Osztott algoritmusok

A benzinkutas példa szimulációja

Müller Csaba

2010. december 4.

1. Bevezetés

Első lépésben talán kezdjük a probléma ismertetésével. Adott két n hosszúságú bináris sorozat (s_1, s_2), ezeket szeretnénk párhuzamosan feldolgozni két processzorral, vagy hogy a benzinkutas hasonlattal éljünk, a két sorban álló összes autót megtankolni. A két benzinkúton különböző típusú benzint lehet kapni $(0, 1)$, és minden sorban álló autó is csak egyféle benzint tankolhat, amit a neki megfelelő bináris szám mutat, röviden fogalmazva rögzített, hogy melyik autót hol lehet feltölteni.

Ebből adódóan a feldolgozásnak, tankolásnak vannak szabályai. Egy lépésben az alábbiakat tehetjük meg:

- megtankolhatjuk a két sor első elemeit, ha azok különbözőek
- megtankolhatjuk az első sor első két elemét, ha azok különbözőek
- megtankolhatjuk az második sor első két elemét, ha azok különbözőek
- megtankolhatjuk az első sor első elemét
- végül a második sor első elemét

Innen látható, hogy globálisan kell tekintenünk a sorozatokra, mert könnyen megeshet, hogy ha csak mohóan mindig próbálnánk párokat kiszolgálni, akkor a későbbiekben csak egyesével tudnánk feldolgozni az elemeket. Természetesen a célunk az, hogy minél kevesebb lépésben végezzünk a két sorozat összes elemével.

A továbbiakban ismertetünk egy polinomiális algoritmust az optimális kiszolgálás lépésszámának megtalálására, valamint 3 különböző közelítő algoritmust.

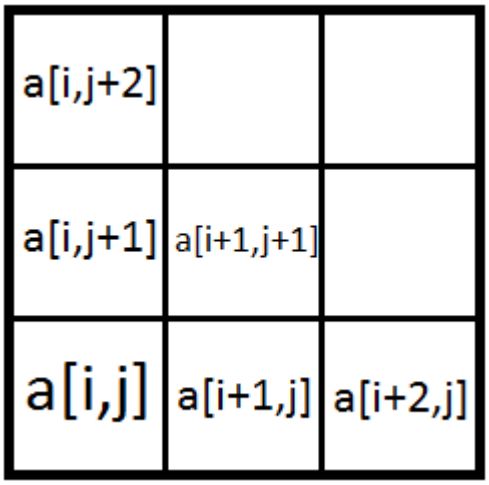
2. Algoritmusok

Akkor ahogyan a bevezetőben említettük, kezdjük is el az optimális algoritmussal.

2.1. Egy optimális algoritmus

A számítások során a dinamikus programozást hívjuk segítségül. Létrehozunk egy $(n+1) \times (n+1)$ méretű mátrixot, amelynek a_{ij} $(i, j = 0, 1, \ldots, n)$ eleme azt adja meg, hogy az első sorozatból az első i darab elem, a másodikból pedig az első j darab elem feldolgozásához (minimum) hány lépés szükséges. Így a feladatunk most az lesz, hogy meghatározzunk a_{nn} értékét, és ez lesz az optimális lépésszám. A kérdés már csak az, hogy ezt hogyan tegyük meg!

A válasz pedig a következő: a_{00} értékét tudjuk, hogy 0. Innen tovább a dinamikus programozással lépünk. Minden lépésben bizonyos pontokból egy lépésben elérhető pontokat keresünk, amit az alábbi ábra segítségével könnyebben leírhatunk.



1. ábra. Adott pontból egy lépésben elérhetőek vizsgálata

Tehát az ábrán $a[i, j]$-vel jelöltük az adott vizsgálandó pontot a mátrixban. Továbbá a megjelölt elemek jelentik azokat, amelyeket egy lépésben elérhetünk, megfelelő feltételek mellett, ezek pedig a következőek:

- $a[i, j + 1]$ és $a[i + 1, j]$ esetén annyi a feltétel, hogy még a mátrixban legyenek, tehát teljesüljön, hogy $j + 1 \leq n$, valamint $i + 1 \leq n$, ugyanis ezek az elemek felelnek meg annak, hogy az egyik sorozatból dolgozunk fel 1 elemet, és azt bármikor megtehetjük

- $a[i, j + 2]$ esetén a feltétel szintén, hogy még benne legyen a mátrixban, tehát $j + 2 \leq n$, valamint szükséges még az is, hogy $s_2[j + 1] \neq s_2[j + 2]$, azaz a második sorozat következő két eleme különböző legyen

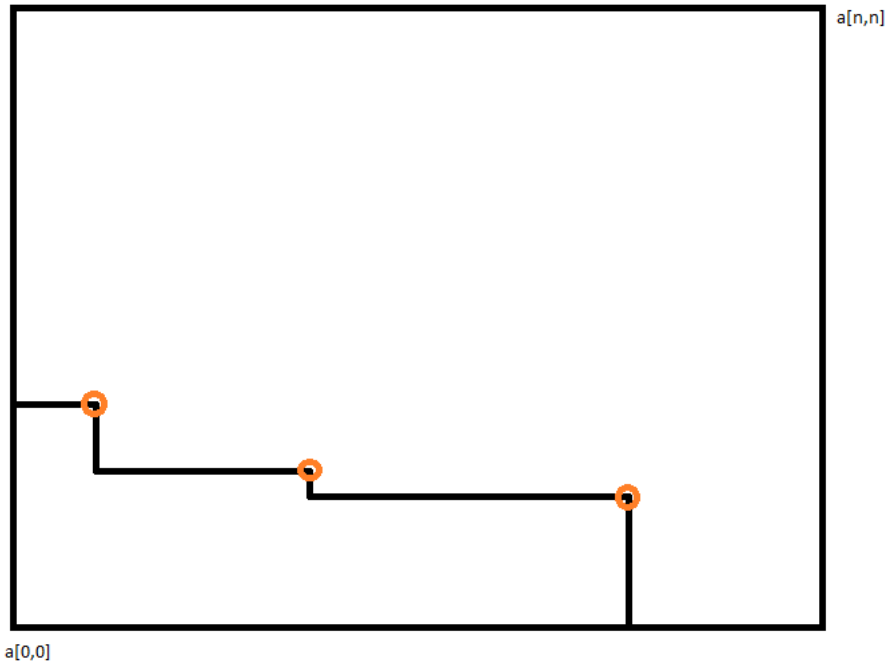
- $a[i + 2, j]$ esetén hasonlóak a feltételek, csak most az első sorozatra vonatkozóan kell, hogy $s_1[i + 1] \neq s_1[i + 2]$ teljesüljön

- végül $a[i + 1, j + 1]$ esetén pedig kell, hogy $i + 1 \leq n, j + 1 \leq n$, továbbá hogy $s_1[i+1] \neq s_2[j+1]$, mivel ez felel meg annak, hogy a két sorozatból 1–1 elemet dolgozunk fel

Tehát ha a pontok megfelelnek az előző feltételeknek, akkor az értékük $a[i, j] + 1$ lesz. Viszont továbbra sem világos, hogy milyen sorrendben határozzuk meg a mátrix elemeit, ehhez nyújt segítséget a 2. ábra.

Minden lépésben meg kell határozni az ábrán látható módon egy „burkát” az eddig meghatározott pontoknak, és csak a pirossal bejelölt csúcsokban kell elvégezni az előzőekben leírt számításokat, ugyanis ha a pirossal jelölt csúcs a_{ij}, akkor $a_{i-1,j}$ és $a_{i,j-1}$ nem lehet jobb, az adott pontnál, mivel az értékük a mátrixban megegyezik, viszont az egyik sorozatból eggyel kevesebb elemet dolgozott fel az adott pont. Részletesebb ismertetésért lásd [1].

Ezek után térjünk rá a közelítő algoritmusok tárgyalására.



2. ábra. A vizsgálandó pontok

2.2. Közelítő algoritmusok

Mint a bevezetésben is említettük 3 különböző algoritmust fogunk tárgyalni, kezdjük is el a legegyszerűbben leírhatóval, a `SIM-k` algoritmussal.

2.2.1. `SIM-k`

Valójában egy egész algoritmus családról beszélünk, ugyanis a nevében szereplő k egy paramétere az algoritmusnak. A működése nagyon egyszerű, a sorozatokat felosztja k hosszúságú részsorozatokra – $\left\lfloor \frac{n}{k} \right\rfloor$ darab – és ezekre futtaja le az optimális algoritmust, a végén megmaradó elemeket pedig egyenként szolgálja ki.

2.2.2. `MEM-k`

Szintén egy paraméteres algoritmusról beszélünk. Az algoritmus minden lépésben mindkét sorozat k hosszúságú szeleteit $(s_1[i], \ldots, s_1[i+k-1]$, $s_2[i], \ldots, s_2[i+k-1])$ vizsgálja, és minden lépésben megpróbálja a lehető legnagyobb prefixet kiszolgálni páronként.

Első lépésben megvizsgálja, hogy kiszolgálható-e a két szelet k lépésben, ha nem, akkor megvizsgálja a két szelet $k-1, k-2, \ldots, 1$ hosszúságú prefixeire is, hogy kiszolgálhatóak-e $k-1, k-2, \ldots, 1$ lépésben, az első olyan érték, amelyre teljesül, azt kiszolgálja, és továbblép, méghozzá úgy, hogy hozzáveszi a szelethez a sorozatok következő elemeit, hogy megint k hosszú legyen. Ha az egy hosszúságú kezdőszelet sem szolgálható ki párban, akkor kiszolgálja őket 2 lépésben, és szintén továbblép (hozzáveszi a sorozatok 1–1 elemét a szeletekhez).

Ennél az algoritmusnál is igaz, hogy amikor már nincs k hosszúságú rész, akkor átvált egyenkénti feldolgozásra.

Megjegyeznénk, hogy a szimulációban csak $k \leq 5$ értékekre valósítottuk meg, mert azt eldönteni, hogy kiszolgálható-e a kezdőszelet optimálisan páronként, bonyolult eldönteni, legalábbis programot írni rá meglehetősen komplikált lenne.

2.2.3. `ALT-k`

A harmadik, és egyben utolsó közelítő algoritmusunk szintén egy paraméteres algoritmus.

Szintén a két sorozatból k elemet lát előre, de ennél az algoritmusnál már nem „szinkronban” történik a feldolgozás, mint az előzőeknél, hanem lehet, hogy a eltérő számú elemet dolgoztunk fel adott időpillanatig.

A működése során megpróbál minden lépésben egy párt feldolgozni, vagy az egyik sorozat elejéről, vagy a két sorozat első elemeit. Ha ezt nem tudja

megtenni, akkor az egyik sorozat első elemét dolgozza fel. Megjegyzésként beszúrnánk, hogy ha nem tudja megtenni, az csak úgy lehet, ha mindkét sorozat első 2–2 eleme megegyezik. Ezért ilyenkor az alapján döntünk, hogy melyik sorozatban következik hamarabb ezektől különböző elem (természetesen csak a k hosszúságú részt tekintve).

Így még nem feltétlenül egyértelmű a feldolgozás, ezért az algoritmusban szerepel két számláló is, α_t, β_t, amelyek azt fejezik ki, hogy a t időpillanatban az első, illetve a második sorozatból hány elemet dolgoztunk fel. Szóval ha olyan eset adódik, amikor nem lenne egyértelmű a lépés, akkor az $|\alpha_t - \beta_t|$ minimalizálásával döntünk, ha esetleg ez sem döntene (pl. $\alpha_t = 4, \beta_t = 5$, és mindhárom pár feldolgozható), akkor az első sorozatot részesítjük előnyben.

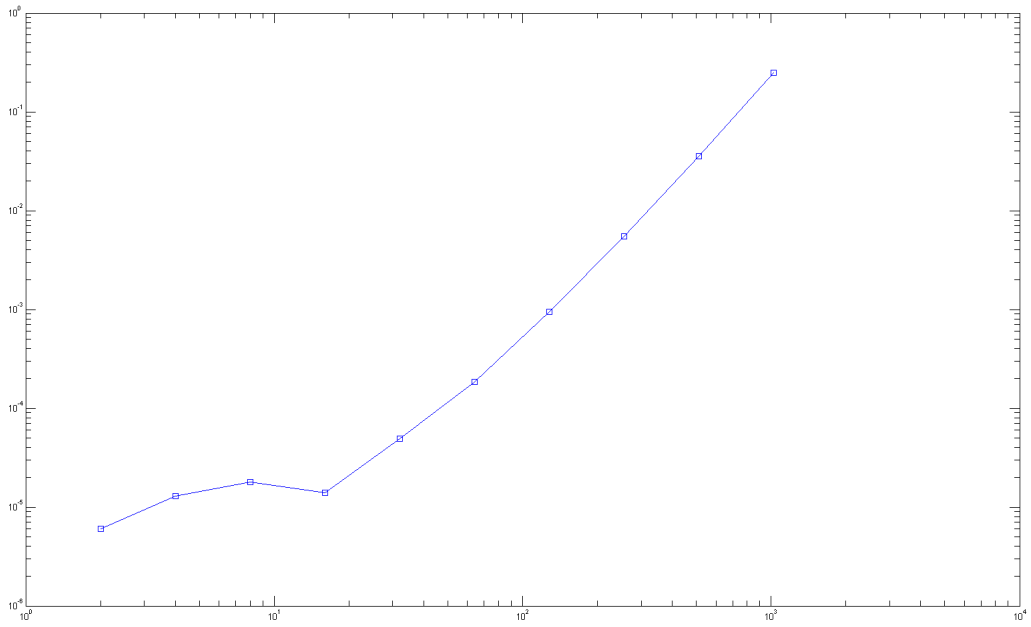
Továbbá amint az egyik sorozat rövidebb lesz, mint k, az algoritmus a maradékot egyenként dolgozza fel.

Megjegyeznénk még, hogy a k paraméter növelésével az algoritmusok által meghatározott lépésszám tart az optimális értékhez. (Részletekért lásd [1].)

3. Szimulációs eredmények

A szimuláció során adott hosszúságú, meghatározott számú véletlen sorozatra futtattuk le az algoritmusokat, és vizsgáltuk az átlagos futásidőket, valamint a meghatározott lépésszámok átlagát. Ezt rövid sorozatok esetén még az optimálissal is összevetettük.

A szövegnél talán érdekesebbek a különböző grafikonok, tehát lássuk is ezeket. Először az optimális algoritmus átlagos futásideje különböző n-ekre, minden esetben 1000 véletlenül generált sorozatpárral.



3. ábra. Optimális algoritmus átlagos futási ideje 1000 futtatással log-log skálán

Megjegyzésként még annyit megemlítenénk, hogy az optimális algoritmust lefuttattuk még $n = 2048$ és $n = 4096$ értékekre is, itt viszont már csak 10, illetve 2 véletlen sorozatra. A kapott átlagos futásidők: 2.4525, valamint 20.008. ($n = 1024$ esetén 0.248 az érték.) Valamint még azt jegyeznénk meg, hogy a közelítő algoritmusok azért nem szerepelnek ezen a grafikonon, mert még $n = 4096, k = 5$ esetén is elhanyagolható az átlagos futásidejük (0.004).

Most hasonlítsuk össze a közelítő algoritmusokat futásidő, és eredmény szempontjából. Szintén 1000 darab véletlen sorozatra $k = 5$ esetén.

n	`SIM` idő	lépés	`MEM` idő	lépés	`ALT` idő	lépés
1000	0.00135	1254	$3.4e-5$	1206	$3.2e-5$	1124
2000	0.00266	2508	$7e-5$	2409	$5.8e-5$	2243
4000	0.00518	5016	0.00014	4814	$9.3e-5$	4468
8000	0.0108	10033	0.00029	9638	0.000167	8949
16000	0.0204	20065	0.00056	19276	0.00038	17890
32000	0.0418	40135	0.0012	38609	0.0007	35770
64000	0.0819	80113	0.0023	76881	0.0014	71502
128000	0.1633	160638	0.0046	154366	0.0029	142884
1024000	1.275	1286204	0.037	1236406	0.0224	1144304

A táblázatból láthatóak alapján elmondhatjuk, hogy az `ALT-k` algoritmus átlagosan gyorsabb a másik kettőnél, és még jobb eredményeket is produkál, jobbat szinte nem is várhatnánk el tőle.

Most vizsgáljuk meg a közelítő algoritmusoknál a k paraméter változtatásának hatását $n = 2000$ mellett, 1000 futtatásra.

k	`SIM` idő	lépés	`MEM` idő	lépés	`ALT` idő	lépés
1	0.0039	2999	$3.4e-5$	2999	$1.3e-5$	2666
2	0.0032	2756	$3.5e-5$	2601	$3.9e-5$	2291
3	0.0029	2626	$4.6e-5$	2455	$4.6e-5$	2255
4	0.0029	2557	$5.8e-5$	2423	$4.7e-5$	2245
5	0.0028	2508	$8e-5$	2409	$4.3e-5$	2243
10	0.0031	2386	–	–	$3.7e-5$	2246
20	0.0023	2296	–	–	$5.1e-5$	2255
50	0.0048	2216	–	–	$4.6e-5$	2281
100	0.01	2175	–	–	$3.8e-5$	2325

Ebből a táblázatból azt szűrhetjük le, hogy a `SIM-k` algoritmusra jó hatással van k növelése, ahogyan ez várható is, viszont az `ALT-k` algoritmusnál már kis k értékekre is jó eredményeket produkál, és kis hatással van

rá k növelése. Mondhatjuk, hogy ez is megfelel a várakozásoknak, mivel nagyon kevés olyan eset van, amikor végig kellene néznie a $2k$ elemet az algoritmusnak.

Továbbá jegyezzük meg azt is, hogy ugyan `SIM-k` egyre jobban viselkedik a paraméter növelésével, de mivel az optimális algoritmust futtatja le k hosszúságú sorozatokra, ezért gyorsan növekszik a futásideje k növelésével.

Végül $n = 14$ esetén az összes lehetséges sorozatra (valójában a felére) lefuttatjuk az optimális, és a három közelítő algoritmust $k = 3$ esetén.

Algoritmus	Átlagos futásidő	Átlagos lépésszám
Optimális	$1.94754e-5$	16.364
`SIM-3`	$1.59668e-5$	19.75
`MEM-3`	$2.93322e-7$	18.2046
`ALT-3`	$2.12632e-7$	18.706

Hivatkozások

[1] A. Iványi, I. Kátai, Modeling of priorityless processing in an interleaved memory with a perfectly informed processor, Autom. Remote Control 520-526 (1985).

[2] A. Iványi, I. Kátai, Estimates for speed of computers with interleaved memory systems, Annales Univ. Sci. Budapest, Sectio Mathematica, 159-164 (1976).