Tartalomjegyzék

Bevezetés

A Sudoku világában sokszor lehet szükségünk arra, hogy egy kitöltött rejtvényről eldöntsük, hogy az helyesen van-e kitöltve. Hasonló probléma a bűvös négyzet kitöltésének és az N-királynő kitöltésének az ellenőrzése is.

A dolgozatban belátjuk, hogy egy táblából az esetek nagy részében elegendő az első sort feldolgozni. Majd e szűkített feladat általánosításának megoldására alkotott algoritmusokat vizsgálunk műveletigény és futásidők tekintetében. Foglalkozunk a műveletigények legjobb, legrosszabb és átlagos esetének nagyságrendjével is. Ahol lehet, ott ezeket matematikai úton be is bizonyítjuk.

A matematikai mellett szimulációs módszerekkel is foglalkozunk, amelyek segítségével a nagyságrendekhez tartozó konstans szorzók értékét szeretnénk meghatározni.

A SudokuSolver program továbbfejlesztésének két módja a véletlenített rejtvény generálás és egy visszalépéses algoritmus megvalósítása. A rejtvény generátor elkészítésénél gyakorlati haszonnal is jár a helyességtesztelő algoritmusok elemzése. A visszalépéses algoritmusok közé tartozó X algoritmust is megismertetjük az olvasóval. Összevetjük a Rózsa algoritmusokkal, majd feltárjuk, hogy miért lehet érdemes a Knuth által javasolt DLX algoritmus helyett egy Rózsa alapokon nyugvó visszalépéses algoritmust választani.

SudokuSolver - Véletlen rejtvény generálása

Feladat:

Négyzetes mátrixok előállítása, ahol a négyzet oldalainak hossza négyzetszám. A mátrixnak n^4 eleme van, amelyek az $[1: n^2] \cup \{\varepsilon\}$ halmazból veszik fel az értékeiket. Ha az adott cella nincs kitöltve, azt jelöljük ε-nal! Az ε-tól különböző elemekre vannak megkötéseink. Egy házban (sorok, oszlopok, blokkok) sem szerepelhet egy szám sem többször. Ezt idáig egy megfejtett Sudoku rejtvény is teljesíti. Ahhoz, hogy ez egy olyan rejtvény legyen, amelyekkel az újságokban találkozunk, a tábla egyes celláinak tartalmát ki kell cserélni ε-nal. Ez egy kényes művelet, mert mindig meg kell nézni, hogy az adott elem törlése után még egyértelműen megoldható-e a rejtvény és milyen nehéz megoldani. Ha túl nehéznek, vagy nem egyértelműen megoldhatónak bizonyul a törlés után a rejtvény, azt az elemet nem szabad törölni. Persze egy rejtvény annál élvezhetőbb, minél több elemet kell benne nekünk kitölteni, tehát a legtöbb ε-ra kell törekedni.

Eredmények

Készítettem egy Rejtvény Generátor programot, amely képes előállítani rejtvényeket méghozzá úgy, hogy a rejtvények mérete és a megoldásban felhasználható „legizmosabb” algoritmus megadható. A generálást a program a bezárásáig végzi. Minden generált rejtvényt egy külön fájlban helyez el, amelynek neve a rejtvény típusából és a generálás időpontjából áll. A „legizmosabb” algoritmus megadása csak felső határ. Ez azt jelenti, hogy az előállított rejtvény nem nehezebb egy bizonyos szintnél, de könnyebb lehet nála.

A rejtvények generálási ideje a méret függvényében gyorsan nő.

- $4 * 4$-es rejtvények: a másodperc tört része alatt.
- $9 * 9$-es rejtvények: 4-6 másodperc alatt.
- $16 * 16$-os rejtvények: 3-10 perc alatt.
- $25 * 25$–ös rejtvények: legalább 2 óra, de volt, hogy 13 óra alatt sem sikerült.

A főprogramban a már elkészített rejtvények közül tudunk sorsolni. Itt lehetőségünk van a rejtvény nehézségét felülről és alulról is korlátozni. Így akár konkrét nehézségű rejtvényt is sorsoltathatunk magunknak, amennyiben olyat már sikerült generálni. A sorsolás mellett lehetőségünk van egy gyors generálásra is, de az előállítási időkre való tekintettel ezt csak $4 * 4$-es és $9 * 9$-es rejtvényekkel tehetjük meg.

Generálás módszere

A rejtvények előállítása két lépcsős folyamat. Először egy helyes kitöltést állítok elő, majd azt redukálom. A redukálás a beírt számok elhagyását jelenti, míg egy olyan állapotba nem érünk, ahonnan már egy számot sem lehet törölni anélkül, hogy a rejtvénynek több megoldása ne legyen. A módszer alapját Csizmazia Albert cikkében [4] olvastam, de attól lényegesen eltérő megvalósítást készítettem.

Helyes kitöltés előállítása

Három különböző módon próbáltam a kitöltéseket előállítani. A különböző módszereket azok megvalósítási sorrendjében írom le. Azért kellett három módszert megvalósítanom, mert az első kettő nem felelt meg az általam kijelölt követelményeknek, így nem árulok el azzal titkot, hogy a harmadik módszert használom végül a programomban.

Triviális helyes megoldás előállítása és annak transzformálása

Csizmazia Albert cikke alapján ez tűnt a legkézenfekvőbb megoldásnak, hiszen $9 * 9$-es Sudoku esetén egy konkrét kitöltésből a transzformációk segítségével nagyságrendileg 10^{12} db külön kitöltés állítható elő.

1. ábra: Triviálisan helyes kitöltés

1	2	3	4	5	6	7	8	9
4	5	6	7	8	9	1	2	3
7	8	9	1	2	3	4	5	6
2	3	4	5	6	7	8	9	1
5	6	7	8	9	1	2	3	4
8	9	1	2	3	4	5	6	7
3	4	5	6	7	8	9	1	2
6	7	8	9	1	2	3	4	5
9	1	2	3	4	5	6	7	8

Triviális helyes kitöltést (1. ábra) úgy kapunk, ha az első sorban növekvően elhelyezzük a számokat, majd a következő $(n-1)$ sorba úgy írjuk be az értékeket, hogy vesszük a kérdéses sor feletti sort, és jobbra eltoljuk n-nel. Így a felső n darab sor és blokk helyesen lesz kitöltve. Ez után az összes sort úgy töltjük ki, hogy az n sorral felette lévő sort eltoljuk jobbra 1-gyel. A blokkok és a sorok ezután is jók lesznek, és az ügyes eltolásnak köszönhetően az oszlopok is helyesek lesznek.

Ezek után 6 féle transzformációval alakíthatjuk a rejtvényünket. Mindegyik módszer mellé odaírom, hány féle előállítást lehet velő készíteni.

1. Transzponálás (2): Az oszlopok felcserélése a sorokkal, tükrözés a főátlóra.

2. Blokkszomszéd sorok permutálása ($n!^n$): Ha olyan sorokat cserélünk fel egymással, amelyeknek az elemei blokkszomszédok egymással, az a blokkok helyességét nem rontja el. Az oszlopok helyessége meg sorok cseréjétől nem tud elromlani. Ebben az esetben az első n darab sort egymás között szabadon cserélgethetjük, ami $n!$ lehetőséget rejt magában. Ettől függetlenül a többi $n-1$ sor n-essel és megtehetjük ezt, így az ezzel a módszerrel előállítható kitöltések száma $n!^n$.

3. Blokkszomszéd oszlopok permutálása ($n!^n$): Analóg módon az előzőhöz.

4. Blokksorok permutációja ($n!$): Egy blokksoron a vízszintesen egymás mellett lévő blokkokat értem. Ha két ilyen blokksort felcserélünk, akkor a blokkok helyesek maradnak. Mivel ez egy összetett sorcserének minősül, így se a sorokat, se az oszlopokat nem rontja el. Blokksorokból n darab van, így ez a módszer $n!$ új kitöltéssel jár.

5. Blokkoszlopok permutációja ($n!$): Analóg módón az előzőhöz.

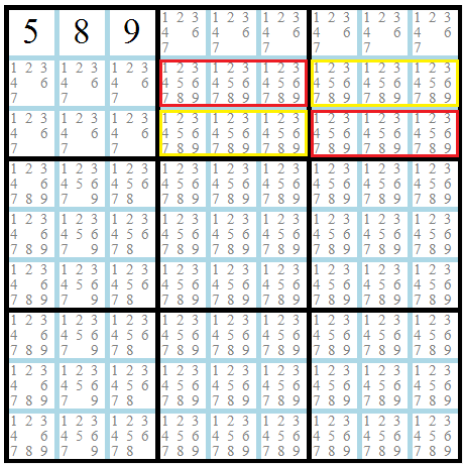
6. Elemek permutálása ($n^2!$): Felírjuk a számokat egy sorba növekvően, és alá ugyanezeknek a számoknak egy permutációját. Ezek után minden elemet a táblában lecserélünk úgy, hogy a felső sorban kikeressük az eredeti értéket és az alsó sorban alatta lévő értéket írjuk a helyére. Mivel a beírható számokból n^2 darab van, ez $n^2!$ új kitöltést eredményez.

Ezek a módszerek egymástól függetlenek, többszöri végrehajtásuk nem jár több eredménnyel, így a triviális helyes kitöltésünkből $2 * (n!^n)^2 * (n!^2) * (n^2!) = 2 * (n!)^{2n+2} * (n^2!)$ helyes kitöltést tudnak előállítani. Ez pontosan 1218998108160 darab helyes kitöltést eredményez egy $9 * 9$-es táblánál.

Az előállítás nagyon gyors, hiszen az egész előállítási procedúra műveletigénye $\theta(n^4)$ nagyságrendű.

Az ok, amiért ezt a módszert végül elvetettem, a rejtvények sablonossága volt. Szabad szemmel is könnyen észrevehető szabályszerűségek mutatkoztak az előállított rejtvényben. Sajnos mikor ezeket a sorokat írom, már nincs meg ennek a generátornak a forráskódja, így példával nem tudom alátámasztani az állításomat, de egy ábrával (2. ábra) illusztrálom, amit tapasztaltam.

2. ábra: Szabályszerűségek a transzformációk után



Ha a bal felső cellákban ezek a számok vannak, akkor vagy a piros vagy a sárga téglalapokban is ugyanezek az értékek fognak szerepelni, csak permutálva.

Hogy melyik színű téglalapba kerülnek, az a blokkoszlopok permutációjától és a blokkszomszéd sorok permutációjától függ, míg a téglalapon belüli permutáció a blokkszomszéd oszlopok permutációjától függ. A többi transzformáció erre a lehetőségre nincs hatással. A transzponálás ezt a helyzetet transzponálva megőrzi.

Ugyanez a szabályszerűség persze függőlegesen is megvan.

Bertram Felgenhauer és Frazer Jarvis sheffieldi matematikusok program segítségével kiszámították, hogy 6670903752021072936960 különböző helyes $(9 * 9$-es)

Sudoku kitöltés létezik [6]. Ha a kitöltött rejtvényeket úgy osztályozzuk, hogy egy osztály elemei egymásba átvihetők legyenek e transzformációk segítségével, akkor minden osztálynak ugyanannyi eleme van. Tehát $5472447994{,}27$ darab osztály van. Ez ellentmondás, hiszen az osztályok diszjunktak, így a számuk a hányados értékével egyenlő és egésznek kellene lennie.

Csizmazia Albert cikkében ez a szám 5472730538. Ha az összes kitöltést elosztom ezzel a számmal, akkor az ő általa számolt osztály elemszámot kapom. Érdekes módon ez a szám sem egészre jön ki. Ebből arra következtetek, hogy a két matematikus állítása pontatlan. Azt viszont nem értem, miért nem egyezik az én általam számolt érték és a cikkben található érték. Egyértelműen látszik, hogy nem kerekítésből ered az eltérés, ugyanakkor túlságosan is hasonlóak ahhoz, hogy más képletből szülessenek.

Véletlen táblakitöltés és gyors helyességtesztelés

A tábla minden cellájába azonos valószínűséggel írok bármilyen értéket, majd a helyességellenőrző algoritmusoknál vizsgáltak közül a leggyorsabbal, a MEMÓRIASZEMETES algoritmussal tesztelem, hogy helyes kitöltés-e.

A módszerrel végre felszámolható az előző módszerből eredendő sablonosság. Ugyanakkor a futási időre csak várhatóértékben lehet gondolni, ami jóval nagyobb az előzőnél.

Elkészítettem a programot, majd futtatáskor azt tapasztaltam, hogy nem talál helyes táblát. Ekkor álltam neki kiszámolni, hogy várhatóan hány próbálkozás után lesz helyes a rejtvény. Az összes kitöltések száma $9^{81} \sim 10^{77}$. Ebből a helyes kitöltések száma $6670903752021072936960 \sim 10^{21}$. Tehát durván várhatóan minden 10^{56}-odik kitöltés lenne helyes. Ezután megnéztem, hogy néhány másodperc alatt hányat próbál ki a programom. Ez a szám kétmillió körül volt, így nagyon rövid ideig tartó fejszámolás után úgy döntöttem, hogy ezt nem fogom kivárni.

Utólag átgondolva lehetett volna a programot úgy módosítani, hogy ne az egész táblát töltse fel újra, hanem egy vagy kevés véletlen elemet. Ekkor egy próba ideje leszorítható lett volna a tesztelés várható idejére, ami $\theta(n)$-es, az eddigi $\theta(n^4)$-hez képest. De mivel $n = 3$ esetén ez valószínűleg 10^2-t sem segített volna az előállítási időn, így marad az a konklúzió, hogy erre nem elég az én életem.

Véletlenített Visszalépéses algoritmus

A sorfolytonos visszalépéses algoritmust alkalmazom, ahol előbb az első soron, majd a többi soron megy végig balról jobbra és próbálja ki 1-től n^2-ig az értékeket. Amitől ez véletlenített lesz, az a permutációs mátrix. A mátrixnak n^4oszlopa és n^2 sora van. Minden oszlop az $[1: n^2]$ halmaz elemeinek egy permutációját tartalmazza. Az oszlopok függetlenek egymástól. Amikor az algoritmus elkezdi kitölteni az i-edik cellát, akkor oda a mátrix i-edik oszlopának megfelelő sorrendben próbálja beírni az értékeket.

Ezzel a módosítással is talál helyes kitöltést, sőt ugyanúgy megtalálja az összest, méghozzá ugyanannyi idő alatt. Viszont az első helyes kitöltés megtalálásának az ideje nagyon érzékeny a permutációs mátrix kitöltésére. Hogy ez mennyire sokat jelent, arra egy példát hozok fel. $25 * 25$-ös helyes kitöltést legenerált már 15 másodperc alatt is, de volt, hogy ez a művelet 13 óra alatt sem járt eredménnyel.

Mivel a cikkben visszalépéses algoritmussal hasznosítható sebességgel állítottak elő helyes kitöltéseket, az idővel úgy gondoltam nem lehet gondom. Ez az ominózus $25 * 25$-ös esettől eltekintve igaz is. Túlnyomó többségben ez a művelet még a $16 * 16$-os tábla esetén is a másodperc töredéke alatt lezajlik.

A programomban ugyan lehetőség nyílik a $25 * 25$-ös rejtvények kezelésére is, de a program fő célja a $9 * 9$-es rejtvények kezelése. A rejtvénybeli n növekedésével az azt megjelenítő felületnek az n^3szeresére kellene nőnie, ezért a program határait a megjelenítésből adódó korlátok szabták meg. A $25 * 25$-ös rejtvényt még benne hagytam a programban, de még senkitől sem láttam, hogy használta volna. Erre egy szemeszternyi ideje volt $30 - 40$ embernek, akik az ELTE-n felvették a Sudoku kurzust, ahol a visszajelzések alapján a többség az én programom nagyprogramként leadott verziójával dolgozott. Ezért ezt így elfogadható hiányosságnak tartom.

Redukció

A módszer nagyon egyszerű. Vegyük a tábla celláinak véletlen sorrendjét. Ebben a sorrendben próbálgatjuk törölni az elemeket. Minden törlés után le kell tesztelni, hogy az új rejtvény megoldható-e. Ha nem, akkor az utoljára törölt elemet vissza kell írni, és az már végleg a rejtvényben marad.

Az így előállított rejtvény redukált

Tételezzük fel, hogy nem az. Ez azt jelenti, hogy van egy olyan kitöltött cella, amelynek a tartalmát ki lehetne törölni és még mindig egy megoldása volna a rejtvénynek. Minden cellát megpróbáltunk egyszer kitörölni. Ha ez a cella ki van töltve, az azt jelenti, hogy az a rejtvény, amiben megpróbáltuk kitörölni, e cella nélkül több megoldással bírt volna. Mivel a végső rejtvényben csak olyan cellák vannak kitöltve, amik abban is ki voltak, így e cella nélkül a végső rejtvénynek is több megoldása kell, hogy legyen. Ez pedig ellentmondás, hiszen csak úgy törlünk, hogy a rejtvény egyértelmű megoldással rendelkezzen.

A redukcióban rejlő lehetőségek

A rejtvény megoldhatóságát nem egy visszalépéses algoritmussal tesztelem, hanem a programban már korábban megvalósított megoldó algoritmussal, amit úgy módosítottam, hogy be lehessen állítani, mely algoritmusokat használja. Így a redukció már nem csak egyértelműen megoldható rejtvényeket keres, hanem olyanokat, amelyeknek a megoldásához elegendő a megoldó algoritmusok egy konkrét halmaza.

Rejtvények nehézsége a statisztika tükrében

A Rózsa megoldó algoritmusoknak van egy meghatározott sorrendje. A rejtvény nehézségét a megoldáshoz felhasznált legnagyobb sorszámú algoritmus sorszámával jellemezzük. Az előállított rejtvények többségben $1, 2, 3, 4, 13, 19, 20$ nehézségű, és vannak olyan nehézségi szintek, amiket egyáltalán nem sikerült generálnom. Ebből arra következtetek, hogy ez a nehézségi rangsorolás nem felel meg a valóságnak.

Az alábbi táblázatban a $9 * 9$-es méretű rejtények nehézség szerinti eloszlása látható a rejtvény generátor által előállított rejtvények és Gordon Royle 17-es rejtvényei [7] esetén.

	Rejtvény generátor	Royle 17
A1 - TISZTA CELLA	60	0
A2 - REJTETT CELLA	141	21905
A3 - TÉGLA	124	15468
A4 - TISZTA PÁR	113	2383
A5 - REJTETT PÁR	87	1832
A6 - X2	34	17
A7 - TISZTA HÁRMAS	27	38
A8 - REJTETT HÁRMAS	8	16
A9 - TISZTA NÉGYES	1	3
A10 - REJTETT NÉGYES	0	0
A11 - X3	6	3
A12 - X4	0	0
A13 - Y2	116	1119
A14 - W2	64	40
A15 - Y3	2	2
A16 - W3	2	0
A17 - Y4	0	2
A18 - W4	0	0
A19 - FELHŐKARCOLÓ	119	2944
A20 - SÁRKÁNY	91	264
A21 - VISSZALÉPÉSES	0	3115
Összesen:	995	49151

Látható, hogy mindkét rejtvény gyűjtemény esetén ugyanazokból a nehézségi szintekből van a legkevesebb. A rejtvény generátor esetén azért nem annyira kiugró az $A2$ és $A3$ nézségű rejtvények darabszáma, mert a program típusonként 20 darabot tartott meg belőlük.

A mellékelt CD-n megtalálhatóak a táblázatban szereplő 50146 darab $9 * 9$-es és még 254 darab $4 * 4$-es, 337 darab $16 * 16$-os valamint 1 darab $25 * 25$-ös rejvény is.

Visszalépéses algoritmusok, Dancing Links

Használatuk oka, előnyeik, hátrányaik

A programba korábban beépített 20 algoritmusnak két nagy hiányossága van. Ezek az algoritmusok tényleges mintázatokat, helyzeteket ismernek fel, és ezekre tudnak egyértelmű lépést adni, ami a megoldás felé vezet. Mi a helyzet, ha a rejtvény megoldása közben egy olyan táblához jutottunk, ahol a fent említett mintázatok, helyzetek egyike sem található meg? Ekkor a megoldó algoritmusok azt a választ adják, hogy nem tudják, van-e a rejtvénynek megoldása. Ez gyakorlatilag jelentheti azt, hogy nincs, pontosan egy vagy akár több megoldás is van. Ezen kívül, ha a rejtvénynek több megoldása van, akkor lehet az akármilyen egyszerű is, az algoritmusok nem tudják megtalálni. Azok ugyanis csak olyan lépést hajtanak végre, ami egyértelműen benne lesz a megoldásban. Több megoldás esetén ezekkel az algoritmusokkal legfeljebb azokat a cellákat lehet kitölteni, amelyek az összes megoldásban megegyeznek.

Kézenfekvő megoldásnak kínálkoznak a visszalépéses algoritmusok. Ezeknek nagy előnye, hogy az összes megoldást megtalálják és a rejtvény nehézségétől függetlenül teszik ezt. Persze semmi sincs ingyen. A korábbi algoritmusok polinomiális futási idejével szemben ezek már exponenciális futási idejűek is lehetnek.

Felhasználás

Mindkét megoldásnak vannak komoly hátrányai. A korábbi algoritmusok nem mindig jutnak el a végeredményig, a visszalépéses meg nagyon lassú. Viszont, ha a két módszert vegyítjük, az segít a helyzeten.

Crook cikke [3] szerint érdemes a visszalépéses módszer minden lépése között lefuttatni a korábbi algoritmusokat. Így remélhetőleg azok sok munkát elvégezhetnek kevés idő alatt, míg a számukra „nehéz” helyzeteket bízzuk csak a visszalépéses algoritmusra.

A két módszer keverésével megőrizzük a visszalépéses algoritmus azon jó tulajdonságát, hogy mindig megtalálja az összes megoldást. A futási idő így is exponenciális marad, de az sokat javulhat. A visszalépéses algoritmusoknál a hatványalap az egy helyre beírható lehetőségek számával, míg a kitevő a még kitöltetlen helyek számával egyezik meg. Ha sok helyet kitöltetünk a korábbi

polinomiális algoritmusokkal, az a kitevő értékét csökkenti, és a program által generált naplók tanulsága szerint ezt igen hatékonyan végzik.

Dancing Links (DLX) – Donald E. Knuth

Többek között a Sudoku rejtvények megoldásához is alkalmas Knuth nem determinisztikus rekurzív X algoritmusa [12,14]. Az X algoritmussal a helyes elhelyezéssel kapcsolatos problémákat lehet megoldani. Ide tartozik a klasszikus N-királynő probléma is. Knuth saját megvalósítása az X algoritmushoz a DLX nevet kapta.

Mátrix felépítése

Az algoritmus hátterében egy mátrix van. Felépítését egy $4 * 4$-es Sudoku megoldásához szükséges mátrixon mutatom be. A mátrixnak minden sora egy cella végső állapotát jelenti. A mi esetünkben 64 sorból áll. Az első 4 sor, az 1. sor 1. cellájába írt különböző értékeknek felel meg, növekvő sorrendben. A második 4 sor az 1. sor 2. cellájába írandó értékeknek és az utolsó 4 sor a 4. sor 4. cellájába írandó értékeknek felel meg.

Az oszlopok felelnek meg a beírt értékek közötti kapcsolatnak. Ebben az esetben az oszlopok száma szintén 64. A cellákba vagy 0-t vagy 1-et írunk. A sorban csak azokra a helyekre kerül 1-es, amelyik jelölt beírása a kérdéses szabállyal kapcsolatos. Az oszlopok 4 féle szabályt valósítanak meg.

1. - 16. oszlop: Ezek a szabályok felelősek azért, hogy egy cellába ne írhassunk több értéket. Az 1. oszlop a sorfolytonos bejárásbeli első, a 16. az utolsó cellára vonatkozó szabályt írja le. Itt azokban a sorokban lesz 1-es, ahol az a kérdéses cellába való jelöltbeírást jelenti. Kitöltésük:

$$\forall i \in [1:64], \forall j \in [1:16]: A[i,j] = 1 \iff \left\lfloor \frac{i}{4} \right\rfloor = j$$

17. – 32. oszlop: Minden sorban minden érték egyszer szerepelhet. Ezek a szabályok felelősek azért, hogy ha egy sor egy cellájába beírtunk egy 3-ast, akkor a többibe már ne lehessen. A 17. oszlop a felelős az 1. sorba írt 1-esért, a 18. a 2-esért és végül a 32. oszlop a 4. sor 4-eséért. Kitöltésük:

$$\forall i \in [1:64], \forall j \in [17:32]:$$

$$A[i,j] = 1 <=> \left\lfloor \frac{i}{16} \right\rfloor = \left\lfloor \frac{j-16}{4} \right\rfloor \bigwedge i = j \text{ modulo } (4)$$

33. – 48. oszlop: Minden oszlopra is teljesülnie kell, hogy egy számot egyszer lehessen beírni. Az oszlopsorszámozás analóg módon történik a 2. 16 oszlopéval. Kitöltésük:

$$\forall i \in [1:64], \forall j \in [33:48]:$$

$$A[i,j] = 1 <=> \left\lfloor \frac{i}{4} \right\rfloor = \left\lfloor \frac{j-32}{4} \right\rfloor \text{ modulo } (4) \bigwedge i = j \text{ modulo } (4)$$

49. – 64. oszlop: Ennek a blokkokra is teljesülnie kell. Számozás szintén analóg. Kitöltés:

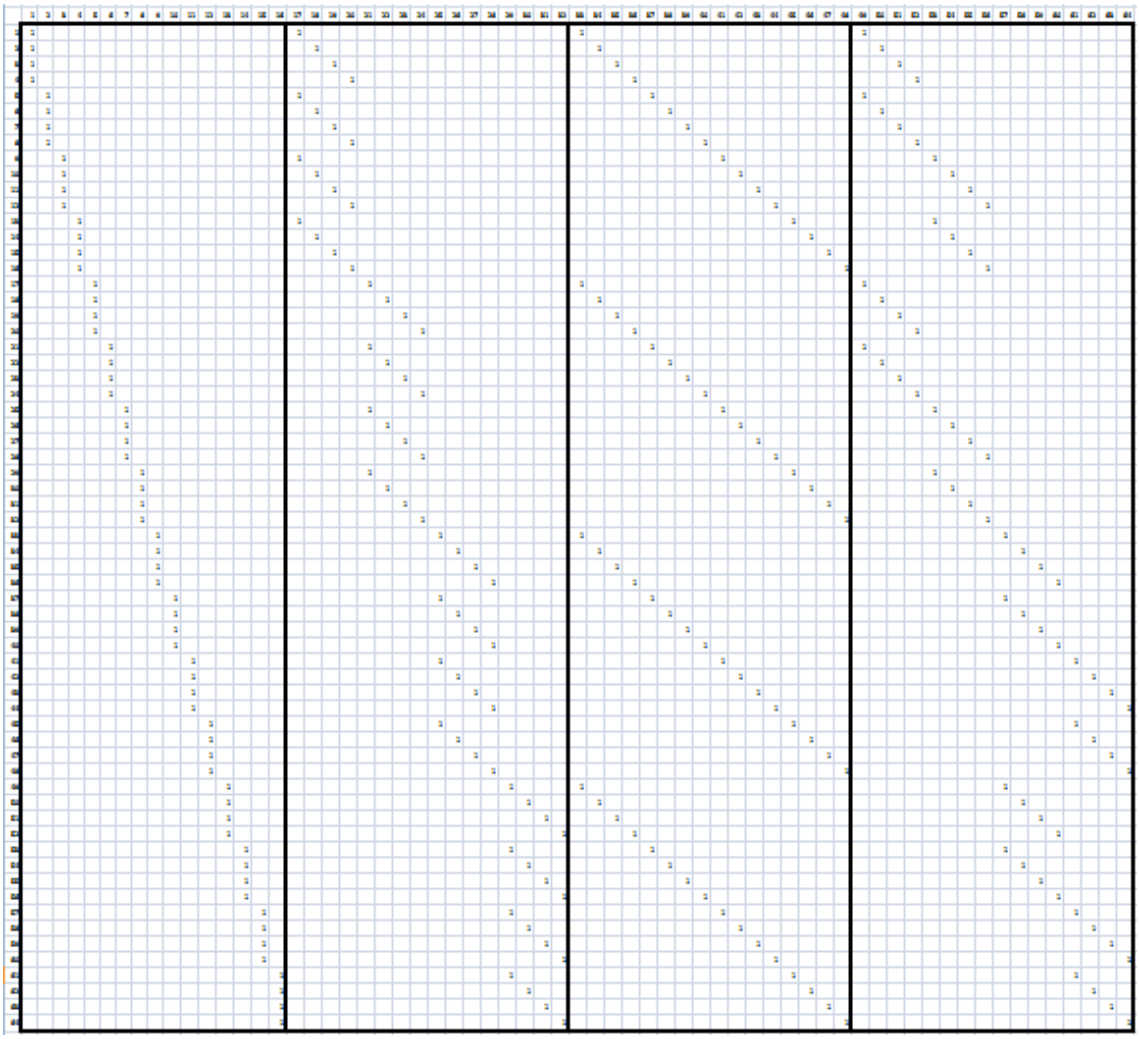
$$\forall i \in [1;64], \forall j \in [49;64]:$$

$$A[i,j] = 1 <=> i = j \; modulo \; (4) \bigwedge (\exists b \in [1;4]: \left\lfloor \frac{b}{2} \right\rfloor = \left\lfloor \frac{i}{16*2} \right\rfloor$$

$$\bigwedge b = \left\lfloor \frac{i}{16} \right\rfloor \text{ modulo } 2 \bigwedge b = \left\lfloor \frac{j-48}{4} \right\rfloor)$$

Ez a mátrix csak a $4 * 4$-es Sudoku esetén négyzetes. A mátrix sorainak száma n^6, míg oszlopainak száma $4 * n^4$. Ha például latin négyzet volna a megoldandó feladat, akkor a 4. oszlophalmaz nem kellene, hiszen ott nincsenek blokkok.

3. ábra: 4*4-es Sudoku tábla mátrixa



A kitöltetlen helyek felelnek meg a nulláknak. Ha jól megfigyeljük, minden sorban 4 darab 1-es van. Ez azért van, mert minden beírt szám 4 szabályban érinti társait. Minden oszlopban n^2 darab 1-es van. Ezek azokban a sorokban vannak, amik az adott szabály értelmében kizárják egymást a megoldásból.

Megoldás menete

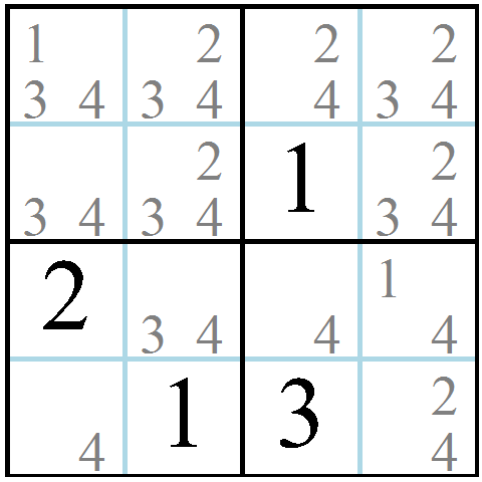
1. Keressük meg az első minimális összegű oszlopot. (j. oszlop) (determinisztikus)
2. Választunk egy sort, úgy hogy $A[i,j] = 1$ (i. sor) (nem determinisztikus)
3. Az i. sort helyezzük el a megoldásbeli sorok halmazában.
4. Vegyük az összes oszlopot, amiben 1-es van az i. sorban. (Ezekben a szabályokban érintett az i. sor) Ezután töröljük az összes olyan sort a

mátrixból, ahol a kérdéses oszlopok bármelyikében szerepel 1-es. (Vagyis töröljük a mátrixból az összes olyan sort, amit valamilyen szabály alapján kizár az i. sor)

5. Ha van még sorunk, folytassuk az 1. ponttól.

Rejtvény feldolgozása

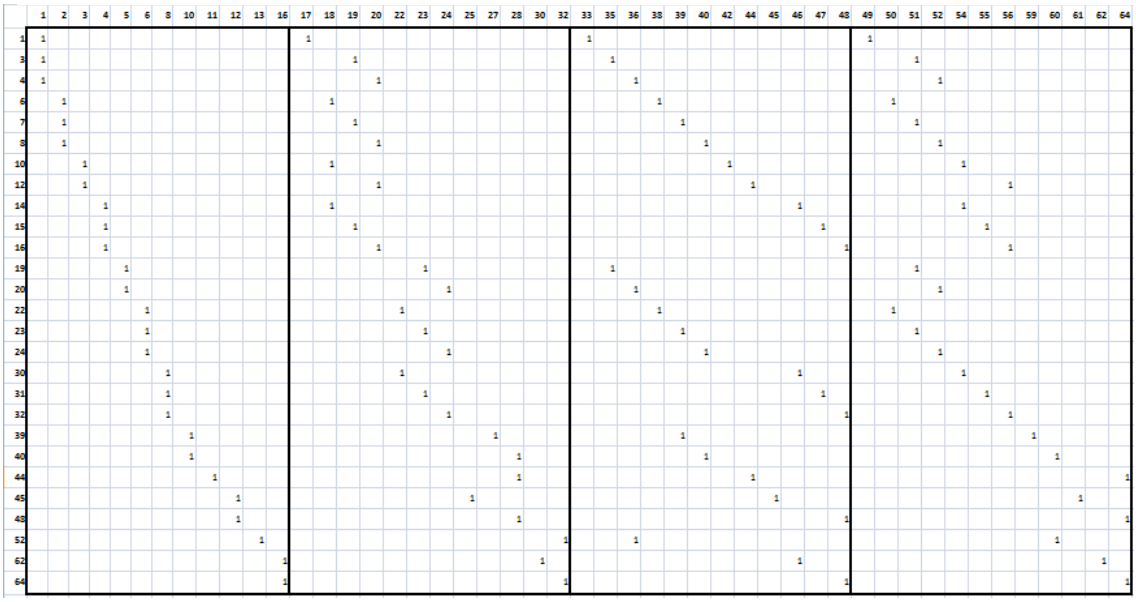
4. ábra: 4*4-es egyértelműen megoldható rejtvény



A továbbiakban ezt az egyértelműen megoldható rejtvényt dolgozom fel.

- Kezdjük az adatok beolvasásával! r2c3 = 1 => 25. sor. Ezt hozzávesszük a megoldáshoz. Ez törli a 26, 27, 28, 17, 21, 29, 9, 41, 57 és a 13. sort.
- Következő adat: 34. sor (r3c1 = 2). Törli: 33, 35, 36, 38, 42, 46, 2, 18, 50 és az 54. sort.
- 53. sor (r4c2 = 1). Törli: 55, 56, 49, 61, 5 és 37. sort.
- 59. sor (r4c3 = 3). Törli: 58, 60, 51, 63, 11, 43 és 47. sort.

5. ábra: Mátrix az adatok beolvasása után



- Most indul az algoritmus. Az oszlopok összegeinek 1 a minimuma. Az 1. 1 összegű oszlop a 11. oszlop. Ez azt jelenti, hogy az $r3c3$ cellába már csak egy megengedett jelölt van, így az egy tiszta cella. A kérdéses sor a 44., ami azt jelenti, hogy egy 4-es írunk be. Ez már a 4. ábra ábrán is látszódott.

- Az így kialakult mátrixban már a 3. oszlop a kiválasztandó és ez a 10. sort jelenti. Vagyis $r1c3 = 2$.

- Ezután sorban a $39, 8, 15, 1, 19, 22, 32, 45, 52$ és 62. sorok kerülnek a megoldásba.

6. ábra: A rejtvény megoldása

1	4	2	3
3	2	1	4
2	3	4	1
4	1	3	2

Visszalépés a DLX-ben

A példában nem fordult elő olyan eset, hogy az oszlopok összegeinek minimuma ne 1 legyen. Ebben az esetben az összes ott szóba kerülő sor mentén folytatni kell a megoldást.

Megoldhatatlanság észlelése

Fontosabb kérdés ez, mint elsőre gondolnánk, hiszen a visszalépéses algoritmus minden ágánál le kell tudni állni, még akkor is, ha ott nincs megoldás. Erre az algoritmus nagyon egyszerű megoldást kínál. A megoldásban akkor tudunk fennakadni, ha van olyan oszlop, ami mentén még nem töröltünk és mégis 0 az összege. Ez konkrétan azt jelenti, hogy már nincs olyan sorunk, ami annak a szabálynak megfelelne, vagyis megoldhatatlan a feladat. Pontosan ezért töröltem az oszlopokat is a korábbi ábrán, mert így könnyen követhető volt, hogy van-e üres oszlop. Ha ezt nem figyeljük, marad az algoritmus leállása. Ha ekkor a megoldásban nem annyi sor van, ahány cella, az ugyanúgy megoldhatatlanságot jelez, csak később vehető észre.

Kapcsolat a Rózsa algoritmusokkal

A mátrix első oszlophalmaza figyeli a cellákban megengedett jelöltek számát. A TISZTA CELLA algoritmus is pont ezt hívatott kezelni. A többi oszlop a különböző házakban figyeli a még be nem írt jelöltek számát. Ez meg pont a REJTETT CELLA algoritmus munkáját fedi le. Addig a pillanatig, amíg az algoritmus a visszalépéses részére nem kerül sor, ugyanazokat a számokat írja be ugyanoda a DLX és az első két Rózsa algoritmus is, csak más sorrendben.

X megvalósítása Rózsa algoritmusok segítségével

Ha a TISZTA és REJTETT CELLA algoritmushoz egy visszalépéses algoritmust párosítok, ami ugyanolyan módszerrel ágazik szét, mint a DLX, akkor kész is a saját X megvalósításom. Ezt a megvalósítást választotta Lakatos Zoltán is, akivel párhuzamosan dolgoztunk. Igaz neki ez a szakdolgozattémája [13]. A szakirodalom ezt a megközelítést Crook nevéhez fűzi [3].

Előnyök és hátrányok

Egyetlen hátrányként a visszalépés szintjein történő állapotmentés esetén kicsit megnövekedett memóriaigényt tudom felhozni. A két algoritmus hasonló futási

időkkel rendelkezik, hiszen mindegyik fő teendője a szabályok figyelése, amit hasonló módon látnak el.

Előnyként a korábban már megírt 20 algoritmus közül a maradék 18 felhasználásának lehetőségét hoznám fel. Elgondolkodtató a kérdés, hogy ez előny-e, hiszen a maradék 18 algoritmus nem ír be semmit. Ez igaz, viszont a jelöltek számát csökkentik, ami az adott szinten a visszalépéses algoritmus elágazásainak számát komolyan csökkentheti, akár olyan mértékben is, hogy elkerülhető legyen a visszalépés. Ezt a kérdést a programom felhasználója könnyedén megvizsgálhatja oly módon, hogy sorsoltat magának egy legalább A3 szintű rejtvényt. Ekkor megoldhatja a korábbi 20 algoritmus felhasználásával visszalépéses algoritmus nélkül, vagy csak az első 2 algoritmust használja, de ez esetben használja a visszalépéses algoritmust. Sok esetben érezhetően gyorsabb a 20 algoritmusból álló módszer, nem is beszélve arról, hogy a generált napló is jóval rövidebb.

DLX elhagyása és helyette saját algoritmus írása

Sokkal több lehetőséget rejtett magában, ha felhasználom korábbi algoritmusaimat, amit a későbbiekben bővítve még hatékonyabb megoldó algoritmust kaphatok. Ezért vetettem el a DLX megvalósításának lehetőségét annak ellenére, hogy a Témabejelentőmben még szerepelt.

A visszalépéses algoritmust viszont teljesen a DLX szellemében írtam meg, vagyis a legkisebb sorszámú minimális összegű oszlop alapján írok be értéket. Ezt úgy valósítom meg, hogy előbb megkeresem a kitöltetlen cellák közül sorfolytonos bejárásban az első minimális jelöltszámú cellát. Majd minden házat a sorok, oszlopok és blokkok sorrendjében összepárosítok a jelöltekkel növekvő sorrendben és ezek közül kiválasztom az első olyat, ahol a legkevesebb helyre írható be a jelölt a házban. Ha ez az érték a ház-jelölt párnál kisebb, akkor azt választom, egyébként a cellát. Majd az így szóba kerülő lehetőségeket járom be a visszalépéses algoritmussal, igaz én ezt determinisztikusan, a lehetőségek sorrendjében végzem el. Megemlíteném, hogy a két vizsgálati sorrend megegyezik a Tiszta és Rejtett Cella módszeres bejárási sorrendjével, így arra csak egy-egy minimumkeresést kellett ráültetnem.

Kitöltés helyességének tesztelése

Feladat meghatározása

Adott egy $n^2 * n^2$-es mátrix. Ennek minden eleme az $[1: n^2]$ halmazból veszi fel az értékét. Ebben a mátrixban házakat definiálunk. Háznak minősülnek a sorok, az oszlopok és a blokkok. Egy mátrix helyesen van kitöltve, ha minden házra teljesül az, hogy minden elem egyszer szerepel benne.

Ennek vizsgálatát végezhetjük úgy, hogy sorra vesszük az összes házat és egyenként megvizsgáljuk. Ha hibát találunk, akkor a többi házat nem vizsgáljuk.

Feltesszük azt, hogy az elemek véletlenszerűen vannak kitöltve, vagyis $\forall i, j, k \in [1: n^2]: P\{A[i, j] = k\} = \frac{1}{n^2}$. Ekkor annak valószínűsége, hogy egy ház helyesen van kitöltve: $\frac{(n^2)!}{(n^2)^{n^2}}$. Ez exponenciális sebességgel tart a 0-hoz. Ez azt jelenti, hogy a tesztelési idő túlnyomó része az első ház tesztelésével telik.

Ezért a későbbiekben csak egy ház tesztelésével foglalkozom. Itt viszont a feladat általánosításával.

Általánosított feladat

Adott egy n elemű S sorozat, amelynek az elemei az $[1: n]$ halmazból veszik fel az értékeiket. Egy sor kitöltése akkor helyes, ha $\forall i, j \in [1: n]: i \neq j \Rightarrow S[i] \neq S[j]$. Feladatunk egy sorozatról eldönteni, hogy helyes-e.

Eredmények, felhasználások

A rejtvénygeneráló programomban egy rejtvény előállítása két lépcsős folyamat. Előbb egy helyesen kitöltött rejtvényt állítok elő, majd abból törlök elemeket. A helyes kitöltés előállítására több módszert is kipróbáltam. Ezek közül az egyik véletlenszerűen kitöltött táblák előállítás volt addig, míg egyik helyes nem lesz. Itt a helyesség tesztelését az ebben a fejezetben részletezett algoritmusok egyikével végeztem.

A diplomamunkám írása közben Iványi Antal, a témavezetőm egy olyan cikken [10] dolgozott, amiben ilyen tesztelő algoritmusokat vizsgált. Ebben a témában rendszeresen konzultáltunk, melynek eredményeként született egy szimulációs

program és a HÁTRA algoritmus átlagos összehasonlítás számának linearitásáról szóló tételbizonyítás is.

Tesztelő algoritmusok definíciója

Az alábbi algoritmusok [9] mindegyikének az S sorozat és az n egész a bemenő adatai. Visszatérési értékük pedig egy logikai érték, ami akkor igaz, ha a sorozat helyes. A programokat az *Új algoritmusok* című könyvben [2] látható pszeudokóddal írom le.

HÁTRA

Ez az algoritmus úgy dönti el a sorozatról, hogy helyes-e, hogy minden elempárt összehasonlít. Ha valamelyik egyezik, akkor rögtön visszatér a hamis értékkel, ellenkező esetben az összes pár megvizsgálása után igaz értékkel tér vissza. A kérdéses cikkben a belső ciklus csökkenő sorrendben járta be a ciklusváltozót. Ez a szimulációnál és a bizonyításnál nem okoz különbséget, de a teljesség igénye miatt megemlítem.

programkód

1. **for** $i \leftarrow 2$ **to** n
2. **do for** j$\leftarrow 1$ **to** $i-1$
3. **do if** $S[i] = S[j]$
4. **then** return ↓
5. return ↑

ELŐRE

Nagyon hasonlít a HÁTRA algoritmushoz. Ugyanazokat a párokat vizsgálja meg, csak más sorrendben.

programkód

1. **for** $i \leftarrow 1$ **to** $n-1$
2. **do for** j$\leftarrow i+1$ **to** n
3. **do if** $S[i] = S[j]$
4. **then** return ↓
5. return ↑

LINEÁRIS

Ez az algoritmus abból indul ki, hogyha helytelen a sorozat, akkor van legalább egy elem, ami többször is előfordul a sorozatban. Ha valamelyik elemnek megtalálja a 2. előfordulását, akkor hamissal tér vissza. Mivel a számlálónk értéke 0 vagy 1 lehet, így elég lenne egy logikai változókból álló vektor is, de ez a vizsgálatok szempontjából érdektelen.

programkód

1. **for** $i \leftarrow 1$ **to** n
2. **do** $számláló[i] \leftarrow 0$
3. **for** $i \leftarrow 1$ **to** n
4. **do if** $számláló[S[i]] > 0$
5. **then** return ↓
6. **else** $számláló[S[i]] \leftarrow számláló[S[i]] + 1$
7. return ↑

KERESŐFÁS

Ez az algoritmus is egy elem második előfordulását keresi, de ez egy bináris keresőfában tárolja az elemeket. Egy bináris keresőfában minden csúcsból két részfa indulhat ki. A bal oldaliban vannak a csúcsban lévőnél kisebb, míg a jobb oldaliban a nagyobb elemek. A beszúrás művelet visszatérési értéke egy logikai érték, ami a beszúrás sikerességét fejezi ki. Ha az elem már a fában volt, akkor a beszúrás sikertelen.

programkód

1. **for** $i \leftarrow 1$ **to** n
2. **do if** $beszúrás(Fa, S[i]) = ↓$
3. **then** return ↓
4. return ↑

EDÉNYES

Az elemeket edényekben helyezi el. Egy edényben azonos tulajdonságú elemek vannak. Ez azért jó, mert annak eldöntéséhez, hogy egy elem szerepelt-e már, nem kell az eddigi összes elemmel összehasonlítani. Elég, ha azokkal hasonlítjuk össze, amelyekkel egy edénybe kerülne. Ha az edények száma megegyezik az elemek számával, akkor a lineáris algoritmust kapjuk. Ebben az algoritmusban $m = \lceil \sqrt{n} \rceil$

darab edény van és az elemek a modulo (m) értékük szerinti edénybe kerülnek. Az elemek egy $\lceil\sqrt{n}\rceil * \lceil\sqrt{n}\rceil$ -es mátrixban vannak tárolva. A mátrix sorai reprezentálják az edényeket. A sorok balról jobbra töltődnek fel, és minden sorhoz tartozik egy számláló, ami a sorban tárolt elemek számát tartja nyilván.

programkód

1. **for** $i \leftarrow 1$ **to** $\lceil\sqrt{n}\rceil$
2. **do** $számláló[i] \leftarrow 0$
3. **for** $i \leftarrow 1$ **to** n
4. **do** $edény\_indexe \leftarrow S[i]\ modulo\ (\lceil\sqrt{n}\rceil)$
5. **for** $j \leftarrow 1$ **to** $számláló[edény\_indexe]$
6. **do if** $edény[edény\_indexe, j] = S[i]$
7. **then** return ↓
8. $számláló[edény\_indexe] \leftarrow számláló[edény\_index] + 1$
9. $edény[edény\_indexe, számláló[edény\_indexe]] \leftarrow S[i]$
10. return true

MEMÓRIASZEMETES

Adott egy vektor, amely a sorozat elemeivel van indexelve. A vektor elemeiben azt tároljuk, hogy a sorozatban hányadik pozícióban található meg az elem. Ezt a vektort azonban nem töltjük fel extremális elemekkel az algoritmus többszöri lefutásakor. Az viszont biztos, hogy ha már dolgoztunk fel ilyen elemet, akkor a vektorban a megfelelő helyen annak az elemnek a sorozatbeli indexe lesz. Ha pedig nem, akkor akár még szerencsénk is lehet.

programkód

1. **for** $i \leftarrow 1$ **to** n
2. **do if** $helyek[S[i]] \neq i$ és $S[helyek[S[i]]] = S[i]$
3. **then** return ↓
4. **else** $helyek[S[i]] \leftarrow i$
5. return ↑

Műveletigények

A műveletigénynél azokat a lépéseket veszem számításba, amik az algoritmus elméleti futtatásához szükséges. Ez lényegében a ciklusváltozók kezelésén kívül mindent magába foglal.

HÁTRA

Az algoritmus csak összehasonlításokat végez. Legkedvezőbb esetben az első vizsgálandó párbeli elemek megegyeznek.

$$\mathrm{Min}_{\mathrm{HÁTRA}}(n) = 1 \in \Theta(1)$$

Legrosszabb esetben minden párt meg kell vizsgálni, erre sor is kerül, ha a sorozat helyes.

$$\mathrm{Max}_{\mathrm{HÁTRA}}(n) = \binom{n}{2} \in \frac{n^2}{2} + \theta(n) \subset \Theta(n^2)$$

1. tétel [9,10]:

$$\mathrm{Átl}_{\mathrm{HÁTRA}}(n) \in n + \Theta\left(\sqrt{n}\right)$$

A bizonyítás két lemmán [8] és a Stirling formulán alapul. A lemmákat Iványi Antal és Kátai Imre bizonyították be.

Lemmák:

$$(n \geq 1) => \left(\sum_{k=0}^{n-1} \frac{n^k}{k!} < \frac{e^n}{2}\right)$$

$$(n \geq 1) => \left(\sum_{k=0}^{n} \frac{n^k}{k!} > \frac{e^n}{2}\right)$$

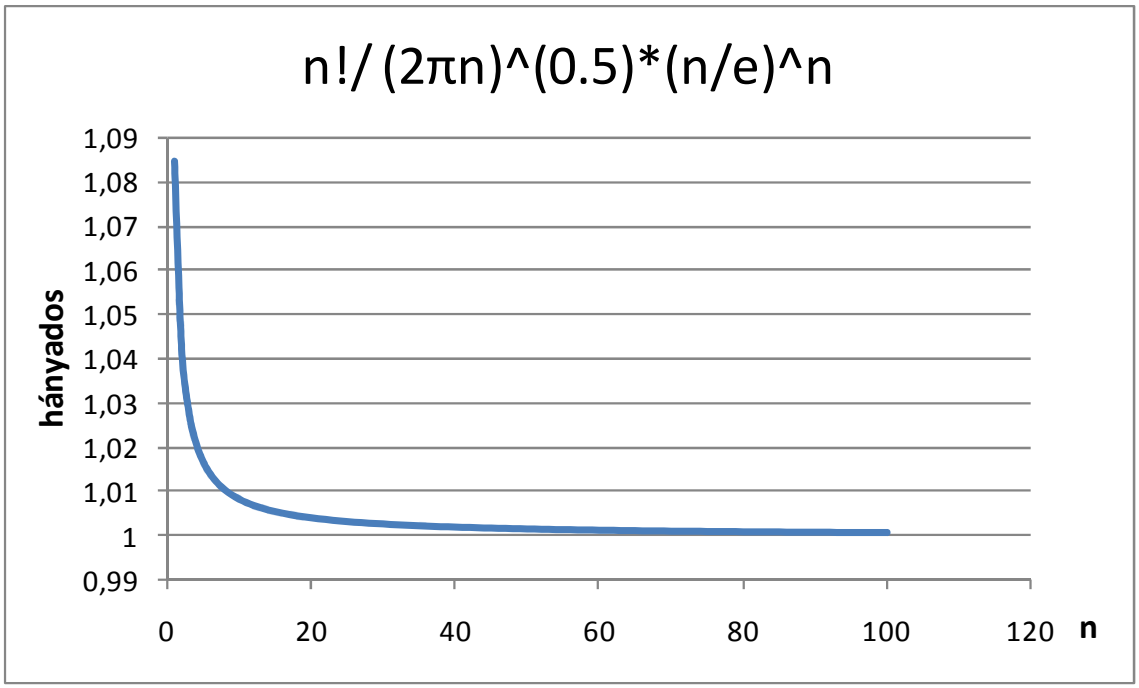
Stirling formula:

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

ahol a "~" jel aszimptotikus egyenlőséget jelent. Ennek magyarázata az alábbi képlet:

$$\lim_{n\to\infty} \frac{n!}{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n} = 1$$

7. ábra: Hányados alakulása



A görbe n = 9-re már 1,01 alatt van. Az *Új algoritmusok* című könyvben [2] az alábbi megközelítést alkalmazzák.

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n * e^{\alpha_n}$$

$$\frac{1}{12n+1} < \alpha_n < \frac{1}{12n} \quad (n \in \{1,2,3,4,\dots\})$$

Ezt felhasználva az alábbi becsléseket kapjuk.

$$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n * e^{\frac{1}{12n+1}} < n! < \sqrt{2\pi n}\left(\frac{n}{e}\right)^n * e^{\frac{1}{12n}}$$

1. tétel bizonyítása:

Az algoritmus abból indul ki, hogy az egy elemű sorozat helyes. Ez után csak hozzávesz új elemeket a helyes prefixhez, és a prefix minden korábbi elemével összehasonlítja.

Vizsgáljuk meg annak a valószínűségét, hogy egy sorozatnak *k* hosszúságú az a legnagyobb prefixe, amiben nincs ismétlődés.

$$(\forall i, j \in [1,k]: S[i] \neq S[j]) \wedge (\exists i \in [1,k]: S[i] = S[k+1])$$

$$P\{\alpha = k\} = \frac{n! * k}{(n-k)! * n^{k+1}} \quad \forall k \in [1:n]$$

A k természetesen nem lehet nagyobb n-nél, hiszen n féle elemből legfeljebb n hosszú ismétlődésmentes sorozatot lehet előállítani.

Ha a legnagyobb prefix k hosszú, akkor az elemeinek az átvizsgálására szükséges összehasonlítások számát az alábbi képlet írja le.

$$\sum_{i=2}^{k}(i-1) = \sum_{j=1}^{k-1} j = \frac{1+(k-1)}{2} * (k-1) = \frac{k^2-k}{2}$$

A $(k+1)$-edik elem fog megegyezni egy korábbi elemmel. Egyforma valószínűséggel bármelyik korábbi elemmel megegyezhet:

$$P\{\beta = i\} = \frac{1}{k}$$

A $(k+1)$-edik elem vizsgálatához az alábbi képlet adja meg az összehasonlítások számát.

$$\sum_{i=1}^{k} P\{\beta = i\} * i = \sum_{i=1}^{k} \frac{1}{k} * i = \frac{1}{k} * \frac{1+k}{2} * k = \frac{k+1}{2}$$

Az alábbi képlet a legfeljebb n hosszú helyes prefix megvizsgáláshoz szükséges összehasonlítások számának a várható értéket adja meg.

$$\sum_{k=1}^{n} P\{\alpha = k\} \left(\sum_{j=2}^{k}(j-1) + \sum_{i=1}^{k} i * P\{\beta = i\} \right)$$

Ezzel már csak az a gond, hogy a mi esetünkben a sorozat n hosszú, vagyis ha a sorozat helyes prefixe n hosszú, akkor nem kell megvizsgálnunk, hogy az $(n+1)$-edik elemben van-e ismétlődés.

A HÁTRA algoritmus várható összehasonlításainak számát az alábbi képlet írja le.

$$C = \sum_{k=1}^{n} P\{\alpha = k\}\left(\sum_{j=2}^{k}(j-1) + \sum_{i=1}^{k} i * P\{\beta = i\}\right) - P\{\alpha = n\} * \sum_{i=1}^{n} i * P\{\beta = i\}$$

$$C = \sum_{k=1}^{n} \frac{n! * k}{(n-k)! * n^{k+1}} * \left(\frac{k^2 - k}{2} + \frac{k+1}{2}\right) - \frac{n! * n}{(n-n)! * n^{n+1}} * \frac{n+1}{2}$$

$$C = \sum_{k=1}^{n} \frac{n! * k}{(n-k)! * n^{k+1}} * \left(\frac{k^2 + 1}{2}\right) - \frac{n!}{n^n} * \frac{n+1}{2}$$

A levezetést szétválasztom.

$$C := C_1 - C_2$$

$$C_1 = \sum_{k=1}^{n} \frac{n! * k}{(n-k)! * n^{k+1}} * \left(\frac{k^2 + 1}{2}\right)$$

$$C_2 = \frac{n!}{n^n} * \frac{n+1}{2}$$

Kezdjük a főtag levezetésével.

$$C_1 = \sum_{k=1}^{n} \frac{n! * k}{(n-k)! * n^{k+1}} * \left(\frac{k^2 + 1}{2}\right) = \sum_{k=1}^{n} \frac{n! * n^{n-k}}{(n-k)! * n^{n+1}} * \left(\frac{k^3 + k}{2}\right)$$

$$C_1 = \frac{n!}{2 * n^{n+1}} * \sum_{k=1}^{n} \frac{n^{n-k}}{(n-k)!} * (k^3 + k)$$

Térjünk át a k futóparaméterről j-re a $j = n - k$ behelyettesítés alapján!

$$C_1 = \frac{n!}{2 * n^{n+1}} * \sum_{j=0}^{n-1} \left[\frac{n^j}{(j)!} * \left((n-j)^3 + (n-j)\right)\right]$$

$$C_1 = \frac{n!}{2 * n^{n+1}} * \sum_{j=0}^{n-1} \left[\frac{n^j}{(j)!} * \left(-j^3 + 3n * j^2 - (3n^2 + 1) * j + (n^3 + n)\right)\right]$$

$$C_1 = \frac{n!}{2 * n^{n+1}} \left(-\sum_{j=0}^{n-1} \left[\frac{n^j}{(j)!} * j^3 \right] + 3n \sum_{j=0}^{n-1} \left[\frac{n^j}{(j)!} * j^2 \right] - (3n^2 + 1) \sum_{j=0}^{n-1} \left[\frac{n^j}{(j)!} * j \right] + (n^3 + n) \sum_{j=0}^{n-1} \frac{n^j}{(j)!} \right)$$

A továbbiakban 2 kifejezéshez a könnyebb olvashatóság érdekében betűt rendelek.

$$S_i(n) = S_i = \sum_{j=0}^{n-1} \left[\frac{n^j}{j!} * j^i \right] \quad i \in [0:3]$$

$$M(n) = M = \frac{n^n}{n!} = \frac{n^{n-1}}{(n-1)!}$$

$$C_1 = \frac{1}{2n * M} * (-S_3 + 3n * S_2 - (3n^2 + 1) * S_1 + (n^3 + n) * S_0)$$

$$S_1 = \sum_{j=0}^{n-1} \left[\frac{n^j}{(j)!} * j \right] = n \sum_{j=1}^{n-1} \frac{n^{j-1}}{(j-1)!} = n \sum_{i=0}^{n-2} \frac{n^i}{(i)!} = n \left(\sum_{i=0}^{n-1} \left[\frac{n^i}{(i)!} \right] - \frac{n^{n-1}}{(n-1)!} \right)$$

$$S_1 = n * S_0 - n * M$$

$$S_2 = \sum_{j=0}^{n-1} \left[\frac{n^j}{(j)!} * j^2 \right] = n \sum_{j=1}^{n-1} \left[\frac{n^{j-1}}{(j-1)!} * j \right] = n \sum_{i=0}^{n-2} \left[\frac{n^i}{(i)!} * (i+1) \right]$$

$$S_2 = n * \left(\sum_{i=0}^{n-1} \left[\frac{n^i}{(i)!} * (i+1) \right] - \frac{n^n}{(n-1)!} \right)$$

$$= n * \left(\sum_{i=0}^{n-1} \left[\frac{n^i}{(i)!} * i \right] + \sum_{i=0}^{n-1} \left[\frac{n^i}{(i)!} \right] \right) - n^2 * M$$

$$S_2 = n * (S_1 + S_0) - n^2 * M = n * (n * S_0 - n * M + S_0) - n^2 * M$$

$$S_2 = (n^2 + n) * S_0 - 2n^2 * M$$

$$S_3 = \sum_{j=0}^{n-1} \left[\frac{n^j}{(j)!} * j^3 \right] = n * \sum_{j=1}^{n-1} \left[\frac{n^{j-1}}{(j-1)!} * j^2 \right] = n * \sum_{i=1}^{n-2} \left[\frac{n^i}{(i)!} * (i+1)^2 \right]$$

$$S_3 = n * \left(\sum_{i=1}^{n-1} \left[\frac{n^i}{(i)!} * (i+1)^2 \right] - \frac{n^{n-1}}{(n-1)!} * (n)^2 \right)$$

$$S_3 = n * \left(\sum_{i=1}^{n-1} \left[\frac{n^i}{(i)!} * (i^2 + 2i + 1) \right] \right) - n^3 * M = n * (S_2 + 2S_1 + S_0) - n^3 * M$$

$$S_3 = n * [(n^2 + n) * S_0 - 2n^2 * M + 2(n * S_0 - n * M) + S_0] - n^3 * M$$

$$S_3 = (n^3 + 3n^2 + n)S_0 - (3n^3 + 2n^2) * M$$

Ezeket visszahelyettesítjük a korábbi képletbe.

$$C_1 = \frac{1}{2n * M} * (-S_3 + 3n * S_2 - (3n^2 + 1) * S_1 + (n^3 + n) * S_0)$$

$$\begin{aligned} C_1 = \frac{1}{2n * M} & \\ & * \left[-\left((n^3 + 3n^2 + n)S_0 - (3n^3 + 2n^2) * M \right) + 3n \right. \\ & * \left((n^2 + n) * S_0 - 2n^2 * M \right) - (3n^2 + 1) * (n * S_0 - n * M) \\ & \left. + (n^3 + n) * S_0 \right] \end{aligned}$$

$$C_1 = \frac{1}{2n * M} * [(2n^2 + n) * M - n * S_0] = \frac{(2n+1) * M - S_0}{2 * M} = n + \frac{1}{2} - \frac{1}{2} * \frac{S_0}{M}$$

A sor legvégén lévő hányadost a lemmák segítségével felülről és alulról is tudom becsülni.

$$\frac{S_0}{M} = \frac{\sum_{j=0}^{n-1} \frac{n^j}{j!}}{\frac{n^n}{n!}} = \frac{n!}{n^n} \sum_{j=0}^{n-1} \frac{n^j}{j!}$$

Felhasználom az 1. lemmát.

$$\frac{S_0}{M} = \frac{n!}{n^n} \sum_{j=0}^{n-1} \frac{n^j}{j!} < \frac{n!}{n^n} * \frac{e^n}{2}$$

Az alsó becsléshez térjünk vissza oda, ahol az 1. lemmát behelyettesítettük!

$$\frac{S_0}{M} = \frac{n!}{n^n}\sum_{j=0}^{n-1}\frac{n^j}{j!} = \frac{n!}{n^n}\left(\sum_{j=0}^{n}\left[\frac{n^j}{j!}\right] - \frac{n^n}{n!}\right) = \frac{n!}{n^n}\sum_{j=0}^{n}\left[\frac{n^j}{j!}\right] - 1$$

Felhasználom a 2. lemmát.

$$\frac{S_0}{M} = \frac{n!}{n^n}\sum_{j=0}^{n}\left[\frac{n^j}{j!}\right] - 1 > \frac{n!}{n^n} * \frac{e^n}{2} - 1$$

Legyen T a mindkét becslésben megtalálható közös tag!

$$T := \frac{n!}{n^n} * \frac{e^n}{2}$$

$$T - 1 < \frac{S_0}{M} < T$$

T értékét a Stirling formula segítségével tudom becsülni.

$$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n * e^{\frac{1}{12n+1}} < n! < \sqrt{2\pi n}\left(\frac{n}{e}\right)^n * e^{\frac{1}{12n}}$$

$$\frac{\sqrt{2\pi n} * \left(\frac{n}{e}\right)^n * e^{\frac{1}{12n+1}}}{n^n} * \frac{e^n}{2} < T = \frac{n!}{n^n} * \frac{e^n}{2} < \frac{\sqrt{2\pi n} * \left(\frac{n}{e}\right)^n * e^{\frac{1}{12n}}}{n^n} * \frac{e^n}{2}$$

$$\frac{\sqrt{2\pi}}{2} * \sqrt{n} * \sqrt[12n+1]{e} = \frac{\sqrt{2\pi n} * e^{\frac{1}{12n+1}}}{2} < T < \frac{\sqrt{2\pi n} * e^{\frac{1}{12n}}}{2} = \frac{\sqrt{2\pi}}{2} * \sqrt{n} * \sqrt[12n]{e}$$

$$\frac{\sqrt{2\pi}}{2} * \sqrt{n} * \sqrt[12n+1]{e} - 1 < \frac{S_0}{M} < \frac{\sqrt{2\pi}}{2} * \sqrt{n} * \sqrt[12n]{e}$$

Ezt visszahelyettesítjük C_1 képletébe.

$$n + \frac{1}{2} - \frac{1}{2} * \left(\frac{\sqrt{2\pi}}{2} * \sqrt{n} * \sqrt[12n]{e}\right) < C_1 < n + \frac{1}{2} - \frac{1}{2} * \left(\frac{\sqrt{2\pi}}{2} * \sqrt{n} * \sqrt[12n+1]{e} - 1\right)$$

$$n - \frac{\sqrt{2\pi}}{4} * \sqrt{n} * \sqrt[12n]{e} + \frac{1}{2} < C_1 < n - \frac{\sqrt{2\pi}}{4} * \sqrt{n} * \sqrt[12n+1]{e} + 1$$

A maradéktag exponenciális sebességgel tart a 0-hoz, így a tételt bebizonyítottuk, de célunk egy alsó és felső becslés előállítása is az átlagos összehasonlításszámra. A maradéktag levezetésével folytatom és rögtön alkalmazom a Stirling formulát.

$$\frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n * e^{\frac{1}{12n+1}}}{n^n} * \frac{n+1}{2} < C_2 = \frac{n!}{n^n} * \frac{n+1}{2} < \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n * e^{\frac{1}{12n}}}{n^n} * \frac{n+1}{2}$$

$$\frac{\sqrt{2\pi n} * e^{\frac{1}{12n+1}}}{e^n} * \frac{n+1}{2} < C_2 < \frac{\sqrt{2\pi n} * e^{\frac{1}{12n}}}{e^n} * \frac{n+1}{2}$$

$$C_2 > \frac{\sqrt{2\pi}}{2} * \frac{\sqrt{n^3} * e^{\frac{1}{12n+1}}}{e^n} + \frac{\sqrt{2\pi}}{2} * \frac{\sqrt{n} * e^{\frac{1}{12n+1}}}{e^n}$$

$$C_2 < \frac{\sqrt{2\pi}}{2} * \frac{\sqrt{n^3} * e^{\frac{1}{12n}}}{e^n} + \frac{\sqrt{2\pi}}{2} * \frac{\sqrt{n} * e^{\frac{1}{12n}}}{e^n}$$

Ezt behelyettesítjük a C-re vonatkozó képletbe.

$$C > n - \frac{\sqrt{2\pi}}{4} * \sqrt{n} * \sqrt[12n]{e} + \frac{1}{2} - \left(\frac{\sqrt{2\pi}}{2} * \frac{\sqrt{n^3} * e^{\frac{1}{12n}}}{e^n} + \frac{\sqrt{2\pi}}{2} * \frac{\sqrt{n} * e^{\frac{1}{12n}}}{e^n}\right)$$

$$C < n - \frac{\sqrt{2\pi}}{4} * \sqrt{n} * \sqrt[12n+1]{e} + 1 - \left(\frac{\sqrt{2\pi}}{2} * \frac{\sqrt{n^3} * e^{\frac{1}{12n+1}}}{e^n} + \frac{\sqrt{2\pi}}{2} * \frac{\sqrt{n} * e^{\frac{1}{12n+1}}}{e^n}\right)$$

$$C > n - \frac{\sqrt{2\pi}}{4} * \sqrt{n} * \sqrt[12n]{e} - \frac{\sqrt{2\pi}}{2} * \frac{\sqrt{n^3} * e^{\frac{1}{12n}}}{e^n} - \frac{\sqrt{2\pi}}{2} * \frac{\sqrt{n} * e^{\frac{1}{12n}}}{e^n} + \frac{1}{2}$$

$$C < n - \frac{\sqrt{2\pi}}{4} * \sqrt{n} * \sqrt[12n+1]{e} - \frac{\sqrt{2\pi}}{2} * \frac{\sqrt{n^3} * e^{\frac{1}{12n+1}}}{e^n} - \frac{\sqrt{2\pi}}{2} * \frac{\sqrt{n} * e^{\frac{1}{12n+1}}}{e^n} + 1$$

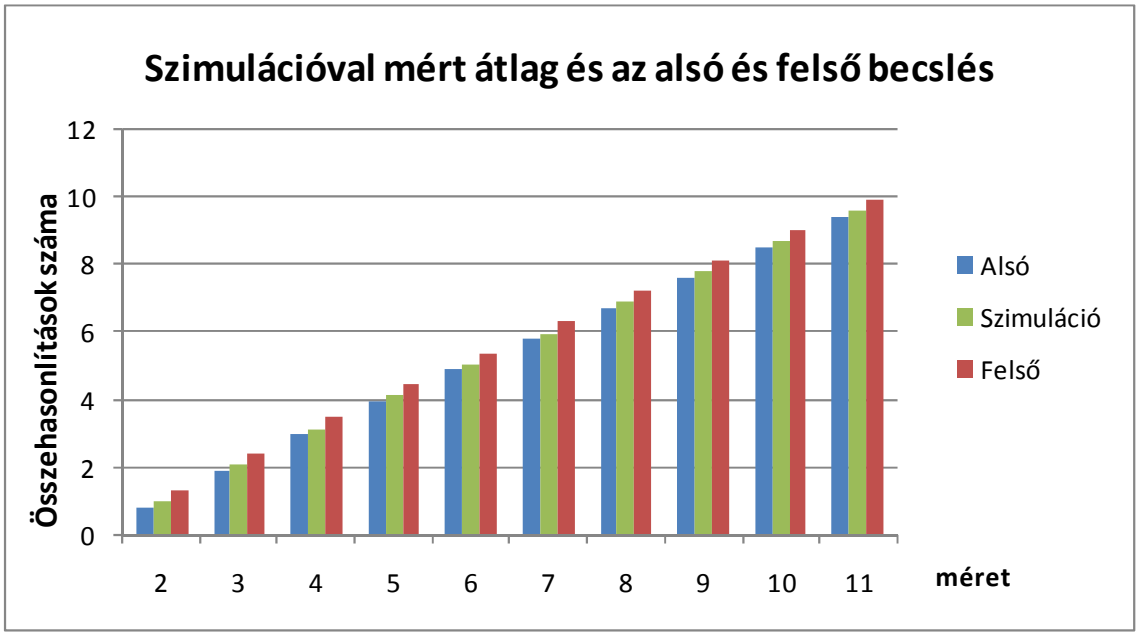
A C valójában egy sorozat, melynek minden tagját felülről és alulról is becsültünk egy $n + \theta(\sqrt{n})$-es sorozattal, így maga a sorozat is ilyen.

$$C \in n + \theta(\sqrt{n})$$

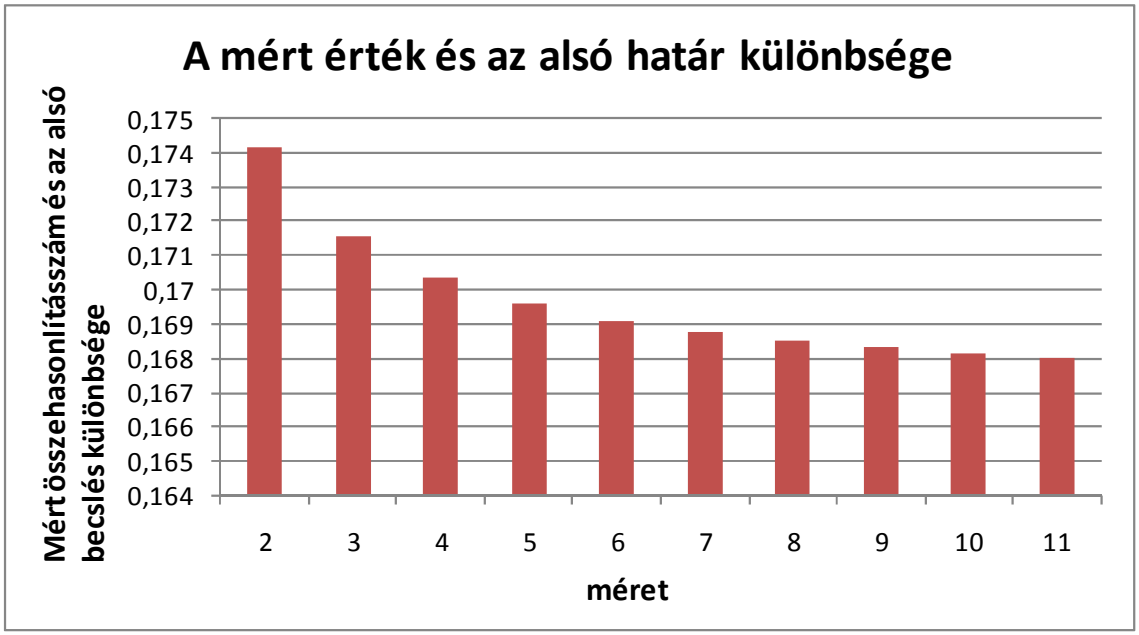
1. tétel bebizonyítva.

Az alábbi diagramok (8. ábra és 9. ábra) a levezetés végén kihozott eredmények és a szimulációs program által az összes lehetséges bemenetre elvégzett összehasonlítások átlagos száma közötti kapcsolatot mutatják.

8. ábra: HÁTRA műveletigénye és a felső és alsó becslés



9. ábra: HÁTRA műveletigénye és a becslés különbsége



ELŐRE

A minimális és a maximális műveletigény megegyezik a HÁTRA algoritmus értékeivel, így azok nagyságrendjei is.

$$\mathrm{Min}_{\mathrm{ELŐRE}}(n) = 1 \in \Theta(1)$$

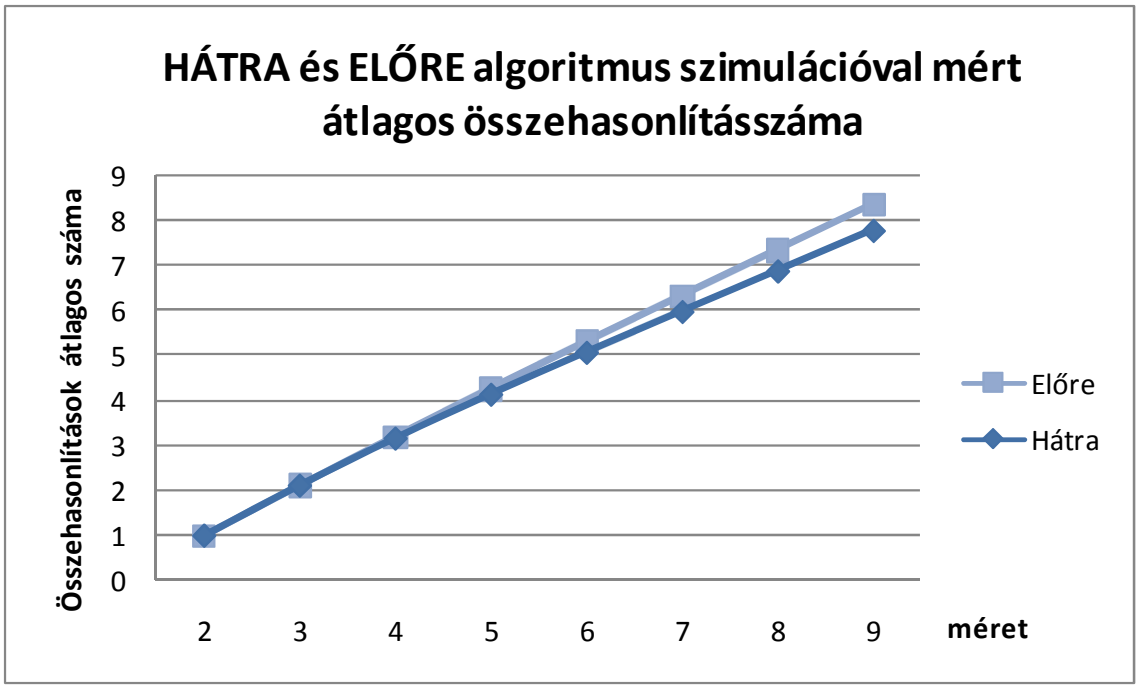
$$\mathrm{Max}_{\mathrm{ELŐRE}}(n) = \binom{n}{2} \in \frac{n^2}{2} + \theta(n) \subset \Theta(n^2)$$

Az átlagos műveletigény nagyságrendje szintén lineáris, de az átlag nem egyezik a HÁTRA algoritmuséval.

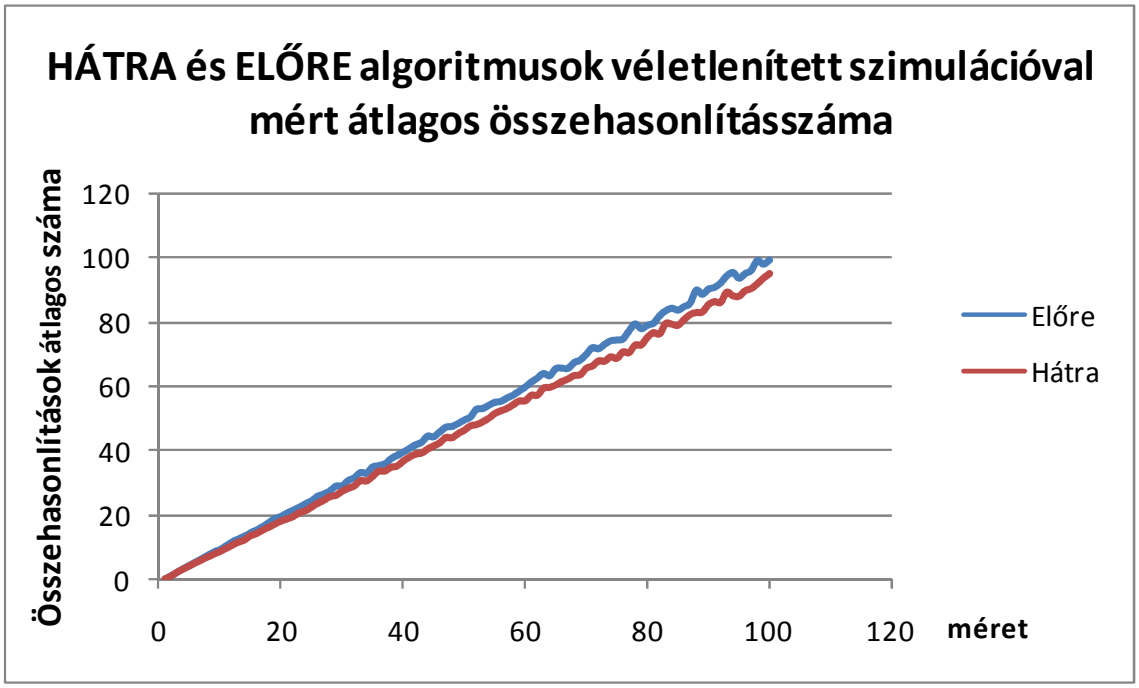
$$\text{Átl}_{\text{ELŐRE}}(n) \in \Theta(n)$$

A 10. ábra ábrán az összes lehetséges bemenetre kiértékelt értékek vannak, míg a 11. ábra ábrán 1-től 100-ig minden méretre 10000 darab véletlenszerűen generált sorozat megvizsgálásának átlagos összehasonlításszáma látható.

10. ábra: HÁTRA - ELŐRE



11. ábra: HÁTRA - ELŐRE véletlenített szimulációval



A különbség mindkét ábrán jól látható. A nagyobb méreteknél a lineáris függvény eltorzul. Ez azért van, mert a méret növekedésével exponenciálisan nő a variációk száma, és ezzel egyre nő a véletlen variációkból eredő átlagos érték ingadozása. Hasonló diagramot próbáltam előállítani 100-tól 2000-ig terjedő sorozathosszokra is, de ott már olyan nagy volt az ingadozás, hogy nem volt látható melyik algoritmus a jobb.

A két algoritmus ugyanazokat a vizsgálatokat végzi el, csak más sorrendben. Pontosan emiatt hittem sokáig azt, hogy a szimulációm hibás. A szimuláció eredményei az alábbi táblázatban láthatóak.

n	Darab	Helyes	ELŐRE	HÁTRA
2	4	2	1	1
3	27	6	2,111111	2,111111
4	256	24	3,203125	3,15625
5	3125	120	4,264	4,1296
6	46656	720	5,302341	5,058642
7	823543	5040	6,32676	5,966451
8	16777216	40320	7,342926	6,866676
9	387420489	362880	8,354165	7,766159

A különbség $n \geq 4$ feltétel teljesülésekor áll fenn. $n = 4$-re ez már papíron is nyomon követhető. Vegyük az összes (4^4 darab) mintát, és csoportosítsuk őket úgy, hogy az egy csoportba eső mintákra a felhasznált összehasonlítások száma egyezzen meg algoritmusonként. Az $\{a, b, c, d\}$ halmaz elemeivel jelölöm a számokat. Mindegyik jelenthet bármilyen beírt számot, de ha egy betű többször fordul elő a mintában, akkor azok ugyanazt a számot jelentik, és különböző betűk különböző számokat jelölnek. X-szel jelöljük azt, ha az adott helyen mindegy, hogy mi van, és a'-vel, azt hogy az adott helyen az a-val jelölt számon kívül bármi szerepelhet.

Minta		aaXX	abaX	abba'	abba	abca	abcb	abcc	abcd	
Darab		64	48	36	12	24	24	24	24	256
Mintánkénti összehasonlítás	ELŐRE	1	2	**4**	3	**3**	5	6	6	
	HÁTRA	1	2	**3**	3	**4**	5	6	6	
Összes összehasonlítás	ELŐRE	64	96	**144**	36	**72**	120	144	144	**820**
	HÁTRA	64	96	**108**	36	**96**	120	144	144	**808**

$$\text{Átl}_{\text{ELŐRE}}(4) = \frac{820}{256} = 3{,}203125$$

$$\text{Átl}_{\text{HÁTRA}}(4) = \frac{808}{256} = 3{,}15625$$

Tehát a táblázat is kihozta a szimuláció eredményét.

LINEÁRIS

Az algoritmus n darab értékadással kezd. A legjobb eset az, ha az első két elem megegyezik. Az első elem feldolgozása során egy összehasonlítás és egy újabb értékadás történik. A második elemnél elég egy összehasonlítás is.

$$\text{Min}_{\text{LINEÁRIS}}(n) = n + 3 \in \Theta(n)$$

A legrosszabb eset az, ha a sorozat helyes, ekkor minden elem feldolgozása egy összehasonlításba és egy értékadásba kerül.

$$\text{Max}_{\text{LINEÁRIS}}(n) = 3n \in \Theta(n)$$

2. tétel:

$$\text{Átl}_{\text{LINEÁRIS}}(n) \in n + \theta(\sqrt{n})$$

2. tétel bizonyítása levezetéssel:

Az összehasonlítások átlagos számát az alábbi levezetéssel lehet becsülni. A HÁTRA algoritmushoz hasonlóan az algoritmus addig dolgozik, míg rá nem jön, milyen hosszú a leghosszabb helyes prefix. A különbség annyi, hogy k hosszú prefix esetén $k+1$ összehasonlításra és $n+k$ értékadásra van szükség. És hasonlóan a HÁTRA algoritmushoz, itt is le kell vonni n hosszú prefix esetén a $+1$ összehasonlítást.

Kezdem az összehasonlítások átlagos számával.

$$C_{ö} = \sum_{k=1}^{n} \left[\frac{n! * k}{(n-k)! * n^{k+1}} * (k+1) \right] - \frac{n! * n}{(n-n)! * n^{n+1}} * (1)$$

$$C_{ö} = \frac{n!}{n^{n+1}} \sum_{k=1}^{n} \left[\frac{n^{n-k}}{(n-k)!} * k(k+1) \right] - \frac{n!}{n^{n}} = \frac{1}{n * M} \sum_{k=1}^{n} \left[\frac{n^{n-k}}{(n-k)!} * k(k+1) \right] - \frac{n!}{n^{n}}$$

$$C_{ö} = \frac{n!}{n^{n+1}} \sum_{i=0}^{n-1} \left[\frac{n^{i}}{i!} * (n-i)(n+1-i) \right] - \frac{n!}{n^{n}}$$

$$C_{ö} = \frac{n!}{n^{n+1}} \sum_{i=0}^{n-1} \left[\frac{n^{i}}{i!} * \left(i^2 - (2n+1)i - (n^2+n) \right) \right] - \frac{n!}{n^{n}}$$

Felhasználom a HÁTRA algoritmusnál a levezetésbeli S_i és M elnevezéseket és az azokból levezetett képleteket.

$$S_i = \sum_{j=0}^{n-1} \left[\frac{n^{j}}{j!} * j^{i} \right] \quad i \in [0:2]$$

$$M = \frac{n^{n}}{n!} = \frac{n^{n-1}}{(n-1)!}$$

$$C_{ö} = \frac{1}{n * M} (S_2 - (2n+1)S_i + (n^2+n)S_0) - \frac{n!}{n^{n}}$$

$$S_1 = n * S_0 - n * M$$

$$S_2 = (n^2+n) * S_0 - 2n^2 * M$$

$$C_{ö} = \frac{1}{n * M}\left(\left((n^2 + n) * S_0 - 2n^2 * M\right) - (2n + 1)(n * S_0 - n * M) + (n^2 + n)S_0\right) - \frac{n!}{n^n}$$

$$C_{ö} = \frac{1}{n * M}(n * S_0 + n * M) - \frac{n!}{n^n} = \frac{S_0}{M} + 1 - \frac{n!}{n^n}$$

A korábbi bizonyításból egy újabb eredményt is újra felhasználhatunk.

$$\frac{\sqrt{2\pi}}{2} * \sqrt{n} * \sqrt[12n+1]{e} - 1 < \frac{S_0}{M} < \frac{\sqrt{2\pi}}{2} * \sqrt{n} * \sqrt[12n]{e}$$

A sor végén lévő kifejezést a Stirling formula segítségével becsülhetjük.

$$\frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n * e^{\frac{1}{12n+1}}}{n^n} < \frac{n!}{n^n} < \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n * e^{\frac{1}{12n}}}{n^n}$$

$$\sqrt{2\pi} * \frac{\sqrt{n} * e^{\frac{1}{12n+1}}}{e^n} < \frac{n!}{n^n} < \sqrt{2\pi} * \frac{\sqrt{n} * e^{\frac{1}{12n}}}{e^n}$$

Ezeket behelyettesítve az alábbi egyenlőtlenségeket kapjuk.

$$C_{ö} > \frac{\sqrt{2\pi}}{2} * \sqrt{n} * \sqrt[12n+1]{e} - \sqrt{2\pi} * \frac{\sqrt{n} * e^{\frac{1}{12n}}}{e^n}$$

$$C_{ö} < \frac{\sqrt{2\pi}}{2} * \sqrt{n} * \sqrt[12n]{e} - \sqrt{2\pi} * \frac{\sqrt{n} * e^{\frac{1}{12n+1}}}{e^n} + 1$$

Tehát az összehasonlítások átlagos száma gyökös nagyságrendű. Hasonló levezetéssel becsülhetjük az értékadások átlagos számát is.

$$C_{é} = n + \sum_{k=1}^{n}\left[\frac{n! * k}{(n-k)! * n^{k+1}} * k\right] = n + \frac{1}{n * M}\sum_{k=1}^{n}\frac{n^{n-k}}{(n-k)!} * k^2$$

$$C_{é} = n + \frac{1}{n * M}\sum_{i=0}^{n-1}\frac{n^i}{i!} * (n-i)^2 = n + \frac{1}{n * M}(S_2 - 2nS_1 + n^2S_0)$$

$$C_{é} = n + \frac{1}{n * M}\left((n^2 + n) * S_0 - 2n^2 * M - 2n(n * S_0 - n * M) + n^2S_0\right)$$

$$C_{é} = n + \frac{1}{n * M}(n * S_0) = n + \frac{S_0}{M}$$

$$n + \frac{\sqrt{2\pi}}{2} * \sqrt{n} * \sqrt[12n+1]{e} - 1 < C_{é} < n + \frac{\sqrt{2\pi}}{2} * \sqrt{n} * \sqrt[12n]{e}$$

Látható, hogy az értékadások átlagos száma is gyökös nagyságrendű lenne, ha az elején nem kellene inicializálni a tömböt, így azonban ez lineáris nagyságrendű.

$$Átl_{LINEÁRIS}(n) = C_{ö} + C_{é} \in n + \theta(\sqrt{n})$$

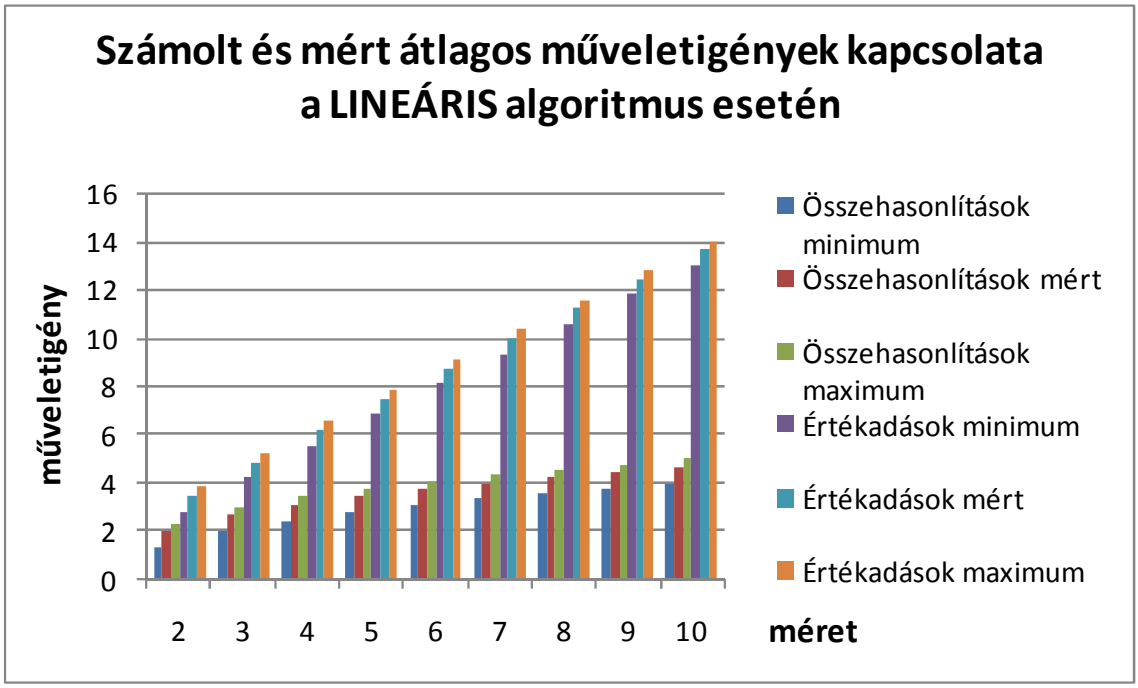
2. tétel bebizonyítva

Az alábbi táblázatban látható a levezetett alsó és felső határ valamint az összehasonlítások és értékadások átlagos számának programmal kiszámolt értéke.

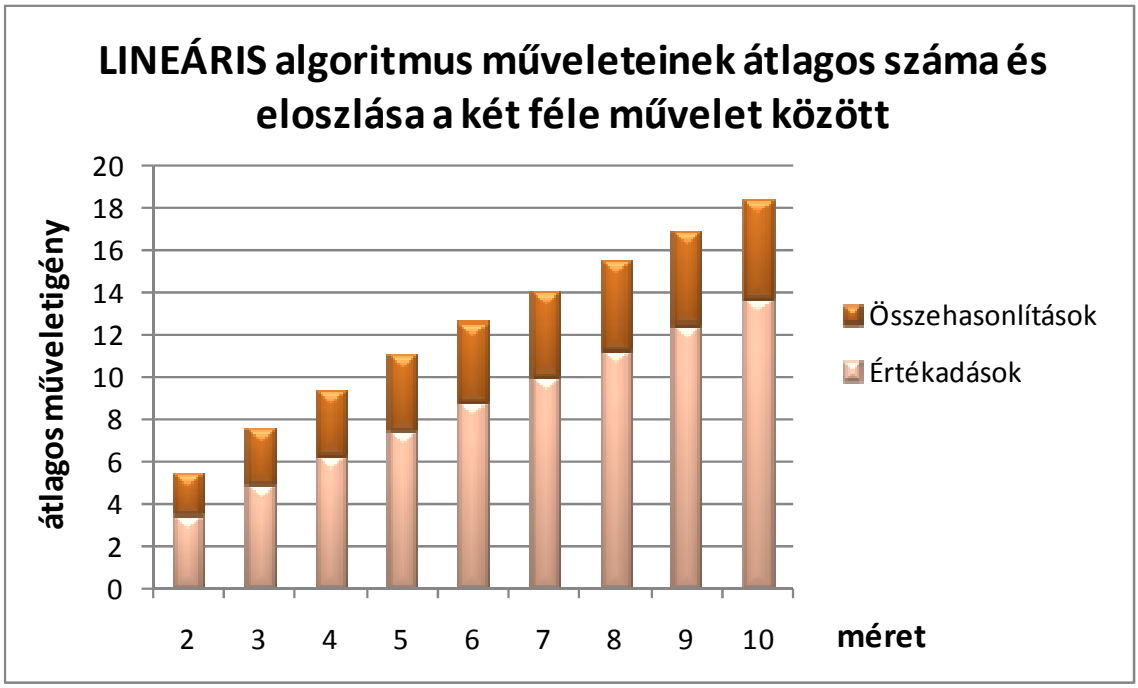
	Összehasonlítások			Értékadások		
n	Minimum	Mért	Maximum	Minimum	Mért	Maximum
2	1,3446261	2	2,3485362	2,8447891	3,5	3,8478663
3	2,0080298	2,6666667	3,0098716	4,2302742	4,8888889	5,2319492
4	2,4645555	3,125	3,4656833	5,5583095	6,21875	6,5593975
5	2,810416	3,472	3,8112049	6,8488168	7,5104	7,8495953
6	3,0968917	3,7592593	4,0974869	8,112324	8,7746914	9,1129162
7	3,3490792	4,0120188	4,34955	9,3551992	10,018139	10,355669
8	3,5792389	4,2426147	4,5796238	10,581642	11,245018	11,582027
9	3,7936594	4,4573791	4,7939818	11,794596	12,458316	12,794918
10	3,9958549	4,6598528	4,9961302	12,996218	13,660216	13,996493

A 12. ábra és 13. ábra ábrán e táblázat adatait láthatjuk, míg a 14. ábra és 15. ábra ábra egy véletlenített szimuláció eredményeit mutatja nagyobb méretekre.

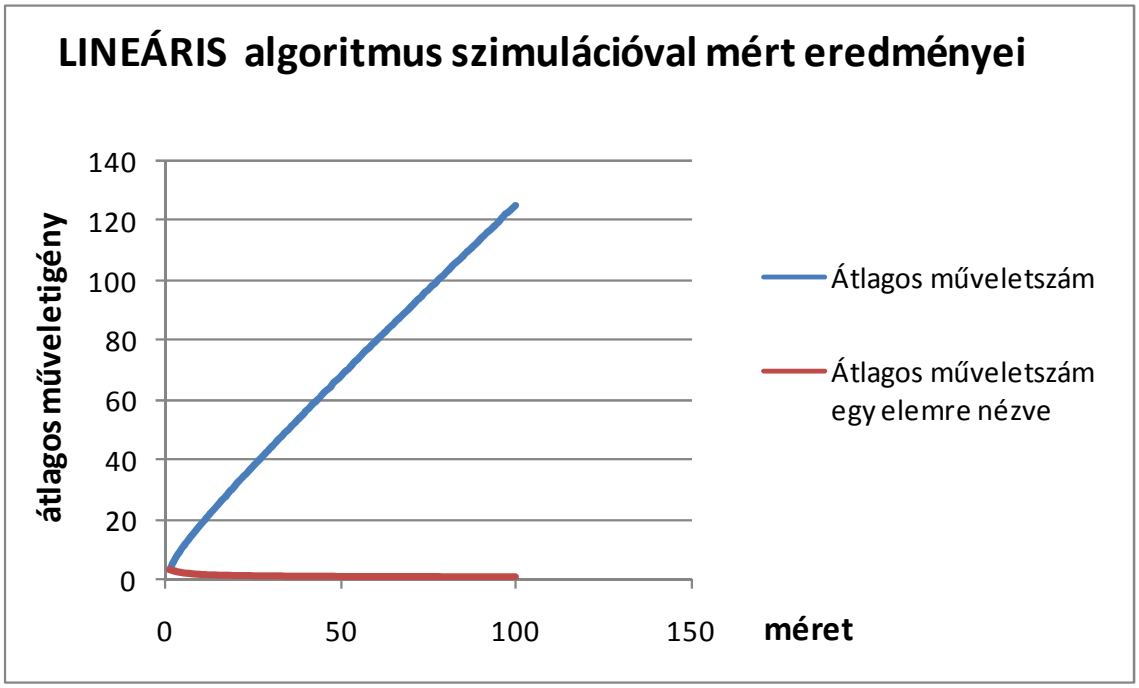
12. ábra: LINEÁRIS műveletigénye és alsó felső becslések



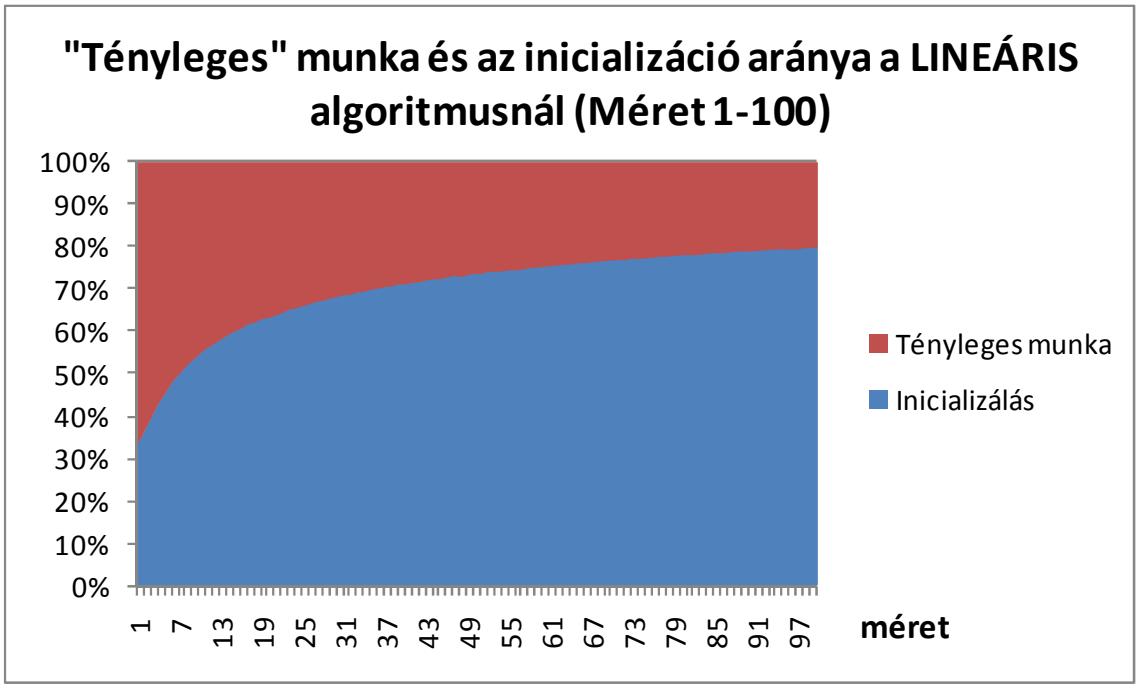
13. ábra: LINEÁRIS műveletigényének eloszlása



14. ábra: LINEÁRIS átlagos műveletigényének linearitása

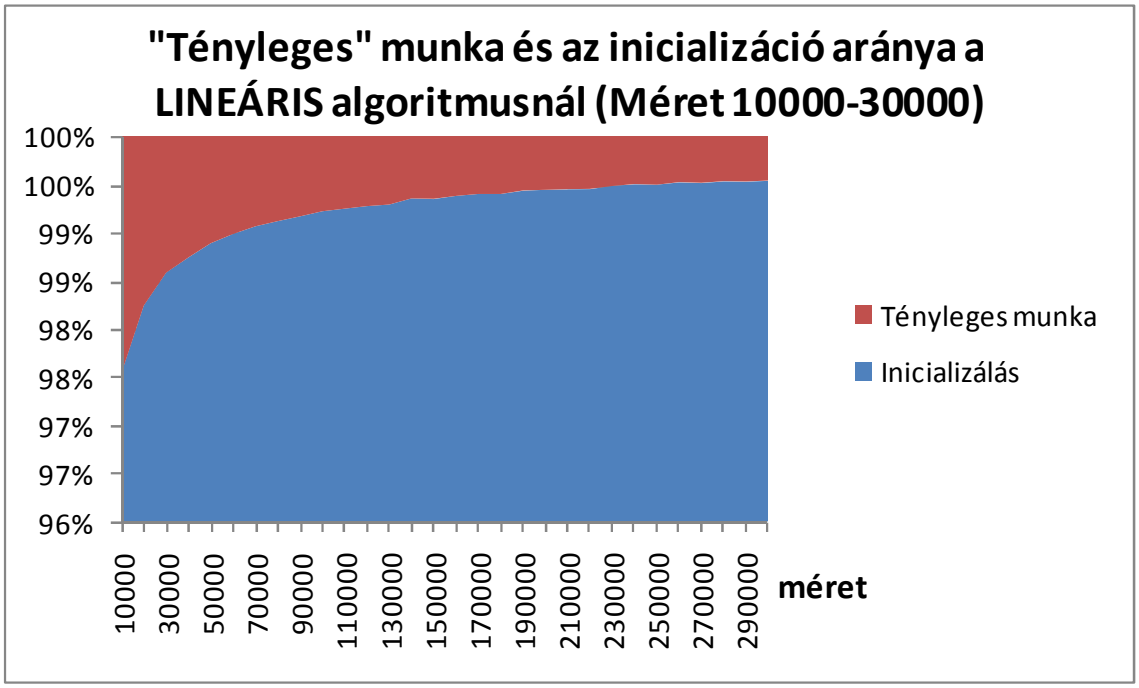


15. ábra: A LINEÁRIS algoritmus munkája és az inicializáció (1)



A 15. ábra ábrán is jól látható, hogy átlagos esetben a műveletigény nagy része az inicializációra kell. Hogy látsszon, hogy ez tényleg 0-hoz tart, készítettem egy hasonló diagramot (16. ábra) nagy méretekre is.

16. ábra: A LINEÁRIS algoritmus munkája és az inicializáció (2)



Vegyük észre, hogy ezen a diagramon 96% és 100% közötti értékek szerepelnek. 300000-es méretnél a teljes munka 99,53%-át az inicializáció teszi ki.

KERESŐFÁS

Egy elem fába való beszúrásának műveletigénye attól függ, hogy a beszúrandó elem helye a fának hányadik szintjén van. Szintenként egy darab összehasonlítással megállapítható, merre kell tovább menni. Ugyan az elágazásnak 3 ága van, de ha olyan összehasonlítást használunk, ami 0-val tér vissza egyezés esetén és pozitív vagy negatív számmal különbözés esetén, akkor az adattal elég egy összehasonlítást végezni. A beszúrásnál végül egy értékadásra van szükségünk. A mutatók kezelése szintenként konstans értékadást jelent, így azokkal nem kell számolni.

Legjobb esetben az első két elem egyezik meg. Ekkor az elsőt be kell tenni a fába. Ez egy értékadás. Majd a másodikat ezzel össze kell hasonlítani.

$$\mathrm{Min}_{\text{KERESŐFÁS}}(n) = 2 \in \Theta(1)$$

Legrosszabb esettel akkor van dolgunk, ha a sorozat helyes és ráadásul rendezett. Ekkor ugyanis a fában mindig egy irányba szúródnak be az új elemek és egy n elemű láncot kapunk. Ekkor minden elem beszúrása előtt meg kell vizsgálni az összes korábbit és még mindet be is kell szúrni. Az összehasonlítások száma így

megegyezik az ELŐRE algoritmus legrosszabb esetével, míg az értékadások száma n.

$$\mathrm{Max}_{\mathrm{KERESŐFÁS}}(n) = \binom{n}{2} + n \in \frac{n^2}{2} + \theta(n) \subset \Theta(n^2)$$

3. tétel:

$$\mathrm{Átl}_{\mathrm{KERESŐFÁS}}(n) \in O(n * \log n)$$

3. tétel bizonyítása

Ez az algoritmus is a leghosszabb helyes prefix megtalálásáig dolgozik. A korábbiak szerint ennek a prefixnek az átlagos hossza gyökös nagyságrendű.

Egyenletes eloszlás mellett a keresőfa átlagos magassága $\theta(\log n)$-es. Ez tananyag volt az egyetemi tanulmányaim során. Megtalálható Fekete István jegyzetében.[5] Ugyanitt az is megtalálható, hogy átlagos esetben egy n elemű keresőfa felépítése kevesebb, mint $2n * \ln n$ összehasonlításba kerül, valamint az, hogy az elemek átlagos magassága a fában kevesebb, mint $2 * \ln n$.

Ezeket a felső becsléseket felhasználva az algoritmus átlagos összehasonlításszámát is becsülhetjük felülről.

$$C_{ö} < \sum_{k=1}^{n} \left[\frac{n! * k}{(n-k)! * n^{k+1}} * (2k * \ln k + 2 * \ln k) \right] - \frac{n!}{n^n} * (2 * \ln n)$$

Az egészet felülről becsülöm, ha a szummában $(\ln k)$-k helyére $(\ln n)$-t írok.

$$C_{ö} < \sum_{k=1}^{n} \left[\frac{n! * k}{(n-k)! * n^{k+1}} * (2k * \ln n + 2 * \ln n) \right] - \frac{n!}{n^n} * (2 * \ln n)$$

$$C_{ö} < \frac{2 * n! * \ln n}{n^{n+1}} \sum_{k=1}^{n} \left[\frac{n^{n-k}}{(n-k)!} * k * (k+1) \right] - \frac{n!}{n^n} * (2 * \ln n)$$

$$C_{ö} < \frac{2 * n! * \ln n}{n^{n+1}} \sum_{j=0}^{n-1} \left[\frac{n^j}{j!} * (n-j) * (n+1-j) \right] - \frac{n!}{n^n} * (2 * \ln n)$$

$$C_{ö} < \frac{2 * n! * \ln n}{n^{n+1}} \sum_{j=0}^{n-1} \left[\frac{n^j}{j!} * \left(j^2 - (2n+1)j + (n^2+n) \right) \right] - \frac{n!}{n^n} * (2 * \ln n)$$

Felhasználgatjuk a korábbi jelöléseket és eredményeket.

$$C_{ö} < \frac{2 * \ln n}{n * M} [S_2 - (2n+1)S_1 + (n^2+n)S_0] - \frac{n!}{n^n} * (2 * \ln n)$$

$$C_{ö} < \frac{2 * \ln n}{n * M} \left[\left((n^2+n)S_0 - 2n^2 * M \right) - (2n+1)(nS_0 - nM) + (n^2+n)S_0 \right] - \frac{n!}{n^n} (2 * \ln n)$$

$$C_{ö} < \frac{2 * \ln n}{n * M} [nS_0 + nM] - \frac{n!}{n^n} * (2 * \ln n) = 2 * \ln n * \frac{S_0}{M} + 2 * \ln n - \frac{n!}{n^n} * (2 * \ln n)$$

Helyettesítsük be a Stirling formula alsó és a hányados felső becslését!

$$C_{ö} < 2 * \ln n * \left(\frac{\sqrt{2\pi}}{2} * \sqrt{n} * \sqrt[12n]{e} \right) + 2 * \ln n - \frac{\sqrt{2\pi n} * \left(\frac{n}{e} \right)^n * \sqrt[12n+1]{e}}{n^n} * (2 * \ln n)$$

$$C_{ö} < \sqrt{2\pi} * \sqrt{n} * \ln n * \sqrt[12n]{e} + 2 * \ln n - 2\sqrt{2\pi} * \frac{\sqrt{n} * \ln n * \sqrt[12n+1]{e}}{e^n}$$

Felső becslésnek ezt a képletet használom a diagram előállításánál, de a kívánt nagyságrend eléréséhez még egy felső becslésre szükség van, aminek köszönhetően kiesik az $\sqrt[12n]{e}$-s tag.

$$C_{ö} < \sqrt{2\pi} * \sqrt[12]{e} * \sqrt{n} * \ln n + 2 * \ln n - 2\sqrt{2\pi} * \frac{\sqrt{n} * \ln n * \sqrt[12n+1]{e}}{e^n} \in O(\sqrt{n} * \log n)$$

Az értékadások száma, viszont megegyezik a LINEÁRIS algoritmus értékadásainak számával, ha kivonjuk belőle az inicializálást. Ezt a szimuláció is alátámasztja. Ugyanazokra a bemenő mintákra a két érték különbsége pontosan n. Az ottani levezetés alapján ez könnyedén levezethető lenne.

$$\frac{\sqrt{2\pi}}{2} * \sqrt{n} * \sqrt[12n+1]{e} - 1 < C_{é} < \frac{\sqrt{2\pi}}{2} * \sqrt{n} * \sqrt[12n]{e}$$

Hogy azt mondhassuk, ez gyökös nagyságrendű, szükséges még egy alsó és felső becslés.

$$\frac{\sqrt{2\pi}}{2} * \sqrt{n} - 1 < C_{é} < \frac{\sqrt{2\pi} * \sqrt[12]{e}}{2} * \sqrt{n}$$

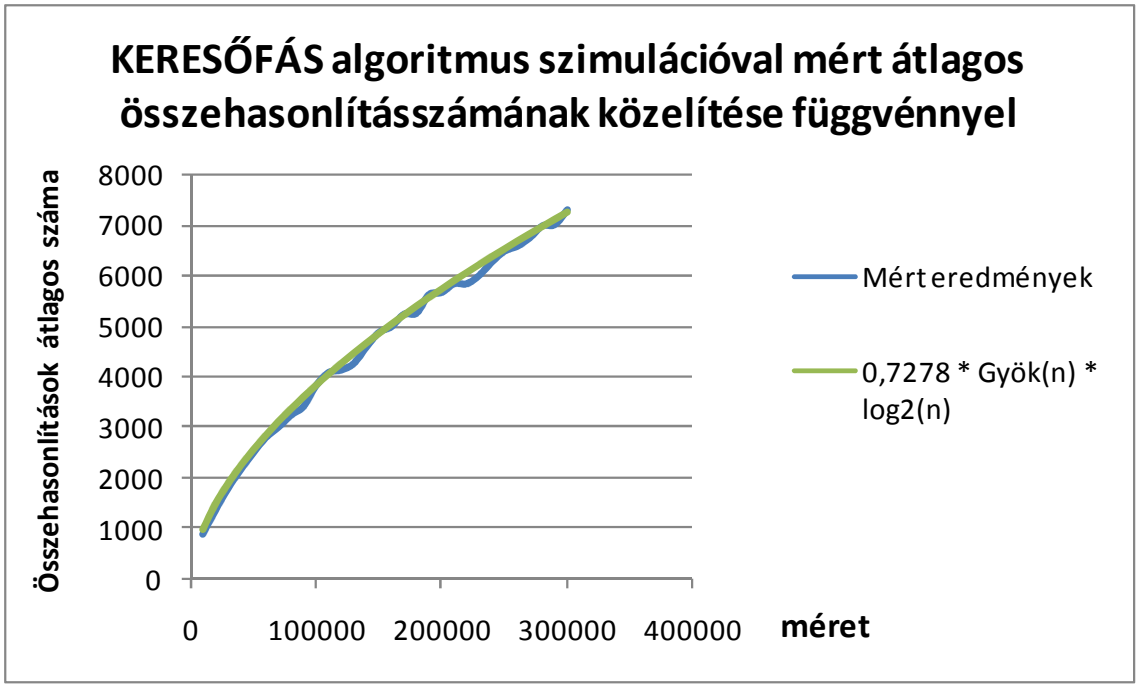
Tehát az értékadások száma gyökös nagyságrendű. A két féle művelet várható darabszámának összege adja az átlagos műveletigényt.

$$Átl_{KERESŐFÁS}(n) = C_{é} + C_{ő} \in O(n * \log n)$$

3. tétel bebizonyítva

A 17. ábra ábrán a véletlenített szimulációval nagy n-re mért összehasonlítások számát közelítem egy $\sqrt{n} * \log n$-es függvénnyel.

17. ábra: KERESŐFÁS átlagos műveletigényének közelítése



$$Átl_{Keresőfás}(n) \in \theta(n * lg\, n)$$

EDÉNYES

Ez az algoritmus is a leghosszabb prefix megtalálásáig dolgozik. Itt azonban a $\lceil\sqrt{n}\rceil$ darab edényhez a számlálók nullázása már az elején $\lceil\sqrt{n}\rceil$ darab értékadást jelent.

Legjobb eset az, ha rögtön az első két elem egyforma. Ekkor az első elemet bemásoljuk, majd a másodikat összehasonlítjuk vele. Ez egy értékadás és egy összehasonlítás.

$$\mathrm{Min}_{\text{EDÉNYES}}(n) = 2 \in \Theta(1)$$

Legrosszabb esetben a sorozat helyes és minden edénybe $\sqrt{n}$ darab elem kerül. Most feltételezem, hogy az bemenet négyzetes, így a $\sqrt{n}$ pontos, ellenkező esetben 1 lehet a különbség az edények méretében, ami nem módosít a nagyságrenden. Ekkor minden edényen belül az odakerülő elemek között a HÁTRA algoritmus játszódik le, és edényenként a HÁTRA algoritmus maximális összehasonlításszáma érvényesül $\sqrt{n}$ méretű bemenetre. Az értékadások terén pedig minden elem egy értékadás.

$$\mathrm{Max}_{\text{Edényes}}(n) = \left(n + \lceil\sqrt{n}\rceil\right) + \left(\lceil\sqrt{n}\rceil * \binom{\lceil\sqrt{n}\rceil}{2}\right) \in \Theta\left(n * \sqrt{n}\right)$$

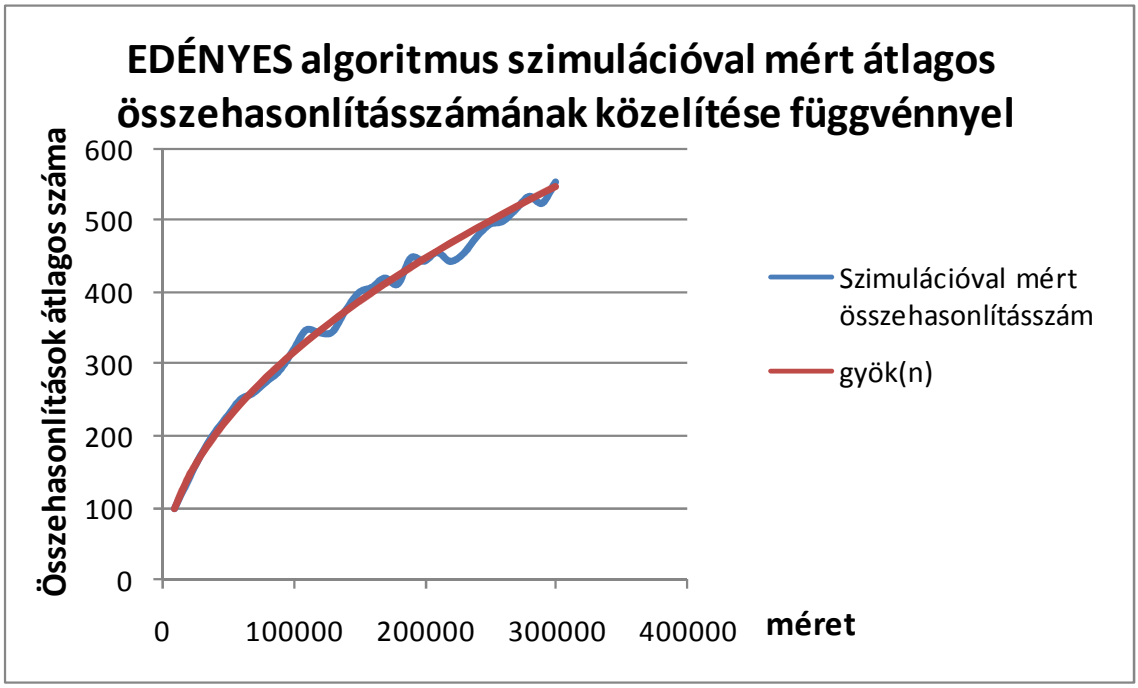
A helyes prefix hosszát most is gyökös nagyságrendűre várjuk. Mivel a sorozat elemei egyenletes eloszlásból származnak, így azok edényenkénti elemszámának várható értéke egy konstanssal becsülhető. Ez azt jelenti, hogy az edényeken belüli egy elem feldolgozásának műveletigénye konstanssal becsülhető.

Az értékadások esetén ugyanaz a helyzet, mint a KERESŐFÁS algoritmusnál. Elemenként egy értékadással dolgozunk. Ettől csak az inicializáció miatt térünk el, de az is gyökös nagyságrendű.

Az összehasonlítások terén szigorúan a korábbi tapasztalatokra építő feltételezésekre és a szimuláció eredményeire támaszkodom.

A 18. ábra ábrán a szimulációval mért átlagos összehasonlításszámot közelítem a $\sqrt{n}$ függvénnyel, ráadásul úgy, hogy nem szorzom meg semmiféle konstanssal. Az inputsorozat megegyezik a KERESŐFÁS algoritmusnál látható 17. ábra inputsorozatával.

18. ábra: EDÉNYES átlagos műveletigényének közelítése



$$\text{Átl}_{\text{EDÉNYES}}(n) \in \Theta(\sqrt{n})$$

MEMÓRIASZEMETES

Ez az algoritmus nem végez inicializációt. Minden elem feldolgozása azzal kezdődik, hogy az algoritmus megvizsgálja, hogy a vektorban a feldolgozandó elemmel indexelt helyen épp a feldolgozandó elem indexe van-e. Ha igen, akkor az elem fel van dolgozva, hiszen önmagával természetesen megegyezik. Ellenkező esetben viszont megvizsgálja, hogy a sorozat vektorból kiolvasott indexű eleme megegyezik-e a most feldolgozandó elemmel. Egyezés esetén leáll az algoritmus, különbözés esetén rájön, hogy az csak memóriaszemét volt, és felülírja a vektorban a megfelelő elemet a most feldolgozott elem indexével. A dolog szépsége, hogy ha szerencsénk van, a memóriában az előző futásokból ránk hagyott index mentén pont megtaláljuk az ismétlődést.

A legjobb eset akkor fordul elő, ha az első elem megegyezik valamely későbbivel, és a memóriában az előző futásból ránk maradt index ehhez az elemhez pont arra a későbbi előfordulására mutat. Ekkor 2 összehasonlítás elegendő.

$$\text{Min}_{\text{MEMÓRIASZEMETES}}(n) = 2 \in \Theta(1)$$

Legrosszabb esetben a sorozat helyes, és a memóriában olyan értékek vannak, hogy minden elemnél szükséges a 2. összehasonlítás is.

$$\mathrm{Max}_{\mathrm{MEMÓRIASZEMETES}}(n) = 3n \in \Theta(n)$$

4. tétel:

$$\mathrm{Átl}_{\mathrm{MEMÓRIASZEMETES}}(n) \in O(\sqrt{n})$$

4. tétel bizonyítása

Ez az algoritmus lényegében a LINEÁRIS algoritmus inicializáció nélküli változata. Itt viszont a szerencsés helyzetek miatt már nem igaz, hogy a leghosszabb helyes prefix megtalálásáig dolgozik, az viszont biztos, hogy legfeljebb addig.

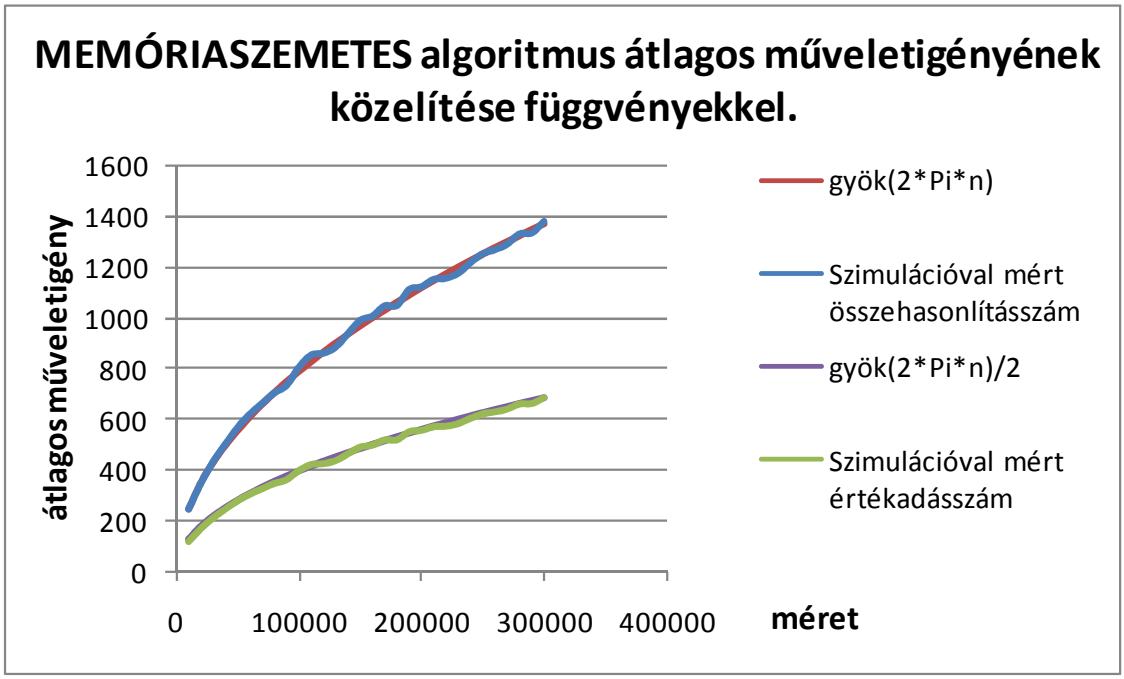
Minden elem feldolgozása egy vagy két összehasonlítást igényel és legfeljebb egy értékadást. Ebből az következik, hogy az algoritmus átlagos műveletigényének nagyságrendje legfeljebb $\sqrt{n}$-es lehet. A kérdés már csak az, hogy a szerencsének elég nagy-e a szerepe ahhoz, hogy ezen javítson.

Ez könnyen bizonyítható. Elég, ha úgy becsüljük felül a műveletigényt, hogy elhagyjuk a szerencsét. Ekkor az inicializációt leszámítva a lineáris algoritmus műveletigényét kapjuk. Igaz itt az összehasonlítások száma kétszer annyi, de ez nem módosít a nagyságrenden.

4. tétel bebizonyítva

A szerencse valószínűsége függ az előző minták által a vektorban hagyott értékektől. Minden futás során az algoritmus 1-től, a legnagyobb feldolgozott indexig ír be számokat a vektorba. Ez azt jelenti, hogy ha a most vizsgált elem az utolsó elemmel egyezik meg, akkor a szerencséhez az kell, hogy a vektorban legyen legalább 1 elem, ami az utolsó elemre indexel és ráadásból ez pont az legyen, ami a most feldolgozott elemmel van indexelve. Mivel az algoritmusunk átlagosan legfeljebb $\sqrt{2\pi}/2 * \sqrt{n}$-ig fut, a vektor tele lesz kis indexű elemekkel. Vagyis a szerencse kiszámításához ismernünk kell az összes olyan korábbi lefutás eredményét, aminek eredménye a vektorban még megtalálható. Ez a feladat meghaladja egy diplomamunka színvonalát így az átlagos műveletigényt a szimulációk alapján adom meg.

19. ábra: MEMÓRIASZEMETES átlagos műveletigényének közelítése



A diagramról könnyedén leolvasható, hogy a szerencse nem avatkozik be a nagyságrendekbe.

$$\text{Átl}_{\text{MEMÓRIASZEMETES}}(n) \in \Theta(\sqrt{n})$$

A következő táblázat a szerencse szerepét mutatja. Mindkét algoritmus ugyanazt az inputot dolgozta fel. Ha nem volna szerencse, akkor a két értékadásszám megegyezne. A kettő közötti különbség tehát a szerencsének tudható be.

Darab	Méret	Algoritmus	Összehasonlítás	Értékadás
1000	300000	Keresőfa	7283,692	689,906
1000	300000	Memóriaszemetes	1378,014	688,201

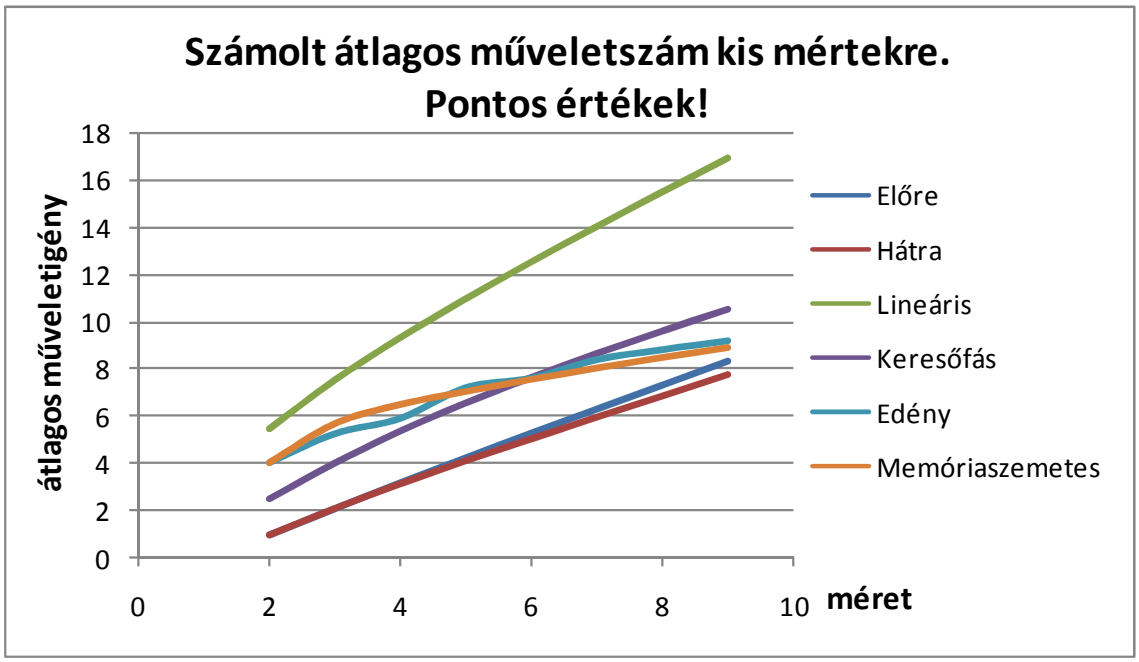
Algoritmusok összehasonlítása

A sok elemzés és vizsgálat után ideje az eredményeket egymással szembe állítani. Kezdetnek egy összesítő táblázat, amiben az elméleti és szimulációs módszerekkel kihozott nagyságrendek láthatóak. A táblázatban vastag betűvel vannak feltűntetve az elméleti levezetések eredményei és vékonnyal azok, amiket csak szimulációval tudok alátámasztani.

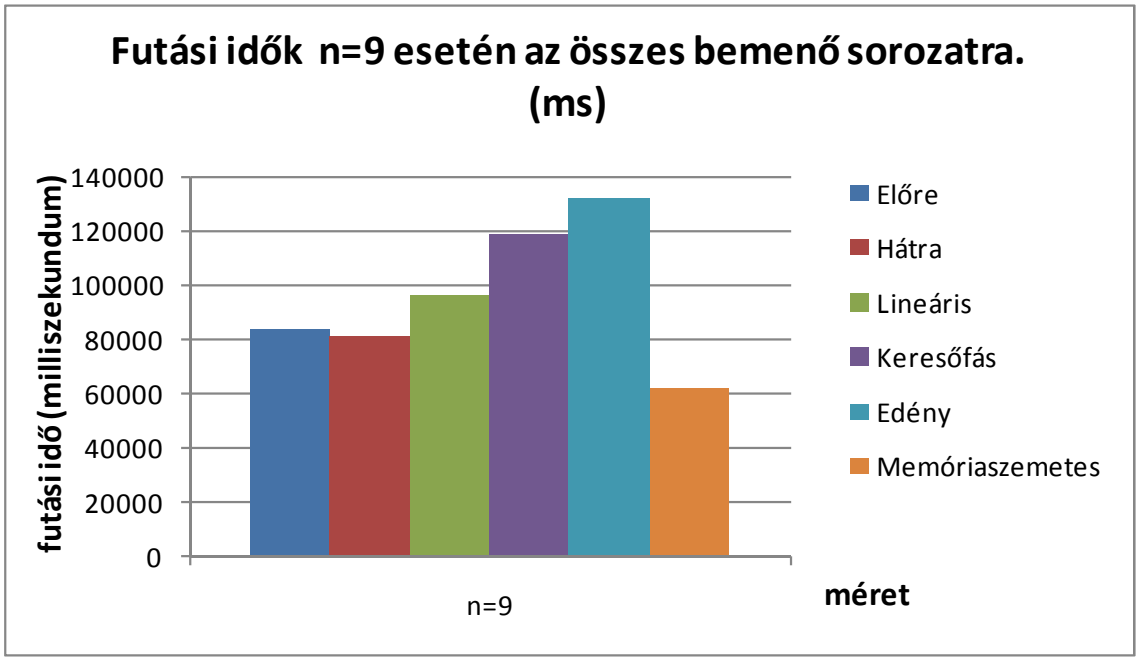
	Legjobb eset	Átlagos eset	Legrosszabb eset
HÁTRA	$\boldsymbol{\theta(1)}$	$\boldsymbol{\theta(n)}$	$\boldsymbol{\theta(n^2)}$
ELŐRE	$\boldsymbol{\theta(1)}$	$\theta(n)$	$\boldsymbol{\theta(n^2)}$
LINEÁRIS	$\boldsymbol{\theta(n)}$	$\boldsymbol{\theta(n)}$	$\boldsymbol{\theta(n)}$
KERESŐFÁS	$\boldsymbol{\theta(1)}$	$\mathbf{O(\sqrt{n} * \log n)}$ $\theta(\sqrt{n} * \log n)$	$\boldsymbol{\theta(n^2)}$
EDÉNYES	$\boldsymbol{\theta(\sqrt{n})}$	$\theta(\sqrt{n})$	$\boldsymbol{\theta(n * \sqrt{n})}$
MEMÓRIASZEMETES	$\boldsymbol{\theta(1)}$	$\mathbf{O(\sqrt{n})}$ $\theta(\sqrt{n})$	$\boldsymbol{\theta(n)}$

Nézzünk meg egy pár diagramot! A 20. ábra és 21. ábra ábrán az adott méretekre az összes sorozatra lefuttatott tesztek eredménye látható.

20. ábra: Átlagos műveletigények kis méretekre az összes bemenő adatra



21. ábra: Algoritmusok futási ideje $n = 9$-re.



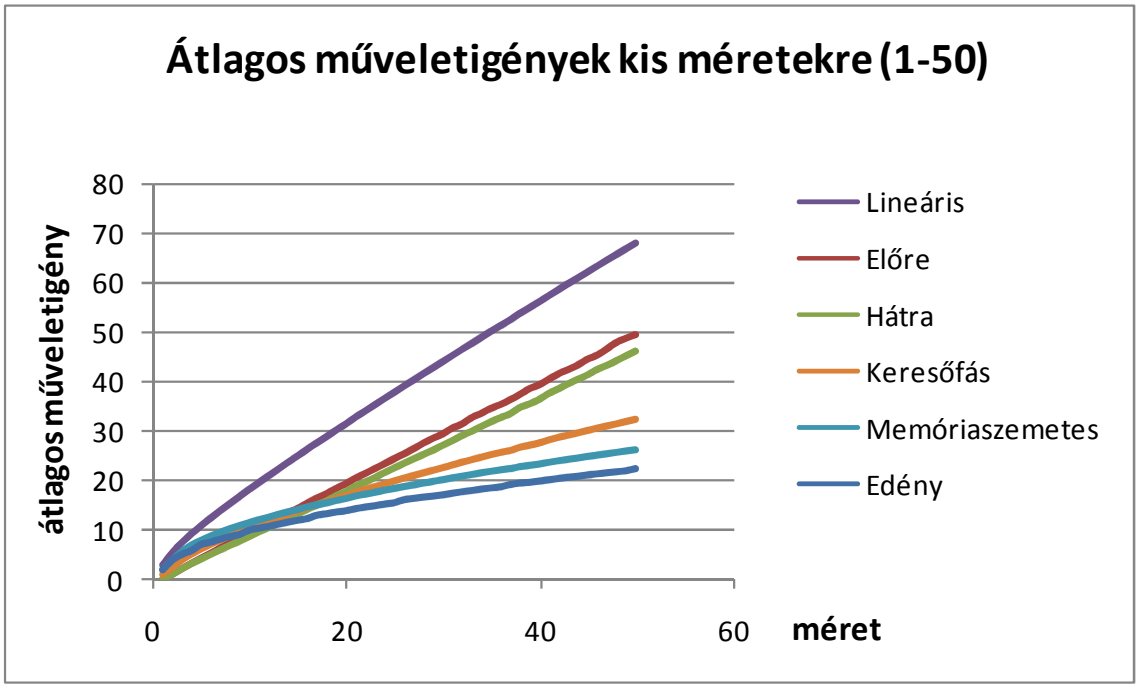
Érdemes megfigyelni a 20. ábra ábrán az Edényes algoritmus ingadozását. Ez a $\sqrt{n}$ és a $\lceil\sqrt{n}\rceil$ különbségéből ered. Legkedvezőbb akkor a helyzet, ha n négyzetszám, mert ekkor a két érték megegyezik.

A műveletigényes diagramból az következik, hogy $n = 9$-ig a HÁTRA algoritmus a legjobb választás. Ez viszont abban az esetben igaz, ha az algoritmusok által kezelt adatszerkezetek, ciklusváltozók kezelésének futási ideje eltörpül a sorozat elemeivel kapcsolatos műveletek futási idejéhez képest. Mivel itt a célunk a Sudoku tábla gyors leellenőrzése, ami pont olyan számokból áll, mint amit a segédváltozók felvehetnek, érdemes foglalkoznunk a valós futási idővel is.

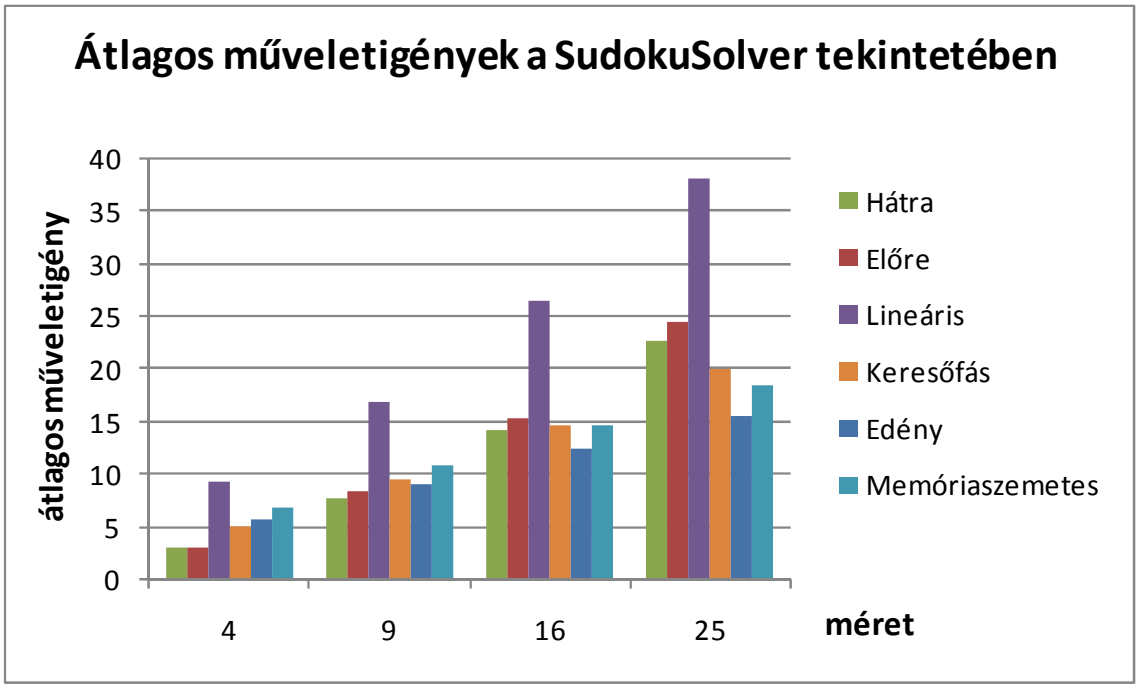
A 21. ábra a futási idők alapján a MEMÓRIASZEMETES algoritmust hozza ki a legjobbnak. Ez nem meglepő, hiszen az egész csak egy ciklusból és azon belül két feltétel kiértékeléséből áll. Ugyanakkor ez az eredmény ki van téve a konkrét számítógép pillanatnyi leterhelésének is, ami akár komoly mértékben is torzíthatja az eredményt.

Folytatásként nézzünk meg 4 diagramot a véletlenített tesztelővel nyert adatokról. A bemenetek hossza 1 és 50 között egyesével nő, darabszámuk pedig konstans 100000.

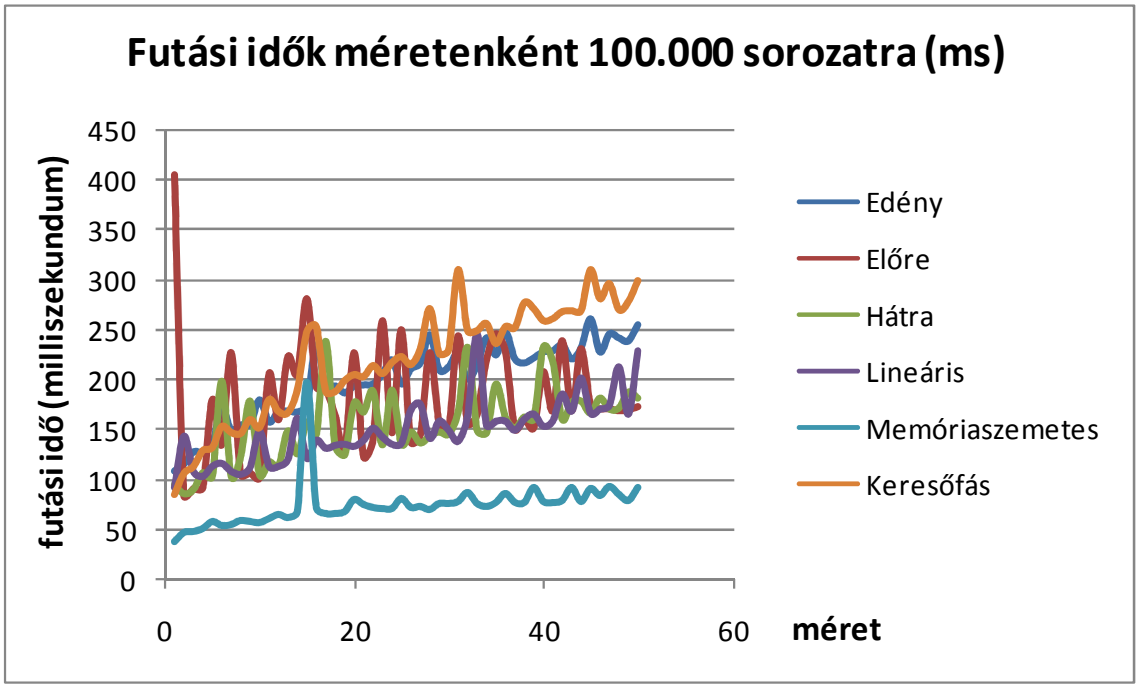
22. ábra: Algoritmusok átlagos műveletigénye kisméretű bemenetekre (1)



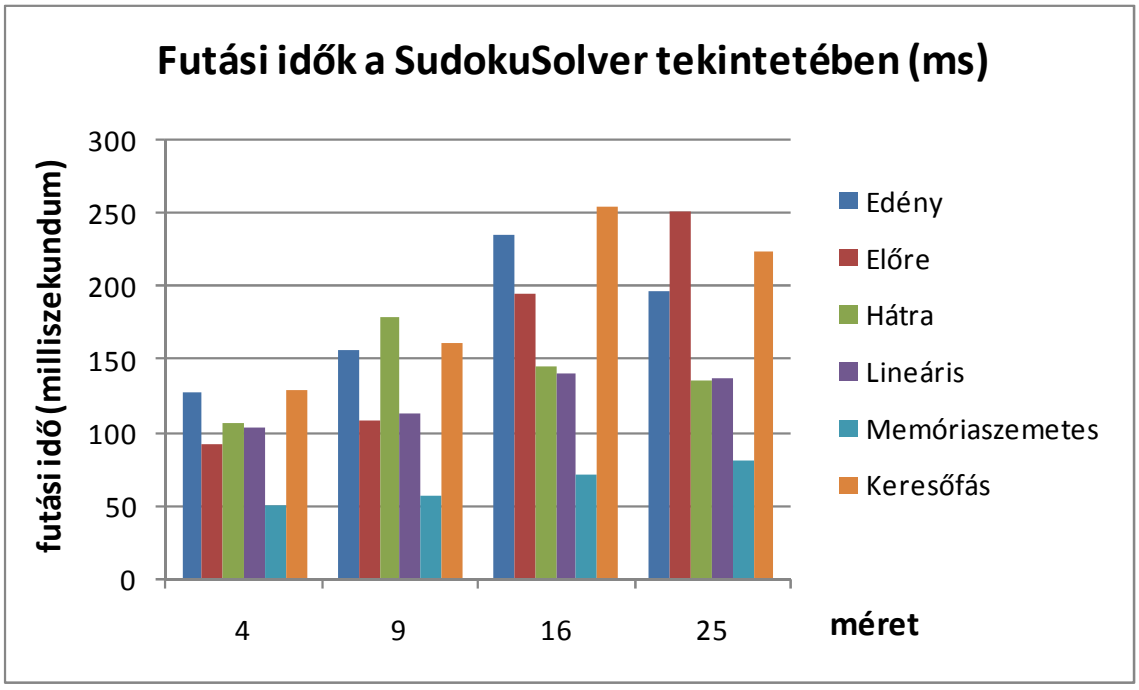
23. ábra: Algoritmusok átlagos műveletigénye kisméretű bemenetekre (2)



24. ábra: Algoritmusok futás ideje kisméretű bemenetekre (1)



25. ábra: Algoritmusok futás ideje kisméretű bemenetekre (2)



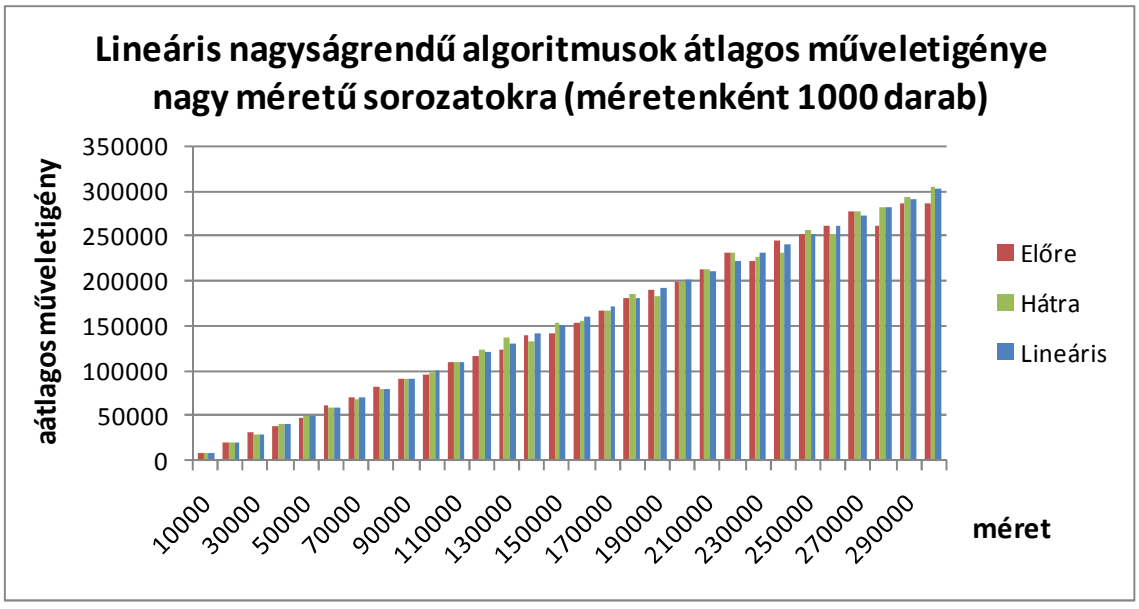
Itt már látható, hogy a HÁTRA algoritmus $n = 13$ közelében elveszíti vezető pozícióját és átadja azt az EDÉNYESNEK. A LINEÁRIS algoritmus tűnik idáig a legrosszabb választásnak.

A futási idők igen kusza eredményt adnak. Ezt a processzorütemezésnek tudom be. Ugyanakkor szignifikánsan elüt a MEMÓRIASZEMETES a többiektől. Az ELŐRE görbéjének az elején a nagy kiugró értéket a háttérben folyó lapcseréléseknek tudom be.

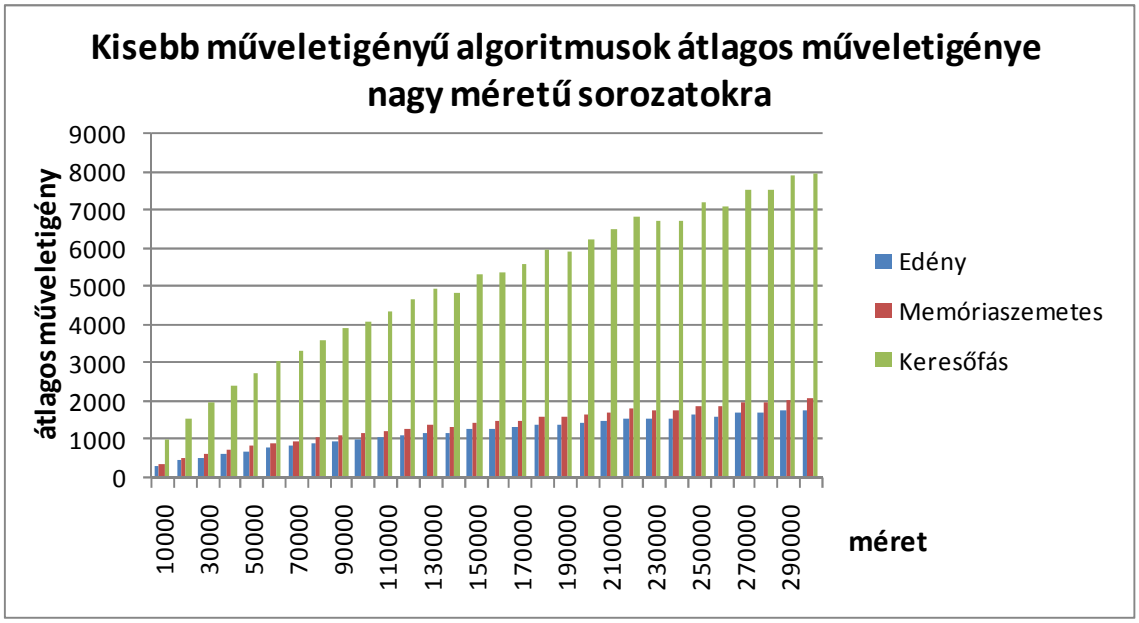
A 23. ábra és 25. ábradiagramom az 22. ábra és 24. ábra diagram kiemelt esetei láthatóak. Ezeken a pozíciókon négyzetes a méret, hiszen a SudokuSolver szempontjából ezek az érdekesek. A 24. ábra ábra miatt hoztam azt a döntést, hogy a véletlen rejtvény generátoromnál a véletlenül kitöltött tábla tesztelésére a MEMÓRIASZEMETES algoritmust használtam.

Következzenek a nagy méretre készített tesztadatok. Itt külön kellett választani a lineáris és nem lineáris nagyságrendű algoritmusokat, mert egy diagramon nem voltak megjeleníthetőek az eredmények.

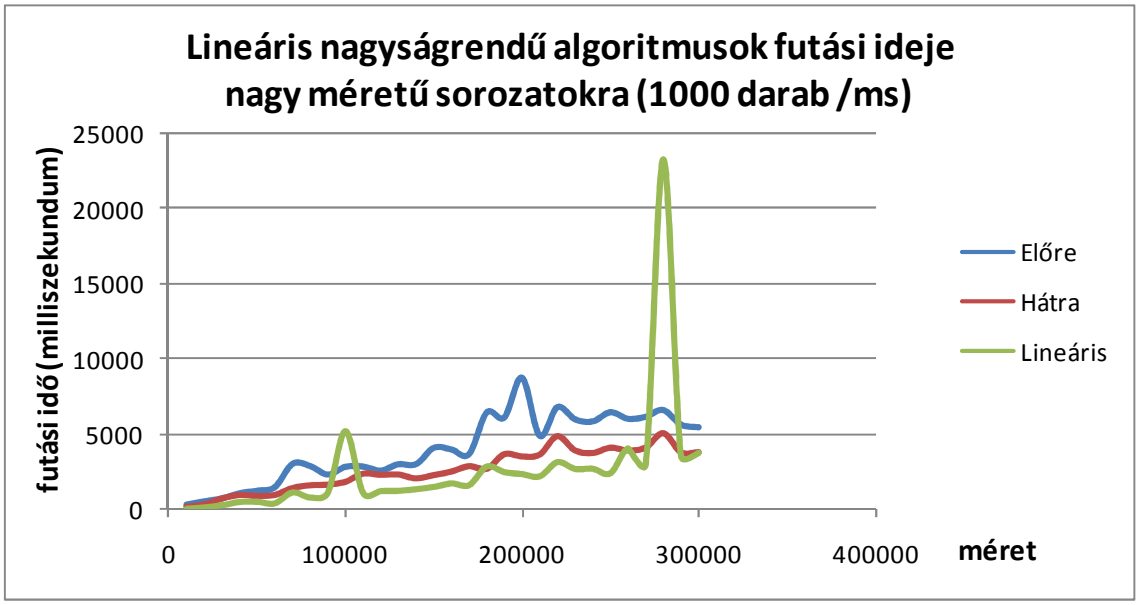
26. ábra: Algoritmusok átlagos műveletigénye nagyméretű bemenetekre (1)



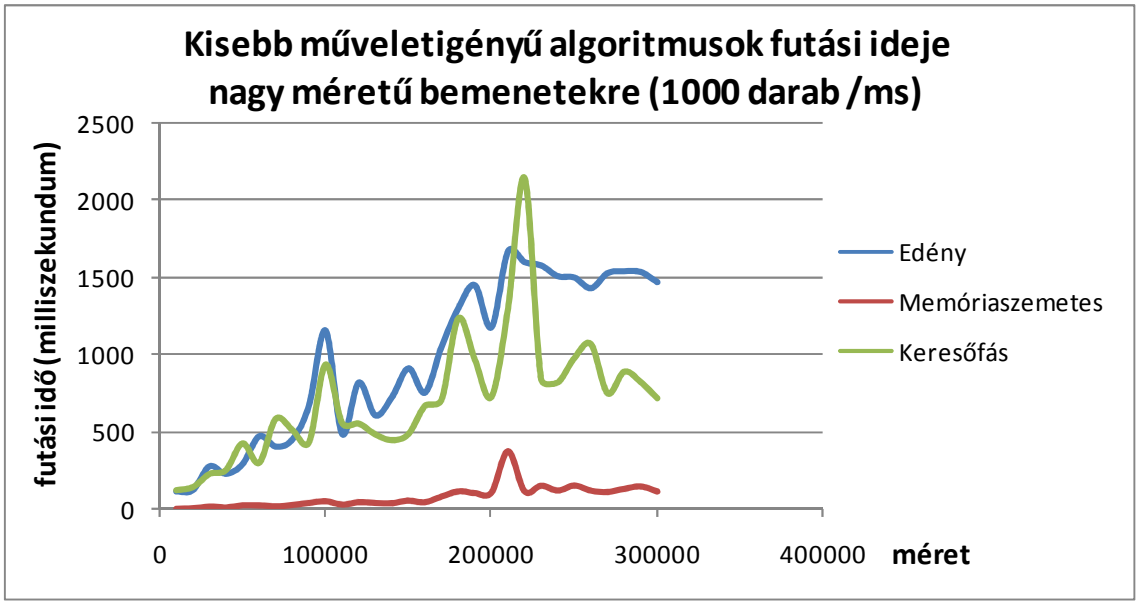
27. ábra: Algoritmusok átlagos műveletigénye nagyméretű bemenetekre (2)



28. ábra: Algoritmusok futási ideje nagyméretű bemenetekre (1)



29. ábra: Algoritmusok futási ideje nagyméretű bemenetekre (2)



A 26. ábra ábrából az szűrődik le, hogy a LINEÁRIS algoritmus a kezdeti gyengeségét leküzdi az ELŐRE és HÁTRA algoritmusokkal szemben. Ezen kívül a 3 algoritmus közti különbség a konkrét mintákon múlik.

A 27. ábra ábrán kirajzolódnak a táblázatban foglalt nagyságrendek, és az is bebizonyosodni látszik, hogy műveletszámban $n \geq 13$ esetben az EDÉNYES algoritmus a legjobb.

A 28. ábra kitűnő példa a processzorütemezésből eredő torzióra.

A 29. ábra szolgált a legtöbb meglepetéssel. Hiába más a műveletvégzésük nagyságrendje, a valós futási időben az EDÉNYES algoritmus mégis a KERESŐFÁS algoritmussal vetekszik. Figyelembe véve, hogy e grafikon adatainak előállítása legalább fél órát vett igénybe, ezt az eredményt már nem a processzorütemezésnek tudom be. Az viszont biztos, hogy a futási időkben most is a MEMÓRIASZEMETES a legjobb, ráadásul szignifikáns különbséggel.

Érdemes megnézni, hogy mit mondhatunk a nagyságrendek mellett a főtaghoz tartozó konstans értékéről. Az alábbi táblázatban a konstans értékét és a főpolinom szorzatát tüntetem fel minden algoritmus esetén. Az előző diagramhoz hasonlóan, ahol ezt bizonyítás is alátámasztja, azt vastag szöveggel jelölöm.

	Legjobb eset	Átlagos eset	Legrosszabb eset
HÁTRA	$\mathbf{1}$	$\mathbf{1 * n}$	$\mathbf{{}^1/_2 * n^2}$
ELŐRE	$\mathbf{1}$	$1 * n$	$\mathbf{{}^1/_2 * n^2}$
LINEÁRIS	$\mathbf{1 * n}$	$\mathbf{1 * n}$	$\mathbf{2 * n}$
KERESŐFÁS	$\mathbf{2}$	$0{,}7278 * \sqrt{n} * \log_2 n$	$\mathbf{1 * n^2}$
EDÉNYES	$\mathbf{1 * \sqrt{n}}$	$1 * \sqrt{n}$	$\mathbf{1 * n * \sqrt{n}}$
MEMÓRIASZEMETES	$\mathbf{2}$	$\frac{3\sqrt{2\pi}}{2} * \sqrt{n}$	$\mathbf{3 * n}$

Összegzés

Hogy melyik algoritmust érdemes választani, az nem egyértelmű. Ezek a levezetések és szimulációk arra az esetre vonatkoznak, ha véletlenszerűen vannak az elemek kitöltve. Ha azonban a rejtvényeket egy valódi ember tölti ki, sokkal nagyobb a valószínűsége a helyes kitöltésnek. Ez azt jelenti, hogy ilyenkor az átlagos műveletszám a mostani átlagos és a legrosszabb eset közé tehető.

A választásba az is beleszólhat, hogy mekkora a táblánk, arról nem is beszélve, hogy az elemekkel kapcsolatos műveletek végrehajtási ideje nagyságrendekkel tovább tart-e, mint a háttérben történő műveletek.

A Memóriaszemetes algoritmus a legtöbb esetben jó választásnak tűnik. Ettől eltérni csak véletlen rejtvények esetén lehet indokolt és ott is csak akkor, ha nagyon kicsi méretű a táblánk, vagy ha futásidőigényes az elemek feldolgozása. Ekkor az első esetben a HÁTRA algoritmust, míg a második esetben az EDÉNYES algoritmust javaslom.

SudokuSolver

Diplomamunkámban célkitűzés volt a nagyprogramként megírt SudokuSolver továbbfejlesztése. Ez a program az elmúlt szemeszterben sikeresen támogatta az oktatást az ELTE Informatika Karán meghirdetett Sudoku témájú kurzuson. Ezen a gyakorlaton megoldó algoritmusokat tanulhatnak azok, akiket ez komolyabban érdekel. Az algoritmusok egymásra épülnek. Bevezetésükkor példákat is kapunk arra, hogy van olyan rejtvény, amit a korábbi algoritmusokkal nem tudunk megoldani, de ha az újat is használjuk, akkor már igen. A megoldásnak determinisztikus sorrendje van, vagyis adott rejtvény esetén mindenkinek, aki ezzel a módszerrel dolgozik, ugyanabban a sorrendben kell a számokat beírni és a jelölteket törölni. A háttérben különféle listák vannak, amelyek alapján dől el, mi legyen a következő lépés. Ezek a tanfolyam vége felé igen bonyolulttá, papíron nehezen követhetővé válnak, és a beadandókként írandó megoldási naplókba könnyen csúszhat hiba. Ez nagyon dühítő tudott lenni, amikor az ember órákig írta a naplót, és már az elején elrontotta. Az elmúlt szemeszterben az órát felvevő diákok szerint, ezt a programom levette a vállukról.

SudokuSolver tudása nagyprogramként

- A program ismerte az összes algoritmust, ami a kurzuson tananyag volt.
- Kompatibilis a többi rejtvénymegoldó program fájltípusával.
- Teljes megoldási napló előállítása, amelynek előállításához használta az összes betáplált algoritmust.
- A rejtvény kitöltését a felhasználó saját maga is végezheti egérrel, nagyon egyszerű módon.
- Tetszőleges kitöltés mellet volt lehetőségünk tippeket kérni. Ekkor lefutott az összes algoritmus és egy könnyen értelmezhető listában megjelenítette a lehetséges lépéseket.
- A tippeket a felhasználó grafikusan, a táblára kirajzolva is láthatta.
- Minden tipphez tartozott rövid szöveges magyarázat is, amiből a laikus számára is érthető volt, hogy miért lehet azt a számot beírni vagy azt a jelöltet törölni.

- A rejtvényről lehetőségünk volt megtudni, hogy a rejtvénynek mi a típusa, anélkül, hogy a megoldást megtudtuk volna. A típusból kiderül, hogy mekkora a rejtvény mérete, hány megoldása van, milyen nehéz a kitöltése és hány elem van kitöltve.
- Támogatta a $4 * 4$-es, $9 * 9$-es, $16 * 16$-os, $25 * 25$-ös méretű táblákat is.

A témabejelentőmben ennél kevesebb eredményt reméltem a nagyprogramomtól. Az alábbi pontokat csak a Diplomamunkámban szándékoztam megvalósítani, de már a nagyprogram is tudta.

- Oktató rendszerré fejlesztés
- A rejtvények nehézség szerinti osztályozása a Rózsa algoritmusok alapján
- Rejtvénymegoldást támogató, felhasználóbarát rendszer

SudokuSolver továbbfejlesztése

A témabejelentőben a programra vonatkozó célkitűzések közül az alábbi pontok teljesítése maradt hátra.

- A keretrendszer feltöltése további algoritmusokkal
- Véletlen új rejtvények előállítása

Erről a két pontról szól két fejezet a diplomamunkámban, de ezen felül más eredményeket is sikerül elérnem. Ez a fejezet ezekről a fejlesztésekről szól.

Átméretezhetőség

Sokan igényelték, hogy a programban legyen lehetőség arra, hogy az ablak nagyobb méretűvé változtatása esetén a rejtvény mérete is megváltozzon. Ez utólag belegondolva tényleg nagy hiányosság volt, de szerencsére ez már a múlté.

Algoritmusok a tábla mérete alapján

A tárgy keretein belül jórészt $9 * 9$-es táblákkal foglalkoztunk, így a paraméterezhető algoritmusokat, csak erre a méretre paramétereztük. Ha a táblánk $16 * 16$-os, akkor a megoldásban hasznos lehet például a TISZTA-8 algoritmus is, ahol 8 cellát keresünk egy házban, amiben összesen 8 féle jelölt van. Ha találunk ilyet, akkor tudjuk, hogy azok a jelöltek biztosan azokba a cellákba fognak kerülni, így a többi cellából

törölhetőek. $9 * 9$-es rejtvény esetén ezt a REJTETT CELLA algoritmus megoldja, de ennél a méretnél ezt csak a REJTETT-9 algoritmus oldaná meg.

A megoldás egyszerű. A már elkészített algoritmusok nagyobb értékkel való paraméterezéssel bekerülnek az algoritmusok sorrendjébe nagyobb sorszámmal. Minden méretű rejtvényhez így külön algoritmuslista áll rendelkezésünkre.

Megoldások mentése, listázása

Sokan zavarónak találták, hogy a megoldási napló elkészítésekor a megjelenített megoldást nincs lehetőség elmenteni. Mivel a program a visszalépéses algoritmus miatt több megoldást is képes előállítani, az azok közötti váltogatást is megvalósítottam.

A visszalépéses algoritmus tesztelése során párszor belefutottam abba, hogy rengeteg megoldást generált a program, ami rengeteg állományt eredményezett. Ezek törlése néha órákba került, bár nem tudom, hogy miért. Ezt a problémát az „ömlesztett” fájltípus használatával oldottam meg. Ebben a rejtvények az SDXX formátumban vannak tárolva, de egy állományba már több rejtvény is kerülhet. Ennek a fájltípusnak a használatát azonban nem engedélyezem a felhasználónak.

Ennek oka a program felépítésében rejlik. A fájlok megnyitása egy külön osztályra van bízva, ami az őt használó osztályok számára elfedi, hogy milyen típusú fájlt dolgoz éppen fel. Az ömlesztett fájltípus esetében a fájlnév mellett egy sorszámot is meg kellene adni a rejtvény kiolvasására, amit a kódban egyszerűen a fájlnév mögé fűzök a karakterláncban. Erre a lehetőségre azonban a Windowsos fájl megnyitás dialógusban nincs lehetőség.

Megoldási menet paraméterezhetősége, párhuzamosítása

Korábban csak teljes napló készítését lehet kérni a programtól. Ez a helyzet több okból sem szerencsés.

A bővített algoritmuslista új elemei a paraméterezésük növekedésével nagyobb műveletigénnyel dolgoznak. $25 * 25$-ös tábla esetén megjelenik például a TISZTA-12 algoritmus, amit még nem is próbáltam kivárni.

A hallgatók számára is könnyebb, ha akkor, amikor egy olyan rejtvény dolgoznak fel, amit az ő álltaluk odáig vett algoritmusok még nem tudnak megoldani, nem azt

látják, hogy a rejtvény megoldható későbbi algoritmusokkal, hanem azt, hogy nem megoldható.

Hasonló módon jó ötletnek tartom a visszalépéses algoritmus használatának kikapcsolhatóságát is.

A megoldási idő megnövekedésével szükségessé vált a folyamat leállításának lehetősége. Ezt már a rejtvény generátornál is meg kellett oldanom, de emellé szükséges az is, hogy az külön szálon fusson. A hosszú várakozási idő esetén hasznosnak tartom a megoldás állapotának megjelenítését is.

Rejtvény megoldhatóság elemzésének paraméterezhetősége, párhuzamosítása

Egy nagyobb méretű tábla esetén komolyan megnőhet ennek a folyamatnak is a futási ideje, főleg ha a visszalépéses algoritmust is használjuk hozzá. Így itt is szükségessé váltak az előbb említett változtatások.

Tippek előállításának párhuzamosítása

Amikor tippeket kérünk az független a rejtvény állapotától abban a tekintetben, hogy az meddig tart. Ilyenkor az összes algoritmussal megvizsgáljuk az egész táblát. A megnövekedett algoritmuslistának köszönhetően már a $16 * 16$-os rejtvény vizsgálata is igen sokáig tart. Ameddig az egész le nem futott, addig a program nem válaszolt. Most a tippek rögtön listára kerülnek, amint előállítottuk. Ez nagyon előnyös, hiszen nagyon ritkán van szükségünk a nagy műveletigényű algoritmusokra. A program jelzi, hogy befejezte-e a vizsgálatot.

Eredeti rejtvény visszaállítása, mentése

A programba került véletlen rejtvény menüpontnak köszönhetően olyan rejtvényeket is megoldhatunk, amiket nem tudunk vagy sok keresgélés árán tudunk csak újra megnyitni. Mire rájövünk, hogy ezt a rejtvényt el szeretnénk menteni, már csak a kitöltött rejtvényt menthetjük el. Ezt a problémát az eredeti rejtvény visszaállításának és mentésének lehetőségével orvosoltam.

Royle 17-es rejtvények feldolgozása

Nagy sejtés a Sudoku világában, hogy $9 * 9$-es rejtvényből nincs az egyértelműen kitölthetőek között olyan, amiben ne lenne legalább 17 elem kitöltve. Gordon Royle 17 elemű egyértelműen megoldható rejtvényeket gyűjt [7]. Ebből jelenleg majdnem 50000 darab van.

Írtam egy programot, ami a rejtvénygenerátor mintájára feldolgozza ezeket a rejtvényeket és besorolja őket nehézség szerint. Az így elkészített rejtvényeket lehetőségünk van kisorsolni a SudokuSolverben.

Ezt a programot egyszeri használatra terveztem, így a kezelését nem a felhasználóra terveztem. Ez abban merül ki, hogy a dizájnt teljesen nélkülözi és nincs benne hibakezelés. Ugyanakkor a később meglelhető rejtvények feldolgozása miatt ezt a programot is csatolom a diplomamunkámhoz.

Felhasználói dokumentáció

Hardver és szoftver követelmények

Szoftver:

Windows operációs rendszer (XP Vagy újabb), amire fel van telepítve a .NET 3.5 keretrendszer.

Hardver:

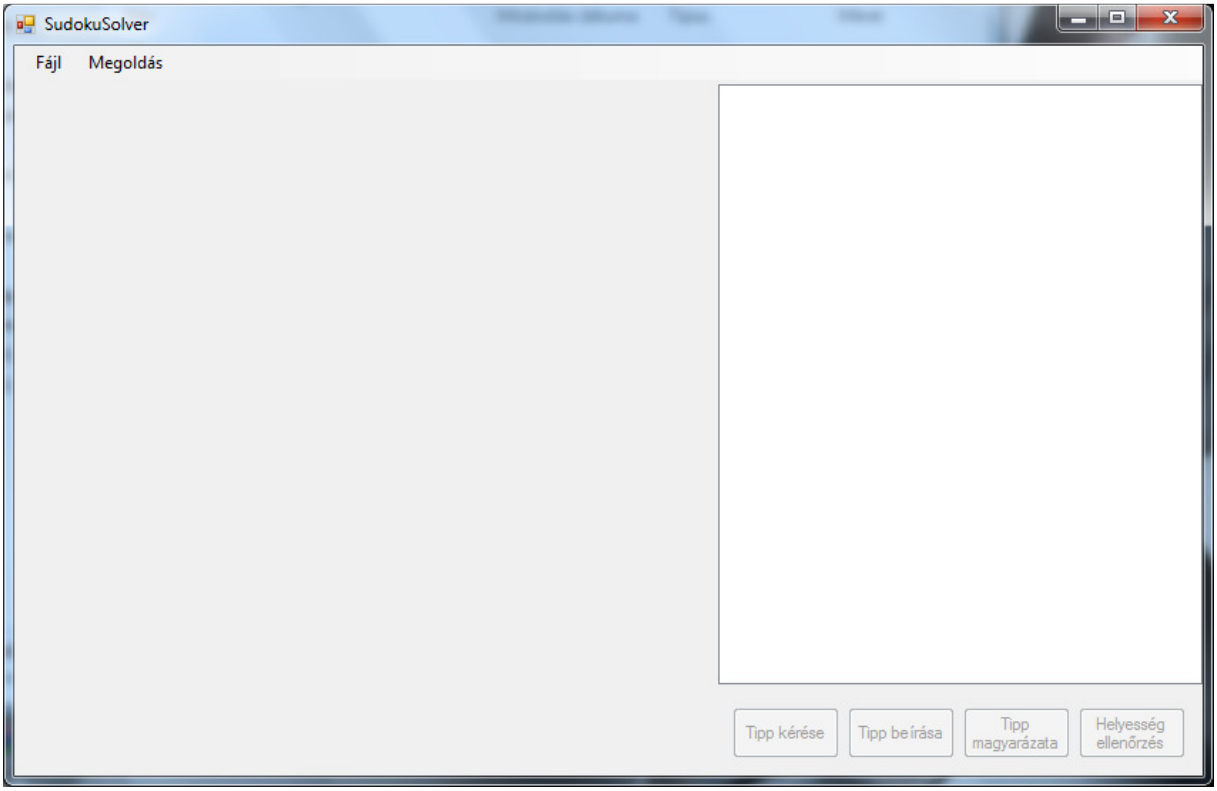
A program maga nem rendelkezik nagy gépigénnyel, így a választott operációs rendszer hardverigényét jelölöm meg. A képernyő felbontása legalább 1024*768 legyen!

Telepítés

A programok nem igényelnek telepítést. Egy-egy „exe" kiterjesztésű futtatható fájlból állnak, melyeket csak el kell indítani. A programok írásjogot igényelnek abban a mappában, amelyből indítják őket.

SudokuSolver

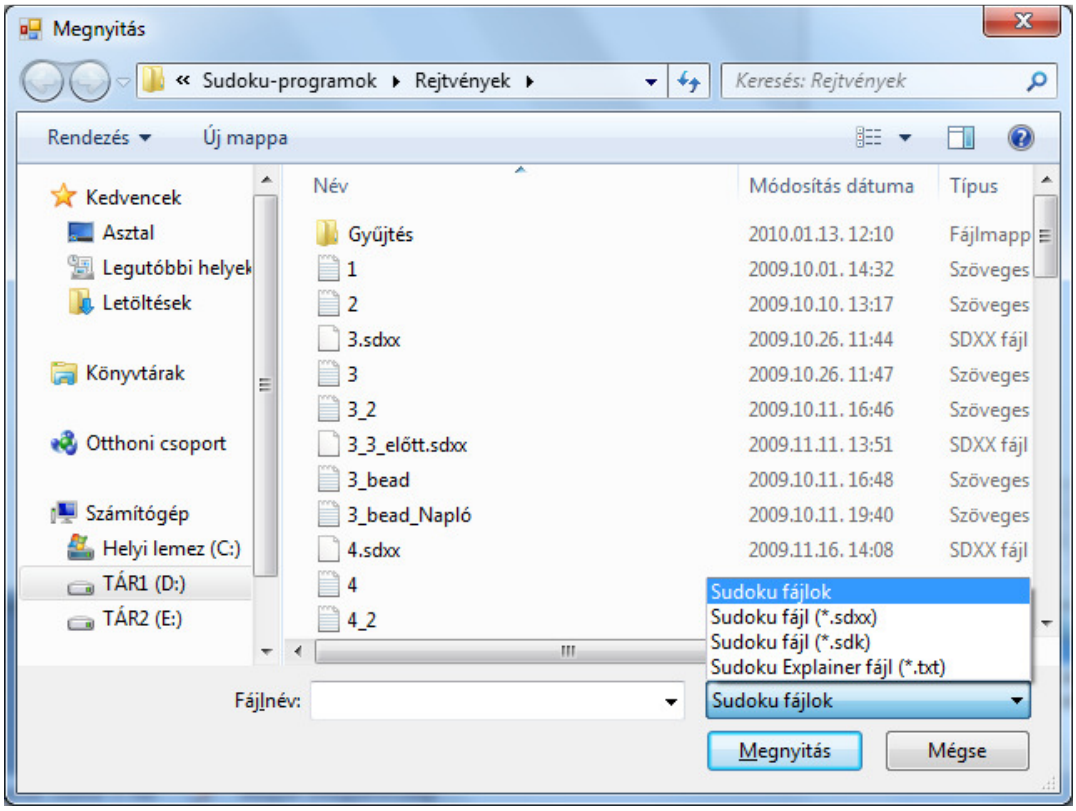
A program elindítása után a következő ablak tárul elénk:



Ebben a helyzetben csak a Fájl menüben található menüpontokkal tudunk továbblépni. Megnyithatunk egy rejtvényt fájlból, létrehozhatunk egy új üres rejtvényt, melynek méretét mi adhatjuk meg vagy kérhetünk véletlen rejtvényt is.

Fájl megnyitása

Ha fájlt akarunk megnyitni, az operációs rendszer beépített fájlkereső ablakra tűnik fel.



Látható, hogy a program három féle fájlt tud megnyitni.

Ezek a következőek:

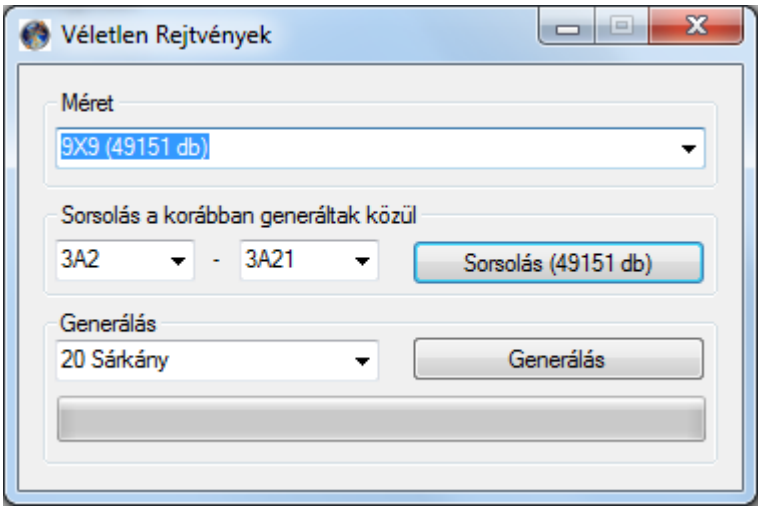
- TXT: SudokuExplainer fájltípusa: (9×9)-es tábla és (4×4)-es tábla mentésére alkalmas. Karakteres módón van benne a tábla letárolva. Számmal jelzi a beírt jelöltet és ponttal, azt ahova nincs beírva.
- SDK: Sudoku fájltípusa, specifikációja megegyezik az Explainer féle TXT-vel.
- SDXX: Sudoku fájltípusa. Ezzel tetszőleges méretű rejtvény eltárolható, ezen kívül a kitöltetlen cellákban a megengedett jelölteket is eltárolja, így

elmenthető a jelölt törlő algoritmusok végeredménye is. A cellákat „space” karakterrel választjuk el egymástól. Ha a cella ki van töltve, akkor a beírt jelölttel reprezentáljuk a cellát. Ha nincs kitöltve, akkor kapcsos zárójelen belül ’;’ jellel elválasztva szerepelnek a megengedett jelöltek.

A fájl megnyitása után a főablakban megjelenik egy Sudoku tábla, ami a megnyitott rejtvényt ábrázolja. Fontosnak tartom megemlíteni, hogy a megjelenített tábla, nem kerülhet olyan állapotba, hogy egy adott házban többször szerepel ugyanaz a jelölt. Ez a beolvasás során hibát jelent, amit a program ki is jelez, de a rejtvény további beolvasásának sikeressége kétséges. Hasonló a helyzet az olyan ellentmondásokkal, amikben egy cellában nincs megengedett jelölt, vagy egy konkrét házon belül egy cella sem tartalmaz egy bizonyos jelöltet. Ezt az ellentmondást megjeleníti, majd megjelenik a hibaüzenet, és a rejtvény további részének beolvasása ezután is kétséges marad.

Véletlen rejtvény

A programban lehetőségünk van a korábban már erre a célra előkészített rejtvények közül sorsolni, vagy egy véletlenszerű rejtvény generálására is. A menüpont megnyitásakor az alábbi ablak tárul elénk.



Felül adható meg a tábla mérete. Minden méret mellé oda van írva zárójelben, hogy hány rejtvény áll belőle rendelkezésre.

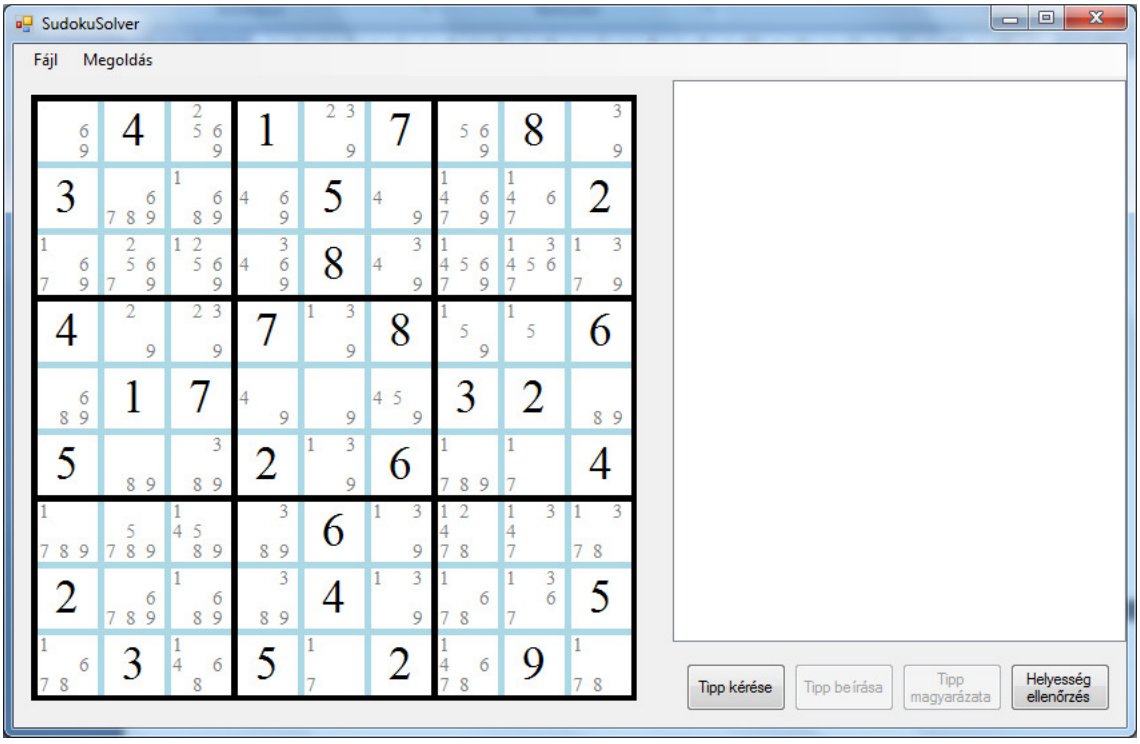
A sorsolás résznél a megfelelő méretből rendelkezésre álló rejtvényeket van lehetőségünk szűrni oly módon, hogy a sorsolandó rejtvény nehézségét felülről és alulról is korlátozzuk. A nehézség beállításánál a legördülő menüben, csak olyan

nehézségek vannak feltűntetve, amiből van is a rendelkezésre álló rejtvények között. A sorsolás gomba kattintva kisorsol nekünk egy rejtvényt, amely a kívánt nehézségi intervallumba esik. A gombon látható szám, a szűrésnek megfelelő rejtvények számát jelenti.

A generálás résznél van lehetőségünk egy gyors generálásra is. Itt azok előállítási ideje miatt csak kis rejtvényeket állíthatunk elő. Beállítható, hogy legfeljebb milyen nehéz legyen a rejtvény.

Rejtvény megnyitás után

Ha megnyitunk egy fájlt, vagy egy új rejtvényt az alábbi képet kapjuk.

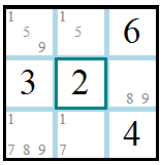


Megjelent a rejtvényt ábrázoló tábla a baloldalon. A jobb oldalon a „Tipp kérése” és a „Helyesség ellenőrzése” gomb aktívvá vált. Ezen kívül a menükben elérhetővé váltak az idáig inaktív funkciók is, de ezeket majd később részletezem.

Tábla használata

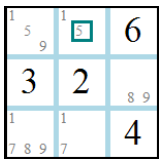
Ahogy az egeret mozgatjuk a tábla felett, észrevehetjük, hogy az egér pozíciója alatt feltűnik egy zöld négyzet, ami a tábla aktív celláját, vagy ha az nincs kitöltve, annak az egér alá eső jelöltjét foglalja magába. Ez szimbolizálja a kijelölést a táblán.

Beírt szám van kijelölve



Ez esetben csak az egér jobb gombjával tudunk kattintani. Ennek eredményeként a beírt számot kitörli a program és az megint csak megengedett jelölt lesz. Ilyenkor a cellába visszakerülnek az ott megengedhető jelöltek és a cellához tartozó házak minden olyan cellájába, ahol ez nem ütközik más beírt cellákkal visszakerül a kitörölt szám a megengedett jelöltek közé.

Megengedett jelölt van kijelölve

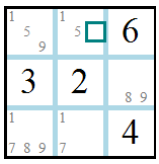


Ez esetben 2 lehetőségünk van.

Kattinthatunk az egér bal gombjával. Ekkor a jelöltet a cellába beírjuk és elvégezzük az összes olyan változást a táblán, ami ezzel jár.

Kattinthatunk az egér jobb gombjával. Ekkor a jelöltet töröljük a megengedett jelöltek közül. Ilyet akkor teszünk, ha rájövünk, hogy az a jelölt abba a cellába nem kerülhet bele.

Törölt jelölt van kijelölve



Ebben az esetben a jelöltek sorfolytonos elhelyezkedéséből tudjuk, hogy ez melyik jelöltnek a helye. Ha az egér bal gombjával kattintunk, azzal megpróbáljuk a jelöltet visszaírni a cellában a megengedett jelöltek közé. Ez persze csak akkor fog sikerülni, ha az adott jelölt a cellát magukba foglaló házak egyikébe sincs beírva.

A módszer előnyei

A táblán csak olyan jelöltek szerepelnek, melyeknek a beírásával nem juthatunk ellentmondásba.

Megoldhatatlanság

Az ellentmondás mentesség még nem jelenti, hogy a tábla megoldható. Ha a kitöltéssel eljutunk egy olyan állapotba, hogy az egyik cellából kifogynak a megengedett jelöltek, vagy van olyan jelölt, ami egy ház egyetlen cellájában sem megengedett, egyértelműen megoldhatatlanná vált a rejtvényünk. Ezt a program jelzi.

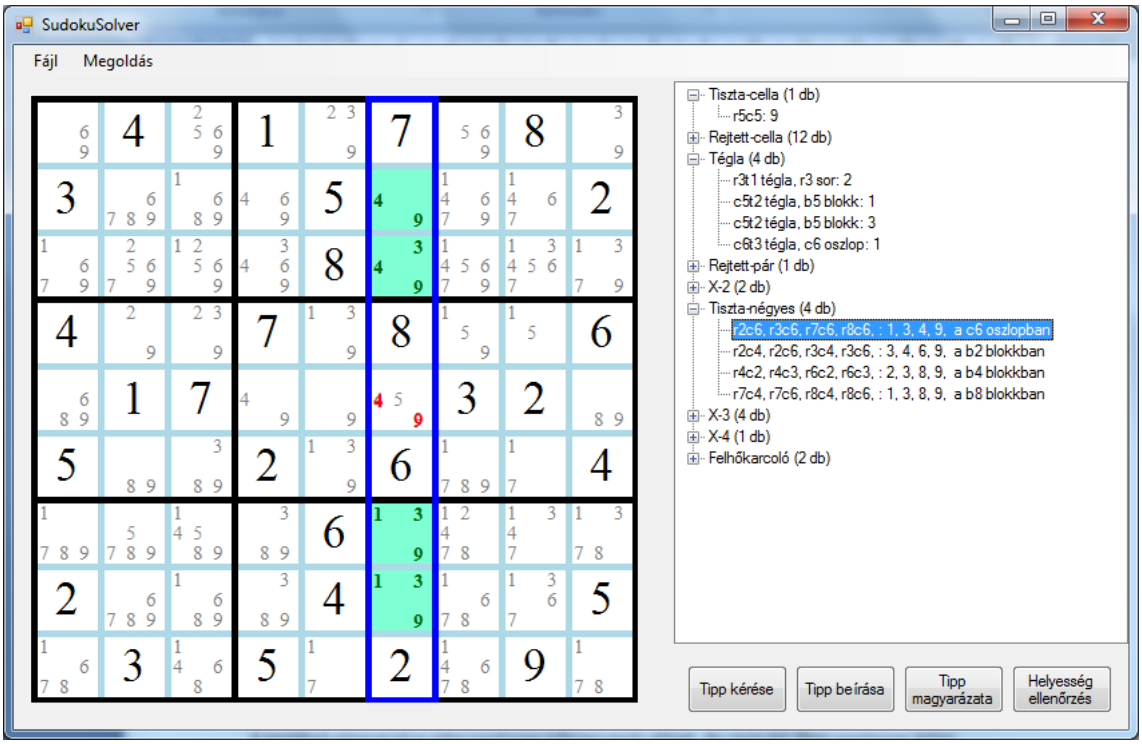
Gombok használata

A gombok elnevezése elég pontosan kifejezi azok céljait, de nem árt őket részletesebben leírni.

Tipp kérése

A rejtvény adott állapotában lehet a megoldás felé vezető tippeket kérni. Ezek a tippek a programba beépített 20 algoritmus által talált lehetőségek egy listáját jelenti.

A gomb megnyomása után megjelenik egy lista a tippekről, a táblán pedig kirajzolódik a lista kiválasztott tippje.



A listán a tippek csoportokba vannak szedve aszerint, hogy a tipp melyik algoritmus futása során „született”.

Észrevehetjük, hogy e gomb megnyomása után vált a középső két gomb is aktívvá.

Nagyobb rejtvények esetén a tippek előállítása sokáig zajlik. Ennek végét a "Tipp kérése" gombon olvasható "Leállítás" szöveg jelzi. Ameddig a gombra vissza nem kerül az eredeti felírat, addig dolgozik a tippgenerálás.

Tipp beírása

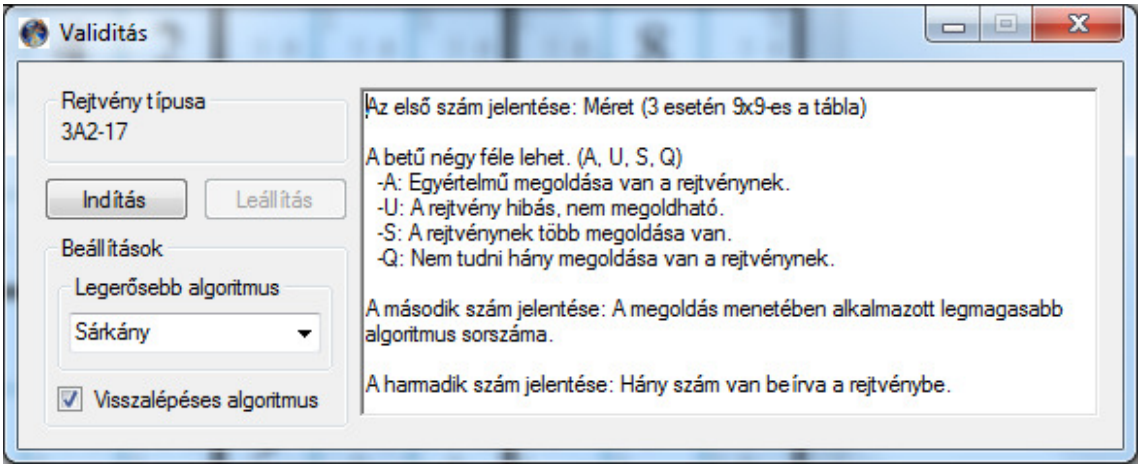
A táblán éppen megjelenített tipp szerint módosítja a táblát. Ez tiszta és rejtett cella esetén jelölt beírását jelenti, míg a többi algoritmus esetén a pírossal jelölt jelöltek törlését.

Tipp magyarázata

Az éppen kijelölt tipp szöveges magyarázatát jeleníti meg egy külön ablakban.

Helyesség ellenőrzés

A gomb megnyomásakor az alábbi ablak tárul elénk.



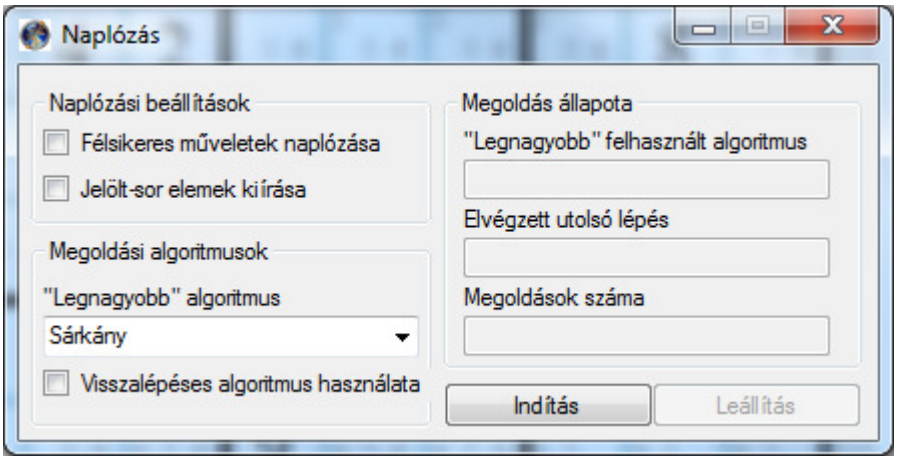
A háttérben a beépített algoritmusokkal megpróbálja megoldani a rejtvényt. A megoldás eredményeképpen megkapjuk a rejtvény típusát. Ebből az adatból sok információ megtudható, többek között az is, hogy a program meg tudja-e oldani a rejtvényt, de még a rejtvény megoldásának nehézségéről is kapunk információt. A nehézségre a „második szám”-ból következtethetünk. Minél több algoritmus kell a megoldáshoz, annál nehezebb a rejtvény.

Lehetőségünk nyílik a hosszú vizsgálat esetén leállítására és egy újabb elindítása előtt a vizsgálat paraméterezésére. Ilyenkor beállítható, hogy mely algoritmusig bezárólag használja a tesztelésre az algoritmusokat, és hogy a végén, ha szükség van rá, használhatja-e a visszalépéses algoritmust.

Megoldás menü

Megoldási Napló (Rózsa)

A menüpont kiválasztása után az alábbi ablak tárul elénk.



Az ablak bal oldalán nyílik lehetőségünk az elkészítendő napló paraméterezésére.

- Félsikeres műveletek naplózása

Ennek aktiválása nélkül a naplóba csak olyan sorok kerülnek be, ahol az épp aktuális műveletnek eredményeképpen változik a tábla. Ez rövid és átlátható. Ám azoknak, akiknek az algoritmus elsajátítása a célja sokat segít egy bonyolultabb algoritmus esetén látni azokat a helyzeteket, amikor majdnem minden megvan ahhoz, hogy a művelet sikeres legyen. Például egy sárkány megtalálása is nagy feladat, de haszontalan eredmény, ha a sárkánynak nincs zsákmánya. Ezek számítanak félsikeres műveletnek.

- Jelölt-sor elemek kiírása

Amikor egy algoritmus a jelölt-sort dolgozza fel, hasznos információ lehet, az hogy éppen melyik jelöltnél tart. Ez is egy olyan beállítás, ami drasztikusan meg tudja növelni a napló hosszát, viszont az algoritmus megértésében nagyon sokat segíthet.

- "Legnagyobb" algoritmus

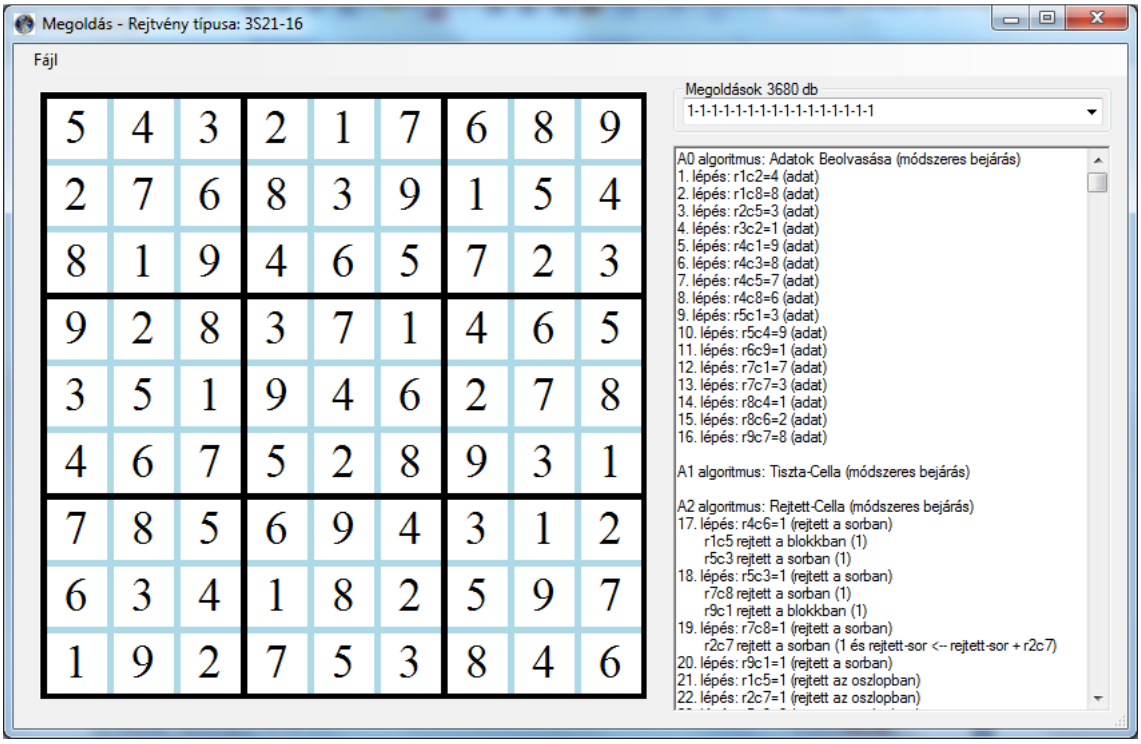
Itt a napló készítése során felhasználható legnagyobb sorszámú algoritmust adhatjuk meg. Azon felül, hogy így a hallgatók a saját tudásuknak megfelelő levezetést kérhetnek egy rejtvényhez, ez azért is fontos, mert nagyméretű tábla esetén lehet, hogy érdemesebb a sokáig tartó munkát elvenni a nagy műveletigényű és nagy sorszámú algoritmusoktól, hogy átadjuk a visszalépéses algoritmusnak.

- Visszalépéses algoritmus használata

Az algoritmus használata nélkül nem biztos, hogy megtaláljuk a megoldást, ugyanakkor adott esetben az is hasznos lehet, ha megnézzük, meddig jutnak el az alap algoritmusok.

A jobb oldalon találhatóak a megoldás állapotát kijelző komponensek. Hosszabb vizsgálat esetén sokat segít, ha van belelátásunk abba, hogy a megoldás halad-e a megoldás felé.

A lenti gombok segítségével indítható el, vagy állítható le a megoldás menete. Ha a megoldás befejeződik, függetlenül attól, hogy miért, az alábbi ablak tárul elénk. Itt hosszabb elemzés esetén várni kell egy kicsit, mert a napló beolvasása hosszadalmas lehet.



Bal oldalt található a megoldott rejtvény, vagy sikertelenség esetén az az állapot, ameddig az algoritmusok eljutottak. Hibás rejtvény esetén a hiba észleléséig végbement változtatásokat tartalmazza.

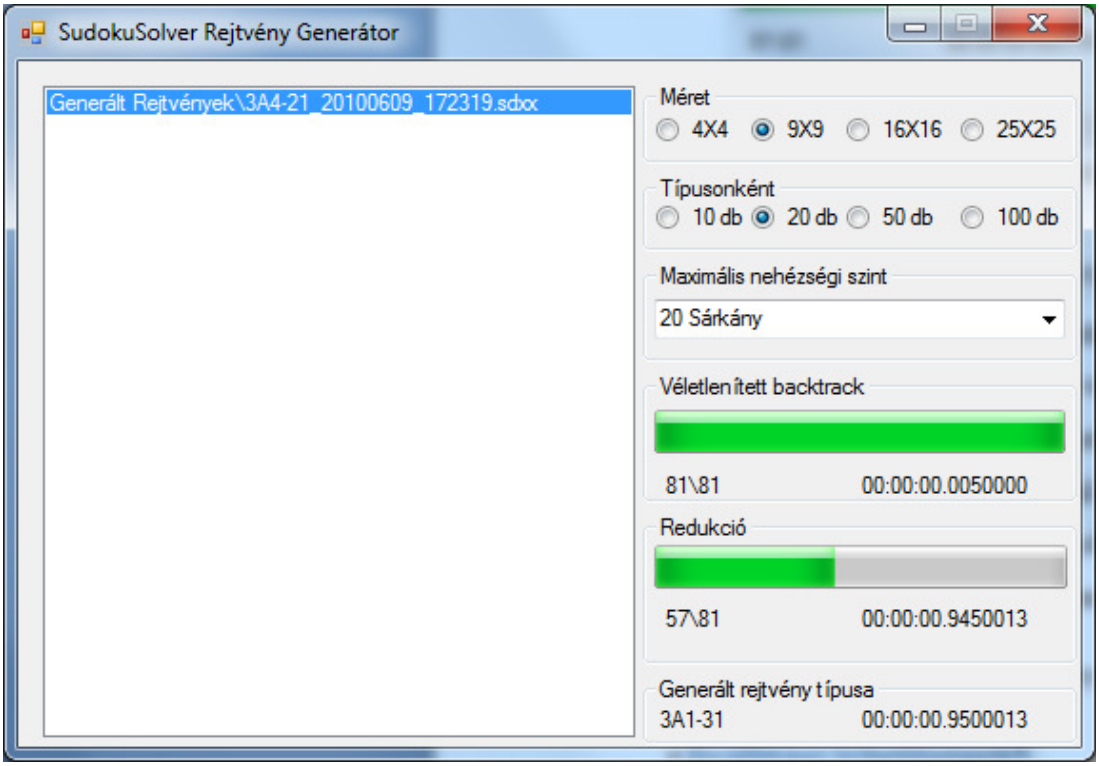
Jobb oldalt kapjuk a megoldás naplót. Ez saját tapasztalataim alapján is nagy segítséget jelent a tárgy beadandóinak elkészítésében, hiszen igen sok változó rejlik a háttérben, amelyek fejben kezelése könnyedén hibákhoz vezethet.

Felül a megoldások listája látható. A számozásuk a visszalépéses algoritmus elágazásait jelölik. Minden szám egy elágazás és a szám értéke az adott elágazásbeli ág sorszámát jelenti. A képen egy Royle féle 17-es rejtvényből töröltem egy elemet és arra futtattam a megoldó algoritmusokat. Lett volna több megoldás is, de nem vártam ki.

A fájl menüben lehetőségünk van a napló „RTF” fájlba történő lementésére és az épp megjelenített rejtvény lementésére.

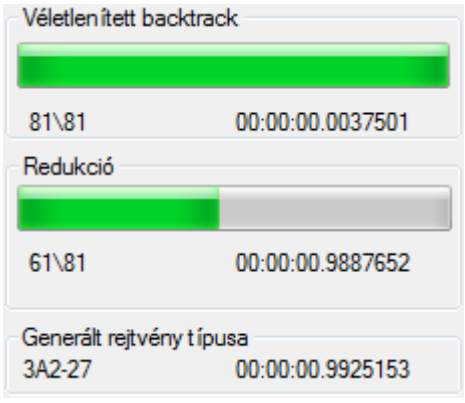
Rejtvény Generátor

Elindítás után ez az ablak jelenik meg, amely az egyetlen megjelenítő felület a programnak.



A program a leállításáig folyamatosan generálja a rejtvényeket. Az elkészített rejtvényeket bal oldalt a listán láthatjuk.

Generálási információ

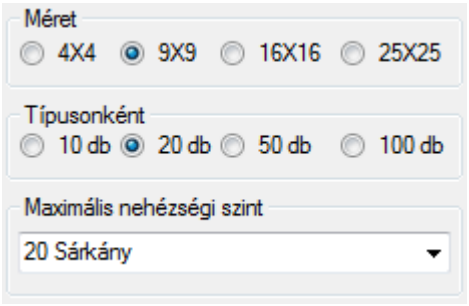


A rejtvény előállítása két lépcsős folyamat. Ennek első eleme a véletlenszerű helyes kitöltés előállítása. Ennek pillanatnyi állását jeleníti meg a felső csík.

A második lépcső a redukció, melynek során a cellák tartalmát véletlenszerűen töröljük. Egy-egy törlés után le kell tesztelni a rejtvényt, hogy milyen nehéz megoldani. Ha túl nehéz, akkor azt az elemet vissza kell írni. Elemenként egyszer elindul a megoldó algoritmus, de a futási ideje attól függ, hogy hány algoritmust kell használnia. Minél inkább telítődik a csík, annál biztosabb, hogy a rejtvény a legutóbb törölt elem törlése után nehezen megoldható vagy több megoldása van. Ezért a csík telítődésének sebessége folyamatosan lassul.

Alul látható a generált rejtvény pillanatnyi típusa, valamint a teljes generálási idő.

Beállítási lehetőségek



Első beállítási lehetőségünk a rejtvény mérete. Ennek megváltoztatásakor leállítódik a generálás, majd újra elindul az új mérettel és az ottani alapértelmezett algoritmuslistával.

A képen alul látható a Maximális nehézségi szint. Itt állítható be, hogy az előállítandó rejtvény legfeljebb milyen nehéz legyen. A mostani beállítás alapján $3A20$ a legnehebb. Ez azonban csak felső korlát, azt semmi sem biztosítja, hogy tényleg ilyen nehéz lesz.

Középen tudjuk beállítani, hogy típusonként maximum mennyi rejtvényt állítson elő. A típusba beletartozik a kitöltött elemek száma is. Ha a program egy olyan rejtvény generál, amilyenből már megvan a megfelelő darabszám, akkor azt eldobja. Ha a darabszám még nem telt meg, akkor is előfordulhat, hogy egy rejtvényt eldob. Ennek az az oka, hogy sikerült az egyik korábbi rejtvénnyel megegyezőt generálni.

Generált fájlok

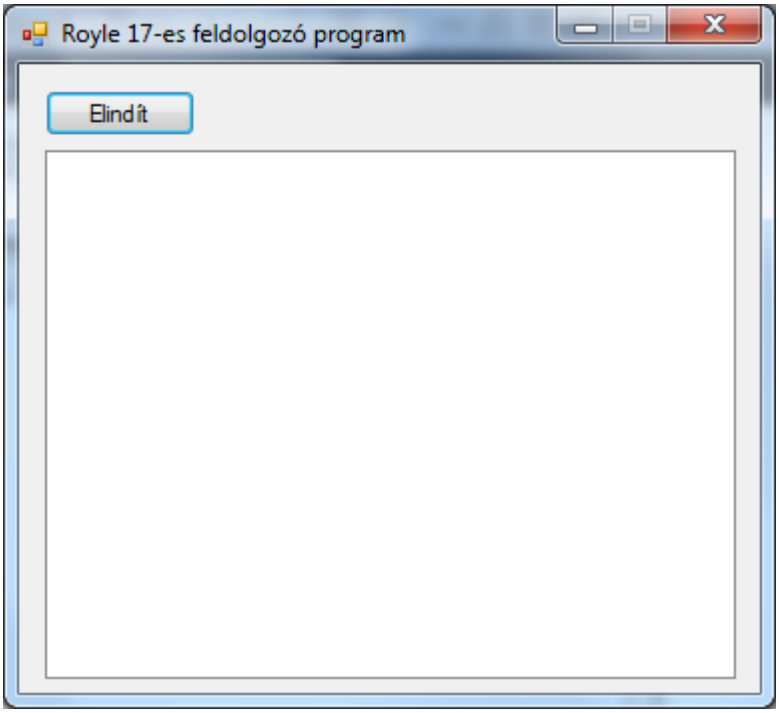
A generált állományok "SDXX" formátumúak. Neveik a rejtvény típusából és a generálás dátumából állnak.

A SudokuSolver "Generált Rejtvények" mappába lehet saját rejtvényt is tenni. Ehhez csak két feltételt kell teljesítenie.

- Az állomány neve ilyen legyen: "3A12-27_Gézuka_rejtvénye.sdxx". A rejtvény típusának meg kell egyeznie a rejtvény tényleges típusával, mert a SudokuSolver a sorsolást ez alapján végzi. A "Gézuka_rejtvénye" rész tetszőleges módon változtatható.
- A fájlnak nem csak a kiterjesztése kell, hogy "SDXX" legyen, de a tartalmának is az "SDXX" fájltípusnak megfelelőnek kell lennie.

Royle 17-es feldolgozó program

A program egyszeri felhasználásra van tervezve, így hibaellenőrzés egyáltalán nincs benne. Elindítás után az alábbi egyszerű ablakot látjuk meg.



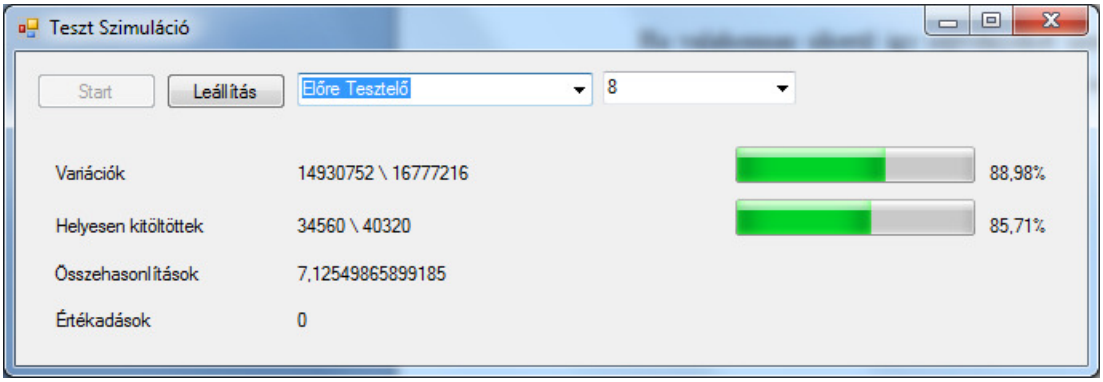
Az elindítás gomb megnyomása után egy kis ablakban kiválaszthatjuk a megnyitandó állományt. Amennyiben ez a művelet sikeres, el is indul a feldolgozás. Ezt leállítani csak a program leállításával együtt van mód. A lista ablakban folyamatosan jelennek meg a feldolgozott rejtvények.

A teljes feldolgozás a mai gépek sebességével fél óra alatt elkészül, és 40 $megabyte$ adatot hoz létre.

Az inputfájlnak soronként kell tartalmaznia $4 * 4$-es vagy $9 * 9$-es rejtvényeket. Egy sorban a cellák sorfolytonosan szerepelnek. Cellánként egy szám van a sorban. Az üres cellák helyén 0, a többi cella helyén pedig a beírt szám van.

Ha valahonnan sikerül így rejtvényeket szerezni, akkor azt ezzel a programmal fel lehet dolgozni, igaz az elkészített rejtvények nevében a "Royle17" kifejezés szerepelni fog.

Szimuláció - Összes lehetséges bemenetre

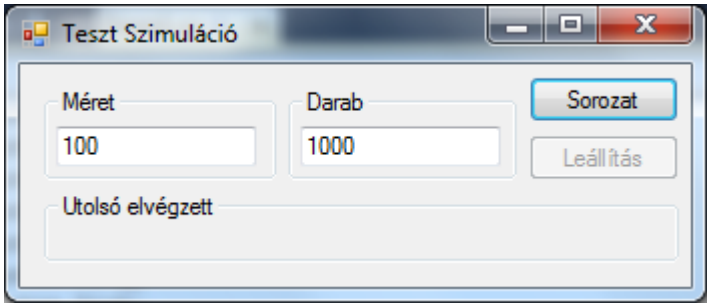


Ez a program a tesztelő algoritmusok átlagos műveletszámát számolja ki adott méreten az összes bemenő adatra. Megadható a tesztelendő algoritmus és a kívánt méret is.

A méretnél a felső határ a 15. Ennek egyszerű az oka. 16-os méretnél 16^{16} darab bemenő adat van, ami egyel több mint amekkorát 64 bites számban el lehet tárolni. Lehetne ugyan speciális számtípussal megoldani a problémát, de az lassítaná a feldolgozást, arról nem is beszélve, hogy a 11 hosszú sorozatok feldolgozása is fél napba került.

A szimuláció eredménye a "teljes.csv" fájlba kerül, amit táblázatkezelő és adatbázis kezelő programokkal meg lehet nyitni.

Szimuláció - Véletlenszerű bemenetekre



Ez a program nagyobb méretekre próbálja előállítani azokat az eredményeket, amelyeket az előző szimuláció. Mivel nagy méretek esetén túl sok bemenő adatot kéne feldolgozni, ezért véletlenszerű mintákat dolgoz fel, melyeknek a darabszámát mi adhatjuk meg.

Az algoritmusok összehasonlításához szükséges, hogy ugyanazokat a bemenő adatokat dolgozzák fel. Ezért a feldolgozás iterációkban történik. Egy iteráció első lépése az input előállítása, majd ezután az összes algoritmussal való feldolgozás.

A programot elsősorban diagramok előállítására használtam, ezért miután egy iteráció lefutott automatikusan elindul a következő, amelyben a méretet a kezdetben megadott mérettel növelem.

A folyamatot a leállítás gombbal állíthatjuk le. Ilyenkor befejeződik az éppen futó iteráció és utána indíthatunk csak újat. Ha a programot kikapcsoljuk, ugyanez megtörténik a háttérben.

Az eredmények a másik szimulációhoz hasonlóan módon a "véletlen.csv" fájlba kerülnek elmentésre.

Összegzés

6 darab algoritmust vizsgáltunk meg, amely eldönti egy n elemű sorozatról, hogy annak minden eleme különböző-e. Ezek átlagos műveletigényét majdnem minden esetben sikerült matematikai úton is meghatározni.

Szimulációs vizsgálatokkal megerősítettük az elméleti eredményeket. Ezen felül sikerült a műveletigények nagyságrendjéhez egy-egy konstanst társítani.

A tanszék tudományos munkájához is sikerült hozzájárulnom a HÁTRA algoritmus átlagos műveletigényének linearitásáról szóló bizonyítással és az algoritmusok szimulációs programjának elkészítésével, melyek segítségül szolgáltak egy cikk [9] megírásában, valamint alapjául szolgálnak egy júliusban, Komáromban tartandó előadásnak, amelynek szerzői Iványi Antal, a témavezetőm és jómagam leszünk [11].

A tanszék munkájához a SudokuSolver program továbbfejlesztésével is hozzájárultam, amely azon felül, hogy kényelmes felületet nyújt rejtvények megoldásához, sokat segít a hallgatóknak a tananyagban szereplő algoritmusok megértésében. A rejtvény generátor a tárgy oktatójának is nagy segítség, hiszen a különböző algoritmusokra így sokkal könnyebben találhat példát.

A diplomamunka az ELTE "TÁMOP-4.2.1.B-09-1-KMR-2010-0003" kódú pályázatának támogatásával készült.

Irodalomjegyzék

[1] Balázs Viktor: Sudoku megoldó program, BSc szakdolgozat, Eötvös Loránd Tudományegyetem, Budapest 2007.

[2] Cormen Tomas H., Leiserson Charles E., Rivest Ronald R., Stein Clifford: *Új algoritmusok,* Scolar kiadó

[3] Crook, J. F.: A pencil-and-paper algorithm for solving Sudoku puzzles. *Notices Amer. Math. Soc.*, Vol. 56 (2009), no. 4, 460–468.

[4] Csizmazia Albert: "Sudoku" összefoglaló
http://progkor.inf.elte.hu/200506.2/BEAD/sudoku.htm

[5] Fekete István: Algoritmusok és adatszerkezetek című egyetemi jegyzetének bináris keresőfákkal foglalkozó fejezete. (Letöltve 2010. június 5-én.)
http://people.inf.elte.hu/fekete/docs_1/binaris_kereso_fak.pdf

[6] Felgenhauer Bertram, Jarvis Bertram: Mathematics of Sudoku I., 2006
http://www.afjarvis.staff.shef.ac.uk/sudoku/felgenhauer_jarvis_spec1.pdf
http://www.afjarvis.staff.shef.ac.uk/sudoku/

[7] Gordon Royle: Minimum Sudoku
http://mapleta.maths.uwa.edu.au/~gordon/sudokumin.php

[8] Iványi Antal, Kátai Imre: Estimates for speed of computers with interleaved memory systems, *Annales Univ. Sci., Budapest., Sectio Mathematica*, Vol. 19 (1976), 159–164.

[9] Iványi Antal, Kátai Imre: Quick testing of random sequences, *Proceedings of ICAI 2010* (benyújtva).

[10] Iványi Antal, Kátai Imre: Testing of random matrices, Annales Univ. Sci., Budapest., Sectio Computatorica (benyújtva).

[11] Iványi Antal, Novák Balázs: Testing of sequences by simulation, *In: Abstract of 8th Joint Conference on Mathematics and Computer Science* (Komárom 2010. július 14-16.)
http://selyeuni.sk/macs/

[12] Knuth Donald Ervin: Dancing links. In: *Millenial Perspectives in Computer Science* (ed. by Jim Davies, Bill Roscoe, and Jim Woodcock). Palgrave, Houndmills, 2000. 187–214.

[13] Lakatos Zoltán: Sudoku megoldó algoritmusok, BSc szakdolgozat, Eötvös Loránd Tudományegyetem, Budapest 2010.

[14] Varga Valéria: Sudoku, BSc szakdolgozat, Debreceni Egyetem, Debrecen 2007.
http://www.math.klte.hu/~tengely/tsz/uploads/Main/VargaValeriaDM.pdf