Algoritmuselmélet

Király Zoltán
ELTE TTK Matematikai Intézet
Számítógéptudományi Tanszék

2026. október 22.

1.93 verzió

Eötvös Loránd Tudományegyetem
Természettudományi Kar

Typotex

2014

© 2014–2021, Király Zoltán, Eötvös Loránd Tudományegyetem, Természettudományi Kar

Lektorálta: ifj. Katona Gyula

Készült a TÁMOP-4.1.2-08/2/A/KMR-2009-0045 számú,
„Jegyzetek és példatárak a matematika egyetemi oktatásához” című projekt keretében,
a Typotex Kiadó gondozásában.

Legfrissebb, on-line verzió:
https://zkiraly.web.elte.hu/Algoritmusok.pdf

Tartalomjegyzék

I. Alapvető algoritmusok 8

1. Bevezetés 9
- 1.1. Algoritmus fogalma, modellezés . . . 9
- 1.2. Barkochba . . . 10
- 1.3. Barkochba előre leírt kérdésekkel . . . 11

2. Keresés és Rendezés 12
- 2.1. Legnagyobb elem . . . 12
- 2.2. Egyszerre a legnagyobb és a legkisebb elem . . . 14
- 2.3. Két legnagyobb elem . . . 15
- 2.4. Felező (bináris) keresés . . . 15
- 2.5. Egyszerű rendezési algoritmusok összehasonlításokkal . . . 17
- 2.6. Adatstruktúrák – bevezetés . . . 19
- 2.7. Kupac . . . 19
  - 2.7.1. Műveletek a kupac adatszerkezetben . . . 21
  - 2.7.2. Kupacos rendezés . . . 22
- 2.8. Mediáns keresés . . . 23
- 2.9. Gyorsrendezés (Quicksort) . . . 24
- 2.10. Kulcsmanipulációs rendezések . . . 25
  - 2.10.1. Leszámláló rendezés . . . 25
  - 2.10.2. Számjegyes rendezés (1887) . . . 26

3. Aritmetika: Számolás nagy számokkal 28
- 3.1. Alapműveletek, hatványozás . . . 28
- 3.2. Mod m számolás . . . 29
- 3.3. Euklideszi algoritmus . . . 30
- 3.4. Kiterjesztett euklideszi algoritmus . . . 30
- 3.5. Mod m osztás . . . 31
- 3.6. Kínai maradéktétel . . . 31

4. Dinamikus programozás 32
- 4.1. Fibonacci számok . . . 32
- 4.2. Hátizsák-feladat . . . 33
- 4.3. Mátrix-szorzás zárójelezése . . . 34

5. Adatszerkezetek 2. **35**
5.1. Láncolt listák . . . 35
5.2. Bináris keresőfa . . . 35
5.3. AVL-fa . . . 37
5.4. Sorok, vermek, gráfok . . . 39

6. Alapvető gráfalgoritmusok **42**
6.1. Szélességi keresés/bejárás . . . 42
6.1.1. Összefüggőség tesztelése szélességi kereséssel . . . 43
6.1.2. Komponensek meghatározása . . . 44
6.1.3. Legrövidebb utak . . . 44
6.1.4. Kétszínezés . . . 45
6.1.5. Erős összefüggőség . . . 46
6.2. Prim algoritmusa . . . 46
6.3. Kupacok alkalmazása a Prim-algoritmusban . . . 48
6.3.1. d-ed fokú kupac . . . 48

7. Legrövidebb utak **50**
7.1. Csúcs-súlyozott eset . . . 50
7.2. Dijkstra algoritmusa . . . 50
7.3. Alkalmazás: legbiztonságosabb út . . . 51
7.4. Alkalmazás: legszélesebb út . . . 52
7.5. Házépítés – PERT módszer . . . 52
7.5.1. Topologikus sorrend . . . 52
7.5.2. PERT módszer . . . 53
7.6. Bellman–Ford-algoritmus . . . 55
7.7. Floyd–Warshall-algoritmus . . . 56
7.8. Johnson algoritmusa . . . 57

8. Hashelés **58**
8.1. Vödrös vagy láncolt hashelés . . . 58
8.2. Nyílt címzés . . . 58
8.3. Univerzális hash függvény család . . . 59
8.4. Kakukk hashelés . . . 60

9. Párosítások **61**
9.1. Stabil párosítás páros gráfban . . . 61
9.2. Maximális párosítás páros gráfban . . . 62
9.3. Párosítás nem páros gráfban . . . 64

10. Hálózati folyamok **65**
10.1. Folyam-algoritmusok . . . 66
10.2. Redukált folyam, folyam felbontása . . . 67
10.3. Menger tételei, többszörös összefüggőség . . . 67

11. Adattömörítés **68**

12. A bonyolultságelmélet alapjai **69**

II. Következő lépés **72**

13. Aritmetika: Számolás nagy számokkal **73**
13.1. Nagy számok szorzása Karacuba [1960] módszerével . . . 73
13.2. A diszkrét Fourier-transzformált . . . 74
13.3. Nagy számok szorzása (Schönhage-Strassen) . . . 77

14. Dinamikus programozás **79**
14.1. Optimális bináris keresőfa . . . 79

15. Mélységi keresés/bejárás és alkalmazásai **81**
15.1. Erősen összefüggővé irányítás . . . 82
15.2. 2-összefüggőség tesztelése . . . 82
15.3. Erősen összefüggő komponensek . . . 83

16. Legrövidebb utak **84**
16.1. Dijkstra algoritmusának további alkalmazásai . . . 84
16.1.1. Legszélesebb, legolcsóbb vagy legprofitábilisabb legrövidebb út . 84
16.1.2. Absztrakt Dijkstra . . . 85
16.1.3. Időfüggő legrövidebb út . . . 88
16.2. Legrövidebb út keresése konzervatív súlyozású irányítatlan gráfban . . . 88
16.3. Minimális átlagú kör keresése . . . 89
16.4. Suurballe algoritmusa . . . 89

17. Párosítások **91**
17.1. A Hopcroft–Karp-algoritmus . . . 91
17.2. Kuhn magyar módszere és Egerváry tétele . . . 92
17.3. Edmonds algoritmusának vázlata . . . 93

18. Hálózati folyamok **94**
18.1. Dinic algoritmusa . . . 94
18.2. Diszjunkt utak . . . 94
18.3. Többtermékes folyamok . . . 95

19. Hálózati kódolás **97**

20. Közelítő algoritmusok **100**
20.1. Definíciók . . . 100
20.2. Metrikus utazó ügynök . . . 101
20.3. Steiner-fa . . . 102
20.4. FPTAS a hátizsák-feladatra . . . 102
20.5. Maximális stabil házasítás . . . 104
20.6. Halmazfedés . . . 105
20.7. Beck–Fiala-tétel . . . 106

21. Fix paraméteres algoritmusok **108**
21.1. Steiner-fa 108
21.2. Lefogó csúcshalmaz 109
21.2.1. Első megoldás k-korlátos mélységű döntési fával 109
21.2.2. Második megoldás k^2 méretű kernellel 109
21.2.3. További megoldások korona-redukcióval 110
21.2.4. További javítások 112
21.3. Egzakt út 112

III. Függelék **113**

22. Pszeudokód **114**

23. Példák **116**

24. Források, ajánlott irodalom **124**

Előszó

Ez a jegyzet elsősorban matematikus és informatikus egyetemi hallgatók számára készült. A cél a „jegyzet” szó hagyományos értelmével összhangban nem a teljesség, sem a nagyon részletes magyarázat, hanem a tömörség volt. Ennek fő oka, hogy a témát a számos kiváló magyar nyelvű irodalom (lásd a 24. fejezetben) együttesen meglehetős teljességgel lefedi, így minden fejezethez hozzá lehet olvasni a részleteket akár magyar nyelven is.

A diákok legtöbbször úgy használják az ilyen jegyzetet, hogy egyoldalasan kinyomtatva elhozzák az előadásra, és ezek után meg tudják spórolni a tábláról való másolás nagy részét, és jobban tudnak koncentrálni az elhangzott bizonyításokra és magyarázatokra, illetve ezek lejegyzésére. Ez a rövid jegyzet kb. 40 darab 45 perces előadás vázlatát tartalmazza, így önálló tanulásra nem igazán alkalmas, a megértéshez fontosak az előadáson elhangzott magyarázatok, részletek és bizonyítások is.

Az első rész az Eötvös Loránd Tudományegyetemen a Matematikai Elemző szakon a „Gráfok és Algoritmusok Elmélete” című tárgy beindításakor tartott előadásaim alapján készült, az első szövegváltozatot begépelte Mátyásfalvi György, valamint Antal Éva és Cseh Péter 2008-ban, akiknek ezért roppant hálás vagyok. Ebben a részben sorszámozva szerepelnek azok a definíciók, állítások, tételek és algoritmusok, amelyek pontos kimondása (és az algoritmusok lépésszáma is) a kollokvium teljesítésének elengedhetetlen feltétele.

A második részben a Matematika és az Alkalmazott Matematika MSc program közös „Algoritmuselmélet” című törzsanyag tárgyának jelentős részéhez találhatók jegyzetek. Itt sokszor csak az algoritmus ötletét adjuk meg, a részletek csak az előadáson hangzanak el. Ám némi rutinnal ezekből az ötletekből egy elegendően sok előismerettel rendelkező MSc-s hallgató önállóan is ki tudja fejteni a részletes algoritmusokat és be tudja bizonyítani a kimondott állításokat és tételeket.

A harmadik rész függelék, a jegyzetben használt pszeudokód-formátum magyarázatát és az alapvető algoritmusokra néhány jól követhető példát tartalmaz.

Budakeszi, 2013. január Király Zoltán

Első rész

Alapvető algoritmusok

1. fejezet

Bevezetés

1.1. Algoritmus fogalma, modellezés

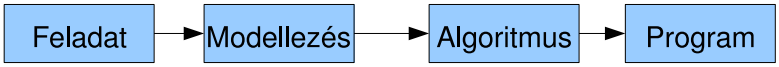
Ez a jegyzet algoritmusokkal foglalkozik, azonban az algoritmus fogalmát pontosan nem definiáljuk, mert nincs jól megfogalmazható, mindenki által elfogadott definíció. Az algoritmus szó intuitív jelentése „feladatmegoldási módszer”. A legelterjedtebb precíz definíció egy konkrét elméleti számítógép-modellt, a Turing gépet használja, lásd ezzel kapcsolatban a 12. fejezetet. Algoritmuson mi mindig egy precízen definiált eljárást fogunk érteni, amely egy rögzített probléma-típus egy adott feladatát már automatikusan és helyesen megoldja.

Egy algoritmus minden bemenetre (továbbiakban sokszor *input*) kiszámol egy kimenetet (továbbiakban általában *output*).

Ha egy adott problémához algoritmust szeretnénk tervezni, ennek első lépése a megfelelő modell megalkotása (melyben már egzakt matematikai fogalmakkal le tudjuk írni a feladatot), ebben a tárgyban ezzel a fontos résszel nem foglalkozunk, ezt a diszciplínát külön egyetemi kurzusok tárgyalják. Azonban elég sokat foglalkozunk majd gráfokkal, melyek az egyik általános modellezési eszköznek is tekinthetők.

Ha már megterveztük az algoritmusunkat, akkor persze azt szeretnénk, hogy az adott bemenetekre egy számítógép végezze el a számításokat, ezért az algoritmus alapján elő kell állítanunk egy számítógép-programot. Ezzel a résszel szintén számos kurzus foglalkozik az egyetemen.

Rövidebb algoritmusokat, illetve az algoritmusok vázát sokszor szövegesen adunk meg. Ennek több hátránya is van, egyrészt az így megadott vázból nehezebb a programot elkészíteni, másrészt sokszor lényeges részletek kimaradnak belőle, harmadrészt kicsit is bonyolultabb algoritmusok ilyen leírása hosszadalmas és sokszor zavaros is lehet. Ezért szokás használni (és mi is főleg ezt fogjuk) az ún. pszeudokódot, mely az algoritmusok leírásának elfogadott tömör leíró nyelve, lásd a 22. fejezetben.



Egy feladatnál azt az algoritmust szeretnénk megtalálni, amely a „legrosszabb esetet” a „legrövidebb futási idő” alatt oldja meg. Az algoritmus futási ideje egy bizonyos bemenetre a végrehajtott alapműveletek vagy „lépések” száma. Kényelmes úgy definiálni a lépést, hogy minél inkább gép-független legyen. Az egyszerűség kedvéért fogadjuk el, hogy az algoritmusok leírásához használt „pszeudokód” mindegyik sorának egyszeri végrehajtásához állandó mennyiségű idő szükséges. Lehet, hogy az egyik sor tovább tart, mint a másik, de feltesszük, hogy az i-edik sor minden végrehajtása c_i ideig tart, ahol c_i állandó (azaz független az input hosszától, a továbbiakban erre a konstans szót használjuk). Az egyes algoritmusok futási idejének meghatározásához tehát elsősorban az egyes utasítások végrehajtásainak számát határozzuk meg, majd ezeket hozzuk tömörebb és egyszerűbben kezelhető alakra.

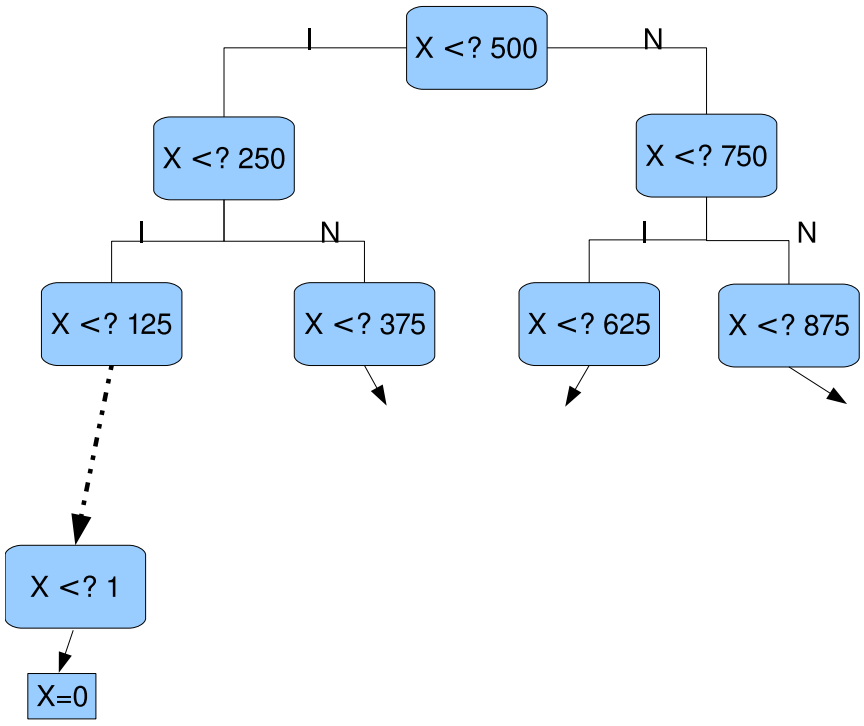
1. Definíció. *Egy A algoritmus lépésszáma a következő $\mathbb{N} \to \mathbb{N}$ függvény:*
$t_A(n) := \max_{\{x:\,|x|=n\}}\{Az\ A\ algoritmus\ hány\ lépést\ tesz\ az\ x\ inputon\}$.
Az input általában egy bitsorozat: $x \in \{0,1\}^n$, ennek hossza: $|x| = n$.

2. Definíció. *Legyenek f és $g: \mathbb{N} \to \mathbb{R}_0^+$ függvények. Azt mondjuk, hogy $f = O(g)$ (ejtsd: nagy ordó g), ha léteznek $n_0, c \in \mathbb{N}$ konstansok, hogy minden $n \geq n_0$ egészre $f(n) \leq c \cdot g(n)$.*

1.2. Barkochba

Feladat: 0 és 999 közötti szám kitalálása.

A következő fa jól ábrázol egy algoritmust arra, hogy hogyan tegyük fel a kérdéseket, hogy minél kevesebb kérdésből kitaláljuk a gondolt számot.



Az így kapott fa leveleire konkrét számokat írunk. A fa mélysége 10 (ez a gyökértől egy legalsó levélig megtett út hossza), ezért 10 kérdés mindig elég. Ennél kevesebb lépés nem lehet elég, mivel 9 kérdésnél a levelek száma csak maximum $2^9 = 512$ lehet. Ezt lehet általánosítani: ha 0 és $(n-1)$ közötti szám (dolog) közül kell egyet kitalálni, akkor $\lceil \log(n) \rceil$ (az alap nélküli log végig a jegyzetben **kettes alapú logaritmust** jelent) lépésből lehet kitalálni, és ennyi kérdés kell is.

1. Algoritmus. (a és b közötti egész szám kibarkochbázására.)

Ha $a > b$, akkor ilyen szám nincs, ha $a = b$, akkor gondolt szám a.

Különben legyen $c = a + \lfloor \frac{b-a}{2} \rfloor$.

Tegyük fel azt a kérdést, hogy a gondolt szám kisebb-egyenlő-e c-nél?

Ha a válasz igen, rekurzívan használjuk ezt az algoritmust, most már csak egy a és c közötti szám kitalálására.

Ha a válasz nem, akkor is rekurzívan használjuk ugyanezt az algoritmust, most azonban egy $c+1$ és b közötti szám kitalálására.

Lépésszám: $O(\lceil \log n \rceil)$

1.3. Barkochba előre leírt kérdésekkel

Feladat: szám (dolog) kitalálása nem adaptív kérdések esetén. Ekkor vajon hány kérdés szükséges?

Azaz itt előre fel kell tenni az összes kérdést. A „gondoló” n-féle dologra gondolhat, azt állítjuk, hogy ekkor is elég $k = \lceil \log n \rceil$ kérdés. Megkérjük a „gondoló” játékost, hogy a lehetséges dolgokat sorszámozza be valahogyan (pl. a lexikografikus sorrend szerint) 0-tól $(n-1)$-ig, és a gondolt dolog sorszámát írja fel kettes számrendszerben pontosan k jegyű számként.

Az algoritmus egyszerűen az, hogy az i. kérdésben megkérdezzük, hogy a sorszám i. bitje 1-e?

Lépésszám: $O(\lceil \log n \rceil)$

1. Állítás. *Barkochba játékban n dolog közül egy kitalálásához elég $\lceil \log n \rceil$ kérdés, de ennyi kell is. Ennyi kérdés akkor is elég, ha a kérdéseket előre le kell írni, azaz egy későbbi kérdés nem függhet az addigi válaszoktól.*

2. fejezet

Keresés és Rendezés

3. Definíció. *Rendezési feladat:*

Input: $a_1, \ldots, a_n \in U$, ahol U (az univerzum) egy tetszőleges rendezett halmaz.

Output: az indexek $i_1, \ldots, i_n$ permutációja, amelyre igaz, hogy $a_{i_1} \leq a_{i_2} \leq \ldots \leq a_{i_n}$.

Keresési/kiválasztási feladat:

Input: Az n (különböző) számból álló $A = \{a_1, a_2, \ldots, a_n\}$ halmaz és egy k szám, amelyre fennáll, hogy $1 \leq k \leq n$.

Output: egy i_k index, melyre teljesül, hogy az A halmazban pontosan $k-1$ elem kisebb, mint a_{i_k}.

Keresési, kiválasztási illetve rendezési feladatra megoldást adó algoritmust többféle-lét is megvizsgálunk. A kiválasztási feladat mindig megoldható úgy is, hogy először rendezünk, utána kiíratjuk a megfelelő elemet, de mi mindig ennél gyorsabb megoldást keresünk.

Ezeknél a feladatoknál az univerzum sokféle lehet, és néha két elem összehasonlítása lassabb, mint pl. egy értékadási/adatmozgatási művelet. Ezért külön megszámoljuk, hogy hány összehasonlítási lépést tettünk, majd külön megadjuk az összes lépések számát, feltételezve, hogy azért egy összehasonlítást konstans sok lépéssel meg lehet valósítani. Általában az összes lépések száma csak konstans-szorosa az összehasonlítások számának.

2.1. Legnagyobb elem

Feladat: Keressük meg, hogy melyik elem a legnagyobb az adott inputban, melyről feltesszük, hogy az $A[1:n]$ tömbben adott, tehát $A(i) = a_i$.

Megjegyzés: Ha egy eljárás inputjaként azt írjuk, hogy $A[1:n]$, akkor feltesszük, hogy az eljárás az n számot is megkapja. Sőt, ha A egy n elemű tömb, de az eljárásunk ennek csak az p. indexű elemtől kezdve a q. indexűig terjedő részén fog dolgozni, akkor inputként azt írjuk, hogy $A[p:q]$, azonban ez alatt értjük, hogy az eljárás igazából megkapja az A azonosítót, valamint az n, a p és a q számokat is.

4 algoritmust adunk, melyekből az első kettő valamint a második kettő lefutása lényegében ugyanaz. Az első algoritmusban sorban minden i-re megkeressük a maximumát az $\{a_1, \ldots, a_i\}$ halmaznak.

2. Algoritmus (MAX-1).

MAX-1($A[1:n]$):
  MAXINDEX:= 1 /* $i=1$-re a_1 *a maximum.*
  for $i=2..n$ /* *Sorra vizsgáljuk a többi elemet.*
    if $A(i) > A$(MAXINDEX) **then** MAXINDEX:= i /* *Amennyiben $A(i)$ nagyobb az eddigi maximumnál, akkor lecseréljük MAXINDEX-et i-re.*
  return($MAXINDEX$)

Összehasonlítások száma: $n-1$. Lépésszám: $O(n)$

Az összehasonlítások száma pontosan $n-1$. Az értékadások száma legfeljebb n. Összesen tehát a lépésszám $O(n)$.

2. Állítás. *A legnagyobb elem megtalálásához elég $n-1$ összehasonlítás, és legalább ennyi kell is.*

Ennél kevesebb összehasonlítással nem lehet megtalálni a legnagyobb elemet. Ugyanis ha legfeljebb $n-2$ összehasonlítást végzünk, akkor legfeljebb $n-2$ elemről derülhet ki, hogy valamely másiknál kisebb. Tehát legalább két olyan elem lesz, amelyikről ez nem derül ki, azaz soha senkinél nem voltak kisebbek. De akkor akármelyikük lehet a legnagyobb elem, hiszen a kettőből bármelyiket átcserélhetjük egy nagyon nagy számra, és az algoritmus ugyanúgy fog lefutni. Így azt is megállapíthatjuk, hogy ez az algoritmus egyben optimális is.

A következő algoritmusunk rekurzív. Ha $n>1$, akkor meghívjuk az eljárást az első $n-1$ elemre, majd a visszakapott indexű elemet összehasonlítjuk az n. elemmel. Kicsit általánosabban írjuk fel, $1 \le a \le b \le n$ esetén az input tömb a. és b. helye közötti részben keressük a maximumot (az eredeti feladatot nyilván az $a=1$, $b=n$ esetén oldjuk meg).

3. Algoritmus (MAX-2). Meghívjuk MAX-2($A[1:n]$)-et

MAX-2($A[a:b]$):
  if $a=b$ **then return**(a)
  j :=MAX-2($A[a:b-1]$)
  if $A(j) \ge A(b)$ **then return**(j)
    else return(b)

Vegyük észre, hogy a MAX-1 és MAX-2 algoritmusok során pontosan ugyanazokat az összehasonlításokat végezzük el, ráadásul ugyanabban a sorrendben. Itt az első felírás rövidebb és áttekinthetőbb, de ez sokszor fordítva van, a rekurzív leírás a kompaktabb.

Következő algoritmusunk során a „győztest” a kieséses sportversenyek mintájára keressük meg. Az egyszerűbb felírás miatt feltesszük, hogy $n=2^k$ kettő-hatvány, így elkerüljük az egészrészek használatát.

4. Algoritmus (MAX-3).

MAX-3($A[1:2^k]$):
  for $i = 1..2^k$ $B(i) := i$ /* *az indexeket tartalmazó B tömb inicializálása*
  for $j = 0..k-1$
    for $i = 2^j..2^k - 2^j$ $(+2^{j+1})$
      if $A(B(i)) \geq A(B(i+2^j))$ **then**
        CSERE$(B(i+2^j), B(i))$ /* $B(i+2^j)$*-be kerül a nagyobbik (azaz a győztes) eredeti indexe*
  return$(B(2^k))$

A külső ciklus első lefutása ($j = 0$) után a B tömbben a páros indexű helyeken lesznek az első forduló győzteseinek sorszámai, második lefutása ($j = 1$) után a 4-gyel osztható indexű helyeken lesznek az második forduló győzteseinek sorszámai, s.í.t. Az összehasonlítások száma $2^{k-1} + 2^{k-2} + \ldots + 1 = 2^k - 1$.

Ugyanez rekurzív eljárás formájában már áttekinthetőbb. Érdemes belegondolni, hogy ez a változat akkor is helyes, ha n nem kettő-hatvány. Világos, hogy már minden n értékre jól működik az algoritmus és pontosan $n-1$ összehasonlítást végez. A rekurzió „mélysége" legfeljebb $\lceil \log n \rceil$. Ezt a fajta hozzáállást szokás „oszd meg és uralkodj" módszernek is nevezni.

5. Algoritmus (MAX-4).

MAX-4($A[a:b]$):
  if $a = b$ **then return**(a)
  j_1 :=MAX-4($A[a : \lfloor \frac{a+b-1}{2} \rfloor]$
  j_2 :=MAX-4($A[\lfloor \frac{a+b+1}{2} \rfloor : b]$
  if $A(j_2) > A(j_1)$ **then return**(j_2)
    else return(j_1)

2.2. Egyszerre a legnagyobb és a legkisebb elem

Feladat: Keressük meg egyszerre a legnagyobb és a legkisebb elemet.

Nyilván $2n-3$ összehasonlítás elég ha kiválasztjuk a legnagyobb elemet, majd a maradékból a legkisebbet. Azonban ennél jobbat szeretnénk elérni.

6. Algoritmus. A MAX-3 algoritmus első fázisával kezdünk: az elemeket párokba állítjuk, és a párokat összehasonlítjuk, majd minden párból a nagyobbat a „nyertesek" a kisebbet a „vesztesek" halmazába tesszük. Így $\frac{n}{2}$ összehasonlítás elvégzése után két részre osztottuk az inputot. A legnagyobb elem a

nyertesek maximuma, a legkisebb elem a vesztesek minimuma. Halmazonként $\frac{n}{2}-1$ összehasonlításra van szükség ahhoz, hogy a vesztesek közül az abszolút vesztest, illetve a nyertesek közül az abszolút nyertest kiválasszuk, így összesen $\frac{n}{2}+\frac{n}{2}-1+\frac{n}{2}-1=\frac{3n}{2}-2$ összehasonlításra van szükség. Könnyű meggondolni, hogy páratlan n esetén összesen $3\frac{n-1}{2}$ lépés kell, itt az első fázisban az n. elemet senkivel nem hasonlítjuk össze, viszont berakjuk a nyertesek halmazába is és a vesztesek halmazába is.

Lépésszám: $O(n)$.

3. Állítás. *Ahhoz, hogy megtaláljuk a legkisebb és a legnagyobb elemet, elég és kell is* $\lceil\frac{3}{2}n\rceil-2$ *összehasonlítás.*

2.3. Két legnagyobb elem

Feladat: Keressük meg a legnagyobb és a második legnagyobb elemet.

7. Algoritmus. Futtatjuk a MAX-4 algoritmust, ami kiadja a győztest (a legnagyobb elemet) $n-1$ összehasonlítás után. Ezt csak $\lceil\log n\rceil$ másik elemmel hasonlítottuk össze. Nyilván a második legnagyobb csak ezek közül kerülhet ki, hiszen minden más elem kisebb volt egy olyan elemnél, amely kisebb volt egy másik elemnél. Ezen elemek maximumát $\lceil\log n\rceil-1$ összehasonlítással megkaphatjuk.

4. Tétel. *Ahhoz, hogy megtaláljuk a legnagyobb és a második legnagyobb elemet, elég és kell is* $n+\lceil\log n\rceil-2$ *összehasonlítás.*

2.4. Felező (bináris) keresés

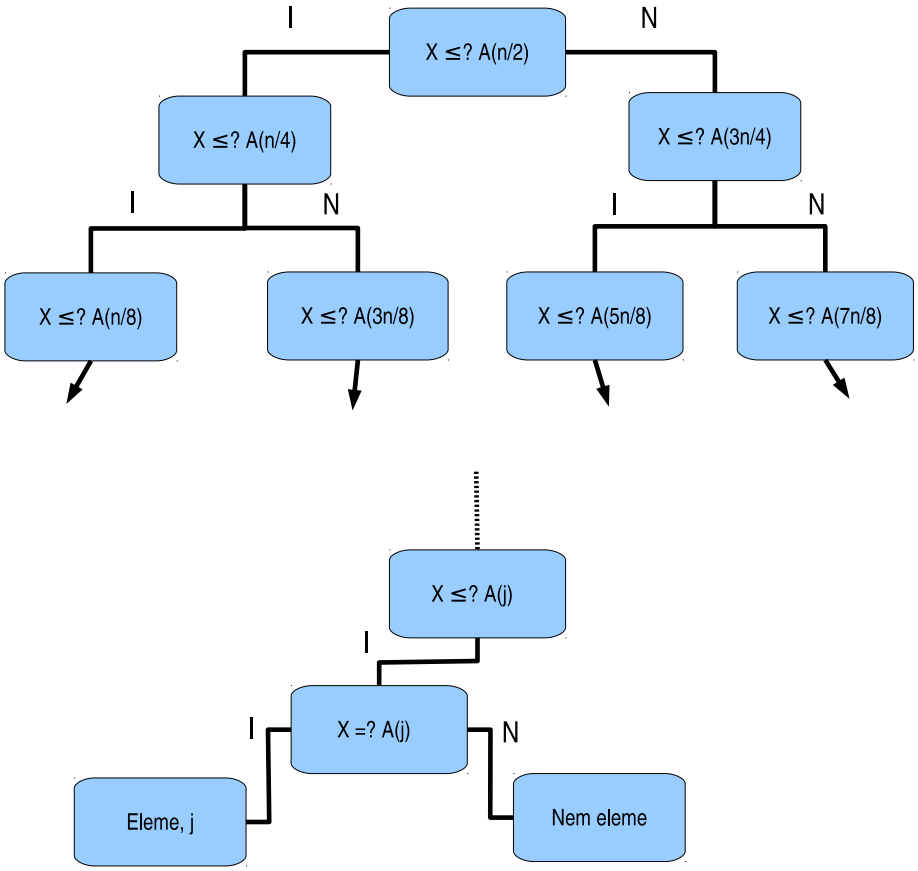
Feladat: Adott egy $(U,<)$ rendezett univerzum (tehát U egy, véges vagy végtelen halmaz, melynek elemein a $<$ egy rendezési reláció). Ennek egy S **részhalmazát** szótárnak nevezzük. Alapfeladat: egy $n=|S|$ elemű szótárat kell tárolnunk úgy, hogy a következő kérdésre gyors választ tudjunk adni: egy $x\in U$ elem benne van-e S-ben, és ha igen, akkor találjuk is meg. (Igazából minden S-beli szóhoz a „jelentését” is tárolnunk kell, de ezzel az egyszerűség kedvéért nem foglalkozunk, mivel a „találjuk meg” azt is jelenti, hogy egyúttal a párját is meg tudnánk találni.)

Kézenfekvő megoldás az ún. lineáris keresés: sorba vesszük a szótár elemeit, és mindegyikre megnézzük, hogy az-e a keresett elem. Itt a lépésszám legrosszabb esetben n, sőt az átlagos lépésszám se sokkal jobb: ha $x\notin S$ akkor ez is n, ha $x\in S$, akkor $n/2$. Vannak olyan helyzetek, amikor nem is tudunk ennél jobbat, pl. ha láncolt listában vannak S elemei, vagy ha egy külső táron rendezetlenül.

Ha azonban előre feldolgozhatjuk a szótárat, vagy – mint sok alkalmazásban – az S elemei eleve rendezve vannak egy tömbben, akkor sokat tudunk javítani. Ha nem

ez utóbbi eset áll fent, akkor először a szótárat rendezzük, és rendezve tároljuk el egy $A[1:n]$ tömbben.

Ezután egy x kérdés esetén kibarkochbázzuk az indexét, és a végén rákérdezünk.



Pontosabban, először megkérdezzük, hogy $x \leq A(\lfloor n/2 \rfloor)$ teljesül-e. Ha igen, akkor az $x \leq A(\lfloor n/4 \rfloor)$ kérdéssel folytatjuk, különben az $x \leq A(\lfloor 3n/4 \rfloor)$ kérdéssel. Amikor, $\lceil \log(n) \rceil$ kérdés után, x csak egy helyen, az $A(i)$-ben lehet, akkor megkérdezzük, hogy $x = A(i)$ igaz-e. Ha nem, akkor x nincs a szótárban. Ha igen, akkor ott van, és az indexe i. Vegyük észre, hogy ez is az „oszd meg és uralkodj” elve alapján működik, így rekurzív eljárásként kényelmesebb felírni. A feladatot a $\mathrm{BinKer}(A[1:n], x)$ hívás oldja meg.

8. Algoritmus (BinKer).

BinKer$(A[i:j],x)$: /* *Az A tömb növekvően rendezve van,* $i \le j$

if $i = j$ **then**

if $A(i) = x$ **then return**(i)

else return(0) /* *A 0 jelenti azt, hogy* $x \notin S$

$k := \lfloor \frac{i+j}{2} \rfloor$

if $x \le A(k)$ **then** BinKer$(A[i:k],x)$

else BinKer$(A[k+1:j],x)$

Összehasonlítások száma: $\lceil \log(n) \rceil + 1$, lépésszám: $O(\log n)$.

5. Állítás. *A felező keresés algoritmusa legfeljebb* $\lceil \log n \rceil + 1$ *összehasonlítással eldönti, hogy a keresett elem benne van-e a rendezett tömbben, és ha benne van, akkor az indexét is megtalálja. A lépésszám* $O(\log n)$.

Megjegyzés: A Barkochba feladatnál láttuk, hogy $\lceil \log(n) \rceil$ összehasonlítás mindig kell a legrosszabb esetben. Szokás úgy is leírni ezt az algoritmust, hogy egy lépésben három eset közül választhatunk: $x < A(i)$ vagy $x = A(i)$ vagy $x > A(i)$. Ha ezt tekintjük egy összehasonlító lépésnek, akkor az ábrázolás kicsit kompaktabb, de az összehasonlítások számában csak egyet nyerünk (sőt, ha n kettő-hatvány, akkor egyet sem), mert itt $\lceil \log(n+1) \rceil$ lesz.

2.5. Egyszerű rendezési algoritmusok összehasonlításokkal

Sokszor az U univerzum nem számokból áll, hanem pl. nevekből, vagy akár olyan objektumokból, melyek összehasonlítása egyáltalán nem nyilvánvaló. Ilyenkor U elemeivel nem tudunk műveleteket végezni, se memóriát címezni. Egyedül azt tehetjük fel (hiszen enélkül esélyünk se lenne rendezni), hogy van egy fekete doboz, ami ha megkapja az U két elemét, akkor megmondja, hogy melyik a nagyobb. Tehát ilyenkor a rendezés során az inputtal kapcsolatos minden információt csak ettől a fekete doboztól szerezhetünk.

6. Tétel. *A rendezési feladatnál az algoritmus döntéseit leíró bináris fának legalább* $n!$ *levele van és így a fa mélysége* $\ge \log(n!) \ge n \cdot \log n - \frac{3}{2} \cdot n$. *Következésképpen minden olyan algoritmusnak, mely összehasonlításokat használva rendez* n *elemet, legalább* $n \cdot \log n - \frac{3}{2} \cdot n$ *összehasonlítást kell tennie a legrosszabb esetben.*

Alapvetően minden maximum keresési algoritmus ad egy rendezési algoritmust. Ha csak az összehasonlítások szintjén vizsgáljuk, akkor az algoritmus és az elemzés egyszerű: A maximumot kiválasztjuk $n-1$ összehasonlítással, utána a többiek közül a maximumot, és így tovább. Az összehasonlítások számára a rekurzió $T(n) = n - 1 + T(n-1)$, aminek a megoldása nyilván $\binom{n}{2}$.

Persze még kell az adatok mozgatása és az output előállítása is. $O(n^2)$ időben ez sem nehéz, a megtalált maximumot kicseréljük az n. elemmel, majd n értékét eggyel csökkentjük (ez ráadásul csak legfeljebb $n-1$ mozgatást igényel).

Megjegyzés: Ez persze csak az értékeket rakja sorba, igazából kell még a B segédtömb az indexekre, mint a MAX-3 algoritmusnál, és minden CSERE$(A(i), A(j))$ utasításnál kell egy CSERE$(B(i), B(j))$ utasítás is. Ezeket a továbbiakban elhagyjuk, és csak az értékekre (kulcsokra) vonatkozó mozgatásokat írjuk le.

Ennek egy „gyengébb" változatát szokták tanítani Buborékrendezés néven, ami kicsit szemléletesebb, és jól be lehet mutatni. Itt sorba megyünk az $i = 1, \ldots, n-1$ értékeken, és amikor $A(i) > A(i+1)$, akkor ezt a kettőt megcseréljük. Amikor a végére értünk, akkor a legnagyobb elem már az n. helyen van, tehát n értékét csökkentjük, és elölről kezdjük, majd így tovább. Ekkor az összehasonlítások száma továbbra is $\binom{n}{2}$, de a mozgatások (cserék) száma is felnő legrosszabb esetben $\binom{n}{2}$-re.

A felező keresésből is származtathatunk egy rendezési algoritmust, amit Beszúró rendezésnek hívnak: rekurzívan rendezzük az első $n-1$ elemet, majd ebbe szúrjuk be az n. elemet úgy, hogy felező kereséssel megkeressük a helyét, ahová be kell szúrni. Ekkor az összehasonlítások száma már $O(n \log n)$ lesz, viszont az adatmozgatásokat nem tudjuk ennyi időben megoldani, csak $O(n^2)$-ben.

A legegyszerűbb (középiskolában is tanítható) $O(n \log n)$ lépésszámú rendezés az Összefésülő rendezés (Merge Sort). Ennek alapötlete az, hogy ha van két *rendezett* sorozatunk: $A[1:p]$ és $B[1:q]$, akkor ezeket össze tudjuk fésülni egyetlen rendezett $C[1:p+q]$ sorozattá legfeljebb $p+q-1$ összehasonlítással és $O(p+q)$ lépésben:

9. Algoritmus (Összefésül).

Merge$(A[1:p], B[1:q])$: /* *Az A és B tömbök növekvően rendezve vannak*

```
i := 1;  j := 1
while i ≤ p && j ≤ q
      if A(i) ≤ B(j) then
                              C(i+j-1) := A(i);   i++
                        else
                              C(i+j-1) := B(j);   j++
if i > p then
      for k = j..q   C(p+k) := B(k)
         else
      for k = i..p   C(q+k) := A(k)
return(C[1:p+q])
```

Ennek használatával könnyű rendezni:

10. Algoritmus (Összefésülő rendezés).
Mergesort($A[i:j]$):
  if $i = j$ **then return**($A[i:j]$)
  Mergesort($A[i:\lfloor(i+j-1)/2\rfloor]$)
  Mergesort($A[\lfloor(i+j+1)/2\rfloor:j]$)
  return (Merge($A[i:\lfloor(i+j-1)/2\rfloor], A[\lfloor(i+j+1)/2\rfloor:j]$))

Könnyű látni, hogy az összehasonlítások $T(n)$ számára $T(n) = T(\lfloor n/2\rfloor)+T(\lceil n/2\rceil)+$ $+n$, innen $T(n) \leq n\lceil\log n\rceil$. És természetesen a lépésszám is $O(n \log n)$. Az Összefésülő rendezésnek nagy szerepe van adatbázisok kezelésében és olyan rendezéseknél, ahol a rendezendő tömb külső tárolón van.

2.6. Adatstruktúrák – bevezetés

Adatstruktúráknak nevezzük az adatok tárolási célokat szolgáló strukturális elrendezését, a lekérdezési algoritmusokkal együtt. Egy algoritmus lépésszáma igen jelentősen függ a benne használt adatszerkezetektől. Ezenkívül nagy rendszerek kifejlesztésében szerzett tapasztalatok azt mutatják, hogy az algoritmus beprogramozásának nehézsége, és a végeredmény teljesítménye és minősége nagy mértékben függ a legmegfelelőbb adatszerkezet kiválasztásától.

Egy adatszerkezetet absztrakt módon úgy adunk meg, hogy megmondjuk, hogy milyen adatokat akarunk tárolni, és ezekkel milyen műveleteket akarunk végezni. Az adatszerkezet elnevezése általában ezen „kívánságlistától" függ, a tényleges megvalósításra a név elé tett jelző utal.

2.7. Kupac

Rekordokat szeretnénk tárolni, egy-egy kitüntetett Kulcs mezővel. Egyelőre két műveletre koncentrálunk: egy új rekordot be szeretnénk illeszteni (beszúrni) az eddig tároltak közé, illetve a legkisebb Kulcsú rekordot le szeretnénk kérdezni, és egyúttal törölni is (ezt a műveletet Mintörlésnek fogjuk hívni).

Itt a legegyszerűbb megvalósítással foglalkozunk, amelyet bináris kupacnak, vagy sokszor egyszerűen kupacnak neveznek. A kupacot sokszor prioritásos sornak is hívják.

Egy fát gyökeresnek nevezünk, ha ki van jelölve az egyik csúcsa, melyet gyökérnek hívunk. Egy csúcs szülője a tőle a gyökér felé vezető úton levő első csúcs (a gyökérnek nincs szülője). Egy v csúcs gyerekei azok a csúcsok, amelyeknek ő a szülője, ha ilyen nincs, a v csúcsot levélnek nevezzük. Egy gyökeres fa mélysége a gyökérből induló leghosszabb út hossza. Bináris fának hívjuk az olyan gyökeres fát, amelyben minden csúcsnak legfeljebb két gyereke van.

Megjegyzés: Hagyományos okokból a gyökeres fákat általában fejjel lefelé rajzoljuk le, tehát a gyökér van felül és a levelek alul.

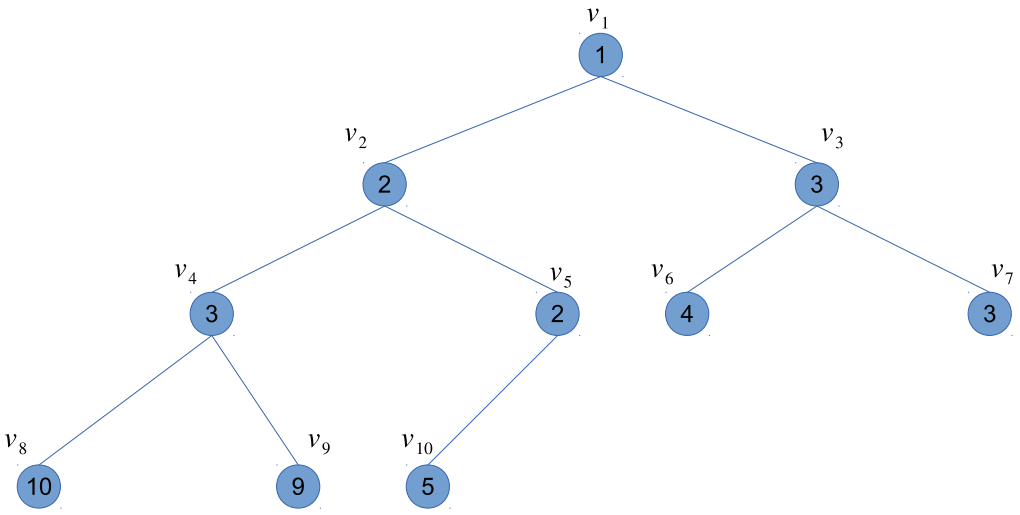
A bináris kupac váza egy majdnem teljesen kiegyensúlyozott bináris fa. Pontosabban olyan fákat használunk erre a célra, amelyekben vagy minden levél ugyanolyan távol van a gyökértől (ezekben pontosan $2^{d+1}-1$ csúcs van, ha mélységük d), vagy pedig a levelek két szinten helyezkednek el (minden levél a gyökértől $(d-1)$ vagy d távolságra van), és ezen belül az alsó szinten levő levelek „balra" vannak zárva (lásd az ábrát). Az ilyen fákat a továbbiakban egyszerűen szép fáknak hívjuk.

4. Definíció. *A bináris kupac egy szép bináris fa melynek csúcsaiban 1-1 rekord van egy kitüntetett KULCS mezővel: $K(v_i)$ jelöli a v_i csúcsban lévő rekord kulcsát. Ezenkívül teljesül a* kupacrendezettség*: minden, a gyökértől különböző v csúcsra igaz, hogy $K(\text{szülő}(v)) \leq K(v)$.*

A kupac-fa csúcsait felülről lefelé, ezen belül balról jobbra számozzuk be. Tehát, a szépség miatt: v_i bal fia: v_{2i}; jobb fia: v_{2i+1}; szülője: $v_{\lfloor \frac{i}{2} \rfloor}$. Emiatt egy kupac elemeit egy tömbben is tárolhatjuk. Most az egyszerűség kedvéért a tömb i. helyén csak a $K(v_i)$ kulcsot fogjuk tárolni, de megjegyezzük, hogy igazából az alkalmazásokban még két másik tömbre is szükség van, az egyik tömb i. helyén a v_i csúcsban tárolt rekord van, a másikban a j. helyen a j. rekordot tároló v_i csúcs i indexe.

Az n csúcsú szép fa mélysége: $\lfloor \log n \rfloor$. Egy n csúcsú fa pontosan akkor szép, ha a fenti tárolással el lehet tárolni egy n hosszú tömbben.

Az alábbi ábrán egy kupac látható, a Kulcsok a csúcsokba vannak beleírva:



Látni fogjuk, hogy a kupacon végzett műveletek lépésszáma a fa magasságával arányos, azaz $O(\log n)$ idejű.

TÁROLÁS:

$A[1:n]$ /* *A kupacot egy n hosszú A tömbben tároljuk, legfeljebb n elem lesz benne.*

$A(i) = K(v_i)$ /* *Az A tömb i-edik eleme egyenlő a „szép fa" i-edik csúcsának kulcsával, (azaz A(1) a gyökér kulcsa).*

$0 \leqslant$ VÉGE $\leqslant n$ /* *A VÉGE számláló megadja, hogy pillanatnyilag hány elem van a kupacban.*

2.7.1. Műveletek a kupac adatszerkezetben

11. Algoritmus (BESZÚRÁS).

BESZÚRÁS(A, új): /* *Az A tömbbe beszúrjuk az új elemet.*
VÉGE++ /* *VÉGE értékét növeljük 1-gyel.*
A(VÉGE) := K(új) /* *Beszúrtuk az elemet.*
FELBILLEGTET(A, VÉGE) /* *Kupacrendezettséget kijavítjuk.*

FELBILLEGTET(A, i):
$AA := A(i)$ /* *Legyen AA a felbillegtetendő elem kulcsa.*
while $i > 1$ **&&** $A(\lfloor \frac{i}{2} \rfloor) > AA$ /* *Szülőben lévő kulcs nagyobb mint AA.*
$A(i) := A(\lfloor \frac{i}{2} \rfloor)$ /* *Szülőben lévő kulcsot levisszük a gyerekbe.*
$i := \lfloor \frac{i}{2} \rfloor$ /* *Az i egyenlő lesz a szülő indexével.*
$A(i) := AA$ /* *Végül az AA érték „felmegy” a leálláskori v_i csúcsba.*

Lépésszám: $O(\log n)$

12. Algoritmus (MINTÖRLÉS). Kicseréljük a gyökeret és az utolsó elemet, majd töröljük ezt az utolsó levelet. Végül helyreállítjuk a kupac tulajdonságot. A MINTÖRLÉSnél két él mentén is elromolhat a kupac tulajdonság, mindig a két gyerek közül kiválasztjuk a kisebbiket, és ha cserélni kell, akkor ezzel cserélünk.

MINTÖR(A): /* *Megadja a legkisebb elemet, és ezt kitörli a kupacból.*
CSERE($A(1), A$(VÉGE)) /* *Kicseréljük az első és az utolsó elemet.*
VÉGE−− /* *VÉGE értékét csökkentjük, így a legkisebb elem kikerült a kupacból.*
LEBILLEGTET(A,1) /* *LEBILLEGTETjük a gyökeret.*
return(A(VÉGE+1)) /* *Ide raktuk a legkisebb elemet.*

LEBILLEGTET(A, i):
$AA := A(i)$; $j := 2i+1$ /* j *a v_i jobb gyerekének indexe.*
while $j \leq$VÉGE
if $A(j-1) < A(j)$ **then** j−− /* *Ha a bal gyerek kisebb, j mutasson arra.*
if $A(j) < AA$ **then** $A(i) := A(j)$ /* *Amennyiben a gyerek kisebb, akkor felvisszük a gyereket a szülőbe.*
$i := j$; $j := 2i+1$ /* *Az indexek frissítése.*
else $j :=$VÉGE+2 /* *Különben úgy állítjuk be j-t, hogy kilépjünk.*

$j{-}{-}$ /* *Csökkentjük j-t, hogy a bal gyerekre mutasson.*
if $j \leq$VÉGE **&&** $A(j) < AA$ **then** $A(i) := A(j)$; $i := j$ /* *Abban az esetben, ha i-nek csak egy gyereke van, és kell cserélni.*
$A(i) := AA$ /* *Végül a tárolt AA elem bekerül a helyére.*

Lépésszám: $O(\log n)$

7. Tétel. *Az n csúcsú szép fa mélysége* $\lfloor \log n \rfloor$*, tehát a BESZÚRÁS művelet* $O(\log n)$ *lépés, és legfeljebb* $\log n$ *összehasonlítás kell. A MINTÖRLÉSnél a lépésszám szintén* $O(\log n)$*, és legfeljebb* $2 \log n$ *összehasonlítás történik.*

2.7.2. Kupacos rendezés

Végül rátérhetünk a rendezési feladatot megoldó algoritmusra.

13. Algoritmus (Kupacos rendezés I.).

VÉGE:$= 0$ /* *Üres kupac*
for $i = 1..n$
  BESZÚRÁS(A, a_i) /* *Elvégzünk n db beszúrást, így felépítünk az inputból egy kupacot.*
for $i = 1..n$
  MINTÖR(A) /* *Majd elvégzünk n db MINTÖRLÉST.*

Lépésszám: $O(n \cdot \log n)$.

A II. változatban a kupacépítés részt lineáris idejűvé tesszük. Tetszőlegesen betesszük az inputot a tömbbe, majd „alulról felfele” haladva, lebillegtetésekkel kupacrendezzük.

14. Algoritmus (Kupacos rendezés II.).

VÉGE:$= n$ /* n *darab inputunk van.*
for $i = 1..n$
  $A(i) := a_i$ /* *Bepakoljuk az elemeket egy tömbbe úgy, ahogyan „érkeznek”.*
for $i = \lfloor \frac{n}{2} \rfloor..1$ (-1) /* *Majd ezután LEBILLEGTETÉSEKKEL rendezzük a tömböt.*
  LEBILLEGTET(A, i)
for $i = 1..n$ /* *Innentől ugyanaz*
  MINTÖR(A)

Megjegyzés: A kupacépítésnél az összes lebillentések száma lényegében
$\frac{n}{2}\times 0 + \frac{n}{4}\times 1 + \frac{n}{8}\times 2 + \ldots + 1\times\lceil \log n \rceil < n\times(\frac{1}{4} + \frac{2}{8} + \frac{3}{16} + \ldots) = n.$

Lépésszám (kupacépítésnél): $O(n)$, a kupacos rendezés továbbra is $O(n \log n)$. De ha csak a k darab legkisebb elemre vagyunk kíváncsiak növekvő sorrendben, akkor $O(n + k \log n)$.

2.8. Mediáns keresés

Feladat: A középső elem kiválasztása.

Megjegyzések: Mediánnak hívjuk a rendezés szerinti középső elem értékét, mediánsnak pedig magát a középső elemet.

Tulajdonképpen a feladatot egy általánosabb kiválasztási algoritmussal oldjuk meg, azaz rögtön a k. legkisebb elemet keressük, mert ezt tudjuk jól rekurzívan meghívni.

Az elemzéshez feltesszük, hogy az egészrészekkel nem kell foglalkozunk, valamint hogy az input különböző értékű elemekből áll.

Az algoritmus az $n > 1$ elemből álló bemeneti tömb k-adik legkisebb elemét határozza meg. Ügyesen kiválasztunk egy x elemet, amely „nagyjából" középen van, ezzel mindenkit összehasonlítunk, a kisebbeket a B, a nagyobbakat a J csoportba tesszük. Ezután ha $|B| = k-1$, akkor x a keresett elem, ha $|B| \geq k$, akkor a B halmazból a k. elemet keressük rekurzívan, ha pedig $|B| < k-1$, akkor a J halmazból a $(k-|B|-1)$. elemet keressük rekurzívan.

15. Algoritmus (k-adik elem).

I. Osszuk a bemeneti tömb n elemét $\frac{n}{5}$ darab 5 elemből álló csoportra.

II. Mind az $\frac{n}{5}$ darab csoportnak keressük meg a mediánsát (ezt csoportonként 6 összehasonlítással megtehetjük, lásd a 4. példát). Legyenek ezek $b_1, b_2, ..., b_{(\frac{n}{5})}$.

III. Az algoritmus rekurzív használatával határozzuk meg a 2. lépésben kapott $\frac{n}{5}$ darab mediáns x mediánsát. Tehát $b_1, b_2, ..., b_{(\frac{n}{5})}$ középső eleme x.

IV. Majd a bemeneti tömböt osszuk fel a mediánsok mediánsa, azaz x szerint úgy, hogy minden elemet összehasonlítunk x-szel. A kisebbeket a B, a nagyobbakat a J csoportba tesszük. Ezután ha $|B| = k-1$, akkor x a keresett elem, ha $|B| \geq k$, akkor a B halmazból a k. elemet keressük rekurzívan, ha pedig $|B| < k-1$, akkor a J halmazból a $(k-|B|-1)$. elemet keressük rekurzívan.

8. Állítás. *A mediáns keresésnél (ha x-et az algoritmus szerint választjuk), akkor $\leq \frac{7}{10}n$ db x-nél nagyobb elem és $\leq \frac{7}{10}n$ db x-nél kisebb elem van. Emiatt a k. legkisebb elem keresésénél az összehasonlítások száma ezzel az algoritmussal legfeljebb $22n$, így lépésszáma $O(n)$.*

2.9. Gyorsrendezés (Quicksort)

A gyorsrendezés olyan rendezési algoritmus, amelynek futási ideje legrosszabb esetben $O(n^2)$. Ennek ellenére a gyakorlatban sokszor ezt használják, mivel átlagos futási ideje $O(n \log n)$, ráadásul a képletben rejlő konstans meglehetősen kicsi, ezenkívül helyben rendez (nincs szükség segéd-tömbökre). A gyorsrendezés algoritmusnak fontos részét képezi az input szétosztása és az algoritmus rekurzív meghívása. Mi a praktikus randomizált változatot tárgyaljuk, mely a szétosztáshoz az osztóelemet véletlenül választja. Az egyszerűsítés végett itt is feltesszük, hogy az input csupa különböző elemből áll. A rendezést a **QuickSort**$(A[1:n])$ hívás végzi el.

16. Algoritmus (Gyorsrendezés).

QuickSort$(A[p:r])$**:** /* *Az* $A[p:r]$ *résztömb rendezése.*
  if $p < r$ **then**
    $q :=$ **Partition**$(A[p:r])$ /* *Kettéosztjuk az* $A[p:r]$ *résztömböt.*
    QuickSort$(A[p:q-1])$ /* *Rekurzívan rendezzük a balra állókat.*
    QuickSort$(A[q+1:r])$ /* *Valamint a jobbra állókat is.*

Partition$(A[p:r])$**:**
  $i :=$**RND**$[p,r]$ /* *Kiválasztunk egy véletlen* i*-t,* $A(i)$ *lesz az osztóelem.*
  CSERE$(A(i), A(r))$ /* *Ezt becseréljük az utolsó helyre.*
  $x := A(r)$ /* *Eltároljuk* x*-be az osztóelem értékét.*
  $i := p-1$
  for $j = p..r$
    if $A(j) \leq x$ **then** $i{+}{+}$; **CSERE**$(A(i), A(j))$ /* A *j indexszel végigmegyünk az egész tömbön és ha a* j*. elem* $\leq x$*, akkor az* x*-nél nagyobb elemektől balra áthelyezzük.*
  return(i) /* *Ez lesz az osztóelem végső indexe,* $A(i)$*-be kerül az osztóelem.*

Példa a szétosztásra: 5. példa.

Azt, hogy ez az algoritmus jól működik, egyszerűen beláthatjuk. Elég időre vonatkozó indukcióval igazolni, hogy minden ciklusvégkor igaz lesz az, hogy ha $k \leq i$, akkor $A(k) \leq x$, ha pedig $i < k \leq j$, akkor $A(k) > x$.

9. Tétel. *A gyorsrendezésben az összehasonlítások számának várható értéke:*

$$< 1{,}39 \cdot n \cdot \log n + O(\log n).$$

Bizonyítás. Jelölje $C(n)$ az összehasonlítások számának várható értékét, és $C(n,j)$ pedig azt a feltételes várható értéket, hogy mennyi az összehasonlítások számának

várható értéke, feltéve, hogy az első Partition a j-edik legkisebb elemet választja ki. Könnyű látni, hogy

$$C(n,j) = n-1+C(j-1)+C(n-j), \text{ valamint hogy}$$

$$C(n) = \frac{1}{n}\Big(C(n,1)+C(n,2)+\ldots+C(n,n)\Big).$$

$$\text{Innen } C(n) = n-1+\frac{2}{n}\Big(C(1)+C(2)+\ldots+C(n-1)\Big)$$

Már csak ezt a rekurziót kell megoldanunk.

$$nC(n) = n(n-1)+2\Big(C(1)+C(2)+\ldots+C(n-1)\Big),$$

$$(n-1)C(n-1) = (n-1)(n-2)+2\Big(C(1)+C(2)+\ldots+C(n-2)\Big),$$

$$nC(n) = (n-1)C(n-1)+2(n-1)+2C(n-1) = 2(n-1)+(n+1)C(n-1).$$

$$f(n) := \frac{C(n)}{n+1} = \frac{2(n-1)}{n(n+1)}+\frac{C(n-1)}{n} < \frac{2}{n}+f(n-1).$$

Innentől könnyű dolgunk van, mivel $f(1)=0$, ezért

$$f(n)<2\sum_{i=2}^{n}\frac{1}{i}=2\big(\log_e n+\gamma-1+O(1/n)\big)<2\big(\log_e n+0{,}578-1+O(1/n)\big)<2\log_e n+O(1/n).$$

Ebből pedig $C(n)=2(n+1)\log_e n+O(1)=\frac{2}{\log e}(n+1)\log n+O(1)<1{,}39\cdot n\cdot\log n+O(\log n)$ adódik.

2.10. Kulcsmanipulációs rendezések

2.10.1. Leszámláló rendezés

A leszámláló rendezést akkor érdemes használnunk, ha az n bemeneti érték mindegyike 0 és m közötti egész szám, ahol m egy nem túl nagy egész. Ezt arra használjuk, hogy így a bemenet elemeivel tudunk egy m méretű C tömböt indexelni. A leszámláló rendezés alapötlete, hogy minden egyes k lehetséges bemeneti értékre meghatározza azoknak az elemeknek a számát, amelyek értéke legfeljebb k. Ezzel az információval egy k értékű elemet közvetlenül a saját pozíciójába tudjuk elhelyezni a kimeneti tömbben. Ha pl. 13 olyan elem van, amely nem nagyobb k-nál akkor a k értékű elem a 13. helyen fog szerepelni (ha csak egy ilyen van).

Mivel itt az elemek kis egészek, itt NEM tesszük fel, hogy a bemenetek különbözőek. Persze ha több k értékű elem is van, akkor érdemes bevezetni egy új, szigorú rendezést: $a_i \prec a_j$, ha vagy $a_i < a_j$, vagy pedig $a_i = a_j$ és $i < j$. Egy rendezés **stabil**, ha eszerint rendez, tehát az egyforma értékű elemek közül a később érkezett kerül későbbre. Ez azt jelenti, hogy a legutolsó k értékű elemet rakjuk a 13. helyre, az utolsó előtti k értékűt a 12. helyre, és így tovább. Tehát a k értékű elemek az outputban ugyanolyan sorrendben lesznek, mint ahogy az inputban is voltak.

5. Definíció. *A leszámláló rendezésnél az input:* $a_1, \ldots, a_n$, *ahol* $0 \leq a_i \leq m$ *egészek. Stabil rendezés esetén az output:* $i_1, i_2, \ldots, i_n$ *permutáció, amelyekre igaz, hogy* $a_{i_1} \leq \leq a_{i_2} \leq \ldots \leq a_{i_n}$, *és ha valamely* $k < l$ *számokra* $a_{i_k} = a_{i_l}$, *akkor* $i_k < i_l$ *is teljesül.*

$A[1:n]$ /* *Bemeneti tömb.*
$B[1:n]$ /* *Kimeneti tömb. Pl.* $B(1) = i$, *ha* a_i *a legkisebb elem.*
$C[0:m]$ /* *Átmeneti munkaterület, ahol* m *a felső becslés az input értékekre.*

17. Algoritmus (Leszámláló rendezés).

LESZREND$(A[1:n], m)$**:**
for $i = 0..m$ $C(i) := 0$ /* *Nullázzuk a* C *tömböt.*
for $j = 1..n$ $C(A(j))$++ /* *Az* A *tömb elemeivel indexeljük a* C *tömböt, végül* $C(i)$ *az* i *értékű elemek számát tartalmazza.*
for $i = 1..m$
$C(i) := C(i) + C(i-1)$ /* *Most* $C(i)$ *értéke az lesz, hogy hány elem* $\leq i$.
for $j = n..1$ (-1) /* *Fordított sorrendben megyünk végig, ez garantálja a stabil rendezést.*
$B(C(A(j))) := j$ /* *Az* $A(j)$ *elem indexét berakjuk a végső helyére.*
$C(A(j))$−− /* *A következő ugyanilyen értékűt már eggyel balra kell rakni.*
return$(B[1:n])$

Lépésszám: $O(n+m)$

Példa a leszámláló rendezésre: 6. példa.

2.10.2. Számjegyes rendezés (1887)

6. Definíció. *A számjegyes rendezésnél az input:* $a_1, \ldots, a_n$ *természetes számok, ahol minden* i*-re* $a_i = \overline{b_i^{k-1} b_i^{k-2} \ldots b_i^0}$ *valamilyen* m *alapú számrendszerben, tehát* $0 \leq b_i^j < m$ *és* $a_i = \sum_{j=0}^{k-1} b_i^j \cdot m^j$.

Minden input szigorúan ugyanannyi (itt k) számjegyből áll (ha eredetileg mégsem, írjunk a rövidebbek elejére 0-kat). Először az utolsó, azaz a legkevésbé fontos számjegy alapján rendezünk, majd az utolsó előtti szerint és így tovább, de mindig stabil rendezést használunk, pl. leszámláló rendezést.

18. Algoritmus (Számjegyes rendezés).

for $j = 0..k-1$
STABILREND$(b_i^j$ **szerint**$)$

10. Állítás. *A számjegyes rendezés jól rendezi az inputot, a lépésszám* $O(k \cdot (n+m))$.

Példa a számjegyes rendezésre: 7. példa.

Ha eredetileg természetes számokat kell rendeznünk, amelyek mindegyike kisebb, mint n^{10}, akkor ezeket n alapú számrendszerbe írva 10-jegyű számokat kapunk, tehát a számjegyes rendezés lépésszáma $O(10(n+n)) = O(n)$ lesz.

3. fejezet

Aritmetika: Számolás nagy számokkal

3.1. Alapműveletek, hatványozás

Eddig két szám összeadását ill. szorzását 1 lépésnek számoltuk. Nagyon nagy számoknál ez nem tisztességes, így ebben a fejezetben kivételesen csak a bitműveletek számítanak egy lépésnek. Az inputok legfeljebb n bites számok. Az alapműveleteket az általános iskolában tanult módon végezzük el (bár szorozni és osztani ennél hatékonyabban is lehet, erre itt nem térünk ki); kivétel lesz a hatványozás. Figyeljük meg, hogy kettes számrendszerben ezeket sokkal könnyebb elvégezni, mint a 10-es számrendszerben, pl. maradékos osztásnál csak összehasonlítás és kivonás kell.

Példa az összeadásra: 8. példa.
Példa a kivonásra: 9. példa.
Példa a szorzásra: 10. példa.
Példa a maradékos osztásra: 11. példa.

11. Állítás. *Legyenek a és b legfeljebb n bites természetes számok. Ekkor az $a+b$ összeadásnak és az $a-b$ kivonásnak a lépésszáma az általános iskolában tanult algoritmusokkal $O(n)$, az $a \cdot b$ szorzás illetve az $a : b$ maradékos osztás lépésszáma pedig $O(n^2)$.*

19. Algoritmus (Hatványozás, azaz a^b kiszámítása).

Először is vegyük észre, hogy az $a, a^2, a^4, a^8, \ldots, a^{2^n}$ sorozat elemei n darab négyzetre-emeléssel (azaz szorzással) kiszámíthatóak. Legyen b kettes számrendszerben felírva $b_{n-1}b_{n-2}\ldots b_2b_1b_0$, tehát $b = \sum_{i=0}^{n-1} b_i \cdot 2^i$. Ekkor

$$a^b = a^{\sum_{i:\, b_i=1} 2^i} = \prod_{i:\, b_i=1} a^{2^i}.$$

Tehát a^b kiszámításához az először kiszámolt hatványok közül legfeljebb n darab összeszorzása elég. Összesen tehát legfeljebb $2n$ szorzást végeztünk.

Sajnos a bitműveletekben számolt műveletigény lehet nagyon nagy is, mivel egyre hosszabb számokat kell összeszoroznunk (az output maga is állhat több, mint 2^n bitből). A módszer mégis hasznos, főleg olyan esetekben, ahol a hosszak nem nőnek meg, lásd később. Jól használható mátrixok hatványozására is.

12. Állítás. *Legyenek a és b legfeljebb n bites természetes számok. Az a^b hatvány kiszámolható legfeljebb $2n$ db szorzással (de egyre hosszabb számokat kell szorozni).*

3.2. Mod m számolás

Input: a, b és m legfeljebb n bites természetes számok, ahol $a < m$ és $b < m$. Az egész számokon bevezetjük ezt az ekvivalencia-relációt: $x \equiv y \pmod{m}$, ha $x - y$ osztható m-mel. Ennek ekvivalencia-osztályait hívjuk maradékosztályoknak. Ilyenekkel szeretnénk számolni, azaz két maradékosztályt összeadni, összeszorozni stb. Egy maradékosztály reprezentánsának mindig azt az elemét nevezzük, mely nemnegatív és kisebb m-nél. Tehát a és b egy-egy maradékosztály reprezentánsa. Természetesen az eredmény maradékosztálynak is a reprezentánsát akarjuk kiszámolni. Ennek megvan az előnye, hogy sose lesz hosszabb n bitnél.

20. Algoritmus (mod m számítások I.).

a + b mod **m:**
$c := a + b$
if $c \geqslant m$ **then** $c := c - m$

Lépésszám: $O(n)$

a − b mod **m:**
$c := a - b$
if $c < 0$ **then** $c := c + m$

Lépésszám: $O(n)$

a · b mod **m:**
$(a \cdot b) : m$ maradéka az eredmény.

Lépésszám: $O(n^2)$, mert egy szorzás és egy maradékos osztás.

13. Állítás. *Legyenek a, b és m legfeljebb n bites természetes számok, valamint $a < m$ és $b < m$. Ekkor $a + b \pmod{m}$ és $a - b \pmod{m}$ kiszámítható $O(n)$ lépésben, $a \cdot b \pmod{m}$ pedig kiszámítható $O(n^2)$ lépésben (egy szorzás és egy maradékos osztás m-mel).*

Az **a : b** mod **m** feladat: Ezt először definiálni kell. A keresett eredmény x egész szám, ahol $0 \leq x < m$ és $a \equiv x \cdot b \pmod{m}$. Ilyen x nem mindig létezik. Abban a fontos esetben, amikor b és m relatív prímek azonban mindig létezik és nemsokára tárgyaljuk a kiszámítását.

Megjegyzés: Ha $(b,m)=d$, akkor nyilván csak akkor létezhet ilyen x, ha $d|a$. Ekkor az $a'=a/d$, $b'=b/d$, $m'=m/d$ jelöléssel egyrészt $(b',m')=1$ és $a\equiv x\cdot b \pmod m$ akkor és csak akkor, ha $a'\equiv x\cdot b' \pmod{m'}$.

21. Algoritmus (mod m hatványozás).

$a^b \mod m$:
Az előző hatványozási algoritmust (19. algoritmus) végezzük el úgy, hogy minden szorzást a most tárgyalt mod m szorzással helyettesítünk. Így minden (közbenső és végső) eredményünk kisebb lesz m-nél, tehát végig legfeljebb $2n$ bites számokkal kell számolnunk.

Megjegyzés: Ha x és y legfeljebb n bites számok, akkor $x,y<2^n$, tehát $xy<2^{2n}$, azaz xy legfeljebb $2n$ bites.

14. Tétel. *Legyenek a, b és m legfeljebb n bites természetes számok, valamint $a<m$ és $b<m$. Ekkor az $a^b \mod m$ hatvány kiszámítható $O(n^3)$ bitművelettel (minden szorzás után csinálunk egy maradékos osztást).*

3.3. Euklideszi algoritmus

Feladat: Legnagyobb közös osztó megkeresése.

22. Algoritmus (Euklideszi algoritmus).

LNKO(A,B)**:**
  if $A>B$ **then lnko**(B,A)
    else lnko(A,B)

lnko(a,b)**:**
  if $a=0$ **then return**(b)
  $b:a$**, azaz** $b=c\cdot a+r$, ahol $0\le r<a$
  lnko(r,a)

15. Tétel. *Az euklideszi algoritmus során maximum $2n$ db maradékos osztást végzünk el, tehát a lépésszám bitműveletekben $O(n^3)$.*

3.4. Kiterjesztett euklideszi algoritmus

Feladat: Az (A,B) legnagyobb közös osztó megkeresése, valamint előállítása $(A,B)=u\cdot A+v\cdot B$ alakban, ahol u,v egész számok.

23. Algoritmus (Kiterjesztett euklideszi algoritmus).

LNKO(A, B)**:**
  if $A > B$ **then lnko**$(B, A, 0,1,1,0)$ /* *Az első argumentum előáll* $0 \cdot A + 1 \cdot B$ *alakban, a második* $1 \cdot A + 0 \cdot B$ *alakban.*
  else lnko$(A, B, 1,0,0,1)$

lnko(a, b, u, v, w, z)**:** /* *Híváskor tudjuk, hogy* $a = u \cdot A + v \cdot B$ *és* $b = w \cdot A + z \cdot B$.
  if $a = 0$ **then return**(b, w, z)
  $b : a$**, azaz** $b = c \cdot a + r$, ahol $0 \leq r < a$
  lnko$(r, a, (w - cu), (z - cv), u, v)$

Lépésszám: $O(n^3)$

3.5. Mod m osztás

Mod m osztás a $(b, m) = 1$ esetben.

24. Algoritmus (mod m osztás).

A kiterjesztett euklideszi algoritmussal ellenőrizzük, hogy $(b, m) = 1$ igaz-e, és ha igen, akkor egyúttal kiszámolunk u és v egészeket, hogy $1 = u \cdot b + v \cdot m$, azaz $u \cdot b \equiv 1 \pmod{m}$. Ezután x legyen az $u \cdot a \mod m$ szorzás eredménye. Könnyű ellenőrizni, hogy $x \cdot b \equiv a \pmod{m}$.

16. Tétel. *Legyenek* a, b *és* m *legfeljebb* n *bites természetes számok. Minden* $a, b \in \in \mathbb{N}$ *számpárhoz létezik* $u, v \in \mathbb{Z}$, *hogy* $(a, b) = u \cdot a + v \cdot b$. *Ilyen* u, v *párt a Kiterjesztett euklideszi algoritmus meg is talál* $O(n^3)$ *lépésben. Ennek segítségével, ha* b *és* m *relatív prímek, akkor az* $a : b \mod m$ *osztás is kiszámítható* $O(n^3)$ *lépésben.*

3.6. Kínai maradéktétel

A Kínai maradéktétel szerint, ha ismerjük egy x ismeretlen számnak a maradékát az egymáshoz páronként relatív prím $m_1, \ldots, m_k$ modulusokra nézve, akkor ez egyértelműen meghatározza x maradékát a $m_1 \cdot \ldots \cdot m_k$ szorzatra nézve. Ezt elég kiszámítani a $k = 2$ esetre, hiszen pl. $k = 3$ esetén először kiszámítjuk a maradékot $m_1 m_2$-re nézve, majd ebből és az m_3-ra vett maradékból kiszámítjuk $m_1 m_2 m_3$-ra nézve.

Tegyük fel, hogy ismerjük az x_1 és x_2 maradékokat. A kiterjesztett euklideszi algoritmust futtatva kapunk q_1 és q_2 egészeket, hogy $1 = q_1 m_1 + q_2 m_2$. Könnyű belátni, hogy $x = x_1 + (x_2 - x_1) q_1 m_1 \mod m_1 m_2$ pont a keresett érték.

4. fejezet

Dinamikus programozás

Egy algoritmust akkor szokás hatékonynak tekinteni, ha lépésszáma a bemenet hosszának valamilyen polinomjával korlátozható. Lásd a 3. példát.

P=$\{$problémák | létezik A algoritmus, ami jól megoldja, és létezik c, hogy minden x inputra A legfeljebb $c\cdot|x|^c$ lépést tesz.$\}$

Elnevezés: P = Polinomiális osztály = „Hatékonyan megoldható problémák”.

A dinamikus programozás az „alulról felfele építkezés” elvén alapszik, akkor alkalmazható, ha a részproblémák nem függetlenek, azaz közös részproblémáik vannak. Kell még, hogy teljesüljön a szuboptimalitás elve, azaz hogy az optimális megoldást szolgáltató struktúra megszorítása egy részfeladatra, annak a részfeladatnak szükségképpen optimális megoldását szolgáltassa.

Amennyiben rekurzívan oldanánk meg az ilyen feladatokat, akkor sokszor számolnánk ki ugyanazon részprobléma megoldását. A dinamikus programozás minden egyes részfeladatot pontosan egyszer old meg.

4.1. Fibonacci számok

Feladat: Fibonacci számok kiszámolása.

Fibonacci számok, rossz verzió
FIB(n):
  if $n\leq 1$ **then return**(n)
  else return(**FIB**$(n-1)$+**FIB**$(n-2)$)

Könnyű ellenőrizni, hogy ez a rekurzív algoritmus F_n-et lényegében F_n idő alatt számítja ki, ami viszont n-ben exponenciális (kb. $\left(\frac{\sqrt{5}+1}{2}\right)^n$). A lassúság oka pedig az, hogy pl. az F_2 értéket exponenciálisan sokszor fogja újra és újra kiszámolni.

25. Algoritmus (Fibonacci számok).

$F(0):=0;\ \ F(1):=1$
  for $i=2..n$
    $F(i)=F(i-1)+F(i-2)$

Lépésszám: $O(n)$

4.2. Hátizsák-feladat

Feladat: Egy hátizsákban tárgyakat szeretnénk kivinni a kincses barlangból. A hátizsáknak van egy teherbírása: W (egész), az n tárgynak van súlya: w_i (egészek, $1 \leq i \leq n$), valamint értéke: e_i (egészek). Cél a hátizsákban maximális legyen a tárgyak összértéke úgy, hogy ne lépjük túl a teherbírást.

7. Definíció. *A Hátizsák-feladatban az input W egész szám (a hátizsák teherbírása), $w_1, w_2, \ldots, w_n$ (egész súlyok), valamint $e_1, e_2, \ldots, e_n$ (egész értékek).*

Output: $I \subseteq \{1,2 \ldots n\}$, hogy $\sum_{i \in I} w_i \leq W$ teljesüljön, és $\sum_{i \in I} e_i$ ezen belül a lehető legnagyobb legyen.

Először helytakarékos megoldást mutatunk, csak a maximális értéket számítjuk ki, azt nem, hogy milyen tárgyakat vigyünk ki.

26. Algoritmus (Hátizsák – csak OPTérték).

for $j = 0..W$ $T(j) := 0$ /* *$T(j)$ lesz a j teherbírású hátizsákban max. kivihető érték.*

for $i = 1..n$ /* *Az i. ciklusban az első i tárgyból max. kivihető értékeket keressük.*

  for $j = W..w_i$ (-1) /* *Minden szóbajöhető j-re frissítjük $T(j)$-t.*

    if $T(j - w_i) + e_i > T(j)$ **then** $T(j) := T(j - w_i) + e_i$

/* *Ha az i. tárgyat beválasztva több értéket tudunk kivinni j tömegben, akkor frissítünk.*

return$(T(W))$ /* *Ez lesz a max. érték, amit W súlyban ki lehet vinni.*

Lépésszám: $O(n \cdot W)$

Az algoritmus lefutására lásd a 12. példát.

A kiviendő tárgyak halmazát sem nehéz meghatározni, ehhez azonban nem elég egy tömb, kell egy mátrix, melynek i. sora lesz az előző $T(j)$ tömb a ciklus i. lefutása után.

27. Algoritmus (Hátizsák).

for $j = 0..W$ $T(0, j) = 0$
for $i = 1..n$
  for $j = 0..W$ $T(i, j) := T(i - 1, j)$
  for $j = w_i..W$
    if $T(i - 1, j - w_i) + e_i > T(i - 1, j)$ **then**
      $T(i, j) := T(i - 1, j - w_i) + e_i$
return(T)

Az I halmaz visszafejtése: Ha $T(n,W)=T(n-1,W)$, akkor az n. tárgy ne kerüljön be, és folytassuk a visszafejtést $T(n-1,W)$-től. Különben pedig az n. tárgy kerüljön be, és folytassuk a visszafejtést $T(n-1,W-w_n)$-től.

Lépésszám: $O(n\cdot W)$

4.3. Mátrix-szorzás zárójelezése

Feladat: adottak $A_1, A_2, ..., A_n$ mátrixok, illetve ezek $r_0\times r_1, r_1\times r_2, \ldots, r_{n-1}\times r_n$ méretei, és a mátrixok $A_1A_2A_3...A_n$ szorzatát kívánjuk kiszámítani. (A szorzat nyilván csak akkor definiált, ha A_i oszlopainak száma $= A_{i+1}$ sorainak számával minden i-re.)

A mátrix szorzás asszociatív, ezért tetszőlegesen zárójelezhetjük a szorzatot, a cél az, hogy úgy zárójelezzük, hogy a lehető legkevesebb szorzást kelljen elvégezni, és a zárójelezés egyértelmű legyen (azaz teljesen zárójelezett). Ha az A mátrix $p\times q$ és a B mátrix $q\times r$ méretű akkor a $C=A\cdot B$ mátrix $p\times r$ méretű. C kiszámításához szükséges idő a szorzások mennyiségével arányos, ami pqr. Lásd a 13. példát.

Jelölje $i\leq j$ esetén $A_{i..j}$ az $A_iA_{i+1}...A_j$ szorzatot, nyilván ennek mérete $r_{i-1}r_j$. Ennek optimális zárójelezésében a legkülső zárójel két részre bontja ezt a szorzatot, valamely A_k és A_{k+1} mátrix között, ahol $i\leqslant k<j$. Tehát az optimális zárójelezés költsége az $A_{i..k}$ és az $A_{k+1..j}$ mátrixok kiszámításának optimális költsége plusz összeszorzásuknak $r_{i-1}r_kr_j$ költsége.

Legyen $m(i,j)$ az $A_{i..j}$ mátrix kiszámításához minimálisan szükséges szorzások száma. Tfh. az optimális zárójelezés az A_k és az A_{k+1} mátrixok között vágja szét az $A_iA_{i+1}...A_j$ szorzatot, ahol $i\leqslant k<j$. Ekkor a szuboptimalitás elve alapján azt kapjuk, hogy $m(i,j)=m(i,k)+m(k+1,j)+r_{i-1}r_kr_j$. Persze k értékét nem ismerjük, azt is meg kell keresni. Ha a fenti mennyiséget minden k-ra kiszámoljuk, akkor az a k lesz a legjobb, amelyre a fenti mennyiség minimális.

28. Algoritmus (Mátrix-szorzás zárójelezése).

for $i=1..n \quad m(i,i):=0$
for $\ell=1..n-1$
  for $i=1..n-\ell$
    $j:=i+\ell$
    $m(i,j):=\min_{k:\, i\leq k<j}(m(i,k)+r_{i-1}\cdot r_k\cdot r_j+m(k+1,j))$
    $h(i,j):=\operatorname{argmin}_{k:\, i\leq k<j}(m(i,k)+r_{i-1}\cdot r_k\cdot r_j+m(k+1,j))$
return$(m(1,n))$

A végén $m(1,n)$ adja meg az összesen minimálisan szükséges szorzások számát, $h(1,n)=k$ pedig az utoljára elvégzendő szorzás helyét. Innen sorban visszafejthetjük a sorrendet, pl. az utolsó előtti két szorzás indexei $h(1,k)$ és $h(k+1,n)$ lesznek.

Lépésszám: $O(n^3)$

5. fejezet

Adatszerkezetek 2.

5.1. Láncolt listák

8. Definíció. *Láncolt lista tartalmaz egy listafejet, ami egy mutató (pointer); valamint listaelemeket, melyek két részből állnak: adat és mutató. A mutató (pointer) mindig a következő listaelem memóriacímét tartalmazza. Az üres mutatót* **nil***-nek nevezzük.*

Előnye, hogy tetszőlegesen bővíthető, amíg van hely a memóriában, valamint hogy a lista elejére mindig be tudunk szúrni $O(1)$ lépésben.

Hátránya, hogy nem lehet benne felező keresést végezni, egy kereséshez sorban végig kell lépkedni a lista elemein. Valamint ha úgy akarunk beszúrni, hogy előtte ellenőrizni kell, hogy a beszúrandó elem szerepel-e a listában, és csak akkor szabad beszúrni, ha nem szerepel, akkor is végig kell nézni az egész listát; viszont ekkor már egyszerűbb a lista végére beszúrni.

Megjegyzés: Sok változatát használjuk. Az adatrész több adatot is tartalmazhat. A Törlés műveletét segítheti, ha ún. *kétszeresen láncolt* listát használunk, amikor minden listaelemhez két pointer tartozik, az egyik a következő, a másik a megelőző listaelemre mutat. Ekkor egy kívülről rámutatott elemet is tudunk konstans időben törölni.

5.2. Bináris keresőfa

Szótárat általában bináris keresőfában, vagy hasheléssel (lásd később) tárolunk.

9. Definíció. *Adott egy U rendezett univerzum. Egy $S \subseteq U$ halmazt szótárnak nevezünk. A bináris keresőfa egy olyan bináris fa, amelynek csúcsaiban rekordok vannak, amelyek kulcsa a szótár egy eleme (minden v csúcsra $K(v) \in S$). Ezenkívül teljesül rá a keresőfa tulajdonság: az összes v csúcsra, a v bal gyerekének minden u leszármazottjára igaz, hogy $K(u) < K(v)$ és a v jobb gyerekének minden w leszármazottjára igaz, hogy $K(w) > K(v)$.*

A három fő szótár-műveletet akarjuk megvalósítani: Keresés, Beszúrás és Törlés. A Beszúrás egyszerű végrehajthatósága miatt – amit mindig egy Keresés után akarunk végrehajtani – a Keresésnél nem elég, hogy ha az elem benne van a szótárban,

akkor megadjuk a helyét, hanem arra is szükségünk lesz, hogy ha nincs benne, akkor megadjuk, hogy „hol nincs" (azaz ha be kell szúrnunk, akkor hová fog kerülni).

Egy gyökeres bináris fát általában csúcsonként 3 pointerrel tárolunk, $p(v)$ mutat v szülőjére, $bal(v)$ a baloldali gyerekére, $jobb(v)$ pedig a jobboldali gyerekére (ha bármelyik nem létezik, akkor a pointer értéke **nil**).

29. Algoritmus (Keresés bináris keresőfában).

Keresés(x, r)**:** /* *Az r gyökerű fában keressük az x kulcsot*

$v := r$

repeat

if $x = K(v)$ **then return(van,** v**, bal)** /* *Benne van a szótárban, v: melyik csúcsban van, bal: ennek itt nincs jelentősége.*

if $x < K(v)$ /* *Ha x kisebb az aktuális csúcsban lévő kulcsnál, akkor*

if $bal(v) \neq$ **nil then** $v := bal(v)$ /* *Ha létezik $bal(v)$, oda lépünk.*

else return(nincs, v**, bal)** /* *Egyébként nincs benne, a visszaadott v és bal jelzi, hogy ha be akarjuk venni, akkor hová kell.*

else /* *Ha pedig $x > K(v)$*

if $jobb(v) \neq$ **nil then** $v := jobb(v)$

else return(nincs, v**, jobb)**

Lépésszám: O(a fa mélysége)

30. Algoritmus (Beszúrás bináris keresőfába).

Beszúrás(b, r)**:**

Elvégzünk egy Keresés(b, r)-et. Ha az első visszaadott érték „van", akkor nem kell beszúrnunk. Egyébként, ha második érték u és a harmadik $irany$ (ami „bal" vagy „jobb"), akkor létrehozzuk u-nak az $irany$ szerinti gyerekét, és oda rakjuk be az új rekordot.

Lépésszám: O(a fa mélysége)

31. Algoritmus (Törlés bináris keresőfában).

Törlés(t, r)**:**

A t kulcsú rekordot kell törölni az r gyökerű fából, ha benne van. Először természetesen egy Keresés(t, r)-et végzünk el, ha nincs benne, leállunk, különben a t-t tartalmazó csúcs legyen x.

1. Eset: x-nek maximum egy gyereke van. Ekkor x úgy törölhető, hogy a gyereke (ha van) feljön a helyére. (Keresőfa tulajdonság megmarad.)

2. Eset: x-nek két gyereke van. Ekkor x-ből egyet balra lépünk, majd amíg lehet jobbra, így elérünk egy y csúcsba. Megjegyzés: $K(y)$ lesz a szótárban a $K(x)$-et közvetlenül megelőző elem.

   Ezután felcseréljük x és y tartalmát, majd az y csúcsot (melynek nincs jobb gyereke) az első eset szerint töröljük.

Lépésszám: O(a fa mélysége)

Megjegyzés: A lépésszámok mind a fa mélységével arányosak. Erről viszont semmit nem tudunk, ha pl. az elemeket nagyság szerint rendezve szúrjuk be, akkor n lesz. Ezért a műveleteket érdemes kicsit bonyolultabban végrehajtani, hogy a fa nagyjából kiegyensúlyozott legyen, azaz a mélysége mindig $O(\log n)$ maradjon.

5.3. AVL-fa

10. Definíció. *Egy bináris keresőfa AVL-fa, ha minden v csúcsra $|m(bal(v))-m(jobb(v))|\leq 1$, ahol $m(w)$ a w csúcs magassága, azaz a belőle lefele vezető leghosszabb út hossza, és $m(\mathbf{nil}) := -1$.*

17. Tétel. *Egy d mélységű AVL-fának $\geq F_{d+3}-1$ csúcsa van, ezért egy n csúcsú AVL-fa mélysége $\leq 1{,}44 \cdot \log n$.*

Keresés, Törlés, Beszúrás: ugyanúgy, mint eddig, azonban a Beszúrás és Törlés művelet elvégzése után ellenőrizni kell, hogy teljesül-e az AVL-fa tulajdonság. Ha elromlik a tulajdonság, akkor helyre kell állítani (ehhez tároljuk minden csúcsban a magasságát). Ezekhez a Billentés nevű, minden bináris keresőfában értelmezett belső műveletet fogjuk használni.

32. Algoritmus (Billentés bináris keresőfában).

Bill(y):

Ha y az x nevű szülőjének bal gyereke, akkor y-t rakjuk x helyére, x lesz y jobb gyereke, az y jobb gyereke pedig x bal gyereke lesz. (Ha y jobb gyerek, akkor szimmetrikusan járunk el.)

Könnyű meggondolni, hogy ez a művelet nem rontja el a keresőfa-tulajdonságot.

33. Algoritmus (Beszúrás AVL-fába).

Beszúrás(s): Beszúrjuk s-et, majd felfelé haladva megkeressük a legközelebbi x ősét, ahol az AVL tulajdonság elromlott; közben persze az érintett csúcsok magasságát átállítjuk. Először $v := s$, és v magasságát 0-ra állítjuk.

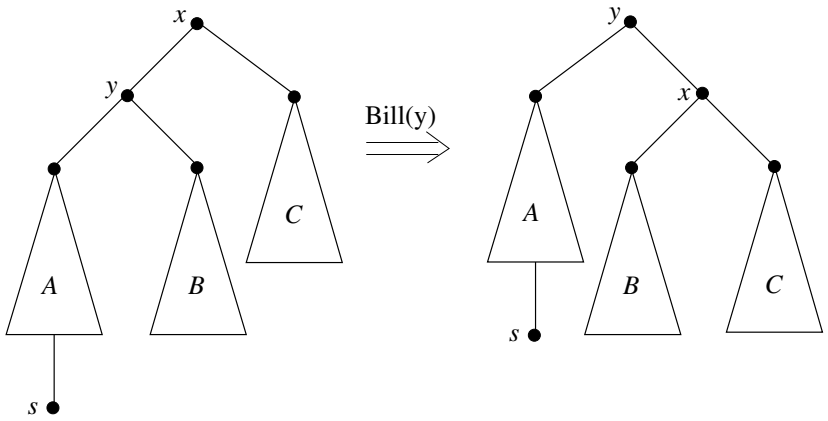
– Ha olyan v gyerekből lépünk fel $u = p(v)$-be, melynek magassága eddig eggyel kisebb volt, mint a testvéréé, akkor most a magassága ugyanaz, mint a testvéréé, ezért megállunk (u magassága nem változik), nincs elromlott csúcs.

– Ha v és testvére magassága eredetileg azonos volt, akkor a Beszúrás miatt v magasságát eggyel növeltük, így most a magassága eggyel nagyobb, mint a testvéréé. Ezért u magasságát is növelni kell, tehát $v := u$, majd növeljük eggyel v magasságát. Ekkor folytatnunk kell felfelé, kivéve ha a v a gyökér.

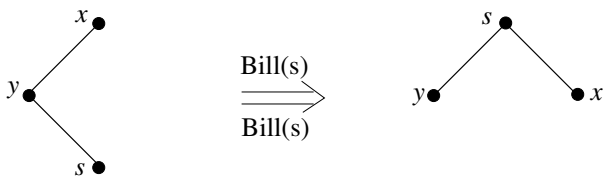
– Ha pedig v magassága nagyobb volt eredetileg, mint a testvéréé, akkor most a magassága kettővel nagyobb. Ekkor $x = u$ az elromlott csúcs.

Majd x-ben egy szimpla vagy egy dupla Billentéssel a tulajdonság helyreállítható az alábbi módon:

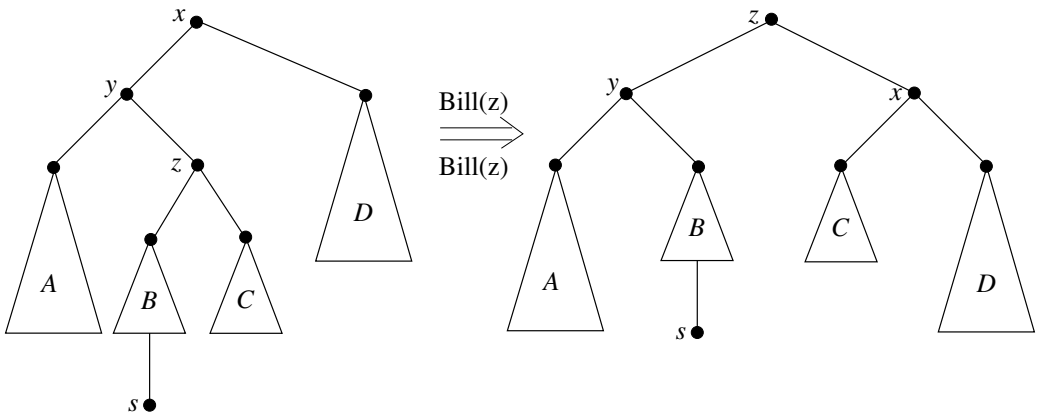
1. eset: s az x bal-bal, vagy jobb-jobb unokájának a leszármazottja



2a. eset: s az x bal-jobb, vagy jobb-bal unokája



2b. eset: s az x bal-jobb, vagy jobb-bal z unokájának valódi leszármazottja



Nyilvánvaló, hogy egy beszúrásnál az x elem helyének megkeresése legfeljebb $\lceil 1{,}44 \cdot \log n \rceil$ lépkedést, így $O(\log n)$ időt vesz igénybe, ezért az egész művelet végrehajtása legfeljebb $\lceil 1{,}44 \cdot \log n \rceil$ magasság átállítással, valamint legfeljebb 2 Billentéssel jár. A megtalált x felett már nem kell állítani magasságot sem, mert a Billentések után azok magassága ugyanaz lesz, mint a beszúrás előtt volt.

Törlés(s): Hasonlóan megy. Legfeljebb $1{,}44 \cdot \log n$ Billentéssel megoldható (lehet, hogy a gyökérig vezető út mentén végig kell billegtetni).

Lépésszám: $O(\log n)$

5.4. Sorok, vermek, gráfok

A **sor** egy olyan dinamikus halmaz, amelyben a legrégebben beérkezett elemet vehetjük ki (FIFO = First In First Out).

11. Definíció. *Sor műveletei:*
UJSOR(S, n)*: létrehozunk egy új,* S *nevű sort* n *mérettel,*
$S \leftarrow u$*: u-t berakjuk az* S *végére,*
$v \leftarrow S$*: v-t kivesszük az* S *elejéről,*
$S \neq \emptyset$*: S sor nem üres-e (feltétel).*

Műveletek megvalósítása: Cél, hogy konstans időben el tudjunk végezni minden műveletet.
A sor méretét meghatározó n érték lehet ismeretlen, ekkor nem tudunk jobbat, mint hogy a sort láncolt listában tároljuk egy ráadás listavég pointerrel. Ha ismerünk valami felső becslést a sor „méretére", akkor jobbat is tudunk:

A) Ha n felső becslés az $S \leftarrow v$ műveletek számára:
ÚJSOR(S, n):
$ST[1:n]$ legyen egy tömb
ELEJE:= 1; VÉGE:= 0

$S \neq \emptyset$: ELEJE $\leq$ VÉGE

$S \leftarrow v$:
VÉGE++
$ST(\text{VÉGE}) := v$

$u \leftarrow S$:
$u := ST(\text{ELEJE})$
ELEJE++

B) Ha csak egy $n \geq |S|$ felső becslést tudunk:
$n+1$ méretű tömbben ciklikusan oldjuk meg a műveleteket.

Verem: Olyan dinamikus halmaz, amelyben a legutoljára beszúrt elemet vehetjük ki (LIFO = Last In First Out).

12. Definíció. *Verem műveletei:*
ÚJVEREM(S, n)*: létrehozunk egy új,* S *nevű vermet* n *mérettel,*
$S \leftarrow u$*: u-t berakjuk az* S *tetejére,*
$v \leftarrow S$*: v-t kivesszük az* S *tetejéről,*
$S \neq \emptyset$*: S verem nem üres (feltétel).*

A verem megvalósítása hasonló, itt a második eset is könnyen megvalósítható és nem kell az ELEJE mutató (mivel értéke mindig 1).

13. Definíció. *A* $G = (V, E)$ *gráf, (ahol* $V = \{1,2 \ldots, n\}$*, amit a továbbiakban végig felteszünk), szomszédsági mátrixa a következő* n*-szer* n*-es mátrix:*

$$A(i,j) = \begin{cases} 1, & \text{ha } ij \in E \\ 0, & \text{ha } ij \notin E. \end{cases}$$

Irányított és irányítatlan gráfok esetében is használható. Irányítatlan gráf esetén ez a mátrix szimmetrikus.

Gráfoknál végig használjuk a csúcsszámra az $n = |V|$ és az élszámra az $m = |E|$ jelöléseket.

14. Definíció. *Az éllistás megadásnál a* $G = (V, E)$ *gráf minden csúcsához egy láncolt lista tartozik, a listafejeket tömbben tároljuk. Az* $i \in V$ *csúcs listájában tároljuk az* i*-ből kimenő éleket (az adat részben a végpont), és ha az éleken súly (vagy költség vagy hossz) is adott, akkor azt is (ekkor az adatrész* 2 *mezőből áll). Irányítatlan gráfnál minden élet mindkét végpontjánál tárolunk.*

Élsúlyozott gráfoknál az éllistás megadás egyszerűbb (kivéve, ha minden súly pozitív, ekkor a mátrixban az 1 helyett az élsúly szerepel). Ez kiváltképp igaz, ha egy élre több szám van írva, pl. súly, kapacitás, hossz és költség, ekkor az éllistánál az adatrész 5 részből áll, első rész a másik végpont, a továbbiak a 4 érték.

Gráfokkal foglalkozó algoritmusokban a leggyakoribb konstrukció a következő:

for $u\underline{v} \in E$

Ez azt jelenti, hogy egy már rögzített u csúcsra végigmegyünk azokon a v csúcsokon, amikbe vezet u-ból él.

Szomszédsági mátrix esetén egy ilyen ciklus végrehajtási ideje n, éllistás tárolásnál az u csúcs fokszáma (irányított gráfok esetén az u csúcs kifokszáma).

6. fejezet

Alapvető gráfalgoritmusok

A gráfok rendkívül gyakran használt struktúrák, nagyon sok feladat modellezésében jól használhatóak. A gráfokon definiált feladatokat megoldó algoritmusok alapvető szerepet játszanak a számítástudományban. Számos érdekes probléma fogalmazható és oldható meg gráfok segítségével. Ebben a részben néhány alapvető problémát vizsgálunk.

6.1. Szélességi keresés/bejárás

Első feladatunk: döntsük el egy gráfról, hogy összefüggő-e.

Megjegyzés: Ez az algoritmus nagyon sok más gráfalgoritmus alapját képezi.

Adott $G = (V, E)$ irányított vagy irányítatlan gráf és egy kitüntetett s kezdő csúcs. Ebből indulva a szélességi keresés módszeresen megvizsgálja G éleit. És így rátalálunk minden s-ből elérhető csúcsra.

Meglátogatjuk tehát s-et, majd ennek a csúcsnak az összes szomszédját. Aztán ezen szomszédok összes olyan szomszédját, ahol még nem jártunk, és így tovább. Általános lépésként vesszük a legrégebben meglátott, de még nem átvizsgált u csúcsot, és átvizsgáljuk: minden, eddig nem látott szomszédját látottnak jelölünk, és „elraktározzuk”, hogy majd még át kell vizsgálnunk.

Faélnek azokat az éleket hívjuk, melyek megvizsgálásukkor még be nem járt pontba mutatnak. Tehát amikor egy már meglátogatott x csúcs szomszédait vizsgáljuk, és mondjuk y-t még nem láttuk, akkor az egyre növekvő fához hozzávesszük az xy élet.

Egy csúcs három különböző állapotban lehet:

I. Nem látott.

II. Látott, de még nem átvizsgált.

III. Átvizsgált.

6.1.1. Összefüggőség tesztelése szélességi kereséssel

34. Algoritmus (Szélességi keresés – összefüggőség).

SZK(G)**:**

for $i = 1..n$ $L(i) := 0$ /* $L(u) = 0$, *ha az u csúccsal még nem találkoztunk*
$L(s) := 1$; $S \leftarrow s$ /* *Az s csúcsot látottnak jelöljük és betesszük a sorba.*
while $S \neq \emptyset$ /* *Amíg a sor ki nem ürül:*
$u \leftarrow S$ /* *A sor első elemét (ez ugyebár a legrégebben várakozó elem) u-ba tesszük.*
for $u\underline{v} \in E$ /* *Sorban megvizsgáljuk az u csúcs v szomszédait.*
if $L(v) = 0$ **then** $L(v) := 1$; $S \leftarrow v$ /* *Ha még nem láttuk, akkor látottnak jelöljük és betesszük a sorba.*

Ez bejárja a gráfot (pontosabban az s csúcs komponensét, ha nem összefüggő), de nem ad outputot. Azonban innen már könnyű az outputot előállítani:

ÖF:$= 1$ /* *Előállítjuk az outputot.*
for $i = 1..n$
if $L(i) = 0$ **then** ÖF:$= 0$
print ÖF

Lépésszám:

Amennyiben az input szomszédsági mátrixszal volt megadva, akkor $O(n^2)$. Ha éllistával, akkor $O(\sum_{u \in V}(d(u)+1)) = O(2 \cdot m+n)$, tehát $O(m+n)$, ami $= O(m)$, ha nincs a gráfban izolált csúcs.

Megjegyzés: Itt találkozunk először azzal, hogy néha az túl durva becslés a lépésszámra, ha a külső ciklus lefutásainak számát beszorozzuk a belső ciklus maximális lépésszámával.

Ennél finomabb, ha a belső ciklus i. lefutásának lépésszáma $c \cdot \ell_i$; és a $c\sum(\ell_i+1)$ felső becslést használjuk.

18. Tétel. *A szomszédsági mátrixszal adott gráf összefüggőségének eldöntéséhez kell* $\binom{n}{2}$ *input olvasás.*

Ezt a nevezetes tételt itt nem bizonyítjuk.

19. Tétel. *Éllistával adott gráf összefüggőségének eldöntéséhez kell legalább* m *olvasó lépés.*

Tehát azt állítjuk, hogy az összesen $2m$ listaelemből legalább a felét el kell olvasni. Feltesszük, hogy $n \leq m \leq n(n-2)/4$ és hogy n és m párosak. Álljon a G gráf két komponensből, mindkettőnek $n/2$ csúcsa és $m/2$ éle van. Ha egy algoritmus helyesen dönt a G gráf összefüggőségéről, akkor legalább az egyik komponens csúcsainak éllistáját végig kell kérdeznie, még akkor is, ha minden listaelemmel együtt megkapja rögtön a mutatót is, így pl. tudja, hogy az a csúcsból kiinduló élek listájának legutolsó eleme-e.

Tegyük fel ugyanis, hogy az algoritmus nem kérdezte végig az első komponens u_1 csúcsának listáját, sem a második komponens u_2 csúcsának listáját. Ekkor azon a módosított inputon, ahol az u_1 listájának végére, valamint az u_2 listájának végére beszúrjuk az u_1u_2 élet, az algoritmus pont ugyanúgy futna le, holott a helyes válasz itt már más (az így módosított gráf már összefüggő). Egy komponensen belül $m/2$ él van, tehát az ide eső csúcsok éllistái hosszának az összege m.

6.1.2. Komponensek meghatározása

35. Algoritmus (Szélességi keresés – komponensek).

SZKK(G)**:**

$k := 0$; **for** $i = 1..n$; $L(i) := 0$
for $i = 1..n$
  if $L(i) = 0$ **then**
    $k{+}{+}; L(i) = k$; $S \leftarrow i$
    while $S \neq \emptyset$
      $u \leftarrow S$
      for $u\underline{v} \in E$
        if $L(v) = 0$ **then** $L(v) := k$**;** $S \leftarrow v$

Ezután u és v egy komponensben van, azaz $u \sim v$, akkor és csak akkor, ha $L(u) = L(v)$.

6.1.3. Legrövidebb utak

Innentől feltesszük, hogy G **összefüggő**.

36. Algoritmus (Szélességi keresés – legrövidebb utak).

SZKL(G, s)**:**

for $i = 1..n$; $p(i) := 0$
$SZ(s) := 0$; $p(s) := s$; $S \leftarrow s$
while $S \neq \emptyset$
  $u \leftarrow S$
  for $u\underline{v} \in E$
    if $p(v) = 0$ **then** $S \leftarrow v$**;** $SZ(v) := SZ(u)+1$; $p(v) := u$

20. Tétel. *Tegyük fel, hogy G összefüggő. A szélességi keresés lefutása után:*
a) $\big\{\{p(v), v\} \mid v \neq s\big\}$ *élek feszítőfát adnak,*
b) $uv \in E \to |SZ(u) - SZ(v)| \leq 1$,
c) $\forall u \in V$ *a legrövidebb* $s \rightsquigarrow u$ *út hossza* $SZ(u)$,
d) $\forall u \in V$ *az egyik legrövidebb* $s \rightsquigarrow u$ *út:* $s, \ldots, p(p(u)), p(u), u$,
e) $\forall v \in V$ *csúcsra a végén* $p(v) > 0$.

6.1.4. Kétszínezés

Feladat: A gráf csúcsait ki akarjuk színezni pirossal és kékkel úgy, hogy azonos színűek között ne menjen él, amennyiben ez lehetséges. Ha nem lehetséges, akkor írjuk ki a gráf egy páratlan körét.

37. Algoritmus (Szélességi keresés – kétszínezés).

SZKkétsz(G)**:**
  for $i = 1..n;\ p(i) := 0$
  $SZ(1) := 0;\ p(1) := 1;\ S \leftarrow 1$
  while $S \neq \emptyset$
    $u \leftarrow S$
    for $u\underline{v} \in E$
      if $p(v) = 0$ **then** $S \leftarrow v;\ SZ(v) := SZ(u) + 1;\ p(v) := u$
      else if $SZ(u) = SZ(v)$ **then PTLANKÖR**(u, v)

Ha nem kerültünk a PTLANKÖR eljárásba, akkor a következő egy jó kétszínezés: v legyen piros, ha $SZ(v)$ páros, különben pedig kék. Ha ez mégsem egy jó kétszínezés, akkor be fogunk kerülni a PTLANKÖR eljárásba. Itt meg kell találni egy páratlan kört és kiírni.

PTLANKÖR(u, v)**:**
  ÚJSOR(Z, n); **ÚJVEREM**(W, n)
  while $u \neq v$
    $W \leftarrow u;\ Z \leftarrow v$
    $u := p(u);\ v = p(v)$
  print u
  while $W \neq \emptyset$
    print $y \leftarrow W$
  while $Z \neq \emptyset$
    print $y \leftarrow Z$

Lépésszám: $O(m)$, ha a gráf éllistával adott; és $O(n^2)$, ha szomszédsági mátrixszal.

6.1.5. Erős összefüggőség

21. Tétel. *Legyen G egy irányított gráf. Egy v csúcsot meglátunk a szélességi keresés során akkor és csak akkor, ha s-ből v-be létezik irányított út. Ekkor is legrövidebb irányított utakat találunk.*

15. Definíció. *A G irányított gráf gyengén összefüggő, ha az irányítástól eltekintve összefüggő.*

A G irányított gráf erősen összefüggő, ha $\forall x, y \in V$ csúcspárra létezik $x \rightsquigarrow y$ (irányított) út.

Erős összefüggőség eldöntése:

- Szélességi keresés az 1 csúcsból.
- A $\overleftarrow{G}$ fordított gráf előállítása. Ez könnyen megy $O(m)$ lépésben, ha G éllistával adott. Ha pedig szomszédsági mátrixszal, akkor vagy transzponáljuk a mátrixot, vagy csak ezt szimuláljuk: az u csúcs szomszédain úgy megyünk végig, hogy a mátrix u. **oszlopán** megyünk végig.
- Ebben újabb szélességi keresés az 1 csúcsból.

A gráf akkor és csak akkor erősen összefüggő, ha mindkét keresésnél minden csúcsot megláttunk.
Lépésszám: $O(m)$

6.2. Prim algoritmusa

Feladat: Egy minimális költségű feszítőfát szeretnénk találni egy összefüggő irányítatlan $G = (V, E)$ gráfban, ahol adott egy $c\colon E \to \mathbb{R}$ költségfüggvény az éleken.

Megjegyzés: Egy legolcsóbb telefonhálózatot szeretnénk építeni úgy, hogy minden városból minden városba lehessen telefonálni. Két város között él vezet, ha lehet építeni közvetlen vonalat, a költsége az, hogy mennyiért. Ez egy fontos alapfeladat, több megoldás ismert rá.

Az alábbi (eredetileg Jarník cseh matematikus írta le először 1930-ban, később függetlenül Prim és Dijkstra újra felfedezték) algoritmus a magyaros megoldást részletezi. Feltesszük, hogy Budapest az 1-es számú csúcs. Először megépítjük a Budapestre illeszkedő legolcsóbb élet. Később mindig vesszük azon városok S halmazát, ahová már lehet Budapestről telefonálni, és az S-ből kilépő élek közül építjük meg a legolcsóbbat. A nehézséget az jelenti, hogy akár cn^2 él is kiléphet S-ből, ezekből gyorsan kell megtalálnunk a legolcsóbbat.

S jelöli azon csúcsok halmazát, amiket már elérünk az 1 csúcsból a T-ben, P pedig $V-S$ azon csúcsait tartalmazza, melyeknek van szomszédja S-ben; M jelöli a maradék csúcsokat ($M = V-S-P$).

T lesz a felépítendő feszítőfa élhalmaza. A $c(T) = \sum_{e \in T} c(e)$ mennyiséget akarjuk minimalizálni a T feszítőfákon. A következő értékeket akarjuk meghatározni és tárolni minden $v \in P$ csúcshoz:

$K(v) = \min\{c(uv) \mid u \in S,\ uv \in E\}$
$p(v) = \operatorname{argmin}\{c(uv) \mid u \in S,\ uv \in E\}$

Ez egy mohó algoritmus, ami erre a feladatra meglepő módon (persze csak első ránézésre meglepő, az érdeklődők olvassanak utána a matroid fogalmának) jól működik.

Az 1-es csúcsból kiindulva növeljük a fát, mindig az S-ből kimenő legolcsóbb éllel.

38. Algoritmus (Prim).

MINFA(G)**:**

$P := \{1\};\ S := \emptyset;\ M := V \setminus \{1\};\ K(1) := 0$
while $P \neq \emptyset$
$u := \operatorname{argmin}\{K(v) \mid v \in P\}$ /* *u a P azon csúcsa, melyre K(u) minimális.*
$S \leftarrow u \leftarrow P$ /* *Áttesszük u-t P-ből az S-be.*
for $uv \in E$
if $v \in P$ **&&** $c(uv) < K(v)$ **then** $K(v) := c(uv);\ p(v) := u$
if $v \in M$ **then** $K(v) := c(uv);\ p(v) := u;\ P \leftarrow v \leftarrow M$

Ezután a T minimális költségű feszítőfa az $\{\{u, p(u)\} \mid u > 1\}$ élekből áll.

Lépésszám: az adatszerkezettől és a megvalósítástól is függ. Ha a minimumokat az első órán tanult módszerrel számoljuk, akkor ennek a résznek a lépésszáma $O(n^2)$, míg a többi sor végrehajtása – a szélességi kereséshez hasonlóan – szomszédsági mátrix esetén $O(n^2)$, éllista esetén $O(m)$. Tehát az input adatszerkezettől függetlenül a lépésszám $O(n^2)$ az egyszerű minimumszámítás esetén.

22. Állítás. *Prim algoritmusa egy minimális költségű feszítőfát ad. Lépésszáma szomszédsági mátrix esetén $O(n^2)$ lépés.*

A jóság bizonyítása: legyen T az algoritmus által adott fa, és $v_1, \ldots, v_n$ a csúcsok nevei az S halmazba kerülés sorrendjében. Azt mondjuk, hogy egy T' feszítőfa k hosszú kezdőszeleten megegyezik a T-vel, ha minden $1 \leq i, j \leq k$ párra igaz, hogy $ij \in T$ akkor és csak akkor, ha $ij \in T'$.

Legyen T^* egy olyan minimális költségű feszítőfa, ami a leghosszabb kezdőszeleten egyezik meg T-vel (legyen ennek hossza k). Ha $k = n$, akkor készen vagyunk.

Indirekt tegyük fel, hogy $k < n$, legyen $v_i v_{k+1}$ a T fa $\{v_1, \ldots, v_k\}$ halmazból kilépő éle. Ekkor a $T^* + v_i v_{k+1}$ tartalmaz egy kört, ennek van legalább egy másik kilépő éle is, például $v_\ell w$. A Prim algoritmus szabálya szerint $c(v_i v_{k+1}) \leq c(v_\ell w)$. Legyen $T' = T^* + v_i v_{k+1} - v_\ell w$. Nyilván ez is egy feszítőfa, ráadásul $c(T') \leq c(T^*)$, tehát ez is optimális. De T' már $k+1$ hosszú kezdőszeleten megegyezik T-vel, ami ellentmond T^* és k választásának.

Ha a gráf szomszédsági mátrixszal van adva, vagy sűrű (valamilyen c konstansra legalább cn^2 éle van), akkor nem várható jobb.

Azt szeretnénk megvizsgálni, hogy ha a G gráf éllistával adott és csak $o(n^2)$ éle van, akkor tudunk-e ennél gyorsabb algoritmust készíteni. Ehhez a fentiek miatt valami ravaszabbat kell csinálni, mint mindig az egyszerű minimum-számítással keresni meg az u csúcsot.

6.3. Kupacok alkalmazása a Prim-algoritmusban

Ha a P-beli csúcsokat a K kulcs szerint kupacban tároljuk, akkor a minimum-keresés és eltávolítás P-ből egy-egy MINTÖRLÉS, de ekkor a kupac karbantartása miatt az utolsó előtti sorban egy-egy KULCS-CSÖKKENTÉS (ez egy új kupacművelet, lásd alább), az utolsó sorban egy-egy BESZÚRÁS kell. Tehát összesen n db MINTÖRLÉS és n db BESZÚRÁS, valamint legfeljebb m db KULCS-CSÖKKENTÉS szükséges. Ha az input éllistával adott, akkor a kupac-műveletek dominálják a lépésszámot. Tehát az összes lépésszám éllista esetén (bináris) kupac használatával $O(m \log n)$.

Ehhez ezt a feladatot kell még megoldanunk: Egy kupac megadott elemének a kulcsát csökkentsük le egy adott mennyiséggel.

39. Algoritmus (KULCS-CSÖKKENTÉS).

KULCS-CSÖKKENTÉS(A, i, Δ): /* *Az A nevű kupac i indexű csúcsában levő rekord kulcsát kell a pozitív Δ-val csökkenteni.*

$A(i) := A(i) - \Delta$
FELBILLEGTET(A, i)

23. Állítás. *A KULCS-CSÖKKENTÉS lépésszáma csak eggyel több a FELBILLEGTET lépésszámánál, tehát $O(\log n)$. Tehát Prim algoritmusának lépésszáma éllistás tárolás esetén, ha kupac-műveletekkel valósítjuk meg, akkor $O(m \log n)$.*

6.3.1. d-ed fokú kupac

16. Definíció. *d-edfokú kupac és megvalósítása. Az $A[0:n-1]$ tömb elemeit egy olyan fa csúcsainak feleltetjük meg, ahol a fa minden csúcsának legfeljebb d gyereke van. A fa gyökerének $A[0]$ felel meg, és az $A[i]$ csúcs gyerekei legyenek az*

$$A[d \cdot i + 1], A[d \cdot i + 2], \ldots, A[d \cdot i + d]$$

elemek (közül azok, melyek indexe kisebb, mint n).

Így $0 < j < n$-re az $A[j]$ csúcs szülőjének indexe $\lfloor (j-1)/d \rfloor$. A fa mélysége $\log_d n$.

Ezt akkor hívjuk d-edfokú kupacnak, ha teljesül rá a kupacrendezettség: azaz minden $v \neq$gyökér csúcsra igaz, hogy $K(\text{szülő}(v)) \leq K(v)$.

A kupac mélysége ekkor tehát csak $\log_d n$ lesz (ami persze $d > 2$ esetén kisebb, mint $\log n$). A d-edfokú kupacban a műveletek a bináris kupac műveleteinek mintájára mennek. A legfontosabb különbség a LEBILLEGTETÉS műveletében van: itt először most a d gyerek közül ki kell választani a legkisebb kulcsút ($d-1$ összehasonlítással), majd az aktuális elem kulcsát ezzel kell összehasonlítani.

Tehát a lépésszámok d-edfokú kupacban:
MINTÖRLÉS: $O(d \cdot \log_d n)$.
BESZÚRÁS: $O(\log_d n)$.
KULCS-CSÖKKENTÉS: $O(\log_d n)$.

Ezek alapján Prim algoritmusának a lépésszáma, ha gráf éllistával adott, és d-edfokú kupacot használunk: $O(d \cdot n \cdot \log_d n + m \cdot \log_d n)$. Ez nagyjából akkor a legjobb, ha d értékét $d^* = \max\left(2, \lceil \frac{m}{n} \rceil\right)$-nek választjuk, ekkor Prim algoritmusának lépésszáma $O(m \log_{d^*} n)$ lesz. Amennyiben $m > n^{1+\varepsilon}$ valamilyen pozitív konstans ε-ra, akkor ez $O(m)$, azaz lineáris idejű.

Megjegyzés: Érdemes meggondolni a $d = 2$ és a $d = 4$ eset viszonyát. Mivel $\log_4 n = \frac{1}{2} \cdot \log n$, ezért a BESZÚRÁS és a KULCS-CSÖKKENTÉS esetén feleannyi összehasonlítást végzünk $d = 4$ esetén, mint a bináris kupacnál. A MINTÖRLÉSnél pedig ugyanannyit, hiszen $2 \log n = 4 \log_4 n$.

Megjegyzés: Persze a d^* választásáról fent leírtak a legrosszabb esetre vonatkoznak, amikor tényleg $c \cdot m$ KULCS-CSÖKKENTÉSt kell végezni, és mindegyiknél a magassággal arányos felbillentés kell. Az átlagos futásidő legjobb csökkentéséhez általában ennél kisebb d értéket érdemes választani.

24. Tétel. *Prim algoritmusának lépésszáma $O(n^2)$; éllistás megadás esetén pedig bináris kupaccal $O(m \log n)$, d-edfokú kupaccal $O(m \cdot \log_d n + d \cdot n \cdot \log_d n)$. Ez $d^* = \max\left(2, \lceil \frac{m}{n} \rceil\right)$ esetén $O(m \log_{d^*} n)$. Még jobb kupaccal (Fibonacci, nem tanultuk) Prim algoritmusának lépésszáma $O(m + n \log n)$.*

7. fejezet

Legrövidebb utak

Feladat: Adott kezdőcsúcsból legrövidebb utat szeretnénk találni minden csúcsba egy élsúlyozott irányított vagy irányítatlan $G = (V, E)$ gráfban. A súlyozatlan esetet már szélességi kereséssel megoldottuk.

7.1. Csúcs-súlyozott eset

Először tekintsük a csúcs-súlyozott esetet. Itt w nemnegatív súlyok adottak a csúcsokon, és egy út „hossza" a rajta szereplő csúcsok súlyainak az összege.

40. Algoritmus (Dijkstra a csúcs-súlyozott esetre).

DIJKSTRA-CS(G, s)**:**

$P := \{s\};\ S := \emptyset;\ M := V \setminus \{s\};\ K(s) := w(s);\ p(s) := s$
while $P \neq \emptyset$
$u := \operatorname{argmin}\{K(v) \mid v \in P\}$ /* *u a P azon csúcsa, melyre $K(u)$ minimális.*
$S \leftarrow u \leftarrow P$
for $u\underline{v} \in E$
if $v \in M$ **then**
$K(v) := K(u) + w(v);\ p(v) := u;\ P \leftarrow v \leftarrow M$

25. Tétel. *Ez az algoritmus helyesen számítja ki a legkisebb csúcs-súlyú utak hosszát. Lépésszáma $O(n^2)$, azonban ha a gráf éllistával van megadva, és az* argmin *műveletet a P halmazon felépített bináris kupac segítségével végezzük (ekkor az utolsó sorban egy kupacba BESZÚRÁS művelet kell), akkor $O(m + n \log n)$.*

7.2. Dijkstra algoritmusa

Ezután azt a nagyon fontos esetet oldjuk meg, amikor a $c(uv)$ élsúlyok („hosszak") nem-negatívak. Itt S azon csúcsok halmaza lesz, amelyekbe már biztosan tudjuk a legrövidebb út hosszát, P továbbra is azon S-en kívüli csúcsok halmaza, ahová megy él S-ből, és M a maradék.

41. Algoritmus (Dijkstra).

DIJKSTRA(G,s):
$P:=\{s\};\ S:=\emptyset;\ M:=V\setminus\{s\};\ K(s):=0;\ p(s):=s$
while $P\neq\emptyset$
$u:=\operatorname{argmin}\{K(v)\,|\,v\in P\}$ /* u a P azon csúcsa, melyre $K(u)$ minimális.
$S\leftarrow u\leftarrow P$
for $u\underline{v}\in E$
if $v\in P$ **&&** $K(u)+c(uv)<K(v)$ **then**
$K(v):=K(u)+c(uv);\ p(v):=u$
if $v\in M$ **then**
$K(v):=K(u)+c(uv);\ p(v):=u;\ P\leftarrow v\leftarrow M$

Az algoritmus lefutását lásd a 14. példán.

17. Definíció. *Egy $s\rightsquigarrow v$ út S-út, ha minden csúcsa S-ben van, kivéve esetleg a v csúcsot.*

26. Tétel. *Dijkstra algoritmusában, ha minden él súlya nem-negatív, akkor minden ciklus végekor:*

Minden $v\in S$-re $K(v)$ a legrövidebb $s\rightsquigarrow v$ út hossza, (és létezik legrövidebb $s\rightsquigarrow v$ út, amely S-út).

Minden $v\in P$-re $K(v)$ a legrövidebb $s\rightsquigarrow v$ S-út hossza.

27. Tétel. *Dijkstra algoritmusának lépésszáma megegyezik a Prim-algoritmus lépésszámával, tehát $O(n^2)$; éllistás megadás esetén pedig bináris kupaccal $O(m\log n)$, d-edfokú kupaccal $O(m\cdot\log_d n+d\cdot n\cdot\log_d n)$. Ez $d^*=\max\left(2,\left\lceil\frac{m}{n}\right\rceil\right)$ esetén $O(m\log_{d^*}n)$. Még jobb kupaccal (Fibonacci, nem tanultuk) Dijkstra algoritmusának lépésszáma $O(m+$ $+n\log n)$.*

Az s-ből egy v csúcsba vezető legrövidebb út ugyanúgy fejthető vissza, mint a szélességi keresésnél, jobbról balra: $s,\ldots,p(p(v)),p(v),v$.

A $\{\{p(v),v\}\mid v\neq s\}$ élek (az összes megtalált legrövidebb út uniója) itt is fát (irányított esetben fenyőt) alkotnak.

Megjegyzés: A következő állítást indukcióval bizonyítva egy alternatív bizonyítást kapunk az algoritmus jóságára: minden ciklusvégkor minden $u\in S$ és $v\in S\cup P$ esetén, ha uv él, akkor $c(uv)\geq K(v)-K(u)$, valamint ha $u=p(v)$, akkor itt egyenlőség van.

7.3. Alkalmazás: legbiztonságosabb út

Feladat: Legbiztonságosabb út megkeresése. Például egy telekommunikációs hálózatban szeretnénk olyan $s\rightsquigarrow t$ összeköttetést találni, amelyre a legnagyobb valószínűséggel nem szakad meg (adott időn belül) a vonal.

Legyenek adottak az élekre a tönkremenés valószínűségei: $p(e)=\Pr($ az e él tönkremegy 1 napon belül). Feltehetjük, hogy $0\leq p(e)<1$ minden e élre. Feltesszük továbbá,

hogy különböző élek tönkremenése független események (ez nem túlságosan valósághű, de így tudjuk csak egyszerűen kezelni a feladatot).

Ekkor egy P útra $\Pr(P \text{ megmarad}) = \prod_{e \in P}(1-p(e))$, az ezen valószínűséget maximalizáló utat szeretnénk megtalálni.

Trükk: Vegyük a fenti kifejezés logaritmusát (a logaritmus szigorúan monoton függvény), és azt maximalizáljuk. A logaritmus maximuma a szigorú monotonitás miatt nyilván ugyanott vétetik fel, ahol az eredeti maximum.

Tudjuk, hogy szorzat logaritmusa a logaritmusok összege, tehát elég egy olyan P utat találni, amelyre a $\sum_{e \in P} \log(1-p(e))$ érték a lehető legnagyobb.

Látszólag bajban vagyunk, mert az út „hosszát" itt maximalizálni kell, de vegyük észre, hogy mivel egynél kisebb számok logaritmusa negatív, ezért minimalizálhatjuk a $\sum_{e \in P} -\log(1-p(e))$ értéket. Ha az élekre a nem-negatív $c(e) = -\log(1-p(e))$ értékeket írjuk, akkor ez pont egy legrövidebb út keresése, tehát erre Dijkstra algoritmusa alkalmazható.

7.4. Alkalmazás: legszélesebb út

Feladat: Keressünk legszélesebb utat, például egy telekommunikációs hálózatban szeretnénk azt az utat megtalálni, amelynek legnagyobb a szabad kapacitása.

Itt $w(e)$ jelölje az e él szélességét. Egy út szélessége: $w(P) = \min_{e \in P} w(e)$, tehát az adott út legkeskenyebb éle fogja az út szélességét megadni, és ezt most nem minimalizálni, hanem értelemszerűen *maximalizálni* szeretnénk. Dijkstra algoritmusának megfelelő átalakításával megoldható a feladat. Három sort kell lecserélni:

$u := \underline{\operatorname{argmax}}\{K(v) \mid v \in P\}$ /* u a P azon csúcsa, melyre $K(u)$ *maximális*.
if $v \in P$ **&&** $\underline{\min(K(u), w(uv)) > K(v)}$ **then**
$K(v) := \underline{\min(K(u), w(uv))}$; $p(v) := u$
if $v \in M$ **then** $K(v) := \underline{\min(K(u), w(uv))}$; $p(v) := u$; $P \leftarrow v \leftarrow M$

7.5. Házépítés – PERT módszer

7.5.1. Topologikus sorrend

18. Definíció. *Egy G irányított gráf aciklikus, ha nincsen benne irányított kör.*

Feladat: El szeretnénk dönteni, hogy egy gráf aciklikus-e.

Megjegyzés: Ez azért fontos, mivel tudjuk, hogy amennyiben negatív súlyokat is megengedünk, akkor a legrövidebb út megtalálása 1000000$-os kérdéssé alakul (ennyit ajánlottak fel az évezred fordulóján az egyik legfontosabb eldöntetlen probléma, a P=?NP megválaszolásáért). Ebből kifolyólag, ahhoz, hogy hatékonyan meg tudjuk oldani a feladatot, fel kell tennünk, hogy a gráf súlyozása olyan, hogy nincs benne negatív összhosszúságú irányított kör. Egy aciklikus gráf minden súlyozása ilyen lesz.

19. Definíció. *A $G=(V,E)$ irányított gráf egy $c: E \to \mathbb{R}$ súlyozása konzervatív, ha G minden irányított C körére $c(C) := \sum_{e \in C} c(e) \geq 0$.*

28. Tétel. *Ha a G irányított gráf aciklikus, akkor*
létezik t nyelő (azaz amelyre $d_{\text{ki}}(t) = 0$),
és létezik s forrás (azaz amelyre $d_{\text{be}}(s) = 0$),
valamint létezik a csúcsoknak topologikus sorrendje: $v_i v_j \in E \to i < j$.

A topologikus sorrendről szóló tétel biztosítja nekünk, hogy egy aciklikus gráfban van forrás, és létezik a gráfnak topologikus sorrendje. Az algoritmus ötlete abban rejlik, hogy először megkeresünk egy forrást, majd ezt kitöröljük, így egy kisebb aciklikus gráfot kapunk, és ezt folytatjuk egészen addig, amíg az összes csúcsot ki nem töröltük. Ez az eljárás egyben egy topologikus sorrendet is fog nekünk adni.

42. Algoritmus (Topologikus sorrend).

ÚJSOR(S, n); $k := 0$;
for $u = 1..n$ $\quad Be(u) := 0$
for $u = 1..n$
  for $u\underline{v} \in E$
    $Be(v)$++ $\quad$ /* *Elsőként megszámoljuk az összes csúcs befokát.*
for $u = 1..n$
  if $Be(u) = 0$ **then** $S \leftarrow u$ $\quad$ /* *A megtalált forrásokat az S sorba tesszük.*
while $S \neq 0$
  $u \leftarrow S$
  k++; $TS(k) := u$ $\quad$ /* *Ez most forrás, ő lesz a topologikus sorrend következő eleme.*
  for $u\underline{v} \in E$ $\quad$ /* *Az u-ból kimenő éleket töröljük.*
    $Be(v)$−−
    if $Be(v) = 0$ **then** $S \leftarrow v$ $\quad$ /* *Ha v forrássá vált, akkor betesszük az S sorba.*
if $k < n$ **then print** „NEM ACIKLIKUS"

Lépésszám: $O(m)$

7.5.2. PERT módszer

Feladat: Házépítés ütemezését és elkészülésének minimális idejét számoljuk ki.

Input: adott egy aciklikus irányított gráf egy darab s=START forrással, egy darab t=KÉSZ nyelővel, a többi csúcsra egy-egy művelet neve és végrehajtási ideje (pl. napokban) van megadva. Egy uv él azt jelenti, hogy az u művelet befejezése után szabad csak elkezdeni a v műveletet.

Először is minden v művelet-csúcsot helyettesítünk egy v^- =(a v művelet kezdete) és egy v^+ =(a v művelet vége) csúccsal, a v végrehajtási idejét a v^-v^+ élre írjuk, és ha uv él volt, berakunk egy u^+v^- élet, melynek hossza 0 lesz (illetve, ha a két adott művelet között van pl. előírt száradási idő, azt is írhatjuk ide). Valamint minden v-re berakunk 0 hosszú sv^- és v^+t éleket.

29. Tétel (PERT MÓDSZER). *A ház elkészüléséhez szükséges minimális idő = a leghosszabb $s \rightsquigarrow t$ út T hosszával.*

Szeretnénk még ún. tartalék-időket is kiszámolni. Pontosabban minden x új csúcshoz (ami tehát most egy művelet kezdete vagy egy művelet vége) még két számot (időpontot) akarunk kiszámolni. Az első az, hogy leghamarabb mikor érhetünk az adott pontba. Könnyű meggondolni, hogy ez pont a leghosszabb $s \rightsquigarrow x$ út hossza lesz. A másik pedig az, hogy legkésőbb mikor kell elérnünk ezt a pontot, hogy a ház még elkészülhessen a terv szerinti időben. Az eddigiek alapján ez pedig T mínusz a leghosszabb $x \rightsquigarrow t$ út hossza lesz.

Vegyük észre, hogyha minden élre az eredetileg ráírt hossz mínusz egyszeresét írjuk, akkor a „leghosszabb út" fogalom éppen átmegy a „legrövidebb út"-ba. Mi ezt fogjuk használni, tehát a feladatunk egy aciklikus gráfban, melynek minden e élére egy tetszőleges $c(e)$ szám van írva, s-ből minden csúcsba megkeresni a legrövidebb utat, valamint minden csúcsból t-be is. Az egyszerűség kedvéért feltesszük, hogy s-ből minden más csúcsba vezet út, és minden csúcsból vezet t-be út. (A házépítési feladatból kapott gráfban azt tettük fel, hogy egyélű út is vezet.)

43. Algoritmus (Aciklikus Dijkstra).

ACIKL-DIJKSTRA(G)**:**

I. A G **gráf csúcsainak topologikus rendezése.** /* *s lesz a $TS(1)$, t a $TS(n)$ csúcs.*

II. $P := \{s\};\ S := \emptyset;\ M := V \setminus \{s\};\ K(s) := 0;\ p(s) := s$
for $i = 1..n$
$u := TS(i)$
$S \leftarrow u \leftarrow P$
for $u\underline{v} \in E$
if $v \in P$ **&&** $K(u)+c(uv) < K(v)$ **then**
$K(v) := K(u)+c(uv);\ p(v) := u$
if $v \in M$ **then**
$K(v) := K(u)+c(uv);\ p(v) := u;\ P \leftarrow v \leftarrow M$

Minden csúcsból t-be pedig úgy keresünk, hogy előállítjuk a $\overleftarrow{G}$ fordított gráfot $O(m)$ időben, majd abban keresünk t-ből legrövidebb utakat. A topologikus sorrendet nem kell újra előállítanunk, az előző sorrend megfordítottja megfelel.

Lépésszám: $O(m)$

7.6. Bellman–Ford-algoritmus

Feladat: Legrövidebb utak problémájának megoldása abban az általános esetben, amikor a gráfunk irányított, az élek között negatív súlyúak is lehetnek, de a súlyozás konzervatív, azaz nincs a gráfban negatív összsúlyú irányított kör. És ha az inputban a súlyozás mégsem konzervatív, akkor ezt vegyük észre, és adjunk vissza bizonyítékként egy negatív összsúlyú kört.

Ha a súlyozás tetszőleges lehetne, akkor a feladat nehézzé válna (tartalmazza az NP-nehéz leghosszabb út feladatot a csupa -1 súlyozás esetén).

30. Állítás. *Ha G erősen összefüggő irányított gráf, c konzervatív súlyozás az éleken, $s, t \in V$, akkor létezik legkisebb összsúlyú $s \rightsquigarrow t$ séta. Ha Q egy legkisebb összsúlyú $s \rightsquigarrow t$ séta, akkor létezik egy P $s \rightsquigarrow t$ út is, melyre $c(P) = c(Q)$.*

44. Algoritmus (Bellman–Ford).

for $i = 1..n$ $p(i) := 0$; $K(i) := +\infty$
$K(s) := 0$; $p(s) := s$ /* *Inicializáljuk.*
for $i = 1..n-1$
  for $u = 1..n$
    for $u\underline{v} \in E$
      if $K(u)+c(uv) < K(v)$
        then $K(v) := K(u)+c(uv)$; $p(v) := u$
for $u = 1..n$
  for $u\underline{v} \in E$
    if $K(u)+c(uv) < K(v)$
      then $p(v) := u$; **return('NEM KONZERVATIV',** K, p, v**)**
return('KONZERVATIV', K, p, s**)**

31. Állítás. *Az i. ciklus végén minden v-re, ha $K(v)$ véges, akkor $K(v)$ egy létező $s \rightsquigarrow v$ séta összsúlya, és nem nagyobb, mint a legkisebb összsúlyú, legfeljebb i élből álló $s \rightsquigarrow v$ sétának az összsúlya (akkor is, ha c nem konzervatív).*

32. Tétel. *Ha G irányított erősen összefüggő gráf és c konzervatív, akkor Bellman–Ford algoritmusa minden csúcsra kiszámítja az oda vezető legkisebb összsúlyú út hosszát $O(n \cdot m)$ időben. Ha c nem konzervatív, akkor ezt az utolsó ellenőrző ciklusban észrevesszük, és a megkapott v csúcsból visszafelé lépkedve a p pointerek mentén a negatív kört is megtalálhatjuk.*

Megjegyzés: Ha G nem erősen összefüggő, akkor az algoritmus végén $K(v)$ véges akkor és csak akkor, ha van út s-ből v-be, ekkor persze $K(v)$ egy legrövidebb út hossza lesz. Csak akkor találunk negatív kört, ha van olyan, ami s-ből elérhető.

33. Tétel. *Ha G irányított gráf, c konzervatív, és G-ben minden csúcs elérhető s-ből, akkor a visszaadott $K : V \to \mathbb{Z}$ egy megengedett potenciál lesz, ami azt jelenti, hogy minden uv élre $c(uv) \geq K(v) - K(u)$ teljesül.*

7.7. Floyd–Warshall-algoritmus

Mi a helyzet, ha minden csúcsból akarunk minden csúcsba legrövidebb utat keresni?

Egy sűrű irányított gráfban ezeket leghatékonyabban az alábbi algoritmussal lehet. Egy élsúlyozott gráfot egy M mátrixszal adunk meg, ahol $M(i,j)$ értéke 0, ha $i = j$; $M(i,j) = c(ij)$, ha ij él, és $+\infty$ egyébként.

45. Algoritmus (Floyd–Warshall).

for $k = 1..n$
  for $i = 1..n$
    for $j = 1..n$
      if $M(i,k) + M(k,j) < M(i,j)$
        then $M(i,j) := M(i,k) + M(k,j)]$

34. Tétel. *A Floyd-Warshall-algoritmus $O(n^3)$ lépésben megkeresi a legrövidebb út hosszát minden egyes csúcspár között, ha a gráf erősen összefüggő és c konzervatív.*

A k. ciklus végén minden i, j párra $M(i,j)$ a legrövidebb olyan $i \rightsquigarrow j$ út hossza lesz, melynek belső csúcsai elemei a $\{1,2,\ldots,k\}$ halmaznak.

A c súlyozás akkor és csak akkor konzervatív, ha az algoritmus végén a főátlóban nincs negatív szám.

Megjegyzés: Az eljárás végén $M(i,j) = +\infty$ akkor és csak akkor, ha nem létezik út i-ből j-be.

7.8. Johnson algoritmusa

Továbbra is minden csúcsból minden csúcsba keressük a legrövidebb utat, de feltesszük, hogy a gráf éllistával adott, és nincsen nagyon sok (azaz $c \cdot n^2$) éle. Először feltesszük, hogy a gráf erősen összefüggő. Ha az élsúlyok nem-negatívak, akkor n darab Dijkstra jó lesz, $O(n^2 \cdot \log n + n \cdot m)$ időben. Ha a súlyozás csak konzervatív, akkor az n db Bellman–Ford jó lenne, de csak $O(n^2 \cdot m)$ időben, ami rosszabb, mint a Floyd-Warshall lépésszáma. Ezért először egy Bellman–Ford, majd n db Dijkstra futtatásával oldjuk meg a feladatot.

- Futtassunk le egy Bellman–Ford-algoritmust, és $\pi(v) := K(v)$ jelölje az általa adott megengedett potenciált minden v csúcsra.
- Módosítsuk az élek súlyát: $c'(uv) := c(uv) + \pi(u) - \pi(v)$ minden uv élre. (Ezután $c'(uv) \geq 0$.)
- Keressünk minden csúcsból legrövidebb utakat a Dijkstra algoritmus segítségével a c' súlyokat használva.
- Ha ez alapján x-ből y-ba legrövidebb út P, és hossza $d'(x,y)$, akkor az eredeti gráfban x-ből y-ba a legrövidebb út szintén P, és hossza $d(x,y) = d'(x,y) - \pi(x) + \pi(y)$.

Lépésszám: $O(n^2 \cdot \log n + n \cdot m)$

Megjegyzés: Ha G nem erősen összefüggő, akkor a Bellman–Ford csak az s csúcsból elérhető csúcsokra számol megengedett potenciált, és még az is előfordulhat, hogy konzervatívnak minősíti a súlyozást, pedig van egy s-ből nem elérhető negatív kör. Ezen úgy segíthetünk, hogy odaképzelünk egy új s' csúcsot, amiből 0 költségű él megy minden eredeti csúcsba. Ezt az s' csúcsot nem kell hozzáadnunk a gráfhoz, könnyű szimulálni, hogy mi történne, ha ott lenne: úgy indítjuk a Bellman–Ford-algoritmust, hogy az inicializálásnál minden i csúcs $K(i)$ értékét nullára állítjuk.

8. fejezet

Hashelés

Feladat: U rendezett univerzum adott elemeiből kell olyan S szótárat létrehozni, amelyben gyorsan el tudjuk végezni a KERES és BESZÚRÁS műveleteket.

Keresünk egy $h: U \to [0,1,...,M-1]$ függvényt, amely az U univerzum egyes elemeinek a „helyét" számolja ki a memóriában, (ahol tipikusan: $|U| \gg M$). Olyan h függvényt szeretnénk keresni, melyre igaz lesz az, hogy $\Pr(h(K)=i) \approx 1/M$ minden $0 \leq i < M$-re és arra az eloszlásra, amely szerint az input kulcsok érkezni fognak. Ilyen h függvény például egy jó álvéletlenszám-generátor, melynek az U elemei megadhatók „mag"-ként.

20. Definíció. *Hash függvénynek nevezünk egy $h: U \to [0,1,\ldots M-1]$ függvényt. Általában akkor jó, ha elég véletlenszerű. Feltesszük, hogy olyat találtunk, amelyre $\Pr(h(K)=i) \approx \frac{1}{M}$ minden i-re, ahol a valószínűség úgy értendő, hogy az ismeretlen, input által definiált eloszlás szerint választunk egy véletlen $K \in U$ kulcsot.*

Ha két különböző kulcshoz ugyanazt az értéket rendeli a függvényünk, akkor ezt ütközésnek hívjuk. Ütközéseket kétféleképpen tudunk feloldani.

21. Definíció. *Ha $K_1 \neq K_2$ és $h(K_1) = h(K_2)$, akkor ezt ütközésnek hívjuk. (A születésnap paradoxon miatt sok ilyen várható, ha $M < |S|^2/2 = \frac{N^2}{2}$.)*

8.1. Vödrös vagy láncolt hashelés

Itt M darab láncolt listát használunk, a listafejek egy $L[0:M-1]$ tömbben vannak. KERES(K) esetén kiszámítjuk $h(K)$ értékét, majd végignézzük a $h(K)$ indexű listát. BESZÚRÁS(K) esetén is így kezdünk, és ha nem találtuk, akkor a lista végére illesztjük.

35. Tétel. *A sikeres keresés várható lépésszáma a vödrös (láncolt) hashelésnél: $1+\frac{\alpha}{2}$, ahol $\alpha := \frac{N}{M}$, (itt N a tárolt szótár mérete, M pedig a láncolt listák száma).*

A sikertelen keresés, illetve a beszúrás várható lépésszáma a vödrös hashelésnél: $1+\alpha$.

8.2. Nyílt címzés

22. Definíció. *Nyílt címzés: a szótár elemeit egy M méretű tömbbe rakjuk, de a K kulcs nem határozza meg előre, hogy hova kerül. Csak $\alpha < 1$ esetén van értelme.*

A nyílt címzésnél a tárolásra egy $T[0:M-1]$ tömböt használunk. Az elnevezés arra utal, hogy a K kulcs nem határozza meg előre pontosan, hogy melyik helyre fogjuk a K kulcsot berakni. Csak a legegyszerűbb fajtáját tárgyaljuk.

Lineáris hashelés. BESZÚRÁS(K) esetén először a $h(K)$ helyre próbáljuk berakni, ha az üres, akkor be is rakjuk. Ha foglalt, akkor a $h(K)-1$ hellyel próbálkozunk. És így tovább, tehát a keresési sorozatunk ez lesz: $h(K), h(K)-1, h(K)-2, \ldots, 0, M-1, M-2, \ldots, h(K)+1$. Ezen haladva vagy megtaláljuk K-t, és akkor nem kell berakni, különben az első üres helyre rakjuk. A KERES(K) műveletnél is ugyanezen sorozat szerint vizsgáljuk a T elemeit, ha egy üres helyhez érünk, akkor tudjuk, hogy K nem volt a táblázatban. Emlékeztető: $\alpha := \frac{N}{M}$ (ahol N a tárolt szótár mérete, M pedig a T tömb mérete, és feltettük, hogy $\alpha < 1$).

23. Definíció. *Lineáris hashelés:*
sorba próbáljuk a $h(K), h(K)-1, \ldots, 1, 0, M-1, M-2, \ldots$ *helyeket, és a beszúrandó* K *kulcsot az első üres helyre rakjuk be. Keresésnél ugyanezeket a helyeket nézzük végig, ha üres helyhez értünk, akkor tudjuk, hogy a* K *kulcs nincs a táblában.*

Az elemzésben a nagy ordó az $\alpha \to 1$ esetére utal.

36. Tétel. *Lineáris hashelésnél*
a sikeres keresés várható lépésszáma: $\frac{1}{2} \cdot (1 + \frac{1}{1-\alpha}) = O(\frac{1}{1-\alpha})$;
a sikertelen keresés és a beszúrás várható lépésszáma: $\frac{1}{2} \cdot (1 + (\frac{1}{1-\alpha})^2) = O((\frac{1}{1-\alpha})^2)$.

Ha a táblázat félig van tele, akkor a sikeres keresés várható lépésszáma 1,5, a sikertelené 2,5, ha pedig kétharmadig, akkor a sikeres keresés lépésszáma 2, a sikertelené 5.

8.3. Univerzális hash függvény család

A modernebb technikák alapja az úgynevezett univerzális hash függvény család. Azt szeretnénk, ha a hash függvényünk minél véletlenszerűbb lenne, de persze van még két fontos szempont: az egész folyamat során egy adott K kulcsra a $h(K)$ értéknek mindig ugyanannak kell lennie, valamint lehetőleg legyen a h függvény gyorsan számolható.

$\mathcal{H}$ legyen $\{h : U \to [0, M-1]\}$ hash függvények egy halmaza.

24. Definíció. $\mathcal{H}$ *univerzális, ha* $\forall K_1 \neq K_2 \in U$*-ra:* $\Pr_{h \in \mathcal{H}}\big(h(K_1) = h(K_2)\big) \leq \frac{1}{M}$.

Csak egy példát mutatunk be. Feltesszük, hogy p elég nagy prímszám, azaz $U \subseteq \{0,1,2,\ldots,p-1\}$. Minden $1 \leq a \leq p-1$-re és $0 \leq b \leq p-1$-re definiáljuk a $h_{a,b}$ hash függvényt így:

$$h_{a,b}(K) := ((aK+b) \mod p) \mod M.$$

37. Tétel. *Legyen* $h_{a,b}(K) := ((aK+b) \mod p) \mod M$ *minden* $1 \leq a < p$*-re és* $0 \leq b < p$*-re. A* $\mathcal{H}_{p,M} = \{h_{a,b}(K) \mid 1 \leq a \leq p-1,\ 0 \leq b \leq p-1\}$ *család univerzális.*

8.4. Kakukk hashelés

Ezt a praktikus módszert Pagh és Rodler fejlesztette ki 2001-ben. Mi csak vázlatosan tárgyaljuk.

Választunk egy univerzális hash függvénycsaládot, és ebből függetlenül két függvényt, h_1-et és h_2-őt. Azt kívánjuk fenntartani, hogy ha $K \in S$, akkor a táblában vagy a $h_1(K)$, vagy a $h_2(K)$ indexű helyen legyen, így a Keresés $O(1)$, mivel csak két helyet kell ellenőriznünk.

A Beszúrás (K) lényegében így történik: ha $K \notin S$, akkor berakjuk a $h_1(K)$ helyre, és ha az nem volt üres, akkor az onnan kitúrt elemet tekintjük ezután K-nak. Ennek van pontosan egy másik lehetséges helye, ezért odarakjuk, és most már az onnan kitúrt elemet tekintjük K-nak, és folytatjuk. Ha egyszer is üres helyre rakjuk be, akkor sikeresek voltunk és megállunk.

Ez persze nem csak hogy sokáig tarthat, de végtelen ciklusba is kerülhet. Ezért jól be kell állítanunk egy paramétert (legyen pl. MAXLOOP a neve), és ennyi kitúrás után mást kell csinálnunk. Ez pedig új tábla felvétele, új h_1 és h_2 sorsolása, és minden szótárelem áthashelése a fenti módszerrel az új táblába. Ha táblánk elég üres, akkor az új tábla ugyanakkora, különben pl. kétszer akkora.

Talán érezhető, hogy a módszer egyszerűsége ellenére az elemzés nem lehet triviális.

Az alapelvet azonban szeretnénk bemutatni. Egy Beszúrás (S, K) műveletnél definiáljuk a kakukk-gráfot, ennek csúcshalmaza $\{0, \ldots, M{-}1\}$, és $S \cup \{K\}$ minden eleméhez tartozik egy él, ami a két lehetséges helyét köti össze. Azt szeretnénk, hogy két tetszőleges fix index csak kis (azaz $O(1/N)$) valószínűséggel legyen egy komponensben.

1. Lemma. *Minden $i \neq j$ indexre, $\ell \geq 1$ egészre, és $c > 1$ valósra, ha $M \geq 2cN$, akkor annak valószínűsége, hogy a kakukk-gráfban i és j között vezet pontosan ℓ hosszú út, legfeljebb $1/(c^\ell M)$.*

Legyen most C a kakukk-gráfban a $h_1(K)$-t tartalmazó komponens, és $K' \in S$. Ekkor $\Pr(h_1(K') \in C) \leq \sum_\ell \frac{1}{c^\ell M} = \frac{1}{(c-1)M}$. Így C méretének várható értéke legfeljebb $2 + 2\frac{N}{(c-1)M} \leq 2 + \frac{1}{c(c-1)} \leq 11{,}1$ (ha pl. $c \geq 1{,}1$).

Megjegyzés: A fenti lemma körökre is működik, annak valószínűsége, hogy egy adott i indexű csúcsa a kakukk-gráfnak benne van egy körben, legfeljebb $\sum_\ell \frac{1}{c^\ell M} = \frac{1}{(c-1)M}$. Innen az analízis úgy folytatódik, hogy az erősebb $M \geq 2(1+\varepsilon)cN$ feltétel mellett εn egymás utáni beszúrásnál az újrahashelések várható darabszáma konstans, így várható ideje $O(N)$, azaz elemenként amortizációs időben $O(1)$.

9. fejezet

Párosítások

9.1. Stabil párosítás páros gráfban

25. Definíció. *Egy gráf élhalmazának egy M részhalmazát párosításnak hívunk, ha semelyik csúcsra nem illeszkedik 1-nél több M-beli él. (Azaz $\forall v \in V$-re $d_M(v) \leq 1$).*

Az M párosítás teljes, ha $\forall v \in V$-re $d_M(v) = 1$.

Feladat: Stabil párosítás megtalálása adott gráfban.
Input: egy páros gráf, ahol az egyik osztályt fiúknak, a másikat lányoknak nevezzük; valamint minden csúcsnál a kimenő éleken egy (szigorú) preferencia-sorrend.

26. Definíció. *Egy $fl \in E$ blokkoló él (instabilitás) M-re nézve, ha*

a) $fl \notin M$, és

b) f és l kölcsönösen jobban szeretik egymást, mint az M-beli párjukat (vagy nincs M-beli párjuk).

Egy M párosítás stabil, ha nem létezik M-re nézve blokkoló él.

46. Algoritmus (Gale–Shapley).

I. Fiúk algoritmusa: minden fiú először megkéri az általa legkedveltebb lányt, ha az kikosarazza, akkor megkéri a következőt, és így tovább.

II. Lányok algoritmusa: egy lány az első kérőt elfogadja ideiglenes partnernek, a további kérőknél eldönti melyik a jobb: a mostani partner, vagy az új kérő; a rosszabbikat kikosarazza, és így tovább. Mindig az eddigi legjobbat tartja meg (ideiglenes) partnernek, és az összes többit kikosarazza.

Miután már nem változik semmi, minden lány feleségül megy a partneréhez (ha van neki; csak akkor nincs, ha soha senki nem kérte meg).

38. Tétel (Gale–Shapley, 1962). *A fenti algoritmus minden input esetén stabil párosítást talál, lépésszáma $O(m)$.*

Főleg ezért az algoritmusért kapott közgazdasági Nobel-díjat Lloyd Shapley 2012-ben (David Gale már 2008-ban elhunyt).

9.2. Maximális párosítás páros gráfban

27. Definíció. *Legyen M egy párosítás.*

a) A v csúcs szabad, ha $d_M(v) = 0$; különben fedett.

b) Egy út alternáló, ha élei felváltva $\in M$, ill. $\notin M$.

c) Egy alternáló út javító, ha mindkét vége szabad csúcs.

A célunk az, hogy sorban újabb javító utat keressünk, és ennek mentén javítsunk, azaz az eddiginél eggyel nagyobb párosítást kapjunk. Javító utat pedig a szélességi keresés egy kicsit módosított változatával keresünk.

Legyen $G = (U \cup V, E)$, azaz az alsó csúcshalmaz: $U = \{u_1, u_2, \ldots, u_{n_1}\}$ és a felső csúcshalmaz: $V = \{v_1, v_2, \ldots, v_{n_2}\}$. A gráf az $e(j)$ éllistákkal van adva: $e(j) = \{i \mid u_j v_i \in E\}$. Segédváltozók: L sor, M, m és p tömbök:

$$M(i) = \begin{cases} 0, & \text{ha } v_i \text{ szabad,} \\ j, & \text{ha } v_i u_j \text{ párosítás-él.} \end{cases}$$

$$m(j) = \begin{cases} 0, & \text{ha } u_j \text{ szabad,} \\ i, & \text{ha } u_j v_i \text{ párosítás-él.} \end{cases}$$

$$p(j) = \begin{cases} 0, & \text{ha } u_j\text{-t még nem láttuk,} \\ j, & \text{ha } u_j \text{ szabad,} \\ j', & \text{ha } u_{j'} \text{ az } u_j \text{ „szülője” (a szélességi keresésnél).} \end{cases}$$

47. Algoritmus („Magyar módszer”).

INIT:
for $i = 1..n_2$ $\quad M(i) := 0$
for $j = 1..n_1$ $\quad m(j) := 0$

ELEJE:

ÚJSOR(L, n_1)
for $j = 1..n_1$
  $p(j) := 0$
  if $m(j) = 0$ **then**
    $L \leftarrow j$
    $p(j) := j$
while $L \neq \emptyset$
  $j \leftarrow L$
  for $i \in e(j)$
    if $M(i) = 0$ **then** $\longrightarrow$ JAVÍTÁS(i, j)
    if $p(M(i)) = 0$ **then**
      $p(M(i)) := j$
      $L \leftarrow M(i)$
print „Párosítás"
for $i = 1..n_2$ **if** $M(i) > 0$ **then** **print** $(v_i, u_{M(i)})$
print „Ore"
for $j = 1..n_1$ **if** $p(j) > 0$ **then** **print** u_j
print „Kőnig"
for $j = 1..n_1$ **if** $p(j) = 0$ **then** **print** u_j
for $i = 1..n_2$ **if** $M(i) > 0$ **&&** $p(M(i)) > 0$ **then** **print** v_i

JAVÍTÁS(i, j):

$M(i) := j$
while $m(j) > 0$
  CSERE$(i, m(j))$
  $j := p(j)$
  $M(i) := j$
$m(j) := i$
$\longrightarrow$ ELEJE

Lépésszám: $O(n \cdot m)$, ahol $n = \min(n_1, n_2)$

Megjegyzés: Ezt a Kőnig Dénestől származó algoritmust az irodalomban sokszor nevezik „magyar módszernek", de ez nem helyes, mert az igazi magyar módszer a 17.2. fejezetben leírt 57. algoritmus, ami a nehezebb, súlyozott esetet oldja meg.

39. Tétel (Frobenius). *Egy $G = (V \cup U, E)$ páros gráfban akkor és csak akkor létezik teljes párosítás, ha $|U| = |V|$ és $\forall X \subseteq V$-re $|\Gamma(X)| \geq |X|$. (Ahol $\Gamma(X)$ az X halmaz szomszédsága: $\Gamma(X) = \{u \in U \mid \exists x \in X,\ xu \in E\}$).*

40. Tétel (Hall). *Egy $G = (V \cup U, E)$ páros gráfban létezik V-t fedő párosítás akkor és csak akkor, ha $\forall X \subseteq V$-re $|\Gamma(X)| \geq |X|$.*

41. Tétel (Ore). *Egy $G = (V \cup U, E)$ páros gráfban $\nu(G)$-vel jelöljük a legnagyobb párosítás méretét. Erre igaz, hogy $\nu(G) = |V| - \max_{X \subseteq V}(|X| - |\Gamma(X)|)$.*

42. Tétel (Kőnig). *Ha $G = (V \cup U, E)$ páros gráf, akkor $\nu(G) = \tau(G)$, ahol $\tau(G) := = \min\{|T| \mid T \subseteq V \cup U,\ \forall uv \in E : T \cap \{u, v\} \neq \emptyset\}$.*

9.3. Párosítás nem páros gráfban

2. Lemma (Berge). *Egy G (nem feltétlenül páros) gráfban $|M| = \nu(G)$ akkor és csak akkor, ha nem létezik javító út.*

43. Tétel (Tutte). *Egy G gráfban akkor és csak akkor létezik teljes párosítás, ha*

$$\forall X \subseteq V\text{-re} \quad q(G-X) \leq |X|,$$

ahol $q(G-X)$ a páratlan csúcsszámú komponensek száma a $G-X$ gráfban.

Erre a feladatra is létezik hatékony algoritmus (Edmonds), melynek eredetileg a lépésszáma $O(n^2m)$, azóta javult.

10. fejezet

Hálózati folyamok

28. Definíció. Hálózat: (G, c, s, t), *ahol* $G=(V, E)$ *irányított gráf;* $c: E \to \mathbb{R}$ *és* $c(e) \geq 0$ *minden* e *élre (az* e *él kapacitása); valamint* $s \neq t \in V$ *csúcsok:* s *a forrás,* t *a nyelő.*

29. Definíció. Folyam: *egy* $f : E \to \mathbb{R}$ *függvény, amelyre az alábbiak teljesülnek:*

a) $\forall e \in E$*-re* $0 \leq f(e) \leq c(e)$

b) $\forall v \in V \setminus \{s, t\}$*-re* $f_{\text{be}}(v) = f_{\text{ki}}(v)$, *ahol* $f_{\text{be}}(v) = \sum_{u:\, uv \in E} f(uv)$ *és* $f_{\text{ki}}(v) = \sum_{w:\, vw \in E} f(vw)$.

c) *A folyam* értéke*:* $|f| := f_{\text{ki}}(s) - f_{\text{be}}(s)$.

Feladat: Maximális (értékű) folyam keresése egy hálózatban.

30. Definíció. (S, T) Vágás: $V = S \cup T$ *partíció, ahol* $s \in S$ *és* $t \in T$.
Vágás kapacitása: $c(S, T) := \sum_{u \in S, v \in T, uv \in E} c(uv)$
Vágás folyamértéke: $f(S, T) := \sum_{u \in S, v \in T, uv \in E} f(uv) - \sum_{u \in S, v \in T, vu \in E} f(vu)$.

44. Tétel. a) *Minden* (S, T) *vágásra* $f(S, T) \leq c(S, T)$

b) *Minden* (S, T) *vágásra* $f(S, T) = |f|$

c) *Következésképpen, ha* f *egy folyam és* (S, T) *egy vágás, amelyekre teljesül, hogy* $f(S, T) = c(S, T)$, *akkor szükségképpen* f *egy maximális értékű folyam és* (S, T) *egy minimális kapacitású vágás.*

31. Definíció. *Maradékhálózat: Adott egy* $(G = (V, E), c, s, t)$ *hálózat, és rajta egy* f *folyam. Elkészítjük a* $(G'=(V, E'), r, s, t)$ *maradékhálózatot: (ebben* $r(e)$ *az* e *él maradék kapacitását fogja tartalmazni):*

- *Minden* $uv \in E$ *élre, ha* telítetlen *(azaz* $f(uv) < c(uv)$*), betesszük* uv*-t az* E'*-be,* $r(uv) := c(uv) - f(uv)$*, és az* uv *élet* előre él*nek jelöljük.*
- *Minden* $uv \in E$ *élre, ha* pozitív *(azaz* $f(uv) > 0$*), betesszük a* vu fordított *élet az* E'*-be,* $r(vu) := f(uv)$*, és a* vu *élet* vissza él*nek jelöljük.*

A definíció alapján könnyen kaphatunk egy algoritmust a maradékhálózat elkészítésére. A G' éllistáját konstruáljuk meg: mindig, ha be kell venni egy uv élet, azt az u csúcs listájának elejére illesztjük be, és ráírjuk az $r(uv)$ maradék kapacitást is, és egy bitet, ami 0, ha előre él, illetve 1, ha vissza él. Egy él beszúrása így $O(1)$ időben megy, és mivel a G' gráfnak maximum $2m$ éle van, a konstruálás összes lépésszáma $O(m)$. A definíció miatt G' minden e élére $r(e) > 0$.

10.1. Folyam-algoritmusok

48. Algoritmus (Ford–Fulkerson).

FF(G, c, s, t) :
  for $e \in E$ $f(e) := 0$
  repeat
    A meglévő f folyamunkhoz elkészítjük a G' maradékhálózatot.
    G'-ben keresünk P : $s \rightsquigarrow t$ utat, legyen S az s-ből elérhető csúcsok halmaza.
    if $t \notin S$ **then return**$(f, S, V-S)$
    else JAVÍT(P, f)

JAVÍT(P, f): /* *P egy $s \rightsquigarrow t$ út G'-ben.*
  $\Delta := \min_{e \in P} r(e)$ /* $\Delta > 0$
  for $uv \in P$
    if uv előre él G'-ben **then** $f(uv) := f(uv) + \Delta$
    else $f(vu) := f(vu) - \Delta$ /* *Ha pedig vissza él*
  return(f)

Példa: 15. példa.

45. Állítás. *A javító utas algoritmus által visszaadott új f szintén egy folyam, és $|f| = |f_{\text{regi}}| + \Delta$, ahol $\Delta = \min_{e \in P} r(e) > 0$.*

46. Tétel (Ford–Fulkerson).
i) *Egy f folyamra $|f|$ maximális akkor és csak akkor, ha a G' maradék hálózatban nem létezik $s \rightsquigarrow t$ út.*
ii) *A maximális folyamérték egyenlő a minimális vágáskapacitással.*
iii) *Ha $c(e)$ egész minden e élre, akkor van olyan f maximális folyam, hogy $f(e)$ szintén egész minden e élre.*

49. Algoritmus (Edmonds–Karp).

Ugyanez, azzal a különbséggel, hogy a G'-ben egy legrövidebb P utat keresünk (pl. a szélességi keresés automatikusan ilyet talál).

47. Tétel (Edmonds–Karp). *Ha G'-ben mindig legrövidebb (azaz a legkevesebb élből álló) $s \rightsquigarrow t$ utat keresünk, akkor legfeljebb $n \cdot m$ iteráció elég. Ezért ekkor az összes lépésszám $O(n \cdot m^2)$.*

10.2. Redukált folyam, folyam felbontása

32. Definíció. *Egy f folyamhoz definiáljuk a $G_f := \{e \in E \mid f(e) > 0\}$ gráfot. Az f folyamot* redukált *folyamnak hívjuk, ha G_f aciklikus.*

48. Tétel. *Ha f egy folyam, akkor $\exists f'$ redukált folyam, hogy $|f'| = |f|$.*
Ha f redukált folyam, akkor $f = \sum \lambda_i P_i$, ahol P_i-k bizonyos $s \rightsquigarrow t$ utak az 1 folyamértékkel véve. Ha f minden élen egész, akkor a λ_i számok is választhatók egésznek.

Ha a csúcsokon is vannak kapacitások, akkor a csúcsokat széthúzzuk a házépítésnél látott módon, és a $v^- v^+$ élekre írjuk a v csúcs kapacitását.

Irányítatlan gráf esetén az éleket oda-vissza irányított élpárokkal helyettesítjük, és redukált folyamot keresünk.

Ha egy termék van, de több forrás és több nyelő, amiknek szintén van kapacitásuk, akkor felveszünk egy új szuperforrást, amiből az adott kapacitásokkal éleket vezetünk az eredeti forrásokba, és egy új szupernyelőt, amibe az adott kapacitásokkal éleket vezetünk az eredeti nyelőkből. Mindhárom esetben az eddigi algoritmusokkal meg tudjuk oldani a feladatot.

10.3. Menger tételei, többszörös összefüggőség

49. Tétel (Él-Menger). *G irányított/irányítatlan gráfban az $s \rightsquigarrow t$-be vezető páronként éldiszjunkt utak maximális száma = a t csúcsot az s-től elszeparáló élek minimális számával. (Ahol egy élhalmaz elszeparáló, ha törlése után már nincs irányított/irányítatlan $s \rightsquigarrow t$ út).*

50. Tétel (Csúcs-Menger). *Tegyük fel, hogy $st \notin E$ (nem él). Az s-ből t-be vezető páronként belül csúcsdiszjunkt utak maximális száma = a t csúcsot az s-től elszeparáló $X \subseteq V \setminus \{s, t\}$ csúcshalmaz minimális méretével.*

33. Definíció. *A G irányítatlan gráf k-szorosan él-összefüggő, ha $G - e_1 - e_2 \ldots - e_{k-1}$ összefüggő marad minden $e_1, e_2, \ldots, e_{k-1} \in E$ esetén.*

34. Definíció. *A G irányítatlan gráf k-szorosan összefüggő, ha $|V| \geq k+1$ és $G - v_1 - - v_2 \ldots - v_{k-1}$ összefüggő marad minden $v_1, v_2, \ldots, v_{k-1} \in V$ esetén.*

51. Tétel (Menger). *G gráf k-szorosan él-összefüggő akkor és csak akkor, ha $\forall x \neq y \in V$-re van k db páronként él-diszjunkt $x \rightsquigarrow y$ út.*

52. Tétel (Menger). *G gráf k-szorosan összefüggő akkor és csak akkor, ha $|V| \geq k+1$ és $\forall x \neq y \in V$-re van k db páronként belül csúcs-diszjunkt $x \rightsquigarrow y$ út.*

11. fejezet

Adattömörítés

35. Definíció. *Ábécének nevezünk egy Σ-val jelölt véges halmazt.*
Szó: $x = a_1 a_2 \dots a_n$, ahol $a_i \in \Sigma$ betűk, a szó hossza $|x| = n$.
A k hosszú szavak halmaza: Σ^k.
Összes szó halmaza: $\Sigma^* = \bigcup_{k=0}^{\infty} \Sigma^k$.
Üres szó: ε, hossza 0.
Konkatenáció: $x, y \in \Sigma^*$ esetén xy (egymás után írás).
Nyelv: $\mathcal{L} \subseteq \Sigma^*$ (szavak halmaza).
Eldöntési problémákat azonosítani lehet a nyelvekkel:
x inputon IGEN a helyes válasz $\Leftrightarrow x \in \mathcal{L}$.

36. Definíció. *Egy $c : \Sigma \to \{0,1\}^*$ injektív függvényt (betű-) kódnak hívunk.*
Egy c kód prefix-mentes kód, *ha $a \neq b \in \Sigma$ esetén $c(a)$ és $c(b)$ egyike sem kezdőszelete a másiknak. Az ilyen kódok bijektíven megfeleltethetők az olyan bináris fáknak, melyek éleire 0 és 1, leveleire pedig a betűk vannak írva.*
(x kezdőszelete y-nak, ha $\exists\ z \in \{0,1\}^*$, hogy $xz = y$.)

37. Definíció. *Huffman-kód. Adott Σ, egy kódolandó $x \in \Sigma^*$ szó, és az összes a_i betű r_i előfordulási száma x-ben. A Huffman-kód a legrövidebb prefix-mentes kód.*
Kódhossz$= \sum d(i) \cdot r_i$, ahol $d(i)$ az a_i betűt tartalmazó levél mélysége.

50. Algoritmus (Huffman).

Felveszünk $|\Sigma|$ db egycsúcsú fát, az a_i csúcsba beleírjuk az r_i számot. Ezután a következő lépést ismételjük, amíg egynél több fánk van:

Vesszük azt a két gyökeret, melyekbe a két legkisebb szám van írva, és ezeket egy újonnan létrehozott gyökér gyerekeivé tesszük. Az új gyökérbe a két gyerekébe írt szám összegét írjuk.

53. Tétel. *A Huffman eljárásával kapott kód optimális.*

12. fejezet

A bonyolultságelmélet alapjai

Az ábécé, szó és nyelv definícióját lásd a 11. fejezet elején.

A Turing-gép informális leírása a következő: van egy központi egysége, melynek véges sok *állapota* van, köztük a kitüntetett START és két megállást jelző ELF és ELUT állapot. Van ezenkívül egy (vagy akár $k > 1$) *szalag*ja, melynek minden *mező*jében az ábécé egy-egy eleme áll, kezdetben az input és rajta kívül az úgynevezett üresjelek. Továbbá van egy (vagy k) író-olvasó *fej*e, mely a szalag egy mezőjét tudja leolvasni, majd felülírni, és utána esetleg egyet jobbra vagy balra lépni.

A működését egy táblázatban lehet leírni, aminek elemei úgy néznek ki, hogy pl. „ha A állapotban vagyunk és a b betűt olvassuk, akkor váltsunk át az A' állapotba, a fej írja ki a c betűt és lépjen egyet jobbra". A gép mindig a START állapotban indul, és a feje az input első mezőjén van; és akkor áll le, ha a központi egység ELF vagy ELUT végállapotba kerül.

Ebből a definícióból úgy érezhető, hogy minden egyes problémához egy újabb Turing-gépet kell konstruálni. Szerencsére ez nincsen így, Turing csinált egy univerzális gépet is, és bebizonyította a következő tételt:

54. Tétel. *Minden k-hoz és Σ-hoz létezik $k+1$-szalagos univerzális T^* Turing-gép a Σ ábécé felett, amely a következőket tudja. Bármely T k-szalagos Σ feletti Turing-géphez van olyan $p \in \Sigma^*$ program, hogy ezt a T^* gép utolsó szalagjára írva szimulálni tudja minden x inputon a T működését, ráadásul $O(time_T(n))$ lépés alatt.*

38. Definíció. *Egy T Turing-gép eldönti az $\mathcal{L}$ nyelvet,*
ha minden inputra leáll véges sok lépésben,
és egy x inputra pontosan akkor áll meg ELF állapotban, ha $x \in \mathcal{L}$.

Egy nyelv algoritmikusan eldönthető, ha van olyan Turing-gép, ami eldönti.

Megjegyzések: Mivel csak megszámlálhatóan végtelen sok Turing-gép van, ellenben kontinuum sok nyelv, nyilván létezik eldönthetetlen nyelv. Az érdekes az, hogy konkrét és érdekes nyelvekre tudjuk ezt be is bizonyítani.

1. Példa. *Algoritmikusan eldönthetetlen például, hogy egy Diofantoszi egyenletnek (pl. $x^3 + 3yz^2 - 2xy^4 = 0$) van-e csupa egész számokból álló megoldása. Tehát* bizonyított, *hogy nem lehet olyan programot írni, amely minden input egyenletre leáll véges sok lépés után, és mindig helyesen dönti el, hogy az egyenletnek van-e egész megoldása.*

39. Definíció. *Nyelvosztály:* $DTIME(t(n)) = \{\mathcal{L}$ *nyelv* $| \exists\ \mathcal{L}$*-et eldöntő* T *Turing-gép, amelyre* $time_T(n) \leq t(n)\ \forall n\}$. *(Azaz minden* x *inputra legfeljebb* $t(|x|)$ *lépést tesz.)*

$\mathrm{P} = \bigcup_{c=1}^{\infty} DTIME(n^c)$.

40. Definíció. NP *osztály: egy* $\mathcal{L}$ *nyelv az* NP *osztályban van, ha minden* $x \in \mathcal{L}$*-re létezik egy* y *polinom hosszú és polinomiális időben ellenőrizhető bizonyíték. Azaz pontosabban, ha* $\mathcal{L}$*-hez létezik polinomiális lépésszámú* E *ellenőrző Turing-gép és* c *konstans, hogy:*

Ha $x \in \mathcal{L}$*, akkor* $\exists y$*, hogy* $|y| \leq |x|^c$ *és* E *az* (x, y) *párt ELFogadja.*

Ha $x \notin \mathcal{L}$*, akkor* $\forall y$*-ra* E *az* (x, y) *párt ELUTasítja.*

2. Példa. a) *Hamilton-kör létezése* $\in$NP*, bizonyíték a Hamilton-kör.*

b) *Egy gráf* 3 *színnel színezhetősége* $\in$NP*, bizonyíték a* 3 *színnel való kiszínezés.*

41. Definíció. *Az* $\mathcal{L}$ *nyelvet* NP*-teljesnek hívjuk, ha* $\mathcal{L} \in$NP*, és ha* $\mathcal{L} \in$P *igaz lenne, akkor* minden $\mathcal{L}' \in$NP *nyelvre* $\mathcal{L}' \in$P *következne.*

Egy (nem feltétlenül eldöntési) problémát NP*-nehéznek hívunk, ha abból, hogy ki lehet minden inputra számolni a megoldást polinom időben, következne, hogy minden* NP*-beli eldöntési probléma* P*-ben van, azaz hogy* NP=P.

55. Tétel (Cook). *Létezik* NP*-teljes nyelv. (A SAT nyelv* NP*-teljes.)*

56. Tétel. a) *Hamilton-kör létezése* NP*-teljes.*

b) *Egy gráfban* k*-nál hosszabb út létezése* NP*-teljes.*

c) *Egy gráf* 3 *színnel színezhetősége* NP*-teljes.*

d) *Egy gráfban* k *méretű teljes részgráf létezése* NP*-teljes.*

e) *A hátizsák-feladat eldöntési verziója* NP*-teljes.*

f) *Adott* n *egész szám:* $a_1, \ldots, a_n$. NP*-teljes annak eldöntése, hogy létezik-e olyan* $I \subseteq \{1, \ldots, n\}$*, melyre*

$$\sum_{i \in I} a_i = \sum_{j \in \{1,\ldots,n\} \setminus I} a_j.$$

Megjegyzés: Garey és Johnson 1979-ben írt „Computers and intractability" című könyve az első, amely számtalan probléma NP-teljességét bizonyítja. Erre a könyvre több mint 80000 hivatkozás van.

Az egymillió dolláros kérdés az, hogy P=?NP, ez egyike a 7 Millenniumi problémának. Ez ekvivalens azzal, hogy vajon valamelyik NP-teljes probléma a P osztályban van-e (azaz van-e rá hatékony algoritmus)?

42. Definíció. *A* co-NP *osztály azokat a nyelveket tartalmazza, melyek komplementere* NP*-ben van, tehát a* NEM *válaszra van polinomiális tanú.*

Fontos osztály az NP$\cap$co-NP*, ide azok az eldöntési problémák tartoznak, amelyeknél mind az* IGEN*, mind a* NEM *válaszra van polinomiális tanú.*

Nagyon sok fontos problémáról, amelyekről kiderült, hogy az NP∩co-NP osztályban vannak, később sikerült azt is bizonyítani, hogy P-ben vannak. Egy fontos kivétel a prímfaktorizáció probléma eldöntési változata: adottak n és k természetes számok, létezik-e n-nek k-nál nem nagyobb valódi osztója?

Ha ez P-ben lenne, akkor tudnánk polinom időben faktorizálni. A matematikusok többsége azt hiszi, hogy nem ez a helyzet, bár nem valószínű, hogy a fenti probléma NP-teljes lenne, hiszen ebből a meglepő NP=NP∩co-NP következne. (Márpedig nem hisszük, hogy van arra polinomiális tanú, hogy egy gráfban NINCS Hamilton-kör).

Második rész

Következő lépés

13. fejezet

Aritmetika: Számolás nagy számokkal

13.1. Nagy számok szorzása Karacuba [1960] módszerével

Tegyük fel, hogy a két szorzandó bináris alakja: $u = u_0 + 2u_1 + 2^2u_2 + \ldots, +2^{n-1}u_{n-1}$ és $v = v_0 + 2v_1 + 2^2v_2 + \ldots, +2^{n-1}v_{n-1}$. Az eredményt is ilyen alakban várjuk: $w = uv = = w_0 + 2w_1 + 2^2w_2 + \cdots + 2^{2n-1}w_{2n-1}$.

A hagyományos módszer ekkor körülbelül n^2 bit műveletet igényel. Egy egyszerű ötlettel kezdjük. Tegyük fel, hogy $n{=}2m$ páros, ekkor a számokat írhatjuk $u{=}2^mU_1{+}U_0$ és $v{=}2^mV_1{+}V_0$ alakban, ahol U_i és V_i m-bites számok. Ekkor $uv{=}2^{2m}U_1V_1{+}2^m(U_0V_1{+} {+}U_1V_0){+}U_0V_0$.

A lényeges megfigyelés az, hogy négyről háromra is levihetjük a rekurzívan hívandó szorzások számát: $U_0V_1 + U_1V_0 = (U_1 - U_0)(V_0 - V_1) + U_0V_0 + U_1V_1$, tehát így $w = uv = = (2^{2m} + 2^m)U_1V_1 + 2^m(U_1 - U_0)(V_0 - V_1) + (2^m + 1)U_0V_0$.

Ezzel a módszerrel két $2m$-bites egész szám szorzását három, m-bites számok közötti szorzásra valamint néhány összeadásra és 2-hatvánnyal való szorzásra redukáltuk. Könnyű utánaszámolni, hogy ezek a további műveletek összesen legfeljebb $22m$ bitműveletet igényelnek. Ha $T(m)$-mel jelöljük két m-bites szám szorzásához szükséges bitműveletek számát, akkor tehát $T(2m) \leq 3T(m) + 22m$.

Ebből az egyenlőtlenségből könnyen lehet felső korlátot adni a bitműveletek számára, legyen $k = \lceil \log n \rceil$, ekkor (indukcióval könnyen bizonyíthatóan):

$T(2^k) \leq 3T(2^{k-1}) + 22 \cdot 2^{k-1} \leq 3^k + 22(3^k - 2^k)$, és innen

$T(n) \leq T(2^k) < 23 \cdot 3^k < 23 \cdot 3^{1+\log n} = 69 \cdot n^{\log 3} = O(n^{1,59})$.

Ehhez hasonló módon Strassen megmutatta, hogy két darab $(2n) \times (2n)$-es mátrix szorzásához elég 7 darab $n \times n$-es mátrix-szorzás és néhány mátrix-összeadás, ami végül ahhoz vezet, hogy két $n \times n$-es mátrix szorzása elvégezhető $O(n^{2,81})$ darab művelettel a triviális $O(n^3)$ helyett (ahol egy művelet két szám szorzása ill. összeadása).

Ezen azóta sokat javítottak. Általában ω jelöli a legkisebb exponenst, amire ismert, hogy két $n \times n$-es mátrix szorzása elvégezhető $O(n^\omega)$ művelettel. Jelenleg $\omega = 2.25$ az OpenAI 2026. szeptemberi (nem ellenőrzött) eredménye szerint.

13.2. A diszkrét Fourier-transzformált

Tegyük fel, hogy az alábbi két $(n-1)$-ed fokú egyváltozós polinomot szeretnénk összeszorozni:

$$P(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}, \quad \text{és} \quad Q(x) = b_0 + b_1 x + \cdots + b_{n-1}x^{n-1}.$$

Ekkor a következő szorzatot kell kiszámítanunk:

$$R(x) = P(x)Q(x) = c_0 + c_1 x + \cdots + c_{2n-2}x^{2n-2},$$

ahol az együtthatók:

$$c_i = a_0 b_i + a_1 b_{i-1} + \cdots + a_i b_0. \tag{13.1}$$

Ezt a $(c_0, \ldots, c_{2n-2})$ sorozatot gyakran az $(a_0, \ldots, a_{n-1})$ és $(b_0, \ldots, b_{n-1})$ sorozatok *konvolúció*jának hívják. Ezzel a képlettel az együtthatók meghatározásához n^2 szorzásra van szükség.

Ennél ügyesebb módszer, ha felhasználjuk, hogy be is helyettesíthetünk a polinomokba. Helyettesítsük be pl. a $0,1,\ldots,2n-2$ értékeket. Más szóval számítsuk ki a $P(0), P(1), \ldots, P(2n-2)$ és $Q(0), Q(1), \ldots, Q(2n-2)$ értékeket, majd ezek $R(j) = P(j)Q(j)$ szorzatait. Ebből R együtthatóit az alábbi egyenletrendszer megoldásai adják:

$$\begin{aligned}
c_0 &= R(0) \\
c_0 + c_1 + c_2 + \cdots + c_{2n-2} &= R(1) \\
c_0 + 2c_1 + 2^2 c_2 + \cdots + 2^{2n-2} c_{2n-2} &= R(2) \\
&\vdots \\
c_0 + (2n-2)c_1 + (2n-2)^2 c_2 + \cdots + (2n-2)^{2n-2} c_{2n-2} &= R(2n-2)
\end{aligned} \tag{13.2}$$

Ez első ránézésre nem tűnik túl jó ötletnek, mert a $P(0), P(1), \ldots, P(2n-2)$, illetve a $Q(0), Q(1), \ldots, Q(2n-2)$ értékek kiszámításához is körülbelül $2 \cdot n^2$ szorzás (és hasonló számú összeadás) kellett; az $R(0), R(1), \ldots R(2n-2)$ értékek meghatározásához kell még $2n-1$ szorzás, ráadásul ezek után n^3 nagyságrendű szorzás, osztás és összeadás kell (13.2) megoldásához, amennyiben Gauss-eliminációt használunk. Viszont mégis látható némi nyereség, ha megkülönböztetjük a konstanssal való szorzást két változó összeszorzásától (ahol most a változóink a P és Q polinomok együtthatói). A konstanssal való szorzás az összeadáshoz hasonlóan lineáris művelet, így már a $P(0), P(1), \ldots, P(2n-2)$ és a $Q(0), Q(1), \ldots, Q(2n-2)$ értékek kiszámítása, valamint a (13.2) egyenletrendszer megoldása csak lineáris műveleteket használ. Így a nemlineáris műveletek száma összesen $2n-1$.

Ennél is jobbat tudunk, ha észrevesszük, hogy igazából a $0,1,\ldots,2n-2$ értékek helyett tetszőleges $2n-1$ valós, vagy akár komplex számot is behelyettesíthetnénk. Komplex egységgyökök helyettesítésével sokkal hatékonyabb módszer kapható polinomok szorzására.

Legyen $P(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$ egy polinom, feltehetjük, hogy $n = 2^k$ egy kettő-hatvány. Vegyük a következő helyettesítési értékeit P-nek:

$$\hat{a}_j = P(\varepsilon^j) = a_0 + a_1\varepsilon^j + a_2\varepsilon^{2j} + \cdots + a_{n-1}\varepsilon^{(n-1)j} \qquad (j = 0, \ldots, n-1), \tag{13.3}$$

ahol ε egy primitív n. egységgyök. Az $(\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_{n-1})$ komplex számsorozatot az $(a_0, a_1, \ldots, a_{n-1})$ sorozat *n-ed rendű diszkrét Fourier-transzformáltj*ának nevezzük.

A diszkrét Fourier-transzformáltaknak számos érdekes tulajdonsága és fontos alkalmazása van, melyek közül csak azt a kettőt tárgyaljuk, ami polinomok szorzásához kapcsolódik.

A következő fontos alaptulajdonsággal kezdjük: a visszatranszformálás is hasonló képlettel történik:

$$a_i = \frac{1}{n}\big(\hat{a}_0 + \hat{a}_1\varepsilon^{-i} + \hat{a}_2\varepsilon^{-2i} + \cdots + \hat{a}_{n-1}\varepsilon^{-(n-1)i}\big) \qquad (i = 0, \ldots, n-1). \tag{13.4}$$

Mostantól feltesszük, hogy a szorzandó P és Q polinomjaink **legfeljebb** $(n-1)/2$ fokúak, azaz a magasabb indexű együtthatóik 0-k, valamint hogy $n = 2^k$. Legyenek $(b_0, \ldots, b_{n-1})$, illetve $(c_0, \ldots, c_{n-1})$ a $Q(x)$ és $R(x) = P(x)Q(x)$ polinomok együtthatói, valamint legyenek $(\hat{b}_0, \ldots, \hat{b}_{n-1})$, illetve $(\hat{c}_0, \ldots, \hat{c}_{n-1})$ ezek n-ed rendű diszkrét Fourier-transzformáltjai. Mivel $\hat{a}_j$ egy P-be való behelyettesítés, azt kapjuk, hogy

$$\hat{c}_j = \hat{a}_j\hat{b}_j. \tag{13.5}$$

A diszkrét Fourier-transzformált legfontosabb tulajdonsága, hogy gyorsan kiszámolható; így ez a módszer (amit FFT-nek, azaz gyors Fourier-transzformációnak hívnak) az egyik legfontosabb algoritmikus eszköz algebrai számításoknál. Az alapötlet a következő azonosság:

$$\begin{aligned}\hat{a}_j &= \sum a_i\varepsilon^{ij} = \big[a_0(\varepsilon^2)^{0\cdot j} + a_2(\varepsilon^2)^{1\cdot j} + \cdots + a_{2i}(\varepsilon^2)^{i\cdot j} + \ldots\big] + \\ &+\varepsilon^j\big[a_1(\varepsilon^2)^{0\cdot j} + a_3(\varepsilon^2)^{1\cdot j} + \cdots + a_{2i+1}(\varepsilon^2)^{i\cdot j} + \ldots\big] = \\ &= \sum a_{2i}\vartheta^{ij} + \varepsilon^j\sum a_{2i+1}\vartheta^{ij},\end{aligned}$$

ahol $\vartheta = \varepsilon^2$ primitív $(n/2)$. egységgyök.

51. Algoritmus (FFT).

FFT$(k, A[0 : 2^k - 1])$**:**
  if $k = 0$ **then return**$(A(0))$
  eps:= $e^{\frac{2\pi i}{2^k}}$
  $E := 1$
  $A0 := (A(0), A(2), \ldots, A(2^k - 2))$
  $A1 := (A(1), A(3), \ldots, A(2^k - 1))$
  $Y0 :=$**FFT**$(k-1, A0)$
  $Y1 :=$**FFT**$(k-1, A1)$
  for $j = 0..(2^{k-1}-1)$
    $t := Y1(j)\cdot E$ /* E *értéke* eps^j
    $Y(j) := Y0(j) + t$
    $Y(j + 2^{k-1}) := Y0(j) - t$ /* $\text{eps}^{2^{k-1}} = -1$
    $E := E\cdot$**eps**
  return(Y)

Jelölje $T(k)$ az összeadások/kivonások szükséges darabszámát, látszik, hogy szorzásból (ami mindig egy egységgyökkel való szorzás) is ugyanennyi kell. Nyilván $T(k) = 2T(k-1)+2^k$, ahonnan $T(k) = k\cdot 2^k = n\log n$, ha $n = 2^k$.

Alkalmazásként térjünk vissza az $(n-1)/2$-ed fokú P és Q polinomok szorzására. Ez a szorzás elvégezhető a polinomok n-ed rendű diszkrét Fourier-transzformálásával, (amihez $O(n\log n)$ aritmetikai művelet kell), majd a $P(\varepsilon^j)$ és $Q(\varepsilon^j)$ értékek összeszorzásával, (ami $O(n)$ műveletet igényel), és végül az inverz Fourier-transzformáltak kiszámításával, ami ugyanazzal a módszerrel végezhető, mint az „előre" Fourier-transzformálás, (tehát ez is megvalósítható $O(n\log n)$ művelettel). Tehát a két polinom szorzata $O(n\log n)$ aritmetikai művelettel kiszámítható.

A diszkrét Fourier-transzformáltak másik alkalmazásaként megmutatjuk, hogy két n-bites szám hogyan szorozható össze $O(n\log^3 n)$ **bitművelettel**. (Ez a korlát sokkal ügyesebben számolva $O(n\log n\log\log n)$-re javítható.) Az ötlet az, hogy használjuk a polinomok szorzását. Legyenek $u = \overline{u_{n-1}\ldots u_1u_0}$ és $v = \overline{v_{n-1}\ldots v_1v_0}$ pozitív egészek bináris alakban. Tekintsük a következő polinomokat:

$$U(x) = u_0+u_1x+u_2x^2+\cdots+u_{n-1}x^{n-1} \quad \text{és}$$
$$V(x) = v_0+v_1x+v_2x^2+\cdots+v_{n-1}x^{n-1}.$$

Ekkor $u = U(2)$ és $v = V(2)$, és így $uv = U(2)V(2)$. Láttuk, hogy az UV polinomszorzat $O(n\log n)$ aritmetikai művelettel kiszámítható. A 2 behelyettesítése további $O(n)$ aritmetikai műveletet igényel.

Viszont most bitművelettel akarunk számolni, nem aritmetikai művelettel. A kiszámított egész számok (a szorzat polinom $2n-1$ darab együtthatója) nem nagyobbak $2n$-nél (és ez igaz minden közben kiszámított szám egész részére is), és így 2 behelyettesítése az UV polinomba maximum $O(n\log n)$ bitműveletet igényel.

A Fourier-transzformáltnak és inverzének kiszámításánál már óvatosabbnak kell lennünk, mivel az egységgyökök komplex irracionális számok, melyeket valamilyen véges, ámde elég pontos alakban kell kiszámítanunk. A komplex számokkal számolás nem okoz gondot, hiszen megfeleltethető kétszer annyi valós számmal való számolásnak, és a köztük végzett aritmetikai műveletek is megfelelnek valós számok közt végzett kettő, illetve hat aritmetikai műveletnek. De a szükséges pontossággal még foglalkoznunk kell.

Ha $2n = 2^k$ és a 2^k-adik primitív egységgyököt $8\log n$ bit pontossággal számoljuk ki, és menet közben a számolásoknál minden szám törtrészét $8\log n$ bitre kerekítjük, akkor azt állítjuk, hogy egyrészt minden egységgyököt és E értéket ki tudjuk számítani $6\log n$ bit pontossággal. Másrészt az odatranszformálás végén a Fourier együtthatókat $3\log n$ bit pontossággal megkapjuk, és a vissza-transzformálás után az UV polinom együtthatóit megkapjuk $\log n$ bit hibával; mivel ezek egészek, így egészre kerekítve már pontos értéket kapunk.

Először vegyük észre, hogy $8\log n$ bit pontosság itt maximum $1/n^8$ hibát jelent, és hogy két, maximum 1 abszolút értékű, δ_1, ill. δ_2 hibával megadott szám szorzásánál a hiba maximum $\delta_1+2\delta_2$ lesz. A rekurzió alacsonyabb szintjein szükséges primitív egységgyökök az előző szinten használt egységgyök négyzete, így a $(k-\ell)$-edik szint eps értékét maximum $3^\ell/n^8$ hibával kapjuk. Az E érték kiszámításánál vétett hiba is könnyen becsülhető indukcióval, a j. ciklusban a hiba maximum $2j$-szer az eredeti, tehát $2j\cdot 3^\ell/n^8 \le 2\cdot 2^{k-\ell-1}\cdot 3^\ell/n^8 \le 2^{k-\ell}\cdot 3^\ell/n^8 \le 3^k/n^8 < 1/n^6$.

Ha ezt egy $O(n)$-nél nem nagyobb egész számmal beszorozzuk, a hiba $O(n^{-5})$ alatt marad, sőt ha $O(n)$ ilyen tagot összeadunk, akkor is a hibánk $O(n^{-4})$ alatt lesz. Tehát az $U(\varepsilon^j)$-t és a $V(\varepsilon^j)$-t $O(n^{-4})$ hibával kapjuk, és mivel $|U(\varepsilon^j)| \le n$ valamint $|V(\varepsilon^j)| \le$ $\le n$, az $U(\varepsilon^j)V(\varepsilon^j)$ szorzatot legfeljebb $O(n^{-3})$ hibával kapjuk. Ezekre inverz Fourier-transzformációt alkalmazva az eredményt $O(n^{-1})$ hibával kapjuk. Ezeket az értékeket a legközelebbi egészre kerekítve megkapjuk uv-t. A számítás során minden szám $O(\log n)$ bites, így egy aritmetikai művelet köztük $O(\log^2 n)$ bitműveletet igényel. Így összesen $O(n \log^3 n)$ bitműveletet használtunk.

13.3. Nagy számok szorzása (Schönhage-Strassen)

Ez az algoritmus végig egész számokkal számol, pontosabban modulo $2^\ell + 1$ maradékosztályokkal (változó ℓ értékre). Bevezetésként mindenki gondolja meg, hogy egy $2n$ bites szám maradékát modulo $2^\ell + 1$ ki lehet számolni $O(n)$ lépésben. A továbbiakban R_ℓ jelöli a modulo $2^\ell + 1$ maradékosztályok gyűrűjét.

Tegyük fel, hogy az a és b pozitív egész számot szeretnénk összeszorozni, ahol a szorzat legfeljebb N bites. Ekkor a szorzást végezhetjük az R_N gyűrűben, hiszen a kiszámolt maradékosztály sztenderd reprezentánsa pont ab lesz. Ezt vissza fogjuk vezetni a $\mathbb{Z}[x] \mod (x^K + 1)$ polinom-gyűrűben való szorzásra, ahol K értéke $\sqrt{N}$ lesz. Ezt továbbá visszavezetjük az $R_n[x] \mod (x^K + 1)$ polinom-gyűrűben való szorzásra, ahol n értéke $2K = 2\sqrt{N}$ lesz. Ezt pedig a Fourier-transzformált egy változata segítségével rekurzívan visszavezetjük az R_n gyűrűben való szorzásra. Ezután ha n elég kicsi, akkor hívjuk meg a Karacuba-féle módszert, egyébként pedig a fenti eljárást hívjuk meg rekurzívan. Az algoritmushoz szükséges algebrai/számelméleti állításokat csak kimondjuk, bizonyításukat az olvasóra bízzuk.

Feltehetjük, hogy $N{=}2^d$, részletesen csak a páros d esetére adjuk meg az algoritmust.

Legyen $K = 2^{d/2}$ és $n = 2K$. Ekkor $K^2 = N$, és a, illetve b egyértelműen felírható $a = \sum_{i=0}^{K-1} a_i 2^{iK}$, illetve $b = \sum_{i=0}^{K-1} b_i 2^{iK}$ alakba, ahol $a_i, b_i < 2^K$, azaz egyszerűen az N bites inputokat felbontjuk K bites részekre. Legyen $A(x) = \sum_{i=0}^{K-1} a_i x^i$ és $B(x) =$ $= \sum_{i=0}^{K-1} b_i x^i$, valamint $C(x) = A(x)B(x)$. Legyen C felírása $C(x) = \sum_{i=0}^{2K-1} c_i x^i$. Ekkor a keresett eredmény $C(2^K) \mod 2^N + 1$, ami

$$(c_0 - c_K) + (c_1 - c_{K+1})2^K \ldots + (c_{K-2} - c_{2K-2})2^{K(K-2)} + (c_{K-1} - c_{2K-1})2^{K(K-1)}.$$

Könnyű látni, hogy minden $i = 0,1,\ldots,K-1$ értékre

$$((i+1) - K)2^{2K} \le c_i - c_{i+K} < (i+1)2^{2K}.$$

Legyen $c_i^* = c_i - c_{i+K} \mod K2^{2K}$, tehát elég ezeket meghatározni minden i-re. Legyen továbbá $d_i = c_i^* \mod K$ és $\bar{c}_i = c_i^* \mod (2^{2K} + 1)$. Ezekből

$$c_i^* = (2^{2K} + 1)\big[(d_i - \bar{c}_i) \mod K\big] + \bar{c}_i.$$

Először a d_i mennyiségeket számoljuk ki. Legyen $a_i^* = a_i \mod K$ és $b_i^* = b_i \mod K$, valamint

$$\bar{a} = \sum_{i=0}^{K-1} a_i^* K^{3i}, \quad \bar{b} = \sum_{i=0}^{K-1} b_i^* K^{3i}.$$

Számítsuk ki a $\bar{d} = \bar{a}\bar{b} = \sum_{i=0}^{2K-2} \bar{d}_i K^{3i}$ szorzatot pl. Karacuba módszerével, ekkor $d_i = \bar{d}_i - \bar{d}_{K+i} \mod K$. Gondoljuk meg, hogy eddig (elég nagy N-ekre) csak $O(N)$ bitműveletet végeztünk, mivel minden szám kisebb, mint K^{6K}, azaz legfeljebb $\ell = 6\sqrt{N} \cdot \log N$ bitesek, és $O(\ell^{1,59}) = O(N)$ elég nagy N értékekre.

Már csak R_n-ben kell kiszámolnunk a $\bar{c}_i$ számokat. Könnyű ellenőrizni, hogy R_n-ben a 4 egy primitív $2K$-adik egységgyök. Ezenkívül bármilyen 2-hatvánnyal gyorsan tudunk szorozni, ráadásul osztani is, mivel $2^\ell \cdot (-2^{n-\ell}) \equiv 1 \pmod{2^n+1}$, tehát az osztás 2^ℓ-lel ekvivalens a $(-2^{n-\ell})$-lel való szorzással.

Legyen $\hat{a}_j = \sum_{i=0}^{K-1} 4^{(2j+1)i} a_i$, valamint $\hat{b}_j = \sum_{i=0}^{K-1} 4^{(2j+1)i} b_i$ (az R_n gyűrűben). Ekkor nem nehéz belátni, hogy

$$\bar{c}_i = \frac{1}{4^i K} \sum_{j=0}^{K-1} 4^{-2ij} \hat{a}_j \hat{b}_j.$$

Ezt persze meg kell gondolni, valamint azt is, hogy ez a fajta transzformált is számolható az FFT eljárással, és az $\hat{a}_j \hat{b}_j$ értékek kiszámításához pedig K db rekurzív hívás kell R_n-ben.

Tehát az összes lépésszám a rekurzió legfelső szintjén $O(N)$ plusz a három FFT, melyek összesen $O(K \log K)$ lineáris műveletet igényelnek $O(K)$ méretű számokkal (tehát összesen $O(N \log N)$ bitműveletet), ezenkívül K darab rekurzív hívás R_n-ben. Tehát, ha $T(N)$ a bitműveletek száma az R_N-ben való szorzásra, akkor $T(N) \leq c \cdot N \log N + K \cdot T(n) = c \cdot N \log N + \sqrt{N} \cdot T(2\sqrt{N})$.

Innen kis számolással adódik, hogy $T(N) = O(N \log N \log \log N)$.

Megjegyzés: Ha $N = 2^d$ és d páratlan, akkor legyen $K = 2^{(d+1)/2}$. Végiggondolható, hogy apróbb változtatásokkal (pl. 4 helyett 16-ot választva egységgyöknek) a fenti algoritmus és elemzés működik.

Megjegyzés: Schönhage és Strassen 1971-ben azt sejtették, hogy a lépésszám levihető $O(N \log N)$-re, de ez alá már nem. A fontos részt 2019 márciusában Harvey és van der Hoeven bizonyították. 2026 szeptemberében az OpenAI (nem ellenőrzött) eredménye szerint szorozni lehet $O(N \log^{1-\varepsilon} N)$ bitművelettel is, tehát megcáfolta a sejtés második felét.

14. fejezet

Dinamikus programozás

14.1. Optimális bináris keresőfa

Statikus szótárat akarunk tárolni úgy, hogy feltételezzük, hogy rengetegszer kell majd benne keresni. Ezen keresések összidejét akarjuk minimalizálni.
Adottak:

- $S = \{a_1 < a_2 < \cdots < a_n\}$ a szótár elemei.
- a_i-t p_i valószínűséggel keressük.
- ha $a_i < b < a_{i+1}$, az ilyen b-ket q_i valószínűséggel keressük.
  - ha $b < a_1$, akkor b-t q_0, ha pedig $a_n < b$, akkor b-t q_n valószínűséggel keressük.

Egy adott T keresőfa esetén egy keresés várható lépésszáma, melyet a fa *költségének* hívunk:

$E(\text{keresés}) = c(T) := \sum_{i=1}^{n} p_i(d(a_i)+1) + \sum_{i=0}^{n} q_i d(i)$, ahol

- $d(a_i)$: az a_i-t tartalmazó csúcs mélysége; a $+1$ azért kell, mert a gyökér keresése sem 0 lépés.
- $d(i)$: az i. (balról jobbra számozva) fiktív levél mélysége.

Feladat: Minimális költségű bináris keresőfa konstrukciója, azaz olyan T, amelyre $c(T)$ minimális.

Megoldás alapötlete: Ha a_k lenne az optimális T fa gyökerében, akkor a bal T_1 részfában az $a_1, \ldots, a_{k-1}$ szavak, a jobb T_2 részfában az $a_{k+1}, \ldots, a_n$ szavak helyezkednének el. Ha T az optimum, akkor szükségképpen T_1 és T_2 is optimális fa (a bennük szereplő a_i-k által meghatározott részfeladatra). Ez a szuboptimalitás elve.

Megjegyzés: Azok a feladatok, amelyekre a szuboptimalitás elve teljesül, többnyire megoldhatóak hatékonyan, erre való a dinamikus programozás, ahol először is jól definiálunk részfeladatokat, utána ezeket a megfelelő sorrendben kiszámoljuk, a régebbi részfeladatok megoldását jól felhasználva.

$T_{i,j}$:= optimális fa az $a_{i+1} \dots a_j$ szavakon. Ennek gyökere $r_{i,j}$, költsége $c_{i,j}$, súlya pedig $w_{i,j} := q_i + p_{i+1} + q_{i+1} + \dots + p_j + q_j$ ($w_{i,j}$ az a valószínűség, hogy belépünk egy $T_{i,j}$ fába).

Inicializálás:
$T_{i,i} = \emptyset$; $c_{i,i} = 0$; $w_{i,i} = q_i$.

Nekünk a $T_{0,n}$ fát kell megkeresni. Vegyük észre, hogy ha tudnánk, hogy $r_{i,j} = k$, akkor a szuboptimalitás elve miatt a $T_{i,j}$ fa gyökerének bal részfája $T_{i,k-1}$, jobb részfája pedig $T_{k,j}$ lesz. Ezért ekkor a költsége

$$c_{i,j} = (c_{i,k-1} + w_{i,k-1}) + p_k + (c_{k,j} + w_{k,j}) = c_{i,k-1} + c_{k,j} + w_{i,j}$$

lesz, mivel a $T_{i,j}$ fában minden csúcs mélysége eggyel nagyobb, mint a $T_{i,k-1}$, ill. a $T_{k,j}$ fákban. Vegyük észre, hogy a költségben $w_{i,j}$ állandó, nem függ a k-tól.

Ezek alapján könnyen készíthetünk egy rekurzív algoritmust:

$R(i,j)$:
$c'_{i,j} := \infty$
for $k := i+1..j$
  $C_1 := R(i, k-1)$
  $C_2 := R(k, j)$
  if $C_1 + C_2 < c'_{i,j}$ **then** $c'_{i,j} := C_1 + C_2$; $r_{i,j} := k$
return$(c_{i,j} := c'_{i,j} + w_{i,j})$

Ez az algoritmus azért **exponenciális**, mert ugyanazt a dolgot sokszor számoljuk ki. A hívások száma az ún. Catalan-szám lesz, ami kb. $\frac{1}{c \cdot n \cdot \sqrt{n}} \cdot 2^{2n}$. Hogy ezt elkerüljük, dinamikus programozással oldjuk meg a feladatot.

Az algoritmus (a fenti Inicializálás után) a következő (az argmin itt azt a k értéket adja vissza, amelyre a min felvétetik):
for $l = 1..n$
  for $i = 0..n-l$
    $j := i + l$
    $c_{i,j} := w_{i,j} + \min_{i<k\le j}(c_{i,k-1} + c_{k,j})$
    $r_{i,j} := \operatorname{argmin}_{i<k\le j}(c_{i,k-1} + c_{k,j})$

Lépésszám: $O(n^3)$

Megjegyzés: Az $r_{i,j}$ értékek segítségével maga az optimális fa is könnyen visszafejthető, gyökere $r_{0,n} =: k$, bal gyereke $r_{0,k-1}$, jobb gyereke $r_{k,n}$ és így tovább.

15. fejezet

Mélységi keresés/bejárás és alkalmazásai

Egy kezdőpontból kiindulva addig megyünk egy-egy él mentén, ameddig el nem jutunk egy olyan csúcsba, amelyből már nem tudunk tovább menni olyan csúcsba, amelyet még nem „látogattunk meg". Ekkor visszamegyünk az út utolsó előtti csúcsához, és onnan próbálunk egy másik él mentén tovább menni. Ha ezen az ágon is minden csúcsot már bejártunk, ismét visszamegyünk egy csúcsot, és így tovább. Közben minden csúcshoz feljegyzünk két sorszámot: a mélységi számát, azaz hogy hányadiknak láttuk meg, és a befejezési számát, azaz hogy hányadikként vizsgáltuk át (léptünk vissza belőle).

52. Algoritmus (Mélységi keresés).

$S := 0;\ \ SZ := 0$
  for $i = 1..n \quad p(i) := 0$ /* *Inicializáljuk*
  for $i = 1..n$
    if $p(i)=0$ **then** $p(i):=i;$ **MB**(i) /* *Minden nem látott csúcsra elvégezzük MB-t*

  MB(u)**:**
  $SZ{+}{+};\ MSZ(u):=SZ$ /* *Amikor először megyünk be u-ba, beállítjuk a mélységi számát*
  for $uv \in E$
    if $p(v) = 0$ **then** $p(v) := u;$ **MB**(v)
  $S{+}{+};\ \ BSZ(u) := S$ /* *u-t átvizsgáltuk, befejezési számát beállítjuk.*

Lépésszám: $O(n+m)$

57. Tétel. *Irányítatlan gráfban mélységi keresés után minden $uv \in E$ élre u őse v-nek vagy fordítva (azaz nincs keresztél).*
Az u csúcs őse a v csúcsnak akkor és csak akkor, ha $MSZ(u) \leq MSZ(v)$ és $BSZ(u) \geq$ $\geq BSZ(v)$.

Az így keletkező $\big\{\{p(i), i\} \mid p(i) \neq i\big\}$ élhalmazt mélységi feszítőerdőnek (összefüggő gráf esetén mélységi feszítőfának) hívjuk.

15.1. Erősen összefüggővé irányítás

Feladat: Az input G irányítatlan összefüggő gráfot szeretnénk erősen összefüggővé irányítani ha lehet; ha pedig nem, akkor kiírni egy elvágó élet.

58. Tétel (Robbins). *Egy G gráfnak akkor és csak akkor van erősen összefüggő irányítása, ha G 2 él-összefüggő.*

53. Algoritmus (Erősen összefüggővé irányítás).

Elvégezzük a mélységi keresést, közben a fa-éleket (ezek a $\{p(v), v\}$ élek) lefelé, tehát a v felé irányítjuk. A többi élet felfelé kell irányítani, tehát azon csúcsa felé, amelynek az MSZ mélységi száma kisebb.

Ez erősen összefüggő irányítást ad, amennyiben a gráf kétszeresen él-összefüggő volt. Ellenőrzés: A $\overleftarrow{G}$ fordított gráfon is indítunk egy mélységi keresést 1-ből. Ha minden csúcsot meglátunk, akkor valóban erősen összefüggő irányítást kaptunk. Egyébként keressük meg a második keresés során nem látott csúcsok közül azt az x csúcsot, amelynek az első keresés során adott mélységi száma a legkisebb. Ekkor kiírathatjuk, hogy $p(x)x$ az input gráf elvágó éle.

Lépésszám: $O(m)$

15.2. 2-összefüggőség tesztelése

Feladat: El akarjuk dönteni, hogy az input G irányítatlan összefüggő gráf 2-szeresen összefüggő-e, ha pedig nem, akkor meg akarjuk találni az összes elvágó csúcsot.

54. Algoritmus (2-összefüggőség tesztelése).

Minden csúcshoz két mennyiséget definiálunk:

$$LEGM(u) := \min(MSZ(v) \mid uv \in E)$$

$$FEL(u) := \min(LEGM(v) \mid v \text{ leszármazottja } u\text{-nak}).$$

Az első mennyiség nyilván a mélységi keresés során könnyen számolható. A második is számolható alulról felfelé, a következő rekurzív összefüggést használjuk:

$$FEL(u) = \min\big(LEGM(u),\ \min\{FEL(v) \mid p(v) := u\}\big).$$

Lépésszám: $O(m)$

59. Tétel. *Az u csúcs elvágó akkor és csak akkor, ha*

- *u gyökér, és egynél több gyereke van, vagy*
- *van olyan v gyereke, amelyre $FEL(v) = MSZ(u)$.*

A gráf kétszeresen összefüggő akkor és csak akkor, ha nincs benne elvágó csúcs.

15.3. Erősen összefüggő komponensek

Irányított gráfokon a mélységi keresést ugyanúgy végezzük, itt azonban lehetnek keresztélek is, azonban csak „jobbról balra".

Ha uv a gráf egy éle, akkor a következő 3 lehetőség egyike teljesül:

- $MSZ(u) < MSZ(v)$ és $BSZ(u) > BSZ(v)$, ekkor v valódi leszármazottja u-nak, és uv faél vagy előreél.
- $MSZ(u) > MSZ(v)$ és $BSZ(u) < BSZ(v)$, ekkor u valódi leszármazottja v-nek, és uv visszaél.
- $MSZ(u) > MSZ(v)$ és $BSZ(u) > BSZ(v)$, ekkor uv keresztél.

43. Definíció. *G irányított gráf. $x \sim y$, ha van irányított út x-ből y-ba és y-ból x-be is. Ennek ekvivalencia osztályait erősen összefüggő komponenseknek nevezzük.*

A következő feladattal foglalkozunk: az input egy G irányított gráf. Az erősen összefüggő komponenseket szeretnénk meghatározni.

Könnyű meggondolni, hogy ha minden erősen összefüggő komponenst egyetlen csúcsra húzunk össze, akkor aciklikus gráfot kapunk. Ennek az „összehúzott" gráfnak egy topologikus sorrendjét is meg szeretnénk határozni.

55. Algoritmus (Erősen összefüggő komponensek).

- Először csinálunk egy mélységi keresést, a befejezési számozás szerint a csúcsokat egy B tömbbe rakjuk.
- Előállítjuk a $\overleftarrow{G}$ fordított gráfot.
- Ebben végzünk egy újabb mélységi keresést, de a külső ciklust lecseréljük: az első keresés befejezési számozása szerinti csökkenő sorrendben megyünk végig a csúcsokon.

  for $i = n..1 \quad (-1)$
  $w := B(i)$
  if $p(w) = 0$ **then** $p(w) := w$; **MB**(w)
- A második keresés fáinak csúcshalmazai lesznek az erősen összefüggő komponensek.
- Az erősen összefüggő komponensek megtalálásának sorrendje az összehúzott gráf topologikus sorrendje. Speciálisan, ha G aciklikus volt, akkor minden erősen összefüggő komponens egy csúcsból áll, tehát a befejezési szám szerinti csökkenő sorrend egy topologikus sorrend.

Lépésszám: $O(m)$

16. fejezet

Legrövidebb utak

16.1. Dijkstra algoritmusának további alkalmazásai

16.1.1. Legszélesebb, legolcsóbb vagy legprofitábilisabb legrövidebb út

Felhasználjuk a 7.4 és a 7.3 fejezeteket.

Adott egy G irányított gráf, és élein két különböző nemnegatív függvény, egy c hosszúság és egy w szélesség vagy költség vagy túlélési valószínűség vagy profit. Adott s csúcsból keresünk adott/minden t csúcsba olyan utat, mely egyrészt egy legrövidebb út, másrészt a legrövidebb utak közül a lehető legszélesebb vagy legolcsóbb vagy legbiztonságosabb vagy legnagyobb összprofitú.

Elsőként keressünk a Dijkstra-algoritmussal s-ből minden más csúcsba legrövidebb utat, ezáltal kapunk $K(v)$ távolságokat. Közben vagy utána kiválogatjuk a szoros éleket, majd ezek gráfján futtatjuk a megfelelő algoritmust.

60. Tétel. *A Dijkstra futása közben minden $u, v \in P \cup S$ csúcsokra, ha uv él, és legalább u és v egyike S-ben van, akkor*

$$K(u) + c(uv) \geq K(v).$$

Ha $p(v) = u$, akkor $K(u) + c(uv) = K(v)$.

A tételt, és a következő tételeket is általánosabban a következő alfejezetben bizonyítjuk.

Egy uv élet **szoros**-nak hívunk, ha $K(u) + c(uv) = K(v)$. Az előző tétel szerint, ha minden csúcs elérhető s-ből, akkor minden csúcsba lép be szoros él, kivéve s-et.

61. Tétel. *Jelölje D a szoros élek gráfját. Ekkor minden t (s-ből elérhető) csúcsra és minden D-beli P s-ből t-be vezető útra igaz, hogy P egy legrövidebb $s \rightsquigarrow t$ út.*

62. Tétel. *Jelölje D a szoros élek gráfját, és tegyük fel, hogy c szigorúan pozitív. Ekkor D aciklikus.*

Tehát, ha a D gráfban keresünk s-ből t-be egy legolcsóbb vagy legszélesebb vagy legbiztonságosabb utat a 7.2., a 7.4., illetve a 7.3. fejezetben leírtak szerint, az pont a célunknak megfelelő út lesz.

Ha pedig c szigorúan pozitív, akkor az előző tétel szerint D aciklikus, így tudunk benne legnagyobb profitú utat keresni az a 7.5. fejezet szerint.

Hátra van még a D gráf előállítása algoritmikusan. Erre több megoldás is létezik, talán a legegyszerűbb az, ha a Dijkstra lefutása után kiválogatjuk a definíció szerint a szoros éleket.

Megjegyzések: Sokszor hasznosabb lenne pl. a legrövidebbnél legfeljebb 10%-kal hosszabb utak közül keresni a legolcsóbbat/legszélesebbet. Ez a probléma viszont NP-nehéz.

Viszont érdekes módon a közel legszélesebb utak közül a legrövidebbet könnyű megtalálni. Legyen a legszélesebb $s \rightsquigarrow t$ út szélessége például 100. Ekkor, ha most a D segédgráfba azokat az éleket vesszük be, amelynek szélessége legalább 90, és D-ben keresünk legrövidebb $s \rightsquigarrow t$ utat, akkor pont egy ilyen feladatot oldunk meg.

16.1.2. Absztrakt Dijkstra

Ebben az alfejezetben a fő szerepet az alábbi algebrai struktúrák játsszák. Jelöljön $\mathbf{F} = \{F, \oplus, 0, \prec\}$ egy (teljesen) rendezett kommutatív null-elemes félcsoportot (röviden **RKF**-et), amit az alábbi tulajdonságok által definiálunk:

i) $\oplus$ kétváltozós asszociatív és kommutatív művelet az F alaphalmazon, a 0 egységelemmel,

ii) $\prec$ egy teljes rendezés F-en,

iii) Minden $a, b, c \in F$-re $0 \preceq a$, valamint ha $a \preceq b$, akkor $a \oplus c \preceq b \oplus c$.

Lássunk néhány fontos példát:

- $\mathbf{F}_1$ A nemnegatív egészek vagy valósak az összeadással és a szokásos rendezéssel.
- $\mathbf{F}_2$ A nemnegatív egészek vagy valósak kibővítve a $+\infty$ szimbólummal, a minimum művelettel és a *fordított* rendezéssel , itt a nulla elem a $+\infty$. Ez jó az út szélességére.
- $\mathbf{F}_2'$ A nemnegatív egészek vagy valósak, a maximum művelettel és a normális rendezéssel. Ezzel minimális költségű feszítőfát lehet keresni páronként különböző élsúlyok esetén.[1]
- $\mathbf{F}_3$ $F_3 = \{x \in \mathbb{R} \mid 0 < x \leq 1\}$, $\oplus$ a szorzás, a nulla elem az 1, a rendezés a fordított. Ez pl. arra jó, ha az éleken a túlélési valószínűségek vannak.

Fontos megjegyezni, hogy két (vagy több) RKF **direkt szorzata** is RKF. Itt a műveleteket koordinátaként végezzük, az összehasonlítást pedig lexikografikusan; a null-elem az lesz, amelynek minden koordinátája az adott RKF null-eleme.

Legyen $\mathbf{F} = \{F, \oplus, 0, \prec\}$ egy RKF, és tegyük fel, hogy a G gráf élein egy $c : E \to F$ súlyozás adott. Keressük az $s \in V$ csúcsból az összes többibe a minimális $\mathbf{F}$-súlyú utat, ahol egy Q út súlya $c(Q) := \bigoplus_{e \in Q} c(e)$. Az argmin itt egy olyan v csúcsot választ, amire $K(v) \preceq K(w)$ minden $w \in P$-re. Feltesszük, hogy a $\prec$ reláció eldönthető és a $\oplus$

[1] Ha az élsúlyok páronként különbözőek, akkor uv él akkor csak akkor van az optimális fában, ha az $\mathbf{F}_2'$ szerinti legrövidebb $u \rightsquigarrow v$ út az egyélű.

művelet kiszámítható konstans időben, tehát a kupacműveletek a szokott lépésszámmal lefutnak.

56. Algoritmus (Absztrakt Dijkstra).

ABSTR-DIJKSTRA(G, s)**:**

$P := \{s\};\ S := \emptyset;\ M := V \setminus \{s\};\ K(s) := 0;\ p(s) := s$
while $P \neq \emptyset$
$u := \operatorname{argmin}\{K(v) \mid v \in P\}$ /* *u a P azon csúcsa, melyre $K(u)$ minimális.*
$S \leftarrow u \leftarrow P$
for $u\underline{v} \in E$
if $v \in P$ **&&** $K(u) \oplus c(uv) \prec K(v)$ **then**
$K(v) := K(u) \oplus c(uv);\ p(v) := u$
if $v \in M$ **then**
$K(v) := K(u) \oplus c(uv);\ p(v) := u;\ P \leftarrow v \leftarrow M$

63. Tétel. *Ez az algoritmus szomszédsági mátrixos megadás esetén $O(n^2)$, éllistás megadás és bináris kupac esetén $O(m \log n)$ lépésben megtalálja minden $v \in V$ csúcsba a minimális súlyú utat (amelybe van út s-ből). Ennek súlya $K(v)$, az út maga a $p(v)$ értékek segítségével fejthető vissza.*

3. Lemma. *Minden ciklusvégkor minden $u \in S$ és $v \in P$ csúcsra $K(u) \preceq K(v)$.*

Bizonyítás. Tegyük fel, hogy a ciklus elején igaz. Mivel P-ből S-be a legkisebb kulcsú u csúcsot rakjuk át, az átrakás után is igaz marad. Valamint nyilván $K(s') \preceq \preceq K(u)$ minden $s' \in S$-re, hiszen a ciklus elején u még a P halmazban volt.

Ha $K(v)$ most kap értéket, vagy értéke változik, akkor ez $K(u) \oplus c(uv)$ lesz. Az előző észrevétel, valamint az iii) tulajdonság miatt $K(s') \preceq K(u) \preceq K(u) \oplus c(uv)$.

64. Tétel. *Minden uv élre, ha $u, v \in S \cup P$ de legalább az egyik S-ben van, akkor*

$$K(v) \preceq K(u) \oplus c(uv).$$

Továbbá, ha $p(v) = u$, akkor $K(v) = K(u) \oplus c(uv)$. Ekkor azt mondjuk, hogy az uv él **szoros**.

Bizonyítás. Első eset: u került be hamarabb S-be.
Ekkor u bekerülése után $K(v) \preceq K(u) \oplus c(uv)$, később $K(v)$ csak csökkenhet. Ha a végén $p(v) = u$, akkor $K(v) = K(u) \oplus c(uv)$ lett, és később már nem változott.

Második eset: v került be hamarabb S-be.
Ekkor $K(v) \preceq K(u)$ az előző lemma miatt, és mivel $K(u)$ csak addig változik, amíg $u \in P$, ez végig igaz marad. Az iii) tulajdonság miatt $K(v) \preceq K(u) \preceq K(u) \oplus c(uv)$. Ez esetben $p(v) = u$ nem lehetséges.

A 63. Tétel bizonyítása. A végén minden csúcs S-ben van (esetleg, ha nem minden csúcs elérhető S-ből, azok M-ben maradnak, azokról nem állítunk semmit). Egyrészt

egy tetszőleges Q útra, melynek csúcsai $s = v_0, v_1, \ldots, v_k = v$, igaz az, hogy $K(v) \preceq \preceq K(v_{k-1}) \oplus c(v_{k-1}v_k) \preceq \ldots \preceq K(s) \oplus \bigoplus_{i=1}^{k} c(v_{i-1}v_i) = c(Q)$ az iii) feltétel miatt, tehát nincs $K(v)$-nál kisebb súlyú út. Másrészt arra a Q útra, melynek utolsó csúcsa v, az azelőtti $p(v)$, az azelőtti $p(p(v))$, stb., az első meg s, nyilván $K(s) \oplus c(Q) = K(v)$, a $K(s) := 0$ inicializálás miatt tehát $K(v) = c(Q)$.

Például, ha $\mathbf{F} = \mathbf{F}_3$, akkor a legbiztonságosabb utat keressük, ha $\mathbf{F} = \mathbf{F}_1 \times \mathbf{F}_1$, akkor a legrövidebb utak közül a legolcsóbbat, ha $\mathbf{F} = \mathbf{F}_1 \times \mathbf{F}_2$, akkor a legrövidebb utak közül a legszélesebbet, ha $\mathbf{F} = \mathbf{F}_1 \times \mathbf{F}_3$, akkor a legrövidebb utak közül a legbiztonságosabbat, ha pedig $\mathbf{F} = \mathbf{F}_1 \times \mathbf{F}_3 \times \mathbf{F}_2$, akkor egy Dijkstrával megkereshetjük a legrövidebb utak közül a legbiztonságosabbak közül a legszélesebbet.

Egy RKF-et nevezzünk **szigorúan** rendezettnek, ha

iii') $a \prec b$ esetén $a \oplus c \prec b \oplus c$ minden c-re.

Például $\mathbf{F}_1$ és $\mathbf{F}_3$ szigorúan rendezettek, míg $\mathbf{F}_2$ és $\mathbf{F}_2'$ nem azok. Szigorúan rendezett RKF-ek direkt szorzata szigorúan rendezett.

Egy uv él szoros, ha $K(v) = c(uv) \oplus K(u)$.

65. Tétel. *Ha a c súlyfüggvény sehol sem 0, és az RKF szigorúan rendezett, akkor a szoros élek D gráfja aciklikus.*

Bizonyítás. Indirekt legyen C egy kör, melynek minden éle szoros. A kör éleire felírva a szorosság definícióját, majd ezeket „összeadva" kapjuk, hogy

$$\bigoplus_{v \in V(C)} K(v) = \bigoplus_{u \in V(C)} K(u) \oplus \bigoplus_{uv \in C} c(uv).$$

Ebből a szigorú rendezettség iii') feltétele miatt $\bigoplus_{uv \in C} c(uv) = 0$, ahonnan a kör minden élére $c(uv) = 0$, ezt pedig kizártuk. (Igazából az is elég lett volna feltételként, hogy a 0 súlyú élek aciklikus gráfot alkossanak.)

A szoros élek D segédgráfját könnyen elkészíthetjük $O(m)$ időben akár menet közben, akár a Dijkstra lefutása során.

66. Tétel. *Ha $\mathbf{F}$ RKF szigorúan rendezett és a 0 súlyú élek nem alkotnak kört, akkor a szoros élek aciklikus D gráfjában minden t (s-ből elérhető) csúcsra és minden D-beli P s-ből t-be vezető útra igaz, hogy P egy $\mathbf{F}$-legrövidebb $s \rightsquigarrow t$ út.*

Bizonyítás. Bármely P s-ből t-be vezető útra a D-ben az élek szorossága miatt

$$\bigoplus_{v \in V(P-s)} K(v) = \bigoplus_{u \in V(P-t)} K(u) \oplus \bigoplus_{uv \in P} c(uv),$$

azaz mindkét oldalhoz hozzáadva $K(s) = 0$-át és $K(t)$-t

$$\bigoplus_{v \in V(P)} K(v) \oplus K(t) = \bigoplus_{v \in V(P)} K(v) \oplus \bigoplus_{uv \in P} c(uv).$$

Ebből iii') miatt

$$K(t) = \bigoplus_{uv \in P} c(uv).$$

A 63. tétel miatt tehát P egy $\mathbf{F}$-legrövidebb $s \rightsquigarrow t$ út.

Ekkor a 7.5. fejezet algoritmusának a megfelelő módosítása szerint maximális $\mathbf{F}$-súlyút is tudunk gyorsan keresni a D gráfban: Cseréljük le a $+$ jelet $\oplus$-ra, és a $<$ jelet a $\succ$ jelre.

Tehát pl. $\mathbf{F} = \mathbf{F}_1 \times \mathbf{F}_3$ esetén, a szoros élek gráfja aciklikus (amennyiben vagy nincs 0 hosszúságú kör, vagy pedig a soha-el-nem-romló élekből nincs kör), így a legrövidebb utak közül a legbiztonságosabbak közül ki tudjuk választani pl. a legprofitábilisabbat (amennyiben nincsen kör olyan élekből, amelyek egyszerre 0 hosszúak és soha nem romlanak el).

16.1.3. Időfüggő legrövidebb út

Sok feladatnál természetes, hogy egy élen való átjutás ideje függ az indulási időponttól, pl. gondoljunk egy tömegközlekedési hálózatra. Itt az input a G irányított gráfon kívül minden uv élre egy $a_{uv}(\tau)$ függvény, amely azt adja meg, hogy ha az u csúcsból az τ időpontban indulunk, akkor mikor érkezünk meg a v csúcsba. Természetesen $a_{uv}(\tau) \geq \tau$.

A legkorábbi érkezési időket szeretnénk kiszámítani, ha adott a kiindulási s csúcs, és az indulás τ_0 időpontja.

Megjegyzések: Ez a feladat ilyen általánosan NP-nehéz, ezért szokás még feltenni az ún. FIFO tulajdonságot, miszerint ezek a függvények monoton növekvőek (nem szigorúan), azaz aki később indul el egy élen, nem érkezhet meg hamarabb. Ezzel lényegében ekvivalens az, hogy azt tesszük fel, hogy az u csúcsba érkezés után szabad ott várakoznunk.

Fontos kérdés, hogy ezek a függvények hogyan vannak megadva. A gyakorlatban általában ezek szakaszonként lineáris (de nem folytonos) függvények, másrészt periodikusak is (pl. 24 óra periódussal), ezért elég könnyen megadhatóak.

Erre a feladatra Dreyfus algoritmusa ad megoldást, mely a Dijkstra-algoritmus mintájára a következőket teszi: most is S lesz az a halmaz, ahová már kiszámítottuk a legkorábbi érkezési időket (melyeket itt is $K(u)$-val jelölünk), és P lesz azon csúcsok halmaza, amelyekbe vezet él S-ből. Egy P-beli v csúcsnak a K kulcsa viszont most a legkorábbi érkezési időpont, feltéve, hogy S-beli csúcsból érkezünk, tehát $K(v) = \min_{u \in S;\, uv \in E} a_{uv}(K(u))$. Az algoritmus során itt is a minimális kulcsú elemet rakjuk át P-ből S-be, könnyű látni, hogy a kulcsok karbantartása és a helyesség bizonyítása analóg módon megy a Dijkstra-algoritmusnál tanultakkal.

16.2. Legrövidebb út keresése konzervatív súlyozású irányítatlan gráfban

44. Definíció. *Egy $G=(V,E)$ irányítatlan gráf egy $c: E \to \mathbb{R}$ súlyozását konzervatívnak nevezzük, ha G minden körének összsúlya nemnegatív.*

45. Definíció. *Adott egy $G=(V,E)$ irányítatlan gráf és egy $T\subseteq V$ csúcshalmaz, melyre $|T|$ páros. Egy $F\subseteq E$ élhalmazt T-kötésnek hívunk, ha $d_F(v)$ akkor és csak akkor páratlan, ha $v\in T$.* (Itt $d_F(v)$ az F-beli fok, azaz a v csúcsra illeszkedő F-élek száma.)

Adott egy $G=(V,E)$ irányítatlan gráf egy c konzervatív súlyozással, valamint $s,t\in \in V$. Keresünk egy legrövidebb $s\rightsquigarrow t$ utat. Felhasználjuk, hogy tetszőleges súlyozás esetén a minimális súlyú T-kötés keresése megoldható $O(n^3)$ lépésben. Ezért a súlyozás konzervativitását is le tudjuk ellenőrizni, ha minimális súlyú T-kötést keresünk a $T=\emptyset$ esetre.

Az algoritmusunk a következő: legyen $T=\{s,t\}$, és keressünk egy F minimális súlyú T-kötést. Könnyű látni, hogy F felparticionálható egy $s\rightsquigarrow t$ útra és néhány körre. Mivel a körök nemnegatívak, ezért a kapott $s\rightsquigarrow t$ út minimális költségű.

16.3. Minimális átlagú kör keresése

A Bellman–Ford-algoritmus lényegében az i. iterációban megtalálja a legrövidebb olyan sétákat, melyek legfeljebb i élből állnak (igazából ez így nem pontos, lásd a 31. állítást). Ezt könnyű úgy módosítani, hogy az i. ciklus végén $K(V)$ (ha véges) a legrövidebb pontosan i élből álló séta hossza legyen.

Egy séta (vagy kör) átlagának a hossz/élszám hányadost nevezzük. Karptól származik a következő algoritmus, amely egy irányított gráf minimális átlagú körét számolja ki. Jelölje $d_i(v)$ a legrövidebb olyan séta hosszát, mely akárhonnan indulhat, pontosan i élből áll, és v-be érkezik. Ezeket sem nehéz meghatározni a Bellman–Ford-algoritmus mintájára: nyilván minden v-re $d_0(v)=0$ és $d_{i+1}(v)=\min(d_i(u)+c(uv)\mid uv\in E)$.

67. Tétel (Karp). *A minimális átlagú kör átlaga*

$$\min_{v\in V}\max_{0\le k\le n-1}\frac{d_n(v)-d_k(v)}{n-k}.$$

Megjegyzések: Itt nincs szükség arra, hogy a hosszúság függvény konzervatív legyen, mivel ha minden él hosszához hozzáadjuk ugyanazt az α valós számot, akkor a minimális körátlag is α-val nő, és a fenti kifejezés is.

Nem nehéz megmutatni, hogy megfelelő szülő-pointerek karbantartása esetén a minimumot adó v csúcsból visszafelé lépkedve egy minimális átlagú kört is meg tudunk kapni.

16.4. Suurballe algoritmusa

Adott G erősen összefüggő irányított gráf nemnegatív c élsúlyokkal. Keresünk s-ből t-be egy P és egy Q utat, amik éldiszjunktak, és $c(P)+c(Q)$ minimális. (Ez a minimális költségű folyam-feladat speciális esete).

Először keresünk Dijkstra algoritmusával s-ből egy legrövidebb utat minden csúcsba, jelölje P' a t-be vezető utat, és legyen $d(s,v)$ a v-be vezető legrövidebb út hossza. Ezután megcsináljuk az imént látott potenciál-eltolást, azaz $c'(uv):=c(uv)+d(s,u)-d(s,v)$. Ekkor minden él új súlya nemnegatív, és P' éleinek új súlya 0. Fordítsuk meg a P' út éleit, így adódik egy G' gráf a c' élsúlyokkal.

Ebben keressünk Dijkstra algoritmusával s-ből egy legrövidebb Q' utat t-be. Könnyű látni, hogy az optimum értéke $c(P')+c'(Q')+d(s,t)$ lesz. A P és Q utakat úgy kapjuk, hogy vesszük a P' és Q' utak unióját, ebből kitöröljük azon szembemenő élpárokat, melyek egyike P'-ben, másika Q'-ben van, és az eredményt felbontjuk két út uniójára.

Ezt az algoritmust (erre a specializált szituációra) először Suurballe írta le, aki rögtön azt is megvizsgálta, hogy hogyan lehetne a feladatot rögzített s esetén egyszerre minden t célcsúcsra megoldani és adott egy $O(n^2 \log n)$ idejű algoritmust. Ezt Tarjannal együtt ritka gráfokra lényegesen megjavították, ügyes adatstruktúrákkal elérték, hogy a teljes futási idő annyi legyen, mint egy Dijkstra-algoritmus futási ideje a d-edfokú kupaccal, azaz $O(m \cdot \log_{\lceil m/n \rceil} n)$.

Feltehetjük, hogy az s csúcsból minden más csúcs elérhető, hiszen ezt egy szélességi kereséssel könnyen ellenőrizhetjük, és az elérhetetlen csúcsokat nyugodtan kitörölhetjük. Az első fázis legyen ugyanaz, mint az előbb, a legrövidebb utak egy s gyökerű T feszítőfenyőt alkotnak, a potenciál-eltolás után ennek minden éle 0 költségű. Mostantól ezt az eltolt súlyfüggvényt hívjuk c-nek; valamint G_v-nek nevezzük azt a gráfot, amelyben az s-ből v-be vezető T-beli utat megfordítjuk. A fentiek alapján a feladat minden v csúcsra a G_v gráfban keresni egy $s \rightsquigarrow v$ legrövidebb utat. Ezek a gráfok viszont elég hasonlóak, így lehet jobbat csinálni, mint mindegyikben külön-külön futtatni egy Dijkstra-algoritmust.

Az ötlet a következő: a T fenyőben v szülőjét $sz(v)$-vel jelöljük, és T-t irányítatlan, de s gyökerű faként kezeljük. Egy tipikus lépésben már az S halmazbeli csúcsokra „mindent" tudunk, és a $T-S$ erdő komponenseit is ismerjük. Mint a Dijkstra-algoritmusban, itt is lesz minden nem S-beli v csúcsra egy ideiglenes $d(v)$ távolságcímkénk, ami kezdetben $+\infty$ (kivéve s-et, ahol 0).

Kiválasztjuk azt a $v \in V-S$ csúcsot, amelyre $d(v)$ minimális, és ezt berakjuk S-be, valamint az őt tartalmazó $T-S$-beli fát széthasogatjuk a $T_0, T_1, \ldots, T_k$ fákra, ahol T_0 tartalmazza $sz(v)$-t, ha még az nincs S-ben (különben $T_0 = \emptyset$), és a többi T_i fa gyökerei v-nek a nem S-beli gyerekei.

Ezután jön a lényeges rész, a címkék módosítása. Ez a vw éleken a szokásos: ha $d(v)+c(vw) < d(w)$, akkor $d(w)$-t erre módosítjuk. A trükkösebb rész az, hogy minden olyan uw élre, ahol u és w két különböző, most keletkezett fa csúcsai ($u \in T_i$, $w \in T_j$, $i \neq j$), megvizsgáljuk, hogy $\underline{d(v)}+c(uw)$ kisebb-e $d(w)$-nél, és ha igen, akkor $d(w)$-t erre csökkentjük.

Nem túlságosan nehéz belátni, hogy ha megfelelően szinkronizálva egyszerre futtatjuk ezt az algoritmust, illetve egy G_t gráfon a Dijkstra-algoritmust, akkor a végén a t csúcs címkéje ugyanaz lesz, tehát az algoritmus jól működik.

17. fejezet

Párosítások

17.1. A Hopcroft–Karp-algoritmus

Az első részben tárgyalt algoritmus páros gráfban legnagyobb párosítás keresésére $O(nm)$ időben fut, ahol n a kisebbik osztály csúcsainak száma. Lehet-e ezt gyorsabban csinálni?

A válasz igenlő. Jelölje S az alsó osztály, T pedig a felső osztály szabad csúcsainak a halmazát, és irányítsuk meg a pillanatnyi párosítás éleit lefelé, a gráf többi élét felfelé. Az eredeti algoritmus ebben keresett egy darab utat S-ből T-be, és utána ennek mentén rögtön javított.

Először is csináljunk egy szélességi keresést S-ből indítva, és szintezzük be a gráfot: S lesz a 0. szinten, az ebből egy élen elérhető csúcsok az elsőn, s.í.t. Hagyjuk el az irányított gráfunk olyan éleit, melyek nem valamely i. szintről lépnek az $(i{+}1)$.-re. Ebben keressünk $S \rightsquigarrow T$ utat, jegyezzük meg, végpontjait és éleit hagyjuk el. Ezt ismételjük, amíg van ilyen út. A talált utak könnyen láthatóan páronként csúcsdiszjunktak lesznek. Ezután minden megtalált út mentén végezzük el az alternálást. Nem nehéz meggondolni, hogy egy ilyen fázis végrehajtható $O(m)$ lépésben. A javítások után persze újabb szintezett irányított segédgráfot készítünk, és ismételjük a fentieket.

68. Állítás. *Ha egy fázis elején a legrövidebb javító út k hosszú volt, akkor a fázis végrehajtása után a legrövidebb javító út már legalább $k+2$ hosszú lesz.*

69. Tétel (Hopcroft–Karp). *Az algoritmus $2\sqrt{n^*}$ fázis után véget ér, ahol $n^* = \nu(G) \leq \min(|U|, |V|) \leq \frac{n}{2}$.*

Ez abból látszik, hogy $\sqrt{n^*}$ fázis után már minden javító út legalább $2\sqrt{n^*}+1$ hosszú lesz, ekkor azonban az aktuális M párosításra és a legnagyobb M^* párosításra igaz, hogy $|M^*| - |M| \leq \sqrt{n^*}$, mert $M \cup M^*$ gráfnak csak azon komponenseiben lehet kevesebb M-él, amelyek javító utak, és minden ilyen komponensnek legalább $2\sqrt{n^*}+$ $+2$ csúcsa van, melyek M^* által fedettek. Ezek az utak diszjunktak, és az uniójuknak legfeljebb $2n^*$ csúcsa lehet.

17.2. Kuhn magyar módszere és Egerváry tétele

Élsúlyozott páros gráfban szeretnénk maximális súlyú párosítást találni. Először néhány egyszerű redukcióval kezdünk. Egy maximális súlyú párosítás nyilván nem tartalmaz negatív súlyú élet, így ezek az inputból elhagyhatóak. Ezután a gráfot kiegészíthetjük teljes páros gráffá, minden hozzávett élhez a 0 súlyt rendelve (ha a két csúcsosztály különböző elemszámú volt, előbb a kisebbet csúcsok hozzávételével kiegészítjük). Nyilván ettől sem változott a maximális súlyú párosítás súlya, viszont így elég maximális súlyú teljes párosítást találni.

Egy, a csúcsokon értelmezett π nemnegatív függvényt lefogásnak nevezünk, ha minden uv élre $\pi(u)+\pi(v) \geq c(uv)$ teljesül, egy ilyen lefogás értéke az összes csúcsra vett π-értékek összege.

70. Tétel (Egerváry Jenő). *Adott egy $G=(S \cup T, E)$ teljes páros gráf (ahol $|S|=|T|$) és egy $c : E \mapsto \mathbb{R}^+$ nemnegatív súlyfüggvény. A maximális súlyú teljes párosítás súlya egyenlő a minimális értékű lefogás $\sum_{u \in S \cup T} \pi(u)$ értékével. Ha c egészértékű, akkor π is választható egészértékűnek.*

Az algoritmus során érdemes megengedni negatív π értékeket is, és csak a végén változtatni nemnegatívra. Adott π lefogás esetén egy élet szorosnak nevezünk, ha $\pi(u)+$ $+\pi(v)=c(uv)$, és G_π jelölje a szoros élek részgráfját. Kezdetben legyen $\pi(s)=C$ minden $s \in S$-re, ahol C a $c(uv)$ értékek maximuma, és $\pi(t) = 0$ minden $t \in T$-re. Vegyük észre, hogy a 47. algoritmus leállásakor visszaadott O Ore-halmaz az S maximális hiányú halmazainak a metszete; ezt az algoritmust fogjuk szubrutinként hívogatni a G_π segédgráfokra.

57. Algoritmus (Magyar módszer, Khun).

Készítsük el a G_π segédgráfot, és hívjuk meg rá a 47. algoritmust.

Ha ez talál egy M teljes párosítást, STOP; egyébként visszaad egy O Ore-halmazt.

Ez utóbbi esetben határozzuk meg a következő δ mennyiséget: Legyen $T' = \Gamma_\pi(O)$ és $\delta = \min\big(\pi(u)+\pi(v)-c(uv) \ : \ u \in O,\ v \in T-T',\ uv \in E\big)$.

Az O-beli csúcsokon csökkentsük π-t δ-val, a T'-belieken pedig növeljük, majd ugorjunk vissza az algoritmus elejére.

A végén, ha π negatív értékeket is felvesz, jelölje $-\mu$ a legnegatívabb értéket, amely mondjuk a $v \in S$ csúcson vétetik fel. Adjunk minden S-beli csúcs π-értékéhez μ-t, a T-beli csúcsokéból pedig vonjuk ki ugyanezt.

71. Tétel. *Ez az algoritmus maximum $O(|E|^2|S|)$ lépést tesz, és a végén a maximális súlyú M teljes párosítást, valamint az ehhez tartozó ugyanilyen értékű lefogást adja vissza.*

17.3. Edmonds algoritmusának vázlata

Súlyozatlan, összefüggő, nem páros gráfban szeretnénk legnagyobb párosítást találni. Berge lemmája szerint egy M párosítás akkor és csak akkor ilyen, ha nem létezik rá nézve javító út. Tehát kiindulunk az üres párosításból, és ismételten javító utat keresünk.

A javító út kereséséhez egy alternáló erdőt növelünk, a fák gyökerei a fedetlen csúcsok. A fák olyan csúcsait, melyek a gyökértől páros távolságra vannak *külső*nek, a többit *belső*nek hívjuk. Egy külső pontból a saját gyökeréhez vezető út alternáló. Az erdőt magyar erdőnek hívjuk, ha semelyik külső pontból nem megy él se valamely másik külső pontba, se az erdő által nem lefedett pontba.

58. Algoritmus (Edmonds, 1 növelés).

Ha az erdő magyar, akkor STOP; egyébként legyen uv egy olyan él, melyre u külső csúcs, v pedig nem belső.

Ha v egy másik fa külső csúcsa, akkor a gyökerekhez vezető utakkal együtt egy javító utat találtunk, menjünk a „Javítás" részre.

Ha v az erdő által nem lefedett pont, akkor párosított, párja legyen w. Az u-t tartalmazó fához adjuk hozzá az uv és vw éleket, majd kezdjük az elejéről a növelő algoritmust.

Végül ha v az u-t tartalmazó fa külső csúcsa, akkor az uv él együtt az u-ból, ill. v-ből a legközelebbi közös őshöz vezető utakkal egy páratlan kört ad. Ezt a kört húzzuk össze egyetlen csúccsá, majd kezdjük az elejéről a növelő algoritmust.

Javítás:

A javító útból az összehúzott körök fordított sorrendben történő kibontásával könnyen kaphatunk egy javító utat az eredeti gráfban, ennek mentén javítunk.

72. Tétel. *(Edmonds) Ez az algoritmus legfeljebb $O(n^2m)$ lépésben (ügyes adatszerkezetekkel és egyéb ötletekkel $O(\sqrt{n}\cdot m)$ lépésben) megadja a legnagyobb párosítást. A magyar erdő belső pontjai a Gallai–Edmonds-gátat, külső pontjainak ősképei az ehhez tartozó faktor-kritikus komponenseket adják meg.*

18. fejezet

Hálózati folyamok

18.1. Dinic algoritmusa

A Ford–Fulkerson-algoritmussal (48. algoritmus) az a baj, hogy ha rosszul választunk javító utakat, akkor irracionális kapacitások esetén nem is véges, és egész súlyok esetén is nagyon lassú lehet (kivéve, ha a maximális folyam értéke kicsi), legrosszabb esetben akár a legnagyobb kapacitással arányos lépésszám is kellhet. Ezt valamennyire kijavítja az Edmonds–Karp-algoritmus (lásd a 47. tételt), mely erősen polinomiális, de az is igényel legrosszabb esetben $\Omega(nm)$ javítást.

Ezt javítja tovább Dinic algoritmusa, mely a Hopcroft–Karp-algoritmushoz hasonlóan egy maradékhálózatban igyekszik „minél többet” javítani a folyamon, és így eléri, hogy csak n alkalommal kell új maradékhálózatot konstruálni. A *szintezett maradékhálózat*ot úgy kapjuk, hogy először megkonstruáljuk a 48. algoritmusnál leírt maradékhálózatot, majd ebben s-ből indított szélességi kereséssel kiszámítjuk a csúcsok szintjét (s-től való távolságát), és ezután kitörlünk minden élet, amely nem valamelyik szintről a következőre lép, valamint az összes csúcsot, amely távolabb van s-től, mint a t.

Ebben a hálózatban úgynevezett *blokkoló folyam*ot keresünk, ami egy olyan f' folyam, hogy az f' által telített éleket törölve már nem marad út s-ből t-be.

A következő állítások mutatják a módszer erejét:

73. Tétel. *Ha a hálózatban az aktuális f folyamra nézve a legrövidebb javító út k hosszú, és f-et javítjuk a szintezett maradékhálózat egy f' blokkoló folyamával, akkor a javított folyamra nézve a legrövidebb javító út már legalább $k+1$ hosszú lesz.*

74. Tétel. *Egy szintezett maradékhálózatban (amely nyilván aciklikus) egy blokkoló folyamot a mélységi keresés ügyes módosításával megkereshetünk $O(nm)$, ill. ügyes adatszerkezetekkel $O(m \log n)$ lépésben. Tehát Dinic algoritmusa lefut az egyszerű megvalósítással $O(n^2 m)$, ügyesen $O(nm \log n)$ lépésben.*

18.2. Diszjunkt utak

Egy irányított gráfban keresünk s-ből t-be a lehető legtöbb, páronként éldiszjunkt utat. Ha az éldiszjunkt utak maximális száma k, akkor persze ezt Ford és Fulkerson 48. algoritmusával $O(km)$ lépésben megtehetjük. Ezen nagy k értékek esetén javíthatunk:

Dinic módszerét használjuk, mivel a gráfban minden kapacitás 1, ezért ez a maradékhálózatra is teljesül. Ebben blokkoló folyamot könnyen tudunk $O(m)$ időben keresni, hiszen egy javító út minden éle telítődik, így azokat mind kitöröljük.

75. Tétel. *A maximális számú páronként éldiszjunkt utakat kereső Dinic-algoritmus legfeljebb $2\sqrt{m}$ iterációt (új maradékhálózat megkonstruálása és abban javítás) használ, ha ráadásul a gráfban nincsenek párhuzamos élek, akkor legfeljebb $2n^{2/3}$ iterációt.*

Tehát az algoritmus lépésszáma egyszerű gráf esetén $O(\min(m\sqrt{m}, mn^{2/3}))$.

Ha páronként csúcsdiszjunkt utakat keresünk, akkor az még gyorsabb. A szokott módon (mint a 7.5. fejezetben) húzzuk szét a csúcsokat. Ekkor egy olyan hálózatot kapunk, melyben s és t kivételével minden csúcsnak vagy a befoka, vagy a kifoka 1. Ebben két út akkor és csak akkor éldiszjunkt, ha az eredeti gráfban csúcsdiszjunkt.

76. Tétel. *Ha egy 1-kapacitású hálózatban minden csúcs (s és t kivételével) befoka vagy kifoka egy, akkor a Dinic-algoritmus lefut legfeljebb $2\sqrt{n}$ iterációval, tehát $O(m\sqrt{n})$ lépésben.*

Megjegyzés: Vegyük észre, hogy lényegében újra megadtuk a Hopcroft–Karp-algoritmust, ráadásul általánosítva.

18.3. Többtermékes folyamok

A cipőt a cipőgyárban gyártják és a cipőboltban adják el, míg a kenyeret a pékségben készítik és az élelmiszerboltban adják el. Így ha egy úthálózaton akarjuk ezeket szállítani, akkor figyelnünk kell, hogy a két terméket nem keverhetjük össze. Persze sok termékre vonatkozó feladat esetén meg kell határozni, hogy mi a jó célfüggvény. Mivel erre nincs általános recept, ezért a többtermékes folyam-feladatnak az eldöntési verzióját szokás elsősorban nézni. Ez formálisan azt jelenti, hogy k termék esetén adott minden $1 \leq i \leq k$-ra a termék s_i forrása, t_i célja és a szállítandó d_i mennyiség. A feladat az, hogy minden i-re találjunk egy d_i nagyságú f_i folyamot, melynek forrása s_i és nyelője t_i, úgy, hogy együttesen a kapacitáskorlát alatt maradjanak, azaz minden e élre $\sum f_i(e) \leq c(e)$, ahol c a G irányított alapgráf élein adott nemnegatív kapacitásfüggvény.

Sokszor kényelmesebb az igényeket egy H igénygráffal megadni, melyben minden i-re s_i-ből t_i-be menő élre írjuk rá a d_i igényt (a H gráfnak ugyanaz a V a csúcshalmaza, mint a G gráfnak.).

Ha tört-folyamot keresünk, akkor ez egy LP-feladat, így megoldható polinomiális időben, sőt Tardos Éva kifejlesztett egy erősen polinomiális algoritmust is rá.

A többtermékes tört-folyam feladatra a Farkas lemmából könnyen bizonyítható, hogy akkor és csak akkor van megoldás, ha minden $\ell : E \mapsto \mathbb{R}^+$ hosszúságfüggvényre

$$\sum_{i=1}^{k} d_i \cdot \mathrm{dist}_\ell(s_i, t_i) \leq \sum_{e \in E} \ell(e) \cdot c(e),$$

ahol $\mathrm{dist}_\ell(s_i, t_i)$ az ℓ hosszúság szerint legrövidebb $s_i \rightsquigarrow t_i$ út hosszát jelöli. Ezt úgy lehet értelmezni, hogy az élek helyébe c keresztmetszetű ℓ hosszú csöveket képzelünk,

a bal oldal egy adott i-re alsó becslés az i. termék által elfoglalt térfogatra, míg a jobb oldal a rendszer teljes térfogata.

Innentől feltesszük, hogy a kapacitások és az igények egészek. Egész többtermékes folyam keresése NP-nehéz, még két termék esetén is, sőt akkor is, ha minden kapacitás 1, azaz páronként éldiszjunkt $s_1 \rightsquigarrow t_1$ és $s_2 \rightsquigarrow t_2$ utakat keresünk (ezek a feladatok irányítatlan gráfokra is NP-nehezek). Irányított esetben még akkor is NP-nehéz marad, ha továbbá még azt is feltesszük, hogy $d_1 = d_2 = 1$ (ez a feladat azonban már irányítatlan gráfokra polinomiális időben megoldható).

Azonban **irányítatlan** gráfra, két termékre és tetszőleges igényekre mégis lehet valami érdekeset mondani. **Vágásfeltételnek** hívjuk a következőt:

Minden $\emptyset \neq X \subset V$ csúcshalmazra $d_H(X) \leq d_G(X)$, ahol $d_H(X)$ az X-ből kilépő H-beli élek igény-értekeinek összege, $d_G(X)$ pedig az X-ből kilépő G-beli élek kapacitás-értekeinek összege.

Ha a vágásfeltétel nem teljesül, akkor nyilván nincs törtértékű folyam sem. Másrészt könnyen látható, hogy a vágásfeltétel nem elégséges egész folyam létezésére: legyen G egy C_4, H a két átlója, és minden kapacitás és igény 1.

Azt mondjuk, hogy a feladat teljesíti az Euler-feltételt, ha minden csúcsra a kimenő G-élek kapacitásainak összege plusz a kimenő H-élek igényeinek összege páros.

77. Tétel (Hu). *Egy irányítatlan G gráfban a kéttermékes folyamfeladatnak akkor és csak akkor van tört megoldása, ha teljesül a vágásfeltétel. Ráadásul, ha van tört megoldása, akkor van félegész megoldása is.*

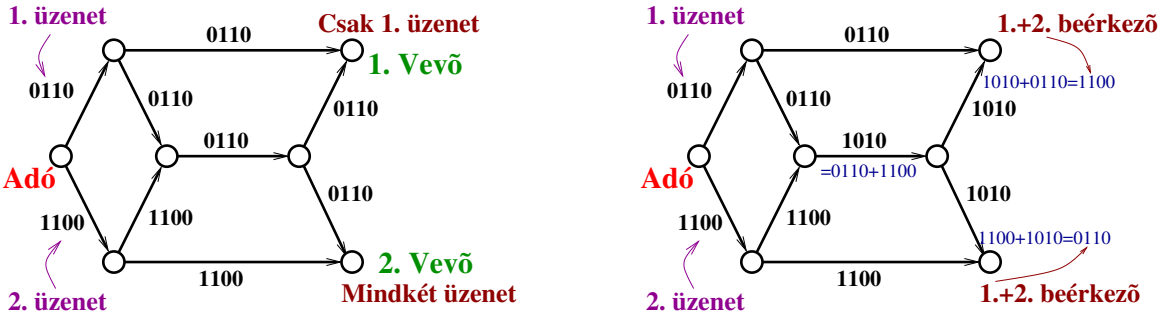
Ez könnyen következik az alábbi tételből:

78. Tétel (Rothschild–Whinston). *Tegyük fel, hogy egy irányítatlan G gráfban a kéttermékes folyamfeladat teljesíti az Euler-feltételt. Ekkor, ha teljesül a vágásfeltétel, akkor a többtermékes folyamfeladatnak van egész megoldása is.*

19. fejezet

Hálózati kódolás

Telekommunikációs hálózatokban a különböző adatfolyamokat sokáig különböző termékekként tekintették. Arra elég korán rájöttek, hogy könnyen megoldható az, hogy ha egy router megkap egy üzenetet, akkor egyszerre több kimenő ágon is „ingyen” továbbíthatja. Ez a két dolog együtt egy nehezen kezelhető modellt adott. 2000-ben jelent meg egy áttörő cikk, ahol felfedezték, hogy a cipő-kenyér párral ellentétben két különböző adatfolyamnak pl. a mod 2 összege is könnyen továbbítható egy kimenő ágon.



19.1. ábra. A pillangó-példa

A következő egyszerűsített modellt vizsgáljuk: az alaphálózat egy $G=(V,E)$ aciklikus irányított gráf (DAG), mely minden élének kapacitása 1 (azaz minden élen időegységenként 1 csomag tud áthaladni, párhuzamos élek megengedettek). Adott még egy $s \in V$ forrás és $T=\{t_1,t_2,...,t_{|T|}\} \subset V$ terminálok (ezek szeretnék megkapni az s által „sugárzott” adatfolyamot. Az adatfolyam csomagokra van osztva, minden csomag ℓ bitből áll, egy csomagra úgy tekintünk, mint az $\mathbb{F}$ $q=2^\ell$-elemű véges test egy elemére. A cél minden időegységben minél több csomag eljuttatása s-ből az összes t_i-be.

Jelölje $\lambda(s,t_i)$ az s -ből t_i-be vezető páronként éldiszjunkt utak maximális számát, és legyen $k=\min_i \lambda(s,t_i)$. Nyilván ennél több csomagot nem tudunk eljuttatni.

Egy hálózati kód lineáris, ha megadható az időben nem változó $b_{uvw} \in \mathbb{F}$ számokkal úgy, hogy ha egy uv élen az $M(uv)$ üzenet megy, akkor $M(vw)=\sum_u b_{uvw}M(uv)$. Egy adott időegységben cél az $\mathbf{M}=(M_1,\ldots,M_k)$ üzenet-k-as eljuttatása.

A meglepő eredmény a következő:

79. Tétel (Lineáris hálózati kódok alaptétele). *Mindig lehetséges k csomag szimultán eljuttatása az összes terminálba. Ráadásul ez megtehető lineáris kóddal, sőt, a testméretet is tetszőlegesen választhatjuk a $q \geq |T|$ feltétel mellett.*

További redukciók, előmunkálatok:

- feltehető, hogy minden csúcs befoka legfeljebb k.
- adott a csúcsoknak és az éleknek egy topologikus sorrendje, melyben uv biztosan megelőzi $u'v'$-t, ha $v = u'$ vagy v megelőzi u'-t.
- s kifoka k (ekkor a k darab s-ből kimenő él az első k él a topologikus sorrendben, azaz $e_1, \ldots, e_k$).
- minden $t_j \in T$ terminálra előre kiszámoltunk és lefixáltunk k db páronként éldiszjunkt $P_1^j, P_2^j, \ldots, P_k^j$ utat, melyek s-ból t_j-be vezetnek. Feltehető, hogy a P_i^j út első éle e_i.

Ugyan az elvárt output a $\{b_{uvw}\}$ számhalmaz, jobb az úgynevezett „szállított" vektorokkal dolgozni, melyeket így definiálunk:

Legyen $\mathbf{e}_i = (0,\ldots,0,1,0,\ldots,0)$ az i-edik egységvektor az $\mathbb{F}^k$ vektortérben. Egy lineáris hálózati kód azonosítható egy $\mathbf{c} : E \to \mathbb{F}^k$ hozzárendeléssel, melyre $i \le k$ esetén $\mathbf{c}(e_i) = \mathbf{e}_i$, és minden további vw élre $\mathbf{c}(vw) = \sum_u b_{uvw}\mathbf{c}(uv)$. Ekkor nyilván egy vw élen a $\mathbf{c}(vw) \cdot (M_1, \ldots, M_k)$ üzenet megy.

Legyen e_t a sorrend t-edik éle, és $t > k$, $1 \le j \le |T|$ esetén $F_{e_t}^j$ az a mátrix, melynek i-edik sora $\mathbf{c}(e(i,j,t))$, ahol $e(i,j,t)$ a P_i^j út utolsó olyan éle, amely a top. sorrendben nincs később, mint az e_t él. A $t \le k$ esetre legyen $F_{e_t}^j = I$.

4. Lemma. *Elérhető, hogy minden t-re és minden j-re az $F_{e_t}^j$ mátrix reguláris (invertálható).*

A lemma bizonyításával készen leszünk a tétel bizonyításával is, hiszen minden t_j terminálra, ha a bejövő (pontosan k él) az $e'_1, \ldots, e'_k$ (ahol $e'_i \in P_i^j$ és ezek közül topologikus sorrendben e'_k a legkésőbbi); és az adott időegységben az e'_i élen az $M(e'_i)$ üzenet érkezik, akkor $(M_1, \ldots, M_k) = (F_{e'_k}^j)^{-1} \cdot (M(e'_1), \ldots, M(e'_k))$. Algoritmikus bizonyítást adunk, mert a hálózati kódot meg is szeretnénk konstruálni. Ehhez még minden e élhez nyilvántartunk $\mathbf{m}(i,j,e)$ vektorokat is, melyek merőlegesek az F_e^j mátrix minden sorára, kivéve az i sort. Kezdetben ($t \le k$ esetén) legyen $\mathbf{m}(i,j,e_t) = \mathbf{e}_i$.

Sorba megyünk a topologikus sorrend szerint, tegyük fel, hogy $e_1, \ldots, e_t$ élekre már mindent kiszámoltunk. Most az e_{t+1} él következik. Ha valamelyik j-re ezen az élen nem megy át a P_i^j utak egyike sem, akkor $F_{e_{t+1}}^j = F_{e_t}^j$ jó lesz. Jelölje J azon j értékek halmazát, melyekre van olyan i_j, hogy a $P_{i_j}^j$ út használja az e_{t+1} élet, legyen ezen út előző éle f_j.

Ha csak egy $j \in J$-re kellene meghatározni az $F_{e_{t+1}}^j$ mátrixot, akkor ez könnyű: az i_j-edik sort kell lecserélni egy olyan vektorra, hogy a mátrix reguláris maradjon. Azonban mi minden rögzített $j \in J$-re akarjuk ezeket meghatározni úgy, hogy $\mathbf{c}(e_{t+1})$ a bejövő éleken már adott $\mathbf{c}$ vektorok ugyanazon lineáris kombinációja legyen.

Ehhez segít az alábbi egyszerű lineáris algebrai lemma.

5. Lemma. *Adott $n \le |\mathbb{F}|$ és $\mathbf{x}_i, \mathbf{y}_i \in \mathbb{F}^k$ úgy, hogy $\mathbf{x}_i \cdot \mathbf{y}_i \neq \mathbf{0}$ minden $1 \le i \le n$-re. Ekkor létezik $\mathbf{u} = \sum_{i \le n} \alpha_i \mathbf{x}_i$, melyre $\mathbf{u} \cdot \mathbf{y}_i \neq \mathbf{0}$ minden $1 \le i \le n$-re. Az α_i számok könnyen kiszámolhatóak.*

Ennek bizonyításához sorban konstruálunk olyan $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n$ vektorokat, hogy minden j-re az $\mathbf{u}_j$ az $\mathbf{x}_i$ vektorok (i=1, …, j) lineáris kombinációja, és $\mathbf{u}_j$ nem merőleges az $\mathbf{y}_1, \ldots, \mathbf{y}_j$ vektorok egyikére sem.

Legyen $\mathbf{u}_1 = \mathbf{x}_1$. Tegyük fel, hogy $\mathbf{u}_1, \ldots, \mathbf{u}_i$ már megvan. Ha $\mathbf{u}_i \cdot \mathbf{y}_{i+1} \neq 0$, akkor legyen $\mathbf{u}_{i+1} = \mathbf{u}_i$, különben minden $\alpha \in \mathbb{F}$-re $(\alpha\mathbf{u}_i + \mathbf{x}_{i+1}) \cdot \mathbf{y}_{i+1} \neq 0$. Egy α-t hívjunk rossznak, ha van olyan $j \leq i$, hogy $(\alpha\mathbf{u}_i + \mathbf{x}_{i+1}) \cdot \mathbf{y}_j = 0$, azaz $\alpha = -\frac{\mathbf{x}_{i+1} \cdot \mathbf{y}_j}{\mathbf{u}_i \cdot \mathbf{y}_j}$ (az indukciós feltevés miatt $\mathbf{u}_i \cdot \mathbf{y}_j \neq 0$, ha $j \leq i$). Mivel $i < |\mathbb{F}|$, és másrészt legfeljebb i db rossz α van, ezért van jó is, legyen ez α^*. Tehát ekkor $\mathbf{u}_{i+1} = \alpha^* \mathbf{u}_i + \mathbf{x}_{i+1}$ jó lesz.

Visszatérve a 4. Lemma bizonyításához, legyenek $\mathbf{x}_i$ vektorok a bejövő f_i élekre már kiszámolt $\mathbf{c}(f_i)$ vektorok és $\mathbf{y}_i = \mathbf{m}(i_j, j, f_i)$. Ekkor a lemma által adott $\mathbf{u}$ vektor jó lesz nekünk, ez lesz $\mathbf{c}(e_{t+1})$. Az új $F^j_{e_{t+1}}$ mátrixok azért lesznek regulárisak, mert az $F^j_{e_t}$ mátrix i_j-edik sorát lecseréltük egy olyan vektorra, mely nem merőleges a többi sor által feszített hipersík $\mathbf{m}(i_j, j, e_t)$ normálvektorára.

Az új $\mathbf{m}(i, j, e_{t+1})$ vektorok újra számítása sem nehéz: fixáljunk le egy $j \in J$ értéket, az $F^j_{e_{t+1}}$ mátrixnak csak az i_j-edik sora változott, tehát $\mathbf{m}(i_j, j, e_{t+1}) = \mathbf{m}(i_j, j, e_t)$ jó marad, de a többi $\mathbf{m}(i, j, e_t)$ vektor nem feltétlenül merőleges az új $\mathbf{c}(e_{t+1})$ vektorra. Fixáljunk egy $i \neq i_j$ indexet, legyen A az az altér, melyet a többi $k-2$ sor feszít. Mind $\mathbf{m}(i_j, j, e_t)$, mind $\mathbf{m}(i, j, e_t)$ merőleges A-ra, így minden lineáris kombinációjuk is. Keresünk egy olyat, amely merőleges $\mathbf{c}(e_{t+1})$-re is. Mivel $\mathbf{m}(i_j, j, e_t)$ nem lehet merőleges $\mathbf{c}(e_{t+1})$-re, ezért $\beta = -\frac{\mathbf{m}(i,j,e_t) \cdot \mathbf{c}(e_{t+1})}{\mathbf{m}(i_j,j,e_t) \cdot \mathbf{c}(e_{t+1})}$ egy testelem, és $\mathbf{m}(i, j, e_{t+1}) = \beta\mathbf{m}(i_j, j, e_t) + \mathbf{m}(i, j, e_t)$ jó lesz.

20. fejezet

Közelítő algoritmusok

20.1. Definíciók

A következő típusú feladatokat tekintjük. Egy $\mathbf{x}$ input esetén ehhez hozzá van rendelve a megengedett megoldások egy $X(\mathbf{x})$ halmaza (például ha az input egy súlyozott gráf, a megengedett megoldások halmaza állhat a gráf Hamilton-köreiből). Adott továbbá egy $f_{\mathbf{x}} : X(\mathbf{x}) \mapsto \mathbb{R}$ kiértékelő függvény (a példában egy Hamilton-körhöz az f függvény az összsúlyát rendelheti). A feladat az, hogy keressünk olyan $\mathbf{y} \in X(\mathbf{x})$ megengedett megoldást, amelyre $f_{\mathbf{x}}(\mathbf{y})$ minimális. Ebben a fejezetben csak olyan feladatokkal foglalkozunk, amelyek NP-nehezek.

Egy A algoritmust α-közelítőnek hívunk, ha minden inputra, amelyre létezik megengedett megoldás, kiad egy megengedett $\mathbf{y}$ megoldást, melynek $f_{\mathbf{x}}(\mathbf{y})$ értéke legfeljebb α-szorosa az optimumnak, melyet a továbbiakban OPT jelöl (itt α nem feltétlenül konstans, lehet az input n hosszának egy függvénye is).

Néha persze nem minimumot, hanem maximumot keresünk, ekkor egy α-közelítő algoritmus olyan megengedett megoldást ad ki, melynek értéke legalább $\frac{1}{\alpha}$-szorosa az optimumnak.

Egy algoritmus ε relatív hibájú, ha $(1+\varepsilon)$-közelítő (illetve maximalizálási feladatnál $\frac{1}{1-\varepsilon} \approx (1+\varepsilon)$-közelítő, azaz a megengedett megoldásunk értéke legalább $(1-\varepsilon)\cdot \mathrm{OPT}$).

Egy adott algoritmusról nem mindig könnyű belátni, hogy α-közelítő, mert ehhez meg kell becsülni az optimum értékét. A legegyszerűbb példa a következő: adott $G = (V, E)$ irányított gráf, keressük az éleinek a legnagyobb olyan részhalmazát, amelyek még aciklikus gráfot alkotnak. A következő egyszerű algoritmus erre 2-közelítést ad: vegyük a csúcsok tetszőleges sorrendjét, és nézzük meg, hogy az előremenő, vagy a visszamenő élek vannak-e többségben, és válasszuk a többséget. Ezek nyilván aciklikus részgráfot alkotnak, és persze legalább $|E|/2$ élből állnak. Az optimum nem lehet nagyobb $|E|$-nél, tehát ez az algoritmus 2-közelítő. Óvatosságra int azonban a komplementer feladat, ahol a legkevesebb olyan élet keressük, amiket elhagyva a gráf aciklikussá válik. Erre nem ismert α-közelítő polinomiális algoritmus semmilyen α konstansra, sőt, a legjobb, amit tudunk, egy $O(\log n \cdot \log\log n)$-közelítés.

Ha minden $\varepsilon > 0$ konstanshoz létezik egy polinomiális idejű A_ε algoritmus, amely az adott feladatot ε relatív hibával oldja meg, akkor ezt a családot *polinom-idejű approximációs séma*nak (PTAS) nevezzük. Ha van egy olyan A algoritmusunk, amely minden ε-ra (amit szintén megkap bemenetként) ε relatív hibájú, és ráadásul lépésszáma poli-

nomja az eredeti bemenet n hosszának **és** $(1/\varepsilon)$-nak is, akkor ezt *teljesen* polinom-idejű approximációs sémának (FPTAS) nevezzük. A kettő között helyezkednek el az ún. *hatékony* polinom-idejű approximációs sémák (EPTAS), amelyek lépésszáma $O(f(\frac{1}{\varepsilon})\cdot n^c)$ valamilyen f függvényre és c konstansra, (tehát itt $1/\varepsilon$ pl. már nem szerepelhet n kitevőjében).

Példák: egy PTAS lépésszáma lehet akár $n^{2^{1/\varepsilon}}$ is, egy EPTAS algoritmusé lehet például $2^{1/\varepsilon}\cdot n^3$, míg egy FPTAS algoritmus lépésszáma például $(1/\varepsilon^2)\cdot n^4$.

APX jelöli a problémák azon osztályát, amelyekre létezik konstans α és egy α-közelítő polinom-idejű algoritmus (igazából még fel szokás tenni, hogy a megfelelő eldöntési probléma NP-ben van). Egy problémát APX-nehéznek hívunk, ha van olyan $\alpha_0 > 1$ konstans, hogy ha létezne valamilyen $\alpha < \alpha_0$-ra polinom-idejű α-közelítő algoritmus, akkor ebből már $P = NP$ következne. Tehát APX-nehéz problémákra nem várható PTAS. Szokás szerint APX-teljesnek nevezünk egy problémát, ha APX-ben van és APX-nehéz.

Például a minimális lefogó csúcshalmaz feladat (olyan legkisebb csúcshalmazt keresünk egy adott gráfhoz, amely minden élnek legalább az egyik végpontját tartalmazza) APX-teljes ($\alpha_0 = 1{,}36$ értékkel). Könnyű adni egy 2-közelítő algoritmust: vegyünk mohón egy tartalmazásra nézve maximális M párosítást, és válasszuk be az összes M-beli él mindkét végpontját. Azonban már 1,99-közelítő polinom-idejű algoritmus nem ismert. Itt is nagyon eltérő a komplementer feladat, ahol legnagyobb független csúcshalmazt keresünk. Ez az egyik legközelíthetetlenebb probléma: ha lenne polinom-idejű $n^{1-\varepsilon}$-közelítés, akkor ebből $P = NP$ következne.

20.2. Metrikus utazó ügynök

A (metrikus) utazó ügynök feladat a következő: adott egy $G=(V,E)$ irányítatlan összefüggő gráf és az élein pozitív $c(uv)$ költségek. Túrának nevezünk egy olyan körsétát, amely a gráf minden csúcsán áthalad. Keressük a legolcsóbb (minimális összköltségű) túrát.

Megjegyzés: Eredetileg az utazó ügynök problémát úgy definiálták, hogy minimális költségű Hamilton-kört keresünk. A „metrikus" jelző arra utal, hogy speciális esetként kezelték azt, amikor a G teljes gráf, és a c költségfüggvény kielégíti a háromszög-egyenlőtlenséget: $c(uw) \leq c(uv)+c(vw)$ minden u, v, w csúcs-hármasra. Könnyű meggondolni, hogy a fent definiált feladat ezzel ekvivalens: először képezzük a G gráf G^* metrikus lezártját, ami egy teljes gráf a V csúcshalmazon, ahol a $c(u,v)$ költségek a G gráfban az u-ból v-be vezető legrövidebb út hosszát jelölik minden $u, v \in V$ csúcspárra. Ha ebben találunk K összköltségű Hamilton-kört, akkor abból könnyű csinálni a G gráfban egy legfeljebb K összköltségű túrát.

1. közelítő algoritmus: Prim algoritmusával számoljunk ki egy T minimális költségű feszítőfát. Ebből úgy kapunk egy $2\cdot c(T)$ költségű túrát (minden csúcsot érintő körsétát), hogy a fát elképzeljük lerajzolva, és körbejárjuk, azaz végülis minden élén kétszer fogunk átmenni.

Mivel az optimális túra tartalmaz egy feszítőfát, így nyilván $c(T) \leq \mathrm{OPT}$. Ez tehát egy $O(n^2)$ idejű 2-közelítő algoritmus. A kettes faktoron jelentősen javíthatunk az alábbi híres algoritmussal.

59. Algoritmus (Christofides $\frac{3}{2}$-közelítő algoritmusa).

Prim algoritmusával számoljunk ki egy T minimális költségű feszítőfát. Jelölje X a T páratlan fokú csúcsainak halmazát. Keressünk $G^*[X]$-ben (a metrikus lezárt X által feszített részgráfjában) M minimális súlyú teljes párosítást. (Erre létezik hatékony, azaz polinom idejű algoritmus; ebben a jegyzetben nem részletezzük.) Ekkor a $T+M$ gráfban már minden fokszám páros, tehát létezik benne Euler-séta, mely minden csúcsot érint.

Itt a $T+M$ gráfot úgy értjük, hogy ha $uv \in T \cap M$, akkor u és v között két párhuzamos élet húzunk be. Nem nehéz belátni, hogy $c(M) \leq \frac{\mathrm{OPT}}{2}$, ezért $c(T)+c(M) \leq \frac{3}{2} \cdot \mathrm{OPT}$.

20.3. Steiner-fa

A Steiner-fa feladat (gráfos verziójában) adott egy összefüggő $G=(V,E)$ gráf c pozitív élköltségekkel és terminálok egy $S \subseteq V$ halmaza. A feladat olyan minimális költségű összefüggő részgráf keresése, amely S minden csúcsát tartalmazza. Jelölje OPT ennek költségét.

Csinálunk egy $G'=(S,E')$ segédgráfot c' költségekkel, mely egy teljes gráf az S csúcshalmazon, és $c'(s_1s_2)$ a c szerinti legrövidebb $s_1 \rightsquigarrow s_2$ út hossza a G-ben. Legyen T' egy minimális c'-költségű feszítőfa G'-ben. A T' minden élének egy legrövidebb út felel meg G-ben. Vegyük ezen utak unióját, majd amíg ez nem körmentes, addig vegyünk egy tetszőleges kört, és töröljünk belőle ki egy tetszőleges élet; végül így kapunk egy T fát G-ben, amely továbbra is tartalmazza S összes csúcsát. Ez lesz a közelítő megoldásunk.

80. Állítás. $c(T) \leq 2 \cdot \mathrm{OPT}$.

Legyen T^* egy optimális megoldás (ez nyilván egy fa), és jelölje $2T^*$ azt a multigráfot, melyben T^* minden éléhez felveszünk egy párhuzamos élet. Vegyünk ebben egy Euler körsétát. Egy S-beli s_1 csúcsból indítva menjünk ezen körbe, legyenek az érintett S-beli csúcsok $s_1, s_2, \ldots, s_k = s_1$ (persze egy S-beli csúcs többször is szerepelhet). A körséta s_i és s_{i+1} közötti része egy Q_i séta. Legyen P_i a legrövidebb út s_i-ből s_{i+1}-be.

Ekkor $\sum c(P_i) \leq \sum c(Q_i) = 2 \cdot \mathrm{OPT}$. Mivel a G' gráfban az s_is_{i+1} élek egy összefüggő feszítő részgráfot alkotnak, ezért $\sum c'(s_is_{i+1}) = \sum c(P_i) \geq c'(T')$, tehát $c(T) \leq c'(T') \leq \leq \sum c(P_i) \leq 2 \cdot \mathrm{OPT}$.

20.4. FPTAS a hátizsák-feladatra

Visszatérünk az NP-nehéz hátizsák-feladatra. Először is megjegyezzük, hogy kis értékek esetén egy másik dinamikus programozási algoritmus működik, ami most használhatóbb lesz a számunkra. Tegyük fel, hogy E egy felső becslés a kivihető legnagyobb értékre, erre a célra $E = \sum e_i$ is jó lehet, de igazából ennél jobbat is tudunk, a tört optimum értékét, amit úgy kapunk meg, hogy a tárgyakat az e_i/w_i fajlagos érték szerint csökkenő sorrendbe rendezzük, veszünk az elejéről annyi tárgyat, amennyi belefér a hátizsákba, majd a következő tárgyból levágunk egy akkora részt, hogy éppen kitöltse a zsákot. Most a kiviendő tárgyak indexeinek I halmazát is kiszámoljuk.

60. Algoritmus (Hátizsák – kis E értékre).

for $j = 1..E$ $T(0,j) := W+1$ /* *$T(i,j)$ lesz a legkisebb teherbírású hátizsák, amiben j érték kivihető az első i tárgyból; $W+1$ a végtelen megfelelője.*

$T(0,0) := 0$
for $i = 1..n$
  for $j = 0..E$ $T(i,j) := T(i-1,j)$
  for $j = e_i..E$
    if $T(i-1,j-e_i)+w_i < T(i,j)$ **then** $T(i,j) := T(i-1,j-e_i)+w_i$
for $j = E..0$ (-1) **if** $T(n,j) \le W$ **then** $e := j$; **print** e; $j := 0$
$I := \emptyset$
for $i = n..1$ (-1)
  if $T(i,e) < T(i-1,e)$ **then** $I := I \cup \{i\}$; $e := e - e_i$

Ennek az $O(E{\cdot}n)$ idejű algoritmusnak a segítségével megadjuk az Ibarra és Kim által kifejlesztett FPTAS algoritmust. Először is feltehetjük, hogy minden tárgy magában belefér a hátizsákba, azaz minden i-re $w_i \le W$ (a többi tárgy törölhető).

Legyen $E = \sum e_i$ és M egy később meghatározandó szám. Definiálunk egy segédfeladatot, amelyben w_i és W ugyanazok, de az értékeket lecsökkentjük: $e'_i = \lfloor \frac{e_i}{M} \rfloor$. Jelölje I az eredeti feladat optimális tárgyhalmazát, I' a segédfeladatét, valamint OPT az eredeti feladatban optimálisan kivehető értéket.

A mi algoritmusunk egyszerűen a következő: megoldjuk a segédfeladatot az iménti algoritmussal, és az így kapott I' tárgyhalmazt adjuk ki megoldásként. A mi megoldásunk értéke tehát $\mathrm{MO} := \sum_{i \in I'} e_i$ lesz.

Hátra van még az M jó megválasztása. Ha túl kicsinek választjuk, akkor a dinamikus programozási algoritmus túl lassú lesz, ha túl nagynak, akkor a tárgyak eredeti értékei „eltűnnek”, tehát várhatóan a megoldásunk távol lesz az optimumtól.

$$\mathrm{MO} = \sum_{i \in I'} e_i \ge M \cdot \sum_{i \in I'} e'_i \ge M \cdot \sum_{i \in I} e'_i \ge \sum_{i \in I} (e_i - M) \ge \mathrm{OPT} - nM.$$

A második egyenlőtlenség azért teljesül, mert I' volt az optimálisan kivihető tárgyhalmaz a segédfeladatra. Az a célunk, hogy $\mathrm{MO} \ge (1-\varepsilon) \cdot \mathrm{OPT}$ teljesüljön, ehhez az kell, hogy $nM \le \varepsilon \cdot \mathrm{OPT}$ teljesüljön. Ehhez megbecsüljük OPT értékét: mivel feltettük, hogy minden tárgy magában belefér a hátizsákba, ezért $\mathrm{OPT} \ge \max_i e_i \ge E/n$. Tehát válasszuk M értékét $M = \dfrac{\varepsilon \cdot E}{n^2}$-nek, ekkor az iménti elvárásunk teljesül. A dinamikus programozási algoritmus lépésszáma pedig legfeljebb $O\left(\dfrac{E \cdot n}{M}\right) = O\left(\dfrac{n^3}{\varepsilon}\right)$.

20.5. Maximális stabil házasítás

A stabil párosítás feladatnak tekintjük azt az általánosítását, amikor a lányoknál a preferencia-sorrend nem szigorú, tartalmazhat döntetleneket is. A Gale–Shapley-algoritmus (46. algoritmus) ekkor is talál stabil párosítást, azonban ilyenkor a különböző stabil párosítások lehetnek különböző számosságúak, és a cél egy lehető legnagyobb méretű stabil párosítás megtalálása. Ez egy APX-nehéz probléma, ha lenne rá polinomiális 1,1-közelítő algoritmus, akkor ebből P=NP következne. Itt egy 3/2-közelítő algoritmust adunk.

Az algoritmus nagyon hasonló a Gale–Shapley-algoritmushoz, azonban néhány fiúnak piros pontot fogunk adni, és ez úgy befolyásolja a sorrendet, hogy ha egy l lány eredetileg egyformán kedveli az f_1 és f_2 fiúkat, és f_1-nek van piros pontja, míg f_2-nek nincs, akkor l **jobban kedveli** f_1-et, mint f_2-t.

Először is futtatjuk a Gale–Shapley-algoritmust, ha egy lány az aktuális kérőt ugyanannyira kedveli, mint a pillanatnyi partnerét, akkor az új kérőt kosarazza ki (például; azonban ez nem lényeges). Ha egy fiút már az összes lány kikosarazott, akkor megkapja a piros pontot. Ezután az eredeti listája szerint még egyszer sorban elkezdi megkérni a lányok kezét. Ha a piros pontjával is minden lány kikosarazza, akkor leáll. Ha minden fiú vagy leállt, vagy van partnere, akkor az utóbbiak elveszik feleségül a partnerüket.

A kérések száma legfeljebb kétszerese az élek számának, így az algoritmus lineáris idejű.

81. Tétel. *Ez az algoritmus mindig egy* 3/2*-közelítő stabil párosítást ad.*

Ez azon múlik, hogy ha vesszük az algoritmus által adott M stabil párosítás és az optimális M^* stabil párosítás szimmetrikus differenciáját, akkor komponensenként igaz, hogy az M^*-élek száma legfeljebb másfélszerese az M-élek számának, kivéve a „rövid javító utakat", azaz az olyan 3 élű komponenseket, melyeknek csak a közepe van M-ben, tehát a két végpontjuk az algoritmus végén szingli maradt. Ilyen azonban nem lehet.

Tegyük fel indirekte, hogy $f_1 l_1 f_2 l_2$ egy ilyen komponens. Mivel az algoritmus végén az f_1 fiú szingli, ezért ő már a piros pontjával együtt is végigkérte az összes számára elfogadható lányt. Mivel az l_2 lány szingli, őt soha senki nem kérte meg, ezért f_2-nek nem lehet piros pontja. Ezért l_1 csak úgy választhatta az f_2 fiút f_1 helyett, hogy szigorúan jobban kedvelte. És persze f_2 azért nem kérte meg soha l_2 kezét, mert l_1-et jobban kedveli. Ekkor azonban az $l_1 f_2$ él blokkolja az M^* stabil párosítást, ellentmondás.

Megjegyzés: Egy hasonló algoritmus akkor is 3/2-közelítő stabil párosítást ad, ha mindkét oldalon lehetnek döntetlenek.

Fontos alkalmazása a fenti algoritmusnak pl. az egyetemi felvételi rendszer, ahol a lányok szerepét az egyetemi szakok játsszák (mindegyikhez tartozik egy kapacitás is), a fiúk szerepét pedig a felvételizők. A felvételizőktől szigorú sorrendet várunk, míg egy szak pontrendszerében több diák érhet el ugyanannyi pontot. Itt persze nem párosítást várunk, hanem olyan részgráfot, ahol a fokok a diákoknál legfeljebb egyek, a szakoknál meg nem haladják meg a kapacitást. Azonban ezt könnyű visszavezetni a párosítás esetre, egy u kapacitású szak helyett felveszünk u darab csúcsot, melyek közül az i-edik az adott szak i-edik legjobb helyét reprezentálja.

20.6. Halmazfedés

Adott az $S=\{1,2,\ldots,n\}$ alaphalmaznak m db részhalmaza: $A_1, A_2, \ldots, A_m$, és minden részhalmazhoz tartozik egy $c(A_i)$ pozitív költség. Keresünk egy minimális összköltségű fedést, azaz olyan részhalmazokat, melyek uniója egyenlő az S alaphalmazzal. Két különböző közelítő algoritmust is megadunk.

61. Algoritmus (Mohó közelítés halmazfedésre).

$C := \emptyset$
while $C \neq S$
$j := \operatorname{argmin}_i\left(\frac{c(A_i)}{|A_i - C|}\right)$
for $s \in A_j - C$ $\quad \operatorname{price}(s) := \left(\frac{c(A_j)}{|A_j - C|}\right)$ $\quad$ /* *ez csak az elemzéshez kell.*
$C := C \cup A_j$

82. Állítás. *A k-adiknak lefedett s_k elemre* $\operatorname{price}(s_k) \leq \frac{\mathrm{OPT}}{n-k+1}$, *ezért a mohó algoritmus (melynek összköltsége* $\sum_{s\in S} \operatorname{price}(s)$*) H_n-közelítő, ahol* $H_n = \sum_{i=1}^{n} \frac{1}{i} \approx \ln n$.

Ehhez elég belátni, hogy mindig létezik az OPTértékű megoldásnak olyan A_ℓ halmaza, amelyre

$$\frac{c(A_\ell)}{|A_\ell - C|} \leq \frac{\mathrm{OPT}}{n - |C|}.$$

Hiszen ekkor $|C| \leq k-1$ esetén

$$\frac{c(A_j)}{|A_j - C|} \leq \frac{c(A_\ell)}{|A_\ell - C|} \leq \frac{\mathrm{OPT}}{n-k+1}.$$

Ez pedig azért világos, mert az optimális megoldás fedi az $S - C$ halmazt is, ezért $\sum |A_\ell - C| \geq n - |C|$. Tehát ha az optimális megoldás minden A_ℓ halmazára $c(A_\ell) > > \mathrm{OPT} \cdot |A_\ell - C|/(n - |C|)$ teljesülne, akkor ezeket összeadva $\mathrm{OPT} > \mathrm{OPT}$ adódna, ami nyilván nem lehetséges.

Most tegyük fel, hogy a halmazrendszerben a maximális fok Δ. Oldjuk meg tetszőleges polinomiális módszerrel a következő LP-feladatot, melyet az eredeti feladat tört relaxáltjának nevezünk:

$$\sum_{j:\, s\in A_j} x_j \geq 1, \quad \forall s \in S$$

$$x_j \geq 0, \quad \forall 1 \leq j \leq m$$

$$\min \sum_j c(A_j) \cdot x_j$$

Ezután az optimális törtmegoldást kikerekítjük: y_j legyen 1, ha $x_j \geq 1/\Delta$ és 0 különben.

83. Állítás. *Azok az A_j halmazok, melyekre $y_j = 1$ egy fedést alkotnak, melynek költsége az optimális törtmegoldásnak legfeljebb Δ-szorosa, és így persze az eredeti feladat optimumának is.*

Egy érdekes alkalmazás: érdemes meggondolni, hogy ebből egy 2-közelítő algoritmus adódik a súlyozott lefogó csúcshalmaz feladatra.

A tört relaxált feladat megoldása és az eredmény kikerekítése rengeteg közelítő algoritmus alapját képezi. Sokféleképpen lehet variálni, pl. randomizáltan végezni a kikerekítést, vagy több menetben (először csak néhány tört változót kerekítünk ki, ezeket rögzítjük, újra számolunk tört optimumot, majd ezeket ismételjük (ezt az eljárást „iterative rounding" névvel szokták illetni); ld pl. a következő fejezetet). Különböző primál-duál változatokat is szokás használni.

A tört relaxált LP alapú közelítő algoritmusoknak fontos tulajdonsága, hogy az igazi optimumra a tört optimumot használják becslésnek, tehát a közelítő hányados becslésénél az optimum helyett a tört optimum szerepel. Az ilyen becslés azonban sokszor nem is lehet nagyon jó, pl. soha nem lehet jobb, mint az úgynevezett „egészségi hányados" (integrality gap), ami a szuprémuma az x inputokon az $\frac{\mathrm{OPT}(x)}{\mathrm{OPT}_{frac}(x)}$ hányadosnak (minimalizálási feladatoknál).

20.7. Beck–Fiala-tétel

84. Tétel (Beck József és Fiala Tibor). *Adott egy hipergráf az $\{1,2,\ldots,n\}$ alaphalmazon, melynek élei $\{A_1, A_2, \ldots, A_m\}$, és tegyük fel, hogy minden pont foka legfeljebb d, és $d \geq 3$. Ekkor létezik $c : \{1,2,\ldots,n\} \mapsto \{-1,+1\}$ kétszínezése az alaphalmaznak, hogy minden A_j élben $|\sum_{i\in A_j} c(i)| \leq 2d-3$.*

Algoritmikus bizonyítást adunk, tehát meg is keressük a kívánt c színezést. Jelölje $\mathcal{J}$ az aktív élek indexeinek halmazát, kezdetben ez $\{1,2,\ldots,m\}$. Felírjuk a következő egyenletrendszert:

$$\sum_{i\in A_j} x_i = 0 \quad (\forall\, j \in \mathcal{J}).$$

Ennek az azonosan 0 egy megoldása. Ezután minden fázisban néhány x_i változó értékét rögzíteni fogjuk (-1-re, ill. $+1$-re), ezeket behelyettesítjük a fentiekbe, a nem rögzített x_i-ket aktív változóknak, indexeiket aktív indexeknek hívjuk. Egy él akkor aktív, ha van benne legalább d darab aktív index. A nem aktív élekhez tartozó egyenlőségeket mindig elhagyjuk a $\mathcal{J}$ rendszerből. A rögzítéseknél vigyázunk, hogy az egyenletrendszernek mindig legyen olyan megoldása, melyben minden aktív változó értéke szigorúan -1 és $+1$ között van. Két fázist hajtunk végre.

Az első fázis addig tart, amíg van olyan aktív él, amelyben **szigorúan több, mint d darab aktív index van**. Ilyenkor kettős leszámlálással látszik, hogy több aktív változónk van, mint aktív élünk, tehát, mivel az egyenletrendszernek van megoldása, ezért megoldásainak halmaza egy legalább 1-dimenziós altér. Mivel a feltevésünk miatt ez tartalmazza a $[-1,+1]^a$ kocka egy belső pontját (ahol a az aktív változók száma), ezért

valahol metszi a kocka felszínét, rögzítsünk egy ilyen metszéspontot. Azon aktív változókat, melyek értéke ebben a pontban -1 vagy $+1$, rögzítsük, dobjuk ki az inaktívvá vált élekhez tartozó egyenleteket, és menjünk a fázis elejére.

A második fázisban minden aktív élben pontosan d darab aktív változónk van, ezeket kerekítsük ki -1-re, ha az ismert megoldásban az értékük negatív, különben $+1$-re.

Azt állítjuk, hogy a kapott kétszínezés teljesíti a tételt. Először nézzünk meg egy olyan élet, amely valamikor az első fázis során inaktívvá vált. Ekkor még teljesült rá az egyenlőség, és csak maximum $d-1$ aktív index maradt benne. Ezeket később kikerekítettük, ezáltal az összeg értéke $< 2(d-1)$-gyel változott, így abszolút értékben legfeljebb $2d-3$-ra nőhetett meg. A második fázisban minden változó értéke maximum eggyel változott meg, így az első fázis végén aktív élekre az összeg abszolút értéke legfeljebb $d \leq 2d-3$ lett.

21. fejezet

Fix paraméteres algoritmusok

Ebben a fejezetben feltesszük, hogy minden input mellé kapunk egy k paramétert is. Például az input egy gráf, és az a feladat, hogy létezik-e a gráfban egy legalább k méretű teljes részgráf, vagy egy legfeljebb k méretű lefogó csúcshalmaz. Ezek a példafeladatok könnyen megválaszolhatók, ha tehetünk $O(n^k k^2)$ lépést, ami azt jelenti, hogy konstans k-ra polinomiálisak. Ez azonban nem igazán kielégítő, mivel pl. $k=100$ esetén az n^{100} elég gyorsan növekedő függvény. Ezért ebben a fejezetben olyan algoritmusokat keresünk, amelyek lépésszámában a k paraméter nem a kitevőben szerepel, azaz a cél $O(f(k)\cdot n^c)$ idejű algoritmus valamilyen f függvényre és c abszolút konstansra. Az ilyen algoritmust hívjuk FPT-nek. Az első (max-klikk) feladatra ilyet nem tudunk, és nem is igazán várható, a másodikra viszont igen, ilyet többet is mutatunk. A különbségek a futási időben igen jelentősek lehetnek, lásd a 16. példát.

Természetesen a paraméteres feladatokhoz is szokás bonyolultsági osztályokat definiálni. A P osztály értelemszerűen azt jelenti, hogy poli(n,k) időben megoldható, az FPT osztályban vannak azok a paraméteres problémák, amelyekre létezik $O(f(k)\cdot n^c)$ lépésszámú algoritmus, az XP pedig jelöli azok osztályát, melyekre létezik $O(n^{f(k)})$ idejű, azaz minden konstans k értékre polinom időben eldönthető. Az FPT és az XP között van az úgynevezett Weft hierarchia. Amit itt lényegesnek tartunk erről megjegyezni, az mindössze annyi, hogy a k méretű klikk létezése a W[1] osztályban van, és ott teljes, a halmazfedés feladat k halmazzal pedig W[2]-teljes.

$$\mathrm{P} \subseteq \mathrm{FPT} \subseteq \mathrm{W}[1] \subseteq \mathrm{W}[2] \subseteq \ldots \subseteq \mathrm{XP}.$$

Első példánk a SAT feladat, ahol a formula hossza n és a változók száma (ami egyben a paraméter) k. Ez nyilván $\in$FPT, hiszen $O(2^k n)$ időben az összes lehetséges értékadást kiértékelhetjük.

21.1. Steiner-fa

A második példánk a Steiner-fa feladat. Adott egy $G=(V,E)$ irányítatlan gráf pozitív c élköltségekkel, és terminálok egy $S\subseteq V$ halmaza. Keresünk egy minimális költségű összefüggő részgráfot, mely S minden csúcsát tartalmazza. Egy ilyen optimális megoldás nyilván egy fa.

Ez a feladat NP-nehéz. Először egy 2-közelítő algoritmust adunk rá.

Csinálunk egy H súlyozott teljes gráfot az S csúcshalmazon, egy uv él súlya a minimális költségű $u \rightsquigarrow v$ út költsége a G gráfban. Legyen T a H gráf minimális súlyú feszítő fája. Könnyű látni, hogy ha a minimális költségű Steiner-fa költsége OPT, akkor T súlya nem lehet több, mint az OPT kétszerese. Másrészt könnyű olyan Steiner-fát csinálni, melynek költsége nem több, mint T súlya. Helyettesítsük T minden élét egy úttal a G-ben, amelynek költsége pont a T-beli súly. Ezen utak uniójából töröljünk ki minden körből (2 hosszúakból is) éleket, amíg csak lehet, a végén egy fát kapunk.

Legyen most $k = |S|$ a paraméterünk. A következő (Dreyfustól és Wagnertől származó) dinamikus programozási algoritmus FPT. Jelölje $\mathrm{OPT}(S', v)$ annak a feladatnak az optimális megoldását, amely az $S' \cup \{v\}$ terminálhalmazra vesz egy ezt feszítő fát, ahol v tetszőleges csúcs, és $S' \subseteq S$ egy terminálhalmaz. Ezek lesznek a részfeladatok, tehát $2^k n$ részfeladatot kell megoldanunk.

Először lefuttatunk n db Dijkstra algoritmust, és minden G-beli u, v pontpárra meghatározzuk a $d(u, v)$ minimális költségű utat (ezzel az $|S'| = 1$ részfeladatokat meg is oldottuk). Utána $|S'|$ egyre növekvő értékeire dinamikus programozással kiszámítjuk az $\mathrm{OPT}(S', v)$ értékeket az alábbi állítást felhasználva:

85. Állítás.

$$\mathrm{OPT}(S', v) = \min_{u \in V} \min_{\emptyset \neq S_1 \subsetneq S'} \big(\mathrm{OPT}(S_1, u) + \mathrm{OPT}(S' - S_1, u) + d(u, v)\big).$$

86. Állítás. *A Dreyfus–Wagner-algoritmus lépésszáma* $O(3^k n^2 + nm + n^2 \log n)$.

21.2. Lefogó csúcshalmaz

A feladat az, hogy létezik-e k db csúcs, hogy minden élnek legalább az egyik végpontja ezek között van. Természetesen feltehetjük, hogy a gráfunk nem tartalmaz izolált csúcsot, ezért a menet közben keletkező izolált csúcsokat is mindig azonnal törölni fogjuk.

21.2.1. Első megoldás k-korlátos mélységű döntési fával

Rekurzív eljárást adunk meg: ha nincs él, készen vagyunk, az eddig megjelölt csúcsok lefogók. Különben, ha még van él, de már van k jelölt csúcsunk, térjünk vissza sikertelenül. Ha még nincs ennyi jelölt, válasszunk egy uv élet. Először jelöljük meg az u csúcsot, hagyjuk el a rá illeszkedő éleket, és hívjuk meg az eljárást rekurzívan a $k-1$ paraméterrel; ha ez sikertelenül tér vissza, akkor jelöljük meg v-t az u helyett, hagyjuk el a rá illeszkedő éleket, és hívjuk meg az eljárást rekurzívan. Ha ez is sikertelenül tér vissza, térjünk mi is vissza sikertelenül. Ha valamelyik hívás visszaadott egy L lefogó halmazt, akkor mi adjuk vissza L-et hozzávéve az eddig megjelölt csúcsokat.

Az egész eljárás leírható egy k mélységű bináris fával, melynek $< 2^{k+1}$ csúcsa van, és egy csúcsban $O(n)$ lépést kell tennünk, így az összes lépésszám $O(2^k n)$.

21.2.2. Második megoldás k^2 méretű kernellel

Egy G inputú k paraméterű feladat $f(k)$-*kernel*ének hívjuk a polinom időben elkészíthető legfeljebb $f(k)$ méretű G' inputot és k' paramétert, ha $k' \leq k$ és a (G', k') feladatra

akkor és csak akkor IGEN a válasz, ha az eredeti (G, k) feladatra az. A kernelt polinomiálisnak hívjuk, ha létezik c, hogy $f(k) \leq k^c$.

Megjegyzés: Ha egy feladatra van FPT algoritmus, akkor mindig létezik kernel is, de polinomiális kernel sok feladatnál valószínűleg nem létezik, mert erős (nem várt) bonyolultságelméleti következményei lennének.

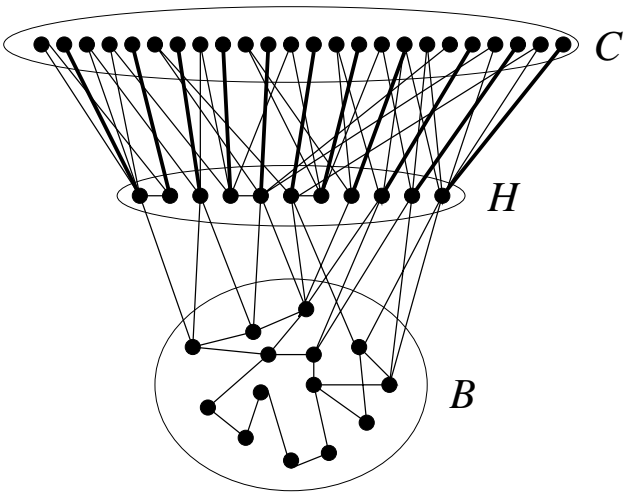
A kernelkészítő eljárásunk egész egyszerű: ha van olyan v csúcsunk, melynek foka nagyobb, mint k, akkor ezt megjegyezzük (neki nyilván benne kell lennie a lefogó csúcshalmazban, ha az legfeljebb k elemű), kitöröljük G-ből és k-t eggyel csökkentjük. Ezt ismételjük (ha k negatívvá válik, leállunk NEM válasszal), ha véget ér, akkor nyilván minden csúcs foka legfeljebb az aktuális k lesz. Ezt tekintjük kernelnek (az izolált csúcsokat persze töröljük), és a lecsökkentett k végső értékét k'-nek.

Ennek a gráfnak akkor és csak akkor van k' csúcsból álló lefogó halmaza, ha az eredetinek volt k csúcsból álló. Ha G'-nek több, mint k'^2 éle van, akkor nyilván az élei nem foghatóak le k' db legfeljebb k' fokú csúccsal. (Hasonlóan akkor sem, ha G'-nek több, mint $k'^2 + k'$ csúcsa van.)

Egyébként pedig (mivel $k' \leq k$), egy legfeljebb k^2 élű gráfról kell döntenünk (melynek legfeljebb $k^2 + k$ csúcsa van), itt kipróbálhatjuk az összes k' elemű részhalmazt legfeljebb 2^{k^2+k} lépésben. Ennél jobban járunk, ha a kernelre az előző alfejezet algoritmusát alkalmazzuk, ekkor a lépésszám csak $O(2^k k^2 + m + n)$.

21.2.3. További megoldások korona-redukcióval

Egy $G = (V, E)$ gráf *korona-felbontás*a a csúcsok partíciója három részre: $V = C \cup H \cup B$, hogy C független halmaz legyen, ne legyen él C-ből B-be, és a C és H közötti páros gráfban legyen H-t fedő párosítás.



21.1. ábra. Korona-felbontás

87. Állítás. *A G gráfban akkor és csak akkor létezik k elemű lefogó csúcshalmaz, ha a $G[B] = G - H - C$ gráfban van $(k - |H|)$ elemű lefogó csúcshalmaz.*

Egy olyan algoritmust adunk meg, mely készít egy $3k$ méretű kernelt (vagy menet közben arra a következtetésre jut, hogy nincs k méretű lefogó csúcshalmaz).

Vegyünk egy tartalmazásra nézve maximális M párosítást. Ha $|M| > k$, akkor álljunk meg NEM válasszal. Legyen X az M által fedett csúcsok halmaza, és $I = V - X$. Ekkor I független, és mivel az izolált csúcsokat töröltük, minden eleméből megy él X-be.

Tekintsük az X és I közötti páros gráfot, ebben határozzuk meg a legnagyobb M^* párosítást, és az ugyanekkora T legkisebb lefogó csúcshalmazt. Ha $|M^*| > k$, akkor NEM válasszal leállunk.

Ha $T \cap X \neq \emptyset$, akkor legyen $H = T \cap X$ és $C = I - T$, valamint B jelölje a maradék csúcsokat. Ez nyilván jó korona-felbontás ($|M^*| = |T|$ miatt M^* felpárosítja H-t a C-be), végezzük el a korona-redukciót, és menjünk az elejére.

Különben nyilván $T = I$, de $|T| = |M^*| \leq k$, másrészt $|M| \leq k$ miatt $|X| \leq 2k$, tehát $|V| \leq 3k$, a kernel elkészült.

Egy ilyen redukció nyilván megoldható $O(|M^*|m) = O(km)$ lépésben, és mivel minden redukciós lépéskor k csökken, a kernel elkészítése összesen $O(k^2 m)$ idő. Utána a kernelben az első alfejezet algoritmusa $O(2^k \cdot k)$ idő alatt végez.

Picit okosabban $2k$ méretű kernel is készíthető, ráadásul közvetlenül, egy lépésben. Csináljunk egy P páros gráfot a $G = (V, E)$ gráfból, melynek alsó osztálya V_1, a felső V_2, és $u_1 w_2$ él, ha G-ben uw él (ekkor persze $w_1 u_2$ is éle P-nek). Ebben keressünk M legnagyobb párosítást és $T \subseteq V_1 \cup V_2$ legkisebb lefogó csúcshalmazt Kőnig algoritmusával. Ha $|T| > 2k$, akkor álljunk meg NEM válasszal, mivel ha létezne a G gráfnak egy $T^* \subseteq V$ legfeljebb k elemű lefogó csúcshalmaza, akkor az $\{u_1, u_2 \mid u \in T^*\}$ a P gráf $2k$ elemű lefogó csúcshalmaza lenne. Tehát Kőnig algoritmusát $2k+1$ javítás után leállíthatjuk, (mivel ekkor már tudjuk, hogy $|T| = |M| \geq 2k+1$), így futási ideje $O(km)$.

Egy $X \subseteq V$ halmazhoz rendeljük hozzá $i = 1,2$ esetén az $X_i = \{v_i \in V_i \mid v \in X\}$ halmazt. Legyen $H = \{v \in V \mid v_1 \in T \text{ és } v_2 \in T\}$ és $B = \{v \in V \mid v_1 \in T \setminus H_1 \text{ vagy } v_2 \in T \setminus H_2\}$; valamint $C = V - H - B = \{v \in V \mid v_1 \in V_1 \setminus T \text{ és } v_2 \in V_2 \setminus T\}$.

Mindezt úgy is mondhatjuk, hogy egy $v \in V$ csúcs aszerint kerül be H-ba, B-be vagy C-be, hogy $|\{v_1, v_2\} \cap T|$ értéke 2, 1, vagy 0.

Ekkor egyrészt C független, mert ha ab él lenne C-n belül, akkor $a_1 b_2$ lefogatlan éle lenne P-nek. Másrészt T lefogó tulajdonsága és $|T| = |M|$ miatt minden M-élnek pontosan az egyik vége van T-ben, azaz egy M-él nem köthet össze két T-beli csúcsot, de T M-fedett. Ha G-ben létezne cb él C-ből B-be, akkor $c_1 b_2$ és $c_2 b_1$ éle lenne P-nek, de b_1 és b_2 közül az egyik nincs T-ben, így megint egy lefogatlan élet kapnánk.

Az M-ről tett megállapítások miatt (használva, hogy nincs cb él) a $B_1 \cap T$ csúcsok mindegyikére illeszkedik egy M-él, ami csak $B_2 \setminus T$-be mehet, hasonlóan a $B_2 \cap T$ csúcsok mindegyikére illeszkedik egy M-él, ami csak $B_1 \setminus T$-be mehet. Ezek együtt $|B|$ darab M-él, amik fedik a $B_1 \cup B_2$ halmazt, tehát a H_1-re illeszkedő M-élek mind C_2-be mennek. Így ezek megfelelői G-ben felpárosítják a H halmazt a C-be. Tehát $V = C \cup H \cup B$ korona-felbontás.

Mivel $|B| + 2|H| = |T| \leq 2k$, ezért a korona-redukció után megmaradó csúcsok száma $|B| \leq 2k$. Tehát $G[B], k - |H|$ a kívánt kernel.

Megjegyzés: Persze azt is megtehetjük, hogy először elkészítjük az első, legfeljebb k^2 élű kernelünket $O(m)$ időben, majd erre alkalmazzuk ezt a megoldást, így a legfeljebb $2k$ méretű kernelünk elkészül $O(m + k^3)$ időben.

21.2.4. További javítások

A 21.2.1 alfejezet algoritmusa is javítható egy egyszerű trükkel.

Válasszunk egy maximális fokszámú v csúcsot. Ha $d(v) \leq 2$, akkor a minimális lefogó csúcshalmaz mérete könnyen meghatározható szélességi keresés segítségével $O(n)$ lépésben (minden komponens egy út vagy egy kör). Ha pedig $d(v) \geq 3$, akkor kipróbáljuk a két lehetőséget: egyik ágon v lesz jelölt és rekurzívan hívunk $k-1$ paraméterrel a $G-v$ gráfra, a másik ágon a v csúcs $N(v)$ szomszédsága lesz jelölt, és rekurzívan hívunk $k-d(v)$ paraméterrel a $G-N(v)$ gráfra. A rekurzív hívások számára $T(k) \leq T(k-1)+$ $+T(k-3)$ lesz igaz, ebből $T(k) < 1{,}466^k$. Tehát az összes lépésszám $1{,}466^k \cdot n$.

Ha ezt az eljárást az előző kernelre alkalmazzuk, akkor annak elkészítésével együtt a lépésszám $O(1{,}466^k \cdot k + m + k^3)$.

A jelenleg ismert legjobb futási idő $O(1{,}274^k + km)$.

Azonban az eddig vizsgált paraméter nem igazán jó, csak igen speciális gráfoknak van pici lefogó csúcshalmazuk. Tudjuk, hogy a legnagyobb párosítás $\nu(G)$ mérete is egy alsó becslés a lefogó csúcsok minimális számára. Egy barátságosabb paraméterezésű feladat a következő: döntsük el, hogy van-e G-nek legfeljebb $\nu(G)+k$ méretű lefogó csúcshalmaza? Erre is ismert $O(4^k \cdot n^c)$ idejű algoritmus.

21.3. Egzakt út

El akarjuk dönteni, hogy egy $G = (V, E)$ gráfban van-e olyan $s \rightsquigarrow t$ út, melynek pontosan k éle van. Erre itt egy randomizált algoritmust adunk, amelyet azonban lehet derandomizálni is, ezt itt nem részletezzük.

Először is tegyük fel, hogy $V - s$ csúcsai ki vannak véletlen módon színezve az $C = \{1,2,\ldots,k\}$ színekkel. Egy utat színteljesnek hívunk, ha s-től különböző pontjai páronként különböző színűek, és C minden eleme szerepel színként. Persze ha találunk egy színteljes utat, akkor az pontosan k db s-től különböző csúccsal fog rendelkezni. Tehát a randomizált algoritmusunk egyszerűen abból áll, hogy sorsolunk sok független színezést, és mindegyikhez keresünk színteljes utat. Ha egyszer is találunk egy t-be vezetőt, akkor az jó út lesz, ha soha, akkor azt válaszoljuk, hogy nincsen pontosan k élű út t-be.

88. Állítás. *Ha $270e^k$ független véletlen színezés egyikére sincs szín-teljes $s \rightsquigarrow t$ út, akkor annak valószínűsége, hogy G-ben van pontosan k élű $s \rightsquigarrow t$ út legfeljebb e^{-270}.*

Színteljes utat pedig könnyű keresni dinamikus programozással. A megoldandó részfeladatok: minden v csúcsra és $C' \subseteq C$ színhalmazra keresünk C'-szín-teljes utat s-ből v-be. Ez könnyedén megy $O(2^k m)$ lépésben.

Harmadik rész

Függelék

22. fejezet

Pszeudokód

A pszeudokód az algoritmusok leírásának tömör nyelve, hasonlóan ahhoz, ahogy matematikai állításokat a matematikai logika nyelvén írunk le. Itt az ebben a jegyzetben használt pszeudokódot írjuk le, más könyvek sokszor ettől akár lényegesen eltérő formátumot használnak.

Értékadás, tömbök

$A := 0$
    /* *Az A változó értéke legyen 0.*

$A := 3 \cdot A + B/2$
    /* *Kiolvassuk az A változó értékét, beszorozzuk 3-mal, majd kiolvassuk a B változó értékét, elosztjuk 2-vel, a két részeredményt összeadjuk, majd az eredményt tároljuk az A változóban.*

$A{+}{+}$
    /* *Az A változó értékét eggyel növeljük.*

$A{-}{-}$
    /* *Az A változó értékét eggyel csökkentjük.*

CSERE(A, B)
    /* *Az A és B változók értékét kicseréljük.*

$A(i)$
    /* *Az A tömb i. eleme.*

$A[p : r]$
    /* *Az A tömb p. elemtől az r. elemig terjedő része.*

$A(i, j)$
    /* *Az A mátrix i. sorának j. eleme.*

Eljárások

k-adik$(A[1 : n], k)$:
    /* *Egy eljárás kezdete és a bemeneti paraméterek.*

return(**„BAL"**, v)
    /* *Az eljárás leáll, és visszatér két értékkel, a „BAL" szöveggel és a v változó jelenlegi értékével.*

print MINTÖR(A)

/∗ Meghívjuk a MINTÖR nevű eljárást az A paraméterrel, és a visszatérési értéket kinyomtatjuk, mint az output következő elemét.

Feltételek

if $A > B$ **then** *utasítás*

/∗ Ha A értéke nagyobb, mint a B-é, akkor végrehajtódik az utasítás, különben nem.

if $A > B \,||\, T(i) = 0$ **then**

utasítás1

utasítás2

/∗ Ha A értéke nagyobb, mint a B-é VAGY a T tömb i. eleme 0, akkor végrehajtódnak az alatta levő, beljebb kezdett utasítások, különben nem.

if $A > B \;\&\&\; T(i) = 0$ **then**

utasítás1

utasítás2

else *utasítás3*

/∗ Ha A értéke nagyobb, mint a B-é ÉS a T tömb i. eleme 0, akkor végrehajtódik utasítás1 és utasítás2, különben pedig végrehajtódik az utasítás3.

Ciklusok

for $i = 0..m \;\; C(i) := 0$

/∗ Nullázzuk a C tömböt elemeit a 0.-tól az m.-ig.

for $j = n..1 \quad (-1)$

utasítás1

utasítás2

/∗ A j változó értéke először n, majd $n-1$, majd így tovább, mindig eggyel csökken egészen az 1 értékig.
Minden értékre végrehajtódnak az alatta levő, beljebb kezdett utasítások.

for $u\underline{v} \in E$

utasítás1

utasítás2

/∗ Az u csúcs fix, a v csúcs végigfut az u ki-szomszédain, és minden lehetséges v szomszédra végrehajtódnak az alatta levő, beljebb kezdett utasítások.

while $i > 1$ **&&** $A(\lfloor \frac{i}{2} \rfloor) > AA$

utasítás1

utasítás2

/∗ Mindaddig, amíg teljesül az, hogy $i > 1$ ÉS $A(\lfloor \frac{i}{2} \rfloor) > AA$, addig végrehajtódnak az alatta levő, beljebb kezdett utasítások.

repeat

utasítás1

utasítás2

*/∗ A beljebb kezdett utasításokat ismételjük, amíg **return** utasításra nem érünk.*

23. fejezet

Példák

3. Példa (Algoritmusok futási idejére).

Az alábbi táblázatban néhány algoritmus hozzávetőleges futási idejét vizsgáljuk, egymilliárd művelet/mp sebességet feltételezve. Baloldalt az n input-méret szerepel, felül a $t_a(n)$ lépésszám.

n	$100000 \cdot n$	$15500 \cdot n \cdot \log n$	$1000 \cdot n^2$	$10 \cdot n^3$	$(1/10) \cdot n^4$	$(8/10^{24}) \cdot 2^n$
100	*10 msec*	*10 msec*	*10 msec*	*10 msec*	*10 msec*	*10 msec*
1000	*100 msec*	*150 msec*	*1 sec*	*10 sec*	*100 sec*	$3 \cdot 10^{261}$ *years*
10000	*1 sec*	*2 sec*	*100 sec*	*3 hours*	*12 days*	$5 \cdot 10^{2970}$ *years*

Megjegyzés: Az univerzum kora kevesebb, mint $1{,}4 \cdot 10^{10}$ év.

4. Példa (Öt elem közül a középső kiválasztására).

1. *Hasonlítsunk össze két elemet, majd másik kettőt.*
2. *Hasonlítsuk össze azt a kettőt, amelyek a kisebbek voltak.*
3. *Most elnevezzük az elemeket az eddigi eredmények alapján: A 2.-nál a kisebb elem a c, akivel őt az 1.-ben hasonlítottuk, az a d. A 2.-nál a nagyobb elem a b, akivel őt az 1.-ben hasonlítottuk, az az a (tehát $c < b < a$ és $c < d$). Akit még nem hasonlítottunk senkivel, az az e.*
4. *Hasonlítsuk össze e-t és d-t.*

   4a) Ha $e > d$ (tehát $c < d < e$), akkor hasonlítsuk össze b-t és d-t.

      5aa) Ha $b > d$, akkor hasonlítsuk b-t e-vel, amelyik kisebb, az lesz a középső.

      5ab) Ha $b < d$, akkor hasonlítsuk d-t a-val, amelyik kisebb, az lesz a középső.

   4b) Ha $e < d$, akkor hasonlítsuk össze b-t és e-t.

      5ba) Ha $e < b$, akkor hasonlítsuk b-t d-vel, amelyik kisebb, az lesz a középső.

      5bb) Ha $e > b$, akkor hasonlítsuk a-t e-vel, amelyik kisebb, az lesz a középső.

5. Példa (A SZÉTOSZTÁSRA).

4	6	3	1	2	7	5

$x = 3$

4	6	5	1	2	7	3

1 **i**	6	5	4	2	7	3

1	2 **i**	5	4	6	7	3

1	2	3 **i**	4	6	7	5

6. Példa (A LESZÁMLÁLÓ rendezésre).

3	4	5	2	1	3	0

A tömb

1	1	1	2	1	1

C tömb az első ciklus után

1	2	3	5	6	7

C tömb a második ciklus után

0				3		

B tömb az utolsó ciklus kétszeri lefutása után

0	2	3	4	6	7

C tömb ugyanekkor

0	1	2	3	3	4	5

B tömb (Az eljárást folytatva a kimeneti B tömb)

7. Példa (A SZÁMJEGYES rendezésre).

3	*4*	*9*	*Először a*	*1*	*1*	*0*	*Utána a*	*1*	*1*	*0*	*Végül az*	*1*	*1*	*0*
3	*5*	*1*	*harmadik*	*3*	*5*	*1*	*második*	*6*	*4*	*3*	*első oszlop*	*2*	*6*	*9*
1	*1*	*0*	*oszlop*	*6*	*4*	*3*	*oszlop*	*3*	*4*	*9*	*szerint.*	*3*	*4*	*9*
6	*4*	*3*	*szerint*	*4*	*5*	*7*	*szerint.*	*3*	*5*	*1*		*3*	*5*	*1*
2	*6*	*9*	*rendezzük.*	*3*	*4*	*9*		*4*	*5*	*7*		*4*	*5*	*7*
4	*5*	*7*		*2*	*6*	*9*		*2*	*6*	*9*		*6*	*4*	*3*

8. Példa (Összeadás).

$$\begin{array}{r} 10110010 \\ +101100100 \\ \hline 1000010110 \end{array}$$

9. Példa (Kivonás).

$$\begin{array}{r} 1000010110 \\ -101100100 \\ \hline 10110010 \end{array}$$

10. Példa (Szorzás).

10110010 · 101
101100100
10110010
1101111010

11. Példa (Maradékos osztás).

1101101 : 101 = 10101
111
1001
100 (ez a maradék)

12. Példa (Hátizsák). *Nézzünk egy egyszerű példát, ahol a hátizsák teherbírása W=2, és a tárgyak súlyai, illetve értékei az alábbi táblázat szerint alakulnak:*

Súly	$w_1 = 1$	$w_2 = 1$	$w_3 = 2$
Érték	$e_1 = 2$	$e_2 = 4$	$e_3 = 3$

Inicializáló for ciklus eredménye: $T(0) := 0;\ T(1) := 0;\ T(2) := 0.$

A további for ciklusok eredményei:

$i = 1;\ \ j = 2$
if $T(2-1)+e_1 > T(2)$ ***then*** $T(2) := T(2-1)+e_1$
$0+2 > 0 \to T(2) = 0+2 = 2$
$i = 1;\ \ j = 1$
if $T(1-1)+e_1 > T(1)$ ***then*** $T(1) := T(1-1)+e_1$
$0+2 > 0 \to T(1) = 0+2 = 2$

$i = 2;\ \ j = 2$
if $T(2-1)+e_2 > T(2)$ ***then*** $T(2) := T(2-1)+e_2$
$2+4 > 2 \to T(2) = 2+4 = 6$
$i = 2;\ \ j = 1$
if $T(1-1)+e_2 > T(1)$ ***then*** $T(1) := T(1-1)+e_2$
$0+4 > 2 \to T(1) = 0+4 = 4$

$i = 3;\ \ j = 2$
if $T(2-2)+e_3 > T(2)$ ***then*** $T(2) := T(2-2)+e_3$
$0+3 > 6$ ***nem teljesül.***

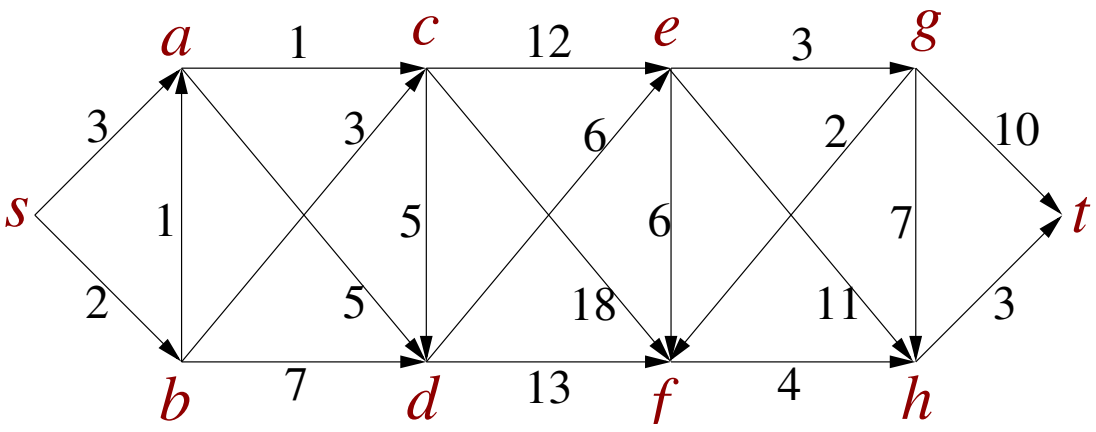
Mivel már nem lépünk be újból semelyik ciklusba se, ezért $T(W) = 6$***. Tehát a maximálisan kivihető érték*** 6***.***

13. Példa (Mátrix-szorzás zárójelezésére). *Az $A \cdot B \cdot C \cdot D$ szorzást kell elvégeznünk, ahol a mátrixok méretei sorban:* $1000 \times 10, 10 \times 1, 1 \times 1000, 1000 \times 10$.

Ha balról jobbra végezzük el: $((A \cdot B) \cdot C) \cdot D$, akkor a szorzások száma: $1000 \cdot 10 \cdot 1 + {} + 1000 \cdot 1 \cdot 1000 + 1000 \cdot 1000 \cdot 10 = 11010000$.

Ha azonban így zárójelezzük: $(A \cdot B) \cdot (C \cdot D)$, akkor a szorzások száma: $1000 \cdot 10 \cdot 1 + {} + 1 \cdot 1000 \cdot 10 + 1000 \cdot 1 \cdot 10 = 30000$.

14. Példa (Dijkstra-algoritmusra).
Az alábbi gráfban keressük s-ből t-be a legrövidebb utat.

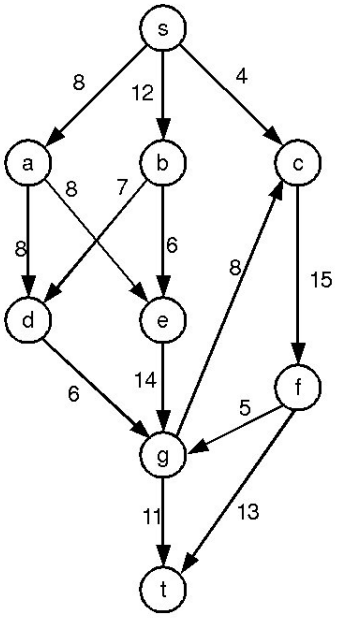


23.1. ábra. (Feladat a Dijkstra-algoritmushoz)

S-be kerülő csúcs és címkéje	*P*-beli csúcsok, ahol címke változik
$s:0$	$a:3$ $b:2$
$b:2$	$c:5$ $d:9$
$a:3$	$c:4$ $d:8$
$c:4$	$e:16$ $f:22$
$d:8$	$e:14$ $f:21$
$e:14$	$g:17$ $h:25$ $f:20$
$g:17$	$t:27$ $h:24$ $f:19$
$f:19$	$h:23$
$h:23$	$t:26$
$t:26$	

Tehát a legrövidebb út hossza 26, *egy legrövidebb út:* $s, b, a, d, e, g, f, h, t$.

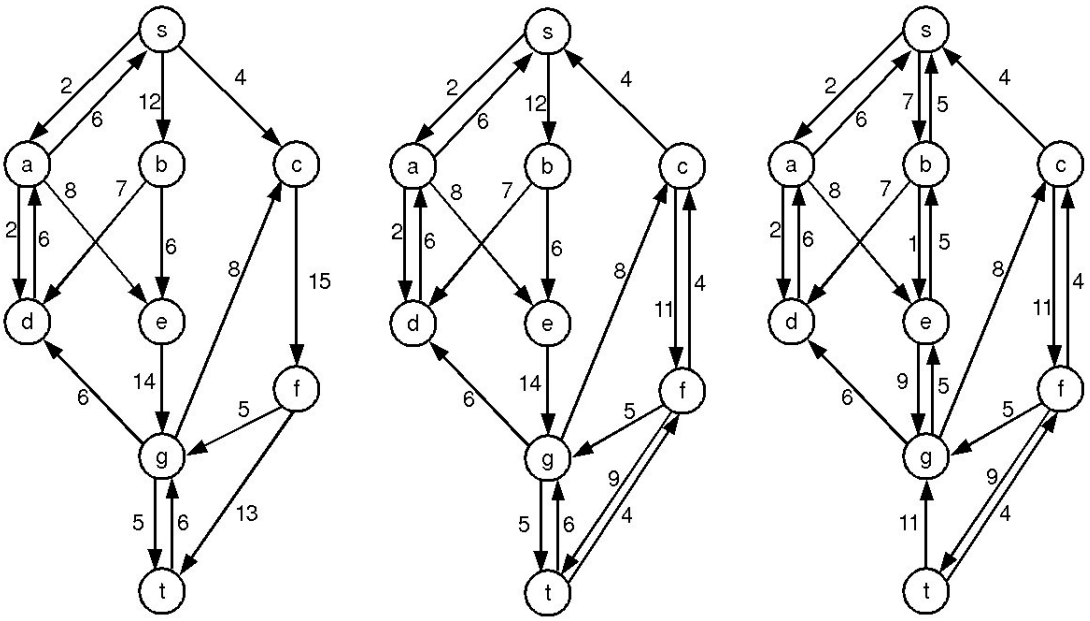
15. Példa (Ford–Fulkerson algoritmusának működésére). *Az alábbi hálózaton keresünk maximális folyamot:*



23.2. ábra. (G,c,s,t)

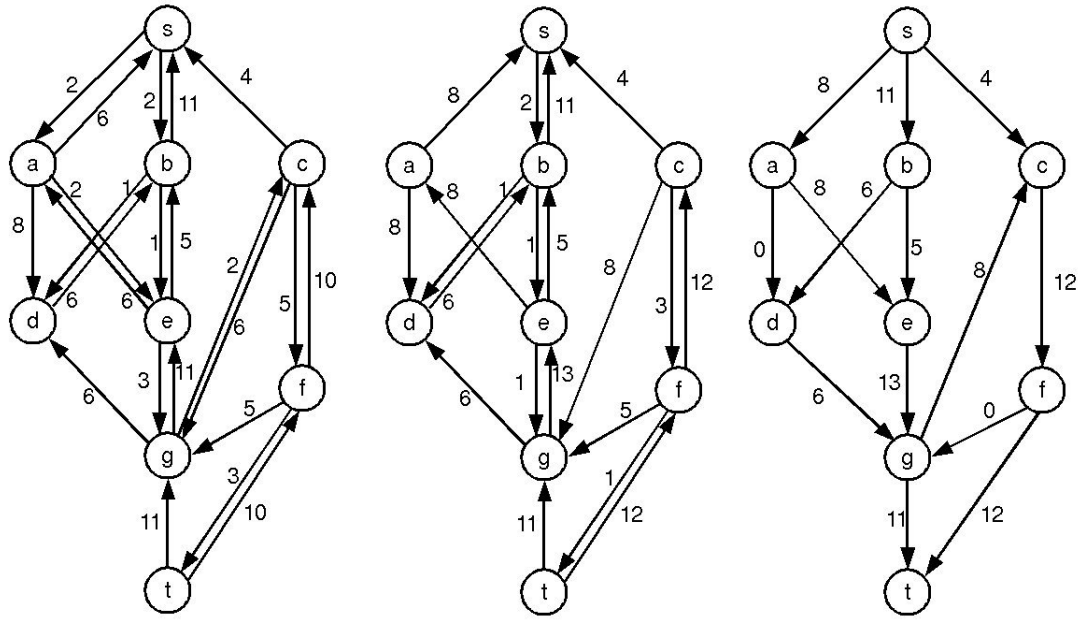
1. lépés: $\forall e : f(e) := 0$
2. lépés: az összes él telítetlen, és előre-él G'*-ben* $(G' = G)$*.*
$\forall e : r(e) := c(e) - f(e)$
3. lépés: $s \rightsquigarrow t$ *út keresése* G'*-n:* $P : s, a, d, g, t.$
4. lépés: Javít(P)*:* $\Delta = 6$
$f(sa) := 6, f(ad) := 6, f(dg) := 6, f(gt) := 6$
5. lépés: új G' *elkészítése: 23.3. ábra.*

6. lépés: $s \rightsquigarrow t$ *út keresése* G'*-n:* $P : s, c, f, t.$
7. lépés: Javít(P)*:* $\Delta = 4$
$f(sc) := 4, f(cf) := 4, f(ft) := 4$
8. lépés: új G' *elkészítése: 23.3. ábra.*
9. lépés: $s \rightsquigarrow t$ *út keresése* G'*-n:* $P : s, b, e, g, t.$
10. lépés: Javít(P)*:* $\Delta = 5$
$f(sb) := 5, f(be) := 5, f(eg) := 5, f(gt) := 11$
11. lépés: új G' *elkészítése: 23.3. ábra.*
12. lépés: $s \rightsquigarrow t$ *út keresése* G'*-n:* $P : s, b, d, a, e, g, c, f, t.$
13. lépés: Javít(P)*:* $\Delta = 6$
$f(sb) := 11, f(bd) := 6, f(da) := 0, f(ae) := 6,$
$f(eg) := 11, f(gc) := 6, f(cf) := 10, f(ft) := 10$



23.3. ábra. 5., 8. és 11. lépések

14. lépés: új G' elkészítése: 23.4. ábra.



23.4. ábra. 14. és 17. lépések, és az elkészült folyam

15. lépés: $s \rightsquigarrow t$ út keresése G'-n: $P : s, a, e, g, c, f, t$.
16. lépés: Javít(P): $\Delta = 2$

$f(sa) := 8, f(ae) := 8, f(eg) := 13, f(gc) := 8, f(cf) := 12, f(ft) := 12$
17. lépés: új G' *elkészítése: 23.4. ábra.*
Itt nem találtunk $s \rightsquigarrow t$ *utat, az* s*-ből elérhető pontok halmaza:*
$S = \{s, a, b, d, e, g\}$*, tehát* $T = \{c, f, t\}$.

A maximális folyam a 23.4. ábrán látható, folyamértéke: $|f| = 8 + 11 + 4 = 23$. *Ellenőrzés:* $c(S, T) = 4 + 8 + 11 = 23$.

16. Példa (Paraméteres algoritmusok lépésszáma). *Az alábbi táblázatban néhány paraméteres algoritmus hozzávetőleges futási idejét vizsgáljuk, 100 millió művelet/mp sebességet feltételezve. Baloldalt az* n *input-méret (most pl. egy gráf csúcsszáma) és a* k *paraméter szerepel, felül a lépésszám. Az univerzum kora* $1{,}4 \cdot 10^{10}$ *év.*

n, k	n^k	$4^k \cdot n$	$2^k \cdot n^2$	$1{,}466^k \cdot n$	$1{,}274^k + k \cdot n$
1000, 10	$3 \cdot 10^{14}$ *years*	10,5 *sec*	10 *sec*	0,5 *msec*	0,1 *msec*
10000, 10	$3 \cdot 10^{24}$ *years*	1,75 *min*	17 *min*	5 *msec*	1 *msec*
1000, 50	$3 \cdot 10^{134}$ *years*	$4 \cdot 10^{17}$ *years*	357000 *years*	34 *min*	2 *msec*

24. fejezet

Források, ajánlott irodalom

Magyar nyelvű könyvek

- Rónyai–Ivanyos–Szabó: Algoritmusok, Typotex, 1998.
- Cormen, Leiserson, Rivest, Stein: Új Algoritmusok, Scolar Kiadó, 2003.
- Katona–Recski–Szabó: A számítástudomány alapjai, Typotex, 2002.
- Gács–Lovász: Algoritmusok, Tankönyvkiadó, 1989.
- Knuth: A számítógép-programozás művészete, Műszaki Kiadó, 1987.
- Aho–Hopcroft–Ullman: Számítógépalgoritmusok tervezése és analízise, Műszaki Kiadó, 1982.

Magyar nyelvű on-line jegyzetek

- Lovász: Algoritmusok bonyolultsága, www.cs.elte.hu/~kiraly/Algbony.pdf
- Király: Adatstruktúrák, www.cs.elte.hu/~kiraly/Adatstrukturak.pdf
- Frank András jegyzetei, www.cs.elte.hu/~frank/jegyzet/jegyzet.html

Angol nyelvű könyvek

- Schrijver: Combinatorial Optimization, Springer, 2003.
- Vazirani: Approximation Algorithms, Springer, 2003.
- Cygan–Fomin–Kowalik–Lokshtanov–Marx–Pilipczuk–Pilipczuk–Saurabh: Parameterized Algorithms, Springer, 2015.