NYILATKOZAT

Név: Kulcsár Gergely

ELTE Természettudományi Kar, szak: Matematika

NEPTUN azonosító: X21UG4

Szakdolgozat címe:
Párhuzamos algoritmusok és paraméteres keresés

A **szakdolgozat** szerzőjeként fegyelmi felelősségem tudatában kijelentem, hogy a dolgozatom önálló szellemi alkotásom, abban a hivatkozások és idézések standard szabályait következetesen alkalmaztam, mások által írt részeket a megfelelő idézés nélkül nem használtam fel.

Budapest, 2022.05.30

a hallgató aláírása

Eötvös Loránd Tudományegyetem
Természettudományi Kar

Párhuzamos algoritmusok és paraméteres keresés

Kulcsár Gergely

Szakdolgozat
Matematika BSc

Témavezető:
Jüttner Alpár, tudományos főmunkatárs
Operációkutatási Tanszék

Budapest, 2022

Köszönetnyílvánítás

Köszönöm Jüttner Alpárnak, hogy segített megtalálni a dolgozat témáját és konzultációs alkalmaink során rengeteg érdekes kérdésbe nyerhettem betekintést. Köszönöm továbbá családomnak, hogy lehetővé tették tanulmányaimat és segítettek elkerülni a megőrülést a távoktatási időszak elején. Köszönöm Stadler Domonkosnak hogy átfutotta dolgozatomat és észrevételeivel érthetőbbé és olvashatóbbá tette azt. Végül, de nem utolsósorban köszönöm Zsigri Bálintnak, aki segített a dolgozatba kerülő ábrák készítéséhez használt alkalmazás elsajátításában és hogy a dolgozat jelentősen letisztultabbá vált azzal, hogy részeit átbeszéltük.

Tartalomjegyzék

Bevezetés

A párhuzamos algoritmusok kiemelt fontosságúak, ha a modern videókártyák, számítógép hálózatok és szuperszámítógépek számítási kapacitásait szeretnénk kihasználni. A dolgozatban a motiváció párhuzamos algoritmusok tárgyalására elméletibb jelentősségű, a Megiddo által [Meg79] és [Meg83a] cikkeiben bevezetett paraméteres keresési módszer formájában.

A dolgozat első felében paraméteres kereséshez és jelen dolgozatban szereplő alkalmazásaihoz szükséges ismereteket mutatunk be párhuzamos és szekvenciális algoritmusokról. Többek között látni fogjuk, hogy a - dolgozatban párhuzamos számításokhoz használt - PRAM modellben hogyan lehet összegezni, minimumot keresni és minimális költségű feszítőfát konstruálni. A PRAM modell különböző változataira és ezek közti összefüggésekre is kitérünk. Párhuzamos rendezésre rendező hálós megoldást fogunk tárgyalni, hogy lefektessük a dolgozat végén ismertetett, Cole [Col87] cikkében publikált gyorsításához szükséges alapokat.

A szakdolgozatban ismertetett paraméteres kereséssel bizonyos feladatokra - általában egy szekvenciális és egy párhuzamos algoritmust felhasználva - konstruálható hatékony szekvenciális algoritmus. Be fogjuk vezetni a standard alakot, ami egységes keretrendszert ad bizonyos típusú paraméteres keresési algoritmusok bemutatására. Ezt felhasználva többek között látni fogunk algoritmust mozgó pontok halmazának minimális átmérőjének kiszámítására és arra, hogy súlyozott csúcsú fa esetén megtaláljuk a maximális λ^* súlyt, hogy k élt elhagyva minden komponens összsúlya legalább λ^*. Ezután rátérünk a standard alak egy könnyebben kezelhető speciális esetére, amivel törtlineáris kombinatorikai optimalizálási feladatokra mutatunk megoldásokat. A dolgozat végén specifikus gyakori esetekre mutatunk további technikákat, amikkel jobb futásidőket lehet elérni.

A paraméteres keresést nehéz hatékonyan implementálni, hiszen más algoritmusok futási módját manipulálja, így az ezzel létrehozott algoritmusok főleg elméleti jelentősségűek. Sok helyen helyettesíthető például a Newton módszerrel, vagy randomizált algoritmussal, viszont van, ahol ezzel valósítható meg optimális futásidejű algoritmus. A módszer gyakorlati alkalmazhatóságát növeli Oostrup és Veltkamp [OV02] keretrendszere, ami sok feladatra jelentősen egyszerűbbé teszi a módszer implementálását.

1. fejezet

Alapvető definíciók, jelölések és szekvenciális algoritmusok

Ebben a fejezetben bevezetjük a dolgozat többi részében használt alapvető definíciókat és jelöléseket és ismertetünk néhány később használt szekvenciális algoritmust.

1.1. Definíciók és jelölések

Algoritmusok futásidejét konstansoktól eltekintve vizsgáljuk ebben a szakdolgozatban. Fontos megjegyezni, hogy az ordó jelölés nagy konstansokat rejthet, mint például az AKS rendező hálónál, így az arra épülő eredmények inkább elméleti jelentősségűek.

Definíció. Adott $f, g : \mathbb{N} \to \mathbb{N}$ függvényekre $f \in O(g)$, ha $\exists\, c, M \in \mathbb{R} : \forall x \geq c : Mg(x) \geq f(x)$

Definíció. Adott $f, g : \mathbb{N} \to \mathbb{N}$ függvényekrere $f \in \Omega(g)$, ha $\exists\, c, M \in \mathbb{R} : \forall x \geq c : Mg(x) \leq f(x)$

A 3.2.3-beli feladat és az ehhez használt 1.3-ben leírt centroid dekompozíciós eljárás egyszerűbb tárgyalása érdekében vezessük be a következő definíciókat.

Definíció. Egy (F, v_0) **gyökeres fagráf**, egy F fa és v_0 kijelölt csúcsból áll, amit gyökérnek nevezünk.

Definíció. Gyökeres fagráfban egy **csúcs gyerekei** azon szomszédjai, amik a gyökértől nála messzebb vannak.

Definíció. Gyökeres fagráfban egy v **csúcs leszármazottjai** azon csúcsok, amikből a gyökérbe vezető út tartalmazza v-t.

Definíció. Gyökeres fagráfban egy v **csúcs szülője** a p csúcs, aminek v a gyereke.

Jelölés. A dolgozat során algoritmusok futásidejét T-vel, annak alsó indexes változataival, illetve $T(n)$-el fogom jelölni. Ez mindenhol függvénye a bemenet méretének, viszont a dolgozat második felében nem fogom kiírni a T függvény paramétereit.

Jelölés. $\log n := \max(\log_2 n, 1)$ a dolgozatban, hogy a futásidőszámítások egyszerűbbek legyenek.

Jelölés. $\oplus$ a bináris kizáró vagy-ot jelöli a dolgozatban.

Jelölés. A dolgozat során használjuk az $n := |V|, m := |E|$ jelöléseket a $G(V, E)$ gráf élei és csúcsainak számára.

Megjegyzés. A dolgozatban egyszerűbb futásidőszámítások kedvéért feltesszük, hogy a tárgyalt gráfok összefüggőek és egyszerűek, így $O(n) \leq O(m) \leq O(n^2)$

1.2. Lineáris idejű mediánkeresés

Ebben a fejezetben megmutatjuk, hogy n szám mediánja megtalálható $O(n)$ időben, a Cormen, Leiserson, Rivest és Stein által írt „Introduction to Algorithms Third Edition" [Cor+09] §II.9.3 alapján. Ezt az algoritmust a paraméteres keresést tárgyaló fejezetben fogjuk használni.

Az „Introduction to Algorithms Third Edition" [Cor+09]§II.9.2-ben is szereplő – legrosszabb esetben $O(n^2)$ futásidejű – quickselect algoritmus következő változatával n számnak $O(n)$ időben megtaláljuk a mediánját:

Algorithm 1.1: `median(x)` megtalálja az x n elemű tömb elemei közül az $\lfloor \frac{n}{2} \rfloor$-edik legkisebbet

```
1  Function median(x):
2      return select(⌊(size of x)/2⌋, x);
   // akár O(n²)-es rendezéssel is jó, max 5 eleműek a rendezendő tömbök
3  Function select(k, x):
4      if size of x<5 then
5          return kth element of x after sorting;
6      medians ← empty list;
7      for i = 0 to ⌈n/5⌉ do
8          append median of x[5i], ··· x[5i + 4] to medians; // túlindexelt elemeket ignorálva
9      pivot ← median(medians); // n/5 elemű halmazra ugyanez a mediánkeresés
10     smaller ← empty list;
11     largereq ← empty list;
12     for i = 0 to n do
13         if x[i] < pivot then
14             append x[i] to smaller;
15         else if x[i] > pivot then
16             append x[i] to larger;
17     if n−size of larger < k then
18         return select(k − (n − size of larger), larger);
19     if size of smaller > k then
20         return select(k, smaller);
21     else
22         return pivot;
```

A mediánok mediánja tranzitivitás miatt nagyobbegyenlő a kis csoportok kisebbik mediánjú feléből darabonként három elemnél, tehát az x elemei közül legalább $\frac{3n}{10} - 4$-nál nagyobbegyenlő. Továbbá szimmetria miatt legalább ugyanennyinél kisebbegynlő. Emiatt a végén legfeljebb $\frac{7n}{10} + 4$ méretű feladatra hívjuk meg rekurzívan a metódust. A futásidőre tehát teljesül a következő:

$$T(n) = \begin{cases} O(1) & \text{ha } n < 100 \\ T(\lceil \frac{n}{5} \rceil) + T(\frac{7n}{10} + 4) + O(n) & \text{ha } x \geq 100 \end{cases}$$

1.2.1. Tétel. *Az 1.1 algoritmus futásideje $T(n) = O(n)$.*

Bizonyítás. Indukcióval megmutatjuk, hogy $T(n) < cn$ valamilyen c konstansra:

$n < 100$ esetén világos, hogy ez egy elég nagy c'-re teljesül. Különben pedig az indukciós hipotézis miatt valamilyen a-ra $T(n) \leq c(\lceil \frac{n}{5} \rceil) + c(\frac{7n}{10} + 4) + an \leq c(\frac{9n}{10} + 5) + an = cn - (\frac{cn}{10} - 5c - an)$.

Ez legfeljebb cn, amennyiben $-(\frac{cn}{10} - 5c - an) \leq 0 \iff c \geq 10a\frac{n}{n-50}$.

$n \geq 100$ miatt $\frac{n}{n-50} \leq 2$, tehát $c \geq 20a$ elégséges, így $c = \max(c', 20a)$ jó választás. □

1.3. Centroid dekompozíció

Ebben a fejezetben a centorid dekompizíciót ismertetem, ami egy közismert technika fagráfokkal kapcsolatos algoritmusoknál. Ezt később 3.2.3-ban fogjuk használni.

Definíció. Egy $T = (V, E)$ fa **centroidjának** nevezzük egy v csúcsát, ha v-t elhagyva minden keletkező komponens mérete $\leq \frac{n}{2}$.

1.3.1. Állítás. *Minden fában van centroid.*

Bizonyítás. Keressünk centroidot a következő algoritmussal:

1. Válasszuk ki a fának egy tetszőleges v csúcsát
2. Amennyiben v-t elhagyva minden komponens mérete $\leq \frac{n}{2}$ centroidot találtunk
3. Különben létezik pontosan 1 komponens, aminek mérete $> \frac{n}{2}$, ennek a komponensnek legyen a v-vel szomszédos csúcsa az új v és térjünk vissza a 2. ponthoz.

Ha az algoritmus nem ér véget, akkor legyen v_2 az első csúcs, ami másodszorra tölti be v szerepét, v_1 pedig az a csúcs, aki közvetlenül v_2 előtt volt v.

Amikor először voltunk v_2-ben, v_1-be mentünk tovább, mert különben v_1-be érkezés előtt elérnénk másodszorra v_2-be. Így:

- v_2-t elhagyva a v_1-et tartalmazó K_1 komponensre $|K_1| > \frac{n}{2}$
- v_1-et elhagyva v_2-t tartalmazó K_2 komponensre $|K_2| > \frac{n}{2}$

Ez a két komponens viszont diszjunkt, ami ellentmondás, tehát az algoritmus véget ér csúcsismétlés nélkül. □

Megjegyzés. Belátható: egy fában legfeljebb 2 centroid van, és ha kettő van, szomszédosak.

Az előbbi bizonyításbeli eljárás egy kis gyakorlati ügyeskedéssel ad egy $O(n)$ futásidejű algoritmust ami megtalálja a centroidot:

Válasszunk egy tetszőleges v gyökeret a fánknak, majd mélységi bejárással számoljuk meg minden u csúcsra a c_u számot: ha a szülőjét elhagynánk hány csúcs lenne a komponensében (leszármazottjainak száma +1).

Innentől amikor v-t váltunk v_0-ról v_1-re $c_{v_0} \leftarrow 1 + \sum_{i=1}^{k} c_{w_i}$ ahol v_0 szomszédjai $v_1, w_1, w_2, \cdots, w_k$.

$$\sim * \sim$$

Centroid dekompozíciónál a fentieket alkalmazva megkeressük az T fa egy c centroidját, majd az ezt elhagyva keletkező minden komponens gyökerét és centroidját rendezett párként adjuk hozzá a c-hez rendelt listához. Mivel minden lépésben a keletkező fák legfeljebb feleakkorák a rekurzió mélysége $O(\log n)$, így a centroid dekompozíció számolható $O(n \log n)$ időben.

Ez gyakran hasznos, ha olyan $\mathcal{T}$ tulajdonságára vagyunk kíváncsiak egy fának, ami gyorsan számítható az egy csúcsot elhagyva keletkező komponensek $\mathcal{T}$ tulajdonságából, hiszen így a gyökér azonosítja a részfát ha minden csúcsban eltároljuk gyerekeinek listáját. Erre 3.2.3-ben látunk majd alkalmazást.

1.4. Lineáris függvények minimumának kiszámítása

Alább a Megiddo által [Meg79] appendixében bemutatott módszert ismertetjük, amivel $O(n)$ időben kiszámítható n lineáris függvény pontonkénti minimuma a következő feladatban leírt formában. Ezt a 3.5 fejezetben fogjuk használni.

1.1. Feladat. *Adottak $f_1(t), \cdots, f_n(t)$ lineáris függvények: $f_i(t) = a_i t + b_i$.*
Számítsuk ki ezek $g(t) := \min_{i \in \{1,\cdots,n\}} f_i(t)$ pontonkénti minimumfüggvényének $-\infty = t_0 < t_1 < \cdots < t_{j+1} = \infty$ töréspontjait, illetve $k_0, \cdots, k_j$-t, hogy $g(t) = f_{k_i}(t)(\forall t \in [t_i, t_{i+1}])(\forall i \in \{1, \cdots, j\})$.

A megoldás kulcsa, hogy g meredeksége minden töréspontnál csökken. Csökkenő sorrendbe rendezzük az (a_i, b_i) párokat lexikografikusan, azaz $(a_i, b_i) < (a_j, b_j) \iff a_i < a_j$ vagy $a_i = a_j$ és $b_i < b_j$.

Ezután az $f_1, \cdots, f_l$-re vonatkozó t_i és k_i értékekből ki tudjuk számítani az $f_1, \cdots, f_{l+1}$-re vonatkozókat – az „Introduction to Algorithms Third Edition" [Cor+09]§VII.33.3-ban is megtalálható Graham's scan algoritmusra emlékeztető módon – az előző részmegoldást módosítva.

Algorithm 1.2: Lineáris függvények minimuma

```
   input  : (a_1, b_1), ··· , (a_n, b_n)
   output: k, t tömbök
 1 s ← permutation of original indices in [(a_1, b_1), ··· , (a_n, b_n)] sorted ; // a fenti rendezés
     szerint
 2 t[0]← −∞;
 3 k[0]← s[0];
 4 l ← 0; // az utolsó töréspont indexe
 5 for j = s[1] to s[n] do
 6     notfound ← true;
 7     while l > 0 and notfound do
 8         mp ← (b_j − b_k[l]) / (a_k[l] − a_j); // a következő lin fv metszéspontja az utolsóval a
             kiszámított minimumban
 9         if mp ≤ t[l] then
10             delete t[l] and k[l];
11             l ← l − 1;
12         else
13             t[l+1]← mp;
14             k[l+1]← j;
15             l ← l + 1;
16             notfound ← false;
17     if l = 0 then
18         if a_l = a_k[0] then
19             k[0]← l;
20         else
21             t[1] ← (b_j − b_k[l]) / (a_k[l] − a_j);
22             k[1] ← l;
23 return k,t;
```

$O(n \log n)$ a rendezéshez szükséges idő. Utána minden egyenes legfeljebb egyszer kerül be a minimum szakaszai közé és legfeljebb egyszer kerül ki, tehát $O(n)$ műveletet végzün. Ez összesen $O(n \log n)$.

1.4.1. Tétel. *1.2 algoritmus futásideje* $T(n) = O(n \log n)$.

2. fejezet

Párhuzamos algoritmusok

Ebben a fejezetben a párhuzamos algoritmusokat fogjuk tárgyalni. Ezek önmagukban is érdekesek és hasznosak a GPU-k és szuperszámítógépek miatt, viszont ebben a dolgozatban a következő fejezetben tárgyalt paraméteres keresés miatt vezetjük be. Paraméteres keresésnél szekvenciálisan akarunk futtatni párhuzamos algoritmust, így az itt tárgyalt PRAM modell megfelel céljainkra.

A fejezet JáJá „An Introduction to Parallel Algorithms" [JáJ92] című könyvének részeit dolgozza fel, kivéve a megjelölt helyeket.

2.1. Általánosan a párhuzamos algoritmusokról

Párhuzamos számítógépek modellezésére szakirodalomban sok különböző modell előfordul, hiszen itt a processzorarchitektúrák is kevésbé szabványosak. Itt nem adódik egyértelműen természetes és kényelmes model, ami lényegében ekvivalens minden másikkal, mint a RAM szekvenciális számítások esetén.

Ebben a dolgozatban a szinkronizált PRAM (Parallel Random Access Machine) architektúrát fogjuk használni, hiszen az ezekre írt kód nagyon hasonló a RAM programokhoz, így használhatjuk az ottani megszokott eszköztárat kevésbé fókuszálva a technikai részletekre.

Ebben a modellben adott p processzor, viszont darabszámuk a bemenet méretének is lehet függvénye. Minden processzor rendelkezik saját memóriával, amit úgy tud használni, mint egy RAM gép, és a processzorok az i-edik (konstansidejű) programsort ugyanabban az időpontban végzik el párhuzamosan. Ezenkívül van egy globális memória, amiből mindegyik processzor tud olvasni, és tud bele írni.

A globális memóriából történő adatlekérdezésre (olvasásra) és abba történő írásra helyezett megkötések szerint szokás megkülönböztetni még az alábbi 5 típusát PRAM-nak:

- EREW (Exclusive Read Exclusive Write) Itt a globális memória adott memóriacíméhez egy időegységben csak egy processzor férhet hozzá.
- CREW (Concurrent Read Exclusive Write) Itt a globális memória egy memóriacíméről egyszerre tetszőlegesen sok processzor olvashat, viszont egy időegységben csak egy processzor írhat egy globális memóriacímre.
- CRCW (Concurrent Read Concurrent Write) Ebben az esetben egyszerre olvashat be több processzor ugyanarról a globális memóriacímről, illetve egyszerre írhatnak is ugyanarra a memóriacímre. Ezen belül az ugyanarra a memóriacímre történő egyszerre írás kezelésének megoldása sem egységes:
  - közönséges CRCW: Csak akkor írhatnak processzorok ugyanarra a memóriacímre, ha ugyanazt írják
  - véletlenszerű CRCW: Ha egyszerre írnak ugyanoda processzorok, akkor egy azok közül véletlenszerűen választott fog érvényesülni.
  - prioritás CRCW: Minden processzornak van egy egyedi azonosítószáma, és egyszerre történő írás esetén a legkisebb azonosítószámú fog írni.

Megjegyzés. Az ERCW (Exclusive Read Concurrent Write) modellel nem szokás foglalkozni, hiszen ha párhuzamosan tudunk, nem realisztikus megkötés hogy olvasni nem lehet.

Világos, hogy ezek fentről lefele haladva egyre megengedőbb modellek, viszont be fogjuk látni 2.6-ben, hogy a legmegengedőbb prioritás CRCW modellre való $O(T)$ futásidejű $O(p)$ processzort használó algoritmus szimulálható $O(T \log P)$ futásidővel EREW-en $O(p)$ processzorral. A következő két állításból következik, hogy ez éles is. Ezek bizonyításai JáJá [JáJ92] könyvének megjelölt fejezeteiben szerepelnek.

2.1.1. Állítás. *([JáJ92]§10.2.5) CREW modellben $\Omega(n)$ idő $x_1, \cdots, x_n$ bináris változóra $\bigvee_{i=0}^{n} x_i$ kiszámítása processzorszámtól függetlenül.*

Megjegyzés. Ez EREW-re is teljesül, hiszen gyengébb modell.

2.1.2. Állítás. *([JáJ92]§10.3.3) EREW modellben p processzorral egy n hosszú monoton bináris sorozatban a nullák számának meghatározásához $\Omega(\log n - \log p)$ idő kell.*

Látni fogjuk, hogy ezek az eggyel megengedőbb modellben $O(1)$ időben kivitelezhetőek.

Az érdeklődő olvasó különböző CRCW változatok közötti szimulációról többet talál JáJá könyvének végén [JáJ92]§10.1-ben.

A paraméteres kereséshez (amint 2.5-ben látni fogjuk) megfelel a prioritás CRCW is, tehát feltételezhető, hogy abban a modellben dolgozunk, viszont minden algoritmusnál jelzem, hogy mi a legmegkötöttebb modell, amiben működik.

A párhuzamosság teljesen precíz kezelését a következő utasítások haszálatával lehet megtenni:

- **global read**(X, Y) a processzor a saját memóriájában levő Y lokális változóba elmenti a globális X változóbeli értéket.
- **global write**(X, Y) a processzor a saját memóriájában lévő Y változóban tároltakat írja a globális X változóba értékként.

Ezeket használva két n hosszú vektor *koordinátánkénti* összeadása a következő módon megtehető EREW-ben $O(1)$ időben n processzorral:

Algorithm 2.1: az i-edik processzor kódja vektorösszeadásnál

input : x, y egyező dimenziójú vektorok
output: z az inputvektorok összege

```
1 global read( x_i, a);
2 global read( y_i, b);
3 c ← a + b;
4 global write( z_i, c);
```

A továbbiakban viszont a tömörség kedvéért amikor globális memóriához nyúl a processzor, ezeket nem fogom kiírni. Ezzel a fenti kód erre egyszerűsödik:

Algorithm 2.2: az i-edik processzor kódja vektorösszeadásnál

input : x, y egyező dimenziójú vektorok
output: z az inputvektorok összege

```
1 z_i ← x_i + y_i;
```

Végezetül használni fogom a **parallel for** $i \in S$ **do** absztrakciót, ami jelzi, hogy az ezt követő ciklusban lévő utasítások egyszerre $|S|$ processzorral követendőek/követhetőek. A dolgozatban szereplő pszeudókód úgy lesz megírva, hogy **parallel for** helyére **for**-t írva kis alakítással helyessé tehetők, kivéve prioritás CRCW esetén. Ebben a modellben ugyanis pont az írási sorrend manipulációja a pluszeszköz, amivel egyes esetekben gyorsabb, vagy legalábbis egyszerűbben leírható gyors algoritmusokat lehet konstruálni a kötöttebb modellekhez képest. Ezzel tehát utoljára a vektorösszeadás:

Algorithm 2.3: vektorösszeadás

input : x, y n-dimenziós vektorok
output: z az inputvektorok összege

```
1 parallel for i = 0 to n do
2     z_i ← x_i + y_i;
3 return z;
```

Ez az absztrakció lehetővé teszi, hogy ilyen módon megírt p processzoros $O(T)$ futásidejű **parallel for** kódblokkot $p' < p$ processzorral futtassunk $O(T\left\lceil \frac{p}{p'} \right\rceil)$ időben azáltal, hogy a rendelkezésre álló processzorok között szétosztjuk a végrehajtandó feladatokat. Ez prioritás CRCW-vel nem működik, viszont a 2.5-ben leírt módszer erre is adaptálható. Ezért viszont valamivel népszerűbb a közönséges CRCW modell, hiszen nincs többletfutásidő abból, ha ilyen módon szimulálunk több processzort. Ez a valóságban relevánsabbá teszi a modellt, ahol bár nagyon sok processzorhoz is lehet hozzáférésünk, a feladat méretében polinom sok nem feltétlenül reális.

Párhuzamos algoritmusoknál szokás $T(n)$ **futásidő** és $P(n)$ **processzorszám** mellett $W(n)$ **összlépésszámról** is beszélni, ami az egyes processzorok által végzett műveletek összege. Ekkor világos, hogy $W \leq TP$, és ezen keresztül összehasonlíthatóbbak párhuzamos algoritmusaink szekvenciális algoritmusokkal olyan tekintetben, hogy mennyi „számítás megy kárba" ahhoz képest, hogy ha optimális szekvenciális algoritmust használtunk volna. Egy másik haszna az összlépészámnak az előbbi p' processzoros gondolatmenet általánosítása:

2.1.3. Állítás. *p' processzorral PRAM-on $T(n)$ futásidejű, $W(n)$ összlépészámú algoritmust legfeljebb $\left\lfloor \frac{W(n)}{p'} \right\rfloor + T(n)$ időben tudjuk futtatni.*

Bizonyítás. Ha az i-edik időegységben az algoritmus $W_i(n)$ számítási lépést végez, akkor azt p' processzorral $\left\lceil \frac{W_i(n)}{p'} \right\rceil$ időben tudjuk szimulálni. Ezt összegezve: $T_{p'}(n) = \sum_{i=0}^{T(n)} \left\lceil \frac{W_i(n)}{p'} \right\rceil \leq \sum_{i=0}^{T(n)} \left\lfloor \frac{W_i(n)}{p'} + 1 \right\rfloor \leq \left\lfloor \frac{W(n)}{p} \right\rfloor + T(n)$ □

Végezetül használni fogom a **parallel call** kulcsszót egyszerre végrehajtandó (jellemzően rekurzív) függvényhívásokra, avagy utasításokra, ahol így egyszerűbb jelezni.

parallel call f() **and** g();

Ez tehát azt jelenti, hogy az eljárások egymástól függetlenül végrehajthatóak és végrehajtandóak. Amikor befejeződnek, szinkronizáció miatt vár a gyorsabban végző processzor a lassabban végzőre.

Ezeket az absztrakciókat használva az ígért $O(1)$ megoldások:

Algorithm 2.4: 2.1.2 állításbeli feladat megoldása CREW-ban $O(1)$ időben

input : $x_1, \cdots, x_n$ monoton $0-1$ sorozat
output: a nullák száma

```
1  t ← 0 ; // van- e két különböző x_i
2  parallel for i = 0 to n − 1 do
3      z_i ← 0;
4      if x_i ≠ x_{i+1} then // ez legfeljebb egy processzorra teljesül
5          t ← 1;
6          if x_i = 0 then
7              r ← i;
8          else
9              r ← n − i;
10 if t = 1 then
11     return r;
12 if x_1 = 0 then
13     return n;
14 else
15     return 0;
```

Algorithm 2.5: 2.1.1 állításbeli feladat megoldása közönséges CRCW-ben $O(1)$ időben

input : $x_1, \cdots, x_n \in \{0,1\}$
output: $\bigvee_{i=1}^{n} x_i$

```
1 y ← 0;
2 parallel for i ∈ {1, ··· , n} do
3     if x_i = 1 then y ← 1;
4 return y;
```

2.1.4. Tétel. *Prioritás CRCW-ben a 2.5 algoritmusra* $T(n) = O(1),\ P(n) = O(n),\ W(n) = O(n)$.

2.2. Alapvető párhuzamos algoritmusok

Ebben a szekcióban három alapvető feladatra fogunk párhuzamos algoritmusokat látni. Ezekben szerepelnek párhuzamos algoritmusokban gyakori technikák, illetve használni fogjuk őket a paraméteres keresést tárgyaló fejezetben.

2.2.1. Összegzés

Gyakori feladat néhány szám összegének kiszámítása. A 2.6 algoritmus ezt a feladatot oldja meg. Csupán az összeadás asszociativitását használja ki, így tetszőleges $\otimes$ asszociatív műveletre is működik, így $\min(a,b)$-re is.

Algorithm 2.6: $O(\log n)$ futásidejű, $O(n)$ összlépésszámú EREW összegző algoritmus

input : $x_1, \cdots, x_n$
output: $\sum_{i=1}^{n} x_i$

```
1 Function sum(x_1, ··· , x_n):
2     if n = 1 then
3         return x_1;
4     else
5         parallel call a ←sum(x_1, ··· , x_⌊n/2⌋) and b ←sum(x_⌈n/2⌉, ··· , x_n);
6         return a + b;  // ide + helyett tetszőleges ⊗ asszociatív műveletet írva
              működik
```

2.2.1. Tétel. *EREW-ben a 2.6 algoritmusra* $T(n) = O(\log n),\ P(n) = O(n),\ W(n) = O(n)$.

Ez az algoritmus egy bináris fát épít fel, aminek az adatok a levelei, és aszerint összegzi őket, minden csúcsba a két gyerekebeli értéket írva. Ez sok párhuzamos algoritmusnál alapvető gondolat.

2.2.2. Prefix összegzés

Definíció. Az x vektornak az $\left(x_1, x_1 + x_2, \cdots, \underbrace{\sum_{j=1}^{i} x_j}_{i\text{-edik elem}}, \cdots, \sum_{j=1}^{n} x_j \right)$ vektor a **prefix összege**.

Ez gyakran hasznos, mert ezzel $O(1)$ időben számolható résztömb összege, hiszen $\sum_{i=k}^{l} x_i = p_l - p_k$.

2.2.2. Tétel. *EREW-ben a 2.7 algoritmusra* $T(n) = O(\log n),\ P(n) = O(n),\ W(n) = O(n)$.

Algorithm 2.7: $O(\log n)$ futásidejű, $O(n)$ összlépésszámú prefix összeget kiszámító EREW algoritmus

input : $x_1, \cdots, x_n$
output: $p_1, \cdots, p_n$ az x_i-k prefix összege
Function `prefixsum`$(x_1, \cdots, x_n)$**:**
  parallel for $i = 0, i < \left\lfloor \frac{n}{2} \right\rfloor$ **do**
    $t_i \leftarrow x_{2i} + x_{2i+1}$;
  $t \leftarrow$`prefixsum`$(t_1, \cdots, t_{\lfloor n/2 \rfloor})$;
  parallel for $i = 1, i \leq n$ **do**
    switch i **do**
      case $i = 1$ **do** $s_1 \leftarrow x_1$;
      case i *is even* **do** $s_i \leftarrow t_{i/2}$;
      case i *is odd* **do** $s_i \leftarrow t_{\lfloor i/2 \rfloor} + x_i$;

2.2.3. Minimumszámítás

Számok minimumának kiszámítása elemi feladat, és ez sok algoritmusnak építőköve. Ez a 2.6 összegző algoritmussal megoldható $O(\log n)$ időben, viszont a következő technikákal ennél jobb futásidő is elérhető. Ezt a jobb futásidőt 3.5-ben használni fogjuk.

Feltehetjük hogy a kapott számok különbözőek, mert a minimumkeresés előtt hozzájuk rendelhetünk tetszőleges különböző számokat, mint párok második tagjai. Ezután lexikografikusan rendezhetjük a kapott párokat. Így a második szám csak akkor dönt, ha van két azonos.

Algorithm 2.8: Konstans idejű minimumkiválasztó algoritmus közönséges CRCW-ben

input : $x_1, \cdots, x_n$ úgy, hogy nincs köztük két egyenlő
output: A minimális elem értéke
parallel for $i, j \in \{1, \cdots, n\}$ **do**
  if $x_i > x_j$ **then** $c_{i,j} \leftarrow 0$;
  else $c_{i,j} \leftarrow 1$;
parallel for $i \in \{1, \cdots, n\}$ **do**
  $isminimal_i \leftarrow \bigwedge_{j=1}^{n} c_{i,j}$;
parallel for $i \in \{1, \cdots, n\}$ **do** `// csak egyre fog teljesülni egyediség miatt`
  if $isminimal_i = 1$ **then** $ret \leftarrow x_i$;
return *ret*;

2.2.3. Tétel. *Közönséges CRCW-ben a 2.8 algoritmusra* $T(n) = O(1),\ P(n) = O(n^2),\ W(n) = O(n^2)$.

$W(n) = O(n^2)$, viszont ez segítségünkre lesz egy $O(\log \log n)$ futásidejű, $O(n)$ összlépésszámú algoritmus létrehozásában, és gyakran hasznos, ha a feladat méretéhez képest kevés objektumnak kell minimumát számítani.

Definíció. Egy n levelű fát **iteráltlogaritmus mélységűnek** nevezünk, ha gyökerének $\sqrt{n}$ gyereke van, és minden gyereke egy $\sqrt{n}$ levelű iteráltlogaritmus mélységű fa gyökere.

Ha egyszerűség kedvéért feltesszük hogy $n = 2^{2^k}$, akkor egyszerű indukcióval látszik, hogy a fa i. szintje $2^{2^k - 2^{k-i}}$ csúcsból áll, és $k + 1 = \log \log n + 1$ szintje van.

Ebből adódik a 2.9 algoritmus. Az összegzéshez hasonlóan a fa minden csúcsában a gyerekeinek minimumát a 2.8 algoritmussal számítva.

2.2.4. Tétel. *Prioritás CRCW-ben a 2.9 algoritmusra* $T(n) = O(\log \log n),\ P(n) = O(n),\ W(n) = O(n \log \log n)$.

Bizonyítás. A 2.9 algoritmus a rekurzió i-edik szintjén a generált iteráltlogaritmus mélységű fának minden csúcsában $O\left((2^{2^{k-i-1}})^2\right)$ műveletet végez, így az i-edik szinten összesen $O\left((2^{2^{k-i-1}})^2 \cdot 2^{2^k - 2^{k-i}}\right) = O(2^{2^k}) = O(n)$ műveletet végez. □

Algorithm 2.9: $O(\log\log n)$ minimumkiválasztás CRCW-ben

input : $x_1, \cdots, x_{2^{2^k}}$
output: A minimális elem értéke

```
1 Function llmin(x_1, ..., x_{2^{2^k}}):
2     parallel for i ∈ {0, ... 2^{2^{k-1}} − 1} do
3         y_i ← llmin(x_{i·2^{2^{k-1}}}, ..., x_{(i+1)·2^{2^{k-1}}−1});
4     ret ← min_{i∈{0,...2^{2^{k-1}}−1}} (y_i); // az előző O(1) algoritmussal
5     return ret;
```

Ezt tovább lehet optimalizálni azzal, hogy a levelektől $\lceil \log\log\log n \rceil$ szintig az összegzésre látott bináris fát növesztjük, és ott váltunk az iteráltlogaritmusos rekurzióra, hogy az így kapott $\frac{n}{\log\log n}$ szám minimumát kiszámoljuk. Így a futásidő $O(\log\log n)$ marad, hiszen az első bináris rekurziós fázis $O(\log\log\log n)$ futásidejű és $O(n)$ összlépésszámú, az iteráltlogaritmus mélységű fában pedig $\frac{n}{\log\log n}$ elem minimumát

$$O\left(\frac{n}{\log\log n}\log\log(\frac{n}{\log\log n})\right) \leq O\left(\frac{n}{\log\log n}\log\log n\right) = O(n)$$

összlépésszámmal számoljuk ki $O(\log\log n)$ időben. Ezzel az összlépésszámot lecsökkentettük $O(n)$-re, és a futásidő $O(\log\log n)$ marad.

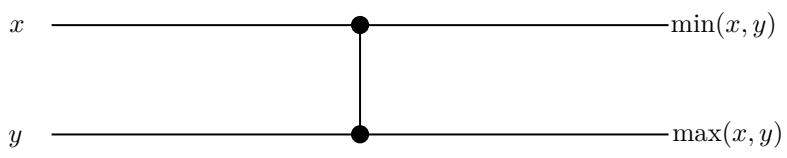
2.2.5. Tétel. *Prioritás CRCW-ben $T(n) = O(\log\log n)$, $P(n) = n$, $W(n) = n$-nel kiszámítható n szám minimuma.*

2.3. Rendező hálózatok

Ebben a szekcióban az „Introduction to Algorithms Second Edition” [Cor+01]§27 alapján mutatom be a rendező hálózatokat. A rendezési hálzatok működési módjának részleteit 3.6-ban fogjuk kihasználni, hogy gyorsabb algoritmust adjunk bizonyos típusú paraméteres keresési feladatokra.

A rendező hálózatok egy, az EREW-nél is erősebben megkötött számítási modell, ami így könnyen szimulálható tetszőleges PRAM-on. Ennek ellenére ebben a modellben is lehet $O(n)$ processzorral $O(\log n)$ időben rendezni. Ebben a fejezetben bemutatom az alapvető működésüket, és egy $O(\log^2 n)$ időben futó $O(n)$ processzoros rendezőalgoritmust fogunk látni rajtuk.

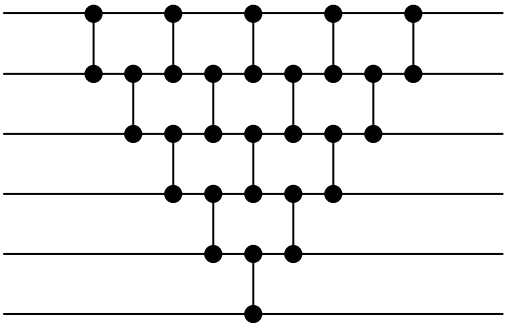
Definíció. Egy **hálózat** n darab vezetékből és F fázisból áll. A vezetékeket fentről lefelé számozzuk $1, \cdot, n$-el, és bal oldalukra kerülnek rá a bemenetként kapott számok. A hálózat minden fázisban összehasonlításokat végez: minden vezeték legfeljebb egy összehasonlításban szerepelhet, és minden összehasonlításban két vezeték szerepel. Amikor két vezetéket összehasonlít a háló, akkor az összehasonlítás után az alsó vezetékre kerül a két vezetéken levő szám maximuma, a felsőre pedig a minimumuk. Így a nagy számok „alulra süllyednek”.



Akkor nevezzünk egy hálózatot **rendező hálózatnak**, ha tetszőleges számokat a háló elejére a vezetékekre téve az utolsó fázis lefutása után a vezetékeken levő kimenet fentről lefelé nő.

Megjegyzés. Mivel a rendező hálózat csak összehasonlításokat végez, a vezetékekre tetszőleges algoritmikusan eldönthető rendezéssel bíró objektumokat tehetünk, viszont az egyszerűség kedvéért ebben a szekcióban feltesszük hogy számok.

Az alábbi ábrán látható a beszúrásos rendezés által megadott rendező hálózat, ami egybeesik a buborékos rendezés által meghatározottal.



Megjegyzés. Ez a a háló $O(n)$ mélységű, és $O(n)$ processzort használ $O(n^2)$ lépéssel.

2.3.1. Állítás. *Ha $f : \mathbb{R} \to \mathbb{R}$ monoton növekvő függvény, $\mathcal{H}$ háló kimenete I és bemenete $\mathcal{O}_I$, akkor $\mathcal{O}_{f(I)} = f(\mathcal{O}_I)$, ahol f-et koordinátánként alkalmazzuk.*

Bizonyítás. Indukcióval szint szerint: k-adik szintig ugyanazt kapjuk ha f-et futtatás előtt alkalmazzuk I-re, mint a k-adik szint után.

f monotonitása miatt $f(\max(x,y)) = \max(f(x), f(y))$ és $f(\min(x,y)) = \min(f(x), f(y))$, tehát tetszőleges összehasonlításnál mindegy hogy f-et előtte vagy utána alkalmazzuk, tehát egy egész szintet futtatva is az. □

2.3.2. Lemma. *Egy hálózat rendező hálózat $\iff$ minden $I \in \{0,1\}^n$ bemenetet jól rendez.*

Bizonyítás. $\implies$ irány triviális: ha minden bemenetet rendez, a $0-1$ sorozatokat is.

$\impliedby$ irányhoz azt látjuk be, hogy ha egy hálózat valamilyen bemenetet rosszul rendez, akkor van $0-1$ sorozat is, amit rosszul rendez:

Ha x inputot nem rendez a háló, a bemenetnél i. vezetéken v_i van, és ez a szám a kimenetnél a $\pi(i)$-edik vezetéken van, akkor $\exists v_i < v_j : \pi(i) > \pi(j)$. Az előző állítás szerint $f(x) = \begin{cases} 0 & \text{ha } x \le v_i \\ 1 & \text{ha } x > v_i \end{cases}$ függvénnyel a $u_i = f(v_i)$ bemenetet is rosszul rendezi, hiszen $u_i < u_j$ és $\pi(i) > \pi(j)$ továbbra is teljesül. □

2.3.1. Odd-Even mergesort

A Lang által [Lan18]-ban leírtakat feldolgozva ebben az alábbiakban $O(\log^2 n)$ fázisú rendező hálózatot fogunk látni, aminek helyességét a 2.3.2 lemma felhasználásával be fogjuk látni.

A rekurzív konstrukció miatt feltesszük, hogy 2^n sok számot akarunk rendezni. Ez aszimptotikusan nem ront a futásidőn, hiszen ha $2^{n-1} < k < 2^n$, akkor $(\lceil \log k \rceil)^2 = n^2 < (\log 2k)^2 = (\log k + 1)^2 = \log^2 k + 2\log k + 1$, és $k > 8$-ra $\log^2 k > 2\log k + 1$, így $\log^2 k + 2\log k + 1 < 2\log^2 k \implies n < 2\log^2 k$.

A következő eljárással, ha $x_0, \cdots x_{n/2-1}$ és $x_{n/2}, \cdots, x_{n-1}$ két monoton növő sorozat, össze tudjuk őket fésülni:

Algorithm 2.10: merge$(x_0, \cdots, x_{n-1})$ ahol x hossza $n > 1$ kettőhatvány

```
1 if n>2 then
2 |  parallel call merge(x_0, x_2, ..., x_{n-2}) and merge(x_1, x_3, ..., x_{n-1});
3 |  parallel for i ∈ {1, 3, ..., n-3} do
4 |  |  compare(x_i, x_{i+1}); // ez összehasonlít és cserél ha szükséges
5 else
6 |  compare(x_0, x_1);
```

2.3.3. Tétel. *EREW-ben a 2.10 algoritmus összefésül $T(n) = O(\log n)$, $P(n) = O(n)$, $W(n) = O(n \log n)$.*

Bizonyítás. Indukcióval látjuk be, és az előbbi lemma miatt elég ezt 0-1 sorozatokra megtennünk.

2 hosszú bemenetre világos, hogy összefésül, hiszen összehasonlítja őket, és cserél ha szükséges.

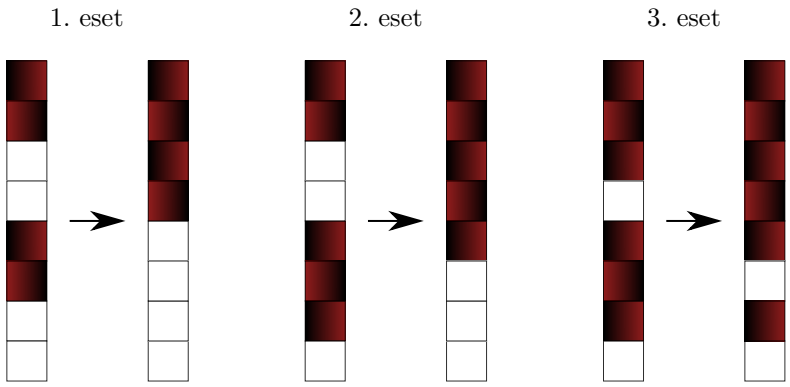
Indukció szerint tudjuk, hogy $x_0, \cdots, x_{n/2-1}$ és $x_{n/2}, \cdots, x_{n-1}$ rendezve vannak, így ezeknek páros indexű $x_0, x_2, \cdots, x_{n/2-2}$ és $x_{n/2}, x_{n/2+2}, \cdots, x_{n-2}$, illetve páratlan indexű $x_1, x_3, \cdots, x_{n/2-1}$ és $x_{n/2+1}, x_{n/2+3}, \cdots, x_{n-1}$ részsorozai is külön-külön rendezettek, tehát ezeket a párokat rekurzívan összefésülhetjük. Ekkor $x_0, x_2, \cdots, x_{n-2}$ és $x_1, x_3, \cdots, x_{n-1}$ rendezettek lesznek.

Egy $x_0, \cdots, x_{m-1}$ rendezett sorozatnak a páros és páratlan indexű részsorozataiban vagy ugyanannyi 0 van, vagy a páros részsorozatban van eggyel több, és ez az 0-k számának paritásától függ.

Így a páratlan részsorozatban 0-val, 1-gyel, vagy 2-vel kevesebb 1-es szerepel a párosnál.

Innen 3 lehetőség van:

1. A páros és páratlan rendezett részsorozatban ugyanannyi 0 szerepel: ekkor az egész is rendezett.
2. A páratlanban eggyel kevesebb van: ekkor is rendezett az egész, hiszen az utolsó páratlan indexű egyes után már csak egy páros sorszámú szerepel.
3. A páratlanban 2-vel kevesebb van: ekkor nem rendezett még az egész, de a szomszédos elemek összehasonlítása és cseréje megjavítja, hiszen egy rés van, ahol páratlan indexben 0 szerepel az őt követő páros indexben lévő 1 előtt.



Ezzel tehát láttuk, hogy összefésül.

A rekurzió mélysége $O(\log n)$, hiszen minden rekurzív hívásnál feleződik a feladat mérete, illetve utána a cserélés konstans időben történik, így a futásidő $O(\log n)$. □

Innen világos, hogy az alábbi algoritmus rendez, hiszen a rekurzió megegyezik a mergesort-éval:

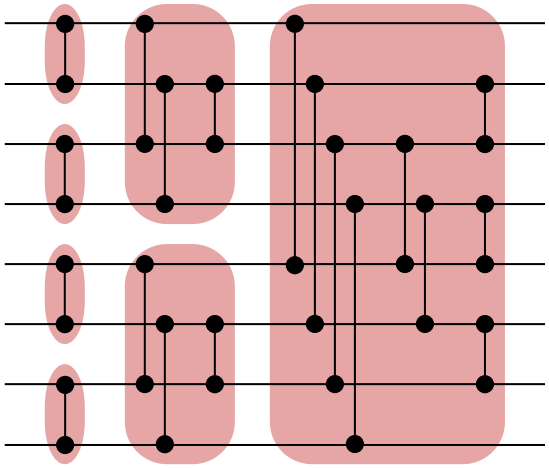
Algorithm 2.11: oesort($x_0, \cdots, x_{n-1}$) ahol x hossza $n > 1$ kettőhatvány

```
1 if n>2 then
2 |   parallel call oesort(x_0, ..., x_{n/2-1}) and oesort(x_{n/2}, ..., x_{n-1});
3 |   merge(x_0, ..., x_{n-1});
4 else
5 |_  compare(x_0, x_1);
```

2.3.4. Tétel. *EREW-ben a 2.11 algoritmusra* $T(n) = O(\log^2 n),\ P(n) = O(n),\ W(n) = O(n \log^2 n)$.

Bizonyítás. Minden rekurzív hívásban feleződik a feladat mérete, tehát $O(\log n)$ rekurzív hívás történik. Minden m méretű rekurzív híváson belül pedig a merge futásideje $O(\log m) \leq O(\log n)$, így a futásidő $O(\log^2 n)$. □

2.1. ábra. merge-ekre bontva a hálózatot $n = 8$-ra:



Megjegyzés. Ajtai, Komlós és Szemerédi [AKS83]-ben megkonstruálják az AKS rendező hálót, ami $O(\log n)$ mélységével optimális, így későbbiekben azt használom futásidőszámításoknál, mert jobb eredményeket ad. Fontos említeni viszont, hogy az expandergráfos konstrukció miatt a futásidőnek nagy konstansszorzója van, így a 2.11 algoritmus $O(\log^2 n)$-es futásideje minden gyakorlati feladatra gyorsabb.

Továbbá léteznek EREW-ben $O(\log n)$ algoritmusok, mint a párhuzamos „pipelined mergesort", ami JáJá kifejt [JáJ92]§4.3-ban szerepel. Emiatt nem szükséges abban a modellben sem az AKS nagy konstansaival élni rendezés esetén.

Bizonyítható, hogy az alábbi Bekbolatovtól származó [Bek15]-ön alapuló iteratív implementáció ekvivalens a rekurzívval. `pair`(i, f, m) kiszámítja az i-edik vezeték párját az f-edik külső algoritmusbeli fázis m-edik összefésülési lépésében.

Algorithm 2.12: iteratív verzió

```
   input : x_0, ··· x_{2^n−1}
   // ha az i-edik vezeték nem szerepel összehasonlításban, önmagának lesz a
      párja
 1 Function pair(i,f,m):
 2     if p = 1 then
 3         return n ⊕ 2^{f−1};
 4     else
 5         scale ← 2^{f−m};
 6         box ← 2^m;
 7         div ← ⌊n/scale⌋;
 8         sn ← div − ⌊div/box⌋ · box;
 9         if sn = 0 or sn = box − 1 then
10             return n;
11         else if sn is even then
12             return n − scale;
13         else
14             return n + scale;
15 for f=1 to n do
16     for m=1 to f do
17         parallel for i=0 to 2^n − 1 do
18             compare(x_i,pair(x_i, f, m)); // figyelve, hogy minden összehasonlítás
                egyszer történjen csak meg
```

2.4. Boruvka/Sollin algoritmusa minimális költségű feszítőfára

Az alábbi szekcióban minimális költségű feszítőfa kiválasztására fogunk hatékony párhuzamos algoritmust látni. A következő állítások JáJá könyvéből származnak ([JáJ92]§5.2), viszont a pszeudókód Wu [Wu90] cikke alapján készült. Ezt az algoritmust paraméteres kereséssel együtt fogjuk használni 3.4.2-ban törtlineáris optimalizálási feladat megoldására.

2.1. Feladat. *Adott egy $G(V, E)$ gráf éllista formájában és egy $s : E \to \mathbb{R}$ súlyfüggvény. Keressük meg a gráf legolcsóbb feszítőfáját.*

Innentől feltesszük, hogy bármely két él költsége különböző, hiszen a minimumkiválasztásnál látott trükkel élhetünk: minden élhez hozzárendelünk egy második egyedi azonosítószámot, és amennyiben a súlyuk ugyanannyi, a kisebb azonosítójút nevezzük kisebbnek.

2.4.1. Állítás. *Az F minimális költségű feszítőfa tartalmazza a v csúcsból kiinduló legolcsóbb $e = (u, v)$ élt.*

Bizonyítás. Ha F nem tartalmazná e-t, akkor a feszítőfabeli u-ból v-be menő út legutolsó élét lecserélve e-re olcsóbb feszítőfát kapnánk, hiszen e olcsóbb ennél az élnél. □

2.4.2. Állítás. *Legyen $V = \bigcup V_i$ tetszőleges partíciója a $G(V, E)$ gráf csúcsainak. Ekkor $\forall i$-re a V_i-ből kimenő legolcsóbb $e_i = (u_i, v_i)$ $(u_i \notin V_i, v_i \in V_i)$ él benne van az F minimális feszítőfában.*

Bizonyítás. A fentiekkel analóg módon: ha F nem tartalmazná e_i-t, akkor az u_i-ből a V_i komponensbe vezető fabeli út utolsó éle helyett e_i-t hozzávéve F-hez olcsóbb feszítőfát kapnánk. □

Ebből a következő stratégiát kapjuk: kezdetben minden csúcsot külön komponensbe teszünk. Minden komponensre a belőle kimenő legolcsóbb élt hozzávesszük a feszítőfához, és ezt ismételjük amíg már csak egy komponens van. Minden fázisban a komponenseknek legalább a fele megszűnik, így $\log n$ fázisban végetér az eljárás.

Ez $O(\log n)$ futásidővel a 2.13 algoritmusban látható módon meg is valósítható prioritás CRCW modellben $O(m)$ processzorral. Feltesszük, hogy a gráf csúcsait $1, \cdots, n$ számokkal azonosítjuk, és $e_i = (u, v)$ él az ezeket tartalmazó számpár és súlya által van reprezentálva.

A komponensből kimenő minimális él a 2.8 algoritmushoz hasonló módon történik. Minden él a súlyának megfelelő prioritással a két csúcsának komponenséhez odaírja sorszámát. Arra figyelünk kell, hogy csak akkor vegyünk be élt, ha valamelyik végén lévő komponens a hivatkozásfája csillag (ez az algoritmusban bővebben ki van fejtve), hogy ne vegyünk be komponensen belül menő éleket.

Algorithm 2.13: Boruvka/Sollin algoritmusa

input : $e_1, \cdots, e_m$ a gráf élei, $s_1, \cdots, s_m$ ezek súlyai
output: selected[1], $\cdots$, selected[m], amik 1, ha az adott él ki van választva, 0 különben

```
parallel for i ∈ {1,···,n} do
    component[i]← i; // a komponenst egy benne lévő csúcs azonosítja, viszont a
     component hivatkozásláncot végigkövetve kapjuk csak meg az igazi
     azonosítót
Function isstar(i):
    // ezt párhuzamosan futtatva minden csúcsra eldől, hogy a komonensének
       hivatkozásstruktúrája csillag-e, ahol csillagnak nevezzük azokat a
       komponenseket, ahol minden csúcs ugyanazt gondolja a
       komponensazonosítónak
    st[i]← 1;
    if component[i]≠component[component[i]] then
        st[i]← 0;
        st[component[component[i]]]← 0;
    if st[component[i]]= 0 then st[i]← 0;
e_1,···,e_m ← e_1,···,e_m sorted; // Ez O(log|E|) ≤ O(log|V|^2) = O(log|V|) idő
while ∃j :component[j] changed last iteration do
    parallel for i ∈ {1,···,n} do
        totake[i]← none; // ebben a lépésben az i indexű csúcsnek melyik élét
         vesszük be
    parallel for i ∈ {1,···,n} do
        // frissítjük, hogy melyik komponensek csillagok
        isstar(i);
    parallel for i ∈ {1,···,m} with i being the priority of the executing processor do
        u_i, v_i ← vertices e_i connects; // minden élre 2 processzorral, hogy mindkét
         rendezése fedve legyen a két csúcsnak
        if st[v_i]= 1 and component[u_i] ≠ component[v_i] then
            component[component[v_i]]←component[u_i];
            totake[component[v_i]]← e_i;
            if totake[component[v_i]]= e_i then selected[e_i]← 1;
            // Mindig csak csillagot csatolunk hozzá másik komponenshez, és
               itt garantáljuk, hogy egy csillagot egyszerre csak egy
               komponenshez csatoljunk hozzá
    parallel for i ∈ {1,···,n} do
        if i ≤component[i] and component[component[i]]= i then
            component[i]← i;
            // a hivatkozásciklusok megszüntetéséhez
    parallel for i ∈ {1,···,n} do
        // minden csúcs komponensazonosítóját az azonosítócsúcsának
           komponensazonosítójára állítunk, hogy a komponenesek
           hivatkozásmélysége csökkenjen
        component[i]←component[component[i]];
```

Kör nem keletkezhet, hiszen akkor a komponenseket csúcsokká összehúzva ez a kör csupa új élből állna, amikre a rendezésben ez adna egy kört: $s_{\pi(1)} < s\pi(2) < \cdots < s_{\pi(k)} < s_{\pi(1)}$, viszont ez nem lehetséges. (itt használjuk, hogy az egyenlőtlenségek élesek, mivel a súlyok között nincs két azonos)

Ennek az algoritmusnak a futásideje $O(\log n)$, hiszen ha a különböző komponensekre összegezzük a hivatkozásfa mélységét, akkor az a hozzácsatolásokkal nem nő, viszont a végén a mélységcsökkentő lépésnél komponensenként legfeljebb $\frac{2}{3}$-a lesz az eddiginek, és amikor ez a mérőszám 1, végetért az algoritmus futása.

2.4.3. Tétel. *Prioritás CRCW-ben a 2.13 algoritmusra* $T(n) = O(\log n)$, $P(n) = O(m)$, $W(n) = O(m \log n)$.

Megjegyzés. [CHL01]-ben Chong, Han és Lam mutatnak egy $T(n) = O(\log n)$, $P(n) = O(m)$, $W(n) = O(m \log n)$ algoritmust EREW modellben.

2.5. Prioritás CRCW szimulálása egy processzorral

A következő tétel hasznos lesz, amikor prioritás CRCW-beli algoritmusokat használunk paraméteres keresésnél a következő fejezetben.

2.5.1. Tétel. *Prioritás CRCW algoritmus szimulálható* 1 *processzorral* $O(W(n))$ *időben.*

Minden változóhoz tartsunk számon egy listát, amibe minden lépésben beleírjuk az oda írni próbáló processzorok sorszámát, és az azáltal a processzor által odaszánt számot.

Ezentúl egy másik l listában tartsuk számon azokat a mezőket, ahova ebben a lépésben írni próbáltunk. Minden párhuzamos lépés szimulálásánál eleinte csak ezekbe a listákba „írunk" a processzorokkal, majd miután ez megtörtént minden processzorral végigmegyünk az összes változón, ahova írni próbáltunk l-en.

Minden változóhoz kiválasztjuk a változó listájából a minimális azonosítószámú processzort, és annak az írását végrehajtjuk. Miután ezt megtettük, töröljük a változó listáját, illetve a változót l-ből. Ha P processzort szimulálunk, ez $O(P)$ időben megtehető legfeljebb $O(P)$ többlet tárhely használatával, tehát futásidőbeli romlást az összlépésszámhoz képest nem eredményez.

2.6. Prioritás CRCW szimulálása EREW-ben

A következő szekcióban JáJá könyvéből [JáJ92]§10.1.1-t feldolgozva megmutatjuk, hogy prioritás CRCW szimulálható EREW-ben $O(\log p)$ szorzóval, ahogyan a fejezet elején említettük.

Legyenek $Q_1, \cdots, Q_p$ a prioritás CRCW algoritmus szimulálandó processzorai. Ezek szimulálására fogjuk használni a $P_1, \cdots, P$ EREW processzorainkat. Ezenkívül globális memóriában a szimulációra fenntartjuk az $M_1, \cdots, M_p$ memóriaegységeket. Az $O(\log p)$ szorzójú szimulációhoz elég a párhuzamos olvasás műveletet és párhuzamos írás műveletet $O(\log p)$ időben szimulálni. Tegyük fel, hogy minden olvasandó adat egy egész szám.

Olvasás

Ha a Q_i processzor a j memóriacímet szeretné olvasni, akkor $M_i \leftarrow (j, i)$. Ezután az M_i-kbe írt számpárokat rendezzük lexikografikusan $O(\log p)$ időben. Ekkor P_i, amennyiben $i = 1$ vagy az M_{i-1}-beli pár első száma különbözik M_i első számától, (azaz máshonnan akarnának olvasni az előző M_i-beli „olvasásparancs" szerint) akkor R_i-be írja a j memóriacímen talált számot, különben 0-t ír oda.

Ezután az adatok szétosztásához egy szegmensenkénti prefix összegző algoritmust futtatunk. Ennek szegmensei az olyan R_i blokkok, ahol az M_i-k első számai megegyeznek. (azaz ugyanahhoz a memóriacímhez szerettek volna hozzáférni az processzorok) Ez $O(\log p)$ időben „szétosztja" a beolvasott adatokat.

Utolsósorban $M_i \leftarrow (M_i, R_i)$, majd ezeket rendezzük az M_i-k második koordinátái szerint (azaz az olvasni akaró processzor indexe szerint) $O(\log p)$ időben. Ekkor P_i megkapja a Q_i által globális memóriából beolvasandó számot, és folytathatja Q_i szimulálását.

Algorithm 2.14: Egy olvasási lépés, ahol Q_i a j_i memóriacímről akar olvasni

```
1  parallel for i ∈ {1, ··· , p} do
2      M[i] ← (i, j_i);
3  sort M lexicographically;
4  parallel for i ∈ {1, ··· , p} do
5      (a_i, b_i) ← M[i];
6      if i = 1 or a_i ≠ a_{i−1} then
7          global read( globalmemory[j],r_i);
8      M[i] ← (b_i, r_i);
9  compute segmented prefix sums of M;
10 sort M lexicographically;
11 parallel for i ∈ {1, ··· , p} do
12     (a_i, r_i) ←M[i];
       // itt a Q_i által beolvasni vágyott információ r_i
```

Írás

Az előzőhöz hasonló gondolatokkal megoldható ez is:

Algorithm 2.15: Egy írási lépés, ahol Q_i a j_i memóriacímre próbál írni w_i-t

```
1  parallel for i ∈ {1, ··· , p} do
2      M[i] ← (j_i, i, w_i);
3  sort M lexicographically;
4  parallel for i ∈ {1, ··· , p} do
5      (a_i, b_i, c_i) ←M[i];
6      if i = 1 or a_i ≠ a_{i−1} then
7          global write( globalmemory[a_i], c_i);
```

Megjegyzés. A szegmensenkénti prefix szumma számítás megoldható a prefix szumma iteratív implementációjához hasonló módon, figyelve arra, hogy szegmenshatárokat ne lépjük át az összegzésnél.

3. fejezet

Paraméteres keresés

Ebben a fejezetben a paraméteres keresést fogjuk bemutatni példákon keresztül. Bevezető egyszerű alkalmazás után bemutatom a Smidtől [Smi02] származó keretrendszert. A keretrendszert felhasználva látni fogunk egy geometriai és két kombinatorikai felhasználást a módszerre. Ezt követően az általánosabb keretrendszer egy speciális alesetére fogunk látni példákat, és szó lesz törtlineáris kombinatorikai optimalizálási feladatok megoldásáról is. A fejezet végén pedig látni fogjuk a Radzik által [Rad98]-ben formalizált, már Megiddo által is használt gyorsításokat minimumkiválasztásra és a Cole [Col87] cikkében szereplő optimalizációt is, ami az AKS rendező hálót felhasználva ér el jobb futásidőket.

3.1. A technika bevezetése

A következő Megiddotól [Meg83a]§2-ból származó egyszerű példán vezetjük be a paraméteres keresést. A feladat demonstrációs célokat szolgál, hiszen egyszerűsége miatt jól szemléltethetőek rajta a Megiddo technikájához szükséges gondolatok.

3.1. Feladat. *Adottak páratlan $n \in \mathbb{N}$-re $f_1(x) = a_1x + b_1, \cdots, f_n(x) = a_nx + b_n$ lineáris függvények úgy, hogy $a_i < 0\ \forall i \in \{1, \cdots, n\}$. Legyen $\alpha(x) := \text{median}\{f_1(x), \cdots, f_n(x)\}$. Keressük λ^*-ot, hogy $\alpha(\lambda^*) = 0$.*

Megjegyzés. Teljesség kedvéért megjegyezzük, hogy ez megoldható $O(n)$ időben is az alább látottaknál egyszerűbben, hiszen λ^* a mediánja az f_i-k $-\frac{a_i}{b_i}$ gyökeinek.

Vegyük észre, hogy mivel $\forall i a_i < 0$ α monoton csökken, hiszen az összes f_i is, és szakaszonként valamelyik f értékét veszi fel, és a váltás akkor történik ha az éppen mediánt felvevő f_i-t keresztezi egy másik.

Adott x értékre ki tudjuk számítani $\alpha(x)$ értékét $O(n)$ időben úgy, hogy minden i-re kiértékeljük $f_i(x)$-et, és ezeknek $O(n)$ időben kiszámítjuk a mediánját.

Legyen $x_{ij} := \frac{b_j - b_i}{a_i - a_j}$ f_i és f_j metszéspontja. A feladatot $O(n^2 \log n)$ időben meg tudjuk oldani úgy, hogy kiszámoljuk az összes x_{ij}-t, rendezzük őket, majd bináris kereséssel megkeressük a nullhelyet határoló két metszéspontot $O(\log n)$-szer $O(n)$ idő alatt kiértékelve $\alpha(x)$-et, hiszen α csak metszéspontnál válthat másik f_i-re.

Ezután, ha nem esik egybe a két határoló metszéspont, egy utolsó mediánkereséssel ki tudnánk számítani, hogy melyik f_i a medián ezen a szakaszon, és adott $\lambda^* = -\frac{b_i}{a_i}$. Ebben a megközelítésben a metszéspontok kiszámítása emészti fel az idő túlnyomó részét.

A paraméteres keresés első fontos gondolata az, hogy futtassuk a mediánkereső algoritmust szimbolikusan az ismeretlen λ^*-ra! Ezt úgy tudjuk megvalósítani, hogy a szimbolikus algoritmusfuttatásban a változóink λ^*-nak lesznek lineáris függvényei (tehát $a\lambda^* + b$ értékű változó esetén az (a, b) számpárt tároljuk el). Az egyetlen nehézség a szimbolikus futtatásban történő szimbolikus változók összehasonlításából származik, viszont ezt is meg tudjuk csinálni:

Ha $f_i(\lambda^*) < f_j(\lambda^*)$-re vagyunk kíváncsiak feltéve hogy $a_i < a_j$, csak azon múlik az összehasonlítás igazságértéke, hogy x_{ij}-hez (ha létezik) λ^* hogy viszonyul. ($a_i > a_j$ esetben analóg módon meg tudjuk tenni)

- ha két lineáris függvény nem metszi egymást, a nagyobb b_i konstans taggal rendelkező lesz $\forall x$-re nagyobb

- különben $x = x_{ij}$-ben $f_i = f_j$
- $x < x_{ij}$-re $a_i < a_j$ miatt $f_i(x) > f_j(x)$
- $x > x_{ij}$-re pedig $f_i(x) < f_j(x)$

Ezt az x_{ij}-t fogjuk az adott szimbolikus összehasonlítás **kritikus értékének** nevezni a továbbiakban.

Megjegyzés. Erre érdemes úgy gondolni, hogy $g_k(x) := f_i(x) - f_j(x)$-t hasonlítjuk össze 0-val. A 3.1 ábrán is ez szerepel.

α monoton csökken, így

- $\alpha(x_{ij}) > 0 \implies \lambda^* > x_{ij}$, tehát $f_i(x) < f_j(x)$
- $\alpha(x) < 0 \implies \lambda^* < x$, tehát $f_i(x) > f_j(x)$

Így minden összehasonlításnál meg tudjuk hívni a numerikus mediánkereső algoritmust $\alpha(x_{ij})$ kiértékelésére. A szimbolikus mediánkereső algoritmus így kiszámolja a *szimbolikus* mediánt λ^*-ben, ami egyrészt egy lineáris kifejezés, másrészt per λ^* definíciója 0. Ha $h(x)$ a kapott medián, akkor a $h(x) = 0$ lineáris egyenlet megoldása λ^*. Ezzel kaptunk egy második $O(n^2)$ futásidejű algoritmust. Itt jön be a párhuzamos algoritmusok szerepe. A párhuzamosságot kihasználva ezen jelentősen tudunk gyorsítani.

Ha egyszerre adott m kiértékelendő szimbolikus összehasonlítás, ezeket egyszerre hatékonyan ki tudjuk értékelni:

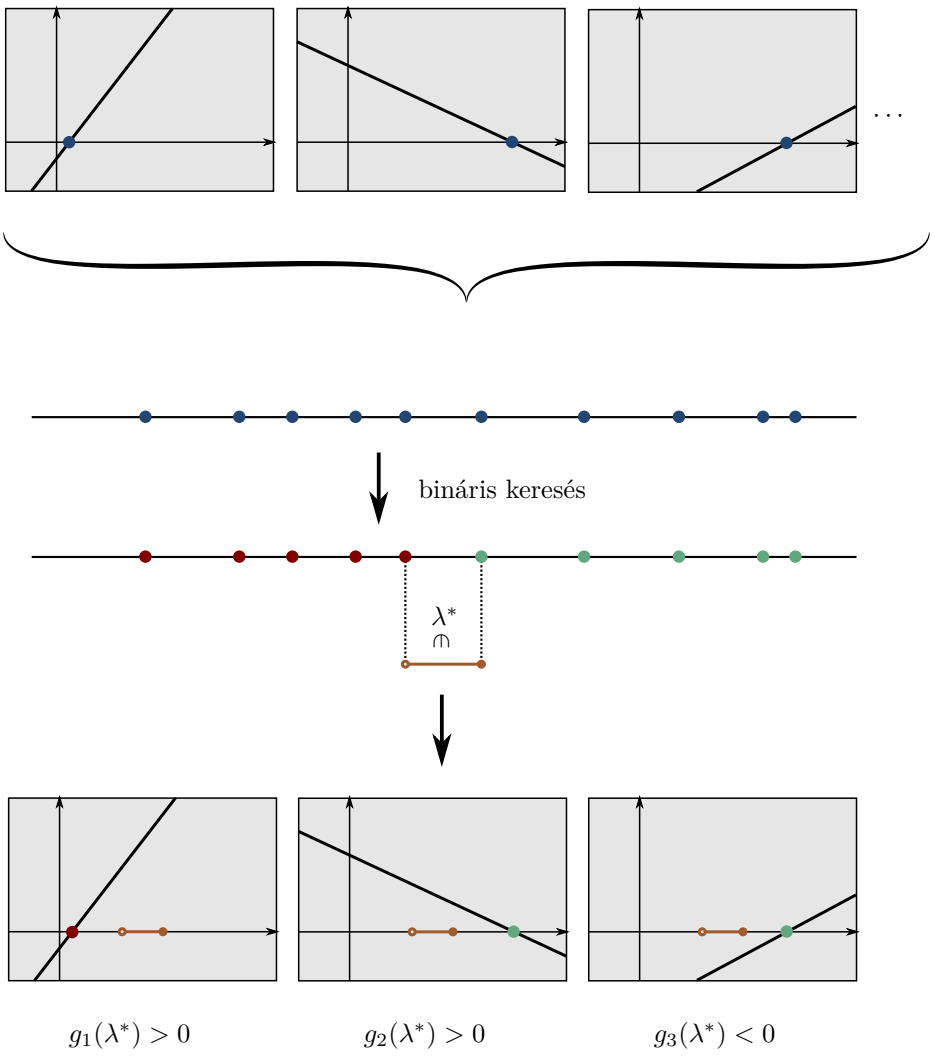
Az összehasonlítások kritikus értékeit először rendezzük sorba: $c_1 < \cdots < c_m$. Mindig az el nem dőltek közül a középső $c_{\lfloor \frac{m}{2} \rfloor}$ kritikus értékére futtatjuk numerikusan kiértékeljük a mediánt. A sorbarendezés miatt: $\lambda^* < c_{\lfloor \frac{m}{2} \rfloor} \implies \lambda^* < c_i\,(\forall i > \lfloor \frac{m}{2} \rfloor)$, ezzel feleztük azon kritikus pontok számát, amiben α előjele ismeretlen. Dobjuk ki az eldőlt kritikus pontokat és ismételjük ezt a felezgetést. Így m összehasonlítást ki tudunk értékelni a $\log m$ darab futtatásával a *numerikus* mediánkereső algoritmusnak.

Ha egy kritikus érték egybeesik a 3.1 ábrán szereplő barna intervallum jobb szélével, akkor két eset lehetséges:

- Kiszámoltuk ott α-t és 0. Ebben az esetben megtaláltuk λ^*-t.
- α ott pozitív: ebben az esetben az λ^*-t tartalmazó barna intervallum jobbról is nyílt, így eldőlt az összehasonlítás.

Megjegyzés. A 3.1 ábrán a λ^*-t tartalmazó barna intervallum azért jobbról zárt, hogy a következő fejezetben szereplő általánosítást tükrözze.

A külső szimbolikus $\alpha(\lambda^*)$ kiértékelésre tehát használjunk az $O(\log n)$ futásidejű AKS rendező hálózatot. Ez $T_p = O(\log n)$ időben rendez $P = O(n)$ processzorral. Mivel rendezési háló könnyen szimulálható szekvenciálisan, így minden szimulált párhuzamos lépésben az $O(n)$ kritikus értéket sorbarendezzük, $O(n \log n)$-idő alatt majd $O(\log n)$-szer kell futtatni a numerikus mediánkeresést, így az párhuzamos algoritmus szekvenciális szimulálásának ideje (összehasonlítások nélkül) $O(n \log n) = T_p \cdot P$ és az összes összehasonlítás kiértékelésének ideje $T_p \cdot (n \log n + \log P \cdot n) = O(n \log^2 n)$, tehát az algoritmus összfutásideje $O(n \log n + n \log^2 n) = O(n \log^2 n)$.



3.1. ábra. párhuzamos szimbolikus lépés kiértékelése

3.2. Standard alak

A következő tétel Smid [Smi02] jegyzetéből származik, és ad egy keretrendszert paraméteres keresési algoritmusok egy típusára. Ez a Megiddo által [Meg83a]-ban bevezetett módszert általánosítja legfeljebb negyedfokú szimbolikus kifejezésekre. Futásidőre használom a Radziktól származó [Rad98]§5.3-beli javítást. A bizonyítás a Smid által [Smi02] jegyzetében szereplőhöz hasonló, de némileg egyszerűbb. Az ezt követő részben szereplő alkalmazások ilyen alakba írhatók, így a keretrendszert használva adunk rájuk algoritmusokat.

3.2.1. Tétel. *Adott* $\alpha : \mathbb{R} \to \{igaz, hamis\}$ *függvény, ami* $\alpha(x) = \begin{cases} hamis & ha\ x < \lambda^* \\ igaz & ha\ x \geq \lambda^* \end{cases}$ *alakú* ($\lambda^* \in [-\infty, \infty]$). *Adott még* $\mathcal{P}$ *párhuzamos algoritmus, ami* T_p *idő alatt kiszámolja* P *processzorral* α *értékét úgy, hogy minden változó, ami szerepel összehasonlításban,* x*-nek alacsonyfokú* ($\deg \leq 4$) *polinomja, és egy* $\mathcal{S}$ *szekvenciális algoritmus, ami* T_s *idő alatt kiszámítja* α *értékét, akkor konstruálható* $\mathcal{M}$ *algoritmus, ami* $O(T_p P + T_p T_s \log P)$ *időben kiszámítja* λ^**-t.*

Megjegyzés. Az α függvény többnyire valamilyen kombinatorikai vagy geometriai struktúrán fog múlni, például: „van-e legfeljebb x költségű feszítőfa egy adott gráfban”, vagy „le lehet-e fedni egy adott ponthalmazt két x sugarú körrel”.

Megjegyzés. Ha $\mathcal{S}$ által használt tárhely $O(H_s)$, és $\mathcal{P}$ által használt tárhely $O(H_p)$, akkor a kapott algoritmus $O(P + H_p + H_s)$ helyet használ.

Megjegyzés. Ha a $\mathcal{P}$ összműveletszáma nem jelentősen nagyobb, mint $\mathcal{S}$ futásideje, akkor a futásidő $O(T_p T_s \log P)$ lesz.

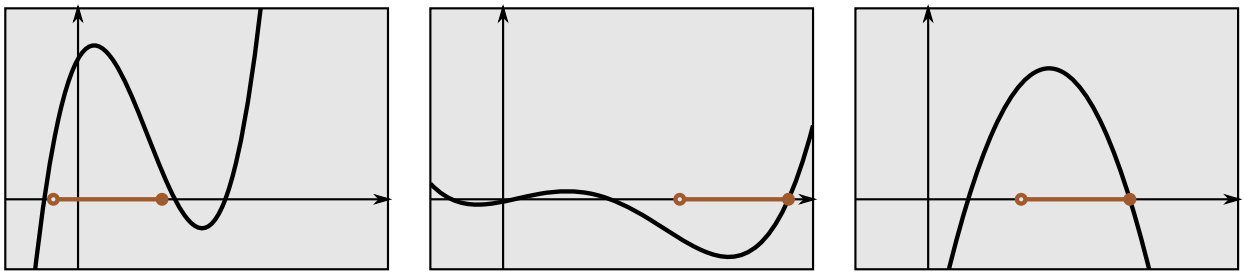
$\mathcal{M}$ futása alatt számontartunk egy I intervallumot, ami kezdetben $[-\infty, \infty]$, futás közben pedig $(a, b]$, ahol a a legnagyobb érték, amit $\mathcal{S}$ hamisra értékelt ki, b pedig a legkisebb, amit igazra. Világos, hogy $\lambda^* \in I$ monotonitás miatt.

Function eval(x, I) Gyakorlatban így néz ki egy kritikus érték kiértékelése $\mathcal{S}$ felhasználásával

input : x a kritikus érték, $I = (a, b]$ a jelenlegi intervallum
output: ($\alpha(x)$ I), ahol J-re frissül I

```
1  if x ≤ a then // Gyakorlati optimalizáció, ekkor α monotonitás miatt igaz
2  |  return (hamis, I);
3  else if x ≥ b then// Gyakorlati optimalizáció, ekkor α monotonitás miatt hamis
4  |  return (igaz, I);
5  else
6  |  α ← S(x) ; // S meghívása mint szubrutin α numerikus kiértékelésre
7  |  if α = igaz then
8  |  |  return (igaz, (a, x]); // Ha a = −∞ ez (igaz, [−∞, x])
9  |  else
10 |  |  return (hamis, (x, b]);
```

Feltehetjük, hogy az összes összehasonlítás „$a \leq b$” alakú $a \leq b \iff \neg(b < a)$ miatt. Az összehasonlítások ugyanúgy azon múlnak, hogy λ^* a két polinom metszéspontjainak x koordinátáihoz hogyan viszonyul. Mivel az összehasonlítandó polinomok foka legfeljebb 4, különbségeikre is teljesül. Így az összehasonlítás egy legfeljebb negyedfokú polinom gyökeinek kiszámítására vezethető vissza.



3.2. ábra. összehasonlítások különböző esetei

Ha I két gyök közé esik, mint a 3.2 ábra jobboldali részén, akkor világos, hogy az egész I-re eldőlt az összehasonlítás eredménye eldőlt és ez ugyanaz minden elemére I-nek. Ez akkor is így van, ha a p polinomra $p(b) = 0$, de b egy baloldali $(b-\varepsilon, b]$ környezetében p monoton nő mint a 3.2 ábra középső részén. Az egyetlen kellemetlen eset az, amikor a jobboldali részhez hasonló módon $p(b) = 0$ és egy $(b-\varepsilon, b]$ jobboldali környezeten p csökken. Ekkor $p(b) \leq 0$, viszont tetszőleges $x \in (a, b)$-re $p(x) > 0$. Emiatt, ha $J = (a, b)$ az algoritmus legvégén fennálló I belseje, akkor minden $x \in J$-re a szimbolikus algoritmus lefutása alatt pontosan ugyanúgy jól döntünk minden elágazásnál, tehát α a J halmazon konstans.

Összehasonlítást tehát tudunk kezelni, így az előző feladaton bemutatott módszer működik: szimuláljuk le a párhuzamos algoritmust a processzorokon végigiterálva minden lépésben, aggregálva az összehasonlítások kritikus értékeit, majd rendezzük őket, és bináris kereséssel találjuk meg a barna intervallumot.

Világos, hogy itt az I-beli kritikus értékekre elég ezt megtenni, mert a többire következik α értéke monotonitásából.

Tehát végigmegyünk az összes processzoron és a szimulált „párhuzamos” lépésre az összes összehasonlítás kritikus pontjait összegyűjtjük. Legfeljebb $4P$ kritikus pontot kapunk, mert a polinomok legfeljebb negyedfokúak. Ezeket $O(P \log P)$ időben rendezünk, és $O(\log P)$ darab meghívásával $\mathcal{S}$-nek bináris kereséssel megkapjuk, hogy

λ^* hogyan viszonyul az összes kritikus ponthoz. Ennek ismeretében ismét végigiterálva az összes processzoron végre lehet hajtani a szükséges lépéseket.

Algorithm 3.1: Általános paraméteres keresésre az $\mathcal{M}$ algoritmus

```
1  I ← [−∞, ∞];
2  foreach parallel step s of P do
3      CriticalPoints ← ∅;
4      foreach processor p do
5          if p does a comparison c in step s then
6              CriticalPoints ← CriticalPoints ∪ {critical points of comparison c};
7      SortedCriticalPoints ←sort(CriticalPoints ∩ I);
8      left ← 0;
9      right ← number of elements in SortedCriticalPoints;
10     for left ≠ right − 1 do // Bináris keresés a kritikus pontokon
11         mid ← ⌊(left + right)/2⌋;
12         α, I ←eval(SortedCriticalpoints[mid], I);
13         if α then
14             right ← mid;
15         else
16             left ← mid;
17     foreach processor p do
18         do step s of p ; // Összehasonlítások kimenetele aszerint, hogy I melyik
                két gyök közé esik
   // itt ha I = (a, b], tehát (a, b) minden elemére ugyanaz az értéke α-nak
19 switch value of I do
20     case I = [−∞, ∞] do
           // triviális eset, nem történik összehasonlítás, ami λ*-on múlik
21         if α(0) = igaz then
22             return −∞;
23         else
24             return ∞;
25     case I = [−∞, b] : b ≠ ∞ do // ekkor α(b) = igaz ismert
26         if α(b − 1) = igaz then
27             return −∞;
28         else
29             return b;
30     case I = (a, b] : a ≠ −∞ do // ekkor α(a) = hamis ismert
31         return b;
```

A bináris keresés előtti rendezés miatt a kapott futásidő $O(T_pP \log P + T_pT_s \log P)$. Ez a következő 3.2 algoritmusban leírt eljárással levihető $O(T_pP + T_pT_s \log P)$-re: keressük mediánját a kritikus pontoknak, és miután arra kiértékeljük α-t dobjuk ki a kritikus pontokat, amikre α monotonitása miatt ezzel eldőlt az értéke. Ezt ismételjük, amíg van kiértékeletlen kritikus pont.

A mediánkeresés futásideje $O(n)$, amint 1.2-ben láttuk (ezt tudnánk adaptálni, hogy páros elemszám esetén a két középső elem átlagát kapjuk, de ilyen formájában is kevesebb mint $\frac{2}{3}$-ára csökken a megmaradó pontok száma), és minden keresés legfeljebb fele akkora halmazon történik, mint az előző. A futásidő egy párhuzamos lépés szimulálására tehát:

$$O\left(\sum_{i=0}^{\lceil \log P \rceil}\left(\frac{P}{2^{i-1}} + T_s\right)\right) = O\left(\sum_{i=0}^{\lceil \log P \rceil}\frac{P}{2^{i-1}} + T_s \log P\right) \leq O(4P + T_s \log P)$$

Ezt minden fázisban megtesszük, tehát T_p-vel szorzódik. Ez az $O(T_pP + T_pT_s \log P)$ a kívánt futásidő.

Algorithm 3.2: Tegyük a következőt a 7-16 sorok helyett:

```
1 while |CriticalPoints| > 0 do
2     m ← median(CriticalPoints);
3     α, I ←eval(m, I);
4     if α then
5         CriticalPoints ← {point ∈ CriticalPoints : point < m};
6     else
7         CriticalPoints ← {point ∈ CriticalPoints : point > m};
```

A következő erősebb becslést is lényegében kihoztuk a futásidőre:

3.2.2. Állítás. *Ha az i-edik lépésben C_i összehasonlítás történik, T_p lépés van, és a párhuzamos összlépésszáma W_p akkor a futásidő*

$$O\left(W_p + \sum_{i=1}^{T_p}(C_i + T_s \log C_i)\right)$$

Megjegyzés. Ez az alak akkor hasznos, ha sok lépés van, amiben nem történik összehasonlítás, vagy az összehasonlítások száma egy lépésben erősen korlátozott.

3.2.3. Állítás. *Az $\mathcal{M}$ algoritmus kiszámítja λ^*-t.*

Bizonyítás. Az (a, b) intervallum minden pontjára ugyanaz α értéke, hiszen az, hogy a szimbolikus futtatás végén $(a, b]$ lett I bizonyítja, hogy pontosan ugyanúgy futna le minden ebből vett pontra az algoritmus. (b-re a fennt említett hibalehetőség miatt nem garantált, sőt általában nem ugyanaz – amint az alábbiakban látni fogjuk.)

- Az $I = [-\infty, \infty]$ eset triviális, hiszen ekkor az egész szimbolikus algoritmusfutás alatt nem történt olyan összehasonlítás, ami a paraméter értékén múlt. Ekkor $\alpha \equiv igaz$, avagy $\alpha \equiv hamis$, tehát elég egy tetszőleges értékre kiszámítani $\mathcal{S}$-el, hogy megtudjuk melyik áll fenn.
- Ha $I = [-\infty, b] : b \neq \infty$, akkor tudjuk, hogy $\alpha(b) = igaz$, és mivel $(-\infty, b)$ halmazon α konstans, csupán annyi a kérdés, hogy van-e igaz elem ebben a halmazban, ami $\alpha(b-1)$-et $\mathcal{S}$-el kiszámítva eldől.
- Ha $I = (a, b] : a \neq -\infty$, akkor tudjuk, hogy $\alpha(a) = hamis$. Mivel (a, b)-n konstans α, ha igaz lenne $\alpha(a) = igaz$ lenne, hiszen α igaz értékeinek halmaza balról zárt. Így tehát $\lambda^* = b$.

□

Megjegyzés. A bizonyításból következik, hogy ha a és b valós az algoritmus végén, akkor a szimbolikus algoritmus futása során kritikus érték volt λ^*.

3.2.4. Állítás. *A fenti futásidővel van $\mathcal{M}$ algoritmus λ^* kiszámítására ha $\alpha(x) = \begin{cases} igaz & ha\, x \leq \lambda^* \\ hamis & ha\, x > \lambda^* \end{cases}$ alakú.*

Bizonyítás. $\alpha'(x) = \alpha(-x)$-re tudjuk futtatni az előbbi algortimust, és a végén kapott $\lambda^{*\prime}$ -1-szerese λ^*. □

3.2.1. DAG-ban legdrágább 0 költségű út

A következő Radziktól származó megoldás a [Rad98]§5-ben tárgyalt feladat törtlineáris optimalizálási feladat megoldására is használható, amint 3.4-ban látni fogjuk.

3.2. Feladat. *Adott egy körmentes irányított gráf, csúcsainak egy $v_1, \cdots, v_n$ topologikus rendezése és élein f, g költségfüggvények, hogy $g > 0$. Mi az a maximális λ, amire a $f - \lambda g$ költségfüggvény szerint legdrágább v_1-ból v_n-be vezető út költsége 0? (tegyük fel, hogy van út v_1-ból v_n-be)*

Itt $\alpha(x) = \begin{cases} hamis & \text{ha } f - \lambda g \text{ szerinti legdrágább út költsége} < 0 \\ igaz & \text{különben} \end{cases}$

A következő algoritmust használhatjuk mint $\mathcal{P}$ és $\mathcal{S}$:

Algorithm 3.3: maximális költségű út DAG-ban

```
1 c_i ← 0 (∀i ∈ {1, ··· , n});
2 for i = 1 to n do
3 |   parallel for j : (v_i, v_j) ∈ E do
4 |   |   c_j ← max(c_j, c_i = c(v_i, v_j));
5 return c_{n−1};
```

Az ebből kapott $\mathcal{M}$ algoritmus futásideje $O(T_p P + T_p T_s \log P) = O(n^2 + nm \log n) = O(nm \log n)$.

3.2.2. Mozgó pontok halmazának minimális átmérője

Ez a Smidtől származó [Smi02]-ben tárgyalt alkalmazás az egyetlen a dolgozatban, amiben nem csak lineáris szimbolikus kifejezések kerülnek összehasonlításra, kihasználva a Standard alak általánosabb voltát.

3.3. Feladat. *Adottak $p_1, \cdots, p_n$ pontok, és p_i helye t időpontban $h_i(t) := (v_i t + a_i, u_i t + b_i)$. Számítsuk ki a λ^* időpontot, amiben a ponthalmaz $D(t) = \max_{i,j}\{d(p_i(t), p_j(t))\}$ átmérője minimális.*

$D(t)$ helyett nézzük $D^2(t)$-t. Ennek világos, hogy ugyanott lesz a minimuma, viszont egyszerűbb számolni a négyzetgyökök nélkül.

Legyen $f_{ij}(t) = d^2(p_i(t), p_j(t)) = (a_i - a_j + (v_i - v_j)t)^2 + (b_i - b_j + (u_i - u_j)t)^2$.

Az egyszerűség kedvéért végett tegyük fel, hogy minden sebességvektor egyedi. Ekkor minden f_{ij} valódi másodfokú függvény. Ekkor $D^2(t) = \max_{i,j}\{f_{ij}(t)\}$ konvex függvény, hiszen konvex függvények maximuma, sőt: monoton csökken λ^*-ig, és onnan monoton nő.

3.2.5. Lemma. *Ha t^*_{ij} az időpont, amikor $d(p_i, p_j)$ minimális, azaz* $\operatorname{argmin} d(p_i(t), p_j(t))$*, akkor* $\lambda^* < t \iff t^*_{ij} < t \; \forall i, j \in \{1, \cdots, n\} : f_{ij}(t) = D^2(t)$

Bizonyítás. $\Longrightarrow$ irány: $\lambda^* < t \implies D^2$ t-ben szigorúan monoton nő, így $\forall i, j : f_{ij}(t) = D^2(t)$ szigorúan monoton nő f_{ij} t-től. Ebből viszont következik, hogy $t < t' \implies D^2(t') > D^2(t)$, így a minimum $\lambda^* < t$ $\Longleftarrow$ irány: $t^*_{ij} < t \; \forall i, j \in \{1, \cdots, n\} : f_{ij}(t) = D^2(t)$, akkor D^2 a t pont után szigorúan monoton nő, hiszen a pontpárok amik nem kisebb távolságúak t-ben csak úgy tudják „utolérni" D^2-et és az értéke lenni t utáni pontban, ha elkezdenek nőni. Ráadásul t egy környezetében is szigoróan monoton nő amiatt, hogy $t^*_{ij} < t$ minden pontpárra ami felveszi a távolságot. Mivel λ^* a minimum, ezért $\lambda^* < t$.

□

Tehát $\mathcal{S}$-nek olyan algoritmust kell választanunk, ami kiszámítja az összes maximum átmérő távolságot felvevő pontpárt. Ezekre a pontpárokra pedig megnézzük, hogy a minimumuktávolságuk felvétele előtt vagyunk-e. Így a Smid által [Smi03]§3.1-ban leírt $O(n \log n)$ futásidejű algoritmussal ezt meg tudjuk tenni.

$\mathcal{P}$-nek válasszuk a párhuzamosított átmérőszámító algoritmust, amit Smidtől [Smi03]§4-ben tárgyal. Ez $O(\log n)$ futásidejű, $O(n)$ processzoros algoritmust, ami először konvex burkot számít, majd párhuzamos bináris kereséseket végez, hogy minden élre kiszámítsa azt a pontot, amin az éltől legmesszebbi azzal párhuzamos egyenes átmegy. Ekkor már csak az élek és azokhoz talált pontok távolságai közül kell kiválasztani a maximumot.

Fontos megjegyezni, hogy az algoritmus a paraméter által meghatározott pontok szögeinek irányait nézi, avagy háromszögek körüljárásait számolja ki, azaz

$$\begin{vmatrix} v_i t + a_i & u_i t + b_i & 1 \\ v_j t + a_j & u_j t + b_j & 1 \\ v_k t + a_k & u_k t + b_k & 1 \end{vmatrix}$$

alakú mátrixdeterminánsok előjelét számolja ki egyes elágazásoknál. Ezentúl ilyen mátrixdeterminánsokat hasonlít is össze, hogy összehasonlítson szögeket. Ezek a determinánsok t-nek másodfokú polinomjai, így ebből nincs probléma.

A kapott $\mathcal{M}$ futásideje $O(T_p P + T_p T_s \log P) = O(n \log n + \log n n \log n \log n) = O(n \log^3 n)$.

3.2.3. Fa max-min k-partícionálása

A következő Megiddotól származó [Meg83a]§7-ben leírt algoritmus egy kombinatorikai optimalizálási feladatot old meg. Ehhez felhasználásra kerül a 1.3-ban látott centroid dekompozíciót és a 2.7 algoritmust is prefix összeg kiszámítására.

3.4. Feladat. *Adott egy T fa és csúcsain egy s nemnegatív súlyfüggvény. Mi az a maximális λ, amire ki lehet törölni k élt a fából, hogy az így keletkező komponensek mindegyikében legalább λ legyen az összsúly?*

Megjegyzés. Ez a feladat fa helyett általános gráfra NP teljes.

Ebben az esetben természetesen adódik

$$\alpha(\lambda) = \begin{cases} igaz & \text{ha el lehet hagyni } k \text{ élt, hogy minden komponens súlya legalább } \lambda \\ hamis & \text{különben} \end{cases}$$

tehát ez a 3.2.4-beli alak, így elég párhuzamos és szekvenciális algoritmust adnunk α kiértékelésére.

α kiértékelésére természetesen adódik a következő dinamikus programozáson alapuló algoritmus:

A fának jelöljünk ki egy gyökeret, majd:

1. Minden levélre aminek súlya legalább λ töröljük a belőle menő élt.
2. Minden csúcsra, aminek csak levél gyereke van, ha az összes gyerekének súlya kisebb λ-nál, akkor ezeket vonjuk össze egy csúcsba.
3. Ha még van él térjünk vissza 1-hez.

Számontartjuk a kitörölt élek k' számát, a végén a számot eggyel csökkentve ha az utolsó csúcs súlya kisebb λ-nál. (amennyiben élek kitörlése nélkül is kisebb az összsúly λ-nál, akkor -1-et kapunk) Ez egy mélységi bejárással $O(n)$ időben megtehető.

Ebből adódik a következő részben párhuzamos algoritmus is:

Vegyük észre, hogy az összegzős ciklus minden csúcsot legfeljebb egyszer érint mint gyerek, tehát az összfutásideje az algoritmus során $O(n)$. Emellett nincs benne elágazás, így a 3.2.2 állításbeli erős futásidőbecslést haszálva, ha a fa átmérője $d(T)$, a kapott $\mathcal{M}$ algoritmus futásideje $O(n + d(T)n \log n) = O(d(T)n \log n)$

Ezen tudunk tovább javítani jobb párhuzamos algoritmus használatával. Először megoldjuk a feladatot abban az esetben, ha T út $O(\log n)$ időben, majd ezt felhasználva kapunk $O(\log^2 n)$ idejű algoritmust általános T-re.

A feladat megoldása ha F út

A paraméteres kereséshez az alábbi feladatot megoldó hatékony párhuzamos algoritmust kell $\mathcal{P}$-nek választanunk, amennyiben az F fa egy út.

Algorithm 3.4: Egyszerű részben párhuzamos algoritmus

```
1  while |T| > 1 do
2      parallel for leaf l of T do
3          if s(l) > λ then
4              k ← k + 1// ez párhuzamosan nem működne, de a szekvenciális
                   szimuláció miatt nem baj
5              delete l;
6      for vertex v: all children are leaves do
7          S_v ← sum of s for v's children;
8          S_v ← S_v + s(v);
9          new v' vertex in place of v with s(v') = S_v;
   // ezen a ponton egy v csúcsa van már csak a fának
10 if s(v) < λ then
11     k ← k − 1;
```

3.5. Feladat. *Legyen $a_1, \cdots, a_n$ nemnegatív számok egy sorozata, és λ adott.*

Legfeljebb hány $(a_1, \cdots, a_{i_1}), (a_{i_1+1}, \cdots, a_{i_2}), \cdots, (a_{i_r}, \cdots, a_n)$ blokkra lehet partícionálni, hogy mindegyikben legalább λ legyen a számok összege?

Minden $i \in \{0, \cdots, n-1\}$-re ki fogjuk számítani $k(i)$ és $s(i)$-t, ahol $a_{i+1}, \cdots, a_n$-t legfeljebb $k(i)$, darabonként legalább λ összsúlyú blokkra tudjuk partícionálni. $s(i)$ pedig a legkisebb index, amire $a_{i+1}, \cdots, a_{s(i)}$ is $k(i)$ darab legalább λ összsúlyú blokkra partícionálható.

$k(0)$ a feladatban kívánt érték, de ennek hatékony párhuzamos kiszámításához hasznos a többi. Ebben $j(i)$: $A_{j(i)-1} - A_i < \lambda \leq A_{j(i)} - A_i$ is segítségünkre lesz, amit minden -re külön bináris kereséssel meg tudunk találni. Ez azt mondja meg, hogy hol kezdődik garantáltan következő blokk, tehát a következő blokk megtalálására fogjuk használni.

Rekurzívan $s(i)$ és $k(i)$ előáll a következő módon: ha $s'(i)$ és $k'(i)$ az $a_1, \cdots, a_{\lfloor n/2 \rfloor}$ számokra vonatkozik, és a $a_{\lfloor n/2 \rfloor + 1}, \cdots, n$ számokra tudjuk s és k-t, akkor ha $j(s'(i)) \leq n$

$$
\begin{aligned}
k(i) = & k'(i) && \text{mivel ennyit már } s'(i)\text{-ig is tudtunk} \\
& + 1 && \text{mivel a } j \text{ miatti „áthidaló ugrás” eggyel növeli a blokkok számát} \\
& + k(j(s'(i))) && \text{mivel ennyit tudunk } j(s'(i))\text{-től } s(j(s'(i)))\text{-ig}
\end{aligned}
$$

és $s(i) = s(j(s'(i)))$ ebből adódó módon. Különben ha $j(s'(i)) > n$, akkor $k(i) = k'(i)$ és $s(i) = s'(i)$.

Algorithm 3.5: Gyors algoritmus útra iteratívan

input : $a_1, \cdots, a_n$
output: k

calculate A_i prefix sums;
parallel for $i \in \{1, \cdots, n\}$ **do**
  find $j(i)$: $A_{j(i)-1} - A_i < \lambda \leq A_{j(i)} - A_i$ using binary search; // $O(\log n)$ idő, és $j(i) \leftarrow \infty$ ha nincs ilyen j
parallel for $i \in \{0, \cdots, n-1\}$ **do**
  if $a_{i+1} \geq \lambda$ **then**
    $k(i) \leftarrow 1$;
    $s(i) \leftarrow i + 1$
  else
    $k(i) \leftarrow 0$;
    $s(i) \leftarrow i$;
for $iteration \in \{2^0, 2^1, \cdots, 2^{\lfloor \log n \rfloor}\}$ **do**
  parallel for $i \in \{0, \cdots, n-1\}$ **do**
    if $iteration \oplus i = 0$ *and* $j(s(i)) < \left\lceil \frac{i+1}{iteration} \right\rceil \cdot iteration + iteration$ **then**
      // i egy páratlanadik $iteration$ méretű blokkban van és $j(s(i))$ legfeljebb eggyel későbbi $iteration$ méretű részbe ugorhat át
      $s(i) \leftarrow s(j(s(i)))$;
      $k(i) \leftarrow k(i) + 1 + k(j(s(i)))$;
return $k(0)$;

3.2.6. Tétel. *A 3.5 algoritmus $O(\log n)$ futásidejű, tehát utakra az eredeti feladat megoldható $O(T_p P + T_p T_s \log P) = O(n \log n + n \log^2 n) = O(n \log^2 n)$ időben.*

Általános fa esete

3.6. Feladat. *Adott egy T fa és csúcsain egy s nemnegatív súlyfüggvény. Legfeljebb hány élt lehet kitörölni a fából, hogy az így keletkező komponensek mindegyikében legalább λ legyen az összsúly?*

Erre akarunk tehát hatékony párhuzamos algoritmust. Legyen egy tetszőleges u csúcs a fa gyökere. A rekurzívan levelektől definiált $g(v)$ függvényt szeretnénk kiszámítani.

$$g(v) := \begin{cases} 0 & \text{ha } v \text{ levél és } s_v \geq \lambda \\ s_v & \text{ha } v \text{ levél és } s_v < \lambda \\ s_v + \sum_{i=1}^{k} g(v_i) & \text{különben, ha ez } < \lambda, \text{ ahol } v_1, \cdots, v_k \ v \text{ gyerekei} \\ 0 & \text{különben, amikor } s_v + \sum_{i=1}^{k} g(v_i) \geq \lambda \end{cases}$$

Tehát ha $g(v) = 0$, akkor a v-ből szülőjébe menő élt kitöröljük, illetve ha az u gyökérre nemnulla, akkor a komponensét hozzácsatoljuk egy tetszőleges törlendő éllel kapcsolódóhoz.

Centroid dekompozíción (lásd 1.3) múlik az algoritmus: ez előre számítható az egész fára $O(n \log n)$ időben, és nem múlik λ értékén.

Ha v_0 a centroid és $v_0, v_1, \cdots, v_r = u$ az onnan a fa gyökerébe vezető út, akkor az összes olyan fára, aminek gyökere az útról „lóg le” kiszámítjuk rekurzívan és párhuzamosan g-t, azaz ha v_j szomszédjai v_{j-1}, v_{j+1} és $u_{j,1}, u_{j,2}, \cdots, u_{j,k(j)}$, akkor az $u_{j,1}, u_{j,2}, \cdots, u_{j,k(j)}$ gyökerű fákra számítjuk ki $\forall j$-re.

Ekkor legyen $a_j := s_{v_j} + \sum_{i=1}^{k(i)} g(u_i)$, amit $O(\log k(i))$ időben ki tudunk számolni 2.2.1-ben látott módon. (de nincs is benne elágazás) Ezzel visszavezettük g kiszámítását az előbbi részbeli egyszerűbb esetre, amiről láttuk, hogy $O(\log r) \leq O(\log n)$ időben tudunk számolni.

Tegyük fel, hogy az algoritmus *párhuzamos* futásideje n csúcsú fára a $T(n)$. Ekkor a rekurzív kiszámítási módból adódóan, mivel a centroid garantálja, hogy minden részfa mérete legfeljebb fele az eredetinek: $T(n) \leq T(n/2) + c\log n$. Ebből $T(n) = O(\log^2 n)$, hiszen indukcióval $T(n) \leq c' \log^2 n$ kijön $c' = \max(c, 1)$ választással:

$$T(n) \leq T(n/2) + c\log n \leq c' \log^2 \frac{n}{2} + c\log n = \\ = c' \log^2 n - 2c' \log n + 1 + c\log n = \\ = c' \log^2 n + \underbrace{(1 - c' \log n)}_{\leq 0} + \underbrace{(c\log n - c' \log n)}_{\leq 0} \leq c' \log^2 n$$

3.2.7. Tétel. *A paraméteres keresésből adódó algoritmus futásideje ezzel a $\mathcal{P}$-vel $O(T_p P + T_p T_s \log P) = O((n + n\log^2 n) + (n\log^3 n)) = O(n\log^3 n)$.*

Megjegyzés. Erre a feladatot Frederickson és Zhou [FZ17]-ben $O(n)$ futásidejű megoldást adnak egy bonyolultabb, paraméteres keresést felhasználó algoritmus formájában.

3.3. Standard alak speciális esetben

Az alábbiakban a standard alak azon alesetét vizsgáljuk, ahol el tudjuk dönteni, hogy optimális-e tetszőleges x. Ez nagyon sok feladatra teljesül és látni fogjuk, hogy a szimbolikus $\mathcal{P}$ algoritmusnak nem kell pont ugyanazt a α-t kiszámítania. Ezzel jelentősen több lehetőségünk lesz, amikor paraméteres keresési algoritmust konstruálunk. Ezt a szabadságot használjuk majd ki 3.6-ban Cole gyorsításánál is, hogy a külső szimbolikus algoritmust rendező algoritmusnak válasszuk. Megiddo [Meg83a] és [Meg79] cikkjeiben ilyen alakú feladatokkal foglalkozik, a további példák többsége tőle származik.

A standard alak körülményessége abból adódott, hogy α kiszámításával nem tudjuk meg, hogy λ^*-ban vagyunk. Ha $\alpha(x) = \begin{cases} \textit{kisebb} & \text{ha } x < \lambda^* \\ \textit{optimum} & \text{ha } x = \lambda^* \\ \textit{nagyobb} & \text{ha } x > \lambda^* \end{cases}$ alakú ugyanaz az algoritmus működik, és azt a gyakorlati optimalizációt érdemes megtenni, hogy leállítjuk az algoritmust, ha $\mathcal{S}$ „$\alpha(x) = \textit{optimum}$”-ot számított ki.

Az általános alak utáni megjegyzés fényében gyakran korán leállítja az algoritmus futását, viszont tényleges aszimptotikus javulást nem eredményez. Előnye viszont, hogy lefutott az algoritmus garantálja, hogy λ^*-re a kapott értéket adná $\mathcal{P}$. Ez hasznos, ha ebből az értékből λ^* visszaszámolható.

Sok alkalmazásnál a párhuzamos $\mathcal{P}$ algoritmus nem α-t számítja ki, hanem valamit, amiből α számolható. Kicsit formálisabban:

3.3.1. Állítás. *Adott egy $\alpha : \mathbb{R} \to \{\textit{kisebb}, \textit{optimum}, \textit{nagyobb}\}$, ami*

$$\alpha(x) = \begin{cases} \textit{kisebb} & \textit{ha } x < \lambda^* \\ \textit{optimum} & \textit{ha } x = \lambda^* \textit{ alakú } (\lambda^* \in [-\infty, \infty]\textit{-ra}). \\ \textit{nagyobb} & \textit{ha } x > \lambda^* \end{cases}$$

Emellett adott egy $\mathcal{S}$ szekvenciális algoritmus, ami T_s idő alatt kiszámítja α értékét. Továbbá adott $\mathcal{P}$ párhuzamos algoritmus, ami T_p idő alatt kiszámítja P processzorral λ-nak kiszámítja úgy, hogy minden változó ami szerepel összehasonlításban λ-nak kis fokú ($\deg \leq 4$*) polinomja, egy $\mathfrak{F}(\lambda)$ függvényét, amiből λ „gyorsan” számolható úgy akkor konstruálható $\mathcal{M}$ algoritmus, ami $O(T_p P + T_p T_s \log P)$ időben kiszámítja λ^*-t.*

3.4. Törtlineáris kombinatorikai optimalizálás

Ebben a szekcióban a Radzik által [Rad98]§2-ben ismertetett módon visszavezetjük a törtlineáris kombinatorikai optimalizálási feladatokat paraméteres keresésre, ezzel ilyen feladatokra általános megoldási módszert adva, amivel gyakran jó futásidők érhetők el.

Definíció. Adott $\mathcal{X} \subseteq \{0,1\}^p$ megengedett megoldások halmaza, és adottak $f : \mathcal{X} \to \mathbb{R}$ és $g : \mathcal{X} \to \mathbb{R}^+$ lineáris függvények. Ekkor **törtlineáris kombinatorikai optimalizálási feladatnak** nevezzük a következő $\mathcal{F}$ feladatot:

$$\mathcal{F}: \text{ keressük } \lambda^* = \max_{x \in \mathcal{X}} \frac{f(x)}{g(x)}\text{-t}$$

Ilyen típusú feladatok például a minimális súlyozott átlagú feszítőfa, a maximális súlyozott átlagú kör és a tört $0-1$ hátizsákfeladat.

3.4.1. Állítás. *A $\mathcal{F}$ maximalizálási feladat ekvivalens a következővel:*

$$\mathfrak{P} : \text{minimalizáljuk } \lambda \in \mathbb{R}{:} f(x) - \lambda g(x) \leq 0 \ (\forall x \in \mathcal{X})$$

Bizonyítás. A feltétel λ^*-re világos, hogy teljesül, hiszen $\frac{f(x)}{g(x)} \leq \lambda^* \ \forall x \in \mathcal{X} \implies f(x) - \lambda^* g(x) \leq 0$. Másrészt $\lambda < \lambda^*$-re $\exists x' \in \mathcal{X} : \frac{f(x)}{g(x)} > \lambda \implies f(x) - \lambda g(x) > 0$, tehát λ^* a minimális lambda ami teljesíti a feltételt és a megoldása $\mathcal{P}$-nek. □

Ezt felhasználva $\alpha(\lambda) = \begin{cases} \text{hamis} & \text{ha } \max\limits_{x \in \mathcal{X}}(f(x) - \lambda g(x)) < 0 \\ \text{igaz} & \text{különben} \end{cases}$ függvényre ez egy standard alakú paraméteres keresési feladat.

Törtlineáris kombinatorikai optimalizálási feladatnál a standard alak speciális esetét alkalmazva: ha $\mathfrak{P}$ kiszámítja $f(x) - \lambda^* g(x)$-et, akkor egyrészt tudjuk hogy λ^* definíciója szerint ez 0, másrészt megkapjuk mint λ^* lineáris függvénye, akkor meg tudjuk oldani az $a\lambda^* + b = 0$ egyenletet, hogy megkapjuk λ^*-t.

Megjegyzés. A bizonyításból következik az is, hogy az λ^*-hez tartozó x-re ami minimalizálja $f(x) - \lambda^* g(x)$-et arra $\frac{f(x)}{g(x)} = \lambda^*$ maximális, így az optimális struktúrát is megtudjuk találni.

3.4.1. Maximális súlyozott átlagú út DAG-ban

A következő Radzik által [Rad98]-ban tárgyalt egyszerű törtlineáris programozási feladatra alkalmazható a fenti módszer. Ez felfogható, mint legjobb össz jutalom-költség arányú út keresése.

3.7. Feladat. *Adott egy körmentes irányított gráf, csúcsainak $v_0, \cdots, v_{n-1}$ topologikus rendezése és élein f, g költségfüggvények, hogy $g > 0$. Mi az $\frac{f}{g}$ költségfüggvény szerinti legdrágább v_0-ból v_{n-1}-be vezető út költsége? (tegyük fel, hogy van út v_0-ból v_1-be)*

Az ebből kapott $\mathfrak{P}$ alakra hozott feladatot oldottunk meg 3.2.1-ban, így az abból kapott λ^* a keresett megoldás. Így a kapott futásidő: $O(nm \log n)$.

3.4.2. Kétfázisú feszítőfa

A következő Megiddo [Meg83a]§3 cikke alapján megoldott feladat a maximális súlyozott átlagú feszítőfa $\mathfrak{P}$ alakja, így a törtlineáris feladatot is megoldja a már látott módon.

3.8. Feladat. *Adott $G(V, E)$ gráf, egy k költségvetési korlát és $w_\lambda(e)$ az éleken λ-ban lineáris költségfüggvény: $w_\lambda(e) = f(e) + \lambda g(e)$ úgy, hogy $g > 0$. Keressük a maximális λ^*-ot, amire a w_{λ^*} szerinti minimális költségű feszítőfa költsége legfeljebb k.*

Megjegyzés. Ez a feladat felfogható úgy, hogy a feszítőfa egy e élének megépítéséhez szükséges anyagokat meg tudjuk venni $f(e)$ költséggel, viszont az építkezés csak az anyagok beszerzése után kezdődhet el, amikorra a munkások által e él megépítésére adott $g(e)$ árajánlata szorzódik egy jelenleg ismeretlen λ-val infláció miatt. Ekkor keressük a legnagyobb λ infláció miatti szorzót, amire k költségvetéssel megépíthető feszítőfa.

Ha $\mathcal{P}$-nek Boruvka/Sollin algoritmusát használjuk, és $\mathcal{S}$-nek Chazelle [Cha00]-ben leírt $O(m \cdot \alpha(m,n))$ futásidejű (ahol α az inverz Ackermann függvény) algoritmusát hasznlájuk, akkor $\mathcal{M}$ futásideje $O(T_p P + T_p T_s \log P) = O(\log n \cdot m + \log n(m \cdot \alpha(m,n)) \log n) = O(m \cdot \alpha(m,n) \log^2 n)$.

Megjegyzés. Ugyanígy kapunk ezzel a futásidővel algoritmust a minimális/maximális törtköltségű feszítőfára is.

3.4.3. Feladatütemezés

A következő Megiddo által [Meg83a]§4-ben tárgyalt feladatütemezési problémára adott megoldása is paraméteres keresést alkalmaz. Ezzel egy számítógépen elvégzendő időérzékeny független feladatokat tudunk ütemezni, a feladatok között minél nagyobb egyforma hosszúságó szüneteket hagyva a gép túlmelegedését minimalizálva.

3.9. Feladat. *Adott n feladat, amiket el akarunk végezni. A feladatok $t_1, \cdots, t_n$ ideig tartanak, halasztási költségeik pedig $c_1, \cdots, c_n$. Egymás után szeretnénk végrehajtani a feladatokat úgy, hogy két egymást követő feladat között λ hosszú szünetet iktatunk be. Ha az i indexű feladatot f_i-kor fejezzük be, ennek költsége $\sum_{i=1}^{n} c_i f_i$. Keressük a maximális λ^* szünethosszt, amit a k költségvetésünkkel megengedhetünk magunknak.*

Ha az $\pi(i)$ indexű feladatot i-edikként végezzük el, akkor az összköltség: $\sum_{i=1}^{n} c_i f_i = \sum_{i=1}^{n} \left(t_{\pi(i)} + \sum_{j=1}^{i-1} (t_{\pi(j)} + \lambda) \right) c_{\pi i}$.

Adott λ-ra $\frac{t_i+\lambda}{c_i}$ szerinti növekvő sorrendben éri meg a feladatokat végrehajtani:
$\pi(k)+1 = \pi(l)$, $\frac{t_k+\lambda}{c_k} > \frac{t_l+\lambda}{c_l}$ esetén a kettő feladat sorrendjét megcserélve a többi feladat befejezési időpontja nem változik, így az összköltség változása $c_k(t_l+\lambda) - c_l(t_k+\lambda) < 0$. Így ha nem $\frac{t_i+\lambda}{c_i}$ szerint növekvő sorrendbe vannak rendezve tudunk ilyen szomszédcserével javítani, tehát ez a rendezés lesz a legolcsóbb.

Ebből adódik, hogy $\mathcal{S}$ eszerint rendez $O(n \log n)$ időben, és kiszámítja a rendezés szerinti költséget még $O(n)$ időben, összesen $O(n \log n)$ futásidővel.

Másrészt, ha adott $\frac{t_i+\lambda^*}{c_i}$-k szerinti rendezés, abból következik a költség mint λ^* lineáris függvénye, szóval $\mathcal{P}$ rendezze ezeket a λ^*-ben lineáris függvényeket.

A kapott $\mathcal{M}$ algoritmus futásideje $O(T_p P + T_p T_s \log P) = O(\log n \cdot n + \log n \cdot n \log n \log n) = O(n \log^3 n)$.

3.4.4. k-szintű metszéspont

Definíció. Adottak $p_1, \cdots, p_n$ síkbeli pont. Ekkor ezeknek **középpontjának** nevezzük p_i-t, ha minden p_i-n átmenő egyenes által meghatározott két zárt félsíkban legalább $\left\lceil \frac{n}{3} \right\rceil$ van a pontok közül.

Matousek [Mat90]-ben ad egy algoritmust, ami n síkbeli pontnak $O(n \log^3 n)$ időben számítja ki egy középpontját. Matousek algoritmusában alfeladat a következő, amit a [Ili14]-ban Ilkin által leírtakhoz hasonlóan tárgyalunk.

3.10. Feladat. *Adottak $e_1, \cdots, e_n$ egyenesek a síkon, számítsuk ki azt a metszéspontjukat, aminek k-adik legkisebb az x koordinátája.*

Legyen λ^* az x koordináta értéke a keresett pontban. Legyen τ^* pedig a ponton átmenő függőleges egyenes.

Vegyük észre, hogy adott λ^* x koordinátából $O(n)$ időben számítható az összes egyenesnek az λ^* x koordinátájú pontja. Ezeket y koordináta szerint rendezve, majd a rendezés szerint sorban végigmenve a pontokon meg tudjuk állapítani $O(n \log n)$ időben, hogy melyik egyenesek metszéspontjainak x koordinátája λ^*. τ^* ismeretében tehát a keresett pont meghatározható $O(n \log n)$ időben.

A $\mathcal{P}$ algoritmus rendezze y koordináta szerint az egyenesek τ^*-val vett $p_1^{\tau^*}, p_2^{\tau^*}, \cdots, p_n^{\tau^*}$ metszéspontjait. Az előbb láttuk, hogy τ^* meghatározza a keresett pontot. $\mathcal{S}$ pedig $p_i^{\tau^*}$ és $p_j^{\tau^*}$ y koordinátáinak összehasonlítását végzi el. Ehhez megnézi, hogy e_i és e_j metszéspontjától jobbra, vagy balra van τ^*.

Az első metszésponttól balra lévő τ-ra a p_1^τ-k rendezése meggondolható, hogy az egyenesek meredeksége szerinti. Ezt a rendezést $O(n \log n)$-ben ki tudjuk előre számolni. Indexeljük eszerint a rendezés szerint a $p_1^*, p_2^*, \cdots, p_n^*$ pontokat. Legyen $p^\tau_{\pi_\tau(i)}$ a τ-val vett metszéspontja e_i-nek úgy, hogy $p^\tau_{\pi_\tau(j_1)} < p^\tau_{\pi_\tau(j_2)} < \cdots < \pi^\tau_{\tau(j_n)}$. Ekkor a π_τ permutáció inverziószáma τ-ban monoton nő. Ennek köszönhetően eldönthető az inverziószám kiszámításával, hogy egy adott metszéspont τ^*-tól milyen irányban van. Ezt $O(n \log n)$-ben meg tudjuk tenni a Halim, Halim és Effendy által [SE20]§2.2.2-ben bemutatott módosított mergesort algoritmussal.

Az így a kapott futásidő $O(T_p P + T_p T_s \log P) = O(n \log n + \log n \cdot n \log^2 n) = O(n \log^3 n)$.

3.5. Minimumszámítás

Ebben a szekcióban [Rad98]§5.5 tematikáját követve fogunk látni két gyorsítást, amiket a $\mathcal{P}$ algoritmusban történő minimumszámítási fázisokra tudunk használni. A minden csúcspár közötti legrövidebb út keresésére közismert [Cor+09]§VI.25.2-ben is megtalálható Floyd-Warshall algoritmus ilyen. Ezt kihasználva fogunk adni egy hatékony algoritmust minimális súlyozott átlagú kör keresésére.

3.5.1. Állítás. *Tegyük fel, hogy a $\mathcal{P}$ algoritmus egy fázisában az $f_1(\lambda^*), \cdots, f_n(\lambda^*)$ λ^*-ban lineáris paraméteres kifejezések minimumát kell kiszámítani. Paraméteres keresési algoritmusban ez a fázis $O\left(n + T_s \log n \log\log n\right))$ futásidővel szimulálható.*

Bizonyítás. Használjuk a 2.2.3-ben leírt minimumszámító algoritmust: ez $O(\log\log n)$ időben számít minimumot $O(n)$ összlépésszámmal. Így a minimumszámítási fázis futásideje a paraméteres keresés $\mathcal{M}$ algoritmusában:

$$O(W_p + T_p T_s \log P) = O\left(n + \log\log n \cdot T_s \log \frac{n}{\log\log n}\right) = \\ = O(n + \log\log n \cdot T_s(\log n_i - \log\log\log n)) = O(n + \log\log n \cdot T_s \log n)$$

□

3.5.2. Állítás. *Tegyük fel, hogy a $\mathcal{P}$ algoritmus egy fázisában az $f_1(\lambda^*), \cdots, f_n(\lambda^*)$ λ^*-ban lineáris paraméteres kifejezések minimumát kell kiszámítani. Paraméteres keresési algoritmusban ez a fázis $O\left((n + T_s) \log n\right)$ futásidővel.*

Bizonyítás. A minimumszámítási fázisban használjuk a 1.4-ben leírt algoritmust a pontonkénti minimum kiszámítására. Ezzel megkaptuk a töréspontokat sorbarendezve $O(n_i \log n_i)$ időben. Bináris kereséssel keressük meg a töréspontok között λ^*-t a standard alakban látott módon.
Ez egy minimumszámítási fázisra $O((n + T_s) \log n)$ futásidőt ad. □

Megjegyzés. Ha egy minimumszámítási fázisban több minimumot is számítunk úgy, hogy ezek $m_{i,1}, \cdots, m_{i,k(i)}$ méretű halmazok minimumai és $m_{i,1} + \cdots + m_{i,k(i)} \leq n_i$, akkor ugyanezekkel a futásidőkkel igazak az előbbi állítások:

- Az első esetében a processzorszám ugyanúgy n_i, viszont $\mathcal{P}$ futásideje $\log\log \max_j(m_{i,j}) < \log\log n_i$.
- A második állításnál a bináris keresés ugyanúgy történhet n_i ponton, és előtte a futásidő összesen $\sum_j m_{i,j} \log m_{i,j} < \sum_j m_{i,j} \log n_i = n_i \log n_i$.

3.5.1. Minimális súlyozott átlagú kör

A következő feladatra tudjuk alkalmazni a fenti technikát, az alábbi Radzik által [Rad98]-ból származó módon.

3.11. Feladat. *Adott egy $G(V, E)$ irányított gráf és élein f, g költségfüggvények, hogy $g > 0$. Mennyi a $\frac{f}{g}$ szerint minimális kör költsége?*

A 3.4-ban tárgyaltak alapján ez ekvivalens a

$$F(\lambda) = f - \lambda g \text{ költségfüggvény szerinti legolcsóbb kör költsége}$$

nullhelyének keresésével. Ez λ-ban monoton csökken, szóval alkalmazható a paraméteres keresés.

Legyen π_{ij}^k a legolcsóbb legfeljebb k él hosszú $i - j$ séta. $\pi_{ij}^{2k} = \min_r(\pi_{ir}^k + \pi_{rj}^k)$, tehát a következő algoritmust kapjuk:

Használjuk ezt mint $\mathcal{P}$ és mint $\mathcal{S}$ használjuk az $O(nm)$ futásidejű Bellman-Ford algoritmust negatív kör detektálására.

Algorithm 3.6: Legolcsóbb körséta kiszámítása

input : $s : V \times V \to \mathbb{R}$ súlyfüggvény, amire $s(i,i) = 0$ és ha nincs (i,j) él $s(i,j) = \infty$

parallel for $i, j \in \{1, \cdots, n\}$ **do**
  $\pi_{ij} \leftarrow s(i,j)$;
for $\lceil \log n \rceil$ *iterations* **do**
  parallel for $i, j \in \{1, \cdots, n\}$ **do**
    $\pi_{ij}^{2^k} \leftarrow \min_r \left(\pi_{ir}^{2^{k-1}} + \pi_{rj}^{2^{k-1}} \right)$;
$ret \leftarrow \min_{i,j}(\pi_{ij}^{2^{\lceil \log n \rceil}})$; // $O(\log \log n)$ időben $O(n^2)$ processzorral
return *ret*;

- 3.5.1-t összegezve a futásidő:

$$O(W_p + T_p T_s \log P) = O(\underbrace{n^2 + nm \log n \log \log n}_{\text{végén a minimum}} + \sum_{i=1}^{\lceil \log n \rceil} (n^3 + nm \log n \log \log n)) = O(n^3 \log n + nm \log^2 n \log \log n)$$

- 3.5.2-t összegezve pedig:

$$O(W_p + T_p T_s \log P) = O(\underbrace{n^2 + nm \log n \log \log n}_{\text{végén a minimum}} + \sum_{i=1}^{\lceil \log n \rceil} (n^3 + nm) \log n) = O(n^3 \log^2 n)$$

3.6. Cole gyorsítása

Cikkében [Col87] Cole megmutatja hogy ha $\mathcal{P}$ rendezés, akkor az AKS rendező háló futását manipulálva $O(\log n)$ faktorral jobb futásidőt lehet elérni az ebből származó paraméteres keresési algoritmusra. Ebben a fejezetben Cole technikáját és alkalmazásait mutatjuk be.

3.6.1. Rendező játék

Alíz és Béla egy játékot játszanak. Adott egy n vezetékes, $d(n)$ fázisú rendező hálózat. Alíz nem ismeri a hálózat bemenetében lévő számokat, viszont Béla igen. Minden körben Alíz megkéri Bélát, hogy végezzen el a hálózaton *megengedett* összehasonlításokat, Béla pedig ezeknek egy részét elvégzi és azok eredményeit elmondja Alíznak. Alíznak célja, hogy minél kevesebb kör elteltével megtudja, hogy mi a számok permutációja a hálózat kimeneténél.

Ahhoz hogy formálisan definiáljuk a játékot szükségünk lesz a következő definíciókra:

Definíció. A hálózat c_1 összehasonlítása **függ** a hálózat c_2 összehasonlításától, ha c_2 korábbi fázisban szerepel és kimenete eljuthat c_1 bemenetéhez. Másszóval ha elindulunk „jobbra” egy tetszőleges vezetéken, és minden összehasonlításnál dönthetünk, hogy átmegyünk a másik vezetékre vagy maradunk a jelenlegin, akkor van olyan döntéssorozat, aminél először a c_2 és c_1 összehasonlításokban is részt veszünnk.

Megjegyzés. Ez nem feltétlenül jelenti azt, hogy van is bemenet, amire egy szám először egyiken, utána a másikon megy keresztül: ha egy kész rendező hálózat után három külön fázisban elvégezzük egymás után a $c_1 = (v_1, v_2)$, $c_2 = (v_2, v_3)$, $c_3 = (v_3, v_4)$ összehasonlításokat, akkor c_3 definíció szerint múlik c_1-en, viszont mivel már rendezve vannak a számok nem cserél egyik sem, így c_1 kimenete nem befolyásolja c_3 kimenetét.

Definíció. Egy összehasonlítást **aktívnak** nevezünk, ha Alíz tudja, hogy a kezdeti számok közül melyikek vannak a bemenetén.

Megjegyzés. Ahhoz hogy egy c összehasonlítás aktív legyen, elég ha Alíz tranzitivitás miatt tudja az összes összehasonlítás eredményét, amitől c függ. Nem kell, hogy azokat mind Béla mondja el neki.

Ezekkel a játékot tudjuk formálisan definiálni:

3.12. Feladat. *Minden körben Alíz minden aktív összehasonlításhoz rendel egy w_i prioritást, és Bélának ezek közül legalább $\frac{\sum_{i=0}^{k} w_i}{2}$ prioritásértéknyinek el kell mondania a kimenetelét.*

3.6.1. Lemma. *Ha Alíz stratégiája az, hogy az i-edik fázisbeli aktív összehasonlításhoz 4^{-i} prioritást rendel, akkor a $k+1$-edik kör elején az összprioritás legfeljebb $\left(\frac{3}{4}\right)^k \cdot \frac{n}{2}$.*

Bizonyítás. Indukcióval látjuk be:

Az első kör elején legfeljebb $\frac{n}{2}$ aktív összehasonlítás van, és ezek súlyai mind 1-ek, így $k = 0$-ra teljesül az állítás.

Elég belátnunk, hogy minden lépésben az összprioritás legalább negyedével csökken.

Amikor egy w súlyú összehasonlítást Béla végrehajt, akkor ezzel legfeljebb kettő új aktív összehasonlítás keletkezik. Ezek legalább egy fázissal a végrehajtott után vannak, így legfeljebb $\frac{w}{4}$ prioritásúak darabonként. Az összprioritás ezzel legalább $\frac{w}{2}$-vel csökken. Mivel Béla az összprioritásnak legalább felét végrehajtotta, legalább negyedével csökkent. □

3.6.2. Állítás. *Ha $k \geq 5(i + \frac{\log n}{2})$, akkor a $k+1$-edik körtől nem lehet i-edik fázisbeli aktív összehasonlítás.*

Bizonyítás. A $k+1$-edik körtől a W összprioritásra $\left(\frac{3}{4}\right)^{5(i+\frac{\log n}{2})} \cdot \frac{n}{2}$ az előző lemma szerint.
$\left(\frac{3}{4}\right)^5 < \frac{1}{4}$, tehát $W < \left(\frac{1}{4}\right)^{(i+\frac{\log n}{2})} \cdot \frac{n}{2} = \left(\frac{1}{4}\right)^i \cdot \frac{1}{2}$, viszont i-edik fázisbeli összehasonlítás súlya $\left(\frac{1}{4}\right)^i$. □

3.6.2.1. Következmény. *$O(d(n) + \log n)$ kör alatt végetér a játék, hiszen az előző állítás szerint nem lesz aktív $d(n)$ szintű összehasonlítás $5(d(n) + \frac{\log n}{2})$ kör után.*

Megjegyzés. [Col87] cikkében Cole röviden tárgyalja a játékot tetszőleges körmentes irányított gráfra rendező háló helyett.

3.6.2. A gyorsítás

3.6.3. Tétel. *Ha paraméteres keresésnél $\mathcal{P}$ rendezési algoritmus, ami $O(k(n))$ a paraméterben lineáris kifejezéseket rendez és $\mathcal{S}$ futásideje a szokásos módon T_s, akkor $O(k(n)\log k(n) + T_s \log k(n))$ futásidejű $\mathcal{C}$ paraméteres keresési algoritmus konstruálható.*

Definíció. Ha adottak $a_1, \cdots, a_n$ számok és $s_1, \cdots, s_n$ súlyok, akkor az a_i-k s_i-k szerinti **súlyozott mediánja** az a_j, amennyiben a $s_j + \sum_{i:a_i<a_j} s_i \geq \frac{1}{2}\sum_{i=1}^{n} s_i$ és $s_j + \sum_{i:a_i>a_j} s_i \geq \frac{1}{2}\sum_{i=1}^{n} s_i$.

Bizonyítás. Játsszuk a fenti játékot az AKS rendező hálózaton úgy, hogy Béla szerepében mindig a kritikus pontok prioritások szerinti súlyozott mediánjára futtatjuk $\mathcal{S}$-t. Ezzel tranzitivitás miatt az összhasonlításoknak prioritás szerint legalább a fele eldől.

Reiser [Rei78]-ban bemutatott algoritmust használva a súlyozott medián $O(k(n))$ időben számítható minden körben. Mivel $O(\log a(n))$ kör van, és a rendezés $O(a(n)\log a(n))$ időt vesz igénybe, ezzel a szimuláció sebessége aszimptotikusan nem romlik, viszon minden körben elég $\mathcal{S}$-et egyszer meghívni. Ezzel a $O(T_pP + T_pT_s \log P)$-beli $O\log P$ faktor eltűnik, tehát $O(k(n)\log k(n) + \log k(n)(T_s + k(n))) = O(k(n)\log k(n) + T_s \log k(n))$, és a kívánt futásidőt kaptuk. □

Megjegyzés. [GP13]-ben Goodrich és Pszona mutatnak egy AKS mentes randomizált algoritmust, ami nagy valószínűséggel ugyanezzel a log faktorral történő javulást eredményezi, viszont gyakorlati feladatoknál hatékonyabb.

Hasonló gyorsulást akkor is el tudunk érni, ha $\mathcal{P}$ m párhuzamosan futó bináris keresés. Itt a játék ugyanúgy abból áll, hogy Alíz a bináris keresések által végrehajtandó összehasonlításokhoz rendel súlyokat. Itt mivel egy összehasonlításból mindig 1 db eggyel későbbi fázisú keletkezik, ha Alíz a 2^{1-j} prioritást rendeli i-edik fázisbeli összehasonlításokhoz, akkor a fenitvel analóg módon belátható:

3.6.4. Lemma. *A $k+1$-edik kör elején az összprioritás legfeljebb $\left(\frac{3}{4}\right)^k \cdot m$.*

3.6.5. Állítás. *Ha $k \geq 3(i + \log m)$, akkor a $k+1$-edik körtől nem lehet i-edik fázisbeli aktív összehasonlítás.*

3.6.5.1. Következmény. *$O(\log m + \log n)$ kör alatt végetér a játék.*

Tehát a paraméteres keresést a fentihez hasonló módon megváltoztatva:

3.6.6. Tétel. *Ha a $\mathcal{P}$ párhuzamos algoritmus m darab párhuzamos bináris keresésből áll, összehasonlításaiban a bemeneti paraméternek lineáris függvényei szerepelnek és $\mathcal{S}$ futásideje $T_s(n)$, akkor a párhuzamos keresési algoritmus futásideje:*

$$O(T_s(n)(\log m + \log n) + m(\log m + \log n)) = O((T_s(n) + m)(\log m + \log n))$$

3.6.3. Alkalmazások

Kétfázisú feszítőfa

Erre 3.4.2-ben láttunk egy $O(T_pP + T_pT_s \log P) = O(\log n \cdot m + \log n(m \cdot \alpha(m,n)) \log n)$ futásidejű megoldást. Vegyük észre, hogy ebben bár azt mondtuk hogy Boruvka/Sollin algoritmusát használjuk kívül, abban az élek súlyainak rendezése az igazi kulcs, ugyanis az már egyértelműen meghatározza hogy mi fog történni az algoritmus maradékában.

Elég tehát rendezni az élek $f(e) + \lambda^* g(e)$ súlyait $\mathcal{P}$-vel, szóval alkalmazható a gyorsítás. A javított futásidő $O(\log n \cdot m + \log n(m \cdot \alpha(m,n))) = O(m \cdot \alpha(m,n) \log n)$.

Feladatütemezés

3.4.3-ben erre $O(T_pP + T_pT_s \log P) = O(\log n \cdot n + \log n \cdot n \log n \log n)$ futásidejű algoritmust adtunk, amiben a $\mathcal{P}$ algoritmus n lineáris kifejezést rendezett, tehát $O(\log n \cdot n + \log n \cdot n \log n) = O(n \log^2 n)$ a javítással.

k-szintű metszéspont

A 3.4.4-ben látott $O(T_pP + T_pT_s \log P) = O(n \log n + \log n \cdot n \log^2 n) = O(n \log^3 n)$ futásidejű megoldás Cole gyorsításával együtt $O(n \log^2 n)$, hiszen az az utolsó $n \log^2 n$-ről lefarag egy $\log n$ szorzót.

Megjegyzés. Erre a feladatra az optimális futásidő $O(n \log n)$, amit [Col+89]-ban Cole és társai ezt módosítva egy approximációs ötlettel érnek el.

3.7. Másodrendű alkalmazás

A következő Megiddo által [Meg83a]§9-ban tárgyaltakat feldolgozó rész az egymásba ágyazott paraméteres keresési algoritmusok esetét mutatja be. Ebben az esetben a paraméteres keresési algoritmus párhuzamosításával megmutatjuk, hogy jobb futásidő érhető el.

Előfordul, hogy a $\mathcal{S}$ algoritmus már egy paraméteres keresésből kapott, azaz egymásba ágyazott paraméteres kereséseket futtatutnk. Ez geometriai feladatoknál a legjellemzőbb, viszont itt egy korábban látott feladatot felhasználó ilyen alkalmazást mutatok be:

3.13. Feladat. *Adott egy G gráf, és élein f, g, h súlyfüggvények, hogy $g, h > 0$. Keressük meg a maximális λ értéket, amire a $\frac{f+\lambda h}{g}$ költségfüggvény szerinti minimális kör költsége legfeljebb b.*

Ha negatív kört T_p időben tudunk detektálni P processzorral, akkor egy lépésben P kritikus érték között fogunk bináris keresni. Ehhez $O(\log P)$-szer kell futtatnunk a $\mathcal{S} = \mathcal{M}_1$ algoritmust, aminek futásideje $T_M = O(T_pP + T_pT_s \log P)$. Így a kapott futásidő: $O(T_pP + T_pT_M \log P) = O(T_pP + T_p(T_pP + T_pT_s \log P) \log P) = O(T_p^2 P \log P + T_p^2 T_s \log^2 P)$, ami a 3.5.1-ben látott $P = n^3, T_p = \log n \cdot \log\log n, T_s = nm$-el:

$$O((\log n \cdot \log\log n)^2 n^3 \log n + (\log n \cdot \log\log n)^2 nm \log^2 n) \leq O(n^3 \log^4 n (\log\log n)^2)$$

A paraméteres keresésben a 3.2 algoritmus helyett $\log n$-el szorozzuk meg a processzorszámot és minden $\frac{n}{\log n}$-edik kritikus értéketre értékeljünk ki párhuzamosan α-t a szekvenciális algoritmussal. Ezután bináris kereséssel a már kiértékelt $\log n$ értéken $O(\log \log n)$ szekvenciális időben a már kiszámított értékeken keresve le tudjuk szűkíteni a kiértékeletlen kritikus értékek intervallumát $\frac{n}{\log n}$ eleműre. Ezt ismégelgetve mindig $\log n$ értékre párhuzamosan futtatva a keresést $O(\frac{\log n}{\log \log n})$ iterációban végetér, így a kapott párhuzamos paraméteres keresési algoritmus párhuzamos futásideje $\tilde{T_p} = O\left(T_p \frac{\log P}{\log \log P}\right)$ úgy, hogy $\tilde{P} = O(P \log P)$ processzort használ. Ezzel $O\left(T_p P \frac{\log^2 P}{\log \log P} + T_p(\frac{\log P}{\log \log P}(T_s \log P + \log \log P))\right) = O\left(T_p P \frac{\log^2 P}{\log \log P} + T_p \frac{\log^2 P}{\log \log P} T_s\right)$, tehát ebben $T_p := \tilde{T_p}$ és $P := P \log P$-vel azt kapjuk, hogy:

$$O\left(T_p P \frac{\log^3 P}{\log \log P} + T_p \frac{\log^3 P}{\log \log P} T_s\right) = O\left(n^3 \log n \log \log n \frac{\log^3 n}{\log \log n} + \log n \log \log n \frac{\log^3 n}{\log \log n} nm\right) \leq O(n^3 \log^4 n)$$

Összefoglalás

A dolgozatban párhuzamos algoritmusok tárgyalását a Megiddo által, [Meg79] és [Meg83a] cikkeiben bevezetett paraméteres keresési módszerrel motiváltuk.

A paraméteres kereséshez és a dolgozatban szereplő alkalmazásaihoz szükséges ismereteket mutattunk be párhuzamos algoritmusokról. Többek között láttuk, hogy PRAM modellben hogyan lehet összegezni, minimumot keresni és minimális költségű feszítőfát konstruálni. A PRAM modell különböző változataira és ezek egymással történő szimulálására is kitértünk. Párhuzamos rendezéshez rendező hálós megoldást tárgyaltunk, hogy lefektessük a dolgozat végén látott, Cole [Col87] cikkében publikált gyorsításához szükséges alapokat.

A szakdolgozatban ismertettünk paraméteres kereséssel bizonyos feladatokra - többnyire egy szekvenciális és egy párhuzamos algoritmust felhasználva - konstruált hatékony szekvenciális algoritmusokat. Bevezettük a standard alakot, ami egységes keretrendszert adott bizonyos típusú paraméteres keresési algoritmusok bemutatására. Ezt felhasználva láttunk fogunk algoritmusokat mozgó pontok halmazának minimális átmérőjének kiszámítására és fa max-min k-partícionálására. Ezután rátértünk a standard alak egy könnyebben kezelhető speciális esetére, amivel törtlineáris kombinatorikai optimalizálási feladatokra mutattunk megoldásokat. A dolgozat végén gyakori esetekre mutattunk további technikákat, amikkel jobb futásidőket lehet elérni minimumszámításokat, illetve rendezést tartalmazó párhuzamos algoritmusokból kapott paraméteres keresési algoritmusokra.

Az érdeklődő olvasók figyelmébe ajánljuk a következő paraméteres keresést használó eredményeket, amiket terjedelmi korlátok miatt a dolgozatban nem tárgyaltunk:

- Katoh és társai különböző „térmegfigyelőkamera-elhelyezési" feladatokra alkalmazzák [Kat+10]-ban, például egy konvex sokszög kerületén két minimális sugarú L_1 normabeli kör elhelyezésére, hogy ezek fedjék a sokszöget $O(m^2 \log^2 m)$ időben, ahol m az oldalszám.
- Agarwal, Sharir és Toledo sokszögek Hausdorff távolságának kiszámítására adnak $O((mn)^2 \log^3(mn))$ futásidejű algoritmust, amiben n és m az alakzatok élszámai [AST94]-ban.
- Agarwal, Sharir és Toledo [AST94] cikkjükben egy adott sokszögben benne lévő leghosszabb szakasz hosszának kiszámítására $O(n^{8/5+\epsilon})$ idejű algoritmust is adnak.
- Alt és Godau két törtvonal Fréchet távolságának kiszámításéra adnak $O(nm \log(nm))$ futásidejű algoritmust [AG95]-ben.
- Megiddo [Meg83b] cikkében n darab $\mathbb{R}^2$-beli pontra a súlyozott össztávolságot minimalizáló pontot $O(n \log^3 n (\log \log n)^2)$ időben megtaláló algoritmust mutat.
- Agarwal és Matousek [AM93] cikkükben egy félegyenest metsző első objektumot gyorsan megkereső adatstruktúra létrehozásához használnak paraméteres keresést.

Irodalomjegyzék

[AST94] P.K. Agarwal, M. Sharir és S. Toledo. "Applications of Parametric Searching in Geometric Optimization". *Journal of Algorithms* 17.3 (1994), 292–318. old. ISSN: 0196-6774. DOI: https://doi.org/10.1006/jagm.1994.1038. URL: https://www.sciencedirect.com/science/article/pii/S0196677484710388.

[AM93] Pankaj K. Agarwal és Jirí Matousek. "Ray Shooting and Parametric Search". *SIAM J. Comput.* 22 (1993), 794–806. old. URL: https://users.cs.duke.edu/~pankaj/publications/papers/query-ps.pdf.

[AKS83] Miklós Ajtai, János Komlós és Endre Szemerédi. "An O(n log n) Sorting Network". *Combinatorica* 3 (1983. jan.), 1–9. old. DOI: 10.1145/800061.808726.

[AG95] Helmut Alt és MICHAEL GODAU. "Computing the Fréchet Distance between Two Polygonal Curves". *Int. J. Comput. Geometry Appl.* 5 (1995. márc.), 75–91. old. DOI: 10.1142/S0218195995000064.

[Bek15] Bekbolatov. *Odd-even mergesort implementation*. 2015. URL: https://gist.github.com/Bekbolatov/c8e42f5fcaa36db38402.

[Cha00] Bernard Chazelle. "A Minimum Spanning Tree Algorithm with Inverse-Ackermann Type Complexity". *J. ACM* 47.6 (2000. nov.), 1028–1047. old. ISSN: 0004-5411. DOI: 10.1145/355541.355562. URL: https://doi.org/10.1145/355541.355562.

[CHL01] Ka Wong Chong, Yijie Han és Tak Wah Lam. "Concurrent Threads and Optimal Parallel Minimum Spanning Trees Algorithm". *J. ACM* 48.2 (2001. márc.), 297–323. old. ISSN: 0004-5411. DOI: 10.1145/375827.375847. URL: https://doi.org/10.1145/375827.375847.

[Col87] Richard Cole. "Slowing down Sorting Networks to Obtain Faster Sorting Algorithms". *J. ACM* 34.1 (1987. jan.), 200–208. old. ISSN: 0004-5411. DOI: 10.1145/7531.7537. URL: https://doi.org/10.1145/7531.7537.

[Col+89] Richard Cole és tsai. "Optimal-time algorithm for slope selection". English (US). *SIAM Journal on Computing* 18.4 (1989), 792–810. old. ISSN: 0097-5397. DOI: https://doi.org/10.1137/0218055.

[Cor+01] Thomas H. Cormen és tsai. *Introduction to Algorithms*. 2nd. MIT Press, 2001, 704–724. old. ISBN: 0262032937.

[Cor+09] Thomas H. Cormen és tsai. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.

[FZ17] Greg N. Frederickson és Samson Zhou. "Optimal Parametric Search for Path and Tree Partitioning". *ArXiv* abs/1711.00599 (2017). URL: https://arxiv.org/pdf/1711.00599.pdf.

[GP13] Michael T. Goodrich és Pawel Pszona. "Cole's Parametric Search Technique Made Practical". *CoRR* abs/1306.3000 (2013). arXiv: 1306.3000. URL: http://arxiv.org/abs/1306.3000.

[Ili14] Ivaylo Ilinkin. "Experimental Evaluation of Parametric Search Algorithms". (2014). URL: https://www.semanticscholar.org/paper/Experimental-Evaluation-of-Parametric-Search-Ilinkin/45153bd53aedbf8274935adab32ab40368020fe7.

[JáJ92] Joseph JáJá. *An Introduction to Parallel Algorithms*. USA: Addison Wesley Longman Publishing Co., Inc., 1992. ISBN: 0201548569.

[Kat+10] Naoki Katoh és tsai. "Parametric search: three new applications". *Frontiers of Mathematics in China* 5.1 (2010. jan.), 65–73. old. ISSN: 1673-3576. DOI: 10.1007/s11464-009-0049-x. URL: https://doi.org/10.1007/s11464-009-0049-x.

[Lan18] Hans Werner Lang. *Odd-even mergesort*. 2018. jún. URL: https://www.inf.hs-flensburg.de/lang/algorithmen/sortieren/networks/oemen.htm.

[Mat90] Jirí Matousek. "Computing the Center of Planar Point Sets". (1990). URL: http://www.ams.org/books/dimacs/006/.

[Meg79] Nimrod Megiddo. "Combinatorial Optimization with Rational Objective Functions". *Mathematics of Operations Research* 4.4 (1979), 414–424. old. ISSN: 0364765X, 15265471. URL: http://www.jstor.org/stable/3689226.

[Meg83a] Nimrod Megiddo. "Applying Parallel Computation Algorithms in the Design of Serial Algorithms". *J. ACM* 30.4 (1983. okt.), 852–865. old. ISSN: 0004-5411. DOI: 10.1145/2157.322410. URL: https://doi.org/10.1145/2157.322410.

[Meg83b] Nimrod Megiddo. "The Weighted Euclidean 1-Center Problem". *Math. Oper. Res.* 8 (1983), 498–504. old. URL: http://theory.stanford.edu/~megiddo/pdf/weight1.pdf.

[OV02] René van Oostrum és Remco C. Veltkamp. "Parametric Search Made Practical". SCG '02 (2002), 1–9. old. DOI: 10.1145/513400.513401. URL: https://doi.org/10.1145/513400.513401.

[Rad98] Tomasz Radzik. "Fractional Combinatorial Optimization". *Handbook of Combinatorial Optimization: Volume1–3*. Szerk. Ding-Zhu Du és Panos M. Pardalos. Boston, MA: Springer US, 1998, 429–478. old. ISBN: 978-1-4613-0303-9. DOI: 10.1007/978-1-4613-0303-9_6. URL: https://doi.org/10.1007/978-1-4613-0303-9_6.

[Rei78] Angelika Reiser. "A linear selection algorithm for sets of elements with weights". *Information Processing Letters* 7.3 (1978), 159–162. old. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(78)90083-2. URL: https://www.sciencedirect.com/science/article/pii/0020019078900832.

[Smi02] Michiel Smid. *Solving geometric optimization problems using parametric search*. 2002. nov. URL: https://people.scs.carleton.ca/~michiel/lecturenotes/ALGGEOM/paramsearch.pdf.

[Smi03] Michiel Smid. *Computing the diameter of a point set: sequential and parallel algorithms*. 2003. nov. URL: https://people.scs.carleton.ca/~michiel/lecturenotes/ALGGEOM/diameter.pdf.

[SE20] Felix Halim Steven Halim és Suhendry Effendy. *Competitive Programming 4: The New Lower Bound of Programming Contests in the 2020s*. 2020. dec. ISBN: 9781716745522.

[Wu90] Michael Wu. "An O(log n) Time Common CRCW Pram Algorithm for Minimum Spanning Tree". (1990. okt.), 12. old. URL: https://www.ideals.illinois.edu/bitstream/handle/2142/74240/B37-ACT_114.pdf?sequence=2&isAllowed=y.