahogy általában, az i index a P_i folyamathoz tartozó rendszeralkotó jelölésére szolgál. Azt mondjuk, hogy a folyamat r menet után *aktív*, ha az r-edik menet végéig nem hibázik.

Az első egyszerű lemma szerint, ha létezik olyan menet, amelyben egyik folyamat sem hibázik, akkor a menet végén az összes aktív folyamat W halmaza megegyezik.

6.1. lemma. . *Ha egy folyamat sem hibázik egy adott r-edik $(1 \leq r \leq f+1)$ menetben, akkor $W_i(r) = W_j(r)$ minden olyan P_i és P_j folyamat esetében, amely az r-edik menet után aktív.*

Bizonyítás. Tegyük fel, hogy az r-edik menetben egy folyamat sem hibázik, és legyen I a folyamatoknak azon halmaza, amelyek aktívak az r-edik menet után (vagy ennek megfelelően, az $(r-1)$-edik menet után). Azt követően, hogy az összes I-beli folyamat elküldi a saját W halmazát az összes többi folyamatnak, az r-edik menet végén mindegyik I-beli folyamat W halmaza pontosan azoknak az értékeknek a halmaza, melyek az r-edik menet előtti valamely I-beli folyamat W halmazába tartoztak. □

A következő állításban belátjuk, hogy ha adott r menet után az aktív folyamatok W halmaza megegyezik, akkor ez igaz a rákövetkező menetekben is.

6.2. lemma. . *Tegyük fel, hogy $W_i(r) = W_j(r)$ minden olyan P_i és P_j folyamat esetében, amely az r-edik menet után aktív. Akkor ez igaz lesz bármely r'-edik $(r \leq r' \leq f+1)$ menetben is, azaz $W_i(r') = W_j(r')$ minden olyan P_i és P_j folyamat esetében, amely az r'-edik menet után aktív.*

Bizonyítás. A bizonyítást a 6-3. gyakorlatra hagyjuk. □

A következő lemma döntő fontosságú a megegyezési követelmény szempontjából.

6.3. lemma. . *Ha P_i és P_j folyamatok mindketten aktívak $f+1$ menet után, akkor az $(f+1)$-edik menet végén $W_i = W_j$.*

Bizonyítás. Minthogy legfeljebb f hibás folyamat van, biztosan létezik egy olyan r-edik $(1 \leq r \leq f+1)$ menet, melyben egyetlen folyamat sem hibázott. A 6.1. lemmából következik, hogy $W_i(r) = W_j(r)$ minden olyan P_i és P_j folyamat esetében, amely az r-edik menet után aktív. Ebből a 6.2. lemma szerint következik, hogy $W_i(f+1) = W_j(f+1)$ minden olyan P_i és P_j folyamat esetében, amely az $(f+1)$-edik menet után aktív. □

6.4. tétel. . *A* HalmazTerjed *algoritmus megoldja a megegyezési problémát megállási hibák esetében.*

Bizonyítás. A befejeződés nyilvánvalóan adódik a döntési szabályból. Az érvényesség bizonyításához tételezzük fel, hogy az összes kezdeti érték egyenlő v-vel. Ekkor a folyamatok egyedül a v értéket küldözgetik. A $W_i(f+1)$ halmazok egyike sem üres, minthogy P_i kezdeti értékét tartalmazzák. Ezekből következik, hogy $W_i(f+1)$ egyedül a $\{v\}$ halmazzal lehet egyenlő, amiből a döntési szabály szerint adódik, hogy az egyetlen lehetséges döntési érték a v.

A megegyezést illetően, legyen P_i és P_j két tetszőleges folyamat, mely döntést hozott. Minthogy a döntések csak az $(f+1)$-edik menet végén keletkeznek, ez azt jelenti, hogy P_i és P_j aktívak az $(f+1)$-edik menet után. Ebből a 6.3. lemma szerint következik, hogy $W_i(f+1) = W_j(f+1)$. Ekkor a döntési szabály szerint P_i és P_j azonos döntést hoz. □

Bonyolultságelemzés.. A HALMAZTERJED algoritmusnak pontosan $f+1$ menetre van szüksége, hogy az összes folyamat hibátlan folyamat döntést hozzon. Az üzenetek száma összesen $\mathcal{O}((f+1)n^2)$. Minden egyes üzenet legfeljebb egy n elemű halmazt tartalmaz (minthogy a halmaz minden elemére igaz, hogy valamely folyamat kezdő értéke), így az üzenetenkénti bitszám $\mathcal{O}(nb)$. Ebből adódik, hogy a kommunikáció bitszáma összesen $\mathcal{O}((f+1)n^3b)$.

Másfajta döntési szabály.. A HALMAZTERJED algoritmusnál megadott döntési szabály kicsit önkényes. Mivel a HALMAZTERJED algoritmus garantálja, hogy az összes hibamentes folyamat ugyanazzal a W halmazzal rendelkezik $f+1$ menet után, több döntési szabályt használva is helyesen fog működni az algoritmus mindaddig, míg az összes folyamat ugyanazon szabály alapján dönt. Például, ha a V halmaz elemein értelmezve van egy teljes rendezés, akkor mindegyik folyamat választhatná a W halmaz legkisebb elemét. Ennek a szabálynak megvan az az előnye, hogy kielégíti a 6.1. alfejezet vége felé közölt erősebb érvényességi feltételt. A HALMAZTERJED algoritmusnál megadott döntési szabály nem feltétlenül biztosítja ezen erősebb feltétel szerinti érvényességet, mivel előfordulhat, hogy a döntés alapértékeként használt v_0 egyetlen folyamatnak sem kezdeti értéke.

Összevetés a kommunikációs hibák esetével.. A HALMAZTERJED algoritmus igazolja, hogy a megegyezési feladat megállási hibák esetében megoldható. Vessük össze ezt a pozitív eredményt az összehangolt támadási feladat megoldhatatlanságával kommunikációs hibákat tartalmazó környezetben. (Lásd 5.1. tétel és 5-1. gyakorlat)

6.2.2.. A kommunikációs bonyolultság csökkentése

Az információ mennyisége csökkenthető a HALMAZTERJED algoritmusban szükséges $\mathcal{O}((f+1)n^2)$ üzenethez és $\mathcal{O}((f+1)n^3b)$ bithez képest. Például, a szükséges üzenetek száma lecsökkenthető $2n^2$ értékre, a kommunikációhoz szükséges bitek száma pedig $\mathcal{O}(n^2b)$ értékre, ha a következő egyszerű gondolatmenetet követjük. Vegyük észre, hogy a befejeződésnél elegendő, ha a W_i halmazok elemeit csak akkor ismerik pontosan a folyamatok, ha $|W_i| = 1$, különben elegendő, ha azt a tényt tudják, hogy $|W_i| \geq 2$. Így kézenfekvő, hogy a folyamatoknak csak az általuk megismert *első két értéket* kell közvetíteniük, nem pedig az összes értéket. Ezen a gondolaton alapszik a következő algoritmus.

Az OPTHALMAZTERJED algoritmus (vázlatosan)

A folyamatok ugyanazt teszik, mint a HALMAZTERJED algoritmusban, kivéve, hogy minden egyes P_i folyamat legfeljebb két értéket közvetít. Az első közreadás az első menetben történik, amikor P_i a saját kezdeti értékét

közvetíti. A második közreadás az első olyan r-edik ($2 \leq r \leq f+1$) menetben zajlik, amelyben igaz, hogy P_i már a menet elején ismer egy saját kezdeti értékétől eltérő v értéket (ha egyáltalán létezik ilyen menet). Ekkor P_i közreadja ezt az új v értéket. (Ha két vagy több új érték van ebben a menetben, akkor közülük bármelyiket közvetítheti.)

Ugyanúgy, mint a HalmazTerjed algoritmusban, P_i döntése v lesz, ha a végső W_i halmaz az egyelemű $\{v\}$ halmazzal egyenlő, és v_0 különben.

Bonyolultságelemzés.. Az OptHalmazTerjed algoritmusban a szükséges menetek száma éppúgy $f+1$, mint a HalmazTerjed-ben. Az üzenetek száma legfeljebb $2n^2$, minthogy minden egyes folyamat legfeljebb két nem *null* üzenetet küld a többi folyamatnak. A kommunikáció bitszáma $\mathcal{O}(n^2 b)$.

Az OptHalmazTerjed algoritmus helyességét a a HalmazTerjed algoritmusra vonatkozó *szimulációs relációra* támaszkodva fogjuk bizonyítani, (ehhez hasonló stratégiát használtunk a 4.1.3. szakaszban az OptMaxTerjed algoritmus helyesség-bizonyításánál, felhasználva a MaxTerjed algoritmusra vonatkozó szimulációs kapcsolatát. Ehhez csakúgy, mint HalmazTerjed-nél, meg kell adnunk az OptHalmazTerjed algoritmus leírásához szükséges részleteket, beleértve a következőket: *menetek*, *döntés*, W változó. A $W_i(r)$ jelölést használjuk a HalmazTerjed algoritmusban r menet utáni W_i értékekre, míg az $\mathcal{O}W_i(r)$ jelölést az OptHalmazTerjed algoritmus hasonló értékeire. A következő lemma az üzenetek terjedését írja le a HalmazTerjed algoritmusban.

6.5. lemma. . *Tegyük fel, hogy a* HalmazTerjed *algoritmusban P_i folyamat küld egy üzenetet az $(r+1)$-edik menetben P_j folyamatnak, amelyet az megkap és feldolgoz. Ekkor $W_i(r) \subseteq W_j(r+1)$*

Bizonyítás. A bizonyítást a 6-9. gyakorlatra hagyjuk. □

A következő lemma fogalmazza meg az OptHalmazTerjed algoritmus kulcsfontosságú ritkító tulajdonságát.

6.6. lemma. . *Tegyük fel, hogy az* OptHalmazTerjed *algoritmusban P_i folyamat küld egy üzenetet az $(r+1)$-edik menetben P_j folyamatnak, amelyet az megkap és feldolgoz. Ekkor*

1. *Ha $|OW_i(r)| = 1$, akkor $\mathcal{O}W_i(r) \subseteq OW_j(r+1)$.*
2. *Ha $|OW_i(r)| \geq 2$, akkor $|OW_j(r+1)| \geq 2$.*

Sőt, e két állítás akkor is igaz, ha P_i folyamat hibátlanul működik az első r menetben és nem küld üzentet az $(r+1)$-edik menetben P_j-nek, minthogy az OptHalmazTerjed *algoritmusban nincs arra kikötés, hogy a folyamatnak ilyen üzenetet kellene küldenie.*

Bizonyítás. A bizonyítást a 6-9. gyakorlatra hagyjuk. □

Most futtassuk OptHalmazTerjed és HalmazTerjed algoritmusokat azonos bemenettel és azonos hibázási mintával. Ez azt jelenti, hogy mindkét végrehajtásban ugyanazok a folyamatok ugyanazokban a menetekben hibáznak. Azonkívül, ha P_i az r-edik menetbeli üzeneteinek csak egy részét küldi el az

egyik algoritmusban, akkor ugyanazon folyamatoknak küld üzenetet a másik algoritmusban; még pontosabban, nincs olyan P_j, melynek P_i az egyik algoritmusban küld üzenetet az r-edik menetben, de a másikban nem. Invariáns állításokat adunk a két algoritmus állapotaira vonatkozólag.

6.7. lemma. . *Tetszőleges r $(0 \leq r \leq f+1)$ menet után fennáll*

1. $\mathcal{O}W_i(r) \subseteq W_i(r)$.
2. *Ha* $|W_i(r)| = 1$, *akkor* $\mathcal{O}W_i(r) = W_i(r)$.

Bizonyítás. A bizonyítást a 6-9. gyakorlatra hagyjuk. □

6.8. lemma. . *Tetszőleges r $(0 \leq r \leq f+1)$ menet után fennáll, hogy ha $|W_i(r)| \geq 2$, akkor $|OW_i(r)| \geq 2$.*

Bizonyítás. A bizonyítás teljes indukcióval történik. Az $r = 0$ alapesetre az állítás nyilvánvaló.

Tegyük fel, hogy a lemma igaz egy adott r-re. Megmutatjuk, hogy akkor igaz $(r+1)$-re is. Legyen $|W_i(r+1)| \geq 2$. Ha $|W_i(r)| \geq 2$, akkor az induktív feltevésből adódik, hogy $|OW_i(r)| \geq 2$, ebből viszont az következik, hogy $|OW_i(r+1)| \geq 2$, és ezt akartuk bizonyítani.

Most legyen $|W_i(r)| = 1$. Ekkor 6.7. lemmából következik, hogy $\mathcal{O}W_i(r) = W_i(r)$. Bontsuk ketté a bizonyítást.

1. $|W_j(r)| = 1$ az összes P_j-re, melyektől P_i üzenetet kap a HalmazTerjed algoritmusban, az $(r+1)$-edik menetben.

   Ekkor a 6.7. lemma szerint az összes ilyen P_j-re $\mathcal{O}W_j(r) = W_j(r)$ is fennáll, tehát $|OW_j(r)| = 1$. A 6.6. lemmából következik, hogy minden ilyen P_j-re $\mathcal{O}W_j(r) \subseteq W_i(r+1)$. Ebből következik, hogy $\mathcal{O}W_i(r+1) = W_i(r+1)$, amire az indukciós lépés bizonyításához szükségünk volt.
2. Legyen $|W_j(r)| \geq 2$, valamelyik olyan P_j-re, amelytől P_i üzenetet kap a HalmazTerjed algoritmusban, az $(r+1)$-edik menetben.

   Ekkor az indukciós feltevésből $|OW_j(r)| \geq 2$ adódik. Majd a 6.6. lemmából következik, hogy $|OW_i(r+1)| \geq 2$, amit igazolni akartunk.

□

6.9. lemma. . *Tetszőleges r $(0 \leq r \leq f+1)$ menet után, a menetek és a döntési változók értékei megegyeznek a* HalmazTerjed *és az* OptHalmazTerjed *algoritmusokban.*

Bizonyításvázlat. A bizonyítás szempontjából az az érdekes, hogy tetszőleges P_i folyamatra igazoljuk, hogy az $(f+1)$-edik menet után mindkét algoritmusban ugyanazt a döntést hozza. Ez következik a 6.7. és 6.8. lemmából, és a két algoritmus döntési szabályaiból. □

6.10. tétel. . *Az* OptHalmazTerjed *algoritmus megoldja a megegyezési problémát megállási hibák esetében.*

Bizonyítás. Az állítás következik 6.9. lemmából és a 6.4. tételből (amely a HalmazTerjed algoritmus helyességének tétele). □

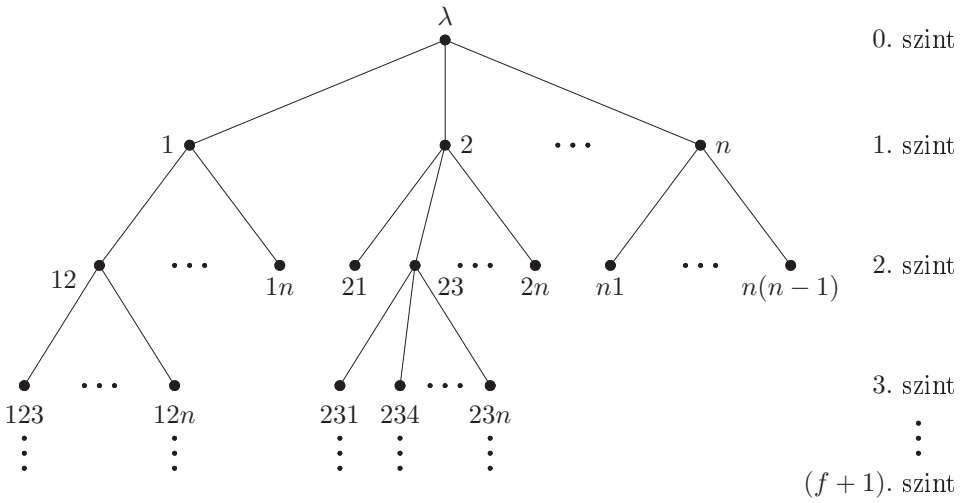
Más módszerek a kommunikációs bonyolultság csökkentésére.. Vannak más lehetőségek is a HALMAZTERJED algoritmus kommunikációs bonyolultságának csökkentésére. Emlékezzünk vissza például arra, hogy ha a V halmazon értelmezve van egy teljes rendezés, a döntési szabályt módosíthatjuk úgy, hogy egyszerűen válasszuk W halmaz legkisebb elemét. Ekkor a HALMAZTERJED algoritmust módosíthatjuk úgy, hogy a csúcspontok csak az általuk megismert legkisebb értéket jegyzik meg és továbbítják, nem az összes értéket. Ennek az algoritmusnak $\mathcal{O}((f+1)n^2b)$ a kommunikációs bitszáma, ami korrekt módon bizonyítható a HALMAZTERJED algoritmusra (a módosított döntési szabállyal) vonatkozó szimulációval. Ez az algoritmus kielégíti a 6.1. fejezetben adott erősebb érvényességi feltételt is.

6.2.3.. Exponenciális információgyűjtő algoritmusok

Ebben a részben olyan, az *exponenciális információgyűjtés* (*EIGY*) stratégián alapuló algoritmusokat mutatunk be, melyek megoldják megegyezési problémát megállási hibák esetében. Az exponenciális információgyűjtésen alapuló algoritmusokban a folyamatok több meneten keresztül küldik és továbbítják a kezdeti értékeket, miközben az általuk megismert, különféle kommunikációs utakon érkezett értékeket egy *EIGY fa* nevű adatszerkezetben rögzítik. Végül egy közösen elfogadott szabály szerint döntenek a fájukban tárolt értékek alapján.

Az *EIGY* algoritmusok általában költségesek a megegyezés megoldásához a megállási hiba modellben, mind a kommunikációhoz szükséges bitek számát tekintve, mind a felhasznált helyi tárolómennyiséget tekintve. A fő ok, amiért itt tárgyaljuk ezt a stratégiát, hogy ugyanezt az *EIGY* fa adatszerkezetet felhasználhatjuk a bizánci megegyezési feladat megoldásához a 6.3.2. szakaszban. A megállási hiba esete bevezet minket az adatszerkezet használatába. A második oka ezen stratégia bemutatásának megállási hibák esetére, hogy az egyszerű *EIGY* megállási hiba algoritmusok könnyen alkalmazhatók a megegyezési feladat megoldására a bizánci hiba modell *hitelesített bizánci hiba* néven ismert, leszűkített változatában. Az *EIGY* algoritmusok alapvető adatszerkezete egy megcímkézett $T = T_{n,f}$ *EIGY fa*, melyben a gyökértől induló utak folyamatok egy láncolatát ábrázolják, melyeken át a kezdeti értékek elterjednek; mindegyik ábrázolt lánc különböző folyamatokból áll. A T fának $f+2$ szintje van, a 0-adik szinttől (a gyökértől) az $(f+1)$-edik szintig (a levelekig). Minden k $(0 \leq k \leq f)$ szinten lévő csúcsnak pontosan $n-k$ gyereke van. T mindegyik csúcsát megcímkézzük egy a folyamatoktól függő szöveggel az alábbi módon. A gyökér címkéje az üres szöveg, jelölje λ, továbbá bármely $i_1 \dots i_k$ szöveggel címkézett csúcsnak pontosan $n-k$ gyereke van, melyek címkéje $i_1 \dots i_k j$, ahol j végigfut $\{1, \dots, n\} - \{i_1, \dots, i_k\}$ elemein. Lásd a 6.1. ábrát.

A megállási hibákra vonatkozó *EIGY* algoritmusban, melyet EIGYSTOP-nak hívunk, a folyamatok az értékeket egyszerűen az összes lehetséges útra szétküldik. Mindegyik folyamat fenntartja az *EIGY* fa egy $T = T_{n,f}$ példányát. A számítás pontosan $f+1$ meneten keresztül zajlik. A számítás folyamán az egyes folyamatok feldíszítik a fájuk csúcsait a V halmazba tartozó értékekkel, vagy a *null* értékkel, a k-adik menet végére k-adik szintig elhelyezkedő összes csúcsot feldíszítik. P_i



6.1.. ábra. A $T_{n,f}$ *EIGY* fa.

fájának gyökerére P_i bemeneti értéke kerül. Amennyiben P_i fájának egy csúcsa az $i_1 \dots i_k$ $(1 \leq k \leq f+1)$ szöveggel van megcímkézve, és a $v \in V$ értékkel feldíszítve, akkor ez azt jelenti, hogy P_{i_k} a k-adik menetben azt mondta P_i-nek, hogy $P_{i_{k-1}}$ a $(k-1)$-edik menetben azt mondta P_{i_k}-nak, hogy ..., hogy P_{i_1} az első menetben azt mondta P_{i_2}-nek, hogy P_{i_1} kezdeti értéke v. Másrészről, ha egy $i_1 \dots i_k$ szöveggel megcímkézett csúcsot a *null* értékkel díszítjük fel, akkor az azt jelenti, hogy a $P_{i_1}, P_{i_2}, \dots, P_{i_k}, P_i$ folyamatok közti kommunikáció hiba miatt megszakadt. $f+1$ menet után a folyamatok a saját maguk által feldíszített fát használják, hogy egy V halmazba tartozó döntést hozzanak, egy (fentebb leírt) közösen elfogadott döntési szabály alapján. Most az algoritmus részletesebb tárgyalása következik.

Az algoritmus most következő leírásában és néhány későbbiben is kényelmesebb, ha úgy tekintjük, mintha valamennyi P_i folyamat önmagának is küldhetne üzenetet a többi folyamatnak küldött üzenet mellett; ez segít bennünket abban, hogy az algoritmusok leírása egységesebb legyen. Ezek az üzenetek a modellben technikailag nincsenek engedélyezve, de kárt nem okozunk, ha megengedjük őket, mert a színlelt átvitelek helyi számításokkal szimulálhatók.

EIGYStop algoritmus (vázlatosan)

Legyen minden folyamatnak, az összes olyan x szövegre, mely a T fa egy csúcsának címkéjeként előfordul, egy *érték*(x) nevű változója. A *érték*(x) változó tárolja azt az értéket, mellyel a folyamat az x címkéjű csúcsot feldíszíti, a *érték*(λ) változót a folyamat saját kezdeti értékére állítja.

Első menet: P_i folyamat elküldi *érték*(λ)-t az összes folyamatnak, beleértve önmagát is. Majd P_i feljegyzi a beérkező adatokat:

1. ha P_i-hez a $v \in V$ üzenetet érkezik P_j-től, akkor P_i a saját *érték*(j) változóját v-re állítja;
2. ha nem érkezik P_i-hez P_j-től olyan üzenet, melynek értéke a V halmazból való, akkor P_i a *érték*(j) változóját *null*-ra állítja.

k-adik menet ($2 \leq k \leq f+1$): P_i szétküldi az (x, v) párokat, ahol x egy T-beli $(k-1)$-edik szinten lévő csúcsnak a címkéje, ami nem tartalmazza az i indexet, $v \in V$ és $v =$ *érték*(x).[1] Majd P_i feljegyzi a beérkező információkat:

1. ha xj a T fában a k-adik szinten lévő csúcsnak a címkéje, ahol x szöveg folyamatok indexeinek sorozata, a j pedig egy index, és a P_i-hez P_j-től érkező üzenet szerint *érték*(x) $= v \in V$, akkor P_i a *érték*(xj)-t v-re állítja;
2. ha xj a T fában a k-adik szinten lévő csúcsnak a címkéje, és nem érkezik P_i-hez olyan üzenet P_j-től, mely szerint *érték*(x) egy V halmazbeli érték, akkor P_i a *érték*(xj) változóját *null*-ra állítja.

Az $(f+1)$-edik menet végén P_i egy döntési szabályt alkalmaz. Nevezetesen, W halmaz elemei legyenek P_i folyamat fáját díszítő nem *null érték*-ek. Ha W halmaz egyelemű, akkor P_i döntése ez az egyedüli elem; egyébként pedig P_i a v_0 döntést hozza.

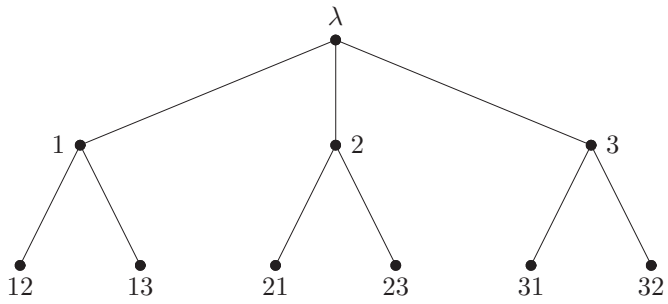
Nem nehéz belátni, hogy a fákat a már korábban bemutatott értékek fogják díszíteni. Tehát a P_i folyamat fájának gyökerét P_i kezdeti értéke díszíti. Továbbá, ha P_i fájának egy csúcsa az $i_1 \dots i_k$, $1 \leq k \leq f+1$ szöveggel van megcímkézve, és a $v \in V$ értékkel feldíszítve, akkor ez azt jelenti, hogy P_{i_k} a k-adik menetben azt mondta P_i-nek, hogy $P_{i_{k-1}}$ a $(k-1)$-edik menetben azt mondta P_{i_k}-nak, hogy …, hogy P_{i_1} az első menetben azt mondta P_{i_2}-nek, hogy P_{i_1} kezdeti értéke v. Valamint, ha P_i fájának egy az $i_1 \dots i_k$, $1 \leq k \leq f+1$ szöveggel megcímkézett csúcsa a *null* értékkel van feldíszítve, akkor az azt jelenti, hogy P_{i_k} nem küldött üzenetet P_i-nek a k-adik menetben, mely továbbította volna az $i_1, \dots, i_{k-1}$-nek megfelelő értéket.

6.2.1. példa. Az EIGYStop algoritmus működése

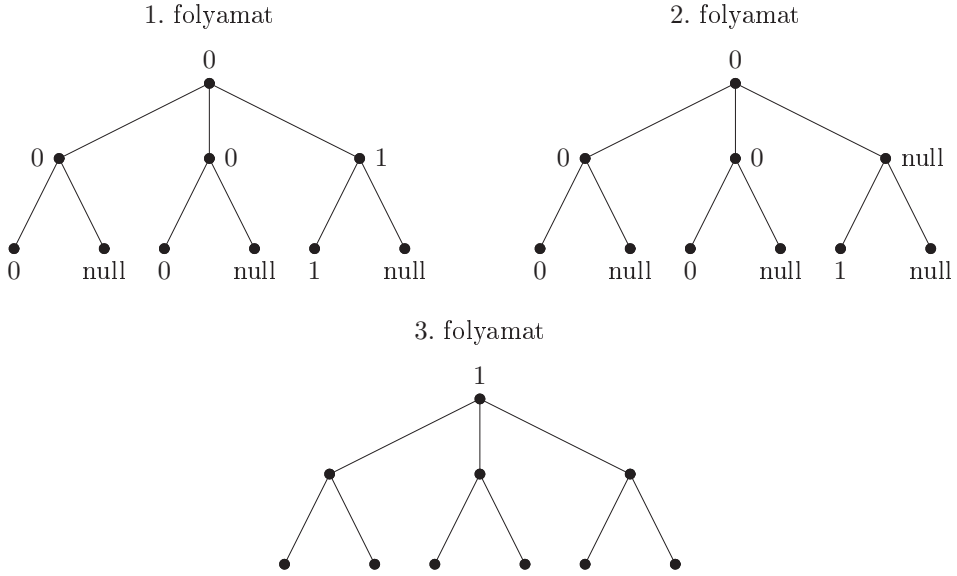
Vizsgáljuk meg egy példán, hogyan működik az EIGYStop algoritmus, tekintsük azt az esetet, amikor három folyamatunk van ($n = 3$), amelyikből az egyik lehet, hogy hibás ($f = 1$). Ekkor a protokoll 2 menetben zajlik, a fának 3 szintje van. A $T_{3,1}$ *EIGY* fa szerkezete a 6.2. ábra szerinti.

Legyen P_1 és P_2 folyamatok kezdeti értéke 0, míg P_3 folyamaté 1. Tegyük fel, hogy P_3 folyamat hibázik: miután az első menetben elküldte az üzenetet P_1-nek, hibássá válik, és P_2-nek nem küld üzenetet. A 6.3. ábra mutatja, hogy ezek után milyen értékekkel lesznek a folyamatok fái kitöltve.

[1] Ha eleget akarunk tenni a formális modellünknek, melyben P_i folyamat csak egy-egy üzenetet küldhet a többieknek egy adott meneten belül, azokat az üzeneteket, melyeknek a címzettje azonos, összecsomagoljuk egy nagy üzenetbe.



6.2.. ábra. A $T_{3,1}$ *EIGY* fa.



6.3.. ábra. Az EIGYSTOP algoritmus végrehajtása; P_3 folyamat hibázik az első menetben.

> Figyeljük meg, hogy P_2-nek egészen addig nem jut tudomására, hogy P_3 folyamat kezdeti értéke 1, amíg a második menetben meg nem hallja P_1-től.

Hogy az EIGYSTOP algoritmus helyességét beláthassuk, először két lemmát mondunk ki, melyek a különböző fák értékeire vonatkoznak. Az első lemma az inicializálást és a fák szomszédos szintjein lévő folyamatokhoz tartozó *érték*-ek közti összefüggést írja le.

6.11. lemma. . *Az* EIGYSTOP *algoritmusban az* $(f+1)$*-edik menet után fenn-*

áll, hogy:

1. *érték*$(\lambda)_i$ P_i *bemeneti értéke;*
2. *ha* xj *egy csúcs címkéje, és* *érték*$(xj)_i = v \in V$*, akkor* *érték*$(x)_j = v$*;*
3. *ha* xj *egy csúcs címkéje, és* *érték*$(xj)_i = null$*, akkor vagy* *érték*$(x)_j = null$*, vagy* P_j *hibás lett, mielőtt* P_i*-nek üzenetet küldött volna az* $(|x| + 1)$*-edik menetben.*

Bizonyítás. A bizonyítást a 6-12. gyakorlatra hagyjuk. □

A második lemma a fa nem feltétlenül szomszédos szintjein lévő csúcsaihoz tartozó *érték*-ek közötti összefüggést írja le. Az első két feltétel a fában tetszőleges helyen szereplő értékek eredetét mutatja. A harmadik egy technikai feltétel, mely azt állítja, hogy bármely a fában megjelenő v-nek meg kell jelennie egy olyan csúcsnál, melynek címkéje nem tartalmazza az i indexet. Lazán fogalmazva, ez azt jelenti, hogy amikor P_i először hall egy adott értékről, az nem lehet annak az eredménye, hogy az értéket saját magának küldte el.

6.12. lemma. . *Az* EIGYSTOP *algoritmusban az* $(f+1)$*-edik menet után fennáll, hogy:*

1. *ha* y *egy csúcs címkéje,* *érték*$(y)_i = v \in V$ *és* xj *az* y *előtagja, akkor* *érték*$(x)_j = v$*;*
2. *ha a* $v \in V$ *érték megjelenik valamely folyamat érték változóinak értékeiből álló halmaz elemeként, akkor van olyan* i*, hogy* $v =$ *érték*$(\lambda)_i$*;*
3. *ha* $v \in V$ *megjelenik, mint a* P_i *folyamat érték változóinak értékeiből álló halmaz egy eleme, akkor van olyan* y *címke, mely nem tartalmazza* i*-t és* $v =$ *érték*$(y)_i$*.*

Bizonyítás. Az állítás első része a 6.11. lemma második részének ismételt alkalmazásával adódik.

A második rész bizonyításához tegyük fel, hogy $v =$ *érték*$(y)_i$. Ha $y = \lambda$, akkor készen vagyunk. Egyébként jelölje j az első indexet y-ban. A lemma első részéből következik, hogy $v =$ *érték*$(\lambda)_j$.

A harmadik rész bizonyításához tegyük fel az állítás ellenkezőjét, hogy v egyedül olyan címkékhez tartozó *érték*-ként jelenik meg, melyekben az i szerepel. Legyen y a legrövidebb olyan címke, melyre $v =$ *érték*$(y)_i$. Ekkor y-nak van egy xi alakú előtagja. De ebből az első rész szerint következik, hogy *érték*$(x)_i = v$, ami ellentmond y kiválasztásának. □

A következő lemma adja a kulcsot a megegyezési feltételhez.

6.13. lemma. . *Ha* P_i *és* P_j *folyamatok hibamentesek, akkor* $W_i = W_j$*.*

Bizonyítás. Feltehetjük, hogy $i \neq j$. Belátjuk, hogy a tartalmazási reláció mindkét irányból fennáll.

1. $W_i \subseteq W_j$.
   Tegyük fel, hogy $v \in W_i$. Ekkor a 6.2. lemmából adódik, hogy $v =$ *érték*$(x)_i$ fennáll egy olyan x esetében, mely az i indexet nem tartalmazza. Tekintsük a következő két esetet.

(a) $|x| \leq f$.
Ekkor $|xi| \leq f + 1$, és mivel az x szöveg nem tartalmazza az i indexet, a (hibamentes) P_i folyamat a $|xi|$-edik menetben közvetíti P_j folyamatnak a v értéket. Ebből következik, hogy $\textit{érték}(xi)_j = v$, tehát $v \in W_j$.

(b) $|x| = f + 1$.
Ekkor, mivel a hibás folyamatok száma legfeljebb f, és az x szövegben az összes index különböző, létezik egy olyan P_l hibamentes folyamat, amelynek indexe szerepel az x-ben. Ezért x-nek van egy yl alakú előtagja, ahol y egy szöveg. Innen a 6.12. lemma szerint adódik, hogy $\textit{érték}(y)_l = v$. Mivel P_l hibamentes, ezért az $|yl|$-edik menetben közvetíti P_j-nek a v értéket. Ebből viszont a $\textit{érték}(yl)_j = v$ adódik, tehát ismét azt kapjuk, hogy $v \in W_j$.

2. $W_j \subseteq W_i$.
Szimmetrikus az előző esetre.

A két eset együttesen bizonyítja a szükséges egyenlőséget. □

6.2.2. példa. A 6.1.3. lemma bizonyításának esetei

A 6.2.1. példa szemlélteti a 6.13. lemmában tárgyalt (a) és (b) esetet. Elsőként a P_1 folyamat díszíti fel a fáját az 1 értékkel az első menetben, és mivel nem ez az utolsó menet, ezért az (a) eset értelmében P_2 a második menetben díszíti fel a fáját az 1 értékkel. Pontosabban $\textit{érték}(3)_1 = 1$, így $\textit{érték}(31)_2 = 1$.

Másrészről viszont, P_2 folyamat először az utolsó, második menetben díszíti a fáját az 1 értékkel, elvégezve a $\textit{érték}(31)_2 = 1$ értékadást. Ebből következik, hogy egy hibamentes folyamat indexe, ebben az esetben az 1, megjelenik a csúcs címkéjében. Innen a (b) eset értelmében az 1 érték a P_1 fájában a 31 címkéjű csúcsnál jelenik meg. Tehát $\textit{érték}(31)_2 = 1$, így $\textit{érték}(31)_1 = 1$.

6.14. tétel. . *Az* EIGYSTOP *algoritmus megoldja a megegyezési problémát megállási hibák esetére.*

Bizonyítás. A befejeződés nyilvánvalóan adódik a döntési szabályból.

Az érvényesség bizonyításához tételezzük fel, hogy az összes kezdeti érték egyenlő v-vel. Ekkor a 6.12. lemma szerint az egyedüli értékek, melyekkel a folyamatok a fáikat díszítik, a v és a *null*. A W_i halmazok egyike sem üres, minthogy P_i kezdeti értékét tartalmazzák. Ebből következik, hogy mindegyik W_i halmaz csak a $\{v\}$ halmazzal lehet egyenlő, amiből a döntési szabály alapján az egyetlen lehetséges döntési érték a v.

A megegyezést illetően, legyen P_i és P_j két tetszőleges folyamat, mely döntést hozott. Minthogy a döntés a végrehajtás végén történik, ez azt jelenti, hogy P_i és P_j folyamatok hibamentesek. Ebből a 6.13. lemma szerint következik, hogy $W_i = W_j$. Ekkor a döntési szabályból adódóan P_i és P_j azonos döntést hoz. □

Bonyolultságelemzés.. A menetek száma $f+1$, az elküldött üzenetek száma $\mathcal{O}((f+1)n^2)$. (Ez az egyesített üzenetek száma, melyet tetszőleges folyamat tetszőleges menetben küld a többi folyamatnak, mint szóló üzenet.) A kommunikáció *bitjeinek* száma exponenciális a hibák számának függvényében: $\mathcal{O}(n^{f+1}b)$.

Másfajta döntési szabály.. Mivel az EIGYStop algoritmus biztosítja, hogy a hibamentes folyamatok fájában előforduló értékek halmaza ugyanaz a W halmaz, több másfajta döntési szabály is helyesen működhet. Például, ha a V halmaz értékein adott egy teljes rendezés, akkor az összes folyamat választhatja a W halmaz legkisebb elemét. Ahogy korábban láttuk, ennek az az előnye, hogy a 6.1. alfejezetben említett erősebb érvényességi feltételt is kielégíti.

A HalmazTerjed algoritmusnál látott módszerrel csökkenthető az EIGYStop algoritmus kommunikáció igénye.

Ahogy már láttuk, elegendő, ha P_i folyamat csak abban az esetben ismeri a W_i halmaz elemeit pontosan, amikor $|W_i| = 1$. Így ismét elegendő, ha a folyamat csak az általa megismert első két értéket közvetíti.

> **Az OptEIGYStop algoritmus (vázlatosan)**
> A folyamatok ugyanazt teszik, mint az EIGYStop algoritmusban, kivéve, hogy minden egyes P_i folyamat legfeljebb két értéket közvetít. Az első közreadás az első menetben történik, amikor P_i a saját kezdeti értékét közvetíti. A második közreadás az első olyan r $(2 \leq r \leq f+1)$ menet valamelyikében zajlik, amelyben igaz, hogy P_i már a menet elején ismer egy saját kezdeti értékétől eltérő v értéket (ha egyáltalán létezik ilyen menet). Ekkor P_i közreadja ezt az új v értéket, azzal az $(r-1)$-edik szinten lévő csúcsnak a címkéjével, mely a v értékkel van feldíszítve. (Ha két vagy több lehetséges (x, v) páros van, akkor közülük bármelyiket választhatja közvetítésre.)
>
> Ugyanúgy, mint az EIGYStop algoritmusban, legyen W azon nem *null értéke*-k halmaza, melyek P_i folyamat fáját díszítik. Ha a W halmaz egyelemű, P_i döntése a W-beli elem lesz, különben pedig v_0.

Bonyolultságelemzés.. Az OptEIGYStop algoritmus $f+1$ menetet használ. Az üzenetek száma legalább $2n^2$, mivel mindegyik folyamat legalább két nem *null* üzenetet küld a többi folyamatnak. $\mathcal{O}(n^2(b+(f+1)\log n))$ a kommunikáció bitjeinek száma: az üzenetek érték részéhez $\mathcal{O}(b)$ bit szükséges, míg a címke rész $\mathcal{O}((f+1)\log n)$ bitből áll.

Az OptEIGYStop algoritmus helyességét az EIGYStop algoritmusra vonatkozó szimulációs kapcsolat alapján bizonyíthatjuk. A bizonyítás hasonló az OptHalmazTerjed algoritmus helyességének bizonyításához. Másik lehetőség, ha az OptEIGYStop algoritmus helyességének bizonyítását az OptHalmazTerjed algoritmusra vonatkoztatva végezzük el. A részleteket a 6-17. gyakorlatra hagyjuk.

6.2.4.. Bizánci megegyezés hitelesítéssel

Bár a fejezetben leírt *EIGY* algoritmusok csak a megállási hibák eltűrésére hivatottak, mégis alkalmasak rosszabb természetű hibák elviselésére is. Nem tudnak

viszont megbirkózni a bizánci hiba modell összes nehézségével, mely modellben a folyamatok önkényesen viselkedhetnek. Mégis, megfelelnek a bizánci hiba modell egy érdekes leszűkítésében, melyben a folyamatoknak egy extra lehetőségük van, *hitelesíthetik* a kommunikációjukat *digitális aláírás* használatával. A P_i folyamat digitális aláírása egy transzformáció, melyet P_i a kimenő üzeneteire alkalmazhat, hogy bizonyíthassa, hogy az üzenet valóban tőle származik. P_i aláírásának előállítására – P_i együttműködése nélkül – egyetlen másik folyamat sem képes. A digitális aláírások valódi lehetőségek, melyek a modern kommunikációs hálózatokban alkalmazhatók.

A hitelesítéses bizánci hiba modell formális definícióját nem adjuk meg – mivel nem ismerünk rá szép definíciót – csak vázlatosan írjuk le. Ebben a modellben feltesszük, hogy a folyamatok digitális aláírást használhatnak, hogy hitelesítsék bármely kimenő üzenetüket. Az irodalomban általában feltételezik, hogy a kezdeti értékek egy közös forrásból származnak, mely szintén aláírással látja el azokat; itt most azt feltételezzük, hogy mindegyik hibamentes folyamat egy egyszerű, a forrás által aláírt bemenő értéket tartalmazó esetébenből indul, míg mindegyik hibás folyamat egy tetszőleges, a forrás által aláírt (szignált) szignált kezdeti értéket tartalmazó tetszőleges állapotból indul. A hibás folyamatok tetszőleges üzeneteket küldhetnek és tetszőleges állapot átmeneteket hajthatnak végre; az egyetlen megkötés, hogy nem képesek a hibamentes folyamatok és a forrás aláírásának előállításra.

A helyesség feltételei ebben a modellben a bizánci megegyezésben szokásos befejeződési és megegyezési feltételek, és a következő érvényességi feltétel.

Érvényesség. Ha mindegyik folyamat pontosan ugyanazon, a forrás által szignált $v \in V$ kezdeti értékkel kezd, akkor csak a v a lehetséges döntési érték a hibamentes folyamatok számára.

Nem nehéz belátni, hogy az EIGYSTOP és OPTEIGYSTOP algoritmusok azzal a módosítással, hogy minden üzenet szignált, és csak a helyesen szignált üzeneteket fogadjuk el, megoldják a megegyezési problémát a hitelesítéses bizánci hiba modellben. A bizonyítás hasonló, mint a megállási hiba modellben, a 6-18. gyakorlatra hagyjuk.

6.3.. Algoritmusok bizánci hibák kezelésére

Ebben a fejezetben, a bizánci megegyezésre mutatunk algoritmusokat, speciálisan n-csúcsú, teljes gráfok esetére. Egy olyan algoritmussal kezdjük, mely az exponenciális információgyűjtést használja. Majd megmutatjuk, hogy egy olyan algoritmus, mely megoldja a bizánci megegyezés problémát egy kételemű $V = \{0, 1\}$ értékhalmazra, hogyan használható „szubrutinként” a bizánci megegyezés megoldására egy általános V értékhalmaz esetében. Végül bemutatunk egy csökkentett kommunikációs bonyolultságú bizánci megegyezés algoritmust.

Ezen algoritmusok közös tulajdonsága, hogy *több, mint háromszor annyi* folyamat szükséges, mint amennyi a hibák száma: $n > 3f$. Ez a körülmény más,

mint a megállási hiba esetében, mivel ott nem volt speciális kikötés az n és f közötti összefüggésre. Ez a korlát a folyamatok számát tekintve jelzi a bizánci hiba modell fokozott bonyolultságát. Sőt, a 6.7. alfejezetben meg fogjuk látni, hogy ez a korlát velejárója a problémának. Ez elsőre meglepőnek tűnhet, mivel úgy képzelnénk, hogy $2f + 1$ folyamat képes eltűrni f bizánci hibát a többségi szavazás algoritmus valamely változatát használva. (Van egy szabvány hibatűrő technika, mely *háromszoros-modul redundancia* néven ismert, abban egy adott feladat három példányban fut és a többségi eredményt fogadjuk el; azt gondolhatnánk, hogy ez használható a bizánci megegyezés megoldásához egy hibás folyamat esetében, de látni fogjuk, hogy ez nem lehetséges.)

6.3.1.. Egy példa

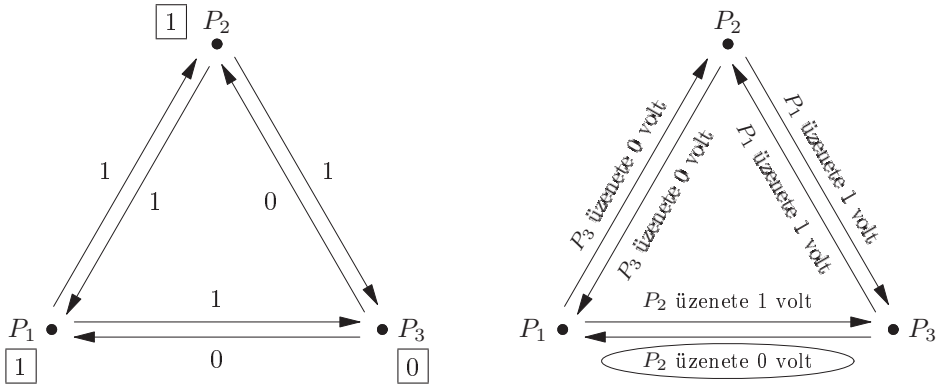
Mielőtt az *EIGY* bizánci megegyezés algoritmust bemutatjuk, gondolkodjunk el azon, hogy miért bonyolultabb a bizánci megegyezési feladat, mint a megegyezési feladat megállási hibák esetében. Egy példán keresztül megvilágítjuk (bár nem bizonyítjuk), hogy három folyamat nem képes megoldani a bizánci megegyezést, ha felmerül annak lehetősége, hogy az egyikük hibázik.

Tegyük fel, hogy P_1, P_2 és P_3 folyamat megoldja a bizánci megegyezési problémát, egy hibát eltűrve. Tegyük fel például, hogy két menet van, a folyamatok a második végén döntenek, valamint egyénre szabott, meghatározott módon cselekszenek: az első menetben mindegyik közreadja a saját kezdeti értékét, míg a második menetben mindegyik jelenti a többinek, hogy mit hallott az első menetben a harmadiktól. Tekintsük a következő végrehajtási sorozatot.

> *Az α_1 végrehajtási sorozat*
> P_1 és P_2 hibátlanok, és az 1 kezdeti értékkel indulnak, P_3 viszont hibázni fog, és a 0 kezdeti értékkel indul. Az első menetben mindegyik folyamat az igazságnak megfelelően küldi el saját értékét. A második menetben P_1 és P_2 az igazságnak megfelelően jelenti, hogy mit hallott az első menetben, míg P_3 a P_1-nek (hibásan) azt üzeni, hogy az első menetben P_2 üzenete 0 volt, egyébként helyesen viselkedik. A 6.4. ábrán láthatjuk az α_1 végrehajtási sorozatban küldött, minket érdeklő üzeneteket. Ebben a végrehajtási sorozatban az érvényességi szabály értelmében P_1 és P_2 mindketten az 1 döntést hozzák.

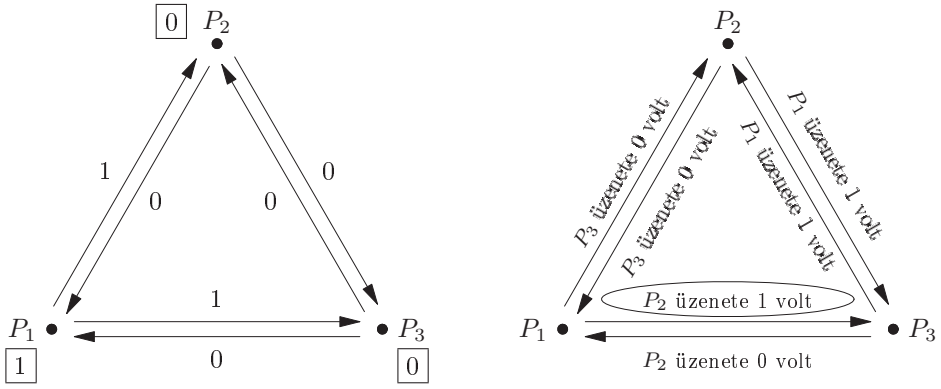
Most tekintsük a második végrehajtási sorozatot.

> *Az α_2 végrehajtási sorozat*
> A végrehajtás szimmetrikus α_1-re. Most P_2 és P_3 hibátlanok, és a 0 kezdeti értékkel indulnak, P_1 viszont hibázni fog, és az 1 kezdeti értékkel indul. Az első menetben mindegyik folyamat az igazságnak megfelelően küldi el saját értékét. A második menetben P_2 és P_3 az igazságnak megfelelően jelenti, hogy mit hallott az első menetben, míg P_1 a P_3-nak (hibásan) azt üzeni, hogy az első menetben P_2 üzenete 1 volt, egyébként helyesen viselkedik. A 6.5. ábrán láthatjuk az α_2 végrehajtási sorozatban küldött, minket érdeklő üzeneteket. Ebben a



6.4.. ábra. Az α_1 végrehajtási sorozat – hibás üzenet kering.

végrehajtási sorozatban az érvényességi szabály értelmében P_2 és P_3 mindketten az 0 döntést hozzák.

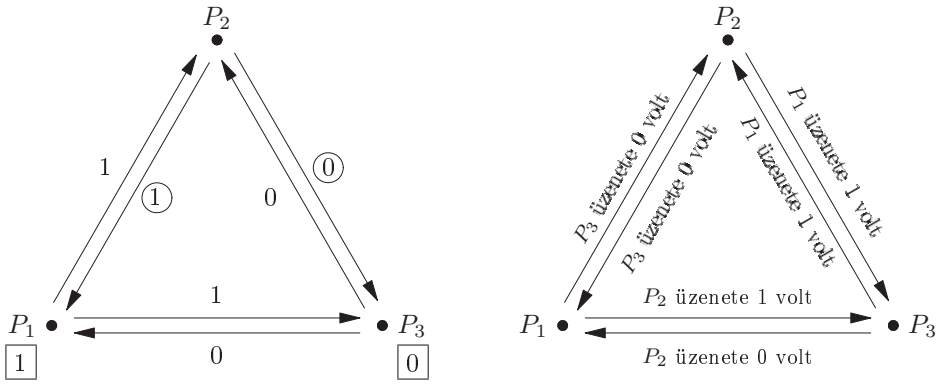


6.5.. ábra. Az α_2 végrehajtási sorozat – hibás üzenet kering.

Nézzük a harmadik végrehajtásti sorozatot, mely ellentmondásra vezet.

Az α_3 végrehajtási sorozat
Most legyen P_1 és P_3 hibátlan, az egyik 1, a másik 0 értékkel indul. P_2 hibásan P_1-nek azt mondja, hogy 1 a kezdeti értéke, P_3-nak pedig azt, hogy 0. A második mentben mindegyik folyamat helyesen viselkedik. A helyzetet a 6.6. ábra mutatja.

Vegyük észre, hogy mindkét menet alatt P_2 ugyanazokat az üzeneteket küldi P_1-nek az α_3-ban, mint az α_1-ben, és ugyanazokat az üzeneteket küldi P_3-nak az α_3-ban, mint az α_2-ben. Ebből könnyen belátható, hogy P_1 számára az α_3



6.6.. ábra. Az α_3 végrehajtási sorozat – egymással ütköző üzenetek keringenek.

és α_1 megkülönböztethetetlen, azaz $\alpha_3 \overset{1}{\sim} \alpha_1$, és hasonlóan $\alpha_3 \overset{3}{\sim} \alpha_2$. Mivel P_1 az α_1-ben 1-est dönt, ugyanígy tesz α_3-ban, és mivel P_3 döntése 0 az α_2-ben, ugyanígy tesz α_3-ban. Ez viszont megsérti a megegyezési szabályt α_3-ban, mely ellentmond annak, hogy P_1, P_2 és P_3 megoldják a bizánci megegyezési problémát. Megmutattuk, hogy ebben a felállásban nincs olyan algoritmus, mely meg tudja oldani a bizánci megegyezést.

Megjegyezzük, hogy P_1 például megmondhatná, hogy α_3-ban valamelyik folyamat hibás, mivel P_2 azt mondja P_1-nek, hogy a kezdeti értéke 1, de P_3 azt mondja P_1-nek, hogy neki P_2 azt mondta, hogy 0 a kezdeti értéke. A probléma ott van, hogy P_1 nem tudja eldönteni hogy P_2 és P_3 közül melyik a hibás.

A példa nem bizonyítja azt, hogy három folyamat nem tudja megoldani a bizánci megegyezést egyszeri hibalehetőség mellett. Ennek az az oka, hogy az érvelés előfeltétele volt, hogy az algoritmus két menetet használ, és az üzenetek egy adott típusba tartoznak. Lehetséges azonban a példa kiterjesztése több menetre és tetszőleges üzenetekre. Meg fogjuk látni a 6.4. alfejezetben, hogy elgondolásunk odáig kiterjeszthető, hogy $n > 3f$ számú folyamat szükséges a bizánci megegyezés megoldásához, f hibát feltételezve.

6.3.2.. Az EIGY algoritmus a bizánci megegyezésre

Most bemutatunk egy *EIGY* algoritmust a bizánci megegyezésre, a neve legyen EIGYBiz. Ellentétben az EIGYStop algoritmussal, EIGYBiz előfeltétele, hogy a folyamatok száma nagy a hibákéhoz képest, nevezetesen $n > 3f$. Ez a 6.3.1. és a 6.4. alfejezetekben leírt korlátok miatt szükséges. Mielőtt elolvasnánk az algoritmus leírását, próbáljunk egy saját megoldást készíteni az $n = 7$ és $f = 2$ esetre.

Az EIGYBiz algoritmus n számú folyamat és f számú hiba esetében ugyanazt a $T_{n,f}$ *EIGY* fa adatszerkezetet használja, mint az EIGYStop. Lényegében ugyanazt a terjesztési stratégiát használjuk, mint az EIGYStop-ban; az egyetlen különbség, hogy az a folyamat, amely egy „helytelen formájú” üzenetet kap,

kijavítja az információt, hogy elfogadható legyen. A döntési szabály egészen más, jóllehet – ez már többé nem az az eset, melyben a folyamatok megbízhatnak az összes olyan értékben, amely a fájukban valahol megjelenik. Most a folyamatok kénytelenek tenni valamit, hogy leleplezzék a hibás üzenetben érkező értékeket.

Az EIGYBiz algoritmus (vázlatosan)

A folyamatok $f+1$ meneten keresztül terjesztik az értékeket, pontosan úgy, mint az EIGYStop algoritmusban, a következő kivételekkel. Ha P_i folyamat valaha is kapna valamely másik P_j-től egy üzenetet, mely nem felel meg a leírásnak (például a tartalma teljesen használhatatlan, vagy P_j fájában ugyanahhoz a csúcshoz kettős értéket rendel), akkor P_i „eldobja" az üzenetet, azaz úgy tesz, mintha P_j semmit sem küldött volna az adott menetben.

Az $(f+1)$-edik menet végén P_i megigazítja a *érték* hozzárendeléseket, a *null* értékeket az alapértékként adott v_0 értékkel helyettesíti.

Ezután, hogy P_i meghatározhassa a döntését, elindul a kiigazított, feldíszített fájában a levelektől felfelé, hogy minden egyes csúcsot feldíszítsen egy *új_érték* értékkel a következők szerint. Minden x címkéjű levélre, *új_érték*(x) := *érték*(x). Minden x címkéjű nem levél csúcsra *új_érték*(x) definíció szerint legyen az az *új_érték*, melyet az x csúcs gyerekeinek szigorúan vett többsége tárol, pontosabban legyen az a $v \in V$ elem, melyre *új_érték*$(xj) = v$ az xj alakú csúcsok többségénél, feltéve, hogy ilyen többség létezik. Ha nem létezik a többség, P_i az *új_érték*$(x) := v_0$ beállítást hajtja végre. P_i végső döntése *új_érték*(λ).

Az EIGYBiz algoritmus helyességének bizonyítását néhány előzetes állítással kezdjük. Az első azt mondja ki, hogy az összes hibátlan folyamat olyan értékek alapján jut megegyezésre, melyek közvetlenül hibátlan folyamatoktól származnak.

6.15. lemma. . *Az* EIGYBiz *algoritmus* $(f+1)$*-edik menete után fennáll a következő. Ha* P_i*,* P_j *és* P_k *hibátlan folyamatok és* $i \neq j$*, akkor* *érték*$(x)_i =$ *érték*$(x)_j$ *minden* x *címkére, mely* k*-ra végződik.*

Bizonyítás. Ha $k \notin \{i, j\}$, akkor az állítás abból a tényből adódik, hogy P_k hibamentes, ezért ugyanazt az üzentet küldi P_i-nek és P_j-nek az $|x|$-edik menetben. Ha $k \in \{i, j\}$, akkor hasonlóan következik abból a megállapodásból, mely szerint a folyamatok saját maguknak is közvetítik az értékeket. □

A következő lemma azt állítja, hogy az összes hibátlan folyamat azon *új_érték* értékek alapján jut megegyezésre, melyeket olyan csúcsokra számítottunk ki, amelyek címkéje egy hibamentes folyamat indexére végződik.

6.16. lemma. . *Az* EIGYBiz *algoritmus* $(f+1)$*-edik menete után fennáll a következő. Tegyük fel, hogy* x *címke egy hibátlan folyamat indexével végződik. Ekkor van olyan* $v \in V$*, hogy* *érték*$(x)_i =$ *új_érték*$(x)_i = v$ *minden hibátlan* P_i *folyamatra teljesül.*

Bizonyítás. A bizonyítás indukcióval történik a fa címkéi alapján, a levelektől felfelé haladva – tehát az $f+1$ hosszúságtól az 1-ig csökkenőleg.

Alapeset: legyen x egy levél, azaz $|x| = f+1$. Ekkor a 6.15. lemmából következik, hogy az összes hibátlan P_i folyamatra a *érték*$(x)_i$ érték ugyanaz; hívjuk ezt a közös értéket v-nek. Továbbá az is igaz, hogy *új_ érték*$(x)_i = v$, minden hibátlan P_i-re, a *új_ érték* levelekre adott definíciója alapján. Így v a kívánt érték.

Indukciós lépés: legyen $|x| = r$, $1 \leq r \leq f$. Ekkor a 6.15. lemmából következik, hogy az összes hibátlan P_i folyamatra a *érték*$(x)_i$ érték ugyanaz; hívjuk ezt az értéket v-nek. Ezért az összes hibátlan P_l folyamat ugyanazt a v, x-hez tartozó értéket küldi el az $(r+1)$-edik menetben, így *érték*$(xl)_i = v$ minden hibamentes P_i és P_l folyamatra. Az indukciós feltevésből következik, hogy az *új_ érték*$(xl)_i = v$ is teljesül minden hibamentes P_i és P_l folyamatra.

Azt állítjuk, hogy az x csúcs gyerekeinél a többség címkéje hibátlan folyamat indexére végződik. Ez igaz, mert x gyerekeinek száma pontosan $n - r \geq n - f$. Valamint feltettük, hogy $n > 3f$, ez a szám szigorúan nagyobb, mint $2f$, és minthogy a gyerekek közül legfeljebb f-nek a címkéje végződik hibás folyamat indexével, a szükséges többség megvan.

Ebből következik, hogy bármely hibamentes P_i-re

az x csúcs xl gyerekeinek többségére *új_ érték*$(xl)_i = v$ fennáll. Ekkor az algoritmusban használt többségi szabályból következik, hogy *új_ érték*$(x)_i = v$ a hibátlan P_i folyamatoknál. Így v a kívánt érték. □

Ellenőrizzük az érvényességet.

6.17. lemma. . *Ha mindegyik hibátlan folyamat ugyanazzal a $v \in V$ kezdeti értékkel kezd, akkor egyedül v a lehetséges döntési érték a hibátlan folyamatoknál.*

Bizonyítás. Ha az összes hibátlan folyamat v-vel kezd, akkor az összes hibátlan folyamat az első menetben közreadja v-t, ezért *érték*$(j)_i = v$ minden hibátlan P_i és P_j folyamatra. A 6.16. lemmából következik, hogy *új_ érték*$(j)_i = v$ minden hibátlan P_i és P_j folyamatra. Ekkor az algoritmusban használt többségi szabályból adódik, hogy *új_ érték*$(\lambda)_i = v$ minden hibátlan P_i-re. Így P_i döntése v, amire szükségünk volt. □

A megegyezési tulajdonság bizonyításához két új definícióra van szükségünk. Az első: egy gyökeres fa csúcsainak egy C halmaza *útvonal lefedés*, ha minden gyökértől levélhez vezető útvonal legalább egy C-beli csúcsot tartalmaz.

A második: tekintsük az EIGYBIZ algoritmus egy tetszőleges α végrehajtási sorozatát. A fa egy x csúcsa *közös* az α-ban, ha az α végrehajtási sorozat $(f+1)$-edik menetében az összes hibamentes P_i ugyanazon *új_ érték*$(x)_i$ értékkel rendelkezik. A fa csúcsainak egy halmaza (például egy útvonal lefedés) *közös* az α-ban, ha a halmazba tartozó csúcsok mindegyike közös az α-ban. Vegyük észre, hogy a 6.16. lemma szerint, ha P_i hibamentes, akkor minden x-re az xi egy közös csúcs.

6.18. lemma. . *Az* EIGYBIZ *algoritmus tetszőleges végrehajtásának $(f+1)$-edik menetében létezik egy olyan útvonal lefedés, mely közös az α-ban.*

Bizonyítás. Legyen C azon xi alakú csúcsok halmaza, ahol P_i hibamentes. Ahogy fentebb már észrevettük, minden C-beli csúcs közös. Megmutatjuk hogy C egy útvonal lefedés. Tekintsünk egy tetszőleges, a gyökértől egy levélhez vezető útvonalat. Ez pontosan $f+1$ nem gyökér csúcsot tartalmaz, és T felépítéséből adódóan mindegyik csúcs címkéje különböző folyamat indexével végződik. Mivel legfeljebb f hibás folyamat van, az útvonalon kell egy olyan csúcsnak lennie, mely címkéje hibátlan folyamat indexére végződik. Ez a csúcs viszont eleme C-nek. □

A következő lemma megmutatja, hogyan terjeszkednek felfelé a fában a közös csúcsok.

6.19. lemma. . *Az* EIGYBiz *algoritmus $f+1$ menete után igaz az alábbi. Legyen x egy tetszőleges címke az EIGY fában. Ha létezik közös útvonal lefedés az x gyökerű részfában, akkor az x közös.*

Bizonyítás. A bizonyítás indukcióval történik a fa címkéi alapján a levelektől a gyökér felé haladva.

Alapeset: legyen x egy levél. Ekkor az x részfa útvonal lefedése csak magából az x csúcsból áll. Így x közös, ami a bizonyítandó volt.

Indukciós lépés: legyen $|x| = r$, $0 \leq r \leq f$. Tegyük fel, hogy az x részfájának van egy C közös útvonal lefedése. Ha x eleme C-nek, akkor x közös, és kész vagyunk, ezért legyen $x \notin C$.

Vegyük x egy *tetszőleges* xl gyerekét. Mivel $x \notin C$, így C generál egy útvonal lefedést az xl gyökerű részfában. Ezért az indukciós feltevés szerint xl közös. Mivel xl tetszőlegesen kiválasztott gyereke volt x-nek, x összes gyereke közös. Ekkor a *új_érték*(x) definíciójából következik, hogy x közös. □

Egyszerű következményként kapjuk az alábbit lemmát.

6.20. lemma. . *Az* EIGYBiz *algoritmus $f+1$ menete után a λ gyökér közös.*

Bizonyítás. A 6.18. és 6.19. lemmákból adódik. □

Most összekötve a részeket, lássuk a fő helyességi tételt.

6.21. tétel. . *Az* EIGYBiz *algoritmus megoldja a bizánci megegyezési problémát n folyamat és f hiba esetében, ha $n > 3f$ teljesül.*

Bizonyítás. A befejeződés nyilvánvaló. Az érvényesség a 6.17. lemmából következik. A megegyezés a 6.20. lemmából és a döntési szabályból adódik. □

Bonyolultságelemzés. A költségek ugyanazok, mint az EIGYStop algoritmus esetében: $f+1$ menet, $\mathcal{O}((f+1)n^2)$ üzenet és $\mathcal{O}(n^{f+1}b)$ a kommunikáció bitszáma. Ezenkívül új követelményként megkívánjuk, hogy a folyamatok száma legyen nagy a hibákhoz képest: $n > 3f$.

6.3.3.. A bináris bizánci megegyezésen alapuló általános bizánci megegyezés

Ebben a szakaszban megmutatjuk, hogyan használható szubrutinként egy olyan algoritmus, mely megoldja a bizánci megegyezést $\{0,1\}$ bemenetre, az általános bizánci megegyezés megoldásához. A plusz költség két többlet-menet, $2n^2$ többlet-üzenet és $\mathcal{O}(n^2b)$ bit a kommunikációra. Ez tekintélyes megtakarításhoz vezethet a kommunikációhoz szükséges bitek számát tekintve, mivel nem szükséges, hogy V-beli értékeket küldjünk, elegendő bináris értékek küldése a szubrutin végrehajtása alatt. Bár ez előrelépés, de nem elegendő arra, hogy a kommunikációhoz szükséges bitek száma exponenciálisról polinomiálisra csökkenjen f függvényében.

Az algoritmus neve, a tervezői után, TurpinCoan. Az algoritmus felteszi, hogy $n > 3f$. Ahogy korábban, most is feltesszük, hogy a folyamat önmagának ugyanúgy küldhet üzenetet, mint a többi folyamatnak.

TurpinCoan algoritmus (vázlatosan)

Minden egyes folyamatnak lokális változói az x, y, z és *szavaz*, x kezdetben a folyamat bemeneti értékét veszi fel, y, z és *szavaz* tetszőleges kezdőértékkel indul.

Első menet. P_i elküldi saját x változójának értékét az összes folyamatnak, beleértve önmagát is. Ha az ebben a menetben érkezett üzenetekben egy adott v érték $\geq n-f$ példányban fordul elő, akkor P_i az $y := v$ beállítást hajtja végre, egyébként az $y := null$ értékadást.

Második menet. P_i elküldi saját y változójának értékét az összes folyamatnak, beleértve önmagát is. Ha az ebben a menetben érkezett üzenetekben egy adott V-beli érték $\geq n-f$ példányban fordul elő, akkor P_i a $szavaz := 1$ beállítást hajtja végre, egyébként a $szavaz := 0$ értékadást. Ugyanekkor, P_i a z változóját a menetben kapott üzenetekben szereplő nem *null* értékek közül a leggyakoribbra állítja be, döntetlen esetében tetszőlegesen döntve; ha mindegyik üzenet *null*, akkor z határozatlan értékű marad.

r-edik menet.$(r \geq 3)$*:* A folyamatok a bináris bizánci megegyezés algoritmusát futtatják, *szavaz* értékét bemenő értékként használva. Ha P_i döntése 1 a szubrutinban, és z értéke definiált, akkor az algoritmus végső döntése z, egyébként az alapértékként megadott v_0 a döntés.

Kulcsfontosságú tény a TurpinCoan algoritmusnál a következő.

6.22. lemma. . *Legfeljebb egy olyan $v \in V$ érték van, melyet a hibamentes folyamatok egymásnak küldenek a második menetben.*

Bizonyítás. Tegyük fel, az állítás ellenkezőjét, hogy a P_i és P_j hibamentes folyamatok a második menetben külön-külön a v és w értékeket tartalmazó üzenetet küldik, $v, w \in V$ és $v \neq w$. Ekkor P_i legalább $n-f$ számú, v-t tartalmazó első menetbeli üzenetet kapott. Minthogy legfeljebb f hibás folyamat lehet, és a hibátlan folyamatok ugyanazt az üzenetet küldik az összes folyamatnak az első menetben,

így P_j legalább $n-2f$ darab v-t tartalmazó üzenetet kap. Minthogy $n > 3f$, P_j-hez legalább $f+1$ darab v-t tartalmazó üzenet érkezik.

Viszont abból, hogy P_j a második menetben w-t küld, az következik, hogy legalább $n-f$ darab első menetbeli w-t tartalmazó üzenetet kapott, ezek összege legalább $(f+1)+(n-f) > n$ üzenet. Azonban a P_j folyamat az első menetben csak n üzenetet kaphatott, tehát ellentmondásra jutottunk. □

6.23. tétel. . *A* TurpinCoan *algoritmus megoldja az általános bizánci megegyezési problémát a bináris bizánci megegyezés algoritmusát szubrutinként használva, ha $n > 3f$ fennáll.*

Bizonyítás. A befejeződés egyszerűen belátható.

Az érvényesség belátásához be kell bizonyítanunk, hogy ha az összes hibamentes folyamat ugyanazzal a v kezdeti értékkel indul, akkor az összes hibamentes folyamat döntése v lesz. Ezért tegyük fel, hogy mindegyik hibamentes folyamat v értékkel kezd. Ezután az összes $\geq n-f$ hibamentes folyamat sikeresen szétküldi az első menetbeli v-t tartalmazó üzeneteket valamennyi folyamatnak. Így az első menetben az összes hibamentes folyamat az y változóját v értékre állítja. Majd a második menetben mindegyik hibátlan folyamat legalább $n-f$ darab v-t tartalmazó üzenetet kap, amiből az következik, hogy a z változójukat v-re állítják, *szavaz* változójukat pedig 1-re. Mivel az összes hibamentes folyamat az 1 bemenetet alkalmazza a bináris bizánci megegyezés szubrutinjában, mindannyiuk döntése 1 lesz a szubrutinban, az algoritmus érvényességi feltételének megfelelően. Ez viszont azt jelenti, hogy a fő algoritmusban mindegyikük döntése v lesz, ami épp az érvényesség feltétele.

Végül belátjuk a megegyezést. Ha a szubrutin döntési értéke 0, akkor a v_0 értéket választja valamennyi hibátlan folyamat a végső döntésként, így a megegyezés alapértelmezés szerint fennáll.

Ezért tegyük fel azt, hogy a szubrutin döntése az 1 érték. Ekkor a szubrutinra vonatkozó érvényességi szabály szerint néhány hibamentes P_i folyamatra a szubrutint kezdetekor a $szavaz_i = 1$ fennáll. Ez azt jelenti, hogy P_i legalább $n-f$ darab, egy adott $v \in V$ értéket tartalmazó, második menetbeli üzenetet kap, és minthogy legfeljebb f hibás folyamat van, így P_i legalább $n-2f$ darab v-t tartalmazó üzenetet kap a hibamentes folyamatoktól. Továbbá, ha P_j egy tetszőleges hibátlan folyamat, akkor a P_j is legalább $n-2f$ darab v-t tartalmazó, második menetbeli üzenetet kap ugyanazon hibátlan folyamatoktól. A 6.22. lemma szerint csak egy adott $v \in V$ értéket küldhetnek a hibátlan folyamatok a második menetben. Ezért P_j nem kaphat f-nél több olyan üzenetet, melynek egy másik, v-től különböző V-beli elem az értéke (ezek a hibás folyamatok üzenetei). Mivel $n > 3f$, így $n-2f > f$, ezért a v érték a leggyakoribb a P_j folyamathoz érkezett második menetbeli üzenetek között. Ebből következik, hogy P_j a második menetben a $z := v$ értékadást végzi el. Mivel a szubrutin döntési értéke 1, ez azt jelenti, hogy P_j döntése v. Minthogy ez tetszőleges hibamentes P_j-re fennáll, a megegyezést beláttuk. □

A TurpinCoan algoritmus bizonyításánál a hibás folyamatok darabszámára adott korlátot felhasználtuk arra, hogy segédállításokat kapjunk a különböző folyamatok nézetei közti hasonlóságról a végrehajtás során. Ezt a fajta érvelést

egyéb egyetértési algoritmusra vonatkozó bizonyításnál is használjuk, például a *közelítő megegyezésnél* a 7.2. fejezetben.

Bonyolultságelemzés.. A menetek száma $r + 2$, ahol r a bináris bizánci megegyezés szubrutin meneteinek száma. A többlet-kommunikáció, amit a TurpinCoan algoritmus a szubrutinon kívül használ, $2n^2$ üzenet, mindegyik legfeljebb b bites, tehát a bitszám összesen $\mathcal{O}(n^2 b)$ bit.

6.3.4.. A kommunikációs költség csökkentése

Bár a TurpinCoan algoritmus alkalmas arra, hogy valamelyest csökkentsük a bizánci megegyezés kommunikációs költségének a bitszámát, de a költség továbbra is exponenciális függvénye lesz f-nek, a hibák számának. A bizánci hiba modellben nehezebb olyan algoritmust találni, mely polinomiális a hibák számára nézve, mint a megállási hibákat tartalmazó modellben. Ebben a részben mutatunk rá egy példát; ez az algoritmus a futási idő költségét tekintve nem lesz optimális, de nagyon egyszerű és néhány érdekes technikát alkalmaz. Az algoritmus speciálisan a $\{0, 1\}$ értékű bizánci megegyezésre vonatkozik, a 6.3.3. szakasz eredményei alapján láthatjuk, hogyan lehet ezt az algoritmust az általános esetre alkalmazni.

Az algoritmus a *következetes üzenetszórás* működési módszert használja az összes kommunikációjához. Ez a működési módszer egy módja annak, hogy a különböző folyamatok által kapott üzenetek között bizonyos mennyiségű összefüggést biztosítsunk. A következetes üzenetszórást alkalmazva egy P_i folyamat *közreadhat* egy (m, i, r) alakú üzenetet az r-edik menetben, és ezt az üzenetet a folyamatok (beleértve P_i-t magát is) tetszőleges ezutáni menetben *elfogadhatják*. A következetes üzenetszórás működési módszer a következő három feltételnek tesz eleget.

1. Ha egy hibátlan folyamat az r-edik menetben közread egy (m, i, r) üzenetet, akkor a hibátlan folyamatok ezt az üzenetet az $(r + 1)$-edik menetig elfogadják (azaz vagy az r-edik, vagy az $(r + 1)$-edik menetben).
2. Ha a P_i hibátlan folyamat nem ad közre az r-edik menetben egy (m, i, r) üzenetet, akkor az (m, i, r) üzenetet soha egyetlen hibátlan folyamat sem fogadja el.
3. Ha egy hibátlan P_j folyamat elfogad egy (m, i, r) üzenetet, mondjuk az r'-edik menetben, akkor ezt az összes hibátlan folyamat elfogadja az $(r' + 1)$-edik menetig.

Az első feltétel kimondja, hogy a hibátlan folyamatok közreadott üzeneteit gyorsan elfogadják, a második szerint hibátlan folyamatnak tévesen soha nem tulajdonítható üzenet. A harmadik feltétel szerint egy hibátlan folyamat által elfogadott üzenetet (akár hibás, akár nem hibás a küldő) nem sokkal később az összes többi hibátlan folyamat is elfogadja.

A következetes üzenetszórás módszere könnyen kivitelezhető.

A KövetkezetesÜzenSzórás algoritmus (vázlatosan)

Az (m, i, r) üzenet r-edik menetbeli közreadásához P_i küld egy („*kezd*”, m, i, r) üzenetet az összes folyamatnak az r-edik menetben. Ha P_j

kap egy („*kezd*", m, i, r) üzenetet P_i-től az r-edik menetben, akkor küld egy („*echó*", m, i, r) üzenetet az összes folyamatnak az $(r+1)$-edik menetben.

Ha tetszőleges $r' \geq r+2$ menet előtt P_j legalább $f+1$ folyamattól kapott („*echó*", m, i, r) üzenetet, akkor P_j („*echó*", m, i, r) üzenetet küld az r'-edik menetben (ha addig még nem tette meg).

Ha tetszőleges $r' \geq r+1$ menet végéig P_j legalább $n-f$ folyamattól kapott („*echó*", m, i, r) üzenetet, akkor P_j elfogadja a kommunikációt az r'-edik menetben (ha addig még nem tette meg).

6.24. tétel. . *A* KövetkezetesÜzenSzórás *algoritmus megoldja a következetes üzenetszórás problémát, ha* $n > 3f$ *fennáll.*

Bizonyítás. Bebizonyítjuk, hogy a három tulajdonság fennáll.

1. Tegyük fel, hogy a hibátlan P_i folyamat közreadja az (m, i, r) üzenetet az r-edik menetben. Ekkor P_i az („*kezd*", m, i, r) üzenetet küldi az r-edik menetben, és az $\geq n-f$ számú hibátlan folyamat mindegyike az („*echó*", m, i, r) üzenetet küldi az $(r+1)$-edik menetben. Majd az $(r+1)$-edik menet végén a hibátlan folyamatok mindegyike („*echó*", m, i, r) üzenetet kap legalább $n-f$ folyamattól, és így elfogadja az üzenetet.
2. Ha a hibátlan P_i folyamat nem ad közre egy (m, i, r) üzenetet az r-edik menetben, akkor nem küld („*kezd*", m, i, r) üzeneteket, így a hibátlan folyamatok soha nem küldenek („*echó*", m, i, r) üzenetet. Ekkor a hibátlan folyamatok soha nem fogják elfogadni ezt az üzenetet, mivel az elfogadás feltétele, hogy az adott folyamathoz legalább $n-f > f$ számú folyamattól *echó* üzenet érkezzen.
3. Tegyük fel, hogy az (m, i, r) üzenetet a hibamentes P_j folyamat elfogadta az r'-edik menetben. Továbbá, hogy P_j („*echó*", m, i, r) üzenetet kap legalább $n-f$ folyamattól az r'-edik menetig. Az $n-f$ folyamat között legalább $n-2f \geq f+1$ a hibátlan folyamatok száma. Mivel a hibátlan folyamatok ugyanazt az üzenetet küldik az összes folyamatnak, így mindegyik hibamentes folyamat legalább $f+1$ számú („*echó*", m, i, r) üzenetet kap az r'-edik menetig. Ebből következik, hogy az $(r'+1)$-edik menetig mindegyik hibamentes folyamat küld egy („*echó*",m,i,r) üzenetet, tehát mindegyik folyamat legalább $n-f$ darab („*echó*",m,i,r) üzenetet kap az $(r'+1)$-edik menetig. Ezért az üzenetet valamennyi hibátlan folyamat elfogadja az $(r'+1)$-edik menetig.

□

Bonyolultságelemzés.. Egyetlen üzenet következetes szórása $\mathcal{O}(n^2)$ üzenetet igényel.

Most bemutatunk egy egyszerű bizánci megegyezés algoritmust, mely a következetes üzenetszórást használja a teljes kommunikációja során. A PoliBiz algoritmus csak az 1 kezdeti értékekről küld szét információt. Egy növekvő küszöbértéket használ az üzenetszóráshoz.

A PoliBiz algoritmus (vázlatosan)

Az algoritmus $f+1$ szintet használ, minden szint két menetből áll. Az elküldött üzenetek (következetes üzenetszórást használva) $(1, i, r)$ alakúak, ahol i egy folyamat indexe, r egy páratlan szám, a menet száma. Tehát az üzenetek elküldése csak az adott szint első menetében történik, és az elküldött információ értéke csak 1 lehet.

Annak feltételei, hogy a P_i folyamat közreadjon egy üzenetet, a következők. Az első menetben P_i akkor adja közre az $(1, i, 1)$ üzentet, ha P_i kezdeti értéke 1. A $(2s-1)$-edik menetben, mely az s-edik szint első menete, ahol $2 \leq s \leq f+1$, P_i pontosan akkor adja közre az $(1, i, 2s-1)$ üzenetet, ha P_i legalább $f+s-1$ számú, különböző folyamatoktól érkezett üzenetet elfogadott a $(2s-1)$-edik menet előtt, és eddig P_i még nem adott közre üzenetet.

A $2(f+1)$-edik menet végén P_i pontosan akkor hoz 1-es döntést, ha a $2(f+1)$-edik menet végéig legalább $2f+1$ különböző folyamat üzenetét elfogadta. Egyébként P_i döntése 0 lesz.

6.25. tétel. . *A* PoliBiz *algoritmus megoldja a bizánci megegyezést, ha* $n > 3f$ *fennáll.*

Bizonyítás. A befejeződés nyilvánvaló.

Az érvényességet tekintve két eset lehetséges. Az első, amikor az összes hibátlan folyamat az 1-es értékkel kezd, ekkor legalább $n - f \geq 2f + 1$ folyamat ad közre üzenetet az első menetben. A következetes üzenetszórás első tulajdonságából adódóan az összes hibamentes folyamat elfogadja ezeket az üzeneteket a második menetig, így a hibamentes folyamatok mindegyike legalább $2f+1$ különböző folyamattól fogad el üzenetet a második menet végéig. Ez elegendő ahhoz, hogy a hibátlan folyamatok mindegyike 1-est döntsön.

A másik eset, ha az összes folyamat a 0 kezdeti értékkel kezd, ekkor a hibamentes folyamatok egyetlen üzenetet sem adnak közre. Ennek az az oka, hogy legalább $f+1$ folyamatnak el kell fogadnia az üzenetet, hogy az közreadhatóvá váljon, aminek elérése lehetetlen anélkül, hogy előzetesen egy hibátlan folyamat közre ne adná. (A következetes üzenetszórás második tulajdonságát használjuk fel itt.) Ebből következik, hogy az összes hibamentes folyamat döntése 0 lesz.

Végül megvizsgáljuk a megegyezést. Legyen a hibátlan P_i folyamat döntése 1; elegendő megmutatni, hogy az összes többi hibamentes folyamat döntése is 1. Mivel P_i döntése 1, P_i-nek legalább $2f+1$ számú különböző folyamattól kell elfogadnia üzenetet a $2(f+1)$-edik menet végéig. Legyen I azoknak a folyamatoknak a halmaza, melyek ezek közül hibamentesek, ekkor $|I| \geq f+1$.

Ha mindegyik I-beli folyamat kezdeti értéke 1, akkor ezt mindannyian közreadják az első menetben, és a következetes üzenetszórás első tulajdonsága miatt valamennyi hibamentes folyamat elfogadja ezeket az üzeneteket a második menetig. Majd a harmadik menet[2] előtt, a hibamentes folyamatok mindegyike legalább $f+1$ különböző folyamattól érkező üzenetet fogadott el, mely elegendő ahhoz, hogy a harmadik menetre ez közreadhatóvá váljon, és ismét a következetes üzenetszórás első tulajdonsága miatt, az összes hibátlan folyamat elfogadja

[2]Feltesszük, hogy $f \geq 1$, ezért áll itt épp a harmadik menet.

ezt az üzenetet a negyedik menetig. Így mindegyik hibamentes folyamat legalább $n - f \geq 2f + 1$ különböző folyamat üzenetét fogadja el a negyedik menet végéig, emiatt a döntésük 1 lesz, amit bizonyítani akartunk.

Másik esetben tegyük fel, hogy az egyik I-bel folyamatnak, legyen az P_j, a kezdeti értéke nem 1. Ekkor P_j a $2s-1$ menet valamelyikében ad közre üzenetet, ahol $2 \leq s \leq f+1$, ami azt jelenti, hogy P_j legalább $f+s-1$ különböző folyamat üzenetét fogadja el a $(2s-1)$-edik menet előtt; mi több az üzenetek egyike sem származik önmagától. Ekkor a következetes üzenetszórás harmadik tulajdonsága miatt ezen $f + s - 1$ számú folyamat üzenetét az összes hibamentes folyamat elfogadja a $(2s-1)$-edik menet végéig, és így az első tulajdonságból következik, hogy a P_j által közreadott üzenetet az összes hibamentes folyamat elfogadja a $2s$-edik menet végéig. Ebből következik, hogy valamennyi hibamentes folyamat elfogadja legalább $(f + s - 1) + 1 = f + s$ különböző folyamat üzenetét a $2s$-edik menet végéig.

Ekkor két eset lehetséges. Ha $s = f + 1$, akkor mindegyik hibamentes folyamat legalább $2f + 1$ különböző folyamat üzenetét fogadja el a $2(f + 1)$-edik menet végéig, mely elegendő, hogy biztosítsa, hogy a folyamatok mindegyikének 1 legyen a döntése. A másik esetben $s \leq f$, akkor az összes hibamentes folyamat elegendően sok üzenetet fogad el a $(2s + 1)$-edik menet előtt ahhoz, hogy közreadjon a $(2s + 1)$-edik menetben, ha eddig még nem tette volna. Majd a következetes üzenetszórás első tulajdonsága szerint, az összes hibátlan folyamat elfogadja valamennyi hibátlan folyamat üzenetét a $(2s + 2)$-edik menet végéig. Ez ismét elegendő ahhoz, hogy valamennyien 1-est döntsenek, amire szükségünk volt. □

Bonyolultságelemzés.. A PoliBiz algoritmus meneteinek száma $2f + 2$. Legfeljebb n a közreadások száma, melyek mindegyike $\mathcal{O}(n^2)$ üzenet jelent; így az üzenetek száma $\mathcal{O}(n^3)$. A bitek száma üzenetenként $\mathcal{O}(\log n)$, mivel az üzenetek a folyamatok indexeit tartalmazzák. Így a kommunikációs bonyolultság bitszáma $\mathcal{O}(n^3 \log n)$.

Kapcsolat a hitelesített bizánci hiba modellel.. Ha a közönséges bizánci modellhez hozzáadjuk a következetes üzenetszórás képességét, egy olyan modellhez jutunk, ami valamiképp hasonló a 6.2.4. szakaszban vázlatosan ismertetett hitelesített bizánci hiba modellhez. Jóllehet a kettő nem pontosan ugyanaz. Például a következetes üzenetszórás csak üzenetszórásra szolgál és nem az egyedi üzenetküldésre. Még kifejezőbben, a következetes üzenetszórás nem védi meg P_i folyamatot attól, hogy közreadjon egy üzenetet, melyben (hibásan) azt állítja, hogy egy adott P_j folyamat egy bizonyos üzenetet küldött; a hibátlan folyamatok mindannyian elfogadják ezt az üzenetet, akkor is, ha a tartalma téves állítás. A hitelesített bizánci hiba modellben, a digitális aláírás lehetővé teszi, hogy a folyamatok az ilyen üzeneteket azonnal elutasítsák. Bár a modellek némileg különbözők, a következetes üzenetszórás elég erős ahhoz, hogy felhasználható legyen a közönséges bizánci modell néhány olyan algoritmusának megvalósításához, melyeket a hitelesített bizánci hiba modellhez terveztek.

6.4.. A bizánci megegyezés folyamatainak száma

Eddig olyan algoritmusokat mutattunk be, melyek megoldják a megegyezési problémát egy teljes hálózati gráfban, megállási, sőt bizánci hibák jelenlétében. Láthattuk, hogy ezek igen költséges algoritmusok. Megállási hibák esetére az általunk adott algoritmusok közül a legjobb a OptHalmazTerjed algoritmus, melynek költségei $f+1$ menet, $2n^2$ üzenet és $\mathcal{O}(n^2b)$ bit a kommunikációra. A bizánci esetben az EIGYBiz algoritmus $f+1$ menetet használ, a kommunikációs költsége exponenciális nagyságrendű, míg a PoliBiz algoritmus $2(f+1)$ menetet használ és a kommunikációs költsége polinomiális nagyságrendű. Mindkét bizánci algoritmus megkívánja, hogy $n > 3f$ fennálljon.

A fejezet hátralevő részében megmutatjuk, hogy nem véletlenek ezek a magas költségek. Ebben a részben először bebizonyítjuk, hogy az $n > 3f$ megszorítás szükségszerű a bizánci megegyezési feladat tetszőleges megoldásában. A következő két alpont ezzel kapcsolatos eredményeket tartalmaz: a 6.5. alfejezet megadja az összefüggőség pontos mértékét arra, hogy tetszőleges nem teljes hálózati gráf esetében a bizánci megegyezés megoldható legyen, míg a 6.6. alfejezetben megmutatjuk, hogy az $n > 3f$ korlát kiterjed a bizánci megegyezésnél gyengébb állításokat tartalmazó problémára is. A fejezet utolsó részében belátjuk, hogy a menetek darabszámára megadott $f+1$ alsó korlát úgyszintén szükséges, még a megállási hibákat tartalmazó egyszerűbb esetben is.

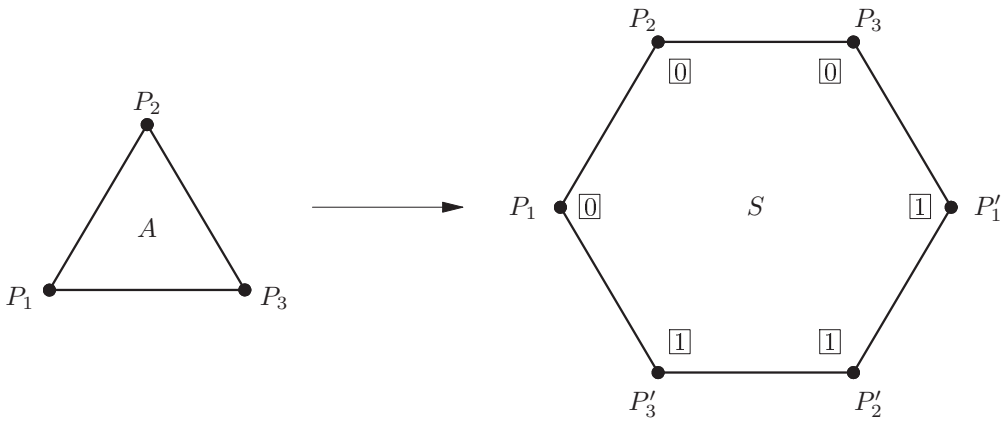
A bizonyítást, hogy f hiba jelenlétében $n \leq 3f$ folyamat nem képes megoldani a bizánci hibát, a legegyszerűbb speciális esettel kezdjük: megmutatjuk, hogy három folyamat nem tudja megoldani a bizánci megegyezést, ha fennáll egy hibának a lehetősége. Ezt az eredményt előrevetítette a 6.3.1. szakaszban leírt példa, bár a példa önmagában nem adott bizonyítást. Most egy általános eredményt adunk, olyan tetszőlegesen választott n, f értékekre, melyekre $n \leq 3f$ fennáll, úgy, hogy „redukáljuk" a problémát a három egy ellen típusú felállásra.

6.26. lemma. . *Három folyamat nem képes a bizánci megegyezés megoldására egy hiba jelentkezése esetében.*

Bizonyítás. Indirekten: tegyük fel, hogy az A algoritmus megoldja a bizánci megegyezési problémát a P_1, P_2, P_3 folyamatokra, akkor is, ha a három közül az egyik hibázhat. Alkossunk egy új S rendszert az A két példányából, és megmutatjuk, hogy S ellentmondásosan viselkedik. Ebből az következik, hogy a feltételezett A algoritmus nem létezik.

Egész pontosan vegyük az A-beli folyamatok két-két példányát, és rendezzük őket egy hatszög alakú S rendszerbe. A folyamatok egyik példányát (a másolt példányt) a 0 bemeneti értékkel indítjuk, a másik példányt (az eredetit) az 1 bemeneti értékkel. Az elrendezés a 6.7. ábrán látható.

Mit mondhatunk az S rendszerről formálisan? Ez egy szinkron rendszer egy hatszög alakú hálózati gráfban, a 2. fejezet általános modelljéhez tartozik. Jegyezzük meg, *nem* állítjuk azt, hogy ez a rendszer megoldja a bizánci megegyezési problémát – nem az a fontos számunkra, hogy mit tud, a lényeg, hogy ez egyfajta szinkron rendszer. Nem fogjuk vizsgálni az S-beli folyamatok működési hibáit.



6.7.. ábra. Az A két példányából összeállított S rendszer.

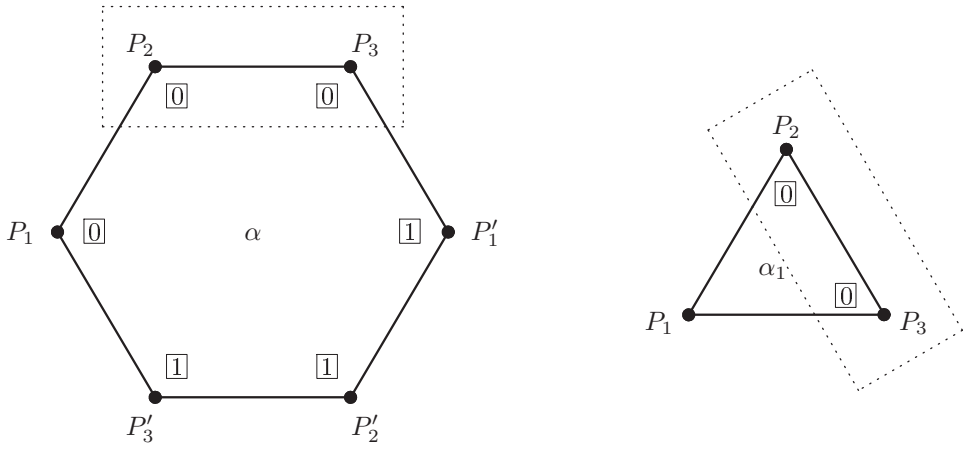
Emlékezzünk rá, hogy azokban a rendszerekben, melyeket a bizánci megegyezési feladat megoldásának tekintettünk, feltettük, hogy mindegyik folyamat ismeri a teljes hálózati gráfot. Például A-ban P_1 ismeri a P_2 és P_3 neveket, és feltételezi, hogy pontosan három csúcspont van, P_1, P_2 és P_3, melyek háromszöget alkotnak. S-ben nem azt feltételezzük, hogy a folyamatok az egész hatszögű gráfot ismerik, hanem csak azt, hogy a folyamatok a szomszédaik egy helyi nevével rendelkeznek. Például S-ben a P_1 azt tudja, hogy két szomszédja van, akiknek P_2 és P_3 a nevük, bár valójában az egyikük P_3'. P_1 nem tudja, hogy a csúcsok két példányban vannak a hálózatban. A helyzet hasonló a 4. fejezetbelihez, ahol a folyamatoknak csak helyi ismereteik voltak a hálózati gráf saját maguk körüli részéről. Azt mondhatjuk, hogy az S-beli hálózat a folyamatok számára gyakorlatilag úgy néz ki, mintha A-beli lenne.

Nem várunk el az S rendszertől semmiféle különleges viselkedést. Azt viszont igen, hogy S bármely bemeneti érték hozzárendelésre *egy adott*, jól definiált viselkedést mutasson. Úgy fogunk ellentmondásra jutni, hogy megmutatjuk, nem lehetséges ilyen jól definiált viselkedése S-nek a fentebb említett bemeneti értékek esetére.

Tegyük fel tehát, hogy S-ben a folyamatok a 6.7. ábrán látható kezdeti értékkel indulnak, a folyamatok másolt példányai a 0-val, az eredeti példányok az 1-gyel; legyen α az eredményül kapott végrehajtási sorozat S-ben.

Elsőként a P_2 és P_3 folyamatok szemszögéből vizsgáljuk az α végrehajtási sorozatot. A P_2 és P_3 folyamatoknak úgy tűnik, hogy egy A, háromszög elrendezésű rendszerben futnak, egy olyan végrehajtási sorozatban, legyen ez α_1, melyben a P_1 folyamat hibás. Az α és az α_1 megkülönböztethetetlen a P_2 és P_3 folyamatok számára, $\alpha \overset{2}{\sim} \alpha_1$ és $\alpha \overset{3}{\sim} \alpha_1$ a „megkülönböztethetetlenség” 2.4. alfejezetben megadott definíciója szerint. Lásd a 6.8. ábrát. Az α_1-ben P_1 egy különös fajta hibás viselkedést mutat – úgy viselkedik, mintha az α-beli P_1', P_2', P_3' és P_1 kombinációja lenne. Bár ez a viselkedés különleges, de megengedhető egy A-beli hibás

folyamatnak, bizánci hibákat feltételezve.



6.8.. ábra. A P_2 és P_3 folyamatok számára az α és α_1 megkülönböztethetetlen végrehajtások.

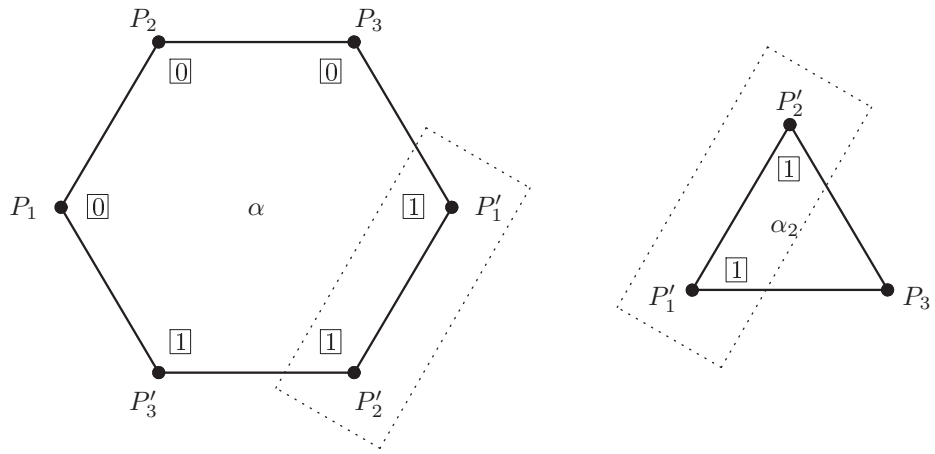
Tudjuk, hogy α_1 egy olyan A-beli végrehajtási sorozat, amelyben a P_1 hibás, P_2 és P_3 a 0 kezdeti értékkel kezd, és mivel feltevésünk szerint A megoldja a bizánci megegyezést, a bizánci megegyezés helyességi követelményeiből adódik, hogy végül α_1-ben P_2-nek és P_3-nak 0-ás döntést kell hoznia. Minthogy α megkülönböztethetetlen α_1-től P_2 és P_3 számára, mindkettőjük döntése az α-ban is 0 lesz.

Most tekintsük az α végrehajtási sorozatot a P_1' és P_2' folyamatok szemszögéből. A P_1' és P_2' folyamatoknak úgy tűnik, hogy egy A, háromszög elrendezésű rendszerben futnak, egy olyan végrehajtási sorozatban, legyen ez α_2, melyben P_3 hibás. Azaz $\alpha \overset{1'}{\sim} \alpha_2$ és $\alpha \overset{2'}{\sim} \alpha_2$. Lásd a 6.9. ábrát. A fentivel azonos érvelésből adódóan P_1' és P_2' végül 1-es döntést hoz α-ban.

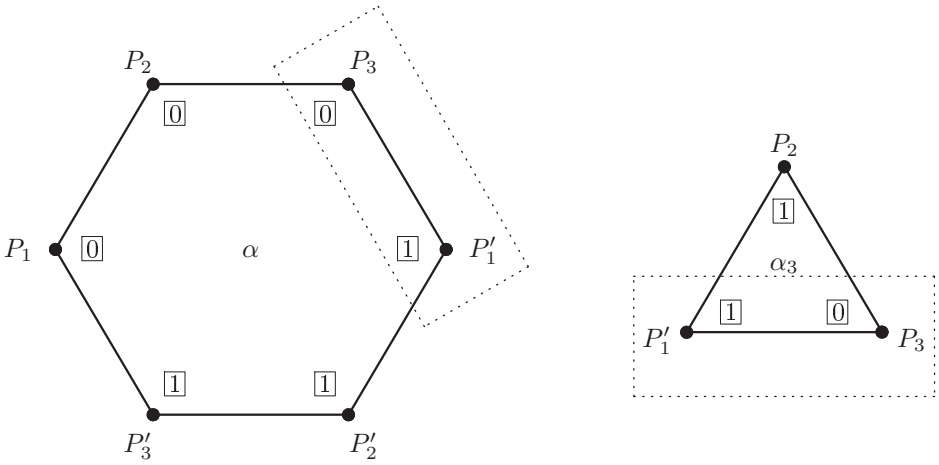
Végül tekintsük az α végrehajtási sorozatot a P_3 és P_1' folyamatok szemszögéből. A P_3 és P_1' folyamatoknak úgy tűnik, hogy egy A, háromszög elrendezésű rendszerben futnak, egy olyan végrehajtási sorozatban, legyen ez α_3, melyben P_2 folyamat hibás. Azaz $\alpha \overset{3}{\sim} \alpha_3$ és $\alpha \overset{1'}{\sim} \alpha_3$. Lásd a 6.10. ábrát. A bizánci megegyezés helyességi feltételeiből adódik, hogy P_3 és P_1' folyamatoknak végül döntést kell hozniuk α_3-ban, és ugyanazt kell dönteniük. Mivel P_3 a 0 kezdeti értékkel indul és P_1 az 1 kezdeti értékkel, nincs elvárás, hogy melyik érték mellett döntenek, de a megegyezési feltétel szerint megegyeznek. Ezért α-ban is ugyanazt döntik.

Ez viszont ellentmondás, mivel azt már tudjuk, hogy α-ban P_3 folyamat a 0 döntést hozza, P_1' pedig az 1-et. □

Most a 6.26. lemmát használva bebizonyítjuk, hogy a bizánci megegyezés megoldása lehetetlen $n \leq 3f$ folyamattal. Ehhez megmutatjuk, hogy ha létezne $n \leq 3f$ folyamattal megoldás, mely f számú hibát eltűr, akkor abból következik, hogy létezik megoldás három folyamatra és egy bizánci hibára, ami ellentmond a



6.9.. ábra. Az P_1' és P_2' folyamatok számára az α és α_2 megkülönböztethetetlen végrehajtási sorozatok.



6.10.. ábra. A P_3 és P_1' folyamatok számára az α és α_3 megkülönböztethetetlen végrehajtási sorozatok.

6.26. lemmának.

6.27. tétel. . *A bizánci megegyezés problémának nincs megoldása n folyamatra f bizánci hiba előfordulása esetében, ha* $2 \leq n \leq 3f$.

Bizonyítás. Arra a speciális esetre, amikor $n = 2$, egyszerűen belátható, hogy nincs megoldás. Vázlatosan elmondva, tegyük fel, hogy az egyik folyamat 0-val kezd, a másik 1-gyel. Ekkor mindegyikőjük –annak lehetőségét megengedve, hogy

a másik hibás– saját értéke szerint dönt, hogy kielégítse az érvényesség feltételét. Azonban, ha egyikőjük sem hibás, ez megsérti a megegyezési feltételt. Így feltehetjük, hogy $n \geq 3$.

Tegyük fel, indirekten, hogy A egy megoldása a bizánci megegyezésnek $3 \leq n \leq 3f$ mellett. Megmutatjuk, hogyan transzformálható A egy olyan B megoldásba, mely három, P_1, P_2, P_3 folyamattal egy hibát eltűrve megoldja a bizánci megegyezést. Mindegyik B-beli folyamat nagyjából az A-beli folyamatok egyharmadát fogja szimulálni.

Az A-beli folyamatokat három nemüres alhálózatra bontjuk, legyenek ezek I_1, I_2 és I_3, mindegyikük legalább f méretű. A B-beli P_i folyamat a következő módon fogja szimulálni I_i-t.

> **B:**
> A P_i folyamatok mindegyike nyomon követi az összes I_i-beli folyamat állapotát, I_i minden eleméhez hozzárendeli saját kezdeti értékét, és szimulálja az I_i-beli folyamatok lépéseit, csakúgy mint az I_i-beli párok közti üzeneteket. Azokat az üzeneteket, melyek I_i-beli folyamatok egy másik alhálózathoz küldenek, P_i elküldi ahhoz a folyamathoz, mely a másik alhálózatot szimulálja. Ha tetszőleges számú folyamat v döntést hoz I_i-ben, akkor P_i döntése is v lesz. (Amennyiben egynél több ilyen érték lenne, P_i tetszőlegesen választ közülük.)

Bebizonyítjuk, hogy B helyesen oldja meg a bizánci megegyezést három folyamatra. Válasszuk A-ban pontosan azokat a folyamatokat hibásnak, amelyeket a B-beli hibásan viselkedő folyamat szimulál[3]. Rögzítsük B egy tetszőleges α végrehajtási sorozatát, melyben legfeljebb egy folyamathiba lép fel, és legyen α' az az A-beli végrehajtás, melyet ez szimulál. Mivel mindegyik B-beli folyamat legalább f számú A-beli folyamatot szimulál, legalább f hibás folyamat van α'-ben. Továbbá feltettük, hogy A megoldja a bizánci megegyezést n folyamat és legfeljebb f hiba esetében, így a szokásos megegyezési, érvényességi és befejeződési feltételek a bizánci megegyezésre fennállnak α'-ben.

Bebizonyítjuk, hogy ezek a feltételek α-ra is átvihetők. A befejeződés igazolásához legyen P_i egy B-beli hibátlan folyamat. Ekkor P_i legalább egy A-beli P_j folyamatot szimulál, és P_j biztosan hibátlan, mivel P_i is az. Az α'-re fennálló befejeződési feltételből következik, hogy P_j-nek végül döntenie kell; és amint ezt megtette, P_i is dönt (ha eddig még nem tett volna így).

Az érvényesség belátásához tudjuk, hogy amennyiben a B-beli összes folyamat egy adott v értékkel kezd, akkor az összes hibamentes A-beli folyamat is v értékkel kezd. Az α' érvényességéből következik, hogy az α'-beli hibátlan folyamatok döntése csak v lehet. Ekkor csak v lehet a döntése az α-beli hibátlan folyamatoknak is.

A megegyezés fennállásához tegyük fel, hogy P_i és P_j hibátlan folyamatok B-ben. Ekkor ezek csak A-beli hibátlan folyamatokat szimulálnak. Az α'-re fennálló megegyezésből adódik, hogy ezek a szimulált folyamatok mind megegyeznek, tehát P_i és P_j is megegyezik.

[3]Felhasználjuk azt a technikai sajátosságot, hogy a bizánci hibás folyamatok teljesen helyesen is viselkedhetnek, hogy ez az osztályozás fenntartható legyen.

Összefoglalva azt kapjuk, hogy B megoldja a bizánci megegyezési problémát három folyamatra egy hiba eltűrése mellett. Ez ellentmond a 6.26. lemmának. □

6.5.. Bizánci megegyezés általános gráfokban

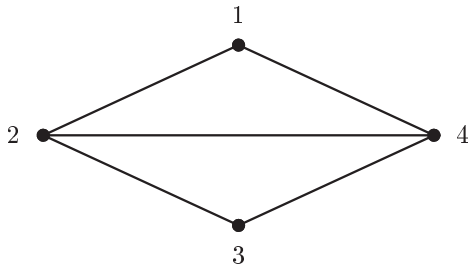
A fejezetben eddig a megegyezési problémákat csak teljes gráfokban vizsgáltuk. Az n csúcsú teljes gráfokra a 6.3. és 6.4. alfejezetekben beláttuk, hogy a bizánci megegyezés csak akkor oldható meg, ha $n > 3f$ fennáll. Ebben a részben általános hálózati gráfban vizsgáljuk a bizánci megegyezési feladatot. Pontosan jellemezzük azokat a gráfokat, amelyekben a feladat megoldható.

Vegyük először azt, amikor a hálózat gráfja egy legalább 3 csúccsal rendelkező fa, ekkor nincs esélyünk a bizánci megegyezési feladat megoldására, ha akár egy folyamat is hibásan viselkedik, hisz bármely hibás folyamat, ami nem levél, lényegében „szétválaszthatja" a fa egyik részében lévő folyamatokat a másik rész folyamataitól. A különböző részben elhelyezkedő hibátlan folyamatok képtelenek a valós kommunikációra, még kevésbé a megegyezésre. Hasonlóan kézenfekvőnek látszik, hogy ha f számú csúcs elegendő a gráf szétválasztására, akkor a bizánci megegyezés nem oldható meg f hibás folyamatra.

A megérzés formalizáláshoz a következő gráfelméleti fogalmat használjuk. Egy G gráf *összefüggőségének* mértéke, jelölje *össz_függ*(G), definíció szerint azon csúcsok számának minimuma, melyeket eltávolítva a gráfból, egy nem összefüggő gráfot, vagy a triviális egy csúcsból álló gráfot kapjuk. A G gráf *c-összefüggő*, ha *össz_függ*$(G) \geq c$.

6.5.1. példa. Összefüggőség

Ha egy fának legalább két csúcsa van, akkor az összefüggősége 1. Az n csúcsú teljes gráf összefüggősége $n - 1$. A 6.11. ábrán egy 2 összefüggőségű gráfot látunk. Ha a 2-es és 4-es csúcsot eltávolítjuk, akkor az eredmény az egymástól elválasztott 1-es és 3-as csúcs.



6.11.. ábra. Egy G gráf, melyre *össz_függ*$(G) = 2$.

A gráfelmélet egy klasszikus tételét fogjuk használni, melyet *Menger tételének* neveznek.

6.28. tétel. . (Menger tétele.) *A G gráf akkor és csak akkor c-összefüggő, ha G bármely két csúcsa legalább c csúcs-független úttal van összekötve.*

Most már jellemezhetjük azokat a gráfokat, melyekben a bizánci megegyezés egy adott számú hiba mellett megoldható. A jellemzéshez a gráf csúcsainak számát és az összefüggőség mértékét vizsgáljuk. A jellemzés megoldhatatlansági részének bizonyításához hasonló módszereket használunk, mint amit a 6.4. alfejezetben használtunk a hibás folyamatok számára adott alsó korlátnál.

6.29. tétel. . *A bizánci megegyezési feladat egy n csúcsú G hálózati gráfban f hiba eltűrése mellett akkor és csak akkor oldható meg, ha mindkét alábbi feltétel teljesül:*

1. $n > 3f$;
2. *össz_függ*$(G) > 2f$.

Bizonyítás. A 6.27. tételben már beláttuk, hogy $n > 3f$ folyamat szükséges, hogy a bizánci megegyezés megoldható legyen egy teljes gráfban. Könnyen beláthatjuk, hogy az $n > 3f$ feltétel fennállása tetszőleges (nem feltétlenül teljes) gráfok esetében is szükséges; hiszen ha egy algoritmus $n \leq 3f$ mellett nem teljes gráfban megoldaná a problémát, akkor azt használhatnánk egy n-csúcsú teljes gráfban is.

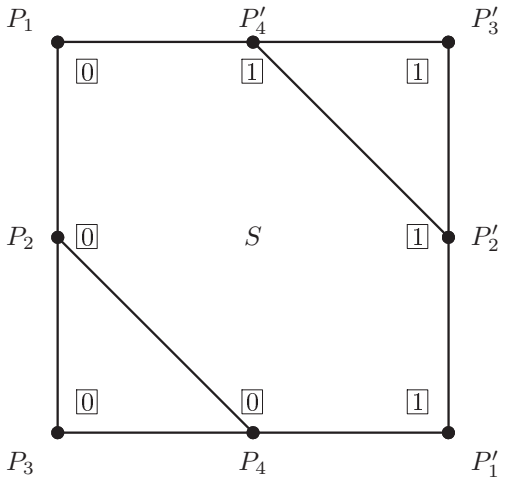
Most megmutatjuk, hogy igaz az állítás *ha* iránya, nevezetesen, a bizánci megegyezés lehetséges, ha $n > 3f$ és *össz_függ*$(G) > 2f$. Mivel G gráf $2f+1$-összefüggő, Menger tételéből, a 6.28. tételből következik, hogy G bármely két csúcsa között legalább $2f+1$ csúcs-független út van. Bármely két hibamenetes P_i és P_j között megvalósítható megbízható kommunikáció úgy, hogy P_i üzenetet küld P_j-nek a köztük lévő $2f+1$ számú útvonalon. Mivel a hibás folyamatok száma legfeljebb f, a P_j folyamathoz a különböző útvonalakon érkező üzenetek többsége helyes lesz.

Ha a hibamentes folyamatok közül bármely kettő között megbízható a kommunikáció, akkor a bizánci megegyezés megoldható tetszőleges olyan algoritmus szimulációjával, mely egy n-csúcsú teljes gráfban megoldja a problémát. A megbízható kommunikáció fentebb megadott megvalósítását használjuk a teljes gráfbeli csúcstól csúcsig történő adatátvitel helyett. Természetesen a bonyolultság megnövekszik, de most ez nem lényeges, – az algoritmus helyesen működik.

Térjünk rá a bizonyítás érdekesebb részére, belátjuk, hogy a bizánci megegyezés csak akkor oldható meg, ha *össz_függ*$(G) > 2f$. Leegyszerűsítve a problémát, azt az esetet vizsgáljuk, amikor $f = 1$; a nagyobb f értékekre történő, hasonlóan elvégezhető bizonyítást a 6-39. gyakorlatra hagyjuk.

Legyen G egy olyan gráf, melyre *össz_függ*$(G) \leq 2$, és amelyben az A algoritmussal megoldható a bizánci megegyezés egy hiba esetében. Ekkor két olyan csúcsa van G-nek, melyek vagy szétválasztják G csúcsait, vagy egy egy-csúcsú gráfra redukálják. Ha egy-csúcsú gráfra redukálják, akkor G csak három csúcsból áll, és azt már tudjuk, hogy egy három csúcsból álló gráfban egy hiba esetében nem oldható meg a bizánci megegyezés. Így feltehetjük, hogy a két csúcs szétválasztja G csúcsait.

Ekkor a 6.11. ábrához hasonló a kép, kivéve hogy az 1-es és 3-as csúcs helyettesíthető tetszőleges összefüggő részgráfokkal, és a 2-es és 4-es csúcs össze lehet kötve az összefüggő részgráfok mindegyik csúcsával. (Hiányozhat a 2-es és 4-es csúcsot összekötő él is, de ez csak rontaná a helyzetet.) Ismét az egyszerűség kedvéért azt az esetet vizsgáljuk, amikor az 1-es és 3-as csak egyszerű csúcsok. Egy S rendszert építünk A két példányának összerakásával. A folyamatok egyik példányát a 0 bemenő értékkel indítjuk, a másikat 1-gyel, ahogy a 6.12. ábrán látjuk. A 6.26. lemma bizonyításához hasonlóan, S az adott bemenetre jól definiált módon viselkedik. Ismét úgy jutunk ellentmondásra, hogy belátjuk, nem lehetséges ez a viselkedés.



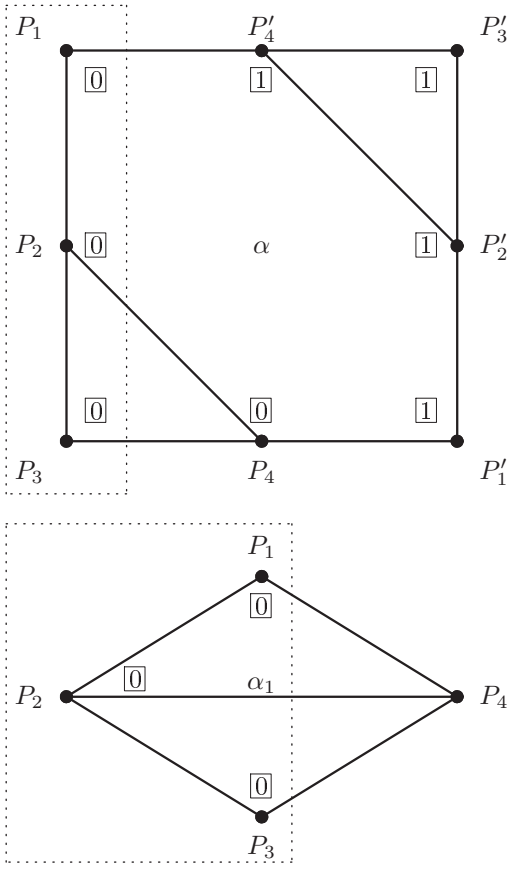
6.12.. ábra. Az A két példányának kombinációjaként kapjuk meg az S algoritmust.

Tegyük fel tehát, hogy S-ben a folyamatok a 6.12. ábrán látható kezdeti értékekkel indulnak, azaz a másolat folyamatok 0-val, az eredetiek 1-gyel; legyen α az eredményül kapott végrehajtás.

Tekintsük az α végrehajtási sorozatot a P_1, P_2 és P_3 folyamatok szemszögéből. A folyamatoknak úgy tűnik, mintha az A rendszerben futnának egy α_1 végrehajtási sorozatban, amelyben P_4 hibás. Lásd a 6.13. ábrát. Ekkor a bizánci megegyezés helyességének feltételeiből adódik, hogy végül α_1-ben P_1, P_2 és P_3 döntése 0 lesz. Minthogy α megkülönböztethetetlen α_1-től a P_1, P_2 és P_3 folyamatok számára, mindhármójuk döntése az α-ban is 0 lesz.

Majd tekintsük az α végrehajtást a P'_1, P'_2 és P'_3 folyamatok szemszögéből. E három folyamatnak úgy tűnik, mintha az A rendszerben futnának egy α_2 végrehajtási sorozatban, amelyben P_4 hibás. Lásd a 6.14. ábrát. Az előzőhöz hasonló bizonyítással azt kapjuk, hogy P'_1, P'_2 és P'_3 folyamatok döntése 1 lesz az α-ban.

Végül tekintsük az α végrehajtási sorozatot a P_3, P_4 és P'_1 folyamatok szemszögéből. Ezeknek a folyamatoknak úgy tűnik, mintha az A rendszerben futnának egy α_3 végrehajtási sorozatban, amelyben P_2 hibás. Lásd a 6.15. ábrát. A bizánci

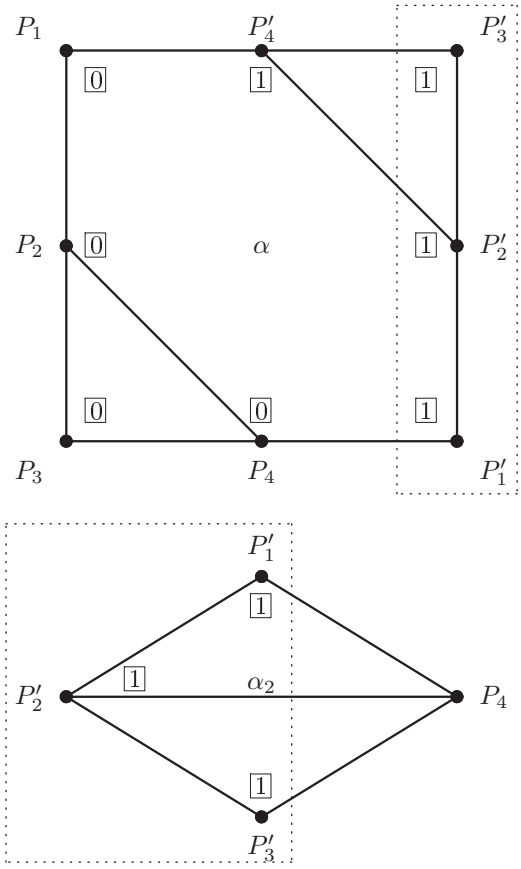


6.13.. ábra. Az α és α_1 megkülönböztethetetlen P_1, P_2 és P_3 folyamatok számára.

megegyezés helyességének feltételeiből adódik, hogy végül e három folyamat döntést hoz az α_3-ban, és döntésük azonos lesz. Ugyanez igaz α-ban.

De ez ellentmondás, mivel már beláttuk, hogy P_3 döntése csak 0 lehet α-ban, P'_1 döntése pedig csak 1. Ebből következik, hogy nem lehet megoldani a bizánci megegyezést a G gráfban *össz_függ*$(G) \leq 2$ és $f = 1$ esetében.

Az eredmények $f > 1$ esetre történő általánosításához ugyanezeket a diagramokat használhatjuk úgy, hogy a P_2 és P_4 csúcsokat az I_2 és I_4, legfeljebb f csúcsból álló halmazokkal helyettesítjük, a P_1 és P_3 csúcsokat pedig tetszőleges I_1 és I_3, csúcsokat tartalmazó halmazokkal. Az összes I_2 és I_4-beli csúcsot eltávolítva, I_1 és I_3 elszakad egymástól. A 6.11. ábra éleit az I_1, I_2, I_3 és I_4-beli csúcsokból álló csoportok közötti élek kötegeinek tekinthetjük. □



6.14.. ábra. Az α és α_2 megkülönböztethetetlen P'_1, P'_2 és P'_3 folyamatok számára.

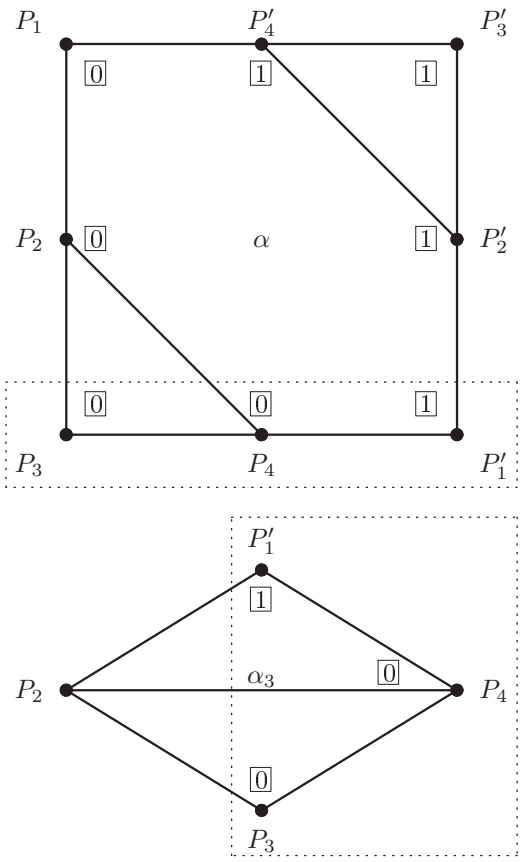
6.6.. Gyenge bizánci megegyezés

Ugyanaz az általános bizonyítási séma, amit a 6.4. és 6.5. alfejezetekben használtunk, hogy bebizonyítsuk a megoldhatatlansági eredményeket a bizánci megegyezésre $n \leq 3f$ vagy *össz_függ* $\leq 2f$ esetében, alkalmazható egyéb megegyezési problémák megoldhatatlansági eredményeinek bizonyításához. Például, ebben az alfejezetben megmutatjuk, hogyan alkalmazható ez a módszer a bizánci megegyezés egy gyengített változatára, melyet *gyenge bizánci megegyezésnek* hívnak.

Az egyetlen különbség a gyenge bizánci megegyezés és az általános bizánci megegyezés között az érvényességi feltételben van. A gyenge bizánci megegyezés érvényessége a következő.

Érvényesség. Ha nincs hibás folyamat és mindegyik folyamat ugyanazzal a $v \in V$ kezdeti értékkel indul, akkor csak v a lehetséges döntési érték.

Az általános bizánci modellben, ha az összes hibátlan folyamat ugyanazzal a



6.15.. ábra. Az α és α_3 megkülönböztethetetlen P_3, P_4 és P'_1 folyamatok számára.

v kezdeti értékkel indul, akkor mindannyiukra nézve kötelező a v döntés, *akkor is, ha vannak hibás folyamatok*. A gyenge bizánci megegyezésben csak akkor várjuk el tőlük a v döntést, ha nincsenek hibák.

Mivel a feladat kikötése gyengébb, mint az általános esetben, ezért az algoritmus, melyet az általános bizánci problémára adtunk, működik a gyenge bizánci megegyezésre is. Másrészről, a megoldhatatlansági eredmények nem vihetők át közvetlenül; elképzelhető, hogy a gyenge bizánci megegyezésre léteznek hatékonyabb algoritmusok. Jóllehet, ki fog derülni (egy apró technikai feltétel mellett), hogy a folyamatok darabszámára és a gráf összefüggésére fennálló korlát továbbra is igaz. (A technikai feltétel, amire szükségünk van: feltesszük, hogy $n \geq 3$, mivel a gyenge bizánci megegyezés $n = 2$ esetére van egy triviális algoritmus.)

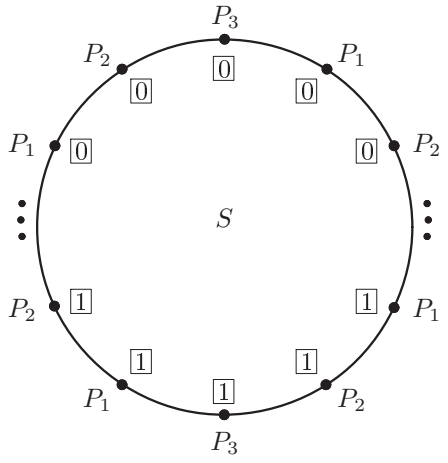
6.30. tétel. . *Legyen $n \geq 3$. A gyenge bizánci megegyezési feladat akkor és csak akkor oldható meg egy n-csúcsból álló G hálózati gráfban f hiba eltűrése mellett, ha mindkét alábbi állítás teljesül:*

1. $n > 3f$;
2. *össz_függ*$(G) > 2f$.

Bizonyítás. A *ha* irány következik azon protokollok létezéséből, melyet a 6.29. tételben az általános bizánci megegyezésre adtunk. Bebizonyítjuk, hogy három folyamat nem képes a gyenge bizánci megegyezés megoldására egy hibalehetőség mellett, és a 6-43. gyakorlatra hagyjuk az állítás $f > 1$ esetre történő kiterjesztésének, valamit az összefüggőségre vonatkozó állításnak a bizonyítását. Az egyszerűség kedvéért tegyük fel, hogy $V = \{0, 1\}$.

Legyen A egy három-folyamatos algoritmus, mely megoldja a gyenge bizánci megegyezést a P_1, P_2 és P_3 folyamatokra, még ha az egyik hibás is. Legyen α_0 A-nak az a végrehajtási sorozata, melyben mindhárom folyamat a 0 értékkel indul és hiba nem lép fel. A befejeződési és érvényességi szabályokból következik, hogy végül mindhárom folyamat 0 döntést hoz az α_0-ban; legyen r_0 azon menetek sorszámai közül a legkisebb, melyre mindegyik folyamat döntésre jut. Hasonlóképpen legyen α_1 az a végrehajtás, melyben mindegyik folyamat 1-gyel kezd és hiba nem lép fel, így végül mindegyik folyamat döntése 1 lesz α_1-ben. Legyen r_1 az ehhez szükséges menetek száma és válasszuk r-nek az $r \geq \max\{r_0, r_1, 1\}$ értéket.

Készítsünk egy új S rendszert, az A rendszer $2r$ számú példányának egy kör mentén való elhelyezésével, azaz $6r$ folyamattal, melyből $3r$ a kör „felső felén" helyezkedik el, $3r$ pedig az „alsó felén". A felső félkörön lévőket 0 kezdeti értékkel indítjuk, az alsó félkörön lévőket 1-gyel. Az elrendezést a 6.16. ábrán láthatjuk. (Most nem bonyolítjuk az ábrát az eredeti és többi, több példányban szereplő A-beli folyamat megkülönböztető jelölésével.) Legyen α az eredményül kapott S-beli végrehajtás.



6.16.. ábra. Az A $2r$ darab példányának együtteseként megkapjuk az S rendszert.

A 6.26. lemma bizonyításához hasonlóan megmutathatjuk, hogy bármely két szomszédos S-beli folyamatnak ugyanarra a döntésre kell jutnia az α végrehajtási

sorozatban, mivel a két folyamat szempontjából úgy tekinthetjük, hogy egy háromszögben vannak elhelyezve, egy harmadik, hibás folyamattal összekapcsolva. Ebből az következik, hogy S mindegyik folyamata *ugyanazt* a döntést hozza az α végrehajtási sorozatban. Tegyük fel, az általánosság megsértése nélkül, hogy valamennyiük döntése 1.

Most, hogy ellentmondásra jussunk, belátjuk, hogy néhány, a felső félkörön elhelyezkedő folyamatnak 0 döntést kell hoznia. Legyen B tetszőleges, az S felső félkörén egymás után következő, $2r + 1$ darabszámú folyamatból álló „blokk"; ezek mindegyike a 0 kezdeti értékkel indul α-ban. Ekkor az összes B-beli folyamat ugyanabból az állapotból indul α-ban, mint az azonos nevű folyamat α_0-ban, és ugyanazokat az üzeneteket küldi az első menetben. Így, az első menetben az összes B-beli folyamat, *kivéve esetleg a blokk két szélén lévő egy-egy folyamatot*, ugyanazt az üzenetet kapja α-ban, mint a névrokonaik α_0-ban, így azonos állapotban maradnak és azonos üzenetet küldenek a második menetben mindkét végrehajtási sorozatban. A második menetben mindegyik B-beli folyamat, kivéve kettőt-kettőt a blokk két szélén, ugyanazt az üzenetet kapja, és ugyanabban az állapotban marad mindkét végrehajtási sorozatban. Ezt folytatva, a k-adik, $1 \leq k \leq r$ menetben az összes B-beli folyamat, kivéve k darabot a két szélen, ugyanazt az üzenetet kapja, és ugyanabban az állapotban marad α-ban és α_0-ban. Más szavakkal az α és α_0 k menetet vizsgálva megkülönböztethetetlen valamennyi B-beli folyamat számára, kivéve a két szélen k darabot. Vázlatosan szólva, ennek az a magyarázata, hogy az információnak nincs ideje arra, hogy a blokk két széléről eljusson ezekhez a folyamatokhoz.

Gyakorlatilag az α és α_0 megkülönböztethetetlen a B blokk közepén elhelyezkedő P_i számára az r-edik menetben. Minthogy P_i folyamat 0 döntést hoz az r-edik menet végére az α-ban, így tesz az α_0-ban is. Ez ellentmond a kiinduláskor tett feltevésünknek, hogy P_i 1 döntést hoz α-ban. □

6.7.. A menetek száma megállási hibák esetében

A fejezet befejezéseként megmutatjuk, hogy a megegyezési feladat nem oldható meg kevesebb, mint $f+1$ menet alatt, akár bizánci, akár megállási hibák esetében. Más szavakkal nem létezik olyan megegyezési protokoll egyik hibafajta esetére sem, melyben az összes hibamentes folyamat az f-edik menetig döntést hoz.

Módszerünk az lesz, hogy feltesszük, hogy létezik f-menetes megegyezési algoritmus, és a bizonyítás során ellentmondásra jutunk. Kényelmi okokból néhány megszorítást teszünk a feltételezett algoritmusra vonatkozóan, de egyik sem okozza az általánosság megcsorbítását. Elsőként feltesszük, hogy a hálózati gráf teljesen összefüggő; egy gyors, nem teljes gráfon futó algoritmus teljes gráfon is működik, így biztosan nem veszítünk az általánosságból ezzel a megszorítással. Feltesszük továbbá, hogy az összes olyan folyamat, amelyik döntést hoz, az ezt pontosan az f-edik menet végén teszi, majd rögtön megáll. Ebben az esetben a bizánci megegyezést megoldó algoritmus szükségképp a megállási problémára is jó (lásd a 6.1. alfejezet megjegyzését a két feladat közötti kapcsolatról). Így célunk, a megoldhatatlansági eredmény eléréséhez elegendő, ha csak a megállási

hiba feladat vizsgálatára szorítkozunk. Azt is feltesszük, hogy mindegyik folyamat mindegyik másik folyamatnak küld üzenetet valamennyi k, $1 \leq k \leq f$ menetben (hacsak, és amíg nem hibázik). Végül, vizsgálatunkat leszűkítjük a $V = \{0, 1\}$ értékhalmazra.

Az 5. fejezetben tárgyalt összehangolt támadási problémához hasonlóan, a bizonyítás kényelmessé válik a kommunikációs minta fogalmának használatával, mely annak jelölése, hogy az egyes menetekben melyik folyamat mely más folyamatoknak küld üzenetet. Az előző definíciót alkalmazva a teljes gráf esetére a kommunikációs mintát úgy definiáljuk, hogy bármely részhalmaza a következő halmaznak:

$$\{(i, j, k) : 1 \leq i, j \leq n, i \neq j, 1 \leq k\}.$$

A kommunikációs minta nem ábrázolja az üzenetek tartalmát, csak azt, hogy melyik folyamat mely folyamatoknak, melyik menetben küld üzenetet.

A kommunikációs mintára vonatkozólag három megszorítást teszünk. Először, mivel a vizsgált algoritmus f menetes, csak azokat a kommunikációs mintákat tekintjük, melyekben mindegyik (i, j, k) hármasban $k \leq f$ fennáll. Másodszor, minthogy a megállási hiba modellel dolgozunk, a lehetséges kommunikációs minták eleget tesznek a következő megkötésnek: ha egy tetszőleges (i, j, k) hármas hiányzik a mintából, akkor ez igaz minden olyan (i, j', k') hármasra, ahol $k' > k$. Azaz, ha P_i folyamat hiba miatt nem küld üzenetet a k-adik menetben, akkor egyetlen rákövetkező menetben sem fog. Harmadszor, mivel azokat a végrehajtási sorozatokat vizsgáljuk, amelyekben legfeljebb f hiba fordul elő, az összes szóba jövő kommunikációs minta legfeljebb f számú hibás folyamatot tartalmaz. (Egy P_i folyamatot *hibásnak* tekintünk egy kommunikációs mintában, ha néhány (i, j, k), $k \leq f$ alakú hármas hiányzik a mintából.) Azt mondjuk (csak a fejezet hátralévő részében), hogy egy kommunikációs minta *jó*, ha kielégíti ezt a három feltételt.

6.7.1. példa. Jó kommunikációs minta

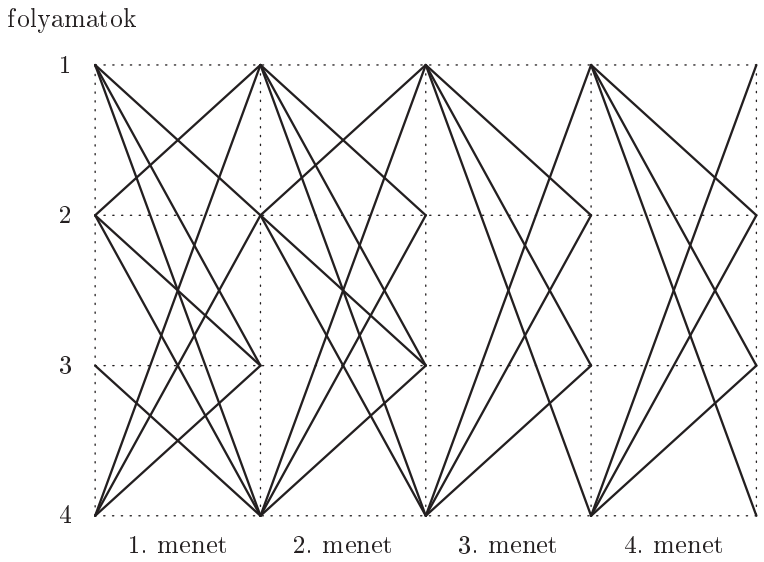
A jó kommunikációs minta egy példáját ábrázolja ($n = f = 4$ esetében) a 6.17. ábra. Ebben a mintában P_3 az első menetben P_4-nek küld üzenetet, de hiba miatt nem küld P_1-nek és P_2-nek. Tehát P_3 megáll az első menetben, és a későbbi menetekben már semmit sem küld. A P_2 folyamat is megáll a második menet végén. P_1 és P_4 hibátlanok.

Most definiáljuk a futam fogalmát, mely a következő kettő kombinációja:

1. bemeneti értékek hozzárendelése az összes folyamathoz;
2. egy jó kommunikációs minta.

(Ez hasonló ahhoz a fogalomhoz, amit az 5.2.1 részben ellenfélnek neveztünk.)

A megegyezési feladat egy bizonyos A algoritmusára minden egyes ρ futam természetes módon definiál egy megfelelő *végrehajtás*(ρ) végrehajtást. Nevezetesen a folyamatok kezdőállapotát meghatározza a bemeneti állapot összetevők beállítása a ρ-ban megadott bemeneti értékek szerint; az elküldésre kerülő üzeneteket a ρ kommunikációs mintája határozza meg, mivel A üzenet függvényét



6.17.. ábra. Példa egy jó kommunikációs mintára.

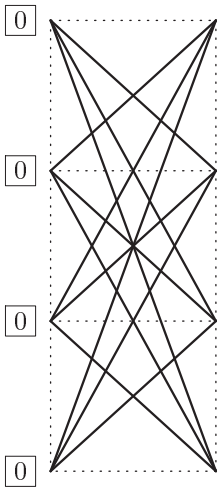
alkalmazzuk a küldő folyamat eredeti állapotára; a kezdőállapotokat követő állapotokat pedig A átmeneti függvénye határozza meg. (Miután egy folyamat hiba miatt többé nem küld üzenetet, arra már nem alkalmazzuk többet az állapotátmenet függvényt.)

Az $f = 1$ speciális esettel kezdjük a bizonyítást, hogy ezzel egy sejtést adjunk az alsó korlátra.

6.31. tétel. . *Legyen $n \geq 3$. Ekkor nincs olyan algoritmus, mely megoldja a megállási hiba problémát n folyamattal, egy hibát eltűrve úgy, hogy az összes hibátlan folyamat minden esetben az első menet végére döntést hozzon.*

Bizonyítás. Tegyük fel indirekten, hogy létezik egy ilyen A algoritmus, és az A kielégíti az alfejezet elején felsorolt összes feltételt.

Konstruáljunk egy láncot az A végrehajtási sorozataiból, melyek legfeljebb egy hibás folyamatot tartalmaznak, és (a) a láncbeli első végrehajtási sorozat egyedüli döntési értékként a 0-át tartalmazza, (b) a láncbeli utolsó végrehajtási sorozat egyedüli döntési értékként az 1-et tartalmazza, és (c) bármely két egymást követő végrehajtási sorozata megkülönböztethetetlen azon folyamat számára, mely mindkettőben hibátlan. Ekkor mivel a hibamentes P_i folyamat számára bármely két egymást követő végrehajtási sorozat megkülönböztethetetlen, P_i ugyanazt a döntést hozza mindkét végrehajtási sorozatban; ebből következik, hogy a két végrehajtási sorozatnak ugyanaz az érték az egyedüli döntési értéke. Ez azt jelenti, hogy a lánc *összes* végrehajtási sorozatának ugyanaz az érték az egyedüli döntési értéke, ami ellentmond az (a) és (b) kikötésnek.



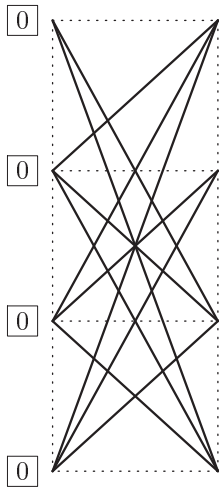
6.18.. ábra. A ρ_0 futam – minden bemenet 0, hibák nincsenek.

Kezdjük a láncot azzal az *végrehajtás*(ρ_0) végrehajtási sorozattal, melyet az a ρ_0 futam határoz meg, amelyben az összes folyamat a 0 kezdeti értékkel indul, és nincs hibás folyamat. Ezt a futamot ábrázolja a 6.18. ábra. Az érvényesség szerint *végrehajtás*(ρ_0)-ban az egyedüli döntési érték a 0, a következő végrehajtási sorozatot úgy kapjuk, hogy egy egyszerű üzenetet elhagyunk – amit P_1 folyamat P_2-nek küld. Az eredmény a 6.19. ábrán látható. Ez a végrehajtási sorozat megkülönböztethetetlen *végrehajtás*(ρ_0)-tól minden folyamat számára, kivéve P_1 és P_2 folyamatokat. Mivel $n \geq 3$, van legalább egy ilyen folyamat. Ez a folyamat hibamentes mindkét végrehajtási sorozatban.

Ezután eltávolítjuk P_1 folyamat P_3-nak küldött üzenetét; ez és az előző végrehajtási sorozat megkülönböztethetetlen minden folyamat számára, kivéve P_1 és P_3 folyamatokat, és legalább egy ilyen folyamat létezik. Ezt az utat folytatva, mindig egy üzenetet eltávolítva a P_1 folyamattól, az egymást követő menetek megkülönböztethetetlenek lesznek egy bizonyos hibátlan folyamat számára.

Mikor eltávolítottuk P_1 összes üzenetét, azzal folytatjuk, hogy megváltoztatjuk P_1 bemeneti értékét 0-ról 1-re. Természetesen az eredményül kapott végrehajtási sorozat megkülönböztethetetlen az előzőtől valamennyi folyamat számára, kivéve P_1-et, minthogy P_1 semelyiknek sem küld üzenetet a két végrehajtási sorozatban. Majd egyesével visszahelyezzük P_1 üzeneteit, és továbbra is igaz marad, hogy az egymást követő végrehajtási sorozatok páronként megkülönböztethetetlenek egynéhány hibamentes folyamat számára. Ezzel a módszerrel elérünk a *végrehajtás*(ρ_1)-hez, ahol ρ_1 definíció szerint az a végrehajtás, melyben P_1 bemeneti értéke 1, az összes többi folyamaté 0, és nincsenek hibák.

Majd ezt az eljárást megismételjük a P_2 folyamatra, elsőként egyesével eltávolítjuk az üzeneteit, utána kicseréljük P_2 bemeneti értékét 0-ról 1-re, és visszahelyezzük az üzeneteit. Eredményül kapjuk a *végrehajtás*(ρ_2) végrehajtást, ahol



6.19.. ábra. Az eredmény, ha egy üzenetet eltávolítunk a ϱ_0 futamból.

ρ_2 az a futam, melyben a P_1 és P_2 folyamat bemeneti értéke 1, a többié 0, és nincsenek hibák. Megismételve ezt a $P_3, \ldots, P_n$ folyamatra végül a *végrehajtás*(ρ_n) végrehajtási sorozatot kapjuk, ahol ρ_n az a futam, melyben az összes folyamat az 1 értékkel kezd és nincsenek hibák.

Ezzel elkészítettük azt az *végrehajtás*(ρ_0)-tól *végrehajtás*(ρ_n)-ig tartó láncot, mely eleget tesz a (c) kikötésnek. Azonban az érvényességi feltételből következik, hogy az egyedüli döntési érték a *végrehajtás*(ρ_0)-ban 0, a *végrehajtás*(ρ_n)-ben viszont 1, ami ellentmond (a) és (b) kikötéseknek. Tehát megkaptuk azt a láncot, ami ellentmondáshoz vezet. □

Mielőtt az általános esetre rátérünk, még egy előzetes elemzést végzünk – az $f = 2$ esetre.

6.32. tétel. . *Legyen $n \geq 4$. Ekkor nincs olyan algoritmus, mely megoldja a megállási hiba problémát n folyamattal, két hibát eltűrve úgy, hogy az összes hibátlan folyamat minden esetben a második menet végére döntést hozzon.*

Bizonyítás. Tegyük fel most is, hogy létezik ilyen algoritmus. Az előzőhöz hasonló módon, ismét készítünk egy láncot, mely eleget tesz ugyanazon (a), (b), (c) feltételeknek, mint amit az előző bizonyításban adtunk. Minden i, $0 \leq i \leq n$ esetében, ρ_i jelölje azt a futamot, melyben a $P_1, \ldots P_i$-nek 1 a bemenete, $P_{i+1}, \ldots P_n$-nek pedig 0, és nincsenek hibák. A lánc az *végrehajtás*(ρ_0)-val indul, *végrehajtás*(ρ_n)-nel fejeződik be, köztük az összes *végrehajtás*(ρ_i) végrehajtási sorozattal.

Az *végrehajtás*(ρ_0)-val kezdve, elsőként a P_1 folyamat kiiktatása a célunk. Amikor csak egy menettel foglalkoztunk, akkor egyszerűen egyesével eltávolítottuk P_1 üzeneteit. Most probléma nélkül megtehetjük, hogy egyesével eltávolítjuk

P_1 második menetbeli üzeneteit, de ha eltávolítjuk P_1 egy másik, P_i folyamat számára küldött első menetbeli üzenetét egy lépésben, akkor már nem marad igaz, hogy a két egymást követő menet megkülönböztethetetlen a többi hibamentes folyamat számára. Ez abból adódik, hogy P_i a második menetben értesítheti a többi folyamatot, hogy kapott-e üzenetet az első menetben P_1-től.

Ezt a problémát úgy oldjuk meg, hogy több lépésben távolítjuk el P_1 folyamat első menetbeli P_i-nek küldött üzenetét. A közbenső végrehajtási sorozatokban P_1 és P_i folyamatokat hibásnak tekintjük; ez megengedhető, hisz $f = 2$. Kezdjük tehát azzal a végrehajtási sorozattal, melyben P_1 üzenetet küld P_i-nek az első menetben és P_i hibátlan. Egyesével eltávolítjuk P_i második menetbeli üzeneteit amíg megkapjuk azt a végrehajtási sorozatot, amelyben P_1 küld P_i-nek üzenetet az első menetben, de P_i nem küld üzenetet a második menetben. Majd eltávolítjuk P_1 első menetbeli P_i-nek küldött üzenetét; az eredményül kapott végrehajtás megkülönböztethetetlen az előzőtől az összes folyamat számára, kivéve P_1 és P_i folyamatokat. Majd egyesével visszahelyezzük a P_i által küldött második menetbeli üzeneteket, amíg megkapjuk azt a végrehajtási sorozatot, amelyben P_1 nem küld P_i-nek üzenetet az első menetben, és P_i hibátlan. Ezzel elértük a célunkat, eltávolítottuk P_1 első menetbeli, P_i-nek küldött üzenetét, miközben az egymást követő végrehajtási sorozatok párosával megkülönböztethetetlenek maradtak egynéhány hibátlan folyamat számára.

Ezzel a módszerrel egyesével eltávolítjuk P_1 első menetbeli üzeneteit, amíg P_1 egyetlen üzenetet sem küld. Ekkor P_1 bemeneti értékét 0-ról 1-re cseréljük, ahogy az előbb is tettük. Folytatjuk az eljárást, csak „ellenkező irányban", egyesével visszahelyezve P_1 első menetbeli üzeneteit. Megismételve ezt az eljárást a $P_2, \ldots, P_n$ folyamatokra megkapjuk a kívánt láncot. □

Most bebizonyítjuk az általános tételt.

6.33. tétel. . *Legyen $n \geq f + 2$. Ekkor nincs olyan algoritmus, mely megoldja a megállási hiba problémát n folyamattal, f hibát eltűrve úgy, hogy az összes hibátlan folyamat minden esetben a f-edik menet végére döntést hozzon.*

A 6.31. és 6.32. tételek bizonyításaiban megtalálhatjuk a 6.33. tétel bizonyításának főbb gondolatait. Az általános eset bizonyításához hosszabb láncokat készítünk, f darab folyamathibát megengedve. A bizonyítás formálisabb lesz, mint a 6.31. tétel és a 6.32. tétel bizonyítása. Ehhez néhány jelölést vezetünk be.

Elsőként, ha ρ és ρ' futamokban a P_i folyamat hibátlan, akkor a $\rho \overset{i}{\sim} \rho'$ azt jelenti, hogy *végrehajtás*$(\rho) \overset{i}{\sim}$ *végrehajtás*(ρ') – vagyis a ρ és ρ' futamok által generált végrehajtások megkülönböztethetetlenek P_i folyamat számára. Jelölje $\rho \sim \rho'$ azt, ha fennáll a $\rho \overset{i}{\sim} \rho'$ néhány P_i folyamatra, melyek ρ és ρ' futamok mindegyikében hibátlanok. Továbbá jelölje a $\sim$ reláció $\rho \approx \rho'$ tranzitív lezárását.

Majd vegyük észre, hogy mindazon kommunikációs mintának, mely a 6.31. és 6.32. tételek bizonyításaiban szereplő láncokban előfordul, van egy bizonyos egyszerű tulajdonsága. Ezt a tulajdonságot jellemezzük a következő definícióval. Azt mondjuk, hogy egy jó kommunikációs minta *szabályos*, ha minden k $(0 \leq k \leq f)$ értékre legfeljebb k darab folyamat hibázik (egy üzenetet sem küld) a

k-adik menet végéig. Azt mondjuk, hogy a futam, vagy végrehajtás *szabályos*, ha a kommunikációs mintája szabályos.

Végül, ha ρ egy tetszőleges futam és $0 \leq k \leq f$, legyen $hn(\rho, k)$ az a futam – a ρ egy olyan változata, mely *hiba nélküli* k időpont után – melynek bemenete azonos ρ bemenetével, a kommunikációs mintája ugyanaz mint a ρ futamé az első k menetben, és a továbbiakban újabb hiba nem fordul elő. Nyilvánvaló módon adódnak a hn futamokra az alábbi tulajdonságok.

6.34. lemma. . *Ha a ρ egy szabályos futam, akkor*

1. *tetszőleges k $(0 \leq k \leq f)$ esetében $hn(\rho, k)$ szabályos;*
2. *ha ρ' azonos ρ-val, kivéve, hogy néhány P_i folyamat, mely ρ-ban hibázik, ρ'-ben későbbi menetben lesz hibás, akkor ρ' szabályos;*
3. *ha a $(k+1)$-edik menetben egyik folyamat sem hibázik, akkor $hn(\rho, k) = hn(\rho, k+1)$.*

A 6.33. tétel bizonyításának lelke a következő erős lemma, mely kimondja, hogy *bármely két* azonos bemenettel rendelkező szabályos végrehajtási sorozat között létrehozhatunk egy láncot.

6.35. lemma. . *Legyen A egy n-folyamatos algoritmus a megállási hiba problémára, mely f számú hibát eltűr, és melyben a hibamentes folyamatok minden esetben döntést hoznak az f-edik menet végéig. Legyen ρ és ρ' két szabályos futama az A algoritmusnak, melyekben a bemenet azonos, ekkor $\rho \approx \rho'$.*

Bizonyítás. A bizonyításhoz az alábbi paraméteres segédtételt fogjuk belátni. A lemma azonnal következik a $k = 0$ esetből.

6.36. segédtétel. . *Legyen k egy egész szám, $0 \leq k \leq f$. Legyen ρ és ρ' két szabályos futama az A algoritmusnak, melyekben azonos a bemenet és azonos a kommunikációs minta is k meneten keresztül. Ekkor $\rho \approx \rho'$.*

Bizonyítás. A 6.36. állítás bizonyítása k szerinti fordított indukcióval történik, a $k = f$ esettel kezdjük, és a $k = 0$ esettel fejezzük be.

Alapeset: $k = f$. Ez az eset triviálisan adódik a feltevésből, hogy ρ és ρ' bemente megegyezik, és kommunikációs mintájuk azonos az f-edik menetig, amiből az következik, hogy ρ és ρ' azonos.

Indukciós lépés: feltesszük, hogy $0 \leq k \leq f-1$, és az állítás igaz $k+1$ értékre. Elegendő belátnunk, hogy bármely szabályos ρ futamra fennáll, hogy $\rho \approx hn(\rho, k)$, mivel ezt kétszer alkalmazva megkapjuk a kívánt állítást. Rögzítsünk egy szabályos ρ futamot. A 6.34. lemmából következik, hogy $hn(\rho, k)$ szabályos.

Indukciós feltevésünk szerint $hn(\rho, k+1) \approx \rho$, tehát elegendő megmutatnunk, hogy $hn(\rho, k) \approx hn(\rho, k+1)$. Ha a $(k+1)$-edik menetben egyik folyamat sem hibázik, akkor a 6.34. lemmából következik, hogy $hn(\rho, k) = hn(\rho, k+1)$, és kész vagyunk. Tegyük fel tehát, hogy legalább egy folyamat hibázik a ρ futam $(k+1)$-edik menetében. Jelölje I azon folyamatok halmazát, melyek így tesznek.

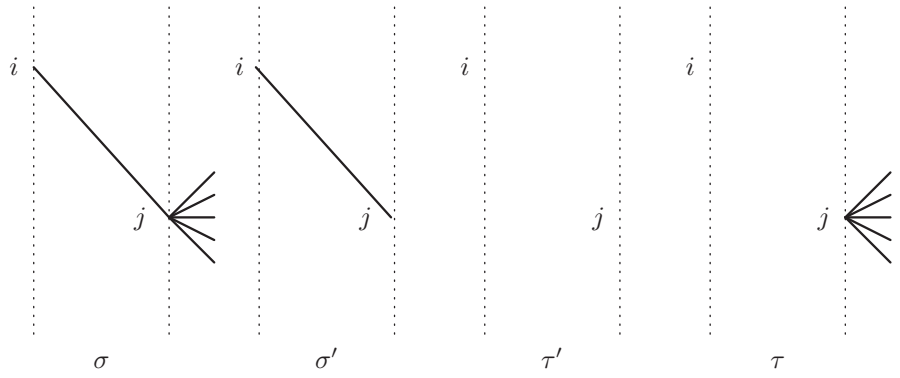
Legyen ρ_0 az a folyamat, mely azonos $hn(\rho, k)$-val, kivéve hogy, az összes I-beli folyamat hibázik az $(f+1)$-edik menet végéig. Ekkor a 6.34. lemma második állításából (ρ-ra alkalmazva) kapjuk, hogy ρ_0 szabályos.

Minthogy ρ_0 és $hn(\rho, k)$ szabályos futamok, melyek azonosak $k+1$ meneten keresztül, alkalmazhatjuk az indukciós feltevést, hogy megmutassuk, $\rho_0 \approx hn(\rho, k)$. Ezért annak bizonyításához, hogy $hn(\rho, k) \approx hn(\rho, k+1)$ fennáll, elegendő belátni, hogy $\rho_0 \approx hn(\rho, k+1)$.

Készíteni fogunk egy olyan szabályos futamokból álló láncot, mely összeköti a ρ_0 és $hn(\rho, k+1)$ futamokat. Az egyetlen különbség ρ_0 és $hn(\rho, k+1)$ között az, hogy néhány olyan üzenet, melyet ρ_0 futam $(k+1)$-edik menetében az I-beli folyamatok küldenek, hiányzik a $hn(\rho, k+1)$ futamban. Ezért egyesével eltávolítjuk ezeket az üzeneteket, úgy, hogy mást nem változtatunk a futamokban.

Példaként tekintsük a P_i által P_j-nek küldött üzenet eltávolítását, ahol $i \in I$. Legyen σ az a futam, mely tartalmazza az üzenetet, τ pedig az, amelyik nem; meg kell mutatnunk, hogy $\sigma \approx \tau$. Ha $k+1 = f$, akkor σ és τ megkülönböztethetetlenek P_i és P_j kivételével az összes folyamat számára; mivel $n \geq f+2$ és P_i hibás, ezért van legalább egy hibátlan folyamat. Tehát $\sigma \approx \tau$, amire szükségünk volt.

Másrészről, ha $k+1 \leq f-1$, akkor definiáljuk a σ' és τ' futamokat úgy, hogy külön-külön azonosak legyenek a σ és τ futamokkal, de P_j a $(k+2)$-edik menet elején hibás lesz (ha eddig még nem lett volna). Lásd a 6.20. ábrát.



6.20.. ábra. A P_i folyamat P_j számára küldött üzenetének eltávolítása a $(k+1)$-edik menetben a 6.36. tétel bizonyításában.

A σ' és τ' futamok szabályosak, mivel σ és τ mindegyike legfeljebb $k+1 \leq f-1$ hibát tartalmaz, és csak egy új hibát engedélyeztünk az új $(k+2)$-edik menetben. Továbbá az indukciós feltevés alapján $\sigma \approx \sigma'$ és $\tau \approx \tau'$. Fennáll $\sigma' \approx \tau'$ is, mert megkülönböztethetetlenek P_i és P_j kivételével az összes folyamat számára. Innen adódik, hogy $\sigma \approx \tau$.

Megmutattuk, hogy a ρ_0 és $hn(\rho, k+1)$ futamokat összekötő lánc létrehozható, így $\rho_0 \approx hn(\rho, k+1)$, ebből $\rho \approx hn(\rho, k)$, amire szükségünk volt. □

Korábban már említettük, hogy a 6.36. állításból közvetlenül adódik a 6.35.

lemma. □

Most kiterjesztjük a 6.35. lemmát különböző bemenetek esetére.

6.37. lemma. . *Legyen A egy n-folyamatos algoritmus a megállási hiba problémára, mely f számú hibát eltűr, és melyben a hibamentes folyamatok minden esetben döntést hoznak az f-edik menet végéig. Legyen ρ és ρ' két szabályos futama az A algoritmusnak, ekkor $\rho \approx \rho'$.*

Bizonyítás. A 6.35. lemma szerint minden ρ futam összefügg a hiba nélküli változatával, azaz $\rho \approx hn(\rho, 0)$. Így az általánosság csorbítása nélkül feltehetjük, hogy a lemma szerinti ρ és ρ' mindketten hiba nélküliek.

Ha ρ és ρ' bemenete ugyanaz, akkor a két futam megegyezik, és nincs mit bizonyítani.

Tegyük fel, hogy ρ és ρ' bemenete különbözik egy adott P_i esetében; legyen a bemenet 0 ρ-ban és 1 ρ'-ben. Legyen σ és σ' külön-külön azonos ρ és ρ' futamokkal, kivéve, hogy P_i az induláskor hibás lesz. Ekkor a 6.35. lemmából következik, hogy $\rho \approx \sigma$ és $\rho' \approx \sigma'$. Valamint $\sigma \approx \sigma'$, mivel σ és σ' megkülönböztethetetlenek mindegyik folyamat számára, kivéve P_i-t. Ebből következik, hogy $\rho \approx \rho'$, amit bizonyítani akartunk.

Végül tegyük fel, hogy ρ és ρ' bemenete nemcsak egy folyamat esetében különbözik. Készítünk egy hiba nélküli folyamatokból álló láncot ρ-tól ρ'-ig úgy, hogy egy lépésben mindig pontosan egy folyamat bemenetét változtatjuk meg a láncban. Az így kapott láncban minden egyes lépésre alkalmazhatjuk az előző esetet. Így beláttuk, hogy $\rho \approx \rho'$. □

A 6.37. lemmára támaszkodva könnyen bizonyítható a 6.33. tétel. Azt már tudjuk, hogy az összes szabályos futam összekapcsolható láncokkal; most megvizsgáljuk a döntési értékeket ezekben a futamokban. Legyen $n > f$, a befejeződési és megegyezési tulajdonságokból következik, hogy minden ρ futamra létezik egy egyedüli döntési érték, legyen ez a *dönt*(ρ), mely *végrehajtás*(ρ) során keletkezik. A következő lemma szerint azon futamokban, melyek között a $\sim$ vagy $\approx$ reláció fennáll, szükségszerűen ugyanaz a döntési érték születik.

6.38. lemma. .

1. *Ha $\rho \sim \rho'$, akkor dönt(ρ) = dönt(ρ').*
2. *Ha $\rho \approx \rho'$, akkor dönt(ρ) = dönt(ρ').*

Bizonyítás. Az első rész bizonyításához emlékezzünk vissza, hogy $\rho \sim \rho'$ azt jelenti, hogy létezik egy olyan P_i folyamat, mely hibátlan a ρ és ρ' futamokban, így *végrehajtás*(ρ) $\overset{i}{\sim}$ *végrehajtás*(ρ'). Ebből következik, hogy P_i ugyanazt a döntést hozza *végrehajtás*(ρ)-ban és *végrehajtás*(ρ')-ben. Tehát *dönt*(ρ) = *dönt*(ρ').

A második rész az elsőből következik.

□

6.33. tétel bizonyítása. Tegyük fel, hogy létezik egy ilyen algoritmus, legyen ez az A; feltesszük, hogy

A eleget tesz az alfejezet elején adott feltételeknek.

Legyen ρ_0 az A algoritmus egyik futama, melyben minden folyamat 0-val kezd, és nincsenek hibák, és ρ_1 egy másik futam, amelyben minden folyamat 1-gyel kezd, és nincsenek hibák. A 6.37. lemmából következik, hogy $\rho_0 \approx \rho_1$. Ekkor a 6.38. lemma második állítása szerint $\mathit{dönt}(\rho_0) = \mathit{dönt}(\rho_1)$. Az érvényességi szabályból viszont az következik, hogy $\mathit{dönt}(\rho_0) = 0$ és $\mathit{dönt}(\rho_1) = 1$, ami ellentmondás. □

Gyengébb érvényességi feltétel.. Vegyük észre, hogy a bizonyítás akkor is helytálló, ha az érvényességi feltételt gyengítjük, ahogy a 6.6. alfejezetben tettük, a gyenge bizánci megegyezésnél. Tehát azt is beláttuk, hogy a gyenge bizánci megegyezési feladat is legalább $f + 1$ menetes az $n \geq f + 2$ kikötés mellett.

6.8.. Megjegyzések a fejezethez

A fejezetben található eredmények nagy része a két alapműből származik, az egyik Pease, Shostak és Lamport [237], a másik Lamport, Shostak és Pease [187] munkája. A két cikk közli a bizánci megegyezéshez szükséges processzorok darabszámára vonatkozó $3f + 1$ alsó és felső korlátot, továbbá egy algoritmust a hitelesítéssel történő megegyezésre, mindegyik eredmény a teljesen összefüggő gráfok esetére vonatkozik. A második cikk a problémát nem folyamatokon, hanem a támadni készülő tábornokok példáján keresztül mutatja be. A *bizánci* elnevezést a második cikk vezeti be erre a hiba modellre.

Kissé bővebben, ez a két cikk definiálja a bizánci megegyezési problémát, mely a repülőgépek SIFT (Software-Implemented Fault Tolerance) irányító rendszerében felmerülő feladat absztrakciójaként vetődik fel [289]. A [237] cikkben leírt algoritmusok egy az *EIGY* fához hasonló exponenciális adatszerkezetet használnak; a bizánci megegyezés algoritmus az EIGYBiz-hez, a hitelesítést használó algoritmus pedig a az EIGYStop-hoz hasonló. A [187]-ben található algoritmusok nagyon hasonlítanak ezekhez, csak rekurzív megfogalmazásban. A [237]-ben közölt, a folyamatok darabszámára vonatkozó $n \leq 3f$ megoldhatatlansági eredmény magában foglalja az általunk is tárgyalt forgatókönyv pontos megadását. A [187]-ben található megoldhatatlansággal kapcsolatos bizonyítás mutatja be a 6.27. tétel bizonyításában használt három az egy ellen esetre való redukálást.

Dolev és Strong [93] által, a hitelesítéssel kibővített bizánci megegyezésre javasolt algoritmusok hasonlóak a HalmazTerjed és OptHalmazTerjed algoritmusokra. Dolev [94] vizsgálta a bizánci megegyezési problémát olyan gráfokra, melyekre nem kötjük ki a teljes összefüggőséget. Ő bizonyította a 6.29. tételben bemutatott összefüggőségi korlátokat, konkrét forgatókönyvek megadásán keresztül. Dolev, Reischuk és Strong [99] javasoltak „korán megálló" algoritmusokat bizonyos kedvező kommunikációs mintákra. Egyéb korán megálló algoritmusokat közöl Dwork és Moses [105] és Halpern, Moses és Waarts [145].

Bar-Noy, Dolev, Dwork és Strong definiálták az *EIGY* fa adatszerkezetet, valamint leírták az EIGYBiz algoritmust lényegében ugyanabban a formában, ahogy ebben a könyvben tárgyaltuk [39]. A TurpinCoan algoritmus leírása [279]-ben található.

A bizánci megegyezésre az első, kommunikációs bonyolultságot tekintve polinomiális algoritmust Dolev és Strong [101] javasolták; az algoritmust később Dolev, Fischer, Fowler, Lynch és Strong [96] továbbfejlesztették, korlátként a $2f + 3$ értéket megadva. Coan [82] készített egy kompromisszumos algoritmust, mely csökkentette a menetek számat $(1 + \epsilon)f$ értékre, tetszőleges $\epsilon > 0$ mellett. A következetes üzenetszórás alapfogalmat és a KövetkezetesÜzenSzórás algoritmust Srikanth és Toueg [269] vezeti be. A PoliBiz algoritmus Srikanth és Toueg [269] és Dolev és mások [96] által közölt algoritmusokon alapszik. További kutatások eredményeként Moses és Waarts [231], Berman és Garay [49] és Garay és Moses [133] előállítottak olyan $f + 1$ menetes algoritmusokat a bizánci megegyezésre, melyek kommunikációs bonyolultsága polinomiális, az utolsó ezek közül eléri az $n = 3f+1$ legkisebb korlátot a folyamatok darabszámát tekintve. Sajnos, ezek az algoritmusok bonyolultak.

Említettük már, hogy a folyamatok darabszámára vonatkozó $n > 3f$ alsó korlát bizonyítása [237, 187] cikkekben, az összefüggőségre vonatkozó alsó korlát bizonyítása pedig [94]-ben található. Bár a könyvben szereplő bizonyítást Fischer, Lynch és Merritt [122] adták. Menger tételét Menger bizonyította [225] és Harary könyvében [147] jelent meg.

A gyenge bizánci megegyezést Lamport definiálja [178]. A gyenge bizánci megegyezéshez szükséges folyamatok számának alsó korlátjára vonatkozó eredmény Lamporttól származik [178], de a könyvünkben adott bizonyítás Fischer, Lynch és Merritt munkája[122].

A megegyezés eléréséhez szükséges menetek számára vonatkozó alsó becslést elsőként Fischer és Lynch [119] bizonyították a bizánci hiba modellben. Később ezt az eredményt Dolev és Strong [93], valamint DeMillo, Lynch és Merritt [88] kiterjesztik a hitelesítéses bizánci hiba modell esetére. A megállási hibák esetére való kiterjesztés Merrittnek [226] tulajdonítható, Dolev és Strong elképzeléseit [101] alapul véve. Az eredmény egy másfajta bizonyítását Dwork és Moses [105] adják; az ő bizonyításukban a különböző esetek futási idejének finomabb elemzését is megtaláljuk. Feldman és Micali [113] konstans idejű véletlenített megoldást adnak „titok megosztás" technikát használva.

Fischer dolgozatában [117] áttekinti a megegyezési problémával kapcsolatos korábbi eredmények nagy részét.

Igen tekintélyes fejlesztő munka folyik a Draper Laboratóriumban hibatűrő mikroprocesszorok és processzor hibadiagnózis algoritmusok tekintetében, a bizánci megegyezésre [172, 173] támaszkodva. Az eredményeket olyan biztonsági szempontból kiemelt alkalmazási területeken használják fel, mint az embernélküli vízalatti járművek, a nukleáris támadófegyverekkel felszerelt tengeralattjárók, valamint az atomerőművek irányítása.

6.9.. Gyakorlatok

6-1. Bizonyítsuk be, hogy tetszőleges olyan algoritmus, mely megoldja a bizánci megegyezés problémát, megoldja a megegyezési problémát megállási hibák esetében is, ha a megállási hiba modellben úgy módosítjuk az érvényességi feltételt,

hogy csak a hibamentes folyamatok megegyezését követeljük meg.

6-2. Bizonyítsuk be, hogy tetszőleges olyan algoritmus, mely megoldja a bizánci megegyezés problémát és amelyben a hibátlan folyamatok mindig egyszerre, ugyanazon menetben hoznak döntést, megoldja a megegyezési problémát megállási hiba modellben is.

6-3. Bizonyítsuk be a 6.2. lemmát.

6-4. Kövessük nyomon a HALMAZTERJED algoritmus végrehajtását négy folyamattal és két hibával, melyben a folyamatok kezdőértékei rendre az 1, 0, 0, 0 értékek. Tegyük fel, hogy P_1 és P_2 folyamatok hibásak, P_1 az első menetben lesz hibás, miután egyedül a P_2 folyamatnak elküldte az üzenetet, P_2 pedig a második menetben lesz hibás, P_1-nek és P_3-nak küld üzenetet, viszont P_4-nek nem.

6-5. Tekintsük a HALMAZTERJED algoritmust f hibára. Tegyük fel, hogy az algoritmus $f+1$ menet helyett csak f menetben fut, ugyanazzal a döntési értékkel. Találjunk egy olyan végrehajtási sorozatot, mely megsérti a helyességi feltételeket.

6-6. Legfeljebb mennyi lehet a hibamentes folyamatok által hozott, egymástól különböző döntési értékek darabszáma, ha a HALMAZTERJED algoritmus $f + 1$ menet helyett csak f menetben fut.

6-7.

(a) Találjunk egy másik lehetséges, helyesen működő döntési szabályt a HALMAZTERJED algoritmusban, amelyik eltér szövegben megadottól.

(b) Adjunk pontos jellemzést azon döntési szabályok halmazáról, amelyek helyesen működnek.

6-8. Terjesszük ki a HALMAZTERJED algoritmust, a helyesség bizonyítását, és az elemzést, tetszőleges (nem szükségképp teljesen) összefüggő gráfokra.

6-9. Készítsük el az OPTHALMAZTERJED algoritmus kódját. Tegyük a szövegben adott bizonyítást teljessé a 6.5., 6.6. és 6.7. lemmák bizonyításával.

6-10. Tekintsük a következő egyszerű algoritmust a megállási hibák mellett történő megegyezésre, egy adott V értékhalmaz esetében. Legyen mindegyik folyamatnak egy *min_érték* változója, melyet induláskor a saját kezdeti értékére állít be. Az $f + 1$ menet mindegyikében a folyamatok közreadják *min_érték* változójuk értékét, majd újra beállítják úgy, hogy a minimuma legyen a *min_érték* változó eredeti értékének, valamint az üzenetekben kapott értékeknek. Végül a folyamat döntési értéke *min_érték* lesz. Készítsük el ennek az algoritmusnak a kódját és bizonyítsuk be (vagy direkt módon, vagy szimulációval), hogy helyesen működik.

6-11. Kövessük nyomon az EIGYSTOP algoritmus végrehajtását négy folyamattal és két hibával, melyben a folyamatok kezdőértékei rendre az 1, 0, 0, 0 értékek. Tegyük fel, hogy P_1 és P_2 folyamatok hibásak, P_1 az első menetben lesz hibás, miután egyedül a P_2 folyamatnak elküldte az üzenetet, P_2 pedig a második menetben lesz hibás, P_1-nek és P_3-nak küld üzenetet, viszont P_4-nek nem.

6-12. Bizonyítsuk be a 6.11. lemmát.

6-13. Bizonyítsuk be a 6.12. lemma első állítását.

6-14. Tekintsük az EIGYSTOP algoritmust f hibára. Tegyük fel, hogy az algoritmus $f+1$ menet helyett csak f menetben fut, ugyanazzal a döntési értékkel.

Találjunk egy olyan végrehajtást mely megsérti a helyességi feltételeket.

6-15. Legfeljebb mennyi lehet a hibamentes folyamatok által hozott, egymástól különböző döntési értékek darabszáma, ha az EIGYSTOP algoritmus $f+1$ menet helyett csak f menetben fut.

6-16. Egy másfajta bizonyítási módszer a HALMAZTERJED algoritmus helyességének bizonyítására, az EIGYSTOP algoritmusra vonatkozó szimulációs reláció módszere. Annak érdekében, hogy ezt megvalósítsuk, elsőként érdemes kiterjeszteni az EIGYSTOP algoritmust, és megengedni, hogy valamennyi P_i folyamat közreadja az összes menetben az összes értéket, nemcsak azon csúcsokhoz kapcsolt értékeket, melyek címkéje nem tartalmazza az i-t. Be kell látni, hogy a kiterjesztés nincs hatással a helyességre. Továbbá az EIGYSTOP leírásában néhány részletet ki kell egészíteni, igy például a *menetek* és *döntések* változók megfelelő módon történő kezelését. Ezután a HALMAZTERJED és a módosított EIGYSTOP algoritmusokat egymás mellett futtathatjuk, ugyanazzal a kezdeti értékhalmazzal indítva, és olyan hibákkal, melyek ugyanazon folyamatoknál pontosan azonos időben történnek.
Bizonyítsuk a HALMAZTERJED algoritmus helyességét ezzel a módszerrel. A bizonyítás lelke az alábbi szimulációs kapcsolat lehet, melyben a két algoritmus állapotai szerepelnek egyező számú menet végrehajtása után.

> **6.9.1. állítás.** *Tetszőleges r, $0 \leq r \leq f+1$ menet után az alábbi két állítás igaz.*
>
> *(a) A menetek és döntések változók értékei megegyeznek a két algoritmus állapotaiban.*
>
> *(b) Minden i értékre a W_i halmaz a* HALMAZTERJED *algoritmusban egyenlő az* EIGYSTOP *algoritmusban P_i fájának csúcsait díszítő érték-ekből álló halmazzal.*

Figyeljünk, hogy azokhoz az EIGYSTOP algoritmust kiegészítő invariáns állításokhoz, melyek a szimuláció létrehozásához szükségesek, minden szükséges definíciót megadjunk, valamint a szükséges bizonyításokat is végezzük el.

6-17. Bizonyítsuk be az OPTEIGYSTOP algoritmus helyességét az alábbi két módszer egyikével:

(a) az EIGYSTOP algoritmus szimulálásával, ahogy az OPTHALMAZTERJED algoritmus bizonyítását végeztük az HALMAZTERJED algoritmusra vonatkozó szimulációval;

(b) az OPTHALMAZTERJED-re vonatkozó szimulációval.

6-18. Bizonyítsuk be az EIGYSTOP és OPTEIGYSTOP algoritmusok helyességét a hitelesített bizánci hiba modellben. Néhány, az EIGYSTOP bizonyításához szükséges alapvető tényt fogalmaz meg a következő állítás, a 6.12. lemma állításaihoz hasonló módon.

6.9.2. állítás. *$f+1$ menet után fennállnak a következők.*

(a) Ha P_i és P_j hibamentes folyamatok, érték$(y)_i = v \in V$ és xj az y előtagja, akkor érték$(x)_j = v$.
(b) Ha v eleme tetszőleges hibamentes folyamat érték-eiből álló halmaznak, akkor v kezdőértéke volt valamely folyamatnak.
(c) Ha P_i egy hibamentes folyamat és $v \in V$ benne van P_i érték-eiből álló halmazban, akkor van olyan y címke, mely i-t nem tartalmazza, és melyre $v =$ érték$(y)_i$.

Ezek a digitális aláírás tulajdonságaiból következnek.

6-19. *Kutatási feladat:* definiáljuk a hitelesített bizánci hiba modellt formálisan, és bizonyítsuk a hatékonysággal és a korlátokkal kapcsolatos eredményeket.

6-20. Mutassunk példát az EIGYStop algoritmus olyan végrehajtási sorozatára, mely bebizonyítja, hogy az EIGYStop algoritmus nem oldja meg a bizánci hiba problémát.

6-21. Tekintsük az EIGYBiz algoritmust hét folyamattal és három menettel. Tetszőlegesen válasszunk két folyamatot, melyek hibásak lesznek, és adjunk véletlen bemeneti értéket valamennyi folyamatnak és azon üzeneteknek, melyeket a hibás folyamatok küldenek. Számítsuk ki a végrehajtási során előállított információt, és bizonyítsuk, hogy a helyességi feltételeket kielégítik.

6-22. Bizonyítsuk be, hogy az EIGYBiz algoritmusban nem szükséges az, hogy az *EIGY* fa valamennyi csúcsa közös legyen.

6-23. Tekintsük az EIGYBiz algoritmust. Keressünk olyan konkrét végrehajtásokat, melyek megmutatják, hogy az algoritmus rossz eredményt adhat, ha
(a) hét csúccsal, két hibával és két menetben fut;
(b) hat csúccsal, két hibával és három menetben fut.

6-24. A TurpinCoan algoritmus az első és második menetben az $n - f$ küszöb értéket használja. Milyen másik két küszöb érték lenne alkalmas, hogy az algoritmus továbbra is helyesen működjön?

6-25. Tekintsük a TurpinCoan algoritmust egy helyett *két*, F és G, hibás folyamatokat tartalmazó halmazzal. Mindkét halmaz legfeljebb f folyamatot tartalmaz. Az F-beli folyamatok helyesen működnek, kivéve az első és második menetet, melyben hibás üzenetet küldhetnek. A G-beli folyamatok a bináris bizánci megegyezés szubrutin végrehajtása alatt (és csak akkor) viselkedhetnek hibásan. Milyen helyességi feltételek biztosíthatók a kombinált algoritmust használva ezen feltételek mellett? Bizonyítsuk.

6-26. Legyen most $n > 4f$ a feltevésünk. Tervezzünk algoritmust, mely a bináris bizánci megegyezés szubrutinját használja, és megoldja a többértékű bizánci megegyezést. Az algoritmust a TurpinCoan algoritmusból kiindulva készíthetjük el, a két plusz menet helyett csak egyet használva.

6-27. Mutassuk meg, hogy nem létezik felső korlát arra az időre a KövetkezetesÜzenSzórás algoritmusban, amíg egy hibátlan folyamat elfogadhatja

az (m, i, r) üzenetet. Azaz tetszőleges t-re állítsuk elő a KövetkezetesÜzenSzórás algoritmus olyan végrehajtását, melyben néhány hibátlan folyamat az üzenetet az $r' \geq r + t$ menetben fogadja el.

6-28. Tervezhető-e algoritmus a következetes üzenetszórás megvalósítására a bizánci hiba modellben $f \gg 1$ hiba esetében, a modellt azzal a további tulajdonsággal kiegészítve, hogy a hibátlan folyamatok soha nem fogadják el az (m, i, r) üzenetet az $(r + 1)$-edik menet után.
Adjunk egy ilyen algoritmust, és bizonyítsuk a helyességét, vagy indokoljuk meg, miért nem létezik ilyen algoritmus.

6-29. Írjuk le a PoliBiz algoritmus legrosszabb esetébenek megfelelő végrehajtási sorozatát, melyben létezik olyan hibamentes P_i folyamat, melyre az a legkorábbi menet, amikorra a folyamat elfogadja $2f + 1$ számú, egymástól különböző folyamat üzenetét, pontosan a $2(f + 1)$-edik menet.

6-30. A Zöldfülűek Számítástechnikai Rt. programozója úgy módosította a PoliBiz algoritmus megvalósítását, hogy a $2s - 1$ alakú menetekben az elfogadás küszöb értéke $f + s - 1$ helyett $s - 1$, és a döntési küszöb értéke $2f + 1$ helyett $f + 1$. Helyes ez a módosítás? Bizonyítsuk, vagy mutassunk ellenpéldát.

6-31. Tervezzünk algoritmust a bizánci megegyezésre, melynek kommunikációs bonyolultsága polinomiális, bemeneti értékeinek halmaza általános, úgy, hogy nem használjuk szubrutinként a bináris bizánci megegyezést. Az algoritmus használhatja a következetes üzenetszórás működési módszert, de egy hatékonyabb megvalósítást kellene tervezni, mint a KövetkezetesÜzenSzórás algoritmus.

6-32. Tervezzünk egy algoritmust a megegyezésre megállási hibák esetében, mely kielégíti az alábbi *korai megállás* tulajdonságot: ha az algoritmus egy menetében csak $f' < f$ számú folyamat hibázik, akkor az az időpont, amelyre az összes hibamentes folyamat döntést hoz legfeljebb kf', ahol k egy konstans. Készítsünk hasonlót a bizánci megegyezésre.

6-33. Tervezzünk protokollt négy folyamatra, melyek egy teljesen összefüggő gráfban helyezkednek el, úgy, hogy *akár* egy bizánci hibát, *akár* három megállási hibát eltűrjön. Próbáljuk meg minimalizálni a menetek számát.

6-34. *Kutatási feladat:* találjunk ki a bizánci megegyezés megoldására *egyszerű* $f + 1$ menetes protokollt, melyhez csak $3f + 1$ folyamat szükséges, és kommunikációs bonyolultsága polinomiális.

6-35. Ennek a gyakorlatnak az a célja, hogy megvizsgáljuk a 6.26. lemma bizonyításában található konstrukciót, mely két háromszög alakú rendszert illeszt egybe úgy, hogy egy hatszögű rendszer az eredmény.

(a) Nagyon körültekintően írjuk le azt az A algoritmust egy három-folyamatos teljes gráfra, mely megoldja a hiba nélküli megegyezési problémát, azaz a bizánci megegyezés problémát arra a speciális esetre, ahol egyetlen folyamat sem hibázik.

(b) Most készítsük el az S rendszert, az A algoritmus két példányának összeillesztésével, ahogy a 6.26. lemma bizonyításában tettük. Írjuk le nagyon

alaposan az S azon végrehajtását, melyben P_1, P_2 és P_3 a 0 bemenettel, P'_1, P'_2 és P'_3 az 1 bemenettel kezd.

(c) Megoldja S a hiba nélküli megegyezési problémát (a hatszög hálózatban)? Bizonyítsuk, hogy igen, vagy adjunk egy végrehajtást, mely bizonyítja, hogy nem.

(d) Találhatunk-e olyan három-folyamatos A algoritmust, melynek *tetszőlegesen sok* példányát összeillesztve egy gyűrűbe, az eredményül kapott rendszer minden esetben megoldja a hiba nélküli megegyezési problémát?

6-36. Mennyi lesz a hibás folyamatok számának maximuma, melyet a bizánci megegyezés algoritmusai az alábbi hálózati gráfokban futva még eltűrnek?

(a) Egy n méretű gyűrű.

(b) Egy három dimenziós kocka, egy oldalán m olyan csúccsal, mely csúcsok csak a három dimenzióban elhelyezkedő szomszédaikkal vannak összekötve.

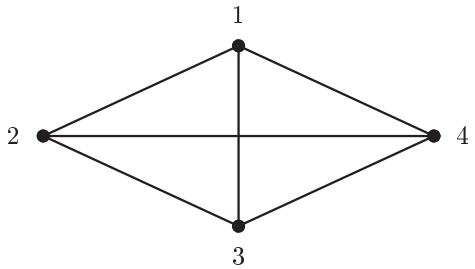
(c) Egy teljes páros gráf, mindkét komponensében m csúccsal.

6-37. Adjunk még alaposabb bizonyítást a bizánci megegyezés megoldhatatlanságára, amikor $n = 2$ és $f = 1$.

6-38. Elemezzük a bizánci megegyezés algoritmusának futási idejét, az üzenetek számát, a kommunikáció bitszámát a 6.29. tétel bizonyításában leírt általános gráfok esetében. Tudnánk-e javítani valamelyiken?

6-39. Mutassuk meg nagyon körültekintően, hogy a 6.29. tétel bizonyításában használt egyszerűsítések, amikor azt bizonyítottuk, hogy a bizánci megegyezés lehetetlen $f = 1$ és $\mathit{össz\_függ}(G) \leq 2$ esetében, megengedhetők. Azaz, lássuk be, hogy egy algoritmus létezése olyan esetre, amelyben az 1-es és 3-as csúcsokat tetszőleges, összefüggő rész-gráfokkal helyettesítjük, maga után vonja, hogy létezik algoritmus arra az esetre, amikor ezek egyszerű csúcspontok.

6-40. Fontoljuk meg újra azt a bizonyítást, hogy a bizánci megegyezés nem érhető el a 6.11. ábrán látható gráfban. Miért lenne hibás ez a bizonyítás, ha kiterjesztenénk a 6.21. ábrán látható gráfra?



6.21.. ábra. A hálózat gráfja a 6-40. gyakorlathoz.

6-41. Bizonyítsuk be, hogy a bizánci megegyezés $f > 1$ számú hiba esetében nem oldható meg abban a G gráfban, melyre $\mathit{össz\_függ}(G) \leq 2f$. A bizonyításhoz

kétfajta módszert is követhetünk, vagy használjuk a folyamatokat csoportosító érvelést, ahogy azt a 6.29. tétel bizonyításának végén vázoltuk, vagy a 6.27. tételhez hasonlóan redukálást használunk.

6-42. Találjunk egyszerű algoritmust a gyenge bizánci megegyezésre abban a hálózati gráfban, mely két csúcspontból és egy a csúcsokat összekötő élből áll.

6-43. Tegyük teljessé a 6.30. tétel bizonyítását, megmutatva a megoldhatatlanságot a következő két esetre:
(a) amikor $n \leq 3f$ és $f > 1$;
(b) amikor $\ddot{o}ssz\_f\ddot{u}gg(G) \leq 2f$.

6-44. Tekintsük a *bizánci kivégző osztag* problémát, melynek definíciója a következő. Egy teljesen összefüggő hálózati gráfban helyezkedik el az n számú folyamat, melyeknek nincs bemeneti értékük, és változó a kezdési idejük. Azaz mindegyik folyamat egy *tétlen* állapotban kezd, mely nem tartalmaz információt, és amelyből csak *null* üzeneteket küld. Nem változik az állapota addig, amíg nem kap egy speciális *ébresztő* üzenetet kívülről, vagy egy nem *null* üzenetet egy másik folyamattól. A folyamat nem tudja, az éppen aktuális menet számát amikor felébred. A modell hasonló ahhoz, amit a 2.1. alfejezetben leírtunk, kivéve, hogy itt nem feltételezzük, hogy mindegyik folyamat kötelezően kap egy *ébresztő* üzenetet – csak a folyamatok egy tetszőleges részhalmaza. Továbbá megengedjük a bizánci hibákat.
A feladat a *tűz* parancsok kiadása a folyamatoknak, az alábbi feltételek mellett.

Megegyezés. Ha egy hibátlan folyamat *tűz* jelet kap valamely menetben, akkor az összes hibátlan folyamat ugyanabban a menetben kap *tűz* jelet, és hibátlan folyamat semelyik másik menetben sem kap *tűz* jelet.

Érvényesség. qinddefférvényesség Ha az összes hibátlan folyamat *ébresztő* üzenetet kap, akkor az összes hibátlan folyamat azonnal *tüzel*; ha hibátlan folyamat nem kap *ébresztő* üzenetet, akkor hibátlan folyamat soha nem fog *tüzelni*.

(a) Tervezzünk algoritmust a bizánci tüzelő osztag feladat megoldására, ha $n > 3f$ igaz.
(b) Bizonyítsuk be, hogy a problémát nem lehet megoldani, ha $n \leq 3f$.

6-45. Mondjuk ki, majd bizonyítsuk a 6.33. tételt az $f = 3$ speciális esetre.

6-46. Fennáll-e továbbra is a 6.37. lemma, ha a folyamatoktól nem kívánjuk meg, hogy szabályosak legyenek. Bizonyítsuk, vagy adjunk ellenpéldát.

6-47. A 6.7. alfejezetben megmutattuk, hogy az f számú hibát eltűrő, megállási hiba melletti megegyezés nem oldható meg f menetben. Egy hosszú láncot hoztunk létre, mely összeköti azt a két menetet, amelyben mindegyik folyamat hibátlan és ugyanaz a bemenetük. Csak a lánc elkészítésének módját adtuk meg.

(a) Milyen hosszú a futamok lánca?
(b) Mennyivel lehetne rövidíteni a láncot, a megállási hibák helyett bizánci hibákat használva?

6-48. *Kutatási feladat.* Adjunk alsó és felső korlátot a megállási feladat megoldásának és/vagy a bizánci feladat megoldásának futási idejére általános (nem szükségszerűen teljes) hálózati gráfokban.

7. fejezet

További megegyezési problémák

Az előző két fejezetben a megegyezési problémával foglalkoztunk: az ötödik fejezetben az összehangolt támadással, a hatodik fejezetben a megegyezéssel. Ebben a fejezetben három további megegyezési feladat elemzésével befejezzük a szinkron osztott megegyezés tárgyalását. Ezek a *k-megegyezés problémája*, a *közelítő megegyezés problémája* és az *osztott adatbázisok véglegesítési problémája*. A hatodik fejezethez hasonlóan csak a folyamatok hibáit vesszük figyelembe.

7.1.. k-megegyezés

Először a k-megegyezés problémáját vizsgáljuk, ahol k egy nemnegatív egész szám. Ez a 6. fejezetben elemzett egyszerű megegyezési feladat természetes általánosítása. Most azonban nem azt várjuk a folyamatoktól, hogy pontosan ugyanazt a meghatározott értéket válasszák, hanem megelégszünk azzal, hogy választásukat egy kis (k-elemű) halmaz elemeire korlátozzák.

Az eredeti motiváció matematikai jellegű volt: érdekesnek látszott annak vizsgálata, hogyan változnak az eredmények, ha ilyen egyszerű módon megváltoztatjuk a feltételeket. Gyakorlati helyzetek is elképzelhetők azonban, melyekben ilyen algoritmus hasznos lehet. Például tekintsük olyan közösen is használható erőforrások hozzárendelését, mint amilyenek egy kommunikációs hálózat üzenetszórási frekvenciái. Kívánatos lehet bizonyos számú folyamat olyan megegyezése, hogy nagy mennyiségű adat (például egy videoszalag) szórására adott, kisszámú frekvenciát használnak. Mivel a kommunikáció üzenetszórással valósul meg, tetszőleges számú folyamat megkaphatja az adatokat – ugyanazokat a frekvenciákat használva. A teljes kommunikációs terhelés minimalizálása érdekében célszerű a frekvenciák k számát alacsonyan tartani.

Ebben a fejezetben pontos alsó és felső korlátokat adunk meg a k-megegyezési feladat teljes hálózati gráfban való megoldásához szükséges menetek számára, csak megállási hibákat feltételezve. Ezeket a korlátokat 3 paraméter függvényében adjuk meg: a folyamatok n száma, a megengedett hibák f száma és a megengedett döntési értékek k száma.

7.1.1.. A feladat

A k-megegyezési problémánál az egyszerű megegyezési problémához hasonlóan feltesszük, hogy a hálózat egy n-csúcsú összefüggő, irányítatlan gráf, a hálózatban n folyamat $(P_1, P_2, \ldots, P_n)$ van és a folyamatok ismerik az egész gráfot. Minden folyamat egy rögzített V értékhalmazból vett bemenettel indul és végül ugyanebből az értékhalmazból választ egy kimenetet. (Most is feltesszük, hogy minden folyamatnak egy kezdőállapota van, amely minden bemeneti értéket tartalmaz.) Feltesszük, hogy legfeljebb f folyamat lehet hibás. Csak megállási hibákat engedünk meg. A szükséges feltételek a következők.

Megegyezés. A V értékhalmaznak van olyan k elemszámú W részhalmaza, hogy minden döntési érték W eleme.

Érvényesség. Minden folyamat döntési értéke valamely folyamat kezdeti értéke.

Befejeződés. Ha egy folyamat nem hibás, akkor végül választ egy értéket.

A megegyezési feltétel a közönséges megegyezési problémánál használt megegyezési feltétel természetes általánosítása. Megjegyezzük, hogy megállási hibákra inkább a 6.1. alfejezet végefelé adott érvényességi feltételt használjuk, mint a hatodik fejezet nagyobb részében használt gyengébb feltételt; erre az erősebb feltételre a 7.1.3. szakaszban, az alsó korlát bizonyításakor lesz szükségünk. A közönséges megegyezési feladat az erősebb érvényességi feltétellel pontosan a k-megegyezési feladat $k = 1$ esete.

Ebben a fejezetben csak teljes gráfokra vontakozó eredményeket tárgyalunk. Feltesszük, hogy a V halmazon teljes rendezés van értelmezve.

A 6.2.1. szakaszhoz hasonlóan azt mondjuk, hogy egy folyamat *aktív* r $(0 \leq r)$ menet után, ha nem hibásodik meg az r-edik menet végéig.

7.1.2.. Egy algoritmus

Most egy nagyon egyszerű algoritmust mutatunk be, melynek neve MinTerjed. Valójában ez pontosan a 6-9. gyakorlatban vázolt algoritmus, de kevesebb menetre van szüksége. A 6-9. gyakorlatban azt állítottuk, hogy ha ez az algoritmus $f + 1$ menetet fut, akkor garatálja a közönséges megállási megegyezést. Belátjuk, hogy a k-megegyezést akkor is biztosítani tudja, ha csak $\lfloor \frac{f}{k} \rfloor + 1$ menetet fut. Egyszerűen fogalmazva ez azt jelenti, hogy ha egy helyett k döntési értéket engedünk meg, akkor a futási idő a k-ad részére csökken.

MinTerjed algoritmus (vázlatosan)

Minden folyamat kezel egy *min_érték* változót, melybe induláskor a saját kezdeti értéket teszi. A folyamatok az $\lfloor \frac{f}{k} \rfloor + 1$ menet mindegyikében szétszórják *min_érték*-eiket, azután minden folyamat a helyettesíti saját *min_érték*-ét a beérkezett *min_érték*-ek és a saját régi *min_érték*-ének minimumával. A döntési érték a végső *min_érték*.

A kód a következő. (Szerkezetét hasonlítsuk össze a 6.2.1. szakaszban szereplő HALMAZTERJED szerkezetével.)

MINTERJED algoritmus (formálisan)
Az üzeneti ábécé V.

állapotok$_i$
menetek $\in \mathbb{N}$, kezdetben 0
döntés $\in V \cup \{$*ismeretlen*$\}$, kezdetben *ismeretlen*
min_érték $\in V$, kezdetben P_i kezdeti értékét veszi fel

üzenetek$_i$
if *menetek* $\leq \lfloor \frac{f}{k} \rfloor$ **then** küldjük el *min_érték*-et minden más folyamatnak

átmenetek$_i$
menetek := *menetek* + 1
legyen m_j a P_j folyamattól kapott üzenet minden olyan folyamatra,
amelytől érkezett üzenet
min_érték := $\min(\{$*min_érték* $\} \cup \{m_j : j \neq i\})$
if *menetek* $= \lfloor \frac{f}{k} + 1 \rfloor$ **then** *döntés* := *min_érték*

Belátjuk az algoritmus helyességét; a bizonyítás hasonló ahhoz, ahogyan a 6.2.1. szakaszban a HALMAZTERJED algoritmus helyességét beláttuk. Legyen $M(r)$ az aktív folyamatok *min_érték*-einek halmaza r menet után. Először azt látjuk be, hogy az egymást követő időpontokban az $M(r)$ halmaz csak csökkenhet.

7.1. lemma. . *Ha* $1 \leq r \leq \lfloor \frac{f}{k} + 1 \rfloor$, *akkor* $M(r) \subseteq M(r-1)$.

Bizonyítás. Tegyük fel, hogy $m \in M(r)$. Akkor m egy olyan P_i folyamat *min_érték*$_i$ értéke, amely r menet után aktív. Ekkor vagy már az r-edik menet előtt is $m =$ *min_érték*$_i$ volt, vagy m éppen az r-edik menetben érkezik P_i-hez, mondjuk P_j-től. De ebben az esetben $r-1$ menet után *min_érték*$_j = m$ és P_j aktív $r-1$ menet után, mivel üzenetet küld az r-edik menetben. Ebből következik, hogy $m \in M(r-1)$. □

7.2. lemma. . *Legyen* $d \in \mathbb{N}^+$. *Ha az* r*-edik* $(1 \leq r \leq \lfloor \frac{f}{k} \rfloor + 1)$ *menetben legfeljebb* $d-1$ *folyamat hibázik, akkor* $|M(r)| \leq d$, *azaz az* r*-edik menet után aktív az folyamatoknak legalább* d *különböző min_érték-e van.*

Bizonyítás. Ellentmondás elérése érdekében tegyük fel, hogy r-edik menet alatt legfeljebb $d-1$ folyamat hibázott, és $M(r) > d$. Legyen $M(r)$ legnagyobb eleme m, és legyen $m' \neq m$ az $M(r)$ halmaz tetszőleges másik eleme. Ekkor a 7.1. lemma szerint $m' \in M(r-1)$. Legyen P_i tetszőleges aktív folyamat, melyre az $(r-1)$-edik menet után *min_érték*$_i = m'$. Ha P_i nem hibázik az r-edik menetben, akkor az r-edik menetben minden folyamat kap egy m'-t tartalmazó üzenetet P_i-től. Ez azonban nem fordulhat elő, mivel r menet után az egyik aktív folyamat *min_érték*-e m-mel egyenlő, és $m > m'$. Ebből következik, hogy P_i hibázik az r-edik menetben.

Az m' értéket azonban tetszőlegesen választottuk úgy, hogy az $M(r)$ halmaznak a legnagyobb elemtől különböző eleme legyen. Ezért $M(r)$ minden $m' \neq m$ eleméhez van olyan aktív folyamat, melynek *min_érték*-e $r-1$ menet után $m' \neq m$, és hibázik az r-edik menetben. Feltételünk szerint az r-edik menetben legfeljebb $d-1$ folyamat hibázik, ezért $M(r)$-nek legfeljebb $d-1$ olyan eleme lehet, amely m-től különbözik. Ezért $|M(r)| \leq d$, ami ellentmondás. □

Most bebizonyíthatjuk a fő helyességi tételt.

7.3. tétel. . *A* MinTerjed *algoritmus a megállási hiba modellre megoldja a k-megegyezési problémát.*

Bizonyítás. A Befejeződés és az érvényesség közvetlenül adódik. Most bebizonyítjuk az új megegyezési feltétel teljesülését. Annak érdekében, hogy ellentmondásra jussunk, tegyük fel, hogy a különböző döntési értékek száma k-nál nagyobb egy olyan végrehajtási sorozatban, amelyben legfeljebb f hiba van. Akkor az aktív folyamatok *min_érték*-einek száma $\lfloor \frac{f}{k} \rfloor + 1$ menet után legalább $k+1$, azaz $|M(\lfloor \frac{f}{k} \rfloor + 1)| \geq k+1$. A 7.1. lemma szerint $|M(r)| \geq k+1$ minden r-re $(0 \leq r \leq \lfloor \frac{f}{k} \rfloor + 1)$. Ekkor a 7.2. lemmából következik, hogy az r-edik $(1 \leq r \leq \lfloor \frac{f}{k} \rfloor + 1)$ menetben legalább k folyamat hibázik. Ez azt eredményezi, hogy a hibák összes száma legalább $(\lfloor \frac{f}{k} \rfloor + 1)k$. Ez azonban határozottan nagyobb, mint f, ami ellentmondás. □

Bonyolultságelemzés.. A menetek száma $\lfloor \frac{f}{k} \rfloor + 1$. Az üzenetek száma legfeljebb $(\lfloor \frac{f}{k} \rfloor + 1)n^2$, és az üzenetekben lévő bitek száma legfeljebb $\left(\lfloor \frac{f}{k} \rfloor + 1\right) n^2 b$, ahol b egy felső korlát a V halmaz egy elemének ábrázolásához szükséges bitek számára.

7.1.3.. Alsó korlát*

Ebben a szakaszban megmutatjuk, hogy az $\lfloor \frac{f}{k} \rfloor + 1$ korlát éles: ugyanis belátjuk, hogy a $|V| \geq k+1$ feltétel mellett egyúttal alsó korlát is. Ez az eredmény pontosan megadja azt a gyorsítást, melyet azzal érhetünk el, hogy egyetlen kimeneti érték helyett k értéket engedünk meg: az idő k-ad részére csökken. Amint az várható, a bizonyítás alapgondolatai a 6.33. tételben a közönséges megegyezésre vonatkozó alsó korlát bizonyításából származnak, de a bizonyítás maga bonyolultabb és érdekesebb. Ezek az ötletek elvezetnek bennünket az algebrai topológia birodalmába.

A fejezet hátralévő részében A olyan algoritmus, amely n folyamatra megoldja a k-megegyezési problémát, ha legfeljebb f folyamat követ el megállási hibát. Feltesszük, hogy az A algoritmus $r < \lfloor \frac{f}{k} \rfloor + 1$ menetben megáll; ekkor $r \leq \lfloor \frac{f}{k} \rfloor$. Ellentmondás elérése érdekében még azt is feltesszük, hogy $n \geq f+k+1$, ami azt jelenti, hogy legalább $k+1$ folyamat soha nem hibázik.

Az általánosság megszorítása nélkül feltehetjük, hogy a folyamatok pontosan az r-edik menet végén döntenek, és azonnal megállnak. Azt is feltesszük, hogy minden folyamat minden másik folyamatnak üzenetet küld a k-adik $(1 \leq k \leq r)$ menetben (amíg meg nem hibásodik). Végül feltesszük, hogy a V értékhalmaz

pontosan $k+1$ elemet $(0,\ 1,\ \ldots, k)$ tartalmaz, mivel mindezekre szükségünk van az ellentmondás eléréséhez.

Az ellentmondást úgy kapjuk meg, hogy megmutatjuk, A végrehajtási sorozatainak egyikében, melyben legfeljebb f folyamat hibásodhat meg, van $k+1$ olyan folyamat, amelyek $k+1$ különböző értéket választanak és ezzel megsértik a k-megegyezést.

Áttekintés.. A 6.33. tétel bizonyítása szerint $f+1$ menet alsó korlát a közönséges megállási megegyezésre. A bizonyítás *láncérvelést* alkalmaz, amelyben végrehajtási sorozatok olyan láncát állítjuk elő, amely a csak 0 döntést megengedő végrehajtási sorozattal kezdődik, és a csak 1 döntést megengedő végrehajtási sorozattal végződik. Szeretnénk ezt a bizonyítást k más értékeire is kiterjeszteni. Sajnos, a k-megegyezési problémában – a közönséges megegyezési problémától eltérően – az adott végrehajtási sorozatban előforduló döntések nem határozzák meg a szorosan kapcsolódó végrehajtási sorozatokban szereplő döntéseket. Például, ha a közönséges megyezési feladat megoldási algoritmus α és α' végrehajtási sorozatai megkülönböztethetetlenek a hibátlan P_i folyamaz számára, akkor nemcsak P_i-nek kell α-ban és α'-ben ugyanazt a döntést hoznia, hanem az összes hibátlan folyamatnak is ugyanezt a döntést kell hoznia. A k-megegyezési algoritmusra az igaz, hogy ha α és α' megkülönböztethetetlenek P_i számára, akkor P_i döntései a két végrehajtási sorozatban azonosak, de a többi folyamat döntése meghatározatlan. Ha α és α' $n-1$ folyamat számára megkülönböztethetetlenek, a fennmaradó folyamat döntése még akkor is meghatározatlan.

Kulcs gondolatunk az, hogy inkább egy k-dimenziós végrehajtási sorozathalmazt készítünk, mint egy egydimenziós láncot. Ennek a halmaznak a szomszédos végrehajtási sorozatai a kijelölt folyamatok számára megkülönböztethetetlenek. A végrehajtási sorozatok kezelésére felhasznált k-dimenziós alakzatot *Bermuda-háromszögnek* nevezzük (mivel bármely feltételezett k-megegyezési algoritmus eltűnik valahol a belsejében).

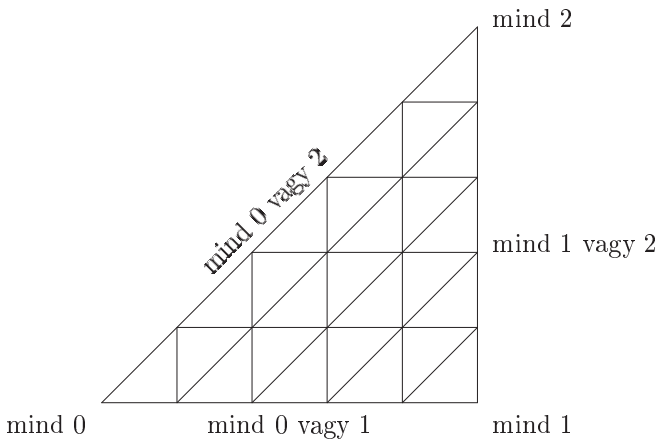
7.1.1. példa. Bermuda-háromszög

A 7.1. ábra egy Bermuda-háromszöget mutat a $k=2$ esetben. Az ábra egy nagy háromszöget mutat, amelyet kisebb háromszögekre bontottunk.

Ha $k>2$, akkor a háromszög k-dimenziós változatára van szükségünk. Szerencsére ilyen általánosítás már ismert az algebrai topológiában: k-dimenziós *szimplex* a neve. Például egy egydimenziós szimplex egy él, egy kétdimenziós szimplex egy háromszög, és egy háromdimenziós szimplex egy tetraéder. (Háromnál több dimenzió esetében már sokkal nehezebb a szimplexeket elképzelni.)

Most tetszőleges k-ra induljunk ki a k-dimenziós euklideszi térben adott k-dimenziós szimplexből. A szimplex *rácspontokat* tartalmaz, amelyek az euklideszi tér egész koordinátájú pontjai. A *k-dimenziós B Bermuda-háromszöget* úgy kapjuk, hogy ezt a szimplexet a rácspontok segítségével felbontjuk, és így kis k-dimenziós kis szimplexek halmazát kapjuk.

A bizonyítás során először B minden csúcsához (rácspontjához) hozzárendelünk egy végrehajtási sorozatot. A B Bermuda-háromszög $k+1$ sarokcsúcsához



7.1.. ábra. Bermuda-háromszög a $k = 2$ esetben.

olyan végrehajtási sorozatokat rendelünk, amelyekben a folyamatok a $\{0, \ldots, k\}$ halmaz ugyanazon elemét kapják bemenetként, és amelyekben nincsenek hibák. Például a $k = 2$ esetben a bal alsó sarokcsúcshoz rendelt végrehajtási sorozatban minden folyamat bemenete 0, a jobb alsó csúcshoz rendelt végrehajtási sorozatban minden bemenet 1, és a jobb felső csúcshoz rendeltben minden bemenet 2 (lásd a 7.1. ábrát). Továbbá, B tetszőleges oldalán (tetszőleges dimenziószám esetében) lévő bármely x csúcshoz rendelt végrehajtási sorozatban csak azok a bemenetek fordulhatnak elő, amelyek előfordulnak az oldallap csúcsaihoz rendelt végrehajtási sorozatban. Például a $k = 2$ esetben az alsó élen lévő csúcsokhoz rendelt végrehajtási sorozatokban lévő bemenetek mindegyike eleme a $\{0, 1\}$ halmaznak.

Ezután B minden csúcsához hozzárendeljük egy olyan folyamat indexét, amely az adott csúcshoz rendelt végrehajtási sorozatban nem hibás. Ezt a hozzárendelést úgy végezzük, hogy minden T kis szimplexhez egyértelműen legyen egy olyan α végrehajtási sorozat, amelyben legfeljebb f hiba van, és amely az alábbi értelemben kompatibilis T sarokpontjaival.

1. A T sarokcsúcsaihoz rendelt folyamatok α-ban nem hibásak.
2. Ha T valamelyik sarokcsúcsához az α' végrehajtási sorozatot és a P_i folyamatot rendeltük, akkor α és α' megkülönböztethetetlenek P_i számára.

A végrehajtási sorozatoknak és folyamatoknak ez a hozzárendelése B csúcsaihoz rendelkezik néhány szép tulajdonsággal. Tegyük fel, hogy α-t és P_i-t hozzárendeljük az x csúcshoz. Ha x a B-nek egy sarokcsúcsa, akkor α-ban minden folyamat ugyanazzal a bemenettel kezd, ezért az érvényességi feltétel szerint P_i-nek α-ban ezt az értéket kell választania. Ha x a B-nek egy külső élén van, akkor minden folyamat az adott él két végpontjához tartozó két érték valamelyikével kezd; az érvényességi feltételből következik, hogy P_i-nek ezen két érték valamelyikét kell választania. Általánosan fogalmazva, ha x a B-nek egy (tetszőleges dimenziójú) oldalán helyezkedik el, akkor α-ban minden folyamat az adott oldal

valamelyik sarokpontjához tartozó értékkel kezd; ezért P_i az érvényességi feltétel szerint ezen értékek közül választ. Végül, ha x B-nek belső pontja, akkor P_i a $k+1$ érték bármelyikét választhatja.

A végrehajtási sorozatoknak és folyamatoknak a most leírt módon történő csúcsokhoz rendeléséhez szükség van arra, hogy a menetek r száma minden végrehajtási sorozatban legfeljebb $\lfloor \frac{f}{k} \rfloor$ legyen, azaz teljesüljön, hogy $f \geq rk$. Ez azért van így, mert a végrehajtási sorozatokat a 6.33. tétel bizonyításában használt láncérvelés k-dimenziós általánosításával rendeljük a csúcsokhoz.

Miután a végrehajtási sorozatokat és indexeket hozzárendeltük a csúcsokhoz, a csúcsokat a $\{0, 1, \ldots, k\}$ halmazból vett „színekkel" kiszínezzük. Nevezetesen, ha az x csúcshoz az α végrehajtási sorozatot és a P_i folyamatot rendeltük, a csúcsot azzal az értékkel színezzük, amelyet P_i választ α-ban. Ennek a színezésnek a következő tulajdonságai vannak.

1. A B háromszög $k+1$ sarokcsúcsának különbözők a színei.
2. A B háromszög külső élén fekvő pontok színe megegyezik az él valamelyik végpontjának színével.
3. Általánosabban, a B bármely (tetszőleges dimenziójú) oldalán fekvő csúcsainak színe megegyezik az oldal valamelyik sarokcsúcsának színével.

A k-szimplexeknek pontosan ezekkel a tulajdonságokkal rendelkező színezését az algebrai topológiában *Sperner-színezésnek* hívják.

Itt alkalmazni fogunk egy nevezetes kombinatorikai eredményt, melyet először 1928-ban bizonyítottak be. A Sperner-lemma szerint egy kis szimplexekre bontott k-dimenziós szimplex Sperner-színezésének tartalmaznia kell legalább egy olyan kis szimplexet, melynek $k+1$ csúcsa $k+1$ különböző színre van színezve. Esetünkben ez a szimplex megfelel egy egy olyan végrehajtási sorozatnak, amelyben legfeljebb f hiba van, és amelyben $k+1$ folyamat $k+1$ különböző értéket választ. Ez azonban ellentmond a k-megegyezési feladat megegyezési feltételének.

Ebből következik, hogy a remélt algoritmus nem létezik, azaz a k-megegyezési feladat megoldására nincs olyan algoritmus, amely f hiba esetében $r \leq \lfloor \frac{f}{k} \rfloor$ menet után megállna. Az alfejezet hátralévő része további részleteket tartalmaz.

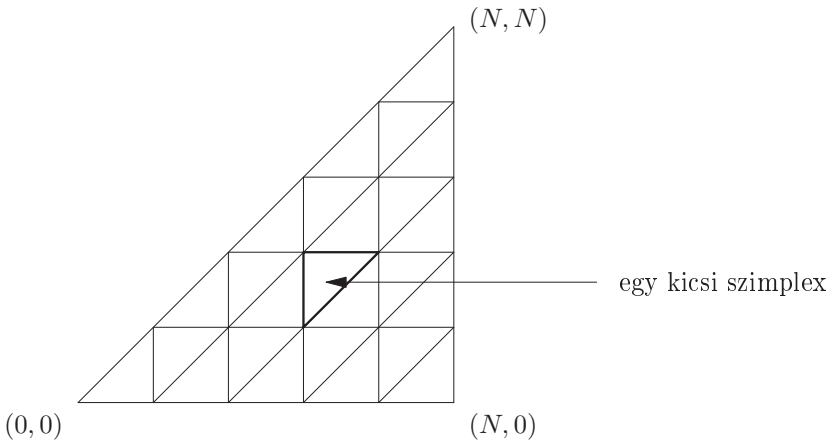
Definíciók.. Felhasználjuk a kommunikációs mintának a 6.7. alfejezetben megadott meghatározását. Újra definiáljuk a *jó* kommunikációs mintát: olyan kommunikációs minta, amelyben $k \leq r$ minden (i, j, k) hármasra, és a hiányzó hármasok konzisztensek a megállási hiba modellel. (Azaz a jó kommunikációs mintának a 6.7. alfejezetben megadott definíciójából az első két feltételt használjuk – csak most a menetek számának felső korlátja f helyett r. Egyelőre nem korlátozzuk a hibák számát.) A jó kommunikációs minta új definícióját használva a *futam* és adott ρ-ra a *végre*(ρ) definíciók ugyanazok, mint a 6.7. alfejezetben. Azt mondjuk, hogy a P_i folyamat egy futam t-edik menete után *csendes*, ha a $(t+1)$-edik vagy nagyobb sorszámú menetekben nem küld üzenetet.

Bermuda-háromszög.. A k-dimenziós euklideszi térben értelmezett k-szimplexszel kezdjük, melynek sarokcsúcsai a k hosszúságú $(0, \ldots, 0)$, $(N, 0, \ldots, 0)$, $(N, N, 0, \ldots, 0)$, …, $(N, \ldots, \mathbb{N})$ vektorok, ahol N egy nagyon nagy egész szám. A B Bermuda-háromszög ez a szimplex a kis szimplexekre való alábbi felbontásával együtt. B csúcsai a szimplexekben lévő rácspontok, azaz az

$x = (x_1, \ldots, x_k)$ alakú csúcsok, ahol a vektorok koordinátái 0 és N közötti egész számok, melyekre $x_1 \geq x_2 \geq \ldots \geq x_k$. A kis szimplexeket a következőképpen definiáljuk: válasszunk ki egy tetszőleges rácspontot, és minden dimenzióban minden pozitív irányban tegyünk meg egy lépést, tetszőleges sorrendben. A bejárás során érintett $k+1$ csúcs megadja egy kis szimplex csúcsait.

7.1.2. példa. A Bermuda-háromszög csúcsainak koordinátái

A 7.2. ábra egy kétdimenziós Bermuda-háromszöget ábrázol.



7.2.. ábra. Kétdimenziós Bermuda-háromszög.

***B* címkézése végrehajtási sorozatokkal és futamokkal..** Ebben a részben leírjuk, hogyan rendelünk végrehajtási sorozatokat B csúcsaihoz (azaz a csúcsokat „megcímkézzük" a végrehajtási sorozatokkal. Ezt úgy végezzük el, hogy először a futamokban lévő bizonyos (i, t) (folyamat, menetszám) párokhoz *jeleket* rendelünk. Ezek a jelek úgy értelmezhetőek, hogy megengedik a P_i folyamatnak, hogy a t-edik menetben vagy azután hibázzon. Ugyanahhoz az (i, t) párhoz több jel is rendelhető.

Pontosabban, az *l-futamot* minden $l > 0$ értékre úgy definiáljuk, hogy az minden t $(1 \leq t \leq r)$ értékre pontosan l jellel úgy van kiegészítve, hogy ha a P_i folyamat a t-edik menetben hibázik, akkor van olyan jel, amely egy (i, t') párhoz $(t') \leq t)$ van hozzárendelve. Ezek szerint egy l-futam pontosan lr jelet tartalmaz. Csak az $l = 1$ és az $l = r$ eset, azaz az 1-futamok és az r-futamok érdekelnek bennünket. A *hibamentes l-futamot* úgy definiáljuk, hogy az olyan l-futam, amelyben nincs hiba, és amelyben minden jel $(1, t)$ alakú párhoz van rendelve (azaz csak a P_1 folyamat hibázhat).

Mivel minden kiegészített futamot futamból állítottunk elő, minden kiegészített futam nyilvánvaló módon alakítható át végrehajtási sorozattá. A *végre*(ρ)

jelölést, amelyet korábban futamokra vezettünk be, kiterjesztjük arra az esetre, amikor ρ kiterjesztett futam. B csúcsainak végrehajtási sorozatokkal való címkézése most k-futamokkal történő címkézést jelent.

Négy műveletet definiálunk l-futamokra, melyek mindegyike csak kis változásokat okoz. Ezek a műveletek csak egyetlen hármast tudnak eltávolítani vagy hozzáadni, vagy egyetlen folyamat bemeneti értékét tudják módosítani, vagy szomszédos indexű folyamatok között tudnak egy jelet mozgatni ugyanabban a menetben. Ezek a műveletek nagyon hasonlítanak a 6.33. tétel bizonyításában alkalmazottakhoz. Ezeket a műveleteket a következőképpen definiáljuk.

1. `eltávolít`(i,j,t), ahol i és j folyamatindexek és t menetszám, $1 \leq t \leq r$. Ez a művelet eltávolítja az (i,j,t) hármast (ami azt jelenti, hogy P_i a t-edik menetben P_j-nek üzenetet küldött), ha van ilyen hármas, egyébként nincs hatása. Csak akkor alkalmazható, ha t menet után P_i és P_j is csendes, és van olyan (i,t') $(t' \leq t)$ pár, amelyhez jel van kapcsolva.
2. `hozzáad`(i,j,t). Ez a művelet hozzáadja az (i,j,t) hármast, ha még nincs hozzáadva a futamhoz, egyébként nincs hatása. Csak akkor alkalmazható, ha P_i és P_j mindketten csendesek t menet után, és P_i aktív $t-1$ menet után.
3. `változtat`(i,v). Ez a művelet P_i bemeneti értékét v-re változtatja, és nincs hatása, ha a bemeneti érték már v. Csak akkor alkalmazható, ha P_i csendes 0 menet után és az $(i,1)$ pár rendelkezik jellel.
4. `mozgat`(i,j,t). Ez a művelet a (i,t) pártól a (j,t) párhoz mozgat egy jelet, ahol j vagy $i+1$ vagy $i-1$. Csak akkor alkalmazható, ha (i,t)-nél van jel és minden hibának engedélye van más jelektől.

A definíciókból következik, hogy ha ezeket a műveleteket l-futamra alkalmazzuk, akkor az eredmény is l-futam lesz.

Definiáljuk minden $v \in \{1,\ldots,k\}$ értékre az `eltávolít, hozzáad, változtat` és `mozgat` műveletek *sorozat*(v) sorozatát, amely bármely ρ hibamentes 1-futamra alkalmazható, és azt olyan hibamentes 1-futammá alakítja, melyben minden folyamat bemenete v. Valójában minden ρ hibamentes 1-futamra ugyanaz a *sorozat*(v) alkalmazható. Az átalakítás a 6.33. tétel bizonyításában alkalmazott módszerekkel végezhető el; a fő különbség a hibákra engedélyt adó jelek tényleges mozgása. Ebben a konstrukcióban a folyamatok bemenetei egyenként változnak v-re, a P_1 folyamattal kezdve. A korábbiakhoz hasonlóan ez a konstrukció r menetben r hibát használ fel.

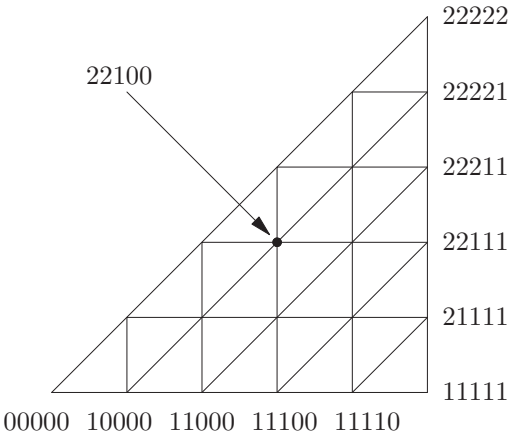
Kiderül, hogy a *sorozat*(v) sorozatok megkonstruálhatók úgy, hogy különböző v-re izomorfak – azaz v választásától eltekintve azonosak. Végül definiálhatjuk a B méretének meghatározásánál használt N paramétert: N egyszerűen a *sorozat*(v) sorozat hossza (tetszőleges v-re).

B csúcsainak címkézésére több *sorozat*(v) sorozatot fogunk használni. Emlékeztetünk arra, hogy az értékkészlet elemei $0,1,\ldots,k$. Minden $v \in \{0,1,\ldots,k\}$ értékre legyen τ_v az a hibamentes 1-futam, amelyben minden folyamat kezdeti értéke egyenlő v-vel. Minden *sorozat*(v) $(1 \leq v \leq k)$ sorozatot arra használunk, hogy a τ_{v-1} hibamentes 1-futamhoz 1-futamok olyan sorozatát állítsuk elő, amelyeket a B-nek a v-edik dimenziós élei (a v-tengelyek) mentén lévő csúcsok *előzetes címkézésére* használhatunk. Ekkor a B x csúcsához rendelendő k-futamot annak

a k darab 1-futamnak az „összefésülésével" kapjuk, amelyek x k-tengelyekre való vetületének előzetes címkéi.

7.1.3. példa. A Bermuda-háromszög címkézése k-futamokkal

Ennek az összefésülésnek a megértését segíti a 7.3. ábra, amelyen a $k = 2$ (és így $V = \{0,\ 1,\ 2\}$) és $n = 5$.



7.3.. ábra. Bermuda-háromszög címkézése k-futamnál.

A diagram csak azokat a csúcsokat tartalmazza, amelyeket hibamentes k-futamokkal címkéztünk. Ehhez csak az ábrázolt csúcsokhoz tartozó bemeneti értékek vektorára van szükségünk. A B háromszög csúcsait címkéző futam minden sarokcsúcsban olyan hibamentes k-futam, melynek elemei azonosak és a bal alsó sarokcsúcsnál nullák, a jobb alsó sarokcsúcsnál egyesek és a felső csúcsnál kettesek. A vízszintes tengelyen lévő láncot *sorozat*(1) segítségével állítjuk elő, és a csupa nullából álló vektorból a csupa egyesből álló vektorhoz vezet, míg a függőleges tengelyen lévő láncot a *sorozat*(2) segítségével állítottuk elő és ez a lánc a csupa egyesből álló vektortól a csupa kettesből álló vektorhoz vezet.

Figyeljük meg a B-ben megjelenő bemenetek mintáját. A vízszintes tengelyen a folyamatok bemenetei egyenként 0-ról 1-re változnak, a P_1 folyamattal kezdve. A függőleges tengely mentén a folyamatok bemenetei egyenként 1-ről kettőre változnak, a P_1 folyamattól kezdve. B belsejében a változások mindkét irányban egyidejűleg mennek végbe. például tekintsük a nyillal megjelölt 22100 csúcsot. A vízszintes és a függőleges tengelyen lévő vetületeit címkéző vektorok 11100 és 22111. A 22100 vektor úgy változtatható 22111-re, hogy az utolsó két folyamat bemenetét 0-ról 1-re változtatjuk, miközben B-ben vízszintesen mozgunk. Hasonlóképpen a 11100 vektor úgy változtatható 22100-ra,

hogy az első két folyamat bemenetét 1-ről 2-re változtatjuk, miközben B-ben függőlegesen mozgunk. A B csúcsait címkéző vektor minden esetben a $\{0, 1, 2\}$ halmaz elemeit tartalmazza, nem növekvő sorrendben.

Most megadjuk az összefésülés formális definícióját. Az 1-futamok $\sigma_1, \ldots, \sigma_k$ 1-futamokból álló sorozatának *összefésültje* a következőképpen definiált ρ k-futam.

1. A P_i folyamat bemeneti értéke ρ-ban v, ahol v az $\{1, \ldots, k\}$ értékek közül a legnagyobb olyan érték, amelyre P_i bemeneti értéke σ_v-ben v, vagy pedig nulla, ha ilyen érték nem létezik.
2. Az (i, j, k) hármas pontosan akkor van benne ρ-ban, ha minden σ_v-ben $(1 \leq v \leq k)$ benne van.
3. Az (i, t) párhoz ρ-ban rendelt jelek száma az összes σ_v-ben lévő (i, t) párhoz rendelt jelek számának összege.

Az első feltétellel kapcsolatban vizsgáljuk meg újra azt a módot, ahogyan az összefésülési műveletet alkalmazzuk. Minden σ_v-t úgy kapunk meg, hogy *sorozat*(v) bizonyos előtagját alkalmazzuk τ_{v-1}-re. Ennek a sorozatnak bizonyos pontján a P_i folyamat bemeneti értéke $v-1$-ről v-re változik. Ha ez már megtörtént σ_v-ben, akkor mondjuk azt, hogy a P_i folyamat „átment" a v-edik dimenzióba. Az első feltétel éppen a legnagyobb v-t választja (ha van ilyen), amelyre igaz, hogy P_i átment a v-edik dimenzióba.

A második feltétel azt mondja, hogy egy üzenet akkor és csak akkor hiányzik a ρ új futamból, ha hiányzik bármely összefésülendő σ_v futamból. A harmadik feltétel összeszámolja a jeleket. Nem nehéz belátni, hogy 1-futamok sorozatának összefésültje valójában k-futam.

Rakjuk össze az elemeket és definiáljuk B k-futamokkal való címkézését. Legyen $x_1, \ldots, x_k$ B tetszőleges csúcsa. Minden $v \in \{1, \ldots k\}$ értékre legyen σ_v az az 1-futam, amelyet úgy kapunk, hogy *sorozat*(v) első x_v műveletét alkalmazzuk τ_{v-1}-re. Akkor az x csúcsot címkéző k-futamot a $\sigma_1, \ldots, \sigma_k$ összefésülésével kapjuk. Megjegyezzük, hogy az összefésült futamban legfeljebb $rk \leq f$ jel van, és ezért legfeljebb f hiba. A bizonyítás hátralévő részére rögzítjük B k-futamokkal (és végrehajtási sorozatokkal) való címkézését.

Ezt a szakaszt azzal fejezzük be, hogy megmutatjuk azt a szoros kapcsolatot, amely B bármely T kis szimplexének csúcsait címkéző k-futamok között van. Legyenek $y_0, \ldots, y_k$ T csúcsai abban a sorrendben, amelyet a T-t előállító „végrehajtás" meghatároz (amint azt a Bermuda-háromszög definíciójában leírtuk). Legyenek $\rho_0, \ldots, \rho_k$ az ezeket a csúcsokat címkéző, megfelelő k-futamok.

Az első lemma azt mondja ki, hogy minden folyamat, amely ezen k-futamok valamelyikében hibás, legalább egy jellel rendelkezik ezen futamok mindegyikében.

7.4. lemma. . *Ha a P_i folyamat hibás egy ρ_v $(0 \leq v \leq k)$ futamban, akkor a P_i folyamathoz minden ρ_v futamban tartozik jel.*

Bizonyításvázlat. Az állítás igaz, mivel a változások minden *sorozat*-ban fokozatosak, és mivel egy jel mozgása és egy hármas eltávolítása két külön lépésben

valósul meg. A részletes bizonyítást meghagyjuk gyakorlatnak (lásd 7-4. gyakorlat). □

A második lemma korlátot tartalmaz a futamokban előforduló összes hiba számára.

7.5. lemma. . *Legyen F_v ($v \in \{0, \ldots, k\}$) azoknak a folyamatoknak a halmaza, amelyek hibásak ρ_v-ben. Legyen $F = \cup_v F_v$. Ekkor $|F| \leq rk \leq f$.*

Bizonyítás. Meghagyjuk gyakorlatnak (lásd 7-5. gyakorlat). A bizonyítás felhasználja a 7.4. lemmát. □

Végül T csúcsainak folyamatindexekkel való címkézését vizsgáljuk meg. T egy *helyi folyamatcímkézése* az $i_0, \ldots, i_k$ indexű folyamatok hozzárendelése az $y_0, \ldots, y_k$ csúcsokhoz olymódon, hogy minden v-re teljesül az, hogy az i_v indexű folyyamatnak nincs jele ρ_v-ben. A T csúcsait címkéző k-futamok fontos tulajdonsága, hogy ha létezik T helyi folyamatcímkézése, akkor T egyetlen végrehajtási sorozattal konzisztens.

7.6. lemma. . *Legyen $i_0, \ldots, i_k$ a T szimplex egy helyi folyamatcímkézése. Akkor van olyan, legfeljebb f hibát tartalmazó ρ futam, hogy minden v-re teljesül, hogy P_{i_v} nem hibás ρ-ban, továbbá végrehajtás(ρ_v) és végrehajtás(ρ) megkülönböztethetetlenek a P_i folyamat számára.*

Bizonyításvázlat. ρ-t a következőképpen definiáljuk. A kezdeti értéket minden ρ-beli P_i folyamatra úgy definiáljuk, hogy legyen egyenlő P_i kezdeti értékével a ρ_v-k valamelyikében. Az (i, j, t) $(1 \leq t \leq r-1)$ hármast pontosan akkor vesszük bele ρ-ba, ha minden ρ_v-ben benne van. Hasonlóképpen az (i, j, r) hármast pontosan akkor vesszük bele, ahol a j fogadó az i_v indexű folyamatok mindegyikétől különbözik, ha minden ρ_v-ben benne van. Végül az olyan (i, j, r) hármast, ahol $j = i_v$ (egy speciális v-re) pontosan akkor vesszük bele, ha benne van ρ_v-ben (ugyanarra a v-re).

Meghagyjuk gyakorlatnak (lásd 7-6. gyakorlat) annak megmutatását, hogy ρ minden szükséges tulajdonsággal rendelkezik: azaz valóban futam, legfeljebb f hibát tartalmaz és az i_v indexű folyamat minden v-re hibátlan ρ-ban, valamint hogy *végrehajtás*(ρ_v) és *végrehajtás*(ρ) megkülönböztethetetlenek az i_v indexű folyamat számára. A bizonyítás a hibaszám korlátjához a 7.5. lemmát használja fel. □

***B* címkézése folyamatindexekkel..** Emlékeztetünk arra, hogy célunk folyamatindexek hozzárendelése B csúcsaihoz úgy, hogy minden T kis szimplexre legyen olyan végrehajtási sorozat, amely kompatibilis a T csúcsait címkéző végrehajtási sorozatokkal és folyamatokkal. A 7.6. lemma módszert ad ennek megoldására: B minden csúcsához kiválasztunk egy folyamatot úgy, hogy a megfelelő k-futamban nincs jel és a kis szimplexek csúcsaihoz választott folyamatok mind különbözők. Ekkor a 7.6. lemmából minden kis szimplexre adódik a szükséges kompatibilitási feltétel.

B globális folyamatcímkézését folyamatoknak B csúcsaihoz való olyan hozzárendeléseként definiáljuk, hogy a tetszőleges x csúcshoz rendelt folyamatnak

nincs jele az x-et címkéző k-futamban, és minden T kis szimplexre fennáll, hogy a T csúcsaihoz rendelt folyamatok különbözők. B globális folyamatcímkézése B minden kis szimplexére nézve helyi folyamatcímkézést eredményez.

Most elkészítjük B globális folyamatcímkézését. (Mivel ez pusztán technikai feladat, az első olvasáskor átugorhatjuk, és közvetlenül a 7.10. lemmával folytathatjuk.) A konstrukciót azzal kezdjük, hogy a B csúcsát címkéző ρ k-futamhoz hozzákapcsolunk egy *élő*(ρ) folyamathalmazt, majd minden *élő*(ρ) halmazból választunk egy folyamatot. A *élő*(ρ) halmazok rendelkeznek a következő tulajdonságokkal.

1. Minden *élő*(ρ) halmaz pontosan $n - rk$ folyamatot tartalmaz. (Mivel feltettük, hogy $n \geq f + k + 1$ és $f \geq rk$, ez azt jelenti, hogy minden *élő*(ρ) halmaz tartalmaz legalább $k + 1$ folyamatot.)
2. A *élő*(ρ) halmazban lévő folyamatokat azok közül választjuk, amelyek ρ-ban nem rendelkeznek jellel.
3. Ha ρ és ρ' B-ben ugyanannak a kis szimplexnek két csúcsát címkéző k-futamok és ha a $P_i \in$ *élő*(ρ) $\cap$ *élő*(ρ'), akkor P_i rangja a két halmazban azonos[1]

Rögzítsük ρ egyik k-futamát. Ez a futam pontosan rk jelet tartalmaz; legyen a *jelek* a folyamatindexeknek egy multihalmaza, amely megadja az egyes folyamatokhoz rendelt jelek számát. A *jelek* multihalmazt „kisimítjuk", hogy megkapjuk az új *új_jelek* multihalmazt, amelyben ugyanannyi jel van, de ebben minden folyamathoz legfeljebb egy jel van hozzárendelve. Ugyancsak teljesül, hogy minden olyan folyamatnak, amelynek volt jele a *jelek* multihalmazban, lesz jele a *új_jelek* multihalmazban is. A SIMÍT eljárás a következőképpen működik.

SIMÍT eljárás (vázlatosan)

új_jelek := *jelek*
while *új_jelek*-ben van többszörös elem **do**
  válasszunk egy ilyen elemet, például i-t
  if van olyan $j < i$, melyre *új_jelek*$(j) = 0$ **then** mozgassunk egy jelet P_i-től a legnagyobb ilyen P_j-hez
  else mozgassunk el egy jelet P_i-től a legkisebb olyan $j > i$ indexű folyamathoz, melyre *új_jelek* $= 0$

Legyen $l(\rho)$ az olyan P_i folyamatok indexeinek halmaza, melyekre *új_jelek*$(i) = 0$.

Az könnyen belátható, hogy az így definiált *élő* halmaz rendelkezik az előbb említett tulajdonságok közül az első kettővel. A harmadik tulajdonság belátásához rögzítsünk egy T kis szimplexet, melynek csúcsai a szimplexet előállító bejárás sorrrendjében legyenek $y_0, y_1, \ldots, y_k$, és legyenek $\rho_0, \rho_1, \ldots, \rho_k$ a megfelelő k-futamok, amelyek ezeket a csúcsokat címkézik. Először jegyezzük meg azt, hogy amikor sorrendben haladunk T csúcsain, akkor ha a P_i folyamat kapott egy jelet, akkor később a bejárás során végig rendelkezni fog jellel.

[1] Egy i elem rangja az L teljesen rendezett halmazban az i-nél nem nagyobb elemek száma.

7.7. lemma. . *Legyen $v < v' < v''$. Ha a P_i folyamatnak nincs jele ρ_v-ben, de van jele $\rho_{v'}$-ben, akkor P_i-nek van jele $\rho_{v''}$-ben.*

Bizonyítás. Meghagyjuk gyakorlatnak (lásd 7-7. gyakorlat). □

Most már be tudjuk bizonyítani az *élő* halmazok harmadik tulajdonságát.

7.8. lemma. . *Ha $P_i \in$ élő$(\rho_v) \cap$ élő(ρ_w), akkor P_i rangja élő(ρ_v)-ben és élő(ρ_w)-ben ugyanaz.*

Bizonyítás. Az általánosság megszorítása nélkül feltehetjük, hogy $v < w$. Mivel $P_i \in$ élő(ρ_v) és $P_i \in$ élő(ρ_w), P_i-nek nincs jele sem ρ_v-ben, sem ρ_w-ben. Ekkor a 7.7. lemmából következik, hogy a $\rho_0, \ldots, \rho_w$ futamok egyikében sincs P_i-nek jele.

Mivel jelek elhelyezése szomszédos k-futamokban legfeljebb egy jelnek egy folyamattól szomszédos folyamathoz való mozgatásával különbözik, és mivel P_i-nek ezen futamok egyikében sincs jele, következik, hogy az i-nél kisebb indexű folyamatoknál lévő összes jel száma ugyanaz, mondjuk s, a $\rho_0, \ldots, \rho_w$ futamok mindegyikére. Mivel $i \in$ élő(ρ_v), ezért a SIMÍT eljárás működéséből következik, hogy $s < i$. (Ha $s \geq i$, akkor az i-nél kisebb indexű folyamatoknál induló jelek a SIMIT eljárásban „túlcsordulnak", azaz P_i-nél végeznek). Ezért biztosítva van az, hogy P_i rangja *élő*ρ_v-ben és *élő*ρ_w-ben ugyanaz az $i - s$ érték. □

Készen állunk arra, hogy B csúcsait megcímkézzük folyamatindexekkel. Legyen $x = (x_1, \ldots, x_k)$ B tetszőleges csúcsa, és legyen ρ a csúcs k-futama; válasszunk egy folyamatindexet az élő(ρ) halmazból. Nevezetesen, legyen *sík*$(x) = \sum_{i=1}^{k} x_i \pmod{(k+1)}$; x-et azzal a folyamattal címkézzük, amelynek rangja élő(ρ)-ban *sík*(x). Ezt a választást B következő tulajdonsága indokolja.

7.9. lemma. . *Ha x és y ugyanannak a kis szimplexnek különböző csúcsai, akkor sík$(x) \neq$ sík(y).*

Most megfogalmazzuk azt az állítást, melyre szükségünk van.

7.10. lemma. . *B folyamatindexekkel való, most ismertetett címkézése globális folyamatcímkézés.*

Bizonyítás. Mivel minden x csúcs indexét a élő(ρ) halmazból választottuk, ahol ρ a csúcshoz kapcsolt k-futam, ehhez az indexhez ρ-ban nem tartozhat jel. Bármely T rögzített kis szimplexre következik a 7.8. és 7.9. lemmákból, hogy a választott indexek különbözők. □

Most összegezzük mindazt, amit az előállított címkézésekről tudunk.

7.11. lemma. . *B k-futamokkal és folyamatokkal való, adott címkézései rendelkeznek a következő tulajdonsággal. Minden olyan T kis szimplexre, amely a $\rho_0, \ldots, \rho_k$ futamcímkékkel és $i_0, \ldots, i_k$ folyamatcímkékkel van ellátva, létezik olyan ρ futam legfeljebb f hibával, hogy minden v-re teljesül az, hogy i_v nem hibás ρ-ban és végrehajtás(ρ_v) és végrehajtás(ρ) megkülönböztethetetlenek az i_v folyamat számára.*

Bizonyítás. Az állítás a 7.6. és 7.10. lemmákból következik. □

Sperner-lemma.. Majdnem készen vagyunk. Csak az van hátra, hogy megfogalmazzuk a Sperner-lemmát (a Bermuda-háromszögre vonatkozó speciális esetben), és alkalmazzuk az ellentmondás eléréséhez. Ez megadja a k-megegyezés eléréséhez szükséges menetek számára vonatkozó alsó korlátot. *B Sperner-színezése* hozzárendeli egy $k+1$ színt tartalmazó színhalmaz valamelyik elemét B minden csúcsához a következőképpen.

1. B $k+1$ sarokcsúcsának színei mind különbözőek.
2. Bármely, a B külső élén fekvő csúcs színe az adott él valamelyik végpontjának színével egyezik meg.
3. Általánosan, egy külső oldal (bármelyik dimenzióban) bármely belső pontjának színe B szomszédos sarokcsúcsai valamelyikének színe.

A Sperner-színezésnek van egy figyelemre méltó tulajdonsága: van legalább egy olyan kis szimplex, melynek $k+1$ csúcsa $k+1$ különböző színnel van kiszínezve.

7.12. lemma. . ***(Sperner-lemma B-re)*** *B bármely Sperner-színezésére van B-ben legalább egy olyan kis szimplex, melynek $k+1$ csúcsa $k+1$ különböző színnel van kiszínezve.*

Emlékeztetünk arra, hogy feltételezésünk szerint A olyan algoritmus a k-megegyezés megoldására, amely képes f hiba eltűrésére, és legfeljebb $\lfloor \frac{f}{k} \rfloor$ menet után megáll. B C_A színezését a következőképpen definiáljuk. Ha egy x csúcs a ρ futammal és a P_i folyamattal van címkézve, akkor x-et arra a színre színezzük, amelyet a P_i folyamat választ az A algoritmus *végrehajtás*(ρ)
végrehajtási sorozatában.

7.13. lemma. . *Ha A egy f hibát tűrő k-megegyezési algoritmus, amely $\lfloor \frac{f}{k} \rfloor$ menet alatt befejezik, akkor C_A a B háromszög Sperner-színezése.*

Bizonyítás. A k-megegyezés érvényességi feltétele segítségével történhet. □

Most már kimondhatjuk a fő tételt.

7.14. tétel. . *Ha $n \geq f+k+1$, akkor a k-megegyezés megoldására nincs olyan algoritmus, amely f hiba esetében biztosítja, hogy minden hibátlan folyamat legfeljebb $\lfloor \frac{f}{k} \rfloor$ menet alatt döntsön.*

Bizonyítás. A 7.13. lemmából következik, hogy C_A Sperner-színezés, ezért a 7.12. Sperner-lemmából következik, hogy van olyan T kis szimplex, melynek minden csúcsa különböző színű a C_A szerint.

Tegyük fel, T k-futam címkéi $\rho_0, \ldots, \rho_k$ és folyamatcímkéi $i_0, \ldots, i_k$. C_A definíciója szerint ez azt jelenti, hogy a $k+1$ különböző i_v indexű folyamat megfelelő *végrehajtás*(ρ_v) végrehajtási sorozataiban mind a $k+1$ különböző döntés előfordult. A 7.11. lemmából következik, hogy van olyan ρ futam legfeljebb f hibával, hogy minden v-re teljesül az, hogy P_{i_v} nem hibás ρ-ban, továbbá *végrehajtás*(ρ_v) és *végrehajtás*(ρ) megkülönböztethetetlenek az i_v indexű folyamat számára. Ez azonban azt jelenti, hogy ρ-ban az $i_0, \ldots, i_k$ indexű folyamatok (számuk $k+1$) $k+1$ különböző értéket választanak, amivel megsértik a k-megegyezési feladat megegyezési feltételét. □

7.2.. Közelítő megegyezés

Ebben az alfejezetben a *közelítő megegyezés* problémáját bizánci hibák előfordulása mellett vizsgáljuk. Ebben a problémában a folyamatok valós bemenetekkel kezdenek, és feltesszük, hogy végül valós kimeneteket választanak. Üzeneteikben valós értékeket küldhetnek. Ezúttal nem pontos értékben kell megegyezniük, mint a közönséges megegyezési problémánál, hanem egymástól legfeljebb egy kis pozitív ϵ-nal eltérő értékekben. Pontosabban a következő feltételeknek kell teljesülniük.

Megegyezés. Bármely két hibátlanul működő folyamat döntési értékei legfeljebb ϵ-nal térnek el egymástól.

Érvényesség. Bármely hibátlanul működő folyamat döntési értéke a hibátlanul működő folyamatok kezdeti értékei által meghatározott tartományba esik.

Befejeződés. Minden hibátlanul működő folyamat végül döntést hoz.

Ez a feladat előfordul például az óraszinkronizáló algoritmusokban, ahol a folyamatok óraértékeket kezelnek, de ezek az értékek nem egyeznek meg pontosan. A gyakorlatban számos osztott hálózati algoritmus dolgozik közelítőleg szinkronizált órákkal, ahol rendszerint elegendő az óraértékek közelítő megegyezése.

Itt csak teljes gráfokban vizsgáljuk a közelítő megegyezési problémát. A feladat megoldásának egyik módja, hogy eljárásként alkalmazzuk a közönséges bizánci megegyezési algoritmust. Ehhez feltesszük, hogy $n > 3f$.

> **BizKözelMegegyezés (vázlatosan)**
> A folyamatok egy közönséges bizánci megegyezés algoritmust futtatnak, hogy minden folyamat számára egy értéket válasszanak. Ezek az algoritmusok párhuzamosan futnak. A P_i folyamat algoritmusa az első menetben üzenetet küld minden más folyamatnak, azután a folyamatok a kapott értékeket bemenő adatként használják fel egy bizánci megegyezést kereső algoritmusban. Amikor ezek az algoritmusok befejeznek, minden jó folyamat ugyanazokat a döntési értékeket választja minden folyamat számára. Minden folyamat a döntési értékek multihalmazából az $\lfloor \frac{n}{2} \rfloor$-edik legnagyobb értéket választja saját döntési értékeként.

A működés jobb megértéséhez megjegyezzük, hogy ha P_i nem hibás, akkor a bizánci megegyezés

érvényességi feltétele biztosítja, hogy a hibamentes folyamatok által P_i számára választott érték P_i aktuális bemeneti értéke. Mivel $n > 3f$, ezért a multihalmaz középső értékének a hibamentes folyamatok kezdeti értékeinek tartományában kell lennie.

7.15. tétel. . *Ha $n > 3f$, akkor a* BizKözelMegegyezés *algoritmus egy n-csúcsú teljes gráfra megoldja a közelítő megegyezés problémáját.*

Most egy másik algoritmust mutatunk be, amely nem alkalmazza a bizánci megegyezést. Főleg azért mutatjuk be ezt az algoritmust, mert könnyű kiterjeszteni az aszinkron hálózati modellre, amelyet majd a 21. fejezetben tárgyalunk. A bizánci megegyezés problémája az aszinkron hálózatokban nem oldható meg. A második megoldásnak olyan tulajdonsága is van, hogy esetenként kevesebb menet után befejeződik, mint amit a bizánci megegyezés igényel. Ez attól függ, milyen messze vannak egymástól a hibátlan folyamatok kezdeti értékei. Az algoritmus a fokozatos közelítésen alapul. Az egyszerűség kedvéért először az algoritmus be nem fejeződő változatát írjuk le, azután a befejeződést külön elemezzzük. Ez az algoritmus is felteszi, hogy $n > 3f$.

Szükségünk van néhány jelölésre és kifejezésre. Először, ha U egy valós számokból álló véges multihalmaz legfeljebb $2f$ elemmel, és $u_1, \ldots, u_k$ a U multihalmaz elemeinek nem csökkenő sorozata, akkor legyen *csökkentés*(U) annak eredménye, hogy U-ból eltávolítjuk az f legkisebb és f legnagyobb elemet, azaz legyen az $u_{f+1}, \ldots, u_{k-f}$ elemekből álló multihalmaz. Ha U egy valós számokból álló, véges, nem üres multihalmaz, és $u_1, \ldots, u_k$ a U multihalmaz elemeinek nem csökkenő sorozata, akkor legyen *kiválasztás*(U) annak eredménye, hogy U-ból kiválasztjuk a legkisebb elemét és azután minden f-edik elemet, azaz $u_1, u_{f+1}, u_{2f+1}, \ldots$. Végül, ha U egy valós számokból álló, véges, nem üres multihalmaz, akkor legyen *átlag*(U) az U multihalmaz elemeinek számtani közepe.

Azt mondjuk, hogy egy valós számokból álló, véges, nem üres multihalmaz *tartománya* a legkisebb intervallum, amely a multihalmaz minden elemét tartalmazza, és *szélessége* a tartomány hossza.

A második megoldás a következő.

KovergensKözelMegegyezés algoritmus (vázlatosan)

A P_i folyamat karbantart egy *érték* változót, amely az utolsó becslést tartalmazza. Kezdetben *érték*$_i$ a P_i folyamat kezdeti értékét tartalmazza. P_i minden menetben a következőket végzi el.

Először, elküldi saját *érték*-ét minden folyamatnak, beleértve sajátmagát is.[2] Ezután az adott menetben kapott összes értéket összegyűjti egy W multihalmazban; ha P_i nem kap értéket egy bizonyos másik folyamattól, akkor a multihalmazban egy előre megadott értéket rendel ahhoz a folyamathoz, és ezzel biztosítja, hogy $|W| = n$ teljesüljön.

Ezután P_i beállítja *érték*-et *átlag*(*választás*(*csökkentés*(W)))-re, azaz P_i eldobja W f legkisebb és f legnagyobb elemét. A megmaradó elemek közül P_i csak az i-edik elemet és attól számítva minden f-edik elemet választja. Végül az így kiválasztott elemek átlagát teszi az *érték*-be.

Azt állítjuk, hogy bármelyik menetre igaz, hogy a hibátlan folyamatok *érték*-ei a hibátlan folyamatoknak közvetlenül az adott menet előtti *érték*-ei által meghatározott tartományban vannak. Továbbá az is igaz minden menetre, hogy a hibátlan folyamatok *érték*-ei multihalmazának szélessége minden menetben lega-

[2] A sajátmagának való küldést rendszerint helyi átmenettel szimulálják.

lább $(\lfloor\frac{n-2f-1}{f}\rfloor+1)$-ed részére csökken. Ha $n>3f$, akkor ez az „osztótényező" egynél nagyobb.

7.16. lemma. . *Tegyük fel, hogy pontosan a* KovergensKözelMegegyezés *egyik végrehajtásának r-edik menete után érték$_i$ = v, ahol P_i egy hibátlan folyamat. Akkor közvetlenül az r-edik menet előtt v a hibátlan folyamatok tartományában van.*

Bizonyítás. Ha W_i a P_i folyamat által az r-edik menetben összeállított multihalmaz, akkor W_i-nek legalább f olyan eleme van, melyek hibás folyamatok által küldött értékek. Ekkor közvetlenül az r-edik menet előtt *csökkentés*(W_i) minden eleme a hibátlan folyamatok *érték*-einek a tartományában van. Ebből következik, hogy ugyanez teljesül *átlag*(*választás*(*csökkentés*(W_i)))-re, azaz a *érték*$_i$ új értékére is. □

7.17. lemma. . *Tegyük fel, hogy pontosan a* KovergensKözelMegegyezés *egyik végrehajtási sorozatának éppen az r-edik menete után lesz az érték$_i$ = v és érték$_{i'}$ = v′, ahol i és i′ két hibátlan folyamat indexe. Ekkor*

$$|v-v'|\leq\frac{d}{\lfloor\frac{n-2f-1}{f}\rfloor+1}\ ,$$

ahol d a hibátlan folyamatokhoz közvetlenül az r-edik menet előtt tartozó érték-ek tartományának szélessége.

Bizonyítás. Legyen W_i, illetve W_i' a P_i és $P_{i'}$ folyamatok által az r-edik menetben összegyűjtött multihalmaz. Legyen S_i, illetve $S_{i'}$ a *kiválasztás*(*csökkentés*(W_i)), illetve a *kiválasztás*(*csökkentés*(W_i')) multihalmaz. Legyen $c=\lfloor\frac{n-2f-1}{f}\rfloor+1$; megjegyezzük, hogy c pontosan az S_i-ben és az S_i'-ben lévő elemek száma. Jelöljük S_i elemeit $u_1,\ldots,u_c$-vel, S_i' elemeit pedig $u_1',\ldots,u_c'$-vel, mindkét esetben nem csökkenő sorrendben. Egy segédtétellel kezdünk, amely szerint a csökkentett multihalmazok mérete legfeljebb f elemmel különbözik.

7.18. segédtétel. . $|$*csökkentés*$(W_i)-$*csökkentés*$(W_{i'})|\leq f$.

Bizonyítás. Mivel a hibátlanul működő folyamatok mindegyike egy-egy értéket ad mind W_i-hez, mind pedig $W_{i'}$-höz, ezért $|W_i-W_{i'}|\leq f$. Meg tudjuk mutatni, hogy egy-egy legkisebb elemet eltávolítása mindkét halmazból nem növeli a különbséghalmazban lévő elemek számát, és ugyanez érvényes egy-egy legnagyobb elem eltávolítására is. Ezt a két megállapítást f-szer alkalmazva megkapjuk a kívánt eredményt. □

A 7.18. segédtétel segítségével bizonyítjuk a következő segédtételt.

7.19. segédtétel. . *Ha* $1\leq j\leq c-1$, *akkor* $u_j\leq u_{j+1}'$ *és* $u_j'\leq u_{j+1}$.

Bizonyítás. Csak az első állítást bizonyítjuk; a második állítás bizonyítása hasonló. Kihasználjuk, hogy u_j a *csökkentés*(W_i) multihalmaz $((j-1)f+1)$-edik

legkisebb eleme, továbbá u'_{j+1} a *csökkentés*$(W_{i'})$ $(jf+1)$-edik legkisebb eleme. Mivel a 7.18. segédtétel szerint *csökkentés*$(W_{i'})$-nek legfeljebb f eleme nem eleme *csökkentés*(W_i)-nek, ezért $u_j \leq u'_{j+1}$. □

A kívánt korlát kiszámításával befejezzük a 7.17. lemma bizonyítását.

$$|v - v'| = |\mathit{átlag}(S_i) - \mathit{átlag}(S_{i'})| = \frac{1}{c}|(\sum_{j=1}^{c}(u_j - u'_j))| \leq$$

$$\leq \frac{1}{c}(\sum_{j=1}^{c}|u_j - u'_j|) = \frac{1}{c}(\sum_{j=1}^{c}(\max(u_j, u') - \min(u_j, u'_j))).$$

A 7.19. segédtétel szerint $\max(u_j, u'_j) \leq \min(u_{j+1}, u'_{j+1})$ teljesül minden olyan j-re, amelyre $1 \leq j \leq c-1$, ezért az utolsó kifejezés értéke legfeljebb

$$\frac{1}{c}(\sum_{j=1}^{c-1}(\min(u_{j+1}, u'_{j+1}) - \min(u_j, u'_j))) + \frac{1}{c}(\max(u_c, u'_c) - \min(u_c, u'_c)),$$

amiből összevonással adódik, hogy

$$\frac{1}{c}(\max(u_c, u'_c) - \min(u_1, u'_1)).$$

Mivel *csökkentés*(W_i) és *csökkentés*$(W_{i'})$ minden eleme a hibátlan folyamatok r-edik menete előtt *érték*-einek tartományában van, ezért az u_c, u'_c, u_1 és u'_1 értékek is ebben a tartományban vannak. Ezért az utolsó kifejezés legfeljebb $\frac{d}{c}$, és éppen ezt akartuk belátni. □

Befejeződés.. A KonvergensKözelMegegyezés algoritmust befejeződő algoritmussá alakítjuk, azaz olyan algoritmussá, amelyben végül minden folyamat dönt. (Valójában végül minden folyamat megáll.) Nevezetesen, minden hibátlan folyamat az első menetben kapott értékek tartományát használja annak a menetszámnak a meghatározásához, amikor már bármely két hibátlan folyamat *érték*-ei legfeljebb ϵ-nal különbözhetnek. Ezt minden folyamat megteheti, mivel ismeri ϵ értékét és a garantált konvergenciasebességet, továbbá tudja, hogy azok az értékek, melyeket az első menetben kap, minden hibátlan folyamat kezdeti értékeit tartalmazzák. Különböző hibátlan folyamatok azonban különböző menetszámokat számolhatnak ki.

Minden P_i folyamat, amely eléri az általa kiszámított menetet, a saját aktuális *érték*-ét választja döntési értéknek. Miután ezt megtette, a P_i folyamat szórja a saját *érték*-ét egy különleges *leállás* címkével együtt, majd leáll. Ha bármely P_j folyamat P_i-től *leállás* címkével együtt kapja az *érték*-et, akkor ezt használja P_i üzeneteként az adott és minden későbbi menetre nézve (mindaddig, amíg a saját kiszámított menetszáma alapján P_j úgy nem dönt, hogy ő is leáll).

Bár a hibátlan folyamatok különböző menetszámokat számolhatnak, világos, hogy a legkisebb ilyen becslés is helyes. Ezért amikor a legelső hibátlan folyamat

leáll, *érték*-einek tartománya már elég kicsi. A későbbi menetek során a hibátlan folyamatok *érték*-einek tartománya sohasem nő, bár arra nincs garancia, hogy tovább csökken.

7.20. tétel. . *Ha $n > 3f$, akkor a fenti befejeződéssel kiegészített* KonvergensKözelMegegyezés *egy n-csúcsú teljes gráfra megoldja a közelítő megegyezési problémát.*

Bonyolultságelemzés.. Nincs az n, f, ϵ értékektől és a hibátlan folyamatok kezdeti értékeiből álló multihalmaz szélességétől függő felső korlát arra, hogy mennyi idő alatt döntenek a KonvergensKözelMegegyezés algoritmusban a hibátlan folyamatok. Ennek oka, hogy a hibás folyamatok az első menetben tetszőleges értéket küldhetnek, ami azt okozhatja, hogy a hibátlan folyamatok akármilyen nagy menetszámot is kiszámolhatnak befejeződésükhöz.

A gyakorlatokban elemezzük azokat a folyamatok számára és az összefüggőségre vonatkozó korlátokat, amelyeknek teljesülniük kell a közelítő megegyezési feladat megoldásához. A 21. fejezetben az aszinkron hálózati modellel kapcsolatban újra tárgyaljuk ezeket a problémákat.

7.3.. A véglegesítési feladat

Ebben a szinkron rendszerek osztott megegyezési problémáiról szóló utolsó alfejezetben az *osztott adatbázis véglegesítés* problémájának kulcsgondolatait mutatjuk be. Amint azt az 5.1. alfejezetben elemeztük, a feladat akkor merül fel, amikor egy adatbázis tranzakció végrehajtásában több folyamat vesz részt. A feldolgozás után minden folyamat kialakítja kezdeti „véleményét” arról, hogy a tranzakciót *véglegesíteni* kell (azaz eredményeit véglegessé és más folyamatok számára hozzáférhetővé tenni), vagy *elvetni* (azaz eredményeit eldobni). Egy folyamat általában a véglegesítést részesíti előnyben, ha a tranzakcióval kapcsolatos számításait sikeresen befejezte, egyébként pedig inkább az eldobást választja. Feltesszük, hogy a folyamatok kommunikálnak, és végül megegyeznek a *véglegesítés* vagy *eldobás* kimenetben. Ha lehetséges, akkor az eredmény *véglegesítés*.

Léteznek olyan erre a problémára megoldások a gyakorlatban előforduló osztott hálózatok estén is, amelyekben folyamat- és vonalhibák együtt is előfordulhatnak. Az 5. fejezet eredményeiből azonban az következik, hogy ha nincs felső korlát a vonalhibák számára, akkor nincs megoldás. Ezért az üzenetvesztésekre bizonyos korlátokat kell feltételeznünk.

7.3.1.. A feladat

A véglegesítési problémának azt az egyszerűsített változatát tekintjük, amelyben a hálózatban nincs üzenetvesztés, csak folyamathiba. Ha ennek a fejezetnek az algoritmusait működő hálózatokban akarjuk megvalósítani, akkor más mechanizmusokat is kell alkalmazni kell, például ismételt átvitelt az elveszett üzenetek pótlására. Tetszőleges számú folyamat megállási hibát megengedünk.

Feltesszük, hogy a bemeneti tartomány $\{0, 1\}$, ahol az 1 a *véglegesítésnek*, 0 pedig az *elvetésnek* felel meg. Figyelmünket arra az esetre korlátozzuk, amelyben a hálózat gráfja teljes. A helyességi feltételek a következők.

Megegyezés. Bármely két folyamatnak ugyanazt az értéket kell választania.

Érvényesség.

1. Ha bármely folyamat 0 értékkel kezd, akkor 0 az egyetlen lehetséges döntési érték.
2. Ha minden folyamat az 1 értékkel kezd és nincs hibás folyamat, akkor az 1 az egyetlen lehetséges döntési érték.

Befejeződés. Ennek két változata van. A *gyenge befejeződési feltétel* (GyBF) azt mondja, hogy ha nincs hiba, akkor végül minden folyamat dönt. Az *erős befejeződési feltétel* (EBF) (amelyet *nemblokkolási feltételnek* is neveznek) azt mondja, hogy végül minden hibátlan folyamat dönt.

Az erős befejeződési feltételt kielégítő véglegesítő algoritmusokat időnként *nem blokkoló véglegesítő algoritmusoknak* is nevezzük, míg a gyenge befejeződési feltételt kielégítő, de az erős befejeződési feltételt nem kielégítő véglegesítő algoritmusokat időnként *blokkoló algoritmusoknak* nevezzük.

Megjegyezzük, hogy az a megegyezési feltételünk, hogy *bármely két folyamat ugyanazt az értéket választja*. Ezek szerint egy hibás folyamatnak sem engedjük meg, hogy a többi folyamattól eltérő értéket válasszon. Ezt azért követeljük meg, mert a véglegesítő protokollok gyakorlati alkalmazásaiban egy folyamat hibássá válhat, majd később újra helyesen működhet. Tegyük fel például, hogy a P_i folyamat hibássá válása előtt a *véglegesítés* döntést választja, később pedig más folyamatok az *eldobást* választják. Ha a P_i folyamat helyreállítódik és fenntartja a *véglegesítést*, ellentmondás jön létre.

A feladat formális megfogalmazása hasonló ahhoz a két esethez, amelyeket korábban már vizsgáltunk: az összehangolt támadáshoz az 5.1. alfejezetben és a megegyezési problémához megállási hibák esetében a 6.1. alfejezetben. A véglegesítési feladat és az összehangolt támadás problémája között a legfontosabb különbség az, hogy itt folyamathibát vizsgálunk, és nem vonalhibát; és különbség van az érvényességi feltételben is. A véglegesítési feladat és a megállási megegyezési feladat közötti fontos különbségek: elsősorban az eltérő érvényességi feltétel, másodsorban pedig a gyengébb befejeződési feltétel. A 6.7. alfejezetnek a megállási megegyezési problémára vonatkozó eredményeiből következik, hogy az erős befejeződési feltétel esetében a véglegesítési feladat megoldásához legalább $n-1$ menetre van szükség. (Megjegyezzük, hogy a 6.33. tétel bizonyítása a véglegesítési érvényességi feltétel esetében is helyes.)

Az alfejezet hátralévő részében két, a gyakorlatban használt véglegesítő algoritmust ismertetünk (azzal az egyszerűsítéssel, hogy csak folyamathibákat engedünk meg). Az első, a *kétfázisú véglegesítés*, blokkoló algoritmus, míg a második, a *háromfázisú véglegesítés*, nem blokkoló. Azután megadunk egy egyszerű alsó korlátot a feladat megoldásához szükséges üzenetek számára abban az esetben, ha csak a gyenge befejeződési feltétel teljesülését követeljük meg.

7.3.2.. Kétfázisú véglegesítés

A gyakorlatban legismertebb véglegesítő algoritmus a *kétfázisú véglegesítés*; ez az egyszerű algoritmus garantálja a gyenge befejeződést.

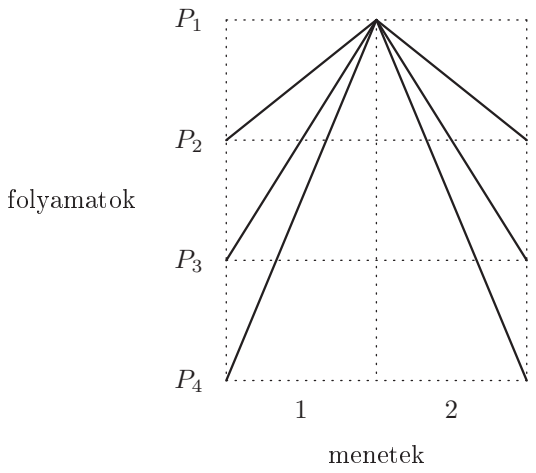
A KétfázisúVéglegesítés algoritmus

Ez az algoritmus feltételez egy kitüntett folyamatot, például P_1-et.

1. menet. P_1 kivételével minden folyamat elküldi kezdeti értékét P_1-hez, és minden olyan folyamat, melynek kezdeti értéke 0, a 0 értéket választja. P_1 ezeket az értékeket és saját kezdeti értékét egy vektorba gyűjti össze. Ha a vektor minden eleme 1, akkor P_1 az 1 értéket választja. Egyébként – azaz, ha van a vektorban olyan elem, amely 0, vagy bizonyos elem hiányzik (mivel a megfelelő folyamattól nem érkezett üzenet) – P_1 a nulla értéket választja.

2. menet. P_1 döntését szórja a többi folyamatnak. Bármely, P_1-től különböző folyamat, amely a második menetben üzenetet kap és az első menetben még nem döntött, azt az értéket választja, amelyet a második menetbeli üzenetben kap.

Lásd a 7.4. ábrát a KétfázisúVéglegesítés hibamentes meneteiben alkalmazott kommunikációs minták illusztrálására.[3]



7.4.. ábra. Kommunikációs minta a KétfázisúVéglegesítés algoritmusban.

7.21. tétel. . *A* KétfázisúVéglegesítés *algoritmus a gyenge befejeződési feltétellel megoldja a véglegesítési problémát.*

Bizonyítás. A megegyezést, érvényességet és befejeződést egyaránt könnyű belátni. □

[3]Menet fogalmunk nem pontosan egyezik meg a kétfázisú protokollokban szokásos fázisok fogalmával. Rendszerint az elején még egy további menetet alkalmaznak, amelyben P_1 a *véglegesítés* vagy *eldobás* értékeket igényli a többi folyamattól. Ekkor az első fázis ebből a többlet menetből és a mi első menetünkből áll. Egyszerűsített modellünknek és feladatleírásunknak köszönhetően nincs szükségünk erre a többlet menetre.

A KétfázisúVéglegesítés azonban nem elégíti ki az EBF-t, azaz blokkoló algoritmus. Ez azért igaz, mert ha P_1 hibázik, mielőtt a második menetben elkezdené a szórást, akkor azok a hibátlan folyamatok, melyek kezdeti értéke 1, sohasem fognak dönteni. A gyakorlatban ha P_1 hibázik, akkor a többi folyamat bizonyos *befejeződési protokollt* hajt végre és bizonyos esetekben sikerül döntést hozniuk. Például ha P_1 hibássá válik, de valamely másik folyamat, például P_i, már az első menetben a 0 értéket választotta, akkor P_1 értesítheti a többi jó folyamatot arról, hogy ő a 0 értéket választotta, és akkor azok biztonságosan választhatják a 0 értéket. A befejeződési protokoll azonban nem minden esetben sikeres. Például tegyük fel azt, hogy P_1 kivételével minden folyamat az 1 bemeneti értékkel kezd, de P_1 hibássá válik, mielőtt bármilyen üzenet küldene. Ekkor egyetlen további folyamat sem értesül P_1 1-es kezdeti értékéről, és így az érvényességi feltétel szerint egyetlen folyamat sem választhatja az 1 értéket. Másrészt, egyetlen folyamat sem választhatja a nullát, mivel bármelyik folyamat mondhatja azt, hogy előfordulhat az, hogy P_1 hibássá válása előtt már az 1 értéket választotta, és az ellentmondás megsértené a megegyezési feltételt.

Bonyolultsági elemzés.. A KétfázisúVéglegesítés algoritmus csak két menetet igényel. Emlékeztetünk arra, hogy a 6.33. tétel $(f+1)$-es alsó korlátot ad a megállási megegyezéshez szükséges menetek számára. A KétfázisúMegegyezés algoritmusra vonatkozó alsó korlát ennek nem mond ellent, mivel a KétfázisúMegegyezés csak a gyenge befejeződési feltételnek tesz eleget. Ha a kommunikációs bonyolultságot a bármely végrehajtásban legrosszabb esetben küldött nem-*null* üzenetek számával mérjük, akkor $2n-2$ értéket kapunk; például egy hibamentes végrehajtásban ennyi az elküldött üzenetek száma.

7.3.3.. Háromfázisú véglegesítés

A HáromfázisúVéglegesítés a KétfázisúVéglegesítés olyan javított változata, amely garantálja az erős befejeződést.

Ennek alapja az, hogy P_1 csak akkor választja az 1 értéket, ha minden olyan folyamat, amely még nem hibázott, „kész” az 1 értéket választani. Egy további menetet igényel, hogy P_1 erről megbizonyosodjon. Először az algoritmus első három menetét írjuk le és elemezzük. Az algoritmus további részét, amely a nem blokkoló tulajdonságot biztosítja, később ismertetjük.

HáromfázisúVéglegesítés algoritmus, első három menet (vázlatosan)

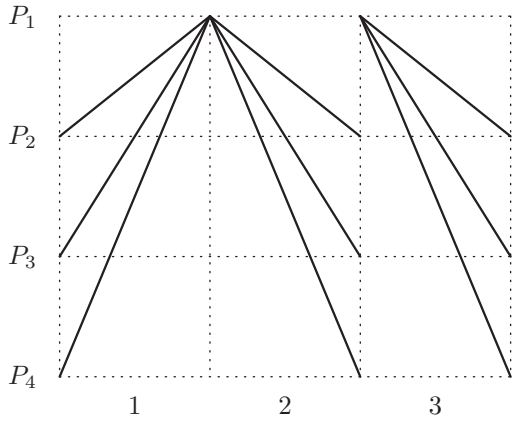
Első menet. P_1 kivételével minden folyamat elküldi kezdeti értékét P_1-nek, és minden olyan folyamat, melynek kezdeti értéke 0, a 0 értéket választja. A P_1 folyamat ezeket a kapott értékekeket és a saját kezdeti értékét egy vektorba gyűjti össze. Ha ennek a vektornak minden eleme 1, akkor P_1 `kész` állapotba kerül, de még nem választ. Egyébként – azaz ha ennek a vektornak van 0 eleme vagy üres eleme (mert bizonyos üzenet nem érkezett meg) – P_1 a 0 értéket választja.

Második menet. Ha a P_1 folyamat a 0 értéket választja, akkor szétküldi

a `dönt`(0) üzenetet. Ha nem, akkor P_1 a `kész` üzenetet szórja. Minden folyamat, amely `kész` üzenetet kap, `kész` állapotba kerül. Ha P_1 még nem választott, akkor az 1 értéket választja.

Harmadik menet. Ha P_1 az 1 értéket választotta, akkor szétküldi a `dönt`(1) üzenetet. Minden folyamat, amely `dönt`(1) üzenetet kap, az 1 értéket választja.

A 7.5. ábra a HÁROMFÁZISÚVÉGLEGESÍTÉS hibamentes futamaiban előforduló kommunikációs mintát ábrázolja.[4]



7.5.. ábra. Kommunikációs minta a HÁROMFÁZISÚVÉGLEGESÍTÉS algoritmusban.

Mielőtt a megfelelő befejeződést biztosító protokollt ismertetnénk, elemezzük az első három menet után kialakuló helyzetet. Minden folyamat (hibás és hibátlan) állapotait négy – egymást kizáró és minden esetet magában foglaló – osztályba soroljuk:

1. 0-*döntés*: azok az állapotok, amelyekben a folyamat a 0 értéket választja;
2. 1-*döntés*: azok az állapotok, amelyekben a folyamat az 1 értéket választja;
3. *kész*: azok az állapotok, amelyekben a folyamat még nem választott, de *kész*;
4. *bizonytalan*: azok az állapotok, amelyekben a folyamat még nem választott és nem *kész*.

A HÁROMFÁZISÚVÉGLEGESÍTÉS kulcstulajdonságait fogalmazza meg a következő lemma: olyan állapotkombinációkat ír le, amelyek együtt nem fordulhatnak elő.

7.22. lemma. . *A* HÁROMFÁZISÚVÉGLEGESÍTÉS *három menete után teljesülnek a következők.*

[4] Az itt alkalmazott menetek nem felelnek meg pontosan a háromfázisú véglegesítési protokollok szokásos fázisainak. Rendszerint hozzáadnak még egy igénylő menetet az elejéhez és valamilyen nyugtázást is használnak.

1. *Ha valamelyik folyamat kész vagy 1-döntés állapotban van, akkor minden folyamat kezdeti értéke 1.*
2. *Ha valamelyik folyamat 0-döntés állapotban van, akkor nincs folyamat 1-döntés állapotban és nincs hibátlan folyamat a kész állapotban.*
3. *Ha valamely folyamat 1-döntés állapotban van, akkor nincs folyamat 0-döntés állapotban és nincs hibátlan folyamat bizonytalan állapotban.*

Bizonyítás. Nyilvánvaló. A bizonyítás egyetlen meggondolandó része a harmadik feltétel belátása. Ehhez megjegyezzük, hogy P_1 csak a második menet végén dönthetaz 1 mellett, miután a *kész* üzeneteket szétküldte. Ez azt jelenti, hogy a második menet végén $P-1$ megtudja, minden más folyamat vagy már megkapta és feldolgozta a *kész* üzenetet, és ezáltal *kész* állapotba került, vagy hibássá vált. (A modell szinkronizáltsága ebben az esetben fontos.) □

Most bebizonyíthatjuk, hogy a bennünket érdeklő feltételek többsége már a harmadik menet után teljesül.

7.23. lemma. . *A* HáromfázisúVéglegesítés *három menete után teljesülnek a következők.*

1. *A megegyezési feltétel teljesül.*
2. *Az érvényességi feltétel teljesül.*
3. *Ha a P_1 folyamat nem hibázott, akkor minden hibátlan folyamat döntött.*

Bizonyítás. A megegyezés feltétele a 7.22. lemmából következik, és az érvényességi feltételnek azt a részét biztosítja, amely szerint ha valamely folyamat a 0 értékkel kezd, akkor 0 az egyetlen lehetséges döntési érték. Az érvényességi feltétel másik része egyszerűen ellenőrizhető.

Végül ha P_1 nem hibás, akkor minden hibátlan folyamat döntött. Ez azért igaz, mert bármit tesz a többi folyamat, P_1 mindenképpen dönteni fog, és ha P_1 döntött, döntését azonnal szétküldi a többi folyamatnak, amelyek ugyanúgy döntenek. □

Ez a három menet azonban még nem elég a nem blokkoló véglegesítési feladat megoldásához, mivel nem garantálják az erős befejeződést. Ha P_1 nem hibás, akkor a 7.23. lemma szerint minden hibamentes folyamat dönt. Ha azonban P_1 hibássá válik, akkor előfordulhat, hogy a többi folyamat bizonytalan állapotban marad. Ennek elkerülése érdekében a többi folyamatnak az első három menet után végre kell hajtania a *befejeződési protokollt*. A pontos részletek többféleképpen is megfogalmazhatók; az alábbiakban egy lehetséges változatot ismertetünk.

HáromfázisúVéglegesítés befejeződési protokoll

4. menet. Minden folyamat (amely még nem hibázott) elküldi jelenlegi állapotát (ami 0-*döntés*, 1-*döntés, kész* vagy *bizonytalan*) P_2-nek. P_2 ezeket az állapotértékekek és a saját állapotát összegyűjti egy vektorba. Nem biztos, hogy a vektor minden eleme ki van töltve – P_2 figyelmen kívül hagyja azokat az elemeket, amelyek nincsenek kitöltve. Ha a vektor csak 0-*döntés* tartalmaz 0-*döntés* értéket, és P_2 még nem döntött, akkor P_2 a 0 értéket választja. Ha a vektor tartalmaz legalább egy 1-*döntés* értéket, és P_2 még

nem döntött, akkor akkor P_2 az 1 értéket választja. Ha a vektor minden kitöltött eleme *bizonytalan*, akkor P_2 a 0 értéket választja. Egyébként – azaz a vektorban csak *bizonytalan* és *kész* értékek vannak, és van legalább egy *kész*, akkor P_2 *kész* állapotba kerül, de még nem dönt.

5. menet. Ebben és a következő menetben P_2 ahhoz hasonlóan viselkedik, ahogy P_1 a 2. és 3. menetben. Ha P_2 már döntött, akkor egy `dönt` üzenetben szétküldi a döntését. Ha nem dönt, akkor a *kész* állapotát szórja. Ha bármely folyamat, amely még nem döntött, `dönt`(0) vagy `dönt`(1) üzenetet kap, akkor az üzenetnek megfelelően a 0, illetve 1 értéket választja. Ha egy folyamat *kész* értéket kap, állapota *kész* lesz. Ha P_2 még nem döntött, akkor az 1 értéket választja.

6. menet. Ha P_2 az 1 értéket választotta, akkor szétküldi a `dönt`(1) értéket. Ha olyan folyamat kap `dönt`(1) üzenetet, amely még nem döntött, az az 1 értéket választja.

A 6. menet után a protokoll hasonló 3-3 menettel folytatódik, melyeket rendre a P_3, $P_4, \ldots,$ P_n koordinálnak.

7.24. tétel. . *A befejeződési protokollt is tartalmazó teljes* HÁROMFÁZISÚVÉGLEGESÍTÉS *nem blokkoló véglegesítő algoritmus.*

Bizonyításvázlat. Először azt mutatjuk meg, hogy a 7.22. lemmában felsorolt 3 tulajdonság nem csak három menet után teljesül(ahogy a lemmában állítottunk), hanem a teljes HÁROMFÁZISÚVÉGLEGESÍTÉS algoritmus *bármelyik* menete után. A bizonyítás a menetek száma szerinti indukcióval történik.

Ezután a megegyezés és az érvényességi feltétel egy része – az, hogy ha valamelyik folyamat a 0 értékkel kezd, akkor 0 az egyetlen lehetséges döntési értéke – a korábbiakhoz hasonlóan következik a 7.22. lemma kiterjesztéséből. Az érvényességi feltétel másik része azért igaz, mert ha nincsenek hibák, akkor minden folyamat dönt az első három menet során.

Az erős befejeződési feltétel van még hátra. Ha minden folyamat hibás, akkor a tulajdonság üres halmazra vonatkozik, ezért biztos igaz. Egyébként tegyük fel, hogy P_i hibátlan. Ekkor azalatt, míg P_i a koordináló folyamat, minden hibátlan folyamat dönteni fog. □

Bonyolultsági elemzés.. A HÁROMFÁZISÚVÉGLEGESÍTÉS itt bemutatott változata $3n$ menetet igényel. Ez még akkor is lényegesen magasabb a 6. fejezetben (ott megállási hibák fordulhattak elő) tanulmányozott megegyezési algoritmusoknál általában elérhető, körülbelül n menetes felső korlátoknál, ha minden folyamat meghibásodhat. Természetesen a megállási hibákat tűrő megegyezési algoritmusok más érvényességi feltételt elégítenek ki, de kis módosítással a véglegesítési érvényességi feltételt is ki tudják elégíteni. Akkor miért jobbak a gyakorlatban a HÁROMFÁZISÚVÉGLEGESÍTÉS-hez hasonló algoritmusok?

A fő ok az, hogy a HÁROMFÁZISÚVÉGLEGESÍTÉS átalakítható úgy, hogy a hibamentes esetben alacsony legyen bonyolultsága. Ha egyetlen folyamat sem hibás, akkor 3 menet alatt minden folyamat dönt. Ekkor egy egyszerű protokoll

segítségével a folyamatok megtudhatják, hogy már minden folyamat döntött, és a befejeződési protokoll további részében már nem vesznek részt. Ezzel a kiegészítéssel az egész algoritmus szükséges meneteinek száma egy kis konstans, az üzenetek száma pedig $\mathcal{O}(n)$.

7.3.4.. Alsó korlát az üzenetek számára

Ezt a fejezetet (és az első részt) azzal zárjuk, hogy megvizsgáljuk a véglegesítési feladat megoldásához szükséges üzenetek számát. Emlékezzünk arra, hogy a KétfázisúVéglegesítés algoritmus a hibamentes esetben $2n-2$ üzenetet használ. A HáromfázisúVéglegesítés valamivel többet, de még mindig $\mathcal{O}(n)$ üzenetet igényel, ha az algoritmust úgy módosítjuk, hogy hamar megálljon. Ebben a szakaszban megmutatjuk, hogy $2n-2$ üzenetnél kevesebb még akkor sem elég a hibamentes esetben, ha megelégszünk egy blokkoló algoritmussal.

7.25. tétel. . *A véglegesítési problémát megoldó algoritmusok még a gyenge befejeződési feltétel esetében is legalább $2n-2$ üzenetet használnak az olyan végrehajtási sorozatokban, amelyekben minden bemenet 1.*

A fejezet hátralévő részében rögzítünk egy A véglegesítő algoritmust, és legyen α_1 A-nak az a hibamentes végrehajtása, amelyben minden bemenet 1. Célunk annak megmutatása, hogy α_1 legalább $2n-2$ üzenetet tartalmaz.

Ismét felhasználjuk a *kommunikációs mintának* a 6.7. alfejezetben megadott definícióját. Ezúttal arra használjuk a kommunikációs mintát, hogy leírja a hibamentes végrehajtási sorozatban elküldött üzenetek halmazát. (A korábbiaktól eltérően nem tételezzük fel, hogy minden folyamat minden menetben minden másik folyamatnak küld üzenetet). Az A algoritmus bármely α hibamentes végrehajtási sorozatából a nyilvánvaló módon készítjük a *minta*(α) kommunikációs mintát.

A különböző folyamatok között különböző időpontokban végbemenő információáramlás leírására felhasználjuk a γ kommunikációs mintának az 5.2.2. szakaszban definiált $\leq_\gamma$ rendezését is. Azt mondjuk, hogy a P_i folyamat a γ kommunikációs mintában hat a P_j folyamatra, ha van olyan k, amelyre $(i,0) \leq_\gamma (j,k)$) teljesül. A következő lemma tartalmazza az alsó korlát alapgondolatát.

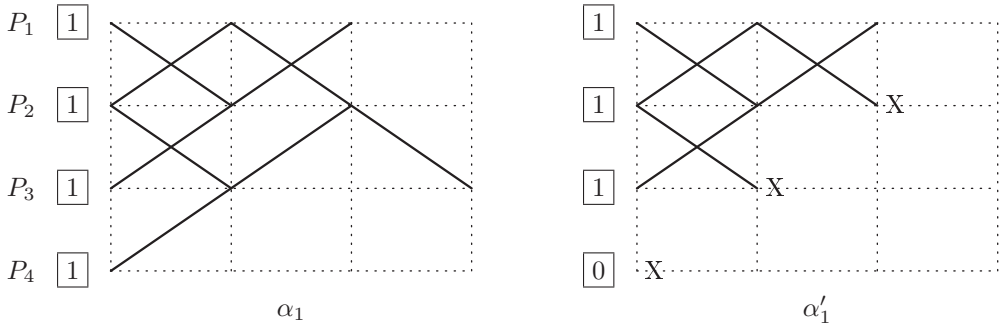
7.26. lemma. . *Bármely P_i és P_j folyamatra fennáll, hogy P_i hat P_j-re a minta(α_1)-ben.*

7.3.1. példa. Alsó korlát a véglegesítésre

A 7.26. lemma bizonyítása előtt bemutatunk egy példát, amely alátámasztja, hogy a lemma helyes. Tegyük fel, hogy α_1 (A-nak az a hibamentes végrehajtási sorozata, amelyben minden bemenet 1) pontosan azokat az üzeneteket tartalmazza, amelyeket a 7.6. ábra bal oldalán megjelöltünk.

Az érvényességi és a gyenge befejeződési feltétel szerint α_1-ben végül minden folyamatnak 1-et kell választania. Vegyük észre, hogy α_1-ben

P_4 nem hat P_1-re; vizsgáljuk meg, milyen problémákat okoz ez. Tekintsünk egy másik, α_1' végrehajtási sorozatot, amely csak annyiban tér el α-tól, hogy P_4 bemenete 0, és minden folyamat hibássá válik közvetlenül azután, hogy először hatott rá P_4. Az α_1' végrehajtási sorozatot a 7.6. ábra jobb oldali része mutatja; a hibákat X jelzi. Közvetlenül megmutatható, hogy $\alpha_1 \overset{1}{\sim} \alpha_1'$, amiből következik, hogy P_1 α_1-ben ugyancsak 1-et választ. Ez azonban megsérti az α_1-re vonatkozó érvényességi feltételt, ami ellenmondás.



7.6.. ábra. Az α_1 és α_1' végrehajtási sorozatban küldött üzenetek.

A 7.26. lemma ugyanazzal a gondolatmenettel bizonyítható, mint amit a 7.3.1. példában alkalmaztunk.

7.26. lemma bizonyítása. Az érvényesség és a gyenge befejeződési feltételek szerint az α_1 mintában végül minden folyamatnak 1-et kell választania. Tegyük fel, hogy a lemma állítása hamis, és rögzítsünk két folyamatot, például P_i-t és P_j-t, amelyekre igaz, hogy P_i nem hat P_j-re $minta(\alpha_1)$-ben. Ekkor i és j definíciója szerint $i \neq j$. Állítsuk elő α_1'-et úgy, hogy α_1-ben P_i bemenetét 0-ra változtatjuk, és minden folyamat meghibásodik közvetlenül azután, hogy P_i először hatott rá. Ekkor $\alpha_1 \overset{j}{\sim} \alpha_1'$, ezért P_j α_1'-ben ugyancsak 1-et választ. Ez megsérti az érvényességi feltételt, ami ellentmondás. □

A 7.25. tétel bizonyításának befejeződéséhez egyszerűen azt kell megmutatnunk, hogy abból a feltételből, hogy minden folyamat hat minden másik folyamatra, következik, hogy összesen legalább $2n-2$ üzenet van. Felhasználjuk a következő, kommunikációs mintákra vonatkozó lemmát.

7.27. lemma. . *Legyen γ tetszőleges kommunikációs minta. Ha γ-ban egy $m \geq 1$ folyamatot tartalmazó halmaz minden eleme hat a rendszernek mind az n folyamatára, akkor γ-ban legalább $n+m-2$ üzenet (hármas) van.*

Bizonyítás. m szerinti indukcióval.

Az indukció alapja: $m = 1$. Legyen P_i az a folyamat, amelyről feltettük, hogy mind az n folyamatra hat. Mivel P_i mind az n folyamatra hat, γ szükségképpen tartalmaz üzenetet mind az $n-1$, P_i-től különböző folyamathoz. Ez legalább $n-1$ üzenet, és éppen erre volt szükségünk.

Indukciós lépés. Feltesszük, hogy a lemma teljesül m-re, és megmutatjuk, hogy $(m+1)$-re is teljesül. Legyen I olyan halmaz, amely $m+1$ olyan folyamatot tartalmaz, amelyek mind az n folyamatra hatnak. Az általánosság megszorítása nélkül feltehetjük, hogy az első menetben legalább egy I-beli folyamat üzenetet küld valamelyik folyamatnak. Ugyanis ha ez nem igaz, akkor eltávolíthatjuk mindazokat a kezdeti meneteket, amelyekben I-beli folyamat nem küld üzenetet; a megmaradó kommunikációs mintában még mindig teljesül, hogy I minden folyamata hat mind az n folyamatra. Legyen P_i egy olyan I-beli folyamat, amely γ-ban az első menetben küld üzenetet.

Tekintsük azt a γ' kommunikációs mintát, amely annyiban különbözik γ-tól, hogy eltávolítottunk belőle egy olyan üzenetet, amelyet az első menetben P_i küldött. Akkor az $I \setminus \{P_i\}$ halmaz minden eleme hat γ'-ben mind az n folyamatra. Az indukciós feltevés szerint γ'-ben legalább $n+m-2$ üzenet van. Ezért γ legalább $n+m-1 = n+(m+1)-2$ üzenetet tartalmaz, és éppen ezt akartuk belátni. □

Most már befejezhetjük a 7.25. tétel bizonyítását.

7.25. tétel bizonyítása. A 7.26. lemma szerint tetszőlegesen választott P_i és P_j folyamatokra igaz, hogy P_i hat P_j-re a $minta(\alpha_1)$-ben. Ezért a 7.27. lemmából következik, hogy $minta(\alpha_1)$-ben legalább $2n-2$ üzenet van. □

7.4.. Megjegyzések a fejezethez

A szakirodalomban a k-megegyezési problémát k-halmaz megegyezési problémának hívják. A problémát először Chaudhuri [73] vetette fel, mint a korábban már alaposan elemzett egyszerű megegyezési feladat természetes általánosítását. A MinTerjed algoritmus Chaudhuri, Herlihy, Lynch és Tuttle [75] cikkéből származik és egy olyan algoritmuson alapul, melyet eredetileg Chaudhuri [73] javasolt. A k-megegyezésre vonatkozó alsó korlát bizonyításának gondolatmenete a [75,76,77] cikkekből származik. Az alsó korlát bizonyításának algebrai topológiai háttere megtalálható Spanier algebrai topológiáról szóló klasszikus könyvében [266]. A Sperner-lemmát eredetileg Sperner [267] igazolta és később Spanier [266] elemezte.

A közelítő megegyezésre vonatkozó rész Dolev, Lynch, Pinter, Stark és Weihl cikkéből [98] származik. Ezzel a problémával kapcsolatosak Fekete, [110,111] valamint Attiya, Lynch és Shavit [24] cikkei is. A véglegesítési problémáról, valamint a KétfázisúVéglegesítés és a HáromfázisúElfogadás algoritmusokról szóló anyag Bernstein, Hadzilacos és Goodman adatbázisok elméletével foglalkozó könyvéből [50] származik. Ez a könyv a miénknél részletesebben tárgyalja a protokollok gyakorlati megvalósítását, például a hibás folyamatok helyreállításával is foglalkozik. A véglegesítéshez szükséges üzenetek számára vonatkozó alsó korlát Dwork és Skeen eredménye [106].

7.5.. Gyakorlatok

7-1. Ha a k-megegyezés MinTerjed algoritmusa $\lfloor \frac{f}{k} \rfloor + 1$ helyett csak $\lfloor \frac{f}{k} \rfloor$ menetet fut, akkor legfeljebb hány döntést hozhatnak a hibátlanul működő folyamatok?

7-2. Adjunk jó alsó korlátot a 7.14. tétel bizonyításában szereplő *sorozat*(v) sorozat hosszára. Ehhez szükség van a sorozat explicit leírására.

7-3. Bizonyítsuk be, hogy az 1-futamok sorozatának összefésültje valóban k-futam. Ez magában foglalja annak belátását, hogy a futam definíciójában megkövetelt feltételek, valamint a jelekre vonatkozó feltételek is teljesülnek.

7-4. Bizonyítsuk be a 7.4. lemmát.

7-5. Bizonyítsuk be a 7.5. lemmát.

7-6. Bizonyítsuk be a 7.6. lemmát.

7-7. Bizonyítsuk be a 7.7. lemmát.

7-8. Legyen $n = 5,\ k = f = 2$ és $r = 1$.

(a) Adjuk meg részletesen az ezekhez a paraméterekhez tartozó Bermuda-háromszöget és annak címkézését k-futamokkal és folyamatindexekkel.

(b) Tekintsük azt az egyszerű A algoritmust, amely a következőképpen működik: minden folyamat egyszer cseréli az értékeket, és minden folyamat a legkisebb értéket választja azok közül, amelyeket kapott. Írjuk le a C_A Sperner-színezést.

(c) Meg tudunk adni az A algoritmushoz egy olyan kis szimplexet, amelyben három különböző döntési érték van?

7-9. Rögzítsünk tetszőleges $n,\ f$ és ϵ értéket, tetszőleges $w \in \mathbb{R}_0^+$-t és $r \in \mathbb{N}$-et úgy, hogy $n > 3f$ teljesüljön. Írjuk le a KonvergensKözelMegegyezés algoritmusnak egy, az adott $n,\ f$ és ϵ értékeknek megfelelő végrehajtási sorozatát, amelyben a hibátlan folyamatok kezdeti értékeit tartalmazó multihalmaz szélessége legfeljebb w, és amelyben a befejeződés r menetnél többet vesz igénybe.

7-10. *Kutatási kérdés.* Módosítsuk a KonvergensKözelMegegyezés algoritmust úgy, hogy az az idő, amíg minden folyamat dönt, korlátozva van $n,\ f,\ \epsilon$, valamint a hibátlan folyamatok kezdeti értékeit tartalmazó multihalmaz w szélességének valamilyen függvényével.

7-11. Tegyük fel, hogy a KonvergensKözelMegegyezés algoritmusban a folyamatok az *átlag*(*kiválasztás*(*csökkentés*(W))) helyett a következő értékek egyikét számítják ki:

(a) *átlag*(*választás*(W));

(b) *átlag*(*csökkentés*(W));

(c) *átlag*(W).

Így is megoldja az algoritmus a közelítő megegyezési problémát? Ha igen, miért igen, ha nem, miért nem?

7-12. Bizonyítsuk be, hogy a közelítő megegyezési feladat akkor és csak akkor oldható meg egy f bizánci hibát tűrő G hálózati gráfban, ha a következő két feltétel mindegyike teljesül:

(a) $n > 3f$;

(b) $össz\_függ(G) > 2f$.

7-13. Tervezzünk egy közelítő megegyezési algoritmust a megállási hibák esetére.

(a) Próbáljuk minimalizálni a megoldáshoz szükséges folyamatok és a hibák számának arányát.

(b) Próbáljuk minimalizálni a megoldáshoz szükséges menetek számát.

7-14. Fogalmazzuk meg a közelítő megegyezési feladat olyan változatát, amely rögzített r számú menetet használ, és amelyben ϵ nincs előre meghatározva. Minden folyamat egy valós értékkel indul, mint korábban. r menet után a folyamatoknak fel kell venniük a végső értéküket. Az érvényességi feltétel ugyanaz, mint korábban. A cél most a legjobb lehetséges megegyezés elérése. A megegyezés jóságát a hibátlan folyamatok végső értékeinek szélessége és a hibátlan folyamatok kezdeti értékeinek szélessége hányadosára vonatkozó felső korláttal jellemezzük.

(a) Milyen arányt ér el a KonvergensKözelMegegyezés ebben a modellben?

(b) Adjunk alsó korlátot az elérhető arányra, n, f és r függvényében. (*Útmutatás.* Alkalmazzuk a láncérvelés gondolatait ahhoz hasonlóan, ahogy a 6.33. tétel bizonyításában tettük. A kapott alsó és a felső korlátok valószínűleg nem egyeznek meg.)

7-15. Írjuk le a teljes HáromfázisúVéglegesítés algoritmus kódját (beleértve a befejeződési protokollt is).

7-16. Bizonyítsuk be részletesen, hogy a 7.22. lemma a HáromfázisúVéglegesítés tetszőleges számú menetére érvényes.

7-17. Írjuk le részletesen a HáromfázisúVéglegesítés algoritmus olyan módosítását, amely lehetővé teszi, hogy a folyamatok hibamentes esetben gyorsan döntsenek és leálljanak. Az algoritmus hibamentes esetben kevés menetet alkalmazzon és $\mathcal{O}(n)$ üzenetet. Bizonyítsuk be az algoritmus helyességét.

7-18. Tervezzünk olyan algoritmust a 6. fejezetben szereplő megállási megegyezési algoritmus stílusában, amely a véglegesítési problémát az erős befejeződési feltétellel megoldja. Igyekezzünk minimalizálni a menetek számát.

7-19. *Kutatási kérdés.* Tervezzünk egy algoritmust, amely a véglegesítési problémát az erős befejeződési feltétellel megoldja. Elérhető-e egyidejűleg, hogy a menetek száma legrosszabb esetben $n+k$ legyen (ahol k konstans), a hibamentes esetben a döntéshez és megálláshoz szükséges menetek száma egy kis konstans legyen és a hibamentes esetben alacsony legyen a kommunikációs bonyolultság?

7-20. Adjuk meg a 7.26. lemma részletes bizonyítását. Hol hibás a bizonyítás, ha α_1' létrehozásakor nem követeljük meg, hogy minden folyamat azonnal hibásodjon meg, miután P_i először hat rá, csak P_i kezdeti értékét változtatjuk meg 1-ről 0-ra?

7-21. Tervezzünk egy nem blokkoló véglegesítő algoritmust, amely a lehető legkevesebb üzenet használ hibamentes futamok esetében. Be tudjuk bizonyítani, hogy ez az üzenetszám optimális?

II. ASZINKRON ALGORITMUSOK

A második rész a 8–22. fejezeteket tartalmazza, azaz a könyv nagyobbik részét. Ezekben a fejezetekben lényegesen megváltozik a számítási modell: a 2–7. fejezetekben tárgyalt szinkron modelltől eljutunk az *aszinkron modellig*, amelyben az egyes komponensek tetszőleges sebességgel végezhetnek műveleteket.

A szinkron modellekhez hasonlóan, az aszinkron modelleket sem nehéz leírni. Új tulajdonság például az *elevenség* feltétele, amely azt írja elő, hogy a rendszer minden elemének esélye legyen lépések elvégzésére. Az aszinkron modelleket azonban az események sorrendjének bizonytalansága miatt nehezebb programozni. Az aszinkron modell kevesebb ismeretet tételez fel az időzítésről, mint általában az osztott rendszerek. Ezért az aszinkron rendszerekre tervezett algoritmusok általánosak és más rendszerekre átvihetőek, mivel biztosítva van, hogy tetszőleges időzítési tulajdonságokkal rendelkező rendszerekben is helyesen fognak működni.

A II. rész első fejezete, azaz a könyv 8. fejezete az aszinkron rendszerek egy általános modelljét mutatja be, a *bemeneti/kimeneti automata modellt*. Ez a fejezet, a könyv első olvasásakor kihagyható, később szükség szerint vissza lehet rá térni. A második rész további fejezetei két csoportot alkotnak: a 9–13. fejezetekben az *aszinkron közös memóriájú algoritmusokat*, a 14–22. fejezetekben pedig az *aszinkron hálózati algoritmusokat* tárgyaljuk.

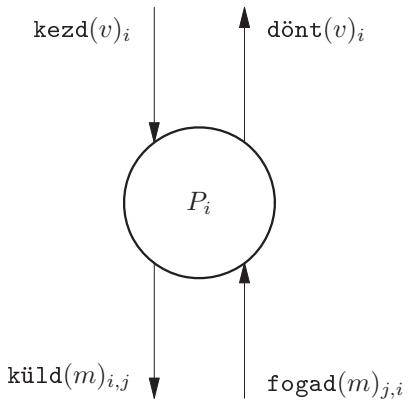
8. fejezet

Modellezés/II. Aszinkron rendszerek modelljei

A jelen fejezet célja, hogy az aszinkron számítások egy formális modelljét vezesse be, a *bemeneti/kimeneti* (b/k) *automata* modelljét. Ez egy nagyon általános modell, amellyel majdnem minden aszinkron konkurens rendszert le lehet írni, pl. az ebben a könyvben tárgyalt aszinkron közös memóriájú rendszert és aszinkron hálózati rendszert is. A b/k-automatának nagyon egyszerű a szerkezete, amely lehetővé teszi, hogy sokféle osztott rendszert modellezzen. Az alapszerkezet kiegészíthető úgy, hogy az automata képes legyen különféle aszinkron rendszertípusok leírására is. A modell segítségével pontosan leírhatjuk a rendszer egymással kapcsolatban álló és egymáshoz képest tetszőleges sebességgel működő összetevőit (úm. folyamatok vagy kommunikációs csatornák), és érvelhetünk velük kapcsolatban.

A b/k-automata definíciójával és működésének a leírásával kezdjük. Azután értelmezünk egy *összekapcsolási (kompozíciós) műveletet*, amelynek segítségével a b/k-automaták egy nagyobb, konkurens rendszert leíró automatává szerkeszthetők. Megmutatjuk, hogy ez a kompozíciós művelet rendelkezik mindazokkal a szép tulajdonságokkal, amelyeket elvárunk tőle. Azután bevezetjük a *pártatlanság* fontos fogalmát, amely azt fejezi ki, hogy a rendszer minden összetevője időnként képes lépések végrehajtására. A pártatlanság a rendszerelemek relatív sebességének korlátozását jelenti, amely kizárja azt, hogy egyes komponensek állandóan lépésképtelenek legyenek. Megmutatjuk, hogyan kapcsolódik a pártatlanság a kompozíciós művelethez. A fejezet többi része b/k-automatákkal modellezett rendszerekkel megoldható feladatok specifikációjának leírásakor alkalmazott módszereket mutat be. Bemutatunk néhány bizonyítási módszert is, amelyek azért hasznosak, mert segítségükkel igazolhatjuk, hogy a rendszerek valóban megoldják a feladatokat.

Ennek a fejezetnek az is célja, hogy hivatkozásul szolgáljon – nemcsak ebben, hanem más könyvekben is leírt – aszinkron rendszerek modellezésének módszereire. Nem szükséges, hogy ezt a fejezetet most alaposan átolvassák. Javasoljuk, hogy az olvasást kezdjék algoritmusokat leíró későbbi fejezetekkel (úm. a 10., 11., 12. és 15. fejezetek), majd térjenek vissza erre a fejezetre (akárcsak a 9. és 14.



8.1.. ábra. Egy folyamat b/k-automata.

fejezetre is), amikor szükségét érzik a formális megalapozásnak.

8.1.. b/k-automaták

Egy b/k-automata egy osztott rendszer komponensét modellezi, amely komponens kapcsolatban állhat a rendszer egyéb részeivel. A b/k-automata az állapotautomatának egy olyan egyszerű típusa, amelyben az átmenetekhez névvel ellátott *műveleteket* rendelünk. A műveletek lehetnek *bemeneti*, *kimeneti* műveletek (röviden: bemenetek, ill. kimenetek) vagy *belső* műveletek. A bemenetek és kimenetek az automata környezetével való kapcsolattartást szolgálják, míg a belső műveletek csak az automata számára érzékelhetők. Feltételezzük, hogy a bemeneti műveleteket az automata nem befolyásolja – ezek kintről érkeznek –, míg az automata maga dönt a belső műveletek és a kimenetek végrehajtásáról.

A b/k-automata egy tipikus példája az aszinkron osztott rendszer egy folyamata. Egy tipikus folyamatautomata kapcsolata a környezetével a 8.1. ábrán látható. A P_i automatát egy körrel ábrázoltuk, a bemenő éleket a bemeneti műveletekkel, míg a kimenő éleket a kimeneti műveleteket címkéztük. A belső műveletek nem szerepelnek az ábrán. Az ábrán látható automata a külvilágtól `kezd`$(v)_i$ formában kapja a bemeneteket, amelyek a bemeneti v érték fogadását jelentik. A kimeneteket `dönt`$(v)_i$ alakban szolgáltatja, és ez egy v-n alapuló döntést jelent. Hogy döntést hozhasson, a P_i folyamat kapcsolatba léphet más folyamatokkal egy üzenetkezelő rendszeren keresztül. Az automata kapcsolata az üzenetkezelő rendszerrel `küld`$(m)_{i,j}$ alakú kimeneti műveletekből áll (amelyek jelentése: a P_i folyamat m tartalmú üzenetet küld a P_j folyamatnak), valamint `fogad`$(m)_{j,i}$ alakú bemeneti műveletekből (amelyek jelentése: a P_i folyamat m tartalmú üzenetet fogad a P_j folyamattól). Amikor az automata végrehajtja az ábrán feltüntetett műveletek bármelyikét (vagy bármely belső műveletét), akkor állapotot is válthat.

8.2.. ábra. Egy csatorna b/k-automata.

A tipikus b/k-automata egy másik példája a FIFO (first in first out, azaz *elsőnek be, elsőnek ki*) rendszerű üzenetcsatorna. Egy $C_{i,j}$-vel jelölt tipikus üzenetcsatorna a 8.2. ábrán látható. A bemeneti műveletek $\texttt{küld}(m)_{i,j}$ alakúak, a kimenetiek pedig $\texttt{fogad}(m)_{i,j}$ alakúak. Amikor egy osztott rendszert írunk le b/k-automatákkal, akkor általában folyamatautomaták és csatornaautomaták összekapcsolása történik, oly módon, hogy egyik automata kimeneteit illesztjük a másik automata azonos nevű bemeneteivel. Így a P_i folyamat szolgáltatta $\texttt{küld}_{i,j}$ kimenet illeszkedik (azaz egyidejűleg hajtódik végre) a $C_{i,j}$ csatorna $\texttt{küld}_{i,j}$ bemenetével. Fontosnak tartjuk megjegyezni, hogy a különböző műveletek végrehajtására előre meg nem határozott sorrendben kerül sor valamely időpontban. Ez különbözik a szinkron rendszerektől, amelyekben a folyamatok egyszerre küldenek és fogadnak üzeneteket a feldolgozás minden adott pillanatában.

Az első dolog, amit egy b/k-automata esetében megadunk, az a „lenyomata", amely nem más, mint a bemeneteinek, a kimeneteinek és a belső műveleteinek a leírása. Feltesszük, hogy van egy *művelet-alaphalmazunk*. Az S *lenyomat* egy olyan hármas, amely a következő három, páronként diszjunkt művelethalmazból áll: $be(S)$, a *bemeneti műveletek* halmaza, $ki(S)$, a *kimeneti műveletek* halmaza, valamint $belső(S)$, a *belső műveletek* halmaza. Értelmezzük a következőket is: a $külső(S)$-sel jelölt *külső műveleteket*, ahol $külső(S) = be(S) \cup ki(S)$, a $helyi(S)$-sel jelölt *helyileg ellenőrzött műveleteket*, ahol $helyi(S) = ki(S) \cup belső(S)$, valamint a $művelethalmaz(S)$-sel jelölt összes műveletet. *Külső lenyomatnak* nevezzük a $(be(S), ki(S), \emptyset)$ lenyomatot, melynek jelölése: $külső\_lenyomat(S)$. Gyakran hivatkozunk majd rá mint *külső felületre*.

Egy A-val jelölt *b/k-automata*, amelyet röviden csak *automatának* nevezünk majd, a következő öt részből áll:

- $lenyomat(A)$, az automata lenyomata;
- $állapotok(A)$, egy nem feltétlenül véges halmaz, amely az *állapotok* halmaza;
- $kezdő(A)$, az állapotok egy *kezdőállapotoknak* nevezett nem üres halmaza;
- $átmenetek(A)$, amely egy következőképpen értelmezett *átmenetreláció*: $átmenetek(A) \subseteq állapotok(A) \times művelethalmaz(lenyomat(A)) \times állapotok(A)$, amely rendelkezik azzal a tulajdonsággal, hogy bármely s állapotra és bármely π bemeneti műveletre (kötelező módon) létezik egy $(s, \pi, s') \in átmenetek(A)$ átmenet;
- $taszkok(A)$, egy *taszkpartícionálás*, amely a $helyi(lenyomat(A))$ halmazon értelmezett olyan ekvivalenciareláció, amelynek legfeljebb megszámlálható ekvivalenciaosztálya van.

A következőkben rövidítéseket használunk, úm. $művelethalmaz(lenyomat(A))$

helyett *művelethalmaz*(A), *be*(*lenyomat*(A)) helyett *be*(A) stb. Az A automatát *zártnak* nevezzük, ha nincs bemenete, vagyis ha *be*$(A) = \emptyset$.

Ez az definíció valamelyest hasonlít a *folyamat* 2. fejezetbeli, a szinkron hálózati modell esetében adott definíciójára. Mégis, a lenyomat általánosabb művelettípusok használatát teszi lehetővé, mint a szinkron modellbeli üzenetküldő és üzenetfogadó műveletek. Itt is, akárcsak a szinkron modell esetében, az állapothalmaz nem feltétlenül véges. Ez az általánosság fontos, mert lehetővé teszi számunkra, hogy olyan rendszereket is modellezünk, amelyeknek nemkorlátos adatstruktúrájuk van, úm. számlálók és nemkorlátos kapacitású sorok. Akárcsak a szinkron esetben, itt is megengedjük, hogy több kezdőállapot legyen, így bizonyos bemeneti információkat belefoglalhatunk a kezdőállapotokba.

Az *átmenetek*(A) reláció egy (s, π, s') elemét az A *átmenetének* vagy *lépésének* nevezünk. Attól függően, hogy π bemeneti, kimeneti stb. művelet, az (s, π, s') átmenetet is *bemeneti átmenetnek*, *kimeneti átmenetnek* stb. nevezzük. A szinkron modellel ellentétben itt az átmenetekhez nem feltétlenül üzenetfogadások kapcsolódnak, hanem bármilyen művelet kapcsolódhat hozzájuk.

Ha A-ban egy adott s állapotra és π műveletre van (s, π, s') átmenet, akkor azt mondjuk, hogy π *megengedett* s-ben. Mivel minden bemeneti műveletnek megengedettnek kell lennie bármely állapotban, az automatát *megengedett bemenetűnek* nevezzük. Ez azt jelenti, hogy az automata semmiképpen sem tudja megakadályozni a bemeneti műveletek bekövetkezését. Ez a feltevés azt jelenti például, hogy egy folyamatnak valamilyen módon mindig képesnek kell lennie bármilyen üzenet kezelésére. Ha az s állapotban csak bemeneti műveletek megengedettek, akkor s-t *tétlen* állapotnak nevezzük.

Azt gondolhatjuk, hogy a megengedett bemenetűség túlságosan erős megkötés egy általános modell esetében, mivel sok rendszerkomponenst eleve úgy terveznek, hogy bizonyos bemenetek bekövetkezésére adott időpontokban *várjanak*. Például, egy automata, melyet erőforrás-hozzárendelés modellezésére terveztek (mint a 11. fejezetben), elvárhatja, hogy egy felhasználó ne nyújtson be két kérést közvetlenül egymás után, mielőtt a rendszer felismerte volna az elsőt. Van azonban más módja is annak, hogy a környezetre vonatkozó ilyen megkötéseket modellezzük, anélkül, hogy letiltanánk egy bemenet végrehajtását. Például, az erőforrás-hozzárendelés esetében azt mondhatjuk, hogy nem feltételezzük, hogy a környezet egy másik kérést is továbbít, mielőtt egy előzőre választ kapott volna, de nem is korlátozzuk az ilyen váratlan bemenet esetében a rendszer viselkedését. Vagy azt is megtehetjük, hogy a rendszert úgy tervezzük, hogy felfedezze a váratlan bemenetet, és hibaüzenetet adjon válaszként.

Két fő előnye van a megengedett bemenetűségnek. Először, a rendszerkomponensek tervezésénél nagyon sok hibaforrás rejlik abban, hogy nem határozzuk meg, hogy a váratlanul megjelenő bemeneteket hogyan kezelje a rendszer. Olyan modell használata, amely megköveteli tetszőleges bemenet figyelembevételét, segít az ilyen hibák kiküszöbölésében. Másodsorban, a megengedett bemenetűség alkalmazásával a modell alapelmélete jól működik. Ez a tulajdonság lehetővé teszi például, hogy egy automatának egyszerűen a külső műveletek sorozata alapján írjuk le a külső viselkedését. (A 8.4. tétel példa arra, hogy egy alapvető állítás dől meg, ha nem tételezzük fel ezt a tulajdonságot.)

Az automata definíciójában az ötödik elem, a *taszkok*(A)-val jelölt taszkpartícionálás, úgy is felfogható, mint a „taszkoknak" vagy „vezérlési szálaknak" az absztrakt leírása az automatán belül. A partícionálást arra használjuk, hogy megadjuk az automata működésére vonatkozó pártatlansági feltételeket. Ezek azok a feltételek, amelyek biztosítják, hogy az automata, működése során folyamatosan pártatlanul lehetőséget ad minden taszknak. Ez hasznos lehet az egynél több tevékenységet végző komponensek modellezésénél – ilyen tevékenység például: részt venni egy folyamatban lévő algoritmusban, ugyanakkor időnként információt szolgáltatni a környezet számára. Szintén hasznos akkor is, amikor több automatát összekapcsolunk, hogy egy egész rendszert leíró, nagyobb automatát eredményezzen. A partícionálás arra szolgál, hogy előírja, hogy az összekapcsolásban szereplő automaták mindegyike lép újra és újra. A partícionálás egy másik alkalmazására kerül sor az aszinkron, közös memóriájú algoritmusok modellezésénél, amint látni fogjuk a 9. fejezetben. Általában csak *taszkokként* fogunk hivatkozni a taszkpartícionálás osztályaira.

Ha azt mondjuk, hogy egy C taszk *megengedett* az s állapotban, ez egyszerűen csak azt jelenti, hogy a C valamely műveletei megengedettek az s állapotban.

A következőkben példát adunk egy egyszerű b/k-automatára. Itt, és a példáink legtöbbjében, az átmenetrelációt *előfeltétel/hatás* formájában adjuk meg. Ez a módszer mindazon átmeneteket gyűjti egyetlen kódrészbe, amelyek adott típusú műveleteknek felelnek meg. A kód a megelőző s állapotra vonatkozó predikátum formájában adja meg azokat a feltételeket, amelyek teljesülése esetében a művelet végrehajtása engedélyezett. Azután leírja azokat a változásokat, amelyek a műveletek eredményeként megjelennek, méghozzá egy egyszerű program formájában, amely az s állapotot s' állapottá alakítja. Az egész kódrész oszthatatlanul (atomi akció formájában) hajtódik végre mint egyetlen átmenet. Az átmeneteknek a nekik megfelelő műveletek szerinti csoportosítása tömör kódot eredményez, mivel azok az átmenetek, amelyek minden műveletet magukban foglalnak, az állapotok tipikusan kis részét használják.

Az előfeltétel/hatás stílusában írott programok általában egyszerű vezérlő szerkezeteket használnak. Ez lehetővé teszi, hogy az ilyen program fordítása b/k-automatára nagyon könnyű legyen, és megkönnyíti, hogy formálisan érveljünk az automatáról.

8.1.1. példa. Csatorna b/k-automata

Példaként b/k-automatára, tekintsük a $C_{i,j}$ kommunikációs csatorna-automatát. Legyen M egy rögzített üzenetábécé. Először megadjuk a *lenyomat*$(C_{i,j})$ lenyomatot. Itt és a következőkben bármikor, ha nem adunk meg egy lenyomatkomponenst (legtöbbször a belső műveleteket), akkor ezt a művelethalmazt üresnek tekintjük.

Lenyomat:

Bemeneti:	Kimeneti:
$\texttt{küld}(m)_{i,j}$, $m \in M$	$\texttt{fogad}(m)_{i,j}$, $m \in M$

Az *állapotok*$(C_{i,j})$ állapotokat és a *kezdő*$(C_{i,j})$ kezdőállapotokat legjobban állapotváltozók listája és kezdő értéke segítségével írhatjuk le,

akárcsak a szinkron esetben.

Állapotok:
sor, egy FIFO sor, melynek elemei M-ből valók, kezdetben üres

$C_{i,j}$ átmeneteit a következőképpen írjuk le.

Átmenetek:

`küld`$(m)_{i,j}$
Hatás:
tegyük be m-et *sor*-ba

`fogad`$(m)_{i,j}$
Előfeltétel:
m első a *sor*-ban
Hatás:
töröljük *sor* első elemét

A fenti kód önmagáért beszél: a `küld` művelet bármikor megjelenhet és hatására az üzenet a *sor* végére kerül, míg a `fogad` csak akkor jelenhet meg, ha a szóbanforgó üzenet a *sor* elején van, és a művelet hatására törlődik.

A *taszkok*$(C_{i,j})$ taszkpartíció a `fogad` műveleteket egyetlen taszkba gyűjti össze. Vagyis, az üzenetek fogadása (azaz átadása) egyetlen taszkként fogható fel.

Taszkok:
$\{$`fogad`$(m)_{i,j} : m \in M\}$

8.1.2. példa. Folyamat b/k-automata

Második példaként b/k-automatára, tekintsük a P_i folyamatautomatát. Az automata külső felületét alább írjuk le. Jelöljön V egy rögzített értékhalmazt, *null* egy nem V-beli különleges értéket, f pedig egy rögzített függvényt: $f : V^n \to V$.

Lenyomat:

Bemeneti:
`kezd`$(\mathrm{v})_i, v \in V$
`fogad`$(v)_{j,i}, v \in V, 1 \leq j \leq n, j \neq i$

Kimeneti:
`dönt(v)`$_i$ $v \in V$
`küld(v)`$_{i,j}, v \in V, 1 \leq j \leq n, j \neq i$

Az állapotok és a kezdőállapotok a következők.

Állapotok:
érték, az $\{1, 2, \ldots, n\}$ halmaz elemeivel indexelt vektor, melynek elemei $V \cup \{null\}$ halmazból valók, kezdetben a vektor minden eleme *null*.

Az átmeneteket a következőképpen írjuk le.

Átmenetek:

`kezd`$(v)_i, v \in V$
Hatás:
érték$(i) := v$

`fogad`$(v)_{j,i}, v \in V$
Hatás:
érték$(j) := v$

`küld`$(\mathrm{v})_{i,j}, v \in V$
Előfeltétel:
érték$(i) = v$
Hatás:
semmi (*nincs állapotváltozás*)

`dönt`$(v)_i, v \in V$
Előfeltétel:
$\forall j, 1 \leq j \leq n:$
érték$(j) \neq null$
$v = f($*érték*$(1), \ldots,$*érték*$(n))$
Hatás:
semmi

Tehát a `kezd` művelet hatására P_i az *érték* vektor saját (azaz i-edik) elemét tölti fel a megadott értékkel, ezzel szemben a `fogad` hatására egy másik elemet tölt fel. Ezeket az értékéket tetszőlegesen sokszor lehet módosítani a `kezd` és `fogad` műveletek többszöri alkalmazásával. P_i akárhányszor elküldheti a saját értékét bármelyik csatornán. P_i tetszőleges számú döntést is hozhat, ha újra és újra alkalmazza az f függvényt saját vektorára.

A *taszkok*(P_i) taszkpartícionálás n taszkot tartalmaz: egyet-egyet minden $j \neq i$-re az összes $\texttt{küld}_{i,j}$ műveletre és egyet az összes `dönt` műveletre. Tehát, az egyes csatornákon az üzenetküldés egyetlen taszknak számít. Ugyancsak egy taszknak számít a döntések bejelentése is.

Taszkok:
$\forall j,\ j \neq i:\ \{\texttt{küld}(v)_{i,j} : v \in V\}$
$\{\texttt{dönt}(v)_i : v \in V\}$

Most pedig leírjuk, hogyan működik egy A b/k-automata. Az A automata egy *végrehajtási sorozatrészlete* vagy egy véges $s_0, \pi_1, s_1, \pi_2, \ldots, \pi_r, s_r$ sorozat vagy egy végtelen $s_0, \pi_1, s_1, \pi_2, \ldots, \pi_r, s_r, \ldots$ sorozat, amelyben váltakoznak az A állapotai és műveletei, és bármely $k \geq 0$ értékre $(s_k, \pi_{k+1}, s_{k+1})$ az automata egy átmenete. Amennyiben a végrehajtási sorozatrészlet véges, akkor állapotban kell végződnie. A kezdőállapottal kezdődő végrehajtási sorozatrészlet neve *végrehajtási sorozat*. Az A végrehajtási sorozatainak a halmazát *végrehajtások*(A) jelöli. Az A egy állapota *elérhető*, ha az A egy véges végrehajtási sorozatának a végállapota.

Ha α az A automata egy véges végrehajtási sorozatrészlete és α' pedig A-nak egy olyan végrehajtási sorozatrészlete, amelynek első eleme azonos α utolsó elemével, akkor az $\alpha \cdot \alpha'$ sorozat az α és α' sorozatok összefűzése, kihagyva az α utolsó elemét, amely kétszer jelenne meg. Nyilvánvaló, hogy az $\alpha \cdot \alpha'$ sorozat is végrehajtási sorozatrészlete A-nak.

Néha csak a b/k-automata külső viselkedése érdekel bennünket. Így, az A automata α végrehajtási sorozatának a *története*, amelyet *történet*(α)-val jelölünk, az α külső műveleteiből álló részsorozat. Azt mondjuk, hogy β az A automata

története, ha β az A egy végrehajtási sorozatának a története. Az A automata történeteinek a halmazát *történetek*(A)-val jelöljük.

8.1.3. példa. Végrehajtási sorozatok

A következőkben bemutatjuk a 8.1.1. példában leírt $C_{i,j}$ automata három végrehajtási sorozatát (feltételezve, hogy az M üzenetábécé egyenlő az $\{1, 2\}$ halmazzal). Itt az állapotokat a *sor*-beli sorozatok szögletes zárójelbe tételével jelöljük. Az üres sorozat jele: λ.

$[\lambda]$, $\texttt{küld}(1)_{i,j}$, $[1]$, $\texttt{fogad}(1)_{i,j}$, $[\lambda]$, $\texttt{küld}(2)_{i,j}$, $[2]$, $\texttt{fogad}(2)_{i,j}$, $[\lambda]$

$[\lambda]$, $\texttt{küld}(1)_{i,j}$, $[1]$, $\texttt{fogad}(1)_{i,j}$, $[\lambda]$, $\texttt{küld}(2)_{i,j}$, $[2]$

$[\lambda]$, $\texttt{küld}(1)_{i,j}$, $[1]$, $\texttt{küld}(1)_{i,j}$, $[11]$, $\texttt{küld}(1)_{i,j}$, $[111]$, ...

Az utolsó két végrehajtási sorozat is elfogadható, annak ellenére, hogy olyan üzeneteket tartalmaz, amelyeket az automata elküld, de nem fogadnak soha. Ez azért van, mert semmi olyan kikötést nem teszünk a végrehajtási sorozatra, hogy csak a megengedett műveleteknek kell bekövetkezniük. A 8.3. alfejezetben bevezetjük a pártatlansági követelményeket, amelyek lehetővé teszik az ilyen kikötések megfogalmazását.

8.2.. Műveletek automatákkal

Ebben a szakaszban értelmezzük a b/k-automata következő műveleteit: *összekapcsolás* (*kompozíció*) és a *kimeneti műveletek elrejtése*.

8.2.1.. Összekapcsolás

Az összekapcsolás művelete lehetővé teszi, hogy egy komplex rendszert leíró automatát a rendszer egyedi összetevőit jellemző automaták összekapcsolásából kapjunk meg. Az összekapcsolás azonosítja egymással az ugyanolyan nevű műveleteket a különböző komponensautomatákban. Amikor egy komponensautomata végrehajt egy olyan lépést, amelyben szerepel π, akkor minden komponensautomata, amely tartalmazza lenyomatában a π-t, végrehajtja ugyanazt a lépést.

Az összekapcsolásban szereplő automatákra bizonyos megszorításokat teszünk. Először, mivel egy A automata belső műveleteinek egy másik B automata számára láthatatlanoknak kell lenniük, csak akkor engedjük meg, hogy egy A automatát egy B-vel összekapcsoljunk, ha az A automata belső műveletei különböznek a B műveleteitől. (Különben, az A egy belső műveletének a végrehajtási sorozata B-t is egy lépés megtételére kényszerítené.) Másodsorban, mivel az összekapcsolási műveletnek szép tulajdonságokkal kell rendelkeznie (mint az alábbi 8.4. tételben), azt a kikötést tesszük, hogy legfeljebb egy komponensautomata „ellenőrzi" bármely művelet végrehajtását, ezért letiltjuk az A és B automaták összekapcsolását, ha azok kimeneti műveletei nem diszjunktak. Harmadsorban,

nem zárjuk ki annak a lehetőségét, hogy megszámlálhatóan végtelen sok automatát kapcsoljunk össze, de ebben az esetben megköveteljük, hogy bármely művelet legfeljebb véges számú komponensautomata művelete legyen. Ez utóbbi kikötés nélkül például a 8.3. tétel nem igaz.

Miért nem zárjuk ki egyszerűen a végtelen sok automata összekapcsolását? Végül is, a valódi számítógéprendszerek véges számú komponensből (számítógépekből, üzenetcsatornákból stb.) állnak. A legfőbb ok az, hogy a b/k-automatákat *logikai rendszerek* és *fizikai rendszerek* modellezésére egyaránt használjuk. Egy logikai rendszer igen nagyszámú logikai komponensből állhat, melyet kevesebb komponensű fizikai rendszerrel valósítunk meg. Tulajdonképpen, bizonyos logikai rendszerek képesek arra, hogy komponenseket hozzanak létre dinamikusan, végrehajtás közben – akár végtelen sok komponenst is létrehozhatnak egy végtelen végrehajtási sorozatban. (Például, adatbázisrendszerekben lehetőség van arra, hogy futás közben új tranzakciópéldányok jöjjenek létre.) Komponens létrehozásának modellezése b/k-automaták segítségével úgy történhet, hogy elképzeljük, hogy már kezdetben jelen van minden elképzelhető komponens, amely valaha is létrejöhet, és különleges *ébresztő* bemeneti műveletek hatására felébrednek, amikor létre kellene jönniük. Ezzel a trükkel a szokásos összekapcsolási művelet megfelelő módon írja le a dinamikusan létrehozott komponensek kapcsolatát a rendszer többi részével. De meg kell engedni, hogy végtelen sok komponenst is összekapcsolhassunk.

Formálisan, a lenyomatok egy megszámlálható $\{S_i\}_{i\in I}$ halmazát *kompatibilisnek* nevezzük, ha minden $i, j \in I, i \neq j$ értékekre a következő feltételek mindegyike teljesül.

1. $\mathit{belső}(S_i) \cap \mathit{műveleth\acute{a}lmaz}(S_j) = \emptyset$;
2. $\mathit{ki}(S_i) \cap \mathit{ki}(S_j) = \emptyset$;
3. egyetlen művelet sem eleme végtelen sok $\mathit{műveleth\acute{a}lmaz}(S_i)$ halmaznak.

Az automaták egy halmazát *kompatibilisnek* nevezzük, ha a lenyomatuk halmaza kompatibilis.

Amikor automaták egy halmazára alkalmazzuk az összekapcsolást, a komponensek kimeneti műveletei az összetett automata kimeneti műveletei lesznek, a komponensek belső műveletei az összetett automata belső műveletei lesznek, míg azok a műveletek, amelyek bizonyos komponensek bemeneti műveletei, de egyetlen komponensnek sem kimeneti műveletei, az összetett automata bemeneti műveletei lesznek. Formálisan a megszámlálható számosságú kompatibilis $\{S_i\}_{i\in I}$ lenyomathalmaz $S = \Pi_{i\in I} S_i$ *összekapcsolása* a következő elemekből álló lenyomat.

- $\mathit{ki}(S) = \cup_{i\in I}\ \mathit{ki}(S_i)$.
- $\mathit{belső}(S) = \cup_{i\in I}\ \mathit{belső}(S_i)$.
- $\mathit{be}(S) = \cup_{i\in I}\ \mathit{be}(S_i) - \cup_{i\in I} \mathit{ki}(S_i)$.

Most már értelmezhetjük az $\{A_i\}_{i\in I}$ megszámlálható számosságú, kompatibilis halmaz b/k-automatáinak $A = \Pi_{i\in I} A_i$ összekapcsolását. Az összetett automata a következő.[1]

[1] A Π jelölés a $\mathit{kezdő}(A)$ és $\mathit{állapotok}(A)$ esetében a megszokott Descartes-szorzat, míg a

- $lenyomat(A) = \Pi_{i \in I}\ lenyomat(A_i)$;
- $állapotok(A) = \Pi_{i \in I}\ állapotok(A_i)$;
- $kezdő(A) = \Pi_{i \in I}\ kezdő(A_i)$;
- $átmenetek(A)$ az (s, π, s') hármasok halmaza úgy, hogy minden $i \in I$ értékre, ha $\pi \in művelethalmaz(A_i)$, akkor $(s_i, \pi, s'_i) \in átmenetek(A_i)$; egyébként $s_i = s'_i$;
- $taszkok(A) = \cup_{i \in I}\ taszkok(A_i)$.

Tehát, az összetett automata állapotai és kezdőállapotai olyan vektorok, amelyek elemei a komponensautomaták állapotai, illetve kezdőállapotai. Az összetett automata átmeneteit úgy kapjuk meg, hogy azon komponensautomaták, amelyek lenyomatában jelen van egy adott π művelet, egyszerre részt vesznek azokban a lépésekben, amelyek a π művelettel kapcsolatosak, a többi komponensautomata pedig nem csinál semmit. Az összetett automata helyileg ellenőrzött műveleteinek a taszkpartícionálása a komponensek taszkparticíonálásának egyesítéséből keletkezik, azaz minden egyes komponensautomata minden ekvivalenciaosztálya az összetett automata egy-egy ekvivalenciaosztálya lesz. Ez azt jelenti, hogy az egyes komponensek taszkstruktúrája megőrződik az összekapcsoláskor. Megjegyezzük, hogy mivel az A_i automaták megengedett bemenetűek, az lesz az összetett automata is. Ebből következik, hogy az $\Pi_{i \in I} A_i$ automata ténylegesen b/k-automata.

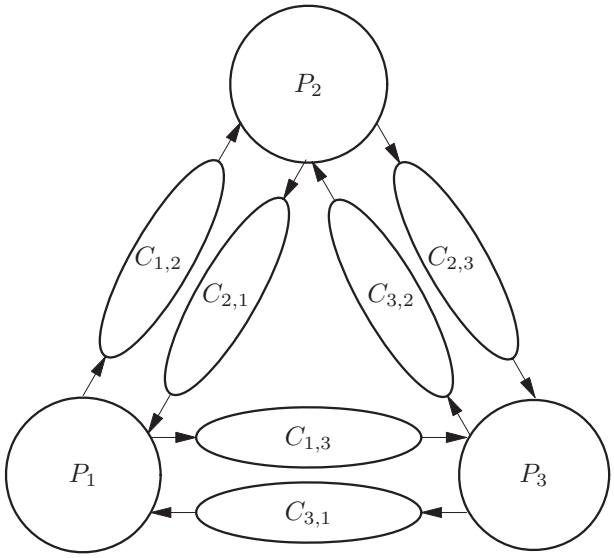
Ha az I halmaz véges, az összekapcsolási műveletre a $\times$ jelet is szoktuk használni. Így például, ha $I = \{1, 2, \ldots, n\}$, akkor $\Pi_{i \in I} A_i$ helyett néha az $A_1 \times A_2 \times \ldots \times A_n$ jelölést használjuk.

Megjegyezzük, hogy ha egy π művelet egy komponens kimenete és egy másik komponens bemenete, akkor az összetett automatában kimeneti műveletnek tekintjük, és nem belsőnek. Ez azért van így, mert szeretnénk lehetővé tenni, hogy π egy későbbi kommunikációban részt vehessen. Például, tegyük fel, hogy π kimeneti művelete az A automatának, ugyanakkor bemeneti művelete a B és C automatáknak. Tehát, π lényegében üzenetszórás A-tól B és C felé az $A \times B \times C$ összetett automatában. Szeretnénk modulárisan felfogni az összekapcsolást, azaz először megvalósítani az $A \times B$ összekapcsolást, majd az eredményt összekapcsolni C-vel. Annak alapján, ahogy az összekapcsolást értelmeztük, az $A \times B \times C$ izomorf az $(A \times B) \times C$ automatával, amikor előbb összekapcsoljuk A és B-t, majd az eredményt összekapcsoljuk C-vel. Ellenben, ha π-t belső műveletnek tekintenénk az $A \times B$ automatában, akkor ez a modularitás nem létezne, és $A \times B$-t nem lehetne összekapcsolni C-vel, mivel az első kompatibilitási tulajdonság nem állna fenn.

Lehetőség van arra, hogy a komponensek közti kommunikációban résztvevő műveleteket „elrejtsük", hogy ezzel megakadályozzuk, hogy későbbi kommunikációban használhatók legyenek. Ezt a 8.2.2. szakaszban értelmezett elrejtési művelet segítségével valósítjuk meg.

8.2.1. példa. Automaták összekapcsolása

$lenyomat(A)$ esetében az éppen értelmezett összekapcsolás. Az s_i az s állapotvektor i-edik komponensét jelöli.



8.3.. ábra. P_i-k és $C_{i,j}$-k összekapcsolása.

Legyen $I = \{1, 2, \ldots, n\}$ egy rögzített indexhalmaz és legyen A a 8.1.2. példában szereplő összes P_i, $i \in I$ folyamatautomata és a 8.1.1. példában szereplő összes $C_{i,j}$, $i, j \in I$ csatornaautomata összekapcsolása. Hogy az összekapcsolás megvalósítható legyen, tételezzük fel, hogy a csatornaautomaták M üzenetábécéje tartalmazza a folyamatautomaták V értékhalmazát. A 8.3. ábra az $n = 3$ speciális eset szerkezetét mutatja be. Az eredmény egyetlen automata, amely egy osztott rendszert ábrázol. A rendszer egy állapota a következőkből áll: minden folyamat egy állapota (folyamatonként egy értékvektor) és minden csatorna egy állapota (úton lévő üzenetek sora). A rendszer mindegyik átmenete magában foglalja a következők valamelyikét:

1. egy `kezd`$(v)_i$ bemeneti művelet, amelyik elhelyezi a v értéket a P_i *érték*(i) változójában, amelyet *érték*$(i)_i$-vel jelölünk[2];
2. egy `küld`$(v)_{i,j}$ kimeneti művelet, amelynek hatására a P_i-nek a *érték*$(i)_i$ értéke bekerül a $C_{i,j}$ csatornába;
3. egy `fogad`$(v)_{i,j}$ kimeneti művelet, amelynek hatására $C_{i,j}$ első üzenete törlődik, és ezzel egy időben bekerül a P_j *érték*$(i)_j$ változójába;
4. egy `dönt`$(v)_i$ kimeneti művelet, amelynek hatására P_i „bejelenti” az aktuálisan kiszámított értékét.

Az összekapcsolás egy mintatörténete $n = 2$-re, ahol a V értékhalmaz $\mathbb{N}$ és az f függvény összeadás, a következő:

[2]Akárcsak a szinkron modellt leíró fejezetben, itt is a változó indexe annak a folyamatnak az indexe, amelyhez a változó tartozik.

$$\texttt{kezd}(2)_1,\ \texttt{kezd}(1)_2,\ \texttt{küld}(2)_{1,2},\ \texttt{fogad}(2)_{1,2},\ \texttt{küld}(1)_{2,1},$$
$$\texttt{fogad}(1)_{2,1},\ \texttt{kezd}(4)_1,\ \texttt{kezd}(0)_2,\ \texttt{dönt}(5)_1,\ \texttt{dönt}(2)_2$$

Ha a fenti történetet használjuk, az egyetlen elérhető rendszerállapotban a P_1 *érték* vektora $(4,1)$, míg a P_2-é $(2,0)$, és mindkét csatorna üres. Természetesen, az összetett rendszer végrehajtási sorzataiban sok más történet is lehetséges.

Ezt a szakaszt három alapvető eredménnyel zárjuk, amelyek kapcsolatot teremtenek az összetett automata és komponensei végrehajtási sorzatai és történetei között. Az első szerint az összetett automata végrehajtási sorozata vagy története „vetítésével" a komponensek végrehajtási sorozatához vagy történetéhez jutunk. Ha $\alpha = s_0, \pi_1, s_1, \ldots$, az A egy végrehajtási sorozata, akkor jelöljük $\alpha|A_i$-vel azt a sorozatot, amelyet α-ból úgy kapunk, hogy elhagyjuk az összes olyan π_r, s_r párt, amelyre π_r nem művelete A_i-nak, és helyettesítjük az összes megmaradt s_r-eket $(s_r)_i$-vel, azaz az s_r állapotnak az A_i automatabeli részével. Ugyanígy, ha β az A egy története (vagy általánosabban, tetszőleges műveletsorozat), akkor $\beta|A_i$ a β egy olyan sorozatrészlete, amely az A_i összes olyan műveletét tartalmazza, amelyek β-ból valók. Szintén a | jelet használjuk egy β műveletsorozat olyan sorozatrészlete jelölésére, amely a β azon műveleteit tartalmazza, amelyek egy adott halmazból valók.

8.1. tétel. . *Legyen $\{A_i\}_{i\in I}$ egy kompatibilis automatahalmaz, és legyen $A = \Pi_{i\in I}A_i$.*

1. *Ha $\alpha \in$ végrehajtások(A), akkor $\alpha|A_i \in$ végrehajtások(A_i) minden $i \in I$ értékre.*
2. *Ha $\beta \in$ történetek(A), akkor $\beta|A_i \in$ történetek(A_i) minden $i \in I$ értékre.*

Bizonyítás. A bizonyítást meghagyjuk gyakorlatnak (lásd 8-2. gyakorlat). □

A másik két tétel az elsőnek a fordítottja. A következő tétel szerint, bizonyos feltételek mellett, a komponensautomaták végrehajtási sorozatai „összeilleszthetők" úgy, hogy az összetett automata egy végrehajtási sorozatát adják.

8.2. tétel. . *Legyen $\{A_i\}_{i\in I}$ egy kompatibilis automatahalmaz, és legyen $A = \Pi_{i\in I}A_i$. Tételezzük fel, hogy α_i az A_i egy végrehajtási sorozata minden $i \in I$-re és β egy műveletsorozat a külső(A)-ból úgy, hogy $\beta|A_i =$történet(α_i) minden $i \in I$-re. Ekkor létezik A-nak egy olyan α végrehajtási sorozata, amelyre $\beta =$történet(α) és $\alpha_i = \alpha|A_i$ minden $i \in I$ értékre.*

Bizonyítás. A bizonyítást meghagyjuk gyakorlatnak (lásd 8-2. gyakorlat). □

A harmadik tétel azt mondja ki, hogy a komponensautomaták történetei szintén összeilleszthetők az összetett automata egy történetévé.

8.3. tétel. . *Legyen $\{A_i\}_{i\in I}$ egy kompatibilis automatahalmaz, és legyen $A = \Pi_{i\in I}A_i$. Tételezzük fel, hogy β egy műveletsorozat külső(A)-ból. Ha $\beta|A_i \in$történetek(A_i) bármely $i \in I$-re, akkor $\beta \in$történetek(A).*

Bizonyítás. A bizonyítást meghagyjuk gyakorlatnak (lásd 8-2. gyakorlat). □

A 8.3. tétel szerint, ha be akarjuk bizonyítani, hogy egy sorozat a rendszer egy története, akkor elegendő bizonyítani, hogy a sorozat vetítése az egyes rendszerkomponensekre az illető komponens egy története.

8.2.2.. Elrejtés

Most pedig értelmezünk egy műveletet, amely „elrejti" egy b/k-automata kimeneti műveleteit, átminősítve ezeket belső műveletekké. Ez lehetővé teszi, hogy többé ne lehessen használni őket semmilyen kommunikációban, és nem is kerülnek be a történetekbe.

Először értelmezzük az elrejtés műveletét a lenyomatra: ha S egy lenyomat és $\Phi \subseteq ki(S)$, akkor $elrejt_\Phi(S)$ az az új S' lenyomat, amelyre $be(S') = be(S)$, $ki(S') = ki(S) - \Phi$ és $belső(S') = belső(S) \cup \Phi$.

A b/k-automata elrejtés műveletét most már könnyű értelmezni: ha A egy automata és $\Phi \subseteq ki(A)$, akkor $elrejt_\Phi(A)$ az az új A' automata, amelyet az A-ból úgy kapunk, hogy $lenyomat(A)$-t helyettesítjük a következővel: $lenyomat(A') = elrejt_\Phi(lenyomat(A))$.

8.3.. Pártatlanság

Osztott rendszerekben általában az összekapcsolásnak csak azon végrehajtási sorozatai érdekelnek bennünket, amelyekben minden komponens pártatlanul kap lehetőséget lépések végrehajtására. Ebben az alfejezetben megfelelően definiáljuk a pártatlanság fogalmát b/k-automatákra.

Emlékezzünk, hogy minden b/k-automatának van egy helyileg ellenőrzött műveletekre vonatkozó partícionálása, a partícionálás minden ekvivalenciaosztálya az automata valamely taszkjait írja le, amelyet az feltételezésünk szerint végrehajt. A pártatlanság, a mi esetünkben, azt jelenti, hogy minden taszknak végtelen sok lehetősége van, hogy valamely műveletét végrehajtsa.

Formálisan, az A b/k-automata egy α végrehajtási sorozatrészletét *pártatlannak* nevezzük, ha a következő feltételek a $taszkok(A)$ minden C osztályára teljesülnek.

1. Ha α véges, akkor C nem megengedett az α végállapotában.
2. Ha α végtelen, akkor α vagy végtelen sok eseményt tartalmaz C-ből vagy végtelen sok olyan állapotot, amelyekben C nem megengedett.

Itt és máshol is, az *esemény* fogalma egy művelet megjelenését jelenti egy sorozatban (pl. végrehajtási sorozatban vagy történetben).

Megérthetjük a pártatlanság definícióját, ha azt mondjuk, hogy minden C taszk (azaz ekvivalenciaosztály) végtelen sok lehetőséget kap műveletei végrehajtására. Bármikor, amikor ez megtörténik, vagy C egy művelete hajtódik végre, vagy C egyetlen műveletét sem lehet végrehajtani, mert nincs megengedett művelete. Egy véges pártatlan végrehajtási sorozatot úgy képzelhetünk el, hogy az automata minden taszknak körkörösen újra és újra lehetőséget ad végrehajtási

sorozata végén, de semmilyen műveletet nem sikerül elvégezni, mert nincs megengedett művelet az utolsó állapotban. Az A automata pártatlan végrehajtási sorozatai halmazának jelölése: *pártatlan_végre*(A). Azt mondjuk, hogy β A-nak egy *pártatlan története*, ha β A egy pártatlan végrehajtási sorozatának a története, és a pártatlan történetek halmazának a jelölése: *pártatlan_történetek*(A).

8.3.1. példa. Pártatlanság

A 8.1.3. példában az első végrehajtási sorozat pártatlan, mivel a végállapotában nincs megengedett `fogad` művelet. A második végrehajtási sorozat nem pártatlan mert véges, és létezik megengedett `fogad` művelet a végállapotában. A harmadik végrehajtási sorozat szintén nem pártatlan, mivel végtelen, nincs benne `fogad` esemény, de az első lépés után minden állapotában a `fogad` művelet megengedett.

8.3.2. példa. Pártatlanság

A pártatlanság definíciójának további illusztrálására, tekintsük az Óra b/k-automatát, amely nem más mint egy diszkrét óra.

8.1. automata. Óra b/k

Lenyomat:

Bemenet:
`igényel`

Kimenet:
`időzít`(t), $t \in \mathbb{N}$

Belső:
`ketyeg`

Állapotok:

számláló $\in \mathbb{N}$, kezdetben 0
bitjelző, logikai változó, kezdetben *hamis*

Átmenetek:

`ketyeg`
Előfeltétel:
igaz
Hatás:
számláló := *számláló* + 1

`időzít`(t)
Előfeltétel:
bitjelző = *igaz*
számláló = t
Hatás:
bitjelző := *hamis*

`igényel`
Hatás:
bitjelző := *igaz*

Taszkok:
{ketyeg}
{időzít(t) : $t \in \mathbb{N}$}

- -

Az ÓRA automata egyszerűen csak örökké „ketyeg", egy számlálót növelve. Ezenkívül, ha igénylés érkezik, az ÓRA visszaadja (egy külön lépésben) a számláló aktuális értékét. A következő műveletsorozat az ÓRA egy pártatlan végrehajtási sorozatából való:

ketyeg, ketyeg, ketyeg, ...

A következő sorozat egy nem pártatlan végrehajtási sorozat műveletsorozata:

ketyeg, ketyeg, ketyeg

Tulajdonképpen, az ÓRA automatának nincs véges pártatlan végrehajtási sorozata, mivel a ketyeg művelet mindig megengedett. A következő sorozat pártatlan:

ketyeg, ketyeg, igényel, ketyeg, ketyeg, időzít(4), ketyeg, ketyeg, ...,

mivel, ha egyszer az ÓRA válaszolt az egyetlen igénylésre, több időzít művelet már nem megengedett. Végül, nem pártatlan a következő

ketyeg, ketyeg, igényel, ketyeg, ketyeg, ketyeg, ...

sorozat, mivel az igényel esemény után, az időzít taszk továbbra is megengedett, de időzít nem fordul elő.

Bebizonyíthatunk a 8.1–8.3. tételekhez hasonlókat a pártatlan végrehajtási sorozatokra is.

8.4. tétel. . *Legyen* $\{A_i\}_{i\in I}$ *egy kompatibilis automatahalmaz, és legyen* $A = \Pi_{i\in I}A_i$.

1. *Ha* $\alpha \in$*pártatlan_végre*(A), *akkor* $\alpha|A_i \in$ *pártatlan_végre*(A_i) *minden* $i \in I$ *értékre.*
2. *Ha* $\beta \in$ *pártatlan_történetek*(A), *akkor* $\beta|A_i \in$ *pártatlan_történetek*(A_i) *minden* $i \in I$ *értékre.*

8.5. tétel. . *Legyen* $\{A_i\}_{i\in I}$ *egy kompatibilis automatahalmaz, és legyen* $A = \Pi_{i\in I}A_i$. *Tételezzük fel, hogy* α_i *az* A_i *egy pártatlan végrehajtási sorozata minden* $i \in I$ *értékre és* β *egy műveletsorozat a külső*(A)*-ból úgy, hogy* $\beta|A_i =$*történet*(α_i) *minden* $i \in I$*-re. Ekkor létezik* A*-nak egy olyan* α *pártatlan végrehajtási sorozata, amelyre* $\beta =$*történet*(α) *és* $\alpha_i = \alpha|A_i$ *minden* $i \in I$ *értékre.*

8.6. tétel. . *Legyen* $\{A_i\}_{i\in I}$ *egy kompatibilis automatahalmaz, és legyen* $A = \Pi_{i\in I}A_i$. *Tételezzük fel, hogy* β *egy műveletsorozat külső*(A)*-ból. Ha* $\beta|A_i \in$ *pártatlan_történetek*(A_i) *bármely* $i \in I$*-re, akkor* $\beta \in$ *pártatlan_történetek*(A).

Bizonyítás. A bizonyításokat meghagyjuk gyakorlatnak (lásd 8-3. gyakorlat). □

A 8.1–8.3. és 8.4–8.6. tételek lehetővé teszik, hogy moduláris módon gondolkozzunk a b/k-automaták összekapcsolásával modellezett osztott rendszerek viselkedéséről.

8.3.3. példa. Pártatlanság

Tekintsük a 8.2.1. példa három folyamatból és három csatornából álló rendszerének a pártatlan végrehajtási sorozatait. Minden pártatlan végrehajtási sorozatban minden elküldött üzenetet egyszer majd fogad a rendszer. Ugyanígy, minden pártatlan végrehajtási sorozatban, amelyben van legalább egy $\texttt{kezd}_i$ esemény minden i-re, mindegyik folyamat végtelen sok üzenetet küld minden más folyamatnak, és minden folyamat végtelen sok `dönt` műveletet végez.

Másfelől, minden olyan pártatlan végrehajtási sorozatban, amely nem tartalmaz legalább egy `kezd` eseményt minden folyamatra, egyetlen folyamat se hajt végre egyelten `dönt` lépést sem. Megjegyezzük, hogy a pártatlanság nem feltételezi a `kezd` események meglétét – a P_i folyamatokhoz rendelt $\texttt{kezd}_i$ események száma lehet véges (akár nulla is) vagy végtelen.

Befejezésül megadunk egy tételt, amely szerint minden véges végrehajtási sorozat (vagy történet) kiterjeszthető úgy, hogy pártatlan legyen.

8.7. tétel. . *Legyen A egy b/k-automata.*

1. *Ha α az A automata egy véges végrehajtási sorozata, akkor létezik A-nak olyan pártatlan végrehajtási sorozata, amelynek α az eleje.*
2. *Ha β az A automata egy véges története, akkor létezik A-nak olyan pártatlan története, amelynek β az eleje.*
3. *Ha α az A automata egy véges végrehajtási sorozata, és β az A bemeneti műveleteinek tetszőleges (véges vagy végtelen) sorozata, akkor létezik A-nak egy olyan $\alpha \cdot \alpha'$ pártatlan végrehajtási sorzata, hogy az α' bemeneti műveleteinek sorozata éppen β.*
4. *Ha β az A automata egy véges története, és β' az A bemeneti műveleteinek tetszőleges (véges vagy végtelen) sorozata, akkor létezik A-nak egy olyan $\alpha \cdot \alpha'$ pártatlan végrehajtási sorozata, hogy történet$(\alpha) = \beta$, és hogy az α' bemeneti műveleteinek sorozata éppen β'.*

Bizonyítás. A bizonyítást meghagyjuk gyakorlatnak (lásd 8-7. gyakorlat). □

8.4.. Feladatok bemeneti és kimeneti adatai

A b/k-automatákkal megoldott feladatoknak bizonyos típusú bemenetei és kimenetei vannak, amelyeket valamilyen módon modelleznünk kell. A szinkron modell esetében ezt általában úgy oldottuk meg, hogy a bemeneteket és kimeneteket

speciális állapotváltozókkal modelleztük. Feltételeztük, hogy a bemenetek a kezdőállapotok külön változóiba épülnek be, míg a kimenetek kizárólagosan-írható változókban jelennek meg. Hasonlóan járhatunk el az aszinkron esetben is. Ebben az esetben viszont, mivel az automatának vannak bemeneti és kimeneti műveletei, sokkal kézenfekvőbb, hogy a rendszer bemeneteit és kimeneteit az automata megfelelő bemeneti és kimeneti műveleteivel írjuk le.

8.5.. Tulajdonságok és bizonyítási módszerek

A b/k-automatákat nemcsak arra használhatjuk, hogy aszinkron rendszereket írjunk le pontosan, hanem arra is, hogy állításokat fogalmazzunk meg és bizonyítsunk be arról, amit a rendszer csinál. Ebben az alfejezetben néhány olyan tulajdonságtípust mutatunk be, amelyeket általában bizonyítani szokás az aszinkron rendszerekről, és ugyanakkor néhány olyan módszert is, amelyeket a bizonyításukban szoktak használni.

A 10–13. és 15–22. fejezetekben, amelyek az aszinkron algoritmusokról szólnak, használni fogjuk az itt leírt módszereket (és még más alkalmi érvelést), hogy aszinkron algoritmusok tulajdonságait bizonyítsuk be. Érveléseinket, akár használjuk a bemutatott módszerek valamelyikét, akár nem, minden esetben pontosan igazolni lehet b/k-automaták segítségével.

8.5.1.. Invariáns állítások

A legfontosabb bizonyítandó tulajdonság az *invariáns állítás*. Ebben a könyvben az A automatára vonatkozó invariáns állításon olyan tulajdonságot értünk, amelyik igaz az A minden elérhető állapotára.[3]

Az invariáns állításokat általában indukcióval bizonyítjuk, ahol az indukciós változó az illető állapothoz vezető végrehajtási sorozat lépéseinek a száma. Általánosabban, be lehet bizonyítani egy adott (vagy néhány) invariáns állítást úgy, hogy felhasználjuk az induktív bizonyításban már bebizonyított invariáns állításokat.

Emlékeztetünk arra, hogy már a szinkron algoritmusoknál is használtunk invariáns állításokat bizonyos tulajdonságok bizonyítására. Szinkron esetben az invariáns állításokat a rendszer tetszőleges számú *menet* utáni állapotaira bizonyítottuk. Aszinkron esetben a menet helyett *lépés* szerepel. Mivel aszinkron algoritmusok esetében az érvelés sokkal aprólékosabb, a bizonyítás általában hosszabb, részletesebb és nehezebb.

8.5.2.. Történetre vonatkozó tulajdonságok

A felhasználó szempontjából egy b/k-automata „fekete doboznak” tekinthető. Amit a felhasználó lát, az csak az automata végrehajtási sorozatának a történetei (vagy pártatlan történetei). Sok, b/k-automatára vonatkozó bebizonyítandó tu-

[3] „Mindig igaz” állítás. *A lektor.*

lajdonságot természetes módon átfogalmazhatunk az automata történeteire vagy pártatlan történeteire vonatkozó tulajdonságokká.

Formálisan, egy *történetre vonatkozó P tulajdonság* a következőkből áll:

- *lenyomat*(P), egy belső műveleteket nem tartalmazó lenyomat;
- *történetek*(P), a *művelethalmaz*(*lenyomat*(P))-beli (véges vagy végtelen) műveletsorozatok egy halmaza.

Azaz, a történetre vonatkozó tulajdonság egyaránt specifikál egy külső felületet és egy sorozathalmazt (vagyis tulajdonságot), amelyet azon a felületen észlelünk. Rövidítésként *művelethalmaz*(*lenyomat*(P)) helyett *művelethalmaz*(P)-t írunk, hasonlóképpen használjuk a *be*(P) jelölést is, és így tovább.

Az az állítás, hogy egy A-val jelölt b/k-automata rendelkezik egy történetre vonatkozó P tulajdonsággal, a következő két feltétel legalább egyikét jelentheti.

1. *külső_ lenyomat*(A) = *lenyomat*(P) és *történetek*$(A) \subseteq$*történetek*(P).
2. *külső_ lenyomat*(A) = *lenyomat*(P) és *pártatlan_ történetek*$(A) \subseteq$ *történetek*(P).

Intuitívan, mindkét esetben, az A automatának minden lehetséges külső viselkedése a P tulajdonság által is megengedett. Megjegyezzük, hogy nem követeljük meg a fordított tartalmazást – azaz azt, hogy P minden története valójában A-nak is egy lehetséges története legyen. Mindazonáltal, a fenti két tartalmazás nem triviális, mivel az a tény, hogy A megengedett bemenetű, biztosítja, hogy *pártatlan_ történetek*(A) (és ugyanúgy *történetek*(A) is) tartalmazza A egy válaszát minden lehetséges bemenetiművelet-sorozatra. Ha *pártatlan_ történetek*$(A) \subseteq$ *történetek*(P), akkor minden eredményül kapott sorozatnak a P tulajdonság részének kell lennie.

Mivel van bizonyos kétértelműség, hogy mit is jelent az, hogy egy automata megfelel egy „történetre vonatkozó tulajdonságnak", minden egyes esetben megmondjuk, hogy mit értünk ezen.

8.5.1. példa. Automatákra és történetekre vonatkozó tulajdonságok

Tekintsünk egy automatát és olyan, történetre vonatkozó tulajdonságokat, amelyeknek a bemenethalmaza $\{0\}$, kimenethalmaza pedig $\{1, 2\}$. Először feltételezzük, hogy *történetek*(A) a $\{0, 1, 2\}$ halmaz feletti olyan sorozatok halmaza, amelyek legalább egy 1-est tartalmaznak. Ekkor *pártatlan_ történetek*$(A) \subseteq$*történetek*(P) azt jelenti, hogy minden pártatlan végrehajtási sorozatban, A-nak legalább egy 1-est kell kimenetként adnia. Könnyű ilyen b/k-automatát tervezni, például az automatának lehet egy taszkja, amelyiknek egyszerűen csak annyi a feladata, hogy 1-est adjon kimenetként. A pártatlanság itt azt biztosítja, hogy ennek a taszknak valójában van is esélye 1-est adni kimenetként. Másfelől, nem létezik egyetlen olyan automata sem, amelyre *történetek*$(A) \subseteq$ *történetek*(P), mivel *történetek*(A) mindig tartalmazza a λ üres sztringet, amelyik természetesen nem tartalmazza az 1-est.

Most tételezzük fel, hogy *történetek*(P) a $\{0,1,2\}$ halmaz feletti olyan sorozatok halmaza, amelyek legalább egy 0-t tartalmaznak. Ebben az esetben nincs olyan A b/k-automata (az adott külső felülettel), amelyre *pártatlan_ történetek*$(A) \subseteq$*történetek*(P), mivel *pártatlan_ történetek*(A)-nak olyan sorozatot is kell tartalmaznia, amelyikben nincs bemenet.

A történetre vonatkozó tulajdonságokra definiálunk egy összekapcsolás műveletet. Pontosabban, azt mondjuk, hogy történetre vonatkozó tulajdonságok egy $\{P_i\}_{i\in I}$ megszámlálható végtelen halmaza *kompatibilis*, ha a lenyomataik kompatibilisek. Ekkor, a $P = \Pi_{i\in I}P_i$ *összekapcsolás* az a történetre vonatkozó tulajdonság, amelyre

1. *lenyomat*$(P) = \Pi_{i\in I}$*lenyomat*(P_i),
2. *történetek*(P) a P azon külső műveleteiből álló β sorozatainak halmaza, amelyre $\beta|$*művelethalmaz*$(P_i) \in$ *történetek*(P_i) minden $i \in I$ értékre.

8.5.3.. Biztonságossági és elevenségi tulajdonságok

Ebben az alfejezetben két fontos, történetre vonatkozó tulajdonságtípust definiálunk, a *biztonságossági tulajdonságot* és a az *elevenségi tulajdonságot*. Azután megadunk két alaperedményt ezekre a tulajdonságokra, és megmutatjuk, hogyan lehet ilyen tulajdonságokat bizonyítani.

Biztonságossági tulajdonságok.. Azt mondjuk, hogy egy történetre vonatkozó P tulajdonság *történetre vonatkozó biztonságossági tulajdonság* vagy röviden csak *biztonságossági tulajdonság*, ha P kielégíti a következő feltételeket:

1. *történetek*(P) nem üres,
2. *történetek*(P) *kezdőszeletre vonatkozóan zárt*, azaz ha $\beta \in$*történetek*(P) és β' a β véges kezdőszelete, akkor $\beta' \in$*történetek*(P);
3. *történetek*(P) *határértékre vonatkozóan zárt*, azaz ha $\beta_1, \beta_2, \ldots$ a *történetek*(P) véges sorozatainak végtelen sorozata, és minden i-re β_i kezdőszelete β_{i+1}-nek, akkor az az egyértelműen meghatározható α, amely β_i határértéke (prefix-rendezés szerint), szintén benne van a *történetek*(P)-ben.

A biztonságossági tulajdonságot gyakran úgy értelmezzük mintha „rendellenes" helyzetek nem következnének be. Feltételezzük, hogy ha valami rendellenes következik be a történetben, akkor az valamilyen egyedi esemény miatt van; ennek következtében a határértékre vonatkozó zártság belefoglalása a definícióba elfogadható feltétel. Ha semmi rendellenesség nem következik be a történetben, akkor semmi rendellenesség nem következik be a történet egyetlen kezdőszeletében sem, tehát a kezdőszeletre vonatkozó zártság is elfogadható feltétel. Végül, semmilyen rendellenesség sem következhet be az első esemény megjelenése előtt, azaz semmi rendellenesség sem lehet az üres λ sorozatban, tehát az is elfogadható feltétel, hogy a *történetek*(P) nem lehet üres.

8.5.2. példa. Történetre vonatkozó biztonságossági tulajdonság

Tételezzük fel, hogy a *lenyomat*(P) `kezd`(v), $v \in V$ bemenetekből és `dönt`(v), $v \in V$ kimenetekből áll. Feltételezzük, hogy *történet*(P) olyan `kezd` és `dönt` műveletekből álló sorozatok halmaza, amelyekben `dönt`(v) soha nem fordul elő egy őt megelőző `kezd`(v) nélkül (ugyanarra a v-re). Ekkor P biztonságossági tulajdonság.

Ha P biztonságossági tulajdonság, akkor a *történetek*(A) $\subseteq$ *történetek*(P) állítás egyenértékű a *pártatlan_ történetek*(A) $\subseteq$ *történetek*(P) állítással, ez pedig egyenértékű azzal, hogy A összes véges története benne van *történetek*(P)-ben. (A bizonyítást meghagyjuk gyakorlatnak – lásd 8-12. gyakorlat) Egy adott A automatára, általában azt a legegyszerűbb bizonyítani, hogy az A véges történetei mind benne vannak *történetek*(P)-ben. Ezt általában teljes indukcióval bizonyítjuk az illető történetet generáló véges végrehajtási sorozat hossza szerint. Ez a stratégia nagyon közel áll ahhoz, amit az invariánsok bizonyításánál használtunk. Tulajdonképpen, ha hozzáadunk A-hoz egy állapotváltozót, amely a már generált történetet tartja nyilván, akkor a P biztonságossági tulajdonságot úgy is megfogalmazhatjuk, mint az automata állapotára vonatkozó invariánst.

Elevenségi tulajdonságok.. Azt mondjuk, hogy egy történetre vonatkozó P tulajdonság *történetre vonatkozó elevenségi tulajdonság* vagy röviden csak *elevenségi tulajdonság*, ha bármely – *műveletthalmaz*(P) feletti – véges sorozatnak van *történetek*(P)-beli kiterjesztése.

Az elevenségi tulajdonságot gyakran úgy értelmezzük, mintha bizonyos „jó” dolgok végül is bekövetkeznének[4] (a formális definícióban természetesen ennél bonyolultabb állítások vannak). Feltételezzük, hogy függetlenül attól, hogy mi történt egy adott pontig, a jövőben valamikor bekövetkezhet valami jó.

8.5.3. példa. Történetre vonatkozó elevenségi tulajdonság

Tételezzük fel, hogy a *lenyomat*(P) `kezd`(v), $v \in V$ bemenetekből és `dönt`(v), $v \in V$ kimenetekből áll. Feltételezzük, hogy *történet*(P) olyan `kezd` és `dönt` műveletekből álló β sorozatok halmaza, amelyekben bármely `kezd` esemény után előfordul β-ban egy `dönt` esemény. Ekkor P elevenségi tulajdonság. Hasonló a helyzet abban az esetben is, ha β bármely `kezd` eseményére végtelen sok `dönt` esemény jelenik meg β-ban valahol később.

Gyakran szeretnénk bebizonyítani, hogy *pártatlan_ történetek*(A) $\subseteq$ *történetek*(P), bizonyos A automatára és P elevenségi tulajdonságra, azaz, hogy az A összes pártatlan története kielégít bizonyos elevenségi tulajdonságokat. A *temporális logikán* alapuló módszerek a gyakorlatban jól használhatók az ilyen követelmények bizonyításában. A temporális logika egy logikai nyelvből, valamint bizonyítási szabályok halmazából áll. A logikai nyelv szimbólumokat tartalmaz olyan temporális fogalmakra, mint például „valamikor” és „mindig”. A bizonyítási szabályok a végrehajtási sorozatok tulajdonságainak leírására és ellenőrzésére szolgálnak.

[4] Ebben az esetben haladási tulajdonságról beszélünk. *A lektor.*

Egy másik módszer elevenségi állítások bizonyítására, amelyet *haladási (variáns) függvények módszerének* nevezünk, kimondottan arra való, hogy bizonyítsa, hogy valamilyen célt valamikor elérünk. Ez a módszer magában foglalja egy „haladó függvény" definiálását az automata állapotain, amelynek értékei egy megalapozott halmazból valók, és annak bizonyítását, hogy bizonyos műveletek biztosítják a függvényérték csökkenését mindaddig amíg a célt el nem érjük. A haladási függvény módszerét formalizálni lehet a temporális logika segítségével.

Ebben a könyvben az elevenségi tulajdonságokat nem formálisan bizonyítjuk, de minden érvelésünk formalizálható a temporális logika segítségével.

A következő két egyszerű alaptétel leírja a biztonságossági és elevenségi tulajdonságok közötti kapcsolatokat. Az első tétel szerint nincs olyan, történetre vonatkozó nem triviális tulajdonság, amely egyidejűleg biztonságossági és elevenségi is.

8.8. tétel. . *Ha P egyidejűleg biztonságossági és elevenségi tulajdonság, akkor P a $műveléthalmaz(P)$ összes (véges vagy végtelen) sorozatának a halmaza.*

Bizonyítás. Tételezzük fel, hogy P egyidejűleg biztonságossági és elevenségi tulajdonság, és legyen β egy tetszőleges sorozat, melynek elemei $műveléthalmaz(P)$-ből valók. Ha β véges, akkor, mivel P elevenségi tulajdonság, a β sorozatnak van β' kiterjesztése $történetek(P)$-ben. De, mivel P biztonságossági tulajdonság is – speciálisan, mivel $történetek(P)$ kezdőszeletre vonatkozóan zárt –, kötelező módon $\beta \in történetek(P)$. Tehát, minden véges sorozat, melynek elemei $műveléthalmaz(P)$-ból valók, benne van $történetek(P)$-ben.

Másfelől, ha β végtelen, akkor minden $i \geq 1$ értékre definiáljuk a β sorozat $\beta_i$vel jelölt i hosszúságú kezdőszeletét. Ekkor, mint azt már láttuk az előző szakaszban, minden β_i eleme $történetek(P)$-nek. Ezért, mivel P biztonságossági tulajdonság – speciálisan, mivel $történetek(P)$ határértékre vonatkozóan zárt –, kötelező módon $\beta \in történetek(P)$. □

A második tétel kimondja, hogy *minden*, történetre vonatkozó tulajdonságot ki lehet fejezni biztonságossági és elevenségi tulajdonságok metszeteként (vagy, ami ezzel egyenértékű, konjunkciójaként).

8.9. tétel. . *Ha P tetszőleges, történetre vonatkozó tulajdonság, amelyre $történetek(P) \neq \emptyset$, akkor létezik egy S biztonságossági tulajdonság és egy L elevenségi tulajdonság úgy, hogy*

1. $lenyomat(S) = lenyomat(L) = lenyomat(P)$,
2. $történetek(P) = történetek(S) \cap történetek(L)$.

Bizonyítás. Legyen $történetek(S)$ a $történetek(P)$ kezdőszeletre és határértékre vonatkozó zártja, azaz az a legkisebb sorozathalmaz $műveléthalmaz(P)$ felett, amelyik egyben kezdőszeletre és határértékre vonatkozóan is zárt, és amelyik tartalmazza $történetek(P)$-t. Nyilvánvalóan, S biztonságossági tulajdonság. Legyen
$történetek(L) = történetek(P) \cup \{\beta : \beta$ véges sorozat és β-nak nincs kiterjesztése $történetek(P)$-ben $\}$.

Ekkor állítjuk, hogy L elevenségi tulajdonság. Hogy ezt bebizonyítsuk, tekintsünk egy tetszőleges véges β sorozatot, amelynek elemei *művelethalmaz*(P)-ből valók. Ha β-nak valamilyen kiterjesztése benne van *történetek*(P)-ben, akkor természetesen ez a kiterjesztés benne van *történetek*(L)-ben, mivel *történetek*$(P) \subseteq$ *történetek*(L). Másfelől, ha β-nak nincs kiterjesztése *történetek*(P)-ben, akkor β kimondottan úgy van definiálva. hogy benne legyen *történetek*(L)-ben. Mindkét esetben, β-nak van kiterjesztése *történetek*(L)-ben, tehát L elevenségi tulajdonság.

Ezután, azt állítjuk, hogy *történetek*(P) = *történetek*$(S) \cap$ *történetek*(L). Nyilvánvaló, hogy *történetek*$(P) \subseteq$ *történetek*$(S) \cap$ *történetek*(L), mivel mind S mind L kimondottan úgy van definiálva, hogy a történeteik magukban foglalják a P történeteit. Be kell bizonyítanunk, hogy *történetek*$(S) \cap$ *történetek*$(L) \subseteq$ *történetek*(P). Tételezzük fel, hogy $\beta \in$ *történetek*$(S) \cap$ *történetek*(L), és ennek ennek ellenére $\beta \notin$ *történetek*(P). Az L definíciója alapján β olyan véges sorozat, amelynek nincs kiterjesztése *történetek*(P)-ben. De $\beta \in$ *történetek*(S), és *történetek*(S) kezdőszeletre és határértékre vonatkozóan zártja *történetek*(P)-nek. Ezért, mivel β véges sorozat, kötelezően egy *történetek*(P)-beli elem kezdőszelete. Ez pedig ellentmondás. □

Eddig csupán a történetekre vonatkozó biztonságossági és elevenségi tulajdonságokat definiáltuk, de hasonló módon végrehajtási sorozatokra vonatkozó biztonságossági és elevenségi tulajdonságokat is definiálhatunk. Az eredmények is hasonlók. A következő fejezetekben gyakran fogjuk osztályozni a végrehajtási sorozatok tulajdonságait mint biztonságossági és elevenségi tulajdonságokat.

8.5.4.. Összekapcsolási érvelés

Az összetett automata tulajdonságainak bizonyításakor gyakran hasznos, ha az automata komponenseit külön is megvizsgáljuk. Ebben a szakaszban néhány példát adunk az ilyen „összekapcsolási" (kompozicionális) érvelésre.

Először, ha $A = \Pi_{i \in I} A_i$ és minden A_i rendelkezik egy történetre vonatkozó P_i tulajdonsággal, akkor A rendelkezik a $P = \Pi_{i \in I} P_i$ szorzattulajdonsággal is. A 8.10. tétel pontosabban fejezi ki ugyanezt.

8.10. tétel. . *Legyen $\{A_i\}_{i \in I}$ automaták egy kompatibilis halmaza, és legyen $A = \Pi_{i \in I} A_i$. Legyen $\{P_i\}_{i \in I}$, történetekre vonatkozó tulajdonságok egy (kompatibilis) halmaza, és legyen $P = \Pi_{i \in I} P_i$.*

1. *Ha külső_lenyomat(A_i) = lenyomat(P_i) és történetek$(A_i) \subseteq$ történetek(P_i) minden i-re, akkor külső_lenyomat(A) = lenyomat(P) és történetek$(A) \subseteq$ történetek(P).*
2. *Ha külső_lenyomat(A_i) = lenyomat(P_i) és pártatlan_történetek$(A_i) \subseteq$ történetek(P_i) minden i-re, akkor külső_lenyomat(A) = lenyomat(P) és pártatlan_történetek$(A) \subseteq$ történetek(P).*

Bizonyításvázlat. Az 1. rész könnyen bizonyítható a 8.1. tétel alapján (amely szerint az összetett A rendszer minden története bármelyik A_i komponensre levetítve az A_i egy történetét adja). A 2. rész hasonlóképpen következik a 8.4. tételből. □

8.5.4. példa. Történetek szorzatára vonatkozó tulajdonság

Tekintsük a 8.2.1. példában szereplő összetett rendszert. Minden P_i folyamatautomata rendelkezik (a történetek tartalmazása értelmében) egy, történetre vonatkozó biztonságossági tulajdonsággal, amelyik szerint minden $\texttt{dönt}_i$ eseményt megelőz egy $\texttt{kezd}_i$ esemény. Minden $C_{i,j}$ csatornaautomata szintén rendelkezik egy, történetre vonatkozó biztonságossági tulajdonsággal, amely szerint a $\texttt{fogad}_{i,j}$ eseményeket tartalmazó üzenetsorozat kezdőszelete a $\texttt{küld}_{i,j}$ eseményeket tartalmazó üzenetsorozatnak.

Ekkor, a 8.10. tételből következik, hogy az összetett rendszer rendelkezik a történetek szorzatára vonatkozó tulajdonsággal. Ez azt jelenti, hogy az összetett rendszer bármely történetében igazak a következők.

1. Minden i-re bármely $\texttt{dönt}_i$ eseményt megelőz egy $\texttt{kezd}_i$ esemény.
2. Minden i-re és j-re, ahol $i \neq j$, a $\texttt{fogad}_{i,j}$ eseményeket tartalmazó üzenetsorozat kezdőszelete a $\texttt{küld}_{i,j}$ eseményeket tartalmazó üzenetsorozatnak.

Másodszor, tételezzük fel, hogy be szeretnénk bizonyítani, hogy egy adott műveletsorozat az $A = \Pi_{i \in I} A_i$ összetett rendszer egy története. Ez általában akkor merül fel, ha A egy absztrakt rendszer, amelyet egy feladat specifikációjában használunk. A 8.3. tétel megmutatja, hogy elegendő bebizonyítani, hogy a sorozat levetítése a rendszer bármely komponensére, annak a komponensnek egy története. A 8.6. tételből hasonló eredmény következik a pártatlan történetekre.

Harmadszor, tekintsük a biztonságossági tulajdonságok kompozicionális bizonyítását. Tételezzük fel, hogy azt szeretnénk bizonyítani, hogy az $A = \Pi_{i \in I} A_i$ összetett rendszer rendelkezik egy történetre vonatkozó P tulajdonsággal. A stratégiánk az, hogy bebizonyítjuk, hogy nincs egyetlen olyan A_i komponens sem, amely elsőként sérti meg a P tulajdonságot. Ez a stratégia hasznos, például, amikor be akarjuk bizonyítani, hogy egy komponenspár betart egy „kézfogási protokollt" a két komponens között, felváltva jeleket küldve. Be lehet bizonyítani, hogy ha egyik komponens sem szegi meg elsőként a protokollt, akkor a protokoll megmarad.

Formálisan definiáljuk a biztonságossági tulajdonságot „megőrző" automata fogalmát: legyen A egy b/k-automata, és legyen P egy biztonságossági tulajdonság, amelyre $\mathit{műveletalmaz}(P) \cap \mathit{belső}(A) = \emptyset$ és $\mathit{be}(P) \cap \mathit{ki}(A) = \emptyset$. Azt mondjuk, hogy A *megőrzi* a P tulajdonságot, ha minden véges β műveletsorozatra, amely nem tartalmazza A egyetlen belső műveletét sem, és minden π eleme $\mathit{ki}(A)$-ra teljesül: ha $\beta|\mathit{műveletalmaz}(P) \in \mathit{történetek}(P)$ és $\beta\pi|A \in \mathit{történetek}(A)$, akkor $\beta\pi|\mathit{műveletalmaz}(P) \in \mathit{történetek}(P)$. Ez azt jelenti, hogy A nem az első, amelyik megsérti P-t. Mindaddig amíg az A környezete csak olyan bemeneteket szolgáltat A-nak, hogy az együttes viselkedés kielégíti P-t, addig A is csak olyan kimeneteket szolgáltat majd, hogy az együttes viselkedés kielégíti P-t.

A biztonságossági tulajdonságok megőrzésének fontos tényezője az, hogy ha egy összetett rendszerben minden komponens megőrzi a biztonságossági tulajdonságot, akkor azt az egész rendszer is megőrzi. Sőt, ha az összetett rendszer

zárt, akkor valójában rendelkezik a biztonságossági tulajdonsággal.

8.11. tétel. . *Legyen $\{A_i\}_{i\in I}$ automaták egy kompatibilis halmaza, és legyen $A = \Pi_{i\in I}A_i$. Legyen P egy biztonságossági tulajdonság, amelyre műveléthalmaz$(P) \cap$ belső$(A) = \emptyset$ és be$(P) \cap$ ki$(A) = \emptyset$.*

1. *Ha A_i megőrzi P-t minden $i \in I$ értékre, akkor A megőrzi P-t.*
2. *Ha A zárt automata, A megőrzi P-t, és műveléthalmaz$(P) \subseteq$külső(A), akkor történetek$(A)|$műveléthalmaz$(P) \subseteq$történetek(P).*
3. *Ha A zárt automata, A megőrzi P-t, és műveléthalmaz$(P) =$külső(A), akkor történetek$(A) \subseteq$történetek(P).*

Bizonyítás. A bizonyítást meghagyjuk gyakorlatnak (lásd 8-15. gyakorlat). □

8.5.5. példa. Tulajdonságokat megőrző automata

Legyen A egy olyan automata, amelynek kimenete `a`, bemenete `b` és B egy olyan automata, amelynek kimenete `b`, bemenete `a`. Tekintsük a P biztonságossági tulajdonságot, amelyre *lenyomat*(P)-nek nincs bemenete, `a` és `b` mindegyike kimenet, és amelyre *történetek*(P) az összes `a`-val kezdődő, csak `a`-t és `b`-t váltakozva tartalmazó véges és végtelen sorozat halmaza (beleértve a λ üres sorozatot is). P egy kézfogási protokollt ábrázol A és B között úgy, hogy A kezdi a kéznyújtást.

Feltételezzük, hogy A-nak van egy *váltás* nevű változója, amely az $\{a, b\}$ halmazból vesz fel értéket, kezdőértéke pedig a. Az A automata átmenetei a következők.

Átmenetek:

`a`

Előfeltétel:
váltás $= a$
Hatás:
váltás $:= b$

`b`

Hatás:
váltás $:= a$

Így tehát, az A automata elvégezheti `a`-t kezdetben, és bármikor ismét, ha bemenetként `b`-t kap. Ha két `b`-t kap, mielőtt még az elsőre válaszolhatna, csak egy `a`-t ad válaszként.

A B automatának van egy *váltás* nevű változója, amely az $\{a, b\}$ halmazból vesz fel értéket, kezdőértéke pedig a, és még egy *hiba* logikai változója, melynek kezdőértéke *hamis*. A B automata átmenetei a következők.

Átmenetek:

b
Előfeltétel:
váltás = *b* **or** *hiba* = *igaz*
Hatás:
if *hiba* = *hamis*
then *váltás* := *a*

a
Hatás:
if *hiba* = *hamis* **then**
if *váltás* = *a* **then** *váltás* := *b*
else *hiba* := *igaz*

B tehát csak b-t végezheti el egyszer-egyszer, mikor a-t kap bemenetként, mindaddig, ameddig a környezet nem szolgáltat egymásután két a-t. Ha két a jelenik meg, akkor B hibajelzőt állít be, amelyik lehetővé teszi, hogy bármikor b-t szolgáltasson kimenetként.

Mindkét automata, A is és B is, megőrzik a P tulajdonságot. A 8.11. tételből következik, hogy az összetett $A \times B$ automata minden története benne van *történetek*(P)-ben.

8.5.5.. Szintekre bontott bizonyítások

Ebben a szakaszban egy, az automaták hierarchiáján alapuló fontos bizonyítási stratégiát mutatunk be. Ez a hierarchia a rendszer vagy algoritmus leírását jelenti különböző absztrakciós szinteken. Azt a folyamatot, amely sorozatos absztrakción keresztül valósul meg, a legfelsőbb szinttől a legalsóbbig, *lépésenkénti finomításnak nevezzük*. A legfelsőbb szint nem más, mint a feladat specifikációja egy automata formájában. A következő szint tipikusan a rendszer egy nagyon absztrakt leírása: ez inkább központosított, mint osztott, vagy elnagyoltak műveletei, vagy egyszerű, de nem hatékony az a adatszerkezete. Az alsóbb szintek egyre inkább hasonlítanak a valós rendszerre vagy algoritmusra: ezek jobban szétosztottak, finomabb műveletei vannak, és optimalizálást is tartalmaznak. A sok részlet miatt az alsóbb szintek nehezebben érthetők, mint a felsőbbek. A legjobb módszer az alsó szintű automaták tulajdonságainak bizonyítására az, hogy felsőbb szintű automatákat feleltetünk meg nekik, és azokon végezzük el a bizonyítást.

A 4. és 6. fejezetben már láttunk példákat az ilyen finomításra szinkron esetben. Például, a 6. fejezetben először bemutattunk egy algoritmust (HalmazTerjed), amely megegyezésre törekszik a meghibásodások kiküszöbölésére, és amely nagyvonalú a kommunikáció szintjén. Aztán bemutattunk egy („alsóbb szintű") javított változatot (OptHalmazTerjed), amelyben nagyon sok üzenetet kiküszöböltünk, ami a kommunikációra kisebb korlátot eredményezett. A javított algoritmus ellenőrzése a két algoritmus állapotaira vonatkozó szimulációs reláció segítségével történt. A bizonyítás megmutatta, a menetek száma szerinti indukcióval, hogy a számítások során a szimulációs reláció mindvégig megőrződött. Lényegében, ez a stratégia abban állt, hogy párhuzamosan futtattuk a két algoritmust, ugyanazokkal a bemenetekkel és ugyanolyan meghibásodási séma szerint, figyelve a két végrehajtási sorozat közötti hasonlóságokat.

Hogyan lehet kiterjeszteni ezt a szimulációs módszert az aszinkron esetre? Az aszinkron modell sokkal nagyobb szabadságot enged, mint a szinkron modell,

mind a komponensek lépései sorrendje tekintetében, mind a műveleteket kísérő állapotváltozásokban. Emiatt nagyon nehéz eldönteni, hogy milyen végrehajtási sorozatokat hasonlítsunk össze. Kiderül, hogy elég csak *egyirányú kapcsolatot* teremteni a két algoritmus között, azaz megmutatni, hogy egy alsó szintű automata bármely végrehajtási sorozatához hozzárendelhető egy „megfelelő" végrehajtási sorozat a magasabb szintű automatában.

Speciálisan, legyen A és B két b/k-automata ugyanazzal a külső felülettel, és tekintsük A-t alsó szintű, míg B-t felső szintű automatának. Legyen f egy bináris reláció *állapotok*(A) és *állapotok*(B) felett, azaz $f \in$*állapotok*$(A)\times$*állapotok*(B). Az $(s,u) \in f$ helyett használjuk az $u \in f(s)$ jelölést is. Ekkor f *szimulációs reláció*, amennyiben igazak az alábbiak.

1. Ha $s \in$ *kezdő*(A), akkor $f(s) \cap$ *kezdő*$(B) \neq \emptyset$.
2. Ha s az A-nak elérhető állapota, $u \in f(s)$ a B-nek elérhető állapota, és $(s,\pi,s') \in$ *átmenetek*(A), akkor létezik B-nek egy u-val kezdődő és valamilyen $u' \in f(s')$-tel végződő α végrehajtási sorozatrészlete úgy, hogy *történet*$(\alpha) =$ *történet*(π).

Az első, vagy *kezdeti feltétel* szerint A bármely kezdő állapotának megfelel B egy kezdő állapota. A második, vagy *lépésfeltétel* szerint A bármely lépéséhez és B bármely, a lépés kezdő állapotának megfelelő állapotához hozzárendelhető B-nek egy lépéssorozata. Ez a hozzárendelt sorozat állhat egyetlen lépésből, több lépésből, vagy akár egyetlen lépésből sem, mindaddig amíg az állapotok közötti megfeleltetés fennáll és a külső viselkedés azonos. A lépésmegfeleltetés egy ábrázolása, amikor π külső művelet, a 8.4. ábrán látható. A következő tétel a szimulációs reláció kulcstulajdonságát adja.

8.12. tétel. . *Ha létezik szimulációs reláció A-ból B-be, akkor történetek$(A) \subseteq$ történetek(B).*

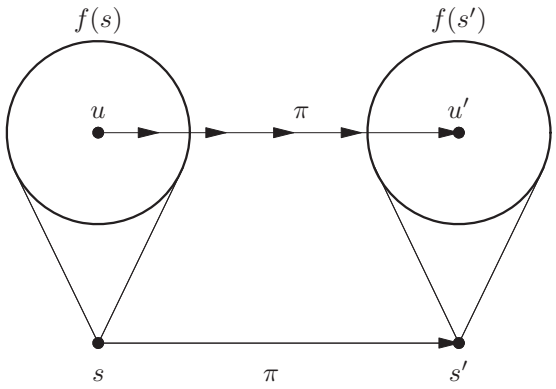
Bizonyítás. A bizonyítást meghagyjuk gyakorlatnak (lásd 8-16. gyakorlat). □

Sajátos esetben a 8.12. tételből következik, hogy ha B rendelkezik egy biztonságossági tulajdonsággal, akkor A is rendelkezik vele: ha P egy történetre vonatkozó biztonságossági tulajdonság, *külső_lenyomat*$(A) =$ *lenyomat*(P) és *történetek*$(A) \subseteq$ *történetek*(B), akkor *külső_lenyomat*$(B) =$ *lenyomat*(P) és *történetek*$(B) \subseteq$ *történetek*(P). A szimulációs reláción alapuló helyességbizonyítások annyira kifinomultak, hogy ajánlatos számítógép segítségét igénybe venni.

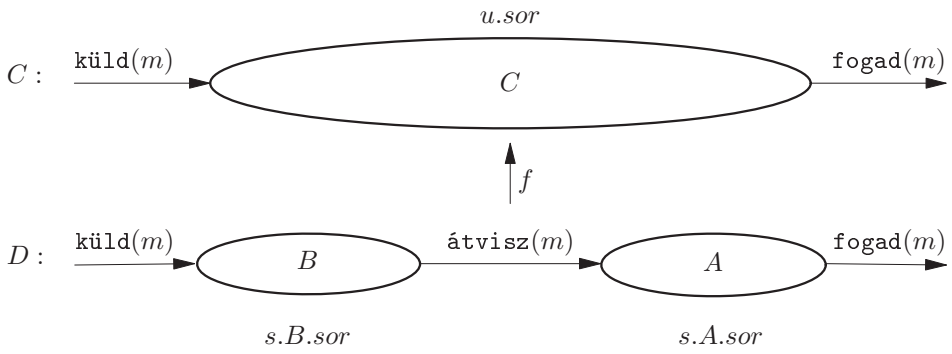
8.5.6. példa. Szimulációs bizonyítás

Egyszerű példaként a szimulációs bizonyításra, megmutatjuk, hogy két csatornaautomata segítségével újabb csatornaautomata valósítható meg.

Legyen C a 8.1.1. példabeli kommunikációs automata. (A következőkben elhagyjuk az indexeket.) Legyenek az A és B automaták azonosak C-vel, csupán bizonyos műveleteket átnevezünk. Így, a B kimenetei legyenek `átvisz`(m) az eredeti `fogad`(m) helyett, az A bemenetei pedig szintén `átvisz`(m) az eredeti `küld`(m) helyett. Legyen D az A



8.4.. ábra. Lépések megfeleltetése szimulációs műveletben.



8.5.. ábra. Az f szimulációs művelet D-ből C-be.

és B összekapcsolásának az eredménye, miután elrejtettük az `átvisz` műveleteket. Megjegyezzük, hogy C-nek és D-nek ugyanaz a külső felülete.

Azt állítjuk, hogy *történetek*$(D) \subseteq$ *történetek*(C). Hogy ezt bebizonyítsuk, definiáljuk az f szimulációs relációt D-ből C-be. (Lásd a 8.5. ábrát.)

Tehát, ha s a D-nek állapota és u a C-nek állapota, akkor legyen $(s, u) \in f$, feltéve, hogy a következő feltétel igaz. (Pontot használunk egy állapot változója értékének a jelölésére, akárcsak az automata jelölésére az összekapcsolásban.)

u.sor az *s.A.sor* és az *s.B.sor* összefűzése (konkatenálása) ebben a sorrendben.

Ahhoz, hogy belássuk, hogy f szimulációs reláció, ellenőriznünk kell

a definíció két feltételét. A kezdeti feltétel triviális, mivel az A, B és C automaták kezdőállapotai üres sorok. A lépésfeltétel igazolására, tételezzük fel, hogy s állapota D-nek, $u \in f(s)$ állapota C-nek és $(s, \pi, s') \in \mathit{átmenet}(D)$. Az elvégzendő műveletek típusától függően, a következő eseteket vizsgáljuk meg.

1. $\pi =$`küld`(m).
   Álljon a C megfelelő végrehajtási sorozatrészlete egyetlen `küld`(m) lépésből. Az adott lépés D-ben hozzáadja m-et az $s.B.sor$ végéhez, míg C-ben hozzáadja m-et az $u.sor$ végéhez. Ez megőrzi az állapotmegfeleltetést f definíciója alapján.
2. $\pi =$`fogad`(m).
   Álljon a C megfelelő végrehajtási sorozatrészlete egyetlen `fogad`(m) lépésből. Az adott lépés D-ben törli m-et az $s.A.sor$ elejéről. Az s és u közötti megfeleltetésből következik, hogy m az $u.sor$ elején is van, amelyből következik, hogy `fogad`(m) művelet u-ban is megengedett. Ekkor az adott lépés C-ben törli m-et az $u.sor$ elejéről. Ez szintén megőrzi az állapotmegfeleltetést f alapján.
3. $\pi =$`átvisz`(m).
   Álljon a C megfelelő végrehajtási sorozatrészlete 0 lépésből. Mivel D lépése nem befolyásolja a két sor összefűzését, az állapotmegfeleltetés megmarad.

Következik tehát, hogy f szimulációs reláció. Ebből pedig a 8.12. tétel alapján következik, hogy $\mathit{történetek}(D) \subseteq \mathit{történetek}(C)$, amit éppen bizonyítani kellett.

A szimuláció gyakran abban is segít, hogy bebizonyítsuk, hogy a B elevenségi tulajdonsága A-ra is vonatkozik. Az ötlet azon alapszik, hogy a szimulációs reláció A-ból B-be nemcsak történettartalmazást jelent, hanem egy szoros megfeleltetést is (bevonva a történeteket és állapotokat is) az A minden és B néhány végrehajtási sorozata között. Az ilyen szoros megfeleltetés, együtt a pártatlansággal A-ra nézve, néha felhasználható az elevenségi tulajdonság bizonyítására.

Példaként bemutatunk egy hasznos formális definíciót a végrehajtási sorozatok közötti erősebb megfeleltetésre. Legyenek A és B azonos bemeneti és kimeneti műveletekkel rendelkező b/k-automaták. Legyen α és α' az A, illetve a B egy-egy végrehajtási sorozata, és legyen f egy, az $\mathit{állapotok}(A)$ és $\mathit{állapotok}(B)$ feletti bináris reláció. Ekkor azt mondhatjuk, hogy α és α' *megfelelnek* egymásnak f szerint, feltéve ha létezik egy g leképezés az α-ban lévő állapotok indexei (előfordulásai) és az α'-ben lévő állapotok indexei között úgy, hogy igazak a következők:

1. g monoton növekvő;
2. g kimeríti az α' végrehajtási sorozatot (azaz a g értékkészletének szuprémuma egyenlő az α'-ben szereplő állapotok indexeinek szuprémumával);
3. a g-megfeleltetés szerinti állapotpárok f által kapcsolódnak egymáshoz;

4. a g-megfeleltetés szerinti egymást követő állapotpárok között az α és α' történetei azonosak.

Nem nehéz belátni, hogy egy szimulációs reláció ilyen típusú megfeleltetést eredményez.

8.13. tétel. . *Ha f szimulációs reláció A-ból B-be, akkor A minden α végrehajtási sorozatához létezik B-nek egy α' végrehajtási sorozata úgy, hogy α és α' megfelelnek egymásnak f szerint.*

A 8.13. tétel felhasználható annak bizonyítására, hogy A rendelkezik egy élénkségi tulajdonsággal, ha B is rendelkezik azzal. Felhasználjuk ezt a stratégiát, például, a kölcsönös kizárás (JEGYKK, 10.40. tétel) és az adatvonal protokoll (Stenning, 22.2. lemma) bizonyításvázlataiban.

8.6.. Bonyolultsági mértékek

Annak ellenére, hogy a b/k-automata aszinkron modell, mégis természetes módon definiálható az időbonyolultsága. Egy adott A automata esetében a *taszkok*(A) taszkpartícionálás ekvivalenciaosztályai minden részhalmazára felső időkorlátot adunk meg. Pontosabban, minden C taszkra megadunk egy *felső*$_C$ korlátot, amely pozitív valós szám vagy végtelen. Az A automata bármely α pártatlan végrehajtási sorozatának minden eseményéhez hozzárendelhetünk egy valós értékű időpontot, amelyre igazak a következők.

1. Az időpontok α-ban monoton növekvők.
2. Ha α végtelen, akkor az időpontok tartanak ∞-hez.
3. α bármely pontjától kezdődően, egy C taszk legfeljebb *felső*$_C$ időpontig megengedett mielőtt valamilyen C-beli művelet megjelenik.

Durván szólva, ez egy *felső*$_C$ felső időhatárt jelent a C taszk két egymás utáni esélye között egy lépésének a végrehajtására. Egy pártatlan végrehajtási sorozat, amelyet időpontokkal látunk el, egy *időzített végrehajtási sorozat*.

Megjegyezzük, hogy az adott *felső*$_C$ korlátok egy halmaza esetében, nagyon sokféleképpen rendelhetünk az α eseményeihez időpontokat, vagyis sokféle időzített végrehajtási sorozat lehetséges. Az α egy adott π eseményéig eltelt *időt* úgy mérjük, hogy vesszük az összes lehetséges időzített végrehajtási sorozatban a π-hez rendelhető időpontok szuprémumát. Hasonlóképpen mérjük az α két eseménye között eltelt *időt* is, mint a két eseményhez rendelhető időpontok szuprémumainak a különbsége.

8.6.1. példa. Időbonyolultság elemzése

Legyen α a 8.2.1. példabeli rendszer egy tetszőleges pártatlan végrehajtási sorozata, amelyben minden folyamatnak vannak `kezd` bemenetei. Minden folyamat minden taszkjához hozzárendelünk egy ℓ felső korlátot, és minden csatorna egyetlen taszkjához egy d felső korlátot. Ekkor, attól kezdve, hogy az utolsó folyamat fogadja a saját első `kezd`

bemenetét α-ban egészen addig, amíg az összes folyamat végrehajtja a `dönt` kimenetet, legfeljebb $\ell + d + \ell = d + 2\ell$ időegység telik el. Ez azért van így, mert legfeljebb ℓ időegység telik el addig, amíg az utolsó folyamat végrehajtja a `küld` eseményeket az összes szomszédja részére, miután fogadott egy `kezd` bemenetet. Azután legfeljebb d időegység telik el, amíg az összes üzenet elmegy, majd ismét legfeljebb ℓ időegység, amíg minden folyamat végrehajtja a `dönt` műveletet.

8.7.. Megkülönböztethetetlen végrehajtási sorozatok

Definiáljuk a megkülönböztethetetlenség fogalmát, amely sok lehetetlenség bizonyításában lesz hasznos. Ez a fogalom hasonló ahhoz, amelyiket a 2.4. alfejezetben adtunk meg a szinkron rendszerek végrehajtási sorozata esetében.

Ha α és α' két összetett automata egy-egy végrehajtási sorozata úgy, hogy mindkettő tartalmazza A-t, akkor azt mondjuk, hogy α és α' *megkülönböztethetetlenek* A-ra nézve, ha $\alpha|A = \alpha'|A$.

8.8.. Véletlenítés

Mint a szinkron rendszerek esetében is, néha hasznos, ha megengedjük az aszinkron rendszerek komponenseinek a véletlenszerű választás lehetőségét, bizonyos valószínűségi eloszlás alapján. Az ilyen véletlenszerű választások modellezésére kiterjesztjük a b/k-automata modelljét egy új, *valószínűségi b/k-automata* modelljére. A valószínűségi b/k-automata olyan, mint egy b/k-automata, azzal a különbséggel, hogy az átmenet fogalmát kicsit módosítjuk: az átmenet az (s, π, s') hármas helyett egy (s, π, P) alakú hármas, ahol P egy valószínűségi eloszlást jelent az állapothalmaz részhalmazai felett. (Ha egy lépésben nincs szükség véletlen választásra, akkor triviális P eloszlást használunk.) Bármely A valószínűségi b/k-automatához hozzárendelhető annak egy $\mathcal{N}(A)$ *nemdeterminisztikus* változata, amelyet úgy kapunk meg, hogy minden (s, π, P) átmenetet helyettesítünk az összes (s, π, s') alakú elemmel, ahol s' a P értelmezési tartomány eleme. Tehát, $\mathcal{N}(A)$ tulajdonképpen egyszerűen csak helyettesíti a véletlen választást a nemdeterminisztikussal. $\mathcal{N}(A)$ egy közönséges b/k-automata.

Az A valószínűségi b/k-automata egy végrehajtási sorozata választáspárok sorozata alapján történik. Minden pár esetében, először egy nemdeterminisztikus választás történik, amellyel meghatározzuk a következő (s, π, P) átmenetet, aztán pedig egy véletlen választás, amikor P segítségével meghatározzuk a következő állapotot. A választásban egyetlen megszorítás van: a következő átmenet nemdeterminisztikus kiválasztásának „pártatlannak" kell lennie abban az értelemben, hogy az $\mathcal{N}(A)$ b/k-automatának az összes lehetséges véletlen választási sorozat által generált végrehajtási sorozatai mind pártatlanok legyenek.

Akárcsak a szinkron esetben, a véletlenített rendszer számításaira tett feltevéseink általában valószínűségiek. Amikor egy feltevést megfogalmazunk, általában az a célunk, hogy az igaz legyen minden bemenetre és a nemdeterminisztikus választások minden pártatlan mintájára. Akárcsak az 5. fejezetben, itt is általában

egy képzelt *ellenfelet* használunk a bemenetek és nemdeterminisztikus választások leírására. Az automatának jól kell viselkednie tetszőleges ellenféllel szemben.

8.9.. Megjegyzések a fejezethez

A b/k-automata modellje először Tuttle diplomamunkájában [217] jelenik meg. A modell fontosabb jellemzőit Lynch és Tuttle foglalja össze [217, 218]. A b/k-automatákkal modellezett algoritmusok leírása és a megfelelő bizonyítások szétszórtan találhatók az osztott algoritmusokról szóló szakirodalomban. Néhány jellegzetes példa megtalálható pl. Afek és társai, valamint Bloom munkáiban [3, 4, 53]. Példa arra, hogy hogyan használható a b/k-automata olyan rendszerek modellezésére, amelyek dinamikus folyamatok segítségével keretet teremtenek az adatbázisok párhuzamosságvezérlő algoritmusai modellezésére Lynch, Merritt, Weihl és Fekete *Atomic Transactions* [207] c. könyvében található. A b/k-automata modelljét sok más konkurens rendszer modellje befolyásolta, ezek közül a legjelentősebb a Lynch és Fischer [216] aszinkron közös memóriájú modellje, Hewitt actor-modellje [7, 81], valamint Hoare-nak az egymással kommunikáló szekvenciális folyamatok modellje (CSP) [159].

Az invariáns állítások fogalmának az eredetéről már írtunk a 2. fejezet végén. A történetre vonatkozó tulajdonságot [217, 218] alapján az „ütemezési modul" definíciójából származtattuk. A biztonságossági és elevenségi tulajdonságok Lamport [175], Alpern és Schneider [8] munkáin alapszanak. A 8.9. tétel [8]-ból származik.

Manna és Pnuelli könyve [219] jó referencia a temporális logikára. Lamport könyve [184] egy hasznos temporális logikai keretet tartalmaz, és egy jól kidolgozott módszert ennek alkalmazására az algoritmusok helyességének bizonyításában.

Az a stratégia, amely megmutatja, hogy egy műveletsorozat levetítése megadja egy A rendszer komponenseinek a történeteit, és amelynek segítségével bebizonyítjuk, hogy ez a sorozat tulajdonképpen az egész rendszer története, megtalálható [207]-ben. Ebben a könyvben az A rendszer az összes tranzakciót sorosan végrehajtó adatbázisrendszer absztrakt leírása. Bebizonyították, a vetületek vizsgálatával, hogy a tranzakciókat párhuzamosan végző adatbázisok bizonyos műveletsorozatai éppen az A történetei. Az ilyen típusú adatbázisok helyességének ez a biztosítéka. A biztonságossági tulajdonságok megőrzésének vizsgálata [218]-ból származik.

A szimulációs reláció fogalma sok forrásból származik. Ez a Lamport által használt [177] finomításon alapuló leképezés fogalmának az általánosítása. Ugyancsak absztrakciója az Owicki és Gries [235] által használt történeti változónak. Ez a fogalom nagyon hasonlít a következőkre is: Park szimulációi [236], Lynch [203, 214], valamint Lynch és Tuttle [217, 218] lehetőségeken alapuló leképezései, Jonsson szimulációi [165]. Az aszinkron rendszerek szimulációs módszere a biztonságossági tulajdonságok igazolására ma már igen jól megalapozott. Sok cikk és könyv, mint például [217, 288, 69, 233, 214, 207, 189, 190], értékes példákat tartalmaznak ennek a módszernek az alkalmazására. Lényeges számú szimulá-

ciós bizonyításnál és ellenőrzésnél használtak számítógépes támogatást és ellenőrzést. Megemlítjük Nipkow [265], valamint Søgaard-Andersen, Garland, Guttag, Lynch és Pogosyants [265] munkáit mint tipikus példákat. Az első az Isabelle-tételbizonyítót, míg a második a Larch-tételbizonyítót tartalmazza. A véletlenített rendszerek modellje Segala és Lynch munkájából [257] való.

A párhuzamos rendszerek általános modelljeiről sok eredmény található az *International Conference on Concurrency Theory* (CONCUR) évenkénti konferencia anyagaiban.

8.10.. Gyakorlatok

8-1. Tekintsük a 8.2.1. példában levő P_i $C_{i,j}$, $1 \leq i, j \leq n$, automaták összekapcsolását.

(a) Írjuk le a példában $n = 2$-re megadott történet egyetlen végrehajtási sorozatában szereplő összes állapotot.

(b) Legyen most $n = 3$ és m tetszőleges természetes szám. Írjunk le egy végrehajtási sorozatot, amelyben m döntési műveletben jelenik meg mindhárom folyamatban. Ebben a végrehajtási sorozatban, az egymást követő $\texttt{kezd}_1$ értékeknek a $0, 4, 8, 12, \ldots$, sorozat egy kezdőszeletét kell képezniük, ugyanígy a $\texttt{kezd}_2$ értékeknek a $0, 2, 0, 2, \ldots$, sorozat egy kezdőszeletét, míg végül a $\texttt{kezd}_3$ értékeknek a $0, 1, 0, 1, \ldots$, sorozat egy kezdőszeletét.

(c) Legyen újra $n = 3$, és legyen m_1, m_2, m_3 tetszőleges természetes szám. Írjunk le egy olyan végrehajtási sorozatot, amelyben m_i a P_i, $i \in \{1, 2, 3\}$ egy döntési műveletében szerepel. Az egymást követő `kezd` értékeknek a három folyamatban a fenti (b) pont szerintieknek kell lenniük.

8-2. Bizonyítsuk be a 8.1., 8.2. és 8.3. tételeket. Hol használjuk a kompatibilitási feltételeket?

8-3. Bizonyítsuk be a 8.4., 8.5. és 8.6. tételeket. Hol használjuk a kompatibilitási feltételeket? Hol használjuk a megengedett bemenetűséget?

8-4. Tekintsük a következő két b/k-automatát. Megjegyezzük, hogy leírásukban nem használjuk az előfeltétel/hatás jelölést, hanem egyszerűen csak leírjuk a komponenseiket.

- A automata:

  $be(A)$=*belső*$(A) = \emptyset$, $ki(A) = \{\texttt{menj}\}$,

  állapotok$(A) = \{s, t\}$;

  kezdő$(A) = \{s\}$;

  átmenetek$(A) = \{(s, \texttt{menj}\ , t)\}$, és

  taszkok$(A) = \{\{\texttt{menj}\}\}$.

- B automata:

  $be(B) = \{\texttt{menj}\}$, $ki(B) = \{\texttt{nyugtáz}\}$, *belső*$(B) = \{\texttt{növel}\}$,

  állapotok$(B) = \{be, ki\} \times \mathbb{N}$,

$$\begin{aligned}
&\mathit{kezdő}(B) = \{(be, 0)\},\\
&\mathit{átmenetek}(B) = \{((be, i), \texttt{növel}, (be, i+1)),\ i \in \mathbb{N}\} \cup\\
&\quad \{((be, i), \texttt{menj}, (ki, i)),\ i \in \mathbb{N}\} \cup\\
&\quad \{((ki, i), \texttt{menj}, (ki, 0)),\ i \in \mathbb{N}\} \cup\\
&\quad \{((ki, i), \texttt{nyugtáz}, (ki, i-1)),\ i \in \mathbb{N} - \{0\}\}, \text{ és}\\
&\mathit{taszkok}(B) = \{\{\texttt{növel}\}, \{\texttt{nyugtáz}\}\}.
\end{aligned}$$

Adjuk meg az A, B és $A \times B$ automatákra a történetek és a pártatlan történetek halmazát.

8-5.

(a) Definiáljunk egy A b/k-automatát, amely egy megbízható üzenetcsatornát ábrázol, amely üzeneteket fogad és kézbesít az M_1 és M_2 ábécék egyesítésén. Az üzenetcsatornát olyannak tételezzük fel, amely megőrzi az azonos ábécébeli üzenetek sorrendjét. Továbbá, ha egy M_1-beli üzenetet előbb küldtünk el, mint egy másikat M_2-ből, akkor a kézbesítéseknek is ebben a sorrendben kell megtörténniük. Viszont, ha egy M_1-beli üzenetet később küldtünk el, mint egy M_2-belit, akkor a kézbesítés történhetik fordítva is. A definiált automatának valójában képesnek kell lennie az összes megengedett külső viselkedés megvalósítására. Vigyázzunk, hogy adjuk meg az A automata összes komponensét: lenyomat, állapotok, kezdő állapotok, lépések és taszkok.

(b Az előző pontban definiált automata esetében adjunk példát a következőkre: pártatlan végrehajtási sorozat, pártatlan történet, nem pártatlan végrehajtási sorozat, nem pártatlan történet.

8-6. Írjunk le egy b/k-automatát, amelynek nincs bemeneti művelete, a kimeneti műveletei pedig a $\{0, 1, 2, \ldots\}$ halmazból valók, és amelynek a pártatlan történetei az S halmazból való sorozatok, amelyet a következőképpen definiálunk. Az S halmaz a kimeneti halmazon definiált összes 1 hosszúságú sorozatból áll, vagyis egyetlen nemnegatív egészből álló sorozatokból.

8-7. Bizonyítsuk be a 8.7. tételt.

8-8. Legyen A egy tetszőleges b/k-automata. Bizonyítsuk be, hogy létezik egy másik B b/k-automata, egyetlen taszkkal úgy, hogy $\mathit{pártatlan\_történetek}(B) \subseteq \mathit{pártatlan\_történetek}(A)$. (Nem kell egyenlőséget bizonyítani, elég a tartalmazást.)

8-9. Legyen A egy egyetlen taszkú b/k-automata. Mutassuk meg, hogy létezik egy másik, szintén egyetlen taszkú B b/k-automata, amely „determinisztikus" abban az értelemben, hogy fennállnak rá a következők:

(a) egyetlen kezdőállapota van;

(b) bármely s állapotára és π műveletére, legfeljebb egy (s, π, s') alakú átmenete van;

(c) bármely állapotában legfeljebb egy, helyileg ellenőrzött művelete megengedett.

Ezenfelül még *pártatlan_ történetek*$(B) \subseteq$ *pártatlan_ történetek*(A) is fennáll. (Nem kell egyenlőséget bizonyítani, elég a tartalmazást.)

8-10. Fogalmazzunk meg egy olyan tételt, amely egyesíti a 8-8. és 8-9. gyakorlatok eredményeit. Bizonyítsuk be ezt a tételt.

8-11. Vizsgáljuk felül a 8-8., 8-9. és 8-10. gyakorlatokat, ha a *pártatlan_ történetek*$(B) =$ *pártatlan_ történetek*(A) egyenlőséget is megköveteljük. Ha ezeket meg lehet oldani ezzel az erősebb feltétellel is, oldjuk meg, ha nem, akkor bizonyítsuk be, hogy nem lehetet megoldani őket.

8-12. Ha P biztonságossági tulajdonság, bizonyítsuk be, hogy a következő három állítás egyenértékű egy adott A b/k-automatára:

(a) *történetek*$(A) \subseteq$ *történetek*(P);

(b) *pártatlan_ történetek*$(A) \subseteq$ *történetek*(P);

(c) az A véges történetei mind benne vannak *történetek*(P)-ben.

8-13. Tekintsük a következő, történetre vonatkozó P tulajdonságokat. Mindegyik esetben a *lenyomat*(P) lenyomatnak nincs bemenete, kimenete pedig az $\{1, 2\}$ halmaz.

(a) Tételezzük fel, hogy *történetek*(P) az $\{1, 2\}$ halmaz feletti olyan sorozatok halmaza, amelyekben nincs 1 után közvetlenül 2. Mutassuk meg, hogy P biztonságossági tulajdonság.

(b) Tételezzük fel, hogy *történetek*(P) az $\{1, 2\}$ halmaz feletti olyan sorozatok halmaza, amelyekben minden 1-est követ valahol egy 2. Mutassuk meg, hogy P elevenségi tulajdonság.

(c) Tételezzük fel, hogy *történetek*(P) az $\{1, 2\}$ halmaz feletti olyan sorozatok halmaza, amelyekben minden 1-est közvetlenül követ egy 2. Mutassuk meg, hogy P sem biztonságossági, sem elevenségi tulajdonság. Mutassuk meg konkrétan, hogyan lehet kifejezni P-t egy biztonságossági és egy elevenségi tulajdonság metszeteként.

8-14. Fogalmazzunk meg pontos definíciókat a végrehajtási sorozatra vonatkozó biztonságossági és elevenségi tulajdonságokra, hasonlóan mint a történetekre tettük. Bizonyítsunk be a 8.8. és 8.9. tételekhez hasonlókat.

8-15. Bizonyítsuk be a 8.11. tételt.

8-16. Bizonyítsuk be a 8.12. tételt.

8-17. Tekintsük a 8.1.1. példa C csatornaautomatájához hasonló D automatát. A különbség abban áll, hogy D megengedi a belső üzenetek többszörözését.

Pontosabban, a `küld` és `fogad` üzenetek mellett D-nek van még két belső művelete, `többszöröz` és `eldob`. Amikor a `küld`(m) megjelenik, az m üzenet a sor végére kerül, együtt egy logikai jelzővel. Az egymásután küldött üzenetek jelzői váltakozóak: $1, 0, 1, 0 \ldots$. A `többszöröz` művelet hatására a sor egy tetszőleges üzenete helyben megkétszereződik a jelzőjével együtt. Az automata megőrzi az legutóbb kézbesített üzenet jelzőjét. Egy `fogad` művelet kézbesíti a sor első

üzenetét, feltéve ha annak a jelzője különbözik a legutóbb kézbesített üzenet jelzőjétől. Az `eldob` művelet eldobja a sor első üzenetét, feltéve ha annak a jelzője egyezik a legutóbb kézbesített üzenet jelzőjével.

(a) Adjuk meg a D automata formális leírását, abban a stílusban, ahogy azt ebben a fejezetben más automatákra is tettük.

(b) Bizonyítsuk be, hogy D megvalósítja C-t, a történethalmazok tartalmazása értelmében. Használjunk szimulációs relációkat.

III.A. ASZINKRON KÖZÖS MEMÓRIÁJÚ ALGORITMUSOK

A következő néhány (9–13.) fejezet az *aszinkron közös memóriájú modell* algoritmusaival foglalkozik, amelyekben a folyamatok aszinkron módon hajtják végre lépéseiket és közös memória révén érintkeznek.

Ebben a részben az első, azaz a 9. fejezet egyszerűen bemutatja az aszinkron közös memóriájú rendszerek formális modelljét. Ahogy azt korábban tettük, ezt a fejezetet lapozzuk át, és a későbbiekben használjuk referenciaként. A 10. fejezet a *kölcsönös kizárás* alapvető, a 11. fejezet pedig az *osztott erőforrás-hozzárendelés* általánosabb feladatával foglalkozik. A 12. fejezet a *megegyezés* szerinti alapvető eredményeket tartalmazza a hibára hajlamos aszinkron rendszerekre vonatkozóan. Végül a 13. fejezet egy tanulmányt tartalmaz az *atomi objektumokról*, amelyek hatékony absztrakt objektumok az osztott rendszerek programozására.

9. fejezet

Modellezés/III. Aszinkron közös memóriájú modell

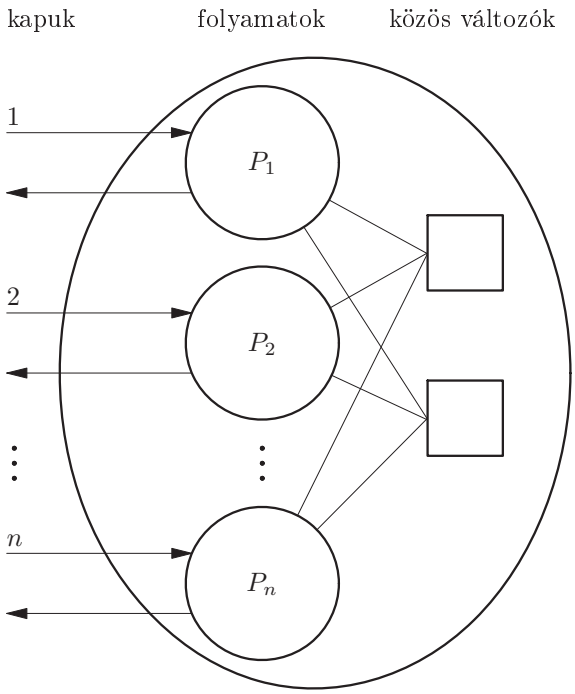
Ebben a fejezetben az aszinkron közös memóriájú rendszerek egy formális modelljét adjuk meg. A modellt az aszinkron rendszerek általános b/k automata modelljében megadva mutatjuk be, amelyet a 8. fejezetben vezettünk be.

A közös memóriájú rendszer kommunikáló folyamatok halmazából áll, hasonlóan a hálózati rendszerhez. Ezúttal azonban a folyamatok pillanatnyi műveleteiket nem a kommunikációs csatornán áthaladó küldő és fogadó üzenetekkel, hanem az közös változókon végzik el.

9.1.. Közös memóriájú rendszerek

Általánosan megfogalmazva, az *aszinkron közös memóriájú rendszer* folyamatok véges halmazából áll, amelyek közös változók véges halmaza révén hatnak egymásra. A rendszerben a változókat csak a folyamatok közötti kommunikációra használjuk. Mivel a külvilág és a közös memóriájú rendszer képes egymásra hatni, mi is feltesszük, hogy minden folyamat rendelkezik egy *kapuval*, amelyen képes kölcsönhatásba lépni a külvilággal, bemeneti és kimeneti műveleteket használva. A kölcsönhatásokat a 9.1. ábra mutatja.

A közös memóriájú rendszert b/k automatával modellezzük, valójában azonban csak egyetlen b/k automatát használunk külső felülettel, amely bemeneti és kimeneti műveletekből áll az összes kapun. Természetesebbnek tűnne minden folyamathoz és közös változóhoz egy-egy automatát használni. Ez azonban némi bonyodalomhoz vezetne, ezért ezzel ebben a könyvben nem foglalkozunk. Például, ha minden folyamat és minden közös változó egy-egy b/k automata lenne és a szokásos módon összekapcsolnánk őket, akkor egy olyan rendszert kapnánk, amelyben a P_i folyamat művelete az x közös változón egy eseménypárral lenne modellezve – egy kéréssel, azaz a P_i folyamat egy kimenetével és az x változó egy bemenetével, és egy azt követő válasszal, azaz az x változó egy kimenetével és a P_i folyamat egy bemenetével. De ekkor a rendszernek is lennének olyan végrehajtásai, amelyekben ezek az eseménypárok kettészakadnak. Előfordulhat, hogy



9.1.. ábra. Egy aszinkron közös memóriájú rendszer.

néhány művelet még azelőtt hívódik meg, mielőtt az első befejeződött volna. Ez a fajta viselkedés nem fordulhat elő a közös memóriájú rendszerek esetében. Ezt próbáljuk meg modellezni.

Az egyik kiút ebből a problémából az lehet, hogy megvizsgáljuk az összes lehetséges végrehajtás egy leszűkített részhalmazát, azokat, amelyekben a kérések és a megfelelő válaszok egymás után következnek. Egy másik kiút, hogy csak a folyamatokat modellezzük b/k automataként, az közös változókat pedig más módon (a kéréseket és a válaszokat egy eseménynek tekintjük), állapotautomatákkal. Ekkor egy új összekapcsolás műveletet kell bevezetnünk, hogy a folyamat és változó automatát egy b/k automatába lehessen egyesíteni. Ezek a megközelítések bonyolult fogalmakat vezetnek be (végrehajtások halmazának szűkített részhalmazai, eseménypárok, új fajta állapotautomaták vagy egy új művelet), ezeket azzal kerüljük ki, hogy az egész rendszert egy nagy, A-val jelölt b/k automatával modellezzük. A folyamatok és változók szerkezetét az A automatán belüli eseményekhez tartozó helyi megszorításokkal írjuk le.

Ahogy a szinkron hálózati modellnél, feltételezzük, hogy a rendszerben a folyamatok 1-től n-ig sorszámozottak. Tegyük fel, hogy minden P_i folyamat rendelkezik az *állapotok* egy kisegítő halmazával (*állapotok*$_i$) és az állapotok közül néhány *kezdőállapotként* kitüntetett (*kezdő*$_i$). Azt is feltesszük, hogy a rendszer minden x közös változója rendelkezik az *értékek* egy kisegítő halmazával

(*értékek*$_x$), és az értékek közül néhány *kezdőértékként* kitüntetett (*kezdeti*$_x$). Ekkor az *állapotok*(A) halmazban minden állapot (az A rendszerautomata állapotainak halmaza) a következőkből áll: egy, az *állapotok*$_i$-ből vett állapotból minden P_i folyamatra, valamint egy, az *értékek*$_x$ halmazból vett értékből minden x közös változóra. Minden *kezdő*(A) halmazbeli állapot a következőkből áll: egy *kezdő*$_i$-ből vett állapotból minden P_i folyamatra, valamint egy, a *kezdeti*$_x$ halmazbeli értékből minden x közös változóra.

Tegyük fel, hogy minden művelet a *műveletthalmaz*(A) halmaz egyik folyamatához kapcsolódik. Ezenkívül néhány belső művelet kapcsolódhat a *belső*(A) halmazból egy közös változóhoz. A P_i folyamathoz kapcsolódó bemeneti és kimeneti műveleteket a P_i folyamat és a külvilág közötti kölcsönhatásra használjuk, és azt mondjuk, hogy megjelennek az i *kapun*. A P_i folyamat belső műveleteit, amelyek nem rendelkeznek kisegítő közös változóval, helyi számításra, a P_i azon belső műveleteit pedig, amelyek rendelkeznek x kisegítő közös változóval, az x-en végzendő műveletek elvégzéséhez használjuk.

Az átmenetek *átmenet*(A) halmaza néhány helyi megszorítással modellezi a rendszerbeli folyamatok és közös változók szerkezetét. Először vizsgáljunk meg egy π műveletet, amely a P_i folyamathoz kapcsolódik, de nincs változója. Ahogy fentebb megjegyeztük, a π műveletet helyi számításra használjuk. Ekkor a π művelet bármely lépésében csak a P_i állapota szerepelhet. Ez azt jelenti, hogy a π átmenetek halmaza az (s, π, s') hármasok néhány halmazából állítható elő, ahol $s, s' \in$ *állapotok*$_i$, oly módon, hogy a többi folyamat állapotának és az közös változók értékeinek tetszőleges együttesét kapcsolja az s és s' állapotokhoz (ugyanazt az együttest mindkettőhöz).

Másrészről vizsgáljunk meg egy π műveletet, amely a P_i folyamathoz és egy x változóhoz kapcsolódik. Ahogy korábban megjegyeztük, a π műveletet a P_i az x-en való művelet elvégzésére használja. Ekkor a π művelet bármely lépésében csak a P_i állapota és az x értéke szerepelhet. Ez azt jelenti, hogy a π átmenetek halmaza az $((s, v), \pi, (s', v'))$ hármasok néhány halmazából állítható elő ($s, s' \in$ *állapotok*$_i$ és $v, v' \in$ *értékek*$_i$), oly módon, hogy a többi folyamat állapotának és a többi közös változó értékeinek tetszőleges együttesét kapcsolja össze. A következő technikai nehézség merül fel: ha a π művelet a P_i folyamathoz és az x változóhoz kapcsolódik, a π akár megengedett, akár nem, csak a P_i folyamat állapotától kellene függenie, bár a keletkező változások függhetnek x értékétől is. Ez azt jelenti, hogy ha a P_i s állapota és az x v értéke mellett π megengedett, a π megengedett a P_i s állapota és x tetszőleges v' értéke mellett is.

A feladat felosztása halmazának (*feladatok*(A)) illeszkednie kell a folyamatok szerkezetéhez, azaz minden ekvivalenciaosztálynak (feladatnak) pontosan egy folyamat helyi vezérlésű műveleteit kell tartalmaznia. Azon esetek többségében, amelyeket vizsgálni fogunk, folyamatonként pontosan egy feladat lesz. Ennek például akkor van értelme, ha minden folyamat egy szekvenciális program. Ekkor a általános meghatározása b/k automatára (amelyet a 8.3. alfejezetben adtunk meg), azt mondja, hogy minden folyamat végtelen sokszor kap lehetőséget, hogy lépéseket tegyen. Általánosabb esetben, amikor több feladat tartozhat egy folyamathoz, a pártatlanság meghatározása azt mondja, hogy minden feladat végtelen sok lehetőséget kap, hogy lépéseket tegyen.

9.1.1. példa. Közös memóriájú rendszer

Legyen V egy rögzített értékkészlet. Tekintsük az A közös memóriájú rendszert, amely 1-től n-ig számozott folyamatokból, valamint egyetlen $V \cup \{ismeretlen\}$ értékkészletből vett x közös változóból áll, amelynek kezdőértéke *ismeretlen*. A bemenetek a $\texttt{kezd}(v)_i$ alakúak, ahol $v \in V$ és i egy folyamat indexe. A kimenetek a $\texttt{választ}(v)_i$, a belső műveletek pedig a $\texttt{hozzáfér}_i$ alakúak. Minden i-vel indexelt művelet a P_i folyamathoz kapcsolódik és ráadásul a `hozzáfér` műveletek az x változóhoz kapcsolódnak.

Miután a P_i folyamat egy $\texttt{kezd}(v)_i$ bemenetet fogad, kiolvassa x értékét. Ha $x = ismeretlen$, akkor v-t értékül adja x-nek és v-t választja. Ha $x = w$, ahol $w \in V$, akkor x értékét nem változtatja, de w-t választja.

Formálisan, minden $állapotok_i$ halmaz helyi változókat tartalmaz.

P_i állapotai:
státus $\in \{tétlen, hozzáférés, választás, elvégzett\}$, kezdetben *tétlen*
bemenet $\in V \cup \{ismeretlen\}$, kezdetben *ismeretlen*
kimenet $\in V \cup \{ismeretlen\}$, kezdetben *ismeretlen*
P_i átmenetei:

$\texttt{választ}(v)_i$
Előfeltétel:
státus = *választás*
kimenet = v
Hatás:
státus := *elvégzett*

$\texttt{kezd}(v)_i$
Hatás:
bemenet := v
if *státus* = *tétlen* **then**
státus := *hozzáférés*

$\texttt{hozzáfér}_i$
Előfeltétel:
státus = *hozzáférés*
Hatás:
if $x = ismeretlen$ **then**
x := *bemenet*
kimenet := x
státus := *választás*

Folyamatonként egy feladat van, amely az összes, az adott folyamatra vonatkozó `hozzáfér` és `választ` műveletet tartalmazza.

Könnyen látható, hogy A minden α pártatlan végrehajtására bármely olyan folyamat, amely egy `kezd` bemenetet fogad, végül egy `választ` kimenetet eredményez. Sőt, minden végrehajtás (akár pártatlan, akár nem, és bárhol, bármennyi `kezd` esemény jelenik meg) rendelkezik az „egyetértési tulajdonsággal”, azaz különböző értékeken azonosak a döntési folyamatok, és az „érvényességi tulajdonsággal”, azaz minden választási érték valamelyik folyamat kezdőértéke.

Ezeket a pártatlansági igényeket történeti tulajdonságok által, a 8.5.2. szakaszban leírt meghatározás alapján fogalmazhatjuk meg. Például legyen P olyan történeti tulajdonság, ahol *lenyomat*(P) = *külső_lenyomat*(A) és a *történetek*(P) a *műveletalmaz*(P) halmaz β műveleteiből álló sorozatok halmaza, amely teljesíti az alábbi feltételeket:

1. Minden i-re, ha pontosan egy $\texttt{kezd}_i$ esemény szerepel β-ban, akkor pontosan egy $\texttt{választ}_i$ esemény szerepel β-ban.
2. Minden i-re, ha egyetlen $\texttt{kezd}_i$ esemény sem szerepel β-ban, akkor $\texttt{választ}_i$ esemény sem szerepel β-ban.
3. (Egyetértés) Ha $\texttt{választ}(v)_i$ és $\texttt{választ}(w)_j$ is szerepel β-ban, akkor $v = w$.
4. (Érvényesség) Ha egy $\texttt{választ}(v)_i$ esemény szerepel β-ban, akkor szerepel $\texttt{kezd}(v)_j$ esemény β-ban (v megegyezik).

Meg lehet mutatni, hogy *pártatlan_ történetek*$(A) \subseteq$ *történetek*(P). A bizonyítás a gyakorlatok között szerepel (lásd 9-1.(a) gyakorlat).

9.2.. Környezeti modell

Néha hasznos a rendszer környezetét is automataként modellezni. Ez egyszerű lehetőséget ad arra, hogy leírjuk feltételezéseinket a környezet viselkedéséről. A 9.1.1. példában ezt úgy határozhatnánk meg, hogy a környezet *pontosan egy* $\texttt{kezd}_i$ bemenetet ad minden i-re, vagy *legalább egyet* minden i-re. A közös memóriájú rendszerek esetében a gyakorlat azt mutatja, hogy a környezetet gyakran független *felhasználói automaták* halmazaként írhatjuk le, kapunként egy automatával.

9.2.1. példa. Környezeti modell

Most a 9.1.1. példában leírt A közös memóriájú rendszer egy környezetét írjuk le. A környezet egyetlen b/k automata, amely minden P_i folyamatra egy U_i *felhasználói automatából* áll. (Az automata létrehozásához használjuk fel a 8.2.1. szakaszban a b/k automatára bevezetett összekapcsolás műveletet.) Az U_i kódja a következő.

9.1. automata. U_i

Lenyomat:

Bemeneti:
`választ`$(v)_i, v \in V$

Kimeneti:
`kezd`$(v)_i, v \in V$

Belső:
`henyél`$_i$

Állapotai:

státus $\in \{$*kérés*, *várakozás*, *elvégzett*$\}$, kezdetben *kérés*
választás $\in V \cup \{$*ismeretlen*$\}$, kezdetben *ismeretlen*
hiba, egy logikai érték, kezdetben *hamis*

Átmenetek:

`kezd`$(v)_i$
Előfeltétel:
státus = *kérés* **or** *hiba* = *igaz*
Hatás:
if *hiba* = *hamis* **then**
státus := *várakozás*

`választ`$(v)_i$
Hatás:
if *hiba* = *hamis* **then**
if *státus* = *várakozás* **then**
választás := v
státus := *elvégzett*
else *hiba* := *igaz*

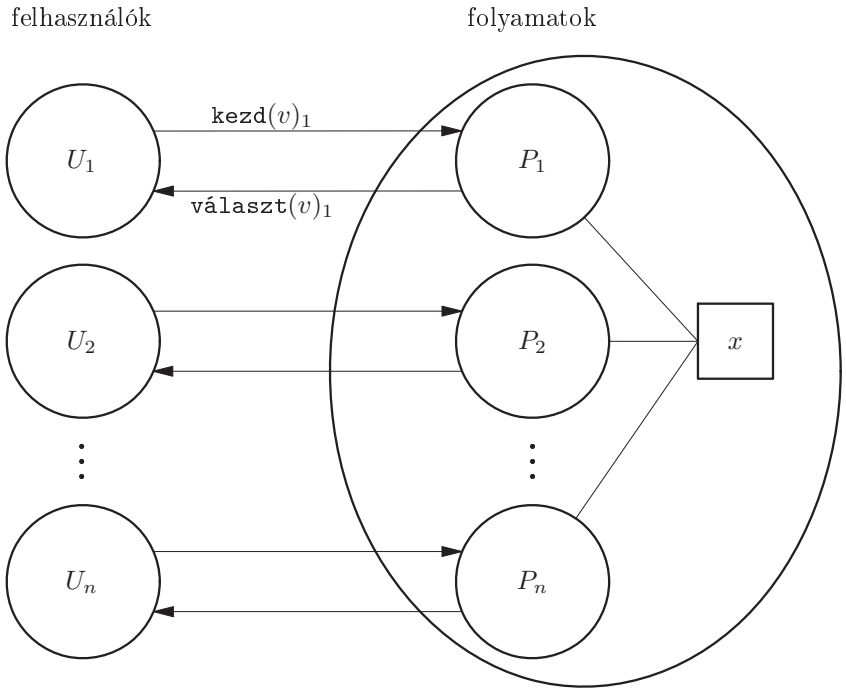
`henyél`$_i$
Előfeltétel:
hiba = *igaz*
Hatás:
Nincs

Taszkok:
Az összes helyi irányítású művelet egy osztályban van.

- -

Így U_i kezdetben végrehajt egy $\texttt{kezd}_i$ műveletet, utána egy választásra vár. Ha a közös memóriájú rendszer $\texttt{kezd}_i$ végrehajtása előtt választ vagy két választ ad, akkor U_i beállít egy *hibajelző bitet*, amely megengedi neki, hogy akármikor, akármennyi `kezd` kimenetet eredményezzen. (A $\texttt{henyél}_i$ művelet jelenléte lehetőséget ad arra is, hogy ne hozzon létre kimeneteket.) Természetesen az adott közös memóriájú rendszernek nem volna szabad ilyen hibát okoznia.

Az A közös memóriájú rendszernek az összes U_i-vel alkotott összekapcsolását ($1 \leq i \leq n$) a 9.2. ábra szemlélteti. Ez az összekapcsolás meglehetősen jól viselkedik: bármely pártatlan végrehajtásában minden i-re pontosan egy $\texttt{kezd}_i$ és pontosan egy $\texttt{választ}_i$ esemény van, továbbá a `választ` esemény rendelkezik az egyetértési és az érvényességi tulajdonsággal.



9.2.. ábra. A felhasználók és a közös memóriájú rendszer.

Formálisan, legyen Q az a történeti tulajdonság, ahol a *lenyomat*(Q) minden i-re és v-re $\texttt{kezd}(v)_i$ kimenetekből és $\texttt{választ}(v)_i$ bemenetekből áll, és ahol *történetek*(Q) az alábbi feltételeket teljesítő, *műveletalmaz*(Q)-beli műveletek β sorozatának halmaza.

1. Minden i-re β-ban pontosan egy $\texttt{kezd}_i$ eseményt pontosan egy $\texttt{választ}_i$ esemény követ.
2. (Egyetértés) Ha $\texttt{választ}(v)_i$ és $\texttt{választ}(w)_j$ is szerepel β-ban, akkor $v = w$.
3. (Érvényesség) Ha egy $\texttt{választ}(v)_i$ esemény szerepel β-ban, akkor néhány $\texttt{kezd}(v)_j$ esemény is szerepel β-ban (ugyanazon v-re).

Megmutatható, hogy *pártatlan_ történetek*$(A \times \prod_{1 \leq i \leq n} U_i) \subseteq$ *történetek*(Q). A bizonyítás a gyakorlatok között szerepel (lásd 9-2. gyakorlat).

9.3.. Megkülönböztethetetlen állapotok

Bevezetjük a megkülönböztethetetlenség fogalmát, amely hasznos lesz a 10. fejezet megoldhatatlansági bizonyításai során.

Tekintsünk egy A, n folyamatból álló közös memóriájú rendszert és az U_i felhasználók egy halmazát ($1 \leq i \leq n$). Legyen az $A \times \prod_{1 \leq i \leq n} U_i$ összetett rendszernek s és s' két állapota. Ha a P_i folyamat állapota, az U_i állapota és minden közös változó értéke megegyezik s-ben és s'-ben, akkor azt mondjuk, hogy s és s' *megkülönböztethetetlen* a P_i folyamat szempontjából, és ezt $s \overset{i}{\sim} s'$-vel jelöljük.

9.4.. Közös változók típusai

A közös memóriájú rendszerekre adott általános meghatározásban nem korlátoztuk a művelet típusát, amelyet hozzáféréskor a folyamat végezhet egy közös változón. Azaz amikor egy P_i folyamat hozzáfér egy x változóhoz, lehetősége van a P_i állapotát és x értékét tetszőlegesen változtatnia, függetlenül a P_i előző állapotától és x értékétől. A gyakorlatban azonban az közös változók általában csak a műveletek rögzített halmazát támogatják, például az olvasás és az írás műveleteket vagy az egyesített olvas/módosít/ír műveletet. Ebben az alfejezetben bevezetjük a *változótípus* fogalmát, és megmondjuk, hogy mit jelent ez típuskorlátozás észlelésénél egy közös memóriájú rendszer esetében.[1]

Egy változótípus a következőkből áll:

- értékek V halmaza;
- $v_0 \in V$ kezdőérték;
- *kérések* halmaza;
- *válaszok* halmaza;
- egy f : *kérések* $\times V \to$ *válaszok* $\times V$ függvény.

Az f függvény azt mondja meg, hogy mi történik, ha egy adott kérés érkezik a változóhoz, és a változó rendelkezik egy adott értékkel; f leírja az új értéket, amelyet a változó vesz fel, és a választ, amelyet visszaad. Jegyezzük meg, hogy a

[1] Az itt használt meghatározás megkívánja, hogy a változó determinisztikusan viselkedjen. A meghatározás ugyan általánosítható nemdeterminisztikus esetekre is, de az ezzel járó problémák miatt ezt inkább elhagyjuk, mivel nem szükséges a könyv eredményeihez.

változótípus *nem* egy b/k automata, még akkor sem, ha néhány összetevője hasonlít is egy b/k automata összetevőire. A változótípus esetében a legfontosabb, hogy a kérések és a válaszok főként úgy fordulnak elő, mint egy függvényalkalmazás, ugyanis a b/k automata modellben a kimenetek és a bemenetek önálló műveletek (és a további műveletek e kettő között fordulnak elő).

Tegyük fel, hogy rendelkezünk egy A közös memóriájú rendszerrel. Mit jelent az, hogy az A rendszerbeli x közös változó egy adott változótípusú? Ez egyrészt azt jelenti, hogy az $\mathit{értékek}_x$ halmaznak egyenlőnek kell lennie a típus értékeinek V halmazával, másrészt azt, hogy az x kezdőértékeinek $\mathit{kezdeti}_x$ halmaza csak egyetlen v_0 elemből áll. Ezenkívül az összes tranzakciónak, amely tartalmazza x-et, leírhatónak kell lennie a típus által lehető tett kérések és válaszok formájában. Ugyanis minden műveletnek, amely tartalmazza x-et, kapcsolatban kell állnia a változótípus néhány a kérésével. Ezenkívül minden P_i folyamat és minden a kérés esetében P_i-t és a-t tartalmazó tranzakciók halmazának leírhatónak kell lennie a következő formában, ahol p egy bizonyos predikátum az $\mathit{állapotok}_i$-n és g egy bizonyos reláció, $g \subseteq \mathit{állapotok}_i \times \mathit{válaszok} \times \mathit{állapotok}_i$. (A kódban az $\mathit{állapot}_i$ jelölést használjuk, hogy jelezzük a P_i folyamat állapotát.)

P_i-t és a-t tartalmazó tranzakciók:
  Előfeltétel:
    $p(\mathit{állapot}_i)$
  Hatás:
    $(b, x) := f(a, x)$
    $\mathit{állapot}_i :=$ bármely s, amelyre $(\mathit{állapot}_i, b, s) \in g$

Ez a kód azt jelenti, hogy a megállapítás, amely szerint az x változóhoz az a kérést használó P_i folyamaton keresztül lehet hozzáférni, összhangban van a p predikátummal (amely csak a P_i állapotát határozza meg). Ha ez a hozzáférés végbemegy, akkor az f függvény a változótípushoz fordul a kérésért és x változó értékéért, hogy meghatározza belőle a b választ és x egy új értékét. A b választ ezután a P_i folyamat használja állapotának frissítésére, amelyet valamilyen módon a g függvény engedélyez.

Ebben a könyvben az osztott algoritmusok leírásaiban különleges típusú közös változókhoz való hozzáférést vonnak maguk után, nem fogjuk pontosan a fenti módon p predikátumban és g relációkban megadni. Elméletileg azonban minden tranzakciót kifejezhetnénk ilyen formában.

9.4.1. példa. közös változók (regiszterek)

A leggyakrabban használt változótípus több processzor esetében a csak olvasó és író műveletet támogató változótípus. Egy ilyen típusú változót *olvasható/írható változónak*, *olvasható/írható regiszternek* vagy egyszerűen *regiszternek* nevezzük.

Egy olvasható/írható regiszter rendelkezik értékek egy tetszőleges V halmazával és $v_0 \in V$ tetszőleges kezdőértékkel. Kérései az `olvas` és az `ír`$(v), v \in V$. Válaszai a $v \in V$ és `nyugtáz`.[2] A regiszter f

[2] A kérések és a válaszok esetenként további információt is tartalmaznak, például a regiszter nevét. Ezt az egyszerűség érdekében legtöbbször elhagyjuk.

függvényét a következőképpen vezetjük be: $f(\texttt{olvas}, v) = (v, v)$ és $f(\texttt{ír}(v), w) = (\texttt{nyugtáz}, v)$.

Megjegyezzük, hogy a 9.1.1 példa rendszerében szereplő x változó nem írható le olvasható/írható regiszterként, mert az adott hozzáférések nem írhatók le a fent megadott formában. Az algoritmus azonban átírható úgy, hogy például egy x regiszter minden hozzáférését szétválasztjuk egy olvasó és egy író lépésre. Az eredményként kapott folyamat kódja alább látható. Az *állapot* lehetséges *hozzáférés* értékét két új lehetséges értékre cseréltük: *olvas* és *ír*.

Átmenetek:

`kezd`$(v)_i$
  Hatás:
    bemenet := v
    if *státus* = *tétlen* **then**
      státus := *olvas*

`olvas`$_i$
  Előfeltétel:
    státus = *olvas*
  Hatás:
    if x = *ismeretlen* **then**
      kimenet := *bemenet*
      státus := *ír*
    else
      kimenet := x
      státus := *választás*

`ír`$(v)_i$
  Előfeltétel:
    státus = *ír*
    v = *bemenet*
  Hatás:
    $x := v$
    státus := *választás*

`választ`$(v)_i$
  Előfeltétel:
    státus = *választás*
    kimenet = v
  Hatás:
    státus := *elvégzett*

A feladatpartíció újból összecsoportosítja a P_i folyamat minden helyileg vezérelt műveletét. Bár ez a kód nem pontosan egy p predikátum és egy g reláció szerint íródott, megjegyezzük, hogy könnyen átírható erre a formára. Például az `olvas`$_i$ esetében a p predikátum egyszerűen a „*státus* = *olvas*”, a g reláció pedig az $(s, b, s') \in$ *állapotok*$_i \times (V \cup \{$*ismeretlen*$\}) \times$ *állapotok*$_i$ hármasok halmaza, ahol s'-t az s-ből a következő kód segítségével kapjuk:

if b = *ismeretlen* **then**
  kimenet := *bemenet*
  státus := *ír*
else
  kimenet := b
  státus := *választás*

Az `ír`(v) művelet esetében a p predikátum egyszerűen a „*státus* = *ír* **and** v = *bemenet*”, a g reláció pedig az $(s, b, s') \in$ *állapotok*$_i \times (V \cup \{$*ismeretlen*$\}) \times$ *állapotok*$_i$ hármasok halmaza, ahol s'-t az s-ből a következő kód segítségével kapjuk:

státus := *választás*

Így x egy olvasható/írható közös változó.

Jegyezzük meg, hogy amikor egy algoritmust ilyen módon újraírunk, akkor a 9.1.1 példában említett egyetértési tulajdonság a továbbiakban nem feltétlenül teljesül.

9.4.2. példa. Olvasható-módosítható-írható közös változók

Egy további érdekes változótípus lehetővé teszi a hatékony olvasható/módosítható/írható művelet használatát. Egy x közös változón végrehajtott azonnali olvas/módosít/ír műveletben egy P_i folyamat az alábbiakat teheti meg:

1. beolvassa x értékét;
2. végrehajt – esetleg x értékét használva – néhány számítást, amely módosítja P_i állapotát és meghatározza x új értékét;
3. új értéket ad x-nek.

Egy általános olvas/módosít/ír műveletet nem egyszerű megvalósítani a multiprocesszorok által nyújtott primitív utasítások segítségével. A közös memóriájú modell nem csak a minden változóhoz való hozzáférés *oszthatatlanságát* követeli meg, hanem azt is, hogy minden folyamat *pártatlan* módon hajtsa végre az ilyen hozzáféréseket. E pártatlansági követelmény megvalósításához néhány alacsony szintű működési elv szükséges.

Amint már leírtuk, nem nyilvánvaló, hogy az olvasható/módosítható/írható változók modellezhetők változótípusként: az olvasható/módosítható/írható művelet két, változóhoz való hozzáférésként jelenik meg a szükséges egy helyett. Egyik lehetséges módja ennek, hogy egy folyamat, amely hozzá szeretne férni a változóhoz, állapotától függően meghatároz egy h függvényt, amelyet úgy használ, mint egy változóhoz tartozó kérést. A h függvény beszerzi a folyamat állapotát, amelyre azért van szüksége, hogy függvény formájában határozhassa meg az átmenetet ahhoz, hogy alkalmazni lehessen azt a változóra. A h függvény a v értékű változó értékét $h(v)$-re változtatja és visszaadja a folyamatnak v előző értékét. A folyamat ezután régi állapotától és v értékétől függően megváltoztathatja saját állapotát.

Formálisan egy olvasható/módosítható/írható változónak tetszőleges V értékhalmaza és tetszőleges $v_0 \in V$ kezdőértéke lehet. Kérései az összes olyan h függvény, amelyre: $h : V \to V$. Válaszai a $v \in V$ értékek. A változó f függvényét az $f(h, v) = (v, h(v))$ bevezeti be, azaz válaszként visszaadja előző értékét, és új értékét az átadott függvény alapján határozza meg.

Például a 9.1.1. példában bevezetett függvény, amelyet egy folyamat ad át a változónak, h_v alakú, ahol

$$h_v(x) = \begin{cases} v, & \text{ha } x = \mathit{ismeretlen}, \\ x, & \text{egyébként.} \end{cases}$$

A folyamat által átadott sajátos h_v függvény a folyamat *bemenetét* a v értékeként használja. Az *ismeretlen* egy visszatérési értéke *kimenetet* eredményez, amelyet a *bemenet* értékéül ad, egy $v \in V$ visszatérési

érték pedig *kimenetet* eredményez, amelyet értékül ad v-nek. Mindkét esetben a *státus* megfelelően módosult.

9.4.3. példa. További változótípusok

Számos változótípus használatos közös memóriájú többprocesszoros rendszerekben, többek között az olvasható/módosítható/írható korlátozott formái, valamint alapvető műveletek, mint például olvasás és írás. Az olvasható/módosítható/írható néhány népszerű korlátozott formája tartalmazza az `összehasonlít_cserél`, `cserél`, `vizsgál_beállít` és `be_hozzáad` műveleteket. Ezeket a műveleteket a következőképpen határozzák meg. Rögzítsünk egy V halmazt és v_0 kezdőértéket.

Az `összehasonlít_cserél` műveletek kérései `összehasonlít_cserél`(u, v) alakúak, ahol $u, v \in V$, és a válaszai elemei V-nek. Az `összehasonlít_cserél` kérések f függvényét

$$f(\texttt{összehasonlít\_cserél}(u,v), w) = \begin{cases} (w, v), & \text{ha } u = w \\ (w, w), & \text{egyébként} \end{cases}$$

szerint határozzuk meg, azaz ha a változó értéke egyenlő az első argumentummal, u-val, akkor a művelet az értéket visszaállítja a második paraméterre, v-re; egyébként a művelet nem változtatja meg a változó értékét. Mindkét esetben a változó eredeti értékét adja vissza.

A `cserél` műveletek kérései `cserél`(u) alakúak, ahol $u \in V$, és a válaszai elemei V-nek. A `cserél` kérések f függvényét

$$f(\texttt{cserél}(u), v) = (v, u)$$

szerint határozzuk meg, azaz a művelet az u bemeneti értéket a változóba írja és visszaadja v kezdőértéket.

A `vizsgál_beállít` műveletek kérései `vizsgál_beállít` alakúak, és a válaszai elemei V-nek. A `vizsgál_beállít` kérések f függvényét

$$f(\texttt{vizsgál\_beállít}, v) = (v, 1)$$

szerint határozzuk meg, azaz a művelet az 1 értéket írja a változóba és visszaadja a v kezdőértéket. (Feltesszük, hogy $1 \in V$).

Végül, a `be_hozzáad` műveletek kérései `be_hozzáad`(u) alakúak, ahol $u \in V$, és a válaszai elemei V-nek. A `be_hozzáad` kérések f függvényét

$$f(\texttt{be\_hozzáad}(u), v) = (v, v + u)$$

szerint határozzuk meg, azaz a művelet hozzáadja az u bemeneti értéket a v változó értékéhez, és visszaadja v eredeti értéket. (Ez a művelet megköveteli, hogy a V halmazon értelmezve legyen az összeadás művelet.)

Természetes módon bevezethetjük a változótípus *végrehajtásait* véges sorozatokként, $v_0, a_1, b_1, v_1, a_2, b_2, v_2, \ldots, v_r$ vagy végtelen sorozatokként, $v_0, a_1, b_1, v_1, a_2, b_2, v_2, \ldots$. Itt v-k V elemei, v_0 a változótípus kezdőértéke, a-k a kérések, b-k a válaszok, és a $v_k, a_{k+1}, b_{k+1}, v_{k+1}$ négyesek kielégítik a típus függvényét. (Azaz $(b_{k+1}, v_{k+1} = f(a_{k+1}, v_k)$.) Tehát a típus *története* a-k és b-k sorozatai, amelyek a típus végrehajtásaiból származnak.

9.4.4. példa. Egy olvasható/írható változótípus története

Az alábbi egy olvasható/írható változótípus története $V = \mathbb{N}$ és $v_0 = 0$ esetében:

$$\texttt{olvas}, 0, \texttt{ír}(8), \texttt{nyugtáz}, \texttt{olvas}, 8.$$

Ezt az alfejezetet változótípusok egyszerű összekapcsolásműveletének bevezetésével fejezzük be. Tekintsük a független változótípusok egy halmazát, mindegyiket a saját műveleteivel, mint egy egyszerű változótípust számos összetevővel és egyedi összetevőkön működő műveletekkel.

A változótípus $\{\mathcal{T}_i\}_{i\in I}$ megszámlálható halmazát *kompatibilisnek* nevezzük, ha a kérések minden halmaza diszjunkt és hasonlóképpen a válaszok minden halmaza is diszjunkt. Ekkor a változótípus egy megszámlálható kompatibilis halmazának $\mathcal{T} = \prod_{i\in I} \mathcal{T}_i$ *összekapcsolását* a következők határozzák meg:

- a V halmaz a $\mathcal{T}_i$ értékkészleteinek Descartes-szorzata;
- a v_0 kezdőérték a $\mathcal{T}_i$ kezdőértékeiből áll;
- a kérések halmaza a $\mathcal{T}_i$ kéréshalmazainak egyesítése;
- a válaszok halmaza a $\mathcal{T}_i$ válaszhalmazok egyesítése;
- az f függvény komponensenként hajtódik végre. Tekintsük az $f(a, w)$-t, ahol a a $\mathcal{T}_i$ egy kérése. Az f függvény alkalmazza a $\mathcal{T}_i$ függvényét a w i-dik komponensére, ahhoz, hogy megkapja (b, v)-t. Visszaadja b-t és w i-dik komponensét v-re állítja.

Ha I egy véges halmaz, néha használjuk a $\times$ belső műveletet az összekapcsolás jelölésére.

9.4.5. példa. Változótípusok összekapcsolása

Leírjuk két, $\mathcal{T}_x$ és $\mathcal{T}_y$ olvasható/írható változótípus összekapcsolását. (Úgy kell gondolni x-re és y-ra, mint két regiszter nevére.) Tegyük fel, hogy az értékkészletek V_x és V_y, illetőleg a kezdőértékek $v_{0,x}$ és $v_{0,y}$.

Csak ennek a két típusnak képezzük az összekapcsolását, ha azok kompatibilisek. Tehát az x és y alsó index-szel megkülönböztethetővé tettük a két típus kéréseit és válaszait. Ezenkívül a $\mathcal{T}_x \times \mathcal{T}_y$ képzett típusnak a $V_x \times V_y$ az értékhalmaz és a $(v_{0,x}, v_{0,y})$ pár a kezdőértéke. Kérései az $\texttt{olvas}_x$, $\texttt{olvas}_y$, $\texttt{ír}(v)_x$, ahol $v \in V_x$ és $\texttt{ír}(v)_y$, ahol $v \in V_y$. Válaszai v_x, ahol $v \in V_x$, v_y, ahol $v \in V_y$, valamint $\texttt{nyugtáz}_x$ és $\texttt{nyugtáz}_y$.

Most gondoljuk át az f függvényt. Legyen $w = (v, v')$ a $V_x \times V_y$ egy tetszőleges eleme. Ezután határozzuk meg w-t a következőkkel: $f(\texttt{olvas}_x, w) = (v_x, w)$, $f(\texttt{olvas}_y, w) = (v'_y, w)$, $f(\texttt{ír}(v'')_x, w) = (\texttt{nyugtáz}_x, (v'', v'))$ és $f(\texttt{ír}(v'')_y, w) = (\texttt{nyugtáz}_y, (v, v''))$. Így egy `olvas` visszaadja a vektor jelzett elemét, míg `ír` módosítja a jelzett komponenst.

9.5.. Bonyolultsági mértékek

Azért, hogy az aszinkron közös memóriájú rendszerekben megmérjük az időbonyolultságot, feltételezünk egy l felső korlátot egy folyamat egy lépésének idejére. Egy ilyen felső korlát lehetőséget ad arra, hogy meghatározzunk egy újabb felső korlátot arra az időre, amely szükséges egy jelentős esemény bekövetkezéséhez (például egy folyamatnak, amely fogadott egy $\texttt{kezd}_i$-t bemenetnek, hogy előállítson egy $\texttt{választ}_i$-t; kimenetnek).

Pontosabban bevezetünk a közös memóriájú rendszerek esetében egy *időbonyolultsági mértéket* mint annak az időbonyolultsági mértéknek egy speciális esetét, amelyet az általános b/k automatáknál vezettünk be a 8.6. alfejezetben. Azaz meghatározunk egy l felső korlátot minden folyamat minden Be feladatára; ez meghatározza az l egy felső korlátját arra az időre, amely a Be feladat két sikeres lehetősége között telik el egy lépés végrehajtásakor. Addig mérjük az *időt*, amíg valamely, az időpontok szuprémuma által kijelölt π esemény, amely a felső korlátokat figyelembe vevő időmegfeleltetések által meg lehet feleltetni π-nek. Hasonlóképpen mérjük az *időt* két jelentős esemény között a hozzájuk rendelhető idők közötti különbségek szuprémumával.

Jegyezzük meg, hogy az időmértékünk nem veszi figyelembe a folyamatok közötti versengésből származó többletköltséget, amely egy közös változóhoz való hozzáféréskor keletkezik. Többprocesszoros esetben, ahol a versengés probléma, az időmértéket ennek megfelelően kell módosítani.

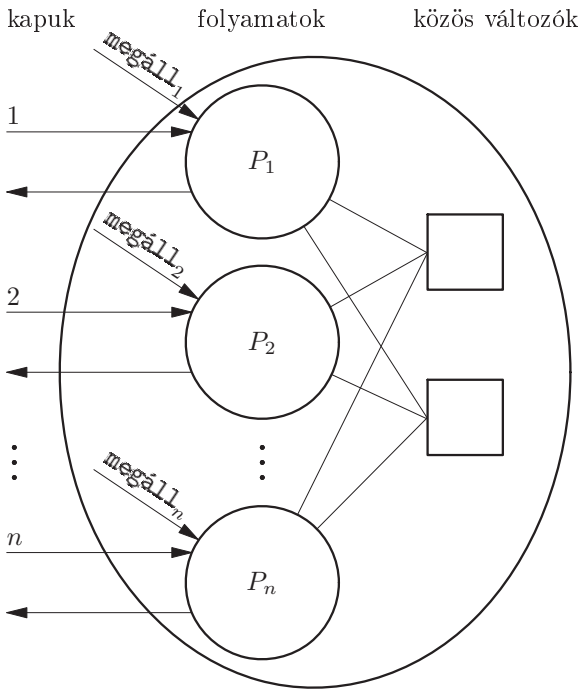
A közös memóriájú rendszerek bonyolultságának további érdekes mértékei valamilyen állandó mértékeket tartalmaznak, mint például az közös változók számát és az értékhalmazaik méretét.

9.6.. Hibák

A közös memóriájú rendszer P_i folyamatának *megállási hibája* egy $\texttt{megáll}_i$ bemeneti művelet használatával van modellezve, amely a P_i folyamat minden feladatának megállási hibáját okozza, de nincs hatással a többi folyamatra. Pontosabban, egy $\texttt{megáll}_i$ esemény csak a P_i folyamat állapotát képes megváltoztatni, tehát mi nem kényszerítjük ezeket az állapotváltozásokat, kivéve, hogy megköveteljük a P_i folyamat minden feladatának véglegesen letiltását. Nyitva hagyjuk annak a kérdését, hogy a P_i folyamat későbbi bemenetei figyelmen kívül maradnak, vagy a P_i folyamatnak ugyanarra az állapotára módosítja, mintha nem lett volna $\texttt{megáll}_i$, vagy valamely más állapotváltozás következik be. Ezek a megkülönböz-

tetések nem okoznak bajt, mert az ilyen állapotváltozások hatásai nem léphetnek kommunikációba más folyamatokkal.

A 9.3. ábra egy aszinkron közös memóriájú rendszer felépítését ábrázolja megállási hibákkal.



9.3.. ábra. Aszinkron közös memóriájú rendszer felépítése megállási hibákkal.

9.7.. Véletlenítés

Egy valószínűségi közös memóriájú rendszer fogalmát a 8.8. alfejezetben egy valószínűségi b/k automata általános meghatározásának speciális eseteként vezettük be, abban a speciális esetben, amikor a b/k automata egy közös memóriájú rendszer.

9.8.. Megjegyzések a fejezethez

Az ebben a fejezetben tárgyalt alapmodellhez nincs jellegzetes referencia. Ez egy „díszkert-típusú” közös memóriájú modell, amelyet korábban a b/k automata keretén belül fogalmaztunk meg. A közös memóriájú rendszerekre másik modellt állított fel Lynch és Fischer [216], amelyben a folyamatok nem külső események

által, hanem az közös változókhoz való azonnali hozzáférés segítségével kommunikálnak. Kruskal, Rudolph és Snir [171] a változók többféle típusát vezették be a közös memóriájú többprocesszoros rendszerekben való használatra.

Dwork, Herlihy és Waarts [103] olyan időbonyolultságot ajánlottak, amely figyelembe veszi a versengést a közös memóriájú elérésért. A valószínűségi közös memóriájú rendszer formális modellezése Lynch, Saias és Segala [208] munkájából származik.

9.9.. Gyakorlatok

9-1. Legyen A a 9.1.1. példában leírt közös memóriájú rendszer.

(a) Bizonyítsuk be, hogy *pártatlan_ történetek*$(A) \subseteq$ *történetek*(P), ahol P a 9.1.1. példában leírt történeti tulajdonság.

(b) Vezessünk be egy érdekes Q történeti biztonságossági tulajdonságot, és mutassuk meg, hogy azt az A (nem feltétlenül pártatlan) történetei kielégítik. Azaz lássuk be, hogy *történetek*$(A) \subseteq$ *történetek*(Q). A Q tulajdonságnak említést kell tennie arról is, hogy mi történik akkor, ha ugyanannak a P_i folyamatnak egynél több $\texttt{kezd}_i$ művelete van.

9-2. Bizonyítsuk be, hogy *pártatlan_ történetek*$(A \times \prod_{1 \le i \le n} U_i) \subseteq$ *történetek*(Q), ahol A a 9.1.1. példában bevezetett közös memóriájú rendszer és Q a 9.2.1. példában bevezetett történeti tulajdonság.

Egyik lehetséges módja a bizonyításnak, hogy átfogalmazzuk Q-t, mintha egy S biztonságossági tulajdonság és egy L élénkségi tulajdonság metszete lenne. S tartalmazhatja az egyetértési és az érvényességi tulajdonságot, valamint az első feltétel azon részét, hogy minden i-re a P_i műveleteinek részsorozata valamely prefixe egy $\texttt{kezd}_i, \texttt{választ}_i$ alakú sorozatnak. L csak azt tudja megmondani, hogy legalább egy $\texttt{kezd}_i$ esemény és legalább egy $\texttt{választ}_i$ esemény fordul elő minden i-re. Mutassuk meg, hogy minden rendszerösszetevő megőrzi S-t és a *történetek*$(A \times \prod_{1 \le i \le n} U_i) \subseteq$ *történetek*(S) belátásához használjuk fel a 8.11. tételt. (Hogy A megtartja S-t, bizonyítható a *történetek*$(A) \subseteq$ *történetek*(P) tényből.) Ezután használjuk a pártatlanság feltételezését, hogy belássuk az élénkséget.

9-3. Bizonyítsuk be, hogy a 9.2.1. példabeli $A \times \prod_{1 \le i \le n} U_i$ rendszer invariánsa a következő: ha $\textit{választás}_{U_i} \neq \textit{ismeretlen}$ és $\textit{választás}_{U_j} \neq \textit{ismeretlen}$, akkor $\textit{választás}_{U_i} \neq \textit{választás}_{U_j}$[3]. Ezt két módon tehetjük meg:

(a) a *történetek*$(A \times \prod_{1 \le i \le n} U_i) \subseteq$ *történetek*(S) tényre alapozva, amelyet a 9.2. gyakorlatban bizonyítottunk.

(b) az invariáns belátásának megszokott módszerét használva: egy adott rendszerállapotig a végrehajtási út hossza szerinti indukcióval.

9-4. A 9.4.1. példában leírt olvasható/írható regiszter alapú rendszer teljesíti-e ugyanazt a P történeti tulajdonságot, mint a 9.1.1. példában szereplő rendszer. Ha igen, bizonyítsuk be. Ha nem, adjunk ellenpéldát és mondjuk ki, hogy ez a legerősebb állítás a rendszer viselkedésére, majd bizonyítsuk be.

[3] Alsó indexet használunk, hogy megkülönböztessük az egyes automatákhoz tartozó változókat.

9-5. *Kutatási kérdés.* Vezessük be a közös memóriájú rendszerek egy másik modelljét úgy, hogy, b/k automatákat csak folyamatok modellezésére használjuk, és az közös változók részére új típusú állapotgépet vezetve be (a változótípus modelljéhez hasonlóan). Vezessünk be egy alkalmas összekapcsolás műveletet ahhoz, hogy összekapcsoljon „kompatibilis" folyamatot és közös változó automatákat egyetlen b/k automatába, hogy az modellezhesse az egész rendszert. Milyen módosítások szükségesek a következő fejezetek eredményeihez, hogy az új meghatározásainknak megfeleljenek?

10. fejezet

Kölcsönös kizárás

Ebben a fejezetben elkezdjük az *aszinkron algoritmusok* tanulmányozását. Az aszinkron algoritmusok általában nagyon különböznek a szinkron algoritmusoktól, mivel meg kell birkózniuk az aszinkronitás és az osztottság által okozott bizonytalansággal. Az aszinkron hálózatokban például az egyes folyamatok lépései és az üzenetküldések tetszőleges sorrendben is történhetnek, nem feltétlenül a záróló lépéses egyidejűség alapján.

Mielőtt az aszinkron hálózatok algoritmusait vizsgálnánk, olyan algoritmusokat tanulmányozunk, amelyek aszinkron elérésű memóriát használnak. Ezt elsősorban azért tesszük, mert ez az eset valamivel egyszerűbb. De, amint ez a 17. fejezetben látható lesz, nagyon szoros kapcsolatot találunk az aszinkron közös memória modellje és az aszinkron hálózat modellje között. Például az aszinkron közös memóriára írt algoritmusokat olyanokká alakíthatjuk át, amelyek aszinkron hálózatokban is futhatnak. Ebben és a 11. fejezetben nem fogjuk túlságosan figyelembe venni a hibákat; az aszinkronitás önmagában is elég sok érdekes kérdést vet fel.

Most a *kölcsönös kizárás* feladatát vizsgáljuk, amelynek lényege annak megoldása, hogy egy meg nem osztható erőforráshoz (pl. egy nyomtatóhoz), egyszerre legfeljebb egy felhasználó férhessen hozzá. Egy másik lehetőség, hogy ezt olyan feladatnak tekintjük, hogyan biztosítsuk bizonyos programrészek végrehajtását a *belépési szakaszban*, ahol egyidőben két program nem tartózkodhat. Nem tudjuk azt, hogy mely felhasználók és mikor igénylik az erőforrást. Ez a feladat az osztott és központosított rendszerekben is fellép.

Dijkstra algoritmusával kezdve bemutatunk néhány kölcsönös kizárási algoritmust az olvasó/író közös memória modellre. A későbbi algoritmusok javítanak Dijkstra algoritmusán, biztosítva a pártatlanságot a különböző felhasználók számára és gyengítve a használt közös memória típusát. Ezután megadunk egy alapvető alsó határt az olvasó/író közös változók számát illetően, amelyek a feladat megoldásához szükségesek. Végül, alsó és felső határra vonatkozó eredményeket adunk meg arra az esetre, amikor a közös memória erősebb, olvasható-módosítható-írható közös változókból áll.

Ez a fejezet meglehetősen hosszú. Ennek az a fő oka, hogy nemcsak arra használjuk, hogy algoritmusokat és megoldhatatlansági eredményeket mutassunk be,

hanem arra is, hogy alapvető ötleteket vezessünk be, amelyeket a könyv későbbi részeiben használunk. Ezek között találjuk a közös memóriát használó rendszerek és környezeteik modellezésének módjait, az aszinkron algoritmusok helyességi feltételeiről szóló állításokat (melyek tartalmazzák a biztonságossági, haladási és pártatlansági feltételeket), az aszinkron algoritmusok bizonyítási módjait (működésen, invariáns feltételeken és kapcsolatok szimulálásán alapuló módszereket), az aszinkron algoritmusok időbonyolultságának meghatározási és elemzési módjait, valamint az alsó határok bizonyításának módszereit.

10.1.. Az aszinkron közös memória modellje

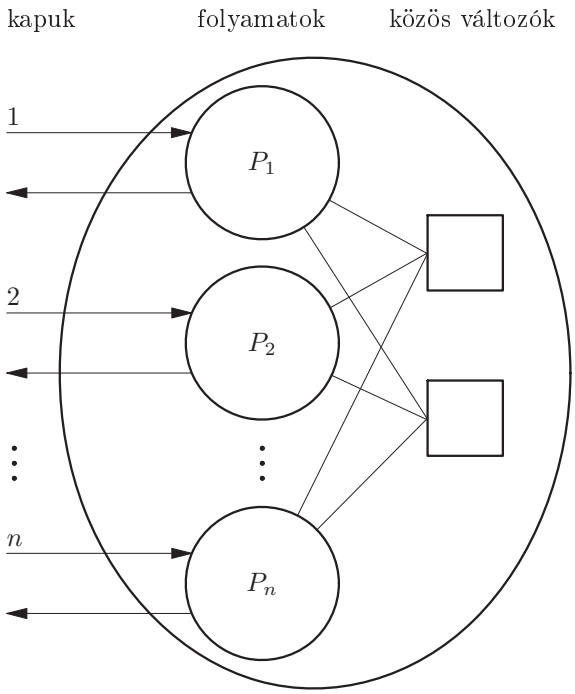
Mielőtt bármilyen algoritmus leírását elkezdenénk, bemutatjuk azt a számítási modellt, amelyet ebben és a következő három fejezetben használunk. Most röviden és vázlatosan írjuk le; teljesebb és egyben formális leírás a 9. fejezetben található.

A rendszert folyamatok és közös változók halmazával modellezzük, amelyek közötti kapcsolatokat a 10.1. ábra mutatja be. Minden P_i folyamat egy állapotgép, amelyet egy $állapotok_i$ állapothalmazzal és az $állapotok_i$ halmaznak a $kezdő\_áll_i$, a kezdőállapotokat leíró részhalmazával jellemzünk, pontosan úgy, mint a szinkron modellekben. Itt azonban a P_i folyamat címkézett *műveletekkel* is rendelkezik, amelyek azokat a tevékenységeket írják le, amelyekben a folyamat részt vesz. Ezek *kimeneti*, *bemeneti* vagy *belső* műveletek. A 10.1. ábrán a folyamatok köreibe bemenő és köreiből kimenő nyilak az adott folyamat különböző bemeneti és kimeneti műveleteit jelképezik. Továbbá megkülönböztetjük a belső műveletek két típusát: azokat, amelyek a közös memóriát használják, és azokat, amelyek szigorúan helyi számításokat végeznek. Ha egy műveletnek szüksége van a közös memóriára, akkor feltételezzük, hogy csak egy közös változót igényel.

A szinkron beállításoktól eltérően ebben a modellben nincs üzeneteket előállító függvény, mivel nincsenek üzenetek. A folyamatok között minden kapcsolattartás a közös változókon keresztül történik.

A rendszerben adott egy *átmenetek* reláció, amely a (s, π, s') hármasok halmazából áll, ahol s és s' *automata állapotok*, azaz a közös folyamatok állapotainak és az közös változók értékeinek együtteseiből állnak. π egy bemeneti, egy kimeneti vagy egy belső művelet címkéje. A folyamatállapotokat és a közös változók értékeit együtt azért nevezzük „automataállapotoknak", mert a 9. fejezet formális modelljében az egész rendszert egyetlen automata modellezi. Az $(s, \pi, s') \in$ *átmenetek* állítás azt jelenti, hogy az s automata állapotból lehetséges az átmenet az s' állapotba, mintegy a π tevékenység végrehajtásának eredményeként. Figyeljük meg, hogy az *átmenetek* inkább *reláció*, mint függvény – a kényelem kedvéért megengedjük, hogy a modellünk sztochasztikusságot is tartalmazzon.

Feltételezzük, hogy a bemeneti műveletek bármikor előfordulhatnak, azaz, hogy a rendszer *bemenet-engedélyezett*. Formálisan ez azt jelenti, hogy minden s automata állapotra és π bemeneti műveletre létezik egy s', úgy, hogy $(s, \pi, s') \in$ *átmenetek*. Ellenben a kimeneti és a belső lépések csak az állapotok részhalmazaként megengedettek. A bemenet-engedélyezés tulajdonság mögötti gondolat az,



10.1.. ábra. Egy aszinkron közös memóriájú rendszer.

hogy a bemeneti műveleteket egy tetszőleges külső felhasználó ellenőrzi, míg a belső és a kimeneti műveleteket maga a rendszer.

Az átmenetek halmazának van néhány „helyi" megszorítása is. Az első az, hogy minden olyan átmenet, amely nem használja a közös memóriát, csak annak a folyamatnak az állapotát érintheti, amely a műveletet végrehajtja. Másrészt pedig, egy olyan átmenetre, amely a P_i folyamatot és az x közös változót foglalja magában, csak a P_i folyamat állapotát és az x változó értékét foglalhatja magában. Feltételezzük, hogy a közös memória általi *megengedés* csak a folyamat állapotától függ, és nem függ a használt közös változó értékétől. A folyamat állapotán és a változó értékén eredményezett változások függhetnek azonban a változó értékétől is.

A közös változó átmeneteire gyakran további megszorítások vonatkoznak, ha azok adott típusú műveletek, például `olvas` vagy `ír`. Az x változó olvasása a folyamat állapotának változását eredményezi, mely az előző állapotától és az x értékétől függ; ugyanakkor az x változó értéke nem változik meg. Az `ír` lépés maga után vonja egy megadott érték beírását az közös változóba, felülírva azt, ami előtte volt; ez is megváltoztathatja a folyamat állapotát. Mi leginkább azt a modellt követjük, amelyben a változók az `olvas` és az `ír` műveletek használatával érhetők el, ugyanakkor figyelembe veszünk néhány erősebb műveletet is, például az olvasható-módosítható-írható műveletet.

Egy aszinkron közös memóriájú rendszer végrehajtása nagyon különbözik a szinkron rendszerekétől. Jelen esetben a folyamatokról azt feltételezzük, hogy egyidejűleg csak egy-egy lépést tesznek, de inkább *tetszőleges* sorrendben, mint sem szinkronizált menetekben. Ez a tetszőleges sorrend az aszinkron modell lényege. Egy végrehajtási sorozat egy $s_0, \pi_1, s_1, \ldots$ váltakozó sorozat, amely automata-állapotokból és műveletekből áll (mindegyik művelet egy adott folyamathoz tartozik), ahol az egymást követő (állapot, művelet, állapot) hármasok teljesítik az átmeneti relációt. Egy végrehajtási sorozat lehet véges vagy végtelen sorozat.

Létezik egy nagyon fontos kivétel a folyamatok lépéseinek tetszőleges sorrendben történő végrehajtása alól. Nem akarjuk megengedni, hogy egy folyamat ne lépjen tovább akkor, amikor feltételezzük róla, hogy tovább kell lépnie, azaz, amikor a folyamat olyan állapotban van, ahol néhány *helyileg ellenőrzött* művelet (azaz, egy nem bemeneti művelet) megengedett. (Bár bemeneti műveleteket mindig megengedünk, nem feltételezzük, hogy ezek valaha is előfordulnak.) Ezt a feltételt nehéz pontosan megfogalmazni.

Például azt mondhatjuk, hogy: „ha egy folyamat csak véges sok lépést tesz, akkor ennek végső állapota olyan, amelyben semmilyen helyileg ellenőrzött művelet nem megengedett." De ez nem elég – mi ki szeretnénk zárni néhány olyan esetet is, amelyben egy folyamat végtelen sok lépést tesz, de egy bizonyos ponton túl minden lépés bemeneti. Meg kell bizonyosodnunk arról, hogy a folyamat lehetőséget kap helyileg ellenőrzött műveletet végrehajtására is.

Megpróbálhatjuk a szükséges feltételt a következőképpen megadni: „ha egy folyamat csak véges sok lépést tesz, akkor ennek a folyamatnak a végső lépése olyan, amelyben nincs megengedett helyileg ellenőrzött művelet, és ha egy folyamat végtelen sok lépést tesz, akkor ezek között végtelen sok lépés helyileg ellenőrzött lépés." Ez nem teljesen igaz – vegyük például azt az esetet, amikor egy folyamat végtelen sok bemenetet kap és nem hajt végre helyileg ellenőrzött műveletet, mert valójában egyetlen helyileg ellenőrzött művelet sem megengedett. Ez a helyzet jónak tűnik, mivel azt mondhatjuk, hogy a folyamat „lehetőséget" kapott a helyileg ellenőrzött lépésekre, de egyszerűen egy sem volt olyan, amelyet végre „akart" hajtani.

Ezeket a lehetőségeket mind figyelembe vesszük a következő meghatározásban. Minden P_i folyamatról feltételezzük, hogy az alábbiak egyike igaz:

1. a teljes végrehajtási sorozat véges és a végső állapotban a P_i folyamat egyetlen helyileg ellenőrzött műveletete sem megengedett;

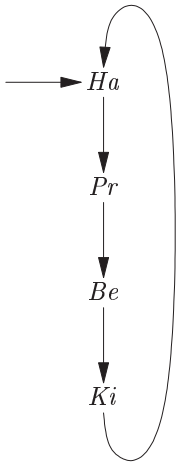
2. a végrehajtási sorozat végtelen és a P_i folyamatban vagy végtelenül sokszor fordulnak elő a helyileg ellenőrzött műveletek, vagy pedig végtelenül sok helyen az ilyen műveletek nem voltak megengedettek.

Ezt a feltételt a közös memóriájú rendszer *pártatlansági feltételének* nevezzük. (A b/k automatára a 8. fejezetben adott meghatározás fogalmait felhasználva azt mondhatjuk, hogy így egy folyamat minden helyileg ellenőrzött művelete egyetlen feladatba tömöríthető.)

10.2.. A feladat

A kölcsönös kizárás feladata magában foglalja egyetlen feloszthatatlan és megoszthatatlan erőforrás hozzárendelését az U_1, U_2, ..., U_n felhasználókhoz. A felhasználókat tekinthetjük alkalmazásoknak is. Az erőforrás lehet például egy nyomtató vagy bármilyen más kimeneti eszköz, amely kizárólagos hozzáférést igényel ahhoz, hogy a kimenet értelmes legyen. Az erőforrás lehet egy adatbázis is, vagy más adatszerkezet, amely kizárólagos hozzáférést kér ahhoz, hogy elkerülje a különböző felhasználók műveleteinek egymásra hatását.

Azt a felhasználót, aki hozzáfér az erőforráshoz, *belépési szakaszban* lévő felhasználóként modellezzük, a belépési szakaszt egyszerűen azonosíthatjuk állapotainak egy kiválasztott részhalmazával. Amíg egy felhasználó semmilyen módon nem igényli az erőforrást, akkor azt mondjuk róla, hogy a *haladási szakaszban* található. Ahhoz, hogy egy felhasználó beléphessen a saját belépési szakaszába, végrehajt egy *próba protokollt* és miután végzett az erőforrás használatával, végrehajt egy (gyakran triviális) *kilépés protokollt*. Ez az eljárás ismételhető, így minden felhasználó egy ciklust követ, a *haladás szakaszától* (Ha) a *próba szakaszáig* (Pr) haladva, azután pedig a *belépési szakaszától* (Be) a *kilépés szakaszáig* (Ki), majd pedig újból vissza a haladás szakaszához. Ezt a ciklust a 10.2. ábra mutatja.

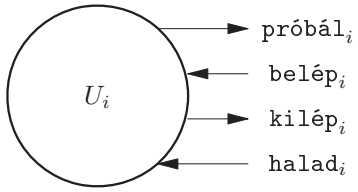


10.2.. ábra. Egy felhasználó szakaszainak ciklusa.

A kölcsönös kizárás algoritmusát a fent leírt közös memória modelljében vizsgáljuk – a felépítést lásd a 10.1. ábrán. A közös memória rendszer n folyamatot tartalmaz ($P_1, \ldots, P_n$), és mindegyik P_i megfelel az U_i felhasználónak. A P_i folyamat bemenete a `próbál`$_i$ művelet, amely az U_i felhasználó erőforrás iránti igényét modellezi és a `kilép`$_i$ művelet, amely az U_i felhasználónak azt a bejelentését ábrázolja, amely szerint már nem használja az erőforrást. A P_i folyamat kimenete a `belép`$_i$ művelet, mely biztosítja azt, hogy az U_i felhasználó megkapja az erőfor-

rást, valamint a $\texttt{halad}_i$ művelet, amely jelzi U_i-nek, hogy folytathatja hátralévő munkáját. A `próbál, belép, kilép` és a `halad` műveletek a közös memóriájú rendszer egyedüli külső műveletei. A folyamatok a próba és a kilépés protokollok végrehajtásáért felelősek. Minden P_i folyamat „ügynök" szerepet játszik az U_i felhasználó érdekében.

Minden U_i felhasználót, $1 \leq i \leq n$, egy állapotgép modellez (formálisan egy *b/k automata*), amely a saját ügynökfolyamatával kommunikál, felhasználva a $\texttt{próbál}_i$, $\texttt{belép}_i$, $\texttt{kilép}_i$ és a $\texttt{halad}_i$ műveleteket. Az U_i felhasználó külső felülete (formálisan a *külső lenyomata*) a 10.3. ábrán látható.



10.3.. ábra. Az U_i felhasználó külső felülete.

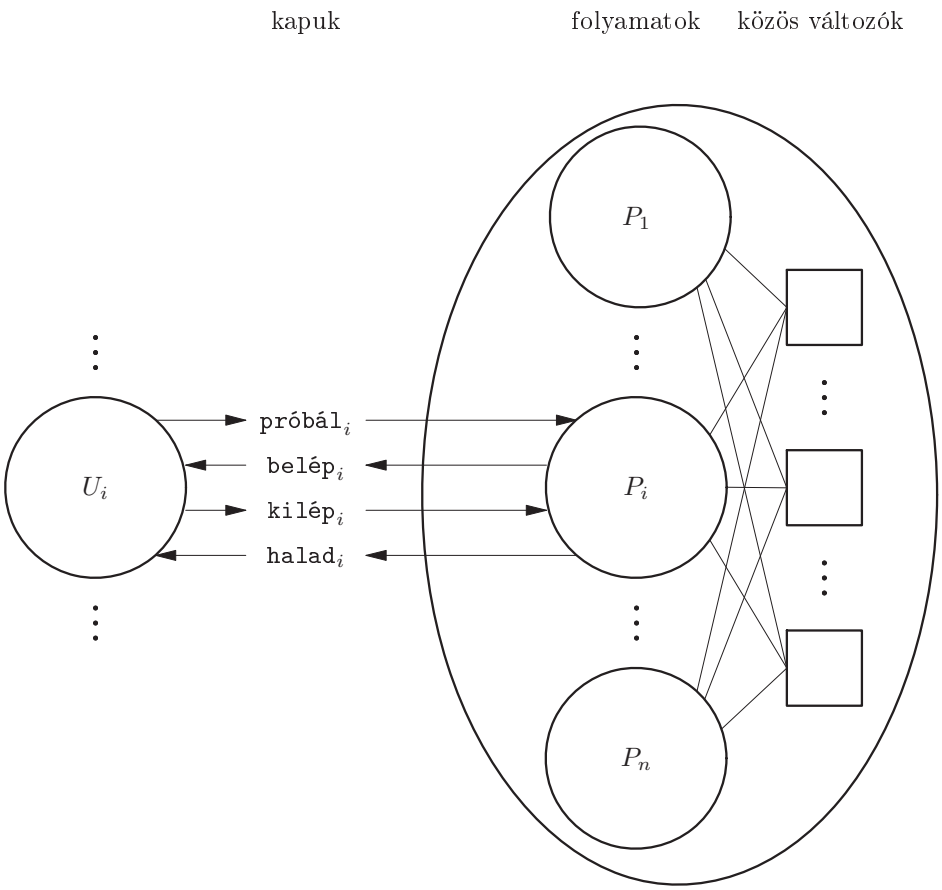
Minden U_i felhasználó valamilyen alkalmazást futtathat. Az egyetlen dolog, amelyet az U_i felhasználóról feltételezünk, az, hogy aláveti magát a szakaszok ciklikusan ismétlődő protokolljának, azaz, hogy az U_i felhasználó nem sérti meg elsőként a $\texttt{próbál}_i$, $\texttt{belép}_i$, $\texttt{kilép}_i$, ... ($\texttt{próbál}_i$-vel kezdődő) műveletek ciklikus sorrendjét önmaga és ügynökének folyamata között. Formálisan, egy $\texttt{próbál}_i$, $\texttt{belép}_i$, $\texttt{kilép}_i$ és a $\texttt{halad}_i$ műveletekből álló sorozatot a P_i felhasználó *jólformált* sorozatának nevezzük, ha ez előtagja a ciklikusan rendezett $\texttt{próbál}_i$, $\texttt{belép}_i$, $\texttt{kilép}_i$, $\texttt{halad}_i$, $\texttt{próbál}_i$, ... sorozatnak. Ezután előírjuk, hogy U_i *megőrizze* a *történet tulajdonságot*, amelyet a P_i felhasználó jólformált sorozatainak halmaza alkot. (A *történet tulajdonság* és *megtartás* fogalmaknak a 8.5.4. szakaszban található meghatározásait használjuk.)

Az U_i olyan végrehajtási sorozataiban, amelyben megmarad a műveletek ciklikus sorrendje, azt mondjuk, hogy az U_i felhasználó

- a *haladás szakaszban* van kezdetben és bármely $\texttt{halad}_i$ és az azt követő $\texttt{próbál}_i$ esemény között;
- a *próba szakaszban* van bármely $\texttt{próbál}_i$ esemény és az azt követő $\texttt{belép}_i$ esemény között;
- a *belépési szakaszban* van bármely $\texttt{belép}_i$ esemény és az azt követő $\texttt{kilép}_i$ esemény között. Ezalatt U_i-ről feltételezzük, hogy szabadon rendelkezhet az erőforrással (bár az erőforrást kifejezetten nem modellezzük);
- a *kilépés szakaszban* van bármely $\texttt{kilép}_i$ esemény és a következő $\texttt{marad}_i$ esemény között.

A 10.4. ábra a rendszerben található minden kölcsönhatást bemutat.

Most megfogalmazhatjuk, mit jelent az A közös memóriájú rendszer számára, hogy *megoldja a kölcsönös kizárás feladatát* egy adott felhasználói csoport szá-



10.4.. ábra. A kölcsönös kizárás feladatának összetevői közötti kölcsönhatások.

mára. Nevezetesen, A és a felhasználók együttesének (formálisan az összekapcsolásának) a következő feltételeket kell teljesítenie.

Jólformáltság. Bármely végrehajtási sorozatnál és bármely i-re az a részsorozat, mely az A és az U_i közötti kapcsolatokat tartalmazza, P_i-nek egy jólformált sorozata.

Kölcsönös kizárás. Nincs olyan elérhető rendszerállapot (mely A-nak egy automata-állapota és minden U_i állapotainak együttese), melyben egynél több felhasználó van a *Be* belépési szakaszban.

Haladás. *Pártatlan végrehajtási sorozat* bármely pontjában:

1. (haladás a próba szakaszban) ha legalább egy felhasználó Pr-ben található és egyetlen egy sem található Be-ben, akkor később valamelyik felhasználó belép a Be-be;
2. (haladás a kilépés szakaszban) ha legalább egy felhasználó Ki-ben található, akkor később valamelyik felhasználó belép Ha-ba.

Azt mondjuk, hogy az A közös memóriájú rendszer *megoldja a kölcsönös kizárás feladatát*, ha megoldja a problémát bármely felhasználócsoport számára.

Megjegyezzük, hogy a helyesség feltételét a felhasználók szakaszainak fogalmaival határozzuk meg. Rendes körülmények között a folyamatok állapotait ugyancsak a saját szakaszaik szerint fogjuk osztályozni, és ezek a szakaszok teljesen megfelelnek a felhasználók szakaszainak. Így a helyességi feltételeket ekvivalens módon a folyamatok szakaszainak fogalmaival is megadhatjuk. Ebben a fejezetben a továbbiakban egymással felcserélhető módon fogunk beszélni a felhasználók és a folyamatok szakaszairól.

Figyeljük meg, hogy a haladás feltételezi, hogy a rendszer végrehajtási sorozata pártatlan, azaz feltételezi, hogy minden folyamat (és minden felhasználó) folyamatosan továbblép. Ha ezt nem feltételeztük volna, akkor nem lenne elfogadható az a kérés, hogy a `kilép` vagy a `halad` kimeneteket véletlenszerűen végezze. Másrészt pedig, nem szükséges pártatlanságot feltételezni ahhoz, hogy a rendszer biztosítsa a jólformáltságot vagy a kölcsönös kizárást. A különbség abban áll, hogy a jólformáltság és a kölcsönös kizárás feltételei *biztonságossági tulajdonságok* (olyan tulajdonságok, amelyek azt mondják, hogy bizonyos „rossz" dolgok sohasem történnek meg), míg a haladás feltétele *elevenségi tulajdonság* (mely azt mondja, hogy néhány „jó" dolog biztosan megtörténik).

Történet tulajdonságok.. Ezeket a helyességi feltételeket ekvivalens módon bemutathatjuk a *történet tulajdonság* fogalmával is, amelyet a 8.5.2. szakaszban határoztunk meg. Például meghatározhatunk egy T történet tulajdonságot, ahol *lenyomat*(T) tartalmazza a `próbál`, `belép`, `kilép` és `halad` kimeneti műveleteket, valamint egy *történetek*(T) halmazt, amely ezen műveletek azon β sorozataiból áll, amelyek a következő három feltételt teljesítik:

1. β jólformált minden i-re;
2. β nem tartalmaz két `belép` eseményt egy közbeeső `kilép` esemény nélkül;
3. β bármely pontjában:
   (a) ha valamely folyamat utolsó eseménye `próbál` és egyetlen folyamatnak sem utolsó eseménye `belép`, akkor létezik egy későbbi `belép` esemény.
   (b) ha valamely folyamat utolsó eseménye a `kilép`, akkor létezik egy későbbi `halad` esemény.

A fentiek segítségével a kölcsönös kizárás feladatának egyenértékű meghatározása a következő: minden B-re, amely A együttese a felhasználókkal, *pártatlan_történetek*$(B) \subseteq$ *történetek*(T). (Emlékeztetünk arra, hogy B külső műveletei éppen a `próbál`, `belép`, `kilép` és `halad` műveletek.) A T történet tulajdonságot is felosztathatjuk két részre: a tulajdonságra, mely a jólformáltságot és a kölcsönös kizárást tartalmazza, valamint a biztonságosság élénkségi tulajdonságra, amely a haladási tulajdonság.

Közös felelősség a haladás biztosításáért.. A megadott helyességi feltételek mellett a teljes rendszer folyamatos haladásáért a felelősség nemcsak a protokollra hárul, hanem a felhasználókra is. Ha egy U_i felhasználó megkapja az erőforrást (egy $\texttt{belép}_i$ esemény által), de azt soha nem adja vissza (egy $\texttt{kilép}_i$ esemény által), az egész rendszer leáll. Ha azonban a felhasználó biztos visszaadja az erőforrást minden esetben, amikor ezt megkapja, akkor a haladás feltétele maga után vonja, hogy az egész rendszer folytatja a haladást, a folyamatokat ismételten újabb szakaszokba juttatva (kivéve, ha bizonyos ponton túl minden felhasználó a saját haladás szakaszában marad).

Kizárás.. A haladás feltétele, amelyet fent állítottunk, nem vonja maga után, hogy bármely, erőforrást igénylő felhasználó valaha is sikeresen eléri a belépési szakaszát. Ez inkább egy „globális" haladási fogalom, amely csak azt állítja, hogy *valamely* felhasználó eléri a belépési szakaszát. A következő forgatókönyv például nem szegi meg a haladás feltételét: bármilyen kezdőállapotból kiindulva, az U_1 felhasználó belép a Pr-be. Ezután az U_2 felhasználó ciklikusan belép a saját négy szakaszába végtelen sokszor, míg az U_1 a Pr-ben marad, a többi folyamat pedig a Ha-ban. A mi haladási feltételünk nem biztosítja azt, hogy az U_1 valaha is eléri Be-t.

A folyamatok tevékenységének korlátozása.. Van egy másik korlátozás – egy technikai korlátozás – amelyet ebben a fejezetben feltételezünk: egy folyamatnak a közös memóriájú rendszerben csak akkor lehet egy helyileg ellenőrzött művelete megengedett, amikor a felhasználója a saját próba vagy kilépés szakaszában található. Ez azt állítja, hogy egy folyamat csak addig foglalt a protokoll végrehajtásával, amíg aktív kérések érkeznek hozzá. Ez a feltételezés megegyezik azzal a szemponttal, amely szerint minden folyamat egyszerűen ügynöke a neki megfelelő felhasználónak.

A gyakorlatban ez a feltételezés vagy értelmes, vagy nem. A kölcsönös kizárás feladatát először egy időosztásos, egyprocesszoros keretben tanulmányozták, ahol egy közös processzoron a felhasználók független logikai folyamatok voltak. Ebben a keretben, ha engedélyt adtunk volna egy állandó folyamatnak arra, hogy az erőforráshoz való hozzáférést kezelje, ez külön környezet-kapcsolást okozott volna a kezelő folyamat és a felhasználó folyamatok között. Egy valós többprocesszoros környezetben elkerülhetjük a környezetváltást, ha egy processzort kizárólag az erőforrás kezelésére használunk. Általában azonban sok erőforrást kell kezelni és minden olyan processzor, amely kizárólag egy erőforrás kezelésére szolgál, nem áll rendelkezésre más számítási feladatokban.

Olvasható/írható közös változók.. A fejezet nagyobb részében (kivéve a 10.9. alfejezetet) feltételezzük, hogy az közös változók olvasó/író változók, amelyeket *regiszter* néven is ismerünk. Egy adott lépésben egy folyamat vagy olvas, vagy pedig ír egy közös változót, de nem végezheti el mindkét műveletet. Így a két művelet, amely magában foglalja a P_i folyamatot és az x regisztert, a következő.

1. (`olvas`) A P_i folyamat olvassa az x regisztert és felhasználja az olvasott értéket a P_i folyamat állapotának megváltozatására.
2. (`ír`) A P_i folyamat értéket ír az x regiszterbe, amelyet az P_i folyamat állapota határoz meg.

Ezt a részt egy egyszerű lemmával zárjuk, amely azt állítja, hogy a folyamatok nem állhatnak meg, míg a saját próba vagy kilépés szakaszaikban tartózkodnak.

10.1. lemma. . *Legyen A egy olyan algoritmus, amely megoldja a kölcsönös kizárás feladatát (minden felhasználói csoportra). Legyen $U_1, \ldots, U_n$ egy tetszőleges felhasználói csoport, és legyen B az A kombinációja az adott felhasználói csoporttal. Legyen s a B egy elérhető állapota.*

Ha a P_i folyamat az s állapotban a próba vagy a kilépés szakaszban van, akkor az s-ben megengedett a P_i folyamat valamelyik helyileg ellenőrzött művelete.

Bizonyítás. Az általánosság megszorítása nélkül feltételezhetjük, hogy minden $U_1, \ldots, U_n$ felhasználó visszaadja az erőforrást.

Legyen α a B-nek egy véges végrehajtási sorozata, amely s-sel zárul. Indirekt módon tegyük fel, hogy a P_i folyamat az s állapotban vagy a próba, vagy pedig a kilépés szakaszban található, és s-ben nem megengedett a P_i folyamat semmilyen helyileg ellenőrzött művelete. Ekkor azt állítjuk, hogy P_i-beli esemény nem fordul elő az α előtag után a B egyetlen az α-t kiterjesztő végrehajtási sorozatában sem. Ez abból a tényből következik, hogy a helyileg ellenőrzött műveletek engedélyezését csak a helyi folyamat állapota határozza meg, valamint abból a tényből, hogy a jólformáltság megakadályozza a P_i folyamat bemeneteit, míg a P_i folyamat Pr-ben vagy Ki-ben található.

Legyen α' a B-nek egy pártatlan végrehajtási sorozata, amely kiterjeszti α-t és amelyben az α előtag után nem fordul elő semmilyen `próbál` esemény. A haladás feltételének ismételt használata és az a tény, hogy a felhasználók mindig visszaadják az erőforrást, maga után vonja, hogy a P_i folyamat biztosan végrehajt majd egy $\texttt{belép}_i$ vagy egy $\texttt{halad}_i$ műveletet. Ez azonban ellentétben áll azzal, hogy α' nem tartalmaz további P_i-beli műveleteket. □

10.3.. Dijkstra algoritmusa a kölcsönös kizárás megoldására

Az első aszinkron író/olvasó közös memória modellt használó kölcsönös kizárási algoritmust Edsger Dijkstra javasolta 1965-ben. Ez az algoritmus Dekker kétfolyamatos megoldását vette alapul. A ma ismert algoritmusok között nem ez a legelegánsabb vagy a leghatékonyabb megoldása, és a legszigorúbb feltételeket sem teljesíti. Azonban számos oknál fogva itt mégis bemutatjuk. Először is azért, mert ez a legelső algoritmus, amelyet „osztottnak" nevezhetünk. Másodszor pedig azért, mert néhány érdekes algoritmus ötletet tartalmaz. És végül harmadszor azért, mert az algoritmus segítségével könnyen be tudjuk mutatni az aszinkron közös memória modellben használt érvelési technikákat.

10.3.1.. Az algoritmus

Először az algoritmus kódját mutatjuk be olyan hagyományos „pszeudokód" segítségével, mint amilyent Dijkstra használt az eredeti cikkben. Bár ennek a kódnak

vázlatosan is eléggé érthető kell lennie, valószínűleg nem eléggé világos, hogyan lehet a mi modellünkbe átültetni. Az algoritmust DijkstraKK algoritmusnak nevezzük.

10.1. algoritmus. A DijkstraKK

Közös változók:
váltás $\in \{1, \ldots, n\}$, kezdetben tetszőleges, minden folyamat írhatja és olvashatja
$\forall i \ (1 \leq i \leq n)$:
    jelző$(i) \in \{0, 1, 2\}$, kezdetben 0, a P_i folyamat írhatja
        és minden folyamat olvashatja

A P_i folyamat:

```
    ** Haladás szakasz **

    próbál_i
L:  jelző(i) := 1
    while váltás ≠ i do
        if jelző(váltás) = 0 then váltás := i
    jelző(i) := 2
    for j ≠ i do
        if jelző(j) = 2 then goto L
    belép_i

    ** Belépési szakasz **

    kilép_i
    jelző(i) := 0
    halad_i
```

- -

A közös változók a következők: *váltás*, amely egy egész érték az $\{1, \ldots, n\}$ halmazból és folyamatonként egy *jelző*(i), $1 \leq i \leq n$, kezdetben minden *jelző*(i) 0, majd pedig a $\{0, 1, 2\}$ halmazból vesz fel értéket. A *váltás* változó egy megosztottan írható/megosztottan olvasható regiszter, amelyet minden folyamat írhat és olvashat. Minden *jelző*(i) egy kizárólagosan írható/megosztottan olvasható regiszter, melyet csak a P_i folyamat írhat és minden folyamat olvashat.

A P_i folyamat első részében saját *jelzőjét* 1-re állítja és ezután ismételten megvizsgálja a *váltás* változót, azaz a *váltás* $= i$ egyenlőséget. Ha a *váltás* nem egyenlő i-vel és ha a *váltás* aktuális tulajdonosa nem aktív, akkor a P_i folyamat elvégzi a *váltás* $:= i$ utasítást. Amint *váltás* $= i$-vel, a P_i folyamat a végrehajtást a második résszel folytatja.

A P_i folyamat a második részben újból beállítja a saját *jelzőjét*, ez alkalommal 2-re, majd megvizsgálja, hogy egy másik folyamat *jelzője* egyenlő-e 2-vel. A többi folyamat *jelzőjének* ellenőrzését bármilyen sorrendben elvégezheti. Ha az ellenőrzés sikerrel ér véget, akkor a P_i folyamat belép saját belépési szakaszába, másképp pedig visszalép az első részhez. Amikor a P_i folyamat elhagyja a belépési szakaszt, akkor *jelzőjét* ismét 0-ra állítja.

Mielőtt bármit bizonyítanánk a DijkstraKK algoritmusról, előbb szükségünk van arra, hogy ezt a formális állapotgép modelljének szemszögéből értsük meg, mert nem nyilvánvaló, hogyan kell a kódot egy automatává alakítani.

Először is elvárnánk, hogy egy folyamat minden egyes állapota megfeleljen saját helyi változói értékének, és néhány olyan információnak, mely a kódban nem kifejezetten szerepel, beleértve

- temporális változókat, amelyeket az éppen olvasott közös változók értékeinek tárolására használjuk;
- egy utasításszámlálót, amely jelzi, hol található a folyamat a saját kódjának

végrehajtásában;

- temporális változókat, amelyeket a program vezérlése vezet be (ilyen halmazváltozót például a for ciklus használhat a sikeresen ellenőrzött folyamatok indexeinek nyomonkövetése céljából);

- szakaszjelölést, *Ha*, *Pr*, *Be* vagy *Ki* (*Ha* jelöli a haladás szakaszt, a *Pr* a kódnak a `próbál`$_i$-től a következő `belép`$_i$-ig terjedő részét, *Be* jelöli a belépési szakaszt és végül *Ki* jelöli a kódnak a `kilép`$_i$ eseményétől a következő `halad`$_i$ eseményéig terjedő részét).

Minden folyamatnak egyedi kezdőállapota van. A kezdőállapot a helyi változók meghatározott értékeiből, a temporális változók tetszőleges értékeiből, az utasításszámlálóból és a haladás szakaszt mutató szakaszjelölőből áll. Minden közös változó kezdőértéke egy megadott értékkel egyenlő.

Az automata lépései a kód lépéseit követi; valójában van néhány olyan kétértelmű rész a kódban, amelyet az automatának kell feloldania. Bár a kódban szerepel a helyi és az közös változók módosítása, nem említi, mi történik az implicit változókkal (a temporális változókkal, az utasításszámlálókkal, valamint a szakaszjelölő változókkal). Például, amikor bekövetkezik a `probál`$_i$ esemény, P_i utasításszámlálójának a kódban az L utasításhoz kellene ugornia és P_i szakaszjelölőjének pedig a *Pr* értéket kellene kapnia. Ezeket a módosításokat mindenképpen le kell írni az automatában.

A kód nem említi azt sem, hogy mely részek tartalmaznak oszthatatlan lépéseket. Ismernünk kell ezeket a lépéseket ahhoz, hogy körültekintően érveljünk az algoritmussal kapcsolatban. A DijkstraKK-ban az oszthatatlan lépések a felhasználói felület `próbál`, `belép`, `kilép` és `halad` lépései, az közös változók egyéni írása és olvasása, valamint néhány helyi számítási lépés. Van egy kis árnyalatnyi különbség is: a *jelző*(*váltás*)=0 teszt nem követel két különböző olvasást, mivel a *váltást* épp az előző sorban olvastuk, ezért használhatjuk a *váltás* helyi másolatát.

Ezeket a kétértelműségeket úgy oldjuk fel, hogy a DijkstraKK algoritmust kézzel átírjuk *előfeltétel/hatás* stílusúra, mely stílust a 8. fejezetben használtunk. Ezáltal a kód jóval hosszabb lesz, azonban ily módon minden átmenetet pontosan leírhatunk. Az olvashatóság kedvéért a különböző műveletek kódrészleteit olyan sorrendben írjuk, amilyen sorrendben feltételezzük a végrehajtásukat, bár megjegyezzük, hogy a formális modellben ennek a sorrendnek nincs jelentősége – bármilyen művelet előfordulhat bármikor, amikor az megengedett. A *Ha*, *Pr*, *Be* és a *Ki* szakaszjelölések az utasításszámláló értékeibe vannak kódolva: *Ha* megfelel a `halad`-nak, *Pr* megfelel a `jelző_beállít_1`, `váltás_vizsgál`, `jelző_vizsgál`, `váltás_beállít`, `jelző_beállít_2`, `ellenőriz`, valamint a `próbál_elhagy`-nak; *Be* megfelel a `belép`-nek és *Ki* megfelel a `beállít` és `kilép_elhagy`-nak. Megjegyezzük, hogy minden kódrészletet oszthatatlanul hajtunk végre.

10.2. algoritmus. A DijkstraKK (átírva)

Közös változók:
váltás $\in \{1, \ldots, n\}$, kezdetben tetszőleges
minden i-re $(1 \leq i \leq n)$:
  jelző$(i) \in \{0, 1, 2\}$, kezdetben 0

A P_i folyamat műveletei:

Bemeneti:
  próbál$_i$
  kilép$_i$

Kimeneti:
  belép$_i$
  halad$_i$

Belső:
  jelző_beállít_1$_i$
  váltás_vizsgál$_i$
  jelző_vizsgál$(j)_i$, $1 \leq j \leq n, j \neq i$
  váltás_beállít$_i$
  jelző_beállít_2$_i$
  ellenőriz$(j)_i$, $1 \leq j \leq n, j \neq i$
  beállít$_i$

A P_i folyamat állapotai:
p_sz $\in \{$*haladás*, *jelző_beállítás_1*, *váltás_vizsgálat*, *jelző*(j)*_vizsgálat*, *váltás_beállítás*, *jelző_beállítás_2*, *ellenőrzés*, *próbál_elhagyás*, *belépés*, *beállítás*, *kilép_elhagyás*$\}$, kezdetben *haladás*
S, a folyamatindexek halmaza, kezdetben $\emptyset$

A P_i folyamat átmenetei:

próbál$_i$
  Hatás:
    p_sz := *jelző_beállítás_1*

jelző_beállít_1$_i$
  Előfeltétel:
    p_sz = *jelző_beállítás_1*
  Hatás:
    jelző(i) := 1
    p_sz := *váltás_vizsgálat*

váltás_vizsgál$_i$
  Előfeltétel:
    p_sz = *váltás_vizsgálat*
  Hatás:
    if *váltás* = i **then**
      p_sz := *jelző_beállítás_2*
    else *p_sz* :=
      jelző_vizsgálat(*váltás*)

jelző_vizsgál$(j)_i$
  Előfeltétel:
    p_sz = *jelző_vizsgálat*(j)
  Hatás:
    if *jelző*$(j) = 0$ **then**
      p_sz := *váltás_beállítás*
    else *p_sz* := *váltás_vizsgálat*

váltás_beállít$_i$
  Előfeltétel:
    p_sz = *váltás_beállítás*
  Hatás:
    váltás := i
    p_sz := *jelző_beállítás_2*

jelző_beállít_2$_i$
  Előfeltétel:
    p_sz = *jelző_beállítás_2*
  Hatás:
    jelző(i) := 2
    $S := \{i\}$
    p_sz := *ellenőrzés*

```
ellenőriz(j)_i
    Előfeltétel:
        p_sz = ellenőrizés
        j ∉ S
    Hatás:
        if jelző(j) = 2 then
            S := ∅
            p_sz := jelző_beállítás_1
        else
            S := S ∪ {j}
            if |S| = n then
                p_sz := próbál_elhagyás

belép_i
    Előfeltétel:
        p_sz = próbál_elhagyás
    Hatás:
        p_sz := belépés

kilép_i
    Hatás:
        p_sz := beállítás

beállít_i
    Előfeltétel:
        p_sz = beállítás
    Hatás:
        jelző(i) := 0
        S := ∅
        p_sz := kilép_elhagyás

halad_i
    Előfeltétel:
        p_sz = kilép_elhagyás
    Hatás:
        p_sz := haladás
```

- -

Az átírás könnyen érthető. Megjegyezzük, hogy az új stílus elősegíti a kisebb javítások kifejezését, például a **váltás_beállít**$_i$ művelet megengedi, hogy a P_i folyamat egyenesen a második állapotba kerüljön anélkül, hogy a *váltás*-t újból tesztelné.[1]

10.3.2.. Az algoritmus helyessége

Ebben a fejezetben röviden bemutatjuk a DijkstraKK algoritmus bizonyítását. Ez egy nyers, *működésen* alapuló bizonyítás lesz, azaz a végrehajtásról szóló érvelés. A következő, 10.3.3. szakaszban egy alternatív bizonyítást mutatunk be, amely a kölcsönös kizárás feltételének stílusosabb bizonyítása és invariáns állításokat vesz alapul.

Három lemma segítségével megmutatjuk, hogy a DijkstraKK teljesíti a követelményeket.

10.2. lemma. . *A* DijkstraKK *algoritmus minden felhasználó számára garantálja a jólformáltságot.*

Pontosabban fogalmazva ezen azt értjük, hogy a DijkstraKK és a felhasználók bármely halmaza együttesének (összekapcsolásának) bármely végrehajtási sorozatára és az U_i felhasználó és a DijkstraKK algoritmus közötti kölcsönhatásokat leíró részsorozat jólformált az i-edik felhasználó számára.

Bizonyítás. A kódot vizsgálva könnyen ellenőrizhetjük, hogy a DijkstraKK algoritmus megtartja a jólformáltság tulajdonságát minden felhasználó esetében.

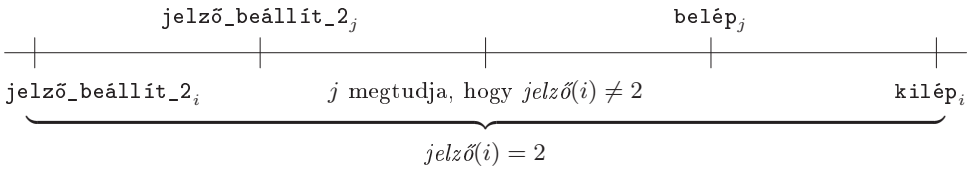
[1] Az állapotokbn használt S jelöléssel kapcsolatban megjegyezzük, hogy a szerző később helyenként S_i-vel jelöli a P_i folyamat S változóját. *A lektor.*

Mivel feltételezésünk szerint a felhasználók is megtartják a jólformáltságot, a 8.11. tétel szerint a rendszer csak jólformált sorozatokat eredményez. □

10.3. lemma. . *A* DijkstraKK *algoritmus eleget tesz a kölcsönös kizárás követelményeinek.*

Ezen azt értjük, hogy a DijkstraKK algoritmus és a felhasználók bármely csoportjának együttesében nincs olyan elérhető állapot, amelyben egynél több felhasználó tartózkodik a *Be* belépési szakaszban.

Bizonyítás. Indirekt módon. Tegyük fel, hogy valamilyen elérhető állapotban az U_i és az U_j $(i \neq j)$ felhasználó egyidejűleg a *Be* szakaszban található. Vegyünk egy olyan végrehajtási sorozatot, amely ehhez az állapothoz vezet. A kód alapján, mindkét folyamat, a P_i és a P_j folyamat végrehajtja a `jelző_beállít_2` lépést mielőtt belépne a saját belépési szakaszába. Vegyünk mindegyik folyamatnál az utolsó ilyen lépést és tegyük fel az általánosság megszorítása nélkül, hogy először a `jelző_beállít_2`$_i$ következik. Akkor a *jelző*(i) ettől a ponttól kezdve egyenlő marad 2-vel addig, amíg P_i elhagyja a *Be*-t, amelynek azután kell megtörténnie, miután a P_j folyamat belép a *Be* szakaszba, aszerint a feltételezés szerint, hogy egyidejűleg mindkettő *Be*-ben van. Így a *jelző*(i) értéke a `jelző_beállít_2`$_j$ eseménytől egyenlő 2-vel, addig amíg a P_j folyamat be nem lép a *Be* szakaszba. Lásd a 10.5. ábrát. Azonban ezalatt a P_j folyamatnak tesztelnie kell a *jelző*(i)-t és 2-től különbözőnek találja, amely ellentmondás. □



10.5.. ábra. Az események sorrendje a 10.3. lemma bizonyításában.

10.4. lemma. . *A* DijkstraKK *algoritmus garantálja a haladást.*

Bizonyítás. A kilépési szakasz bizonyítása könnyű: ha egy pártatlan végrehajtási sorozat bármely pontján az U_i felhasználó a kilépési szakaszban van, akkor a P_i folyamat továbblép. Legfeljebb két ilyen lépés után a P_i folyamat végrehajt egy `halad`$_i$ műveletet U_i-t a saját haladási szakaszába küldve.

Most a próba szakaszban vizsgáljuk a haladás tulajdonságot. Tegyük fel az ellentmondás kedvéért, hogy α egy pártatlan végrehajtási sorozat, amely elér egy olyan pontot, ahol legalább egy felhasználó a *Pr* szakaszban van és nincs egy felhasználó sem a *Be* szakaszban, valamint feltételezzük, hogy ezután a pont után egy felhasználó sem fog többé belépni a *Be* szakaszba.

Néhány bonyolult elem törlésével kezdjük. Először, bármely folyamat a *Ki*-ben továbblép, így legfeljebb két lépés után el kell érnie a *Ha*-t. Tehát, néhány α-beli elem után, minden folyamat a *Pr*-be vagy a *Ha*-ba kerül. Másodszor, mivel csak végesen sok folyamat van a rendszerben, néhány α-beli elem után, semmilyen új folyamat nem lép be *Pr*-be. Így néhány α-beli pont után, minden folyamat a *Pr*-ben vagy a *Ha*-ban található és egyetlen folyamat sem vált többé szakaszt. Ebből az következik, hogy α-nak van egy α_1 utótagja, amelyben a *Pr*-beli (*Pr*-ben rögzített számú folyamat van) folyamatok végtelen sokszor továbblépnek és már nem váltanak szakaszt. Ezeket a folyamatokat *versenyzőknek* nevezzük.

Megjegyezzük, hogy legfeljebb egy α_1-beli lépés után minden P_i versenyző biztosítja azt, hogy a *jelző*$(i)\geq 1$ és $\geq$ 1-nek marad α_1 további részében is. Így feltételezhetjük, az általánosság megszorítása nélkül, hogy minden P_i versenyzőre a *jelző*$(i) \geq 1$ az α_1-ben.

Azaz, ha a *váltás* módosul az α_1-ben, akkor ezt egy versenyző indexére cseréli. Sőt mi több, a következő segédállítást is megfogalmazhatjuk. □

10.5. segédtétel. . *α_1-ben a váltás biztosan felveszi egy versenyző indexét.*

Bizonyítás. Feltételezzük, hogy nem igaz, azaz feltételezzük, hogy a *váltás* értéke egyenlő marad egy nem-versenyző indexével α_1-ben. Vegyünk egy tetszőleges P_i versenyzőt.

Ha p_sz_i valaha eléri a `váltás_vizsgál`-t (azaz, a while ciklus kezdetét az eredeti kódban), akkor azt állítjuk, hogy P_i a *váltás*-t i-re módosítja. Ez azért van, mert P_i először végrehajt egy `váltás_vizsgál`$_i$-t és azt találja, hogy a *váltás* egyenlő valamilyen $j \neq i$-vel. Azután végrehajt egy `jelző`(j)`_vizsgál`$_i$-t és azt találja, hogy *jelző*(j)=0, mivel P_j nem versenyző. A P_i folyamat ezért végrehajtja a `váltás_beállít`$_i$ műveletet, *váltás*-t i-re állítva.

Most bebizonyítjuk, hogy P_i eléri a `váltás_vizsgál`-t. Az egyetlen eset, amikor nem érheti el, akkor valósul meg, amikor P_i sikeres az összes többi folyamat *jelzőjének* ellenőrzésében (az eredeti kód második állapotában) és végrehajtja a `próbál_elhagy` műveletet. Azonban az α_1-ről feltételeztük, hogy ebben a P_i soha nem éri el a *Be* szakaszt. Ez csak úgy lehetséges, hogy valamelyik `ellenőriz` művelet nem igaz és visszaállítja P_i-t a `jelző_beállít_1`-re, amelyet majd a `váltás_vizsgál` művelettel folytat.

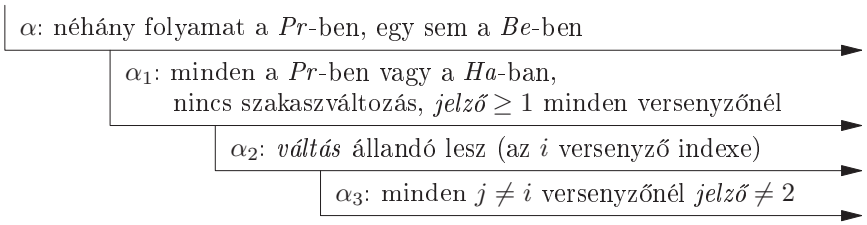
Így P_i eléri a `váltás_vizsgál`-t és azután elvégzi a *váltás*:= i műveletet. Mivel P_i versenyző volt, a szükséges ellentmondáshoz jutottunk. □

Amint egyszer a *váltás* értékét egy versenyző indexére állítottuk, azután mindig egyenlő lesz *valamelyik* versenyző indexével, habár a *váltás* értéke különböző versenyzők értékét is felveheti. (Ez azért lehetséges, mert néhány folyamat ugyanazon időpontban tarthat a `váltás_beállít` műveletnél.) Bármely későbbi `váltás_vizsgál` és rákövetkező `jelző_vizsgál` a *jelző*(*váltás*)$\geq$ 1-hez vezet, mivel minden P_i versenyzőre *jelző*$(i) \geq 1$. Így a *váltás* nem módosul a fenti tesztek eredményeként. Ezért végül is a *váltás* megállapodhat egy végső (versenyző) indexnél. Legyen α_2 az α_1-nek egy olyan utótagja, amelyben a *váltás* értéke megállapodott valamely versenyző indexénél, legyen ez az index i.

A következőkben azt állítjuk, hogy α_2-ben bármely $j \neq i$ versenyző végül is oda jut, hogy a versenyző utasításszámlálója végtelen ciklus kerül a `váltás_vizsgál` és a `jelző_vizsgál` között. (Azaz végtelen ciklusba kerül a while utasításnál.) Ugyanis ha valamikor eléri az `ellenőriz` műveletet (a második részben), mivel nem éri el a *Be*-t, végül vissza kell térnie a `jelző_beállít_1`-hez. Itt azonban végtelen ciklusba kerül, mivel $váltás = i \neq j$ és $jelző(i) \neq 0$ az α_2-ben. Legyen α_3 az α_2-nek egy olyan utótagja, amelyben minden P_i-től különböző versenyző a `váltás_vizsgál` és a `jelző_vizsgál` közötti végtelen ciklusban található. Megjegyezzük, hogy ez azt jelenti, hogy minden P_i-től különböző versenyző *jelzője* végig egyenlő 1-gyel az α_3-ban.

Magyarázatunk következtetéseként állíthatjuk, hogy α_3-ban a P_i folyamatnak (annak a folyamatnak, amelynek indexe egyenlő a *váltás* indexével) semmi sem áll útjában a *Be* eléréséhez. Például, ha P_i végrehajtja a `váltás_vizsgál`-t, akkor P_i azt találja, hogy $váltás = i$ és így folytatja a *beállít_jelző_2*-vel. Mivel semmilyen más folyamatnál sem áll fenn a $jelző = 2$ egyenlőség, a P_i folyamat minden *ellenőrzés* műveletet sikeresen hajt végre és belép a *Be* szakaszba.

Az egymás utáni utótagok ábrázolása, amelyek ebben a bizonyításban szerepelnek, a 10.6. ábrán látható.



10.6.. ábra. Egymás utáni utótagok a 10.4. lemma bizonyításában.

10.6. tétel. . *A* DijkstraKK *algoritmus megoldja a kölcsönös kizárási feladatot.*

Bizonyítás. Bár a fenti állítások helyesek, ezek inkább bonyolult és ötletszerű állítások. Jó lenne, ha rendszerezettebben tudnánk ilyen bizonyításokat levezetni. A következő fejezetben megadunk egy alternatív bizonyítást a kölcsönös kizárás feltételére, invariáns állításokat használva. A haladás feltétele is rendszerezettebben bizonyítható a temporális logika segítségével, de ezt ebben a könyvben nem tesszük meg. □

10.3.3.. A kölcsönös kizárás feltételének bizonyítása állítások segítségével

A szinkron hálózati modellben sok szép és rendezett bizonyítás a rendszer néhány menet utáni állapotáról szóló invariáns állításokon alapszik. Az aszinkron

környezetben nem létezik a menet fogalom, azonban invariánsok még mindig használhatók. Ahhoz, hogy az egyes folyamatok által megtett tetszőleges számú lépés után igazolni tudjuk a rendszerállapotokról szóló állítások teljesülését, finomabb felbontást kell használni. Természetesen általában nehezebb meghatározni aszinkron rendszer állapotáról szóló, tetszőleges számú lépés után is érvényes állításokat, mint szinkron rendszer esetében. Ilyen állítások bizonyítása is jóval nehezebb általában. Azonban legtöbbször megéri a fáradtságot, az invariánsok által nyújtott éleslátás miatt. Az invariáns állítás az egyetlen igazán fontos formális eszköz, amelyet aszinkron algoritmusok helyességének bizonyításánál alkalmazunk.

Most a DIJKSTRAKK algoritmus kölcsönös kizárásra adott feltételének, állításokon alapuló bizonyítását adjuk meg.

10.3. lemma bizonyítása. A kölcsönös kizárás bizonyításához meg kell mutatnunk a következőket.

10.3.1. állítás. *A rendszer bármely elérhető állapotában teljesül*[2] $|\{i : p\_sz_i = \textit{belépés}\}| \leq 1$.

Ezt az állítást egy végrehajtási sorozat hossza szerinti indukcióval szeretnénk bebizonyítani. Azonban, mint általában, az adott állítás nem elég erős ahhoz, hogy önállóan bizonyítsuk – szükségünk van néhány segédinvariánsra. A 10.3.1. állítást a következő két állítás következményeként bizonyítjuk.

10.3.2. állítás. *Ha bármely elérhető állapotban* $p\_sz_i \in \{\textit{próbál\_elhagyás, belépés, beállítás}\}$, *akkor* $|S_i| = n$.

10.3.3. állítás. *Egyik elérhető állapotban sem létezik olyan* i *és* j, $i \neq j$, *hogy* $i \in S_j$ *és* $j \in S_i$.

Ha mindkét állítás, a 10.3.2. és a 10.3.3. igaz, akkor a 10.3.1. állítás azonnal következik. Az ellentmondás kedvéért tegyük fel, hogy a rendszernek egy bizonyos elérhető állapotában létezik két egymástól különböző P_i és P_j folyamata úgy, hogy a $p\_sz_i = p\_sz_j =$ `belép`. Akkor a 10.3.2. állítás szerint $|S_i| = |S_j| = n$. Akkor azonban $j \in S_i$ és $i \in S_j$, amely ellentmond a 10.3.3. állításnak.

A 10.3.2. állítást könnyen be lehet bizonyítani a végrehajtási sorozat hossza szerinti indukcióval. Alapesetben az állítás igaz, mivel minden folyamata kezdeti állapotban *Ha*-ban van. Az indukciós lépést esetelemzéssel bizonyítjuk egyenként figyelembe véve minden művelettípust. Ebben az esetben, megsérthetik az állítást azok a lépések, amelyek p_sz_i-t a felsorolt értékek halmazába viszik, valamint azok, amelyek S_i-t visszaállítják az $\emptyset$ halmazra, azaz az `ellenőriz`$_i$ és a `beállít`$_i$ műveletek. Az `ellenőriz`$_i$ esetében az egyetlen módja annak, hogy az $p\_sz_i \in$ {`próbál_elhagy`, `belép`, `beállít`} feltétel teljesüljön a lépés után, hogy $|S_i| =$

[2]Emlékezzünk arra, hogy a rendszer állapota a felhasználók és a folyamatok állapotának együttese, valamint a közös változók értékei.

n, amely épp a szükséges állítás. A $\texttt{beállít}_i$ esetében a folyamat a megjelölt értékek halmazát változatlanul hagyja, tehát az állítás nyilvánvalóan teljesül.

Maradt még a 10.3.3. állítás bizonyítása. Ez két egyszerű tényt vesz figyelembe. Az első azt korlátozza, hogy a P_i folyamat hol tarthat kódjának végrehajtásában, ha $S_i \neq \emptyset$.

10.3.4. állítás. *Ha bármely elérhető állapotban $S_i \neq \emptyset$, akkor $p_sz_i \in \{$ellenőrzés, próbál_elhagyás, belépés, beállítás$\}$.*

Ez az állítás is a végrehajtási sorozat hossza szerinti egyszerű indukcióval bizonyítható. Az alapeset könnyű, mivel a rendszer kezdeti állapotában $|S_i| = \emptyset$. Az indukciós lépésben az állítás megsértését azok az események okozhatják, amelyek S_i-t $\emptyset$-tól különbözőre állítják, valamint azok, amelyek arra kényszerítik p_sz_i-t, hogy a fent felsorolt elemekből álló halmazon kívül kerüljön, azaz a $\texttt{jelző\_beállít\_2}_i$, az $\texttt{ellenőriz}_i$, és a $\texttt{beállít}_i$. A $\texttt{jelző\_beállít\_2}_i$ azonban végrehajtja a $p\_sz_i := \texttt{ellenőriz}$ értékadást. Hasonló módon, amikor az $\texttt{ellenőriz}_i$ a p_sz_i-t a felsorolt értékeken kívülire állítja, akkor elvégzi az $S_i := \emptyset$ utasítást is. Végül, $\texttt{beállít}$ végrehajtja az $S_i := \emptyset$-t. Így a fenti események mind megtartják a feltételt.

A második tény azt állítja, hogy a $jelző(i) = 2$ akkor, amikor a P_i folyamat a kód bizonyos pontjainál tart.

10.3.5. állítás. *Ha bármely elérhető állapotban $p_sz_i \in \{$ellenőrzés, próbál_elhagyás, belépés, beállítás$\}$, akkor $jelző(i) = 2$.*

Ez is a végrehajtási sorozat hossza szerinti egyszerű indukcióval bizonyítható. Összevetve a fenti tényeket a következőt állíthatjuk.

10.3.6. állítás. *Ha bármely elérhető állapotban $S_i \neq \emptyset$, akkor $jelző(i) = 2$.*

Most bebizonyíthatjuk a 10.3.3. állítást, újból a végrehajtási sorozat hossza szerinti indukcióval. Az alapeset könnyű, mivel a rendszer kezdőállapotában minden S_i halmaz üres. Az indukciós lépésben csak az az egyetlen esemény sértheti meg az állítást, amelyik hozzáad egy j elemet az S_i-hez, valamely i-re és j-re, ahol $i \neq j$, azaz az $\texttt{ellenőriz}(j)_i$ esemény. Tekintsük azt az esetet, amikor j az S_i-be kerül egy $\texttt{ellenőriz}(j)_i$ hatására. Ez az esemény csak akkor történhet meg, ha $jelző(i) \neq 2$. Ekkor azonban a 10.3.6. állítás szerint $S_j = \emptyset$, tehát $i \notin S_j$. Így ez a lépés nem sértheti meg az állítást. □

10.3.4.. Futásidő

Ebben a szakaszban megadunk és bizonyítunk egy felső korlátot arra az időtartamra, amelyet egy végrehajtási sorozat olyan pontjától számítunk, amelyben

egy bizonyos folyamat a Pr-ben található és nincs egy folyamat sem a Be-ben, egészen addig amíg egy folyamat belép a Be-be.

Az „idő" fogalmának a pontatlansága az első nehézség, amellyel ebben a felső korlátra vonatkozó bizonyításban találkozunk – ellentétben a szinkron esettel, most nem lehet a fordulókat számlálni. Feltételezzük, hogy minden lépés bizonyos valós idejű pillanatban történik és feltételezzük azt is, hogy a végrehajtás a valós idejű 0-nál kezdődik. Feltételezzük, hogy minden folyamat két egymás utáni lépése (amikor ezek megengedett lépések) között eltelt idő felső korlátja l; emlékezzünk arra, hogy egy esemény előfeltétel/hatás alakban megadott teljes kódjáról azt feltételeztük, hogy egyetlen lépésnek számít. Ugyanúgy feltételezzünk, hogy tetszőleges felhasználó Be szakaszban eltöltött maximális idejének felső korlátja c. Ezekből a feltételezett korlátokból érdekes események bekövetkezi idejének felső korlátaira következtethetünk.

10.7. tétel. . *Ha a* DijkstraKK *algoritmus végrehajtása során feltételezzük, hogy egy adott pillanatban valamelyik felhasználó a Pr-ben található és nincs egy felhasználó sem a Be-ben, akkor $\mathcal{O}(ln)$ időn belül valamelyik felhasználó belép a Be szakaszba.*

A nagy-$\mathcal{O}$ képletében szereplő konstans független l-től, c-től és n-től. Ez a bizonyítás egyedi és egy kissé bonyolult, a haladási feltételek bizonyításából vesz át néhány elemet.

Bizonyítás. Indirekt tegyük fel, hogy a lemma hamis és vegyünk egy olyan végrehajtási sorozatot, amelyben egy bizonyos ponton P_i a Pr-ben található és nincs egy folyamat sem a Be-ben, valamint feltételezzük, hogy legalább kln ideig, elég nagy k konstans esetében, egy folyamat sem lép be a Be szakaszba. A következőkben a k konstans jóval nagyobb, mint a nagy-$\mathcal{O}$-ban szereplő konstansok értéke.

Először is könnyű belátni, hogy az elemzés kezdeti pontjától eltelt idő addig, hogy egyetlen felhasználó sincs a Be-ben vagy a Ki-ben, legfeljebb $\mathcal{O}(l)$.

Másodszor, állítjuk, hogy további $\mathcal{O}(ln)$ időn belül a P_i folyamat végrehajt egy `váltás_vizsgál`$_i$ műveletet. Mindez azért van, mert legrosszabb esetben P_i ennyi időt tölthet a második szakaszban a jelzők vizsgálatával, mielőtt visszafordulna a `jelző_beállít_1`-hez. Tudjuk, hogy vissza kell forduljon a `jelző_beállít_1`-hez, mert másképp belép a Be-be, amelyről azt feltételeztük, hogy nem történhet meg ilyen gyorsan.

Harmadszor, állítjuk, hogy további $\mathcal{O}(l)$ idő szükséges a P_i folyamat `váltás_vizsgál`$_i$ műveletétől egészen addig, amíg a *váltás* változó egyenlő egy versenyző indexével. Ezt a következő esetelemzésben látjuk be. Ha amikor P_i végrehajtja a `váltás_vizsgál`$_i$ műveletet, a *váltás* már egyenlő egy versenyző indexével, akkor már megkaptuk az eredményt, így feltételezzük, hogy nem ez az eset áll fenn, azaz feltételezzük, hogy *váltás*$= j$, ahol j nem versenyző. Ezen teszt után $\mathcal{O}(l)$ időn belül P_i végrehajtja a `jelző_vizsgál`$(j)_i$ műveletet. Ha P_i azt találja, hogy *jelző*$(j) = 0$, akkor P_i *váltás*-t i-re állítja, amely egy versenyző indexe és újból megkaptuk az eredményt. Azonban ha azt találja, hogy *jelző*$(j) \neq 0$, akkor azt jelenti, hogy a `váltás_vizsgál`$_i$ és a `jelző_vizsgál`$(j)_i$ között a P_j

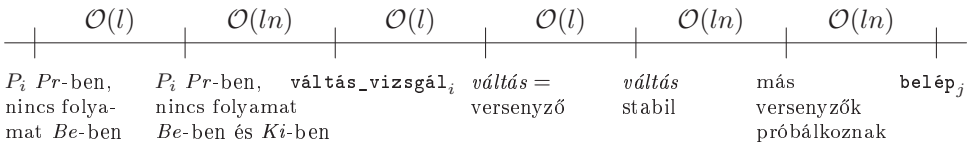
folyamat belépett a próba szakaszba és versenyzővé vált. Ha a *váltás* időközben nem módosult, akkor a *váltás* egyenlő a P_j versenyző indexével és ekkor készen vagyunk. Ha a *váltás* időközben módosult, akkor megkapta egy versenyző indexét és ekkor is megkapjuk az eredményt.

Negyedszer, $\mathcal{O}(l)$ idő után olyan pontba érkezünk, ahol a *váltás* értéke megállapodott egy bizonyos versenyző indexénél, legyen ez j, és ezután egy folyamat sem hajt végre további `váltás_vizsgál`-t vagy `jelző_beállít_2`-t (legalábbis az elemzés kezdetétől számított *kln* időn belül).

Ötödször, azt állítjuk, hogy minden P_j-től különböző versenyző programszámlálója további $\mathcal{O}(ln)$ idő elteltével a $\{$*váltás_vizsgál*, *jelző_vizsgál*$\}$ halmazból vesz fel értéket. Másképp ezek a versenyzők elérték volna a *Be* szakaszt, amiről azt feltételeztük, hogy nem történhet meg ilyen hamar.

Végül hatodszor, újabb $\mathcal{O}(ln)$ idő elteltével P_j sikeresen belép a *Be* szakaszba. Ez ellentétben van azzal a feltételezéssel, hogy egy folyamat sem lép be a *Be* szakaszba ennyi idő alatt.

A bizonyításban szereplő események és időhatárok sorrendje a 10.7. ábrán látható. □



10.7.. ábra. Az események és időhatárok sorrendje a 10.7. tétel bizonyításában.

10.4.. Erősebb feltételek a kölcsönös kizárás algoritmusaira

Bár a DijkstraKK algoritmus garantálja a kölcsönös kizárást és a haladást, van néhány fontos tulajdonság, amelyeket nem garantál. Nem garantálja, hogy a *belépési* szakasz *pártatlanul* elérhető legyen a különböző felhasználók számára; például megengedi egy felhasználónak, hogy ismétlődően belépjen a *belépési* szakaszába miközben más felhasználók hiába próbálkoznak a hozzáféréssel, végtelen hosszú ideig akadályba ütköznek. Ezt a helyzetet néha *kizárásnak* vagy *éhezésnek* nevezik.

Figyeljük meg, hogy az itt tárgyalt pártatlanság különbözik az eddigiekben vizsgálttól. Eddig a folyamatok (és a felhasználói automaták) lépéseinek pártatlan végrehajtási sorozatáról beszéltünk, míg most az erőforrások pártatlan elosztásával foglalkozunk. Annak érdekében, hogy ezt a kétfajta pártatlanságot megkülönböztessük, a továbbiakban a folyamatok (és a felhasználói automaták) lépéseinek pártatlan végrehajtási sorozatát alacsony szinten pártatlannak, az erőforrások pártatlan elosztását pedig magas szinten pártatlannak fogjuk nevezni.

A gyakorlatban a magas szintű pártatlanság nem mindig belépési követelmény, számos gyakorlati helyzetben, amelyben kölcsönös kizárást használnak, a felhasználók közötti versengés elég ritka ahhoz, hogy egy felhasználó megtehesse, hogy megvárja amíg az összes vele konfliktusban lévő felhasználó lehetőséget kap. A magas szintű pártatlanság fontossága az erőforrásért folyatott versengés mértékének, valamint az egyes felhasználói programok jelentőségének függvénye.

A DijkstraKK algoritmus egy másik, kevésbé vonzó tulajdonsága, hogy egy *megosztottan írható/megosztottan olvasható* közös regisztert (*váltás*) használ. Ilyen változó megvalósítása számos többprocesszoros rendszerben (és majdnem az összes üzenetküldéses rendszerben) bonyolult és költséges. Ezért jobb lenne olyan algoritmust tervezni, amelyik csak *kizárólagosan írható/megosztottan olvasható* regisztert használ, vagy még jobb olyat, amelyik csak *kizárólagosan írható/kizárólagosan olvasható* regisztert.

Különböző módszerekkel sok, a DijkstraKK algoritmust felülmúló, kölcsönös kizárást biztosító algoritmust terveztek. A fejezet hátralévő részében ezen algoritmusok egy reprezentatív gyűjteményét fogjuk áttekinteni.

Mielőtt nekilátnánk az algoritmusok vizsgálatának, definiáljuk pontosan, hogy egy kölcsönös kizárás algoritmus esetében mit jelent, hogy garantálja a magas szintű pártatlanságot. Attól a környezettől függően, amelyben az algoritmust használjuk, a magas szintű pártatlanságot többféleképpen is megfogalmazhatjuk – mi három jelentést definiálunk. Ezen tulajdonságok mindegyikét $U_1, \ldots, U_n$ felhasználók egy egyedi csoportjával összekapcsolt egyedi A kölcsönös kizárás algoritmusra fogalmazzuk meg.

Kizárásmentesség. Bármely alacsony szinten pártatlan végrehajtási sorozat rendelkezik a következő tulajdonsággal.

1. (Kizárásmentesség a *próba* szakaszra.) Ha minden felhasználó mindig visszaadja az erőforrást, akkor bármely felhasználó, amelyik eléri a Pr szakaszt, a végrehajtási sorozat valamely későbbi pontjában belép a Be szakaszba.
2. (Kizárásmentesség a *kilépés* szakaszra.) Bármely felhasználó, amelyik eléri a Ki szakaszt, a végrehajtási sorozat valamely későbbi pontjában belép a Ha szakaszba.

Vegyük észre, hogy a kizárásmentességi feltétel, az alapvető jólformáltsági, kölcsönös kizárási és haladási feltételhez hasonlóan, kifejezhető úgy, mint egy történet tulajdonság.

A b időkorlát. Bármely alacsony szinten pártatlan végrehajtási sorozatra a megadott időkkel teljesül a következő két állítás.

1. (b időkorlát a *próba* szakaszra.) Ha minden felhasználó mindig visszaadja az erőforrást a megszerzése után c időn belül, és minden Pr vagy Ki szakaszban lévő folyamatra az egymást követő lépések közötti idő legfeljebb l, akkor bármely felhasználó, amelyik eléri a Pr szakaszt, belép a Be szakaszba b időn belül.

2. (b időkorlát a *kilépés* szakaszra.) Ha minden Pr vagy Ki szakaszban lévő folyamatra az egymást követő lépések közötti idő legfeljebb l, akkor bármely felhasználó, amelyik eléri a Ki szakaszt, belép a Ha szakaszba b időn belül.

(Megjegyezzük, hogy b értéke általában l és c függvénye).

Az a megkerülések száma. Vizsgáljunk meg bármely olyan intervallumot a végrehajtási sorozatban, amely akkor kezdődik, amikor a P_i folyamat végrehajt egy helyileg ellenőrzött lépést Pr-ben, és addig tart, ameddig P_i Pr-ben marad. Ez alatt az intervallum alatt, bármely másik P_j, $j \neq i$, felhasználó legfeljebb a-szor léphet be a Be szakaszba.

Az első két fenti esetben a *kilépés* szakaszra olyan magas szintű pártatlansági feltételeket fogalmaztunk meg, amelyek hasonlóak a *próba* szakaszra adott feltételekhez. Azonban a legtöbb algoritmusban a *kilépés* szakasz valójában egészen triviális.

Azt mondjuk, hogy egy A algoritmus *kizárásmentes*, ha a felhasználók minden csoportjára garantálja a kizárásmentességet. A többi magas szintű pártatlansági definíciót hasonlóan kiterjesztjük. Ezek között a pártatlansági feltételek között van néhány egyszerű kapcsolat.

10.8. tétel. . *Legyen A egy kölcsönös kizárás algoritmus, $U_1, \ldots, U_n$ felhasználók egy csoportja, és B az A-nak $U_1, \ldots, U_n$-nel vett összekapcsolása. Ha B-nek van valamilyen véges korlátja a megkerülések számára és kizárásmentes a kilépés szakaszra, akkor B kizárásmentes.*

Bizonyítás. Tekintsük B-nek egy alacsony szinten pártatlan ütemezését, amelyben a felhasználók mindig visszaadják az erőforrást, és tegyük fel, hogy a végrehajtási sorozat valamely pontján P_i Pr-ben van. Indirekt tegyük fel, hogy P_i soha nem lép be a Be szakaszba. A 10.1. lemma alapján ha eddig még nem tette, akkor P_i-nek valamely későbbi pontban végre kell hajtania egy helyileg ellenőrzött lépést. A haladási feltétel ismételt felhasználásával és azzal a feltevéssel, hogy a felhasználók mindig visszaadják az erőforrást együttesen kapjuk, hogy a rendszerben végtelen sok szakasz váltás történik. De ha ez így van, akkor valamelyik P_i-től különböző folyamat végtelen sokszor belép a Be szakaszba, amíg a P_i a Pr-ben marad, viszont így megsérül az az állítás, hogy létezik véges korlát a megkerülések számára. □

10.9. tétel. . *Legyen A egy kölcsönös kizárás algoritmus, $U_1, \ldots, U_n$ felhasználók egy csoportja, és B az A-nak $U_1, \ldots, U_n$-nel vett összekapcsolása. Ha B-nek van valamilyen b időkorlátja (mind a próba, mind a kilépés szakaszra), akkor B kizárásmentes.*

Bizonyítás. Tekintsük B-nek egy alacsony szinten pártatlan ütemezését, amelyben a felhasználók mindig visszaadják az erőforrást, és tegyük fel, hogy a végrehajtási sorozat valamely pontján P_i Pr-ben van. Rendeljünk időt a végrehajtási

sorozat eseményeihez bármilyen monoton, nemcsökkenő, nemkorlátos módszerrel, úgy hogy minden folyamatra a lépések ideje legfeljebb l, és a belépési szakasz ideje legfeljebb c.

Mivel az algoritmusnak b az időkorlátja, ezért P_i legfeljebb b időn belül belép a Be szakaszba, amiből természetesen az is következik, ami a kizárásmentességhez kell, azaz hogy P_i valamely későbbi pontban belép a Be szakaszba. □

A következő alfejezetekben megvizsgálunk néhány algoritmust, amely kielégíti ezen erősebb magas szintű pártatlansági feltételek némelyikét.

10.5.. Kizárásmentes kölcsönös kizárási algoritmusok

Az első továbbfejlesztés, amit bemutatunk egy algoritmus hármas, amelyet Peterson tervezett és amelynek mindhárom tagja garantálja a kizárásmentességet. Az első algoritmus csak két folyamatra működik, de bemutatja a legtöbb alapötletet. Ezt az algoritmust azután kétféle módon kiterjesztjük $n > 2$ folyamatra, az első módszer, hogy a két folyamatra vonatkozó algoritmus egy változatát alkalmazzuk egy $n - 1$ mérkőzésből álló sorozatra, míg a másik, hogy a két folyamatra adott algoritmus egy változatát alkalmazzuk egy versenyre, amelynek révén egyetlen egy győztest kapunk.

10.5.1.. Az algoritmus két folyamat esetében

Kezdjük a két-folyamatos megoldással, amit PETERSON2FOLY-nek hívunk. Általában a két folyamatot a két-folyamatos rendszerben folyamat 1-nek és 2-nek nevezzük. Ebben a részben a kényelem kedvéért mod 2-vel számolunk, és a 2-t azonosítjuk 0-val, így a két folyamatot 0-nak és 1-nek fogjuk hívni. Ha $i \in \{0, 1\}$, akkor az $1 - i$ -t, azaz a másik folyamat indexét $\bar{i}$-vel jelöljük. Az alábbiakban megadjuk a kódot hagyományos stílusban.

10.3. algoritmus. PETERSON2FOLY

Közös változók:
váltás $\in \{0,1\}$, kezdőértéke tetszőleges, írható és olvasható minden folyamat számára
Minden i-re, $i \in \{0,1\}$:
jelző(i) egy logikai változó, kezdőértéke 0,
írható a P_i és olvasható a $P_{\bar{i}}$ folyamat számára

P_i folyamat:

** Haladás szakasz **

`próbál`$_i$
jelző(i) $:= 1$
váltás $:= i$
waitfor *jelző(*$\bar{i}$*)* $= 0$ **or** *váltás* $\neq i$
`belép`$_i$

** Belépési szakasz **

`kilép`$_i$
jelző(i) $:= 0$
`halad`$_i$

A PETERSON2FOLY algoritmusban a DIJKSTRAKK algoritmus folyamataihoz hasonlóan a P_i folyamat először beállítja a *jelző*-jét 1-re, de itt közvetlenül ezután elvégzi a *váltás* $:= i$ beállítást is. Ezek után vár, amíg észre nem veszi, hogy vagy a másik folyamat *jelző*-je 0, vagy pedig a *váltás* $\neq i$ feltétel teljesül. Azaz vagy a másik folyamat nem vesz részt jelenleg az erőforrásért folytatott versenyben, vagy a másik folyamat visszaállította a *váltás* változót, mióta P_i legutóbb beállította azt. Így (kissé furcsán) a *váltás* változó értékének beállítása a másik folyamat indexére engedélyt ad a P_i számára, hogy belépjen a belépési szakaszba.

Hogyan tudjuk átalakítani a formális modellben ezt a programot egy állapotautomatává? Mint a korábbiakban, most is szükségünk van egy program számláló, temporális változók és egy szakasz megjelölés bevezetésére. A kódban van egy kétértelműség, amit fel kell oldanunk, nevezetesen, hogy milyen sorrendben vizsgálja a P_i folyamat a *jelző* és a *váltás* változókat a waitfor szerkezetben. A helyességhez mindkét változó ismételt ellenőrzése szükséges, az egyszerűség kedvéért feltesszük, hogy az ellenőrzés felváltva történik, bár ennél gyengébb feltételeket is használhatnánk.

A könnyebb bizonyíthatóság érdekében átírjuk az algoritmust előfeltétel/hatás alakra. Itt a *Ha* szakasz megjelölés megfelel a *haladás*; a *Pr* megjelölés a *jelző_ beállítás*, *váltás_ beállítás*, *jelző_ ellenőrzés*, *váltás_ ellenőrzés* és *elhagy_ próba*; a *Be* megjelölés a *belépés*; és a *Ki* megjelölés a *visszaállítás* és az *elhagy_ kilépés* állapotoknak.

10.4. algoritmus. PETERSON2FOLY (átírva)

Közös változók:
váltás $\in \{0,1\}$, kezdőértéke tetszőleges, írható és olvasható minden folyamat számára
Minden i-re, $i \in \{0,1\}$:
jelző(i) egy logikai változó, kezdőértéke 0,
írható a P_i és olvasható a $P_{\bar{i}}$ folyamat számára

A P_i folyamat műveletei:

Bemeneti:
- `próbál`$_i$
- `kilép`$_i$

Kimeneti:
- `belép`$_i$
- `halad`$_i$

Belső:
- `jelző_beállít`$_i$
- `váltás_beállít`$_i$
- `jelző_ellenőriz`$_i$
- `váltás_ellenőriz`$_i$
- `visszaállít`$_i$

A P_i folyamat állapotai:
p_sz $\in$ {*haladás, jelző_beállítás, váltás_beállítás, jelző_ellenőrzés, váltás_ellenőrzés, elhagy_próba,belépés, visszaállítás, elhagy_kilépés*},
kezdőállapot a *haladás*

A P_i folyamat átmenetei:

`próbál`$_i$
Hatás:
p_sz := *jelző_beállítás*

`jelző_beállít`$_i$
Előfeltétel:
p_sz = *jelző_beállítás*
Hatás:
jelző(i) := 1
p_sz := *váltás_beállítás*

`váltás_beállít`$_i$
Előfeltétel:
p_sz = *váltás_beállítás*
Hatás:
váltás := i
p_sz := *jelző_ellenőrzés*

`jelző_ellenőriz`$_i$
Előfeltétel:
p_sz = *jelző_ellenőrzés*
Hatás:
if *jelző*($\bar{i}$) = 0 **then**
p_sz := *elhagy_próba*
else
p_sz := *váltás_ellenőrzés*

`váltás_ellenőriz`$_i$
Előfeltétel:
p_sz = *váltás_ellenőrzés*
Hatás:
if *váltás* $\neq i$ **then**
p_sz := *elhagy_próba*
else
p_sz := *jelző_ellenőrzés*

`belép`$_i$
Előfeltétel:
p_sz = *elhagy_próba*
Hatás:
p_sz := *belépés*

`kilép`$_i$
Hatás:
p_sz := *visszaállítás*

`visszaállít`$_i$
Előfeltétel:
$p\_sz = visszaállítás$
Hatás:
$jelző(i) := 0$
$p\_sz := elhagy\_kilépés$

`halad`$_i$
Előfeltétel:
$p\_sz = elhagy\_kilépés$
Hatás:
$p\_sz := haladás$

- -

Most belátjuk, hogy a PETERSON2FOLY algoritmus helyes. A jólformáltsági feltétel teljesülését könnyű ellenőrizni.

10.10. lemma. . *A* PETERSON2FOLY *algoritmus kielégíti a kölcsönös kizárás feltételét.*

Bizonyítás. A bizonyítás invariáns állításokon alapul. Indukcióval könnyű megmutatni a következő állítást.

> **10.5.1. állítás.** *Ha bármely elérhető rendszerállapotban* $jelző(i) = 0$, *akkor* $p\_sz_i \in \{elhagy\_kilépés,\ haladás,\ jelző\_beállítás\}$.

A 10.5.1 állítás felhasználásával indukcióval megmutathatjuk a következő állítás helyességét.

> **10.5.2. állítás.** *Ha bármely elérhető rendszerállapotban* $p\_sz_i \in \{elhagy\_próba,\ belépés,\ visszaállítás\}$ *és* $p\_sz_{\bar{i}} \in \{jelző\_ellenőrzés,\ váltás\_ellenőrzés,\ elhagy\_próba,\ belépés,\ visszaállítás\}$, *akkor* $váltás \neq i$.

Azaz, ha P_i nyerte a mérkőzést, és $P_{\bar{i}}$ egy versenyző, akkor a *váltás* változó a P_i számára kedvezően van beállítva, vagyis az értéke $\bar{i}$. A 10.5.2. állítás bizonyításának indukciós lépésében az ellenőrizendő kulcsesemények a következők:

1. „sikeres” `jelző_ellenőriz`$_i$ események, azaz azok, amelyeknek hatására p_sz_i felveszi az *elhagy_próba* értéket;
2. „sikeres” `váltás_ellenőriz`$_i$ események;
3. `váltás_beállít`$_{\bar{i}}$ események, amelyeknek hatására $p\_sz_{\bar{i}}$ felveszi a *jelző_ellenőrzés* értéket;
4. `váltás_beállít`$_i$ események, amelyek elrontják a $váltás \neq i$ következményt.

Ha P_i végrehajt egy sikeres `jelző_ellenőriz`$_i$ eseményt, akkor teljesülnie kell a $jelző(\bar{i}) = 0$ feltételnek, amelyből a 10.5.1. állítás alapján következik, hogy $p\_sz_{\bar{i}} \notin \{jelző\_ellenőrzés,\ váltás\_ellenőrzés,\ elhagy\_próba,\ belépés,\ visszaállítás\}$, amiből már azonnal következik az állítás. Ha P_i végrehajt egy sikeres `váltás_ellenőriz`$_i$ eseményt, akkor teljesülnie kell a $váltás \neq i$ feltételnek,

ami elegendő az állítás teljesüléséhez. Ha $P_{\bar{i}}$ végrehajt egy `váltás_beállít`$_{\bar{i}}$ eseményt, akkor ezzel közvetlenül igazra állítja a *váltás* $\neq i$ feltételt, ami elegendő az állítás teljesüléséhez. Végül, ha P_i végrehajt egy `váltás_beállít`$_i$ eseményt, akkor ennek hatására $p\_sz_i = jelző\_ellenőrzés$ lesz, amiből már azonnal következik az állítás.

Ezzel beláttuk a 10.5.2. állítást. Innen már a kölcsönös kizárás feltételének teljesülése egyszerűen adódik. Tegyük fel, hogy valamely elérhető állapotban mind P_i, mind $P_{\bar{i}}$ a *Be* szakaszban van. Ekkor a 10.5.2. állítást kétszer alkalmazva – P_i-re és $P_{\bar{i}}$-re – azt kapjuk, hogy *váltás* $\neq i$ és *váltás* $\neq \bar{i}$ is teljesül, ami ellentmondás. □

10.11. lemma. . *A* Peterson2Foly *algoritmus garantálja a haladást.*

Bizonyítás. Indirekt tegyük fel, hogy α egy alacsony szinten pártatlan végrehajtási sorozat, amely elér egy olyan ponthoz, ahol legalább a folyamatok egyike, nevezzük P_i-nek, a *Pr* szakaszban van és egyetlen folyamat sincs a *Be* szakaszban, és tegyük fel, hogy ez után a pont után egyetlen folyamat sem lép be a *Be* szakaszba. Vizsgáljunk meg két esetet. Először, ha $P_{\bar{i}}$ a *Pr* szakaszban van valahol az adott pont után α-ban, akkor mindkét folyamatnak véglegesen benn kell ragadnia az `ellenőriz` ciklusában, hiszen egyik sem léphet be a *Be* szakaszba. De ez nem történhet meg, mivel a *váltás* változónak biztosan fel kell vennie valamilyen értéket, így a folyamatok egyike előnyt fog élvezni a másikkal szemben.

Másrészről tegyük fel, hogy $P_{\bar{i}}$ az adott pont után sosem kerül a *Pr* szakaszba α-ban. Ebben az esetben megmutathatjuk, hogy valamely későbbi pontban a *jelző(ī)* értéke nullává válik, és a továbbiakban az is marad, ami ellentmond annak a feltevésünknek, hogy P_i bennragad az `ellenőriz` ciklusában. □

10.12. lemma. . *A* Peterson2Foly *algoritmus kizárásmentes.*

Bizonyítás. A bizonyítás a *kilépés* szakaszra triviális; vizsgáljuk meg a *próba* szakaszt. Megmutatjuk azt az erősebb feltételt, hogy a megkerülések számának felső korlátja 2, és felhasználjuk a 10.8. tételt.

Indirekt tegyük fel ennek az ellenkezőjét, azaz, hogy az α végrehajtási sorozat valamely pontján a P_i a *Pr*-ben van, miután végrehajtotta a `jelző_beállít`$_i$ átmenetet, és ezek után míg P_i a *Pr*-ben marad, $P_{\bar{i}}$ háromszor belép a *Be* szakaszba. Figyeljük meg, hogy mind a második, mind a harmadik alkalommal, $P_{\bar{i}}$-nek először be kell állítania a *váltás* értékét $\bar{i}$-re, és azután azt kell észlelnie, hogy *váltás* $= i$, hiszen azt nem észlelheti, hogy *jelző(i)* $= 0$, mivel a *jelző(i)* változó értéke 1 marad. Ez azt jelenti, hogy az adott pont után legalább kétszer megjelenik a `váltás_beállít`$_i$ esemény α-ban, hiszen csak P_i tudja a *váltás* változó értékét beállítani i-re. Azonban P_i egy *próba* szakasz alatt csak egyszer hajthatja végre a `váltás_beállít`$_i$ eseményt, ami ellentmondás. □

Így megkapjuk a 10.13. tételt.

10.13. tétel. . *A* Peterson2Foly *algoritmus megoldja a kölcsönös kizárási feladatot és garantálja a kizárásmentességi feltételt.*

Bonyolultságelemzés.. Mint a DIJKSTRAKK algoritmus elemzésénél, most is legyen l és c egy felső korlát a folyamat futási idejére, illetve a *belépési* szakasz idejére. Annak pontos megértéséhez, hogy mit is jelentenek ezek a korlátok, szükség lehet a 10.3.4. szakasz kezdeti részének áttanulmányozására.

10.14. tétel. . *A* PETERSON2FOLY *algoritmusban egy adott P_i folyamat Pr-be való belépésétől a Be szakaszba való belépéséig szükséges idő legfeljebb $c + \mathcal{O}(l)$.*

Bizonyításvázlat. Indirekt tegyük fel, hogy a korlát nem jó, és vizsgáljunk meg egy végrehajtási sorozatot, amelynek valamely pontján P_i belép Pr-be, de ezután legalább $c + kl$ ideig nem lép be a Be szakaszba, ahol k egy tetszőlegesen nagy konstans. A k konstanst jóval nagyobbra választjuk, mint a következő elemzésben a nagy ordó kifejezésben szereplő konstanst.

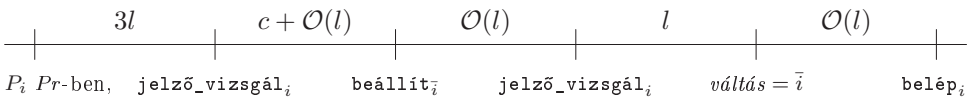
Először $3l$ időn belül a P_i folyamat végrehajt egy $\texttt{jelző\_ellenőriz}_i$ átmenetet. Ez egy esetelemzéssel belátható, amely azon alapszik, hogy P_i milyen állapotokban lehet a próba szakaszán belül. Vegyük észre, hogy P_i-nek nem lehetett egyetlen sikeres `ellenőriz` eseménye sem ez idő alatt, hiszen akkor $\mathcal{O}(l)$ idő alatt belépett volna a Be szakaszba, mi pedig feltettük, hogy ez nem történik meg ilyen gyorsan. Így amikor P_i végrehajtja ezt az $\texttt{jelző\_ellenőriz}_i$ eseményt, azt kell találnia, hogy $jelző(\bar{i}) = 1$, mert ellenkező esetben P_i belépne a Be szakaszba $\mathcal{O}(l)$ idő alatt. Így a 10.5.1. állítás alapján abban a pontban teljesülni kell a $p\_sz_{\bar{i}} \in \{váltás\_beállítás, jelző\_ellenőrzés, váltás\_ellenőrzés, elhagy\_próba, belépés, visszaállítás\}$ feltételnek.

Ekkor azt állítjuk, hogy vagy egy $\texttt{belép}_i$ esemény történik további $\mathcal{O}(l)$ idő alatt, vagy egy $\texttt{visszaállít}_{\bar{i}}$ esemény további $c + \mathcal{O}(l)$ idő alatt. Ez ugyancsak egy esetelemzéssel bizonyítható, amely a *váltás* változó értékén, illetve azon alapszik, hogy hol tartanak a folyamatok a kódjukban; a kulcspont az, hogy ha a *váltás* változó beállt valamilyen értékre, akkor ezzel előnyt fog biztosítani valamelyik folyamat részére. Az első eset azonban megint azt jelentené, hogy P_i hamar belépne a Be szakaszba, így a második esetnek kell fennállnia, vagyis egy $\texttt{visszaállít}_{\bar{i}}$ esemény történik további $c + \mathcal{O}(l)$ idő alatt.

Most további $\mathcal{O}(l)$ időn belül P_i újra végrehajt egy $\texttt{jelző\_ellenőriz}_i$ eseményt. Mint az előbb, most újra azt kell tapasztalnia, hogy $jelző(\bar{i}) = 1$. Ez azt jelenti, hogy $P_{\bar{i}}$ a $\texttt{visszaállít}_{\bar{i}}$ után újra belépett Pr-be. Ekkor a *váltás* változó vagy már felvette a $\bar{i}$ értéket, vagy fel fogja venni további l időn belül. Így újabb $\mathcal{O}(l)$ időn belül a P_i folyamat azt fogja találni, hogy a feltételek kedvezőek számára ahhoz, hogy belépjen a Be szakaszba. Ez ellentmond annak a feltevésnek, hogy P_i ennyi idő alatt nem lép be a Be szakaszba. A 10.8. ábra bemutatja a bizonyításban szereplő események sorrendjét és a közöttük levő időkorlátokat. □

10.5.2.. Egy n folyamatra adott algoritmus

Ha n folyamatunk van, akkor ismétlődően felhasználhatjuk a PETERSON2FOLY algoritmus ötletét egy $n-1$ mérkőzésből álló sorozatban az $1, 2, \ldots, n-1$ szinten. Az algoritmus minden sikeres mérkőzésnél biztosítja, hogy lesz legalább egy *vesztes*. Így mind az n folyamat részt vehet az első szintű mérkőzésen, de legfeljebb



10.8.. ábra. A 10.14. tétel bizonyításában szereplő események sorrendje és a közöttük levő időkorlátok.

$n-1$ folyamat győzhet. Általánosan legfeljebb $n-k$ folyamat győzhet a k-adik szinten. Így legfeljebb egy folyamat győzhet az $(n-1)$-edik szinten, ami kiadja a kölcsönös kizárás feltételét.

Az alábbiakban megadjuk a kódot. Itt visszatérünk ahhoz a hasznos szokáshoz, hogy a folyamatokat $1,\ldots,n$-nek nevezzük. Az algoritmus neve PETERSONnFOLY.

10.5. algoritmus. PETERSONnFOLY

Közös változók:
Minden k-ra, $k \in \{1,\ldots,n-1\}$:
váltás(k) $\in \{1,\ldots,n\}$, kezdőértéke tetszőleges,
írható és olvasható minden folyamat számára
Minden i-re, $1 \le i \le n$:
jelző$(i) \in \{1,\ldots,n-1\}$, kezdőértéke 0,
írható a P_i folyamat számára és olvasható az összes P_j, $j \neq i$ folyamat számára

P_i **folyamat:**
** Haladás szakasz **

próbál$_i$
for $k = 1$ **to** n **do**
jelző$(i) := k$
váltás$(k) := i$
waitfor $[\forall j \neq i : \mathit{jelző}(j) < k]$ **or** $[\mathit{váltás}(k) \neq i]$
belép$_i$

** Belépési szakasz **

kilép$_i$
jelző$(i) := 0$
halad$_i$

A P_i folyamat minden $1 \le k \le n-1$ szinten egy mérkőzésben szerepel. Most minden k szintnek megvan a saját *váltás* változója, *váltás*(k). Minden k szinten a P_i folyamat hasonlóan viselkedik, mint egy folyamat a PETERSON2FOLY algoritmusban: végrehajtja a *váltás*$(k) := i$ értékadást, majd vár, hogy észlelje,

hogy vagy az összes többi folyamat *jelző* változója szigorúan kisebb, mint k, vagy pedig *váltás*$(k) \neq i$. Azaz vagy a többi folyamat egyike se vesz részt jelenleg a k-adik szintű mérkőzésben, vagy pedig a *váltás*(k) változó értékét átállította valamelyik másik folyamat azóta, hogy P_i legutóbb beállította azt.

Mint korábban, most is van néhány kétértelműség a kódban, amit fel kell oldanunk. Először is a waitfor kifejezésbeli feltételek egyike tartalmazza az összes többi folyamat *jelző* változóját. A mi modellünkben ezeket a változókat nem lehet egyszerre ellenőrizni. Ehelyett mi egyszerre csak egy változót akarunk ellenőrizni, és a feltételt kielégítettnek tekintjük, ha ezek során az ellenőrzések során minden értékre azt találjuk, hogy kisebb, mint k. Másodszor adnunk kell valamilyen feltételeket arra, hogy a P_i folyamat milyen sorrendben ellenőrzi a *jelző* változókat és a *váltás*(k) változót a waitfor kifejezésben. Az egyszerűség kedvéért feltesszük, hogy P_i körkörösen végigmegy az ellenőrzéseken, minden menetben először tetszőleges sorrendben ellenőrizve az összes *jelző* változót, majd ezek után a *váltás*(k) változót.

A részleteket lásd lent. A kód nagyon hasonló a PETERSON2FOLY algoritmuséhoz. Figyeljük meg a helyi *szint* változó használatát, amelyben azt tartjuk nyilván, hogy a folyamat melyik mérkőzéssel foglalkozik (vagy melyik mérkőzéssel kész foglalkozni), és az S használatát, amelyben nyilvántartjuk, hogy mely folyamatoknál vizsgáltuk már meg, hogy a *jelző* változójuk értéke kisebb k-nál.

10.6. algoritmus. PETERSONnFOLY (átírva)

Közös változók:

Minden k-ra, $k \in \{1, \ldots, n-1\}$:
  váltás$(k) \in \{1, \ldots, n\}$, kezdőértéke tetszőleges
Minden i-re, $1 \leq i \leq n$:
  jelző$(i) \in \{1, \ldots, n-1\}$, kezdőértéke 0

A P_i folyamat műveletei:

Bemeneti:
  próbál$_i$
  kilép$_i$
Kimeneti:
  belép$_i$
  halad$_i$

Belső:
  jelző_beállít$_i$
  váltás_beállít$_i$
  jelző_ellenőriz$(j)_i$, $1 \leq j \leq n,\ j \neq i$
  váltás_ellenőriz$_i$
  visszaállít$_i$

A P_i folyamat állapotai:

p_sz $\in$ {*haladás*,*jelző_beállítás*,*váltás_beállítás*,
jelző_ellenőrzés,*váltás_ellenőrzés*,*elhagy_próba*,*belépés*,
visszaállítás,*elhagy_kilépés*}, kezdőállapot a *haladás*
szint $\in \{1, \ldots, n-1\}$, kezdetben 1
S, folyamat indexek halmaza, kezdetben üres halmaz

A P_i folyamat átmenetei:

`próbál`$_i$
  Hatás:
    $p\_sz := jelző\_beállítás$

`jelző_beállít`$_i$
  Előfeltétel:
    $p\_sz = jelző\_beállítás$
  Hatás:
    $jelző(i) := szint$
    $p\_sz := váltás\_beállítás$

`váltás_beállít`$_i$
  Előfeltétel:
    $p\_sz = váltás\_beállítás$
  Hatás:
    $váltás(szint) := i$
    $S := \{i\}$
    $p\_sz := jelző\_ellenőrzés$

`jelző_ellenőriz`$(j)_i$
  Előfeltétel:
    $p\_sz = jelző\_ellenőrzés$
    $j \notin S$
  Hatás:
    if $jelző(j) < szint$ **then**
      $S := S \cup \{j\}$
      if $|S| = n$ **then**
        $S := \emptyset$
        if $szint < n-1$ **then**
          $szint := szint + 1$
          $p\_sz := jelző\_beállítás$
        else
          $p\_sz := elhagy\_próba$
    else
      $S := \emptyset$
      $p\_sz := váltás\_ellenőrzés$

`váltás_ellenőriz`$_i$
  Előfeltétel:
    $p\_sz = váltás\_ellenőrzés$
  Hatás:
    if $váltás(szint) \neq i$ **then**
      if $szint < n-1$ **then**
        $szint := szint + 1$
        $p\_sz := jelző\_beállítás$
      else
        $p\_sz := elhagy\_próba$
    else
      $S := \{i\}$
      $p\_sz := jelző\_ellenőrzés$

`belép`$_i$
  Előfeltétel:
    $p\_sz = elhagy\_próba$
  Hatás:
    $p\_sz := belépés$

`kilép`$_i$
  Hatás:
    $p\_sz := visszaállítás$

`visszaállít`$_i$
  Előfeltétel:
    $p\_sz = visszaállítás$
  Hatás:
    $jelző(i) := 0$
    $szint := 1$
    $p\_sz := elhagy\_kilépés$

`halad`$_i$
  Előfeltétel:
    $p\_sz = elhagy\_kilépés$
  Hatás:
    $p\_sz := haladás$

Bebizonyítjuk, hogy a PetersonnFoly algoritmus helyes. A jólformáltsági feltétel teljesülése világos. A kölcsönös kizárás bizonyításánál pedig az alapgondolat az, hogy a k-adik szintű mérkőzésen csak $n-k$ győztes lehet.

A PetersonnFoly algoritmus bármely rendszerállapotában akkor mondjuk, hogy a P_i folyamat *győztes* a k-adik szinten, ha $szint_i > k$, vagy pedig $szint_i = k$ és $p\_sz_i \in \{elhagy\_próba, belépés, visszaállítás\}$. (Ez utóbbi feltétel csak $k = n-1$ esetében állhat fenn.) Azt pedig, hogy a P_i folyamat *versenyző* a k-adik szinten, akkor mondjuk, ha P_i győztes a k-adik szinten, vagy $szint_i = k$ és $p\_sz_i \in$

{*jelző_ ellenőrzés,váltás_ ellenőrzés*}.

10.15. lemma. . *A* PETERSON*n*FOLY *algoritmus kielégíti a kölcsönös kizárás feltételét.*

Bizonyítás. A lemma bizonyításához bebizonyítjuk a következő állítást, amely hasonló a PETERSON2FOLY algoritmusra bizonyított 10.5.2. állításhoz. Egy fontos különbség, hogy most az állításnak a jelző ellenőrzések közötti részekre is vonatkoznia kell.

> **10.5.3. állítás.** *A* PETERSON*n*FOLY *algoritmus bármely elérhető rendszerállapotában igazak a következők.*
>
> 1. *Ha a P_i folyamat versenyző a k-adik szinten, p_sz_i = jelző_ ellenőrzés és minden S_i beli ($j \neq i$) folyamat versenyző a k-adik szinten, akkor váltás$(k) \neq i$.*
>
> 2. *Ha a P_i folyamat győztes a k-adik szinten, és ha bármelyik másik folyamat versenyző a k-adik szinten, akkor váltás$(k) \neq i$.*

Az indukciós bizonyítást meghagyjuk gyakorlatnak (lásd 10-9. gyakorlat). A 10.5.3. állítás felhasználásával bebizonyítjuk a következőket.

> **10.5.4. állítás.** *A* PETERSON*n*FOLY *algoritmus bármely elérhető rendszerállapotában, ha van legalább egy versenyző a k-adik szinten, akkor váltás(k) értéke az egyik olyan folyamat indexe, amelyik versenyző a k-adik szinten.*

Az indukciós bizonyítást újra meghagyjuk gyakorlatnak (lásd 10-10. gyakorlat). Végül megmutatjuk a következő állítást, amelyből közvetlenül következik a kölcsönös kizárás feltétele.

> **10.5.5. állítás.** *A* PETERSON*n*FOLY *algoritmus bármely elérhető rendszerállapotában, és minden k-ra, $1 \leq k \leq n-1$, legfeljebb $n-k$ győztes van a k-adik szinten.*

A 10.5.5. állítás bizonyítása szintén indukcióval történik, de nem a végrehajtási sorozat hossza szerinti, hanem k értéke szerinti indukcióval.

Alapeset: $k = 1$. Ha az állítás hamis $k = 1$-re, az azt jelenti, hogy mind az n folyamat győztes az első szinten. Ekkor a 10.5.3. állításból következően *váltás*(1) értéke nem lehet egyik folyamat indexe sem, ami ellentmondás.

Indukciós lépés. Tegyük fel, hogy az állítás igaz k-ra, $1 \leq k \leq n-2$, és mutassuk meg, hogy igaz $(k+1)$-re. Indirekt tegyük fel, hogy az állítás hamis $(k+1)$-re, azaz, hogy több mint $n-(k+1)$ győztes van a $(k+1)$-edik szinten; legyen W ezeknek a győzteseknek a halmaza. Minden folyamatnak, amely győztes a $(k+1)$-edik szinten, győztesnek kellett lennie a k-adik szinten is, és az indukciós

feltevésünk szerint a k-adik szinten a győztesek száma legfeljebb $n-k$. Ebből következik, hogy a W-beli folyamatok a győztesek a k-adik szinten, és $|W| = n-k \geq 2$.

A 10.5.3. állításból következik, hogy *váltás*$(k+1)$ értéke nem lehet egyetlen W-beli folyamat indexe sem. A 10.5.4. állításból pedig következik, hogy *váltás*$(k+1)$ értéke az egyik olyan folyamat indexe, amely versenyző a $(k+1)$-edik szinten. De minden olyan folyamat, amely versenyző a $(k+1)$-edik szinten, győztes a k-adik szinten, azaz benne van W-ben, ami ellentmondás. □

A haladás bizonyításához elegendő a kizárásmentesség bizonyítása (lásd 10-6. gyakorlat). A 10.9 tétel alapján pedig ha az algoritmusra tudunk adni valamilyen időkorlátot, akkor abból már következik a kizárásmentesség. A *belépési* szakaszra egyszerűen lehet időkorlátot találni; a következő tétel pedig ad egy időkorlátot a *próba* szakaszra. Vigyázat: nem állítjuk, hogy ez a korlát szigorú − a szigorításával való próbálkozást meghagyjuk gyakorlatnak (lásd 10-11. gyakorlat) − de a kizárásmentesség bizonyításához bármilyen korlát elegendő.

10.16. tétel. . *A* PetersonnFoly *algoritmusban egy adott P_i folyamat Pr-be való belépésétől a Be szakaszba való belépéséig szükséges idő legfeljebb $2^{n-1}c + \mathcal{O}(2^n nl)$.*

Bizonyítás. Az időkorlátot rekurzióval bizonyítjuk. Legyen $T(0)$ egy folyamat Pr-be való belépésétől a Be szakaszba való belépéséig szükséges maximális idő. Minden k-ra, $1 \leq k \leq n-1$, legyen $T(k)$ egy folyamat k-adik szinten való győztessé válásától a Be szakaszba való belépéséig szükséges maximális idő. A $T(0)$-ra szeretnénk felső korlátot adni.

A kódból tudjuk, hogy $T(n-1) \leq l$, mivel csak egy lépés szükséges ahhoz, hogy egy folyamat belépjen a Be szakaszba, miután megnyerte a végső mérkőzést. Ahhoz, hogy kapjunk egy felső korlátot $T(0)$-ra, kifejezzük $T(k)$-t rekurzívan $T(k+1)$-ből, ahol $0 \leq k \leq n-2$.

Tegyük fel, hogy a P_i folyamat éppen most lett győztes a k-adik szinten, ha $k \geq 1$, vagy éppen most lépett be Pr-be, ha $k = 0$. Ekkor $2l$ időn belül P_i végrehajt egy `váltás_beállít`$_i$ eseményt, miáltal beállítja *váltás*$(k+1)$ értékét i-re. Jelöljük π-vel ezt a `váltás_beállít`$_i$ eseményt. Két esetet vizsgálunk meg.

Egyrészt, ha a *váltás*$(k+1)$ változó a π esemény után $T(k+1)+c+(2n+2)l$ időn belül valamilyen i-től eltérő értéket kap, akkor további nl időn belül P_i győztes lesz a $(k+1)$-edik szinten. Ekkor további $T(k+1)$ időn belül P_i belép a Be szakaszba. Ebben az esetben a π-től a P_i folyamat Be szakaszba való belépéséig eltelt teljes idő legfeljebb $2T(k+1)+c+(3n+2)l$.

Másrészről, tegyük fel, hogy a *váltás*$(k+1)$ változó nem kap valamilyen i-től eltérő értéket a π esemény után $T(k+1)+c+(2n+2)l$ időn belül. Így egyetlen folyamat sem állíthatja a *jelző* változója értékét $k+1$-re a π esemény után $T(k+1)+c+(2n+1)l$ időn belül. Legyen I azon $j \neq i$ folyamatok halmaza, amelyekre *jelző*$(j) \geq k+1$, amikor π bekövetkezik. Ekkor minden I-beli folyamat győztes lesz a $(k+1)$-edik szinten a π esemény után legfeljebb nl időn belül (mivel felfedezi, hogy *váltás*$(k+1)$ értéke nem egyezik meg az indexével), azután további $T(k+1)$ időn belül belép a Be szakaszba, majd újabb c időn belül elhagyja azt és

további l időn belül végrehajt egy *visszaállít* eseményt. Azaz π után $nl + T(k+1) + c + l = T(k+1) + c + (n+1)l$ időn belül minden I-beli folyamat nullára állítja a *jelző* változója értékét.

Így minden $j \neq i$ folyamat, amelyre *jelző*$(j) \geq k+1$, amikor π végrehajtódik, π után $T(k+1) + c + (n+1)l$ időn belül nullára állítja a *jelző* változója értékét. Mint az előzőekben feltettük, ezt követően újabb nl időn belül egyetlen folyamat sem állítja a *jelző* változója értékét $k+1$-re. Ez elegendő idő a P_i folyamat számára, hogy felfedezze, hogy minden *jelző* változó értéke kisebb, mint $k+1$, és így győztes legyen a $(k+1)$-edik szinten. Azaz ebben az esetben a P_i folyamat győztes lesz a $(k+1)$-edik szinten π után $T(k+1) + c + (2n+1)l$ időn belül. Ezt követően pedig újabb $T(k+1)$ időn belül P_i belép a Be szakaszba. Ebben az esetben a π-től a P_i folyamat Be szakaszba való belépéséig eltelt teljes idő legfeljebb $2T(k+1) + c + (2n+1)l$.

Így a legrosszabb esetben a szükséges idő $2l$ plusz az előbbi két esetben számított idők maximuma, azaz összesen $2T(k+1)+c+(3n+4)l$. Tehát a következő rekurziót kell $T(0)$-ra megoldanunk:

$$\begin{aligned} T(k) &\leq 2T(k+1) + c + (3n+4)l (\text{minden } 0 \leq k \leq n-2)-\text{re} \\ T(n-1) &\leq l. \end{aligned}$$

Ennek a rekurziónak a megoldásával kapjuk a keresett időkorlátot. (Lásd a következő 10.5.3. szakaszt egy hasonló rekurzió részletesebb megoldására.) □

Így kapjuk a következő állítást.

10.17. tétel. . *A* PetersonnFoly *algoritmus megoldja a kölcsönös kizárási feladatot és kielégíti a kizárásmentesség feltételét.*

10.5.3.. Verseny algoritmus

A másik módszer, amivel a Peterson2Foly algoritmust kiterjeszthetjük több folyamatra, hogy az alap kétfolyamatos algoritmus egy változatát mint építőkockát felhasználjuk egy *versenyben*. Az egyszerűség kedvéért tegyük fel, hogy n, a folyamatok száma, egy 2-hatvány. Megint számozzuk a folyamatokat nullától kezdve $0, \ldots, (n-1)$-gyel az $1, \ldots, n$ helyett. Minden folyamat részt vesz egy $\log n$ mérkőzésből álló sorozatban, hogy megszerezze az erőforrást. A mérkőzéseket úgy kell elképzelni, mintha egy teljes, n levelű, bináris *versenyfába* lennének rendezve; az n levél balról jobbra nézve megfelel a $0, \ldots, n-1$ folyamatnak.

Szükségünk van valamilyen jelölésre, a különböző mérkőzések, a folyamatok által a mérkőzésekben játszott szerepek és a lehetséges ellenfelek elnevezéséhez, amely jelöléseket minden folyamat használhat minden mérkőzésben. Minden $0 \leq i \leq n-1$-re és $1 \leq k \leq \log n$-re definiáljuk a következő jelöléseket.

- *mérkőzés*(i,k), P_i folyamat *k-adik szintű mérkőzése*, egy karaktersorozat, amely i bináris reprezentációjának felső $\log(n)-k$ bitjéből áll. A versenyfát nézve *mérkőzés*(i,k) használható annak a belső csúcsnak az elnevezésére, amely P_i levelének a k-adik szintű őse. A gyakorlatban a gyökércsúcs

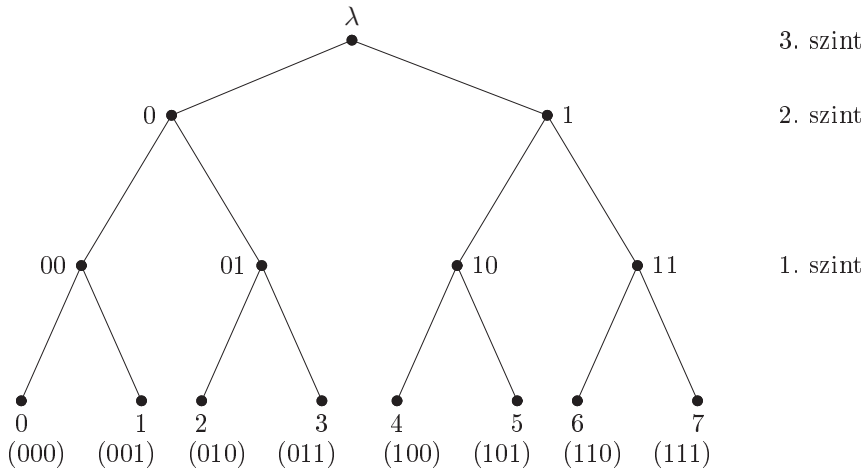
elnevezésére a λ-t, az üres karaktersorozatot használjuk.

- *szerep(i,k)* a P_i folyamat *szerepe* P_i k-adik szintű mérkőzésében, a $\log(n) - k + 1$. bit i bináris reprezentációjában. A versenyfát nézve $szerep(i,k)$ jelöli, hogy P_i levele a $mérkőzés(i,k)$ mérkőzéshez tartozó csúcs bal vagy jobb gyerekének utódja-e.

- $ellenfelek(i,k)$ a P_i folyamat *ellenfelei* P_i k-adik szintű mérkőzésében, azon folyamat indexek halmaza, amelyek felső $\log(n)-k$ bitje megegyezik i ilyen bitjeivel, míg a $(\log(n)-k+1$-edik bitjük ellentétes értékű, mint i ilyen bitje. A versenyfát nézve az $ellenfelek(i,k)$-beli folyamatok azok, amelyek levelei a $mérkőzés(i,k)$ csúcs másik gyerekének utódjai, vagyis annak a gyereknek, amelyik nem őse P_i levelének.

10.5.1. példa. Versenyfa

A 10.9 ábra bemutatja $n = 8$-ra a versenyfát. Például figyeljük meg, hogy $mérkőzés(5,2)=1$, $szerep(5,2)=0$ és $ellenfelek(5,2) = \{6,7\}$.



10.9.. ábra. A mérkőzések nevei a VERSENY algoritmusban.

Az algoritmust VERSENY *algoritmusnak* nevezzük.

10.7. algoritmus. VERSENY

Közös változók:
Minden x bináris karaktersorozatra, amelynek hossza legfeljebb $\log(n)-1$:
váltás$(x) \in \{0,1\}$, kezdőértéke tetszőleges, írható és olvasható pontosan azon P_i folyamatok számára, amelyekre x előtagja i bináris reprezentációjának.
Minden i-re, $0 \leq i \leq n-1$:
jelző$(i) \in \{0,\ldots,\log n\}$, kezdőértéke 0, írható a P_i folyamat számára és olvasható az összes $P_j,\ j \neq i$ folyamat számára

P_i **folyamat:**
** Haladás szakasz **

próbál$_i$
for $k = 1$ **to** $\log n$ **do**
jelző$(i) := k$
váltás$(mérkőzés(i,k)) :=$*szerep*(i,k)
waitfor $[\forall j \in ellenfelek(i,k) : jelző(j) < k]$ **or** $[váltás(mérkőzés(i,k)) \neq szerep(i,k)]$
belép$_i$

** Belépési szakasz **

kilép$_i$
jelző$(i) := 0$
halad$_i$

Ez a kód nagyon hasonlít a PETERSONnFOLY algoritmuséra. A fő eltérés az, hogy minden mérkőzésben a folyamat csak azon folyamatok jelzőit ellenőrzi, amelyek ellenfelei az adott mérkőzésben. Mint a PETERSONnFOLY algoritmusban, úgy itt is feltesszük, hogy egy folyamat valamilyen tetszőleges sorrendben ellenőrzi az ellenfeleit, egy időben csak egyet. A vizsgálatot valamilyen szisztematikus módszerrel kell váltogatni, például végezhetjük egy olyan körben, amelyben először az összes *jelzőt* teszteljük és utána a *váltás*-ot. Mi csak röviden vázoljuk a VERSENY algoritmus helyességének bizonyítását, mivel a módszer annyira hasonlít a PETERSON2FOLY és a PETERSONnFOLY algoritmusoknál vizsgáltakra.

Először az algoritmust át kell írni előfeltétel/hatás stílusura, pontosan meghatározva a program számlálókat és azokat a változókat, amelyekben azon folyamatok halmazait tároljuk, amelyek jelzői már ellenőrizve lettek. (Lásd 10-14.(a) gyakorlat.) A jelöléseket, mint a „győztes a k-adik szinten" és a „versenyző a k-adik szinten" definiálni kell a VERSENY algoritmus esetében is, hasonló módszerrel, mint a PETERSONnFOLY algoritmusnál tettük. (Lásd 10-14.(b) gyakorlat.)

10.18. lemma. . *A* VERSENY *algoritmus kielégíti a kölcsönös kizárás feltételét.*

Bizonyításvázlat. A bizonyítás hasonló elv alapján történik, mint a PETERSON2FOLY és a PETERSONnFOLY algoritmusok invariáns állítást használó bizo-

nyításai.

10.5.6. állítás. *A* VERSENY *algoritmus bármely elérhető rendszerállapotában és minden k-ra, $1 \leq k \leq \log n$ bármely részfából, amelynek a gyökere a k-adik szinten van, legfeljebb egy folyamat lehet győztes a k-adik szinten.*

Ez közvetlenül következik egy invariánsból, amely hasonló a 10.5.3 állítás második részéhez.

10.5.7. állítás. *Ha a P_i folyamat győztes a k-adik szinten, és ha bármelyik olyan folyamat versenyző a k-adik szinten, amely P_i-nek ellenfele a k-adik szinten, akkor váltás(mérkőzés(i,k)) $\neq$ szerep(i,k).*

Mint a 10.5.3 állítás második felét, úgy a 10.5.7. állítást sem bizonyíthatjuk közvetlenül indukcióval. Mint a korábbiakban, most is erősítenünk kell az állítást, hogy arról is tartalmazzon információt, hogy mi történik a **waitfor** ciklus belsejében, miután a folyamat felfedezte, hogy valahány ellenfele *jelző* változójának értéke szigorúan kisebb, mint k. Az erősítést és az indukciós bizonyítást meghagyjuk gyakorlatnak az olvasó számára (lásd 10-14.(c) gyakorlat).

□

A haladás és a kizárásmentesség bizonyításához bizonyítunk egy időkorlátot.

10.19. tétel. . *A* VERSENY *algoritmusban egy adott folyamat Pr-be való belépésétől a Be szakaszba való belépéséig szükséges idő legfeljebb* $(n-1)c + \mathcal{O}(n^2 l)$.

Bizonyítás. A bizonyítás hasonlít a 10.16. tétel bizonyításához. Legyen $T(0)$ egy folyamat Pr-be való belépésétől a Be szakaszba való belépéséig szükséges maximális idő. Minden k-ra, $1 \leq k \leq n-1$, legyen $T(k)$ egy folyamat k szinten való győztessé válásától a Be szakaszba való belépéséig szükséges maximális idő. A $T(0)$-ra szeretnénk felső korlátot adni. A kódból tudjuk, hogy $T(\log n) \leq l$, mivel csak egy lépés szükséges ahhoz, hogy egy folyamat belépjen a Be szakaszba, miután megnyerte a végső mérkőzést. A $T(k)$-ra felső korlátot adunk a $T(k+1)$ függvényében, ahol $0 \leq k \leq \log(n) - 1$.

Tegyük fel, hogy a P_i folyamat éppen most lett győztes a k-adik szinten, ha $k \geq 1$, vagy éppen most lépett be Pr-be, ha $k = 0$. Jelöljük x-el *mérkőzés*$(i, k+1)$-et. Ekkor $2l$ időn belül a P_i folyamat beállítja *váltás*(x) értékét *szerep*$(i, k+1)$-re. Jelöljük π-vel ezt az eseményt; két esetet vizsgálunk meg.

Egyrészt, ha a *váltás*(x) változó értéke megváltozik a π esemény után $T(k+1)+c+(2^{k+1}+4)l$ időn belül, akkor további $(2^k+1)l$ időn belül P_i győztes lesz a $(k+1)$-edik szinten. Ekkor további $T(k+1)$ időn belül P_i belép a Be szakaszba. Ebben az esetben a π-től a P_i folyamat Be szakaszba való belépéséig eltelt teljes idő legfeljebb $2T(k+1) + c + (2^{k+1} + 2^k + 5)l$.

Másrészről, tegyük fel, hogy a a *váltás*(x) változó értéke nem változik meg a π esemény után $T(k+1)+c+(2^{k+1}+4)l$ időn belül. Így egyetlen olyan folyamat,

amelyik ellenfele P_i-nek a $(k+1)$-edik, szinten sem állíthatja a *jelző* változója értékét $k+1$-re a π esemény után $T(k+1)+c+(2^{k+1}+3)l$ időn belül. Ha P_j egy olyan folyamat, amelyik ellenfele P_i-nek a $(k+1)$-edik szinten és teljesül rá, hogy *jelző(j)* $\geq k+1$, amikor π bekövetkezik, akkor π után $(2^k+1)l+T(k+1)+c+l = T(k+1)+c+(2^k+2)l$ időn belül a P_j folyamat beállítja a *jelző* változója értékét 0-ra.

Így π után $T(k+1)+c+(2^k+2)l$ időn belül minden olyan folyamat, amely ellenfele P_i-nek a $(k+1)$-edik szinten és amelyre *jelző(j)* $\geq k+1$, amikor π bekövetkezik beállítja a *jelző* változója értékét 0-ra. Mint az előbbiekben feltettük, ezután további $(2^k+1)l$ időn belül egyetlen folyamat sem állíthatja be a a *jelző* változója értékét $(k+1)$-re. Ami elegendő idő a P_i folyamat számára, hogy észrevegye, hogy minden olyan folyamatnak, amely ellenfele a $(k+1)$-edik szinten a *jelző* változója kisebb mint $k+1$, és így győztes legyen a $(k+1)$-edik szinten. Azaz ebben az esetben P_i folyamat győztes lesz a $(k+1)$-edik szinten π után $T(k+1)+c+(2^{k+1}+3)l$ időn belül. További $T(k+1)$ időn belül P_i belép a Be szakaszba. Ebben az esetben a π-től a P_i folyamat Be szakaszba való belépéséig eltelt teljes idő legfeljebb $2T(k+1)+c+(2^{k+1}+3)l$.

Így a legrosszabb esetben a szükséges idő $2l$ plusz az előbbi két esetben számított idők maximuma, azaz összesen $2T(k+1)+c+(2^{k+1}+2^k+7)l$. Tehát a következő rekurziót kell $T(0)$-ra megoldanunk:

$$\begin{aligned} T(k) &\leq 2T(k+1)+c+(2^{k+1}+2^k+7)l, \text{minden } 0 \leq k \leq \log(n-1)-\text{re} \\ T(\log n) &\leq l \end{aligned}$$

Válasszunk valamilyen a konstanst úgy, hogy $(2^{k+1}+2^k+7) \leq a \cdot 2^k$. Ekkor azt kapjuk, hogy:

$$\begin{aligned} T(0) &\leq 2T(1)+2^0c+a2^0l \\ &\leq 2^2T(2)+(2^0+2^1)c+a(2^0+2^2)l \\ &\leq 2^3T(2)+(2^0+2^1+2^2)c+a(2^0+2^2+2^4)l \\ &\vdots \\ &\leq 2^kT(k)+(2^0+2^1+\ldots+2^{k-1})c+a(2^0+2^2+\ldots+2^{2k-2})l \\ &\vdots \\ &\leq 2^{\log n}T(\log n)+(2^0+2^1+\ldots+2^{\log n-1})c \\ &\quad +a(2^0+2^2+\ldots+2^{2(\log n-1)})l \\ &\leq (n-1)c+nl+\mathcal{O}(n^2l) \\ &= (n-1)c+\mathcal{O}(n^2l). \end{aligned}$$

□

10.20. tétel. . *A* VERSENY *algoritmus megoldja a kölcsönös kizárási feladatot és teljesíti a kizárásmentesség feltételét.*

Korlát a megkerülések számára. A VERSENY algoritmus nem garantál semmilyen korlátot a megkerülések számára. Ennek szemléltetésére nézzünk egy végrehajtást, amelyben a P_0 folyamat belép a versenybe a levelénél és a közbeeső lépéseket pontosan annyi idő alatt hajtja végre, mint a feltett l felső korlát. Ezalatt a P_{n-1} folyamat belép a versenybe a levelénél, és sokkal gyorsabban halad. A P_{n-1} folyamat elérhet a tetőre és győzhet, és valójában ezt megismételheti végtelen sokszor, mielőtt a P_0 folyamat győzne az első szinten. Ennek az az oka, hogy nem tettünk fel semmilyen alsó korlátot a folyamatok lépésidejére.

Vegyük észre, hogy nincs ellentmondás az időkorlát és az között, hogy a megkerülések számára nem tudunk felső korlátot adni. Egyetlen folyamat sincs éheztetve túl sokáig, a megkerülések számára csak azért nem tudunk felső korlátot adni, mert néhány folyamat nagyon gyorsan működik.

10.6.. Kizárólagosan írható közös regiszteres algoritmus

Az eddig tárgyalt kölcsönös kizárás algoritmusok több folyamat által írható (megosztottan írható) osztott regisztereket (*váltás* változókat), valamint egy folyamat által írható (kizárólagosan írható) osztott regisztereket (*jelző* változókat) használnak. Mivel a több folyamat által írható regiszterek megvalósítása gyakran nehéz, érdemes tanulmányozni azokat az algoritmusokat, amelyek csak egy folyamat által írható közös regisztereket használnak. Ebben és a következő alfejezetben bemutatunk két ilyen algoritmust.

Az ebben az alfejezetben szereplő algoritmus megoldja a kölcsönös kizárási feladatot (mint mindig, most is beleértjük a haladási feltételt), de nem biztosít semmilyen magas szintű pártatlansági feltételt. Algoritmus valamennyi közös regisztere bináris. A 10.7. alfejezetben szereplő algoritmus kizárásmentes, de hátránya, hogy nem korlátos méretű változókat használ.

Ezt az első algoritmust – feltalálójáról, J. Burnsről – BURNSKK algoritmusnak hívjuk.

10.8. algoritmus. BURNSKK

Közös változók:
minden i-re $(1 \leq i \leq n)$:
jelző $\in \{0, 1\}$, kezdetben 0, P_i által írható és minden P_j $(j \neq i)$ által olvasható

P_i **folyamat:**

```
    ** Haladási szakasz **

    próbál_i
L:  jelző(i) := 0
    for j, 1 ≤ j ≤ i − 1 do
        if jelző(j) = 1 then goto L
    jelző(i) := 1
    for j, 1 ≤ j ≤ i − 1 do
        if jelző(j) = 1 then goto L
M:  for j, i + 1 ≤ j ≤ n do
        if jelző(j) = 1 then goto M

    belép_i

    ** Belépési szakasz **

    kilép_i
    jelző(i) := 0
    halad_i
```

- -

A BurnsKK algoritmusban a *jelző* értéke 0 és 1 lehet, ellentétben a DijkstraKK algorimussal, ahol 0, 1 és 2. Minden folyamat három **for** ciklust futtat. Az első két ciklus ellenőrzi a kisebb indexű folyamatok *jelzőit*, míg a harmadik ciklus a nagyobb indexű folyamatok esetében teszi ugyanezt. Ha a P_i folyamat mindhárom ciklusban szereplő feltételeknek megfelel, akkor belép a belépési szakaszába.

10.21. lemma. . *A* BurnsKK *algoritmus teljesíti a kölcsönös kizárást.*

Bizonyítás. A bizonyítás az első (az algoritmus működését elemző) bizonyításhoz hasonlítható, azaz, hogy DijkstraKK teljesíti a kölcsönös kizárást (lásd 10.3. lemmát). A fő különbség, hogy a *jelző* változók most az 1 értéket veszik fel, noha a DijkstraKK-ban a 2-t.

Így, ha a P_i és P_j folyamatok egyidejűleg Be-ben vannak, tegyük fel, hogy elsőként P_i állítja a *jelző* értékét 1-re. Ekkor *jelző*(i) mindaddig megtartja az 1 értéket, amíg a P_i folyamat a Be-t el nem hagyja. Miután P_j beállítja a *jelző*(j)-t 1-re, P_j-nek ellenőriznie kell, hogy *jelző*$(i) = 0$, mielőtt P_j Be-be léphet. (Ha $i < j$, akkor ez a második ciklusban történik, míg ha $i > j$, akkor a harmadikban). Ennek az ellenőrzésnek akkor kell megtörténnie, amikor *jelző*$(i) = 1$, ami ellentmondáshoz vezet. □

Jegyezzük meg, hogy az első ciklus a kódban nem szükséges a kölcsönös kizárás feltétel teljesítéséhez.

10.22. lemma. . BurnsKK *biztosítja a haladást.*

Bizonyítás. A kilépési szakasz esetében könnyű a bizonyítás. A próba szakaszhoz indirekt feltesszük, hogy α egy alacsony szintű pártatlan végrehajtási sorozat, amely elér egy ponthoz, ahol legalább egy folyamat Pr-ben van, Be-ben nincs folyamat, és e pont után nincs olyan folyamat, amely valaha is belépne Be-be. Az indoklás hasonló, mint a 10.4. lemmában: feltehetjük az általánosság megszorítása nélkül, hogy minden α-beli folyamat Pr-ben vagy Ha-ban van, és hogy nincs folyamat, amely szakaszt váltana α-ban. Legyenek a versenyzők a Pr-beli folyamatok.

A versenyzőket két halmazra osztjuk: azokra, amelyek valaha elérik az M címkét, és amelyek soha. Az első halmazt P-nek, a másodikat Q-nak hívjuk. Biztos létezik olyan α-beli pont, ameddig az összes P-beli folyamat már elérte az M címkét; jegyezzük meg, hogy ezek a folyamatok az M elérése után többé nem ugornak vissza M címke előtti kódrészre. Legyen α_1 az α egy olyan utótagja, amelyben minden P-beli folyamat az utolsó **for** ciklusban van az M címke után.

Azt állítjuk, hogy van legalább egy ilyen P-beli folyamat. Speciálisan, a versenyzők közötti legkisebb indexű folyamat nem blokkolt az M címke elérése óta.

Legyen i a P-beli folyamatok indexei közül a legnagyobb. Azt állítjuk, hogy α_1-ben bármely olyan $P_j \in Q$ folyamat *jelző*(j) értéke biztosan és tartósan 0 lesz, amelyre $j > i$. Ez azért igaz, mert akárhányszor P_j végrehajtja az első két ciklus közül az egyiket, felfedez egy kisebb indexű versenyzőt és visszatér L-hez. Ahányszor ezt teszi, *jelző*(j) értékét 0-ra állítja, és ha egyszer így tesz, akkor többé nem jut elég messze ahhoz, hogy *jelző*(j)-t 1-re állítsa. Legyen α_2 az α_1 suffixe, amelyben minden i-nél nagyobb indexű Q-beli folyamat állandóan megtartja *jelző*-jének 0 értékét.

Az α_2-ben semmi sem akadályozhatja meg P_i-t, hogy elérje Be-t: minden nagyobb indexű P_j folyamat esetében *jelző*$(j) = 0$, így P_i sikeresen befejezheti a harmadik ciklust. Így P_i belép Be-be, ami ellentmondás. □

10.23. tétel. . BURNSKK *megoldja a kölcsönös kizárási feladatot.*

10.7.. A VÁRÓTEREM algoritmus

Ebben az alfejezetben a VÁRÓTEREM algoritmust mutatjuk be a kölcsönös kizárás megoldására. Némileg egy olyan váróteremhez hasonlóan működik, ahol a várakozók belépéskor sorszámot húznak, és a kiszolgálásuk a sorszám szerint történik.

A VÁRÓTEREM algoritmus csak kizárólagosan írható/megosztottan olvasható osztott regisztereket használ. Valójában a *biztonságos regiszterként* ismert gyengébb típusú regisztert használja, amelyben az írásokkal párhuzamosan végrehajtódó olvasások során tetszőleges válaszokat adhatnak.

A VÁRÓTEREM algoritmus biztosítja a kizárásmentességet és a jó időkorlátot. Biztosítja a korlátozott megkerülést és egy ehhez kapcsolódó feltételt, a „FIFO egy várakozásmentes bejárat után"-t (meghatározását lásd később). A VÁRÓTEREM algoritmus egy kellemetlen tulajdonsága, hogy nem korlátos méretű regisztereket használ.

A kód következik. Megjegyezzük, hogy az itt megadott kód egyszerűsíthető, ha a regisztereknek csak megszokott fajtáit használjuk (és nem a regiszterek gyengébb típusait, mint például a biztonságos regisztereket). Ezt az egyszerűsítést meghagyjuk gyakorlatnak (lásd 10-27. gyakorlat).

10.9. algoritmus. VÁRÓTEREM

Közös változók:
minden i-re $(1 \leq i \leq n)$:
$\mathit{választás}(i) \in \{0,1\}$, kezdetben 0, P_i által írható és minden P_j $(j \neq i)$ által olvasható
$\mathit{sorszám}(i) \in \mathbb{N}$, kezdetben 0, P_i által írható és minden P_j $(j \neq i)$ által olvasható

P_i **átmenetei:**

** Haladási szakasz **

$\texttt{próbál}_i$
$\mathit{választás}(i) := 1$
$\mathit{sorszám}(i) := 1 + \max_{j \neq i} \mathit{sorszám}(j)$
$\mathit{választás}(i) := 0$
for $j \neq i$ **do**
  waitfor $\mathit{választás}(j) = 0$
  waitfor $\mathit{sorszám}(j) = 0$ **or** $(\mathit{sorszám}(i), i) < (\mathit{sorszám}(j), j)$
$\texttt{belép}_i$

** Belépési szakasz **

$\texttt{kilép}_i$
$\mathit{sorszám}(i) := 0$
$\texttt{halad}_i$

A VÁRÓTEREM algoritmus próba szakaszának első részét (addig a pontig, ahol a P_i folyamat végrehajtja a $\mathit{választás}(i) := 0$ értékadást), *bejáratnak* nevezzük. Amíg a P_i folyamat a bejáratban tartózkodik, választ egy olyan *sorszám*-ot, amely nagyobb minden más sorszámnál, amelyet a többi folyamatnál olvasott le. A többi folyamat *sorszám*-ai közül egyszerre csak egyet olvas, tetszőleges sorrendben, majd a saját *sorszám*-át írja át. Amíg a P_i folyamat olvas és sorszámot választ, P_i biztosítja, hogy $\mathit{választás}(i) = 1$ legyen, ez egy jel a többi folyamat számára.

Vegyük figyelembe, hogy egyszerre két folyamat is lehet a bejáratban, amely azt okozhatja, hogy ugyanazt a sorszámot választják. Az ilyen eldöntetlen helyzetek megszüntetése érdekében a folyamatok nem csak a *sorszámukat*, hanem a $(\mathit{sorszám}, \mathit{index})$ párjukat hasonlítják össze. Az összehasonlítás lexikografikusan történik, így az eldöntetlen helyzetek megszüntetésekor a kisebb indexű folyamatot támogatja.

A próba szakasz többi részében a folyamat arra vár, hogy a többi folyamat befejezze a választást és, hogy a $(sorszám, index)$ párosa a legkisebbé váljon.

A helyesség belátásához jelölje Bej a bejáratot (azaz a bejáratban lévő folyamat állapotainak halmazát), és $Pr - Bej$ a próba szakasz maradék részét. A jólformáltság könnyen látható. A kölcsönös kizárás megmutatásához egy lemmát használunk.

10.24. lemma. . *A* VÁRÓTEREM *algoritmus minden elérhető rendszerállapotára és bármely P_i és P_j folyamatra ($i \neq j$) a következő áll fenn: ha P_i Be-ben van és P_j a $(Pr - Bej) \cup Be$-ben, akkor $(sorszám(i), i) < (sorszám(j), j)$.*

A továbbiakban egy, az algoritmus működését elemző bizonyítást adunk, mivel ez könnyebben kiterjeszthető a biztonságos regiszter esetére.

Bizonyítás. Rögzítünk egy s pontot a végrehajtási sorozatban, amelyben P_i Be-ben, P_j a $(Pr - Bej) \cup Be$-ben van. (Formálisan, s egy rendszerállapot egy előfordulása). Az s pontban a $sorszám(i)$ és $sorszám(j)$ értékeit ezen változók *helyes* értékeinek nevezzük.

A P_i folyamatnak – mielőtt belépne Be-be – $választás(j) = 0$ értéket kell olvasnia az első waitfor ciklusban. Jelölje π ezt az olvasási eseményt, így π megelőzi s-t. Amikor π bekövetkezik, P_j nincs a „választási szakaszban" (azaz a bejáratnak azon szakaszában, ahol már $választás(j) = 1$). De mivel P_j a $(Pr - Bej) \cup Be$-ben van az s pontnál, P_j-nek valamely pontban keresztül kell haladnia a választási szakaszon. Két esetet kell átgondolni:

1. P_j π után lép be a választási szakaszba. Ekkor a helyes $sorszám(i)$ ki lett választva, mielőtt P_j választása elkezdődött volna, biztosítva, hogy P_j a helyes *sorszám(i)*-t látja, amikor választ. Ezért az s pontban $sorszám(j) > sorszám(i)$, ami kielégíti az állítást.
2. P_j π előtt hagyja el a választási szakaszt. Ekkor valahányszor P_i a második waitfor ciklusában olvassa P_j sorszámát, a helyes $sorszám(j)$-t kapja. De bárhogyan is lépjen P_i a Be-be, teljesülnie kell a $(sorszám(i), i) < (sorszám(j), j)$ feltételnek. Ez szintén kielégíti az állítást.

□

10.25. lemma. . *A* VÁRÓTEREM *algoritmus teljesíti a kölcsönös kizárást.*

Bizonyítás. Tegyük fel, hogy valamely elérhető állapotban két folyamat, P_i és P_j is Be-ben van. Ekkor a 10.24. lemmát kétszer alkalmazva: $(sorszám(i), i) < (sorszám(j), j)$ és $(sorszám(j), j) < (sorszám(i), i)$ is teljesül. Ez ellentmondás. □

10.26. lemma. . *A* VÁRÓTEREM *algoritmus biztosítja a haladást.*

Bizonyítás. A kilépési szakaszra könnyű belátni az állítást. A próba szakasz esetében indirekt úton bizonyítunk. Tegyük fel, hogy az algoritmus a haladást

nem biztosítja. Ebben az esetben az algoritmus végrehajtása biztosan elér egy olyan pontot, amely után minden folyamat Pr-ben vagy Ha-ben van, és nem történik új szakaszváltás. A kód szerint minden Pr-beli folyamat biztosan elhagyja a bejáratot és eléri $(Pr - Bej)$-t. Ekkor a legalacsonyabb $(sorszám, index)$ párral rendelkező folyamat elérheti Be-t. □

10.27. lemma. . *A* VÁRÓTEREM *algoritmus biztosítja a kizárásmentességet.*

Bizonyítás. Tekintsünk egy konkrét Pr-beli P_i folyamatot, és tegyük fel, hogy soha nem éri el Be-t. A P_i folyamat biztosan elhagyja a bejáratot, és eléri $(Pr–Bej)$-t. Ezután bármely új folyamat, amely belép a bejáratba, látja P_i legutolsó *sorszám*-át és egy annál nagyobb sorszámot választ. Így, mivel P_i nem éri el Be-t, egyik új folyamat sem éri el Be-t, mivel a második várakozási loop ciklusban a $sorszám(i)$ tesztelésénél mindegyikük blokkolt.

A 10.26. lemma ismételt alkalmazásával következik, hogy kell lennie olyan folyamatos haladásnak, amely végtelenül sok `belép` eseményt tartalmaz. Ez ellentmond annak, hogy minden, a próba szakaszba belépő új folyamat blokkolt. □

10.28. tétel. . *A* VÁRÓTEREM *algoritmus megoldja a kölcsönös kizárási feladatot, és kizárásmentes.*

Bonyolultságelemzés.. Egy P_i folyamat próba szakaszba lépésétől a belépési szakaszba lépéséig eltelt idő felső korlátja $(n-1)c + \mathcal{O}(n^2 l)$. Ezt nem könnyű látni; csak egy rövid vázlatot adunk, a részletezést gyakorlatnak hagyjuk (lásd 10-28. gyakorlat).

Először: a P_i folyamatnak $\mathcal{O}(nl)$ idő szükséges, hogy elhagyja a bejáratot; felső korlátot kell adnunk arra az I időintervallumra, amennyit P_i a $(Pr–Bej)$-ben tölt. Legyen P azon más folyamatok halmaza, amelyek már Pr-ben vannak, amikor P_i a $(Pr–Bej)$-be lép. Ekkor csak a P-beli folyamatok tudnak Be-be lépni azelőtt, hogy P_i belépne, és ezek mindegyike is csak egyszer. Ebből az következik, hogy a teljes idő az I intervallumon belül – ami alatt valamely folyamat a Be-ben van – legfeljebb $(n-1)c$, és, hogy a teljes idő az I intervallum belül, ami alatt valamely folyamat a bejáratban van, legfeljebb $\mathcal{O}(n^2 l)$ lehet.

Hátra van még a *fennmaradó idő* korlátozása az I intervallumon belül, vagyis az a teljes idő az I-n belül, amikor sem a Be-ben, sem a bejáratban nincs folyamat. Úgy korlátozzuk a fennmaradó időt, hogy figyelembe vesszük a $P \cup \{P_i\}$-beli folyamatok előrehaladását. Megjegyezzük, hogy a fennmaradó idő alatt ezen folyamatok közül soha egy sem volt blokkolt az első waitfor ciklusában, mivel az összes *választás* változó 0. Ezenkívül valamelyik $P \cup \{P_i\}$-beli folyamat a második waitfor ciklusának egyetlen lépésében sem lesz blokkolt, és így az $\mathcal{O}(nl)$ fennmaradó idő alatt be fog lépni Be-be. Miután befejeződött ez a folyamat, valamely más $P \cup \{P_i\}$-beli folyamat nem lesz már blokkolt, és így az is további $\mathcal{O}(nl)$ fennmaradó időn belül be fog lépni Be-be, és így tovább. Ez addig folytatódik, amíg P_i belép Be-be, a teljes hátralévő idő pedig $\mathcal{O}(n^2 l)$.

FIFO egy várakozásmentes bejárat után.. A VÁRÓTEREM algoritmus egy magas szintű pártatlansági feltételt biztosít, amely egy kicsit erősebb, mint a kizárásmentesség. Mégpedig, ha a P_i folyamat elhagyja a bejáratot, mielőtt P_j a Pr-be lépne, akkor P_j nem tud Be-be lépni P_i belépése előtt. Jegyezzük meg, hogy az algoritmus valójában nem a Pr-be való belépési idő szerinti FIFO, még az első, helyileg ellenőrzött Pr-beli lépés szerinti FIFO sem. Például, a P_1 folyamat beléphet és végrehajthatja a *választás*(1) := 1 műveletet, majd beléphet a P_2 folyamat is, választ egy sorszámot és elhagyja a bejáratot; P_1 ekkor választhatja a sorszámát. Ebben az esetben a P_1 folyamat egy P_2-nél nagyobb sorszámot választana, megengedve ezzel P_2-nek, hogy őt megelőzve lépjen Be-be.

Nem lenne célszerű egyszerűen azt állítani, hogy egy algoritmus „bejárat utáni FIFO", mivel nincs pontos meghatározás, mikor ér véget egy bejárat. (Ha a bejárat éppen Be-be való belépés előtt fejeződik be, akkor ez az állítás triviálisan teljesül.) A VÁRÓTEREM algoritmusban szereplő bejárat azonban egy érdekes tulajdonsággal rendelkezik: a bejárat *várakozásmentes*, ami az jelenti, hogy a folyamat biztosan elhagyja, amennyiben a folyamat további lépéseket tesz, függetlenül attól, hogy bármely más folyamat is ezt teszi.

Így a „FIFO egy várakozásmentes bejárat után" tulajdonságot, amely egy nem-triviális és érdekes magasszintű pártatlansági feltétel, kielégíti a VÁRÓTEREM algoritmus.

10.8.. Alsó korlát a regiszterek számára

Bemutattunk különböző kölcsönös kizárást megoldó algoritmusokat, amelyek olvasható/írható közös memóriát használnak. Valamennyi biztosítja a kölcsönös kizárás és a haladás alapvető feltételeit, és a legtöbb biztosítja a magasszintű pártatlanság valamilyen fajtáját: kizárásmentességet, egy időkorlátot vagy egy megkerülési korlátot. Egy dolog közös minden algoritmusban, hogy mindegyikük legalább n közös változót használ.

Ebben az alfejezetben megmutatjuk, hogy ez nem véletlen: kiderül, hogy a kölcsönös kizárás feladat egyáltalán nem oldható meg n-nél kevesebb olvasható/írható osztott változóval. Ez még akkor is így van, ha csak az alapvető feltételeket követeljük meg, – a kölcsönös kizárást és haladást. Ezen alsó korlát bizonyításához nem szükséges magasszintű pártatlansági követelmény. A megoldhatatlansági eredmény független az közös változók méretétől is (a felvehető értékek számával mérve), amelyek lehetnek olyan kicsik, mint egy bit, vagy akár nem korlátos méretűek. Ez az eredmény a közös memóriájú rendszerek kifejezőerejéről állít egy alapvető korlátot.

Két meghatározásra van szükségünk. Elsőként: hasonlóan ahhoz, ahogy a 9.3. alfejezetben tettük, ha a P_i folyamat állapota, az U_i állapota és az összes közös változó értéke azonos s és s'-ben, akkor azt mondjuk, hogy ez az s és s' rendszerállapot a P_i folyamat szempontjából *megkülönböztethetetlen*, amit $s \overset{i}{\sim} s'$-vel jelölünk. Másodszor, ha minden folyamat a haladási szakaszában van s-ben, akkor azt mondjuk, hogy az s rendszerállapot *üresjárat*.

A bizonyításban felhasználói automaták rögzített halmazát tekintjük. Fel-

tesszük, hogy minden U_i felhasználó a lehető legnagyobb mértékben nemdeterminisztikus, azaz bármikor végrehajthatja *belépési* és *kilépési* kimenetét, csak a jólformáltság feltételének kell teljesülnie. Az általánosság megszorítása nélkül fordíthatjuk a figyelmet a felhasználók ezen halmazára, mert feltesszük, hogy az algoritmus a felhasználók bármely halmazára működik. Annak igazolását, hogy minden P_i-re valóban létezik egyetlen olyan U_i b/k automata, amely pontosan a megengedhető nemdeterminisztikusságot mutatja, meghagyjuk gyakorlatnak (lásd 10-29. gyakorlat).

10.8.1.. Alapvető tények

A bizonyítás két alapvető tényt használ. Az első, hogy egy egyedül futó folyamat üresjárati állapotból elérheti a belépési szakaszát.

10.29. lemma. . *Tegyük fel, hogy egy A algoritmus megoldja a kölcsönös kizárás feladatot (azaz biztosítja a jólformáltságot, a kölcsönös kizárást és a haladást) $n \geq 2$ folyamatra, csak olvasható/írható közös változókat használva. Tegyük fel, hogy s egy elérhető üresjárati állapot és legyen P_i egy tetszőleges folyamat.*

Ekkor létezik olyan, az s állapotból induló és csak a P_i lépéseit tartalmazó végrehajtási részsorozat[3]*, amelyben P_i folyamat eléri Be-t.*

Bizonyítás. Következik a haladás feltételéből. (Formálisan, a haladás feltételét alkalmazva egy s-t tartalmazó alacsony szintű pártatlan végrehajtási sorozatra, amelyben P_i s előfordulása után belép Pr-be.) □

Egyszerű következményként azt kapjuk, hogy egy egyedül futó folyamat egy üresjárati állapotnak *tűnő* rendszerállapotból elérheti Be-t.

10.30. lemma. . *Tegyük fel, hogy A megoldja a kölcsönös kizárás feladtot $n \geq 2$ folyamatra csak olvasható/írható közös változókat használva. Legyen s és s' elérhető rendszerállapotok, amelyek P_i szempontjából megkülönböztethetetlenek és tegyük fel, hogy s' egy üresjárati állapot.*

Ekkor létezik olyan, az s állapotból induló, és csak a P_i lépéseit tartalmazó végrehajtási részsorozat, amelyben P_i folyamat eléri Be-t.

A második alapvető tény, hogy bármely folyamat, amelyik eléri Be-t, biztosan ír valamit a közös memóriába, mielőtt elérné azt.

10.31. lemma. . *Tegyük fel, hogy A megoldja a kölcsönös kizárási feladatot $n \geq 2$ folyamatra csak olvasható/írható közös változókat használva. Tegyük fel, hogy s egy elérhető állapot, amelyben a P_i folyamat a haladási szakaszban van. Tegyük fel, hogy a P_i folyamat eléri Be-t egy végrehajtási részsorozatban, amely s-ből indul és csak P_i lépéseit tartalmazza. Ekkor, útközben, P_i-nek írnia kell valamelyik közös változót.*

[3] Ez egyszerűen egy olyan végrehajtási sorozat, amely tetszőleges állapotból indul, nem feltétlenül az algoritmus egy kezdőállapotából.

Bizonyítás. Legyen α_1 tetszőleges véges végrehajtási részsorozat, amely s állapotból indul, csak P_i lépéseit tartalmazza, és Be-ben fejeződik be a P_i folyamat lépésével. Tegyük fel indirekt, hogy α_1 nem tartalmaz közös változó írását. Jelölje s' az α_1-et befejező állapotot. Mivel a P_i folyamat nem ír közös változót, az s és s' közötti különbség csak P_i folyamat és U_i felhasználó állapotában van. Így $s \overset{j}{\sim} s'$ minden $j \neq i$-re.

A haladás feltételének ismételt alkalmazásából következik, hogy létezik olyan s-ből induló végrehajtási részsorozat, amely nem tartalmazza a P_i folyamat egyik olyan lépését sem, amelyben valamely folyamat elérné Be-t. Mivel $s \overset{j}{\sim} s'$ bármely $j \neq i$-re, szintén létezik ilyen, az s'-ből induló végrehajtási részsorozat is.

Ez alapján könnyen adhatunk ellenpéldát: az α végrehajtási sorozatot. Az α végrehajtási sorozat az s elérhető állapotba vezető befejezett végrehajtási részsorozattal kezdődik, α_1-gyel folytatódik, így P_i-t beengedi Be-be közös változó írása nélkül. Azzal fejeződik be, hogy egy másik s'-ből induló folyamat Be-be mehet P_i lépései nélkül. Ez megsérti a kölcsönös kizárási feltételt, mert α befejeződésekor két folyamat van Be-ben. □

10.8.2.. Kizárólagosan írható közös változók

Ha a közös változókat kizárólagosan írható/megosztottan olvasható olvasó/író regiszterekre korlátozzuk (ahogy a változókat a BurnsKK és Váróterem algoritmusban használtuk), akkor a 10.29. és 10.31. lemmákból azonnal következik az alsó korlát.

10.32. lemma. . *Ha az A algoritmus megoldja a kölcsönös kizárási feladatot $n \geq 2$ folyamat esetében csak kizárólagosan írható/megosztottan olvasható olvasó/író közös változókat használva, akkor A legalább n közös változót használ.*

Bizonyítás. Tekintsünk egy tetszőleges P_i folyamatot. A 10.29. lemma szerint az egyedül futó P_i elérheti Be-t az A kezdeti (üresjárati) rendszerállapotából. Ekkor a 10.31. lemmából következik, hogy P_i-nek útközben kell írnia közös változót. Mivel ez fennáll minden P_i folyamatra, és mivel minden közös változót csak egyetlen folyamat írhat, legalább n közös változó szükséges. □

10.8.3.. Megosztottan írható közös változók

Megjegyezzük, hogy még ahhoz az algoritmushoz is, amelyet bemutattunk, s amely megosztottan írható regisztereket használ (mint a DijkstraKK és Peterson algoritmus), szükséges legalább n változó. Ebben a szakaszban kiterjesztjük a 10.32 tételt a megosztottan írható regiszterek esetére.

10.33. tétel. . *Ha az A algoritmus megoldja a kölcsönös kizárási feladatot $n \geq 2$ folyamat esetében csak olvasható/írható közös változókat használva, akkor A-nak legalább n közös változót kell használnia.*

Hogy a bizonyítás alapgondolatát bemutassuk, két speciális eset vizsgálatával kezdünk. Elsőként megmutatjuk a két folyamat-egy változó, aztán a három folyamat-két változó lehetetlenségét. Ezután kiterjesztjük gondolatmenetünket az általános esetre.

Két folyamat és egy változó.. A következő meghatározást többször használjuk a bizonyításban: azt mondjuk, hogy a P_i folyamat *befedi* x közös változót az s rendszerállapotban, feltéve, hogy az s állapotban a P_i folyamat írhatja x-et. (Azaz a P_i folyamat írhatja x-et a következő lépésében.)

10.34. tétel. . *Nincs olyan algoritmus, amely csak egy olvasható/írható közös változót használva megoldja a kölcsönös kizárási feladatot két folyamat esetében.*

Bizonyítás. Tegyük fel indirekt, hogy létezik olyan A algoritmus, amely egyetlen x közös regisztert használ. Legyen s egy kezdeti (üresjárati) rendszerállapot. Létrehozzuk A egy végrehajtását, amely megsérti a kölcsönös kizárást.

A 10.29. és 10.31. lemmából következik, hogy létezik olyan végrehajtási sorozat, amely csak a P_1 folyamatot tartalmazza, s-ből indul, a P_1 folyamat belép Be-be, és ez előtt ír az egyetlen x közös változóba. P_1 befedi x-et közvetlenül x írása előtt. Legyen α_1 ezen végrehajtás előtagja addig az első pontig, ahol a P_1 folyamat befedi x-et, és jelölje s' az α_1 befejező állapotát. Jegyezzük meg, hogy $s \overset{2}{\sim} s'$, mivel a P_1 folyamat semmit sem ír a közös memóriába α_1 alatt. Ekkor a 10.30. lemmából következik, hogy egyedül a P_2 folyamat érheti el Be-t az s' állapotból indulva.

Az α ellenpélda végrehajtási sorozat α_1-gyel kezdődik, így a P_1 folyamatot az s' állapotba hozza, ahol befedi x-et. Ezután megengedi P_2-nek, hogy elérje Be-t, amely egyedül fut s'-ből. Majd folytatjuk P_1-t, megengedve, hogy x-et írja, ezáltal bármit felülírva, P_2 folyamat pedig írhatott volna útközben Be felé. Ez kiküszöböli a P_2 folyamat végrehajtásának összes történetét. Így a P_1 folyamat folytathatja futását éppen úgy, ahogy az egyedüli végrehajtásában tenné, és eléri Be-t. De mindkét folyamatot Be-be teszi, ami ellentmond a kölcsönös kizárás követelményének.

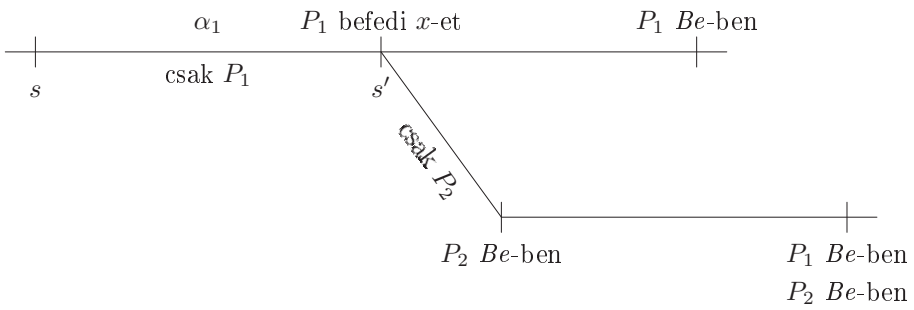
Az α végrehajtási sorozatot a 10.10. ábra írja le. Ez „fonja össze" olyan végrehajtási sorozatba a P_2 folyamat néhány lépését, amely csak P_1-et tartalmazza. □

Három folyamat és két változó.. Most megmutatjuk a megoldhatatlanságot három folyamat és két változó esetében.

10.35. tétel. . *Nincs olyan algoritmus, amely megoldja a kölcsönös kizárási feladatot három folyamat és két írható/olvasható változó esetében.*

Bizonyítás. Tegyük fel indirekt, hogy létezik olyan A algoritmus, amely x és y közös regisztereket használ. Legyen s egy kezdeti rendszerállapot. Létrehozzuk A egy végrehajtását, amely megsérti a kölcsönös kizárást.

A következő stratégiát használjuk. Kiindulunk s-ből, és csak P_1 és P_2 folyamat között lépkedünk egy olyan pontig, ahol mindkettő befedi x és y egyikét; ezenkívül

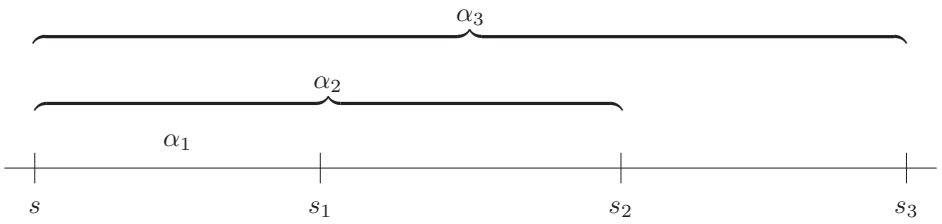


10.10.. ábra. A 10.34. tétel bizonyításában szereplő α végrehajtási sorozat.

az így elért s' állapot megkülönböztethetetlen lesz a P_3 folyamat szempontjából egy elérhető üresjárati állapottól. Ezután önmagában futtatjuk a P_3 folyamatot az s' állapottól, amíg az el nem éri Be-t. A 10.30. lemmából következik, hogy ez lehetséges.

Ezután lehetővé tesszük, hogy P_1 és P_2 folyamat is egy lépést tegyen. Miután mindkettő befedi a két közös változó egyikét, kiküszöbölhetik a P_3 folyamat végrehajtásának minden történetét. Ekkor lehetővé tesszük, hogy P_1 és P_2 folyamat további lépéseket tegyen; mivel kiküszöbölték a P_3 folyamat minden történetét, úgy futhatnak, mintha a P_3 folyamat soha nem lépett volna be a próba szakaszába. Így a haladás feltétele miatt P_1 vagy P_2 végül eléri Be-t. De ez ahhoz vezet, hogy két folyamat van Be-ben, ami ellentmond a kölcsönös kizárási feltételnek.

Még az maradt hátra, hogy megmutassuk, hogyan lépkedünk a P_1 és P_2 folyamat között, hogy befedjék a két közös változót, amíg a P_3 folyamat még az Ha-ben tartózkodik. Ezt a következőképpen tesszük (lásd 10.11. ábrát).



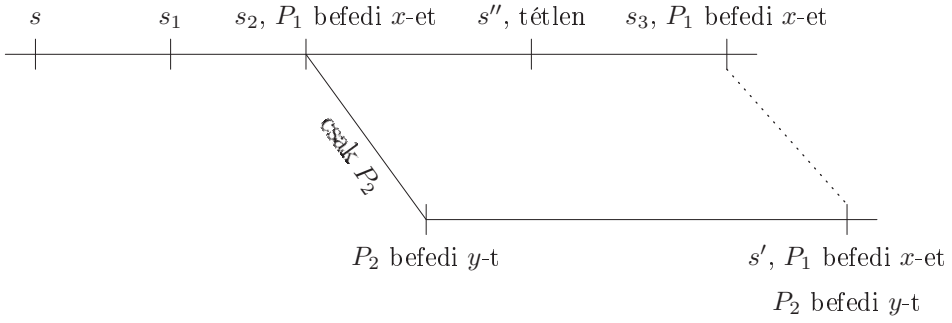
10.11.. ábra. A P_1 folyamat egyedül fut.

Először létrehozunk egy α_1 végrehajtási sorozatot úgy, hogy a P_1 folyamatot önmagában futtatjuk s-ből, amíg először le nem fed egy közös változót. Ezután kiterjesztjük α_1-et α_2-re a P_1 folyamat önmagában való továbbfuttatásával, amíg belép Be-be, Ki-be, Ha-be, majd Pr-be, újból és újból befed valamely közös változót. Kiterjesztjük α_2-t α_3-ra azonos módon. Legyen α_1-nek, α_2-nek és α_3-nak a befejező állapota sorrendben s_1, s_2, illetve s_3.

Mivel csak két közös változó van, a P_1 folyamatnak a három állapot (s_1, s_2, illetve s_3) közül kettőben *ugyanazt a változót* kell befednie. Konkrétabban: tegyük fel, hogy az s_2 és s_3 állapotban a P_1 folyamat az x változót fedi be. (Az összes többi esetre hasonló bizonyítást adhatunk.)

Most nézzük meg, mi történik, ha a P_2 folyamatot önmagában futtatjuk az s_2 állapotból indítva. Azt állítjuk, hogy a P_2 folyamat beléphet Be-be; ez a 10.30. lemmából következik, mivel a P_2 folyamat szempontjából az s_2 állapot megkülönböztethetetlen az utolsó α-beli megelőző állapottól, amelyben a P_1 folyamat Ha-ben van. Ezenkívül azt állítjuk, hogy útközben a P_2 folyamatnak a másik, y közös változót írnia kell. Egyébként a P_2 folyamat elérheti Be-t, majd a P_1 megtehet egy lépést felülírva bármit, amit a P_2 folyamat írt az x változóba, és így kiküszöböli P_2 minden történetét, majd ezután a P_1 folyamat folytathatja és megsértheti a kölcsönös kizárást.

Most létrehozunk egy α ellenpélda végrehajtási sorozatot (lásd 10.12. ábrát). Az α végrehajtási sorozat α_2-vel kezdődik, így a P_1 folyamatot egy olyan pontba viszi, ahol befedi x-et. Majd úgy folytatódik, hogy a P_2 folyamat addig az első pontig futhat, ahol befedi y-t. Ebben a pontban a P_1 és P_2 folyamat rendre befedi x-et és y-t. De még nem vagyunk készen, mert az eredményül kapott állapotnak megkülönböztethetetlennek kell lennie a P_3 folyamat szempontjából valamely üresjárati állapottól. (Az előbbi szituációban a P_2 folyamat írhatta volna x-t azután, hogy legutóbb elhagyta Ha-t, amelyet a P_3 folyamat vehetne észre.)



10.12.. ábra. α létrehozása.

Attól a ponttól folytatjuk a P_1 folyamatot, ahol a P_2 folyamat befedi y-t. P_1 először írhatja x-et, ezáltal kiküszöbölve P_2 folyamat minden történetét. Ekkor P_1 folytathatja futását éppen úgy, ahogy egyedüli végrehajtásakor, és elér egy pontig, amely olyan, mintha α_3 utáni pont lenne, ahol ismét befedi x-et. Ez befejezi az α létrehozását; legyen s' az α befejező állapota.

Azt állítjuk, hogy α minden olyan tulajdonsággal rendelkezik, amelyeket akarunk. Könnyen látható, hogy P_1 és P_2 rendre befedi x és y változót az s' állapotban. Meg kell mutatnunk, hogy s' megkülönböztethetetlen a P_3 folyamat szempontjából az elérhető üresjárati állapotból. Legyen s'' az utolsó üresjárati állapot, amely bekövetkezik α_3-ban. Ekkor az s'' és s_3 között az egyetlen különb-

ség csak P_1 folyamat és U_1 felhasználó állapotában van, míg s' és s_3 között az egyetlen különbség csak P_2 folyamat és U_2 felhasználó állapotában van. Ebből következik, hogy $s' \overset{3}{\sim} s''$, amit be akartunk látni. □

Az általános eset.. Az általános eset bizonyítása a két speciális eset természetes kiterjesztése, amelyben a változók száma szerinti indukciót használunk. Ehhez még egy alaptényre, a 10.31. lemma erősebb változatára van szükségünk. Ez azt mondja, hogy egy folyamatnak nem csak valamely változót kell írnia a *Be*-ig tartó úton, hanem valójában olyan változót kell írnia, amelyet nem fed le más folyamat. Ez az ötletet korábban már a 10.35. tétel bizonyításában használtuk.

10.36. tétel. . *Tegyük fel, hogy A megoldja a kölcsönös kizárási feladatot $n \geq 2$ folyamatra csak olvasható/írható közös változókat használva. Tegyük fel, hogy s egy elérhető rendszerállapot, amelyben a P_i folyamat a haladási szakaszban van. Tegyük fel, hogy a P_i folyamat eléri Be-t egy s-ből induló végrehajtási részsorozatban, amely csak a P_i folyamat lépéseit tartalmazza. Ekkor útközben, P_i-nek írnia kell valamely olyan közös változót, amelyet s-ben nem fed le más folyamat.*

Bizonyításvázlat. A bizonyítás a 10.31. lemmáéhoz hasonló. A fő különbség, hogy biztosítanunk kell, hogy a más folyamatokat is magába foglaló végrehajtási részsorozat az egyes folyamatok egy olyan lépésével kezdődjön, amely befed egy közös változót, így felülírva azt a változót. Így lehetséges, hogy a többi folyamat kiküszöböli a P_i folyamat minden történetét. A részletes bizonyítást meghagyjuk gyakorlatnak (lásd 10-30. gyakorlat). □

Most bebizonyíthatjuk a fő lemmát. Bármely k-ra ($1 \leq k \leq n$) azt mondjuk, hogy egy rendszerállapot egy másikból *k-elérhető*, ha csak a $P_1, \ldots, P_k$ folyamatok lépésit használva el lehet érni.

10.37. lemma. . *Tegyük fel, hogy A megoldja a kölcsönös kizárási feladatot $n \geq 2$ folyamatra csak olvasható/írható közös változókat használva. Feltesszük, hogy pontosan $n-1$ közös változó van. Legyen s egy tetszőleges elérhető üresjárati rendszerállapot. Tegyük fel, hogy $1 \leq k \leq n-1$. Ekkor létezik két olyan rendszerállapot, s' és s'', amelyek k-elérhetőek s-ből, és amelyek rendelkeznek a következő tulajdonságokkal:*

1. *s'-ben a $P_1, \ldots, P_k$ folyamatok k különböző változót fednek le;*
2. *s'' egy üresjárati állapot;*
3. *$s' \overset{i}{\sim} s''$ minden i-re ($k+1 \leq i \leq n$).*

Bizonyítás. k szerinti indukcióval.

Alapeset: $k = 1$. Egyedül a P_1 folyamatot futtatjuk az s-ből amíg először befed egy közös változót. A 10.29. és 10.31. lemmákból következik, hogy ez lehetséges. Legyen s' az eredményül kapott állapot, ekkor $s'' = s$ adja a szükséges tulajdonságokat.

Indukciós lépés. Tegyük fel, hogy a lemma fennáll k-ra, ahol $1 \leq k \leq n-2$; bebizonyítjuk $(k+1)$-re. Felhasználva az indukciós feltevést, egy t_1 állapotot kapunk, amely k-elérhető s-ből, és amelyben a $P_1, \ldots, P_k$ folyamatok befednek k

különböző változót; azonban t_1 megkülönböztethetetlen a $P_{k+1}, \ldots, P_n$ folyamatok szempontjából valamely olyan üresjárati állapottól, amely szintén k-elérhető s-ből. Majd megengedjük minden $P_1, \ldots, P_k$ folyamatnak, hogy tegyen egy lépést t_1-ből, ezáltal írják az általuk befedett változót. Ekkor megengedjük a $P_1, \ldots, P_k$ folyamatoknak, hogy tovább haladjanak Ha-be, amelyek így egy új, u_1 üresjárati állapotot eredményeznek.

Most ismét alkalmazzuk az indukciós feltevést, hogy egy olyan t_2 állapotot kapjunk, amely k-elérhető az u_1-ből és amelyben a $P_1, \ldots, P_k$ folyamatok befednek k különböző változót, sőt amely még megkülönböztethetetlen a $P_{k+1}, \ldots, P_n$ folyamatok szempontjából egy üresjárati állapotból, amely k-elérhető u_1-ből. Ismét megengedjük a $P_1, \ldots, P_k$ folyamatoknak, hogy írják a befedett változóikat és visszatérjenek egy u_2 üresjárati állapothoz.

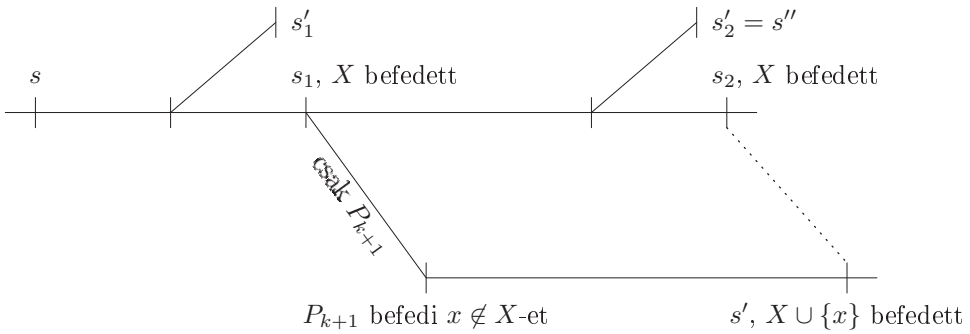
Ismételjük ezt az eljárást összesen $\binom{n-1}{k} + 1$-szer $t_1, \ldots, t_{\binom{n-1}{k}+1}$ „befedő állapotokat" kapva. Most, a skatulyaelv szerint e $\binom{n-1}{k} + 1$ befedő állapot között kell lennie kettőnek, amelyekben a $P_1, \ldots, P_k$ folyamatok a k közös változóknak ugyanazon halmazát fedik le; jelölje ezt a halmazt X. Legyen s_1 az első ilyen befedési állapot, s_2 a második. Ezen kívül legyen s_1' az az üresjárati állapot, amelyet úgy hoztunk létre, hogy s_1-nek megfelel, illetve s_2' az s_2-nek megfelelő. Így $s_1 \overset{i}{\sim} s_1'$ és $s_2 \overset{i}{\sim} s_2'$ minden i-re, $k+1 \leq i \leq n$.

Most vizsgáljuk meg, mi történik, ha önállóan futtatjuk a P_{k+1} folyamatot s_1 rendszerállapotból. Mivel $s_1 \overset{k+1}{\sim} s_1'$ és s_1' egy elérhető üresjárati állapot, a 10.30. lemmából következik, hogy a P_{k+1} folyamat bizonyosan belép Be-be. Útközben, a 10.36. lemma szerint, írnia kell valamelyik X-en kívüli x változót.

Most már készen vagyunk arra, hogy meghatározzuk a szükséges s' és s'' állapotokat, amelyek mindketten $(k+1)$-elérhetőek az eredeti s állapotból. (Lásd 10.13. ábrát). Elsőként az s' meghatározásához (az az állapot, amelyben $k+1$ változó befedett) futassuk a P_{k+1} folyamatot az s_1-ből addig, amíg először fed le egy nem X-beli közös változót. Ezután továbbindítjuk a $P_1, \ldots, P_k$ folyamatokat az első befedett változó írásáig, majd tovább az s_2-vel egyező pontig, ahol X-et ismét befedik. Legyen s' az eredményül kapott állapot. Jegyezzük meg, hogy s' azonos s_2-vel, kivéve a P_{k+1} folyamat állapotát. Az s'' (üresjárati állapot) meghatározásához legyen $s'' = s_2'$.

Azt állítjuk, hogy s' és s'' rendelkezik az összes szükséges tulajdonsággal. Elsőként jegyezzük meg, hogy csak a $P_1, \ldots, P_{k+1}$ folyamatok tartoznak bele a konstrukcióba (beleértve az indukciós feltevés alkalmazásait), így s és s'' $(k+1)$-elérhetőek s-ből. Azt is könnyen látni, hogy $k+1$ változó befedett s'-ben: a k változó X-ben és az új x változó, amelyet a P_{k+1} folyamat fed le. Továbbá az s_2' meghatározásából kapjuk, hogy $s'' = s_2'$ egy üresjárati állapot.

Már csak azt kell megmutatni, hogy s' és s'' megkülönböztethetetlenek minden P_i $(k+2 \leq i \leq n)$ folyamat szempontjából. De s_2 és s_2' meghatározásából következik, hogy s_2 és $s'' = s_2'$ megkülönböztethetetlenek minden P_i $(k+1 \leq i \leq n)$ folyamat szempontjából. Azt már megjegyeztük, hogy s_2 és s' megkülönböztethetetlenek minden folyamat szempontjából, kivéve a P_{k+1} folyamatot. E kettőből együtt következik a szükséges feltétel. □



10.13.. ábra. Az általános eset létrehozása.

Most a 10.33. tétel bizonyítása következik.

Bizonyítás. Tegyük fel indirekt, hogy az A algoritmus megoldja a kölcsönös kizárási feladatot $n \geq 2$ folyamatra legfeljebb $n-1$ olvasható/írható közös változót használva. Az általánosság megsértése nélkül feltehetjük, hogy A-nak pontosan $n-1$ közös változója van.

Legyen s az A tetszőleges kezdeti rendszerállapota. Ekkor a 10.37. lemmából következik, hogy van s' és s'' rendszerállapot, amely $(n-1)$-elérhető s-ből úgy, hogy mind az $n-1$ közös változót befedik a $P_1, \ldots, P_{n-1}$ folyamatok s'-ben, s'' üresjárati állapot és $s' \overset{n}{\sim} s''$. A 10.30. lemmából következik, hogy létezik egy s'-ből induló végrehajtási szelet, amely csak P_n folyamat lépéseit tartalmazza, és amelyben a P_n folyamat eléri Be-t. A 10.36. lemmából következik, hogy ebben a végrehajtási szeletben a P_n folyamatnak írnia kell valamelyik közös változót, amely nincs befedve s'-ben. De az összes $n-1$ változó befedett s'-ben, így ez ellentmondás. □

Ismét hangsúlyozzuk, hogy a 10.33. lemma nem függ a közös változók méretétől. Ezenkívül nem szükséges magasszintű pártatlanság; a haladás feltétele az egyetlen olyan élénkségi feltétel, amely szükséges ehhez a megoldhatatlansági eredményhez.

10.9.. Kölcsönös kizárás olvasható-módosítható-írható típusú közös változókkal

Ebben az utolsó alfejezetben áttekintjük az *olvasható-módosítható-írható* közös memóriát használó kölcsönös kizárást. Ez azt jelenti, hogy egy folyamat képes az új változóérték és folyamatállapot meghatározásához egyetlen, egy pillanatig tartó lépésben elérni az közös változót, használni értékét és a folyamat állapotát. Egy formális meghatározás a 9.4. alfejezetben található.

Azt gondolhatjuk, hogy a kölcsönös kizárás feladat áttekintése az olvasható-módosítható-írható modellben triviális feladat, mert ez a modell nagyon erőteljes, hatásos, hatékony. Az olvasható-módosítható-írható modell pártatlan, kizáróla-

gos jogot ad minden közös változó eléréséhez; minden folyamat pártatlanul kap lehetőséget, hogy elérje a változót, s amikor ezt megteszi, végrehajthat egy tetszőleges számítást mielőtt elengedné a változót. Ez nagyon hasonlít ahhoz, amit egy pártatlan kölcsönös kizárás algoritmustól megkövetelünk, azaz a pártatlan, kizárólagos hozzáférést a belépési szakaszhoz. Ez majdnem úgy néz ki, mintha feltételeznénk annak a nagy problémának egy megoldását, amelyet megpróbálunk megoldani.

Valóban, a közös memória ilyen erőteljes formája jelentős mértékben leegyszerűsíti a helyzetet, de nem oldja meg az összes nehézséget. Algoritmusok egy halmazával együtt be fogunk mutatni néhány nemtriviális alsó korlát eredményt.

Először áttekintjük a kölcsönös kizárás alapproblémáját, majd megnézzük, mi történik, ha hozzáadunk egy magasszintű pártatlansági követelményt: korlátozott megkerülést vagy kizárásmentességet.

Az alfejezet hátralévő részében feltesszük, hogy a közös memóriájú rendszer csak *egyetlen* közös változót tartalmaz. Az olvasható-módosítható-írható modellben ez nem jelenti az általánosság megszorítását, mert több olvasható-módosítható-írható változó egyesíthető egyetlen többrészes olvasható-módosítható-írható változóban. Ez ellentétben áll az olvasható/írható modell szituációjával, amelyhez már megmutattuk, hogy a kölcsönös kizárás feladat megoldásának létezése ugyancsak függ az olvasható/írható közös változók számától.

10.9.1.. Az alapfeladat

Az olvasható-módosítható-írható és az olvasható/írható modell közötti különbség látásához, tekintsük a következő triviális, egy változós EGYSZERŰKK algoritmust. Ebben az algoritmusban az x közös változónak pontosan akkor 1 az értéke, ha valamely folyamat számára erőforrás van engedélyezve. Bármely próba szakaszban lévő folyamat egyszerűen addig vizsgálja x-et, amíg azt nem találja, hogy $x = 0$; ekkor azonnal végrehajtja az $x := 1$ utasítást. Kilépéskor a folyamat visszaállítja x-et 0-ra. Egyszerű belátni, hogy a EGYSZERŰKK algoritmus megoldja a kölcsönös kizárási feladatot.

10.10. algoritmus. EGYSZERŰKK

Közös változók:
$x \in \{0, 1\}$, kezdetben 0

P_i műveletei:

Bemeneti:
- `próbál`$_i$
- `kilép`$_i$

Kimeneti:
- `belép`$_i$
- `halad`$_i$

Belső:
- `vizsgál`$_i$
- `visszaállít`$_i$

P_i állapotai:
$p\_sz \in \{haladás, vizsgálat, elhagy\_próba, belépés, visszaállítás, elhagy\_kilépés\}$,
kezdetben *haladás*

P_i átmenetei:

próbál$_i$
Hatás:
$p\_sz := vizsgálat$

vizsgál$_i$
Előfeltétel:
$p\_sz = vizsgálat$
Hatás:
if $x = 0$ **then**
$x := 1$
$p\_sz := elhagy\_próba$

visszaállít$_i$
Előfeltétel:
$p\_sz = visszaállítás$
Hatás:
$x := 0$
$p\_sz := elhagy\_kilépés$

belép$_i$
Előfeltétel:
$p\_sz = elhagy\_próbál$
Hatás:
$p\_sz := belépés$

kilép$_i$
Előfeltétel:
$p\_sz = visszaállítás$

halad$_i$
Előfeltétel:
$p\_sz = elhagy\_próba$
Hatás:
$p\_sz := haladás$

10.38. tétel. . *Az* EGYSZERŰKK *algoritmus megoldja kölcsönös kizárási feladatot.*

10.9.2.. Korlátozott megkerülés

Az EGYSZERŰKK algoritmus nem biztosít magasszintű pártatlansági feltételeket. Könnyen kaphatunk azonban igen erős magasszintű pártatlansági feltételeket, még akár egy FIFO feltételt is (ami azon alapul, hogy az első, helyileg vezérelt lépést minden egyes folyamat a próba szakaszában teszi), ami még szintén csak egyetlen közös változót használ. Például ilyen a SORKK algoritmus.

SORKK algoritmus (vázlatosan)

A folyamatok fenntartják a folyamatindexek egy sorát a közös változóban, amely kezdetben üres. Egy *Pr*-be lépő folyamat hozzáadja az indexét ennek a sornak a végéhez. Egy folyamat, amely az indexét a sor elején találja, *Be*-be megy; és amikor egy folyamat elhagyja *Be*-t, törli magát a sorból.

Formálisabban előfeltétel/hatás alakban kifejezve a következőt kapjuk.

10.11. algoritmus. SorKK

Közös változók:
sor, a folyamatok sorszámainak egy FIFO sora, kezdetben üres

P_i műveletei:

Bemeneti:
- `próbál`$_i$
- `kilép`$_i$

Kimeneti:
- `belép`$_i$
- `halad`$_i$

Belső:
- `kér`$_i$
- `vizsgál`$_i$
- `visszaállít`$_i$

P_i állapotai:
$p\_sz \in \{haladás, kérés, vizsgálat, elhagy\_próba, belépés, visszaállítás, elhagy\_kilépés\}$,
kezdetben *haladás*

P_i átmenetei:

`próbál`$_i$
- Hatás:
  - $p\_sz :=$ *kérés*

`kér`$_i$
- Előfeltétel:
  - $p\_sz =$ *kérés*
- Hatás:
  - adjuk hozzá i-t a *sor*-hoz
  - **if** i az első elem a *sor*-ban **then**
    - $p\_sz :=$ *elhagy_próba*
  - **else** $p\_sz :=$ *vizsgálat*

`vizsgál`$_i$
- Előfeltétel:
  - $p\_sz =$ *vizsgálat*
- Hatás:
  - **if** i az első elem a *sor*-ban **then**
    - $p\_sz :=$ *elhagy_próba*

`belép`$_i$
- Előfeltétel:
  - $p\_sz :=$ *elhagy_próba*
- Hatás:
  - $p\_sz :=$ *belépés*

`kilép`$_i$
- Előfeltétel:
  - $p\_sz =$ *visszaállítás*

`visszaállít`$_i$
- Előfeltétel:
  - $p\_sz =$ *visszaállítás*
- Hatás:
  - töröljük *sor* első elemét
  - $p\_sz :=$ *wlhagy_kilépés*

`halad`$_i$
- Előfeltétel:
  - $p\_sz =$ *elhagy_kilépés*
- Hatás:
  - $p\_sz :=$ *haladás*

Könnyen látható, hogy a SorKK biztosítja a jólformáltságot, a kölcsönös kizárást és a haladást. Továbbá, teljesíti a magasszintű pártatlansági feltételt,

amely szerint a belépési szakaszba való belépés olyan FIFO-nak felel meg, amely tekintettel van a próba szakaszban történt, első helyileg vezérelt műveletére (`kér` művelet). Így a SorKK biztosítja a korlátos megkerülést (az 1 korláttal).

10.39. tétel. . *A* SorKK *megoldja a kölcsönös kizárási feladatot és biztosítja a korlátozott megkerülést.*

A SorKK algoritmus egyszerű és gyors is, legalábbis az időmértékünk szerint, de probléma, hogy a közös változó nagyon nagy. Létezik $n! + (n-1)! + \ldots$ különböző sor, amely legfeljebb n különböző indexből áll, így a változónak fel kell tudnia venni ezeket a különböző értékeket. Ehhez $\Omega(n \log n)$ bitre van szükség. Jobb lenne csökkenteni a közös változó méretét, nemcsak azért, hogy közös memóriát spóroljunk meg, hanem azért is, mert nem ésszerű pillanatig tartó elérést feltételezni ilyen nagy változó esetében. Érdekes kérdés, hogy milyen nagynak kell lennie a változónak ahhoz, hogy biztosítsa a magasszintű pártatlanságot. Meg tudjuk oldani a problémát egy olyan változóval, amely az értékeknek egy n-ben lineáris számát veszi fel? És állandó értékkel?

Nem nehéz megvalósítani ugyanazt a FIFO típusú viselkedést, mint a SorKK algoritmusnál, egy változóval és csak n^2 értékkel ($2 \log n$ bit). Például használhatunk olyan algoritmust, amely „jegyeket" osztogat a belépési szakaszhoz való belépéshez.

JegyKK algoritmus (vázlatosan)

A közös változó felvesz egy (*következő*, *kiadott*) párt a $\{0, \ldots, n-1\}$ értékekből, amely kezdetben $(0, 0)$. A *következő* komponens fejezi ki a következő „jegyet" a belépési szakaszhoz, amely kiosztásra kerül egy folyamatnak, míg a *kiadott* komponens az utolsó „jegyet" fejezi ki, amelyet kiadtak a belépési szakaszba való belépés engedélyezéséhez. Amikor egy folyamat belép a próba szakaszba, „jegyet kap", azaz másolja és növeli a *következő* komponenst. Ha egy folyamat jegye egyenlő a *kiadott* komponenssel, a belépési szakaszba lép. Ha egy folyamat kilép, megnöveli a *kiadott* komponenst modulo n.

Most az algoritmust egy formálisabb előfeltétel/hatás alakban mutatjuk be..

10.12. algoritmus. JegyKK

Közös változók:
(*következő*, *kiadott*) pár $\{0, \ldots, n-1\}$ elemekkel, kezdetben $(0, 0)$

P_i műveletei:

Bemeneti:
- $\texttt{próbál}_i$
- $\texttt{kilép}_i$

Kimeneti:
- $\texttt{belép}_i$
- $\texttt{halad}_i$

Belső:
- $\texttt{kér}_i$
- $\texttt{vizsgál}_i$
- $\texttt{visszaállít}_i$

P_i állapotai:
$p\_sz \in \{haladás, kérés, vizsgálat, elhagy\_próba, belépés, visszaállítás, elhagy\_kilépés\}$, kezdetben *haladás*
$jegy \in \{0, \ldots, n-1\} \cup \{üres\}$, kezdetben *üres*

P_i átmenetei:

$\texttt{próbál}_i$
Hatás:
$p\_sz := kérés$

$\texttt{vizsgál}_i$
Előfeltétel:
$p\_sz = vizsgálat$
Hatás:
if $jegy = kiadott$ **then**
$p\_sz := elhagy\_próba$

$\texttt{belép}_i$
Előfeltétel:
$p\_sz := elhagy\_próba$
Hatás:
$p\_sz := belépés$

$\texttt{kilép}_i$
Előfeltétel:
$p\_sz = visszaállítás$

$\texttt{kér}_i$
Előfeltétel:
$p\_sz = kérés$
Hatás:
$jegy := következő$
$következő := következő + 1 \pmod n$
if $jegy = kiadott$ **then**
$p\_sz := elhagy\_próba$
else $p\_sz := vizsgálat$

$\texttt{visszaállít}_i$
Előfeltétel:
$p\_sz = visszaállítás$
Hatás:
$kiadott := kiadott + 1 \pmod n$
$jegy := üres$
$p\_sz := elhagy\_kilépés$

$\texttt{halad}_i$
Előfeltétel:
$p\_sz = elhagy\_kilépés$
Hatás:
$p\_sz := haladás$

A JegyKK teljesíti ugyanazokat a helyességi feltételeket, mint a SorKK, beleértve a FIFO tulajdonságot a próba szakaszban első helyileg vezérelt műve-

letre vonatkoztatva. A következő tétel bizonyítását a 10.9.4. alfejezetben adjuk meg.

10.40. tétel. . *A* JegyKK *megoldja a kölcsönös kizárási feladatot és biztosítja a korlátozott megkerülést a közös memória n^2 értékét használva.*

Megtehetjük ezt n^2-nél kevesebbel? A következő tétel egyszerű alsó korlátot ad n-re a korlátozott megkerülésű kölcsönös kizáráshoz szükséges közös változó értékei számára.

10.41. tétel. . *Legyen A egy n folyamatú kölcsönös kizárás algoritmus, amely egyetlen olvasható-módosítható-írható közös változót használva garantálja a korlátozott megkerülést. Ekkor azon különböző értékek száma, amelyeket a változó felvehet, legalább n.*

Bizonyítás. Tegyük fel, hogy az A n-folyamatú kölcsönös kizárás algoritmus biztosítja a korlátozott megkerülést egy a megkerülési korláttal. Az általánosság megszorítása nélkül feltehetjük (ahogy a 10.8. alfejezetben tettük), hogy az U_i felhasználók a lehető legnagyobb mértékben nemdeterminisztikusak. Indirekt bizonyítással egy ellentmondást konstruálunk, amelyben valamely folyamatot több mint a-szor kerülünk meg.

A bizonyítást az $\alpha_1, \alpha_2, \ldots, \alpha_n$ véges végrehajtási sorozatok sorozatának meghatározásával kezdjük, ahol mindegyik az előző egy kiterjesztése. Az α_1 végrehajtási sorozatot a P_1 folyamat kezdeti rendszerállapotból Be-be való lépéséig tartó önálló futásából kapjuk. (A haladás feltételéből következik, hogy ez lehetséges.) Úgy kapjuk α_2-t, hogy az α_1-t kiterjesztjük azzal, hogy megengedjük P_2 folyamatnak, hogy belépjen a próba szakaszba és egy helyileg vezérelt lépést tegyen. A P_2 folyamatnak nyilvánvalóan a próba szakaszában kell maradnia α_2 után is ahhoz, hogy elkerülje a kölcsönös kizárás megsértést. Ekkor α_i végrehajtási sorozat a $(3 \leq i \leq n)$ az α_2-höz hasonló módon állítható elő: α_{i-1} végénél kezdve megengedjük a P_i folyamatnak, hogy belépjen a próba szakaszba és egy helyileg vezérelt lépést tegyen. Minden P_i folyamat $(3 \leq i \leq n)$ szintén a saját próba szakaszában marad.

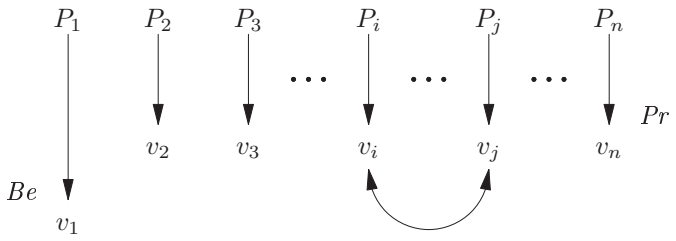
Legyen s_i rendszerállapot és v_i az közös változó értéke α_i után $(1 \leq i \leq n)$. Azt állítjuk, hogy $v_i \neq v_j$ $(1 \leq i, j \leq n, i \neq j)$, ami magában foglalja az eredményt.

Tegyük fel indirekt, miszerint bizonyos i-re és j-re $v_i = v_j$ $(i \neq j)$, és az általánosság megszorítása nélkül tegyük fel, hogy $i < j$. Ekkor $s_i \overset{k}{\sim} s_j$ minden P_k folyamatra, ahol $1 \leq k \leq i$. (Azaz az α_i és α_j utáni rendszerállapotok ugyanazokat az állapotokat tartalmazzák a $P_1, \ldots, P_i$ folyamatokra és $U_1, \ldots, U_i$ felhasználókra nézve, és a közös változó értéke is azonos.)

Most van egy olyan alacsony szintű pártatlan végrehajtás sorozat, ami α_i kiterjesztése csak a $P_1, \ldots, P_i$ folyamatokat tartalmazza, és azt okozza, hogy valamely folyamat végtelen sokszor lép be Be-be. Ez a haladás feltételéből következik (amelyet csak az alacsony szintű pártatlan végrehajtási sorozatokra alkalmazhatunk). Ugyanaz a lépés alkalmazható α_j után is, ismét engedélyezve egy olyan

végrehajtási sorozatot, amelyben ugyanaz a folyamat végtelen sokszor lép be Be-be. Megjegyezzük, hogy ez az új végrehajtás *nem* pártatlan alacsony szinten: a $P_{i+1}, \ldots, P_j$ folyamatok nem hajtanak végre lépéseket az α_j utáni végrehajtási részsorozatban, noha valamennyien a Pr-ben vannak. De ez nem okoz gondot: az alacsony szintű pártatlanság nem szükséges a megkerülési korlát megsértéséhez. Elég ennek a végrehajtási sorozatnak egy elég nagy részét futtatni, hogy a P_j folyamatot a-nál többször kerülje meg valamelyik másik folyamat, amely ellentmondáshoz vezet.

A konstrukciót a 10.14. ábra mutatja. □



10.14.. ábra. A 10.41. tétel bizonyításában szereplő végrehajtási sorozat létrehozása.

Vajon ez az alsó korlát a lehető legnagyobb, vagy növelhető $\Omega(n^2)$-ig? Kiderül, hogy nem: van egy *ellenpélda algoritmus* (azaz egy olyan algoritmus, amely önmagában nem túl érdekes, nem gyakorlatias, nem is szép, de ellenpéldát tud adni egy lehetetlenségi sejtésre), amelyhez csak $\mathcal{O}(n)$ érték kell. Valójában az algoritmushoz csak $n + k$ érték kell, ahol k kis konstans. Ezt PufferFőKK algoritmusnak hívjuk, hamarosan kiderülő okok miatt.

A PufferFőKK algoritmus (vázlatosan)

A PufferFőKK algoritmus alapvető ötlete a következő: A próba szakaszt két részre osztjuk, *puffer* szakaszra és *fő* szakaszra. Amikor folyamatok belépnek a próba szakaszba, akkor a puffer-ba kerülnek. A folyamatok között nincs sorrend felállítva. Amikor a fő szakasz üres, a pufferből minden folyamat a fő szakaszba megy, kiürítve a puffert. A fő szakaszból a folyamatok tetszőleges sorrendben, de egyesével követik egymást a belépési szakaszba.

Ennek az ötletnek a megvalósításához kommunikációs működési elvre van szükség, hogy a folyamatok tudják, mikor kell szakaszt váltaniuk. A megvalósítás első lépéseként tegyük fel, hogy a szokásos $P_1, \ldots, P_n$ „ügynökfolyamatokon" kívül megengedünk egy dedikált *felügyelő* folyamatot, amely bármikor tehet lépéseket. Egy olyan megoldást fogunk tervezni, amely központosítja a felügyelő folyamatban a rendszerirányítást: a felügyelő nyomon követi, hogy a folyamatoknak mikor kell szakaszt váltaniuk, és erről értesíti őket. Később leírjuk, hogyan küszöböljük ki egy ilyen felügyelő folyamat szükségességét.

A felügyelő folyamatot a következő stratégiával használjuk. Először a változónak legyen két komponense, egy a számlálónak ($számláló \in \{0, \ldots, n\}$ és egy az *üzenet*-nek, amelyet a vezérlő utasítások egy kijelölt, véges halmazából választunk. Ez összesen kn érték valamilyen konstans k-ra, de optimalizálhatjuk ezt $(n+k)$-ra, prioritási rendet használva, amely megengedi, hogy különböző típusú kommunikációkhoz újrahasználjuk az értékeket.

A felügyelő rendben tartja a helyi *puffer_ számláló* és *fő_ számláló* változókat, számolja azon folyamatok számát, amelyekről tud, hogy hányan vannak a puffer és fő szakaszban. Amikor egy folyamat belép a próba szakaszba, megnöveli az közös változó *számláló* komponensének értékét, így tájékoztatja a felügyelőt, hogy új folyamat lépett be, és a pufferben várakozik. A felügyelő amikor nem nulla *számláló*-értéket lát az közös változóban, hozzáadja a saját helyi *puffer_ számláló* változójához, és visszaállítja a változó *számláló* komponensét 0 értékre.

A felügyelő ki tudja számolni, mikor kell a pufferben lévő folyamatokat a fő szakaszba mozgatni: amikor a *fő_ számláló* értéke 0. Egyesével mozgatja át őket a puffer szakaszból a fő szakaszba úgy, hogy az közös változó *üzenet* komponensébe a *fő_ belép* üzeneteket teszi. A felügyelő akkor állítja meg ezt az átmozgatást, ha azt látja, hogy a *puffer_ számláló* és a *számláló* változó értéke egyaránt 0. Ekkor a felügyelő a folyamatokat átmozgatja a fő szakaszból a belépési szakaszba úgy, hogy az közös változó *üzenet* komponensébe a *krit_ belép* üzeneteket írja.

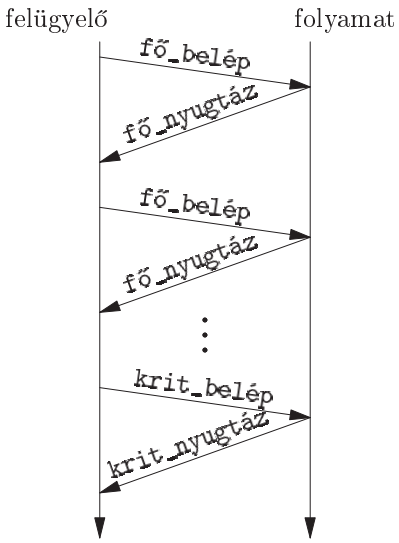
A vezérlőüzenet többféle lehet.

- *fő_ belép*: a felügyelő akkor helyezi el ezt az üzenetet az közös változó *üzenet* komponensében, amikor egy folyamatot a puffer szakaszból a fő szakaszba mozgat. Az első folyamat a puffer szakaszban, amely meglátja ezt az üzenetet, elkapja, és a folyamat a fő szakaszban folytatódik.
- *fő_ nyugtáz*: Az a folyamat, amelyik elkapott egy *fő_ belép* üzenetet az közös változótól, ezt az üzenetet teszi a helyébe, hogy értesítse a felügyelőt. A felügyelő ezt fogja elkapni.
- *krit_ belép*: A felügyelő ezt akkor helyezi el a közös változó *üzenet* komponensében, amikor egy folyamatot a fő szakaszból a belépési szakaszba mozgat. Az első olyan folyamat a fő szakaszban, amelyik meglátja ezt az üzenetet, elkapja, és a folyamat a belépési szakaszban folytatódik.
- *krit_ nyugtáz*: Az a folyamat, amely elkap egy *krit_ belép* üzenetet az osztott váltózótól, ezt az üzenetet teszi a helyébe, hogy értesítse a felügyelőt.
- *elvégzett*: Egy, a belépési szakaszból kilépő folyamat helyezi el ezt az üzenetet az közös változó *üzenet* komponensében, hogy bejelentse, hogy befejeződött a belépési szakasz.

Most körvonalazzuk, hogyan tudjuk elkerülni a két külön komponenst a közös változóban. Megjegyezzük, hogy a változó két célra szolgál: megjegyzi az újonnan belépett folyamatok számát és vezérlőüzeneteket továbbít. Az

közös változót most „időosztásos” változóként fogjuk használni, hogy megfeleljen mindkét célnak, de nem egyidőben. A változó minden pillanatban értéket tárol, azaz *valamikor* számlálót, *valamikor* vezérlőüzenetet, de nem mindkettőt.

Jegyezzük meg, hogy az eddig leírt algoritmusban a vezérlőüzenet kommunikáció összhangban halad egy egyszerű szekvenciális „vezérlőszállal”, ahogy a 10.15. ábra mutatja.



10.15.. ábra. Vezérlőszál a korlátozott megkerülésű kölcsönös kizárás algoritmushoz.

Tegyük fel, hogy ez a szál egy új P_i folyamat belépésekor megszakadt, amikor az felülír egy vezérlőüzenetet egy számlálóértékkel (1-gyel, ezzel jelentve be saját érkezését). Ekkor (mivel csak véges sok folyamat léphet be a rendszerbe) a rendszer biztosan elér egy stabil állapotot. A felügyelő végül minden száminformációt kiolvas az közös változóból, és a változóban a *számláló* értékét véglegesen 0-ra állítja. Ettől kezdve lehetséges lenne, hogy a P_i folyamat a felülírt üzenetet visszateszi a változóba, lehetővé téve ezáltal a vezérlőszálnak, hogy folytatódjon.

Pontosabban a következő történik. Amikor a P_i folyamat belép a próba szakaszba és egy vezérlőüzenetet talál a közös változóban, megjegyzi azt és kicseréli az 1-es számmal. A P_i folyamat addig tartja meg az üzenetet addig, amíg azt nem látja, hogy a *számláló* $= 0$-val, ekkor felülírja a megjegyzett üzenettel. Az eredmény egy kölcsönös kizárás algoritmus korlátozott megkerüléssel, amely az közös változó $n+6$ értékét használja, feltéve, hogy egy dedikált felügyelő folyamat elérhető.

Most úgy módosítjuk ezt az algoritmust, hogy azon a modellen dolgoz-

zon, amelyet tanultunk, azaz dedikált felügyelő folyamat nélkül. Az ötlet az, hogy megengedjük a folyamatoknak, hogy együttműködjenek a felügyelő osztott szimulációjában. (A szimulációnak osztottnak kell lennie, mivel nincs folyamat, amely minden pillanatban biztosan elérhető lenne.) A folyamatok egyszerűen felváltva végzik a szimulációt; nevezetesen, ha egy folyamat bármikor a belépési szakaszban van, akkor az lesz a felügyelő szimulálásáért felelős folyamat.

A fő nehézség ebben a szimulációban, hogy a *Be*-t elhagyó folyamatnak át kell adnia a felügyelő szimulálásának felelőségét a következő *Be*-be lépő folyamatnak. Ebből következik, hogy át kell adnia minden, a felügyelőnek szükséges állapotinformációt (nevezetesen a *puffer_számláló*-t és a *fő_számláló*-t) a következő folyamatnak. A közös változót kell erre az új típusú kommunikációra is használnunk, valamint a két másik kommunikációtípusra is, amelyről már korábban szó esett, ismét időosztást használva. Megjegyezzük, hogy az új állapotkommunikáció nem működik egy időben a vezérlőüzenet kommunikációval, tehát nincs ütközés. Továbbá, az újonnan érkezett folyamatok száma és az állapotinformáció kommunikációja közötti ütközést azonos módon lehet kezelni, mint a számlálók és a vezérlőüzenetek közötti ütközést.

Néha előfordulhat, hogy amikor egy folyamat elhagyja a *Be*-t, nincs másik folyamat, amelynek átadhatná a felügyelő szimulációjának felelősségét. Ez azt jelenti, hogy nincs másik folyamat a próba szakaszban. De ekkor nincs érdekes információ a felügyelőállapotban, így a folyamat egyszerűen elhárítja a felelősséget, és egy speciális jelzőt hagy a közös változóban.

10.42. tétel. . *A* PufferFőKK *algoritmus megoldja a kölcsönös kizárási feladatot, biztosítja a korlátozott megkerülést, miközben egyetlen egy, csupán $n + k$ értéket felvevő olvasható-módosítható-írható közös változót használ (valamilyen kicsi k konstanssal).*

10.9.3.. Kizárásmentesség

A 10.41. tételben szereplő alsó korlát csak akkor áll fenn, ha a magasszintű pártatlansági követelmény a korlátozott megkerülés. A kizárásmentesség gyengébb feltétele mellett a bizonyítás nem működik. A probléma az, hogy a kizárásmentesség alacsony szintű pártatlan végrehajtások tulajdonsága, és a bizonyításban létrehozott rossz végrehajtás nem pártatlan a $P_{i+1}, \ldots, P_j$ folyamatokkal. Valójában a 10.41. tétel eredménye nem áll fenn kizárásmentesség esetében. Van azonban egy másik ellenpélda algoritmusunk egy meglepően kicsi korláttal. Ezt az algoritmust Végrehajt algoritmusnak nevezzük.

Végrehajt algoritmus (vázlatosan)

Az algoritmus ötlete a következő. Amint a PufferFőKK algoritmusban láttuk, minden belépő folyamat megnövel egy *számláló*-t az osztott változóban, de ebben az algoritmusban a *számláló* csak $0, \ldots, \frac{n}{2}$ értékeket vehet fel,

mielőtt ismét 0-t vesz fel. A *számláló*-t kiolvassa egy (szimulált) felügyelő, ahogy korábban.

Amikor a *számláló* felveszi a 0-t, a folyamatok $\frac{n}{2}+1$ elemű halmaza átmenetileg „rejtett" lesz a rendszer többi része előtt; az eredményül kapott rendszerállapot megkülönbözhetetlen minden más folyamat számára attól az állapottól, amelybe közvetlenül az előtt voltak, hogy beléptek. Ha ezek a rejtett folyamatok nem tesznek további lépéseket, a rendszer többi része úgy fog viselkedni, mintha a rejtett folyamatok még mindig *Ha*-ben lennének. Azonban egy alacsony szintű pártatlan végrehajtásban a rejtett folyamatok további lépéseket fognak tenni, így jelenlétük ismertté válhat.

Hogy meggyőződjünk, a folyamatok nem maradnak rejtettek, kiválaszthatjuk azt a folyamatot, amely *végrehajtóként*, végrehajtja az átmenetet $\frac{n}{2}$-ből 0-ba, és átadjuk neki a rejtett folyamatokkal kapcsolatos felelősséget. A végrehajtó egy speciális *alszik* üzenetet küld $\frac{n}{2}$ folyamat egy tetszőleges részhalmazának a pufferbe, hogy egy rövid időre elaltassa őket. Ekkor, miután a végrehajtó kiemelt $\frac{n}{2}$ folyamatot a versenyből, visszalép a rendszerbe, és saját érdekében még egyszer megnöveli a *számlálót*. Most az $\frac{n}{2}$ alvó folyamattal az algoritmus pontosan úgy fut, mint a PufferFőKK algoritmus; a közös változó nem csordulhat túl, mert legfeljebb $\frac{n}{2}$ megmaradó aktív folyamat van, és a számláló a változóban elérheti az $\frac{n}{2}$-t. A legfontosabb, hogy egy második párhuzamosan működő végrehajtó nem jöhet létre. A PufferFőKK algoritmus viselkedésének megfelelően a végrehajtó végül eléri *Be*-t.

Amikor a végrehajtó eléri *Be*-t, gondoskodik az alvó folyamatokról úgy, hogy egy *felkelt* üzenetet küld és szól róluk a felügyelőnek. Ezekhez az újfajta kommunikációkhoz ismét időosztásos változóra van szükségünk, most egy kissé bonyolultabb prioritássémával.

10.43. tétel. . *A* Végrehajtó *algoritmus megoldja a kölcsönös kizárási feladatot, biztosítja a kizárásmentességet egyetlen csak $\frac{n}{2}+k$ értékű olvasható-módosítható-írható közös változót használva (valamilyen kis k konstanssal).*

Ezt az alfejezetet azzal fejezzük be, hogy kizárásmentes kölcsönös kizárás esetében a szükséges értékek számára talált alsó korlát megközelítőleg $\sqrt{n}$. Ez a korlát nem erős az $\frac{n}{2}+k$ felső korlátjára vonatkozóan, de a bizonyítás tartalmaz egy érdekes módszert rossz, alacsony szintű pártatlan végrehajtási sorozatok létrehozásához.

10.44. tétel. . *Legyen $k \geq 2$. Legyen A egy tetszőleges $n \geq \frac{k^2-k}{2}+1$ folyamatú rendszer, amely megoldja a kölcsönös kizárási feladatot és biztosítja a kizárásmentességet egyetlen olvasható-módosítható-írható közös változót használva. Ekkor a változó különböző értékeinek száma, amelyet felvehet az A elérhető állapotaiban, legalább k.*

Bizonyítás. Ismét feltesszük, hogy a felhasználók a lehető legnagyobb mértékben nemdeterminisztikusak. A bizonyításunk k-szerinti indukcióval történik.

Alapeset. $k = 2$. Ekkor az egyenlőtlenség szerint $n \geq 2$. Könnyű megmutatni, hogy a változó legalább két értéket vesz fel, mivel ellenkező esetben a folyamatok nem kommunikálhatnának. A formális érvelés hasonló a 10.31. lemmában használthoz.

Indukciós lépés. Tegyük fel, hogy a tétel fennáll $k \geq 2$ esetre, megmutatjuk, hogy $(k+1)$-re is fennáll. Tegyük fel, hogy $n \geq \frac{(k+1)^2-(k+1)}{2} + 1$, és tegyük fel indirekt, hogy az közös változók értékeinek száma szigorúan kisebb $(k+1)$-nél. Az indukciós feltevés szerint ebből következik, hogy az értékek száma legalább k, így pontosan k-nak kell lennie. Most létrehozunk egy rossz végrehajtási sorozatot, hogy ellenmondást kapjunk.

Az α_1 véges végrehajtási sorozatot P_1 folyamat Be-be való belépéséig tartó futásával határozzuk meg; legyen az eredményül kapott rendszerállapot s_1. Ezután kiterjesztjük α_1-t α_2-re a P_2 folyamatot addig futtatva, amíg eléri az s_2 rendszerállapotot, amelyben az közös változónak olyan értéke van, amelyet a P_2 folyamat okozhat, amelyet a P_2 folyamat végtelen sokszor tud ismételni azzal, hogy az s_2 állapotból egyedül fut. Ilyen állapot létezik, mivel a változó csak véges sok értéket vehet fel. Hasonlóképpen, meghatározzuk α_i-t $(3 \leq i \leq n)$ a P_i folyamatot futtatva α_{i-1} után, amíg elér egy s_i rendszerállapotot, amelyben az közös változónak olyan értéke van, amelyet a P_i folyamat végtelen sokszor tud ismételni azzal, hogy az s_i állapotból egyedül fut. Legyen v_i a változó értéke az s_i rendszerállapotban minden $1 \leq i \leq n$-re. Mivel csak k érték van, amelyet az közös változó felvehet, a skatulyaelvből következik, hogy kell lennie két folyamatnak, P_i-nek és P_j-nek $(n - k \leq i < j \leq n)$, hogy $v_i = v_j$. Rögzítsük ezt az i-t és j-t. Megjegyezzük, hogy $s_i \overset{m}{\sim} s_j$ minden P_m $(1 \leq m \leq i)$ folyamatra.

Most a $P_1, \ldots, P_i$ folyamatok egy legalább $\frac{k^2-k}{2}+1$ folyamatból álló rendszert alkotnak, amely kizárásmentességgel oldja meg a kölcsönös kizárási feladatot. Így az indukciós lépés szerint a közös memóriának mind a k értékét használniuk kell. Valójában szigoríthatjuk ezt a követelményt: minden s rendszerállapotra, amely i-elérhető[4], az s_i rendszerállapotból és a közös memória minden v értékére kell lennie egy rendszerállapotnak, amely i-elérhető s-ből, és amelyben a közös változó értéke v. (Ha nem, akkor használhatjuk bármely, s-ből i-elérhető üresjárati állapotot, mint a rendszer egy kezdőállapotát, amely tartalmazza a $P_1, \ldots, P_j$ folyamatokat, amelyekben a változók k-nál kevesebb értéket vesznek fel. Ez ellentmond az indukciós feltevésnek.) Ezt a szigorúbb követelményt használva létrehozhatjuk a $P_1, \ldots, P_i$ folyamatok egy alacsony szintű pártatlan végrehajtás sorozatát, amely kiterjeszti α_i-t, és amelyben az közös változó minden k értéke végtelen sokszor fordul elő.

Most létrehozunk egy α rossz végrehajtási sorozatot a következőképpen. Ez α_j-vel kezdődik, amely a $P_1, \ldots, P_j$ folyamatokat a rendszerbe hozza, a rendszerállapotot pedig s_j-be és a változó értéket v_j-ről v_i-re. Utána futtatjuk a $P_1, \ldots, P_i$ folyamatokat a fent leírt módon, de az s_j állapotból az s_i állapot helyett, ezek a folyamatok azt okozzák, hogy az közös változó mind a k értéke végtelen sokszor fordul elő.

[4] Ahogy a 10.8. alfejezetben már bevezettük, ez azt jelenti, hogy az s állapot csak a $P_1, \ldots, P_i$ folyamatok lépéseit használva érhető el.

Emlékezzünk vissza, hogy minden P_m folyamat $(i+1 \leq m \leq j)$ s_m helyi állapotából képes arra, hogy az s_m állapotbeli változóérték végtelen sokszor ismétlődjön. Így a következőképpen fésüljük össze a $P_{i+1}, \ldots, P_j$ folyamatok néhány lépését a $P_1, \ldots, P_i$ folyamatok fő végrehajtási sorozatába: minden olyan pillanatban, amikor a fő végrehajtási sorozatban az közös változó értékét beállítjuk egy v_m értékre $(i+1 \leq m \leq j)$, futtatjuk a P_m folyamat lépéseit addig (de legalább egyet), amíg az közös változó értékét a v_m-nek adja vissza. Ezek a beállítások egy olyan végtelen végrehajtási sorozatot hoznak létre, amely pártatlan minden folyamat számára és amely kizár minden P_m folyamatot, ahol $i+1 \leq m \leq j$. □

A fő gondolat, amelyre a 10.44. tétel bizonyításából emlékezhetünk, az alacsony szinten, pártatlan, rossz végrehajtási sorozatokat végrehajtási részsorozatokkal összefésülve hozzuk létre. Tehát jegyezzük meg, hogy a 10.44. tételből következik, ami első látásra paradoxonnak látszódhat. Ugyanis a létezik a kizárásmentes kölcsönös kizárásnak egy nemtriviális vele járó költsége, még akkor is, ha a modellünk már tartalmaz egy ahhoz közel álló dolgot, amire szükségünk van: pártatlan kizárólagos hozzáférést egy közös változóhoz tetszőleges számítás céljára.

Jegyezzük meg, hogy létezik egy rés a 10.43. tételben szereplő felső korlát és a 10.44. tételben szereplő alsó korlát között. Ennek a résnek a megszüntetése kutatási kérdés.

10.9.4.. Szimulációs bizonyítás

Ezt az alfejezetet az előző alfejezetben bemutatott JegyKK algoritmus helyességi bizonyításának körvonalazásával zárjuk. Bizonyításunk a 8.5.5. szakaszban leírt szimulációs eljárást használja. Már használtuk a szimulációs eljárást néhány algoritmus helyességének megmutatására – például az OptMaxBeállít algoritmusnál – a szinkron modellben, azonban ez az első érdekes használata ennek az eljárásnak aszinkron algoritmusok esetében.

Azt szeretnénk belátni, hogy a JegyKK algoritmus biztosítja ugyanazokat a helyességi feltételeket, amelyeket a SorKK algoritmus: jólformáltság, kölcsönös kizárás, haladás és FIFO tulajdonság a Pr-beli első, helyileg vezérelt eseményre vonatkozóan. Ennek megmutatásából a 10.40. tétel következne. Kiderül, hogy a JegyKK algoritmus megértésének egy jó módja, ha nem a SorKK algoritmussal, hanem egy új VégtelenJegyKK algoritmussal vetjük össze, amely éppen olyan, mint a JegyKK algoritmus, kivéve, hogy jegyek végtelen sorozatát használja ahelyett, hogy modulo n számolna. Ekkor a JegyKK algoritmust úgy tekinthetjük, mint a VégtelenJegyKK csökkentett bonyolultságú változatát. Hogy megkapjuk az új VégtelenJegyKK algoritmust, a JegyKK következők módosításaira van szükség:

10.13. algoritmus. VÉGTELENJEGYKK

Közös változók:
(*következő*, *kiadott*) $\in \mathbb{N} \times \mathbb{N}$, kezdetben $(0, 0)$

P_i **műveletei:**
Ahogy a JEGYKK algoritmusnál

P_i **állapotai:**
jegy $\in \mathbb{N} \cup \{$*üres*$\}$, kezdetben *üres*

P_i **átmenetei:**

`kér`$_i$
Előfeltétel:
p_sz = *kérések*
Hatás:
jegy := *következő*
következő := *következő* + 1
if *jegy* = *kiadott* **then**
p_sz := *elhagy_próba*
else *p_sz* := *vizsgálat*

`visszaállít`$_i$
Előfeltétel:
p_sz = *visszaállítás*
Hatás:
kiadott := *kiadott* + 1
jegy := *üres*
p_sz := *elhagy_kilépés*

Könnyű megmutatni, hogy a VÉGTELENJEGYKK teljesíti mindazokat a tulajdonságokat, amelyeket a JEGYKK algoritmus esetében állítottunk, mivel egy pillanatban csak egy jegy van kiadva és a jegyeket nem lehet újból felhasználni. Ekkor megmutathatjuk a JEGYKK helyességét formálisan a VÉGTELENJEGYKK-ra utalva és a szimulációs módszert használva. Az ötlet az, hogy a két algoritmust egymás mellett futassuk, ezzel bizonyítva, hogy bizonyos erős kapcsolat áll fenn a két végrehajtás között.

Bizonyos invariánsok jól használhatók a VÉGTELENJEGYKK esetében. (Ezek természetesen mindenképp bizonyítandók az algoritmus tulajdonságainak ellenőrzése során.)

10.45. lemma. . *A* VÉGTELENJEGYKK *algoritmus bármely elérhető rendszerállapotában a következők állnak fent:*

1. *egy P_i folyamat pontosan akkor rendelkezik egy nem üres jeggyel, ha $p\_sz_i \in$ {vizsgálat, elhagy_próba, belépés, visszaállítás};*
2. *A nem üres jegy értékei pontosan a [kiadott, következő) intervallum egészei és mindegyikhez pontosan egy folyamat tartozik;*
3. *kiadott $\leq$ következő $\leq$ kiadott + n.*
4. *Ha $p\_sz_i \in$ {elhagy_próba, belépés, kilépés}, akkor $jegy_i$ = kiadott.*

A következő lépés, hogy meghatározzuk az f szimulációs relációt a JEGYKK és VÉGTELENJEGYKK rendszerállapotai között, amikor a két algoritmust a felhasználók azonos halmazával kapcsolták össze. A megfeleltetés egyszerű: legyen

$(s, u) \in f$ (másképp írva $u \in f(s)$), feltéve, hogy a két állapot megegyezik, kivéve, hogy a különféle megfelelő jegykomponensek modulo n kell azonos eredményt adniuk. Az alábbiakban a pont jelölést az adott állapotbeli adott változó értékének jelölésére használjuk.

1. Minden felhasználói állapot megegyezik s-ben és u-ban.
2. Minden i-re $s.p\_sz_i = u.p\_sz_i$.
3. $s.kiadott = u.kiadott \mod n$.
4. $s.következő = u.következő \mod n$.
5. Minden i-re $s.jegy_i = u.jegy_i \mod n$.

Megmutatjuk, hogy f egy szimulációs reláció. Pontosabban: legyen J és V JegyKK, illetve VégtelenJegyKK rendszerek, illetve mindegyik egy kicsit módosított úgy, hogy minden művelet mint külső osztályozott. Megmutatjuk, hogy f egy szimulációs reláció J-ből V-be. A két feltétel, amelyre szükségünk van, a következő:

1. ha s a J egy kezdőállapota, akkor $f(s)$ tartalmazza az V egy kezdőállapotát:
2. ha s a J egy elérhető állapota, $u \in f(s)$ az V egy elérhető állapota és (s, π, s') a J egy átmenete, akkor létezik az V egy (u, π, u') lépése, ahol $u' \in f(s')$.

10.46. lemma. . *f egy szimulációs reláció J-ből V-be.*

Bizonyítás. A fenti két feltételt egyszerűen bizonyíthatjuk. Az első feltételhez, azaz a kezdeti feltételhez, a J egy s kezdőállapota áll a JegyKK algoritmus egyetlen kezdőállapotából és a felhasználók tetszőleges kezdőállapotaiból. Könnyű megmutatni, hogy a VégtelenJegyKK algoritmus egyetlen kezdőállapota a felhasználók azonos kezdőállapotaival együtt az $f(s)$-ben van.

A második feltétel, azaz, a lépésfeltétel esetszétválasztással bizonyítható, az elvégzett művelet típusának megfelelően. A felhasználó bármely helyileg vezérelt művelete pontosan utánozható. Az algoritmus minden helyileg vezérelt műveletére egy J-beli (s, π, s') lépés létezéséből közvetlenül következik, hogy a π azonos művelet az V megfelelő állapotában megengedett, mert az őrfeltétel csak a p_sz értékétől függ. Tehát, minden esetben az u' új állapot a VégtelenJegyKK algoritmusban egyértelműen meghatározott. Már csak azt kell megmutatnunk, hogy $u' \in f(s')$.

De ez is egyszerű. Az egyetlen érdekes eset a $vizsgál_i$ alakú művelet, ahol egy P_i folyamat döntést hoz a $jegy_i = kiadott$ feltétel alapján, hogy Be-be kelle lépnie. Ellenőriznünk kell, hogy a két algoritmus nem hoz különböző döntést. Mivel a JegyKK csak $0, \ldots, n-1$ értékű jegyeket használ, és s-ben és u-ban a megfelelő értékek egyenlők modulo n, az egyetlen lehetőség, hogy a döntés különböző legyen, ha az egyenlőség fennáll a JegyKK-ban, de a VégtelenJegyKK-ban nem. Azaz, a veszély az, hogy a megnövelt jegy értéke modulo n elmossa a különbségeket, amelyek fontosak a VégtelenJegyKK viselkedésének meghatározásához.

De ez nem probléma. Tegyük fel, hogy $s.jegy_i = s.kiadott$. Ekkor $u \in f(s)$-ből következik, hogy $u.jegy_i = u.kiadott \mod n$. A 10.45. lemmában bizonyított

invariánsból következik, hogy $u.kiadott \leq u.jegy_i < u.következő$ és $u.következő \leq u.kiadott + n$. Ezért $u.kiadott \leq u.jegy_i < u.kiadott + n$. Így az kapjuk, hogy $u.jegy_i = u.kiadott$, amit vártunk. □

Hogyan segít a 10.46. lemma a JEGYKK helyességének bizonyításában?

10.40. tétel bizonyításvázlata. A 10.46. lemmából és 8.12. tételből következik, hogy *történetek*$(J) \subseteq$ *történetek*(V). A jólformáltságot, kölcsönös kizárást és a FIFO feltételt ki lehet fejezni mint a történetek tulajdonságai (amikor minden műveletet tartalmaznak, ahogy itt is). Így abból, hogy ez a három feltétel fennáll V-re következik, hogy fennállnak J-re is. Ebből következik, hogy a JEGYKK biztosítja a jólformáltságot, kölcsönös kizárást és a FIFO feltételt.

De nem bizonyítja, hogy a JEGYKK biztosítja a haladást. A haladás feltétele abban különbözik a másik három feltételtől, hogy feltesszük: egy algoritmusnak csak a *pártatlan* végrehajtása áll fenn. Hogy megmutassuk, hogy ez a feltétel átvihető a VÉGTELENJEGYKK-ből JEGYKK-ba, tudni szeretnénk, hogy *pártatlan_ történetek*$(J) \subseteq$ *pártatlan_ történetek*(V). Kiderül, hogy az f szimulációs reláció használható arra is, hogy segítsen bizonyítani ezt a tartalmazást.

Az alapgondolat az, hogy egy szimulációs relációból valójában több következik, mint csupán történetek halmazának tartalmazása – ez valójában egy szoros összefüggést állít fel a két algoritmus *végrehajtási sorozatai* között. Lásd a 8.5.5. alfejezetet egy ilyen összefüggés formális meghatározásához. A jelenlegi helyzetben a 8.5.5. alfejezetben található 8.13. tételből következik, hogy J bármely α végrehajtási sorozatára létezik az V egy α' végrehajtási sorozata, amely megfelel neki a következő nagyon erős értelemben:

1. a műveletek sorrendje α-ban és α'-ben megegyezik;
2. az α és α' azonos helyein az állapotok f relációban vannak.

Egy erős összefüggést kapunk, mert J és V minden művelete külső.

Most bizonyítjuk, hogy *pártatlan_ történetek*$(J) \subseteq$ *pártatlan_ történetek*(V). Legyen $\beta \in$ *pártatlan_ történetek*(J) és legyen α J egy pártatlan végrehajtási sorozata úgy, hogy $\beta =$ *történet*(α). Ekkor a 8.13. tétel szerint létezik az V egy α' megfelelő végrehajtási sorozata, amelyre teljesül a két fenti feltétel. Konkrétabban: α és α' történetei azonosak, mivel $\beta =$ *történet*(α'). Azt állítjuk, hogy α' az V egy pártatlan végrehajtási sorozata.

Két mód létezik, hogy ne legyen pártatlan. Első: létezhet egy P_i folyamat, amely engedélyezett (hogy helyileg vezérelt lépést tegyen) egy bizonyos pontból α'-ben, de ilyen lépés nem fordul elő e pont után α'-ben. Ekkor az erős összefüggésből következik, hogy a P_i folyamat is engedélyezett ugyanebből a pontból α-ban, de ilyen lépés nem fordul elő e pont után α-ban. Ez megsérti az α pártatlanságát, ellentmondás. Második: létezhet egy olyan felhasználói feladat, amely engedélyezett egy bizonyos pontból α'-ből, és még nincs lépés, hogy előforduljon egy folyamat azután a pont után α'-ben. Ismét a megfeleltetésből következik, hogy ugyanaz történik α-ban, amely megsérti α pártatlanságát.

Ebből az következik, hogy α' az V egy pártatlan végrehajtási sorozata, amelyből következik, hogy $\beta \in$ *pártatlan_ történetek*(V). Így *pártatlan_ történetek*$(J) \subseteq$ *pártatlan_ történetek*(V). Mivel a haladás feltétele kifejezhető a pártatlan történetek egy tulajdonságaként (amikor minden műveletet tartalmaz, mint itt is), ebből

az következik, hogy a haladás feltétele átviszi a VÉGTELENJEGYKK-t JEGYKK-ba. □

10.10.. Megjegyzések a fejezethez

A DIJKSTRAKK algoritmus Dijkstra egyik rövid jegyzetében [90] jelent meg. Ez Dekker kétfolyamatú algoritmusának kiterjesztése tetszőleges számú folyamatra. Ezen eredmények előtt nem volt teljesen világos, hogy a feladat megoldható-e csak olvasható/írható közös memóriával. A tulajdonságokon alapuló bizonyítás, amely szerint a DIJKSTRAKK kielégíti a kölcsönös kizárás feltételét, Goldman és Lynch, a közös memóriájú modellről szóló cikkből [141] származik. Dijkstra eredeti jegyzetét Knuth, de Bruijn, valamint Eisenberg és McGuire válaszai [168, 86, 108] követték; valamennyi a korábbi megoldást javította új, magasszintű pártatlansági feltételek és/vagy jobb teljesítménytulajdonságok hozzáadásával.

A PETERSON2FOLY és PETERSONnFOLY algoritmusokat Peterson [238] tervezte. A VERSENY algoritmus a PETERSON2FOLY és Peterson és Fischer [242] versenyprotokolljának egyesítésén alapszik. A mi VERSENY algoritmusunk egyszerűbb, és helyességét könnyebb belátni, mint a [242]-ben szereplő versenyalgoritmusét; hátránya azonban, hogy megosztottan írható változókat használ, míg az eredeti csak kizárólagosan írható változókat használ.

A BURNSKK algoritmus Burnstől [60], a VÁRÓTEREM algoritmus Lamporttól [174] származik. Lamport későbbi dolgozata [180] a kölcsönös kizárás algoritmusainak további javításait tartalmazza. A kölcsönös kizárás problémája megoldásához szükséges regiszterek számának alsó korlátja Burnsnek és Lynchnak [63] köszönhető.

A JEGYKK algoritmust Fischer, Lynch, Burns és Borodin [120, 121] készítette. Burns, Fischer, Jackson, Lynch és Peterson dolgozatában [62] jelent meg az olvasható-módosítható-írható közös memóriával rendelkező korlátozott megkerülésű és kizárásmentes kölcsönös kizárás. Ezek az eredmények Cremers és Hibbard korábbi munkájára [84] építenek. Konkrétan: a PUFFERFŐKK algoritmus teljesen [84] algoritmusán alapul. A [62] másik eredménye – amelyet nem tárgyaltunk ebben a fejezetben – az, hogy a kizárásmentes kölcsönös kizáráshoz a közös memória legalább $\frac{n}{2}$ értéke szükséges, azzal a speciális feltevéssel, hogy a folyamatoknak csak egyetlen haladási állapota van. (Azaz nem őrizhetnek meg az algoritmus előző futásából semmilyen memóriatartalmat.) Cremers és Hibbard [85] is tervezett egy $n+k$ algoritmust annak érdekében, hogy megvalósítsák a belépési szakasz FIFO elérését olvasható-módosítható-írható közös memória használatával.

Manna és Pnueli könyvében [219] a temporális logikáról jó forrás található, amely – ebben a fejezetben és a könyv más részeiben – az élénkség bizonyításainak formalizálására használható.

A k-kizárási feladatot, amelyet a 10-13. gyakorlatban tekintünk át, elsőként Fischer, Lynch, Burns és Borodin [120] határozta meg, később Shavit [261] tanulmányozta. Raynal könyve [249] sok kölcsönös kizárás algoritmus leírását tartalmazza, egyaránt aszinkron közös memóriára és aszinkron hálózati modellre. A

kölcsönös kizárást Ben-Ari [45], valamint Peterson és Silberschatz [262] könyve is tárgyalja.

10.11.. Gyakorlatok

10-1. Tekintsük a kölcsönös kizárás problémájának egy másik meghatározását, amelyet csak az A közös memóriájú rendszer történeteinek alapján határozzuk meg, ahelyett, hogy az A és a felhasználók együttesére hivatkoznánk. Azaz határozzunk meg egy Q történettulajdonságot úgy, hogy *történetek*(Q) a `próbál`, `belép`, `kilép` és `halad` műveletek sorozatainak halmaza, és teljesíti a következő feltételeket:

(a) β-ban tetszőleges i-re nem a rendszer sérti meg először a jólformáltságot;
(b) ha β jólformált minden i-re, akkor β nem tartalmaz két `belép` eseményt úgy, hogy nincs közte `kilép` esemény;
(c) ha beta jólformált minden i-re, akkor a következők állnak fenn:

   i. Ha β-ban valamely pontnál valamely folyamat utolsó eseménye `próbál`, és egyik folyamat utolsó eseménye sem `belép`, akkor van egy későbbi `belép` esemény;
   ii. Ha valamely pontnál β-ban valamely folyamat utolsó eseménye `kilép`, akkor van egy későbbi `halad` esemény.

Bizonyítsuk be, ha *pártatlan_ történetek*(A) $\subseteq$ *történetek*(Q), akkor A és a felhasználók tetszőleges csoportja együttesen megfelelnek a 10.2. alfejezetben adott kölcsönös kizárási feladat meghatározásának.

10-2. Írjuk le a DijkstraKK algoritmus egy pártatlan végrehajtási sorozatát, amelyben egy konkrét folyamat kizárt.

10-3. Mutassuk meg, hogy a DijkstraKK algoritmus második fázisa (ahol a *jelző* értéke 2-re emelkedik, és a többi folyamat *jelző* változóját már megvizsgálták) szükséges a feladat helyes megoldásához.

10-4. Részletezzük a DijkstraKK algoritmus kölcsönös kizárásának induktív bizonyítását.

10-5. Gondoljuk át a DijkstraKK időanalízisét attól az időponttól, amikor valamely felhasználó Pr-ben van és nincs felhasználó Be-ben, addig a pontig, amikor valamely felhasználó belép Be-be.

(a) Finomítsuk az analízist, hogy $k_1nl + k_2l$ alakban fejezhesse ki a korlátot, ahol k_1 és k_2 állandók. Próbáljuk k_1-et és k_2-t olyan kicsire választani, amennyire csak lehetséges.
(b) Hozzuk létre a DijkstraKK egy végrehajtási sorozatát, amelyben a korlát akkora, amekkorát csak meg tud adni; próbáljuk összemérni a kiszámolt felső határral.

10-6. A kizárásmentesség feltételének olyan algoritmusok esetében van jelentősége, amelyek biztosítják a jólformáltságot, de nem feltétlenül biztosítják a

kölcsönös kizárást vagy a haladást. Bizonyítsuk kellő alapossággal, hogy ha egy algoritmus biztosítja a jólformáltságot és kizárásmentességet (a felhasználók minden halmazára), akkor biztosítja a haladást is (a felhasználók minden halmazára).

10-7. Módosítsuk úgy a PETERSON2FOLY algoritmus folyamatait úgy, hogy a `jelző_ellenőriz` és `váltás_ellenőriz` műveletek ne feltétlenül pontos váltakozással hajtódjanak végre, de teljesüljön valamilyen enyhébb tulajdonság. Győződjünk meg arról, hogy a kapott algoritmus még mindig kizárásmentes kölcsönös kizárás algoritmus. Bizonyítsuk, hogy a módosított algoritmus helyes, és analizáljuk időbonyolultságát.

10-8. Tervezzünk egy kétfolyamatú kizárásmentes kölcsönös kizárás algoritmust, amely csak *kizárólagosan írható*/megosztottan olvasható írható/olvasható regisztereket használ. Bizonyítsuk algoritmusának helyességét, lehetőleg invariánsok bevezetésével. (*Útmutatás.* Ha megakadunk, nézzük meg a [242] kétfolyamatú megoldását. Bár az invariáns bizonyítását egyedül kell elvégeznünk.)

10-9. Bizonyítsuk a 10.5.3. állítást.

10-10. Bizonyítsuk a 10.5.4. állítást.

10-11. Ismét gondoljuk át a PETERSONnFOLY algoritmus 10.16. tételben bizonyított időkorlátját. Éles? Vagy mutassunk egy végrehajtási sorozatot, amelyben az exponenciális magatartás valóban megvalósul, vagy ellenkező esetben, adjunk egy finomabb analízist egy kisebb bonyolultsági korláttal.

10-12. Biztosítja-e a PETERSONnFOLY algoritmus a korlátozott megkerülést? Bizonyítsuk, hogy igen, vagy adjunk ellenpéldát.

10-13. Módosítsuk a PETERSONnFOLY algoritmust, hogy a *k-kizárás* feladat ($2 \leq k \leq n$) egy megoldását kapja. Ez a feladat megengedi k folyamatnak, hogy egyszerre tartózkodjanak a belépési szakaszban. Formálisan, a kölcsönös kizárás feltételét módosítsuk úgy, hogy k-nál több felhasználó ne lehessen egyszerre Be-ben. A próba szakasz haladás feltétele is módosul, így azt mondja, hogy ha létezik legalább egy felhasználó Pr-ben, és legfeljebb $k-1$ felhasználó Be-ben, akkor valamelyik felhasználó végül belép Be-be. Bizonyítsuk, hogy algoritmusa helyesen működik. Figyelmesen mondjuk ki a magasszintű pártatlansági feltételt, amelyet az algoritmusunk teljesít.

10-14.

(a) Írjuk át a VERSENY algoritmust előfeltétel/hatás formájúra.

(b) Az átírt forma jelöléseivel gondosan határozzuk meg a „győztes” és „versenyző” fogalmakat.

(c) Bizonyítsuk a 10.5.7. állítást. (*Útmutatás.* Tegyük erősebbé azzal, hogy hozzáveszünk néhány információt arról, mi történik, amikor egy folyamat a waitfor ciklusban van, miután a folyamat észrevette, hogy néhány versenytársa *jelző* változója szigorúan kisebb k-nál. Ezután indukcióval bizonyítsuk a szigorított invariánst az eredetivel együtt.)

(d) Fejezzük be a bizonyítást, hogy a VERSENY algoritmus biztosítja a kölcsönös kizárást.

10-15. Mutassuk meg, hogy a VERSENY algoritmus hogyan alkalmazható n folyamat esetében, ahol n nem 2-hatvány. Mi történik az időbonyolultsággal?

10-16. *Kutatási kérdés.* Találjuk ki a VERSENY protokoll egy másik változatát, amely megosztottan írható/megosztottan olvasható regiszterek helyett kizárólagosan írható/megosztottan olvasható regisztereket használ. Adjunk teljes helyességbizonyítást és elemzést.

10-17. Mi történik a BURNSKK algoritmus viselkedésével, ha a második ciklust elhagyjuk. Vagy bizonyítsuk, hogy ez is megoldja a kölcsönös kizárás problémáját, vagy mutassunk egy végrehajtási sorozatot ellenpéldaként.

10-18. Adjunk egy tulajdonságokon alapuló bizonyítást, amely megmutatja, hogy a BURNSKK algoritmus teljesíti a kölcsönös kizárás feltételét. Ehhez az algoritmust át kell írnunk előfeltétel/hatás formájába, és pontosan meg kell határoznunk a változókat, amelyek nyomon követik az ellenőrzött folyamatokat a ciklusokban.

10-19. Mutassuk be a BURNSKK algoritmusnak egy olyan alacsony szintű pártatlan végrehajtási sorozatát, amelyben egy adott folyamat kizárt.

10-20. Hajtsunk végre egy időelemzést a haladási feltételre a BURNSKK algoritmus esetében. Azaz tegyük fel, hogy a belépési szakasz idejére és a folyamat lépésidejére a c és l felső korlátok; és az időt attól vegyük figyelembe, amikor egy folyamat Pr-ben van és nincs folyamat Be-ben, és addig vegyük figyelembe, amíg egy folyamat belép Be-be.

(a) Bizonyítsunk egy felső korlátot erre az időtartamra.

(b) Mutassunk egy olyan végrehajtási sorozatot, amelyben ez az időtartam a lehető legnagyobb.

Próbáljuk meg a lehető legközelebb megadni az (a) és (b) feladatban szereplő nem korlátosak.

10-21. Adjuk meg a VÁRÓTEREM algoritmus egy olyan végrehajtási sorozatát, amelyben a *sorszám* regiszterek által felvett értékek nem korlátosak.

10-22. Miért hibázik a VÁRÓTEREM algoritmus, ha az egész számokat kicseréljük (valamilyen nagyon nagy b-re) mod b-vel vett számokkal? Adjunk egy speciális végrehajtási sorozatot ellenpéldaként.

10-23. Írjuk át a VÁRÓTEREM algoritmust előfeltétel/hatás formára. Amíg ezen dolgozunk, próbáljuk némileg általánosítani az algoritmust azzal, hogy a műveletek sorrendjében a lehető legtöbb nemdeterminisztikusságot engedi meg (Az előfeltétel/hatás jelöléssel általában könnyebb a nemdeterminisztikuságot kifejezni, mint a gyakran használt szekvenciális vezérlésleírással.) Adjunk egy megerősítő bizonyítást a kölcsönös kizárás feltételére az általánosított algoritmus esetében.

10-24. Bizonyítsuk, hogy a VÁRÓTEREM algoritmus még a következő nagyon gyenge modellben is helyesen működik. Tegyük fel, hogy az olvasások már nem pillanatnyiak, hanem van időtartamuk. Tegyük fel, hogy a közös regisztereknek csak a *biztonságuk* biztosított, azaz helyes értéket csak párhuzamos olvasás és írás

nélkül adnak. Amikor egy olvasás átlapol egy írást, az olvasás bármilyen értéket visszaadhat.

10-25. Bizonyítsuk, hogy a VÁRÓTEREM algoritmus biztosítja a korlátozott megkerülést.

10-26. Vajon Burns kölcsönös kizárás algoritmusa akkor is működik, ha az közös regiszterek mindegyike biztonságos regiszter (a 10.25. gyakorlatbeli meghatározás szerint)? Miért vagy miért nem?

10-27. Tegyük fel, hogy a VÁRÓTEREM algoritmus csak az azonnali elérésű közös memória esetében kell hogy működjön, nem szükséges a biztonságos regisztereket használó általánosabb modell. Adjuk meg az algoritmus egy egyszerűsített változatát, amely biztosítja ugyanazt a kölcsönös kizárást és magasszintű pártatlansági feltételeket, mint az eredeti VÁRÓTEREM algoritmus. Bizonyítsa állítását.

10-28. Részletezzük a VÁRÓTEREM algoritmus bonyolultsági elemzését, amelyet a 10.7. alfejezet végén vázoltunk.

10-29. Adjunk pontos kódot olyan U_i speciális automatára, amely bemutatja az összes U_i felhasználóra vonatkozó nemdeterminizmust, azaz képesnek kell lennie arra, hogy bármikor végrehajtsa $\texttt{próbál}_i$ és $\texttt{kilép}_i$ műveleteit, vagy sohasem, és csak a jólformáltság feltételének kell megfelelniük. Az új automatáknak rendelkezniük kell azzal a tulajdonsággal, hogy minden i-re és tetszőleges másik V_i felhasználó automatára teljesül: *pártatlan_ történetek*$(V_i) \subseteq$ *pártatlan_ történetek*(U_i).

10-30. Adjunk gondos bizonyítást a 10.36. lemmára.

10-31. *Kutatási kérdés.* Hogyan hat a 10.8. fejezet eredményeire, ha – a kölcsönös kizárás problémája helyett – a következőket tekintjük:

(a) a k-kizárás problémát ($2 \leq k \leq n$) a 10.13. gyakorlat meghatározása szerint;

(b) a k-kizárás feladat egy gyengébb változatát, amely a fenti módon használja a módosított kölcsönös kizárás feltételét, de megtartja az eredeti haladási feltételt.

10-32. A Zöldfülűek Számítástechnikai Rt. programozói a következő algoritmust tervezték n folyamatú kölcsönös kizárásra. Azt állítják, hogy algoritmusuk biztosítja a kölcsönös kizárást és haladást, de nem állítanak egyetlen magasszintű pártatlansági feltételt sem.

10.14. algoritmus. Az A

közös változók:
$x \in \{1, \ldots, n\}$, kezdetben tetszőleges
$y \in \{0, 1\}$, kezdetben 0

P_i **átmenetei:**

```
    ** Haladási szakasz **

    próbál_i
L:  x := i
    if y ≠ 0 then goto L
    y := 1
    if x ≠ i then goto L
    belép_i

    ** Belépési szakasz **

    kilép_i
    y := 0
    halad_i
```

Rendelkezik ez a protokoll a két megkövetelt tulajdonsággal? Vagy bizonyítsuk, hogy igen, vagy adjunk pontos végrehajtási sorozatot ellenpéldaként annak bizonyítására, hogy nem.

10-33. *Kutatási kérdés.* Gondoljuk át a haladási feltételt egy olyan általánosított *k-párhuzamos haladás* esetében, amely csak akkor követeli meg a haladást, ha nincs k-nál több felhasználó párhuzamosan *Ha*-n kívül:

> **k-párhuzamos haladás.**
>
> Bármely *pártatlan végrehajtási sorozatban*, amelyben soha nem létezik k-nál több felhasználó *Ha*-en kívül egyszerre:
>
> (a) (a próba szakasz k-párhuzamos haladása) ha legalább egy felhasználó *Pr*-ben van és nincs felhasználó *Be*-ben, akkor egy adott későbbi pontban valamelyik felhasználó belép *Be*-be;
> (b) (a kilépési szakasz k-párhuzamos haladása) ha legalább egy felhasználó *Ki*-ben van, akkor egy adott későbbi pontban adott felhasználó belép *Ha*-be.

Adjuk meg a lehető legjobb felső és alsó korlátot az osztott olvasható/írható változók számára, amelyek szükségesek a jólformáltság, kölcsönös kizárás és k-párhuzamos haladás megvalósításához.

10-34. Tervezzünk egy jó kölcsönös kizárás algoritmust egy olvasható/írható közös memória modellre, amely egy kicsit különbözik az ebben a fejezetben használt

modelltől. Ebben az új modellben létezik egy további *felügyelő folyamat* a szokásos $P_1, \ldots, P_n$ „ügynök" folyamatok mellett, amely bármikor tehet lépéseket. A modellnek kizárólagosan írható/megosztottan olvasható közös változókat kell használnia. Bizonyítsuk algoritmusunk helyességét, és analizáljuk bonyolultságát.

10-35. Bizonyítsuk a SorKK algoritmus valamennyi megkövetelt tulajdonságát. A bizonyítást a rendszer fő invariánsainak meghatározásával és bizonyításával kezdjük, majd használjuk ezeket fel a kölcsönös kizárás tulajdonság bizonyításához. A haladás szokás szerint indirekt úton bizonyítható. A FIFO feltétel fennállásának bizonyításához egyedi érvelés használható Pontosabban, a bizonyításainknak a 10.39. tételt kell eredményezniük.

10-36. A VÁRÓTEREM algoritmusban nem lehetséges tovább csökkenteni a nem korlátos jegyek értékeit, úgy, hogy valamilyen modulo egész számmal számolunk; azonban ez mégis működik a JegyKK algoritmusban. Indokoljuk meg a különbséget.

10-37. Tekintsük át a PufferFőKK algoritmust.

(a) Írjunk előfeltétel/hatás kódot a felügyelővel rendelkező algoritmushoz a felügyelő és az „ügynök" folyamatokra. Bizonyítsuk az algoritmus helyességét.

(b) Csináljuk ugyanazt, mint az algoritmus utolsó változatában, de most felügyelő nélkül. (*Útmutatás.* Megpróbálhatjuk összekapcsolni ezt a felügyelővel rendelkező változattal szimulációs bizonyítást használva.)

10-38. Tervezzünk egy új algoritmust, amely megoldja a kölcsönös kizárást egyetlen olvasható-módosítható-írható közös változót használva, és amely FIFO tulajdonságú (a próba szakasz első helyileg vezérelt lépésére vonatkoztatva). Próbáljuk minimalizálni az közös változó által felvett értékek számát. Felvehetünk még egy dedikált felügyelő folyamatot. (*Útmutatás.* $n+k$ elérhető egy kis k konstanssal.)

10-39. Mutassuk meg, hogy a VÉGREHAJT algoritmus nem biztosítja a korlátozott megkerülést.

10-40. *Kutatási kérdés.* Írjunk kódot a VÉGREHAJT algoritmushoz és bizonyítsuk helyességét. Bizonyításunk alapozható-e formálisan a PufferFőKK algoritmus helyességére? Használható-e szimulációs bizonyítás?

10-41. Adjunk felső határt arra az időre, ami egy folyamat Pr-be való belépésétől Be-be való belépéséig tart:

(a) a PufferFőKK algoritmus esetében;

(b) a VÉGREHAJT algoritmus esetében.

Elemzésünket ne a kódra, hanem a neki megfelelő b/k automatára alapozzuk. Érdemes először az algoritmusok előfeltétel/hatás kódját elkészíteni.

10-42. Miért nem általánosítottuk a VÉGREHAJT algoritmust úgy, hogy csak $\frac{n}{3}$ értéket engedjen a változóknak?

10-43. *Kutatási kérdés.* Szüntessük meg a 10.43. és 10.44. tételben szereplő felső és alsó korlát közötti eltérést. (*Útmutatás.* Egy részeredmény megtalálható [62]-ben.)

10-44. Részletezzük a 10.46. lemma bizonyítását.

10-45. *Kutatási kérdés.* Alakítsuk át az ebben a fejezetben található élénkségi feltételek (haladási és kizárásmentesség) bizonyításait úgy, hogy formális temporális logikát használunk.

11. fejezet

Erőforrások hozzárendelése

A 10. fejezetben a kölcsönös kizárás problémájával, egy elvont erőforrás-hozzárendelési problémával foglalkoztunk, melyben egyetlen, nem megosztható erőforrást rendelünk hozzá egymással versengő felhasználókhoz. Ebben a fejezetben általánosítjuk a problémát, mégpedig oly módon, hogy az egyetlen erőforrást helyettesítjük több erőforrással. Ez az általánosítás olyan alkalmazások modellezésénél hasznos, melyeknek végrehajtásuk során több, különböző erőforrásra, például egy nyomtatóra, egy adatbázisra és egy hálózati csatlakozóra van szükségük.

Az erőforrások elosztásának problémáját az általunk itt vizsgáltnál általánosabban is meg lehet fogalmazni.

1. Nem vizsgáljuk (néhány általános meghatározást és gyakorlatot kivéve) azt a lehetőséget, hogy egy felhasználó – egy bizonyos erőforrás lefoglalása helyett – hajlandó lehet egy alternatív erőforráshalmaz bármelyik elemét igénybe venni. Például egy felhasználó kérhet egyszerűen „valamilyen nyomtatót", ahelyett, hogy egy pontosan megadott nyomtatót kérne.
2. Nem vizsgáljuk azt a lehetőséget, hogy egy erőforrás megosztható is lehet. Például egyedi adatobjektumok egy adatbázisban olyan erőforrásoknak tekinthetők, melyeket az adatbázis-műveletekhez rendelünk hozzá. Ebben az esetben a megosztás bizonyos fajtái jellemzően megengedettek; például ha van két műveletünk, amelyek csak olvasni akarják ugyanazt az objektumot, akkor megengedhető, hogy egyidőben hozzáférjenek.

A továbbiakban először definiáljuk az általános erőforrás-hozzárendelési problémát, valamint az étkező filozófusok problémát, mint ennek egy érdekes, speciális esetét. Ezután megadunk néhány jellemző megoldást. Az utolsó megoldásunk egy véletlenített protokoll, az első aszinkron keretrendszerű véletlenített protokoll példánk.

11.1.. A feladat

Ebben az alfejezetben először megadunk néhány módszert a felhasználók közötti konfliktusok leírására. Ezután megadjuk, hogyan használhatók ezek a módszerek

erőforrás-hozzárendelési feladatok leírására. Végül pedig definiáljuk az étkező filozófusok problémát.

11.1.1.. Explicit erőforrás-leírások és kizárási leírások

A kölcsönös kizárás problémáját két eltérő szemszögből közelíthetjük meg: mint egy egyértelműen megadott erőforrás lefoglalásának problémáját, vagy mint egy olyan problémát, melyben azt kell biztosítanunk, hogy egyszerre csak egy felhasználó lehessen a kritikus szakaszában. Az általános erőforrás-hozzárendelési problémát is megközelíthetjük ugyanebből a két szemszögből. Így definiáljuk mind az *explicit erőforrás-leírásokat*, mind a *kizárási leírásokat*, mint a felhasználók közötti konfliktusok leírásának alternatív módszereit.

Egy n felhasználós $\mathcal{R}$ *explicit erőforrás-leírás* a következőkből áll.

1. R: objektumok egy univerzális, véges halmaza (ezek az objektumok lesznek az *erőforrások*).
2. Minden i-re, $1 \leq i \leq n$, egy $R_i \subseteq R$ halmaz.

Az R_i halmaz azokat az erőforrásokat tartalmazza, amelyekre az U_i felhasználónak szüksége van ahhoz, hogy elvégezhesse a feladatát. Azt mondjuk, hogy az U_i és U_j felhasználók egy adott explicit erőforrás-leírás szempontjából *konfliktusban vannak*, ha van olyan erőforrás, amelyre mindkettőjüknek szüksége van, azaz ha $R_i \cap R_j \neq 0$.

11.1.1. példa. Explicit erőforrás-leírás

Tekintsünk egy négy felhasználós $(U_1, ..., U_4)$ explicit erőforrás-leírást. Az erőforrások R halmaza legyen $\{r(1), r(2), r(3), r(4)\}$. A négy felhasználó erőforrásigénye legyen a következő:

U_1 : $\{r(1), r(2)\}$,
U_2 : $\{r(1), r(3)\}$,
U_3 : $\{r(2), r(4)\}$,
U_4 : $\{r(3), r(4)\}$.

Tehát az U_1 felhasználónak az $r(1)$ és az $r(2)$ erőforrásokhoz való kizárólagos hozzáférésre van szüksége feladata elvégzéséhez, az U_2 felhasználónak az $r(1)$ és $r(3)$ erőforrásokhoz való hozzáférésre stb. Láthatjuk, hogy az U_1 és U_2 felhasználók konfliktusban vannak egymással, mint ahogy az U_1 és U_3, az U_2 és U_4, és az U_3 és U_4 felhasználók is.

A másik lehetséges módszer, a *kizárási leírás* egyáltalán nem foglalkozik az erőforrásokkal, ehelyett a leírást úgynevezett „rossz halmazok" egy $\mathcal{E}$ gyűjteményével adjuk meg. Ezek a „rossz halmazok" azon felhasználók indexeit tartalmazzák, amelyeknek nincs engedélyezve, hogy egymással egyidejűleg működjenek. A kizárási leírással szemben csak egy megkötésünk van, mégpedig az, hogy a „rossz halmazok" struktúrája zárt legyen a halmaz tartalmazásra. Azaz, ha egy adott E „rossz" halmaz egy $\mathcal{E}$ kizárási leírás eleme, akkor minden olyan halmaz is, amelynek E részhalmaza, $\mathcal{E}$-be tartozik.

11.1.2. példa. Kizárási leírás

A kölcsönös kizárás feltétele leírható a következő kizárási leírással: $\mathcal{E} = \{E \subseteq \{1, ..., n\} : \mid E \mid > 1\}$.

11.1.3. példa. További kizárási leírás

A k-kizárás feltétele (amelyben mindig legfeljebb k felhasználó lehet egyszerre a kritikus szakaszában) megadható az alábbi kizárási leírással: $\mathcal{E} = \{E \subseteq \{1, ..., n\} : \mid E \mid > k\}$. A k-kizárás fogalmát a 10-13. gyakorlatban vezettük be.

11.1.4. példa. Egy további kizárási leírás

Négy felhasználó esetében tekintsük azt az $\mathcal{E}$ kizárási leírást, amely a következő kételemű halmazokból áll: $\{1,2\}$, $\{1,3\}$, $\{2,4\}$ és $\{3,4\}$, valamint az összes olyan halmazból, amelynek ezen kételemű halmazok valamelyike a része[1]. Ebben a kizárási leírásban az U_1 felhasználó nem zárja ki az U_4 felhasználót, és U_2 sem zárja ki U_3-at. Ez azt jelenti, hogy az U_1 és U_4 felhasználó egyidőben is végezheti a feladatát, csakúgy, mint az U_2 és U_3 felhasználó.

Figyeljük meg, hogy bármely explicit erőforrás-leírás alapján készíthetünk egy kizárási leírást, amely megegyezik vele abban, hogy ugyanazon felhasználóknak engedi meg, hogy egymással egyidejűleg dolgozzanak. Ez a kizárási leírás pontosan azokból a halmazokból áll, amelyek tartalmaznak legalább két olyan felhasználót, melyeknek van közös erőforrásigényük.

11.1.5. példa. Megfelelő leírások

A 11.1.1. példában szereplő explicit erőforrás-leírásnak megfelelő kizárási leírás a következő kételemű halmazokból áll: $\{1,2\}$, $\{1,3\}$, $\{2,4\}$ és $\{3,4\}$, valamint az összes olyan halmazból, amelynek ezen kételemű halmazok valamelyike a része.

Az viszont nem igaz, hogy minden kizárási leíráshoz létezik neki megfelelő explicit erőforrás-leírás. Ennek bizonyítását meghagyjuk gyakorlatnak (lásd 11-1. gyakorlat).

[1] Véletlen egybeesés, hogy a halmazok (ha csak az indexeket tekintjük) azonosak a 11.1.1. példában szereplő erőforrásigényeket tartalmazó halmazokkal. Szeretnénk felhívni a figyelmet, hogy abban a példában az erőforrások indexei szerepelnek, míg itt a felhasználók indexei. Ugyanez a megjegyzés vonatkozik a 11.1.5. példában található, felhasználói indexeket tartalmazó halmazokra is. *A fordító.*

11.1.2.. Az erőforrás-hozzárendelési feladat

A továbbiakban leírjuk, hogy a közös memóriájú rendszerek esetében hogyan használhatók fel az explicit erőforrás-leírások és a kizárási leírások az erőforrás-hozzárendelési feladat megoldásánál. Hogy konkrétak legyünk, tekintsünk egy rögzített $\mathcal{E}$ kizárási leírást (amely származhat akár egy explicit erőforrás-leírásból is).

A felépítés pontosan megegyezik azzal, amit a 10. fejezetben a kölcsönös kizárás problémájánál használtunk – felhasználói automaták és egy közös memóriájú rendszerautomata együttese (lásd a 10.4. ábrát). A felhasználói automaták működése ismét egy ciklussal modellezhető, amely a *haladás* (*Ha*), *próba* (*Pr*), *kritikus* (*Be*) és *kilépés* (*Ki*) szakaszokból áll, ahogy a 10.2. ábra bemutatja. Az U_i felhasználó és a közös memóriájú rendszer közötti kölcsönhatások sorozata *jólformált* az U_i felhasználó számára, ha működésében betartja ezt a ciklikus rendet.

A jólformáltság feltétele az összetett rendszerben ugyanaz, mint a korábbiakban.

Jólformáltság. Bármely végrehajtásban és minden i-re az U_i felhasználó és az A rendszer közötti kölcsönhatást leíró részsorozat jólformált i-re.

A kölcsönös kizárás feltételét most az általánosabb kizárási feltétellel helyettesítjük.

Kizárás. Nincs elérhető rendszerállapot, amelyben az a halmaz, amely az adott pillanatban a kritikus szakaszukban levő felhasználókat tartalmazza, eleme lenne $\mathcal{E}$-nak.

A haladási feltétel nem változik a korábbiakhoz képest.

Haladás. Egy *pártatlan végrehajtás* bármely pontján érvényes a következő.

1. (Haladás a *próba* szakaszra.) Ha van legalább egy felhasználó a *Pr* szakaszban és nincs felhasználó a *Be* szakaszban, akkor a végrehajtás valamely későbbi pontján valamelyik felhasználó belép a *Be* szakaszba.
2. (Haladás a *kilépés* szakaszra.) Ha van legalább egy felhasználó a *Ki* szakaszban, akkor a végrehajtás valamely későbbi pontján valamelyik felhasználó belép a *Ha* szakaszba.

Azt mondjuk, hogy egy A közös memóriájú rendszer *a felhasználók egy adott csoportjára megoldja az általános erőforrás-hozzárendelési problémát*, ha ezekkel a felhasználókkal biztosítja a jólformáltsági, a kizárási és a haladási feltétel teljesülését. Továbbá azt mondjuk, hogy egy A rendszer *megoldja az általános erőforrás-hozzárendelési problémát*, ha a felhasználók minden csoportjára megoldja.

A *próba* szakaszra vonatkozó haladási feltétel gyengébb, mint amilyen a jelenlegi kertek között lehetne. Az általános erőforrás-hozzárendelési problémánál azt is szeretnénk, hogy azok a felhasználók, melyek nincsenek egymással konfliktusban, ne akadályozhassák meg egymást a kritikus szakaszba való belépésben,

még akkor sem, ha véglegesen lefoglalják az erőforrásokat. Nem ismerünk jó módszert arra, hogy kizárási leírással leírjunk ilyen feltételt, explicit erőforrás-leírás segítségével azonban legalább a következő feltételeket kifejezhetjük.

Független haladás. Egy *pártatlan végrehajtás* bármely pontján érvényes a következő.

1. (Független haladás a *próba* szakaszra.) Ha az U_i felhasználó a Pr szakaszban és minden vele konfliktusban lévő felhasználó a Ha szakaszban van, akkor a végrehajtás valamely későbbi pontján vagy U_i lép be a Be szakaszba, vagy valamelyik vele konfliktusban lévő felhasználó lép be a Pr szakaszba.
2. (Független haladás a *kilépés* szakaszra.) Ha az U_i felhasználó a Ki szakaszban és minden vele konfliktusban lévő felhasználó a Ha szakaszban van, akkor a végrehajtás valamely későbbi pontján vagy U_i lép be a Ha szakaszba, vagy valamelyik vele konfliktusban lévő felhasználó lép be a Pr szakaszba.

A magas szintű pártatlansági feltételekhez a kölcsönös kizárás problémájánál leírtakhoz hasonló kizárásmentességi és időkorlát feltételek is értelmezhetők az általános erőforrás-hozzárendelési problémához. A korlátozott megkerülés feltételével itt nem foglalkozunk. Most ezen tulajdonságok közötti néhány egyszerű összefüggés következik. (Vesd össze a 10.9. tétellel és a 10-6. gyakorlattal.)

11.1. lemma. .

1. *Ha egy általános erőforrás-hozzárendelési algoritmusnak van egy b időkorlátja, akkor kizárásmentes.*
2. *Ha egy algoritmus az ebben a fejezetben tárgyalt modellben garantálja a jólformáltságot és a kizárásmentességet, akkor garantálja a haladást is.*

Bizonyítás. A bizonyítást meghagyjuk gyakorlatnak (lásd 11-3. gyakorlat). □

Történet tulajdonságok.. Ahogyan a 10. fejezetben a hasonló tulajdonságokra tettük, úgy ekvivalensen itt is kifejezhetjük a jólformáltság, kölcsönös kizárás, haladás, független haladás és a kizárásmentesség feltételeit történet tulajdonságokkal. Ezen P történet tulajdonságok mindegyikének *próba*, *kritikus*, *kilépés* és *haladás* kimenetből áll a lenyomata (és nem tartalmaz bemenetet). Az összetett rendszer külső műveletei is pontosan ezek a műveletek, és minden esetben az az elvárásunk, hogy az egyesített rendszer minden helyes története benne legyen *történetek(P)*-ben.

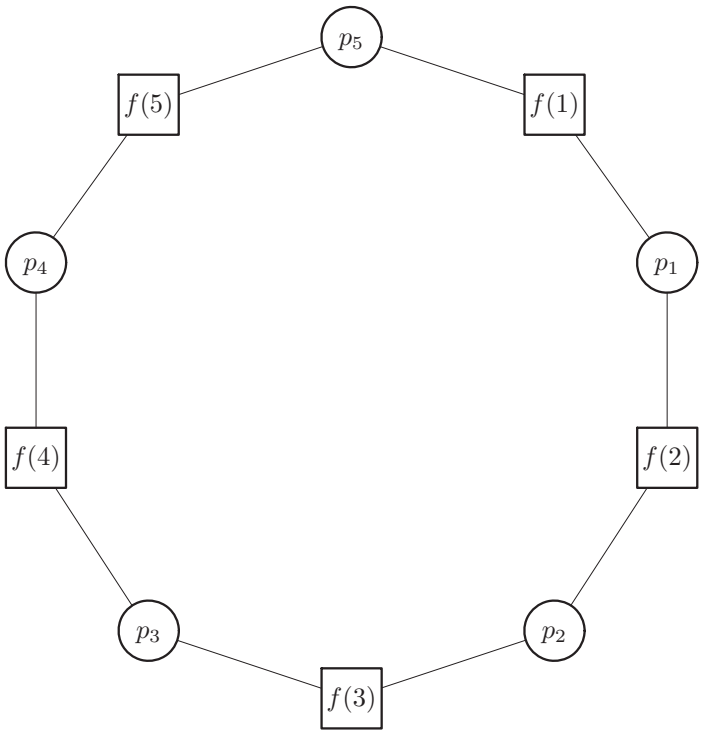
Folyamatok tevékenységének korlátozása.. Ahogyan a 10. fejezetben, úgy ebben a fejezetben is feltesszük, hogy egy folyamatnak lehet helyileg vezérelt művelete, de a közös memóriájú rendszerben ez a művelet csak akkor megengedett, ha a felhasználója a próba vagy a kilépés szakaszban van. Ennek következtében az folyamatok csak addig lehetnek aktívan lefoglalva a protokoll végrehajtásában, amíg van aktív igényük.

11.1.3.. Az étkező filozófusok feladat

Az *étkező filozófusok feladat*, amely az osztott rendszerek elméletének egyik legjobban ismert feladata, az általános erőforrás-hozzárendelési problémának egy egyszerű speciális esete. Általában egy explicit erőforrás-leírás kikötéseivel szokták megadni.

A problémát hagyományosan a következő vázlatos forgatókönyvvel írják le. Van n filozófus (a felhasználók), akik egy kerek asztal körül ülnek. A filozófusok többnyire gondolkodnak (ez felel meg a *Ha* szakasznak). Bármely két szomszédos filozófus között van egy villa (ez lesz az erőforrás). Időről időre valamelyik filozófus megéhezhet (belép a *Pr* szakaszba) és megpróbál enni (belépni a *Be* szakaszba). Ahhoz, hogy étkezhessen, szüksége van mindkét mellette fekvő villa kizárólagos használatára. Miután evett, lemond a két villáról, visszateszi azokat az asztalra (azaz végrehajt egy *Ki* kilépési protokollt), és visszatér a gondolkodáshoz (a *Ha* szakaszhoz).

Minden p_i filozófusra a tőle jobbra (azaz az óramutató járásával ellentétes irányban) levő villát $f(i)$-vel, míg a tőle balra (azaz az óramutató járásával megegyező irányban) lévő villát $f(i+1)$-gyel jelöljük. (Szokás szerint az összeadást modulo n értve, azaz n azonos lesz 0-val.) A filozófusok és a villák elrendezésére n=5 esetében lásd a 11.1. ábrát.



11.1.. ábra. Az étkező filozófusok feladat (n=5).

Formális modellünkben minden filozófushoz egy felhasználó és egy ágens eljárás tartozik. Szokás szerint a felhasználó meghatározza, mikor igényli és mikor adja vissza az erőforrásokat, és az ágens eljárás végrehajtja az algoritmust.

Az n étkező filozófus kizárási leírása az $\{\{i, i+1\} : 1 \leq i \leq n\}$ kételemű halmazokat tartalmazza, valamint az összes olyan halmazt, amelynek ezen kételemű halmazok valamelyike része.

11.1.4.. A megoldások korlátozott formája

Minden megoldás, amivel ebben a fejezetben foglalkozunk, sajátos formájú: minden erőforráshoz pontosan egy olvasható-módosítható-írható közös változó tartozik, amely csak azon eljárások számára elérhető, melyek felhasználói igényelték a megfelelő erőforrást.

Az étkező filozófusok feladat megoldásainak szerkezetét ebben a korlátozott formában mutatja be a 11.2. ábra. Vegyük észre, hogy a diagram nagyon hasonló a 11.1. ábra diagramjához; az új diagram tartalmazza az U_i felhasználókat, a folyamatokat pedig az indexeikkel jelöljük. A közös változók pontosan megfelelnek az $f(1)$, ..., $f(5)$ villáknak. Figyeljük meg, hogy minden P_i folyamat az $f(i)$ és $f(i+1)$ villa változókhoz fér hozzá.

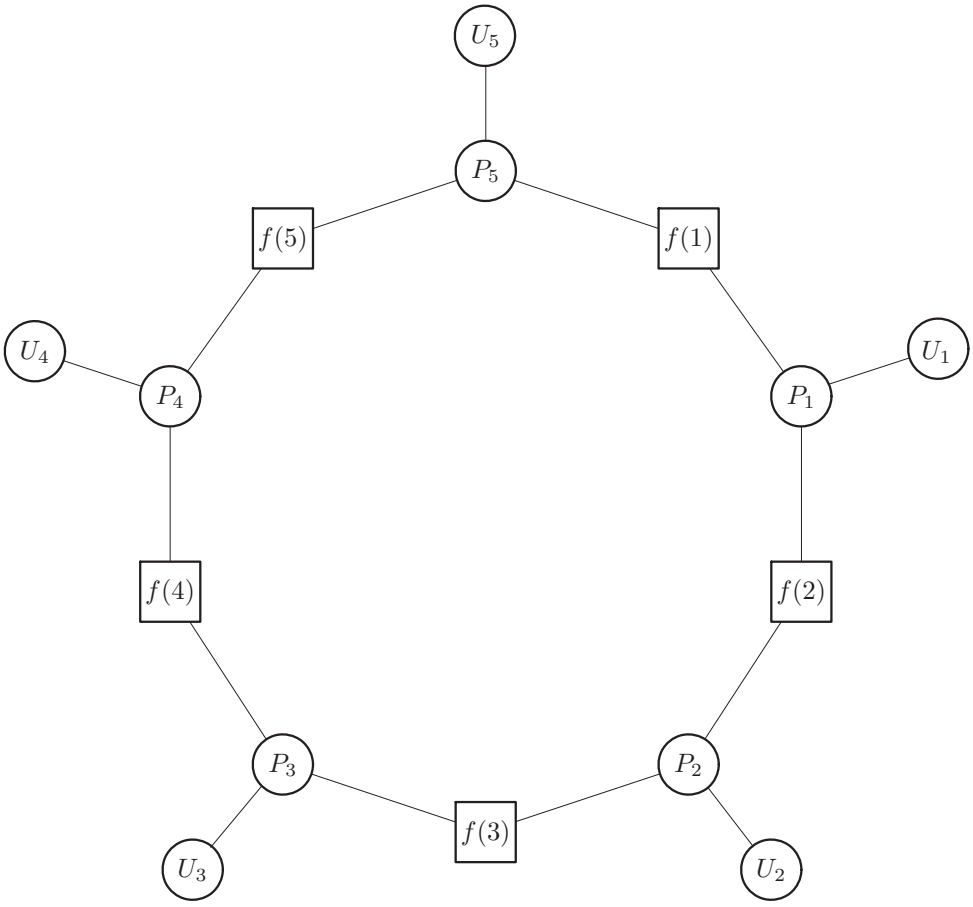
11.2.. Nincs szimmetrikus megoldás az étkező filozófusok problémájára

A lehetséges étkező filozófus algoritmusok egy érdekes osztálya a szimmetrikus algoritmusok osztálya. Az adott keretek között akkor mondjuk egy algoritmusról, hogy *szimmetrikus*, ha minden folyamat egyforma, és csak az általuk elérhető villa változókra hivatkozhat, a lokális $f(bal)$ és $f(jobb)$ néven, és ezenkívül az összes közös változó kezdőértéke megegyezik. Mint ahogy a 3. fejezetbeli vezetőválasztási problémánál, úgy itt sem nehéz belátni, hogy az étkező filozófusok feladat nem oldható meg a szimmetrikus esetben. A bizonyítás hasonló a 3.1. tétel bizonyításához.

11.2. tétel. . *Az étkező filozófusok problémájának nincs szimmetrikus megoldása.*

Bizonyítás. Az indirekt bizonyítás érdekében tegyük fel, hogy van egy A szimmetrikus algoritmusunk n folyamattal. Vizsgáljuk meg A egy α végrehajtási sorozatát, melynek kezdetekor minden folyamat ugyanabban a folyamatállapotban van és minden közös változónak ugyanaz a kezdőértéke. Ekkor az α „körkörösen" halad előre, a folyamatok megegyező lépéseket hajtanak végre 1,...,n,1,..., sorrendben, és minden folyamat a `próbál` lépéssel kezd. Továbbá minden nemdeterminisztikus lehetőségnél ugyanaz az érték lesz kiválasztva.

Például, ha egy $\texttt{próbál}_1$ művelet bekövetkezik, akkor minden másik folyamatnál be fog következni egy `próbál` művelet, és minden ezzel a művelettel kapcsolatos helyi állapotváltozás ugyanaz lesz, mint az egyes sorszámú folyamatnál. Egy másik példát nézve, ha az egyes sorszámú folyamat megszerzi a bal változóját,



11.2.. ábra. Az étkező filozófusok feladat felhasználói automatákkal.

akkor az összes többi folyamat is megszerzi a bal változóját, és az állapot, illetve változóérték változások ugyanazok lesznek, mint az egyes sorszámú folyamatnál.

Ezután egyszerű megmutatni a körkörös menetek száma (r) szerinti indukcióval, hogy r menet után minden folyamat ugyanabban az állapotban lesz és minden változónak ugyanaz lesz az értéke. A haladási tulajdonság alapján azonban néhány folyamat be fog lépni a *Be* (kritikus) szakaszba. Ez maga után vonja, hogy az összes többi folyamat is belép a *Be* szakaszba ugyanabban a menetben. Ez viszont ellentmond a kölcsönös kizárási tulajdonságnak. □

Példaként tekintsük a következő egyszerű szimmetrikus algoritmust.

HibásÉtkFil algoritmus (vázlatosan)

Minden folyamat a *próba* szakaszba belépve először a jobb villájára vár, majd a bal villájára. Miután megszerezte mindkét villát, belép a *Be* sza-

kaszba. Amikor egy folyamat kilép *Be*-ből, visszaadja mindkét villát, mielőtt visszatérne a *Ha* szakaszba.

Ezután nézzük a formális kódot; a *jobb* és *bal* a P_i folyamat által használt lokális nevek, melyeket az i és $i+1$ indexek helyett használ (a két villájára való hivatkozáskor).

11.1. algoritmus. HibásÉtkFil

Osztott változók:
Minden i-re, $1 \leq i \leq n$:
$f(i)$ logikai változó, kezdőértéke *hamis*, hozzáférhető P_i és P_{i+1} számára

A P_i folyamat műveletei:

Bemeneti:
- `próbál`$_i$
- `kilép`$_i$

Kimeneti:
- `belép`$_i$
- `halad`$_i$

Belső:
- `jobb_vizsgál`$_i$
- `bal_vizsgál`$_i$
- `jobb_visszaállít`$_i$
- `bal_visszaállít`$_i$

A P_i folyamat állapotai:

$p\_sz \in \{haladás, vizsgál\_jobb, vizsgál\_bal, elhagy\_próba, belépés,$
$visszaállítás\_jobb, visszaállítás\_bal, elhagy\_kilépés\}$, kezdőállapota *haladás*

A P_i folyamat átmenetei:

`próbál`$_i$
Hatás:
$p\_sz := vizsgál\_jobb$

`jobb_vizsgál`$_i$
Előfeltétel:
$p\_sz = vizsgál\_jobb$
Hatás:
if $f(jobb) = hamis$ **then**
$f(jobb) := igaz$
$p\_sz := vizsgál\_bal$

`bal_vizsgál`$_i$
Előfeltétel:
$p\_sz = vizsgál\_bal$
Hatás:
if $f(bal) = hamis$ **then**
$f(bal) := igaz$
$p\_sz := elhagy\_próba$

`belép`$_i$
Előfeltétel:
$p\_sz = elhagy\_próba$
Hatás:
$p\_sz := belépés$

`kilép`$_i$
Hatás:
$p\_sz := visszaállítás\_jobb$

`jobb_visszaállít`$_i$
Előfeltétel:
$p\_sz = visszaállítás\_jobb$
Hatás:
$f(jobb) := hamis$
$p\_sz := visszaállítás\_bal$

`bal_visszaállít`$_i$
Előfeltétel:
$p\_sz = visszaállítás\_bal$
Hatás:
$f(bal) := hamis$
$p\_sz := elhagy\_kilépés$

`halad`$_i$
Előfeltétel:
$p\_sz = elhagy\_kilépés$
Hatás:
$p\_sz := haladás$

Mivel a HibásÉtkFil algoritmus szimmetrikus, a 11.2. tételből következik, hogy nem oldja meg az étkező filozófusok problémáját. Érdekes lehet viszont megvizsgálni, hogy miért nem. Az könnyen látható, hogy a HibásÉtkFil algoritmus garantálja a jólformáltságot és teljesíti a kölcsönös kizárási feltételt, ez utóbbit azért, mert a kód biztosítja, hogy az a folyamat, amelyik belép a *Be* szakaszába, birtokában van mindkét vele szomszédos villának.

A haladási feltétel azonban nem teljesül. Nézzünk egy végrehajtást, amelyben minden folyamat egymás után belép a *próba* szakaszába. Ezután minden folyamat megfogja a jobboldali villáját. Ennél a pontnál minden folyamat készen áll arra,

hogy megpróbálja megszerezni a baloldali villáját. De mivel már minden villát felvettek, ez egyik folyamatnak sem sikerül. A rendszer így holtpontba kerül, mivel nincs lehetőség továbbhaladásra.

A 11.2. tétel alapján a gyűrű hálózat szimmetriáját mindenképpen meg kell bontanunk ahhoz, hogy megoldhassuk az étkező filozófusok problémáját. Ezt többféleképpen is megtehetjük. Például a folyamatok használhatnak különböző programokat; vagy megegyező programokat, de különböző kezdőértékekkel vagy egyedi azonosítókkal; esetleg a változók kezdőértékét állíthatjuk be eltérően; vagy használhatunk véletlenítést. Ebben a fejezetben a továbbiakban ezen lehetőségek némelyikét fogjuk szemléltetni.

11.3.. Jobb-bal algoritmus az étkező filozófusok problémájára

Ebben az alfejezetben bemutatunk egy (helyes) étkező filozófusok algoritmust, melyet JobbBalÉtkFil algoritmusnak nevezünk. Ez az algoritmus, azon felül, hogy a minimálisan megkívánt tulajdonságokat teljesíti, a kizárásmentességet is garantálja. Az algoritmusnak jó a legrosszabb esetbeli időigénye: konstans, ami független a gyűrű méretétől. A JobbBalÉtkFil algoritmus úgy töri meg a szimmetriát, hogy olyan folyamatai vannak, amelyeket két osztályba lehet sorolni. A két osztály neve legyen „jobb" és „bal". Az eltérő típusú folyamatok némileg eltérő programokat hajtanak végre, mivel nem ugyanazt a szomszédos villát keresik először; az osztályuk neve mutatja meg, melyiket először.

11.3.1.. Várakozási láncok

A konstans időigény különösen figyelemreméltó. Egy osztott rendszerben kétségkívül kívánatos a rendszer méretétől független idő. De hogyan lehet ezt elérni?

A JobbBalÉtkFil algoritmus egyike azon általános algoritmusosztály tagjainak, amelyben a folyamatok szekvenciálisak, először az egyik villára várnak és azután a másikra. Az ilyen típusú algoritmusoknál óvatosan kell megválasztanunk a villák keresésének sorrendjét. Például, ha minden folyamat először a jobboldali villáját próbálja felvenni, akkor fennáll a lehetőség, hogy holtpontba kerülnek, mint a HibásÉtkFil algoritmusban. Vannak más sorrendek is, amelyek nem engedik, hogy a folyamatok holtpontba kerüljenek, viszont nagyon rossz lehet az időigényük. Különösen néhány sorrend vezethet a folyamatok hosszú *várakozási láncaihoz*, amelyben minden folyamat egy olyan erőforrásra várakozik, amelyet egy másik, a láncban őt megelőző folyamat már lekötött.

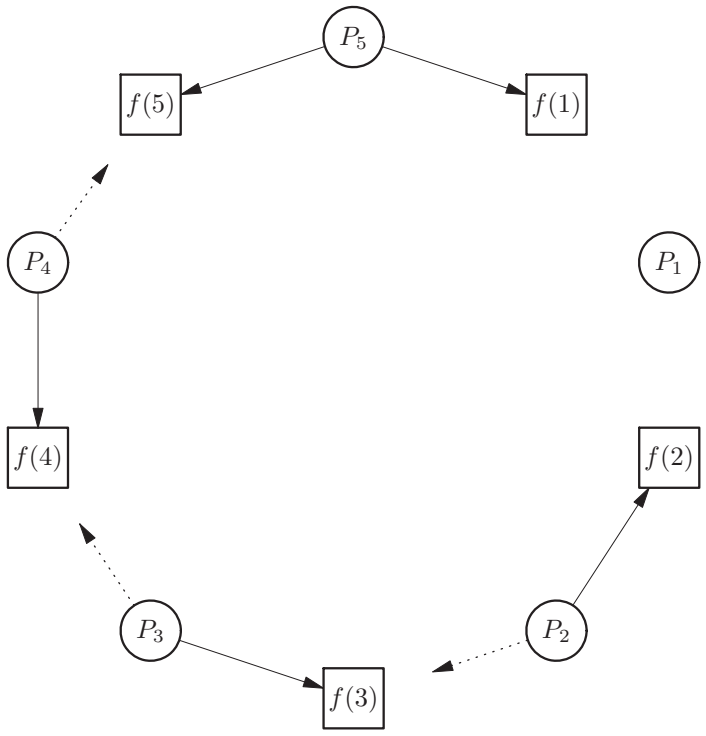
11.3.1. példa. Várakozási lánc

Nézzünk egy ötelemű gyűrűt. Tegyük fel, hogy egy algoritmusnak ebben a gyűrűben van egy végrehajtási sorozata, melyben az alábbi események következnek be a megadott sorrendben.

P_5 megszerzi mindkét villáját.

P_4 megszerzi a jobboldali villáját, azután várakozik a baloldalira.
P_3 megszerzi a jobboldali villáját, azután várakozik a baloldalira.
P_2 megszerzi a jobboldali villáját, azután várakozik a baloldalira.

Ez egy olyan láncot eredményez, melyben P_2 egy olyan villára várakozik, melyet P_3 kötött le, amelyik egy olyan villára várakozik, melyet P_4 kötött le, amelyik viszont egy olyan villára várakozik, melyet P_5 kötött le. Azaz ez egy olyan várakozási lánc, amelynek a hossza 3. Lásd a 11.3. ábrát.



11.3.. ábra. Egy várakozási lánc. A megszakítás nélküli nyilak jelzik a villa birtoklását, míg a szaggatott nyilak a várakozást.

Hasonló példa bármely $n \geq 3$ értékre olyan várakozási láncot eredményez, melynek hossza $n - 2$.

Vegyük észre, hogy a várakozási láncban lévő folyamatoknak szekvenciálisan kell belépniük a kritikus szakaszukba. Így minden olyan algoritmusra, amelyik ebbe az általános osztályba tartozik, annak az időszükséglete, hogy egy próba

szakaszban levő folyamat belépjen a kritikus szakaszába, a legrosszabb esetben legalábbis arányos a leghosszabb előállítható várakozási lánc hosszával. Éppen ezért a kis időkorlát eléréséhez garantálnunk kell egy alacsony korlátot a várakozási láncok maximális hosszára. A JobbBalÉtkFil algoritmus által előállított várakozási lánc maximális hossza 3.

11.3.2.. Az alap algoritmus

A JobbBalÉtkFil algoritmusban minden villához olyan közös változó tartozik, amelyik egy legfeljebb 2 hosszúságú, folyamatok indexeiből álló FIFO sort tartalmaz. Ez a sor azoknak a folyamatoknak az indexeit tartalmazza, amelyek a villára várakoznak, mégpedig abban a sorrendben, ahogy megpróbáltak hozzáférni a villához. Mivel minden villához csak két folyamat férhet hozzá, ezért elegendő a 2 hosszúságú sor.

Az egyszerűség kedvéért itt tegyük fel, hogy a gyűrűben lévő folyamatok száma páros. Egy egyszerű módosítással, amit meghagyunk gyakorlatnak (lásd 11-7. gyakorlat), az algoritmus alkalmassá tehető arra, hogy páratlan számú folyamat esetében működjön.

> **JobbBalÉtkFil algoritmus (vázlatosan, ha n páros)**
>
> Két különböző program van: egy azon folyamatok számára, melyek indexe páros, egy pedig azok számára, melyeké páratlan. Az alap stratégia nagyon egyszerű: a páratlan indexű folyamatok először a jobboldali villájukat próbálják megszerezni, míg a páros indexű folyamatok először a baloldali villájukat. Egy folyamat úgy próbál megszerezni egy villát, hogy beteszi az indexét az adott villához tartozó sor végére. Egy folyamat akkor szerzi meg a villát, ha az indexe a villához tartozó sor elejére kerül. Amikor egy folyamat kilép a *Be* szakaszából, visszaadja mindkét villáját, azaz kiveszi az indexét a villákhoz tartozó sorokból, mielőtt belépne a *Ha* szakaszába.

Megadjuk egy páratlan i indexű folyamat kódját előfeltétel/hatás stílusban. Páros i-kre a kód hasonló.

11.2. algoritmus. JobbBalÉtkFil (n páros, i páratlan)

Osztott változók:
Minden i-re, $1 \leq i \leq n$:
$f(i)$ egy legfeljebb 2 hosszúságú, folyamatok indexeit tartalmazó sor, kezdetben üres, hozzáférhető P_i és P_{i-1} számára

A P_i folyamat műveletei:

Bemeneti:
- `próbál`$_i$
- `kilép`$_i$

Kimeneti:
- `belép`$_i$
- `halad`$_i$

Belső:
- `jobb_vizsgál`$_i$
- `bal_vizsgál`$_i$
- `jobb_visszaállít`$_i$
- `bal_visszaállít`$_i$

A P_i folyamat állapotai:

$p\_sz \in \{haladás, vizsgál\_jobb, vizsgál\_bal, elhagy\_próba, belépés, visszaállítás\_jobb, visszaállítás\_bal, elhagy\_kilépés\}$, kezdőállapota *haladás*

A P_i folyamat átmenetei:

`próbál`$_i$
- Hatás:
  - $p\_sz := vizsgál\_jobb$

`jobb_vizsgál`$_i$
- Előfeltétel:
  - $p\_sz = vizsgál\_jobb$
- Hatás:
  - **if** i nincs benn $f(i).sor$-ban **then**
    - tegyük be i-t $f(i).sor$-ba
  - **if** i az első $f(i).sor$-ban **then**
    - $p\_sz := vizsgál\_bal$

`bal_vizsgál`$_i$
- Előfeltétel:
  - $p\_sz = vizsgál\_bal$
- Hatás:
  - **if** i nincs benn $f(i+1).sor$-ban **then**
    - tegyük be i-t $f(i+1).sor$-ba
  - **if** i az első $f(i+1).sor$-ban **then**
    - $p\_sz := elhagy\_próba$

`belép`$_i$
- Előfeltétel:
  - $p\_sz = elhagy\_próba$
- Hatás:

`kilép`$_i$
- Hatás:
  - $p\_sz := visszaállítás\_jobb$

`jobb_visszaállít`$_i$
- Előfeltétel:
  - $p\_sz = visszaállítás\_jobb$
- Hatás:
  - vegyük ki i-t $f(i).sor$-ból
  - $p\_sz := visszaállítás\_bal$

`bal_visszaállít`$_i$
- Előfeltétel:
  - $p\_sz = visszaállítás\_bal$
- Hatás:
  - vegyük ki i-t $f(i+1).sor$-ból
  - $p\_sz := elhagy\_kilépés$

`halad`$_i$
- Előfeltétel:
  - $p\_sz = elhagy\_kilépés$
- Hatás:
  - $p\_sz := haladás$

Most megvizsgáljuk a helyességet. A jólformáltság feltétele nyilvánvalóan teljesül. A kölcsönös kizárási feltétel teljesülése is könnyen látható, mivel a kód biztosítja, hogy az a folyamat, amelyik belép a *Be* szakaszába, mindkét villájához tartozó sorban az első helyen szerepel. Bizonyítani fogunk egy explicit felső korlátot arra az időre, amely alatt minden próbálkozó folyamat eljut a *kritikus* szakaszába. Egy kis (n-től független) felső korlátot pedig könnyű adni a *kilépés* szakaszra. A 11.1. lemma alapján ezek a korlátok elegendőek a kizárásmentesség

biztosításához, ami viszont elegendő a haladás garantálásához.

Az időkorlátokhoz, mint a korábbiakban, most is feltesszük, hogy l egy felső korlát minden folyamat lépésidejére, és c egy felső korlát arra az időtartamra, amelyet bármely felhasználó a *kritikus* szakaszában tölt.

11.3. lemma. . *A* JobbBalÉtkFil *algoritmusban bármely P_i folyamat Pr szakaszába való belépésétől a Be szakaszába való belépéséig tartó idő legfeljebb* $3c + 18l$.

Bizonyítás. Az alapgondolat az, hogy egy villa, amely két folyamat között van, vagy mindkettő számára az első villa, vagy mindkettő számára a második. Ebből adódik (az általunk feltett esetben, ahol n páros), hogy egy várakozási lánc maximális hossza 2.

Legyen T bármely P_i folyamat *próba* szakaszába való belépésétől a *kritikus* szakaszába való belépéséig tartó maximális idő. A célunk, hogy egy felső korlátot adjunk T-re. Ehhez definiáljunk egy S segédváltozót a következőképp: legyen S bármely P_i folyamatnak az első villája megszerzésétől a *kritikus* szakaszába való belépéséig tartó maximális idő. Formálisan azt mondjuk, hogy egy P_i folyamat abban a pillanatban szerezi meg az első villáját, amikor az i index lesz az első elem a villához tartozó sorban. (Ezt vagy a P_i folyamat egyik lépése, vagy annak a szomszédos folyamatnak az egyik lépése eredményezheti, amellyel a P_i folyamat megosztja a villát.)

Először adjunk egy felső korlátot T-re az S függvényében. Tekintsünk egy olyan P_i folyamatot, amely éppen belép a *próba* szakaszába. Ekkor l időn belül végrehajt egy π `vizsgál` eseményt, hogy megpróbálja megszerezni az első villáját. Ha ez azonnal sikerül, akkor további S időn belül a P_i folyamat belép a *kritikus* szakaszába. Így ebben az esetben a teljes idő legfeljebb $l + S$.

Ellenkező esetben a szomszédos folyamat, amelyikkel P_i megosztja a villát, nevezzük P_j-nek, birtokolja a villát, amikor π végrehajtódik. Ahogy a fentiekben említettük, ez feltétlenül P_j első villája. Így az a további idő, amíg a P_j folyamat elengedi a villát, legfeljebb $S + c + l$ (elegendő idő a P_j számára, hogy belépjen a *kritikus* szakaszába, elhagyja azt, majd visszaadja az első villáját). Abban a pillanatban, ahogy P_j elengedi a villáját, P_i megszerzi azt. Ez egyrészt abból adódik, ahogyan a „villa megszerzését" definiáltuk, másrészt abból, hogy a π végrehajtásakor a P_i folyamat betette az indexét a villához tartozó sorba. Így további S idő elteltével P_i eléri a *kritikus* szakaszát. Ez alapján ebben az esetben a P_i folyamat összesen legfeljebb $l + (S + c + l) + S = c + 2l + 2S$ idő alatt belép a *kritikus* szakaszába.

Ebből következik, hogy

$$T \;\leq\; \max\{l + S,\; c + 2l + 2S\} \;=\; c + 2l + 2S. \tag{11.1}$$

Ezután adjunk felső korlátot S-re. Tekintsünk egy olyan P_i folyamatot, amely éppen megszerezte az első villáját (azaz első lett a villához tartozó sorban). Ezt a tényt l időn belül észreveszi, és további l időn belül végrehajt egy `vizsgál` eljárást a második villájára. Ha ezt a második villát azonnal megszerzi, akkor további l idő alatt belép a *kritikus* szakaszába, így a teljes idő legfeljebb $3l$.

Ellenkező esetben a szomszéd, amelyikkel P_i megosztja a villát, birtokolja azt, és ez a villa a szomszéd folyamatnak is a második villája. Az az idő, amíg a szomszéd elengedi a második villát, legfeljebb $2l + c + 2l$ (elegendő idő, hogy a szomszédos folyamat észrevegye, hogy övé a villa, belépjen a *kritikus* szakaszába, elhagyja azt, és visszaadja a két villát). Miután a szomszéd visszaadta a villát, legfeljebb l ideig tart, amíg a P_i folyamat észreveszi ezt, és további l idő, amíg belép a *kritikus* szakaszába. Ez alapján ebben az esetben a P_i folyamat belép a *kritikus* szakaszába, legfeljebb $2l + (2l + c + 2l) + 2l = c + 8l$ idő alatt.

Ebből következik, hogy

$$S \;\leq\; \max\{3l,\; c + 8l\} \;=\; c + 8l. \tag{11.2}$$

A 11.1 és 11.2 egyenletet egyesítve kapjuk, hogy

$$T \;\leq\; 3c + 18l.$$

□

Mivel könnyű belátni, hogy az algoritmus a független haladás feltételét is teljesíti, a következő tételt kapjuk.

11.4. tétel. . *A* JobbBalÉtkFil *algoritmus megoldja az étkező filozófusok problémáját, valamint garantálja a kizárásmentességet, a független haladást, egy* $(3c + 18l)$*-es időkorlátot a próba szakaszra, és egy* $3l$*-es időkorlátot a kilépés szakaszra.*

Tehát a JobbBalÉtkFil algoritmus megtöri a szimmetriát, különbséget téve a páros és páratlan indexű folyamatok között. Attól a környezettől függően, amelyben az algoritmus fut, ésszerű vagy ésszerűtlen lehet feltenni, hogy a folyamatoknak birtokában van ez az ismeret. Például, ha az algoritmus egy osztott hálózatban fut (ahogy azt a 17. fejezetben vizsgáljuk), akkor lehet, hogy egy további protokoll szükséges a paritási információ meghatározásához, és ezen információ továbbadásához a folyamatok felé.

11.3.3.. Egy általánosítás

Megadunk egy egyszerű módszert arra, hogy hogyan lehet a JobbBalÉtkFil algoritmusban használt stratégiát általánosítani egy tetszőleges explicit erőforrásleírással megadott, tetszőleges erőforrás-hozzárendelési problémára. Az általánosított algoritmusnak továbbra is megmarad az az előnyös tulajdonsága, hogy van a folyamatok számától független időkorlátja. Bár az időkorlát itt nem olyan kicsi – így ezen a ponton még lehetőség van a javításra.

A folytatáshoz tegyük fel, hogy minden erőforráshoz tartozik egy közös változó, amely azok között az erőforrások között van megosztva, melyeknek szüksége van az adott erőforrásra. Mint a JobbBalÉtkFil algoritmusban, itt is feltesszük, hogy ez a változó tartalmaz egy FIFO sort, amelyben az van feljegyezve, hogy kik várakoznak az adott erőforrásra. Mint a JobbBalÉtkFil algoritmusban, most is minden folyamat egyenként vár a számára szükséges erőforrásokra. A

holtpont elkerülése érdekében feltesszük, hogy az erőforrások teljesen rendezettek és minden folyamat csak a rendezés szerinti sorrendben, a kisebbtől a nagyobb felé haladva juthat hozzá az általa igényelt erőforrásokhoz. Ezt a stratégiát *hierarchikus erőforrás-hozzárendelésnek* nevezzük.

Nem nehéz látni, hogy a hierarchikus erőforrás-hozzárendelés garantálja a haladást. Vázlatosan ez abból adódik, hogy ha a P_i folyamat egy olyan erőforrásra várakozik, amely a P_j folyamat által van lekötve, akkor a P_j csak olyan erőforrásra várakozhat, amely szigorúan nagyobb (az erőforrások közötti rendezés szerint), mint az, amelyre a P_i várakozik. Így, mivel csak véges sok folyamat van, az a folyamat, amelyik által a legnagyobb erőforrás van lekötve, nem blokkolt. Mivel a közös változóknál FIFO sorokat használunk, ezért kizárás sem fordulhat elő.

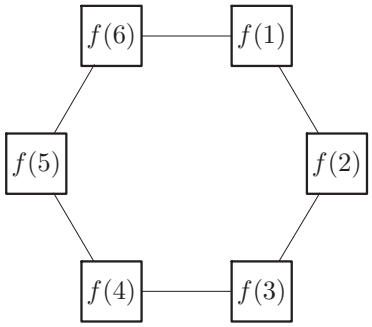
Bár a hierarchikus erőforrás-hozzárendelés biztosítja a haladást és a kizárásmentességet, ennek a stratégiának az időigénye általában nem túl jó. A várakozási láncok hosszára az egyetlen felső korlát a folyamatok teljes száma, azaz n, így az időigény is legalább arányos n-nel. Például a 11.3.1. példában leírt várakozási lánc előállhat egy olyan hierarchikus erőforrás-hozzárendelési algoritmusnál, melyben az erőforrások teljes rendezése éppen az argumentum szerinti rendezés, $f(1), f(2), f(3), f(4), f(5)$.

Amit mi szeretnénk, az az erőforrások egy „jó” teljes rendezése, amely a lehető legkisebb időkorlátot eredményezi. Egy ésszerű stratégia, ha megpróbáljuk minimalizálni az előállított várakozási láncok hosszát.

Tegyük fel, hogy van egy adott $\mathcal{R}$ explicit erőforrás-leírás, egy R univerzális erőforráshalmazzal és a folyamatok R_i egyedi erőforrásigényeivel. Egy „jó” teljes rendezés megkonstruálásához, először konstruáljuk meg az *erőforrásgráfot* ehhez a leíráshoz. A gráf csúcsai felelnek meg az erőforrásoknak, és pontosan akkor van él két csúcs között, ha van olyan folyamat, amely használja mindkét erőforrást.

11.3.2. példa. Erőforrásgráf

Az étkező filozófusok problémájához 6 csúcs esetében az erőforrásgráf olyan, ahogy azt a 11.4. ábra bemutatja.



11.4.. ábra. Erőforrásgráf az étkező filozófusok problémájához (n=6).

Ezután *beszínezzük* a gráf csúcsait olymódon, hogy a szomszédos csúcsok eltérő színűek legyenek. Megpróbáljuk minimalizálni a használt színek számát. (Most nem foglalkozunk azzal a problémával, hogy hogyan lehet elérni, hogy kevés szín is elegendő legyen. Valójában a minimális szám elérése egy NP-teljes feladat, de számunkra itt a színek alacsony száma elérhető. Például egy mohó algoritmus be tudja színezni a gráfot legfeljebb $d+1$ színnel, ahol d a gráf csúcsainak fokszámára egy felső korlát.)

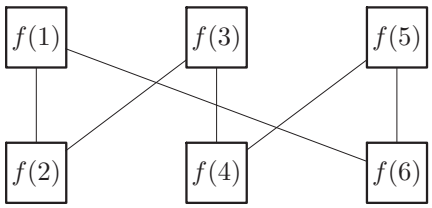
11.3.3. példa. Erőforrásgráf színezése

A 11.4. ábra erőforrásgráfjának színezéséhez elegendő csak két szín, például úgy, hogy a páratlan indexű erőforrásokat színezzük az egyik színnel, míg a párosakat a másikkal.

Most vegyünk egy tetszőleges teljes rendezést a színek között. Ez egy *parciális* rendezést eredményez az erőforrások között, melyben $r(i) < r(j)$ akkor és csak akkor, ha az $r(i)$ színe kisebb a színek közötti rendezés szerint, mint $r(j)$ színe. Bár ez csak egy parciális rendezés, vegyük észre, hogy egy adott folyamat által használt erőforrások között ez a rendezés teljes. Mivel mi egy teljes rendezést keresünk az összes erőforrás között, ezért kiegészítjük ezt a parciális rendezést teljes rendezéssé valamely tetszőleges módon (azaz a parciális rendezés egy *topologikus rendezését* használjuk).

11.3.4. példa. Az erőforrások parciális rendezése

A színezés a 11.3.3. példában a 11.5. ábra által bemutatott parciális rendezést eredményezi. A „kisebb" erőforrások a diagram tetején jelennek meg.



11.5.. ábra. Parciális rendezés az erőforrásokon.

Most leírhatjuk az algoritmust.

SZÍNEZ algoritmus (vázlatosan)

Minden folyamat a fentiekben leírt, a színezésen alapuló teljes rendezés szerinti növekvő sorrendben próbálja lekötni az erőforrásait. Egy folyamat úgy próbál megszerezni egy erőforrást, hogy az indexét berakja az erőforráshoz

tartozó sor végére. Egy folyamat megszerzi az erőforrást, ha az indexe az erőforrás sorának az elejére kerül. Ha a folyamat kilép a *Be* szakaszából, akkor visszaadja az összes erőforrását, azaz eltávolítja az indexét a hozzájuk tartozó sorokból.

Mivel bármely két, egyazon folyamat által használt erőforrás egymás figyelembevételével van rendezve (különböző színűre van festve), a SZÍNEZ algoritmusával ekvivalens leírás, ha azt mondjuk, hogy minden folyamat a parciális rendezés szerinti sorrendben próbálja lekötni az erőforrásait. Vegyük észre, hogy az étkező filozófusok problémájának esetében, ha páros méretű gyűrűvel számolunk, akkor a SZÍNEZ algoritmus a JOBBBALÉTKFIL algoritmussá egyszerűsödik.

A SZÍNEZ algoritmusban a várakozási láncok maximális hossza kisebb, vagy legfeljebb egyenlő a színezéshez használt színek számával. Ez azzal indokolható, hogy ha egy P_i folyamat egy olyan erőforrásra várakozik, amelyet a P_j folyamat tart lekötve, akkor a P_j csak egy „nagyobb" színű erőforrásra várakozhat.

Egy érdekes tulajdonság, amit igazolunk a SZÍNEZ algoritmus esetében, a folyamatok és erőforrások számától független időkorlát. Szokás szerint legyen l egy felső korlát a folyamatok lépésszámára és c egy felső korlát a *kritikus* szakasz idejére. Legyen k az erőforrások színezésére felhasznált színek száma, és m egy maximális érték arra, hogy egy tetszőleges erőforrást legfeljebb hány folyamat igényel. Megmutatjuk, hogy az időkorlát a legrosszabb esetben $\mathcal{O}(m^k c + km^k l)$. Ezt úgy értelmezhetjük, hogy az idő csak „helyi" paraméterektől függ, mivel a színek és az erőforrásokkénti felhasználók számának nem kell függnie a rendszer méretétől. Ha m és k kicsi n-hez képest (ahol n a folyamatok teljes száma a rendszerben), akkor ez a korlát javításnak számít a tetszőleges teljes rendezést használó hierarchikus erőforrás-hozzárendelési stratégiához képest, mivel az általános stratégia mellett n-hez közeli hosszúságú várakozási láncok is létrejöhetnek. De azt is vegyük észre, hogy ez a korlát nem olyan kicsi, mint amilyen lehetne – ahelyett, hogy arányos lenne a várakozási láncok maximális hosszával, azaz arányos lenne k-val, exponenciális k-ban.

11.5. lemma. . *A* SZÍNEZ *algoritmus egy példányában legyen k a színek száma és m egy felső korlát arra, hogy egy erőforrást hány felhasználó használ. Ekkor bármely P_i folyamat Pr szakaszába való belépésétől a Be szakaszába való belépéséig tartó idő $\mathcal{O}(m^k c + km^k l)$.*

Bizonyításvázlat. Feleltessük meg a színeket az $1, \ldots, k$ egészeknek. Legyen $T(i, j)$, ahol $1 \leq i \leq k$ és $1 \leq j \leq m$, a legrosszabb idő, amely alatt egy folyamat, amely elért valamely j-nél kisebb vagy egyenlő pozíciót, valamely i-nél nagyobb vagy egyenlő színű erőforrás várakozási sorában belép a *kritikus* szakaszába. Szeretnénk T-re felső korlátot adni, ami a *próba* szakaszba való belépéstől a *kritikus* szakaszba való belépésig tartó idő a legrosszabb esetben. Miután egy folyamat belép a *próba* szakaszába, legfeljebb l időt vesz igénybe, amíg az indexét beteszi valamely erőforrás sorába. Így,

$$T \leq l + T(1, m).$$

Felső korlátot adunk $T(i,j)$-re rekurzív egyenletek felállításával, ahogy azt a JobbBalÉtkFil algoritmus esetében is tettük. Az alapeset az, amikor egy folyamat első a legnagyobb színű erőforráshoz tartozó sorban:

$$T(k,1) \ \leq \ 2l.$$

Ez elegendő idő a folyamat számára, hogy felfedezze, hogy ő az első a sorban és azután belépjen a *kritikus* szakaszába.

A másik eset az, amikor a folyamat első valamely, a legnagyobbtól különböző színű erőforrás sorában:

$$T(i,1) \ \leq \ \max(2l, 2l + T(i+1,m)) = 2l + T(i+1,m) \text{ minden } i\text{−re}, \ 1 \leq i < k.$$

Ez időt hagy a folyamatnak, hogy észrevegye, hogy ő az első a sorban és azután vagy belépjen a *kritikus* szakaszába, vagy továbbmenjen egy másik, szükségképpen nagyobb színű erőforrás sorára.

Az utolsó eset, amikor egy folyamat valamely, az elsőtől eltérő helyen van valamelyik sorban:

$$T(i,j) \ \leq \ T(i,j-1) + c + kl + T(i,1) \text{ minden } j\text{−re}, \ 1 < j \leq m.$$

Ez elegendő idő a folyamatot a sorban megelőző másik folyamat számára, hogy belépjen a *kritikus* szakaszába, elhagyja azt, visszaadja az összes ($\leq k$) erőforrását (aminek hatására az eredeti folyamat az adott sor első helyére kerül), és azután a folyamat számára, amely most az első helyen áll, hogy belépjen a *kritikus* szakaszába.

Ezen egyenlőtlenségek megoldásával megkapjuk a keresett felső korlátot. Minden i-re, $1 \leq i \leq k$ kapjuk, hogy

$$T(i,m) \ \leq \ (m-1)(c+kl) \ + \ mT(i,1).$$

Így

$$T(i,m) \ \leq \ m(c+(k+2)l) \ + \ mT(i+1,m) \text{ minden } i\text{−re}, \ 1 \leq i < k,$$

és

$$T(k,m) \ \leq \ m(c+(k+2)l).$$

Ezért

$$T(1,m) \ \leq \ (c+(k+2)l)\sum_{i=1}^{k} m^i = \mathcal{O}(m^k(c+(k+2)l)).$$

Így

$$T = \mathcal{O}(m^k(c+(k+2)l)) = \mathcal{O}(m^k c + km^k l).$$

□

11.6. tétel. . *A* Színez *algoritmus megoldja az erőforrás-hozzárendelési problémát és garantálja a kizárásmentességet, a független haladást, egy $\mathcal{O}(m^k c + km^k l)$-es időkorlátot a próba szakaszra, és egy $\mathcal{O}(kl)$-es időkorlátot a kilépés szakaszra.*

A Színez algoritmusnak van néhány végrehajtása, melynek időigénye ehhez az exponenciális korláthoz közeli. Ilyen végrehajtások keresését meghagyjuk gyakorlatnak (lásd 11-8. gyakorlat). Természetesen szép lenne a színek számában exponenciális időkorlátot lecsökkenteni lineárisra, de ez egy másik algoritmust igényel.

11.4.. Véletlenített algoritmus az étkező filozófusok problémájára*

Az utolsó általunk bemutatott algoritmus egy *véletlenített* étkező filozófus algoritmus, amely garantálja a kölcsönös kizárást (biztosan) és biztosítja a haladást 1 valószínűséggel. Ezt az algoritmust a kitalálói után LehmannRabin algoritmusnak nevezzük. Ebben az algoritmusban minden folyamat azonos, a szimmetriát a véletlenítés használatával törjük meg.

Van néhány pont, amire szeretnénk felhívni a figyelmet az algoritmus bemutatása kapcsán. Az első, hogy az algoritmus demonstrálja, hogy a véletlenített algoritmusokat ugyanúgy lehet használni aszinkron esetben, mint szinkron esetben, és néha olyan dolgokat is megvalósíthatnak, melyeket nemvéletlenített algoritmusok nem. Például a LehmannRabin algoritmus akkor is megoldja az étkező filozófusok problémáját, ha a folyamatok azonosak, míg a 11.2. tételből következően ilyen esetben egyetlen determinisztikus algoritmus sem oldaná meg. Valójában persze óvatosnak kell lennünk, amikor azt mondjuk, hogy ez az algoritmus „megoldja az étkező filozófusok problémáját": a helyességi feltételek, amiket kielégít, nem egészen azok, mint amiket megadtunk korábban, mivel a haladási feltételt csak 1 valószínűséggel teljesíti, nem abszolút bizonyossággal.

A második dolog, hogy megmutatjuk, milyen jelentős valószínűségi állításokat lehet adni aszinkron véletlenített rendszerekre. Ennek módszere nem magától értetődő, mivel egy véletlenített algoritmus nem egyedül önmagában okoz egy valószínűség eloszlást a végrehajtási sorozatokon. Például az a sorrend, amely szerint egy aszinkron algoritmusban a folyamatok lépnek, inkább tetszőleges, nem pedig véletlenszerűen meghatározott. Egy valószínűségi eloszlás definiálása érdekében ezt a sorrendet valahol meg kell határozni.

A harmadik dolog, hogy bemutatunk egy Markov-stílusú elemzési technikát a valószínűségi időkorlát tulajdonságok bizonyítására. Ezeket a tulajdonságokat pedig felhasználhatjuk valószínűségi élénkségi tulajdonságok bizonyítására.

11.4.1.. Az algoritmus*

Mivel a folyamatok azonosak, feltesszük, hogy a villáikra azok lokális nevén hivatkoznak. Mint korábban, most is feltesszük, hogy minden folyamat az $f(jobb)$ és $f(bal)$ lokális neveket használja a villákra. A következő jelölést használjuk:

$$\bar{j} = \begin{cases} bal, & \text{ha } j = jobb, \\ jobb, & \text{ha } j = bal. \end{cases}$$

Az *első* := *véletlen* értékadás pedig itt azt jelenti, hogy az *első* változó értéke vagy *jobb* lesz vagy *bal*, mindkettő 1/2 valószínűséggel. Most nézzük az algoritmus vázlatos leírását.

11.3. algoritmus. LehmannRabin

Osztott változók:
Minden i-re, $1 \leq i \leq n$:
$f(i)$, logikai változó, kezdetben *hamis*, hozzáférhető P_i és P_{i-1} számára

A P_i folyamat:
** Haladás szakasz **

```
próbál_i
    do forever
        első := véletlen
        wait until f(első) = hamis
            f(első) := igaz
            if f(ellenkező(első)) = hamis then
                f(ellenkező(első)) := igaz
                goto L
            else f(első) := hamis
L:
belép_i
```

** Kritikus szakasz **

```
kilép_i
    Visszaadja mindkét villáját.

halad_i
```

Tehát egy, a próba szakaszában lévő P_i folyamat egy ciklust hajt végre, melynek minden menetében megpróbálja megszerezni mindkét villáját. Minden menetben véletlenszerűen kiválasztja az első villát, és addig vár, ameddig szükséges, hogy megszerezze. Miután megszerezte az első villát, nem vár bizonytalan ideig a második villájára, csak egyszer megnézi, hogy az elérhető-e. Ha igen, akkor a P_i folyamat megszerzi azt és folytatja a végrehajtást a *Be* szakasszal. Ha nem, akkor a P_i folyamat ebben a menetben feladja a kísérletezést, visszaadja az első villáját, és a következő menetben újra próbálkozik a villák megszerzésével.

A félreérthetőség elkerülése végett megadjuk az előfeltétel/hatás kódot is.

11.4. algoritmus. LehmannRabin (átírva)

Osztott változók:
Minden i-re, $1 \leq i \leq n$:
$f(i)$, logikai változó, kezdőértéke *hamis*, hozzáférhető P_i és P_{i+1} számára

A P_i folyamat műveletei:

Bemeneti:
- `próbál`$_i$
- `kilép`$_i$

Kimeneti:
- `belép`$_i$
- `halad`$_i$

Belső:
- `feldob`$_i$
- `vár`$_i$
- `második`$_i$
- `eldob`$_i$
- `jobb_visszaállít`$_i$
- `bal_visszaállít`$_i$

A P_i folyamat állapotai:
$p\_sz \in \{$*haladás,feldobás,várakozás,második,eldobás,elhagy_próba,belépés, visszaállítás_jobb,visszaállítás_bal,elhagy_kilépés*$\}$, kezdőértéke *haladás*
első $\in \{$*jobb,bal*$\}$, kezdőértéke tetszőleges

A P_i folyamat átmenetei:

`próbál`$_i$
Hatás:
$p\_sz := feldobás$

`feldob`$_i$
Előfeltétel:
$p\_sz = feldobás$
Hatás:
első $:=$ *véletlen*
$p\_sz := várakozás$

`vár`$_i$
Előfeltétel:
$p\_sz = várakozás$
Hatás:
if $f(első) = hamis$ **then**
$f(első) := igaz$
$p\_sz := második$

`második`$_i$
Előfeltétel:
$p\_sz = második$
Hatás:
if $f(\overline{első}) = hamis$ **then**
$f(\overline{első}) := igaz$
$p\_sz := elhagy\_próba$
else $p\_sz := eldobás$

`eldob`$_i$
Előfeltétel:
$p\_sz = eldobás$
Hatás:
$f(első) := hamis$
$p\_sz := feldobás$

`belép`$_i$
Előfeltétel:
$p\_sz = elhagy\_próba$
Hatás:
$p\_sz := belépés$

`kilép`$_i$
Hatás:
$p\_sz := visszaállítás\_jobb$

`jobb_visszaállít`$_i$
Előfeltétel:
$p\_sz = visszaállítás\_jobb$
Hatás:
$f(jobb) := hamis$
$p\_sz := visszaállítás\_bal$

`bal_visszaállít`$_i$
Előfeltétel:
$p\_sz = visszaállítás\_bal$
Hatás:
$f(bal) := hamis$
$p\_sz := elhagy\_kilépés$

`halad`$_i$
Előfeltétel:
$p\_sz = elhagy\_kilépés$
Hatás:
$p\_sz := haladás$

Ha formálisan nézzük, akkor ezzel a kóddal egy, a 8.8. alfejezetben definiáltak szerinti *valószínűségi b/k automatát* adtunk meg. A véletlen választás lépései pontosan a `feldob` lépések; egy új állapot helyett minden ilyen lépésnek egy valószínűségi eloszlása van, amely két lehetséges következő állapotot tartalmaz, mindkettőt 1/2 valószínűséggel. Vegyük észre, hogy a rendszer végrehajtási sorozata *nemdeterminisztikus választások* és *valószínűségi választások* kombinációjának segítségével halad előre. A nemdeterminisztikus választások megadják, hogy melyik folyamat lép a következő lépésben, és ezáltal azt is, hogy mi lesz a következő lépés, míg a valószínűségi választások meghatározzák a `feldob` lépés új állapotát.

11.4.2.. Helyesség*

Könnyű látni, hogy a LehmannRabin algoritmus garantálja a jólformáltságot, a kölcsönös kizárást és a független haladást; ezen állítások egyikébe sincs valószínűség belevonva. Formálisan ezek a rendszer 8.8. alfejezetben definiáltak szerinti *nemdeterminisztikus változatának* tulajdonságai. Azonban a haladási tulajdonság nem teljesül biztosan.

11.4.1. példa. A LehmannRabin algoritmus olyan végrehajtási sorozata, amely nem teljesíti a haladás feltételét

> Tekintsük a LehmannRabin algoritmus egy α végrehajtási sorozatát, melyben a folyamatok körkörös rendben követik egymást, és mindig ugyanazt a véletlen választást csinálják. Vegyük észre, hogy α egy pártatlan végrehajtási sorozat (a rendszer nemdeterminisztikus változatában). Azonban α-ban egyetlen folyamat sem éri el a *Be* szakaszt.

A számunkra érdekes dolog, amit bizonyíthatunk a LehmannRabin algoritmusról, hogy 1 valószínűséggel garantálja a haladást. Valójában ahelyett, hogy ezt bizonyítanánk, egy erősebb *valószínűségi időkorlát* tulajdonságot fogunk bizonyítani $\mathcal{P} \xrightarrow[p]{t} \mathcal{B}$ formában. Informálisan ez azt jelenti, hogy minden elérhető állapotból, amelyben néhány folyamat a *Pr* szakaszában van, legalább p valószínűséggel és t időn belül néhány folyamat belép a *Be* szakaszába. Az 1 valószínűségű haladási feltétel teljesülése ezen állítás ismételt felhasználásával bizonyítható.

Ahhoz, hogy állításokat mondjunk egy bizonyos esemény valószínűségéről, szükségünk van egy valószínűségi eloszlásra a végrehajtási sorozatokon. Ahogy már leírtuk, a rendszer tartalmaz nemdeterminisztikus választásokat – azaz, hogy melyik folyamat lép a következő lépésben – valamint valószínűségi választásokat. A nemdeterminisztikus választásokat fel kell oldani ahhoz, hogy egy tisztán

valószínűségi rendszert kapjunk. Valójában azt állíthatjuk, hogy a rendszernek megvan a kívánt tulajdonsága attól függetlenül, hogy hogyan oldjuk fel a nemdeterminisztikus választásokat.

Hasznos úgy elképzelni, hogy a nemdeterminisztikus választások egy *ellenfél* ellenőrzése alatt állnak. Megengedjük az ellenfélnek, hogy tetszőleges folyamatot kiválasszon egészen addig, amíg lehetővé teszi minden olyan folyamat pártatlan haladását, amely a *próba* vagy a *kilépés* szakaszában van. Valójában mivel valószínűségi időkorlát állításokat bizonyítunk, az ellenfélnek nem csak azt engedjük meg, hogy kiválassza, melyik folyamat lépjen a következő lépésben, hanem azt is, hogy mikor kerüljön sor a lépésre. Az időzítés eldöntése függ a folyamatok lépésidejére vonatkozó l felső korláttól, és a *kritikus* szakasz idejére vonatkozó c felső korláttól, valamint attól a követelménytől, hogy ha a végrehajtás végtelen, akkor az időnek is végtelenbe kell átmennie. A legerősebb eredmény eléréséhez lehetővé kell tennünk az ellenfélnek, hogy olyan hatékony legyen, amennyire csak lehetséges. Ezért feltesszük, hogy amikor eldönti, hogy ki lép a következő lépésben és a lépés eredménye mikor lesz észlelhető, akkor *teljes tudása* van az eddigi végrehajtási sorozatról, beleértve a folyamatállapotok és az eddigi véletlen választások ismeretét is.

Formálisan az $\mathcal{A}$ ellenfél egy függvény, amely hozzárendeli a véges végrehajtási sorozatokat (folyamat,idő) párokhoz, így meghatározva, hogy melyik lesz a következő folyamat, amelyik lép, és a lépés megtétele melyik időpillanatban történik meg. Véletlen döntések minden egyes $\mathcal{D}$ sorozatára létezik egy, az $\mathcal{A}$ ellenfél által a $\mathcal{D}$-ben megadott véletlen választásokkal generált *végrehajtás*$(\mathcal{A}, \mathcal{D})$ egyedi ütemezett végrehajtás. Az ellenfél korlátozott, így minden ütemezett végrehajtási sorozatnak *végrehajtás*$(\mathcal{A}, \mathcal{D})$-ben megvan az előző bekezdésben leírt pártatlansági és ütemezési tulajdonsága.

Egy rögzített $\mathcal{A}$ ellenfél meghatároz egy valószínűségi eloszlást az algoritmus ütemezett végrehajtásainak halmazán. Mivel minden véletlen válasz „jobb" vagy „bal", mindkettő $1/2$ valószínűséggel, ezért a döntések minden sorozatához (azok minden mérhető halmazához) tartozik egy valószínűség. Ez a valószínűségi eloszlás a $\mathcal{D}$ sorozatokon meghatároz egy valószínűségi eloszlást a *végrehajtás*$(\mathcal{A}, \mathcal{D})$ ütemezett végrehajtásokon.

Még egy jelölésre szükségünk van a bizonyításhoz. Ha $\mathcal{U}$ és $\mathcal{U}'$ állapotok halmazai, akkor az $\mathcal{U} \xrightarrow[p]{t} \mathcal{U}'$ jelölés a következőket jelenti. Minden $\mathcal{A}$ ellenfélre, ha az algoritmus az $\mathcal{U}$-beli állapotból indul, akkor az $\mathcal{A}$ által meghatározott végrehajtási sorozatok valószínűségi eloszlásában annak a valószínűsége, hogy az algoritmus t időn belül elér egy $\mathcal{U}'$-beli állapotot, legalább p. Mint a következő lemma mutatja, az ilyen állítások összekapcsolhatók.

11.7. lemma. .

1. *Ha* $\mathcal{U} \xrightarrow[p]{t} \mathcal{U}'$ *és* $\mathcal{U}' \xrightarrow[p']{t'} \mathcal{U}''$, *akkor* $\mathcal{U} \xrightarrow[pp']{t+t'} \mathcal{U}''$.
2. *Ha* $\mathcal{U} \xrightarrow[p]{t} \mathcal{U}'$, *akkor* $\mathcal{U} \cup \mathcal{U}'' \xrightarrow[p]{t} \mathcal{U}' \cup \mathcal{U}''$.

Ez most már elegendő eszköz ahhoz, hogy bizonyítani tudjuk a haladási tulajdonságot. Még egy technikai dolog: a bizonyításunk némely része csak abban az esetben működik, ha a gyűrű n mérete legalább 3. Ezért ezt (azaz, hogy n legalább 3) ennek a fejezetnek a további részében feltesszük, és azt az (egyszerűbb) esetet, amikor $n = 2$, meghagyjuk gyakorlatnak (lásd 11-13. gyakorlat).

Legyen

- $\mathcal{P}$ a LehmannRabin algoritmus azon elérhető állapotainak halmaza, melyekben néhány folyamat a *Pr* szakaszában van;
- $\mathcal{B}$ pedig azon elérhető állapotainak halmaza, melyekben némely folyamat a *Be* szakaszában van.

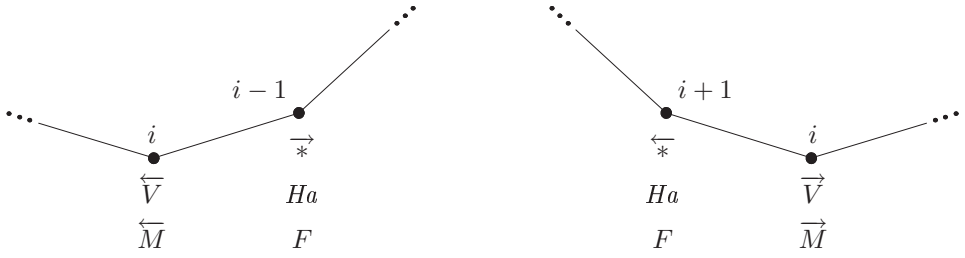
Megmutatjuk, hogy $\mathcal{P} \xrightarrow[1/16]{14l} \mathcal{B}$. Azaz, hogy bármely olyan elérhető állapotból indulva, melyben van folyamat a *Pr* szakaszában, teljesül, hogy legalább $1/16$ valószínűséggel valamelyik folyamat $14l$ időn belül a *Be* szakaszában lesz. Ezt az állítást a 11.8.-11.12. lemmák és a 11.7. lemmában megadott általános szabályok felhasználásával bizonyítjuk.

Néhány egyszerűbb jelölés hasznos lehet a folyamatok állapotainak osztályozásához. Jelölje F, V, M, D és E azon folyamatállapotok halmazait, ahol rendre $p\_sz =$ *feldobás, várakozás, második, eldobás* és *elhagy_próba*; az állapotoknak ez az öt halmaza felosztja a *Pr próba* szakaszt. Továbbá szétosztjuk a V, M, D állapotokat az *első* értéke szerint: $\overrightarrow{V}, \overrightarrow{M}, \overrightarrow{D}$ jelöli értelemszerűen V, M, D azon részhalmazait, melyekben *első* $=$ *jobb*, és $\overleftarrow{V}, \overleftarrow{M}, \overleftarrow{D}$ értelemszerűen azokat, melyekben *első* $=$ *bal*. A $\overrightarrow{*}$ jelölést használjuk $\overrightarrow{V} \cup \overrightarrow{M} \cup \overrightarrow{D}$-re, és hasonlóan a $\overleftarrow{*}$ jelölést $\overleftarrow{V} \cup \overleftarrow{M} \cup \overleftarrow{D}$-re. Most definiáljuk a rendszer állapotainak azon halmazait, melyekre szükségünk van a segédtételekben.

Legyen

- $\mathcal{E}$ azon elérhető állapotok halmaza, melyekben néhány folyamat az E-ben van (azaz az *elhagy_próbá*-nál);
- $\mathcal{HP}$ a $\mathcal{P}$ azon részhalmaza, amelyik azokat az állapotokat tartalmazza, melyekben minden folyamat vagy a *haladás* vagy a *próba* szakaszában van;
- $\mathcal{F}$ a $\mathcal{HP}$ azon részhalmaza, amelyik azokat az állapotokat tartalmazza, melyekben néhány folyamat F-ben van (azaz a *feldobás*-nál);
- $\mathcal{J}$ a $\mathcal{HP}$ azon részhalmaza, amelyik azokat az állapotokat tartalmazza, melyekben létezik egy P_i folyamat, amelyre a következők valamelyike teljesül:
  - $i \in \overleftarrow{V} \cup \overleftarrow{M}$ és $i-1 \in \overrightarrow{*} \cup Ha \cup F$
  - $i \in \overrightarrow{V} \cup \overrightarrow{M}$ és $i+1 \in \overleftarrow{*} \cup Ha \cup F$.

Az első három halmaz magától értetődő. Az utolsó halmaz, $\mathcal{J}$ a „jó” állapotok halmaza, amelyekben két folyamat olyan helyzetben van, hogy nagy valószínűséggel egyikük hamarosan megszerzi mindkét villáját. A $\mathcal{J}$-ben megengedett két helyzetet mutatja be a 11.6. ábra. Egy durva megközelítés, hogy egy jó beállításban a két szomszédos folyamatnak nagy valószínűséggel ugyanaz a második villája. Ha ez így van, akkor amelyik először megpróbál hozzáférni ehhez a villához, az megszerzi azt, és sikeresen belép a *Be* szakaszába.



11.6.. ábra. A LehmannRabin algoritmus jó állapotai.

Ezután megmutatjuk a következő állításokat:

- $\mathcal{P} \xrightarrow[1]{3l} \mathcal{HP} \cup \mathcal{B}$;
- $\mathcal{HP} \xrightarrow[1]{3l} \mathcal{F} \cup \mathcal{E}$;
- $\mathcal{F} \xrightarrow[1/4]{2l} \mathcal{J} \cup \mathcal{E}$;
- $\mathcal{J} \xrightarrow[1/4]{5l} \mathcal{E}$;
- $\mathcal{E} \xrightarrow[1]{l} \mathcal{B}$.

A 11.7. lemma alapján ezeket az állításokat összekapcsolhatjuk, és így megkapjuk a kívánt eredményt, azaz hogy $\mathcal{P} \xrightarrow[1/16]{14l} \mathcal{B}$.

Először bebizonyítjuk a három 1 valószínűségű állítást, mivel ezek a legkönnyebbek. Valójában ezek biztosan igazak, nem csak 1 valószínűséggel.

11.8. lemma. . $\mathcal{E} \xrightarrow[1]{l} \mathcal{B}$.

Bizonyítás. Ha egy folyamat az *elhagy_próba* szakasznál van, akkor l időn belül ugyanaz a folyamat végrehajt egy lépést, és így belép a *Be* szakaszba. □

11.9. lemma. . $\mathcal{P} \xrightarrow[1]{3l} \mathcal{HP} \cup \mathcal{B}$.

Bizonyítás. Ha valamelyik folyamat kezdetben a *Be* szakaszban van, vagy belép oda $3l$ időn belül, akkor nem kell mit bizonyítanunk, úgyhogy tegyük fel, hogy nem ez az eset áll fenn. Ekkor minden folyamat a $Ha \cup Pr \cup Ki$-ben van legalább $3l$ ideig, és egyetlen folyamat sem lép be *Ki* szakaszba ez alatt az idő alatt (mivel feltettük, hogy nincs folyamat a *Be* szakaszban). De bármely folyamat, amely a *Ki* szakaszban van visszatér a *Ha* szakaszba $3l$ idő alatt. Ezért minden folyamat kénytelen belépni a $Ha \cup Pr$-be $3l$ idő alatt, ahogy megkívántuk. □

11.10. lemma. . $\mathcal{HP} \xrightarrow[1]{3l} \mathcal{F} \cup \mathcal{E}$.

Bizonyítás. Ha kezdetben bármelyik folyamat az $(F \cup E)$-ben van, vagy $3l$ idő alatt belép E-be, akkor az állítás teljesül, ezért tegyük fel, hogy nem ez az eset áll fenn. Ekkor egyetlen folyamat sem lép be a *Be* szakaszba $3l$ ideig, így minden rendszerállapot, amely $3l$ időn belül előfordul, $\mathcal{HP}$-ben van. Ezért ha bármely folyamat belép F-be $3l$ időn belül, akkor a rendszer $\mathcal{F}$-ben lesz, és az állítás teljesül, ezért tegyük fel, hogy ez az eset sem áll fenn. Ebből az következik, hogy egyetlen folyamat sem lép be a *próba* szakaszba $3l$ időn belül.

Így kizárásos alapon kezdetben minden folyamat a $(Ha \cup V \cup M \cup D)$-ben van. De ha valamelyik folyamat kezdetben a $(M \cup D)$-ben van, vagy belép oda l időn belül, akkor további $2l$ időn belül belép $(F \cup E)$-be, ami ellentmondás. Így az egyetlen lehetőség, hogy kezdetben minden folyamat a $(Ha \cup V)$-ben van, és (mivel $\mathcal{HP} \subseteq \mathcal{P}$) néhány folyamat V-ben van, továbbá egyetlen folyamat sem lép be $(M \cup D)$-be l időn belül. Vegyük észre, hogy ez azt jelenti, hogy kezdetben egyetlen folyamat sem birtokol villát.

Mivel néhány folyamat V-ben van, tudjuk, hogy valahány folyamatnak lépnie kell l időn belül. Legyen P_i az első folyamat, amelyik lép. Ha P_i kezdetben a Ha szakaszában van, akkor belép a *próba* szakaszába, ami ellentmondás. Másrészről, ha P_i kezdetben V-ben van, akkor, mivel nincs egyetlen villa sem lefoglalva, P_i azonnal megszerzi az első villáját. Viszont ezáltal P_i belép M-be, ami szintén ellentmondás. □

Mostanáig nem foglalkoztunk valószínűségekkel. A maradék két állításnál ezt is figyelembe vesszük. Ezek közül az első megmutatja, miért igaz, hogy ha egy tetszőleges olyan állapotból indulunk, melyben néhány folyamat a *feldobás*-nál van, akkor legalább $1/4$ valószínűséggel vagy elérünk egy „jó” állapotot, vagy néhány folyamat eléri az *elhagy_ próba* szakaszt.

11.11. lemma. . $\mathcal{F} \xrightarrow[1/4]{2l} \mathcal{J} \cup \mathcal{E}$.

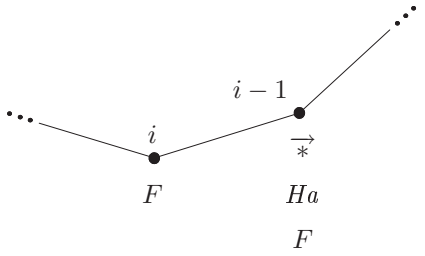
Bizonyítás. Ha valamelyik folyamat kezdetben E-ben van, készen vagyunk. Tegyük fel, hogy nem ez az eset áll fenn. Legyen P_i egy olyan folyamat, amely kezdetben F-ben van. Ekkor a következők valamelyikének teljesülnie kell kezdetben.

1. $i-1 \in \overrightarrow{*} \cup Ha \cup F$.

   Lásd a 11.7. ábrát. Ebben az esetben $1/4$ valószínűséggel P_i következő véletlen választása *bal*, míg P_{i-1}-é *jobb*. Tegyük fel, hogy ezek lesznek a választások.

   Így l időn belül P_i végrehajtja a `feldob` lépést, és ezáltal belép $\overleftarrow{V}$-be, mivel feltettük, hogy a következő véletlen választása *bal*. Ilyenkor két eset állhat fenn.

   (a) Időközben P_{i-1} nem foglalja le a közös villát. Ekkor azt állítjuk, hogy a P_{i-1} állapotának még mindig a $\overrightarrow{*} \cup Ha \cup F$ halmazban kell lennie; ezt



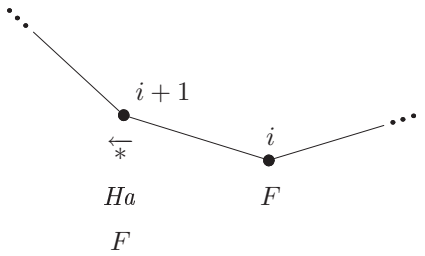
11.7.. ábra. A kezdőállapot az első esetre a 11.11. lemma bizonyításában.

a halmazból kivezető lehetséges átmenetek vizsgálatából szűrhetjük le annak figyelembe vételével, hogy P_{i-1} következő véletlen választása *jobb*. Így a rendszer állapota $\mathcal{J}$-be kerül, ami elegendő számunkra.

(b) Időközben P_{i-1} lefoglalja a közös villát. Ekkor tekintsük az első időpillanatot, amikor ezt megteszi. Ebben a pillanatban a közös villa feltétlenül P_{i-1} második villája. (A $\overrightarrow{D}, Ha$ és F esetekben ez abból adódik, hogy P_{i-1} következő véletlen választása *jobb*). Ekkor P_{i-1} megszerzi a második villáját és belép L-be, ami elegendő számunkra.

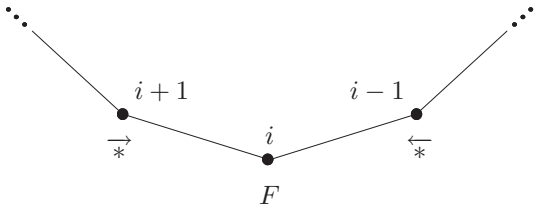
2. $i+1 \in \overleftarrow{*} \cup Ha \cup F$.

Lásd a 11.8. ábrát. Ez hasonló az előző esethez.



11.8.. ábra. A kezdőállapot a második esetre.

3. $i-1 \in \overleftarrow{*}$ és $i+1 \in \overrightarrow{*}$.

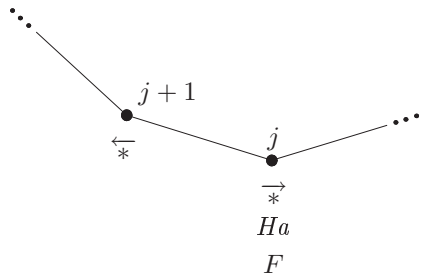


11.9.. ábra. A kezdőállapot a harmadik esetre.

Lásd a 11.9. ábrát.

Ez egy érdekes eset, mivel a helyzet egyáltalán nem hasonlít arra, ahogy feltételezésünk szerint egy „jó” helyzet fest. De mivel egy gyűrűben dolgozunk, valójában az P_i körüli kedvezőtlen helyzet maga után vonja, hogy

valahol máshol a gyűrűben kell lennie egy P_j folyamatnak, amelyre teljesül, hogy $j+1 \in \overleftarrow{*}$ és $j \in \overrightarrow{*} \cup Ha \cup F$. (Ennek belátását meghagyjuk egy egyszerű gyakorlatnak – lásd 11-11. gyakorlat.) Lásd a 11.10. ábrát. Ekkor a dolgok sokkal jobban festenek a P_j folyamat körül.



11.10.. ábra. Máshol a gyűrűben.

Ha $j+1 \in \overleftarrow{V} \cup \overleftarrow{M}$, akkor a kezdőállapot már $\mathcal{J}$-ben van, és kész vagyunk. Az egyetlen másik lehetőség, hogy $j+1 \in \overleftarrow{D}$ és $j \in \overrightarrow{*} \cup Ha \cup F$. De ebben az esetben $1/4$ valószínűséggel P_{j+1} következő véletlen választása *bal*, és P_j következő véletlen választása *jobb*, tegyük fel ezt. Ekkor $2l$ időn belül a P_{j+1} folyamat végrehajt két lépést, és így bekerül $\overleftarrow{V}$-be. Ez azt jelenti, hogy az eredményül kapott rendszerállapot $\mathcal{J}$-ben lesz, kivéve, ha időközben j kikerülne a $\overrightarrow{*} \cup Ha \cup F$ halmazból. Ha azonban ez történik, akkor P_j megszerzi a második villáját és belép E-be, ami szintén elegendő számunkra.

□

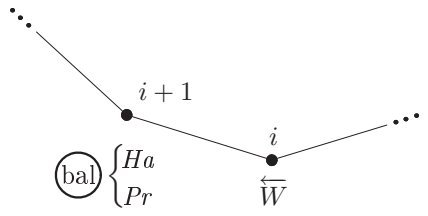
Az utolsó segédtétel megmutatja, hogy miért jó egy „jó” állapot: egy „jó” állapotból legalább $1/4$ valószínűséggel néhány folyamat hamarosan eléri E-t.

11.12. lemma. . $\mathcal{J} \xrightarrow[1/4]{5l} \mathcal{E}$.

Bizonyítás. Mivel a kezdőállapot $\mathcal{J}$-ben van, ezért $\mathcal{HP}$-ben van, és a $\mathcal{J}$ definíciójában szereplő két feltétel közül (legalább) az egyik teljesül. Az általánosság megszorítása nélkül feltehetjük, hogy az első teljesül, azaz, hogy létezik olyan P_i folyamat, melyre $i \in \overleftarrow{V} \cup \overleftarrow{M}$ és $i-1 \in \overrightarrow{*} \cup Ha \cup F$. A bizonyítás a második feltételre szimmetrikus.

Felhasználunk három segédtételt. Ezek állításai nem tartalmaznak valószínűségeket explicit módon, ehelyett bizonyos jövőbeli véletlen választásokra hivatkoznak, miáltal természetesen implicit valószínűségek is megjelennek bennük. Az első arra ad felső korlátot, hogy legfeljebb mennyi ideig várakozik egy folyamat az első villájára abban az esetben, ha egyik szomszédja kedvező helyzetben van.

11.13. segédtétel. . *Ha $i+1 \in Ha \cup Pr$, a következő véletlen választás bal és $i \in \overleftarrow{V}$, akkor $4l$ időn belül vagy $i \in \overleftarrow{M}$ vagy $i+1 \in E$ teljesülni fog.*



11.11.. ábra. A 11.13. állításban vizsgált helyzet.

Lásd a 11.11. ábrát.

Bizonyítás. A bizonyítás a P_{i+1} folyamat állapota szerinti (némileg talán unalmas) esetekre bontással történik.

1. $i+1 \in E$.

   Ebben az esetben készen vagyunk.

2. $i+1 \in \overleftarrow{*} \cup Ha \cup F$.

   Ekkor kezdetben P_{i+1} nem birtokolja a közös villát. A P_i folyamat l időn belül megpróbálja elérni a villát. Ha időközben a P_{i+1} folyamat nem foglalja le, akkor P_i megszerzi, és belép $\overleftarrow{M}$-be, ahogy megköveteltük.

   Másfelől tegyük fel, hogy a P_{i+1} folyamat lefoglalja időközben a közös villát, és vizsgáljuk meg az első ilyen időpontot. A P_{i+1} állapotára vonatkozó feltevésünk miatt a közös villa feltétlenül P_{i+1} második villája. (A $\overleftarrow{D}, Ha$ és F esetekben ez abból adódik, hogy P_{i+1} következő véletlen választása *bal*.) Mivel P_{i+1} megszerzi ezt a villát, ezért sikeresen belép a E szakaszba, ahogy megköveteltük. Figyeljük meg, hogy ebben az esetben a szükséges idő legfeljebb l.

3. $i+1 \in \overrightarrow{D}$.

   Ekkor l időn belül P_{i+1} eldobja a villáját, ami által a két folyamat olyan helyzetbe kerül, melyben $i+1 \in F$, következő véletlen választás *bal*, és $i \in \overleftarrow{V}$. (A P_i folyamatnak azért kell még $\overleftarrow{V}$-ben lennie, mivel addig nem szerezheti meg az első villáját, amíg P_{i+1} el nem dobja azt.) Az eredményül kapott állapot megegyezik az előző pontban vizsgálttal, így további l időn belül vagy $i \in \overleftarrow{M}$ vagy $i+1 \in E$ teljesülni fog. Tehát ebben az esetben a szükséges idő $2l$.

4. $i+1 \in \overrightarrow{M}$.

   Ebben az esetben l időn belül P_{i+1} megpróbálja elérni a baloldali villáját, és vagy közvetlenül továbbmegy E-be, miáltal teljesül a feltétel, vagy belép $\overrightarrow{D}$-be, ami visszavezet az előző esethez. Így itt a szükséges idő legfeljebb $3l$.

5. $i+1 \in \overrightarrow{V}$. Ekkor l időn belül P_i és P_{i+1} is megpróbálja elérni a közös villájukat. Amelyik ezt először próbálja, annak sikerül lefoglalnia a villát. Ha a P_i az első, akkor $i \in \overleftarrow{M}$ teljesülni fog, ahogy megköveteltük. Ellenkező

esetben $i+1 \in \overrightarrow{M}$ fog teljesülni, ami visszavezet az előző esethez. Ebből következik, hogy ebben az esetben a szükséges idő legfeljebb $4l$.

□

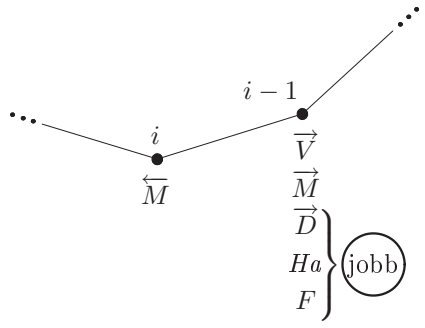
A második segédtétel arra ad felső korlátot, hogy attól a pillanattól kezdve, melyben egy folyamat kész arra, hogy megpróbálja elérni a második erőforrását, és egyik szomszédja kedvező helyzetben van, legfeljebb mennyi idő szükséges ahhoz, hogy valamelyikük elérje E-t.

11.14. segédtétel. . *Tegyük fel, hogy $i \in \overleftarrow{M}$ és vagy $i-1 \in \overrightarrow{V} \cup \overrightarrow{M}$ vagy $i-1 \in \overrightarrow{D} \cup Ha \cup F$ és a következő véletlen választás jobb. Ekkor l időn belül valamelyik folyamat belép E-be.*

Lásd a 11.12. ábrát.

Bizonyítás. A P_i folyamat l időn belül megpróbálja elérni a közös villát, megszerzi azt, és belép E-be, kivéve, ha P_{i-1} megszerezte azt időközben. De, ha P_{i-1} megszerezte, akkor ő lép be E-be, mivel a közös villa feltétlenül a második villája.

□



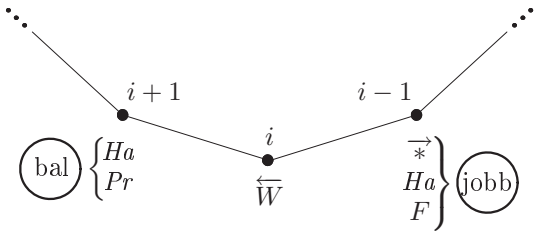
11.12.. ábra. A 11.14. állításban vizsgált helyzet.

Az utolsó segédtétel összekapcsolja az előző kettőt. Felső korlátot ad arra, hogy attól a pillanattól kezdve, hogy egy folyamat az első villájára vár, és mindkét szomszédja kedvező helyzetben van, legfeljebb mennyi időt vesz igénybe, míg valamelyikük belép E-be.

11.15. segédtétel. . *Ha $i+1 \in Ha \cup Pr$ és a következő véletlen választás bal, továbbá $i \in \overleftarrow{V}$, $i-1 \in \overrightarrow{*} \cup Ha \cup F$ és a következő véletlen választás jobb, akkor $5l$ időn belül valamelyik folyamat belép E-be.*

Lásd a 11.13. ábrát.

Bizonyítás. A 11.13. segédtétel alapján $4l$ időn belül vagy P_i belép $\overleftarrow{M}$-be, vagy P_{i+1} belép E-be. Az utóbbi esetben készen vagyunk, ezért tegyük fel, hogy P_i lép be $\overleftarrow{M}$-be.



11.13.. ábra. A 11.15. állításban vizsgált helyzet.

Ha időközben P_{i-1} belépett E-be készen vagyunk, így tegyük fel, hogy nem ez történt. Ekkor P_{i-1}-nek még $\overrightarrow{*} \cup Ha \cup F$-ben kell lennie. Továbbá, ha P_{i-1} még $\overrightarrow{D} \cup Ha \cup F$-ben is benne van, akkor a következő véletlen választása *jobb*. Ezért a 11.14. segédtételből következik, hogy további l időn belül valahány folyamat belép E-be. □

Most térjünk vissza a 11.12. lemma bizonyításához. Emlékezzünk rá vissza, hogy feltettük, hogy van egy P_i folyamat, melyre $i \in \overleftarrow{V} \cup \overleftarrow{M}$ és $i-1 \in \overrightarrow{*} \cup Ha \cup F$. Lásd a 11.6. ábra első diagrammját. Ha $i \in \overleftarrow{V}$, akkor a bizonyítás következik a 11.15. segédtételből – az $1/4$ valószínűség onnan adódik, hogy ennyi annak a valószínűsége, hogy P_{i+1} következő véletlen választása *bal*, és P_{i-1} következő véletlen választása *jobb*. Másrészről, ha $i \in \overleftarrow{M}$, akkor a bizonyítás a 11.14. segédtételből adódik – az $1/2$ valószínűség onnan következik, hogy ennyi annak a valószínűsége, hogy P_{i-1} következő véletlen választása *jobb*. □

Így megkapjuk a következő tételt.

11.16. tétel. . *A* LehmannRabin *algoritmus $n \geq 3$ esetében rendelkezik a* $\mathcal{P} \xrightarrow[1/16]{14l} \mathcal{B}$ *tulajdonsággal.*

A 11.16. tétel ismételt alkalmazásával megmutathatjuk, hogy 1 valószínűséggel valamely későbbi pontban valaki belép a *kritikus* szakaszba. Ennek megmutatását meghagyjuk gyakorlatnak (lásd 11-12. gyakorlat).

11.17. tétel. . *A* LehmannRabin *algoritmus garantálja a jólformáltságot, a kölcsönös kizárást és a független haladást. Valamint garantálja a haladást 1 valószínűséggel.*

11.5.. Megjegyzések a fejezethez

Az étkező filozófusok problémáját eredetileg Dijkstra [91] definiálta, aki kitalált egy algoritmust egy aszinkron, közös memóriájú modellre, amely tartalmaz egy globális közös szemafor változót. A JobbBalÉtkFil algoritmus folklórnak tűnik, az általánosítása, a Színez algoritmus Lynch-nek [213] köszönhető.

A LehmannRabin algoritmust Lehmann és Rabin [192] tervezte. Egy informális bizonyításvázlat megjelent [192]-ben, de nem világos, hogy ezt a vázlatot hogyan lehet formalizálni. Az itt közölt bizonyítást Lynch, Saias és Segala [208] adta közzé Pnueli és Zuck [244] hasonló stílusú korábbi bizonyítása alapján. Lehmann és Rabin [192] készített egy módosított LehmannRabin algoritmust is, amelyik nagy valószínűséggel garantálja a kizárásmentességet is.

Ebben a fejezetben valamennyi algoritmus közös változókat használ. Az erőforrás-hozzárendelési problémákkal azonban sokat foglalkoztak az aszinkron hálózati modellben is; lásd a 20. fejezetet. Például Chandy és Misra [67] adott egy megoldást egy általános erőforrás-hozzárendelési problémára aszinkron hálózatban, valamint ennek egy kiterjesztését a feladat dinamikusabb változatára, az ivó filozófusok problémájára, amelyben a folyamatok erőforrás-igényei időben változhatnak. Hasonlóan Choy és Singh [80] és Awerbuch és Saks [37] is készített erőforrás-hozzárendelési algoritmusokat aszinkron hálózatokra; ezeknek az algoritmusoknak jó az időbonyolultsága.

11.6.. Gyakorlatok

11-1. Mutassuk meg, hogy nem minden kizárási leíráshoz tartozik vele ekvivalens explicit erőforrás-leírás.

11-2. Az explicit erőforrás-leírás definícióját általánosíthatjuk megengedve alternatív erőforráslehetőségek használatát. Nevezetesen minden i-re a definíció tartalmazza a szükséges erőforrások leírását egy – az $\mathcal{R}$ halmaz feletti – monoton logikai képlet (amely csak $\vee$-okat és $\wedge$-eket tartalmaz) formájában. Egy $f(R_1, R_2, ..., R_k)$ képlet megadja az „elfogadható” erőforráshalmazok egy gyűjteményét a következő módon: az erőforrások egy S halmaza elfogadható, ha az S-beli erőforrásokhoz *igazat*, az összes többihez pedig *hamisat* rendelő kiértékelés az $f(R_1, R_2, ..., R_k)$ képlethez igaz értéket rendel. A képlet jelentése az, hogy az elfogadható erőforráshalmazok azok a halmazok, melyek kizárólagos birtoklása esetében a felhasználó beléphet a *kritikus* szakaszba.

(a) Adjunk egy általánosított explicit erőforrás-leírást, amely használható a 10. fejezet 10.13. példájában definiált k-kizárás problémájának leírására.

(b) Mutassuk meg, hogy minden általánosított erőforrás-leíráshoz létezik vele ekvivalens kizárási leírás.

(c) Mutassuk meg, hogy minden kizárási leíráshoz létezik vele ekvivalens általánosított erőforrás-leírás.

11-3. Bizonyítsuk be a 11.1. lemmát.

11-4. *Kutatási kérdés.* Definiáljuk a független haladás fogalmát úgy, hogy alkalmazható legyen általános kizárási feltételekkel kifejezett erőforrás-hozzárendelési feladatok esetében.

11-5. Általánosítsuk a 11.2. tételt az erőforrás-hozzárendelési problémák egy nagyobb osztályára, amely nem csak az étkező filozófusok problémáját tartalmazza. Próbáljuk megtalálni a lehető legnagyobb osztályt, amit tudunk.

11-6. Éles a JobbBalÉtkFil algoritmushoz adott felső időkorlát? Adjunk meg egy végrehajtási sorozatot, amelynek az időigénye olyan közel van a megadott $3c + 18l$-es időkorláthoz, amennyire csak meg tudjuk közelíteni.

11-7. Módosítsuk a JobbBalÉtkFil algoritmust úgy, hogy páratlan számú folyamat gyűrűjére működjön. Adjunk egy felső korlátot a módosított algoritmus időigényére. Legyen a korlát független n-től.

11-8. Konstruáljunk a Színez erőforrás-hozzárendelési algoritmushoz egy olyan végrehajtási sorozatot, amelynek az időigénye olyan közel van a megadott $\mathcal{O}(m^k c + km^k l)$ időkorláthoz, amennyire csak meg tudjuk közelíteni.

11-9. *Kutatási kérdés.* Konstruáljunk egy új algoritmust az ebben a fejezetben található általános erőforrás-hozzárendelési feladat egy olyan modelljére, melyben minden erőforráshoz egy, csak az erőforrást igénylő folyamatok számára elérhető olvasható-módosítható-írható változó van rendelve. Legyen az új algoritmusunknak a Színez algoritmusnál jobb időigénye.

Terjesszük ki a az algoritmusunkat úgy, hogy az erőforrás-hozzárendelési problémák több típusára is alkalmazható legyen, ne csak az ebben a fejezetben leírt két típusra.

11-10. Mutassuk meg, hogy a LehmannRabin véletlenített étkező filozófusok algoritmushoz létezik olyan ellenfél, amelyre annak a valószínűsége, hogy kizárunk egy folyamatot, nem nulla. Mekkora a legnagyobb valószínűség, amit elérhetünk?

11-11. Bizonyítsuk be a 11.11. lemma bizonyításában a 3. esetnél felhasznált segédtételt, vagyis azt, hogy az adott feltételek esetében létezik olyan P_j folyamat, melyre $j+1 \in \overleftarrow{*}$ és $j \in \overrightarrow{*} \cup Ha \cup F$.

11-12. A 11.16. tétel segítségével bizonyítsuk be a LehmannRabin algoritmusra a következőket.

(a) Minden olyan állapotból kiindulva, amelyben valamely folyamat Pr-ben van, teljesül 1 valószínűséggel, hogy a végrehajtás egy későbbi pontján valamely folyamat elkerülhetetlenül belép a Be szakaszába.

(b) Minden olyan állapotból kiindulva, amelyben valamely folyamat Pr-ben van, minden $t \geq 0$-ra teljesül, hogy $f(t)$ valószínűséggel valahány folyamat eléri Be-t t időn belül. (Nekünk kell definiálnuk f-et – próbáljuk meg olyan kicsire választani, amennyire csak lehetséges.)

11-13. Nézzük a LehmannRabin algoritmust az $n = 2$ speciális esetben. Fejezzünk ki és bizonyítsunk egy $\mathcal{P} \xrightarrow[p]{t} \mathcal{B}$ alakú érdekes állítást ebben az esetben.

11-14. A Lamer Számítástechnikai Rt. egy kezdő programozója a LehmannRabin algoritmus tanulásakor megpróbált javítani az algoritmus időigényén olymódon, hogy elhagyta az algoritmusból az első villára való várakozást. Ehelyett az ő algoritmusában a folyamat egyszerűen csak egy alkalommal megpróbálja elérni az első villáját, ahogy ezt egyébként a második villájával teszi. Ha a villa nem elérhető, akkor a folyamat visszatér a kezdetekhez és `feldob` újra.

Magyarázzuk el türelmesen a programozónak, hogy mi a rossz az algoritmusában.

11-15. *Kutatási kérdés.* Általánosíthatjuk-e a LehmannRabin algoritmust az étkező filozófusok problémájánál általánosabb erőforrás-hozzárendelési problémákra úgy, hogy továbbra is garantálja a kölcsönös kizárás és a független haladás feltételét, valamint a haladás feltételét 1 valószínűséggel?

12. fejezet

Megegyezés

Ebben a fejezetben bevezetünk egy újabb bonyodalmat, amely az általunk vizsgált aszinkron, közös memóriájú modell esetében felmerülhet, mégpedig a hiba lehetőségét. Csak a hibás folyamatok esetével foglalkozunk, a hibás memóriával nem. Valójában a folyamatoknak is csak a legegyszerűbb típusú hibáját fogjuk vizsgálni, az úgynevezett *megállási hibát*, amikor egy folyamat minden figyelmeztetés nélkül egyszer csak megáll.

Az e fejezetben tárgyalt feladat egyike a *megegyezéssel* kapcsolatos problémáknak. A szinkron, üzenetváltásos rendszerek esetében felmerülő megegyezési problémákkal az 5., 6. és 7. fejezetekben már részletesen foglalkoztunk. Folyamathibák esetében megmutattuk, hogy az alapvető megegyezési problémák megoldhatók nemcsak a megállási hibára, hanem a kevésbé jól viselkedő bizánci hibákra is. Arra vonatkozóan is adtunk ezenkívül néhány eredményt, hogy a megoldások költségei a folyamatok számának és a szükséges futási időnek a függvényében kifejezve minden esetben nagyok.

Meglepőnek tűnhet majd, hogy aszinkron környezetben a helyzet teljesen más lesz, mint az előzőekben, legalábbis olvasható/írható közös memória esetében. Nevezetesen, megmutatunk egy alapvető megoldhatatlansági eredményt, amelyik azt mondja ki, hogy az alap megegyezési feladat *egyáltalán nem oldható meg* olvasható/írható közös memória esetében, még abban az esetben sem, ha tudjuk, hogy legfeljebb egy folyamat lesz hibás. Amint később majd látni fogjuk a 17. és 21. fejezetben, ugyanez az eredmény mondható ki aszinkron hálózati környezetben is, és e két eredmény alapvető oka lényegében megegyezik.

A megegyezés megoldhatatlansága az egyik alapvető eredménye az osztott számítások elméletének. Gyakorlati következményei minden fajta osztott alkalmazásra vonatkoznak, amelyben valamiféle megegyezésre van szükség. Például egy adatbázis-kezelő rendszer folyamatainak meg kell egyezniük, hogy egy tranzakció elfogadása vagy elvetése valósuljon-e meg. Egy kommunikációs rendszer folyamatainak meg kell egyezniük, hogy egy üzenet megérkezett-e. Egy ellenőrző rendszer folyamatainak meg kell egyezniük, hogy egy másik folyamat hibás-e vagy nem. A megoldhatatlansági eredményből következik, hogy nem létezhet tisztán aszinkron algoritmus, amelyik a kívánt megegyezést elérné és a hibákkal szemben is ellenálló.

Mindez azt jelenti a gyakorlatban, hogy a tervezőknek az ilyen problémák megoldásához ki kell lépniük az aszinkron modell keretein kívülre. Például ezt megtehetik úgy, hogy az időzítésre vonatkozó információkra támaszkodnak, vagy meg kell elégedniük azzal, hogy a rendszer csak egy adott valószínűséggel lesz helyes.

12.1.. A feladat

Meghatározunk egy speciális megegyezési problémát a közös memória esetére. Tárgyalási módunk nem teljesen formális, de a 9. fejezetben megadott modell fogalmainak segítségével formalizálható. A 10. fejezet a kölcsönös kizárási feladat hasonlóan vázlatos tárgyalását tartalmazza, valamint útmutatást ad arra vonatkozóan, hogyan tehető a tárgyalásmód formálissá. Hasznos lehet az Olvasó számára, ha most előbb röviden áttekinti a 10.1. és 10.2. alfejezeteket.

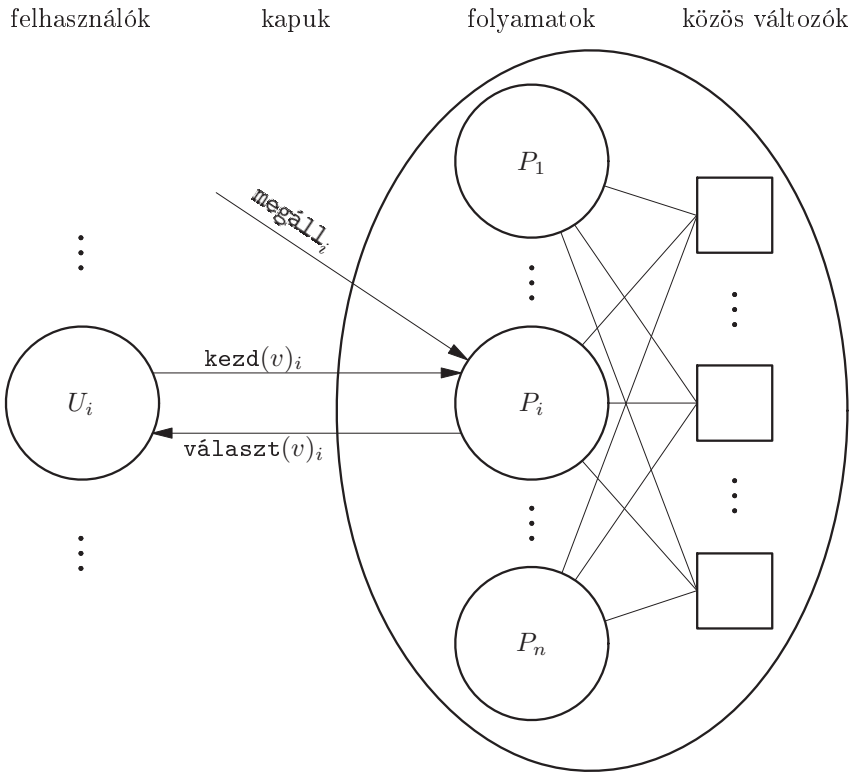
Az itt használt felépítés alapjaiban megegyezik a 9–11. fejezetekben használttal. Ebben azt feltételeztük, hogy a folyamatok a környezettel kapukon keresztül lépnek kapcsolatba, egymással pedig közös változókon keresztül kommunikálnak. Ehhez példaként tekintsük a 10.1. ábrát. Feltesszük, hogy $n \geq 2$, ahol n a kapuk száma. A folyamatoknak és változóknak az együttesét egyszerű b/k automataként modellezzük. A felhasználókat most is U_i automataként modellezzük, ahogyan azt a 10. és 11. fejezetben tettük. Tekintsük a 9.2.1. példát, mint felhasználók egy csoportját a megegyezési problémára. Feltételezzük, hogy a lehetséges bemenetek és döntések értékeinek halmaza a rögzített V halmaz, melyre $|V| \geq 2$.

Most azt fogjuk feltenni, hogy minden U_i felhasználó külső felülete a $\texttt{beállít}(v)_i$ kimeneti műveletekből és a $\texttt{dönt}(v)_i$ bemeneti műveletekből áll, ahol $v \in V$ az első esetben egy bemeneti értéke, a második esetben pedig egy döntési értéke a közös memóriájú rendszernek. A közös memóriájú rendszer külső felülete tartalmazza az összes $\texttt{beállít}(v)_i$ bemeneti műveletet – ahol most $v \in V$ egy bemeneti érték és i egy kapu neve (vagyis egy folyamat index) –, és az összes $\texttt{dönt}(v)_i$ kimeneti műveletet, ahol $v \in V$ egy döntési érték, és i egy kapu neve. Vagyis azt tételezzük fel, hogy a feladat bemenetei a felhasználóktól érkeznek bemeneti műveletek formájában. (Megjegyezzük, hogy a legtöbb publikáció a témában felteszi, hogy a kezdőértékek a kezdeti folyamatállapotoknak előre meghatározott változóiban jelennek meg, és a döntések előre meghatározott állapotváltozókba kerülnek. Az általunk használt formális leírás jobban illeszkedik ahhoz a stílushoz, amit a könyv más részeiben használtunk.) Mindegyik felhasználói automatának ki kell elégítenie a következő megszorítást: a végrehajtás során legfeljebb egy $\texttt{beállít}_i$ eseményt hajthat végre, vagyis feltesszük, hogy minden folyamat legfeljebb egy bemenetet kap.

A fentieket nem nehéz formalizálni a b/k automata fogalmainak segítségével, ahogyan az a 10. és 11. fejezetben szerepel. Ebben a fejezetben végig feltesszük, hogy minden folyamathoz pontosan egy feladat tartozik. A 8.8. gyakorlat ismeretében ez nem jelentős megszorítás.

Feltesszük továbbá, hogy a folyamatok csak a *megállási hibát* követhetik el, amin azt értjük, hogy megállhatnak minden figyelmeztetés nélkül. Formálisan ezt

úgy fogjuk modellezni, hogy bevezetünk egy speciális $\texttt{megáll}_i$ bemeneti műveletet a közös memóriájú rendszer minden folyamatához, amely műveletet a folyamatok külső felülete (külső története) fog tartalmazni. Egy $\texttt{megáll}_i$ műveletnek az lesz a hatása, hogy letiltja az i folyamat minden további helyileg ellenőrzött műveletét. A $\texttt{megáll}_i$ műveleteket nem tekintjük a felhasználói automaták külső felületének részeként, hanem úgy tekintjük, mintha azok valamilyen meg nem határozott külső környezetből érkeznének (lásd a 9.6. alfejezetet). Az imént elmondottak teljes felépítését a 12.1. ábrán láthatjuk. A hibák ilyenfajta modellezésével – a 8. fejezet formális meghatározásait is figyelembe véve – a rendszer pártatlan végrehajtásainak azokat tekintjük, amelyekben minden hiba nélküli folyamat, valamint az összes felhasználói feladat végtelenül sok helyileg ellenőrzött lépés végrehajtására kap lehetőséget. Azt mondjuk, hogy a rendszer végrehajtási sorozata hibamentes, ha nincs benne $\texttt{megáll}$ esemény.



12.1.. ábra. Közös memóriájú rendszer a megegyezési problémára.

Azt mondjuk, hogy az i felhasználó $\texttt{beállít}_i$ és $\texttt{dönt}_i$ műveleteiből álló sorozat jólformált, ha az a következő alakú sorozatnak egy kezdőszeletét alkotja: $\texttt{beállít}(v)_i$, $\texttt{dönt}(w)_i$. (Ez lehet az üres sorozat, egyetlen $\texttt{beállít}(v)_i$ művelet, vagy a $\texttt{beállít}(v)_i$, $\texttt{dönt}(w)_i$ műveletekből álló kételemű sorozat.) Ez azt je-

lenti, hogy a sorozat nem tartalmaz ismétlődő bemeneteket vagy döntéseket az i kapunál, és nem tartalmaz olyan döntést, amelyet nem előzött meg bemenet. A felhasználói automatára vonatkozó feltevéseinkből következik, hogy minden U_i megőrzi a jólformáltságot (a „megőrzés”-re vonatkozó 8.5.4. szakaszban szereplő definíciót figyelembevéve). A rendszer valamennyi végrehajtási sorozatától elvárjuk a következő tulajdonságokat (pártatlanoktól és nem pártatlanoktól egyaránt).

Jólformáltság. Minden i-re az U_i és a rendszer közötti interakciók jól formáltak.

Megegyezés. Minden döntési érték azonos.

Érvényesség. Ha minden előforduló `beállít` művelet ugyanazt a v értéket tartalmazza, akkor v az egyetlen lehetséges döntési érték.

Vegyük észre, hogy a megegyezési és érvényességi feltételek hasonlóak a 6.1. alfejezetben szereplő megfelelő feltételekhez, ahol a szinkron modell megegyezéses megállás problémáját vizsgáltuk. A fő különbség a bemenet/kimenet-megállapodásokban van.

Szükségünk van még valamilyen befejeződési feltételre. A legfontosabb elvárás hibamentes végrehajtási sorozatok esetében a következő.

Hibamentes befejeződés. Bármely pártatlan, hibamentes végrehajtásban, amelyben minden kapun szerepel `beállít` művelet, minden kapun szerepelnie kell `dönt` műveletnek is.

Azt mondjuk, hogy az A közös memóriájú rendszer az U_i felhasználók egy adott csoportjára *megoldja a megegyezési problémát*, ha biztosítja a jólformáltságot, a megegyezést, az érvényességet és a hibamentes befejeződést minden U_i számára. Azt mondjuk, hogy megoldja a megegyezési problémát, ha a felhasználók tetszőleges csoportjára megoldja azt.

Megadunk néhány erősebb befejeződési feltételt is, amelyek a hibatűréssel kapcsolatban tartalmaznak további elvárásokat. A legerősebb általunk vizsgált feltétel a következő. Ez olyan végrehajtásokra vonatkozik, amelyben tetszőleges számú folyamat lehet hibás.

Várakozásmentes befejeződés. Egy pártatlan végrehajtásban, amelyben minden kapun szerepel `beállít` művelet, minden nem hibás kapun szerepelnie kell `dönt` műveletnek is. (Vagyis minden olyan i kapun, amelyen nem szerepel $\texttt{megáll}_i$ művelet.)

Ez azt jelenti, hogy minden nem hibás folyamat előbb-utóbb döntést fog hozni, függetlenül a többi folyamat hibáitól. Ez a feltétel hasonló a 6.1. alfejezetben szereplő befejeződési feltételhez, ahol a szinkron eset megegyezéses megállás problémáját vizsgáltuk. Ezt a feltételt *várakozásmentességnek* nevezzük, mert azt eredményezi, hogy egyetlen folyamat sem lesz soha blokkolva, vagyis nem fog vég nélkül várakozni a többi folyamat segítségére.

Figyeljük meg, hogy a várakozásmentesség feltételének kimondásakor azt feltételeztük, hogy minden kapun érkezik bemenet. A feltételt azonban úgy is kimondhattuk volna, hogy csak azt tesszük fel, hogy az i kapun érkezzen bemenet. Feladatként az Olvasóra hagyjuk annak megmutatását, hogy az ilyen módon átfogalmazott definíció ekvivalens az eredetivel.

Mivel e fejezet legfontosabb eredménye, a megoldhatatlansági eredmény, arra az esetre vonatkozik, amikor csak egyetlen hibás folyamat van, szükségünk van még egy további befejeződési feltételre.

f-hibás befejeződés, $0 \leq f \leq n$. Bármely hibamentes végrehajtásban, amelyben minden kapun szerepel `beállít` művelet, és legfeljebb f kapun szerepel `megáll` művelet, minden nem hibás kapun szerepelnie kell `dönt` műveletnek is.

Azonnal látható, hogy a hibamentes és a várakozásmentes befejeződési feltételek speciális esetei az f-hibás befejeződési feltételnek, mégpedig $f = 0$, illetve $f = n$ esetében. Az *egyetlen hibás befejeződési feltétel* az a speciális eset, amikor $f = 1$.

12.1. lemma. . *Legyen A egy a fentiekben megadott felépítésű algoritmus, U_i pedig a felhasználók egy halmaza, ahol $1 \leq i \leq n$. Ekkor*

1. *Ha A várakozásmentes befejeződést garantál az U_i felhasználók számára, akkor A f-hibás befejeződést is biztosít az U_i-k számára, minden f-re, ahol $0 \leq f \leq n$.*
2. *Ha A f-hibás befejeződést garantál az U_i-k számára minden f-re, ahol $0 \leq f \leq n$, akkor A hibamentes befejeződést is biztosít az U_i -k számára.*

Azt mondjuk, hogy egy közös memóriájú rendszer *várakozásmentes befejeződést*, *f-hibás befejeződést* stb. biztosít, ha a megfelelő befejeződést biztosítja felhasználók tetszőleges csoportjára.

Történetkövetési tulajdonságok.. Ahogyan azt a 10. és 11. fejezetben tettük, ugyanúgy e fejezet helyességi feltételeit is kifejezhetjük ekvivalens módon, történetkövetési tulajdonságokkal. Minden ilyen P tulajdonságnak van egy lenyomata, amelyik `beállít`$(v)_i$ és `dönt`$(v)_i$ kimenetekből és `megáll`$_i$ bemenetekből áll. Az összetett rendszer külső műveletei szintén ugyanezek a műveletek, és a követelmény minden esetben az, hogy az egész rendszer pártatlan lenyomatai benne legyenek *történet*(P)-ben.

Szinkron befejeződési feltételek.. A várakozásmentes befejeződési feltétel ahhoz a befejeződési feltételhez hasonló, amit a 6.1. alfejezetben a megegyezéses megállási problémánál használtunk a szinkron modell esetében, továbbá ahhoz az erős befejeződési feltételhez, ami a 7.3. alfejezetben az elfogadási problémánál szerepelt. A hibamentes befejeződési feltétel a 7.3. alfejezetben szereplő gyenge befejeződési feltételhez hasonló.

A fejezet nagy részében az olvasható/írható közös memória esetével fogunk foglalkozni, hiszen ebben a környezetben lesznek igazak a megoldhatatlansági eredmények. Megengedjük, hogy a változók többszörösen írható, illetve többszörösen olvasható regiszterek legyenek. A fejezet vége felé, a 12.3. és 12.4. alfejezetben további, másfajta változótípusokat is vizsgálni fogunk majd röviden.

12.2.. Megegyezés olvasható/írható közös memóriával

Ebben az alfejezetben végig feltesszük, hogy A egy olyan algoritmus az olvasható/írható közös memóriájú modellben, amelyik megoldja a megegyezési problémát, és 1-hibás befejeződést biztosít. Az lesz a célunk, hogy ellentmondáshoz jussunk, amivel azt bizonyítjuk, hogy ilyen A nem létezhet.

Először néhány egyszerűsítő korlátozást teszünk A-ra, olyanokat, amelyek nem mennek az általánosság rovására, majd bevezetünk néhány szükséges kifejezést. Ezután a bemeneti értékekre vonatkozóan bizonyítunk be egy eredményt. Ezt követően megmutatjuk (mivel ennek bizonyítása egyszerűbb), hogy a megegyezési feladat nem oldható meg az olvasható/írható közös memóriájú modellben, ha a nagyon erős *várakozásmentes befejeződési feltételt* megköveteljük. Végül bebizonyítjuk a fejezet legfontosabb eredményét, ami azt mondja ki, hogy egyetlen hiba sem engedhető meg.

12.2.1.. Korlátozások

Az egyszerűség kedvéért, és az általánosság megsértése nélkül az alábbi négy feltételezést tesszük. Elsőként feltesszük, hogy a V értékhalmaz a 0, 1 halmaz. Másodszor, az A-t speciális felhasználók egy csoportjával együtt fogjuk vizsgálni. Ezek mind triviális automaták lesznek, amelyek egyetlen (de tetszőleges) `beállít` eseményt generálnak és semmi mást nem tesznek.

Harmadik feltételként azt tesszük fel, hogy A „determinisztikus" a következő értelemben: az automatának egyetlen kezdőállapota van; bármely automataállapotból minden folyamatnak legfeljebb egy helyileg ellenőrzött lépése van; és végül bármely automataállapotra és bármely `beállít` bemenetre egyetlen eredmény állapot létezik. A fentiek azért nem mennek az általánosság rovására, mert ha adott egy nemdeterminisztikus megoldás, akkor minden esetben egyszerűen levághatjuk egy kivételével az összes alternatívát. (Ez a fajta determinisztikusság hasonló a 8.9. gyakorlatban említetthez.)

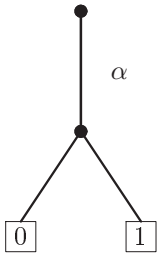
Végül a negyedik feltételezésünk az, hogy minden nem hibás folyamatnak mindig van engedélyezett helyileg ellenőrzött lépése, még az után is, hogy már esetleg döntést hozott. Ez azért nem megy az általánosság rovására, mert mindig beiktathatunk üres belső lépéseket.

12.2.2.. Terminológia

Kezdőértékek beállításának fogjuk nevezni az A-nak és olyan felhasználók együttesének egy végrehajtását, ahol a felhasználók pontosan n számú `beállít` lépésből állnak, minden kapura egyből, az indexek sorrendjében. Így egy kezdőértékek beállításának a története a következő alakú: $\texttt{beállít}(v_1)_1$,$\texttt{beállít}(v_2)_2$, ..., $\texttt{beállít}(v_n)_n$ ahol $v_1, \ldots, v_n \in V$. Egy α végrehajtási sorozatot *bemenettel kezdődőnek* fogunk nevezni, ha az kezdőértékek beállításával kezdődik. Bizonyításainkban csak ilyen, bemenettel kezdődő végrehajtási sorozatok fognak szerepelni.

Egy α véges végrehajtási sorozatot 0-*értékűnek* nevezünk, ha az α-ban, illetve az α kiterjesztéseiként előálló végrehajtási sorozatokban csak a 0 érték fordul elő

a `dönt` műveletekben. Azt, hogy a 0 érték szerepeljen valamelyik döntési műveletben, azt egyben meg is követeljük. Egy 0-értékű végrehajtási sorozat után az algoritmus már csak a 0 döntési értékhez juthat, még akkor is, ha a tényleges `dönt`(0) esemény még nem fordult elő. A fentiekhez hasonlóan α-t 1-*értékűnek* nevezünk, ha az egyetlen előforduló döntési érték az 1. Az α-ra azt mondjuk, hogy *egyértékű*, ha 0-értékű vagy 1-értékű, és *kétértékűnek* nevezzük, ha mindkét döntési érték előfordul valamely kiterjesztésében. A 12.2. ábra egy kétértékű végrehajtási sorozatot mutat.



12.2.. ábra. Az α kétértékű végrehajtási sorozat.

A következő lemma azt mondja ki, hogy a fenti osztályozás minden esetet kimerít, amennyiben hiba nem fordulhat elő. Vagyis nincs olyan véges, hibamentes végrehajtási sorozat, amely után nem fordul elő döntés.

12.2. lemma. . *A-nak minden véges, hibamentes α végrehajtási sorozata vagy egyértékű, vagy kétértékű.*

Bizonyítás. Minden ilyen α-t kiterjeszthetünk egy pártatlan, hibamentes α' végrehajtási sorozattá. Ekkor viszont az A által biztosított hibamentes befejeződési feltétel garantálja, hogy α'-ben minden folyamat előbb-utóbb dönteni fog. □

Ha α egy véges, hibamentes végrehajtási sorozat, és P_i egy folyamat, akkor legyen *kiterjesztés*(α,i) az a végrehajtás, amit úgy kapunk, hogy α-t kiegészítjük P_i egyetlen lépésével. Az, hogy ez egy valóban helyes definíció, az a korábban általunk megadott következő két korlátozáson múlik. Egyrészt feltettük, hogy minden nem hibás folyamatnak mindig van engedélyezett helyileg ellenőrzött lépése, másrészt feltettük, hogy a rendszer determinisztikus. A fenti jelölés kiterjesztését alkalmazni fogjuk folyamatindexek sorozatára is, pl. *kiterjesztés*(α,ij) = *kiterjesztés*(*kiterjesztés*(α,i),j).

12.2.3.. Kétértékű kezdőérték-beállítások

Először azt fogjuk megmutatni, hogy A-nak egy *kétértékű kezdőérték-beállítással* is kell rendelkeznie. Ezt azt jelenti, hogy csak a bemenetekből még nem határozható meg a végső döntési érték. Ha az algoritmustól nem várjuk el, hogy hiba esetében is működjön, akkor léteznek olyan egyszerű megegyezési algoritmusok,

amelyeknél a bemeneti értékek egyértelműen meghatározzák a végső döntési értéket. Ilyen algoritmusok megalkotását meghagyjuk gyakorlatnak (lásd 12-2. gyakorlat).

12.3. lemma. . *A-nak van kétértékű kezdőérték-beállítása.*

Bizonyítás. Tegyük fel az ellenkezőjét, vagyis hogy az összes kezdőérték-beállítás egyértékű.

Tekintsük az α_0 kezdőérték-beállítást, amelyik csupa 0-ból áll, ennek 0-értékűnek kell lennie az érvényességi feltétel miatt. Hasonlóképpen, a csupa 1-ből álló α_1 1-értékű kell, hogy legyen.

Most létrehozzuk kezdőérték-beállításoknak egy olyan láncát, amelyik α_0-tól α_1-ig tart.[1] A láncban minden lépésben egyetlen folyamat kezdőértékét változtatjuk meg 0-ról 1-re. Így bármely két egymást követő kezdőérték-beállítás a láncban, csak egyetlen folyamat bemenetében különbözik. A feltételezésünk szerint a láncban minden kezdőérték-beállítás egyértékű, így kell lennie két szomszédosnak, legyenek ezek α és α', úgy hogy α 0-értékű, α' pedig 1-értékű. Tegyük fel, hogy ezek a P_i folyamat kezdőértékében különböznek.

Vegyünk egy tetszőleges pártatlan végrehajtási sorozatot, amelyik α-nak egy kiterjesztése, és amelyben a P_i folyamat hibás lesz közvetlenül a kezdőérték-beállítás után (vagyis a következő művelet $\texttt{megáll}_i$), de amelyben az összes többi folyamat soha nem lesz hibás. Ekkor a P_i kivételével az összes folyamatnak döntést kell hoznia az 1-hibás befejeződési feltétel miatt. Mivel α 0-értékű, ez a döntés csak 0 lehet.

Azt állítjuk, hogy ki lehet terjeszteni α'-t is ugyanilyen módon, úgy, hogy a döntés most is 0 legyen. Ez abból következik, hogy α és α' az i kezdőértékétől eltekintve megegyezik, P_i mindkét kiterjesztésben közvetlenül a kezdőérték-beállítás után hibás lesz, így az összes többi folyamat pontosan egyformán viselkedhet α' és α után (lásd a 12.3. ábrát).

Ez viszont ellentmond annak, hogy α' 1-értékű.

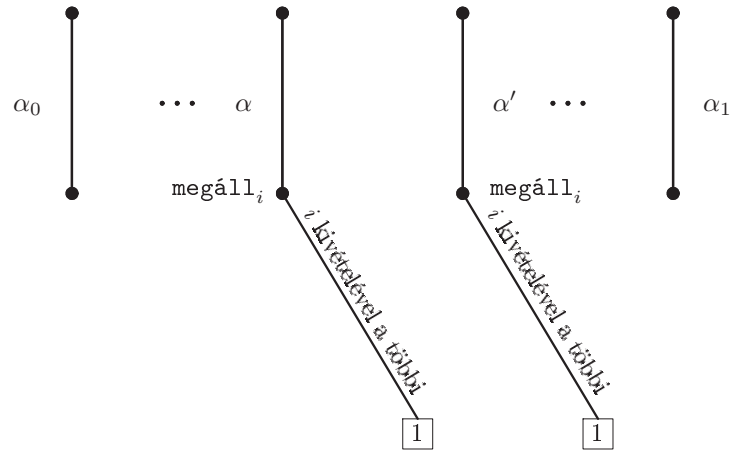
□

12.2.4.. A várakozásmentes befejeződés megoldhatatlansága

Most bebizonyítjuk az első (egyszerűbb) megoldhatatlansági eredményt, amelyik a várakozásmentes befejeződésre vonatkozik. Vagyis ebben a szakaszban azt fogjuk feltenni, hogy az A algoritmus rendelkezik a várakozásmentes befejeződési tulajdonsággal, ami erősebb a korábban feltételezett 1-hibás befejeződési feltételnél. A várakozásmentes befejeződési tulajdonság segítségével egy ellentmondásra fogunk jutni.

Az ellentmondás azon alapul, hogy meghatározunk egy olyan utat, amelyben döntést lehetne hozni. Először definiáljuk a választó fogalmát. Az α véges, hibamentes, bemenettel kezdődő végrehajtási sorozat *választó*, ha kielégíti a következő feltételeket.

[1] Ez a konstrukció hasonló a 6.33. tétel bizonyításában alkalmazotthoz, ahol a megegyezéshez szükséges menetek számára vonatkozó alsó korlátról volt szó szinkron környezetben.



12.3.. ábra. Konstrukció a 12.3. lemmához.

1. α kétértékű.
2. Bármely i-re *kiterjesztés*(α,i) egyértékű.

Vagyis egy választó végrehajtási sorozat után a döntés még nem meghatározott, de bármely további folyamatlépés (ami nem megállás) már meghatározza a döntést. Bebizonyítjuk, hogy A-nak (a várakozásmentes befejeződési tulajdonság esetében) van választó végrehajtása.

12.4. lemma. . *A-nak van választó végrehajtása.*

Bizonyítás. Tegyük fel az ellenkezőjét, vagyis, hogy bármely kétértékű, hibamentes, bemenettel kezdődő végrehajtásnak van egylépéses, kétértékű, hibamentes kiterjesztése.

Ekkor egy kétértékű kezdőérték-beállításból kiindulva (ennek létezését garantálja a 12.3. lemma), elő tudunk állítani egy végtelen, hibamentes, bemenettel kezdődő α végrehajtási sorozatot, amelynek minden véges kezdőszelete kétértékű. Ez azt jelenti, hogy α-ban a folyamatok sohasem döntenek. A konstrukció egyszerű: minden lépésben egy kétértékű, hibamentes, bemenettel kezdődő végrehajtási sorozatból indulunk ki, és azt egyetlen lépéssel egy másik kétértékű, hibamentes végrehajtási sorozattá bővítjük. A bizonyítás elején tett feltételezésünk épp azt mondja ki, hogy ezt megtehetjük.

Mivel α végtelen, valamelyik folyamat végtelen sok lépését kell tartalmaznia. Legyen ez a folyamat P_i. Azt állítjuk, hogy P_i-nek döntenie kell α-ban, ami ellentmondáshoz vezet.

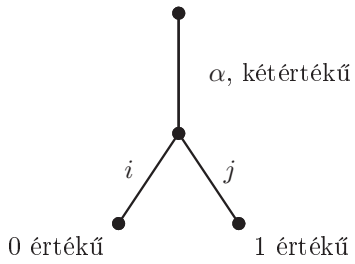
Ennek belátásához módosítsuk α-t úgy, hogy minden olyan j folyamatra, amelyik csak véges sok lépést tesz, szúrjunk be egy `megáll`$_j$ eseményt a j utolsó lépése után. A módosított végrehajtási sorozat nevezzük α'-nek. Ekkor α' egy pártatlan végrehajtási sorozat, amelyben a P_i folyamat nem lesz hibás. Így a várakozásmentes befejeződés feltételéből következik, hogy P_i-nek döntést kell hoznia

α'-ben. De az α és az α' végrehajtási sorozatok azonosnak látszanak a P_i folyamat számára, így P_i-nek α-ban is döntést kell hoznia. Ez ellentmondás, és épp erre volt szükségünk a lemma bizonyításához. □

Most eljuthatunk ahhoz az ellentmondáshoz, amire szükségünk van a várakozásmentes befejeződés megoldhatatlanságának bizonyításához.

12.5. lemma. . *A nem létezik.*

Bizonyítás. A 12.4. lemma alapján rögzíthetünk egy α választó végrehajtási sorozatot. Mivel α kétértékű, ezért létezik két olyan folyamat, legyenek ezek P_i és P_j, hogy α után P_i egy lépését véve egy 0-értékű végrehajtási sorozathoz jutunk, α után P_j egy lépését véve pedig egy 1-értékű végrehajtási sorozatot kapunk. Ez más szavakkal kifejezve azt jelenti, hogy *kiterjesztés*(α,i) 0-értékű, *kiterjesztés*(α,j) pedig 1-értékű. Nyilvánvalóan $i \neq j$. Lásd a 12.4. ábrát.



12.4.. ábra. Az α végrehajtási sorozat választó. A *kiterjesztés*(α, i) 0 értékű, a *kiterjesztés*(α, j) pedig 1 értékű.

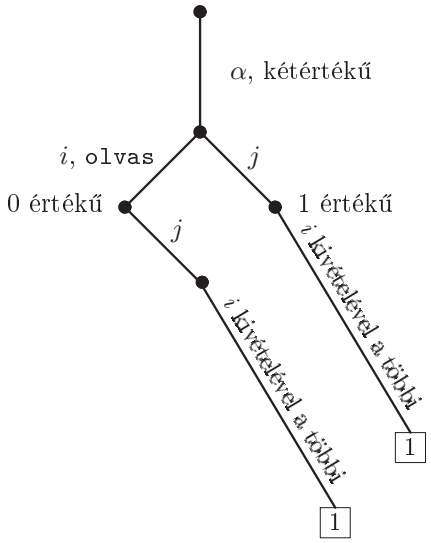
A bizonyítást a továbbiakban a lehetséges esetek elemzésével folytatjuk, úgy, hogy minden esetben ellentmondásra jussunk.

1. A P_i folyamat lépése egy olvasás.

   Vegyük egy olyan kiterjesztését *kiterjesztés*(α, j)-nek, amelyben egyetlen folyamat sem lesz hibás, a P_i folyamat nem tesz több lépést, a többi folyamat viszont végtelenül sok lépést tesz. Ez a P_i-n kívül az összes többi folyamat számára egy pártatlan végrehajtási sorozat lesz, amelyben a P_i folyamat azonnal hibás lesz, a többi folyamat viszont soha nem lesz hibás. Ezért a várakozásmentes befejeződési feltétel miatt (valójában itt elég az 1-hibás befejeződés is) P_i kivételével valamennyi folyamat döntést hoz, és mivel *kiterjesztés*(α, j) 1-értékű, e döntés eredménye csak 1 lehet.

   Vegyük észre, hogy az α és a *kiterjesztés*(α, i) utáni állapotok az i-n kívül a többi folyamat számára megkülönböztethetetlenek abban az értelemben, ahogy azt a 9.3. alfejezetben meghatároztuk. Ez abból adódik, hogy P_i lépése csupán egy olvasás, ami csak a P_i folyamat állapotát változtatja meg. Ezért vehetjük pontosan ugyanazokat a (P_j lépésével kezdődő) lépéseket, amiket az előbb α után futtattunk, és most futtathatjuk ezeket *kiterjesztés*(α, i) után. A P_i-n kívüli folyamatok ez esetben is az 1 értékű döntést

hozzák, ami ellentmond annak, hogy *kiterjesztés*(α, i) 0-értékű (lásd a 12.5. ábrát).



12.5.. ábra. Konstrukció az 1-es esethez.

2. A P_j folyamat lépése egy olvasás.

   Ez az eset teljesen hasonló az 1. esethez, és a bizonyítás megegyezik az ott elmondottakkal.

3. A P_i és a P_j folyamat lépése is egy írás.

   Ebben az esetben két alesetet különböztetünk meg.

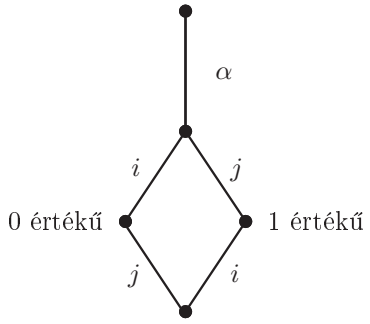
   (a) A P_i és P_j folyamatok különböző változókba írnak.

   Vegyünk két végrehajtási sorozatot, amelyek mindketten kiterjesztései α-nak. Az egyik kiterjesztés először a P_i folyamat lépését hajtsa végre, majd a P_j folyamat lépését, a másik kiterjesztés pedig fordított sorrendben hajtsa végre e két lépést. Mivel a két lépés különböző folyamatokra és különböző változókra vonatkozik, ezért a rendszer állapota mindkét végrehajtási sorozat után ugyanaz lesz (lásd a 12.6. ábrát).

   Ekkor azonban egy olyan közös rendszerállapothoz jutunk, amelyet egy 0-értékű és egy 1-értékű végrehajtási sorozat után is elérhetünk. Ha ebből az állapotból az összes folyamatot hiba nélkül futtatjuk, akkor valamennyiüknek döntést kell hozniuk. Bármelyik döntés ellentmondáshoz vezet. Ha például a 0 döntéshez jutunk, akkor az lesz a feladat, hogy ez egy olyan végrehajtási sorozatban szerepel, amelyik egy 1-értékű kezdőszelet kiterjesztéséből adódott.

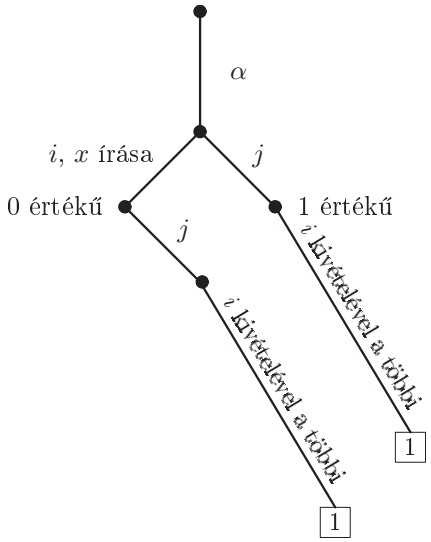
   (b) A P_i és P_j folyamatok ugyanabba a változóba írnak.

   Ugyanúgy, ahogyan az 1-es esetben, most is futtathatjuk P_i kivételével a többi folyamatot *kiterjesztés*(α, j) után, addig, amíg 1-értékű döntést hoznak. Ez esetben a *kiterjesztés*(α, j) és a *kiterjesztés*(α, ij) utáni állapotok



12.6.. ábra. Konstrukció a 3(a) esethez.

lesznek megkülönböztethetetlenek P_i kivételével a többi folyamat számára. Ennek az az oka, hogy a P_j folyamat lépése felülírja a P_i által beírt értéket, és így P_i lépésének nem marad nyoma máshol, mint a P_i folyamat állapotában. Ezért vehetjük ugyanazokat a lépéseket, amelyeket korábban *kiterjesztés*(α, j) után futtattunk, és most futtathatjuk őket *kiterjesztés*(α, ij) után. Ez esetben is P_i kivételével az összes folyamat az 1 döntést fogja hozni, ami ellentmond a *kiterjesztés*(α, i) 0-értékűségének (lásd a 12.7. ábrát).



12.7.. ábra. Konstrukció a 3(b) esethez.

Így tehát minden lehetséges esetben ellentmondáshoz jutottunk, amiből az következik, hogy a fentiekben megadott tulajdonságú A algoritmus nem létezik. □

Ezzel bebizonyítottuk az első megoldhatatlansági tételt, amely így szól.

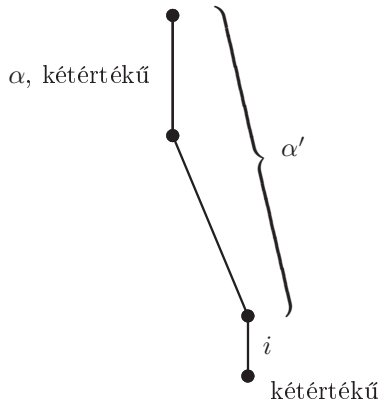
12.6. tétel. . *$n \geq 2$-re nem létezik olyan algoritmus az olvasható/írható közös memóriájú modellben, amelyik megoldja a megegyezési problémát és várakozásmentes befejeződést biztosít.*

12.2.5.. Az 1-hibás befejeződés megoldhatatlansága

Vegyük észre, hogy az előző szakaszban szereplő 12.6. tétel bizonyítása nem működik arra az esetre, ha csak az 1-hibás befejeződést tesszük fel. A gond a 12.4. lemma bizonyításánál van, ahol a várakozásmentes befejeződés feltételét használjuk annak megmutatására, hogy a P_i folyamatnak döntést kell hoznia egy olyan pártatlan végrehajtási sorozatban, ahol ő nem lesz hibás. Ebben a szakaszban egy erősebb formáját adjuk meg a 12.6. tételnek, amely a megfelelő eredményt az olyan rendszerekre mondja ki, ahol az 1-hibás befejeződés feltétele igaz.

Ezúttal a bizonyítás azon a lemmán fog alapulni, amelyik azt mondja ki, hogy egy kétértékű végrehajtási sorozat kiterjeszthető oly módon, hogy egy adott folyamatnak még egy lépést engedélyezünk, úgy, hogy a kétértékűség továbbra is fennálljon.

12.7. lemma. . *Legyen α egy kétértékű, hibamentes, bemenettel kezdődő végrehajtási sorozata A-nak, és P_i egy tetszőleges folyamat. Ekkor van olyan α' hibamentes kiterjesztése α-nak, hogy a kiterjesztés(α', i) kétértékű (lásd a 12.8. ábrát).*

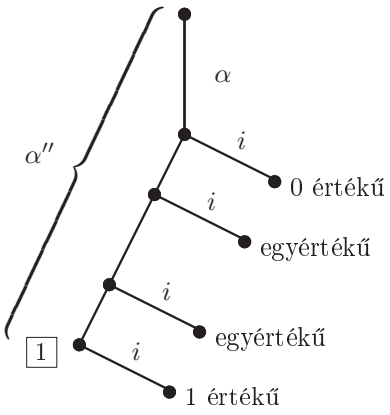


12.8.. ábra. A kétértékűség megőrzése, amíg a P_i folyamat egy további lépést tesz

Ezen a ponton megteheti az Olvasó, hogy előrelapoz a 12.8. tétel bizonyításához, hogy lássa, hogyan is következik a 12.7. lemmából a megoldhatatlansági eredmény, mielőtt elmerülne a lemma bizonyításának technikai részleteiben.
Bizonyítás. A lemmát úgy fogjuk bebizonyítani, hogy feltesszük az ellenkezőjét, vagyis, hogy a lemma nem igaz. Ekkor kell, hogy létezzen A-nak egy α kétértékű,

hibamentes, bemenettel kezdődő végrehajtási sorozata, és egy P_i folyamat, úgy, hogy α-nak minden hibamentes α' kiterjesztésére a *kiterjesztés*(α', i) egyértékű. Ez egyben azt is jelenti, hogy *kiterjesztés*(α, i) is egyértékű, tegyük fel az általánosság megszorítása nélkül, hogy ez 0-értékű.

Mivel α kétértékű, létezik olyan α" kiterjesztése, amelyikben az 1 döntés szerepel. Az általánosság megszorítása nélkül feltehetjük, hogy α" hibamentes, ellenkező esetben egyszerűen kihagyhatnánk a `megáll` műveleteket, ami nem érinti a döntést. Az eddigiekből következik, hogy a *kiterjesztés*(α", i) 1-értékű. Nézzük meg, mi történik, ha i tesz egy lépést az α-tól α"-ig tartó útvonal minden egyes pontjában (lásd a 12.9 ábrát).



12.9.. ábra. A P_i folyamat lépése egyértékűséghez vezet.

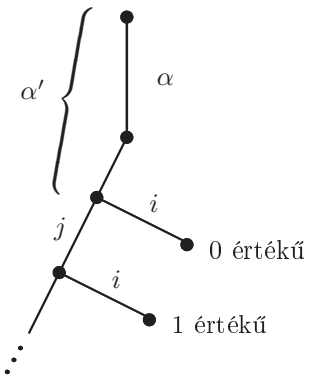
Az útvonal elején P_i lépése 0-értékűséget eredményez, a végén viszont 1-értékűséget. Minden közbenső pontban P_i lépése egyértékűséghez vezet. Ezért kell lennie két olyan szomszédos pontnak az útvonalon, hogy P_i lépése az elsőből 0-értékűséget, míg a másodikból 1-értékűséget eredményez. Legyen α' az a végrehajtási sorozat, ami az első pontig tart (lásd a 12.10. ábrát).

Tegyük fel, hogy P_j az a folyamat, amelyik a két pont közötti lépést megteszi. Először belátjuk, hogy $j \neq i$. Ennek azért kell igaznak lennie, mert ha $j = i$ volna, akkor olyan helyzet állna elő, amiből P_i egyetlen lépése 0-értékűséghez, míg P_i két lépése 1-értékűséghez vezet. Mivel a folyamatok determinisztikusak, így ez egy 0-értékű végrehajtási sorozatnak egy 1-értékű kiterjesztését eredményezné, ami lehetetlen.

A bizonyítást a lehetséges esetek vizsgálatával folytatjuk, hasonlóan, mint ahogy azt a 12.5. lemma bizonyításánál tettük, úgy, hogy minden esetben ellentmondáshoz jussunk.

1. A P_i folyamat lépése egy olvasás.

   Ekkor azt állítjuk, hogy a *kiterjesztés*(α', ji) és a *kiterjesztés*(α', ij) utáni állapotok megkülönböztethetetlenek a P_i-től különböző összes folyamat számára. Ez azért igaz, mert a fenti két kiterjesztésben P_i mindkét lépése



12.10.. ábra. Két egymást követő pont, amelyekből a P_i folyamat különböző értékekhez vezet.

olvasás, ami semmi mást nem érint a P_i folyamat állapotán kívül.

Vegyünk a *kiterjesztés*(α', ij)-nek egy olyan kiterjesztését, amelyben i nem tesz több lépést, az összes többi folyamat viszont végtelenül sok lépést teszt. Az 1-hibás befejeződési feltétel alapján az i től különböző folyamatoknak döntést kell hozniuk, és mivel a *kiterjesztés*(α', i) 0-értékű, ezért e döntésnek 0-nak kell lennie. Az imént említett megkülönböztethetetlenség miatt ugyanazokat a lépéseket, amiket *kiterjesztés*(α', ij) után futtattunk, futtathatjuk *kiterjesztés*(α', ji) után is. A folyamatok ez esetben is a 0 döntést hozzák meg, ami ellentmond a *kiterjesztés*(α', ji) 1-értékűségének (12.11. ábrát).

2. A P_j folyamat lépése egy olvasás.

   A bizonyítás hasonló az 1. eset okfejtéséhez. Ezúttal a *kiterjesztés*(α', i) és a *kiterjesztés*(α', ji) utáni állapotok lesznek megkülönböztethetetlenek a j-től különböző összes folyamat számára. Most a j-től különböző folyamatok futását fogjuk megengedni a *kiterjesztés*(α', i) után, amivel a 0 döntésre kényszeríthetjük őket. Majd ugyanezeket a lépéseket futtatjuk a *kiterjesztés*(α', ji) után. A folyamatok ismét a 0 döntésre jutnak, ami ellentmond a *kiterjesztés*(α', ji) 1-értékűségének.

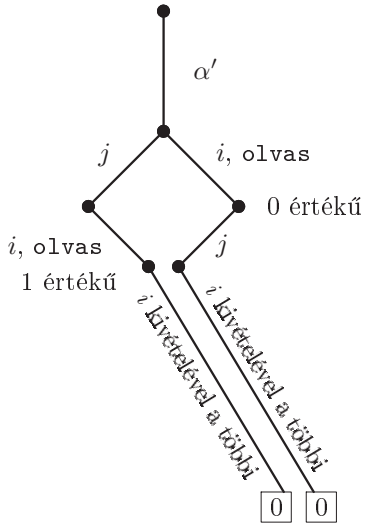
3. A P_i és a P_j folyamat lépése írás.

   (a) A P_i és a P_j folyamatok különböző változókba írnak.

   Ez esetben ugyanazon forgatókönyv szerint járunk el, mint ahogyan azt a 12.5. lemma bizonyításánál a 3(a) esetben tettük (12.6. ábrát). Itt is ugyanolyan módon jutunk ellentmondáshoz, mint abban a bizonyításban.

   (b) A P_i és P_j folyamatok ugyanabba a változóba írnak.

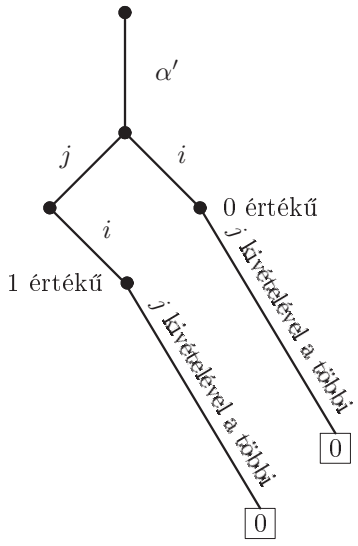
   Ez esetben a *kiterjesztés*(α', i) és a *kiterjesztés*(α', ji) utáni állapotok megkülönböztethetetlenek lesznek a P_j-től különböző összes folyamat számára, mivel a P_i lépése felülírja a P_j lépését. A P_j-től különböző folyamatokat



12.11.. ábra. Konstrukció az 1-es esethez.

kiterjesztés(α', i) és *kiterjesztés*(α', ji) után futtatva, ugyanolyan ellentmondáshoz jutunk, mint a korábbiakban (lásd a 12.11. ábrát).

□



12.12.. ábra. Konstrukció a 3(b) esethez.

Most már bebizonyíthatjuk a fejezet fő tételét.

12.8. tétel. . *$n \geq 2$-re nem létezik olyan algoritmus az olvasható/írható közös*

memóriájú modellben, amelyik megoldja a megegyezési problémát és 1-hibás befejeződést biztosít.

Bizonyítás. A 12.7. lemma segítségével létrehozunk egy pártatlan, hibamentes, bemenettel kezdődő végrehajtási sorozatot, amelyben egyetlen folyamat sem hoz döntést. Ez ellentmondást jelent majd a hibamentes befejeződés követelményével szemben.

A konstrukció egy kétértékű kezdőérték-beállítással kezdődik, amelynek a létezését a 12.3. lemma biztosítja. Ezután vég nélkül ismétlődően kiterjesztjük az aktuális végrehajtási sorozatot, úgy, hogy az első kiterjesztés legalább egy lépését tartalmazza a P_1 folyamatnak, a második kiterjesztés legalább egy lépését tartalmazza a P_2 folyamatnak, és így tovább. Ezt az eljárást folytatjuk körbekörbe haladva a folyamatokon, miközben végig megőrizzük a kétértékűséget és elkerüljük a hibák előfordulását. A 12.7. lemma biztosítja, hogy minden egyes kiterjesztési lépés lehetséges lesz.

A végeredményül kapott végrehajtási sorozat pártatlan, mivel minden folyamat végtelenül sok lépést tesz. A folyamatok egyike sem hoz viszont döntést, ami épp az elérni kívánt ellentmondáshoz vezet. □

12.3.. Megegyezés olvasható/módosítható/írható közös memóriával

Az olvasható/írható közös memória esetével ellentétben, ha olvasható/módosítható/írható közös memóriát használunk, nagyon egyszerűen megoldható a megegyezési feladat a várakozásmentes befejeződés feltételének biztosításával. Valójában egyetlen olvasható/módosítható/írható közös változó elegendő a megoldáshoz.

OMIMegegyez algoritmus
Az közös változó értéke kezdetben legyen *ismeretlen*. Minden folyamat hozzányúl a változóhoz a következő módon. Ha a változóban az *ismeretlen* értéket találja, saját kezdőértékére változtatja azt, majd döntést hoz ezzel az értékkel; ha azonban a változóban a $v \in V$ értéket találja, azt nem változtatja meg, hanem elfogadja saját döntési értékeként.

A P_i folyamat előfeltétel/hatás kódja az OMIMegegyez algoritmus esetében ugyanaz, mint ami a 9.1.1. példában szerepelt, néhány, a hibakezelésre vonatkozó sorral kiegészítve. Ezek konkrétan a következők: Az állapot egy további összetevőt is tartalmaz, a *megállt* komponenst. Ez folyamatoknak egy halmaza, amely kezdetben üres. Van egy új, `megáll`$_i$ művelet, amely a P_i folyamatot a *megállt* halmazba teszi. A `hozzáfér`$_i$ és a `dönt`$_i$ műveletek egy további előfeltétellel rendelkeznek, ami a következő: $P_i \notin$ *megállt*. Végül a `beállít`$_i$ művelet csak akkor végzi el a változtatásait, ha $P_i \notin$ *megállt*.

12.9. tétel. . *Az* OMIMegegyez *algoritmus megoldja a megegyezési problémát és várakozásmentes befejeződést biztosít.*

Bizonyítás. A bizonyítás magától értetődő. A várakozásmentes befejeződés azért lesz igaz, mert minden P_i folyamat, miután megkapta a $\texttt{beállít}_i$-től a bemenetét, azonnal végrehajthatja a $\texttt{hozzáfér}_i$ és a $\texttt{dönt}_i$ műveleteket. A megegyezés és az érvényesség abból következik, hogy a közös döntési értéket az első `hozzáfér` műveletet végrehajtó folyamat határozza meg. □

12.4.. Más típusú közös memória

A megegyezési feladat más, további változótípusokra is vizsgálható az olvasható/írható és az olvasható/módosítható/írható típusok mellett. Például vehetünk olyan változókat, amelyekre a következő műveletek valamelyikét hajthatjuk végre: `cserél`, `ellenőriz_beállít`, `olvas_hozzáad`, `összehasonlít_cserél`. Ezeket a műveleteket a 9.4.3. példában definiáltuk. Ezekkel kapcsolatban az ismertebb eredmények a következők.

12.10. tétel. . *A megegyezési feladat bármely n-re megoldható várakozásmentes befejeződéssel, ha van egy olyan közös változó, amelyre az* `összehasonlít_cserél` *művelet végrehajtási sorozata engedélyezett.*

12.11. tétel. . *$n \geq 3$ esetében a megegyezési feladat nem oldható meg várakozásmentes befejeződéssel, ha olyan közös változók adottak, amelyekre a* `cserél`*,* `ellenőriz_beállít`*,* `olvas_hozzáad`*,* `olvas` *és* `ír` *műveletek tetszőleges együttesei hajthatók végre.*

Bizonyítás. A bizonyításokat meghagyjuk gyakorlatnak (lásd 12-7. és 12-9. gyakorlat). □

12.5.. Kiszámíthatóság aszinkron közös memóriájú rendszerekben*

A megegyezési feladat csupán az egyike azoknak a „döntési" problémáknak, amelyek az aszinkron, közös memóriájú modellben felmerülnek, ahol még a megállási hiba előfordulása is megengedett. Ebben az alfejezetben megadjuk a döntési problémának egy általános meghatározását, bemutatunk néhány példát, és bizonyítás nélkül kimondunk néhány jellemző, kiszámíthatóságra vonatkozó eredményt.

A döntési problémára vonatkozó meghatározásunkat a döntési leképezés definíciójára alapozzuk majd, így először ez utóbbit adjuk meg. Legyen adott egy n hosszúságú w bemeneti vektor, amely a rögzített V értékhalmazból veszi értékeit. Ekkor egy D *döntési leképezés* minden ilyen w-re meghatároz egy nem üres $D(w)$ n hosszú döntési vektorokból álló halmazt. A w vektor tartalmazza az 1,...,n, folyamatok bemeneteit, a $D(w)$ vektorok mindegyike pedig a folyamatok döntéseit az indexek sorrendjében.

A D döntési leképezést az aszinkron, közös memóriájú rendszerben megoldandó feladat formalizálásához használjuk. A közös memóriájú rendszer külső

felülete most is $\texttt{beállít}(v)_i$, $\texttt{dönt}(v)_i$ és $\texttt{megáll}_i$ műveletekből áll, mint a megegyezési feladat esetében. A jólformáltság feltétele és a különböző befejeződési feltételek szintén ugyanúgy határozhatók meg, mint a megegyezési problémánál. Az ott használt megegyezés és érvényesség feltételek helyett azonban most csak az érvényesség feltételét követeljük meg.

Érvényesség. Bármely végrehajtási sorozat esetében, amelynél minden kapun szerepel `beállít` esemény, a folyamatok által hozott döntések vektora kiegészíthető egy $D(w)$-beli vektorrá, ahol w a bemeneti vektort jelöli.

12.5.1. példa. A megegyezési feladat, mint döntési feladat

A megegyezési feladat egy példája a döntési feladatokknak, amely az alábbiakban megadott D döntési leképezésen alapul. Legyen $w = v_1, \ldots, v_n$ egy tetszőleges V-beli értékekből álló bemeneti vektor. Ekkor a megengedett döntési vektorokat tartalmazó $D(w)$ halmazt a következőképpen határozzuk meg:

1. Ha $v_1 = v_2 = \ldots = v_n = v$, akkor $D(w)$ egyetlen $x_1, \ldots, x_n$ vektort tartalmaz, amelyre $x_1 = x_2 = \ldots = x_n = v$.
2. Ha $v_i \neq v_j$ valamely i-re és j-re, akkor $D(w)$ azokból az $x_1, \ldots, x_n$ vektorokból áll, amelyekre $x_1 = x_2 = \ldots = x_n$.

Könnyen látható, hogy a fenti D-n alapuló döntési feladat megegyezik a megállási feladattal. (Az egyetlen látszólagos különbség az az, hogy az általános döntési feladat csak olyan végrehajtási sorozatokról beszél, amelynél minden kapun szerepel `beállít` művelet, a megállási feladat definíciója viszont másfajta végrehajtási sorozatokat is tartalmaz. Ez azonban nem lényeges különbség, mivel minden véges végrehajtási sorozatot kiterjeszthetünk egy olyan végrehajtási sorozattá, amelyben minden kapura érkezik bemenet.)

Két további fontos példa a döntési feladatra a k-megegyezési feladat, és a közelítő megegyezés feladataa. Mindkettővel foglalkoztunk már a 7. fejezetben, a szinkron hálózati modell tárgyalásának keretében. A k-megegyezési feladat esetében, ahol k tetszőleges pozitív szám, a megegyezési és érvényességi feltételeket a következő feltételek váltják fel.

Megegyezés. Bármely végrehajtási sorozat esetében van a V-nek egy k elemű W részhalmaza ($|W| = k$), amely minden döntési értéket tartalmaz.

Érvényesség. Bármely végrehajtási sorozat esetében a folyamatok összes (akármelyik) döntési értéke valamelyik folyamat kezdőértékével egyezik meg.

A megegyezési feltétel gyengébb, mint a szokásos megegyezés esetében volt, mert ez most k döntési értéket enged meg a szokásos egy helyett. Az érvényességi feltétel viszont egy kicsivel erősebb, mint a szokásos megegyezés érvényességi feltétele. A k-megegyezési feladatot,mint döntési feladatot nem nehéz formalizálni. A következő eredmények mondhatók ki, amelyek bizonyítását itt most nem részletezzük, hanem azokat meghagyjuk gyakorlatnak (lásd 12-11. és 12-12. gyakorlat).

12.12. tétel. . *A k-megegyezési feladat megoldható $(k-1)$-hibás befejeződés esetében az aszinkron olvasható/írható közös memóriájú modellben, egy-író/több-olvasó regiszterek használatával.*

12.13. tétel. . *A k-megegyezési feladat nem oldható meg k-hibás befejeződés esetében az aszinkron olvasható/írható közös memóriájú modellben több-író/több-olvasó regiszterek használatával.*

A *közelítő megegyezési* feladat esetében a V értékhalmaz a valós számok halmaza, és a folyamatok valós értékű adatokat küldhetnek az üzeneteikben. Most a pontos megegyezés helyett a követelmény az, hogy a folyamatok közelítőleg egyezzenek meg, egy kicsi, pozitív ϵ tűréshatáron belül maradva. Vagyis most a megegyezési és érvényességi feltételeket a következő feltételek fogják felváltani.

Megegyezés. Bármely végrehajtási sorozat esetében bármely két döntési érték legfeljebb ϵ távolságra lehet egymástól.

Érvényesség. Bármely végrehajtási sorozat esetében a döntési értékek a kezdőértékek közelében vannak.

Ismét nem nehéz a fenti problémát egy döntési feladatként formalizálni.

12.14. tétel. . *A közelítő megegyezési feladat megoldható a várakozásmentes befejeződéssel az aszinkron olvasható/írható közös memóriájú modellben, egyíró/több-olvasó regiszterek használatával.*

Bizonyítás. A bizonyítást meghagyjuk gyakorlatnak (lásd 12-14. gyakorlat). □

A fejezetet egy olyan tétellel zárjuk, amely néhány feltételt ad meg, amelyek teljesülése esetében a döntési feladat nem oldható meg 1-hibás befejeződéssel. Ez a tétel a 12.8. tételnek egy általánosítása.

A V elemeiből álló n hosszú vektorokra definiálunk egy gráfot. A gráf csúcsai az n hosszú vektorok lesznek, az élei pedig olyan vektorokból álló párok, mely vektorok pontosan egy pozíción különböznek egymástól.

12.15. tétel. . *Legyen D egy olyan döntési leképezés, amelyhez tartozó döntési feladat megoldható 1-hibás befejeződéssel az olvasható/írható közös memóriájú modellben. Ekkor létezik olyan D' döntési leképezés, amelyre $D'(w) \subseteq D(w)$ minden w-re, és amelyre igaz az alábbi két állítás.*

1. *Ha a w és w' bemeneti vektorok pontosan egy pozíción különböznek egymástól, akkor léteznek olyan $y \in D'(w)$ és $y' \in D'(w')$ vektorok, amelyek legfeljebb egy pozíción különböznek egymástól.*
2. *Bármely w-re a $D'(w)$ által meghatározott gráf összefüggő.*

A 12.15. tétel bizonyítását meghagyjuk gyakorlatnak (lásd 12-17. gyakorlat). A bizonyítás hasonló ötleteken alapul, mint a 12.8. tétel bizonyítása.

12.6.. Megjegyzések a fejezethez

Hibára hajlamos rendszerek esetében a megegyezésre vonatkozó első megoldhatatlansági eredményt Fischer, Lynch és Paterson [123] bizonyította. Ez az eredmény az aszinkron üzenettovábbító rendszerekre vonatkozott 1-hibás befejeződés esetére. Később ezt az eredményt és a bizonyítását Loui és Abu-Amara [199]

kiterjesztették az aszinkron közös memóriájú környezetre, ami egy kicsivel erősebb modell. (Az aszinkron közös memória modellje és az aszinkron hálózati modell kapcsolatára vonatkozóan lásd a 17. fejezetet.) A várakozásmentes megegyezés megoldhatatlanságára vonatkozó eredményt Loui és Abu-Amara [199], illetve Herlihy [150] bizonyította be egymástól függetlenül. E fejezet tárgyalása a [199]-beli bizonyításokat követi.

A megegyezésre vonatkozó olyan eredmények, ahol az olvasható/írható típustól eltérő típusú közös változókat használunk, Herlihy [150] nevéhez fűződnek. Herlihy munkája nem csak osztályozza a változókat aszerint, hogy melyekkel lehet a megegyezési problémát megoldani, hanem azt is megadja, mely típusokkal mely más típusokat lehet megvalósítani.

A k-megegyezési problémáját eredetileg Chaudhuri [73] vetette fel aszinkron hálózati környezetben. Chaudhuri bebizonyította, hogy a k-megegyezés megoldható $(k-1)$-hibás befejeződés esetében, de az a kérdés, hogy vajon megoldható-e k-hibás befejeződés esetében is, évekig nyitott volt. A kérdésre vonatkozó (negatív) válasz egyidejűleg jelent meg Herlihy és Shavit [152], Borowsky és Gafni [55], valamint Saks és Zaharoglou [253] cikkeiben. Herlihy és Shavit az általuk elért eredményekben egy topológikus jellemzését adták meg azoknak a problémáknak, amelyek megoldhatók a hibára hajlamos, aszinkron olvasható/írható közös memóriájú rendszerekben. Ezt a jellemzést [151]-ben továbbfejlesztették. A jellemzésben vizsgálták a bemeneti vektorok szűkített halmazait is, ellentétben könyvünkkel, ahol csak a teljes halmazokat vettük figyelembe.

A közelítő megegyezési problémáját aszinkron rendszerekben eredetileg Dolev, Lynch, Pinter, Stark és Weihl [98] írták le. Az ő munkájuk az aszinkron hálózati modellre vonatkozott. A közelítő megegyezésre egy várakozásmentes, aszinkron, közös memóriájú algoritmust Attiya, Lynch és Shavit [24] dolgozott ki.

Biran, Moran és Zaks [51] jellemezte azokat a döntési problémákat, amelyek megoldhatók aszinkron, olvasható/írható közös memóriájú környezetben 1-hibás befejeződéssel. Ez a munka egy korábbi megoldhatatlansági eredményen alapult, amelynek szerzői Moran és Wolfstahl [230] voltak. Az említett jellemzés vizsgálja a bemeneti vektorok szűkített halmazait is. A 12.15. tétel e fenti két cikkbeli eredményekből következik. Ezeket az eredményeket eredetileg aszinkron hálózati környezetre bizonyították be, de a bizonyítások érvényesek az aszinkron olvasható/írható közös memóriájú rendszerekre is.

Többen megadtak véletlenen alapuló megoldásokat a megegyezési problémára olvasható/írható közös memória használatával, többek között Chor, Israeli és Lewis [78], Abrahamson [2], valamint Aspnes és Herlihy [16].

12.7.. Gyakorlatok

12-1. Mutassuk meg, hogy a várakozásmentes befejeződés erősebb formája, amelyben csak azt követeljük meg, hogy az i kapura érkezzen bemenet, ekvivalens a korábban megadott feltétellel. Konkrétan mutassuk meg, hogyan kell egy adott A algoritmust módosítani, amelyik biztosítja a jólformáltságot, a megegyezést, az érvényességet és a várakozásmentes befejeződést, úgy, hogy a módosított vál-

tozat is biztosítsa ezeket a feltételeket, de a várakozásmentességnek az erősebb formáját biztosítsa.

12-2. Adjunk meg egy algoritmust, amely megoldja a megegyezési problémát (hibatűréssel kapcsolatos követelmények nélkül) az olvasható/írható közös memóriájú modellben és eleget tesz a következő feltételnek is:
(a) van kétértékű kezdőérték-beállítása.
(b) minden kezdőérték-beállítása egyértékű.

12-3. Igaz vagy hamis az alábbi állítás?
(a) Ha A egy (nem hibatűrő) megegyezési protokoll az olvasható/írható közös memóriájú modellben, amely kielégíti a 12.1. alfejezetben megadott követelményeket, továbbá van kétértékű kezdőérték-beállítása, akkor A-nak van választó végrehajtási sorozata.
(b) Ha A teljesíti az (a)-ban megadott feltételeket, továbbá igaz rá, hogy ő két folyamatból álló protokoll, akkor A-nak van választó végrehajtási sorozata.

12-4. Mutassuk meg, hogy a 12.8. tétel igaz marad akkor is, ha megváltoztatjuk a követelményeket oly módon, hogy az érvényességi feltételt az alábbi gyengébb feltétellel helyettesítjük: létezik két, bemenettel kezdődő végrehajtási sorozat, α_0 és α_1, amelyekre igaz, hogy az α_0-ban valamelyik folyamat 0 döntést hoz, α_1-ben pedig valamelyik folyamat 1 döntést hoz.

12-5. Tekintsük ismét a megegyezési problémát olvasható/írható közös memória használatával. Vegyünk most a hibákra vonatkozóan egy szigorúbb modellt, mint az általános megállási hiba esetében, ami most azt jelenti, hogy a folyamatok csak a számításaik legelején lehetnek hibásak. (E szerint a `megáll` eseményeknek meg kell előzniük minden más eseményt.) Megoldható-e a megegyezési feladat ebben a modellben
(a) 1-hibás befejeződéssel?
(b) várakozásmentes befejeződéssel?
Mindkét esetben vagy egy algoritmust adjunk meg, vagy a megoldhatatlanság bizonyítását.

12-6. Mutassuk meg, hogy az olvasható/módosítható/írható közös memóriájú modellben minden megegyezési protokollnak, amelyik 1-hibás befejeződést biztosít, kétértékű kezdőérték-beállítással kell rendelkeznie.

12-7. Bizonyítsuk be a 12.10. tételt.

12-8. Mutassuk meg, hogy $n \geq 3$-ra az n folyamatra vonatkozó megegyezési feladat várakozásmentes befejeződéssel nem oldható meg, tetszőlegesen sok változó használatával sem, amennyiben a változók típusai hatással vannak egymásra a következő értelemben. Ha a-val és b-vel jelöljük a változótípusokat, $f_2(a, v)$-vel pedig $f(a, v)$ második vetületét (vagyis a változó új értékét), legalább egy teljesül az alábbi feltételek közül:
(a) (a és b kommutatív) $f_2(a, f_2(b, v)) = f_2(b, f_2(a, v))$ minden $v \in V$-re.
(b) (a felülírja b-t) $f_2(a, f_2(b, v)) = f_2(a, v))$ minden $v \in V$-re.
(c) (b felülírja a-t) $f_2(b, f_2(a, v)) = f_2(b, v))$ minden $v \in V$-re.
A fentiekben a 9.4. alfejezet jelöléseit használtuk.

12-9. A 12.8. gyakorlat eredményének felhasználásával bizonyítsuk be a 12.11. tételt.

12-10. Fejezzük ki az alábbiakat formálisan egy döntési feladatként, úgy, hogy megadjuk a döntési leképezéseket:
(a) a k-megegyezési feladat;
(b) A közelítő megegyezési feladat.

12-11. Bizonyítsuk be a 12.12. tételt.

12-12. Bizonyítsuk be a 12.13. tételt. (Vigyázat, ez nagyon nehéz feladat.)

12-13. Tekintsük a közelítő megegyezési problémát $n = 2$-re, vagyis két folyamat esetében. Adjunk meg egy várakozásmentes algoritmust az aszinkron, közös memóriájú modellben, egy-író/több-olvasó regiszterek használatával. Bizonyítsuk be az algoritmus helyességét, és elemezzük az időbonyolultságát.

12-14. Általánosítsuk a 12.13. gyakorlat eredményét tetszőleges n-re, vagyis bizonyítsuk be a 12.14. tételt.

12-15. A Pehely Számítógépes Vállalat egy gyakorlott programtervezője egy okos ötlettel áll elő, amivel e fejezet megegyezési feladatát szeretné megoldani tetszőleges számú megállási hiba esetében, $V = \{0,1\}$-re. Ötletének lényege, hogy veszi a 0 és 1 számokat, mint valós számokat, és az ezekre vonatkozó közelítő megegyezési feladat várakozásmentes megoldását „alprogramként" használja. Vagyis amikor a folyamatok megadják a válaszaikat a közelítő megegyezési problémára, utána egyszerűen vehetik a 0 és 1 közül a megoldásukhoz közelebb esőt, és ez lesz a végső döntésük.
Magyarázzuk meg, hogy mi a hiba a fenti gondolatmenetben.

12-16. Vegyük a két folyamatra vonatkozó várakozásmentes közelítő megegyezési algoritmust, amit a 12.13. gyakorlat megoldásaként kaptunk. (Feltesszük, hogy ez az algoritmus „determinisztikus" a 12.2.1. szakaszban megadott értelemben.) Tetszőleges (v_1, v_2) bemeneti vektor esetében definiáljuk meg a $D'(v_1, v_2)$-t azoknak a döntési vektoroknak a halmazaként, amelyeket olyan, bemenettel kezdődő végrehajtási sorozatokra kapunk, ahol az 1-es folyamat bemenete a v_1, a 2-es folyamat bemenete pedig v_2.
(a) Adjuk meg a $D'(0, 1)$ halmazt és azt a hozzá tartozó gráfot, amit a 12.15. tétel előtt definiáltunk.
(b) Vegyük azt a hibamentes, végtelen hosszú végrehajtási sorozatot, amelyben az 1-es és a 2-es folyamat először a 0, illetve az 1 bemenetet kapja meg, majd a folyamatok lépései váltakozva követik egymást (1,2,1, ...). E végrehajtási sorozat minden bemenettel kezdődő α kezdőszeletére adjuk meg a döntési vektoroknak azt a halmazát, amelyet az α kiterjesztéseiben megkapunk.
(c) Adjuk meg a $D'(v_1, v_2)$ halmazt minden egyes (v_1, v_2) bemeneti vektorra.
(d) Mutassuk meg, hogy bármely (v_1, v_2)-re a $D'(v_1, v_2)$ által meghatározott gráf összefüggő.

12-17. Bizonyítsuk be a 12.15. tételt. (*Útmutatás.* Rögzítsünk egy tetszőleges A algoritmust, amely megoldja D-t 1-hibás befejeződéssel az olvasható/írható

közös memóriájú modellben. Minden w bemeneti vektorra definiáljuk $D'(w)$-t azoknak a döntési vektoroknak a halmazaként, amelyeket ténylegesen megkapunk A-nak azokban a bemenettel kezdődő végrehajtási sorozataiban, amelyek bemeneti vektora w. A tétel 1. állításának bizonyításához használjunk hasonló gondolatmenetet, mint amit a 12.3. lemmánál alkalmaztunk. A 2. állítást indirekt módon bizonyítsuk, a 12.8. tételhez hasonló módon. Ezúttal minden véges, hibamentes, bemenettel kezdődő végrehajtási sorozatra vizsgáljuk meg, hogy a lehetséges döntési vektorok halmaza összefüggő-e vagy sem. Majd egy, a 12.7. lemmához hasonló, segédállítást használjunk, ami azt mondja ki, hogy bármely „nem összefüggő" α kiterjeszthető egy másik „nem összefüggő" α'-vé, miközben bármely adott P_i folyamat tehet egy lépést.)

12-18. A 12.15. tétel segítségével bizonyítsuk be a 12.8. tételt.

12-19. A 12.15. tétel segítségével mutassuk meg, hogy az általános megegyezési feladaton kívül további más döntési feladatok sem oldhatók meg az aszinkron, olvasható/írható közös memóriájú rendszerekben 1-hibás befejeződéssel. Adjunk meg minél több érdekes feladatot, amelyekre a megoldhatatlanság így bizonyítható.

12-20. Terjesszük ki a 12.15. tétel érvényességét és adjuk meg egy általános jellemzését azoknak a döntési feladatoknak, amelyek megoldhatók az aszinkron olvasható/írható közös memóriájú rendszerekben 1-hibás befejeződéssel. (Vigyázat, ez nagyon nehéz feladat.)

13. fejezet

Atomi objektumok

Ebben a fejezetben, amely az aszinkron közös memóriájú modell utolsó fejezete, bevezetjük az *atomi objektum* fogalmát. Egy adott típusú atomi objektum nagyon hasonló az ugyanilyen típusú közönséges közös változóhoz. A különbség az, hogy az atomi objektumhoz egyidejűleg több folyamat is hozzáférhet, míg a közös változóhoz való hozzáférés oszthatatlan. Annak ellenére, hogy a hozzáférés egyidejű, az atomi objektumok biztosítják, hogy a folyamatok választ kapjanak, ami azt eredményezi, hogy úgy látszik, mintha a hozzáférések időben egyenként, olyan szekvenciális sorrendben történnének, amely megegyezik a kérések és válaszok sorrendjével.

Az atomi tulajdonság mellett a legtöbb tanulmányozott atomi objektum rendelkezik érdekes hibatűrő tulajdonsággal is. Ezek közül a legerősebb a várakozásmentes megállás feltétele, ami azt mondja, hogy minden nem hibázó kapura érkező kérés végül választ kap. Ez a feltétel gyengíthető úgy, hogy csak az olyan kérés kapjon választ, amely kapuk egy kitüntetett I halmazába eső kapura, vagy egy bizonyos f számú kapura érkezik. Az egyetlen hibatípus, amivel ebben a fejezetben foglalkozunk, az a megállási hiba.

Az atomi objektumokat többprocesszoros rendszerek építésének építőelemeiként javasolták. Az elv az, hogy kiindulhatunk elemi atomi objektumokból, mint amilyenek az egyedi-író/egyedi-olvasó olvasható/írható atomi objektumok, amelyek elég egyszerűek ahhoz, hogy hardver szinten megvalósíthatók legyenek. Aztán ezekből az atomi objektumokból kiindulva építhetünk egyre bonyolultabb atomi objektumokat. Az így keletkező rendszerek szerkezete egyszerű, moduláris és helyességük bizonyítható. A feladat, ami még megoldásra vár, hogy olyan atomi objektumokat építsünk, amelyek eléggé gyors választ biztosítanak és így a gyakorlatban hasznosak.

Az atomi objektumok vitán felül hasznosak, azonban mint az aszinkron hálózati rendszerek építőelemei. Sok osztott hálózati algoritmust terveztek arra, hogy a felhasználó számára biztosítson olyan lehetőséget, mint egy központosított, koherens közös memória. Formálisan, ezek közül több tekinthető atomi objektumok osztott megvalósításának. Később látunk erre példákat a 17.1. alfejezetben és a 18.3.3. szakaszban.

A 13.1. alfejezetben az atomi objektumok tanulmányozásának formális kere-

tét adjuk. Vagyis megadjuk az atomi objektumok definícióját és azok alapvető tulajdonságait, különösen az azonos típusú közös változókhoz való viszonyára vonatkozó eredményeket és olyan eredményeket, amelyek megmutatják, hogy hogyan használhatók rendszerek építésében.

A fejezet befejező részében algoritmusokat adunk adott típusú atomi objektumok megvalósítására más típusú atomi objektumokkal (vagy ezzel egyenértékűen, közös változókkal). Atomi objektum típusok közül az olvasható/írható, olvasható/módosítható/írható és fénykép típusú objektumokat vizsgáljuk. Az itt közölt eredmények csak példák – sok hasonló eredmény található a kutatási irodalomban, és még több elvégzendő kutatás van.

13.1.. Alapfogalmak és eredmények

Először megadjuk az atomi objektumok definícióját és alapvető tulajdonságaikat. Aztán megadjuk az adott típusú kanonikus várakozásmentes atomi objektum konstrukcióját, majd néhány alapvető eredményt bizonyítunk atomi objektumok összekapcsolására és közös változók helyettesítésére közös memóriájú rendszerekben. Ezek az eredmények felhasználhatók annak igazolására, hogy atomi objektumok más atomi objektumokból hierarchikusan felépíthetők. A fejezet több fogalma meglehetősen bonyolult. Fontosak azonban, nem csak ezen fejezet eredményei miatt, hanem a 17. és 21. fejezetben tárgyalt hibatűrés miatt is. Ezért lassan haladunk és az elképzeléseket a szokásosnál formálisabban adjuk meg. Az első olvasáskor az olvasó átugorhatja a bizonyításokat és csak a definíciókat és eredményeket olvassa el. Valójában kezdheti a 13.1.1. szakasz definícióinak elolvasásával, majd átugorhat a 13.2. alfejezetre, azután szükség szerint visszatérhet hivatkozásokkor ehhez az alfejezethez.

13.1.1.. Atomi objektum definíciója

Az atomi objektum meghatározása a 9.4. alfejezetben bevezetett változótípus fogalmán alapszik. Most újraolvashatjuk ezt a részt. Emlékeztetünk, hogy a változótípus tartalmaz egy V értékhalmazt, egy v_0 kezdőértéket, *kérések* és *válaszok* halmazát és az $f : \mathit{kérések} \times V \to \mathit{válaszok} \times V$ függvényt. Ez a függvény adja meg, hogy adott kérés és változó érték esetében mi a válasz és mi a megváltozott érték.

Arra is emlékeztetünk, hogy egy változótípus *végrehajtási sorozat* az olyan $v_0, a_1, b_1, v_1, a_2, \cdots, v_r$ véges, vagy $v_0, a_1, b_1, v_1, a_2, \ldots$, végtelen sorozatok, ahol az a-k kérések, a b-k válaszok és az egymás utáni négyesek az f függvény alkalmazásával kaphatók. Továbbá, a változótípus *történetei* olyan sorozatok, amelyeket végrehajtásokból kapunk, ha belőlük csak az a és b értékeket vesszük. A $\mathcal{T}$ változótípusú A atomi objektumot egy b/k automataként definiáljuk (felhasználva a b/k automata 8. fejezetben adott általános definícióját), amely teljesíti a következő oldalakon ismertetendő feltételeket. Különösen rendelkeznie kell egy külső felülettel (külső lenyomattal) és ki kell elégítenie bizonyos „jólformáltság" és „atomiság", valamint élénkségi feltételeket.

A külső felület leírásával kezdjük. Feltesszük, hogy A az 1-től n-ig sorszámozott n kapun keresztül érhető el. Minden i kapuhoz hozzátartoznak a_i bemenet műveletek, ahol a a változótípus kérése és b_i kimenet műveletek, ahol b a változótípus válasza. Ha a_i bement művelete az i kapunak, akkor ez azt jelenti, hogy a megengedett kérés az i kapun, ha b_i kimenet művelete az i kapunak, akkor ez azt jelenti, hogy b megengedett válasz az i kapun. Egy technikai feltétellel élünk: ha a_i bemenet az i kapun és $f(a,v) = (b,w)$ valamely v és w értékekre, akkor az i kapun b_i kimenetnek kell lennie. Tehát ha az a kérés megengedett az i kapun, akkor valamennyi a-hoz tartozó válasz is megengedett az i kapun.

Ráadásul, mivel vizsgáljuk az atomi objektumok megállási hibatűrését, feltételezzük, hogy minden i kapunak van $\texttt{megállít}_i$ bemenete. A külső felületet a 13.1. ábra szemlélteti.

13.1.1. példa. olvasható/írható atomi objektum külső felülete

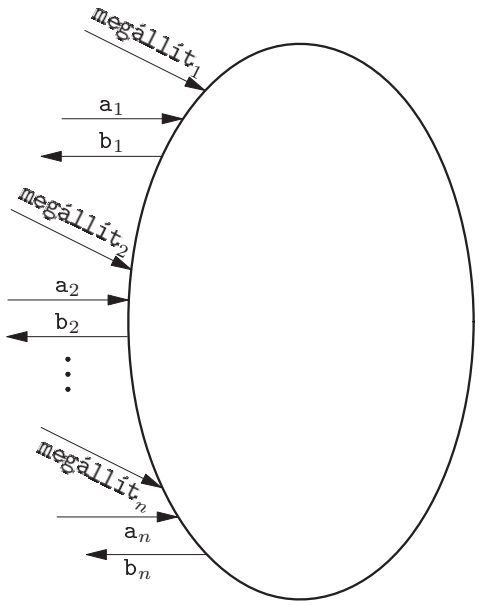
Megadjuk az *1-író/2-olvasó* V tartományú atomi objektum külső felületét. Az objektumnak három kapuja van, 1, 2 és 3 címkével ellátva. Az 1. kapu *író* kapu, csak írás műveletek megengedettek, míg a 2. és 3. kapuk *olvasó* kapuk, csak olvasó műveletek megengedettek. Pontosabban, az 1. kapuhoz tartoznak a $\texttt{ír}(v)_1$, $v \in V$ bemenet műveletek és az egyetlen $\texttt{nyugtáz}_1$ kimenet művelet. A 2. kapu egyetlen bement művelete az $\texttt{olvas}_2$ és minden $v \in V$ értékre a v_2 kimenet művelet. A 3. kapu műveletei hasonlók. Vannak továbbá a $\texttt{megállít}_1$, $\texttt{megállít}_2$ és $\texttt{megállít}_3$ bemenet műveletek a megfelelő kapukhoz. A külső felületet a 13.2. ábra szemlélteti.

Ezután megadjuk a $\mathcal{T}$ változótípusú A atomi objektum automata megkövetelt viselkedését. Csakúgy, mint a 10–12. fejezetekben, feltételezzük, hogy A az egyes kapukhoz rendelt U_i felhasználói automaták kollekciója. Az U_i automata kimenetei az i-edik kapu kérései, a bemenetei pedig az i-edik kapu válaszai. A $\texttt{megállít}_i$ műveletek nem elemei az U_i lenyomatának, nem az U_i generálja ezeket, hanem valamilyen nem meghatározott külső forrásból erednek.

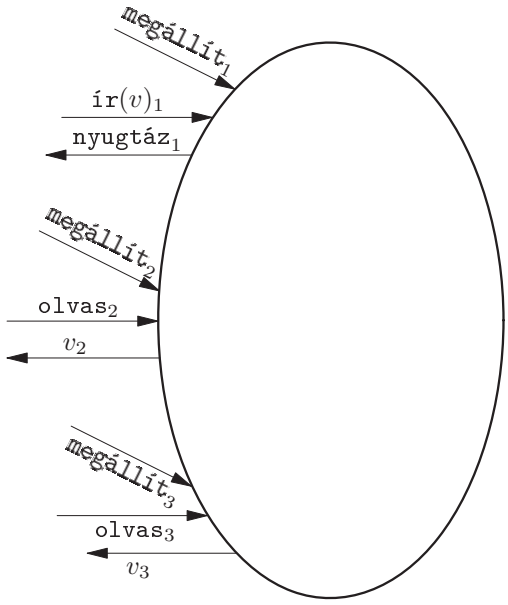
Az egyetlen további tulajdonság, amit U_i-től megkövetelünk, hogy megőrzi a következőkben definiált „jólformáltság" feltételt. Az U_i külső műveleteinek egy sorozatát jólformáltnak nevezzük az i felhasználó számára, ha kéréssel kezdődik és váltakozva vannak a kérések és válaszok. Feltételezzük, hogy minden U_i megőrzi a jólformáltságot (a 8.5.4. szakaszban adott formális definíció szerint). Tehát feltesszük, hogy minden kapu szigorúan szekvenciális a kérés műveletekre nézve, mindegyik várakozik a megelőző kérés művelet megválaszolására. Megjegyzendő, hogy a szekvencialitás csak az egyedi kapukra vonatkozik, a különböző kapukra egyidejűleg érkezhet kérés művelet.[1] A fejezet további részében az U jelölést használjuk a felhasználói U_i automaták összekapcsolásának jelölésére, azaz $U = \prod U_i$.

Megköveteljük, hogy $A \times U$, az A és az U komponensekből összetett rendszer

[1] A gyakorlatban megengedhetnénk, hogy az egyedi kapukra is érkezhessen egy időben több kérés is. Ez megkívánná a fejezetben tárgyalt elmélet kibővítését, de mi elkerüljük az ezzel járó bonyodalmakat és így az alapvető elképzeléseket meglehetősen egyszerűen tálalhatjuk.



13.1.. ábra. Egy atomi objektum külső felülete.



13.2.. ábra. 1-író/2-olvasó olvasó/író atomi objektum külső felülete.

több tulajdonsággal rendelkezzen. Először a jólformáltság feltételt nézzük, amely hasonló a 10., 11. és 12. fejezetekben használt feltételekhez.

Jólformáltság. Az $A \times U$ minden végrehajtási sorozatára és minden i-re az A és az U_i közötti kölcsönhatás jólformált az i számára.

Mivel már feltételeztük, hogy a felhasználók megőrzik a jólformáltságot, ami azt eredményezi, hogy A szintén megőrzi a jólformáltságot. Ez azt jelenti, hogy az $A \times U$ összetett rendszerben a kérések és válaszok váltakozva érkeznek minden kapura, és először kérés érkezik.

A legnehezebben megérthető feltétel a következő. Ez a műveletek látszólagos atomiságát fejezi ki egy adott $\mathcal{T}$ változótípusra. Megjegyzendő, hogy $\mathcal{T}$ egy története kérések egy sorozatára adott helyes válaszokat írja le, amikor is a műveletek szekvenciálisan hajtódnak végre, vagyis minden kérés megvárja az előző kérdés megválaszolását. Az atomiság feltétel azt eredményezi, hogy ha egy összetett rendszer megengedi kérés műveletek egyidejű megjelenését különböző kapukon, akkor minden története „olyannak látszik", mintha $\mathcal{T}$ története lenne.

Ennek formális kifejezése a vártnál kicsit bonyolultabb, mivel olyan feltételt akarunk, amelynek van értelme az $A \times U$ olyan végrehajtási sorozataira is, amikor néhány kérés – bizonyos kapukon az utolsó – nem teljes, azaz nincs rá válasz. Tehát feltesszük, hogy minden végrehajtás olyannak látszik, mintha a befejezett és *néhány nem befejezett* művelet intervallumuk egy pillanatában hajtódnának végre.

Az $A \times U$ rendszer atomiságának definiálása érdekében először egy még elemibb definíciót adunk a felhasználói műveletek atomiságára. Nevezetesen, tegyük fel, hogy β az $A \times U$ külső műveleteinek egy (véges vagy végtelen) sorozata, amely jólformált minden i-re (tehát minden i-re $\beta|\mathit{külső}(U_i)$ jól-formált). Azt mondjuk, hogy β teljesíti az *atomi tulajdonságot* $\mathcal{T}$-re, ha az alábbiak mindegyike elvégezhető:

1. minden befejezett π műveletre szúrjunk be egy $*_\pi$ *sorbarendező* pontot π kérése és válasza közé β-ban;
2. válasszuk ki befejezetlen műveletek egy Φ halmazát;
3. minden $\pi \in \Phi$ műveletre vegyük a választ;
4. minden $\pi \in \Phi$ műveletre szúrjunk be egy $*_\pi$ *sorbarendező* pontot π kérése és válasza közé β-ban.

A műveleteket és válaszokat, valamint a sorbarendező pontokat meg kell tudni úgy megválasztani, hogy az alábbi módon megadott sorozat $\mathcal{T}$ egy története legyen:

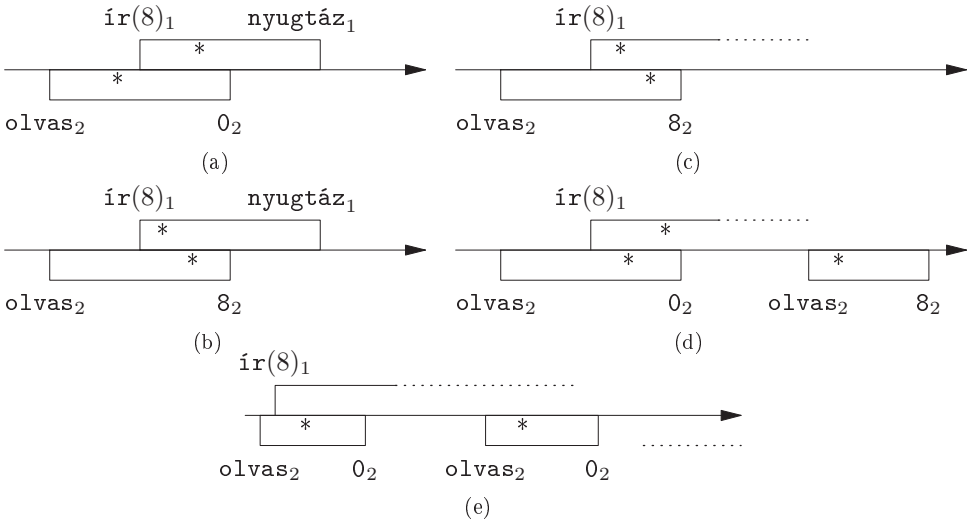
> Minden befejezett π művelet kérését és válaszát mozgassuk el a π_* sorbarendező pontjáig a β-beli előfordulások sorrendjében. (Tehát „zsugorítsuk" π intervallumát a sorbarendező pontjára.) Hasonlóan, minden $\pi \in \Phi$ műveletre tegyük a kérést és a hozzá kiválasztott választ a π_* pontra. Végül töröljük az összes $\pi \notin \Phi$ befejezetlen műveletet.

Jegyezzük meg, hogy az atomiság feltétel csak a kérés és válasz eseményektől függ – nem említi a `megállít` eseményeket. Könnyen kiterjeszthetjük ezt a definíciót

A és $A \times U$ végrehajtási sorozataira. Nevezetesen, tegyük fel, hogy α egy ilyen végrehajtási sorozat, amely jólformált minden i-re (tehát minden i-re $\alpha|\mathit{külső}(U_i)$ jólformált i-re.) Azt mondjuk, hogy α kielégíti az atomi tulajdonságot $\mathcal{T}$-re, ha $\mathit{történet}(\alpha)$, azaz a külső műveleteinek a sorozata kielégíti az atomi tulajdonságot $\mathcal{T}$-re.

13.1.2. példa. Végrehajtás sorbarendező pontokkal

A 13.3. ábra az egyedi-író/egyedi-olvasó olvasható/írható objektumnak, ahol $V = \mathbb{N}$ és a kezdőérték $v_0 = 0$, néhány olyan végrehajtási sorozatát mutatja, amelyek teljesítik az atomi tulajdonságot olvasható/írható regiszter változótípusra. A sorbarendező pontokat a $*$ jelöli. Tegyük fel, hogy a 1. kapu az író, a 2. pedig az olvasó.



13.3.. ábra. Egyedi-ír/egyedi-olvas objektum atomi tulajdonságú végrehajtási sorozatai.

Az (a) esetben az `olvas` művelet, amely 0-t eredményez, és az `ír(8)` művelet átlapolják egymást, és az `olvas` sorbarendező pontja előbb van, mint `ír(8)`-é. Tehát ha a műveletek intervallumait a sorbarendező pontjaikra zsugorítjuk, akkor az $\texttt{olvas}_2, 0_2, \texttt{ír}(8)_1, \texttt{nyugtáz}_1$ sorozatot kapjuk. Ez pedig a változótípus története (lásd a 9.4.4. példát.)

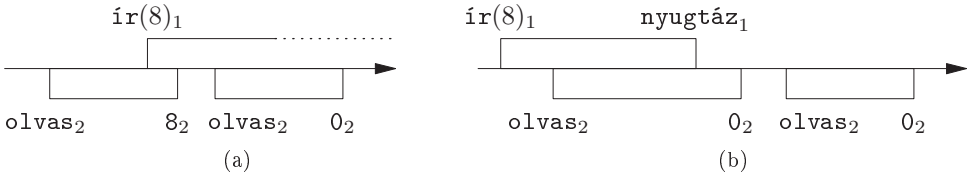
A (b) esetben az egymást követő műveletekhez tartozó sorbarendező pontok sorrendje fordított. Az így kapott kérések és válaszok sorozata $\texttt{ír}(8)_1, \texttt{nyugtáz}_1, \texttt{olvas}_2, 8_2$, ami ismét egy története a változótípusnak.

A (c) és (d) esetek mindegyike tartalmazza a `ír(8)` befejezetlen műveletet. Mindkét esetben megadunk sorbarendező pontot, mert a művelet eredményét láthatja az `olvas` művelet. A műveletek intervallumainak zsugorítása a (*c*) esetben az $\texttt{ír}(8)_1, \texttt{nyugtáz}_1, \texttt{olvas}_2, 8_2$ sorozatot, míg a (*d*) esetben az $\texttt{olvas}_2, 0_2, \texttt{ír}(8)_1, \texttt{nyugtáz}_1, \texttt{olvas}_2, 8_2$ sorozatot eredményezi. Mindkettő története a változótípusnak. (Ismét csak lásd a 9.4.4. példát.)

Az (e) eset végtelen sok `olvas` műveletet tartalmaz, amelyek 0-t adnak, így a $\texttt{ír}(8)$ befejezetlen művelethez nem rendelhetünk sorbarendező pontot.

13.1.3. példa. Nem sorrendesíthető végrehajtási sorozatok

A 13.4. ábra az egyedi-író/egyedi-olvasó olvasható/írható objektumnak néhány olyan végrehajtási sorozatát mutatja, amelyek nem teljesítik az atomi tulajdonságot. Az (a) esetben nem lehet úgy beszúrni sorbarendező pontot, amely értelmessé tenné a 8-at adó `olvas` és az ezt követő, 0-t adó `olvas` műveleteket. A (b) esetben nem magyarázható az `ír`(8) művelet befejeződése utáni 0-at adó `olvas`.



13.4.. ábra. Egyedi-ír/egyedi-olvas objektum atomi tulajdonságot nem teljesítő végrehajtási sorozatai.

Most már (tényleg) készen állunk, hogy definiáljuk az atomiság feltételt az $A \times U$ rendszerre.

Atomiság. Legyen α az $A \times U$ (véges avagy végtelen) végrehajtása, amely jólformált minden i-re. Ekkor α teljesíti az atomi tulajdonságot (amit az imént definiáltunk a 13.1.2. példában).

Az atomiság feltételt kifejezhetjük történet tulajdonságként is (a történet tulajdonság definíciója a 8.5.2. szakaszban található). Nevezetesen, definiáljuk azt a P történet tulajdonságot, amelynek *lenyomat*(P) lenyomata megegyezik $A \times U$ külső felületével és a történetek *történetek*(P) halmaza pontosan azokat a sorozatokat tartalmazza, amelyek kielégítik a következő két feltételt:

1. minden i-re jólformált;
2. atomi tulajdonság a $\mathcal{T}$ változótípusra.

(Kényelmi okok miatt bevesszük a P lenyomatba a `megállít` műveletet is annak ellenére, hogy nincs megemlítve a jólformáltság és atomiság feltételekben.) Az az érdekes, hogy a P biztonságossági tulajdonság, amit a 8.5.3. részben definiáltunk. Vagyis *történetek*(P) nem üres, kezdőszeletre-zárt és határértékre-zárt. Ez nem nyilvánvaló, mivel az atomi tulajdonság eléggé bonyolult definíció, amely tartalmaz sorbarendező pontok elhelyezését és műveletek és azok válaszainak kiválasztását.

13.1. tétel. . *P (a fent definiált történet tulajdonság, amely a jólformáltság és az atomiság kombinációja) biztonságossági tulajdonság.*

A 13.1. tétel bizonyítása felhasználja a végtelen fákra alapvető kombinatorikus Kőnig-féle lemmát.

13.2. lemma. (Kőnig-lemma). *Ha a G végtelen fa minden csúcsának csak véges sok fia van, akkor G-ben létezik a gyökérből induló végtelen út.*

13.1. tétel bizonyításvázlata. Az világos, hogy P nem üres, mert $\lambda \in$ *történetek*(P). A kezdőszeletre-zártság bizonyításához tegyük fel, hogy $\beta \in$ *történetek*(P) és β' véges kezdőszelete β-nak. Mivel $\beta \in$ *történetek*(P), ezért kiválasztható a befejezetlen műveleteknek egy olyan Φ halmaza és Φ műveleteihez tartozó válaszok, továbbá sorbarendező pontok, amelyek együttesen biztosítják β helyességét. Megmutatjuk, hogyan lehet ilyen választásokat elvégezni β'-re.

Jelölje γ azt a sorozatot, amelyet β-ból kapunk, ha elvégezzük a sorbarendező pontok beszúrását. Legyen γ' az a kezdőszelete γ-nak, amely β' utolsó elemével végződik. γ' tartalmaz sorbarendező pontokat β' minden befejezett és néhány befejezetlen műveletére. Válasszuk Φ'-nek a β' azon befejezetlen műveleteit, amelyeknek vannak sorbarendező pontjai γ'-ben. Minden $\pi \in \Phi'$ művelethez válasszuk meg a választ a következőképpen: Ha π befejezetlen β-ban, azaz $\pi \in \Phi$, akkor ugyanazt a választ vegyük, amit β-ban vettünk π-hez. Egyébként vegyük azt a választ, amely ténylegesen szerepel β-ban. Ekkor nem nehéz belátni, hogy az így választott Φ' és válaszaik a sorbarendező pontokkal biztosítják β' helyességét. Ez igazolja a kezdőszeletre-zártság tulajdonságot.

Végül megmutatjuk a határértékre-zártság teljesülését. Legyen β olyan végtelen sorozat, amelynek minden véges kezdőszelete eleme *történetek*(P)-nek. A Kőnig-lemmát fogjuk használni.

Az a G fa, amelyet azért konstruálunk, hogy alkalmazni tudjuk a Kőnig lemmát, a sorbarendező pontok lehetséges β-beli elhelyezését írja le. G minden pontja β egy kezdőszeletével van címkézve, ahol bizonyos β-beli műveletekhez sorbarendező pontokat illesztettünk be. Csak olyan címkéket alkalmazunk, amelyek „helyesek" abban az értelemben, hogy kielégítik a következő három feltételt.

1. Minden befejezett művelethez pontosan egy sorbarendező pont tartozik, és a kérés és a válasz között helyezkedik el.
2. Minden befejezetlen művelethez legfeljebb egy sorbarendező pont tartozik, és a kérés után helyezkedik el.
3. Minden π művelethez pontosan az a válasz tartozik, amelyet az adott $\mathcal{T}$ változótípus függvénye ad meg. (A v_0 kezdőértékkel indulva, az előfordulások sorrendjében alkalmazva a függvényt úgy, hogy az első argumentum a kérés. A π-re kiszámított választ úgy kapjuk, hogy a függvényt a $*_\pi$ sorbarendező pontra alkalmazzuk.)

Továbbá, G címkézésére az alábbiak teljesülnek.

1. A gyökér címkéje λ.
2. Minden nem gyökér pont címkéje a szülő pont címkéjének kiterjesztése.
3. Minden nem gyökér pont címkéje a β egy elemével végződik.
4. Minden nem gyökér pont címkéje eggyel több elemet tartalmaz β-ból, mint a szülő pont címkéje (és esetleg több sorbarendező pontot).

Tehát a G fa minden pontjában az elágazások döntést jelentenek sorbarendező pont két β-beli elem közé történő beillesztésére vonatkozóan. A kezdőszeletzártságot felhasználó fenti konstrukcióból látható, hogy a kapott G fa olyan, hogy

β minden véges β' kezdőszelete a β' utolsó előtti eleméig elvégezhető összes lehetséges helyes sorbarendező pont elhelyezéssel előfordul valamely pont címkéjeként.

Alkalmazzuk most Kőnig-lemmát G-re. Először is világos, hogy G minden pontjának csak véges sok fia van. Ez azért igaz, mert csak az eddig alkalmazott műveleteket kell tekinteni és csak véges sok lehetőség van sorbarendező pont beillesztésére.

Másodszor, elvárjuk, hogy G-ben legyen akármilyen hosszú, gyökérből induló út. Ez azért igaz, mert β minden véges β' kezdőszelete eleme *történetek*(P)-nek, ami azt jelenti, hogy léteznek alkalmas sorbarendező pontok β'-höz. Ez igazolja, hogy G-ben van gyökérből induló $|\beta'|$ hosszú út.

Mivel G-ben van akármilyen hosszú gyökérből induló út, ezért végtelen. A Kőnig lemmából (13.2. lemma) következik, hogy G tartalmaz a gyökérből induló végtelen hosszú utat. Ezen az úton található pontok címkéi egy lehetséges sorbarendező pont választást adnak az egész β sorozatra. □

Miután definiáltuk atomi objektumokra a biztonságossági tulajdonságot – jólformáltság és atomiság – tekintsük az élénkségi tulajdonságot. Olyan élénkségi tulajdonságot vizsgálunk, amely megállási feltétel, hasonló a 12.1. alfejezetben a megegyezés problémára adott feltételre. A legegyszerűbb a feltétel a hibamentes végrehajtásokra, tehát az olyanokra, amelyek nem tartalmaznak `megállít` műveletet.

Hibamentes megállás. $A \times U$ minden pártatlan hibamentes végrehajtásában minden kérésre van válasz.

Ezzel az élénkségi tulajdonsággal definiálhatjuk az „atomi objektum” fogalmát. Nevezetesen, azt mondjuk, hogy A a $\mathcal{T}$ változótípus *atomi objektuma*, ha felhasználók minden kollekciójára biztosítja a jólformáltság, a $\mathcal{T}$-re vonatkozó atomiság és a hibamentes megállás feltételeket.

Megjegyzendő, hogy ha csak a hibamentes esetet tekintenénk, akkor egyszerűsíthetnénk az atomiságra vonatkozó állítást, mert sohasem kellene befejezetlen műveleteket tekinteni. Azért adtuk meg az atomiság bonyolultabb feltételét, mert hibás esetet is fogunk vizsgálni.

Mint a kölcsönös kizárás esetében a 10.2. alfejezetben, most is lehetőség van az atomi objektum ekvivalens átfogalmazására kizárólag egy P történet tulajdonságot használva. Ekkor a *lenyomat*(P) tartalmazza az atomi objektum külső felületének minden műveletét, a `megállít` műveleteteket csakúgy, mint a kéréseket és a válaszokat. A *történetek*(P) kifejezi a jólformáltságot, az atomiságot és a hibamentes megállás feltételét. Ekkor egy A automata egy helyes felülettel $\mathcal{T}$ változótípusú atomi objektum, pontosan akkor, ha felhasználók minden kollekciójára *pártatlan_ történetek*$(A \times U) \subseteq$ *történetek*(P).

Még erősebb megállási feltételt is vizsgálunk, amely hibatűrést tartalmaz.

Várakozásmentes megállás. $A \times U$ minden pártatlan végrehajtásában minden hibátlan kapura érkező kérésre van válasz.

Vagyis, minden olyan kapura érkező kérésre van válasz, amely nem hibázik, függetlenül attól, hogy bármely más kapura érkezett-e `megállít` művelet. Általá-

nosítjuk ezt a tulajdonságot, hogy megadjuk a megállást arra az esetre, amikor meghatározott számú hiba előfordulhat.

f-hibás befejeződés, $0 \leq f \leq n$. $A \times U$ minden pártatlan végrehajtásában, ha legfeljebb f kapura érkezik `megállít` művelet, akkor minden hibátlan kapura érkező kérésre van válasz.

A hibamentes, illetve a várakozásmentes megállás speciális esete az f-hibás megállásnak, $f = 0$, illetve $f = n$ értékekre. Egy további általánosítás lehetővé teszi, hogy kapuk adott halmazának hibázásáról beszéljünk.

I-hibás befejeződés, $I \subseteq \{1, \ldots, n\}$. $A \times U$ minden pártatlan végrehajtásában, ha legfeljebb I-beli kapura érkezik `megállít` művelet, akkor minden hibátlan kapura érkező kérésre van válasz.

Tehát az f-hibás befejeződés feltétele megegyezik azzal, hogy kapuk minden legfeljebb f elemszámú I halmazára I-hibás. Azt mondjuk, hogy A *biztosítja a várakozásmentes megállást*, *biztosítja az I-hibás befejeződést*, és így tovább, ha A teljesíti a megfelelő feltételt felhasználók minden kollekciójára.

Az alfejezetet egy olyan közös memóriájú rendszer egyszerű példájával zárjuk, amely atomi objektum.

13.1.4. példa. Egy olvasható/növelhető atomi objektum

Definiáljuk azt az *olvasható/növelhető* változótípust, amelynek tartománya $\mathbb{N}$, kezdőértéke 0, műveletei az `olvas` és `növel`.

Legyen A egy olyan közös memóriájú rendszer, amely n folyamatból áll és minden i kapun támogatja mind az `olvas`, mind az `növel` műveletet. A-nak n számú közös *olvasható/írható* regisztere van, $x(i), 1 \leq i \leq n$, mindegyik tartománya $\mathbb{N}$ és kezdőértékük 0. Az $x(i)$ közös változót írhatja a P_i folyamat és olvashatja bármelyik.

Amikor $\texttt{növel}_i$ bemenet érkezik az i-edik kapura, a P_i folyamat egyszerűen megnöveli az $x(i)$ közös változója értékét. Ezt csak az `ír` művelettel hajthatja végre, megjegyezve az $x(i)$ értékét a helyi állapotában. Amikor $\texttt{olvas}_i$ érkezik az i-edik kapura, akkor a P_i folyamat egyesével kiolvassa az összes $x(j)$ közös változót tetszőleges sorrendben, majd összeadja azokat és visszaadja az összeget. Nem nehéz belátni, hogy A olvasható/növelhető atomi objektum és várakozásmentes megállást biztosít. Például, az atomiság feltételt nézve, tekintsünk egy tetszőleges végrehajtását $A \times U$-nak. Legyen Φ azoknak a befejezetlen növel műveleteknek a halmaza, amelyekre írás előfordul a közös változóra. Minden olyan π `növel` műveletre, amely vagy befejezett, vagy eleme Φ-nek, helyezzük a $*_\pi$ sorbarendező pontot a `ír` műveletre.

Most vegyük észre, hogy minden befejezett (magasabb szintű) π `olvas` művelet olyan v értéket ad, amely nem *kisebb*, mint az összes

$x(i)$ összege amikor `olvas` hívodik, és nem *nagyobb*, mint az összes $x(i)$ összege amikor `olvas` befejeződik. Mivel minden `növel` pontosan 1-el növeli ezt az összeget, lennie kell egy olyan pontnak π intervallumában, ahol az $x(i)$-k összege pontosan v. Helyezzük a sorbarendező $*_\pi$ pontot erre a pontra. Ez a választás biztosítja a kívánt zsugorítást, ami az atomisághoz kell.

13.1.2.. Kanonikus várakozásmentes atomi objektum automata

Ebben az alfejezetben példát adunk adott $\mathcal{T}$ változótípusú és adott külső felületű C atomi objektum automatára. A C automata várakozásmentes megállást biztosít. C erősen nemdeterminisztikus és gyakran az adott típusú és külső felületű „kanonikus várakozásmentes atomi objektum automatának" tekintik. Ez felhasználható annak megmutatására, hogy más adott automaták várakozásmentes atomi objektumom.

C automata (vázlatosan)

C a $\mathcal{T}$ típusú közös változó másolatát lokálisan tárolja, aminek kezdőértéke v_0. Van két puffere, a *kérés_puffer* a függőben lévő kérésekre és a *válasz_puffer* a függőben lévő válaszokra, kezdetben mindkettő üres. Végül, nyomon követi a *megállított* halmazban azokat a kapukat, amelyeken `megállít` előfordult, ennek kezdőértéke üres.

Amikor egy kérés érkezik, C egyszerűen tárolja azt a *kérés_puffer*-ben. C bármely pillanatban kivehet a *kérés_puffer*-ből függőben lévő kérést és végrehajthatja a műveletet a közös változó külső másolatán. Amikor ezt teszi, a válasz eredményét a *válasz_puffer*-be teszi. C szintén bármikor kivehet egy függőben lévő választ a *válasz_puffer*-ből és továbbíthatja a választ a felhasználónak.

A $\texttt{megállít}_i$ esemény csak hozzáveszi i-t a *megállított* halmazhoz és ezzel végrehajthatóvá teszi a $\texttt{semmis}_i$ műveletet, aminek nincs hatása. Nem teszi végrehajthatóvá azonban a többi i-hez tartozó helyileg vezérelt műveletet. Minden i-re egy taszkba csoportosítjuk az összes helyileg vezérelt műveletet, beleértve $\texttt{semmis}_i$-t is. Ez azt jelenti, hogy $\texttt{megállít}_i$ után azok a műveletek, amelyek i-t érintik, megállhatnak, de nem kötelező ezt tenniük.

Az automata kódja a következő.

13.1. automata. C

Lenyomat:

Bemeneti:
- a_i, a külső felület művelete
- $\texttt{megállít}_i, 1 \leq i \leq n$

Kimeneti:
- $\texttt{b}_i$, a külső felület művelete

Belső:
- $\texttt{végrehajt}(a)_i,\ a_i$ a külső felület művelete $1 \leq i \leq n$
- $\texttt{semmis}_i, 1 \leq i \leq n$

Állapotok:
ért $\in V$, kezdetben v_0
kérés_puffer az (i, a) párok halmaza, ahol a a külső felület művelete, kezdetben üres
válasz_puffer az (i, b) párok halmaza, ahol b a külső felület művelete, kezdetben üres
megállított $\subseteq \{1, \cdots, n\}$, kezdetben üres

Átmenetek:

a_i
Hatás:
kérés_puffer := *kérés_puffer* $\cup \{(i, a)\}$

b_i
Előfeltétel:
$(i, b) \in$ *válasz_puffer*
Hatás:
válasz_puffer :=
válasz_puffer $- \{(i, b)\}$

végrehajt$(a)_i$
Előfeltétel:
$(i, a) \in$ *kérés_puffer*
Hatás:
kérés_puffer :=
kérés_puffer $- \{(i, a)\}$
$(b,$ *ért*$) := f(a,$ *ért*$)$
válasz_puffer :=
válasz_puffer $\cup \{(i, b)\}$

megállít$_i$
Hatás:
megállított := *megállított* $\cup \{i\}$

semmis$_i$
Előfeltétel:
$i \in$ *megállított*
Hatás:
nincs

Taszkok:
Minden i-re:
$\{$végrehajt$(a)_i : a_i$ bemenet$\} \cup \{b_i : b_i$ kimenet$\} \cup \{semmis_i\}$

- -

13.3. tétel. . *C adott típusú és külső felületű atomi objektum, amely várakozásmentes megállást biztosít (felhasználók minden kollekciójára).*

Bizonyításvázlat. A jólformáltság nyilvánvaló. A várakozásmentesség végett tekintsünk egy α pártatlan végrehajtását $C \times U$-nak és tegyük fel, hogy az i-edik kapu nem hibázik α-ban. Ekkor a semmis$_i$ sohasem lesz végrehajtható α-ban. α pártatlansága miatt az i-edik kapu minden kérése kiváltja a végrehajt$_i$ eseményt és az ezt követő választ.

Hátra van az atomiság megmutatása. Legyen α egy tetszőleges végrehajtása $C \times U$-nak. Legyen Φ azoknak a befejezetlen műveleteknek a halmaza, amelyekre végrehajt előfordul α-ban. Adjuk meg a $*_\pi$ sorbarendező pontot α minden olyan π műveletére, amely vagy befejezett, vagy eleme Φ-nek: helyezzük $*_\pi$-t a végrehajt pontra. Minden $\pi \in \Phi$ műveletre a válasz legyen a végrehajt művelet eredménye. Ezek a választások lehetővé teszik az atomisághoz szükséges zsugorítást. □

A C automata felhasználásával bizonyítani tudjuk, hogy más automaták szintén várakozásmentes atomi objektumok.

13.4. tétel. . *Tegyük fel, hogy A olyan b/k automata, amelynek külső felülete megegyezik C külső felületével, továbbá pártatlan_történetek$(A \times U) \subseteq$ pártatlan_történetek$(C \times U)$ felhasználói automaták minden U összekapcsolására. Ekkor A várakozásmentes megállást biztosító atomi objektum.*

Bizonyításvázlat. A bizonyítás a 13.3. tételből következik. A jólformáltság és az atomiság igazolásához felhasználjuk azt a tényt, hogy e két tulajdonság együttesen biztonságossági feltételt jelent (13.1. tétel), és még azt a tényt, hogy minden véges történet kiterjeszthető pártatlan történetté (8.7. tétel). A várakozásmentesség közvetlenül következik a definíciókból. □

A 13.4. tétel fordítottja is igaz, ami azt mondja, hogy minden pártatlan történetet, amely megengedett egy várakozásmentes atomi objektumra, ténylegesen előállítja C.

13.5. tétel. . *Tegyük fel, hogy A olyan b/k automata, amelynek külső felülete megegyezik C külső felületével, továbbá A várakozásmentes megállást biztosító atomi objektum. Ekkor pártatlan_történetek$(A \times U) \subseteq$ pártatlan_történetek$(C \times U)$ felhasználói automaták minden U összekapcsolására.*

Bizonyítás. A bizonyítás gyakorlatként elvégezhető (lásd 13-7. gyakorlat). □

13.1.3.. Atomi objektumok összekapcsolása

Ebben az alfejezetben bebizonyítjuk, hogy atomi objektumok minden összekapcsolása szintén atomi objektum (a 8.2.1. szakaszban definiált közönséges b/k automata összekapcsolást használva). Emlékeztetünk, hogy a változótípusok kompatibilitásának és összekapcsolásának definíciója 9.4. alfejezet végén szerepel.

13.6. tétel. . *Legyen $\{A_j\}_{j\in J}$ a $\{\mathcal{T}_j\}_{j\in J}$ kompatibilis változó típusú atomi objektumok megszámlálható halmaza, ahol mindegyik objektum kapuinak halmaza $\{1, \cdots, n\}$. Ekkor az $A = \prod_{j\in J} A_j$ összekapcsolás atomi objektum lesz, amelynek változótípusa $\mathcal{T} = \prod_{j\in J} \mathcal{T}_j$ és kapuinak halmaza $\{1, \cdots, n\}$.*

Továbbá, ha minden A_j I-hibás befejeződésű, akkor A-is az.

Az A atomi objektum i-edik kapuja mindazon kérések és válaszok kezelését végzi, amelyek valamely A_j automata az i-edik kapujára érkeznek. A összekapcsolás definíciója értelmében A egy állapota minden A_j-ből tartalmaz egy állapotot Az A_j-hez tartozó kérés és válasz csak az A_j-beli állapot komponenst érinti. A `megállít`$_i$ műveletek azonban minden állapotkomponensre hatnak. A 13.6. tétel bizonyítását meghagyjuk gyakorlatnak (lásd 13-8. gyakorlat).

13.1.4.. Atomi objektumok avagy közös változók

Az atomi objektum definíciója azt mondja, hogy történetei „hasonlóak” az azonos típusú szekvenciális elérésű közös változók történeteihez. Mire jó ez?

Az atomi objektumok legfontosabb tulajdonsága, rendszerek építését tekintve, hogy közös változós rendszerekben közös változókat helyettesíthetnek.

Ez lehetővé teszi rendszerek moduláris építését: lehet először közös memóriájú rendszert tervezni, aztán helyettesíteni a közös változót azonos típusú atomi objektummal. Bizonyos körülmények között az így megépített rendszer „ugyanúgy viselkedik", mint az eredeti közös memóriájú rendszer, legalábbis a felhasználók szemével nézve.

Ebben az részben megadjuk ezt a helyettesítési technikát. Először megadjuk azokat a technikai feltételeket, amelyeket az eredeti közös memóriájú rendszernek ki kell elégítenie ahhoz, hogy a helyettesítés helyesen működjön. Ezután adjuk meg a helyettesítő konstrukciót. Végül megadjuk, hogy milyen értelemben viselkedik ugyanúgy a helyettesítéssel kapott rendszer az eredetivel, és bebizonyítjuk, hogy a feltételek mellett a kapott rendszer valóban ugyanúgy viselkedik. Habár az alapötlet meglehetősen egyszerű, azonban néhány részletet óvatosan kell kezelni ahhoz, hogy a helyettesítési technika helyes eredményt adjon.

Legyen A egy tetszőleges közös változós rendszert használó algoritmus a 9. fejezet értelmében. Feltételezzük, hogy A az $U_i, 1 \leq i \leq n$ felhasználói automatákkal van kölcsönhatásban. Megengedjük, hogy A minden P_i folyamatának akárhány taszkja legyen. Szintén bevezetünk $\texttt{megállít}_i$ műveleteket, amelyeket a 9.6. alfejezetben tárgyaltunk, és feltételezzük, hogy minden $\texttt{megállít}_i$ esemény véglegesen végrehajthatatlanná teszi a P_i folyamat minden taszkját.

Most lássuk a már említett technikai feltételeket. Tekintsük A-t felhasználói automaták egy tetszőleges U_i kollekciójával. Feltételezzük, hogy minden i kapuhoz van egy $\texttt{követ}_i$ függvény, amely minden α véges végrehajtásra vagy *rendszer* vagy *felhasználó* értéket ad. Azt feltételezzük, hogy ez a fügvény adja meg, hogy az α után kinek kell a következő lépést elvégeznie. Speciálisan, ha $\texttt{követ}_i(\alpha) = \textit{rendszer}$, akkor U_i kimenet műveletet nem tud végrehajtani az α utáni állapotában, és ha $\texttt{követ}_i(\alpha) = \textit{felhasználó}$, akkor az A P_i folyamata nem tud végrehajtani kimenet vagy helyi, azaz helyileg vezérelt műveletet α utáni állapotában.

Például, a 10. fejezet minden kölcsönös kizárásos és a 11. fejezet minden erőforrás-hozzárendelő algoritmusa teljesíti ezeket a feltételeket (ha bevesszük a `megállít` műveletet). Ezekben az esetekben $\texttt{követ}(\alpha) = \textit{rendszer}$, ha U_i α után próba vagy kijárat tartományban van, és $\texttt{követ}(\alpha) = \textit{felhasználó}$, ha U_i kritikus vagy maradék tartományban van. A megkövetelt feltételek ténylegesen következnek a folyamat aktivitására kirótt megszorításokból, amit a 10.2. alfejezet végén, illetve a 11.1.2 rész végén adtunk meg.

A 12. fejezetben tárgyalt megegyezés algoritmusok esetében legyen $\texttt{követ}(\alpha) = \textit{rendszer}$, ha α tartalmaz $\texttt{kezd}_i$ eseményt, és $\texttt{követ}(\alpha) = \textit{felhasználó}$ egyébként. A kívánt feltételek teljesüléséhez további megszorítást kell tenni, nevezetesen, a P_i folyamat nem csinálhat semmit amíg $\texttt{kezd}_i$ be nem következik. Ezt a feltétel csak a 12. fejezet OMÍMEGEGYEZÉS algoritmusa teljesíti.

Most megadjuk a helyettesítést. Tegyük fel, hogy A minden x közös változójához adott a B_x ugyanolyan típusú atomi objektum automata a megfelelő külső felülettel. Tehát B_x kapui $1, \cdots, n$, A minden folyamatához egy kapu. Minden kapun megengedett mindazon kérés és válasz, amelyet az A algoritmus P_i folyamata használ az x közös változó elérésére. Van a $\texttt{megállít}_i$ bemenet minden kapura, mint az szokásos.

Ekkor az A automata átalakított változatát, a TRANSZ(A) automatát a következőképpen definiáljuk:

TRANSZ(A) automata (vázlatosan)

TRANSZ(A) b/k automaták összekapcsolása, ahol minden P_i folyamathoz és az A algoritmus minden x közös változójához tartozik egy komponens automata. Minden x változóhoz a B_x atomi objektum automata tartozik. Minden i folyamathoz az a P_i automata tartozik, amelyet a következőképpen definiálunk:

P_i bemenetei az A i-edik kapujának bemenetei, plusz az összes B_x válaszai az i-edik kapun, plusz a $\texttt{megállít}_i$ művelet. P_i kimenetei az A i-edik kapujának kimenetei, plusz az összes B_x kérései az i-edik kapun.

P_i lépései az A algoritmus i-edik folyamatának lépéseit szimulálják a következő eltéréssel. Amikor A i-edik folyamata végrehajt egy hozzáférést az x közös változóhoz, akkor e helyett P_i végrehajtja B_x megfelelő kérését. Ezt követően felfüggeszti aktivitását és várakozik amíg B_x válaszol a kérésre. Amikor a válasz megérkezik, P_i folytatja az A i-edik folyamatának szimulálását a szokásos módon. A minden i folyamatának minden taszkjához tartozik egy taszkja P_i-nek.

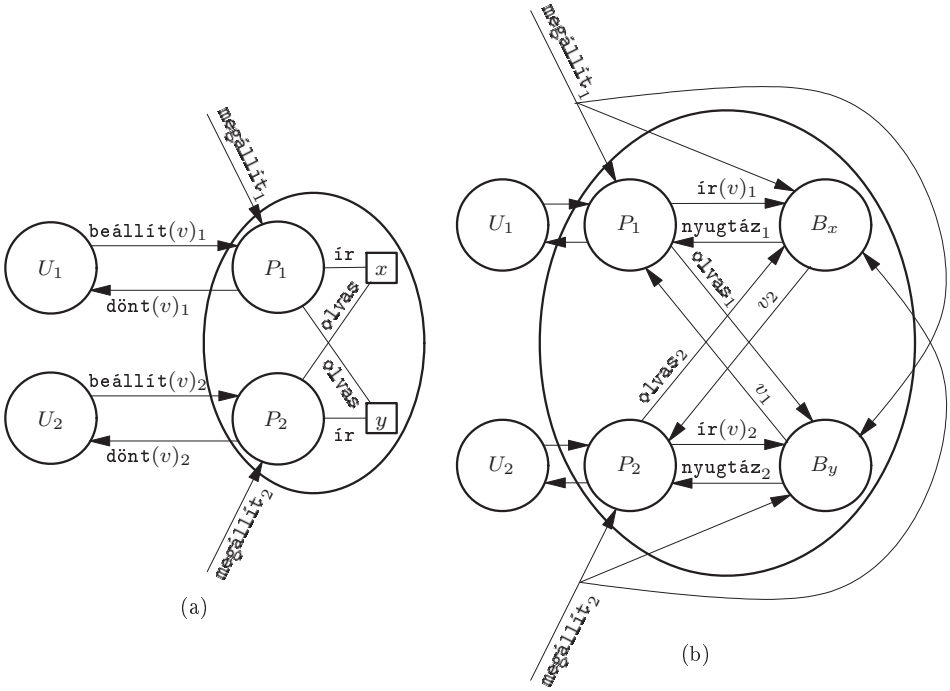
Ha $\texttt{megállít}_i$ esemény bekövetkezik, akkor ezt követően P_i minden taszkja végrehajthatatlanná válik.

13.1.5. példa. *A és* TRANSZ(A)

Tekintsük egy két-folyamatos közös memóriájú rendszert, amely valamely megegyezési feladat megoldását végzi, felhasználva az x és y ír/olvas közös változókat. Feltesszük, hogy a P_1 folyamat írja az x-et és olvassa az y-t, míg a P_2 folyamat írja az y-t és olvassa az x változót. A felület A és minden U_i között tartalmazza a $\texttt{kezd}(v)_i$ műveletet, amely kimenete U_i-nek és bemenete A-nak, a $\texttt{dönt}(v)_i$ műveletet, amely kimenete A-nak és bemenete U_i-nek. Ezen kívül $\texttt{megállít}_i, i \in \{1,2\}$ bemenetei A-nak. Ennek a rendszernek az architektúráját a 13.5. ábra (a) része szemlélteti.

Az átalakított rendszernek az architektúráját a 13.5. ábra (b) része szemlélteti. Figyeljük meg a B_x és B_y automaták külső felületét. Például B_x bemenetei $\texttt{ír}(v)_1$, $\texttt{olvas}_2$ és kimenetei $\texttt{nyugtáz}_1$, v_2. [2] B_x-nek szintén van $\texttt{megállít}_1$ és $\texttt{megállít}_2$ bemenete, amelyek a P_1 $\texttt{megállít}_1$ illetve P_2 $\texttt{megállít}_2$ bemenetével azonosak. Ez azt jelenti például, hogy $\texttt{megállít}_1$ egyszerre érvényteleníti P_1 minden taszkját és akármilyen hatása lehet B_x megvalósításában.

[2] A félreértés elkerülése végett ezeket a kéréseket és válaszokat lehetne indexelni a megfelelő x illetve y objektumokkal. Ettől a részletezéstől eltekintünk a példában, mert nem merül fel félreértés.



13.5.. ábra. Közös memóriájú rendszer átalakítása atomi objektummá.

Most jön az a tétel, amely megmondja, hogy mit őriz meg a TRANSZ átalakítás. A 13.7. tétel először megadja azokat a feltételeket, amelyek teljesülnek TRANSZ(A) minden α végrehajtására. Az α végrehajtásnak nem kell pártatlannak lennie ahhoz, hogy a feltételek teljesüljenek. Ezek a feltételek azt mondják, hogy a felhasználók számára α úgy néz ki, mint A egy α' végrehajtása. Továbbá, ugyanazok a `megállít` események fordulnak elő α-ban és α'-ben, habár megengedjük azt a lehetőséget, hogy a `megállít` események különböző helyen forduljanak elő a két végrehajtásban.

Ezután a 13.7. tétel továbbmegy, és azonosítja azokat a feltételeket, amelyek esetében az A rendszer α' szimulált végrehajtása pártatlan végrehajtás lesz. Amint az várható, az egyik feltétel éppen az lesz, hogy α pártatlan végrehajtása legyen a TRANSZ(A) rendszernek. Ez azonban nem elég – biztosítanunk kell azt is, hogy a B_x objektum automata ne állítsa meg a végrehajtást. Ezért bevezetünk további két feltételt, amelyek együttesen biztosítják, hogy ez ne fordulhasson elő. Nevezetesen, minden hiba ami α-ban előfordulhat, kapuk egy I halmazára esik és minden B_x objektum automata elviseli az I-beli kapukon keletkező hibákat (formálisan, I-hibás befejeződést biztosítanak).

13.7. tétel. . *Tegyük fel, hogy α a* TRANSZ(A) $\times U$ *rendszer egy tetszőleges végrehajtása. Ekkor van olyan α' végrehajtása $A \times U$-nak, amelyre teljesül a következő két feltétel:*

1. α és α' megkülönböztethetetlen U számára; [3]

2. minden i-re, `megállít` *pontosan akkor fordul elő α-ban, ha előfordul α'-ben. Továbbá, ha α pártatlan végrehajtás és minden i, amelyre* `megállít`$_i$ *előfordul α-ban az I halmazba esik, és B_x I-hibás befejeződést biztosít (felhasználók minden kollekciójára), akkor α' is pártatlan végrehajtás.*

Bizonyításvázlat. α-t az alábbiak szerint módosítjuk, hogy α'-t kapjuk. Mivel minden B_x atomi objektum, ezért $*_\pi$ sorbarendező pontokat helyezhetünk el α-ban B_x minden π befejezett műveletének kérése és válasza közé, és minden kiválasztott Φ-be eső befejezetlen műveletének kérése után. Φ minden műveletére kapunk választ. Ezek a sorbarendező pontok és válaszok biztosítják az atomiság feltételhez szükséges zsugorítást.

Ezután mozgassuk át B_x minden π befejezett műveletének kérését és válaszát a $*_\pi$ sorbarendező pontra. Hasonlóan, minden $\pi \in \Phi$ befejezetlen műveletre – vagyis minden befejezetlen műveletre, amelyekhez sorbarendező pontot választottunk – helyezzük a kérést és az újonnan választott választ a $*_\pi$ pontra. Továbbá, minden olyan befejezetlen művelet esetében, amely nem eleme Φ-nek – vagyis minden olyan befejezetlen műveletet, amelyhez nem választottunk sorbarendező pontot – egyszerűen töröljük a kérés eseményt. Van még egy technikai dolog: ha α-ban bármely `megállít`$_i$ esemény a P_i folyamat kérése és a sorbarendező pont után fordulna elő, akkor tegyük `megállít`$_i$-t is a sorbarendező ponthoz közvetlenül, a kérés és válasz utánra. Ilyen módon minden x közös változóra mozgatunk, hozzáadunk és eltávolítunk eseményeket.

Az a kívánalom, hogy az összes olyan eseményt, amelyet a fenti konstrukcióban mozgattunk, át lehessen mozgatni úgy, hogy a P_i-beli sorrendjük ne változzon meg (egy technikai kivételtől eltekintve: B_x P_i-nek adott válasza `megállít`$_i$ elé mozgatható). Ez két ok miatt teljesül. Az első az, hogy a konstrukció szerint P_i nem hajt végre semilyen helyileg vezérelt műveletet, amíg kérés válaszára vár. A második, amíg P_i válaszra vár, a rendszer következik lépni. Ez azt jelenti, hogy U_i nem fog végrehajtani kimenet műveletet, tehát P_i nem kap bemenetet.

Hasonlóan, azt szeretnénk, ha hozzáadhatnánk a válaszokat, illetve eltávolíthatnánk azokat a kéréseket, amelyeket a konstrukcióban hozzáadtunk, illetve eltávolítottunk, anélkül, hogy egyébként módosítanánk P_i viselkedését. Ez azért lehetséges, mert ha P_i befejezetlen műveletet hajt végre α-ban, akkor utána nem hajt végre műveletet. Nem számít, ha P_i megáll éppen mielőtt kiadja a kérést, vagy amíg kérésre várakozik, vagy éppen miután fogadta a választ.

Mivel semmi lényegeset nem változtattunk ezzel a mozgatással, hozzáadással és eltávolítással, egyszerűen ugyanúgy kitölthetjük a P_i folyamat állapotait, mint α-ban. (Technikai kivétel: ha P_i válasza `megállít`$_i$ elé kerül, akkor másképpen kell eljárni P_i állapotának módosításával, mint α esetében tettük.) Az eredmény egy új α_1 végrehajtás, ami szintén végrehajtása a $\textsc{Transz}(A) \times U$ rendszenek. Továbbá, világos, hogy α_1 és α megkülönbözhetetlen U számára és ugyanazon kapukra van `megállít` esemény.

Nos, α_1 végrehajtása $\textsc{Transz}(A) \times U$-nak, de nem pontosan az, ami nekünk kell; nekünk az $A \times U$ végrehajtási sorozata kell. Vegyük észre, hogy α_1-ben

[3] A megkülönböztethetetlenség 8.7. alfejezetben adott definícióját használjuk.

B_x minden kérése és válasza párban egymást követően fordul elő. Így a párokat helyettesítésével a megfelelő közös változóhoz való hozzáférést kapjuk, tehát a $A \times U$ rendszer egy α' végrehajtását kapjuk. Ekkor α és α' megkülönböztethetetlenek U számára és ugyanazon kapukon jelentkezik `megállít` művelet. Ez bizonyítja a tétel első felét.

A második rész bizonyításához tegyük fel, hogy α pártatlan végrehajtása TRANSZ$(A) \times U$-nak, hogy $I \subseteq \{1, \cdots, n\}$ azon i kapuk halmaza, amelyekre `megállít`$_i$ előfordul α-ban és minden B_x I-hibás befejeződést biztosít. Ekkor minden B_x csak olyan `megállít`$_i$ bemenetet fogad, ahol $i \in I$. Így mivel minden B_x I-hibás befejeződést biztosít, ezért a P_i folyamat minden kérésére választ kell adnia B_x-nek ha `megállít`$_i$ nem fordul elő α-ban. Ez a tény és a P_i folyamatra tett pártatlanság feltétel együttesen elegendő ahhoz, hogy α' pártatlan végrehajtási sorozata legyen $A \times U$-nak. □

Tehát a 13.7. tételből következik, hogy közös változós modell minden algoritmusa (néhány egyszerű korlátozással) átalakítható úgy, hogy közös változó helyett atomi objektummal működjön, és a felhasználó nem lássa a különbséget.

A 13.7. tétel következményeként kapjuk azt a speciális esetet, amikor minden B_x atomi objektum várakozásmentes megállást biztosít. Ebben az esetben azt kapjuk, hogy α' pártatlanságához elegendő α pártatlansága.

13.8. következmény. . *Tegyük fel, hogy valamennyi B_x várakozásmentes befejeződést biztosít. Legyen α tetszőleges pártatlan végrehajtása* TRANSZ$(A) \times U$*-nak. Ekkor van olyan α' pártatlan végrehajtása $A \times U$-nak, amelyre a következő két feltétel teljesül:*

1. *α és α' megkülönböztethetetlen U számára;*
2. `megállít`$_i$ *minden i-re pontosan akkor fordul elő α-ban, ha előfordul α'-ben.*

Bizonyítás. Közvetlenül adódik a 13.7. tételből, ha $I = \{1, \cdots, n\}$. □

Abban a speciális esetben, ha A maga is atomi objektum, a 13.7. tételből következik, hogy TRANSZ$(A) \times U$ is atomi objektum lesz. Hibázásra vonatkozó feltételt is alkalmazva az alábbi következményt kapjuk.

13.9. következmény. . *Tegyük fel, hogy A és valamennyi B_x atomi objektum I-hibás befejeződést biztosít. Ekkor* TRANSZ(A) *is atomi objektum és I-hibás befejeződést biztosít.*

Bizonyítás. Először legyen α tetszőleges végrehajtása TRANSZ(A)-nak, és legyen U_i felhasználók egy tetszőleges kollekciója. Ekkor a 13.7. tétel szerint van olyan α' végrehajtása $A \times U$-nak, hogy α és α' megkülönbözhetetlen U számára. Mivel A atomi objektum, ezért α' teljesíti a jólformáltság és atomiság feltételt. Mivel ez a két feltétel tulajdonsága az U külső felületnek is, továbbá α és α' megkülönbözhetetlen U számára, α szintén teljesíti a jólformáltság és atomi tulajdonságot.

Hátra van az I-hibás befejeződési tulajdonság. Legyen α tetszőleges olyan végrehajtása TRANSZ(A)-nak és Ui felhasználók egy tetszőleges kollekciója, hogy minden i-re ha `megállít`$_i$ előfordul α-ban, akkor $i \in I$. Mivel B_x I-hibás befejeződést biztosít, a 13.7. tétel szerint létezik $A \times U$-nak olyan α' pártatlan végrehajtása, hogy α és α' megkülönbözhetetlen U számára, továbbá α és α' ugyanazon

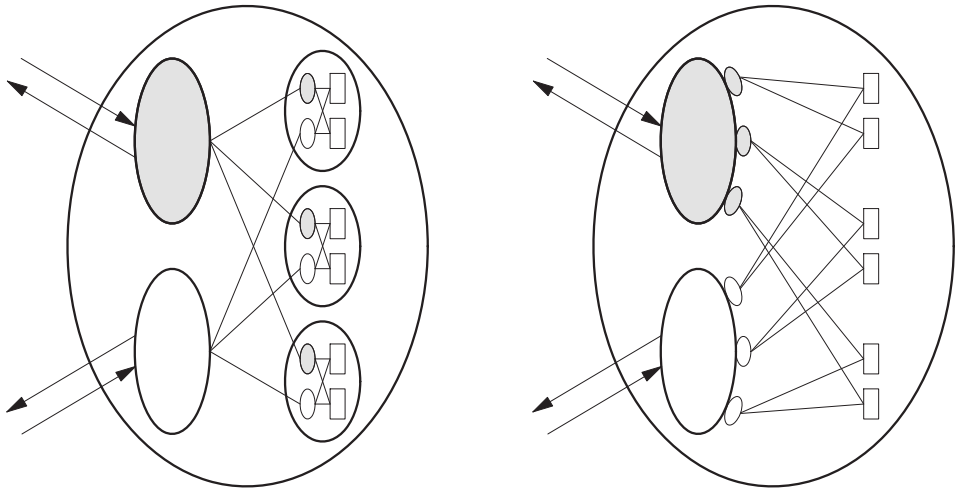
kapukra tartalmaz műveletet. Tehát minden i, amelyre $\texttt{megállít}_i$ előfordul α'-ben eleme I-nek.

Tekintsünk egy tetszőleges kérést α-ban olyan i kapun, amelyre nincs $\texttt{megállít}_i$ α-ban – azaz i hibátlan kapu. Mivel α és α' megkülönböztethetetlen U számára, ugyanez a hívás előfordul α'-ben. Amiatt, hogy A I-hibás befejeződést biztosít, következik, hogy van hozzá tartozó válasz esemény α'-ben. Miután α és α' megkülönböztethetetlen U számára, ezért ez a válasz szintén előfordul α-ban. Ez elegendő az I-hibás befejeződéshez. □

Közös memóriájú rendszerek hierarchikus építése.. Abban a speciális esetben, amikor minden B_x maga is közös memóriájú rendszer, azt várjuk, hogy TRANSZ(A) szintén közös memóriájú rendszernek tekinthető. Nevezetesen,TRANSZ(A) minden i folyamata (közös memóriájú rendszernek tekintve) kombinációja aTRANSZ(A) P_i folyamatának és valamennyi B_x közös memóriájú rendszer i-vel indexelt folyamatainak. Ez a kombináció nem pontosan b/k automatáknak összekapcsolása, mivel B_x folyamatai nem b/k automaták. A kombinációt azonban könnyű leírni: a TRANSZ(A) i-edik folyamatának állapot halmaza Descartes-féle szorzata P_i állapothalmazának és minden B_x i-vel indexelt folyamata állapothalmazának, és hasonlóan kapjuk a kezdő állapotokat is. TRANSZ(A) i-edik folyamatának műveletei pontosan az i-edik komponens folyamat műveletei lesznek, és hasonlóan a taszkok is.

A helyzetet a 13.6. ábra mutatja. Az (a) rész TRANSZ(A)-t szemlélteti, A minden x közös változójához beágyazott B_x osztott memóriájú rendszerekkel. (Az egyszerűség végett nem rajzoltuk fel a `megállít` bemeneti nyilakat.) Minden satírozott folyamat az 1. kapuhoz tartozik. A (b) rész ugyanazt a rendszert mutatja, mint az (a) rész, csak az összevont folyamatok egy csoportba vannak vonva. Tehát az (a) rész összes satírozott folyamata egyetlen egy folyamatba vannak kombinálva a (b) részben. TRANSZ(A) definíciója szerint egy $\texttt{megállít}_i$ esemény hatása az (a) rendszerben az, hogy megállítja az i-edik kapuhoz rendelt összes folyamatot — P_i taszkjait csakúgy, mint a B_x-ek minden i taszkját. Ez ugyanaz, mint ha azt mondanánk, hogy $\texttt{megállít}_i$ megállítja az összetett P_i folyamat minden taszkját a (b) rendszerben, a $\texttt{megállít}_i$ pontosan az, amit $\texttt{megállít}_i$-nek végeznie kell, ha a rendszert közös memóriájú rendszernek tekintjük.

Atomi objektumok hierarchikus építése.. Végül tekintsük azt a nagyon speciális esetet, amikor az A közös memóriájú rendszer atomi objektum és I-hibás megállást biztosít, továbbá minden B_x atomi objektum közös memóriájú rendszer, I-hibás befejeződést biztosítva. Ekkor az előző rész 13.9. következményéből azt kapjuk, hogy TRANSZ(A) atomi objektum, amely I-hibás befejeződést biztosít és szintén közös memóriájú rendszer. Ez az észrevétel azt mondja, hogy a közös memóriájú modellben az atomi objektumok megvalósításának két egymást követő szintje összevonható egybe.



13.6.. ábra. Közös memóriájú rendszerek hierarchikus építése.

13.1.5.. Elegendő feltétel atomiság kimutatására

Mielőtt meghatározott atomi objektum konstrukciókat mutatnánk, egy elegendő feltételt adunk annak igazolására, hogy egy közös memóriájú rendszer teljesíti az atomiság feltételt. Ez a lemma lehetővé teszi, hogy a legtöbb esetben elkerüljük a befejezetlen műveletekkel való közvetlen érvelést objektumok atomiságának bizonyításakor.

A lemma végett feltesszük, hogy A közös memóriájú rendszer a $\mathcal{T}$ változó típusú atomi objektumhoz alkalmas külső felülettel. Szintén feltesszük, hogy $U_i, 1 \leq i \leq n$ A felhasználóinak egy kollekciója és $U = \prod U_i$.

13.10. lemma. . *Tegyük fel, hogy az $A \times U$ összetett rendszer biztosítja a jólformáltság és hibamentes megállás feltételeket. Tegyük fel, hogy $A \times U$ minden olyan α (véges avagy végtelen) végrehajtása, amely nem tartalmaz befejezetlen műveletet, kielégíti az atomi tulajdonságot. Ekkor ugyanez igaz $A \times U$ minden végrehajtására, azokra is, amelyek befejezetlen műveleteket tartalmaznak.*

Bizonyítás. Legyen α az összetett $A \times U$ rendszer egy tetszőleges véges vagy végtelen végrehajtása, amely esetleg befejezetlen műveleteket is tartalmaz. Meg kell mutatnunk, hogy α kielégíti az atomiság feltételt, vagyis $\alpha|\mathit{külső}(U)$ rendelkezik az atomi tulajdonsággal.

Ha α véges, akkor a `megállít` eseményeknek a közös memóriájú rendszerekben való kezelése miatt következik, hogy van olyan véges α_1 hibamentes végrehajtási sorozat, amelyet úgy kapunk, hogy α-ból törlünk minden `megállít` eseményt (és esetleg módosítunk néhány bemenetre állapotváltást olyan kapukon, amelyeken `megállít` előfordult), hogy $\alpha_1|\mathit{külső}(U) = \alpha|\mathit{külső}(U)$. A b/k auto-

maták alapvető tulajdonsága (elsősorban a 8.7. tétel) miatt α_1 kiterjeszthető az $A \times U$ egy α_2 pártatlan, hibamentes végrehajtási sorozatává. Mivel A hibamentes megállást biztosít, α_2-ben minden művelet befejezett. Ezért a feltétel miatt α_2 teljesíti az atomi tulajdonságot, vagyis $\alpha_2|\mathit{külső}(U)$ teljesíti az atomi tulajdonságot. $\alpha_1|\mathit{külső}(U)$ azonban kezdőszelete $\alpha_2|\mathit{külső}(U)$-nek. Mivel a 13.1. tétel szerint az atomiság és a jólformáltság együttesen biztonságossági tulajdonság, amiből következik a kezdőszeletre zártság, így $\alpha_1|\mathit{külső}(U)$ teljesíti az atomi tulajdonságot. Mivel $\alpha_1|\mathit{külső}(U) = \alpha|\mathit{külső}(U)$, azt kaptuk, hogy $\alpha|\mathit{külső}(U)$ teljesíti az atomi tulajdonságot, és ezt akartuk bizonyítani.

Másrészről, tegyük fel, hogy α végtelen. Éppen most bizonyítottuk, hogy α minden α_1 véges kezdőszeletére teljesül, hogy $\alpha_1|\mathit{külső}(U)$ teljesíti az atomi tulajdonságot. $\alpha|\mathit{külső}(U)$ azonban határértéke az $\alpha_1|\mathit{külső}(U)$ alakú sorozatoknak. A 13.1. tétel szerint az atomiság és jólformáltság együttesen biztonságossági tulajdonság, így határértékre-zárt, amiből az következik, hogy $\alpha|\mathit{külső}(U)$ teljesíti az atomi tulajdonságot, amit bizonyítani akartunk. □

13.2.. olvasható/módosítható/írható atomi objektum megvalósítása olvasható/írható változókkal

olvasható/módosítható/írható atomi objektumoknak közös memóriájú rendszerekben olvasható/írható változókkal történő megvalósítását vizsgáljuk. (Lásd a 9.4. alfejezetet az olvasható/módosítható/írható változótípus definíciója végett.) Pontosabban, rögzítsünk egy tetszőleges n értéket, és tegyük fel, hogy a megvalósítandó objektumnak n kapuja van és minden kapuján támogat tetszőleges számú módosít bemenet műveletet.

Ha csak atomi objektumot akarunk kapni és nem tekintünk hibatűrést, akkor van egyszerű megoldás. Például a következő algoritmus.

> **OMÍmegvalOÍ algoritmus (vázlatosan)**
> A megvalósítandó objektumhoz tartozó olvasható/módosítható/írható változó utolsó értékét tároljuk az x olvasható/írható közös változóban. Az x-től különböző olvasható/írható közös változók egy halmazát használva, amikor a folyamatok az atomi objektumon akarnak műveletet végrehajtani, végrehajtják a próba részét a kitekintés-mentes kölcsönös kizárás algoritmusnak (például a 10.5.2. szakasz PETERSONnFOLY algoritmusát). Amikor a P_i folyamat a kölcsönös kizárás kritikus tartományába belép, akkor kizárólagos hozzáférést kap az x változóhoz. Ezután a P_i folyamat végrehajtja az olvasható/módosítható/írható műveletét, először `olvas`, majd `ír` lépést elvégezve. A P_i folyamat ezen lépések befejezése után végrehajtja a kilépés részét a kölcsönös kizárás algoritmusnak.

Ez az algoritmus azonban nem hibatűrő: egy folyamat hibázhat, miközben a kritikus tartományban tartózkodik, ezzel megakadályozva, hogy más folyamatok hozzáférjenek a szimulált olvasható/módosítható/írható változóhoz. Valójában ez a korlátozás nem véletlen. Mutatunk negatív eredményt még az egy hibát eltűrő esetre is.

13.11. tétel. . *Nem létezik olyan közös memóriájú rendszer, amely olvasható/írható közös változókat használ és az olvasható/módosítható/írható atomi objektumot valósítja meg és 1-hibás befejeződést biztosít.*

Bizonyítás. Indirekt módon tegyük fel, hogy van ilyen, mondjuk B rendszer. Legyen A az OMÍMEGEGYEZ megegyezés algoritmusa az olvasható/módosítható/írható közös memóriájú modellben, amit a 12.3. alfejezetben adtunk. A 12.9. tétel szerint A várakozásmentes megállást biztosít, tehát 1-hibás befejeződést is biztosít (amit megegyezés algoritmusokra a 12.1. alfejezetben definiáltunk). Alkalmazzuk most a 13.1.4. rész átalakítását A-ra, feltéve, hogy az egyedi olvasható/módosítható/írható közös változó helyén a B szerepel. Jelölje TRANSZ(A) a kapott rendszert.

13.12. segédtétel. . TRANSZ(A) *megoldása a 12. fejezet megegyezés feladatának, és 1-hibás megállást biztosít.*

Bizonyítás. A bizonyítás hasonló a 13.9. következmény bizonyításához. Először legyen α tetszőleges végrehajtása a TRANSZ(A)-nak és legyen U_i felhasználók tetszőleges kollekciója. Ekkor a 13.7. tétel szerint létezik olyan α' végrehajtása $A \times U$-nak, hogy α és α' megkülönböztethetetlen U számára. Mivel A megoldása a megegyezés feladatnak, így α' teljesíti a jólformáltság, megegyezés és helyesség feltételeket. Mivel α és α' megkülönböztethetetlen U számára, ezért α is teljesíti a jólformáltság, megegyezés és érvényesség feltételeket.

Az maradt hátra, hogy belássuk az 1-hibás befejeződés feltételt. Legyen α egy tetszőleges pártatlan végrehajtása TRANSZ(A)-nak és legyen U_i felhasználók tetszőleges kollekciója, ahol `kezd` esemény előfordul minden kapun és legfeljebb egy kapun van `megállít` esemény. Mivel B 1-hibás befejeződést biztosít, a 13.7. tétel miatt van olyan α' pártatlan végrehajtása $A \times U$-nak, hogy α és α' megkülönböztethetetlen U számára és ugyanazon kapukon van `megállít`, mint α-ban. Tehát α'-ben `kezd` előfordul minden kapun és `megállít` legfeljebb egy kapun.

Legyen i olyan kapu, amelyen $\texttt{megállít}_i$ esemény előfordul α-ban. Mivel α és α' ugyanazon kapukon tartalmaz `megállít` eseményeket, ezért van $\texttt{megállít}_i$ esemény α'-ben. A 1-hibás befejeződést biztosít, ezért van $\texttt{dönt}_i$ esemény α'-ben. Mivel α és α' megkülönböztethetetlen U számára, ezért $\texttt{dönt}_i$ is előfordul α-ban. Ez elegendő az 1-hibás befejeződés bizonyításához. □

A 13.1. alfejezet végén mondottak szerint azonban TRANSZ(A) maga is közös memóriájú rendszer az olvasható/írható közös memóriájú modellben. Ezzel TRANSZ(A) ellentmond a 12.8. tételnek, amely szerint megegyezés lehetetlen 1-hibás megállással olvasható/írható közös memóriájú modellben. □

13.3.. Közös memóriák atomi fényképei

A fejezet hátralévő részében adott típusú atomi objektumoknak más típusú atomi objektumokkal, vagy ami ezzel egyenértékű, közös változókkal történő megvalósításával foglalkozunk. Ebben az alfejezetben a fénykép atomi objektumot, a következőben pedig az olvasható/írható atomi objektumot vizsgáljuk.

Az olvasható/írható közös memória modellben hasznos lenne, ha egy folyamat képes lenne *fényképet* készíteni a közös memória teljes állapotáról. Természetesen az olvasható/írható modell közvetlenül nem nyújtja ezt a képességet – csak egyedi közös változók olvasását teszi lehetővé.

Ebben az alfejezetben ilyen fénykép objektum megvalósítását vizsgáljuk. A problémát úgy fogalmazzuk meg, mint megvalósítását egy speciális típusú atomi objektumnak, amit *fénykép* atomi objektumnak nevezünk, felhasználva az olvasható/írható közös memória modellt. A fénykép atomi objektumhoz tartozó változótípus V tartománya rögzített elemszámú vektorok halmaza, amelyek komponensei egy elemibb W tartományból vannak. Két fajta művelet van: egyedi vektor komponens írása, amit `módosít műveletnek` nevezünk, és egy teljes vektor olvasása, amit `fényképez` műveletnek nevezünk. A fénykép atomi objektum egyszerűsítheti olvasható/írható rendszer programozásának feladatát, megengedve, hogy egy folyamat hozzáférjen a teljes közös memóriához ezekkel a nagyhatású műveletekkel.

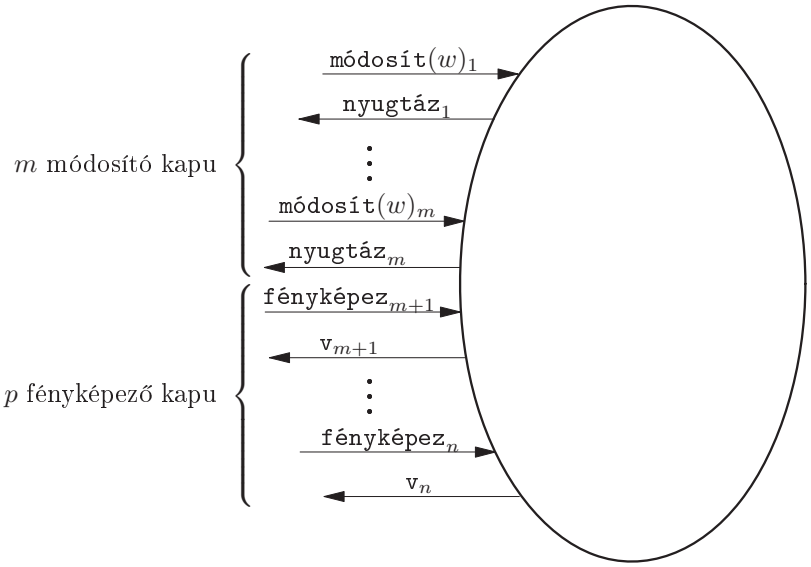
A feladat leírásával kezdjük, aztán egyszerű algoritmust adunk, amely korlátlan méretű olvasható/írható közös változókat használ. Majd megmutatjuk, hogy a konstrukció hogyan módosítható úgy, hogy a megvalósítás korlátos méretű közös változókat használjon. A 13.4.5. rész a fénykép atomi objektum egy alkalmazását tartalmazza az olvasható/írható atomi objektum megvalósításában.

13.3.1.. A feladat

Először definiáljuk azt a $\mathcal{T}$ változótípust, amely a fénykép atomi objektumhoz tartozik, amit *fénykép változótípusnak* nevezünk.

Induljunk ki egy W tartományból, és a w_0 kezdőértékből. A $\mathcal{T}$ tartománya az a V halmaz, amely a rögzített m elemű, W komponensű vektorok halmaza. A v_0 kezdőérték az a vektor, amelynek minden komponense w_0. A kérések `módosít`(i, w) alakúak, ahol $1 \leq i \leq m$ és $w \in W$ és a válasz ekkor `nyugtáz`, illetve `fényképez`, amire a válasz $v \in V$. A `módosít`(i, w) kérés az aktuális vektor i-edik komponensét w-ra változtatja és kiváltja a `nyugtáz` választ. A `fényképez` kérés nem okoz változást a vektoron, de válaszként visszaadja a teljes vektor értékét.

Ezután definiáljuk azt a külső felületet, amelyet használni fogunk. Feltételezzük, hogy pontosan $n = m + p$ kapu van, ahol m a vektorok rögzített hossza és p tetszőleges pozitív egész szám. Az első m kapu a `módosító kapu`, a maradék p kapu pedig a *fényképező* kapu. Minden $i, 1 \leq i \leq m$ kapun csak `módosít`(i, w) kérés megengedett – azaz az i-edik kapun csak a vektor i-edik komponense módosítható. Néha egyszerűsítjük a redundáns `módosít`$(i, w)_i$ jelölést, amely az i-edik kapura érkező `módosít`(i, w) műveletet jelöli és csak `módosít`(i, w)-t írunk. Minden $i, m+1 \leq i \leq n$ kapun csak `fényképez` kérés megengedett. Lásd a 13.7. ábrát. Jegyezzük meg, hogy az általános problémának egy speciális esetét vizsgáljuk, amikor is minden `módosít` kérés csak meghatározott kapuhoz érkezhet, ezért szekvenciálisan érkeznek. Lehetséges még általánosabban vizsgálni a kérdést, amikor több kapun keresztül lehet módosítani ugyanazt a vektor komponenst. Nyilvánvalóan még azt is megengedhetnénk, hogy ugyanazon kapun `módosít` és `fényképez`



13.7.. ábra. Fénykép atomi objektum külső felülete (a `megállít` műveleteket nem jelöltük).

kérés is előfordulhasson.

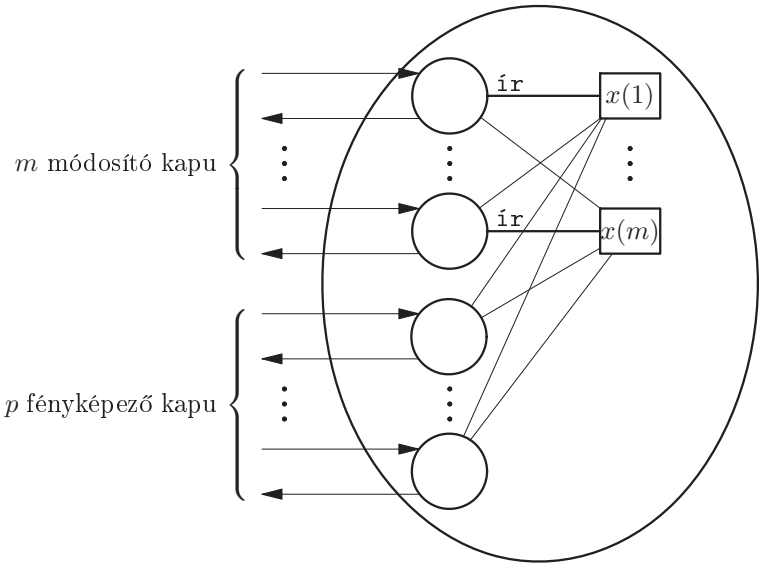
Olyan atomi objektum megvalósítását vizsgáljuk közös memóriájú rendszerrel és kapunként egy, azaz n folyamattal, amelyet a fenti változótípus és külső felület jellemez. Feltételezzük, hogy minden közös változó 1-író/n-olvasó olvasó/író közös változó. A vizsgálandó megvalósítás várakozásmentes megállást biztosít.

13.3.2.. Egy nemkorlátos változót használó algoritmus

A NemkorlátosFénykép algoritmus az $x(i), 1 \leq i \leq m$ m darab 1-író/n-olvasó olvasó/író közös változókat használja. Minden $x(i)$ változót írhatja a P_i folyamat (az, amelyik az i-edik kapuhoz tartozik, ami a `módosít`(i, w) művelet kapuja) és olvashatja bármelyik folyamat. Az architektúrát a 13.8. ábra szemlélteti. Minden $x(i)$ változó értékei a W elemei lehetnek, és még néhány kiegészítő érték, amire az algoritmusnak van szüksége. Az egyik ilyen kiegészítő érték a „rag" nem korlátos egész érték.

A NemkorlátosFénykép algoritmusban minden P_i folyamat az $x(i)$ változóba írja a `módosít`$_i$ kérésben szereplő értéket. Minden folyamatnak, amely `fényképez` műveletet hajt végre, valahogy ellentmondásmentes értékeit kell kapnia az összes közös változóknak, tehát olyanokat, amelyeket valamikor egyidejűleg tartalmazott a közös memória. A megoldás a következő két észrevételen alapszik.

1. Észrevétel. Tegyük fel, hogy minden P_i folyamat amikor végrehajt egy `módosít(w)`$_i$ kérést, nem csak a w értéket írja az $x(i)$ változóba, hanem egy „rag" értéket is, amely egyértelműen azonosítja a *módosítást*. Valahányszor egy



13.8.. ábra. NemkorlátosFénykép algoritmus architektúrája.

P_j folyamat `fényképez` műveletet kíván végrehajtani, akkor egymás után kétszer kiolvassa a közös változók értékeit, a második olvasást csak azután kezdi el, miután az elsőt befejezte, és ha azt találja, hogy minden $x(i)$ változóban a rag értéke megegyezik az első és a második olvasáskor, akkor a két olvasáskor kapott közös vektor értékek ténylegesen egy olyan vektort jelentenek, amely a közös memória tartalma volt valamely időpontban a `fényképez` művelet intervallumában. Nevezetesen, ez a vektor a közös memória tartalma az első olvasás befejezése és a második olvasás megkezdése közötti intervallum bármely pontjában.

Az első észrevétel a következő egyszerű algoritmust sugallja. Minden P_i folyamat `módosít(w)` művelet végrehajtásakor a w értéket beírja az $x(i)$ változóba az egyedi lokális rag értékkel együtt. A rag értékét úgy határozza meg, hogy az az i-re vonatkozó első `módosít` esetében 1, és minden további i-re vonatkozó `módosít` eggyel növeli ezt az értéket.

Minden P_j folyamat, amikor végrehajt egy `fényképez` műveletet, akkor ismételten kiolvassa az összes közös változó értékét, mindaddig ismételve, amíg két egymást követő kiolvasás „ellentmondásmentes” nem lesz, azaz minden $x(i)$-hez tartozó rag megegyezik. Amikor ez bekövetkezik, akkor a második olvasáskor kapott vektor (aminek meg kell egyeznie az első olvasáskor kapott értékkel) értékét adja vissza a `fényképez` művelet.

Könnyen látható, hogy amikor az egyszerű algoritmus befejezi a műveletet, akkor a válasz mindig „helyes” lesz, vagyis kielégíti a jólformáltság és az atomiság feltételt. Még a hibamentes megállást sem teljesíti azonban, mert előfordulhat hibátlan folyamat esetében is, hogy a `fényképez` műveletet sohasem fejeződik be, mert folyamatosan új `módosít` hívódik, miközben a `fényképez` aktív. Ebből

a problémából a kivezető utat a második észrevétel adja.

2. Észrevétel. Ha a P_j folyamat, amíg a `fényképez` végrehajtását végezve ugyanazon $x(i)$ változó ismételt kiolvasásakor a rag négy különböző – rag_1, rag_2, rag_3, rag_4 – értékét látja, akkor biztos lehet benne, hogy egy `módosít`$_i$ művelet teljes egészében benne van az aktuális `fényképez` intervallumában. Nevezetesen, annak a `módosít`$_i$ műveletnek, amelyik a rag_3 értéket írta, teljes egészében benne kell lennie az aktuális `fényképez` intervallumában.

Hogy ez tényleg így van, először is vegyük észre, hogy az a `módosít`, amely a rag_3-at írta, később kezdődött, mint amikor `fényképez` elkezdődött. Ez azért igaz, mert később kezdődött, mint amikor az a `módosít` befejeződött, amely rag_2-t írta és ennek befejezése később volt, mint amikor `fényképez` elkezdődött (mert `fényképez` látta a rag_1 értéket).

Másodszor, vegyük észre, hogy az a `módosít`, amely a rag_3-at írta, előbb befejeződött, mint `fényképez`. Ez azért igaz, mert előbb befejeződött, mint amikor `módosít` a rag_4-et írta és `fényképez` látta a rag_4 értéket.

Az első és második észrevétel sugallja a NemkorlátosFénykép algoritmust. Ez kiterjeszti a fenti egyszerű algoritmust oly módon, hogy mielőtt a P_i `módosít` folyamat írja az $x(i)$ változót, előbb végrehajtja saját `belső_fényképez` szubrutinját, amely olyan mint a `fényképez`. Aztán miután végrehajtotta az értéke és a rag beírását $x(i)$-be, a saját `belső_fényképez` érékét is beírja az $x(i)$-be. Az a `fényképez`, amely többször ismételt egymást követő olvasásra sem látja ugyanazt a rag értéket, visszatérhet a `belső_fényképez` értékével, mint alapértelmezett fényképez értékkel. A pontosabb leírás ezután következik. Ebben a leírásban minden közös változó egy több mezőt tartalmazó rekord, a mezőkiválasztásra a pont-jelölést használjuk.

NemkorlátosFénykép algoritmus (vázlatosan)

Minden $x(i), 1 \leq i \leq m$ változót írhatja a P_i folyamat és olvashatja bármelyik. Minden változó a következő mezőket tartalmazza:

érték $\in W$, kezdetben w_0,
rag $\in \mathbb{N}$, kezdetben 0,
nézet, W elemeiből álló, $\{1, \ldots, m\}$ számokkal indexelt vektor, kezdetben mind w_0.

Amikor `fényképez`$_j$ bemenet jelentkezik a $j, m+1 \leq j \leq n$ kapun, akkor a P_j folyamat a következőképpen viselkedik. Ismételten végrehajtja `olvas` műveletek egy halmazát, ahol egy halmaz m `olvas` műveletből áll, minden $x(i), 1 \leq i \leq m$ közös változóra egy művelet, tetszőleges sorrendben elvégezve. Ezt addig ismétli, amíg az alábbi feltétel valamelyike nem teljesül:

1. `olvas` műveletek két egymást követő halmazára teljesül, hogy minden i-re $x(i).rag$ értéke megegyezik az előzőével. Ekkor a `fényképez` művelet az utolsó olvasáskor kapott az $x(i).érték, 1 \leq i \leq m$ komponensekből álló vektorral tér vissza (ami megegyezik az előző olvasás eredményével.)

2. Valamely i-re az $x(i).rag$ négy különböző értéke fordul elő. Ekkor `fényképez` azzal az $x(i).nézet$ értékkel tér vissza, amelyik a négy $x(i).rag$ közül a harmadikhoz tartozik.

Amikor `módosít`$_i$ bemenet jelentkezik, akkor a P_i folyamat a következőt teszi. Először végrehajtja a `belső_fényképez` műveletet. Ez ugyanazt a munkát jelenti, mint a `fényképez`, kivéve, hogy az eredményt a P_i folyamat lokálisan tárolja és nem adja vissza a felhasználónak. Másodszor, a P_i folyamat végrehajtja $x(i)$ egy egyszerű írását, beállítva $x(i)$ három mezőjének értékét:

1. $x(i) := w$.
2. $x(i).rag$ az i-re nem használt legkisebb rag értéke legyen.
3. $x(i).nézet$ a `belső_fényképez` által visszaadott vektor legyen.

Végül a P_i folyamat `nyugtáz` kimenetet ad.

13.13. tétel. . *A* NemkorlátosFénykép *algoritmus fénykép atomi objektum, amely várakozásmentes megállást biztosít.*

Bizonyítás. A jólformáltság teljesülése világos. A várakozásmentes megállást is könnyű belátni: az a kulcsa, hogy minden `belső_fényképez` befejeződik legfeljebb $3m + 1$ egymást követő olvasó halmaz után. Ez azért van, mert $3m + 1$ olvasó halmaz után kell lenie vagy két egymást követő azonos eredményt adónak, vagy valamely $x(i)$ változónak négy különböző raggal. Bármelyik esetben a művelet befejeződik.

Az atomiság megmutatása maradt hátra. Rögzítsük a NemkorlátosFénykép algoritmus és a felhasználók egy tetszőleges α végrehajtási sorozatát. A 13.10. lemma értelmében az általánosság megszorítása nélkül feltételezhetjük, hogy α nem tartalmaz befejezetlen műveletet. Leírjuk, hogy hogyan lehet sorbarendező pontokat elhelyezni minden műveletre.

Minden `módosít` műveletre a sorbarendező pontot az `ír` előfordulásának helyére rakjuk. `fényképez` művelet sorbarendező pontjának elhelyezése kicsit bonyolultabb. A sorbarendező pont megadása végett hasznosnak bizonyul nem csak a `fényképez`, de a `belső_fényképez` művelet számára is meghatározni sorbarendező pontot.

Először tekintsük azt az esetet, amikor a `fényképez` vagy `belső_fényképez` azért ér véget, mert két egymást követő olvasás halmaz ugyanazt eredményezi. Ebben az esetben a sorbarendező pontot helyezzük az első olvasás vége és a második kezdete közé tetszőlegesen.

Másodszor, tekintsük azt a `fényképez` és `belső_fényképez` műveletet, amelyek azért fejeződnek be, mert négy különböző rag értéket találnak ugyanazon változóban. Ezen műveletekre a sorbarendező pontokat lépésről-lépésre, a válaszuk sorrendjében helyezzük el. Jegyezzük meg, hogy minden ilyen π műveletre az a vektor, amit visszaad a művelet, annak a ϕ `belső_fényképez` műveletnek az eredménye, amelynek intervallumát teljes egészében tartalmazza π intervalluma. Vegyük észre, hogy ehhez a ϕ művelethez már rendeltünk sorbarendező pontot, mert előbb fejeződött be, mint π. Helyezzük π sorbarendező pontját ϕ sorbarendező pontjára.

Könnyű látni, hogy minden sorbarendező pont a megkívánt intervallumban van. A `módosít` valamint azon `fényképez` és `belső_fényképez` művelet esete, amikor a művelet azért ér véget, mert két egymást követő olvasás ellentmondásmentes vektort talál, nyilvánvaló. Azon `fényképez` és `belső_fényképez` műveletekre, amelyek négy különböző rag előfordulása miatt érnek véget, a bizonyítás a válasz események α-beli száma szerinti indukcióval bizonyítható.

Azt kell még megmutatni, hogy a műveleteknek a sorbarendező pontjaira való zsugorításával a fénykép változótípus történetét kapjuk. Ezért először jegyezzük meg, hogy α minden α' véges kezdőszeletére pontosan egy olyan V vektor van, amely α'-beli `ír` események eredménye. Nevezzük ezt az α' utáni *helyes vektornak*. Elegendő megmutatni, hogy minden `fényképez` és `belso_fényképez` azt a vektort adja vissza, amely α-nak a művelet sorbarendező pontjáig vett kezdőszelete utáni helyes vektor. Még erősebben, azzal érvelünk, hogy minden `fényképez` művelet a sorbarendező pontjához tartozó helyes vektort adja vissza.

Ez világos abban az esetben, ha a művelet két egymást követő ellentmondásmentes vektor jelenléte miatt fejeződik be. A másik eset bizonyítása elvégezhető a válasz események α-beli száma szerinti indukcióval. □

Bonyolultságelemzés.. A NemkorlátosFénykép algoritmus m számú közös változót használ, minden változó által felvehető értékek halmaza nem korlátos. Még akkor sem, ha a W tartomány véges, mert a rag értékek halmaza végtelen. Hibátlan folyamat időbonyolultsága a következőképpen számítható. A `fényképez` végrehajtása legfeljebb $3m + 1$ halmaz olvasást végez, vagy $(3m + 1)m$ közös memória hozzáférést, amelynek összideje $\mathcal{O}(m^2 l)$, ahol l a folyamat egy lépésének felső időkorlátja. Hibátlan folyamat `módosít` végrehajtása során szintén $\mathcal{O}(m^2)$ közös memória hozzáférést végez, aminek összideje $\mathcal{O}(m^2 l)$, a `belső_fényképez` művelet miatt.

13.3.3.. Egy korlátos változókat használó algoritmus*

Az fő feladat a NemkorlátosFénykép algoritmussal, hogy korlátlan méretű közös változókat használ a ragok értéke tárolására, mert azok nem korlátosak. Ebben az alfejezetben javított algoritmust vázolunk, amit KorlátosFénykép algoritmusnak nevezünk. Ebben a nem korlátos ragokat korlátos adattal helyettesítjük. Annak érdekében, hogy elérjük a hatékonyság javítását, a KorlátosFénykép algoritmus olyan mechanizmust használ, amely bonyolultabb az egyszerű ragnál.

Jegyezzük meg, hogy a NemkorlátosFénykép algoritmusban a ragokat csak arra használtuk, hogy a `fényképez` és a `belső_fényképez` műveleteket végző folyamatok kideríthessék, hogy közben új `módosít` művelet hajtódott végre. Ez az információ azonban a rag módszernél kevésbé hatékony módon is közölhető, nevezetesen, *kézfogás bitek* és *átkapcsoló bit* használatával.

A kézfogás bitek a következőképpen működnek. Most $n + m$ közös változó van, ugyanúgy, mint a NemkorlátosFénykép algoritmusban az $x(i), 1 \leq i \leq m$, és még az $y(j), 1 \leq j \leq n$ új változók. Minden $x(i)$ változót írhatja a P_i módosító folyamat és olvashatja minden folyamat, mint korábban. Minden

$y(j), 1 \leq j \leq m$ változót írhatja a P_j módosító folyamat (különösen a P_j folyamat `belső_fényképez` része), és olvashatja minden módosító folyamat, továbbá, minden $y(j), m+1 \leq j \leq n$ változót írhatja a P_j `fényképez` folyamat és olvashatja minden módosító folyamat. Jegyezzük meg, hogy ellentétben a NemkorlátosFénykép algoritmussal, a KorlátosFénykép algoritmusban `fényképez` és `belső_fényképez` végrehajtása közös memóriába írást is tartalmaz.

Minden $i, 1 \leq i \leq m$ módosító folyamathoz tartozik n pár *kézfogás bit*, minden P_j folyamathoz egy pár. Az (i, j) bitpár lehetővé teszi, hogy a P_i folyamat közölje a P_j folyamattal, hogy új módosítást végzett a P_i folyamat, és a P_j folyamat nyugtázhatja, hogy vette ezt az információt. Speciálisan, az $x(i)$ változó tartalmaz egy n-elemű *közös* bitvektort, ahol a P_i folyamat az $x(i)$ változó *közös*(j) komponensében – amit $x(i).$*közös*(j)-vel jelölünk – közli a P_j folyamattal az általa végzett új módosít művelet megjelenését. Az $y(j)$ változó egy m-elemű `nyugtáz` bitvektort tartalmaz, ahol az $y(j)$ `nyugtáz`(i) i-edik komponensét – amit $y(j).$ `nyugtáz`(i)-vel jelölünk – a P_j folyamat arra használ, hogy nyugtázza a P_i folyamatnak, hogy észrevette, hogy a P_i folyamat új módosítást végzett. Tehát minden (i, j)-re a kézfogás bitpárok az $x(i).$*közös*(j) és $y(j).$`nyugtáz`(i) bitek.

A kézfogás bitek durván a következőképpen működnek. Amikor a P_i folyamat `módosít`(w) műveletet hajt végre, akkor azzal kezdi, hogy kiolvassa az összes $y(j).$`nyugtáz`(i) kézfogás bitet. Ezután végrehajtja az $x(i)$ változóba való írást, ezt úgy végzi, mint a NemkorlátosFénykép algoritmusban, a w értéket és a belső felvétel `nézet` választ beírja, és ráadásul beírja a kézfogás biteket *közös*-be. Nevezetesen, minden j-re a *közös*(j) bit értéke az $y(j).$`nyugtáz`(i) bit művelet előtti értékének *ellentettje* lesz.

A `fényképez` vagy `belső_fényképez` műveletet végző P_j folyamat olvasások két halmazát próbálja ismételten elvégezni, arra figyelve, hogy semmi sem változott a két olvasás között. Most azonban a kézfogás biteket, és nem égész értékű ragokat használunk a változás kiderítésére. Nevezetesen, minden kísérlet, amely két ellentmondásmentes olvasást akar kideríteni, előtt a P_j folyamat először kiolvassa az összes $x(i).$`nyugtáz`(j) kézfogás bitet és az $y(i).$`nyugtáz`(j) értékét az $x(i).$`nyugtáz`(j) bitnek az éppen kiolvasott értékére állítja be. (Tehát a `módosít` műveletek a kézfogás biteket ellentétesre, a `fényképez` és `belső_fényképez` műveletek pedig azonosra igyekeznek beállítani.) A P_j folyamat két olvasás között a *közös*(j) kézfogás biteket figyeli a változás kiderítésére, ha $2m+1$ különböző kísérletben tapasztal ilyen változást, akkor biztos lehet abban, hogy egyazon P_i folyamat négy különböző módosításának eredményét látta, ezért veheti a harmadik által eredményezett *nézet* értékét.

A most vázolt kézfogás protokoll egyszerű, és „helyes” abban az értelemben, hogy minden alkalommal, amikor `fényképez` vagy `belső_fényképez` művelet változást észlel, akkor ténylegesen történt új módosítás. Kiderül azonban, hogy a kézfogás módszer nem elegendő minden módosítás kiderítésére – előfordulhat, hogy egy P_i folyamat által végzett két egymást követő módosítást nem vesz észre egy másik P_j folyamat. Tekintsük az alábbi helyzetet.

13.3.1. példa. A kézfogás bitek elégtelensége

Tegyük fel, hogy a végrehajtás egy adott időpontjában $x(i).közös(j) = 0$ és $y(j).közös(i) = 1$, vagyis a kézfogás bitek azt mondják j-nek, hogy az i módosításai nem egyenlőek. Ekkor a következő események előfordulhatnak a leírás sorrendjében. (Az i és j folyamatokhoz tartozó műveletek két külön oszlopban vannak.)

módosít$(w_1)_i$
i olvassa $y(i)$.nyugtáz$(j) = 1$ értékét
i írja w_1-et és $x(i).közös(j) := 0$ beállítást végez
nyugtáz$_i$
módosít$(w_2)_i$
i olvassa $y(j)$.nyugtáz$(i) = 1$ értékét
fényképez$_j$
j olvassa $x(i).közös(j) = 0$ értékét
j $y(j).közös(i) := 0$ beállítást végez
j olvassa $x(i).közös(j) = 0$ értékét
i írja w_2-t és $x(i).közös(j) := 0$ beállítást végez
nyugtáz$_i$
j olvassa $x(i).közös(j) = 0$ értékét és azt gondolja,
hogy $x(i)$ értéke nem változott az előző olvasás óta

Az eseménysorban a P_j folyamat háromszor olvassa az $x(i).közös(j)$ értékét. A első ezek közül csak előzetes teszt, a második és harmadik két ellentmondásmentes olvasás megtalálását célzó kísérlet részei. A P_j folyamat befejezi a kísérlet ismétlését, mert a második és harmadik olvasás azonos, ami azt jelenti, hogy nem történt módosítás. Ez azonban hibás.

A feladat kiküszöbölése végett minden $x(i)$ változót kiegészítünk egy *kapcsoló* bittel, amelyet a P_i folyamat minden *írás* lépésben átkapcsol. Ez biztosítja, hogy minden módosít megváltoztatja az $x(i)$ közös változót. Egy kicsit részletesebben, a protokoll az alábbiak szerint működik.

KorlátosFénykép algoritmus (vázlatosan)

Minden $x(i)$ közös változót írhatja az $i, 1 \leq i \leq m$, és olvashatja minden folyamat. A változó az alábbi mezőket tartalmazza:

érték $\in W$, kezdetben w_0,
közös, $\{0,1\}$ bitvektor az $\{1,\ldots,m\}$ számokkal indexelve,
, kezdetben mind 0,
kapcsoló $\in \{0,1\}$, kezdetben 0,
nézet, W elemeiből álló, $\{1,\ldots,m\}$ számokkal indexelt vektor,
kezdetben mind w_0.

Minden $y(j)$ közös változót írhatja az $j, 1 \leq j \leq n$, és olvashatja minden $i, 1 \leq i \leq m$ folyamat. A változó az alábbi mezőt tartalmazza:

`nyugtáz`, $\{0,1\}$ bitvektor az $\{1,\ldots,m\}$ számokkal indexelve, kezdetben mind 0.

Amikor `fényképez`$_j$ bemenet jelentkezik a $j, m+1 \leq j \leq n$ kapun, akkor a P_j folyamat a következőképpen viselkedik. Ismételve kísérletet tesz, hogy két „ellentmondásmentesnek" látszó olvasást kapjon. Pontosabban, minden kísérletben a P_j folyamat először minden i-re, $1 \leq i \leq m$ tetszőleges sorrendben kiolvassa a lényeges $x(i).közös(j)$ kézfogás biteket. Ezután a P_j folyamat minden i-re beállítja az $y(j)$.`nyugtáz`(i) értékét az $x(i).közös(j)$ kiolvasott értékére, és mindezt egy `ír` lépésben teszi. Majd két teljes olvas halmazt hajt végre, az első előbb fejeződik be, mint amikor a második elkezdődik. Minden i-re $x(i).közös(j)$ és $x(i).kapcsoló$ azonos a két olvasásra, továbbá, a $közös(j)$ közös érték megegyezik azzal, amit a P_j folyamat olvasott az aktuális kísérlet kezdetén, aztán a `fényképez` az utolsó olvasáskor kapott $x(i).érték$ elemek alkotta vektorral tér vissza. Különben a P_j folyamat feljegyzi, hogy mely $x(i)$ változók módosultak.

Ha a P_j folyamat valamikor azt tapasztalja, hogy három különböző kísérletben ugyanazon $x(i)$ változott, tekintsük ezen kísérletek közül a másodikat. A `fényképez`$_j$ művelet ezen kísérlet során az $x(i)$ utolsó olvasásakor kapott $x(i).nézet$ elemek alkotta vektorral tér vissza. (Biztosított, hogy ez a vektor olyan `módosít` művelet elvégzésekor íródott, amelyet teljesen tartalmaz az adott `fényképez`$_j$ intervalluma.)

Amikor `módosít`$(w)_i$ bemenet jelentkezik az $i, 1 \leq i \leq m$ kapun, akkor a P_i folyamat a következőképpen viselkedik. Először kiolvassa a lényeges $y(j)$.`nyugtáz`(i) ,$1 \leq j \leq n$ kézfogás biteket. Másodszor, végrehajt egy `belső_fényképez` műveletet, amely megegyezik a `fényképez` művelettel, attól eltekintve, hogy a kapott vektort nem adja vissza a felhasználónak. Harmadszor, a P_i folyamat elvégzi az $x(i)$ változó módosítását az alábbi négy mezőjének megváltoztatásával:

1. $x(i).érték := w$;
2. minden j-re $x(i).közös(j)$ értékét $y(j)$.`nyugtáz`(i) azon értékére állítja be, amelyet $y(i)$ kezdeti olvasásakor kapott;
3. $x(i).kapcsoló$ értéke ellenkezőjére változik;
4. $x(i).nézet$ értéke megkapja a `belső_fényképez` által visszaadott vektort.

Végül a P_i folyamat `nyugtáz`$_i$ kimenetet ad.

13.14. tétel. . *A* KorlátosFénykép *algoritmus fénykép atomi objektum és várakozásmentes megállást biztosít.*

Bizonyításvázlat. A jólformáltságot és a várakozásmentességet könnyű belátni, mint a 13.13. tételben a NemkorlátosFénykép algoritmusnál. Az atomiság megmutatása maradt hátra. Az érvelés hasonló a NemkorlátosFénykép algoritmusnál használthoz.

Ismét, legyen α tetszőleges végrehajtás és az általánosság megszorítása nélkül tegyük fel (a 13.10. lemma fényében), hogy α nem tartalmaz befejezetlen műveletet. A sorbarendező pontokat ugyanúgy adjuk meg, mint a NEMKORLÁTOS-FÉNYKÉP algoritmusnál tettük. Például, egy `fényképez` vagy `belső_fényképez` műveletre, amely azzal fejeződik be, hogy két egymást követő olvasási halmaz ugyanazt eredményezi, helyezzük a sorbarendező pontot az első olvasás vége és a második olvasás kezdete közé. Mint korábban, most is könnyű belátni, hogy a sorbarendező pontok a megkívánt intervallumokba esnek. Annak bizonyítása maradt hátra, hogy a műveletek intervallumainak a sorbarendező pontokra való zsugorítása a fénykép változó történetét eredményezi. Mint korábban, most is elég megmutatni, hogy minden befejezett `fényképez` vagy `belső_fényképez` művelet α-nak a sorbarendező pontig tekintett kezdőszeletére nézve helyes vektort ad vissza.

Most nem olyan könnyű belátni ezt a tulajdonságot olyan `fényképez` vagy `belső_fényképez` műveletre, amelyik két egymást követő ellentmondásmentes olvasást találva fejeződik be. Ennek igazolásához elegendő bizonyítani a következő elvárás teljesülését.

13.15. segédtétel. . *Ha egy* `fényképez` *vagy* `belső_fényképez` *művelet két egymást követő ellentmondásmentes olvasást találva fejeződik be, akkor minden i-re teljesül az alábbi. A P_i folyamatnak nincs olyan írás eseménye, amely az $x(i)$ változó első és második olvasása közé esne.*

Bizonyítás. Indirekt módon bizonyítunk, részletesebb műveleti érvelést használva. Tegyük fel, hogy a j kapun `fényképez` művelet befejeződik két egymást követő ellentmondásmentes olvasást találva, és a P_i folyamat `ír` művelete jelentkezik az $x(i)$ első π_1 és második π_2 olvasása közben. (Az érvelés ugyanez a `belső_fényképez` művelet esetében.) Legyen ϕ az utolsó ilyen `ír`, azaz $x(i)$-nek π_2 előtti írása.

Amiatt, hogy a két olvasás ellentmondásmentes, következik, hogy a π_1 és π_2 által olvasott $x(i).közös(j)$ értékek megegyenek, továbbá, megegyezik a π_1 előtt $y(j).\texttt{nyugtáz}(i)$-be írt utolsó értékkel (ami része a P_j folyamat sikeres kísérletének, hogy ellentmondásmentes olvasások halmazát kapja). Jelölje b a közös értéket, és legyen π_0 az utolsó ilyen írás esemény. A konzisztencia miatt az $x(i).kapcsoló$ értékek a π_1 és π_2 olvasáskor megegyeznek. Jelölje t ezt a közös értéket.

Mivel ϕ az $x(i)$ utolsó írása π_2 előtt, ezért $x(i).közös(j) := b$ és $x(j).kapcsoló := t$ értékadásokat végre kell hajtania. A ϕ-t tartalmazó `módosít` művelet $y(j)$-nek egy korábbi ψ olvasását biztosan tartalmazza. A `módosít` művelet viselkedése miatt a ϕ által olvasott $y(j).\texttt{nyugtáz}(i)$ értéke szükségképpen $\overline{b}$. (Itt a felülvonás jelölést használjuk a bit-komplemens jelölésére.) Ebből következik, hogy π_0 megelőzi ψ-t.

Tehát, a különböző `olvas` és `ír` események sorrendje szükségképpen a következő. (Az i és j folyamatot magukban foglaló műveletek különböző oszlopokban jelennek meg.)

ψ : A P_i folyamat olvasása, amikor $y(j).\texttt{nyugtáz}(i) = \overline{b}$;

π_0 : A P_j folyamat írása, $y(j)$.`nyugtáz`$(i) := b$ értékadással;
π_1 : A P_j folyamat olvasása, amikor $x(i).közös(j) = b$ és
$x(j).kapcsoló = t$.
ϕ : A P_i folyamat írása, az $x(i).közös(j) := b$ és
$x(j).kapcsoló := t$ értékadással;
π_2 : A P_j folyamat olvasása, amikor $x(i).közös(j) = b$ és
$x(j).kapcsoló = t$.

Jegyezzük meg azonban, hogy a ψ olvasás esemény része ugyanazon a `módosít` műveletnek, amelynek része a ϕ írás esemény. Ebből következik, hogy a π_1 és π_2 olvasás esemény a P_i folyamat két egymást követő írásának értékével tér vissza. π_1 és π_2 azonban a *kapcsoló* bitek egyenlő értékével térnek vissza, ami ellentmond a kapcsoló bitek kezelésének. □

Ez bizonyítja a 13.15. elvárást, amiből következik, hogy minden `fényképez` és `belső_fényképez` művelet, amely két ellentmondásmentes olvasás megtalálásával befejeződik, ténylegesen helyes vektorral tér vissza α-nak a sorbarendező pontig tartó kezdőszelete után. A másik fajta `fényképez` és `belső_fényképez` műveletet tekintve az érvelés ugyanaz, mint a NemkorlátosFénykép algoritmusnál, tehát az α-beli válasz események száma szerinti indukció alkalmazható. □

Bonyolultságelemzés.. A KorlátosFénykép algoritmus $n+m$ közös változót használ. Minden $x(i)$változó $|W|^{m+1}2^{n+1}$ értéket vehet fel, és minden $y(j)$ változó 2^{n+1} értéket vehet fel. Az időbonyolultságot tekintve, minden hibátlan folyamat `fényképez` végrehajtásakor legfeljebb $2m+1$ kísérletet tesz ellentmondásmentes olvasási halmaz megtalálására. Minden kísérletben legfeljebb $4m$ közös memória elérést végez, aminek összideje $\mathcal{O}(m^2 l)$. Ugyanezek a korlátok teljesülnek a `módosít` műveletre.

Fénykép objektumok használata olvasható/írható közös memóriájú rendszerek programozásánál.. A fénykép közös változók a közös memóriák hatékony típusait reprezentálják. Például, egyetlen fénykép közös változót használva jelentősen egyszerűsíthető a 10.7. alfejezetben adott Váróterem kölcsönös kizárás algoritmusa. Ez gyakorlatként elvégezhető.

A 13.1.4. szakasz technikáját és az ebben a részben adott fényképez algoritmust felhasználva egy A algoritmus, amely fénykép közös változókat használ, átírható olyan ekvivalens algoritmussá, amely csak kizárólagosan-írható/megosztottan-olvasható olvasható/írható közös változót használ. Ez az átalakítás megkívánja, hogy egy egyszerű korlátozást tegyünk A-ra, amit a 13.1.4. részben tárgyaltunk. (Hasonlóan, technikailag megköveteljük, hogy az átalakításban szereplő fénykép atomi objektum minden kapujához az A egy folyamata tartozzék, az A i-edik folyamata ugyanazon i kapun mind *módosít*, mind `fényképez` műveletet kiadhat. Nem okoz gondot a fénykép atomi objektum külső felületének és megvalósításának olyan mósosítása, amely ezt lehetővé teszi.)

olvasható/írható/fényképez változók.. A fényképez közös változó egy hasznos változata az, amelyik csak `módosít` és `fényképez` műveleteket enged meg, az

olvas/módosít/fényképez közös változó, amely `olvas` műveletet támogat a közös vektor bizonyos helyein a teljes vektort visszaadó `fényképez` művelet mellett. Természetesen az olvas/módosít/fényképez modell nem erősebb, mint a fénykép változó modellje, mert az `olvas` művelet megvalósítható `fényképez` művelettel. Olvas/módosít/fényképez közös változók használata azonban hatékonyabb programozást tesz lehetővé, mert olvas/módosít/fényképez atomi objektum megvalósítható úgy, hogy az `olvas` művelet nagyon gyors. Ez elvégezhető gyakorlatként (lásd 13-19. gyakorlat).

13.4.. olvasható/írható atomi objektumok

Az ovas/ír közös változók (regiszterek) a alapvető építőelemei a közös memóriájú multiprocesszoroknak. Ebben az alfejezetben hathatós megosztottan-írható/megosztottan-olvasható regisztereknek kevésbbé hatásos regiszterekkel, mint az egyedi-író/egyedi-olvasó regiszter, történő megvalósításával foglalkozunk. Pontosabban, megosztottan-írható/megosztottan-olvasható olvasható/írható atomi objektumoknak egyedi-író/egyedi-olvasó közös változókkal történő megvalósításának problémáját tekintjük.

13.4.1.. A feladat

Rözítsünk egy V tartományt és egy $v_0 \in V$ kezdőértéket.

A 13.1.1. szakaszban leírtuk az 1-író/2-olvasó olvasható/írható V tartományú atomi objektum egy külső felületét. Általánosan, egy m-író/p-olvasó olvasható/írható V tartományú atomi objektum külső felülete hasonlóan adható meg, ahol az $1, \ldots, m$ kapuk az író kapuk, az $m+1, \ldots, m+p$ kapuk pedig az olvasó kapuk. Ismét az $n = m + p$ jelölést használjuk.

Mivel olvasható/írható atomi objektumokat olvasható/írható közös változókkal akarunk megvalósítani, ezért valamilyen módon meg kell különböztetnünk a felhasználók által alkalmazott magasabb szintű olvas és ír műveleteket az olvasható/írható közös változókra vonatkozó alacsonyabb szintű olvas és ír műveletektől. Azt a megállapodást alkalmazzuk, hogy a magasabb szintű műveleteket nagybetűkkel írjuk. Tehát az $i, 1 \leq i \leq m$ kapuhoz tartozik a $\texttt{ÍR}(v)_i$ bemenet, $v \in V$-re, a $\texttt{NYUGTÁZ}_i$ kimenet, és a $j, m+1, \ldots, n$ kapukhoz a $\texttt{OLVAS}_j$ bemenet valamint a $v_j, v \in V$ kimenet. (Az értékeket nem írjuk nagybetűkkel.) Vannak $\texttt{MEGÁLLÍT}_i$ $(1 \leq i \leq n)$ bemenetek is.

Ilyen m-író/p-olvasó atomi objektum megvalósítását tekintjük, ahol $n = m+p$, felhasználva egy közös memóriájú rendszert és n folyamatot, minden kapuhoz egyet. Feltételezzük, hogy a rendszer minden közös változója olvasható/írható közös változó, de az olvasók és írók száma a tárgyalandó különböző algoritmusok esetében más és más lehet. Minden általunk vizsgált megvalósítás várakozásmentes megállást biztosít.

13.4.2.. Egy másik lemma atomiság kimutatására

Egy technikai lemmával kezdjük, amely hasznosnak bizonyul annak igazolására, hogy egy olvasó/író atomi objektum külső felületének műveleteiből álló β sorozat kielégíti az olvasó/író objektumok atomiság feltételét. Ez a lemma felsorol három feltételt, amelyek egy, a β műveletein értelmezett részbenrendezést tartalmaznak. Ha létezik olyan rendezés, amely kielégíti ezt a három feltételt, akkor biztosan megadhatók a sorbarendező pontok úgy, hogy az kielégítse az atomiság tulajdonságot. Algoritmusok érvelése során, gyakran könnyebb megmutatni ilyen rendezés létezését, mint ténylegesen megadni a sorbarendező pontokat.

13.16. lemma. . *Legyen β egy olvasó/író atomi objektum külső felületének műveleteiből alló (véges avagy végtelen) sorozata. Tegyük fel, hogy β jólformált minden i-re és nem tartalmaz befejezetlen műveletet. Jelölje Π a β összes műveletének halmazát. Tegyük fel, hogy $\prec$ olyan irreflexív részbenrendezés a műveletek Π halmazán, amely kielégíti az alábbi négy feltételt.*

1. *Ha a π válasz eseménye megelőzi a ϕ kérés eseményét β-ban, akkor $\phi \prec \pi$ nem lehetséges.*
2. *Ha π `ÍR` művelet Π-ben és $\phi \in \Pi$ tetszőleges művelet, akkor vagy $\pi \prec \phi$ vagy $\phi \prec \pi$.*
3. *Minden `OLVAS` művelet által visszaadott érték az az érték, amelyet a $\prec$ szerinti utolsó `ÍR` művelet írt (vagy v_0, ha nincs ilyen megelőző `ÍR`).*

Ekkor β rendelkezik az atomi tulajdonsággal.

Bizonyítás. Azzal kezdjük, hogy minden $\pi \in \Pi$ műveletre csak véges sok olyan ϕ művelet lehet, hogy *phi* $\prec \pi$. Most megmutatjuk, hogyan lehet $*_\pi$ sorbarendező pontokat elhelyezni minden $\pi \in \Pi$ műveletekre. Nevezetesen, minden $*_\pi$ sorbarendező pontot közvetlenül π utolsó kérése és az olyan ϕ műveletek kérése utánra helyezzük, amelyre $\phi \prec \pi$. A bizonyítás elején megfogalmazott kívánalom biztosítja, hogy ez a pozíció jól definiált. Sorbarendezzük a $*$-okat, ezáltal folytonos elhelyezést kapunk, olyan módon, hogy az ellentmondásmentes legyen a műveletek $\prec$ rendezésével, vagyis ha π és ϕ két olyan művelet, hogy a $*$-uk egymást követő és $\phi \prec \pi$, akkor $*_\phi$ megelőzi $*_\pi$-t.

Azt várjuk, hogy a sorbarendező pontok teljes rendezése ellentmondásmentes legyen $\prec$-el, azaz Π minden π és ϕ műveletére, ha $\phi \prec \pi$, akkor $*_\phi$ megelőzi $*_\pi$-t. Ennek belátása érdekében tegyük fel, hogy $\phi \prec \pi$. A konstrukció szerint $*_\phi$ a ϕ legutolsó kérése, és az összes olyan művelet után áll, amelyek $\prec$ szerint megelőzik ϕ-t. Továbbá $*_\pi$ a π utolsó kérése és az összes olyan művelet után áll, amelyek $\prec$ szerint megelőzik π-t. Mivel $\phi \prec \pi$, ezért minden művelet, amely $\prec$ szerint megelőzi ϕ-t, megelőzi π-t is. Mivel a sor megszakadna, ha $*_\pi$ előbb állna a sorrendben, mint $*_\phi$, amiből következik, hogy $*_\phi$ megelőzi $*_\pi$-t, amit elvártunk.

Ezután, azt várjuk, hogy ezek a sorbarendező pontok a megkívánt intervallumokba esnek. Tekintsünk egy tetszőleges $\pi \in \Pi$ műveletet. A konstrukció szerint a $*_\pi$ sorbarendező pont π kérése után áll. Megmutatjuk, hogy $*_\pi$ a π válasza előtt fordul elő. Az ellentmondás végett tegyük fel, hogy a π válasza után áll. A konstrukció miatt ez azt jelenti, hogy a π válasza megelőzi (β-ban) valamely ϕ művelet válaszát, amelyre $\phi \prec \pi$ igaz. De ez ellentmond az 1. feltételnek.

Az maradt hátra, hogy megmutassuk, hogy a műveleti intervallumoknak a sorbarendező pontjaira zsugorítása az olvasható/írható változótípus történetét adja. Ez azt jelenti, hogy minden π `OLVAS` művelet annak az `ÍR` műveletnek az értékével tér vissza, amely a $*_\pi$ sorbarendező pont előtti utolsó művelet (vagy v_0, ha nincs ilyen `ÍR` művelet).

De a 3. feltétel azt mondja, hogy $\prec$ rendezi a Π minden `ÍR` műveletét Π összes műveletéhez képest. Adjuk ehhez a 4. feltételt az `OLVAS` műveletekre, amely szerint minden `OLVAS` művelet által visszaadott érték az az érték, amelyet a $\prec$ szerinti utolsó `ÍR` művelet írt (vagy v_0, ha nincs ilyen `ÍR` művelet). Mivel a sorbarendező pontok teljes rendezése ellentmondásmentes a $\prec$ rendezéssel, azt kapjuk, hogy π a kívánt értékkel tér vissza. □

Az alfejezet hátralévő részében a 13.16. lemma felhasználásával megmutatjuk néhány objektumra, hogy teljesíti az atomiságot.

13.4.3.. Egy korlátlan változókat használó algoritmus

Az első algoritmusunk a VitanyiAwerbuch algoritmus, amely m-író/p-olvasó olvasó/író atomi objektumot valósít meg egyedi-író/egyedi-olvasó regiszterekkel. (Emlékeztetünk, hogy $n = m + p$.) Ez az algoritmus egyszerű, de hátránya, hogy korlátlan méretű közös változókat használ.

		az *ÍR* folyamatok írnak			az *OLVAS* folyamatok írnak		
		1	$\cdots$	m	$m+1$	$\cdots$	n
az *ÍR* folyamatok olvasnak	1	$x(1,1)$	$\cdots$	$x(1,m)$	$x(1,m+1)$	$\cdots$	$x(1,n)$
	$\vdots$	$\vdots$		$\vdots$	$\vdots$		$\vdots$
	m	$x(m,1)$	$\cdots$	$x(m,m)$	$x(m,m+1)$	$\cdots$	$x(m,n)$
az *OLVAS* folyamatok olvasnak	$m+1$	$x(m+1,1)$	$\cdots$	$x(m+1,m)$	$x(m+1,m+1)$	$\cdots$	$x(m+1,n)$
	$\vdots$	$\vdots$		$\vdots$	$\vdots$		$\vdots$
	n	$x(n,1)$	$\cdots$	$x(n,m)$	$x(n,m+1)$	$\cdots$	$x(n,n)$

13.9.. ábra. A VitanyiAwerbuch algoritmusban használt X mátrix.

VitanyiAwerbuch algoritmus (vázlatosan)

Az algoritmus n^2 közös változót használ, amelyet egy $n \times n$ mátrix elrendezésben képzelhetünk el, amit a 13.9. ábra mutat. A változók nevei $x(i,j) : 1 \leq i, j \leq n$. Minden $x(i,j)$ változót csak a P_i folyamat olvashatja és csak a P_j folyamat írhatja, azaz minden P_i folyamat olvashatja X i-edik sorának minden változóját és írhatja a i oszlopának minden változóját.

Minden $x(i,j)$ változó az alábbi mezőket tartalmazza:
érték $\in V$, kezdetben v_0
$rag \in \mathbb{N}$, kezdetben 0
$index \in \{1, \cdots, n\}$, kezdetben 1

A *ragpár* rövidítést használjuk a $(rag, index)$ pár jelölésére. A párokat lexikografikusan rendezzük.

Amikor $\texttt{ÍR}(v)_i$ bemenet jelentkezik, akkora a P_i folyamat a következőképpen viselkedik. Először kiolvassa az összes $x(i,j), 1 \leq j \leq n$ változót (tetszőleges sorrendben). Legyen k a rag mezők legnagyobb értéke. Ezután a P_i folyamat végrehajt egy `ír` műveletet minden $x(j,i)$ változóra, az alábbi értékadásokat végezve:

1. $x(j,i).érték := v$
2. $x(j,i).rag := k+1$
3. $x(j,i).index := i$

Végül $\texttt{NYUGTÁZ}_i$ kimenetet ad.
Amikor $\texttt{OLVAS}_i$ bemenet jelentkezik, akkor a P_i folyamat a következőképpen viselkedik. Először kiolvassa az összes $x(i,j), 1 \leq j \leq n$ változót (tetszőleges sorrendben). Legyen (v,k,j) egy olyan $(érték, rag, index)$ hármas, amelyre a $ragpár = (rag, index)$ maximális. Ezután a P_i folyamat végrehajt egy `ír` műveletet minden $x(j,i)$ változóra, az alábbi értékadásokat végezve:

1. $x(j,i).érték := v$
2. $x(j,i).rag := k$
3. $x(j,i).index := j$

(Vagyis elterjeszti az olvasott legjobb információt az összes változónak, amelyet írhat.) Végül a P_i folyamat a v_i kimenetet adja (azaz kiadja a v értéket az i kapun).

13.17. tétel. . *A* VitanyiAwerbuch *algoritmus olvasható/írható atomi objektum, amely várakozásmentes megállást biztosít.*

A VitanyiAwerbuch algoritmus helyességének bizonyítását végezhetnénk úgy, mint a fényképez algoritmusét tettük, azaz ténylegesen elhelyezve sorbarendező pontotat és aztán a megmutatnánk, hogy teljesül az atomi tulajdonság. A VitanyiAwerbuch algoritmus esetében azonban nem könnyű belátni (ellentétben az előző algoritmussal), hogy a sorbarendező pontokat pontosan hová kell helyezni. Itt természetesebb az a bizonyítási stratégia, hogy megállapítunk a *ragpár* értékek alapján a műveleteken részbenrendezést, aztán megmutatjuk, hogy ez a rendezés kielégíti a 13.16. lemma feltételeit.
Bizonyítás. A jólformáltságot és az várakozásmentes megállást könnyű belátni. Az atomiságot a 13.16. lemma segítségével bizonyítjuk.

Legyen α a VitanyiAwerbuch algoritmus egy teszőleges végrehajtása. A 13.10. lemma értelmében az általánosság megszorítása nélkül feltehetjük, hogy α nem tartalmaz befejezetlen műveletet. Egy egyszerű elvárással kezdjük.

13.18. segédtétel. . *Minden $x(i,j)$ változóra a $ragpár = (rag, index)$ értékek monoton nemcsökkenő sorozatot alkotnak α-ban.*

Bizonyítás. Rögzítsük i és j értékét. Jegyezzük meg, hogy $x(i,j)$ csak a P_j folyamat által írható és a jólformáltság miatt j minden művelete sorrendben fordul elő. Továbbá, j bármely teljes műveletsora után a j-edik oszlop minden változójában a *ragpár* értéke ugyanaz.

Valahányszor j végrehajt egy műveletet, mindig azzal kezdi, hogy kiolvassa a j-edik sor minden változóját, beleértve az „átlóban" lévő $x(j,j)$ értékeket. Ezután az általa írt *ragpár*-t úgy választja meg, hogy az legalább akkora legyen, mint amit $x(j,j)$-ben talált. De ez ugyanaz, mint az $x(i,j)$ *ragpár* művelet előtti értéke. Tehát a *ragpár* értéke $x(i,j)$-ben a művelet után legalább akkora, mint amekkora a művelet előtt volt. Ez elég a segédtétel bizonyításához. □

Ezután legyen Π az α-ban előforduló összes művelet halmaza. Minden $\pi \in \Pi$ (`Ír` vagy `OLVAS`) műveletre legyen *ragpár*(π) az az egyértelműen meghatározott *ragpár* érték, amelyet a művelet ír.

13.19. segédtétel. . *Különböző* `Ír` *műveletekre a ragpár*(π) *értékek különbözők* α*-ban.*

Bizonyítás. A különböző kapukon jelentkező `ÍR` műveletekre ez biztosan igaz, mert az *index* értékek különbözőek.

Tehát tekintsünk különböző műveleteket ugyanazon kapura. A jólformáltság miatt ezek a műveletek sorrendben egymás után jelentkeznek. Legyen π és ϕ két `ÍR` művelet az i kapun, és tegyük fel, hogy π megelőzi ϕ-t. Ekkor π befejezi az i-edik oszlop minden változójába való írást, mielőtt ϕ elkezdené kiolvasni az i-edik sorból. Speciálisan, ϕ olyan *ragpár* értéket lát az „átlóban" lévő $x(i,i)$ változóban, amit a π vagy egy későbbi művelet írt be. A 13.18. segédtétel szerint ez a *ragpár* érték legalább akkora, mint a π által írt. Ezért ϕ nagyobb, tehát különböző *ragpár* értéket választ. □

Most definiálunk egy részbenrendezést a Π művelethalmazon. Nevezetesen, azt mondjuk, hogy $\pi \prec \phi$, akkor és csak akkor, ha az alábbi két feltétel valamelyike teljesül.

1. *ragpár*$(\pi) <$ *ragpár*(ϕ).
2. *ragpár*$(\pi) =$ *ragpár*(ϕ), π `ÍR` és ϕ `OLVAS` művelet.

Elegendő megmutatni, hogy az így definiált rendezés kielégíti a 13.16. lemma négy feltételét (ahol $\beta = \mathit{történet}(\alpha) = \alpha|\mathit{végrehajt}(A \times U)$).

1. Minden $\pi \in \Pi$ műveletre csak véges sok olyan ϕ művelet van, amelyre $\phi \prec \pi$. Az ellentmondás eléréséhez tegyük fel, hogy π-t végtelen sok ϕ művelet megelőzi a $\prec$ szerint. A 13.19. segédtétel szerint nem lehet végtelen sok megelőző művelet, amely `ÍR` művelet, tehát végtelen sok megelőző `OLVAS` művelet van. Az általánosság megszorítása nélkül feltételezhetjük, hogy π `ÍR` művelet.

   Ekkor végtelen sok `OLVAS` művelet van ugyanazon t *ragpár* értékkel, amely t kisebb, mint *ragpár*(π). Azon tény miatt, hogy π befejeződik α-ban, következik, hogy a *ragpár*(π) érték ténylegesen beíródik minden sor valamely változójába. Miután ez megtörtént, a 18.18. segédtétel értelmében következik, hogy minden ezt követő `OLVAS` művelet látja és olyan *ragpár* értéket

olvas ki, amely $\geq$ *ragpár*$(\pi) > t$. Ez ellentmond annak, hogy végtelen sok OLVAS művelet van ugyanazon t *ragpár* értékkel.

2. Ha a π válasz eseménye megelőzi a ϕ kérés eseményét β-ban, akkor $\phi \prec \pi$ nem lehetséges. Tegyük fel, hogy a π válasza megelőzi ϕ kérését. Amikor π befejeződik, akkor az ő *ragpár* értéke beíródik az oszlopának minden változójába. Tehát a 13.18. segédtétel miatt, amikor ϕ olvassa a sorának változóit, olyan *ragpár* értékeket olvas, amelyek legalább akkorák, mint *ragpár*(π) értéke. Tehát *ragpár*(ϕ) legalább akkorára választódik, mint *ragpár*(π). Továbbá, ha ϕ ÍR művelet, akkor *ragpár*(ϕ) szigorúan nagyobbra választódik, mint *ragpár*(π).

   Mivel *ragpár*$(\pi) \leq$ *ragpár*(ϕ), ezért csak akkor lehet $\phi \prec \pi$, ha *ragpár*$(\pi) =$ *ragpár*(ϕ), π OLVAS és ϕ ÍR művelet. Ez azonban lehetetlen, mert ha ϕ ÍR művelet akkor, amint azt az előbb megjegyeztük, *ragpár*$(\phi) >$ *ragpár*(π). Tehát $\phi \prec \pi$ nem teljesülhet.

3. Ha π ÍR művelet Π-ben és $\phi \in \Pi$ tetszőleges művelet, akkor vagy $\pi \prec \phi$ vagy $\phi \prec \pi$.

   A 13.19. segédtétel szerint minden ÍR művelet különböző *ragpár* értéket kap. Ebből következik, hogy az ÍR műveletek teljesen rendezettek, és hogy az OLVAS műveletek is rendezettek az összes ÍR művelethez képest.

4. Minden OLVAS művelet által visszaadott érték az az érték, amelyet a $\prec$ szerinti utolsó ÍR művelet írt (vagy v_0, ha nincs ilyen megelőző ÍR.)

   Legyen π OLVAS művelet. A π által visszaadott v az az érték, amely a legnagyobb t *ragpár* értékhez tartozik a saját sorában, és ez a t szintén a π *ragpár*-jává válik. Két eset lehetséges:

   (a) A v értéket egy olyan ϕ ÍR művelet írta, melynek *ragpár*-ja t.
   Ebben az esetben a rendezés definíciója biztosítja, hogy ϕ a $\prec$ szerinti utolsó ÍR művelet, amely megelőzi π-t, amit akartunk.

   (b) $v = v_0$ és $t = 0$.
   Ebben az esetben a rendezés definíciója biztosítja, hogy nincs olyan ÍR művelet, amely $\prec$ szerint megelőzné π-t, ahogy kell.

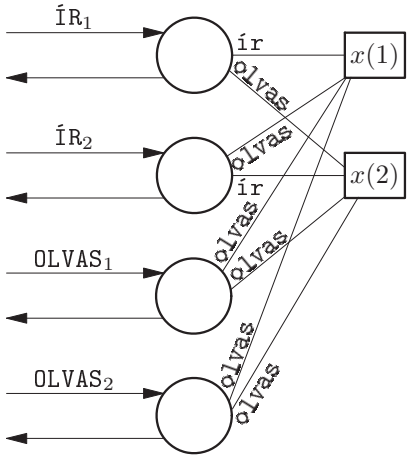
□

Bonyolultságelemzés.. A VitanyiAwerbuch algoritmus n^2 közös változót használ, mindegyik korlátlan méretű, még akkor is, ha a V tartomány véges. Minden OLVAS és ÍR művelet végrehajt $4n$ közös memória hozzáférést, aminek az összesített időbonyolultsága $\mathcal{O}(nl)$.

13.4.4.. Egy korlátos algoritmus két íróra

Mint a NemkorlátosFénykép algoritmusnak, a VitanyiAwerbuch algoritmusnak is az a hátránya, hogy nem korlátos méretű közös változókat használ a nem korlátos *rag* értékek tárolására. Több alternatív algoritmust terveztek, amelyek csak korlátos adatokat használnak, de sajnálatos módon ezek többsége meglehetősen bonyolult (és nem hatékony ahhoz, hogy a gyakorlatban alkalmazható

legyen). Ebben az alfejezetben csak egy nagyon egyszerű algoritmust mutatunk be egy speciális esetre.



13.10.. ábra. A BLOOM algoritmus architektúrája két olvasó esetére.

Nevezetesen, a BLOOM algoritmust, amely 2-író/p-olvasó olvasható/írható atomi objektumot valósít meg az $x(1)$ és $x(2)$ 1-író/$(p+1)$-olvasó regiszterekkel. (Most $n = 2+p$.) Mindegyik $x(i)$ változót írhatja a P_i folyamat ÍR művelete, és olvashatja bármely folyamat. A 13.10. ábra szemlélteti az architektúrát két olvasó speciális esetére. Az algoritmus egyszerű, de nem általánosítható több író esetére.

BLOOM algoritmus (vázlatosan)

Az algoritmus két közös változót használ, $x(1)$-et és $x(2)$-t, ahol $x(i)$-t írhatja a P_i folyamat és olvashatja minden folyamat. A továbbiakban legyen $\bar{i}$ 2, ha $i = 1$ és 1, ha $i = 2$. Az $x(i)$ regiszter az alábbi mezőket tartalmazza:

érték $\in V$, kezdetben v_0
rag $\in \{0, 1\}$, kezdetben 0

Amikor ÍR$(v)_i$ művelet jelentkezik az i kapun, $i \in \{1, 2\}$, akkor a P_i folyamat a következőképpen viselkedik. Először kiolvassa $x(\bar{i})$-t, legyen b az itt talált *rag* értéke. Ezután $x(i)$-be ír a következő értékadásokat végrehajtva:

1. $x(i).$*érték* $:= v$,
2. $x(i).$*rag* $:= b + i \ (\mod 2)$.

Végül NYUGTÁZ$_i$ kimenetet ad.

Tehát amikor a P_i ÍR folyamat ÍR műveletet hajt végre, akkor nem csak beírja az új értéket a változójába, hanem el akarja érni, hogy a két változóban lévő *rag* értékek összege egyenlő legyen saját indexe moduló 2.

Tehát a P_1 folyamat mindig arra törekszik, hogy a *rag* értékek különbözők, a P_2 folyamat pedig, hogy egyenlők legyenek. Amikor $\texttt{OLVAS}_i$ jelentkezik az i kapun, $3 \leq i \leq n$, akkor a P_i folyamat az alábbiak szerint működik. Először kiolvassa mindkét regisztert, legyen b a két regiszterben talált *rag* értékének összege moduló 2. Ezután az $x(1)$ regisztert olvassa, ha $b = 1$, egyébként pedig az $x(2)$-t, és az ott talált *érték*-kel tér vissza.

Tehát mindegyik `OLVAS` folyamat ugyanúgy viselkedik. Minden `OLVAS` folyamat kiolvassa mindkét regisztert, hogy megállapítsa, hogy azok *rag* értéke egyenlő avagy nem egyenlő. Ha a *rag*-ok egyenlők, akkor $x(1)$-ből, egyébként az $x(2)$-ből vett értékkel tér vissza.

13.20. tétel. . *A* Bloom *algoritmus olvasható/írható atomi objektum és várakozásmentes megállást biztosít.*

Most is elvégezhetnénk a helyesség bizonyítását úgy, hogy ténylegesen beszúrnánk a sorbarendező pontokat és aztán megmutatnánk, hogy az atomi tulajdonság teljesül. Most azonban egy érdekes stratégiát alkalmazunk, amely a 13.16. lemmának és a 8.5.5. szakaszban definiált szimulációs bizonyításnak a kombinációja. Először definiáljuk a Bloom algoritmus egy változatát, az EgészBloom algoritmust, amely egész értékű *rag*-okat használ bitek helyett. A 13.16. lemma felhasználásával megmutatjuk, hogy az EgészBloom algoritmus helyes. Aztán bizonyítjuk, hogy a Bloom algoritmus helyes, a Bloom-ból EgészBloom-ba való szimulációs reláció megadásával.

EgészBloom algoritmus (vázlatosan)

Az algoritmus két közös változót használ, $x(1)$-et és $x(2)$-t, ahol $x(i)$-t írhatja a P_i folyamat és olvashatja minden folyamat. Az $x(i)$ regiszter az alábbi mezőket tartalmazza:

érték $\in V$, kezdetben v_0;
rag $\in \mathbb{N}$, kezdetben 0, ha $i = 1$ és 1, ha $i = 2$.

Amikor $\texttt{ÍR}(v)_i$ művelet jelentkezik az i kapun, $i \in \{1, 2\}$, akkor a P_i folyamat a következőképpen viselkedik. Először kiolvassa $x(\bar{i})$-t, legyen t az itt talált *rag* értéke. Ezután $x(i)$-be ír a következő értékadásokat végrehajtva:

1. $x(i).\mathit{érték} := v$;
2. $x(i).\mathit{rag} := t + 1$.

Végül `NYUGTÁZ` kimenetet ad.

Amikor $\texttt{OLVAS}_i$ jelentkezik az i kapun, $3 \leq i \leq n$, akkor a P_i folyamat az alábbiak szerint működik. Először kiolvassa mindkét regisztert, legyen t_1 és t_2 a két regiszterben talált *rag* értéke. Ekkor két eset lehet: Ha $|t_1 - t_2| \leq 1$, akkor a P_i folyamat azt a regisztert olvassa, amelyekben a nagyobb *rag* érték van, és az ott talált *érték*-kel tér vissza. (A regiszter egyértelműen meghatározott, mert mint azt a 13.21. lemma állítja, $x(1)$-ben a *rag* mindig páros, $x(2)$-ben pedig mindig páratlan.) Egyébként, – vagyis ha $|t_1 - t_2| > 1$

— a P_i folyamat nem meghatározott módon választ egy regisztert és az abban lévő *érték*-kel tér vissza.

A következő lemma az EgészBloom algoritmus egy alapvető tulajdonságát fejezi ki. Könnyű bizonyítani.

13.21. lemma. . *Az* EgészBloom *algoritmus minden elérhető állapotában a következők teljesülnek:*

1. $x(1).rag$ *páros.*
2. $x(2).rag$ *páratlan.*
3. $|x(1).rag - x(2).rag| \leq 1$.

13.22. tétel. . *A* EgészBloom *algoritmus olvasható/írható atomi objektum és várakozásmentes megállást biztosít.*

Bizonyítás. A bizonyítás hasonló a 13.17. tételéhez. A jólformáltságot és a várakozásmentességet könnyű belátni. Az atomisághoz felhasználjuk a 13.16. lemmát. Legyen α az EgészBloom algoritmus egy tetszőleges végrehajtása. Mint korábban is, az általánosság megszorítása nélkül feltesszük, hogy α-ban nincs befejezetlen művelet.

13.23. segédtétel. . *Minden $x(i)$ változóra a rag értékek sorozata α-ban monoton nem csökkenő.*

Jelölje Π az α összes műveletének halmazát. Minden $\pi \in \Pi$ `ÍR` műveletre definiáljuk a $rag(\pi)$ értékét úgy, mint az a *rag* érték, amelyet π az író lépésében ír.

Most definiálunk egy részbenrendezést a műveletek Π halmazán. Először rendezzük a `ÍR` műveleteket a *rag* értékük alapján. Ha két `ÍR` műveletnek ugyanaz a *rag* értéke, akkor azok ugyanazon íróhoz tartoznak, és ekkor a sorrendjük legyen az előfordulásuk sorrendje. Ezután rendezzük Π összes `OLVAS` műveletét úgy, hogy az közvetlenül az után az `ÍR` művelet után következzen, amelyiknek az *érték*-ét olvassa (vagy az összes `ÍR` után, ha nincs ilyen `ÍR`).

Elegendő igazolni, hogy teljesül a 13.16. lemma négy feltétele (ahol $\beta = történet(\alpha) = \alpha|végrehajt(A \times U)$). A 3. és 4. feltétel közvetlenül teljesül, így csak az 1. és 2. feltételt kell igazolnunk. Ehhez a következő állítás hasznos lesz.

13.24. segédtétel. . *Ha a π `ÍR` művelet ír lépése megelőzi a ϕ `ÍR` művelet kérését, akkor $\pi \prec \phi$.*

Bizonyítás. Ha π és ϕ ugyanazon kapuhoz tartozik, akkor a 13.23. segédtételből következik, hogy $rag(\pi) \leq rag(\phi)$, és a $\prec$ definíciója szerint $\pi \prec \phi$. Másrészről, ha π és ϕ különböző kapukhoz tartozik, akkor ϕ olvassa vagy π, vagy a π kapujának egy későbbi `ÍR` műveletének az eredményét. A 13.23. segédtétel szerint a ϕ által olvasott *rag* nagyobb vagy egyenlő, mint $rag(\pi)$. Tehát $rag(\pi) < rag(\phi)$, tehát $\pi \prec \phi$ ismét teljesül. □

13.25. segédtétel. . *Ha a π* `ÍR` *művelet* `ír` *lépése megelőzi a ϕ* `OLVAS` *művelet kérését, akkor $\pi \prec \phi$.*

Bizonyítás. Azt kell megmutatnunk, hogy ϕ a π eredményével tér vissza, vagy más olyan ψ `ÍR` eredményével, amelyre $\pi \prec \psi$. Legyen $t = rag(\pi)$ és tegyük fel, hogy π az i kapun jelentkezik.

Amikor ϕ hívódik, a 13.23. segédtétel szerint $x(i).rag \geq t$, és a 13.21. lemmából következik, hogy $|x(1).rag - x(2).rag| \leq 1$. Tehát amikor ϕ hívódott, akkor $x(\bar{i}).rag \geq t - 1$. A $\prec$ definíciója és a 13.23. segédtétel miatt egyedül akkor van gond, ha ϕ olyan `ÍR` eredményével tér vissza, amelyre $rag = t - 1$. Tehát tegyük fel ezt az esetet.

Ekkor ϕ-nek látnia kell a $x(\bar{i}).rag = t-1$ értéket, vagy az első, vagy a második olvasáskor, és a harmadik olvasáskor is. Ha ϕ az $x(i).rag = t$ értéket látja akár az első, akár a második olvasáskor, akkor a t és $t-1$ ragok kombinációja miatt ϕ az $x(i)$, és nem az $x(\bar{i})$ regisztert választja, ami ellentmondás. Tehát ϕ szükségképpen látja a $x(i).rag > t$ esetet. A 13.21. lemmából következik azonban, hogy amikor ϕ látja a $x(i).rag > t$ esetet, akkor az is teljesül, hogy $x(\bar{i}).rag > t - 1$. Ez azt jelenti, hogy ϕ nem láthatja az $x(\bar{i}).rag = t-1$ esetet a harmadik olvasáskor, ami szintén ellentmondás. □

A 13.24. és 13.25. segédtételeket felhasználva az 1. feltétel könnyen megmutatható, amit gyakorlatra hagyunk.

A 2. feltétel belátásához tegyük fel, hogy π válasz eseménye megelőzi a ϕ kérés eseményét β-ban. Ha π `ÍR` művelet, akkor a 13.24. és 13.25. segédtételekből következik, hogy $\pi \prec \phi$. Tegyük fel, hogy π `OLVAS` művelet. Az ellenmondás végett tegyük fel, hogy $\phi \prec \pi$.

Ha ϕ `ÍR` művelet, akkor π nyilvánvalóan nem térhet vissza ϕ eredményével, mivel ϕ nem hajtja végre írás lépését amíg π be nem fejeződött. Tehát az egyetlen problémás eset akkor van, ha π olyan ψ `ÍR` művelet eredményével tér vissza, amelyre $\phi \prec \psi$. De ekkor a ψ írás lépése megelőzi π befejeződését, tehát megelőzi ϕ kérését. Ekkor a 13.24. segédtétel miatt $\psi \prec \phi$, ami ellentmondás.

Másrészről, ha ϕ `OLVAS` művelet, akkor $\phi \prec \pi$ feltételből következik, hogy léteznie kell olyan ψ `ÍR` műveletnek, hogy $\phi \prec \psi$ és π a ψ eredményét kapja. Mivel π a ψ eredményét kapja, ezért a ψ írás lépése megelőzi π befejeződését, így megelőzi ϕ kérését. De ekkor a 13.25. segédtételből következik, hogy $\psi \prec \phi$, ami ellentmondás. □

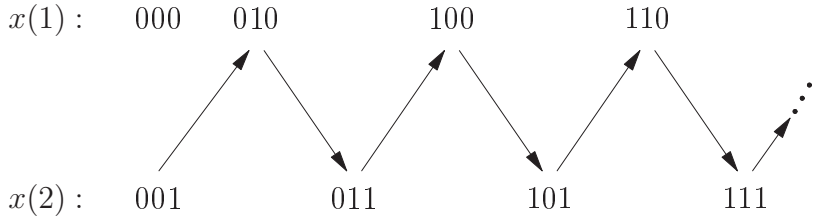
Most megmutatjuk a Bloom és az EgészBloom algoritmusok közötti kapcsolatot szimulációs relációt használva. Az általános stratégiát a 8.5.5. részben adtuk meg és használtuk a 8.5.6. példa bizonyításában, továbbá a 10.9.4. szakaszban.

Kiderül, hogy a két algoritmus közötti kapcsolat (eléggé furcsa módon) az, hogy a Bloom algoritmus $\{0,1\}$-értékű ragjai megegyeznek az EgészBloom algoritmus egész értékű ragjainak a második legalacsonyabb helyértékű bitjeivel.

13.4.1. példa. Bitek avagy egészek a Bloom algoritmusban

Tekintsünk egy olyan végrehajtását az EgészBloom algoritmusnak, amelyben az `ÍR` műveletek váltakozva jelentkeznek az 1. és a 2. ka-

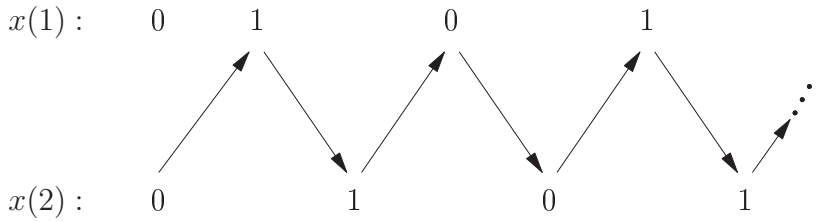
pun, az 1. kapun kezdve, és minden ÍR azután kezdődik, hogy az előző befejeződött. Ekkor minden ÍR folyamatosan nagy *rag* értékeket produkál. A regiszterek *rag* értékének bináris alakját mutatja a 13.11. ábra. Kezdetben $x(1)$ és (2) *rag* értéke 0, illetve 1. Az első ÍR_1 végrehajtja az $x(1).rag := 2$ értékadást, az ÍR_2 a $x(2).rag := 3$ értékadást, és így tovább.



13.11.. ábra. A két regiszterben egymást követő *rag* értékek az EgészBloom algoritmusban.

A Bloom megfelelő végrehajtásában szereplő *rag* értékeket a 13.12. ábra mutatja. Kezdetben mindkét regiszterben a $rag = 0$. Minden ÍR_1 művelet $x(1).rag$ értékét úgy állítja be, hogy ellentétes legyen $x(2).rag$ értékével, az ÍR_1 művelet pedig úgy állítja be $x(2).rag$ értékét, hogy megegyezzen $x(1).rag$-al.

Vegyük észre, hogy minden esetben *rag* értéke a Bloom végrehajtásában pontosan a második legalacsonyabb helyértékű bitje az EgészBloom végrehajtásában szereplő *rag*-nak.



13.12.. ábra. A két regiszterben egymást követő *rag* értékek a Bloom algoritmusban.

Kiderül, hogy a 13.4.1. példában bemutatott kapcsolat a két algoritmus minden végrehajtására teljesül. Ha s és u állapotai a Bloom, illetve az EgészBloom rendszernek (algoritmusok és felhasználók), akkor definiáljuk az $(s, u) \in f$ relációt (vagy $u \in f(s)$), feltéve, hogy az állapot komponensek megegyeznek, kivéve ha az u egész értékű ragja t, akkor s-ben a *rag* értéke megegyezik t második legalacsonyabb helyértékű bitjével.

13.26. lemma. . *f szimulációs reláció a* Bloom *és* EgészBloom *rendszerek között.*

Bizonyításvázlat. Mivel a két algoritmus egyértelmű kezdő állapotai az f relációban vannak, a szimuláció kezdő feltétele nyilvánvalóan teljesül. A lépés feltétel megmutatása az igazán érdekes. Elég megmutatni, hogy a Bloom minden (s, π, s') lépésére, és minden $u \in f(s)$-re, ahol s és u elérhető állapotok, létezik olyan (u, ϕ, u') lépése az EgészBloom algoritmusnak, hogy $u' \in f(s')$ és ϕ „majdnem" megegyezik π-vel. Különösen, ϕ és π ugyanaz, kivéve, hogy ϕ egész értéket, π pedig annak második legalacsonyabb helyértékű bitjét használja. A π szerinti eseteket tekintjük. Ha π kérés vagy válasz esemény, akkor az érvelés magától értetődő. Az az érdekes, amikor a lépés az `ÍR` művelet ír lépése, vagy `OLVAS` műveletben a harmadik olvasás lépés.

Tegyük fel, hogy (s, π, s') a Bloom algoritmus olyan lépése, amikor az 1 folyamat az $\texttt{ÍR}_1$ művelet részeként az $x(1)$ változóba ír. Ekkor a P_1 folyamat az $x(1).rag$ értékét úgy állítja be, hogy az ne legyen egyenlő azzal a b értékkel, amelyet $x(2).rag$-ból olvasott ki. Tehát $s'.x(1).rag \neq b$. Az EgészBloom algoritmusban a megfeleltetett u állapotban a P_i folyamat at $x(2).rag$-ból olvassa ki a t egész értéket. Mivel $u \in f(s)$, ezért b szükségképpen a második legalacsonyabb helyértékű bitje t-nek. Legyen u' az EgészBloom algoritmusban keletkező egyértelműen meghatározott új állapot. Ekkor $u'.x(1).rag = t + 1$. Az $u' \in f(s')$ tartalmazáshoz meg kell mutatnunk, hogy $u'.x(1).rag$ második legalacsonyabb helyértékű bitje megegyezik $s'.x(1).rag$-al, azaz $t+1$ második legalacsonyabb helyértékű bitje nem egyenlő b-vel. Ez azonban következik abból a tényből, hogy t páratlan (a 13.21. lemma miatt) és hogy b a második legalacsonyabb helyértékű bitje t-nek.

Az érvelés hasonló arra az esetre, amikor az (s, π, s') lépésben a P_2 folyamat az $x(2)$ változóba ír.

Most tegyük fel, hogy (s, π, s') a Bloom algoritmus azon lépése, amikor a P_i folyamat harmadszor olvas az `OLVAS` művelet részeként. A kulcs az, hogy az EgészBloom algoritmus megengedi, hogy a P_i folyamat akár az $x(1)$, akár az $x(2)$ regisztert olvassa. Tegyük fel, hogy a Bloom algoritmus az s állapotban a b_1 és b_2 ragokat olvasta ki az $x(1)$, illetve az $x(2)$ változókból. Hasonlóan, tegyük fel, hogy az EgészBloom algoritmus az u állapotban a t_1 és t_2 ragokat olvasta ki az $x(1)$, illetve az $x(2)$ változókból. Mivel $u \in f(s)$, tudjuk, hogy b_1 második legalacsonyabb helyértékű bitje t_1-nek, és b_2 második legalacsonyabb helyértékű bitje t_2-nek. Három eset lehetséges:

1. $t_1 = t_2 + 1$.
   Ekkor a 13.21. lemmából következik, hogy t_1 és t_2 második legalacsonyabb helyértékű bitjei különbözőek. Ekkor mind a Bloom, mind az EgészBloom algoritmus az $x(1)$ regiszterből olvas.
2. $t_2 = t_1 + 1$.
   Ekkor a 13.21. lemmából következik, hogy t_1 és t_2 második legalacsonyabb helyértékű bitjei megegyeznek. Ekkor mind a Bloom, mind az EgészBloom algoritmus az $x(2)$ regiszterből olvas.
3. $t_1 \neq t_2 + 1$ és $t_2 \neq t_1 + 1$.

Ekkor a 13.21. lemmából következik, hogy $|t_1 - t_2| > 1$. Ekkor az EGÉSZBLOOM algoritmus bármelyik regiszterből olvashat.

□

Most már bebizonyíthatjuk a 13.20. tételt, amely igazolja a BLOOM algoritmus helyességét.

13.20. tétel) bizonyítása. A 13.26. lemmából és a 8.12. tételből következik, hogy a BLOOM rendszer minden története az EGÉSZBLOOM rendszernek is története. (Emlékeztetünk, hogy itt a történetek tartalmaznak `megállít` eseményeket is a kérések és válaszok mellett.) A 13.22. tételből következik, hogy a jólformáltság és az atomiság feltételek teljesülnek az EGÉSZBLOOM rendszerre. Mivel a jólformáltság és az atomiság feltételek történet tulajdonságként vannak megfogalmazva, ezért ezek átvivődnek a BLOOM algoritmusra is.

A várakozásmentességet könnyű belátni. □

Bonyolultságelemzés.. A BLOOM algoritmus két közös változót használ, mindegyik $2|V|$ különböző értéket vehet fel. Minden művelet konstans számú közös memória elérést igényel, azaz $\mathcal{O}(l)$ időt.

13.4.5.. Egy fénykép objektumot használó algoritmus

Az utolsó szakaszban a várakozásmentes m-író/p-olvasó olvasó/író atomi objektumnak fénykép közös változókkal való megvalósítását, a FÉNYKÉPREGISZTER algoritmust mutatjuk be. (Ismét legyen $n = m+p$.) A FÉNYKÉPREGISZTER és az atomi objektumok 13.3.-beli megvalósításának kombinációja, felhasználva a 13.9. következményt, a várakozásmentes m-író/p-olvasó olvasó/író atomi objektumok 1-író/n-olvasó közös regiszterrel történő megvalósítását adja.

A FÉNYKÉPREGISZTER algoritmusban a fénykép közös változók korlátlan méretűek, még akkor is, ha a megvalósítandó olvasó/író atomi objektum V tartománya véges. Lehetséges módosítani a FÉNYKÉPREGISZTER algoritmust, ámbár meglehetősen bonyolult, úgy, hogy csak korlátos közös fényképez változókat használjon.

FÉNYKÉPREGISZTER algoritmus (vázlatosan)

Az algoritmus egyetlen x közös változót használ, amely m komponensű vektorokra alapozott fénykép objektum. Az x minden komponensének W tartománya $(érték, rag)$ párok halmaza, ahol $érték \in V$ és $rag \in \mathbb{N}$; a w_0 kezdőérték $(v_0, 0)$.

Minden $P_i, 1 \leq i \leq m$ `ÍR` folyamat `módosít`(i, w) és `fényképez` műveletet hajt végre az x változón, míg minden $P_i, m+1 \leq i \leq n$ `OLVAS` folyamat csak `fényképez` műveletet hajt végre az x változón.

Amikor `OLVAS`$_i$ bemenet jelentkezik az $i, m+1 \leq i \leq n$ kapun , akkor a P_i folyamat a következőképpen viselkedik. Először `fényképez` műveletet hajt végre x-en, amely az u vektort eredményezi. Legyen j az az index $(1 \leq j \leq m)$, amelyre a párok lexikografikus rendezésében az $(u(j).rag, j)$ pár a legnagyobb. Az i folyamat az $u(j).érték$ értékkel tér vissza.

Amikor ÍR$(v)_i$ bemenet jelentkezik az $i, 1 \leq i \leq m$ kapun, akkor a P_i folyamat a következőképpen viselkedik. Először `fényképez` műveletet hajt végre, amely az u vektort eredményezi. Mint az előbb, legyen j az az index $(1 \leq j \leq m)$, amelyre a párok lexikografikus rendezésében az $(u(j).rag, j)$ pár a legnagyobb. A P_i folyamat `módosít`$(i, (v, u(j).rag + 1))$ műveletet hajt végre. Végül a P_i folyamat NYUGTÁZ$_i$ kimenetet ad.

A FÉNYKÉPREGISZTER algoritmus kissé hasonlít a VITANYIAWERBUCH algoritmushoz, de egyszerűbb a fénykép közös memória nagyobb hatása miatt.

13.27. tétel. . *A* FÉNYKÉPREGISZTER *algoritmus olvasható/írható atomi objektum és várakozásmentes megállást biztosít.*

Bizonyításvázlat. A bizonyítás hasonló a VITANYIAWERBUCH és az EGÉSZ-BLOOM algoritmusok bizonyításához, felhasználva a 13.16. lemmát. A bizonyítás gyakorlatként elvégezhető. □

Bonyolultságelemzés.. A FÉNYKÉPREGISZTER algoritmus egy fényképez közös változót használ, amely korlátlan méretű, még akkor is, ha a V tartomány véges. Minden művelet konstans számú közös memória elérést igényel, azaz összesen $\mathcal{O}(l)$ időt.

Hierarchikus építés.. A 13.27. tétel és a fénykép atomi objektum minden várakozásmentes megvalósítása együttesen várakozásmentes megvalósítását adják m-író/p-olvasó olvasható/írható atomi objektumoknak 1-író/$m + p$-olvasó közös regiszterekkel. A bizonyítás a 13.9. következményen alapszik. (Technikailag, hogy alkalmazni tudjuk a 13.9. következményt, olyan fénykép atomi objektum kell, amelynek csak $n = m + p$ kapuja van, folyamatonként egy – például, az i ÍR folyamat mindkét, `módosít` és `fényképez` műveletét végrehajthatja ugyanazon kapun. Nem okoz gondot a fényképez atomi objektum külső felületének és megvalósításának ehhez igazodó módosítása.)

Általánosítások.. Számos érdekes általánosítása van a FÉNYKÉPREGISZTER algoritmusnak, amelyek szintén helyesen működnek. Először, a ÍR$_i$ során, ha $i = j$ – vagyis ha a P_i folyamat maga is a legnagyobb *ragpárt* tartalmazza –, akkor i használhatja a *rag* előzőleg kapott értékét is. Másodszor, lehetséges nem negatív valós *rag* értékeket alkalmazni egész értékek helyett. Ekkor az i író választhat minden olyan valós számot *rag*-ként, amely szigorúan nagyobb, mint amit látott. Mégegyszer, ha i maga is a legnagyobb *rag* értéket tartalmazza, akkor újra választhatja az előző *rag*-ot. Mindkét általánosítás hasznos lehet olvasható/írható atomi objektumok megvalósítása helyességének bizonyításakor, amikor fénykép objektumokat használunk.

13.5.. Megjegyzések a fejezethez

Az „atomi objektum” ötlete Lamport [181, 182] olvasható/írható atomi objektumokkal foglakozó munkájából származtatható. Herlihy és Wing [153] kiterjesz-

tette az atomiság fogalmát tetszőleges változótípusra és *linearizálhatóságnak* nevezte. A Kőnig lemmát eredetileg Kőnig [170] bizonyította, egy bizonyítás megtalálható Knuth [169] könyvében. A kanonikus várakozásmentes atomi objektum automata Merritt [3] munkájából származik. Az atomi objektumok és a közös változók közötti kapcsolat Lamport és Schneider [186] valamint Goldman és Yelick [139] munkáiból származik. Az olvasható/módosítható/írható atomi objektumoknak olvasó/író objektumokkal történő megvalósításának lehetetlenségét Herlihy [150] bizonyította.

A fényképez atomi objektum ötletét Afek, Attiya, Dolev, Gafni, Merritt és Shavit [3], valamint Anderson [11, 12] vezette be, amit Chandy és Lamport [68] osztott hálózatok ellentmondásmentes globális fényképei sugalltak. A fénykép atomi objektumok itteni megvalósításai, mind a NemkorlátosFénykép, mind a KorlátosFénykép, Afek [3]és a többiektől származik. A *KorlátosFénykép* protokollban használt kézfogás stratégiát Petterson [240] alkotta meg. Egy friss atomi objektum algoritmust, amely csak $\mathcal{O}(nl \log n)$ idejű és nem négyzetes, Attiya és Rachman [26] fejlesztett ki.

Számos algoritmust terveztek olvasható/írható atomi objektumoknak egyszerűbb típusú olvasó/író regiszterekkel történő megvalósítására. A VitanyiAwerbuch algoritmus Vitanyi és Awerbuch [283] cikkében jelent meg. A cikk tartalmaz korlátos közös változókat alkalmazó algoritmust is, de az hibás. A Bloom algoritmust Bloom [53] alkotta, és a FényképRegiszter algoritmus Gawlick, Lynch és Shavit [135] munkájából származik. Kizárólagosan-írható/megosztottan-olvasható atomi objektumoknak kizárólagosan-írható/kizárólagosan-olvasható regisztereket használó, korlátos algoritmusos megvalósítását Singh, Anderson és Gouda [263], valamint Haldar és Vidyasankar [144] tervezte. megosztottan-írható/megosztottan-olvasható atomi objektumoknak kizárólagosan-írható/megosztottan-olvasható regisztereket használó, korlátos algoritmusos megvalósítását adta Peterson és Burns [241], Schaffer [254], Israeli és Li [162], Li, Tromp és Vitanyi [196] valamint Dolev és Shavit [100]. A Shaffer-algoritmus [254] kijavította Peterson és Burns algoritmusának hibáit. Gawlick, Lynch és Shavit [135] adta a megvalósítását megosztottan-írható/megosztottan-olvasható atomi objektumoknak korlátos fénykép változók felhasználásával és az általános FényképRegiszter algoritmushoz viszonyított szimulációs bizonyítását adták az algoritmus helyességének. Számos konstrukció használja az „korlátos időbélyegzés" fogalmát. Korlátos időpecsételéses algoritmust tervezett Israeli és Li [162], Dolev és Shavit [100], Gawlick, Lynch és Shavit [135], Israeli és Pinchasov [163], Dwork és Waarts [107], valamint Dwork, Herlihy, Plotkin és Waarts [102].

Attiya és Welch összehasonlította olvasható/írható atomi objektumok megvalósításának a költségét azzal az esettel, amikor kissé gyengébb ellentmondásmentességi feltételt követelünk meg [28]. A munkájukat aszinkron hálózati modellben végezték.

13.6.. Gyakorlatok

13-1. Adjunk külső felületet a 2-író/1-olvasó atomi objektumhoz és adjunk érdekes példákat ezen külső felülethez tartozó olyan sorozatokra, amelyek kielégítik az atomiság feltételt, és olyanokat is, amelyek nem elégítik ki az atomiság feltételt. Biztosan legyenek a példák között véges és végtelen sorozatok is, továbbá nem befejezettek is.

13-2. Tekintsük azt az olvasható/módosítható/írható atomi objektumot, amelynek V tartománya az egész számok halmaza, kezdőértéke 0. (Az olvasható/módosítható/írható változótípus definícióját lásd a 9.4. alfejezetben – emlékeztetünk, hogy az olvasható/módosítható/írható változó visszatérési értéke a változónak a művelet előtti értéke.)
Az objektumnak két kapuja van, az 1. kapu csak `növel` műveletet támogat (amely hozzáad 1-et a változó értékéhez), a 2. kapu csak `csökkent` műveletet támogat (amely csökkenti 1-el a változó értékét). Az alábbi sorozatok közül melyek elégítik ki az atomiság feltételt?

(a) $\texttt{növel}_1, \texttt{csökkent}_2, 0_1, 0_2$

(b) $\texttt{növel}_1, \texttt{csökkent}_2, -1_1, 0_2$

(c) $\texttt{növel}_1, \texttt{csökkent}_2, 0_1, 1_2$

(d) $\texttt{csökkent}_2, \texttt{növel}_1, 0_1, \texttt{növel}_1, 1_1, \texttt{növel}_1, 2_1, \texttt{növel}_1, 3_1, \cdots$

(e) $\texttt{csökkent}_2, \texttt{növel}_1, 0_1, \texttt{növel}_1, 0_1, \texttt{növel}_1, 1_1, \texttt{növel}_1, 2_1, \cdots$

13-3. Egészítsük ki a 13.1. tétel néhány hiányzó részletét. Különösen, az általunk megadottnál részletesebben mutassuk meg, hogy létezik a gyökértől induló tetszőleges hosszú út, és hogy egy végtelen út helyes választást biztosít a teljes β sorozatra.

13-4. Általánosítsuk a változótípus fogalmát úgy, hogy megengedünk véges sok kezdőértéket egy helyett, továbbá megengedünk nemdeterminisztikus választást függvény helyett. Általánosítsuk és bizonyítsuk be a 13.1. tétel megfelelőjét az új definícióra. Mi történik, ha megengedünk végtelen nemdeterminisztikusságot?

13-5. Tegyük fel, hogy úgy módosítjuk a 13.1.4. példát, hogy a rendszer megenged `csökkent` műveletet is `olvas` és `növel` művelet mellett. Az algoritmus ugyanaz, mint korábban volt, a következő hozzáadásával: amikor $\texttt{csökkent}_i$ bemenet jelentkezik az i kapun, akkor a P_i folyamat csökkenti $x(i)$ értékét.

olvasható/növelhető/csökkenthető atomi objektum lesz-e az így keletkező rendszer? Vagy bizonyítsuk be az atomiságot, vagy adjunk olyan végrehajtást, ami ellenpélda.

13-6. Bizonyítsuk be a 13.4. tételt.

13-7. Bizonyítsuk be a 13.5. tételt.

13-8. Bizonyítsuk be a 13.6. tételt.

13-9. Mutassuk meg, hogy a 13.7. tétel nem igaz, ha elhagyjuk az *A követ* függvényére kirótt speciális követelményt.

13-10. Adjuk meg az OMÍMEGVALOÍ algoritmus formális leírását előfeltételhatás jelöléssel. A leírásnak modulárisnak kell lennie abban az értelemben, hogy a kölcsönös kizárás komponenst egy külön automata reprezentálja, kombinálva az OMÍMEGVALOÍ algoritmus fő részével, amely b/k automata összekapcsolást használ. Bizonyítsuk be, hogy az algoritmus helyesen működik (feltételezve a kölcsönös kizárás komponens helyességét).

13-11. Tekintsük a NEMKORLÁTOSFÉNYKÉP algoritmus azon változatát, amelyben a `fényképez` és `belső_fényképez` művelet valamely $x(i)$ változóra négy helyett három különböző *rag* érték előfordulását figyeli. Így is helyes lesz-e a módosított algoritmus? Vagy bizonyítsuk be az állítást, vagy adjunk olyan végrehajtást, ami ellenpélda.

13-12. Tekintsük a NEMKORLÁTOSFÉNYKÉP algoritmus azon változatát, amelyben a P_i folyamat `fényképez` és `módosít` műveletek végrehajtása során is növeli az $x(i).rag$-ot. (Az $x(i).érték$ és $x(i).nézet$ komponensek nem változnak, és a `belső_fényképez` műveletet egyáltalán nem módosítjuk.)

Így is helyes lesz-e a módosított algoritmus? Vagy bizonyítsuk be az állítást, vagy adjunk olyan végrehajtást, ami ellenpélda.

13-13. *Kutatási feladat.* Tudná olyan alternatív bizonyítását adni a NEMKORLÁTOSFÉNYKÉP algoritmus helyességének, amely a megfelelő kanonikus várakozásmentes atomi objektumra vonatkozó formális kapcsolaton alapszik?

13-14. Tervezzük meg a KORLÁTOSFÉNYKÉP algoritmus olyan változatát, amely kiküszöböli a *kapcsoló* biteket. Az algoritmusban a `fényképez` folyamat úgy határozhatná meg két olvasás ellentmondásmentességét, hogy nem csak a kézfogás biteket, hanem az *érték* mezőket is vizsgálná. Bizonyítsuk be, hogy az algoritmus helyes.

13-15. *Kutatási feladat.* Tervezzük meg a várakozásmentes fénykép atomi objektum olyan megvalósítását, amely hatékonyabb a KORLÁTOSFÉNYKÉP algoritmusnál, de továbbra is korlátos méretű kizárólagosan-író/megosztottan-olvasható olvasható/írható közös változókat használ. Tudunk olyan megvalósítást tervezni, amely a folyamatok számában mérve lineáris, nem pedig négyzetes?

13-16. *Kutatási feladat.* Tervezzük meg a fényképez atomi objektum olyan megvalósítását, amely lehetővé teszi, hogy ugyanazon komponens módosítása különböző kapukon is előfordulhasson (tehát párhuzamosan).

13-17. Adjuk egyszerűbb változatát a 10.7. alfejezet VÁRÓTEREM algoritmusának amely fényképez közös változót használ. Bizonyítsuk be a helyességet.

13-18. Fogalmazzuk meg pontosan és bizonyítsuk be azt az állítást, amely a megegyezés probléma 1-hibás befejeződéses megoldásának lehetetlenségét állítja, ha fényképez atomi objektumokat használunk.

13-19. Adjuk hatékony megvalósítását az olvas/módosít/fényképez atomi objektumnak kizárólagosan-írható/megosztottan-olvasható olvasható/írható közös változókkal. Bizonyítsuk a helyességet és elemezzük a bonyolultságot.

13-20. Adjunk egyszerűbb változatát a 10.7. alfejezet VÁRÓTEREM algoritmusának, amely olvas/módosít/fényképez közös változókat használ. Próbáljunk amennyire csak tudunk egyszerű és hatékony algoritmust adni. Bizonyítsuk a helyességet és elemezzük a bonyolultságot. A bonyolultsági elemzés során tekintsük az olvas/módosít/fényképez változók megvalósításának költségét a felhasznált kizárólagosan-írható/megosztottan-olvasható olvasható/írható változók modelljében, mint amit a 13.19. példában leírtunk.

13-21. Általánosítsuk a 13.16. lemmát arra az esetre, amely megenged tetszőleges változótípust és nem csak olvasható/írható típusút.

13-22. A VITANYIAWERBUCH algoritmusban szükség van-e az `OLVAS` protokoll „elterjesztő fázisára"? Vagy bizonyítsuk be a szükségességét, vagy mutassunk ellenpéldát.

13-23. Adjunk a VITANYIAWERBUCH algoritmus helyességének bizonyítására olyan változatot, amely ténylegesen megadja a sorbarendező pontok beszúrását tetszőleges olyan végrehajtási sorozatba, amelyben minden művelet befejezett, aztán megmutatja, hogy teljesül az atomiság feltétel.

13-24. Tervezzünk olyan egyszerűsített változatot a VITANYIAWERBUCH algoritmusra, ahol az olvasható/írható közös változók kizárólagosan-írható/megosztottan-olvasható változók. Szükség van az `OLVAS` protokoll elterjesztő fázisára? Bizonyítsuk a helyességet és elemezzük a bonyolultságot.

13-25. Bizonyítsuk be, hogy a BLOOM algoritmusban az `OLVAS` protokoll harmadik olvasása szükséges. Tehát adjuk meg a módosított algoritmus olyan helytelen végrehajtási sorozatát, amelyben a `OLVAS` egyszerűen a megfelelő regiszter már olvasott (az első vagy második olvasáskor talált) értékével tér vissza.

13-26. Az EGÉSZBLOOM algoritmus leírásának a végén az szerepel, hogy ha $|t_1 - t_2| > 1$, akkor a P_i folyamat tetszőlegesen választja azt a regisztert, amelyből újraolvas. Adjunk olyan végrehajtást, amelyben ez az eset előfordul.

13-27. Egészítsük ki a 13.26. lemma hiányzó részleteit. Ez a BLOOM és EGÉSZBLOOM algoritmusok előfeltétel/hatás kódjának megadását kívánja.

13-28. *Kutatási feladat.* Általánosítsuk a BLOOM algoritmust több, mint két íróra.

13-29. Bizonyítsuk be a 13.27. tételt.

13-30. Adjunk olyan végrehajtást amely megmutatja, hogy a FÉNYKÉPREGISZTER algoritmus esetében az alábbi módon megadott sorbarendező pontok egyike sem helyes.

(a) Helyezzük a sorbarendező pontot *OLVAS* esetében a `fényképez`, *ÍR* esetében a `módosít` műveletre.

(b) Minden műveletre helyezzük a sorbarendező pontot a hozzá tartozó `fényképez` műveletre.

13-31. Adjuk meg a FÉNYKÉPREGISZTER algoritmus olyan módosítását, amely a 13.4.5. szakasz végén leírt *mindkét* mód szerinti általánosítása. Vagyis, az az *Ír* folyamat, amelynek a *rag*-ja a legnagyobb, újra felhasználhatja azt (bár nem kötelezően), és amikor valós értékű *rag* is megengedett. Az algoritmus nem determinisztikus legyen, amennyire csak lehet.

13-32. Tervezzünk algoritmust, amely m-író/p-olvasó olvasható/írható V tartományú és v_0 kezdőértékű atomi objektumot valósít meg fényképez közös változóval. A FÉNYKÉPREGISZTER algoritmussal ellentétben, a felhasznált fényképez változó *korlátos* legyen, ha V véges halmaz. (Figyelmeztetés: ez nagyon nehéz feladat.)

13-33. *Kutatási feladat.* A 13.16. lemma felhasználásával bizonyítsuk be az irodalomban megadott néhány atomi regiszter megvalósításának a helyességét.

13-34. *Kutatási feladat.* Tervezzünk *hatékony* és egyszerű megvalósítását a megosztottan-írható/megosztottan-olvasható olvasható/írható atomi objektumnak korlátos méretű egyedi-író/egyedi-olvasó regiszterrel.

13-35. *Kutatási feladat.* Tervezzünk atomi objektumoknak olyan hierachiáját, amely hatékony és egyszerű ahhoz, hogy többprocesszoros rendszerek fejlesztésének alapjául szolgáljon.

III.B. ASZINKRON HÁLÓZATI ALGORITMUSOK

A 14–22. fejezetek *aszinkron hálózati modellekre* érvényes algoritmusokkal foglalkoznak. E modellekben a folyamatok aszinkron hajtják végre az egyes lépéseket és üzenetek cseréjével kommunikálnak. Az ezekben a fejezetekben kifejtett gondolatok több érdekes szálon kapcsolódnak az I. és II.A. részben bemutatottakhoz.

Szokásos módon a formális modellünket tartalmazó fejezettel kezdünk. A 15. fejezet az aszinkron hálózatokra vonatkozó alapalgoritmusok összefoglalását tartalmazza, amelyek mindegyike a modell segítségével programozott. Mivel ezeknek az algoritmusoknak egy része túlságosan bonyolult, a 16–19. fejezetekben négy módszert vezetünk be az aszinkron hálózatok programozásának egyszerűbbé tételére. Az első ilyen módszer, amelyet a 16. fejezetben ismertetünk, a *szinkronizáló* bevezetése. A második, a 17. fejezetben ismertetett technika az aszinkron osztott memóriamodell szimulációja az aszinkron hálózati modell segítségével. A 18. fejezetben leírt harmadik módszer az ellentmondásmentes *logikai idők* hozzárendelése az eseményekhez egy aszinkron osztott hálózatban. A 19. fejezet tartalmazza negyedik módszerünket, az aszinkron hálózati algoritmusok futás közben történő megfigyelését.

Ezek után visszatérünk az aszinkron hálózati környezet speciális problémáinak tanulmányozásához. A 20. fejezet az *erőforrás-hozzárendelés* problémáját tanulmányozza. A 21. fejezet az aszinkron hálózatban végzett számítások problematikáját tekinti át hibás hálózati működést megengedve. Végül a 22. fejezet az *adatvonal problémát*, a megbízható kapcsolattartás megvalósításának problémáját tekinti megbízhatatlan hálózatban.

14. fejezet

Modellezés/IV. Aszinkron hálózati modell

Ebben a fejezetben újra számítási paradigmát váltunk, most az aszinkron közös memóriájú rendszerekről áttérünk az aszinkron hálózatokra. Egy aszinkron hálózat egy kommunikációs alrendszer segítségével kommunikáló folyamatok összessége. A modell legelterjedtebb változatában ez a kapcsolattartás *ponttól pontig* alapon, `küld` és `fogad` műveletek segítségével zajlik. A modell egyéb változatai megengedik az *üzenetszóró* – amikor egy folyamat üzenetet küld a hálózat minden folyamatának (önmagát is beleértve) –, illetve a *többletes üzenetszóró* – amikor egy folyamat folyamatok egy részhalmazának küld üzenetet – kommunikációt. A többletes üzenetszóró modellnek lehetnek speciális esetei, például egy olyan modell, ami kombinálja az üzenetszórást és a ponttól pontig kommunikációt. Minden esetben számítani lehet a hálózat különféle hibás viselkedésére, beleértve az üzenetek elvesztését és többszöröződését is.

A fejezet három fő alfejezetből áll, amelyek sorrendben a küld/fogad-, az üzenetszóró-, és a többletes üzenetszóró rendszerekkel foglalkoznak.

14.1.. Küld/fogad rendszerek

A 2. fejezetben definiált szinkron hálózati modellhez hasonlóan, itt is egy n csúcsból álló, $G = (V, E)$ irányított gráfból indulunk ki. A korábbiakhoz hasonlóan, az irányított gráf i csúcsának kimenő, illetve bejövő szomszédait $ki\_szomszédok_i$-vel, illetve $be\_szomszédok_i$-vel, a G-ben i-ből j-be vezető legrövidebb irányított út hosszát $távolság(i, j)$-vel, a gráf két legtávolabbi csúcsa közötti távolságot pedig $átm$-mel jelöljük.

A szinkron hálózati modellhez hasonlóan, G csúcsaihoz *folyamatokat* rendelünk, amelyek az irányított élekhez rendelt *csatornákon* keresztül kommunikálhatnak. A szinkron modelltől eltérően azonban a kommunikációban nincsenek szinkron fordulók: megengedjük az aszinkronitást mind a folyamat lépéseiben, mind pedig a kommunikációban. Az aszinkronitás leírása céljából a folyamatokat és a csatornákat b/k automataként modellezzük. Legyen M egy rögzített

üzenetábécé.

14.1.1.. Folyamatok

Minden i csúcshoz rendelt folyamatot egy P_i b/k automataként modellezünk. P_i általában rendelkezik valamilyen bemeneti és kimeneti műveletekkel, amelyek segítségével egy külső felhasználóval kommunikál; ez lehetővé teszi, hogy az aszinkron hálózatokkal megoldandó problémákat a „felhasználói felület" történeteinek fogalmaival adjuk meg. Ezenfelül P_i rendelkezik $\texttt{küld}(m)_{i,j}$ alakú kimenetekkel, ahol j kimenő szomszédja i-nek, m pedig egy üzenet (és mint ilyen, eleme M-nek), illetve $\texttt{fogad}(m)_{j,i}$ alakú bemenetekkel, ahol j i bejövő szomszédja. Ezeket a külső felületre vonatkozó megszorításokat leszámítva, P_i tetszőleges b/k automata lehet. (Különleges eredmények elérése érdekében néha további megszorításokat vezetünk be P_i-re vonatkozólag, például korlátozzuk a taszkok, illetve az állapotok számát.) A 8.1.2. példában egy b/k folyamatautomata látható.

A csúcsok folyamatai részéről kétféle hibás viselkedést vizsgálunk: a *megállási hibát*, és a *bizánci hibát*. P_i megállási hibáját egy P_i külső felületébe betett $\texttt{megállít}_i$ bemenő művelettel modellezzük, amely végleg leállítja P_i összes taszkját. (Nem teszünk megszorítást sem a $\texttt{megállít}_i$, sem pedig az azt követő bemeneti tevékenységek által okozott állapotváltozásra. Ezekre az állapotváltozásokra nem fontos megszorításokat tenni, mivel hatásuk P_i-n kívül úgysem látható.) P_i bizánci hibáját úgy modellezzük, hogy megengedjük, hogy P_i-t lecseréljük egy tetszőleges, ugyanolyan külső felülettel rendelkező, b/k automatára.

14.1.2.. Küld/fogad csatornák

A G gráf minden éléhez hozzárendelünk egy csatornát, és az (i, j) irányított élhez rendelt csatornát egy $C_{i,j}$ b/k automataként modellezzünk. Ennek külső felülete $\texttt{küld}(m)_{i,j}$ alakú bemenetekből, és $\texttt{fogad}(m)_{i,j}$ alakú kimenetekből áll, ahol $m \in M$. Általánosságban, ennek a külső felületre vonatkozó megszorításnak a kivételével, a csatorna tetszőleges b/k automata lehet. Az érdekes kommunikációs csatornák azonban rendelkeznek külső viselkedésükre vonatkozó megszorításokkal, például, hogy a fogadott üzeneteket valójában valamivel korábban már elküldték. Az egy csatorna külső felületére vonatkozóan szükséges megszorításokat általánosan egy P történet tulajdonság segítségével fejezhetjük ki, a 8.5.2. szakaszban megadott módon. A megengedett csatornák azok a b/k automaták, amelyek külső leírása *lenyomat*(P), pártatlan történetei pedig *történetek*(P)-ben vannak.

Az ilyen P történet tulajdonság megadása két gyakran alkalmazott módon történhet: *axiómák* felsorolásával, vagy pedig egy konkrét, *lenyomat*(P) külső felülettel, és *történetek*(P) pártatlan történetekkel rendelkező, b/k automata megadásával. Az axiómák megadásának megvan az az előnye, hogy így könnyebb több csatornát megadni, melyek mindegyike az axiómák különböző részhalmazainak felel meg. Másrészről, egy b/k automata explicit megadásának az az előnye, hogy ebben az esetben a folyamatokból és a legáltalánosabb megengedett csatornákból álló teljes rendszer leírható b/k automaták összekapcsolásaként, amely

önmaga szintén b/k automata. Ez lehetővé teszi az automatákra kidolgozott bizonyítási módszerek használatát. Például ez egy, a teljes rendszerre – folyamatokra és csatornákra egyaránt – vonatkozó „állapot" fogalmat biztosít, amelyet invariáns állításokban és szimulációs bizonyításokban használhatunk.

Néha a kívánt történet tulajdonság b/k automataként történő megadása céljából szükség lehet meglehetősen bosszantó programozásra; ez különösen akkor fordul elő, amikor a történet tulajdonság bonyolult *élénkségi megszorításokat* tartalmaz. Ez gyakran egy olyan *kevert stratégiához* vezet, amelyben a biztonságossági tulajdonságokat egy, az invariánsok és a szimulációs bizonyítások támogatásához szükséges gépezetet biztosító alap automatával írjuk le, míg az élénkségi tulajdonságokat speciális élénkségi axiómak segítségével adjuk meg. Ekkor a teljes P történet tulajdonság történetei meg vannak adva, úgy, hogy azok az alap automatának pontosan azok a történetei legyenek, amelyek eleget tesznek az élénkségi axiómáknak.

A szakasz hátralévő részében néhány, a 15–22. fejezetekben használt konkrét küld/fogad csatornáról írunk.

Megbízható FIFO csatorna.. A kommunikációs csatornáról – amint azt a kutatási irodalomban gyakran megteszik – felteszük, hogy *megbízható FIFO csatorna*. Ezen csatorna megengedett viselkedései könnyen leírhatók egy b/k automata helyes történeteként a megfelelő külső felületekkel, melyek állapota egy üzenetekből álló sor. A $\texttt{küld}(m)_{i,j}$ művelet hozzáadja m-et a sor végéhez. A $\texttt{fogad}(m)_{i,j}$ művelet csak akkor engedélyezett, ha m első a sorban, és a hatása az, hogy eltávolítja a sorból az első üzenetet. A taszk partícionálás minden helyileg vezérelt műveletet egy bizonyos osztályban helyez el. Ennek az automatának a formális definícióját a 8.1.1. példában már megadtuk.

Ez az automata nemcsak a megbízható FIFO csatornák megengedett viselkedésének leírása, hanem maga is egy megbízható FIFO csatorna, így ezt az adott külső felülettel rendelkező *univerzális megbízható FIFO csatornának* nevezzük.

Most axiómák segítségével megadjuk a megbízható FIFO csatornák megengedett viselkedésének egy másik leírását. Nevezetesen, definiálunk egy P történet tulajdonságot, ahol $lenyomat(P)$ az adott lenyomat, $történetek(P)$ pedig a $lenyomat(P)$-ben található műveletek olyan β sorozatainak halmaza, amelyek kielégíti az alábbi feltételt.

> Létezik egy *okoz* függvény, mely minden egyes β-ban levő `fogad` eseményt leképez egy azt megelőző, β-ban található `küld` eseményre, úgy, hogy:
>
> 1. minden egyes π *fogad* esemény esetében π-nek és $okoz(\pi)$-nek ugyanazt az üzenet argumentumot kell tartalmaznia;
> 2. *okoz* egy szürjektív leképezés,
> 3. *okoz* egy injektív leképezés;
> 4. *okoz* sorrendtartó, azaz nem léteznek olyan π_1 és π_2 *fogad* események, hogy π_1 megelőzi π_2-t a β-ban, miközben $okoz(\pi_2)$ pedig $okoz(\pi_1)$-t előzi meg a β-ban.

Az *okoz* függvény egy olyan eszköz, mely segítségével azonosítani tudjuk, mely `küld` esemény „okozta" az adott `fogad` eseményeket. Az 1. feltétel azt mondja ki,

hogy csak helyes üzenetek továbbíthatók. A 2. feltétel azt mondja ki, hogy az üzenetek nem vesznek el, a 3. feltétel azt állítja, hogy azok nem többszöröződhetnek meg, a 4. feltétel pedig azt, hogy azok sorrendje nem változhat meg.

Meg kell jegyeznünk, hogy (ezen különleges P történet tulajdonság esetében) az *okoz* függvény a *történetek*(P) minden egyes sorozatára egyedi.

Megbízható újrarendező csatorna.. A csatornák egy másik, gyakran vizsgált típusa, amely garantálja az üzenetek pontosan egyszer történő továbbítását, de nem szükségszerűen tartja meg azok sorrendjét. Az ilyen típusú csatornák számára megengedett viselkedés nem írható le olyan könnyen b/k automaták segítségével, mint az előző esetben, így az automata helyett axiómákat fogunk használni. Nevezetesen, a leírás ugyanaz, mint a megbízható FIFO csatornák fentebb megadott P axiomatikus leírása, azzal a kitétellel, hogy az *okoz* függvényre vonatkozó 4. feltételt kihagyjuk.

Ha egy korábban említett keverék stratégiát alkalmazunk, akkor egy másik, ezzel ekvivalens leírás is megadható. Azaz egy A alap b/k automatát alkalmazunk a biztonságossági tulajdonságok leírására, és további axiómákat az élénkség leírására. Az A alap automatát a következőképpen adjuk meg (itt $\cup$ és $\in$ multihalmaz operátorok).

14.1. automata. *A*

Lenyomat:

Bemeneti:
`küld`$(m)_{i,j}$, $m \in M$

Kimeneti:
`fogad`$(m)_{i,j}$, $m \in M$

Állapotok:

be_ átmenet, M elemeinek multihalmaza, kezdetben üres

Átmenetek:

`küld`$(m)_{i,j}$
Hatás:
$be\_\ átmenet := be\_\ átmenet \cup \{m\}$

`fogad`$(m)_{i,j}$
Előfeltétel:
$m \in be\_\ átmenet$
Hatás:
távolítsuk el m egy példányát
be_ átmenet-ből

Taszkok:

tetszőleges

A taszk partíció nem okoz gondot, mert jelen esetben azt nem használtuk. A P történet tulajdonságot az A automata segítségével adjuk meg. A leírás ugyanaz, mint *lenyomat*(A), a *történetek*(P) pedig az A (nem feltétlenül helyes) α végrehajtási történeteinek halmaza, ahol α kielégíti az alábbi feltételt:

ha α bármely pontjában, bármely $m \in M$ esetében, $m \in$ *be_ átmenet*, akkor

α egy későbbi pontjában bekövetkezik egy `fogad`(m) esemény.

Csatornák hibával.. Olyan küld/fogad csatornákat is vizsgálunk, amelyekben hibák léphetnek fel. A könyvben csak az üzenetvesztésből és többszörözésből származó hibákat tárgyaljuk.

Az olyan csatornák, amelyek üzenetvesztést megengednek, de többszörözést nem, vagy többszörözést igen, de üzenetvesztést nem, vagy mindkettőt megengedik, az *okoz* függvényt alkalmazva ugyanazzal a módszerrel leírhatók, mint a megbízható újrarendező csatornák. Az egyetlen dolog, amit tennünk kell, a 2. feltétel és/vagy a 3. feltétel megfelelő elhagyása.

Gyakran szeretnénk azonban *korlátozott számú* üzenetvesztést és/vagy többszörözést feltételezni. Például, ha üzenetvesztést vizsgálunk, általában nem kívánjuk azt az esetet vizsgálni, mikor *minden* üzenet elvész, mivel ebben az esetben semminek a bekövetkezése sem garantálható. Az üzenetvesztés egy tipikus korlátozása az, amikor azt mondjuk, hogy egy végtelen sokszor elküldött üzenetnek végtelen sokszor meg kell érkeznie. Ennek a formális kifejezésére az *okoz* függvényre az alábbi feltevést tesszük.

Erős veszteségkorlátozás (EVK). Ha (minden egyes adott m-re) végtelen sok `küld` esemény következik be β-ban, akkor végtelen sok `küld` esemény található az *okoz* függvény értékkészletében.

Jegyezzük meg, hogy ezek szerint végtelen sok *különböző* `küld` esemény üzenete kézbesítődött sikeresen. Nem teljesül ez a feltétel például egy olyan sorozatban, amelyben végtelen sok `fogad` esemény fordul elő, de ezeket ugyanaz a `küld` esemény váltotta ki.

A másik jellemző veszteségkorlátozó feltételnél nem említünk semmiféle m-et, hanem azt várjuk el, hogy végtelen sok `küld` esemény végtelen sok `fogad` üzenetet okozzon.

Gyenge veszteségkorlátozás (GyVK). Ha végtelen sok `küld` esemény következik be β-ban, akkor az *okoz* függvény értékkészlete végtelen.

Lehet, hogy a többszörözés esetében azt szeretnénk, ha minden üzenetnek csak véges sok másolata fordulna elő, vagy hogy a másolatok száma egy adott k értéket nem haladna meg.

Véges többszörözés. Az *okoz* függvény csak véges sok `fogad` eseményt rendel hozzá minden egyes adott `küld` eseményhez.

Eddig minden egyes hibát megengedő csatornát axiómák segítségével írtunk le. Most kevert stratégiát fogunk alkalmazni két ilyen csatorna leírására.

14.1.1. példa. Veszteséges FIFO csatorna

Definiálunk egy csatornát, amely megengedi a korlátozott veszteséget, véges többszörözést, de az átrendezést nem. (Ezt a csatornát fogjuk felhasználni a 22.3. alfejezetben a BitVáltó kommunikációs protokoll leírásánál.) Az automata a következő.

14.2. automata. *A*

Lenyomat:
mint rendesen

Állapotok:
sor, M elemeiből álló FIFO sor, kezdetben üres
Átmenetek:

`küld`$(m)_{i,j}$
Hatás:
adjuk hozzá m véges számú másolatát *sor*-hoz

`fogad`$(m)_{i,j}$
Előfeltétel:
m az első a *sor*-ban
Hatás:
távolítsuk el *sor* első elemét

Taszkok:
tetszőleges

A definíciója biztosítja, hogy a csatorna nem rendezi át az üzeneteket és bármely üzenetnek csak véges sok másolatát továbbítja. Ugyanakkor két további, élénkségi feltételre is sükségünk van:

1. ha bármely időpontban a *sor* nem üres, akkor valamely későbbi időpontban egy `fogad` esemény következik be;
2. ha végtelen sok `küld` esemény következik be, akkor közülük végtelen sok helyezi el sikerrel üzeneteit (ezek legalább egy másolatát) *sor*-ban.

Mint korábban is, A-nak és az élénkségi feltételeknek kombinációját használják a történet tulajdonság definiálására. Ebből a történet tulajdonságból következik, hogy ha végtelen sok `küld` esemény következik be, akkor közülük végtelen sokhoz tartozik egy megfelelő `fogad` esemény, azaz ebből a tulajdonságból következik a gyenge veszteségkorlátozás (GyVK).

14.1.2. példa. Veszteséges átrendező csatorna

Definiálunk egy csatornát, amely megengedi a korlátozott veszteséget, a véges többszörözést és az átrendezést. (Ezt a csatornát fogjuk felhasználni a 22.2. alfejezetben a Stenning kommunikációs protokolljának leírásánál.) Az automata a következő.

14.3. automata. *A*

Lenyomat:
mint rendesen

Állapotok:
be_ átmenet, M elemeinek multihalmaza, kezdetben üres

Átmenetek:
`küld`$(m)_{i,j}$
Hatás:
adjuk m véges számú másolatát
be_ átmenet-hez

`fogad`$(m)_{i,j}$
Előfeltétel:
$m \in$ *be_ átmenet*
Hatás:
távolítsuk el m egy másolatát
be_ átmenet-ből

Taszkok:
Tetszőleges

- -

Két élénkségi feltételt fűzünk hozzá:

1. ha bármely pontban *be_ átmenet* nem üres, akkor valamely későbbi pontban egy *fogad* esemény következik be;
2. ha végtelen sok `küld` esemény létezik, akkor közülük végtelen soknak sikerül az üzeneteit (legalább egy másolatban) elhelyezni a *be_ átmenet*-ben.

Ugyanúgy, mint a 14.1.1. példában, a kapott történet tulajdonságból következik, hogy ha végtelen sok `küld` esemény következik be, akkor közülük végtelen sokhoz tartozik megfelelő `fogad` esemény, azaz következik a GyVK feltétel.

Jegyezzük meg, hogy minden egyes történet, amely a 14.1.1. példa leírása alapján megengedett, ezen leírás szerint is megengedett. Léteznek azonban olyan történetek, amelyek e leírás szerint megengedettek, de az előző szerint nem.

14.1.3.. Aszinkron küld/fogad rendszerek

Egy G irányított gráfhoz tartozó *aszinkron küld/fogad hálózati rendszer* a folyamat és csatorna b/k automaták összekapcsolásával adódik a közönséges b/k automata összekapcsolás alkalmazásával. Egy ilyen rendszerre láthatunk példát a 8.3. ábrán. Az összekapcsolás definíció számításba veszi a komponensek közötti helyes kölcsönhatásokat; például, ha a P_i folyamat végrehajt egy `küld`$(m)_{i,j}$ kimeneti műveletet, egy egyidejű `küld`$(m)_{i,j}$ bemeneti művelet hajtódik végre a $C_{i,j}$ csatornán. A megfelelő állapotváltások mindkét komponensben létrejönnek.

Néha kényelmes egy küld/fogad rendszer felhasználóit úgy modellezni, mint egy másik, U b/k automatát. U külső műveletei éppen a felhasználói felületnél levő folyamatok hatásai lesznek. Az U felhasználói automatát gyakran írják

le úgy, mint az alapul vett gráf i-edik csúcsához rendelt U_i felhasználói automaták együttesének összekapcsolását. Ebben az esetben az U_i külső műveletei ugyanazok, mint P_i műveletei a felhasználói felületnél. (Ha megállási hibákat is tekintenek, a `megállít` művelet nem tartozik a felhasználók műveleteihez.)

14.1.4.. Megbízható FIFO csatornákkal rendelkező küld/fogad rendszerek tulajdonságai

A 18. és a 19. fejezetekben történő felhasználáshoz megadunk egy alaptételt az univerzális megbízható FIFO csatornákkal rendelkező aszinkron küld/fogad hálózati rendszerekről. Ez meghatározza azokat a körülményeket, amelyek mellett egy helyes történet eseményei átrendezhetők úgy, hogy ez másik helyes történetet eredményez. (Megjegyezzük, hogy a b/k automata összekapcsolásának formális definíciója alapján, a történetek tartalmazzák a `küld` és `fogad` eseményeket, továbbá a felhasználói felületnél levő eseményeket is.) Ami szükséges, az az, hogy az átrendezés vegyen figyelembe bizonyos alapvető függőségeket: a `fogad` esemény függését a megfelelő `küld` eseménytől (egy egyértelműen meghatározott *okoz* függvényre vonatkozóan) és a (lehetséges) függését minden egyes eseménynek minden más korábbi eseménytől az ugyanazon csúcspontban futó folyamatnál.

Rögzítsünk egy univerzális megbízható FIFO csatornákkal rendelkező A aszinkron küld/fogad rendszert. Legyen β A egy tetszőleges története. Definiáljunk egy $\rightarrow_\beta$ irreflexiv parciális rendezést β eseményein a következőképpen. Ha π és ϕ két esemény β-ban, és π megelőzi ϕ-t, akkor azt mondjuk, hogy $\pi \rightarrow_\beta \phi$, vagy ϕ *függ* π-től, ha az alábbi feltételek egyike teljesül:

1. π és ϕ ugyanannak a P_i folyamatnak az eseményei;
2. π $\texttt{küld}(m)_{i,j}$ alakú és ϕ a megfelelő $\texttt{fogad}(m)_{i,j}$ esemény;
3. π és ϕ 1. és 2. típusú relációk láncán keresztül kapcsolódik egymáshoz.

14.1. tétel. . *Legyen A egy univerzális megbízható FIFO csatornákkal rendelkező aszinkron küld/fogad rendszer, és legyen β A-nak egy helyes története. Legyen γ a β eseményeinek egy átrendezett sorozata, amely megtartja a $\rightarrow_\beta$ rendezést. Akkor γ is helyes története A-nak.*

Bizonyítás. A 8.4. tétel maga után vonja, hogy $\beta|P_i \in$ *pártatlan_ történetek*(P_i) minden egyes i-re. Mivel $\gamma|P_i = \beta|P_i$ minden egyes i-re, következik, hogy $\gamma|P_i \in$ *pártatlan_ történetek*(P_i) minden egyes i-re.

A 8.4. tételből az is következik, hogy $\beta|C_{i,j} \in$ *pártatlan_ történetek*$(C_{i,j})$ minden egyes i és j-re. Mivel $\gamma|C_{i,j}$ az eseményeknek ugyanazt a halmazát tartalmazza, mint $\beta|C_{i,j}$ és az átrendezés megőrzi az események sorrendjét P_i-nél, az események sorrendjét P_j-nél és a `fogad` események sorrendjét a nekik megfelelő `küld` események után, következik, hogy $\gamma|C_{i,j} \in$ *pártatlan_ történetek*$(C_{i,j})$.

Ekkor a 8.6. tételből következik, hogy $\gamma \in$ *pártatlan_ történetek*(A). □

A 14.1. tétel következménye, hogy pártatlan végrehajtások bizonyos átrendezései is pártatlan végrehajtások.

14.2. következmény. . *Legyen A egy univerzális megbízható FIFO csatornákkal rendelkező aszinkron küld/fogad rendszer, és legyen α A-nak egy pártatlan végrehajtási sorozata. Legyen γ a $\beta =$ történet(α) eseményeinek egy átrendezett sorozata, amely megtartja a $\rightarrow_\beta$ rendezést. Akkor létezik A-nak egy pártatlan α' végrehajtási sorozata úgy, hogy történet$(\alpha') = \gamma$, továbbá α és α' megkülönböztethetetlen*[1] *minden egyes P_i folyamat számára.*

Bizonyításvázlat. A 14.1. tétel szerint $\gamma \in$ *pártatlan_történetek*(A). Ezek után a 8.4. és a 8.5. tételek használhatók a kívánt α' létezésének bizonyítására. □

Az α' végrehajtási sorozat – melynek létezését a 14.2. következmény biztosítja – az A-beli folyamatok számára nem különböztethető meg az α eredeti végrehajtási sorozattól (még akkor sem, ha információikat kombinálják). Ez azt jelenti, hogy a folyamatok nem ismerik az események teljes rendezettségét egy végrehajtási sorozatban; nem tudják meghatározni az események sorrendjét különböző folyamatoknál, ha azok az események nem kapcsolódnak az üzenet és a $\rightarrow_\beta$ parciális rendezés által leírt folyamat függőségekkel.

14.1.5.. Bonyolultsági mértékek

A kommunikációs bonyolultságot a küldött és/vagy fogadott üzenetek számával mérjük. Figyelembe vehetjük az üzenetek által tartalmazott bitek számát is.

Az időbonyolultság mérésére a 8.6. fejezetben a b/k automatákra bevezetett általános időbonyolultsági mérték egy speciális esetét használjuk. Azaz minden egyes folyamat minden egyes taszkjához egy ℓ felső korlátot társítunk; ez egy ℓ korlátot ró ki az egymás utáni alkalmak között eltelt időre, mikor az adott taszk egy-egy lépést végrehajthat. Szükségünk van feltevésekre az üzenet kézbesítéséhez szükséges időről is. Az univerzális megbízható FIFO csatorna esetében rendszerint egy d felső korlátot adunk meg az egyes csatornák `fogad` műveletét tartalmazó egyszerű taszkra; ez egy d felső korlátot ró ki a *legrégebbi* üzenet kézbesítési idejére a csatornán. Így a mi szokásos időbonyolultsági mértékünk figyelembe veszi az üzenetek feltorlódásának költségeit a csatornában – a k-adik üzenet a csatorna sorban biztosan kézbesítve lesz kd időn belül.

Néha egy kevésbé reális, de egyszerű feltevéssel élünk a kézbesítési időt illetően: d felső korlát a kézbesítési időre *minden egyes* üzenetre a csatornában, függetlenül a torlódásoktól. Ez a feltevés *nem* fejezhető ki csupán az egyes taszkokhoz rendelt időkorlátok segítségével (mindamellett van értelme). Ezenfelül magától értetődő módon kiterjeszthetjük a csatorna időkorlát feltevéseket a nemuniverzális FIFO csatornákra is.

14.2.. Üzenetszóró rendszerek

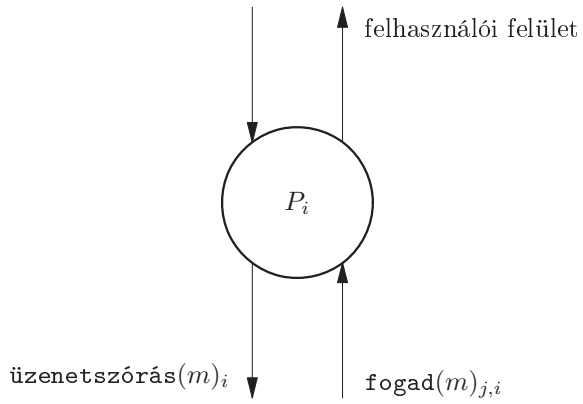
Egy üzenetszóró rendszer folyamatok 1-től n-nig számozott halmazából és egy, az üzenetszóró kommunikációs alrendszert modellező egyszerű üzenetszóró *csa-*

[1] A „megkülönböztethetetlen" 8.7. alfejezetben megadott formális definícióját használjuk.

tornából áll. Legyen újra M egy adott véges üzenet ábécé.

14.2.1.. Folyamatok

Az i-edik folyamatot az üzenetszóró rendszerben egy P_i b/k automatával modellezzük. Mint a küld/fogad hálózati rendszerekben tekintett folyamatok esetében, P_i-nek rendszerint vannak bemeneti és kimeneti műveletei, amelyekkel a külső felhasználókkal kommunikál. Ráadásul P_i-nek vannak `üzenet_szór`$(m)_i$ alakú kimenetei, ahol $m \in M$, és (mint korábban) `fogad`$(m)_{j,i}$ alakú kimenetei, ahol $m \in M$. Ezen külső felületére vonatkozó megszorítások kivételével P_i egy tetszőleges b/k automata lehet. Lásd a 14.1. ábrát.



14.1.. ábra. b/k folyamatautomata egy aszinkron üzenetszóró rendszerhez.

14.2.2.. Üzenetszóró csatorna

Egy üzenetszóró csatornát egyszerű b/k automataként modellezünk. Külső felülete `üzenet_szór`$(m)_i$ alakú bemenetekből és `fogad`$(m)_{i,j}$ alakú kimenetekből áll, ahol $m \in M$. Ebben a könyvben csak megbízható üzenetszóró csatornákat tekintünk, de más típusú üzenetszóró csatornákat is definiálhatunk, amelyek megengednek különböző hibákat.

Megbízható üzenetszóró csatornák.. Egy *megbízható üzenetszóró csatorna* minden kibocsátott üzenetet kézbesít minden egyes folyamathoz, beleértve a feladót is. Egy feltevést fogadunk el az üzenetek kézbesítését illetően: a kézbesítés sorrendje FIFO minden egyes adott folyamatpár között. A megengedett viselkedések egy ilyen csatorna esetében könnyen leírhatók mint egy olyan B egyszerű b/k automata pártatlan történetei, amely külön sorokat tart fenn minden egyes rendezett folyamatpárra.

14.4. automata. *B*

Lenyomat:

Bemeneti:
`üzenet_szór`$(m)_i,\ m \in M,\ 1 \leq i \leq n$

Kimenet:
`fogad`$(m)_{i,j},\ m \in M,\ 1 \leq i,j \leq n$

Állapotok:

$\forall i,j$-re, $1 \leq i,j \leq n$:
legyen $sor(i,j)$, M elemeiből álló FIFO sor, kezdetben üres

Átmenetek:

`üzenet_szór`$(m)_i$
Hatás:
$\forall j$: adjuk hozzá m-et a $sor(i,j)$-hez

`fogad`$(m)_{i,j}$
Előfeltétel:
m az első a $sor(i,j)$-ben
Hatás:
távolítsuk el m-et $sor(i,j)$-ből

Taszkok:

$\forall i,j : \{$`fogad`$(m)_{i,j} :\ m \in M\}$

B-t *univerzális megbízható üzenetszóró csatornának* nevezzük az adott külső felülettel.

14.2.3.. Aszinkron üzenetszóró rendszerek

Egy aszinkron üzenetszóró rendszert a folyamat és az üzenetszóró csatorna b/k automata összekapcsolásával kapunk.

14.2.4.. Megbízható üzenetszóró csatornával rendelkező üzenetszóró rendszerek tulajdonságai

A 14.1.4. szakasz definíciói és eredményei módosíthatók az univerzális megbízható üzenetszóró csatornával rendelkező üzenetszóró rendszerekre. A releváns függőségek most egy `fogad` esemény függősége a hozzátartozó `üzenet_szór` eseménytől és tetszőleges esemény (lehetséges) függősége minden korábbi eseménytől egy adott csúcsbeli folyamatnál.

Rögzítsünk egy univerzális megbízható üzenetszóró csatornával rendelkező A aszinkron üzenetszóró rendszert. Legyen β A egy tetszőleges története. Definiáljunk egy irreflexív parciális rendezést a β eseményein a következőképpen. Ha π és ϕ két esemény β-ban, és π megelőzi ϕ-t, akkor azt mondjuk, hogy $\pi \rightarrow_\beta \phi$, vagy ϕ *függ* π*-től*, ha az alábbi feltételek egyike teljesül:

1. π és ϕ ugyanannak a P_i folyamatnak az eseményei;
2. π `üzenet_szór`$(m)_i$ alakú és ϕ a megfelelő `fogad`$(m)_{i,j}$ esemény;
3. π és ϕ 1. és 2. típusú relációk láncán keresztül kapcsolódik egymáshoz.

14.3. tétel. . *Legyen A egy univerzális megbízható üzenetszóró csatornával rendelkező aszinkron üzenetszóró rendszer és legyen β A-nak egy helyes története. Legyen γ a β eseményeinek egy átrendezett sorozata, amely megtartja a $\rightarrow_\beta$ rendezést. Akkor γ is helyes története A-nak.*

Bizonyítás. A bizonyítást meghagyjuk gyakorlatnak (lásd 14-11. gyakorlat). □

14.4. következmény. . *Legyen A egy univerzális megbízható üzenetszóró csatornával rendelkező aszinkron üzenetszóró rendszer és legyen α A-nak egy pártatlan végrehajtása. Legyen γ a β = történet(α) eseményeinek egy átrendezett sorozata, amely megtartja a $\rightarrow_\beta$ rendezést. Ekkor létezik egy pártatlan α' végrehajtása A-nak úgy, hogy történet(α') = γ, továbbá α és α' megkülönböztethetetlen minden egyes P_i folyamat számára.*

Bizonyítás. A bizonyítást meghagyjuk gyakorlatnak (lásd 14-12. gyakorlat). □

14.2.5.. Bonyolultsági mértékek

A kommunikációs bonyolultságot vagy az `üzenet_szór`, vagy a `fogad` események számával mérhetjük.

Az időbonyolultság mérésére a b/k automatákra bevezetett időbonyolultsági mérték egy speciális esetét használjuk. Nevezetesen minden egyes folyamat minden egyes taszkjához egy ℓ felső korlátot társítunk. Az univerzális megbízható üzenetszóró csatorna speciális esetében pedig rendszerint egy d felső korlátot adunk meg minden egyszerű taszkra; ez egy d felső korlátot ró ki a bármely P_i-től bármely P_j-hez átmenő *legrégebbi* üzenet kézbesítési idejére. Így ismét figyelembe vesszük az üzenetek feltorlódásának költségeit.

Alkalmasint ismét erősebb feltevéssel élhetünk a kézbesítési időt illetően: d felső korlát a kézbesítési időre *minden egyes* üzenetre, továbbá kiterjeszthetjük a csatorna idő feltevéseket a nem-univerzális megbízható üzenetszóró csatornákra is.

14.3.. Többletes üzenetszóró rendszerek

Mind a küld/fogad, mind az üzenetszóró rendszereknek általánosításai a *többletes üzenetszóró rendszerek*, amelyek minden folyamatnak megengedik, hogy üzenetet küldjenek a folyamatok egy részhalmazának a hálózatban. Egy többletes üzenetszóró rendszer tartalmazza *folyamatoknak* egy 1-től n-ig számozott halmazát, továbbá egy *egyszerű többletes üzenetszóró csatornát* a többletes üzenetszóró alrendszer modellezésére. A rendszert (i, I) alakú párok egy $\mathcal{I}$ halmazával parametrizáljuk, ahol i egy folyamatindex és I folyamatindexek egy halmaza. Minden egyes (i, I) pár azt jelzi, hogy az i folyamat az I halmazt használhatja mint üzeneteinek célhalmazát. M ismét egy adott üzenet ábécé.

14.3.1.. Folyamatok

Ismét egy P_i b/k automatát használunk. A felhasználói interfésznél levő műveletek kiegészítéseként P_i-nek vannak $\texttt{t\_üzenetszór}(m)_{i,I}$ alakú kimenetei, ahol m egy üzenet és $(i, I) \in I$, valamint $\texttt{fogad}(m)_{i,j}$ alakú bemenetei. Ezen külső interfészre vonatkozó megszorítások kivételével P_i egy tetszőleges b/k automata lehet.

14.3.2.. Többletes üzenetszóró csatorna

Egy többletes üzenetszóró csatornát úgy modellezünk, mint egy egyszerű b/k automatát. Külső felülete $\texttt{t\_üzenet\_szór}(m)_i$ ($(i, I) \in \mathcal{I}$´alakú bemenetekből és $\texttt{fogad}(m)_{i,j}$ alakú kimenetekből állnak. Csak megbízható többletes üzenetszóró csatornákat tekintünk.

Megbízható többletes üzenetszóró csatornák.. Egy $\mathcal{I}$-beli párokra épülő *megbízható többletes üzenetszóró csatorna* megengedett viselkedései könnyen leírhatók, mint a következő B b/k automata helyes történetei.

14.5. automata. B

Lenyomat:

Bemeneti: $\texttt{t\_üzenet\_szór}(m)_i,\ m \in M,\ 1 \leq i \leq n$

Kimeneti: $\texttt{fogad}(m)_{i,j},\ m \in M,\ 1 \leq i, j \leq n$

Állapotok:

$\forall i, j (1 \leq i, j \leq n)$:

$sor(i, j)$, M elemeiből álló FIFO sor, kezdetben üres

Átmenetek:

$\texttt{t\_üzenet\_szór}(m)_i$

Hatás:

$\forall j \in I$:-re

add hozzá m-et $sor(i, j)$-hez

$\texttt{fogad}(m)_{i,j}$

Előfeltétel:

m az első $sor(i, j)$-ben

Hatás:

távolítsuk el m-et $sor(i, j)$-ből

Taszkok:

$\forall i, j$-re : $\{\texttt{fogad}(m)_{i,j}\ :\ m \in M\}$

B-t *univerzális* megbízható többletes üzenetszóró csatornának nevezzük az adott külső felülettel.

Megbízható többletes üzenetszóró csatornák egy érdekes speciális esete az, amikor a megengedett célhalmazok pontosan az egyelemű halmazok és az összes folyamat teljes $\{1, \ldots, n\}$ halmaza. Ez a csatorna támogatja a ponttól pontig és az üzenetszóró kommunikációk együttesét. Megjegyezzük, hogy a FIFO rendezés biztosított a szórt és a ponttól pontig üzenetek esetében is.

14.3.3.. Aszinkron többletes üzenetszóró rendszerek

Egy aszinkron többletes üzenetszóró rendszert a folyamat és a többletes üzenetszóró csatorna b/k automata összekapcsolásával kapunk. Magától értetődő a 14.1.4 szakaszbeli definíciók és eredmények kiterjesztése a többletes üzenetszóró rendszerekre, amelyek univerzális megbízható többletes üzenetszóró csatornákon alapulnak. Hasonlóképpen az üzenetszóró rendszerekre vonatkoztatott bonyolultsági mértékek is kiterjeszthetők többletes üzenetszóró rendszerekre.

14.4.. Megjegyzések a fejezethez

Általában nem használunk speciális forrást az aszinkron küld/fogad-, üzenetszóró- és többletes üzenetszóró hálózatoknál; hasonló anyag sok – az osztott algoritmusokról és a hálózati protokollok formális verifikációjáról szóló – cikkben található. Az *okoz* függvény használata az üzenetküldés és -fogadás eseményei közötti explicit kapcsolat leírására Fekete, Lynch, Mansour és Spinelli [112], illetve Afek, Attiya, Fekete, Fischer, Lynch, Mansour, Wang és Zuck [4] munkáiból származik.

Az üzenetszóró és többletes üzenetszóró csatornákra megadott modelljeink csak az alapvető helyességi és bonyolultsági tulajdonságokat tartalmazzák. Több erőfeszítés történt üzenetszóró és többletes üzenetszóró rendszerek implementációjára és használatára erősebb feltételek mellett, beleértve szigorúbb rendezési követelményeket és hibatűrési tulajdonságokat. Hadzilacos és Toueg cikke [143] erről jó áttekintést ad.

14.5.. Gyakorlatok

14-1. Legyen P egy megbízható FIFO küld/fogad csatorna megengedett viselkedéseinek leírásához a 14.1.2. szakaszban definiált történet tulajdonság. Bizonyítsuk be, hogy *történetek*(P) pontosan megegyezik az ugyanazzal a külső felülettel rendelkező univerzális megbízható FIFO csatorna pártatlan történeteinek halmazával.

14-2. Legyen A egy b/k automata, amely egy B univerzális megbízható FIFO küld/fogad csatornát valósít meg, azaz A-nak ugyanaz a külső lenyomata, mint B-nek és *pártatlan_ történetek*(A) $\subseteq$ *pártatlan_ történetek*(B). Bizonyítsuk be, hogy valójában *pártatlan_ történetek*(A) = *pártatlan_ történetek*(B). (Ezek szerint minden megbízható FIFO csatorna szükségképpen univerzális.)

14-3. Tekintsünk egy Q alternatív történet tulajdonságot, mint egy megbízható FIFO küld/fogad csatorna megengedett viselkedésének specifikációját. Q ugyanaz, mint P, csupán azt nem követeli meg, hogy *okoz*(π) megelőzze π-t. Mutassuk meg, hogy minden egyes, a megfelelő külső felülettel rendelkező A b/k automatára akkor és csak akkor lesz *pártatlan_ történetek*(A) $\subseteq$ *történetek*(Q), ha *pártatlan_ történetek*(A) $\subseteq$ *történetek*(P).

14-4. Adjuk meg egy olyan C küld/fogad csatorna gondos leírását explicit b/k automataként, amely üzeneteket veszíthet, de ezeket nem többszörözheti és nem rendezheti át. Tegyük fel, hogy C akár összes üzenetét is elveszítheti. C-nek azonban az *összes* olyan lehetséges történettel rendelkeznie kell, amely kielégíti ezt a feltételt. Például nem *követelhetjük meg*, hogy elveszítsen üzeneteket. Definiáljunk egy szimulációs relációt (mint ahogy az a 8.5.5. szakaszban történt) a 14.1.2. szakasz univerzális megbízható FIFO küld/fogad csatornájából C-be és bizonyítsuk be, hogy az tényleg szimulációs reláció.

14-5.

(a) Bizonyítsuk be, hogy a megbízható átrendező küld/fogad csatorna megengedett viselkedéseire megadott két specifikáció ekvivalens.

(b) Definiálhatók-e a megbízható átrendező küld/fogad csatorna megengedett viselkedései ekvivalens módon egy b/k automata segítségével? Azaz van-e olyan, a megfelelő külső lenyomattal rendelkező, b/k automata, amelynek helyes történetei éppen a műveletek leírt sorozatai?

14-6. (Csatorna multiplexelés) Lehetőség van egy egyszerű „valós" küld/fogad csatorna használatára két vagy több „logikai" küld/fogad csatorna megvalósításához, amelyek különböző algoritmusokhoz, vagy egy algoritmus különböző részeihez szükségesek. Formálisan tegyük fel, hogy a P_1 és P_2 két különböző csatorna helyességi követelményeit diszjunkt M_1 és M_2 üzenet ábécék segítségével leíró történet tulajdonságok. Ekkor a szorzat történet tulajdonság $P_1 \times P_2$ (lásd a 8.5.2. szakaszt a történet tulajdonságok szorzatának definíciójára) felfogható mint mindkét feltételrendszert kielégítő másik csatorna leírása.
Példaként tegyük fel, hogy P_1 és P_2 írják le az M_1 és M_2 ABC-ket használó csatornák megengedett viselkedéseit, továbbá P írja le az $M_1 \bigcup M_2$ üzenet ábécét használó csatorna megengedett viselkedéseit.

(a) Bizonyítsuk be, hogy $\mathit{történetek}(P) \subseteq \mathit{történetek}(P_1 \times P_2)$.
Ebből következik, hogy minden egyes P-t megvalósító A b/k automata (abban az értelemben, hogy $\mathit{külső\_lenyomat}(A) = \mathit{lenyomat}(P)$ és $\mathit{pártatlan\_történetek}(A) \subseteq \mathit{történetek}(P)$) ténylegesen a P_1 és P_2 csatornák mindegyikét megvalósítja (abban az értelemben, hogy $\mathit{külső\_lenyomat}(A) = \mathit{lenyomat}(P_1 \times P_2)$ és $\mathit{pártatlan\_történetek}(A) \subseteq \mathit{történetek}(P_1 \times P_2)$).

(b) Mutassuk meg, hogy $\mathit{történetek}(P) \neq \mathit{történetek}(P_1 \times P_2)$.
Ez azt jelenti, hogy P viselkedése jobban korlátozott, mint amennyire az szükséges lenne a P_1 és P_2 csatornák megvalósításához.

14-7. Ismételjük meg a 14-6. gyakorlatot, de a megbízható FIFO küld/fogad csatornák helyett vegyünk olyan csatornákat, amelyek megengedik a tetszőleges átrendezést, az erős veszteségkorlátozást (EVK),

(a) de a többszörözést nem;

(b) és a véges többszörözést;

(c) és a tetszőleges többszörözést.

14-8. Bizonyítsuk be, hogy a FIFO feltevés megbízható küld/fogad csatornákra nem szükséges. Speciálisan, mutassuk meg, hogyan lehet leképezni bármely meg-

bízható FIFO csatornákon alapuló A küld/fogad rendszert megbízható átrendező csatornán alapuló $T(A)$ küld/fogad rendszerbe, amely a környezet számára ugyanolyannak tűnik az alábbi értelemben. $T(A)$ minden egyes pártatlan α végrehajtási sorozatához létezik egy A-nak α' végrehajtási sorozata, amely a műveleteknek ugyanazt a sorozatát veszi tervbe a felhasználói felületnél. Bizonyosodjunk meg az állításunk pontosságáról.

14-9. Bizonyítsuk be, hogy a FIFO feltételezés a megbízható üzenetszóró csatornára nem szükséges. Azaz, mutassuk meg, hogy e feltételezés mellett egy A rendszer transzformálható egy ennek a feltételnek eleget nem tevő $T(A)$ rendszerré, amely környezete számára ugyanolyannak látszik. Bizonyosodjunk meg állításunk pontosságáról.

14-10. Élesítsük a 14.1. tételt úgy, hogy az foglalja magában azt az igényt, hogy mi maradjon meg a felhasználói felületen.

14-11. Bizonyítsuk be a 14.3. tételt.

14-12. Bizonyítsuk be a 14.4. következményt.

15. fejezet

Alapvető aszinkron hálózati algoritmusok

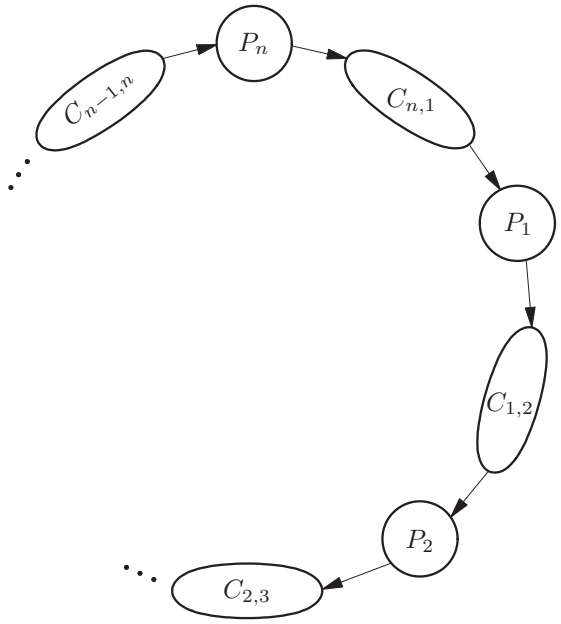
Ebben a fejezetben számos, alapvető problémára (vezetőválasztás, tetszőleges feszítőfa felépítése, üzenetszórás és konvergens üzenetszórás, szélességi keresés, legrövidebb utak megtalálása, minimális feszítőfa megtalálása) megoldást nyújtó algoritmust írunk le az aszinkron hálózati modellben megbízható, FIFO típusú küldő/fogadó csatornák esetében. A problémák nagyrészt azonosak azokkal, melyekkel már a 4. fejezetben találkoztunk a szinkron hálózati modellnél. A problémák itt is arra irányulnak, hogy ki kell választani egy, a hálózattal kapcsolatos számításokat elvégző folyamatot, és fel kell építeni a hatékony kommunikációt támogató struktúrákat. Ebben a fejezetben nem foglalkozunk a hibalehetőségekkel.

Ebben a fejezetben található összes algoritmus a „csupasz" aszinkron hálózatmodell közvetlen programozásával jött létre. Gyorsan beláthatjuk, hogy ezt a modellt sokkal nehezebb programozni, mint a szinkron hálózati modellt. Ez arra fog ösztönözni bennünket, hogy lehetőségeket keressünk a programozás egyszerűsítésére és rendszerezésére. Az ez utáni négy fejezetben (16–19.) bemutatunk négy ilyen egyszerűsítési technikát: *szinkronizálók, közös memória szimulálása, logikai idő és futásidő-figyelés.*

15.1.. Vezető folyamat megválasztása gyűrűben

A vezetőválasztás problémájával már találkoztunk a 3. fejezetben. A feladat aszinkron változatánál az alap irányított gráf ismét egy n folyamatból álló gyűrű, melyben a folyamatok 1-től n-ig vannak számozva az óramutató járásának megfelelően. Mint korábban tettük, gyakran számítjuk ki mod n értékét, megengedve, hogy a 0 jelentse az n-edik folyamatot és így tovább. A gyűrű lehet egy- vagy kétirányú. A 15.1. ábra egy aszinkron egyirányú gyűrűs hálózat architektúráját mutatja be a folyamatokkal és csatornákkal.

A folyamatokat és csatornákat most b/k automataként modellezzük. A szinkron beállításhoz hasonlóan, a folyamatok nem ismerik sem az indexeiket, sem a szomszédjaik indexei, hanem helyi, relatív neveket használnak. Ez lehetővé teszi,



15.1.. ábra. Egyirányú gyűrűs hálózat architektúrája.

hogy tetszőleges folyamatokat tetszőleges sorrendben szervezzünk gyűrűbe. A P_i folyamatautomata csatornákkal való kapcsolattartását biztosító *küld* és `fogad` műveleteken túl a P_i rendelkezik egy `vezető`$_i$ kimeneti művelettel, mellyel bejelentheti vezetővé választását. Itt és az egész fejezetben feltételezzük, hogy a csatornák megbízható FIFO küld/fogad csatornák. Szintén feltételezzük, hogy a folyamatok rendelkeznek UID-kkel. Pontosan egy folyamat esetében a feladat végső soron az, hogy hogyan álljon elő a *vezető* kimenet.

15.1.1.. Az LCR algoritmus

A 3.3. alfejezetben leírt LCR algoritmus könnyen alkalmazható aszinkron hálózat esetében. Emlékezzünk vissza, hogy az LCR algoritmusban minden egyes folyamat körbeküldte az azonosítóját a gyűrűben. Amikor egy folyamat egy bejövő azonosítót fogad, az összehasonlítja a saját azonosítójával. Ha a beérkező azonosító nagyobb a saját azonosítónál, továbbadja, ha kisebb, elveti az azonosítót. Ha az azonosító egyenlő a saját azonosítóval, akkor a folyamat *vezető*$_i$ kimenetet állít elő.

Ugyanez a gondolat működik aszinkron hálózat esetében is. A fő különbség abban áll, hogy most minden egyes folyamat *küldés* pufferének képesnek kell lennie nemcsak egy, hanem bármennyi (egészen n darabig) üzenet tárolására. A különbség oka, hogy az aszinkronitásból kifolyólag az UID-k felhalmozódhatnak a csomópontokban. A módosított algoritmust AszinkLCR-nek nevezzük.

Az alábbi kódban az AszinkLCR$_i$ nevet használjuk az AszinkLCR algoritmusbeli P_i folyamat alternatív neveként. Az algoritmus tárgyalásakor mind az AszinkLCR$_i$ és a P_i neveket használjuk, valamint néha egyszerűen „i-edik folyamat” néven hivatkozunk erre a folyamatra. Hasonló konvenciókat használunk másutt is.

- -

15.1. automata. AszinkLCR$_i$

Lenyomat:

Bemeneti:
- `fogad`$(v)_{i-1,i}$, v egy UID

Kimeneti:
- *küld*$(v)_{i,i+1}$, v egy UID
- `vezető`$_i$

Állapotok:
u, egy UID, kezdetben i UID-je
küld, egy UID-kből álló FIFO sor, kezdetben csak i UID-jét tartalmazza
státus lehetséges értékei: {*ismeretlen, kiválasztott, visszaadott*}
alapértelmezés: *ismeretlen*

Átmenetek:
$küld(v)_{i,i+1}$
Előfeltétel:
v az első *küld* bekövetkeztekor
Hatás:
töröljük *küld* első elemét

$\texttt{vezető}_i$
Előfeltétel:
státus = *kiválasztott*
Hatás:
státus := *visszaadott*

$\texttt{fogad}(v)_{i-1,i}$
Hatás:
case
$v > u$: adjuk v-t *küld*-höz
$v = u$: *státus* := *kiválasztott*
$v < u$: ne csináljunk semmit
endcase

Taszkok:
{$küld(v)_{i,i+1}$: v egy UID}
{$\texttt{vezető}_i$}

Az átmenetek magukért beszélnek. Az P_i-edik folyamat a következő két feladatot hajtja végre: üzeneteket küld az (P_{i+1})-edik folyamatnak, és vezetőnek jelenti be magát. Így két feladata van, egy az összes *küld* művelet és egy a `vezető` művelet számára. Az AszinkLCR viselkedése alapvetően azonos az LCR-évél, de valószínűleg időben „elferdül" attól.

Azért, hogy bebizonyítsuk, az AszinkLCR megoldja a vezetőválasztás problémáját, invariáns állításokat fogunk használni, miként a szinkron LCR algoritmus esetében is tettük. Az invariáns állításokon alapuló bizonyítások éppúgy működnek aszinkron hálózatok esetében is, mint szinkron hálózatok esetében. A fő különbség, hogy a módszert most mélyebben alkalmazva menetek helyett önálló eseményeket kell figyelembe venni.

Technikailag az invariáns állításokon alapuló bizonyításnál minden egyes csatornaautomata állapotszerkezetét ismernünk kell. Ennél fogva kényelmi okokból feltételezzük, hogy a $C_{i,i+1}$ csatornák *univerzális* FIFO-megbízható csatornák a 14.1.2. szakaszban definiáltaknak megfelelően. Ekkor tudjuk, hogy az egyes $C_{i,i+1}$-k állapota egyetlen *sor* komponensből áll, melyre $sor_{i,i+1}$-ként fogunk hivatkozni. Ez a feltételezés nem korlátozza az eredmények általánosságát, mivel egy univerzálisan megbízható FIFO csatornák esetében helyesen működő algoritmusnak tetszőleges megbízhatóságú FIFO csatornák esetében is működnie kell. Ugyanezzel a feltételezéssel fogunk élni az összes megbízható FIFO csatornával

rendelkező küld/fogad rendszer helyességbizonyítása során is.

Jelölje i_{max} a maximális UID-jű folyamat indexét, és jelölje u_{max} az UID-jét. A szinkron esethez hasonlóan itt is két dolgot kell belátnunk:

1. Az i_{max} folyamaton kívül más folyamat nem állít elő idez`vezető` kimenetet.
2. Az i_{max} folyamat esetenként előállítja a „`vezető`" kimenetet.

A két feltétel közül az első egy biztonságossági tulajdonság, míg a másik egy élénkségi tulajdonság.

15.1. lemma. . *Az i_{max} folyamaton kívül más folyamat nem állít elő „`vezető`" kimenetet.*

Bizonyítás. A szinkron eset 3.3.3. állításához hasonló invariánst fogunk használni. Emlékezzünk vissza, hogy a 3.3.3. állítás szerint nincs olyan v UID, mely az i_{max} és a v eredeti i otthona közti bármely *küld* sort elérhetné. Mivel az AszinkLCR algoritmus csatornaautomatákat is magában foglal, olyan kissé erősebb állításra lesz szükségünk, mely a csatornaállapotokban szereplő és a folyamatállapotokban szereplő UID-ket is magában foglalja. Szokás szerint a folyamatállapot-komponenseket a folyamat indexe szerinti, a csatornaállapot-komponenseket pedig a csatorna két indexe szerinti alsóindexben jelöljük.

15.1.1. állítás. *Bármely elérhető állapot esetében igazak az alábbiak:*

1. *Ha $i \neq i_{max}$ és $j \in [i_{max}, i)$, akkor u_i nem jelenik meg $küld_j$-ben.*
2. *Ha $i \neq i_{max}$ és $j \in [i_{max}, i)$, akkor u_i nem jelenik meg $sor_{j,j+1}$-ben.*

A 15.1.1. állítás az adott állapothoz vezető véges végrehajtási sorozat lépésszámán vett indukció alapján bizonyítható. A bizonyítás általában hasonló a 3.3.3. állítás bizonyításához. Most azonban esetelemzést kell végrehajtanunk az egyedi `küld`, `fogad`, és `vezető` eseményeken. A kulcseset a $\texttt{küld}(v)_{j-1,j}$ esemény, ahol $j = i_{max}$. Ebben az esetben ha $v = u_i$ és $i \neq i_{max}$, akkor v eldobódik.

A 15.1.1. állítás segítségével beláthatjuk 15.1.2. helyességét.

15.1.2. állítás. *Bármely elérhető állapot esetében igazak az alábbiak:*
Ha $i \neq i_{max}$, akkor $státus_i = ismeretlen$.

Ekkor könnyen beláthatjuk, hogy i_{max} kivételével egyetlen folyamat sem állít elő $vezető_i$ kimenetet, mivel a művelet előfeltétele sohasem teljesül. □

Most vizsgáljuk meg az élénkségi tulajdonságot. Vegyük észre, hogy ehhez feltételezni kell, hogy az AszinkLCR végrehajtási sorozata *pártatlan*. Ez a formális fogalom azt jelenti, hogy a folyamatok és a csatornák folytatják a munkájuk végrehajtási sorozatát.

15.2. lemma. . *Pártatlan végrehajtási sorozat esetében az i_{max} folyamat esetenként vezető kimenetet állít elő.*

Bizonyítás. E tulajdonság bizonyítása az AszinkLCR esetében nagymértékben eltér a szinkron LCR eset 3.2. lemmában kapott megfelelő eredményének igazolásától. Emlékezzünk vissza, hogy a szinkron esetben egy nagyon erős invariáns állítást használtunk. A 3.3.2. állítás azt írta le, hogy r menet után a maximális UID pontosan hová érkezik. Most nincs menet fogalmunk. Azt is lehetetlen jellemezni, hogy pontosan mi történik a számítás során, mivel az aszinkronitás oly sok bizonytalanságot von maga után. Tehát más módszert kell választanunk.

A bizonyításunk a vezetőválasztás mint fő cél felé vezető mérföldkövek meghatározásán alapul. Részletesen, induktív módon belátjuk, hogy $0 \leq r \leq n-1$-be eső r esetében u_{max} esetenként megjelenik a $küld_{i_{max+r}}$ pufferben. Ezt az állítást $r = n-1$ esetben felhasználva bemutatjuk, hogy u_{max} a $C_{i_{max-1},i_{max}}$ csatornára kerül, és ezután *esetenként* az i_{max} folyamat megkapja, majd i_{max} ezt követően esetenként *vezető* kimenetet állít elő. A folyamat és a csatorna b/k automata pártatlan tulajdonságai használhatók ezen tulajdonságok bebizonyítására.

Tekintsünk például egy α pártatlan végrehajtási sorozat s állapotát, melyben bármely v UID a $küld_i$ puffer elején jelenik meg. Azt állítjuk, hogy $\texttt{küld}(v)_i$ esetenként fellép. Ha nem, akkor az AszinkLCR$_i$ folyamat átmeneteinek vizsgálata azt mutatja, hogy v a $küld_i$ puffer elején marad mindörökre. Ebből az következik, hogy a $\texttt{küld}_i$ feladat engedélyezve marad, és a pártatlanság miatt valamely $\texttt{küld}_i$ eseménynek be kell következnie. De mivel v a $\texttt{küld}_i$ puffer elején lévő üzenet, ez azt vonja maga után, hogy $\texttt{küld}(v)_i$ esetenként be kell következzen.

Ugyanakkor, ha v a $\texttt{küld}_i$ puffer k-adik pozícióján jelenik meg, $k \geq 1$ esetében beláthatjuk, hogy $\texttt{küld}(v)_i$ bekövetkezik. Ehhez k szerinti indukciót kell alkalmazni, a fent megadott $k = 1$ esetből kiindulva. Az induktív lépésben beláthatjuk, hogy egy $k > 1$ pozícióban lévő v UID esetenként eléri a $k-1$ pozíciót, amikor a puffer elejét eltávolítják, s ez az indukciós feltételezés miatt maga után vonja, hogy $\texttt{küld}(v)_i$ esetenként bekövetkezik.

Hasonló érvelés alkalmazható a csatornákban lévő UID-kre. □

Ezeket az állításokat összesítve a következő tételt kapjuk:

15.3. tétel. . *Az* AszinkLCR *megoldja a vezetőválasztás problémáját.*

A következőkben az AszinkLCR algoritmus összetettségét fogjuk vizsgálni. A szinkron LCR algoritmushoz hasonlóan az üzenetek száma $\mathcal{O}(n^2)$. Emlékezzünk vissza, hogy az LCR időkorlátja n menet. Az AszinkLCR időelemzésénél minden folyamat minden feladatára egy ℓ felső korlátot feltételezünk, és minden csatorna sor esetében a legrégebbi üzenet kézbesítésének felső korlátja legyen d.

A 15.2. lemma bizonyításánál használt érvelésbe időkorlátot integrálva egy egyszerű elemzéssel kapjuk az $\mathcal{O}(n^2(\ell + d))$ időkorlátot. Nevezetesen, bármely folyamat $\texttt{küld}$ pufferének és csatorna *sorának* maximális hossza n. Emiatt egy $\texttt{küld}$ pufferben lévő UID a szomszédos csatornába $n\ell$ idő alatt kerül, és egy csatorna *sor*ából nd idő alatt kerül a következő folyamathoz. A végső időkomplexitás tehát $\mathcal{O}(n^2(\ell + d))$.

Ugyanakkor egy részletesebb elemzéssel azt kapjuk, hogy a felső korlát csak $\mathcal{O}(n(l + d))$. Ezen elemzésnek az a lényege, hogy bár a $\texttt{küld}$ pufferek és a *sorok* mérete elérheti az n-et, ez nem fordulhat elő mindenhol. Egy feltorlódás létrejöttéhez bizonyos UID-knek le kell hagyniuk másokat, tehát a legrosszabb felső

korlátnál *gyorsabban* kell utazniuk. A kapott összidő nem rosszabb, mintha az UID-k azonos sebességgel utaztak volna. Belátjuk a következő lemmát:

15.4. lemma. . *Egy pártatlan végrehajtási sorozat során bármely r-re,* $0 \leq r \leq n-1$*, és bármely i-re igazak a következők:*

1. *Az* $r(\ell+d)$ *időpontra az* u_i *UID vagy eléri a* $\texttt{küld}_{i+r}$ *puffert, vagy törlésre kerül.*
2. *Az* $r(\ell+d)+\ell$ *időpontra az* u_i *UID vagy eléri a* $sor_{i+r,i+r+1}$ *sort, vagy törlésre kerül.*

Bizonyítás. r szerinti indukcióval.

Alap : $r = 0$. Az u_i UID $\texttt{küld}_i$-ből indul ki és ℓ idő alatt $sor_{i,i+1}$-be kerül.

Induktív lépés: Tegyük fel, hogy az állítás igaz $r-1$ esetében, és bebizonyítjuk r-re. Rögzítsünk egy i-t. Az első részhez tegyük fel, hogy u_i nem törlődik $r(\ell+d)$ időpontig. Ekkor az induktív feltételezésből kifolyólag $t = (r-1)(\ell+d)+\ell$ időpontra az u_i UID eléri a $sor_{i+r-1,i+r}$-et. □

15.5. segédtétel. . *Ha t időpontra az* u_i *nem kerül az i+r folyamathoz, akkor az* u_i *t időpontra eléri a* $sor_{i+r-1,i+r}$ *sor elejét.*

Bizonyítás. Indirekt módon tegyük fel, hogy u_i nem érkezik el az P_{i+r} folyamathoz és nem is éri el a $sor_{i+r-1,i+r}$ sor elejét t időpontra. Ekkor létezik egy u_i előtt álló u_j UID a $sor_{i+r-1,i+r}$ sorban a t időpillanatban. Ez egy feltorlódás, melyben u_i megelőzte u_j-t. Abból, hogy u_i még nem tett meg r távolságot a gyűrűben következik, hogy még u_j sem tett meg $r-1$ távolságot.

Azonban az indukciós feltevésből következően, u_j vagy eléri $\texttt{küld}_{j+r-1}$-et (azaz legalább $r-1$ távolságra eltávolódik), vagy törlődik *(r-1)*$(\ell+d)<t$ időpontra. Ez maga után vonja, hogy u_j a t időpillanatban nem lehet a $sor_{i+r-1,i+r}$-ban, ami ellentmondás. □

Így u_i a t időpontban vagy eléri az P_{i+r} folyamatot vagy a $sor_{i+r-1,i+r}$ fejét éri el. Ez utóbbi esetben további d idő múlva u_i eléri az P_{i+r} folyamatot. Bármelyik esetben u_i $t+d = r(\ell+d)$ időpontra eléri az P_{i+r} folyamatot, és bekerül a $\texttt{küld}_{i+r}$ pufferbe.

A második rész bizonyítása hasonló.

15.6. tétel. . *Egy vezető esemény bekövetkeztéig eltelt idő bármely pártatlan* AszinkLCR *végrehajtási sorozat esetében legfeljebb* $n(\ell+d)+\ell$ *vagy* $\mathcal{O}(n(\ell+d))$.

Bizonyítás. A 15.4. lemma szerint $r = n-1$ esetében u_{max} UID $(n-1)(\ell+d)+\ell$ időpontra eléri $sor_{i_{max}-1,i_{max}}$-ot, valamint a 15.4. lemma bizonyításánál használt érvelés alapján erre az időpontra eléri a sor első pozícióját. Ekkor egy további d idő múlva u_{max} eléri a i_{max} folyamatot, mely ekkor egy további ℓ idő eltelte után *vezető* kimenetet generál. Az összidő ekkor $n(\ell+d)+\ell$, mint korábban állítottuk. □

Ébresztések.. A vezetőválasztás szokásos be- és kimeneti paramétereit úgy is módosíthatjuk, hogy a bemeneteket (jelen esetben az UID-ket) egy külső U felhasználó speciális `ébreszt`$(v)_i$ üzenetekben küldi a folyamatokhoz ahelyett, hogy a kezdőállapotokban rendelkezésre állnának. Ekkor a helyességi feltételeket úgy kell módosítani, hogy feltételezzük azt, hogy pontosan egy `ébreszt`$(v)_i$ fordul elő minden egyes i esetében. Ekkor az AszinkLCR algoritmus könnyen módosítható úgy, hogy kielégítse az új helyességi feltételeket: egyszerűen egyetlen P_i folyamat sem hajt végre helyi műveletet, amíg `ébreszt` üzenetet nem kapott. Ha a P_i az `ébreszt` megkapása előtt más üzeneteket kapott, akkor ezeket egy új *fogad* pufferben eltárolja, és az `ébreszt` megkapása után dolgozza fel.

15.1.2.. A HS algoritmus

Emlékezzünk vissza, hogy a 3.4. alfejezet HS algoritmusában a folyamatok felkutató üzeneteket küldenek mindenkét irányba a sikeres távolságduplázások érdekében. Világosan látható, hogy ez az algoritmus folyamat b/k automatára megfelelően átírva helyesen működik az aszinkron hálózati modellben is. Mint korábban, a kommunikációs összetettséget itt is $\mathcal{O}(n \log n)$ fejezi ki. Az időbonyolultság felső határának meghatározását feladatként az olvasóra hagyjuk.

15.1.3.. A Peterson-féle vezetőválasztási algoritmus

A HS algoritmus (mind a szinkron, mind az aszinkron esetben) csak $\mathcal{O}(n \log n)$ üzenetet kíván, és kétirányú kommunikációt használ. Ebben a szakaszban bemutatjuk a PetersonVezető algoritmust, mely szintén $\mathcal{O}(n \log n)$ bonyolultság mellett, csak egyirányú kommunikációt használ. Ez az algorimus nem feltételezi a gyűrű csomópontjai számának (n) ismeretét, csak az UID-k összehasonlítását használja. Tetszőleges folyamatot megválaszt vezetőnek, nem csak a legnagyobb vagy legkisebb UID értékűt. Az $\mathcal{O}(n \log n)$ bonyolultságban a konstans tényező csak egy kis szám (közelítőleg 2).

A PetersonVezető algoritmus (vázlatosan)

Az algoritmus végrehajtási sorozata során minden egyes folyamat vagy *aktív* módban vagy *továbbító* módban van. Kezdetben minden folyamat *aktív* módban van. Az aktív folyamatok végzik az „igazi” munkát, a továbbító folyamatok csak továbbadják az üzeneteket. A PetersonVezető algoritmus (aszinkron módon meghatározott) *fázisokra* van osztva. Minden egyes fázisban az aktív folyamatok száma legalább 2-es faktorral csökken, tehát legfeljebb $\log n$ fázis van.

Az algoritmus első fázisában minden egyes P_i folyamat az óramutató járásával megegyező irányban két lépéssel arréb küldi az UID-jét. Ekkor a P_i folyamat összehasonlítja az UID-jét két elődjével az óramutató járásával ellenkező irányban. Ha az ebben az irányban vett szomszéd UID-je a legnagyobb a három közül, azaz $u_{i-1} > u_{i-2}$ és $u_{i-1} > u_i$, akkor a P_i folyamat aktív marad az óramutató járásával ellenkezően vett szomszédja u_{i-1} UID-jét felvéve „ideiglenes UID-nek”. Másrészt, ha valamelyik másik UID

a legnagyobb, akkor az P_i folyamat egyszerűen továbbító lesz a hátralévő végrehajtási sorozatban.

Minden egyes rákövetkező fázis hasonlóan működik. Minden aktív P_i folyamat elküldi az ideiglenes UID-jét a következő illetve a másodikként következő aktív folyamatnak az óramutató járásával megegyezően, és vár a két aktív elődjének az óramutató járásával ellenkező irányból érkező ideiglenes UID-jeire. Ekkor ha az első aktív előd UID-je a legnagyobb a három közül, a P_i folyamat aktív marad ezen előd UID-jét új ideiglenes UID-nek felvéve. Másrészt, ha a maradék két UID között van a legnagyobb, akkor a P_i folyamat továbbítóvá válik.

Ha bármely fázisban egy P_i folyamat azt érzékeli, hogy a közvetlen aktív elődjétől kapott ideiglenes UID azonos a saját ideiglenes UID-jével, akkor a P_i tudja, hogy csak ő maradt aktívnak. Ebben az esetben P_i magát választja meg vezetőnek.

Világos, hogy bármely olyan fázisban, ahol egynél több aktív folyamat van legalább egy folyamat találkozik olyan UID-kombinációval, mely lehetővé teszi, hogy aktív maradjon a következő fázisban. Továbbá, az aktív folyamatok legfeljebb fele élhet túl egy adott fázist, mivel minden aktívan maradó folyamat legalább egy olyan elődde rendelkezik, ami továbbítóvá vált.

15.2. automata. PetersonVezető$_i$

Lenyomat:

Bemeneti:
- `fogad`$(v)_{i-i,i}$, v egy UID

Kimeneti:
- `küld`$(v)_{i,i+1}$, v egy UID
- `vezető`$_i$

Belső:
- `második_uid_be`$_i$
- `harmadik_uid_be`$_i$
- `fázisváltás`$_i$
- `továbbító_lesz`$_i$
- `továbbít`$_i$

Állapotok:

mód $\in \{$*aktív, továbbító*$\}$, kezdetben *aktív*
státus $\in \{$*ismeretlen, kiválasztott, visszaadott*$\}$, kezdetben *ismeretlen*
$uid(j), j \in \{1,2,3\}$, mindegyik egy UID vagy *null*;
kezdetben $uid(1) = P_i$ UID-je, $uid(2) = uid(3) = null$
`küld`, egy UID-kből álló FIFO sor, kezdetben csak P_i UID-jét tartalmazza
fogad, egy UID-kből álló FIFO sor, kezdetben üres

Átmenetek:

`második_uid_be`$_i$
Előfeltétel:
$mód = aktív$
$fogad$ nem üres
$uid(2) = null$
Hatás:
$uid(2) :=$ $fogad$ első eleme
távolítsuk el $fogad$ első elemét
vegyük fel $uid(2)$-t `küld`-be
if $uid(2) = uid(1)$
then $státus := kiválasztott$

`harmadik_uid_be`$_i$
Előfeltétel:
$mód = aktív$
$fogad$ nem üres
$uid(2) \neq null$
$uid(3) = null$
Hatás:
$uid(3) :=$ $fogad$ első eleme
távolítsuk el $fogad$ első elemét

`fázisváltás`$_i$
Előfeltétel:
$mód = aktív$
$uid(3) \neq null$
$uid(2) > max\{uid(1), uid(3)\}$
Hatás:
$uid(1) := uid(2)$
$uid(2) := null$
$uid(3) := null$
vegyük fel $uid(1)$-et `küld`-be

`továbbító_lesz`$_i$
Előfeltétel:
$mód = aktív$
$uid(3) \neq null$
$uid(2) \leq max\{uid(1), uid(3)\}$
Hatás:
$mód :=$ $továbbító$

`továbbító`$_i$
Előfeltétel:
$mód = továbbító$
$fogad$ nem üres
Hatás:
tegyük be $fogad$ első elemét `küld`-be

`vezető`$_i$
Előfeltétel:
$státus = kiválasztott$
Hatás:
$státus = jelentve$

`küld`$(v)_i$
Előfeltétel:
v első a `küldés`-nél
Hatás:
távolítsuk el `küld` első elemét

`fogad`$_i(v)$
Hatás:
vegyük fel v-t $fogad$-ba

Taszkok:
{`küld`$(v)_{i,i+1}$: v egy UID}
{`második_uid_be`$_i$, `harmadik_uid_be`$_i$, `fázisváltás`$_i$, `továbbító_lesz`$_i$, `továbbító`$_i$}
{`vezető`$_i$}

- -

15.7. tétel. . *A* PetersonVezető *megoldja a vezetőválasztás problémát.*

Most az összetettséget fogjuk elemezni. Mint korábban állítottuk, az aktív folyamatok száma minden fázisban legalább megfeleződik, míg egyetlen aktív folyamat nem marad. Ez azt jelenti, hogy a vezetőválasztásig legfeljebb összesen $\lfloor \log n \rfloor + 1$ fázis van. Minden egyes fázis során minden egyes folyamat (aktív vagy

továbbító) legfeljebb két üzenetet küld. Így az algoritmus bármely végrehajtási sorozata során legfeljebb $2n(\lfloor \log n \rfloor + 1)$ üzenet keletkezhet. Ez $O(n \log n)$, sokkal jobb konstans tényezővel, mint a HS algoritmusban.

Ami az időbonyolultságot illeti, nem nehéz bebizonyítani, hogy $\mathcal{O}(n \log n(\ell + d))$ egy egyszerű felső korlát. Ez azért igaz, mert $\mathcal{O}(\log n)$ fázis van, és beláthatjuk, hogy bármely p esetében, az első p fázis befejeződik $\mathcal{O}(pn(\ell + d))$ időn belül. (Minden egyes fázisban minden egyes UID $\mathcal{O}(n)$ távolságot tesz meg a gyűrűben. Egy csomóponttól a következőig egy üzenet legfeljebb $\ell + d$ ideig utazik, feltéve, hogy egy halmozódás nem blokkolja az utat. A 15.4. lemma bizonyításánál használt módszerrel beláthatjuk, hogy egy felhalmozódás nem érinti a legrosszabb esetre megadott korlátot.)

Egy tovább finomított elemzéssel $\mathcal{O}(n(\ell + d))$ felső korlátot kapunk:

15.8. tétel. . *A* PetersonVezető *bármely pártatlan végrehajtási sorozata során egy vezető esemény bekövetkezéséig eltelt idő* $\mathcal{O}(n(\ell + d))$.

Itt csak a bizonyítás fő vázlatát adjuk meg, a bizonyítást összetett gyakorlatként az olvasóra bízzuk.

Bizonyításvázlat. A felhalmozódásokat először figyelmen kívül hagyhatjuk, mivel a 15.4 lemmában használt érveléssel beláthatjuk, hogy a legroszabb eset felső korlátját nem befolyásolják. Az alábbi állítás hasznos lesz az elemzés során. □

15.9. segédtétel. . *Ha az i-edik és j-edik folyamat eltérő, és egy p fázisban mindketten aktívak, akkor az óramutató járásával megegyezően pontosan i után és pontosan j előtt létezik egy k folyamat, amely aktív a p-1-edik fázisban.*

Az időbonyolultság egy olyan üzenetlánc hosszával arányos, amely az utolsó p fázisban olyan üzenettel végződik, mely a vezető i_p-t *kiválasztja*. Abban az üzenetben az UID magában i_p-ben keletkezik és a p fázisban összesen n távolságra kerül. Az i_p folyamat akkor indítja ezt az UID-t, amikor a p fázisba lép. Ez közvetlenül az után történik, hogy i_p a $(p-1)$-edik fázisban megkapja $uid(3)$-ját. Ez az $uid(3)$ történetesen i_p második elődjében keletkezik, mely a $p-1$ fázisban aktív, i_{p-1}-ben, amikor i_{p-1} belép a $(p-1)$-edik fázisba. A 15.9. állítás szerint létezik olyan i_p-től eltérő folyamat, amely eléri a $p-1$ fázist. Ebből az következik, hogy ezen UID által megtehető legnagyobb távolság $p-1$ fázisban n.

Tovább folytatjuk a lánc visszafelé követését. Az i_{p-1} folyamat akkor lép a $p-1$ fázisba, amikor $uid(3)$-ját megkapja a $p-2$ fázisban. Ez az $uid(3)$ i_{p-1} második elődjében keletkezik, mely a $p-2$ fázisban aktív, i_{p-2}-ben, amikor i_{p-2} belép a $p-2$-edik fázisba. A 15.9. állítás felhasználásával beláthatjuk, hogy i_{p-2} nincs hátrább i_{p-1}-től, mint i_{p-1} első elődje, mely aktív a $p-1$ fázisban. Visszafelé haladva definiáljuk az $i_{p-3}, \ldots, i_1$ sorozatot, amelyben minden egyes i_{q-1}-re igaz, hogy nincs hátrébb i_q-tól, mint i_q első elődje, amely aktív a q fázisban.

A 15.9. állítást újra felhasználva bemutathatjuk, hogy a lánc i_{p-1}-től i_1-ig vett teljes hossza legfeljebb n. Ebből az következik, hogy a lánc teljes hossza $3n$, mely $\mathcal{O}(n(\ell + d))$ időkorlátot jelent. □

15.1.4.. Alsó korlát a kommunikációs bonyolultságra

Az előbbiekben bemutattunk két olyan aszinkron hálózati vezetőválasztási algoritmust a PETERSONVEZETŐ-t, és a HS aszinkron változatát, melyek $\mathcal{O}(n \log n)$ kommunikációs bonyolultsággal rendelkeznek. Ebben az alfejezetben belátjuk, hogy létezik egy $\Omega(n \log n)$ alsó határ is. Ebben az alfejezetben, az általánosság elvesztése nélkül feltételezzük, hogy a csatornák univerzálisan megbízható FIFO csatornák.

Emlékezzünk vissza, hogy a szinkron beállítások mellett a 3.9. és a 3.11. tételekben már kaptunk két $\Omega(n \log n)$ alsó határ eredményt a vezetőválasztásra. A 3.9. tétel *összehasonlítás alapú* algoritmusokra adott alsó határt. Ez kétirányú kommunikációt tesz lehetővé és segítségével a folyamatok megállapíthatják a hálózatban lévő csomópontok számát. Ez az eredmény közvetlenül átvihető az aszinkron esetre, mivel a szinkron modell az aszinkron modell egy megszorításaként is megfogalmazható.

15.10. tétel. . *Legyen A egy olyan összehasonlításon alapuló algoritmus, mely n méretű aszinkron gyűrűhálózatban vezetőt választ, ahol kétirányú a kommunikáció, és n ismert a folyamatok számára. Ekkor létezik A egy pártatlan végrehajtási sorozata, melyben $\Omega(n \log n)$ üzenet küldésére kerül sor a vezető megválasztásáig.*

A 3.11. tétel alsó határt ad az olyan algoritmusokra, melyek az UID-ket tetszőleges módon, de rögzített időkorláttal és nagy azonosítótérrel használják. Mindezek mellett kétirányú kommunikációt tesz lehetővé és a folyamatok számára ismert a csomópontok száma. Ezen eredmény egy verzióját átvisszük az aszinkron esetre:

15.11. tétel. . *Legyen A tetszőleges (nem feltétlenül összehasonlításon alapuló) algoritmus, mely n méretű aszinkron gyűrűben vezetőt választ, ahol az UID-k tere végtelen, kétirányú a kommunikáció, és n ismert a folyamatok számára. Ekkor létezik A egy pártatlan végrehajtási sorozata, melyben $\Omega(n \log n)$ üzenet küldésére kerül sor a vezető megválasztásáig.*

Bizonyításvázlat. Ha létezik egyáltalán A olyan pártatlan végrehajtási sorozata, amelyben több mint $n \log n$ üzenet kerül küldésre, mire a vezető megválasztásra kerül, készen vagyunk. Tehát feltehetjük, hogy ez az eset nem áll fenn. „Korlátozzuk" A-t egy olyan S szinkron algoritmusra, amelyben minden egyes menetben bizonyos számú üzenetküldés van. Mivel A bármely pártatlan végrehajtási sorozatában legfeljebb $n \log n$ üzenetküldés történik a vezetőválasztásig, ez azt jelenti, hogy ahhoz, hogy S vezetőt válasszon, legfeljebb $n \log n$ menetre van szükség. Mivel az UID-tér végtelen, a 3.11. tételből következően létezik S-nek olyan végrehajtási sorozata, amelyben $\Omega(n \log n)$ üzenetküldés van a vezetőválasztásig. Ez átkonvertálható A egy pártatlan végrehajtási sorozatává, amelyben $\Omega(n \log n)$ üzenetküldés van a vezetőválasztásig. □

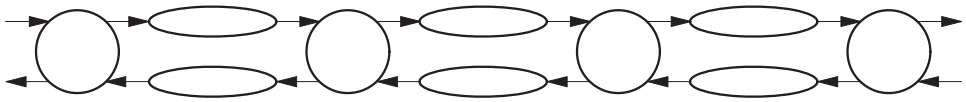
Mivel a 3.11. tétel e könyv egy csillagozott alfejezetében fordul elő, bemutatunk egy alternatív elemibb alsóhatár-bizonyítást nem összehasonlításon alapuló

algoritmusokra. E bizonyítás meglehetősen eltér a 3.9. és 3.11. tételeknél használtaktól abban, hogy az a szinkronitáson és azon a feltételezésen alapul, hogy a folyamatok nem ismerik a gyűrű méretét.

15.12. tétel. . *Legyen A tetszőleges (nem feltétlenül összehasonlításon alapuló) algoritmus, mely egy tetszőleges méretű gyűrűben vezetőt választ, ahol az UID-k tere végtelen, kétirányú a kommunikáció, és a gyűrűméret nem ismert a folyamatok számára. Ekkor létezik A egy pártatlan végrehajtási sorozata, melyben $\Omega(n \log n)$ üzenet küldésére kerül sor.*

A bizonyításhoz szükség van néhány előzetes meghatározásra. Tegyül fel, rendelkezésre áll folyamatautomaták egy univerzális végtelen $\mathcal{P}$ halmaza. Tételezzük fel $\mathcal{P}$ folyamatairól, hogy az UID-k kivetélével azonosak. Szintén tegyük fel, hogy csak helyi nevekkel, mondjuk „bal" és „jobb", ismerik szomszédaikat.

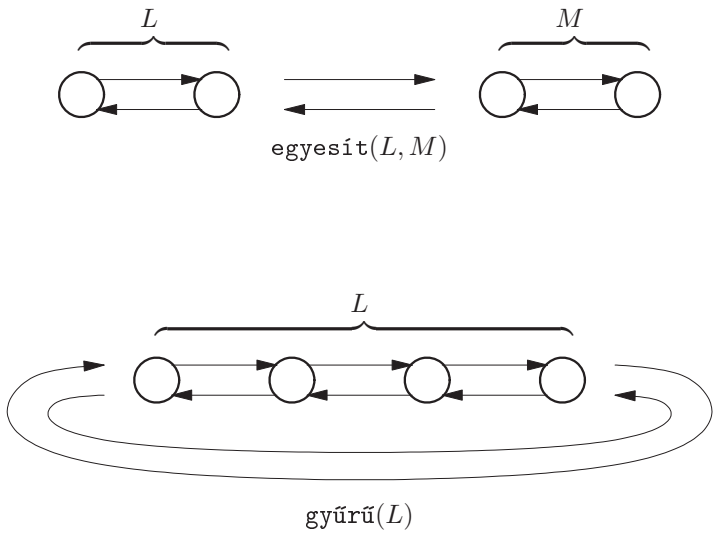
Főleg az érdekel bennünket, hogy hogyan viselkedik egy $\mathcal{P}$-ből vett folyamatautomata gyűjtemény, amikor egy gyűrűbe van rendezve. Azonban annak vizsgálata is hasznos, hogy hogyan viselkednek egyenes vonalba rendezve, ahogy a 15.2. ábra mutatja. $\mathcal{P}$-ből vett különböző folyamatautomaták (b/k automata összekapcsolási művelettel alkotott) mindkét irányú megbízható FIFO küld/fogad csatornákkal összekapcsolt lineáris összekapcsolását *lánc*nak nevezzük.



15.2.. ábra. Automatalánc.

Azt mondjuk, hogy két lánc *diszjunkt*, ha nem tartalmaz közös folyamatautomatát, azaz nincs közös UID. Ha L és M diszjunkt automataláncok, akkor *egyesít* (L, M)-et úgy definiáljuk, mint azt a láncot, mely L és M konkatenációjával jön létre, olyan módon, hogy új megbízható FIFO küld/fogad csatornákat szúrunk be L legjobboldalibb, illetve M legbaloldalibb folyamatába. Az *egyesít* operátor asszociatív, tehát kiterjeszthetjük akárhány láncra. Ha L tetszőleges automatalánc, *gyűrű* (L)-et úgy definiáljuk, mint azt a gyűrűt, melyet az L körbefuttatásával kapunk mindkét irányba új megbízható FIFO küld/fogad csatornákat szúrva L legbaloldalibb illetve legjobboldalibb folyamataiba. Minden folyamat láncbeli jobbszomszédja óramutató járásával megegyező szomszéd lesz a gyűrűben. A *gyűrű* és *egyesít* műveleteket a 15.3. ábra mutatja be. (A csatornákat itt ellipszisek helyett csak nyilakkal ábrázoljuk.)

Ha α egy lánc vagy egy gyűrű végrehajtási sorozata, $C(\alpha)$ definíció szerint az α-ban küldött üzenetek száma. Ha R gyűrű, $C(R)$ akkor $sup\{C(\alpha) : \alpha$ R egy végrehajtási sorozata$\}$, azaz R bármely végrehajtási sorozata során küldött üzenetek maximuma. Lánc esetében a küldhető üzenetek számát „önmagában" számítjuk, azaz nem vesszük figyelembe a sor végén, a környezetből végeken lévő folyamatokba érkező üzeneteket. Így, ha L egy lánc, $C(L)$-et úgy definiáljuk,



15.3.. ábra. Az `egyesít` és `gyűrű` műveletek.

mint $sup\{C(\alpha) : \alpha$ L egy bemenet nélküli végrehajtási sorozata$\}$, azaz L bármely végrehajtási sorozata során küldött üzenetek maximuma egyetlen kívülről végponthoz érkező üzenet nélkül.

Azt mondjuk, hogy egy gyűrű s állapota *néma*, ha nem létezik olyan s-ből kiinduló végrehajtási sorozatbeli szakasz, melynél új üzenetet adnak fel. Azt mondjuk, hogy egy lánc s állapota *néma*, ha nem létezik olyan s-ből kiinduló bemenet nélküli végrehajtási sorozati szakasz, melynél új üzenetet adnak fel. Megjegyezzük, hogy ha egy gyűrű vagy egy sor néma állapotban van, akkor ez *nem* jelenti azt, hogy nem lehetséges további aktivitás. Ez csak azt jelenti, hogy a továbbiakban üzenetküldési esemény nem fordul elő. A folyamatok továbbra is kaphatnak üzeneteket, végrehajthatnak belső lépéseket, és előállíthatnak *vezető* kimenetet.

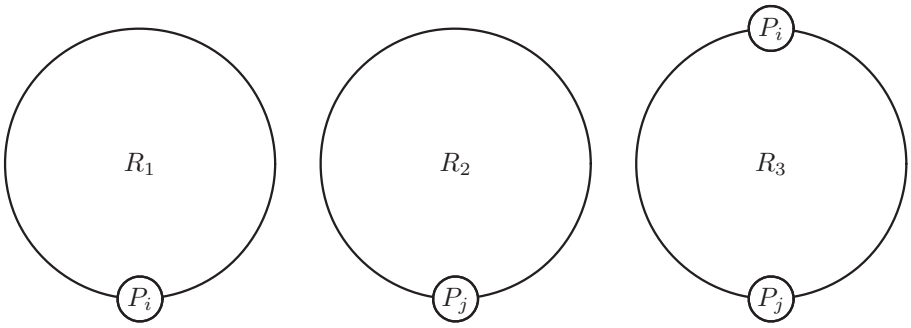
Egy előzetes lemmával fogunk kezdeni.

15.13. lemma. . *$\mathcal{P}$-ben létezik folyamataautomaták egy végtelen halmaza, melyben minden egyes automatára igaz az, hogy legalább egy üzenetet tud küldeni mielőtt bármilyen üzenetet kap.*

Bizonyítás. Valami erősebbet fogunk megmutatni: $\mathcal{P}$-ben egyetlen folyamataautomata kivételével minden automata tud legalább egy üzenetet küldeni, mielőtt bármilyen üzenetet kap.

Ellentmondásként tegyük fel, hogy P_i és P_j két olyan $\mathcal{P}$-beli folyamat, hogy egyik sem tud üzenetet küldeni mielőtt kap egyet. Ekkor tekintsük a 15.4. ábrán bemutatott R_1, R_2, R_3 gyűrűket. (Az egyszerűség kedvéért most a csatornaautomatákat egyáltalán nem ábrázoljuk.)

Mivel sem P_i, sem P_j nem küldhet anélkül üzenetet, hogy egyet kapott volna, a fenti három gyűrűben semmilyen végrehajtási sorozat mellett sem fordulhat elő



15.4.. ábra. A 15.13. lemma bizonyításánál használt R_1, R_2 és R_3 gyűrűk.

üzenetküldés. Így az P_i és P_j folyamatok helyi számításokat és *vezető* műveleteket végezve függetlenül fognak működni, és sohasem hajtanak végre kommunikációs műveleteket. Mivel R_1 megoldja a vezetőválasztás problémát, következésképpen P_i R_1 bármely pártatlan végrehajtási sorozata mellett *vezető* kimenetet produkál. Hasonlóan, mivel R_2 megoldja a vezetőválasztás problémát, következtetésképpen P_j R_2 bármely pártatlan végrehajtási sorozata mellett *vezető* kimenetet produkál. Tekintsük most R_3 bármely α pártatlan végrehajtási sorozatát. Mivel nincs kommunikáció, α-t az P_i folyamat nem tudja megkülönböztetni R_1 valamely pártatlan végrehajtási sorozatától (a 8.7. alfejezetben használt megkülönböztethetetlenség fogalmat felhasználva), végső fokon tehát P_i *vezető* kimenetet generál α-ban. Hasonlóan, α-t a P_j folyamat nem tudja megkülönböztetni R_2 valamely pártatlan végrehajtási sorozatától, tehát P_j *vezető* kimenetet generál α-ban. Ez azonban azt eredményezi, hogy R_3-ban két vezető kerül megválasztásra, ami ellentmondás.

Megmutattuk, hogy nem lehet két olyan P_i és P_j folyamat $\mathcal{P}$-ben, hogy egyik sem küld üzenetet anélkül, hogy üzenetet kapna előbb. Azaz legfeljebb egyetlen olyan folyamat van $\mathcal{P}$-ben, amely nem tud üzenetet küldeni anélkül, hogy kapott volna. Mivel $\mathcal{P}$ végtelen halmaz, ezt az egy folyamatot eltávolítva olyan végtelen halmazt kapunk, amelyben minden egyes folyamat tud küldeni üzenetet anélkül, hogy előbb kapna. □

A 15.12. tétel bizonyítása a következő kulcslemmát használja.

15.14. lemma. . *Minden $r \geq 0$ esetében, létezik páronként diszjunkt láncok olyan $\mathcal{L}_r$ végtelen halmaza, hogy minden $L \in \mathcal{L}_r$ esetében $|L| = 2^r$ és $C(L) \geq r2^{r-2}$.*

Bizonyítás. r szerinti indukcióval.

Alap: $r = 0$. Legyen $\mathcal{L}_0$ $\mathcal{P}$ összes folyamatához tartozó egy csomópontból álló láncok halmaza. Az állítás triviális.

Alap: $r = 1$. Legyen $\mathcal{L}_1$ bármely végtelen halmaza olyan diszjunkt két csomópontból álló láncoknak, melyekben a résztvevő folyamatok bármelyike küldhet üzenetet anélkül, hogy kapna először. Egy ilyen halmaz létezése a 15.13. lemmából következik. Ekkor, ha L egy tetszőleges lánc $\mathcal{L}_1$-ből, léteznie kell L egy

bemenet nélküli olyan végrehajtási sorozatának, ahol legalább egy üzenetküldés történik: egyszerűen küldjön a kettőből az egyik folyamat egy üzenetet anélkül, hogy kapna. Ez kielégíti az állítást.

Induktív lépés: Tegyük fel, hogy $r \geq 2$ és a lemma igaz $(r-1)$-re, azaz létezik egy páronként diszjunkt láncok olyan $\mathcal{L}_{r-1}$ végtelen halmaza, hogy minden $L \in \mathcal{L}_{r-1}$ esetében $|L| = 2^{r-1}$ és $C(L) \geq (r-1)2^{r-3}$. Legyen $n = 2^r$.

Legyen továbbá L, M, N három tetszőleges lánc $\mathcal{L}_1$-ből. Tekintsük a három lánc hat lehetséges egyesítését: *egyesít*(L, M), *egyesít*(M, L), *egyesít*(L, N), *egyesít*(N, L), *egyesít*(M, N), *egyesít*(N, M). A következőkben belátjuk, az alábbi segédtétel helyességét.

15.15. segédtétel. . *E hat láncból legalább egynek van olyan bemenet nélküli végrehajtási sorozata, melyben legalább* $\frac{n}{4}\log n = r2^{r-2}$ *üzenet kerül küldésre.*

A lemma igazsága ezután a 15.15. segédtételből következik, mivel $\mathcal{L}_{r-1}$-ből végtelenül sok három láncból álló halmaz választható ki anélkül, hogy újra használnánk bármelyik folyamatot.

Bizonyítás (15.15. segédtétel) Ellentmondó módon tegyük fel, hogy a hat láncból eggyel sem lehet küldetni $\frac{n}{4}\log n$ üzenetet. Az induktív feltevés szerint létezik egy olyan α_L véges, bemenet nélküli L-beli végrehajtási sorozat, melyre $C(\alpha_L) \geq (r-1)2^{r-3} = \frac{n}{8}\log\frac{n}{2}$. Az általánosság elvesztése nélkül feltételezhetjük, hogy α_L végállapota néma, különben α_L kiterjeszthető lenne egy olyan hosszabb véges végrehajtási sorozattá, amelyben több üzenet generálódik. (Ez a kiterjesztés csak meghatározott módon kivitelezhető, mivel tudjuk, hogy L önmagában nem tud $\frac{n}{4}\log n$ üzenetet küldeni) Hasonlóan, véges bemenet nélküli α_M és α_N végrehajtási sorozatokat kapunk M-ből illetve N-ből ugyanezen tulajdonságokkal.

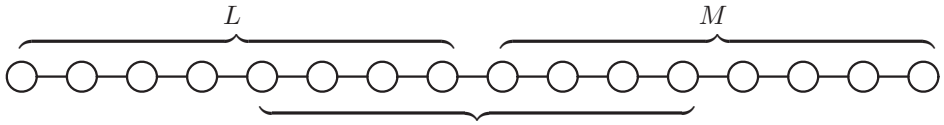
Most szerkesszük meg az *egyesít*(L, M) lánc egy véges $\alpha_{L,M}$ végrehajtási sorozatát. Az $\alpha_{L,M}$ végrehajtási sorozat α_L futtatásával kezdődik L-ben, és α_M-évelM$-ben úgy, hogy az L és M láncokat összekötő csatornákon küldendő üzenetek halasztásra kerülnek. Ekkor $\alpha_{L,M}$-ben legalább $2(\frac{n}{8})\log\frac{n}{2} = \frac{n}{4}(\log(n)-1))$ üzenetküldés van.

A továbbiakban $\alpha_{L,M}$ egy néma állapottal folytatódik. Megjegyezzük, hogy annak ellenére, hogy ez történik, a következő kiterjesztésben küldött további üzenetek számának kifejezetten kevesebbnek kell lennie, mint $\frac{n}{4}$, különben $\alpha_{L,M}$ összes üzenetei számának legalább $(\frac{n}{4}\log n)$-nek kellene lennie, ami ellentmond a feltételezésünknek.

A kiterjesztést olyan módon konstruáljuk meg, hogy csak az LM kapcsolathoz legközelebb eső $\frac{n}{4}-1$ L-beli folyamatnak, illetve a M-ből is csak a kapcsolathoz legközelebb eső $\frac{n}{4}-1$ folyamatnak engedjük meg, hogy lépéseket hajtsanak végre α_L és α_M után addig, amíg a rendszer olyan állapotba nem kerül, hogy egyetlen folyamat sem tud küldeni egyetlen üzenetet sem. Azt állítjuk, hogy az eredményképpen létrejövő *egyesít*(L, M) állapot néma. Ha ugyanis nem lenne az, a kezdeti α_L és α_M szakaszok után legalább $\frac{n}{4}-1$ üzenet küldésére kerülne sor, melyek a kapcsolódásról szállítanak információkat egy, a kapcsolódástól $\frac{n}{4}$ távolságra lévő folyamatnak, s ezzel lehetővé teszik, hogy a folyamat egy újabb üzenetet küldjön. (Ennek belátását az olvasóra bízzuk). Ez azonban összesen legalább $\frac{n}{4}$ újabb

üzenetet jelent α_L és α_M kiterjesztésében, ami lehetetlen. Így a szóban forgó *egyesít*(L, M) állapotnak némának kell lennie.

Informálisan azt mondhatjuk, hogy $\alpha_{L,M}$ után az L és M láncok csatlakozásáról jövő információk nem érik el sem L, sem M középpontját. Mindkét oldalon $\frac{n}{4}$ folyamat tudhat a csatlakozásról, és a csatlakozástól pontosan $\frac{n}{4}$ távolságra lévő két folyamat nem küldhet olyan üzenetet, mely ezen ismeret eredményeképpen jött létre. A 15.5. ábra $n = 16$ esetben ábrázolja L és M csatlakozását.



15.5.. ábra. $\alpha_{L,M}$.

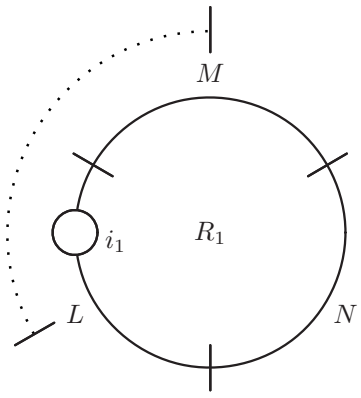
Hasonlóképpen definiálhatjuk az $\alpha_{M,L}$, $\alpha_{L,N}$ stb. véges végrehajtási sorozatokat.

Most az L, M és N sorokból különböző gyűrűket alkotunk, hogy ellentmondást kapjunk. Először legyen R_1 *gyűrű*(*egyesít*(L, M, N)), ahogy a 15.6. ábra bemutatja. Definiáljuk R_1 α_1 pártatlan végrehajtási sorozatát a következőképpen: α_1 végrehajtási sorozata α_L-lel, α_M-mel és α_N-nel kezdődik a három különálló láncot L-et, M-et és N-et némává téve. Ezután α_1 úgy folytatódik, mint $\alpha_{L,M}$, $\alpha_{M,N}$ és $\alpha_{N,L}$ esetében. Mivel azok a folyamatok, melyek tudomással bírnak a csatlakozásról a szomszédos láncokban legfeljebb félútig terjedhetnek, e három kiterjesztés nem zavarja egymást. Továbbá e három kiterjesztés után a teljes gyűrű néma. Ezután α_1 tetszőleges pártatlan módon folytatódik. A helyességi feltételekből következően valamely vezető, például i_1, megválasztásra kerül α_1-ben. Az általánosság elvesztése nélkül feltételezhejük, hogy az i_1 folyamat L középpontja és M középpontja között helyezkedik el, mint ahogy a 15.6. ábra mutatja.

A következőkben legyen $R_2 =$ *gyűrű*(*egyesít*(L, N, M)), és definiáljunk egy α_2 pártatlan végrehajtási sorozatot R_2-ben α_1-hez hasonló módon (ezúttal α_L, α_M, α_N, $\alpha_{L,N}$, $\alpha_{N,M}$ és $\alpha_{M,L}$, segítségével). Ekkor valamilyen i_2 vezető megválasztásra kerül α_2-ben. (Lásd a 15.7. ábrát.)

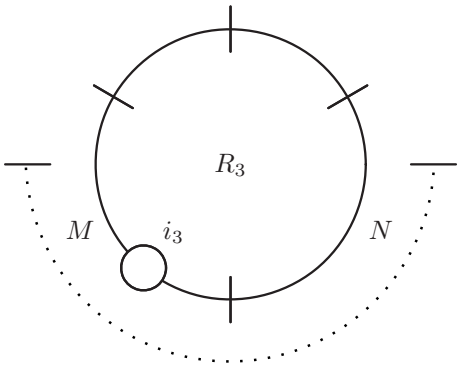
Ezután legyen $R_3 =$ *gyűrű*(*egyesít*(M, N)), és definiáljunk egy α_3 pártatlan végrehajtási sorozatot R_3-ban. (α_M, α_N, $\alpha_{M,N}$ és $\alpha_{N,M}$ segítségével). Itt is valamilyen i_3 vezető megválasztásra kerül α_3-ban. (Lásd a 15.8. ábrát.) Azt állítjuk, hogy i_3 R_3 alsó felébe esik, mint, ahogy ezt a 15.8. ábra mutatja, azaz valahol N középpontja és M középpontja közé az óramutató járásával megegyezően. Ugyanis ha i_3 R_3 felső felében lenne, akkor α_1 és α_3 megkülönböztethetetlen lenne az i_3 folyamat számára, azaz α_1-ben szintén megválasztásra kerülne. De ekkor α_1-ben két különböző folyamat (i_1, i_3) is megválasztásra kerülne, ami ellentmondás. (Az i_1 és i_3 folyamatok különbözőek, mivel i_1 L és M középpontja közé, míg i_3 M és N középpontja közé esik.

Mivel i_3 R_3 alsó felében van, α_2 és α_3 megkülönböztethetetlen i_3 számára. Így



15.6.. ábra. $R_1 = \mathit{gyűrű}(\mathit{egyesít}(L, M, N))$.

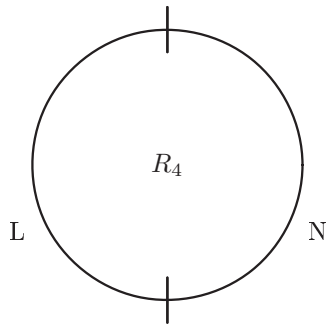
15.7.. ábra. $R_2 = \mathit{gyűrű}\,(\mathit{egyesít}(L, N, M))$.



15.8.. ábra. $R_3 = \mathit{gyűrű}\,(\mathit{egyesít}(M, N))$.

i_3 szintén megválasztásra kerül R_2-ben. Megjegyezzük, hogy i_3 N középpontja és M középpontja közé esik R_2-ben. Mivel α_2-ben csak egy vezető választható, az kapjuk, hogy $i_2 = i_3$. Lásd a 15.7 ábrát.

Végül, legyen $R_4 = \mathit{gyűrű}\,(\mathit{egyesít}(L, N))$, és definiáljunk egy α_4 pártatlan végrehajtási sorozatot R_4-ben (α_L,α_N, $\alpha_{L,N}$, $\alpha_{N,L}$ segítségével). Tekintsük a 15.9 ábrát. Azt állítjuk, hogy α_4-ben nem lehet vezetőt választani. Ha ugyanis lehetne vezetőt választani R_4 felső feléből, akkor az a vezető szintén megválasztásra kerülne α_2-ben, s ezzel két vezető lenne α_2-ben. Ha R_4 alsó részégből lehet vezetőt választani, akkor az a vezető szintén megválasztásra kerülne α_1-ben, s ezzel két vezető lenne α_1-ben. Mindkét út ellentmodásra vezet.



15.9.. ábra. $R_4 =$*gyűrű* (*egyesít*(L, N)).

Az a tény azonban, hogy α_4-ben nincs vezetőválasztás, ellentmond a feltételrendszerünknek. Ez az ellentmondás bizonyítja az állításunkat. □

A lemma közvetlenül következik a 15.15. segédtételből, mint a bizonyítás előtt már jeleztük.

Így mostmár a 15.14 lemmát felhasználva könnyű befejezni a 15.12 tétel bizonyítását.

15.12. tétel bizonyítása. Először, tegyük fel, hogy n kettőhatvány, mondjuk $n = 2^r$. Legyen L egy tetszőleges lánc $\mathcal{L}_r$-ben. A 15.14. lemmából következően $|L| = n$ és $C(L) \geq \frac{n}{4} \log n$. Legyen α L egy bemenet nélküli végrehajtási sorozata, melyre $C(\alpha) \geq \frac{n}{4} \log n$. Legyen $R =$*gyűrű* (L), azaz illesszük L-et egy gyűrűbe. Legyen α' egy olyan R-beli végrehajtási sorozat, mely pontosan úgy viselkedik, mint α L-en, azaz addig, amíg legalább $\frac{n}{4} \log n$ üzenet küldésre nem került, nem küld üzenetet L végpontjai kapcsolódásán keresztül. Ekkor $C(\alpha') \geq \frac{n}{4} \log n$, ami bizonyítja, hogy $C(R) \geq \frac{n}{4} \log n$.

A nem kettőhatvány n értékek esetére a bizonyítást meghagyjuk gyakorlatnak. □

Megjegyezzük, hogy az aszinkronitás és az ismeretlen gyűrűméret voltak a 15.12. tétel bizonyításában kritikus szerepet játszó részek.

15.2.. Vezető folyamat megválasztása tetszőleges hálózatban

Ebben a fejezetben eddig olyan algoritmusokat vizsgáltunk, melyek vezetőt választanak egy aszinkron gyűrűs hálózatban. Most megvizsgáljuk a vezetőválasztás problémáját általánosabb gráfokon alapuló hálózatok esetében. Ebben az alfejezetben feltételezzük, hogy az alapgráf nem irányított, azaz kétirányú kommunikáció van az éleken, és hogy a gráf összefüggő. A folyamatokat az UID-k kivételével azonosnak tételezzük fel.

Emlékezzünk vissza a 4.1.2. alfejezetbeli szinkron hálózatokra vonatkozó MaxTerjed algoritmusra. Ez igényli, hogy a folyamat ismerje *átm*-et, azaz a hálózat átmérőjét. Abban az algoritmusban minden folyamat megjegyzi azt a maximális UID-t, amivel találkozik. Ez kezdetben a saját UID-je. Minden szinkron menetben a folyamat az összes csatornáján elküldi ezt a maximumot. Az algoritmus *átm* menet után véget ér. Az az adott folyamat, mely megtudja, hogy ő rendelkezik a maximális UID-vel, vezetőnek hirdeti ki magát.

A MaxTerjed algoritmus nem terjeszthető ki közvetlenül az aszinkron esetre, mivel az aszinkron modellben nincsenek futamok. Azonban aszinkron módon is lehetséges futamokat szimulálni. Egyszerűen megköveteljük minden egyes folyamattól, hogy egy r-edik menetbeli üzenetet jelöljön meg r-rel, a menet sorszámával. A címzett az összes szomszédjától megvárja az r címkéjű üzenetet, mielőtt végre tudán hajtani az r-edik menetbeli átmenetét. A *átm* menetek szimulálásával az algoritmus helyesen befejeződik.

A szinkron esetnél megadtuk a MaxTerjed algoritmus egy optimalizálását, melyet OptMaxTerjed-nek hívtunk. Ebben az egyes folyamatok csak akkor küldtek üzenetet, ha új információval rendelkeztek, azaz a maximális UID-jük éppen megváltozott. Nem világos, hogy ez az optimalizáció hogyan szimulálható aszinkron esetben. Ha a MaxTerjed-hez hasonlóan egyszerűen megcímkézzük az üzeneteket a menetszámokkal, akkor egy olyan folyamat, mely nem kap üzenetet az r-edik menetben valamely szomszédjától, nem tudja eldönteni, hogy az összes, r-edik menetre vonatkozó üzenetet megkapta-e, tehát nem tudja végrehajtani az r-edik menetbeli átmenetét. Az egymással egyébként nem kommunikáló szomszédok között természetesen küldhetünk üres üzeneteket, de ezek elrontják az optimalizációt.

Egy másik megoldásként, tisztán aszinkron módon szimulálhatjuk az OptMaxTerjed algoritmust: ha egy folyamat új maximális UID-t kap, ezt az UID-t egy későbbi időpontban elküldi a szomszédainak. Ez a stratégia végső soron minden folyamat számára propagálja a maximális UID-t. Van azonban egy feladat: a folyamatok nem tudják mikor kell leállni.

Több különböző megoldást fejlesztettek ki az általános aszinkron hálózatokbeli vezetőválasztás problémájára a következő alfejezetekben és fejezetekben bemutatott technikák segítségével. Az alábbi technikák ezek közé tartoznak.

1. A 15.3. alfejezetbeli keresési algoritmusra alapozott aszinkron üzenetszórás és konvergens üzenetszórás
2. Konvergens üzenetszórás feszítőfával (15.5. alfejezet)
3. Szinkronizáló felhasználása egy szinkron algoritmus szimulálására (16.5.1. szakasz)
4. Egy aszinkron algoritmus befejeződése észlelésére felhasznált globális pillanatkép (19.2.3. szakasz)

15.3.. Feszítőfa konstruálása, üzenetszórás és konvergens üzenetszórás

Egy aszinkron hálózatban a legfontosabb végrehajtandó feladatok közé tartozik egy adott i_0 forráscsomópontból kiinduló feszítőfa megkonstruálása és felhasználása üzenetszórásra és konvergens üzenetszórásra. Ebben az alfejezetben ezen feladatokat elvégző protokollokat mutatunk be. Itt is feltételezzük, hogy az alapgráf $G = (V, E)$ nem irányított és összefüggő. A folyamatoknak nem kell szükségképpen tudniuk a hálózat méretét vagy átmérőjét. Nincs szükség UID-kre.

A feszítőfa problémában az a követelmény, hogy a hálózat minden egyes folyamata esetenként egy *szülő* kimenetes műveleten keresztül jelenteni tudja a G gráf feszítőfájabeli szülőjének a nevét. Emlékezzünk vissza, hogy a 4.2. alfejezetben leírtunk egy olyan szinkron algoritmust, a SzinkSZK-t, mely megkonstruál egy i_0 gyökerű *szélességi* feszítőfát. A SzinkSZK i_0-ból kiindulóan szinkron módon bejárja a gráfot lehetővé téve a nem forrás P_i folyamatok számára, hogy szülőnek jelentsék be az elsőnek megismert szomszédot. Ez az algoritmus futtatható aszinkron beállítások mellett is, és így is garantáltan előállítja a feszítőfát, bár nem feltétlenül a szélességi feszítőfát. Az aszinkron algoritmus kódja a következőkben látható.

15.3. automata. AszinkFeszFa$_i$

Lenyomat:

Bemeneti:
`fogad`(„keres”)$_{j,i}$, $j \in$ *szomszédok*

Kimeneti:
`küld`(„keres”)$_{i,j}$, $j \in$*szomszédok*
`szülő`$(j)_i$, $j \in$*szomszédok*

Állapotok:

szülő $\in$ *szomszédok* $\cup$ {null}, kezdetben null
jelentve, egy logikai, kezdetben *hamis*
$\forall j \in$ *szomszédok* esetében
`küld`$(j) \in$ { *keres*,*null*}, kezdetben *keres*, ha $i = i_0$, egyébként *null*

Átmenetek:

`küld`(„keres”)$_{i,j}$
  Előfeltétel:
    `küld`(j) =*keres*
  Hatás:
    `küld`$(j) :=$ *null*

szülő$(j)_i$
  Előfeltétel:
    szülő $= j$
    jelentve $=$ *hamis*
  Hatás:
    jelentve $:=$ *igaz*

`fogad`(„keres”)$_{j,i}$
  Hatás:
    if $i \neq i_0$ **and** *szülő* $=$ null **then**
      szülő $:= j$
      $\forall k \in$*szomszédok* $-\{j\}$ **do**
        küld$(k) :=$ *keres*

Taszkok:

$\{$*szülő*$(j)_i : j \in$ *szomszédok*$\}$
$\forall j \in$ *szomszédok* esetében
  $\{$`küld`(„keres”)$_{i,j}\}$

15.16. tétel. . *Az* AszinkFeszFa *algoritmus felépíti a feszítőfát.*

Bizonyításvázlat. A bizonyítás kulcsállítása a következő

> **15.3.1. állítás.** *Bármely elérhető állapot esetében, a szülő változók által meghatározott élek G egy i_0-t tartalmazó részgráfjának feszítőfáját alkotják. Továbbá, ha bármely $C_{i,j}$ csatorna esetében létezik egy üzenet, akkor i a feszítőfa része.*

Ez a szokásos indukcióval bizonyítható. Az élénkségi feltétel bebizonyításához, azaz, hogy minden egyes csomópont bekerül a feszítőfába, egy másik invariánst használunk:

> **15.3.2. állítás.** *Bármely elérhető állapot esetében, ha $i = i_0$ vagy szülő$_i \neq$ null és ha $j \in$szomszédok$_i - \{i_0\}$, akkor vagy szülő$_j \neq$ null vagy $C_{i,j}$ egy „keresés” üzenetet tartalmaz vagy* `küld`$(j)_i$ *tartalmaz egy „keresés” üzenetet.*

Ekkor azzal érvelhetünk, hogy bármely $i \neq i_0$ esetében, *távolság*$(i_0, i) \cdot (\ell + d)$ időn belül *szülő*$_i \neq$ *null*, s ebből következik az élénkségi feltétel. □

Bonyolultságelemzés.. AszinkFeszFa bármely pártatlan végrehajtási sorozata esetében az összes üzenet száma $\mathcal{O}(|E|)$, és i_0 kivételével minden folyamat *átm*$(\ell + d) + \ell$ időn belül előállít *szülő* kimenetet. (A felhalmozódások itt nem számítanak, mivel bármely csatornán csak egyetlen üzenet kerül küldésre.)

Megjegyezzük, hogy az AszinkFeszFa által előállított útvonalak a hálózat átmérőjénél hosszabbak lehetnek. Ez azért van, mert egy aszinkron hálózatban

előfordulhat, hogy az üzenetek a hosszabb útvonalakon gyorsabban haladnak, mint a rövidebb útvonalakon. Mindazonáltal, a fa felépítésének idejét az átmérő korlátozza, mivel az egyes folyamatoknál az első *keresés* üzenet megkapásának ideje nem nagyobb, mint az az idő, amely alatt egy üzenet hozzá az i_0-tól a legrövidebb úton utazik.

Üzenetszórás.. Mint a SzinkSZK algoritmusnál is láttuk, az AszinkFeszFa algoritmust könnyen kibővíthetjük abból a célból, hogy az i_0 forrásból kiinduló üzenetszórást megvalósítjuk. Ehhez a feszítőfa felépítése során *keresés* üzenetekkel együtt kell továbbítani az üzeneteket. Ezen üzenetszórás kommunikációs bonyolultsága $\mathcal{O}(|E|)$, ideje $\mathcal{O}(\acute{a}tm(\ell + d))$.

Gyerekmutatók.. Az AszinkFeszFa szintén könnyen kiterjeszthető úgy, hogy a szülők tudják, hogy kik a gyerekeik. Mivel itt kétirányú kommunikációt feltételezünk, csak arra van szükség, hogy egy *keresés* üzenet minden egyes címzettje közvetlenül küldje a megfelelő *szülő* vagy *nem szülő* választ.

Egy gyerekmutatókkal rendelkező, előre felépített feszítőfa felhasználható arra, hogy egy i_0 folyamatból a hálózat többi folyamatához *üzeneteket szórjunk*. i_0 minden egyes üzenetét elküldi a gyerekeinek, majd ezek a szülőktől a gyerekek felé haladva továbbítódnak, amíg a fa levélelemeit el nem érik. Üzenetszórásonként az üzenetek összes száma $\mathcal{O}(n)$, az időbonyolultság $\mathcal{O}(h(\ell + d))$, ahol h a feszítőfa magassága. Itt egy érdekes *időzítési anomália* lép fel: az AszinkFaszítőfa algorimussal készített feszítőfa esetében az üzenetszórás időbonyolultsága $\mathcal{O}(n(\ell + d))$. Ez nem feltétlenül $\mathcal{O}(\acute{a}tm(\ell + d))$, bár maga az AszinkFeszFa algoritmus idejét korlátozza az átmérő. Ez azért van, mert az AszinkFeszFa által létrehozott fa magassága nagyobb lehet, mint az átmérő.

Egy gyerekmutatókkal rendelkező, előre felépített feszítőfa felhasználható üzenetek *konvergens szórására* is a fa összes folyamatától i_0-ba. Ez a szinkron esethez hasonlóan működik: a levélfolyamatok az információt a szüleiknek küldik. Minden i_0-tól eltérő belső folyamat addig vár, míg minden gyerekétől információt nem kap, majd ezeket kombinálva a sajátjával az eredményt elküldi a szülőjének. Végül i_0 addig vár, míg minden gyerekétől nem kap információt, majd ezeket a sajátjával kombinálva előállítja a végeredményt. Az üzenetek száma $\mathcal{O}(n)$, az idő $\mathcal{O}(h(\ell + d))$. Mint a szinkron esetnél is, ez a séma felhasználható osztott bemeneteken alapuló funkciók elvégzésére.

Az üzenetszórás és a konvergens üzenetszórás kombinációja felhasználható arra, hogy i_0 üzenetet küldjön az összes folyamatnak, és fogadja az összes folyamattól jövő nyugtákat. Minden egyes levél, egyszerűen egy konvergens üzenetszórást kezdeményez, amikor megkap egy üzenetszórással érkező üzenetet. Az üzenetek száma szintén $\mathcal{O}(n)$, az idő szintén $\mathcal{O}(h(\ell + d))$.

i_0 számára az is lehetővé tehető, hogy üzenetszórást kezdeményezzen, és a nyugtákat fogadja az összes folyamattól a feszítőfa felépítése közben. Legyen W azon értékek halmaza, melyek üzenetszórhatók. Legyen az M üzenethalmaz $\{(\text{„üzenetszórás”}, w) : w \in W\} \cup \{\text{„nyugta”}\}$.

15.4. automata. AszinkSzórásNyugtáz$_i$

Lenyomat:
Bemeneti:
`fogad`$(m)_{j,i}, m \in M, j \in$ *szomszédok*
Kimeneti:
`küld`$(m)_{i,j}, m \in M, j \in$ *szomszédok*
Belső:
`jelent`$_i$

Állapotok:
$w \in W \cup \{null\}$, kezdetben ha $i = i_0$ az üzenetszórandó érték, különben *null*
szülő $\in$ *szomszédok* $\cup\{null\}$, kezdetben *null*
jelentve, logikai, kezdetben *hamis*
nyugtázva, *szomszédok* egy részhalmaza, kezdetben $\emptyset$
$\forall j \in$ *szomszédok* esetében
`küld`(j), egy FIFO üzenetsor M-ben;
if $i = i_0$, akkor ez az egyetlen („*üzenetszórás*”, w) elemet tartalmazza, ahol $w \in W$ az üzenetszórandó érték. Egyébként ez üres.

Átmenetek:
`küld`$(m)_{i,j}$
Előfeltétel:
m az első `küld`(j)-ben
Hatás:
`küld`(j) első elemének eltávolítása

`fogad`(„*nyugta*”)$_{j,i}$
Hatás:
nyugtázva := *nyugtázva*$\cup\{j\}$
`jelent`$_i(i \neq i_0$ esetében)
Előfeltétel:
szülő$\neq$ *null*
elfogadva = *szomszédok* − *{szülő}*
jelentve = *hamis*
Hatás:
adjuk „*nyugta*”-t *küld*(*szülő*)-höz
jelentve := *igaz*

`fogad`(„üzenetszórás”, $w)_{j,i}$
Hatás:
if *val* = *null* **then**
val := w
szülő:= j
for all $k \in$ *szomszédok*$-\{j\}$ **do**
(„*üzenetszórás*”, w) hozzáadása
`küld`(k)-hoz
else adjuk „*nyugta*”-t *küld*(j)-hez
`jelent`$_i(i = i_0$ esetében)
Előfeltétel:
nyugtázva = *szomszédok*
jelentve = *hamis*
Hatás:
jelentve := *igaz*

Taszkok:
{`jelent`$_i$}
$\forall j \in$ *szomszédok* : $\{$*küld*$(m)_{i,j} : m \in M\}$

Bonyolultságelemzés.. Az összes kommunikáció $\mathcal{O}(|E|)$, az idő $\mathcal{O}(n(\ell+d))$. Az idő felső határa n-től függ és nem az átmérőtől (*átm*) a fent leírt időzítési anomália miatt: az üzenetszórás egy hoszabb úton gyorsan terjedhet és a válaszként érkező nyugták lassan, ha ugyanazon az úton érkeznek vissza. A 16. fejezetben meglátjuk, hogy hogyan kaphatunk olyan algoritmust, amely időbonyolultsága csak az átmérőtől (*átm*) függ.

Hulladékgyűjtés.. Ha az AszinkSzórásNyugtáz-beli fa csak egyetlen üzenet és nyugtája küldésére kell, akkor az egyes folyamatok törölhetik az összes, algoritmussal kapcsolatos információt a `jelentés` művelet végrehajtási sorozata és a *nyugta* küldése után. Ezt a módosítást és bizonyítását feladatnak hagyjuk.

Alkalmazás a vezetőválasztásra.. Az aszinkron üzenetszórás és konvergens üzenetszórás felhasználható a vezetőválasztás problémájának megoldására tetszőleges, megkülönböztetett forráscsomópont nélküli gráfok esetében és olyan esetben, amikor a folyamatok nem ismerik a csomópontok számát vagy a hálózat átmérőjét. A folyamatoknak most UID-kre van szükségük. Egyszerűen megengedjük *minden* csomópontnak, hogy üzenetszórást vagy konvergens üzenetszórást kezdeményezzenek a hálózat maximális UID-jének megismerésének céljából. Az a csomópont, mely úgy találja, hogy a maximális UID egyenlő a saját UID-jével, kinevezi magát vezetőnek. Ez az algoritmus $\mathcal{O}(n|E|)$ üzenetet használ. Az időbonyolultság meghatározását feladatnak hagyjuk.

Ezt az alfejezetet két alapvető, egymáshoz közel álló feladat bemutatásával zárjuk. Mindkét feladat összefüggő, irányítatlan hálózati gráfra vonatkozik, ahol csak helyi ismeretek vannak, nincs kitüntetett csomópont, de vannak UID-ek:

1. a gráf feszítőfájának megtalálása (gyökér kijelölése nélkül);
2. vezető csomópont megválasztása.

Először is, ha adott egy gyökérmegjelölés nélküli feszítőfa, akkor a vezetőt az itt következő módon megválaszthatjuk. Az ötlet ugyanaz, mint amit a 4.4. fejezet végén, a szinkron esetben használtunk.

> **FF-Vezető algoritmus**
>
> Az algoritmus konvergens üzenetszórással *választás* üzeneteket küld a fa levélelemeiből kiindulva. Kezdetben minden egyes levél küldhet *választás* üzenetet az egyedi szomszédjának. Bármely csomópont, mely *választás* üzenetet kap egy kivételével minden szomszédjától, küldhet *választás* üzenetet a maradék szomszédjainak.
>
> Végül két lehetőség van: egy adott folyamat a *választás* üzenet elküldése előtt minden csatornáján kap egy a *választás* üzenetet vagy egy adott élen *választás* üzenetek mindkét irányban utaznak. Az első esetben az a folyamat, melybe a *választás* üzenetek konvergálnak, megválasztja magát vezetőnek. A második esetben, a kérdéses éllel szomszédos két folyamat egyike, mondjuk a nagyobb UID-vel rendelkező, megválasztja magát vezetőnek.

15.17. tétel. . *Az* FF-Vezető *algoritmus egy olyan összefüggő, nem irányított, feszítőfával rendelkező hálózati gráfban, ahol a folyamatok csak helyi ismeretekkel rendelkeznek és van UID-jük, megválaszt egy vezetőt.*

Bonyolultságelemzés.. Az FF-Vezető algoritmus legfeljebb csak n üzenetet használ, és csak $\mathcal{O}(n(\ell + d))$ az időigénye.

Megfordítva, ha adott egy vezető, akkor már megmutattuk, hogy az Aszink-FeszFa algoritmus segítségével hogyan építhetünk fel egy feszítőfát. Ehhez

$\mathcal{O}(|E|)$ üzenetre és $\mathcal{O}(\mathit{átm}(\ell + d))$ időre van szükség. Tehát a vezetőválasztás problémája és egy tetszőleges feszítőfa megtalálása a két algoritmus (meglehetősen alacsony) költségét figyelembe véve ekvivalensek.

15.4.. Szélességi keresés és legrövidebb utak

Vizsgáljuk most meg a 4.2. alfejezetben tárgyalt szélességi keresés (SZK) és a 4.3. fejezetben tárgyalt legrövidebb utak megtalálásának problémáját aszinkron hálózatok esetében. Tegyük most fel, hogy az alapgráf $G = (V, E)$ egy összefüggő, nem irányított gráf, és létezik egy megkülönböztetett i_0 forráscsomópont. A legrövidebb utak esetében azt is feltételezzük, hogy minden egyes irányítatlan él $(i, j) \in E$ rendelkezik egy nemnegatív valós *súllyal* ($\mathit{súly}(i, j)$), melyet a két végpont folyamatai ismernek. Feltételezzük, hogy a folyamatok nem ismerik a hálózat méretét vagy átmérőjét, és hogy nincsenek UID-k.

A szélességi keresésnél a megoldandó feladat az, hogy a hálózat minden egyes folyamata egy `szülő` kimenő művelettel jelentse, hogy ki a szülője egy szélességi feszítőfában. Emlékezzünk vissza, hogy a szinkron esetben ezt az egyszerű SzinkSZK algoritmussal értük el. A SzinkSZK aszinkron változata, a 15.3. szakaszban bemutatott AszinkFeszFa algoritmus. Ez garantáltan előállítja a feszítőfát, de nem feltétlenül egy szélességi feszítőfát.

Az AszinkFeszFa algoritmust lehetséges úgy módosítani, hogy a folyamatok javítsák a hibás `szülő` megjelöléseket, azaz ha egy P_i folyamat eredetileg egy P_j szomszédját szülőnek azonosít, és később egy P_k szomszédtól rövidebb úton kap információt, akkor a P_i folyamat a `szülő` megjelölését P_k-ra változtathatja. Ebben az esetben a P_i folyamatnak tudósítania kell más szomszédait a javításról, hogy azok is módosíthassák `szülő` megjelölésüket. A megfelelő kód a következő.

15.5. automata. AszinkSZK$_i$

Lenyomat:

Bemeneti:	Kimeneti:
`fogad`$(m)_{j,i}, m \in \mathbb{N}, j \in$ *szomszédok*	`küld`$(m)_{i,j}, m \in \mathbb{N}, j \in$ *szomszédok*

Állapotok:

táv$\in \mathbb{N} \cup \{\infty\}$, kezdetben 0 ha $i = i_0$, különben ∞
szülő $\in$ *szomszédok* $\cup \{null\}$, kezdetben *null*
$\forall j \in$ *szomszédok*:
  küld(j), egy FIFO sor $\mathbb{N}$ -beli elemekből,
    kezdetben csak a 0 elemet tartalmazza, ha $i = i_0$
    egyébként üres

Átmenetek:
küld$(m)_{i,j}$
Előfeltétel:
w első *küld*(j)-n
Hatás:
töröljük *küld*(j) első elemét

fogad$(m)_{j,i}$
Hatás:
if $m+1 < \mathit{táv}$ **then**
$\mathit{táv} := m+1$
$\mathit{szülő} := j$
for all $k \in \mathit{szomszédok}$ **do**
vegyük fel *táv*-ot *küld*(k)-ba

Taszkok:
$\forall j \in \mathit{szomszédok}$:
$\{\mathit{küld}(m)_{i,j} : m \in \mathbb{N}\}$

- -

15.18. tétel. . *Az* AszinkSZK *bármely pártatlan végrehajtási sorozata során a rendszer végül egy olyan állapotban stabilizálódik, ahol a szülő változók egy szélességéi feszítőfát reprezentálnak.*

Bizonyításvázlat. Először bebizonyítjuk a következőket:

15.4.1. állítás. *Bármely elérhető állapotra igazak a következők.*

1. *Minden $i \neq i_0$ folyamat esetében, ha $\mathit{táv}_i \neq \infty$, akkor valamely i_0-ból i-be vezető G-beli p út hossza $\mathit{táv}_i$, ahol i elődje $\mathit{szülő}_i$.*
2. *Minden $C_{i,j}$ csatornabeli m üzenet esetében, valamely i_0-ból i-be vezető p út hossza m.*

Ebből következik, hogy bármely i folyamat mindig rendelkezik információról *valamely* i_0-ból i-be vezető p útról. Azért, hogy az élénkségi feltételt belássuk, azaz, hogy minden folyamat végül is megtudja az egyik legrövidebb utat, egy olyan újabb invariánsra lesz szükségünk, melyből következik, hogy a legrövidebb utak „megőrződnek”.

15.4.2. állítás. *Bármely elérhető állapotra igaz a következő. Bármely i és j szomszédpár esetében vagy $\mathit{táv}_j \leq \mathit{táv}_i + 1$, vagy különben akár* küld$(j)_i$*, akár $C_{i,j}$ tartalmazza a $\mathit{táv}_i$ értéket.*

Ezután azzal érvelhetünk, hogy bármely i esetében, $\mathit{táv}_i = \mathit{távolság}(i_0, i)$ *távolság*$(i_0, i) \cdot n(\ell + d)$ időn belül. Ez a *távolság*(i, i_0)-n vett indukcióval bizonyítható. (A felhalmozódásokat itt figyelembe vettük). Ez elégséges az élénkségi feltétel bizonyítására. □

Bonyolultságelemzés.. Az AszinkSZK algoritmus végrehajtási sorozata során küldött üzenetek száma $\mathcal{O}(n|E|)$. Ez azért ennyi, mert minden egyes csomópont legfeljebb n különböző becslést kaphat i_0-tól vett távolságára, s ezek az érintkező éleken konstans számú üzenetet generálnak. Az az idő, amíg a rendszer stabil állapotba nem kerül $\mathcal{O}(\mathit{átm} \cdot n(\ell + d))$, mivel az i_0-tól bármely más csomópontba vett legrövidebb út hossza legfeljebb *átm*, és legfeljebb n üzenet van bármikor a csatornákon. (A felhalmozódásokat itt is számításba vettük.)

Befejeződés.. Az AszinkSZK algoritmus problémája az, hogy egy folyamat nem tudja eldönteni, hogy szükség van-e további javításokra. (Megoldható lenne, ha a hálózatméret ismert.) Így az algoritmus technikailag nem megoldás az SZK problémára, mivel sohasem állítja elő a kívánt *szülő* kimeneteket. Az összes üzenet nyugtázásával valamint az AszinkSzórásNyugtáz-hoz hasonlóan a nyugtákat konvergensen i_0-ba szórva az AszinkSZK algoritmus kiterjeszthető úgy, hogy előállítsa e kimeneteket. Ez ugyanis lehetővé teszi i_0 számára, hogy megtudja mikor kerül a rendszer egy stabil állapotba, s ekkor üzenetszórással jelezhet az összes folyamatnak, hogy állítsák elő a *szülő* kimeneteket.

Ez a konvergens üzenetszórás egy kicsit bonyolult, mivel az AszinkSzórás-Nyugtáz-tól eltérően egy i-edik folyamatnak lehet, hogy többször is közre kell működnie. Minden egyes alkalommal, amikor egy i folyamat megkap egy új *táv* becslést a j szomszédjától, és a többi szomszédjának elküldi a módosításokat, meg kell várnia az összes szomszéd nyugtáját, mielőtt maga is nyugtát küldhetne j-nek. A különböző nyugtahalmazokat külön-külön kell naplózással nyilvántartani. Ezt a feladatot gyakorlatnak hagyjuk.

Ismert átmérő.. Ha *átm* ismert, akkor az AszinkSZK algoritmus feljavítható úgy, hogy csak olyan távolságbecsléseket használunk, amelyekben a távolság kisebb vagy egyenlő *átm*-mel. Ezzel a módosítással minden egyes csomópont legfeljebb *átm* különböző becslést kaphat az i_0-tól vett távolságára. Ez $\mathcal{O}(\mathit{átm}|E|)$ kommunikációs, és $\mathcal{O}(\mathit{átm}^2(\ell+d))$ időbonyolultsághoz vezet. A bonyolultsági korlátok nem változnak a fent leírt megállással együtt sem.

Most megadunk egy másik megoldást. Ez előállítja a kívánt *szülő* kimeneteket. A hálózati gráf átmérőjének vagy méretének ismeretére nem lesz szükség. Ez a megoldás kisebb kommunikációs bonyolultsággal rendelkezik, mint az AszinkSZK bármely változata, de nagyobb az időbonyolultsága, mint az AszinkSZK-nek ismert átmérő (*átm*) mellett.

RétegzettSZK algoritmus (vázlatosan)

Az SZK fát ún. *rétegekben* hozzuk létre, ahol minden egyes k réteget a fa k mélységben lévő csomópontjai alkotnak. A rétegeket az i_0 folyamat által koordinált fázissorozatok hozzák létre, melyek egy egy réteghez kapcsolódnak.

Az első fázisban az i_0 *keresés* üzenetet küld minden egyes szomszédjának, és vár a nyugtákra. Egy, az első fázisban *keresés* üzenetet kapó folyamat pozitív nyugtát küld. Ez a fa 1-es mélységben lévő összes folyamata számára lehetővé teszi, hogy meghatározzák a szülőjüket, nevezetesen i_0-t, és i_0 természetesen ismeri a gyerekeit. Ez hozza létre az első réteget.

Induktív módon tegyük fel, hogy k fázis befejeződött, és az első k réteg létrejött: minden egyes, legfeljebb k mélységben lévő folyamat ismeri az SZK fában a szülőjét, és a legfeljebb $k-1$ mélységben lévő ismeri a gyerekeit. Továbbá, az i_0 tudja, hogy ez a szint már elkészült. A $(k+1)$-edik réteg elkészítéséhez a $(k+1)$-edik fázisban az i_0 egy *új_fázis* üzenetet szór a feszítőfa már felépített élei mentén a k mélységben lévő folyamatokhoz.

A k mélységben lévő folyamatok az *új_fázis* üzenet hatására *keresés* üzenetet küldenek a *szülőjük* kivételével az összes szomszédjuknak, és várnak a nyugtára. Amikor egy nem-i_0 folyamat egy végrehajtási sorozat során megkapja az első *keresés* üzenetet, a feladót *szülőnek* minősíti, és pozitív nyugtát küld. Amikor egy nem-i_0 folyamat egy végrehajtási sorozat során egy következő *keresés* üzenetet kap, negatív nyugtát küld. i_0 bármilyen *keresés* üzenetre negatív nyugtát küld. Amikor egy k mélységben lévő folyamat az összes *keresés* üzenetére megkapja a nyugtát, a pozitív nyugtát küldő folyamatokat a gyerekeinek minősíti.

Ezután a k mélységben lévő folyamatok konvergens úton visszaküldik a k mélységű feszítőfán i_0-nak azt az információt, hogy meghatározták a gyerekeiket. Azt az egy bites információt is visszaküldik, hogy találtak-e $k+1$ mélységű csomópontot. Az i_0 folyamat akkor állítja le az algoritmust, ha nincs több felfedezett új csomópont.

15.19. tétel. . *A* RétegzettSZK *algoritmus előállítja a szélességi feszítőfát.*

Bonyolultságelemzés.. A RétegzettSZK algoritmus $\mathcal{O}(|E| + n \cdot \mathit{átm})$ üzenetet használ. Összesen $\mathcal{O}(|E|)$ *keresés* és nyugta üzenet van, mivel az egyes élek irányonként legfeljebb egyszer derítődnek fel. Szintén, minden fázisban minden faélen végighalad legfeljebb egyszer egy *új_fázis* üzenet és a konvergens üzenetek. Mivel legfeljebb $\mathit{átm}+1$ fázis van, ilyen üzenetből legfeljebb $\mathcal{O}(n \cdot \mathit{átm})$ van. Az egyes fázisok $\mathcal{O}(\mathit{átm}(\ell + d))$ időt vesznek igénybe, tehát az időbonyolultság $\mathcal{O}(\mathit{átm}^2(\ell + d))$.

Az ismert *átm* átmérő melletti AszinkSZK algoritmus és a RétegzettSZK algoritmus a kommunikációs és az időbonyolultság közti kompromisszumot illusztrálják. Ez a kompromisszum tovább illusztrálható az AszinkSZK algoritmus és a RétegzettSZK algoritmus keverésével. A HibridSZK algoritmus egy m paramétert használ, ahol $1 \leq m \leq \mathit{átm}$. Ha $m = 1$, a HibridSZK azonos a RétegzettSZK-val. Ha $m = \mathit{átm}$, akkor a HibridSZK olyan, mint az ismert *átm* átmérő melletti AszinkSZK. m közbenső értékeire az idő- és kommunikációs bonyolultsági értékek a RétegzettSZK és az ismert átmérő melletti AszinkSZK idő- és kommunikációs bonyolultsági értékei közé esnek.

HibridSZK algoritmus (vázlatosan)

Az algoritmus *fázisokban* működik. Az egyes fázisokban az SZK fa m rétege kerül meghatározásra (nem csak egy, mint a RétegzettSZK esetében). Az egyes fázisokban a következő m réteg aszinkron módon kerül feltárásra olyan helyesbítésekkel, mint amilyeneket az AszinkSZK algoritmus használ. A nyugták konvergens üzenetszórással kerülnek vissza i_0-hoz. A konvergens üzenetszórás befejeződésekor az i_0 folyamat tudja, hogy az aktuális fázisban a felfedezett réteg minden folyamata helyes távolságbecsléssel stabilizálódott.

Bonyolultságelemzés.. A HibridSZK algoritmus kommunikációs bonyolultsága $\mathcal{O}(m|E| + \frac{n \cdot \mathit{átm}}{m})$. Összesen $\mathcal{O}(m|E|)$ *keresés* és nyugta üzenet van, mivel

az egyes élek legfeljebb m különböző távolságbecslésről hordoznak információt. Az *új_fázis* és a konvergens üzenetek fázisonként minden faélen legfeljebb egyszer haladnak végig. Mivel legfeljebb $\mathcal{O}(\frac{átm}{m})$ fázis van, ilyen üzenetből legfeljebb $\mathcal{O}(\frac{n \cdot átm}{m})$ van. Az egyes fázisok $\mathcal{O}(átm(\ell + d) + m^2(\ell + d))$ időt vesznek igénybe. (Az m^2 érték egy önálló csatornán feltorlódható m üzenetszámból jön.) Így az időbonyolultság $\mathcal{O}(\frac{átm^2}{m}(\ell + d) + átm \cdot m(\ell + d))$.

A szélességi keresés problémájára három algoritmust adtunk meg: Az AszinkSZK algoritmust (befejeződéssel), a RétegzettSZK algoritmust és a HibridSZK algoritmust. Hogy egyszerűen összehasonlíthassuk a három algoritmust, az AszinkSZK azon verzióját tekintjük, ahol befejeződik, és az *átm* ismert. Elhanyagoljuk az ℓ helyi feldolgozási időt és a kapcsolatokon lévő felhalmozódások hatását. Egy csatornán futó üzenet kézbesítési idejének felső határát d-vel jelöljük. Ekkor a következőket kapjuk:

	Üzenetek száma	Idő
AszinkSZK	$\mathcal{O}(átm\|E\|)$	$\mathcal{O}(átm \cdot d)$
RétegzettSZK	$\mathcal{O}(\|E\| + n \cdot átm)$	$\mathcal{O}(átm^2 d)$
HibridSZK	$\mathcal{O}(m\|E\| + \frac{n \cdot átm}{m})$	$\mathcal{O}(\frac{átm^2}{m} d)$

Tekintsük most egy súlyozott irányítatlan gráfra épülő aszinkron hálózatban a legrövidebb útvonalak megtalálásának problémáját. A feladat az, hogy a hálózat összes folyamatának meg kell határoznia és jelentenie kell a szülőjét és az i_0-tól vett távolságát egy, az i_0-ból indított legrövidebb úton. A szélességi keresés problémája a legrövidebb utak problémájának csak egy speciális esete, ahol az összes súly 1.

Emlékezzünk vissza, hogy a szinkron esetben a BellmanFord algoritmus megoldotta a legrövidebb utak problémáját. Bár ez az algoritmus szinkron, ki kell javítania a távolságbecslés hibáit. A következő kód segítségével a BellmanFord algoritmus aszinkron módon is futtatható. Ez az AszinkSZK kódjának természetes általánosítása. Az AszinkBellmanFord algoritmust használta az ARPANET forgalomirányításra 1969 és 1980 között.

15.6. automata. AszinkBellmanFord$_i$

Lenyomat:
Bemeneti:
`fogad`$(w)_{j,i}$, $w \in R^{\geq 0}$, $j \in$*szomszédok*
Kimeneti:
küld$(w)_{j,i}$, $w \in R^{\geq 0}$, $j \in$*szomszédok*

Állapotok:
táv $\in R^{\geq 0} \cup \{\infty\}$, kezdetben 0 ha $i = i_0$, különben ∞
szülő $\in$ *szülők*$\cup\{null\}$, kezdetben *null*
Minden $j \in$ *szülők* esetében
küld(j), egy FIFO sor $R^{\geq 0}$-beli elemekből,
kezdetben csak a 0 elemet tartalmazza, ha $i = i_0$,
egyébként üres

Átmenetek:
küld$(w)_{i,j}$
Előfeltétel:
w első *küld*(j)-n
Hatás:
töröljük *küld*(j) első elemét

fogad$(w)_{j,i}$
Hatás:
if $w +$ *súly*$(j,i) <$ *táv* **then**
táv $:= w +$ *súly*(j,i)
szülő $:= j$
for all $k \in$ *szomszédok* $- \{j\}$ **do**
vegyük fel *táv*-ot küld(k)-ba

Taszkok:
Minden $j \in$ *szomszédok* esetében
{küld$(w)_{i,j} : w \in R^{\geq 0}$}

- -

15.20. tétel. . *Az* AszinkBellmanFord *algoritmus bármely pártatlan végrehajtási sorozata mellett a rendszer végül egy olyan állapotban stabilizálódik, melyben a szülőváltozók egy i_0 gyökerű legrövidebb utat reprezentálnak, és amelyben a távolságváltozók az i_0-tól vett távolságot helyesen tartalmazzák.*

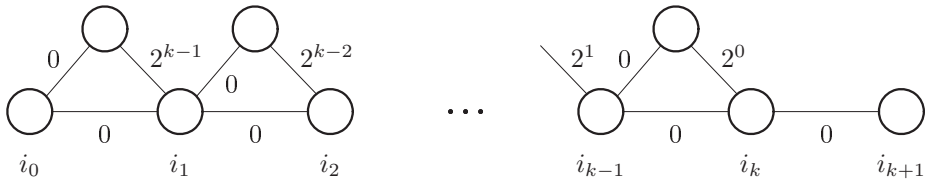
Az AszinkBellmanFord egy problémája, miként az AszinkSZK algoritmusé is, hogy egy folyamat nem tudja, hogy nem kell további javításokat eszközölnie. Így az algoritmus nem teljesen helyes, mivel sohasem állítja elő a kívánt kimenetet. Az AszinkBellmanFord algoritmus az AszinkSZK-hez hasonlóan kiterjeszthető konvergens nyugtaküldéssel, hogy a kívánt kimenetet megkapjuk.

Az AszinkBellmanFord bonyolultságelemzése érdekes, főleg azért, mert a legroszabb esetű üzenetszám és időbonyolultság rendkívül rossz: mindkettő n-től függően exponenciálisan nő. Összehasonlításképp emlékezzünk vissza, hogy a szinkron BellmanFord algoritmus csak $(n-1)|E|$ üzenetet és $n-1$ futamot használt, míg az AszinkSZK algoritmus (ismeretlen átmérővel és megállás nélkül) csak $\mathcal{O}(n|E|)$ üzenetet és $\mathcal{O}(\acute{a}tm \cdot n(\ell + d))$ időt használt.

15.21. tétel. . *Legyen $n \geq 4$ egy tetszőleges páros szám. Ekkor létezik egy olyan n csomópontú G súlyozott gráf, melyben az* AszinkBellmanFord *algoritmus legalább $\Omega(c^n)$ üzenetet küld, és valamely $c > 1$ konstansra a legrosszabb esetben legalább $\Omega(c^n d)$ időt vesz igénybe a stabilizáció. (c-t vehetjük $2^{\frac{1}{2}}$-nek.)*

Bizonyítás. Tételezzük fel, hogy a csatornák univerzálisan megbízható FIFO csatornák. Legyen $k = \frac{n-2}{2}$. Legyen G a 15.10. ábrán lerajzolt súlyozott gráf. A

G gráf legtöbb élének súlya 0. A nullától eltérő súlyú élek csak a jobbra lévő ferde élek, és súlyaik 2 csökkenő hatványai.



15.10.. ábra. Rosszul súlyozott gráf az AszinkBellmannFord algoritmusnál.

Azt állítjuk, hogy az AszinkBellmanFord algoritmus G gráfon vett végrehajtási sorozata során az i_k folyamat által felvett véges *táv* becslések pontosan a következő halmazban lévő értékek: $\{2^k-1, 2^k-2, \ldots, 3, 2, 1, 0\}$. Mindegyikük generálható egy i_0-ból i_k-ba menő adott úton haladó üzenetfolyam mentén. Valójában azt állítjuk, hogy i_k-nál ki lehet erőszakolni azt, hogy rendben felvegye mindezen becsléseket a legnagyobb értéktől a legkisebbig *ugyanabban a végrehajtási sorozatban* a következők szerint.

Tegyük fel, hogy a felső utakon az üzenetek nagyon gyorsan haladnak, így i_k a 2^k-1 becslést kapja. A következőkben az i_{k-1}-ből i_k-ba menő üzenet a lenti úton érkezik, s ezzel i_k új becslése 2^k-2 lesz. A következőkben az i_{k-2}-ből i_{k-1}-be menő üzenet a lenti úton érkezik, s ezzel i_{k-1} új becslése 2-vel csökken, 2^k-2 helyett 2^k-4 lesz. Ekkor az i_{k-1} folyamat mindkét úton elküldi a csökkentett becslését i_k-nak. Most tegyük fel újra, hogy a felső úton gyorsabban halad az üzenet. Tehát az i_k először a 2^k-3, majd a 2^k-4 becslést kapja.

A következőkben az i_{k-3}-ból i_{k-2}-be menő üzenet a lenti úton érkezik, s ezzel i_{k-2} csökkenti becslését. Ily módon folytatva elérhetjük, hogy az i_k folyamat rendre felvegye a $2^k-1, \ldots, 0$ becsléseket.

A rendszert lehetséges olyan módon üzemeltetni, hogy a folyamatok és a csatornák a $C_{i_k,i_{k+1}}$ csatornák kivételével nagyon gyorsan működjenek. Ez $C_{i_k,i_{k+1}}$-ban egy 2^k elemű üzenetsort eredményez, ami $\Omega(2^{\frac{n}{2}})$ üzenet, vagyis $\Omega(c^n)$ üzenet. Továbbá, ha mindezen üzenetek a maximális idő alatt kerülnek kézbesítésre, akkor az időbonyolultság $\Omega(c^n d)$, amit bizonyítani akartunk. □

A következőkben tekintsük az AszinkBellmanFord algoritmus bonyolultságának felső határait. Bármely $C_{i,j}$ csatornán küldött üzenetek száma a küldő i folyamat által kapott különböző becslések számától függ. Az ilyen becslések száma bizonyosan nem nagyobb, mint a gráfban az i_0-tól i-be vezető különböző egyszerű utak száma, ami $\mathcal{O}(n^n)$. (Valójában kisebb, de ezt a fejlesztést gyakorlatnak hagyjuk.) Így a teljes kommunikációs bonyolultság $\mathcal{O}(n^n|E|)$. Az időbonyolultság egy felső határa $\mathcal{O}(n^{n+1}(\ell+d))$ felhasználva az egy csatornán haladó üzenetek számának n^n határát.

Megjegyezzük, hogy az időkorlátok nagyban függnek a csatornákban kialakuló felhalmozódásoktól. Ha azt az egyszerűbb feltételezést használtuk volna, amit az elméleti kutatási irodalom is néha használ, hogy bármely üzenetnél a küldéstől a

fogadásig eltelt idő legfeljebb d (ha figyelmen kívül hagyjuk a helyi feldolgozási időt), akkor az AszinkBellmanFord időkorlátja egyszerűen $\mathcal{O}(nd)$ lenne. Ez természetesen nem egy valós elemzése az algoritmusnak.

15.5.. Minimális feszítőfa

A fejezet legutolsó alfejezetében visszatérünk egy tetszőlegesen összefüggő irányítatlan gráfon alapuló hálózat minimális súlyú feszítőfája megkonstruálásának problémájához. A 4.4. alfejezetben megadtuk a SzinkGHS algoritmust, mely a szinkron esetben megoldja ezt a problémát. Most bemutatjuk, hogy hogyan módosítható ez az algoritmus úgy, hogy aszinkron esetben is használható legyen. Az eredményül kapott algoritmus, melyet a felfedezőiről Gallagerről, Humbletről és Spiráról egyszerűen GHS-nek nevezzünk, az egyik legismertebb algoritmus az osztott számítástechnikai elméletben. Ez egy gondosan kidolgozott, összetett algoritmus, mely elég érdekes ahhoz, hogy alanyul szolgáljon algoritmus-ellenőrző módszerek tanulmányozásánál.

Azt javasoljuk, hogy ezen a ponton olvassa újra a 4.4. alfejezetet, mely a GHS algoritmus alapjául szolgáló elméletet és a GHS-hez szükséges számos ötletet felvonultató SzinkGHS algoritmust tartalmazza.

15.5.1.. A feladat megfogalmazása

Mint korábban, itt is feltételezzük, hogy az alapgráf $G = (V, E)$ nem irányított és összefüggő továbbá feltételezzük, hogy az élekhez súlyok vannak rendelve. Azt akarjuk, hogy a folyamatok működjenek együtt egy a G gráf egy minimális súlyú feszítőfájának (MFF) felépítésében. Azaz olyan fát keresünk G-ben, melyben az élsúlyok összege kisebb vagy egyenlő G bármely más feszítőfájának élsúlyai összegével.

Feltételezzük, hogy a folyamatok rendelkeznek UID-vel, és az egyes élek súlya ismert a szomszédos csomópontok számára. Egyetlen technikai feltételezéssel élünk: feltesszük, hogy az összes élsúly egyedi. A 4.4. alfejezet végén megadott érvelést felhasználva beláthatjuk, hogy ez az egyediség nem lényeges. Az azonos súllyal rendelkező élek közti kötés feloldható szomszédos folyamat UID használatával. Feltesszük, hogy a folyamatok csak helyi ismerettel rendelkeznek a gráfról, azaz nem ismerik a csomópontok számát vagy az átmérőt.

Feltesszük, hogy a folyamatok kezdetben *némák*, azaz kezdőállapotukban nem engedélyezettek a helyi vezérlésű műveletek. Feltesszük, hogy minden egyes folyamat rendelkezik egy `ébreszt` bemenő üzenettel, mellyel a környezet jelzi, hogy meg kell kezdeni egy MFF algoritmus végrehajtási sorozatát. Megengedjük, hogy a végrehajtási sorozat során bármennyi folyamat kaphasson `ébreszt`-t, így az algoritmusnak működnie kell tekintet nélkül a számítást kezdeményező folyamatok számára és a kezdeményezés időpontjára. Megjegyezzük, hogy csak a folyamatok kezdőállapotáról feltételezzük, hogy *némák*. Megengedjük, hogy egy folyamat bármely bemenet hatására felébredjen, legyen az akár egy `ébreszt` vagy egy üzenet

egy másik folyamattól.[1] Az algoritmus eredménye az MFF-et felépítő élhalmaz, azaz minden egyes folyamattól azt várjuk, hogy jelentse azon szomszédos éleit, melyek szerepelnek az MFF-ben.

15.5.2.. A szinkron algoritmus: áttekintés

Emlékezzünk vissza, hogy a SzinkGHS algoritmus az MFF 4.3. és a 4.4. lemmákban megadott két alapvető tulajdonságára épül. Ezek a tulajdonságok arra szolgálnak, hogy igazolják azt a stratégiát, mely szerint minden közbenső állapotban az algoritmus az MFF-ben szereplő élekből készít egy feszítőerdőt. Ekkor a feszítőerdő összetevőinek bármely tetszőleges részhalmaza függetlenül meghatározza a saját minimális súlyú kimenő élét (MSKÉ) tudván, hogy az összes ilyen élnek szerepelnie kell az MFF-ben.

A SzinkGHS algoritmus „szintekben" működik. A 0-dik szintű feszítőerdő egyedi csomópontokból áll, és nem tartalmaz éleket. Ha adott egy k-adik szintű feszítőerdő, az algoritmus létrehozza a $(k+1)$-edik szintű feszítőerdőt úgy, hogy lehetővé teszi az *összes* k szintű feszítőerdő-összetevő számára, hogy meghatározzák MSKÉ-jüket, majd ezen élek mentén összekapcsolja az összes összetevőt. Ebből következik, hogy minden k szintű összetevő legalább 2^k csomópontot tartalmaz.

Egy komponens MSKÉ-jének a meghatározását a komponens egy megkülönböztetett vezető csomópontja irányítja. Ennek az UID-je használatos a komponens azonosítására. A vezető üzenetszórás segítségével kéri a komponensélek mentén az MSKÉ-k meghatározását, majd a folyamat egy lekérdező portokoll segítségével megtudja, hogy mely szomszédai vannak ugyanabban, illetve különböző összetevőkben. Ezután a folyamat konvergens módon elküldi a vezetőnek az információit. A komponensek összevonása a vezetőtől az MSKÉ-vel szomszédos folyamat irányába történő kommunikációt foglalja magában. Körültekintő naplózással a folyamatok biztosíthatják, hogy a kommunikációs bonyolultság $\mathcal{O}(n \log n + |E|)$, a menetek száma pedig $\mathcal{O}(n \log n)$ legyen.

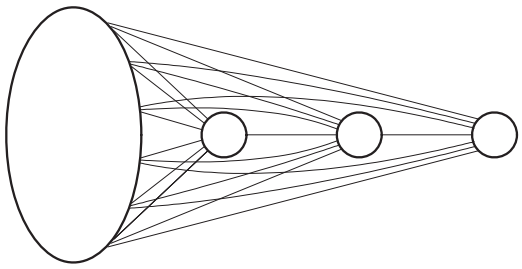
Ha megpróbáljuk futtatni a SzinkGHS algoritmust egy aszinkron hálózatban, néhány problémába ütközünk. Íme néhány ezek közül:

1. nehézség. A SzinkGHS algoritmusban, amikor az P_i folyamat megkérdezi a P_j szomszédját, hogy megtudja vajon P_j az aktuális feszítőerdőnek ugyanabban a komponensében van-e, tudja, hogy P_j a konstrukciós lépés ugyanazon szintjén van. Ebből következően, ha a P_j folyamat eltérő komponensazonosítóval rendelkezik, akkor P_j bizonyosan nincs ugyanabban a komponensben. Aszinkron esetben azonban előállhat olyan helyzet, hogy a P_j ugyanabban a komponensben van mint P_i, de még nem tud róla (mivel a legfrissebb komponensazonosítót küldő üzenetet még nem kapta meg).

2. nehézség. A SzinkGHS algoritmus az $\mathcal{O}(n \log n + |E|)$ üzenetköltséget azáltal éri el, hogy a szintek folyamatosan szinkronizálva vannak. Minden k szintű komponens legalább 2^k csomópontot tartalmaz, amiből az következik, hogy a

[1] Az `ébreszt` üzenettel kapcsolatos feltételezéseink itt eltérnek a 15.1.1. alfejezet végén tett feltételezéseinktől. Ott azt feltételeztük, hogy az `ébreszt` üzenetek az *összes* folyamathoz megérkeznek.

szintek száma legfeljebb $\log n$. Aszinkron esetben fellép a kiegyensúlyozatlan felépítés veszélye, mely sokkal több üzenetet eredményez. Egy komponens által az MSKÉ megtalálása céljából küldött üzenetek száma legalább a komponensbeli csomópontok számával arányos. El kell kerülnünk azt a helyzetet, amikor egy nagy komponens ismételten felfedezi, hogy az MSKÉ-je egy egycsomópontú komponensbe vezet (lásd 15.11. ábra), mivel ez $\Omega(n^2)$ üzenetet eredményez.



15.11.. ábra. Egy nagy komponens lehet, hogy csak csomópontonként nő.

3. nehézség. A SzinkGHS algoritmusban a szintek szinkronizáltak maradnak, míg az aszinkron esetben bizonyos összetevők magasabb szintig juthatnak mint mások. Nem világos, hogy a különböző szinteken lévő szomszédos komponensek által az MSKÉ-k megismerésére irányuló egyidejű keresések milyen interferenciát okozhatnak.

Ezek a nehézségek gondos megfontolásokat igényelnek a SzinkGHS algoritmus aszinkron hálózatra történő adaptálása során.

15.5.3.. A GHS algoritmus vázlata

A GHS algoritmus erősen követi a SzinkGHS algoritmust: ugyanazt a kommunikációs bonyolultságot – $\mathcal{O}(n \log n + |E|)$ – éri el, és a megfelelő időkorlát: $\mathcal{O}(n \log n(\ell + d))$.

A GHS algoritmusban a folyamatok maguk alakulnak komponensekké, melyek majd maguk is nagyobb komponensekké alakulnak. A kiinduló komponensek az egyedi csomópontok. Minden komponensnek van egy kitüntetett vezető csomópontja és egy feszítőfája, mely az MFF részgráfja.

Bármely komponens esetében a folyamatok egy algoritmusban működnek együtt a teljes komponens MSKÉ-jének megtalálása érdekében. Ez magában foglalja azt a vezetőből induló üzenetszórást, mely felkéri a komponensbeli folyamatokat arra, hogy határozzák meg azon saját minimális súlyú élüket, mely kivezet a komponensből. Ezután az összes ilyen élről szóló információ konvergensen visszakerül a vezetőhöz, ami meghatározza a teljes komponens MSKÉ-jét. Ez az MSKÉ kerül majd be az MFF-be.

Ha megvan az MSKÉ, ezen az élen a túloldalon lévő komponens felé egy speciális üzenet kerül elküldésre. A két komponens ezután egy új, nagyobb komponenssé alakul. Ebben az esetben az egész eljárás megismétlődik az új kompo-

nensre. Elég sok átalakulás után a gráf összes csomópontja bekerül az egyetlen végső komponensbe, amely feszítőfája a keresett MMF.

Számos olyan feladat van, melyet meg kell oldani ahhoz, hogy ez az algoritmus helyesen működjön. Először is, hogyan tudja meghatározni egy P_i folyamat, hogy mely éle vezet ki az aktuális komponensből? Bizonyosan megoldást kell találnunk a komponensek elnevezésére abból a célból, hogy két folyamat neveket tudjon használni annak eldöntésére, hogy azonos komponensben vannak-e. Sőt ez a kérdés bonyolultabb, mint ahogy az 1. nehézségben említettük, lehetséges, hogy egy eltérő komponensnévvel rendelkező szomszédos P_j folyamat valójában ugyanabban a komponensben van, mint a kérdező P_i folyamat, de a kommunikációs késleltetések miatt még nem tud erről. Annak biztosításához, hogy a P_j folyamat nem jelenti azt, hogy másik komponensben van, kivéve ha ismeri aktuális komponensnevét, valamilyen szinkronizációra van szükség.

A 2. nehézségben korábban leírt második feladat azzal kapcsolatos, hogy a kiegyensúlyozatlan összekombinálás során nagyon sok üzenet keletkezhet. Hogy kezeljük ezt a nehézséget, megpróbáljuk az összevonandó komponensek méretét nagyjából egyenlőnek tartani. Pontosabban, minden komponenshez rendelünk egy *szintet*, mint ahogy a SzinkGHS algoritmusban is tettük. Mint a SzinkGHS esetében, az összes kezdeti egycsomópontú komponenshez a 0 szintszámot rendeljük. Az összes k szintű komponens legalább 2^k csomóponttal rendelkezik. Egy $k+1$ szintű komponens csak pontosan két k szintű komponensből fog létrejönni, ezzel biztosítva a méretkorlátot. Ez a stratégia picit eltér a SzinkGHS-ben használttól: a SzinkGHS algoritmusban tetszőleges számú k szintű komponenst lehetett összevonni egy $k+1$ szintű komponenssé.

Mint kiderül, ezek a *szintek* nemcsak az összevonások kiegyensúlyozásánál hasznosak, hanem olyan azonosítási információkat is szolgáltatnak, melyek segítenek a folyamatoknak meghatározni, hogy azonos komponensben vannak-e.

A 3. nehézségben említett harmadik feladat, az a tény, hogy bizonyos komponensek magasabb szintekre juthatnak mint mások, és ez interferenciákhoz vezethet a különböző szinten lévő szomszédos komponensek egyidejű MSKÉ keresései során. Ezt az interferenciát valamilyen szinkronizációval meg kell előzni.

15.5.4.. A GHS algoritmus részletesebben

A GHS algoritmus két különböző módon vonja össze a komponenseket. Ezeket *összeolvasztásnak*,, illetve *elnyelésnek* nevezzük.

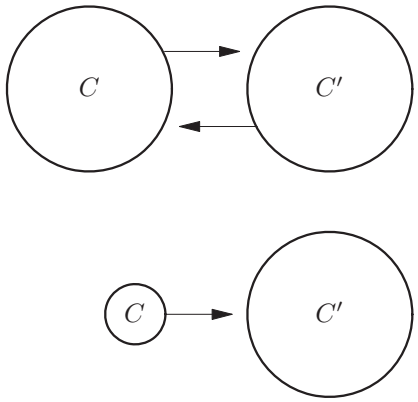
Összeolvasztás. Ez az összevonás művelet két olyan komponensre, C-re és C'-re vonatkozik, ahol $szint(C)$ a C *szintje* és egyenlő $szint(C')$-vel, valamint C-nek és C'-nek van közös MSKÉ-je. Az `összeolvasztás` eredménye egy olyan új komponens, mely C és C' összes csomópontját és élét tartalmazza, valamint a közös MSKÉ-t. Az új komponenshez a $szint{=}szint(C)+1$ szintszámot rendeljük.

Elnyelés. Ez az összevonás művelet két olyan C és C' komponensre vonatkozik, ahol $szint(C) < szint(C')$, valamint a C MSKÉ-je egy C'-beli csomópontba

vezet. Az `elnyelés` eredménye egy olyan új komponens, mely C és C' összes csomópontját és élét tartalmazza, és C MSKÉ-jét is. Az új komponenshez C' *szintjét* rendeljük szintszámként. Valójában jobb nem úgy elképzelni az `elnyelés` műveletet, mintha egy „új komponenst" hozna létre, hanem mint ami a létező C'-höz adja C-t.

Az `elnyelés` művelet akkor hasznos, ha valamely folyamatok lemaradnak mások mögött. Tegyük fel, hogy sok csomópont egy nagy C' magas *szintű* komponenssé alakul sok `összeolvasztás` művelettel, míg valamennyi kis komponens alacsonyabb *szinten* marad. Ha valamely kis C komponens felfedezi, hogy az MSKÉ-je C'-be vezet, akkor C' C' MSKÉ-je megismerése nélkül elnyelheti C-t. Ez egy olcsó művelet lesz.

E két összevonási stratégiát mutatja be nagyjából a 15.12. ábra. Megjegyezzük, hogy az `elnyelés` művelet során abból, hogy $szint(C) < szint(C')$ nem következik, hogy C kevesebb csomóponból állna mint C'. Az ábra csak a „tipikus" esetet mutatja be.



15.12.. ábra. Összeolvasztás és elnyelés.

Az `összeolvasztás` és az `elnyelés` műveletek a *szinteket* olyan módon manipulálják, hogy az kezeli a 2. nehézséget. Garantálják, hogy bármely k szintű komponens legalább 2^k csomópontból áll. Most bebizonyítjuk, hogy az `összeolvasztás` és az `elnyelés` műveletek elegendőek az összes komponens összevonásához a teljes gráf MFF-jévé.

15.22. lemma. . *Tegyük fel, hogy egy olyan kezdeti helyzetből indulunk ki, amelyben minden egyes komponens egyetlen csomópontból áll, a szintjük* 0*, és alkalmazzuk az* `összeolvasztás` *és* `elnyelés` *utasítások bármely megengedett véges sorozatát. Ezután az utasítássorozat után vagy egyetlen komponens marad, vagy valamely* `összeolvasztás` *vagy* `elnyelés` *művelet engedélyezett.*

Bizonyítás. Tegyük fel, hogy több komponens is van egy `összeolvasztás`-sokból és `elnyelés`-ekből álló utasítássorozat után. Megmutatjuk, hogy létezik

valamely alkalmazható művelet.

Tekintsük azt a G' „komponens irányított gráfot", melynek csomópontjai az aktuális komponensek, irányított élei pedig megfelelnek az aktuális MSKÉ-nek úgy, hogy abból a komponensből indulnak ki, melynek ezek az MSKÉ-i. A 4.4. lemma következtében G' bármely gyengén összefüggő részében létezik egy egyedi 2 hosszúságú ciklus. Ez azt jelenti, hogy van két C és C' komponens, melyek MSKÉ-i egymásra mutatnak. Könnyű belátni, hogy ebben az esetben a két MSKÉ szükség szerűen az eredeti G gráf ugyanazon éle.

Most azt állítjuk, hogy C és C' összevonható az `összeolvasztás` vagy az `elnyelés` művelettel. Ugyanis ha $szint(C) = szint(C')$, akkor az `összeolvasztás` művelet engedélyezett, ha pedig C és C' eltérő szinten van, akkor a kisebb *szintű* `elnyelhető` a nagyobb *szintű* által. □

Vizsgáljuk most meg részletesebben, hogy egy adott komponens esetében hogyan lehet megtalálni az MSKÉ-t. Ehhez a komponens minden egyes P_i folyamatának meg kell határoznia a saját minimális súlyú, komponensből kivezető $mské(i)$ élét (feltéve, hogy létezik). Ezután az összes folyamat elküldi az információt egy vezető folyamatnak, mely kiválasztja a végső minimális súlyú élet. Ehhez további mechanizmusokra van szükség. Kell először is egy olyan mechanizmus, mely az egyes komponensek vezető folyamatát kiválasztja. Másodsorban pedig szükség van egy olyan módszerre, mely eldönti egy folyamat egy éléről, hogy az a komponens kimenő éle-e.

E feladatok támogatása céljából minden 1 vagy magasabb szinten lévő komponensnél azonosítunk egy speciális élet, a *magélt*. Ezt az élet a komponenst létrehozó `összeolvasztás`, illetve `elnyelés` műveletek határozzák meg.

- `összeolvasztás` művelet után a *magél* a két eredeti komponens közös MSKÉ-je lesz.
- `elnyelés` művelet után a *magél* az eredeti komponensek közül a magasabb szintű magéle lesz.

Így egy komponens *magja* az az él, melyet az utolsó `összeolvasztás` használt a komponens létrehozásakor.

Egy 1-es vagy magasabb szinten lévő komponensnél *magélből* (technikailag a magél súlya) és a *szintből* álló párt használjuk a komponens azonosítására. Ez értelmes megoldás, mivel az élsúlyokról feltételeztük, hogy egyediek. A *magél* egyik végpontját, például a magasabb UID-jűt, pedig kijelöljük vezetőnek. Egy 0 szintű komponens esetében természetesen maga az egyedi csomópont a vezető.

Tegyük most fel, hogy a P_i folyamat meg szeretné határozni, vajon a szomszédos P_j folyamatba futó éle kimenő éle-e P_i aktuális komponensének. Ha P_j aktuális komponensazonosítója azonos P_i-vel, akkor a P_i folyamat biztos benne, hogy P_j ugyanabban a komponensben van, mint ő. Azonban, ha P_j komponensazonosítója eltér P_i-étől, akkor még mindig lehetséges, hogy P_i és P_j ugyanabban a komponensben van, de P_j még nem szerzett tudomást az aktuális komponensazonosítójáról. Egy speciális eset van, ami feloldható: ha P_j komponensazonosítója eltér P_i-étől, és P_j ismert *szintje* legalább olyan magas mint a P_i folyamaté, akkor bizonyosan nem lehet ugyanabban a komponensben. Ez azért van így, mert a

végrehajtási sorozat során egy csomópont *szintenként* legfeljebb csak egy komponensazonosítóval rendelkezhet, és mivel P_i aktívan keresi a kimenő éleit, biztos, hogy P_i komponensazonosítója helyesen ismert.

Így ha P_i és P_j azonos komponensazonosítóval rendelkezik, P_j azt válaszolja, hogy azonos komponensben vannak. Hasonlóan, ha P_i és P_j eltérő komponensazonosítóval rendelkezik, és P_j *szintje* legalább ugyanolyan magas, mint P_i-é, akkor P_j azt válaszolja, hogy elérő komponensben vannak. Az egyetlen fennmaradó eset az, amikor P_j *szintje* szigorúan kisebb, mint P_i-é. Ebben az esetben a P_j folyamat egyszerűen vár addig a válasszal, míg a saját szintje legalább olyan magas nem lesz mint P_i-é. Ez kezeli az 1. nehézséget.

Vegyük azonban észre, hogy most újra bizonyítani kell a végrehajtási sorozat előrehaladását, mivel ez az új késleltetés blokkolhatja az előrehaladást. Az a tény, hogy bizonyos folyamatok egy komponensen belül késleltethetődnek a saját minimális súlyú kimenő élük megtalálásában azt jelenti, hogy a komponens egészként késleltetheti az MSKÉ-jének meghatározását. Azt kell megvizsgálnunk, hogy ez a rendszert olyan állapotba viszi-e, melyben sem `összeolvasztás`, sem `elnyelés` művelet nem hajtható végre.

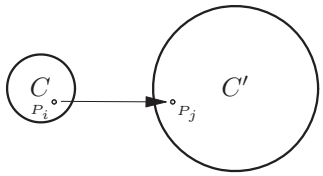
Ahhoz, hogy belássuk, ez nem fordulhat elő, alapvetően ugyanazt az érvelést használjuk, mint korábban, de most csak azokat a komponenseket tekintjük, melyek egy legalacsonyabb – mondjuk k – *szinten* vannak. Ezen komponensek összes folyamata szükségszerűen sikeresen határozza meg a minimális súlyú kimenő élét, következtetésképp ezek a komponensek sikeresen határozzák meg az MSKÉ-iket. Ha bármely k szintű komponens úgy találja, hogy az MSKÉ-je egy magasabb szintű komponensbe vezet, akkor egy `elnyelés` művelet lehetséges. Másrészt, ha minden k szintű komponens úgy találja, hogy az MSKÉ-je egy másik k szintű komponensbe vezet, akkor a 4.5. lemma következtében kell lennie egy 2 hosszúságú ciklusnak, mely k szintű komponensekből áll, tehát egy `összeolvasztás` művelet lehetséges. Így az újfajta késleltetés ellenére az algoritmus nem áll le, míg a teljes MFF-et meg nem találja.

Így láttuk, hogy az egyes folyamatok hogyan határozzák meg a minimális súlyú kimenő élüket (ha létezik). Ezután a fent leírt módon a komponens vezetője üzenetszórás és konvergens üzenetszórás segítségével meghatározza a komponens MSKÉ-jét a végső minimális súlyú élt választva.

A 3. nehézséget, a különböző szintű szomszédos komponensek egyidejű MSKÉ-keresésének problémáját, még mindig kezelni kell. Tekintsük át, hogy mi történik, ha egy alacsonyabb szintű C komponenst elnyel egy magasabb szintű C', mialatt C' keresi a saját MSKÉ-jét. Tegyük fel, hogy C MSKÉ-je C P_i csomópontját és C' P_j csomópontját köti össze. Tekintsük a 15.13. ábrát.

Két esetet kell megvizsgálnunk. Először is tegyük fel, hogy P_j még nem határozta meg a komponensből kivezető minimális súlyú kimenő élét, amikor az `elnyelés` bekövetkezett. Ebben az esetben az algoritmus C és C' összevont komponensének MSKÉ-jét keresi. Az a tény, hogy P_j még nem határozta meg $mské(j)$-t azt jelenti, hogy nem késő C-t bevonni a keresésbe.

Másrészt, tegyük fel, hogy a P_j folyamat már meghatározta $mské(j)$-t, amikor az `elnyelés` bekövetkezett. Ebben az esetben azt állítjuk, hogy $mské(j) \neq (i, j)$, azaz j minimális súlyú éle nem lehet azonos C MSKÉ-jével. Ez abból a tényből



15.13.. ábra. A C komponenst elnyeli a C', míg C' keresi a saját MSKÉ-jét.

következik, hogy *mské*(j) már meghatározásra került, s ebből következően egy legalább C'-vel azonos *szinten* lévő komponensbe vezet. (Technikai megjegyzés: az a tény, hogy *mské*(j) másik végpontja által j-nek megadott *szint* legalább akkora, mint C' *szintje*, maga után vonja, hogy még mindig legalább akkora, mivel egy folyamat által megismert *szint* nem csökkenhet.) Mivel azonban C-t elnyelte C', tudjuk, hogy *szint*(C) szigorúan kisebb, mint *szint*(C'). Tehát *mské*$(j) \neq (i, j)$, mint ahogy állítottuk. Ebből következik, hogy *mské*(j) súlya szigorúan kisebb, mint (i, j) súlya.

Ezután azt állítjuk, hogy az összevont komponens MSKÉ-je nem lehet szomszédos egy C-beli csomóponttal. Ez azért igaz, mivel (i, j) a C résznek MSKÉ-je, tehát nem lehet olyan C-ből kivezető él, melynek költsége kisebb, mint (i, j)-é, s ettől fogva nincs olyan C-ből kivezető él, melynek költsége kisebb, mint a már felfedezett *mské*(j). Így, ha *mské*(j) már megvan amikor az `elnyelés` bekövetkezett, C-ben már nem kell keresnie az algoritmusnak az összevont komponens MSKÉ-jét. Ez szerencse, mivel már túl késő is lehet keresni benne. A P_j folyamat már lehet, hogy jelentette a minimális súlyú kimenő élét, és a C' komponens lehet, hogy már azon van, hogy meghatározza a végső MSKÉ-t anélkül, hogy ismerné az újonnan elnyelt csomópontokat.

15.5.5.. Konkrét üzenetek

Most nagyobb részletességgel adjuk meg a GHS algoritmusban előforduló üzeneteket. Az üzenetek az alábbi típusokba sorolhatók:

- `kezdeményezés`. Egy `kezdeményezés` üzenet egy komponensen keresztülhaladó üzenetszórás, mely a vezetőből kiindulva végighalad a komponens feszítőfájának élein. A szokásos[2] esetben arra készteti a folyamatokat, hogy kezdjék meg a minimális súlyú kimenő élük (*mské*) megtalálását. Az üzenet hordozza a komponensazonosítót (*mag* és *szint*) is.
- `jelentés`. Egy `jelentés` üzenet konvergens módon küldi vissza a minimális súlyú kimenő élekről szóló információkat a vezetőnek.
- `teszt`. Egy P_i folyamat `teszt` üzenetet küld egy P_j szomszédjának abból a célból, hogy megállapítsa, P_j ugyanabban a komponensben van-e mint P_i. Ez része annak az eljárásnak, mellyel a P_i folyamat meghatározza a saját *mské*-jét.

[2] Van egy speciális eset, melyet később tárgyalunk.

- `elfogad` és `visszautasít`. Ezek az üzenetek a `teszt` üzenetre válaszként küldődnek. Megmondják a tesztelő csomópont számára, hogy a válaszoló csomópont egy másik (`elfogad`) vagy ugyanazon (`visszautasít`) komponensben van-e.
- `gyökérváltás`. Egy komponens vezetője egy `gyökérváltás` üzenetet küld annak a szomszédos komponensfolyamatnak, mely szomszédos a komponens MSKÉ-jével, miután az MSKÉ meghatározásra került. Arra használatos, hogy megmondjuk a folyamatnak, hogy kísérelje meg az összevonást az MSKÉ másik oldalán lévő komponenssel.
- `összekötés`. Egy C komponens MSKÉ-jén egy `összekötés` üzenet kerül küldésre, amikor a komponens megpróbál kombinálódni a másik komponenssel. Amikor ugyanazon él mindkét irányában végig fut az `összekötés` üzenet, akkor `összevonás` művelet kerül végrehajtásra. Ha egy *összekötés* üzenet egy, a küldőtől magasabb szintű folyamat felé kerül küldésre, akkor `elnyelés` művelet kerül végrehajtásra.

A *teszt-elfogad-visszautasít* protokollban a tesztelő P_i folyamatnak valamennyit naplóznia kell abból a célból, hogy a kommunikációs bonyolultságot alacsony szinten tartsa. Ez hasonló a SzinkGHS algoritmusban korábban leírt naplózáshoz. Nevezetesen, a P_i folyamat fenntart egy listát az érintkező élekről súly szerint növekvő sorrendben. Ez az érintkező éleket három kategóriába osztályozza.

- *ág* élek, melyekről már megállapítást nyert, hogy az MFF részei.
- `visszautasított` élek, melyekről már megállapítást nyert, hogy nem az MFF részei, mivel ugyanazon komponensben lévő csomópontokba vezetnek.
- *alap* él: minden más él. Ezek azok az élek, melyekről a P_i folyamat még nem állapította meg, hogy az MFF részei-e vagy sem.

Kezdetben minden él *alap*.

Amikor az P_i folyamat keresi a minimális súlyú kimenő élét, csak az *alap* éleken kell `teszt` üzenetet küldenie. A legkisebb súlyútól a legnagyobbig szekvenciálisan teszteli az *alap* éleket. Minden egyes *alap* élen az P_i folyamat a saját C komponensének azonosítóját (*mag* és *szin*)-t tartalmazó `teszt` üzenetet küld. Egy `teszt` üzenet P_j címzettje ellenőrzi, hogy a saját legutóbb megismert komponensazonosítója az P_i feladóéval azonos-e. Ha igen, akkor egy `visszautasít` üzenettel válaszol. Amikor az P_i egy `visszautasít` üzenetet kap, a kérdéses élet átteszi a *visszautasított* osztályba. Ha a P_j címzett *mag*-ja eltér a P_i-étől és a *szint*-je nem nagyobb a P_i-étől, a P_j egy `elfogad` üzenettel válaszol. (Ennek hatására P_i nem sorolja másik osztályba az élet.) Végül, ha a P_j címzett *magja* eltér a P_i-étől és a *szintje* szigorúan kisebb P_i-étől, a P_j folyamat egyszerűen elhalasztja a választ addig, amíg a korábbi szabályoknak megfelelően nem tud `visszautasít` vagy `elfogad` választ küldeni.

Megjegyezzük, hogy P_i kaphat `elfogad` választ egy (i, j) élre, de (i, j) ekkor nem feltétlenül a teljes C komponens MSKÉ-je. Ebben az esetben ugyanazon (i, j) él a további keresésekben újra tesztelésre kerül. Az P_i folyamat akkor sorol egy élt az *ág* osztályba, ha felfedezi, hogy az az MFF része. Például akkor, amikor egy P_i folyamat egy `gyökérváltás` üzenetet kap a kérdéses élre, vagy egy `összekötés`

üzenetet kap a kérdéses élen.

Amikor két `összekötés` üzenet egyetlen élen keresztezi egymást, egy `összevonás` művelet kerül végrehajtásra. Ekkor a közös él lesz az új *mag*, a *szint* eggyel nő, és a nagyobb UID-jű végpont lesz az új vezető. Az új vezető ezután üzenetszórással egy `kezdeményezés` üzenetet küld az `összevonás`-sal létrejött új komponens MSKÉ-jének meghatározására. Amikor egy `összekötés` üzenet érkezik egy alacsonyabb szintű komponensbeli folyamattól, akkor `elnyelés` művelet kerül végrehajtásra. A fogadó folyamat tudja, hogy megtalálta-e már a minimális súlyú kimenő élét, s evégből tudja, hogy kezdeményeznie kell-e egy keresést az újonnan elnyelt komponensben. Bármely esetben egy `kezdeményezés` üzenetet kell szórnia a megfelelő komponensbe az ott lévő folyamatok tájékoztatására a legfrissebb komponensazonosítóval kapcsolatban.[3]

Megjegyezzük, hogy minden egyes folyamat elő tudja állítani a kimenetét, amikor már nincs *alap* osztályú érintkező éle. A kimenet egyszerűen az *ág* osztályú élek halmaza.

15.23. tétel. . *A* GHS *algoritmus tetszőleges, összefüggő, irányítatlan, súlyozott gráf alapú hálózatban megoldja az MFF problémát.*

15.5.6.. Bonyolultságelemzés

A kommunikációs bonyolultság elemzése hasonló a SzinkGHS algoritmuséhoz, és ugyanazt az $\mathcal{O}(n \log n + |E|)$ eredményt adja. Az üzeneteket két osztályba soroljuk, az $\mathcal{O}(n \log n + |E|)$ kifejezésűek és az $\mathcal{O}(|E|)$ kifejezésűek osztályába. Az $\mathcal{O}(|E|)$ kifejezésűek közé tartoznak azok a `teszt` üzenetek, melyek visszautasításhoz vezetnek, plusz az összes élen előforduló `visszautasít` üzenetek. Ez összesen $\mathcal{O}(|E|)$, mert minden egyes él legfeljebb egyszer kerül visszautasításra. Miután P_i egy `visszautasít` üzenetet kapott egy élen, már sohasem teszteli azt.

Az összes többi üzenet: a `teszt-elfogad` pár, mely lehetővé teszi egy folyamat számára, hogy egy élt elfogadjon *minimális súlyú kimenő élének*, a `kezdeményezés` és a `jelentés` üzenetek, melyek üzenetszórásra és konvergens üzenetszórásra használhatók, a *gyökérváltoztatás* és az `összekötés` üzenetek, melyek, miután egy komponens meghatározta MSKÉ-jét, arra használhatók, hogy egy adott komponensben (azaz egy adott *mag* és *szint* mellett) megtalálják az MSKÉ-t. Ebben a feladatban egy komponens számára ezek az üzenetek csomópontokhoz rendelhetők oly módon, hogy minden egyes csomóponthoz legfeljebb egy fajta üzenet rendelődik. (Konkrétan, minden egyes folyamat legalább egy sikeres `teszt` üzenetet küld.) Így egy C komponenshez rendelhető üzenetek száma $\mathcal{O}(|C|)$, ahol $|C|$-vel jelöljük a C komponens csomópontjainak számát. Az összes üzenetszám tehát az alábbi kifejezéstől függ:

$$\sum_C |C|.$$

[3] Ez a korábban hivatkozott kivételes eset.

A komponenseket *szintjeik* szerint rendezve ezt a kifejezést átírhatjuk:

$$\sum_{k:0\leq k\leq \log n} \Big(\sum_{C:szint(C)=k} |C| \Big).$$

Minden k *szintre* a belső összeg legfeljebb n, mivel a k szinten egyetlen csomópont sem jelenik meg egynél több komponensben. Tehát ez a kifejezés legfeljebb ezzel egyenlő:

$$\sum_{0}^{\log n} n = \mathcal{O}(n \log n).$$

Ebből következik, hogy az algoritmus teljes kommunikációs bonyolultsága $\mathcal{O}(n \log n + |E|)$, ahogy állítottuk.

Ami az időbonyolultságot illeti, kényelmesen bevezethetünk egy előprotokollt, ami olyan gyorsan felébreszti az összes folyamatot, amilyen gyorsan csak lehet. Ezután k szerinti indukcióval belátható, hogy az össze folyamat számára legalább a k szint eléréséhez szükséges idő $\mathcal{O}(kn(\ell + d))$. Így a teljes idő $\mathcal{O}(n \log n(\ell + d))$.

Alsó határ.. Megjegyezzük, hogy a kommunikációs bonyolultságnak legalább bizonyos gráfok esetében $\Omega(n \log n)$-nek kell lennie. Például ha az MFF kommunikációs bonyolultsága gyűrűkben kisebb lenne, mint ez, akkor egy kommunikációhatékony algoritmust kombinálhatnánk az FF-Vezető algoritmussal, hogy egy olyan vezetőválasztó algoritmust kapjunk, melynek kommunikációs bonyolultsága szintén kisebb ennél. Ez azonban ellentmondana a 15.12. tételnek, miszerint $\Omega(n \log n)$ üzenet szükséges egy n méretű gyűrűben a vezetőválasztáshoz.

15.5.7.. A GHS algoritmus helyességének bizonyítása

A GHS algoritmus az első ebben a könyvben, amelyre még csak nem is körvonalazzuk a helyességi bizonyítást. Jó okunk van erre: jelenleg nem ismert egyszerű bizonyítás. Az algoritmusról legalább négy ízben változatos módszerekkel bebizonyították, hogy helyes, de egyik bizonyítás sem elég szépen rendezett ahhoz, hogy röviden körvonalazzuk.

Az egyik használható megoldás a szokásos, invariáns állítások módszere. Ebben az esetben ez meglehetősen nagy számú, az algoritmus által végrehajtott összes különböző feladatot leíró invariáns összegyűjtését vonja maga után. Például, vannak olyan invariánsok, amelyek az üzenetszórás és a konvergens üzenetszórás taszkjainak a helyes működését írják le, vannak olyanok, amik a `teszt-elfogad-visszautasít` protokollt írják le, és vannak olyanok, melyek a `gyökérváltás-összekötés` protokollt írják le. Mindezek az invariánsok egy óriási induktív állítással bizonyíthatók. Egy ilyen bizonyítás nagyszámú esetet és sok unalmas részletet foglal magában, de – elméletben – meglehetősen egyszerű feladat.

Azonban egy ilyen nyers erejű bizonyítás nem tűnik olyannak, ami kiaknázza az algoritmusban jelen levő modularitás előnyeit. Például, az algoritmus szétbonthatónak tűnik olyan különböző feladatokra, mint az üzenetszórás – konvergens üzenetszórás és a tesztelés, bár ez a szétbontás formálisan nincs kifejezve (pl. b/k

automataösszekapcsolás művelettel). Tehát nem világos, hogy hogyan lehetne különállóan bizonyítani az egyes feladatok helyességét, majd az eredményeket kombinálni.

A nyers erejű, invariáns állításokon alapuló bizonyítás nem aknázza ki az algoritmusban lévő magas szintű intuíciót sem. Megjegyezzük, hogy az algoritmusok tárgyalásának nagy része olyan magas szintű fogalmakat használ, mint a gráfok, komponensek, szintek és MSKÉ-k, ahelyett, hogy olyan alacsony szintű fogalmakat használna, mint az üzenetek és helyi változók. Úgy tűnik, hogy egy jó bizonyításnak, amennyire csak lehetséges, magas szintű fogalmakat kell használnia. Valójában egy másik működő megoldás az, amikor az algoritmusnak egy olyan magas szintű leírását adják meg, mint egy gráfokon, komponenseken stb. manipuláló automata, és ennek helyességét bizonyítják invariáns állításokkal. Ez után be lehet bizonyítani, hogy a részletes algoritmus helyesen szimulálja a magas szintű leírást. Az alacsony szintű és a magas szintű algoritmusok közötti formális megfeleltetés a *szimuláció reláció*, ami a 8.5.5. szakaszban került meghatározásra. A szimulációs bizonyításra példa a 10.9.4. szakaszban VÉGTELENJEGYKK kölcsönös kizárás algoritmus bizonyítása, valamint az EGYSZERŰSZINKR és a BIZTSZINKR szinkronizáló algoritmusok bizonyítása a 16. fejezetben. A szinkronizáló algoritmusok bizonyítása különösen szépen mutatja be, hogy egy összetett aszinkron hálózati algoritmus két módon bontható részekre: a különböző feladatokkal kapcsolatos érvelés szétbontása b/k automatát összekapcsolásával és a lehető legmagasabb absztrakciós szintű érvelést lehetővé tevő szimuláció relációk használatával.

A GHS algoritmus helyessége bizonyításának egy másik megközelítése az, hogy viselkedését formálisan megpróbáljuk kötni az algoritmus szinkron változatának, a SZINKGHS-nek a viselkedéséhez. Hétköznapian szólva, a megfelelés elég közelinek tűnik. Megjegyezzük, hogy az a kapcsolat nem lehet egy egyszerű szimuláció reláció, mert az aszinkron algoritmusnál a hálózat különböző részei nagymértékben kieshetnek az aktuális *szintekkel* meghatározott szinkronból. Bármely megfeleltetés esetében meg kell engedni a hálózat különböző részein zajló tevékenységek valamiféle újrarendezését.

A GHS algoritmus helyességének egy szépen szétbontott bizonyítását érdekes, nyílt problémának tekintjük. A modularitás elérése céljából elfogadható lenne az algoritmus kismértékű módosítása, ha a módosítások nem érintenék az algoritmus ötleteit és bonyolultságát.

A 16–22. fejezetekben különböző aszinkron hálózati algoritmusokat fogunk látni, melyeket különféle módszerekkel bontunk fel. Reméljük, hogy a GHS-hez hasonló algoritmusokkal kapcsolatos komplikációk meggyőzték az Olvasót, hogy ilyen felbontások megtalálása mennyire fontos.

15.5.8.. Egy egyszerűbb „szinkron” stratégia

Megjegyezzük, hogy a GHS algoritmusban számos olyan nehézség van, ami nem merül fel a SZINKGHS algoritmusban. Mindezek a nehézségek annak a ténynek a következményei, hogy a hálózat különböző részei nagymértékben kieshetnek az aktuális *szintek* által meghatározott szinkronból. Ezen nehézségek elkerülésének egyik módja, hogy a szomszédos folyamatok *szintjeit* egymáshoz közel tartva meg-

próbáljuk szimulálni a SzinkGHS-t, olyan pontosan, amennyire csak lehetséges.

EgyszerűMFF algoritmus (vázlatosan)

Ez az algoritmus szintén komponensek kombinálásán alapul, ahol a komponensek egy hozzájuk rendelt *szinttel* rendelkeznek. Csak a kezdeti komponensek az egyedi csomópontok 0 *szinttel*. Most a k szintű komponensek csak $k+1$ szintű komponenssé vonhatók össze a SzinkGHS algoritmusban használt általános stratégiával.

Az egyes P_i folyamatok egy *helyi_szint* változóval rendelkeznek, amiben az P_i folyamat által ismert legutolsó *szint* értéket tárolják. A *helyi_szint* kezdetben 0, és amikor az P_i folyamat megtudja, hogy egy új, k *szintű* komponens tagja, akkor a *helyi_szintet* k-ra emeli.

A kulcsgondolat az, hogy egy *helyi_szint* $= k$ értékkel rendelkező P_i folyamat addig nem próbál részt venni a k szintű komponense MSKÉ-jének megkeresésében, amíg a hálózat összes folyamata nem rendelkezik legalább k *helyi_szinttel*. Valóban ehhez drága globális szinkronizációra van szükség. Ténylegesen egy gyengébb helyi szinkronizáció is elégséges: az egyes folyamatok csak arra várnak, hogy az alapgráfbeli összes *szomszédjuk helyi_szint*-je legyen legalább k. Így ezt az összes folyamat maga felderítheti a *helyi_szint* minden egyes növekedése után a szomszédba vezető éleken küldött üzenettel.

Az EgyszerűMFF algoritmus időbonyolultságának felső határa ugyanaz, mint a GHS-é, nevezetesen $\mathcal{O}(n \log n(\ell + d))$, és az algoritmus természetesen sokkal egyszerűbb, mint a GHS. A kommunikációs bonyolultság azonban rosszabb a minden egyes szinten használt szinkronizációs üzenetek miatt. Ez most $\mathcal{O}(|E| \log n)$.

15.5.9.. A vezetőválasztás alkalmazása

Egy MFF algoritmus felhasználható tetszőleges, UID-kkel rendelkező, összefüggő, irányítatlan, súlyozott gráfban a vezetőválasztás problémájának megoldására. Nevezetesen, az MFF megtalálása után, a folyamatok részt vesznek a vezető az FF-Vezető protokollal történő megválasztásában.

Megjegyezzük, hogy a folyamatoknak nem kell tudniuk, hogy az MFF algoritmus mikor fejezi be a hálózatban a végrehajtási sorozatát. Elégséges, ha az egyes P_i folyamatok megvárják a helyi befejeződést, azaz amikor az MFF-be tartozó érintkező élek halmaza kimenetként előáll. Ha a P_i folyamat – mielőtt az MFF protokollhoz tartozó kimenetét előállítja – az FF-Vezető algoritmus részét képező üzenetet kap, akkor az üzenetet egyszerűen késlelteti addig, amíg nem végzett az MFF-fel. Az ötlet ugyanaz, mint a 15.1.1. szakaszban leírt, beérkező `ébreszt` üzenetek kezelésének általános stratégiájánál.

Ha az MMF megtalálásához a GHS algoritmust használják, egy vezető megválasztásához szükséges üzenetek összes száma $\mathcal{O}(n \log n + |E|)$, az összes idő pedig $\mathcal{O}(n \log n(\ell + d))$.

15.6.. Megjegyzések a fejezethez

Az AszinkLCR és a HS algoritmus aszinkron verziója, mint ezen algoritmusok szinkron verziója is, Le Lann ([191]), Chang–Roberts ([71]) és Hirschberg–Sinclair ([156]) dolgozataiból származik. A PetersonVezető algoritmust Peterson fejlesztette ki és a [239]-ben jelent meg. Dolev, Klawe és Rodeh [97] kifejlesztettek egy másik $\mathcal{O}(n \log n)$ kommunikációs bonyolultságú irányítatlan algoritmust. Higham és Przytycka [155] fedezte fel az aszinkron gyűrűbeli vezetőválasztás kommunikációs bonyolultságának jelenleg ismert legkisebb felső korlátját. A 15.1.4. alfejezet elején tekintett megfigyelések, melyek az mutatják be, hogy hogyan lehet a szinkron eseti kommunikációs bonyolultság alsó korlát eredményeit átvinni aszinkron esetre, Gafni [129] eredményei. Az aszinkron eset alsó korlátjának közvetlen bizonyítását Burns adta meg [61].

Afek és Gafni ([6]) fejlesztett ki bonyolultsági korlátokat teljes aszinkron hálózatokbeli vezetőválasztásra.

Az egyszerű feszítőfa, üzenetszórás és konvergens üzenetszórás algoritmusok kulcsötletei Segall ([258]) és Chang ([72]) dolgozataiból származnak. Az AszinkSZK és az AszinkBellmanFord algoritmusok Bellmann és Ford [43, 125] szekvenciális legrövidebb utak algoritmusán alapulnak. Az AszinkBellmanFord algoritmus alapvetően az az algoritmus, melyet az ARPANET használt útvonalkiválasztásra 1969 és 1980 között [223]. Az ebben a fejezetben leírt AszinkSZK és AszinkBellmanFord algoritmusokhoz megadott befejeződés protokoll Dijkstra és Scholtena „terjesztő számításokhoz" [92] megadott befejeződésjelzésén alapul. Ezt a munkát a 19.1. alfejezetben mutatjuk be. A RétegzettSZK és annak m-réteges változata Gallager [131] munkájának a hatására jött létre. Ezeket az eredményeket később Awerbuch és Gallager [33] fejlesztette tovább. Gabow [128] tervezett egy másik érdekes legrövidebb út algoritmust.

A GHS protokollt Gallager, Humblet és Spira fejlesztette ki [130]. A dolgozatukban szereplő kód kicsit más stílusú, mint az ebben a könyvben megadott előfeltétel/hatás kód. Az algoritmusnak ebben a könyvben szereplő stílushoz közelebb álló leírása Welch Ph.D. dolgozatában [287] megtalálható. A GHS algoritmus vagy variánsai helyességének bizonyítása több cikkben megjelent. Welch, Lamport és Lynch [288] szimulációs módszerekkel bizonyította a helyességet. Chou és Gafni [79] az algoritmus egy kissé módosított változatát bizonyította a szinkron algoritmussal való megfeleltetéssel. Stomp–de Roever [87] és Janssen–Zwiers [164] szintén adtak bizonyításokat. Awerbuch ([31]) kifejleszett egy $\mathcal{O}(nd)$ idejű és $\mathcal{O}(n \log n)$ üzenetszámú MFF algoritmust. Garay, Kutten és Peleg [132] kifejlesztett egy $\mathcal{O}((\acute{a}tm + \sqrt{n})d)$ idejű algoritmust. Awerbuch, Goldreich, Peleg és Vainish [34] bebizonyított egy alsó korlát eredményt, miszerint egy minimális feszítőfa felépítéséhez szükséges üzenetek száma $\Omega(|E|)$. Ez az eredmény korlátozott hosszúságú üzeneteket feltételez. Az EgyszerűMFF algoritmus Awerbuch eredménye.

Humblet [160] tervezett egy aszinkron osztott algoritmust a minimális feszítőfa irányított gráfú hálózatban való megtalálására.

15.7.. Gyakorlatok

15-1. Adjuk meg az AszinkLCR algoritmus egy olyan alternatív helyességbizonyítását, mely formálisan a szinkron LCR algoritmussal való kapcsolatán alapul.

15-2. Készítsünk a 15.1.1. szakaszban leírt AszinkLCR-módosításhoz egy olyan előfeltétel/hatás kódot, mely magában foglalja az `ébreszt` bemenetet és a *fogad* puffereket.

15-3. A HS algoritmus aszinkron változatához:
(a) írjunk előfeltétel/hatás kódot;
(b) a kódon alapulva bizonyítsuk be az algoritmus helyességét;
(c) elemezzük annak időbonyolultságát – feltételezve az egyes taszkok egyes folyamatainak szokásos ℓ felsőkorlátját, és a *legrégebbi* üzenet bármely csatornán tekintett d kézbesítési idejét;
(d) elemezzük annak időbonyolultságát – egy *tetszőleges* üzenet kézbesítési idejének d felső korlátját feltételezve, és figyelmen kívül hagyva a helyi feldolgozási időt.

15-4. Tekintsük a PetersonVezető algoritmust egy $n = 15$ csúcspontból álló gyűrűben, ahol a $P_1, \ldots, P_{16}$ folyamatok UID-jei rendre 25, 3, 6, 15, 19, 8, 7, 14, 4 22, 21, 18, 24, 1, 10, 23. Mely folyamat lesz vezetőnek megválasztva?

15-5. Tervezzük meg a PetersonVezető algoritmus egy, a 2. és 3. fejezetben leírt, szinkron hálózati modellre vonatkozó változatát. Az algoritmus folyamatai ismerhetik n-et. Törekedjünk arra, hogy az algoritmus olyan egyszerű legyen (leírni és megérteni), amennyire csak lehetséges, de őrizzük meg az irányítatlanságot és az $\mathcal{O}(n \log n)$ kommunikációs bonyolultságot. Elemezzük az algoritmus időbonyolultságát (a menetek számát).

15-6. Adjunk meg a PetersonVezető algoritmus időbonyolultságának $\mathcal{O}(n(\ell + d))$ felső korlátjára egy részletes bizonyítást.

15-7. Tervezzük meg a PetersonVezető vezetőválasztó algoritmus egy változatát kétirányú kommunikációval rendelkező gyűrűre. Az algoritmus új változatában a versengő UID-knek nem kell a gyűrűben forogniuk, maradhatnak az eredeti helyükön. Minden egyes fázisban az egyes folyamatok egyszerűen összegyűjtik az UID-ket a két aktív szomszédtól. Adjunk meg egy előfeltétel/hatás kódot az algoritmushoz. Elemezzük az üzenetek számát és az időbonyolultságot.

15-8. Bővítsük ki az AszinkLCR, az aszinkron HS és a PetersonVezető algoritmusokat úgy, hogy a nem-vezetők egy $\texttt{nem\_vezető}_i$ műveleten keresztül azt is bejelentik, hogy nem vezetők. Elemezzük a létrejött algoritmusok kommunikációs és időbonyolultságát.

15-9. A 15.11. tétel bizonyításvázlata alapján dolgozzuk ki a bizonyítás részleteit.

15-10. Adjunk meg egy alapos érvelést a 15.15. segédtétel bizonyítása induktív lépésében tett azon állításnak az igazolására, hogy az $\alpha_{L,M}$ utáni állapot *néma*.

15-11. Bővítsük ki a 15.12 tétel bizonyítását úgy, hogy alkalmazható legyen olyan gyűrűk esetében is, melyek mérete nem a 2 hatványa.

15-12. Tekintsük a vezetőválasztás problémáját kétirányú *vonalgráfokon* alapuló hálózatokban. Egy ilyen gráf $1, \ldots, n$-nel számozott n folyamatból áll, melyek egy vonalban vannak elhelyezve a szomszédok között kétirányú élekkel. Tegyük fel, hogy az egyes folyamatok a „jobb" és „bal" helyi nevekkel ismerik egymást. Tegyük fel, hogy az egyes folyamatok tudják, hogy végpontok-e vagy sem. Tegyük fel, hogy a folyamatok nem ismerik n-et.

(a) Adjunk meg ilyen hálózatokra egy vezetőválasztási algoritmust, mely kis számú üzenetet tartalmaz.

(b) Miért mond ellent ez az eredmény a 15.14. lemmában szereplő alsó korlátnak?

15-13. Tekintsük az OptMaxTerjed algoritmusnak a 15.12. alfejezetben leírt aszinkron szimulációját, melyben a folyamatok nem tudják, hogy mikor kell leállni.

(a) Írjunk egy előfeltétel/hatás kódot az aszinkron szimulációra.

(b) Egy tetszőleges G gráf és UID-hozzárendelés esetében hasonlítsuk össze a szimulációban küldött üzenetek maximális számát a szinkron OptMaxTerjed algoritmusban küldött üzenetek maximális számával.

15-14. Dolgozzuk ki a 15.16. tétel bizonyításának részleteit.

15-15. Adjunk meg az AszinkSzórásNyugtáz algoritmushoz egy gondos helyességbizonyítást.

15-16. Írjunk egy előfeltétel/hatás kódot az AszinkSzórásNyugtáz algoritmus egy módosításához, melyben az egyes folyamatok a `jelentés` művelet és a *nyugták* kiküldése után hulladékgyűjtést hajtanak végre az összes az algoritmusról szóló információval kapcsolatban. Bizonyítsuk be a helyességét, és elemezzük a bonyolultságát.

15-17. Tervezzünk egy algoritmust aszinkron hálózatokbeli üzenetszórásra és konvergens üzenetszórásra, melynél az időbonyolultság inkább a hálózat átmérőjétől, mint az összes csomópont számától függ.

15-18. Bővítsük ki a 15.3. alfejezet feszítőfa, üzenetszórás és konvergens üzenetszórás algoritmusait arra az esetre, amikor a hálózat erősen összegfüggő irányított gráfon alapul. Elemezzük az algoritmusai bonyolultságát.

15-19. Adjunk meg az AszinkSzórásNyugtáz algoritmus leírása után megadott vezetőválasztási stratégiához egy alapos leírást és helyességbizonyítást. Elemezzük az időbonyolultságot a következő két feltétel mellett: ha a csatorna legrégebbi üzenete kézbesítési idejének d a felső korlátja, ha egy csatornában egy tetszőleges üzenet kézbesítési idejének d a felső korlátja. Az utóbbi esetben a helyi feldolgozási időt figyelmen kívül hagyhatja.

15-20. Írjuk le részletesen egy olyan algoritmust, amely megengedi egy tetszőleges összefüggő, irányítatlan G gráfon alapuló aszinkron hálózat egy megkülönböztetett i_0 folyamata számára, hogy kiszámítsa G csomópontjainak számát. Vázolja fel a helyesség bizonyítását.

15-21. Dolgozzuk ki a 15.18. tétel bizonyításának részleteit.

15-22. Az AszinkSZK algoritmushoz

(a) Adjunk meg egy végrehajtási sorozatot, amely annyi üzenetet használ, amennyit csak kezelni tud. Próbáljuk elérni az adott $\mathcal{O}(n|E|)$ felső korlátot.

(b) Adjunk meg egy végrehajtási sorozatot, amely a kezelhető leghosszabb időt veszi igénybe a stabil állapot eléréséig. Próbálja elérni az adott $\mathcal{O}(\textit{átm} \cdot n(\ell + d))$ felső korlátot.

15-23. Írjunk egy előfeltétel/hatás kódot az AszinkSZK algoritmus egy módosításához, melyben a folyamatok *szülő* kimenetet állítanak elő egy nyugtázási protokoll segítségével. A hálózati gráf méretének vagy átmérőjének ismeretét ne feltételezze.

Bizonyítsuk be a protokolljának a helyességét, és elemezze a bonyolultságát. (*Tipp:* a kommunikációs bonyolultságnak azonosnak kell lennie az eredeti AszinkSZK algoritmuséval. Az időbonyolultság nagyobb lesz az AszinkFeszFa és a AszinkSzórásNyugtáz algoritmusoknál tárgyalt időzítési anomália miatt.)

15-24. Ismételjük meg a 15.23 feladatot, az AszinkSZK egy olyan módosítására, melyben az *átm* ismert, és amiben a folyamatok *szülő* kimenetet állítanak elő.

15-25. Írjunk egy előfeltétel/hatás kódot a RétegzettSZK algoritmusra, és bizonyítsa be annak helyességét.

15-26. Adjuk meg a HibridSZK algoritmus egy részletes leírását vagy előfeltétel/hatás kóddal vagy nagyon precíz magyarsággal. Bizonyítsuk be a helyességét.

15-27. Tervezzünk egy olyan hatékony algoritmust, amely megengedi egy tetszőleges összefüggő, irányítatlan G gráfon alapuló aszinkron hálózat egy megkülönböztetett i_0 folyamata számára, hogy meghatározza a maximális k távolságot az i_0-tól a hálózat legtávolabbi csomópontjáig. Elemezze az üzenetek számát és az időbonyolultságot.

15-28. Adjunk meg egy felső korlátot az AszinkBellmanFord legrövidebb utak algoritmus időbonyolultságára. A korlátnak a lehető legszorosabbnak kell lennie.

15-29. Írjunk egy előfeltétel/hatás kódot az AszinkBellmanFord algoritmus egy módosításához, melyben a folyamatok *szülő* és *távolság* kimenetet állítanak elő egy nyugtázási protokoll segítségével. Bizonyítsuk be a protokolljának a helyességét, és elemezzük a bonyolultságát.

15-30. Tervezzünk egy olyan algoritmust, mely i_0 fix forráscsomópontból a hálózat összes más csomópontjához megtalálja a legrövidebb utat. Az algoritmusnak az AszinkBellmanFord algoritmusnál sokkal jobb időkorláttal, mondjuk $\mathcal{O}(n(\ell + d))$-vel kell rendelkeznie.

15-31. Terjesszük ki a 15.4. alfejezetbeli szélességi keresés és legrövidebb utak algoritmusokat arra az esetre, ahol a hálózat erősen összefüggő irányított gráfon alapul. Elemezzük az algoritmus bonyolultságát.

15-32. Adjunk meg teljes előfeltétel/hatás kódot a GHS minimális feszítőfa algoritmushoz.

15-33. Tekintsük a GHS minimális feszítőfa algoritmust.

(a) Találjuk meg és bizonyítsuk annak az időnek a felső korlátját, ami az első folyamat felébresztésétől az utolsó folyamat eredményei bejelentéséig telik el. Feltételezhetjük egy elsődleges protokoll használatát az összes csomópont lehető leggyorsabb felébresztésére.

(b) Mennyire szoros az (a) részben bizonyított felső korlát? Azaz írjuk le az algoritmus azon sajátos végrehajtási sorozatát, mely a felső korlátjához lehető legközelebbi időt veszi igénybe.

15-34. Írjuk le a GHS egy végrehajtási sorozatát, melyben a $C_{i,j}$ csatornán a P_i folyamathoz egy `visszautasít` üzenet érkezik a P_i-ből korábban küldött `teszt` hatására akkor, amikor a P_i az (i,j) élt *ágnak* minősíti. Bizonyítsuk be, hogy az algoritmus ezt az esetet helyesen kezeli.

15-35. Tegyük fel, hogy a GHS algoritmus egy végrehajtási sorozatának egy pontján a C komponensbeli i folyamat egy `összekötés` üzenetet küld valamely olyan (i,j) élen, mely a C-vel azonos *szinten* lévő C' komponensbe vezet. Bizonyítsuk be, hogy végül a C vagy összeolvad C'-vel, vagy elnyelődik egy olyan komponensbe, mely tartalmazza C'-t.

15-36. *Kutatási kérdés.* Hasonlítsuk össze a GHS minimális feszítőfa algoritmus működését a SzinkGHS algoritmuséval. Például mi a kapcsolat a két esetben létrejövő komponensek között? (Egy ilyen kapcsolat felhasználható a GHS egy formális helyességbizonyításánál.)

15-37. *Kutatási kérdés.* Találjunk egy olyan szép, egyszerű bizonyítást a GHS algoritmus helyességére, mint amilyen e fejezetben és a [130]-ban szerepel. Ha az segít, az algoritmust kissé módosíthatja feltéve, hogy az alapvető algoritmusgondolatok megmaradnak, valamint az üzenet- és időbonyolultság nem változik.

15-38. Az EgyszerűMFF algoritmushoz

(a) Írjunk előfeltétel/hatás kódot.

(b) Bizonyítsuk a helyességét.

15-39. *Kutatási kérdés.* Találjunk MFF algoritmust $\mathcal{O}(\textit{átm} \cdot d)$ időbonyolultsággal, ahol az összes üzenet mérete $\mathcal{O}(\log n)$.

15-40. Adjunk meg a 15.5.9. szakaszban leírt vezetőválasztási stratégiához egy formális leírást egy b/k automata összekapcsolás formájában, amely előállítja az MFF-et és egy olyan b/k automatát, amely ezt az MFF-et használja egy vezető megválasztására. Gondosan írja le a két automatakészlet közti interakciókat, azonosítva azokat a műveleteket, melyek a két automatakészlet közti kommunikációra használatosak, pontosan megadva, hogy az egyes automatakészletek mit várnak el a másiktól.

15-41. Tekintsünk egy, a 15.12. feladatban leírt *vonalgráfokon* alapuló hálózatot. Azaz, egy ilyen gráf $1, \ldots, n$-nel számozott n folyamatból áll, melyek egy vonalban vannak elhelyezve a szomszédok között kétirányú élekkel. Az egyes folyamatok

a „jobb” és „bal” helyi nevekkel ismerik egymást. Az egyes folyamatok tudják, hogy végpontok-e vagy sem. A folyamatok nem ismerik n-et.

Tegyük fel, hogy minden egyes P_i folyamat rendelkezik egy nagy egészeket tároló v_i változóval, és ez bármely időpontban csak konstans számú ilyen értéket tud tárolni. Tervezzen egy algoritmust, ami rendezi az értékeket a folyamatok között, azaz hatására az összes P_i folyamat egy o_i kimenő értéket állít elő, ahol a kimenetek multihalmaza egyenlő a bemenetek multihalmazával, és $o_1 \leq o_2 \ldots \leq o_n$. Próbáljuk a leghatékonyabb algoritmust megtervezni mind az üzenetek száma, mind a végrehajtási idő szempontjából. Bizonyítsuk be állításaitnkat.

15-42. Tekintsünk egy aszinkron, összefüggő, irányítatlan tetszőleges topológiájú hálózatot, melyben minden egyes folyamat rendelkezik egy UID-vel. Tegyük fel, hogy minden P_i folyamat kezdetben kap egy v_i bemenő egész értéket. Tervezzen egy olyan algoritmust, amely hatására az egyes folyamatok visszaadják a hálózatban előforduló összes input összegét. Próbáljuk az üzenetek számában mért kommunikációs bonyolultságot alacsonyan tartani. Bizonyítsuk állításait.

15-43. Tekintsünk egy „banki rendszert”, melyben a hálózat minden egyes folyamata egy számmal rendelkezik, ami egy pénzösszeget képvisel. Az egyszerűség kedvéért feltesszük, hogy nincs külső betét vagy kivét, de a folyamatok között tetszőleges sokszor üzenetek utaznak, melyek az egyik helyről egy másikra „átutalt” pénzösszeget tartalmaznak. A csatornák a FIFO sorrendet követik.

Tervezzünk egy osztott hálózati algoritmust, mely lehetővé teszi az egyes folyamatok számára, hogy meghatározzák (azaz kimenetként előállítsák) a saját egyenlegüket úgy, hogy az egyenlegek teljes összege a rendszerben lévő pénz helyes összege legyen. Tételezze fel, hogy az algoritmus végrehajtási sorozatát egy kívülről jövő jelzés váltja ki, mely a rendszer egy vagy több pontján állhat elő. (Ezek a jelzések bármikor bekövetkezhetnek, és bekövetkezhetnek különböző helyeken és különböző időpontokban.)

Az algoritmus nem állhat le, és nem késleltetheti „szükségtelenül” az átutalásokat. Adjunk meggyőző érvelést az algoritmus helyes működéséről.

15-44. Tervezzük meg a 4.5. alfejezetben leírt LubyMFH algoritmus egy olyan változatát, mely működik aszinkron hálózatokban. Fogalmazzunk meg pontos állítást azzal kapcsolatban, hogy mit garantál az algoritmusuk, és bizonyítsuk be az állítást.

16. fejezet

Szinkronizátorok

A 15. fejezetben több példát is mutattunk az osztott algoritmusok közvetlenül aszinkron hálózati modellen való programozására. Amint ez mostanra már világosan kiderült, ez a modell nagyon nehézkesen programozható közvetlenül, bizonytalan volta miatt. Ennek megfelelően olyan egyszerűbb modellek használatára van szükség, amelyek könnyebben programozhatók, és az ott írt programok átalakíthatók az általános aszinkron hálózati modellben használható programokká.

Korábban már bemutattunk két modellt, amelyek az aszinkron hálózati modellnél egyszerűbbek – a *szinkron hálózati modellt* és az *aszinkron közös memóriájú modellt* –, és ismertettünk számos, ezekhez a modellekhez megírt algoritmust. Ebben a fejezetben megmutatjuk, hogyan alakíthatók át a szinkron hálózati modellhez megírt algoritmusok az aszinkron hálózati modell algoritmusaivá, a 17. fejezetben pedig az aszinkron közös memóriájú algoritmusok aszinkron hálózati algoritmusokká való transzformációját ismertetjük. Ezekkel az transzformációkkal lehetővé válik, hogy a két egyszerűbb modell algoritmusai aszinkron hálózatokban is fussanak.

A szinkron hálózati algoritmusok aszinkron hálózati algoritmusokká való transzformációjának gondolatát néhány – a 15. fejezetben bemutatott – algoritmusnál már javasoltuk, nevezetesen a 15.2. alfejezetben a MaxTerjed eljárás szimulálásakor menetszámok megadásával üzenetekhez, illetve a 15.5.8. szakasz EgyszerűMFF algoritmusa esetében.

A szinkronról aszinkron hálózati algoritmussá való transzformáció stratégiája csak nem hibatűrő algoritmusok esetében működik. Egy ilyen transzformáció hibatűrő algoritmusokra ugyanis nem működhet, mivel mint azt a 21. fejezetben megmutatjuk, a hibatűrés számára nyújtott lehetőségek alapvetően mások a szinkron és aszinkron hálózatokban.

A szinkron hálózati modellről aszinkron hálózati modellre való transzformációt egy (helyi) *szinkronizátornak* nevezett rendszermodullal adjuk meg. Ezt követően bemutatjuk a szinkronizátor több osztott megvalósítását. Az összes megvalósítás magában foglalja a rendszer szinkronizálását *minden szinkronizálási menetben*; ez azért szükséges, mert a transzformációk úgy vannak tervezve, hogy tetszőleges szinkron algoritmusokra működjenek. A ritkább időközönkénti szinkronizálhatóság (mint például az EgyszerűMFF algoritmus esetében) az algorit-

mus azon speciális tulajdonságaitól függ, amelyek biztosítják a helyes működést abban az esetben is, ha megengedett a folyamat lépéseinek tetszőleges egymásba ágyazása két szinkronizálási pont között.

A bemutatott szinkronizátor-megvalósításaink az osztott algoritmusok moduláris felbontásának igen jó példáit adják. Számos algoritmusfelbontási technikát használunk, amelyek többségét a 8. fejezetben közöljük. Először a helyességet adjuk meg „globálisan" egy b/k automata segítségével. Ezután absztrakt módon definiálunk egy helyi szinkronizátort, majd megmutatjuk, hogy ez megvalósítja a globális specifikációt; ehhez az események részben rendezésén alapuló technikákra van szükség. Ezt követően több alternatív lehetőséget mutatunk a helyi szinkronizátor megvalósítására; ezek mindegyikének működését a 8.5.5. szakaszban található szimulációs eljárással ellenőrizhetjük. Ezen megvalósítások legtöbbje azonban jól ki tud használni további felbontási lépéseket. Emiatt definiálunk egy *biztos szinkronizátor* nevű másik rendszermodult, megmutatjuk, hogyan valósítható meg segítségével a helyi szinkronizátor, majd elkészítünk több osztott algoritmust a biztos szinkronizátor megvalósításaként. Mindez jól mutatja, hogy mennyire hatékonyak a felbontási módszerek a bonyolult osztott algoritmusok egyszerű leírásában (és helyességük bizonyításában).

A fejezet végén meghatározzuk egy szinkron hálózati algoritmus aszinkron hálózaton való futtatása időigényének alsó korlátját, amikor a szinkronizációs követelmények nagyon erősek.

16.1.. A feladat

Ebben az alfejezetben a szinkronizátor által megoldandó problémát ismertetjük. A kiindulópont egy szinkron hálózati modell egy irányítatlan $G = (V, E)$ gráf csúcspontjaiban futó n számú szinkron folyamattal, amelyek a gráf élein át küldött üzenetek útján kommunikálnak egymással. A modell 2. fejezetben ismertetett leírásában minden P_i folyamat egy állapotgépként van megadva üzenetgeneráló és átmeneti függvényekkel. Itt most eltérünk a korábbi eljárástól, és ezúttal minden P_i folyamatot egy U_i „felhasználói folyamat" b/k automataként reprezentálunk.[1]

Legyen M a szinkron rendszerben használt rögzített üzenetábécé. *Címkézett üzeneten* egy (m, i) párt értünk, ahol $m \in M$, és $1 \leq i \leq n$.

Az U_i felhasználói automata a szomszédjainak történő üzenetküldéshez $\texttt{felhasználó\_küld}(T, r)_i$ kimeneti műveletekkel rendelkezik, ahol T címkézett üzenetek egy halmaza, és $r \in \mathbb{N}^+$. Egy címkézett üzenet címkéje az üzenet célállomását jelzi, az r argumentum pedig a menet sorszámát. Ha az U_i-nek nincsenek küldendő üzenetei az r-edik menetben, akkor egy $\texttt{felhasználó\_küld}(\emptyset, r)_i$ műveletet hajt végre. A szomszédoktól érkező üzenetek fogadásához az U_i rendelkezik $\texttt{felhasználó\_fogad}(T, r)_i$ alakú bemeneti műveletekkel is, ahol T címkézett üzenetek egy halmaza, és $r \in \mathbb{N}^+$. A címke itt az üzenet forrását jelzi, az r argumentum pedig ismételten a menet sorszámát. Az U_i rendelkezhet még más

[1] Ezekre a folyamatokra itt „felhasználói folyamatokként" hivatkozunk, mivel azok a szinkronizátor rendszer felhasználói, amely az általunk tanulmányozott fő rendszerkomponens.

külső műveletekkel is, amelyek segítségével a külső világgal kommunikálhat. A 2. fejezetben ismertetett, állapotok alapján történő kódolás helyett a felhasználói automaták bemenetét és kimenetét most a bemeneti és kimeneti műveletek segítségével modellezzük.

16.1.1. példa. `felhasználó_küld` **és** `felhasználó_fogad` **műveletek**

Legyen $n = 4$. A $\texttt{felhasználó\_küld}(\{(m_1, 1), (m_2, 2)\}, 3)_4$ művelet azt jelenti, hogy a harmadik menetben az U_4 felhasználó elküldi az m_1 üzenetet az U_1 felhasználónak, az m_2 üzenetet az U_2 felhasználónak, és nem küld több üzenetet. A $\texttt{felhasználó\_fogad}(\{(m_1, 1), (m_2, 2)\}, 3)_4$ művelet pedig azt jelzi, hogy a harmadik menetben az U_4 felhasználó fogadja az m_1 üzenetet az U_1 felhasználótól, az m_2 üzenetet az U_2 felhasználótól, és nem kap több üzenetet.

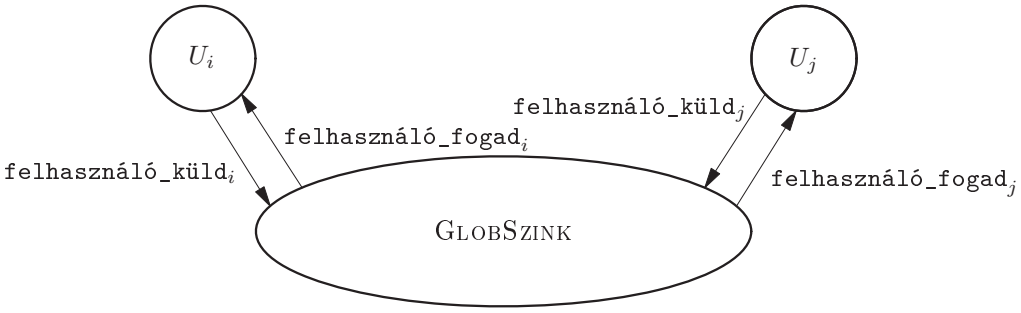
Az U_i kielégíti a *jólformáltság* feltételét, azaz a $\texttt{felhasználó\_küld}_i$ és a $\texttt{felhasználó\_fogad}_i$ műveletek váltakoznak úgy, hogy a műveletek sora egy $\texttt{felhasználó\_küld}_i$ művelettel kezdődik, és a műveleti párok a menetek sorrendjében követik egymást. Tehát ezek a műveletek egy alábbi alakú végtelen sorozat előtagjait jelentik:

$$\texttt{felhasználó\_küld}(T_1, 1)_i, \texttt{felhasználó\_fogad}(T_1', 1)_i,$$
$$\texttt{felhasználó\_küld}(T_2, 2)_i, \texttt{felhasználó\_fogad}(T_2', 2)_i,$$
$$\texttt{felhasználó\_küld}(T_3, 3)_i, \ldots .$$

Van még egy további feltétel is, amit az U_i-nek ki kell elégítenie – az élénkségi tulajdonság: minden jólformált pártatlan végrehajtási sorozatban az U_i-nek el kell végeznie egy $\texttt{felhasználó\_küld}_i$ műveletet minden olyan r-edik menetben, amikor minden korábbi menetben volt $\texttt{felhasználó\_fogad}_i$ művelet. Más szavakkal, a felhasználók mindaddig folytatják az üzenetek küldését, amíg a rendszer válaszol.

A rendszer további részét mint a GlobSzink *globális szinkronizátort* írjuk le. Ennek feladata, hogy minden menetben összegyűjtse a felhasználói automatáktól a menetben elvégzett `felhasználó_küld` műveletekben szereplő üzeneteket, és kézbesítse azokat a `felhasználó_fogad` műveletekben szereplő felhasználói automatákhoz. A szinkronizáció globális, azaz az egyes menetekben a `felhasználó_küld` események után és a `felhasználó_fogad` események előtt történik. A 16.1. ábrán a felhasználói és a GlobSzink automaták együttese, azaz a GlobSzink *rendszer* látható. Vegyük észre, hogy a `felhasználó_küld` műveletek a bemeneti, a `felhasználó_fogad` műveletek pedig a GlobSzink kimeneti műveleteit adják.

A GlobSzink egyszerűen leírható egy b/k automataként.



16.1.. ábra. A GlobSzink rendszer architektúrája.

16.1. automata. GlobSzink

Lenyomat:
Bemeneti:
`felhasználó_küld`$(T, r)_i$, T címkézett üzenetek egy halmaza, $r \in \mathbb{N}^+$, $1 \leq i \leq n$
Kimeneti:
`felhasználó_fogad`$(T, r)_i$, T címkézett üzenetek egy halmaza, $r \in \mathbb{N}^+$, $1 \leq i \leq n$

Állapotok:
tálca, $\{1, \ldots, n\} \times \mathbb{N}^+$-szal indexelt, címkézett üzenetek halmazait tartalmazó tömb, kezdetben minden eleme üres
felh_küld, *felh_fogad*, $\{1, \ldots, n\} \times \mathbb{N}^+$-szal indexelt, logikai értékeket tartalmazó tömbök, kezdetben minden elem *hamis*

Átmenetek:

`felhasználó_küld`$(T, r)_i$
Hatás:
felh_küld$(i, r) := igaz$
$\forall j \neq i$ **do**
tálca$(j, r) :=$ *tálca*$(j, r) \cup$
$\{(m, i) | (m, j) \in T\}$

`felhasználó_fogad`$(T, r)_i$
Előfeltétel:
$\forall j$ *felh_küld*$(j, r) = igaz$
felh_fogad$(i, r) = hamis$
$T =$ *tálca*(i, r)
Hatás:
felh_fogad$(i, r) := igaz$

Taszkok:
$\forall i, r$: {`felhasználó_fogad`$(T, r)_i : T$ címkézett üzenetek egy halmaza}

Ebben a kódban a *tálca*(i, r) tömb tárolja a szomszédaiból az U_i-hez küldött üzeneteket; ezek az üzenetek a feladójuk indexével vannak címkézve. A *felh_küld* és *felh_fogad* komponensek pusztán követik a `felhasználó_küld` és `felhasználó_fogad` események bekövetkeztét.

Nem nehéz belátni, hogy a 2. fejezet szinkron hálózati modelljében tárgyalt

összes algoritmus leírható ebben az új formában – az U_i felhasználói automaták és a GlobSzink automata kombinációjaként. Ennek megmutatását meghagyjuk gyakorlatnak (lásd 16-1. gyakorlat).

A szinkronizálási feladat a GlobSzink automata „megvalósítása" egy aszinkron hálózati algoritmussal, a G alapgráf minden i csúcsában egy P_i folyamattal, és egy $C_{i,j}$ megbízható FIFO küld/fogad csatorna megadásával a G minden (i,j) élének mindkét irányában. Ennek a megvalósításnak biztosítania kell, hogy az egyes U_i felhasználói automaták ne tudjanak különbséget tenni a megvalósítási rendszerben (ami a felhasználói automatákat plusz az osztott algoritmust jelenti) és a GlobSzink rendszerben való futás között. Vagyis azt szeretnénk biztosítani, hogy a megvalósítási rendszer minden α pártatlan végrehajtási sorozatához létezzen a specifikált rendszer egy α' pártatlan végrehajtási sorozata úgy, hogy minden i esetében α és α' megkülönböztethetetlen legyen U_i-re.[2]

Vegyük észre, hogy nem követeljük meg a különböző felhasználókhoz tartozó események relatív sorrendjének megőrzését, csak az egyes felhasználók ismeretét. Erre a megjegyzésre a 16.6. alfejezetben még visszatérünk.

16.2.. Helyi szinkronizátor

Az általunk leírt összes szinkronizátor-megvalósítás „lokális" abban az értelemben, hogy ezek csak a hálózat szomszédos, nem pedig tetszőleges elemei között szinkronizálnak. A csak lokális szinkronizáció használatának előnye, hogy így csökken a kommunikációs és időbonyolultság. Ebben szakaszban definiáljuk a GlobSzink egy helyi változatát, a HelyiSzink-et; az algoritmusokat a HelyiSzink megvalósításaiként mutatjuk be.

A HelyiSzink és a GlobSzink majdnem megegyezik. Az egyetlen különbség a `felhasználó_fogad` átmenetekben van, amelyek alakja most a következő.

16.2. automata. HelyiSzink

Átmenetek:

`felhasználó_fogad`$(T,r)_i$
  Előfeltétel:
    $\forall j \in szomszédok \cup \{i\}$
      $felh\_küld(j,r) = igaz$
    $felh\_fogad(i,r) = hamis$
    $T = tálca(i,r)$
  Hatás:
    $felh\_fogad(i,r) := igaz$

Vagyis a HelyiSzink-ben az r-edik menet üzenetei elküldhetők az U_i-nek, amint beérkeznek az U_i-től és annak szomszédaitól származó r-edik menetbeli

[2] Itt a „megkülönböztethetetlenség" 2.4. alfejezetben megadott definícióját használjuk, amely szerint a két végrehajtási sorozat U_i azonos végrehajtási sorozatait eredményezi.

üzenetek; nem szükséges megvárni a teljes hálózat összes felhasználójától beérkező üzeneteket.

16.1. lemma. . *Ha α a* HelyiSzink *rendszer (azaz felhasználók plusz* HelyiSzink*) egy tetszőleges pártatlan végrehajtási sorozata, akkor létezik a* GlobSzink *rendszernek egy olyan α' pártatlan végrehajtási sorozata, amely megkülönböztethetetlen az α-tól minden U_i-re.*

A bizonyításhoz nem használhatjuk azokat a szimulációs technikákat, amelyeket például a 10.9. alfejezet JegyKK algoritmusának bizonyításánál használtunk. Ennek az az oka, hogy a különböző csúcsokban végrehajtott külső műveletek relatív sorrendje időnként eltér a két rendszerben. Emiatt inkább egy olyan eljárást használunk, ami az események *részben rendezésén* alapul.

Bizonyításvázlat. Jelölje sorrendben L és G a HelyiSzink és a GlobSzink rendszert kis módosítással, a felhasználói automaták belső műveleteit kimenetként újraosztályozva. (Vagyis a rendszerek külső műveletei pontosan a felhasználói automaták összes műveletei lesznek.) L bizonyos eseményei más eseményektől „függenek": egy `felhasználó_fogad` eseményfügg az adott menetbeli, ugyanazon vagy szomszédos csúcsokban bekövetkező `felhasználó_küld` eseményektől, továbbá egy felhasználói automata bármelyik eseménye függhet az ugyanazon automatán bekövetkezett bármelyik előző eseménytől. Ha β az L egy tetszőleges története, akkor definiáljuk a β-beli események halmazának egy $\rightarrow$ β-val jelölt, irreflexív részben rendezését a következőképpen. (Ez hasonló a 14.1.4. és 14.2.4. szakaszban definiált függőségi relációkhoz.) Ha π és ϕ két β-beli esemény, ahol π megelőzi ϕ-t, akkor azt mondjuk, hogy $\pi \rightarrow \beta\phi$, illetve hogy ϕ *függ* π-től, feltéve, hogy teljesül az alábbi feltételek egyike.

1. π és ϕ ugyanazon U_i felhasználó eseményei.
2. $\pi = \texttt{felhasználó\_küld}(T, r)_i$ és $\phi = \texttt{felhasználó\_fogad}(T', r)_j$, ahol $j \in \mathit{szomszédok}_i$.
3. π-t és ϕ-t az 1. és 2. típusú relációk egy sorozata (vagy lánca) kapcsolja össze.

Ezen relációk legfontosabb tulajdonságát mondja ki a következő segédtétel. Eszerint a $\rightarrow$ β relációk eleget megállapítanak a β pártatlan történetben lévő függőségekről ahhoz, hogy egy, a függőségeket megőrző átrendezés még pártatlan történetet biztosítson. (Ez az segédtétel hasonló a 14.1. és 14.3. tételhez.)

16.2. segédtétel. . *Ha β az L egy pártatlan története, γ pedig a β-beli események $\rightarrow$ β rendezést megőrző átrendezésével készített sorozat, akkor γ szintén pártatlan története az L-nek.*

A 16.2. segédtételt felhasználva, a lemma bizonyításához induljunk ki az L egy tetszőleges α pártatlan végrehajtási sorozatából, és legyen $\beta = \mathit{történet}(\alpha)$. A β eseményeit átrendezve, készítsünk el egy új γ történetet, amelyben a menetek globálisan vannak „sorbaállítva": ezt úgy érjük el, hogy egy adott r-edik menet összes `felhasználó_küld` eseményét ugyanazon r-edik menet `felhasználó_fogad` eseményei előtt soroljuk föl. Ez az új rendezési feltétel összhangban van a $\rightarrow$ β függőségi feltételeivel, mivel azok sohasem követelik meg a fordított sorrendet, még

tranzitív alkalmazásuk esetében sem. A 16.2. segédtétel értelmében γ ugyancsak L egy pártatlan története. Továbbá, mivel minden egyes r-edik menet minden `felhasználó_küld` eseménye megelőzi ugyanazon r-edik menet összes `felhasználó_fogad` eseményét, nem nehéz megmutatni, hogy γ a G egy története. A bizonyítás befejezéséhez kitöltjük a γ-beli állapotokat G egy végrehajtási sorozatához, a felhasználói állapotokat ugyanúgy töltve ki, mint α-ban. A kitöltés formálisan a b/k automata általános összekapcsolási tételeivel, nevezetesen a 8.4. és a 8.5. tételek segítségével adható meg. □

Az alábbiakban egyszerű példát mutatunk egy osztott algoritmusra, amely a HELYISZINK-et valósítja meg.

Az EgyszerűSzink algoritmus (vázlatosan)

Minden r-edik menetben, egy `felhasználó_küld`$(T,r)_i$ alakú bemenet fogadása után, az EgyszerűSzink$_i$ folyamat először üzenetet küld EgyszerűSzink$_j$ szomszédainak, amely az r-edik menetszámot és a T-ben fellelhető, összes U_i-ból U_j-be küldött üzenetet tartalmazza. Ha az EgyszerűSzink$_i$ minden szomszédjától fogadta az r-edik menetbeli üzenetet, előállít egy `felhasználó_fogad`$(T',r)_i$ kimenetet, ahol T' a feladóval címkézett, fogadott üzenetek halmazát jelöli.

Formálisan leírva, EgyszerűSzink$_i$ a következő automata.

16.3. automata. EgyszerűSzink$_i$

Lenyomat:

Bemeneti:
- `felhasználó_küld`$(T,r)_i$, T címkézett üzenetek egy halmaza, $r \in \mathbb{N}^+$
- `fogad`$(N,r)_{j,i}$, N üzenetek egy halmaza, $r \in \mathbb{N}^+$, $j \in$ *szomszédok*

Kimeneti:
- `felhasználó_fogad`$(T,r)_i$, T címkézett üzenetek egy halmaza, $r \in \mathbb{N}^+$
- `küld`$(N,r)_{i,j}$, N üzenetek egy halmaza, $r \in \mathbb{N}^+$, $j \in$ *szomszédok*

Állapotok:

felh_küld, *felh_fogad*, $\mathbb{N}^+$-szal indexelt, logikai értékeket tartalmazó vektorok, kezdetben minden elemük *hamis*

pkt_küld, *pkt_fogad*, *szomszédok* $\times \mathbb{N}^+$-szal indexelt, logikai értékeket tartalmazó tömbök, kezdetben minden elemük *hamis*

kimenő, *szomszédok* $\times \mathbb{N}^+$-szal indexelt, üzeneteket tartalmazó tömb, kezdetben minden eleme üres

bejövő, $\mathbb{N}^+$-szal indexelt, címkézett üzeneteket tartalmazó vektor, kezdetben minden eleme üres

Átmenetek:

`felhasználó_küld`$(T,r)_i$
Hatás:
$kimenő(j,r) := \{m|(m,j) \in T\}$ $\forall j \in$ *szomszédok* **do**
$felh\_küld(r) := igaz$

`küld`$(N,r)_{i,j}$
Előfeltétel:
$felh\_küld(r) = igaz$
$pkt\_küld(j,r) = hamis$
$N = kimenő(j,r)$
Hatás:
$pkt\_küld(j,r) := igaz$

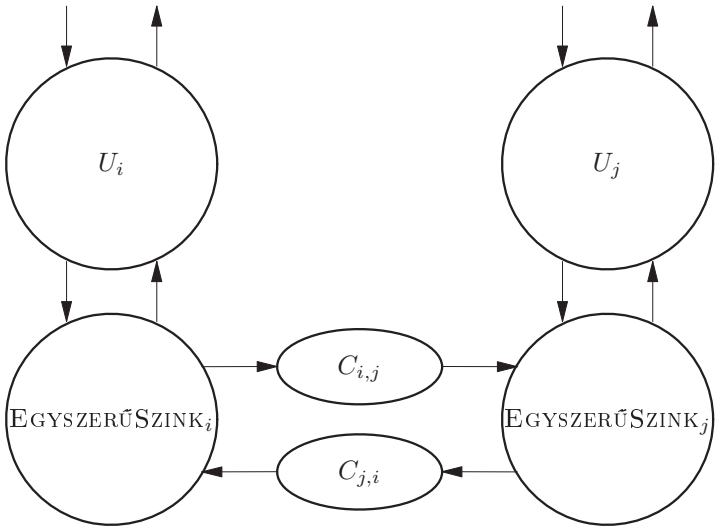
`fogad`$(N,r)_{j,i}$
Hatás:
$bejövő(r) := bejövő(r) \cup \{(m,j)|m \in N\}$
$pkt\_fogad(j,r) := igaz$

`felhasználó_fogad`$(T,r)_i$
Előfeltétel:
$felh\_küld(r) = igaz$
$\forall j \in$ *szomszédok* $pkt\_fogad(j,r) = igaz$
$T = bejövő(r)$
$felh\_fogad(r) = hamis$
Hatás:
$felh\_fogad(r) := igaz$

Taszkok:
$\forall r:$ $\{$`felhasználó_fogad`$(T,r)_i : T$ címkézett üzenetek egy halmaza$\}$
$\forall j \in$ *szomszédok* és $\forall r:$ $\{$`küld`$(N,r)_{i,j} : N$ üzenetek egy halmaza$\}$

- -

Az EgyszerűSzink rendszer az EgyszerűSzink$_i$ folyamatok, az élekhez kapcsolt $C_{i,j}$ megbízható FIFO küld/fogad csatornák és a felhasználók megadásával jön létre. Lásd 16.2. ábra.



16.2.. ábra. A EgyszerűSzink rendszer architektúrája.

16.3. lemma. . *Ha α az* EgyszerűSzink *rendszer egy tetszőleges pártatlan végrehajtási sorozata, akkor létezik a* HelyiSzink *rendszernek egy olyan α' pártatlan végrehajtási sorozata, amely megkülönböztethetetlen az α-tól minden U_i-re.*

Bizonyításvázlat. A 16.1. lemma bizonyításától eltérően, ezúttal nem kell átrendezni a különböző felhasználókhoz tartozó eseményeket, a megfeleltetés szimulációs módszerek segítségével mutatható meg. Jelölje sorrendben S és L az EgyszerűSzink és a HelyiSzink rendszert azzal a kis módosítással, hogy a külsőként osztályozott műveletek pontosan a felhasználói automaták összes műveletét jelentsék. (Vagyis a felhasználók belső műveletei kimenetekként vannak újraosztályozva, a `küld` és `fogad` műveletek pedig „rejtettek”, azaz belsőkként vannak újraosztályozva.) Ha s és u sorrendben S és L állapota, akkor definiáljuk az $(s,u) \in f$ relációt úgy, hogy az összes alábbi feltétel teljesüljön:

1. s-ben és u-ban minden felhasználói állapot megegyezik;

2. $u.felh\_küld(i,r) = s.felh\_küld(r)_i$;
3. $u.felh\_fogad(i,r) = s.felh\_fogad(r)_i$;
4. $u.tálca(i,r) = \bigcup_{j \neq i}\{(m,j) : m \in s.kimenő(i,r)_j\}$.

Annak megmutatásához, hogy f szimulációs reláció, szükségünk van az S-re vonatkozó következő invariáns állításra.

16.2.1. állítás. *Ha az* EgyszerűSzink *rendszer minden elérhető állapotában* $pkt\_fogad(j,r)_i = igaz$, *akkor*

1. $felh\_küld(r)_j = igaz$;
2. $\{m : (m,j) \in bejövő(r)_i\} = kimenő(i,r)_j$.

Ennek az invariáns állításnak a bizonyításához más köztes invariáns állításokat is használunk, így az átmenő üzenetek helyességét is. (A korábbiakhoz hasonlóan, az ilyen invariáns állításokban és azok bizonyításában feltételezzük, hogy a csatornák univerzális megbízható FIFO csatornák.) A 16.2.1. állítást felhasználva f szimulációs reláció voltának bizonyítása kézenfekvő; az egyetlen érdekes eset a `felhasználó_fogad`, amely bizonyításában a 16.2.1. állítást alkalmazza. Az invariáns és szimulációs állítások részletes bizonyítását meghagyjuk gyakorlatnak (lásd 16-4. gyakorlat).

A szimulációs reláció megléte azt eredményezi, hogy S minden története egyúttal L-nek is története. (Emlékezzünk vissza, hogy az ezekhez a történetekhez tartozó műveletek éppen a felhasználói automaták műveletei.) Nekünk azonban többre van szükségünk – nevezetesen arra, hogy S pártatlansági feltételeiből L pártatlansági feltételei következzenek. Belátjuk, hogy $pártatlan\_történetek(S) \subseteq pártatlan\_történetek(L)$, majd a felhasználói állapotok kitöltéséhez és a végrehajtási sorozatok közötti összefüggés megállapításához alkalmazzuk a b/k automaták általános összekapcsolási tételeit (8.4. és 8.5. tétel).

A pártatlan történet tartalmazási relációjának a megmutatásához azt az összefüggést használjuk, hogy egy szimulációs reláció nem csak történettartalmazást garantál – szoros megfeleltetést teremt a végrehajtási sorozatok között is, a 8.5.5. szakaszban tárgyalt módon. Legyen $\beta \in pártatlan\_történetek(S)$, és legyen α az S egy tetszőleges pártatlan végrehajtási sorozata úgy, hogy $\beta = történet(\alpha)$. Ekkor a 8.13. tétel értelmében van L-nek egy α' végrehajtási sorozata, amely megfelel az α-nak az f szerint. Azt állítjuk, hogy α' pártatlan végrehajtási sorozata az L-nek.

Kétféleképpen történhet meg az, hogy α' ne bizonyuljon pártatlannak. Először megtörténhet, hogy van egy olyan felhasználói taszk, amely ugyan engedélyezett α' egy pontjától, de ennek a taszknak nincsenek lépései α' azon pontjától. Ekkor a megfeleltetés miatt ugyanaz a felhasználói taszk engedélyezve van α egy pontjától, de a taszknak nincsenek lépései, ami az adott felhasználói taszkra vonatkozóan ellentmond α pártatlan voltának.

Másodszor létezhet olyan i és r, amelyre az r-edik menet $\texttt{felhasználó\_fogad}_i$ taszkja az α' egy adott pontjától engedélyezve van, de ennek a taszknak nincsenek lépései. Ez azt jelenti, hogy α' ezen pontjától kezdődően, a $felh\_küld(j,r) = igaz$ minden $j \in szomszédok_i \cup \{i\}$-re, és $felh\_fogad(i,r) = hamis$. Ekkor a

megfeleltetés miatt α megfelelő pontjától kezdve $felh\_küld(r)_j = igaz$ minden $j \in szomszédok_i \cup \{i\}$-re, és $felh\_fogad(r)_i = hamis$.

Felhasználjuk az alábbi állítást.

16.2.2. állítás. *Az* EgyszerűSzink *rendszer minden elérhető állapotában igaz a következő. Ha $pkt\_küld(i,r)_j = igaz$, akkor vagy tartalmaz egy üzenetet a $C_{j,i}$ csatorna, vagy $pkt\_fogad(j,r)_i = igaz$.*

Továbbá minden $j \in szomszédok_i$-re, a `küld` taszk r-edik menetbeli pártatlan volta miatt, a $pkt\_küld(j,r)_i$ *igaz* lesz α-ban. Továbbá a 16.2.2. állításból és a csatorna pártatlanságából következően $pkt\_fogad(j,r)_i$ *igaz* lesz. Az S-beli, r-edik menethez tartozó $\texttt{felhasználó\_fogad}_i$ taszk pártatlanságából adódóan ennek a taszknak van további lépése α-ban, és így a megfeleltetés miatt α'-ben is, ami ellentmondás. □

Vegyük észre, hogy a 16.3. lemma bizonyítása megmutatja a $pártatlan\_történetek(S) \subseteq pártatlan\_történetek(L)$ relációt is az egyes felhasználók megkülönböztethetetlensége mellett. A 16.1. és 16.3. lemmából következik a

16.4. tétel. . *Ha α az* EgyszerűSzink *rendszer egy tetszőleges pártatlan végrehajtási sorozata, akkor létezik a* GlobSzink *rendszernek egy olyan α' pártatlan végrehajtási sorozata, amely megkülönböztethetetlen az α-tól minden U_i-re.*

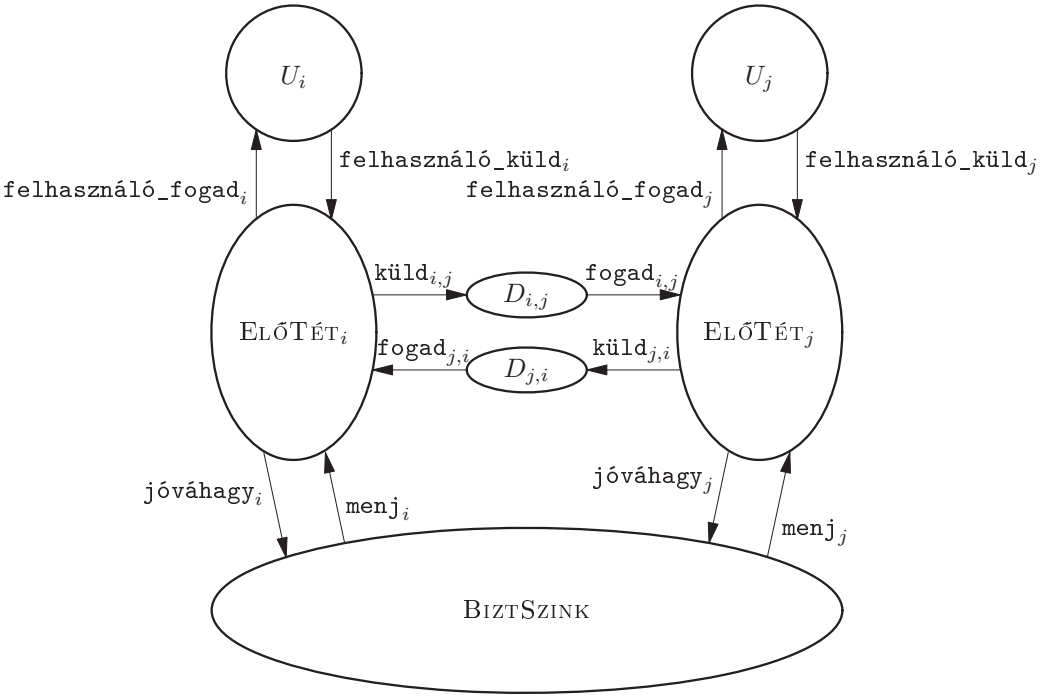
Bonyolultságelemzés.. Minden menethez $2|E|$ üzenet tartozik, a gráf minden éléhez mindkét irányban egy. Tegyük fel, hogy c időbeli felső korlátja annak, hogy egy $\texttt{felhasználó\_küld}_i$ esemény bekövetkezzen minden korábbi menetbeli $\texttt{felhasználó\_fogad}_i$ bekövetkezte után; ℓ felső korlát bármely folyamat bármely taszkjának végrehajtási idejére; és d időbeli felső korlátja a legrégebbi üzenet kézbesítésének bármelyik csatornán. Ekkor r menet szimulálásához szükséges idő legfeljebb $r(c + d + \mathcal{O}(\ell))$.

16.3.. Biztonságos szinkronizátor

Az EgyszerűSzink algoritmus időbonyolultságát nem lehet lényegesen csökkenteni, a kommunikációs bonyolultságát viszont igen. Nevezetesen, ha nincs üzenet U_i-ből az U_j szomszédhoz az r-edik menetben az alapul szolgáló szinkron algoritmusban, akkor elkerülhetünk egy r-edik menetbeli, a P_i folyamattól a P_j folyamathoz küldött üzenetet az aszinkron algoritmusban. Ezeket az üzeneteket azonban nem hagyhatjuk ki csak egyszerűen. Minden folyamatnak meg kell határoznia, hogy megkapta-e a szomszédaitól az r-edik menetre neki küldött összes üzenetet, mielőtt egy `felhasználó_fogad` kimenetet generálhatna az r-edik menetre. Ennek eldöntéséhez és a felhasználó üzeneteinek kézbesítéséhez az EgyszerűSzink algoritmus üzenetei nyújtanak segítséget. A kommunikáció egyszerűsítésének alapötlete ezen két funkció szétválasztása.

Ezért a HelyiSzink megvalósítását több részre bontjuk: egy „előrészre”, ahol minden csúcsban egy ElőTét áll, amely a szomszédos csúcsok ElőTét-jeivel

speciális $D_{i,j}$ csatornákon kommunikál, valamint egy a BIZTSZINK „biztonságos szinkronizátorra". Ez az új architektúra a 16.3. ábrán látható. Minden ELŐTÉT$_i$ feladata a `felhasználó_küld`$_i$ események során az U_i felhasználótól kapott üzenetek kézbesítése. Minden r-edik menetben, egy `felhasználó_küld`$_i$ vétele után, az ELŐTÉT$_i$ az r-edik menet kimenő üzeneteit „kimenő" tömbökbe rendezi. Ezután elküldi az összes nem üres *kimenő* tömb tartalmát a megfelelő j szomszédnak a $D_{i,j}$ csatornán keresztül, majd nyugtázásra vár a $D_{j,i}$ csatornán. Amikor az ELŐTÉT$_i$ minden üzenetére megkapta a nyugtázást, *biztonságosnak* nevezhető; ez azt jelenti, ELŐTÉT$_i$ összes üzenetét megkapták a megfelelő szomszédos ELŐTÉT-ek. Eközben az ELŐTÉT$_i$ összegyűjti és nyugtázza a szomszédos ELŐTÉT-ektől kapott üzeneteket.



16.3.. ábra. A HELYISZINK felbontása a BIZTSZINK segítségével.

Mikor engedélyezhető az ELŐTÉT$_i$ számára, hogy `felhasználó_fogad`$_i$ műveletet hajtson végre az r-edik menetben, vagyis hogy kézbesítse U_i-nek az r-edik menetben a szomszédaitól összegyűjtött összes üzenetet? Csak akkor teheti meg ezt, ha meggyőződött róla, hogy már valóban megkapta az r-edik menetbeli összes üzenetet, amit csak kaphat az r-edik menetben. Ennek értelmében, az ELŐTÉT$_i$-nek elég azt meghatároznia, hogy az összes vele szomszédos ELŐTÉT biztonságos az r-edik menetben, azaz az érintett szomszédok tudják, hogy r-edik menetbeli összes üzeneteiket a megfelelő ELŐTÉT automaták megkapták.

Így a BIZTSZINK biztonságos szinkronizátor automata feladata, hogy értesít-

sen minden ElőTét automatát, ha annak szomszédai biztonságosak. A BiztSzink ehhez a `jóváhagy` bemeneti műveleteket használja, amelyek az ElőTét automaták kimenetei, és amelyekkel az ElőTét automaták biztonságos voltukat jelzik a BiztSzink-nek. A BiztSzink $\texttt{menj}_i$ üzenetet küld az ElőTét_i-nek, miután megkapta a `jóváhagy`-ot az i összes szomszédjától és az i-től is.

Miután az ElőTét_i megkapja a $\texttt{menj}_i$ üzenetet, végrehajthat egy $\texttt{felhasználó\_fogad}_i$ műveletet. A szakasz hátralevő részében ezt a felbontást ismertetjük részletesen.

16.3.1.. ElőTét automata

16.4. automata. ElőTét_i

Lenyomat:

Bemeneti:

- $\texttt{felhasználó\_küld}(T, r)_i$, T címkézett üzenetek egy halmaza, $r \in \mathbb{N}^+$
- $\texttt{fogad}(\text{„}\texttt{üzen}\text{”}, N, r)_{j,i}$, N üzenetek egy halmaza, $r \in \mathbb{N}^+$, $j \in$ *szomszédok*
- $\texttt{fogad}(\text{„}\texttt{nyugtáz}\text{”}, r)_{j,i}$, $r \in \mathbb{N}^+$, $j \in$ *szomszédok*
- $\texttt{menj}(r)_i$, $r \in \mathbb{N}^+$

Kimenet:

- $\texttt{felhasználó\_fogad}(T, r)_i$, T címkézett üzenetek egy halmaza, $r \in \mathbb{N}^+$
- $\texttt{küld}(\text{„}\texttt{üzen}\text{”}, N, r)_{i,j}$, N üzenetek egy halmaza, $r \in \mathbb{N}^+$, $j \in$ *szomszédok*
- $\texttt{küld}(\text{„}\texttt{nyugtáz}\text{”}, r)_{i,j}$, $r \in \mathbb{N}^+$, $j \in$ *szomszédok*
- $\texttt{jóváhagy}(r)_i$, $r \in \mathbb{N}^+$

Állapotok:

felh_küld, *felh_fogad*, $\mathbb{N}^+$-szal indexelt, logikai értékeket tartalmazó vektorok, kezdetben minden elemük *hamis*

pkt_for, *pkt_küld*, *pkt_fogad*, *nyugtáz_fogad*, *szomszédok* $\times \mathbb{N}^+$-szal indexelt, logikai értékeket tartalmazó tömbök, kezdetben minden elemük *hamis*

nyugtáz_küld, *szomszédok* $\times \mathbb{N}^+$-szal indexelt, logikai értékeket tartalmazó tömb, kezdetben minden eleme *hamis*

kimenő, *szomszédok* $\times \mathbb{N}^+$-szal indexelt, üzeneteket tartalmazó tömb, kezdetben minden eleme üres

bejövő, $\mathbb{N}^+$-szal indexelt, címkézett üzeneteket tartalmazó vektor, kezdetben minden eleme üres

jóváhagy_adott, *menj_lát*, $\mathbb{N}^+$-szal indexelt, logikai értékeket tartalmazó vektorok, kezdetben minden elemük *hamis*

Átmenetek:

`felhasználó_küld`$(T,r)_i$
Hatás:
$felh\_küld(r) := igaz$
$\forall j \in szomszédok$,
if $\exists m, (m,j) \in T$ **do**
$kimenő(j,r) := \{m|(m,j) \in T\}$
$pkt\_for(j,r) := igaz$

küld(„üzen”, $N,r)_{i,j}$
Előfeltétel:
$pkt\_küld(j,r) = hamis$
$pkt\_for(j,r) = igaz$
$N = kimenő(j,r)$
Hatás:
$pkt\_küld(j,r) := igaz$

fogad(„nyugtáz”, $r)_{j,i}$
Hatás:
$nyugtáz\_fogad(j,r) := igaz$

fogad(„üzen”, $N,r)_{j,i}$
Hatás:
$bejövő(r) := bejövő(r)$
$\cup \{(m,j)|m \in N\}$
$pkt\_fogad(j,r) := igaz$

`felhasználó_fogad`$(T,r)_i$
Előfeltétel:
$menj\_lát(r) = igaz$
$T = bejövő(r)$
$felh\_fogad(r) = hamis$
Hatás:
$felh\_fogad(r) := igaz$

küld(„nyugtáz”, $r)_{i,j}$
Előfeltétel:
$pkt\_fogad(j,r) = igaz$
$nyugtáz\_küld(j,r) = hamis$
Hatás:
$nyugtáz\_küld(j,r) := igaz$

`menj`$(r)_i$
Hatás:
$menj\_lát(r) := igaz$

`jóváhagy`$(r)_i$
Előfeltétel:
`felhasználó_küld`$(r) = igaz$
$\forall j \in szomszédok$
if $pkt\_for(j,r) = igaz$ **then**
$nyugtáz\_fogad(j,r) = igaz$
$jóváhagy\_adott(r) = hamis$
Hatás:
$jóváhagy\_adott(r) := igaz$

Taszkok:

$\forall r$:
{`felhasználó_fogad`$(T,r)_i$: T címkézett üzenetek egy halmaza}
{`jóváhagy`$_S(r)_i$}
$\forall j$ és $\forall r$:
{küld(„üzen”, $N,r)_{i,j}$: N üzenetek egy halmaza}
{küld(„nyugtáz”, $r)_{i,j}$}

16.3.2.. Csatorna automaták

Minden ElőTét$_i$–ElőTét$_j$ előtét automatapár két, $D_{i,j}$, illetve $D_{j,i}$ csatorna-automata segítségével kommunikál. Ezek megbízható küld/fogad csatornák az i és a j, illetve a j és az i között, ahogyan ezt a 14.1.2. szakaszban definiáltuk.

16.3.3.. A biztonságos szinkronizátor

A BiztSzink biztonságos szinkronizátor összes feladata, hogy megvárja a ElőTét$_i$ és annak összes szomszédjától beérkező `jóváhagy` üzeneteket, majd végrehajtsa a `menj`$_i$ műveletet.

16.5. automata. BIZTSZINK

Lenyomat:

Bemeneti:
`jóváhagy`$(r)_i$, $r \in \mathbb{N}^+$, $1 \leq i \leq n$

Kimeneti:
`menj`$(r)_i$, $r \in \mathbb{N}^+$, $1 \leq i \leq n$

Állapotok:

jóváhagy_lát, *menj_adott*, $\{1, \ldots, n\} \times \mathbb{N}^+$-szal indexelt, logikai értékeket tartalmazó tömbök, kezdetben minden elemük *hamis*

Átmenetek:

`menj`$(r)_i$
Előfeltétel:
$\forall j \in szomszédok_i \cup \{i\}$
$jóváhagy\_lát(j, r) = igaz$
$menj\_adott(i, r) = hamis$
Hatás:
$menj\_adott(i, r) := igaz$

`jóváhagy`$(r)_i$
Hatás:
$jóváhagy\_lát(i, r) := igaz$

Taszkok:

$\forall i, r :$ {`menj`$(r)_i$}

16.3.4.. Helyesség

16.5. lemma. *Ha α a* BIZTSZINK *rendszer (azaz* ELŐTÉT*, csatorna,* BIZTSZINK *és felhasználói automaták, lásd 16.3. ábra) egy tetszőleges pártatlan végrehajtási sorozata, akkor létezik a* HELYISZINK *rendszernek egy olyan α' pártatlan végrehajtási sorozata, amely megkülönböztethetetlen az α-tól minden U_i-re.*

Bizonyításvázlat. Ezt az állítást a BIZTSZINK rendszerből a HELYISZINK rendszerbe megadott szimulációs reláció segítségével bizonyítjuk. Ugyanazt a módszert követjük, mint az EGYSZERŰSZINK algoritmus esetében a 16.3. lemma bizonyításában, így ugyanazt az f relációt használjuk. Ezúttal azonban az algoritmus bonyolultabb volta miatt a részletek leírása kicsit bonyolultabban történik. A szimulációs bizonyítás egyetlen édekes esete ismét csak a `felhasználó_fogad` művelet, amelyhez az alábbi invariáns állítást használjuk.

> **16.3.1. állítás.** *A* BIZTSZINK *rendszer minden elérhető állapotában teljesül, hogy ha $menj\_lát(r)_i = igaz$, akkor minden $j \in szomszédok_i$ esetében*
>
> 1. $felh\_küld(r)_j = igaz$;
> 2. $\{m : (m, j) \in bejövő(r)_i\} = kimenő(i, r)_j$.

Ennek az invariáns állításnak a bizonyításához további kiegészítő invariáns állításokra is szükség van, így például az alábbira.

16.3.2. állítás. *A* BIZTSZINK *rendszer minden elérhető állapotában a következő teljesül. Ha* $jóváhagy\_lát(j,r) = igaz$*, akkor* [3]

1. $felh\_küld(r)_j = igaz$;
2. $\{m : (m,j) \in bejövő(r)_i\} = kimenő(i,r)_j$ *minden* $i \in szomszédok_j$ *esetében.*

A további részletek megmutatását az olvasóra bízzuk (lásd 16-6. gyakorlat). □

A 16.1. és a 16.5. lemmából következik a

16.6. lemma. . *Ha* α *a* BIZTSZINK *rendszer egy tetszőleges pártatlan végrehajtási sorozata, akkor létezik a* GLOBSZINK *rendszernek egy olyan* α' *pártatlan végrehajtási sorozata, amely megkülönböztethetetlen az* α*-tól minden* U_i*-re.*

Tartozunk még a BIZTSZINK automata osztott algoritmussal való megvalósításával. A következő szakaszban több módszert is mutatunk erre. Szükség van a $D_{i,j}$ csatornák megvalósítására is az aktuális $C_{i,j}$ küld/fogad csatornák felhasználásával. Ezt a $C_{i,j}$ „multiplexelésével" érhetjük el, amelyek így nem csak a BIZTSZINK osztott megvalósításának csatornáit adják meg, hanem a $D_{i,j}$ megvalósításait is. A multiplexelési stratégiával a 14-6. gyakorlat foglalkozik.

16.4.. A biztonságos szinkronizátor megvalósításai

Ebben az alfejezetben a BIZTSZINK osztott algoritmusokkal való megvalósításait ismertetjük. Bemutatjuk a két fő, ALFA és BÉTA megvalósítást, és az ezek kombinációjával nyert GAMMA hibrid megvalósítást.

Emlékezzünk vissza, hogy a BIZTSZINK biztonságos szinkronizátor feladata, hogy minden menetben minden i-re megvárja az ELŐTÉT$_i$-től és annak összes szomszédjától beérkező `jóváhagy` üzeneteket, majd végrehajtsa a `menj`$_i$ műveletet.

16.4.1.. Az ALFA szinkronizátor

A BIZTSZINK szinkronizátor legközvetlenebb megvalósítása az ALFA szinkronizátor, amely a következőképpen működik.

ALFA szinkronizátor (vázlatosan)

Ha valamelyik r-edik menetre valamelyik ALFA$_i$ folyamat egy `jóváhagy`$_i$ üzenetet kap, elküldi ezt az információt az összes szomszédjának. Ha az ALFA$_i$ értesül róla, hogy az összes szomszédja megkapta a `jóváhagy`-ot az r-edik menetre, és maga az ALFA$_i$ is megkapta a `jóváhagy`-ot az r-edik menetre, akkor generál egy `menj`$_i$ kimenetet.

[3] Emlékezzünk rá, hogy a *jóváhagy_lát* a BIZTSZINK komponens állapotainak része.

Az olvasóra bízzuk az ALFA$_i$ folyamatok előfeltétel/hatás kódrészeinek megírását; ezen kódrészek struktúrája hasonló az EGYSZERŰSZINK$_i$ kódjához. A helyesség – biztonság és élénkség – egyszerűen megmutatható az ALFA rendszert (ALFA$_i$, ELŐTÉT, $D_{i,j}$ és felhasználói automaták) a BIZTSZINK rendszernek megfeleltető szimulációs technikák segítségével.[4] A következő állítást kapjuk.

16.7. tétel. . *Ha α az* ALFA *rendszer egy tetszőleges pártatlan végrehajtási sorozata, akkor létezik a* GLOBSZINK *rendszernek egy olyan α' pártatlan végrehajtási sorozata, amely megkülönböztethetetlen az α-tól minden U_i-re.*

Bonyolultságelemzés.. Megvizsgáljuk a teljes ALFA rendszer bonyolultságát. A kommunikáció bonyolultsága az alapul szolgáló szinkron algoritmus által küldött üzenetek számától függ: ha a szinkron algoritmus összesen m számú nemüres üzenetet küld r menetben, akkor az ALFA rendszer legfeljebb $2m+2r|E|$ üzenetet küld az r menet szimulálásához. A $2m$ tag az ELŐTÉT-ek által küldött `üzen` és `nyugtáz` üzenetekre, a $2r|E|$ pedig az ALFA rendszeren belül küldött üzenetekre vonatkozik. Ez utóbbi tag az összes menetben, az összes élen, az összes irányban küldhető üzenetek számát összegzi.

Ha c, ℓ és d ugyanúgy van definiálva, mint az EGYSZERŰSZINK algoritmusnál, akkor az r menet szimulációjához szükséges idő legfeljebb $r(c+3d+\mathcal{O}(\ell))$. (Figyelembe véve az alapul szolgáló csatornák ütközését is.) Így az ALFA rendszer kommunikációs és időbonyolultsága egyaránt rosszabb, mint az EGYSZERŰSZINK esetében.

Az EGYSZERŰSZINK-hez hasonlóan az ALFA-t is mérsékelt időbonyolultság, ám nagyfokú kommunikációs bonyolultság jellemzi. A következő szakaszban az EGYSZERŰSZINK egy másik megvalósítását közöljük, amelynek a kommunikációs bonyolultsága kisebb, azonban az időbonyolultság növekedésének rovására.

16.4.2.. A BÉTA szinkronizátor

A BÉTA szinkronizátor feltételezi, hogy a teljes G gráfnak van gyökeres fája – célszerűen olyan, amelynek a magassága kicsi.

> **BÉTA szinkronizátor (vázlatosan)**
> Az r-edik menetben minden folyamat konvergens üzenetszórással eljuttatja a `jóváhagy` információit a gyökérhez a feszítőfa élein keresztül. Miután a gyökér összegyűjtötte ezeket az információkat az összes folyamattól, `menj` kimenetek generálását engedélyező üzenetet szór ugyancsak a feszítőfa élein keresztül.

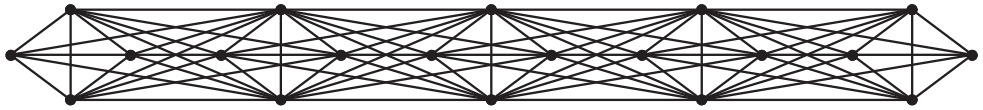
Ismét az olvasóra bízzuk a BÉTA rendszer BÉTA$_i$ folyamatai számára az előfeltétel/hatás kódrészek megírását (lásd 16-8. gyakorlat). Ugyanaz az ötlet használható, ami a 15.3. alfejezetben az üzenetszórásnál és a konvergens üzenetszórásnál.

[4] Ez a stratégia talán kevésbé tűnik modulárisnak, mivel ugyanazon felhasználói, ELŐTÉT és $D_{i,j}$ automaták jelennek meg mindkét rendszerben. Mindazonáltal ezek triviálisan kezelhetők, ha a szimulációs relációval nem módosítjuk őket. Egy másik megközelítés egy elvontabb (és általánosabb) környezetet formalizálna a BIZTSZINK automata számára.

A helyesség szintén egyszerűen megmutatható a BÉTA rendszert a BIZTSZINK rendszernek megfeleltető szimulációs technikákkal.

16.8. tétel. . *Ha α a* BÉTA *rendszer (*BÉTA$_i$, ELŐTÉT, $D_{i,j}$ *és felhasználói automaták) egy tetszőleges pártatlan végrehajtási sorozata, akkor létezik a* GLOBSZINK *rendszernek egy olyan α' pártatlan végrehajtási sorozata, amely megkülönböztethetetlen az α-tól minden U_i-re.*

Bonyolultságelemzés.. Ha az alapul szolgáló szinkron algoritmus összesen m számú nem *null* üzenetet küld r menetben, akkor a BÉTA rendszer összesen legfeljebb $2m+2rn$ üzenetet küld az r menet szimulálásához. A $2m$ ugyanazt jelenti, mint az ALFA esetében, míg a $2rn$ a szórt és összegyűjtött üzenetekre vonatkozik. Ha h a feszítőfa magasságának felső korlátja, akkor az r menet szimulációjához szükséges összes idő legfeljebb $r(c + 2d + \mathcal{O}(\ell) + 2h(d + \mathcal{O}(\ell)))$, vagy $r(c + \mathcal{O}(hd) + \mathcal{O}(h\ell))$.



16.4.. ábra. A G hálózati gráf.

16.4.3.. A GAMMA szinkronizátor

AZ ALFA és a BÉTA szinkronizátor elveit kombinálva tudjuk elkészíteni a hibrid GAMMA algoritmust, amely (a G gráf struktúrájától függően) egyidejűleg tud úgy működni, mint az idő szempontjából az ALFA, illetve a kommunikáció szempontjából a BÉTA rendszer.

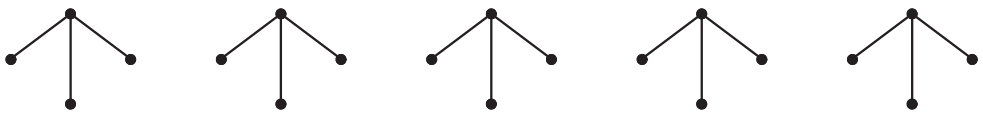
A GAMMA algoritmus egy feszítőerdőt tételez fel a G-hez, amelyben az erdő minden fája gyökeres. Minden fát *csoportnak* nevezünk, és egy C csoport csúcsainak halmazát *csúcsok*(C)-vel jelöljük. (Egy megfelelő feszítőerdő elkészítése önmagában is érdekes feladat, amit itt most nem ismertetünk.) A GAMMA a BÉTA egy változatát használja a csúcsok szinkronizálásához a csoportokon belül, illetve az ALFA egy változatát a csoportok szinkronizálásához.

Abban a szélsőséges helyzetben, amikor minden csoport egy csúcsot tartalmaz, a GAMMA megegyezik az ALFA-val, illetve amikor egyetlen csoport tartalmaz minden csúcsot, a GAMMA megegyezik a BÉTA-val. A köztes esetekben a GAMMA kommunikációs és időbonyolultsága az ALFA és a BÉTA ezen értékei közé esik.

16.4.1. példa. Csoportokra bontás

Tartalmazzon a G hálózati gráf p számú teljes gráfot, amelyek mindegyike k csúcsból áll. A teljes gráfok egyvonalban állnak, a szomszédos

párok csúcsai pedig össze vannak kötve. A 16.5. ábrán látható esetben $p = 5$ és $k = 4$. (Az ábrán egyes élek nem láthatók, mert más élek „alatt" futnak.) Vizsgáljuk meg most a 16.5. ábrán bemutatott G gráf csoportokra bontását.



16.5.. ábra. A G csoportokra bontása.

A felbontás minden C csoportja egy, a G egyik k-csúcsú teljes gráfjához tartozó fa lesz. A csoportok fáinak gyökerét a felül lévő csúcs adja. A GAMMA a BÉTA egy változatát használja a k-csúcsú fákon belüli szinkronizáláshoz, és az ALFA egy változatát a p fa közötti szinkronizáláshoz.

Mivel a GAMMA két algoritmus kombinációja, ezért a BIZTSZINK magasszintű felbontásával kezdjük; az eredményül kapott automatákat CSOPORTSZINK és ERDŐSZINK automatáknak nevezzük. Egyetlen ERDŐSZINK automatát és minden C_k csoporthoz egy CSOPORTSZINK$_k$ automatát adunk meg. Az architektúra a 16.6. ábrán látható.

A CSOPORTSZINK$_k$ automatának két feladata van minden r-edik menetben és minden C_k csoportra. Először, amint beérkeztek a $\texttt{jóváhagy}_i$ bemenetek a C_k minden i csúcsáról, egy $\texttt{csoport\_jóváhagy}_k$ kimenetet generál az ERDŐSZINK-nek. Másodszor (teljesen független feladatként), amint beérkezik egy $\texttt{csoport\_menj}_k$ bemenet az ERDŐSZINK-től, a CSOPORTSZINK$_k$ végrehajt egy $\texttt{menj}_i$ műveletet a C_k minden i csúcsára. A feladatoknak ez a kombinációja nagyjából megfelel a BÉTA tevékenységeinek. CSOPORTSZINK$_k$ leírása absztrakt automataként a következő.

16.6. automata. CSOPORTSZINK$_k$

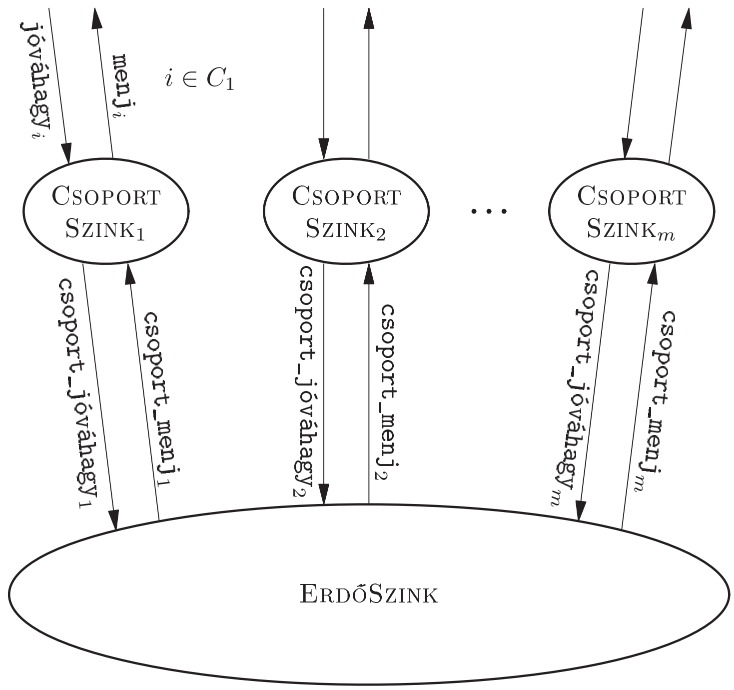
Lenyomat:

Bemeneti:	Kimeneti:
$\texttt{jóváhagy}(r)_i$, $r \in \mathbb{N}^+$, $i \in$ *csúcsok*(C_k)	$\texttt{menj}(r)_i$, $r \in \mathbb{N}^+$, $i \in$ *csúcsok*(C_k)
$\texttt{csoport\_menj}(r)_k$, $r \in \mathbb{N}^+$	$\texttt{csoport\_jóváhagy}(r)_k$, $r \in \mathbb{N}^+$

Állapotok:

jóváhagy_lát, *menj_adott*, *csúcsok*$(C_k) \times \mathbb{N}^+$-szal indexelt, logikai értékeket tartalmazó tömbök, kezdetben minden elemük *hamis*

csoport_jóváhagy_adott, *csoport_menj_lát*, $\mathbb{N}^+$-szal indexelt, logikai értékeket tartalmazó vektorok, kezdetben minden elemük *hamis*



16.6.. ábra. A BIZTSZINK felbontása a CSOPORTSZINK és az ERDŐSZINK automatákra.

Átmenetek:

`jóváhagy`$(r)_i$
 Hatás:
  $jóváhagy\_lát(i,r) := igaz$

`csoport_jóváhagy`$(r)_k$
 Előfeltétel:
  $\forall i \in csúcsok(C_k)$
   $jóváhagy\_lát(i,r) = igaz$
  $csoport\_jóváhagy\_adott(r) = hamis$
 Hatás:
  $csoport\_jóváhagy\_adott(r) := igaz$

`csoport_menj`$(r)_k$
 Hatás:
  $csoport\_menj\_lát(r) := igaz$

`menj`$(r)_i$
 Előfeltétel:
  $csoport\_menj\_lát(r) = igaz$
  $menj\_adott(i,r) = hamis$
 Hatás:
  $menj\_adott(i,r) := igaz$

Taszkok:

$\forall r :$ {`csoport_jóváhagy`$(r)_k$}
$\forall i, r :$ {`menj`$(r)_i$}

- -

Az ErdőSzink automata (a külső műveletek átnevezéséig) a G gráf G' *csoportgráfjának* biztonságos szinkronizátora, ahol G' csúcsait a G csoportjai jelentik, és pontosan akkor létezik él a G' gráf C_k és C_l csúcsai között, ha található él a G gráfban a C_k *valamelyik* pontjából a C_l *valamelyik* pontjába. A CsoportErdő rendszer definíció szerint a CsoportSzink, ErdőSzink, Előtét, $D_{i,j}$ és a felhasználói automatákból épül fel.

16.9. lemma. . *Ha α a* CsoportErdő *rendszer egy tetszőleges pártatlan végrehajtási sorozata, akkor létezik a* BiztSzink *rendszernek egy olyan α' pártatlan végrehajtási sorozata, amely megkülönböztethetetlen az α-tól minden U_i-re.*

Bizonyításvázlat. Használható szimulációs bizonyítás, de a változatosság kedvéért a rendszer működésén és a műveletek végrehajtásán alapuló érvelést vázolunk. A legszükségesebb azt megmutatni, hogy amikor $\texttt{menj}(r)_i$ következik be, akkor korábban már $\texttt{jóváhagy}(r)_j$-nek kellett bekövetkeznie minden $j \in \mathit{szomszédok}_i \cup \{i\}$-re. Két eset lehetséges.

1. i és j ugyanabban a C_k csoportban van (esetleg $i = j$).
   Ekkor a CsoportSzink_k kódjának megfelelően, a $\texttt{menj}(r)_i$ előtt már lennie kell $\texttt{csoport\_menj}(r)_k$-nak. Az ErdőSzink definíciója alapján, a $\texttt{csoport\_menj}(r)_k$ előtt pedig szerepelnie kell egy $\texttt{csoport\_jóváhagy}(r)_k$-nak. Ebből viszont az következik, hogy lennie kell egy megelőző $\texttt{jóváhagy}(r)_j$-nek, és ezzel bizonyítottuk az állítást.
2. i a C_k csoportba, j pedig a C_l csoportba esik, és $k \neq l$.
   Mivel $j \in \mathit{szomszédok}_i$, ezért a C_k és C_l csoportoknak szomszédosoknak kell lenniük a G' csoportgráfban (a csoportgráf szomszédos csoportjainak definíciója szerint). A korábbiakhoz hasonlóan, a $\texttt{menj}(r)_i$ előtt már lennie kell $\texttt{csoport\_menj}(r)_k$-nak. Az ErdőSzink definíciója szerint pedig ez előtt szerepelnie kell egy $\texttt{csoport\_jóváhagy}_l$-nek. Ugyanúgy, mint az előző esetben, ebből már következik, hogy lennie kell egy megelőző $\texttt{jóváhagy}(r)_j$-nek.

□

A Gamma szinkronizátor leírásának befejezéseképpen megmutatjuk, hogyan lehet megvalósítani az ErdőSzink és a CsoportSzink automatákat osztott algoritmusokkal. A CsoportSzink_k a Béta szinkronizátor C_k gyökeres fára alkalmazott változatának segítségével valósítható meg. Azaz a gyökér először összegyűjti a jóváhagy-okat, majd generál egy `csoport_jóváhagy` kimenetet. A gyökér kap egy `csoport_menj`-t is, ami után `menj` végrehajtására felszólító üzenetet küld (szór) a $\mathit{csúcsok}(C_k)$ minden csúcsának. (Ez a két tevékenység valójában két különálló automata segítségével formalizálható.)

Alkalmas átnevezésekkel a BiztSzink minden megvalósítása felhasználható az ErdőSzink megvalósításához; mi az Alfa szinkronizátort használjuk. Technikai problémát jelent, hogy az Alfa-t nem futtathatjuk közvetlenül az adott

osztott hálózaton, mivel az ALFA a szinkronizálandó elemeknek (ebben az esetben teljes csoportoknak) megfeleltetett folyamatokon fut, a szomszédos elemek (itt csoportok) közötti éleknek megfeleltetett csatornákon keresztül. Az adott modell csak a G csúcsainak és éleinek megfeleltetett folyamatok és csatornák használatatát engedélyezi. Mindazonáltal a szükséges folyamatok és csatornák megvalósítása nem túl bonyolult: a csoportokhoz tartozó folyamatokat a csoportok gyökércsúcsában futtatjuk, és közvetlen kommunikációt szimulálunk két szomszédos csoport folyamatai között, a két csoport gyökércsúcsai között kijelölt út segítségével. Léteznie kell ilyen útnak, mivel a csoportok összefüggőek és léteznek olyan csúcsok a két csoportban, amelyek szomszédosak G-ben. Ezeknek az utaknak a meghatározásához ismét szükség van előfeldolgozásra, amire most itt nem térünk ki. A `csoport_jóváhagy` és a `csoport_menj` műveletek a csoportok gyökércsúcsaiban lévő folyamatok belső műveleteiként vannak megvalósítva.

16.4.2. példa. Az ALFA megvalósítása

Tekintsünk a 16.4.1. példában szereplő G hálózati gráfot és csoportokra bontást. Erre a gráfra és felbontásra az egyes csoportokra vonatkozóan az ALFA folyamatot a csoportok fájának gyökerében futtatjuk (a 16.5. ábrán ez a felső csúcsokat jelenti). A szomszédos csoportok ALFA folyamatainak kommunikációja az alapul szolgáló G gráf (lásd 16.4. ábra) csoportjainak gyökerei között vezető él segítségével szimulálható.

A GAMMA teljes megvalósításában a G gráf i csúcsaihoz tartozó folyamatok formálisan három folyamatból tevődnek össze: ELŐTÉT$_i$, a CSOPORTSZINK megvalósításában lévő P_i folyamat és az ERDŐSZINK megvalósításában lévő P_i folyamat. Minden $C_{i,j}$ csatorna három csatornát valósít meg: a $D_{i,j}$-t, továbbá a CSOPORTSZINK, illetve ERDŐSZINK megvalósításában az i-ből j-be vezető csatornát. A GAMMA rendszert teljes megvalósításként definiálva szimulációs technikákat használhatunk a következő állítás bizonyításához.

16.10. tétel. . *Ha α a* GAMMA *rendszer egy tetszőleges pártatlan végrehajtási sorozata, akkor létezik a* GLOBSZINK *rendszernek egy olyan α' pártatlan végrehajtási sorozata, amely megkülönböztethetetlen az α-tól minden U_i-re.*

Ortogonális felbontások.. Érdeklődésre tarthat számot az a megfigyelés, hogy a teljes GAMMA rendszernek két természetes felbontása van. Az egyik logikai, a végzett funkciók (adatkommunikáció, csoportszinkronizáció és erdőszinkronizáció) szempontjából történhet. A másik térbeli, a teljes megvalósítás folyamatainak és csatornáinak szempontjából történhet. Ez a két felbontás az algoritmust felépítő egyszerű b/k automaták eltérő sorrendben történő csoportosítására vonatkozik. Mivel a csoportosítás művelete asszociatív, ezért a felbontás módjától függetlenül, végeredményképpen ugyanazt az algoritmust kapjuk.

Bonyolultságelemzés.. Legyen h a csoportfák maximális magassága, és legyen e' a gyökerek közötti kommunikációhoz használt utakban lévő élek teljes száma.

	Üzenetek	Idő
ALFA:	$\mathcal{O}(pk^2)$	$\mathcal{O}(d)$
BÉTA:	$\mathcal{O}(pk)$	$\mathcal{O}(pd)$
GAMMA:	$\mathcal{O}(pk)$	$\mathcal{O}(d)$

Ha az alapul szolgáló szinkron algoritmus összesen m számú nem *null* üzenetet küld r menetben, akkor a GAMMA rendszer összesen legfeljebb $2m+\mathcal{O}(r(n+e'))$ üzenetet küld. Az $\mathcal{O}(rn)$ tag a CSOPORTSZINK megvalósításában használt csoportfákon belül küldött üzenetekre vonatkozik. Az $\mathcal{O}(re')$ pedig az ERDŐSZINK megvalósításában levő gyökerek közötti üzenetekre vonatkozik. Az r menet szimulációjához szükséges idő $\mathcal{O}(r(c+\mathcal{O}(hd)+\mathcal{O}(h\ell)))$. Ha $n+e' << |E|$, akkor a GAMMA az ALFA-nál kevesebb üzenetet használ, továbbá ha a csoport feszítőfájának maximális magassága sokkal kisebb, mint a teljes hálózat feszítő fájának magassága, akkor a GAMMA kevesebb időt igényel, mint a BÉTA.

16.4.3. példa. Az ALFA, BÉTA és GAMMA bonyolultságának összehasonlítása

Tekintsünk ismét a 16.4.1. példában szereplő G hálózati gráfot és csoportokra bontást. Erre a gráfra és felbontásra összehasonlítjuk a bemutatott három biztonságos szinkronizátor igényeit. Az igényeket egy menetre számítjuk, és figyelmen kívül hagyjuk a felhasználók, ELŐTÉT-ek és $D_{i,j}$-k fellépő igényeit, amelyek ugyanazok mindhárom algoritmus esetében; eltekintünk továbbá a helyi feldolgozási időtől is. A BÉTA algoritmusra feltételezzük, hogy a használt fa a lehető legalacsonyabb, a magassága körülbelül p.

Ha p és k körülbelül egyenlő, akkor a GAMMA nagyságrendileg javítja mind az ALFA, mind a BÉTA algoritmust.

16.5.. Alkalmazások

Az előző alfejezetekben bemutatott szinkronizátor-algoritmusok egy hibamentes aszinkron hálózat számára lehetővé teszik bármely nem hibatűrő szinkron hálózati algoritmus megvalósítását. (A szinkronizátorok nem működnek olyan hibatűrő algoritmusokra, mint a 6. fejezetben közöltek.) Ebben az alfejezetben néhány példát adunk szinkronizátorokkal elkészített aszinkron algoritmusokra.

Emlékezzünk rá, hogy ebben a fejezetben csak nem irányított hálózatokkal foglalkozunk. A fejezet összes vizsgálatában figyelmen kívül hagyjuk a helyi feldolgozás lépéseinek időigényét.

16.5.1.. Vezetőválasztás

A szinkronizátorok használatával az olyan szinkron gyűrűben futó, vezetőválasztó algoritmusok, mint például az LCR és HS, aszinkron gyűrűben is futtathatók. Ez

azonban nem túl érdekes, mivel ezek az algoritmusok már működnek aszinkron hálózatokban, a szinkronizátorok szabta igények nélkül.

Egy tetszőleges, nem irányított, ismert *átm* átmérőjű gráfon alapuló aszinkron hálózatban egy szinkronizátor segítségével futtatható a MaxTerjed szinkron vezetőválasztó algoritmus. Az Alfa szinkronizátort használva az eredményül kapott algoritmus $\mathcal{O}(|E| \cdot \mathit{átm})$ üzenetet küld $\mathcal{O}(\mathit{átm} \cdot d)$ idő alatt a szükséges *átm* szinkron menet szimulálásához.

Ugyancsak egy szinkronizátor segítségével futtatható az OptMaxTerjed szinkron vezetőválasztó algoritmus, amely csak abban különbözik a MaxTerjed algoritmustól, hogy a csúcsok csak új információ közlésekor küldenek üzenetet. Ha az Alfa szinkronizátort használjuk, elveszítjük az optimalizálás előnyét, mivel a szinkronizátor minden menetben minden csatornán üzeneteket küld. Ha viszont a Béta szinkronizátort használjuk, igen alacsonyan tudjuk tartani a kommunikációs bonyolultságot (az időigény terhére).

16.5.2.. Szélességi keresés

Emlékezzünk vissza, hogy a 4.2. alfejezetben közölt SzinkSzK algoritmusnak $\mathcal{O}(|E|)$ üzenetre és $\mathcal{O}(\mathit{átm})$ menetre van szüksége egy *átm* átmérőjű hálózatban; a folyamatoknak nem kell tudniuk, mekkora az *átm*. A szinkronizátorok segítségével a SzinkSzK algoritmus aszinkron hálózatban is futtatható. Az Alfa szinkronizátort használva, az eredményül kapott algoritmus $\mathcal{O}(|E| \cdot \mathit{átm})$ üzenetet küld, és $\mathcal{O}(\mathit{átm} \cdot d)$ időt igényel az *átm* menet szimulálásához, amely az összes folyamat számára szükséges ahhoz, hogy a szüleikre vonatkozó információkat elküldjék. A Béta alkalmazásakor (legfeljebb *átm* magasságú fát használva), az algoritmus csak $\mathcal{O}(|E| + n \cdot \mathit{átm})$ üzenetet küld $\mathcal{O}(\mathit{átm}^2 \cdot d)$ idő alatt, ami megegyezik a 15.4. alfejezetben megadott RétegzettSzK algoritmus igényeivel. A Gamma felhasználásával az időbonyolultság csökkenthető a kommunikációs bonyolultság növekedésének terhére.

Felmerül egy technikai feladat: nem nyilvánvaló, hogyan érjenek véget a szinkronizátorokkal elkészített SzK algoritmusok. A leírtak alapján a megvalósítás a végtelenségig folytatja a menetek szimulációját, így végtelen sok üzenetet generál. (Ha a folyamatok ismernék az *átm* értékét, akkor egyszerűen befejeződhetnének *átm* menet szimulálása után, de feltételeztük, hogy a folyamatok nem ismerik az *átm* értékét.) Egy azonnali megoldás erre a problémára, hogy a szüleiket meghatározó felhasználói automaták már csak egy további menetet hajtsanak végre azért, hogy értesítsék a szomszédaikat, majd álljanak le.

16.5.3.. Legrövidebb utak

A szinkronizátor kiválóan alkalmas a legrövidebb út megkereséséhez egy kijelölt forrásból. Idézzük fel, hogy az AszinkBellmanFord algoritmus üzenet- és időbonyolultsága a csúcsok számának exponenciális függvénye. A szinkron BellmanFord algoritmusnak a kommunikációs bonyolultsága azonban „csak” $\mathcal{O}(n|E|)$, a menetbonyolultsága pedig $\mathcal{O}(n)$ egy ismert n méretű hálózatra. A szinkron BellmanFord algoritmust futtathatjuk például az Alfa szinkroni-

zátor segítségével, ahol az eredményül kapott algoritmus $\mathcal{O}(n|E|)$ üzenetet küld $\mathcal{O}(nd)$ idő alatt a szükséges n menet szimulációjához. Az EgyszerűSzink szinkronizátort ugyancsak használhatjuk.

16.5.4.. Üzenetszórás és nyugtázás

Tervezhető olyan szinkron algoritmus, amely lehetővé teszi a folyamatai számára, hogy üzeneteket küldjenek a többi folyamatnak és viszonzásul nyugtákat kapjanak, és ehhez $\mathcal{O}(|E|)$ üzenetet és $\mathcal{O}(\mathit{átm})$ menetet használ (lásd 4-8. gyakorlat). Ezt az algoritmust az Alfa szinkronizátor segítségével futtathatjuk, így egy olyan üzenetszóró és -nyugtázó aszinkoron algoritmushoz jutunk, ami $\mathcal{O}(|E| \cdot \mathit{átm})$ üzenetet küld $\mathcal{O}(\mathit{átm} \cdot d)$ idő alatt. Hasonlítsuk össze az algoritmus bonyolultságát a 15.3. alfejezetben tárgyalt AszinkSzórásNyugtáz algoritmuséval.

16.5.5.. Maximális független halmaz

A szinkronizátorok véletlenített szinkron algoritmusokhoz is használhatók, így például a LubyMFH-hoz. A részletes kidolgozást az olvasóra bízzuk.

16.6.. Alsó korlát a futási időre

A szinkronizátorokra vonatkozó eredmények egy informális átfogalmazása:

> Minden (nem hibatűrő) szinkron algoritmus áttranszformálható egy megfelelő aszinkron algoritmussá, az erőforrásigények túlzott növekedése nélkül.

Az Alfa vagy az EgyszerűSzink szinkronizátorok használatakor például teljes mértékben elkerülhető az időigény növekedése. Ebben az alfejezetben bemutatjuk a szinkronizátoros megközelítés egy korlátozását, alsó korlátot megadva egy aszinkron hálózati algoritmus adott feladat megoldásához szükséges futási idejére. Létezik azonban nagyon gyors szinkron algoritmus is ugyanezen feladat megoldására, ami informálisan elmondva a következőt jelenti.

> Nem minden szinkron algoritmus transzformálható hasonló időbonyolultságú aszinkron algoritmussá.

Ez a két informális megjegyzés ellentmondásosnak tűnhet. Kiderül azonban, hogy az eltérés oka a szinkronizátorok által garantált helyességi feltétel *helyi* tulajdonsága. Erre a megjegyzésre az alsó korlátra adott bizonyítás után térünk vissza.

Ez az alfejezet eredményül a könyvben szereplő egyetlen alsó korlátot adja egy aszinkron osztott rendszer időbonyolultságára.

Az itt vizsgált feladat az ú.n. „soros feladat". Legyen $G = (V, E)$ egy gráf a szokásos *átm* átmérővel. A rendszer környezettel érintkező felülete $\texttt{fény}_i$ kimeneti műveletekből áll, G minden i csúcsában; a $\texttt{fény}_i$ az i csúcs folyamatautomatájának kimenete. A `fény` műveleteket absztrakt műveletekként kezeljük, de

gondolhatunk úgy rájuk, mint egy adott számítási feladattal elkészülő folyamat jeladásaira.

A *sor*-t a úgy definiáljuk, mit `fény`-ek egy olyan sorozatát, amely tartalmaz legalább egy $\texttt{fény}_i$-t minden i-re. Minden nem negatív egész k-ra, a *k-soros feladat* legalább k számú diszjunkt sor elvégzését jelenti tetszőleges pártatlan végrehajtási sorozatban.

16.6.1. példa. A k-soros feladat motivációja

> A k-soros problémát eredetileg egy, az aszinkron közös memóriájú modellre vonatkozó mátrixszámítási feladat vetette fel. Tekintsük aszinkron párhuzamos folyamatoknak egy olyan halmazát, amely egy $m \times m$ méretű logikai értékű mátrix tranzitív lezártjának koordinált kiszámítását végzi. Kezdetben a mátrixot a közös memóriában tároljuk, továbbá a részeredemények és a végső kimenet ugyancsak a közös memóriába íródik.
>
> Minden i, j, k-hoz tartozik egy $P_{i,j,k}$ folyamat, ahol $1 \leq i, j, k \leq m$. Minden $P_{i,j,k}$ folyamat feladata egyszerűen csak annyi, hogy a kimeneti mátrix (i, j) elemét 1-re állítsa, amennyiben az (i, k) és (k, j) helyeken 1-et lát. Így minden folyamat egy egyszerű cikluson halad végig, beolvasva az (i, k) és (k, j) elemeket, majd (esetleg) módosítva az (i, j) elemet. A közös memória minden olvasási és írási műveletét egy `fény` kimenet reprezentálja.
>
> A mátrixok alaptulajdonságaiból következően a számítás végrehajtása akkor lesz helyes, ha van „elég" átfedés a folyamat lépései között. Speciálisan, $\mathcal{O}(\log n)$ sor elegendő. Nem okoz problémát, ha a folyamatok túllépik az olvasási és írási időt – amennyiben elegendő átfedés van beiktatva, helyes kimenet fog képződni.

A feladat egy egyszerűbb változatában, amelyre hasonló alsó korlát mutatható meg, minden folyamatnak *pontosan egy fényt* kell adnia minden sorban. A feladat általunk vizsgált változata kevesebb megszorítást tartalmaz, ezért eredményül erősebb alsó korlátot ad.

A k-soros feladat triviálisan megoldható szinkron hálózati környezetben. Összesen csak egy $\texttt{fény}_i$ kimenetet kell rendelnünk minden P_i folyamathoz a k menet mindegyikében. Nincs szükség a folyamatok kommunikációjára; pontosan k menet szükséges.

Aszinkron hálózati környezetben a folyamatokat a szokásos módon, b/k automatákkal modellezük, amelyeket megbízható FIFO küld/fogad csatornák kötnek össze. Az általánosság megszorítása nélkül feltehetjük, hogy a csatornák univerzálisak. Az időket a megszokott módon rendeljük az eseményekhez, ℓ a folyamatok taszkvégrehajtási idejének felső korlátja, d pedig a csatornákon lévő legrégebbi üzenetek kézbesítési idejének felső korlátja. Feltesszük, hogy $\ell \ll d$, és az eredményeinkben és bizonyításunkban az ℓ-et valójában nem vesszük figyelembe. Idézzük fel a 8.6. alfejezetből, hogy egy olyan pártatlan végrehajtási sorozatot, melynek

eseményeihez időket rendelünk a megadott korlátozásoknak megfelelően, *időzített végrehajtási sorozatnak* nevezzük.

Ezek után definiáljuk az A algoritmus $T(A)$ *időmértékét*. Az A minden α időzített végrehajtási sorozatához definiáljuk $T(\alpha)$-t, mint azoknak az időknek a pontos felső korlátját, amin belül `fény` esemény következik be az α-n belül. (Azért használunk pontos felső korlátot maximum helyett, mivel végtelen sok ilyen esemény bekövetkezhet.) Legyen ezek után

$$T(A) = \sup\{T(\alpha) : \alpha \text{ az } A \text{ egy időzített végrehajtási sorozata}\}.$$

Azaz $T(A)$ azoknak az időknek a pontos felső korlátja, amin belül egy `fény` esemény következik be az A időzített végrehajtási sorozatain belül.

Most kimondjuk és bebizonyítjuk az alsó korlátra vonatkozó tételt.

16.11. tétel. . *Tegyük fel, hogy az A aszinkron hálózati algoritmus megoldja a k-soros problémát a G gráfon. Ekkor $T(A) \geq (k-1)\mathit{átm} \cdot d$.*

Ahhoz, hogy összehasonlítsuk ezt az eredményt a szinkron rendszer k menetre vonatkozó felső korlátjával, célszerű az üzenetkézbesítés maximális idejét d-nek venni minden menetre. Ekkor a 16.11. tétel alapvető alsó korlátja és a kd kis felső korlát körülbelül egy *átm* tényezőben különbözik. Ez azt bizonyítja, hogy az aszinkronitás miatti alapvető késedelem a soros problémára egy *átm* tényező.

Bizonyítás. Az általánosság megszorítása nélkül feltehetjük, hogy A minden művelete külső. Indirekt módon bizonyítunk.

Tegyük fel, hogy létezik egy A algoritmus, úgy hogy $T(A) < (k-1) \cdot \mathit{átm} \cdot d$. A-nak egy időzített végrehajtási sorozatát *lassúnak* nevezzük, ha minden üzenet kézbesítéséhez a maximális d időre van szükség. Legyen α az A egy tetszőleges lassú időzített végrehajtási sorozata; megjegyezzük, hogy α-nak ettől az időinformációtól függetlenül A egy pártatlan végrehajtási sorozatának kell lennie. Mivel az A helyes, ezért az α-nak k sort kell tartalmaznia. Feltétel szerint az α-ban nem történhet `fény` esemény a $(k-1) \cdot \mathit{átm} \cdot d$ időpontban vagy azután. Az α-t így előállíthatjuk az $\alpha' \cdot \alpha''$ konkatenált alakban, ahol az α' utolsó eseményének ideje szigorúan kisebb, mint $(k-1) \cdot \mathit{átm} \cdot d$, és ahol az α'' nem tartalmaz `fény` eseményt. Továbbá, az α' felbontható $k-1$ kisebb részre, azaz $\alpha_1 \cdot \alpha_2 \cdot \ldots \cdot \alpha_{k-1}$ konkatenált alakba, ahol az α_r $(1 \leq r \leq k-1)$ részekben az első és utolsó eseményhez tartozó idők különbsége szigorúan kisebb, mint $\mathit{átm} \cdot d$.

Készítsük el most az A egy β pártatlan történetét, ahol a β egy közönséges *nem időzített* pártatlan történet – az eseményeihez rendelt idők nélkül. A β történetet a $\beta = \beta_1 \cdot \beta_2 \ldots \beta_{k-1}\beta''$ konkatenációs alakban adjuk meg, ahol minden β_r-et az α_r műveleteinek átrendezésével (és az idők elhagyásával) képezzük, a β'' pedig az α'' műveleteinek sorozatát tartalmazza (az idők elhagyásával). Megmutatjuk, hogy a β történet k-nál kevesebb sort tartalmaz, ami ellentmond az A helyességének.

A β elkészítésénél használt összes átrendezés megőrzi az α műveletei közötti fontos függőségi viszonyokat, így speciálisan egy `fogad` esemény függését a megfelelő `küld` eseménytől, továbbá egy P_i folyamat bármely (lehetséges) eseményének

függését ugyanazon folyamat egy korábbi eseményétől. A 14.1.4. szakaszban definiált $\rightarrow_{történet(\alpha)}$ jelölést használjuk az ezen függőségeket leíró, irreflexív részben rendezés jelölésére. A 14.1. tételt felhasználva megmutatjuk, hogy β valójában az A egy pártatlan története.

A következő segédtétel azokat a tulajdonságokat írja le, amelyeket az átrendezett β_r sorozatoktól megkövetelünk. Rögzítsük a G azon két tetszőleges j_0 és j_1 csúcsát, amelyek távolsága *átm*, továbbá legyen

$$i_r = \begin{cases} j_0, & \text{ha } r \text{ páros,} \\ j_1, & \text{ha } r \text{ páratlan.} \end{cases}$$

16.12. segédtétel. . *Minden r esetében, ahol $1 \leq r \leq k-1$, létezik az A műveleteinek egy olyan β_r sorozata, amelyre a következő tulajdonságok teljesülnek.*

1. *β_r az α_r műveletsorozatának átrendezésével keletkezik a $\rightarrow_{történet(\alpha)}$ rendezés megőrzése mellett.*
2. *β_r felírható a $\gamma_r\delta_r$ konkatenációs alakban, ahol a γ_r nem tartalmazza az i_{r-1} folyamat, a δ_r pedig az i_r folyamat eseményeit.*

Először megmutatjuk, hogyan fejezhető be a tétel bizonyítása a 16.12. segédtétel segítségével. Mivel az események átrendezése csupán egyedi β_r sorozatokra vonatkozik, és mivel az átrendezés megőrzi a $\rightarrow_{történet(\alpha)}$ függőségeket, a 14.1. tételből következően β az A pártatlan története. Meg tudjuk mutatni azonban, hogy β legfeljebb $k-1$ sort tartalmaz. Nincs olyan sor, amelyet a γ_1 teljes egészében tartalmazna, mivel a γ_1 nem tartalmazza az i_0 eseményeit. Hasonlóan, egyetlen $\delta_{r-1}\gamma_r$ alakú szegmens sem tartalmazhat egy sort teljes egészében, mivel ez a szegmens nem tartalmazza az i_{r-1} folyamat eseményeit. Ebből az következik, hogy minden sornak valamely γ_r–δ_r határ *mindkét oldalán* kell eseményeket tartalmaznia. Azonban csak $k-1$ ilyen határ, így legfeljebb csak $k-1$ sor létezik. Ezzel a β megsérti az A helyességét, ami ellentmondás.

Tartozunk még a 16.12. segédtételhez szükséges β_r sorozatok elkészítésével. Rögzítsünk ehhez egy tetszőleges r értéket, ahol $1 \leq r \leq k-1$. Tekintsük a következő eseteket.

1. α_r nem tartalmaz i_{r-1}-beli eseményt.
   Ekkor legyen β_r az α_r-beli műveletek átrendezés nélküli sorozata. A $\gamma_r = \beta_r$ és a $\delta_r = \lambda$ (az üres sorozat) választás biztosítja a kívánt tulajdonságokat.
2. α_r nem tartalmaz i_r-beli eseményt.
   Ekkor legyen β_r az α_r-beli műveletek átrendezés nélküli sorozata. A $\gamma_r = \lambda$ és a $\delta_r = \beta_r$ választás megfelelő.
3. α_r tartalmaz legalább egy i_{r-1}-beli és legalább egy i_r-beli eseményt.
   Ekkor legyen π az i_{r-1} első α_r-beli eseménye, és legyen ϕ az i_r utolsó α_r-beli eseménye. Azt állítjuk, hogy nem teljesülhet $\pi \rightarrow_{történet(\alpha)} \phi$, azaz a ϕ nem függhet a π-től. Ez azért van így, mert az α egy lassú végrehajtási sorozat, így egy α-beli üzenet eljutásának ideje az i_{r-1} folyamattól az i_r folyamathoz legalább $átm \cdot d$; mindazonáltal, az α_r első és utolsó eseménye közötti idő szigorúan kisebb, mint $átm \cdot d$.

Ezután azt állítjuk (és gyakorlatnak hagyjuk annak megmutatását – lásd 16-18. gyakorlat), hogy átrendezhetők az α_r eseményei úgy, hogy a ϕ megelőzze a π-t a $\rightarrow_{történet(\alpha)}$ részben rendezés további megőrzése mellett. Legyen β_r az események eredményül kapott sorozata, γ_r a β_r-nek a ϕ-re végződő előtagja, δ_r pedig a β_r megmaradó része. Az így előállított sorozatok rendelkeznek a kívánt tulajdonságokkal.

□

Ismét hangsúlyozzuk, hogy a 16.11. tétel bizonyításában megkonstruált β történet eseményeihez nincsenek idők rendelve. Az ellentmondás abból következik, hogy β nem tartalmaz elég sort, és nem a β időzítési tulajdonságaiból. A bizonyításban időzítési információt használtunk annak levezetéséhez, hogy bizonyos események nem függhetnek másoktól az α lassú időzített végrehajtási sorozatban.

A helyesség helyi fogalma.. A 16.11. tétel ellentmondásosnak tűnhet a szinkronizátorok egyes eredményeivel kapcsolatban – azokkal, amelyek szinkron algoritmusok aszinkronná transzformálását adják meg konstans időszükségletekkel. Az a különbség, hogy a szinkronizátorok a helyességnek csak egy „helyi" fogalmát garantálják. A felhasználók (azaz szinkron folyamatok) csoportos viselkedésének egészben történő megőrzése helyett a szinkronizátorok a felhasználók viselkedését külön-külön őrzik meg, megengedve a különböző felhasználók eseményeinek átrendezését.

Számos osztott alkalmazásban nem számít a különböző felhasználók eseményeinek sorrendje; például a speciális adatfeldolgozó és pénzügyi alkalmazások általában képesek a különböző felhasználók tranzakcióinak sorrendtől független lebonyolítására. Az olyan alkalmazásokban azonban, ahol jelentős a felhasználók osztott rendszeren kívüli kommunikációja, fontos lehet a különböző felhasználók eseményeinek sorrendje.

16.7.. Megjegyzések a fejezethez

Awerbuch [29] vezette be a szinkronizátor általános fogalmát, valamint a szinkronizációs feladat felbontását egy adatkommunikációs részre és egy biztonságos szinkronizátorra. Awerbuch cikkében definiálja még az Alfa, Béta és Gamma szinkronizátorokat, és algoritmusokat közöl a Gamma alkalmas csoportokra bontásához. Szinkronizátorok alkalmazásáról hatékony aszinkron algoritmusok elkészítéséhez a szélességi kereséshez és a maximális terjedéshez a [29,30]-ban olvashatunk. A [35,36,32] további eredményeket közöl hatékony csoportokra bontásokról. A b/k automatákat használó szinkronizátorok formális leírása Devarajannak [89] köszönhető, aki Fekete, Lynch és Shrira [109] korábbi munkáját folytatja.

Az alsó korlátra vonatkozó bizonyítás Arjomandi, Fischer és Lynch [14] nevéhez fűződik, akik eredményeiket közös memóriájú modellre közölték. Az ebben a fejezetben leírtak Attiya és Mavronicolas [17] bizonyos egyszerűsítéseit használják fel. Attiya és Mavronicolas [17] kiterjesztette az alsó határra vonatkozó eredményt részben szinkron rendszerekre is. Raynal egy teljes könyvet szentelt a szinkronizátoroknak [250].

16.8.. Gyakorlatok

16-1. Fogalmazzunk meg és bizonyítsunk be egy szoros megfeleltetést a 2. fejezet szinkron modellje és a 16.1. alfejezetben megadott, U_i felhasználói automatákból és GlobSzink-ből felépülő aszinkron modell között.

16-2. Dolgozzuk ki a 16.1. lemma bizonyításának részleteit. Speciálisan be kell bizonyítani a 16.2. segédtételről, hogy valóban megkapható a γ a β eseményeinek átredezésével a $\rightarrow \beta$ rendezés megtartása mellett.

16-3. Jelölje H, illetve G a HelyiSzink, illetve a GlobSzink rendszert, azzal a kis módosítással, hogy a külső műveletek pontosan a felhasználói automaták műveletei. (Azaz a felhasználók belső műveletei kimenetekként vannak újraosztályozva.) Végrehajtási ellenpélda keresésével bizonyítsuk be, hogy a *pártatlan_ történetek*$(H) \subseteq$ *pártatlan_ történetek*(G) nem teljesül.

16-4. Egészítsük ki az EgyszerűSzink rendszerre vontakozó bizonyítást és bonyolultságvizsgálatot. Nevezetesen

(a) fogalmazzuk meg és bizonyítsuk be a szükséges invariáns állításokat;

(b) bizonyítsuk be, hogy f egy szimulációs reláció;

(c) végezzük el precízen a pártatlansági vizsgálatot a a 8.13 tétel alapján;

(d) adjunk bizonyítást az időbonyolultsági állításra (ne feledkezzünk meg róla, hogy a d feltételezett korlátja csak a csatornákon lévő aktuálisan „legrégebbi" üzenet kézbesítésére vonatkozik).

16-5. Jelölje E, illetve G az EgyszerűSzink, illetve a GlobSzink rendszert, azzal a kis módosítással, hogy a külső műveletek pontosan a felhasználói automaták műveletei. (Azaz a felhasználók belső műveletei kimenetekként vannak újraosztályozva, és a `küld` és `fogad` műveletek rejtettek – vagyis belsőként vannak újraosztályozva.)

(a) Végrehajtási ellenpélda keresésével bizonyítsuk be, hogy a *pártatlan_ történetek*$(E) \subseteq$ *pártatlan_ történetek*(G) nem teljesül.

(b) Az S módosításával készítsünk el egy új, ugyancsak felhasználói automatákból és egy osztott algoritmusból álló S' rendszert úgy, hogy *pártatlan_ történetek*$(E) \subseteq$ *pártatlan_ történetek*(G) teljesüljön. Elemezzük a bonyolultságát.

16-6. Dolgozzuk ki a 16.5. lemma bizonyításának részleteit.

16-7. Írjunk előfeltétel/hatású kódot az Alfa$_i$ automatára, és bizonyítsuk be a helyességét állító 16.7. tételt. Használjunk egy szimulációs relációt az Alfa és a BiztSzink rendszerek között.

16-8. Írjunk előfeltétel/hatású kódot a Béta$_i$ automatára, és bizonyítsuk be a helyességét állító 16.8. tételt. Használjunk egy szimulációs relációt a Béta és a BiztSzink rendszerek között.

16-9. Igaz vagy hamis?
Jelölje B, illetve G a Béta, illetve a GlobSzink rendszert, ismét azzal a kis módosítással, hogy a külsőként osztályozott műveletek pontosan a felhasználói au-

tomaták műveletei. Ekkor *pártatlan_ történetek*$(B) \subseteq$ *pártatlan_ történetek*(G). Válaszunkat indokoljuk.

16-10. Adjunk meg előfeltétel/hatású kódot a csúcsok folyamataira a CSOPORTSZINK és az ERDŐSZINK automaták GAMMA szinkronizátorban való megvalósításához. Bizonyítsuk be a 16.10. tételt.

16-11. Adjunk meg egy tetszőleges G hálózati gráfon működő olyan osztott algoritmust, amely minimális magasságú gyökeres feszítőfát állít elő a BÉTA szinkronizátor használatához. Feltételezhetjük, hogy a csúcsok egyedi azonosítókkal rendelkeznek, de nincs kijelölt csúcs. Milyen hatékony algoritmust tudunk készíteni?

16-12. Adjunk meg egy tetszőleges G hálózati gráfon működő olyan osztott algoritmust, amelyik egy „jó" feszítőerdőt állít elő a GAMMA szinkronizátor használatához. Állítsuk elő a szomszédos csoportok gyökerének érintkezéséhez használt megkülönböztetett utakat is. Feltételezhetjük, hogy a csúcsok egyedi azonosítókkal rendelkeznek, de nincs kijelölt csúcs. Az algoritmusnak alacsony fákat és rövid érintkezési utakat kell szolgáltatnia.

16-13. Tekintsünk egy $\sqrt{n} \times \sqrt{n}$ csúcsú G négyzetrácsgráfot. Tekintsünk egy k^2 egyenlő méretű csoportokra való P_k felbontást az oldalak k egyenlő részre való felosztásával. Mennyi a P_k felbontáson alapuló GAMMA szinkronizátor érintkezési és időbonyolultsága az n és k paraméterre nézve? (Feltételezhető a lehető legjobb feszítőfák és érintkezési utak használata.)

16-14. A Zöldfülűek Számítástechnikai Rt. egy, a hibatűrő algoritmusokkal kapcsolatban jelentős tapasztalatokkal rendelkező programozójának az a briliáns ötlete támadt, hogy szinkronizátort használjon hibatűrő aszinkron hálózati programozásban. Beismeri, hogy az elképzelése csak egy teljesen összefüggő G hálózatban működik, ennek ellenére felfedezését nagyon fontosnak tartja.

A szinkronizátora hasonlít a GLOBSZINK rendszerre, azzal a különbséggel, hogy minden r-edik menetben egy $\texttt{felhasználó\_fogad}_i$ esemény végrehajtásához az r-edik menet `felhasználó_küld` eseményeinek beérkezését nem az összes n, csak legalább $n - f$ folyamattól (beleértve a P_i-t is) várja meg.

Mutassuk meg a főnökeinek, hogy ez az algoritmus helytelen, mielőtt még egy repülésirányító környezetbe helyeznék azt. (*Útmutatás.* Tekintsünk egy helyes szinkron megegyezési algoritmust a tervezett algoritmus helyett, például a HALMAZTERJED-et. Állítsuk elő a kombinált algoritmus egy helytelen végrehajtási sorozatát.)

16-15. Bizonyítsuk be, hogy a SZINKSZK-ra szinkronizátorral leírt befejezési stratégia helyesen működik.

16-16. Fogalmazzunk meg és bizonyítsunk be egy állítást azon fontos tulajdonságok meglétéről, amelyeket a LUBYMFH kedvelt szinkronizátorunkkal való futtatásával nyert aszinkron algoritmus garantál.

16-17. Bizonyítsuk be, hogy $\mathcal{O}(\log n)$ sor elégséges a 16.6.1. példában leírt logikai mátrix, tranzitív lezárt feladat megoldásához. Mi a legjobb konstans, amit bizonyítani tudunk?

16-18. Bizonyítsuk be a 16.11. tétel bizonyításából hiányzó állítást, mely szerint át lehet rendezni az α_k eseményeit úgy, hogy ϕ megelőzze π-t a $\rightarrow_{történet(\alpha)}$ rendezés megtartásával.

16-19. Adjunk meg lehető legjobb *felső korlátot* a k-soros problémára adott aszinkron megoldás időbonyolultságára. Általánosítsuk az algoritmusunkat tetszőleges szinkron algoritmus aszinkron megvalósítására. Milyen helyességi feltételek garantálhatók?

16-20. Végezzük el újra a 15-40. gyakorlatot, ezúttal az ebben a fejezetben közölt egyik algoritmusfelbontási elv segítségével. Alkalmazzunk a lehető legtöbb modularitást. Absztrakt automatákat adhatunk meg például az MFF és a vezetőválasztáshoz MFF-t használó algoritmus által megkövetelt viselkedés reprezentálásához.

17. fejezet

Közös memória és hálózatok

Az előző fejezetben olyan szinkronizátorokat ismertettünk, amelyek egy módszert adnak az aszinkron hálózatok egyszerűbb programozására. Ez a módszer lehetővé teszi, hogy például a 4. fejezetben leírt, (nem hibatűrő) szinkron hálózati algoritmusokat aszinkron hálózatokban alkalmazzuk. Ebben a fejezetben az aszinkron hálózatok programozásának egyszerűsítésére egy másik stratégiát is mutatunk: az aszinkron közös memóriájú rendszerek szimulálására való felhasználásukat. Ez lehetőséget teremt, hogy a 10., 11. és 13. fejezetben leírt aszinkron közös memóriájú algoritmusokat aszinkron hálózatokban használjuk. Sok más aszinkron közös memóriájú algoritmus is adaptálható aszinkron hálózatokban való futtatáshoz, mint például gyakorlati algoritmusok tudományos célú programozáshoz vagy pénzügyi adatbázisokhoz. Ennek a stratégiának az az alapgondolata, hogy az aszinkron közös memóriájú modell könnyebben programozható, mint az aszinkron hálózati modell.

Általánosabban fogalmazva, ez a fejezet az aszinkron közös memóriájú modell és az aszinkron hálózati modell közötti kapcsolattal foglalkozik. Kiderül, hogy mindkét irányban erős transzformációs eredmények érhetők el, amelyek némelyike még bizonyos hibatűrő tulajdonságokat is megőriz. Ebből az következik (a hatékonyságok különbözőségét leszámítva), hogy a két modell lényegében ugyanaz.

Ezek a transzformációs eredmények az aszinkron hálózati modell egyszerűbb programozhatóságának biztosítása mellett más következményekkel is járnak. Például egy hálózati modellből közös memóriájú modellbe való hibatűrő transzformáció esetén az aszinkron közös memóriájú modell bizonyos lehetetlen eredményei lehetetlen eredményekhez vezetnek az aszinkron hálózati modellben.

A 18.3.3. szakasz bemutat egy másik típusú transzformációt az aszinkron közös memóriájú modellből az aszinkron hálózati modellbe. Ez a transzformáció a *logikai idő* fogalmának aszinkron hálózatra való bevezetésén alapul.

17.1.. A közös memória modell transzformálása a hálózati modellre

Ebben az alfejezetben több eljárást mutatunk be az aszinkron közös memóriájú rendszerek aszinkron küld/fogad hálózati rendszerekre való transzformációjára. A 17.1.1. szakasz megadja a transzformációk által kielégítendő helyességi feltételeket. A 17.1.2. szakasz nem hibatűrő stratégiákat, a 17.1.3. szakasz pedig hibatűrő stratégiákat tartalmaz. A hibák közül egyedül csak a folyamat megállási hibákat vesszük figyelembe.

17.1.1.. A feladat

A 9. fejezet modellje szerinti A közös memóriájú rendszerrel kezdjük a vizsgálatot. A szokott módon feltételezzük, hogy A a környezetével való információcseréhez $1, \ldots, n$ indexszel ellátott n kaput használ, és az i-edik kapun az U_i automatával kommunikál. A 10–13. fejezetek felhasználói automatáihoz hasonlóan feltételezzük, hogy U_i külső műveletei éppen azok, amelyeket az A-val való kommunikációhoz használ. Ebben a fejezetben A minden folyamata tetszőleges számú taszkkal rendelkezhet (a kapukra és A folyamataira rendszerint az i indexükkel hivatkozunk). Mivel egyes transzformációink megőrzik a hibatűrő tulajdonságokat, ezért a 9.6. alfejezetnek megfelelően $\texttt{megállít}_i$ bemeneti műveleteket használunk, és feltesszük, hogy minden $\texttt{megállít}_i$ esemény véglegesen letiltja P_i minden taszkját.

Az a helyzet, hogy a transzformációink helyes működéséhez meg kell adnunk A-ra egy technikai korlátozást. Ez a technikai korlátozás a 13.1.4. szakaszban használttal megegyező. Vagyis A-t kombináljuk a felhasználói automaták minden lehetséges csoportjával. Feltételezzük, hogy minden i kapuhoz létezik egy olyan $\mathit{váltás}_i$ függvény, amely a kombinált rendszer minden α véges végrehajtási sorozatára *rendszer* vagy *felhasználó* értéket ad. Ez azt mutatja meg, hogy az α után minek a végrehajtása történjen a következő lépésben. Azt írjuk elő, hogy ha $\mathit{váltás}_i(\alpha) = \mathit{rendszer}$, akkor U_i nem hajthat végre kimeneti lépést ebben az állapotban az α után, ha pedig $\mathit{váltás}_i(\alpha) = \mathit{felhasználó}$, akkor az A rendszer P_i folyamatának nincsen kimeneti vagy belső – azaz helyileg vezérelt – lépése ebben az állapotban az α után. Azaz ugyanazt a közös memóriájú modellt tételezzük fel, amit a 13.1.4. szakaszban.

Az általános problémát (a kapuk tetszőleges I halmazára megadott hibatűrési követelménnyel együtt) egy P_i $(1 \leq i \leq n)$ folyamatokkal rendelkező B aszinkron küld/fogad hálózati rendszer megtervezése jelenti, azaz A következőképpen definiált *I-szimulációja*: B minden α végrehajtási sorozatához az U_i felhasználók minden csoportja mellett léteznie kell ugyanazon felhasználókkal az A egy olyan α' végrehajtási sorozatának, amely teljesíti az alábbi feltételeket:

1. α és α' megkülönböztethetetlen[1] az U-ra (az U_i felhasználók együttesére) nézve;

[1] Itt a 8.7. alfejezet „megkülönböztethetetlen” fogalmát használjuk.

2. minden i-re pontosan akkor jelenik meg egy `megállít`$_i$ az α-ban, amikor megjelenik egy `megállít`$_i$ az α'-ben.

Továbbá, ha α egy pártatlan végrehajtási sorozat és minden i, amelyre `megállít`$_i$ jelenik meg α-ban, I-be esik, akkor α' is egy pártatlan végrehajtási sorozat. Ha B az A *I-szimulációja* minden olyan I-re, amelyre $|I| \leq f$, akkor a B-t az A rendszer *f-szimulációjának* nevezzük.

Ezek a feltételek hasonlónak tűnhetnek a 13.7. tételben megfogalmazottakhoz. Ezt a kapcsolatot kihasználjuk ebben a fejezetben annak bizonyítására, hogy bizonyos hálózati rendszerekkel közös memóriájú rendszereket szimulálhatunk.

A 14.1.1. szakaszban leírtaknak megfelelően, a B rendszer minden `megállít`$_i$ eseménye véglegesen letiltja a P_i folyamat minden taszkját. Mindazonáltal, egy `megállít` eseménynek nincs hatása a csatornákra.

17.1.2.. Hibamentességet feltételező stratégiák

Hibamentesség esetén egyszerű stratégiák is használhatók. Ezek legtöbbje *egy-másolatú* vagy *több-másolatú* sémaként osztályozható, a hálózatban használt közös változók másolatainak száma alapján.

Egy-másolatú sémák.. A legegyszerűbb szimulációs stratégia az A közös változóinak B folyamatai közötti tetszőleges elosztását foglalja magában, ahol minden közös változó egyetlen folyamathoz tartozik. A stratégia tetszőleges típusú közös változókra működik.

> **EgyszerűKözösVáltSzim algoritmus (vázlatosan)**
>
> A minden x közös változójáról feltételezzük, hogy B egyetlen P_i folyamatához tartozik. P_i feladata kettős: A megfelelő i folyamatának szimulálása, és az ahhoz tartozó közös változók kezelése.
>
> P_i műveletei a felhasználói felületen minden i-re megegyeznek az A rendszer i folyamatának műveleteivel. P_i lépései az i folyamat lépéseit közvetlenül szimulálják a következő kivételével: ha az A rendszer P_i folyamata egy x közös változóhoz fér hozzá, P_i ehelyett üzenetet küld az x változót tartalmazó P_j megszólításával. (Ha maga P_i tartalmazza a változót, akkor ezt a kérést egy „szubrutinhoz" továbbítja.) Ezután P_i mindaddig felfüggeszti az i folyamat minden helyileg vezérelt szimulációs lépését, amíg választ nem kap a kérésre. Ha megérkezik a válasz, a P_i a szokásos módon folytatja az A rendszer i folyamatának szimulálását.
>
> Ha egy x közös változót tartalmazó folyamat az x hozzáférésére vonatkozó üzenetet (vagy helyi megszólítási kérést) kap, akkor azt egyszerűen alkalmazza x-re egyetlen oszthatatlan lépésben. A válasz válaszüzenet formájában érkezik meg a kérés feladójához (vagy ha a kérés helyi, akkor a ő szimulációs taszkhoz lesz visszaküldve).

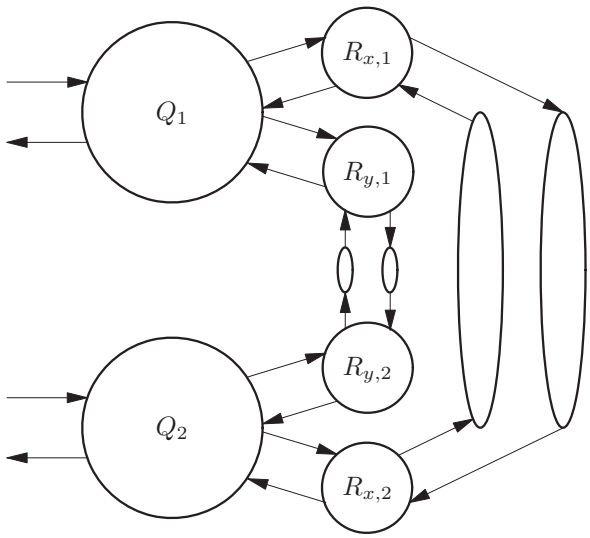
Az EgyszerűKözösVáltSzim algoritmus érdekes modularitással rendelkezik. Minden P_i folyamat kifejezhető egy, az A rendszer i folyamatának szimulációs

jáért felelős Q_i b/k automata, és a közös változókhoz rendelt $R_{x,i}$ B/K automaták összekapcsolásaként.[2]

A Q_i-hez használhatjuk egyszerűen a 13.1.4. szakasz TRANSZ(A) algoritmusának P_i automatáját. Pontosabban fogalmazva, feltételezzük, hogy a Q_i automata kimenetei $a_{x,i}$ alakú műveletek, ahol az a kérést az A rendszer i folyamata az x közös változó hozzáférésére használja, továbbá hogy a bemeneti műveletek $b_{x,i}$ alakúak, ahol b az x közös változótól az i folyamatnak küldött válasz.

Minden $R_{x,i}$ automata $a_{x,i}$ bementekkel és $b_{x,i}$ kimenetekkel rendelkezik. Kényelmi okból feltételezzük, hogy minden $R_{x,i}$ automata megbízható FIFO küld/fogad csatornákat használ az egymás közötti kommunikációhoz, minden x közös változóra. A 14-6. gyakorlathoz hasonlóan, az x-hez használt csatornák mindegyike szimulálható a megadott FIFO megbízható csatornákkal. Ki fog derülni, hogy minden x-re az $R_{x,i}$ automatáknak a közöttük lévő csatornákkal együtt képzett összetétele az x változó típusának megfelelő atomi objektumot alkot.

A 17.1. ábra az EGYSZERŰKÖZÖSVÁLTSZIM algoritmus architektúráját szemlélteti, két folyamat és két közös változó esetére. Nem jelöltük explicit módon a `megállít` műveleteket – feltételezzük, hogy minden $\texttt{megállít}_i$ bemenete a Q_i-nek és minden $R_{x,i}$-nek.



17.1.. ábra. Az EGYSZERŰKÖZÖSVÁLTSZIM architektúrája – két folyamat és két közös változó.

Az alábbiakban közöljük az $R_{x,i}$ kódját. Két részre bontjuk aszerint, hogy az x változó a P_i-hez tartozik-e, vagy sem. Mivel a $\texttt{megállít}_i$ műveletet az $R_{x,i}$ lenyomata tartalmazza, megadjuk a $\texttt{megállít}_i$ művelet kezelésének explicit leírását: ez

[2]Ezen kívül a közöttük lévő kommunikációs műveleteket is elrejtjük.

egyszerűen beállít egy *leállítva* mutatót, amely letilt minden helyileg vezérelt műveletet, és megakadályozza a bemeneti műveletekhez kapcsolódó módosításokat. (Ez a megközelítés nem különösebben érdekes itt, mivel nem támasztunk követelményeket az algoritmus viselkedésével szemben hibák esetére.) Az egyértelműség kedvéért, a csatornaműveleteket alsó indexben a változó és a két csúcspont nevével jelöljük.

- -

17.1. automata. $R_{x,i}$**,** x P_i**-hez tartozik**

Lenyomat:

Bemeneti:

$\texttt{a}_{x,i}$, a az i folyamat x-re vonatkozó kérése

$\texttt{fogad}(\text{„}\textit{kérés}\text{''}, a)_{x,j,i}$, a a j x-re vonatkozó kérése, $j \neq i$

$\texttt{megállít}_i$

Kimeneti:

$\texttt{b}_{x,i}$, b az x válasza az i folyamatnak

$\texttt{küld}(\text{„}\textit{válasz}\text{''}, b)_{x,i,j}$, b az x válasza j-nek, $j \neq i$

Belső:

$\texttt{végrehajt}(a, j)_{x,i}$, a az x-re vonatkozó kérés, $1 \leq j \leq n$

Állapotok:
érték, egy érték az x értelmezési tartományából, kezdetben az x kiindulási értéke
kér_puffer, (a, j) párok halmaza, a egy kérés, $1 \leq j \leq n$, kezdetben üres
vál_puffer, b válaszok halmaza, kezdetben üres
leállítva, logikai érték, kezdetben *hamis*
$\forall j \neq i$: *küld_puffer*(j), válaszok egy FIFO sora, kezdetben üres

Átmenetek:

`végrehajt`$(a, j)_{x,i}$
  Előfeltétel:
    leállítva=*hamis*
    $(a, j) \in$ *kér_puffer*
  Hatás:
    kér_puffer := *kér_puffer* $- \{(a, j)\}$
    $(b,$ *érték*$) := f(a,$ *érték*$)$
    if $j = i$ **then**
      vál_puffer := *vál_puffer* $\cup \{b\}$
    else
      adjuk hozzá b-t *küld_puffer*(j)-hez

`fogad`(*„kérés”*$, a)_{x,j,i}$
  Hatás:
    if *leállítva*=*hamis* **then**
      kér_puffer := *kér_puffer* $\cup \{(a, i)\}$

`küld`(*„válasz”*$, b)_{x,i,j}$
  Előfeltétel:
    leállítva=*hamis*
    b az első *küld_puffer*(j)-ben
  Hatás:
    töröljük *küld_puffer*(j) első elemét

`b`$_{x,i}$
  Előfeltétel:
    leállítva = *hamis*
    $b \in$ *vál_puffer*
  Hatás:
    vál_puffer := *vál_puffer* $- \{b\}$

`a`$_{x,i}$
  Hatás:
    if *leállítva* = *hamis* **then**
      kér_puffer := *kér_puffer* $\cup \{(a, i)\}$

`megállít`$_i$
  Hatás:
    leállítva := *igaz*

Taszkok:
$\{$`b`$_{x,i}$: b egy válasz$\}$
$\forall j$:
  $\{$`küld`(*„válasz”*$, b)_{x,i,j}$: b egy válasz$\}$
  $\{$`végrehajt`$(a, j)_{x,i}$: a egy kérés$\}$

- -

- -

17.2. automata. $R_{x,i}$, x nem tartozik P_i-hez

Lenyomat:
Bemeneti:
  `a`$_{x,i}$, a a P_i folyamat x-re vonatkozó kérése
  `fogad`(*„válasz”*$, b)_{x,j,i}$, b az x válasza i-nek, x j-hez tartozik
  `megállít`$_i$
Kimeneti:
  `b`$_{x,i}$, b az x válasza az i folyamatnak
  `küld`(*„kérés”*$, a)_{x,i,j}$, a a P_i folyamat x-re vonatkozó kérése, x j-hez tartozik

Állapotok:
vál_puffer, b válaszok halmaza, kezdetben üres
küld_puffer, kérések egy FIFO sora, kezdetben üres
leállítva, logikai érték, kezdetben *hamis*

Átmenetek:

$\texttt{a}_{x,i}$
Hatás:
if *leállítva*=*hamis* **then**
adjuk hozzá a-t a *küld_puffer*-hez

$\texttt{küld}(\text{„}kérés\text{”}, a)_{x,i,j}$
Előfeltétel:
leállítva=*hamis*
a az első a *küld_puffer*-ben
Hatás:
töröljük *küld_puffer* első elemét

$\texttt{fogad}(\text{„}válasz\text{”}, b)_{x,j,i}$
Hatás:
if *leállítva*=*hamis* **then**
vál_puffer := *vál_puffer* $\cup \{b\}$

$\texttt{b}_{x,i}$
Előfeltétel:
leállítva = *hamis*
$b \in$ *vál_puffer*
Hatás:
vál_puffer := *vál_puffer* $- \{b\}$

$\texttt{megállít}_i$
Hatás:
leállítva := *igaz*

Taszkok:
$\{\texttt{b}_{x,i}$:b egy válasz$\}$
$\{\texttt{küld}(\text{„}kérés\text{”},a)_{x,i,j} : a$ egy kérés$\}$

- -

17.1. tétel. . *Az A-n alapuló* EgyszerűKözösVáltSzim *algoritmus az A egy 0-szimulációja. (Nem követelünk meg hibatűrő tulajdonságokat.)*

Bizonyításvázlat. Először azt mutatjuk meg, hogy minden x-re az $R_{x,i}$ automaták, $1 \leq i \leq n$, és a közöttük lévő csatornák (a `küld` és `fogad` műveletek elrejtésével) együttesen az x változó típusának megfelelő B_x atomi objektumot alkotnak, melynek felületét a 13.1.4. szakaszban a Transz definíciójában adtuk meg. (Nem követelünk meg azonban hibatűrő tulajdonságokat B_x-re.) Ezekkel a B_x atomi objektumokkal a B rendszer pontosan megegyezik a Transz(A) rendszerrel. Ez lehetővé teszi a 13.7. tétel alkalmazását, $I = \emptyset$ mellett: ha α a B egy végrehajtási sorozata U_i felhasználókkal, akkor a 13.7. tétel biztosítja A egy olyan α' végrehajtási sorozatát ugyanazon felhasználókkal, amely kielégíti a 0-szimuláció definíciójában szereplő összes feltételt. □

Közös változók elhelyezése.. Az EgyszerűKözösVáltSzim algoritmus engedélyezi, hogy a változókat tetszőleges folyamatok tartalmazzák. Általános útmutatásként azonban az tanácsolható, hogy legjobb teljesítmény eléréséhez a változókat azokba a folyamatokba célszerű elhelyezni, amelyek a leggyakrabban érik el azokat.

Ha például egy x kizárólagosan író/megosztottan olvasható közös változót gyakrabban írunk, mint olvasunk, akkor természetesen az írásnak megfelelő folyamatba célszerű elhelyezni. Ha ezt tesszük, akkor az `ír` hozzáférések gyorsak lesznek, mivel a végrehajtásuk helyileg történik. Ebben az esetben természetesen az `ír` folyamaton kívüli minden olvasási hozzáférés lassú lesz, mivel ehhez üzenetcserékre van szükség a hálózaton belül. Ha az `ír` hozzáférések gyakoribbak, mint az `olvas`, ez az elrendezés jól működik, de kevésbé kielégítő, ha az `ír` viszonylag ritka.

Hibatűrés.. Az EGYSZERŰKÖZÖSVÁLTSZIM algoritmusnak nincsenek érdekes hibatűrő tulajdonságai. Ha például megjelenik egy $\texttt{megállít}_i$, akkor az ezután következő egyik folyamat sem érheti el a P_i változóit.

Lekötve várakozás.. Bizonyos közös memóriájú algoritmusok, mint például a 10.7. alfejezet kölcsönösen kizáró VÁRÓTEREM algoritmusa vagy a 11.3. alfejezet JOBBBALÉTKFIL étkező filozófusok algoritmusa, *lekötve várakozó hurkokat* tartalmaznak, amelyekben egy folyamat ismétlődően ellenőriz egy közös változót, egy feltétel teljesülésére várva. Az EGYSZERŰKÖZÖSVÁLTSZIM algoritmus módosítható úgy, hogy ezek a hurkok eltűnjenek, amennyiben a változót tartalmazó folyamat értesíti a lekötve várakozó folyamatot a változó értékének megváltozásáról (illetve a kívánt feltétel teljesüléséről). Ezzel csökkenthető a kommunikáció bonyolultsága.

Több-másolatú sémák.. Időnként hasznos lehet annak engedélyezése, hogy több folyamat kezelhesse egy és ugyanazon x közös változó másolatait. Tekintsük például azt az esetet, amikor x egy olvasható/írható közös változó, amelyre az `olvas` műveletek gyakoriak, az `ír` műveletek pedig ritkák. (A legtöbb adatbázisban ez a helyzet.) Ekkor, ha több folyamat is kezeli az x „belső" másolatait, sok `olvas` művelet hajtható végre helyileg, alacsony „költségek" mellett. A feladat azonban az, hogy az `ír` műveletek a korábbinál költségesebbé válnak, mivel az x összes másolatán végre kell őket hajtani. Ez azt jelenti, hogy az író folyamatoktól üzenetet kell küldeni minden olyan folyamathoz, amely x egy másolatát kezeli.

A feladat azonban még ennél is súlyosabb. Tegyük fel például, hogy x egy többszörösen írható regiszter. Ekkor két folyamat, például a P_1 és P_2, egyszerre megpróbálhatja *írni* az x-et, és az x másolatait kezelő két folyamat, például a P_3 és P_4, üzeneteket tud fogadni a P_1-től és a P_2-től, fordított sorrendben. Ez ahhoz vezethet, hogy az `ír` műveleteket más sorrendben próbálják meg alkalmazni a másolatukra, ami inkonzisztenciát eredményez a későbbi `olvas` műveletekben.

Még abban az esetben is, amikor x egyszeresen írható regiszter, rendellenesség történhet. Ha az író `ír` üzenetet küld ki, az üzenet sokkal hamarabb megérkezhet az egyik, például P_1 folyamathoz, mint egy másik, például P_2-höz. Az üzenet fogadása után az új értéket használó helyi `olvas` művelet történhet a P_1-ben, míg később, az üzenet fogadása előtt, még a régi értéket használó helyi `olvas` művelet történhet a P_2-ben. Amennyiben az első `olvas` a második `olvas` megkezdődése előtt véget ér, ez a viselkedés nem fogadható el egy olvasható/írható atomi objektumban.

Ezért egy intelligensebb protokollra van szükség az `ír` műveletek kezeléséhez. Például egy író működhet két fázisban: az első fázisban „zárolja" és módosítja

az x minden másolatát, majd a második fázisban feloldja a zárolást. Az `olvas` művelet késleltetve van a helyi másolat zárolása alatt. Ebben az esetben ügyelni kell rá, hogy a megkezdett műveletek végrehajtása valóban megtörténjen.

Az ilyen típusú algoritmus példa egy *párhuzamosságot vezérlő algoritmusra*. Speciálisan a fent vázolt algoritmus egy *olvasható/írható zárolási algoritmus*, amely egy *atomi tranzakciót* valósít meg az x összes másolatának írásával. Más szóval, ezt az x-en műveleteket végrehajtó folyamatok úgy látják, hogy az írási műveletek az x összes másolatára azonnal végrehajtódnak az őket tartalmazó `ír` művelet intervallumának bizonyos „sorrendi pontján". Sok más típusú párhuzamosságot vezérlő protokoll is létezik, így az olvasható/írható változók mellett más típusú közös változókat zároló algoritmusok, *időbélyeg-alapú algoritmusok*, a zárolást és az időbélyegek használatát kombináló *kevert algoritmusok* és az *optimista algoritmusok*. Ezeket nem tárgyaljuk itt, a (jelen könyvhöz hasonló stílusú) teljes leírásuk megtalálható Lynch, Merritt, Weihl és Fekete *Atomic Transactions* című könyvében.

Az olvasható/írható közös változókhoz egy népszerű többmásolatú algoritmus a TöbbségiSzavazás *algoritmus*. Az algoritmus megvalósításának lényege, hogy minden x olvasható/írható közös változóhoz egy olvasható/írható atomi objektumot készít el. Ez pedig a megvalósítás alapjául szolgáló atomi tranzakciók elkészítésén alapul.

TöbbségiSzavazásObjektum algoritmus (vázlatosan)

Az n folyamat mindegyike az x egy másolatát kezeli, amelyek kezdetben az x kezdőértékével egyeznek meg, továbbá egy nemnegatív *címke* egészet, amelynek kezdőértéke $\emptyset$.

Egy x-et *olvasni* vagy *írni* akaró folyamat végrehajt egy atomi tranzakciót, amely az x egy vagy több másolatát érinti. Az atomi tranzakció műveletek sorából épül fel, amelyek látszólag azonnal végrehajtásra kerülnek bizonyos „sorrendi pontokon" a tranzakció végrehajtása során. (Egy befejezetlen műveletnek lehet is, és nem is lehet sorrendi pontja.) A tranzakciók kétfázisú zárolással, időbélyegző-alapú, hibrid vagy optimista párhuzamosságot vezérlő módszerekkel valósíthatók meg bizonyos prioritási mechanizmusokkal kibővítve, amelyek célja, hogy minden tranzakció ténylegesen befejeződjön (ha mindegyik folyamat hibamentes).

Ahhoz, hogy a P_i folyamat *olvassa* az x-et az szükséges, hogy P_i olvassa az x másolatainak többségét. Ezek közül kiválasztja a legnagyobb *címke* értékűt, és visszatér az x ennek megfelelő értékével. Ezek a lépések ugyanannak az atomi tranzakciónak a részei, így „látszólag azonnal" végrehajtásra kerülnek.

Ahhoz, hogy a P_i folyamat egy `ír`(v) műveletet hajtson végre x-en, először végrehajt egy `beágyazott_olvas` műveletet, ami pontosan ugyanaz, mint a fent leírt `olvas`. A `beágyazott_olvas` eredményéből a P_i meghatározza a legnagyobb t *címkét*. Ezután $(v, t+1)$-et ír az x másolatainak többségébe. Mindezek a lépések–azaz a `beágyazott_olvas` és a másolatok írásának lépései–ugyanannak az atomi tranzakciónak a részei, így „látszólag

azonnal" végrehajtásra kerülnek.

17.2. lemma. . *A* TöbbségiSzavazás *algoritmus egy olvasható/írható atomi objektum.*

Bizonyításvázlat. Ellenőrizzük a 13.1.1. szakaszban megadott atomi objektum definíciójában szereplő feltételeket. A jólformáltság és a hibamentes befejeződés teljesülése könnyen látható. Az atomiság feltételének megmutatásához rögzítsük a TöbbségiSzavazás algoritmus egy tetszőleges α végrehajtási sorozatát (tetszőleges felhasználói automaták mellett). Legyen a Φ halmaz azon nem befejezett műveleteknek a halmaza, amelyek az alapul szolgáló tranzakció-megvalósításnak a sorrendi pontjaihoz vannak rendelve, és adoptálják a Φ-beli műveleteknek szóló válaszokat, valamint a tranzakció-megvalósítás sorrendi pontjait. Az összehúzódási tulajdonság fennállásának megmutatásához tudnunk kell, hogy minden `olvas` a sorrendben pontosan előtte álló `ír` értékét kapja meg, amennyiben ilyen létezik; ha nincs ilyen művelet, a kiindulási v_0 értéket kapja meg.

A bizonyítás kulcslépéseit a következők adják.

1. Az `ír` műveletek az $1, 2, \ldots$ *címkéket* a sorrendi pontjaik sorrendjében kapják.
2. Minden `olvas` és `beágyazott_olvas` egy sorrendben előtte álló `ír` által írt legnagyobb *címkét* kapja meg (illetve 0-t ha nincs ilyen) a hozzá tartozó értékkel együtt.

Ezek az állítások igazak, mivel minden `olvas` és `beágyazott_olvas` a másolatok többségét beolvassa, a legnagyobb *címke* a másolatok többségébe beíródik, és az összes ilyen módon értelmezett „többségnek" van közös eleme. □

Ezek után, ha az A közös memóriájú rendszer olvasható/írható közös változókat használ, definiáljuk úgy az A-n alapuló TöbbségiSzavazás algoritmust, hogy ugyanazokat a Q_i komponenseket tartalmazza, mint az EgyszerűKözösváltSzim algoritmus, TöbbségiSzavazásObjektum-okkal minden olvasható/írható közös változóhoz. Ekkor a 17.2 lemma alapján adódik a

17.3. tétel. . *Tegyük fel, hogy az A olvasható/írható közös változókat használ. Ekkor az A-n alapuló* TöbbségiSzavazás *algoritmus az A egy 0-szimulációja.*

Hibatűrés.. Habár a TöbbségiSzavazásObjektum algoritmus rugalmasságot biztosít az olvasásra vagy írásra kerülő többség kiválasztásában, általában nem teszi lehetővé egy atomi objektum hibatűrő megvalósítását az x-re. Ennek az az oka, hogy a szokásos tranzakció-megvalósítások nem hibatűrőek. Például egy olvasható/írható zárolási algoritmusban, egy `olvas` tranzakciót végrehajtó folyamat üzeneteket küldhet ki a másolatok többségének olvasásához, zárolva ezzel a másolatok többségét. Ezek után a folyamatban hiba történhet a zárolás feloldása előtt. Ez minden későbbi `ír` tranzakciót meggátol abban, hogy elérje a szükséges zárolt másolatokat. A gyakorlatban ez a feladat kezelhető időzítések használatával a folyamathibák detektálásához (ami nem tehető meg aszinkron hálózati modellben), és/vagy a hibatűrési követelmények csökkentésével.

17.1.3.. A folyamatok hibáit tűrő algoritmus

Ahogy már említettük, a 17.1.2. szakaszban leírt stratégiáknak nincsenek érdekes hibatűrő tulajdonságaik. Ebben a szakaszban bemutatjuk Attiya, Bar-Noy és Dolev ABD *algoritmusát*, amely korlátozott, f számú folyamatleállítási hiba mellett működik; a hálózatot megbízhatónak tételezzük fel. Feltételezzük, hogy az összes folyamatok száma, n szigorúan nagyobb, mint $2f$, azaz a folyamatok *többsége* nem hibázik. Csak a kizárólagosan írható/megosztottan olvasható olvasható/írható közös memória esetét vizsgáljuk.

Az ABD algoritmus lényegét egy f-hibás végrehajtást garantáló olvasható/írható atomi objektum megvalósítása adja, minden x olvasható/írható közös változóhoz. Ezt a megvalósítás az egyszerűség kedvéért azzal a feltételezéssel írjuk le, hogy az 1 kapun csak `ír`, míg a $2, \ldots, n$ kapukon csak `olvas` műveletek történnek; később ezt a megvalósítást kicsit módosítanunk kell az általános szimulációban való használathoz. Az algoritmus a TÖBBSÉGISZAVAZÁS és a 13.4.3. szakasz VITANYIAWERBUCH algoritmusából merít ötleteket. Az elképzelés lényege, hogy az `ír` műveleti eredmények a hálózat csomópontjainak többségében tárolásra kerülnek, az `ír` befejeződéséig.

ABDOBJEKTUM algoritmus (vázlatosan)

Mind az n folyamat az x egy másolatát kezeli, amely kezdetben az x kezdőértékével egyezik meg, továbbá egy kezdetben $\emptyset$ kezdőértékű, *címke* nemnegatív egészet.

Amikor az egyedi író folyamat `ír`(v)-t akar végrehajtani az x-re, akkor először t-nek azt a legkisebb *címkét* választja, ami még nincs `ír`-hez rendelve. Ezután az x helyi másolatát, illetve a helyi *címkét* állítja be v-re, illetve t-re, majd egy *(„ír”,v,t)*üzenetet küld minden más folyamatnak. Az üzenetet megkapó folyamatok ugyanezen módon frissítik helyi x másolatukat és annak *címkéjét*, amennyiben t nagyobb az aktuális *címke* értéknél; a folyamatok minden esetben nyugtát küldenek az író folyamatnak. Amint az író értesül róla (a nyugtákon, valamint saját helyi viselkedésének ismeretén keresztül), hogy a folyamatok többségének *címkéje* felvette a t értéket, *nyugtáz* értéket ad vissza.

Ha egy P_i folyamat *olvasni* akarja az x-et, *olvas* üzeneteket küld az összes többi folyamatnak, továbbá kiolvassa a saját x másolatának és *címkéjének* értékét. Egy ilyen üzenetet kapó folyamat a saját x, illetve *címke* legutolsó értékével válaszol. Ha P_i a folyamatok többségétől megkapta az x, illetve *címke* értékeket, felkészül az x legnagyobb t *címkéhez* tartozó v értékének visszaadására. Ennek végrehajtása előtt azonban P_i szétküldi a (v,t) értékpárt a folyamatok többségéhez: frissíti a saját x és *címke* értékét, és másodszor is üzeneteket küld ki a többi folyamatnak (az írót kivéve). Az üzenetet megkapó folyamatok ennek megfelelően frissítik helyi x másolatukat és annak *címkéjét*, amennyiben t nagyobb az aktuális *címke* értéknél; a folyamatok minden esetben nyugtát küldenek a P_i-nek. Amint a P_i értesül róla (a nyugtákon, valamint saját helyi viselkedésének ismeretén keresztül),

hogy a folyamatok többségének *címkéje* legalább t értékű, visszaadja a v értéket.

A kódot az alábbiakban közöljük. ABD_1 az író folyamat, $ABD_2, \ldots, ABD_n$ pedig olvasó folyamatok. Az egyszerűség kedvéért nem írjuk ki explicit módon a `megállít` műveleteket, amelyekről feltételezzük, hogy az EGYSZERŰKÖZÖSVÁLTSZIM algoritmusnál leírt módon vannak kezelve. Eltekintünk továbbá az x index explicit feltüntetésétől az egyes műveletekben. (A kód a részletes leírás nélkül is éppen elég hosszú.) Feltételezzük, hogy V az x értéktartománya, az x változó kezdeti értéke pedig v_0.

17.3. automata. ABDOBJEKTUM$_1$

Lenyomat:

Bemeneti:
- `ír`$(v)_1$, $v \in V$
- `fogad`$(\text{„}ír\_nyugtáz\text{”}, t)_{j,1}$, $t \in \mathbb{N}^+$, $j \neq 1$
- `fogad`$(\text{„}olvas\text{”}, u)_{j,1}$, $u \in \mathbb{N}^+$, $j \neq 1$

Kimeneti:
- `nyugtáz`$_1$
- `küld`$(\text{„}ír\text{”}, v, t)_{1,j}$, $v \in V$, $t \in \mathbb{N}^+$, $j \neq 1$
- `küld`$(\text{„}olvas\_nyugtáz\text{”}, v, t, u)_{1,j}$, $v \in V$, $t \in \mathbb{N}$, $u \in \mathbb{N}^+$, $j \neq 1$

Állapotok:

érték $\in V$, kezdetben v_0
címke $\in \mathbb{N}$, kezdetben 0
státus $\in \{$*tétlen, aktív*$\}$, kezdetben *tétlen*
száml $\in \mathbb{N}$, kezdetben 0
$\forall j \neq 1$:
küld_puffer(j), válaszok egy FIFO sora, kezdetben üres

Átmenetek:

`ír`$(v)_1$
Hatás:
érték $:= v$
címke $:=$ *címke* $+ 1$
státus $:=$ *aktív*
száml $:= 1$
$\forall j \neq 1$ do
adjuk hozzá („*ír*”,*v*,*címke*)-t a *küld_puffer*(j)-hez

`nyugtáz`$_1$
Előfeltétel:
státus $=$ *aktív*
száml $> \frac{n}{2}$
Hatás:
száml $:= 0$
státus $=$ *tétlen*

`küld`$(m)_{1,j}$
Előfeltétel:
m az első a *küld_puffer*(j)-ben
Hatás:
töröljük *küld_puffer*(j) első elemét

`fogad`$(\text{„}ír\_nyugtáz\text{”}, t)_{j,1}$
Hatás:
if *státus* $=$ *aktív* **and** $t =$ *címke* **then**
száml $:=$ *száml* $+ 1$

`fogad`$(\text{„}olvas\_nyugtáz\text{”}, t)_{j,1}$
Hatás:
adjuk hozzá („*olvas_nyugtáz*”,*érték*,*címke*,*u*)-t *küld_puffer*(j)-hez

Taszkok:
$\{\texttt{nyugtáz}_1\}$
$\forall j$:
$\{\texttt{küld}(m)_{1,j} : m$ egy válasz$\}$

- -

Vegyük észre, hogy a TÖBBSÉGISZAVAZÁS és aVITANYIAWERBUCH algoritmusokkal szemben az ABDOBJEKTUM algoritmusban egy új *címke* könnyen kiválasztható, mivel csak egy író van. Az alábbiakban az olvasó folyamatok kódját közöljük.

- -

17.4. automata. ABDOBJEKTUM$_i$ $(2 \leq i \leq n)$

Lenyomat:
Bemeneti:
$\texttt{olvas}_i$
$\texttt{fogad}(\text{„}ír\text{”}, v, t)_{1,i}$, $v \in V$, $t \in \mathbb{N}^+$
$\texttt{fogad}(\text{„}olvas\_nyugtz\text{”}, \text{v,t,u})_{j,i}$, $v \in V$, $t \in \mathbb{N}$, $u \in \mathbb{N}^+$, $j \neq i$
$\texttt{fogad}(\text{„}terj\_nyugtáz\text{”}, u)_{j,i}$, $u \in \mathbb{N}^+$, $j \notin \{1, i\}$
$\texttt{fogad}(\text{„}olvas\text{”}, u)_{j,i}$, $u \in \mathbb{N}^+$, $j \notin \{1, i\}$
$\texttt{fogad}(\text{„}terjeszt\text{”}, v, t, u)_{j,i}$, $v \in V$, $t \in \mathbb{N}$, $u \in \mathbb{N}^+$, $j \notin \{1, i\}$
Kimeneti:
v_i, $v \in V$
$\texttt{küld}(\text{„}ír\_nyugtáz\text{”}, t)_{i,1}$, $t \in \mathbb{N}^+$
$\texttt{küld}(\text{„}olvas\text{”}, u)_{i,j}$, $u \in \mathbb{N}^+$, $j \neq i$
$\texttt{küld}(\text{„}terjeszt\text{”}, v, t, u)_{i,j}$, $v \in V$, $t \in \mathbb{N}$, $u \in \mathbb{N}^+$, $j \notin \{1, i\}$
$\texttt{küld}(\text{„}olvas\_nyugtáz\text{”}, v, t, u)_{i,j}$, $v \in V$, $t \in \mathbb{N}$, $u \in \mathbb{N}^+$, $j \notin \{1, i\}$
$\texttt{küld}(\text{„}terj\_nyugtáz\text{”}, u)_{i,j}$, $u \in \mathbb{N}^+$, $j \notin \{1, i\}$

Állapotok:
érték $\in V$, kezdetben v_0
címke $\in \mathbb{N}$, kezdetben 0
válasz_ért $\in V$, kezdetben v_0
olvas_címke $\in \mathbb{N}$, kezdetben 0
státus $\in \{$*tétlen, aktív1, aktív2*$\}$, kezdetben *tétlen*
száml $\in \mathbb{N}$, kezdetben 0
$\forall j \neq i$:
küld_puffer(j), válaszok egy FIFO sora, kezdetben üres

Átmenetek:

```
olvas_i
  Hatás:
    olvas_címke := olvas_címke + 1
    státus := aktív1
    száml := 1
    ∀j ≠ i do
      adjuk hozzá
        („olvas”,olvas_címke)-t
        küld_puffer(j)-hez

küld(m)_{1,j}
  Előfeltétel:
    m az első a küld_puffer(j)-ben
  Hatás:
    töröljük küld_puffer(j) első elemét

fogad(„olvas_nyugtáz”, v, t, u)_{j,i}
  Hatás:
    if státus = aktív1 and
      u = olvas_címke then
        száml := száml + 1
      if t > címke then
        érték := v
        címke := t
      if száml > n/2 then
        válasz_érték := érték
        státus := aktív2
        száml := 1
        ∀j ∉ {1, i} do
          adjuk hozzá
            („terjeszt”,érték,címke,
            olvas_címke)-t
            küld_puffer(j)-hez

fogad(„terj_nyugtáz”, u)_{j,i}
  Hatás:
    if státus = aktív2 and
        u = olvas_címke
          then száml := száml + 1

v_i
  Előfeltétel
    státus = aktív2
    száml > n/2
    v = válasz_ért
  Hatás:
    száml := 0
    státus := tétlen

fogad(„ír”, v, t)_{1,i}
  Hatás:
    if t > címke then
      érték := v
      címke := t
      adjuk hozzá („ír_nyugtáz”,t)-t
        a küld_puffer(1)-hez

fogad(„olvas”, u)_{j,i}
  Hatás:
    adjuk („olvas_nyugtáz”,ért,címke,u)-t
      a küld_puffer(j)-hez

fogad(„terjeszt”, v, t, u)_{j,i}
  Hatás:
    if t > címke then
      érték := v
      címke := t
    adjuk hozzá („terj_nyugtáz”,u)-t
      a küld_puffer(j)-hez
```

Taszkok:

$\{$v$_i\}$
$\forall j$:{küld$(m)_{i,j}$: m egy üzenet}

- -

Ebben a kódban az *olvas_címke* annak nyomon követésére szolgál, hogy melyik nyugtázta tartozik az aktuális művelethez. A *válasz_ért*-t a visszaadandó érték megőrzésére használjuk terjesztés közben. Vegyük észre, hogy nem szükséges a válaszértéket eljuttatnunk az íróhoz, mivel az már rendelkezik a legfrissebb információval.

17.4. tétel. . *Az* ABDOBJEKTUM *algoritmus* $n > 2f$ *esetében egy* f*-hibás vég-*

rehajtást garantáló olvasható/írható atomi objektum.

Bizonyításvázlat. Az állítás a 13. fejezet VITANYIAWERBUCH és EGÉSZ-BLOOM bizonyításához hasonlóan mutatható meg. A jól megformáltság könnyen látható. Az f-hibás végrehajtás bizonyítása ugyancsak egyszerűen belátható, mivel minden művelet csak a folyamatok többségének részvételét igényli, és $n > 2f$. Így a szokásos módon, az atomi tulajdonság megmutatása a bizonyítás kulcsa. A 13.16. lemmát használjuk fel.

Legyen α az ABDOBJEKTUM algoritmus egy tetszőleges végrehajtási sorozata. A 13.10. lemma aszinkron hálózatra való átfogalmazásával, az általánosság megőrzése mellett feltételezhetjük, hogy az α nem tartalmaz befejezetlen műveleteket.

Legyen Π az α-ban lévő műveletek halmaza. Definiáljunk egy részben rendezést Π-n az alábbiak szerint. Először rendezzük el az `ír` műveleteket a végrehajtásuk sorrendjében, azaz a hozzájuk tartozó *címkék* szerint. Ezután helyezzük el az egyes `olvas` műveleteket közvetlenül azután az `ír` után, amelynek *címkéjét* az illető `olvas` tartalmazza, illetve ha nincs ilyen művelet, akkor az összes `ír` művelet elé.

A következő kulcstulajdonságokat kell megmutatnunk.

1. Ha egy *címke* $= t$ címkéjű π `ír` művelet befejeződik egy ϕ `olvas` megkezdése előtt, akkor a ϕ által tartalmazott *címke* legalább t.

   Ez azért van így, mert a π *címkéjét* a másolatok többsége megkapja, a ϕ olvassa a másolatok többségét, és a többségek metszete nem üres.

2. Ha egy π `olvas` művelet befejeződik egy ϕ `olvas` megkezdése előtt, akkor a ϕ által tartalmazott *címke* legalább akkora, mint a π által tartalmazott.

   Ez az állítás hasonlóan belátható, mivel π az információit a másolatok többségéhez eljuttatja.

A két tulajdonságot felhasználva könnyen megmutatható, hogy a 13.16. lemma által megkövetelt mind a négy feltétel teljesül, ami bizonyítja az atomi tulajdonságot. □

Az ABDOBJEKTUM algoritmus nyilvánvalóan módosítható úgy, hogy az 1 kapu helyett bármelyik i lehessen az `ír` kapu. Az algoritmus egyszerűen módosítható úgy is, hogy az `olvas` műveletek az egyetlen `ír` kapun is engedélyezve legyenek. Ezeket a módosításokat elvégezve továbbra is garantálható az f-hibás végrehajtás. Az A-n alapuló teljes ABD algoritmus, az EGYSZERŰKÖZÖSVÁLTSZIM és a TÖBBSÉGISZAVAZÁS algoritmusokhoz hasonlóan az TRANSZ(A) folyamataiból és az x közös változókhoz megadott atomi objektumokból konstruálható meg. Minden atomi objektum az ABDOBJEKTUM megfelelően módosított változata.

17.5. tétel. . *Tegyük fel, hogy az A egyszeri író/többszöri olvasó közös memóriát használ, és $n > 2f$. Ekkor az A-n alapuló* ABD *algoritmus az A egy f-szimulációja.*

Bizonyítás. A 17.4. és a 13.7. tétel segítségével. □

Korlátozott *címkék*.. Az ABD algoritmus korlátozatlan *címke* értékeket használ. Az algoritmus módosítható úgy, hogy a *címke* értékek korlátozva legyenek. Ennek megmutatását gyakorlatnak hagyjuk meg.

Alkalmazások.. Az ABD algoritmus segítségével számos érdekes hibatűrő és közös memóriájú algoritmus egyszeri író/többszöri olvasó regisztereken alapuló, osztott megvalósítása készíthető el. Például a 13. fejezetben közölt atomi fényképet és atomi többszöri író regiszteres algoritmusok átalakíthatók az ABD algoritmus segítségével ugyanazon objektumokat aszinkron küld/fogad hálózati modellben megvalósító algoritmusokká. Vegyük viszont észre, hogy míg az eredeti algoritmusok ezekben az esetekben várakozásmentes végrehajtást garantálnak, az átalakított változatok csak f hibát tűrnek el, ahol $n > 2f$.

17.1.4.. Egy megoldhatatlansági eredmény $\frac{n}{2}$ hiba esetében

Nem nehéz belátni, hogy az ABD algoritmus nem tűr el f hibát, ha $n \leq 2f$. Ennek az az oka, hogy ilyen számú folyamat hibája esetében a többi folyamat már nem tudja biztosítani a munka befejezéséhez szükséges többséget. Megmutatjuk, hogy ez a korlátozás elkerülhetetlen. Az alábbi kulcseredményben egy korlátozást adunk meg az olvasható/írható atomi objektum megvalósítások hibatűrésére aszinkron hálózatokban. Egy erősebb állításhoz jutunk, ha az eredményt küld/fogad rendszerek helyett üzenetszóró rendszerekre fogalmazzuk meg.

17.6. tétel. . *Legyen $n = m + p$, ahol $m, p \geq 1$, és tegyük fel, hogy $n \leq 2f$. Ekkor nincs olyan algoritmus az aszinkron üzenetszóró modellben (megbízható üzenetszóró csatornákkal), amely meg tudna valósítani egy f-hibás végrehajtást garantáló, m írót és p olvasót használó olvasható/írható atomi objektumot.*

Bizonyítás. Az ellentmondás megmutatásához tegyük fel, hogy létezik egy ilyen A algoritmus. Az ilyen indirekt bizonyításokban megszokott módon feltételezzük, hogy a felhasználók a lehető legkevésbé determináltak.

Legyen G_1 az $1, \ldots, n-f$, G_2 pedig az $n-f+1, \ldots, n$ elemek halmaza. A feltevés szerint $|G_1| \leq f$ és $|G_2| \leq f$.

Tekintsük a rendszer (A és a felhasználók) egy α_1 pártatlan végrehajtási sorozatát, amely az 1 kapun tartalmaz egy $\mathit{ír}(v)_1$ kérést, ahol $v \neq v_0$, és nincsenek más kérések. Tegyük fel továbbá, hogy `megállít` bemenetek jelennek meg pontosan a G_2-beli kapukon, és ezek az események rögtön a végrehajtás megkezdésekor bekövetkeznek; ez azt eredményezi, hogy a G_2-beli indexű folyamatok sosem hajtanak végre helyileg vezérelt műveleteket. Az f-hibás végrehajtáshoz az `ír` műveletnek egy megfelelő $\mathit{nyugtáz}_1$ művelettel kell befejeződnie. Legyen α_1' az α_1 előtagja a $\mathit{nyugtáz}_1$ végződéssel.

Tekintsünk most egy α_2 másik pártatlan végrehajtási sorozatot, amely az n kapun tartalmaz egy olvas_n kérést, és nincsenek további kérései. Tegyük fel továbbá, hogy `megállít` események jelennek meg pontosan a G_1-beli kapukon a végrehajtás megkezdésekor. Az f-hibás végrehajtáshoz az `olvas` műveletnek be kell fejeződnie, a visszaadott értéknek pedig v_0-nak kell lennie. Legyen α_2' az α_2 előtagja ezzel a v_0 végződéssel.

Konstruáljunk most egy olyan α véges végrehajtási sorozatot, ami nem elégíti ki az atomi tulajdonságot, ellentmondást eredményezve. Az α végrehajtási sorozat elégítse ki a következő feltételeket.

1. Az α megkülönböztethetetlen az α_1'-től a G_1-beli indexű folyamatokra.
2. Az α megkülönböztethetetlen az α_2'-től a G_2-beli indexű folyamatokra.
3. Az α-ban a $nyugtáz_1$ válaszesemény megelőzi az $olvas_n$ kérőeseményt.

Ez ellentmond az atomiság feltételének, amely szerint az `olvas` eseménynek az `ír` által írt v értéket kell visszaadnia a v_0 kezdőérték helyett.

Az α végrehajtási sorozat konstrukciója a következőképpen történik. Nem tartalmaz `megállít` eseményeket. Az α_1'-beli tevékenységekkel kezdődik, a G_2-beli indexű folyamatokban található `megállít` és `fogad` események kivételével. Mivel a G_2-beli folyamatok rögtön a kezdetkor hibáznak az α_1'-ben, ezért ezen események törlése továbbra is egy olyan végrehajtási sorozatot eredményez, amely megkülönböztethetetlen α_1'-től a G_1-beli folyamatokra. Az α végrehajtási sorozat ezután az α_2'-beli tevékenységekkel fejeződik be, a G_1-beli indexű folyamatokban található `megállít` és `fogad` események kivételével.

Így az α-ban, a G_1-, illetve G_2-beli folyamatok viselkedése független a másik csoportban lévő folyamatoktól. A G_1-beli folyamatok által szórt üzenetek nem jutnak el a G_2-beli folyamatokhoz, és fordítva. Könnyen ellenőrizhető, hogy α kielégíti a kívánt feltételeket. □

A 17.6. tételből következik, hogy minden rögzített n és f esetében, ahol $n \geq 2$ és $f \geq \frac{n}{2}$, nem létezik általános eljárás f-szimulációk készítéséhez n-folyamatból álló, közös memóriájú algoritmusokhoz, még ha az alapul szolgáló közös változók csak egyszeri író/egyszeri olvasó regiszterek is. Ennek belátásához vegyük észre, hogy minden ilyen n-hez létezik egy triviális várakozásmentes közös memóriájú A algoritmus, amely egy 1-író/$n-1$-olvasó olvasható/írható atomi objektumot valósít meg egy egyszerű 1-író/$n-1$-olvasó olvasható/írható regiszter segítségével. Az A egy f-szimulációja (ha létezik ilyen) egy olyan küld/fogad hálózati algoritmust eredményezne, ami egy f-hibás végrehajtású, 1-író/$n-1$-olvasó olvasható/írható atomi objektumot valósítana meg. (Ez a 13.9. következmény bizonyításához hasonlóan mutatható meg.) Ez viszont ellentmond a 17.6. tétel állításának.

17.2.. A hálózati modell transzformálása a közös memóriájú modellre

Most áttérünk az ellenkező irányú, az aszinkron hálózati modellből közös memóriájú modellbe történő transzformációk leírására. Ezek a transzformációk eltűrik a folyamatleállási hibákat: egy legfeljebb f folyamathibájú közös memóriájú rendszer szimulálni tud egy legfeljebb f folyamathibájú hálózatot (megbízható kommunikáció mellett). Ezúttal nincs különleges elvárás a hibák számára vonatkozóan – az ellentétes irányú transzformációkkal szemben, ezek a konstrukciók $n \leq 2f$ esetben is működnek. Ezen kívül ezek a konstrukciók sokkal egyszerűbbek, mint az ellentétes irányú transzformációk.

Annak az oka, hogy ezen konstrukciók egyszerűbb és erősebb eredményeket adnak az, hogy az aszinkron közös memóriájú modell bizonyos értelemben *hatékonyabb*, mint az aszinkron hálózati modell. A nagyobb hatékonyság a megbízható közös memória elérhetőségének köszönhető.

Ezek a transzformációk felhasználhatók aszinkron hálózati algoritmusok aszinkron közös memóriájú rendszerekben való futtatásához. Ez azonban valószínűleg kevésbé érdekes, mivel a közös memóriájú rendszert könnyebb programozni. Egy fontosabb felhasználási terület az aszinkron közös memóriájú modell megoldhatatlansági eredményeinek átvitele az aszinkron hálózati modellre. Például az az állítás, amely a megegyezés megoldhatatlanságát mondja ki a hibák fellépésekor, és amelyet közös memóriájú modellre a 12.8. tételben bizonyítottunk, kiterjeszthető a hálózati modellre ezen transzformációkkal.

Két transzformációt mutatunk be: egyet a küld/fogad, egyet pedig az üzenetszóró rendszerekre.

17.2.1.. Küld/fogad rendszerek

Tegyük fel, hogy adott a 14. fejezetben közölt modell aszinkron küld/fogad A rendszere, amely az irányított G gráfon alapul, P_i $(1 \leq i \leq n)$ folyamatokkal és $C_{i,j}$ megbízható FIFO csatornákkal. Mint korábban, minden $\texttt{megállít}_i$ esemény rögtön letiltja P_i összes taszkját, de nincs hatása a csatornákra.

Az általános feladat (beleértve a hibatűrési követelményeket is) egy olyan közös memóriájú, n folyamattal rendelkező, egyszeri író/egyszeri olvasó közös regisztereket használó B rendszer elkészítése, amely az A-t „szimulálja". A-t ugyanabban az értelemben kell szimulálnia, mint az ellentétes irányú transzformációk esetében. B tetszőleges α végrehajtási sorozatára, tetszőleges U_i felhasználók mellett, létezni kell A egy α' végrehajtási sorozatának ugyanazon felhasználókkal a következő feltételek teljesülése mellett.

1. α és α' megkülönböztethetetlen az U-ra nézve.
2. Minden i-re, egy $\texttt{megállít}_i$ pontosan akkor jelenik meg az α-ban, ha egy $\texttt{megállít}_i$ megjelenik az α'-ben.

Ha továbbá α egy pártatlan végrehajtási sorozat, és minden i, amire $\texttt{megállít}_i$ következik be az α-ban, I-be esik, akkor α' is egy pártatlan végrehajtási sorozat. Ha B ilyen módon szimulálja az A-t egy adott I-re, akkor B-t az A *I-szimulációjának* nevezzük. Ha B az A I-szimulációja minden olyan I-re, amelyre $|I| \leq f$, akkor B-t az A *f-szimulációjának* nevezzük.

Megadjuk az EgyszerűKFSzim algoritmust, ami tetszőleges számú hibára működik, azaz egy n-szimuláció.

EgyszerűKFSzim algoritmus (vázlatosan)

Az alapul szolgáló irányított G gráf minden irányított (i,j) élére B tartalmaz egy egyszeri író/egyszeri olvasó olvasható/írható $x(i,j)$ közös változót, amelyet az i folyamat írni, a j folyamat pedig olvasni tud. $x(i,j)$ az üzenetek egy sorát tartalmazza, amely kezdetben üres. Az i folyamat csak üzeneteket ad a sorhoz; az üzenetek sohasem kerülnek törlésre.

B i folyamata A P_i folyamatát szimulálja. A felhasználói felület lépéseinek és P_i belső lépéseinek szimulációja közvetlenül történik. P_i egy $\texttt{küld}(m)_{i,j}$ műveletének szimulálásához A i folyamata hozzáadja az m üzenetet a sor végéhez az $x(i,j)$ változóban. (Ezt egyetlen `ír` művelettel megteheti a sor két helyi másolatának használata mellett.) Az i folyamat időről-időre ellenőrzi az összes „bejövő" $x(j,i)$ változót, hogy érkeztek-e azokba újabb üzenetek az utolsó ellenőrzés óta. Ha igen, az i folyamat ugyanúgy kezeli azokat az üzeneteket, mint P_i.

A kódot az alábbiakban közöljük. Ügyeljünk rá, hogy minden folyamathoz több taszk tarozik. Az $\texttt{ellenőr}(j)_i$ kódjában a $\texttt{fogad}(M)_{j,i}$ jelölést a $\texttt{fogad}(m_1)_{j,i}, \texttt{fogad}(m_2)_{j,i}, \ldots$ műveletsorozat rövidítéseként használjuk, ahol M az $m_1, m_2, \ldots$ üzenetek sorozata. Ebben a kódrészben az M sorozat az i folyamat utolsó ellenőrzése óta az $x(j,i)$ változóban megjelent új üzeneteket tartalmazza.

17.5. algoritmus. EGYSZERŰKFSZIM

Közös változók:
G minden (i,j) élére: $x(i,j)$, üzenetek egy FIFO sora, kezdetben üres

i műveletei:
Mint P_i-nek, kivéve:
Bemeneti:
Hagyjunk ki minden `fogad` műveletet.
Kimeneti:
Hagyjunk ki minden `küld` műveletet.
Belső:
$\texttt{küld}(m,j)_i$ minden $\texttt{küld}(m)_{i,j} \in \mathit{ki}(P_i)$-re
$\texttt{ellenőriz}(j)_i$ minden $j \in \mathit{be\_szomszédok}$-ra

i állapotai:
$\mathit{foly\_áll} \in \mathit{áll}(P_i)$, kezdetben egy kezdőállapot
minden $j \in \mathit{ki\_szomszédok}$-ra:
$\mathit{ki\_üzenetek}(j)$, üzenetek egy FIFO sora, kezdetben üres
minden $j \in \mathit{be\_szomszédok}$-ra:
$\mathit{be\_szomszédok}(j)$, üzenetek egy FIFO sora, kezdetben üres
$\mathit{feld\_üzenetek}(j)$, üzenetek egy FIFO sora, kezdetben üres

i átmenetei:
π, P_i egy bemenete $\neq \texttt{fogad}$
Hatás:
$\mathit{foly\_áll} :=$ bármely s, amelyre $(\mathit{foly\_áll}, \pi, s) \in \mathit{átm}(P_i)$

π, P_i egy helyileg vezérelt művelete $\neq \texttt{küld}$
Előfeltétel:
π engedélyezett $\mathit{foly\_áll}$-ban
Hatás:
$\mathit{foly\_áll} :=$ bármely s, amelyre $(\mathit{foly\_áll}, \pi, s) \in \mathit{átm}(P_i)$

`küld`$(m,j)_i$
Előfeltétel:
`küld`$(m)_{i,j}$ engedélyezett *foly_áll*-ban
Hatás:
adjuk hozzá $m$et *ki_üzenetek*(j)-hez
$x(i,j) :=$ *ki_üzenetek*(j)
foly_áll := bármely s, amelyre
$(\textit{foly\_áll}, \texttt{küld}(m)_{i,j}, s) \in \textit{átm}(P_i)$

`ellenőr`$(j)_i$
Előfeltétel:
igaz
Hatás:
feld_üzenetek(j) := *be_üzenetek*(j)
be_üzenetek$(j) := x(j,i)$
foly_áll := bármely olyan
végrehajtási részsorozat utolsó állapota,
amelynek kezdete *foly_áll* és a
`fogad`$(M)_{j,i}$ műveletsorozat,
ahol *feld_üzenetek*$(j) \cdot M =$ *be_üzenetek*(j)

i **taszkjai:**
Mint P_i-nek, kivéve:
cseréljük minden `küld`$(m)_{i,j}$-t `küld`$(m,j)_i$-re
$\forall j$: adjuk hozzá {`ellenőriz`$(j)_i$}-t

- -

Ezek után nem nehéz belátni, hogy a szimuláció helyes.

17.7. tétel. . *Ha A egy aszinkron küld/fogad rendszer megbízható FIFO küld/fogad csatornákkal, akkor az* EgyszerűKFSzim *algoritmus A egy n-szimulációja.*

Bizonyítás. A bizonyítást meghagyjuk gyakorlatnak (lásd 17-13. gyakorlat). □

17.2.2.. Üzenetszóró rendszerek

Az EgyszerűKFSzim algoritmushoz hasonló konstrukcióval szimulálhatunk egy megbízható üzenetszóró csatornával rendelkező aszinkron üzenetszóró rendszert. A szimulációra vonatkozó helyességi feltételek ugyanazok, mint a küld/fogad rendszerekre. A legfontosabb különbség, hogy az új szimuláció egyszeri író/egyszeri olvasó regiszterek helyett egyszeri író/*többszöri olvasó* regisztereket használ.

EgyszerűÜzenetSzóróSzim algoritmus
B minden i-re ($1 \le i \le n$) tartalmaz egy egyszeri író/többszöri olvasó $x(i)$ osztott változót, amelyet az i folyamat írni, minden más folyamat (az i-t is beleértve) pedig olvasni tud. Egy üzenetsort tartalmaz, amely kezdetben üres.

Mint korábban, a B i folyamata az A P_i folyamatát szimulálja a felhasználói felület lépéseinek és a P_i belső lépéseinek közvetlen szimulálásával. A P_i egy `szór`$(m)_i$ műveletének szimulálásához A i folyamata hozzáadja az m üzenetet a sor végéhez az $x(i)$ változóban. Az i folyamat időről-időre ellenőrzi az $x(j)$ változókat (beleértve az $x(i)$-t is), hogy érkeztek-e újabb üzenetek. Ha igen, az i folyamat ugyanúgy kezeli azokat az üzeneteket, mint P_i.

17.8. tétel. . *Ha A egy aszinkron üzenetszóró rendszer megbízható üzenetszóró csatornával, akkor az* EgyszerűÜzenetSzóróSzim *algoritmus A egy n-szimulációja.*

17.2.3.. Megegyezés megoldhatatlansága aszinkron hálózatokban

A 17.8. tétel segítségével bizonyítható be a 12. fejezetben közölt alapvető megegyezési feladat aszinkron hálózatban való megoldhatatlansága, *még abban az esetben is, ha a hálózat megbízható üzenetszórást garantál, garantáltan legfeljebb egy folyamathiba történhet és az egyetlen hibatípus a leállás!* Ez a megoldhatatlansági eredmény az aszinkron hálózatok számítási képességeinek egy alapvető korlátozását adja.

Ezt az eredményt összevethetjük a 6. fejezetben a szinkron hálózati modellre vonatkozó megállási megegyezési problémára kapott eredményekkel. Abban az esetben a feladat megoldható, bár a problémának van egy nem triviális belső időigénye, amely függ az elfogadható hibák számától. Az alsó időkorlátra vonatkozó 6.33. tétel bizonyítása azon alapul, hogy egy folyamat *üzenetszórás közben is* leállhat. Az aszinkron modellben ezzel szemben a megoldhatatlansági eredmény még a részleges üzenetszórások lehetősége nélkül is fennáll.

A 12.1. alfejezetben megadott állítást használjuk az 1-hibás végrehajtású megegyezési problémához. (Az állítás megfogalmazható történettulajdonságok segítségével is, amely így értelmet nyer az aszinkron hálózati rendszerekre és a közös memóriájú rendszerekre is.)

17.9. tétel. . *Nem létezik olyan algoritmusegy megbízható üzenetszóró csatornával rendelkező aszinkron üzenetszóró modellben, amely megoldaná a megegyezési problémát és 1-hibás végrehajtást garantálna.*

Bizonyítás. Indirekt módon tegyük fel, hogy létezik egy ilyen A algoritmus. Ekkor a 17.8. tétel megad egy olyan B algoritmust az egyszeri író/többszöri olvasó közös memóriájú modellben, ami az A egy n-szimulációja. Az n-szimuláció definíciója alapján B megoldja a megegyezési problémát és 1-hibás végrehajtást garantál. Ez viszont ellentmond a 12.8. tételnek, ami a megegyezési feladat megoldhatatlanságát állítja az olvasható/írható közös memóriájú modellben. □

17.3.. Megjegyzések a fejezethez

Párhuzamos folyamatokat vezérlő algoritmusokhoz felhasználható atomi tranzakciók megvalósításához ajánlható a Lynch, Merritt, Weihl és Fekete [207], valamint a Bernstein, Hadzilacos és Goodman [50] által írt könyv.

A TöbbségiSzavazás algoritmus Giffordnak [137] köszönhető. Ezt Herlihy [154, 149], valamint Goldman és Lynch [140] általánosította; ez utóbbi kiterjesztés [207]-ben is megtalálható.

AzABD algoritmust Attiya,Bar-Noy és Dolev [18] készítette. Cikkük tartalmaz egy, Israeli és Li [162] ötletén alapuló, korlátozott *címkéket* használó algoritmust is, továbbá az ABD szimuláció egyéb alkalmazásait. Az $n \leq 2f$ esetre vonatkozó megoldhatatlansági eredmény Bracha és Toueg [56], valamint Attiya, Bar-Noy, Dolev, Peleg és Reischuk [20] hasonló bizonyításának átdolgozása.

A megegyezés megoldhatatlanságát hibázó aszinkron hálózatokra kimondó 17.9. tétel Fischer,Lynch ésPaterson [123] nevéhez fűződik. Az eredményt a hálózati modell, nem pedig a bemutatott transzformáció segítségével bizonyították be közvetlenül.

17.4.. Gyakorlatok

17-21. Bizonyítsuk be a 17.1. tétel bizonyítási vázlatában szereplő segédállítást – azaz, hogy minden x-re, az $R_{x,i}$ automaták a közöttük lévő csatornákkal együtt (a `küld` és `fogad` műveletek elrejtésével) egy megfelelő típusú és felületű B_x atomi objektumot alkotnak.

17-22. Mondjunk ki és bizonyítsunk be egy állítást, annak a B rendszernek az időbonyolultságára, amelyet az EgyszerűKözösVáltSzim algoritmus A közös memóriájú rendszerre és annak időbonyolultságára alkalmazva kaptunk. Ügyeljünk a használt feltételek gondos ellenőrzésére.

17-23. Legyen B egy aszinkron hálózati algoritmus, amelyet az EgyszerűKözösVáltSzim 10.5.2. szakaszban szereplő PetersonnnFoly algoritmusra való alkalmazásával kaptunk. Adjunk lehető legjobb felső korlátot B időigényére, pontosabban minden $\texttt{Pr}_i$ esemény idejéből a megfelelő $\texttt{Be}_i$ esemény idejére. Hogyan hasonlítható ez össze a 17-2. gyakorlatban kapott általános időigénnyel?

17-24. *Kutatási kérdés.* Mondjunk ki és bizonyítsunk be egy állítást arra vonatkozóan, hogy mi garantálható az EgyszerűKözösVáltSzim transzformáció olyan véletlenített közös memóriájú rendszerre való alkalmazásakor, mint például a 11.4. alfejezetben szereplő LehmannRabin algoritmus.

17-25. Adjunk meg előfeltétel/hatás kódot a 17.1.2. szakaszban vázolt olvasható/írható zárolási algoritmusra, egyszeri író/többszöri olvasó közös memóriájú algoritmusok aszinkron hálózatban való szimulálásához. (Ez a vázlat a TöbbségiSzavazásObjektum algoritmus leírása előtti bekezdésekben van kifejtve.) Egy x közös változó minden olvasójának az x egy helyi másolatát kell kezelnie és olvasnia (ha az elérhető). Az írónak az egyes másolatok írását egykétfázisú zárolási protokollal kell megoldania. Garantálni kell minden művelet befejeződését. Mondjunk ki és bizonyítsunk be egy helyességi állítást.

17-26. Általánosítsuk a 17-5. gyakorlatnál kapott eredményt többszöri író/többszöri olvasó közös memóriájú algoritmusokra.

17-27. Tekintsük a 10.7. alfejezetben szereplő kölcsönös kizárási VáróTerem algoritmus alábbi két módon való transzformálását aszinkron hálózatban futáshoz.

(a) Az EgyszerűKözösVáltSzim használatával.

(b) A 17-5. gyakorlatban szereplő kétfázisú zárolási stratégia használatával. Hasonlítsuk össze az eredményül kapott két algoritmus idő- és kommunikációs bonyolultságát.

17-28. Általánosítsuk a TÖBBSÉGISZAVAZÁSOBJEKTUM algoritmust úgy, hogy az `olvas` műveletek a másolatok többsége helyett a másolatok egy *olvasási elegendőségét*, az `ír` műveletek pedig a másolatok egy *írási elegendőségét* érjék el. Az olvasási és írási elegendőségnek nem kell szigorúan többséginek lennie; milyen feltételeket kell kielégíteniük? Az algoritmust előfeltétel/hatás jelöléssel írjuk le, és mutassuk meg a helyességét.

17-29. Szükséges az ABDOBJEKTUM megvalósításában szereplő olvasó kód „terjesztési fázisa"? Ismertessük, ha az a véleményünk, hogy az algoritmus működne nélküle, vagy adjunk ellenpéldát.

17-30. Terjesszük ki az ABDOBJEKTUM algoritmust úgy, hogy egy többszöri író/többszöri olvasó, f-hibás befejeződést garantáló olvasható/írható atomi objektumot valósítson meg, ha $n > 2f$. Mutassuk meg, hogyan olvasztható be ez a kiterjesztés a közös memóriájú modell hibatűrő aszinkron hálózattal való szimulálásába, többszöri író/többszöri olvasó közös regiszterekkel.

17-31. Módosítsuk úgy az ABDOBJEKTUM algoritmust, hogy korlátlan helyett korlátos *címkéket* használjon. (*Útmutatás.* Nem elég csak az egészeket mod k tekinteni valamilyen k-val; szükség van egy érdekesebb szerkezettel rendelkező D véges adattípusra. A [162] tartalmaz egy ilyen működő adattípust. Az írónak a D adattípustól függően olyan „nagyobb" *címkéket* kell választania, amelyre minden lassabb folyamatok által kezelt korábbi *címkék* elérhetők lesznek az új *címkéknél* „kisebb" folyamatok által. Vagyis amikor az író kiválaszt egy új *címkét*, figyelemmel kell lennie a folyamatok által kezelhető összes *címkére*. Ahhoz, hogy az író nyomon tudja követni ezt az eljárást, minden helyi *címkét* módosító folyamatnak meg kell győződnie arról, hogy a folyamatok többsége tudomást szerzett az új *címke* értékről. Így az író mindig megtudhatja a folyamatok lehetséges *címke* értékeit, kikérve a folyamatok többségétől ezt az információt. További segítségért lásd [18].)

17-32. Mondjunk ki és bizonyítsunk be egy, a 17.6. tételhez hasonló állítást egy fénykép atomi objektum megvalósításáról egy aszinkron hálózatban, ha $n \leq 2f$.

17-33. Bizonyítsuk be a 17.7. tételt.

17-34. Adjunk meg előfeltétel/hatás kódot az EGYSZERŰÜZENETSZÓRÓSZIM algoritmusra az EGYSZERŰKFSZIM algoritmusra adott módon. Bizonyítsuk be a helyességét (17.8. tétel).

18. fejezet

Logikai idő

Ebben a fejezetben az aszinkron hálózat programozásának egyszerűbbé tételére szolgáló fő módszereink közül a harmadikat mutatjuk be: bevezetjük a *logikai idő* fogalmát. Aszinkron hálózati modellünkben nincs beépített *valós idő* fogalom, azonban a logikai idő fogalma speciális protokollok révén bevezethető. A logikai idő néha a valós idő helyett használható, azokban az esetekben, amikor a rendszer felhasználóit nem érdekli a különböző hálózati helyeken történő események egymáshoz viszonyított sorrendje.

18.1.. Logikai idő aszinkron hálózatokra

Az alapgondolat szerint egy A aszinkron hálózati rendszer egy végrehajtásának minden eseményéhez hozzárendelünk egy „logikai időt", amely egy rögzített, teljesen rendezett T halmaz eleme.[1] Tipikusan, ez a T halmaz vagy a nemnegatív egész, vagy a nemnegatív valós számok halmaza (esetleg más típusú értékekkel együtt, mint amilyenek a megkülönböztetésre használt folyamatindexek). Ezen logikai időknek nem kell, hogy bármilyen köze legyen a valós időhöz, azonban a különböző események logikai idejének tekintettel kell lennie az A rendszerben levő események közötti összes lehetséges függőségére, mint ahogyan azt a 14.1.4. szakaszban már leírtuk. Ezen feltételezések mellett bebizonyítható, hogy a logikai idejű folyamatkiosztás „olyannak tűnik", mint a valós idejű folyamatkiosztás.

A logikai időt külön vizsgáljuk a küld/fogad és az üzenetszóró rendszerekre. A fejezet során azt feltételezzük, hogy a csatornák a 14. fejezetben definiált *univerzális* csatornák. A hibákkal nem foglalkozunk.

18.1.1.. Küld/fogad rendszerek

Tekintsünk egy univerzális megbízható FIFO küld/fogad csatornával rendelkező aszinkron küld/fogad hálózati rendszert. Feltételezzük, hogy az ennek megfelelő G hálózati gráf egy tetszőleges, erősen összefüggő irányított gráf. Emlékezzünk

[1] T-nek ki kell elégítenie az alábbi technikai feltételt: léteznie kell egy T elemeiből álló $t_1, t_2, \ldots$ növekvő sorozatnak úgy, hogy minden $t \in T$-nek valamely t_i felső korlátja.

vissza, hogy egy ilyen rendszer eseményei a következő típusúak lehetnek: felhasználói felület események, amelyek segítségével a folyamatautomaták a rendszer felhasználóival kommunikálnak, `küld` és `fogad` események, melyek segítségével a folyamatautomaták kapcsolatba lépnek a csatornaautomatákkal, és a folyamatautomaták belső eseményei. (A csatornák belső eseményeivel nem kell foglalkoznunk, mivel az általunk használt konkrét univerzális csatornáknak nincsenek belső eseményei.)

Legyen α egy A aszinkron küld/fogad hálózati rendszer egy végrehajtási sorozata. Ekkor α *logikai idejű kiosztása* egy olyan hozzárendelés, amely α minden eseményéhez egy T-beli értéket rendel az α-beli események közötti összes lehetséges függőséggel „ellentmondásmentes" módon. Közelebbről ez a következő négy tulajdonságot követeli meg.

1. Ugyanaz a logikai idő nem rendelhető két különböző eseményhez.
2. Az egyes folyamatok eseményeinek logikai ideje, az α-ban vett előfordulási sorrendjüknek megfelelően, szigorúan növekvő.
3. Minden `küld` esemény logikai ideje szigorúan kisebb, mint a hozzá tartozó `fogad` eseményé.
4. Bármely konkrét $t \in T$-re csak véges sok esemény rendelkezhet t-nél kisebb logikai idővel.

A 2. és 3. tulajdonság maga után vonja, hogy a logikai idők sorrendjének összhangban kell lennie a 14.1.4. szakaszban meghatározott $\rightarrow_{\texttt{történet}(\alpha)}$ rendezéssel. Azt azonban megengedjük, hogy különböző folyamatok bizonyos eseményeinek logikai ideje az α-beli sorrendjükhöz képest fordított sorrendben legyen rendezve.

Azt állítjuk, hogy bármely logikai idejű kiosztás „úgy néz ki", mint a hálózat minden folyamatának egy valós idejű kiosztása. Speciálisan, bármely *log_idő* logikai idejű kiosztással ellátott α pártatlan végrehajtás minden folyamat számára olyannak tűnik, mint egy másik, α' pártatlan végrehajtás, amelyben a *log_idő*-k valós időkként viselkednek – azaz amelyben az események bekövetkezésének sorrendje megegyezik *log_idő*-edik sorrendjével.

18.1. tétel. . *Legyen α az univerzális megbízható FIFO csatornákkal rendelkező A küld/fogad hálózati rendszer egy pártatlan végrehajtása, és legyen log_idő egy α-hoz tartozó logikai idejű kiosztás. Ekkor A-nak létezik egy másik α' pártatlan végrehajtási sorozata, amelyre igaz, hogy:*

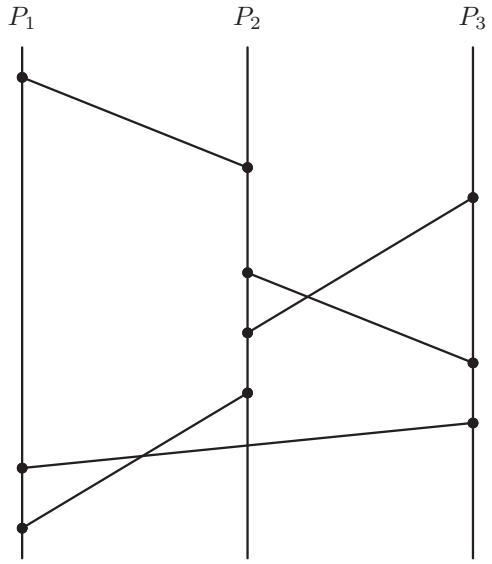
1. *α' ugyanazokat az eseményeket tartalmazza mint α;*
2. *Az α' eseményei az α-ban lévő log_idő-ik sorrendjében fordulnak elő;*
3. *α' megkülönböztethetetlen α-tól minden folyamatautomata számára.*[2]

A 18.1. tétel azt mondja ki, hogy minden egyes konkrét folyamat eseménysorrendjének α-ban és α'-ben meg kell egyeznie. Megengedi azonban a különböző folyamatok eseményeinek átrendezését.

[2]Itt a „megkülönböztethetetlen" 8.7. alfejezetben adott formális definícióját használjuk.

Bizonyítás. Legyen γ egy olyan sorozat, melyet az α eseményeinek a hozzájuk rendelt *log_ idő*-k sorrendjében történő átrendezésével kapunk. A logikai idő definíciójának 1. és 4. tulajdonsága miatt ilyen sorozat egyértelműen létezik. Ezek után a 14.2. következmény segítségével a szükséges α' pártatlan végrehajtási sorozat létezése bizonyítható. A 14.2. következmény alkalmazásánál a folyamatok által végzett összes műveletet külső műveletnek tekintjük (azaz újraosztályozzuk őket). A logikai idő definíciójának 2. és 3. tulajdonságából következik, hogy az újrarendezés megőrzi a $\rightarrow_{\texttt{történet}(\alpha)}$-t. Éppen ez szükséges a 14.2. következményhez. □

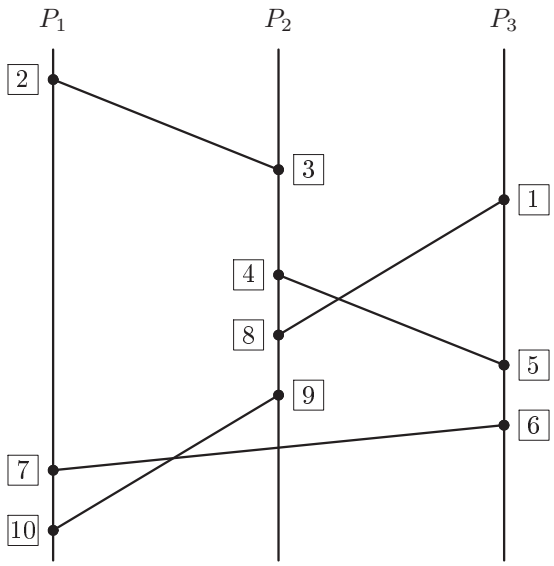
18.1.1. példa. Küld/fogad diagram
Tekintsünk egy egy három csúccsal rendelkező, teljes, irányítatlan gráfon alapuló A küld/fogad rendszert. Tekintsük az A egy α végrehajtási sorozatát, melyben az üzeneteket a 18.1. ábra szerint küldik és fogadják.



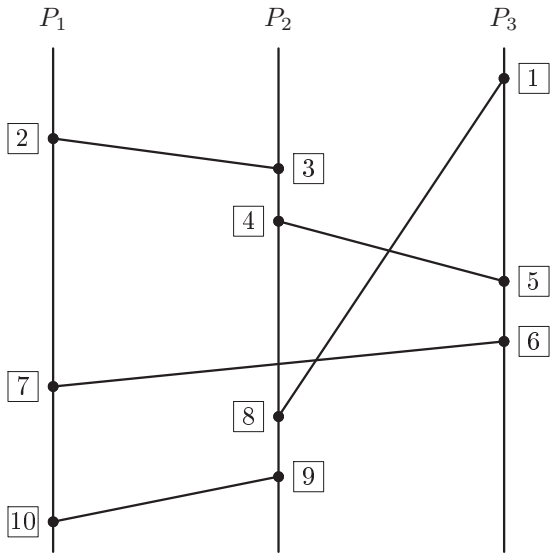
18.1.. ábra. Az α végrehajtási sorozat küld/fogad diagramja.

Ebben a *küld/fogad diagramban* minden folyamat végrehajtási sorozatát egy függőleges vonal jelöli, amelyben az idő lefelé halad. A pontok jelölik a `küld` és a `fogad` eseményeket, és minden ferde vonal egy adott üzenet `küld` eseményét kapcsolja össze annak `fogad` eseményével. Itt nem írunk le más eseményeket, azaz sem a folyamatok belső eseményeit, sem pedig azokat az eseményeket, amelyekkel a folyamatok kommunikálnak a felhasználókkal. Ezeket a függőleges vonalakon elhelyezett további pontokkal ábrázolhatnánk. A 18.2. ábrán egy α-hoz tartozó *log_ idő* logikai idejű kiosztás látható (feltételezve, hogy α csak `küld` és `fogad` eseményeket tartalmaz). Mivel az idő lefelé halad, ezért az *log_ idő* sorrendje nem egyezik meg α eseményeinek sorrendjével, azonban összhangban van az α-ban levő események közötti összes lehetséges függőséggel.

A 18.3. ábra az α eseményeinek a hozzájuk rendelt *log_ idő*-k sorrendje szerinti átrendezését mutatja be, amely α'-t adja a 18.1. tételben leírtak szerint. Megjegyezzük, hogy minden folyamat esetében az események sorrendje α-ban és α'-ben megegyezik.



18.2.. ábra. α egy logikai idejű kiosztása.



18.3.. ábra. Az α' átrendezett végrehajtási sorozat küld/fogad diagramja.

Figyeljük meg ezen alfejezet gondolatai és a helyi és globális szinkronizátorokkal kapcsolatban a 16.2. alfejezetben leírtak közötti szoros párhuzamot! Mindkét esetben egy függőségi sorrendet definiálunk egy végrehajtási sorozat eseményein,

amely magában foglalja az események közti összes lehetséges függőséget. A végrehajtási sorozat eseményei ezek után mindkét esetben átrendezésre kerülnek, megőrizve az összes függőséget, de az idő általános fogalmának (szinkron menetek vagy logikai idő) megfelelően kiigazítva őket. (A helyi szinkronizátor és a logikai idő fogalmát használjuk annak bemutatására, hogy ez lehetséges.) A következtetés mindkét esetben az, hogy az újrarendezett és az eredeti végrehajtási sorozat helyileg nem megkülönböztethető. Így az eredeti végrehajtási sorozat minden résztvevője úgy érzi, mintha globálisan szinkronban működnének.

18.1.2.. Üzenetszóró rendszerek

Az univerzális megbízható üzenetszóró csatornákkal rendelkező megbízható aszinkron üzenetszóró rendszerek számára is definiálhatunk logikai időt. Ebben az esetben az események a felhasználói felület eseményei, az `üzen_szór` és `fogad` események, valamint a folyamatok belső eseményei lehetnek.

Legyen α egy aszinkron üzenetszóró rendszer végrehajtási sorozata. *α logikai idejű kiosztása* egy olyan hozzárendelés, amely α minden eseményéhez egy T-beli értéket rendel úgy, hogy az a 3. tulajdonság kivételével ugyanazokat a feltételeket elégítse ki, mint a küld/fogad rendszerek esetében. A 3. tulajdonság helyett pedig tekintsük a következőt:

3'. bármely `üzen_szór` esemény logikai ideje szigorúan kisebb, mint a hozzá tartozó `fogad` eseményé.

Mint a küld/fogad rendszerek esetében, itt is igaz az alábbi

18.2. tétel. . *Legyen α az univerzális megbízható üzenetszóró csatornával rendelkező A üzenetszóró rendszer egy pártatlan végrehajtási sorozata, log_idő pedig legyen egy α-hoz tartozó logikai idejű kiosztás. Ekkor A-nak létezik egy másik, α' pártatlan végrehajtási sorozata, melyre:*

1. *α' ugyanazokat az eseményeket tartalmazza mint α;*
2. *Az α' eseményei az α-ban lévő log_idő-ik sorrendjében fordulnak elő;*
3. *α' megkülönböztethetetlen α-tól minden folyamatautomata számára.*

Bizonyításvázlat. Hasonló a 18.1. tétel bizonyításához, de most a 14.4. következményt kell használni. Ennek kidolgozását meghagyjuk gyakorlatnak (lásd 18-1. gyakorlat). □

18.2.. Logikai idő hozzáadása az aszinkron algoritmusokhoz

Az előző alfejezetben definiáltuk a logikai időt az aszinkron küld/fogad és üzenetszóró rendszerekre. Most két algoritmust adunk arra vonatkozóan, hogyan lehet egy adott A aszinkron küld/fogad hálózati algoritmus eseményeihez logikai időket generálni. Ezen algoritmusok mindegyike valójában egy algoritmus transzformáció, amely az adott A algoritmust egy új, ugyanazzal az irányított hálózati gráffal

rendelkező aszinkron küld/fogad $L(A)$ algoritmussá „transzformálja". A transzformáció folyamatról folyamatra működik, és az $L(A)_i$-t (az $L(A)$ rendszer i-edik folyamatát) A_i-vel (az A rendszer i-edik folyamatával) fejezi ki. Az $L(A)$-ban lévő folyamatok együttműködnek annak érdekében, hogy valahogyan „szimulálják" az A pártatlan végrehajtási sorozatát, ahol minden $L(A)_i$ a hozzátartozó A_i-t szimulálja. Valahányszor $L(A)$ egy folyamata A egy lépését szimulálja, „generál" egy logikai idő értéket. Az a tény, hogy néhány fogalmat idézőjelbe tettünk (azaz „transzformálja", „szimulálja", „generálja"), azt mutatja, hogy nem adunk hozzájuk egy világos és egyértelmű jelentést, hanem különböző helyzetekben kissé eltérő módon értelmezzük őket.

Mindkét itt megadásra kerülő transzformáció módosítható úgy, hogy üzenetszóró rendszerekben is használhatók legyenek.

18.2.1.. Az óra előreállítása

A következő egyszerű algoritmus-transzformáció egy adott A aszinkron küld/fogad hálózati algoritmus egy végrehajtási sorozatának logikai idejeit állítja elő. Ezt felfedezője, Lamport nyomán LamportIdő *transzformációnak* nevezzük. Ez a helyi órák karbantartásán alapul, az üzenetek megérkezésekor előreállítva őket, azért, hogy megfelelően szinkronban tartsuk őket. A T logikai idő tartomány olyan (c, i) párok halmaza, ahol c nemnegatív egész, i pedig egy folyamatindex. A párok rendezése lexikografikusan történik.

> **LamportIdő transzformáció (vázlatosan)**
> A LamportIdő$(A)_i$ folyamat karbantartja A_i állapotát, plusz egy *óra* nevű helyi változót, amely nemnegatív egész értékeket vesz fel, kezdetben 0-át. Az *óra* változó értéke minden egyes, az LamportIdő(A_i) folyamatban bekövetkező esemény hatására (beleértve a felhasználói felület eseményeit, a `küld` és `fogad` eseményeket, és a belső eseményeket is) legalább eggyel növekszik. Egy esemény logikai ideje az *óra* változó *közvetlenül* az esemény *után* vett értéke, a megkülönböztetést végző i folyamatindexszel párosítva.
>
> Mindahányszor a LamportIdő$(A)_i$ folyamat végrehajt egy `küld` eseményt, először megnöveli saját *óra* változóját, hogy megkapja a `küld` eseményre vonatkozó v *óra* értéket, amelyet *időbélyegként* az elküldendő üzenethez csatol. Ha a LamportIdő$(A)_i$ folyamat egy `fogad` eseményt hajt végre, akkor úgy növeli saját *óra* változóját, hogy az nemcsak az előző értékénél, hanem az üzenet időbélyegénél is szigorúan nagyobb legyen. A `fogad` eseményhez az új *óra* érték rendelődik.

Pontosabban megadva, a LamportIdő(A) algoritmus i folyamatának kódja a következő.

18.1. algoritmus. LamportIdő$(A)_i$

Lenyomat:

Ugyanaz, mint A_i esetében, leszámítva, hogy a `küld`$(m)_i$ és a `fogad`$(m)_i$ műveleteket rendre a `küld`$(m, c)_i$ és a `fogad`$(m, c)_i$ $(c \in \mathbb{N})$ műveletekkel kell helyettesíteni.

Állapotok:
Ugyanazok, mint A_i esetében, plusz
óra $\in \mathbb{N}$, kezdetben 0

Átmenetek:
Ugyanazok, mint A_i esetében, az alábbi módosításokkal:

Bemeneti művelet $\neq$ `fogad`
  Hatás:
    Ugyanaz, mint A_i esetében, plusz:
    óra := *óra* + 1

Helyileg vezérelt művelet $\neq$ `küld`
  Előfeltétel:
    Ugyanaz, mint A_i esetében.
  Hatás:
    Ugyanaz, mint A_i esetében.
    óra := *óra* + 1

`küld`$(m, c)_i$
  Előfeltétel:
    Ugyanaz, mint az A_i-beli
      `küld`$(m)_i$ esetében, plusz:
      c = *óra* + 1
  Hatás:
    Ugyanaz, mint az A_i-beli
      `küld`$(m)_i$ esetében, plusz:
      óra := c

`fogad`$(m, c)_i$
  Hatás:
    Ugyanaz, mint az A_i-beli
      `fogad`$(m)_i$ esetében, plusz:
      óra := max(*óra*, c) + 1

Taszkok:
Ugyanazok, mint A_i esetében (modulo cserék).

Mivel az egyes folyamatok minden lépésben növelik saját *óra* változójuk értékét, valamint a megkülönböztető folyamatindexek miatt[3], könnyen látható, hogy a LamportIdő(A) kielégíti a logikai idő definíciójának 1. és 2. tulajdonságát. A 3. tulajdonság a `fogad` események kezeléséből következik. A 4. tulajdonság abból a tényből fakad, hogy minden esemény legalább 1-gyel megnöveli a hozzá tartozó *óra* változót.

Az alfejezet elején körvonalazott informális feltételek fogalmaival azt mondhatjuk, hogy minden A_i LamportIdő$(A)_i$-vé történő „transzformációja" egyszerűen egy új *óra* komponens, plusz az ezt karbantartó utasítások hozzáadását jelenti. Ez nem jelenti azonban teljesen újfajta műveletek hozzáadását, és nem késleltet eseményeket sem. A „szimuláció" lépésről lépésre történik, ami közvetlenül A egy pártatlan végrehajtási sorozatát adja. Amikor a LamportIdő$(A)_i$ folyamat A_i egy lépését szimulálja, a „generált" logikai idő érték pontosan a (c, i) pár, ahol c a lépés utáni *óra* érték.

Üzenetszórás. A LamportIdő transzformáció könnyen módosítható úgy, hogy aszinkron üzenetszóró rendszerekben is alkalmazható legyen.

[3]Lásd a LamportIdő transzformáció leírását. *A lektor.*

18.2.2.. Jövőbeli események késleltetése

Ebben a szakaszban egy alternatív algoritmus-transzformációt adunk meg egy A küld/fogad hálózati algoritmus egy végrehajtási sorozata logikai idejeinek előállítására. Ezt felfedezője, Welch nyomán WelchIdő-nek nevezzük. A LamportIdő-höz hasonlóan a WelchIdő is a helyi órák karbantartásán alapszik, csak itt az órákat nem az üzenetek átvételére adott válaszként állítjuk előre, hanem a „túl korán" érkező üzeneteket késleltetjük. Ez a transzformáció egy bizonyos értelemben „tolakodóbb", mint a LamportIdő transzformáció, mert az alapul szolgáló végrehajtási sorozat eseményei közé beveszi a késleltetéseket is. A T logikai idő tartomány olyan (c, i, k) hármasok halmaza, ahol c nemnegatív valós szám, i egy folyamatindex, $k \in \mathbb{N}^+$, a hármasok rendezése pedig lexikografikusan történik.

WelchIdő transzformáció (vázlatosan)

Minden WelchIdő$(A)_i$ folyamat rendelkezik egy *óra* nevű, nemnegatív valós értékű helyi változóval. Feltételezzük, hogy az WelchIdő(A_i) folyamat *óra* értékeit egy külön taszk kezeli, amely biztosítja azt, hogy az *óra* értékei monoton nem csökkenőek és korlát nélküliek legyenek.

Egy esemény logikai ideje az *óra* esemény bekövetkezésekori értéke egy elsőrendű megkülönböztetőként szolgáló folyamatindexszel és (az egy folyamaton belül azonos *óra* értékkel rendelkező események megkülönböztetése végett) egy másodrendű megkülönböztetőként funkcionáló sorszámmal, amely a végrehajtás sorrendjét adja meg. Figyeljük meg, hogy az adott A algoritmus egyetlen végrehajtási sorozata során sem változik meg az *óra* értéke. Egy `küld` esemény *óra* értékét *időbélyegként* az elküldendő üzenethez kapcsoljuk.

Minden WelchIdő$(A)_i$ rendelkezik egy *fogadó_puffer* nevű FIFO sorral, amelyben azon üzeneteket tárolja, amelyek időbélyegei a helyi *óra* értéknél nagyobbak, vagy azzal egyenlőek. Amikor WelchIdő$(A)_i$ kap egy üzenetet, megvizsgáljuk annak időbélyegét. Ha ez az időbélyeg kisebb a jelenlegi *óra* értéknél, akkor az üzenet azonnal feldolgozásra kerül, különben pedig elhelyezzük a *fogadó_puffer*ben. Minden helyileg vezérelt óra nélküli lépéskor WelchIdő$(A)_i$ először törli a *fogadó_puffer*ben található összes, jelenlegi *óra* értéknél kisebb időbélyeggel rendelkező üzenetet, és feldolgozza azokat. Ezen üzenetek a *fogadó_puffer*ben való előfordulásuk sorrendjében kerülnek feldolgozásra.

Azt mondjuk, hogy az algoritmus akkor szimulálja A egy `fogad`$(m)_i$ eseményét, amikor a megfelelő üzenet feldolgozásra kerül (nem pedig akkor, amikor az először WelchIdő$(A)_i$-hez érkezik). A `fogad` eseményhez kapcsolt *óra* érték az *óra* változó értéke az üzenet feldolgozásakor.

A WelchIdő(A) 4. tulajdonsága a helyi *óra* változók korlátozatlanságából következik. Ugyanebből következik az is, hogy a *fogadó_puffer* minden üzenete fel lesz dolgozva, így végül minden `fogad` esemény szimulálva lesz, és lesz hozzá rendelve logikai idő. Így valóban minden esemény kap egy logikai időt. Az 1.

és 2. tulajdonságok a megkülönböztetők létezéséből és a helyi órák monotonitásából következnek. A 3. tulajdonság teljesülését a *fogadó_puffer* használatának szabályai biztosítják.

A korábban, az alfejezet elején körvonalazott informális feltételek fogalmaival azt mondhatjuk, hogy minden A_i WelchIdő$(A)_i$-vé történő „transzformációja" hozzáadja és kezeli az *óra*, a *fogadó_puffer* és a megkülönböztető sorszám komponenseket. Ebben a transzformációban az A_i `fogad` műveletei késleltethetők. A „szimuláció" A-nak egy olyan pártatlan végrehajtási sorozatát eredményezi, amely átrendezi A bizonyos `fogad` eseményeit a többi eseményhez képest. Minden alkalommal, amikor WelchIdő$(A)_i$ A egy lépését szimulálja, a „generált" logikai idő érték pontosan az $(óra, i, k)$ hármas lesz, ahol k a másodrendű megkülönböztetőként használt sorszám.

Figyeljük meg, hogy a WelchIdő transzformáció által bevezetett késleltetés különösen nagy akkor, amikor a helyi órák közel sem szinkronizáltak. Ez az algoritmus akkor működik a legjobban, ha az órák éppen jól vannak szinkronizálva.

Üzenetszórás. A WelchIdő algoritmus-transzformációt könnyen lehet úgy módosítani, hogy aszinkron üzenetszóró rendszerekben is alkalmazható legyen.

18.3.. Alkalmazások

Ebben az alfejezetben a logikai idő aszinkron hálózati algoritmusokhoz adásának néhány egyszerű alkalmazását mutatjuk be.

18.3.1.. Banki rendszerek

Tekintsük a 15.43. példát, ahol egy olyan banki rendszerben kell a teljes pénzmennyiséget megszámolni, amelyben nincsenek külső betétek és kivétek, és a folyamatok közötti pénzátutalás üzenetek segítségével megy végbe.

A banki rendszer modellje egy A aszinkron küld/fogad hálózati algoritmus, amelynek felhasználói felületén nincsenek műveletek. Minden folyamat rendelkezik egy *pénz* nevű helyi változóval, amely az azon a helyen aktuálisan lévő pénz mennyiségét tartalmazza. A `küld` és `fogad` műveletek argumentumai a pénzmennyiségeket ábrázolják. A folyamatai döntik el, hogy mikor, hová, és mennyi pénzt kell küldeni. Egy olyan technikai feltételezéssel élünk, miszerint minden folyamat végtelen sok üzenetet küld minden szomszédjának. Ez nem nagy megszorítás, mivel 0€-ról szóló „üres" üzeneteket mindig hozzáadhatunk a rendszerhez.

Egy olyan aszinkron küld/fogad hálózati algoritmust szeretnénk, amelyben az egyes folyamatok egy helyi egyenleg felett rendelkeznek úgy, hogy az egyenlegek összege a rendszerben található tényleges pénzmennyiség legyen. Az algoritmus végrehajtását kívülről érkező jelekkel kell kiváltani, a rendszer egy vagy több helyéről. (Ezek a jelek bármikor érkezhetnek, és különböző helyeken eltérő számban fordulhatnak elő.)

Feltesszük, hogy az A algoritmust valamilyen módon (pl. a LamportIdő vagy WelchIdő segítségével) egy új, A-t szimuláló és annak eseményeihez logikai időket generáló $L(A)$ rendszerré transzformáljuk. Ekkor a szükséges PénztSzámol

algoritmus $L(A)$-ból egy újabb transzformációval nyerhető, amelyben a PénztSzámol minden PénztSzámol$_i$ folyamata az $L(A)$ megfelelő $L(A)_i$ folyamata tevékenységének „figyeléséért” felelős.[4]

PénztSzámol algoritmus (vázlatosan)

Az algoritmus központi eleme egy előre meghatározott $t \in T$ logikai időt használó „szubrutin”. Feltételezzük, hogy t az összes folyamat számára ismert, ekkor az általános stratégia a következő:

1. A minden folyamatára, a t-nél kisebb-egyenlő logikai idővel rendelkező események után, de a t-nél nagyobb logikai idővel rendelkező események előtt határozzuk meg a *pénz* változó értékét;
2. minden csatornára határozzuk meg a t-nél kisebb-egyenlő logikai időben elküldött, de a t-nél szigorúan nagyobb logikai időben megkapott összes üzenetben található pénzmennyiséget.

Speciálisan, az egyes PénztSzámol$_i$ folyamatok felelősek az A_i folyamatok *pénz* változói értékeinek, valamint az A_i-be beérkező csatornákban található pénzmennyiségeknek a meghatározásáért.

Ezek meghatározása érdekében a PénztSzámol$_i$ minden elküldött üzenethez időbélyegként hozzácsatolja a `küld` esemény logikai idejét. Az A_i folyamat *pénz* változója értékének meghatározása céljából a PénztSzámol$_i$ folyamat nyomon követi a *pénz* változó A_i legutóbb szimulált eseménye előtti és utáni értékeit. Amikor ez A_i első, t-nél szigorúan nagyobb logikai idővel rendelkező eseményét szimulálja, a PénztSzámol$_i$ visszaadja a *pénz* változó ezen esemény előtt megjegyzett értékét. (Ilyen esemény biztosan létezik, hiszen A_i végtelen sok műveletet hajt végre, de csak véges sok üzenet rendelkezik t-nél kisebb vagy egyenlő logikai idővel.)

A j-től i-be vezető csatornában lévő pénzmennyiség meghatározása érdekében a PénztSzámol$_i$ folyamatnak azokat az üzeneteket kell meghatároznia, amelyek `küld`$_j$ eseményei t-nél kisebb vagy egyenlő, `fogad`$_i$ eseményei pedig t-nél szigorúan nagyobb logikai idővel rendelkeznek. Így A_i első, t-nél nagyobb logikai idejű eseményétől (azaz attól, amelynél a PénztSzámol$_i$ meghatározza A_i esetében a *pénz* értékét) kezdve, a PénztSzámol$_i$ folyamat megjegyzi a csatornába beérkező üzeneteket. Egészen addig jegyzi meg őket, amíg a hozzájuk kapcsolt időbélyeg kisebb-egyenlő t-nél. Ha a csatornába egy t-nél szigorúan nagyobb időbélyeggel rendelkező üzenet érkezik, akkor a PénztSzámol$_i$ visszaadja a megjegyzett üzenetekben szereplő pénzmennyiségek összegét. (Ilyen üzenet biztosan érkezik, mivel A_j végtelen sok eseményt küld A_i-nek.)

Az egyes PénztSzámol$_i$ folyamatok által (a szubrutinban) kiszámított egyenleg az általuk az A_i folyamatokra és az összes bejövő csatornára meghatározott értékek összege.

[4] Ez a konstrukció a következő technikai feltevésekkel él az $L(A)$ transzformált algoritmusra vonatkozólag: a szimuláció lépésről lépésre történik, és a transzformált algoritmus az A lépéseit és azok logikai idejeit a PénztSzámol folyamatai által azonosítható formában adja meg.

Ne feledjük, hogy ehhez egy előre definiált t logikai időre van szükség. Mivel ilyen előre definiált t nem létezik, ezért azt a folyamatoknak kell valahogyan meghatározniuk. (Nem lehet a t-t tetszőlegesre választani, mert az a logikai idő esetleg a szubrutin végrehajtásának kezdése előtt már néhány folyamatot időben túlhaladhat.) Például a folyamatok használhatnak egy olyan előredefiniált, logikai időket tartalmazó, növekvő $t_1, t_2, \ldots$ sorozatot, hogy minden $t \in T$-re igaz, hogy $t \leq t_i$ valamely i-re, és megpróbálhatják (párhuzamosan) befejezni az ezekhez tartozó szubrutinokat. Eredményeik üzenetszórásával a folyamatok meghatározhatják az első olyan t_i-t, amelynek szubrutinja mindenhol sikeresen végrehajtódik, és ezen szubrutin eredményeit használhatják.

Most belátjuk a szubrutin helyességét, tetszőleges t-re. Annak belátásához, hogy az általános stratégia a pénz tényleges összmennyiségét adja, tekintsünk a PÉNZTSZÁMOL egy tetszőleges, rögzített, pártatlan végrehajtási sorozatát. Ez a végrehajtási sorozat A egy *log_idő* logikai idejű kiosztással rendelkező α pártatlan végrehajtási sorozatát szimulálja. Ekkor a 18.1. tételből következik, hogy A-nak létezik egy másik, ugyanazon eseményeket tartalmazó α' pártatlan végrehajtási sorozata, amely az összes A_i folyamat számára megkülönböztethetetlen α-tól, és amelyben minden esemény *log_idő*-ik sorrendjében fordul elő. Az általános stratégia „elvágja" az α' végrehajtási sorozat közvetlenül azon események után, amelyekre *log_idő* $= t$, és megjegyzi, hogy ebben a pillanatban mennyi pénz van a folyamatoknál és a csatornákban. Így az általános stratégia egy *azonnali globális fényképet* ad a rendszerről az α' végrehajtási sorozat közben, amely biztosan a pénz tényleges összmennyiségét eredményezi a banki rendszerben.

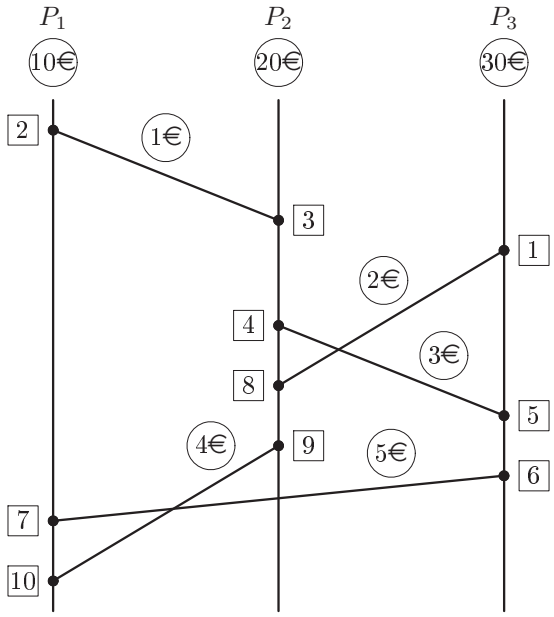
Egyszerűen belátható, hogy az osztott algoritmus valóban helyesen valósítja meg az általános stratégiát.

18.3.1. példa. A PÉNZTSZÁMOL algoritmus végrehajtási sorozata

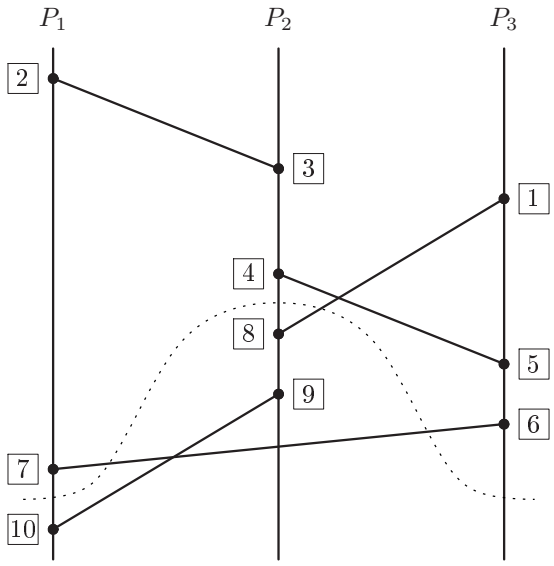
A 18.4. ábrán egy A banki algoritmus α pártatlan végrehajtási sorozatának küld/fogad diagramja látható, az $L(A)$ által hozzárendelt logikai idő értékekkel. Az ábrán az A_i-nek megfelelő folyamatot P_i-vel jelöltük. Az egyes folyamatok kezdeti pénzmennyisége a megfelelő idővonal felett látható, míg az üzeneteket jelképező élek az átutalt pénz mennyiségével vannak címkézve.

Most tekintsük a PÉNZTSZÁMOL algoritmus egy, az α-t szimuláló pártatlan végrehajtási sorozatát. Tételezzük fel, hogy ebben a végrehajtási sorozatban a $t = 7.5$ értéket használjuk. A 18.5. ábra az előző diagramot egy pontozott vonallal bővíti ki, amely azokat a helyeket mutatja (és kapcsolja össze), ahol a 7.5 logikai idő metszi a folyamatok idővonalait.

A PÉNZTSZÁMOL végrehajtási sorozatában P_1 az A_1 folyamat *pénz* értékét $10€ - 1€ + 5€ = 14€$-ban határozza meg; P_2 az A_2 folyamat számára határozza meg a $20€ + 1€ - 3€ = 18€$ értéket; a P_3 által A_3 számára meghatározott érték pedig a



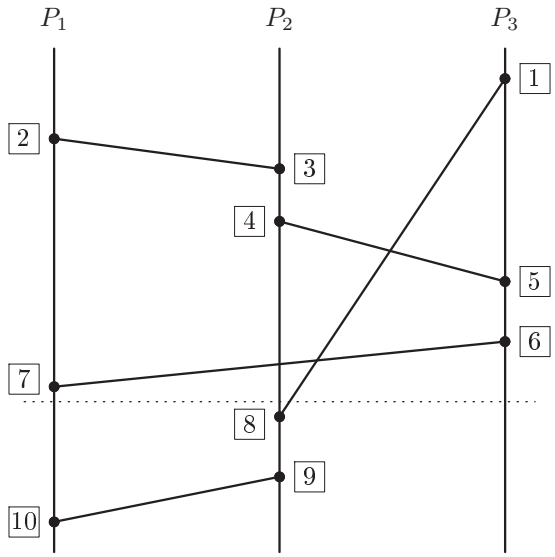
18.4.. ábra. Az A banki algoritmus α végrehajtási sorozata.



18.5.. ábra. Pontozott vonal a $t = 7.5$ helyen.

30€ − 2€ + 3€ − 5€ = 26€ érték. A P_3-tól P_2-höz vezető csatorna kivételével – amely P_2 szerint 2€-t tartalmaz – az összes csatorna üres. A meghatározott összmennyiség így 14€ + 18€ + 26€ + 2€ = 60€, ami pontosan a helyes összeg.

A 18.6. ábrán az α' átrendezett végrehajtási sorozat küld/fogad diagramja látható, amelyen az események logikai idejük szerinti sorrendben helyezkednek el. Itt a $t = 7.5$-nek megfelelő pontozott vonal vízszintes, és pontosan egy élet keresztez, mégpedig a 3. folyamattól a 2. folyamathoz vezető egyetlen élet. Könnyen belátható, hogy a kiszámolt mennyiségek a folyamatok és a csatornák α'-beli helyzetét írják pontosan le a 7.5 időpillanatban.



18.6.. ábra. Az α' átrendezett végrehajtási sorozat egy vízszintes vonallal a $t = 7.5$ helyen.

Megjegyezzük, hogy a PénztSzámol algoritmus nem okoz az A működésében semmilyen további késleltetést az $L(A)$ által okozotton felül.

18.3.2.. Globális fényképek

A PénztSzámol algoritmus ötlete egy egyszerű banki rendszerről egy tetszőleges A aszinkron küld/fogad rendszerre általánosítható. (Mint ahogyan korábban is, most is feltesszük, hogy minden A_i folyamat végtelen sok üzenetet küld minden szomszédjának.) Tegyük fel, hogy A egy végrehajtási sorozata közben egy azonnali globális fényképet szeretnénk kapni a rendszer állapotáról. Ez igen hasznos lehet például nyomkövetés esetében, alkalmas lehet hiba esetében a rendszer egy

régebbi állapotának visszaállítására, vagy bizonyos globális tulajdonságok felderítésére, például hogy az algoritmus véget ért-e mindenhol. Addig késleltetve az összes folyamatot és üzenetet, amíg rögzíteni tudjuk az összes szükséges információt, egy azonnali globális fényképet kaphatunk. Azonban ez a stratégia a legtöbb valós méretű elosztott rendszerben nem célravezető.

Bizonyos alkalmazásokban lehet, hogy nincs is szükség egy valódi, azonnali globális fényképre, hanem egy „annak tűnő" rendszerállapot is megteszi. Néhány ilyen alkalmazást az előző bekezdésben már megemlítettünk, további példák a 19. fejezetben olvashatók. Ilyen esetben az a stratégia, amely egy banki rendszerben található pénz összmennyiségét határozza meg, átalakítható egy A aszinkron küld/fogad hálózati rendszer elfogadható globális fényképe biztosításának érdekében. A korábbiakhoz hasonlóan, A-t először kiegészítjük logikai időkkel.

LogikaiIdejűFénykép algoritmus (vázlatosan)

A PénztSzámol-hoz hasonlóan, az algoritmus központi eleme itt is egy előre meghatározott $t \in T$ logikai időt használó szubrutin. Feltesszük, hogy t az összes folyamat számára ismert. Ekkor a stratégia a következő:

1. meghatározzuk A minden folyamatának a t-nél kisebb, vagy vele egyenlő logikai idejű események utáni, és a t-nél nagyobb logikai idejű események előtti állapotát;
2. minden csatornára meghatározzuk a t-nél kisebb, vagy vele egyenlő logikai időben küldött, de a t-nél szigorúan nagyobb logikai időben fogadott üzenetek sorozatát;

Ezt az információt a PénztSzámol-ban használt osztott algoritmus segítségével kapjuk meg.

A LogikaiIdejűFénykép minden pártatlan végrehajtási sorozata A-nak egy *log_idő* logikai idejű kiosztással rendelkező α pártatlan végrehajtási sorozatát szimulálja. Ekkor a 18.1. tételből következik, hogy a visszaadott globális állapot A egy másik, olyan α' pártatlan végrehajtási sorozatának egy azonnali globális fényképe, amely ugyanazokat az eseményeket tartalmazza *log_idő* szerinti sorrendben, és amely α-tól az összes folyamatra nézve megkülönböztethetetlen. Ez elégséges például a rendszerállapot egy elfogadható biztonsági másolatának létrehozásához.

18.3.3.. Egyállapotú gépek szimulálása

Logikai idő segítségével egy elosztott rendszer szimulálhat egy központosított állapotgépet, vagy más szóval, egy közös változót. Emlékezzünk vissza a *változótípus* 9.4. alfejezetben megadott formális definíciójára: ez értékek V halmazából, egy v_0 kezdeti értékből, *kérések* és *válaszok* halmazából, és egy $f : \mathit{kérések} \times V \to \mathit{válaszok} \times V$ függvényből áll. Megmutatjuk, hogyan kell az aszinkron üzenetszóró hálózati modellben egy adott típusú x közös változót „megvalósítani".

Tekintsünk egy olyan környezetet, amelyben n felhasználói folyamat küld kéréseket x-hez, és fogad válaszokat x-től, és a hálózat minden i csúcspontjában egy U_i felhasználói folyamat van. Minden felhasználói folyamatról feltesszük, hogy a

kéréseket sorosan küldi, azaz egy új kérés elküldése előtt addig várakozik, amíg az előző kérésre választ nem kap. Azt szeretnénk, hogy a felhasználók egy olyan nézethez jussanak, amely összhangban van azzal, hogy van x-nek egy önálló másolata, amelyre az összes műveletet alkalmazzuk. Pontosabban, a hálózatnak, mint egésznek („elrejtve" a `küld` és `fogad` műveleteket), az adott típus *atomi objektumának* kell lennie, mint ahogyan azt a 13.1. alfejezetben definiáltuk. Nem kívánunk meg semmilyen rugalmassági követelményt, csak a jólformáltságot, az atomi tulajdonságot és a hibamentes végrehajtási sorozatot követeljük meg.

Ennek a problémának sok lehetséges megoldása van, amelyek közül néhányat a 17.1. alfejezetben írtunk le. Például egy folyamat karbantarthatja x egy önálló másolatát, és minden műveletet ezen az egy másolaton végezhet – lásd az EgyszerűOsztVáltSzim algoritmust. Itt egy olyan megoldást tekintünk, amelyben *minden* folyamat rendelkezik x-ről egy saját másolattal, minden kérés üzenetszórással lesz továbbítva az összes folyamathoz, amelyek a műveleteket saját másolataikon hajtják végre. Egy műveletet kezdeményező folyamat meg tudja határozni a szükséges választ, amikor végrehajt egy műveletet saját másolatán. Annak érdekében, hogy ez a stratégia helyesen működjön – azaz annak biztosítása érdekében, hogy minden folyamat ugyanabban a sorrendben hajtsa végre a tevékenységeket a saját másolatán, és hogy azok az időpontok, amelyekben a műveletek bekövetkezhetnek, benne legyenek a műveletek megfelelő kérés-válasz intervallumában –, egy bizonyos fokú szinkronizációra van szükség. A szükséges szinkronizációt a logikai idő segítségével érjük el.

MásoltÁllapotgép algoritmus (vázlatosan)

Az algoritmus egy teljesen hétköznapi A aszinkron algoritmussal kezdődik. Minden A_i folyamat egyszerűen csak fogadja az U_i felhasználótól érkező kéréseket, és szétszórja őket. (Az nem számít, hogy a folyamatok mit csinálnak ezekkel az üzenetekkel akkor, amikor megkapják őket.) Ezen felül A_i szükség szerint üres üzeneteket is szór, így biztosítva azt, hogy az üzenetszórás végtelen sokszor megtörténjen. Ezt követően, az előzőekhez hasonlóan, A-hoz egy transzformáció segítségével adunk logikai időt, amely $L(A)$-t eredményezi.

A fő algoritmus $L(A)$-t használja. x helyi másolatán kívül minden $L(A)_i$ rendelkezik egy *kérés_puffer* lokális változóval, amelyben az összes olyan kérést tárolja, amelyről valaha is hallott, azok `üzen_szór` eseményeinek logikai idejeivel együtt. $L(A)_i$ egy helyi kérést akkor helyez a *kérés_puffer*ébe, amikor végrehajtja a kérésre vonatkozó `üzen_szór` műveletet, egy távoli (azaz más folyamatnál előforduló) kérést pedig akkor, amikor a kérésre vonatkozó `fogad` kerül végrehajtási sorozatra.

Minden $L(A)_i$ karbantart egy *ismert_idő* vektort is, amely minden folyamat esetében nyomon követi, hogy melyik az a legnagyobb logikai idő, amelyről hallott. Ez kezdetben 0. Így *ismert_idő*$(i)_i$ pontosan az $L(A)_i$-ben legutoljára bekövetkezett esemény logikai ideje, *ismert_idő*$(j)_i$ pedig, $j \neq i$ esetében, az `üzen_szór` esemény logikai ideje arra a legutóbbi üzenetre, amelyet az L(A_i) folyamat L(A_j)-től kapott.

A következő feltételek teljesülése esetében minden $L(A)_i$ számára megengedett, hogy egy, a *kérés_puffer*-ében levő π kérést saját, x-ről készült másolatára alkalmazza:

1. a π kérés a legkisebb logikai idejű olyan kérés a *kérés_puffer*$_i$-ben, amelyet $L(A)_i$ még nem alkalmazott x-re,
2. minden j-re, az *ismert_idő*$(j)_i$ legalább akkora, mint π logikai ideje.

Amikor $L(A)_i$ egy helyileg kezdeményezett műveletet alkalmaz saját, x-ről szóló másolatára, akkor továbbítja a választ x-től a felhasználónak.

18.3. lemma. . *A* MásoltÁllapotgép *algoritmus egy atomi objektumot valósít meg.*

Bizonyítás. A jólformáltság könnyen belátható. Most bebizonyítjuk a megállást. Tekintsünk egy tetszőleges α pártatlan végrehajtási sorozatot. A logikai idő 1. tulajdonságából következik, hogy α minden kéréséhez hozzá van rendelve egy egyedi logikai idő. A 4. tulajdonságból az következik, hogy α-ban csak véges sok, bármely adott t-nél kisebb logikai idővel rendelkező kérés van. Így létezik az α-ban levő kéréseknek egy olyan egyedi módon megadott Π sorozata, amely a kérések `üzen_szór` eseményeinek logikai ideje szerint van rendezve.

A megbízható üzenetszórás biztosítja, hogy végül minden folyamat minden kérést elhelyezzen saját *kérés_puffer*-ébe. Mivel A minden folyamatában végtelen sok esemény következik be, a 4. tulajdonságból következik, hogy minden egyes folyamat logikai ideje korlát nélkül nő. Abból, hogy minden folyamat végtelen sok alkalommal végez üzenetszórást, következik, hogy az egyes folyamatok *ismert_idő* vektorának egyes komponensei is korlát nélkül nőnek. Ekkor a Π sorozatban szereplő kérések helyzetére vonatkozó indukcióval beláthatjuk, hogy végeredményben minden kérést x minden másolatára alkalmazunk. Ebből az következik, hogy minden kérésre keletkezik egy válasz, ami a megállást bizonyítja.

Most belátjuk az atomi tulajdonságot. Tekintsünk egy tetszőleges (véges vagy végtelen) α végrehajtási sorozatot. A korábbiakhoz hasonlóan (lásd pl. a 17.4. tétel bizonyítását) feltehetjük, hogy α nem tartalmaz befejezetlen műveleteket.

Először azt állítjuk, hogy minden folyamat az x-ről készült helyi másolatán a műveleteket azok logikai idejeinek sorrendjében alkalmazza, kihagyások nélkül. Ez azért van, mert amikor $L(A)_i$ egy t logikai idővel rendelkező π műveletet alkalmaz x-re, explicit módon ellenőrzi, hogy nem tud-e valamely, t-nél kisebb logikai idővel rendelkező, várakozó kérésről, és hogy saját *ismert_idő*-i minden folyamatra legalább egyenlőek-e t-vel. Ezután az egyes folyamatpárok közötti üzenetszóró csatorna FIFO tulajdonsága miatt $L(A)_i$ sosem fog semmilyen más, t-nél kisebb logikai idővel rendelkező eseményről hallani.

Most α minden műveletéhez megadunk egy sorrendi pontot: minden $L(A)_i$ által kezdeményezett olyan π műveletre, amelynek `üzen_szór` eseményének logikai ideje t, a sorrendi pontot válasszuk meg úgy, hogy az a legkorábbi olyan pont legyen, amelyre a rendszer minden folyamata elérte a $\geq t$ logikai időt. (Az egyezések a sorrendi pontok logikai idők szerinti sorba rendezésével szüntethetők meg.) Egy ilyen pontot biztosan el kell tudnunk érni α-ban, hiszen már beláttuk, hogy $L(A)_i$-nek a saját, x-ről készült helyi másolatára kell alkalmaznia a π műveletet;

azonban ezt nem teheti meg addig, amíg az összes *ismert_idő*-je legalább t nem lesz, amiből következik, hogy már minden folyamat elérte a legalább t logikai időt.

Figyeljük meg, hogy egy π művelet sorrendi pontja, a logikai idő 2. tulajdonságának, és annak a ténynek köszönhetően, hogy t π `üzen_szór`-jának logikai ideje, nem előzheti meg π kérését. Másfelől, π sorrendi pontja nem lehet π válasza után, mivel a π-t kezdeményező folyamat addig nem válaszol a felhasználónak, amíg minden *ismert_idő*-je legalább a t-t el nem éri. Így a sorrendi pontok mindig a műveleti intervallumokba esnek.

Mivel a sorrendi pontok a logikai idők sorrendjében fordulnak elő, amely sorrend megegyezik azzal, amely alapján a műveletek a helyi másolatokon végrehajtódnak, az atomi tulajdonsághoz szükséges „zsugorodó" tulajdonság fennáll. □

Nem nyilvánvaló, hogy a MásoltÁllapotgép algoritmusnak vannak-e előnyei az egyszerű, központosított EgyszerűOsztVáltSzim algoritmussal szemben. Mindazonáltal a MásoltÁllapotgép alapvetően minden folyamattól azt követeli meg, hogy végezze el azt a munkát, amit a központosított algoritmusban egy folyamat végez el. Egy előnyt abban az esetben láthatunk, ha a különböző folyamatok logikai idejei jól szinkronizáltak maradnak. Ebben az esetben az EgyszerűOsztVáltSzim algoritmusban egy művelet végrehajtási sorozatára rendelkezésre álló idő körülbelül egy kétirányú („retúr") üzenetnyi késleltetés. A MásoltÁllapotgép-ben viszont egy $L(A)_i$ kezdeményező folyamat egy π műveletet azonnal végrehajthat, mihelyst megtudja, hogy az összes többi folyamat elérte a π `üzen_szór` eseményéhez rendelt logikai időt. Ha az órák jól szinkronizáltak, akkor ez körülbelül csak egy egyirányú üzenetnyi késleltetést igényel.

A MásoltÁllapotgép egy közös memóriával rendelkező rendszer osztott megvalósításában szereplő közös változók megvalósítására használható. Ez a megközelítés a 17.1. alfejezetben javasolt megvalósítási technikák alternatívája.

Az olvasási műveletek speciális kezelése.. Tegyük fel, hogy egy x közös változón operáló műveletek némelyike *olvasási* műveletként (vagy általánosabban, tetszőleges olyan műveletként, amely nem változtatja meg a változó értékét, csak egy választ ad vissza) van megvalósítva. Ekkor a MásoltÁllapotgép módosítható úgy, hogy ezeket a műveleteket helyileg, a *kérés_puffer* mechanizmus használata nélkül hajtsa végre. Ez a változtatás az atomi objektuménál gyengébb helyességi garanciákat eredményez, de sok alkalmazás számára még így is elfogadható lehet.

Banki osztott adatbázis.. A MásoltÁllapotgép algoritmus egy olyan környezetben is használható, ahol az x közös változó egy teljes banki adatbázist reprezentál. Ennek tipikus műveletei a `betesz`, `kivesz`, `kamatot_emel` stb. Az adatbázisról másolatok készíthetők – például a bank egyes leányvállalatainál. Egy ilyen adatbázisban a frissítések sorrendje sok művelet esetében igen fontos. Például, alacsony egyenleg esetében különféle eredményre juthatunk akkor, ha egy `kivesz` műveletet egy `betesz` előtt alkalmazunk, nem pedig utána. Így a műveletek alkalmazásának a MásoltÁllapotgép algoritmus által biztosított következetes sorrendje igen lényeges.

Az önálló leányvállalatok számára sokszor hasznos lehet, hogy képesek legyenek az adatbázis helyi másolatában tárolt információk olvasására még akkor is, ha ez az információ nem teljesen naprakész. Ebben az esetben hasznos lehet az `olvas` műveletek fentebb leírt speciális kezelése.

Kölcsönös kizárás.. A kölcsönös kizárás problémáját az aszinkron közös memória modellek számára a 10. fejezetben, az aszinkron hálózati modell számára pedig a 20. fejezetben adtuk meg. Röviden, a felhasználók egy erőforrás kizárólagos használatát egy `próbál` művelettel kérik, amit a rendszer egy `belép` művelet segítségével engedélyez. A felhasználók az erőforrást egy `kilép` művelettel adják vissza, amelyre a rendszer egy `halad` művelettel válaszol. Feltételezzük, hogy a rendszer garantálja azt, hogy egy időben legfeljebb egy felhasználó rendelkezzen az erőforrással, és azt, hogy kérések esetében az erőforrás engedélyezése folyamatos legyen. Itt megköveteljük a *kizárásmentességet*, azaz azt, hogy minden kérelmet előbb-utóbb ki kell szolgálni.

A MásoltÁllapotgép algoritmus használható egy üzenetszóró hálózat kölcsönös kizárási problémájának megoldására is. Ez esetben az x közös változó folyamatindexek egy `hozzáad`(i), `első`(i) és `eltávolít`(i) műveletekkel rendelkező FIFO sora. A `hozzáad`(i) művelet a megadott indexet a sor végére helyezi. Az `első`(i) művelet egy olyan lekérdezés, amely *igazat* ad akkor, ha i a sor első eleme, különben pedig *hamisat*. Az `eltávolít`(i) művelet az i index összes előfordulását törli a sorból. Legyen B_x egy atomi objektum x számára, ahol az i kapu az összes i argumentummal rendelkező műveletet támogatja.

Ha az U_i felhasználó egy `próbál`$_i$ esemény segítségével egy kritikus Be szakaszba kíván belépni, akkor az U_i folyamat a B_x atomi objektumon végrehajt egy `hozzáad`(i) műveletet, amely U_i-t a sor végére helyezi. Ezután U_i újra meg újra meghívja az `első`(i) műveletet, arra várva, hogy a válasz *igaz* legyen, amely azt jelzi, hogy i elérte a sor legelső pozícióját. Ha i *igaz* választ kap, engedélyezi az i felhasználónak, hogy egy `belép`$_i$ művelettel belépjen a kritikus Be szakaszba. Mihelyst az U_i felhasználó egy `kilép`$_i$ művelettel kilép a kritikus Be szakaszból, az i folyamat a B_x atomi objektumon végrehajt egy `töröl`(i) műveletet. Miután ez a művelet befejeződött, az i folyamat az i felhasználót egy `halad`$_i$ művelettel a Ha szakaszba engedi. (Ez lényegében a 10.9.2. szakaszban leírt SorKK algoritmus.) Ez a B_x atomi objektum bármilyen, például a MásoltÁllapotgép segítségével történő megvalósításával megoldja a kölcsönös kizárás problémáját (kizárásmentességgel).

Azonban a MásoltÁllapotgép használata esetében lehetőség van egy egyszerű optimalizálásra. Nevezetesen, változtassuk meg a `hozzáad`(i) műveletet úgy, hogy az adjon is vissza egy értéket: vagy a sorban i-t megelőző j indexét, ha ilyen van, vagy *null*-t, ha ilyen nincs. Ha a visszatérési érték *null*, az azt jelenti, hogy i-nek nincs megelőzője, így az i folyamat azonnal végrehajthatja a `belép`$_i$-t. Egyébként az i folyamat egyszerűen addig vár, amíg a sorról készített saját helyi másolatán végre nem hajtja az i-t megelőző j-re a `töröl`(j)-t (amikor is tudja, hogy a j felhasználó visszaadta az erőforrást). Ezután végrehajtja a `belép`$_i$-t. A `kilép`$_i$-t ugyanúgy kezeljük, mint korábban.

18.4.. Valósidejű és logikai idejű algoritmusok*

Minden, eddig leírt algoritmus egy logikai idővel bővített A aszinkron algoritmuson alapult. Egy másik tervezési stratégia azt javasolja, hogy induljunk ki egy, a „valós idő" fogalmát használó algoritmusból, majd transzformáljuk azt egy, a valós idő helyett logikai időt használó algoritmusba.

Tegyük fel, hogy egy A aszinkron küld/fogad hálózati rendszerből indulunk ki, amelyben minden A_i folyamat rendelkezik egy $\mathbb{R}_0^+$-beli értékeket felvenni képes *valós_idő* lokális változóval, amely kezdetben 0 értéket vesz fel. Tegyük fel, hogy az összes folyamat *valós_idő* változóját egy globális ValósIdő b/k automata tartja karban, `ketyeg`(t) kimeneteken keresztül, amelyek az összes folyamat *valós_idő* változóját egyszerre állítják be t-re. (A b/k automata modell megengedi, hogy egy egyedi kimeneti műveletet egynél több bemeneti művelettel lehessen szinkronizálni.) A ValósIdő automatára vonatkozó egyetlen megszorítás, hogy az a szám, amely a kimeneti eseményeiben argumentumként előfordul, ne csökkenjen és korlátozott legyen, minden pártatlan végrehajtási sorozat esetében.[5] Az A_i folyamatok nem változtathatják meg a *valós_idő* változókat.

Ezután minden A_i folyamatot egy olyan B_i folyamatba lehet transzformálni, amely a ValósIdő nélkül, logikai idők használatával működik. B_i nem rendelkezik *valós_idő* változóval, de helyette van egy *óra* változója, amelyet ugyanúgy használ, mint ahogyan A_i a *valós_idő*-t. Az *óra* változókat a B_i-k tartják karban egy $\mathbb{R}_0^+$ (vagy $\mathbb{R}_0^+$ egy részhalmaza) logikai idő tartománnyal rendelkező logikai idő megvalósítás segítségével.

Annak leírása érdekében, hogy ez a transzformáció mit garantál, tekintsük mind A-t, mind pedig annak transzformált változatát, B-t, amelyekhez U_i felhasználói automaták tartoznak (minden i csúcshoz egy U_i). Ekkor a következő tételt kapjuk.

18.4. tétel. . *A B rendszer (azaz a B, plusz a felhasználói automaták) minden α pártatlan végrehajtási sorozatára létezik az A rendszernek (A, plusz a* ValósIdő *automata és a felhasználók) egy α' pártatlan végrehajtási sorozata, amely minden U_i-re megkülönböztethetetlen α-tól.*

Azaz az egyéni felhasználók számára B minden pártatlan végrehajtási sorozata olyan, mint A egy végrehajtási sorozata.

18.4.1. példa. Banki rendszer

Lehetőség van egy PénztSzámol-hoz hasonló, de valós időt használó algoritmust tervezni egy bankban található összes pénz mennyiségének kiszámításához. Nevezetesen, minden i folyamat feljegyzi saját *pénz* változójának értékét éppen azelőtt a lépés előtt, amikor úgy találja, hogy *valós_idő* változója meghaladja t-t. Ezután minden olyan

[5] Mivel a ValósIdő csak egy közönséges b/k automata, nem tételezhetünk fel semmit kimeneteinek előfordulási „gyakoriságáról". A 23–25. fejezetekben megvizsgálunk egy olyan modellt, amelyben az ilyen, gyakoriságra vonatkozó felvetések kifejezhetők.

bejövő üzenetet feljegyez, amelyet akkor küldtek el, amikor a küldő *valós_idő* változója kisebb volt t-nél vagy egyenlő volt vele, de amikor megkapta, az i folyamat *valós_idő* változója nagyobb volt t-nél.

Az így kapott algoritmust a fenti módon logikai időt használó algoritmussá lehet transzformálni.

18.5.. Megjegyzések a fejezethez

A logikai idő fogalma Lamport nevéhez fűződik, ő írt először róla híres cikkében, a „Time, Clocks and the Ordering of Events in a Distributed System"-ben (Az idő, az órák és az események elrendezése egy elosztott rendszerben) [176]. Ez a cikk leírja a LamportIdő algoritmus-transzformációt, valamint a MásoltÁllapotgép algoritmus kulcsfogalmainak rövid leírását is. Lamport később a másoló állapotgép megközelítést kiterjesztette úgy, hogy alkalmas legyen korlátozott számú hiba tűrésére [179]. Schneider [255] a másoló állapotgépek hibatűrő szolgáltatások megvalósításában való hasznosságáról írt egy áttekintő tanulmányt.

A WelchIdő algoritmus-transzformációt Welch [286] adta meg; ugyanezt a transzformációt tanulmányozta Neiger és Toueg [232] is, Chaudhuri, Gawlick és Lynch [74] pedig egy részben szinkron modellé bővítették.

A PénztSzámol és a LogikaiIdejűFénykép algoritmusok Chandy és Lamport [68] ellentmondásmentes globális fénykép algoritmusával állnak szoros kapcsolatban.

A fejezetben előforduló, banki adatbázisokról szóló példákat Lynch, Merritt, Fekete és Weihl a [207]-ben alaposan tárgyalják. Ott a hangsúly a banki és egyéb adatbázisok atomi tranzakcióin van.

A 18-17. gyakorlatban vázolt „vektor órák" algoritmus Mattern-nek [222], Liskov-nak és Ladin-nak [197], valamint Fidge-nek [115] köszönhető. Ezt alkalmazzák az Isis rendszerben [52]. A vektor órák alkalmazásairól a [256]-ban jelent meg egy áttekintő tanulmány.

18.6.. Gyakorlatok

18-1. Bizonyítsuk be a 18.2. tételt.

18-2. Írjunk „kódot" a WelchIdő algoritmus-transzformációhoz ugyanabban az általános stílusban, mint ahogyan a LamportIdő kódja van megadva.

18-3. Adjunk meg egy $\mathbb{R}_0^+$ logikai idő tartománnyal rendelkező logikai idő megvalósítást egy küld/fogad rendszer számára.

18-4. A Zöldfülűek Számítástechnikai Rt. nevű cégnél egy késő éjszakába nyúló pénteki munka alkalmával, pizza mellett, néhány programozó kidolgozta az aszinkron küld/fogad hálózati rendszerek „nemlogikus idejének" négy alapfogalmát. A nemlogikus idő alapfogalmait úgy kapjuk, hogy a logikai időhöz szükséges négy tulajdonságából pontosan egyet rendre elhagyunk. Úgy gondolták, hogy ezek bizonyos alkalmazások számára hasznosak lehetnek. Minden fogalomra,

(a) tervezzünk egy algoritmus-transzformációt, amely ezt a fajta nemlogikus időt egy adott A aszinkron hálózati algoritmus végrehajtási sorozataira alkalmazza;
(b) adjunk meg lehetséges alkalmazásokat.

18-5. A PÉNZTSZÁMOL algoritmust egy A banki rendszerre alkalmazott dupla algoritmus transzformációként adtuk meg, amely nehezen átláthatóvá teszi azt, hogy mi is zajlik valójában. Egyesítsük a különféle részeket egyetlen algoritmusba.

(a) Írjunk előfeltétel/hatás kódot valamely, a 18.3.1. szakaszban megengedett típusú A banki rendszer számára. Ehhez az egyes folyamatok kezdeti pénzmennyiségét, plusz néhány olyan szabályt kell megadni, amelyek meghatározzák, hogy mikor, kihez és mennyi pénzt küldenek.
(b) Írjunk előfeltétel/hatás kódot az (a) részben megadott A algoritmusunknak egy olyan módosított változatára, amely logikai időket is tartalmaz. A logikai idők generálását végezzük kedvenc algoritmusunkkal.
(c) Írjunk előfeltétel/hatás kódot a (b) részben megadott algoritmusunknak egy olyan módosított változatára, amely a szükséges egyenlegek létrehozására a PÉNZTSZÁMOL stratégiáját alkalmazza. Ne hagyjuk ki a megfelelő t logikai idő meghatározására alkalmas mechanizmust sem.

18-6. Vegyük újra fontolóra a 18.3.1. szakaszban megadott banki rendszer példáját. Most tegyük fel, hogy egy ilyen A banki rendszer az átutalásokon felül betéteket és kivéteket is lehetővé tesz (amelyek a rendszer felhasználói felületén bemeneti műveletként vannak modellezve). Ha ugyanazt a PÉNZTSZÁMOL transzformációt alkalmazzuk, mint korábban, akkor mit állapíthatunk meg az így kapott rendszer kimenetéről?

18-7. Alkalmazzuk a LOGIKAIIDEJŰFÉNYKÉP algoritmust küld/fogad rendszerek helyett üzenetszóró rendszerekre. Gondosan állapítsuk meg, hogy algoritmusunk mit garantál.

18-8. A PÉNZTSZÁMOL és a LOGIKAIIDEJŰFÉNYKÉP algoritmusokban a logikai idő minden üzenethez hozzá van kapcsolva. Fejlesszünk ki egy olyan alternatív algoritmust, amely nem az üzenetekhez rendeli a logikai időket, hanem e helyett minden csatornára egy önálló extra *jelző* üzenetet küld, amely jelzi a t-nél kisebb-egyenlő, illetve a t-nél nagyobb logikai időben elküldött üzenetek közötti osztópontot. Bizonyítsuk be ennek helyességét.

18-9. A 13-21. gyakorlat alapulján adjunk egy másik bizonyítást a 18.3. lemmára.

18-10. Tegyük fel, hogy a „nemlogikus idő"-t, ami valójában egyfajta logikai idő, amely kielégíti az 1., 2. és 4. tulajdonságokat, de a 3'-t nem, használjuk a MÁSOLTÁLLAPOTGÉP algoritmusban. Mely tulajdonságok garantáltak?

18-11. Fejlesszük ki a 18.3.3. szakaszban leírt közös változó olyan módosított megvalósítását, amely az `olvas` műveleteket helyileg kezeli. Mutassuk meg, hogy ez általános esetben *nem* egy atomi objektum megvalósítása. Állapítsuk meg, hogy ez milyen helyességi feltételeket elégít ki.

18-12. A 18.3.3. szakasz végén megadott optimalizált kölcsönös kizárási algoritmus több darabban van leírva: egy logikai idővel kibővített egyszerű A aszinkron algoritmusból, a MÁSOLTÁLLAPOTGÉP algoritmusból, és egy fő algoritmusból, amely a másolt sort használja. Írjunk előfeltétel/hatás kódot algoritmusunkra, és vázoljuk fel a helyesség bizonyítását.

18-13. A MÁSOLTÁLLAPOTGÉP algoritmus logikai időt használ egy atomi objektum üzenetszóró hálózati modellben történő megvalósítására. Hogyan lehet ezt úgy módosítani, hogy a küld/fogad hálózati modellben is működjön?

18-14. Adjuk meg a 18.4. tétel bizonyítását. Ez a transzformáció pontos leírását igényli.

18-15. Tervezzünk egy algoritmust *logikai időn* alapuló, egy író/több olvasó közös memória algoritmusok szimulálására egy aszinkron küld/fogad hálózatban. Ez a módszer legyen alternatívája a 17.1.2. szakaszban leírt *kétfázisú zárolás* stratégiájának. Egy x közös változó minden olvasójának rendelkeznie kell x-ről egy helyi másolattal. Az x-en végzett minden `olvas` és `ír` művelethez egy logikai időt kell rendelni, és ezeket a műveleteket a helyi másolatokon kell elvégezni, azok logikai idejeinek sorrendjében. Minden műveletnek garantáltan be kell fejeződnie.

Adjunk előfeltétel/hatás kódot, állapítsuk meg és bizonyítsuk be a helyességet, és elemezzük a bonyolultságot.

18-16. Általánosítsuk a 18.15. gyakorlatra adott válaszunkat a több író/több olvasó közös memória algoritmusokra vonatkozólag.

18-17. Tekintsük a logikai idő definíciójának egy gyengítését: a *gyenge logikai időt* úgy kapjuk, hogy nem követeljük meg a T-től a teljes rendezettséget, hanem megengedjük, hogy T csak részben rendezett halmaz legyen. A logikai idő négy tulajdonságának azonban teljesülnie kell. Így nem az összes, hanem csak az egymástól függő eseményeknek kell a logikai idők sorrendjében relációban állniuk (ugyanazon csúcspont eseményeinek, illetve az egyes `küld` és az azokhoz tartozó `fogad` eseményeknek).

(a) Adjunk a 18.1. tételre egy olyan változatot, amely egy gyenge logikai idejű kiosztás esetében fennáll. Ezt az adott részleges rendezéssel összeférő tetszőleges teljes rendezés fogalmaival kell megállapítani. Az eredményként kapott tételt bizonyítsuk is be.

(b) Fejlesszünk ki egy algoritmus-transzformációt, amellyel adott A aszinkron hálózati algoritmus egy végrehajtási sorozata számára *gyenge logikai idejű kiosztást* lehet létrehozni. Az eseményekhez társított időknek csak akkor kell a megfelelő T részleges rendezésben relációban lenniük, ha az események között függőség van. (*Útmutatás.* Egy algoritmus nemnegatív egészek n hosszúságú vektorainak T halmazán alapulhat. Azt mondjuk, hogy $C <_T C'$, feltéve, hogy $C(i) \leq C'(i)$, minden i-re, és $C(i) < C'(i)$, valamely i-re, azaz a C' vektor minden komponensében legalább akkora, mint a C, és valamely komponensében szigorúan nagyobb annál.

Minden A_i folyamat kezel egy helyi *órát*, amely egy T-beli vektor, kezdetben nullvektor. Ha az A_i-ben bekövetkezik egy esemény, az $óra_i(i)$-t legalább

1-gyel növelni kell. Ha A_i egy üzenetet küld, először megnöveli az $óra_i(i)$-t, majd az így kapott vektort időbélyegként az üzenethez csatolja. Ha A_i fogad egy üzenetet, akkor először megnöveli az $óra_i(i)$-t, majd beállítja *óra* vektorát úgy, hogy az az újonnan megnövelt *óra* vektor és az üzenet vektor időbélyegének a komponensenkénti maximuma legyen.)

Mutassuk meg, hogy az általunk adott transzformáció valóban egy gyenge logikai idejű kiosztást eredményez, és hogy az események időpontjai csak akkor állnak relációban T-ben, ha az események függnek egymástól.

19. fejezet

Ellentmondásmentes globális fényképek és stabil tulajdonságjelzés

Ebben a fejezetben bemutatjuk az aszinkron hálózatok programozását egyszerűsítő négy eljárásunk közül az utolsót, nevezetesen egy A aszinkron hálózati algoritmus futás közbeni figyelését. Egy figyelő algoritmus például

- segíthet A nyomon követésében, például a kívánt invariánsok megsértésének vizsgálatával;
- előállíthatja A globális állapotának biztonsági változatait;
- felismerheti, hogy A mikor fejezi be a végrehajtási sorozatot;
- felismerheti, ha A egyes folyamatai „holtpontra" jutnak, a folyamatok arra várnak, hogy a többi csináljon valamit;
- kiszámíthat egy A által kezelt globális mennyiséget (például a teljes pénzösszeget);

Ebben a fejezetben két fogalomra összpontosítunk: az *ellentmondásmentes globális fényképre* és a *stabil tulajdonság jelzésére*. Egy globális fénykép visszaadja A egy globális állapotát, azaz A folyamataihoz és csatornáihoz tartozó állapotok egy gyűjteményét. A fényképet „ellentmondásmentesnek" nevezzük, ha a folyamatokat *egy adott pillanatban tekinti a teljes rendszerben*. Egy ilyen fénykép a fent felsorolt feladatok mindegyikében hasznos. Az A globális állapotának minden olyan tulajdonsága stabil tulajdonság, amely ha egyszer teljesül, akkor utána mindig teljesülni fog. Stabil tulajdonság például a rendszer befejeződése és a holtpont.

Minden figyelő algoritmust az eredeti A algoritmus $B(A)$ transzformáltjaként írunk le; részletesebben, $B(A)$ ugyanazon a gráfon alapul, mint A, továbbá minden $B(A)_i$ folyamat csak a megfelelő A_i folyamat segítségével van megadva. $B(A)_i$ *nem* valamilyen új b/k automata és A_i együtteseként van kifejezve, mivel az új $B(A)_i$) folyamatnak hozzá kell férnie A_i állapotához. Ehelyett, $B(A)_i$ bizonyos új állapotkomponensek és műveletek hozzáadásával, valamint a régi műveleteken végrehajtott megfelelő módosításokkal van leírva. Ezeket a módosításokat

úgy hajtjuk végre, hogy ne zavarjuk meg nagyon A működését.

19.1.. Befejeződés jelzése terjesztő algoritmusokhoz

Először csak a befejeződésjelzési feladat vizsgálatát végezzük el egy olyan aszinkron küld/fogad A algoritmusra, amely különösen egyszerű típusú, ú.n. *terjesztő algoritmus*.

19.1.1.. A feladat

Tegyük fel, hogy az alapul szolgáló G gráf egy tetszőleges összefüggő, irányítatlan gráf. Tegyük fel továbbá, hogy A-ban minden folyamat kezdőállapota *csendes* (ahogy ezt a 8.1. alfejezetben definiáltuk). Vagyis csak bemeneti állapotok érhetők el. A-t egy olyan környezetben tekintjük, amely egy (tetszőleges) folyamathoz egyetlen bemeneti eseményt támogat. A b/k automata definíciójának megfelelően egy ilyen bemenet folyamatba való beérkezésekor kiválthatja a folyamat helyileg vezérelt műveleteinek végrehajtását, beleértve a többi folyamathoz való üzenetküldést is. Ezek az üzenetek aktiválhatják a címzett folyamatokat, amelyek további üzeneteket küldhetnek, és így tovább. Az A algoritmust *terjesztő* algoritmusnak nevezzük, mivel minden tevékenység a bemenet beérkezésének helyétől indul, majd üzenetek formájában „terjed végig" a hálózat egyes részein.

A egy globális állapotát *csendesnek* nevezzük, ha a folyamatok nem hajthatnak végre helyileg vezérelt műveletet, és nincsenek üzenetek a csatornákon. (Ez megint megfelel a *csendes* 8.1. alfejezetben szereplő definíciójának, ezúttal a teljes A algoritmust reprezentáló egyetlen b/k automatára értve.) Az A-ra vonatkozó *befejeződésjelzési feladat*[1] a következő: ha az A algoritmus elér egy csendes globális állapotot, miután egy A_i folyamatnál bemenet jelentkezik, akkor meg kell jelennie egy $\texttt{elvégezte}_i$ kimenetnek az i csúcsban.

A `elvégezte` kimenetet is tartalmazó tényleges befejeződés jelzése egy $B(A)$ *figyelő algoritmussal* tehető meg. A $B(A)$ algoritmusnak is egy küld/fogad hálózati algoritmusnak kell lennie, amely ugyanazon a G gráfon alapul, mint A. A $B(A)$ figyelő algoritmus minden $B(A)_i$ folyamatautomatáját a megfelelő A_i folyamatautomata segítségével kell definiálni. A $B(A)_i$ meghatározásához az A_i következő módosításait engedélyezzük.

- $B(A)_i$ az A_i algoritmus összes állapotkomponense mellett tartalmazhat új állapotkomponenseket.
- $B(A)_i$ kezdőállapotainak A_i állapotkomponenseire való projekciójának pontosan meg kell egyeznie A_i kezdőállapotaival.
- $B(A)_i$ az A_i algoritmus műveletei mellett tartalmazhat új bemeneteket, kimeneteket és belső műveleteket.
- A_i műveleteire új információk épülhetnek $B(A)_i$-ben, így például egy $\texttt{küld}(m)_i$ művelet átalakítható egy $\texttt{küld}(m, c)_i$ műveletté. A_i műveletei megtartják előfeltételeiket, és a taszkrész ugyanazon osztályában maradnak

[1] Ebben a fejezetben a „befejeződést" a csendességre értjük; a könyv más részeiben a legtöbb helyen úgy értelmezzük, mint a rendszer által adott választ.

$B(A)_i$-ben. Ugyanaz lesz a hatásuk, mint korábban A_i állapotkomponenseire, de hatással lesznek az új állapotkomponensekre is.

- $B(A)_i$ új bemeneti műveletei csak $B(A)_i$ új állapotkomponenseinek értékeit módosíthatják.
- $B(A)_i$) új helyileg vezérelt műveleteinek előfeltételei $B(A)_i$ teljes állapotát érinthetik, a régi és új állapotkomponenseket egyaránt. Az új helyileg vezérelt műveletek azonban csak $B(A)_i$ új állapotkomponenseire lehetnek hatással. Ők $B(A)_i$ taszkrészének új osztályaiba lesznek csoportosítva.

19.1.2.. A DijkstraScholten algoritmus

Most bemutatjuk a terjesztő algoritmusok befejeződésének jelzésére használható DijkstraScholten algoritmust. Az algoritmus alapötlete az alapul szolgáló A algoritmus kiterjesztése az A-ban jelenleg szereplő gráfcsúcsok feszítőfájának elkészítésével és kezelésével. A fa gyökere a *forráscsúcs*, ami az a csúcs, ahol a bemenet megjelenik. A feszítőfa elkészítése hasonlóan történik, mint a 15.3. alfejezet AszinkFeszFa algoritmusának esetében, ezúttal azonban a konstrukció bonyolultabb, mivel lehetővé teszi a fa ismételt zsugorítását és bővítését ugyanazon csúcs többszöri bevonásával. (Ugyanez a megközelítés megfigyelhető a 15.4. alfejezet AszinkSZK befejezési protokolljánál.)

A DijkstraScholten algoritmus (vázlatosan)

Az algoritmus által használt üzenetek A üzenetei, valamint még egy *nyugtáz* üzenet. A üzeneteit az AszinkFeszFa algoritmusban szereplő `keres` üzenetekhez hasonlóan kezeljük. A forráson kívül az összes folyamat a feszítőfában szülőként megjelöli azt a szomszédját, amelyből egy A üzenetet kap. A minden további üzenete azonnali nyugtázásra kerül; csak az első nem kerül nyugtázásra (egyelőre). A forrásfolyamat szintén azonnal nyugtázza a kapott A üzeneteket. Így A üzeneteinek hálózaton belüli szétküldésével elkészül a csúcspontoknak a protokollban szereplő feszítőfája.

Ezután a befejeződések forrásfolyamathoz való visszajelzéséhez egy konvergens üzenetszóró eljárással engedélyezzük a feszítőfa „zsugorítását". Speciálisan, minden DijkstraScholten$(A)_i$ folyamat olyan helyzetet keres, amikor a következő helyi feltételek egyszerre teljesülnek:

1. A_i állapota csendes;
2. A minden kimenő üzenete nyugtázva van.

Ha megtalálja azt, „szabadítani" kezd: egy, a forrásfolyamattól eltérő folyamat nyugtát küld a szülőjének, és töröl minden információt a protokollról, mialatt egy forrásfolyamat beszámol a teljesítésről.

Hasonló szabadító folyamatra láthatunk példát a 15.4. alfejezet AszinkSzórásNyugtáz algoritmusának leírásában. Jelen esetben viszont egy folyamat a szabadítás után kaphat egy másik A-üzenetet, aminek eredményeként ismét részt

vesz a feszítőfa elkészítésében. Valójában ez akárhányszor megtörténhet az alapul szolgáló A algoritmus üzenetküldő mintájától függően. Vagyis a DIJKSTRA-SCHOLTEN(A) algoritmus feszítőfája különböző időkben különböző módon folyamatosan zsugorodhat és bővülhet.

19.1.1. példa. A feszítőfa bővítése és zsugorítása

Tegyük fel, hogy az alapul szolgáló G gráf az 1, 2, 3 és 4 csúcsokból áll, a 19.1. ábrán látható módon összekötve (az ábrán a csúcsokhoz tartozó A_1, A_2, A_3 és A_4 komponensek vannak), és tekintsük az ábrán látható konfigurációt. A leírásban a DIJKSTRASCHOLTEN(A)$_i$ folyamat rövidítéséhez a DS(A)$_i$ jelölést használjuk.

(a) A_1 bemenetet kap, aktiválódik és üzenetet küld A_2 és A_3 szomszédainak.

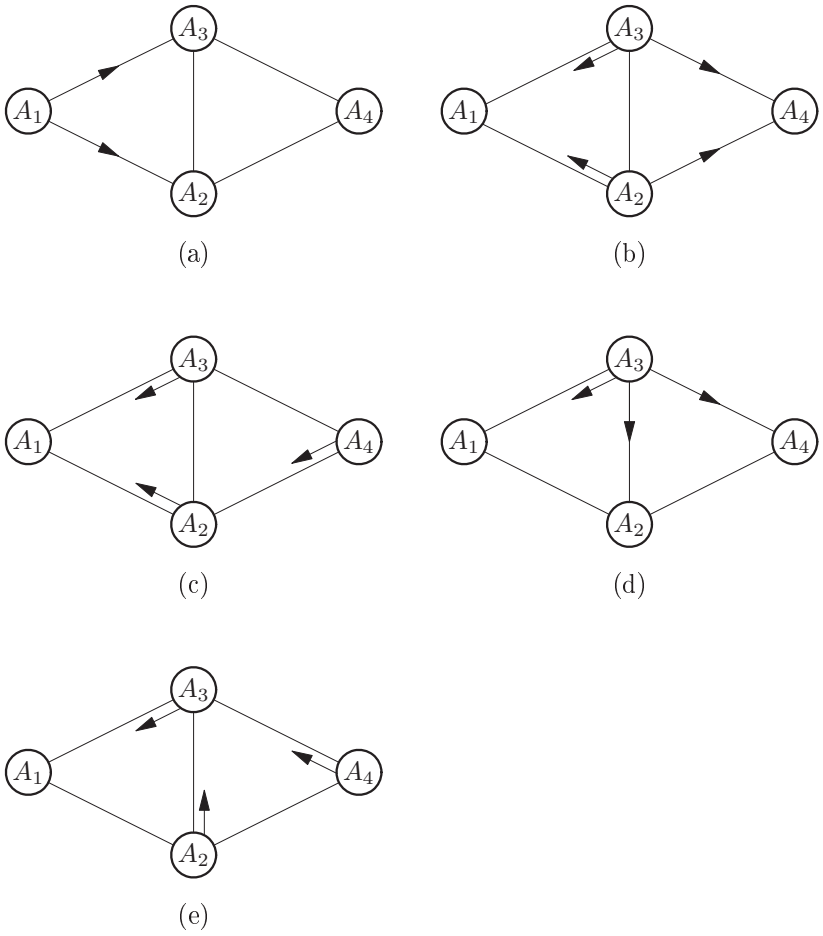
(b) $DS(A)_2$ és $DS(A)_3$ megkapják az üzenetet az A_1-től, és úgy állítják be a *szülő* mutatójukat, hogy az az 1 csúcsra mutasson. Ekkor A_2 és A_3 aktiválódnak és üzenetet küldenek egymásnak. Mivel mind $DS(A)_2$-nek, mind pedig $DS(A)_3$-nak már van szülője, nyugtával válaszolnak. Ezután A_2 és A_3 is üzenetet küld A_4-nek.

(c) A_2 üzenete először a $DS(A)_4$ folyamatot éri el, így a $DS(A)_4$ a *szülő* mutatóját 2-re állítja, és azonnal nyugtázza az A_3-tól jött üzenetet. Az A_1, A_2, A_3 és A_4 folytatják munkájukat szükség esetében üzenetek küldésével egymásnak; minden üzenet azonnal nyugtázásra kerül. Ezután A_2 csendes állapotba kerül. $DS(A)_2$ még nem kezdheti meg a szabadítást, mivel még nem kapott nyugtát A_4-hez küldött kezdeti üzenetére.

(d) A_4 csendes állapotba kerül. Mivel $DS(A)_4$-nek nincsenek még nem nyugtázott A-üzenetei, nyugtát küld $DS(A)_2$ szülőjének, majd szabadításba kezd, elfelejtve mindent a protokollban való részvételéről. Amikor a $DS(A)_2$ megkapja ezt a nyugtát, az A_2 még mindig csendes állapotban van, és a $DS(A)_2$ mostanra nyugtát kapott az összes kimenő A-üzenetére; így $DS(A)_2$ nyugtát küld $DS(A)_1$ szülőjének, majd szabadítani kezd. Ezután A_3 üzenetet küld A_2-nek és A_4-nek.

(e) Amikor A_2 és A_4 megkapja ezt az üzenetet, a korábbihoz hasonló módon aktiválódnak, visszaállítják a *szülő* mutatójukat 3-ra, és folytatják A munkáját.

Ez a végrehajtási sorozat ezen a módon bizonytalan ideig folytatódhat a feszítőfa egyes részeinek zsugorodásával és bővülésével és A megfelelő részeinek csendesedését követve. Ha azonban egyszer az összes A algoritmus csendessé válik, a feszítőfa az 1 forráscsúccsá zsugorodik össze. Ha A_1 csendes állapotba kerül, és $DS(A)_1$ minden kimenő üzenetére nyugtát kap, akkor $DS(A)_1$ jelezheti a befejeződést.



19.1.. ábra. A DijkstraScholten(A) algoritmus egy végrehajtási sorozata. Az élen látható nyíl egy küldött üzenetre utal; egy éllel párhuzamos nyíl a *szülő* mutatót jelzi.

Az alábbiakban a DijkstraScholten(A) algoritmus i folyamatának kódját ismertetjük. A *hiány* változót az elintézetlen nyugták számának nyomon követéséhez használjuk.

19.1. automata. DijkstraScholten(A)$_i$

Lenyomat:
mint az A_i esetében, továbbá:
Bemeneti:
`fogad(„nyugtáz”)`$_{j,i}$, $j \in$ *szomszédok*
Belső:
`szabadít`$_i$
Kimeneti:
`küld(„nyugtáz”)`$_{i,j}$, $j \in$ *szomszédok*
`elvégezte`$_i$

Állapotok:
Mint az A_i esetében, továbbá:
státus $\in \{$*tétlen, forrás, nem_forrás*$\}$, kezdetben *tétlen*
szülő $\in$ *szomszédok* $\cup \{$*null*$\}$, kezdetben *null*
$\forall j \in$ *szomszédok*:
küld_puffer(j), `nyugtáz` üzenetek egy FIFO sora, kezdetben üres
hiány(j) $\in \mathbb{N}$, kezdetben 0

Átmenetek:

A_i bemenete $\neq$ `fogad`
  Hatás:
    mint A_i esetében, továbbá:
    státus := *forrás*

`fogad(m)`$_{j,i}$, m egy A-üzenet
  Hatás:
    mint A_i esetében, továbbá:
    if *státus* = *tétlen* **then**
      státus := *nem_forrás*
      szülő := j
    else adjuk „`nyugtáz`”-t
      küld_puffer(j)-hez

A_i helyileg vezérelt művelete $\neq$ `küld`
  Előfeltétel:
    mint A$_i$ esetében
  Hatás:
    mint A$_i$ esetében

`küld(m)`$_{i,j}$, m egy A-üzenet
  Előfeltétel:
    mint A_i esetében
  Hatás:
    mint A_i esetében, továbbá:
    hiány(j) := *hiány*(j) + 1

`küld(„nyugtáz”)`$_{i,j}$
  Előfeltétel:
    „`nyugtáz`” az első
      küld_puffer(j)-ben
  Hatás:
    töröljük *küld_puffer*(j) első elemét

`fogad(„nyugtáz”)`$_{j,i}$
  Hatás:
    hiány(j) := *hiány*(j) − 1

`szabadít`$_i$
  Előfeltétel:
    státus = *nem_forrás*
    A_i állapota csendes
    $\forall k \in$ *szomszédok*
      hiány(k) = 0
  Hatás:
    adjuk „`nyugtáz`”-t
      küld_puffer(*szülő*)-höz
    státus := *tétlen*
    szülő := *null*

`elvégezte`$_i$
  Előfeltétel:
    státus = *forrás*
    A_i állapota csendes
    $\forall k \in$ *szomszédok*
      hiány(k) = 0
  Hatás:
    státus := *tétlen*

Taszkok:

Mint A_i esetében, továbbá:
{`elvégezte`$_i$}
{`szabadít`$_i$}
$\forall k \in$ *szomszédok* : {`küld(„nyugtáz”)`$_{i,j}$}

- -

Nyilvánvaló, hogy a DijkstraScholten(A) algoritmus minden globális állapota az A egy globális állapotát, illetve a DijkstraScholten(A) minden pártatlan végrehajtási sorozata A egy pártatlan végrehajtási sorozatát adja. Annak megmutatásához, hogy DijkstraScholten(A) valóban helyesen ismeri fel az A befejeződését, először belátunk egy több részből álló invariáns lemmát. A két utolsó invariáns a kulcs: az utolsó előtti azt állítja, hogy a *szülő* mutatók a nem *tétlen* folyamatok egy feszítőfáját alkotják, míg az utolsóból az következik, hogy egy `elvégezte` üzenet A csendességére utal.

19.1. lemma. . DijkstraScholten(A) *minden állapotában egy i csúcson lévő bemenetet tartalmazó végrehajtási sorozat után a következők állnak fenn:*

1. *státus$_i \in$ {forrás, tétlen} és szülő$_i$ = null;*
2. *minden $j \neq i$ esetében státus$_j \in$ {tétlen, nem_forrás}, és ha státus$_j$ = nem_forrás, akkor szülő$_j \neq$ null;*
3. *minden j-re, ha státus$_j$ = tétlen, akkor az A_j vetített állapota csendes, szülő$_j$ = null és hiány$(k)_j = 0$ minden k-ra;*
4. *minden j-re és k-ra a hiány$(k)_j$ a következő mennyiségek összege: a j-ből k-ba vezető csatornán lévő A-üzenetek száma, a küld_puffer$(j)_k$-ban lévő* `nyugtáz` *elemek száma, a k-ból j-be vezető csatornán lévő* `nyugtáz` *elemek száma, plusz 1, ha a szülő$_k = j$;*
5. *ha státus$_i$ = forrás, akkor a szülő mutatók egy i gyökerű irányított fát alkotnak, amely pontosan az státus $\neq$ tétlen tulajdonságú csúcsokat feszíti ki;*
6. *ha státus$_i$ = tétlen, akkor minden j-re státus$_j$ = tétlen, és minden csatorna üres.*

Bizonyítás. A bizonyítást meghagyjuk gyakorlatnak (lásd 19-2. gyakorlat). □

19.2. tétel. . *A* DijkstraScholten*(A) algoritmus felismeri egy A terjesztő algoritmus befejeződését.*

Bizonyítás. A 19.1. lemma 6. és 3. pontjából következik, hogy amikor a DijkstraScholten(A) algoritmus befejeződést jelez, akkor A valóban csendessé válik. Meg kell mutatnunk még a szükséges élénkségi tulajdonságot: ha A csendessé válik, akkor a DijkstraScholten(A) algoritmus ténylegesen befejeződést jelez.

Az ellentmondás eléréséhez tekintsük a DijkstraScholten(A) algoritmus egy olyan α pártatlan végrehajtási sorozatát, amelyben az A algoritmus csendessé válik, és amelyben nem jelenik meg `elvégezte` esemény. Ekkor a csendessé válás után nem lesznek küldve és fogadva további A üzenetek; vagyis a *szülő* mutatók által meghatározott fa (ahogyan ez a 19.1. lemma 5. pontjában szerepel) nem bővülhet tovább. Végül a fának abba kell hagynia a zsugorodást, és meg kell állapodnia egy rögzített T fánál. (Ennek a fának legalább a forráscsúcsot tartalmaznia kell, mivel feltételezzük, hogy egyáltalán nem kerül sor `elvégezte` eseményre.) Mivel nincsenek további A-üzenetek, és a fa sem változik, így a globális állapotban sehol nem lesznek további `nyugtáz` üzenetek. Ezt követően a 19.1. lemma 4. pontjában szereplő összeg első három tagjának 0-nak kell lennie minden *hiány*$(k)_j$-re, egy *hiány*$(k)_j$ pedig csak úgy lehet nemnulla, ha *szülő*$_k = j$. Ekkor azonban a T minden i levele végrehajthat `szabadít` műveletet, amire így végül sor is kerül. Ebből viszont az következik, hogy T tovább zsugorodik, ami ellentmondás. Vagyis az α-ban meg kell jelennie egy `elvégezte` eseménynek. □

Bonyolultságelemzés.. Tekintsük a DijkstraScholten(A) algoritmus egy `elvégezte` eseményt tartalmazó végrehajtási sorozati sorozatát. Az üzenetküldések száma az α-ban összesen $2m$, ahol m az A algoritmus tartalmazott végrehajtási sorozatában küldött üzenetek száma. $\mathcal{O}(m(\ell + d))$ a felső korlátja az A

lecsendesedése és az `elvégezte` esemény bekövetkezte között eltelt időnek, ahol ℓ és d a szokott módon definiált. Vegyük észre, hogy a kommunikációs és időbonyolultság nem a hálózat méretétől függ közvetlenül, hanem a küldött A üzenetek számától. Ha A csak egy rövid ideig a hálózat egy kis részén működik, akkor általában kis számú üzenetet fog küldeni, így a DijkstraScholten(A) csak kis költségekkel jár. Ha viszont A sok üzenetet küld, akkor a DijkstraScholten(A) költségessé válik.

19.1.2. példa. Szélességi keresés

Emlékezzünk vissza a 15.4. alfejezet AszinkSZK algoritmusára, amelyben a folyamatok helyesbítik a hibás *szülő* információkat azok stabilizálódásáig. A leírtaknak megfelelően az algoritmus nem fejeződik be, mivel a folyamatok nem tudják megállapítani, hogy mikor válik csendessé az algoritmus.

Az AszinkSZK terjesztő algoritmusként való reprezentálásához egy kis módosítást hajtunk végre, csendessé téve induláskor az i_0 folyamatot, és egy `ébreszt` bemeneti művelettel aktiválva azt. Ezután alkalmazzuk a DijkstraScholten(A) algoritmust egy befejező szélességi kereső algoritmus megadásához. Ez az AszinkSZK algoritmusnál közölt ad hoc befejezési stratégia egy módszeres változata.

19.2.. Ellentmondásmentes globális fényképek

Most megvizsgáljuk egy *ellentmondásmentes globális fénykép* elkészítésének problémáját egy futó A aszinkron küld/fogad hálózati algoritmusról. Szemléletesen akkor nevezünk egy fényképet „ellentmondásmentesnek”, ha az a folyamatokat egy adott pillanatban egyszerre tekinti a teljes rendszerben.

19.2.1.. A feladat

Ismét feltesszük, hogy az alapul szolgáló G gráf egy tetszőleges összefüggő, irányítatlan gráf. Az alapul szolgáló A algoritmus ezúttal egy tetszőleges küld/fogad hálózati algoritmus. A fényképet egy $B(A)$ figyelő algoritmus készíti, ami ugyancsak egy G gráfon alapuló küld/fogad hálózati algoritmus. A $B(A)$ figyelő algoritmus minden $B(A)_i$ folyamat-automatája ismételten a megfelelő A_i folyamat segítségével van megadva.

Ezúttal kicsit általánosabb típusú módosításokat engedélyezünk, mint a 19.1.1. szakaszban, de ezek még mindig elegendőek ahhoz, hogy a $B(A)$ minden pártatlan végrehajtási sorozata „tartalmazza” A egy pártatlan végrehajtási sorozatát. A különbség az, hogy most azt is engedélyezzük, hogy a $B(A)_i$ „késleltesse” A_i egy $\texttt{küld}(m)_{i,j}$ üzenetét addig, amíg $B(A)_i$ egy másik m előtti üzenetet helyez az i-ből j-be vezető csatornára.

Tegyük fel, hogy minden $B(A)_i$-nek van egy $\texttt{fényképez}_i$ bemeneti művelete, amely A fényképének elkészítésére utasít. Megköveteljük, hogy a $B(A)$ minden

legalább egy `fényképez` bemeneti eseményt tartalmazó pártatlan végrehajtási sorozatában minden $B(A)_i$ állítson elő egy `jelentés`$_i$ kimenetet, amely tartalmazza A_i egy állapotát és az A_i-be beérkező összes A-beli csatorna állapotát.

A $B(A)_i$-k által jelentett összes állapot A egy globális állapotát adják. Megköveteljük, hogy ez az állapot ellentmondásmentes legyen. Nevezetesen, legyen α A-nak az a pártatlan végrehajtási sorozata, amely a $B(A)$ adott pártatlan végrehajtási sorozatában szerepel. A-nak rendelkeznie kell egy olyan α' pártatlan végrehajtási sorozattal, amely kielégíti a következő feltételeket:

1. az α' megkülönböztethetetlen az α-tól minden A_i folyamatra;
2. az α' az α-nak az α_1 előtagjával kezdődik, amely a B(A) adott végrehajtási sorozatában elsőként megjelenő `fényképez` esemény előtt szerepel;
3. az α' az α-nak az α_2 utótagjára végződik, amely a B(A) adott végrehajtási sorozatában utolsóként megjelenő `jelent` esemény után szerepel;
4. a visszaadott állapot pontosan az az α' egy előtagja után lévő globális állapot lesz, amely tartalmaz minden α_1-et, de egyetlen α_2-t sem.

Így amennyire a folyamatoktól megtudható, a visszaadott globális állapot az A végrehajtási sorozatának egy adott pontján azonnal elkészül. Továbbá, ez a pont valahol a fényképező algoritmus végrehajtási sorozatának kezdete és vége között található.

19.2.1. példa. Banki rendszer

Legyen A a 18.3.1. példában szereplő banki rendszer. A 18.4. ábra A egy végrehajtási sorozatát mutatja, amely öt pénzmozgást tartalmaz a rendszer három folyamata között. (Ne vegyük figyelembe a diagramok logikai időcímkéjét.) Tegyük fel, hogy a $B(A)$ figyelő algoritmus egy folyamata egy `fényképez` bemenetet kap a végrehajtási sorozat kezdetekor. Ekkor a 18.5. ábrán egy ellentmondásmentes globális fénykép algoritmus által visszaadható globális állapotra láthatunk példát. Itt $B(A)_1$, $B(A)_2$ és $B(A)_3$ az A_1, A_2 és A_3 állapotaiként $14\mathcal{E}$-t, $18\mathcal{E}$-t, illetve $26\mathcal{E}$-t ad vissza. Minden csatornát üresnek tekintünk az A_3-ból A_2-be vezető csatorna kivételével, amelyről a $B(A)_2$ jelentést ad, hogy egy $2\mathcal{E}$ értékű üzenetet tartalmaz. A szükséges alternatív α' végrehajtási sorozat a 18.6. ábrán látható.

19.2.2.. A ChandyLamport algoritmus

Már leírtunk egy megoldást az ellentmondásmentes globális fénykép feladat megoldására – a 18.3.2. szakasz LogikaiIdejűFénykép algoritmusát. Most egy másik algoritmust, a ChandyLamport *globális fényképező algoritmust* közöljük, ami nagyon hasonlít a LogikaiIdejűFénykép algoritmusra, de nem használ t explicit logikai időt. Helyette (ahogy ez a 18-8. gyakorlatban javasoltuk) új `jelez` üzeneteket használ azon osztáspontok megjelölésére, amelyek a t alkalommal vagy annál kevesebbszer, illetve többször elküldött üzeneteket elválasztják.

A ChandyLamport algoritmus (vázlatosan)

Amikor egy korábban a fényképező algoritmusba be nem vont ChandyLamport$(A)_i$ folyamat egy `fényképez`$_i$ bemenetet kap, feljegyzi az A_i aktuális állapotát. Ezután rögtön szétküld egy `jelez` üzenetet minden kimeneti csatornáján; ez a `jelez` megjelöli a határt a helyi állapot feljegyzése előtt és azután kiküldött üzenetek között. [2]

Ezután a ChandyLamport$(A)_i$ megkezdi a bejövő csatornákon érkező üzenetek rögzítését a csatorna egy állapotának megállapításához; a csatornán lévő üzeneteket egy `jelez` megjelenéséig rögzíti. Ezen a ponton a ChandyLamport$(A)_i$ a csatornán küldött összes üzenetet rögzíti, mielőtt a másik végen lévő szomszéd rögzítené a helyi állapotát.[3]

Még egy esetet kell áttekintenünk: tegyük fel, hogy a ChandyLamport$(A)_i$ folyamat `jelez` üzenetet kap, mielőtt rögzítené az A_i állapotát. Ebben az esetben rögtön az első `jelez` üzenet beérkezésekor a ChandyLamport$(A)_i$ rögzíti az A_i aktuális állapotát, a `jelez` üzeneteket küld ki, és megkezdi a beérkező üzenetek rögzítését. Üresként rögzíti azt a csatornát, amelyen a `jelez` érkezett.

A kód a következő..

19.2. automata. ChandyLamport$(A)_i$

Lenyomat:
mint A_i esetében, továbbá:
Bemeneti:
`fényképez`$_i$
`fogad(„jelez”)`$_{j,i}$, $j \in$ *szomszédok*
Kimeneti:
`jelent`$(s, C)_i$, $s \in$ *státusok*(A_i), C a *szomszédok* leképezése A-üzenetek véges sorozatára
`küld(„jelez”)`$_{i,j}$, $j \in$ *szomszédok*
Belső:
`belső_küld(m)`$_{i,j}$, $j \in$ *szomszédok*, m az A algoritmus egy üzenete

Állapotok:
mint A_i esetében, továbbá:
státus $\in$ {*kezdő, fényképező, jelentett*}, kezdetben *kezdő*
fénykép_ állapot, A_i egy állapota vagy *null*, kezdetben *null*
$\forall j \in$ *szomszédok*:
fényképezett_ csat(j), egy logikai érték, kezdetben *hamis*
küld_ puffer(j), A-üzenetek és `jelez` üzenetek FIFO sora, kezdetben üres
csatorna_ fénykép(j), A-üzenetenek FIFO sora, kezdetben üres

[2] Ha például A egy olyan banki rendszer, amilyet a 19.2.1. példában leírtunk, akkor a jelző előtt küldött pénz *nem* lesz feljegyezve a feladó helyi állapotában, míg a jelző után küldött pénz *igen*.

[3] A banki példában ez azt jelenti, hogy a ChandyLamport$(A)_i$ összeszámolja a szomszéd által küldött összes pénzt a helyi állapot rögzítése előtt, így azt a szomszéd nem számolja el.

Átmenetek:

`fényképez`$_i$
  Hatás:
    if *státus* = *kezdő* **then**
      fénykép_ állapot := A_i állapota
      státus := *fényképező*
      $\forall j \in$ *szomszédok* **do**
        adjuk `jelez`-t a
          `küld_puffer`(j)-hez

A_i bemenete $\neq$ `fogad`
  Hatás:
    mint A_i esetében.

`fogad(m)`$_{j,i}$, m egy A-üzenet
  Hatás:
    Mint A_i esetében, továbbá:
    if *státus* = *fényképező*
      and *fényképezett_ csat*(j) = *hamis*
        then adjuk m-et a
          csatorna_fénykép(j)-hez

`jelent`$(s, C)_i$
  Előfeltétel:
    státus = *fényképező*
    $\forall j \in$ *szomszédok*
      fényképezett_ csat(j) = *igaz*
    s = *fénykép_ állapot*
    $\forall j \in$ *szomszédok*
      $C(j)$ = *csatorna_fénykép*(j)
  Hatás:
    státus := *jelentett*

`fogad`(„`jelez`”)$_{j,i}$
  Hatás:
    if *státus* = *kezdő* **then**
      fénykép_ állapot := A_i állapota
      státus := *fényképező*
      $\forall j \in$ *szomszédok* **do**
        adjuk `jelez`-t
          küld_ puffer(j)-hez
    fényképezett_ csat(j) := *igaz*

A_i helyileg vezérelt művelete $\neq$ `küld`
  Előfeltétel:
    Mint A_i esetében.
  Hatás:
    Mint A_i esetében.

`belső_küld`$(m)_{i,j}$
  Előfeltétel:
    Mint a `küld`(m)$_{i,j}$-re A_i-ben.
  Hatás:
    adjuk m-et *küld_ puffer*(j)-hez

`küld`$(m)_{i,j}$
  Előfeltétel:
    m az első a *küld_ puffer*(j)-ben
  Hatás:
    töröljük *küld_ puffer*(j) első elemét

Taszkok:
mint az A_i esetében, kivéve:
A_i-beli `küld`-öknek megfelelő taszkokban szereplő `belső_küld` műveletek
továbbá vannak új taszkok is:
  $\{$`jelent`$(s, C)_i : s \in$ *státusok*(A_i), C egy leképezés$\}$
  $\forall j \in$ *szomszédok* : $\{$`küld`$(m)_{i,j}$: m egy üzenet$\}$

- -

19.3. tétel. . *A* ChandyLamport*(A) algoritmus meghatározza az A algoritmus egy ellentmondásmentes globális fényképét.*

Bizonyítás. Rögzítsük a ChandyLamport(A) algoritmus egy olyan pártatlan végrehajtási sorozatát, amelyben valamelyik folyamat egy `fényképez` bemenetet kap. Először azt mutatjuk meg, hogy minden folyamat végül elkészít egy `jelent`

kimenetet. Amint egy `fényképez` bemenet megjelenik valamelyik CHANDYLAMPORT$(A)_i$ folyamatnál, a folyamat rögzíti A_i állapotát, és `jelez` üzeneteket küld szét az összes csatornáján. Ahogy egy másik CHANDYLAMPORT$(A)_j$ folyamat `jelez` üzenetet kap valamelyik csatornáján, rögzíti A_j állapotát, és `jelez` üzeneteket küld szét az összes csatornáján, ha ezt korábban még nem tette meg. A gráf összefüggősége miatt a `jelez` üzenetek végül minden folyamathoz eljutnak, és minden folyamat rögzíti a helyi állapotát. Továbbá az összes CHANDYLAMPORT$(A)_i$ folyamat egy idő után befejezi a bejövő csatornáin érkező üzenetek összegyűjtését (amikor már minden csatornáján kapott `jelez` üzenetet). Ezt követően, a várt módon minden CHANDYLAMPORT$(A)_i$ folyamat végül végrehajt egy `jelent` műveletet.

Most azt látjuk be, hogy a visszaadott globális állapot ellentmondásmentes. Jelölje α az A algoritmus tartalmazott pártatlan végrehajtási sorozatát (ahol az α `küld` eseményei megfelelnek a CHANDYLAMPORT(A) végrehajtási sorozatában szereplő `belső_küld` eseményeknek), és megadjuk a szükséges α' alternatív végrehajtási sorozatt és annak szükséges előtagját. Nevezetesen, legyen α_1 az α végrehajtási sorozat első `fényképez` előtti része, α_2 pedig az α végrehajtási sorozat utolsó `jelent` utáni része. Az α' végrehajtási sorozat α_1-gyel kezdődik és α_2-re végződik; az átrendezés csak az α első `fényképez` és utolsó `jelent` közötti eseményeit érinti.

α minden első `fényképez` és utolsó `jelent` közötti eseménye valamelyik CHANDYLAMPORT$(A)_i$ folyamatban jelenik meg. Ezeket az eseményeket két csoportba sorolhatjuk: S_1 – azok, amelyek megelőzik a CHANDYLAMPORT$(A)_i$ azon eseményét (`fényképez`$_i$ vagy `fogad(jelez)`$_{j,i}$), ahol az A_i állapota rögzítésre került, illetve S_2 – azok, amelyek ezt az eseményt követik. Az átrendezés az S_1-beli eseményeket az S_2-beli események elé helyezi, az A_i-k eseményeinek és a (`belső_küld`-ökből származtatott) `küld`-ök sorrendjének megőrzésével a megfelelő `fogad`-ra nézve. Egy ilyen átrendezés lehetősége azon múlik, hogy az A_i állapotának rögzítése után nincs olyan `belső_küld`$(m)_{i,j}$ esemény, amelyhez tartozó `fogad`$(m)_{i,j}$ esemény megelőzné az A_j állapotának rögzítését. (Ha az A_i állapotának rögzítése után egy `belső_küld`$(m)_{i,j}$ esemény következik, akkor az m a *küld_puffer*$(j)_i$-be kerül a `jelez` után. Ebből viszont az következik, hogy a `jelez` az m előtt érkezik meg a CHANDYLAMPORT$(A)_j$-hez, ami azt jelenti, hogy az m beérkezésekor A_j állapota már rögzítve van.) α eseményeinek ilyen átrendezése, valamint A_i állapotai α-nak megfelelő kitöltése megadja az α' sorozatot.

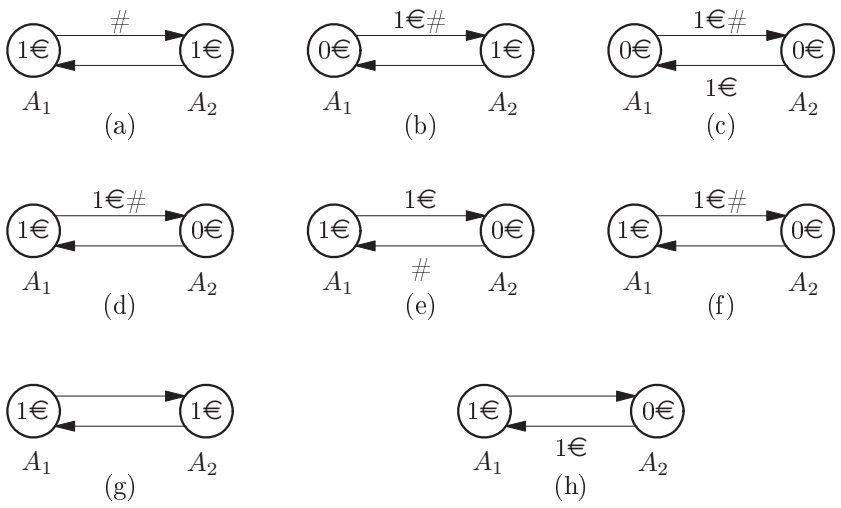
Tekintsük most α'-nek az α_3 előtagját, amely pontosan az S_1-beli események után végződik. Azt állítjuk, hogy α' és annak α_3 előtagja kielégíti a kívánt feltételeket; ennek kulcsa, hogy a folyamatok által visszaadott eredmények pontosan az A-nak az α_3 utáni globális állapotát adják. Vegyük észre, hogy egy A_i visszaadott állapota pontosan az A_i-nek az α_3 utáni állapota, mivel az α_3 definíció szerint pontosan az A_i állapotának rögzítése előtt szereplő A_i-beli eseményekből áll. Azt is ellenőriznünk kell, hogy a csatornák rögzítései pontosan az A csatornáin α_3 után lévő üzeneteket adják. α_3 után i-ből j-be menő üzenetek azonban pontosan azok az üzenetek, amelyek `belső_küld`$(\mathrm{m})_{i,j}$ eseményei A_i állapotának rögzítése előtt, `fogad`$(m)_{i,j}$ eseményei pedig az A_j állapotának rögzítése után jelennek meg. Ezek pontosan azok az üzenetek, amelyek a CHANDYLAMPORT$(A)_i$-ből a

ChandyLamport$(A)_j$-be a `jelez` előtt érkeznek meg, miután a ChandyLamport$(A)_j$ is rögzíti A_j állapotát, és egyúttal pontosan azok az üzenetek, amelyeket ChandyLamport$(A)_j$ rögzít erre a csatornára. □

Irányított gráfok.. Könnyen látható, hogy a ChandyLamport algoritmus erősen összefüggő irányított gráfokban és összefüggő irányítatlan gráfokban egyaránt működik.

19.2.2. példa. Kéteurós bank

Legyen A a 18.3.1. példában szereplő banki rendszer egyszerű, speciális változata, amelyben az alapul szolgáló G gráfnak csak két csúcsa van, 1 és 2, és ahol a teljes rendszerben mozgó pénzösszeg $2\mathcal{E}$. Tegyük fel, hogy minden folyamat $1\mathcal{E}$-val indul. A ChandyLamport$(A)_i$ folyamat rövid kiírásához a CL(A)$_i$ jelölést használjuk.



19.2.. ábra. A ChandyLamport algoritmus végrehajtási sorozata kéteurós bankra.

Tekintsük a ChandyLamport(A) algoritmus 19.2. ábrán látható pártatlan végrehajtási sorozatát. Ezen az ábrán a # szimbólum jelöli a `jelez` üzeneteket.

(a) `fényképez`$_1$ jelenik meg, ezért CL$(A)_1$ rögzíti A_1 állapotát $1\mathcal{E}$-ként. Ezután a $L(A)_1$ `jelez` üzenetet küld $L(A)_2$-nek, és megkezdi a bejövő üzenetek rögzítését.

(b) A_1 $1\mathcal{E}$-t küld A_2-nek; az euro a `jelez` után kerül CL$(A)_1$-ből CL$(A)_2$-be vezető csatornára.

(c) A_2 $1\mathcal{E}$-t küld az A_1-nek.

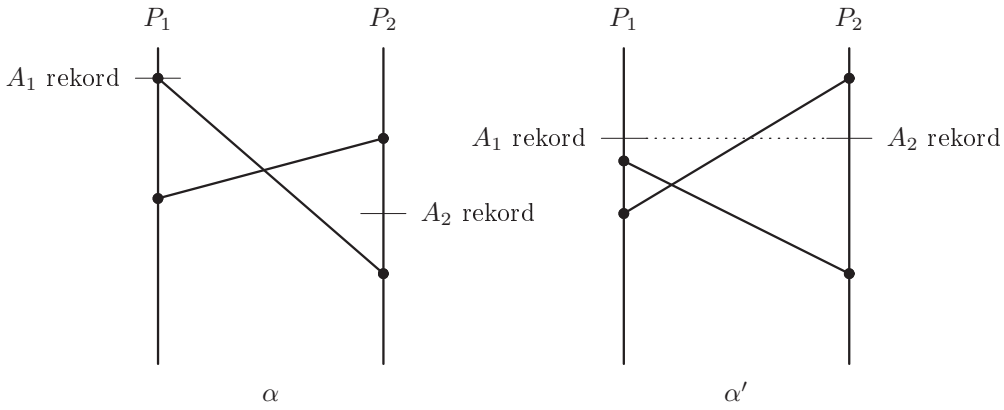
(d) S_1 megkapja az eurot, és a $\mathrm{CL}(A)_1$ rögzíti azt a `csatorna_fénykép`$(2)_1$-ben.

(e) $\mathrm{CL}(A)_2$ megkapja a `jelez`-t $\mathrm{CL}(A)_1$-től, rögzíti az A_2 állapotát $0\mathcal{E}$-ként, `jelez`-t küld a $\mathrm{CL}(A)_1$-nek, üresként rögzíti a bejövő csatorna állapotát, és jelenti az eredményeit.

(f) A $\mathrm{CL}(A)_1$ megkapja a `jelez`-t a $\mathrm{CL}(A)_2$-től, a bejövő csatorna állapotát egy üzenetet (a `jelez` előtt kapott $1\mathcal{E}$-t) tartalmazó sorozatként rögzíti, és jelenti az eredményeit.

(g) A_2 megkapja az eurot.

(h) Az algoritmus által visszaadott globális állapot az ábra (h) részén látható. Eszerint $1\mathcal{E}$ van az A_1-ben, $1\mathcal{E}$ az A_2-ből A_1-be vezető csatornán, és nincs pénz A_2-ben, illetve az A_1-ből A_2-be vezető csatornán. Ez együttesen megadja a teljes $2\mathcal{E}$ összeget.



19.3.. ábra. Küld/fogad diagramok az α és α' végrehajtási sorozatokra.

Vegyük észre, hogy a fényképező algoritmus által visszaadott globális állapot nem jelenik meg ténylegesen A tartalmazott α pártatlan végrehajtási sorozatában. Megjelenik viszont A egy másik α' pártatlan végrehajtási sorozatában, amelyben az események a következő sorrendben követik egymást: (a) A_2 $1\mathcal{E}$-t küld A_1-nek. (b) A_1 $1\mathcal{E}$-t küld A_2-nek. (c) A_1 kap $1\mathcal{E}$-t. (d) A_2 kap $1\mathcal{E}$-t. A 19.3. ábra küld/fogad diagramokat mutat az α-hoz és az α'-höz. Az α-ra vonatkozó diagram azt jelzi, hogy az A_1 és A_2 állapota hol került rögzítésre a Chandy-Lamport(A) végrehajtási sorozatában. Az α'-re vonatkozó diagram megmutatja, hogy ez a két rögzítési pont hogyan helyezkedik el az α' konstrukciójában. A fényképező algoritmus által visszaadott állapotot a második diagram vízszintes vonalával ábrázolt állapot adja meg.

Bonyolultságelemzés.. A ChandyLamport(A) algoritmus A üzenetei mellett $\mathcal{O}(|E|)$ üzenetet használ. Az első `fényképez` eseménytől az utolsó `jelent` eseményig eltelő idő A-nak a csatornákon és a *küld_puffer*-ekben felhalmozódott üzenetei számától függ. Ha ezeket figyelmen kívül hagyjuk, akkor $\mathcal{O}(\mathit{átm}(\ell + d))$ időkorlátot kapunk, de valószínűleg nem ésszerű ezektől eltekinteni. A valósághoz közelebb álló időkorlátot kapunk, ha A-nak a fénykép elkészítésével kapott globális állapotban bárhol előforduló összes üzenetével számolunk.

19.2.3.. Alkalmazások

Ebben a szakaszban ellentmondásmentes globális fényképekre mutatunk példát.

Banki rendszer.. A ChandyLamport algoritmus – vagy minden más, ellentmondásmentes globális fényképet előállító algoritmus – felhasználható a fejezetben leírt banki rendszerben forgó pénzösszeg összeszámolásához. Ez a stratégia általánosítható az alapul szolgáló A algoritmus által kezelt más mennyiség meghatározására is.

Osztott nyomkövetés.. Egy ellentmondásmentes globális fényképező algoritmus segítséget nyújthat osztott algoritmusok nyomon követéséhez. Egy A osztott algoritmus tervezője leírhatja (és le is kell írnia) invariáns állításokkal A kulcstulajdonságait A globális állapotáról. Egy nyomkövető engedheti futni A-t időről-időre ellentmondásmentes globális fényképeket készítve, majd ellenőrizve, hogy az invariánsok teljesülnek-e minden fényképre. Mivel a fényképező algoritmus által visszaadott minden globális állapot A egy elérhető globális állapota, az invariánsoknak teljesülniük kell ezekre az állapotokra. A tervező ezt az ellenőrzést elvégezheti, mielőtt részletes induktív bizonyítást adna az invariánsokra. A 15.3. alfejezetben szereplő AszinkFeszFa algoritmusnak például két invariánsa van, a 15.3.1. és a 15.3.2. állítás, amelyeket ezen a módon lehet ellenőrizni.

Ellenőriznünk kell, hogy az invariánsok teljesülnek-e a visszaadott globális állapotra. Például a globális állapot információ eljuttatható egy folyamatnak, ami helyileg tudja ellenőrizni az invariánsokat. Használható ezen kívül egy osztott algoritmus, amely bemenetként kapja meg a fénykép algoritmus által visszaadott információt. A 15.3.1. állítás ellenőrizhető például egy osztott algoritmussal, megvizsgálva azt, hogy *szülő* mutatók egy halmaza egy adott i_0 csúcsban gyökerező irányított feszítőfát alkot-e; meghagyjuk gyakorlatnak egy ilyen algoritmus elkészítését (lásd 19-10. gyakorlat). A 15.3.2. állítás szintén ellenőrizhető egy osztott algoritmussal. Ebben az esetben az osztott algoritmus különösen egyszerű, mivel az invariáns állítás olyan tulajdonságok egy halmazával ábrázolható, amelyek mindegyike helyileg ellenőrizhető. (A helyi ellenőrzés eredményei eljuttathatók i_0-ba.)

Egy másik nyomkövetési stratégia az A egy központosított szimulációjának használata egyetlen processzoron. Ebben az esetben az A szimulált állapotának segítségével az invariánsok az A minden szimulált lépése után (vagy időről-időre) ellenőrizhetők. Ilyenkor nincs szükség globális fényképező algoritmusra; az eljárás hátránya, hogy a szimuláció tovább tart, mivel egyetlen processzoron hajtjuk végre.

Stabil tulajdonság felismerése. . Egy A aszinkron küld/fogad algoritmus P *stabil tulajdonságán* A globális állapotainak azt a tulajdonságát értjük, amely kielégíti a következő feltételt: ha P igaz az A minden elérhető s állapotára, akkor P igaz az s-ből elérhető összes állapotra is. Szemléletesen ez azt jelenti, hogy amennyiben P igazzá válik A egy végrehajtási sorozatában, akkor ettől kezdve P igaz is marad.

Egy egyszerű módszer annak eldöntésére, hogy a P stabil tulajdonság igaz-e A egy globális állapotára, ha egy globális fényképező algoritmus segítségével meghatározunk egy ellentmondásmentes globális állapotot, majd megvizsgáljuk, hogy P igaz-e vagy sem a visszaadott globális állapotra. Ez is eldönthető úgy, hogy egy P-t helyileg meghatározni képes folyamattal, illetve egy, a fénykép algoritmus által visszaadott bemenettel rendelkező osztott algoritmus segítségével összegyűjtjük az információt. Az ellentmondásmentes globális fényképező algoritmus helyességi feltételeiből az alábbiak következnek:

1. ha P igaz a fénykép állapotára, akkor P igaz az A fényképező algoritmus utolsó `jelent` műveletét követő utolsó állapotára is;
2. ha P hamis a fénykép állapotra, akkor P hamis az A fényképező algoritmus első `fényképez` művelete előtti utolsó állapotra is.

Az első állítás igaz, mivel az A utolsó `jelent` műveletet követő állapota elérhető a fénykép állapotából, míg a második állítás azért teljesül, mivel a fénykép állapot elérhető az A algoritmus első `fényképez` műveletét megelőző állapotából. Az algoritmus arról nem ad információt, hogy a P igaz-e az A-nak azon globális állapotaira, amelyek a fénykép algoritmus működése során lépnek fel.

Befejeződés jelzése.. Térjünk most vissza a befejeződés jelzésének problémájára. Tekintsünk ezúttal egy külső bemenetekkel nem rendelkező A küld/fogad algoritmust, ahol azonban a kezdőállapotok nem feltétlenül csendesek. Ha A bármikor elér egy csendes globális állapotot (amelyben a folyamatok nem hajthatnak végre lokálisan vezérelt műveleteket, és üzenetek sincsenek a csatornákon), egy befejeződést jelző algoritmusnak `elvégezte` kimenetet kell generálnia.

Mivel A-nak nincsenek külső bemenetei, a csendesség egy stabil tulajdonság. Így a befejeződés vizsgálható a stabil tulajdonság meglétét kereső általános stratégiának a segítségével is: készítsünk egy globális fényképet, majd döntsük el, hogy a visszaadott globális állapot csendes-e. Ebben az esetben a fénykép elkészítése után minden i folyamat el tudja dönteni, hogy az A_i rögzített állapota csendes-e, valamint hogy a bejövő csatornaállapotok üresek-e. Az eredmények (minden folyamatnál egy bit mondja meg, hogy az információ csendességet mutat-e vagy sem) ezután eljuttathatók egy kijelölt folyamathoz egy feszítőfán keresztül. Valójában minden folyamatnak egyetlen bitet kell elküldenie annak közlésére, hogy a részfában lévő összes folyamat csendességet jelentett-e.

Ha ez a stratégia A befejeződését jelzi, akkor valóban ez áll fenn. Továbbá ha többször ismételve készül a fénykép, akkor ez a stratégia garantáltan jelzi a befejeződést.

19.2.3. példa. Szélességi keresés és legrövidebb utak

A fent leírt stratégia felhasználható az AszinkSZK és az AszinkBellmanFord algoritmus befejeződésének jelzésére is. A fénykép az i_0 forráscsúccsal aktiválható. Ha a válasz pozitív, vagyis ha az alapul szolgáló algoritmus befejeződött, akkor az i_0 folyamat a kimeneti eredményeik közlésére felszólító üzenetet küldhet a többi folyamatnak. Másrészt, ha a válasz negatív, vagyis ha az alapul szolgáló algoritmus még nem fejeződött be, akkor az i_0 folyamatnak folytatnia kell a fényképek készítését, amíg az egyik pozitív választ nem ad.

19.2.4. példa. Vezetőválasztás

A 15.2. alfejezet aszinkron OptMaxTerjed vezetőválasztó algoritmusa bővíthető a ChandyLamport fényképező algoritmuson alapuló befejeződés-jelzéssel, elkészítve így a vezetőválasztás egy befejeződő algoritmusát tetszőleges összefüggő irányítatlan gráfban. Egy fénykép elkészítését kiválthatja például minden olyan folyamat, amelynek maximális ismert UID-ja módosul. A befejeződés jelzéséig több fényképre is szükség van. A fényképek elkülönítéséhez azok sorszámával indexelhetjük a különböző fényképek üzeneteit.

Érdekes összehasonlítani ennek a befejeződést jelző stratégiának a költségét a DijkstraScholten algoritmuséval, még ha ezek valamelyest eltérő típusú algoritmusokra működnek is. Emlékezzünk vissza, hogy a DijkstraScholten(A) algoritmus kommunikációs és időbonyolultsága nem a hálózat méretétől, hanem a küldött A-üzenetek teljes számától függ. Így, ha A csak egy rövid ideig a hálózat egy kis részén működik, akkor a DijkstraScholten(A) kisebb költségekkel jár. Másrészt a fénykép stratégia érinti a hálózat összes folyamatát, ezért a költsége nagyban függ a hálózat méretétől. Viszont abban az esetben, amikor a fényképet csak egyszer kell elkészíteni (és nem torlódnak fel az A-üzenetek), a fénykép költsége nem függ a küldött A-üzenetek számától. Vagyis ha A sokáig működik és sok üzenetet küld, a fénykép stratégia hatékonyabb, mint a DijkstraScholten(A).

Holtpont jelzése.. A *holtpont-jelző problémának* csak egy változatát tárgyaljuk; sok más változat is létezik. Tekintsünk egy A küld/fogad hálózati algoritmust, amelyben minden A_i folyamat rendelkezik azt jelző helyi állapotokkal, hogy a folyamat várakozik a szomszédságában lévő folyamatok egy részhalazára (például erőforrások felszabadítása kapcsán). Feltesszük, hogy amikor az A_i nem nulla számú szomszédjára várakozik, az egy csendes állapot; valójában a folyamat nem hajthat végre helyileg vezérelt lépéseket addig, amíg üzenetet nem kap az összes szomszédjától, amelyekre várakozik (például arról, hogy egy erőforrás felszabadult). Miután A_i üzenetet kap egy olyan folyamattól, amelyre várakozott, folytatja a várakozást a többi folyamatra. Feltételezzük, hogy A_i-nek nincsenek külső bemenetei.

A egy globális állapotában lévő *holtpont* egy egy két vagy több folyamatból álló kör, ahol minden folyamat a körben következőre vár, és nincs üzenetküldés

egy folyamattól sem a körben előtte állóhoz. A holtpont egy stabil tulajdonság, mivel amint egy ilyen kör kialakul, a körben szereplő folyamatok egyike sem hajthat végre több lokálisan vezérelt lépést. Így a holtpont vizsgálható a stabil tulajdonság meglétét kereső általános stratégiának a segítségével is: készítsünk egy globális fényképet, majd döntsük el, hogy van-e holtpont a visszaadott globális állapotban. Ez a döntés egy folyamat információinak begyűjtésével, majd egy soros kör-kereső algoritmus (például szélességi keresés) végrehajtásával tehető meg. Másik lehetőségként ez a döntés egy, a fényképek eredményét használó osztott kör-kereső algoritmus segítségével hozható meg.

Ez a stratégia garantáltan csak a tényleges holtpontokat jelzi. Továbbá ha a fényképek folyamatosan készülnek, minden előforduló holtpont jelzésére garantáltan sor kerül.

19.3.. Megjegyzések a fejezethez

A DijkstraScholten algoritmust Dijkstra és Scholten [92] készítette. A cikkükben közölt leírás meglehetősen eltérő az ittenitől; a bizonyításban egy „származtatott" algoritmust használnak. A DijkstraScholten egy általánosítását, amely tevékenységét több pontról is indíthatja, Francez és Shavit tanulmányozta. A befejeződés jelzésével foglalkozik még Francez munkája [126]. A ChandyLamport ellentmondásmentes globális fénykép algoritmus és annak stabil tulajdonságok felismerésére való használata Chandy és Lamport [68] nevéhez fűződik. Az algoritmus Lamport korábbi, logikai időre vonatkozó munkájából [176] származik. Fischer, Griffeth és Lynch [118] egy másik algoritmust tervezett ellentmondásmentes globális fényképekhez, ami tranzakció-alapú rendszerekhez készült (ahogyan ez a 19.8. példában szerepel).

Az alapul szolgáló A algoritmus módosíthatóságára vonatkozó korlátozások a Chandy és Misra [69] által tervezett Unity programnyelv *szuperpozíció* műveletének definíciójából származnak. Az osztott holtpont-jelzés problémájáról Isloor és Marsland [161], Menasce és Muntz [224], Gligor és Shattuck [138], Obermarck [238], Ho és Ramamoorthy [157], Chandy, Misra és Haas [70], valamint Bracha és Toueg [57] írt cikkeket. A holtpont-jelzés ezen fejezetben való megközelítése Bracha és Toueg [57] megközelítéséhez áll a legközelebb. Tay és Loke olyan modellt tervezett, ami segít bizonyos holtpont-jelző algoritmusok megértésében [274].

19.4.. Gyakorlatok

19-1. A DijkstraScholten algoritmusban a bevont folyamatok feszítőfája folyamatosan bővülhet és zsugorodhat, többször felhasználva ugyanazt a folyamatot. Ez a tulajdonság nem figyelhető meg az AszinkSzórásNyugtáz szemétgyűjtő változatánál – ott amint egy folyamat felszabadítja az állapotát, többet már nem kell részt vennie az algoritmusban. Mi az oka ennek az eltérő viselkedésnek?

19-2. Bizonyítsuk be a 19.1. lemmát.

19-3. Adjuk meg a lehető legjobb felső korlátot a kommunikációs és időbonyolultságra a 19.1. alfejezetben leírt befejező szélességi kereső algoritmusra, amelyet a DijkstraScholten AszinkSZK-re való alkalmazásával nyertünk.

19-4. Írjuk le, hogyan készíthetünk egy befejező legrövidebb utak algoritmust a DijkstraScholten és a 15.4. alfejezetben szereplő AszinkBellmanFord együttes felhasználásával. Adjuk meg a lehető legjobb felső korlátot a kommunikációs és időbonyolultságára.

19-5. Tekintsünk egy csendes globális állapotban induló A algoritmust (mint például egy terjesztő algoritmus), amelyet azonban egy környezettel együtt használunk, ami bemeneteket tud küldeni tetszőleges számú helyre (egy helyre egyet). Tervezzünk egy algoritmust annak felismerésére, hogy az A mikor ér el egy csendes globális állapotot. Akkor jelzünk befejeződést, ha a környezettől bemeneteket kapó összes folyamat `elvégezte` kimenetet ad.

19-6. Részletezzük a 19.3. tétel bizonyítását.

19-7. A 19.2.1. példa az A banki rendszer egy α végrehajtási sorozatát írja le egy globális állapottal együtt, ami egy ellentmondásmentes globális fénykép algoritmus helyes eredménye.

(a) Írjuk le a ChandyLamport(A) egy olyan végrehajtási sorozatát, ami visszaadja ezt a globális fényképet. Engedélyezhetjük `fényképez` bemenetek megjelenését bármikor a folyamatok tetszőleges részhalmazán.

(b) Általánosítsuk az (a) rész eredményét.

19-8. Az ebben a fejezetben bemutatott banki rendszer egy általánosításával foglalkozunk most. Tegyük fel, hogy meg van adva egy A küld/fogad rendszer, amelyben a folyamatok egy osztott adatbázist kezelnek úgy, hogy minden folyamat bizonyos adatelemeket kezel. Az A csak *tranzakció* tevékenységeket hajt végre. Itt egyszerűen úgy definiáljuk a tranzakciót, mint egy adatelemeken végrehajtott műveletek sorából álló szekvenciális programot; az atomiság nem feltétele a teljes tranzakciónak.

A problémát egy olyan új $B(A)$ rendszer megtervezése jelenti az A transzformálásával, amellyel meghatározható A-nak egy „ésszerű" *tranzakció-ellentmondásmentes fényképe*. Egy tranzakció-ellentmondásmentes fénykép egy állapotot tartalmaz minden A_i-hez, amelyek eredményét bizonyos tranzakciók befejeződése után kapjuk meg. Egy fényképet ésszerűnek tekintünk, ha tartalmazza a fénykép algoritmus megkezdése előtt befejeződő összes tranzakciót, és olyan további tranzakciók egy tetszőleges részhalmazát, amelyek a fénykép befejeződése előtt kezdődnek meg.

A B transzformációnak nem szabad szükségtelenül befolyásolnia az A működését; például nem állíthatja le az összes tranzakciót a fénykép készítése alatt.

19-9. Bizonyítsunk be egy felső korlátot a ChandyLamport(A) időbonyolultságára, a bonyolultságot A-nak a fényképezés alatti globális állapotban megjelenő üzeneteinek számával kifejezve.

19-10. Tekintsünk egy összefüggő nem irányított G gráfot egy kitüntetett i_0 csúccsal. Tervezzünk egy G gráf alapú, A aszinkron küld/fogad algoritmust annak

ellenőrzéséhez, hogy a *szülő* mutatóknak egy rögzített halmaza a G egy i_0-ban gyökerező részgráfjának irányított feszítőfáját adja. Pontosabban tegyük fel, hogy az A minden folyamata rendelkezik egy *szülő* mutatóval, amelynek értéke vagy egy szomszédos folyamat indexe vagy *null*. A kimenetet az i_0 folyamatnak kell szolgáltatnia. Adjunk előfeltétel/hatás kódot az algoritmusunkhoz, bizonyítsuk be a helyességét, és elemezzük a bonyolultságát.

19-11. Tekintsük a ChandyLamport fényképező algoritmussal a befejeződés jelzéséhez kibővített AszinkSZK algoritmust, ahogy ez a 19.2.3. példában szerepel.

(a) Adjunk meg egy explicit végrehajtási sorozatot (tetszőleges G gráfra), amelyben az i_0 folyamat először az AszinkSZK-t, majd egy fényképet aktivál, és amelyben a fénykép által visszaadott állapot nem csendes.

(b) Tegyük fel, hogy az i_0 mindannyiszor aktivál egy másik fényképet, amikor az előző negatív választ adott. Adható-e felső korlát azon fényképek számára, amelyeket aktiválni kell ahhoz, hogy végül az egyikük pozitív választ adjon?

19-12. A DijkstraScholten és a befejeződéshez használt fénykép megközelítések összehasonlítása csak olyan A algoritmusoknál indokolt, amelyekre mindkét féle befejeződési stratégia alkalmazható. Adjuk meg azt a legbővebb algoritmusosztályt, amelynek elemeire mindkét stratégia alkalmazható.

19-13. Tekintsük folyamatoknak egy olyan halmazát, ahol mindegyik folyamat várakozhat a szomszédaira. Vagyis minden folyamatnak van egy *várakozás* rögzített helyi értéke azon szomszédok leírására, amelyekre a folyamat várakozik.

(a) Tervezzünk (például előfeltétel/hatású kód megadásával) egy osztott körkereső algoritmust a folyamatoknak erre a halmazára. Az algoritmusnak el kell tudnia dönteni, hogy van-e olyan két vagy több folyamatból álló kör, ahol minden folyamat a körben következőre vár, és nincs üzenetküldés egy folyamattól sem a körben előtte állóhoz.

(b) Bizonyítsuk be az algoritmus helyességét, és vizsgáljuk meg a bonyolultságát.

(c) Mutassuk meg, hogyan használható ez az algoritmus holtpontok felismerésére egy alapul szolgáló A aszinkron algoritmusban, a 19.2.3. szakaszban adott feladatleírásnak megfelelően.

19-14. A holtpont feladat egy másik változatában a folyamatok a 19.2.3. szakaszban leírtak szerint várakoznak a szomszédaikra, de ezúttal a várakozó folyamatoknak elég, ha nem az összestől, hanem csak a szomszédok *egyikétől* kapnak információt. Adjunk meg egy ehhez a problémához igazodó holtpont meghatározást, és tervezzünk egy ellentmondásmentes globális fényképeken alapuló algoritmust ezen új típusú holtpontok felismeréséhez.

19-15. Mutassunk példát a küld/fogad hálózati algoritmusok figyelését végző ellentmondásmentes globális fényképek más alkalmazási területeire.

20. fejezet

Hálózati erőforrások hozzárendelése

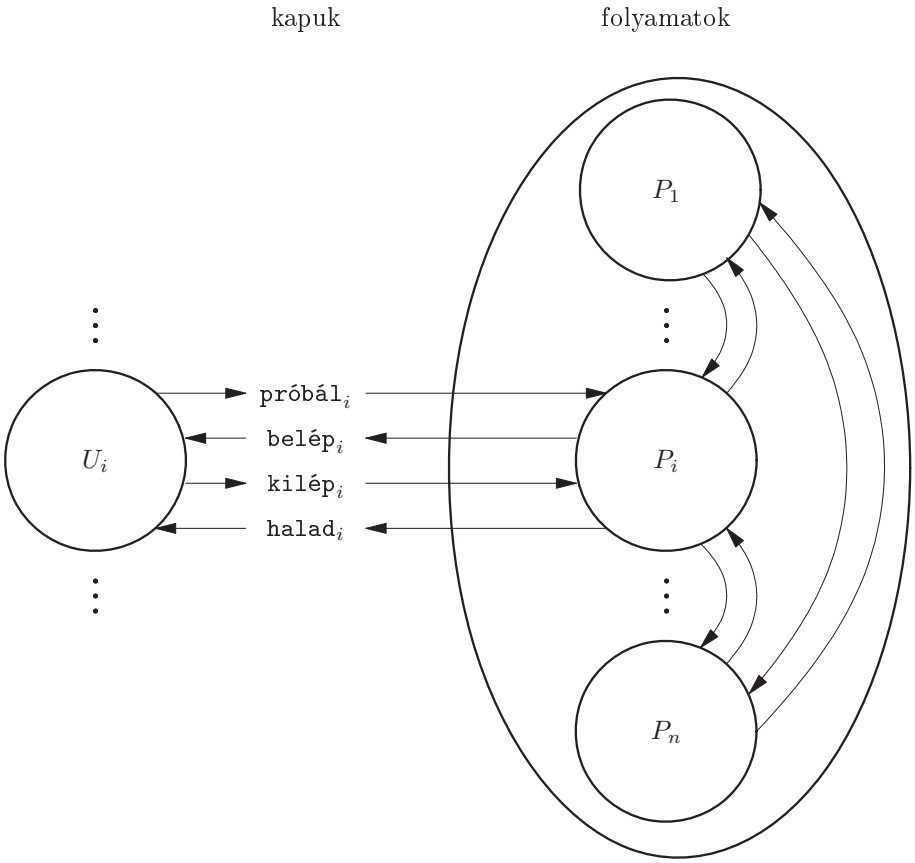
Miután a 16–19. fejezetekben bemutattuk az aszinkron hálózatok programozásának általános módszereit, most elemzésünket az aszinkron hálózatokban fellépő speciális problémákkal folytatjuk. Ebben a fejezetben visszatérünk a kölcsönös kizárási problémához és az erőforrás-hozzárendelés általánosabb problémájához, melyeket a 10. és 11. fejezetekben az aszinkron közös memóriával kapcsolatban már tárgyaltunk. Majd a 21. fejezetben az aszinkron hálózatokban fellépő megegyezési és más olyan problémákat tárgyalunk, melyekben egyes folyamatok hibásan is működhetnek. Az aszinkron számításokkal kapcsolatos utolsó, 22. fejezetben a megbízhatatlan csatornákon keresztül megvalósuló megbízható kommunikációval foglalkozunk.

20.1.. Kölcsönös kizárás

Kezdjük a kölcsönös kizárási feladat megoldásával.

20.1.1.. A feladat

A feladat meghatározása ugyanaz, mint a 10.2. alfejezetben volt. Feltételezzük, hogy adott n felhasználó (U_1, U_2, ..., U_n), akiket a 10.2. alfejezethez hasonlóan a jólformáltságot megőrző b/k-automataként definiálunk. Most azt tételezzük fel, hogy a feladat megoldására szolgáló A rendszer egy aszinkron hálózati rendszer, melyben minden U_i felhasználónak egy P_i folyamat felel meg. Feltesszük, hogy az U_i b/k automata és P_i közötti kommunikációra a $\texttt{próbál}_i$, $\texttt{belép}_i$, $\texttt{kilép}_i$ és $\texttt{halad}_i$ műveleteket alkalmazzuk. Küld/fogad hálózatok esetében a P_i folyamatok a megbízható $C_{i,j}$ FIFO csatornákon keresztül kommunikálnak, amint azt a 20.1. ábra mutatja. Üzenetszóró rendszereket, valamint küld/fogad csatornák és üzenetszóró csatornák együttesét tartalmazó rendszereket is vizsgálunk. (Egy ilyen együttes tekinthető úgy is, mint egy többletes üzenetszóró csatorna speciális esete – lásd a 14.3.2. szakaszt.)



20.1.. ábra. A komponensek közötti kölcsönhatás a kölcsönös kizárási problémára. Az ellipszis alakú rendszeren belüli nyilak a küld/fogad csatornákat szemléltetik.

A rendszertől elvárt alapvető helyességi feltételek ugyanazok, mint amiket a 10.2. alfejezetben definiáltunk. Azaz megköveteljük, hogy az A rendszer és a felhasználók együttese kielégítse a következő feltételeket.

Jólformáltság. Bármilyen végrehajtási sorozatban és bármilyen i-re az U_i és A közötti kölcsönhatást leíró végrehajtási részsorozat jólformált U_i-re.

Kölcsönös kizárás. Nincs olyan elérhető rendszerállapot (azaz A globális állapotának és az összes U_i állapotának olyan együttese), melyben egynél több felhasználó van a Be kritikus szakaszban.

Haladás. Egy *pártatlan végrehajtási sorozat* bármely pontjában igaz a következő két állítás.

1. (Haladás a próbálkozási szakaszban.) Ha legalább egy felhasználó van a *Pr* szakaszban és nincs felhasználó a *Be* szakaszban, akkor később valamelyik felhasználó belép a *Be* szakaszba.
2. (Haladás a kilépési szakaszban.) Ha legalább egy felhasználó van a *Ki* szakaszban, akkor később legalább egy felhasználó belép a *Ha* szakaszba.

Azt mondjuk, hogy egy A aszinkron hálózati rendszer *megoldja a kölcsönös kizárási problémát*, ha azt a felhasználók bármely halmazára megoldja.

Ebben a fejezetben elhagyjuk azt a korlátozást, amelyet a 10.2. alfejezetben tettünk arra vonatkozóan, hogy egy folyamat helyileg irányított műveleteket csak akkor hajthat végre, ha felhasználója a *Pr* vagy a *Ki* szakaszban van. Ez a korlátozás indokolt a közös memória esetében, mivel a közös változók úgy kezelik az információt, hogy az mindig minden folyamat számára hozáférhető. A hálózati környezetben azonban nincsenek közös változók, így a folyamatoknak kell kezelniük ezt az információt és szükség esetében nekik kell más folyamatokkal ezt az információt kicserélni.

Itt is alkalmazzuk ugyanazt a zárolásmentességi feltételt, mint a 10.4. alfejezetben.

Zárolásmentesség. Bármilyen pártatlan végrehajtás esetében teljesül a következő két feltétel.

1. (Zárolásmentesség a próbálkozási szakaszban.) Ha minden felhasználó mindig visszaadja a megszerzett erőforrást, akkor bármelyik felhasználó, amelyik eléri a *Pr* szakaszt, előbb-utóbb belép a *Be* szakaszba.
2. (Zárolásmentesség a kilépési szakaszban.) Bármelyik felhasználó, amelyik eléri a *Ki* szakaszt, előbb-utóbb belép a *Ha* szakaszba.

Ebben a fejezetben esetenként olyan igények kommunikációs és időbonyolultságát is elemezni fogjuk, amelyek „elszigetelve” lépnek fel. Azt mondjuk, hogy egy felhasználó egy igénye *elszigetelve* lép fel, ha az adott felhasználó `próbál` és `belép` művelete közötti időben minden más folyamat a saját *Ha* szakaszában marad.

A fejezet hátralévő részében néhány olyan algoritmust mutatunk be, amelyek aszinkron hálózatokban megoldják a kölcsönös kizárási problémát.

20.1.2.. A közös memória szimulálása

A tizedik fejezet jónéhány olyan algoritmust tartalmaz, amelyek közös memória esetében megoldják a kölcsönös kizárást. A 17. fejezet módszereivel ezeket az algoritmusokat átalakíthatjuk aszinkron hálózati modellben működő algoritmusokká. Például a 10.7. alfejezet VÁRÓTEREM algoritmusa elég hatékonyan megvalósítható aszinkron küld/fogad hálózatokban.

20.1.3.. A KörbejáróJel algoritmus

Az aszinkron küld/fogad hálózati modell legegyszerűbb kölcsönös kizárási algoritmusa akkor alkalmazható, amikor a hálózat egy egyirányú gyűrű.

A KörbejáróJel algoritmus (vázlatosan)

A gyűrűben egy *vezérlő jel* jár körbe, amely az erőforrás kezelését vezérli. Amikor a P_i folyamat megkapja a jelet, ellenőrzi, van-e az U_i felhasználótól származó, kielégítetlen igény. Ha ilyen igény nincs, P_i továbbítja a jelet P_{i+1}-nek. Másrészt, ha van ilyen kielégítetlen igény, P_i odaadja az erőforrást U_i-nek és addig magánál tartja a jelet, amíg U_i vissza nem adja az erőforrást. Amikor U_i visszadja az erőforrást, P_i továbbítja a jelet P_{i+1}-nek.

Az algoritmus formális kódja a következő.

20.1. algoritmus. KörbejáróJel

Lenyomat:

Bemeneti:
- $\texttt{próbál}_i$
- $\texttt{kilép}_i$
- $\texttt{fogad}(\text{„}jel\text{”})_{i-1,i}$

Kimeneti:
- $\texttt{belép}_i$
- $\texttt{halad}_i$
- $\texttt{küld}(\text{„}jel\text{”})_{i,i+1}$

Állapotok:

jel_státus $\in \{$*nem_itt,szabad,használatban,használt*$\}$,
kezdetben *szabad*, ha $i = 1$, egyébként *nem_itt*
szakasz $\in \{$*Ha, Pr, Be, Ki*$\}$, kezdetben *Ha*

Átmenetek:

$\texttt{próbál}_i$
Hatás:
szakasz := *Pr*

$\texttt{belép}_i$
Előfeltétel:
szakasz = *Pr*
jel_státus = *szabad*
Hatás:
szakasz := *Be*
jel_státus := *használatban*

$\texttt{kilép}_i$
Hatás:
szakasz := *Ki*

$\texttt{halad}_i$
Előfeltétel:
szakasz = *Ki*
Hatás:
szakasz := *Ha*
jel_státus := *használt*

$\texttt{fogad}(\text{„}jel\text{”})_{i-1,i}$
Hatás:
jel_státus := *szabad*

$\texttt{küld}(\text{„}jel\text{”})_{i,i+1}$
Előfeltétel:
jel_státus = *használt* **or**
(*jel_státus* = *szabad* **and**
szakasz = *Ha*)
Hatás:
jel_státus := *nem_itt*

Taszkok:

minden helyileg vezérelt művelet egy taszkot foglal magában

20.1. tétel. . *A* KörbejáróJel *algoritmus megoldja a kölcsönös kizárási problémát és garantálja a kizárásmentességet.*

Bizonyításvázlat. Közvetlenül. A kölcsönös kizárás biztosítva van, mivel csak egy jel van és csak az a felhasználó lehet *Be*-ben, akinél a jel van. A haladás biztosítva van, mivel a jel addig jár körbe, amíg igényt nem talál. A kizárásmentesség biztosítva van, mivel egyik folyamat sem elégít ki két igényt egymás után anélkül, hogy lehetővé tenné közben a jel körbejárását a gyűrűben, ezzel minden más folyamatnak esélyt adva. □

Bonyolultságelemzés.. Először a KörbejáróJel algoritmus kommunikációs bonyolultságát vizsgáljuk. Nem világos, hogy mit is mérjünk, mivel az üzenetek

nincsenek természetes módon az igényekhez kötve. Például akkor is vannak elküldött üzenetek, amikor nincs is igény. Azt például mondhatjuk, hogy egy $\texttt{próbál}_i$ üzenet és a neki megfelelő $\texttt{belép}_i$ üzenet között elküldött üzenet összes száma legfeljebb n. Amortizált elemzést végezhetünk a „nagy terhelés” esetére, amikor minden csúcsban van legalább egy aktív igény. (Formálisan, minden $\texttt{halad}_i$ műveletet azonnal követ egy $\texttt{próbál}_i$ művelet). Ebben az esetben igényenként csak konstans számú üzenetre van szükség.

Az időbonyolultsággal kapcsolatban szokás szerint feltesszük, hogy l egy felső időkorlát minden folyamat taszk végrehajtásái idejére és d egy egy felső korlát a legrégebbi üzenet késleltetésére nézve bármely csatornában. Azt is feltesszük, hogy c egy felső korlát arra az időre, amelyet egy felhasználó a Be szakaszban tölthet. Ekkor egy $\texttt{próbál}_i$ esemény és a neki megfelelő $\texttt{belép}_i$ esemény közötti idő legfeljebb $c(n-1)+dn+\mathcal{O}(ln)$. Megjegyezzük, hogy ez a korlát tartalmaz egy dn tagot, amely nagyon alacsony terhelésnél is fellép, például egyetlen elszigetelt igény esetében.

Látszólagos gyűrűk.. Ha a folyamatok látszólagos gyűrűt alkotnak, akkor a KörbejáróGyűrűk algoritmus tetszőleges, erősen összefüggő, irányított G gráfon alapuló küld/fogad hálózatban alkalmazható. A gyűrűben egymást követő folyamatoknak nem kell okvetlenül szomszédoknak lenniük G-ben – G erős összefüggősége miatt bármely folyamatpár elemei közötti kommunikáció szimulálható a hálózat irányított útjai mentén lezajló kommunikációk sorozatával. Az így kapott algoritmus hatékonysága jelentős mértékben függ a G gráftól és attól a sorrendtől, amely szerint a folyamatok a gyűrűben el vannak rendezve – lényeges dolog a szimuláció során használt utak összhosszának minimalizálása.

Hibatűrés.. A gyakorlatban a KörbejáróJel algoritmus bizonyos típusú hibákkal szemben rugalmassá tehető. Például ha egy folyamat teljesen leáll oly módon, amely más folyamatok számára felismerhető, akkor a többi folyamat átalakulhat egy új gyűrűvé. Egy másik példa, ha a jel elveszett, ismét csak olyan módon, amely más folyamatok számára felismerhető, akkor gyűrűk vezető folyamatának választására szolgáló protokoll – például a 15.1. alfejezetben tárgyaltak valamelyike – segítségével új jel állítható elő.

Az aszinkron modellben a szokásos folyamatmegállító hibák és üzenetvesztések nem ismerhetőek fel, mivel nincs mód arra, hogy a folyamatok a hibát megkülönböztessék attól, hogy a folyamatok vagy üzenetek egyszerűen késnek. Ezért a hibatűrés érdekében erősebb modellt kell feltételezni, amely tartalmaz az említett típusú hibákat jelző eseményeket. A gyakorlatban ezeket az eseményeket rendszerint időtúllépésekkel valósítják meg.

20.1.4.. Logikai időn alapuló algoritmus

A 18.3.3. szakaszban egy aszinkron hálózat, az üzenetszóró kölcsönös kizárási problémájának egy másik megoldását írtuk le. Az a megoldás a TöbbszörösÁllapotAutomata algoritmust alkalmazta az erőforrást igénylő folyamatok indexei sorát ábrázoló atomi objektum megvalósítására és logikai időt használt. Az algoritmust több részletben írtuk le.

Ebben a szakaszban hasonló algoritmust mutatunk be, de a fejezet többi algoritmusával való könnyebb összehasonlíthatóság érdekében a részeket együtt mutatjuk be. Az egyszerűség kedvéért a logikai műveleteket nem kezeljük olyan speciális módon, mint ahogyan a 18.3.3. szakaszban tettük. Az így kapott algoritmust LogikaiIdőKK algoritmusnak nevezzük.

LogikaiIdőKK algoritmus (vázlatosan)

Ez az algoritmus az események logikai idejét a LamportIdő stratégiával állítja elő, a helyi, nemnegatív egész *óra* értékek alapján. A logikai idő egy (c, i) pár, ahol $c \in \mathbb{N}$ és i a folyamat indexe; ezek a párok lexikografikusan vannak rendezve.

Ez az algoritmus üzenetszóró és küld/fogad kommunikációt is alkalmaz, ahol a küld/fogad kommunikáció bármely két különböző folyamat között meg van engedve. Külön *hívás_puffer* és sor helyett minden P_i folyamat egyetlen *történet* adatszerkezetet kezel. Minden j-re *történet*$(j)_i$ tartalmazza az összes üzenetet, amelyet P_i valaha is kapott P_j-től, egy nemnegatív egész c értékkel együtt, amely az adott üzenet `szór` vagy `küld` üzenetéhez tartozó *óra* érték. A `próbál` és a `kilép` igények üzenetszórásosak, szemben az előzőekkel. Üres üzenetek szórása helyett minden folyamat minden `próbál` üzenetre egy `nyugtáz` üzenettel válaszol.

P_i akkor hajthat végre egy `belép`$_i$ műveletet, amikor utolsó `próbál` igénye elérte P_i *történet*(i)-jét, feltéve, hogy minden más, kisebb logikai idejű üzenet, amelyet P_i kapott, már ki van elégítve, továbbá azt is feltéve, hogy P_i minden más folyamattól kapott nagyobb logikai idejű üzenetet. (A két utóbbi tulajdonság együtt biztosítja, hogy nincs kisebb logikai idejű aktuális üzenet, és később sem érkezhet ilyen üzenet.) Mihelyt P_i utolsó `kilép` üzenete elérte P_i *történet*(i)-jét, P_i végrehajthat egy `halad`$_i$ műveletet.

A formális kódot a következő automata-leírás tartalmazza. Ebben $\leq$ a logikai idő párok lexikografikus rendezését jelöli.

20.2. automata. LogikaiIdőKK$_i$

Lenyomat:

Bemeneti:
- `próbál`$_i$
- `kilép`$_i$
- `fogad`$(m)_{j,i}$, $m \in \{$„`próbál`”, „`kilép`”, „`nyugtáz`”$\} \times \mathbb{N}$, $1 \leq j \leq n$

Kimeneti:
- `belép`$_i$
- `halad`$_i$
- `küld`$(m)_{i,j}$, $m \in \{$„`nyugtáz`”$\} \times \mathbb{N}, j \neq i$
- `szór`$(m)_i$, $m \in \{$„`próbál`”, „`kilép`”$\} \times \mathbb{N}$

Állapotok:
szakasz $\in$ {*Ha, Pr, Be, Ki*}, kezdetben *Ha*
óra $\in \mathbb{N}$, kezdetben 0
szór_puffer, a { „`próbál`”, „`kilép`” }
elemeiből álló FIFO sor, kezdetben üres
$\forall j(1 \leq j \leq n)$ indexre:
    történet(j)$\subseteq$ { „`próbál`”, „`kilép`”, „`nyugtáz`” $\times\mathbb{N}$, kezdetben üres halmaz
$\forall j \neq i$ indexre:
    küld_puffer(j) a {„`nyugtáz`” $\times\mathbb{N}$} elemeiből álló FIFO sor, kezdetben üres

Átmenetek:

`próbál`$_i$
    Hatás:
        óra := *óra* + 1
        szakasz := *Pr*
        tegyük be „`próbál`”-t a
            szór_puffer sorba

`szór`$(m, c)_i$
    Előfeltétel:
        szór_puffer első eleme m
        c = *óra* + 1
    Hatás:
        óra := c
        távolítsuk *szór_puffer* első elemét

`fogad`$(m, c)_{j,i}$
    Hatás:
        óra := max(*óra*, c) + 1
        történet(j) := *történet*$(j) \cup \{(m, c)\}$
        if m = „*próba*” **and** $j \neq i$
            then tegyük be „*nyugtáz*”-t a
                küld_puffer$_j$ sorba

`küld`$(m, c)_{i,j}$
    Előfeltétel:
        m *küld_puffer*(j) első eleme
        c = *óra* + 1
    Hatás:
        óra := c
        távolítsuk el *küld_puffer* első elemét

`belép`$_i$
    Előfeltétel:
        szakasz = *Pr*
        („`próbál`”,c) $\in$ *történet*(i)
        $\nexists$(„`kilép`”, c) $\in$ *történet*(i), ahol $c' > c$
        $\forall j \neq i$:
            if („`próbál`”,c) $\in$ *történet*(j),
                $(c', j) < (c, i)$ **then**
                    $\exists$(„`kilép`”, c”) $\in$ *történet*(j),
                    ahol $c'' > c'$
                $\exists(m, c') \in$ *történet*),
                    ahol $(c, i) > (c', j)$
    Hatás:
        óra := *óra* + 1
        szakasz := *Be*

`kilép`$_i$
    Hatás:
        óra := *óra* + 1
        szakasz := *Ki*
        tegyük be „`kilép`”-et
            a *szór_puffer* sorba

`halad`$_i$
    Előfeltétel:
        szakasz = *Ki*
        („`kilép`”,c) $\in$ *történet*(i)
        $\nexists$(„*próbál*”,c') $\in$
            történet(i), melyre $c' > c$
    Hatás:
        óra := *óra* + 1
        szakasz = *Ha*

Taszkok:
belép$_i$
halad$_i$
szór$(m)_i$: $m \in$ {„`próbál`”„`kilép`”$\times\mathbb{N}$}
$\forall i \neq j$: {`küld`$(m)_{i,j}$: $m \in$ {„`nyugtáz`”$\times\mathbb{N}$}

- -

20.2. tétel. . *A* LogikaiIdőKK *algoritmus megoldja a kölcsönös kizárási problémát és garantálja a zárolásmentességet.*

Bizonyítás. Érvelésünk az algoritmus működésére támaszkodik. Ellentmondás segítségével bizonyítjuk, hogy az algoritmus megoldja a kölcsönös kizárási problémát. Tegyük fel, hogy adott elérhető rendszerállapotban a P_i és a P_j folyamatok egyidejűleg a *Be* szakaszban vannak. Az általánosság megszorítása nélkül feltehetjük, hogy P_i utolsó `próbál` műveletének t_i logikai ideje kisebb, mint P_j utolsó `próbál` üzenetének t_j logikai ideje. Ekkor a $\texttt{belép}_j$ művelet elvégzéséhez és a *Be* szakaszba való belépéshez P_j-nek a *történet*(i)-ben látnia kell egy P_i-től származó üzenetet, melynek logikai ideje t_j-nél nagyobb, és ezért t_i-nél is nagyobb. Ekkor a P_i-től P_j-hez vezető kommunikációs csatorna FIFO tulajdonsága miatt P_j-nek a $\texttt{belép}_j$ művelet elvégzése előtt látnia kellett P_i aktuális `próbál` üzenetét. $\texttt{belép}_j$ előfeltételéből következik, hogy P_j-nek látnia kellett a P_i-től származó `kilép` üzenetet. Ebből következik, hogy P_i már elhagyta a *Be* szakaszt, amikor P_j végrehajtotta a $\texttt{belép}_j$ műveletet, ami ellentmondás.

Most bizonyítjuk a zárolásmentességet, amiből következik a haladás. A *Pr* szakasz zárolásmentessége adódik abból a tényből, hogy az igények kiszolgálása `próbál` üzeneteik logikai idejének sorrendjében történik. Azt látjuk be, hogy az aktuális igények `próbál` üzenetei közül a legkisebb logikai idejűre végül `belép` válasz érkezik. Mivel csak véges sok olyan `próbál` üzenet van, melynek logikai ideje kisebb egy adott `próbál` üzenet logikai idejénél, indukcióval adódik, hogy végül minden folyamat beléphet a kritikus szakaszba.

Tegyük fel, hogy P_i a *Pr* szakaszban van és `próbál` üzenetének t_i logikai ideje a legkisebb az aktuális igényekhez tartozó logikai idők közül. Azt látjuk be, hogy a $\texttt{belép}_i$ előfeltételeinek végül teljesülniük kell és a $\texttt{belép}_i$ végrehajtásáig érvényben kell maradniuk. Az üzenetszóró csatorna pártatlansági tulajdonságaiból következik, hogy P_i végül megkapja a saját `próbál` üzenetét és elhelyezi *történet*$(i)_i$-ben. Továbbá, mivel a `próbál` üzenetek megkapják a megfelelő `nyugtáz` üzeneteket és az *óra* változókat a LamportIdő algoritmussal kezeljük, P_i végül üzenetet kap minden olyan folyamattól, melynek logikai ideje nagyobb, mint t_i. Végül, mivel P_i igénye a legkisebb logikai idejű aktuális igény, a nála kisebb logikai idejű igényekre már korábban végrehajtódott a megfelelő `kilép` művelet. Ekkor az üzenetszóró csatorna pártatlansági tulajdonságaiból következik, hogy P_i végül megkapja ezeket a `kilép` üzeneteket. Ilymódon végül $\texttt{belép}_i$ valamennyi előfeltételének teljesülnie kell.

A kilépési szakasz zárolásmentessége közvetlenül adódik. □

Bonyolultságelemzés.. A kommunikációs bonyolultsággal kapcsolatban megjegyezzük, hogy a LogikaiIdőKK algoritmus esetében, a KörbejáróJel algoritmustól eltérően, minden üzenet természetes módon hozzá van egy igényhez rendelve. Így az igényenkénti üzenetek számát vizsgáljuk. Minden igényhez tartozik egy `próbál` és egy `kilép` szórás, ami összesen $2n$ üzenet, és még $n-1$

`nyugtáz` üzenet, amelyek a `próbál` üzenetekre adott válaszok. Az összeg igényenként pontosan $3n-1$ üzenet.

Az időbonyolultsággal kapcsolatban először az U_i felhasználó elszigetelt igényét vizsgáljuk. Valójában „szigorúan elszigetelt" igényt tekintünk, amelynél azt is megköveteljük, hogy a $\texttt{próbál}_i$ esemény bekövetkeztekor ne legyenek a rendszerben korábbi igényekkel kapcsolatos üzenetek. Ebben az esetben a $\texttt{próbál}_i$ és $\texttt{belép}_i$ események közötti idő legfeljebb $2d + \mathcal{O}(l)$, ahol d a tetszőleges P_i folyamattól tetszőleges P_j folyamathoz való legrégebbi (szóró vagy ponttól pontig) üzenet továbbításának idejére vonatkozó felső korlát. Megemlítjük, hogy a KörbejáróJel algoritmus időbonyolultsága még elszigetelt igény esetében is tartalmaz egy dn tagot.

A $\texttt{próbál}_i$ és $\texttt{belép}_i$ események közötti idő felső korlátjának legrosszabb esetben való meghatározását meghagyjuk gyakorlatnak (lásd 20-7. gyakorlat).

20.1.5.. A LogikaiIdőKK algoritmus javításai

Most bemutatjuk a LogikaiIdőKK algoritmusnak egy egyszerű változatát, melyet a kommunikációs bonyolultság csökkentésére terveztek. A tervezői után RicartAgrawalaKK nevű algoritmus igényenként csak $2n-1$ üzenetet használ. A LogikaiIdőKK algoritmust azzal javítja meg, hogy az üzeneteket úgy nyugtázza, hogy nincs szükség a `kilép` üzenetekre. Az algoritmus szóró és küld/fogad kommunikációt is alkalmaz, ahol a küld/fogad kommunikáció bármely, különböző folyamatokból álló párnak meg van engedve.

RicartAgrawalaKK algoritmus (vázlatosan)

Az események logikai idejét a LogikaiIdőKK algoritmus szerint állítjuk elő. Csak a `próbál` üzenetet szórjuk és csak a a `jóváhagy` üzenetet küldjük a küld/fogad csatornán. Mindegyik üzenet tartalmazza a `szór` és `küld` eseményének *óra* értékét.

A $\texttt{próbál}_i$ bemenő üzenet után P_i a LogikaiIdőKK algoritmushoz hasonlóan szórja a `próbál` üzeneteket, és miután az összes többi folyamattól megkapta a következő `elfogad` üzenetet, beléphet a *Be* szakaszba. Az algoritmus érdekes része az a szabály, amely szerint egy P_i folyamat `elfogad` üzenetet küldhet egy másik folyamatnak. A szabály alapja egy elsőbbségi séma. A P_j-től kapott `próbál` üzenet hatására P_i a következőket teszi.

1. Ha P_i *Ki*-ben, *Ha*-ban vagy *Pr*-ben van az aktuális igényének megfelelő `próbál` üzenet szórása előtt, akkor `elfogad` üzenetet küld.
2. Ha P_i *Be*-ben van, akkor elhalasztja a válaszadást addig, amíg el nem éri a *Ki* szakaszt, és akkor egyszerre elküldi az összes késleltetett `elfogad` üzenetet.
3. Ha P_i *Pr*-ben van és aktuális igénye szét van szórva, akkor összehasonlítja saját igényének (a `szór` eseménynek) t_i logikai idejét P_j beérkező `próbál` üzenetének t_j logikai idejével. Ha $t_i > t_j$, akkor P_i kisebb prioritást ad a saját igényének és `elfogad` üzenetet küld. Egyébként P_i a saját igényének ad elsőbbséget és a válaszadást elhalasztja addig,

amíg be nem fejezi a következő *Be* szakaszát. Akkor azonnal elküldi az összes késleltetett `elfogad` üzenetet. Miután P_i kap egy `kilép`$_i$ üzenetet, bármikor végrehajthatja a `halad`$_i$ műveletet.

Más szavakkal, egymással ütköző igények esetében a RicartAgrawalaKK algoritmus a „korábbi" igény javára dönt, a logikai idők alapján.

20.3. tétel. . *A* RicartAgrawalaKK *algoritmus megoldja a kölcsönös kizárási problémát és garantálja a zárolásmentességet.*

Bizonyítás. Érvelésünk az algoritmus működésére hivatkozik. Először ellentmondás segítségével bebizonyítjuk a kölcsönös kizárást. Tegyük fel, hogy adott elérhető rendszerállapotban P_i és P_j egyidejűleg vannak *Be*-ben. Az általánosság megszorítása nélkül feltehetjük, hogy P_i utolsó `próbál` műveletének t_i logikai ideje kisebb, mint P_j utolsó `próbál` üzenetének t_j logikai ideje. A *Be* szakaszba való belépésük előtt szükségképpen P_i és P_j is küldött a másik folyamatnak `próbál` és `jóváhagy` üzenetet. Továbbá mindkét folyamatra fennáll, hogy a `próbál` üzenetnek a másik folyamattól való megkapása megelőzte a megfelelő `elfogad` üzenet elküldését. Ezek a megállapítások még a különböző események többféle sorrendjét megengedik. A 20.2. ábra mutat közülük néhányat.

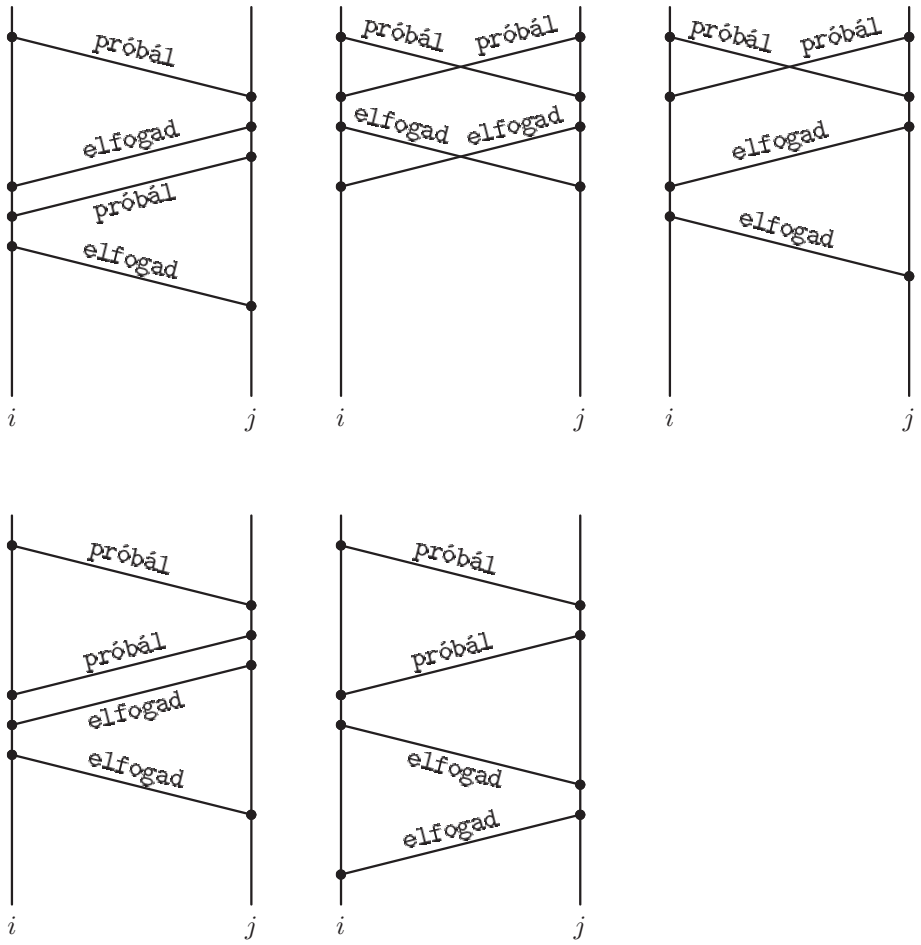
Most azt állítjuk, hogy a P_j utolsó `próbál` üzenetéhez tartozó `fogad` esemény azután következik be, hogy P_i szétszórta utolsó `próbál` üzenetét. Ha nem így van, akkor a logikai idő tulajdonságaiból következik, hogy ennek a `fogad`-nak a logikai ideje nagyobb, mint t_j, és hogy a P_i `szór` eseményének t_i logikai ideje nagyobb, mint ezé a `fogad` eseményé. Így $t_i > t_j$, ami ellentmondás.

Ezért abban az időpontban, amikor P_i megkapja P_j `próbál` üzenetét, P_i vagy *Pr*-ben van, vagy *Be*-ben. P_i szabályai mindkét esetben azt írják elő, hogy addig halassza el az `elfogad` üzenet elküldését, amíg be nem fejezi saját *Be* szakaszát. Ezért P_j addig nem léphet be *Be*-be, amíg P_i el nem hagyja, ami ellentmondás.

Most a haladást bizonyítjuk be, ugyancsak ellentmondás segítségével. A haladás a *Ki* szakaszra közvetlenül adódik. Tegyük fel, hogy egy α pártatlan végrehatási sorozatban elérünk egy olyan pontot, ahol van felhasználó a *Pr*-ben és nincs felhasználó *Be*-ben és amely időpont után felhasználó többé nem lép be *Be*-be. Ekkor (a 10.4. lemma bizonyításához hasonló módon érvelve) α megfelelő α_1 utótagjában minden felhasználó *Ha*-ban vagy *Pr*-ban van és nincs további szakaszváltás. Ekkor van α_1-nek olyan α_2 utótagja, amelyben minden *Pr*-ben lévő folyamat rendelt már logikai időt utolsó igényéhez és amelyben még átmenő igény sincs. Az α_2-ben *Pr*-ben lévő folyamatok között legyen P_i az a folyamat, melynek utolsó igényéhez a legkisebb logikai idő tartozik, mondjuk t_i.

Mivel P_i végleg *Pr*-ben marad, szükségképpen valamelyik másik P_j folyamat sohasem válaszol P_i utolsó `próbál` üzenetére. Két oka lehet, miért P_j nem küld `elfogad` üzenetet közvetlenül azután, hogy megkapta P_i-től a `próbál`-t.

1. P_j *Be*-ben van, amikor megkapja `próbál`-t. Ebben az esetben, mivel α_2 során nincsenek folyamatok *Be*-ben, P_j-nek α_2 megkezdése előtt be kell fejeznie a kritikus szakaszát és ezután el kell küldenie a késleltetett `jóváhagy` üzenetet P_i-nek.
2. P_j *Be*-ben van, amikor megkapja a `próbál`-t, és az igényéhez tartozó logikai időre $t_j < t_i$. Ebben az esetben, mivel P_i igényének van a legkisebb logikai



20.2.. ábra. Az események néhány lehetséges sorrendje a RicartAgrawalaKK algoritmus esetében.

ideje azok közül a folyamatok közül, amelyek α_2-ben Pr-ben vannak, P_j szükségképpen eléri és befejezi saját kritikus szakaszát azután, hogy megkapja a `próbál`-t P_i-től és azelőtt, hogy α_2-t elkezdené. De mégegyszer, ez azt jelenti, hogy P_j-nek el kell küldenie a késleltetett `elfogad` üzenetet P_i-nek.

P_i mindkét esetben megkap minden szükséges `elfogad` üzenetet és eljut Be-be, ami ellentmondás. A zárolásmentesség bizonyítását meghagyjuk gyakorlatnak (lásd 20-11. gyakorlat). □

Bonyolultságelemzés.. Könnyen belátható, hogy az algoritmus igényenként pontosan $2n - 1$ üzenetet küld. Az időbonyolultság elemzését meghagyjuk gya-

korlatnak (lásd 20-13. gyakorlat).

Egy másik optimalizálás.. A RicartAgrawalaKK algoritmus tovább javítható, ha az `elfogad` üzeneteknek más értelmezést adunk. Mármost ha most P_i `elfogad` üzenetet küld egy másik folyamatnak, P_j-nek, akkor azzal nemcsak jóváhagyja P_j aktuális igényét, hanem egyúttal engedélyt ad arra is, hogy P_j tetszőleges sokszor újra belépjen Be-be – mindaddig, amíg P_j egy `elfogad` üzenetet nem küld P_i-nek válaszul P_i egy `próbál` üzenetére. A `próbál` üzenet megválaszolási szabályai ugyanazok, mint a RicartAgrawalaKK algoritmus esetében.

Az algoritmusnak ez a változata különösen jól használható olyan esetekben, amikor egyetlen felhasználó ismételten igényel erőforrást, és közben más felhasználóktól nem érkezik igény. Ebben az esetben az igénylő felhasználó ismételten beléphet Be szakaszába anélkül, hogy az első igényére vonatkozó üzenet után bármilyen további üzenetre szükség lenne.

20.2.. Általános erőforrás-hozzárendelés

Most az aszinkron hálózatok általánosabb erőforrás-hozzárendelési problémáit vizsgáljuk.

20.2.1.. A feladat

A feladat definíciója nagyjából ugyanaz, mint a 11.1. alfejezetben volt. Felhasználjuk az ott definiált *explicit erőforrás-leírást* és *kizárás-leírást*. Ugyanolyan típusú felhasználói automatákat tételezünk fel, mint a 20.1. alfejezetben.

A rendszertől megkövetelt alapvető helyességi feltételek ugyanazok, mint a 11.1. alfejezetben. Nevezetesen megkívánjuk, hogy egy adott $\mathcal{E}$ kizárás-leírás mellett a rendszer és a felhasználók együttese rendelkezzen a következő tulajdonságokkal.

Jólformáltság. Az U_i és A közötti kölcsönhatást leíró részsorozat minden i-re és minden végrehatási sorozatban jólformált legyen $U_i$re.

Kizárás. Nincs olyan elérhető rendszerállapot, melyben a Be szakaszukban lévő felhasználó halmaza $\mathcal{E}$-beli.

Haladás. Egy *pártatlan végrehatási sorozat* bármely pontjában igaz a következő két állítás.

1. (Haladás a próbálkozási szakaszban.) Ha legalább egy felhasználó van a Pr szakaszban és nincs felhasználó a Be szakaszban, akkor később valamelyik felhasználó belép a Be szakaszba.
2. (Haladás a kilépési szakaszban.) Ha legalább egy felhasználó van a Ki szakaszban, akkor később legalább egy felhasználó belép a Ha szakaszba.

Azt mondjuk, hogy egy aszinkron hálózati rendszer megoldja az általános erőforrás-hozzárendelési problémát, ha azt a felhasználók minden csoportjára megoldja. Az explicit erőforrás-leírások tekintetében figyelembe vesszük a „független haladás” fogalmát is.

Független haladás. Egy *pártatlan végrehatási sorozat* bármely pontjában igaz a következő két állítás.

1. (Független haladás a próbálkozási szakaszban.) Ha U_i a Pr szakaszban van és minden vele versenyző felhasználó a Ha szakaszban van, akkor később vagy U_i belép a Be szakaszba, vagy valamelyik vele versenyző felhasználó belép a Pr szakaszba.
2. (Független haladás a kilépési szakaszban.) Ha egy U_i felhasználó a Ki szakaszban és minden, vele versenyző felhasználó a Ha szakaszban van, akkor egy későbbi időpontban vagy U_i lép be a Ha szakaszba, vagy pedigvalamelyik vele versenyző felhasználó lép be a Pr szakaszba.

Ugyanazt a zárolásmentességi feltételt vesszük figyelembe, mint a kölcsönös kizárásnál. Ugyanúgy, mint a kölcsönös kizárásnál tettük, elhagyjuk azt a korlátozást, hogy egy folyamat helyileg vezérelt műveleteket csak akkor hajthat végre, amikor felhasználója a próbálkozási vagy a kilépési szakaszban van.

Adott $\mathcal{R}$ erőforrás leírás esetében azt mondjuk, hogy egy felhasználó adott igénye elszigetelt, ha a `próbál` esemény és a `belép` esemény közötti időben a versenyző igényekkel rendelkező felhasználók haladási szakaszaikban vannak.

Ivó filozófusok.. Az általános erőforrás-hozzárendelési feladatnak az a változata, amelyet a 20.2.5. szakaszban fogunk vizsgálni, lehetővé teszi az U_i felhasználó számára, hogy különböző időpontokban különböző erőforrásokat igényeljen. A problémának ez a változata adott $\mathcal{R}$ erőforrás-leíráson alapul, és feltesszük, hogy a $\texttt{próbál}_i$ művelethez minden i-re hozzátartozik egy tetszőleges R_i halmaz, az U_i felhasználó által igényelt erőforrások halmaza. A *kizárás* feltételét újraértelmezzük, hogy inkább az utoljára igényelt *aktuális* erőforrásokra vonatkozzon, mint a $\mathcal{R}$ segítségével leírt potenciális erőforrásigényekre. Azaz megköveteljük, hogy ne legyen olyan elérhető rendszerállapot, melyben van két olyan felhasználó a Be szakaszában, melyek aktuális igényhalmazainak van közös eleme. A *haladás* feltétele és a *zárolásmentesség* feltétele ugyanaz, mint korábban. A *független haladás* feltételét és az *elszigetelt* igény definícióját újraértelmezzük, hogy az *aktuális* igényekre vonatkozzanak.

Az erőforrás-hozzárendelési problémának ez a változatát úgy is nevezik, hogy *ivó filozófusok problémája*, az erőforrásokat pedig *palackoknak* is szokták nevezni.

20.2.2.. Színezési algoritmus

A 11.3.3. szakaszban szereplő SZÍNEZ algoritmus módosítható úgy, hogy megoldja a G összefüggő, irányítatlan gráffal adott aszinkron küld/fogad hálózat általánosított erőforrás-hozzárendelési feladatát – adott $\mathcal{R}$ erőforrás-leírásra. Ennek egyik útja az, hogy a közös memóriájú algoritmusok 17. fejezetben leírt szimulációinak

egyikét alkalmazzuk. A speciálisan az adott célre irányuló szimuláció azonban hatékonyabb.

SZÍNEZ algoritmus (vázlatosan)

A SZÍNEZÉS algoritmus közös memóriát kezelő folyamatait szimuláló folyamatokhoz hozzáveszünk még egy folyamatot az összes erőforrás kezelésére. A hálózati algoritmus minden P_i folyamata pontosan a közös memóriájú algoritmus egy folyamatát és az erőforrás-folyamatok bizonyos részhalmazát szimulálja. Amikor az U_i felhasználó `próbál`$_i$ műveletet végez, a P_i folyamat egyidejűleg összegyűjti a szükséges erőforrásokat, szín szerint növekvő sorrendben, mint korábban, de ezúttal a megfelelő erőforrás-folyamatoknak is küld üzenetet. P_i minden üzenet elküldése után vár a válaszra. Minden erőforrás-folyamat FIFO sorban tárolja az igénylő felhasználókat, és minden új igényt hozzáad a sor végéhez. Amikor az i index eléri az erőforrás-folyamat sorának elejét, az erőforrás-folyamat viszontüzenetet küld P_i-nek, amely azután áttér a következő erőforrás igénylésére. Amikor P_i az összes szükséges erőforrást megkapta, akkor végrehajtja a `belép`$_i$ műveletet. Ha P_i `kilép`$_i$ műveletet hajt végre, akkor minden érintett erőforrás-folyamatnak üzenetet küld, hogy az i indexet eltávolíthatják a sorukból. Ha P_i mindezeket az üzeneteket elküldte anélkül, hogy válaszra várna, akkor végrehajtja a `halad` műveletet.

Az algoritmus megkívánja, hogy minden P_i folyamat képes legyen a kommunikációra az összes olyan P_j folyamattal, amely kezeli az adott $\mathcal{R}$ erőforrás-leírás által i-hez rendelt erőforrásokat. Ha a megfelelő csúcsok a G gráfban össze vannak kötve, akkor ez a kommunikáció megvalósulhat közvetlenül, egyébként pedig a kommunikáció szimulálható a G gráf éleiből képezett úttal.

A SZÍNEZÉS algoritmus ezen változatának hálózatokra vonatkozó elemzése hasonló ahhoz, ahogyan a 11.3.3. szakaszban a közös memóriára vonatkozó elemzést végeztük. Ebben az esetben az időkorlát függ a folyamatok által végzett lépések végrehajtási idejének felső korlátjától, az üzenettovábbítás idejétől, a *Be* szakaszban töltött időtől, továbbá az erőforrásgráf színezésére felhasznált színek számától és az egy erőforrást igénylő felhasználók maximális számától.

20.2.3.. Logikai időn alapuló algoritmusok

A RICARTAGRAWALAKK algoritmus általánosítható úgy, hogy tetszőleges $\mathcal{R}$ erőforrás-leírásra megoldja az erőforrás-hozzárendelési problémát. Most tegyük fel, hogy a hálózatban a többletes üzenetszórás és a küld/fogad kommunikáció kombinációja valósul meg. (A 14.3.2. szakasz szerint ez a kombináció úgy is tekinthető, mint a többletes üzenetszórás speciális esete.) Adott folyamatnak minden olyan folyamatcsoport felé meg kell engedni a többletes üzenetszórást, amellyel van közös erőforrása, és a küld/fogad kommunikációt minden olyan folyamatpár között lehetővé kell tenni, amelyeknek van közös erőforrása.

RICARTAGRAWALAEH algoritmus (vázlatosan)

A folyamatok a logikai időt a LAMPORTIDŐ algoritmus szerint számítják ki.

Ha a P_i folyamat kap egy `próbál`$_i$ üzenetet, akkor megfelelő *óra* értéket is tartalmazó `próbál` üzenetet szór valamennyi olyan folyamat felé, amellyel van közös erőforrása. A P_i folyamat akkor léphet be a Be szakaszba, ha ezután mindegyik ilyen folyamattól megkapja az üzenetére küldött `elfogad` választ. A folyamatok az `elfogad` üzenetek küldésénél ugyanazt a szabályt alkalmazzák, mint a RicartAgrawalaKK algoritmusban, az elsőbbség eldöntéséhez a logikai időket használva.

A P_i folyamat egy `kilép`$_i$ üzenet beérkezése után bármikor végrehajthat egy `halad`$_i$ műveletet.

20.4. tétel. . *A* RicartAgrawalaEH *algoritmus adott erőforrás-leírásra megoldja az általános erőforrás-hozzárendelési problémát és garantálja a zárolásmentességet és a független haladást.*

Amint azt a RicartAgrawalaKK algoritmusnál is tettük, a RicartAgrawalaEH algoritmust is módosíthatjuk úgy, hogy a `jóváhagy` üzenetek kiterjesszék (P_i számára) a jogosultságot, amíg az explicit módon visszavonásra nem kerül.

20.2.4.. Algoritmus körmentes irányított gráfra

A RicartAgrawalaKK algoritmusban a logikai időket inkább arra használjuk, hogy prioritásokat rendeljünk a versenyző igényekhez, mintsem a holtversenyek eldöntésére. Alternatív stratégiák alkalmazhatóak a holtversenyek eldöntésére, például az összes folyamatot tartalmazó körmentes irányított gráf alkalmazása.

Az egyszerűség kedvéért a következő két megszorításnak eleget tevő $\mathcal{R}$ explicit erőforrás-leírást tekintjük.

1. Minden erőforrás pontosan két felhasználó erőforrás-halmazában fordul elő.
2. Minden felhasználópár legfeljebb egy erőforrást oszt meg.

Ezeknek a megszorításoknak az elhagyását meghagyjuk gyakorlatnak (lásd 20-18. gyakorlat).

Egy G összefüggő, irányítatlan gráffal megadható küld/fogad hálózatot tételezünk fel. Feltesszük, hogy az egy erőforrást megosztó folyamatok G egy élével közvetlenül össze vannak kötve. A tárgyalásmód egyszerűbbé tétele érdekében azt is feltesszük, hogy G összes éle olyan folyamatokat köt össze, amelyek közösen akarnak erőforrást használni.

KörmentesIrGráfEH algoritmus (vázlatosan)

Az algoritmus alapja az, hogy a G éleinek irányítását úgy végezzük, hogy az irányított éleket tartalmazó H irányított gráf minden időpontban körmentes legyen. Minden él irányítását helyi *irányítás* változókban tartjuk nyilván az él két végpontjában lévő folyamatokban és olyan `vált` üzenetekkel változtatjuk, amelyeket az irányított él kiinduló csúcsánál lévő folyamat küld a végpontnál lévő csúcsban lévő folyamatnak. Fel kell tennünk, hogy a kiinduló élirányítás körmentes gráfot határoz meg.

Ha a P_i folyamat a Pr szakaszban van és minden illeszkedő éle befelé van irányítva, akkor P_i végrehajthatja a `belép`$_i$ műveletet. Ha a P_i folyamat

a *Ki* szakaszban van, akkor végrehajthatja a `halad`$_i$ műveletet, minden *irányítás* változóját kifelé irányíthatja és `vált` üzenetet küldhet minden illeszkedő élen, és mindezt egyetlen lépésben. A `vált` üzenetek egyidejűleg helyi *küld_puffer*-ekbe kerülnek (minden szomszéddal kapcsolatban külön puffer van). Tehát, ha a P_i folyamat a *Ha* szakaszban van és minden él befelé van irányítva, akkor P_i az *irányítás* változókat kifelé irányíthatja és minden élen `vált` üzenetet küldhet, ezúttal is egyetlen lépésben.

20.5. tétel. . *Az* AciklikusIrGráfEH *algoritmus a megadott két megszorítás mellett megoldja az erőforrás-hozzárendelési problémát és biztosítja a zárolásmentességet.*

Bizonyításvázlat. Azzal kezdjük, hogy a korábbinál gondosabb meghatározást adunk tetszőleges elérhető állapot minden élének irányítására. Nevezetesen azt mondjuk, hogy egy él P_i-ből P_j-be van irányítva, ha P_i *irányítás* változója az adott élre „kifelé" értéket mutat és vagy P_i vagy P_j *irányítás* változója „befelé" értéket mutat pedig van egy `vált` üzenet a P_i-ből P_j-be vezető úton (a *küld_puffer*$(j)_i$-ben vagy a P_i-ből P_j-be vezető csatornában). Invariáns bizonyítás adható arra, hogy ez a szabály minden elérhető állapotban és minden élre egyértelmű irányítást eredményez.

Ezután bebizonyítjuk azt az invariáns tulajdonságot, hogy amikor egy P_i folyamat a *Be* szakaszban van, akkor minden illeszkedő él befelé mutat és ezeken az éleken nincsenek egyik irányban sem átmenő `vált` üzenetek. Ebből következik a kizárási tulajdonság.

Bebizonyítjuk azt a kulcstulajdonságot, hogy a H gráf körmentes. Feltettük, hogy ez kezdetben igaz. Csak azok a lépések tehetik hamissá ezt az állítást, melynek során bizonyos él irányítása megváltozik. De minden olyan lépés, amely megváltoztatja egy adott P_i csúcsra illeszkedő él irányítását, egyúttal az adott csúcsra illeszkedő *valamennyi* él irányítását is megváltoztatja úgy, hogy a lépés után ezek az élek kifelé irányulnak. Mivel az adott lépés után nem lesz P_i felé irányuló él, nem lesz olyan kör sem, amely az újonnan irányított éleket tartalmazza. Ebből következik, hogy a lépés során nem jöhet létre kör.

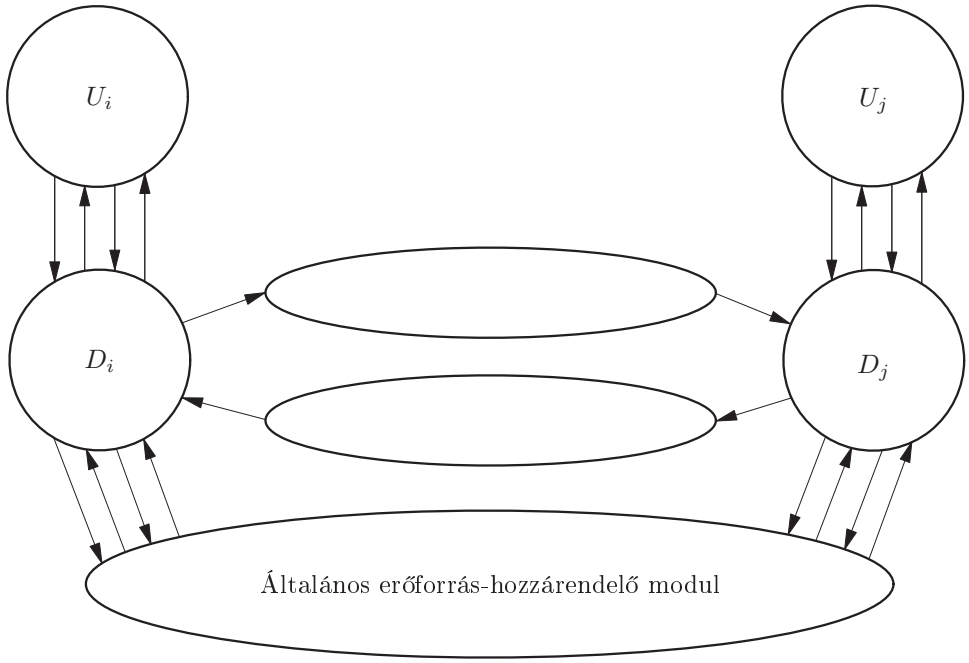
Most megmutatjuk a zárolásmentességet, amiből következik a haladás. Csak *Pr* zárolásmentességét vizsgáljuk; a *Ki* szakaszra vonatkozó feltétel könnyen belátható. Mivel a gráf a végrehajtás bármely pontjában körmentes, a H gráf P_i csúcsának *magasságát* definiálhatjuk olymódon, mint a P_i csúcsban kezdődő leghosszabb irányított út hossza. Először megjegyezzük, hogy egy csúcs magassága sohasem nő addig, amíg a csúcs el nem éri a nulla magasságot (és esélyt ad a csúcsnál lévő folyamatnak arra, hogy belépjen a *Be* szakaszba). Ezután megmutatjuk, hogy minden nulla magasságú csúcs végül minden hozzá illeszkedő élet kifelé irányít. Ezeket a tényeket felhasználva megmutatjuk, hogy minden $h > 0$ magasságú csúcs végül elér egy kisebb, h' magasságot, amiből következik, hogy bármely csúcs, amelyre $h' > 0$, végül eléri a 0 magasságot. Ez esélyt ad az adott csúcsnál lévő folyamatnak a kritikus szakaszba lépésre. □

Azonos folyamatok.. A KörmentesIrGráfEH algoritmus érdekes tulajdonsága, hogy a folyamatok „majdnem" azonosak: az algoritmus sem felhasználói

azonosítókat, sem más megkülönböztető információt nem használ, csak az élek kezdeti irányítását. Ha a problémát tetszőleges gráfra meg akarjuk oldani, akkor a 11.2. tételben használt érveléshez hasonló gondolatmenettel belátható, hogy valahogyan meg kell törnünk a szimmetriát. Itt a szimmetriát azzal törjük meg, hogy feltesszük, a H gráf kezdetben körmentes.

20.2.5.. Ivó filozófusok*

Most bemutatjuk az adott R erőforrás-leírásra vonatkozó ivó filozófusok problémának egy G összefüggő, irányítatlan gráfra támaszkodó, megbízható FIFO csatornákkal rendelkező küld/fogad hálózatban való megoldását. Ez a megoldás moduláris – tetszőleges zárolásmentes algoritmust használhat, amely adott $\mathcal{R}$-re megoldja az általános erőforrás-hozzárendelési problémát. A MODULÁRISIVÓFIL algoritmus felépítését a 20.3. ábra mutatja. Az U_i felhasználó és a megfelelő D_i közötti kommunikáció $\texttt{próbál}(B)_i$, $\texttt{belép}_i$, $\texttt{kilép}_i$ és $\texttt{halad}_i$ műveletekkel történik. Itt $B \subseteq B_i$, ahol B_i az $\mathcal{R}$ szerint P_i számára szükséges palackok (erőforrások) halmaza.



20.3.. ábra. A MODULÁRISIVÓFIL algoritmus szerkezete.

A D_i-k és az általános erőforrás-hozzárendelő modul közötti kommunikáció $\texttt{belső\_próbál}_i$, $\texttt{belső\_belép}_i$, $\texttt{belső\_kilép}_i$ és $\texttt{belső\_halad}_i$ műveleteket használ. A műveleteknek a félreértések elkerülése érdekében adtunk új neveket.

Az ivó filozófusok problémájának az adott modellben való teljes megoldásának tartalmaznia kell a 20.3. ábrán szereplő általános erőforrás-hozzárendelő modulnak egy olyan A küld/fogad hálózati algoritmussal történő megvalósítását, amely a G gráfon alapul. A teljes megoldásban minden P_i folyamat D_i és A megfelelő folyamatának összetevése. A teljes megoldásban minden $C_{i,j}$ csatornának meg kell valósítania mind a 20.3. ábrán D_i-től D_j-hez vezető csatornát, mind pedig A megfelelő csatornáját.

Az egyszerűség kedvéért éljünk ismét azzal az $\mathcal{R}$-re vonatkozó megszorítással, amellyel a KörmentesIrGráfEH esetében: minden palack pontosan két felhasználó erőforrás-halmazában szerepel. Azt is feltesszük, hogy minden olyan folyamatpár, amelyeknek van közös palackja, éllel közvetlenül össze van kötve G-ben.

ModulárisIvóFil algoritmus (vázlatosan)

Amikor D_i megkapja a `próbál`$(B)_i$ üzenetet, `igényel` üzeneteket küld azoknak a palackoknak, amelyekre szüksége van, de pillanatnyilag hiányoznak neki. Az `igényel` üzenet D_j címzettje kielégíti az igényt, ha U_j Ki-ben van vagy Ha-ban. Ha U_j Pr-ben van vagy Be-ben, akkor D_j elhalasztja az igény kielégítést addig, amíg U_j befejezi Be szakaszát.

Annak megelőzésére, hogy két folyamat elhalassza egymás igényének kielégítését és így blokkolja a *haladást*, az *általános erőforrás-hozzárendelő modult* alkalmazzuk, hogy a folyamatok között elsőbbséget hozzzon létre. Így amint egy D_i folyamat a Pr szakaszába belépve megteheti, egy `belső_próbál`$_i$ műveletet elvégezve megpróbál elsőbbséghez jutni. Amikor D_i megkapja a `belső_belép`$_i$ bemenetet, miközben még mindig a próbálkozási szakaszban van – azaz amikor *belső kritikus szakaszába* lép – akkor `követel` üzeneteket küld a még szükséges, de hiányzó palackokhoz. A `követel` üzenetet megkapó D_j mindig kielégíti a követelést, ha nála van a palack – kivéve azt az esetet, amikor U_j éppen a Be szakaszban van és használja a palackot; ebben az esetben D_j elhalasztja a követelést és akkor elégíti ki, amikor U_j befejezi Be szakaszát.

Belátható, hogy ha már D_i a belső kritikus szakaszában van, akkor végül minden szükséges palackot megkap. Ha D_i a próbálkozási szakaszában van és minden szükséges palackot megszerzett, akkor beléphet a Be szakaszába. Ha már D_i a kritikus Be szakaszban van, adhat `belső_kilép`$_i$ kimenetet, mivel többé nincs szüksége a belső kritikus szakaszához kapcsolódó elsőbbségre.

A kódot a következő automata-leírás tartalmazza.

20.3. automata. $\mathbf{D}_i$

Lenyomat:

Bemeneti:
- $\texttt{próbál}(B)_i,\ B \subseteq B_i$
- $\texttt{kilép}_i$
- $\texttt{belső\_kilép}_i$
- $\texttt{belső\_halad}_i$
- $\texttt{fogad}(m)_(j,i),\ m \in \{$„igény”, „palack”, „követelés”$\} \times (B_i \cap B_j),\ j \in$ *szomszédok*

Kimeneti:
- $\texttt{belép}_i$
- $\texttt{halad}_i$
- $\texttt{belső\_próbál}_i$
- $\texttt{belső\_kilép}_i$
- $\texttt{küld}(m)_{i,j},\ m \in \{$„igény”, „palack”, „követel”$\} \times (B_i \cap B_j),\ j \in$ *szomszédok*

Állapotok:

szakasz $\in \{Ha, Pr, Be, Ki\}$, kezdetben *Ha*
belső_szakasz $\in \{Ha, Pr, Be, Ki\}$, kezdetben *Ha*
szükséges $\subseteq B_i$, kezdetben $\emptyset$
palackok $\subseteq B_i$, kezdetben tetszőleges, de eleget tesz annak az általános korlátozásnak, hogy az összes folyamathoz tartozó *palack* halmazok partícionálják $\mathcal{R}$ összes palackjainak halmazát
elhalasztott $\subseteq B_i$, kezdetben $\emptyset$
aktuális, logikai, kezdetben *hamis*
$\forall j \in$ *szomszédok*:
- *küld_puffer*(j) az üzenetek egy FIFO sora, amelyek elemei az {„igény”, „palack”, „követelés”} $\times (B_i \cap B_j)$ halmaznak, kezdetben üres

Átmenetek:

$\texttt{próbál}(B)_i$
Hatás:
szakasz $:= Pr$
szükséges $:= B$
$\forall j \in$ *szomszédok*:
$\forall b \in$ (*szükséges* $\cap B_j$) $-$ *palackok* értékre,
do adjuk hozzá („*igény*”, b)-t a *küld_puffer*(j) sorhoz

$\texttt{küld}(m)_{i,j}$
Előfeltétel:
m *küld_puffer*(j) első eleme
Hatás:
távolítsuk el *küld_puffer* első elemét

$\texttt{belső\_próbál}_i$
Előfeltétel:
szakasz $= Pr$
belső_szakasz $= Ha$
Hatás:
belső_szakasz $:= Pr$

$\texttt{fogad}$(„követelés”,$b)_{j,i}$
Hatás:
óra $:= \max($*óra*$, c) + 1$
történet$(j) :=$ *történet*$(j) \cup \{(m, c)\}$
if $m =$ „*történet*” **and** $j \neq i$
then „*nyugtáz*”-t tegyük be *küld_puffer*$_j$-be

$\texttt{fogad}$(„palack”,$b)_{j,i}$
Hatás:
palackok $:=$ *palackok* $\cup \{b\}$

$\texttt{fogad}$(„*igény*”, b)*j,i
Hatás:
if *szakasz* $\in \{Pr, Be\}$ és $b \in$ *szükséges* **then**
elhalasztott $:=$ *elhalasztott* $\cup \{b\}$
else
tegyük be („*palack*”,b)-t a *küld_puffer*$(_j)$ sorba
palackok $:=$ *palackok* $- \{b\}$

$\texttt{belső\_belép}_i$
Hatás:
belső_szakasz $:= Be$
if *szakasz* $:= Pr$ **then**
aktuális $:=$ *igaz*
$\forall j \in$ *szomszédok*:
$\forall b \in$ (*szükséges* $\cap B_j$) $-$ *palackok* értékre, **do**
(„*követelés*”,b)-t tegyük be a *küld_puffer*$(_j)$ *sorba*

$\texttt{belső\_kilép}_i$
Előfeltétel:
belső_szakasz $= Be$
aktuális $=$ *hamis*
Hatás:
belső_szakasz $= Ki$

$\texttt{belső\_halad}_i$
Hatás:
belső_szakasz $:= Ha$

A kód két pontja igényel magyarázatot. Először, belátható, hogy amikor D_i megkapja a („*igényel*", b) üzenetet, tényleg nála van a b palack. Ezért nincs szükség arra, hogy az `igényel` igény teljesítése vagy a teljesítés elhalasztása előtt D_i megvizsgálja, teljesül-e, hogy $b \in palackok$. Másrészt viszont az lehetséges, hogy D_i olyankor kap egy („*követel*", b) üzenetet, amikor nincs nála a b palack. Ezért egy `követel` kielégítése előtt D_i megvizsgálja, teljesül-e, hogy $b \in palackok$.

Másodszor, az $aktuális_i$ jelzőbit mutatja, hogy vajon van-e aktuális belső kritikus Be szakasz, amelyik még mindig foglalt, és adható-e elsőbbség U_i aktuális igényének. Az $aktuális_i$ jelzőbit *igaz* értéket kap, ha `belső_belép` üzenet érkezik, miközben $szakasz_i = Pr$. Ha `belép`$_i$ üzenet érkezik, akkor a jelzőbit értéke hamis lesz. Amikor $aktuális_i = hamis$, D_i végrehajthatja a $belső\_kilép_i$ műveletet, befejezve ezzel a belső kritikus szakaszt.

20.6. tétel. . *A* ModulárisIvóFil *algoritmus az általános erőforrás-hozzárendelési feladat bármely zárolásmentes megoldásának felhasználásával megoldja az ivó filozófusok problémáját és garantálja a zárolásmentességet.*

Bizonyításvázlat. A jólformáltságot könnyű belátni. A kizárási feltétel abból a tényből következik, hogy a *palackok* halmazok és a `palack` üzenetek közvetlenül leírják a palackokat, továbbá abból a tényből, hogy a folyamatok az összes szükséges *palackot* igénylik a `belép` kimenethez. Belátjuk a zárolásmentességet, amiből következik a haladás. Ehhez az általános erőforrás-hozzárendelő modul tulajdonságait használjuk fel.

Először, a kódból látszik, hogy az erőforrás-hozzárendelő modul környezete megőrzi a modul jólformáltságát. Másodszor, a modul tulajdonságaiból következik, hogy a rendszer minden végrehatási sorozata teljesíti a modulra nézve a jólformáltság és a kizárás feltételeit. Tehát minden pártatlan végrehatási sorozat teljesíti a modulra nézve a zárolásmentesség feltételét.

20.7. segédtétel. . *A* ModulárisIvóFil *rendszer bármely pártatlan végrehatási sorozatára igaz, hogy ha bármely* `belép` *elemet követ a megfelelő* `kilép`*, akkor minden* `belső_belép` *után a megfelelő* `belső_kilép` *következik.*

Bizonyításvázlat. Tegyük fel, hogy egy α pártatlan végrehajtási sorozatban adott időpontban egy `belső_kilép`$_i$ műveletre kerül sor, és ezután nem fordul elő `belső_belép`$_i$; legyen α_1 α-nak az az utótagja, amelyik közvetlenül `belső_belép`$_i$ után kezdődik. Ekkor $aktuális_i$ α_1-ben végig igaz marad, mivel ha bármikor *hamis*-ra változna, `belső_kilép`$_i$ előfeltétele teljesülne és végül sor kerülne `belső_kilép`$_i$ végrehajtására. Tehát a modul kizárási feltétele szerint D_i egyik szomszédja sem lehet α_1 alatt belső kritikus szakaszában.

Amikor a `belső_belép`$_i$ esemény bekövetkezik, teljesülnie kell a $szakasz_i = Pr$ egyenlőségnek, mivel $aktuális_i$ *igaz*-ra van állítva. Ezért a `belső_belép`$_i$ esemény részeként D_i `követel` üzeneteket küld minden, számára szükséges palackért. Tekintsük egy ilyen („*igényel*", b) üzenet D_j fogadóját. Ha D_j-nél van a b palack és

éppen nem használja (azaz nincs a kritikus szakaszában úgy, hogy $b \in szükséges_j$), akkor („*palack*", b) üzenetet küld D_i-nek. Másrészt, ha D_j használja b-t, akkor – mivel minden `belép`$_j$-t követ egy `kilép`$_j$, D_j végül befejezi kritikus szakaszát és teljesíti az elhalasztott `igényel` üzenetet. Ezért D_i végül megkapja a szükséges palackokat. Azt állítjuk, hogy addig tartja magánál őket, amíg végre nem hajtja a `belép`$_i$ műveletet. Ez azért van, mert α_1 alatt nem kap `igényel` üzenetet egyik palackra sem; ehhez azt a tényt használjuk ki, hogy α_1 alatt D_i egyik szomszédja sincs a belső kritikus szakaszában. (Bizonyos invariánsokra itt szükség van.)

Mivel D_i megkap minden szükséges palackot, végül végrehajtja a `belép`$_i$ műveletet. Ennek hatására azonban *aktuális* értéke *hamis* lesz, ami ellentmondás. □

A 20.7. segédtétel lehetővé teszi, hogy bebizonyítsuk a következő, kulcsfontosságú állítást.

20.8. segédtétel. . *A* ModulárisIvóFil *rendszer bármely pártatlan végrehatási sorozatára igaz, hogy ha bármely* `belép` *elemet követ a megfelelő* `kilép`*, akkor minden* `próbál` *után a megfelelő* `belép` *következik (azaz minden igény ki lesz elégítve).*

Bizonyításvázlat. Tegyük fel, hogy egy α pártatlan végrehatási sorozat bizonyos pontján végrehajtódik a `próbál`$_i$ művelet és azután `belép`$_i$ sohasem fordul elő. legyen α_1 α-nak az az utótagja, amelyik közvetlenül `próbál`$_i$ után kezdődik.

Ha α_1-ben előfordul egy `belső_belép`, akkor a 20.7. segédtételből következik, hogy egy `belső_kilép`-re is sor kerül. Az *aktuális* jelzőbit kezelése miatt ez azonban csak úgy fordulhat elő, ha időközben egy `belép`$_i$ eseményre kerül sor. Ez ellentmondás. Ezért feltehetjük, hogy α_1-ben nem fordul elő *belső_ belép*.

Ha α_1 alatt *belső_ szakasz* valamikor is egyenlő Pr-rel, akkor a modul zárolásmentességi tulajdonságából következik, hogy végül sor kerül egy *belső_ belép* műveletre, ami ellentmondás. Ezért feltehetjük, hogy α_1 során végig fennáll, hogy *belső_ szakasz* $\neq Pr$. Ha *belső_ szakasz* α_1 alatt bármikor egyenlő Ha-val, akkor akkor végül végrehajtódik egy *belső_ próbál*$_i$, ami ahhoz vezet, hogy *belső_ szakasz* $= Pr$, és ez is ellentmondás. Ezért feltehetjük, hogy α_1-ben végig fennáll, hogy *belső_ szakasz* $\neq Ha$. A modul zárolásmentessége alapján meg fogjuk mutatni, hogy α_1 során *belső_ szakasz* $\neq Ki$ teljesül.

Így csak az a lehetőség maradt, hogy α_1-ben végig *belső_ szakasz* $= Be$. Mivel α_1 közvetlenül egy `próbál`$_i$ esemény után kezdődik, ezért α_1-ben szükségképpen *aktuális* $=$ *hamis*. Ekkor azonban végül sor kerül egy *belső_ kilép*$_i$ eseményre, aminek eredménye *belső_ szakasz* $= Ki$ lesz, ami ellentmondás. □

A ModulárisIvóFil algoritmus Pr szakaszának zárolásmentessége az ezután bizonyítandó 20.8. segédtételből következik, míg a *kilépés* szakasz zárolásmentessége közvetlenül belátható. □

Bonyolultságelemzés.. A ModulárisIvóFil algoritmus bonyolultsági korlátai az általános erőforrás-hozzárendelési modul megvalósításának költségeitől függenek. Az algoritmus D_i elemei igényenként legfeljebb $3k$ üzenetet küldenek, ahol k a G gráf csúcsai fokszámainak maximuma.

Az időbonyolultság számításához legyen l és d a szokásos, és legyen c az U_i folyamatok kritikus szakaszai hosszának egy felső korlátja. Tegyük fel, hogy T_1, ill. T_2 felső korlátai annak az időnek, amelyet egy tetszőleges folyamat a belső próbálkozási és a belső kilépési szakaszában eltölthet. (T_1 rendszerint a belső kritikus szakasz hosszára vonatkozó felső korlát függvénye.) A belső kritikus szakasz hosszának egy felső korlátja $c+3d+\mathcal{O}(l)$. Ekkor a `próbál` és a megfelelő `kilép` közötti idő legfeljebb $T_1+T_2+c+3d+\mathcal{O}(l)$.

Ha a korábbi igényekkel kapcsolatos összes üzenet már továbbítódott, azaz a vizsgált igény „szigorúan elszigetelt", akkor az időbonyolultság legfeljebb $2d+\mathcal{O}(l)$.

20.3.. Megjegyzések a fejezethez

A KörbejáróJel algoritmus Le Lanntól [191] származik. Cikke tartalmazza a kölcsönös kizárási algoritmusok hibatűrése különböző formáinak elemzését is, beleértve az elveszett jel visszaállítását a vezetőkiválasztó algoritmussal. A LogikaiIdőKK algoritmus Lamportnak [176], a RicartAgrawalaKK algoritmus Ricartnak és Agrawalának köszönhető [252]. A 20.1. alfejezet végén lévő optimalizálás Carvalho és Roucairol [64] eredménye. Raynal könyve [250] nagy számú kölcsönös kizárási algoritmust tartalmaz – mind az aszinkron hálózatokra, mind pedig az aszinkron közös memóriájú modellekre.

Az ivó filozófusok problémáját először Chandy és Misra [67] írták le. Cikkük tartalmaz egy, az AciklikusIrGráfEH-hez nagyon hasonló általános erőforrás-hozzárendelési algoritmust is, valamint az ivó filozófusok olyan megoldását, melyet az általános erőforrás-hozzárendelési algoritmusuk módosításával kaptak. A ModulárisIvóFil algoritmust az itt bemutatott formában Welch és Lynch írták le [285], Chandy és Misra ötletei alapján. A cikk többek között közvetlenül rámutatott arra, hogy a Chandy–Misra algoritmus moduláris.

A hálózatok erőforrás-hozzárendelési problémáiról szóló friss művekhez tartoznak Styer és Peterson [272], Choy és Singh, [80] valamint Awerbuch és Sachs [37] művei. Ezen írások középpontjában a futásidő és/vagy a hibatűrés javításának vizsgálata áll.

20.4.. Gyakorlatok

20-1. Adjuk meg a Váróterem kölcsönös kizárási algoritmus egyik megvalósításának előfeltétel/hatékony kódját az aszinkron küld/fogad hálózati feltételek mellett. Elemezzük az algoritmus bonyolultságát. (*Megjegyzés.* A megvalósítás előállítható az eredeti Váróterem algoritmus átalakításával vagy másképpen is.)

20-2. Adjuk meg a Peterson$n$Foly kölcsönös kizárási algoritmus előfeltétel/hatékony kódját aszinkron küld/fogad hálózatra. Elemezzük az algoritmus bonyolultságát.

20-3. Adjuk meg a 20.1. tétel bizonyításának részleteit.

20-4. Tegyük fel, hogy G egy összefüggő, irányítatlan gráf. Tervezzünk egy adott G gráfon alapuló hatékony küld/fogad hálózati algoritmust, amely megoldja, hogy a hálózatban lévő összes folyamat látszólagos gyűrűvé alakul. tegyük fel, hogy a folyamatoknak van felhasználói azonosítójuk. Minden folyamatnak kimenetként meg kell adnia a gyűrűben utána következő folyamat azonosítóját és az ehhez a rákövetkező folyamathoz vezető úton lévő folyamatok azonosítóját. A cél a rákövetkező folyamatokhoz vezető utak összhosszának minimalizálása.

20-5. Oldjuk meg a 20-4. gyakorlatot abban az esetben, amikor G erősen összefüggő irányított gráf.

20-6. Bizonyítsuk be az invariáns állítások módszerével a LogikaiIdőKK algoritmus kölcsönös kizárási tulajdonságát. (*Útmutatás.* A kulcs invariáns az, hogy ha a P_i folyamat *Be*-ben van, akkor a `próbál` üzenetéhez tartozó logikai idő kisebb, mint bármely másik olyan `próbál` üzeneté, amelyhez nem tartozik egy azt követő `kilép` üzenet.)

20-7. Adjunk a LogikaiIdőKK algoritmusra vonatkozó általános felső korlátot a $\texttt{próbál}_i$ esemény és a megfelelő $\texttt{belép}_i$ esemény között eltelő idő legrosszabb esetére. Ne hanyagoljuk el a lehetséges csatornacsomagokat.

20-8. „Optimalizáljuk" a LogikaiIdőKK algoritmust úgy, hogy a *történet* változók nem tárolják az összes beérkezett üzenetet. Tömörítsük úgy a megőrzött információt, hogy az még minden folyamat számára lehetővé tegye a korábbi viselkedést. Bizonyítsuk be optimalizált algoritmusunk helyességét a LogikaiIdőKK algoritmus szimulálása útján.

20-9. Tegyük fel, hogy úgy módosítjuk a LogikaiIdőKK algoritmust, hogy üzenet beérkezésekor minden folyamat megnöveli helyi órájának az értékét, de nem alkalmaz kiegészítő növelést annak érdekében, hogy a helyi óra értéke nagyobb legyen, mint a beérkezett üzenetben lévő érték. (Ez a 18-4. gyakorlatban leírt „nemlogikus időhöz" vezet). Milyen helyességi tulajdonságokat őriz meg ez az algoritmus? Bizonyítsuk be az állítást (mind a pozitív, mind a negatív részét.)

20-10. Írjunk egy előfeltétel/hatékony kódot a RicartAgrawalaKK algoritmusra és használjuk fel egy formális helyességbizonyításhoz. A kölcsönös kizárás bizonyítása során használjunk invariáns állításokat.

20-11. Bizonyítsuk be, hogy a RicartAgrawalaKK algoritmus kizárásmentes és adjunk felső korlátot a $\texttt{próbál}_i$ és a megfelelő $\texttt{belép}_i$ esemény közti időre.

20-12. Írjunk előfeltétel/hatékony kódot a RicartAgrawalaKK algoritmus azon javított változatára, amelyben `jóváhagy` üzenetek engedélyt adnak a kritikus szakaszba való ismételt belépésre. Bizonyítsuk be az algoritmus helyességét.

20-13. Elemezzük a 20.2.2. szakaszban bemutatott Színez algoritmus kommunikációs és időbonyolultságát.

20-14. A RicartAgrawalaEH algoritmussal kapcsolatban végezzük el a következőket.

(a) Írjunk előfeltétel/hatékony kódot.

(b) Bizonyítsuk be a helyességét.
(c) Elemezzük a bonyolultságát.
(d) Adjunk meg egy olyan végrehajtási sorozatot, amelyben a `próbál`$_i$ és a megfelelő `belép`$_i$ események közötti idő a lehető legnagyobb.

20-15. Írjunk előfeltétel/hatékony kódot a RicartAgrawalaEH algoritmus javított változatához, melyben `jóváhagy` üzenetek közvetítik az engedélyt a kritikus szakaszba való ismételt belépéshez. Bizonyítsuk be a helyességét.

20-16. A KörmentesIrGráfEH algoritmusra végezzük el a következőket.
(a) Írjunk előfeltétel/hatékony kódot.
(b) Adjunk részletes bizonyítást a helyességre.
(c) Határozzuk meg, hogy biztosítja-e a független haladást.
(d) Elemezzük a bonyolultságát.
(e) Adjunk meg egy olyan végrehajtásti sorozatot, amelyben a `próbál` eseménytől a `belép` eseményig eltelő idő akármilyen nagy lehet.
(f) Adjunk felső korlátot egy elszigetelt igény kielégítésének idejére.

20-17. Magyarázzuk meg, miért tekinthető a KörbejáróJel algoritmus a KörmentesIrGráf algoritmus speciális esetébenek.

20-18. Általánosítsuk a KörmentesIrGráfEH algoritmust úgy, hogy az erőforrás-leírásra vonatkozó két megszorítás elhagyható legyen.

20-19. Adjunk hatékony algoritmust, amely egy G összefüggő, irányítatlan gráffal megadható küld/fogad hálózatra az összes élet irányítja úgy, hogy körmentes irányított gráfot kapjunk. Feltehetjük, hogy minden folyamatnak van felhasználói azonosítója.

20-20. Fogalmazzuk meg és bizonyítsuk be a 11.2. tételhez hasonló állítást aszinkron hálózatokra vonatkozóan.

20-21. Definiáljuk a *várakozási lánc* fogalmát a 11.3.1. szakaszban leírtakhoz hasonlóan. Ennek azonban csak olyan algoritmusokra van értelme, mint a RicartAgrawalaEH és a KörmentesIrGráfEH, amelyekben a folyamatok nem szereznek meg egyéni erőforrásokat. Alkalmazzuk a definíciót az ebben a fejezetben tárgyalt erőforrás-hozzárendelő algoritmusok várakozási láncai hosszának elemzésére.

20-22. A Zöldfülűek Számítástechnikai Rt. programozói úgy döntöttek, mepróbálják megjavítani a KörmentesIrGráfEH algoritmust. Nevezetesen, ha egy *Ha* szakaszban lévő folyamatnak minden éle befelé van irányítva, csak akkor irányítja éleit kifelé, ha valamelyik szomszédjától `próbál` üzenetet kap. Ha a P_i folyamat az U_i felhasználótól `próbál`$_i$ üzenetet kap, akkor `próbál` üzenetet küld minden szomszédjának. Miért rossz ez a stratégia?

20-23. *Kutatási kérdés.* Tervezzünk küld/fogad hálózati algoritmust adott $\mathcal{R}$ erőforrás-leíráson alapuló általános erőforrás-hozzárendelő feladat megoldására. Tegyük fel, hogy minden olyan folyamatpár, amelynek van közös erőforrása, a hálózatot megadó G gráfban éllel össze van kötve. Olyan algoritmust tervezzünk,

melynek alacsony az egy olyan igényre eső időbonyolultsága, melyeknél kevés (k) az egymást időben átfedő, versenyző igények száma. Törekedjünk k-ban lineáris korlát elérésére.

20-24. *Kutatási kérdés.* Tervezzünk küld/fogad hálózati algoritmust adott $\mathcal{R}$ erőforrás-leíráson alapuló általános erőforrás-hozzárendelési feladat megoldására. Tegyük fel, hogy minden olyan folyamatpár, amelyeknek van közös erőforrása, a hálózatot megadó G gráfban éllel össze van kötve. Olyan algoritmust tervezzünk, amely bármely P_i folyamat számára biztosítja a kizárásmentességet még akkor is, ha olyan folyamatok megállási hibája is előfordulhat, amelyek távolsága G-ben legalább k. Törekedjünk k minimalizálására.

20-25. Dolgozzuk ki a 20.6. tétel bizonyításának összes részletét. Több invariáns állításra szükségünk lesz, többek között a következőre.

> **20.4.1. állítás.** *Ha $b \in$ palackok$_i$ és egy („igény”,b) üzenet halad D_j-től D_i felé, akkor szakasz$_j$ = Pr, belső_szakasz$_j$ = Be és aktuális$_j$ = igaz.*

20-26. Bizonyítsuk be, hogy $T_1 + T_2 + c + 3d + \mathcal{O}(l)$ felső korlát a ModulárisIvóFil időbonyolultságára.

20-27. Alkalmazzuk a hálózatok kezelésére átalakított Színez algoritmust felhasználó ModulárisIvóFil algoritmust az általános erőforrás-hozzárendelő modul megvalósítára. Adjunk felső korlátot arra az időre, amíg egy felhasználónak igénye kielégítésére várnia kell. Bizonyítsuk be a korlát helyességét.

20-28. Általánosítsuk úgy a ModulárisIvóFil algoritmust, hogy az erőforrás-leírásra vonatkozó korlátozást elhagyjuk.

21. fejezet

Aszinkron hálózati számítás folyamat hibákkal

Ebben a fejezetben azt vizsgáljuk, hogy mit tudunk és mit nem tudunk kiszámolni aszinkron hálózatokban folyamat megállási hibák előfordulásakor. Itt csak folyamat hibákat tekintünk, és feltételezzük, hogy a kommunikáció megbízható.

Annak megmutatásával kezdjük, hogy kiszámíthatósági eredmények elérése szempontjából nem számít, hogy küld/fogad vagy üzenetszóró rendszereket tekintünk.

Ezután újrafogalmazzuk az osztott megegyezés feladatára vonatkozó megoldhatatlansági eredményt aszinkron hálózati modellben. Ez az eredmény azt mondja, hogy a megegyezési feladat még akkor sem oldható meg aszinkron hálózatokban, ha biztosított, hogy nincs egynél több folyamat hiba. A 12. fejezetben tárgyaltuk ezt a feladatot, és hasonló megoldhatatlansági eredményt kaptunk aszinkron közös memóriájú környezetben. Ahogy a 12. fejezet elején megjegyeztük, az ilyen megoldhatatlansági eredmények gyakorlati következményekkel járnak olyan osztott alkalmazásokra, amelyeknél megköveteljük a megegyezést. Ezek magukban foglalják azokat az adatbázis rendszereket, amelyek megegyezést követelnek a tranzakciók elfogadásáról vagy elutasításáról, azokat a kommunikációs rendszereket, amelyek megegyezést követelnek az üzenet kézbesítésről, és azokat a folyamat ellenőrző rendszereket, amelyek megegyezést igényelnek a hiba meghatározásáról. A megoldhatatlansági eredményekből következik, hogy nincs helyesen működő tisztán aszinkron algoritmus.

A fejezet hátralévő részében néhány módszert írunk le ennek az alapvető nehézségnek a kikerülésére: alkalmazunk véletlenítést, a modellt hibajelző mechanizmusokkal erősítjük, egy értékű helyett inkább több értékű halmazon való megegyezést, és pontos helyett inkább közelítő megegyezést használunk.

A fejezet erősen támaszkodik korábbi fejezetekre, különösen a 7., 12. és 17. fejezetre. Főként sok, aszinkron hálózatokban való kiszámíthatósággal kapcsolatos eredmény következik közvetlenül analóg aszinkron olvasható/írható közös memóriájú rendszerekben való kiszámíthatósági eredményből általános átalakítások révén.

21.1.. A hálózati modell

Ebben a fejezetben feltételezzük, hogy a modell egy aszinkron üzenetszóró rendszer megbízható üzenetszóró csatornákkal és folyamat megállási hibákkal (amelyeket `megállít` eseményekkel modellezünk). Tekinthetnénk megbízható FIFO küld/fogad csatornájú aszinkron küld/fogad rendszereket is különböző folyamatok összes párja között, azonban ki fog derülni, hogy a két modell kiszámíthatósági szempontból egyenértékű. Nem nehéz látni, hogy az üzenetszóró modell legalább olyan hatásos, mint a küld/fogad modell. A következő tétel azt mutatja, hogy nem hatásosabb.

21.1. tétel. . *Ha A egy tetszőleges aszinkron üzenetszóró rendszer megbízható üzenetszóró csatornával, akkor létezik olyan B aszinkron küld/fogad rendszer megbízható FIFO küld/fogad csatornákkal, amelynek a felhasználói felülete megegyezik A-éval, és „szimulálja" A-t az alábbi módon. B minden α végrehajtási sorozatára van A-nak olyan α' végrehajtási sorozata, amely eleget tesz a következő feltételeknek:*

1. *α és α' megkülönbözhetetlen egymástól U (az U_i felhasználók összessége) számára;*
2. *minden i esetében $\texttt{megállít}_i$ pontosan akkor fordul elő α-ban, ha előfordul α'-ben.*

Továbbá, ha α helyes, akkor α' is helyes.

Bizonyításvázlat. Az A rendszer minden egyes P_i folyamatához tartozik a B rendszer egy Q_i folyamata. Mindegyik Q_i felelős P_i szimulálásáért, továbbá részt vesz az üzenetszóró csatorna szimulálásában.

Q_i a P_i egy $\texttt{szór}(m)_i$ kimenetét szimulálja, végrehajtva $\texttt{küld}(m,t)_{i,j}$ kimeneteket minden $j \neq i$ esetében, ahol t egy lokális egész értékű *címke*, majd végrehajtva egy belső lépést, szimulálva $\texttt{fogad}(m)_{i,i}$-t. A Q_i által használt *címke* érték 1-gyel indul, és minden egyes sikeres `szór` esetében nő 1-gyel. Ha Q_i egy Q_j által küldött (m,t) üzenetet kap, akkor segít P_j üzenetszórásának szimulálásában, továbbítván az üzenetet. Speciálisan, elküldi az (m,t,j) üzenetet az összes i-től és j-től különböző folyamatnak. Ha Q_i (m,t,j)-t kapja k-tól, akkor folytatván a segítséget elküldi (m,t,j)-t az összes i, j és k-tól különböző folyamatnak, amelynek még nem küldte el (m,t,j)-t.

Ezalatt Q_i összegyűjti az eredetileg P_j-k, $j \neq i$, által sugárzott címkézett (m,t) üzeneteket; ezeket vagy közvetlenül Q_j-től kapja, vagy pedig továbbítás útján. Időnként megengedjük, hogy Q_i végrehajtson egy belső lépést szimulálva az A rendszer $\texttt{fogad}(m)_{j,i}$ eseményét. Speciálisan ez történik, amikor Q_i egy eredetileg P_j által sugárzott (m,t) üzenetet kap, Q_i már továbbította (m,t,j)-t az összes i-től és j-től különböző folyamatnak, és Q_i már szimulálta a $\texttt{fogad}_{j,i}$ eseményeket az olyan P_j-től származó üzenetekre, amelyek *címke* értéke szigorúan kisebb, mint t.

A bizonyítás néhány kulcs gondolata a következő. Először megjegyezzük, hogy nincs olyan Q_i folyamat, amely egy $\texttt{fogad}(m)_{j,i}$ eseményt szimulál bármely j esetében, amíg nem sikerült elküldeni a megfelelő (m,t)-t az összes többi folyamatnak, és így ezután már biztosított, hogy végül az összes folyamat megkapja

(m,t)-t j-től. Másodszor vegyük észre, hogy bár egy Q_i folyamat kaphatja attól eltérő sorrendben az eredetileg P_j által sugárzott üzeneteket ahogy P_j sugározta, a *címke* értékek lehetővé teszik, hogy Q_i a helyes sorrendbe rendezze őket. Harmadszor, hogy ha bármelyik Q_i folyamat elküldött egy t címkéjű üzenetet, akkor az összes kisebb *címke* értékű, eredetileg P_i-től származó üzenet már szükségképp korábban el lett küldve az összes folyamatnak. □

A 21.1. tétel alapján kiszámíthatósági szempontból nem jelent problémát, hogy üzenetszóró rendszereket vagy küld/fogad rendszereket tekintünk. Természetesen a bonyolultságuk eltérő – a `fogad` események összes számát a fent leírt szimulációban közelítőleg n-nel kell szorozni – azonban a bonyolultsággal nem foglalkozunk ebben a fejezetben. Azért választottuk az üzenetszóró rendszereket, mert a megoldhatatlansági eredményeket kissé élesebbekké teszik, és mert az algoritmusok könnyebben írhatók le.

21.2.. Megegyezés megoldhatatlansága hibák esetében

A megegyezési feladat 12.1. alfejezetbeli definícióját használjuk. Bár ott közös memóriájú rendszerekre fogalmaztuk meg, a feladat értelmes aszinkron (üzenetszóró vagy küld/fogad) hálózati rendszerekre is. Ezt tekintjük át itt.

Az A rendszer felhasználói felülete $\texttt{kezd}(v)_i$ bemeneti műveletekből és $\texttt{választ}(v)_i$ kimeneti műveletekből áll, ahol $v \in V$ és $1 \leq i \leq n$; A rendelkezik még $\texttt{megállít}_i$ kimeneti műveletekkel. Minden i indexű műveletre azt mondjuk, hogy az i-edik *kapu*nál fordul elő. Mindegyik U_i felhasználó rendelkezik $\texttt{kezd}(v)_i$ kimenetekkel és $\texttt{választ}(v)_i$ bemenetekkel, $v \in V$. Feltételezzük, hogy U_i legfeljebb egy $\texttt{kezd}_i$ műveletet hajt végre minden egyes végrehajtási sorozatnál.

Műveleteknek egy $\texttt{kezd}_i$ és $\texttt{választ}_i$ sorozatát *jólformáltnak* nevezzük i-re, ha prefixe egy $\texttt{kezd}_i(v)$, $\texttt{választ}_i(w)$ alakú sorozatnak. A következő feltételeket vesszük figyelembe az A-ból és U_i felhasználókból álló összetett rendszerre.

Jólformáltság. Bármely végrehajtási sorozatnál és tetszőleges i esetében az U_i és A közötti kölcsönhatás jólformált i–re.

Megegyezés. Bármely végrehajtási sorozatnál az összes döntési érték megegyezik.

Érvényesség. Bármely végrehajtási sorozatnál, ha az összes előforduló `kezd` művelet ugyanazt a v értéket tartalmazza, akkor v az egyetlen lehetséges döntési érték.

Hibamentes befejezés. Bármely olyan helyes, hibamentes végrehajtási sorozatnál, ahol az összes kapunál előfordul `kezd` esemény, egy `választ` esemény mindegyik kapunál bekövetkezik.

Azt mondjuk, hogy egy aszinkron hálózati rendszer *megoldja az egyetértési feladatot*, ha (a felhasználók minden együttesére) biztosítja a jólformáltságot, a megegyezést, az érvényességet és a hibamentes befejezést. Szükségünk van még a következő meghatározásra.

f-hiba befejezés, $0 \leq f \leq n$. Bármely olyan helyes végrehajtási sorozatnál, ahol minden kapunál előfordul `kezd` esemény, ha legfeljebb f kapunál van `megállít` esemény, akkor egy `választ` esemény minden hibamentes kapunál bekövetkezik.

A *várakozásmentes befejezést* az f-hibás befejezés azon speciális eseteként definiáljuk, amikor $f = n$.

Természetesen könnyű megoldani a megegyezési feladatot aszinkron üzenetszóró modellben akkor, amikor a hibatűrés követelménye nem áll fenn. Például mindegyik folyamat egyszerűen sugározhatja a kezdő értékét, és alkalmazzunk egy alkalmas korábban egyeztetett függvényt a kapott kezdeti értékek vektorára. Mivel biztosított, hogy minden folyamat értékeknek ugyanazt a vektorát kapja, az összes ugyanazt az eredményt szolgáltatja.

Az üzenetszóró rendszerekre vonatkozó fő megoldhatatlansági eredmény (megismételve a 17.2.3. szakaszból) az alábbi állítás.

21.2. tétel. . *A megbízható üzenetszóró csatornájú aszinkron üzenetszóró rendszerben nincs olyan algoritmus, amely megoldja a megegyezési feladatot és biztosítja az 1-hibás befejezést.*

A 17.2.3. szakaszban adott bizonyítás azon múlik, hogy az aszinkron üzenetszóró rendszerek átalakíthatók aszinkron közös memóriájú rendszerekké (17.8. tétel), és a megegyezési feladatra vonatkozó megoldhatatlansági eredmény fennáll aszinkron közös memóriájú modellre (12.8. tétel). Be lehet bizonyítani a megoldhatatlansági eredményt közvetlenül is — a 12.8. tételéhez hasonló bizonyítást használva. Ezt az alternatív bizonyítást meghagyjuk gyakorlatnak (lásd 21-4. gyakorlat).

21.3.. Egy véletlenített algoritmus

A 21.2. tétel azt állítja, hogy a megegyezési feladatot még egy megállási hiba esetében sem lehet megoldani aszinkron hálózati rendszerben. Azonban ez a feladat olyan alapvető az osztott számítások elméletében, hogy fontos módszereket találni ezen lényeges korlát megkerülésére. Ahhoz, hogy egy algoritmust kapjunk, késznek kell lennünk akár a helyességi követelmények gyengítésére, akár a modell szigorítására, akár mindkettőre.

Ebben a fejezetben mindkettővel élünk. Megmutatjuk, hogy a megegyezési feladat megoldható *véletlenített* aszinkron hálózatban. Ez a modell erősebb, mint az egyszerű aszinkron hálózati modell, mivel megengedi, hogy a folyamatok véletlen választásokat tegyenek a számítás során. Másrészt a helyességi feltételek egy kicsit gyengébbek, mint korábban: bár még biztosítják a jólformáltságot, megegyezést és érvényességet, a befejezési feltétel most véletlen. Nevezetesen, az összes hibamentes folyamat legalább $p(t)$ valószínűséggel választ az összes bemenet beérkezése után t idővel, ahol p egy speciális monoton nemcsökkenő, nemkorlátos függvény. Ez magával vonja az 1 valószínűségű végső befejezést.

A következő alfejezetekben más módszereket tekintünk a 21.2. tételben kimondott lényeges korlátok megkerülésére, beleértve hibajelzők használatát, megengedve egynél több döntési értéket, és megengedve pontos helyett közelítő megegyezést.

A Ben-Or által adott algoritmus $n > 3f$ és $V = \{0, 1\}$ esetében működik. Formálisan az algoritmus egy példa a 8.8. alfejezetben leírt valószínűségi modellre.

BenOr algoritmus (vázlatosan)

Mindegyik P_i folyamat lokális változója x és y kezdetben *nulla* értékkel. Egy $\texttt{kezd}(v)_i$ bemenet hatására a P_i folyamat elvégzi az $x := v$ értékadást. P_i *szintek* egy $1, 2, \ldots$-vel számozott sorozatát hajtja végre, ahol minden egyes szint két *menetből* áll. P_i az első szinttel kezd, miután megkapta a $\texttt{kezd}_i$ bemenetbeli kezdeti értékét. Ezután az algoritmus vég nélkül, még döntés után is, folytatja a végrehajtást.

P_i a következőket teszi minden egyes $s \geq 1$ szinten.

1. menet. P_i sugározza $(\text{„}első\text{”}, s, v)$-t, ahol v az x aktuális értéke, majd vár amíg nem kap $n - f$ számú $(\text{„}első\text{”}, s, *)$ alakú üzenetet. Ha az összes ilyen üzenet ugyanazzal a v értékkel rendelkezik, akkor P_i az $y := v$ értéket választja, egyébként pedig $y := \mathit{nulla}$ lesz.

2. menet. P_i sugározza $(\text{„}második\text{”}, s, v)$-t, ahol v az y aktuális értéke, majd vár amíg nem kap $n-f$ számú $(\text{„}második\text{”}, s, *)$ alakú üzenetet. Három eset van. Először, ha az összes ilyen üzenet ugyanazt a $v \neq \mathit{nulla}$ értéket tartalmazza, akkor P_i elvégzi az $x := v$ értékadást és végrehajtja $\texttt{választ}(v)_i$-t, ha még nem tette meg. Másodszor, ha legalább $n - 2f$ számú, de nem az összes üzenet ugyanazt a $v \neq \mathit{nulla}$ értéket veszi fel, akkor P_i az $x := v$ értéket választja (azonban nem dönt). (Az $n > 3f$ feltételből következik, hogy nem lehet két különböző ilyen v érték.) Egyébként P_i véletlenszerűen 0 vagy 1 értéket ad x-nek egyenlő valószínűséggel.

Megjegyezzük, hogy a BenOr algoritmus szervezése hasonló a 6.3.3. szakaszbeli TurpinCoan algoritmuséhoz.

21.3. lemma. . *A* BenOr *algoritmus biztosítja a jólformáltságot, a megegyezést és az érvényességet.*

Bizonyítás. A jólformáltság közvetlenül adódik. Az érvényességhez tegyük fel, hogy az összes $\texttt{kezd}$ esemény, amely egy végrehajtáson belül fordul elő, ugyanazt a v értéket tartalmazza. Ekkor könnyen látható, hogy bármely olyan folyamat, amely befejezi az első szintet, a v értéket kell, hogy válassza ezen a szinten. Ez azért igaz, mert v az egyetlen érték, amelyet egy folyamat küld vagy fogad egy $(\text{„}első\text{”}, 1, *)$ üzenetben, így v az egyetlen érték, amely elküldésre kerülhet egy $(\text{„}második\text{”}, 1, *)$ üzenetben.

A megegyezéshez tegyük fel, hogy a P_i folyamat v-t választja az s szinten, és egyetlen folyamat sem választ egy kisebb számú szinten. Ekkor szükségképp $n-f$ számú $(\text{„}második\text{”}, s, v)$ üzenetet fogad P_i. Ebből következik, hogy legalább $n-2f$ számú $(\text{„}második\text{”}, s, v)$ üzenetet kap minden más P_j folyamat, amely befejezi az s-edik szintet, mivel az összes, de legfeljebb f számú olyan folyamattól kap üzenetet, amelytől P_i is kapott. Ez azt jelenti, hogy P_j nem választhat egy v-től különböző értéket az s szinten; továbbá, P_j az $x := v$ értéket adja az s szinten. Mivel ez minden olyan P_j-re teljesül, amely befejezi az s szintet, (mint az érvényességre adott érvelésnél) következik, hogy minden folyamat, amely befejezi az $(s+1)$-edik szintet, a v értéket kell, hogy válassza az $(s+1)$-edik szinten. □

Innentől a befejezést tekintjük. Először, nem nehéz látni, hogy az algoritmus egymást követő szinteken át halad előre; ez nem függ a valószínűségektől.

21.4. lemma. . *A* BenOr *algoritmus minden olyan helyes végrehajtása esetében, ahol az összes kapunál bekövetkezik* `kezd` *esemény, minden hibamentes folyamat végtelen sok szintet fejez be. Továbbá, ha ℓ egy felső korlát minden folyamat feladatidejére, és d egy felső korlát a legrégebbi üzenet kézbesítési idejére minden egyes P_i és P_j közötti átvitelnél, akkor minden hibamentes folyamat az utolsó* `kezd` *esemény után $\mathcal{O}(s(d+\ell))$ idő alatt befejezi az s szintet.*

Azonban a 21.4. lemmából nem következik, hogy minden hibamentes folyamat végül választ. Ki fog derülni, hogy a BenOr algoritmus nem biztosítja ezt a tulajdonságot, ez csak véletlenszerűen érvényes.

21.3.1. példa. Választásmentes végrehajtás

A BenOr algoritmus egy olyan helyes végrehajtását írjuk le $n = 3f + 1$ esetében, amelyben egyetlen folyamat sem választ. Minden s szinten ugyanúgy, az alábbi módon járunk el.

m számú folyamat, ahol $f + 1 \leq m \leq 2f$, kezdjen $x = 0$-val, a többi pedig $x = 1$-gyel. Az első menet után minden folyamatnál $y = \textit{nulla}$, a második menetben pedig válassza egy folyamat az ő x–ének új értékét véletlenszerűen. Ekkor a véletlen választások közül m' számú, ahol $f + 1 \leq m' \leq 2f$, 0 lesz, a többi pedig 1, ami arra vezet, hogy m' számú folyamat $x = 0$-val, a maradék pedig $x = 1$-gyel kezdi az $(s + 1)$-edik szintet.

Mint a 11.4. alfejezetben, gondoljuk azt, hogy az algoritmusban minden nemdeterminisztikus választás – mi lesz a következő művelet, mikor következik be, és milyen állapot az eredménye – egy *ellenfél* ellenőrzése alatt áll. Kényszerítjük az ellenfelet arra, hogy érvényre jusson az összes b/k automata folyamat és üzenetszóró csatorna automata pártatlansági feltétele. Arra is kényszerítjük, hogy betartsa a szokásos idő korlátozásokat: a folyamatokon belüli feladatok idejére adott ℓ felső korlátot, és a legrégebbi üzenet kézbesítési idejére adott d felső korlátot bármely P_i-től P_j-ig tartó átvitel esetében. Végül megköveteljük, hogy az ellenfél minden kapunál engedjen meg `kezd` eseményeket. Feltételezzük, hogy az ellenfél teljes tudással bír a múltbeli végrehajtási sorozatokról. Minden ilyen ellenfél egy valószínűségi eloszlást határoz meg az algoritmus végrehajtási sorozatain.

21.5. lemma. . *Bármely ellenfél és $s \geq 0$ esetében minden hibamentes folyamat az $(s + 1)$-edik szintig legalább $1 - (1 - \frac{1}{2^n})^s$ valószínűséggel választ.*

Bizonyításvázlat. Az $s = 0$ eset nyilvánvaló. Tekintsünk egy tetszőleges $s \geq 1$ szintet. Belátjuk, hogy legalább $\frac{1}{2^n}$ valószínűséggel minden hibamentes folyamat ugyanazt az x értéket választja az s szint végén (függetlenül attól, hogy más szinteken hogyan döntöttünk a véletlen választásoknál). Ebben az esetben, a

megegyezésnél használt érvelés alapján, minden hibamentes folyamat választ az $(s+1)$-edik szint végére.

Tekintsünk ezen az s szinten egy olyan legrövidebb α véges végrehajtási sorozatot, amelyben egy hibamentes folyamat, mondjuk P_i, $n-f$ számú (*„első”*, s, $*$) üzenetet kapott. (Így α ezen üzenetek egyikének kézbesítésével fejeződik be.) Ha legalább $f+1$ ilyen üzenet tartalmaz egy bizonyos v értéket, akkor ezt a v-t α utáni *jó* értéknek nevezzük; vagy egy, vagy két jó érték lehet. Azt állítjuk, hogy ha csak egy jó v érték van α után, akkor minden (*„második”*, s, $*$) üzenetnek, amelyet α egy kiterjesztése küldött, tartalmaznia kell vagy a v, vagy a *nulla* értéket. Ez azért van, mert ha P_i v-nek $f+1$ számú példányát kapja, akkor az összes többi folyamat megkapja v legalább egy példányát, és így nem küldhet egy (*„második”*, s, $\overline{v}$) üzenetet. (Itt $\overline{v}$-vel jelöltük az $1-v$ értéket.) Hasonlóan, ha két jó érték van α után, akkor minden (*„második”*, s, $*$) üzenetnek, amelyet α egy kiterjesztése küldött, tartalmaznia kell *nulla*-t.

Ebből következik, hogy ha v az egyetlen jó érték, akkor v az egyetlen olyan érték, amelyre kikényszeríthető, hogy egy *determinisztikus* felsorolás révén mindegyik folyamat x értéke legyen az s szint végén α minden kiterjesztése esetében. Hasonlóan, ha két jó érték van, akkor egyetlen értéket sem kényszeríthetünk ezen az úton. Mivel egyetlen folyamat sem választ véletlenszerűen α-ban az s szinten, ezért az s szinten kényszeríthető értékek meghatározhatók, mielőtt bármilyen véletlen választást is tennénk az s szinten.

Így, ha pontosan egy jó érték van, akkor legalább $1/2^n$ valószínűséggel az összes olyan folyamat jó értéket választ, amely az x értékét véletlenszerűen választotta, így megegyezik azokkal, amelyek determinisztikusan választanak. Hasonlóan, ha két jó érték van, akkor legalább $1/2^n$ valószínűséggel az összes folyamat ugyanazt az x értéket választja (véletlenszerűen). Mindegyik esetben minden hibamentes folyamat legalább $1/2^n$ valószínűséggel ugyanazzal az x értékkel fejeződik be az s szint végén.

Eddig minden egyes s szintbeli okoskodás csak az s szinten tett véletlen választásoktól függött, ezek pedig függetlenek a más szinteken tett választásoktól. Így összeszorozhatjuk a különböző szintekhez tartozó valószínűségeket, amelyből látható, hogy legalább $1-(1-\frac{1}{2^n})^s$ valószínűséggel minden hibamentes folyamat ugyanazt az x értéket kapja egy s', $1 \le s' \le s$, szint végén. Ezért legalább $1-(1-\frac{1}{2^n})^s$ valószínűséggel minden hibamentes folyamat választ az $(s+1)$-edik szintig. □

Definiáljunk most egy $\mathbb{N}^+$-ból $\mathbb{R}_0^+$-ba képező T függvényt úgy, hogy mindegyik hibamentes folyamat az utolsó `kezd` esemény után $T(s)$ idővel fejez be egy tetszőleges s szintet. A 21.4. lemma alapján $T(s)$ választható úgy, hogy az $\mathcal{O}(s(d+\ell))$ legyen. Definiáljuk még $p(t)$-t úgy, hogy 0 legyen ha $t < T(1)$ és $1-(1-\frac{1}{2^n})^{s-1}$ ha $s \ge 1$ és $T(s) \le t \le T(s+1)$. A 21.5. és 21.4. lemmából következik a

21.6. lemma. . *Bármely ellenfél és $t \ge 0$ esetében $p(t)$ valószínűséggel minden hibamentes folyamat az utolsó* `kezd` *esemény után t időn belül választ.*

A fő helyességi eredmény a

21.7. tétel. . *A* BenOr *algoritmus biztosítja a jólformáltságot, a megegyezést és*

az érvényességet. Az algoritmus azt is biztosítja, hogy minden hibamentes folyamat végül 1 valószínűséggel választ.

Bizonyítás. A 21.3., 21.6. és 21.4. lemma alapján. (A 21.4. lemma annak megmutatásához szükséges, hogy $p(t)$ nemkorlátos.) □

Véletlenített kontra determinisztikus protokollok.. Az egyik ok, amiért a BenOr algoritmus jelentős, az az, hogy rámutat a lényeges különbségre a véletlenített és a determinisztikus aszinkron hálózati modellek között. Nevezetesen, a megegyezési feladat egyáltalán nem oldható meg folyamat hibák jelenléte esetében a determinisztikus modellben, azonban könnyen (1 valószínűséggel) megoldható a véletlenített modellben. Hasonló ellentétet mutat a LehmannRabin algoritmus a 11.4. alfejezetben.

A bonyolultság csökkentése.. A BenOr algoritmus nem gyakorlatias, mivel nagy a valószínűségi időkorlátja. Lehet úgy javítani az időbonyolultságon, hogy megnöveljük a valószínűségét annak, hogy a különböző folyamatokbeli véletlen értékek megegyeznek ugyanazon a szinten. Ez azonban kriptográfiai módszerek használatát igényli, amely kívül esik az itt tárgyalt modellen.

21.4.. Hibajelzők

Egy másik út a megegyezési feladat megoldására hibára hajlamos aszinkron hálózatoknál az, ha újfajta rendszerkomponensek, ún. *hibajelzők* hozzáadásával bővítjük a modellt. A hibajelző egy olyan modul, amely a korábbi folyamat hibákról ad információt a folyamatok számára egy aszinkron hálózatban. Különféle hibajelzők léteznek aszerint, hogy a megállásokkal kapcsolatos információk mindig helyesek-e, vagy a hibajelző teljes-e. A legegyszerűbb a *tökéletes hibajelző*, amely biztosítja azt, hogy csak a valóban megtörtént hibákat jelenti, és végül minden ilyen hibát jelent az összes hibamentes folyamatnak.

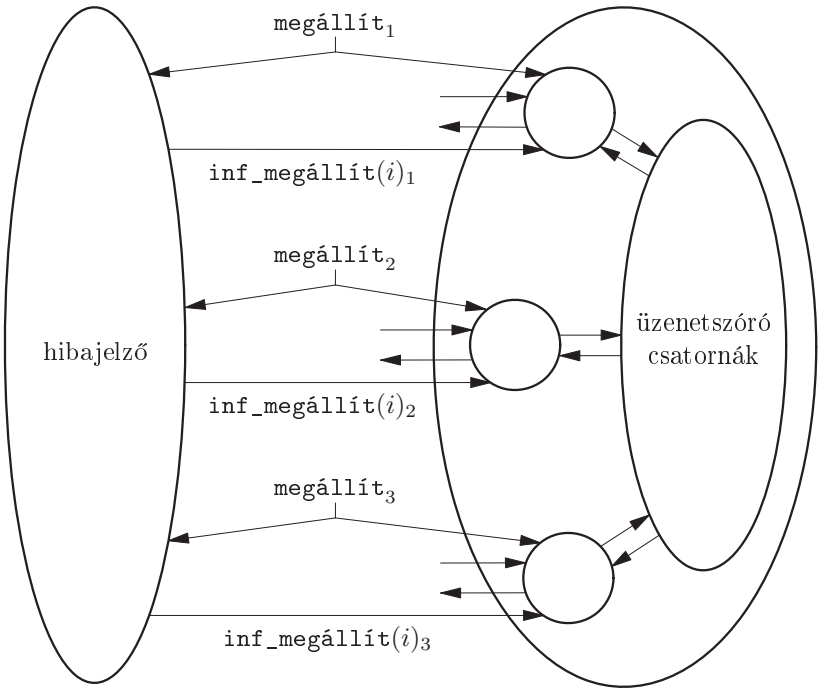
Formálisan egy olyan A rendszert tekintünk, amely egy aszinkron hálózati rendszerrel megegyező szerkezettel bír attól eltekintve, hogy további `inf_megállít`$(j)_i$ bemeneti műveletekkel rendelkezik az i és j kapuk, $i \neq j$, minden párja esetében. Egy *tökéletes hibajelző* egy olyan b/k automata az A rendszer számára, amelynek `megállít`$_i$, $1 \leq i \leq n$, műveletei, mint bemenetek, és `inf_megállít`$(j)_i$, $1 \leq i, j \leq n$, műveletei, mint kimenetek, vannak. Az ötlet az, hogy a hibajelző értesül a megállási hibákról, amelyek valahol a hálózatban fordulnak elő, és tájékoztatja róluk a többi folyamatot. Egy `inf_megállít`$(j)_i$ művelet egy értesítés kíván lenni az i-edik kapunál arról, hogy a j-edik folyamat megállt. A 21.1. ábra egy háromfolyamatú rendszer felépítését mutatja. A következő algoritmus megoldja a megegyezési feladatot amennyiben tökéletes hibajelzőt használunk.

TökéletesHJMegegyezés algoritmus (vázlatosan)

Mindegyik P_i folyamat két adatot próbál meg stabilizálni.

1. Egy *érték* vektort, amely $\{1, \ldots, n\}$-nel van indexelve és értékei a $V \cup \{nulla\}$-ból vannak. Ha $érték(j) = v \in V$, akkor az azt jelenti, hogy P_i arról értesült, hogy P_j kezdeti értéke v.
2. A folyamatok indexeinek egy *megállított* halmaza. Ha $j \in$ *megállított*, akkor az azt jelenti, hogy P_i arról értesült, hogy P_j megállt.

A P_i folyamat állandóan sugározza az érvényes *érték* és *megállított* adatát, és frissíti azokat a *megállított* halmazon kívüli folyamatoktól származó új adatok átvételével. A már *megállított* halmazba került folyamatoktól származó üzeneteket figyelmen kívül hagyja. P_i emellett nyomon követi azokat a folyamatokat, amelyek „megerősítik" az adatait, azaz amelyektől ugyanazokat a (*érték*, *megállított*) adatokat kapja, amelyekkel már rendelkezett. Amikor P_i elér egy olyan pontot, ahol az adatai már „stabilizálódtak", azaz amikor megerősítést kapott az érvényes adataira az összes nem megállt folyamattól, akkor P_i az *érték* vektorában lévő legkisebb indexű nem *nulla* értéket választja.



21.1.. ábra. Tökéletes hibajelzővel ellátott aszinkron üzenetszóró berendezés felépítése.

A kód a következő. Jelölje W a $V \cup \{nulla\}$ elemeiből képzett, $\{1, \ldots, n\}$-nel indexelt vektorok halmazát. Definiáljunk egy parciális rendezést a (w, I) párokon, ahol $w \in W$ és $I \subseteq \{1, \ldots, n\}$. Nevezetesen, azt írjuk, hogy $(w, I) \leq_d (w', I')$,

és azt mondjuk, hogy (w', I') dominálja (w, I)-t, ha a következők mindegyike fennáll:

1. minden k-ra, ha $w(k) \in V$, akkor $w(k) = w'(k)$;

2. $I \subseteq I'$.

Ez azt a képet ragadja meg, hogy (w', I') legalább annyi információt tartalmaz mint (w, I).

A zűrzavar elkerülése végett nem írjuk le világosan P_i viselkedését, miután egy `megállít`$_i$ esemény előfordul. Ez éppen a szokásos – a folyamat megáll.

21.1. automata. TökéletesHJMegegyezés

Lenyomat:

Bemeneti:
- $\texttt{kezd}(v)_i$, $v \in V$
- $\texttt{fogad}(w, I)_{j,i}$, $w \in W$, $I \subseteq \{1, \ldots, n\}$, $1 \leq j \leq n$
- $\texttt{inf\_megállít}(j)_i$, $j \neq i$

Kimeneti:
- $\texttt{szór}(w, I)_i$, $w \in W$, $I \subseteq \{1, \ldots, n\}$
- $\texttt{választ}(v)_i$, $v \in V$

Állapotok:

érték $\in W$, kezdetben azonosan *nulla*
megállított $\subseteq \{1, \ldots, n\}$, kezdetben üres
megerősített $\subseteq \{1, \ldots, n\}$, kezdetben üres
választott, logikai, kezdetben *hamis*

Átmenetek:

$\texttt{kezd}(v)_i$
- Hatás:
  - *érték*$(i) := v$
  - *megerősített* $:= \{i\}$

$\texttt{inf\_megállít}(j)_i$
- Hatás:
  - *megállított* $:=$ *megállított* $\cup \{j\}$
  - *megerősített* $:= \{i\}$

$\texttt{szór}(w, I)_i$
- Előfeltétel:
  - $w =$ *érték*
  - $I =$ *megállított*
  - *érték*$(i) \neq$ *nulla*
- Hatás:
  - nincs

$\texttt{fogad}(w, I)_{j,i}$
- Hatás:
  - **if** $j \notin$ *megállt* **then**
    - **if** $(w, I) =$ (*érték*, *megállított*) **then**
      - *megerősített* $:=$ *megerősített* $\cup \{j\}$
    - **else if** $(w, I) \not\leq_d$ (*érték*, *megállított*) **then**
      - *megállított* $:=$ *megállított* $\cup I$
      - $\forall\ k,\ 1 \leq k \leq n$, **do**
        - **if** *érték* $=$ *nulla* **then** *érték* $:= w(k)$
      - *megerősített* $:= \{i\}$

$\texttt{választ}(v)_i$
- Előfeltétel:
  - *megerősített* $\cup$ *megállított* $= \{1, \ldots, n\}$
  - $v =$ *érték*(j), ahol j a legkisebb $v(j) \neq$ *nulla* indexű
  - *választott* $=$ *hamis*
- Hatás:
  - *választott* $:=$ *igaz*

Taszkok:

$\{\texttt{szór}(w, I)_i : w \in W,\ I \subseteq \{1, \ldots, n\}\}$
$\{\texttt{választ}(v)_i : v \in V\}$

21.8. tétel. *TökéletesHJMegegyezés, ha egy tökéletes hibajelzővel használjuk, megoldja a megegyezési feladatot és biztosítja a várakozásmentes befejezést.*

Bizonyítás. A jólformáltságot és az érvényességet könnyű látni. A várakozásmentes befejezéshez tekintsünk egy olyan α helyes végrehajtási sorozatot, ahol minden kapunál előfordul `kezd` esemény, és legyen i egy hibamentes kapu; meg-

mutatjuk, hogy P_i már α során választ. Megjegyezzük, hogy minden alkalommal, amikor P_i ($\mathit{érték}_i, \mathit{megállított}_i$) adatai változnak α-ban, az új párnak dominálnia kell a régit. Mivel csak véges sok lehetséges pár van, ezek az adatok végül egy végső $(w_{\text{vég}}, I_{\text{vég}})$ értékre állnak be. Ha P_i ezen pont előtt választ, akkor készen vagyunk, így tegyük fel, hogy ez nem áll fenn. Ekkor azt állítjuk, hogy ennek az értéknek az elérése után $\mathit{megerősített}_i \cup \mathit{megállított}_i = \{1, \ldots, n\}$, ami elegendő ahhoz, hogy egy $\texttt{választ}_i$ esemény előforduljon. Ezen állítás igazolásához elég megmutatni, hogy minden $j \neq i$ folyamat, amely sohasem hibázik, megerősíti ezt a $(w_{\text{vég}}, I_{\text{vég}})$ párt.

Tekintsünk így egy tetszőleges $j \neq i$ folyamatot, amely sohasem hibázik. Lesz egy olyan $(w_{\text{vég}}, I_{\text{vég}})$-t tartalmazó üzenet, amelyet P_i sugároz és P_j fogad, és amely után a P_j-hez tartozó pár mindig dominálja $(w_{\text{vég}}, I_{\text{vég}})$-t. Azonban a P_j-hez tartozó pár sosem dominálhatja *szigorúan* $(w_{\text{vég}}, I_{\text{vég}})$-et, mivel ha így lenne, akkor P_j-nek sikerülne közölni ezt az új információt P_i-vel. Így végül a P_j-hez tartozó pár egyenlő lesz $(w_{\text{vég}}, I_{\text{vég}})$-vel, és így is marad a továbbiakban. Ezután lesz egy olyan $(w_{\text{vég}}, I_{\text{vég}})$-t tartalmazó üzenet, amelyet P_j sugároz és P_i fogad. Ez $\mathit{megerősített}_i$-be teszi j-t, ahogy kellett.

Végül a megegyezést bizonyítjuk. Tegyük fel, hogy P_i az első olyan folyamat, amely választ, és legyenek egyenként w és I az $\mathit{érték}_i$ és $\mathit{megállított}_i$ értékek, amikor a $\texttt{választ}_i$ esemény, jelöljük π-vel, bekövetkezik. Ekkor az összes I-beli folyamat π-nél korábban hibázik α-ban, és így sohasem választ. Legyen $J = \{1, \ldots, n\} \setminus I$; belátjuk, hogy az összes J-beli folyamatnak, amely választ, ugyanazt az értéket kell választania, mint amit P_i választ.

Mindegyik J-beli folyamatnak $\mathit{megerősített}_i$-ben kell lennie, amikor π bekövetkezik, így a lokális adata szükségképpen (w, I) egy π előtti t_j időpontban. Azt állítjuk, hogy minden J-beli j folyamatnak örökre meg kell tartania az $\mathit{érték}{=}\, w$ értéket a t_j időpont után, amiből következik, hogy ha választ, akkor megegyezik P_i-vel.

Tegyük így fel, hogy nem ez a helyzet, és legyen j az az első J-beli folyamat, amely olyan információra tesz szert az *érték* vektorában, amely nincs benne w-ben (azaz a vektor egy eleme V-beli, míg w megfelelő eleme *nulla*). Ekkor ez egy olyan t_j után bekövetkező $\texttt{fogad}_{k,j}$ esemény eredménye kell legyen, ahol az üzenetszóró P_i folyamat *érték* vektorában lévő információ a sugárzáskor nincs benne w-ben. Mivel P_j az összes I-beli folyamatot figyelmen kívül hagyja t_j után, az üzenetszóró P_k folyamatnak J-ben kell lenni. Ez azonban ellentmond annak a választásnak, hogy j az első olyan folyamat J-ben, amely w-n kívüli információt szerez. □

Bonyolultság.. A TökéletesHJMegegyezés algoritmus kommunikációs bonyolultsága és időbonyolultsága nemkorlátos. Ez nem olyan nagy baj, mivel ebben a fejezetben csak kiszámíthatósági kérdésekkel foglalkozunk. Lehetséges azonban hasonló protokollt szerkeszteni korlátos bonyolultsággal. Ezt meghagyjuk gyakorlatnak (lásd 21-10. gyakorlat).

21.5.. k-megegyezés

Most a feladat megfogalmazásának egy gyengítését tekintjük. A *k-megegyezési feladat*, ahogy a 7.1. és 12.5. alfejezetekben leírtuk szinkron hálózati környezetben és aszinkron közös memóriájú környezetben, egy olyan változata a megegyezési feladatnak, amely megoldható aszinkron hálózatokban korlátozott számú ($f < k$) hibával. A feladatot ugyanúgy definiáljuk, mint a 12.5. fejezetben: azaz a feladat jólformáltsági és befejezési feltételei legyenek ugyanazok, mint a közönséges megegyezési feladaté, és a megegyezési és érvényességi feltételeket helyettesítsük az alábbiakkal, ahol $k \geq 1$ egy egész.

Megegyezés. Bármely végrehajtási sorozatnál van olyan W részhalmaza V-nek, hogy $|W| = k$ és minden döntési érték W-beli.

Érvényesség. Bármely végrehajtási sorozatnál minden folyamat minden döntési értéke egy folyamat kezdeti értéke.

A megegyezési feltétel abban gyengébb a közönséges megegyezésnél, hogy 1 helyett megenged k számú döntési értéket. Az érvényességi feltétel kis élesítése a közönséges érvényességi feltételnek. Van egy nyilvánvaló algoritmus a k-megegyezési feladat megoldására aszinkron üzenetszóró hálózatban amikor $f < k$.

EGYSZERŰkMEGEGYEZÉS algoritmus

A $P_1, P_2, \ldots, P_k$ folyamatok csak a saját kezdeti értékeiket sugározzák. Minden P_i folyamat az első kapott értéket választja.

21.9. tétel. . EGYSZERŰkMEGEGYEZÉS *megoldja a k-megegyezési feladatot és biztosítja az f-hibás befejezést, ha $f < k$.*

Nem nehéz egy stabil vektorokon alapuló k-megegyezési algoritmust szerkeszteni TÖKÉLETESHJMEGEGYEZÉS mintájára. Ezt gyakorlatnak hagyjuk (lásd 21-13. gyakorlat). Alternatív módon, a 17.5. tételt használva, aszinkron közös memóriájú modellre ismert algoritmusokból nyerhetünk k-megegyezési algoritmusokat aszinkron hálózati modellre, átültetve őket közös memóriájú modellről hálózati modellre. Azonban ennek a megközelítésnek az a hátránya, hogy csak $n > 2f$ esetében működik, míg EGYSZERŰkMEGEGYEZÉS és a stabil vektorokon alapuló algoritmus akkor is működik ha $n \leq 2f$.

Kiderül, hogy a k-megegyezési feladat nem oldható meg, ha a hibák száma nagyobb vagy egyenlő, mint k.

21.10. tétel. . *A k-megegyezési feladat nem oldható meg k-hibás befejezéssel az aszinkron üzenetszóró modellben.*

Bizonyítás. A 12.13. és 17.8. tételek alapján. □

21.6.. Közelítő megegyezés

Ismét gyengítjük a feladat megfogalmazását. A megegyezési feladat egy másik változata, mint a 7.2. és 12.5. alfejezetben leírtuk szinkron hálózati környezetben és aszinkron közös memóriájú környezetben, a *közelítő megegyezés feladata*. A feladatot ugyanúgy definiáljuk mint a 12.5. fejezetben. Azaz az értékek V halmaza a valós számok egy halmaza, és megengedjük, hogy a folyamatok valós értékű adatokat küldjenek az üzenetekben. A megegyezési feladatbeli pontos egyetértés helyett azt követeljük meg, hogy a folyamatok egyezzenek meg egy kis pozitív ε tűrésen belül. A feladat jólformáltsági és befejezési feltételei legyenek ugyanazok, mint a közönséges megegyezési feladaté, a megegyezési és érvényességi feltételeket helyettesítsük az alábbiakkal.

Megegyezés. Bármely végrehajtási sorozatnál bármely két döntési érték ε-on belül van egymástól.

Érvényesség. Bármely végrehajtási sorozatnál minden döntési érték a kezdeti értékek tartományába esik.

Egy, a 7.2. alfejezetbeli KonvKözelMegegyez algoritmushoz hasonló algoritmus működik megállási hibákkal terhelt aszinkron környezetben ha $n > 3f$. Minden P_i folyamat szintek egy sorozatát hajtja végre, minden szinten addig vár, amíg az összes n folyamat helyett bármely $n - f$ darab folyamattól üzenetet nem kap. (Nem várhat addig, amíg az összes folyamattól üzenetet kap, hiszen az utolsó f folyamat megállíthatja.) Mivel csak megállási hibákat tekintünk, nem szükséges, hogy extrém értékek figyelmen kívül hagyásával „csökkentsük” P_i értékeinek halmazát. A következő leírásban használt *átlag* és *kiválaszt* függvényeket és néhány más jelölést, mint például valós számok egy halmazának *szélességét*, a 7.2. fejezetben definiáltuk.

AszinkKözelMegegyez algoritmus

Feltesszük, hogy $n > 3f$. Minden P_i karbantart egy *érték* változót, amely a legutolsó becslést tartalmazza. Ezt kezdetben azzal a v értékkel tölti fel, amely a `kezd`$(v)_i$ bemenettel érkezik. P_i a következőt csinálja az összes szinten. Először sugározza az *érték* változója értékét, amelyet a szint s számával címkéz meg. Ezután összegyűjti a W halmazba azt az első $n - f$ értéket, amelyet az s szint számára kapott. Végül az *érték* változónak az *átlag*(*kiválaszt*(W)) értéket adja.

Nyilvánvaló kell legyen, hogy minden s szinten bármelyik folyamat által választott *érték* az összes, $s - 1$ szintbeli folyamat által választott *érték*-ek tartományába esik (vagy a kezdeti értékek tartományába ha $s = 1$). Azt állítjuk, hogy minden szinten az *érték*-ek halmazának szélessége legalább $\lfloor \frac{n-f-1}{f} \rfloor + 1$ tényezővel csökken. Mivel $n > 3f$, ez adja a konvergenciát.

21.11. lemma. . *Legyenek v és v' a P_i és $P_{i'}$ folyamatok által választott érték$_i$ és érték$_{i'}$ értékei az s szinten az* AszinkKözelMegegyez *egy végrehajtási sorozata során. Ekkor*

$$|v - v'| \leq \frac{d}{\lfloor \frac{n-f-1}{f} \rfloor + 1},$$

ahol d az $s-1$ szinten választott érték-ek halmaza tartományának a szélessége ha $s \geq 2$, és a kezdeti értékek szélessége ha $s = 1$.

Bizonyítás. A 7.17. lemma mintájára. □

Befejeződés.. Minden amit eddig mondtunk AszinkKözelMegegyez-ről ($n > 3f$ helyett), az $n > 2f$ feltétel mellett is működik. Azonban még nem kaptunk teljes algoritmust, mivel nem mondtuk meg, hogy a folyamatok mikor választanak. A befejeződés elérésének segítésére külön folyamatokat használunk.

Nem használhatjuk azt az egyszerű befejezési eljárást, amelyet KonvKözelMegegyez számára használtunk, mert egy folyamat nem várhat addig, amíg az összes folyamattól üzenetet nem kap az 1 szinten, és így nem mindig határozható meg egy felső korlát a kezdeti értékek halmazának a tartományára. Azonban kissé módosíthatjuk ezt az eljárást, hozzávéve az algoritmus elejéhez egy speciális kezdeti értékbeállítás szintet, a 0 szintet. A 0 szinten az összes P_i folyamat sugározza *érték* változóját, összegyűjti $n-f$ számú *érték* egy halmazát, és ennek a halmaznak a *medián*-ját választja új *érték*-nek az 1 szinten. Mivel $n > 3f$, könnyen ellenőrizhető, hogy bármelyik P_i folyamat által a 0 szinten választott bármelyik *érték* beleesik bármelyik P_j folyamatnak a 0 szinten összegyűjtött halmazának a tartományába. Így az összes P_i használhatja annak a halmaznak a tartományát, amelyet a 0 szinten gyűjtött össze arra, hogy kiszámolja azt a szint számot, amelyre biztos, hogy bármely két, s szintbeli folyamat *érték* vektorának értékei legfeljebb ε-ra vannak egymástól. Az eljárás hátralévő része ugyanaz, mint KonvKözelMegegyez-re.

Az AszinkKözelMegegyez algoritmus nem optimális abban az értelemben, hogy valójában a feladat akármilyen $n > 2f$ esetében megoldható. Ehhez azonban egy összetettebb algoritmus szükséges. Például $n > 2f$ esetére működő algoritmus kapható olyan kizárólagosan-írható/megosztottan-olvasható osztott regisztereken alapuló A közös memóriájú közelítő megegyezési algoritmusból, amely biztosítja a várakozásmentes befejezést. A 12.14. alfejezet azt állítja, hogy van ilyen A algoritmus (és egy ilyet találhat az olvasó [24]-ben). Ezután a 17.5. tételt használhatjuk arra, hogy olyan aszinkron hálózati algoritmus létezésére következtessünk, amely megoldja a közelítő megegyezési feladatot, és biztosítja az f-hibás befejezést amennyiben $n > 2f$.[1] Másrészt nem nehéz látni, hogy a közelítő megegyezési feladat nem oldható meg, ha $n \leq 2f$.

21.12. tétel. . *A közelítő megegyezési feladat nem oldható meg f-hibás befejezéssel az aszinkron üzenetszóró modellben, ha $n \leq 2f$.*

[1] Ahhoz, hogy alkalmazhassuk a 17.5. tételt, A-nak eleget kell tenni a 17.1.1. szakaszbeli „fordulat” megszorításnak. A közös memóriájú közelítő megegyezési algoritmus megkonstruálható úgy, hogy kielégítse ezt a feltételt.

Bizonyításvázlat. A bizonyítás a 17.6. tétel bizonyításához hasonló. Röviden, feltételezzük, hogy van egy ilyen algoritmus, és legyen G_1 az $1, \ldots, n-f$ számok, G_2 az $n-f+1, \ldots, n$ számok halmaza. Tekintsünk egy olyan α_1 helyes végrehajtási sorozatot, amelyben az összes folyamat egy v_1 értékkel kezd, és az összes G_2-beli indexű folyamat hibázik induláskor. Az f-hibás befejezés miatt az összes G_1-beli folyamatnak választania kell, és az érvényességi feltételből következik, hogy v_1-t kell választaniuk. Szimmetrikusan, tekintsünk egy olyan α_2 második helyes végrehajtási sorozatot, amelyben az összes folyamat egy v_2 értékkel kezd, ahol $|v_1 - v_2| > \varepsilon$, és az összes G_1-beli folyamat hibázik induláskor. α_2-ben az összes G_2-beli folyamatnak végül v_2-t kell választani.

Ezután α_1 és α_2 kombinálása révén egy α véges végrehajtási sorozatot konstruálunk úgy, mint a 17.6. tétel bizonyításában. Az α végrehajtási sorozat során a G_1-beli folyamatok v_1-t, a G_2-beliek pedig v_2-t választják, ami ellentmond a megegyezési feltételnek. □

21.7.. Kiszámíthatóság aszinkron hálózatokban*

Ugyanaz a konstrukció, amelyet a 17.6. és 21.12. tétel bizonyításához használtunk, alkalmazható annak megmutatására, hogy az általános összehangolás sok más problémája sem oldható meg aszinkron hálózatokban, ha a folyamatok fele hibázhat.

Ahogy a 12.5. alfejezetben, tekinthetjük tetszőleges döntési feladatok megoldhatóságát aszinkron hálózatokban. A közönséges megegyezés, a k-megegyezés és a közelítő megegyezés mind egy példa döntési feladatra, és a fő eredményeket már megadtuk az ilyen feladatok aszinkron hálózatokban való kiszámíthatóságáról. Mint az olvasható/írható közös memóriájú modellnél, egy tételt mondunk ki, ami olyan feltételeket ad, amelyekből következik, hogy egy feladat nem oldható meg 1-hibás befejezéssel aszinkron hálózati modellben.

21.13. tétel. . *Legyen D egy olyan döntési leképezés, amely döntési feladata megoldható 1-hiba befejezéssel az aszinkron üzenetszóró modellben. Ekkor kell lenni egy olyan D' döntési leképezésnek, amelyre $D'(w) \subseteq D(w)$ minden w-re, és fennáll a következők mindegyike:*

1. *Ha a w és w' bemeneti vektorok pontosan egy állapotban különböznek, akkor vannak olyan $y \in D'(w)$ és $y' \in D'(w')$ vektorok, hogy y és y' legfeljebb egy állapotban különbözik.*
2. *Minden w-re a $D'(w)$ által meghatározott gráf összefüggő.*

Bizonyítás. A 12.15. és 17.8. tételek alapján. □

Általában a kiszámíthatóságra vonatkozó megoldhatalansági eredmények olvasható/írható közös memóriájú környezetből kiterjeszthetőek a 17.8. tételben használt hálózati környezetre. Az algoritmusok, a 17.5. tételt használva, szintén kiterjeszthetőek, azonban csak a 17.5. tétel által igényelt megszorítások mellett, beleértve az $n > 2f$ követelményt is.

21.8.. Megjegyzések a fejezethez

A 21.2. tételt, a megegyezés megoldhatatlanságát megállási hibák jelenlétekor, először Fischer, Lynch és Paterson [123] bizonyította. Az eredeti bizonyítás nem átalakítás útján, hanem közvetlenül aszinkron üzenetszóró modellben lett megadva. Loui és Abu-Amara [199] vette észre, hogy a 21.2. tétel kiterjeszthető, lényegében ugyanazt a bizonyítást használva, olvasható/írható közös memóriájú modellekre. A 12.8. tételre adott bizonyításunk Loui és Abu-Amara tárgyalását követi. Fischer, Lynch Paterson eredeti bizonyítását, Bridgland és Watro [58] javaslatai szerint, a 21.2., 21.3. és 21.4. gyakorlatokban vázoljuk.

A BenOr algoritmust Ben-Or [46] találta ki. Rabin [248] és Feldman [114] későbbi munkáikban más, jobb (valójában konstans) időbonyolultságú véletlenített algoritmusokat fejlesztettek ki. Ezek a „titkos megosztás" módszerét használják arra, hogy növeljék a valószínűségét annak, hogy a különböző folyamatok által választott véletlen értékek megegyezzenek ugyanazon a szinten.

A hibajelző fogalmát Chandra és Toueg [66] és Chandra, Hadzilacos és Toueg [65] definiálta és fejlesztette ki. Ezek a cikkek nem csak az itt tárgyalt tökéletes hibajelzőt írják le, hanem sok kevésbé tökéletes változatot is, beleértve azokat a hibajelzőket is, amelyek tévesen azonosítanak hibásnak folyamatokat, és azokat, amelyek nem értesítik az összes folyamatot a hibákról. Az ilyen gyengébb hibajelzőkkel is megoldható a megegyezési feladat, és néhány alkalmazható valóságos osztott rendszerekben időtúllépés használatával. Hadzilacos és Toueg [143] szintén tárgyalja a hibajelzőket.

A 7. és 12. fejezetekhez fűzött megjegyzésekben már megtárgyaltuk a k-megegyezési feladat és a közelítő megegyezési feladat eredetét. Attiya, Bar-Noy, Dolev, Koller, Peleg és Reischuk [19,20,40] ismertet néhány más érdekes feladatot, amelyek megoldhatók hibával terhelt aszinkron hálózatokban, beleértve a folyamat átnevezési feladatot és a rekeszes kizárási feladatot. Bridgland és Watro [58] egy olyan erőforrás-hozzárendelési feladatot ír le, amely megoldható hibával terhelt aszinkron hálózatokban. A stabil vektor algoritmus ötlete Attiyától és társaitól [20] származik.

A 21.2. tétel bizonyítását Bracha és Toueg [56] és Attiya, Bar-Noy, Dolev, Peleg és Reischuk [20] bizonyításainak átdolgozásával kaptuk. Biran, Moran és Zaks [51], Moran és Wolfstahl [230] egy korábbi megoldhatatlansági eredményére támaszkodva, jellemezte azokat a választási feladatokat, amelyek megoldhatók 1-hibás befejezésű aszinkron hálózattal. A 21.13. tétel ennek a két cikknek az átdolgozása.

21.9.. Gyakorlatok

21-1. Bizonyítsuk be a 21.1. tételt.

21-2. Tegyük fel, hogy $V = \{0, 1\}$. Ha A egy olyan aszinkron üzenetszóró rendszer, amely megoldja a megegyezési feladatot, akkor definiáljuk a 0-értékűséget, 1-értékűséget, egyértékűséget és kétértékűséget A véges végrehajtási sorozataira, továbbá A kezdeti értékbeállítását ugyanúgy mint a 12.2.2. szakaszban.

(a) Adjunk példát olyan A rendszerre, amelyben van kétértékű kezdeti értékbeállítás.
(b) Adjunk példát olyan A rendszerre, amelyben minden kezdeti értékbeállítás egyértékű.
(c) Bizonyítsuk be, hogy ha A biztosítja az 1-hibás befejezést, akkor van kétértékű kezdeti értékbeállítás.

21-3. Legyen V, A ugyanaz mint a 21-2. gyakorlatban. Definiáljunk egy α *választó* végrehajtási sorozatot egy olyan véges hibamentes bemenettel-kezdő végrehajtási sorozatként, amely kielégíti a következő feltételeket. Valamilyen i-re
(a) α kétértékű;
(b) α-nak van olyan α_0 0-értékű hibamentes kiterjesztése, hogy α_0-nak az α utáni része csak P_i lépéseiből áll;
(c) α-nak van olyan α_1 1-értékű hibamentes kiterjesztése, hogy α_1-nek az α utáni része csak P_i lépéseiből áll.

Azaz egy önálló P_i folyamat kétféleképpen hathat saját magára (például, kétféle úton fésüli össze a helyben felügyelt és üzenet-fogadó lépéseket, vagy pedig üzeneteknek két különböző sorozatát fogadja), oly módon, ahogy a végső választást oldja meg két különböző módon.

Bizonyítsuk be, hogy ha A-nak van kétértékű kezdeti értékbeállítása, akkor A-nak van választó végrehajtási sorozata. Megjegyezzük, hogy csak azt tételeztük fel, hogy A megoldja a megegyezési feladatot; nem tettünk feltételt a hibatűrésre. (*Útmutatás.* Tekintsük a 12.7. lemma bizonyítását.)

21-4. A 21-2. és 21-3. gyakorlatok eredményeinek felhasználásával bizonyítsuk be a 21.2. tételt.

21-5. Tekintsük ismét, az üzenetszóró modellt használva, a fejezet megegyezési feladatát. Ezúttal egy, a megszokott megállási hibáknál korlátozottabb hiba modellt tekintünk, amelyben a folyamatok csak a számítások kezdetekor hibázhatnak. (Azaz az összes `megállít` esemény megelőzi az összes többi eseményt.) Megoldható-e ebben a modellben a megegyezési feladat úgy, hogy biztosítva legyen
(a) az 1-hibás befejezés;
(b) a várakozásmentes befejezés?

Adjunk mindegyik esetben vagy algoritmust, vagy bizonyítást a megoldhatatlanságra.

21-6. Tervezzünk olyan változatot a BenOr algoritmushoz, amelyben végül minden hibamentes folyamat megáll.

21-7. Tervezzünk olyan változatokat a BenOr algoritmushoz, amelyek működnek a következő esetekre:
(a) szinkron hálózati modell megállási hibákkal;
(b) szinkron hálózati modell bizánci hibákkal;
(c) aszinkron hálózati modell bizánci hibákkal (ahogy a 14.1.1. szakaszban utaltunk rá, egy P_i folyamat bizánci hibája modellezhető úgy, hogy P_i-t egy tetszőleges, ugyanolyan külső felülettel rendelkező b/k automatával helyettesítjük).

Próbáljunk olyan algoritmust tervezni mindegyik esetben, amely a megengedett hibák f számához képest a lehető legkevesebb folyamattal működik.

21-8. Tervezzünk olyan véletlenített szinkron algoritmust a megegyezésre megállási hibák esetében, amely a $\{0,1\}$ helyett tetszőleges V értékhalmazt használ. Próbáljuk minimalizálni a folyamatok számát. (*Útmutatás.* Kombináljuk a TurpinCoan algoritmus ötletét a BenOr algoritmuséval.)

21-9. Ismételjük meg a 21-8. feladatot bizánci hibák esetére.

21-10. Szerkesszünk egy olyan alternatív protokollt TökéletesHJMegegyezés-hez, amely szintén tökéletes hibajelzőt használ a várakozásmentes megegyezés megvalósítására, azonban „kisebb" kommunikációs- és időbonyolultsággal bír. Próbáljuk meghatározni azt a legkisebb kommunikációs- és időbonyolultságot, amelyet tudunk.

21-11. Definiáljunk egy *hibás hibajelzőt* a következőképp. A külső felülete egyezzék meg egy tökéletes hibajelzőével hozzáadva még egy $\texttt{inf\_nem\_állít\_meg}(j)_i$ műveletet minden j és i, $j \neq i$, esetében. Ezt arra használjuk, hogy kijavítsunk egy korábbi $\texttt{inf\_megállít}(j)_i$ műveletet, azaz értesítsük a P_i folyamatot arról, hogy egy előző téves értesítés ellenére P_j valójában nem állt meg. Egy hibás hibajelző akárhányszor váltogathatja a $\texttt{inf\_megállít}(j)_i$ és $\texttt{inf\_nem\_állít\_meg}(j)_i$ eseményeket. Azonban a hibajelző bármely helyes α végrehajtási sorozata esetében csak véges sok ilyen esemény lehet minden i-re és j-re, és az utolsó ilyen eseménynek helyes információt kell tartalmaznia – megmondva, hogy a $\texttt{megállít}_j$ esemény előfordul-e vagy sem.

Tegyük fel, hogy $n > 2f$. Szerkesszünk olyan algoritmust, amely hibás hibajelzőt használva megoldja a megegyezési feladatot, biztosítva az f-hibás befejezést.

21-12. Bizonyítsuk be, hogy $n \leq 2f$ esetében nincs olyan algoritmus, amely tetszőleges, a 21.11. gyakorlatban definiált hibás hibajelzőt használva megoldja a megegyezési feladatot, biztosítva az f-hibás befejezést.

21-13. Adjunk a k-megegyezési feladat megoldására TökéletesHJMegegyezés-hez hasonló előfeltétel-hatás kódot a „stabil vektor" algoritmushoz. Bizonyítsuk be, hogy a kód helyesen működik, ha $f < k$. (*Útmutatás.* Az állapotok csak az *érték*, *megerősített* és *választott* komponenseket tartalmazzák, a *megállt* komponenst nem. A választás elvégezhető, amikor $|\textit{megerősített}| \geq n - f$.)

21-14. Egy k-megegyezési algoritmus α véges végrehajtási sorozatát m-értékűnek nevezzük, ha pontosan m számú különböző döntési érték van, amely α kiterjesztéseiben előfordul. Definiáljuk a kezdeti értékbeállítást úgy, mint a 12.2.2. szakaszban. Bizonyítsuk be (a 21.10. tétel használata nélkül), hogy minden k-megegyezési algoritmusnak az aszinkron üzenetszóró modellben, amely biztosítja a k-hibás befejezést, van $(k+1)$-értékű kezdeti végrehajtási sorozata. (*Útmutatás.* Használjuk a 7.1. alfejezet ötleteit, többek között a Sperner-lemmát.)

21-15. Adjunk teljes előfeltétel-hatás kódot, beleértve a befejezési protokollt, az AszinkKözelMegegyez algoritmushoz. Bizonyítsuk a helyességet.

21-16. Módosítsuk az AszinkKözelMegegyez algoritmust és bizonyítását úgy, hogy működjön bizánci hibák esetére. Hány folyamat szükséges? (*Útmutatás.* Használjuk a KonvKözelMegegyez algoritmus ötleteit a szinkron bizánci környezetre.)

21-17. Bizonyítsuk azt a legáltalánosabb megoldhatatlansági eredményt, amelyet tudunk, a 21.12. tétel bizonyításában használt konstrukció alapján.

21-18. Adjunk általános jellemzést azokra a *döntési feladatokra*, amelyek megoldhatók 1-hibás befejezéssel aszinkron hálózatokban. (*Figyelmeztetés.* Nagyon nehéz.)

22. fejezet

Adatkapcsolat protokollok

A kommunikációs hálózatok egyik alapvető feladata, hogy nem tökéletesen megbízható csatornák igénybevételével megbízható FIFO kommunikációt biztosítsanak. Egy „nem tökéletesen megbízható" csatornán üzenetek veszhetnek el vagy megérkezhetnek több példányban is, az elindított üzenetek sorrendje felcserélődhet, illetve egy folyamat katasztrófája az állapotinformációk elvesztéséhez vezethet.

A fejezet algoritmusai kétrésztvevős hálózatok esetében biztosítanak megbízható kommunikációt. Bemutatunk két közismert egyszerű protokollt, a STENNING protokollt és a BITVÁLTÓ protokollt. A STENNING protokollban a küldő oldali folyamat (méretkorlátozás nélküli) egész címkékkel látja el a felhasználó által küldött üzeneteket; ezzel a módszerrel javíthatóvá válik az üzenetek elveszése, többszörözése és sorrendjük felcserélése. A BITVÁLTÓ protokoll ezzel szemben korlátozott méretű címkékkel látja el az üzeneteket, így a protokoll helyes marad üzenetvesztés és többszörözés esetében, de a sorrend változása hibás működéshez vezethet.

A protokollok leírásában a kommunikáció két szintjét különböztetjük meg: a felhasználói szintet és a csatorna szintjét. A különböző szinteken küldött üzenetek megkülönböztetésére a *magas*, illetve *alacsony szintű* elnevezést használjuk. Általában a magas és alacsony szintű ábécé az M, illetve az M' lesz. Hasonlóképpen megkülönböztetjük a felhasználói felület és a csatornafelület műveleteit: az előbbieket nagybetűvel (`KÜLD`, `FOGAD`), az utóbbiakat kisbetűvel (`küld`, `fogad`) jelöljük.

A fejezetben használt módszerek (b/k automata, összekapcsolás, szimulációs reláció) sokféle rétegzett kommunikációs rendszer, így az ISO hierarchia modellezésére is alkalmasak.

22.1.. A feladat

Megbízható FIFO kommunikációt olyan aszinkron kétirányú hálózaton kívánunk megvalósítani, amelyet az 1-gyel és 2-vel jelölt csúcsokat egyetlen irányítatlan éllel összekötő kétcsúcsú G gráf modellez. A kommunikáció a csúcsoknál elhelyezkedő

U_1 és U_2 felhasználók között folyik; az U_1 által a P_1 folyamatnak átadott magas szintű üzeneteknek sorrendben el kell U_2-höz érkezniük. Az U_2-nek minden üzenetet pontosan egyszer kell megkapnia, és az üzeneteknek az elküldés sorrendjében kell megérkezniük.

A problémát a 14.1.2. szakaszban és a 8.1.1. példában bemutatott F univerzális megbízható kétirányú FIFO csatornával modellezhetjük, amely jelen esetben 1 és 2 között az M ábécé feletti üzeneteket továbbítja és külső műveletei $m \in M$ esetében $\texttt{KÜLD}(m)_{1,2}$ és $\texttt{FOGAD}(m)_{1,2}$. Feladatunk tehát egy F-et „megvalósító" protokoll létrehozása, azaz egy olyan protokollé, amelynek minden pártatlan α végrehajtási sorozata az F protokoll külső műveleteire vetítve az F pártatlan történetét adja. A b/k automaták 8. fejezetben bevezetett formális jelölésével megfogalmazva tehát a követelmény $\alpha|\mathit{külső}(F) \in \mathit{pártatlan\_történet}(F)$.

Mivel az F univerzális megbízható FIFO csatorna lényegében egy korlátlan méretű sor, elvileg F minden megvalósítása végtelen tárat igényel. Ezt a problémát kiküszöbölheti az a modell, amelyben U_1 és a csatorna között kétirányú kapcsolat van, azaz a csatorna megmondhatja U_1-nek, mikor küldheti a következő magas szintű üzenetet. A végtelen tárméret kiküszöbölésének ugyanakkor az az ára, hogy a csatorna és a felhasználó közötti kommunikáció nagyon bonyolulttá teszi a protokoll modelljét, ezért ezzel a kérdéssel a továbbiakban nem foglalkozunk.

Mind az F-et megvalósító két folyamatot, mind a csatorna két irányát b/k automatákkal modellezzük. A két csatornaautomata nem szükségszerűen teljesíti a megbízható FIFO kommunikáció követelményeit: az üzenetek elvesztése, többszörözése és a sorrend megváltoztatása megengedett lehet.

Modellünkben nem engedjük meg a csatorna bizonyos típusú meghibásodásait. Nem engedjük meg, hogy a csatorna hamis üzeneteket hozzon létre. Korlátozzuk az üzenetvesztés mértékét is: általában megköveteljük a csatorna élénkségét, amelynek egy lehetséges megfogalmazása a következő.

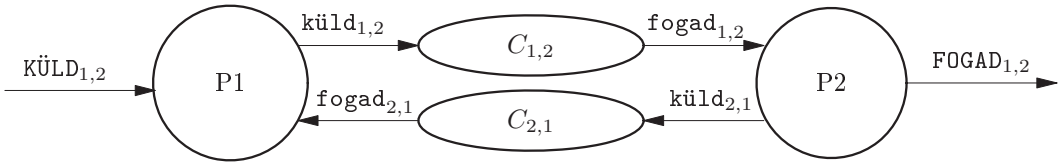
> Ha végtelen sok üzenetet küldünk, akkor ezek közül végtelen sok meg is érkezik.

A 14.1.2. szakaszban a fenti tulajdonság kétféle formális definícióját láttuk, a gyenge és az erős veszteségkorlátozást (GyVK és EVK). A kettő közötti különbség úgy fogalmazható meg, hogy a GyVK tulajdonsággal bíró csatorna minden egyes üzenettípust előbb-utóbb továbbít. Ebben a fejezetben szükség szerint mindkét feltételt használni fogjuk. Gyakran szükségünk lesz ezen felül arra is, hogy az üzenetek többszöröződését korlátozzuk.

A következőkben ismertetendő protokollok legtöbbje a 14.1.2. szakaszban formálisan definiált valamelyik megengedett csatorna-viselkedés esetében működik. A csatornaviselkedések leírására részben a b/k automaták, részben a `küld` és `fogad` események közötti *ok* függvényre adott axiómák, részben pedig az automata megadását kiegészítő élénkségi feltételek alkalmasak. A fejezetben mindhárom definíció-típust használni fogjuk.

A kommunikációt a 22.1. ábrán látható módon valósítjuk meg, amely a P_1 és P_2 folyamat-automatából, illetve a $C_{1,2}$ és $C_{2,1}$ csatorna-automatából áll. A folyamat-automaták a felhasználóval a `KÜLD` és `FOGAD` műveleteken, a csatorna

automatákkal pedig a `küld` és `fogad` műveleteken keresztül állnak kapcsolatban. A 22.5. alfejezetben további műveleteket kell bevezetnünk, hogy a folyamatok katasztrófáit is modellezhessük.



22.1.. ábra. Az adatkapcsolat protokollok megvalósítása.

22.2.. A STENNING protokoll

A legegyszerűbb, nem tökéletesen megbízható csatornán megbízható FIFO kommunikációt megvalósító protokoll N. Stenningtől származik. A protokoll háromféle csatornahibát javít: (korlátozott) üzenetvesztést, az üzenetek (véges) többszöröződését és az üzenetek sorrendjének átrendezését.

A Stenning protokoll (vázlatosan)

A P_1 folyamat az U_1 felhasználó által küldött magas szintű üzeneteket eggyel kezdődően folytatólagosan egész számokkal címkézi és a $puffer_1$-ben tárolja; innen az első üzenetet és címkéjét ismételten P_2-nek továbbítja. A P_2 elfogad minden olyan üzenetet, amelynek címkéje eggyel nagyobb, mint az előzőleg elfogadott üzenet címkéje. Az elfogadott üzenetek sorrendben a $puffer_2$-be kerülnek, ahonnan P_2 sorrendben U_2-nek továbbítja azokat.

Az elfogadott magas szintű üzeneteket P_2 azzal nyugtázza, hogy címkéjüket visszaküldi P_1-nek. Amikor P_1-hez megérkezik a $puffer_1$-ben található első üzenetet címkéje, továbblép és a pufferben található következő üzenetet küldi P_2-nek.

22.1. protokoll. Stenning

A P_1 folyamat
Lenyomat:
Bemeneti:
$\texttt{KÜLD}(m)_{1,2}$, $m \in M$
$\texttt{fogad}(k)_{2,1}$, $k \in \mathbb{N}$
Kimeneti:
$\texttt{küld}(m,k)_{1,2}$, $m \in M$, $k \in \mathbb{N}$

Állapotok:
$puffer_1$, egy M elemeiből álló FIFO sor, kezdetben üres
$címke_1 \in \mathbb{N}$, kezdőértéke 1

A P_2 folyamat
Lenyomat:
Bemeneti:
$\texttt{fogad}(m,k)_{1,2}$, $m \in M$, $k \in \mathbb{N}$
Kimeneti:
$\texttt{FOGAD}(m)_{1,2}$, $m \in M$
$\texttt{küld}(k)_{1,2}$, $k \in \mathbb{N}$

Állapotok:
$puffer_2$, egy M elemeiből álló FIFO sor, kezdetben üres
$címke_2 \in \mathbb{N}$, kezdőértéke 0

Átmenetek:
KÜLD$(m)_{1,2}$
Hatás:
tegyük be m-et a *puffer*$_1$-be

küld$(m,k)_{2,1}$
Előfeltétel:
m a *puffer*$_1$ első eleme
$k = címke_1$
Hatás:
üres

fogad$(k)_{2,1}$
Hatás:
if $k = címke_1$ **then**
távolítsuk el *puffer*$_1$ első elemét (ha van ilyen)
$címke_1 := címke_1 + 1$

Taszkok:
{küld$(m,k)_{1,2} : m \in M, k \in \mathbb{N}$}

Átmenetek:
FOGAD$(m)_{1,2}$
Előfeltétel:
m *puffer*$_2$ első eleme
Hatás:
távolítsuk el *puffer*$_2$ első elemét

fogad$(m,k)_{1,2}$
Hatás:
if $k = címke_2 + 1$ **then**
tegyük be m-et *puffer*$_2$-be
$címke_2 := címke_2 + 1$

küld$(k)_{2,1}$
Előfeltétel:
$k = címke_2$
Hatás:
üres

Taszkok:
{FOGAD$(m)_{1,2} : m \in M$}
{küld$(k)_{2,1} : k \in \mathbb{N}$}

A STENNING protokoll az egymással a $C_{1,2}$ és $C_{2,1}$ csatornákon kommunikáló P_1 és P_2 folyamatokból áll, a teljes protokoll ezek összekapcsolásából nyerhető. A protokollban a $C_{1,2}$ és $C_{2,1}$ csatornáknak a műveletek megfelelő átnevezésével a 14.1.2. példában leírt *veszteséges átrendező csatornák* követelményeit kell teljesíteniük. A csatornákon tehát korlátozott üzenetvesztés, üzenetek véges többszöröződése és az üzenetek sorrendjének átrendezése történhet. A megengedett csatornaviselkedést a 14.1.2. példában úgy adtuk meg, hogy egy automata leírását a *gyenge veszteségkorlátozás* (GyVK) kiegészítő élénkségi feltételével adtuk meg. Ez a megadás azért előnyös, mert a csatornához minden pillanatban egy állapotot rendel, amely lehetővé teszi, hogy a protokoll működése során invariánsokat és szimulációs relációkat definiálhassunk.

A protokoll helyességének bizonyítását olyan invariánsok bevezetésével kezdjük, amelyek érintik a csatorna állapotát is. Ezért technikai okokból kénytelenek vagyunk a bizonyítást a csatornát leíró egyetlen bizonyos automata esetére megadni. Az invariánsokat tehát a 14.1.2. példában látott $A_{1,2}$ és $A_{2,1}$ alap b/k automaták esetére adjuk meg. A *be_tranzit* változók a két automata állapotösszetevői.

22.1. lemma. . *A* STENNING *protokollt az $A_{1,2}$ és $A_{2,1}$ csatornákon alkalmazva minden elérhető állapotra teljesülnek a következők.*

1. $címke_2 \leq címke_1 \leq címke_2 + 1$.
2. *Ha (m,k) a be_tranzit$_{1,2}$ értékei között van, akkor $k \leq címke_1$.*
3. *Ha $(m,címke_1)$ a be_tranzit$_{1,2}$ értékei között van, akkor puffer$_1$ első eleme*

m.

4. *Ha k a be_tranzit$_{2,1}$ értékei között van, akkor $k \leq$ címke$_2$.*
5. *Ha címke$_1$ = címke$_2$, vagy bármelyik be_tranzit$_{1,2}$ vagy be_tranzit$_{2,1}$-beli üzenet címkéje címke$_1$-gyel egyenlő, akkor puffer$_1$ nem lehet üres.*

Bizonyítás. Az egyszerű, indukción alapuló bizonyítást meghagyjuk gyakorlatnak (lásd 22-1. gyakorlat). □

Egy technikai lemmával folytatjuk annak bizonyítását, hogy a STENNING protokoll a megengedett csatornatípusok esetében megbízható FIFO üzenetátadást biztosít. A lemma a helyességet az egyes *csatornaspecifikációkat* és nem *tetszőleges megengedett csatornákat* tekintve mondja ki; az utóbbi eset azonban az előbbiből egyszerűen következik.

A következő lemma jelölése kissé nehézkes, ám valójában nem bonyolult. A feltételek azt mondják ki, hogy α vetületeként mind a P_1 és P_2 folyamatok, mind a veszteséges átrendező csatorna pártatlan végrehajtását kapjuk. A lemma állítása pedig azt jelenti, hogy α az F pártatlan története, tehát megbízható FIFO üzenetátadást biztosít.

22.2. lemma. . *Legyen α a* STENNING *protokoll tetszőleges végrehajtási sorozata az $A_{1,2}$ és $A_{2,1}$ csatornákon. Tegyük fel, hogy*

1. *$\alpha|P_1$ és $\alpha|P_2$ pártatlan; és*
2. *$\alpha|A_{1,2}$ és $\alpha|A_{2,1}$ teljesíti a 14.1.2. példa élénkségi feltételeit.*

Ekkor $\alpha|$külső$(F) \in$ pártatlan_történetek(F).

Bizonyításvázlat. Legyen $\beta =$ *történet*(α). Könnyen belátható, hogy $\beta \in$ *történet*(F), azaz a β `FOGAD` eseményeiben szereplő magas szintű üzenetek sorrendje a `KÜLD` eseményekben szereplő üzenetekének egy kezdőszelete. Ez úgy igazolható, hogy az $A_{1,2}$ és $A_{2,1}$ csatornákon végrehajtott STENNING protokoll (egyértékű) szimulációs relációját tekintjük F-re nézve és a 8.13. tételt alkalmazzuk. A szimulációs reláció bizonyítása a 22.1. lemmán alapul. A bizonyításnak ebben a szakaszában csak az élénkségi feltételeket használjuk, az 1–2. feltételekre nincs is szükség. A teljes bizonyítást meghagyjuk gyakorlatnak (22-2. és 22-3. gyakorlatok).

Az állítás igazolásához azt kell még belátnunk, hogy P_2 minden, a P_1-nek átadott magas szintű üzenetet egy idő után a fogadó oldali felhasználóhoz juttat. (A küldő és fogadó események egymáshoz tartozását F definíciója egyértelműen meghatározza.) Tegyük fel tehát, hogy m az első P_1-nek átadott magas szintű üzenet, amelyet P_2 nem kézbesít; az üzenet címkéje legyen k. Mivel $\alpha|P_2$ pártatlan, az üzenet nem kerülhet P_2 pufferébe, hiszen akkor P_2-nek azt át kellene adnia. Ebből az következik, hogy P_2 címkéje soha nem lehet nagyobb, mint $k-1$.

A következő lépésben megmutatjuk, hogy a fenti m üzenetnek egy idő után P_1 pufferében legelölre kell kerülnie. Ez a $k = 1$ esetben nyilvánvaló, hiszen ekkor m az elsőként indított üzenet. Ha $k \geq 1$, akkor az m-et megelőző üzenetet a feltételek szerint P_2 egy idő után kézbesíti; ekkor *címke*$_2 = k-1$ és ettől kezdve a címke értéke változatlan marad. Ám ebben az esetben $\alpha|P_2$ pártatlanságából következik, hogy P_2 örökké „$k-1$” üzeneteket fog küldeni; így az $A_{2,1}$ csatornára

vonatkozó gyenge veszteségkorlátozás (GyVK) feltétel szerint P_1-hez előbb-utóbb megérkezik ennek az üzenetnek egy példánya. Ekkor a $k-1$ címkéjű üzenet $puffer_1$-ből törlődik és így m legelölre kerül.

Amint az m üzenet elsővé válik P_1 pufferében, onnan el nem kerülhet, hiszen P_2 azt nem fogadja. Így $\alpha|P_1$ pártatlansága következtében P_1 folyamatosan az (m,k) üzeneteket fogja küldeni; az $A_{1,2}$ csatornára vonatkozó GyVK feltétel szerint P_2-höz előbb-utóbb ennek egy példánya megérkezik. Ellentmondáshoz jutottunk, hiszen ekkor P_2-nek az m üzenetet át kell vennie. □

A pártatlansági feltétel bizonyításának egy másik lehetséges módja a végrehajtások megfeleltetése. Ekkor a Stenning protokollról F-re vonatkozó szimulációt és a 8.13. tételt használhatjuk. A bizonyítást meghagyjuk gyakorlatnak (lásd 22-3. gyakorlat).

Végezetül kimondhatjuk a fő tételünket, amely szerint a Stenning protokoll a veszteséges átrendező csatornák esetében megbízható FIFO üzenetátadást biztosít. A tétel a 22.2. lemmából következik.

22.3. tétel. . *A* Stenning *protokoll bármely, a 14.1.2. példában megadott veszteséges átrendező csatornán alkalmazva megvalósítja F-et a következő értelemben. A protokoll és a csatorna minden α pártatlan végrehajtási sorozata esetében $\alpha|$külső$(F) \in$ pártatlan_ történetek(F).*

Bizonyításvázlat. A bizonyítást, amelyhez a 22.2 lemma mellett a b/k automaták összekapcsolásának a 8.2. és 8.4. tételekben szereplő alaptulajdonságait kell használni, meghagyjuk gyakorlatnak (lásd 22-4. gyakorlat). □

22.3.. A BitVáltó protokoll

A BitVáltó protokoll (BVP) a Stenning protokoll olyan változata, amely az átküldött üzenetekhez korlátlan méretű egész címkék helyett csak $\{0,1\}$ értékűeket rendel. A protokoll lényegében a Stenning „optimalizált" változata, az üzenetek mindössze az eredeti címke legalacsonyabb helyiértékű bitjét kapják. Az egyszerűsítés áraként azonban a csatornának erősebb követelményeket kell teljesítenie ahhoz, hogy a BVP helyesen működjön. Érdekességként megjegyezzük, hogy a BVP hosszú időn át alappéldául szolgált a különböző protokollvizsgálati módszerek bemutatására.

A BitVáltó protokoll (vázlatosan)

A P_1 folyamat az U_1 felhasználó által küldött magas szintű üzeneteket váltakozva 0 és 1 értékekkel címkézi és a $puffer_1$-ben tárolja. A $puffer_1$-ben található első üzenetet és címkéjét ismételten P_2-nek továbbítja. A P_2 először egy 1 címkéjű üzenetet, majd ezután minden további olyan üzenetet elfogad, amelynek címkéje az előzőleg elfogadott üzenetétől eltérő. Az elfogadott üzenetek sorrendben a $puffer_2$-be kerülnek, ahonnan P_2 sorrendben U_2-nek továbbítja azokat.

Az elfogadott magas szintű üzeneteket P_2 azzal nyugtázza, hogy címkéjüket visszaküldi P_1-nek. Amikor P_1-hez megérkezik a $puffer_1$-ben található első üzenetet címkéje, akkor továbblép és a pufferben található következő üzenetet küldi P_2-nek.

22.2. protokoll. BitVáltó

A P_1 folyamat

Lenyomat:
Bemeneti:
`KÜLD`$(m)_{1,2}$, $m \in M$
`fogad`$(b)_{2,1}$, $b \in \{0,1\}$
Kimeneti:
`küld`$(m,b)_{1,2}$, $m \in M$, $b \in \{0,1\}$

Állapotok:
$puffer_1$, egy M elemeiből álló FIFO sor, kezdetben üres
$címke_1 \in \{0,1\}$, kezdőértéke 1

Átmenetek:
`KÜLD`$(m)_{1,2}$
Hatás:
tegyük be m-et $puffer_1$-be

`küld`$(m,b)_{1,2}$
Előfeltétel:
m a $puffer_1$ első eleme
$b = címke_1$
Hatás:
üres

`fogad`$(b)_{2,1}$
Hatás:
if $b = címke_1$ **then**
távolítsuk el a $puffer_1$ első elemét (ha van ilyen)
$címke_1 := címke_1 + 1 \mod 2$

Taszkok:
$\{$`küld`$(m,b)_{1,2} : m \in M, b \in \{0,1\}\}$

A P_2 folyamat

Lenyomat:
Bemeneti:
`fogad`$(m,b)_{1,2}$, $m \in M$, $b \in \{0,1\}$
Kimeneti:
`FOGAD`$(m)_{1,2}$, $m \in M$
`küld`$(b)_{2,1}$, $b \in \{0,1\}$

Állapotok:
$puffer_2$, egy M elemeiből álló FIFO sor, kezdetben üres
$címke_2 \in \{0,1\}$, kezdőértéke 0

Átmenetek:
`FOGAD`$(m)_{1,2}$
Előfeltétel:
m a $puffer_2$ első eleme
Hatás:
távolítsuk el $puffer_2$ első elemét

`fogad`$(m,b)_{1,2}$
Hatás:
if $b \neq címke_2$ **then**
tegyük be m-et a $puffer_2$-be
$címke_2 := címke_2 + 1 \mod 2$

`küld`$(b)_{2,1}$
Előfeltétel:
$b = címke_2$
Hatás:
üres

Taszkok:
$\{$`FOGAD`$(m)_{1,2} : m \in M\}$
$\{$`küld`$(b)_{2,1} : b \in \{0,1\}\}$

A BitVáltó protokoll az egymással a $C_{1,2}$ és $C_{2,1}$ csatornákon keresztül kapcsolatot tartó P_1 és P_2 folyamatokból áll, a teljes protokoll ezek összekapcsolásával nyerhető. Helyes működéséhez megbízhatóbb csatornákra van szükség, mint a Stenning protokoll esetében: könnyen belátható, hogy a protokollban

hibát okozhat az üzenetek sorrendjének felcserélése. A $C_{1,2}$ és $C_{2,1}$ csatornák tehát a 14.1.1. példában leírt*veszteséges FIFO csatornák* lehetnek (a műveletek megfelelő átnevezésével), ahol a csatornákon korlátozott üzenetvesztés és véges üzenetismétlődés történhet, de az üzenetek sorrendje nem rendeződhet át. Az előzőekhez hasonlóan a megengedett csatornaviselkedést az automaták leírása mellett a *gyenge veszteségkorlátozás* (GyVK) kiegészítő élénkségi feltételével adjuk meg.

A BVP helyességét azzal a módszerrel bizonyítjuk, hogy szimulációval visszavezetjük működését a Stenning protokoll veszteséges FIFO (tehát most nem átrendező) csatornákon való működésére. Mivel a BVP és a Stenning protokoll eltérő címkékkel, így eltérő üzenetekkel és a csatornák felé eltérő átmenetekkel rendelkezik, megkülönböztetésül a következőkben $A_{1,2}$ és $A_{2,1}$ a BVP, $A'_{1,2}$ és $A'_{2,1}$ pedig a Stenning protokoll illesztőfelületéhez csatlakozó azon csatornaautomatákat fogja jelölni, amelyek a 14.1.1. példában leírt típusú veszteséges FIFO tulajdonságot teljesítik.

A bizonyítás kulcslépése az, hogy a Stenning protokoll $A'_{1,2}$ és $A'_{2,1}$ csatornáira egy új invariánst kell bevezetni. A következő, az invariáns tulajdonságait jellemző lemmában a két veszteséges FIFO csatorna $sor_{1,2}$ és $sor_{2,1}$ FIFO sorait vizsgáljuk.

22.4. lemma. . *A* Stenning *protokollt az $A'_{1,2}$ és $A'_{2,1}$ csatornákon alkalmazva minden elérhető állapotra teljesül a következő. A következő, egész számokból álló T sorozat nem csökkenő, továbbá az első és utolsó érték különbsége legfeljebb 1:*

1. *a $sor_{2,1}$-ben levő címkék értéke, az első üzenettől az utolsó felé haladva;*
2. *címke$_2$;*
3. *a $sor_{1,2}$-ben levő üzenetek címke részének értéke;*
4. *és végül címke$_1$.*

Bizonyítás. A bizonyítást az Olvasóra bízzuk (lásd 22-6. gyakorlat). □

A következő lemma az F BVP visszavezetését adja az F' Stenning protokollra, hiszen azt állítja, hogy veszteséges FIFO csatornákon a BVP minden α végrehajtási sorozatához a Stenning protokoll olyan α' végrehajtási sorozata tartozik, amely a felhasználói felületen azonos vele, azaz $\alpha|külső(F) = \alpha'|külső(F')$.

22.5. lemma. . *Legyen α az F BVP tetszőleges végrehajtási sorozata az $A_{1,2}$ és $A_{2,1}$ csatornákon. Tegyük fel, hogy*

1. *$\alpha|P_1$ és $\alpha|P_2$ pártatlan; és*
2. *$\alpha|A_{1,2}$ és $\alpha|A_{2,1}$ teljesíti a 14.1.1. példa élénkségi feltételeit.*

Tekintsük az F' Stenning *protokollt a P'_1 és P'_2 folyamatokkal az $A'_{1,2}$ és $A'_{2,1}$ csatornákon. Létezik a* Stenning *protokoll olyan α' végrehajtási sorozata, amelyre*

1. *$\alpha'|P'_1$ és $\alpha'|P'_2$ pártatlan;*
2. *$\alpha'|A'_{1,2}$ és $\alpha'|A'_{2,1}$ teljesíti a 14.1.1. példa élénkségi feltételeit; és*
3. *$\alpha|külső(F) = \alpha'|külső(F)$.*

A bizonyítás a BVP és a STENNING végrehajtási állapotainak megfeleltetésén alapul; ehhez hasonló gondolatmenettel találkozunk a 10.9.4. és a 16. fejezetekben. A fenti két protokoll közti megfeleltetés egy másik lehetséges módját a 8.5.5. szakaszban láttuk.

Bizonyításvázlat. A bizonyításban az $A_{1,2}$ és $A_{2,1}$ csatornákon működő BVP az $A'_{1,2}$ és $A'_{2,1}$ csatornákon működő STENNING protokollal szimuláljuk. Az f szimulációs reláció a BVP 0-1 értékű címkéit a STENNING protokoll címkéinek legalsó helyiértékű bitjeiből nyeri. Az f definíciója tehát a következő: ha s és u a BVP és a STENNING protokoll egy-egy állapota, akkor $(s, u) \in f$ pontosan akkor, ha

1. $s.puffer_1 = u.puffer_1$ és $s.puffer_2 = u.puffer_2$;
2. $s.címke_1 = u.címke_1 \mod 2$ és $s.címke_2 = u.címke_2 \mod 2$;
3. $s.sor_{1,2}$ és $u.sor_{1,2}$ elemszáma azonos, továbbá sorrendben az $u.sor_{1,2}$ egyes (m, k) elemeihez az $s.sor_{1,2}$ sorban az $(m, k \mod 2)$ elem tartozik;
4. $s.sor_{2,1}$ és $u.sor_{2,1}$ elemszáma azonos, továbbá sorrendben az $u.sor_{2,1}$ egyes k elemeihez az $s.sor_{2,1}$ sorban a $k \mod 2$ elem tartozik.

Könnyű belátni, hogy f szimulációs reláció: a megkövetelt tulajdonságok legtöbbje közvetlenül adódik az f definíciójából, illetve a BVP és a STENNING protokollok átmeneteinek hasonlóságából. A 22.4. lemma használható ahhoz, hogy a szimulációs reláció 2. (lépés) tulajdonságát a `fogad` lépések esetére igazoljuk, a következőképpen: a BVP minden egyes $(s, \texttt{fogad}_{1,2}, s')$ lépésére, ahol m-et P_2 elfogadja, meg kell mutatni, hogy a STENNING megfelelő `fogad` lépésében P'_2 is átveszi az üzenetet. A BVP akkor veszi át az üzenetet, ha $b \neq s.címke_2$; a STENNING megfelelő u állapotában ekkor az f szimulációs reláció definíciója szerint az érkező (m, k) üzenetre teljesül, hogy $k \neq u.címke_2 \mod 2$. Ekkor a 22.4. lemmából következik, hogy a szükséges $k = s.címke_2 + 1$ feltétel is teljesül.

A szimuláció önmagában még nem biztosítja az élénkségi tulajdonságok teljesülését. Azonban az f reláció a szokásosnál erősebb feltételeket is teljesít, hiszen a BVP minden lépését a STENNING egy azonos lépésével szimulálja, azzal az egyetlen eltéréssel, hogy ahol a STENNING esetében egy nemnegatív egész k címke szerepel, ott a BVP csak k legalacsonyabb helyiértékű b bitjét használja.

A BVP lemmabeli α végrehajtási sorozatához válasszuk ezek után a STENNING protokoll f szimuláció által adott α' végrehajtási sorozatát. Ekkor a következők igazak:

1. az α és α' végrehajtási sorozatokban található műveletek sorozata a fent említett eltérés kivételével azonos; és
2. az α és α' végrehajtási sorozatok sorrendben egymásnak megfelelő helyein álló műveleteire az f reláció teljesül.

Ez a két tulajdonság elegendő ahhoz, hogy belőle a lemma állításait levezessük. □

A 22.5. lemmából következik a BVP pártatlan végrehajtási sorozatait jellemző következő lemma, amely szerint ha a végrehajtási sorozathoz tartozó csatornaviselkedések teljesítik a veszteséges FIFO csatornák követelményeit, akkor a BVP megbízható FIFO üzenetátadást valósít meg.

22.6. lemma. . *Legyen α a BVP tetszőleges végrehajtási sorozata az $A_{1,2}$ és $A_{2,1}$*

csatornákon. Tegyük fel, hogy

1. $\alpha|P_1$ *és* $\alpha|P_2$ *pártatlan; és*

2. $\alpha|A_{1,2}$ *és* $\alpha|A_{2,1}$ *teljesíti a 14.1.1. példa élénkségi feltételeit.*

Ekkor $\alpha|$*külső*$(F) \in$ *pártatlan_ történetek*(F).

Bizonyításvázlat. Az állítás a 22.2. és 22.5. lemmákból következik. □

A 22.6. lemmából közvetlenül következik a BVP helyességi tétele, amely szerint a BVP protokoll megbízható FIFO üzenetátadást biztosít veszteséges FIFO csatornák esetében.

22.7. tétel. . *A* BVP *protokoll bármely, a 14.1.2. példában megadott veszteséges FIFO csatornán alkalmazva megvalósítja* F*-et a következő értelemben. A protokoll és a csatorna minden* α *pártatlan végrehajtási sorozata esetében* $\alpha|$*külső*$(F) \in$ *pártatlan_ történetek*(F).

Bizonyításvázlat. A bizonyítást, amelyhez a 22.6 lemma mellett a b/k automaták összekapcsolásának a 8.2. és 8.4. tételekben szereplő alaptulajdonságait kell használni, az olvasóra bízzuk. □

Végtelen üzenettöbbszöröződés.. Megjegyezzük, hogy a BVP akkor is működik, ha a csatornákra egy kicsit gyengébb feltételt téve megengedjük, hogy az üzenetek végtelenül ismétlődjenek. Ezek a csatornák sem rendezhetik azonban át az üzenetek sorrendjét és teljesíteniük kell a GyVK feltételt. Az egyetlen változást az jelenti ekkor, hogy a csatornák végtelen sokszor ismételhetik az *utolsóként* elküldött üzenetet, és így értelemszerűen csak véges számú üzenetküldés történhet. Az ilyen típusú csatornák megengedése annyiban nehezítené a 22.5. és a 22.6. lemmák, valamint a 22.7. tétel bizonyítását, hogy ezeket a csatornákat automaták helyett egyszerűbb axiómákkal jellemezni, ami viszont nem tenné lehetővé az invariánsok és szimulációk alkalmazását.

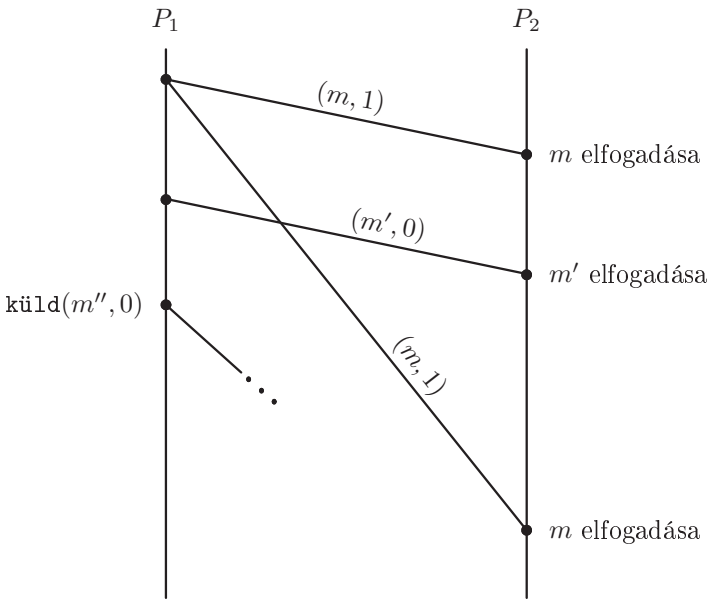
22.4.. Átrendeződés-tűrő korlátos címkéjű protokollok

A következő célunk olyan protokollok bemutatása, amelyekben az üzenetekhez rendelt címkék korlátosak, azonban az üzenetek sorrendjének felcserélődése esetében is helyesen működnek. Az előző két fejezetben látott protokollok megbízható FIFO kommunikációt biztosítanak abban az esetben, ha a csatornán az alacsony szintű üzenetek korlátozott elvesztése és véges többszöröződése előfordulhat. A STENNING protokoll ezen felül az üzenetek sorrendjének tetszőleges felcserélődését is javítani tudja, ehhez azonban korlátlan méretű címkéket alkalmaz. A BVP címkéi ugyanakkor egyetlen bitből állnak, viszont a protokoll az alacsony szintű üzenetek sorrendjének felcserélődése esetében nem működik. Célunk tehát a két protokoll előnyeinek ötvözése.

Bevezetésképpen megállapíthatjuk, hogy a BVP hibásan működik, ha az üzenetek a csatornán rossz sorrendben érkeznek: a protokoll P_2 folyamata tévesen elfogadhat egy korábbi m magas szintű üzenetet abban az esetben, ha annak a

címkéje megegyezik az éppen soron következő üzenetével. A BVP hibás működésére példa a 22.2. ábrán látható küld/fogad diagram. Az U_2 felhasználó tehát egy üzenetet több példányban is megkaphat, vagyis a protokoll megsérti a megbízható kommunikáció követelményeit.

Ebben a szakaszban három eredményt bizonyítunk a megbízható FIFO kommunikáció létezéséről illetve megoldhatatlanságáról. Mindhárom eredmény korlátos címkéket használó protokollok működését vizsgálja az üzenetek sorrendjét esetlegesen felcserélő csatornákon. Az egyszerűség céljából feltesszük, hogy a vizsgát protokollokban mind a magas, mind az alacsony szintű M és M' ábécé véges. Először a 22.4.1. szakaszban megmutatjuk, hogy nem létezik helyes protokoll akkor, ha az üzenetek felcserélődhetnek és ismétlődhetnek is. A 22.4.2. szakaszban ezután egy bonyolult protokollt mutatunk be, amely az üzenetek elvesztését és a sorrend tetszőleges felcserélődését is tolerálja, de a többszörözést nem. Végül a 22.4.3. szakaszban indokoljuk az előző protokollban fellépő nehézségeket azzal, hogy a fenti tulajdonságú protokollokra egy „hatékonysági" korlátot adunk.



22.2.. ábra. A BVP P_2 folyamata tévesen elfogadja az m üzenet második példányát.

22.4.1.. Megoldhatatlanság átrendezés és többszörözés esetében

Bebizonyítjuk, hogy nincs olyan (véges címkéjű) protokoll, amely megbízható FIFO kommunikációt biztosít olyan csatornák esetében, ahol az üzenetek többszöröződhetnek és a sorrendjük is megváltozhat. Az egyszerűség kedvéért a bizonyításban a csatornák megengedett viselkedését axiomatikusan írjuk le.

A következőkben a folyamatautomaták és a csatornatörténetek közti kapcsolat leírására újabb fogalmakat kell bevezetnünk. Amennyiben P_1 és P_2 egy protokollt megvalósító két automata, $Q_{1,2}$ és $Q_{2,1}$ pedig az őket összekötő csatornák történeteinek leírása, akkor azt mondjuk, hogy a $P_1 \times P_2$ egy α végrehajtási sorozata *ellentmondásmentes* $Q_{1,2}$-vel, ha $\alpha|\mathit{külső}(Q_{1,2}) \in \mathit{történetek}(Q_{1,2})$. A $Q_{2,1}$ konzisztenciáját hasonlóképpen definiáljuk. Továbbá α *végesen ellentmondásmentes* $Q_{1,2}$-vel (vagy hasonlóképpen $Q_{2,1}$-gyel), ha $\alpha|\mathit{külső}(Q_{1,2})$ a $\mathit{történetek}(Q_{1,2})$ valamelyik sorozatának véges kezdőszelete.

22.4.1. példa. b/k automaták és a konzisztencia fogalma.

Legyenek $A_{1,2}$ és $A_{2,1}$ megfelelő külső csatornaillesztővel ellátott tetszőleges b/k automaták és a $\mathit{történetek}(Q_{1,2})$ és a $\mathit{történetek}(Q_{2,1})$ legyen pontosan a két automata pártatlan történeteivel egyenlő. Ekkor

- a $P_1 \times P_2$ pártatlan végrehajtási sorozataihoz tartozó olyan történetek, amelyek a $Q_{1,2}$-vel és $Q_{2,1}$-gyel is ellentmondásmentesek, pontosan a $P_1 \times P_2 \times A_{1,2} \times A_{2,1}$ összekapcsolás pártatlan történetei; és
- a $P_1 \times P_2$ pártatlan végrehajtási sorozataihoz tartozó olyan *véges* történetek, amelyek a $Q_{1,2}$-vel és $Q_{2,1}$-gyel is *végesen* ellentmondásmentesek, pontosan a $P_1 \times P_2 \times A_{1,2} \times A_{2,1}$ összekapcsolás véges történetei.

A fenti két állítás könnyen levezethető a 8. fejezetbeli összekapcsolási tételek, elsősorban a 8.1., 8.3., 8.4. és 8.6. tételek segítségével.

A továbbiakban rögzítsük $Q_{1,2}$-t és $Q_{2,1}$-t úgy, hogy az $m \in M'$ esetében a $\texttt{küld}(m)_{1,2}$ bementekkel és a $\texttt{fogad}(m)_{1,2}$ kimenetekkel, illetve a $\texttt{küld}(m)_{2,1}$ és $\texttt{fogad}(m)_{2,1}$ ki- és bemenetekkel rendelkezzenek, és pontosan azokat a történeteket tartalmazzák, amelyekben nincsen veszteség és csak véges üzenettöbbszörözés történik, tetszőleges sorrendcsere ugyanakkor megengedett (formálisan olyan, a 14.1.2. szakaszban is látható *ok* függvényt választhatunk, amelyik injektív és véges sok elemet képezhet egyre). Ezután röviden azt mondhatjuk, hogy egy végrehajtási sorozat *ellentmondásmentes* vagy *végesen ellentmondásmentes*, ha ez a tulajdonság mind a $Q_{1,2}$-vel, mind a $Q_{2,1}$-gyel teljesül.

Készen állunk arra, hogy üzenetvesztő és sorrendet felcserélő csatornák esetében bebizonyítsuk a megbízható FIFO kommunikáció megoldhatatlanságát.

22.8. tétel. . *Az F FIFO kommunikáció az üzeneteket többszöröző és átrendező $Q_{1,2}$-t és $Q_{2,1}$ csatornákon nem valósítható meg korlátos címkéket használó protokollal. Azaz nincs olyan (P_1, P_2) folyamatpár, amelyre a $P_1 \times P_2$ minden α pártatlan végrehajtási sorozatára $\alpha|\mathit{külső}(F) \in \mathit{pártatlan\_történetek}(F)$.*

Bizonyítás. Tételezzük fel, hogy van ilyen (P_1, P_2) folyamatpár. A bizonyítás gondolatmenete abból áll, hogy amennyiben a kellően nagy számú magas szintű üzenetet indítunk, elérhetjük, hogy a csatornán a véges ábécé minden lehetséges

üzenete legalább egyszer megérkezzen P_2-höz. Az ezen felül indított üzenetek esetében P_2 számára nem eldönthető, hogy korábbi üzenetek másolatai, vagy egy újabb üzenet érkezett-e.

Formálisan a fenti gondolatmenet első lépésében előállítjuk a $P_1 \times P_2$ egy olyan α_1 végesen ellentmondásmentes végrehajtási sorozatát, amely minden lehetséges $\texttt{küld}(m)_{1,2}$ eseményt tartalmaz, azaz ha létezik olyan α végesen ellentmondásmentes végrehajtási sorozat, amelyben a $\texttt{küld}(m)_{1,2}$ esemény előfordul, akkor legalább egyszer előfordul α_1-ben is. Az α_1 végrehajtási sorozat úgy adható meg, hogy folyamatosan addig bővítjük az α_1-t KÜLD eseményekkel, amíg minden esemény elő nem fordul; mivel az üzenetek halmaza véges, a bővítés véges lépésben végetér. Legyen az α_1-beli KÜLD események száma n.

A következő lépésben legyen α_2 az α_1 olyan pártatlan, ellentmondásmentes bővítése, amely pontosan egy további KÜLD eseményt tartalmaz. A feltevés szerint az U_1 által küldött minden üzenetnek meg kell U_2-höz érkeznie, tehát α_2-ben pontosan $n+1$ FOGAD esemény lesz. Legyen α_3 az α_2 utolsó FOGAD eseményéig tartó kezdőszelete.

Végül ellentmondásra egy olyan α_4 végrehajtási sorozat megadása vezet, amely

1. α_1 bővítése;
2. P_1 számára α_1-től megkülönböztethetetlen; és
3. P_2 számára α_3-tól megkülönböztethetetlen.

A fenti tulajdonságokkal rendelkező végrehajtási sorozatát úgy kapjuk, hogy α_3-ból kihagyjuk P_1 összes α_1 utáni eseményét. A P_2 az α_1-belieken felül további belső eseményeket, illetve fogad, küld és FOGAD eseményeket is tartalmazhat. α_4 véges ellentmondásmentességének bizonyításában csak a fogad események okoznak problémát: meg kell mutatnunk, hogy P_2 annak ellenére fogadhatja α_3-beli alacsony szintű m üzeneteket, hogy azokat P_1 nem indíthatja el. Ez azonban lehetséges, hiszen P_1 már α_1-ben elindított minden elképzelhető m üzenetet, tehát a $\texttt{fogad}(m)_{2,1}$ események ezen üzenetek másolataihoz tartozhatnak.

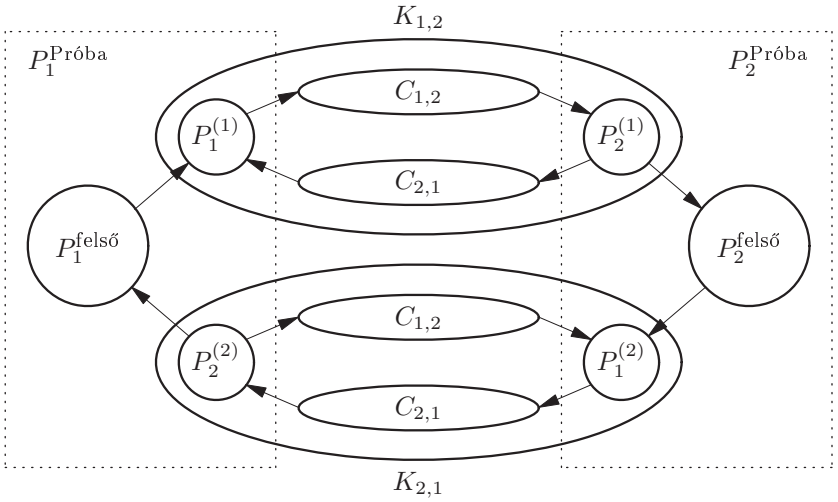
Ellentmondásra jutottunk: az α_4 végrehajtási sorozat n KÜLD és $n+1$ FOGAD eseményt tartalmaz. Ugyanakkor a protokoll pártatlan és ellentmondásmentes végrehajtási sorozatává bővíthető egyszerűen úgy, hogy nem adunk hozzá további KÜLD eseményeket. □

Bebizonyítottuk tehát, hogy az üzeneteket végesen többszöröző és tetszőlegesen átrendező csatornák esetében megbízható FIFO kommunikáció még abban az esetben sem hozható létre, ha üzenetvesztés nem következhet be.

22.4.2.. Egy üzenetvesztést és átrendezést tűrő korlátos címkés protokoll

Az előző szakaszban látott megoldhatatlansági bizonyítás után megismerkedünk az elméletileg érdekes és meglepő PRÓBA protokollal, amely korlátos címkék alkalmazásával megbízható kommunikációt valósít meg veszteséges és a sorrendet átrendező csatornák esetében. A protokoll az előző eredményeknek megfelelően természetesen nem lesz megbízható abban az esetben, ha az üzeneteket többszörözheti is a csatorna. Fontos megjegyeznünk, hogy nagyfokú bonyolultsága

és az egy magas szintű üzenet továbbításához szükséges alacsony szintű üzenetek magas száma miatt a PRÓBA protokoll a gyakorlatban nem alkalmazható. A protokollra tehát elsősorban úgy tekinthetünk, mint egy *ellenpéldára*, amely megmutatja, hogy *nem adható megoldhatatlansági bizonyítás* a fentiekben megadott csatornán való kommunikáció esetére.



22.3.. ábra. A PRÓBA protokoll felépítésének szintjei. A két multiplexelve használt $C_{1,2}$ és $C_{2,1}$ csatorna veszteséges átrendező, a két köztes $K_{1,2}$ és $K_{2,1}$ pedig veszteséges FIFO csatorna.

A PRÓBA protokoll (vázlatosan)

A PRÓBA protokollt legegyszerűbben a 22.3. ábrán látható módon, két szintre bontva képzelhetjük el, ahol az egyes szintek automatái a b/k automaták összekapcsolásának módszerével illeszkednek egymáshoz. Az *alsó szint* feladata az, hogy az eredeti, veszteséges és átrendező csatornák segítségével két olyan $K_{1,2}$ és $K_{2,1}$ *köztes csatornát* valósítson meg, amelyek nem rendezhetik át a sorrendet, ám az üzenetek elvesztése mellett a többszörözésük is előfordulhat. Az alsó szint csatornáin ezután a *felső szint* megbízható FIFO kommunikációt valósít meg, például a 22.3. szakaszbeli BVP segítségével.

Az alsó szinten a $K_{1,2}$ (és azonos módon a $K_{2,1}$) csatornát megvalósító P_1 automata csak abban az esetben küld a másik P_2 automatának alacsony szintű üzenetet, ha előzőleg P_2-től egy *próba* üzenet érkezett hozzá. Ebben az esetben az U_1 felhasználótól (amely most a csatorna bemenete) *utolsóként* átvett üzenetet továbbítja, amelyet addig az *utolsó* változóban tárol (a többi üzenet tárolása szükségtelen, hiszen a K köztes csatornákon az üzenetvesztés megengedett). Abból a célból, hogy mindig csak *próba*

üzenetekre válaszul küldhessen üzeneteket, P_1 egy *válaszolatlan* változóban számolja a megválaszolatlan *próba* üzenetek számát: érkezésükkor a változót eggyel növeli, üzenetküldéskor pedig eggyel csökkenti.

A P_2 automata P_1-nek folyamatosan *próba* üzeneteket küld, amelyek számát a *folyamatban* számlálóban tárolja. Egy adott t időpont előtt indított *próba* üzenetek számát jelölje *ezelőtt*. Amennyiben egy tetszőleges m üzenet a t időpont után P_2-höz több, mint *ezelőtt* példányban érkezik meg, biztosak lehetünk benne, hogy ezek egyike olyan üzenet, amelyet P_1 már a t időpont után indított.

A P_2 automata lépéseit osszuk ezek után fázisokba. Az első fázis az automata indításával kezdődik és minden fázis az első olyan lépéssel ér véget, amelyben az automata átad egy magas szintű üzenetet az U_2 kimenetnek. Az *ezelőtt* értéke az előző fázis végéig indított *próba* üzenetek száma; egy *számláló*(m) állapotváltozó pedig minden m üzenet esetében tárolja az m üzenet azon példányainak számát, amelyek a folyamatban levő fázisban érkeztek (a számlálók tárolása megoldható, mivel m véges sok értéket vehet csak fel). Amennyiben egy m alacsony szintű üzenetre *számláló*$(m) >$ *ezelőtt*, az m-et, mint magas szintű üzenetet az automata U_2-nek átadja és a folyamatban levő fázis befejeződik.

A protokollt megvalósító $P_1^{\text{PRÓBA}}$ és $P_2^{\text{PRÓBA}}$ automaták két $P_1^{(1)}$–$P_2^{(1)}$, $P_1^{(2)}$–$P_2^{(2)}$ *alsó* és egy $P_1^{\text{felső}}$–$P_2^{\text{felső}}$ *felső* szintű automatapár összekapcsolásaiként állnak elő. A 22.3. ábrán látható módon $P_1^{\text{PRÓBA}}$-t úgy kapjuk, hogy $P_1^{\text{felső}}$ kimenetéhez $P_1^{(1)}$-et, bemenetéhez pedig $P_2^{(2)}$-t kapcsoljuk. Hasonlóképpen $P_2^{\text{PRÓBA}}$ a $P_2^{\text{felső}}$ kimenetéhez $P_1^{(2)}$-t, bemenetéhez pedig $P_2^{(1)}$-et kapcsolva adódik.

A két alsó szintű automatapár mindegyike a 14.1.2. szakaszban megadott *ok* függvénnyel axiomatizált $C_{1,2}$ és $C_{2,1}$ csatornákon valósít meg kommunikációt, amelyekről megköveteljük, hogy ne engedjék meg az üzenetek többszörözését, azonban teljesítsék az EVK veszteségkorlátozási feltételt. Formálisan megfogalmazva az *ok* függvény injektív, ám nem feltétlenül szürjektív vagy monoton. Később látni fogjuk, hogy a szokásosnál erősebb EVK feltétel azért szükséges, mert a csatornákat a két protokoll közösen használja.

Az alsó szintű automaták be- és kimenetei $P_1^{\text{felső}}$ és $P_2^{\text{felső}}$ között egy $K_{1,2}$, illetve egy $K_{2,1}$ *köztes csatornát* határoznak meg, amelyekről megköveteljük, hogy a 22.3. alfejezetben látott *veszteséges FIFO csatornák* legyenek, végtelen üzenettöbbszörözést megengedve.

A felső szintű automaták például a BVP protokoll automatái lehetnek. A 657. oldali megjegyzés szerint ezek az automaták végtelen többszörözést megengedő veszteséges FIFO csatornák esetében is megbízható FIFO kommunikációt valósítanak meg.

Az alsó szintű automaták leírása bonyolultabb. A két azonos automatapárt a felső indexet elhagyva P_1-gyel és P_2-vel jelölve együtt adjuk meg; az automaták be- és kimeneti csatolóit U_1-gyel és U_2-vel jelöljük. Mind a $K_{1,2}$, mind a $K_{2,1}$ köztes csatorna a fenti automatapár egy-egy példányával valósítható meg, amelyek a $C_{1,2}$ és $C_{2,1}$ csatornákat egyidejűleg, „multiplexelve” használhatják. A

multiplexelés például a 14-7. gyakorlatban látott módon valósítható meg.

22.3. protokoll. PRÓBA

Az alsó szintű P_1 folyamat

Lenyomat:
Bemeneti:
  KÜLD$(m)_{1,2}$, $m \in M$
  fogad(„*próba*”)$_{2,1}$
Kimeneti:
  küld$(m)_{1,2}$, $m \in M$

Állapotok:
utolsó $\in M \cup \{null\}$, kezdetben *null*
válaszolatlan $\in \mathbb{N}$, kezdetben 0

Átmenetek:
KÜLD$(m)_{1,2}$
  Hatás:
    utolsó $:= m$

küld$(m)_{1,2}$
  Előfeltétel:
    válaszolatlan > 0
    $m =$ *utolsó*
  Hatás:
    válaszolatlan $:=$
      $:=$ *válaszolatlan* $- 1$

fogad(„*próba*”)$_{2,1}$
  Hatás:
    válaszolatlan $:=$
      $:=$ *válaszolatlan* $+ 1$

Taszkok:
{küld$(m)_{1,2} : m \in M$}

Az alsó szintű P_2 folyamat

Lenyomat:
Bemeneti:
  fogad$(m)_{1,2}$, $m \in M$
Kimeneti:
  FOGAD$(m)_{1,2}$, $m \in M$
  küld(„*próba*”)$_{2,1}$

Állapotok:
folyamatban $\in \mathbb{N}$, kezdetben 0
ezelőtt $\in \mathbb{N}$, kezdetben 0
minden $m \in M$ esetében
  számláló$(m) \in \mathbb{N}$, kezdetben 0

Átmenetek:
FOGAD$(m)_{1,2}$
  Előfeltétel:
    számláló$(m) >$ *ezelőtt*
  Hatás:
    ezelőtt $:=$ *folyamatban*
    minden $m' \in M$ esetében
      számláló$(m') := 0$

küld(„*próba*”)$_{2,1}$
  Hatás:
    folyamatban $:=$
      $:=$ *folyamatban* $+ 1$

fogad$(m)_{1,2}$
  Hatás:
    számláló$(m) :=$
      $:=$ *számláló*$(m) + 1$

Taszkok:
{FOGAD$(m)_{1,2} : m \in M$}
{fogad(„*próba*”)$_{2,1}$}

A PRÓBA protokoll helyességének bizonyításához először azt mutatjuk meg, hogy többszörözést meg nem engedő csatornákon az alsó szint P_1 és P_2 automatái (a felső indexüket a továbbiakban elhagyjuk) egy, az üzenetek sorrendjének felcserélődését meg nem engedő köztes csatornát valósítanak meg. A lemmához elegendő azt megkövetelnünk, hogy a csatornák a GyVK veszteségkorlátozási feltételt teljesítsék.

22.9. lemma. . *Legyen $C_{1,2}$ és $C_{2,1}$ két olyan, ok függvénnyel axiomatizált csa-*

torna, amely teljesíti a GyVK feltételt és nem engedi meg az üzenetek sorrendjének átrendezését, azaz az ok függvény egy injektív, de nem feltétlenül szürjektív vagy monoton. Legyen P_1 és P_2 a PRÓBA *protokoll alsó szintjének automatapárja, amely a $C_{1,2}$ és $C_{2,1}$ csatornákon kommunikál. Ekkor a két automata egy, a 22.3. alfejezetben látott K veszteséges FIFO csatornát valósít meg abban az értelemben, hogy minden α pártatlan végrehajtási sorozatra $\alpha|\mathit{külső}(P_1, P_2) \in \mathit{pártatlan\_történetek}(K)$.*

Bizonyításvázlat. Először megmutatjuk, hogy K nem rendezi át az üzenetek sorrendjét. Ebből a célból osszuk az automaták lépéseinek sorozatát fázisokba. Az első fázis az automaták indításával kezdődjön és minden fázis az első olyan lépéssel érjen véget, amelyben a P_2 automata átad egy m üzenetet az U_2 kimenet felhasználójának. Mivel minden fázisban egyetlen üzenet kerül átadásra, elegendő megmutatnunk, hogy az m üzenet legalább egy példányát P_1 a vizsgált fázisban, tehát már az előző üzenet átvétele után indította. Ez következik abból, hogy az m átvételének feltétele *számláló*$(m) >$ *ezelőtt*, ahol *ezelőtt* a fázist megelőzően elindított *próba* üzenetek száma. Tehát ha az m üzenet ennél több példányban érkezik, akkor legalább egy példánya már a vizsgált fázisban indult, hiszen P_1 legfeljebb annyi üzenetet indít, ahány *próba* üzenet hozzá érkezik. Megjegyezzük, hogy a protokoll a fázisban esetlegesen elindított minden további üzenetet elveszít.

A bizonyítással készen vagyunk, ha megmutatjuk, hogy K teljesíti a GyVK feltételt, azaz végtelen sok `KÜLD` esemény közül végtelen sokhoz kell `FOGAD` eseménynek tartoznia. Tételezzük tehát fel, hogy csak véges sok `FOGAD` esemény történik és tekintsük az utolsó, be nem fejeződő fázist, amelyben már nincs `FOGAD` esemény. Amennyiben az m alacsony szintű üzenetek (véges) r lehetséges értéket vehetnek fel, $r \cdot$ *ezelőtt* $+ 1$ alacsony szintű üzenet érkezése esetében legalább az egyikre teljesülne *számláló*$(m) >$ *ezelőtt*.

Mivel feltettük, hogy újabb `FOGAD` esemény nem következik be, a P_2-höz az utolsó fázisban érkező alacsony szintű üzenetek száma legfeljebb $r \cdot$ *ezelőtt*, az összes fázisban együttesen tehát véges. Ám ekkor a $C_{1,2}$ csatornán indított üzenetek száma is véges a $C_{1,2}$ GyVK feltétele szerint. Az, hogy P_1-hez végtelen sok `KÜLD` esemény érkezik, de csak véges sok üzenetet indít, csak úgy lehetséges, hogy véges sok *próba* üzenet érkezik hozzá. Így ellentmondásba kerültünk a $C_{2,1}$ csatorna GyVK feltételével, hiszen azután, hogy P_2 a hozzá érkező véges sok üzenet átveszi, csak *próba* üzeneteket küldő lépést hajt végre, végtelen sokszor. □

A PRÓBA protokoll a 22.3. ábrán látható két alsó és egy felső szintű protokoll összekapcsolásából, illetve a két alsó szintű protokoll által – a 14-7. gyakorlatban látott módon – multiplexelve használt $C_{1,2}$ és $C_{2,1}$ csatornákból áll.

A protokoll helyességének bizonyítása látszólag egyszerűen következik a fenti lemmából és a 22.3. alfejezetbeli BVP helyességéből. Az egyetlen technikai nehézséget a $C_{1,2}$ és $C_{2,1}$ csatornák közös használata jelenti. Biztosítanunk kell, hogy a csatornák a két $P_1^{(1)}$–$P_2^{(1)}$ és $P_1^{(2)}$–$P_2^{(2)}$ alsó szintű automatapár üzeneteire külön-külön is teljesítsék a GyVK feltételt. Elképzelhető ugyanis, hogy a csatornák ugyan végtelen elindított üzenetből végtelen sokat át is adnak, azonban véges kivétellel az üzenetek mindegyike csak az egyik, például $K_{1,2}$-t megvalósító

automatapárhoz tartozik.

A csatornák közös használatából eredő nehézséget azzal küszöböljük ki, hogy a $C_{1,2}$ és $C_{2,1}$ csatornákra nemcsak a GyVK, hanem az EVK feltételt is kikötjük. Az EVK (erős veszteségkorlátozás) feltétel szerint a csatorna minden egyes végtelen sokszor indított üzenet*típust* végtelen sokszor továbbít. Így a GyVK feltétel a két alsó szintű folyamat által indított üzenetekre, mint üzenettípusokra külön-külön is következik.

22.10. tétel. . *A* Próba *protokoll megvalósítja az F megbízható FIFO csatornát abban az esetben, ha az alkalmazott csatornákon az üzenetek többszörözése nem megengedett és teljesítik az EVK feltételt, azaz minden α pártatlan végrehajtási sorozatra $\alpha|_{külső(F)} \in pártatlan\_történetek(F)$.*

Bizonyítás. Az EVK élénkségi feltétel a csatornákat multiplexelve használó mindkét alsó szintű folyamatpár számára biztosítja, hogy a csatornán általuk küldött üzenetekre külön-külön teljesüljön a gyengébb GyVK feltétel. Így a 22.9. lemma alkalmazható, tehát a közbülső K csatorna mindkét irányban veszteséges FIFO csatorna lesz. A tétel következik a felső szinten alkalmazott BVP helyességére vonatkozó 22.7. tételből. □

Hatékonyság-elemzés.. A fejezet többi protokolljához hasonlóan a Próba protokoll bonyolultságának formális elemzését is mellőzni fogjuk. Megjegyezzük azonban, hogy a protokoll hatékonysága rendkívül rossz. Könnyű látni, hogy minden egyes újabb magas szintű üzenet továbbítása legalább az előzőleg elveszített alacsony szintű üzenetek számával arányos mennyiségű kommunikációt igényel. Ha tehát egy tetszőleges időpontig k alacsony szintű üzenet elveszik, akkor minden további magas szintű üzenet továbbítása $k+1$ alacsony szintű üzenetet igényel még akkor is, ha további üzenetvesztés nem történik. A következő fejezetben látni fogjuk azonban, hogy a korlátos címkéjű átrendezéstűrő protokollok kommunikációjának elkerülhetetlenül rossz a hatásfoka.

22.4.3.. A veszteséges átrendező csatornák hatékonysági korlátja

A Próba protokoll megbízható FIFO kommunikációt biztosít veszteséges átrendező, ám az üzeneteket nem többszöröző csatornák esetében. Az előző szakaszban megállapítottuk, hogy ez a protokoll nem hatékony, hiszen az újabb és újabb magas szintű üzenetek továbbításához egyre több alacsony szintű üzenetváltásra lehet szükség. Ebben a szakaszban megmutatjuk, hogy nem létezik olyan, a fenti feltételt teljesítő protokoll, amely a Próba protokollnál lényegesen hatékonyabb lenne.

A szakaszbeli bizonyítás formalizmusa az előző, 22.8. tételben látott megoldhatatlansági eredményéhez hasonlóan a megengedett csatornaviselkedéseket meghatározó történetek axiomatikus tulajdonságaira fog épülni. Használni fogjuk a 22.4.1. szakaszban alkalmazott, a folyamatautomatákat a csatornatörténet-tulajdonságokkal összekapcsoló *konzisztencia*-fogalmakat is. Beszélni fogunk tehát arról, hogy a $P_1 \times P_2$ egy végrehajtási sorozata (*végesen*) *ellentmondásmentes* a csatornaillesztő felületek történeteit jellemző tulajdonsággal.

A szakasz további részében rögzítjük a $Q_{1,2}$ és $Q_{2,1}$ történeteit jellemző tulajdonságokat és azt mondjuk, hogy egy végrehajtási sorozat ellentmondásmentes vagy végesen ellentmondásmentes, ha a megfelelő tulajdonság mindkét csatornára nézve teljesül. Ebből a célból tekintsük azt a csatornát, amelynek történetei pontosan a többszörözést nem, de veszteséget és átrendezést megengedő tulajdonságúak és amelyek a GyVK veszteségkorlátozási feltételt teljesítik. Ezután legyen $Q_{1,2}$ a csatorna egy példánya a $\texttt{küld}(m)_{1,2}$, $m \in M'$ bementekkel és $\texttt{fogad}(m)_{1,2}$, $m \in M'$ kimentekkel, $Q_{2,1}$ pedig egy másik példánya a fordított csatornairányhoz tartozó ki- és bementetekkel.

A következőkben definiáljuk a kommunikáció költségének fogalmát, melynek segítségével bizonyítani fogjuk, hogy minden ellentmondásmentes végrehajtási sorozatú protokoll legalább akkora költséggel kommunikál, mint az előző szakaszbeli PRÓBA protokoll. Megjegyezzük, hogy a PRÓBA protokoll minden pártatlan végrehajtási sorozata valóban ellentmondásmentes.

Nevezzük a $P_1 \times P_2$ egy α végesen ellentmondásmentes végrehajtási sorozatát *teljes*nek, ha α azonos számú `KÜLD` és `FOGAD` lépést tartalmaz. Másképpen ez azt jelenti, hogy a protokoll minden egyes U_1 által indított üzenetet át tudott adni U_2-nek.

A kommunikáció költségét a *k-üzenetkorlátosság* fogalmával fogjuk mérni, amely lényegében azt jelenti, hogy a protokoll a *legjobb esetben* véges k alacsony szintű üzenetváltással mindig át tud adni egy új magas szintű üzenetet. Legyen α egy teljes végrehajtási sorozat, $k \in \mathbb{N}^+$ és $m \in M$. Ekkor az α egy α' kiterjesztése az m-re nézve *k-kiterjesztés*, ha teljesülnek a következők.

1. Az α' bővített, α utáni részén pontosan két, a felhasználói felületet érintő lépés található, egy $\texttt{KÜLD}(m)_{1,2}$ és egy $\texttt{FOGAD}(m)_{1,2}$. (Ami azt jelenti, hogy a bővítésben pontosan egy, m-et tartalmazó magas szintű üzenet sikeres továbbítása történik, következésképpen az α' is teljes.)
2. Minden olyan alacsony szintű üzenetet, amelyet P_2 az α'-nek α utáni részén vesz át, P_1 az α utáni részen indít. (Azaz a bővítésben nem érkezik régi alacsony szintű üzenet.)
3. Az α'-nek α utáni részében legfeljebb k darab $\texttt{fogad}_{1,2}$ esemény található.

Egy protokoll *k-üzenetkorlátos*, ha minden α teljes végrehajtási sorozata és minden $m \in M$ esetében létezik α-nak k-kiterjesztése m-re nézve. Egy protokoll *üzenetkorlátos*, ha k-üzenetkorlátos valamely $k \in \mathbb{N}^+$ esetében.

Az üzenetkorlátosság definíciója a protokollal szemben meglehetősen gyenge követelményt támaszt. Elegendő, ha a protokoll csak a legjobb esetben képes az újabb magas szintű üzenetet hatékonyan átadni, azaz például akkor, amikor az újabb üzenetet továbbító lépések során alacsony szintű üzenetek nem vesznek el. Meg fogjuk azonban mutatni, hogy a feltételt gyengesége ellenére sem tudja egyetlen olyan protokoll sem teljesíteni, amely veszteséges átrendező csatornákon megbízható FIFO kommunikációt biztosít.

22.11. tétel. . *Nem létezik olyan (P_1, P_2) üzenetkorlátos, korlátos címkéjű protokoll, amely az F megbízható FIFO kommunikációt a $Q_{1,2}$ és $Q_{2,1}$ veszteséges átrendező csatornákon megvalósítja, azaz $\alpha|\textit{külső}(F) \in \textit{pártatlan\_történetek}(F)$ nem teljesülhet a $P_1 \times P_2$ minden α ellentmondásmentes pártatlan végrehajtási*

sorozatára.

Bizonyítás. Tételezzük fel, hogy egy $k \in \mathbb{N}$ érték esetében létezik a tételbeli P_1, P_2 protokoll. A bizonyítás a 22.8. tételéhez hasonlóan két olyan történet megadásával történik, amelyek a protokoll számára nem lehetnek egymástól megkülönböztethetők. A következő a szükséges történeteket a következő két feltételnek eleget tevő M' elemeiből álló T multihalmaz, a $P_1 \times P_2$ egy α teljes végrehajtási sorozata, valamint az α egy tetszőleges α' k-bővítése segítségével adjuk meg. Először a tételt bizonyítjuk be; az alábbi tulajdonságok igazolására ez után fogunk rátérni.

1. Az α végrehajtása után T minden eleme P_1-ből P_2 felé *be_ tranzit* állapotban van[1] (azaz elküldésre kerültek, de még nem érkeztek meg).
2. Az α'-nek α utáni részén a P_2 által fogadott alacsony szintű üzenetek a T rész-multihalmazát alkotják.

Tételezzük fel, hogy egy $k \in \mathbb{N}$ értékre állításunkkal ellentétben létezik a keresett tulajdonságú k-üzenetkorlátos (P_1, P_2) protokoll. Ellentmondásra a 22.8. tétel bizonyításához hasonló konstrukcióval jutunk. A fenti 1–2. tulajdonságoknak eleget tevő α, α' és T segítségével megadunk egy α_1 végesen ellentmondásmentes végrehajtási sorozatát, amelyre

- α_1 az α bővítése;
- P_1 számára α_1 és α nem megkülönböztethető; és
- P_2 számára α_1 és α' nem megkülönböztethető.

Az α_1-t úgy kaphatjuk meg, hogy α'-ből törlünk minden P_1-t érintő α utáni műveletet, miközben P_2 lépéseit változatlanul hagyjuk. Ez a konstrukció a protokoll pártatlan végrehajtási sorozatát adja, hiszen a P_2 számára szükséges α utáni `fogad` események alacsony szintű üzenetei a T multihalmaz tulajdonságai alapján már α-ban mind *be_ tranzit* állapotban vannak. Így a 22.8. tétel bizonyításához hasonlóan ellentmondásra jutunk, mert létrehozunk egy pártatlan és ellentmondásmentes végrehajtási sorozatát, amely több `FOGAD`, mint `KÜLD` eseményt tartalmaz. □

Rátérünk az 1–2. tulajdonságokat teljesítő történetek megadására. A szükséges konstrukció kulcslépése a következő segédtétel, amely szerint ha alacsony szintű üzenetek egy, a feltételeket nem teljesítő T multihalmaza *be_ tranzit* állapotban van, akkor T egy nagyobb, hasonló tulajdonságú multihalmazzá bővíthető. Először az alábbi segédtétel segítségével adjuk meg az ellentmondáshoz szükséges T halmazt, α végrehajtási sorozatot és α' bővítést. A 22.12. segédtétel igazolását a teljes bizonyítás végére hagyjuk.

22.12. segédtétel. . *Legyen α egy teljes végrehajtási sorozat és T alacsony szintű üzenetek olyan multihalmaza, amelynek minden eleme P_1-ből P_2 felé be_ tranzit állapotban van az α végrehajtása után. Tételezzük fel, hogy T minden elemet legfeljebb k példányban tartalmaz. Ekkor a következő két állítás közül legalább egy teljesül.*

[1] Az elküldött, de meg nem érkezett üzenetek multihalmazát α egyértelműen meghatározza.

1. *Létezik olyan m és az α olyan α' k-bővítése m-re nézve, amelyre az α'-nek α utáni részén a P_2 által fogadott alacsony szintű üzenetek a T részmultihalmazát alkotják.*
2. *Létezik α-nak olyan α' teljes bővítése és olyan T' multihalmaz, amely α' végrehajtása után be_tranzit állapotban lévő alacsony szintű üzenetekből áll, minden elemet legfeljebb k példányban tartalmaz és T a T' valódi részhalmaza.*[2]

Az 1–2. tulajdonságok bizonyítása. Megadjuk alacsony szintű üzenetek multihalmazainak egy $T_0, T_1, \ldots$ és teljes végrehajtási sorozatoknak egy $\alpha_0, \alpha_1, \ldots$ sorozatát úgy, hogy minden i-re α_{i+1} az α_i bővítése és $T_i \subsetneq T_{i+1}$; továbbá T_i minden elemet legfeljebb k példányban tartalmaz és minden eleme α_i végrehajtása után P_1-től P_2 felé *be_tranzit* állapotban van.

A sorozatok kezdőelemeit úgy választjuk, hogy α_0 csak a P_1 és P_2 kezdőállapotait tartalmazza, T_0 pedig legyen üres. Ez a választás a 22.12. segédtétel feltételeit teljesíti. Ezután minden $i \geq 0$ esetében ha a 22.12. segédtétel 2. esete áll fenn, akkor a sorozatot az $\alpha_{i+1} = \alpha'$ és $T_{i+1} = T'$ választással folytathatjuk úgy, hogy az állítás feltételei továbbra is teljesülnek. Amennyiben viszont bármikor a segédtétel 1. esete áll fenn, a tétel bizonyításával készen leszünk, tehát feltehető, hogy a két sorozat végtelen.

Megmutatjuk végül, hogy a $T_0, T_1, \ldots$ sorozat nem lehet végtelen. Mivel minden elem az őt követő valódi részhalmaza, a multihalmazok méretének minden határon túl kellene nőnie. Azonban a véges M' alacsony szintű üzenethalmaz minden eleme legfeljebb k példányban fordulhat egy T_i-ben elő, így a sorozat hossza legfeljebb $k \cdot |M'| + 1$ lehet. □

22.12. segédtétel bizonyítása. Legyen $m \in M$ egy tetszőleges magas szintű üzenet és legyen α_1 az α k-kiterjesztése m-re nézve. Az α_1 létezése a protokoll k-üzenetkorlátosságából következik. Feltételezhetjük, hogy az $\alpha' = \alpha_1$ választásra nem teljesül a segédtétel 1. esete, tehát van olyan $p \in M'$, hogy az α_1-ben α utáni $\texttt{fogad}(p)_{1,2}$ események száma nagyobb, mint p előfordulásainak száma a T multihalmazban. Mivel α_1 egy k-kiterjesztés, az előbbi szám legfeljebb k és így p előfordulásainak száma legfeljebb $k-1$. Legyen $T' = T \cup \{p\}$, azaz növeljük meg p előfordulásainak számát eggyel; ezzel tehát az előfordulások száma még mindig legfeljebb k marad.

Célunk most egy olyan α' teljes k-bővítés megadása, amelyben a T' halmaz marad *be_tranzit* állapotban. Egyrészt tudjuk, hogy T *be_tranzit* állapotban marad, mivel a k-bővítés definíciója szerint egy α utáni `fogad` esemény nem tartozhat α előtti `küld` eseményhez. Másrészt biztos, hogy α_1-ben α után legalább egy $\texttt{küld}(p)_{1,2}$ esemény bekövetkezik, hiszen a k-bővítés definíciója szerint egy α utáni $\texttt{fogad}(p)_{1,2}$ eseményhez egy α utáni $\texttt{küld}(p)_{1,2}$ esemény tartozik. Legyen α_2 az α_1-nek az α utáni első $\texttt{küld}(p)_{1,2}$ eseményig terjedő kezdőszelete; ez az

[2] Egy T multihalmaz akkor valódi részhalmaza T'-nek, ha T'-höz képest a T legalább egy elemet legalább eggyel több példányban tartalmaz.

α-nak végesen ellentmondásmentes bővítése és a két legutóbbi megállapításunk értelmében T'-t *be_tranzit* állapotban hagyja.

A bizonyítást azzal fejezzük be, hogy α_2-t egy teljes α' bővítéssé alakítjuk, amely a segédtétel 1. esetét teljesíti. Az $\alpha' = \alpha_2$ választással azonnal készen vagyunk, ha α_2 a $\texttt{KÜLD}(m)_{1,2}$ és $\texttt{FOGAD}(m)_{1,2}$ lépéseket egyszerre vagy tartalmazza, vagy nem tartalmazza. Tételezzük tehát fel, hogy csak az első eseményt tartalmazza. Az α'-t ekkor az α_2 egy olyan bővítéseként keressük, amely egy további $\texttt{FOGAD}(m)_{1,2}$ eseményt tartalmaz, de P_2 nem fogad T'-beli alacsony szintű üzenetet. Először egy ezzel a tulajdonsággal rendelkező α_3 ellentmondásmentes végrehajtási sorozatát adunk meg, amelynek a $\texttt{FOGAD}(m)_{1,2}$ eseményig terjedő kezdőszelete választható a 2. esetet teljesítő α'-nek.

Elegendő tehát megmutatnunk, hogy a $P_1 \times P_2$ minden végesen ellentmondásmentes α_2 végrehajtása kiegészíthető egy olyan pártatlan és ellentmondásmentes α_3 végrehajtási sorozattá, amely további $\texttt{KÜLD}_{1,2}$ eseményeket nem tartalmaz és minden új $\texttt{küld}_{1,2}$ esemény egy új, tehát nem α_2-beli $\texttt{fogad}_{1,2}$ eseményhez tartozik. Ez következik a $P_1 \times P_2$ protokoll helyességét arra az esetre alkalmazva, amikor minden *be_tranzit* állapotban levő eseményt α_2 végeztével a csatornák elveszítenek. A megbízható kommunikáció feltétele szerint előbb-utóbb a $\texttt{FOGAD}(m)_{1,2}$ eseménynek be kell következnie, ezt pedig a protokoll csak új alacsony szintű üzenetek indításával és fogadásával teljesítheti. □

22.5.. Katasztrófatűrő protokollok

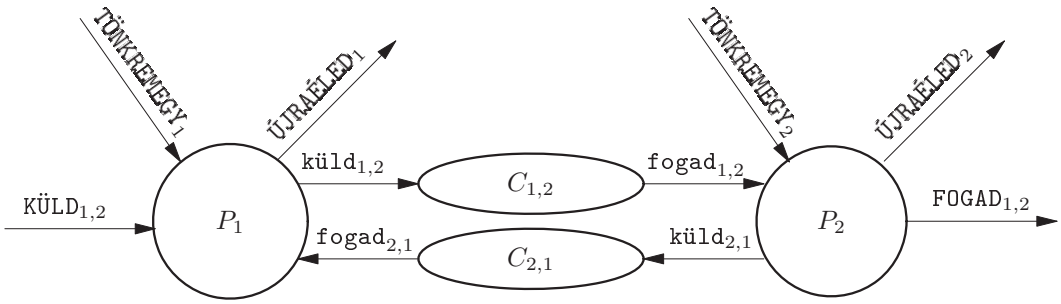
A fejezet eddigi eredményei a megbízható FIFO kommunikáció megvalósíthatóságának kérdését szinte minden elképzelhető esetben megválaszolják – legalábbis akkor, ha a folyamatok megbízhatóak és csak a csatorna hibás viselkedésével kell foglalkoznunk. Az eddigiektől eltérő helyzet áll azonban elő, ha a folyamatok viselkedését is bevonjuk a vizsgálatba. Két kommunikáló végpont esetében természetesen értelmetlen a folyamatok megállási vagy bizánci hibáiról beszélnünk. Olyan *katasztrófa* azonban előfordulhat, amely után a folyamat később *újraéled*. Nem okoz problémát a katasztrófa abban az esetben, ha újraéledéskor a folyamat egyszerűen a katasztrófa előtti állapotból folytatódik, hiszen ez mindössze a folyamat ideiglenes felfüggesztődését jelenti. Gondot okozhat azonban, ha egy folyamat katasztrófája bizonyos állapotinformációk elvesztésével is járhat.

Vizsgáljuk meg, hogy megvalósítható-e a megbízható FIFO kommunikáció abban az esetben, ha a folyamatokban információvesztéssel járó katasztrófa állhat be, például amikor a valós processzor illékony memóriával (vagy stabil és illékony memória kombinációjával) rendelkezik, amely egy katasztrófa során tartalmát elveszti. A katasztrófa utáni újraéledés a stabil memóriában tárolt állapotok és az illékony memória alapértelmezésbeli értékeinek segítségével történhet. A valós helyzetekben alkalmazott újraéledési protokollt, amely az illékony memória tartalmát a stabil memória segítségével helyreállítja, formális modellünkben egyetlen *újraéledési lépés*sel helyettesítjük.

Első eredményünk (22.5.1. szakasz) megmutatja, hogy a teljes állapotinformáció elvesztésével járó katasztrófák esetében megbízható FIFO kommunikáció

nem valósítható meg. A 22.5.2. szakaszban ugyanezt a megoldhatatlansági eredményt abban az esetben is megismételjük, amikor nem a teljes állapotinformáció veszik el. Végül a 22.5.3. szakaszban egy, a gyakorlatban kivitelezhető protokollt mutatunk, amely bizonyos katasztrófa és csatornahiba típusok esetében mégis képes megbízható FIFO kommunikációt biztosítani.

Az alfejezet további részében feltesszük, hogy a 22.4. ábrán látható módon minden egyes P_i folyamat rendelkezik egy KATASZTRÓFA$_i$ be- és egy ÚJRAÉLED$_i$ kimenettel, amelyek közül az utóbbi egy taszkot is magában foglal. A KATASZTRÓFA$_i$ bekövetkezése lehetővé tesz egy hozzá tartozó ÚJRAÉLED$_i$ lépést, és annak bekövetkeztéig egyben P_i minden további lépését meg is akadályozza, beleértve, hogy egyetlen bemenet (még a KATASZTRÓFA$_i$) sem lehet az automata állapotára hatással. A feltételek szerint minden pártatlan végrehajtási sorozat során a katasztrófát előbb-utóbb újraéledésnek kell követnie.



22.4.. ábra. A P_1 és P_2 folyamatok külső illesztőfelületeivel modellezhetjük a folyamatok katasztrófáit.

22.5.1.. Egy egyszerű megoldhatatlansági bizonyítás

Megmutatjuk, hogy amennyiben az ÚJRAÉLED$_i$ esemény a folyamatautomatát mindig a kezdőállapotba viszi, azaz a KATASZTRÓFA$_i$ és az azt követő ÚJRAÉLED$_i$ események során minden állapotinformáció elveszik, akkor megbízható FIFO kommunikáció nem valósítható meg. A bizonyítás egyszerű és meglepő módon még abban az esetben is érvényes, ha maga a csatorna megbízható FIFO tulajdonságú.

22.13. tétel. . *Amennyiben az* ÚJRAÉLED$_i$ *esemény a folyamatautomatát kezdőállapotba viszi, nem létezik olyan protokoll, amely az F megbízható FIFO kommunikációt megbízható FIFO csatornákon megvalósítja, azaz amelynek minden α pártatlan végrehajtási sorozatára teljesül, hogy $\alpha|$külső$(F) \in$ pártatlan_ történetek(F).*

Bizonyítás. Indirekt módon bizonyítunk: feltételezzük, hogy a keresett (P_1, P_2) protokoll létezik. A bizonyítás fő gondolata az, hogy P_2 egy katasztrófa után nem

tudja megállapítani, hogy az utolsó magas szintű üzenetet még éppen át tudta-e adni U_2-nek, vagy ebben már a katasztrófa megakadályozta.

Legyen α_1 a protokoll tetszőleges olyan pártatlan végrehajtási sorozata, amelyben csak egyetlen KÜLD esemény történik és nincs KATASZTRÓFA. A pártatlan történet feltétele szerint a KÜLD eseményt később egy hozzá tartozó FOGAD eseménynek kell követnie. Legyen α_2 és α_2' az α_1 FOGAD eseményt közvetlenül megelőző, illetve azzal végződő két kezdőszelete.

Legyen α_3 az a végrehajtási sorozat, amelyben az α_2-t egyetlen KATASZTRÓFA$_2$, majd ÚJRAÉLED$_2$ esemény követ. Az α_3 kiterjeszthető egy olyan α_4 pártatlan végrehajtási sorozattá, amely további KÜLD és KATASZTRÓFA eseményeket nem tartalmaz. Mivel α_4 is pártatlan, tartalmaznia kell egy, az egyetlen KÜLD eseményhez tartozó FOGAD eseményt; ez csak az ÚJRAÉLED$_2$ esemény után következhet.

Ellentmondásra egy α_5 pártatlan végrehajtási sorozat megadásával jutunk, amelyben az α_4-hez képest az egyetlen FOGAD esemény duplán szerepel, ellentmondva a pártatlan történet feltételének. Ebből a célból α_2'-t folytatjuk egy KATASZTRÓFA$_2$, majd ÚJRAÉLED$_2$ eseménypárral. Ekkor P_2 kezdőállapotban van, tehát az α_4-nek α_3-t követő szakaszával folytatható egy pártatlan α_5 végrehajtási sorozat, amely egy fölös FOGAD eseményt fog tartalmazni. □

Megjegyezzük, hogy a 22.13. tétel bizonyítása még abban az esetben is működik, ha minden KATASZTRÓFA eseményt közvetlenül követ egy ÚJRAÉLED, illetve ezen felül még a katasztrófák számának végessége is ki van kötve.

22.5.2.. Egy erősebb megoldhatatlansági bizonyítás

A 22.13. tétel megoldhatatlansági bizonyításának túlzott egyszerűsége alapján azt gondolhatjuk, hogy a tétel feltételeit túl erősnek választottuk, azaz túlzott követelményeket támasztottunk a keresett katasztrófatűrő protokollal szemben. Ebben a szakaszban azonban megmutatjuk, hogy még igen nagyfokú gyengítés mellett sem létezik a keresett tulajdonságú protokoll. A 22.13. tételnél erősebb eredményt fogunk mutatni, amelynek azonban a bizonyítása is jóval bonyolultabb lesz.

A katasztrófatűrő protokoll létezésének kérdését viszonylag enyhe feltételek mellett fogjuk vizsgálni. A katasztrófamodell változatlan marad, tehát a folyamat katasztrófája esetében minden állapotinformáció elveszik. Gyengébb követelményt fogunk azonban támasztani a külső csatolófelülettel szemben, mivel megengedjük, hogy bizonyos esetekben a protokoll üzenetet veszítsen. A csatornával szemben pedig viszonylag erős követelményt támasztunk. Az előző szakasszal ellentétben ugyan nem követeljük meg, hogy a csatorna megbízható FIFO tulajdonsággal rendelkezzen, azonban a hibák közül egyedül az üzenetvesztést engedélyezzük.

Megoldhatatlansági eredményünket a veszteséges FIFO kommunikációt megvalósító külső csatolófelület történeteit jellemző következő B tulajdonság esetére mondjuk ki. A (magas szintű) üzenetek többszörözését nem engedjük meg, a sorrendjük felcserélését azonban igen. Veszteséget is megengedünk: egy üzenet átadását csak abban az esetben követeljük meg, ha a KÜLD eseményt nem követi ÚJRAÉLED. Tehát a legutolsó ÚJRAÉLED előtti összes üzenet elveszhet, illetve ha

végtelen sok KATASZTRÓFA–ÚJRAÉLED eseménypár következik be, akkor akár az is előfordulhat, hogy egyetlen üzenet sem érkezik meg.

A következőkben olyan $Q_{1,2}$ és $Q_{2,1}$ csatornákat tekintünk, amelyeken az üzenetek többszörözése és a sorrend felcserélődése nem megengedett, a veszteségeket pedig az EVK feltétellel korlátozzuk. Azt mondjuk, hogy a protokoll egy végrehajtási sorozata *ellentmondásmentes* vagy *végesen ellentmondásmentes*, ha a végrehajtási sorozat mindkét $Q_{1,2}$ és $Q_{2,1}$ csatornára nézve rendelkezik a megfelelő tulajdonsággal.

A bizonyításban a $Q_{1,2}$ és $Q_{2,1}$ csatorna *be_ tranzit* állapotban levő üzeneteit fogjuk tekinteni. Mivel egyik csatorna sem többszöröz és nem cseréli fel az üzenetek sorrendjét, *be_ tranzit* állapotban pontosan az utolsó átvett alacsony szintű üzenet indítása óta indított üzenetek sorozata van (az indítást és az átvételt egymáshoz egy *ok* függvénnyel kapcsolhatjuk). Az üzenetek rögzített sorrendje miatt beszélhetünk tehát arról, hogy egy T üzenet*sorozat be_ tranzit* állapotban van, amely alatt azt értjük, hogy az időben rendezett *be_ tranzit* állapotú üzenetek részsorozatát alkotja. A veszteségeket is figyelembe véve a csatorna kimenetén mindig valamelyik T *be_ tranzit* állapotban levő sorozatot fogadhatjuk abban az esetben is, ha további `küld` események nem következnek be.

A következő tétel úgy foglalható röviden össze, hogy nem létezik olyan protokoll, amely veszteséges alacsony szintű csatornákat használva többszörözés és veszteség nélkül át tud minden, az összes katasztrófa és újraéledés lezajlása után indított üzenetet adni.

22.14. tétel. . *Amennyiben az* ÚJRAÉLED$_i$ *esemény a folyamatautomatát kezdőállapotba viszi, nem létezik olyan* (P_1, P_2) *protokoll a* $Q_{1,2}$ *és* $Q_{2,1}$ *veszteséges FIFO csatornákon, amelynek felhasználói felületi történeteit a* B *tulajdonság jellemzi, azaz amelynek minden pártatlan ellentmondásmentes* α *végrehajtási sorozatára* $\alpha|\mathit{külső}(B) \in \mathit{történetek}(B)$.

Fő gondolatunk az előző szakasz megoldhatatlansági tételéhez hasonlóan az, hogy egy katasztrófa bekövetkezése előtti eseményekre az újraéledés után már nem emlékezhetünk. Az előző szakaszban egy FOGAD esemény „elfeledése” vezetett ellentmondásra; a következő bizonyításban viszont egy KÜLD eseményé. A feltételezett protokollnak helyesen kellene működnie abban az esetben is, ha a KÜLD eseményt közvetlenül katasztrófa követi, azaz amennyiben az üzenetet tévedésből újra elküldjük, azt nem szabad másodszor átvenni. A bizonyítás során tehát olyan helyzetet fogunk előállítani, amelyben a fogadó oldali protokoll nem képes eldönteni, hogy az érkező üzenet egy katasztrófa következtében tévedésből indított másodpéldány, vagy egy újabb átveendő üzenet.

A fent vázolt gondolatmenetben az okozza a nehézséget, hogy a bizonyításnak alacsony szintű üzenetekből álló tetszőlegesen kifinomult nyugtázási rendszerekre is működnie kell. Ezért a protokollokat alacsonyabb szinten, lényegében lépésről lépésre fogjuk vizsgálni és illesztőfelületi lépéseit katasztrófák hozzáadásával módosítani. A kulcslépés, amelyet külön állításként fogalmazzuk meg, a feltételezett (P_1, P_2) protokoll tetszőleges α katasztrófát nem tartalmazó végesen ellentmondásmentes végrehajtási sorozatából indul ki. Ezt a végrehajtási sorozatát azután

katasztrófa-események hozzáadásával α'-vé egészíti ki úgy, hogy mindkét folyamat ugyanabban az állapotban legyen α és α' végrehajtása után, azonban az utóbbi esetben az egyik csatornairány *minden* α-ban elindított alacsony szintű üzenete *be_ tranzit* állapotban maradjon.

A bizonyításban a következő jelöléseket használjuk. Az $\bar{i}$ jelöli az i-vel ellentétes folyamat számát, azaz $\bar{1} = 2$ és $\bar{2} = 1$. Ha α a $P_1 \times P_2$ végesen ellentmondásmentes végrehajtási sorozata és $i \in \{1, 2\}$, akkor jelölje

- $be(\alpha, i)$ a P_i által α-ban fogadott alacsony szintű üzeneteket;
- $ki(\alpha, i)$ a P_i által α-ban küldött alacsony szintű üzeneteket;
- $státusz(\alpha, i)$ a P_i állapotát α végrehajtási sorozata után.

22.15. segédtétel. . *Tételezzük fel, hogy létezik a tételbeli (P_1, P_2) protokoll. Legyen α a protokoll olyan végesen ellentmondásmentes végrehajtási sorozata, amely* KATASZTRÓFA *eseményt nem tartalmaz. Legyen $i \in \{1, 2\}$. Tételezzük fel, hogy α vagy nem tartalmaz egyetlen lépést sem, vagy az utolsó lépése P_i-hez tartozik. Ebben az esetben a $P_1 \times P_2$ protokollhoz létezik olyan végesen ellentmondásmentes α' végrehajtási sorozat, amelyre a következő teljesül:*

1. *mindkét automata esetében az α és α' végrehajtási sorozata utáni állapot azonos, $státusz(\alpha, 1)$, illetve $státusz(\alpha, 2)$;*
2. *a P_i automata az utolsó* ÚJRAÉLED *esemény után az α minden illesztőfelületi lépését sorrendben végrehajtja; és*
3. *a $ki(\alpha, i)$ sorozat P_i-ből $P_{\bar{i}}$ felé be-tranzit állapotban van.*

Az α' tartalmazhat KATASZTRÓFA *és* ÚJRAÉLED *eseményeket, azonban minden* KATASZTRÓFA *után egy hozzá tartozó* ÚJRAÉLED *lépés következik.*

Bizonyítás. Az állítást egyszerre bizonyítjuk $i = 1$ és 2 esetében úgy, hogy indukciót alkalmazunk az α-beli lépések számára. Az indukció során az α végéről „levesszük" az utolsóként lépő P_i automata lépéseit, és $P_{\bar{i}}$ utolsó lépése után egy KATASZTRÓFA–ÚJRAÉLED párt illesztünk. A bizonyítás lényegi része annak megmutatása lesz, hogy ezután P_i lépéseit változatlanul hagyva megismételhetjük a $P_{\bar{i}}$ összes α-beli korábbi lépését, ezzel teljesítve a segédtétel mindhárom követelményét.

Az indukció kezdőlépéseként tekintsük azt az esetet, amikor rögzített i értékre α nem tartalmazza a $P_{\bar{i}}$ egyetlen lépését sem. Ekkor $\alpha' = \alpha$ választásával az állítás teljesül.

Az indukciós lépésben feltételezhetjük, hogy rögzített i mellett az α tartalmazza a $P_{\bar{i}}$ legalább egy lépését. Legyen α_1 az α végrehajtási sorozat $P_{\bar{i}}$ utolsó lépéséig terjedő kezdőszelete. Mivel α nem üres, az utolsó lépése P_i-hez tartozik és így α_1 valódi kezdőszelet. Mivel a $P_{\bar{i}}$-hoz α_1 folytatásában újabb lépés már nem tartozik, $státusz(\alpha_1, \bar{i}) = státusz(\alpha, \bar{i})$, valamint $be(\alpha, i)$ részsorozata a $ki(\alpha_1, \bar{i})$-nek.

Cseréljük most fel i és $\bar{i}$ szerepét és alkalmazzuk az indukciós feltevést az $\bar{i}$ értékére és az α_1 végrehajtási sorozatra. Ekkor létezik olyan α_1' végrehajtási sorozat, amelyben minden KATASZTRÓFA után egy hozzá tartozó ÚJRAÉLED lépés következik, és

1. mindkét automata α_1 és α_1' végrehajtási sorozata utáni állapota azonos, *státusz*$(\alpha_1, 1)$, illetve *státusz*$(\alpha_1, 2)$; valamint
2. a $ki(\alpha, \bar{i})$ sorozat $P_{\bar{i}}$-ből P_i felé be-tranzit állapotban van.

Mivel $be(\alpha, i)$ részsorozata a $ki(\alpha_1, \bar{i})$-nek, az is teljesül, hogy $be(\alpha, i)$ a $P_{\bar{i}}$-ből P_i felé be-tranzit állapotban van. A rendszer állapotának α_1' és α' közötti eltérését a 22.5. ábrán szemléltetjük.

Most megadjuk a keresett α' végrehajtási sorozatát úgy, hogy α_1'-t kiegészítjük a P_i további lépéseivel. Mivel a $P_{\bar{i}}$ állapotát már nem változtatjuk, az előbb látott indukciós feltevés szerint $P_{\bar{i}}$ teljesíti az α'-re tett követelményeket. Először vigyük P_i-t kezdőállapotba úgy, hogy az α_1'-höz adjunk egy `KATASZTRÓFA`$_i$ és `ÚJRAÉLED`$_i$ eseménypárt. Ezután P_i működését folytassuk úgy, hogy ismételtessük meg vele az α összes lépését. A csatornaillesztő műveletei megfelelő sorrendben rendelkezésre állnak, hiszen a P_i-hez érkező $be(\alpha, i)$ üzenetek az indukciós feltétel értelmében sorrendben mind be-tranzit állapotban vannak, a további szükségtelen üzeneteket pedig egyszerűen „elveszíthetjük" a veszteséges csatornán. Így P_i a szükséges $ki(\alpha, i)$-beli alacsony szintű üzeneteket a csatornába helyezheti, majd a kívánt végállapotba kerülhet. □

22.14. tétel bizonyítása. A segédtételt a következőképpen alkalmazzuk. Tekintünk egy egyetlen `KÜLD`$_1$ és hozzá tartozó `FOGAD`$_2$ párból álló α végrehajtási sorozatot, amely a `FOGAD`$_2$ művelettel ér véget. Előállítjuk a segédtételbeli α_1 végrehajtási sorozatot; ez a két automata végállapotát az α-val azonosan tartja, azonban a P_1 minden illesztőfelületi műveletét, így `KÜLD`$_1$-et is megismétli.

A feltételek szerint az α_1 pártatlan, ellentmondásmentes α_2 végrehajtási sorozattá egészíthető ki, amely nem tartalmaz további `KÜLD`, `KATASZTRÓFA` és `ÚJRAÉLED` eseményt és amelyről ráadásul feltehetjük, hogy α_1 után nem fogad α_1-ben indított üzenetet – hiszen ezek az üzenetek a csatornán elveszhetnek. Az α_2 a pártatlanság következtében egy további `FOGAD` eseményt tartalmaz.

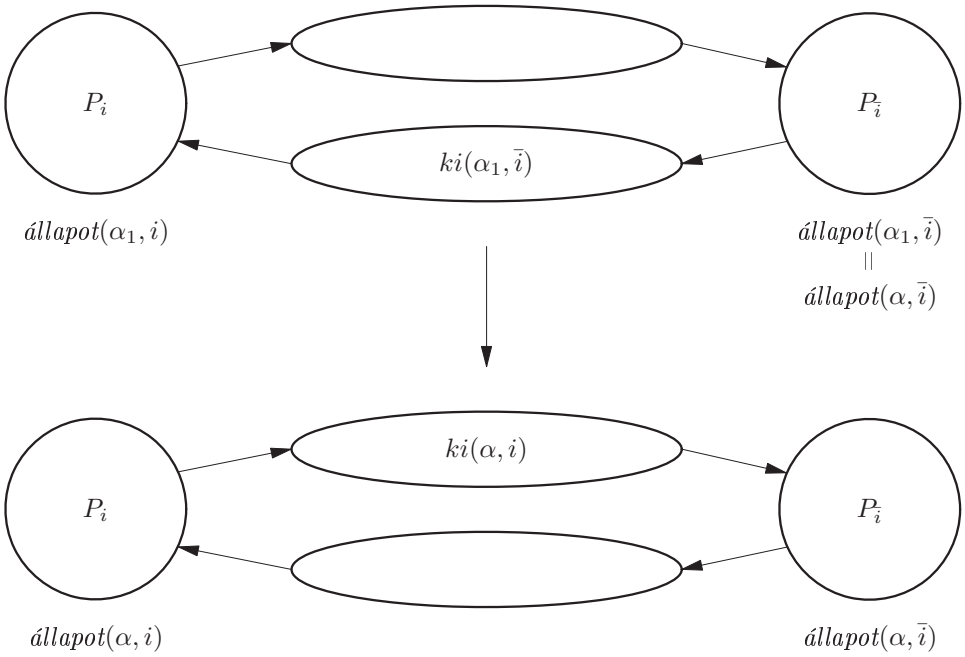
Ellentmondásra úgy juthatunk, hogy az α_2 sorozat α_1 utáni részét – mivel az automaták állapota azonos – az α után illesztjük. Az így kapott α_3 végrehajtási sorozat nem pártatlan, hiszen két `FOGAD` esemény mellett csak egy `KÜLD` eseményt tartalmaz. □

22.5.3.. Az ÖtCsomag protokoll

Mivel valós helyzetekben elkerülhetetlen a folyamatok időnkénti katasztrófája, nagy jelentősége van minden olyan protokollnak, amely az előző szakaszokban bemutatott negatív eredmények ellenére – bizonyos feltételekkel – katasztrófák esetében is megbízható kommunikációt biztosít. A következőkben bemutatunk egy ilyen protokollt, a TCP, az ISO TP-4 és több más adatátviteli protokoll alapját képező ÖtCsomag protokollt.

Az ÖtCsomag[3] protokollban úgy kerüljük meg az előző szakaszok megoldhatatlansági tételeit, hogy bizonyos információt stabil, nem illékony módon tá-

[3] A protokollban szereplő *csomag* szóra úgy gondolhatunk, mint az alacsony szintű üzenet szinonimájára.



22.5.. ábra. A rendszer állapotainak változása az α'_1-ből α'-be történő átmenet során.

rolunk. Fizikailag létező stabil memóriára azonban nem lesz szükség: stabil információként a rendszer által létrehozott egyedi üzenetazonosító (angol nevének rövidítése UID) fog szolgálni. Meg fogjuk követelni, hogy a protokoll működése során egy azonosítót legfeljebb egyszer generáljon és ismétlődés még akkor se történjen, ha katasztrófák állnak be. Így a kiosztásra kerülő azonosítók stabil adatok, hiszen mindig tudnunk kell, hogy egy adott érték szerepelt-e már.

A protokoll elméleti megvalósítása stabil memóriát igényel, amelyben a már kiosztott azonosítók *használt* halmazát tárolhatjuk és amelynek tartalma katasztrófa során is megőrződik. A gyakorlatban azonban számos módot ismerünk arra, hogy egyedi azonosítókat osszunk stabil memória igénybevétele nélkül – például véletlenszám-generátor vagy valós idejű óra segítségével.

Az ÖtCsomag protokoll a 22.5.2. szakasz B csatorna-tulajdonságát teljesíti, azaz üzeneteket nem többszöröz és az utolsó `ÚJRAÉLED` után indított minden üzenetet átad, sőt ezen túlmenően még az üzenetek sorrendjét sem cseréli meg. Az alacsony szintű üzenetek pedig igen gyenge megbízhatóságú csatornán haladhatnak, amelyen az üzenetek elvesztése, többszörözése és sorrendjük felcserélése is megengedett. Két enyhe kikötést kell csak tennünk: egyrészt csak véges többszörözést engedünk meg, másrészt megköveteljük az erős veszteségkorlátozás feltételét.

Az ÖtCsomag protokoll (vázlatosan)

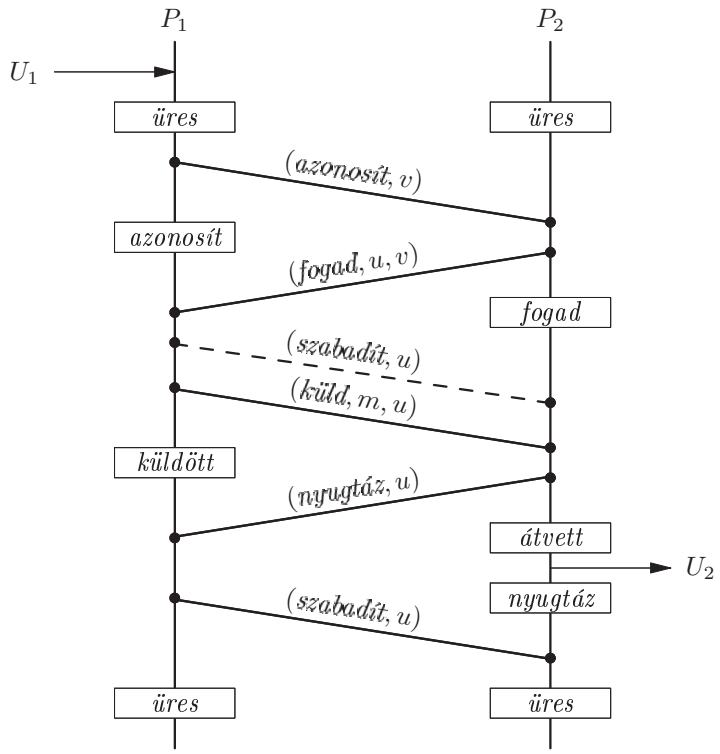
A P_1 folyamat az U_1 felhasználó által küldött magas szintű üzeneteket kívánja sorrendben P_2-höz továbbítani. Az üzenettovábbítást azonban megelőzi egy kétlépéses (alacsonyszintű) üzenetváltás, amelyben P_1 és P_2 megegyeznek az üzenethez rendelt egyedi azonosító értékében. Egészen pontosan *mindkét* automata egy-egy egyedi értéket rendel az elindítandó üzenethez, így a P_1-oldali katasztrófa sem okozhat többszöri indítást és a P_2-oldali sem többszöri átvételt. A magas szintű üzenet elindítása után ismét két lépésben a folyamatok az átvételt nyugtázzák és az azonosítókat felszabadítják. A protokoll lépéseit a 22.6. ábrán követhetjük.

Egy magas szintű üzenet indítása úgy történik, hogy *szabad* állapotban a P_1 elindítja a $(azonosít, v)$ üzenetet. Amennyiben P_2 is *szabad* állapotban van, a $(fogad, u, v)$ üzenettel válaszol és *fogad* állapotba lép. Az u és v értékek egyedi azonosítók, tehát a protokoll működése során korábban nem fordulhattak elő. Így ha P_1-hez válasz érkezik, biztos lehet benne, hogy az u az általa indított v-hez tartozik. Bármelyik oldalon a katasztrófa az u és v értékek elvesztéséhez vezet, így az u vagy v azonosítókkal ellátott üzenetek a későbbiekben csak akkor lehetnének hatással, ha az u vagy v értékeket az automaták újra választanák – ez azonban kikötésünk szerint nem következik be.

Miután az automaták az azonosítópárban megegyeztek, P_1 az m magas szintű üzenetet a $(küld, m, u)$ kiegészítő információkkal elindíthatja és *küldött* állapotba léphet. A *fogad* állapotban lévő P_2 először az m üzenetet az U_2 felhasználóhoz továbbítja és *átvett* állapotba lép, majd nyugtázza az átvételt és *nyugtáz* állapotba lép. A nyugtázásra válaszul a *küldött* állapotú P_1 az azonosítót felszabadítja, majd mindkét automata *szabad* állapotba kerül.

A fentiekben a protokoll normál működését ismertettük. Katasztrófákkal és üzenettöbbszörözéssel azonban könnyen elérhetjük, hogy a két automata eltérő u_1, u_2 vagy v_1, v_2 azonosítóval rendelkezzen. Ebben az esetben a következő kiegészítő lépésekre lesz szükség.

- Ha P_1 olyan $(fogad, u, v)$ üzenetet kap, amelyre $v \neq v_1$, akkor egy idő előtti $(szabadít, u)$ üzenettel válaszol. Ezt a P_2 úgy értelmezi, hogy az u, v azonosítópár a továbbiakban érvénytelen és *szabad* állapotból új üzenetváltás indul.
- Ha P_2 olyan $(küld, m, u)$ üzenetet kap, amelyre $u \neq u_1$, akkor az m magas szintű üzenetet nem adja át, azonban ekkor is nyugtázást küld. Ebben az esetben előfordulhat, hogy egy P_1 által nem észlelt üzenetvesztés történik.
- A két fenti eseten felüli minden nem megfelelő állapotban érkező vagy nem megfelelő azonosítóval rendelkező alacsony szintű üzenet figyelmen kívül hagyható.



22.6.. ábra. Az ÖtCsomag protokoll ötlépéses üzenetváltása és az idő előtti *szabadít* üzenet lehetséges helye (szaggatott vonal).

22.4. protokoll. ÖtCsomag

A P_1 folyamat
Lenyomat:
Bemeneti:
- KÜLD$(m)_{1,2}$, $m \in M$
- fogad$(fogad, u, v)_{2,1}$, u és v azonosítók
- fogad$(nyugtáz, u)_{2,1}$, u azonosító
- KATASZTRÓFA$_1$

Kimeneti:
- küld$(azonosít, v)_{1,2}$, v azonosító
- küld$(küld, m, u)_{1,2}$, $m \in M$, u azonosító
- küld$(szabadít, u)_{1,2}$, u azonosító
- ÚJRAÉLED$_1$

Belső:
- választ(v), v azonosító

A P_2 folyamat
Lenyomat:
Bemeneti:
- fogad$(azonosít, v)_{1,2}$, v azonosító
- fogad$(küld, m, u)_{1,2}$, $m \in M$, u azonosító
- fogad$(szabadít, u)_{1,2}$, u azonosító
- KATASZTRÓFA$_2$

Kimeneti:
- FOGAD$(m)_{1,2}$, $m \in M$
- küld$(fogad, u, v)_{2,1}$, u és v azonosítók
- küld$(nyugtáz, u)_{2,1}$, u azonosító

Állapotok:
$státusz_1 \in \{szabad, azonosít, küldött\}$,
kezdetben *szabad*
$puffer_1$, egy M elemeiből álló FIFO sor,
kezdetben üres
u_1 azonosító vagy *null*,
kezdőértéke *null*
v_1 azonosító vagy *null*,
kezdőértéke *null*
$használt_1$ azonosítóhalmaz,
kezdetben üres
$küld\_puffer_1$,
egy alacsony szintű üzenetekből
álló FIFO sor, kezdetben üres

Átmenetek:
KÜLD$(m)_{1,2}$
Hatás:
tegyük be m-et $puffer_1$-be

választ(v)
Előfeltétel:
$státusz_1 = szabad$
$puffer_1$ nem üres
$v \notin használt_1$
Hatás:
$v := v_1$
$használt := használt \cup \{v\}$
$státusz_1 := azonosít$

küld$(azonosít, v)_{1,2}$
Előfeltétel:
$státusz_1 = azonosít$
$v = v_1$
Hatás:
üres

fogad$(fogad, u, v)_{2,1}$
Hatás:
if $státusz_1 = azonosít$ **and** $v = v_1$
then
$u_1 := u$
$státusz_1 := küldött$
else
if $u \neq u_1$ **then**
$küld\_puffer_1 \leftarrow (szabadít, u)$

Állapotok:
$státusz_2 \in \{szabad, fogad, átvett, nyugtáz\}$,
kezdetben *szabad*
$puffer_2$, egy M elemeiből álló FIFO sor,
kezdetben üres
u_2 azonosító vagy *null*,
kezdőértéke *null*
v_2 azonosító vagy *null*,
kezdőértéke *null*
utolsó, azonosító vagy *null*,
kezdőértéke *null*
$használt_2$ azonosítóhalmaz, kezdetben üres
$küld\_puffer_2$, egy alacsony szintű üzenetekből
álló FIFO sor, kezdetben üres

Átmenetek:
FOGAD$(m)_{1,2}$
Előfeltétel:
$státusz_2 = átvett$
m a $puffer_2$ első eleme
Hatás:
távolítsuk el a $puffer_2$ első elemét
$státusz_2 := nyugtáz$

fogad$(azonosít, v)_{1,2}$
Hatás:
if $státusz_2 = szabad$ **then**
$u_2 :=$ tetsz. azonosító $\notin használt_2$
$használt_2 := használt_2 \cup \{u_2\}$
$státusz_2 := fogad$
$v_2 = v$

küld$(fogad, u, v)_{2,1}$
Előfeltétel:
$státusz_2 = fogad$
$v = v_1$
$u = u_1$
Hatás:
üres

küld($küld, m, u$)$_{1,2}$
Előfeltétel:
$státusz_1 = küldött$
a $puffer_1$ első eleme m
$u = u_1$
Hatás:
üres

fogad($nyugtáz, u$)$_{2,1}$
Hatás:
if $státusz_1 = küldött$ **and** $u = u_1$
then
távolítsuk el
$puffer_1$ első elemét
$v_1 := null$
$u_1 := null$
$státusz_1 := szabad$
$küld\_puffer_1 \leftarrow (szabadít, u)$

küld($szabadít, u$)$_{1,2}$
Előfeltétel:
$küld\_puffer_1$ első eleme $(szabadít, u)$
Hatás:
távolítsuk el $küld\_puffer_1$ első elemét

KATASZTRÓFA$_1$, majd ÚJRAÉLED$_1$
Hatás:
$puffer_1 :=$ üres
$v_1 := null$
$u_1 := null$
$küld\_puffer_1 :=$ üres
$státusz_1 := szabad$

Taszkok:
{küld($azonosít, v$)$_{1,2} : v$ azonosító}
{küld($küld, m, u$)$_{1,2} : m \in M, u$ azonosító}
{küld($szabadít, u$)$_{1,2} : u$ azonosító}
ÚJRAÉLED$_1$
{választ(v) : v azonosító}

fogad($küld, m, u$)$_{1,2}$
Hatás:
if $státusz_2 = fogad$ **and** $u = u_1$
then
$puffer_2 \leftarrow m$
$utolsó := u$
$státusz_2 := átvett$
else
if $u \neq utolsó$ **then**
$küld\_puffer_2 \leftarrow (nyugtáz, u)$

küld($nyugtáz, u$)$_{2,1}$
Előfeltétel:
$státusz_2 = nyugtáz$ **and** $utolsó = u$
Hatás:
üres
vagy
Előfeltétel:
$küld\_puffer_2$ első eleme $(nyugtáz, u)$
Hatás:
távolítsuk el $küld\_puffer_2$ első elemét

fogad($szabadít, u$)$_{1,2}$
Hatás:
if $(státusz_2 = fogad$ **and** $u = u_2)$
or
$(státusz_2 = nyugtáz$ **and** $u = utolsó)$
then
$v_2 := null$
$u_2 := null$
$utolsó := null$
$státusz_2 := szabad$
tegyünk $(szabadít, u)$ üzenetet
$küld\_puffer_2$-be

KATASZTRÓFA$_2$, majd ÚJRAÉLED$_2$
Hatás:
$puffer_2 :=$ üres
$u_2 := null$
$v_2 := null$
$utolsó := null$
$küld\_puffer_2 :=$ üres
$státusz_2 := szabad$

Taszkok:
{FOGAD(m)$_{1,2} : m \in M$}
{küld($fogad, u, v$)$_{2,1} : u, v$ azonosítók}
{küld($nyugtáz, u$)$_{2,1} : u$ azonosító}
ÚJRAÉLED$_2$

Most rátérünk a protokoll helyességének bizonyítására. hogy az erős veszteségkorlátozás tulajdonságot teljesítő veszteséges, átrendező és az üzenetek vé-

ges többszörözését megengedő csatornákat használva az ÖtCsomag protokoll a 22.5.2. szakaszbeli B csatornatulajdonságot teljesíti. A protokoll során veszteség előfordulhat – például a P_2 katasztrófájával az u_2 értéke elveszik, ám az érkező $(\textit{küld}, m, u)$ üzenetet P_2-nek nyugtáznia kell anélkül, hogy azt a felhasználónak átadná. A B csatorna az üzenetek átrendezését és többszörözését nem engedi meg, és a veszteség is csak abban az esetben megengedett, ha a `KÜLD` esemény után katasztrófa áll be.

22.16. tétel. . *Az* ÖtCsomag *protokoll külső csatolófelülete a B tulajdonsággal rendelkezik, amennyiben a protokollt tetszőleges, az üzeneteket végesen többszöröző és az EVK feltételt teljesítő csatornán alkalmazzuk. Azaz tetszőleges pártatlan α végrehajtási sorozat esetében $\alpha|\textit{külső}(B) \in \textit{történetek}(B)$.*

Bizonyításvázlat. Könnyen bizonyítható, hogy a protokoll az üzeneteket nem többszörözi és sorrendjüket nem rendezi át. Nehézséget mindössze a protokoll élénkségének igazolása jelent, amelyhez meg kell mutatnunk, hogy a protokoll minden esetben képes újabb magas szintű üzenetet átadni.

Először azzal az esettel foglalkozunk, amikor $\textit{státusz}_1 = \textit{azonosít}$ és $\textit{státusz}_2 = \textit{fogad}$, de $v_1 \neq v_2$. Ebben az esetben P_1 folyamatosan $(\textit{azonosít}, v)$ üzeneteket indít, ám ezeket P_2 figyelmen kívül hagyja. Meg kell tehát mutatnunk, hogy egy idő után a v_2 érték felszabadul és P_2 válaszolni tud az érkező üzenetekre. Indirekt módon tételezzük fel ennek ellenkezőjét. Ekkor a pártatlanságból következik, hogy P_2 végtelen sok $(\textit{fogad}, u, v_2)$ üzenetet küld, amelyek közül az erős veszteségkorlátozás értelmében P_1-hez végtelen sok meg is érkezik. Ezekre az üzenetekre P_1 az idő előtti $(\textit{szabadít}, u)$ küldésével válaszol, amelyek az erős veszteségkorlátozás értelmében egy idő után a v_2 értéket felszabadítják.

Az előző lépésben azt igazoltuk, hogy a P_2-nél található téves v_2 érték előbb-utóbb felszabadul; nem biztos azonban, hogy az új v_2 érték a P_1-nél tárolt v_1-gyel egyezni fog. Előfordulhat ugyanis, hogy P_2-höz nem az utoljára indított, hanem egy korábbi $(\textit{azonosít}, v)$ üzenet érkezik. Ebben az esetben fel kell használnunk, hogy a csatorna az üzeneteket csak végesen többszörözi. Mivel csak az utolsó `KATASZTRÓFA`–`ÚJRAÉLED` pár utáni üzenetek átadásával kell foglalkoznunk, feltételezhetjük, hogy v_1 értéke változatlan. Így a korábban indított $(\textit{azonosít}, v)$ üzenetek összesen is csak véges sok másodpéldánya egy idő után elfogy és az új v_2 érték ezután már csak v_1-gyel azonos lehet. □

Stabilizálódási tulajdonság. . Az ÖtCsomag protokoll hasznos tulajdonsága a gyakorlatban, hogy a felhasználói felületen utoljára bekövetkezett eseményt (`KÜLD`, `KATASZTRÓFA` vagy `ÚJRAÉLED`) követően egy idő után az automaták állapota stabilizálódik – a 22.16. tétel bizonyításában látott módon felszabadul minden folyamatban levő azonosító – és a *használt* halmazok kivételével további információ tárolása szükségtelen. Ezt a stabilizálódási tulajdonságot úgy is megfogalmazhatjuk, hogy a protokoll egy idő után „elfelejt” minden korábban történt eseményt. A protokoll tehát csak addig igényel tárterületet, amíg a két felhasználója között üzenetváltás folyik. Így egy folyamatpár a hálózat nagy számú felhasználója között párhuzamosan is képes arra, hogy az ÖtCsomag protokoll

lépéseit szimulálja[4]. Mindaddig, amíg az aktívan kommunikáló (U_1, U_2) párok száma alacsony, a teljes tárigény kellően alacsony marad.

Korlátos azonosítók. . Az ÖTCSOMAG protokoll alkalmazhatóságát elvileg korlátozza az a követelmény, hogy minden üzenet más azonosítót kap, így tetszőlegesen sok üzenet indítása korlátlanul nagy azonosítókat igényel. A gyakorlatban az azonosítók méretét például úgy lehet korlátozni, hogy azokat modulo valamilyen kellően nagy egész n tekintjük. A protokoll ilyen módon „végessé" tett változata elvileg hibázhat, mivel az azonosítók n értékén „átfordulva" újra felveszik a korábban már használt értékeket. Ez nem okoz problémát abban az esetben, ha a korábban indított azonos értékű üzenetek az átfordulás pillanatáig a rendszerből távoznak; a protokoll helyessége ekkor például az eredeti protokollra vonatkozó szimulációs relációval bizonyítható. A sokkal régebben indított üzenetek távozását egyrészt a hálózat gyakorlati tulajdonságainak – például az üzenet-továbbítási, feldolgozási idők és a magas szintű üzenetek indítási gyakoriságának – ismeretében egyrészt fel is tételezhetjük. Másrészt kiegészíthetjük a protokollt azzal, hogy ne vegye figyelembe a nagyon régen indított üzeneteket, és így biztosíthatjuk az azonosítók korlátosságát.

22.6.. Megjegyzések a fejezethez

Az ISO többszintű kommunikációs hierarchiát a [54, 290, 273] művek írják le. A STENNING protokoll N. Stenningtől [270] származik. A BITVÁLTÓ protokoll (BVP) először Bartlett, Scantlebury és Wilkenson [42] cikkében található meg. Amellett, hogy a protokoll önmagában is érdekes és jól alkalmazható, alappéldául szolgál különböző protokollvizsgálati módszerek bemutatására [177, 59, 229, 38, 146, 260, 280].

Az átrendező és többszöröző csatorna megoldhatatlansági bizonyítását Wang és Zuck [284] alapján ismertettük. A PRÓBA protokollt Afek és Gafni [5] munkája alapján Afek, Attiya, Fekete, Fischer, Lynch Mansour, Wang és Zuck [4] fejlesztették ki. Ezt megelőzően egy hasonló tulajdonságokkal bíró protokollt mutatott Attiya, Fischer, Wang és Zuck [23], amely azonban a PRÓBA protokoll többszintű felépítésével nem rendelkezett. A 22.11. tételt Afek és társai [4] bizonyították be. Hasonló megoldhatatlansági bizonyításokat adott még Mansour és Schieber [220], Wang és Zuck [284], valamint Tempero és Ladner [277, 278].

A katasztrófák esetére vonatkozó 22.14. erős megoldhatatlansági tételt egymástól függetlenül Lynch, Mansour és Fekete [206], illetve Spinelli [268] bizonyították; a két bizonyítás gondolatait egyesíti a [112] cikk. Spinelli [268] további megoldhatósági és megoldhatatlansági tételeket is igazolt. Baratz és Segall [41] mutatta meg, hogy katasztrófák esetében már egészen kis mennyiségű stabil memória alkalmazásával is megbízható kommunikáció valósítható meg; ugyanők fogalmazták meg azt a sejtést, hogy stabil memória nélkül megbízható kommunikáció nem jöhet létre, amelyet aztán Attiya, Dolev és Welch [21] bizonyítottak.

Az ÖTCSOMAG protokoll egyike a Belsnes [44] által kifejlesztett protokollok

[4] A szimuláció az egyes folyamatok b/k automata összekapcsolásaként történhet

sorozatának, amelyek egyre több üzenet váltásával egyre erősebb megbízhatósági és hibatűrési tulajdonságokat érnek el. Az ÖtCsomag a TCP, az ISO TP-4 és több más adatátviteli protokoll alapját is képező szabványos hálózati protokoll. Lampson, Lynch és Søgaard-Andersen [188, 190, 264] igazolja, hogy a protokoll egy általánosított változata, illetve egy hasonló, időzítést is alkalmazó protokoll helyes.

22.7.. Gyakorlatok

22-1. Igazoljuk a 22.1. lemmát.

22-2. Bizonyítsuk be, hogy a 14.1.2. példa $A_{1,2}$ és $A_{2,1}$ csatornáit alkalmazó Stenning protokoll és az F univerzális megbízható FIFO csatorna között létezik szimulációs reláció. Az f reláció úgy definiálható, hogy ha s a Stenning protokoll és u az F egy-egy állapota, akkor $(s, u) \in f$ pontosan akkor, ha

(a) az $s.címke_1 = s.címke_2$ egyenlősége esetében az $u.sor$ előáll úgy, hogy az $s.puffer_1$ első elemének[5] elhagyásával keletkező elemeket $s.puffer_2$ elemei mögé illesztjük; és

(b) a címkék különbözősége esetében pedig az $u.sor$ úgy áll elő, hogy az $s.puffer_1$ elemeit $s.puffer_2$ mögé illesztjük.

22-3. A 22-2. gyakorlat eredményét alkalmazva igazoljuk a 22.2. lemmát. A pártatlanság bizonyításakor alkalmazzuk a szimulációs relációt és a végrehajtások megfeleltetését.

22-4. Igazoljuk a 22.3. tételt.

22-5. Módosítsuk a 22.2. alfejezetben látott csatornákat annyiban, hogy engedjük meg az üzenetek végtelen többszörözését, azaz töröljük a 14.1.2. példában bemutatott A automata programkódjából a `küld` üzenet hatására tett végességi korlátozást.

(a) Bizonyítsuk be, hogy a Stenning protokoll a fent ismertetett csatornán hibásan működik, mivel nem teljesíti a megbízható FIFO csatornák pártatlansági feltételét.

(b) Mutassuk meg, hogy a csatorna élénkségi feltételének erősítésével megjavítható a protokoll működése.

22-6. Igazoljuk a 22.4. lemmát.

22-7. Bizonyítsuk be, hogy a 22.5. lemma bizonyításában alkalmazott f valóban szimulációs reláció.

22-8. Igazoljuk a 22.7. tételt.

22-9. Módosítsuk a BitVáltó protokollt úgy, hogy modulo 2 helyett egy rögzített $k > 2$ érték mellett modulo k egész címkéket alkalmazzon, és az eredetihez képest változatlan (FIFO) csatornákon megbízható FIFO kommunikációt valósítson meg. A módosított protokollban engedjük meg, hogy P_1 a $puffer_1$ első p

[5] A 22.1. lemma alapján $s.puffer_1$ ekkor nem lehet üres.

elemét sorban még azelőtt elindítsa, hogy az első üzenet nyugtája visszaérkezne. A k függvényében adjuk meg p lehetséges legnagyobb értékét.

22-10. Mutassuk meg, hogy a BVP abban az esetben is működik, ha megengedjük, hogy a csatornák az üzeneteket végtelen példányban többszörözzék. Az üzenetek sorrendjének átrendezése változatlanul nem megengedett és a veszteségeket a GyVK feltétellel korlátozzuk.

22-11. Igaz marad-e a 22.8. tétel, ha a csatornák az alacsony szintű üzeneteket legfeljebb k példányban többszörözhetik? Feltételezhetjük, hogy k értéke ismert, továbbá a csatornák üzeneteket nem veszítenek, de a sorrendjüket tetszőlegesen átrendezhetik. Adjunk vagy megoldhatatlansági bizonyítást, vagy egy algoritmust.

22-12. Egészítsük ki a 22.9. lemma bizonyításából hiányzó részeket. Adjuk meg az *ok* függvény definícióját és mutassuk meg, hogy a K csatornára szabott követelményeket teljesíti.

22-13. Helyes marad-e a PRÓBA protokoll, ha a $\texttt{fogad}(m)_{1,2}$ esemény hatását a *folyamatban* := *folyamatban* − 1 sorral egészítjük ki? Adjunk helyességi bizonyítást vagy ellenpéldát.

22-14. Bizonyítsuk be azt a 22.12. segédtétel bizonyításában felhasznált állítást, hogy tetszőleges végesen ellentmondásmentes $P_1 \times P_2$ végrehajtási sorozat kiegészíthető olyan pártatlan, ellentmondásmentes végrehajtási sorozattá, amely nem tartalmaz új $\texttt{KÜLD}_{1,2}$ eseményt és új $\texttt{fogad}_{1,2}$ esemény oka csak új $\texttt{küld}_{1,2}$ esemény lehet.

22-15. Egészítsük ki a 22.11. tétel eredményét az egymás után következő magas szintű üzenetek átadásához legjobb esetben szükséges alacsony szintű üzenetek számára vonatkozó alsó korláttal.

22-16. Engedjük meg, hogy a 22.5.2. szakasz B követelményeit gyengítve még az utolsó ÚJRAÉLED esemény után indított első legfeljebb k magas szintű üzenet is elveszhessen. Amennyiben véges sok KATASZTRÓFA és ÚJRAÉLED történik, az utolsó ÚJRAÉLED utáni első k darabot követően azonban magas szintű üzenet ne veszhessen el. A B-re tett további követelmények maradjanak változatlanul, azaz legyen az átrendezés megengedett, de a többszörözés nem. Vagy adjunk meg olyan protokollt, amely a 22.5.2. szakaszbeli $Q_{1,2}$ és $Q_{2,1}$ csatornákat alkalmazva teljesíti a gyengített B csatorna-előírást, vagy egészítsük ki erre az esetre a 22.14. tétel megoldhatatlansági bizonyítását.

22-17. *Kutatási feladat.* A 22-16. gyakorlat kérdését válaszoljuk meg abban az esetben, ha a B követelményeit még tovább gyengítjük és csak azt követeljük meg, hogy az utolsó ÚJRAÉLED eseményt követően *egy idő után* minden üzenet megérkezzen.

22-18. Bizonyítsuk be, hogy az ÖTCSOMAG protokoll üzenetet nem veszít és a sorrendet nem rendezi át.

22-19. Adjuk meg az ÖTCSOMAG protokoll olyan végrehajtási sorozatát, amelyben szükséges, hogy P_2 egy korábban indított $u \neq u_2$ azonosítójú $(\mathit{küld}, m, u)$ üzenetre $(\mathit{nyugtáz}, u)$ üzenettel válaszoljon.

22-20. Adjunk teljes bizonyítást arra az állításra, hogy az ÖTCSOMAG protokoll teljesíti az élénkségi feltételeket, azaz az utolsó `ÚJRAÉLED` esemény után indított minden magas szintű üzenet előbb-utóbb megérkezik P_2-höz.

22-21. Tekintsük az ÖTCSOMAG protokollt abban az esetben, amikor véges sok `KÜLD`, `KATASZTRÓFA` és `ÚJRAÉLED` esemény következik be. Bizonyítsuk be, hogy ebben az esetben mindkét folyamat állandó állapotban marad, amely a kezdőállapottól legfeljebb a *használt* halmazok tartalmában térhet el.

22-22. Tervezzünk két felhasználó között megbízható FIFO kommunikációt megvalósító algoritmust, amely tetszőleges irányítatlan hálózaton üzemel. A hálózat minden csúcsán egy folyamatautomata kapcsolódjon az éleken elhelyezkedő kétirányú megbízható FIFO csatornákhoz.

IV. RÉSZBEN SZINKRON ALGORITMUSOK

A könyv befejező része a 23–25. fejezetekből áll. Ezek a fejezetek algoritmusokat és alsó korlát eredményeket tartalmaznak *részben szinkron modellre*, ahol a rendszer összetevőinek van némi információjuk az időzítésről, de az információ nem teljes, mint a szinkron modellben. Az ilyen részleges információ az időzítési ismeretek olyan valóságos modelljét eredményezheti, amely megfelel a létező osztott rendszereknek.

A szokásnak megfelelően az első, 23. fejezet a formális modell leírását tartalmazza. Ezt követi két algoritmus fejezet, a 24. a *kölcsönös kizárásra* részben szinkron közös memóriájú rendszerekre, a 25. pedig a *megegyezésre* részben szinkron hálózati rendszerekben. Ezek a fejezetek a kezdetét jelentik valaminek, ami várhatóan érdekes új része lehet az osztott algoritmusok elméletének.

23. fejezet

Modellezés/V. Részben szinkron rendszerek modelljei

A könyv hátralévő három fejezete rövid bevezetését adja a *részben szinkron*, vagy másképpen *időzítés-alapú* osztott algoritmusok elméletének. Emlékeztetünk, hogy az I. rész (a 2–7. fejezetek) a *szinkron* osztott algoritmusokat vizsgálta, míg a II. rész (a 8–22. fejezetek) az *aszinkron* osztott algoritmusokkal foglalkozott. Kiderült, hogy van egy érdekes osztálya a modelleknek és algoritmusoknak a két szélsőséges eset között, amit mi *részben szinkronizáltnak* nevezünk. Részben szinkronizált rendszerben az összetevőknek van némi információjuk az időzítésről, ámbár az információ nem pontos. Például, részben szinkronizált hálózatok folyamatai hozzáférhetnek majdnem szinkronizált órákhoz, vagy ismerhetik a folyamat lépéseinek vagy az üzenetküldéseinek az időbeli korlátait.

A részben szinkron modellek minden bizonnyal valósághűbb modellek, mint akár a teljesen szinkron, akár a teljesen aszinkron modellek, mivel a valóságos rendszerek tipikusan használnak időzítési információt. Azonban a részben szinkron rendszerek elmélete közel sem olyan jól kidolgozott, mint a szinkron és az aszinkron rendszereké. Az itt tárgyalt elképzelések csak a kezdetét jelentik azon érdekes kutatásoknak, amelyek megítélésünk szerint az időzítés-alapú számítások megalapozását képezik.

Ebben a fejezetben bevezetését adjuk az időzítés-alapú osztott algoritmusok modelljeinek és bizonyítási módszereinek. A 23.1. alfejezetben az *időzített automata* modell bevezetésével kezdjük, amit *MMT modellnek* nevezünk a megalkotóikról, akik Merritt, Modugno és Tuttle. Az MMT modell egyszerű változata a b/k automata modellnek, és alkalmas a legtöbb időzítés-alapú algoritmus modellezésére. Bizonyos alapvető bizonyítási módszerek – különösen az invariáns állítások és a szimuláció – ezen modellben való felhasználhatósága miatt hasznosnak bizonyul, hogy minden MMT automatát át tudunk alakítani egy másik típusú automatává, amit *általános időzített automatának* nevezünk (*GTA*). A GTA modellt a 23.2. alfejezetben vezetjük be, és itt adjuk meg az MMT automatának GTA automatává transzformálását. A 23.3. alfejezetben ezen modellben használható bizonyítási technikákat vizsgálunk.

A 24. és 25. fejezetekben kezdeti eredményeket adunk a kölcsönös kizárás és

a megegyezés problémákra részben szinkron rendszerekben.

23.1.. MMT időzített automata

Az MMT időzített automata modellt egyszerűen úgy kapjuk a b/k automata modellből, hogy a helyesség feltételt helyettesítjük az időre kirótt alsó és felső korláttal. Megjegyzendő, hogy ha a helyesség feltételt csak felső korláttal helyettesítenénk, akkor a modell nem eredményezne semmi érdekes új képességet, mert egyedül a felső korlát nem korlátozná a b/k automata lehetséges végrehajtási sorozatainak halmazát. (Valójában a könyv „szinkron" fejezeteiben már ténylegesen hozzárendeltünk az algoritmusok taszkjaihoz felső korlátokat, hogy elemezni tudjuk az időbonyolultságot. Ezen elemzések hasznossága azon múlik, hogy a korlátok nem befolyásolják az algoritmusok viselkedését.) Azonban, mind alsó, mind felső korlátok bevezetése több képességet eredményez, mivel lehetővé teszik számunkra a végrehajtási sorozatok halmazának szűkítését. Valóban, sok időzítésalapú algoritmus helyessége nagymértékben a végrehajtási sorozatok megszorításától függ, amit az időkorlátok eredményeznek.

23.1.1.. Alapfogalmak

Induljunk ki egy olyan A b/k automatából, amelynek csak véges sok taszkja van. Az A automata b *korlátfüggvénye* egy *alsó-felső* függvénypár, amelyek minden taszkhoz alsó, illetve felső korlátot rendelnek. Minden C taszkra megköveteljük, hogy *alsó*(C) és *felső*(C) teljesítse a $0 \leq$ *alsó*$(C) < \infty$, $0 <$ *felső*$(C) \leq \infty$ és *alsó*$(C) \leq$ *felső*(C) egyenlőtlenségeket. Tehát az alsó korlát nem lehet ∞ és a felső korlát nem lehet 0 és az alsó korlát nem lehet nagyobb, mint a felső korlát. Egy *MMT automata* egy A b/k automata az A-hoz rendelt korlátfüggvénnyel .

Most definiáljuk az MMT automata végrehajtási sorozatait. Egy $B = (A, b)$ MMT automata *időzített végrehajtási sorozata* olyan $\alpha = s_0, (\pi_1, t_1), s_1, (\pi_2, t_2), \ldots, (\pi_r, t_r), s_r$ véges vagy $\alpha = s_0, (\pi_1, t_1), s_1, (\pi_2, t_2), \ldots, (\pi_r, t_r), s_r, \ldots$ végtelen sorozat, ahol az s-ek az A b/k automata állapotai, a π-k az A műveletei és a t-k időértékek $\mathbb{R}^{\geq 0}$-ből. Megköveteljük, hogy az $s_0, \pi_1, s_1, \cdots$ sorozat – tehát amit az α-ból úgy kapunk, hogy elhagyjuk az idő értékeket – közönséges végrehajtási sorozata legyen az A b/k automatának. Megköveteljük továbbá, hogy az α sorozatban a t_r idő értékek nemcsökkenő sorozatot alkossanak, és teljesítsék a b korlátfüggvény által kifejezett alsó korlát és felső korlát feltételt.

Mit jelent az, hogy α kielégíti az alsó és felső korlát feltételt? Ahhoz, hogy ezt formálisan meg tudjuk adni, definiáljuk a C taszk r *kezdeti indexét* úgy, hogy C végrehajtható az s_r állapotban, és az alábbi feltételek valamelyike igaz:

1. $r = 0$;
2. C nem végrehajtható az s_{r-1} állapotban,
3. $\pi_r \in C$.

A kezdeti index olyan időpontot jelöl, amelytől kezdve mérjük az időkorlátokat. A C taszk minden r kezdeti indexére teljesülnie kell a következő két feltételnek.

Felső korlát. Ha van olyan $k > r$, hogy $t_k > t_r + \mathit{felső}(C)$, akkor létezik olyan $k' > r$, hogy $t_{k'} \leq t_r + \mathit{felső}(C)$ és vagy $\pi_{k'} \in C$ vagy C nem végrehajtható $s_{k'}$-ben.

Alsó korlát. Nem létezik olyan $k > r$, amelyre $t_k < t_r + \mathit{alsó}(C)$ és $\pi_k \in C$.

A felső korlát feltétel azt fejezi ki, hogy a C taszk minden r kezdeti indexére teljesül, hogy ha az idő túlhalad C felső korlátján, akkor közben vagy C valamely művelete végrehajtódik, vagy C nem lesz végrehajtható. Az alsó korlát feltétel azt fejezi ki, hogy a C taszk minden r kezdeti indexére az alsó korlát letelte előtt C egyetlen művelete sem hajtódik végre.

Jelölje *idő_végrehajtási sorozatok*(B) a B automata időzített végrehajtási sorozatainak halmazát. Egy állapotot *elérhetőnek* nevezünk B-ben, ha végső állapota a B valamely időzített végrehajtási sorozatának.

Az alsó és felső korlát feltételek biztonságossági feltételek. Egy alapvető élénkségi feltétel is érdekel bennünket: azt mondjuk, hogy egy időzített végrehajtási sorozat *megengedett*, ha a következő feltétel teljesül.

Megengedhetőség. Ha az α időzített végrehajtási sorozat végtelen, akkor az időértékek sorozata ∞-hez tart. Ha α véges sorozat és végső állapota s, akkor minden s-ben végrehajtható C taszkra $\mathit{felső}(C) = \infty$.

A megengedhetőség feltétel azt fejezi ki, hogy az idő normálisan halad és a feldolgozás nem ér véget, ha az automatának még további munka elvégzése elő van írva. Jelölje $m\_i\_\mathit{végre}(B)$ a B automata megengedett időzített végrehajtási sorozatainak halmazát. Ebben a könyvben főleg megengedett időzített végrehajtási sorozatokra összpontosítunk.

Vegyük észre, hogy megengedett időzített végrehajtási sorozat esetében ha egy C taszk *felső* korlátja ∞, akkor semmi kényszer nincs arra vonatkozóan, hogy C valamely művelete valaha is bekövetkezzen. Ez némiképpen eltér attól, amit az aszinkron fejezetekben tettünk. A 8.6. alfejezetben az időzített végrehajtási sorozat egy másik definícióját adtuk azzal, hogy megköveteltük, hogy minden taszk kielégítse a pártatlanság feltételt, *továbbá* hogy valamelyikre teljesüljenek a felső korlátok. Ezt a kombinált feltételt az időbonyolultság elemzésére használtuk. Most teljesen ejtjük a pártatlanság feltételt és csak az időkorlátokat alkalmazzuk. Definiálhatnánk az MMT modell olyan változatát, ahol bizonyos taszkokra időkorlátot, bizonyosakra pedig pártatlanság feltételt szabnánk, de ezt formálisan nem adjuk meg a könyvben. Valójában informálisan tárgyaljuk majd az időkorlát és a pártatlanság feltételek kombinációját olyan algoritmusoknál, ahol ez előfordul.

Valamilyen megengedhetőségi feltétel szükséges minden hasznos időzítésalapú számítási modellben, hogy kizárhassunk néhány nagyon különös viselkedést, mint például amikor az automata véges időn belül végtelen sok kimenetet bocsát ki.[1] Habár formálisan értelmezhetőek az ilyen számítások, a valóságban

[1] Ezt a viselkedést néha Zénó viselkedésnek nevezik, utalva a Zénó-féle paradoxonra. A Zénó paradoxonban a futó Achillész végtelen sok lépést tesz, egyre kisebbeket, egyre jobban megközelítve célját (a teknősbékát), de azt sohasem éri el.

értelmetlenek és nehéz ilyenekről gondolkodni. Az időzítés-alapú rendszerek jó modelljének lehetővé kell tennie az ilyen esetek elkerülését.

Az MMT automata külső viselkedésének leírására bevezetjük az *időzített történet* fogalmát. Jelölje *id_ történet*(α) a B automata α időzített végrehajtási sorozati sorozatához tartozó *időzített történetét*, ami az α azon részsorozata, amely csak a külső műveleteket a hozzájuk tartozó idővel párosítva tartalmazza. A *megengedett időzített történet* az olyan időzített történet, amely megengedett végrehajtási sorozatból kapható, ezek halmazát *m_ i_ történet*(B) jelölje.

Az MMT automata az időzítés-alapú rendszerek sokféle komponensének leírására alkalmazható. Különösen alkalmas számítógép-rendszerek alacsony szintjének modellezésére, mivel a taszkrendszer és a taszkok időkorlátai természetes módját adják a fizikai rendszer-komponensek modellezésének és sebességük vizsgálatának. Kevéssé alkalmasak azonban a rendszerek magasabb szintjeinek leírására, vagy helyességi feltételek megadására. Ennek az oka, hogy a meglehetősen stilizált taszk és korlát fogalom nem mindig a legjobb „nyelv" a megkövetelt viselkedés kifejezésére.

23.1.1. példa. Csatorna MMT automata

Definiáljuk a $D_{i,j} = (C_{i,j}, b)$ MMT automatát, amely a 8.1.1. példában megadott $C_{i,j}$ univerzális megbízható FIFO küld/fogad csatorna automatából származik. $D_{i,j}$ b korlátfüggvénye a d rögzített valós szám felső korlátot rendeli a csatornában lévő legrégebbi üzenet kikerülési idejéhez. b nem követel meg semmilyen alsó korlátot. $D_{i,j}$ formális leírása az olyan csatornáknak, amelyeket az aszinkron algoritmusok fejezeteiben gyakran alkalmaztunk idő-hatékonysági elemzések kidolgozására.

Tehát, ha fog a $C_{i,j}$ egyetlen taszkját jelöli, akkor a b definíció szerint az (*alsó*, *felső*) pár, ahol *alsó*$(fog) = 0$ és *felső*$(fog) = d$, valamely rögzített $d \in \mathbb{R}^+$ értékre. A következő sorozatok mindegyike megengedett időzített története $D_{i,j}$-nek:

$$(\texttt{küld}(1)_{i,j}, 0), (\texttt{küld}(2)_{i,j}, 0), (\texttt{fogad}(1)_{i,j}, d), (\texttt{fogad}(2)_{i,j}, 2d)$$

$$(\texttt{küld}(1)_{i,j}, 0), (\texttt{küld}(2)_{i,j}, 0), (\texttt{fogad}(1)_{i,j}, 0), (\texttt{fogad}(2)_{i,j}, 0)$$

$$(\texttt{küld}(1)_{i,j}, 0), (\texttt{fogad}(1)_{i,j}, d), (\texttt{küld}(2)_{i,j}, d), (\texttt{fogad}(2)_{i,j}, 2d), (\texttt{küld}(3)_{i,j}, 2d), (\texttt{fogad}(3)_{i,j}, 3d), \ldots .$$

Másrészről, a következő sorozatok nem megengedett időzített történetei $D_{i,j}$-nek:

$$(\texttt{küld}(1)_{i,j}, 0), (\texttt{küld}(2)_{i,j}, 0), (\texttt{fogad}(1)_{i,j}, d)$$
$$(\texttt{küld}(1)_{i,j}, 0), (\texttt{fogad}(1)_{i,j}, 2d)$$
$$(\texttt{küld}(1)_{i,j}, 0), (\texttt{fogad}(1)_{i,j}, d), (\texttt{küld}(2)_{i,j}, d), (\texttt{fogad}(2)_{i,j}, d), (\texttt{küld}(3)_{i,j}, d), (\texttt{fogad}(3)_{i,j}, d), \ldots .$$

A három közül az első sorozat nem megengedett időzített történet, mivel véges és a fog taszk végrehajtható a végső állapotban. Általánosan, ha minden olyan megengedett időzített végrehajtási sorozat, ami k darab `küld` bemenetet tartalmaz, legalább k darab megfelelő `fogad` kimenetet kell tartalmazzon, mert a felső korlát és a megengedhetőség feltétel együttes teljesülése maga után vonja a szokásos pártatlanság feltétel teljesülését a fog taszkra. A második sorozat azért nem megengedett időzített történet, mert megsérti a felső korlát feltételt. A harmadik sorozat azért hibás, mert nem teljesíti a megengedhetőség feltételt – nem engedi meg, hogy az idő d-nél nagyobb értékre növekedjen, még akkor sem, ha végtelen sok művelet hajtódik végre.

23.1.2. példa. Időhiány MMT automata

Definiáljuk azt a P_2 MMT automatát, amely egy másik P_1 folyamattól érkező üzenetre várakozik, és ha az üzenet nem érkezik meg egy adott időn belül, akkor végrehajt egy `időhiány` műveletet. P_2 úgy méri az eltelt időt, hogy leszámlálja saját lépéseit rögzített $k \geq 1$ számszor, amire teljesül az ismert l_1 alsó és l_2 felső korlát, ahol $0 < l_1 \leq l_2 < \infty$. Az `időhiány` művelet legkésőbb az l időben végrehajtódik, miután a *számláló* 0-vá vált. Megjegyezzük, hogy a taszkokra az alsó és felső korlátokat zárt intervallum formájában adtuk meg, ezt a konvenciót gyakran alkalmazzuk majd.

Egy megengedett időzített végrehajtási sorozatban a P_2 automata egyszerűen csökkenti a *számláló* számlálója értékét, amíg vagy *számláló*$= 0$ nem lesz, vagy `fogad`(m) végrehajtódik és ezáltal érvényteleníti az időhiányt. Miután *számláló* elérte a 0 értéket, P_2 végrehajtja az `időhiány` műveletet (feltéve, hogy korábban nem érkezett `fogad` művelet). Nem nehéz belátni, hogy P_2 minden megengedett időzített végrehajtási sorozatában ha `időhiány` végrehajtódik, akkor az a $[kl_1, kl_2 + l]$ intervallumban történik. Továbbá, ha `időhiány` végrehajtódik, akkor nem lehetett korábban `fogad`. Végül, P_2 minden megengedett időzített végrehajtási sorozatában ha `fogad` sohasem hajtódik végre, akkor `időhiány` egyszer ténylegesen végrehajtódik.

23.1. automata. P_2

Lenyomat:

Bemeneti:
`fogad`$(m)_{1,2}, m \in M$

Belső:
`csökkent`

Kimeneti:
`időhiány`

Állapotok:

számláló $\in \mathbb{N}$, kezdőértéke k
státusz $\in \{$*aktív*, *bevégzett*, *kizárt*$\}$, kezdőértéke *aktív*

Átmenetek:

`csökkent`
Előfeltétel:
státusz = *aktív*
számláló > 0
Hatás:
számláló := *számláló*−1

$fogad(m)_{1,2}$
Hatás:
if *státusz* = *aktív* **then**
státusz := *bevégzett*

`időhiány`
Előfeltétel:
státusz = *aktív*
számláló = 0
Hatás:
státusz := *bevégzett*

Taszkok és korlátok:

{`csökkent`}, korlátok: $[l_1, l_2]$
{`időhiány`}, korlátok: $[0, l]$

23.1.3. példa. Kéttaszkos verseny

Definiáljuk azt az egyszerű MMT automatát, amelyet VERSENY-nek nevezünk és két taszkja van, a *fő* és az *meg_szakítás* (interrupt). A *fő* taszk mindaddig növeli a *számláló* értékét, amíg a *jelző* logikai állapotjelző értéke hamis. Az *meg_szakítás* taszk egyszerűen végrehajtja a *jelző* := *hamis* utasítást. Amikor a *jelző* = *igaz*, a *fő* taszk csökkenti a *számláló* értékét mindaddig, amíg az 0 nem lesz, és ekkor befejeződést jelez. A *fő* taszk korlátai l_1 és l_2 , ahol $0 < l_1 \leq l_2 < \infty$, míg az *int* taszknak csak az l felső korlátja van.

23.2. automata. VERSENY

Lenyomat:

Bemeneti:
nincs
Kimeneti:
`jelez`

Belső:
`növel`
`csökkent`
`beállít`

Állapotok:

számláló $\in \mathbb{N}$, kezdőértéke 0
jelző logikai, kezdőértéke *hamis*
jelzett logikai, kezdőértéke *hamis*

Átmenetek:

`növel`
Előfeltétel:
jelző = *hamis*
Hatás:
számláló := *számláló* +1

`csökkent`
Előfeltétel:
jelző = *igaz*
számláló > 0
Hatás:
számláló := *számláló*−1

`beállít`
Előfeltétel:
jelző= *hamis*
Hatás:
jelző := *igaz*

`jelez`
Előfeltétel:
jelző = *igaz*
számláló = 0
jelzett = *hamis*
Hatás:
jelzett := *igaz*

Taszkok és korlátok:
fő = {`növel`, `csökkent`, `jelez`}, korlátok: $[l_1, l_2]$
meg_szakítás = {`beállít`}, korlátok: $[0, l]$

23.1.2.. Műveletek

A b/k automatákhoz hasonlóan definiáljuk az összekapcsolás (compositon)) és az elrejtés (hiding) műveleteket az MMT automatákra.

Összekapcsolás.. Az MMT automaták összekapcsolása teljesen hasonló módon elvégezhető, mint a közönséges b/k automatáké, oly módon, hogy azonosítjuk azokat az műveleteket, amelyek neve megegyezik a komponens automatákban. Azonban, a b/k automatáktól eltérően, csak *véges* sok komponensből álló MMT automata összekapcsolását definiáljuk. Ez azért van, mert MMT automatának csak véges sok taszkja lehet.

MMT automaták egy véges halmazát *kompatibilisnek* nevezzük, ha a megfelelő b/k automaták halmaza kompatibilis a kompatibilitás 8.2.1. definíciója alapján. Ekkor az MMT automaták egy véges $\{(A_i, b_i)\}_{i \in I}$ kompatibilis halmazának *össszekapcsolása* az az $(A, b) = \prod_{i \in I} (A_i, b_i)$ MMT automata, amelyet a következőképpen definiálunk:

- $A = \prod_{i \in I} A_i$, tehát A a megfelelő A_i b/k automaták összekapcsolása.
- Az A minden C taszkjára a b szerinti *alsó* és *felső* korlát megegyezik annak az egyetlen A_i b/k automatának a b_i alsó, illetve felső korlátjával, amelyben a C taszk szerepel.

Mint a b/k automaták esetében, most is gyakran használjuk az infix $\times$ műveleti jelet az összekapcsolás jelölésére. Például, ha $I = \{1, \ldots, n\}$, akkor néha azt írjuk $\prod_{i \in I} A_i$ helyett, hogy $A_1 \times \cdots \times A_n$.

23.1.4. példa. MMT automaták összekapcsolása

Három MMT automata összekapcsolását vizsgáljuk. Az első az a P_1 folyamat, amely lehet élő vagy halott (ami nemdeterminisztikusan a kezdőállapotból határozható meg). Ha életben van, akkor periodikusan üzenetet küld a kimeneti csatornán, az üzenetek valamely rögzített M üzenet ábécéből vannak és az átfedési idő legfeljebb $l > 0$.

23.3. automata. P_1

Lenyomat:

Bemeneti:
nincs

Kimeneti:
küld$(m)_{1,2}, m \in M$

Állapotok:

státusz$\in \{$*élő*, *halott*$\}$, kezdőértéke tetszőleges

Átmenetek:

küld$(m)_{1,2}$
Előfeltétel:
státusz = *élő*
Hatás:
nincs

Taszkok és korlátok:

küld$(m)_{1,2}, m \in M$, korlátok: $[0, l]$

Az összekapcsolásban szereplő másik két automata a 23.1.1. példában megadott $D_{i,j}$ csatorna automata és a 23.1.2. példában adott P_2 időhiány automata. Ha $kl_1 > l + d$, akkor minden megengedett időzített végrehajtási sorozatban a összekapcsolás pontosan akkor hajt végre időhiány műveletet, ha P_1 halott. Továbbá, a időhiány művelet legkésőbb a $kl_2 + l$ időben végrehajtódik.

Ezt az alfejezetet a 8.1–8.3. tételekhez hasonló eredmények ismertetésével zárjuk. Ezek az állítások az összekapcsolással megalkotott és a komponens MMT automaták megengedett időzített végrehajtási sorozatainak történeteit hasonlítják össze. Az első azt fejezi ki, hogy az összekapcsolás megengedett időzített végrehajtási sorozata vagy a megengedett időzített történetének projekciója megengedett időzített végrehajtási sorozatot, illetve megengedett időzített történetet eredményez a komponens automatában.

Legyen $\{(A_i, b_i\}_{i \in I}$ MMT automatáknak egy kompatibilis halmaza és legyen $(A, b) = \prod_{i \in I}(A_i, b_i)$. Jelölje minden i-re B_i az (A_i, b_i) MMT automatát és B az (A, b) automatát. B minden $\alpha = s_0, (\pi_1, t_1), s_1, \ldots$ időzített végrehajtási sorozatára jelölje $\alpha|B_i$ azt az időzített végrehajtási sorozatát A_i-nek, amelyet úgy kapunk, hogy α-ból törlünk minden $(\pi_r, t_r), s_r$ párt, ha π_r nem művelete a B_i automatának és helyettesítjük s_r-t $(s_r)_i$-vel, azaz az s_r állapotnak A_i automatához

tartozó komponensével. Hasonlóan, a B automata minden β időzített története esetében (vagy általánosan, minden sorozat, amely (művelet, idő) párokból áll) jelölje $\beta|B_i$ a β azon részsorozatát, amely az A_i komponens automata műveleteit tartalmazza.

23.1. tétel. . *Legyen $\{B_i\}_{i\in I}$ MMT automatáknak egy kompatibilis halmaza és legyen $B = \prod_{i\in I} B_i$.*

- *Ha $\alpha \in m\_i\_végre(B)$, akkor $\alpha|B_i \in m\_i\_végre(B_i)$ minden $i \in I$ esetében.*
- *Ha $\beta \in m\_i\_végre(B)$, akkor $\beta|B_i \in m\_i\_végre(B_i)$ minden $i \in I$ esetében.*

Bizonyítás. A bizonyítás gyakorlatként elvégezhető (lásd 23-4. gyakorlat). □

A következő két tétel a 23.1. tétel megfordítása. Ezek közül az első azt mondja, hogy bizonyos feltételek teljesülése esetében a komponens MMT automaták megengedett időzített végrehajtási sorozataiból összerakható a szorzat automata egy megengedett időzített végrehajtási sorozata.

23.2. tétel. . *Legyen $\{B_i\}_{i\in I}$ MMT automatáknak egy kompatibilis halmaza és legyen $B = \prod_{i\in I} B_i$. Tegyük fel, hogy α_i egy megengedett időzített végrehajtási sorozata B_i-nek minden $i \in I$-re, és β (művelet, idő) párok olyan sorozata, hogy minden β-beli művelet eleme külső(A)-nak úgy, hogy $\beta|B_i = id\_történet(\alpha_i)$ minden $i \in I$-re. Ekkor létezik olyan α megengedett időzített végrehajtási sorozata B-nek, hogy $\beta = id\_történet(\alpha)$ és $\alpha_i = \alpha|B_i$ minden $i \in I$-re.*

Bizonyítás. A bizonyítás gyakorlatként elvégezhető (lásd 23-4. gyakorlat). □

23.3. tétel. . *Legyen $\{(B_i\}_{i\in I}$ MMT automatáknak egy kompatibilis halmaza és legyen $B = \prod_{i\in I} B_i$. Tegyük fel, hogy β (művelet, idő) párok alkotta sorozat, és minden művelet eleme külső(A). Ha $\beta|B_i \in m\_i\_történet(B_i)$ minden $i \in I$-re akkor $\beta \in m\_i\_történet(B)$.*

Bizonyítás. A bizonyítás gyakorlatként elvégezhető (lásd 23-4. gyakorlat). □

Elrejtés.. Az elrejtés művelet MMT automatákra definiálható a közönséges b/k automaták elrejtés műveletével, amit a 8.2.2. szakaszban definiáltunk. Nevezetesen, ha $B = (A, b)$ egy MMT automata és $\Phi \subseteq ki(A)$, akkor $rejt_\Phi(B)$ a $(rejt_\Phi(A), b)$ MMT automata lesz. Mint a b/k automatáknál, az elrejtés művelet egyszerűen besorolja a kimenet műveleteket a belső műveletek közé.

23.2.. Általános időzített automata

Az időzítési megszorításokat MMT automata esetében a végrehajtási sorozatra kirótt alsó és felső korlátok felállításával adjuk meg. Alternatív megoldásként az időzítési megszorításokat *közvetlenül* bevihetjük az *automata állapotaiba és az átmeneteibe*. Ez a megoldás azzal az előnnyel jár, hogy lehetővé teszi néhány

állapot-alapú bizonyítási módszer alkalmazását, mint az invariáns állítások módszere és a szimulációs relációk módszere, amelyeket időzítés-alapú rendszerek helyességének és időzítésének bizonyítására használhatunk.

Ebben az alfejezetben egy másik időzített automata modellt adunk, amit *általános időzített automata* (GTA) modellnek nevezünk. Az általános időzített automata esetében nincs „külső" időzítési megszorítás – minden időzítési megkötés az állapotokba és az átmenetekbe van kódolva. Amint majd megmutatjuk, az MMT automata tekinthető speciális általános időzített automatának, ahol az időzítési megszorítás kódolva van. Létezik olyan GTA, amely nem MMT automata, sőt, ténylegesen létezik olyan GTA, amelynek a viselkedése nem adható meg MMT automata viselkedéseként.

23.2.1.. Alapfogalmak

Műveletek univerzális halmazát tételezzük fel, amely tartalmaz speciális $\nu(t)$, $t \in \mathbb{R}^+$ *időátmenet* műveleteket. A $\nu(t)$ időátmenet művelet t időtartam elteltét jelöli. Az S *időzített lenyomat* olyan rendezett négyes, amely a műveletek négy diszjunkt halmazából áll: a $be(S)$ *bemeneti műveletek*, a $ki(S)$ *kimeneti műveletek*, a $bel(S)$ *belső műveletek* valamint az időátmeneti műveletek halmazából. A műveletek halmazának az alábbi részhalmazait definiáljuk:

- a *látható műveletek* $lát(S)$ halmaza a bemeneti és kimeneti műveletek halmazának $be(S) \cup ki(S)$ egyesítése;
- A *külső műveletek* $kül(S)$ halmaza a látható és az időátmeneti műveletek halmazának $lát(S) \cup \{\nu(t) : t \in \mathbb{R}^+\}$ egyesítése.
- A *diszkrét műveletek* $diszk(S)$ halmaza a látható és belső műveletek halmazának $lát(S) \cup bel(S)$ egyesítése;
- A *lokálisan vezérelt műveletek* $helyi(S)$ halmaza a kimeneti és a belső műveletek halmazának $ki(S) \cup bel(S)$ egyesítése;
- $műv(S)$ az S összes műveletének a halmaza.

Egy A GTA a következő négy komponensből áll:

- $lenyomat(A)$ az időzített lenyomat;
- $állapot(A)$ az állapotok halmaza;
- $start(A)$ az $állapot(A)$ nemüres részhalmaza, a kezdő állapotok halmaza;
- $átmenet(A)$ az állapot-átmenet reláció, $átmenet(A) \subseteq állapot(A) \times műv(lenyomat(A)) \times állapot(A)$.

Ellentétben a b/k automatával, a GTA-nak nincs $taszk(A)$ komponense. Mint korábban is, a $műv(A)$ rövidítést használjuk $műv(lenyomat(A))$ helyett és hasonlóan használjuk a $bel(A)$, és a többi jelöléseket. Az A automatának két egyszerű axiómát kell kielégítenie.

A1. Ha $(s, \nu(t), s')$ és $(s', \nu(t'), s'')$ eleme az $átmenet(A)$ halmaznak, akkor $(s, \nu(t + t'), s'') \in átmenet(A)$.

A2. Ha $(s, \nu(t), s') \in \textit{átmenet}(A)$ és $0 < t' < t$ akkor van olyan s'' állapot, hogy $(s, \nu(t'), s'') \in \textit{átmenet}(A)$ és $(s'', \nu(t-t'), s') \in \textit{átmenet}(A)$.

Az A1 axióma lehetővé teszi, hogy ismételt időátmeneti lépéseket egyetlen lépéssé vonjunk össze, míg az A2 axióma egyfajta fordítottja az A1-nek, mivel megengedi, hogy egy időátmenet lépést kettébontsunk.

Az A GTA *időzített végrehajtási szeletén* egy $\alpha = s_0, \pi_1, s_1, \pi_2, \ldots, \pi_r, s_r$ véges, vagy $\alpha = s_0, \pi_1, s_1, \pi_2, \ldots, \pi_r, s_r \ldots$ végtelen sorozatot értünk, ahol minden s állapota, minden π művelete A-nak (bemeneti, kimeneti, külső vagy időátmeneti) és minden k-ra $(s_k, \pi_{k+1}, s_{k+1})$ átmenete A-nak. Megjegyzendő, hogy ha a sorozat véges, akkor állapottal kell végződnie. Az olyan időzített végrehajtási sorozati szeletet, amely kezdő állapottal kezdődik, *időzített végrehajtási sorozatnak* nevezünk.

Ha α időzített végrehajtási szelet és π_r valamely diszkrét művelet α-ban, akkor azt a valós számot, amely az összes π_r-et megelőző időátmeneti műveletek összege, a π_r művelet előfordulási idejének nevezzük. Egy α időzített végrehajtási szeletet *megengedettnek* nevezünk, ha az α-beli időátmenet műveletekben szereplő valós számok összege ∞. Jelölje $m\_i\_\textit{végre}(A)$ az A összes megengedett időzített végrehajtási sorozatának halmazát. Mi elsősorban megengedett végrehajtási sorozatokkal foglalkozunk, de néha vizsgálunk véges időzített végrehajtási sorozatokat is, amelyek véges sorozatok. Az A automata egy állapotát elérhetőnek nevezzük, ha utolsó állapota A valamely időzített végrehajtási sorozatának.

Egy α időzített végrehajtási szelete *időzített története* az a sorozat, amely α látható műveleteiből áll, párosítva a hozzájuk tartozó előfordulási időkkel. Az A automata *megengedett időzített történetei* az A megengedett időzített végrehajtásaihoz tartozó időzített történetek halmaza, amit $m\_i\_\textit{történet}(A)$-val jelölünk. Megjegyzendő, hogy egy A automata megengedett időzített történeteinek halmaza lehet véges, jóllehet (végtelen) megengedett időzített végrehajtási sorozatokból származik.

Az A1 és A2 axiómák miatt nincs sok különbség két olyan időzített végrehajtási sorozati szelet között, amelyek csak bizonyos időátmenet lépések szétbontása és összevonása miatt különböznek. Definiáljuk azt az ekvivalencia relációt az időzített végrehajtási szeleteken, amely azt fejezi ki, hogy azok azonosak, kivéve az időátmeneteket. Nevezetesen, azt mondjuk, hogy egy α időzített végrehajtási szelet *időátmenet finomítása* egy α' időzített végrehajtási sorozati szeletnek, ha α és α' azonosak lesznek, ha α-ban bizonyos időátmenet lépéseket helyettesítünk olyan véges időátmenet sorozattal, ahol a kezdő és végső állapot megegyezik és a helyettesített lépés ideje megegyezik a sorozatban szereplő idők összegével. Azt mondjuk, hogy az α és α' időzített végrehajtási sorozati szelet *időátmenet ekvivalensek*, ha létezik közös időátmenet finomításuk.

23.2.1. példa. Egy általánosított időzített automata

Definiáljuk azt a $D'_{i,j}$ általánosított időzített automatát, amely szorosan megfelel a 23.1.1 példában adott $D_{i,j}$ MMT automatának. A két automata megengedett időzített történeteinek halmaza megegyezik.

$D'_{i,j}$ egyszerűen kódolja a $D_{i,j}$ időmegszorítását – ami a legrégebbi üzenet csatornából való kikerülési idejének d felső korlátja – az állapotaiba és az átmeneteibe. Ezt úgy valósítja meg, hogy nyomon követi az aktuális időt a *most* változóban tárolva és nyomon követi a *utolsó* változóban azt a legkésőbbi időt, amikor a következő üzenet kiküldése bekövetkezhet. Jegyezzük meg, hogy a *utolsó* változó értéke abszolút idő és nem növekmény.

A $D'_{i,j}$ automatát most is olyan előfeltétel/hatás jelölésekkel definiáljuk, mint más automaták esetében, csak most az időátmenet műveletek kódját is megadjuk, csakúgy mint a diszkrét műveletekét.

Amikor egy `küld` művelet végrehajtódik, a *sor* úgy módosul, mint korábban, de most ráadásul az *utolsó* változó értéke felveszi a $most + d$ értéket, hogy megvalósuljon az a követelmény, hogy a következő üzenet kikerülése d időn belül bekövetkezzen. Amikor `fogad` hajtódik végre, akkor az *utolsó* értéke $most + d$ értékre változik ha a sor nem üres a művelet után, hogy megvalósuljon a következő üzenet kiküldésére kirótt időkorlát, másrészről ha a sor üressé vált, akkor az *utolsó* felveszi a ∞ értéket, hogy jelezze, nincs ütemezett továbbítandó üzenet.

A $\nu(t)$ időátmenet műveletek kódjának leírása nagyon hasonló a többi műveletéhez. A $\nu(t)$ művelet hatása egyszerűen az, hogy megnöveli a *most* aktuális idő értékét t-vel. Vegyük észre, hogy $\nu(t)$-re van egy nem triviális előfeltétel: $most + t \leq utolsó$. Ez azt fejezi ki, hogy az idő nem haladhat túl a következő üzenet beütemezett kikerülési határidején. Ez elsőre furcsának tűnhet – végül is hogyan állíthatná meg egy program vagy gép az idő múlását? Azonban az időátmenetek leírásának ez a stílusa csak formális módja annak, hogy megköveteljük, hogy az automata valamilyen műveletet végrehajtson, mielőtt letelne egy adott időtartam.

23.4. automata. $D'_{i,j}$

Időzített lenyomat:

Bemeneti:
- `küld`$(m)_{i,j}, m \in M$ Kimeneti:
- `fogad`$(m)_{i,j}, m \in M$

Belső:
- nincs

Időátmeneti:
- $\nu(t), t \in \mathbb{R}^+$

Állapotok:

sor, M-beli elemek FIFO sora, kezdőértéke üres
$most \in \mathbb{R}_0^+$, kezdőértéke 0
$utolsó \in \mathbb{R}^+ \cup \{\infty\}$, kezdőértéke ∞

Átmenetek:

küld$(m)_{i,j}$
  Hatás:
    m bekerül a *sor* sorba
    if $|sor| = 1$ **then**
      utolsó := *most*+d

fogad$(m)_{i,j}$
  Előfeltétel:
    m első a sorban
  Hatás:
    távolítsuk el *sor*-ból az első elemet
    if *sor* nem üres **then**
      utolsó := *most* +d
    else *utolsó* := ∞

$\nu(t)$
  Előfeltétel:
    most +$t \leq$ *utolsó*
  Hatás:
    most := *most*+t

Nem nehéz belátni, hogy $D'_{i,j}$ és $D_{i,j}$ megengedett időzített történeteinek halmaza megegyeznek. Ennek igazolása gyakorlatként elvégezhető (lásd 23-6. gyakorlat).

A 23.2.1. példa ötletet adhat az olvasónak arra, hogy az MMT automata hogyan tekinthető az általános időzített automata speciális esetébenek: az (A, b) MMT automata b korlátfüggvénye által kifejezett idő-megszorítások belekódolhatók a megfelelő GTA állapotaiba és átmeneteibe. Ezt úgy kaphatjuk, hogy az *utolsó* állapot komponens nyilvántartja a felső korlát követelményt, az *első* komponens pedig az alsó korlát követelmény teljesítését biztosítja. A részletes konstrukciót a 23.2.2. szakaszban adjuk meg.

A GTA modell azonban általánosabb, mint az MMT modell. A következő példa egy másik csatorna GTA-t tartalmaz, és mint majd kiderül, ez nem fejezhető ki MMT automatával.

23.2.2. példa. Nem MMT általános időzített automata

Egy másik GTA-t adunk, ez a $D''_{i,j}$, amely megbízható FIFO csatornát reprezentál, de itt nem csak az utolsóra, hanem minden üzenetre megköveteljük a d időkorlátot. Ezúttal az üzenet kikerülési határidőket is tároljuk a sorban magukkal az üzenetekkel, és nem egy különálló *utolsó* komponensben. A határidők kezelése azonban hasonló.

23.5. automata. $D''_{i,j}$

Időzített lenyomat:

Bemeneti:
$\texttt{küld}(m)_{i,j}, m \in M$
Kimeneti:
$\texttt{fogad}(m)_{i,j}, m \in M$
Belső:
nincs
Időátmeneti:
$\nu(t), t \in \mathbb{R}^+$

Állapotok:

sor, $M \times \mathbb{R}^+$-beli elemek FIFO sora, kezdőértéke üres
$most \in \mathbb{R}_0^+$, kezdőértéke 0

Átmenetek:

$\texttt{küld}(m)_{i,j}$
Hatás:
adjuk $(m, most + d)$-t *sor*-hoz

$\texttt{fogad}(m)_{i,j}$
Előfeltétel:
(m, t) első a *sor*-ban valamely t-re
Hatás:
távolítsuk el a *sor* első elemét

$\nu(t)$
Előfeltétel:
if *sor* üres **then**
$most + t \leq t'$, ahol t'
sor első párjának az ideje
Hatás:
$most := most + t$

Azt állítjuk (a bizonyítás gyakorlatként elvégezhető, lásd 23-7. gyakorlat), hogy nincs olyan MMT automata, amelynek megengedett időzített történeteinek halmaza megegyezne $D''_{i,j}$-ével. Ezt úgy lehetne értelmezni, hogy $D''_{i,j}$ fizikailag nem valósítható meg. Azonban, mint azt már láttuk a megelőző fejezetekben, $D''_{i,j}$ egy kényelmes absztrakciót biztosít olyan algoritmusok időbonyolultságának elemzésére, ahol nem törődünk az üzenetek csatornából történő kikerülésével.

A következő példa egy anomáliát mutat: olyan GTA-t, amelynek nincs megengedett időzített végrehajtási sorozata. Ámbár ez furcsa, de nincs mód az általános modellben ennek kizárására. Az MMT automaták esetében (és következésképpen az olyan GTA-k esetében, amelyek a 23.2.2. szakaszban leírtak szerint MMT automatából származnak), ez az anomália nem fordul elő (lásd a 23-1. gyakorlatot). További megszorítást kell hozzávenni a GTA-hoz, hogy kizárjuk ezt az esetet, de mivel mi főleg olyan algoritmusokra koncentrálunk, amelyek kifejezhetők MMT automatával, ezért ilyen megszorításokat nem vizsgálunk.

23.2.3. példa. Általánosított időzített automata, amelynek nincs megengedett időzített végrehajtási sorozata

Tekintsük azt az A „folyamat automatát”, amely végtelen sokszor kiküldi ugyanazt az m üzenetet. Az egymást követő kiküldési időpontok azonban egyre közelebb és közelebb lesznek, megközelítve az 1 határértéket.

23.6. automata. A

Időzített lenyomat:

Bemeneti:
nincs
Kimeneti:
küld(m)

Belső:
nincs
Időátmeneti:
$\nu(t), t \in \mathbb{R}^+$

Állapotok:

$most \in \mathbb{R}_0^+$, kezdőértéke 0
$utolsó \in \mathbb{R}_0^+ \cup \{\infty\}$, kezdőértéke 0

Átmenetek:

küld(m)
Előfeltétel:
$most = utolsó$
Hatás:
$most := most + \frac{1-most}{2}$

$\nu(t)$
Előfeltétel:
$most + t \leq utolsó$
Hatás:
$most := most + 1$

Valójában a helyzet még rosszabb – a GTA definíciója még azt is megengedi, hogy az automatának egyáltalán ne legyen időátmeneti művelete.

A GTA modell az időzítés-alapú rendszereknek nem a lehető legáltalánosabb modellje. Például nem tartalmaz olyan tulajdonságot, amely az élénkség (a megengedhetőséget kivéve) feltételét fejezné ki. Az élénkség feltétele kevésbé fontos az időzített, mint a nem időzített esetben, mert sok élénkségi feltétel (vagyis ami azt fejezi ki, hogy valaminek végső soron történnie kell) helyettesíthető megfelelő felső korlát követelménnyel (tehát olyan feltétellel, amely szerint az esemény t időn belül bekövetkezik). Gyakran hasznos azonban, ha mind az időkorlát, mind az élénkség feltételét ki tudjuk fejezni ugyanarra a rendszerre vonatkozóan.

A GTA modell szintén nem eléggé általános ahhoz, hogy részletes leírását adja hibrid rendszereknek – amelyek analóg fizikai és diszkrét számítógépes komponensekből épülnek fel. A modell azonban megfelelő a könyv céljait tekintve.

23.2.2.. MMT automata átalakítása általános időzített automatával

Azt mondtuk, hogy az általános időzített automata modell általánosítása az MMT automata modellnek. Ez azonban formálisan nem igaz, mivel a két modell eltérően fejezi ki az időzítési megszorításokat: az MMT automata modell

korlátfüggvényeket használ, míg a GTA modell a megszorításokat belekódolja az állapotokba és az átmenetekbe. Ahhoz, hogy az MMT modellt a GTA modell speciális esetébenek tekinthessük, el kell végeznünk némi munkát. Ebben a szakaszban megmutatjuk, hogy bármely (A, b) MMT automatát hogyan lehet átalakítani olyan $A' = gen(A, b)$ általános időzített automatává, amely természetes módon megfelel (A, b)-nek.

A konstrukció hasonló ahhoz, amellyel a $D'_{i,j}$ automatát megalkottuk a $D_{i,j}$ automatából a 23.2.1. példában. Ez a módszer határidőknek az állapotokba történő beépítését alkalmazza és nem engedi meg, hogy az idő túlhaladjon azokon a határidőkön, amíg azok érvényesek. A nem időátmeneti műveletekre is bevezetünk feltételeket, hogy kifejezzük az alsó korlát követelményeket.

Speciálisan, a megfelelő b/k automata állapothalmazát kibővítjük egy *most* változóval, továbbá minden taszkhoz bevezetjük az *első*(C) és *utolsó*(C) új állapotokat. Az *első(C)* azt a legkorábbi, *utolsó(C)* pedig azt a legkésőbbi időpontot tartalmazza, amelyben a C taszk következő műveletének a végrehajtási sorozata megengedett. A *most*, *első* és *utolsó* komponensek mindegyike abszolút és növekményes időt tartalmaznak. A műveletek halmazát is kibővítjük a $\nu(t)$ időátmenet műveletekkel.

Az *első*(C) és *utolsó*(C) komponensek értéke természetes módon aktualizálódik a különböző lépések során a b korlátfüggvény *alsó* és *felső* korlátoknak megfelelően. A $\nu(t)$ időátmenet műveletek kifejezett előfeltétele, hogy az idő egyetlen *utolsó*(C) értéket sem haladhat meg, mert ezek a taszkok határidejét jelentik. Minden C taszk minden műveletéhez bevezetjük azt a korlátozást, amely azt fejezi ki, hogy a *most* aktuális idő legalább akkora legyen, mint az *első*(C) alsó korlát értéke.

Részletesebben, az $A' = gen(A, b)$ automata időzített lenyomata megegyezik az A lenyomatával, kiegészítve a $\nu(t), t \in \mathbb{R}^{\geq 0}$ időátmenet műveletekkel. A' minden állapota az alábbi komponensekből áll:

alap $\in$ *állapot*(A), kezdetben A egy kezdő állapota;
most $\in \mathbb{R}^{\geq 0}$, kezdetben 0;
A minden C taszkjára:
  első$(C) \in \mathbb{R}_0^+$, kezdetben *alsó*(C), ha C megengedett az *alap* állapotban;
    különben 0;
  utolsó$(C) \in \mathbb{R}^+ \cup \{\infty\}$, kezdetben *felső*$(C)$, ha C megengedett az *alap* állapotban,
    különben ∞.

Az átmenetek definíciója a következő.

Ha $\pi \in$ *műv*(A), akkor $(s, \pi, s') \in$ *átm*(A') akkor és csak akkor, ha a következő feltétel mindegyike teljesül.

1. $(s.alap, \pi, s'.alap) \in$ *átm*(A).
2. $s'.most = s.most$.
3. Minden $C \in$ *taszk*(A)-ra:

   (a) ha $\pi \in C$ akkor $s.$*első*$(C) \leq s.most$.

(b) ha C megengedett az $s.alap$ és az $s'.alap$ állapotokban és $\pi \notin C$, akkor $s.első(C) = s'.első(C)$ és $s.utolsó(C) = s'.utolsó(C)$.
(c) ha C megengedett az $s'.alap$ állapotban és vagy C nem megengedett $s.most$-ban vagy $\pi \in C$, akkor $s'.első(C) = s.most{+}alsó(C)$ és $s'.utolsó(C) = s.most{+}felső(C)$.
(d) ha C nem megengedett az $s'.alap$ állapotban, akkor $s'.első(C) = 0$ és $s'.utolsó(C) = \infty$.

Ha $\pi = \nu(t)$, akkor $(s, \pi, s') \in átm(A')$ pontosan akkor, ha az alábbi feltételek mindegyike teljesül:
1. $s'.most = s.most$;
2. $s'.most = s.most + t$;
3. minden $C \in taszk(A)$-ra
   (a) $s'.most \leq s.utolsó(C)$;
   (b) $s'.első(C) = s.első(C)$ és $s'.utolsó(C) = s.utolsó(C)$.

23.4. tétel. . *Ha (A, b) MMT automata, akkor $gen(A, b)$ általános időzített automata. Továbbá, m_i_történet(A, b) =m_i_történet$(gen(A, b))$*

23.5. lemma. . *A $gen(A, b)$ automata minden elérhető állapotára és A minden C taszkjára teljesül az alábbi négy egyenlőtlenség:*
1. *most$\leq$ utolsó(C);*
2. *ha C végrehajtható, akkor utolsó$(C) \leq$most+felső(C);*
3. *első$(C) \leq$most+alsó(C);*
4. *első$(C) \leq$utolsó(C).*

Triviális komponensek elhagyása.. Ha a b által megkövetelt valamely időzítési feltétel triviális módon teljesül – vagyis ha valamely alsó korlát 0, vagy felső korlát ∞ –, akkor egyszerűsíthető a $gen(A, b)$ automata a megfelelő komponensek elhagyásával. Ezt tesszük a példáinkban.

23.2.4. példa. Átalakított MMT automata

Legyen (A, b) a 23.1.4. példában leírt MMT automata, amely a P_1, P_2 és a $D_{1,2}$ csatorna MMT automaták összekapcsolása. Közvetlenül megadjuk az $A' = gen(A, b)$ GTA automata kódját. Amint azt előbb említettük, elhagyjuk a triviális korlátokat. Tehát csak minden taszk felső korlátját és a P_2 taszk csökkentő műveletének az alsó korlátját kell beépítenünk. A következő neveket használjuk a taszkok elnevezésére: *küld* a P_1 egyetlen taszkja, *fog* a $D_{1,2}$ egyetlen taszkja, valamint *csökk* és *időhiány* a P_2 két taszkja.

23.7. automata. A'

Időzített lenyomat:

Bemeneti:
nincs
Kimeneti:
`küld`$(m)_{1,2}, m \in M$
`fogad`$(m)_{1,2}, m \in M$
`időhiány`

Belső:
`csökkent`
Időátmeneti:
$\nu(t), t \in \mathbb{R}^+$

Állapotok:

státusz $\in \{$*élő*, *halott*$\}$, kezdőértéke tetszőleges
sor, M-beli elemek FIFO sora, kezdőértéke üres
számláló$_2 \in \mathbb{N}$, kezdőértéke 0
státusz$_2 \in \{$*aktív*, *bevégzett*, *hibás* $\}$, kezdőértéke *aktív*
most$\in \mathbb{R}^{\geq 0}$, kezdetben 0
utolsó(*küld*)$\in \mathbb{R}_0^+$, kezdőértéke l ha *statusz*=*élő*, egyébként ∞
utolsó(*fog*)$\in \mathbb{R}^+ \cup \{\infty\}$, kezdőértéke ∞
első(*csökk*)$\in \mathbb{R}^{\geq 0}$, kezdőértéke l_1
utolsó(*csökk*)$\in \mathbb{R}^+ \cup \{\infty\}$, kezdőértéke l_2
utolsó(*időhiány*)$\in \mathbb{R}^+ \cup \{\infty\}$, kezdőértéke ∞

Átmenetek:

`küld`$(m)_{1,2}$
Előfeltétel:
státusz=*élő*
Hatás:
tegyük be m-et a *sor*-ba
utolsó(*küld*) := *most*+l
if $|sor| = 1$ **then**
utolsó(*fog*) := *most* +d

`csökkent`
Előfeltétel:
státusz$_2$=*aktív*
számláló$_2 > 0$
most $\geq$ *első*(*csökk*)
Hatás:
számláló$_2$:= *számláló*$_2 - 1$
if *számláló*$_2 > 0$ **then**
első(*csökk*) := *most* +l_1
utolsó(*csökk*):=*most*+l_2
else
első(*csökk*):=0
utolsó(*csökk*):=∞
utolsó(*időhiány*):=*most*+l

`fogad`$(m)_{1,2}$
Előfeltétel:
m első a sorban
Hatás:
az első elem kikerül a sorból
if *státusz*$_2$=*aktív* **then**
státusz$_2$:=*hibás*
if *sor* nem üres **then**
utolsó(*fog*):=*most*+d
else *utolsó*(*fog*):=∞
első(*csökk*):=0
utolsó(*csökk*):=∞
utolsó(*időhiány*):=∞
`időhiány`
Előfeltétel:
státusz$_2$=*aktív*
számláló$_2 = 0$
Hatás:
státusz$_2$:= *bevégzett*
utolsó(*időhiány*):=∞

$\nu(t)$
Előfeltétel:
most+$l \leq$ *utolsó*(*küld*)
most+$l \leq$ *utolsó*(*fog*)
most+$l \leq$ *utolsó*(*csökk*)
most+$l \leq$ *utolsó*(*időhiány*)
Hatás:
most := *most* +l

23.2.3.. Műveletek

Összekapcsolás.. Most definiáljuk az összekapcsolás műveletet általánosított időzített automatákra, ami általánosítása az MMT automaták esetére már definiált összekapcsolásnak. Azt mondjuk, hogy időzített lenyomatok $\{S_i\}_{i\in I}$ véges halmaza *kompatibilis*, ha minden $i, j \in I$ esetében, ha $i \neq j$, akkor

1. $bel(S_i) \cap műv(S_j) = \emptyset$;
2. $ki(S_i) \cap ki(S_j) = \emptyset$.

GTA-k egy halmaza *kompatibilis*, ha az időzített lenyomatuk halmaza kompatibilis.

Időzített lenyomatok $S_{i\in I}$ véges halmazának összekapcsolása az az $S = \prod_{i\in I} S_i$ időzített lenyomat, amelyet az alábbiak szerint definiálunk:

- $ki(S) = \cup_{i\in I} ki(S_i)$
- $bel(S) = \cup_{i\in I} bel(S_i)$
- $be(S) = \cup_{i\in I} be(S_i) - \cup_{i\in I} ki(S_i)$

GTA-k egy véges kompatibilis $\{A\}_{i\in I}$ halmazának $A = \prod_{i\in I} A_i$ összekapcsolása az az automata, amelyet a következőképpen definiálunk:[2]

- $lenyomat(A) = \prod_{i\in I} lenyomat(A_i)$;
- $állapot(A) = \prod_{i\in I} állapot(A_i)$,
- $start(A) = \prod_{i\in I} start(A_i)$;
- $átmenet(A)$ az olyan (s, π, s') hármasok halmaza, ahol minden $i \in I$ esetében, ha $\pi \in műv(A_i)$, akkor $(s_i, \pi, s'_i) \in átmenet(A_i)$, különben $s_i = s'_i$.

Az összekapcsolással kapott automata átmenetét úgy kapjuk, hogy megengedjük, hogy minden olyan komponens, amelynek a lenyomatában a π művelet szerepel, egyidejűleg működjön a π-t tartalmazó lépésben, míg azok a komponensek, amelyekben nem szerepel, nem csinálnak semmit. Megjegyzendő, hogy ebből következik, hogy minden komponens részt vesz minden időátmenet lépésben és ugyanazon időtartammal. Ismét használjuk a $\times$ szimbólumot az összekapcsolás infix jelölésére.

23.6. tétel. . *Általánosított időzített automaták kompatibilis halmazának összekapcsolása általánosított időzített automata.*

Összekapcsolás avagy általánosítás.. Kiderül, hogy MMT automaták kompatibilis halmazát tekintve mindegy, hogy előbb vesszük az automaták összekapcsolását, majd alkalmazzuk a *gen* transzformációt, vagy először alkalmazzuk a *gen* transzformációt a komponensekre, majd vessük a komponensek összekapcsolását. Az eredményül kapott két GTA (a gépek elérhető része) izomorfizmus erejéig azonos.

[2] A $\prod$ jelölés a $start(A)$ és $állapot(A)$ definíciójában a közönséges Descartes-féle szorzást, míg a $lenyomat(A)$ definíciójában a most megadott időzített lenyomatok összekapcsolását jelöli. Itt most s_i az s állapotvektor i-edik komponensét jelöli.

Ismét kapunk a 8.1–8.3. tételekhez hasonló projekciós és pásztázó eredményeket. Legyen $\{B_i\}_{i\in I}$ GTA-k egy kompatibilis halmaza és legyen $B = \prod_{i\in I} B_i$. B minden $\alpha = s_0, \pi_1, s_1, \ldots$ időzített végrehajtási sorozatára jelölje $\alpha|B_i$ azt a sorozatot, amelyet úgy kapunk, hogy α-ból elhagyjuk a π_r, s_r párokat, ha π_r nem művelete a B_i komponensnek, és helyettesítjük s_r-t $(s_r)_i$-vel, tehát az A_i automata megfelelő komponensével. Hasonlóan, B minden β időzített története (vagy még általánosabban, minden művelet és idő párok alkotta sorozat) esetében legyen $\beta|B_i$ β-nak a B_i műveleteit tartalmazó részsorozata.

23.7. tétel. . *Legyen $\{B_i\}_{i\in I}$ általánosított időzített automaták egy kompatibilis halmaza és legyen $B = \prod_{i\in I} B_i$.*

1. *Ha $\alpha \in m\_i\_végre(B)$, akkor $\alpha|B_i \in m\_i\_végre(B_i)$ minden $i \in I$-re.*
2. *Ha $\alpha \in m\_i\_történet(B)$, akkor $\alpha|B_i \in m\_i\_történet(B_i)$ minden $i \in I$-re.*

Bizonyítás. A bizonyítás gyakorlatként elvégezhető (lásd 23-10. gyakorlat). □

Az első, 23.3.8. pásztázó tétel tartalmaz egy kis technikai részletet, ami következménye annak a ténynek, hogy a GTA modell megengedi, hogy több időátmenet lépés szerepeljen egymás után a végrehajtási sorozatban. Nevezetesen, ha az α megengedett időzített végrehajtási sorozat az α_i megengedett időzített végrehajtási sorozatokból lett „összepásztázva", a projekciók nem fogják pontosan visszaadni az eredeti α_i sorozatokat, hanem olyan megengedett időzített végrehajtási sorozatokat adnak, amelyek időátmenet ekvivalensek az eredeti α_i-kel.

23.8. tétel. . *Legyen $\{B_i\}_{i\in I}$ általánosított időzített automaták egy kompatibilis halmaza és legyen $B = \prod_{i\in I} B_i$. Tegyük fel, hogy minden $i \in I$-re α_i megengedett időzített végrehajtási sorozata B_i-nek, és tegyük fel, hogy β olyan (művelet, idő) párokból álló sorozat, ahol a művelet eleme lát(B)-nek és $\beta|B_i = id\_történet(\alpha_i)$ minden $i \in I$-re. Ekkor van olyan α megengedett időzített végrehajtási sorozata B-nek, hogy $\beta = m\_i\_történet(\alpha)$, és α_i időátmenet ekvivalens $\alpha|B_i$-vel minden $i \in I$-re.*

Bizonyítás. A bizonyítás gyakorlatként elvégezhető (lásd 23-10. gyakorlat). □

23.9. tétel. . *Legyen $\{B_i\}_{i\in I}$ általánosított időzített automaták egy kompatibilis halmaza és legyen $B = \prod_{i\in I} B_i$. Tegyük fel, hogy β olyan (művelet, idő) párokból álló sorozat, ahol a művelet eleme lát(A)-nak. Ha $\alpha|B_i \in m\_i\_történet(B_i)$ minden $i \in I$-re, akkor $\beta \in m\_i\_történet(B)$.*

Bizonyítás. A bizonyítás gyakorlatként elvégezhető (lásd 23-10. gyakorlat). □

Elrejtés.. Ha A GTA és $\Phi \subseteq ki(A)$, akkor $rejt_\Phi(A)$ az a GTA, amely azonos A-val, kivéve, hogy a Φ-beli műveletek a belső műveletek közé tartoznak.

23.3.. Tulajdonságok és bizonyítási módszerek

Az időzítés-alapú algoritmusok és rendszerek helyessége csakúgy, mint azok hatékonysága gyakran erősen függ az időzítési feltételektől. Ellentétben az aszinkron

esettel, az időzítési feltétel kisebb változása az időzítés-alapú algoritmus viselkedésének drasztikus változását eredményezheti. Az időzítéstől való ilyen függésre vonatkozó okfejtés azonban különösen nehéz lehet, még az olyan nagyon egyszerű algoritmusok esetében is, mint amelyeket e fejezetben ismertetünk. Szisztematikus bizonyítási módszerek nagy segítséget eredményezhetnek.

Ebben az alfejezetben időzítés-alapú algoritmusok két fontos bizonyítási technikáját ismertetjük, ezek az *invariáns állítások* módszere és a szimulációs relációk módszere. Mivel ezeket a módszereket sikerrel használtuk a szinkron és az aszinkron esetben, természetes, hogy megpróbáljuk alkalmazni őket az időzítés-alapú esetre is. Bevezetünk *időzített történet tulajdonság* fogalmat is, amely hasonló a 8.5.2. alfejezetben definiált *történet tulajdonság* fogalmához.

23.3.1.. Invariáns állítások

invariáns állítás/() Az A általánosított időzített automatára vonatkozó *invariáns állítás* definíció szerint olyan tulajdonság, amely igaz az A minden elérhető állapotára.

Ez a definíció formálisan ugyanaz, mint amit az aszinkron esetben használtunk. Van azonban egy különbség: aszinkron rendszereknél az állapot olyan közönséges adatokból áll, mint helyi vagy közös változók értékei és csatornákban tárolt üzenetek sorozatai. Időzítés-alapú rendszereknél azonban az állapot tartalmaz *idő információt* is, mint az aktuális idő és későbbi események időzített határideje. Például, ha egy esemény a csatornában várakozik, akkor az állapot tartalmazhat információt későbbi idő tartományra, amely azt mondja meg, hogy az esemény mikor kerülhet ki. Ez azt jelenti, hogy időzítés esetében az invariáns állítás tartalmazhat idő információt is a közönséges adatok mellett.

Ámbár időzítés esetében az állapotban lévő információ típusa gazdagabb, az invariánsok bizonyítási módszere ugyanaz, mint korábban – indukció. Most az indukció alapja a kérdéses állapothoz vezető végrehajtási sorozatban megtett lépések száma.

Jegyezzük meg, hogy az invariáns állítások módszerét mi itt általánosított időzített automatákra alkalmazzuk. Ha MMT automatákra akarjuk használni, akkor azokat előbb konvertálni kell GTA-ra.

23.3.1. példa. Az időhiány rendszer egy invariánsa

Tekintsük a 23.2.4. példában adott A' időhiány rendszert azzal a feltétellel, hogy $kl_1 > l + d$. Jó lenne bizonyítani, hogy a rendszer csak akkor hajt végre `időhiány` műveletet, ha a tartalmazott P_1 folyamat ténylegesen halott. A következő invariáns állítás használatával ez bizonyítható.

> **23.3.1. állítás.** *A' minden elérhető állapotára teljesül, hogy ha státusz$_1$ = élő, akkor számláló$_2 > 0$.*

Sajnálatosan, mint az gyakran előfordul, a 23.3.1. állítás nem bizonyítható egyedül indukcióval – további segédállításra van szükség. Itt

most először a következő állítást bizonyítjuk (egyszerűen indukcióval).

23.3.2. állítás. *A' minden elérhető állapotára teljesül, hogy ha státusz$_2$ = bevégzett, akkor számláló$_2 = 0$.*

Ezután bizonyítjuk a 23.3.1. állítás erősített változatát nem oly triviális indukcióval. Megjegyzendő, hogy ez az állítás tartalmaz hivatkozást az állapot *első* és *utolsó* idő komponensére.

23.3.3. állítás. *A' minden elérhető állapotára igaz, hogy ha státusz$_1$ =élő, akkor teljesülnek a következő feltételek:*

1. *számláló$_2 > 0$.*
2. *vagy utolsó(`küld`) $+d <$első(`csökk`)$+($számláló$_2 - 1)l_1$, és a sor nem üres, vagy statusz$_2$ = hibás.*
3. *Ha a sor nem üres, akkor vagy utolsó(fog) $<$ első(csökk) $+$ (számláló$_2 - 1)l_1$, vagy státusz$_2$ = hibás.*

Az első feltétel csupán átfogalmazása a 23.3.1. állításnak. A második és harmadik feltétel mindegyike egyenlőtlenségben tartalmazza az *első*(`csökk`)$+$ (*számláló*$_2 - 1)l_1$ kifejezést. Ez a kifejezés azt a legkorábbi időt írja le, amelyben a *számláló*$_2$ elérheti a 0 értéket, feltéve, hogy aktuálisan pozitív. Vagyis, *első*(*csökk*) a legkorábbi idő a következő `csökkent` műveletre, és további *számláló*$_2 - 1$ csökkent kell ahhoz, hogy *számláló*$_2$ 0-vá váljon, és a legkevesebb l_1 idő van mindegyik csökkent számára. A második feltétel azt mondja, hogy vagy úgy van ütemezve az üzenet, hogy elegendő idő van arra, hogy megérkezzen, mielőtt *számláló*$_2$ nullává válna, vagy az üzenet már továbbítás alatt áll, vagy egy már meg is érkezett (ezzel megszüntetve az időhiányt). A harmadik feltétel azt fejezi ki, hogy ha egy üzenet továbbítás alatt áll, akkor vagy megérkezik egy üzenet, mielőtt *számláló*$_2$ nullává válna, vagy egy már meg is érkezett. Tehát események időzítésére vonatkozó kívánalom tömör megfogalmazást nyer invariáns formájában, felhasználva az állapot *első* és *utolsó* határidő komponensét.

A 23.3.3. állítás bizonyítható az időzített végrehajtási sorozatban megtett lépések száma szerinti indukcióval. Az érvelés nyilvánvaló (valójában unalmas), de mégis leírjuk itt, mert további hasonló bizonyítások modellje lehet.

Alaplépés. Kezdetben *számláló*$_2 = k > 0$, a sor üres és *első*(*csökk*)$=$ l_1. Ebből következik az első és a harmadik feltétel. Továbbá, ha *státusz*$_1$ =*élő*, akkor *utolsó*(`küld`) $= l$, tehát

$$\textit{utolsó}(\texttt{küld}) + d \quad = \quad l + d < kl_1 = \textit{első}(\textit{csökk}) + (\textit{számláló}_2 - 1)l_1.$$

Ez pedig a második feltételt mutatja.

Indukciós lépés. Szokás szerint a műveletek fajtája szerinti esetszétválasztásos elemzést végzünk, de ez alkalommal a $\nu(t)$ időátmeneti műveleteket is tekinteni kell. Tegyük fel, hogy $(s, \pi, s') \in \mathit{átmenet}(A')$, és az s állapot kielégíti az invariánst. Tegyük fel, hogy $s'.\mathit{státusz}_1 = \mathit{élő}$, akkor $s.\mathit{státusz}_1 = \mathit{élő}$.

1. $\pi = \texttt{küld}(m)_{1,2}$.
   Ekkor $s.\mathit{első}(\mathit{csökk}) = s'.\mathit{első}(\mathit{csökk})$, $s.\mathit{számláló}_2 = s'.\mathit{számláló}_2$ és $s.\mathit{státusz}_2 = s'.\mathit{státusz}_2$. Ez a lépés nem módosítja az első feltételt és a második feltétel teljesülését okozza. Nézzük a harmadik feltételt. Ha az $s.\mathit{sor}$ nem üres, akkor $s.\mathit{utolsó}(\mathit{fog}) = s'.\mathit{utolsó}(\mathit{fog})$, tehát abból, hogy a harmadik feltétel teljesül az s állapotra következik, hogy teljesül az s' állapotra is. Tehát tegyük fel, hogy az $s.\mathit{sor}$ üres. Az indukciós feltétel (második feltétel) miatt vagy $s.\mathit{utolsó}(\texttt{küld}) + d < s.\mathit{első}(\mathit{csökk}) + (s.\mathit{számláló}_2 - 1)l_1$ vagy $s.\mathit{státusz}_2 = \mathit{hibás}$. Az utóbbi esetben készen vagyunk, tehát tegyük fel, hogy az első eset igaz. Ekkor $s'.\mathit{utolsó}(\mathit{fog}) = s.\mathit{most} + d \leq s.\mathit{utolsó}(\texttt{küld}) + d$ a 23.5. lemma szerint $< s.\mathit{első}(\mathit{csökk}) + (s.\mathit{számláló}_2 - 1)l_1$, ami elegendő.
2. $\pi = \texttt{fogad}(m)_{1,2}$.
   Az indukciós feltétel szerint $s.\mathit{számláló}_2 > 0$. Ezt az összefüggést nem érint ez a lépés, így az első feltétel teljesül. Hasonlóan, a 23.3.2. állításból következik, hogy $s.\mathit{státusz}_2 \neq \mathit{bevégzett}$. Tehát $s'.\mathit{státusz}_2 = \mathit{hibás}$, amiből következik a második és harmadik feltétel.
3. $\pi = \texttt{időhiány}$.
   Az indukciós feltétel szerint $s.\mathit{számláló}_2 > 0$. Tehát az `időhiány` művelet nem hajtható végre s-ben, ami azt jelenti, hogy ez az eset nem következhet be.
4. $\pi = \texttt{csökkent}$.
   Az első feltételt indirekt módon bizonyítjuk. Ha $s'.\mathit{számláló}_2 = 0$, akkor $s.\mathit{számláló}_2 = 1$, így az indukciós feltétel szerint (második és harmadik feltétel) vagy $s.\mathit{utolsó}(\texttt{küld}) + d < s.\mathit{első}(\mathit{csökk})$, $s.\mathit{utolsó}(\mathit{fog}) < s.\mathit{első}(\mathit{csökk})$, vagy $s.\mathit{státusz}_2 = \mathit{hibás}$. A `csökkent` művelet előfeltétele miatt az utóbbi nem lehetséges. Tehát azt kapjuk, hogy

   $$\mathit{min}(s.\mathit{utolsó}(\texttt{küld}), s.\mathit{utolsó}(\mathit{fog})) < s.\mathit{első}(\mathit{csökk}) \leq s.\mathit{most}.$$

   De $s.\mathit{most} \leq s.\mathit{utolsó}(\texttt{küld})$ és $s.\mathit{most} \leq s.\mathit{utolsó}(\mathit{fog})$ a 23.5. lemma miatt, így $s.\mathit{most} \leq \mathit{min}(s.\mathit{utolsó}(\texttt{küld}), s.\mathit{utolsó}(\mathit{fog}))$. Ez pedig ellentmondás. A második és harmadik feltételhez elegendő megmutatni, hogy az $s.\mathit{első}(\mathit{csökk}) + (s.\mathit{számláló}_2 - 1)l_1$ kifejezés értéke nem növekszik a lépés során. Ez abból következik, hogy a második tag értéke pontosan l_1-el csökken, míg az első tag értéke legalább l_1-el nő. (Ez azért van, mert $s.\mathit{első}(\mathit{csökk}) \leq s.\mathit{most}$ és $s'.\mathit{első}(\mathit{csökk}) = s.\mathit{most} + l_1$.)

5. $\pi = \nu(t)$.
   Ez a lépés nem érinti egyik feltételt sem, mert csak a *most* értékét változtatja, de az sehol sem fordul elő a feltételekben.

23.3.2.. Időzített történet tulajdonságok

Emlékeztetünk, hogy aszinkron rendszerek sok bizonyítandó tulajdonsága természetes módon megfogalmazható volt történet vagy pártatlan történet tulajdonságaként. Kiderül, hogy hasonló módon, az időzített rendszerek több érdekes tulajdonsága megfogalmazható azok megengedett időzített történeteinek tulajdonságaként. Az ilyen módon kifejezhető tulajdonságok közé tartoznak mind a hatékonysági, mind a közönséges helyességi tulajdonságok.

Egy P *időzített történet tulajdonság*-ot a következő két komponenssel definiáljuk:

- *lenyomat*(P), olyan időzített lenyomat, amely nem tartalmaz belső műveleteket;
- *id_ történet*(P), ami (*művelet*, *idő*) párokból álló sorozatok halmaza; az idő komponensek sorozata monoton nemcsökkenő, és ha a sorozat végtelen, akkor nem korlátos.

Az olyan állítást, amely azt mondja, hogy egy A GTA kielégít egy P történet tulajdonságot, úgy értelmezzük, hogy $be(A) = be(P)$, $ki(A) = ki(P)$ és $m\_\,i\_$ *történet*$(A) \subseteq$ *id_ történet*(P).

23.3.2. példa. Időzített történet tulajdonság
Legyen P az alábbiak szerint megadott időzített történet tulajdonság.
A *lenyomat*(P) lenyomat a következő:

Bemeneti:	Belső:
`fogad`$(m)_{1,2}$, $m \in M$	nincs
Kimeneti:	Időátmeneti:
`időhiány`	$\nu(t), t \in \mathbb{R}^+$

Az időzített történetek *id_ történet*(P) halmaza legyen az összes olyan (*művelet*, *idő*) párokból álló β sorozat, amely monoton és nem korlátos, továbbá

1. ha van (`időhiány`, t) tag a β sorozatban, akkor $kl_1 \leq t \leq kl_2 + l$;
2. ha van `időhiány`-t tartalmazó tag β-ban, akkor nincs ezt megelőzően a sorozatban `fogad` műveletet tartalmazó pár;
3. ha nincs `fogad` műveletet tartalmazó tag β-ban, akkor van `időhiány`-t tartalmazó tag β-ban.

A *gen*(P_2) automata, ahol P_2 a 23.1.2. példában megadott MMT automata, kielégíti a P időzített történet tulajdonságot abban az értelemben, hogy $m\_\,i\_$ *történet*$($*gen*$(P_2)) \subseteq$ *id_ történet*(P).

23.3.3.. Szimuláció

A szimuláció módszere éppúgy használható időzítés-alapú rendszerekről való okoskodásra, mint a szinkron és az aszinkron rendszereknél. E célból bevezetjük két általános időzített automata állapotai között értelmezett „időzített szimulációs reláció" fogalmát. A definíció nagyon hasonló a 8.5.5. szakaszban a b/k automatákra bevezetett szimulációs reláció definíciójához.

Legyen A és B két általános időzített automata azonos bemeneti és kimeneti műveletekkel. Tegyük fel, hogy f egy bináris reláció az *állapot*(A) és *állapot*(B) halmazok között; az $u \in f(s)$ jelölés egy alternatív jelölés az $(s, u) \in f$ kifejezésre. Ekkor f *időzített szimulációs reláció* A-ból B-be, ha a következő két feltétel mindegyike teljesül:

1. ha $s \in$ *start*(A), akkor $f(s) \cap$ *start*$(B) \neq \emptyset$;
2. ha s elérhető állapota A-nak és $u \in f(s)$ elérhető állapota B-nek és $(s, \pi, s') \in$ *átmenet(A)*, akkor létezik olyan α időzített végrehajtási sorozati szelet, amely u-val kezdődik és olyan $u' \in f(s')$ állapottal végződik, amelyre
   (a) *id_ történet*$(\alpha) =$ *id_ történet*(s, π, s'),
   (b) Az α-beli időátmenetek összege megegyezik (s, π, s') időátmenetével.

Tehát a kezdő állapotokra kirótt feltétel ugyanaz, mint a b/k automaták esetében. A lépés feltétel egy kicsit különböző – most megköveteljük, hogy a megfeleltetés megőrizze az időzített történetet, tehát látható műveletek sorozatát, párosítva az előfordulásuk idejével, plusz a végrehajtási sorozati összidőt. Megjegyzendő, hogy a lépés feltételben π lehet akár időátmenet, akár diszkrét művelet. Ha π látható művelet, akkor α-nak tartalmaznia kell π lépést, esetleg előtte és/vagy utána lehetnek belső lépések. Ha π belső művelet, akkor α csak belső műveletekből állhat. Ha $\pi = \nu(t)$, akkor α csak időátmeneti és belső műveleteket tartalmazhat és az idejük összege megegyezik t-vel.

Mint korábban, most is igaz, hogy mivel a lépés feltételekben s és u elérhető állapotok, A és B állapotaira vonatkozó invariáns állítások alkalmazhatók, feltéve, hogy f időzített szimulációs reláció.

A következő tétel az időzített szimulációs relációk egy alapvető tulajdonágát fejezi ki.

23.10. tétel. . *Ha létezik időzített szimulációs reláció A-ból B-be, akkor* *m_ i_ történet*$(A) \subseteq$ *m_ i_ történet*(B).

Bizonyítás. A bizonyítást meghagyjuk gyakorlatnak (lásd 23-12. gyakorlat). □

A szakasz hátralévő részében olyan példákat adunk, amelyek megmutatják, hogy az időzített szimulációs reláció hogyan használható időzített rendszerek tulajdonságainak bizonyítására. Az ilyen szimuláció egyik érdekes felhasználása az időzítési feltételeket tartalmazó rendszerek idő korlátjainak bizonyítása. Ez úgy végezhető el, hogy az időzítési specifikációt megfogalmazzuk, mint egy B GTA-t, amelynek *utolsó* és *első* komponense kifejezi az időbeli viselkedés feltételét (a megfelelő felső és alsó korlátokat). A megvalósítást szintén egy A GTA formalizálja, megfelelő *utolsó* és *első* komponensekkel, amelyek az időzítési feltételeket

reprezentálják. A-ból B-be való időzített szimuláció létezése biztosítja, hogy A kielégíti az időzítési követelményeket.

Mivel a szimuláció használható időzítéses esetben az időzítési tulajdonságok bizonyítására, a szimulációs módszer hatásosabb az időzítéses esetben, mint az aszinkron esetben. Az aszinkron esetben gyakran érdekelt bennünket élénkségi tulajdonság, míg az időzítéses esetben gyakran az időzítési korlátok érdekesek számunkra. Az élénkség feltételének formális bizonyítása a szimuláció mellett gyakran felhasznál extra gépezetet, mint a temporális logika, de az időzítési korlátok csupán szimulációt felhasználva is bizonyíthatók.

23.3.3. példa. Szimulációs bizonyítás időhiány folyamat időzítési korlátaira

Megmutatjuk, hogy a 23.1.2. példában adott időhiány MMT automata végrehajt `időhiány` műveletet a $[kl_1, kl_2 + l]$ intervallumban, ha nem érkezik üzenet. A dolog egyszerűsítése végett definiáljuk P_2-nek azt az A változatát, amelynek lenyomata egyáltalán nem tartalmaz `fogad` műveletet. A kódja a következő.

23.8. automata. A

Időzített lenyomat:

Bemeneti:
nincs

Belső:
`csökkent`

Kimeneti:
`időhiány`

Állapotok:

számláló $\in \mathbb{N}$, kezdőértéke k
státusz $\in$ { *aktív*, *bevégzett* }, kezdőértéke *aktív*

Átmenetek:

`csökkent`
Előfeltétel:
státusz = *aktív*
számláló > 0
Hatás:
számláló := *számláló* -1

`időhiány`
Előfeltétel:
státusz = *aktív*
számláló $= 0$
Hatás:
státusz := *bevégzett*

Taszkok és korlátok:

csökk={`csökkent`}, korlátok: $[l_1, l_2]$
időhiány={`időhiány`}, korlátok: $[0, l]$

Ekkor a $gen(A)$ kódja az alábbi.

23.9. automata. $gen(A)$

Időzített lenyomat:

Bemeneti:
nincs
Kimeneti:
`időhiány`

Belső:
`csökkent`
Időátmeneti:
$\nu(t),\ i \in \mathbb{R}^+$

Állapotok:
számláló $\in \mathbb{N}$, kezdőértéke k
státusz $\in$ { *aktív*, *bevégzett* }, kezdőértéke *aktív*

$most \in \mathbb{R}^{\geq 0}$, kezdőértéke 0
$első(csökk) \in \mathbb{R}_0^+$, kezdőértéke l_1
$utolsó(csökk) \in \mathbb{R}^+ \cup \{\infty\}$, kezdőértéke l_2
$utolsó(időhiány) \in \mathbb{R}^+ \cup \{\infty\}$, kezdőértéke ∞

Átmenetek:

`csökkent`
Előfeltétel:
$státusz = aktív$
$számláló > 0$
$most \geq első(csökk)$
Hatás:
$számláló := számláló - 1$
if $számláló > 0$ **then**
$első(csökk) := most + l_1$
$utolsó(csökk) := most + l_2$
else
$első(csökk) := 0$
$utolsó(csökk) := \infty$
$utolsó(időhiány) := most + l$

`időhiány`
Előfeltétel:
$státusz = aktív$
$számláló = 0$
Hatás:
$státusz := bevégzett$
$utolsó(időhiány) := \infty$

$\nu(t)$
Előfeltétel:
$most + t \leq utolsó(csökk)$
$most + t \leq utolsó(időhiány)$
Hatás:
$most := most + t$

Az A automata egyszerűen leszámlál k-tól 0-ig, és aztán végrehajtja az `időhiány` műveletet. Informálisan könnyű belátni, hogy pontosan egy `időhiány` fordul elő a megkívánt $[kl_1, kl_2 + l]$ időintervallumban. Ennek formális bizonyításához az időzítési követelményeket kifejezzük egy triviális magasabb szintű GTA-val. Ez a GTA $gen(B)$ alakú, ahol B a következő triviális MMT automata.

23.10. automata. B

Időzített lenyomat:

Bemeneti:
nincs

Kimeneti:
`időhiány`

Állapotok:

státusz $\in \{$*aktív*, *bevégzett*$\}$, kezdőértéke *aktív*

Átmenetek:

`időhiány`
Előfeltétel:
státusz = *aktív*
Hatás:
státusz := *bevégzett*

Taszkok és korlátok:

időhiány =$\{$`időhiány` $\}$, korlátok: $[kl_1, kl_2 + l]$

Megalkotunk egy *gen*(A)-ból *gen*(B)-be való f időzített szimuláció relációt és ezzel megmutatjuk, hogy A teljesíti az időzítési követelményeket. Ha s állapota *gen*(A)-nak és u állapota *gen*(B)-nek, akkor $(s, u) \in f$ akkor és csak akkor, ha a következő négy feltétel teljesül:

1. $s.most = u.most$;
2. $s.státusz = u.státusz$;
3. $u.utolsó(időhiány) \geq$

$$\begin{cases} s.utolsó(csökk) + (s.számláló - 1) \cdot l_2 + l & \text{ha } s.számláló > 0 \\ s.utolsó(időhiány) & \text{egyébként.} \end{cases}$$

4. $u.első(időhiány) \leq$

$$\begin{cases} s.első(csökk) + (s.számláló - 1) \cdot l_l & \text{ha } s.számláló > 0 \\ s.első(időhiány) & \text{egyébként.} \end{cases}$$

A *most* és a *státusz* értékeket tartalmazó kapcsolatok szerepe nyilvánvaló. Az *utolsó* és az *első* határidőket tartalmazó kapcsolatok az érdekesek. Az $u.utolsó(időhiány)$ érték (*gen*(B)-ben) úgy korlátozott, hogy legalább olyan nagy legyen, mint egy bizonyos érték, amelyet a *gen*(A) állapotából (beleértve a határidő komponenseket is) számítunk ki. Ez az érték felső korlátja annak a legkésőbbi időnek, amelyben a *gen*(A) `időhiány` műveletet hajthat végre. Két eset van: ha *számláló* > 0, akkor az időnek felső korlátja az az idő, amikor az első *csökkent* művelet előfordulhat, plusz az az idő, amely *számláló* $- 1$

pótlólagos *csökkent* és az ezt követő *időhiány* végrehajtási sorozatához kell. Mivel minden csökkentő művelet legfeljebb l_2 ideig tarthat, az időhiány művelet pedig legfeljebb l ideig, így a pótlólagos idő legfeljebb $(számláló - 1) \cdot l_2 + l$. Másrészről, ha $számláló = 0$, akkor az idő felső korlátja az a legkésőbbi idő, amikor *időhiány* előfordulhat. Az egyenlőtlenség azt a tényt fejezi ki, hogy az így számított korlátja a tényleges *időhiány*-nak legfeljebb megegyezik a bizonyítandó felső korláttal.

Az *első*(*időhiány*) egyenlőtlenség értelmezése szimmetrikus – az *első*(*időhiány*) értékei nem lehetnek nagyobbak, mint az a számított felső korlátja annak az időnek, amely a legkorábbi időpont, amikor a $gen(A)$ automata végrehajt `időhiány` műveletet.

Annak bizonyításához, hogy f időzített szimuláció, először bebizonyítunk egy egyszerű invariánst.

23.3.4. állítás. *A $gen(A)$ automata minden elérhető állapotára, ha számláló > 0, akkor státusz = aktív.*

A bizonyítás a szokásos módon folytatódik szimulációval, ami a kezdő feltétel és a lépés feltétel teljesülésének igazolását jelenti. Az egyenlőtlenségeket ugyan úgy kezeljük, mint minden más, állapotok közötti reláció esetében. Mint a 23.3.1. példában, itt is kidolgozunk néhány részletet, hogy modellt adjunk a hasonló bizonyításokra, és gyakorlatokra hagyjuk a többi részletet (lásd 23-15. gyakorlat).

A kezdő feltételt tekintve, legyen s a $gen(A)$, és u a $gen(B)$ automata egyetlen kezdő állapota. Meg kell mutatnunk, hogy $u \in f(s)$. f definíciójának 1. és 2. feltétele közvetlenül teljesül. Tekintsük a 3. feltételt. A $gen(B)$ definíciójából következik, hogy $u.utolsó(időhiány) = kl_2 + l$, a $gen(B)$ definíciójából pedig, hogy $utolsó(csökk) + (s.számláló - 1) \cdot l_2 + l = l_2 + (k-1)l_2 + l = kl_2 + l$. Tehát $u.utolsó(időhiány) = s.utolsó(csökk) + (s.számláló - 1) \cdot l_2 + l$, ami igazolja a 3. feltételt. A 4. feltétel ellenőrzése hasonló a 3. feltételéhez.

A lépés feltételhez tegyük fel, hogy $(s, \pi, s') \in átmenet(gen(A))$, s elérhető, és u elérhető állapot $f(s)$-ben. A műveletek fajtája szerinti esetszétválasztást végzünk, beleértve az időátmeneteket is.

Például, tekintsük azt az esetet, amikor $\pi = csökkent$. Az *időhiány* előfeltétele szerint $s.számláló > 0$. Az $u \in f(s)$ azt jelenti, hogy $s.most = u.most$, $s.státusz = u.státusz$, $u.utolsó(időhiány) \geq s.utolsó(csökk) + (s.számláló - 1) \cdot l_2 + l$ és $u.első(időhiány) \leq s.első(csökk) + (s.számláló - 1) \cdot l_1$. Ez elegendő az $u \in f(s')$ megmutatásához.

Az 1. és 2. feltétel azonnal véghezvihető. Tegyük fel, hogy $s'.számláló > 0$. A 3. feltételhez vegyük észre, hogy az egyenlőtlenség *utolsó*(*időhiány*) bal oldala nem változik ebben a lépésben, a jobb

oldala pedig nem növekszik. Ez utóbbi tulajdonság azért igaz, mert *utolsó*(*csökk*) legfeljebb l_2-vel növekszik, a második tag pedig pontosan l_2-vel csökken és a harmadik tag nem változik. (*utolsó*(*csökk*) azért növekszik legfeljebb l_2-vel, mert $s.most \leq s.utolsó(csökk)$ és $s'.utolsó(csökk) = s.most + l_2$.) Ez azt jelenti, hogy az egyenlőtlenség a lépés után is teljesül. Hasonló érveléssel lehet bizonyítani a 4. feltételt, amikor $s'.számláló = 0$.

A többi fajtájú műveletek esetére más, de hasonló érvelés alkalmazható. Ha $\pi =$ `időhiány`, az érdekes az, hogy megmutassuk, hogy az *első*(*időhiány*) $\leq$ *most* előfeltétel teljesül az u állapotban. Ez az egyenlőtlenség azért teljesül, mert $s.első(időhiány) \leq s.most = u.most$ következik a π művelet $gen(A)$-beli előfeltételéből, és a 4. feltételből következik, hogy $u.első(időhiány) \leq s.első(időhiány)$.

Ha π időátmenet művelet, akkor az $u'.időhiány \leq u.utolsó(időhiány)$ előfeltétel igazolása érdekes. Ez az egyenlőtlenség azért teljesül, mert π $gen(A)$-beli előfeltételéből következik, hogy $u'.most = s'.most \leq min(s.utolsó(csökk), s.utolsó(időhiány))$, és $min(s.utolsó(csökk), s.utolsó(időhiány)) \leq u.utolsó(időhiány)$ következik a 3. feltételből. Az időátmeneti lépések az f definíciójában szereplő értékek közül csak a *most* értéket módosítják, így könnyű megmutatni, hogy megőrzik az f-beli kapcsolatokat.

Mivel f időzített szimuláció, a 23.10. tétel szerint $m\_i\_történet(gen(A)) \subseteq m\_i\_történet(gen(B))$, és így a 23.4. tételből következik, hogy $m\_i\_történet(A) \subseteq m\_i\_történet(B)$. Ez azt jelenti, hogy A kielégíti az időzítési követelményeket.

Természetesen az időzített szimuláció mellett van más módszer is időzítéses rendszerek idő korlátjainak bizonyítására. Például, a 23.13. példa invariánsán alapuló műveleti érveléssel bizonyítható, hogy kl_2+l felső korlát arra az időre, amíg `időhiány` bekövetkezik.

23.3.4. példa. Kéttaszkos verseny

Szimulációs bizonyítást vázolunk annak igazolására, hogy $l + l_2 + Ll$ egy felső korlát arra az időre, amikor a VERSENY automata végrehajt `jelez` műveletet. Most a specifikációt az $gen(B')$ adja, ahol a B' MMT automata definíciója hasonló a 23.3.3. példa B automatájáéhoz.

23.11. automata. B'

Időzített lenyomat:

Bemeneti: nincs

Kimeneti: `jelez`

Állapotok:

jelzett logikai, kezdőértéke *hamis*

Átmenetek:

`jelez`
Előfeltétel:
$jelzett = hamis$
Hatás:
$jelzett := igaz$

Taszkok és korlátok:

jelez={`jelez`}, korlátok: $[0, l + l_2 + Ll]$

Intuitíven, az ok, hogy $l + l_2 + Ll$ helyes felső korlát, a következő. Az *meg_szakítás* taszk l időn belül *igaz*-ra állítja a *jelző* értékét. A legnagyobb érték, amit a *számláló* ez alatt az idő alatt elérhet $\frac{l}{l_1}$. Ezután legalább $\frac{l}{l_1} l_2 = Ll$ idő kell ahhoz, hogy a *fő* taszk 0-ra csökkentse a *számláló* értékét, és még legfeljebb l_2 idő kell a `jelez` végrehajtási sorozatához.

Most definiáljuk a g időzített szimulációt a $gen(\textsc{Verseny})$ és a $gen(B')$ automata között. Ha s állapota $gen(\textsc{Verseny})$-nek és u állapota $gen(B')$-nek, akkor $(s, u) \in g$ akkor és csak akkor, ha a következő három feltétel teljesül:

1. $s.most = u.most$;
2. $s.jelzett = u.jelzett$;
3. $u.utolsó(jelez) \geq$

$$\begin{cases} s.utolsó(meg\_szakítás) + (s.számláló + 2) \cdot l_2 + \\ +L(s.utolsó(meg\_szakítás) - s.első(fő)), \\ \quad \text{ha} \quad s.jelző = hamis \quad \text{és} \quad s.első(fő) \leq s.utolsó(meg\_szakítás) \\ s.utolsó(fő) + (s.számláló)l_2, \quad \text{egyébként.} \end{cases}$$

A harmadik feltételt az alábbi indokolja. Ha $jelző = igaz$, akkor a *jelez*-ig hátralévő idő pontosan a *fő* taszk megmaradt *csökkent* lépéseinek ideje plusz a végső *jelez* ideje. Hasonló okfejtés alkalmazható, ha *jelző* értéke még *hamis*, de *igaz*-ra kell váltania, mielőtt van idő másik *növel* előfordulására, vagyis $s.első(fő) > s.utolsó(meg\_szakítás)$. Egyébként, $jelző = igaz$ és $s.első(fő) \leq s.utolsó(meg\_szakítás)$, ami azt jelenti, hogy van még idő legalább egy *növel* előfordulására. Ezután alkalmazható az egyenlőtlenség első esete a $utolsó(jelez)$-re.

Ebben az esetben, *beállít* után $(számláló + 1)l_2$ ideig tart, amíg a *fő* taszk leszámlál és végrehajtja a *jelez* műveletet. De a *számláló* aktuális értéke növekedhet a *fő* taszk előtt bizonyos *növel* műveletek hatására. Ezek legnagyobb lehetséges száma $1 + [utolsó(meg\_szakítás) - első(fő)]/l_1$. Ezt szorozva l_2-vel megkapjuk a *számláló* pótlólagos csökkentéséhez szükséges időt.

Annak bizonyítása, hogy g időzített szimuláció reláció, fő vonalaiban hasonlóan végezhető, mint a 23.3.3. példa bizonyítása. Ennek elvégzését gyakorlatra hagyjuk (lásd 23-16. gyakorlat).

23.4.. Közös memóriájú és hálózati rendszerek modellezése

Ezt a fejezetet azzal zárjuk, hogy megmutatjuk, hogyan modellezhetők a részben szinkron közös memóriájú rendszerek és a részben szinkron hálózatok MMT és GTA automatákkal. Ezeket a modelleket a 24. és 25. fejezetben fogjuk használjuk.

23.4.1.. Közös memóriájú rendszerek

Részben szinkron közös memóriájú rendszert egy (A, b) MMT automatával modellezzük. Feltételezzük, hogy az A b/k automata a 9. fejezetben megadott definíció szerinti aszinkron közös memóriájú rendszer, az egyetlen új követelmény, hogy A-nak csak véges sok taszkja van. A b korlátfüggvény korlátokat rendel a taszkokhoz.

A legtöbb esetben feltételezzük, hogy minden folyamatnak csak egy taszkja van, és a korlátfüggvény minden taszkhoz ugyanazt az l_1 alsó és az l_2 felső korlátot rendeli, ahol $0 < l_1 \leq l_2 < \infty$. Ekkor az $L = l_1/l_2$ jelölést alkalmazzuk, mint korábban is tettük, ahol L a rendszer időzítési bizonytalanságának a mértéke.

23.4.2.. Hálózatok

A részben szinkron esetben csak küld/fogad hálózatokat tekintünk, sem üzenetszóró, sem többletes üzenetszóró hálózatokat nem vizsgálunk. Feltételezzük, hogy a $G = (V, E)$ irányított gráf tartozik a hálózathoz. A részben szinkron küld/fogad hálózati rendszer modellje a gráf csúcsaihoz rendelt folyamat automatákból és az éleihez rendelt csatorna automatákból áll.

A gráf minden i csúcsához a P_i MMT automata tartozik. P_i-nek vannak bemenet és kimenet műveletei, amelyekkel kommunikál a külső felhasználóval, és ezeken felül van $küld(m)_{i,j}$ kimeneti művelete, ahol m üzenet, j kimeneti szomszéd, és van $fogad(m)_{i,j}$ bemeneti művelete, ahol j bemeneti szomszéd. A folyamat megállási hiba modellezésére bevezetünk $\texttt{megállít}_i$ bemeneti műveleteket. A $\texttt{megállít}_i$ műveleteknek az a hatása, hogy véglegesen blokkolja a P_i folyamat minden taszkját. Általában feltételezzük, hogy minden P_i folyamat minden (véges sok) taszkjának l_1 és l_2 a korlátja, ahol $0 < l_1 \leq l_2 < \infty$.

Minden (i, j) irányított élhez tartozó csatorna automata a $C_{i,j}$ GTA. Ennek „látható felülete" a $küld(m)_{i,j}$ bemeneti és $fogad(m)_{i,j}$ kimeneti műveletekből áll. A csatorna külső viselkedésére vonatkozó megszorításokat egy P időzített történet tulajdonsággal fejezünk ki. A P által definiált csatornák azok a GTA-k, amelyek látható műveletei megegyeznek P műveleteivel, és amelyek megengedett időzített történetei elemei $id\_történet(P)$-nek. Két általános eset van:

1. Minden $C_{i,j}$ megegyezik a 23.2.1. példában adott $D'_{i,j}$ GTA-val, vagyis megbízható FIFO csatorna automata, a legkésőbbi üzenet kikerülési idejének felső korlátja d.

2. Minden $C_{i,j}$ megegyezik a 23.2.2. példában adott $D''_{i,j}$ GTA-val, vagyis megbízható FIFO csatorna automata, és minden üzenet kikerülési idejének felső korlátja d.

Ismét használjuk az $L = l_1/l_2$ jelölést a rendszer időzítési bizonytalanságára.

23.5.. Megjegyzések a fejezethez

Az MMT automata modellt Merritt, Modugno és Tuttle tervezte [227]. A modelljük valamivel általánosabb, mint amit ebben a könyvben használunk, mert megengednek végleges és valós értékű felső korlátokat is. Az itt használt modell változat nagyon közeli ahhoz, amit Lynch és Attiya definiált [215]. A kéttaszkos verseny példát Pnueli [243] javasolta mint teszt esetet az időzítés-alapú rendszerek bizonyítási módszerére.

Az általános időzített automata modell Lynch és Vaandrager [210, 212, 211] időzített automata modelljén alapszik, ami hasonló Alur és Dill [9] időzített automatájához. Megengedett időzített végrehajtási sorozatokat Gawlick, Segala, Søgaard-Andersen és Lynch [136] tanulmányozott. Az MMT automata transzformációját általános időzített automatává Lynch és Attiya [215] alkotta. GTA-kon értelmezett műveletek [212]-ből származnak, amely könyv az összekapcsolás és elrejtés mellett több más műveletet is leír, többek között a szekvenciális összekapcsolást, különféle választást, megszakítást és időhiányt.

Határidőket tartalmazó invariánsokat Tel [275], Lewis [194], Shankar [259], Abadi és Lamport [1], Lynch [204] és mások használtak. Időzítési tulajdonságok bizonyítására szimulációs módszert először Lynch és Attiya [215] használt. A 23.3.3. példában szereplő időkorlát szimulációs bizonyítása [215]-ből és az [204] [205] áttekintő cikkekből származik. Más típusú GTA szimulációkat definiált Lynch és Vaandrager [210, 211].

Egyéb időzített szimulációs bizonyításokat végzett Søgaard-Andersen [210, 211], Lampson és Lynch [264, 190], Heitmeyer és Lynch [148] és Luchangco [201]. Kezdeti kutatásokat végzett Luchangco, Söylemez, Garland, és Lynch automatikus tételbizonyító segítségével végzendő szimulációs bizonyításokra [202]. Kutatásaikban a Larch Prover [134] rendszert használták.

23.6.. Gyakorlatok

23-1. Legyen (A, b) tetszőleges MMT automata és α egy tetszőleges időzített végrehajtási sorozata (A, b)-nek. Bizonyítsuk be a következőket.

(a) Létezik (A, b)-nek olyan (megengedett) időzített végrehajtási sorozata, amely α-val kezdődik.

(b) Legyen β bemeneti művelet és idő párok olyan tetszőleges véges sorozata, hogy az idők sorozata nemcsökkenő, és mindegyik legalább akkora, mint a legnagyobb idő α-ban. Ekkor létezik (A, b)-nek olyan α' (megengedett) időzített végrehajtási sorozata, amely α-val kezdődik és β azon bemeneti műveletekből álló részsorozat, amelyek ideje α'-ből azok, amelyek α után vannak.

(c) Legyen β

(c) Legyen β bemeneti művelet és idő párok olyan tetszőleges véges sorozata, hogy az idők sorozata nemcsökkenő és nem korlátos, és mindegyik legalább akkora, mint a legnagyobb idő α-ban. Ekkor létezik (A, b)-nek olyan α' (megengedett) időzített végrehajtási sorozata, amely α-val kezdődik és β azon bemeneti műveletekből álló részsorozat, amelyek ideje α'-ből azok, amelyek α után vannak.

23-2. Tegyük fel, hogy az MMT automata definíciójában megengedjük, hogy véges sok helyett megszámlálhatóan végtelen sok taszk is lehet. Mutassuk meg, hogy ekkor létezik az új definíció szerinti olyan (A, b) MMT automata, amelynek nincs (megengedett) időzített végrehajtási sorozata.

23-3. Írjuk le körültekintően a 23.1.4. példában adott összetett MMT automata viselkedését arra az esetre, amikor $kl_1 \leq l + d$.

23-4. Bizonyítsuk be a 23.1., 23.2. és 23.3. tételeket.

23-5. Tekintsük a 15.4. alfejezetben adott AszinkBellmanFord algoritmusnak azt az időkorlátos vátozatát, ahol

- minden folyamat automata az az MMT automata, amely az adott b/k automatából áll, és minden taszk korlátja $[l_1, l_2]$, $0 < l_1 \leq l_2 < \infty$;
- minden csatorna a 23.1.1. példában definiált $D_{i,j}$ MMT automata.

Elemezzük az algoritmus kommunikációs és idő bonyolultságát.

23-6. Bizonyítsuk be, hogy a 23.2.1. példában definiált $D'_{i,j}$ GTA és a 23.1.1. példában definiált $D_{i,j}$ MMT automata megengedett időzített történeteinek halmaza megegyezik.

23-7. Bizonyítsuk be, hogy nem létezik olyan MMT automata, amelynek megengedett időzített történeteinek halmaza megegyezik a 23.2.2. példában definiált $D''_{i,j}$ GTA megengedett időzített történeteinek halmazával.

23-8. Adjuk meg előfeltétel/hatás kódját egy olyan A GTA-nak, amelynek a viselkedése a következő: A minden megengedett időzített végrehajtási sorozatában pontosan két, a és b kimenet műveletet hajt végre ebben a sorrendben,

mindkettőt 1 időegység alatt. Továbbá, mind a, mind b tetszőleges időben előfordulhat több megengedett időzített végrehajtási sorozatban, a fenti korlátozással. Bizonyítsuk be, hogy nem létezik olyan MMT automata, amelynek megengedett időzített történetei megegyeznek A-éval.

23-9. Adjuk meg explicit a 23.1.3. példában definiált VERSENY MMT automatára a *gen*(VERSENY) GTA előfeltétel/hatás kódját. A kód stílusa hasonló legyen a 23.2.4. példa kódjáéhoz.

23-10. Bizonyítsuk be a 23.7., 23.8. és 23.9. tételeket.

23-11. Mutassuk meg, hogy a 23.8. tétel átfogalmazott állítása, hogy $\alpha_i = \alpha | B_i$ elhagyva az időátmenet ekvivalenciát, hamis.

23-12. Bizonyítsuk be a 23.10. tételt.

23-13. Bizonyítsuk be a 23.2.4. példában adott A' rendszerre a következő többrészes invariánst: ha $státusz_1 = halott$, akkor

(a) a sor üres;
(b) $státusz_2 \neq hibás$;
(c) ha $számláló_2 > 0$, akkor $utolsó(csökk) + (számláló_2 - 1)l_2 \leq kl_2$.
(d) ha $számláló_2 = 0$, akkor $utolsó(időhiány) \leq kl_2 + l$.

23-14. A 23.3.4. alfejezetben tárgyalt szimuláció módszerével bizonyítsuk be a 11.3. lemmában a JOBBBALIVÓFIL algoritmusra adott időkorlátot.

23-15. Tegyük teljessé a 23.3.3. példa bizonyítását, hogy f időzített szimuláció reláció, a 23.3.3 példa felhasználásával.

23-16. Bizonyítsuk be, hogy a 23.3.4. példában definiált g reláció időzített szimuláció reláció.

24. fejezet

Kölcsönös kizárás részleges szinkronizációval

Ebben a fejezetben harmadszor is visszatérünk a kölcsönös kizárási problémára – ezúttal részben szinkronizált közös memóriát feltételezve. Csak az alapvető eredményeket ismertetjük: egyszerű időzítés-alapú algoritmusokat és ezek elemzését, továbbá könnyen érthető megoldhatatlansági eredményeket.

24.1.. A feladat

A 10. fejezetéhez nagyon hasonló elrendezést vizsgálunk: az $U_1,\ldots,U_n$ felhasználókkal kölcsönhatásban álló n kapus közös memóriás rendszert. A $\texttt{próbál}_i$, $\texttt{belép}_i$, $\texttt{kilép}_i$ és $\texttt{halad}_i$ műveletekből álló külső felület pontosan megegyezik a korábbival. Most azonban a felhasználókat és a közös memóriás rendszert nem b/k automatával, hanem a 23.1. alfejezetben definiált MMT automatával modellezzük. A fejezetben tekintett architektúrát mutatja a 10.4. ábra.

A korábbiakhoz hasonlóan minden U_i felhasználótól megköveteljük a jólformáltság megőrzését. A felhasználókra vonatkozó tetszőleges időzítési megszorításokat megengedünk. Formálisan tekintve minden U_i MMT automata (A_i, b_i) alakú, ahol A_i a 10.2. alfejezetben bevezetett, csupán véges sok taszkkal rendelkező b/k automata, b_i pedig tetszőleges korlát. Megengedettek a triviális korlátok, így a 0 triviális alsó korlát és a ∞ triviális felső korlát is.

A rendszer további részét a közös memóriás rendszert reprezentáló egyetlen $B = (A, b)$ MMT automata alkotja. Az alapul szolgáló A b/k automata olyan alakú, mint amilyet a 10. fejezetben az aszinkron közös memóriás modell keretében a kölcsönös kizárási feladat megoldására használtunk. Kapunként egy, összesen n folyamatból áll. A fejezet során végig feltesszük, hogy egy folyamathoz csupán egy taszk tartozik. A b korlát minden taszkhoz olyan ℓ_1 alsó és ℓ_2 felső korlátot rendel, melyekre $0 < \ell_1 \leq \ell_2 < \infty$ teljesül. A korábbiakhoz hasonlóan használjuk a $L = \ell_2/\ell_1$ hányadost, amely a rendszer *időzítési bizonytalanságának* mértéke.

A fejezet során három további megszorítást teszünk. Először is a folyama-

tok műveleteit a 10. fejezetével megegyező módon korlátozzuk: a közös memóriás rendszerben a P_i folyamathoz tartozó egyetlen taszk csak akkor megengedett, ha U_i a Pr vagy a Ki szakaszban van. Másodszor azt is feltesszük, hogy a P_i folyamat egyetlen taszkja valóban engedélyezve van minden olyan esetben, amikor U_i a Be vagy a Ki szakaszban található. (Ámbár azt a lehetőséget sem zárjuk ki, hogy az egyetlen megengedett művelet az üres művelet legyen, amely nem okoz állapot-átmenetet.) Harmadszor, csak olvas-ír változókat használó közös memóriás rendszereket vizsgálunk.

Nagyjából ugyanazokat a helyességi feltételeket követeljük meg, mint a 21. fejezetben. Időzített automatákra a következőképpen mondhatók ki.

Jólformáltság. Az egyesített rendszer bármely időzített végrehajtása során minden i-re teljesül, hogy az U_i és a (B, b) közti kölcsönhatást leíró szelet i-re vonatkozóan jólformált.

Kölcsönös kizárás. a rendszernek nincs olyan elérhető állapota, amelyben egynél több felhasználó található a Be kritikus szakaszban.

Haladás. Bármely *megengedett időzített végrehajtási sorozat*[1] során

1. (A próbál szakaszra vonatkozó haladás) Ha legalább egy felhasználó Pr-ben van, és egyik sincs Be-ben, akkor egy későbbi időpontban valamelyik felhasználó belép Be-be.
2. (A kilép szakaszra vonatkozó haladás) Ha van legalább egy felhasználó Ki-ben, akkor később valamelyikük belép Ha-ba.

Azt mondjuk, hogy B *megoldja a kölcsönös kizárási feladatot*, ha tetszőlegesen választott felhasználók esetében megoldást nyújt (azaz garantálja a jólformáltságot, a kölcsönös kizárást és a haladást). Ezeket a helyességi feltételeket egy P időzített történet tulajdonsággal is megfogalmazhattuk volna a 23.3.2. szakaszban definiált módon.

24.2.. Egyregiszteres algoritmus

Ebben a szakaszban bemutatjuk a FischerKK részben szinkron kölcsönös kizárási algoritmust, amely csupán egyetlen olvasható/írható regisztert használ. Már ez az egyszerű algoritmus is szemlélteti a részben szinkron és az aszinkron modell közti nagy különbséget, hiszen – amint azt a 10.33. tételben megmutattuk – bármely olvasható/írható közös memóriájú szinkron kölcsönös kizárási algoritmushoz legalább n osztott regiszter szükséges.

Az algoritmus kiindulópontja a következő helytelen *aszinkron* algoritmus.

HibásFischerKK (vázlatosan)

Az algoritmus a *váltás* nevű egyetlen ír/olvas közös változót használja, melyet bármelyik folyamat írhat és olvashat. Ha a P_i folyamat hozzá akar férni az erőforráshoz, ismételten teszteli *váltás* értékét, míg csak 0 nem lesz. Ha a tesztelés eredménye *váltás* $= 0$, akkor a P_i folyamat *váltás* értékét saját sorszámára, i-re állítja be. Ezután újra ellenőrzi, hogy *váltás* még mindig i-e. Ha igen, a P_i folyamat belép a kritikus szakaszba; ha nem, elölről kezdi a *váltás* $= 0$ vizsgálatával. Ha a P_i folyamat kilép, *váltás* értékét visszaállítja 0-ra.

A 10. fejezet közös memóriás programjaihoz hasonló írásmódot használva mindezt a következőképpen adhatjuk meg.

[1] A 23.1. alfejezethez hasonlóan ezt úgy definiáljuk, hogy az idő normálisan telik, és a feldolgozás akkor sem áll meg, ha több elvégzendő munka van.

24.1. algoritmus. A HibásFischerKK

Osztott változók:
váltás $\in \{0, 1, \ldots, n\}$, kezdőértéke 0, bármely folyamat írhatja és olvashatja.

A P_i folyamat:

** a maradék szakasz **	** a kritikus szakasz **
`próbál`$_i$	`kilép`$_i$
L: **if** *váltás* $\neq 0$ **then goto** L	*váltás* $:= 0$
váltás $:= i$	`halad`$_i$
if *váltás* $\neq i$ **then goto** L	
`belép`$_i$	

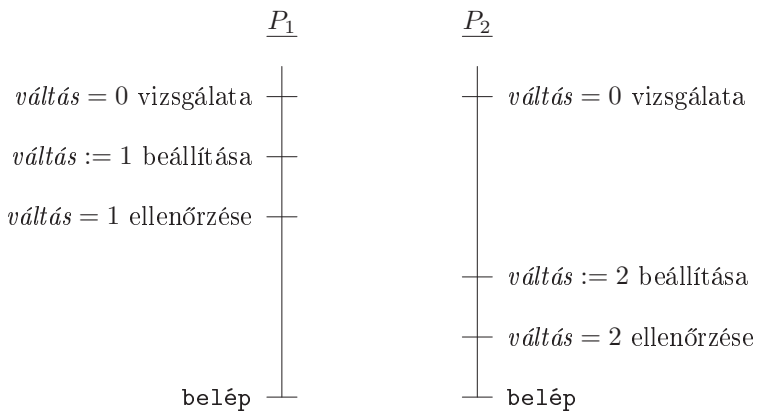
A HibásFischerKK algoritmus azért hibás, mert nem tudja biztosítani a kölcsönös kizárást. (Tudjuk is, hogy hibásnak kell lennie, különben ellentmondana a 10.33. tételnek.)

24.2.1. példa. A HibásFischerKK algoritmussal kapott hibás végrehajtási sorozat

Tekintsünk egy olyan végrehajtási sorozatot, amely során mindkét résztvevő folyamat, 1 és 2, megvizsgálja *váltás*-t, és mindkettő 0-nak találja. Az 1 folyamat azonnal végrehajtja a *váltás* := 1 értékadást, majd rögtön ellenőrzi is, s azt találja, hogy *váltás* = 1. Ezután a 2 folyamat hajtja végre a *váltás* := 2 értékadást, megvizsgálja *váltás*-t, és azt találja, hogy *váltás* = 2. Ezt követően mindkét folyamat tovább lép a kritikus szakaszba. A végrehajtást a 24.1. ábra mutatja.

Az események ilyen szerencsétlen összejátszása elkerülhető, ha egyszerű időzítési korlátozásokat veszünk fel. Nevezetesen, ha a P_i folyamat végrehajtotta a *váltás* := i értékadást, *váltás* ellenőrzését elhalaszthatja ℓ_2-nél, a folyamat lépésidejének feltételezett felső korlátjánál hosszabb időre. Az összes többi lépés normális sebességgel, a folyamat egymást követő lépései között valamely $[\ell_1, \ell_2]$ időintervallumba eső részidőkkel zajlik. Ez a megszorítás kizárja a 24.1. ábrán látható szerencsétlen összjátékot, mivel a P_i folyamatot *váltás* := i beállítása után az ellenőrzés végrehajtása előtt elég sokáig várakoztatjuk annak biztosításához, hogy bármely j folyamat, amely még esetleg a *váltás* := i értékadás előtt ellenőrizte *váltás*-t (és azután saját indexének megfelelően szeretné beállítani), már végrehajtsa az értékadást. Vagyis mire a P_i folyamat eljut az ellenőrzéshez, már nem lesznek a *váltás*-t átállítani szándékozó másik folyamatok.

A feltételt biztosító kód ismertetése következik. Ennek során feltesszük, hogy a_1 és a_2 az $\ell_2 < a_1 \leq a_2$ feltételt teljesítő pozitív valós számok. Figyeljük meg, hogy a kód minden P_i folyamathoz két taszkot ad meg: `fő`$_i$-t az $[\ell_1, \ell_2]$ és `vizsgál`$_i$-t az $[a_1, a_2]$ korlátokkal. Ezt techikailag nem engedhető meg, hiszen



24.1.. ábra. A HibásFischerKK algoritmussal kapott hibás végrehajtási sorozat.

modellünk minden folyamathoz csak egyetlen taszkot engedélyez az $[\ell_1, \ell_2]$ korlátokkal. Az algoritmus azonban egyszerűen módosítható: minden `vizsgál` elé illesszünk be $k-1$ explicit `késleltet` lépést úgy, hogy $k\ell_1 > \ell_2$ legyen, és foglaljuk össze az egy folyamathoz tartozó műveleteket egy $[\ell_1, \ell_2]$ korlátos taszkba. Az eredményül kapott algoritmus „úgy viselkedik", mint a FischerKK algoritmus az $a_1 = k\ell_1$ és az $a_2 = k\ell_2$ korlátokkal. A formális részleteket elhagyjuk.

24.2. algoritmus. FischerKK

Közös változók:
váltás $\in \{0, 1, \ldots, n\}$, kezdőértéke 0

P_i műveletei:

Bemeneti:
- `próbál`$_i$
- `kilép`$_i$

Kimeneti:
- `belép`$_i$
- `halad`$_i$

Belső:
- `vizsgál`$_i$
- `beállít`$_i$
- `ellenőriz`$_i$
- `visszaállít`$_i$

P_i állapotai:
elő_felt $\in \{$*halad*,*vizsgál*,*beállít*,*ellenőriz*,*elhagy_próbál*,*belép*, *visszaállít*, *elhagy_kilép*$\}$,
kezdőértéke *halad*

P_i átmenetei:

`próbál`$_i$
Hatás:
elő_felt := *vizsgál*

`vizsgál`$_i$
Előfeltétel:
elő_felt = *vizsgál*
Hatás:
if *váltás* $= 0$ **then**
elő_felt := *beállít*

`belép`$_i$
Előfeltétel:
elő_felt = *elhagy_próbál*
Hatás:
elő_felt := *kritikus*

`kilép`$_i$
Hatás:
elő_felt := *visszaállít*

`beállít`$_i$
Előfeltétel:
elő_felt=*beállít*
Hatás:
váltás := i
elő_felt := *ellenőriz*

`ellenőriz`$_i$
Előfeltétel:
elő_felt = *ellenőriz*
Hatás:
if *váltás* = i **then**
elő_felt := *elhagy_próbál*
else *elő_felt* := *vizsgál*

`visszaállít`$_i$
Előfeltétel:
elő_felt = *visszaállít*
Hatás:
váltás := 0
elő_felt := *elhagy_kilép*

`halad`$_i$
Előfeltétel:
elő_felt = *elhagy_kilép*
Hatás:
elő_felt := *halad*

A taszkok és korlátaik:
$fő_i = \{$`vizsgál`$_i$, `beállít`$_i$, `belép`$_i$, `visszaállít`$_i$, `halad`$_i\}$, a korlátok $[\ell_1, \ell_2]$
$ellenőriz_i = \{$`ellenőriz`$_i\}$, a korlátok $[a_1, a_2]$

24.1. tétel. . *A* FischerKK *algoritmus $\ell_2 < a_1$ esetében megoldja a kölcsönös kizárási feladatot.*

Bizonyítás. A FischerKK algoritmust felhasználók tetszőleges halmazára alkalmazzuk. A jólformáltság könnyen belátható. A kölcsönös kizárási tulajdonság teljesülését az egyesített rendszerre (az algoritmus plusz a felhasználók) teljesülő következő invariáns állítás segítségével kívánjuk igazolni.

> **24.2.1. állítás.** *Egyetlen elérhető állapotban sem létezik olyan i és j, hogy $i \neq j$ és elő_felt$_i$ = elő_felt$_j$ = belép.*

Az állítás indukciós bizonyításához, amint azt megszoktuk, további „segédinvariánsokra” lesz szükség. Most azonban olyan invariánsok kellenek, amelyek a közönséges programváltozók mellett az időzítésre vonatkozó információt is tartalmaznak.

Ezért a rendszert a 23.2.2. szakaszban leírt általánosított időzítésű automatává (ÁIA) alakítjuk át. Ez a transzformáció nem az időzített végrehajtásokra vonatkozó „külső” kikötésekkel fejezi ki az időzítési megszorításokat, hanem a rendszer állapotaival és átmeneteivel kódolja őket. Így például az állapot olyan *első*(*ellenőriz*$_i$) és *utolsó*(*fő*$_i$) komponenseket tartalmaz, melyek azt a legkorábbi, illetve legkésőbbi időpontot adják meg, amikor a következő `ellenőriz`$_i$ művelet elvégezhető, illetve a `fő`$_i$ következő művelete még végrehajtható. A 24.2.1. állítást és az alábbiakban található hasonló észrevételeket úgy tekintjük, mint az ezen transzformációval kapott ÁIA állapotainak tulajdonságait.

A következő, indukcióval bizonyítható kulcsfontosságú állítás szerint az a legkorábbi jövőbeli időpont, amikor egy sikeres `ellenőriz`$_i$ hajtható végre, már azutánra esik, hogy bármely másik, a `vizsgál`$_j$-n sikerrel túljutó P_j folyamat végrehajtotta `beállít`$_j$-t is. Ezt az állítást használjuk a 24.2.1. példában látott hibás helyzet kiszűrésére.

24.2.2. állítás. *Minden elérhető állapotra teljesül a következő: ha elő_felt$_i$ = ellenőriz, váltás = i és elő_felt$_j$ = beállít, akkor első(ellenőriz$_i$) > utolsó(fő$_j$).*

A 24.2.2. állítás a kérdéses állapothoz vezető időzített végrehajtási lépések száma szerinti egyszerű indukcióval bizonyítható. Most a közönséges bemeneti, kimeneti és belső lépések mellett beszámítjuk az idő múlásával kapcsolatos lépéseket is. A 23.3.1. példa modellként szolgálhat az ilyen bizonyítások menetére vonatkozóan. A 24.2.2 állítással kapcsolatban azok az (s, π, s') alakú lépésekkel kapcsolatos megfontolások az érdekesek, melyekben π vagy egy $\texttt{beállít}_i$, vagy egy olyan sikeres $\texttt{vizsgál}_j$, ahol $j \neq i$ (az állítás kimondásában szereplő i és j indexekkel).

1. $\pi = \texttt{beállít}_i$.
   Ekkor $s'.első(ellenőriz_i) = s.most + a_1$. Továbbá a 23.5. segédtétel szerint ha $s'.elő\_felt_j = beállít$, akkor $s'.utolsó(fő_j) \leq s.most + \ell_2$. Mivel $\ell_2 < a_1$, megkaptuk a bizonyítandó egyenlőtlenséget.

2. $\pi = \texttt{vizsgál}_j$ és $s.váltás = 0$ (vagyis a vizsgálat sikeres volt).
   Ekkor $s'.váltás = 0$, tehát az állítás üresen teljesül.

A 24.2.2. állítás segítségével igazolható a következő észrevétel, amely azt mondja ki, hogy ha a P_i folyamat a kritikus szekcióban, közvetlenül előtte, vagy közvetlenül utána van, akkor *váltás* $= i$, és egyetlen másik folyamat sem készülhet éppen *váltás* beállítására. Figyeljük meg, hogy a 24.2.2. állítástól eltérően a 24.2.3. állítás nem tartalmaz semmi időzítéssel kapcsolatos információt. Indukciós bizonyítása során azonban felhasználunk majd időzítési információkat.

24.2.3. állítás. *Minden elérhető állapotra teljesül a következő: ha elő_felt$_i$ $\in$ {elhagy_próbál, belép, visszaállít}, akkor váltás $= i$ és minden j-re igaz elő_felt$_j$ $\neq$ beállít.*

A bizonyítás ismét indukcióval történik. Most azokra az (s, π, s') lépésekre vonatkozó meggondolások az érdekesek, amelyekben π egy sikeres $\texttt{ellenőriz}_i$, $\texttt{beállít}_j$ vagy $\texttt{visszaállít}_j$ és $j \neq i$; illetve olyan sikeres $\texttt{vizsgál}_j$, amikor $j \neq i$.

1. $\pi = \texttt{ellenőriz}_i$ és $s.váltás = i$ (vagyis az ellenőrzés sikeres volt).
   Ekkor $s'.váltás = i$. Tegyük fel, hogy van olyan j, amelyre $s'.elő\_felt_j = beállít$ teljesül. Ekkor $s.elő\_felt_j = beállít$ is igaz. A 24.2.2. állításból következik, hogy $s.első(\texttt{ellenőriz}_i) > s.utolsó(\texttt{fő}_j)$. De a 23.5. segédtétel azt mondja ki, hogy $s.most \leq s.utolsó(\texttt{fő}_j)$. Így tehát $s.első(\texttt{ellenőriz}_i) > s.most$, ami ellentmond az ÁIA-ra vonatkozó időzítési megszorításoknak. Tehát nincs olyan j, amelyre $s'.elő\_felt_j = beállít$ lenne.

2. $\pi = \texttt{beállít}_j$ és $j \neq i$.
   Tételezzük fel, hogy $s'.elő\_felt_i \in \{elhagy\_próbál, kritikus, visszaállít\}$. Ekkor $s.elő\_felt_i \in \{elhagy\_próbál, kritikus, visszaállít\}$. Az indukciós feltevésből következik, hogy nincs olyan j, amelyre $s.elő\_felt_j = beállít$. De ekkor π s-ben nem megengedett, ami ellentmondás.

3. $\pi = \texttt{visszaállít}_j$ és $j \neq i$.
   Tételezzük fel, hogy $s'.elő\_felt_i \in \{elhagy\_próbál, kritikus, visszaállít\}$. Ekkor $s.elő\_felt_i \in \{elhagy\_próbál, kritikus, visszaállít\}$, és az indukciós feltevés miatt $s.váltás = i$. De mivel π megengedett s-ben, $s.elő\_felt_j = visszaállít$, s így – az indukciós feltevés miatt – $s.váltás = j$, ami ellentmondás.

4. $\pi = \texttt{vizsgál}_j$, $j \neq i$ és $s.váltás = 0$ (vagyis a vizsgálat sikeres volt).
   Az indukciós feltétel szerint $s.elő\_felt_i \notin \{elhagy\_próbál, kritikus, visszaállít\}$, így $s.'elő\_felt_i \notin elhagy\_próbál, kritikus, visszaállít\}$ is igaz, ami azt jelenti, hogy a feltétel üresen teljesül.

A 24.2.1. állításban megfogalmazott kölcsönös kizárási tulajdonság közvetlenül adódik a 24.2.3. állításból.

Végül vizsgáljuk meg a haladási feltételt. Ehhez hasznunkra lesz egy további, indukcióval könnyen belátható állítás.

24.2.4. állítás. *Minden elérhető állapotra igaz, hogy ha váltás* $= i$*, akkor elő_felt*$_i \in \{ellenőriz, elhagy\_próbál, kritikus, visszaállít\}$.

A működés alapján beláthatjuk a haladást, ha felhasználjuk a 24.2.4. állítást, és hasonlóan érvelünk, mint a DijkstraKK algoritmus haladását kimondó 10.4. segédtétel bizonyításában. Azaz tekintsünk egy olyan α megengedett időzítésű végrehajtást, amely egy olyan pontig jutott el, amikor van legalább egy felhasználó Pr-ben, de Be-ben nincs senki. Indirekt módon tételezzük fel, hogy ezután sem lép be egyetlen felhasználó sem Be-be. Ekkor belátható, hogy valahonnan kezdve α-ban nem fordulnak elő szakaszváltások, minden folyamat Pr-ben vagy Ha-ban van, és legalább egy folyamat Pr-ben tartózkodik. A 24.2.4. állítást felhasználva megmutathatjuk, hogy *váltás* értéke előbb-utóbb valamelyik versengő (Pr-ben lévő) folyamat indexét veszi fel. Ezután *váltás* csak ilyen értékeket vehet fel, bár változhat, hogy éppen melyik versengő folyamat indexét kapja meg. Legvégül azonban *váltás* értéke stabilizálódik, beáll mondjuk i-re. Ismét a 24.2.4. állítást felhasználva ebből már következik, hogy a P_i folyamat belép Be-be, ami ellentmondás.

Ezzel a 24.1. tétel bizonyítását befejeztük. □

Vizsgáljuk meg a FischerKK algoritmus időbonyolultságát!

24.2. tétel. . *A* FischerKK *algoritmus bármely időzített végrehajtása teljesíti az alábbiakat.*

1. *Ha valamely időpontban valamely P_i folyamat a próbál szakaszban van, akkor legfeljebb $2a_2 + 5\ell_2$ idő múlva belép egy folyamat a kritikus szakaszba.*
2. *Ha valamely időpontban valamely P_i folyamat a kilép szakaszban van, akkor legfeljebb $2\ell_2$ időre van szükség ahhoz, hogy a fennmaradó szakaszba kerüljön.*

Bizonyítás. A *kilép* szakaszra vonatkozó korlát magától értetődő. A *próbál* szakaszra vonatkozó korlátot bizonyíthatnánk a működésre alapozva, de a változatosság kedvéért a 23.3.3. szakaszban leírt időzített szimulációt használó bizonyítást ismertetünk. Figyeljük meg, hogy a 24.1. tétel bizonyítása során a FischerKK haladását annak alapján láttuk be, hogy a végrehajtás idővel bizonyos „mérföldkövekhez" ér: valamelyik versengő folyamat „megszerzi" a *váltás* változót, és *váltás* értéke „beáll" valamelyik versengő folyamat indexére. Ezeket a mérföldköveket a rájuk vonatkozó időkorlátokkal együtt a B „absztrakt kölcsönös kizárási algoritmusban" egyesítjük. Szimulációt definiálunk FischerKK-ból B-be, s ezzel igazoljuk a FischerKK-ra vonatkozó időkorlátokat.

A B absztrakt algoritmus a következő MMT automatának felel meg.

24.3. automata. B

Lenyomat:

Bemeneti:
- $\texttt{próbál}_i, 1 \leq i \leq n$
- $\texttt{kilép}_i, 1 \leq i \leq n$

Kimeneti:
- $\texttt{belép}_i, 1 \leq i \leq n$
- $\texttt{halad}_i, 1 \leq i \leq n$

Belső:
- `megszerez`
- `stabilizál`

Állapotok:

$státusz \in \{kezdő, megszerzett, stabil\}$, kezdőértéke *kezdő*
$\forall i (\leq i \leq n)$: $szakasz_i \in \{Be, Ki, Ha, Pr\}$, kezdőértéke Ha

Átmenetek

$\texttt{próbál}_i$
- Hatás:
  - $szakasz_i := Pr$

`megszerez`
- Előfeltétel:
  - $\exists i : szakasz_i = Pr$
  - $\forall i : szakasz_i \neq Be$
  - $státusz = kezdő$
- Hatás:
  - $státusz := megszerzett$

`stabilizál`
- Előfeltétel:
  - $státusz = megszerzett$
- Hatás:
  - $státusz := stabil$

$\texttt{belép}_i$
- Előfeltétel:
  - $szakasz_i = Pr$
  - $státusz = stabil$
- Hatás:
  - $szakasz_i := Be$
  - $státusz := kezdő$

$\texttt{kilép}_i$
- Hatás:
  - $szakasz_i := Ki$

$\texttt{halad}_i$
- Előfeltétel:
  - $szakasz_i = Ki$
- Hatás:
  - $szakasz_i := Ha$

A taszkok és korlátaik:
$megszerez = \{\texttt{megszerez}\}$, a korlátok $[0, a_2 + 3\ell_2]$
$stabilizál = \{\texttt{stabilizál}\}$, a korlátok $[0, \ell_2]$
$kritikus = \{\texttt{belép}_i : 1 \le i \le n\}$, a korlátok $[0, a_2 + \ell_2]$
továbbá minden $1 \le i \le n$-re
$halad_i = \{\texttt{halad}_i\}$,$1 \le i \le n$, a korlátok $[0, 2\ell_2]$

- -

A B algoritmus nagyon absztrakt: csak a jólformáltsági és a kölcsönös kizárási feltételeket, továbbá a Pr szakasz globális mérföldköveit (időkorlátaikkal együtt) és az egyes folyamatokra a kilép szakaszban érvényes időkorlátokat tartalmazza. Mivel a mérföldkövek korlátainak összege a Pr szakaszra bizonyítani kívánt $[2a_2 + 5\ell_2]$ korlátot adja, könnyen belátható, hogy B valóban a kölcsönös kizárási feladat megoldása, valamint érvényesek rá az előírt időkorlátok. A továbbiakban megadunk egy f időzített szimulációt a FischerKK rendszerből (algoritmus plusz felhasználók) a B rendszerbe (ugyanazokkal a felhasználókkal). Mivel f időzített szimuláció, a 23.10. tétel alapján ebből következően a FischerKK algoritmus is a megadott korlátokon belül marad.

Definiáljuk az $(s, u) \in f$ relációt a következő módon (feltesszük, hogy a kvalifikáció nélkül használt folyamatindexekre univerzális kvantorok vonatkoznak).

1. $s.most = u.most$
2. s-ben és u-ban ugyanazok a felhasználói állapotok szerepelnek.
3. $u.szakasz_i = \begin{cases} Be & \text{ha } s.elő\_felt_i = halad, \\ Ha & \text{ha } s.elő\_felt_i \in \{vizsgál, beállít, ellenőriz, elhagy\_próbál\}, \\ Be & \text{ha } s.elő\_felt_i = kritikus, \\ Ki & \text{ha } s.elő\_felt_i \in \{visszaállít, elhagy\_kilép\}. \end{cases}$
4. $u.státusz = \begin{cases} kezdő, & \text{ha } s.váltás = 0 \text{ vagy } \exists i : s.elő\_felt_i \in \{kritikus, visszaállít\}, \\ megszerzett, & \text{ha } s.váltás \ne 0,\ \nexists i : s.elő\_felt_i \in \{kritikus, visszaállít\} \\ & \quad \text{és } \exists i : s.elő\_felt_i = beállít, \\ stabil, & \text{ha } s.váltás \ne 0 \text{ és } \nexists i : s.elő\_felt_i \in \{kritikus, visszaállít, beállít\}. \end{cases}$
5. Ha $s.elő\_felt_i = visszaállít$, akkor $u.utolsó(megszerez) \ge s.utolsó(fő_i) + a_2 + 2\ell_2$.
6. Ha $s.váltás = 0$, akkor $u.utolsó(megszerez) \ge \min_i\{g(i)\}$, ahol
$g(i) = \begin{cases} s.utolsó(ellenőriz_i) + 2\ell_2 & \text{ha } s.elő\_felt_i = ellenőriz, \\ s.utolsó(fő_i) + \ell_2 & \text{ha } s.elő\_felt_i = vizsgál, \\ s.utolsó(fő_i) & \text{ha } s.elő\_felt_i = beállít. \\ \infty & \text{egyébként.} \end{cases}$
7. Ha $s.elő\_felt_i = beállít$, akkor $u.utolsó(stabil) \ge s.utolsó(fő_i)$.
8. $u.utolsó(kritikus) \ge \begin{cases} s.utolsó(ellenőriz_i) + \ell_2 & \text{ha } s.elő\_felt_i = ellenőriz \text{ és } s.váltás = i, \\ s.utolsó(fő_i) & \text{ha } s.elő\_felt_i = elhagy\_próbál. \end{cases}$
9. $u.utolsó(halad_i) \ge \begin{cases} s.utolsó(fő_i) + \ell_2 & \text{ha } s.elő\_felt_i = visszaállít, \\ s.utolsó(fő_i) & \text{ha } s.elő\_felt_i = elhagy\_kilép. \end{cases}$

A *most* változóra, a felhasználókra és a szakaszokra vonatkozó összefüggések nyilvánvalóak. A *státusz* változóra vonatkozó megfeleltetés a FischerKK algoritmus során előforduló versengés természetes definícióját adja: ha *váltás* $= 0$ vagy valamelyik folyamat éppen a kritikus szakaszban, illetve közvetlenül utána van, akkor a verseny állapota *kezdő*. Ha *váltás* értéke valamelyik versengő folyamat indexével egyezik meg (vagyis nem-nulla, és különbözik a krikus szakaszban vagy közvetlenül utána levő folyamat indexétől), továbbá ha valamelyik folyamat még módosítani tudja *váltás*-t, akkor a verseny állapota *megszerzett*. Ha *váltás* értéke valamelyik versengő folyamat indexe, és már egyik folyamat sem tudja módosítani *váltás*-t, akkor a verseny állapota *stabil*.

Az *utolsó*(*megszerez*)-re vonatkozó első egyenlőtlenség szerint ha valamelyik folyamat éppen *visszaállít*-ra készül, akkor a *visszaállít* előfordulása után legfeljebb $a_2 + 2\ell_2$ idő múlva lesz *váltás* értéke *megszerzett*. A *megszerzett*-re vonatkozó második egyenlőtlenség jelentése: ha *váltás* $= 0$ (amiből következik, hogy egyetlen folyamat sincs sem a *kritikus*, sem a *visszaállít* állapotban), akkor a *váltás* megszerzéséhez szükséges időt azon lehetséges idők minimumaként kapjuk, amennyire a *váltás* beállítására szóbajöhető folyamatoknak szüksége lenne. Ha például *elő_felt*$_i$ = *beállít* – vagyis a P_i folyamat *váltás* beállítására készül – akkor a megfelelő idő az a legkésőbbi időpont lesz, amikor a legközelebbi lépést végrehajthatja. Viszont ha *elő_felt*$_i$ = *vizsgál* – vagyis a P_i folyamat a változót vizsgálja – a teszt végrehajtásáig eltelt időt plusz ℓ_2-t kell vennünk. A további egyenlőtlenségek hasonlóan értelmezhetők.

Ezek után könnyen belátható, hogy f valóban időzített szimuláció. A bizonyítás a 23.3.3. és a 23.3.4. példákban látható stílusban adható meg. Eközben a 24.2.3. és a 24.2.4. állításokat kell fölhasználni; az elsőt a *beállít*, *kritikus* és a *visszaállít* kapcsán, a másodikat az idő múlásával kapcsolatos lépéseknél. A szimuláció során a FischerKK rendszer minden külső lépése a B rendszer megfelelő külső lépését szimulálja. A *beállít* lépés, amely *váltás* értékét valamelyik folyamat indexére állítja be, a *megszerez*-t szimulálja. A *stabilizál* szimulációja az olyan *beállít*, amely nem engedi meg több *beállít* végrehajtását egyetlen folyamat számára sem. A mindkét feltételt kielégítő *beállít* lépés mind *megszerez*-t, mind *stabilize*-t szimulálja (ebben a sorrendben). Az összes további lépés műveletek nélküli, triviális időzített végrehajtás-szeleteket szimulál. A részleteket meghagyjuk gyakorlatnak (lásd a 24-5. gyakorlatot).

A 23.10. tételből következik, hogy a FischerKK rendszer megengedett időzítésű nyomai részhalmazát alkotják a B rendszer megengedett időzítésű nyomainak. A szükséges időkorlátok innen adódnak. □

Módosítjuk a FischerKK algoritmust a kód ismertetése előtt leírt módon $k-1$ explicit `késleltet` lépés beiktatásával (itt $k\ell_1 > \ell_2$), hogy megfeleljen az általunk használt modellnek! Ekkor a próbál szakaszra vonatkozó időkorlát $2k\ell_2 + 5\ell_2$ lesz. A lehető legkisebb k-t választva (ez a $k = \lfloor L \rfloor + 1$ érték, ahol $L = \ell_2/\ell_1$), a $2L\ell_2 + \mathcal{O}(\ell_2)$ időkorlátot kapjuk.

Nyújtás.. Az időzítési bizonytalanság miatt „megnyúlhat” az algoritmusok időkorlátja, mint például az előbb kapott $2L\ell_2 + \mathcal{O}(\ell_2)$ korlát. Ha $L = 1$, azaz $\ell_1 = \ell_2$, a rendszerben nincs időzítési bizonytalanság. Ekkor az időkorlát mind-

össze $\mathcal{O}(\ell_2)$, s csupán a folyamatok egymás utáni lépései között eltelt valós időre vonatkozó ℓ_2 felső korláttól függ. Viszont ha L értéke nem 1, az időkorlát ennek megfelelően megnő. Ténylegesen a korlátban szereplő ℓ_2 valós időt *szorozni* kell az L időzítési bizonytalansággal.

Az $L\ell_2$ tag a következő okokból jelenik meg a FischerKK algoritmus kapcsán. Minden folyamat számlálja saját lépéseit; csak így lehet biztos benne, hogy adott hosszúságú (valós) idő, mondjuk t, már eltelt. Elég sok lépést kell leszámlálni, hogy még ha az egyes lépések a lehető legkevesebb időt (ℓ_1) vették is igénybe, akkor is biztosan elteljen legalább t valós idő. Ehhez legalább t/ℓ_1 lépés kell. De az egyes lépések elhúzódhatnak akár ℓ_2 ideig is, amiből az összes szükséges időre a $(t/\ell_1)\ell_2 = Lt$ alsó korlátot kapjuk.

Ez durván azt jelenti, hogy az L időzítési bizonytalanságú rendszer folyamatainak Lt időre van szüksége ahhoz, hogy biztosak legyenek t valós idő elteltében. Ebben az értelemben az időbonyolultság az időzítési bizonytalanságnak megfelelő tényezővel „megnyúlt". A „megnyúlási hatás" előfordult már az időtúllépésnél a 23.1.4. példában is. Ott az időtúllépés helyes működéséhez a $k\ell_1 > \ell + d$ egyenlőtlenségre volt szükség; a túllépést kezelő folyamat lényegében azt ellenőrzi, hogy $\ell + d$-nél hosszabb (valós) idő telt-e már el. De ez esetben még a $k\ell_2 + \ell > L(\ell + d) + \ell$ időpontig is elhúzódhat az időtúllépés bekövetkezése.

Az időkorlátok és a pártatlanság elegyítése.. Tekinthetjük a FischerKK algoritmus azon változatát is, melyben csak a lehetséges `ellenőriz` műveletek tényleges megtörténtéig eltelt időre adunk meg alsó, és a `beállít` művelet végrehajtási idejére felső korlátot. Az `ellenőriz`-től különböző bármely más helyileg vezérelt lehetséges művelettől csak annyit kívánunk meg, hogy *valamikorra* végrehajtódjék. Nem nehéz belátni, hogy még ez a változat is megoldja a kölcsönös kizárási feladatott, bár nem reprezentálható a könyvben leírt MMT modellel. Olyan modellt kívánna, amely megengedi bizonyos taszkok számára időkorlátok, a többiek számára meg pártatlansági feltételek előírását. Időkorlátok természetesen nem bizonyíthatók az algoritmus ezen változatára.

24.3.. Időzítési hibákkal szembeni ellenállóképesség

A FischerKK algoritmus helyessége kritikus módon függ az időzítési megszorításoktól. Még legfontosabb helyességi feltétele, a kölcsönös kizárás érvényessége is meghiúsulhat olyan időzített végrehajtások során, melyek megsértik a lényeges időzítési korlátokat: az `ellenőriz` lépések a_1 alsó, vagy a `beállít` lépések ℓ_2 felső korlátját. Jó lenne az algoritmust úgy tökéletesíteni, hogy legalább a kölcsönös kizárási feltétel teljesüljön, bármi történik is az időzítéssel. Általános tervezési kritérium lehet, hogy az időzítés alapú algoritmusok leglényegesebb biztonsági tulajdonságait kívánatos az időzítési változatoktól függetlenül garantálni.

A FischerKK algoritmus ilyen irányú javításának egyik módja lehet, hogy a kritikus szakaszát egy másik S algoritmus próbál, kritikus és kilép szakaszaival helyettesítjük. Az S algoritmusnak – tekintet nélkül lépéseinek időzítésére – saját kritikus szakaszára vonatkozóan mindig garantálnia kell a kölcsönös kizárási feltételt.Viszont az időzítési megszorítások teljesülése esetében S nem is

akadályozhatja a FischerKK algoritmus haladását. S lehet a kölcsönös kizárási feladatot megoldó (tehát a jólformáltságot, a kölcsönös kizárást és a haladási feltételt teljesítő) bármelyik aszinkron algoritmus, de sajnos a 10.33. tétel miatt minden ilyen algoritmushoz legalább n közös regiszter szükséges. Szerencsére S-re nem szükséges ilyen erős haladási feltételeket kikötni, ehelyett használhatjuk a következő gyengébb haladási feltételt.

1-**párhuzamos haladás.** Ha egy *megengedett időzítésű végrehajtás* során bármikor legfeljebb egy felhasználó van *Ha*-n kívül, akkor

1. (1-párhuzamos haladás a próbál szakaszban) ha U_i *Pr*-ben van, akkor később valamikor belép *Be*-be;
2. (1-párhuzamos haladás a kilép szakaszban) ha U_i *Ki*-ben van, akkor később valamikor belép *Ha*-ba.

Itt természetesen az S algoritmus felhasználóiról és szakaszairól van szó.

A továbbiakban megadunk egy lehetséges, a kívánt feltételeket teljesítő S aszinkron algoritmust. Figyeljük meg, hogy az algoritmus csak két közös regisztert használ.

24.4. algoritmus. *S*

Osztott változók:
az x folyamatindex, melyet bármely folyamat írhat és olvashat,
kezdőértéke tetszőleges
$y \in \{0,1\}$, melyet bármely folyamat írhat és olvashat,
kezdőértéke 0

A P_i folyamat:

** a maradék szakasz **	** a kritikus szakasz **
`próbál`$_i$	`kilép`$_i$
M: $x := i$	$y := 0$
if $y \neq 0$ **then** goto M	`halad`$_i$
$y := 1$	
if $x \neq i$ **then** goto M	
`belép`$_i$	

24.3. tétel. . *Az S aszinkron közös memóriájú algoritmus biztosítja a jólformáltsági, a kölcsönös kizárási és az* 1*-párhuzamos haladási feltétel teljesülését.*

Bizonyítás. A bizonyítás hasonlít a könyv kölcsönös kizárási algoritmusokkal kapcsolatos számos bizonyításához, ezért meghagyjuk gyakorlatnak (lásd a 24-11. gyakorlatot). □

Az S és a FischerKK algoritmus együttese a következő kóddal adható meg.

24.5. algoritmus. FischerS

Közös változók:
váltás $\in \{0, 1, \ldots, n\}$, melyet bármely folyamat írhat és olvashat,
kezdőértéke 0
az x folyamatindex, melyet bármely folyamat írhat és olvashat,
kezdőértéke tetszőleges
$y \in \{0, 1\}$, melyet bármely folyamat írhat és olvashat,
kezdőértéke 0

A P_i folyamat:

** Haladási szakasz **	** a kritikus szakasz **
`próbál`$_i$	`kilép`$_i$
L: **if** *váltás* $\neq 0$ **then goto** L	$y := 0$
váltás $:= i$	
if *váltás* $\neq i$ **then goto** L	*váltás* $:= 0$
M: $x := i$	`halad`$_i$
if $y \neq 0$ **then goto** M	
$y := 1$	
if $x \neq i$ **then goto** M	
`belép`$_i$	

A FischerS kódja felfogható akár aszinkron, akár részben szinkron algoritmus leírásának. Ha aszinkron algoritmusnak tekintjük, akkor minden folyamatra érvényes pártatlansági feltételeket tételezünk fel.

24.4. tétel. . *A* FischerS*, mint aszinkron algoritmus biztosítja a jólformáltsági és a kölcsönös kizárási feltétel teljesülését.*

Bizonyítás. A jólformáltság bizonyítása könnyű. A kölcsönös kizárás következik a 24.3. tételben kimondott állításból, nevezetesen abból, hogy S garantálja a kölcsönös kizárást. □

A FischerS haladási tulajdonságainak meghatározását a gyakorlatokra hagyjuk (lásd a 24-13. és 24-14. gyakorlatokat).

Ha viszont FischerS kódját részben szinkron algoritmus leírásának tekintjük, akkor a FischerKK-hoz hasonlóan azt tételezzük fel, hogy minden P_i folyamathoz két taszk tartozik, az egyik $[a_1, a_2]$, a másik $[\ell_1, \ell_2]$ korlátokkal, ahol $\ell_2 < a_1$. Az első taszk csak azt a lépést tartalmazza, melyben P_i ellenőrzi *váltás* értékét, minden más tevékenység a második taszkhoz tartozik.

24.5. tétel. . *A részben szinkron algoritmusként felfogott* FischerS *megoldja a kölcsönös kizárási feladatot, tehát biztosítja a jólformáltságot, a kölcsönös kizárást és a haladást.*

Bizonyítás. A jólformáltság és a kölcsönös kizárás következik a 24.4. tételből. A kilép szakaszra vonatkozó haladási feltétel könnyen belátható. Mutassuk meg a

próbál szakaszra vonatkozó haladási feltétel teljesülését! Az alábbi meggondolásokban *Ha*, *Pr*, *Be* és *Ki* a FischerS algoritmus szakaszait jelölik. A FischerKK *próbál szakaszán* *Pr*-nek az M cimke előtti részét értjük, *Pr* további részét az S *próbál szakaszának* hívjuk. Hasonlóan értelmezzük az S *kilép szakaszát*, nevezetesen *Ki*-nek az $y := 0$ értékadás előtti részét, és a FischerKK *kilép szakaszát*, vagyis *Ki* maradék részét. Definiáljuk a FischerKK *kritikus szakaszát*, mint az S próbál szakaszának, *Be*-nek és az S kilép szakaszának kombinációját.

Tegyük fel, hogy valamely megengedett végrehajtás során van legalább egy felhasználó *Pr*-ben, és senki sincsen *Be*-ben. Ha egy későbbi időpontban valamelyik folyamat az S próbál szakaszában van, akkor (felhasználva, hogy a FischerKK garantálja a kölcsönös kizárást) az S-re kirótt 1-párhuzamos haladási feltételből következik a bizonyitandó állítás, vagyis hogy előbb-utóbb valamelyik folyamat belép *Be*-be.

Másfelől ha azt tételezzük fel, hogy később már egyetlen folyamat sem jut el S próbál szakaszába, akkor S 1-párhuzamos haladási feltétele miatt előbb-utóbb kiürül S kilép szakasza. Ez azt jelenti, hogy a FischerKK kritikus szakasza is üres, ekkor viszont a FischerKK haladási feltételéből következik, hogy előbb-utóbb belép valamelyik folyamat a FischerKK kritikus szakaszába. Tehát a folyamat az S próbál szakaszába is bekerült, ami ellentmondás. □

24.4.. Megoldhatatlansági eredmények

A fejezetet két megoldhatatlansági eredménnyel zárjuk. Az első a részben szinkron modellen belül a kölcsönös kizárási feladat megoldására vonatkozó alsó időkorlát. A második megoldhatatlansági eredmény a *végső időkorlátos* esetre vonatkozik, amikor az időkorlátok érvényességét csak „valamikortól” követeljük meg.

24.4.1.. Alsó időkorlát

A FischerKK algoritmus a részben szinkron modell keretein belül megoldja a kölcsönös kizárási feladatot. A próbál szakaszban való haladás legrosszabb esetében a időbonyolutsága $2L\ell_2 + \mathcal{O}(\ell_2)$. Az eredmény még javítható; belátható az $L\ell_2 + \mathcal{O}(\ell_2)$ korlát is (lásd a 24-8. gyakorlatot). De elérhetünk-e ennél többet? Vagyis létezik-e a modell keretein belül olyan gyorsabb algoritmus, amely még mindig csak konstans számú változót használ? Az egyváltozós esetre ismertetünk egy egyszerű eredményt, amely szorosan kapcsolódik a 10.34. tételhez.

24.6. tétel. . *A részben szinkron olvasható/írható közös memóriájú modell keretein belül nem létezik a kölcsönös kizárási feladatot két folyamatra megoldó, egyetlen olvasható/írható közös változót használó, a próbál szakasz haladási idejére vonatkozóan $L\ell_2$ felső korláttal bíró algoritmus.*

A 24.6. tétel bizonyításában olyan érdekes, az időzített végrehajtási sorozatokat „megnyújtó” és „összehúzó” technikát használunk, amely továbbra is tiszteletben tartja az időzítési kikötéseket. A bizonyítás nagyjából a 10.34. tétel igazolásán alapul.

Bizonyítás. Indirekt feltételként fogadjuk el, hogy létezik ilyen, egyetlen x közös regisztert használó A algoritmus. Megkonstruáljuk A-nak egy olyan időzített végrehajtási sorozatát, amely megsérti a kölcsönös kizárást.

Tekintsük az A egy olyan megengedett időzítésű α_1 végrehajtását, amely során csak a P_1 folyamat fut, mégpedig a lehető leglassabban, vagyis az egymás utáni lépések közt ℓ_2 idő telik el. Az időkorlátra vonatkozó feltétel miatt P_1-nek α_1-ben $L\ell_2$ idő alatt el kell érnie Be-t. A 10.8. alfejezet okfejtéseihez hasonlóan P_1-nek a Be-be való belépés előtt írnia kell a közös változóba. Legyen α_2 az α_1 azon kezdőszelete, amely pontosan azelőtt ér véget, mielőtt P_1 először ír x-be.

Hasonlóképpen vegyünk egy olyan megengedett időzítésű lassú α_3 végrehajtást, amely az α_1-gyel megegyező kezdőállapotból indul, csak a P_2 folyamat fut, és P_2 $L\ell_2$ idő alatt jut el Be-be. Legyen α_4 az α_3-nak a Be-be való belépéssel végződő kezdőszelete. Legyen α_5 az A olyan alternatív véges időzített végrehajtása, amely pont olyan, mint α_4, csak minden fel van gyorsítva („össze van nyomva") az $L = \ell_2/\ell_1$ tényezővel. Így α_5-ben a P_2 folyamat ℓ_2 idő múlva lép be Be-be.

Az ellenpéldát szolgáltató α időzített végrehajtás α_2-vel kezdődik, s így a P_1 folyamatot eljuttatja az x-be való írásig. Ezen a ponton várakozást engedélyezünk P_1-nek. Most a P_2 folyamat lépéseinek végrehajtását engedélyezzük az α_5 gyors időzített végrehajtásnak megfelelően. (Mivel α_2 során a P_1 folyamat nem ír x-be, a P_2 folyamat nem tudja, hogy P_1 aktív, s emiatt úgy hajtódhat végre, mintha egyedül lenne.) Tehát ℓ_2 idővel működésének megkezdése után a P_2 eléri Be-t. A P_1-nek pontosan ℓ_2 ideig tartó várakozást engedélyezünk, ami elegendő ahhoz, hogy P_2 elérje Be-t. Ezután megengedjük, hogy P_1 α_1-nek megfelelően folytassa futását. P_1 legelőször x-be ír, felülírva bármit, amit a P_2 folyamat a Be-hez vezető út során oda írhatott. Ezzel eltünteti a P_2 folyamat futásának bármely jelét, s lehetővé válik, hogy P_1 pontosan az α_1-nek megfelelően fusson le, tehát végül Be-be érjen. Így viszont két folyamat kerül egyidejűleg Be-be, ami ellentmond a kölcsönös kizárási követelménynek. □

A 24.6. tétel alsó korlátja kiterjeszthető több közös változó esetére is, de az erről szóló, pillanatnyilag ismert eredmények nem túl élesek. A 10.8. alfejezet módszerei bizonyos részeredményeket szolgáltatnak.

24.4.2.. Végső időkorlátos megoldhatatlansági eredmények*

Ha részben szinkron módon futtatjuk a FischerS algoritmust, akkor megoldja a kölcsönös kizárási feladatot (beleértve a haladást is), s aszinkron futtatása is biztosítja legalább a kölcsönös kizárási tulajdonságot. Lehetséges-e a haladás garantálása gyengébb feltételek mellett, például ha az algoritmus valameddig aszinkron módon fut, de *valahonnan kezdve* kielégíti az időkorlátokat? Nem nehéz belátni, hogy a FischerS algoritmus nem nyújt ilyen garanciát. A bizonyítást meghagyjuk gyakoratnak hagyjuk (lásd a 24-12. gyakorlatot). Megmutatjuk, hogy valójában *egyetlen algoritmus* sem képes erre.

24.7. tétel. . *Nem létezik olyan aszinkron A algoritmus, amely $n \geq 2$ folyamat esetében teljesíti az alábbiakat:*

1. *aszinkron futtatva biztosítja a jólformáltságot és a kölcsönös kizárást;*

2. *biztosítja a haladást, ha úgy futtatjuk, hogy minden folyamat lépéskorlátjai végül az* $[\ell_1, \ell_2]$ *intervallumba esnek*[2]*;*
3. *n-nél kevesebb közös olvasható/írható regisztert használ.*

Bizonyításvázlat. A bizonyítás szorosan követi a 10.33. tételét. A fő lemma analóg a 10.37. lemmával, s olyan k-elérhető rendszerállapot létezését biztosítja, amelyben k különböző változót k folyamat „fed le". A lemmában nem szerepelnek időzítési megszorítások. A fő lemma indukcióval bizonyítható. A 10.37. lemma bizonyításával megegyező konstrukciót használhatunk. Az egyetlen különbség, hogy ahol a korábbi bizonyítás az általános haladási feltételt használta, ott most a gyengébb „végső időkorlátos" haladási feltétellel kell boldogulnunk. Valahányszor arra akarjuk kényszeríteni a folyamatokat, hogy haladjanak, egyszerűen úgy kezdjük futtatni őket, hogy attól kezdve teljesüljenek az időzítésre vonatkozó korlátozásaik.

A konstrukciónak van egy kicsit trükkös aspektusa: amikor a $(k+1)$-edik folyamat számításait „betápláljuk" a $P_1, P_2, \ldots, P_k$ folyamatokat tartalmazó fő számítási folyamatba, P_{k+1} számításait „össze kell nyomni", hogy beférjenek, mielőtt a többi folyamat megteszi következő lépését, azután meg P_{k+1}-nek olyan hosszú várakozást kell engedélyezni, amely lehetővé teszi a többi folyamat számításainak befejezését. Az időzítési módosítások az időzítési feltételek megsértését eredményezhetik. Ez nem okoz gondot, mivel a segédtétel nem kívánja a megkonstruált végrehajtástól, hogy bármilyen időzitési korlátozásoknak megfeleljen. □

24.5.. Megjegyzések a fejezethez

A FischerKK algoritmus [116] Fischer nevéhez fűződik. Az algoritmust mostanában szokás az időzítés alapú rendszerekkel kapcsolatos formális módszerek képességeinek demonstrálára szolgáló próbakőként használni. A FischerKK algoritmusra a kölcsönös kizárási tulajdonság teljesülését Abadi és Lamport [1], továbbá Luchango [201] bizonyításai alapján mutattuk meg. A FischerKK algoritmus időkorlátjára vonatkozó bizonyítás Luchango és Lynch cikkeiből [201,204,205] származik. A [201]-ben megtalálható egy jobb időkorlát bizonyítása is. A Larch [202] tételbizonyító program segítségével számítógéppel ellenőrizték a FischerKK algoritmussal kapcsolatos összes bizonyítást. A DijkstraKK algoritmus időkorlátjának vázlatos bizonyítása megtalálható [204]-ben.

A FischerS algoritmus (lásd [209]), valamint a 24.4. alfejezet megoldhatatlansági eredményei Lynch-től és Shavit-tól származnak. Alur és Taubenfeld [10] kedvező időbonyolultságú részben szinkron kölcsönös kizárási algoritmusokat fedezett fel korlátos számú konkurens kérés esetére. Modelljük és a használt mérték némileg eltér az itt tárgyalttól. A kölcsönös kizárás időbonyolultságára vonatkozó alsó és felső korlátokat tartalmaz Attiya és Lynch [25] cikke részben szinkron hálózatok esetére. Az általuk vizsgált feladat abban különbözik az ittenitől, hogy náluk a rendszer nem kap explicit értesítést a kritikus szakaszok lezárásáról.

[2] Formálisan bevezethetnénk az MMT automaták *végső időzítésű* végrehajtásait, és a feltételt kimondhatnánk végső időzítésű végrehajtásokkal. Lemondunk a formális tárgyalásról.

24.6.. Gyakorlatok

24-1. Bizonyítsuk be a 24.2.4. állítást.

24-2. Egészítsük ki a 24.1. tétel bizonyításában a haladás működésen alapuló igazolásának hiányzó részleteit.

24-3. Mutassuk meg, hogy a FischerKK algoritmus megengedi folyamatok kizárását.

24-4. Kielégíti-e a HibásFischerKK algoritmus a haladási feltételt? Bizonyítsuk be, vagy adjunk meg ellenpéldát.

24-5. Egészítsük ki a 24.2. tétel szimulációs bizonyításának hiányzó részleteit.

24-6. Bizonyítsuk be a FischerKK algoritmus $2a_2 + 5\ell_2 - a_1$ javított időkorlátját.

24-7. Adjunk meg a FischerKK algoritmushoz egy olyan időzített végrehajtási sorozatot, amely során a lehető legtöbb idő telik el attól kezdve, míg valamely folyamat Be-ben van addig, míg egy bizonyos folyamat Ha-ban lesz. Honnan származik az a_2 előtti 2 együttható?

24-8. Tervezzünk meg a részben szinkron közös memóriájú modell keretén belül olyan alternatív kölcsönös kizárási algoritmust, amely csak egy olvasható/írható közös változót használ és $L\ell_2 + \mathcal{O}(\ell_2)$ alakú időkorláttal bír (tehát nincs az $L\ell_2$ tag előtt a 2 szorzó).

24-9. Legyen P bemeneti művelet nélküli, egyetlen a kimeneti művelettel ellátott MMT automata. Tegyük fel, hogy P-nek csak egyetlen taszkja van a hozzá rendelt $[\ell_1, \ell_2]$ korlátokkal, melyekre $0 < \ell_1 \leq \ell_2 < \infty$ teljesül, továbbá legyen ez a taszk mindig lehetséges. Tegyük fel, hogy P minden megengedett időzített végrehajtási sorozatban d-nél több vagy vele egyenlő valós idő alatt egyetlen a műveletet hajt végre. Bizonyítsuk be, hogy van P-nek olyan időzített végrehajtási sorozata, amely során az a kimenti művelet legalább Ld valós idő eltelte után hajtódik végre ($L = \ell_2/\ell_1$).

24-10. Vizsgáljuk meg újra a 10.3. alfejezet DijkstraKK algoritmusát. Bizonyítsuk be a $(3n + 11)\ell$ időkorlátot az ahhoz szükséges időre vonatkozóan, míg valamelyik folyamat a próbál szakaszban van attól az időponttól, míg egy folyamat a kritikus szakaszba lép (feltesszük, hogy ℓ a folyamatok lépésidejének felső korlátja). Úgy járjunk el, hogy az algoritmust tekintsük MMT automatának. Használjunk a 24.2. tétel bizonyításában megadotthoz hasonló időzített szimulációt.

24-11. Bizonyítsuk be a 24.3. tételt. (*Útmutatás.* Legyen I_1 azon P_i folyamatok halmaza, amelyekre $x = i$ és P_i y-t készül beállítani. Jelölje I_2 azon P_i folyamatok halmazát, amelyekre $x = i$ és P_i x-et készül megvizsgálni. Végül tartozzanak az I_3-hoz azok a folyamatok, amelyek éppen Be-ben, vagy közvetlenül előtte, illetve utána vannak. Hasznosak lehetnek a következő invariánsok:

(a) $|I_1| \cup |I_2| \cup |I_3| \leq 1$;

(b) ha $|I_2 \cup I_3| > 0$, akkor $y = 1$;

(c) ha minden folyamat Ha-ban van, akkor $y = 0$.)

24-12. Mutassuk meg, hogy az S algoritmus (konkurens igények esetében) nem garantálja a haladást. Ehhez adjunk meg egy olyan konkrét végrehajtási sorozatot, amely nem teljesíti a haladási feltételt.

24-13. Az aszinkron algoritmusnak tekintett FischerS algoritmus teljesíti-e az 1-párhuzamos-haladás feltételét? Ha igen, bizonyítsuk be, ha nem, adjunk meg ellenpéldát.

24-14. Adjuk meg az aszinkron algoritmusnak tekintett FischerS algoritmus egy olyan konkrét végrehajtási sorozatát, amelyre nem teljesül a haladási feltétel.

24-15. Adjunk meg egy másik olyan algoritmust, amely rendelkezik a FischerS-től megkövetelt összes helyességi tulajdonsággal (vagyis aszinkron módon futtatva garantálja a jólformáltságot és a kölcsönös kizárást, részben szinkron módon futtatva pedig a haladást), de három helyett csak *két* közös olvasható/írható változót használ.

24-16. Bizonyítsuk be, hogy nem létezik olyan algoritmus, amely rendelkezik a FischerS-től megkövetelt összes helyességi tulajdonsággal, de három helyett csak *egy* közös olvasható/írható változót használ.

24-17. *Kutatási téma.* Tekintsük a 10-32. gyakorlatban definiált k-párhuzamos-haladási feltételt. Tervezzünk olyan algoritmust, amely aszinkron módon futtatva kielégíti a jólformáltsági, a kölcsönös kizárási és a k-párhuzamos-haladási feltételt, továbbá kielégíti a haladási feltételt is, ha aszinkron futtatjuk. Próbáljuk meg minimalizálni a közös változók számát.

24-18. *Kutatási téma.* Készítsünk időzítésen alapuló, a kölcsönös kizárási feladatot megoldó (tehát a jólformáltságot, a kölcsönös kizárást és a haladást garantáló) algoritmust. Ezen kívül az algoritmusnak teljesítenie kell az alábbi időkorlátokkal megadott összes feltételt.

(a) Ha valamelyik felhasználó Be-ben van, akkor a legrosszabb esetben $\mathcal{O}(L\ell_2)$ idő kell ahhoz, hogy valaki Ha-ba kerüljön.

(b) Ha valamelyik P_i felhasználó Be-ben, minden más felhasználó Pr-ben van, akkor a legrosszabb esetben $\mathcal{O}(\ell_2)$ idő kell ahhoz, hogy vagy P_i belépjen Ha-ba, vagy egy másik felhasználó belépjen Be-be.

(c) Ha valamelyik felhasználó Ki-ben van, akkor a legrosszabb esetben $\mathcal{O}(\ell_2)$ idő kell ahhoz, hogy eljusson Pr-be.

Kíséreljük meg általánosítani eredményeinket, és készítsünk ugyanezen feladat megoldására egy másik algoritmust, melyben a második követelményt terjesztjük ki: ha egyidejűleg legfeljebb k felhasználó van Pr-en kívül, akkor jó felső korlátot követelünk meg a próbál szakaszban való haladásra (k rögzített érték, $1 \leq k \leq n$).

24-19. Vezessünk le a részben szinkron modellben két közös olvasható/írható változó esetében érvényes alsó időkorlátot a próbál szakaszban való haladásra vonatkozóan. (*Útmutatás.* Vizsgáljuk meg a 10.8. alfejezet bizonyításait. Az alsó korlát $cL\ell_2$ alakú lesz; itt c kis konstans.)

24-20. *Kutatási téma.* Határozzunk meg tetszőleges k-ra ($1 \leq k \leq n$) a próbál szakaszban való haladás legrosszabb esetében érvényes szoros alsó és felső korlátokat a k osztott változós részben szinkron olvasható/írható közös memóriás modell kölcsönös kizárási algoritmusaira.

24-21. Adjunk meg azt bizonyító végrehajtást, hogy a FischerS algoritmus nem elégíti ki a 24.7. tétel kimondásakor felsorolt feltételeket.

24-22. Készítsük el a 24.7. tétel részletesebb bizonyítását.

24-23. Vizsgáljuk a kölcsönös kizárási feladat megoldhatóságát az *ismeretlen időkorlátos* modell esetében. Ilyenkor feltételezzük a folyamatok lépésidejére vonatkozó ℓ_1 alsó és ℓ_2 felső korlátok létezését, de ezeket a korlátokat „nem ismerik" a folyamatok. (Vagyis különböző végrehajtásokhoz eltérő korlátok tartozhatnak, bár minden végrehajtás során állandóak a korlátok.)

Bizonyítsuk be a 24.7. tétel ismeretlen időkorlátos modellben érvényes megfelelőjét.

24-24. *Kutatási téma.* Dolgozzuk ki a részben szinkron algoritmusok általánosabb erőforrás-hozzárendelési feladatainak elméletét.

25. fejezet

Megegyezés részleges szinkronizációval

Ebben a zárófejezetben immár negyedszer térünk vissza a megegyezési problémára, ezúttal részben szinkron hálózatokat feltételezve. Csak a megállási hibákat vizsgáljuk. Látni fogjuk, hogy a megegyezési problémára vonatkozó eredmények a részben szinkron esetben teljesen eltérnek mind a szinkron, mind az aszinkron esettől. Először egy alapalgoritmust és egy alsó korlátot ismertetünk. Mindkettőt a szinkron elrendezésre vonatkozó megfelelő eredményekből vezetjük le. A két eredmény időbonyolultsága között az időzítési bizonytalanságon alapuló hézag van. Ezután megadunk egy olyan bonyolultabb algoritmust és egy nehezebben elérhető eredményt az alsó korlátról, melyek jórészt kitöltik ezt a hézagot. A gyengébb időzítési modellekre vonatkozó eredményekkel és a jövőbeli lehetséges kutatásra való kitekintéssel zárjuk a fejezetet.

25.1.. A feladat

A **megegyezési feladatot** a 12.1 és a 21.2. alfejezetekhez nagyon hasonlóan értelmezzük. Nevezetesen az A rendszer külső felülete a $\texttt{kezd}(v)_i$ bemeneti és a $\texttt{választ}(v)_i$ kimeneti műveletekből áll (itt $1 \leq i \leq n$, $v \in V$), továbbá tartalmazza még a $\texttt{megállít}_i$ kimeneti műveleteket. Az U_i felhasználóknak $\texttt{kezd}(v)_i$ kimenetei és $\texttt{választ}(v)_i$ bemenetei vannak minden $v \in V$-re. Az U_i MMT automata bármely időzített végrehajtási sorozat során legfeljebb egy $\texttt{kezd}_i$ műveletet hajt végre.

$\texttt{kezd}_i$ és $\texttt{választ}_i$ műveletek egy sorozata i-re vonatkozóan *jólformált*, ha kezdőszelete valamely $\texttt{kezd}(v)_i, \texttt{választ}(w)_i$ alakú sorozatnak. Az A-ból és az U_i felhasználókból összerakott rendszerben a következő feltételek teljesülését vizsgáljuk.

Jólformáltság. Az egyesített rendszer bármely időzített végrehajtási sorozatában minden i-re igaz, hogy az i-dik kaput tekintve az U_i és az A közti interakciók jólformáltak i-re vonatkozóan.

Megegyezés. Bármely időzített végrehajtási sorozat során az összes döntési érték azonos.

Érvényesség. Ha egy időzített végrehajtási sorozat során az összes előforduló `kezd` művelet ugyanazt a v értéket tartalmazza, akkor ez a v az egyetlen lehetséges döntési érték.

Hibamentes befejezés. Minden olyan megengedett időzítésű végrehajtási sorozatában, amikor minden kapun előfordul *kezd* esemény, a *dönt* esemény is bekövetkezik mindenütt.

f-hibás befejezés, ha $0 \leq f \leq n$. Ha egy megengedett időzítésű végrehajtási sorozatában minden kapun előfordul a `kezd` esemény, és legfeljebb f kapun jelentkezik a `megáll` esemény, akkor minden nem hibázó kapun bekövetkezik a *dönt* esemény.

A *várakozásmentes befejezés* az f-hibás befejezés azon speciális esetét jelenti, amikor $f = n$.

Feltételezzük, hogy A a 23.4.2. szakaszban leírt részben szinkron küld/fogad hálózati rendszer. Minden P_i folyamat (véges sok) taszkjára vonatkozó ℓ_1 és ℓ_2 korlátokkal ellátott MMT automata ($0 \leq \ell_1 \leq \ell_2 < \infty$). Legyen $L = \ell_2/\ell_1$. A folyamatok megállási hibáknak vannak kitéve. A csatornákról föltételezzük, hogy a 23.4.2. szakaszban bevezetett második típusnak felelnek meg, vagyis *minden üzenet* továbbítási idejére érvényes d felső korláttal rendelkező megbízható FIFO csatornák.

Akkor mondjuk, hogy A *megoldja a megegyezési problémát*, ha felhasználók tetszőleges együttesére garantálja a jólformáltságot, a megegyezést, az érvényességet és a hibamentes befejezést. Olyan algoritmusokat tekintünk, amelyek különböző f értékekre garantálják az f-hibás befejezést. Azt vizsgáljuk, hogy az összes bemenet megérkezése után mennyi idő kell ahhoz, hogy minden nemhibázó folyamat döntésre jusson. Az időbonyolultságnál az L bizonytalansági paraméter szerepére összpontosítjuk figyelmünket. A fejezet során a feladat speciális esetével foglalkozunk. Ugyanis feltesszük, hogy a hálózat gráfja teljesen összefüggő és $V = \{0, 1\}$. Azt is feltételezzük, ℓ_1 és ℓ_2 sokkal kisebb d-nél, sőt még $n\ell_2$ és $L\ell_2$ is kicsi d-hez képest.

Szükségünk van még egy technikai jellegű kikötésre: a nemhibázó folyamatok taszkjai mindig lehetséges taszkok (bár a taszkok egyetlen lehetséges művelete lehet olyan „üres” tevékenység, amely nem okoz állapotváltozást). Az alsó korlátok bizonyításánál így válik lehetővé, hogy a lépésidőkre vonatkozó egyszerű mintákkal dolgozzunk.

25.2.. Hibajelző

A fejezet algoritmusainak hasznos építőköve az F „tökéletes hibajelző”. A 21.4. alfejezetben az aszinkron esetre definiáltunk hibajelzőket. Mint bizonyára emlékezünk rá, a hibajelző a $\texttt{megállít}_i$ bemeneti és az $\texttt{inf\_megállt}(j)_i$ kimeneti

műveleteket tartalmazza, ahol $j \neq i$. Az `inf_megállt`$(j)_i$ művelet arra szolgál, hogy P_i tudassa a P_j folyamat megállását. A tökéletes hibajelző garantáltan csak a ténylegesen előforduló hibákat jelenti, s azokat előbb-utóbb minden nem-hibázó folyamattal tudatja. A 21.4. alfejezettől való egyetlen eltérés az, hogy a hibajelzőt most nem b/k automatának, hanem általános időzített automatának (GTA) tekintjük.

A fejezetben feltételezett modellnek megfelelő, tökéletes hibajelzőt megvalósító részben szinkron hálózati modellt adunk meg. Az alapötlet hasonlít a 23.1.2 példa időtúllépést kezelő MMT automatájában használthoz. **PSzinkHJ**

algoritmus

Folyamatonként egy taszkot felhasználva minden P_i folyamat állandóan üzeneteket küld az összes többi P_j-nek. Ha valamelyik P_i elegendően nagy m számú lépést hajt végre anélkül, hogy P_j-től üzenetet kapna, akkor feljegyzi, hogy P_j megállt, és $\texttt{inf\_megállt}(j)_i$ kimenetet ad.

Az m lépésszámként az a legkisebb egész vehető, amely $(d+\ell_2)/\ell_1+1$-nél határozottan nagyobb.

25.1. tétel. . PSzinkHJ *tökéletes hibajelző.*

Bizonyítás. Nyivánvaló, hogy minden nemhibázó folyamat előbb-utóbb észrevesz minden hibát. Meg kell mutatnunk, hogy csak a tényleges hibákat észlelik. Tegyük fel tehát, hogy P_i $\texttt{inf\_megállt}(j)_i$-t hajt végre. Így ezt megelőzően P_i $(d+\ell_2)/\ell_1+1$-nél több lépést hajtott végre anélkül, hogy P_j-től üzenetet kapott volna. Ezért $d+\ell_2$-nél határozottan több idő múlt el, mióta P_i egyáltalán nem kapott üzenetet P_j-től. De mivel P_i legfeljebb ℓ_2 időközönként küld üzeneteket P_j-nek, és minden üzenet maximum d idő alatt megérkezik, az egymás utáni `fogad` műveletek közti idő legfeljebb $d+\ell_2$ lehet. Így P_j-nek valóban meg kellett állnia. □

Szükségünk lesz a PSzinkHJ két időzítési tulajdonságára is. Az első azt mondja ki, hogy a hibaértesítés csak a hiba bekövetkezése után legalább d idő elteltével fordulhat elő. A második felső korlát arra az időre, míg hibaértesítések bekövetkezhetnek.

25.2. tétel. . *1. Ha a* PSzinkHJ *valamely időzített végrehajtási sorozatában mind* $\texttt{megáll}_j$*, mind* $\texttt{inf\_megállt}(j)_i$ *esemény előfordult, akkor a* $\texttt{megáll}_j$ *eseménytől az* $\texttt{inf\_megállt}(j)_i$ *eseményig eltelt idő határozottan nagyobb d-nél.*

2. Ha a PSzinkHJ *valamely időzített végrehajtási sorozatában előfordult* $\texttt{megáll}_j$ *esemény, akkor a* $\texttt{megáll}_j$ *esemény után* $Ld+d+\mathcal{O}(L\ell_2)$ *időn belül vagy egy* $\texttt{inf\_megállt}(j)_i$*, vagy egy* $\texttt{megáll}_i$ *esemény következik be.*

Bizonyítás.

1. A 25.1. tétel bizonyításához hasonlóan az $\texttt{inf\_megállt}(j)_i$ esemény előfordulásakor P_i már valamely $a > d+\ell_2$ idő óta nem kapott üzenetet P_j-től. Tegyük fel, hogy az $\texttt{inf\_megállt}(j)_i$ esemény a t időpontban következik be. Ekkor a $(t-a,t)$ időintervallumban már nem érkezett üzenet P_j-től P_i-nek. Így szükségképpen P_j nem is küldhetett üzenetet P_i-nek a $(t-a,t-a+\ell_2]$ időintervallumban, mert különben annak meg kellett volna érkezni P_i-hez a $(t-a,t-a+\ell_2+d]$ időintervallumban, amelyet tartalmaz $(t-a,t)$. Ez viszont azt jelenti, hogy P_j-nek a $t-a+\ell_2 < t-d$ időpontig már meg kellett állnia, s éppen ezt kellett belátnunk.
2. Tekintsük PSzinkHJ egy olyan megengedett időzítésű végrehajtási sorozatát, amely során $\texttt{megáll}_j$ a t időpontban fordul elő. Ekkor a t időpont után P_j már nem küld üzenetet P_i-nek, így a $t+d$ időpont után P_i nem is kaphat

üzeneteket P_j-től. Az utolsó üzenet vétele után P_i legfeljebb $m\ell_2$ idő alatt tesz meg m lépést. Mivel $m = \lceil (d+\ell_2)/\ell_1 + 1 \rceil, m\ell_2 = Ld + \mathcal{O}(L\ell_2)$. Tehát ha P_i nem hibázik közben, akkor a $\texttt{megáll}_j$-től a $\texttt{inf\_megállt}(j)_i$-ig eltelt összes idő $Ld + d + \mathcal{O}(L\ell_2)$, amit bizonyítani kellett.

□

A 25.2 tétel első részének van egy fontos következménye. Eszerint ha a P_i folyamat P_j-nél időtúllépést érzékelt, akkor biztos lehet benne, hogy a hiba bekövetkezése előtt P_j-nek küldött üzenetek már mind célbaértek.

Mivel feltettük, hogy $L\ell_2$ kicsi d-hez képest, a hibaértesítés időkorlátját közelítőleg $(Ld + d)$-nek vehetjük.

25.3.. Alapvető eredmények

Először azt vizsgáljuk, hogy mit tudunk a megegyezési problémáról a korábbi fejezetek eredményei alapján, és megpróbáljuk kiterjeszteni ezeket az eredményeket a részben szinkron esetre. Legfontosabbaknak a 6.2. és a 7.1. alfejezetekben található, a $k+1$ menetes f hibás megegyezésre vonatkozó alsó, illetve felső korlátok bizonyulnak, melyek a szinkron modellre vonatkoznak.

25.3.1.. Felső korlát

A 6.2. alfejezet tartalmaz néhány olyan algoritmust, melyek a szinkron hálózati modellben megállási hibák esetében megoldják a megegyezési problémát. Az f megállási hibát megengedő legtöbb algoritmusnak pontosan $f+1$ menetre van szüksége. Bármelyik ilyen algoritmus átalakítható részben szinkron futtatásra. Az átalakítás a következőképpen történhet.

Legyen A egy teljes gráf hálózatra készült tetszőleges szinkron hálózati algoritmus. Emlékezzünk vissza, hogy a szinkron modellre vonatkozó megállapodások szerint a bemenetek a kezdőállapotoknál jelennek meg, a kimenetek pedig kizárólagosan írható helyi változókba kerülnek. Az A segítségével a részben szinkron hálózati modellnek megfelelő A' algoritmust definiálunk.

A' algoritmus

Minden P_i folyamat két MMT automata kompozíciója: a Q_i automatáé, amely a PSzinkHJ algoritmusnak az i-dik csúcshoz tartozó részét tartalmazza, meg az R_i fő automatáé. R_i bemenetei az $\texttt{inf\_megállt}_i$ műveletek. R_i-nek van egy *megállított* változója, ahol azon P_j folyamatok j indexeinek halmazát tárolja, melyektől $\texttt{inf\_megállt}(j)_i$ bemenetet kapott, vagyis amelyekről tudja, hogy hibáztak. R_i-nek van egy másik változója is, melyben az A algoritmus P_i folyamatának szimulált állapotát tárolja.

Az r-dik menet szimulációja úgy zajlik, hogy R_i először meghatározza és szétküldi az A algoritmus r-dik menetének összes üzenetét (ehhez célfolyamatonként egy-egy taszkot használ). Az üzenetek meghatározása az A algoritmus $\textit{üzenetek}_i$ függvényeinek segítségével történik. Ezután R_i addig várakozik, míg minden $j \neq i$-re vagy az r-dik menethez tartozó üzenetet

kap R_j-től, vagy észreveszi, hogy $j \in$ *megállított*. Ezután a kapott üzenetek alapján (minden olyan folyamatnál, melytől nem kapott üzenetet, a `null` üzenetet használva és figyelembe véve A előző állapotát) meghatározza A új szimulált állapotát.

A továbbiakban rögzítsük f-et, és tételezzük fel, hogy A f hibát megengedő, pontosan $f+1$ menetes, a megegyezési problémát megoldó szinkron hálózati algoritmus. A fentiek szerint megalkotjuk az A algoritmus részben szinkron A' változatát, ami már *majdnem* megfelel elvárásainknak. Az egyetlen eltérés abban van, hogy A' a fejezetben használtaktól eltérő beviteli/kiviteli szabályokat használ. Ezért állítsuk elő A' módosításával a következő B algoritmust: B-ben R_i működésének megkezdésekor addig nem kezdi el A szimulálását, míg nem kap egy $\texttt{kezd}(v)_i$ bemenetet. Ekkor elhelyezi v értékét szimulált bemeneti változójába, és elkezdi az első menet szimulációját. (Viszont Q_i rögtön az időzített végrehajtási sorozat elején elkezdi az időtúllépések figyelését.) Továbbá B-ben az R_i a v változó értékének kimenő változójába írása után azonnal végrehajt egy $\texttt{dönt}(v)_i$ kimenő műveletet.

25.3. tétel. . *B megoldja a részben szinkron modellben a megegyezési problémát és biztosítja az f-hibás befejezést. Továbbá minden olyan megengedett időzítésű végrehajtási sorozatban, amikor az összes kapura érkezik bemenet, és legfeljebb f hiba fordul elő, az utolsó* `kezd` *esemény bekövetkezése után legfeljebb $f(Ld+d)+d+\mathcal{O}(fL\ell_2)$ idő múlva minden nemhibázó folyamat döntésre jut.*

Bizonyítás. Könnyen belátható, hogy B helyesen szimulálja A-t, s így B valóban megoldja a megegyezési problémát. Az időkorlátot a működés alapján igazoljuk. Rögzítsük B valamely α megengedett időzítésű végrehajtási sorozatát. Legyen S a PSzinkHJ algoritmus olyan felső korlátja, melyre $S = Ld + d + \mathcal{O}(L\ell_2)$. A 25.2. tétel szerint ilyen S létezik. Definiáljuk az időre vonatkozó $T(0), T(1), \ldots$ mérföldkövek sorozatát. Látni fogjuk, hogy a $T(r)$ mérföldkő annak az időpontnak felső korlátja, amikorra minden még nem hibázott folyamat befejezi az r-dik menet szimulálását.

Legyen $T(0)$ az α sorozatban az utolsó `kezd` esemény bekövetkezésének időpontja. Ezután legyen

$$T(1) = \begin{cases} T(0) + \ell_2 + S & \text{ha valamelyik folyamat hibázik a } T(0)+\ell_2 \text{ időpontig,} \\ T(0) + \ell_2 + d & \text{egyébként.} \end{cases}$$

Végül $r \geq 2$ esetében legyen

$$T(r) = \begin{cases} T(r-1) + \ell_2 + S & \text{ha valamelyik folyamat hibázik a} \\ & (\,T(r-2)+\ell_2, T(r-1)+\ell_2\,) \text{ intervallumban,} \\ T(r-1) + \ell_2 + d & \text{egyébként.} \end{cases}$$

Mivel S a hibák érzékelési idejének felső korlátja, könnyen igazolható az alábbi állítás.

25.4. segédtétel. . *Legyen $r \geq 0$, és jelöljön j tetszőleges folyamatindexet. Ha a P_j folyamat hibázik a $T(r) + \ell_2$ időpontig, akkor minden nemhibázó folyamat legkésőbb a $T(r+1)$ időpontig érzékeli, hogy j hibázott.*

Most már beláthatjuk a következő kulcsfontosságú észrevételt.

25.5. segédtétel. . *Minden $r \geq 0$-ra $T(r)$ felső időkorlát az A r-dik menetének szimulációjára bármely még nem hibázott folyamat esetében.*

Bizonyítás. r szerinti indukcióval bizonyítunk.

Az $r = 0$ eset triviális.

Az indukciós lépésnél tegyük fel, hogy $r \geq 1$. Ha a P_j folyamat a $T(r-1)+\ell_2$ időpontig hibázott, akkor a 25.4. állításból következik, hogy az összes még nem hibázott folyamat legkésőbb $T(r)$-ig érzékeli P_j időtúllépését. Másrészt ha a P_j folyamat a $T(r-1)+\ell_2$ időpontig nem hibázik, akkor az r-dik menethez tartozó összes üzenetét szét tudja küldeni a $T(r-1)+\ell_2$ időpontig. Ezek az üzenetek legkésőbb a $T(r-1)+\ell_2+d \leq T(r)$ időpontig célbaérnek. Vagyis minden folyamat be tudja fejezni az r-dik menet szimulációját $T(r)$-ig. □

A 25.3. tétel bizonyításának befejezéséhez megmutatjuk a kívánt időkorlát érvényességét. A 25.5. állítás szerint $T(f+1)$ felső időkorlát a nemhibázó folyamatokban az $(f+1)$-dik menet szimulációjának befejezésére, ezért $T(f+1)+\mathcal{O}(\ell_2)$ felső időkorlát a nemhibázó folyamatok `dönt` kimeneti műveletének végrehajtási idejére. Mivel legfeljebb f hiba lehet, a mérföldkövek definíciója alapján adódik

$$T(f+1) \leq T(0) + f(\ell_2 + S) + (\ell_2 + d).$$

Az S-re vonatkozó korlátot beírva

$$T(f+1) \leq T(0) + f(Ld + d) + d + \mathcal{O}(fL\ell_2),$$

amiből a kívánt korlát következik. □

25.3.2.. Alsó korlát

A 6.33. tételben megmutattuk, hogy a szinkron hálózati modellben f hibás folyamat esetében a megegyezési feladat megoldásához szükséges menetek számának $f+1$ alsó korlátja. Kis erőfeszítéssel ez a korlát kiterjeszthető a részben szinkron modellre is, most az $(f+1)d$ korlát adódik. Figyeljük meg, hogy a korlátban nem szerepel az L időzítési bizonytalanság.

25.6. tétel. . *Tegyük fel, hogy $n \geq f+2$. Ekkor az f-hibamentességet garantáló részben szinkron modellben nem létezik olyan n-folyamatos megegyezési algoritmus, amelynél a nemhibázó folyamatok minden esetben szigorúan az $(f+1)d$ időpont előtt döntenének.*

Bizonyításvázlat. Vegyük azt az indirekt feltételt, hogy mégis létezik ilyen A algoritmus. A-t átalakítjuk f-menetes A' szinkron algoritmussá, s ezzel ellentmondásra jutunk a 6.33. tétellel.

Az A algoritmusnak természetesen akkor is helyesen kell működnie, ha csak a részben szinkron modell azon speciális eseteit vesszük, melyekben csupán bizonyos átfedési és időzítési megszorításokat teljesítő időzített végrehajtási sorozatokat tekintünk.

1. Az összes bemenet mindjárt kezdetben, a $t = 0$ időpontban érkezik.
2. Minden folyamat olyan lassan halad, amennyire csak az ℓ_2 felső korlát megengedi; így a folyamatok minden helyileg vezérelt lépése az ℓ_2 egész számú többszörösének megfelelő időpontban következik be.[1]
3. Az $[rd, (r+1)d]$ időintervallumban elküldött minden üzenet pontosan az $(r+1)d$ időpontban ér célba minden $r \in \mathbb{N}$ esetében. Továbbá minden P_i folyamat a neki egyidejűleg eljuttatott üzeneteket a küldő folyamatok sorszámának megfelelő sorrendben kapja meg.
4. Az olyan időpontokban, amelyek ℓ_2-nek és d-nek is többszörösei, az összes üzenet kézbesítése a helyileg vezérelt lépések megtétele előtt történik.

Nevezzük a fenti megkötésekkel kiegészített részben szinkron modellt *erős időzítésű* modellnek. Az A algoritmust az erős időzítésű modellben tekintjük. Az általánosság megszorítása nélkül feltehető, hogy A „determinisztikus" abban az értelemben, hogy az összes folyamat minden taszkjának bármely állapotában legfeljebb egy lokálisan vezérelt megengedett művelet van, továbbá minden állapothoz és művelethez legföljebb egy új lehetséges állapot van. Mivel az összes üzenet továbítása a d többszöröseinek megfelelő időpontokban történik. és a folyamatok még szigorúan az $(f+1)d$ időpont előtt döntenek, az általánosság csorbítása nélkül még azt is feltehetjük, hogy a folyamatok az fd időpontban kapott üzeneteket közvetlenül követő lépésben döntenek.

Ki fog derülni, hogy az A algoritmus viselkedése az erős időzítésű modellben nagyon közel áll egy f-menetes szinkron hálózati algoritmuséhoz. Kicsit részletesebben minden $r \geq 1$-re igaz, hogy az $[(r-1)d, rd]$ intervallumban elküldött üzeneteket egyértelműen meghatározza a folyamatoknak közvetlenül az $(r-1)d$ időpontban megkapott üzenetek utáni állapota, mivel az $(r-1)d$ és rd időpontok között nem érkeznek üzenetek. Így megpróbálhatjuk egy szinkron algoritmus r-dik menetének megfelelő üzeneteknek tekinteni az összes ilyen üzenetet.

Van azonban egy lényeges technikai különbség. Ha a szinkron modellben a P_i folyamat az r-dik menetben hibázik, akkor minden $j \neq i$-re a P_i vagy el tudja juttatni az r-dik menetre vonatkozó *összes* információját P_j-nek, vagy *semmit* sem tud átadni P_j-nek. Az A algoritmusban ez annak felelne meg, hogy ha P_j-nek át tudja adni az r-dik menetre vonatkozó összes információt, P'_j-nek meg nem, akkor az $[(r-1)d, rd]$ intervallumban az *összes* üzenetet P_j-nek küldi, és P'_j-nek ebben az intervallumban *semmit* sem küld. Ez a viselkedés azonban nem lehetséges az erős időzítésű modellben.

Ahhoz, hogy A-t szinkron algoritmussá alakítsuk, célszerű magát a szinkron modellt kissé általánosítani. Nevezetesen azt is megengedni, hogy az r-dik menet során minden P_i nemcsak *egyetlen*, hanem *véges sok* üzenetet küldhessen a többi folyamatnak. Megengedjük, hogy P_i hibázása esetében ez az üzenetsorozat bármely kezdőszelet után megszakadhasson. Nem nehéz belátni, hogy a 6.33. tétel

[1] Emlékezzünk vissza, hogy minden taszknak mindig van megengedett lépése.

bizonyítása erre a kicsit általánosabb modellre is kiterjeszthető. Csupán az üzenetsorozat tagjainak egyenként hozzáadásához és eltávolításához szükséges extra lépésekkel kell kibővíteni a 6.33. tétel bizonyításában megkonstruált láncot.

Ezzel lehetővé vált, hogy az A megegyezési algoritmust ebben az erősebb szinkron modellben olyan A' algoritmussá alakítsuk át, amelynek minden végrehajtási sorozata megfelel az A egy időzített végrehajtási sorozatának. Az A' r-dik menetében a P_i folyamat által elküldött üzenetek sorozata azokból az üzenetekből áll, melyeket P_i az A-beli r-dik menet lépései során küld ki az $[(r-1)d, rd]$ intervallumban. Az A'-ben előforduló hibák által okozott viselkedés így már megfelel az A lehetséges viselkedésének. A kapott A' algoritmus az erősebb szinkron modell f-menetes egyetértési algoritmusa az $n \geq f+2$ esetre. Ez viszont ellentmond a 6.33. tételnek. □

A 25.6. tétel egy másik lehetséges bizonyítása a 6.33. tételéhez hasonló új láncot használó olyan érvelés lenne, amely közvetlenül az erős időzítésű modell fogalmi apparátusát használná. Most is extra lépéseket kellene beiktatni a láncba a „menetek közben" küldött üzenetek hozzáadására, illetve eltávolítására.

25.4.. Egy hatékony algoritmus

A 25.3. alfejezetben ismertetett két eredmény után még marad egy érdekes hézag az időbonyolultságok között. Míg a felső korlát közelítőleg $fLd+(f+1)d$, az alsó csupán $(f+1)d$. A legszembetűnőbb különbség, hogy az L időzítési bizonytalanság csak a felső korlátban szerepel. Meg szeretnénk érteni, hogy a problémában rejlő bonyolultság hogyan függ az időzítési bizonytalanságtól.

L az időbonyolultságra gyakorolt hatásának megértése csak L nagyságától függő mértékben bír gyakorlati jelentőséggel. Ha az A algoritmus minden P_i folyamatát úgy futtatjuk külön dedikált processzoron, hogy P_i lépéseinek sebességét a processzor nagy pontosságú órája vezérli, akkor L tipikusan nagyon kicsi lesz, és A bonyolultságának L-től való függése nem sok vizet zavar. Másrészt viszont ha a folyamatok sebességét egyéb tényező (például a folyamatok cseréjéhez szükséges idő) határozza meg, akkor L lehet egészen nagy, s a tőle való függés fontossá válhat. A kérdés elméleti szempontból mindenképpen érdekes.

Az Olvasó először azt gondolhatja, hogy a 25.6. tételből kapott alsó korlátot javítani lehetne egy L tényező beiktatásával. Látni fogjuk, hogy ez az út nem járható, mivel van olyan elég „ravasz" algoritmus, amelynek futásideje közelítőleg $Ld + (2f+2)d$. Ez durván szólva azt jelenti, hogy csak egy üzenet továbbítási ideje „nyúlik meg" az L időzítésibizonytalansággal. Létezik olyan bonyolultabb alsó korlát bizonyítás is, amelyből az $Ld+(f-1)d$ korlát adódik. Az algoritmust ebben az alfejezetben, az alsó korlátot a következő 25.5. alfejezetben ismertetjük.

25.4.1.. Az algoritmus

Megadjuk a PSzinkMegegyezés részben szinkron algoritmust, amely legfeljebb f hiba esetében az $Ld + (2f+2)d + \mathcal{O}(f\ell_2 + L\ell_2)$ időkorláttal biztosítja a várakozásmentes befejezést. A PSzinkMegegyezés algoritmus leírása nagyon

egyszerű, de működését elég nehéz megérteni. Azt tanácsoljuk, hogy az Olvasó próbáljon egy saját megoldást kitalálni az algoritmus elolvasása előtt.

Amikor a PSzinkMegegyezés algoritmusban azt mondjuk, hogy egy folyamat küldjön bizonyos üzeneteket „minden folyamatnak", akkor ebbe beleértjük magát a küldő folyamatot is. Modellünk valójában nem engedi meg ezt a viselkedést, de a szokásos módon, belső lépések beiktatásával szimulálhatjuk.

PSzinkMegegyezés algoritmus (vázlatosan)
Az algoritmus ugyanúgy használja a PSzinkHJ algoritmust, mint a 25.3.1. szakasz B automatája. Vagyis a PSzinkMegegyezés minden P_i folyamata két MMT automata kompozíciója: a Q_i automatáé, amely a PSzinkHJ algoritmusnak az i-dik csúcshoz tartozó részét tartalmazza, meg az R_i fő automatáé. R_i bemenetei az $\texttt{inf\_megállt}_i$ műveletek. R_i-nek van egy *megállított* változója, ahol azon P_j folyamatok j indexeinek halmazát tárolja, melyektől $\texttt{inf\_megállt}(j)_i$ bemenetet kapott, vagyis amelyekről tudja, hogy hibáztak.

Az algoritmus a $0, 1, 2, \ldots$ „menetekből" áll. Minden menet során az összes R_i dönteni próbál, de a páros sorszámú menetekben csak 0-t, a páratlan sorszámúakban meg csak 1-et dönthet. R_i csak az első bemenet megérkezése után kezdi el a 0. menetet. R_i-nek van egy *döntött* nevű változója, ebben jegyzi fel azon folyamatokat, amelyektől `döntött` üzenetet kapott.

0. *menet*:
1 bemenetre R_i sorban a következőket végzi:
  `goto`(1) üzenetet küld minden folyamatnak
  áttér az 1. menetre
0 bemenetre R_i sorban a következőket végzi:
  `goto`(2) üzenetet küld minden folyamatnak
  $\texttt{döntött}(0)_i$ kimenetet ad
  `döntött` üzenetet küld minden folyamatnak

r-dik menet ($r > 0$):
R_i addig vár, míg vagy $\texttt{goto}(r+1)$ üzenetet nem kap, vagy $\texttt{goto}(r)$ üzenetet nem kapott minden olyan folyamattól, amely nem eleme a *megállított* $\cup$ *döntött* halmaznak. Ha $\texttt{goto}(r+1)$ üzenet érkezett, akkor R_i sorban a következőket végzi:
  $\texttt{goto}(r+1)$ üzenetet küld minden folyamatnak
  áttér az $(r+1)$-dik menetre
Egyébként, vagyis ha nem kapott $\texttt{goto}(r+1)$ üzenetet, de minden olyan folyamattól, amely nem eleme a *megállított* $\cup$ *döntött* halmaznak, megérkezett a $\texttt{goto}(r)$ üzenet, akkor R_i sorban a következőket végzi:
  $\texttt{goto}(r+2)$ üzenetet küld minden folyamatnak
  $\texttt{döntött}(r \bmod 2)_i$ kimenetet ad
  `döntött` üzenetet küld minden folyamatnak

R_i tehát saját kezdőértékének megvizsgálásával kezdi működését. Ha a kezdőérték 1, R_i továbblép az 1. menetre, miután a többi folyamatot is erre utasította. Ha viszont a kezdeti értéke 0, akkor R_i a 0 érték mellett dönt, miután a többi folyamatot a 2. menetre való áttérésre utasítja. Ezzel egyben meggátolja a többi folyamat első menetbeli döntését, valamint kiküszöböli az ebből származó ellentmondásos helyzetet. (Figyeljük meg, hogy az algoritmus előnyben részesíti a kezdeti 0 döntést.)

Ha R_i egy későbbi r-dik menet során olyan utasítást kap, hogy térjen át az $(r+1)$-dik menetre, akkor így is tesz, miután a többi folyamatot is erre utasította. Ha viszont R_i-nek nem mondták, hogy térjen át az $(r+1)$-dik menetre, és értesül arról, hogy minden nemhibázó vagy döntésre jutó folyamat már elérte az r-dik menetet, akkor dönthet az r mod 2 érték mellett.

25.4.2.. Biztonságossági tulajdonságok

Először a biztonságossági tulajdonságokat: a jólformáltságot, a megegyezést és az érvényességet igazoljuk. A bizonyítás két lemmán alapul. Azt mondjuk, hogy a P_i folyamat *dönteni próbál* az r-dik menetben, ha legalább egy `goto`$(r+2)$ üzenetet küld az r-dik menetben való döntés előkészítéseként.

25.7. lemma. . *Tetszőleges $r \geq 0$ esetében a* PSzinkMegegyezés *algoritmus bármely időzített végrehajtási sorozatában igazak az alábbiak:*

1. *ha legalább egy folyamat* `goto`$(r+2)$ *üzenetet küld, akkor valamelyik folyamat dönteni próbál az r-dik menet során;*
2. *ha legalább egy folyamat eléri az $(r+2)$-dik menetet, akkor valamelyik folyamat dönteni próbál az r-dik menetben.*

Bizonyítás. Az első `goto`$(r+2)$ üzenet csak ezen a módon jöhet létre. A folyamatok csak egy `goto`$(r+2)$ üzenet vétele után térnek át az $(r+2)$-dik menetre. □

25.8. lemma. . *Tetszőleges $r \geq 0$ esetében a* PSzinkMegegyezés *algoritmus bármely időzített végrehajtási sorozatában, ha a P_i folyamat dönt az r-edik menet során, akkor az alábbiak teljesülnek:*

1. *R_i nem küld* `goto`$(r+1)$ *üzenetet;*
2. *R_i* `goto`$(r+2)$ *üzenetet küld minden folyamatnak;*
3. *Egyetlen folyamat sem próbál dönteni az $(r+1)$-dik menetben.*

Bizonyítás. Az első két állítás világos az algoritmus leírása alapján. A harmadik igazolásához vegyük azt az indirekt feltevést, hogy R_j dönteni próbál az $(r+1)$-dik menetben. Ez azt jelenti, hogy az $(r+1)$-dik menet közben valamikor R_j már `goto`$(r+1)$ üzenetet kapott minden olyan folyamattól, amely nincs benne a $megállított_j \cup döntött_j$ halmazban, de nem kapott egyetlen `goto`$(r+2)$ üzenetet sem. Mivel R_i nem küld `goto`$(r+1)$ üzenetet, $i \in megállított_j \cup döntött_j$.

Ha ebben a pillanatban $i \in megállított_j$, akkor a 25.2. tételből következik, hogy R_j-nek már meg kellett kapnia R_i-től az annak hibázása előtt küldött összes

üzenetet. De a második állítás szerint ezek között kell lennie `goto`$(r+2)$ üzenetnek is, ami ellentmondás.

Ha viszont $i \in$ *döntött*$_j$, akkor R_j-nek `döntött` üzenetet kellett kapnia R_i-től. De R_i csak `goto`$(r+2)$ üzenetének elküldése után küld ilyen üzenetet R_j-nek, ami a csatornák FIFO tulajdonsága miatt azt jelenti, hogy a kérdéses időpontra már R_j-nek meg kellett volna kapnia a `goto`$(r+2)$ üzenetet, ami ismét ellentmondás. □

Most már meg tudjuk mutatni a biztonsági tulajdonságok teljesülését.

25.9. tétel. . *A* PSzinkMegegyezés *algoritmus biztosítja a jólformáltságot, a megegyezést és az érvényességet.*

Bizonyítás. A jólformáltság nyilvánvaló. Az érvényességhez gondoljuk meg a következőket. Ha minden folyamat a 0 értékkel indul, akkor egyik sem fogja soha elhagyni a 0. menetet. Mivel az 1 döntés csak a páratlan sorszámú menetekben lehetséges, egyetlen folyamat sem dönthet az 1 mellett. Ha viszont minden folyamat az 1 értékkel kezd, akkor egyik sem próbál a 0 érték mellett dönteni a 0. menetben. Ekkor az 25.7. lemmából következik, hogy egyetlen folyamat sem éri el a 2. menetet, amiből következik, hogy egyik sem dönthet 0-t.

A megegyezéshez tegyük fel, hogy R_i úgy dönt az r-dik menetben, hogy korábban még egyik folyamat sem döntött. Így a 25.8. lemma szerint egyik folyamat sem próbál dönteni az $(r+1)$-dik menetben. A 25.7. lemma miatt egyik folyamat sem érheti el az $(r+3)$-dik menetet. Vagyis a folyamatok csak az r-dik vagy az $(r+2)$-dik menetben dönthetnek. De mivel ezek paritása megegyezik, az összes döntés meg fog egyezni. □

25.4.3.. Élénkség és bonyolultság

A továbbiakban a hibamentes befejezést és az időkorlátot bizonyítjuk. A megengedett időzítésű végrehajtási sorozatokra vonatkozó élénkségi segédtétellel kezdünk.

25.10. lemma. . *A* PSzinkMegegyezés *algoritmus bármely megengedett időzítésű végrehajtási sorozata során minden folyamat folyamatosan halad menetről menetre mindaddig, míg nem hibázik vagy nem dönt.*

Bizonyítás. Ha az állítás nem lenne igaz, legyen az r-dik az első menet, amely során valamelyik folyamat „beragad". Ekkor r legalább 1. Jelölje i egy ilyen beragadt folyamat indexét. Az összes többi olyan R_j-re, amely valamikor hibázik, R_i-nek érzékelni kell a hibát és j-t be kell tenni *megállított*$_i$-be. Továbbá az összes többi olyan R_j esetében, amelyek soha nem hibáznak, de valamikor döntenek, R_i-nek előbb-utóbb fel kell fedeznie, hogy R_j döntött, és j-t el kell helyeznie *döntött*$_i$-be. Jelölje I a megmaradó folyamatok halmazát, vagyis azokét, amelyek soha nem döntenek és nem is hibáznak.

Ekkor az I minden folyamatának el kell érnie előbb-utóbb az r-dik menetet, mivel r az első menet, amely során valamelyik folyamat beragadhat. Mivel $r \geq 1$, minden $j \in I$ esetében az R_j-nek `goto`(r) üzenetet kell küldenie R_i-nek,

amit R_i végül meg is kap. Így azonban teljesülnek az R_i döntéséhez szükséges feltételek, ezért R_i-nek vagy döntenie kell, vagy tovább kell lépnie az $(r+1)$-dik menetbe. Ez pedig ellentmond kiinduló feltételünknek, miszerint R_i beragadt az r-dik menetben. □

Az élénkségi és a bonyolultsági bizonyítások során hasznos lesz a következő fogalom, melynek rögtön meg is mutatjuk néhány tulajdonságát. A PSzinkMegegyezés algoritmus valamely megengedett időzítésű végrehajtási sorozatában az r-dik menet *csendes*, ha *van olyan* folyamat, amely soha, egyetlen másik folyamattól sem kap `goto`$(r+1)$ üzenetet. Ezt az új definíciót egyes korábbi lemmákkal ötvözve az alábbiakat kapjuk.

25.11. lemma. . *Tetszőleges $r \geq 0$ esetében a* PSzinkMegegyezés *algoritmus bármely időzített végrehajtási sorozatában igazak az alábbiak:*

1. *ha egyik folyamat sem próbál dönteni az r-dik menetben, akkor az $(r+1)$-dik menet csendes.*
2. *ha valamelyik folyamat dönt az r-dik menetben, akkor az $(r+2)$-dik menet csendes.*

Bizonyítás. Az első állítás azonnal adódik a 25.7. lemmából. A másodikkal kapcsolatban a 25.8. lemmából az következik, hogy ha valamelyik folyamat dönt az r-dik menetben, akkor egyik folyamat sem próbál dönteni az $r+1$-dik menetben, tehát az első állítás szerint az $(r+2)$-dik menet csendes. □

A csendes menet fogalmának fontosságát az adja, hogy soha egyetlen folyamat sem léphet túl egy csendes meneten.

25.12. lemma. . *Ha a* PSzinkMegegyezés *algoritmus valamely időzített végrehajtási sorozatában az r-dik menet csendes, akkor soha egyetlen folyamat sem éri el az $(r+1)$-dik menetet.*

Bizonyítás. Ha az R_i folyamat áttér az $(r+1)$-dik menetre, akkor előtte minden folyamatnak `goto`$(r+1)$ üzenetet küld. Ezek végül meg is érkeznek, vagyis az r-dik menet nem csendes. □

Most azt mutatjuk meg, hogy elő kell fordulnia csendes menetnek.

25.13. lemma. . *A* PSzinkMegegyezés *algoritmus minden legfeljebb f hibát tartalmazó megengedett időzítésű végrehajtási sorozatában legkésőbb az $f+2$ sorszámú menet csendes lesz.*

Bizonyítás. Ha valamelyik folyamat dönt az f-dik menetig, akkor az állítás következik a 25.11. lemmából. Ezért tegyük fel, hogy egyik folyamat sem döntött az f-dik menetig. Mivel legfeljebb f hiba fordulhat elő, van olyan $0 \leq r \leq f$, hogy az r-dik menetben egyik folyamat sem hibázik.

Azt állítjuk, hogy az r-dik menetben egyik folyamat sem próbál dönteni. Az ellentmondás kedvéért tegyük fel, hogy valamely P_i folyamat dönteni próbál. Ekkor – mivel P_i nem hibázik az r-edik menetben – valóban döntenie kell az r-dik menetben a megengedettség definíciója miatt. Ez viszont ellentmond annak a feltevésnek, hogy egyetlen folyamat sem dönt az f-edik menetig.

Mivel egyik folyamat sem próbál dönteni az r-edik menetben, az $(r+1)$-edik menet csendes lesz a 25.11. lemma miatt. □

Most már beláthatjuk a várakozásmentes befejeződést.

25.14. tétel. . *A* PSzinkMegegyezés *algoritmus biztosítja a várakozásmentes befejeződést.*

Bizonyítás. Tekintsünk egy olyan megengedett időzítésű végrehajtási sorozatot, amely során minden kapun előfordul a `kezd` esemény. Legyen i tetszőleges hibátlan kapu. Azt állítjuk, hogy R_i előbb-utóbb dönteni fog.

A 25.10. lemma szerint R_i menetről-menetre folyamatosan halad tovább, míg csak nem dönt. De a 25.13. lemmából az következik, hogy lesz egy r-dik csendes menet, ezután a 25.12. lemma alapján R_i már nem tud tovább lépni az $(r+1)$-dik menetbe. Ebből következik, hogy R_i-nek dönteni kell. □

Befejezésül a bonyolultsági korlátot bizonyítjuk. Rögzítsük úgy f-et, a hibák számát, hogy $0 \leq f \leq n$ legyen.

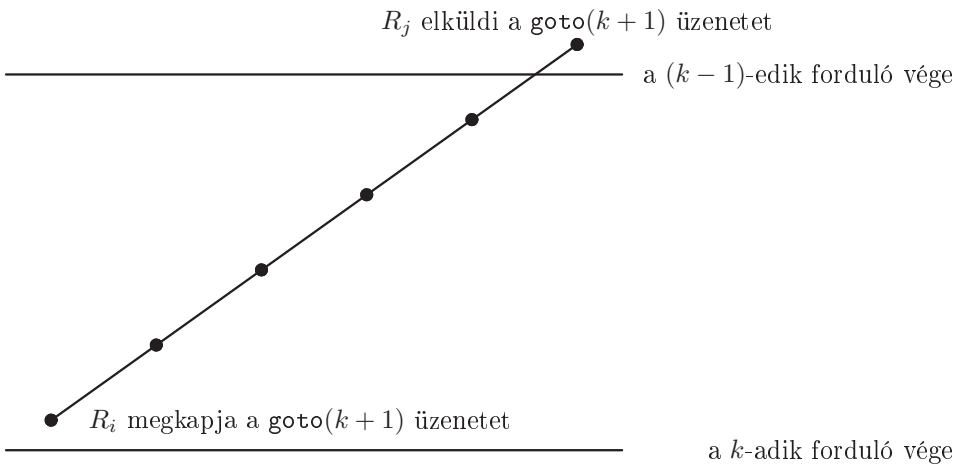
25.15. tétel. . *A* PSzinkMegegyezés *algoritmus minden olyan megengedett időzítésű végrehajtási sorozatában, amikor az összes kapun érkezik bemenet, és legfeljebb f hiba fordul elő, az utolsó* `kezd` *esemény bekövetkezése után legfeljebb $Ld + (2f+2)d + \mathcal{O}(f\ell_2 + L\ell_2)$ idő múlva minden nemhibázó folyamat döntésre jut.*

Bizonyításvázlat. A 25.14. tétel bizonyítása és a hozzá tartozó lemmák azt mutatják, hogy a végrehajtási sorozatnak legfeljebb $f+1$ olyan nem-csendes menetből kell állnia, amelyet egyetlen csendes menet követ (legyen ez az r-edik). Minden hibátlan folyamatnak dönteni kell anélkül, hogy túlhaladna az r-dik meneten.

Legyen S a PSzinkMegegyezés algoritmus végrehajtási idejének felső korlátja (vehető $S = Ld + d + \mathcal{O}(L\ell_2)$). Vezessük be az időre vonatkozó $T', T(0), T(1), \ldots, T(r)$ mérföldkövek sorozatát a következő módon. Legyen T' az utolsó `kezd` esemény bekövetkezésének időpontja. Tetszőleges $0 \leq k \leq r$ esetében legyen $T(k)$ az a legkorábbi időpont, amikorra minden folyamat vagy döntött, vagy hibázott, vagy áttért a $(k+1)$-dik menetre. Vagyis $T(r)$-ig minden nemhibázó folyamat dönt. Könnyen belátható, hogy a nulladik menet időtartama, $T(0) - T'$ $\mathcal{O}(L\ell_2)$ nagyságrendű. Továbbá $k \geq 1$ esetében a k-dik menet időtartama, $T(k) - T(k-1)$, legfeljebb $S + \mathcal{O}(\ell_2)$, vagyis alig több, mint az egy hiba érzékeléséhez szükséges idő. Tehát $T(k) - T(k-1) \leq Ld + d + \mathcal{O}(L\ell_2)$.

Még érdekesebb az a tény, hogy bármely $0 \leq k \leq (r-1)$ esetében a *nem-csendes* k-dik menet végrehajtási ideje, $T(k) - T(k-1)$, nem függ az L időzítési bizonytalanságtól. Ennek belátásához vegyünk egy tetszőleges R_i folyamatot. Mivel a k-dik menet nem csendes, R_i kapni fog `goto`$(k+1)$ üzenetet. A megérkezéséhez szükséges időre adunk meg felső korlátot.

Valamely–a $(k-1)$-dik menet során dönteni próbáló–R_j folyamat által (esetleg közvetítők sorozatán keresztül) elküldött eredeti `goto`$(k+1)$ üzenetből kellett erednie ennek az üzenetnek (lásd a 25.1 ábrát).



25.1.. ábra. Az R_j-től R_i-hez továbbított *goto*$(k+1)$ üzenetek.

25.16. segédtétel. . *Jelölje f_k a* goto$(k+1)$ *üzenet küldése közben hibázó folyamatok számát. Ekkor az eredeti* goto$(k+1)$ *üzenet R_j általi elküldése és a* goto$(k+1)$ *üzenetnek az R_i-hez való megérkezése közt eltelt összes idő legfeljebb $(f_k+1)d+\mathcal{O}(f_k\ell_2)$.*

Bizonyítás. R_j minden folyamatnak próbál goto$(k+1)$ üzenetet küldeni, beleértve R_i-t is. Ha R_j nem hibázik ezen próbálkozás közben, akkor sikerül R_i-nek elküldenie az üzenetet, amit R_i az elküldéstől számított d időn belül meg is kap. Még ha hibázik is R_j, a hiba bekövetkezése előtt feladott üzenetek mind célba érnek elküldésük után legfeljebb d idővel.

Hasonló a helyzet minden olyan $R_{j'}$ folyamattal, amely az üzenet R_j-től R_i-hez való továbbításában vesz részt. Ezek a goto$(k+1)$ üzenetet minden folyamatnak megkísérlik eljuttatni, beleértve R_i-t is. Ismét igaz, hogy ha $R_{j'}$ nem hibázik, akkor sikerül R_i-nek elküldenie az üzenetet, amit R_i az elküldéstől számított d időn belül megkap. Még ha hibázik is $R_{j'}$, a sikeresen elküldött üzenetei d időn belül célba érnek.

Ebből következik, hogy az eredeti goto$(k+1)$ üzenet R_j általi elküldése és az R_i-hez eljutó valamely goto$(k+1)$ üzenet megérkezése között összesen legfeljebb $(f_k+1)d+\mathcal{O}(f_k\ell_2)$ idő telik el. (Az ℓ_2 tényező az aközben eltelt időért felelős, amíg valamelyik továbbító folyamat a goto$(k+1)$ üzenetet vételétől kezdve szétküldi saját goto$(k+1)$ üzeneteit.) □

Mivel az eredeti goto$(k+1)$ üzenetet R_j a $(k-1)$-dik menet során küldte el, ez legkésőbb a $T(k-1)$ mérföldkőnél történhetett. Mivel minden folyamat kap az $(f_k+1)d+\mathcal{O}(f_k\ell_2)$ időn belül goto$(k+1)$ üzenetet, mindegyik vagy továbblép a $(k+1)$-dik menetre, vagy hibázik, vagy $T(k-1)+(f_k+1)d+\mathcal{O}(f_k\ell_2)+\mathcal{O}(\ell_2)=T(k-1)+(f_k+1)(d+\mathcal{O}(\ell_2))$ időn belül dönt. Ebből következik, hogy $T(k)-T(k-1)\leq(f_k+1)(d+\mathcal{O}(\ell_2))$. Mint korábban említettük, ez az érték

nem függ az L időzítési bizonytalanságtól.

Mivel $T(0) - T' = \mathcal{O}(\ell_2)$, $T(k) - T(k-1) \leq (f_k + 1)(d + \mathcal{O}(\ell_2))$ bármely $1 \leq k \leq r-1$ esetében, továbbá $T(r) - T(r-1) \leq Ld + d + \mathcal{O}(L\ell_2)$, a következőt kaptuk:

$$T(r) - T' \leq \sum_{k=1}^{r-1} (f_k + 1)(d + \mathcal{O}(\ell_2)) + Ld + d + \mathcal{O}(L\ell_2).$$

Azonban $\sum_{k=1}^{r-1} f_k \leq f$ és $r \leq f + 2$, így

$$T(r) - T' \leq Ld + (2f + 2)d + \mathcal{O}(f\ell_2 + L\ell_2),$$

ami éppen a bizonyítandó bonyolultsági korlát. □

25.5.. Az időzítési bizonytalanságot tartalmazó alsó korlát*

A 25.4. alfejezetben megadtuk a PSzinkMegegyezés részben szinkron megegyezési algoritmust, amely közelítőleg $Ld + (2f + 2)d$ időben fut. A PSzinkMegegyezés algoritmussal nagyot léptünk előre a 25.2. alfejezetben bizonyított egyszerű közelítő $fLd + (f+1)d$ felső, és $(f+1)d$ alsó korlátok közti hézag kitöltésében. A PSzinkMegegyezés algoritmus azt is demonstrálja, hogy az fLd tagot tartalmazó alsó korlát bizonyítása reménytelen. Ebben a fejezetben belátjuk az $Ld + (f-1)d$ alsó korlátot, amely függ L-től. Ezzel még mindig marad hézag az alsó és a felső korlát között, de legalább világossá válik az időbonyolultságnak az L időzítési bizonytalanságtól való függése.

25.17. tétel. . *Tegyük fel, hogy $n \geq f + 1$. A részben szinkron modellben nem létezik olyan f-hibás befejezést garantáló n-folyamatos megegyezési algoritmus, amely során minden nemhibázó folyamat szigorúan $Ld + (f-1)d$ időn belül dönt.*

A 25.17. tétel bizonyítása kimondottan érdekes, mivel a korábbi fejezetekben kifejlesztett módszerek kombinációit használja: a 6. fejezetben leírt láncérvelést, a 12. fejezet különböző döntési értékek elérhetőségén alapuló megfontolásait és a 24. fejezetben az időzített végrehajtási sorozatok megnyújtásával, illetve összenyomásával kapcsolatos érveléseket.

A fejezet hátralevő részében azzal az indirekt feltevéssel dolgozunk, hogy létezik olyan olyan f-hibás befejezést garantáló n-folyamatos A megegyezési algoritmus, amely során minden nemhibázó folyamat szigorúan $Ld + (f-1)d$ időn belül dönt. A 25.6. tétel bizonyításához hasonlóan az általánosság megszorítása nélkül most is feltehetjük, hogy A „determinisztikus". Lemmák sorozatának bizonyításán át jutunk arra a végkövetkeztetésre, hogy A nem létezhet.

Először azt igazoljuk a 25.18. lemmában, hogy ha az A algoritmus helyes, akkor nem nem fordulhat elő az időzített végrehajtási sorozatok bizonyos „rossz kombinációja". Ez a „rossz kombináció" olyan „0-értékű" α_0 és „1-értékű" α_1 végrehajtási sorozatokból áll, amelyeket legfeljebb egy nemhibázó folyamat tud megkülönböztetni, mindegyikük eljut legalább az $(f-1)d$ időpontig és együtt is

csak kevés hibájuk van. A 25.18. lemmát nyújtási és összenyomási okoskodással bizonyítjuk. Másodjára a 25.19. lemmában azt igazoljuk, hogy valóban létezik az előbbivel rokon kombináció, amely ugyanezeket a feltételeket teljesíti, kivéve a 0- és az 1-értékűséget, melyek helyett csak a 0-nak az α_0-ban, illetve az 1-nek az α_1-ben való elérhetőségét kötjük ki. A 25.19. lemmát láncérveléssel igazoljuk. Harmadszor a 25.20. lemmában egy olyan „kétértékű" α időzített végrehajtási sorozatot állítunk elő, amelyben kevés hiba fordul elő és legalább $(f-1)d$ ideig fut. A 25.20. lemma közvetlenül adódik a 25.18. és a 25.19. lemmából. Végül a negyedik lépésben, a 25.21. lemmában a 25.20. lemmát élesítjük egy „maximalitási" tulajdonság bevezetésével, mely alapján közvetlenül adódik az α két kiterjesztése, a 0-értékű α_0 és az 1-értékű α_1. Ez az α_0 és α_1 „rossz kombinációt" alkot, s ezzel ellentmondásra jutottunk.

Következzék a részletes leírás. Tüntessük ki az A időzített végrehajtási sorozatainak egy részhalmazát, a „szinkron" végrehajtási sorozatok halmazát. Egy *szinkron* végtelen időzített végrehajtási sorozathoz megadható időpontok olyan $t_0 = 0, t_1, t_2, \ldots$ végtelen sorozata, melyben minden $k \leq 0$-ra $\ell_1 \leq t_{k+1} - t_k \leq \ell_2$, továbbá teljesülnek az alábbi feltételek.

1. Minden bemenet kezdetben, rögtön a $t = 0$ időpontban érkezik.
2. A hibátlan folyamatok minden taszkja pontosan a $t_1, t_2, \ldots$ időpontokban hajtja végre lépéseit. Ezek lesznek az *aktív időpontok*.[2] Továbbá az összes taszk lépései előírt sorrendben követik egymást.
3. A P_i folyamathoz azonos időben érkező üzenetek átadása a küldő folyamatok indexeinek megfelelő sorrendben történik.
4. Az aktív időpontokban az üzenetek kézbesítése a folyamatok összes helyileg vezérelt lépésének végrehajtása előtt történik.

Ezek a feltételek némileg hasonlók a 25.6. tétel bizonyításvázlatában használtakhoz. Egy szinkron végtelen időzített végrehajtási sorozat felosztható a $B_0, B_1, B_2, \ldots$ végtelen sok „blokkra"; a B_k blokk tartalmazza a t_k időpontban végrehajtott összes bemeneti és üzenettovábbítási lépést, de nem tartalmazza a folyamatok t_k-hoz tartozó lokálisan vezérelt lépéseit. Vagyis B_0 csak a bemeneti eseményekből áll, $k \leq 1$ esetében B_k a t_{k-1} időpontbeli lokálisan vezérelt lépésekkel kezdődik és a t_k időpontban végrehajtott üzenettovábbítási lépésekkel fejeződik be. *Véges szinkron időzített végrehajtási sorozaton* valamely végtelen szinkron időzített végrehajtási sorozat véges számú teljes blokkból álló kezdőszeletét értjük.

Legyenek α és α' szinkron időzített végrehajtási sorozatok, α az α' véges kezdőszelete és $k \geq 0$. Az mondjuk, hogy α az α' *k-blokk kezdőszelete*, ha α pontosan az α' $B_0, B_1, B_2, \ldots, B_k$ teljes blokkjaiból áll. (A 0-blokk kezdőszelet speciálisan az egyetlen B_0 blokkot tartalmazza.) Ha α az α' k-blokk kezdőszelete valamely $k \geq 0$-ra, akkor azt mondjuk, hogy α az α' *blokk kezdőszelete*, és α' az α *blokk-kiterjesztése*.

Főleg bizonyos speciális blokk-kiterjesztések érdekelnek bennünket. Nevezetesen ha $k \geq 0$, α véges szinkron időzített végrehajtási sorozat, α' (véges vagy

[2] Újfent emlékeztetjük az Olvasót, hogy feltételezésünk szerint minden taszknak mindig van megengedett lépése.

végtelen) szinkron időzített végrehajtási sorozat és α az α' k-blokk kezdőszelete, akkor azt mondjuk, hogy α'

1. az α *gyors végrehajtása*, ha α'-ben az α utáni összes lépés a minimális ℓ_1 idő alatt hajtódik végre, azaz $t_{i+1} - t_i = \ell_1$ bármely $i \geq k$-ra,
2. az α *lassú végrehajtása*, ha α'-ben az α utáni összes lépés a maximális ℓ_2 idő alatt hajtódik végre, azaz $t_{i+1} - t_i = \ell_2$ bármely $i \geq k$-ra;
3. az α *hibamentes kiterjesztése*, ha α'-ben az α után nem fordul elő a `megáll` esemény;
4. az α *fff-kiterjesztése*, ha az α gyors és hibamentes kiterjesztése.

Hangsúlyozzuk, hogy a kiterjesztések előző típusai mind teljes blokkokból álló blokk-kiterjesztések. Figyeljük meg, hogy a „lassú" és a „gyors" jelző csak a folyamatok lépésidejére vonatkozik, az üzenetek kézbesítési ideje továbbra is a $[0, d]$-be eső tetszőleges érték lehet.

Most néhány olyan fogalmat vezetünk be, melyek hasonlítanak a 12. fejezet aszinkron modelljében az egyetértésre vonatkozó megoldhatatlansági eredmények során használtakra. Azt mondjuk, hogy valamely $v \in \{0, 1\}$ érték *fff-elérhető* az α véges szinkron időzített végrehajtási sorozatból, ha létezik α-nak olyan α' *fff*-kiterjesztése, amely során valamelyik folyamat v-t dönt. (Ez a döntés előfordulhat magában α-ban, vagy az α' α utáni részében.) Az α véges szinkron időzített végrehajtási sorozat 0-*értékű*, ha α-ból csak a 0 érték *fff*-elérhető, 1-*értékű*, ha csak az 1 érték *fff*-elérhető, végül *kétértékű*, ha mindkettő *fff*-elérhető. Az α időzített végrehajtási sorozat *egyértékű*, ha vagy csak 0-értékű vagy csak 1-értékű.

Szükségünk lesz még egy fogalomra, két véges szinkron időzített végrehajtási sorozat valamely P_i folyamat általi „megkülönböztethetetlenségére". Hasonló fogalmakkal találkoztunk már a könyv szinkron és aszinkron fejezeteiben. Az itt megkövetelt fogalom kicsit bonyolultabb a korábbiaknál, mivel tekintetbe veszi a végrehajtási sorozatok végén a P_i felé úton levő üzeneteket is. Nevezetesen az α és α' azonos aktív időkkel bíró véges szinkron időzített végrehajtási sorozatokat akkor mondjuk P_i-re vonatkozóan *megkülönböztethetetlennek* ha a következők teljesülnek.

1. Az α és α' P_i-re való projekciói, $\alpha|P_i$ és $\alpha'|P_i$ időátmenet ekvivalensek.[3]
2. α-ban és α'-ben P_i ugyanazoktól a folyamatoktól ugyanazokban az időpontokban, azonos sorrendben ugyanazokat az üzeneteket kapja.

A következő lemma az időzített végrehajtási sorozatok egy olyan rossz kombinációját írja le, amely az A algoritmus helyessége esetében nem fordulhat elő.

25.18. lemma. . *Nem létezhet két olyan k-blokkos szinkron időzített végrehajtási sorozat, α_0 és α_1, melyre az összes alábbi kikötés teljesülne:*

1. *α_0 és α_1 aktív időjei, $t_1, \ldots, t_k$ megegyeznek és $t_k \geq (f-1)d$;*
2. *α_0 0-értékű;*
3. *α_1 1-értékű;*

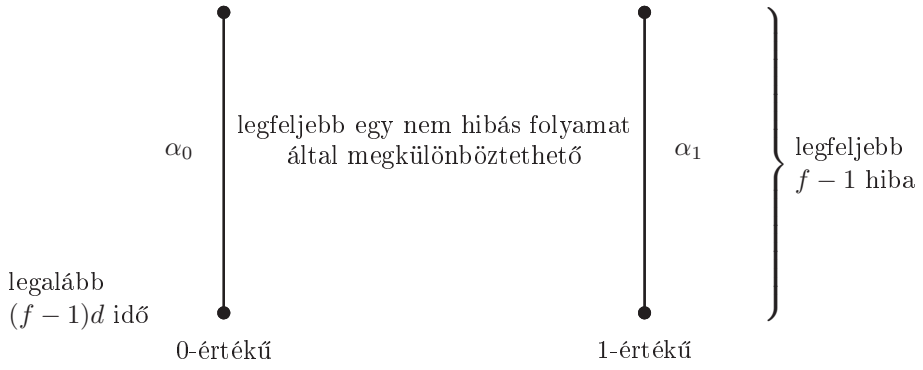
[3] A | projekció műveletet a 23.2.3. , az időátmenet ekvivalenciát pedig a 23.2.1. szakaszban vezettük be.

4. *Az α_0-ban vagy α_1-ben hibázó folyamatok F halmazának számosságára $|F| \leq f - 1$ teljesül.*
5. *α_0 és α_1 legfeljebb egy, nem az F-hez tartozó folyamatra vonatkozóan különböztethető meg.*

Ezt a rossz együttest mutatja a 25.2. ábra.

Bizonyítás. Az ellentmondás kedvéért tegyük fel, hogy létezik ilyen α_0 és α_1. Megkonstruáljuk az α_0 és α_1 ugyanazon döntéshez, mondjuk a 0-hoz vezető β_0 illetve β_1 lassú végrehajtásait. Ezután felgyorsítjuk β_1-et és eltávolítjuk a hibák egy részét, hogy olyan β_1' *fff*-kiterjesztést kapjunk, amely továbbra is 0-t dönt. Ezzel ellentmondásra jutunk α_1 egyértékűségével.

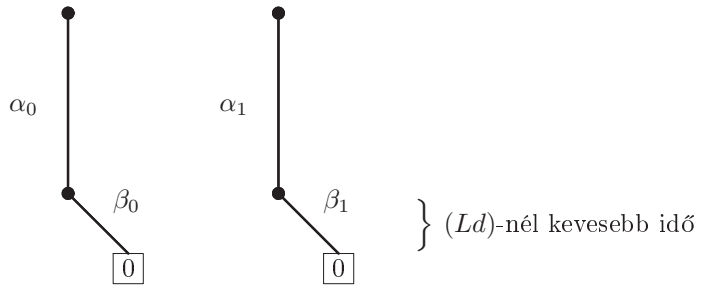
Kicsit részletesebben jelölje G azt a halmazt, amelyet F-ből az α_0 és α_1 időzített végrehajtási sorozatokat megkülönböztető folyamatnak a hozzávételével kapunk (ha egyáltalán van ilyen). Ekkor $|G| \leq f$. Az α_0 és α_1 két lassú végrehajtási sorozatát, β_0-t és β_1-et a következő módon állítjuk elő.



25.2.. ábra. Az időzített végrehajtási sorozatoknak a 25.18. lemmában vizsgált rossz együttese.

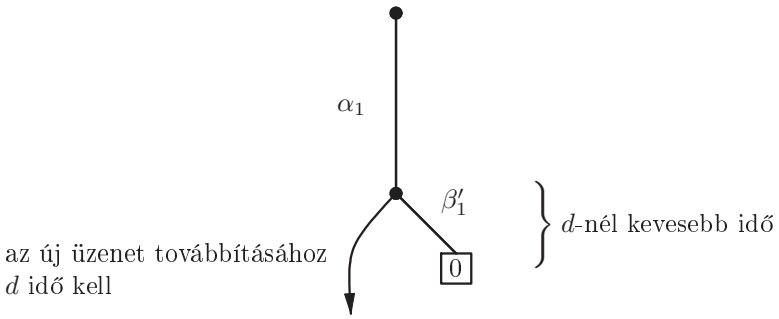
Először a t_k időpontban minden olyan G-beli folyamathoz, amely még nem hibázott, `megállít` eseményeket juttatunk. Ezután α_0-t és α_1-et azonos módon kiterjesztjük további hibákat nem tartalmazó lassú végrehajtásokkal. Az azonos módon való kiterjesztés azért lehetséges, mert α_0 és α_1 a G-hez tartozók kivételével minden más folyamat számára megkülönböztethetetlen. Az A-ra feltételezett felső korlát miatt β_0 és β_1 minden folyamatának még szigorúan az $Ld + (f-1)d \leq t_k + Ld$ időpont előtt dönteni kell. Vagyis a két időzített végrehajtási sorozat döntés előtti új részében Ld-nél határozottabban kevesebb idő múlik el. Továbbá β_0-ban és β_1-ben ugyanarra a döntésre jutunk, mivel α_0-t és α_1-et ugyanúgy terjesztettük ki. Az általánosság megszorítása nélkül tegyük fel, hogy ez a közös érték a 0 (lásd a 25.3. ábrát).

Tekintsük most az α_1 olyan alternatív szinkron időzített β_1' végrehajtási sorozattá való kiterjesztését, amely β_1-től eltérően az α_1 *fff*-kiterjesztése lesz. A β_1' időzített végrehajtási sorozat ugyanaz, mint β_1, csak az α_1 utáni rész az L



25.3.. ábra. A 25.18. lemma bizonyításában szereplő β_0 és β_1 kiterjesztések.

tényezővel „fel van gyorsítva”, hogy *gyors* legyen. Továbbá β_1'-ben α_1 után már egyetlen folyamat sem hibázik, viszont a β_1' α_1 utáni részében az összes G-hez tartozó folyamat által küldött üzenet kézbesítése a maximális d időt igényli. Tehát a $t_k + d$ időpont előtt β_1' pontosan úgy viselkedik, mint a β_1 felgyorsított változata. (Figyeljük meg, hogy β_1-ben és β_1'-ben egészen másként nézhetnek ki a dolgok a G-hez tartozó folyamatok által küldött üzenetek megérkezése után, de ez nem zavar.) Továbbá mivel az összes nem G-beli folyamat β_1-ben szigorúan a $t_k + Ld$ időpont előtt 0-t dönt, ezek β_1'-ben is 0-t döntenek, mégpedig szigorúan a $t_k + d$ időpont előtt, lásd a 25.4. ábrát.



25.4.. ábra. A 25.18. lemma bizonyításában szereplő β_1' kiterjesztés.

De mivel β_1' az α_1 *fff*-kiterjesztése, ez ellentmond az α_1 1-értékűségének. □

A tételben kimondott állítással annak bizonyításával jutunk ellentmondásra, hogy ténylegesen elő kell fordulni az időzített végrehajtási sorozatok 25.18. lemmában leírt formájú rossz együttesének. Először egy rokon együttest kapunk.

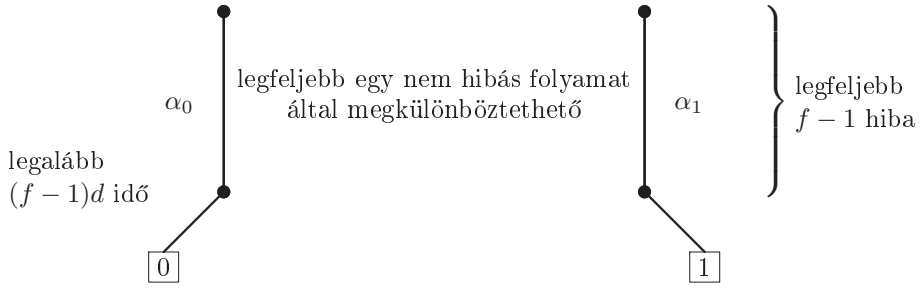
25.19. lemma. . *Bizonyos k-ra létezik két olyan k-blokk szinkron időzített végrehajtási sorozat, α_0 és α_1, melyekre az alábbiak teljesülnek:*

1. *α_0 és α_1 aktív időjei, $t_1, \ldots, t_k$ megegyeznek és $t_k \geq (f-1)d$;*
2. *α_0-ból fff-elérhető a 0;*

3. α_1*-ből fff-elérhető az* 1*;*
4. *Az* α_0*-ban vagy* α_1*-ben hibázó folyamatok* F *halmazának számosságára* $|F| \leq f - 1$ *teljesül,*
5. α_0 *és* α_1 *legfeljebb egy, nem az* F*-hez tartozó folyamatra vonatkozóan különböztethető meg.*

Figyeljük meg, hogy a rossz együttesben és az előző feltételekben csak annyi eltérés van, hogy a 2. és a 3. feltétel csupán a 0 és az 1 *fff*-elérhetőségét kívánja meg α_0-tól és α_1-től a 0-értékűség, illetve az 1-értékűség helyett (lásd a 25.5. ábrát).

Bizonyításvázlat. A 6.33. tétel bizonyításában használt láncérveléshez hasonlóval dolgozhatunk. A bizonyítást meghagyjuk gyakorlatnak (lásd a 25-10. gyakorlatot). □



25.5.. ábra. A 25.19. lemmában szereplő α_0 és α_1 időzített végrehajtási sorozatok.

A 25.18. és a 25.19. lemma összekapcsolásával közvetlenül adódik a

25.20. lemma. . *Létezik az összes alábbi feltételt teljesítő* α *véges szinkron időzített végrehajtási sorozat:*

1. α *utolsó aktív időpontja,* t_k*, legalább* $(f-1)d$*;*
2. α *kétértékű;*
3. α*-ban legfeljebb* $f - 1$ *folyamat hibázik.*

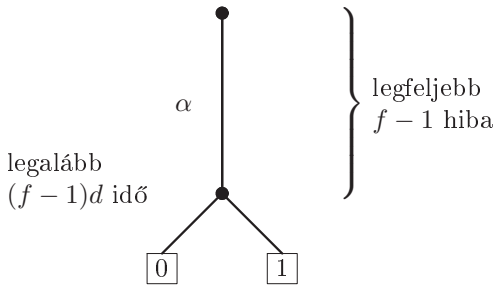
Lásd a 25.6. ábrát.

Bizonyítás. Legyen α_0 és α_1 két olyan szinkron időzített végrehajtási sorozat, melyek létezését a 25.19. lemma biztosítja. A 25.18. lemma alapján nem lehet α_0 0-értékű és ugyanekkor α_1 1-értékű. Tehát α_0 és α_1 közül legalább az egyik kétértékű és teljesíti a kívánt feltételt. □

A továbbiakban a 25.20. lemmát egy „maximalitási” tulajdonság hozzávételével erősítjük.

25.21. lemma. . *Létezik az összes alábbi feltételt teljesítő* α *véges szinkron időzített végrehajtási sorozat:*

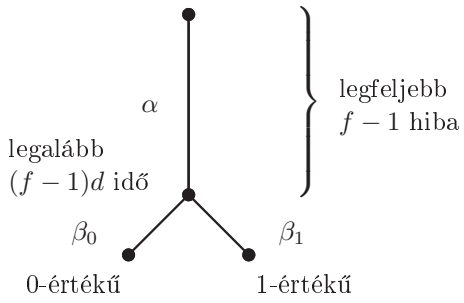
1. α *utolsó aktív időpontja,* t_k*, legalább* $(f-1)d$*;*



25.6.. ábra. A 25.20. lemmában szereplő α időzített végrehajtási sorozat.

2. *α kétértékű;*
3. *α-ban legfeljebb $f-1$ folyamat hibázik;*
4. *megadható α-nak két olyan egyetlen blokkal való kiterjesztése, β_0 és β_1, melyekre az alábbiak igazak:*
   (a) *β_0 0-értékű;*
   (b) *β_1 1-értékű;*
   (c) *β_0 és β_1 legfeljebb egy folyamat által különböztethető meg.*

Lásd a 25.7. ábrát. (Figyeljük meg a hasonlóságot a 12.6. tétel bizonyításában használt *döntnök* és azon konfiguráció között, melynek létezését az előbb mondtuk ki.)



25.7.. ábra. A 25.21. lemma α_0,β_0 és β_1 időzített végrehajtási sorozatai.

Bizonyításvázlat. Legyen α olyan véges szinkron időzített végrehajtási sorozat, melynek létezését a 25.20. lemmában állítottuk. Terjesszük ki α-t a következő „program” végrehajtási sorozatával:

while létezik α-nak valódi kétértékű *fff*-kiterjesztése **do**
  $\alpha :=$ „egy ilyen kiterjesztés”

Tudjuk, hogy ez a program előbb-utóbb befejeződik, mivel a döntések az α bármely hibamentes kiterjesztésében megelőzik az $Ld+(f-1)d$ időpontot. Tekintsük a program végrehajtási sorozatával kapott utolsó α-t.

Az állítjuk, hogy α rendelkezik az összes szükséges tulajdonsággal. Kielégíti a kívánt időkorlátra, a kétértékűségre és a hibára vonatkozó feltételeket. Továbbá mivel α kétértékű, de nem terjeszthető ki hosszabb kétértékű időzített végrehajtási sorozattá, létezni kell α két olyan egy blokkos *fff*-kiterjesztésének, γ_0-nak és γ_1-nek, melyekre

1. γ_0 0-értékű.
2. γ_1 1-értékű.

Ez még nem pontosan a várt eredmény, mivel lehetséges, hogy γ_0 és γ_1 egynél több folyamat számára is megkülönböztethető.

A kívánt β_0 és β_1 előállításához végezzünk még egy lánc-konstrukciót γ_0-n és γ_1-en. A γ_0-ból kiindulva egyszerűen módosítsuk úgy a lánc minden lépésénél az egyik P_i folyamatnak kézbesített üzeneteket, hogy megegyezzenek a γ_1-ben találhatókkal. A lánc bármely két egymást követő időzített végrehajtási sorozata legfeljebb egy folyamat számára különböztethető meg. Mivel az összes ilyen időzített végrehajtási sorozatnak egyértékűnek kell lenni, léteznie kell a láncban két egymás utáni olyan β_0 és β_1 végrehajtási sorozatnak, hogy β_0 0-értékű és β_1 1-értékű. Ezzel megkaptuk az összes kívánt tulajdonságot. □

Most már ellentmondásra juthatunk.

25.22. lemma. . *Az A algoritmus nem létezik.*

Bizonyítás. A β_0 és β_1 időzített végrehajtási sorozatok, melyek létezését a 25.21. lemma biztosítja, teljesítik a rossz együttesre a 25.18. lemmában felsorolt feltételeket. Ez pedig ellentmondás. □

Ezzel beláttuk a 25.17. tételt.

25.6.. További eredmények*

Ebben a fejezetben azt vizsgáljuk, hogyan változnak az egyetértési problémára vonatkozó eredmények az időzítési modell különböző gyengítései esetében. Megelégszünk az informális tárgyalással.

25.6.1.. Szinkron folyamatok aszinkron csatornákkal*

Tegyük fel, hogy modellünket úgy gyengítjük, hogy a 14. fejezetben definiált, az üzenetkézbesítésre felső korlátot nem tartalmazó, csak az előbb-utóbb megtörténő kézbesítést biztosító megbízható FIFO csatornákat használunk. A folyamatokra azonban továbbra is az $[\ell_1, \ell_2]$ korlátok vonatkoznak. Ebben az esetben nem nehéz belátni, hogy még egyetlen megállási hiba esetében sem oldható meg a megegyezési feladat. Ez még akkor is igaz, ha $\ell_1 = \ell_2$, vagyis a folyamatok lépésideje pontosan megjósolható.

25.23. tétel. . *A szinkron folyamatokból és aszinkron csatornákból álló modellben nem létezik a megegyezési problémát megoldó és az* 1*-hibás befejezést garantáló algoritmus.*

Bizonyításvázlat. Vegyük azt az indirekt feltevést, hogy létezik a kívánt A algoritmus. Ekkor A „szimulálható" az aszinkron modellben a 18. fejezetben bevezetett logikai idő implementálásával. Ilymódon az aszinkron hálózati modellben előállítható 1-hibás befejezést garantáló megegyezési algoritmus, ami ellentmond a 21.2. tételnek. A részleteket meghagyjuk gyakorlatnak (lásd a 25-14. gyakorlatot). □

25.6.2.. Aszinkron folyamatok szinkron csatornákkal*

Tételezzük fel, hogy a részben szinkron modellt az üzenettovábbításra vonatkozó d korlát megtartásával most úgy gyengítjük, hogy a folyamatoktól csak a méltányosságot követeljük meg, időkorlátokat nem írunk elő. Ismét könnyen igazolható, hogy a megegyezési feladat még egyetlen megállási hiba esetében sem oldható meg.

25.24. tétel. . *Az aszinkron folyamatokból és d-korlátos csatornákból álló modellben nem létezik a megegyezési problémát megoldó és az* 1*-hibás befejezést garantáló algoritmus.*

Bizonyítás. Vegyük azt az indirekt feltevést, hogy létezik a kívánt A algoritmus. Futtassuk ugyanezt az A algoritmust az aszinkron modellben. Ekkor A-nak az aszinkron modellben vett bármely α pártatlan végrehajtási sorozata „időzíthető" úgy, hogy az összes üzenetre teljesüljön a d felső korlát. Ez azt jelenti, hogy a végrehajtási sorozat kielégíti a megegyezési feladat 1-hibás befejezésére vonatkozó összes feltételt. Azonban a feltételek egyike sem függ az időtől, tehát ugyanez igaz a tekintett α pártatlan végrehajtási sorozatra. Mivel a fenti technika alkalmazható A bármely α pártatlan végrehajtási sorozatára, A az aszinkron modellben 1-hibás befejezéssel megoldja a megegyezési problémát, ami újfent ellentmond a 21.2. tételnek. □

25.6.3.. Végső időkorlátok*

A könyvben tárgyalt utolsó eredményhez tekintsük a 24.4.2. szakasszal megegyező végső időkorlátos esetet. Vagyis azt a modellt vesszük, melyben az algoritmus egy darabig aszinkron fut, de *valahonnan kezdve* megfelel az időkorlátoknak. Kiderül, hogy ebben a modellben megoldható a megegyezési feladat. Azonban a részben szinkron esettől eltérően (ahol az időkorlátok mindig teljesültek) a megoldáshoz $n \geq 2f$ szükséges. A 17.6. tétel bizonyításához hasonló érveléssel nem nehéz belátni, hogy a feladat ebben a modellben az $n \leq 2f$ esetben nem oldható meg. A részleteket meghagyjuk a 25-15. gyakorlatra.

25.25. tétel. . *A megegyezési feladat f-hibás befejezéssel $n \leq 2f$ esetében megoldható, ha a modellben a folyamatok taszkjaira végső fokon az $[\ell_1, \ell_2]$ időkorlát, az összes üzenetre pedig a d korlát teljesül.*

A feladat megoldásának megtervezése ebben a modellben nem olyan könnyű. A PSzinkMegegyezés protokollnál használthoz hasonló stratégiák, melyek a hibás folyamatok időtúllépésére alapoztak, most nem működnek. Mielőtt az időkorlátok érvényesülni kezdenek, egyes folyamatok tévesen azt hihetik, hogy más folyamatok hibáztak. Más stratégián alapuló algoritmust vázolunk fel.

Az algoritmus lelke a 2. fejezet *szinkron* modelljének egy változatára készült A algoritmus, melyben a legfeljebb f hibán túl az üzenetek egy része is elveszhet. Feltesszük, hogy bármelyik üzenet elveszhet, de üzenetvesztés csak véges sok menet során fordulhat elő. Egy bizonyos időpont után már minden üzenet biztosan kézbesítődik. A folyamatok azonban nem tudják, hogy ez mikor következik be.

Az A algoritmus a következő módon dolgozik. Ismét feltételezzük, hogy a folyamatok mind maguknak, mind a többi folyamatnak küldenek üzeneteket.

Az *A* algoritmus

A folyamatok az $1, 2, \ldots$ „szintekre" oszlanak. Az s szint a $4s-3, 4s-2, 4s-1, 4s$ négy egymás utáni menetből áll. Az s szint „tulajdonosa" a $tulajdonos(s)$ folyamat; ez azt a folyamatot jelenti, amelynek indexe modulo n ekvivalens s-sel.

A folyamatok különböző időpontokban *zárolhatnak* egy $v \in \{0, 1\}$ értéket a hozzá tartozó s szintszámmal együtt. Ha a P_i folyamat zárolja (v, s)-t, ez azt jelenti, hogy szerinte a $tulajdonos(s)$ folyamat az s szinten dönthet v-t. A P_i folyamat fenntarthat v-re vonatkozóan valamilyen zárolást egészen addig, míg azt hiszi, hogy $tulajdonos(s)$ dönthet v-t az s szinten. Egy v érték P_i számára *elfogadható*, ha nincs $\bar{v}$-re vonatkozó zárolása. Kezdetben nincsenek zárolt értékek.

Ha az s-dik menet tulajdonosa a P_i, az s-dik menet során a feldolgozás a következő módon történik.

A $4s-3$ *menet*: Az összes folyamat megküldi P_i-nek az elfogadható értékeit. Ezután P_i megpróbál egy *javasolt* értéket választani. P_i a v értéket akkor javasolhatja, ha arról értesült, hogy legalább $n-f$ folyamat (saját magát is beleértve) az s-dik szinten elfogadhatónak tartja v-t. Lehet több olyan érték is, amit P_i javasolhat, ilyenkor a saját kezdeti értékét választja.

A $4s-2$ *menet*: Ha P_i eldöntötte, hogy milyen v-t javasoljon, akkor minden folyamatnak küld egy $(\texttt{zárol}, v)$ üzenetet. Ha egy folyamat ilyen üzenetet kap, akkor zárolja (v, s)-t, és felenged minden ugyanezen v-re vonatkozó korábbi zárolást.

A $4s-1$ *menet*: Minden olyan folyamat, amely a $(4s-2)$-dik menetben $(\texttt{zárol}, v)$ üzenetet kapott, `nyugtáz` üzenetet küld P_i-nek. Ha P_i legalább $f+1$ folyamattól kap ilyen `nyugtáz` üzenetet, akkor a javasolt v érték mellett dönt.

A $4s$ *menet*: Minden folyamat olyan üzeneteket küld az összes többinek, amely tartalmazza az ő pillanatnyilag érvényes minden zárolását. Ha a P_i folyamat zárolta (v, s')-t, de olyan $(\bar{v}, s'')$ üzenetet kapott, melyben $s'' > s'$

(vagyis az ellentétes értékre vonatkozó újabb zárolást), akkor felengedi a korábbi zárolást.

25.26. lemma. . *Az A algoritmus $n > 2f$ esetében megoldja a megegyezési problémát és biztosítja az f-hibás befejeződést.*

Bizonyítás. Először a következő segédtételt mondjuk ki.

25.27. segédtétel. . *Minden s szinten legfeljebb egy javasolt v érték van, s így legföljebb egy v létezik, mellyel valamelyik folyamatnak valaha is (v, s) zárolása lehet.*

Ezután a menetek száma szerinti egyszerű indukcióval a következő segédtételt kapjuk (felhasználva, hogy a folyamatok előnyben részesítik kezdőértéküket):

25.28. segédtétel. . *Ha minden folyamat kezdőértéke v, akkor $\bar{v}$ soha nem lesz sem javasolt, sem zárolt.*

Mivel a folyamatok csak az általuk javasolt érték mellett dönthetnek, az érvényesség teljesülni fog. Ezután megmutatjuk az alábbi segédtételt.

25.29. segédtétel. . *Ha a P_i folyamat az s-dik szinten v mellett dönt, akkor minden ezutáni menet végén legalább $f+1$ folyamatnak lesz olyan v-re vonatkozó zárolása, melyhez tartozó szintindex nagyobb vagy egyenlő, mint s.*

Bizonyítás. Az algoritmus biztosítja, hogy a $(4s-2)$-dik menet végén legalább $f+1$ folyamat zárolja (v, s)-t. Azt állítjuk, hogy ezen folyamatok közül soha egyik sem engedi fel a v zárolását anélkül, hogy azonnal ne szerezne újabb zárolást ugyanerre a v-re.

Az ellentmondás kedvéért tegyük fel, hogy az egyik ilyen folyamat, mondjuk P_i, felengedi v zárolását anélkül, hogy azonnal újra zárolná v-t. P_i-nek azért kellett felengednie a zárolást, mert valamely $s' > s$-re tudomást szerzett egy $(\bar{v}, s')$ zárolásról. Ez azt jelenti, hogy *tulajdonos*(s') az s'-dik szinten a $\bar{v}$ értéket javasolja. Legyen s' az első olyan s utáni szint, melyen $\bar{v}$-t javasolják.

Ekkor viszont az s'-t közvetlenül megelőző szinten legalább $f+1$ v-re vonatkozó zárolásnak kell lennie, ami meggátolja, hogy *tulajdonos*(s') a $(4s'-3)$-dik menetben a szükséges számú, legalább $n-f$ folyamat $\bar{v}$-hez való hozzájárulását megkapja. Ezzel ellentmondásra jutottunk. □

Térjünk vissza 25.26. lemma bizonyításához. Most a megegyezést igazoljuk. Tegyük fel, hogy a P_i folyamat az s-dik szinten v mellett dönt. Ekkor egyetlen másik folyamat sem választhatja $\bar{v}$-t ezen a szinten. Továbbá a 25.29. segédállítás szerint az s-dik szinttől kezdve mindig legalább $f+1$ v-re vonatkozó zárolás lesz. Ez lehetetlenné teszi, hogy bármelyik folyamat megszerezze a $\bar{v}$ javasolásához szükséges $n-f$ folyamat hozzájárulását. Tehát soha nem javasolják $\bar{v}$-t, és egyik folyamat sem választja $\bar{v}$-t.

A befejeződés igazolásához nézzük meg, mi történik azon feltételezett időpont elérése után, amikortól minden üzenet megbízhatóan kézbesítődik. Minden ezt követő s szintre könnyen belátható, hogy a rendszer összes nemhibázó folyamatát

tekintve legfeljebb egy zárolt érték lehet. Ez a 25.27. segédtétel és a $4s$-dik szint felengedési szabálya miatt igaz. Ha viszont ez a helyzet, akkor bármelyik szint tulajdonosa meg tudja szerezni a döntésének megengedéséhez szükséges összes hozzájárulást és nyugtázást (ha nem hibázik).

□

25.25. tétel bizonyításvázlata. A végső időkorlátos modellre vonatkozó B algoritmus megkonstruálásának csak alapötletét ismertetjük. A B minden P_i folyamatának van egy nemnegatív egész értékű lokális *óra* változója, melynek kezdőértéke 0. Minden *óra* változó értéke monoton nemcsökkenő. Legyen $C = \max\{\textit{óra}_i : 1 \leq i \leq n\}$. C tekinthető a rendszer által fenntartott egyfajta „globális órának". Az *óra* változók értékének közlésére és beállítására vonatkozó protokoll segítségével (ennek tárgyalását elhagyjuk) a folyamatok elérhetik, hogy attól a p időponttól kezdve, amikor az időkorlátok teljesülnek,

1. C növekedésének a valós időhöz mért aránya mind alulról, mind felülről ismert konstansokkal korlátozható;
2. Minden *óra* változó értéke legfeljebb egy (ismert) additív konstanssal lehet több C-nél.

A folyamatok *óra* változói így végül is elég jól szinkronizáltak lesznek.

B minden P_i folyamata az *óra* változó kezelésén túl szimulálja az A algoritmus ráeső részét, miközben lokális órájának értéke alapján dönti el, hogy melyik menetet szimulálja. Minden menet szimulációjára elég sok, de előre megjósolható számú óraértéket szánunk. Elegendően sokat ahhoz, hogy az r-dik menet szimulációjakor a p időpont elérése után bármely P_i által a menet szimulációjának kezdetén elküldött üzenetek eljussanak minden P_j-hez, mielőtt még P_j befejezné az r-dik menet szimulációját.

Figyeljük meg, hogy a p időpont előtt megtörténhet, hogy valamelyik P_i nem bírja befejezni az r-dik menet szimulációját mielőtt még az *óra* változója túl messze előrelépne. Ebből semmi baj sem származik, ha P_i egyszerűen nem küldi el a hátralévő további üzeneteket; végül is A-ban ezek akár el is veszhetnének. Ám P_i-nek végre kell hajtani az r-dik menet állapot-átmeneteinek szimulációját. Ezt megteheti az r-dik menet szimulációjának félbeszakítását követő első lépése során.

Ilymódon B szimulálja az A algoritmust, és ugyanazokat a helyességi feltételeket teljesíti. □

25.7.. Utóirat

A mostani és az előző fejezetben néhány egyszerű eredményt ismertettünk a részben szinkron modell keretein belül az osztott számítások két alapvető problémájára, a kölcsönös kizárásra és a megegyezésre. Már ez a néhány eredmény is bizonyítja, hogy a részben szinkron osztott számítások elmélete teljesen különböző akár a szinkron, akár az aszinkron osztott számításokétól.

Ám még sok a tennivaló ezen a területen. Az osztott számítások sok érdeklődésre számottartó egyéb problémája is vizsgálható a részben szinkron esetben.

Közéjük tartozik a könyvben ismertetett sok feladat, így például a hálózati keresés, a feszítőfák konstrukciója, az erőforrás-hozzárendelés, a fényképek és a stabilitási tulajdonság érzékelése. Sok más olyan problémát is tartalmaznak, melyek a valóságos kommunikációs rendszerekből, az osztott operációs rendszerekből és a valós idejű folyamatvezérlő rendszerekből származnak.

Hasznosak lennének az olyan általános karakterizációs eredmények is, melyek pontosan megmondanák, hogy a részben szinkron rendszerekben mi, mekkora időbonyolultsággal számítható ki. Jók lennének az olyan transzformációs eredmények is, melyek a részben szinkron modellek képességeit hasonlítanák össze a szinkron és az aszinkron modellekével.

25.8.. Megjegyzések a fejezethez

A fejezet konstrukcióinak és eredményeinek többségét, beleértve a PSzinkHJ hibaérzékelőt, a 25.3. és a 25.6. tétel egyszerű felső és alsó korlátokra vonatkozó eredményeit, továbbá a 25.15. és a 25.17. tétel bonyolultabb felső és alsó korlátait Attiya, Dwork és Lynch [22] cikkében bizonyították. Ponzio a [247,245]-ben a 25.4.1. szakasz eredményeit terjesztette ki az erősebb „sending-omission" hibamodellre, és megadott egy kevésbé hatékony algoritmust bizánci hibák esetére. Berman és Bharali [48]-ban Ponzio „sending-omission" algoritmusának bonyolultsági korlátján javítottak. Ponzio jó alsó felső korlátokat is kapott a hibaérzékelés bonyolultságára két csomópontos rendszerek esetében [246]-ban. A szinkron folyamatokra és aszinkron csatornákra vonatkozó megoldhatatlansági eredményt, a 25.23. tételt először Dolev, Dwork és Stockmeyer bizonyította [95]-ben. Az itt vázolt, a WelchIdő-n alapuló bizonyítást Welch írta le [287]-ben. A végső időkorlátos modellre vonatkozó 25.25. tételt Dwork, Lynch, és Stockmeyer bizonyította [104]-ben. Ez a cikk – más hibamodellekre vonatkozók mellett – tartalmaz az ismeretlen időkorlátos esetre is egy hasonló eredményt. Lamport Paxos algoritmusa [183] nagyon hasonlít a [104] algoritmusaihoz.

A parciálisan szinkron modellre vonatkozó egyéb eredmények közé tartoznak Attiya és Mavronicolas [17] alsó és felső korlátai a 16.6. alfejezet „session" problémájának megoldási idejére; Wang és Zuck [284] korlátai a megbízható magasszintű üzenetátvitelhez szükséges alacsony szintű üzenet-ábécé méretére vonatkozóan; továbbá Kleinberg, Attiya és Lynch [167] cikke a kapcsolatkezelési protokollok üzenetkézbesítési és nyugalmi idejére vonatkozó alsó és felső korlátok összefüggéseiről.

25.9.. Gyakorlatok

25-1. Írjunk előfeltétel/hatékony kódot a PSzinkHJ algoritmus P_i folyamata számára.

25-2. Az összes üzenet d időn belüli kézbesítését biztosító csatornák helyett tegyük fel, hogy olyan csatornákat használunk, amelyek csak a legrégebbi üzenet d időn belüli kézbesítését garantálják.

(a) Módosítsuk a PSzinkHJ algoritmust úgy ennek a modellnek megfelelően, hogy lehetőleg minimalizáljuk a keletkező időbonyolultságot.
(b) Bizonyítsunk alsó korlátot a fenti helyzetnek megfelelő hibajelzők időbonyolultságára.

25-3. *Kutatási téma.* Készítsük el a részben szinkron modellben a lehető leghatékonyabb algoritmust a megállási hibákat tartalmazó szinkron algoritmusok szimulációjára. El tudjuk-e érni az r menet szimulációjához szükséges időre az $Ld + rd$ (plusz alacsonyabb rendű tagok) felső korlátot?

25-4. A részben szinkron hálózatok megyezési problémája megoldható a következő alternatív stratégiával abban a speciális esetben, ha az összes bemenet a 0 időpontban érkezik.

A folyamatok a 6.2.3. szakasz EIGYStop algoritmusát szimulálják az ott leírt módon a megkapott információk beérkezés utáni azonnali továbbításával és az értékek saját *EIGY* fáikba történő feljegyzésével. Minden folyamatnak tudnia kell, hogy ő befejezte az értékek beírását saját fájába. Erről annak biztosításával gondoskodhat, hogy legalább $(f + 1)(d + l)$ idő teljen el.

Adjuk meg egy ilyen algoritmus részletes kódját, bizonyítsuk be helyes működését és elemezzük időbonyolultságát.

25-5. Adjuk meg a 6.33. tétel megfelelőjét a 25.6. tétel bizonyításában bevezetett és használt általános szinkron modellre vonatkozóan. (*Útmutatás.* A bizonyítás nagyon hasonló a 6.33. tételhez.)

25-6. Egészítsük ki a 25.15. tétel bizonyításának hiányzó részleteit: mutassuk meg, hogy a 0. menet ideje, $T(0) - T'$, $\mathcal{O}(\ell_2)$, továbbá $k \geq 1$ esetében a k-dik menet ideje, $T(k) - T(k-1)$ legfeljebb $S + \mathcal{O}(\ell_2)$.

25-7. Adjuk meg tetszőleges f-re ($0 \leq f \leq n$) a PSzinkMegegyezés algoritmus olyan speciális megengedett időzítésű végrehajtási sorozatát, amely során minden kapura érkezik bemenet, legfeljebb f hiba fordul elő, és a lehető leghosszabb idő telik el, míg minden folyamat vagy hibázik, vagy dönt.

25-8. Az összes üzenet d időn belüli kézbesítését biztosító csatornák helyett tegyük fel, hogy olyan csatornákat használunk, amelyek csak a legrégebbi üzenet d időn belüli kézbesítését garantálják. Módosítsuk a PSzinkMegegyezés algoritmust úgy ennek a modellnek megfelelően, hogy lehetőleg minimalizáljuk a keletkező időbonyolultságot.

25-9. *Kutatási téma.* Készítsünk a részben szinkron modellre a PSzinkMegegyezés-nél hatékonyabb algoritmust. El tudjuk-e érni az időre vonatkozó $Ld + fd$ (plusz alacsonyabb rendű tagok) felső korlátot?

25-10. Bizonyítsuk be a 25.19. lemmát. (*Útmutatás.* Használjunk a 6.33. tétel bizonyításához hasonló láncérvelést. Ezt alapozzuk a szinkron időzített végrehajtási sorozatok azon részhalmazára, melyek minden $r \in \mathbb{N}$-re megfelelnek az időkorlátoknak, az összes üzenetüket az $[rd, (r + 1)d)$ intervallumba eső időpontokban küldik el, és ezek az üzenetek pontosan az $(r + 1)d$ időpontban érnek célba. Ezen részhalmaz időzített végrehajtási sorozatai hasonlóak a szinkron modelléhez, és ugyanolyan láncérvelés alkalmazható rájuk.)

25-11. *Kutatási téma.* Bizonyítsunk a 25.17. tételénél jobb alsó korlátot a megegyezés elérési idejére a részben szinkron modellben. El tudjuk érni az $Ld + fd$ korlátot? Esetleg jobbat is?

25-12. *Kutatási téma.* Határozzuk meg a részben szinkron modellben a bizánci megegyezési feladat lehető legjobb alsó és felső korlátjait.

25-13. *Kutatási téma.* Tekintsük a 21.5. alfejezetben definiált k-megegyezési problémát az f megállási hibás részben szinkron hálózati modellben. Határozzunk meg jó alsó és felső korlátokat az összes nemhibázó folyamat döntéséhez szükséges időre. El tudjuk-e közelítőleg érni az $Ld + \frac{f}{k}d$ korlátokat? (Ezt sugalmazta a MinTerjed algoritmus és a szinkron hálózati elrendezésre vonatkozó 7.14. tétel.)

25-14. Bizonyítsuk be a 25.23. tételt, a szinkron folyamatokból és aszinkron csatornákból álló küld/fogad hálózatok egyetértési problémájára vonatkozó lehetetlenségi eredményt. (*Útmutatás.* Mutassuk meg, hogyan szimulálható az aszinkron modellben a logikai idő WelchIdő algoritmus szerinti implementációjával ezen új modell valamely algoritmusa.)

25-15. Bizonyítsuk be, hogy a végső időkorlátos modellben $n \leq 2f$ esetében nem oldható meg a megegyezési feladat.

25-16. Egészítsük ki a 25.25. tétel bizonyítását, tehát

(a) definiáljuk pontosan az *óra* változó kezelését;

(b) fogalmazzuk meg pontosan a szükséges állításokat a szinkronizáció fokára és a növekedés arányára vonatkozóan;

(c) egészítsük ki a B algoritmus ismertetését az órán alapuló szimuláció leírásával;

(d) bizonyítsuk be, hogy B f-hibás befejezéssel biztosítja a megegyezési problémára vonatkozó helyességi feltételek teljesülését.

25-17. Elemezzük a 25-16. gyakorlatban konstruált B algoritmus időbonyolultságát.

25-18. Vizsgáljuk a megegyezési feladat megoldhatóságát az *ismeretlen időkorlátos* modellben. Ebben a modellben feltételezzük a folyamatok lépésidejére vonatkozó ℓ_1 alsó és ℓ_2 felső időkorlátok ($0 < \ell_1 \leq \ell_2 < \infty$), továbbá az összes üzenet kézbesítési idejére vonatkozó d felső időkorlát létezését, de ezek a korlátok „ismeretlenek" a folyamatok számára. (Vagyis különböző végrehajtási sorozatok során eltérőek is lehetnek, bár minden végrehajtási sorozat során rögzített korlátok érvényesek.)
Bizonyítsuk be a 25.25. tétel megfelelőjét az ismeretlen időkorlátos esetre.

25-19. *Kutatási téma.* Készitsük el a PSzinkHJ és a PSzinkMegegyezés algoritmus időkorlátjaira vonatkozó eredmények bizonyítását a 23.3.3. szakasz szimulációs módszereivel.

25-20. Dolgozzunk ki a részben szinkron hálózati modellben jó alsó és felső korlátokat a 16.6. alfejezet „session" problémájára.

25-21. Dolgozzunk ki a részben szinkron közös memóriájú modellben jó alsó és felső korlátokat a 13.3. alfejezetnek megfelelően meghatározott fénykép atomi objektum megvalósítási problémájának időbonyolultságára. (Irjuk le pontosan, hogy mit is mérünk.)

25-22. *Kutatási téma.* Dolgozzunk ki jó alsó és felső korlátokat az osztott számítások más érdeklődésre számot tartó problémáinak időbonyolultságára a részben szinkron esetben. A könyvben említetteken túl vizsgáljuk a valóságos kommunikációs rendszerek, az osztott operációs rendszerek és a valós idejű folyamatvezérlő rendszerek területén felmerülő más problémákat is. Nézhetünk a részben szinkron elrendezés könyvünkben használt speciális megfogalmazásán túlmutató eseteket is.

25-23. *Kutatási téma.* Dolgozzunk ki olyan általános karakterizációs tételeket, melyek megmutatják, hogy pontosan mi is számítható ki a részben szinkron rendszerekben, megadják az ehhez tartozó időbonyolultságokat, valamint olyan transzformációs eredményeket, melyek a részben szinkron rendszerek képességeit hasonlítják össze a szinkron és az aszinkron modellekkel.

Irodalomjegyzék

[1] M. Abadi, L. Lamport: An old-fashioned recipe for real time. In J. W. de Bakker et al. (szerk.), *Real-Time: Theory in Practice* (REX Workshop, Mook, 1991), *LNCS*, 600, p. 1-27. Springer-Verlag, New York, 1992.

[2] K. Abrahamson: On achieving consensus using a shared memory. In *Proc. 7th Ann. ACM Symp. Princ. Distr. Comp.*, p. 291-302, Toronto, 1988.

[3] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 41(4):873-890, 1993.

[4] Y. Afek, H. Attiya, A. D. Fekete, M. Fischer, N. A. Lynch, Y. Mansour, D. Wang, L. Zuck. Reliable communication over unreliable channels. *J. ACM*, 41(6):1267-1297, 1994.

[5] Y. Afek, E. Gafni: End-to-end communication in unreliable networks. In *Proc. 7th Ann. ACM Symp. Princ. Distr. Comp.*, p. 131-148, Toronto, 1988. ACM, New York.

[6] Y. Afek, E. Gafni: Time and message bounds for election in synchronous and asynchronous complete networks. *SIAM* J. Comp., 20(2):376-394, 1991.

[7] G. A. Agha: *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, 1986.

[8] B. Alpern, F. B. Schneider: Defining liveness. *Inf. Process. Letters*, 21(4):181-185, 1985.

[9] R. Alur, D. L. Dill: A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183-235, 1994.

[10] R. Alur, G. Taubenfeld: Results about fast mutual exclusion. In *Proc. Real-Time Syst. Symp.*, p. 12-21, Phoenix. IEEE, Los Alamitos, 1992.

[11] J. H. Anderson: Composite registers. *Distr. Comp.*, 6(3):141-154, 1993.

[12] J. H. Anderson: Multiwriter composite registers. *Distr., Comp.*, 7(4):175-195, 1994.

[13] D. Angluin: Local and global properties in networks of processors. In. *Proc. 12th Ann. ACM Symp. Theory Comp.*, p. 82-93, Los Angeles, 1980.

[14] E. Arjomandi, M. J. Fisher, N. A. Lynch: Efficiency of synchronous versus asynchronous distributed systems. *J. ACM*, 30(3):449-456, 1983.

[15] E. A. Ashcroft: Proving assertions about parallel programs. *J. Comput. Syst. Sci.* 10(1):110-135, 1975.

[16] J. Aspnes, M. Herlihy: Fast randomized consensus using shared memory. *J. Algorithms*, 11(3):441-461, 1990.

[17] H. Attiya, M. Mavronicolas: Efficiency of semisynchronous versus asynchronous networks. *Math. Syst. Theory*, 27(6):547-571, 1994.

[18] H. Attiya, A. Bar-Noy, D. Dolev: Sharing memory robustly in memory-passing systems. *J. ACM*, 42(1):124-142, 1995.

[19] H. Attiya, A. Bar-Noy, D. Dolev, D. Koller, D. Peleg, R. Reischuk: Achievable cases in an asynchronous environment. In *28th Ann. Symp. Found. Comp. Sci.*, p. 337-346. Los Alamitos, 1987.

[20] H. Attiya, A. Bar-Noy, D. Dolev, D. Koller, D. Peleg, R. Reischuk: Renaming in an asynchronous environment. J. ACM, **37**(3):524-548, 1990.

[21] H. Attiya, S. Dolev, J. L. Welch: Connection management without retaining information. In *Proc. 28th Ann. Hawaii Int. Conf. Syst. Sci.*, vol. II. p. 622-631. Wailea, 1995. IEEE, Los Alamitos, 1995.

[22] H. Attiya, C. Dwork, N. A. Lynch, L. Stockmeyer: Bounds on the time to reach agreement in the presence of timing uncertainty. *J. ACM*, 41(1):122-152, 1994.

[23] H. Attiya, M. Fischer, D. Wang, L. Zuck: Reliable communication using reliable channels. Kézirat, 1989.

[24] H. Attiya, N. A. Lynch, N. Shavit: Are wait-free algorithms fast? *J. ACM*, 41(4):725-763, 1994.

[25] H. Attiya, N. A. Lynch: Time bounds for real-time process control in the presence of timing uncertainty. *Inf. Comput.*, 110(1):183-232, 1994.

[26] H. Attiya, O. Rachman: Atomic snapshots in $O(n \log n)$ operations. In *Proc. 12th Ann. ACM Symp. Princ. Distr. Comp.*, p. 29-40, Ithaca, 1993.

[27] H. Attiya, M. Snir, M. K. Warmuth: Computing in an anonymous ring. *J. ACM*, 35(4):845-875, 1988.

[28] H. Attiya, J. L. Welch: Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2):91-122, 1994.

[29] B. Awerbuch: Complexity of network synchronization. *J. ACM*, 32(4):804-823, 1985.

[30] B. Awerbuch: Reducing complexities of the distributed max-flow and breadth-first search algorithms by means of network synchronization. *Networks*, 15(4):425-437, 1985.

[31] B. Awerbuch: Optimal distributed algorithms for minimum weight spanning tree, counting, leader election and related problems. *Proc. 19th Ann. Proc. Symp. Theory Comp.*, p. 230-240. New York, 1993.

[32] B. Awerbuch, B. Berger, L. Cowen, D. Peleg: Near-linear cost sequential and distributed constructions of sparse neighborhood covers. In *34th Annual Symp. Found. Comp. Sci.*, p. 638-647, Palo Alto, 1993. IEEE, Los Alamitos.

[33] B. Awerbuch, R. G. Gallager: Distributed BFS algorithms. In *26th Ann. Symp. Found. Comp. Sci.*, p. 250-256, Palo Alto, IEEE, Portland, 1985.

[34] B. Awerbuch, O. Goldreich, D. Peleg, R. Vainish: A tradeoff between information and communication in broadcast protocols. *J. ACM*, 37(2):238-256, 1990.

[35] B. Awerbuch, D. Peleg: Sparse partitions. In *31st Ann. Symp. on Foundations of Computer Science*, vol. II, p. 503-513, St. Louis, IEEE, Los Alamitos, 1990.

[36] B. Awerbuch, D. Peleg: Routing with polynomial communication-space trade-off. *SIAM J. Discr. Math.*, 5(2):151-162, 1992.

[37] B. Awerbuch, M. Saks: A Dining Philosophers algorithm with polynomial response time. In *31st Annual Symp. Found. Comp. Sci.*, vol. I., p. 65-74, St. Louis, 1990. IEEE, Los Alamitos, 1990.

[38] J. C. M. Baeten, W. P. Weijland: *Process Algebra. Cambridge Tracts in Theoretical Computer Science* 18. Cambridge University Press, Cambridge, U.K., 1990.

[39] A. Bar-Noy, D. Dolev, C. Dwork, H. R. Strong: Shifting gears: Changing algorithms on the fly to expedite Byzantine agreement. In *Proc. 6th Ann. ACM Symp. Princ. Dist. Comp.*, p. 42-51, Vancouver, 1987.

[40] A. Bar-Noy, D. Dolev, D. Koller, D. Peleg: Fault-tolerant critical section management in asynchronous environments. *Inform. Comput.*, 95(1):l-20, 1991.

[41] A. E. Baratz, A. Segall: Reliable link initialization procedures. *IEEE Trans. Commun.*, 36(2):144-152, 1988.

[42] K. A. Bartlett, R. A. Scantlebury, P. T. Wilkinson: A note on reliable full-duplex transmission over half duplex links. *Commun. ACM*, 12(5):260-261, 1969.

[43] R. Bellman: On a routing problem. *Quart. Appl. Math.*, 16(1):87-90, 1958.

[44] D. Belsnes: Single-message communication. *IEEE Trans. Commun.*, COM-24(2):190-194, 1976.

[45] M. Ben-Ari: *Principles of Concurrent Programming.* Prentice-Hall, Englewood Cliffs, 1982.

[46] M. Ben-Or: Another advantage of free choice: Completely asynchronous agreement protocols. In *Proc. 2nd Ann. ACM Symp. Princ. Dist. Comp.*, p. 27-30, Montreal, 1983.

[47] C. Berge: *Graphs and Hypergraphs.* North-Holland, Amsterdam, 1973.

[48] P. Berman, A. A. Bharali: Distributed consensus in semisynchronous systems. In *Proc. 6th Int. Parallel Proc. Symp.*, p. 632-635, Beverly Hills, IEEE, Los Alamitos, 1992.

[49] P. Berman, J. A. Garay: Cloture voting: $n/4$-resilient distributed consensus in $t+1$ rounds. *Math. Syst. Theory*, 26(1):3-20,1993.

[50] P. A. Berstein, V. Hadzilacos, N. Goodman: *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, 1987.

[51] O. Biran, S. Moran, S. Zaks: A combinatorial characterization of the distributed 1-solvable tasks. *J. Algorithms*, 11(3):420-440, 1990.

[52] K. P. Birman, T. A. Joseph: Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47-76, 1987.

[53] B. Bloom: Constructing two-writer atomic registers. *IEEE Tr. Comm.*, 37(12):1506-1514, 1988.

[54] G. Bochmann, J. Gecsei: A unified method for the specification and verification of protocols. In B. Gilchrist (szerk.), *Information Processing* 77 (Toronto, 1977), vol. 7 Proc. of IFIP Congress, p. 229-234. North-Holland, Amsterdam, 1977.

[55] E. Borowsky, E. Gafni: Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proc. 25th Ann. Proc. Ann. ACM Symp. Theory Comp.*, p. 91-100, San Diego, 1993.

[56] G. Bracha, S. Toueg: Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824-840, 1985.

[57] G. Bracha, S. Toueg: Distributed deadlock detection. *Distr. Comp.*, 2(3):127-138, 1987.

[58] M. F. Bridgland, R. J. Watro: Fault-tolerant decision making in totally asynchronous distributed systems. In *Proc. 6th Ann. Proc. Ann. ACM Symp. Princ. Dist. Comp.*, p. 52-63, Vancouver, 1987.

[59] M. Broy: Functional specification of time sensitive communicating systems. In W. P. de Roever et al. (szerk.), *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness* (REX Workshop, Mook, 1989), vol. 430 *LNCS*, p. 153-179. Springer-Verlag, New York, 1990.

[60] J. E. Burns: Mutual exclusion with linear waiting using binary shared variables. *ACM SIGACT News*, 10(2):42-47, 1978.

[61] J. E. Burns: A formal model for message passing systems. Technical Report TR-91, Comp. Sci. Dept., Indiana University, Bloomington, 1980.

[62] J. E. Burns, P. Jackson, N. A. Lynch, M. J. Fischer, G. L. Peterson: Data requirements for implementation of N-process mutual exclusion using a single shared variable. *J. ACM*, 29(1):183-205, 1982.

[63] J. E. Burns, N. A. Lynch: Bounds on shared memory for mutual exclusion. *Inform. Comput.*, 107(2):171-184, 1993.

[64] O. S. F. Carvalho, G. Roucairol: On mutual exclusion in computer networks. *Commun. ACM*, 26(2):146-148, 1983.

[65] T. D. Chandra, V. Hadzilacos, S. Toueg: The weakest failure detector for solving consensus. *J. ACM*, 43(4):685-722, 1996.

[66] T. D. Chandra, S. Toueg: Unreliable failure detectors for asynchronous systems. *J. ACM*, 43(2):225-267, 1996.

[67] K. M. Chandy, J. Misra: The Drinking Philosophers problem. *ACM Trans. Program. Lang. Systems*, 6(4):632-646, 1984.

[68] K. M. Chandy, L. Lamport: Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comp. Syst.*, 3(1):63-75, 1985.

[69] K. M. Chandy, J. Misra: *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, 1988.

[70] K. M. Chandy, J. Misra, L. M. Haas: Distributed deadlock detection. *ACM Trans. Comput. Syst.*, 1(2):144-156, 1983.

[71] E. Chang, R. Roberts: An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM*, 22(5):281-283, 1979.

[72] E. J. H. Chang: Echo algorithms: Depth parallel operations on general graphs. *IEEE Trans. Softw. Eng.*, SE-8(4):391-401, 1982.

[73] S. Chaudhuri: More *choices* allow more *faults*: Set consensus problems in totally asynchronous systems. *Inform. Comput.*, 105(1):132-158, 1993.

[74] S. Chaudhuri, R. Gawlick, N. A. Lynch: Designing algorithms for distributed systems with partially synchronized clocks. *Proc. 12th Annual ACM Symp. Princ. Dist. Comp.*, p. 121-132, Ithaca, 1993.

[75] S. Chaudhuri, M. Herlihy, N. A. Lynch, M. R. Tuttle: Tight bounds for k-set agreement. Technical Report 95/4, Digital Equipment Corporation, Cambridge Research Lab, Cambridge and *J. ACM*, 47(5):912-943.

[76] S. Chaudhuri, M. Herlihy, N. A. Lynch, M. R. Tuttle: A tight lower bound for k-set agreement. In *34th Ann. Symp. Found. Comp. Sci.*, p. 206-215, Palo Alto. IEEE, Los Alamitos, 1993.

[77] S. Chaudhuri, M. Herlihy, N. A. Lynch, M. R. Tuttle: A tight lower bound for processor coordination. In D. S. Fussell and M. Malek (szerk.): *Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems*, chapter l, p. 1-18. Kluwer Academic, Boston, 1995. (Selected papers from *2nd Int. Workshop on Responsive Comp. Syst.*, Lincoln, 1993.)

[78] B. Chor, A. Israeli, M. Li: On processor coordination using asynchronous hardware. In *Proc. 6th Ann. ACM Symp. Princ. Dist. Comp.*, p. 86-97, Vancouver, 1987.

[79] Ching-Tsun Chou, E. Gafni: Understanding and verifying distributed algorithms using stratified decomposition. In *Proc. 7th Ann. ACM Symp. Princ. Dist. Comp.*, p. 44-65, Toronto, 1988.

[80] M. Choy, A. K. Singh: Efficient fault tolerant algorithms for resource allocation in distributed systems. In *Proc. 24th Ann. ACM Symp. Theory Comp.*, p. 593-602, Victoria, 1992.

[81] W. D. Clinger: *Foundations of Actor Semantics*. Ph.D. thesis, Dept. of Math., MIT, Cambridge, 1981. University Microfilms, Ann Arbor.

[82] B. A. Coan: *Achieving Consensus in Fault-Tolerant Distributed Computer Systems: Protocols, Lower Bounds, and Simulations.* Ph.D. thesis, Dept. of Electr. Eng. and Comp. Sci., MIT, Cambridge, 1987.

[83] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein: *Introduction to Algorithms.* MIT Press/McGraw-Hill, Cambridge/New York, 2001.

[84] A. B. Cremers, T. N. Hibbard: Mutual exclusion of N processors using an $O(N)$-valued message variable. In G. Ausiello, C. Böhm (szerk.), *Automata, Languages and Programming: 5th Coll.* (5th ICALP, Udine, 1978), vol. 62 of *LNCS*, p. 165-176. Springer-Verlag, New York, 1978.

[85] A. B. Cremers, T. N. Hibbard: Arbitration and queueing under limited shared storage requirements. Technical Report 83, Dept. of Informatics, Univ. of Dortmund, 1979.

[86] N. G. de Bruijn: Additional comments on a problem in concurrent programming control. *Commun. ACM*, **10**(3):137-138, 1967.

[87] W. P. de Roever, F. A. Stomp: A correctness proof of a distributed minimum-weight spanning tree algorithm. In *Proc. of the 7th Int. Conf. Distr. Comp. Syst.*, p. 440-447, Berlin, IEEE, Los Alamitos, 1987.

[88] R. A. DeMillo, N. A. Lynch, M. J. Merritt: Cryptographic protocols. In *Proc. 14th Ann. Proc. Ann. ACM Symp. Theory Comp.*, p. 383-400, San Francisco, 1982.

[89] H. Devarajan: A correctness proof for a network synchronizer. Master's thesis, Dept. of Electr. Eng. and Comp. Sci., MIT, Cambridge, 1993. Technical Report MIT/LCS/TR-588.

[90] E. W. Dijkstra: Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

[91] E. W. Dijkstra: Hierarchical ordering of sequential processes. *Acta Inform.*, 1(2):115-138,1971.

[92] E. W. Dijkstra, C. S. Scholten: Termination detection for diffusing computations. *Inf. Proc. Letters*, 11(1):1-4, 1980.

[93] D. Dolev, H. R. Strong: Authenticated algorithms for Byzantine agreement. *SIAM J. Comp.*, 12(4):656-666, 1983.

[94] D. Dolev: The Byzantine generals strike again. *J. ACM*, 3(1):14-30, 1982.

[95] D. Dolev, C. Dwork, L. Stockmeyer: On the minimal synchronism needed for distributed consensus. *J. ACM*, 34(1):77- 97, 1987.

[96] D. Dolev, M. J. Fischer, R. Fowler, N. A. Lynch, H. R. Strong: An efficient algorithm for Byzantine agreement without authentication. *Inform. Control*, 52(3):257-274, 1982.

[97] D. Dolev, M. Klawe, M. Rodeh: An $O(nlogn)$ *unidirectional distributed algorithm for extrema finding in a circle.* *J. ACM*, 3(3):245-260, 1982.

[98] D. Dolev, N. A. Lynch, S. S. Pinter, E. W. Stark, W. E. Weihl: Reaching approximate agreement in the presence of faults. *J. ACM*, 33(3):499-516, 1986.

[99] D. Dolev, R. Reischuk, H. R. Strong: Early stopping in Byzantine agreement. *J. ACM*, 37(4):720-741, 1990.

[100] D. Dolev, N. Shavit: Bounded concurrent time-stamp systems are constructible. In *Proc. 21st Ann. Proc. Ann. ACM Symp. Theory Comp.*, p. 454-466, Seattle, 1989. *SIAM J. Comp.*, 26(1):273-290.

[101] D. Dolev, H. R. Strong: Polynomial algorithms for multiple processor agreement. In *Proc. 14th Ann. Proc. Ann. ACM Symp. Theory Comp.*, p. 401-407, San Francisco, 1982.

[102] C. Dwork M. Herlihy, S. A. Plotkin, O. Waarts: Timelapse snapshots. In D. Dolev et al. (szerk.): *Theory of Computing and Systems* (ISTCS '92, Israel Symposium, Haifa, 1992), vol. 601 of *LNCS*, p. 154-170. Springer-Verlag, New York, 1992.

[103] C. Dwork, M. P. Herlihy, O. Waarts: Contention in shared memory algorithms. In *Proc. 25th Ann. ACM Symp. Theory Comp.*, p. 174-183, San Diego, 1993. Expanded version in Technical Report CLR 93/ 12, Digital Equipment Corporation, Cambridge Research Lab., Cambridge.

[104] C. Dwork, N. A. Lynch, L. Stockmeyer: Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288-323, 1988.

[105] C. Dwork, Y. Moses: Knowledge and common knowledge in a Byzantine environment: Crash failures. *Inform. Comp.*, 88(2):156-186, 1990.

[106] C. Dwork, D. Skeen: The inherent cost of nonblocking commitment. In *Proc. 2nd Ann. ACM Symp. Princ. Dist. Comp.*, p. 1-11, Montreal, 1983.

[107] C. Dwork, O. Waarts: Simple and efficient bounded concurrent timestamping and the traceable use abstraction. In *Proc. 24th Ann. ACM Symp. Theory Comp.*, p. 655-666 Victoria, 1992. Végleges változat: *J. ACM*, **46**(5):633-666, 1999.

[108] M. A. Eisenberg, M. R. McGuire: Further comments on Dijkstra's concurrent programming control problem. *Commun. ACM*, 15(11):999, 1972.

[109] A. D. Fekete, N. A. Lynch, L. Shrira: A modular proof of correctness for a network synchronizer. In J. van Leeuwen (szerk.): *Distributed Algorithms* (2nd International Workshop, Amsterdam, 1987), vol. 312 of *LNCS*, p. 219-256. Springer-Verlag, New York, 1988.

[110] A. D. Fekete: Asymptotically optimal algorithms for approximate agreement. *Distr. Comp.*, 4(1):9-29, 1990.

[111] A. D. Fekete: Asynchronous approximate agreement. *Inform. Comp.*, 115(1):95-124, 15, 1994.

[112] A. Fekete, N. A. Lynch, Y. Mansour, J. Spinelli: The impossibility of implementing reliable communication in the face of crashes. *J. ACM*, 40(5):1087-1107, 1993.

[113] P. Feldman, S. Micali: An optimal probabilistic protocol for synchronous Byzantine agreement. Preliminary version appeared as Technical Report MIT/LCS/TM-425.b, Cambridge, 1992. *SIAM J. Comp.*, 26(4):873-933, 1983.

[114] P. N. Feldman: Optimal Algorithms for Byzantine Agreement. Ph.D. thesis, Dept. Math., MIT, Cambridge, 1988.

[115] C. J. Fidge: Timestamps in message-passing systems that preserve the partial ordering. In *Proc. 11th Australian Comp. Sci. Conf.*, p. 56-66, Brisbane, 1988.

[116] M. J. Fischer: Re: Where are you? E-mail message to Leslie Lamport. Arpanet message number 8506252257.AA07636@YALE-BULLDOG.YALE.ARPA (47 lines), June 25, 1985.

[117] M. J. Fischer: The consensus problem in unreliable distributed systems (a brief survey). Research Report YALEU DCS RR-273, Yale University, Dept. Comp. Sci., New Haven, 1983.

[118] M. J. Fischer, N. D. Griffeth, N. A. Lynch: Global states of a distributed system. *IEEE Trans. Soft. Eng.*, SE8(3):198-202, 1982.

[119] M. J. Fischer, N. A. Lynch: A lower bound for the time to assure interactive consistency. *Inf. Proc. Letters*, 14(4):183- 186, June 1982.

[120] M. J. Fischer, N. A. Lynch, J. E. Burns, A. Borodin: Resource allocation with immunity to limited process failure. In *20th Annual Symp. Found. Comp. Sci.*, p. 234-254, San Juan, Puerto Rico, October 1979. IEEE, Los Alamitos.

[121] M. J. Fischer, N. A. Lynch, J. E. Burns, A. Borodin: Distributed FIFO allocation of identical resources using small shared space. *ACM Trans. Program. Lang. Syst.*, 11(1):90-114, 1989.

[122] M. J. Fischer, N. A. Lynch, M. Merritt: Easy impossibility proofs for distributed consensus problems. *Distr. Comp.*, 1(1):26-39, 1986.

[123] M. J. Fischer, N. A. Lynch, M. S. Paterson: Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374-382, 1985.

[124] R. W. Floyd: Assigning meanings to programs. In *Mathematical Aspects of Computer Science* (New York, April 1966), volume 19 of *Proc. Symp. Appl. Math.*, p. 19-32. American Mathematical Society, Providence, 1967.

[125] L. R. Ford, Jr., D. R. Fulkerson: *Flows in Networks*. Princeton University Press, Princeton, 1962.

[126] N. Francez: Distributed termination. *ACM Trans. Program. Lang. Syst.*, 2(1):42-55, 1980.

[127] G. N. Frederickson, N. A. Lynch: Electing a leader in a synchronous ring. *J. ACM*, 34(1):98-115, 1987.

[128] H. N. Gabow: Scaling algorithms for network problems. *J. Comp. Syst. Sci.*, 31 (2):148-168, 1985.

[129] E. Gafni: Személyes közlés, 1994.

[130] R. G. Gallager, P. A. Humblet, P. M. Spira: A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66-77, 1983.

[131] R. G. Gallager: Distributed minimum hop algorithms. Technical Report LIDS-P-1175, Lab. for Inform. and Decision Syst., MIT, Cambridge, 1982.

[132] J. A. Garay, S. Kutten, D. Peleg: A sub-linear time distributed algorithm for minimum-weight spanning trees. In *34th Ann. Symp. Found. Comp. Sci.*, p. 659-668, Palo Alto, Calif., 1993. IEEE, Los Alamitos.

[133] J. A. Garay, Y. Moses: Fully polynomial Byzantine agreement in $t+1$ rounds. In *Proc. 25th Annual ACM Symp. Theory Comp.*, p. 31-41, San Diego, 1993.

[134] S. J. Garland, J. V. Guttag: A guide to LP, the Larch Prover. Research Report 82, Digital Systems Research Center, Palo Alto, 1991.

[135] R. Gawlick, N. A. Lynch, N. Shavit: Concurrent time-stamping made simple. In D. Dolev et al. (szerk.): *Theory of Computing and Systems* (ISTCS '92, Israel Symposium, Haifa, 1992), volume 601 *LNCS*, p. 171-185. Springer-Verlag, New York, 1992.

[136] R. Gawlick, R. Segala, J. Søgaard-Andersen, N. A. Lynch: Liveness in timed and untimed systems. In S. Abiteboul and E. Shamir (szerk.): *Automata, Languages and Programming* (2lst Int. Coll., ICALP '94, Jerusalem, 1994), volume 820 *LNCS*, p. 166-177. Springer-Verlag, New York,1994.

[137] D. K. Gifford: Weighted voting for replicated data. In *Proc. 7th Symp. Operating Syst. Princ.*, p. 150-162, Pacific Grove, Calif., 1979. ACM, New York.

[138] V. D. Gligor, S. H. Shattuck: On deadlock detention in distributed systems. *IEEE Trans. Softw. Eng.*, SE6(5):435-440, 1980.

[139] K. J. Goldman, K. Yelick: A unified model for shared-memory and message-passing systems. Technical Report WUCS-93-35, Washington Univ., St. Louis, 1993.

[140] K. J. Goldman, N. A. Lynch: Quorum consensus in nested transaction systems. *ACM Trans. Database Syst.*, 19(4):537-585, 1994.

[141] K. J. Goldman, N. A. Lynch: Modelling shared state in a shared action model. In *Proc. 5th Annual IEEE Symp. Logic Comp. Sci.*, p. 450-463, Philadelphia, 1990.

[142] J. N. Gray: Notes on data base operating systems. In R. Bayer et. al. (szerk.): *Operating Systems: An Advanced Course*, volume 60 *LNCS*, chapter 3.F, page 465. Springer-Verlag, New York, 1978.

[143] V. Hadzilacos, S. Toueg: Fault-tolerant broadcasts and related problems. In S. Mullender (szerk.): *Distributed Systems*, second edition, chapter 5, p. 97-145. ACM Press Addison-Wesley, New York Reading, Mass.,1993.

[144] S. Haldar, K. Vidyasankar: Constructing 1-writer multireader multivalued atomic variables from regular variables. *J. ACM*, 42(1):186-203, 1995.

[145] J. Y. Halpern, Y. Moses, O. Waarts: A characterization of eventual byzantine agreement. In *Proc. 9th Annual ACM Symp. Princ. Distr. Comp.*, p. 333-346, Quebec City, August 1990.

[146] J. Y. Halpern, L. D. Zuck: A little knowledge goes a long way: Knowledge-based proofs for a family of protocols. *J. ACM*, 39(3):449-478, July 1992.

[147] F. Harary: *Graph Theory*. Addison-Wesley, Reading, 1972.

[148] C. Heitmeyer, N. Lynch: The generalized railroad crossing: A case study in formal verification of real-time systems. Technical Memo MIT/LCS/TM-511, 1994. Abbreviated version in *Proc. Real-Time Syst. Symp.*, p. 120-131, San Juan, 1994. IEEE, Los Alamitos. Later version to appear in C. Heitmeyer and D. Mandrioli (szerk.): *Formal Methods for Real-time Computing*, chapter 4, Trends in Software series, John Wiley & Sons, New York.

[149] M. P. Herlihy: A quorum-consensus replication method for abstract data types. *ACM Trans. Comp. Syst.*, 4(1):32-53, 1986.

[150] M. P. Herlihy: Wait-free synchronization. *ACM Trans. Progr. Lang. Syst.*, 13(1):124-149, 1991.

[151] M. P. Herlihy, N. Shavit: A simple constructive computatability theorem for wait-free computation. In *Proc. 26th Ann. ACM Symp. Theory Comp.*, p. 243-262, Montreal 1994.

[152] M. P. Herlihy, N. Shavit: The asynchronous computability theorem for t-resilient tasks. In *Proc. 25th Ann. ACM Symp. Theory Comp.*, p. 111-120, San Diego, 1993.

[153] M. P. Herlihy, J. M. Wing: Linearizability: A correctness condition for concurrent objects. *ACM Trans. Progr. Lang. Syst.*, 12(3):463-492, 1990.

[154] M. P. Herlihy: Replication Methods for Abstract Data Types. Ph.D. thesis, Dept. Electr. Eng. Comp. Sci., MIT, Cambridge, 1984. Technical Report MIT/LCS/TR-319.

[155] L. Higham, T. Przytycka: A simple, efficient algorithm for maximum finding on rings. In A. Schiper (szerk.): *Distributed Algorithms* (7th Int. Workshop, WDAG 93, Lausanne, 1993), vol. 725 *LNCS*, p. 249- 263. Springer-Verlag, New York, 1993.

[156] D. S. Hirschberg, J. B. Sinclair: Decentralized extrema-finding in circular configurations of processes. *Commun. ACM*, 23(11):627-628, 1980.

[157] G. S. Ho, C. V. Ramamoorthy: Protocols for deadlock detection in distributed database systems. *IEEE Tr. Soft. Eng.* SE-8(6):554-557, 1982.

[158] C. A. R. Hoare: Proof of correctness of data representations. *Acta Inform.*, 1(4):271-281, 1972.

[159] C. A. R. Hoare: *Communicating Sequential Processes*. Prentice-Hall International, United Kingdom, 1985.

[160] P. A. Humblet: A distributed algorithm for minimum weight directed spanning trees. *IEEE Tr. Soft. Eng.*, COM-31(6):756- 762, 1983.

[161] S. S. Isloor, T. A. Marsland: An effective "on-line" deadlock detection technique for distributed database management systems. In *Proc. of COMPSAC* 78: *IEEE Computer Society's 2nd Int. Comp. Softw. Appl. Conf.*, p. 283-288, Chicago, 1978.

[162] A. Israeli, M. Li: Bounded time-stamps. *Distr. Comp.*, 6(4):205-209, 1993.

[163] A. Israeli, M. Pinhasov: A concurrent time-stamp scheme which is linear in time and space. In A. Segall and S. Zaks (szerk.): *Distr. Alg.: 6th International Workshop* (WDAG '92, Haifa, 1992), vol. 647 *LNCS*, p. 95-109. Springer-Verlag, New York, 1992.

[164] W. Janssen, J. Zwiers: From sequential layers to distributed processes: Deriving a distributed minimum weight spanning tree algorithm. In *Proc. 11th Ann. ACM Symp. Princ. Dist. Comp.*, p. 215-227, Vancouver, 1992.

[165] B. Jonsson: Compositional specification and verification of distributed systems. *ACM Trans. Lang. Sys.* 16(2):259-303, 1994.

[166] R. M. Karp, V. Ramachandran: Parallel algorithms for shared-memory machines. In J. van Leeuwen, (szerk.): *Algorithms and Complexity*, volume A of *Handbook of Theoretical Computer Science*, chapter 17, p. 869-942. North-Holland/ MIT Press, New York/Cambridge, 1990.

[167] J. Kleinberg, H. Attiya, N. Lynch: Trade-offs between message delivery and quiesce times in connection management protocols. In *Proc. of ISTCS 1995: 3rd Israel Symp. Theory Comp. Systems*, p. 258-267, Tel Aviv, IEEE, Los Alamitos, 1995.

[168] D. E. Knuth: Additional comments on a problem in concurrent programming control. *Commun. ACM*, 9(5):321-322, 1966.

[169] D. E. Knuth: *Fundamental Algorithms*, vol. 1 of *The Art of Computer Programming*. Harmadik, javított és bővített kiadás. Addison-Wesley, Reading, 1997.

[170] D. Kőnig: Sur les correspondances multivoques des ensembles. *Fundam. Math.*, 8:114-134, 1926.

[171] C. P. Kruskal, L. Rudolph, M. Snir: Efficient synchronization on multiprocessors with shared memory. In *Proc. 5th Ann. ACM Symp. Princ. Dist. Comp.*, p. 218-228 Calgary, 1986.

[172] J. H. Lala: A Byzantine resilient fault-tolerant computer for nuclear power plarzt applications. In *FTCS: l6th Ann. Int. Symp. Fault-Tolerant Comp. Syst.*, p. 338-343, Vienna. IEEE, Los Alamitos, 1986.

[173] J. H. Lala, R. E. Harper, L. S. Alger: A design approach for ultrareliable real-time systems. *Computer*, 24(5):12-22, 1991.

[174] L. Lamport: A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453-455, 1974.

[175] L. Lamport: Proving the correctness of multiprocess programs. *IEEE Tr. Softw. Eng.*, SE-3(2):125-143, 1977.

[176] L. Lamport: Time, clocks, the ordering of events in a distributed system. *Commun. ACM*, 21(7):558-565, 1978.

[177] L. Lamport: Specifying concurrent program modules. *ACM Trans. Lang. Syst.*, 5(2):190-222, 1983.

[178] L. Lamport: The weak Byzantine generals problem. *J. ACM*, 30(3):669-676, 1983.

[179] L. Lamport: Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Lang. Syst.*, 6(2):254-280, 1984.

[180] L. Lamport: The mutual exclusion problem. Part II: Statement and solutions. *J. ACM*, 33(2):327-348, 1986.

[181] L. Lamport: On interprocess communication, Part I: Basic formalism. *Distr. Comp.*, 1(2):77-85, 1986.

[182] L. Lamport: On interprocess communication, Part II: Algorithms. *Distr. Comp.*, 1(2):86-101, 1986.

[183] L. Lamport: The part-time parliament. Research Report 49, Digital Systems Research Center, Palo Alto, 1989.

[184] L. Lamport: The temporal logic of actions. *ACM Trans. Lang. Syst.*, 16(3):872-923, 1994.

[185] L. Lamport, N. Lynch: Distributed computing: Models and methods. In J. van Leeuwen (szerk.): *Formal Models and Semantics*, vol. B of *Handbook of Theoretical Computer Science*, chapter 18, p. 1157-1199. North-Holland/MIT Press, New York/Cambridge, 1990.

[186] L. Lamport, F. B. Schneider: Pretending atomicity. Research Report 44, Digital Equipment Corporation, Systems Research Center, Palo Alto, 1989.

[187] L. Lamport, R. Shostak, M. Pease: The Byzantine generals problem. *ACM Trans. Prog. Lang. Syst.*, 4(3):382-401, 1982.

[188] B. Lampson, N. A. Lynch, J. Søagaard-Andersen: At-most-once message delivery: A case study in algorithm verification. In W. R. Cleaveland (szerk.): *CONCUR'92* (Third Int. Conf. on Concurrency Theory, Stony Brook, 1992), vol. 630 of *LNCS*, p. 317-324. Springer-Verlag, New York, 1992

[189] B. Lampson, W. Weihl, U. Maheshwari: *Principles of Computer Systems: Lecture Notes for 6.826*, Fall 1992. Research Seminar Series MIT/LCS/RSS 22, Cambridge, 1993.

[190] B. W. Lampson, N. A. Lynch, J. F. Søgaard-Andersen: Correctness of at-most-once message delivery protocols. In R. L. Tenney et al. (szerk.): *Formal Description Techniques VI* (Proc. of the IFIP TC6 WG6.1 6th Int. Conf. on Formal Description Techniques, FORTE '93, Boston, 1993) *IFIP Transactions C*, p. 385-400. North-Holland, Amsterdam,1994.

[191] G. Le Lann: Distributed systems-towards a formal approach. In B. Gilchrist (szerk.): *Information Processing* 77 (Toronto 1977), vol. 7 of *Proc. of IFIP Congress*, p. 155-160. North-Holland, Amsterdam, 1977.

[192] D. Lehmann, M. O. Rabin: On the advantages of free choice: A symmetric and fully distributed solution to the Dining Philosophers problem. In *Proc. 8th Ann. ACM Symp. Princ. Progr. Lang.*, p. 133-138, Williamsburg, 1988.

[193] F. T. Leighton: *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, San Mateo, 1992.

[194] H. R. Lewis: Finite-state analysis of asynchronous circuits with bounded temporal uncertainty. Technical Report TR-15-89, Center for Research in Computing Technology, Aiken Computation Lab., Harvard Univ., Cambridge, 1989.

[195] H. R. Lewis, C. H. Papadimitriou: *Elements of the Theory of Computation.* Prentice-Hall, Englewood Cliffs, 1981.

[196] M. Li, P. M. B. Vitanyi : How to share concurrent wait-free variables. *J. ACM*, 43(4):723-746, 1996.

[197] B. Liskov, R. Ladin: Highly-available distributed services and fault-tolerant distributed garbage collection. In *Proc. 5th Ann. ACM Symp. Princ. Dist. Comp.*, p. 29-39, Calgary, 1986.

[198] B. Liskov, A. Snyder, R. Atkinson, C. Schaffert: Abstraction mechanisms in CLU. *Commun. ACM*, 20(8):564- 576, 1977.

[199] M. C. Loui, H. H. Abu-Amara: Memory requirements for agreement among unreliable asynchronous processes. In F. P. Preparata (szerk.): *Parallel and Distributed Computing*, vol. 4 of *Advances in Computing Research*, p. 163-183. JAI Press, Greenwich, 1987.

[200] M. Luby: A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comp.*, 15(4):1036-1053, 1986.

[201] V. Luchangco: Using simulation techniques to prove timing properties. Master's thesis, Dept. of Electr. Eng. Comp. Sci., MIT, Cambridge, 1995.

[202] V. Luchangco, E. Söylemez, S. Garland, N. A. Lynch: Verifying timing properties of concurrent algorithms. In D. Hogrefe and S. Leue (szerk.): *Formal Description Techniques VII: Proc. 7th IFIP WG6.1 Int. Conf. Formal Descr. Techniques* (FORTE '94, Berne, 1994), p. 259-273. Chapman and Hall, New York,1995.

[203] N. A. Lynch: Concurrency control for resilient nested transactions. In *Proc. of the Second ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, p. 166-181, Atlanta, 1983.

[204] N. A. Lynch: Simulation techniques for proving properties of real-time systems. In W. P. de Roever et al. (szerk.): *A Decade of Concurrency: Reflections and Perspectives* (REX School Symposium, Noordwijkerhout, 1993), vol. 803 *LNCS*, p. 375-424. Springer-Verlag, New York,1994.

[205] N. A. Lynch: Simulation techniques for proving properties of real-time systems. In S. H. Son (szerk.): *Advances in Real-Time Systems*, chapter 13, p. 299-332. Prentice-Hall, Englewood Cliffs, 1995.

[206] N. A. Lynch, Y. Mansour, A. Fekete: The data link layer: Two impossibility results. In *Proc. 7th Ann. ACM Symp. Princ. Dist. Comp.*, p. 149-170, Toronto, 1988.

[207] N. A. Lynch, M. Merritt, W. Weihl, A. D. Fekete: *Atomic Transactions.* Morgan Kaufmann, San Mateo, 1994.

[208] N. A. Lynch, I. Saias, R. Segala: Proving time bounds for randomized distributed algorithms. In *Proc. 13th Ann. ACM Symp. Princ. Dist. Comp.*, p. 314-323, Los Angeles, 1994.

[209] N. A. Lynch, N. Shavit: Timing-based mutual exclusion. In *Proc. Real-Time Syst. Symp.*, p. 2-11, Phoenix, IEEE, Los Alamitos, 1992.

[210] N. A. Lynch, F. Vaandrager: Forward and backward simulations for timing-based systems. Technical Memo MIT/LCS/TM- 458 and in J. W. de Bakker et al. (szerk.): *Real-Time: Theory in Practice* (REX Workshop, Mook, 1991), vol. 600 *LNCS*, p. 397-446. Springer-Verlag, New York,1992.

[211] N. A. Lynch, F. Vaandrager: Forward and backward simulations Part II: Timing-based systems. *Inform. Comput.*, 128(1): 1-25, 1996.

[212] N. A. Lynch, F. Vaandrager: Action transducers and timed automata. Technical Memo MIT/LCS/TM- 480.b, Cambridge, 1994.

[213] N. A. Lynch: Upper bounds for static resource allocation in a distributed system. *J. Computer System Sci.*, 23(2):254-278, 1981.

[214] N. A. Lynch: Multivalued possibilities mappings. In W. P. de Roever et al. (szerk.): *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness* (REX Workshop, Mook, 1989), vol. 430 of *LNC*, p. 519-543. Springer-Verlag, New York, 1990.

[215] N. A. Lynch, H. Attiya: Using mappings to prove timing properties. *Distr. Comp.*, 6(2):121-139, 1992.

[216] N. A. Lynch, M. J. Fischer: On describing the behavior and implementation of distributed systems. *Theor. Comp. Sci.*, **13**(1):17-43, 1981.

[217] N. A. Lynch, M. R. Tuttle: Hierarchical correctness proofs for distributed algorithms. Master's thesis, Dept. Electr. Eng. Comp. Sci., MIT, Cambridge, 1987. Technical Report MIT/LCS/TR- 387. Rövidített változat in *Proc. 6th Ann. ACM Symp. Princ. Dist. Comp.*, p. 137-151, Vancouver, 1987.

[218] N. A. Lynch, M. R. Tuttle: An introduction to input output automata. *CWI Quart.*, 2(3):219-246, 1989. Technical Memo MIT/LCS/TM373, Cambridge, 1988.

[219] Z. Manna, A. Pnueli: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer-Verlag, New York, 1992.

[220] Y. Mansour, B. Schieber: The intractability of bounded protocols for on-line sequence transmission over non-FIFO channels. *J. ACM*, **39**(4):783-799, 1992.

[221] J. C. Martin: *Introduction to Languages and the Theory of Computation.* McGraw-Hill, New York, 1991.

[222] F. Mattern: Virtual time and global states of distributed systems. In Michel Cosnard et al. (szerk.): *Parallel and Distributed Algorithms: Proc. Int. Workshop Parallel Distr. Alg.* (Gers, October, 1988), p. 215-226. North-Holland, Amsterdam, 1989.

[223] John M. McQuillan, G. Falk, I. Richer: A review of the development and performance of the ARPANET routing algorithm. *IEEE Trans. Comm.*, COM-26(12):1802-1811, 1978.

[224] D. A. Menasce, R. R. Muntz: Locking and deadlock detection in distributed data bases. *IEEE Tr. Softw. Eng.*, SE-5(3):195-202, 1979.

[225] K. Menger: Zur allgemeinen Kurventheorie. *Fundam. Math.*, 10:96-115, 1927.

[226] M. Merritt, 1985: Kiadatlan jegyzetek.

[227] M. Merritt, F. Modugno, M. R. Tuttle: Time constrained automata. In J. C. M. Baeten and J. F. Goote (szerk.): *CONCUR'91: 2nd Int. Conf. Concurrency Theory* (Amsterdam, 1991), vol. 527 *LNCS*, p. 408-423. Springer-Verlag, New York,1991.

[228] R. Milner: An algebraic definition of simulation between programs. In *2nd Int. Joint Conf. on Art. Int.*, p. 481- 489, Imperial College, London, 1971. British Computer Society, London.

[229] R. Milner: *Communication and Concurrency.* Prentice-Hall International, United Kingdom,1989.

[230] S. Moran, Y. Wolfstahl: Extended impossibility results for asynchronous complete networks. *Inf. Proc. Letters*, 26(3):145-151, 1987.

[231] Y. Moses, O. Waarts: Coordinated traversal: $(t + 1)$-round Byzantine agreement in polynomial time. *J. ACM*, 17(1):110-156, 1994.

[232] G. Neiger, S. Toueg: Simulating synchronized clocks and common knowledge in distributed systems. *J. ACM*, 40(2):334-367, 1993.

[233] T. Nipkow: Formal verification of data type refinement: Theory and practice. In W. P. de Roever et al. (szerk.): *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness* (REX Workshop, 1989), vol. 430 *LNCS*, p. 561-591. Springer-Verlag New York,1990.

[234] R. Obermarck: Distributed deadlock detection algorithm. *ACM Trans. Database Syst.*, 7(2):187-208, 1982.

[235] S. Owicki, D. Gries: An axiomatic proof technique for parallel programs, I. *Acta Inform.*, 6(4):319-340, 1976.

[236] D. Park: Concurrency and automata on infinite sequences. In P. Deussen (szerk.): *Theoretical Computer Science* (5th GI Conference, Karlsruhe, 1981), vol. 104 *LNCS*, p. 167-183. Springer-Verlag, New York, 1981.

[237] M. Pease, R. Shostak, L. Lamport: Reaching agreement in the presence of faults. *J. ACM*, 27(2):228-234, 1980.

[238] G. L. Peterson: Myths about the mutual exclusion problem. *Inf. Proc. Letters*, 12(3):115-116, 1981.

[239] G. L. Peterson: An $O(n \log n)$ unidirectional distributed algorithm for the circular extremal problem. *ACM Trans. Lang. Syst.*, 4(4):758-762, 1982.

[240] G. L. Peterson: Concurrent reading while writing. *ACM Trans. Lang. Syst.*, 5(1):46-55, 1983.

[241] G. L. Peterson, J. E. Burns: Concurrent reading while writing II: The multi-writer case. In *8th Ann. Symp. Found. Comp. Sci.*, p. 383-392, Los Angeles, IEEE, Los Alamitos, 1987.

[242] G. L. Peterson, M. J. Fischer: Economical solutions for the critical section problem in a distributed system. In *Proc. Ninth Ann. ACM Symp. Theory Comp.*, p. 91-97, Boulder, 1977.

[243] A. Pnueli: Személyes közlés, 1988.

[244] A. Pnueli, L. Zuck: Verification of multiprocess probabilistic protocols. *Distr. Comp.*, **1**(1):53-72, 1986.

[245] S. Ponzio: Consensus in the presence of timing uncertainty: Omission and Byzantine failures. In *Proc. 10th Ann. ACM Symp. Princ. Dist. Comp.*, p. 125-138, Montreal, 1991.

[246] S. Ponzio: Bounds on the time to detect failures using bounded-capacity message links. In *Proc. Real-time Systems Symp.*, p. 236-245, Phoenix, IEEE, Los Alamitos, 1992.

[247] S. J. Ponzio: The real-time cost of timing uncertainty: Consensus and failure detection. Master's thesis, Dept. Electr. Eng. Comp. Sci., MIT, Cambridge, 1991. Technical Report MIT/LCS/TR- 518.

[248] M. O. Rabin: Randomized Byzantine generals. In *4th Annual Symp. Found. Comp. Sci.*, p. 403-409, Tucson, IEEE, Los Alamitos, 1983.

[249] M. Raynal: *Algorithms for Mutual Exclusion*. MIT Press, Cambridge, 1986.

[250] M. Raynal: *Networks and Distributed Computation: Concepts, Tools, and Algorithms*. MIT Press, Cambridge, 1988.

[251] M. Raynal, Jean-Michel Helary: *Synchronization and Control of Distributed Systems and Programs*. John Wiley & Sons, Ltd., Chichester, 1990.

[252] G. Ricart, A. K. Agrawala: An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24(1 :9-17, 1981. Corrigendum in *Commun. ACM*, **24**(9 :578, 1981.

[253] M. Saks, F. Zaharoglou: Wait-free k-set agreement is impossible: The topology of public knowledge. In *Proc. 5th Ann. Proc. Symp. Theory Comp.*, p. 101-110, San Diego, 1993.

[254] R. Schaffer: On the correctness of atomic multi-writer registers. Technical Memo MIT/LCS/TM-364, Cambridge, 1988.

[255] F. B. Schneider: Implementing fault-tolerant services using the state machine approach. *ACM Comp. Surveys*, 22(4):299-319, 1990.

[256] R. Schwarz, F. Mattern: Detecting causal relationships in distributed computations: In search of the holy grail. *Distr. Comp.*, 7(3):149-174, 1994.

[257] R. Segala, N. A. Lynch: Probabilistic simulations for probabilistic processes. *Nordic J. Comp.*, 2(2 :250-273, 1995.

[258] A. Segall: Distributed network protocols. *IEEE Trans. Inf. Theory*, IT-29(1):23-35, 1983.

[259] A. U. Shankar: A simple assertional proof system for real-time systems. In *Proc. Real-Time Systems Symp.*, p. 167-176, Phoenix, IEEE, Los Alamitos, 1992.

[260] A. U. Shankar, S. S. Lam: A stepwise refinement heuristic for protocol construction. *ACM Trans. Lang. Syst.*, 14(3 :417-461, 1992.

[261] N. Shavit: Concurrent Time Stamping. Ph.D. thesis, Dept. Comp. Sci., Hebrew University, Jerusalem, 1990.

[262] A. Silberschatz, J. L. Peterson, P. B. Galvin: *Operating System Concepts*, harmadik kiadás. Addison-Wesley, Reading, 1992.

[263] A. K. Singh, J. H. Anderson, M. G. Gouda: The elusive atomic register. *J. ACM*, 41(2 :311-339, 1994.

[264] J. F. Søgaard-Andersen: Correctness of Protocols in Distributed Systems. Ph.D. thesis, Dept. of Comp. Sci., Technical Univ. of Denmark, Lyngby, 1993. ID-TR: 1993-131.

[265] J. F. Søgaard-Andersen, S. J. Garland, J. V. Guttag, N. A. Lynch, A. Pogosyants: Computer-assisted simulation proofs. In C. Courcoubetis (szerk.): *Computer-Aided Verification* (5th Int. Conf., CAV '93, Elounda, 1993), vol. 697 *LNCS*, p. 305-319. Springer-Verlag, New York, 1993.

[266] E. H. Spanier: *Algebraic Topology*. McGraw-Hill, New York, 1966.

[267] E. Sperner: Neuer Beweis für die Invarianz der Dimensionszahl und des Gebietes. *Abhandl. Math. Sem. Hamburgischen Univ.*, 6:265-272, 1928.

[268] J. M. Spinelli: Reliable communication on data links. Technical Report LIDS-P-1844, Lab. Inf. and Dec. Systems, MIT, Cambridge, 1988.

[269] T. K. Srikanth, S. Toueg: Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distr. Comp.* 2(2):80-94, 1987.

[270] N. V. Stenning: A data transfer protocol. *Computer* Networks, 1(2):99-110, 1976.

[271] T. Stoppard: *Rosencrantz & Guildenstern Are Dead*. Jeffrey Norton Pub., New York, 1986.

[272] E. Styer, G. L. Peterson: Improved algorithms for distributed for resource allocation. In *Proc. 7th Ann. ACM Symp. Princ. Dist. Comp.*, p. 105-116, Toronto, 1988.

[273] A. S. Tanenbaum: *Computer Networks*, harmadik kiadás. Prentice-Hall, Englewood Cliffs, 1997.

[274] Y. C. Tay, W. T. Loke: On deadlocks of exclusive AND-requests for resources. *Distr. Comp.* 9(2):77-94, 1995.

[275] G. Tel: Assertional verification of a timer based protocol. In T. Lepist and A. Salomaa (szerk.): *Automata, Languages and Programming* (l5th Int.Coll., ICALP '88, Tampere, 1988), vol. 317 *LNCS*, p. 600-614. Springer-Verlag, New York,1988.

[276] G. Tel: *Introduction to Distributed Algorithms*. Cambridge University Press, Cambridge, 2001.

[277] E. Tempero, R. Ladner: Recoverable sequence transmission protocols. *J. ACM*, 42(5):1059-1090, 1995.

[278] E. Tempero, R. E. Ladner: Tight bounds for weakly-bounded protocols. In *Proc. 9th Ann. ACM Symp. Princ. Dist. Comp.*, p. 205-218, Quebec City, 1990.

[279] R. Turpin, B. A. Coan: Extending binary Byzantine agreement to multivalued Byzantine agreement. *Inf. Proc. Letters*, 18(2):73-76, 1984.

[280] J. L. A. van de Snepscheut: The sliding-window protocol revisited. *Formal Aspects Comp.*, 7(1):3-17, 1995.

[281] George Varghese, N. A. Lynch: A tradeoff between safety and liveness for randomized coordinated attack protocols. In *Proc. llth Ann. ACM Symp. Princ. Dist. Comp.*, p. 241-250, Vancouver, 1992.

[282] P. M. B. Vitanyi: Distributed elections in an Archimedean ring of processors. In *Proc. l6th Ann. ACM Symp. Theory Comp.*, p. 542-547, Washington, 1984.

[283] P. M. B. Vitanyi, B. Awerbuch: Atomic shared register access by asynchronous hardware. In *7th Annual Symp. Found. Comp. Sci.*, p. 233-243, Toronto, IEEE, Los Alamitos, 1986., Corrigendum in *8th Ann. Symp. Found. Comp. Science*, p. 487, Los Angeles, 1987.

[284] Da-Wei Wang, L. D. Zuck: Tight bounds for the sequence transmission problem. In *Proc. 8th Ann. ACM Symp. Princ. Dist. Comp.*, p. 73-83, Edmonton, 1989.

[285] J. L. Welch, N. A. Lynch: A modular Drinking Philosophers algorithm. *Distr. Comp.*, 6(4):233-244, 1993.

[286] J. L. Welch: Simulating synchronous processors. *Inform. Comput.*, 74(2):159-171, 1987.

[287] J. L. Welch: Topics in Distributed Computing: The Impact of Partial Synchrony, and Modular Decomposition of Algorithms. Ph.D. thesis, Dept. Electr. Eng. Comp. Sci., MIT, Cambridge, 1988.

[288] J. L. Welch, L. Lamport, N. A. Lynch: A lattice-structured proof technique applied to a minimum spanning tree algorithm. In *Proc. 7th Ann. ACM Symp. Princ. Dist. Comp.*, p. 28-43, Toronto, 1988.

[289] J. H. Wensley et al: SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proc. of the IEEE*, 66(10):1240-1255, 1978.

[290] H. Zimmerman: OSI reference model-the ISO model of architecture for open systems interconnection. *IEEE Tr. Softw. Eng.*, COM-28(4):425-432, 1980.

Az irodalomjegyzékben az aláhúzás azt jelenti, hogy a DVI és PDF elektronikus változatban a megfelelő szövegrész él (az aláhúzott részre kattintva az olvasó eljut a megfelelő honlapra). A hivatkozások végén lévő számok a szöveg azon részeit jelzik, amelyekben az adott műre hivatkozás történt.

A folyóiratok neveit a *Computing Reviews* szerint rövidítettük.

Tárgymutató

Ez a tárgymutató a következő szempontok szerint készült.

Először a matematikai jelöléseket soroljuk fel, azután a tárgyszavakat.

A számokat és görög betűket tartalmazó tárgyszavakat kiejtésük szerint rendezzük: például az „1-értékű”-t „egyértékékű”-ként, a λ-t „lambda”-ként. A jelölést tartalmazó tárgyszavakat elemeik szerint rendezzük: például a „k-megegyezés”-t „k megegyezés”-ként.

A különböző típusú objektumokat lehetőség szerint tipográfiailag is megkülönböztettük: a műveletek nevét egységesen „teletype” betűkkel írtuk, mint például `küld`. Az algoritmusok (továbbá automaták és protokollok) neveit általában kiskapitális betűkkel írtuk, mint például LamportIdő – az egybetűs neveket azonban dőlttel, mint például A, B, C. Az algoritmusok kódjában a programozási alapszavakat félkövéren szedtük, mint például **if, then, else**.

Az algoritmusok nevében kiskötőjelet használtunk, viszont a változók neveiben – követve számos programozási nyelv szokásait – alsó kötőjelet. A változók neveit és a matematikai jelöléseket dőlt betűk emelik ki, például $\Omega(n \log n)$ vagy *pártatlan_ történet*. Az egyes fogalmak meghatározásának helyére a tárgymutató dőlt oldalszámmal utal.

Elsősorban az algoritmusokat tárgyaló tankönyvek matematikai jelöléseit alkalmaztuk. A ritkán használt jelöléseket az első fejezet végén és a tárgymutató elején is összefoglaltuk.

Az oldalszámok felsorolásánál – az eredeti könyvet követve – nem törekedtünk teljességre.