ALGORITMUSELMÉLET

Jegyzetek és példatárak a matematika egyetemi oktatásához sorozat

Algoritmuselmélet
Algoritmusok bonyolultsága
Analitikus módszerek a pénzügyben és a közgazdaságtanban
Analízis feladatgyűjtemény I
Analízis feladatgyűjtemény II
Bevezetés az analízisbe
Complexity of Algorithms
Differential Geometry
Diszkrét matematikai feladatok
Diszkrét optimalizálás
Geometria
Igazságos elosztások
Introductory Course in Analysis
Mathematical Analysis – Exercises I
Mathematical Analysis – Problems and Exercises II
Mértékelmélet és dinamikus programozás
Numerikus funkcionálanalízis
Operációkutatás
Operációkutatási példatár
Parciális differenciálegyenletek
Példatár az analízishez
Pénzügyi matematika
Szimmetrikus struktúrák
Többváltozós adatelemzés
Variációszámítás és optimális irányítás

Király Zoltán

ALGORITMUSELMÉLET

Eötvös Loránd Tudományegyetem
Természettudományi Kar

Typotex

2014

Lektorálta: ifj. Katona Gyula

ISBN 978 963 279 241 5
Készült a Typotex Kiadó (`http://www.typotex.hu`) gondozásában
Felelős vezető: Votisky Zsuzsa
Műszaki szerkesztő: Gindilla Orsolya

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

KULCSSZAVAK: Algoritmus, gráf, rendezés, kiválasztás, aritmetika, dinamikus programozás, adatszerkezet, legrövidebb utak, hasítás, párosítás, folyam, közelítő algoritmus, fix paraméteres algoritmus.

ÖSSZEFOGLALÁS: E jegyzet elsősorban matematikus és informatikus egyetemi hallgatók számára készült. Célja a tömörség volt: mintegy 40, egyenként 45 perces előadás vázlatát tartalmazza, így önálló tanulásra nem igazán alkalmas – a megértéshez fontosak az előadáson elhangzottak is. Az első rész az ELTE Matematikai Elemző szakán a „Gráfok és Algoritmusok Elmélete” című tárgy beindításakor tartott előadásaim alapján készült. A második részben a Matematika és az Alkalmazott Matematika MSc program közös „Algoritmuselmélet” című törzsanyag tárgyának jelentős részéhez találhatók jegyzetek. A harmadik rész függelék, a jegyzetben használt pszeudokód-formátum magyarázatát és az alapvető algoritmusokra néhány jól követhető példát tartalmaz.

Tartalomjegyzék

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 23. 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 egy-oldalasan 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

I. 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. A legelterjedtebb egy konkrét elméleti számítógép-modellt, a Turing gépet használó definíció, 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 bementekre 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 21. fejezetben.

Megjegyzés. 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 gépfüggetlen 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. 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: $t_A(n) := \max_{\{x:\ |x|=n\}}\{\text{Az } A \text{ algoritmus hány lépést tesz az } x \text{ inputon}\}$. Az input általában egy bitsorozat: $x \in \{0,1\}^n$, ennek hossza: $|x| = n$.

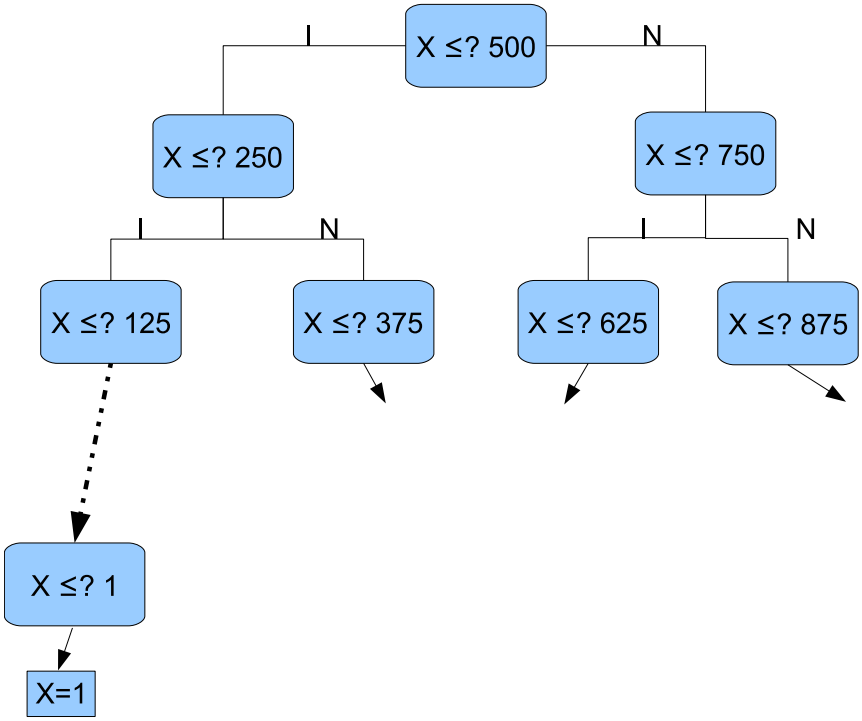
2. Definíció. Legyenek f és $g : \mathbb{N} \to \mathbb{R}_0^+$ függvények. Azt mondjuk, hogy $f = O(g)$ (ejtsd: nagy ordó g), ha léteznek $n_0, c \in \mathbb{N}$ konstansok, hogy minden $n \geq n_0$ egészre $f(n) \leq c \cdot g(n)$.

1.2. Barkochba

Feladat: 0 és 1000 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. A 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 $a_{i_1} \leq a_{i_2} \leq \leq \ldots \leq a_{i_n}$.

2. Tétel. *A rendezési feladatnál az algoritmus döntéseit leíró fának legalább* $n!$ *levele van és így a fa mélysége* $\geq \log(n!) \geq n \cdot \log n - \frac{3}{2} \cdot n$. *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.*

Keresési/kiválasztási feladat:

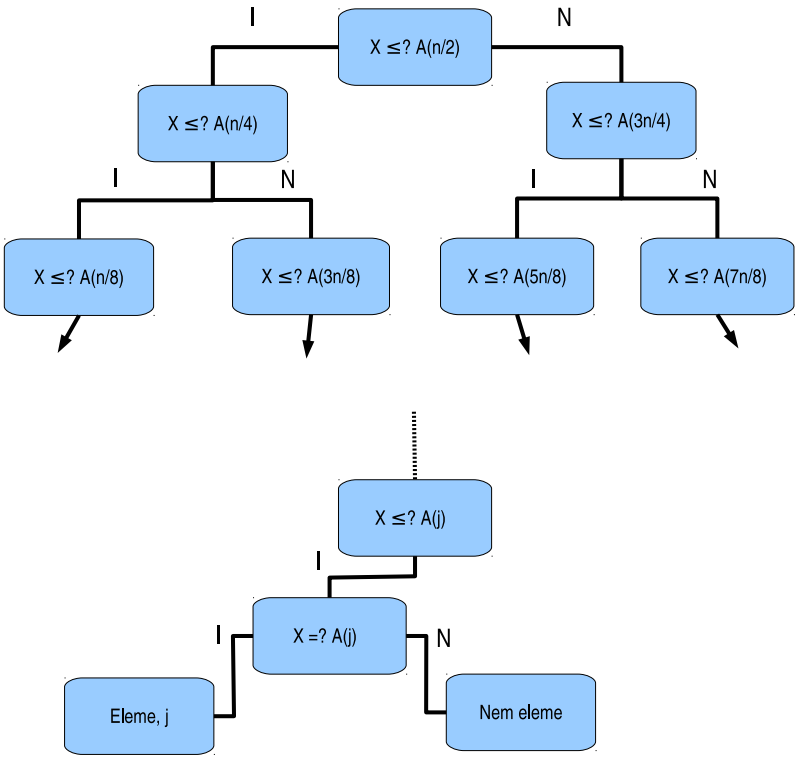
Bemenet: Az n (különböző) számból álló $A = \{a_1, a_2, \ldots, a_n\}$ halmaz és egy k szám, amelyre fennáll, hogy $1 \leq k \leq n$.

Kimenet: Az $a_{i_k} \in A$ elem, amelyik A pontosan $k-1$ eleménél nagyobb, valamint ennek i_k indexe a bemenetben.

Megjegyzés. Természetesen meg szokás engedni, hogy az inputban azonos számok is legyenek, mi most az algoritmusok és elemzéseik egyszerűsítése miatt ezt a keresési feladatoknál megtiltjuk. Keresési, kiválasztási, illetve rendezési feladatra megoldást adó algoritmust többfélét is megvizsgálunk, a közismertebb rendezési algoritmusokat a hallgatóság a gyakorlat keretében ismétli át. A kiválasztási feladat mindig megoldható úgy is, hogy először rendezünk, utána kiíratjuk a megfelelő elemet, de mi mindig ennél gyorsabb megoldást keresünk.

2.1. Felező keresés

Feladat: Adott egy U rendezett univerzum, ennek egy S részhalmazát szótárnak nevezzük. 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.)



2. Algoritmus. Egy $A[1 : n]$ tömbbe rendezzük (növekedően) a szótárat. Ezután egy x kérdés esetén kibarkochbázzuk az indexét, és a végén rákérdezünk. Pontosabban (lásd az ábrát): 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.

Lépésszám: $\lceil \log(n) \rceil + 1$

3. 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.*

2.2. Legnagyobb elem

Feladat: Keressük meg melyik elem a legnagyobb az adott inputban.

3. Algoritmus.

```
M := a(1)    /* Beválasztunk egy elemet az inputból.
H := 1       /* Megjegyezzük a helyét.
for i = 2..n    /* Sorra vizsgáljuk a többi elemet.
  if a(i) > M then M := a(i); H := i   /* Amennyiben a(i) nagyobb
                                          M-nél, akkor lecseréljük M-et a(i)-re és H-t pedig i-re.
```

Lépésszám: $O(n)$

Elemzés: Nyilván a végén M fogja tartalmazni a maximális értéket, H pedig ezen elem indexét az inputban.

Az összehasonlítások száma pontosan $n - 1$. Az értékadások száma legfeljebb $2n$. Összesen tehát a lépésszám $O(n)$.

4. 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. Így azt is megállapíthatjuk, hogy ez az algoritmus egyben optimális is.

2.3. Egyszerre a legnagyobb és a legkisebb elem

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

Nyilván $2n - 3$ összehasonlítás elég az előző módszerrel, azonban ennél jobbat szeretnénk elérni.

4. Algoritmus. Az elemeket párokba állítjuk, és a párokat összehasonlítjuk, majd 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.

Lépésszám: $O(n)$.

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

2.4. Két legnagyobb elem

Feladat: Keressük meg a két legnagyobb elemet.

5. Algoritmus. Az elemeket párba állítjuk, és összehasonlítjuk a párokat. Utána vesszük a nagyobbakat, ezeket ismét párba állítjuk, és összehasonlítjuk a párokat. És így tovább, nyilván $\lceil \log n \rceil$ forduló és pontosan $n - 1$ összehasonlítás után megkapjuk a legnagyobb elemet. A legnagyobb elemet csak $\lceil \log n \rceil$ másik elemmel hasonlítottuk össze, és nyilván a második legnagyobb csak ezek közül kerülhet ki. Ezen elemek maximumát $\lceil \log n \rceil - 1$ összehasonlítással megkaphatjuk.

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

2.5. Adatszerkezetek – bevezetés

Adatszerkezetnek 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.

2.6. 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.

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 csúcs gyerekei azok a csúcsok, amelyeknek ő a szülője, ha ilyen nincs, a 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.

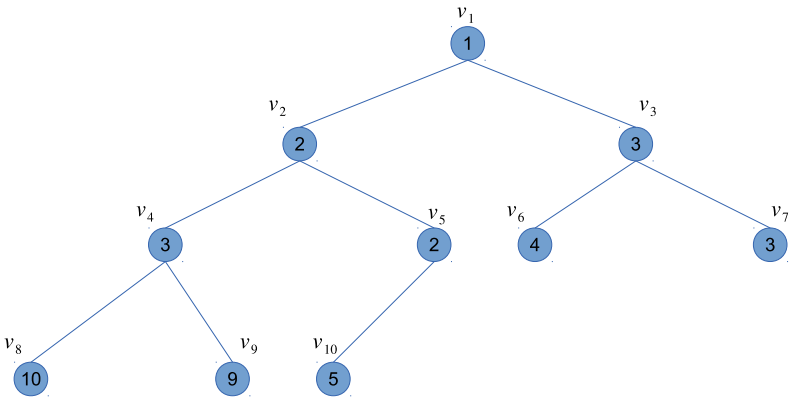
A bináris kupac váza egy majdnem teljesen kiegyensúlyozott bináris fa. Pontosabban olyan fákat használunk erre a célra, amelyekben vagy minden levél ugyanolyan távol van a gyökértől (ezekben pontosan $2^{d+1}-1$ csúcs van, ha mélységük d), vagy pedig a levelek két szinten helyezkednek el (minden levél 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 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 $v \neq$gyökér 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 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, max. 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.*

2.6.1. Műveletek a kupac adatszerkezetben

6. Algoritmus (BESZÚR)**.**

BESZÚR(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) */* Kupac-rendezettséget 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: $O(\log n)$

7. 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$(A(1), A(\text{VÉGE}))$ /* *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$(A(\text{VÉGE}+1))$ /* *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. Állítás. *Az n csúcsú szép fa mélysége $\lfloor \log n \rfloor$, tehát a BESZÚR művelet $O(\log n)$ lépés, és legfeljebb $\log n$ összehasonlítás kell. A MINTÖRLÉS-nél a lépésszám szintén $O(\log n)$, és legfeljebb $2 \log n$ összehasonlítás történik.*

2.6.2. Kupacos rendezés

Végül rátérhetünk a rendezési feladatot megoldó algoritmusra.

8. Algoritmus (Kupacos rendezés I.)**.**

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

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 kupac-rendezzük.

9. 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*
print MINTÖR(A)

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

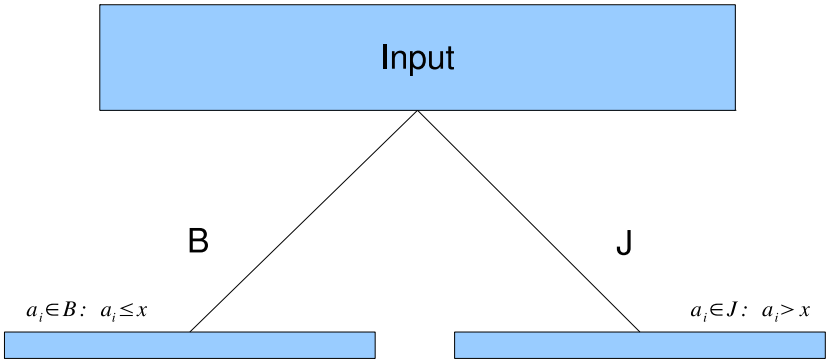
Lépésszám (kupacépítésnél): $O(n)$, a kupacos rendezés továbbra is $O(n \log n)$. De ha csak a k darab legkisebb elemre vagyunk kíváncsiak növekvő sorrendben, akkor $O(n + k \log n)$.

2.7. Mediáns keresés

Feladat: A középső elem kiválasztása.

Megjegyzés. 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.

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 kisebb-egyenlőket a B, a nagyobbakat a J csoportba tesszük. Ezután attól függően, hogy $|B|$ nagyobb-e k-nál, vagy a B rész k. elemét keressük rekurzívan, vagy a J rész $(k - |B|)$. elemét.



10. Algoritmus (k-adik elem)**.**

1. Osszuk a bemeneti tömb n elemét $\frac{n}{5}$ darab 5 elemből álló csoportra.
2. Mind az $\frac{n}{5}$ darab csoportnak keressük meg a mediánsát (ezt csoportonként 6 összehasonlítással megtehetjük, lásd a 3. példát). Legyenek ezek $b_1, b_2, ..., b_{(\frac{n}{5})}$.
3. 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.
4. Majd a bemeneti tömböt osszuk fel a mediánsok mediánsa, azaz x szerint úgy, hogy minden elemet összehasonlítunk x-szel. Legyen B a felosztás során az x-nél kisebb-egyenlő elemek halmaza, J pedig a nagyobbaké. Az algoritmus rekurzív meghívásával keressük meg a k-adik legkisebb elemet. Ha $|B| \geq k$, akkor ez az elem lesz az alsó részben (B-ben) a k. elem, különben pedig ez az elem lesz a felső részben (J-ben) a $k - |B|$. elem.

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 $O(n)$.*

2.8. 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. 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 (mint a keresési feladatokban), hogy az input csupa különböző elemből áll.

A rendezést a **GYORSR**$(A[1:n])$ hívás végzi el.

11. Algoritmus (Gyorsrendezés)**.**

GYORSR$(A[p:r])$**:**
  if $p < r$ **then**
    $q :=$ **SZÉTOSZT**$(A[p:r])$ /* *SZÉTOSZTJUK az A tömböt.*
    GYORSR$(A[p:q-1])$ /* *Rekurzívan rendezzük a balra állókat.*
    GYORSR$(A[q+1:r])$ /* *Rekurzívan rendezzük a jobbra állókat is.*

SZÉTOSZT$(A[p:r])$**:**
  $i :=$**RND**$[p,r]$ /* *Véletlenül kiválasztjuk 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 $A(j) \le x$ **then** $i{+}{+}$; **CSERE**$(A(i), A(j))$ /* *A j indexszel végigmegyünk az egész tömbön és ha a j. elem $\le x$, akkor az x-nél nagyobb elemektől balra áthelyezzük.*
  return(i) /* *Ez lesz az osztóelem indexe, $A(i)$ lesz az osztóelem.*

Példa a szétosztásra: 4. példa.

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

9. Tétel. *A gyorsrendezésben az összehasonlítások várható száma:*

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

2.9. Leszámláló rendezés

A leszámláló rendezésnél feltételezzük, hogy az n bemeneti elem mindegyike 0 és m közötti egész szám, ahol m egy nem túl nagy egész. Ez azért lesz fontos, mivel a bemenet elemeivel fogjuk a C tömböt indexelni. A leszámláló rendezés alapötlete, hogy minden egyes x bemeneti értékre meghatározza azoknak az elemeknek a számát, amelyek $\le x$. Ezzel az információval az x elemet közvetlenül a saját pozíciójába tudjuk elhelyezni a kimeneti tömbben. Ha pl. 13 olyan elem van, amely nem nagyobb az x-nél akkor az x a 13. helyen fog szerepelni. Persze ha több x értékű elem is van, akkor kicsit gondosabban kell eljárni. Tulajdonképpen bevezetünk egy új, szigorú rendezést: $a_i \prec a_j$,

ha vagy $a_i < a_j$, vagy pedig $a_i = a_j$ és $i < j$. Ha eszerint rendezünk a fenti módszerrel, akkor ráadásul stabil rendezést is kapunk, azaz az egyforma értékű elemek közül a később érkezett kerül későbbre. Ez azt jelenti, hogy a legutolsó x értékű elemet rakjuk a 13. helyre, az utolsó előtti x értékűt a 12. helyre, s.í.t.

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 $a_{i_1} \leq a_{i_2} \leq \ldots \leq a_{i_n}$, 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.*
$C[0:m]$ /* *Átmeneti munkaterület, ahol m a felső becslés az inputokra.*

12. Algoritmus (Leszámláló rendezés)**.**

LESZREND$(A[1:n])$**:**
  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: 5. példa.

2.10. Számjegyes rendezés

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

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.

13. Algoritmus (Számjegyes rendezés)**.**

for $i = k..1 \ \ (-1)$
    STABILREND(i. **számjegy szerint**)

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

Példa a számjegyes rendezésre: 6. 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: 7. példa.
Példa a kivonásra: 8. példa.
Példa a szorzásra: 9. példa.
Példa a maradékos osztásra: 10. 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 $O(n^2)$.*

14. 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 szám-

rendszerben felírva $b_{n-1}b_{n-2}\dots b_2b_1b_0$, 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 bit-műveletekben számolt műveletigény lehet nagyon nagy is, mivel egyre hosszabb számokat kell összeszoroznunk (az output maga is állhat akár 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.

15. 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 $a + b \mod m$ és $a - b \mod m$ kiszámítható $O(n)$ lépésben, $a \cdot b \mod m$ pedig kiszámítható $O(n^2)$ lépésben (egy szorzás és egy maradékos osztás m-mel).*

Az $\mathbf{a} : \mathbf{b} \mod \mathbf{m}$ 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 $a' \equiv x \cdot b' \pmod{m'}$.

16. Algoritmus (mod m hatványozás)**.**

$\mathbf{a^b} \mod \mathbf{m}$:
Az előző hatványozási algoritmust (14. 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 n bites számokkal kell számolnunk.

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

3.3. Euklideszi algoritmus

Feladat: Legnagyobb közös osztó megkeresése.

17. Algoritmus (Euklideszi algoritmus)**.**

LNKO(A, B)**:**
  if $A > B$ **then CSERE**(A, B) /* *Ezt csak egyszer kell elvégezni.*
  lnko(A, B) /* *A tényleges rekurzív algoritmus meghívása.*

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

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

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.

18. Algoritmus (Kiterjesztett euklideszi algoritmus)**.**

LNKO(A, B)**:**
  if $A > B$ **then CSERE**(A, B) /* *Ezt csak egyszer kell elvégezni.*
  lnko$(A, B, 1,0,0,1)$ /* *A tényleges rekurzív algoritmus hívása, az első argumentum előáll* $1 \cdot A + 0 \cdot B$ *alakban, a második* $0 \cdot A + 1 \cdot B$ *alakban.*
  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.

19. 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 $1 = u \cdot b + v \cdot m$. 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* $a, b \in \mathbb{N}$ *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.*

4. fejezet

Dinamikus programozás

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

Elnevezés: $P =$ „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.

20. 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.

8. 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.

21. Algoritmus (Hátizsák – csak OPT érték)**.**

for $j = 0..W$ $\quad T(j) := 0$ $\quad$ /* *$T(j)$ 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 tömegben ki lehet vinni.* */

Lépésszám: $O(n \cdot W)$

Az algoritmus lefutására lásd a 11. 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.

22. Algoritmus (Hátizsák)**.**

for $j = 0..W$ $\ T(0, j) = 0$
for $i = 1..n$
for $j = 0..W$

if $j \geq w_i$ && $T(i-1, j-w_i) + e_i > T(i-1, j)$ **then**
$T(i,j) := T(i-1, j-w_i) + e_i$
else $T(i,j) := T(i-1, j)$
return$(T(n, W))$

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 12. 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.

23. 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

9. 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.

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.

5.2. Bináris keresőfa

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

10. 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 végrehajtása 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).

24. Algoritmus (Keresés bináris keresőfában)**.**

Keresés(x, r)**:** /* *Az r gyökerű fában keressük x-et*
$v := r$
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,*
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)

25. 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)

26. 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

11. 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 műveletet fogjuk használni.

27. 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.)

28. Algoritmus (Beszúrás AVL fába)**.**

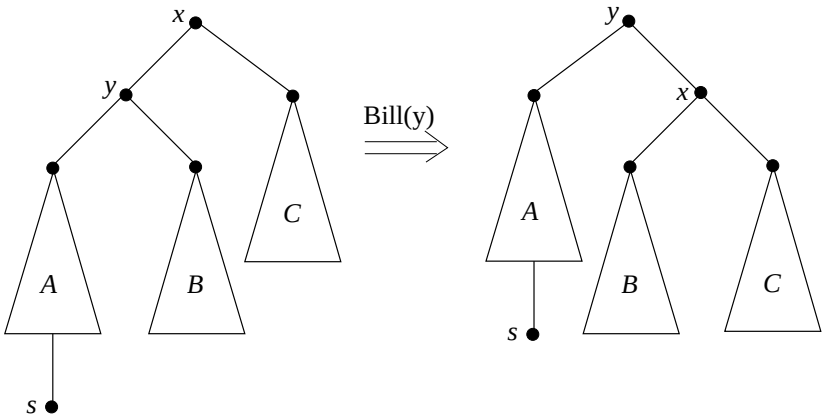
Beszúrás(s): Beszúrjuk s-et, majd felfelé haladva megkeressük a legközelebbi x ősét, ahol az AVL tulajdonság elromlott; közben persze az érintett csúcsok magasságát átállítjuk:

- ha olyan v gyerekből lépünk fel $u = p(v)$-be , melynek magassága eddig eggyel kisebb volt, mint a testvéréé, akkor megállunk, nincs elromlott csúcs,
- ha v és testvére magassága eredetileg azonos volt, akkor a Beszúrás miatt v-ét eggyel növeltük, így u-ét is kell,

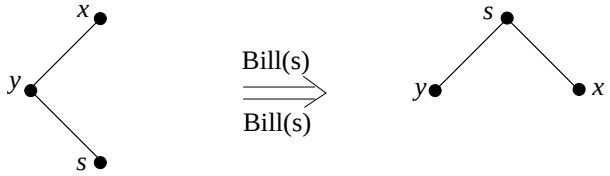
- ha pedig v magassága nagyobb volt eredetileg, mint a testvéréé, akkor $x = u$ az elromlott csúcs.

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

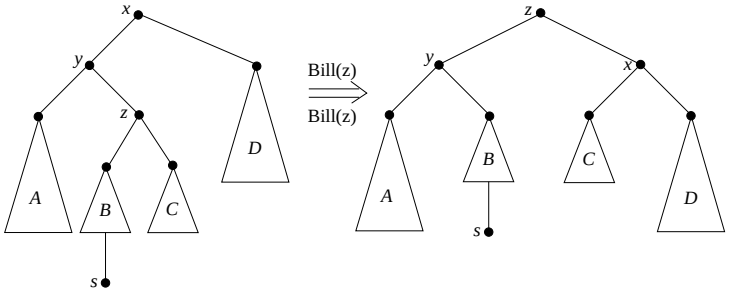
1. eset: s az x bal-bal, vagy jobb-jobb unokájának a leszármazottja



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 és 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).

12. 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(VÉGE) := v

$u \leftarrow S$:
    $u := ST$(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).

13. 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).

14. 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.

15. 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).

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

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 csúcsból 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:

1. Nem látott.
2. Látott, de még nem átvizsgált.
3. Átvizsgált.

6.1.1. Összefüggőség tesztelése szélességi kereséssel

29. Algoritmus (Szélességi keresés – összefüggőség)**.**

SZK(G)**:**
  for $i = 1..n$ $L(i) := 0$
  $L(s) := 1$; **ÚJSOR**(S, n); $S \leftarrow s$ /* *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.*
  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 $\sum_{u \in V} (d(u) + 1) = 2 \cdot m + n$, tehát $O(m + n)$, ami $O(m)$, ha nincs izolált csúcs.

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

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

6.1.2. Komponensek meghatározása

30. Algoritmus (Szélességi keresés – komponensek)**.**

SZKK(G)**:**
  $k := 0$; **ÚJSOR**(S, n); **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**.

31. Algoritmus (Szélességi keresés – legrövidebb utak)**.**

SZKL(G, s)**:**
for $i = 1..n$; $p(i) := 0$
$SZ(s) := 0$; $p(s) := s$; **ÚJSOR**(S, n); $S \leftarrow s$
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. *Legyen 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* $p(v) > 0$.

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.

32. Algoritmus (Szélességi keresés – kétszínezés)**.**

SZKkétsz(G)**:**
for $i = 1..n$; $p(i) := 0$
$SZ(1) := 0$; $p(1) := 1$; **ÚJSOR**(S, n); $S \leftarrow 1$
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.*

16. 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).
- 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 $c: E \to \mathbb{R}$ költségfüggvény az éleken.

S lesz azon csúcsok halmaza, amiket már elértünk, P pedig $V-S$ azon csúcsait tartalmazza, melyekből megy él S-be; M jelöli a maradék csúcsokat ($M = V-S-P$).

T lesz a felépítendő fa élhalmaza. A következő értékeket akarjuk meghatározni és tárolni minden $v \in P$ csúcshoz:

$K(v) = \min_{u\in S,\ uv\in E}\{c(uv)\}$

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

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 a legolcsóbb éllel.

33. Algoritmus (Prim)**.**

MINFA(G)**:**
  $P := \{1\};\ S := \emptyset;\ M := V \setminus \{1\}$
  $T := \emptyset;\ K(1) := 0;\ p(1) := 1$
  while $P \neq \emptyset$
    legyen u **a** P **halmaz minimális** $K(u)$ **kulcsú eleme**
    $S \leftarrow u \leftarrow P$ /* *Áttesszük u-t P-ből az S-be.*
    if $u \neq 1$ **then** $T := T + \{u, p(u)\}$
    for $u\underline{v} \in E$
      if $v \in P$ **&&** $c(uv) < K(v)$ **then** $K(v) := c(uv);\ p(v) := u$
      if $v \in M$ **then** $K(v) := c(uv);\ p(v) = u;\ P \leftarrow v \leftarrow M$

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 minimum-számítás esetén.

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

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-CSÖK (ez egy új kupacművelet, lásd a köv. fejezetben), az utolsó sorban egy-egy BESZÚRás kell. Tehát összesen n db MINTÖRLÉS és n db BESZÚRás, valamint m db KULCS-CSÖK szükséges. Ha az input éllistával adott, akkor a kupacműveletek dominálják a lépésszámot. Tehát az összes lépésszám éllista esetén (bináris) kupac használatával $O(m \log n)$.

6.3. Kupacok alkalmazása a Prim-algoritmusban

Feladat: Egy kupac megadott eleme kulcsának csökkentése.

34. Algoritmus (KULCS-CSÖK)**.**

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

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

23. Állítás. *A KULCS-CSÖK lépésszáma* $O(\log n)$*. Tehát Prim algoritmusának lépésszáma éllistás tárolás esetén, ha kupac-műveletekkel valósítjuk meg, akkor* $O(m \cdot \log n)$*.*

6.3.1. d-ed fokú kupac

17. Definíció. d-edfokú kupac és megvalósítása. Az $A[0 : n-1]$ tömb elemeit egy olyan fa csúcsainak feleltetjük meg, ahol minden csúcsának $\leq d$ 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, így $j > 0$-ra 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. A d-edfokú kupacban a műveletek a bináris kupac műveleteinek mintájára mennek. A legfontosabb különbség a LEBILLEGTETÉS műveletében van: itt először 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 $O(m \cdot \log_{d^*} n)$ lesz. Amennyiben $m > n^{1+\varepsilon}$ valamilyen pozitív konstans ε-ra, akkor ez $O(m)$, azaz lineáris idejű.

24. Állítás. *Prim algoritmusának lépésszáma d-edfokú kupaccal* $O(m \log_d n + dn \log_d n)$*. Ez* $d^* = \max\left(2, \lceil \frac{m}{n} \rceil\right)$ *esetén* $O(m \cdot \log_{d^*} n)$*. 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. Dijkstra algoritmusa

Először azt az esetet oldjuk meg, amikor a $c(uv)$ élsúlyok („hosszak") nemnegatívak. 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.

35. Algoritmus (Dijkstra)**.**

DIJKSTRA(G, s)**:**
$P := \{s\}$; $S := \emptyset$; $M := V \setminus \{s\}$; $K(s) := 0$; $p(s) := s$
while $P \neq \emptyset$
  legyen u **a** P **halmaz minimális** $K(u)$ **kulcsú eleme**
  $S \leftarrow u \leftarrow P$
  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 13. példán.

18. Definíció. Egy $s \rightsquigarrow v$ út S-út, ha minden csúcsa S-ben van, kivéve esetleg v-t.

25. 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.

Tehát Dijkstra algoritmusa jól számítja ki a legrövidebb utakat, lépésszáma a különböző megvalósítások esetén megegyezik Prim algoritmusának megfelelő megvalósításának lépésszámával.

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

7.2. Alkalmazás: legbiztonságosabb út

Feladat: Legbiztonságosabb út megkeresése. Például egy telekommunikációs hálózatban szeretnénk olyan összeköttetést találni, amelyre a legnagyobb valószínűséggel nem szakad meg a vonal.

Legyenek az élsúlyok a tönkremenés valószínűségei: $p(e) = \mathbb{P}($ az e él tönkremegy 1 napon belül), feltehetjük, hogy $0 \leq p(e) < 1$ minden e élre. Feltesszük, hogy különböző élek tönkremenése független események (ez nem túlságosan valósághű, de így tudjuk csak egyszerűen kezelni a feladatot). Egy P útra $\mathbb{P}(P \text{ megmarad}) = \prod_{e \in P}(1 - p(e))$, ezt szeretnénk maximalizálni.

Trükk: Vegyük a fenti kifejezés logaritmusát (a logaritmus szigorúan monoton függvény), és azt maximalizáljuk. Szorzat logaritmusa a logaritmusok összege, valamint egynél kisebb számok logaritmusa negatív.

$$\max \prod_{e \in P}(1 - p(e)) \Leftrightarrow \max \log \prod_{e \in P}(1 - p(e)) =$$

$$= \max \sum_{e \in P} \log(1 - p(e)) = -\min \sum_{e \in P}(-\log(1 - p(e))).$$

Ebben az esetben $c(e) := -\log(1 - p(e)) \geq 0$ élsúlyokkal alkalmazhatjuk Dijkstra algoritmusát.

7.3. 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 szeretnénk *maximalizálni*. Dijkstra algoritmusának megfelelő átalakításával megoldható a feladat. Három sort kell lecserélni:

legyen u **a** P **halmaz maximális** $K(u)$ **kulcsú eleme**
if $v \in P$ **& &** $\min(K(u), w(uv)) > K(v)$ **then**
$K(v) := \min(K(u), w(uv))$; $p(v) := u$
if $v \in M$ **then** $K(v) := \min(K(u), w(uv))$; $p(v) = u$; $P \leftarrow v \leftarrow M$

7.4. Házépítés – PERT módszer

7.4.1. Topologikus sorrend

19. 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 1 000 000 $-os kérdéssé alakul. 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.

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

26. 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.

36. 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** $\quad S \leftarrow u \quad$ /* *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** $\quad S \leftarrow v \quad$ /* *Ha v forrássá vált, akkor betesszük az S sorba.*
if $k < n$ **then print** „NEM ACIKLIKUS”

Lépésszám: $O(m)$

7.4.2. 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.

27. 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 idő-

ben. 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.)

37. 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: $O(m)$

7.5. Bellman–Ford-algoritmus

Feladat: Legrövidebb utak problémájának megoldása abban az általános esetben, amikor az élek között negatív súlyúakat is találhatunk, de a súlyozás konzervatív.

28. Tétel. *Ha* G *erősen összefüggő irányított gráf,* c *konzervatív,* $s \neq v \in V$*, akkor létezik legrövidebb* $s \rightsquigarrow v$ *séta. Ha* Q *egy legrövidebb* $s \rightsquigarrow v$ *séta, akkor létezik* P $s \rightsquigarrow v$ *út is, hogy* $c(P) = c(Q)$*.*

38. 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(„NEG”,**K, p, v**)**
  return(„KONZ”,K, p, s**)**

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

30. Tétel. *Ha G irányított erősen összefüggő gráf és c konzervatív, akkor Bellman–Ford algoritmusa minden csúcsra kiszámítja az oda vezető legrövidebb 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.*

8. fejezet

Hasítás (Hash-elés)

Feladat: U rendezett univerzum adott elemeiből kell olyan S szótárat létrehozni, amelyben gyorsan el tudjuk végezni a KERES és BESZÚR 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 $\mathbb{P}(h(K)=i) \approx \approx 1/M$ minden $0 \leq i < M$-re és arra az eloszlásra, amely szerint az input kulcsok érkezni fognak. Ilyen h függvény például egy jó ál-véletlenszám-generátor, melynek az U elemei megadhatók „mag"-ként.

21. Definíció. Hasító függvénynek nevezünk egy $h: U \to [0,1,\dots M-1]$ 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.

22. 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 $M < |S|^2/2$.)

Vödrös vagy láncolt hasítás. 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. BESZÚR(K) esetén is így kezdünk, és ha nem találtuk, akkor a lista végére illesztjük.

31. Tétel. *A sikeres keresés várható lépésszáma a vödrös (láncolt) hasításnál:* $1+\frac{\alpha}{2}$, *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 hasításnál: $1 + \alpha$.

23. 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.

Nyílt címzés: 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. Két fajtáját tanultuk:

Lineáris hasítás. BESZÚR(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 $\alpha < 1$).

24. Definíció. Lineáris hasítás: sorba próbáljuk a $h(K), h(K)-1, \ldots, 0, M-1, M-2, \ldots$ helyeket, és az első üres helyre rakjuk be.

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

32. Tétel. *Lineáris hasításnál*

a sikeres keresés várható lépésszáma: $\frac{1}{2} \cdot \left(1 + \frac{1}{1-\alpha}\right) = O\left(\frac{1}{1-\alpha}\right)$;

a sikertelen keresés és a beszúrás várható lépésszáma: $\frac{1}{2} \cdot \left(1 + \left(\frac{1}{1-\alpha}\right)^2\right) = O\left(\left(\frac{1}{1-\alpha}\right)^2\right)$.

Ha a táblázat 50%-ig van tele, akkor a sikeres keresés várható lépésszáma 1,5, a sikertelené 2,5, ha pedig 90%-ig, akkor a sikeres keresés lépésszáma 5,5, a sikertelené 50,5.

Dupla hasítás. Itt csinálunk egy másik, $h' : U \to [1,2,...,M-1]$ függvényt is, mely lehetőleg minél függetlenebb a h-tól, másrészt minden $K \in U$-ra $h'(K)$ relatív prím az M-hez. Ekkor a keresési sorozat: $h(K), h(K) - h'(K), h(K) - 2h'(K), h(K) - 3h'(K), \ldots, h(K) - (M-1)h'(K) \mod M$. A műveleteket ezen sorozat felhasználásával a lineáris esethez hasonlóan végezzük.

25. Definíció. Dupla hasítás: van egy szintén véletlenszerű második h' hasító függvényünk. Ekkor sorba próbáljuk a $h(K), h(K) - h'(K), h(K) - 2h'(K), \ldots$ helyeket

(mod M), és az első üres helyre rakjuk be. (Itt azt tesszük fel, hogy $K_1 \neq K_2$ esetén $\mathbb{P}(h(K_1) = h(K_2) \wedge h'(K_1) = h'(K_2)) \approx \frac{1}{M^2}$).

33. Tétel. *Dupla hasításnál*

a sikeres keresés várható lépésszáma: $\frac{1}{\alpha}\left(\ln \frac{1}{1-\alpha}\right) = O\left(\log \frac{1}{1-\alpha}\right)$*;*

a sikertelen keresés és beszúrás várható lépésszáma: $\frac{1}{1-\alpha}$.

Ha a táblázat 50%-ig van tele, akkor a sikeres keresés lépésszáma 1,39, a sikertelené 2, ha pedig 90%-ig, akkor a sikeres keresés lépésszáma 2,56, a sikertelené 10.

9. fejezet

Párosítások

9.1. Stabil párosítás páros gráfban

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

27. Definíció. Egy $fl \in E$ blokkoló (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 pár.

39. Algoritmus (Gale–Shapley)**.**

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

2. 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).

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

9.2. Maximális párosítás páros gráfban

28. 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:

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

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

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

40. Algoritmus („Magyar módszer”)**.**

INIT:

```
for i = 1..n2   M(i) := 0
for j = 1..n1   m(j) := 0
```

```
ELEJE:  ÚJSOR(L, n_1)
        for j = 1..n_1   p(j) := 0
                         if m(j) = 0 then   L ← j
                                            p(j) := j
        while L ≠ ∅      j ← L
                         for i ∈ e(j)   if M(i) = 0 then
                                              ⟶ JAVÍTÁS(i, j)
                                        if p(M(i)) = 0 then
                                              p(M(i)) := j
                                              L ← 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
                ⟶ 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 50. algoritmus, ami a nehezebb, súlyozott esetet oldja meg.

35. 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:* $\Gamma(X) = \{u \in U \mid \exists x \in X,\ xu \in E\}$*).*

36. 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 $|\Gamma(X)| \geq |X|$.*

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

38. 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

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

39. 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 $O(n^2 m)$, azóta javult.

10. fejezet

Hálózati folyamok

29. 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.

30. 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.

31. 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: $f(S, T) := \sum_{u \in S, v \in T, uv \in E} f(uv) - \sum_{u \in S, v \in T, vu \in E} f(vu)$.

40. 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.*

32. 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

41. 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: 14. példa.

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

42. 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.*

42. Algoritmus (Edmonds–Karp)**.**

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

43. 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

33. 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.

44. 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

45. 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).*

46. 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.*

34. 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.

35. 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.

47. 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.*

48. 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

36. 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 $\Leftrightarrow x \in \mathcal{L}$.

37. 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 $xz = y$.)

38. 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.

43. 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.

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

12. fejezet

A bonyolultságelmélet alapjai

A Turing-gép informális leírása a következő: van egy központi egysége, melynek véges sok *állapota* van, köztük a kitüntetett START és a két megállást jelző ELF és ELUT állapotok. Van ezenkívül egy (vagy akár több) *szalag*ja, melynek minden *mező*jében az ábécé egy eleme áll, kezdetben az input és rajta kívül az úgynevezett üresjelek. Továbbá van egy 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.

39. 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 olyan Turing-gép, ami őt eldönti.

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.

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

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

41. 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 idejű 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.

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

50. Tétel (Cook)**.** *Létezik NP-teljes nyelv.*

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

b) *Létezik-e k-nál hosszabb út: NP-teljes.*

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

d) *Hátizsák-feladat NP-teljes.*

Az 1 000 000 $-os kérdés az, hogy $P =?NP$. Ez ekvivalens azzal, hogy vajon valamelyik NP-teljes probléma a P osztályban van-e (azaz van-e rá hatékony algoritmus)?

II. rész

Következő lépés

13. fejezet

Aritmetika: számolás nagy számokkal

13.1. Nagy számok szorzása Karacuba 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 $22m$ bit-műveletet igényelnek. Ha $T(m)$-mel jelöljük két m-bites szám szorzásához szükséges bit-műveletek számát, akkor tehát $T(2m) \leq 3T(m) + 22m$.

Ebből az egyenlőtlenségből könnyen lehet felső korlátot adni a bit-műveletek 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

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

13.2. A diszkrét Fourier-transzformált

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

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

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,...,$2n-2$ 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:

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

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). Emlékezzünk vissza, hogy 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 nem-lineáris műveletek száma összesen $2n-1$.

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

Legyen $P(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1}$ 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-transzformáltjának* 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. 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-transzformáltnak hívnak) az egyik legfontosabb algoritmikus eszköz algebrai számításoknál.

44. 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 *lesz*
$Y(j) := Y0(j) + t$
$Y(j + 2^{k-1}) := Y0(j) - t$
$E := E \cdot \mathbf{eps}$
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 $T(k) = (k+1)2^k = n(\log n+1)$, 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 $O(n \log^3 n)$ bit-művelettel. (Ez a korlát sokkal ügyesebben számolva $O(n \log n \log\log n)$-re javítható.) Az ötlet az, hogy használjuk a polinomok szorzását. Legyenek $u = \overline{u_{n-1} \dots u_1 u_0}$ és $v = \overline{v_{n-1} \dots v_1 v_0}$ pozitív egészek bináris alakban. Tekintsük a következő polinomokat:

$$U(x) = u_0 + u_1 x + u_2 x^2 + \cdots + u_{n-1} x^{n-1} \quad \text{és}$$
$$V(x) = v_0 + v_1 x + v_2 x^2 + \cdots + v_{n-1} x^{n-1}.$$

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

Viszont most bit-művelettel akarunk számolni, nem aritmetikai művelettel. A kiszámított egész számok (a szorzat polinom $2n-1$ darab együtthatója) nem nagyobbak n-nél (és ez igaz minden közben kiszámított szám egész részére is), és így 2 behelyettesítése az UV polinomba maximum $O(n \log n)$ bit-műveletet 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 $2n-2=2^k$ 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 $2j\cdot 3^\ell/n^8 \le$ $\le 2\cdot 2^{k-\ell-1}\cdot 3^\ell/n^8 \le 2^{k-\ell}\cdot 3^\ell/n^8 \le 4^k/n^8 = 1/n^6$.

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 $|V(\varepsilon^j)|\le n$, 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 $O(\log^2 n)$ bit-műveletet igényel. Így összesen $O(n\log^3 n)$ bit-műveletet használtunk.

13.3. Nagy számok szorzása Schönhage és Strassen módszerével

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^n+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. Ezt vissza fogjuk vezetni a $\mathbb{Z}[x] \mod (x^K+1)$ polinom-gyűrűben való szorzásra, ahol K értéke $\sqrt{N}$ lesz (mi csak arra az egyszerűsített esetre adjuk meg az algoritmus és az elemzés vázlatát, amikor $N=2^{2^d}$, de lásd az alfejezet végén az erre vonatkozó megjegyzést). Ezt továbbá visszavezetjük az $R_n[x] \mod (x^K+1)$ polinom-gyűrűben való szorzásra, ahol a mi egyszerűsített ese-

tünkben n értéke is $\sqrt{N}$ 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 ismételjük. Az algoritmushoz szükséges algebrai állításokat csak kimondjuk, bizonyításukat az olvasóra bízzuk.

Legyen tehát $k = 2^{d-1}$ és $K = 2^k$ 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 $C(x) = \sum_{i=0}^{2K-2} c_i x^i$. Ekkor a keresett eredmény $C(2^K) \mod 2^N + 1$, ami

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

Legyen $c_{2K-1} = 0$ és jelölje $c_i^* = c_i - c_{i+K}$-t, ezeket szeretnénk meghatározni $i = 0,1,\ldots,K-1$ értékekre. Könnyű látni, hogy

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

Tehát elég meghatározni a c_i^* mennyiségeket modulo $K2^{2K}$. Legyen $d_i = c_i^* \mod K$ és $\bar{c}_i = c_i^* \mod (2^{2K}+1)$. Ezekből

$$c_i^* \mod K2^K = (2^{2K}+1)\big((d_i - \bar{c}_i) \mod K\big) + \bar{c}_i.$$

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 $O(N)$ bit-műveletet végeztünk, mivel minden szám kisebb, mint K^{3K}.

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 és 16 egy primitív K-adik egységgyök. Ezenkívül bármilyen 2-hatvánnyal gyorsan tudunk szorozni és 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 $\hat{b}_j = \sum_{i=0}^{K-1} 4^{(2j+1)i} b_i$. Ekkor

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

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 $O(N \log N)$ bit-műveletet), ezenkívül K darab rekurzív hívás R_n-ben. Tehát, ha $T(N)$ a bit-műveletek 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 könnyen adódik, hogy $T(N) = O(N \log N \log\log N)$.

Megjegyzés. Azt, hogy $N = 2^\ell$ tényleg feltehetjük, mivel N-et maximum kétszerezve ez elérhető. Ha ℓ páros, akkor működik a fenti érvelés. Ha ℓ páratlan, akkor legyen $K = 2^{(\ell+1)/2}$. Végiggondolható, hogy apróbb változtatásokkal (pl. 4 és 16 helyett 16-ot, ill. 256-ot választva) a fenti algoritmus és elemzés működik.

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:

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

- $d(a_i)$: 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 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).

45. Algoritmus (Mélységi keresés)**.**

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

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

Lépésszám: $O(n + m)$

52. Állítás. *Irányítatlan gráfban mélységi keresés után* $uv \in E \implies u$ *őse* v*-nek vagy fordítva.*
u *őse* v*-nek* $\Leftrightarrow MSZ(u) \leq MSZ(v)$ *és* $BSZ(u) \geq BSZ(v)$.

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.

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

46. 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.

47. Algoritmus (2-összefüggőség tesztelése)**.**

Minden csúcshoz két mennyiséget definiálunk:

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

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

Az első mennyiség nyilván a mélységi keresés során könnyen számolható. A második is számolható alulról felfelé, a következő rekurzív összefüggést használjuk:
$FEL(u) = \min(LEGM(u), \min\{FEL(v) \mid \ p(v) = u\})$.

Lépésszám: $O(m)$

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

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

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

15.3. Erősen összefüggő komponensek

Feladat: Input egy G irányított gráf. Def: $x \sim y$, ha van irányított út x-ből y-ba és y-ból x-be is. Ennek ekvivalencia osztályait erősen összefüggő komponenseknek nevezzük. Ezeket kell meghatározni.

48. 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:
  for $i = n..1 \ \ (-1)$
  $w := B(i)$
  if $p(w) = 0$ **then** $p(w) := w$; **MB**(w)
- A második keresés fáinak csúcshalmazai lesznek az erősen összefüggő komponensek.

Lépésszám: $O(m)$

16. fejezet

Legrövidebb utak

16.1. Disjkstra algoritmusának további alkalmazásai

16.1.1. Legszélesebb, ill. legolcsóbb legrövidebb út

Adott egy G irányított gráf, és élein két különböző pozitív függvény, egy c hosszúság és egy w szélesség vagy költség. Adott s csúcsból keresünk adott t csúcsba olyan utat, mely egyrészt egy legrövidebb út, másrészt a legrövidebb utak közül a lehető legszélesebb vagy legolcsóbb.

Elsőként keressünk a Dijkstra-algoritmussal s-ből minden más csúcsba legrövidebb utat, ezáltal kapunk $d(s, v)$ távolságokat. Utána keressünk a fordított gráfban t-ből minden más csúcsba legrövidebb utat, ezáltal kapunk az eredeti gráfban $d(v, t)$ távolságokat. Készítünk egy G' segédgráfot, mely azon uv élekből áll, amelyekre $d(s, u) + c(uv) + d(v, t) = d(s, t)$.

55. Állítás. *A G' gráf aciklikus, melyben s az egyetlen forrás és t az egyetlen nyelő. G' tartalmazza a G gráf összes $s \rightsquigarrow t$ legrövidebb útját. Továbbá a G' gráfban minden $s \rightsquigarrow t$ út egy legrövidebb út G-ben, azaz hossza $d(s, t)$.*

Tehát, ha a G' gráfban keresünk s-ből t-be egy legolcsóbb vagy legszélesebb utat a 7.1. illetve a 7.3. alfejezetben leírtak szerint, az pont a célunknak megfelelő út lesz.

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 út szélessége 100. Ekkor, ha a G' segédgráfba bevesszük az összes olyan élet, amelynek szélessége legalább 90,

és G'-ben keresünk legrövidebb utat, akkor pont egy ilyen feladatot oldunk meg.

16.1.2. 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 $a_{uv}(i)$ függvény, amely azt adja meg, hogy ha az u csúcsból az i időpontban indulunk, akkor mikor érkezünk meg a v csúcsba. Természetesen $a_{uv}(i) \geq i$.

Megjegyzések. 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.

A legkorábbi érkezési időket szeretnénk kiszámítani, ha adott a kiindulási s csúcs, és az indulás i_0 időpontja. Erre 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. Floyd–Warshall-algoritmus

Egy sűrű gráfban minden csúcsból minden csúcsba legrövidebb utat keresni 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.

49. Algoritmus (Floyd–Warshall)**.**

for $k = 1..n$
  for $i = 1..n$
    for $j = 1..n$
      $M(i,j) := \min[M(i,j), M(i,k) + M(k,j)]$

56. Állítás. *A fenti algoritmus $O(n^3)$ lépésben megkeresi a legrövidebb út hosszát minden egyes csúcspár között, ha c konzervatív. A c súlyozás akkor és csak akkor konzervatív, ha az algoritmus végén a főátlóban nem jelenik meg negatív szám.*

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 29. 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 $d_{i+1}(v) = \min(d_i(u) + c(uv) \mid uv \in E)$.

57. Tétel (Karp)**.** *A minimális átlagú kör hossza*

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

Megjegyzések. Itt nincs szükség arra, hogy a hossz-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. Johnson algoritmusa

Minden csúcsból minden csúcsba keressük a legrövidebb utat. 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, először egy Bellman–Ford, majd n db Dijkstra kell:

- Futtassunk le egy Bellman–Ford-algoritmust, és $\pi(v) := K(v)$ jelölje az általa adott potenciált minden v csúcsra.
- Módosítsuk az élek súlyát: $c'(uv) := c(uv) + \pi(u) - \pi(v)$ minden uv élre. (Ezután $c'(uv) \ge 0$.)

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

16.5. Suurballe algoritmusa

Adott G erősen összefüggő irányított gráf nemnegatív c élsúlyokkal. Keresünk s-ből t-be egy P és egy Q utat, amik páronként 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. Az első fázis 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 a 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 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 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.

Ezt algoritmust később Suurballe Tarjannal közösen úgy implementálta, hogy a teljes futási idő annyi legyen, mint egy Dijkstra-algoritmus futási ideje a d-edfokú kupaccal.

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.

58. 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.*

59. Tétel (Hopcroft–Karp)**.** *Az algoritmus $2\sqrt{n}$ fázis után véget ér.*

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 $2\sqrt{n}$ csúcsa van.

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.

60. 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 egész-értékű, akkor π is választható egész-értékűnek.*

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 40. algoritmus leállásakor visszaadott O Ore-halmaz az S maximális hiányú halmazainak a metszete; ezt az algoritmust fogjuk szubrutinként hívogatni a G_π segédgráfokra.

50. Algoritmus (Magyar módszer, Khun)**.**

Készítsük el a G_π segédgráfot, és hívjuk meg rá a 40. 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.

61. 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.

51. 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.

62. 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 (41. 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, akár a legnagyobb kapacitással arányos lépésszám is kellhet. Ezt valamennyire kijavítja az Edmonds–Karp-algoritmus (lásd a 42. 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 41. 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:

63. 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.*

64. 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 algo-*

ritmusa lefut az egyszerű megvalósítással $O(n^2m)$, 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 41. 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.

65. 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 7.4. 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.

66. 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.

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 $1 \leq i \leq$ $\leq k$-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-értékeinek összege, $d_G(X)$ pedig az X-ből kilépő G-beli élek kapacitás-értékeinek 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 négyzet, 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.

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

Ez könnyen következik az alábbi tételből:

68. 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

Közelítő algoritmusok

19.1. 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.

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 polinomja 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 $1/\varepsilon$ 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. 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.

19.2. Metrikus utazó ügynök

Metrikus utazó ügynök feladat: Adott egy G teljes gráf és az éleken olyan költségek, hogy $c(uv) + c(vw) \geq c(uw)$ minden u, v, w csúcs-hármasra. Keresünk egy minimális költségű Hamilton-kört.

Megjegyzés. Ha találunk egy olyan c^* költségű Q körsétát, mely minden csúcsot érint, akkor ebből könnyen kaphatunk egy C Hamilton-kört, melynek költsége $c(C) \leq c(Q) = c^*$. Ugyanis induljunk Q mentén, és amikor olyan

csúcsba lépnénk, ahol már jártunk, akkor ugorjunk rögtön a séta mentén az első olyan csúcsba, ahol még nem jártunk. A háromszög-egyenlőtlenség miatt ettől nem lesz hosszabb a sétánk.

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

Mivel az optimális Hamilton-kör bármely élét elhagyva egy feszítőfát kapunk, 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.

52. 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 (az X által feszített részgráfban) minimális súlyú M 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, és ebből a fenti módszerrel kaphatunk egy nem hosszabb Hamilton-kört.

Nem nehéz belátni, hogy $c(M) \leq \frac{\mathrm{OPT}}{2}$.

19.3. FPTAS a hátizsák-feladatra

Visszatérünk az NP-nehéz hátizsák-feladatra. Először is megjegyezzük, hogy kis érték esetén egy másik dinamikus programozási algoritmus is működik, ami most kényelmesebb lesz nekünk. 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 növekvő sorrendbe rendezzük, veszünk az elejéről annyi tárgyat, amennyi belefér a hátizsákba, majd a következő tárgyból levágunk egy akkora részt, hogy éppen kitöltse a zsákot.

53. Algoritmus (Hátizsák – kis E értékre)**.**

for $j = 0..E$ $T(j) := W + 1$ /* *$T(j)$ lesz a legkisebb teherbírású hátizsák, amiben j érték kivihető; $W+1$ a végtelen megfelelője.* */
  for $i = 1..n$ /* *Az i. ciklusban az első i tárgyból keresünk.* */
    for $j = E..e_i$ (-1)
      if $T(j - e_i) + w_i < T(j)$ **then** $T(j) := T(j - e_i) + w_i$
  for $j = E..0$ (-1) **if** $T(j) \leq W$ **then return** $T(j)$

Ennek az $O(E \cdot n)$ idejű algoritmusnak a segítségével megadjuk Ibarra és Kim FPTAS algoritmusát. Először is feltehetjük, hogy minden tárgy magában belefér a hátizsákba, azaz minden i-re $w_i \leq W$ (a többi tárgy törölhető).

Legyen $E = \sum e_i$ és M egy később meghatározandó szám. Definiálunk egy segédfeladatot, amelyben w_i és W ugyanaz, de az értékek $e'_i = \lfloor \frac{e_i}{M} \rfloor$. 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 (pontosabban azzal a kiterjesztéssel, amely a tárgyhalmazt is meghatározza), 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.

$$\mathrm{MO} = \sum_{i \in I'} e_i \geq M \cdot \sum_{i \in I'} e'_i \geq M \cdot \sum_{i \in I} e'_i \geq \sum_{i \in I} (e_i - M) \geq \mathrm{OPT} - nM.$$

A második egyenlőtlenség azért teljesül, mert I' volt az optimálisan kivihető tárgyhalmaz a segédfeladatra. Az a célunk, hogy $\mathrm{MO} \geq (1 - \varepsilon) \cdot \mathrm{OPT}$ teljesüljön, ehhez az kell, hogy $nM \leq \varepsilon \cdot \mathrm{OPT}$ teljesüljön. Ehhez megbecsüljük OPT értékét: a feltételezésünk szerint $\mathrm{OPT} \geq \max_i e_i \geq E/n$. Tehát válasszuk M értékét $M = \dfrac{\varepsilon \cdot E}{n^2}$-nek, ekkor az iménti elvárásunk teljesül.

A dinamikus programozási algoritmus lépésszáma pedig legfeljebb $O\left(\dfrac{E \cdot n}{M}\right) = O\left(\dfrac{n^3}{\varepsilon}\right)$.

19.4. 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 (39. algoritmus) ilyenkor 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 lesz 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.

69. Tétel. *Ez az algoritmus mindig egy* 3/2*-közelítő stabil párosítást ad.*

19.5. Halmazfedés

Adott az $S = \{1,2,\dots,n\}$ alaphalmaznak m db részhalmaza: $A_1, A_2, \dots, A_m$, 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.

54. Algoritmus (Mohó közelítés halmazfedésre)**.**

$C := \emptyset$
while $C \neq S$
$\alpha := \min_i\left(\frac{c(A_i)}{|A_i - C|}\right)$
$j := \operatorname{argmin}_i\left(\frac{c(A_i)}{|A_i - C|}\right)$
for $s \in A_j - C$ $\quad \operatorname{price}(s) := \alpha$
$C := C \cup A_j$

70. Állítás. *A* k*-adiknak lefedett* s_k *elemre* $\operatorname{price}(s_k) \leq \frac{\mathrm{OPT}}{n-k+1}$*, ezért a mohó algoritmus* H_n*-közelítő, ahol* $H_n = \sum_{i=1}^{n} \frac{1}{i} \approx \ln n$*.*

Most tegyük fel, hogy a halmazrendszerben a maximális fok Δ. Oldjuk meg tetszőleges polinomiális módszerrel a következő LP-feladatot:

$$\sum_{j:\ s \in A_j} x_j \geq 1, \quad \forall s \in S,$$

$$x_j \geq 0, \quad \forall 1 \leq j \leq m,$$

$$\min \sum_j c(A_j) \cdot x_j.$$

Ezután az optimális törtmegoldást kikerekítjük: y_j legyen 1, ha $x_j \geq 1/\Delta$ és 0 különben.

71. 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.

19.6. Beck–Fiala-tétel

72. Tétel (Beck–Fiala)**.** *Adott egy hipergráf az $\{1,2,\dots,n\}$ alaphalmazon, melynek élei $\{A_1, A_2, \dots, A_m\}$, és tegyük fel, hogy minden pont foka legfeljebb d, és $d > 2$. Ekkor létezik $c : \{1,2,\dots,n\} \mapsto \{-1,+1\}$ kétszínezése az alaphalmaznak, hogy minden A_j élben $|\sum_{i\in A_j} c(i)| \le 2d-3$.*

Algoritmikus bizonyítást adunk. $\mathcal{J}$ jelölje az aktív élek indexeinek halmazát, kezdetben ez $\{1,2,\dots,m\}$. 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 sorban 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 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.

20. fejezet

Fix paraméteres algoritmusok

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 példa-feladatok 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 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 (klikk) feladatra ilyet nem tudunk, és nem is igazán várható, a másodikra viszont igen.

20.1. Steiner-fa

Az első 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 a 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 (Dreyfus-tól és Wagnertől származó) dinamikus programozási algoritmus FPT. Jelölje $\mathrm{OPT}(S', u)$ annak a feladatnak az optimális megoldását, amely az $S' \cup \{u\}$ terminálhalmazra vesz egy ezt feszítő fát, ahol u 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 $\mathrm{OPT}(S', u)$ értékeket az alábbi állítást felhasználva:

73. Állítás.

$$\mathrm{OPT}(S', v) = \min_{u \in V} \min_{\emptyset \neq S_1 \subset S'} \big(\mathrm{OPT}(S_1, u) + \mathrm{OPT}(S' - S_1, u) + d(u, v)\big).$$

74. Állítás. *A Dreyfus–Wagner-algoritmus lépésszáma*

$$O(3^k n^2 + nm + n^2 \log n).$$

20.2. 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.

20.2.1. Első megoldás k-korlátos mélységű döntési fával

Rekurzív eljárást adunk meg: ha nincs él, készen vagyunk, az eddig megjelölt csúcsok lefogók. Különben, ha 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, ha ez sikertelenül tér vissza, akkor jelöljük meg v-t, hagyjuk el a rá illeszkedő éleket, és hívjuk meg az eljárást rekurzívan. Ha ez is sikertelenül tér vissza, térjünk mi is vissza sikertelenül.

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 $O(2^k n)$.

20.2.2. Második megoldás k^2 méretű kernellel

Egy G inputú k paraméterű feladat *kernel*ének hívjuk a polinom időben elkészíthető 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.

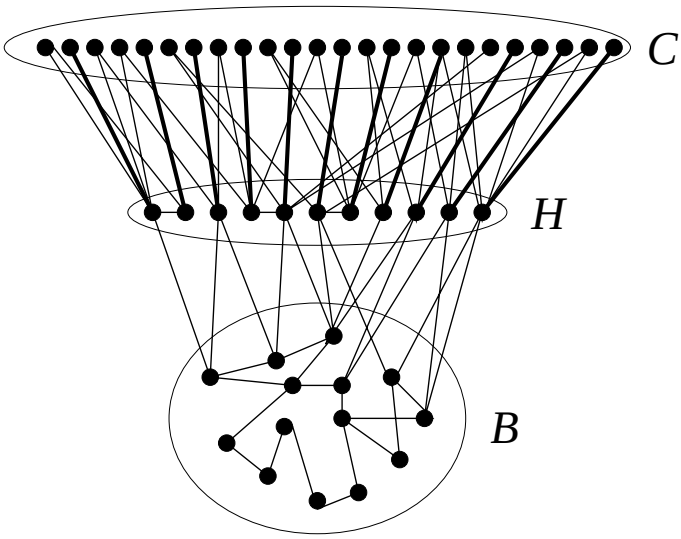
A kernelkészítő eljárásunk egész egyszerű: ha van olyan v csúcsunk, melynek foka nagyobb, mint k, akkor ezt megjegyezzük (neki nyilván benne kell lennie a lefogó csúcshalmazban, ha az legfeljebb k elemű), kitöröljük G-ből és k-t eggyel csökkentjük. Ezt ismételjük (ha k negatívvá válik, leállunk NEM válasszal), 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 $k'(k'+1)$ csúcsa van, akkor nyilván az élek nem foghatóak le k' db legfeljebb k' fokú csúccsal.

Egyébként pedig (mivel $k' \leq k$), egy legfeljebb $k^2 + k$ csúcsú gráfról kell döntenünk, 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 $O(2^k k^2 + m + n)$.

20.2.3. Harmadik megoldás 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.



20.1. ábra. Korona-felbontás

75. Állítás. *A G gráfban akkor és csak akkor létezik k elemű lefogó csúcshalmaz, ha a $G - H - C$ 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 $C \subseteq I$ álljon a H M^*-szerinti párjaiból, B pedig a maradék csúcsokból. Ez nyilván jó korona-felbontás, 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 $O(k^2m)$ idő. Utána a kernelben az első alfejezet algoritmusa $O(k \cdot 2^k)$ idő alatt végez.

20.3. Egzakt út

El akarjuk dönteni, hogy egy $G = (V, E)$ gráfban van-e olyan $s \rightsquigarrow t$ út, melynek pontosan k belső csúcsa van. Erre itt egy randomizált algoritmust adunk, amelyet azonban lehet derandomizálni.

Először is tegyük fel, hogy $V - s - t$ csúcsai ki vannak véletlen módon színezve az $C = \{1,2,\ldots,k\}$ színekkel. Egy utat szín-teljesnek hívunk, ha belső pontjai páronként különböző színűek, és C minden eleme szerepel színként. Persze ha találunk egy szín-teljes utat, akkor az pontosan k belső csúccsal fog rendelkezni. Tehát a randomizált algoritmusunk egyszerűen abból áll, hogy sorsolunk sok független színezést, és mindegyikhez keresünk szín-teljes utat. Ha egyszer is találunk, akkor az jó út lesz, ha soha, akkor azt válaszoljuk, hogy nincs k belső csúccsal rendelkező út.

76. Állítás. *Ha $240e^k$ független véletlen színezés egyikére sincs szín-teljes $s \rightsquigarrow t$ út, akkor annak valószínűsége, hogy G-ben van pontosan k belső csúccsal rendelkező $s \rightsquigarrow t$ út legfeljebb e^{-240}.*

Szín-teljes 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.

III. rész

Függelék

21. 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 a q. 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 $A > B \mid\mid T(i) = 0$ **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 $j = n..1 \;\; (-1)$
        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 $\underline{uv} \in E$
        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* $i > 0$ *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.*

22. fejezet

Példák

3. 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.

4. 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

5. 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)

6. 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

7. Példa (Összeadás)**.**

$$\begin{array}{r} 10110010 \\ +101100100 \\ \hline 1000010110 \end{array}$$

8. Példa (Kivonás)**.**

$$\begin{array}{r} 1000010110 \\ -101100100 \\ \hline 10110010 \end{array}$$

9. Példa (Szorzás)**.**

10110010 · 101
101100100
10110010
1101111010

10. Példa (Maradékos osztás)**.**

1101101 : 101 = 10101
111
1001
100 (ez a maradék)

11. 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.**

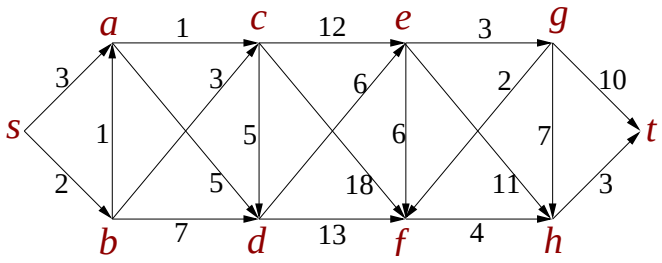
Mivel már nem lépünk be újból semelyik ciklusba se, ezért $T(W) = 6$**. Tehát a maximálisan kivihető érték** 6**.**

12. Példa (Mátrix-szorzás zárójelezésére)**.** Az $A \cdot B \cdot C \cdot D$ szorzást kell elvégeznünk, ahol a mátrixok méretei sorban: $1000 \times 10, 10 \times 1, 1 \times 1000, 1000 \times \times 10$.

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: $1000 \cdot 10 \cdot 1 + 1 \cdot 1000 \cdot 10 + 1000 \cdot 1 \cdot 10 = 30000$.

13. Példa (Dijkstra-algoritmusra)**.**
Az alábbi gráfban keressük s-ből t-be a legrövidebb utat.

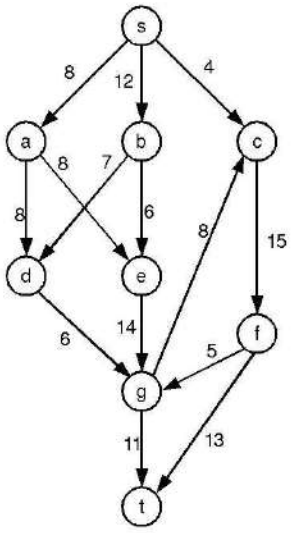


22.1. ábra. (Feladat a Dijkstra-algoritmushoz)

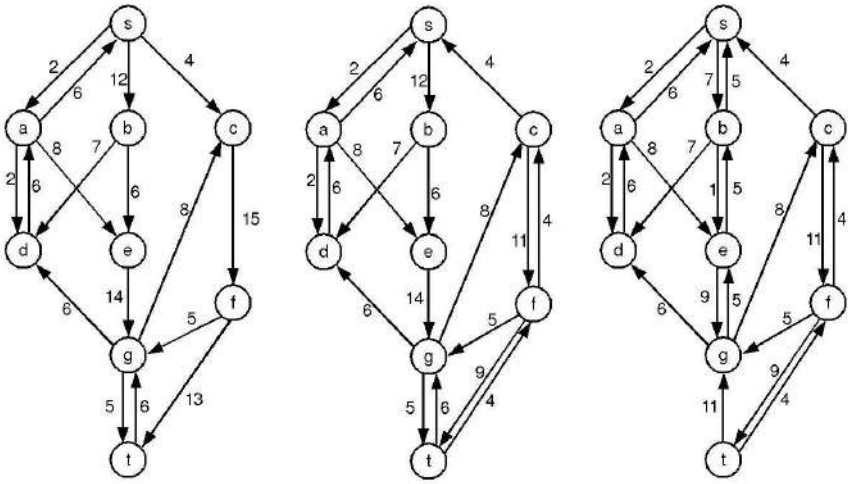
S-be kerülő csúcs és címkéje	P-beli csúcsok, ahol címke változik
$s : 0$	$a : 3$ $b : 2$
$b : 2$	$c : 5$ $d : 9$
$a : 3$	$c : 4$ $d : 8$
$c : 4$	$e : 16$ $f : 22$
$d : 8$	$e : 14$ $f : 21$
$e : 14$	$g : 17$ $h : 25$ $f : 20$
$g : 17$	$t : 27$ $h : 24$ $f : 19$
$f : 19$	$h : 23$
$h : 23$	$t : 26$
$t : 26$	

Tehát a legrövidebb út hossza 26, egy legrövidebb út: $s, b, a, d, e, g, f, h, t$.

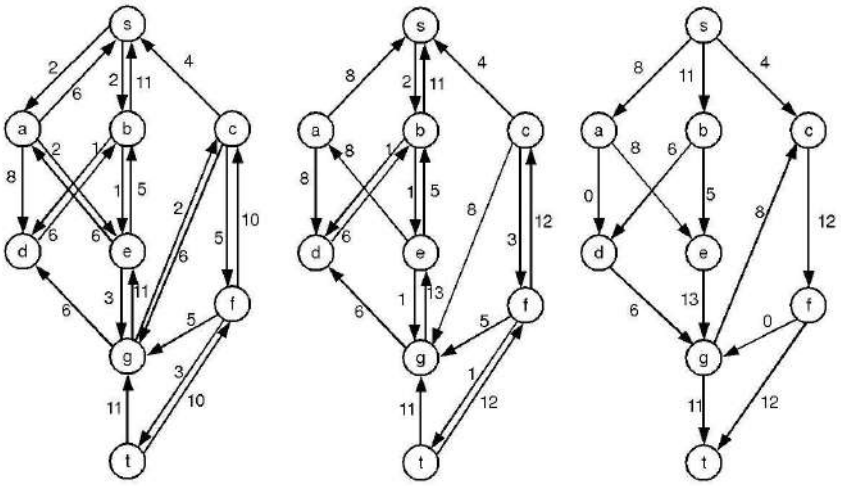
14. Példa (Ford–Fulkerson algoritmusának működésére)**.** Az alábbi hálózaton keresünk maximális folyamot:
1. lépés: $\forall e : f(e) := 0$
2. lépés: az összes él telítetlen, és előre-él G'-ben ($G' = G$).
$\forall e : r(e) := c(e) - f(e)$
3. lépés: $s \rightsquigarrow t$ út keresése G'-n: $P : s, a, d, g, t$.
4. lépés: Javít(P): $\Delta = 6$
$f(sa) := 6, f(ad) := 6, f(dg) := 6, f(gt) := 6$
5. lépés: új G' elkészítése: 22.3. ábra.
6. lépés: $s \rightsquigarrow t$ út keresése G'-n: $P : s, c, f, t$.
7. lépés: Javít(P): $\Delta = 4$



22.2. ábra. (G,c,s,t)



22.3. ábra. 5., 8. és 11. lépések



22.4. ábra. 14. és 17. lépések, és az elkészült folyam

$f(sc) := 4, f(cf) := 4, f(ft) := 4$
8. lépés: új G' elkészítése: 22.3. 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: 22.3. ábra.
12. lépés: $s \rightsquigarrow t$ út keresése G'-n: $P : s, b, d, a, e, g, c, f, t$.
13. lépés: Javít(P): $\Delta = 6$
$f(sb) := 11, f(bd) := 6, f(da) := 0, f(ae) := 6,$
$f(eg) := 11, f(gc) := 6, f(cf) := 10, f(ft) := 10$
14. lépés: új G' elkészítése: 22.4. ábra.
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: 22.4. 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 22.4. ábrán látható, folyamértéke: $|f| = 8 + 11 + 4 = 23$. Ellenőrzés: $c(S, T) = 4 + 8 + 11 = 23$.

23. 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.
- Niedermeyer: Invitation to fixed-parameter algorithms, Oxford University Press, 2006.