Algoritmusok és adatszerkezetek II.
előadásjegyzet: Fák

Ásványi Tibor – asvanyi@inf.elte.hu

2023. szeptember 11.

Tartalomjegyzék

Hivatkozások

[1] Ásványi Tibor, Algoritmusok és adatszerkezetek II.
Útmutatások a tanuláshoz, Tematika, fák, gráfok, mintaillesztés, tömörítés
http://aszt.inf.elte.hu/~asvanyi/ad/ad2jegyzet/

[2] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.,
magyarul: Új Algoritmusok, *Scolar Kiadó*, Budapest, 2003.
ISBN 963 9193 90 9
angolul: Introduction to Algorithms (Third Edititon),
The MIT Press, 2009.

[3] Fekete István, Algoritmusok jegyzet
http://ifekete.web.elte.hu/

[4] Rónyai Lajos – Ivanyos Gábor – Szabó Réka, Algoritmusok,
TypoTEX Kiadó, 1999. ISBN 963 9132 16 0
https://www.tankonyvtar.hu/hu/tartalom/tamop425/2011-0001-526_ronyai_algoritmusok/adatok.html

[5] Weiss, Mark Allen, Data Structures and Algorithm Analysis,
Addison-Wesley, 1995, 1997, 2007, 2012, 2013.

[6] Carl Burch, Ásványi Tibor, B+ fák
http://aszt.inf.elte.hu/~asvanyi/ad/ad2jegyzet/B+ fa.pdf

[7] Ásványi Tibor, Algoritmusok és adatszerkezetek I. előadásjegyzet (2021)
http://aszt.inf.elte.hu/~asvanyi/ad/ad1jegyzet/ad1jegyzet.pdf

[8] Wirth, N., Algorithms and Data Structures,
Prentice-Hall Inc., 1976, 1985, 2004.
magyarul: Algoritmusok + Adatstruktúrák = Programok, *Műszaki Könyvkiadó*, Budapest, 1982. ISBN 963 10 3858 0

1. AVL fák

Az AVL fák kiegyensúlyozásának szabályai megtalálhatók az 1-6. ábrákon.

Az *AVL fák* magasság szerint kiegyensúlyozott bináris keresőfák. Egy bináris fa *magasság szerint kiegyensúlyozott*, ha minden csúcsa kiegyensúlyozott. (Mostantól *kiegyensúlyozott* alatt *magasság szerint kiegyensúlyozottat* értünk.) Egy bináris fa egy $(*p)$ csúcsa *kiegyensúlyozott*, ha a csúcs $(p \to b)$ egyensúlyára (balance) $|p \to b| \leq 1$. A $(*p)$ csúcs egyensúlya definíció szerint:

$$p \to b = h(p \to right) - h(p \to left)$$

Az AVL fákat láncoltan reprezentáljuk. A csúcsokban a b egyensúly attribútumot expliciten tároljuk. (Egy másik lehetőség a két részfa magasságainak tárolása lenne; egy harmadik, hogy csak az aktuális részfa magasságát tároljuk.) A Node osztályt tehát a következőképpen módosítjuk:

Node
+ $key : \mathcal{T}$ // $\mathcal{T}$ is some known type + $b : -1..1$ // the balance of the node + $left, right$: Node*
+ Node() { $left := right := \oslash$; $b := 0$ } // create a tree of a single node + Node(x:$\mathcal{T}$) { $left := right := \oslash$; $b := 0$; $key := x$ }

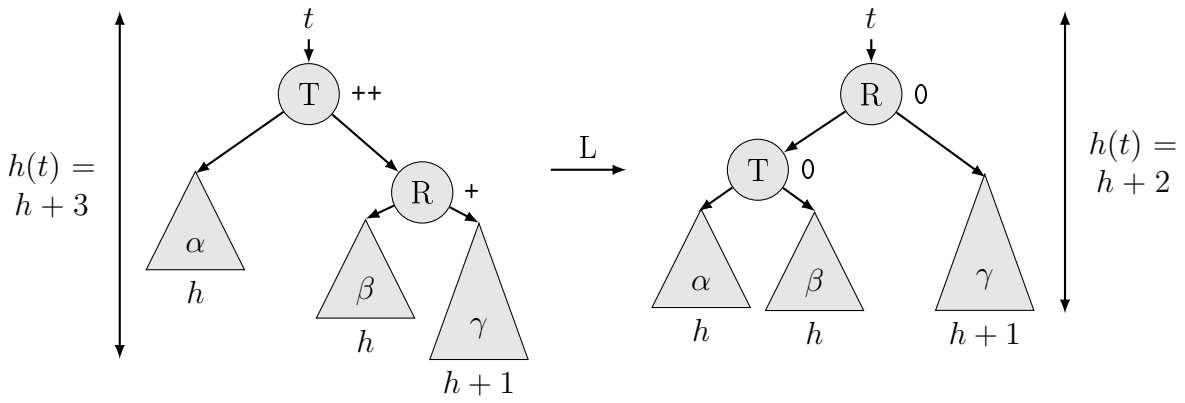
Egyelőre részletes bizonyítás nélkül közöljük az alábbi eredményt:

Tétel: Tetszőleges n csúcsú nemüres AVL fa h magasságára:

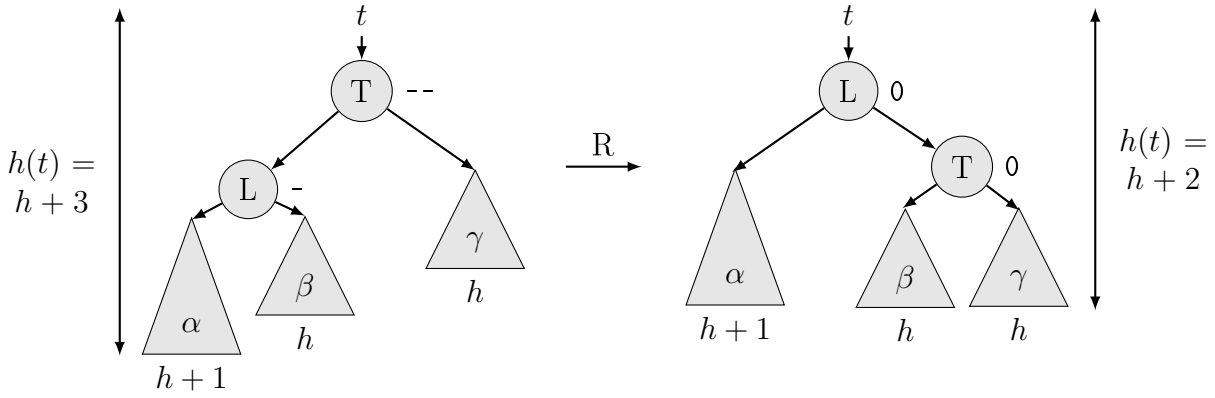
$$\lfloor \log n \rfloor \leq h \leq 1,45 \log n, \quad \text{azaz} \quad h \in \Theta(\log n)$$

A bizonyítás vázlata: Először a h magasságú, nemüres KBF-ek (kiegyensúlyozott, bináris fák) n méretére adunk alsó és felső becslést. Az $n < 2^{h+1}$ becslésből azonnal adódik $\lfloor \log n \rfloor \leq h$. Másrészt meghatározzuk a h mélységű, legkisebb méretű KBF-ek csúcsainak f_h számát. Erre kapjuk, hogy $f_0 = 1$, $f_1 = 2$, $f_h = 1 + f_{h-1} + f_{h-2} \quad (h \geq 2)$. Ezért az ilyen fákat *Fibonacci fáknak hívjuk.* Mivel tetszőleges h magasságú KBF n méretére $n \geq f_h$, némi matematikai ügyességgel kaphatjuk a $h \leq 1,45 \log n$ egyenlőtlenséget.

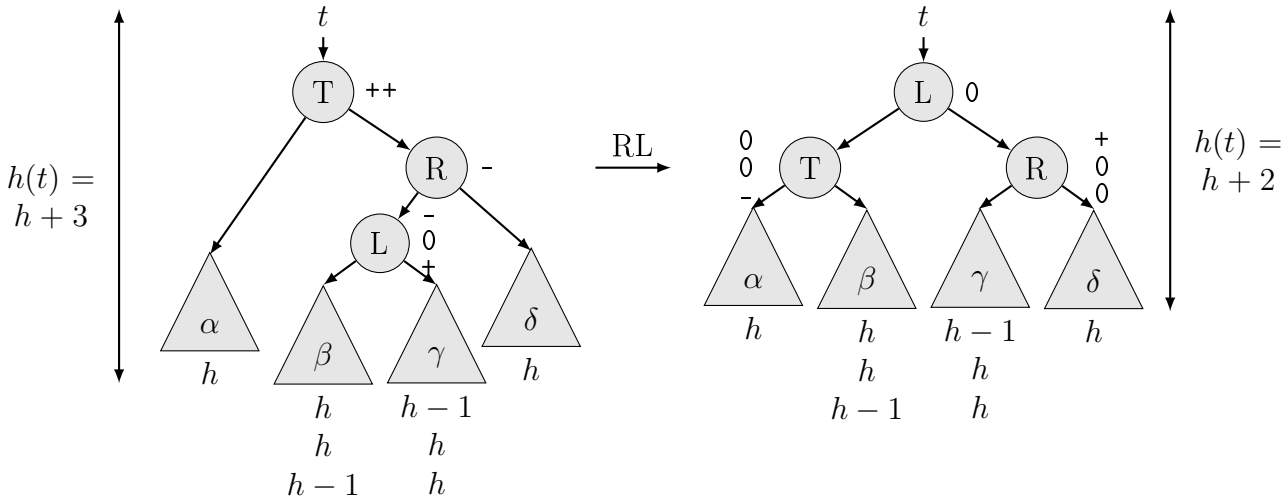
Mivel az AVL fák magassága $\Theta(\log n)$, ezért a bináris keresőfák search(t, k), min(t) és max(t) függvényeire t AVL fa esetén automatikusan
$MT(n) \in \Theta(\log n)$, ahol $n = |t|$.



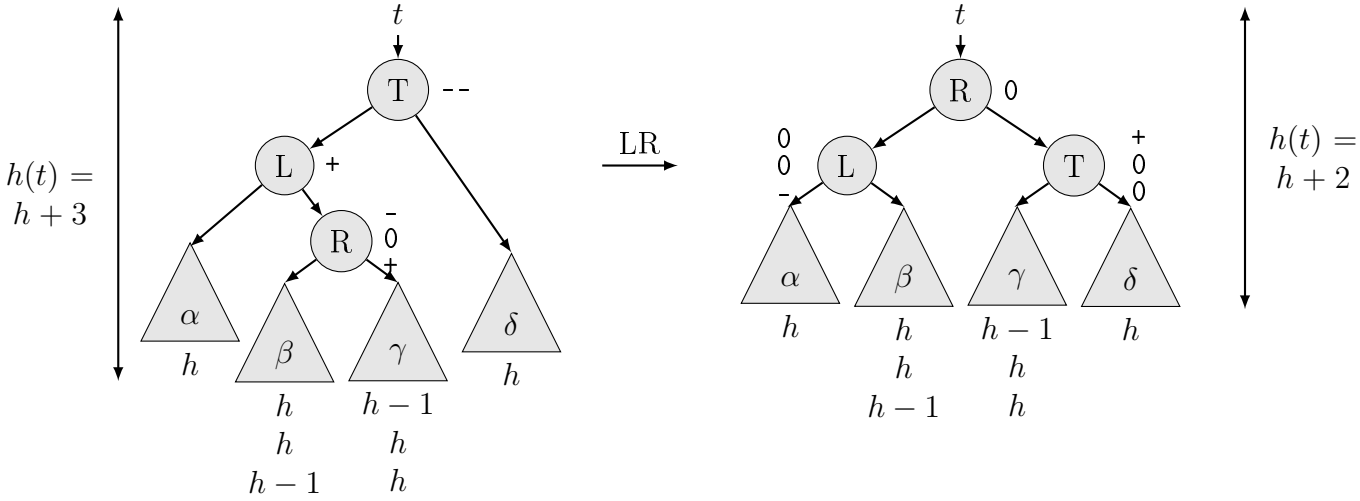
1. ábra. $(++, +)$ forgatás.



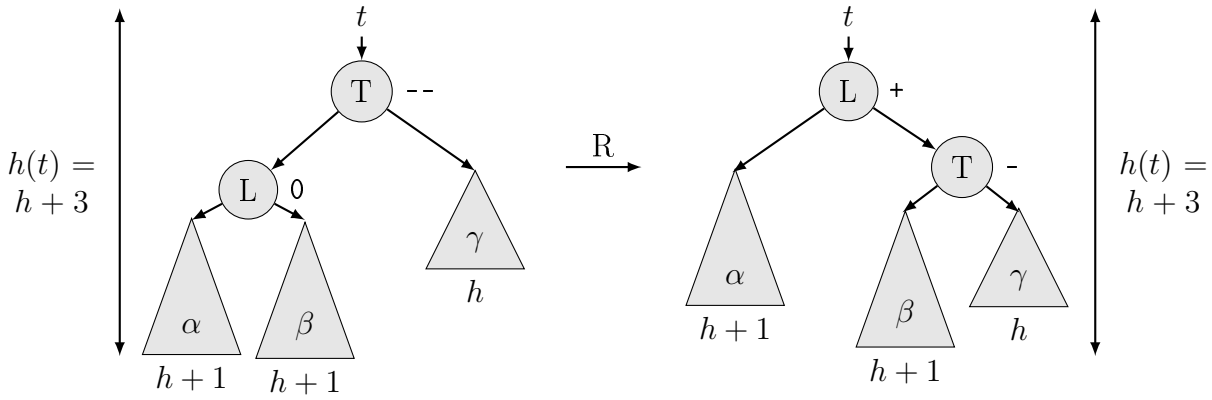
2. ábra. $(--, -)$ forgatás.



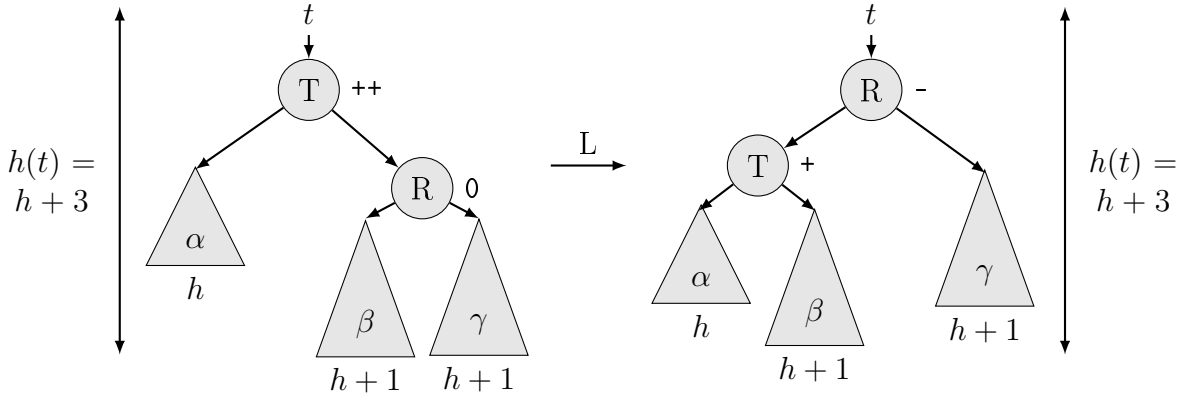
3. ábra. $(++, -)$ forgatás.



4. ábra. $(--, +)$ forgatás.



5. ábra. $(--, 0)$ forgatás.



6. ábra. $(++, 0)$ forgatás.

Az insert(t, k), a del(t, k), a remMin$(t, minp)$ és a remMax$(t, maxp)$ eljárások azonban változtatják a fa alakját. Így elromolhat a kiegyensúlyozottság, és már nem garantált a fenti műveletigény. Ennek elkerülésére ezeket az eljárásokat úgy módosítjuk, hogy minden egyes rekurzív eljáráshívás után ellenőrizni fogjuk, hogyan változott a megfelelő részfa magassága, és ez hogyan befolyásolta a felette levő csúcs kiegyensúlyozottságát, szükség esetén helyreállítva azt. Ez minden szinten legfeljebb konstans mennyiségű extra műveletet fog jelenteni, és így a kiegészített eljárások futási ideje megtartja az $MT(n) \in \Theta(\log n)$ (ahol $n = |t|$) nagyságrendet.

A példákhoz a *(bal_részfa gyökér jobb_részfa)* jelölést használjuk, ahol az üres részfákat elhagyjuk, és a könnyebb olvashatóság kedvéért [] és {} zárójeleket is alkalmazunk.

Például a {[2]4[(6)8(10)]} bináris keresőfa gyökere a 4; bal részfája a [2], ami egyetlen levélcsúcsból (és annak üres bal és jobb részfáiból) áll; jobb részfája a [(6)8(10)], aminek gyökere a 8, bal részfája a (6), jobb részfája a (10). *Az így ábrázolt fák inorder bejárása a zárójelek elhagyásával adódik.* A belső csúcsok egyensúlyait *kb* formában jelöljük, ahol k a csúcsot azonosító kulcs, b pedig a csúcs egyensúlya, a 0:○, 1:+, 2:++, -1:−, -2:−− megfeleltetéssel. Mivel a levélcsúcsok egyensúlya mindig nulla, a leveleknél nem jelöljük az egyensúlyt. A fenti AVL fa például a megfelelő egyensúlyokkal a következő: {[2]4+[(6)8○(10)]}

Az AVL fák műveletei során a kiegyensúlyozatlan részfák kiegyensúlyozásához *forgatásokat* fogunk használni. Az alábbi forgatási sémákban görög kisbetűk jelölik a részfákat (amelyek minden esetben AVL fák lesznek), latin nagybetűk pedig a csúcsok kulcsait (1. és 2. ábrák, az egyensúlyok nélkül):

Balra forgatás (Left rotation): $[\alpha\ \mathrm{T}\ (\beta\ \mathrm{R}\ \gamma)] \to [(\alpha\ \mathrm{T}\ \beta)\ \mathrm{R}\ \gamma]$
Jobbra forgatás (Right rotation): $[(\alpha\ \mathrm{L}\ \beta)\ \mathrm{T}\ \gamma] \to [\alpha\ \mathrm{L}\ (\beta\ \mathrm{T}\ \gamma)]$

Vegyük észre, hogy a fa inorder bejárását egyik forgatás sem változtatja, így a bináris keresőfa tulajdonságot is megtartják. Mint látni fogjuk, a kiegyensúlyozatlan részfák kiegyensúlyozása minden esetben egy vagy két forgatásból áll. *Ezért a bináris keresőfa tulajdonságot a kiegyensúlyozások is megtartják.*

1.1. AVL fák: beszúrás

Nézzük most először a bináris keresőfák insert(t, k) eljárását!

A {[2]4+[(6)8○(10)]} AVL fából
az 1 beszúrásával az {[(1)2−]4○[(6)8○(10)]} AVL fa adódik,
a 3 beszúrásával pedig az {[2+(3)]4○[(6)8○(10)]} AVL fa.

A fenti {[2]4+[(6)8∘(10)]} AVL fából
a 7 beszúrásával azonban a {[2]4++[(6+{7})8−(10)]} fát kapjuk,
a 9 beszúrásával pedig a {[2]4++[(6)8+({9}10−)]} fát.
Mindkettő bináris keresőfa, de már nem AVL fa, mivel a 4++ csúcs nem kiegyensúlyozott.

A legutolsó esetben a bináris keresőfa könnyen kiegyensúlyozható, ha meggondoljuk, hogy az a következő sémára illeszkedik, amiben görög kisbetűk jelölik a kiegyensúlyozott részfákat, latin nagybetűk pedig a csúcsok kulcsait:
[α T++ (β R+ γ)], ahol T=4, α=[2], R=8, β=(6) és γ=({9}10−).

A kiegyensúlyozáshoz szükséges transzformáció a fa *balra forgatása* (1. ábra):

$$[\ \alpha\ \mathrm{T}{+}{+}\ (\beta\ \mathrm{R}{+}\ \gamma)\ ] \to [\ (\alpha\ \mathrm{T}{\circ}\ \beta)\ \mathrm{R}{\circ}\ \gamma\ ]$$

A fenti példán ez a következőt jelenti:
{[2]4++[(6)8+({9}10−)]} → { [(2)4∘(6)] 8∘ [(9)10−] }

A transzformáció helyességének belátásához bevezetjük a $h = h(\alpha)$ jelölést. Ebből a kiinduló fára a T++ és R+ egyensúlyok miatt
$h((\beta\ \mathrm{R}\ \gamma)) = h{+}2$, $h(\gamma) = h{+}1$ és $h(\beta) = h$ adódik, innét pedig az eredmény fára $h(\alpha) = h = h(\beta)$ miatt T∘; továbbá $h((\alpha\ \mathrm{T}{\circ}\ \beta)) = h + 1 = h(\gamma)$ miatt R∘ adódik.

Az eredeti {[2]4+[(6)8∘(10)]} AVL fába a 7 beszúrásával adódó
{ [2]4++[(6+{7})8−(10)] } kiegyensúlyozatlan bináris keresőfa kiegyensúlyozásával pedig {[(2)4−] 6∘ [(7)8∘(10)]} adódik, amire az

$$\{\ \alpha\ \mathrm{T}{+}{+}\ [(\beta\ \mathrm{L}{-}{\circ}{+}\ \gamma)\ \mathrm{R}{-}\ \delta]\ \} \to \{\ [\alpha\ \mathrm{T}{\circ}{\circ}{-}\ \beta]\ \mathrm{L}{\circ}\ [\gamma\ \mathrm{R}{+}{\circ}{\circ}\ \delta]\ \}$$

séma (3. ábra) szolgál, ahol α, β, γ és δ AVL fák, és a három jelből álló egyensúlyok sorban három esetet jelentenek úgy, hogy a bal oldalon az i-edik alternatívának a jobb oldalon is az i-edik eset felel meg ($i \in \{1; 2; 3\}$).

A fenti séma a példában T=4, α = [2], R=8, δ = (10), L=6, + egyensúllyal (harmadik eset), $\beta = \oslash$ és $\gamma = \{7\}$ helyettesítéssel alkalmazható.

A transzformációt *kettős forgatásnak* is tekinthetjük: Az
{ α T [(β L γ) R δ] } fára először az R csúcsnál alkalmaztunk egy jobbra forgatást, aminek eredményeképpen az { α T [β L (γ R δ)] } bináris keresőfát kaptuk, majd az eredmény fát balra forgattuk, ami az { [α T β] L [γ R δ] } AVL fát eredményezte.

A kettős forgatás helyességének ellenőrzéséhez kiszámítjuk az eredmény fa egyensúlyait. Most is bevezetjük a $h = h(\alpha)$ jelölést. Mivel a kettős forgatás

előtti fában T++, azért $h([(\beta\ \mathrm{L}\ \gamma)\ \mathrm{R}\ \delta]) = h+2$. Innét R− miatt $h((\beta\ \mathrm{L}\ \gamma)) = h+1$ és $h(\delta) = h$. Most L lehetséges egyensúlyai szerint három eset van.
(1) L− esetén $h(\beta) = h$ és $h(\gamma) = h-1 \Rightarrow$ az eredmény fában T○ és R+.
(2) L○ esetén $h(\beta) = h$ és $h(\gamma) = h \Rightarrow$ az eredmény fában T○ és R○.
(3) L+ esetén $h(\beta) = h-1$ és $h(\gamma) = h \Rightarrow$ az eredmény fában T− és R○.
Mindhárom esetben $h([\alpha\ \mathrm{T}\ \beta]) = h+1 = h([\gamma\ \mathrm{R}\ \delta])$, így L○.
Ezzel beláttuk a kettős forgatási séma helyességét.

Vegyük észre, hogy a fentiek alapján, az L csúcsnak a kettős forgatás előtti egyensúlyát s-sel, a T és a R csúcsoknak pedig a kettős forgatás utáni egyensúlyát rendre s_t-vel, illetve s_r-vel jelölve a következő összefüggéseket írhatjuk fel:

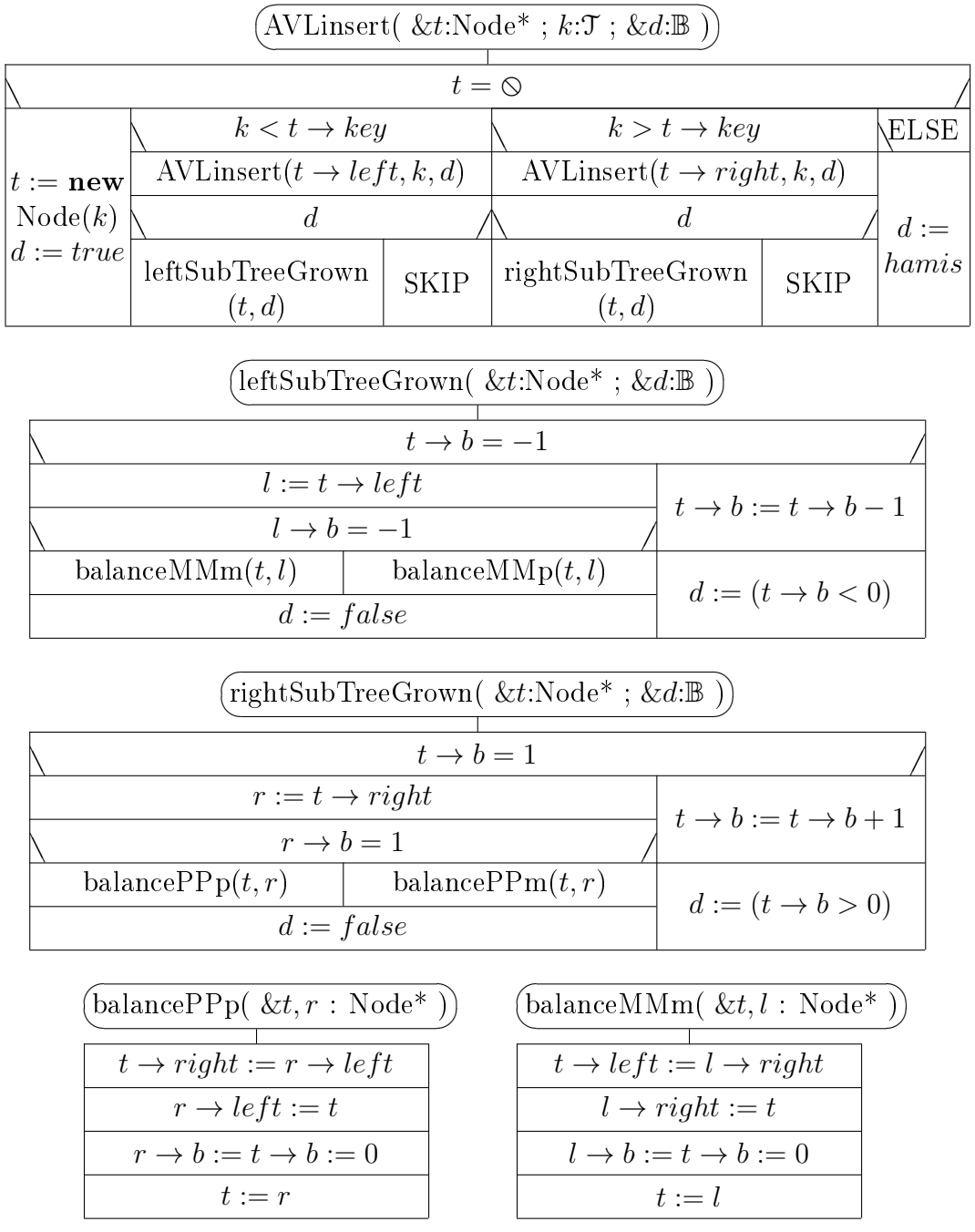
$$s_t = -\lfloor (s+1)/2 \rfloor \quad \text{és} \quad s_r = \lfloor (1-s)/2 \rfloor$$

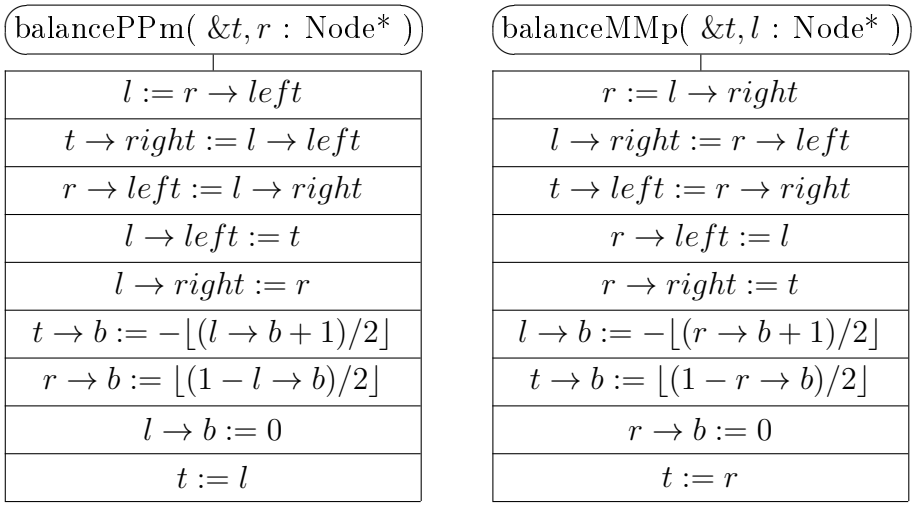
Az eddigi két kiegyensúlyozási séma mellett természetes módon adódnak ezek tükörképei, a forgatások előtt a T−− csúccsal a gyökérben. Ezek tehát a következők (2. és 4. ábra):

$$[\ (\alpha\ \mathrm{L}-\ \beta)\ \mathrm{T}--\ \gamma\ ] \rightarrow [\ \alpha\ \mathrm{L}\circ\ (\beta\ \mathrm{T}\circ\ \gamma)\ ]$$
$$\{\ [\alpha\ \mathrm{L}+\ (\beta\ \mathrm{R}-\circ+\ \gamma)]\ \mathrm{T}--\ \delta\ \} \rightarrow [\ (\alpha\ \mathrm{L}\circ\circ-\ \beta)\ \mathrm{R}\circ\ (\gamma\ \mathrm{T}+\circ\circ\ \delta)\ ]$$
$$s_l = -\lfloor (s+1)/2 \rfloor \quad \text{és} \quad s_t = \lfloor (1-s)/2 \rfloor$$

ahol s az R csúcs kettős forgatás előtti egyensúlya; s_l, illetve s_t pedig rendre az L és T csúcsoknak a kettős forgatás utáni egyensúlya

Eddig nem beszéltünk arról, hogy az AVL fában az új csúcs beszúrása után mely csúcsoknak és milyen sorrendben számoljuk újra az egyensúlyát, sem hogy mikor ütemezzük be a kiegyensúlyozást. Csak a beszúrás nyomvonalán visszafelé haladva kell újraszámolnunk az egyensúlyokat, és csak itt kell kiegyensúlyoznunk, ha kiegyensúlyozatlan csúcsot találunk. Így a futási idő a fa magasságával arányos marad, ami AVL fákra $O(\log n)$. Most tehát részletezzük a beszúró és kiegyensúlyozó algoritmus működését. Ennek során a fa magassága vagy eggyel növekszik ($d = true$), vagy ugyanannyi marad ($d = false$).


AVLinsert(&t:Node* ; k:ℑ ; &d:𝔹)
t = ⊘
t := new Node(k)
d := true
k < t → key
AVLinsert(t → left, k, d)
d
leftSubTreeGrown (t, d)
SKIP
k > t → key
AVLinsert(t → right, k, d)
d
rightSubTreeGrown (t, d)
SKIP
ELSE
d := hamis
leftSubTreeGrown(&t:Node* ; &d:𝔹)
t → b = −1
l := t → left
l → b = −1
balanceMMm(t, l)
balanceMMp(t, l)
d := false
t → b := t → b − 1
d := (t → b < 0)
rightSubTreeGrown(&t:Node* ; &d:𝔹)
t → b = 1
r := t → right
r → b = 1
balancePPp(t, r)
balancePPm(t, r)
d := false
t → b := t → b + 1
d := (t → b > 0)
balancePPp(&t, r : Node*)
t → right := r → left
r → left := t
r → b := t → b := 0
t := r
balanceMMm(&t, l : Node*)
t → left := l → right
l → right := t
l → b := t → b := 0
t := l



Az AVL fába való beszúrást röviden összefoglalva:
1. Megkeressük a kulcs helyét a fában.
2. Ha a kulcs benne van a fában, STOP.
3. Ha a kulcs helyén egy üres részfa található, beszúrunk az üres fa helyére egy új, a kulcsot tartalmazó levélcsúcsot. Ez a részfa eggyel magasabb lett.
4. Ha a gyökércsúcsnál vagyunk, STOP. Különben egyet fölfelé lépünk a keresőfában. Mivel az a részfa, amiből fölfele léptünk, eggyel magasabb lett, az aktuális csúcs egyensúlyát megfelelőképp módosítjuk. (Ha a jobb részfa lett magasabb, hozzáadunk az egyensúlyhoz egyet, ha a bal, levonunk belőle egyet.)
5. Ha az aktuális csúcs egyensúlya 0 lett, akkor az aktuális csúcshoz tartozó részfa alacsonyabb ága hozzánőtt a magasabbikhoz, tehát az aktuális részfa most ugyanolyan magas, mint a beszúrás előtt volt, és így egyetlen más csúcs egyensúlyát sem kell módosítani: STOP.
6. Ha az aktuális csúcs új egyensúlya 1 vagy -1, akkor előtte 0 volt, ezért az aktuális részfa magasabb lett eggyel. Ekkor a 4. ponttól folytatjuk.
7. Ha az aktuális csúcs új egyensúlya 2 vagy -2[1], akkor a hozzá tartozó részfát ki kell egyensúlyozni. A *kiegyensúlyozás után az aktuális részfa visszanyeri a beszúrás előtti magasságát*, ezért már egyetlen más csúcs egyensúlyát sem kell módosítani: STOP.

Az az állítás, hogy ebben az algoritmusban a *kiegyensúlyozás után az aktuális részfa visszanyeri a beszúrás előtti magasságát*, még igazolásra vár. Azt az esetet nézzük meg, amikor a kiegyensúlyozandó részfa gyökere T++. A T−−

[1] A 2 és -2 eseteket a struktogramban nem számoltuk ki expliciten, hogy az egyensúly tárolására elég legyen két bit.

eset hasonlóan fontolható meg. A T++ esethez tartozó kiegyensúlyozási sémák (1. és 3. ábra):

$$[\ \alpha\ \mathrm{T}{+}{+}\ (\beta\ \mathrm{R}{+}\ \gamma)\ ] \to [(\alpha\ \mathrm{T}{\circ}\ \beta)\ \mathrm{R}{\circ}\ \gamma]$$
$$\{\ \alpha\ \mathrm{T}{+}{+}\ [(\beta\ \mathrm{L}{-}{\circ}{+}\ \gamma)\ \mathrm{R}{-}\ \delta]\ \} \to \{\ [\alpha\ \mathrm{T}{\circ}{\circ}{-}\ \beta]\ \mathrm{L}{\circ}\ [\gamma\ \mathrm{R}{+}{\circ}{\circ}\ \delta]\ \}$$

Először belátjuk, hogy ha a T csúcs jobboldali R gyereke + vagy − súlyú, akkor a fenti sémák közül a megfelelő alkalmazható: Mivel a beszúró algoritmus a fában a beszúrás helyétől egyesével fölfele lépked, és az első kiegyensúlyozatlan csúcsnál azonnal kiegyensúlyoz, ez alatt nincs kiegyensúlyozatlan csúcs, azaz az α, β és γ, illetve a δ részfák is kiegyensúlyozottak, ez pedig éppen a fenti forgatások feltétele, amellett, hogy bináris keresőfát akarunk kiegyensúlyozni, amit viszont a kiegyensúlyozás nélküli beszúró algoritmus garantál.

Most még be kell látni, hogy a fenti sémák minden esetet lefednek, azaz R+ vagy R−: egyrészt, R nem lehet a beszúrás által létrehozott új csúcs, mert különben T-nek a beszúrás előtti jobboldali részfája üres lett volna, tehát most nem lehetne T++. Másrészt, ha a fölfele lépkedés során nulla egyensúlyú csúcs áll elő, akkor a fölötte levő csúcsok egyensúlya már nem módosul, így kiegyensúlyozatlan csúcs sem állhat elő. Márpedig most T++. Így tehát az új csúcstól T-ig fölfelé vezető úton minden csúcs, azaz R egyensúlya is + vagy −.

Most belátjuk, hogy a kiegyensúlyozások visszaállítják a részfa beszúrás előtti magasságát. A T++ kiegyensúlyozatlan csúcs a beszúrás előtt kiegyensúlyozott volt. Mivel a beszúrás, a beszúrás helyétől kezdve T-ig fölfelé vezető úton mindegyik részfa magasságát pontosan eggyel növelte, a beszúrás előtt T+ volt. Ezért a beszúrás előtt, $h = h(\alpha)$ jelöléssel, a T gyökerű részfa $h+2$ magas volt. A beszúrás után tehát T++ lett, és így a T gyökerű részfa $h+3$ magas lett. A beszúrás utáni, de még a kiegyensúlyozás előtti állapotot tekintve a továbbiakban megkülönböztetjük a T jobboldali R gyerekére a R+ és a R− eseteket.

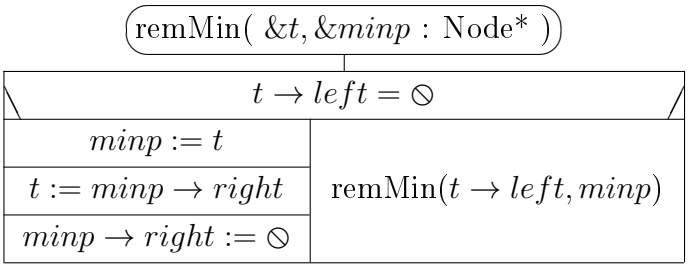
R+ esetén már láttuk, hogy $h(\alpha) = h = h(\beta)$ és $h(\gamma) = h+1$. Ezért a kiegyensúlyozás után $h([(\alpha\ \mathrm{T}\ \beta)\ \mathrm{R}\ \gamma]) = h+2$.

R− esetén pedig már láttuk, hogy $h(\alpha) = h = h(\delta)$ és $h([\beta\ \mathrm{L}\ \gamma]) = h+1$. Ezért $h(\beta), h(\gamma) \leq h$. Így a kiegyensúlyozás után $h(\{[\alpha\ \mathrm{T}\ \beta]\ \mathrm{L}\ [\gamma\ \mathrm{R}\ \delta]\}) = h+2$.

Ezzel beláttuk, hogy a kiegyensúlyozások mindkét esetben visszaállítják a részfa beszúrás előtti magasságát, mégpedig úgy, hogy a beszúrás által eggyel megnövelt magasságot eggyel csökkentik. Szimmetria okokból ez hasonlóan látható be T−− esetén is, figyelembe véve a L− és a L+ eseteket.

1.2. AVL fák: a remMin$(t, minp)$ eljárás

Bináris keresőfákra a remMin eljárás:

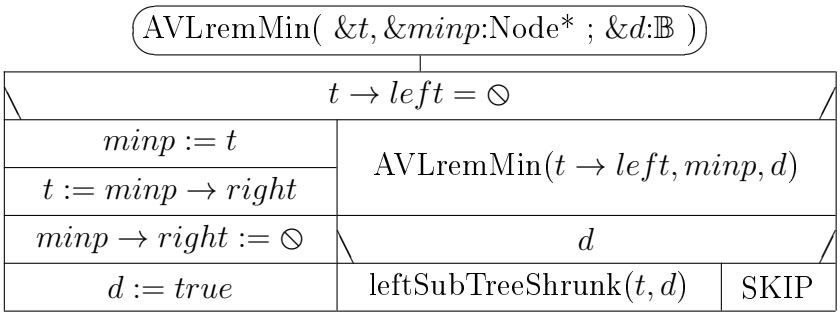


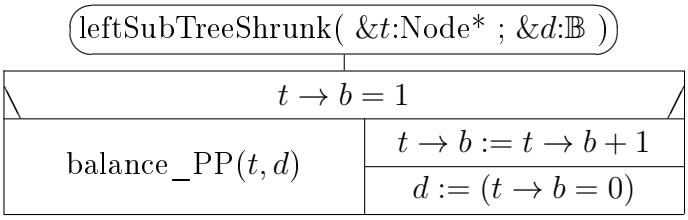
Ezt például a { [2]4+[(6)8∘(10)] } AVL fára alkalmazva, a $minp \to key = 2$ eredmény mellett, a { 4++[(6)8∘(10)] } kiegyensúlyozatlan bináris keresőfa adódna. Látjuk, hogy a kiegyensúlyozatlan 4++ csúcs jobboldali gyereke a 8∘. Ennek az esetnek a kezelésére új kiegyensúlyozási sémát kell alkalmaznunk. Szerencsére egy balra forgatás, a megfelelő egyensúly beállítások mellett most is megoldja a problémát (6. ábra):

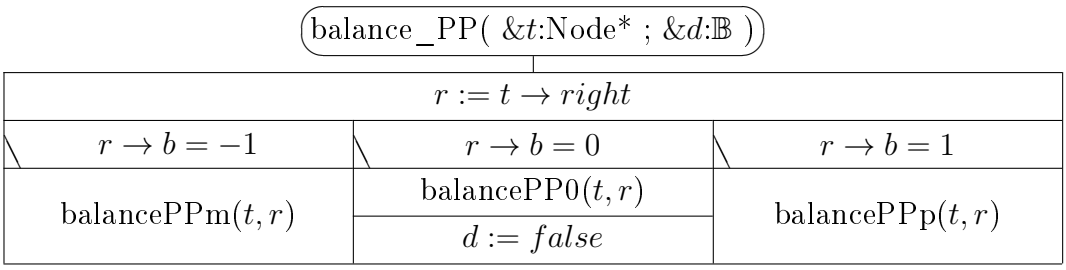
$$\{\ \alpha\ \mathrm{T{+}{+}}[\beta\ \mathrm{R}\circ\ \gamma]\ \} \to \{\ [\alpha\ \mathrm{T{+}}\ \beta]\ \mathrm{R}{-}\ \gamma\ \}.$$

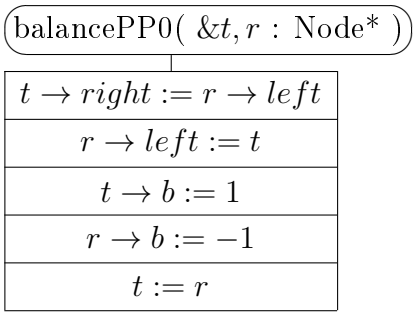
T++ miatt $h = h(\alpha)$ jelöléssel nyilván $h(\beta) = h + 1 = h(\gamma)$, így a forgatás után az egyensúlyokat a fenti séma szerint kell beállítani. A fa magassága a forgatás előtt és után is $h + 3$ lesz, ez a fajta kiegyensúlyozás tehát az eddigiekkel szemben *nem* csökkenti az aktuális részfa magasságát.

Ezután az AVL fákra alkalmazott, a megfelelő kiegyensúlyozásokkal kiegészített AVLremMin eljárás pl. a következő lehet (d most azt jelöli, hogy csökkent-e az aktuális részfa magassága):









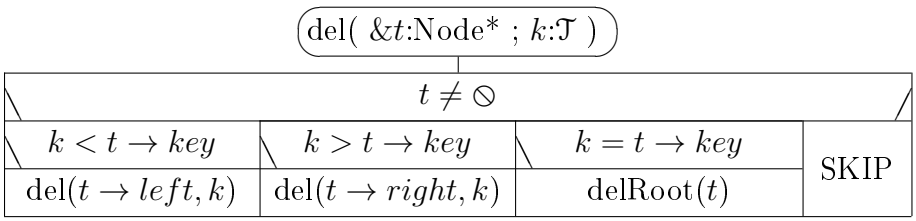
Az algoritmus helyessége hasonlóan gondolható meg, mint a beszúrás esetén. Lényeges különbség azonban, hogy ott minden beszúrást csak egyetlen kiegyensúlyozás követ, míg itt előfordulhat, hogy a minimális csúcs eltávolítása után, minden felette lévő szinten ki kell egyensúlyozni.

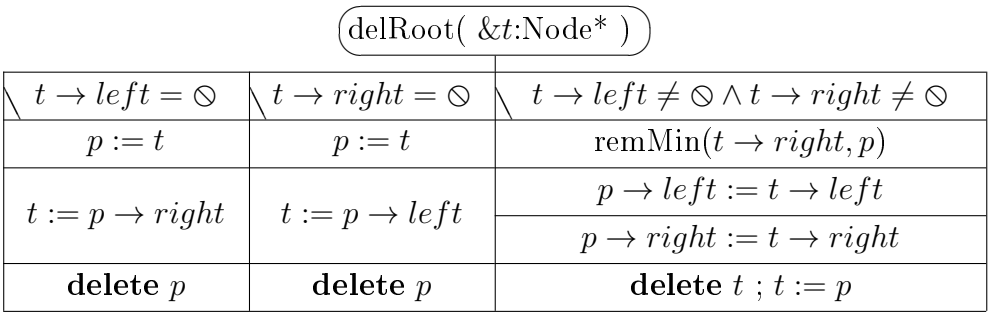
1.1. Feladat. *Mutassunk ilyen, legalább négy magasságú fát!*

1.2. Feladat. *Írjuk meg az AVLremMin(t, minp, d) eljárás mintájára az AVLremMax(t, maxp, d) eljárást és segédeljárásait!*

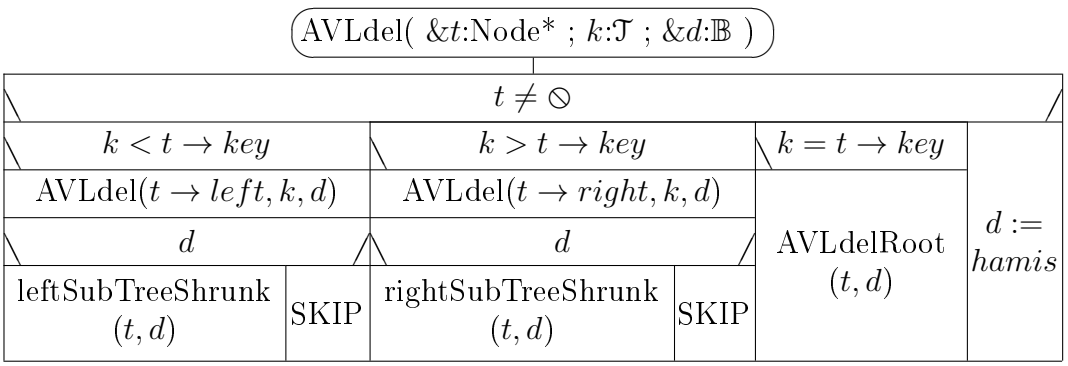
1.3. AVL fák: törlés

Bináris keresőfákra a del(t, k) és a delRoot(t) eljárások:





Ezeket fogjuk most az AVLremMin$(t, minp, d)$ eljárás mintájára kiegészíteni a d paraméterrel; majd a rekurzív töröl hívásokból, valamint az AVLremMin eljárásból való visszatérés után megkérdezzük, csökkent-e az aktuális részfa magassága. Ha igen, az AVLremMin$(t, minp, d)$ eljárás mintájára módosítjuk a $t \to b$ értéket, ha kell, kiegyensúlyozunk, és ha kell, d-t beállítjuk.



AVLdelRoot(&t:Node* ; &d:$\mathbb{B}$)

$t \to left = \oslash$	$t \to right = \oslash$	$t \to left \neq \oslash \wedge t \to right \neq \oslash$	
$p := t$	$p := t$	rightSubTreeMinToRoot(t, d)	
$t := p \to right$	$t := p \to left$		
delete p	**delete** p	d	
$d := true$	$d := true$	rightSubTreeShrunk(t, d)	SKIP

rightSubTreeMinToRoot(&t:Node* ; &d:$\mathbb{B}$)

AVLremMin$(t \to right, p, d)$
$p \to left := t \to left$; $p \to right := t \to right$; $p \to b := t \to b$
delete t ; $t := p$

Itt is lehetséges, hogy több szinten, a legrosszabb esetben akár minden szinten is ki kell egyensúlyozni. Mivel azonban egyetlen kiegyensúlyozás sem tartalmaz se rekurziót, se ciklust, és ezért konstans számú eljáráshívásból áll, ez sem itt, sem az AVLremMin eljárásnál nem befolyásolja a futási időnek az AVLinsert eljárásra is érvényes $MT(n) \in \Theta(\log n)$ (ahol $n = |t|$) nagyságrendjét.

1.3. Feladat. *A leftSubTreeShrunk(t, d) eljárás mintájára dolgozzuk ki a rightSubTreeShrunk(t, d) eljárást, ennek segédeljárásait, és az ehhez szükséges kiegyensúlyozási sémát (megoldás: 5. ábra)! Mutassuk be az AVLdel(t, k) eljárás működését néhány példán! Mutassunk olyan, legalább négy magasságú fát és kulcsot, amelyre az AVLdel(t, k) eljárás minden, a fizikailag törölt csúcs feletti szinten kiegyensúlyozást végez!*

1.4. Feladat. *Írjuk meg az AVLdel(t, k) eljárás egy olyan változatát, amely abban az esetben, ha a k kulcsot olyan belső csúcs tartalmazza, aminek két gyereke van, ezt a törlendő csúcsot a bal részfája legynagyobb kulcsú csúcsával helyettesíti!*

1.4. Az AVL fák magassága*

Tétel: Tetszőleges n csúcsú nemüres AVL fa h magasságára:

$$\lfloor \log n \rfloor \leq h \leq 1,45 \log n$$

Bizonyítás:

Az $\lfloor \log n \rfloor \leq h$ egyenlőtlenség bizonyításához elég azt belátni, hogy ez tetszőleges nemüres bináris fára igaz. Egy tetszőleges bináris fa nulladik (gyökér) szintjén legfeljebb $2^0 = 1$ csúcs van, az első szintjén maximum $2^1 = 2$ csúcs, a második szintjén nem több, mint $2^2 = 4$ csúcs. Általában, ha az i-edik szinten 2^i csúcs van, akkor az $i+1$-edik szinten legfeljebb 2^{i+1} csúcs, hiszen minden csúcsnak maximum két gyereke van. Innét egy h mélységű bináris fa csúcsainak n számára $n \leq 2^0 + 2^1 + 2^2 + \cdots + 2^h = 2^{h+1} - 1 < 2^{h+1}$. Innét
$\lfloor \log n \rfloor \leq \log n < \log 2^{h+1} = h + 1$, amiből $\lfloor \log n \rfloor \leq h$.

A $h \leq 1,45 \log n$ egyenlőtlenség bizonyításához elég azt belátni, hogy ez tetszőleges nemüres, kiegyensúlyozott bináris fára (KBF-re) igaz. Ehhez először meghatározzuk egy $h \geq 0$ magasságú (azaz nemüres) KBF csúcsainak minimális f_h számát. Nyilván $f_0 = 1$ és $f_1 = 2$, hiszen egy nulla magasságú KBF csak a gyökércsúcsból áll, az egy magasságú KBF-ek pedig ((B)G(J)), ((B)G), vagy (G(J)) alakúak.

Az is világos, hogy az $< f_0, f_1, f_2, \cdots >$ sorozat szigorúan monoton növekvő. (Ennek igazolásához vegyünk egy $i+1$ magas, f_{i+1} csúcsú t KBF-et! Ekkor a t bal és jobb részfái közül az egyik magassága i. Legyen ez az f részfa! Jelölje most $s(b)$ egy tetszőleges b bináris fa csúcsainak számát! Akkor $f_{i+1} = s(t) > s(f) \geq f_i$.)

Ezután $h \geq 2$ esetén $f_h = 1 + f_{h-1} + f_{h-2}$. (Ha ugyanis t egy h magasságú, minimális, azaz f_h méretű KBF, akkor ennek bal és jobb részfáiban is, a részfák magasságához mérten a lehető legkevesebb csúcs van. Az egyik részfája kötelezően $h-1$ magas, ebben tehát f_{h-1} csúcs van. Mivel t KBF, a másik részfája $h-1$ vagy $h-2$ magas. A másik részfában tehát f_{h-1} vagy f_{h-2} csúcs van, és $f_{h-2} < f_{h-1}$, tehát a másik részfában f_{h-2} csúcs van, és így $f_h = 1 + f_{h-1} + f_{h-2}$.)

A képlet emlékeztet a Fibonacci sorozatra:
$F_0 = 0$, $F_1 = 1$, $F_h = F_{h-1} + F_{h-2}$, ha $h \geq 2$.

Megjegyzés: A h magasságú, és f_h méretű KBF-eket ezért *Fibonacci fáknak* hívjuk. Az előbbiek szerint egy $h \geq 2$ magasságú Fibonacci fa mindig $(\varphi_{h-1}$ G $\varphi_{h-2})$ vagy $(\varphi_{h-2}$ G $\varphi_{h-1})$ alakú, ahol φ_{h-1} és φ_{h-2} $h-1$, illetve $h-2$ magasságú Fibonacci fák.

1.5. Feladat. *Rajzoljunk $h \in 0..4$ magasságú Fibonacci fákat!*

Megvizsgáljuk, van-e valami összefüggés az
$\langle f_h : h \in \mathbb{N} \rangle$ és az $\langle F_h : h \in \mathbb{N} \rangle$ sorozatok között.

h	0	1	2	3	4	5	6	7	8	9
F_h	0	1	1	2	3	5	8	13	21	34
f_h	1	2	4	7	12	20	33			

A fenti táblázat alapján az első néhány értékre $f_h = F_{h+3} - 1$. Teljes indukcióval könnyen látható, hogy ez tetszőleges $h \geq 0$ egész számra igaz: Feltéve, hogy $0 \leq h \leq k \geq 1$ esetén igaz, $h = k + 1$-re:
$f_h = f_{k+1} = 1 + f_k + f_{k-1} = 1 + F_{k+3} - 1 + F_{k+2} - 1 = F_{k+4} - 1 = F_{h+3} - 1$.

Tudjuk, hogy

$$F_h = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^h - \left(\frac{1-\sqrt{5}}{2} \right)^h \right]$$

Az F_h-ra vonatkozó explicit képlet segítségével összefüggést adunk tetszőleges KBF n mérete és h magassága között.

$$n \geq f_h = F_{h+3} - 1 = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^{h+3} - \left(\frac{1-\sqrt{5}}{2} \right)^{h+3} \right] - 1 \geq$$

$$\geq \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^{h+3} - \left| \left(\frac{1-\sqrt{5}}{2} \right)^{h+3} \right| \right] - 1$$

Mivel $2 < \sqrt{5} < 3$, azért $|1 - \sqrt{5}|/2 < 1$ és $\left| \left(\frac{1-\sqrt{5}}{2} \right)^{h+3} \right| < 1$. Eszerint

$$n > \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^{h+3} - 1 \right] - 1 = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^3 \left(\frac{1+\sqrt{5}}{2} \right)^h - \frac{1+\sqrt{5}}{\sqrt{5}}$$

Mivel

$$\frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^3 = \frac{1 + 3\sqrt{5} + 3(\sqrt{5})^2 + (\sqrt{5})^3}{8\sqrt{5}} = \frac{16 + 8\sqrt{5}}{8\sqrt{5}} = \frac{2+\sqrt{5}}{\sqrt{5}}$$

Ezt behelyettesítve az előbbi, n -re vonatkozó egyenlőtlenségbe:

$$n > \frac{2+\sqrt{5}}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^h - \frac{1+\sqrt{5}}{\sqrt{5}}$$

Az első együtthatót tagokra bontva, a disztributív szabállyal:

$$n > \left(\frac{1+\sqrt{5}}{2}\right)^h + \frac{2}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^h - \frac{1+\sqrt{5}}{\sqrt{5}}$$

Most

$$\frac{2}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^h - \frac{1+\sqrt{5}}{\sqrt{5}} \geq 0 \iff \left(\frac{1+\sqrt{5}}{2}\right)^h \geq \frac{1+\sqrt{5}}{2} \iff h \geq 1$$

Eszerint $h \geq 1$ esetén

$$n > \left(\frac{1+\sqrt{5}}{2}\right)^h$$

$h = 0$ -ra pedig $n = 1$, és így $n = \left(\frac{1+\sqrt{5}}{2}\right)^h$

A fentiekből tetszőleges, nemüres KBF n méretére és h magasságára

$$n \geq \left(\frac{1+\sqrt{5}}{2}\right)^h$$

Innét, ha vesszük mindkét oldal kettes alapú logaritmusát, majd $\log \frac{1+\sqrt{5}}{2}$ -tel osztunk:

$$h \leq \frac{1}{\log \frac{1+\sqrt{5}}{2}} \log n$$

Mivel $1,44 < 1,44042 < \frac{1}{\log \frac{1+\sqrt{5}}{2}} < 1,4404201 < 1,45$, azért tetszőleges, nemüres KBF n méretére és h magasságára

$$h \leq 1,45 \log n$$

2. Általános fák

Az általános fák esetében, a bináris fákkal összehasonlítva, egy csúcsnak tetszőlegesen sok gyereke lehet. Itt azonban, tetszőleges csúcshoz tartozó részfák száma pontosan egyenlő a gyerekek számával, azaz nem tartoznak hozzá üres részfák. Ha a gyerekek sorrendje lényeges, akkor *rendezett fákról* beszélünk.

Általános fákkal modellezhetjük például a számítógépünkben a mappák hierarchiáját, a programok blokkstruktúráját, a függvénykifejezéseket, a családfákat és bármelyik hierarchikus struktúrát.

Vegyük észre, hogy ugyan minden konkrét általános fában van az egy csúcshoz tartozó gyerekek számára valamilyen r felső korlát, de ettől a fa még nem tekinthető r-árisnak, mert ez a korlát nem abszolút: a fa tetszőleges csúcsa gyerekeinek száma tetszőlegesen növelhető. Másrészt, mivel itt nem értelmezzük az *üres részfa* fogalmát, ez mégsem általánosítása az r-áris fa fogalmának. Értelmezzük azonban a gyökércsúcs (nincs szülője) és a levélcsúcs (nincs gyereke) fogalmát, azaz továbbra is gyökeres fákról beszélünk.[2]

Az általános fák természetes ábrázolási módja a *bináris láncolt* reprezentáció. Itt egy csúcs szerkezete a következő (ahol most *child*1 az első gyerekre, *sibling* pedig a következő testvérre mutat). A $*p$ csúcs akkor levél, ha $p \to child1 = \oslash$; és a $*p$ csúcs akkor utolsó testvér, ha $p \to sibling = \oslash$.

Node
+ *child*1, *sibling* : Node* // *child*1: első gyerek; *sibling*: következő testvér + *key* : T // T ismert típus
+ Node() { *child*1 := *sibling* := $\oslash$ } // egycsúcsú fát képez belőle + Node(x:T) { *child*1 := *sibling* := $\oslash$; *key* := x }

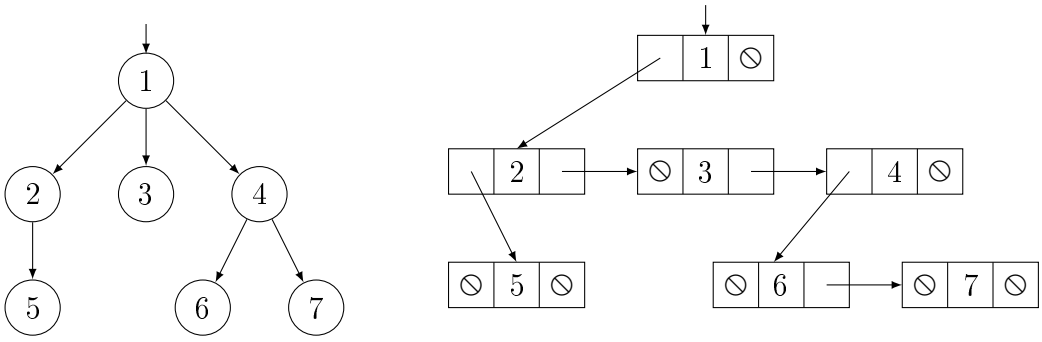
Természetesen itt is lehet *szülő* pointer, ami a gyerekek közös szülőjére mutat vissza.

2.1. Feladat. *Próbáljunk megadni az általános fákra másfajta láncolt reprezentációkat is! Hasonlítsuk össze az általunk adott ábrázolásokat és a bináris láncolt reprezentációt, memóriaigény és rugalmasság szempontjából!*

A szöveges (zárójeles) reprezentációban az általános fáknál a gyökeret előre szokás venni. Így egy nemüres fa általános alakja (G $t_1 \dots t_n$), ahol G a gyökércsúcs tartalma, $t_1 \dots t_n$ pedig a részfák.

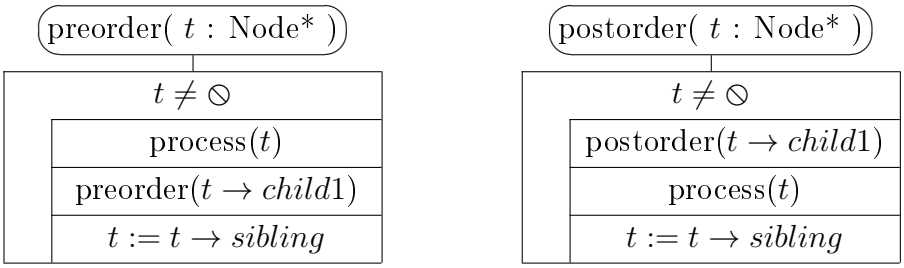
Így pl. az { 1 [2 (5)] (3) [4 (6) (7)] } általános fában az 1 van a gyökérben, a gyerekei a 2, a 3 és a 4 kulcsú csúcsok, a hozzájuk tartozó részfák pedig sorban a [2 (5)], a (3) és a [4 (6) (7)]. A fa levelei az 5, a 3, a 6 és a 7 kulcsú csúcsok (7. ábra).

2.2. Feladat. *Írjunk programot, ami kiír egy binárisan láncolt fát a fenti zárójeles alakban! Írjunk olyat is, ami egy szövegfájlból visszaolvassa! Készítsük el a kiíró és a visszaolvasó programokat az általunk kidolgozott alternatív láncolt ábrázolásokra is! (A visszaolvasó programokat általában nehezebb megírni.)*



7. ábra. Az { 1 [2 (5)] (3) [4 (6) (7)] } általános fa absztrakt szerkezete (balra) és bináris reprezentációja (jobbra).

Az általános fa preorder bejárása[3] a $child1 \sim left$ és $sibling \sim right$ megfeleltetéssel a bináris reprezentáció preorder bejárását igényli. Az általános fa postorder bejárásához[4] azonban (az előbbi megfeleltetéssel) a bináris reprezentáció inorder bejárása szükséges.[5]



2.3. Feladat. *Lássuk be a fenti bejárások helyességét! (Ötlet: A testvérek listájának mérete szerinti teljes indukció pl. célravezető.) Lássuk be egy ellenpélda segítségével, hogy az általános fa inorder bejárása*[6] *nem szimulálható a bináris reprezentáció egyik nevezetes bejárásával*[7] *sem! Írjuk meg a bináris láncolt reprezentációra az általános fa inorder bejárását és szintfolytonos bejárását is!*

[2]Ellentétben az ún. *szabad fákkal*, amik irányítatlan, körmentes gráfok.

[3]először a gyökér, majd sorban a gyerekeihez tartozó részfák

[4]előbb sorban a gyökér gyerekeihez tartozó részfák, végül a gyökér

[5]A fájlrendszerekben általában preorder bejárás szerint keresünk, míg a függvénykifejezéseket (eltekintve most a lusta kiértékeléstől, aminek a tárgyalása túl messzire vezetne) postorder bejárással értékeljük ki.

[6]A (G $t_1 \dots t_n$) fára $n > 0$ esetén előbb t_1-et járja be, majd feldolgozza G-t, aztán bejárja sorban a $t_2 \dots t_n$ részfákat; $n = 0$ esetén csak feldolgozza G-t.

[7]preorder, inorder, postorder, szintfolytonos

2.4. Feladat. *Írjuk meg a fenti bejárásokat az általunk adott alternatív láncolt reprezentációkra is! Írjunk olyan programokat, amelyek képesek tetszőleges általános fa egy adott reprezentációjából egy adott másik ábrázolású másolatot készíteni!*

3. B+ fák és műveleteik

http://aszt.inf.elte.hu/∼asvanyi/ad/ad2jegyzet/B+ fa.pdf

B+ fák

Az alábbiakban Dr. Carl Burch B+-trees című Internetes tananyagának kissé bővített fordítását adjuk.

Tartalom:

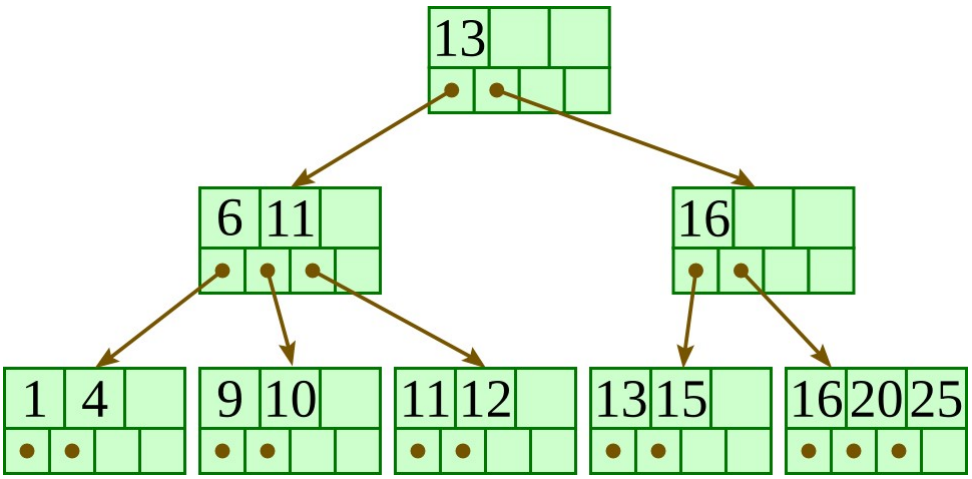
1. Mi a B+ fa?

Az adathalmazokon a legtöbb keresés és módosítás gyorsabban, hatékonyabban hajtható végre, ha a típusértékeket rendezve tároljuk. Nem praktikus azonban a rekordokat szekvenciálisan elhelyezni, mert így a legtöbb beszúrás és törlés kapcsán a tároló jelentős részének újraírása válna szükségessé. A rendezett vagy rendezetlen láncolt listákon pedig a keresés nem elég hatékony.

Ez arra vezet bennünket, hogy az adatokat keresőfába rendezve képzeljük el. Az első ötletünk az AVL fák vagy a piros-fekete fák alkalmazása lehetne, most azonban az adatokat egy véletlen elérésű háttértáron, pl. egy mágneslemezen kívánjuk elhelyezni. A mágneslemezek pedig úgy működnek, hogy egyszerre az adatok egy egész blokkját, tipikusan 512 byte vagy négy kilobyte mennyiségű adatot mozgatunk. Egy bináris keresőfa egy csúcsa ennek csak egy töredékét használná, ezért olyan struktúrát keresünk, ami jobban kihasználja a mágneslemez blokkjait.

Innét jön a B+ fa, amiben minden csúcs legfeljebb d mutatót ($4 \le d$), és legfeljebb $d-1$ kulcsot tartalmaz, ahol d a fára jellemző állandó, *a B+ fa fokszáma*. Úgy tekintjük, hogy a belső csúcsokban mindegyik referencia két kulcs "között" van, azaz egy olyan részfa gyökerére mutat, amiben minden érték a két kulcs között található (mindegyik csúcshoz hozzáképzelve balról egy "mínusz végtelen", jobbról egy "plusz végtelen" értékű kulcsot).

Íme $d=4$ fokszámra egy elég kicsi fa:



Szöveges formában:
- Egy negyedfokú B+ fa tetszőleges részfájának szöveges formája, ha a részfa nem egyetlen levélcsúcsból áll, a következő sémák valamelyikére illeszkedik: *(T1,k1,T2), (T1,k1,T2,k2,T3), (T1,k1,T2,k2,T3,k3,T4),* ahol *Tj* a *j*-edik részfa, *kj* pedig a hasító kulcs az *j*-edik és a *(j+1)*-edik részfa

között.
- A negyedfokú B+ fák tetszőleges levelének szöveges formája a következő sémák valamelyikére illeszkedik: *(k1,k2), (k1,k2,k3),* ahol *kj* a B+ fa által reprezentált halmaz kulcsa.
- A példákban, a részfák különböző szintjei szerint, háromféle zárójelpárt használunk: { }, [ ] és ( ).
A fenti fa szöveges, más néven zárójeles formája tehát a következő:

{ [(1, 4) 6 (9, 10) 11 (11, 12)] 13 [(13, 15) 16 (16, 20, 25)] }

Az adatok a levélszinten vannak. A belső kulcsok csak *hasító kulcsok*. Egy adott kulcsú adat keresése során ezek alapján tudhatjuk, melyik ágon keressünk tovább. A levélszinten minden kulcshoz tartozik egy mutató, ami a megfelelő adatrekordra hivatkozik. (A leveleket a *d*-edik mutatókkal gyakran listába fűzik.)

A B+ fák esetében megköveteljük, hogy a gyökércsúcstól mindegyik levél azonos távolságra legyen. Így a fenti ábrán látható fában tárolt 11 kulcs bármelyikére rákeresve (amelyek mindegyikét a legalsó, a levél szinten találjuk: a belső csúcsok kulcsai csak a tájékozódást szolgálják), három csúcsot érintünk (a gyökércsúcsot, egyik középső szintű csúcsot, és az egyik levelet).

A gyakorlatban persze *d* sokkal nagyobb. Tegyük fel például, hogy egy-egy blokk 4KB, a kulcsaink 4 byte-os egész számok, és mindegyik mutató egy 6 byte-os relatív cím (a fájl kezdőcíméhez képest). Akkor a *d* értékét úgy választjuk, hogy a lehető legnagyobb egész szám legyen, amire $4\,(d-1) + 6\,d \leq 4096$. Ezt az egyenlőtlenséget *d*-re megoldva $d \leq 410$ adódik, tehát a *d=410* értéket választjuk. Mint láthatjuk, *d* egészen nagy lehet.

Tetszőleges *d*-ed fokú B+ fa a következő invariánsokat teljesíti, ahol $4 \leq d$ állandó:

- Minden levélben legfeljebb *d-1* kulcs, és ugyanennyi, a megfelelő (azaz ilyen kulcsú) adatrekordra hivatkozó mutató található.
- A gyökértől mindegyik levél ugyanolyan távol található. (Más szavakkal, minden levél azonos mélységben, a legalsó szinten van.)
- Minden belső csúcsban eggyel több mutató van, mint kulcs, ahol *d* a felső határ a mutatók számára.
- Minden *Cs* belső csúcsra, ahol *k* a *Cs* csúcsban a kulcsok száma: az első gyerekhez tartozó részfában minden kulcs kisebb, mint a *Cs* első kulcsa; az utolsó gyerekhez tartozó részfában minden kulcs nagyobb-egyenlő, mint a *Cs* utolsó kulcsa; és az *i*-edik gyerekhez tartozó részfában ($2 \leq i \leq k$) lévő tetszőleges *r* kulcsra $Cs.kulcs[i-1] \leq r < Cs.kulcs[i]$.
- A gyökércsúcsnak legalább két gyereke van (kivéve, ha ez a fa egyetlen csúcsa, következésképpen az egyetlen levele is).
- Minden, a gyökértől különböző belső csúcsnak legalább $floor(d / 2)$ gyereke van. ($floor(d / 2) = d/2$ alsó egész-rész.)
- Minden levél legalább $floor(d / 2)$ kulcsot tartalmaz (kivéve, ha a fának egyetlen csúcsa van).
- A B+ fa által reprezentált adathalmaz minden kulcsa megjelenik valamelyik levélben, balról jobbra szigorúan monoton növekvő sorrendben.

A belső csúcsokban található *hasító kulcsok* segítségével tetszőleges levélcsúcsbeli kulcsot gyorsan megtalálhatunk (illetve megtudhatjuk, ha nincs a levelekben), a fenti invariánsok alapján: a csúcsokban is logaritmikusan keresve, és mindig a megfelelő ágon továbbhaladva, *O(log n)* lépésben (ahol *n* a B+ fával ábrázolt adathalmaz mérete). A hasító kulcsok nem okvetlenül szerepelnek a levelekben. (Mint látni fogjuk, a törlő algoritmus ténylegesen is létrehoz ilyen fákat.)

Ugyan a fa magassága *O(log n)*, a gyakorlatban a fa *h* magassága a *log n* érték töredéke:
HF: Bizonyítsuk be, hogy a $log[d](n/(d-1)) \leq h \leq log[floor(d/2)](n/2)$ egyenlőtlenség teljesül!

(*log[d](n)* = az *n* szám *d* alapú logaritmusa.)

Ez azt jelenti, hogy a fenti *d=410* értékkel *log[410](n/409) ≤ h ≤ log[205](n/2).*
Pl. ha *n=1.000.000.000 (n=egymilliárd),* akkor *2,4 < h < 3,8*, vagyis *h=3*, azaz a fának pontosan négy szintje van. Egy ilyen fánál a felső két szintet az adatbázis megnyitásakor betöltjük a központi tárba. A levélszintről viszont még egyet lépünk a tényleges adatrekord eléréséhez. Ez azt jelenti, hogy egy-egy adat eléréséhez összesen háromszor olvasunk a lemezről, *egymilliárd* rekordot tartalmazó adatbázis esetén. Ha pedig a keresett kulcsú rekord nincs az adatbázisban, csak kétszer olvasunk a lemezről.

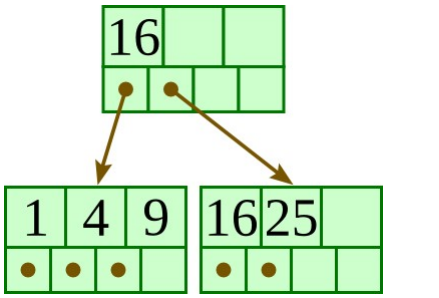
Az alábbi példákban továbbra is a *d=4* értékkel dolgozunk. A fenti invariánsok alapján ez azt jelenti, hogy minden levél legalább két kulcsot tartalmaz; minden belső csúcsnak pedig legalább két gyereke és legalább egy hasító kulcsa van.

2. A beszúrás algoritmusa

Ha a fa üres, hozzunk létre egy új levélcsúcsot, ami egyben a gyökércsúcs is, és a beszúrandó kulcs/mutató pár a tartalma! Különben keressük meg a kulcsnak megfelelő levelet! Ha a levélben már szerepel a kulcs, a beszúrás sikertelen. Különben menjünk az 1. pontra!

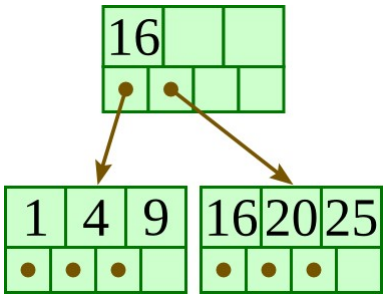
1. Ha a csúcsban van üres hely, szúrjuk be a megfelelő kulcs/mutató párt kulcs szerint rendezetten ebbe a csúcsba!
2. Ha a csúcs már tele van, vágjuk szét két csúccsá, és osszuk el a *d* darab kulcsot egyenlően a két csúcs között! Ha a csúcs egy levél, vegyük a második csúcs legkisebb értékének másolatát, és ismételjük meg ezt a beszúró algoritmust, hogy beszúrjuk azt a szülő csúcsba! Ha a csúcs nem levél, vegyük ki a középső értéket a kulcsok elosztása során, és ismételjük meg ezt a beszúró algoritmust, hogy beszúrjuk ezt a középső értéket a szülő csúcsba! (Ha kell, a szülő csúcsot előbb létrehozzuk. Ekkor a B+ fa magassága nő.)

Eredeti:



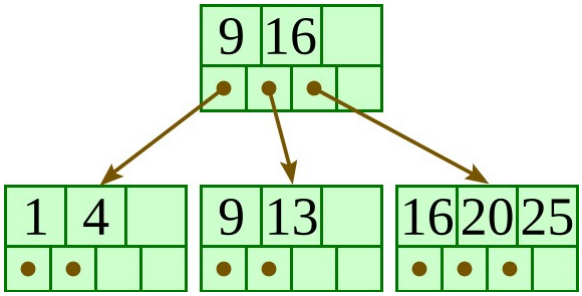
[(1, 4, 9) 16 (16, 25)]

A 20 beszúrása:
Beszúródik a megfelelő levélbe.



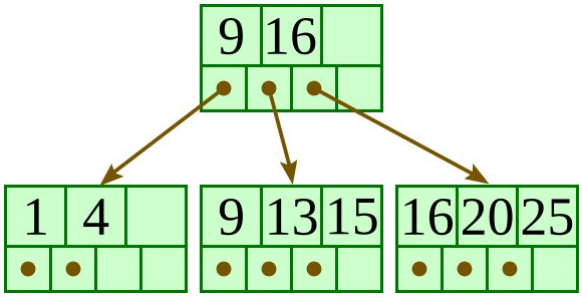
[(1, 4, 9) 16 (16, 20, 25)]

A 13 beszúrása:
(1, 4 | 9, 13)
hasad, a 2. levél
minimuma a
szülőbe másolódik
új hasítókulcsnak.



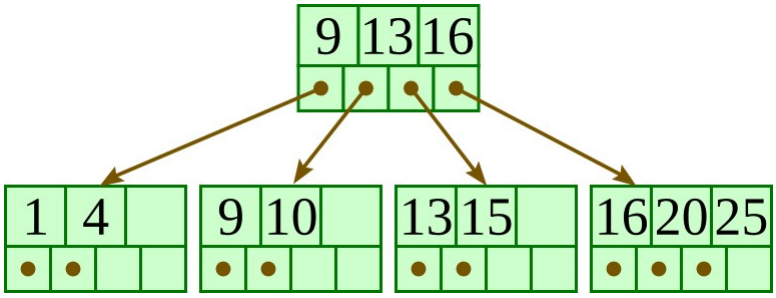
[(1, 4) 9 (9, 13) 16 (16, 20, 25)]

A 15 beszúrása:
Beszúródik a
megfelelő
levélbe.



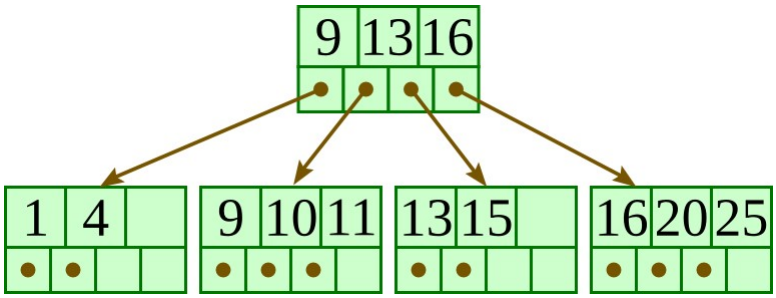
[(1, 4) 9 (9, 13, 15) 16 (16, 20, 25)]

A 10 beszúrása:
(9, 10 | 13, 15)
hasad, (13, 15)
minimuma a
szülőbe másolódik
új hasítókulcsnak.



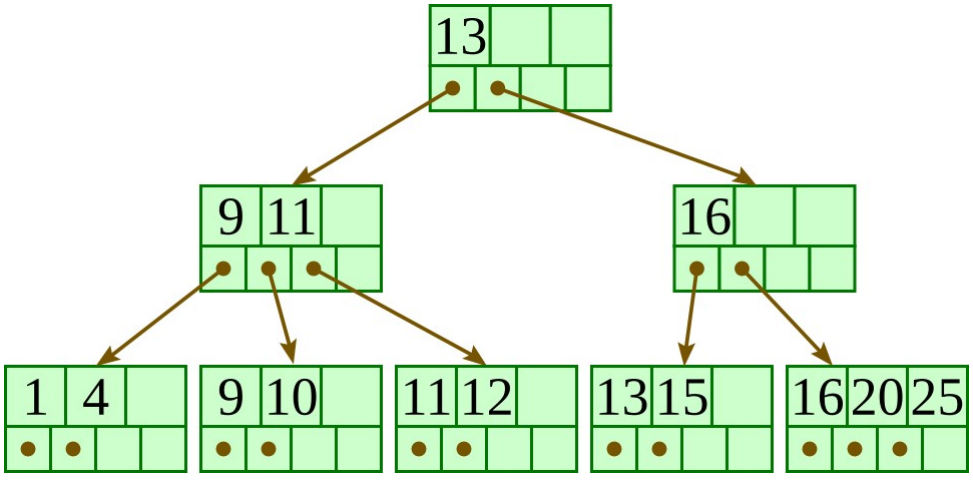
[(1, 4) 9 (9, 10) 13 (13, 15) 16 (16, 20, 25)]

A 11 beszúrása:
Beszúródik a
megfelelő
levélbe.



[(1, 4) 9 (9, 10, 11) 13 (13, 15) 16 (16, 20, 25)]

A 12 beszúrása:
(9, 10 | 11, 12)
hasad, (11, 12)
minimuma a
szülőbe másolódik
új hasítókulcsnak;
(9, 11 | 13, 16)
hasad, (13, 16)
minimuma
felmegy az
újonnan létrehozott
gyökérbe.



{[(1,4) 9 (9,10) 11 (11,12)] 13 [(13,15) 16 (16,20,25)]}

3. A törlés algoritmusa

Keressük meg a törlendő kulcsot tartalmazó levelet! Ha ilyen nincs, a törlés meghiúsul.
Különben a törlő algoritnus futása vagy az A esettel fejeződik be; vagy a B esettel folytatódik, ami után a C eset (nullaszor, egyszer, vagy többször) ismédlődhet, és még a D eset is sorra kerülhet végül.

A, A keresés során megtalált levélcsúcs egyben a gyökércsúcs is:

1. Töröljük a megfelelő kulcsot és a hozzá tartozó mutatót a csúcsból!
2. Ha a csúcs tartalmaz még kulcsot, kész vagyunk.
3. Különben töröljük a fa egyetlen csúcsát, és üres fát kapunk.

B, A keresés során megtalált levélcsúcs nem a gyökércsúcs:

1. Töröljük a megfelelő kulcsot és a hozzá tartozó mutatót a levélcsúcsból!
2. Ha a levélcsúcs még tartalmaz elég kulcsot és mutatót, hogy teljesítse az invariánsokat, kész vagyunk.
3. Ha a levélcsúcsban már túl kevés kulcs van ahhoz, hogy teljesítse az invariánsokat, de a következő, vagy a megelőző testvérének több van, mint amennyi szükséges, osszuk el a kulcsokat egyenlően közte és a megfelelő testvére között! Írjuk át a két testvér közös szülőjében a két testvérhez tartozó hasító kulcsot a két testvér közül a második minimumára!
4. Ha a levélcsúcsban már túl kevés kulcs van ahhoz, hogy teljesítse az invariánst, és a következő, valamint a megelőző testvére is a minimumon van, hogy teljesítse az invariánst, akkor egyesítsük egy vele szomszédos testvérével! Ennek során a két testvér közül a (balról jobbra sorrend szerinti) másodikból a kulcsokat és a hozzájuk tartozó mutatókat sorban átmásoljuk az elsőbe, annak eredeti kulcsai és mutatói után, majd a második testvért töröljük. Ezután meg kell ismételnünk a törlő algoritmust a szülőre, hogy eltávolítsuk a szülőből a hasító kulcsot (ami eddig elválasztotta a most egyesített levélcsúcsokat), a most törölt második testvérre hivatkozó mutatóval együtt.

C, Belső — a gyökértől különböző — csúcsból való törlés:

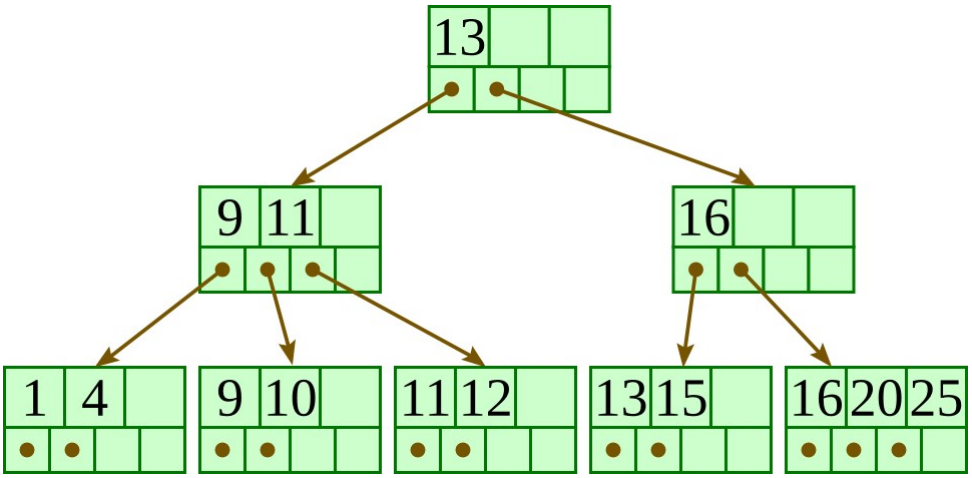
1. Töröljük a belső csúcs éppen most egyesített két gyereke közti hasító kulcsot és az egyesítés során törölt gyerekére hivatkozó mutatót a belső csúcsból!
2. Ha a belső csúcsnak van még floor(d/2) gyereke, (hogy teljesítse az invariánsokat) kész vagyunk.
3. Ha a belső csúcsnak már túl kevés gyereke van ahhoz, hogy teljesítse az invariánsokat, de a következő, vagy a megelőző testvérének több van, mint amennyi szükséges, osszuk el a gyerekeket és a köztük levő hasító kulcsokat egyenlően közte és a megfelelő testvére között, a hasító kulcsok közé a testvérek közti (a közös szülőjükben lévő) hasító kulcsot is beleértve! A gyerekek és a hasító kulcsok újraelosztása során, a középső hasító kulcs a testvérek közös szülőjében a két testvérhez tartozó régi hasító kulcs helyére kerül úgy, hogy megfelelően reprezentálja a köztük megváltozott vágási pontot! (Ha a két testvérben a gyerekek összlétszáma páratlan, akkor az újraelosztás után is annak a testvérnek legyen több gyereke, akinek előtte is több volt!)
4. Ha a belső csúcsnak már túl kevés gyereke van ahhoz, hogy teljesítse az invariánst, és a következő, valamint a megelőző testvére is a minimumon van, hogy teljesítse az invariánst, akkor egyesítsük egy vele szomszédos testvérével! Az egyesített csúcsot a két testvér közül a (balról jobbra sorrend szerinti) elsőből hozzuk létre. Gyerekei és hasító kulcsai először a saját gyerekei és hasító kulcsai az eredeti sorrendben, amiket a két testvér közti (a közös szülőjükben lévő) hasító kulcs követ, és végül a második testvér gyerekei és hasító kulcsai jönnek, szintén az eredeti sorrendben. Ezután töröljük a második testvért. A két testvér egyesítése után meg kell ismételnünk a törlő algoritmust a közös szülőjükre, hogy eltávolítsuk a szülőből a hasító kulcsot (ami eddig elválasztotta a most

egyesített testvéreket), a most törölt második testvérre hivatkozó mutatóval együtt.

D, A gyökércsúcsból való törlés, ha az nem levélcsúcs:

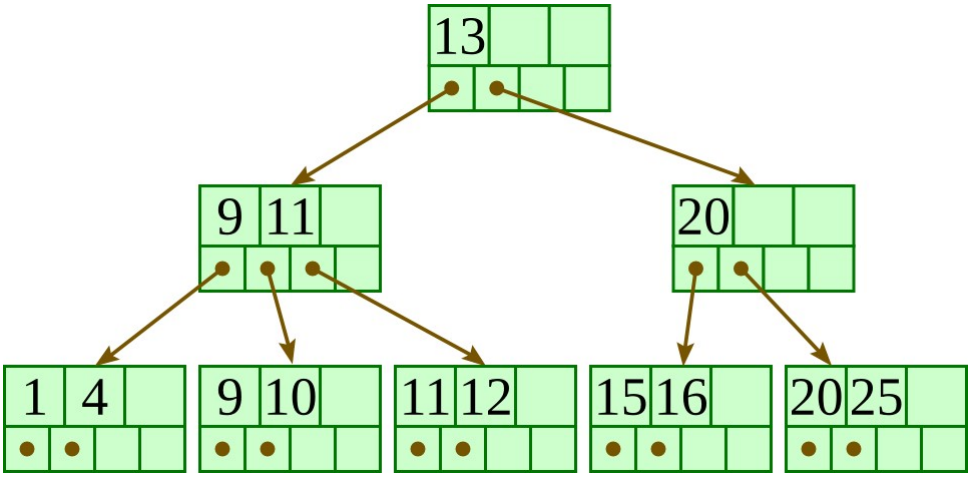
1. Töröljük a gyökércsúcs éppen most egyesített két gyereke közti hasító kulcsot és az egyesítés során törölt gyerekére hivatkozó mutatót a gyökércsúcsból!
2. Ha a gyökércsúcsnak van még 2 gyereke, kész vagyunk.
3. Ha a gyökércsúcsnak csak 1 gyereke maradt, akkor töröljük a gyökércsúcsot, és a megmaradt egyetlen gyereke legyen az új gyökércsúcs! (Ekkor a B+ fa magassága csökken.)

Eredeti:



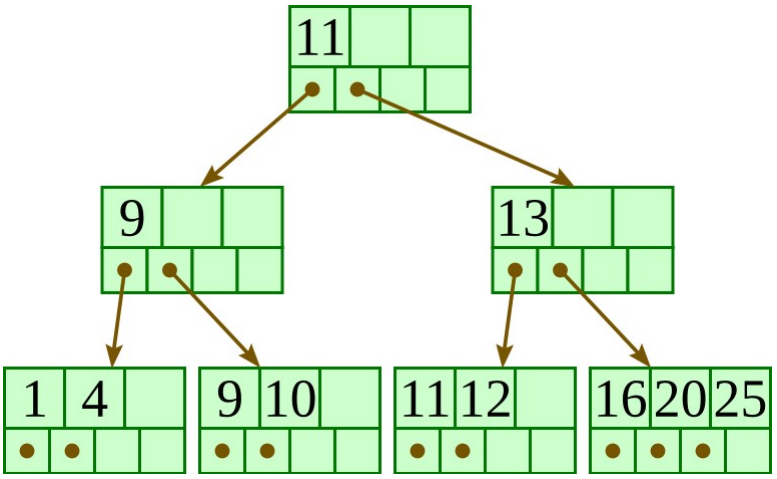
{[(1,4) 9 (9,10) 11 (11,12)] 13 [(13,15) 16 (16,20,25)]}

A 13 törlése:
A 15 egyedül marad, a 16-ot átveszi a jobboldali testvérétől, majd a szülőjükben átíródik a megfelelő hasító kulcs.



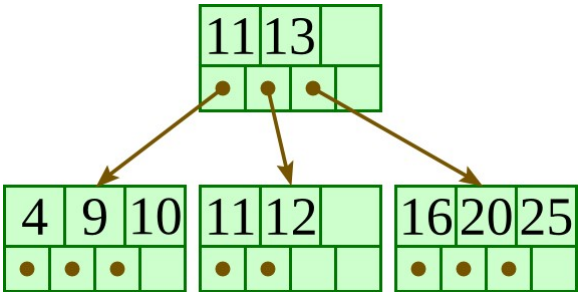
{[(1, 4) 9 (9, 10) 11 (11, 12)] 13 [(15, 16) 20 (20, 25)]}

A 15 törlése:
A két testvér levél egyesül, a 20 hasító kulcs törlődik a szülőből, aminek 1 gyereke marad, és egyet átvesz a testvérétől; a 13 a nagyszülőből lejön, és a 11 a 13 helyére megy.



{ [(1, 4) 9 (9, 10)] 11 [(11, 12) 13 (16, 20, 25)] }

Az 1 törlése:
(4, 9, 10) egyesül,
szülők egyesülnek,
11 lejön az egyesült szülőbe, a gyökérnek
1 gyereke marad:
törlődik.



[(4, 9, 10) 11 (11, 12) 13 (16, 20, 25)]

Algoritmusok és adatszerkezetek II. előadásjegyzet:

Elemi gráfalgoritmusok

Ásványi Tibor – asvanyi@inf.elte.hu

2023. október 9.

Tartalomjegyzék

Hivatkozások

[1] Ásványi Tibor, Algoritmusok és adatszerkezetek II.
Útmutatások a tanuláshoz, jelölések, tematika,
fák, gráfok,
mintaillesztés, tömörítés
http://aszt.inf.elte.hu/~asvanyi/ad/ad2jegyzet/

[2] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.,
magyarul: Új Algoritmusok, *Scolar Kiadó*, Budapest, 2003.
ISBN 963 9193 90 9
angolul: Introduction to Algorithms (Third Edititon),
The MIT Press, 2009.

[3] Fekete István, Algoritmusok jegyzet
http://ifekete.web.elte.hu/

[4] Rónyai Lajos – Ivanyos Gábor – Szabó Réka, Algoritmusok,
TypoTEX Kiadó, 1999. ISBN 963 9132 16 0
https://www.tankonyvtar.hu/hu/tartalom/tamop425/2011-0001-526_ronyai_algoritmusok/adatok.html

[5] Tarjan, Robert Endre, Data Structures and Network Algorithms,
CBMS-NSF Regional Conference Series in Applied Mathematics, 1987.

[6] Weiss, Mark Allen, Data Structures and Algorithm Analysis,
Addison-Wesley, 1995, 1997, 2007, 2012, 2013.

[7] Ásványi Tibor, Algoritmusok és adatszerkezetek I. előadásjegyzet (2021)
http://aszt.inf.elte.hu/~asvanyi/ad/ad1jegyzet/ad1jegyzet.pdf

1. Egyszerű gráfok és ábrázolásaik ([2] 22)

Gráfok segítségével pl. hálózatokat, folyamatokat modellezhetünk. A modelleket természetesen ábrázolnunk kell tudni a számítógépben, illetve matematikailag; vagy éppen szemléletesen, a jobb megértés céljából.

1.1. Gráfelméleti alapfogalmak

1.1. Definíció. Gráf *alatt egy* $G = (V, E)$ *rendezett párost értünk, ahol* V *a csúcsok (vertices) tetszőleges, véges halmaza,* $E \subseteq V \times V \setminus \{(u, u) : u \in V\}$ *pedig az élek (edges) halmaza. Ha* $V = \{\}$*, akkor üres gráfról, ha* $V \neq \{\}$*, akkor nemüres gráfról beszélünk.*

Tehát már a definíció szintjén kizárjuk a gráfokból párhuzamos éleket és a hurokéleket. Nincs ugyanis semmilyen eszközünk arra, hogy két (u, v) élet megkülönböztessünk (párhuzamos élek), az (u, u) alakú, ún. hurokéleket pedig expliciten kizártuk.

Így a továbbiakban gráf alatt tulajdonképpen *egyszerű gráfot* értünk.

1.2. Definíció. *A* $G = (V, E)$ *gráf* irányítatlan*, ha tetszőleges* $(u, v) \in E$ *élre* $(u, v) = (v, u)$*.*

1.3. Definíció. *A* $G = (V, E)$ *gráf* irányított*, ha tetszőleges* $(u, v), (v, u) \in E$ *élpárra* $(u, v) \neq (v, u)$*. Ilyenkor azt mondjuk, hogy az* (u, v) *él* fordítottja *a* (v, u) *él, és viszont.*

Mint látható, az irányítatlan gráfoknál tetszőleges (u, v) éllel együtt (v, u) is a gráf éle, hiszen ez a két él egyenlő.

Irányított gráfoknál általában – de nem szükségszerűen – lesz a gráfnak olyan (u, v) éle, hogy ennek fordítottja, (v, u) nem éle a gráfnak.

1.4. Definíció. *A* $G = (V, E)$ *gráf csúcsainak* (V) *egy* $\langle u_0, u_1, \ldots u_n \rangle$ *(*$n \in \mathbb{N}$*) sorozata* a gráf egy útja*, ha tetszőleges* $i \in 1..n$*-re* $(u_{i-1}, u_i) \in E$*. Ezek az* (u_{i-1}, u_i) *élek az út élei. Az* út hossza *ilyenkor* n*, azaz az utat alkotó élek számával egyenlő.*

1.5. Definíció. *Tetszőleges* $\langle u_0, u_1, \cdots, u_n \rangle$ *út* rész-útja $0 \leq i \leq j \leq n$ *esetén az* $\langle u_i, u_{i+1}, \cdots, u_j \rangle$ *út.*

A kör *olyan út, aminek kezdő és végpontja (csúcsa) azonos, a hossza* > 0*, és az élei páronként különbözőek.*

Az egyszerű kör *olyan kör, aminek csak a kezdő és a végpontja azonos.*

Tetszőleges út *akkor* tartalmaz kört*, ha van olyan rész-útja, ami kör.*

Körmentes út *alatt olyan utat értünk, ami nem tartalmaz kört.*

Körmentes gráf *alatt olyan gráfot értünk, amiben csak körmentes utak vannak.*

Vegyük észre, hogy (az „Algoritmusok" témakörben elterjedt szokásnak megfelelően) az utak köröket is tartalmazhatnak! A fentiek szerint tetszőleges kör hossza ≥ 2.

1.6. Definíció. DAG *alatt irányított, körmentes gráfot értünk (directed acyclic graph).*

A DAG-ok modelezhetnek például összetett folyamatokat, ahol a gráf csúcsai elemi műveletek, az élei pedig az ezek közötti rákövetkezési kényszerek.

1.7. Definíció. *Tetszőleges* $G = (V, E)$ irányított gráf irányítatlan megfelelője *az a* $G' = (V, E')$ *irányítatlan gráf, amire*
$E' = \{(u, v) : (u, v) \in E \vee (v, u) \in E\}$.

1.8. Definíció. *A* G irányítatlan gráf összefüggő, *ha* G *tetszőleges csúcsából bármelyik csúcsába vezet út.*

A G irányított gráf összefüggő, *ha az irányítatlan megfelelője összefüggő.*

1.9. Definíció. *Az irányítatlan, körmentes, összefüggő gráfokat* szabad fáknak, *más néven* irányítatlan fáknak *nevezzük.*

1.10. Definíció. *Az* u *csúcs a* G *irányított gráf* generátor csúcsa, *ha* u*-ból a* G *tetszőleges* v *csúcsa elérhető, azaz létezik* $u \rightsquigarrow v$ *út.*

1.11. Tulajdonság. *Ha a* G *irányított gráfnak van generátor csúcsa, akkor összefüggő, de fordítva nem igaz az állítás.*

1.12. Definíció. T gyökeres fa, *más néven* irányított fa, *ha* T *olyan irányított gráf, aminek van generátor csúcsa, nincs olyan éle, aminek a fordítottja is éle a gráfnak, és a* T *irányítatlan megfelelője körmentes. Ilyenkor a generátor csúcsot* a fa gyökér csúcsának *is nevezzük.*

1.13. Tulajdonság. *Tetszőleges (gyökeres vagy szabad) nemüres fának pontosan eggyel kevesebb éle van, mint ahány csúcsa.*

1.14. Definíció. *A* $G = (V, E)$ *gráfnak* részgráfja *a* $G' = (V', E')$ *gráf, ha* $V' \subseteq V \wedge E' \subseteq E$*, és mindkét gráf irányított, vagy mindkettő irányítatlan.*

A G *gráfnak* valódi részgráfja *a* G' *gráf, ha* G*-nek részgráfja* G'*, de* $G \neq G'$ *és* G' *nemüres.*

1.15. Definíció. *Két (rész)gráf* diszjunkt, *ha nincs közös csúcsuk (és ebből következően közös élük sem).*

1.16. Definíció. *A G gráf* összefüggő komponense *a G' gráf, ha G-nek részgráfja G' és G' összefüggő, de G-nek nincs olyan összefüggő részgráfja, aminek G' valódi részgráfja.*

1.17. Tulajdonság. *Tetszőleges gráf vagy összefüggő, vagy felbontható (egymástól diszjunkt) összefüggő komponensekre (amelyek együtt kiadják a teljes gráfot).*

1.18. Definíció. *A G gráf erdő, ha összefüggő komponensei fák (vagy egyetlen fából áll).*

1.19. Tulajdonság. *A G irányítatlan gráf erdő $\iff$ G körmentes.*

A G irányított gráf erdő $\iff$ G irányítatlan megfelelője körmentes, és G mindegyik összefüggő komponensének van generátor csúcsa.

1.2. Bevezetés a gráfábrázolásokhoz

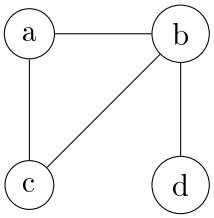
A gráfábrázolásoknál a $G = (V, E)$ gráfról általában föltesszük, hogy $V = \{v_1, \cdots, v_n\}$, ahol $n \in \mathbb{N}$, azaz hogy a gráf csúcsait egyértelműen azonosítják az $1..n$ sorszámok.

Grafikus és *szöveges* reprezentációban (ld. alább) a csúcsok sorszámait a szemléletesség kedvéért gyakran az angol ábécé kisbetűivel jelöljük. Ilyenkor például $a = 1, b = 2, \cdots, z = 26$ lehet. (Szemléltető ábráinkhoz ez bőven elég lesz.)

1.3. Grafikus ábrázolás

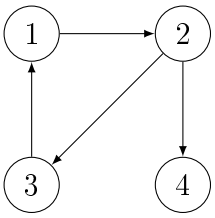
A gráfoknál megszokott módon a csúcsokat kis körök jelölik, az éleket irányított gráfoknál a körök közti nyilak, irányítatlan esetben a köröket összekötő vonalak reprezentálják. A csúcsok sorszámát (illetve az azt reprezentáló betűt) általában a körökbe írjuk.
Az 1. ábrán egy egyszerű, irányítatlan gráfot láthatunk, grafikus és szöveges reprezentációban, a csúcsokat kisbetűkkel azonosítva.

A 2. ábrán pedig egy hasonló, irányított gráfot találunk, újra grafikus és szöveges reprezentációban is, a csúcsokat 1-től 4-ig sorszámozva. (Irányítatlan gráfoknál is sorszámozhatjuk a csúcsokat és irányított gráfoknál ugyanúgy használhatunk betű azonosítókat is.)



a – b ; c.
b – c ; d.

1. ábra. Ugyanaz az irányítatlan gráf grafikus (balra) és szöveges (jobbra) ábrázolással.



$1 \to 2.$
$2 \to 3$; 4.
$3 \to 1.$

2. ábra. Ugyanaz az irányított gráf grafikus (balra) és szöveges (jobbra) ábrázolással.

1.4. Szöveges ábrázolás

Az irányítatalan gráfoknál „u – $v_{u_1}; \cdots ; v_{u_k}$." azt jelenti, hogy a gráfban az u csúcsnak szomszédai a $v_{u_1}, \cdots, v_{u_k}$ csúcsok, azaz $(u, v_{u_1}), \cdots, (u, v_{u_k})$ élei a gráfnak. (Ld. az 1. ábrát!)

Az irányított gráfoknál pedig „$u \to v_{u_1}; \cdots ; v_{u_k}$." azt jelenti, hogy a gráfban az u csúcsból az $(u, v_{u_1}), \cdots, (u, v_{u_k})$ irányított élek indulnak ki, azaz az u csúcs közvetlen rákövetkezői, vagy más néven gyerekei a $v_{u_1}, \cdots, v_{u_k}$ csúcsok. (Ld. a 2. ábrát!)

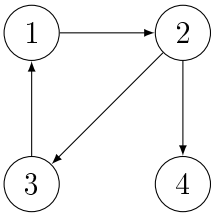
1.5. Szomszédossági mátrixos (adjacency matrix), más néven csúcsmátrixos reprezentáció

A szomszédossági mátrixos, vagy más néven csúcsmátrixos ábrázolásnál a $G = (V, E)$ gráfot ($V = \{v_1, \cdots, v_n\}$) egy $A/1 : bit[n, n]$ mátrix reprezentálja, ahol $n = |V|$ a csúcsok száma, $1..n$ a csúcsok sorszámai, azaz azonosító indexei,
type *bit* **is** $\{0, 1\}$; és tetszőleges $i, j \in 1..n$ csúcssorszámokra

$$A[i, j] = 1 \iff (v_i, v_j) \in E$$

$$A[i, j] = 0 \iff (v_i, v_j) \notin E.$$

A 3. ábrán látható irányított gráfot például a mellette lévő bitmátrix reprezentálja.



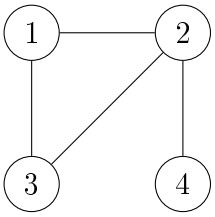
A	1	2	3	4
1	0	1	0	0
2	0	0	1	1
3	1	0	0	0
4	0	0	0	0

3. ábra. Ugyanaz az irányított gráf grafikus (balra) és szomszédossági mátrixos (jobbra) ábrázolással.

A főátlóban mindig nullák vannak, mert csak egyszerű gráfokkal foglalkozunk (amelyekben nincsenek hurokélek).

Vegyük észre, hogy irányítatlan esetben a szomszédossági mátrixos reprezentáció mindig szimmetrikus, hiszen $(v_i, v_j) \in E \iff (v_j, v_i) \in E$.

Az 1. ábráról már ismerős irányítatlan gráf csúcsmátrixos ábrázolása a szokásos $a = 1, b = 2, c = 3, d = 4$ megfeleltetéssel a 4. ábrán látható.



A	1	2	3	4
1	0	1	1	0
2	1	0	1	1
3	1	1	0	0
4	0	1	0	0

4. ábra. Ugyanaz az irányítatlan gráf grafikus (balra) és szomszédossági mátrixos (jobbra) ábrázolással.

Az irányítatlan gráfoknál tehát tetszőleges v_i és v_j csúcsokra $A[i, j] = A[j, i]$, valamint $A[i, i] = 0$. Elég azért az alsóháromszög (vagy a felsőháromszög) mátrixot ábrázolnunk, a főátló nélkül, pl. sorfolytonosan. A ténylegesen ábrázolt alsóháromszög mátrix így egy elemet tartalmaz a 2. sorban, kettőt a 3. sorban, ... $(n-1)$ elemet az utolsó sorban, ami

$$n^2 \text{ bit helyett csak } 1 + 2 + \cdots + (n-1) = n * (n-1)/2 \text{ bit.}$$

Az A mátrix helyett tehát felvehetünk pl. egy $B : bit[n * (n-1)/2]$ tömböt, ami az $a_{ij} = A[i, j]$ jelöléssel az

$$\langle a_{21}, a_{31}, a_{32}, a_{41}, a_{42}, a_{43}, \ldots, a_{n1}, \ldots, a_{n(n-1)} \rangle$$

absztrakt sorozatot reprezentálja sorfolytonosan. Innét

$A[i,j] = B[(i-1)*(i-2)/2+(j-1)]$ ha $i > j$ (az alsóháromszög mátrixban)
$A[i,j] = A[j,i]$ ha $i < j$ ($A[i,j]$ a felsőháromszög mátrixban)
$A[i,i] = 0$. ($A[i,i]$ a főátlón van)

Ha ui. meg szeretnénk határozni tetszőleges $a_{ij} = A[i,j]$, az alsóháromszög mátrixban található elem helyét a ténylegesen is létező B tömbben, akkor azt kell megszámolnunk, hogy hány elem előzi meg sorfolytonosan az a_{ij} elemet a B tömbben. Mivel a B tömböt nullától indexeljük, a_{ij} indexe a B-ben egyenlő lesz az a_{ij}-t megelőző elemek számával. Az alsóháromszög mátrixban az a_{ij} elemet az alábbi elemek előzik meg sorfolytonosan:

a_{21}
a_{31}, a_{32}
a_{41}, a_{42}, a_{43}
$\vdots$
$a_{(i-1)1}, a_{(i-1)2}, \cdots, a_{(i-1)(i-2)}$
$a_{i1}, a_{i2}, \cdots, a_{i(j-1)}$

Ez pedig összesen $(1+2+3+\cdots+(i-2))+(j-1) = (i-1)*(i-2)/2+(j-1)$ elem.

A csúcsmátrixos (más néven szomszédossági mátrixos) ábrázolásnál $\Theta(1)$ idő alatt eldönthető a $(v_i, v_j) \in E$ kérdés, így olyan algoritmusoknál előnyös, ahol gyakori ez a művelet.

Adott csúcs (irányított gráfoknál) gyerekeinek, vagy (irányítatlan gráfoknál) szomszédainak felsorolásához viszont n lépésre van szükségünk, ami általában lényegesen több, mint ahány gyerek vagy szomszéd ténylegesen van.

1.6. Szomszédossági listás (adjacency list) reprezentáció

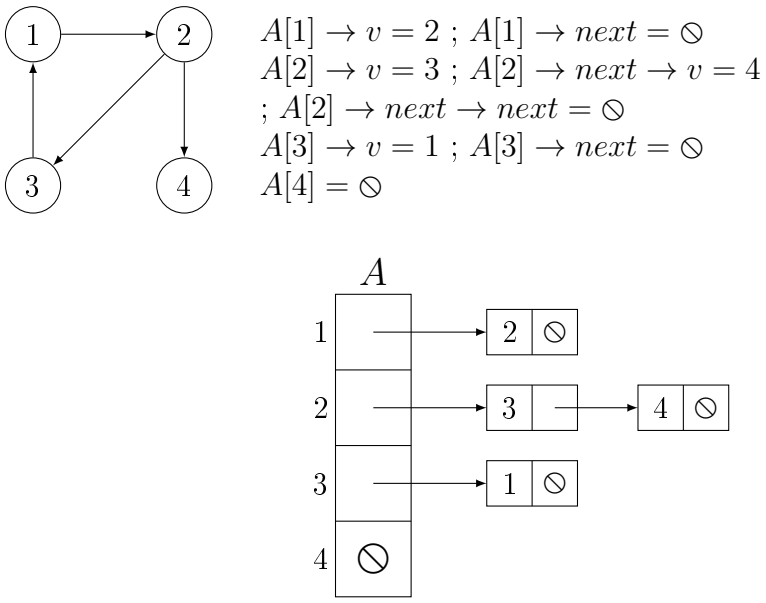
A szomszédossági listás ábrázolás hasonlít a szöveges reprezentációhoz. A $G = (V, E)$ gráfot ($V = \{v_1, \cdots, v_n\}$) az $A/1 : Edge^*[n]$ pointertömb

Edge
$+v : \mathbb{N}$
$+next : Edge*$

segítségével ábrázoljuk, ahol irányítatlan gráf esetében a v_i csúcs szomszédainak sorszámait az $A[i]$ S1L tartalmazza ($i \in 1..n$). A v_i csúcs szomszédainak indexeit tehát az $A[i]$ lista elemeinek v adattagjai tartalmazzák. Így az $A[i]$

lista elemei a v_i csúcshoz kapcsolódó éleknek felelnek meg. Irányítatlan gráfok esetén azért minden élet kétszer ábrázolunk, hiszen ha pl. a v_i csúcsnak szomszédja v_j, akkor a v_j csúcsnak is szomszédja v_i.

Irányított gráfok esetén hasonló a reprezentáció, de az $A[i]$ S1L csak az v_i csúcs gyerekeinek (más néven közvetlen rákövetkezőinek) sorszámait tartalmazza ($i \in 1..n$). Ilyen módon mindegyik élet csak egyszer kell ábrázolnunk. Egy példát láthatunk az 5. ábrán.



5. ábra. Ugyanaz az irányított gráf grafikus (balra) és szomszédossági listás (jobbra) ábrázolással.

A szomszédossági listás ábrázolásnál S1L-ek helyett természetesen másfajta listákat is alkalmazhatunk.

A szomszédossági listás ábrázolásnál a $(v_i, v_j) \in E$ kérdés eldöntéséhez meg kell keresnünk a j indexet az $A[i]$ listán, így olyan algoritmusoknál, ahol gyakori ez a művelet, lehet, hogy érdemes inkább a csúcsmátrixos reprezentációt választani.

Adott csúcs (irányított gráfoknál) gyerekeinek, vagy (irányítatlan gráfoknál) szomszédainak felsorolásához viszont pontosan annyi lépésre van szükségünk, mint ahány gyerek vagy szomszéd ténylegesen van. Mivel ebben a jegyzetben a legtöbb gráfalgoritmusnak ez a leggyakoribb művelete, ezt a reprezentációt gyakran előnyben részesítjük a csúcsmátrixos ábrázolással szemben.

1.7. Számítógépes gráfábrázolások tárigénye

A továbbiakban a $G = (V, E)$ gráf csúcsainak számát $n = |V|$, éleinek számát pedig $m = |E|$ fogja jelölni. Világos, hogy $0 \leq m \leq n * (n-1) \leq n^2$, így $m \in O(n^2)$.

A *ritka gráf*okat az $m \in O(n)$, míg a *sűrű gráf*okat az $m \in \Theta(n^2)$ összefüggéssel jellemezzük.

1.7.1. Szomszédossági mátrixok

A szomszédossági mátrixos (más néven csúcsmátrixos) ábrázolás tárigénye alapesetben n^2 bit. Irányítatlan gráfoknál, csak az alsóháromszög mátrixot tárolva, $n * (n-1)/2$ bit. Mivel $n * (n-1)/2 \in \Theta(n^2)$, az aszimptotikus tárigény mindkét esetben $\Theta(n^2)$.

1.7.2. Szomszédossági listák

Szomszédossági listás reprezentációnál a pointertömb n db mutatóból áll, a szomszédossági listáknak pedig összesen m vagy $2 * m$ elemük van, aszerint, hogy a gráf irányított vagy irányítatlan. Ezért az aszimptotikus tárigény mindkét esetben $\Theta(n + m)$.

Ritka gráfoknál (amik a gyakorlati alkalmazások többségénél sűrűn fordulnak elő) $m \in O(n)$ miatt $\Theta(n + m) = \Theta(n)$, ami azt jelenti, hogy ritka gráfokra a szomszédossági listás reprezentáció tárigénye aszimptotikusan kisebb, mint a csúcsmátrixosé.

Sűrű gráfoknál viszont $m \in \Theta(n^2)$ miatt $\Theta(n + m) = \Theta(n^2)$, azaz szomszédossági listás és a csúcsmátrixos reprezentáció tárigénye aszimptotikusan ekvivalens.

Teljes gráfoknál a szomszédossági listáknak összesen $n*(n-1)$ elemük van, és egy-egy listaelem sok bitből áll. Teljes vagy közel teljes gráfoknál tehát a szomszédossági listás ábrázolás tényleges tárigénye jelentősen nagyobb lehet, mint a csúcsmátrixosé, ahol a mátrix egy-egy eleme akár egyetlen biten is elfér.

2. Az absztrakt *halmaz*, az absztrakt *sorozat* és az absztrakt *gráf* típus

A halmazok és sorozatok leírásához bevezetünk két típuskonstruktort: Ha $\mathcal{T}$ tetszőleges típus, akkor
- $\mathcal{T}\{\}$ jelöli $\mathcal{T}$ típusú elemek tetszőleges, véges halmazát, és
- $\mathcal{T}\langle\rangle$ jelöli $\mathcal{T}$ típusú elemek tetszőleges, véges sorozatát.

A halmazokra a matematikában szokásos halmazműveleteken kívül még értelmezzük az u **from** S műveletet, ahol S tetszőleges nemüres halmaz. Ennek hatására kiválasztjuk az S halmaz egy tetszőleges elemét, u-nak értékül adjuk, majd eltávolítjuk S-ből. Az üres halmazt önmagában álló $\{\}$ jelöli.

A sorozatokat a matematikában szokásos módon, egytől indexeljük, és az indexet alsó indexként jelöljük, továbbá,
- ha $u, v : \mathcal{T}\langle\rangle$, akkor az $u + v$ kifejezés a konkatenáljukat jelöli;
- $\langle\rangle$ önmagában az üres sorozat.

Mint általában a strukturált típusú változókat, a halmaz és a sorozat típusú változókat is deklarálni kell. Feltesszük, hogy az $s : \mathcal{T}\langle\rangle$ deklaráció hatására s üres sorozattal inicializálódik, illetve a $h : \mathcal{T}\{\}$ deklaráció hatására h üres halmazzal inicializálódik. Ha a zárójelek között megadunk – pontosvesszőkkel elválasztva – néhány $\mathcal{T}$ típusú elemet, akkor a halmaz illetve sorozat a deklaráció kiértékelődése után ezeket fogja tartalmazni.

A gráfok absztrakt algoritmusainak leírásához bevezetjük a $\mathcal{V}$ (vertex, azaz csúcs) absztrakt típust. A $\mathcal{V}$ lesz a gráfok csúcsainak absztrakt típusa. Ez egy olyan elemi típus, amelyben mindegyik csúcshoz tetszőlegesen sok, névvel jelölt címke társítható, és mindegyik címkéhez tartozik valamilyen érték.

Ezek az értékkel bíró címkék tulajdonképpen a csúcsokon értelmezett parciális függvények, amelyek az absztrakt algoritmusokban közönséges értékadó utasításokkal hozhatók létre, és ezekkel is módosíthatók. Ha már léteznek, akkor hatáskörük és láthatóságuk is a teljes absztrakt program, és az absztrakt algoritmus futásának végéig élnek.

A v csúcshoz tartozó *name* címkeértéket $name(v)$ jelöli. Ha tehát végrehajtjuk a $name(v) := x$ értékadást, akkor a v csúcs *name* címkéjének értéke x lesz. Másképp fogalmazva, a *name* címke (azaz a $\mathcal{V}$ halmazon értelmezett parciális függvény) a $name(v) := x$ értékadással a v csúcsnál az x értéket veszi fel.

A $\mathcal{V}$ halmazt az algoritmusok implementációiban legtöbbször az $\mathbb{N}$ halmaz reprezentálja, egy n csúcsú gráf csúcsait pedig egyszerűen az $1..n$ vagy a $0..(n-1)$ halmaz, attól függően, hogy a tömböket egytől vagy nullától kezdve indexeljük. A címkéket ui. gyakran tömbök reprezentálják. Emiatt viszont a láthatóságuk, a hatáskörük és az élettertamuk is korlátozott. Az ebből fakadó problémákat az absztrakt algoritmus implementálásakor kell megoldani.

Az értékkel bíró címkék mellett használni fogunk egyszerű címkéket is, amikor a gráfok csúcsaihoz és/vagy éleihez egyszerű számértékeket vagy neveket társítunk.

Most már minden készen áll az élek ($\mathcal{E}$) és élsúlyozatlan absztrakt gráfok ($\mathcal{G}$) leírásához. Gyakorlati megfontolásból elegendő az egyszerű gráfokra szorítkoznunk, amelyek nem tartalmaznak sem párhuzamos, sem hurokéleket.

$\mathcal{E}$
$+\ u, v : \mathcal{V}$

$\mathcal{G}$
$+\ V : \mathcal{V}\{\}$
$+\ E : \mathcal{E}\{\}$ // $E \subseteq V \times V \setminus \{(u,u) : u \in V\}$ // edges
$+\ A : V \to 2^V$ // $A(u) = \{\, v \in V \mid (u,v) \in E \,\}$
// $A(u) =$ the adjacent vertices of vertex u.

Az egyes gráfalgoritmusoknál a gráf-objektumoknak tipikusan vagy csak az E, vagy csak az A attributumára fogunk hivatkozni. Az A hivatkozások esetén a szomszédossági listás gráfreprezentáció az alapértelmezett (ami azért adott esetben megváltoztatható), míg az E hivatkozások esetében ez további megfontolás tárgya.

3. Elemi gráfalgoritmusok ([2] 22)

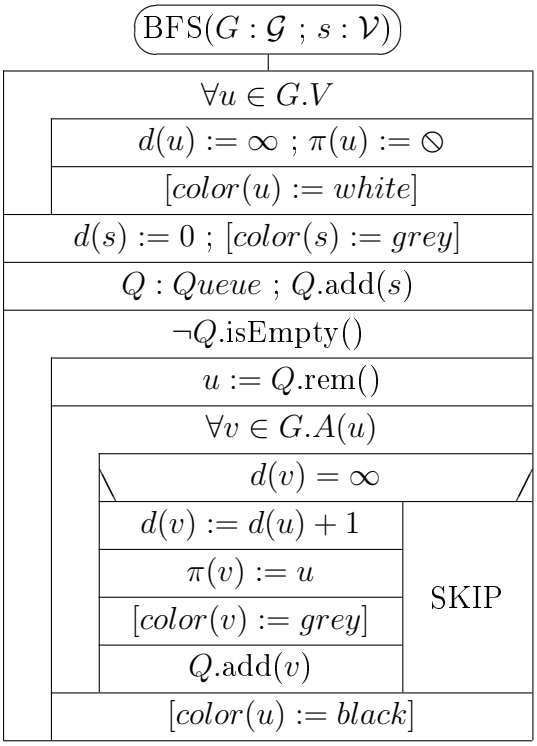
Elemi gráfalgoritmusok alatt élsúlyozatlan gráfokon értelmezett algoritmusokat értünk. Élsúlyozatlan gráfokban tetszőleges út hossza egyszerűen az út mentén érintett élek száma. Az *algoritmusok* témakörben szokásos fogalmak szerint az út tartalmazhat kört. Ha a gráfban tetszőleges u és v csúcsokra az (u, v) élt megkülönböztetjük a (v, u) éltől, *irányított gráfról* beszélünk. Az *irányítatlan gráfokban* viszont ezeket definíció szerint egyenlőknek tekintjük.

Ebben a fejezetben a két alapalgoritmust és ezek néhány alkalmazását tárgyaljuk.

3.1. A szélességi keresés (BFS: Breadth-first Search)

Ezt az algoritmust irányított és irányítatlan gráfokra is értelmezzük. Meghatározzuk a start csúcsból (s) a gráf minden, s-ből elérhető csúcsába a legkevesebb élet tartalmazó utat (ha több ilyen van, akkor az egyiket). Élsúlyozatlan gráfokban ezeket tekintjük az s-ből induló *legrövidebb*, vagy más néven *optimális* utaknak. A csúcsok fontosabb címkéi:
d – a megtalált úttal hány élen keresztül jutunk a csúcsba, és
π – melyik csúcsból jutunk közvetlenül a csúcsba (ki a szülője).
Ha az u csúcs s-ből nem érhető el, akkor $d(u) = \infty$ és $\pi(u) = \oslash$ lesz. Ezenkívül $\pi(s) = \oslash$ is igaz lesz, ami azt jelöli, hogy a legrövidebb $s \rightsquigarrow s$ út csak az s csúcsot tartalmazza, nincs benne él, és ezért s-nek szülője sincs.

Használni fogunk még egy *color* nevű címkét is, aminek az értéke nem befolyásolja a program futását, ezért a rá vonatkozó utasítások az algoritmusból elhagyhatók; csak szemléletes tartalmuk van:
- a fehér csúcsokat a gráfbejárás/keresés még nem találta meg;
- a szürke csúcsokat már megtalálta, de még nem dolgozta fel;
- a fekete csúcsokkal viszont már nincs további tennivalója.



Az első ciklust végrahajtva a gráf minden csúcsára az $d(u) = \infty$ (ami azt jelenti, hogy még egyik csúcsot sem értük el), és ennek megfelelően $\pi(u) = \oslash$ lesz[1].

Az s start (más néven source, azaz forrás) csúcsra azonban tudjuk, hogy $d(s) = 0$ az optimális $s \rightsquigarrow s$ út hossza[2]. Azok a csúcsok, amiket már elértünk, de még a gyerekeiket nem néztük meg, a Q sorba kerülnek, így most az elején s is.

A második, azaz a fő ciklus tehát addig fut, amíg van már elért, de még fel nem dolgozott csúcs. Ez először az $u = s$ csúcsot veszi ki, és mindegyik gyerekére beállítja, hogy s-ből egy lépésben (azaz egyetlen élen keresztül) elérhető, a szülője s,[3] és bekerülnek a sorba.[4]

Most a sorban azok az u csúcsok vannak, akik s-ből egy lépésben elérhetők. A fő ciklus ezeket egyesével kiveszi, megtalálja azokat a v csúcsokat, akik s-ből minimum két lépésben (azaz két élen keresztül) érhetők el (hiszen ezek azoknak a gyerekei, akik egy lépésben elérhetők), beállítja a $d(v) = 2$ és

[1] [valamint $color(u) = white$]

[2] [és ezzel el is kezdtük az s feldolgozását, így $color(s) = grey$ lett]

[3] [szürkére színezi őket]

[4] [Végül s-et feketére színezi, mert már befejezte ezt a csúcsot.]

$\pi(v) = u$ értékeket a szülőjüknek megfelelően,[5] és bekerülnek a sor végére[6]. Ha valamelyik v gyerekcsúcsra az (u, v) él feldolgozásakor már $d(v) \neq \infty$, akkor ez a csúcs már korábbról ismert[7], így $d(v) \in \{0, 1, 2\}$, ezért az újonnan v-be talált út ennél biztosan nem rövidebb, és a BFS algoritmus ennek megfelelően figyelmen kívül is hagyja (ld. az elágazást).

Mire a BFS az összes, s-ből egy lépésben elérhető, azaz $d = 1$ távolságra lévő csúcsot kiveszi a sorból és *kiterjeszti*, azaz a belőle kimenő éleket is feldolgozza, a sorban az s-ből minimum két lépésben elérhető, azaz $d = 2$ távolságra lévő csúcsok maradnak. Miközben ezeket dolgozza fel, azaz kiveszi a sorból és kiterjeszti őket, az s-től $d = 3$ távolságra lévő csúcsok kerülnek a sor végére. Ennélfogva, mire a BFS a feldolgozza az s-től $d = 2$ távolságra levő csúcsokat, a sorban éppen a $d = 3$ távolságra lévők maradnak, és így tovább.

Általában, mikor a sort már az s-től k távolságra levő csúcsok alkotják, megkezdődik azoknak a feldolgozása. Közben az s-től $d = k + 1$ távolságra lévő csúcsokat találjuk meg, beállítjuk a címkéiket és a sor végére tesszük őket. Mire az s-től k távolságra levő csúcsokat feldolgozzuk, a sort már az s-től $k + 1$ távolságra levő csúcsok alkotják stb.

Azt mondjuk, hogy az s-től k távolságra levő csúcsok vannak a gráf k-adik szintjén. Így a BFS a gráfot szintenként járja be, először a nulladik szintet, aztán az elsőt, majd a másodikat stb. Minden szintet teljesen feldolgoz, mielőtt a következőre lépne, közben pedig éppen a következő szinten levő csúcsokat találja meg. Innét jönnek a *szélességi bejárás* és a *szélességi keresés* elnevezések.

Mivel a gráf véges, végül nem lesz már $k + 1$ távolságra levő csúcs, Q kiürül és a BFS megáll. Azokat a csúcsok, amelyek s-ből elérhetők, az algoritmus valamelyik szinten meg is találja, és megfelelően beállítja a d és a π címkéiket is. A többi csúcs tehát s-ből nem érhető el. Ezekre $d = \infty$ és $\pi = \oslash$ marad.[8]

3.1. Feladat. *A BFS algoritmusában milyen, az elágazás „$d(v) = \infty$" feltételével ekvivalens feltételt tudnánk adni? Miért?*

3.1.1. A szélességi fa (Breadth-first Tree)

A BFS minden s-ből elérhető, de tőle különböző csúcsra, annak π címkéjével hivatkozik annak szülőjére, az s-ből a csúcsba vezető (a BFS által megha-

[5] [szürkére színezi őket]

[6] [majd a szülőjüket feketére színezi, mert azt már befejezte]

[7] [már nem fehér, hanem szürke vagy fekete]

[8] [Az s-ből elérhető csúcsok tehát végül feketék lesznek, míg a többiek fehérek maradnak.]

tározott) legrövidebb úton. Természetesen több csúcsnak is lehet ugyanaz a szülője, a szülőcsúcs viszont a BFS által meghatározott legrövidebb utakon egyértelmű.

Az s-ből elérhető csúcsok π hivatkozásai tehát egy általános fát definiálnak, aminek a gyökere s, és ennek megfelelően $\pi(s) = \oslash$. Ezt a fát *szélességi fának* és *legrövidebb utak fájának* is nevezzük, mivel – fordított irányban – mindegyik, az s-ből elérhető csúcsra a BFS által meghatározott legrövidebb utakat tartalmazza.

Világos, hogy ez a fordított ábrázolás azért célszerű, mert minden csúcsnak legfeljebb egy szülője, viszont több gyereke is lehet, így tehát tömörebb ábrázolás érhető el.

3.2. Feladat. *Tegyük fel, hogy a G gráfra már lefutott a szélességi BFS(G, s) algoritmus. Írja meg a printShortestPathTo(v)* rekurzív *eljárást, ami kiírja az s-ből v-be vezető (a BFS által kiszámolt) legrövidebb utat, feltéve, hogy s-ből v elérhető!*

Vegyük észre, hogy az előbbi előfeltétellel nincs szükségünk az s paraméterre! Az algoritmus ne használjon segéd adatszerkezetet!

$MT(d) \in \Theta(d)$, *ahol* $d = d(v)$.

3.3. Feladat. *Tegyük fel, hogy a G gráfra már lefutott a szélességi BFS(G, s) algoritmus. Írja meg a printShortestPathTo(v)* nemrekurzív *eljárást, ami kiírja az s-ből v-be vezető (a BFS által kiszámolt) legrövidebb utat, feltéve, hogy s-ből v elérhető!*

Vegyük észre, hogy az előbbi előfeltétellel nincs szükségünk az s paraméterre! Milyen adatszerkezettel váltotta ki a rekurziót?

$MT(d) \in \Theta(d)$, *ahol* $d = d(v)$.

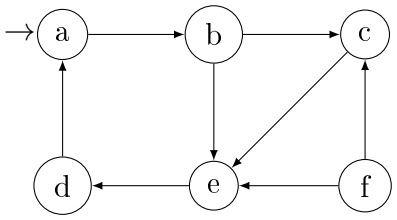
3.4. Feladat. *Tegyük fel, hogy a G gráfra már lefutott a szélességi BFS(G, s) algoritmus. Írja meg a printShortestPath(s, v) eljárást, ami kiírja az s-ből v-be vezető (a BFS által kiszámolt) legrövidebb utat, ha s-ből v elérhető! Különben a kiírás azt adja meg, hogy melyik csúcsból melyik nem érhető el!*

$MT(d) \in \Theta(d)$, *ahol* $d = d(v)$, *ha s-ből v elérhető; különben* $d = 1$.

3.1.2. A szélességi keresés szemléltetése

A 6. ábrán látható gráfra szemléltetjük a BFS működését, az alábbi táblázat segítségével, ahol most „a” a start csúcs. Az új csúcsok természetesen mindig a sor végére kerülnek.

Most is és a későbbiekben is, *indeterminisztikus esetekben* a kisebb indexű csúcsot részesítjük előnyben. (Ez a leggyengébb szabály, de a zárthelyiken,

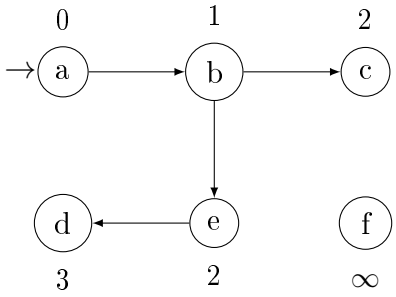


a → b.
b → c ; e.
c → e.
d → a.
e → d.
f → c ; e.

6. ábra. Ugyanaz az irányított gráf grafikus (balra) és szöveges (jobbra) ábrázolással (s = a).

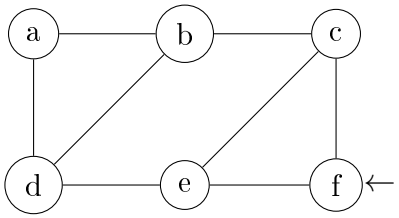
vizsgákon is elvárjuk, hogy tartsák be.) Alább pl. ez akkor érvényesül, amikor a „b” csúcs gyerekeit „c,e” sorrendben tesszük be a sorba.

d változásai						kiter-	Q :	π változásai					
a	b	c	d	e	f	jesztett	Queue	a	b	c	d	e	f
0	∞	∞	∞	∞	∞	csúcs : d	$\langle$ a $\rangle$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$
	1					a:0	$\langle$ b $\rangle$		a				
		2		2		b:1	$\langle$ c, e $\rangle$			b		b	
						c:2	$\langle$ e $\rangle$						
			3			e:2	$\langle$ d $\rangle$				e		
						d:3	$\langle\ \rangle$						
0	1	2	3	2	∞	végső d és π értékek		$\oslash$	a	b	e	b	$\oslash$



7. ábra. A 6. ábráról ismerős gráf szélességi fája, ha s = a.

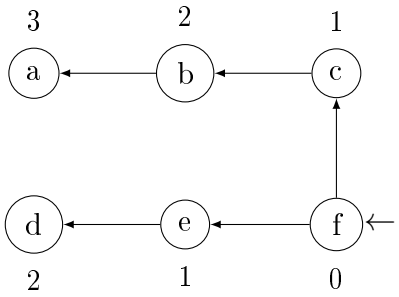
Most pedig a 8. ábrán látható gráfra szemléltetjük a BFS működését, az alábbi táblázat segítségével, ahol most „f” a start csúcs.



a – b ; d.
b – c ; d.
c – e ; f.
d – e.
e – f.

8. ábra. Ugyanaz az irányítatlan gráf grafikus (balra) és szöveges (jobbra) ábrázolással ($s = \mathrm{f}$).

d változásai						kiter-	Q :	π változásai					
a	b	c	d	e	f	jesztett	Queue	a	b	c	d	e	f
∞	∞	∞	∞	∞	0	csúcs : d	$\langle$ f $\rangle$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$
		1		1		f:0	$\langle$ c, e $\rangle$			f		f	
	2					c:1	$\langle$ e, b $\rangle$		c				
			2			e:1	$\langle$ b, d $\rangle$				e		
3						b:2	$\langle$ d, a $\rangle$	b					
						d:2	$\langle$ a $\rangle$						
						a:3	$\langle$ $\rangle$						
3	2	1	2	1	0	végső d és π értékek		b	c	f	e	f	$\oslash$



9. ábra. A 8. ábráról ismerős gráf szélességi fája, ha $s = \mathrm{f}$.

3.1.3. A szélességi keresés hatékonysága

A gráfalgoritmusokban a továbbiakban is a szokásos $n = |G.V|$ és $m = |G.E|$ jelöléseket használjuk.

Az első, az inicializáló ciklus n-szer iterál. A második, a fő ciklus annyiszor iterál, ahány csúcs elérhető s-ből (önmagát is számítva), ami legfeljebb szintén n, minimum 1.

Ennek megfelelően, irányított gráfoknál a belső ciklus legfeljebb m-szer iterál összesen (ha s-ből mindegyik csúcs elérhető, akkor minden él sorra kerül), irányítatlan gráfoknál pedig belső ciklus legfeljebb $2m$-szer iterál összesen (ha s-ből mindegyik csúcs elérhető, akkor minden él sorra kerül mindkét irányból). Az is lehetséges viszont, hogy a belső ciklus egyszer sem iterál (ha s-ből nem megy ki egyetlen él sem).

Összefoglalva: $MT(n,m) \in \Theta(n+m)$ és $mT(n,m) \in \Theta(n)$.

3.1.4. A szélességi keresés implementációja szomszédossági listás és szomszédossági mátrixos gráfábrázolás esetén

A $G = (V,E)$ gráfról föltesszük, hogy $V = \{v_1, \cdots, v_n\}$, ahol $n \in \mathbb{N}$, azaz a gráf csúcsait egyértelműen azonosítják az $1..n$ sorszámok. Az absztrakt v_i csúcsokat úgy ábrázoljuk, hogy d és π címkéinek megfeleltetjük a $d/1, \pi/1 : \mathbb{N}[n]$ tömböket, ahol $d(v_i)$ reprezentációja $d[i]$ és $\pi(v_i)$ reprezentációja $\pi[i]$. A *color* címkék reprezentálása felesleges. A $\oslash$ reprezentációja lehet például a 0 számérték: pl. $\pi[s] = 0$ absztrakt jelentése $\pi(v_s) = \oslash$. Mivel a szélességi keresésnél n csúcsú gráfon tetszőleges csúcshoz vezető út hossza legfeljebb $n-1$, ami egyben a véges d-értékek maximuma, a ∞ helyett használhatjuk pl. az n számot.

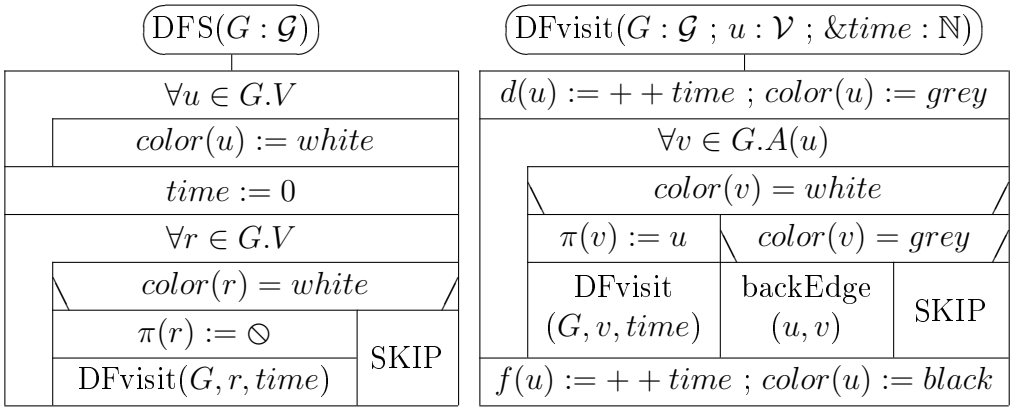
3.5. Feladat. *Írja meg a BFS($A/1 : Edge*$*[n] ; s : 1..n ; d/1, \pi/1 : \mathbb{N}[n]$) eljárást, ami a szélességi keresés implementációja szomszédossági listás (1.6) gráfábrázolás esetére. Ügyeljen arra, hogy az absztrakt algoritmus hatékonyságának aszimptotikus nagyságrendje megmaradjon!*

3.6. Feladat. *Írja meg a BFS($A/1 : bit[n,n]$; $s : 1..n$; $d/1, \pi/1 : \mathbb{N}[n]$) eljárást, ami a szélességi keresés implementációja szomszédossági mátrixos (1.5) gráfábrázolás esetére. Tartható-e az absztrakt algoritmus hatékonyságának aszimptotikus nagyságrendje? Ha nem, hogyan változik?*

3.2. A mélységi keresés (DFS: Depth-first Search)

Mélységi bejárásnak (Depth-first Traversal) vagy mélységi gráfbejárásnak is nevezik, mivel a gráf összes csúcsát és élét érinti.

Ebben a jegyzetben csak egyszerű irányított gráfokra értelmezzük. Fehér csúcs: érintetlen, szürke csúcs: belőle elérhető csúcsokat járunk be éppen, fekete csúcs: befejeztük.



$d(u)$: elérési idő (discovery time) / $f(u)$: befejezési idő (finishing time)

3.2.1. Mélységi feszítő erdő (Depth-first forest)

Mindegyik, a DFS eljárásból indított DFvisit egy-egy *mélységi fát* számol ki. Ezek együtt adják a *mélységi feszítő erdőt.*

$r \in G.V$ egy mélységi fa gyökere $\iff \pi(r) = \oslash$
$(u, v) \in G.E$ egy mélységi fa éle $\iff u = \pi(v)$

3.2.2. Az élek osztályozása (Classification of edges)

3.7. Definíció. *A gráf éleinek osztályozása:*

(u, v) fa-él (tree edge) $\iff$ (u, v) *valamelyik mélységi fa egyik éle. (A fa-élek mentén járjuk be a gráfot.)*

(u, v) vissza-él (back edge) $\iff$ v *az* u *őse egy mélységi fában.*

(u, v) előre-él (forward edge) $\iff$ (u, v) *nem fa-él, de* v *az* u *leszármazottja egy mélységi fában.*

(u, v) kereszt-él (cross edge) $\iff$ u *és* v *két olyan csúcs, amelyek ugyanannak a mélységi fának két különböző ágán vannak, vagy két különböző mélységi fában találhatók.*

3.8. Tétel. *A gráf éleinek felismerése:*
A mélységi bejárás során tetszőleges (u, v) *él feldolgozásakor az él a következő kritériumok alapján osztályozható.*

(u, v) fa-él (tree edge) $\iff$ *a* v *csúcs még fehér.*

(u, v) vissza-él (back edge) $\iff$ *a* v *csúcs éppen szürke.*

(u, v) előre-él (forward edge) $\iff$ *a v csúcs már fekete* $\wedge\ d(u) < d(v)$.

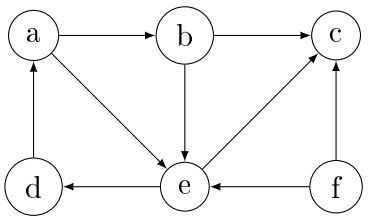
(u, v) kereszt-él (cross edge) $\iff$ *a v csúcs már fekete* $\wedge\ d(u) > d(v)$.

3.9. Feladat. *A mélységi bejárás során miért nem fordulhat elő, hogy az (u, v) él feldolgozásakor $d(u) = d(v)$? Milyen feltételekkel fordulhatna ez elő? Hogyan kellene ekkor kiegészíteni a 3.7. definíciót úgy, hogy a 3.8. tétel megfogalmazásán ne kelljen módosítani?*

3.2.3. A mélységi bejárás szemléltetése

A 10. ábrán látható gráf mélységi bejárását szemléltetjük. A bejárás indeterminisztikus abban, hogy az egyes ciklusokban a csúcsokat milyen sorrendben veszi sorra. Ezt a szemléltetésben úgy fogjuk feloldani, hogy a csúcsokat ábécé rendben, illetve indexek szerint monoton növekvően dolgozzuk fel. (Ennek a konvenciónak a betartása az összes gráfalgoritmus esetében, a zh-kon és a vizsgákon is elvárás.)

A csúcsok felett illetve alatt olvasható d/f alakú címkék a csúcs *elérési/befejezési* idejét (*discovery/finishing* time) mutatják. A fa-éleket dupla szárú nyillal, a nemfa-éleket pedig a megfelelő címkékkel jeleztük: V:visszaél, E:előre-él, K:kereszt-él.

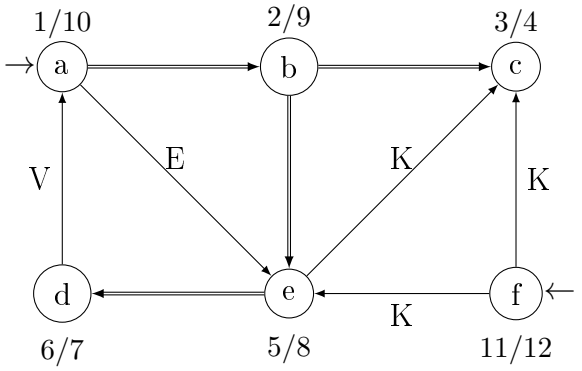


a → b ; e.
b → c ; e.
d → a.
e → c ; d.
f → c ; e.

10. ábra. Ugyanaz az irányított gráf grafikus (balra) és szöveges (jobbra) ábrázolással.

A 11. ábrán követhető a gráf mélységi bejárása. A bejárás eredménye, a mélységi feszítő erdő két mélységi fából áll: az egyiknek az **a** csúcs a gyökere és elérési sorrendben **b, c, e** és **d** a további csúcsai, a másik csak az **f** gyökércsúcsból áll.

A mélységi bejárás (DFS) az első mélységi vizittel (DFvisit) és ennek megfelelően az első mélységi fa építésével kezdődik. [Nemüres gráf mélységi bejárása mindig egy vagy több mélységi vizitből áll, és ennek megfelelő számú mélységi fát épít fel: ezekből áll majd a mélységi feszítő erdő. Ennek fái lefedik az összes csúcsot, és egymástól diszjunktak.]



11. ábra. A 10. ábrán látható gráf mélységi bejárása.

Alfabetikus konvenciónk alapján az első mélységi vizitet és egyben az első mélységi fa építését az **a** csúcsnál, mint gyökércsúcsnál kezdjük $time = 1$ idővel. Az idő számláló mindig akkor lép egyet, amikor elérünk, vagy befejezünk egy csúcsot. Azok a csúcsok fehérek, amelyeket még nem értünk el. Azok szürkék, amelyeket már elértünk, de még nem fejeztünk be. Azok feketék, amelyeket már befejeztünk. Először tehát az összes csúcs fehér volt, de most az **a** csúcsnak beállítottuk az elérési idejét és szürkére színeztük. Az **a** csúcs gyerekei a **b** és az **e** csúcsok. Alfabetikus konvenciónkat követve a **b** felé megyünk tovább. A **b** csúcs még fehér, így az **(a,b)** él fa-él lesz. (Ld. a 3.8. tételt!) Ha kijelölünk egy fa-élt, mindig tovább is lépünk abba a csúcsba, amibe mutat.

Most $time = 2$ idővel elértük a **b** csúcsot, beállítjuk az elérési idejét és szürkére színezzük. Gyerekei a **c** és **e** csúcsok. A **c** csúcs felé folytatjuk a bejárást. Mivel **c** még fehér, **(b,c)** fa-él lesz.

Most $time = 3$ idővel elértük a **c** csúcsot, beállítjuk az elérési idejét és szürkére színezzük. Mivel nincs gyereke, $time = 4$ idővel befejezzük, beállítjuk a befejezési idejét és feketére színezzük, majd visszalépünk a bele mutató fa-élen az éppen épülő mélységi fában a szülőjébe, ami a **b** csúcs.

A **b** csúcsnak van még egy címkézetlen kimenő éle, a **(b,e)**. Ez fehér csúcsba mutat, így **(b,e)** fa-él lesz.

Most $time = 5$ idővel elértük az **e** csúcsot, beállítjuk az elérési idejét és szürkére színezzük. Két kimenő élünk van, az **(e,c)** és az **(e,d)**. Először az **(e,c)** élet dolgozzuk fel. Ez fekete (azaz befejezett) csúcsba mutat, aminek az elérési ideje kisebb, mint az **e** csúcsé, így **(e,c)** élet kereszt-élnek címkézzük. (Ld. a 3.8. tételt!) Ha nemfa-élt találunk, sosem lépünk tovább az általa hi-

vatkozott csúcsba. [A kereszt-élnek és az előre-élnek megfelelő címkézést nem találhatjuk meg az algoritmus struktogramjában: ez csak a szemléltetéshez tartozik.] Másodszor az **(e,d)** élet dolgozzuk fel. Ez fehér csúcsba mutat, így az **(e,d)** fa-él lesz.

Most $time = 6$ idővel elértük a **d** csúcsot, beállítjuk az elérési idejét és szürkére színezzük. A **d** csúcsnak egy kimenő éle van, a **(d,a)**. Ez szürke, azaz elért, de még befejezetlen csúcsba mutat, így **(d,a)** vissza-él lesz. (Ld. a 3.8. tételt!) [Vegyük észre, hogy ha vissza-élt találunk, egyúttal irányított kört is találtunk a gráfban! (Ld. a 3.7. definíciót!)]

A **d** csúcsnak nincs több kimenő éle, így $time = 7$ idővel befejezzük, beállítjuk a befejezési idejét és feketére színezzük, majd visszalépünk a bele mutató fa-élen az éppen épülő mélységi fában a szülőjébe, ami az **e** csúcs.

Az **e** csúcsnak sincs már címkézetlen, azaz feldolgozatlan kimenő éle, így $time = 8$ idővel befejezzük, beállítjuk a befejezési idejét és feketére színezzük, majd visszalépünk a bele mutató fa-élen az éppen épülő mélységi fában a szülőjébe, ami a **b** csúcs.

A **b** csúcsnak sincs már címkézetlen kimenő éle, így $time = 9$ idővel befejezzük, beállítjuk a befejezési idejét és feketére színezzük, majd visszalépünk a bele mutató fa-élen az éppen épülő mélységi fában a szülőjébe, ami az **a** csúcs.

Az **a** csúcsnak még címkézetlen az **(a,e)** kimenő éle, ami az **e** fekete csúcsba mutat. Ennek az elérési ideje nagyobb, mint az **a** csúcsnak, tehát az **(a,e)** élt előre-élnek címkézzük. (Ld. a 3.8. tételt!)

Az **a** csúcsnak nincs már címkézetlen, azaz feldolgozatlan kimenő éle, így $time = 10$ idővel befejezzük, beállítjuk a befejezési idejét, feketére színezzük, és ezzel befejezzük az **a** gyökércsúcsú mélységi fa építését.

Ezzel visszalépünk a DFvisit eljárásból a DFS eljárásba. Újabb fehér csúcsot keresünk. [Vegyük észre, hogy ezen a ponton csak fehér és fekete csúcsokat találhatunk, szürkéket nem!] Sorra vesszük a **b, c, d, e** csúcsokat, amelyek mindegyike fekete már, majd megtaláljuk az **f** csúcsot, ami még fehér. Ezt beállítjuk a második mélységi fa gyökércsúcsának, majd innét indítjuk a következő mélységi vizitet.

Most $time = 11$ idővel elértük az **f** csúcsot, beállítjuk az elérési idejét és szürkére színezzük. Az **f** csúcsnak két kimenő éle van, az **(f,c)** és az **(f,e)**. Először az **(f,c)** élet dolgozzuk fel. Ez fekete (azaz befejezett) csúcsba mutat, aminek az elérési ideje kisebb, mint az **f** csúcsé, így az **(f,c)** élet kereszt-élnek címkézzük. Hasonlóan járunk el az **(f,e)** éllel.

Ezután az **f** csúcsnak sincs már címkézetlen, azaz feldolgozatlan kimenő éle, így $time = 12$ idővel befejezzük, beállítjuk a befejezési idejét, feketére színezzük, és ezzel befejezzük az **f** gyökércsúcsú mélységi fa építését.

Visszalépünk a DFvisit eljárásból a DFS eljárásba. Újabb fehér csúcsot

keresünk, de nincs már több csúcs, amit megvizsgálhatnánk. Befejeződik tehát a mélységi bejárás, és vele együtt a mélységi feszítő erdő létrehozása.

3.2.4. A mélységi keresés futási ideje

A szokásos $n = |G.V|$ és $m = |G.E|$ jelölésekkel a DFS mindkét ciklusa n-szer fut le. Mindegyik csúcsra, amikor először érjük el, azaz, amikor még fehér, pontosan egyszer, összesen n-szer hívódik meg a DFvisit rekurzív eljárás. A DFvisit ciklusa mindegyik csúcsra annyit iterál, amennyi a kimeneti fokszáma. Ez a ciklus tehát a gráf éleit dolgozza fel, mindegyiket egyszer. Ennélfogva összesen m iterációt végez. A DFS csak egyszer hívódik meg. Feltesszük most, hogy a backEdge eljárás mindegyik végrehajtása csak megjelöl egy vissza-élet, így $\Theta(1)$ műveletigényű, és műveletigénye hozzávehető az őt meghívó ciklusiterációéhoz.

Pontosan $3n+m+1$ lépést számolhatunk össze. Ebből a mélységi keresésre $MT(n), mT(n) \in \Theta(n+m)$.

3.2.5. A DAG tulajdonság eldöntése

3.10. Definíció. *A G irányított gráf akkor DAG (Directed Acyclic Graph = körmentes irányított gráf), ha nem tartalmaz irányított kört.*

A DAG-ok a gráfok fontos osztályát képezik: sok hasznos algorimus DAG-ot vár a bemenetén, így az input ellenőrzése is szükséges lehet. (Ld. pl. a *topologikus rendezést* (3.2.6) és a *DAG legrövidebb utak egy forrásból* algoritmust!)

Világos, hogy amennyiben a mélységi bejárás egy (u, v) vissza-élet talál, azzal irányított kört is talált a gráfban, mert ekkor a vissza-él a definíciója szerint egy mélységi fában az u csúcs egyik őse a v csúcs, és a v-ből u-ba vezető, fa-élekből álló út az (u, v) éllel együtt irányított kört alkot.

Bebizonyítható, hogy ha a G irányított gráf nem DAG, azaz irányított kört tartalmaz, akkor a DFS fog vissza-élet, és ezzel irányított kört találni [2]. [Az viszont nem garantált, hogy az összes irányított kört megtalálja. Könnyű olyan gráfot találni, ahol nem találja meg az összes irányított kört, mert ugyanaz a vissza-él több irányított körnek is a része lehet.]

3.11. Feladat. *Rajzoljon olyan három csúcsú, négy élű egyszerű gráfot, amelyben két egyszerű irányított kör van, és a DFS lehet, hogy megtalálja mindkettőt, de lehet, hogy csak az egyiket! Indokolja is az állítását!*

A fentiekből adódik a következő tétel.

3.12. Tétel. *A G irányított gráf DAG $\iff$ a mélységi bejárás nem talál G-ben vissza-élet.*

Ha a DFS talál egy (u, v) vissza-élet, akkor az $\langle u, \pi(u), \pi(\pi(u)), \ldots, v, u\rangle$ csúcssorozat visszafelé olvasva egyszerű irányított kört ad.

A fenti tétel alapján a backEdge eljárás akár ki is nyomtathatja a megtalált irányított kört. Ebben az esetben a maximális futási ideje nyilván $\Theta(n)$ lesz. (Szélsőséges esetben a megtalált irányított körök kinyomtatása akár meg is növelheti a mélységi bejárás műveletigényének aszimptotikus nagyságrendjét.)

3.13. Feladat. *Írja meg a backEdge(u, v) eljárás struktogramját abban az esetben, ha ennek feladata a megtalált irányított kör explicit, jól olvasható megjelenítése is! Ügyeljen a $\Theta(n)$ maximális műveletigényre!*

3.14. Feladat. *Adja meg irányított gráfok egy olyan sorozatát, amelyekre $m \in O(n)$, és így alapesetben $MT_{DFS}(n, m) \in \Theta(n)$, de ha a backEdge eljárásba az irányított körök kinyomtatását is beleértjük, akkor ezen a gráfsorozaton a DFS maximális műveletigénye $\Omega(n^2)$ lesz! Indokolja is az állítását!*

3.15. Feladat. *Módosítsa a mélységi bejárás algoritmusát úgy, hogy irányítatlan gráfokon tudjunk vele kört keresni! Hogyan ismerjük fel a vissza-éleket? [Irányítatlan gráfban, ha (u, v) fa-él, világos, hogy (v, u) nem lehet vissza-él, mert $(v, u) = (u, v)$.] Mi a helyzet az előre- és a kereszt-élekkel?*

3.2.6. Topologikus rendezés

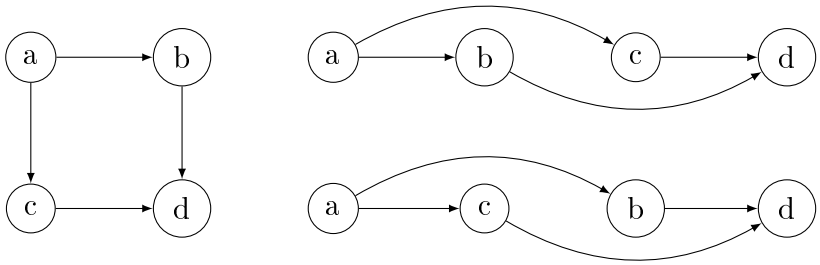
3.16. Definíció. *Irányított gráf topologikus rendezése alatt a gráf csúcsainak olyan sorba rendezését értjük, amelyben minden él egy-egy később jövő csúcsba (szemléletesen: balról jobbra) mutat.*

Ugyanannak a gráfnak lehet egynél több topologikus rendezése, mint a 12. ábra mutatja.

3.17. Tétel. *Tetszőleges irányított gráfnak pontosan akkor van topologikus rendezése, ha nincs irányított kör a gráfban, azaz a gráf DAG.*

Bizonyítás.

$\Rightarrow$ Ha van irányított kör a gráfban, jelölje $\langle u_1, u_2, \ldots, u_k, u_1\rangle$! Ekkor egy tetszőleges topologikus rendezésben u_1 után jön valahol u_2, az után valahol u_3, és így tovább, végül is u_1 után jön u_k, és u_k után u_1, ami ellentmondás. Tehát ekkor nincs topologikus rendezés (a gráf csúcsain).



12. ábra. Ugyanaz a DAG háromféleképpen lerajzolva: balra a szokásos módon ábrázolva, jobbra pedig a csúcsokat kétféleképpen, topologikus rendezésüknek megfelelően sorba rakva.

⇐ Ha nincs irányított kör a gráfban, akkor nyilván van olyan csúcs, aminek nincs megelőzője. Ha veszünk egy megelőzővel nem rendelkező csúcsot, és töröljük a gráfból, akkor a maradék gráfban nem keletkezik irányított kör, lesz megint legalább egy olyan, amelyiknek nincs megelőzője. Sorban a törölt csúcsok adják a topologikus rendezést.

□

A topologikus rendezést végrehajthatjuk például az irányított gráf mélységi bejárása segítségével.

Tetszőleges DAG-ra a topologikus rendezés algoritmusa:

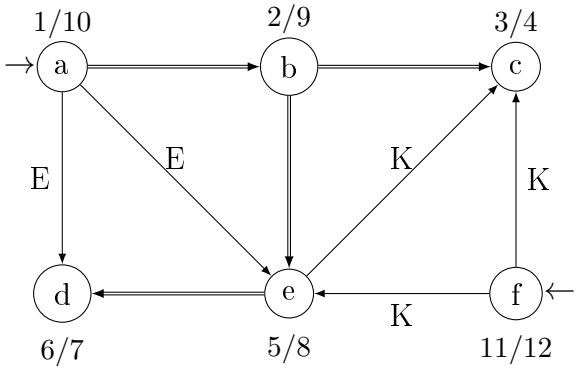
Először létrehozunk egy üres vermet, majd végrehajtuk gráf mélységi bejárását úgy, hogy valahányszor befejezünk egy csúcsot, a verem tetejére tesszük. Végül a verem tartalmát kiolvasva megkapjuk a gráf csúcsainak topologikus rendezését.

Vegyük észre, hogy ez az algoritmus képes elenőrizni a saját előfeltételét! Ha ugyanis a DFS vissza-élt talál, akkor a gráf irányított kört tartalmaz, és az algoritmus eredménye az, hogy nincs topologikus rendezés. (Ilyenkor a verem tartalma nem használható fel.)

Az algoritmus végrehajtására a 13. ábrán láthatunk egy példát.

A topologikus rendezés egy kézenfekvő alkalmazása az ún. *egygépes ütemezési probléma* megoldása: A csúcsok (munka)folyamatok, az élek a köztük lévő rákövetkezési kényszerek, és a folyamatokat ezeknek megfelelően kell sorba rakni.

3.18. Feladat. *Írja meg (1) a mélységi keresés és (2) a topologikus rendezést struktogramját (A) szomszédossági listás és (B) szomszédossági mátrixos gráfábrázolások esetén! Mit tud mondani az algoritmusok hatékonyságáról?*



13. ábra. A DAG topologikus rendezésésében a csúcsok a befejezési idők szerint szigorúan monoton csökkenően jelennek meg. Így a fenti gráfra a DFS a szokásos alfabetikus konvencióval a következő topologikus rendezést adja: $\langle f, a, b, e, d, c\rangle$. Vegyük észre, hogy ha az algoritmus indeterminizmusát más konvenció mentén oldanánk fel, gyakran az előbbitől különböző topologikus rendezést kapnánk!

3.19. Feladat. *Vegyük észre, hogy a 3.17. tétel bizonyításának második fele algoritmusként is értelmezhető! Írja meg ennek alapján a topologikus rendezés egy, a DFS-től független struktogramját! Hogyan lehetne az input gráf lebontását $O(n)$ munkatár segítségével elkerülni? Mit tud mondani az algoritmus hatékonyságáról? Ez az algoritmus hogyan fog viselkedni, ha a gráf irányított kört tartalmaz?*

Algoritmusok és adatszerkezetek II. előadásjegyzet

Élsúlyozott gráfok és algoritmusaik

Ásványi Tibor – asvanyi@inf.elte.hu

2023. augusztus 28.

Tartalomjegyzék

Hivatkozások

[1] Ásványi Tibor, Algoritmusok és adatszerkezetek II.
Útmutatások a tanuláshoz, jelölések, tematika,
fák, gráfok,
mintaillesztés, tömörítés
http://aszt.inf.elte.hu/~asvanyi/ad/ad2jegyzet/

[2] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.,
magyarul: Új Algoritmusok, *Scolar Kiadó*, Budapest, 2003.
ISBN 963 9193 90 9
angolul: Introduction to Algorithms (Third Edititon),
The MIT Press, 2009.

[3] Fekete István, Algoritmusok jegyzet
http://ifekete.web.elte.hu/

[4] Rónyai Lajos – Ivanyos Gábor – Szabó Réka, Algoritmusok,
TypoTEX Kiadó, 1999. ISBN 963 9132 16 0
https://www.tankonyvtar.hu/hu/tartalom/tamop425/2011-0001-526_ronyai_algoritmusok/adatok.html

[5] Tarjan, Robert Endre, Data Structures and Network Algorithms,
CBMS-NSF Regional Conference Series in Applied Mathematics, 1987.

[6] Weiss, Mark Allen, Data Structures and Algorithm Analysis,
Addison-Wesley, 1995, 1997, 2007, 2012, 2013.

[7] Ásványi Tibor, Algoritmusok és adatszerkezetek I. előadásjegyzet (2021)
http://aszt.inf.elte.hu/~asvanyi/ad/ad1jegyzet/ad1jegyzet.pdf

[8] Ásványi Tibor, Detecting negative cycles with Tarjan's breadth-first scanning algorithm (2016)
http://ceur-ws.org/Vol-2046/asvanyi.pdf

1. Élsúlyozott gráfok és ábrázolásaik ([2] 22)

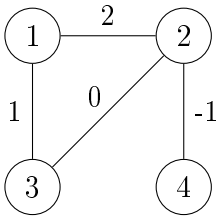
1.1. Definíció. Élsúlyozott gráf *alatt egy* $G = (V, E)$ *gráfot értünk a* $w : E \to \mathbb{R}$ *súlyfüggvénnyel, ahol* V *a csúcsok (vertices) tetszőleges, véges halmaza,* $E \subseteq V \times V \setminus \{(u, u) : u \in V\}$ *az élek (edges) halmaza.*

A $w : E \to \mathbb{R}$ *minden egyes élhez hozzárendeli annak* súlyát, *más néven* hosszát, *illetve* költségét. *(Az* élsúly, élhossz *és* élköltség *elnevezések szinonímák.)*

1.2. Definíció. *Élsúlyozott gráfban tetszőleges út* hossza, *más néven* költsége, *illetve* súlya *az út mentén található élek összsúlya. Hasonlóképpen tetszőleges gráf/fa súlya az élei súlyainak összege.*

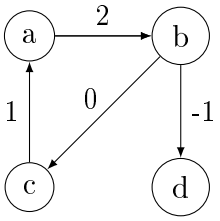
1.1. Grafikus ábrázolás

Az éleket súlyukkal címkézzük.



1 – 2, 2 ; 3, 1.
2 – 3, 0 ; 4, -1.

1. ábra. Ugyanaz az élsúlyozott irányítatlan gráf grafikus (balra) és szöveges (jobbra) ábrázolással.



a → b, 2.
b → c, 0 ; d, -1.
c → a, 1.

2. ábra. Ugyanaz az élsúlyozott irányított gráf grafikus (balra) és szöveges (jobbra) ábrázolással.

1.2. Szöveges ábrázolás

Az irányítatalan gráfoknál „$u - v_{u_1}, w_{u_1}; \cdots ; v_{u_k}, w_{u_k}$." azt jelenti, hogy $(u, v_{u_1}), \cdots , (u, v_{u_k})$ élei a gráfnak, sorban $w(u, v_{u_1}) = w_{u_1}, \cdots , w(u, v_{u_k}) = w_{u_k}$ súlyokkal. (Ld. az 1. ábrát!)

Az irányított gráfoknál pedig „$u \to v_{u_1}, w_{u_1}; \cdots ; v_{u_k}, w_{u_k}$." azt jelenti, hogy a gráfban az u csúcsból az $(u, v_{u_1}), \cdots , (u, v_{u_k})$ irányított élek indulnak ki, most is sorban $w(u, v_{u_1}) = w_{u_1}, \cdots , w(u, v_{u_k}) = w_{u_k}$ súlyokkal. (Ld. a 2. ábrát!)

1.3. Szomszédossági mátrixos (adjacency matrix), más néven csúcsmátrixos reprezentáció

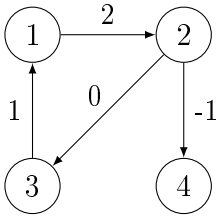
A szomszédossági mátrixos, vagy más néven csúcsmátrixos ábrázolásnál a $G = (V, E)$ gráfot a $w : E \to \mathbb{R}$ súlyfüggvénnyel ($V = \{v_1, \cdots , v_n\}$) egy $A/1 : \mathbb{R}_\infty[n, n]$ mátrix reprezentálja, ahol $n = |V|$ a csúcsok száma, $1..n$ a csúcsok sorszámai, és tetszőleges $i, j \in 1..n$ csúcssorszámokra

$$A[i, j] = w(v_i, v_j) \iff (v_i, v_j) \in E$$

$$A[i, i] = 0$$

$$A[i, j] = \infty \iff (v_i, v_j) \notin E \wedge i \neq j$$

A 3. ábrán látható irányított gráfot például a mellette lévő szomszédossági mátrix reprezentálja.



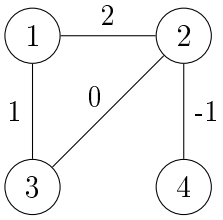
A	1	2	3	4
1	0	2	∞	∞
2	∞	0	0	-1
3	1	∞	0	∞
4	∞	∞	∞	0

3. ábra. Ugyanaz az élsúlyozott, irányított gráf grafikus (balra) és szomszédossági mátrixos (jobbra) ábrázolással.

A főátlóban mindig nullák vannak, mert csak egyszerű gráfokkal foglalkozunk (amelyekben nincsenek hurokélek), és tetszőleges csúcsból önmaga közvetlenül, nulla költségű úton érhető el.

Vegyük észre, hogy irányítatlan esetben a szomszédossági mátrixos reprezentáció mindig szimmetrikus, ui. $(v_i, v_j) \in E$ esetén $(v_j, v_i) = (v_i, v_j) \in E$.

Az 1. ábráról már ismerős irányítatlan gráf csúcsmátrixos ábrázolása a 4. ábrán látható.



A	1	2	3	4
1	0	2	1	∞
2	2	0	0	-1
3	1	0	0	∞
4	∞	-1	∞	0

4. ábra. Ugyanaz az élsúlyozott, irányítatlan gráf grafikus (balra) és szomszédossági mátrixos (jobbra) ábrázolással.

1.4. Szomszédossági listás (adjacency list) reprezentáció

A szomszédossági listás ábrázolás hasonlít a szöveges reprezentációhoz. A $G = (V, E)$ gráfot a $w : E \to \mathbb{R}$ súlyfüggvénnyel ($V = \{v_1, \cdots, v_n\}$) az $A : Edge^*[n]$ pointertömb segítségével ábrázoljuk, ahol az $Edge$ típus a következő.

$Edge$
$+v : \mathbb{N}$
$+w : \mathbb{R}$
$+next : Edge*$

A $next$ és a v attributumok szerepe ugyanaz, mint az élsúlyozatlan gráfoknál, w pedig a megfelelő él súlya. Az élsúlyozatlan gráfokhoz hasonlóan az élsúlyozott gráfoknál is: irányítatlan gráfok esetén minden élet kétszer ábrázolunk, irányított gráfok esetén csak egyszer. Élsúlyozott, irányított gráfra láthatunk példát az 5. ábrán.

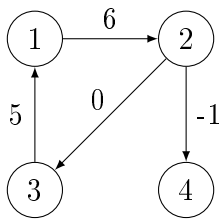
1.5. Élsúlyozott gráfábrázolások tárigénye

1.5.1. Szomszédossági mátrixok

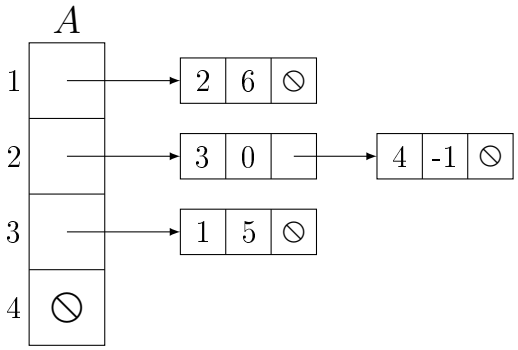
Tárigényük hasonlóan számolható, mint élsúlyozatlan esetben. Feltéve, hogy egy valós számot egy gépi szóban tárolunk, a szomszédossági mátrixos (más néven csúcsmátrixos) ábrázolás tárigénye alapesetben n^2 szó. Irányítatlan gráfoknál, csak az alsóháromszög mátrixot tárolva, $n * (n-1)/2$ szó. Mivel $n * (n-1)/2 \in \Theta(n^2)$, az aszimptotikus tárigény mindkét esetben $\Theta(n^2)$.

1.5.2. Szomszédossági listák

Mindegyik élben eggyel több mező van, mint az élsúlyozatlan esetben. Ez az aszimptotikus tárigényt nyilván nem befolyásolja.



$A[1] \to v = 2$; $A[1] \to w = 6$; $A[1] \to next = \oslash$
$A[2] \to v = 3$; $A[2] \to w = 0$; $A[2] \to next \to v = 4$
; $A[2] \to next \to w = -1$; $A[2] \to next \to next = \oslash$
$A[3] \to v = 1$; $A[3] \to w = 5$; $A[3] \to next = \oslash$
$A[4] = \oslash$

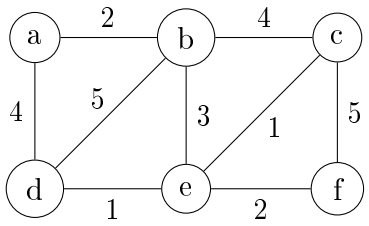


5. ábra. Ugyanaz az élsúlyozott, irányított gráf grafikus (balra) és szomszédossági listás (jobbra) ábrázolással.

1.6. Élsúlyozott gráfok absztrakt osztálya

$\mathcal{G}_w$
$+ V : \mathcal{V}\{\}$
$+ E : \mathcal{E}\{\}$ // $E \subseteq V \times V \setminus \{(u,u) : u \in V\}$
$+ A : V \to 2^V$ // $A(u) = \{ v \in V \mid (u,v) \in E \}$
// $A(u)$ = the adjacent vertices of vertex u.
$+ w : E \to \mathbb{R}$ // weights of edges

2. Minimális feszítőfák ([2] 23)



a – b, 2 ; d, 4.
b – c, 4 ; d, 5 ; e, 3.
c – e, 1 ; f, 5.
d – e, 1.
e – f, 2.

6. ábra. Összefüggő, élsúlyozott, irányítatlan gráf. A csúcsok pl. városok, az élek lehetséges összeköttetések, a megépítésük költségeivel. A lehető legkisebb építési költséggel szeretnénk elérni, hogy tetszőleges városból bármelyik másikba el lehessen jutni.

A 6. ábrán megfogalmazott (és még sok más hasonló) feladat megoldása céljából definiáljuk a *minimális feszítőfa* (MST = Minimum Spanning Tree) fogalmát. Ebben a fejezetben tetszőleges összefüggő, irányítatlan, élsúlyozott gráf *minimális feszítőfáját* keressük. (Az élsúlyok negatívak is lehetnek.)

2.1. Definíció. *A $G = (V, E)$ irányítatlan gráf* feszítő erdeje *a $T = (V, F)$ gráf, ha $F \subseteq E$, valamint T (irányítatlan) erdő (azaz T olyan irányítatlan gráf, aminek mindegyik komponense irányítatlan fa, a fák páronként diszjunktak, és együtt éppen lefedik a G csúcshalmazát).*

2.2. Definíció. *A $G = (V, E)$ irányítatlan, összefüggő gráf* feszítőfája *a $T = (V, F)$ gráf, ha $F \subseteq E$, és T (irányítatlan) fa.*

2.3. Definíció. *Amennyiben $G = (V, E)$ élsúlyozott gráf (fa, erdő stb.) a $w : E \to \mathbb{R}$ súlyfüggvénnyel, akkor a G súlya az élei súlyainak összege:*

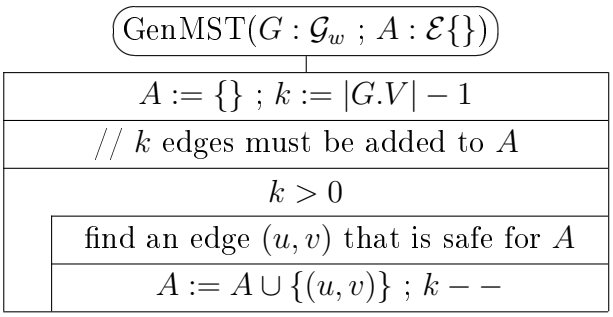
$$w(G) = \sum_{e \in E} w(e)$$

2.4. Definíció. *A G irányítatlan, összefüggő, élsúlyozott gráf* minimális feszítőfája *(*minimum spanning tree: MST*) T, ha T a G feszítőfája, és G bármely T' feszítőfájára $w(T) \leq w(T')$.*

2.1. Egy általános módszer

Az alábbi általános módszer az $A = \{\}$ üres élhalmazból indul, és ezt úgy bővíti újabb és újabb élekkel, hogy A végig a G összefüggő, irányítatlan, élsúlyozott gráf valamelyik minimális feszítőfája élhalmazának a részhalmaza

marad: Éppen az így választott éleket nevezzük az A *élhalmazra nézve biztonságosnak* (*safe for* A). Amikor az élek száma eléri a $|G.V|-1$ értéket, az A szükségszerűen feszítőfa, és így minimális feszítőfa is lesz.



2.5. Definíció. *Tegyük fel, hogy $G = (V, E)$ élsúlyozott, irányítatlan, összefüggő gráf, és $A \subseteq$ a G valamelyik minimális feszítőfája élhalmazának! Ekkor az $(u, v) \in E$ él* biztonságosan hozzávehető *az A élhalmazhoz (*safe for A*), ha $(u, v) \notin A$ és $A \cup \{(u, v)\} \subseteq$ a G valamelyik (az előzővel nem okvetlenül egyező) minimális feszítőfája élhalmazának.*

2.6. Következmény. *Tegyük fel, hogy $G = (V, E)$ élsúlyozott, irányítatlan, összefüggő gráf! Ha egy kezdetben üres A élhalmazt újabb és újabb biztonságosan hozzávehető éllel bővítünk, akkor $|G.V|-1$ bővítés után éppen a G egyik minimális feszítőfáját kapjuk meg.*

A kérdés most az, hogyan tudunk mindig biztonságos élet választani az A élhalmazhoz. Ehhez lesz szükségünk az alábbi fogalmakra és tételre.

2.7. Definíció. *Ha $G = (V, E)$ gráf és $\{\} \subsetneq S \subsetneq V$, akkor a G gráfon $(S, V \setminus S)$ egy* vágás.

2.8. Definíció. *$G = (V, E)$ gráfon az $(u, v) \in E$ él* keresztezi *az $(S, V \setminus S)$* vágást, *ha $(u \in S \wedge v \in V \setminus S) \vee (u \in V \setminus S \wedge v \in S)$.*

2.9. Definíció. *$G = (V, E)$ élsúlyozott gráfon az $(u, v) \in E$* könnyű él *az $(S, V \setminus S)$* vágásban, *ha (u, v) keresztezi a vágást, és $\forall (p, q)$, a vágást keresztező élre $w(u, v) \leq w(p, q)$.*

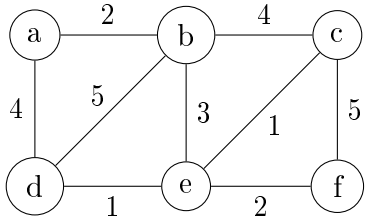
2.10. Definíció. *A $G = (V, E)$ gráfban az $A \subseteq E$ élhalmazt* elkerüli *az $(S, V \setminus S)$* vágás, *ha az A egyetlen éle sem keresztezi a vágást.*

2.11. Tétel. *Ha a $G = (V, E)$ irányítatlan, összefüggő, élsúlyozott gráfon*
(1) A részhalmaza a G valamelyik minimális feszítőfája élhalmazának,
(2) az $(S, V \setminus S)$ vágás elkerüli az A élhalmazt, és
(3) az $(u, v) \in E$ könnyű él *az $(S, V \setminus S)$* vágásban,
$\Longrightarrow$ az (u, v) él biztonságosan hozzávehető az A élhalmazhoz.

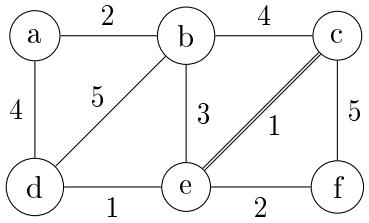
Bizonyítás. $(u, v) \notin A$, ui. (u, v) keresztezi az A-t elkerülő vágást.
Legyen $T = (V, T_E)$ olyan MST, amire $A \subseteq T_E$
a, $(u, v) \in T_E \Rightarrow$ készen vagyunk.
b, $(u, v) \notin T_E$ esetén megjegyezzük, hogy T feszítőfa, tehát T-ben el lehet jutni u-ból v-be, és a T-ben az u-ból a v-be vezető út egyértelmű. Akkor ezen az úton van olyan él, ami keresztezi az $(S, V \setminus S)$ vágást. Legyen (p, q) egy ilyen él! Ekkor $w(p, q) \geq w(u, v) \wedge (p, q) \notin A$. A (p, q) él törlésével a T MST szétesik (azaz két fából álló „feszítő erdő" lesz, az egyik fában az u, a másikban a v csúccsal), de ha az eredményhez az (u, v) élet hozzávesszük, akkor az így adódó T' újra feszítőfa lesz:
$T' := T \setminus (p, q) \cup (u, v)$. Akkor $w(T') = w(T) - w(p, q) + w(u, v) \leq w(T)$ viszont mivel T MST volt, $w(T') \geq w(T)$, azaz $w(T') = w(T)$, és T' is MST kell, hogy legyen. □

Az előbbi tétel segítségével már módszeresen tudunk minimális feszítőfát építeni. Jejöljük a gráfban dupla vonallal az A élhalmaz elemeit!

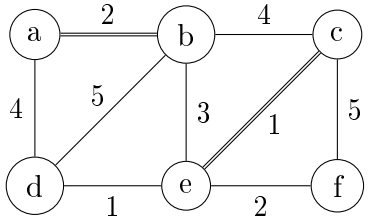
Az alábbi példában minden lépésben választunk egy A-t elkerülő vágást, majd ebben egy könnyű élt, amit hozzáveszünk A-hoz. A hozzávétel eredménye a következő lépés kiinduló pontja.



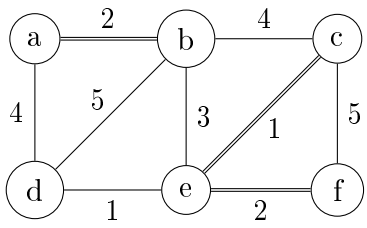
Kezdetben $A = \{\}$, így bármelyik vágás jó. Válasszuk pl. az $(\{a, b, c\}, \{d, e, f\})$ vágást! Ebben (c, e) a könnyű, tehát biztonságos él. Vegyük hozzá A-hoz!



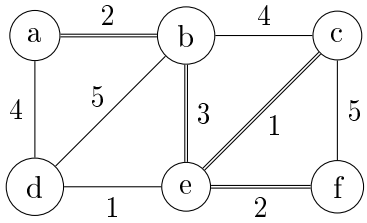
Most $A = \{(c, e)\}$. Ezt elkerüli pl. az $(\{a, f\}, \{b, c, d, e\})$ vágás, amiben (a, b) és (e, f) a könnyű, tehát biztonságos élek. Válasszuk pl. (a, b)-t!



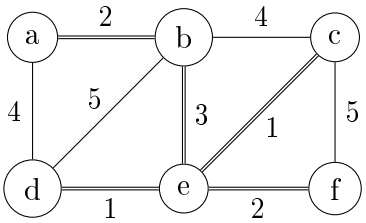
Most $A = \{(a, b), (c, e)\}$. Ezt elkerüli pl. az $(\{a, b, c, d, e\}, \{f\})$ vágás, amiben (e, f) a könnyű, tehát biztonságos él.



Most $A = \{(a,b),(c,e),(e,f)\}$. Ezt elkerüli pl. az $(\{a,b\},\{c,d,e,f\})$ vágás, amiben (b,e) a könnyű, tehát biztonságos él.



$A = \{(a,b),(b,e),(c,e),(e,f)\}$. Ezt egyedül az $(\{a,b,c,e,f\},\{d\})$ vágás kerüli el, amiben (d,e) a könnyű, tehát biztonságos él.



$A = \{(a,b),(b,e),(c,e),(d,e),(e,f)\}$. $|A| = 5 = |G.V|-1$, tehát A egy minimális feszítőfa (MST) élhalmaza.

A fenti módszer még nem tekinthető szigorú értelemben vett algoritmusnak, mert nem adtuk meg, hogyan válasszunk ki az A halmazt elkerülő vágást, és abban könnyű élt. A következő alfejezetekben ismertetendő **Kruskal** és **Prim** algoritmusok éppen ezt teszik, mégpedig

$$O(m * \lg n)\,,$$

tehát nagyon jó maximális műveletigénnyel. (Mint mindig, n a gráf csúcsainak, m pedig az éleinek száma.) Érdekes módon egyik sem adja meg az A-t elkerülő vágást expliciten, a vágásban (az egyik) könnyű élt viszont igen. Ilyen módon, mivel lokálisan optimalizálnak, mindkettő *mohó algoritmus*. A 2.11. tétel szerint viszont a mohóság most kifizetődik, azaz mindkét algoritmus minimális feszítőfát számol.

2.12. Feladat. *Adjon meg olyan 3 csúcsú, összefüggő irányítatlan, élsúlyozott gráfot, amelynek pontosan 2 minimális feszítőfája van. Hány feszítőfája van összesen? Adjon olyan 3 csúcsú gráfot, amelynek 3 minimális feszítőfája van! Létezik olyan 3 csúcsú gráf, aminek háromnál több feszítőfája van? (Gráf alatt, mint mindig, most is egyszerű gráfot értünk.)*

2.2. Kruskal algoritmusa

Kruskal algoritmusa a $G = (V, E)$ gráf éleit a súlyuk (hosszuk) szerint monoton növekvően veszi sorba. Azokat az éleket eldobja, amelyek az A bizonyos éleivel együtt kört képeznének. A többit hozzáveszi A-hoz. Az explicit körkeresés azonban nem lenne hatékony, mert minden egyes e élre bejárást kellene indítani a $(V, A \cup \{e\})$ gráfon. Ehelyett a következő definíción és invariánson alapuló megoldáshoz folyamodunk.

2.13. Definíció. *A $G = (V, E)$ gráf* feszítő erdeje *a (V, A) gráf, ha egymástól diszjunkt fákból, mint komponensekből áll, és $A \subseteq E$. (Két fa egymástól diszjunkt, ha nincs közös csúcsuk [és így közös élük sem].)*

A **Kruskal algoritmus invariánsa,** hogy (V, A) *a $G = (V, E)$ összefüggő, irányítatlan, élsúlyozott gráf feszítő erdeje, és A részhalmaza a G valamelyik minimális feszítőfája élhalmazának.*

Kruskal algoritmusa: **$A = \{\}$-val indulunk**, ami azt jelenti, hogy a kezdeti feszítő erdő fái a $G = (V, E)$ gráf egycsúcsú fái. A G **gráf éleit a súlyuk (hosszuk) szerint monoton növekvően vesszük sorba, és tetszőleges élet pontosan akkor veszünk hozzá A-hoz, ha a (V, A) erdő két fáját köti össze** (azaz nem egy fán belül fut, és így nem zár be egyetlen kört sem). Ezért minden egyes él hozzávételével eggyel csökken az erdő fáinak száma, de továbbra is feszítő erdőt alkotnak. Mivel G összefüggő, bármelyik két fa között van út, így előbb-utóbb összekapcsolódnak és már csak egy T fából áll az erdő, T feszítőfa. A fenti invariáns miatt a T élhalmaza részhalmaza valamelyik M minimális feszítőfa élhalmazának. A G minden feszítőfájának $|V|-1$ éle van, így a T és M élhalmaza megegyezik. Mindkettő csúcshalmaza V, tehát $T = M$, azaz T minimális feszítőfa.

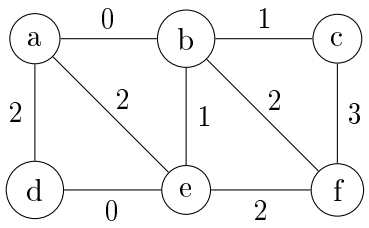
Kérdés még, hogy a fenti invariáns miért igaz. Kezdetben a (V, A) úgy feszítő erdő, hogy $A = \{\}$, tehát A részhalmaza a G bármelyik minimális feszítőfája élhalmazának, azaz az invariáns igaz.

Tekintsünk most az algoritmus futása során egy olyan pillanatot, amikor az invariáns még igaz, és egy újabb e élet készülünk megvizsgálni. Belátjuk, hogy az invariáns az e él feldolgozása után is igaz marad.

Ha e a jelenlegi feszítő erdő valamelyik fáján belül fut, eldobjuk, a feszítő erdő nem változik és az invariáns igaz marad.

Ha e a jelenlegi feszítő erdő két fáját köti össze, legyen S az egyik fa csúcshalmaza, és tekintsük az $(S, V \setminus S)$ vágást, ami nyilván elkerüli A-t. Ekkor e könnyű él a vágásban [mert a G gráf éleit a súlyuk (hosszuk) szerint monoton növekvően vesszük sorba, tehát az aktuális élnél kisebb súlyú éleket már

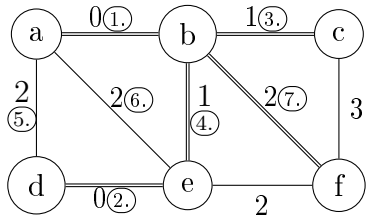
korábban feldolgoztuk, így ezek vagy benne vannak A-ban, vagy nincsenek benne A-ban, de a (V, A) feszítő erdő egyik fájának két csúcsát kötik össze, ezért eldobtuk őket]. Teljesülnek tehát a 2.11. tétel feltételei. Ennélfogva e biztonságosan hozzávehető A-hoz, és hozzá is vesszük. A fentiek szerint tehát az invariáns ebben az esetben is igaz marad.



a – b, 0 ; d, 2 ; e, 2.
b – c, 1 ; e, 1 ; f, 2.
c – f, 3.
d – e, 0.
e – f, 2.

7. ábra. Összefüggő, élsúlyozott, irányítatlan gráf

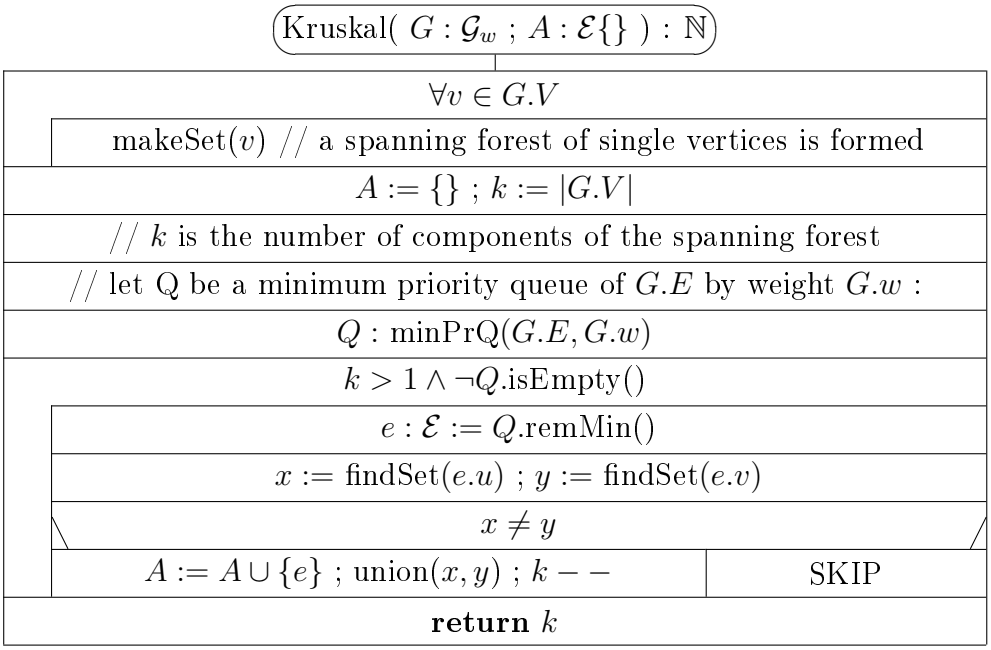
2.14. Példa. *Szemléltessük Kruskal algoritmusát a 7. ábrán látható gráfon!*



Az MST szöveges ábrázolása:
a – b, 0.
b – c, 1 ; e, 1 ; f, 2.
d – e, 0.

8. ábra. A 7. ábráról ismerős gráfon a *kapott* minimális feszítőfa (MST) éleit dupla vonallal jelöltük. Az élsúly mellett vagy alatt jelöltük bekarikázva az él feldolgozásának sorszámát. Az 5. és a 6. lépésben az élt nem vettük hozzzá a feszítő erdőhöz mert a két végpontja már ugyanabban a fában volt. Az e–f és a c–f éleket nem dolgoztuk fel, mert a feszítő erdő a 7. lépés után már csak egy fából áll, ami a keresett MST. Alább táblázattal is szemléltetjük az algoritmust. A komponenseket a feszítőerdő fái csúcsainak sztringjeivel jelöltük. Az *indeterminisztikus esetekben* alfabetikus sorrendet alkalmaztunk. Ez az élek leírására is vonatkozik (pl. nem b–a, hanem a–b él).

lépés	komponensek	él	biztonságos?
①	a, b, c, d, e, f	a$\overset{0}{-}$b	+
②	ab, c, d, e, f	d$\overset{0}{-}$e	+
③	ab, c, de, f	b$\overset{1}{-}$c	+
④	abc, de, f	b$\overset{1}{-}$e	+
⑤	abcde, f	a$\overset{2}{-}$d	–
⑥	abcde, f	a$\overset{2}{-}$e	–
⑦	abcde, f	b$\overset{2}{-}$f	+
-	abcdef	-	-



Mint látható, ez az algoritmus kivételesen „ellenőrzi”, hogy G összefüggő-e. Ha ugyanis összefüggő, $k = 1$ értékkel tér vissza. Ha nem, $k > 1$ lesz.

A műveletigény-számítást azzal a feltételezéssel végezzük el, hogy a makeSet(v) és a union(x, y) eljárások futási ideje $\Theta(1)$, a findSet(v) függvényé pedig $O(\log n)$. E feltételezések jogosságát a 2.2.1. alfejezetben igazoljuk. Feltesszük még, hogy a Q prioritásos sort bináris kupaccal valósítjuk meg. Így, mivel a gráf éleit tároljuk benne, inicializálása $\Theta(m)$, remMin() művelete pedig $O(\log m)$ időt igényel.

Az első ciklus műveletigénye az előző bekezdés alapján $\Theta(n)$, a két ciklus közti részé pedig $\Theta(m)$. Az $e := Q$.remMin() utasítás $O(\log m)$ futási ideje egyben $O(\log n)$, mivel G összefüggő és irányítatlan, és így $n - 1 \leq m \leq n * (n-1)/2 < n^2$, amiből $(\log n) - 1 < \log(n-1) \leq \log m < \log n^2 = 2 * \log n$, azaz $(\log n) - 1 < \log m < 2 * \log n$, tehát $\log m \in \Theta(\log n)$, ahonnét $\Theta(\log m) = \Theta(\log n)$, következőleg $O(\log m) = O(\log n)$. Az „$x :=$ findSet($e.u$) ; $y :=$ findSet($e.v$)” utasításoké a feltevésünk szerint szintén $O(\log n)$, az elágazásé pedig $\Theta(1)$. A fő ciklus egy iterációja tehát maga is $O(\log n)$ időt igényel, és mivel legfeljebb m-szer iterál, a teljes műveletigénye $O(m * \log n)$. Ezt aszimptotikus értelemben már nem módosítja a fő ciklust megelőző inicializálások nála nagyságrenddel kisebb $\Theta(n + m)$ futási ideje.

2.15. Feladat. *Hogyan tudná kifinomultabban értelmezni azt az esetet, ha a Kruskal algoritmus $k > 1$ értékkel tér vissza? Mit jelent a k értéke általá-*

ban? Tudna-e értelmet tulajdonítani a Kruskal algoritmus által kiszámolt A élhalmaznak, ha végül $k > 1$ értéket ad vissza?

2.16. Feladat. *Implementálja a Kruskal algoritmust az elméleti $O(m * \lg n)$ maximális műveletigény megtartásával!*

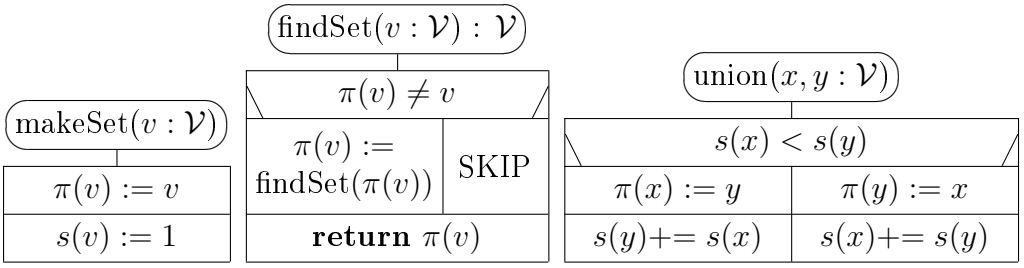
2.2.1. A Kruskal algoritmus halmazműveletei

A fentiekben feltételeztük, hogy a makeSet(v) és a union(x, y) eljárások futási ideje $\Theta(1)$, a findSet(v) függvényé pedig $O(\log n)$. A táblázatos szemléltetésből nyilvánvaló, hogy az élek kezelése mellett nem szükséges még a feszítő erdő fáit is expliciten reprezentálni, elég a fák (a feszítő erdő komponensei) csúcshalmazait megfelelő adatszerkezettel ábrázolni. Ezeknek az adatszerkezetnek a kezelésére vonatkozik tehát az előbb felsorolt három művelet, amelyeknek a kívánt hatékonyságú megvalósítása nemtriviális. A klasszikus megoldás az ún, "Unió-Holvan" adatszerkezet (disjoint-set data structure), aminek a segítségével a gráf csúcsait diszjunkt komponensekbe osztályozhatjuk, ahol a komponensek lefedik a gráf teljes csúcshalmazát. Az adatszerkezetet a fenti három művelettel kezeljük. Mindegyik csúcshalmaznak van egy reprezentáns eleme, ami azonosítja a csúcshalmazt.

- A makeSet(v) eljárás $\Theta(1)$ műveletigénnyel létrehoz egy egyelemű halmazt, ami csak a v csúcsot tartalmazza.
- A findSet(v) függvény $O(\log n)$ műveletigénnyel meghatározza a v csúcsot tartalmazó komponens reprezentáns elemét.
- A union(x, y) eljárás $\Theta(1)$ műveletigénnyel uniózza az x és az y csúcsok által reprezentált diszjunkt halmazokat, és beállítja az új halmaz reprezentáns elemét. A helyes működés előfeltétele, hogy x és y két különböző halmazt reprezentálnak. A mi megvalósításunkban a union(x, y) eljárás végrehatása után az eredetileg nagyobb komponens reprezentáns eleme lesz az egyesített halmazt reprezentáló csúcs.

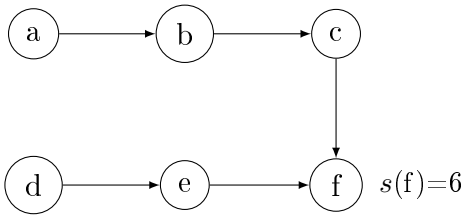
A csúcshalmazokat a reprezentáns elemük felé irányított fákkal ábrázoljuk. Ezért mindegyik csúcsnak van egy π címkéje, aminek értéke az ő szülője, kivéve az irányított fák gyökércsúcsait; tetszőleges r gyökércsúcsra viszont $\pi(r) = r$, $s(r)$ pedig az r-hez tartozó fa mérete. (A fákban a gyökércsúcstól különböző csúcsok s címkéje érdektelen.) Így mindegyik irányítatlan fát a neki megfelelő irányított fa gyökércsúcsával azonosítjuk be. Az irányítatlan feszítőerdő nyilvántartásához tehát egy másik, irányított erdőt is kezelünk. Az irányítatlan feszítőerdő minden egyes (irányítatlan) fájának megfelel az irányított erdő egy (irányított) fája, aminek ugyanaz a csúcshalmaza, az

irányított és az irányítatlan fa élhalmaza viszont általában, irányítástól eltekintve is különböző. A fentieknek megfelelően adódnak a következő struktogramok.

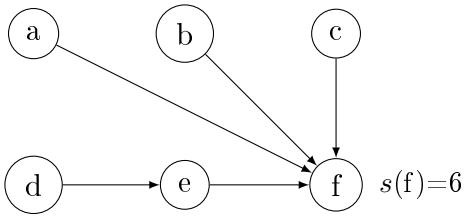


Mint látható, a makeSet(v) művelet a gráf v csúcsából egyelemű irányított fát képez. (Ezt a Kruskal algoritmus kezdetén a gráf mindegyik csúcsára meghívjuk.)

A findSet(v) függvény megállapítja, hogy a v csúcs melyik irányítatlan fában van, azaz megkeresi a csúcsot tartalmazó irányított fa *gyökércsúcsát*. Közben a v csúcs és mindegyik őse π mutatóját a gyökércsúcsra állítja. Ezzel azok a későbbi findSet(x) hívások, amelyekre az $x \to$ *gyökércsúcs* út most rövidebb lett, hatékonyabbak lesznek. Ez ún. spekulatív munka, ami a tapasztalatok szerint, nagy gráfokon általában kifizetődik.



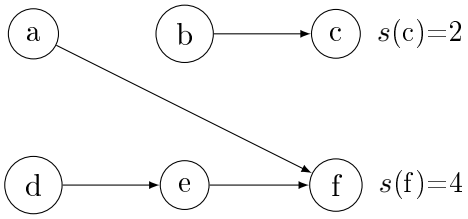
9. ábra. Gyökerük (f) felé irányított fák. Pl. π(d)=e, π(e)=f, π(f)=f. Fent a findSet(a) függvényhívás egy lehetséges inputja, lent ugyanennek a függvényhívásnak a mellékhatása. (A hívás az f csúccsal tér vissza.)



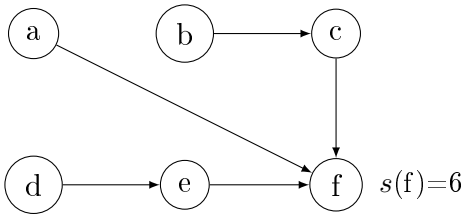
Pl. tegyük fel, hogy adott a 9. ábrán látható felső irányított fa, ahol a gyökércsúcs önmagára mutató pointerének berajzolásától eltekintettünk. Ha erre a fára végrehajtjuk a findSet(a) függvényhívást, ez az f csúccsal tér vissza, mellékhatása pedig az ábra alsó részén látható eredménnyel módosítja a fát.

Arról, hogy ez a heurisztikusnak tűnő *útrövidítés*, nagy gráfokra tényleges *hatékonyságnövekedést* hoz magával, az „Új Algoritmusok" [2] könyvben részletes elemzés olvasható. Az alábbi hatékonyságelemzésnél eltekintünk az *útrövidítésből* eredő *hatékonyságnövekedéstől*. Célunkat az így adódó pontatlanabb felső becslésekkel is elérjük.

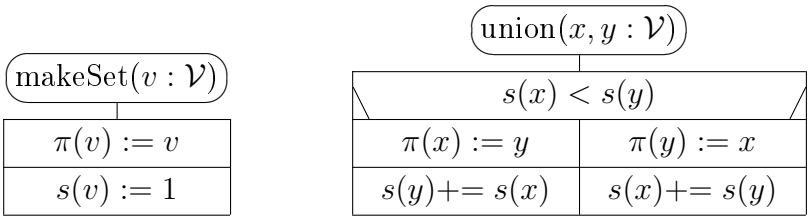
Annak megfelelően, hogy a Kruskal algoritmusban az „$x :=$ findSet$(e.u)$; $y :=$ findSet$(e.v)$" utasítások az e él két végpontjához tartozó gyökércsúcsokat határozzák meg, a union(x, y) művelet később a megfelelő irányított fák gyökércsúcsait köti össze, feltéve, hogy $x \neq y$. A fenti módszerrel a union(x, y) művelet a kisebb méretű irányított fa gyökerét a nagyobb gyökere alá köti be. (Egyenlő méretek esetén mindegy, hogy x vagy y lesz az új gyökér.) Alább belátjuk, hogy ezzel a módszerrel sosem lesz az egyes fák magassága nagyobb, mint a méretük kettes alapú logaritmusa. (Ld. a 2.17. tulajdonságot!) Ebből pedig közvetlenül következik az, hogy a findSet(v) függvény műveletigénye $O(\log n)$, ahol $n = |G.V|$. A union() művelet hatására a 10. ábrán láthatunk példát.



10. ábra. A union(c,f) eljáráshívás inputja (fent) és outputja (lent).



2.17. Tulajdonság. *A makeSet(v) és a union(x, y) eljárások hatására létrejövő, gyökércsúcsuk felé irányított fák magassága sosem lesz nagyobb, mint a méretük kettes alapú logaritmusa, azaz, ha r egy ilyen fa gyökércsúcsa, s(r) a mérete és h(r) a magassága, akkor* $\log s(r) \geq h(r)$.



Bizonyítás. A makeSet(v) eljárás egy csúcsú, nulla magasságú fákat hoz létre, és $\log 1 = 0$, tehát a bizonyítandó tulajdonság kezdetben igaz. Elegendő belátni, hogy a union(x, y) eljárás is tartja a fenti egyenlőtlenséget. Legyen r a union(x, y) eljárás hatására létrejövő fa gyökere! Feltehető, hogy az eljáráshívás előtt $\log s(x) \geq h(x) \wedge \log s(y) \geq h(y)$. Azt kell belátnunk, hogy a union(x, y) eljáráshívás után $\log s(r) \geq h(r)$. Feltehető még, hogy $s(x) \geq s(y)$. Ekkor az x gyökércsúcs alá csatoljuk be az y gyökércsúcsot, így $h(r) = max(h(x), h(y) + 1)$. Két eset van.

- $h(x) > h(y) \Rightarrow h(r) = h(x) \wedge s(r) \geq s(x) \Rightarrow \log s(r) \geq \log s(x) \geq h(x) = h(r) \Rightarrow \log s(r) \geq h(r)$.
- $h(x) \leq h(y) \Rightarrow h(r) = h(y) + 1 \wedge s(r) = s(x) + s(y) \geq 2 * s(y) \Rightarrow \log s(r) \geq \log(2 * s(y)) = 1 + \log s(y) \geq 1 + h(y) = h(r) \Rightarrow \log s(r) \geq h(r)$.

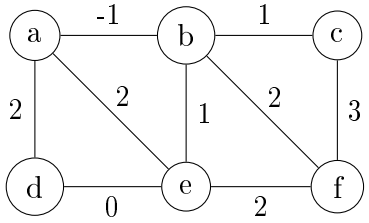
□

Most belátjuk a Kruskal algoritmus műveletigény-számításával kapcsolatos feltételezések jogosságát, azaz, hogy a makeSet(v) és a union(x, y) eljárások futási ideje $\Theta(1)$, a findSet(v) függvényé pedig $O(\log n)$.

Jelölje ez utóbbihoz h a v csúcsot tartalmazó irányított fa magasságát, s pedig a méretét! A findSet(v) függvény műveletigénye nyilván $O(h)$. Másrészt a 2.17. tulajdonság alapján $h \leq \log s$. Ezeket az $s \leq n$ egyenlőtlenséggel összevetve a findSet(v) függvény futási ideje durva felső becsléssel $O(\log n)$. A makeSet(v) és a union(x, y) műveleteké pedig $\Theta(1)$, hiszen sem ciklust, sem eljáráshívást nem tartalmaznak. Ezzel a Kruskal algoritmus műveletigény-számításával kapcsolatos feltételezések jogosságát beláttuk.

2.2.2. A Kruskal algoritmus táblázatos szemléltetése a halmazműveletekkel együtt

A 11. ábrán látható gráfra szemléltetjük a 8. ábránál részletesebb módon a Kruskal algoritmus működését, az alábbi táblázat segítségével. Az algoritmus által kezelt feszítő erdő fáinak csúcshalmazait a megelőző, 2.2.1. alfejezetben ismertetett módon, a gyökerük felé irányított fák segítségével reprezentáljuk. Az alábbi típusú táblázatos illusztrációknál mindig feltesszük, hogy a gráf csúcsait az a,...,z betűk azonosítják.



a – b, -1 ; d, 2 ; e, 2.
b – c, 1 ; e, 1 ; f, 2.
c – f, 3.
d – e, 0.
e – f, 2.

11. ábra. Összefüggő, élsúlyozott, irányítatlan gráf

Ennél a szemléltetésnél tekintettel vagyunk arra, hogy nagy gráfok esetén a feszítő erdő fáinak csúcshalmazait nemtriviális hatékony módon kezelni. Ezért ebben a példában ezeknek a csúcshalmazoknak a hatékony kezelését is bemutatjuk, az előző, 2.2.1. alfejezetben ismertetett megfontolásoknak megfelelő módon.

komponensek	a	b	c	d	e	f	$u \overset{w}{—} v$	$u\pi^* s$	$v\pi^* s$	$\pm$
a, b, c, d, e, f	a1	b1	c1	d1	e1	f1	a $\overset{-1}{—}$ b	a1	b1	+
ab, c, d, e, f	b	b2					d $\overset{0}{—}$ e	d1	e1	+
ab, c, de, f				e	e2		b $\overset{1}{—}$ c	b2	c1	+
abc, de, f		b3	b				b $\overset{1}{—}$ e	b3	e2	+
abcde, f		b5			b		a $\overset{2}{—}$ d	ab5	deb5	−
abcde, f				b			a $\overset{2}{—}$ e	ab5	eb5	−
abcde, f							b $\overset{2}{—}$ f	b5	f1	+
abcdef		b6				b	-			

A táblázat első oszlopában a feszítő erdő komponensei (irányítatlan fái) csúcshalmazainak alakulását láthatjuk, a lehető legegyszerűbb módon, az aktuális állapot halmazait a csúcsai betűi szerint rendezett sztringekkel ábrázolva.

A következő hat oszlop soraiban a gráf csúcsai aktuális π és s értékeinek változásait mutatjuk meg. Ezen hat oszlop első sorában az inicializálás utáni állapotot láthatjuk: Minden $v \in G.V$ csúcsra végrehajtottuk a makeSet(v)

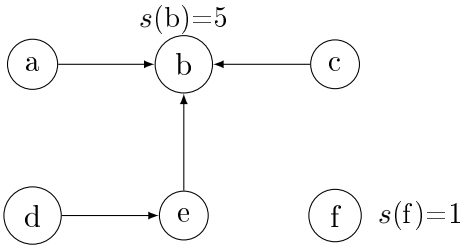
műveletet. A további sorokban mindig a relatíve előző sorban elvégzett findSet() és union() műveleteknek a π és s értékekre gyakorolt hatását illusztráltuk. Az s-értékek csak az irányított fák gyökércsúcsainál érdekesek.

Az $u \underline{^{w}} v$ oszlopban a feszítő erdő aktuális állapotának kialakulása után feldogozásra kiválasztott élt találjuk. Ezt az élsúly szerint növekvő sorrend alapján választjuk ki. (Az eddig is alkalmazott konvenció szerint mindegyik irányítatlan él két végpontját betű szerint növekvő sorrendben írjuk le, pl. a$\underline{^{-1}}$b; az egyenlő súlyú éleket pedig lexikografikusan növekvő sorrendben választjuk ki feldogozásra.)

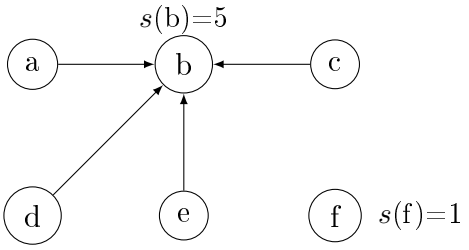
Az $u\pi^*s$ oszlopban azt a sztringet ábrázoljuk, amelynek első betűje az előbb kiválasztott él bal végpontja; a sztringben tetszőleges két, egymást követő betű közül a második az első π értéke; a szám előtt a sztring utolsó betűje a hozzá és az őt megelőző betűkhöz tartozó irányított fa gyökere; a sztring végén található szám pedig az aktuális fa mérete, ami a fa gyökeréhez tartozó s érték. A sztringben található betűk a rekurzív findSet() függvény által bejárt csúcsokat azonosítják. A függvény mellékhatása, hogy hatására mindegyik csúcs π értéke a gyökércsúcsra hivatkozik. Ez az útrövidítés. Ennek tényleges hatása azonban csak a sztring utolsó két betűje előtti csúcsokra van. Ez azt jelenti, hogy a mi példánk esetében az ebben az oszlopban illusztrált findSet() hívásoknak nincs valódi melléhatása.

A $v\pi^*s$ oszlopban levő sztring első betűje az aktuális él jobb oldali végpontja, egyébként a felépítése ugyanolyan, mint az előző oszlopban található sztringé. Az ötödik sorban találjuk a deb5 sztringet, ami szerint a d már jobbról a harmadik betű, és ennek megfelelően a következő sorban már π(d)=b. A mi példánkban ez az egyetlen tényleges útrövidítés (a megfelelő irányított fában). Ld. a 12. ábrát!

Tekintsük most azt az esetet, amikor az $u\pi^*s$ és a $v\pi^*s$ oszlopban levő sztringnek a számok előtti utolsó betűi egyenlők! Világos, hogy tetszőleges sorban ez akkor és csak akkor teljesül, ha a két sztring első betűinek megfelelő csúcsok azonos irányított fában vannak. Mivel az irányítatlan és az irányított fák (élei nem, de) csúcshalmazai megfelelnek egymásnak, ebben az esetben az $u \underline{^{w}} v$ oszlopban levő él az irányítatlan feszítő erdő egyik fájának két csúcsát köti össze, így az él nem biztonságos, mert kört zárna be. Ezért az irányítatlan feszítő erdőhöz nem vesszük hozzá. Ezt az utolsó oszlopban "$-$" jellel jelezzük, és a feszítő erdő a következő sorban ugyanaz marad, bár az esetleges útrövidítések miatt az irányított erdő egy vagy két fájában néhány, a gyökérbe vezető út lerövidülhet. (A mi példánkban csak az előző bekezdésben tárgyalt egyetlen egy esetben van útrövidülés: a "deb" irányított út rövidül "db"-re. Ld. a 12. ábrát!)



12. ábra. A táblázat ötödik sorában, a gráf a$\stackrel{2}{-}$d élének feldolgozása alatt végrehajtódik a findSet(d) hívás. Ennek során a b gyökerű irányított fában a "deb" út (ld. fent) "db"-re rövidül, azaz π(d)=b lesz (ld. lent). Ez a mellékhatás a táblázat következő sorában tükröződik. (Mivel az a$\stackrel{2}{-}$d él feldolgozása során azt találjuk, hogy findSet(a)=findSet(d), ez a két csúcs az irányítatlan feszítő erdőnek is azonos fájában van, ezért kört képezne, s így nem vesszük hozzá az irányítatlan feszítő erdő, azaz a a kialakulóban lévő MST élhalmazához.)

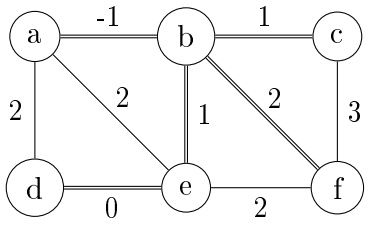


Ha pedig az $u\pi^*s$ és a $v\pi^*s$ oszlopban levő sztringnek a számok előtti utolsó betűi különbözők, akkor a két sztring első betűinek megfelelő u és v csúcsok különböző irányított fákban vannak. Így az u–v él biztonságos, mert az irányítatlan feszítő erdőnek is két különböző fáját köti össze. Ekkor ezt az élet hozzávesszük a feszítő erdőhöz, amivel a két irányítatlan fát egyetlen fává egyesítjük. Az irányítatlan fák új csúcshalamazait a táblázatunk következő sorának első oszlopa mutatja.

Ahhoz viszont, hogy a táblázat utolsó három oszlopában bemutatott *biztonságos él vizsgálat* helyesen működjön, ez utóbbi esetben a két irányítatlan fa gyökércsúcsát is össze kell kötni, a kisebb s-értékűtől a nagyobb s-értékű felé mutató éllel, az egyesített fa gyökércsúcsánál a fa méretének megfelelő s-értéket az eredeti s-értékek összegére átírva. (Azonos méretű irányított fák esetében a szokásos alfabetikus konvenciónk szerint az új él a kisebb betűjű

csúcstól a nagyobb betűjű felé mutat.)

Az algoritmus eredménye a +-szal jelölt sorokban kiválasztott irányítatlan élek, azaz a biztonságosnak talált élek halmaza. A kapott MST pedig a gráf csúcshalmazából és ezen élekből áll össze. Ld. a 13. ábrát!



Az MST itt kapott élhalmaza:
$\{ a \overset{-1}{—} b,\ d \overset{0}{—} e,\ b \overset{1}{—} c,\ b \overset{1}{—} e,\ b \overset{2}{—} f \}$.

13. ábra. A 11. ábra gráfjából az utána következő táblázattal kapott biztonságos élhalmaz (jobbra), és a hozzá tartozó MST a gráfban (balra). A gráfnak a minimális feszítőfához tartozó éleit a grafikus ábrázolásban dupla vonallal jelöltük. Látható, hogy a $b \overset{2}{—} f$ helyett az $e \overset{2}{—} f$ élt is választhattuk volna az MST-be, ha a (tulajdonképpen önkényes, de a számonkéréseken kötelezően alkalmazandó) konvenciónk helyett az utolsó lépésben az algoritmus indeterminizmusát másképp oldottuk volna fel.

2.3. Prim algoritmusa

Kijelölünk a $G = (V, E)$ összefüggő, élsúlyozott, irányítatlan gráfban egy tetszőleges $r \in V$ csúcsot. A $T = (\{r\}, \{\})$, egyetlen csúcsból álló fából kiindulva építünk fel egy minimális (V, F) feszítőfát: Minden lépésben a $T = (N, A)$ fához egy újabb biztonságos élet és hozzá tartozó csúcsot adunk.

Ez azt jelenti, hogy a $T = (N, A)$ fa végig ennek a minimális feszítőfának a része marad, azaz végig igaz az $N \subseteq V \wedge A \subseteq F$ invariáns. Ehhez mindegyik lépésben egy könnyű élet választunk ki az $(N, V \setminus N)$ vágásban. (Ld. a 2.11. tételt!)

A megfelelő könnyű él hatékony meghatározása céljából egy Q min-prioritásos sorban tartjuk nyilván a $V \setminus N$ csúcshalmazt. Mindegyik $u \in V \setminus N$ csúcshoz tartozik egy $c(u)$ és egy $p(u)$ címke. Ha van él az u csúcs és a T fa N csúcshalmaza között, azaz az $(N, V \setminus N)$ vágásban, akkor a $(p(u), u)$ a gráf egyik éle a vágásban, $c(u) = w(p(u), u)$, és minden olyan x csúcsra, amelyre (x, u) vágásbeli él, $w(x, u) \geq w(p(u), u)$.

Ez azt jelenti, hogy $(p(u), u)$ minimális súlyú él azok között, amelyek az u csúcsot a T fához kapcsolják. Ha nincs él a u csúcs és a T fa között, akkor $p(u) = \oslash \wedge c(u) = \infty$.

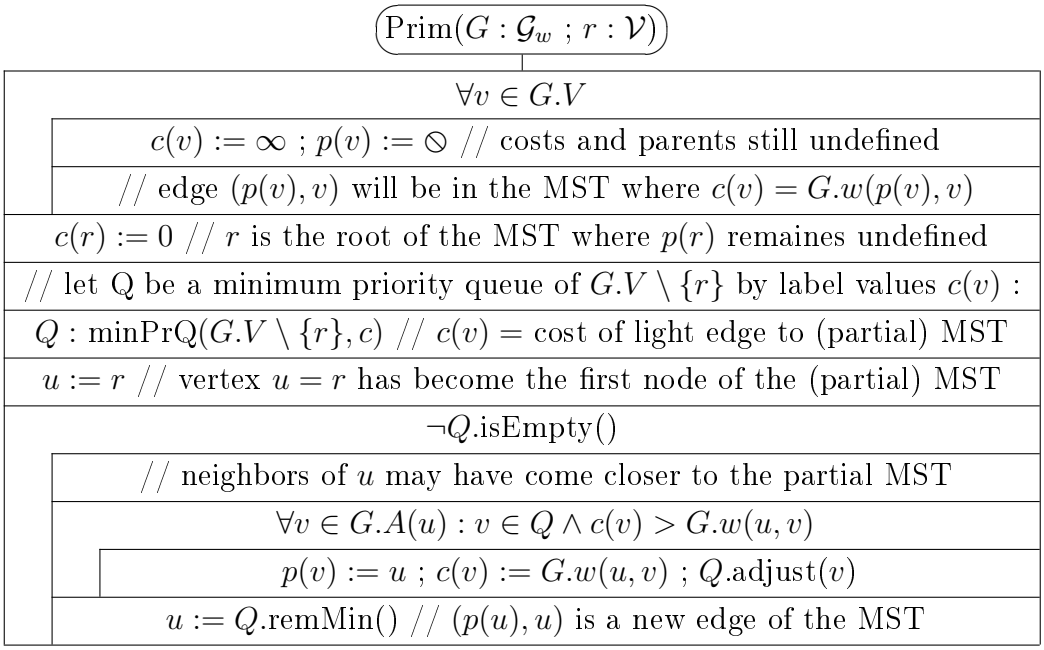
A Q min-prioritásos sorban a csúcsokat a $c(u)$ értékeik szerint tartjuk nyilván. Az $u := Q$.remMin() utasítás tehát egy olyan u csúcsot vesz ki Q-ból, amire a $(p(u), u)$ könnyű él az $(N, V \setminus N)$ vágásban. Ezt az élt így a 2.11. tétel szerint biztonságosan adjuk hozzá a T fához, azaz T továbbra is kiegészíthető minimális feszítőfává, ha még nem az.

Így az eredetileg egyetlen csúcsból álló T fa, $n-1$ db ilyen $(p(u), u)$ él hozzáadásával MST (minimális feszítőfa) lesz.

Amikor u bekerül a T fába, a Q-beli v szomszédai közelebb kerülhetnek a fához, így ezeket meg kell vizsgálni, és ha némelyikre $w(u, v) < c(v)$, akkor a $c(v) := w(u, v)$ és a $p(v) := u$ utasításokkal aktualizálni kell a v csúcs címkéit.

Ilyen módon a $c(v)$ érték csökkenése miatt a Q helyreállítása is szükségessé válhat. Ha például Q reprezentációja egy minimum kupac, a v csúcsot lehet, hogy meg kell cserélni a szülőjével, esetleg újra és újra, mígnem a szülőjének c értéke $\leq c(v)$ lesz, vagy v fölér kupac gyökerébe.

Az alábbi algoritmusban valaki hiányolhatja a T fa explicit reprezentációját. Világos, hogy implicite $N = V \setminus Q$, T éleit pedig az $N \setminus \{r\}$-beli x csúcsok és $p(x)$ címkéik segítségével $(p(x), x)$ alakban definiálhatjuk.



A Prim algoritmus műveletigénye: Feltételezzük, hogy Q reprezentációja bináris minimum kupac, $n = |G.V| \wedge m = |G.E|$.

Az első ciklus futási ideje $\Theta(n)$. Az első és a második ciklus közti rész műveletigényét a Q kupac inicializálása határorozza meg, amire így szintén $\Theta(n)$ adódik.

A második ciklus mindegyik iterációja kiveszi a Q legkisebb c-értékű elemét. Ez tehát $n-1$-szer fut le, és a belső ciklustól eltekintve mindegyik iterációja $O(\log n)$ időt igényel, ami összesen $O(n * \log n)$.

Ehhez hozá kell még adni a belső ciklus futási idejét, ami a gráf mindegyik élére legfeljebb kétszer fog lefutni, hiszen mindegyik élet mindkét végpontja felől megtaláljuk, kivéve azokat, amelyek egyik vépontja a Q-ban utoljára megmaradt csúcs. (A ciklusfejben a $\forall v : (u, v) \in G.E$ rész után következő $v \in Q \wedge c(v) > G.w(u, v)$ szűrő feltétel valójában egy implicit, beágyazott elágazás feltétel része.) Itt a Q.adjust(v) utasítás $O(\log n)$ műveletigénye aszimptotikusan domináns a többi $\Theta(1)$-es futási idő felett. A belső ciklus tehát, összes végrehajtását tekintve is biztosan lefut $O(m * \log n)$ idő alatt.

A fenti részeredményeket összeadva a Prim algoritmus teljes futási idejére $\Theta(n) + \Theta(n) + O(n * \log n) + O(m * \log n) = O((n+m) * \log n)$ adódik. Mivel a gráf összefüggő, $m \geq n-1$, így végül

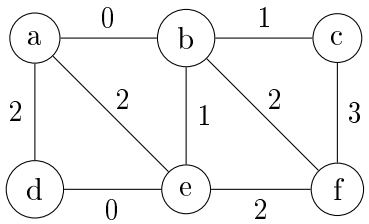
$$MT_{Prim}(n, m) \in O(m * \log n).$$

2.18. Feladat. *Implementálja a Prim algoritmust szomszédossági mátrixos gráfábrázolás esetén! A prioritásos sort rendezetlen tömbbel reprezentálja!*

Mekkora lesz a műveletigény? Lehet-e az aszimptotikus műveletigényen javítani a prioritásos sor kifinomultabb megvalósításával?

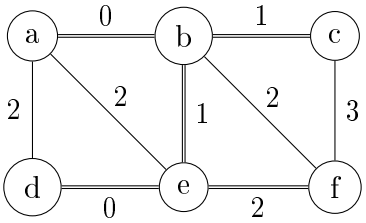
2.19. Feladat. *Implementálja a Prim algoritmust szomszédossági listás gráfábrázolás esetén! Ügyeljen arra, hogy a prioritásos sor megfelelő megvalósításával biztosítani kell az elméleti $O(m * \lg n)$ műveletigényt!*

Az alábbi, már ismerős gráfon szemléltetjük a Prim algoritmus működését, a **d** csúcsból indítva.



a – b, 0 ; d, 2 ; e, 2.
b – c, 1 ; e, 1 ; f, 2.
c – f, 3.
d – e, 0.
e – f, 2.

c értékek Q-ban						minimális feszítő-fába:	p címkék változásai					
a	b	c	d	e	f		a	b	c	d	e	f
∞	∞	∞	0	∞	∞		$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$
2	∞	∞		0	∞	d	d				d	
2	1	∞			2	e		e				e
0		1			2	b	b		b			
		1			2	a						
					2	c						
						f						
0	1	1	0	0	2	eredmény	b	e	b	$\oslash$	d	e



A Kruskal algoritmus eredményéhez képest most a gráf másik minimális feszítőfáját kaptuk meg.

3. Legrövidebb utak egy forrásból([2] 24 ; [5, 8])

Feladat: A $G : \mathcal{G}_w$ gráf tetszőlegesen rögzített s start csúcsából (source vertex) legrövidebb, azaz optimális utat keresünk mindegyik, az s-ből G-ben elérhető csúcsba.

A most következő algoritmusokban az irányítatlan gráfokat olyan irányított gráfoknak tekintjük, ahol a gráf tetszőleges (u, v) élével együtt (v, u) is éle a gráfnak, és $w(u, v) = w(v, u)$.

A feladat pontosan akkor oldható meg, ha G-ben nem létezik s-ből elérhető negatív kör, azaz olyan kör, amely mentén az élsúlyok összege negatív.

Mivel az irányítatlan gráfokat ebben a fejezetben speciális irányított gráfoknak tekintjük, ez a feltétel irányítatlan gráfok esetében azt jelenti, hogy a feladatot akkor tudjuk megoldani, ha nincs a gráfban s-ből elérhető negatív él. (Ha pl. az irányítatlan gráfban (u, v) s-ből elérhető negatív él, akkor a fenti egyszerűsítés miatt $\langle u, v, u\rangle$ egy s-ből elérhető negatív kör.)

Amennyiben a feladat megoldható, mindegyik $v \in G.V\backslash\{s\}$ csúcsra két lehetőség van:
- Ha létezik út s-ből v-be, $d(v)$-ben az optimális út hosszát szeretnénk megkapni, $\pi(v)$-ben pedig egy ilyen optimális úton a v csúcs közvetlen megelőzőjét, azaz szülőjét.
- Ha nem létezik út s-ből v-be, a $d(v) = \infty$ és $\pi(v) = \oslash$ értékeket szeretnénk kapni.

Az s csúcsra a helyes eredmény $d(s) = 0$ és $\pi(s) = \oslash$, mivel az optimális út csak az s csúcsból áll. (Ez már az alábbi algoritmusok inicializálása után teljesül, és végig igaz is marad.)

A legrövidebb utakat kereső algoritmusok futása során az optimális utakat fokozatosan közelítjük. Jelölje $s \rightsquigarrow v$ az adott pillanatig kiszámolt s-ből v-be vezető legrövidebb utat! Az alábbi algoritmusok közös, az inicializálásuk után már teljesülő invariánsa, hogy $d(v)$ ennek a hossza, $\pi(v)$ pedig ($v \neq s$ esetén) ezen az úton a v csúcs közvetlen megelőzője. Ha még nem számoltunk ki $s \rightsquigarrow v$ utat, akkor végtelen hosszúnak tekintjük, azaz $d(v) = \infty$ és $\pi(v) = \oslash$.

Az optimális utak fokozatos közelítéséhez a gráf (u, v) éleit szisztematikusan vizsgáljuk, és ha $s \rightsquigarrow u \to v$ rövidebb mint $s \rightsquigarrow v$, akkor az előbbi lesz az új $s \rightsquigarrow v$. Kód szinten ez azt jelenti, hogy végrehajtjuk az alábbi elágazást, amit *közelítésnek (relaxation)* nevezünk. (Az egyes algoritmusok specialitásai miatt a közelítés egyéb utasításokat is tartalmazhat.)

$d(v) > d(u) + G.w(u, v)$	
$\pi(v) := u$; $d(v) := d(u) + G.w(u, v)$	SKIP

Az alábbi algoritmusok közös vonása még, hogy ismételten kiválasztanak és ki is vesznek az ún. *feldolgozandó* csúcsok halmazából egy csúcsot, és a csúcsból kimenő összes élre végeznek közelítést. Ezeket a közelítéseket együtt a csúcs *kiterjesztésének* nevezzük, míg a csúcs kiválasztását (és kivételét a feldolgozandók közül) a kiterjesztésével együtt a *feldolgozásának* nevezzük.

Főbb különbségeik a következők:

- A Dijkstra algoritmus kezdetben kiválasztja kiterjesztésre s-et, a feldolgozandó csúcsok halmazát pedig $G.V \setminus \{s\}$-sel inicializálja. Először tehát s-et terjeszti ki, majd a feldolgozandók közül mindig egy olyan csúcsot dolgoz fel, amibe az adott időpontig a többihez képest legrövidebb utat talált. Így ez egy *mohó algoritmus*, mert lokálisan optimalizál, azaz mindig a legígéretesebb csúcsot dolgozza fel. Kiderül, hogy így mindig olyan csúcsot terjeszt ki, amibe már eddig optimális utat talált, és egy csúcsot sem kell kétszer kiterjesztenie.

- A DAGshP algoritmus először meghatározza az s-ből elérhető csúcsokat, és egyúttal sorbarendezi ezeket olyan módon, ami garantálja, hogy az adott sorrend szerint feldolgozva őket, mindegyik kiválasztásának az időpontjában már megvan bele az optimális út. Ilyen módon ez is csak egyszer terjeszti ki, és csak az s-ből elérhető csúcsokat.

- Az előbbi két algoritmusnak speciális előfeltételei vannak. Egyik sem tudja a lehető legáltalánosabb esetben megoldani a *legrövidebb utak egy forrásból* problémát. A Dijkstra algoritmus elégséges előfeltétele, hogy a gráfban ne legyen s-ből elérhető negatív súlyú él, míg a DAGshP-é, hogy a gráf s-ből elérhető része DAG legyen. Cserébe mindkét algoritmus a legrosszabb esetben is meglehetősen hatékony. (DAGshP: $\Theta(n+m)$, Dijkstra: $O(n+m) * \log n$.)

- Ezekkel szemben a QBF algoritmus a lehető legáltalánosabb esetben oldja meg a feladatot. Szükséges és elégséges előfeltétele tehát, hogy a gráfban ne legyen az s-ből elérhető negatív kör. Cserébe csak $O(n*m)$ műveletigényt tudunk garantálni, mert ugyanazt a csúcsot többször is feldolgozhatja. Szerencsére átlagos esetben ennél sokkal hatékonyabb, implementációikat összehasonlítva sok nemnegatív élsúlyú gráfon a Dijkstra algoritmusnál is gyorsabb (amihez hozzá kell tenni, hogy talán még gyakrabban ez fordítva van).

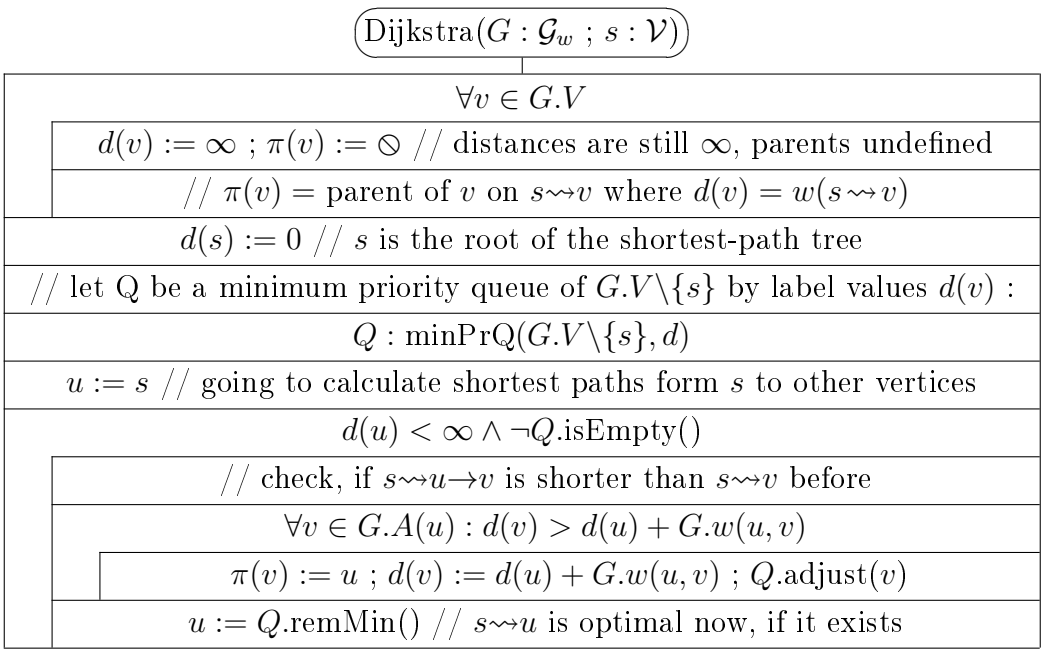
- Mivel a DAGshP és a QBF algoritmusok előfeltétele nemtriviális, ezeknél így az algoritmus része az előfeltételének az ellenőrzése, sőt, a

nem megfelelő bemenetek visszautasítása is oly módon, hogy mindkettő megad egy olyan kört a gráfban, ami miatt nem tudja a feladatot megoldani. (A QBF ebben az esetben negatív kört ad meg.)

3.1. Dijkstra algoritmusa

Előfeltétel: A $G : \mathcal{G}_w$ gráfban $\forall (u,v) \in G.E$-re $G.w(u,v) \geq 0$, azaz a gráf mindegyik élének élsúlya nemnegatív.

Ebben az esetben nem lehet a gráfban negatív kör, tehát a *legrövidebb utak egy forrásból* feladat megoldható. Ez is



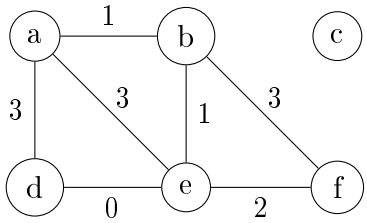
$MT_{Dijkstra}(n,m) \in O((n+m) * \lg n)$ a Prim algoritmusnál már bemutatott módon adódik, tekintettel a két algoritmus közötti meglepő hasonlóságra.

Másrészt $mT_{Dijkstra}(n,m) \in \Theta(n)$, mivel a 2. ciklus előtti rész műveletigénye már $\Theta(n)$, amihez csak a második ciklus egyetlen iterációja és a belső ciklus nullaszori iterációja adódik hozzá, amennyiben nincs a gráfban s-nek rákövetkezője. Ebben az esetben a második ciklus műveletigénye $O(\log n)$ (pontosabban $\Theta(1)$), ami nagysárenddel kisebb, mint $\Theta(n)$.

3.1. Feladat. *Implementálja a Dijkstra algoritmust szomszédossági mátrixos gráfábrázolás esetén! A prioritásos sort rendezetlen tömbbel reprezentálja! Mekkora lesz a műveletigény? Lehet-e az aszimptotikus műveletigényen javítani a prioritásos sor kifinomultabb megvalósításával?*

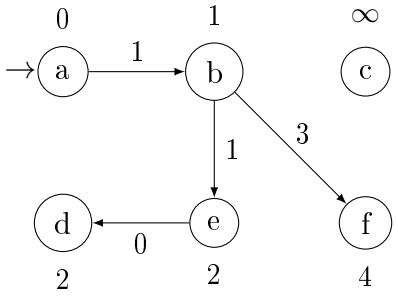
3.2. Feladat. *Implementálja a Dijkstra algoritmust szomszédossági listás gráfábrázolás esetén! Ügyeljen arra, hogy a prioritásos sor megfelelő megvalósításával biztosítani lehet az elméleti $O((m+n) * \lg n)$ műveletigényt!*

Az alábbi gráfon szemléltetjük a Dijkstra algoritmus működését, az **a** csúcsból indítva. A Dijkstra algoritmusnál a *ki nem terjesztett* csúcsokat röviden *nyílt* csúcsoknak nevezzük.



a – b, 1 ; d, 3 ; e, 3.
b – e, 1 ; f, 3.
c.
d – e, 0.
e – f, 2.

nyílt csúcsok d értékei						kiter-	π címkék változásai					
a	b	c	d	e	f	jesztett	a	b	c	d	e	f
0	∞	∞	∞	∞	∞	csúcs : d	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$
	1	∞	3	3	∞	a : 0		a		a	a	
		∞	3	2	4	b : 1					b	b
		∞	2		4	e : 2				e		
		∞			4	d : 2						
		∞				f : 4						
0	1	∞	2	2	4	eredmény	$\oslash$	a	$\oslash$	e	b	b



A legrövidebb utak fája $s =$ a esetén. Az s-ből elérhetetlen csúcsot vagy csúcsokat is feltüntetjük, ∞ d értékkel.

3.3. Tétel. *Amikor az u csúcsot kiválasztjuk kiterjesztésre (először az $u := s$, később az $u := Q.remMin()$ utasítással), akkor u-ba már optimális utat találtunk, feltéve, hogy $d(u) < \infty$.*

Bizonyítás. $u = s$-re nyilván igaz az állítás. Tegyük fel indirekt módon, hogy nem mindegyik csúcsra igaz!

Eszerint van egy olyan v csúcs, amire az algoritmus futása során először lesz igaz, hogy kiválasztjuk kiterjesztésre, de $w(s \overset{opt}{\rightsquigarrow} v) < d(v) < \infty$, ahol $s \overset{opt}{\rightsquigarrow} v$ egy s-ből v-be vezető optimális út, $w(s \overset{opt}{\rightsquigarrow} v)$ pedig ennek a hossza. Világos, hogy $v \neq s$. Továbbá, az $s \overset{opt}{\rightsquigarrow} v$ úton az s csúcsot már kiterjesztettük, a v csúcsot viszont még nem. Van tehát az $s \overset{opt}{\rightsquigarrow} v$ úton egy első csúcs, amit még nem terjesztettünk ki.

Legyen t az $s \overset{opt}{\rightsquigarrow} v$ úton az első csúcs, amit még nem terjesztettünk ki! Szelméletesen: $s \overset{opt}{\rightsquigarrow} t \overset{opt}{\rightsquigarrow} v$. Mivel s-et már kiterjesztettük, $t \neq s$. Továbbá, az $s \overset{opt}{\rightsquigarrow} t$ úton t közvetlen x megelőzőjét is kiterjesztettük már, és az x kiterjesztésekor még teljesült, hogy $d(x) = w(s \overset{opt}{\rightsquigarrow} x)$. Akkor viszont az $x \to t$ él feldolgozása óta már $d(t) = w(s \overset{opt}{\rightsquigarrow} t)$ is teljesül. Mivel a $t \overset{opt}{\rightsquigarrow} v$ úton minden él súlya (azaz hossza) nemnegatív, és t az $s \overset{opt}{\rightsquigarrow} v$ úton van, azaz $s \overset{opt}{\rightsquigarrow} t \overset{opt}{\rightsquigarrow} v$, szükségszerűen $w(s \overset{opt}{\rightsquigarrow} t) \leq w(s \overset{opt}{\rightsquigarrow} v)$.

A fentieket összegezve $d(t) = w(s \overset{opt}{\rightsquigarrow} t) \leq w(s \overset{opt}{\rightsquigarrow} v) < d(v) < \infty$, azaz $d(t) < d(v)$, ami ellentmond az indirekt feltételezésnek, ami szerint a v csúcsot választottuk kiterjesztésre. □

3.4. Tétel. *Ha a kiterjesztésre kiválasztott u csúcsra $d(u) = \infty$, akkor $Q \cup \{u\}$ egyetlen eleme sem érhető már el az s csúcsból.*

Bizonyítás. Nyilván u az első és egyetlen olyan csúcs, amit $d(u) = \infty$ értékkel választottunk ki, mert ezzel megáll a fő ciklus. A korábban kiterjesztésre kiválasztott x csúcsokra tehát $d(x) < \infty$, és ezekbe az előző tétel szerint már optimális utat találtunk.

Tegyük fel indirekt módon, hogy $Q \cup \{u\}$-nak van olyan v eleme, amely elérhető s-ből! Ekkor létezik $s \overset{opt}{\rightsquigarrow} v$ is, amin kell lennie egy első t csúcsnak, amely eleme $Q \cup \{u\}$-nak, de az előző tétel bizonyítása szerint erre már $d(t) = w(s \overset{opt}{\rightsquigarrow} t) < \infty = d(u)$, tehát $d(t) < d(u)$, ami ellentmond annak, hogy az u csúcsot választottuk ki kiterjesztésre. □

3.5. Következmény. *Amíg tehát a kiterjesztésre kiválasztott u csúcsra $d(u) < \infty$, addig olyan csúcsot választunk ki, amelyikbe már optimális utat találtunk.*

Ha $d(u) < \infty$ mindegyik kiválasztott csúcsra igaz, akkor és csak akkor a fenti algoritmus 2. ciklusa $n-1$ iterációval kiüríti Q-t, és megáll, miután a gráf mindegyik csúcsába optimális utat talált.

Ha pedig egyszer a kiterjesztésre kiválasztott u csúcsra már $d(u) = \infty$, akkor $Q \cup \{u\}$ egyetlen eleme sem érhető már el az s csúcsból, és megállhatunk: mindegyik d-értéke végtelen, π-érteke pedig $\oslash$, míg a korábban, véges

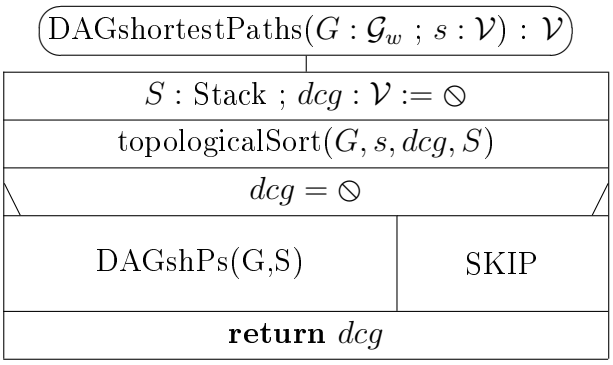
d-értékkel kiterjesztett csúcsok éppen azok, amelyek elérhetők s-ből, és ezekbe találtunk is optimális utat.

3.2. DAG legrövidebb utak egy forrásból (DAGshP)

Előfeltétel: A $G : \mathcal{G}_w$ gráf irányított, és a gráfban nincs s-ből elérhető irányított kör.

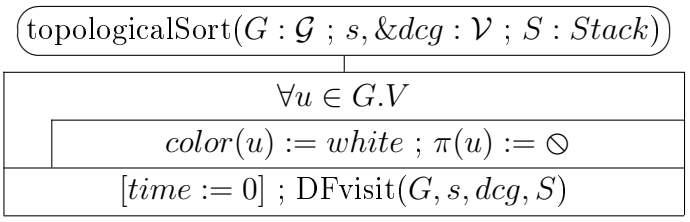
Ebben az esetben nincs a gráfban s-ből elérhető negatív kör sem, így a *legrövidebb utak egy forrásból* feladat megoldható.

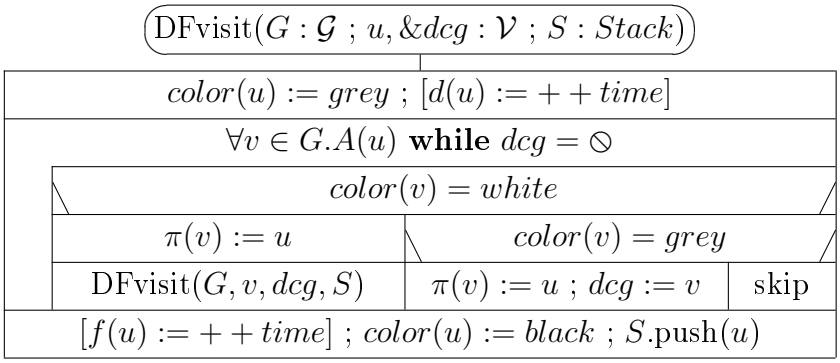
Ez az algoritmus ellenőrzi az előfeltételét. Ha teljesül, a DAGshortestPaths() függvény $\oslash$ értékkel tér vissza. Különben megtalál egy irányított kört, aminek egyik csúcsával tér vissza. Ebből indulva a π címkék mentén a kör fordított irányban bejárható.

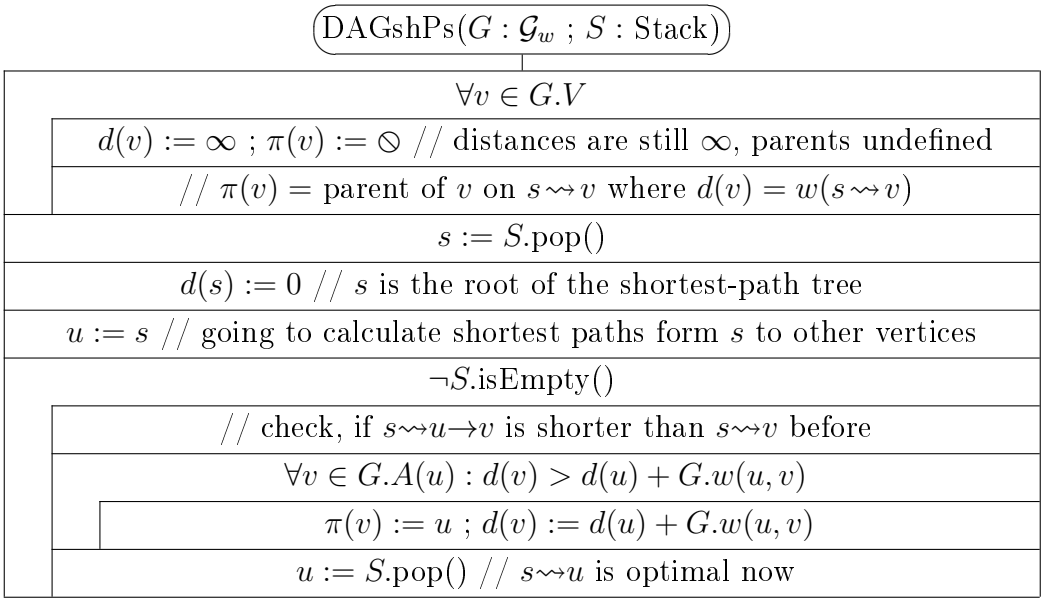


A topologikus rendezés csak az s-ből elérhető részgráfot próbálja topologikusan rendezni.

A *time* nevű változóra – amit az egyszerűség kedvéért globálisnak képzelünk – és a hozzá kapcsolódó utasításokra csak a topologikus rendezés szemléltetésének megkönnyítése végett van szükségünk. Ezek az implmentációkból elhagyhatók, ezért a vonatkozó utasításokat szögletes zárójelbe tettük.







$$MT(n,m) \in \Theta(n+m) \quad \wedge \quad mT(n,m) \in \Theta(n)$$

3.6. Feladat. *Mikor lesz a legkisebb az algoritmus futási ideje? Mikor lesz a legnagyobb? Miért?*

3.7. Tétel. *A legrövidebb utak egy forrásból algoritmus az s-ből elérhető csúcsokba optimális utat talál. Az s-ből el nem érhető x csúcsokra $d(x) = \infty$ és $\pi(x) = \oslash$ lesz.*

Bizonyítás. A kis s-ből indított részleges topologikus rendezés pontosan az s-ből elérhető csúcsokat teszi a nagy S verembe, a megfelelő sorrendben, és így pontosan ezeket a csúcsokat terjesztjük ki, a topologikus sorrendnek megfelelően. A többi x csúcsra az inicalizálás eredményeként $d(x) = \infty$ és

$\pi(x) = \oslash$ marad, ui. ha egy csúcs nem érhető el s-ből, akkor egyetlen G-beli megelőzője sem.

Elegendő belátni, hogy amikor egy u csúcsot kiválasztjuk kiterjesztésre (először az $u := s$, később az $u := S.\mathrm{pop}()$ utasítással), akkor u-ba már optimális utat találtunk.

Az előbbi állítás $u = s$ esetben nyilván igaz. Tegyük fel, hogy adott $v \neq s$ csúcs kiválasztása előtt mindegyik kiterjesztésre kiválasztott csúcsra igaz volt. Mivel a v csúcsnak topologikus sorrend szerinti és így G-beli megelőzőit is a v kiválasztásának pillanatában már kiterjesztettük, a v csúcsnak adott $s \overset{opt}{\rightsquigarrow} v$ úton vett közvetlen u megelőzőjét is, mégpedig úgy, hogy u-ba a kiterjesztésekor már optimális utat találtunk, és így legkésőbb az $u \to v$ él feldolgozásakor v-be is optimális utat találtunk már. □

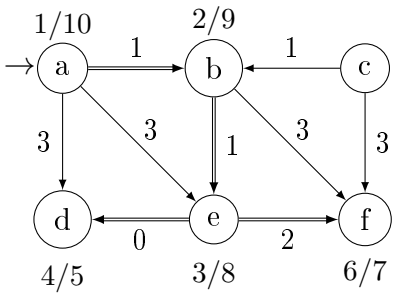
3.8. Feladat. *Implementálja a* DAG legrövidebb utak egy forrásból *algoritmust szomszédossági mátrixos gráfábrázolás esetén!*

Mekkora lesz a műveletigény? Tartható-e az elméleti maximális, illetve minimális műveletigény ezzel az ábrázolással?

3.9. Feladat. *Implementálja a* DAG legrövidebb utak egy forrásból *algoritmust szomszédossági listás gráfábrázolás esetén!*

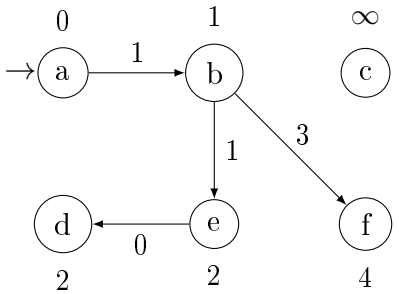
Mekkora lesz a műveletigény? Tartható-e az elméleti maximális, illetve minimális műveletigény ezzel az ábrázolással?

A következő oldalon látható gráfon szemléltetjük a *DAG legrövidebb utak egy forrásból* algoritmust, ahol $s = \mathrm{a}$. Először topologikusan rendezzük az s-ből elérhető részgráfot. (A dupla nyilak a mélységi fa *fa-éleit* jelölik.) Ezután – a szokásos inicializálásokat követően – a csúcsokat a topologikus sorrendjüknek (S) megfelelően terjesztjük ki.



a → b, 1 ; d, 3 ; e, 3.
b → e, 1 ; f, 3.
c → b, 1 ; f, 3.
e → d, 0 ; f, 2.
$S = \langle$ a, b, e, f, d $\rangle$

d értékek változásai						kiter-	π címkék változásai					
a	b	c	d	e	f	jesztett	a	b	c	d	e	f
0	∞	∞	∞	∞	∞	csúcs : d	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$
	1		3	3		a : 0		a		a	a	
				2	4	b : 1					b	b
			2			e : 2				e		
						f : 4						
0	1	∞	2	2	4	eredmény	$\oslash$	a	$\oslash$	e	b	b



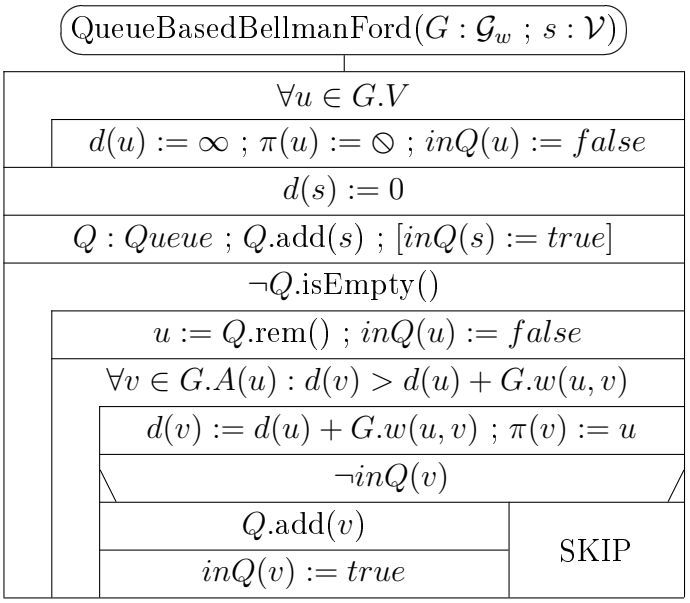
A legrövidebb utak fája $s =$ a esetén. Az s-ből elérhetetlen csúcsot vagy csúcsokat is feltüntetjük, ∞ d értékkel.

3.3. Sor-alapú (Queue-based) Bellman-Ford algoritmus (QBF)

Előfeltétel: Nincs az s-ből elérhető negatív kör. (Irányított gráf esetén lehetnek negatív élsúlyok.)

A *Sor-alapú Bellman-Ford algoritmus (QBF)* (tudományos elemzés és negatív kör vizsgálat nélkül) az informatikai folklórból származik. Legjobb tudásunk szerint először Tarján Róbert Endre amerikai matematikus[1] elemezte és publikálta, *Breadth-first Scanning* néven [5].

A QBF hasonlít a szélességi kereséshez, de az utak hosszát a Dijkstra, a DAGshP (és a Bellman-Ford [2, 3]) algoritmushoz hasonlóan, mint az út menti élsúlyok összegét határozza meg. Kezdetben csak az s start (forrás) csúcsot teszi a sorba. A BFS-hez hasonlóan a szokásos inicializálások után, a fő ciklusban mindig kiveszi a sor első elemét és kiterjeszti, de most mindegyik közvetlen rákövetkezőjére meg is vizsgálja, nem talált-e bele rövidebb utat mint eddig, és ha igen, megfelelően módosítja a d és a π értékét. Ha a módosított d és π értékű csúcs pillanatnyilag nincs benne a sorban, beteszi a sor végére.



Mivel a sor (Queue) adattípusra nincs „$\in$” műveletünk, vagy *átlátszó sor* típust kellene bevezetnünk és megvalósítanunk, vagy – mint itt is – minden v csúcsra értelmezünk egy $inQ(v)$ logikai értékű címkét, ami pontosan akkor

[1] Tarján professzor úr szülei magyar származásúak.

lesz igaz, amikor v eleme a sornak. (Implementációs szinten – feltéve, hogy a csúcsokat 1-től n-ig sorszámoztuk – az $inQ(v)$ címkéket reprezentálhatjuk pl. az $inQ/1 : \mathbb{B}[n]$ tömbbel. Ez a tömb persze az *átlátszó sor* típus része is lehetne.)

Ha nincs a gráfban s-ből elérhető negatív kör, az előző bekezdésben ismertetett algoritmus, miután minden s-ből elérhető v csúcsra meghatározott egy $s \overset{opt}{\rightsquigarrow} v$ utat, üres sorral megáll. Ha a gráf tartalmaz az s-ből elérhető negatív kört, akkor a kiterjesztések egy ilyen mentén „körbejárnak", a sor sosem ürül ki, és az előbbi algoritmus végtelen ciklusba kerül.

3.3.1. A negatív körök kezelése

Arra az esetre, ha nem eleve kizárt, hogy a gráf s-ből elérhető negatív kört tartalmaz, egy ilyen meghatározására Tarján több módszert is kidolgozott [5]. Itt e jegyzet szerzőjének egy viszonylag egyszerű, de könnyen ellenőrizhető kritériumát fogjuk használni [8].

Bevezetjük a gráf v csúcsaira az $e(v)$ címkét: ennyi élt tartalmaz $s \rightsquigarrow v$. Belátható, hogy ha nincs a gráfban s-ből elérhető negatív kör $\iff$ a QBF futása során minden, a fő ciklusban kezelt v csúcsra $e(v) < n$.

A fenti állítást fordítva megfogalmazva, ha van a gráfban s-ből elérhető negatív kör $\iff$ a QBF futása során van olyan, a fő ciklusban kezelt v csúcs, amelyre $e(v) \geq n$. Ilyenkor ez a v csúcs vagy része egy negatív körnek, amit a csúcsok π címkéi mutatnak meg, vagy a π címkéken keresztül el lehet belőle jutni egy ilyen negatív körbe, ami gráf csúcsainak n száma miatt összesen legfeljebb n lépésben azonosítható.

A továbbiakban tehát az s-ből elért v csúcsokra a $d(v)$, a $\pi(v)$ és az $inQ(v)$ mellett az $e(v)$ címkét is nyilvántartjuk.

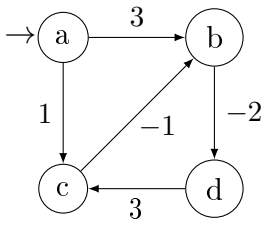
Ha valamely (u, v) él mentén végzett közelítés során úgy találjuk, hogy $e(v) \geq n$, akkor a fentiek szerint meghatározzuk valamelyik negatív kör egyik csúcsát, és az algoritmus ezzel tér vissza.

Ha a QBF futása során mindegyik közelítés után $e(v) < n$, akkor üres sorral állunk meg, és a $\oslash$ extremális hivatkozással térünk vissza.

Belátható, hogy az így kegészített QBF $O(n*m)$ idő alatt minden esetben befejeződik.

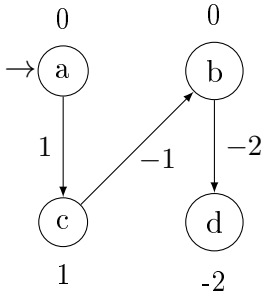
3.3.2. A legrövidebb utak fája meghatározásának szemléltetése

Mindegyik s-ből elérhető v csúcsra kiszámítunk egy $s \overset{opt}{\rightsquigarrow} v$ utat.



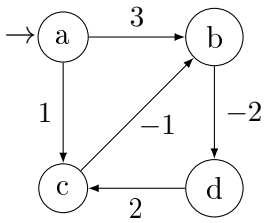
a → b, 3 ; c, 1.
b → d, −2.
c → b, −1.
d → c, 3.

$d; e$ változásai				kiter-	Q :	π változásai			
a	b	c	d	jesztett	Queue	a	b	c	d
$0; 0$	∞	∞	∞	csúcs: $d; e$	⟨ a ⟩	⊘	⊘	⊘	⊘
	$3; 1$	$1; 1$		a: $0; 0$	⟨ b, c ⟩		a	a	
			$1; 2$	b: $3; 1$	⟨ c, d ⟩				b
	$0; 2$			c: $1; 1$	⟨ d, b ⟩		c		
				d: $1; 2$	⟨ b ⟩				
			$-2; 3$	b: $0; 2$	⟨ d ⟩				b
				d: $-2; 3$	⟨ ⟩				
0	0	1	-2	végső d és π értékek		⊘	c	a	b



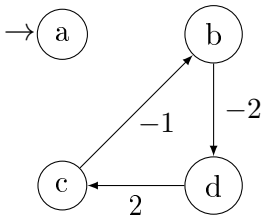
A legrövidebb utak fája s = a esetén. A csúcsoknál csak a d értékeket tüntettük föl.

3.3.3. A negatív körök kezelésének szemléltetése



a → b, 3 ; c, 1.
b → d, −2.
c → b, −1.
d → c, 2.

$d; e$ változásai				kiter-	Q :	π változásai			
a	b	c	d	jesztett	Queue	a	b	c	d
$0; 0$	∞	∞	∞	csúcs: $d; e$	⟨ a ⟩	⊘	⊘	⊘	⊘
	$3; 1$	$1; 1$		a: $0; 0$	⟨ b, c ⟩		a	a	
			$1; 2$	b: $3; 1$	⟨ c, d ⟩				b
	$0; 2$			c: $1; 1$	⟨ d, b ⟩		c		
				d: $1; 2$	⟨ b ⟩				
			$-2; 3$	b: $0; 2$	⟨ d ⟩				b
		$0; 4$		d: $-2; 3$	⟨ ⟩			d	

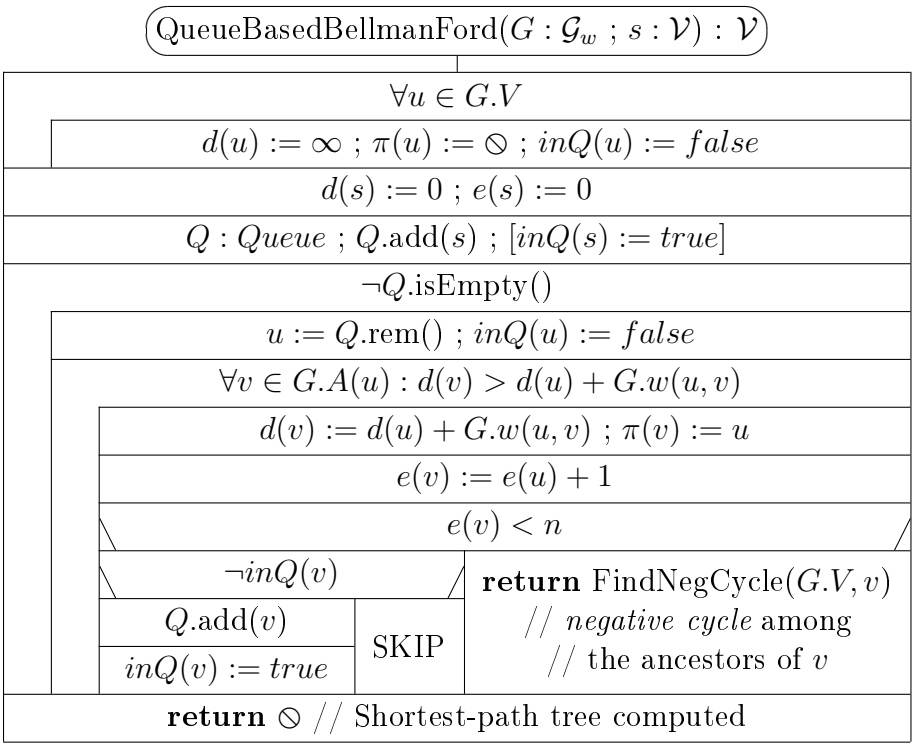


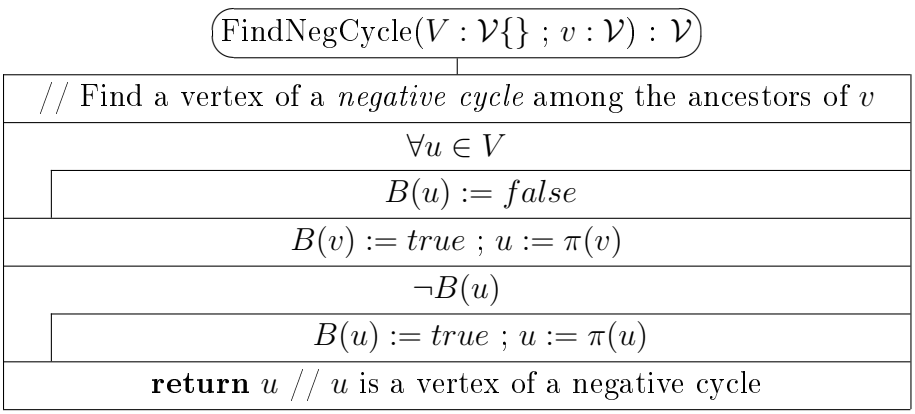
Itt megállunk, mert $e(\mathrm{c}) = 4 = n$, tehát negatív kör található a „c” csúcs ősei közt. A π értékek szerint visszafelé haladva megtalálhatjuk a negatív kört.

3.3.4. A negatív körök kezelésével kiegészített struktogramok

Vegyük figyelembe, hogy egy tetszőleges v csúcs d, e és π címkéjének módosítása után a v csúcsot akkor és csak akkor tesszük a sor végére, ha $e(v) < n$, és v pillanatnyilag nincs benne a sorban!

Ha biztosan nincs a gráfban s-ből elérhető negatív kör, akkor az algoritmus fentebbi változata alkalmazható.





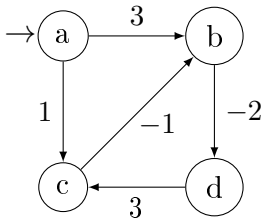
3.3.5. A sor-alapú Bellman-Ford algoritmus (QBF) elemzése

Az alábbi elemzésben arra az esetre szorítkozunk, amikor nincs a gráfban s-ből elérhető negatív kör.

Az algoritmus helyességének bizonyítása és az algoritmus hatékonyságának vizsgálata szempontjából is alapvető a QBF futásának menetekre bontása.

3.10. Definíció. *Menet rekurzív definíciója:*
– 0. menet: a start csúcs (s) feldolgozása.
– (i+1). menet: az i. menet végén a sorban levő csúcsok feldolgozása.

Vegyük pl. az első gráfunkon az algoritmus szemléltetését menetszámlálással együtt! **(Az $s \overset{opt}{\rightsquigarrow} v$ utak megkeresésének szemléltetése.)**



a → b, 3 ; c, 1.
b → d, −2.
c → b, −1.
d → c, 3.

$d; e$ változásai				kiter-	Q :	π változásai				
a	b	c	d	jesztett	Queue	a	b	c	d	Menet
$0; 0$	∞	∞	∞	csúcs: $d; e$	$\langle$ a $\rangle$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	
	$3; 1$	$1; 1$		a: $0; 0$	$\langle$ b, c $\rangle$		a	a		0.
			$1; 2$	b: $3; 1$	$\langle$ c, d $\rangle$				b	1.
	$0; 2$			c: $1; 1$	$\langle$ d, b $\rangle$		c			1.
				d: $1; 2$	$\langle$ b $\rangle$					2.
			$-2; 3$	b: $0; 2$	$\langle$ d $\rangle$				b	2.
				d: $-2; 3$	$\langle\ \rangle$					3.
0	0	1	-2	végső d és π értékek		$\oslash$	c	a	b	-

3.11. Tulajdonság. *A gráf tetszőleges u csúcsára, ha s-ből nem érhető el negatív kör, és az u csúcsba vezet k élből álló $s \overset{opt}{\rightsquigarrow} u$ út, akkor a k. menet elején már $d(u) = w(s \overset{opt}{\rightsquigarrow} u)$ és $\langle s, \ldots, \pi^2(u), \pi(u), u\rangle$ egy optimális út.*

Bizonyítás. k szerinti teljes indukcióval. □

3.12. Lemma. *Ha s-ből nem érhető el negatív kör, akkor tetszőleges u elérhető csúcsra van olyan $s \overset{opt}{\rightsquigarrow} u$ út, ami legfeljebb $n-1$ élből áll.*

Bizonyítás. Ebben az esetben az s-ből induló körmentes utak biztosan nem hosszabbak, mint a kört is tartalmazók. Tetszőleges körmentes útnak viszont legfeljebb $n-1$ éle van. Ebből azonnal adódik, hogy véges sok körmentes út van a gráfban. Akkor s-ből tetszőleges v csúcsba is véges sok körmentes út van, és így létezik ezek között optimális. □

3.13. Tétel. *(A fenti Lemma és Tulajdonság következménye)*
Ha s-ből nem érhető el negatív kör $\Rightarrow$ tetszőleges s-ből elérhető u csúcsra van olyan $s \overset{opt}{\rightsquigarrow} u$ út, amit az $n-1$. menet elejére már a QBF kiszámolt. $\Rightarrow$ az $n-1$. menet végére kiürül a sor, az algoritmus pedig $O(n*m)$ időben megáll, azaz

$$MT(n,m) \in O(n*m).$$

Az elméleti műveletigény becslés meglehetősen rossz hatékonyságot sugall. Gyakorlati tesztek pozitítív élsúlyú, véletlenszerűen generált, nagyméretű, s-ből összefüggő ritka gráfokra[2] ($m \in O(n)$) statisztikusan $\Theta(n)$ átlagos műveletigényt adtak, szomszédossági listás gráfábrázolás esetén. Ez viszont aszimptotikusan az elméleti minimummal egyezik. (A hálózatok jelentős része nagyméretű, ritka gráffal modellezhető. Nem véletlen, hogy hálózatokon való útkeresésnél alkalmazzák is ezt a viszonylag egyszerű, de általános algoritmust.)

[2]Egy gráf egy *adott csúcsból összefüggő*, ha ebből a csúcsból a gráf mindegyik csúcsa elérhető. A *ritka gráfokra* az $m \leq k*n$, ahol k előre adott konstans, gyakran $k \leq 4$.

4. Utak minden csúcspárra ([2] 25)

Ebben a fejezetben két algoritmust ismertetünk. Mindkettő a gráfok csúcsmátrixos reprezentációjára épül, műveletigényük $\Theta(n^3)$ és hasonló elven is működnek.[3] Floyd ún. Floyd-Warshall algoritmusa általánosabb feladatot old meg és egyben későbbi, mint Roy, illetve Warshall, tetszőleges gráf tranzitív lezártját meghatározó algoritmusa [2]. Az irányítatlan gráfokat mindkét algoritmus olyan irányított gráfnak tekinti, amelyben minden (u, v) élnek megvan az ellentétes irányú (v, u) élpárja és $w(u, v) = w(v, u)$.

4.1. Legrövidebb utak minden csúcspárra: a Floyd-Warshall algoritmus (FW)

4.1. Jelölés. $\mathbb{R}_\infty = \mathbb{R} \cup \{\infty\}$

Az $A/1 : \mathbb{R}_\infty[n, n]$ csúcsmátrixszal adott élsúlyozott gráf alapján meghatározza a $D/1 : \mathbb{R}_\infty[n, n]$ mátrixot, ahol $D[i, j]$ az i indexű csúcsból a j indexű csúcsba vezető optimális út hossza, vagy ∞, ha nincs út i-ből j-be. Megadja még a $\pi/1 : \mathbb{N}[n, n]$ mátrixot is, ahol $\pi[i, j]$ az algoritmus által kiszámolt, az i indexű csúcsból a j indexű csúcsba vezető optimális úton a j csúcs közvetlen megelőzője, ha $i \neq j$ és létezik út i-ból j-be. Különben $\pi[i, j] = 0$.

Előfeltétel: A gráfban nincs negatív kör. (Ezt az algoritmus ellenőrzi.)

A továbbiakban gráf csúcsait 1-től n-ig indexeljük és a csúcsokat az indexükkel azonosítjuk.

Az algoritmusunk a $\langle (D^{(0)}, \pi^{(0)}), (D^{(1)}, \pi^{(1)}), \ldots, (D^{(n)}, \pi^{(n)}) \rangle$ mátrixpársorozatot állítja elő, amit a következőképpen definiálunk.

4.2. Jelölés. *$i \overset{k}{\rightsquigarrow} j$ az i csúcsból a j csúcsba vezető út, azzal a megszorítással, hogy az úton a közbenső csúcsok indexe $\leq k$ ($i > k$ és $j > k$ megengedett, továbbá $i, j \in 1..n$ és $k \in 0..n$).*

4.3. Jelölés. *$i \underset{opt}{\overset{k}{\rightsquigarrow}} j$ az i csúcsból a j csúcsba vezető* legrövidebb *körmentes út, azzal a megszorítással, hogy az úton a közbenső csúcsok indexe $\leq k$. Ha több ilyen út lenne, azt vesszük, ahol a közbenső csúcsok indexeinek maximuma a lehető legkisebb.*

[3] Mindkettő az ún. *dinamikus programozás* iskolapéldája [2].

4.4. Mj. *k szerinti indukcióval adódik, hogy amennyiben létezik $i \overset{k}{\rightsquigarrow} j$ út $\Rightarrow$ az $i \underset{opt}{\overset{k}{\rightsquigarrow}} j$ út egyértelműen definiált, ui. a negatív körök hiánya miatt $i = j$ esetén $i \underset{opt}{\overset{k}{\rightsquigarrow}} j = \langle i \rangle$, $i \neq j$ esetén pedig*

- $\exists\, i \underset{opt}{\overset{0}{\rightsquigarrow}} j = \langle i, j \rangle \iff (i, j) \in G.E.$
- *$k \in 1..n$ esetén pedig egy $i \underset{opt}{\overset{k}{\rightsquigarrow}} j$ úttal kapcsolatban két lehetőség van:* $i \underset{opt}{\overset{k}{\rightsquigarrow}} j = i \underset{opt}{\overset{k-1}{\rightsquigarrow}} j \quad \vee \quad i \underset{opt}{\overset{k}{\rightsquigarrow}} j = i \underset{opt}{\overset{k-1}{\rightsquigarrow}} k \underset{opt}{\overset{k-1}{\rightsquigarrow}} j.$

4.5. Definíció. $D_{ij}^{(k)} = \begin{cases} w(i \underset{opt}{\overset{k}{\rightsquigarrow}} j) & \text{ha létezik } i \overset{k}{\rightsquigarrow} j \\ \infty & \text{ha nem létezik } i \overset{k}{\rightsquigarrow} j \end{cases}$

4.6. Definíció.
$\pi_{ij}^{(k)} = \begin{cases} \text{az } i \underset{opt}{\overset{k}{\rightsquigarrow}} j \text{ úton a } j \text{ csúcs közvetlen megelőzője,} & \\ \qquad\qquad\qquad\qquad \text{ha } i \neq j \text{ és létezik } i \overset{k}{\rightsquigarrow} j & \\ 0 \qquad\qquad \text{ha } i = j \text{ vagy nem létezik } i \overset{k}{\rightsquigarrow} j & \end{cases}$

4.7. Tulajdonság. *A $D^{(0)}$ mátrix megegyezik a gráf A csúcsmátrixával, a $D^{(n)}$ mátrix pedig éppen a kiszámítandó D mátrix.*

4.8. Tulajdonság.
$\pi_{ij}^{(0)} = \begin{cases} i & \text{ha } i \neq j \wedge (i, j) \text{ a gráf éle} \\ 0 & \text{ha } i = j \vee (i, j) \text{ nem éle a gráfnak} \end{cases}$
A $\pi^{(n)}$ mátrix pedig megegyezik a kiszámítandó π mátrixszal.

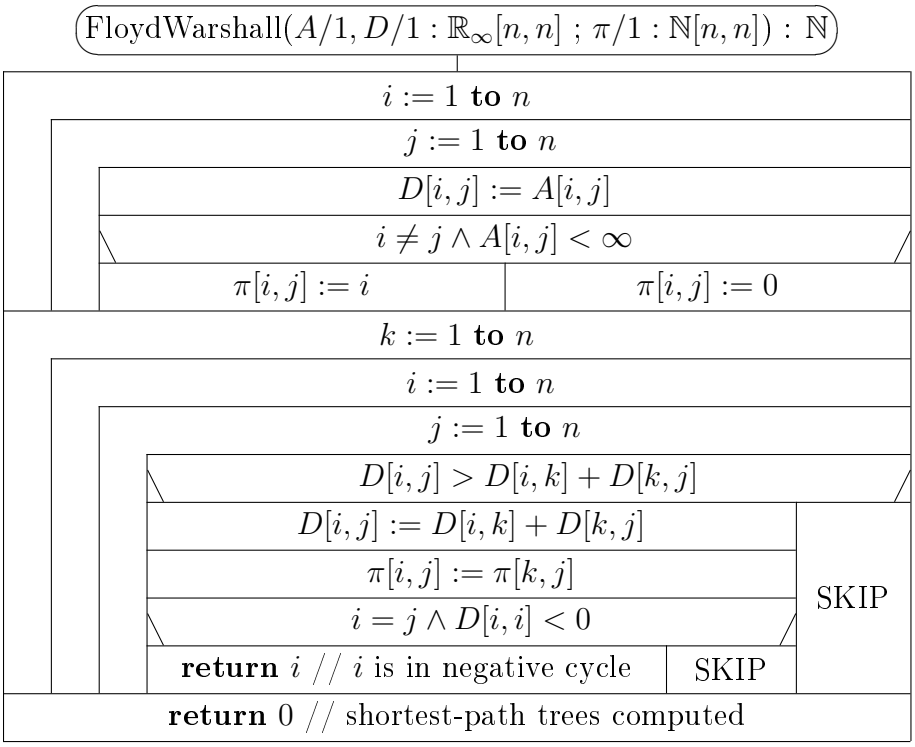
4.9. Tulajdonság. *A 4.4. megjegyzés alapján, $k \in 1..n$ esetén:*
Ha $D_{ij}^{(k-1)} > D_{ik}^{(k-1)} + D_{kj}^{(k-1)}$
akkor $D_{ij}^{(k)} = D_{ik}^{(k-1)} + D_{kj}^{(k-1)} \wedge \pi_{ij}^{(k)} = \pi_{kj}^{(k-1)}$
különben $D_{ij}^{(k)} = D_{ij}^{(k-1)} \wedge \pi_{ij}^{(k)} = \pi_{ij}^{(k-1)}$

4.10. Következmény. *$k \in 1..n$ esetén:*
$D_{ik}^{(k)} = D_{ik}^{(k-1)} \wedge \pi_{ik}^{(k)} = \pi_{ik}^{(k-1)} \quad \wedge \quad D_{kj}^{(k)} = D_{kj}^{(k-1)} \wedge \pi_{kj}^{(k)} = \pi_{kj}^{(k-1)}$

Ez azt jelenti, hogy a $D^{(k)}$ mátrix k-adik sora és oszlopa ugyanaz, mint $D^{(k-1)}$-é, és a $\pi^{(k)}$ mátrix k-adik sora és oszlopa is ugyanaz, mint $\pi^{(k-1)}$-é.

Mivel $D_{ij}^{(k)}$ csak a $D_{ij}^{(k-1)}$, a $D_{ik}^{(k-1)}$ és a $D_{kj}^{(k-1)}$ értékektől függ, valamint $D_{ik}^{(k)} = D_{ik}^{(k-1)} \wedge D_{kj}^{(k)} = D_{kj}^{(k-1)}$, a D mátrix kiszámításához nem kell segédmátrixokat tárolni. A fizikailag is létező D mátrixot az elméleti $D^{(0)}$ mátrix elemeivel inicializáljuk, az alábbi függvény első, kettős ciklusával, majd $k = 1$-től n-ig kiszámoljuk $D^{(k-1)}$-ből $D^{(k)}$-t, az alábbi függvény háromszorosan beágyazott ciklusával, fizikailag végig ugyanazt a D mátrixot használva. Ugyanígy, a π mátrixból is elég egy példány.

Ha a D mátrix főátlójában negatív érték jelenik meg, akkor negatív kört találtunk (ennek meggondolását az Olvasóra bízzuk).



Az algorimus műveletigényéhez elég meggondolni, hogy az első külső ciklus n-szer, a belső n^2-szer fut le, így az inicializálás műveletigénye $\Theta(n^2)$. Ugyanígy, ha nincs a gráfban negatív kör, akkor a második külső ciklus mindegyik iterációjának futási ideje $\Theta(n^2)$, így az összes iterációjáé $\Theta(n^3)$, ami egyben a teljes algoritmus maximális műveletigénye is, azaz $MT(n) \in \Theta(n^3)$. Ha van a gráfban negatív kör, ez akár a második külső ciklus első iterációja alatt kiderülhet, így $mT(n) \in \Theta(n^2)$.

4.1.1. A legrövidebb utak minden csúcspárra probléma *visszavezetése* a legrövidebb utak egy forrásból feladatra

Vegyük észre, hogy a Floyd-Warshall (FW) algoritmus által D és π mátrixok i-edik sora a *legrövidebb utak egy forrásból* feladat megoldása $s = i$ esetére. Megfordítva, ha minden $s \in 1..n$ értékre megoldjuk a *legrövidebb utak egy forrásból* feladatot, megkapjuk a *legrövidebb utak minden csúcspárra* probléma megoldását is.

Az alábbiakban – a teljesség igénye nélkül– megvizsgálunk néhány esetet.

Ha például a gráfunk DAG, ilyen módon a *legrövidebb utak minden csúcspárra* probléma megoldása a legrosszabb esetben is $\Theta(n * (n + m))$ idő alatt számolható ki, ami ritka gráfokon $\Theta(n^2)$, tehát nagysárenddel gyorsabb, mint az FW algoritmus. Sűrű gráfokon viszont – feltéve, hogy a csúcsok többségéből a gráfunk nagy része elérhető – $\Theta(n^3)$ a műveletigény, ami a gyakorlatban a nagyobb rejtett konstansok miatt lassúbb futást eredményez.

Hasonló eredményeket kaphatunk, ha élsúlyozatlan gráfokra szélességi keresést alkalmazunk.

Ha a gráfunk élsúlyai nemnegatívak, a Dijkstra algoritmust minden $s \in 1..n$ értékre végrehajtva $MT(n, m) \in O(n * (n + m) * \log n)$. Ritka gráfokra ez még mindig csak $O(n^2 * \log n)$, ami aszimptotikusan lényegesen jobb, mint az FW algoritmus $\Theta(n^3)$ műveletigénye, tehát nemnegatív élsúlyú, nagyméretű, ritka gráfokon ez lesz a jobb megoldás. Sűrű gráfokra a *felső becslés* most $MT(n, m) \in O(n^3 * \log n)$, ami rosszabb, mint az FW esetén.

Ha csak annyit tudunk, hogy nincs a gráfunkban negatív kör, a QBF algoritmust minden $s \in 1..n$ értékre végrehajtva $MT(n, m) \in O(n * n * m)$, ami – legalábbis a legrosszabb esetet tekintve, és feltételezve, hogy nincs lényegesen kevesebb él, mint csúcs a gráfban – sosem ad jobb *felső becslést*, mint amit az FW algoritmus esetén kaptunk.

4.2. Gráf tranzitív lezártja (TC)

Ebben az alfejezetben csak azzal foglalkozunk, hogy egy hálózatban (gráfban) honnét hova lehet eljutni. Az eljutás módja és költsége most nem érdekel bennünket. Ebből célból vezetjük be a címben jelzett fogalmat.

4.11. Definíció. *A $G = (V, E)$ gráf tranzitív lezártja alatt a $T \subseteq V \times V$ relációt értjük, ahol*

$(u, v) \in T \iff$ *ha G-ben az u csúcsból elérhető a v csúcs.*

4.12. Jelölés. $\mathbb{B} = \{0; 1\}$

Amennyiben a gráfunk csúcsai az $1..n$ indexekkel azonosíthatók, a gráfot ábrázolhatjuk az $A/1 : \mathbb{B}[n, n]$ csúcsmátrixszal (az esetleges élsúlyoktól eltekintve), tranzitív lezártját pedig a $T/1 : \mathbb{B}[n, n]$ mátrix segítségével, ahol

$T[i, j] \iff$ ha az A mátrixszal ábrázolt gráfban van út az i csúcsból a j csúcsba.

A T mátrix kiszámításához definiáljuk a $\langle T^{(0)}, T^{(1)}, T^{(n)} \rangle$ mátrixsorozatot, aminek utolsó tagja magával a T mátrixszal lesz egyenlő.

4.13. Definíció. $T_{ij}^{(k)} \iff \exists\, i \overset{k}{\rightsquigarrow} j \quad (k \in 0..n \ \wedge \ i, j \in 1..n)$

4.14. Tulajdonság. *(A T mátrixok rekurzív megadása.)*

$T_{ij}^{(0)} = A[i, j] \vee (i = j) \qquad (i, j \in 1..n)$

$T_{ij}^{(k)} = T_{ij}^{(k-1)} \vee T_{ik}^{(k-1)} \wedge T_{kj}^{(k-1)} \quad (k \in 1..n \ \wedge \ i, j \in 1..n)$

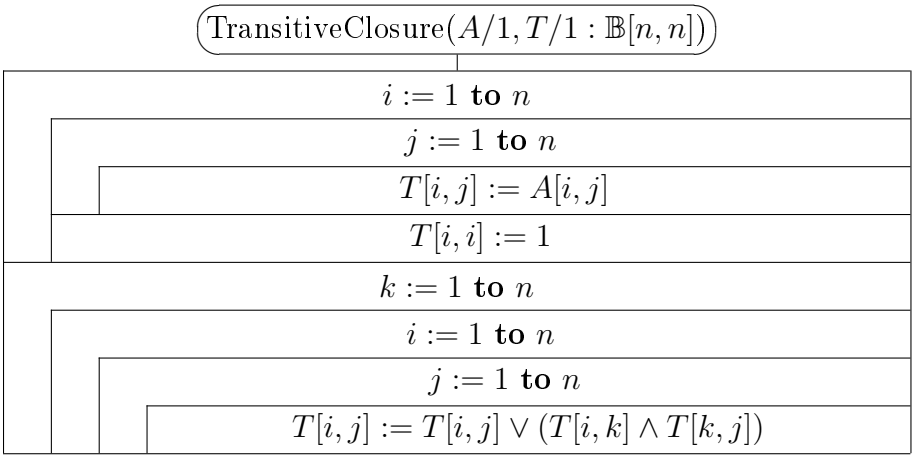
4.15. Következmény.

$T_{ik}^{(k)} = T_{ik}^{(k-1)} \quad \wedge \quad T_{kj}^{(k)} = T_{kj}^{(k-1)} \quad (k \in 1..n \ \wedge \ i, j \in 1..n)$

Ez azt jelenti, hogy a $T^{(k)}$ mátrix k-adik oszlopa és sora ugyanaz, mint a $T^{(k-1)}$ mátrixé. $\quad (k \in 1..n)$

Mivel $T_{ij}^{(k)}$ csak a $T_{ij}^{(k-1)}$, a $T_{ik}^{(k-1)}$ és a $T_{kj}^{(k-1)}$ értékektől függ, valamint $T_{ik}^{(k)} = T_{ik}^{(k-1)} \wedge T_{kj}^{(k)} = T_{kj}^{(k-1)}$, a T mátrix kiszámításához nem kell segédmátrixokat tárolni. A fizikailag is létező T mátrixot az elméleti $T^{(0)}$ mátrix elemeivel inicializáljuk, az alábbi eljárás első, kettős ciklusával, majd $k = 1$-től n-ig kiszámoljuk $T^{(k-1)}$-ből $T^{(k)}$-t, az alábbi eljárás háromszorosan beágyazott ciklusával, fizikailag végig ugyanazt a T mátrixot használva:

Amikor a háromszorosan beágyazott ciklus végrehajtása során $T[i, j]$ új értékét számolom ki, az elméleti mátrixsorozat $T_{ij}^{(k)}$ elemét határozom meg. Az értékadás végrehajtása előtt még egyrészt $T[i, j] = T_{ij}^{(k-1)}$, másrészt pedig $T[i, k] = T_{ik}^{(k)} = T_{ik}^{(k-1)}$ és $T[k, j] = T_{kj}^{(k)} = T_{kj}^{(k-1)}$, tehát mindegy, hogy a második külső ciklus adott iterációján belül a $T[i, k]$, illetve a $T[k, j]$ már értéket kapott-e.



A fenti algoritmusra $T(n) \in \Theta(n^3)$ az FW algoritmusnál már alkalmazott gondolatmenethez teljesen hasonlóan adódik.

Az is világos, hogy a tranzitív lezárt eredményképpen $T[i,j] \iff D[i,j] < \infty$ az FW algoritmus végrehajtása után ($i, j \in 1..n$).

Ez az algoritmus ezt az egyszerűbb feladatot gyakorlatilag mégis hatékonyabban oldja meg, mint az FW a magáét, az egyszerűbb ciklusmagok miatt.

4.2.1. A tranzitív lezárt kiszámításának *visszavezetése* szélességi keresésre

A szélességi keresés után, $s = i$ esetben éppen azok a csúcsok lesznek feketék, amelyek elérhetők i-ből.

Ha tehát lefuttatjuk a szélességi keresés egyszerűsített változatát, ahol $T[i,j] \iff \mathit{color}(j) \neq \mathit{white}$, éppen a T mátrix megfelelő sorát kapjuk meg, $MT(n,m) \in \Theta(n+m)$ műveletigénnyel.

A mátrix összes sorát így $\Theta(n * (n+m))$ maximális futási idővel kapjuk meg, ami ritka gráfok esetén $\Theta(n^2)$, azaz aszimptotikusan jobb, mint a TC algoritmus, a gráf sűrűségétől független műveletigénye.

Sűrű gráfok esetében viszont a szélességi keresésekkel is $\Theta(n^3)$ lesz a műveletigény, ami a gyakorlatban hosszabb futási időt eredményez, a nagyon egyszerű TC algoritmussal összehasonlítva.

Legrövidebb utak minden csúcspárra *

Vadász Péter

1 Elméleti összefoglaló [2]

Adott egy élsúlyozott gráf, keressük minden csúcspárra a két csúcs között lévő legrövidebb utat. Hasonló feladatot lehet elképzelni egy autóstérkép készítésekor is. A feladatot megoldhatjuk a Dijkstra vagy a Sor-alapú Bellman-Ford algoritmussal is, vizsgáljuk meg, hogy ez milyen futási idő költséggel járna:

- **Dijkstra algoritmus**: futtassuk le a Dijkstra algoritmust, a gráf összes csúcsára mint start csúcsra. (Ekkor nem engedünk meg negatív éleket.) Ha az algoritmus által használt prioritásos sort bináris kupac segítségével reprezentáljuk, akkor a futási idő
  – ritka gráfokra, azaz $m \in O(n)$ esetén $O(n(n+m)\log n) = O(n^2 \log n)$,
  – sűrű gráfokra, azaz $m \in \Theta(n^2)$ esetén $O(n(n+m)\log n) = O(n^3 \log n)$

- **Sor-alapú Bellman-Ford (QBF) algoritmus**: futtassuk a QBF algoritmust a gráf összes csúcsára, mint start csúcsra. (Irányított gráf esetén megengedünk negatív éleket, negatív összsúlyú köröket azonban nem. Irányítatlan gráf esetén most sem lehetnek negatív élek a gráfban.)
  – Ritka gráfok ($m \in O(n)$) esetén a futási idő: $O(n * n * m) = O(n^3)$,
  – sűrű gráfokra, azaz $m \in \Theta(n^2)$ esetén $O(n * n * m) = O(n^4)$

Láthatjuk, hogy sűrű gráfok esetén a fenti módszerek egyike sem túl hatékony, ezért egy olyan módszert is használhatunk, ami minden gráf esetén $\Theta(n^3)$ költséggel működik.

*Dr. Ásványi Tibor jegyzete alapján (lektor: Ásványi Tibor)

2 Floyd-Warshall algoritmus

Tegyük fel, hogy G gráf csúcsmátrixos reprezentációval adott. Az egyes csúcsok közötti távolságok számításához egy D-vel jelölt mátrixot fogunk használni, D_{ij} az i és j csúcs közti optimális út hosszát tartalmazza. Egy π mátrixban a közvetlen megelőző csúcsokat tároljuk, π_{ij} az i és j csúcs között vezető optimális úton j csúcs közvetlen megelőzőjét tartalmazza. Az optimális utakat úgynevezett **belső csúcsok** mentén keressük.

Definíció 2.1. Belső csúcs [3] Egy $p = \langle v_1, v_2, ..., v_k \rangle$ út belső csúcsa minden v_1-től és v_k-tól különböző csúcs. Tehát $\{v_2, ..., v_{k-1}\}$

Az algoritmus n lépésben határozza meg a csúcspárok közti legrövidebb utakat. A k-adik lépése után a $D_{ij}^{(k)}$ az i és j csúcsok között lévő szuboptimális út hosszát tartalmazza, azzal a megszorítással, hogy a belső csúcsok címkéje legfeljebb k. A k címkéjű csúcs vagy szerepel az úton vagy nem, attól függően, hogy az út ettől rövidebb lesz-e vagy sem.

A $D^{(k)}$ és $\pi^{(k)}$ mátrixok kiszámítása [2]:

$$D_{ij}^{(0)} = \begin{cases} 0, \text{ ha } i = j \\ w(i,j), \text{ ha } (i,j) \in G.E \wedge i \neq j \\ \infty, \text{ ha } i \neq j \wedge (i,j) \notin G.E \end{cases}$$

$$\pi_{ij}^{(0)} = \begin{cases} 0, \text{ ha } i = j \\ i, \text{ ha } (i,j) \in G.E \wedge i \neq j \\ 0, \text{ ha } i \neq j \wedge (i,j) \notin G.E \end{cases}$$

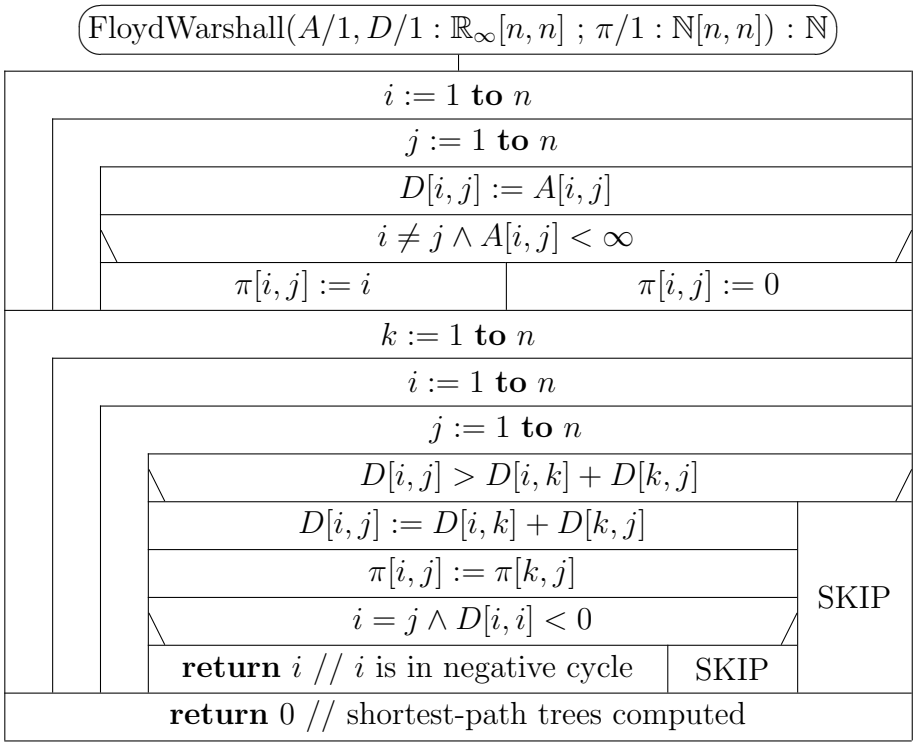
$D^{(0)}$ mátrix lényegében megegyezik a csúcsmátrixal.

$$D_{ij}^{(k)} = \begin{cases} D_{ik}^{(k-1)} + D_{kj}^{(k-1)}, \text{ ha } D_{ij}^{(k-1)} > D_{ik}^{(k-1)} + D_{kj}^{(k-1)} \\ D_{ij}^{(k-1)} \text{ különben} \end{cases}$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{kj}^{(k-1)}, \text{ ha } D_{ij}^{(k-1)} > D_{ik}^{(k-1)} + D_{kj}^{(k-1)} \\ \pi_{ij}^{(k-1)} \text{ különben} \end{cases}$$

Az algoritmus mátrixpárok sorozatát állítja elő 0-tól n-ig, ami $n + 1$ db mátrixpár. Az algoritmus működése során mégis elegendő egyetlen D és π mátrixot fenntartani, mivel a k-adik sor és k-adik oszlop a $k - 1$ és k-adik lépés között nem változik meg, hiszen a k

címkéjű csúcsból saját magán keresztül nem találhatunk rövidebb utat sehova, valamint a k címkéjű csúcsba sem találhatunk rövidebb utat önmagán keresztül semelyik másik csúcsból sem.



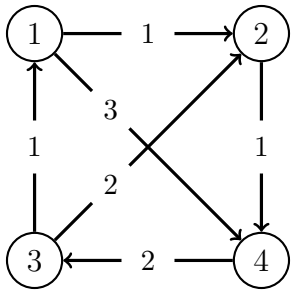
Ábra 1.: Floyd-Warshall algoritmus [1]

Floyd-Warshall algoritmus esetén, ha a gráf tartalmaz negatív összsúlyú kört, akkor a főátlóban negatív értékek jelennek meg, ezt az algoritmus is figyelembe veszi.

2.1 Példák

A feladatok megoldása során minden lépésben felrajzoljuk a mátrixokat, irányítatlan esetben a D mátrixok szimmetrikusak. A k-adik lépésben végigmegyünk D mátrix celláin, k-adik sor és k-adik oszlop kivételével és megnézzük, hogy az (i,j) cella esetén (i,k) és (k,j) cellák összege, optimálisabb-e, mint (i,j), ha igen, akkor módosítjuk az értéket (π mátrixban is).

1. példa: Szemléltessük a Floyd-Warshall algoritmus működését az alábbi gráfon! Adjuk meg D és π mátrixokat minden iterációban. Adjuk meg a csúcspárok közti optimális utakat is!



$D^{(0)}$	**1**	**2**	**3**	**4**
1	0	1	∞	3
2	∞	0	∞	1
3	1	2	0	∞
4	∞	∞	2	0

$\pi^{(0)}$	**1**	**2**	**3**	**4**
1	0	1	0	1
2	0	0	0	2
3	3	3	0	0
4	0	0	4	0

Láthatjuk, hogy $D^{(0)}$ mátrix megegyezik a gráf csúcsmátrixos reprezentációjával.

$D^{(1)}$	**1**	**2**	**3**	**4**
1	0	1	∞	3
2	∞	0	∞	1
3	1	2	0	④
4	∞	∞	2	0

$\pi^{(1)}$	**1**	**2**	**3**	**4**
1	0	1	0	1
2	0	0	0	2
3	3	3	0	①
4	0	0	4	0

$D^{(1)}$ és $\pi^{(1)}$ mátrixokan dupla szegéllyel jelöltük azokat a sorokat, melyek nem változnak az első iterációban. Nézzük meg, hogyan határozhatjuk meg a kék színű cella értékét a piros színű cellák segítségével! A cella a mátrix 3. sorában és 4. oszlopában van, a fenti rekurzív képlet alapján, ha $D_{34}^{(0)} > D_{31}^{(0)} + D_{14}^{(0)}$, akkor $D_{34}^{(1)} = D_{31}^{(0)} + D_{14}^{(0)}$, $D_{31}^{(0)} + D_{14}^{(0)} = 4$, ami határozottan kisebb, mint ∞ ($D_{34}^{(0)}$ értéke), tehát 4 lesz a cella új értéke. A $\pi_{34}^{(1)}$ értékét $\pi_{14}^{(0)}$ értéke határozza meg. A felülírt mezők új értékét bekarikáztuk.

$D^{(2)}$	**1**	**2**	**3**	**4**
1	0	1	∞	②
2	∞	0	∞	1
3	1	2	0	③
4	∞	∞	2	0

$\pi^{(2)}$	**1**	**2**	**3**	**4**
1	0	1	0	②
2	0	0	0	2
3	3	3	0	②
4	0	0	4	0

$D^{(3)}$	**1**	**2**	**3**	**4**
1	0	1	∞	2
2	∞	0	∞	1
3	1	2	0	3
4	③	④	2	0

$\pi^{(3)}$	**1**	**2**	**3**	**4**
1	0	1	0	2
2	0	0	0	2
3	3	3	0	2
4	③	③	4	0

$D^{(4)}$	**1**	**2**	**3**	**4**
1	0	1	④	2
2	④	0	③	1
3	1	2	0	3
4	3	4	2	0

$\pi^{(4)}$	**1**	**2**	**3**	**4**
1	0	1	④	2
2	③	0	④	2
3	3	3	0	2
4	3	3	4	0

Az egyes csúcsok közötti legrövidebb utakat a $\pi^{(4)}$ mátrix alapján tudjuk meghatározni:

- $1 \rightsquigarrow 2$: $1 \to 2$ [összsúly: 1], $D^{(4)}_{12}$ érték alapján
- $1 \rightsquigarrow 3$: $1 \to 2 \to 4 \to 3$ [összsúly: 4] ($\pi^{(4)}_{13}$ értéke 4, azaz az 1-es és 3-as csúcs között vezető optimális úton 3-as csúcs közvetlen megelőzője a 4-es csúcs. Most meg kell néznünk, hogy hogyan jutunk a 4-es csúcsba, azaz mi a 4-es megelőzője, ezt a $\pi^{(4)}_{14}$ érték mutatja, ami 2, végül pedig a 2-es csúcs megelőzője kell, amit a $\pi^{(4)}_{12}$ értéke ad meg, ez már 1, tehát megtaláltuk az $1 \rightsquigarrow 3$ úton szereplő összes csúcsot.)
- $1 \rightsquigarrow 4$: $1 \to 2 \to 4$ [összsúly: 2]
- $2 \rightsquigarrow 1$: $2 \to 4 \to 3 \to 1$ [összsúly: 4]
- $2 \rightsquigarrow 3$: $2 \to 4 \to 3$ [összsúly: 3]
- $2 \rightsquigarrow 4$: $2 \to 4$ [összsúly: 1]
- $3 \rightsquigarrow 1$: $3 \to 1$ [összsúly: 1]
- $3 \rightsquigarrow 2$: $2 \to 3 \to 2$ [összsúly: 2]
- $3 \rightsquigarrow 4$: $3 \to 2 \to 4$ [összsúly: 3]
- $4 \rightsquigarrow 1$: $4 \to 3 \to 1$ [összsúly: 3]
- $4 \rightsquigarrow 2$: $4 \to 3 \to 2$ [összsúly: 4]

- $4 \rightsquigarrow 3$: $4 \to 3$ [összsúly: 2]

3 Gráf tranzitív lezártja

Definíció 3.1. Tranzitív lezárt: Egy $G(V, E)$ gráf tranzitív lezártja egy $G'(V, E')$ gráf, ahol $(u, v) \in G'.E'$ akkor és csak akkor, ha G-ben vezet út u-ból v-be.

A tranzitív lezárt kiszámításához használhatjuk a Warshall algoritmust, amely a Floyd-Warshall algoritmus egy módosított változata. Egy T $n \times n$-es logikai mátrixot használunk, az élsúlyok most nem számítanak, a gráf lehet irányított, irányítatlan.

A Floyd-Warshall algoritmushoz hasonlóan a Warshall algoritmus mátrixok sorozatát állítja elő (n iterációval), de a korábbi indoklás miatt itt is elég egy mátrixot használni. A k-adik iteráció után $T_{ij}^{(k)}$ értéke igaz, ha az i és j címkéjű csúcsok között vezet olyan út, amelyben a belső csúcsok címkéje legfeljebb k.

A $T^{(k)}$ mátrixok kiszámítása (1 jelentése "igaz", 0 jelentése "hamis"):

$$T_{ij}^{(0)} = \begin{cases} 1, \text{ ha } i = j \vee (i, j) \in G.E \\ 0, \text{ különben} \end{cases}$$

$$T_{ij}^{(k)} = T_{ij}^{(k-1)} \vee (T_{ik}^{(k-1)} \wedge T_{kj}^{(k-1)})$$

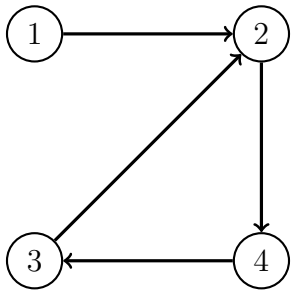
```
TransitiveClosure(A/1, T/1 : B[n, n])
    i := 1 to n
        j := 1 to n
            T[i, j] := A[i, j]
        T[i, i] := 1
    k := 1 to n
        i := 1 to n
            j := 1 to n
                T[i, j] := T[i, j] ∨ (T[i, k] ∧ T[k, j])
```

Ábra 2.: Warshall algoritmus [1]

Futási idő: $\Theta(n^3)$

3.1 Példák

2. példa: Szemléltessük a Warshall algoritmus működését az alábbi gráfon! Adjuk meg T mátrixot az inicializálása és a fő ciklus mindegyik iterációja után is!



$T^{(0)}$	**1**	**2**	**3**	**4**
1	1	1	0	0
2	0	1	0	1
3	0	1	1	0
4	0	0	1	1

$T^{(1)}$	**1**	**2**	**3**	**4**
1	1	1	0	0
2	0	1	0	1
3	0	1	1	0
4	0	0	1	1

$T^{(2)}$	**1**	**2**	**3**	**4**
1	1	1	0	①
2	0	1	0	1
3	0	1	1	①
4	0	0	1	1

$T^{(3)}$	**1**	**2**	**3**	**4**
1	1	1	0	1
2	0	1	0	1
3	0	1	1	1
4	0	①	1	1

$T^{(4)}$	**1**	**2**	**3**	**4**
1	1	1	①	1
2	0	1	①	1
3	0	1	1	1
4	0	1	1	1

Irodalomjegyzék

[1] Dr Ásványi Tibor *Algoritmusok és adatszerkezetek II. előadásjegyzet - Élsúlyozott gráfok és algoritmusaik*

[2] Koruhely Gábor, Szalay Richárd *Algoritmusok és adatszerkezetek 2*
(hallgatói jegyzet, lektorált és javított)
http://aszt.inf.elte.hu/∼asvanyi/ad/

[3] Fekete István, Hunyadvári László *Algoritmusok és adatszerkezetek* `http://tamop412.elte.hu/tananyagok/algoritmusok/index.html`

Algoritmusok és adatszerkezetek II. előadásjegyzet:

Mintaillesztés, Tömörítés

Ásványi Tibor – asvanyi@inf.elte.hu

2023. augusztus 21.

Tartalomjegyzék

Hivatkozások

[1] Ásványi Tibor, Algoritmusok és adatszerkezetek II.
Útmutatások a tanuláshoz, jelölések, tematika,
fák, gráfok,
mintaillesztés, tömörítés
http://aszt.inf.elte.hu/~asvanyi/ad/ad2jegyzet/

[2] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.,
angolul: Introduction to Algorithms (Fourth Edititon), chapter 32 (String Matching) *The MIT Press*, 2022.
magyarul: Új Algoritmusok, 32. fejezet *Scolar Kiadó*, Budapest, 2003. ISBN 963 9193 90 9

[3] Fekete István, Algoritmusok jegyzet
http://ifekete.web.elte.hu/

[4] Rónyai Lajos – Ivanyos Gábor – Szabó Réka, Algoritmusok, *TypoTEX Kiadó*, 1999. ISBN 963 9132 16 0
https://www.tankonyvtar.hu/hu/tartalom/tamop425/2011-0001-526_ronyai_algoritmusok/adatok.html

[5] Weiss, Mark Allen, Data Structures and Algorithm Analysis, *Addison-Wesley*, 1995, 1997, 2007, 2012, 2013.

[6] Ásványi Tibor, Algoritmusok és adatszerkezetek I. előadásjegyzet (2022)
http://aszt.inf.elte.hu/~asvanyi/ad/ad1jegyzet/ad1jegyzet.pdf

1. Mintaillesztés (String-matching [2])

1.1. Jelölések. *(Notations)*
$\mathbb{N} = \{\, i \in \mathbb{Z} \mid i \geq 0 \,\}$ $\qquad$ $i\,..\,k = \{\, j \in \mathbb{N} \mid i \leq j \leq k \,\}$
$[i\,..\,k) = \{\, j \in \mathbb{N} \mid i \leq j < k \,\}$ $\qquad$ $(i\,..\,k) = \{\, j \in \mathbb{N} \mid i < j < k \,\}$
$\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_d\}$ *az ábécé (alphabet), ahol* $d \in \mathbb{N} \wedge d > 0$
$T : \Sigma[n]$ *a szöveg (text), ami betűk egy sztringje* 0*-tól* $n-1$*-ig indexelve.*
$T[i\,..\,j) = T[i\,..\,j-1]$
$P : \Sigma[m]$ *a minta (pattern), ahol* $0 < m \leq n$.

A továbbiakban feltesszük, hogy $T : \Sigma[n]$ és $P : \Sigma[m]$, valamint a hosszuk, azaz m és n változatlanok, ahol $1 \leq m \leq n$ (a minta nem üres, és legfeljebb olyan hosszú, mint a szöveg).

A $T : \Sigma[n]$ szövegben keressük a $P : \Sigma[m]$ minta előfordulásait (occurrences). Más szavakkal, a T szövegben P minta azon eltolásait keressük, amelyekre $T[s\,..\,s+m) = P[0\,..\,m)$. Ezekre az eltolásokra $s \in 0\,..\,n-m$ nyilvánvalóan teljesül.

1.2. Definíció.
$s \in 0\,..\,n-m$ *a* P *minta* lehetséges eltolása (possible shift) *a* T *szövegen.*
$s \in 0\,..\,n-m$ érvényes eltolás (valid shift), **ha** $T[s\,..\,s+m) = P[0\,..\,m)$.
Különben $s \in 0\,..\,n-m$ érvénytelen eltolás (invalid shift).

1.3. Probléma. *(Mintaillesztés.) Az érvényes eltolások* V *halmazát szeretnénk meghatározni. Ehhez eltolások egy* S *halmazába gyűjtjük a mintaillesztő keresések futása során talált érvényes eltolásokat. A problémát megoldó algoritmusok közös utófeltétele* $S = V$*, ahol*

$$V = \{\, s \in 0\,..\,n-m \mid T[s\,..\,s+m) = P[0\,..\,m) \,\}.$$

1.1. Egyszerű mintaillesztés (Naive string-matching)

Vegyük bevezetésként a következő példát! A $P[0\,..\,4) = BABA$ mintát keressük a $T[0\,..\,11) = ABABBABABAB$ szövegben. ($\underline{B}$: B-t sikeresen illesztette a szöveg megfelelő betűjére; $\not{B}$: sikertelenül illesztette.)

Ennél az algoritmusnál sorban megvizsgáljuk a lehetséges eltolásokat, és kiszűrjük közülük az érvényeseket. Úgymond nyers erővel (brute-force) oldjuk meg a problémát.

(The naive string-matching [the Brute-Force] algorithm checks each possible shift in order and collects the valid shifts with maximal (i.e. worst-case) time complexity $\Theta((n-m+1) * m)$, which is $\Theta(n^2)$ if $m = \lfloor n/2 \rfloor$. [2])

$i=$	0	1	2	3	4	5	6	7	8	9	10
$T[i]=$	A	B	A	B	B	A	B	A	B	A	B
	~~B~~	A	B	A							
		$\underline{B}$	$\underline{A}$	$\underline{B}$	~~A~~						
			~~B~~	A	B	A					
				$\underline{B}$	~~A~~	B	A				
$s=4$					$\underline{B}$	$\underline{A}$	$\underline{B}$	$\underline{A}$			
						~~B~~	A	B	A		
$s=6$							$\underline{B}$	$\underline{A}$	$\underline{B}$	$\underline{A}$	
								~~B~~	A	B	A

$$S = \{\, 4; 6 \,\} = V$$

BruteForce($T : \Sigma[n]$; $P : \Sigma[m]$; $S : \mathbb{N}\{\}$)

$S := \{\}$

$s := 0$ **to** $n - m$

match(T, s, P) // $T[s\,..\,s{+}m) = P[0\,..\,m)$

$S := S \cup \{s\}$ | SKIP

match($T : \Sigma[]$; $s : \mathbb{N}$; $P : \Sigma[m]$) : $\mathbb{B}$

// check, whether $T[s\,..\,s{+}m) = P[0\,..\,m)$

$j := 0$

$j < m \wedge T[s + j] = P[j]$

$j{+}{+}$

return $j \geq m$

A $T[s\,..\,s{+}m) = P[0\,..\,m)$ egyenlőségvizsgálatra: $MT_e(m) \in \Theta(m)$
$mT_e(m) \in \Theta(1)$
Innét a teljes algoritmusra: $MT_{\mathrm{BF}}(n, m) \in \Theta((n - m + 1) * m)$
$mT_{\mathrm{BF}}(n, m) \in \Theta(n - m + 1)$

Ha egy feladatosztályon valamely $0 < k < 1$ konstansra $m \leq k * n$, akkor
$1 * n \geq n - m + 1 > n - k * n = (1 - k) * n$, ahol $1 > 1 - k > 0$ konstans.
Innét a $\Theta(\cdot)$ függvényosztályok definíciója szerint
$n - m + 1 \in \Theta(n)$ és $(n - m + 1) * m \in \Theta(n * m)$.
Ebből pedig az $\cdot \in \Theta(\cdot)$ reláció tranzitivitása miatt:

1.4. Következmény. *Ha egy feladatosztályon a k és az m konstansokra*
$0 < k < 1 \wedge m \leq k * n \Rightarrow mT_{BF}(n, m) \in \Theta(n) \wedge MT_{BF}(n, m) \in \Theta(n * m)$

Másrészt, amennyiben egy feladatosztályon m az n-hez képest nem is elhanyagolható, azaz valamely $\varepsilon > 0$ konstansra $m \geq \varepsilon * n$, akkor
$\varepsilon * n * n \leq n * m \leq n * n$. Következésképp $n * m \in \Theta(n^2)$.
Ebből az 1.4 következménnyel adódik az alábbi eredmény:

1.5. Következmény. *Ha egy feladatosztályon az ε és a k konstansokra* $0 < \varepsilon \leq k < 1 \wedge \varepsilon * n \leq m \leq k * n \Rightarrow MT_{BF}(n, m) \in \Theta(n^2)$

1.2. Quicksearch

A gyorsabb keresés érdekében ennél és a következő (KMP) algoritmusnál általában egynél nagyobb lépésekben növeljük a $P[0\,..\,m)$ minta eltolását a $T[0\,..\,n)$ szöveghez képest úgy, hogy biztosan ne ugorjunk át egyetlen érvényes eltolást sem. Mindkét algoritmus, a tényleges mintaillesztés előtt egy előkészítő fázist hajt végre, ami nem függ a szövegtől, csak a mintától.

A Quicksearch-nél ebben az előkészítő fázisban az ábécé σ elemeihez $shift(\sigma) \in 1\,..\,m{+}1$ címkéket társítunk, ahol $P[0\,..\,m)$ a keresett minta.

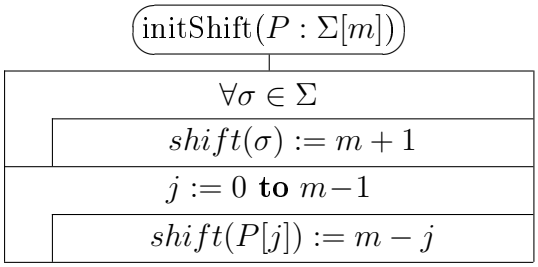
Tegyük fel most, hogy $\sigma = T[s + m]$. Ekkor a $shift(\sigma)$ érték megmondja, hogy a $T[s\,..\,s{+}m) = P[0\,..\,m)$ összehasonlítás után legalább mennyivel kell (jobbra) eltolni a P mintát a szövegen ahhoz, hogy a $T[s{+}m]$ alapján legyen esély a mintának a megfelelő szövegrészhez való illeszkedésére.
– $\sigma \in P[0\,..\,m)$ esetén a $shift(\sigma) \in 1\,..\,m$ érték azt mondja meg, hogy legalább mennyivel kell tovább tolni a P mintát ahhoz, hogy a $T[s + m]$ betűhöz kerülő karaktere maga is σ legyen. Világos, hogy a σ legjobboldali P-beli előfordulásához tartozik a legkisebb ilyen eltolás.
– $\sigma \notin P[0\,..\,m)$ esetén $shift(\sigma) = m + 1$ lesz, azaz a minta *átugorja* a $T[s + m]$ karaktert.

Arra az esetre, amikor az ábécé $\Sigma = \{\text{A,B,C,D}\}$, a minta pedig $P[0\,..\,4)$=CADA, az alábbi félig absztrakt példákban xxxx mutatja a CADA mintával az eltolás előtt összehasonlított szövegrészt, maga a CADA pedig a minta eltolás utáni helyzetét. (Ezután természetesen újabb összehasonlítás kezdődik a szöveg megfelelő része és a minta között stb.)

```
Szöveg: ...xxxxA......xxxxB......xxxxC......xxxxD...
 Minta:     CADA           CADA       CADA      CADA
```

A megfelelő $shift$ értékek értékeket a következő táblázat mutatja.

σ	A	B	C	D
$shift(\sigma)$	1	5	4	2



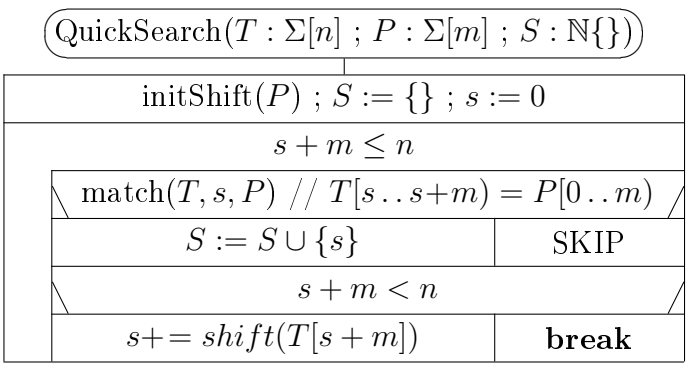
Az ábécé méretét konstansnak tekintve $T_{\text{initShift}}(m) \in \Theta(m)$ adódik.

A $P[0\,..\,4)$=CADA mintával az initShift() majd a Quicksearch() működése:

σ	A	B	C	D
initial $shift(\sigma)$	5	5	5	5
C			4	
A	3			
D				2
A	1			
final $shift(\sigma)$	1	5	4	2

$i=$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
$T[i]=$	A	D	A	B	A	B	C	A	D	A	B	C	A	B	A	D	A	C	A	D	A	D	A
	$\not{C}$	A	D	A																			
		$\not{C}$	A	D	A																		
$s=6$							$\underline{C}$	$\underline{A}$	$\underline{D}$	$\underline{A}$													
												$\underline{C}$	$\underline{A}$	$\not{D}$	A								
														$\not{C}$	A	D	A						
$s=17$																		$\underline{C}$	$\underline{A}$	$\underline{D}$	$\underline{A}$		
																				$\not{C}$	A	D	A

$$S = \{\,6; 17\,\} = V$$



$mT(n,m) \in \Theta\left(\frac{n}{m+1} + m\right)$ (pl. ha $T[0\,..\,n)$ és $P[0\,..\,m)$ diszjunktak)
$MT(n,m) \in \Theta((n-m+2)*m)$ (pl. ha $T = AA\ldots A$ és $P = A\ldots A$)

A minimális műveletigény nagyságrenddel jobb, mint az egyszerű mintaillesztésnél, a tényleges (nem az aszimptotikus) maximális futási idő viszont még egy kicsit rosszabb is. Szerencsére, a tapasztalatok szerint az átlagos futási idő inkább a legjobb esethez áll közel, mintsem a legrosszabbhoz. Időkritikus alkalmazásokban viszont egy stabilabb futási idejű, minden esetben hatékony algoritmusra lenne szükségünk.

1.3. Mintaillesztés lineáris időben (Knuth-Morris-Pratt, azaz KMP algoritmus)

A KMP algoritmus futási ideje minden esetben $\Theta(n)$, a legjobb és a legrosszabb eset között körülbelül kétszeres szorzóval, ami nem csak hatékony működést, de nagyon stabil futási időt is jelent. Ellentétben a korábban ismertetett megoldásokkal, sohasem lép vissza a T szövegen, így a KMP algoritmus könnyen implementálható szekvenciális fájlokra.

A következő speciális jelöléseket fogjuk használni.

1.6. Jelölések.

- *Ha x és y két sztring, akkor $x+y$ a konkatenáltjuk. Pl. $x = AB$ és $y = BA$ esetén $x+y = ABBA$ és $y+x = BAAB$.*
- *ε az üres sztring. Nyilván tetszőleges x sztringre $x+\varepsilon = x = \varepsilon + x$.*
- *Ha x és y két sztring, akkor $x \sqsubseteq y$ (x az y* prefixe*) azt jelenti, hogy $\exists z$ sztring, amire $x+z = y$. ($\varepsilon, A, AB, ABB, ABBA \sqsubseteq ABBA$)*
- *Ha x és y két sztring, akkor $x \sqsubset y$ (x az y* valódi prefixe [proper prefix]*) azt jelenti, hogy $x \sqsubseteq y \wedge x \neq y$. ($\varepsilon, A, AB, ABB \sqsubset ABBA$)*
- *Ha x és y két sztring, akkor $x \sqsupseteq y$ (x az y* szuffixe*) azt jelenti, hogy $\exists z$ sztring, amire $z+x = y$. ($ABBA, BBA, BA, A, \varepsilon \sqsupseteq ABBA$)*
- *Ha x és y két sztring, akkor $x \sqsupset y$ (x az y* valódi szuffixe [proper suffix]*) azt jelenti, hogy $x \sqsupseteq y \wedge x \neq y$. ($BBA, BA, A, \varepsilon \sqsupset ABBA$)*
- *$P_{:j} = P[0\,..\,j) = P[0\,..\,j-1]$ (csak ebben az alfejezetben) $P_{:j}$ a P sztring j hosszúságú prefixe, azaz kezdőszelete. $P_{:0} = \varepsilon$ az üres prefixe. Hasonlóan $T_{:i} = T[0\,..\,i)$.*
  [*Eszerint minden sztringnek* szuffixe *az üres sztring, azaz $P_{:0} \sqsupseteq P_{:j}$*

$(j \in 0\,..\,m)$, *és minden nemüres sztringnek* valódi szuffixe *az üres sztring, azaz* $P_{:0} \sqsupset P_{:j} \quad (j \in 1\,..\,m)$.]

A sztringek alábbi, nyilvánvaló tulajdonságai hasznosak lesznek. [2].

1.7. Lemma. *A szuffixum reláció tranzitivitása (Transitivity of the suffix relation)* $\quad x \sqsupset y \wedge y \sqsupseteq z \Rightarrow x \sqsupset z.$

1.8. Lemma. *Átfedő szuffix (Overlapping-suffix) lemma*
Tegyük fel, hogy x, y és z olyan sztringek, amelyekre $x \sqsupseteq z$ *és* $y \sqsupseteq z$.
$|x| \leq |y| \Rightarrow x \sqsupseteq y. \quad |x| < |y| \Rightarrow x \sqsupset y. \quad |x| = |y| \Rightarrow x = y.$

1.9. Lemma. *Szuffix kiterjesztési (Suffix-extension) lemma*
$P_{:j} \sqsupseteq T_{:i} \wedge P[j] = T[i] \iff P_{:j+1} \sqsupseteq T_{:i+1}.$
$P_{:i} \sqsupset P_{:j} \wedge P[i] = P[j] \iff P_{:i+1} \sqsupset P_{:j+1}.$

Bevezetés a KMP algoritmushoz: Tekintsük a következő példát!

1.10. Példa. *Feltesszük, hogy van egy hosszabb T szövegünk, de most ennek csak a* $T[i-5\,..\,i+2] = BABABABB$ *részét vesszük figyelembe. A minta a* $P_{:6} = BABABB$. *Az aktuális eltolás* $i-5$. *A sikeresen illesztett betűket aláhúztuk. A sikertelenül illesztett karaktert pedig áthúztuk.*

...	T_{i-5}	T_{i-4}	T_{i-3}	T_{i-2}	T_{i-1}	T_i	T_{i+1}	T_{i+2}
...	B	A	B	A	B	A	B	B
$P_{:6} =$	$\underline{B}$	$\underline{A}$	$\underline{B}$	$\underline{A}$	$\underline{B}$	$\not{B}$		
			B	A	B	$\underline{A}$	$\underline{B}$	$\underline{B}$

A táblázat harmadik sorában, sikeresen illesztettük a minta $P_{:5}$ *kezdőszeletét a* $T[i-5\,..\,i)$ *szövegrészhez, de* $P[5] \neq T[i]$. *Következésképpen* $i-5$ *egy érvénytelen eltolás.*

Most a legkisebb további eltolást *keressük úgy, hogy a minta* $P_{:5}$ *kezdőszeletének az a* $P_{:k}$ $(0 \leq k < 5)$ *prefixe, ami még a szöveg* $T[i-k\,..\,i)$ *része alatt van, illeszkedjen arra, azaz* $P_{:k} = T[i-k\,..\,i)$ *teljesüljön. (Lásd a fenti táblázat utolsó sorát!) Ez a* $P_{:k}$*-t meghatározó* legkisebb további eltolás *legfeljebb 5, mert* $P_{:0} \sqsupset T_{:i}$. *Ezzel a* $P_{:k}$*-t meghatározó eltolással biztosan nem ugrunk át egyetlen érvényes eltolást sem. A mi esetünkben két betűvel kell tovább tolni a mintát és* $k = 3$. *Ezután azt találjuk, hogy* $P[3] = T[i]$, $P[4] = [i+1]$ *és* $P[5] = T[i+2]$. *Következőleg* $i-3$ *egy érvényes eltolás. Bármely ennél nagyobb eltolás átugorta volna az* $i-3$ *érvényes eltolást.*

Az előbbi példa felveti a kérdést, hogyan lehetne a k értékét hatékonyan meghatározni. Bár a példa szerint továbbra is $j = 5$, a következő gondolatmenet tetszőleges $P_{:m}$ minta és $T_{:n}$ szöveg esetében alkalmazható, ha $j \in 1\,..\,m$, ahol $i-j$ az aktuális eltolás, $P_{:j} = T[i-j\,..\,i)$ azaz $P_{:j} \sqsupseteq T_{:i}$, valamint $(P[j] \neq T[i] \vee j = m)$.

Világos, hogy nagyobb *további eltoláshoz* kisebb k érték, míg kisebb *további eltoláshoz* nagyobb k érték tartozik, mivel a *további eltolás* $+\ k = j$. Ezenkívül k a *legkisebb további eltoláshoz* tartozik, amelyre $P_{:k} \sqsupset T_{:i}$, ahol ez a *további eltolás* legfeljebb j, ui. $P_{:0} \sqsupset T_{:i}$. Ezért k a legnagyobb h, amire $P_{:h} \sqsupset T_{:i} \wedge 0 \leq h < j$ teljesül. Továbbá $P_{:h} \sqsupset T_{:i}$ ekvivalens a $P_{:h} \sqsupset P_{:j}$ relációval, mert $P_{:j} \sqsupseteq T_{:i} \wedge 0 \leq h < j$. (Más szavakkal, $P[0\,..\,h) = T[i-h\,..\,i)$ ekvivalens a $P[0\,..\,h) = P[j-h\,..\,j)$ relációval, mert $P[0\,..\,j) = T[i-j\,..\,i) \wedge 0 \leq h < j$.)

Következésképpen k csak $P_{:j}$-től függ. Mivel $P_{:m}$ fix, k csak j-től függ. Más szavakkal, a leghosszabb $P_{:h}$-t keressük, amire $P_{:h} \sqsubset P_{:j} \wedge P_{:h} \sqsupset P_{:j}$. Ennek hosszát a π prefix függvény határozza meg.

1.11. Definíció. $\pi(j) = \max\{\, h \in 0\,..\,j-1 \mid P_{:h} \sqsupset P_{:j} \,\} \quad (j \in 1\,..\,m)$

Ennek a függvénynek a hatékony kiszámítását később, 1.3.2-ben részletezzük. Előbb azonban megnézzük a KMP algoritmus még egy, konkrét esetét, majd 1.3.1-ben formálisan is megadjuk és elemezzük a fő eljárás struktogramját. A π prefix függvény következő tulajdonságai megkönnyítik a számolást.

1.12. Tulajdonság. $j \in 1\,..\,m \Rightarrow \pi(j) \in 0\,..\,j-1$

Bizonyítás. A 1.11. definíció szerint $\pi(j)$ a $0\,..\,j-1$ egy részhalmazának a maximuma. □

1.13. Lemma. $j \in [1\,..\,m) \Rightarrow \pi(j+1) \leq \pi(j)+1$

Bizonyítás.
– Ha $\pi(j+1) = 0 \Rightarrow \pi(j+1) = 0 \leq 0+1 \leq \pi(j)+1$ mivel $\pi(j) \geq 0$ az 1.12. tulajdonság szerint.
– Ha $\pi(j+1) > 0 \Rightarrow$ a prefix függvény definíciója (1.11) szerint
$P_{:(\pi(j+1)-1)+1} = P_{:\pi(j+1)} \sqsupset P_{:j+1} \Rightarrow$ az 1.9. lemmával $P_{:\pi(j+1)-1} \sqsupset P_{:j} \Rightarrow$ újra az 1.11. definícióval $\pi(j+1)-1 \leq \pi(j) \Rightarrow \pi(j+1) \leq \pi(j)+1$. □

1.14. Példa. *Kiszámítjuk a π függvényt az 1.11. definíció, az 1.12. tulajdonság és az 1.13. lemma szerint a $P_{:8} = P[0\,..\,8) = BABABBAB$ mintára.*

$P[j-1] =$	B	A	B	A	B	B	A	B
$j =$	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
$\pi(j) =$	*0*	*0*	*1*	*2*	*3*	*1*	*2*	*3*

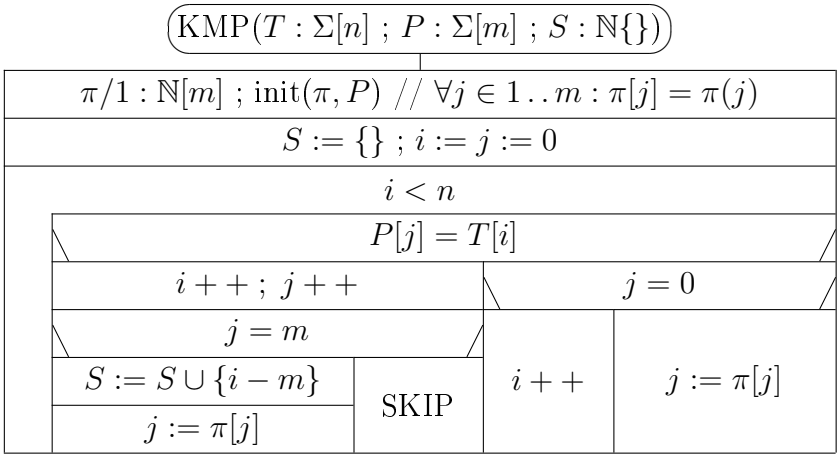
1.15. Példa. *Alkalmazzuk az eddig tanultakat a következő esetben! A* $P_{:8} = P[0\,..\,8) = BABABBAB$ *minta előfordulásait keressük a* $T_{:18} = T[0\,..\,18) = ABABABABBABABABBAB$ *szövegben. (A minta elején a jelöletlen betűkről „illesztés nélkül is tudja" az algoritmus, hogy illeszkednek a szöveg megfelelő karakterére.* $\underline{B}$*: B-t sikeresen illesztette a szöveg megfelelő betűjére;* $\not{B}$*: sikertelenül illesztette.)*

$i=$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$T[i]=$	A	B	A	B	A	B	A	B	B	A	B	A	B	A	B	B	A	B
	$\not{B}$																	
		$\underline{B}$	$\underline{A}$	$\underline{B}$	$\underline{A}$	$\underline{B}$	$\not{B}$											
$s=3$				B	A	B	$\underline{A}$	$\underline{B}$	$\underline{B}$	$\underline{A}$	$\underline{B}$							
									B	A	B	$\underline{A}$	$\underline{B}$	$\not{B}$				
$s=10$											B	A	B	$\underline{A}$	$\underline{B}$	$\underline{B}$	$\underline{A}$	$\underline{B}$
																B	A	B

$$S = \{\,3; 10\,\} = V$$

1.3.1. A KMP algoritmus fő eljárása

Az 1.10. példával kapcsolatos megfontolások alapján megadjuk a KMP algoritmus fő eljárásának struktogramját.



A 1.3.2 alfejezetben fogjuk látni, hogy az init(π, P) eljáráshívás a π prefix-függvény értékeit összegyűjti a π tömbbe, $\Theta(m)$ műveletigénnyel. Az init(π, P) eljáráshívás utófeltétele az, hogy $(\forall j \in 1\,..\,m : \pi[j] = \pi(j))$.

A fő eljárás elemzése a ciklusa 1.17. invariánsán alapszik, ahol $i-j$ a P minta aktuális eltolása a T szövegen. Az invariáns helyességének bizonyításához hasznos lesz a következő lemma.

1.16. Lemma. $j \in 1\,..\,m \wedge P_{:j} \sqsupseteq T_{:i} \Rightarrow$
nincs érvényes eltolás az $(i{-}j\,..\,i{-}\pi(j))$ *intervallumban.*

Bizonyítás. Tegyük fel indirekt módon, hogy $k \in (\pi(j)\,..\,j)$ és $i{-}k$ érvényes eltolás! Ez azt jelenti, hogy $T[i{-}k\,..\,i{-}k{+}m) = P[0\,..\,m)$. Nyilván $k < j \leq m$. Innét $k < m$, amiből $i{-}k{+}m > i$ adódik. Ezért $T[i{-}k\,..\,i) = P[0\,..\,k)$, vagy másképp írva $P_{:k} \sqsupseteq T_{:i}$. Továbbá $P_{:j} \sqsupseteq T_{:i} \wedge k < j$. Következésképpen $P_{:k} \sqsupset P_{:j}$ az Átfedő szuffix lemma (1.8) miatt, de $k > \pi(j)$, amiből viszont $P_{:k} \not\sqsupset P_{:j}$ következik, ui. $P_{:\pi(j)}$ a $P_{:j}$ leghosszabb valódi prefixe, ami egyben szuffixe is, a π prefix függvény definíciója (1.11) alapján. □

1.17. Tétel. *Az* (Inv) *állítás a* $\mathrm{KMP}(T,P,S)$ *eljárás ciklusának invariánsa.*
(Inv) $\quad P_{:j} \sqsupseteq T_{:i} \wedge 0 \leq j \leq i \leq n \wedge j < m \ \wedge S = V \cap [0\,..\,i{-}j).$

Bizonyítás. Kezdetben $i = j = 0$ miatt (Inv) a következő nyilvánvaló állításnak felel meg.
$P_{:0} \sqsupseteq T_{:0} \wedge 0 \leq 0 \leq 0 \leq n \wedge 0 < m \ \wedge S = V \cap [0\,..\,0) = V \cap \{\} = \{\}.$

A továbbiakban igazoljuk, hogy a ciklusmag végrehajtása (mind a négy programágon) tartja (Inv)-et. Állításainkhoz mindig hozzáértjük $\mathrm{init}(\pi,P)$ utófeltételét. Feltéve, hogy $i < n$, akkor belépünk a ciklusba, és
(Inv1) $\quad P_{:j} \sqsupseteq T_{:i} \wedge 0 \leq j \leq i < n \wedge j < m \ \wedge S = V \cap [0\,..\,i{-}j)$ $\quad$ teljesül.

- Ha $P[j] = T[i]$, akkor a Szuffix kiterjesztési lemma (1.9) szerint $P_{:j+1} \sqsupseteq T_{:i+1}$, majd i és j növelése után
  (Inv2) $\quad P_{:j} \sqsupseteq T_{:i} \wedge 0 < j \leq i \leq n \wedge j \leq m \ \wedge S = V \cap [0\,..\,i{-}j).$

  1. Amennyiben $j = m$, akkor $P_{:m} \sqsupseteq T_{:i}$, vagy másképpen írva $P[0\,..\,m) = T[i{-}m\,..\,i)$. Tehát $i{-}m$ érvényes eltolás, amit S-hez hozzávéve
     (Inv3) $\quad P_{:j} \sqsupseteq T_{:i} \wedge 0 < j \leq i \leq n \wedge j \leq m \ \wedge S = V \cap [0\,..\,i{-}j].$
     Mivel $j \in 1\,..\,m \wedge P_{:j} \sqsupseteq T_{:i}$, az 1.16. lemma szerint nincs érvényes eltolás az $(i{-}j\,..\,i{-}\pi(j))$ intervallumban. Ezért
     $P_{:j} \sqsupseteq T_{:i} \wedge 0 < j \leq i \leq n \wedge j \leq m \ \wedge S = V \cap [0\,..\,i{-}\pi(j)).$
     Figyelembe véve, hogy $P_{:\pi(j)} \sqsupset P_{:j} \sqsupseteq T_{:i}$, a szuffixum reláció tranzitivitása (1.7. lemma) és $\pi[j] = \pi(j)$ alapján:
     $P_{:\pi[j]} \sqsupset T_{:i} \wedge 0 < j \leq i \leq n \wedge j \leq m \ \wedge S = V \cap [0\,..\,i{-}\pi[j]).$
     Tekintettel arra, hogy $\pi[j] = \pi(j) \in [0\,..\,j)$, a $j := \pi[j]$ értékadás után már $0 \leq j < i \wedge j < m$ teljesül. Innét
     $P_{:j} \sqsupset T_{:i} \wedge 0 \leq j < i \leq n \wedge j < m \ \wedge S = V \cap [0\,..\,i{-}j)$ adódik, amiből az első programág végén közvetlenül következik (Inv).
  2. Amennyiben $j \neq m$, akkor viszont (Inv2) szerint $j < m$, és így a második programág végén is teljesül (Inv).

- Ha $P[j] \neq T[i]$, akkor a Szuffix kiterjesztési lemma (1.9) szerint $P_{:j+1} \not\sqsupseteq T_{:i+1}$, vagy ezt másképpen írva $P[0\,..\,j] \neq T[i-j\,..\,i]$. (Inv1)-ből $j < m$ figyelembevételével ebből $P[0\,..\,m) \neq T[i-j\,..\,i-j+m)$ következik. Tehát az $i-j$ érvénytelen eltolás. Ebből az (Inv1), azaz a $P_{:j} \sqsupseteq T_{:i} \wedge 0 \leq j \leq i < n \wedge j < m \ \wedge S = V \cap [0\,..\,i-j)$ tulajdonsággal (Inv4) $P_{:j} \sqsupseteq T_{:i} \wedge 0 \leq j \leq i < n \wedge j < m \ \wedge S = V \cap [0\,..\,i-j]$ adódik.

  3. Ha $j = 0$, akkor (Inv4) figyelembevételével a $P_{:0} \sqsupseteq T_{:i} \wedge 0 = j \leq i < n \wedge j < m \ \wedge S = V \cap [0\,..\,i-j]$ állítást kapjuk, amiből i növelése után $P_{:0} \sqsupseteq T_{:i} \wedge 0 = j \leq i \leq n \wedge j < m \ \wedge S = V \cap [0\,..\,i-j)$ következik, tehát a harmadik programág végén is teljesül (Inv) $P_{:j} \sqsupseteq T_{:i} \wedge 0 \leq j \leq i \leq n \wedge j < m \ \wedge S = V \cap [0\,..\,i-j)$.
  4. Ha viszont $j \neq 0$, akkor (Inv4) figyelembevételével $P_{:j} \sqsupseteq T_{:i} \wedge 0 < j \leq i < n \wedge j < m \ \wedge S = V \cap [0\,..\,i-j]$ teljesül. Ennek közvetlen következménye (Inv3) $P_{:j} \sqsupseteq T_{:i} \wedge 0 < j \leq i \leq n \wedge j \leq m \ \wedge S = V \cap [0\,..\,i-j]$. Az első programág vizsgálatánál már láttuk, hogy (Inv3) esetén a $j := \pi[j]$ értékadást végrehajtva teljesül (Inv), tehát az utolsó programág végén is igaz.

□

A KMP(T, P, S) eljárás parciális helyessége:

1.18. Tétel. *Ha a KMP algoritmus megáll, akkor megoldja az 1.3. (mintaillesztési) problémát, azaz $S = V$ teljesül, amikor a* KMP(T, P, S) *eljárás befejeződik.*

Bizonyítás. A KMP(T, P, S) eljárás ciklusának (Inv) invariánsa (1.17), azaz $P_{:j} \sqsupseteq T_{:i} \wedge 0 \leq j \leq i \leq n \wedge j < m \wedge S = V \cap [0\,..\,i-j)$, és a ciklusfeltétel tagadása, vagyis $i \geq n$ szerint $i = n \wedge j < m \wedge S = V \cap [0\,..\,n-j)$ teljesül, mikor a ciklus befejeződik. Ezenkívül, $j < m \Rightarrow n-j > n-m \Rightarrow [0\,..\,n-j) \supseteq 0\,..\,n-m \supseteq \{\, s \in 0\,..\,n-m \mid T[s\,..\,s+m) = P[0\,..\,m) \,\} = V \Rightarrow [0\,..\,n-j) \supseteq V$. Ezért $S = V \cap [0\,..\,n-j) = V$. Következőleg, $S = V$ teljesül, amikor a KMP(T, P, S) eljárás befejeződik. □

A KMP(T, P, S) eljárás megállása és hatékonysága: A terminálás igazolásához az 1.3.2. alfejezetben be fogjuk látni, hogy az init(π, P) eljárás $\Theta(m)$ idő alatt lefut, most pedig azt, hogy a KMP(T, P, S) eljárás ciklusa legfeljebb $2n$ iterációt hajt végre.

A KMP(T, P, S) eljárás lineáris, pontosabban $\Theta(n)$ műveletigényének igazolásához elég belátni, hogy a fő ciklusának a futási ideje $\Theta(n)$, ui. $m \in 1\,..\,n$ miatt ezen az init(π, P) eljárás (1.3.2) és egyéb inicializálások $\Theta(m)$ műveletigénye aszimptotikus nagyságrendben már nem módosít. A fő ciklus $\Theta(n)$ műveletigényének igazolásához pedig elegendő azt belátni, hogy az legalább n és legfeljebb $2n$ iterációt hajt végre. A ciklus (Inv) invariánsa (1.17) szerint $i \in 0\,..\,n \wedge j \in 0\,..\,(m{-}1) \wedge j \leq i$.

- Mivel az első iteráció előtt $i = 0$, és mindegyik iteráció legfeljebb eggyel növeli az i változót, továbbá a ciklusfeltétel $i < n$, azért legalább n iteráció szükséges ahhoz, hogy $i = n$ legyen, és az eljárás befejeződjék.
- Tekintsük a $2i{-}j$ kifejezést! Az $i \in 0\,..\,n \wedge j \in 0\,..\,(m{-}1) \wedge j \leq i$ invariáns tulajdonság miatt $2i{-}j \in 0\,..\,2n$.

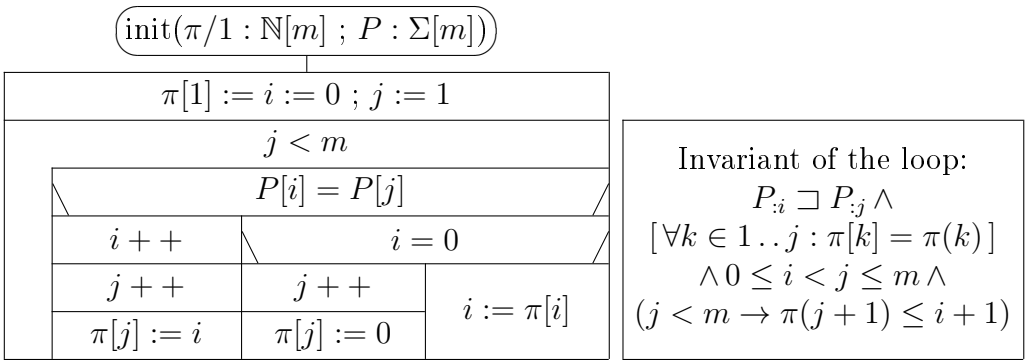
  Az első iteráció előtt $2i{-}j = 0$, ami mindegyik iterációval (mind a négy programágon) szigorúan monoton nő, és végig $2i{-}j \leq 2n$, így legfeljebb $2n$ iteráció után megáll a ciklus.

1.3.2. A π prefix tömb inicializálása

A következő, a π prefix *függvényre* vonatkozó lemma hasznos lesz.

1.19. Lemma. $P_{:i} \sqsupset P_{:j} \wedge 0 < i < j < m \wedge \pi(j{+}1) \leq i \Rightarrow \pi(j{+}1) \leq \pi(i){+}1$

Bizonyítás. Ha $\pi(j{+}1) = 0$, akkor $\pi(j{+}1) < 0{+}1 \leq \pi(i){+}1$, mert definíció szerint $\pi(i) \geq 0$. Másrészt, amennyiben $\pi(j+1) > 0$, $k := \pi(j+1) - 1$. Így $k \geq 0$ és $k + 1 = \pi(j + 1)$. A π függvény definíciója szerint $P_{:k+1} \sqsupset P_{:j+1}$. Következésképpen $P_{:k} \sqsupset P_{:j}$. Figyelembe véve, hogy $P_{:i} \sqsupset P_{:j}$ és $k < i$, a $P_{:k} \sqsupset P_{:i}$ állítást kapjuk. Következőleg $k \leq \pi(i)$. Innét pedig $\pi(j + 1) = k + 1 \leq \pi(i) + 1$ adódik. □



Az init(π, P) eljárás elemzése a kódja mellett, és az alábbi, 1.20. tételben található ciklusinvariánsán alapszik.

1.20. Tétel. *Az* (inv) *állítás az* $\mathrm{init}(\pi, P)$ *eljárás ciklusának invariánsa.*

(inv) $\quad P_{:i} \sqsupset P_{:j} \wedge (\,\forall k \in 1\,..\,j : \pi[k] = \pi(k)\,) \wedge$
$\quad 0 \le i < j \le m \wedge (\,j < m \to \pi(j+1) \le i+1\,).$

Bizonyítás.Közvetlenül az első ciklusiteráció előtt a $\pi[1] := i := 0$; $j := 1$ inicializálások miatt (inv) a következő állításnak felel meg: $P_{:0} \sqsupset P_{:1} \wedge (\,\forall k \in 1\,..\,1 : \pi[k] = \pi(k) = \pi(1) = 0\,) \wedge 0 \le 0 < 1 \le m \wedge (\,1 < m \to \pi(2) \le 1\,)$. Ez utóbbi tényezői igazak, sorban, mert a $P_{:0}$ üres sztring bármely nem nemüres sztringnek valódi szuffixe, továbbá az 1.12. tulajdonság szerint $\pi(1) = 0$, valamint a P minta m mérete nem nulla, és végül a 1.13. lemma alapján $\pi(1+1) \le \pi(1) + 1 = 1$, feltéve, hogy $m > 1$.

Belátjuk még, hogy a ciklusiterációk tartják az (inv) tulajdonságot, azaz, ha tetszőleges ciklusiteráció előtt (inv) és a ciklusfeltétel igaz, akkor a ciklusmag bármelyik ágának a végére jutva ugyancsak igaz lesz. Amikor belépünk a ciklusmagba, akkor (inv) és a ciklusfeltétel alapján (inv1) teljesül:

(inv1) $\quad P_{:i} \sqsupset P_{:j} \wedge (\,\forall k \in 1\,..\,j : \pi[k] = \pi(k)\,) \wedge$
$\quad 0 \le i < j < m \wedge \pi(j+1) \le i+1.$

1. Ha $P[i] = P[j]$, akkor a az 1.9. lemma és (inv1)-ből $P_{:i} \sqsupset P_{:j}$ alapján $P_{:i+1} \sqsupset P_{:j+1}$. Innét a π prefix függvény definíciója (1.11) szerint $\pi(j+1) \ge i+1$, (inv1) alapján pedig $\pi(j+1) \le i+1$. Következésképpen $\pi(j+1) = i+1$. Az $i{+}{+}; j{+}{+}; \pi[j] := i$ értékadások után pedig $P_{:i} \sqsupset P_{:j} \wedge (\,\forall k \in 1\,..\,j : \pi[k] = \pi(k)\,) \wedge 0 < i < j \le m \wedge \pi(j) = i$. Amennyiben $j < m$, az 1.13. lemma és $\pi(j) = i$ szerint $\pi(j+1) \le \pi(j) + 1 = i + 1$. A ciklusmag első ágának végén tehát teljesül az (inv) állítás.

2-3. Ha $P[i] \neq P[j]$, akkor az 1.9. lemma alapján $P_{:i+1} \not\sqsupset P_{:j+1}$. Innét a π prefix függvény definíciója (1.11) szerint $\pi(j+1) \neq i+1$, (inv1) alapján pedig $\pi(j+1) \le i+1$, tehát $\pi(j+1) \le i$.
Ezt (inv1)-gyel összevetve, a belső elágazás előtt (inv2) teljesül:

(inv2) $\quad P_{:i} \sqsupset P_{:j} \wedge (\,\forall k \in 1\,..\,j : \pi[k] = \pi(k)\,) \wedge$
$\quad 0 \le i < j < m \wedge \pi(j+1) \le i.$

   2. Amennyiben $i = 0$, akkor (inv2)-ből a $\pi(j+1) \le i$ állítást figyelembe véve $\pi(j+1) = 0$ adódik, mivel a π függvény nemnegatív. Ezt (inv2)-vel összevetve, a $j{+}{+};\ \pi[j] := 0$ értékadások után $P_{:i} \sqsupset P_{:j} \wedge (\,\forall k \in 1\,..\,j : \pi[k] = \pi(k)\,) \wedge 0 = i < j \le m \wedge \pi(j) = i$. Amennyiben $j < m$, az 1.13. lemma és $\pi(j) = i$ szerint $\pi(j+1) \le \pi(j) + 1 = i + 1$. Tehát a ciklusmag második ágának végén is teljesül az (inv) állítás.

3. Amennyiben $i \neq 0$, akkor (inv2) szerint (inv3) teljesül:
   (inv3) $\quad P_{:i} \sqsupset P_{:j} \wedge (\,\forall k \in 1\,..\,j : \pi[k] = \pi(k)\,) \wedge$
   $0 < i < j < m \wedge \pi(j+1) \leq i.$
   Mivel $i > 0$, az 1.19. lemma feltételei teljesülnek. Tehát $\pi(j+1) \leq \pi(i) + 1$. Másrészt, (inv3) 2. és 3. tényezője figyelembevételével $\pi[i] = \pi(i)$. Ebből π prefix függvény definíciójával (1.11) $P_{:\pi[i]} \sqsupset P_{:i}$ adódik. Ebből az (inv3) a $P_{:i} \sqsupset P_{:j}$ állításával együtt, az 1.7. tranzitivitási lemma alapján $P_{:\pi[i]} \sqsupset P_{:j}$.
   Ezekből (inv3) alapján, az $i := \pi[i]$ értékadás után
   $P_{:i} \sqsupset P_{:j} \wedge (\,\forall k \in 1\,..\,j : \pi[k] = \pi(k)\,) \wedge$
   $0 \leq i < j < m \wedge \pi(j+1) \leq i + 1.$
   Tehát a ciklusmag utoló ágának a végén is teljesül az (inv) állítás.

□

Az init(π, P) eljárás parciális helyessége:

1.21. Következmény. *Ha az* init(π, P) *eljárás megáll, akkor a visszatérésekor teljesül az utófeltétele:*

$$\forall k \in 1\,..\,m : \pi[k] = \pi(k)$$

Bizonyítás. Az *init*(π, P) eljárásnak az 1.20. tételből ismerős (inv) invariánsa és a ciklusfeltétel tagadása, azaz $j \geq m$ alapján $j = m$. Figyelembe véve még az (inv) invariáns $(\,\forall k \in 1\,..\,j : \pi[k] = \pi(k)\,)$ tényezőjét, az init(π, P) eljárás fenti utófeltétele azonnal adódik. □

Az init(π, P) eljárás megállása és hatékonysága: A megálláshoz az alábbiakban igazoljuk, hogy az eljárás ciklusa legfeljebb $2m-2$ iterációt hajt végre.

Mivel a ciklus minden egyes iterációja $\Theta(1)$ műveletigényű, a lineáris, pontosabban $\Theta(m)$ műveletigény igazolásához elég belátni, hogy az init(π, P) eljárás ciklusa legalább $m-1$ és legfeljebb $2m-2$ iterációt hajt végre. A ciklus (inv) invariánsa szerint $0 \leq i < j \leq m$.

- Mivel az első iteráció előtt $j = 1$, továbbá mindegyik iteráció legfeljebb eggyel növeli a j változót, és a ciklusfeltétel $j < m$, legalább $m-1$ iteráció szükséges ahhoz, hogy $j = m$ legyen, és az eljárás befejeződjék.

- Tekintsük a $2j-i$ kifejezést! $0 \leq i < j \leq m$ miatt $2j-i \in 2\,..\,2m$; ui. $j-i \geq 1 \wedge j \geq 1 \Rightarrow 2j-i \geq 2$; továbbá $j \leq m \Rightarrow 2j \leq 2m \Rightarrow 2j-i \leq 2m$, mivel $i \geq 0$.

Az első iteráció előtt $2j-i = 2$, ami mindegyik iterációval (mind a három programágon) szigorúan monoton nő, és végig $2j-i \leq 2m$, így legfeljebb $2m-2$ iteráció után megáll a ciklus.

1.3.3. A KMP algoritmus összegzése

A fentiekben egy *viszonylag* egyszerű és rövid bizonyítást sikerült adnunk a lineáris műveletigényű KMP algoritmus helyességére és hatékonyságára. (Érdemes összehasonlítani a [2]-ben olvasható bizonyítással. Kikerültük a *mintaillesztő automaták* bevezetését, elemzését és szimulálását.) Ehhez

- megvizsgáltuk a kérdéses sztringek megfelelő tulajdonságait,
- meghatároztuk a KMP(T, P, S) és az init(π, P) eljárások ciklusainak megfelelő invariáns tulajdonságait, valamint
- a ciklusokhoz tartozó alkalmas terminátor kifejezéseket.

Láttuk, hogy a KMP algoritmusnál legjobb és a legrosszabb eset absztrakt műveletigénye között kétszeres szorzó van, így a futási idő is nagyon stabil, és a legrosszabb esetben is meglepően hatékony, a szöveg hosszának közelítőleg lineáris függvénye. Online alkalmazásoknál a hatékonyság (a legrosszabb esetben is) és a futási idő stabilitása együtt, határozott előny a másik két algoritmusal szemben, bár a Quicksearch várható futási ideje jobb, mint a KMP algoritmusé, így offline alkalmazásoknál ez lehet előnyösebb.

A KMP(T, P, S) eljárás i változója sohasem csökken. Mivel a $T[0\,..\,n)$ szövegre csak a $T[i]$ kifejezésen keresztül hivatkozunk, a szövegen sohasem kell visszalépni. Ezért ez a KMP algoritmus kényelmesen, hatékonyan implementálható abban az esetben is, ha a szöveg (amiben keressük a mintát) egy szekvenciális fájlban van. Mivel a Brute-force és a Quicksearch algoritmusoknál a szövegen esetleg $m-2$ karakternyit is vissza kell lépni, ezeknél – feltéve, hogy a szöveg egy szekvenciális input fájlban van – annak az utoljára beolvasott $m-1$ karakterét folyamatosan nyilván kell tartani egy átmeneti tárolóban.

1.3.4. A KMP algoritmus szemléltetése

Az init(π, P) algoritmus szemléltetése az $ABABBABA$ mintán:
(A három programág mindegyikének az elején kezdünk új sort.)

i	j	$\pi[j]$	$\overset{0}{A}$	$\overset{1}{B}$	$\overset{2}{A}$	$\overset{3}{B}$	$\overset{4}{B}$	$\overset{5}{A}$	$\overset{6}{B}$	$\overset{7}{A}$
0	1	0		$\not{A}$						
0	2	0			$\underline{A}$					
1	3	1			A	$\underline{B}$				
2	4	2			A	B	$\not{A}$			
0	4	2					$\not{A}$			
0	5	0						$\underline{A}$		
1	6	1						A	$\underline{B}$	
2	7	2						A	B	$\underline{A}$
3	8	3								

A végeredmény:

$P[j-1] =$	A	B	A	B	B	A	B	A
$j =$	1	2	3	4	5	6	7	8
$\pi[j] =$	0	0	1	2	0	1	2	3

Példa:
A $P[0\,..\,8) = ABABBABA$ mintát keressük
a $T[0\,..\,17) = ABABABBABABBABABA$ szövegben.

A mintához tartozó $\pi/1 : \mathbb{N}[8]$ tömböt fentebb már kiszámoltuk.

A keresés:

$i = 0$		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$T[i]=$	A	B	A	B	A	B	B	A	B	A	B	B	A	B	A	B	A
	$\underline{A}$	$\underline{B}$	$\underline{A}$	$\underline{B}$	$\not{B}$												
s=2			A	B	$\underline{A}$	$\underline{B}$	$\underline{B}$	$\underline{A}$	$\underline{B}$	$\underline{A}$							
s=7								A	B	A	$\underline{B}$	$\underline{B}$	$\underline{A}$	$\underline{B}$	$\underline{A}$		
													A	B	A	$\underline{B}$	$\not{B}$
															A	B	$\underline{A}$

$$S = \{\, 2; 7 \,\} = V$$

2. Információtömörítés ([4] 5)

2.1. Naiv módszer

A tömörítendő szöveget karakterenként, fix hosszúságú bitsorozatokkal kódoljuk.

$\Sigma = \langle \sigma_1, \sigma_2, \ldots, \sigma_d \rangle$ az ábécé.

Egy-egy karakter $\lceil \lg d \rceil$ bittel kódolható, ui. $\lceil \lg d \rceil$ biten $2^{\lceil \lg d \rceil}$ különböző bináris kód ábrázolható, és $2^{\lceil \lg d \rceil} \geq d > 2^{\lceil \lg d \rceil - 1}$, azaz $\lceil \lg d \rceil$ biten ábrázolható d-féle különböző kód, de eggyel kevesebb biten már nem.

$In : \Sigma\langle\rangle$ a tömörítendő szöveg. $n = |In|$ jelöléssel $n * \lceil \lg d \rceil$ bittel kódolható.

Pl. az ABRAKADABRA szövegre $d = 5$ és $n = 11$, ahonnét a tömörített kód hossza $11 * \lceil \lg 5 \rceil = 11 * 3 = 33$ bit. (A 3-bites kódok közül tetszőleges 5 kioszthatóaz 5 betűnek.) A tömörített fájl a kódtáblázatot is tartalmazza.

A fenti ABRAKADABRA szöveg kódtáblázata lehet pl. a következő:

karakter	kód
A	000
B	001
D	010
K	011
R	100

A fenti kódtáblázattal a tömörített kód a következő lesz:
00000110000001100001000000011000000.

Ez a tömörített fájlba foglalt kódtáblázat alapján könnyedén 3 bites szakaszokra bontható és kitömöríthető. A kódtáblázat mérete miatt a gyakorlatban csak hosszabb szövegeket érdemes így tömöríteni.

2.2. Huffman-kód

A tömörítendő szöveget karakterenként, változó hosszúságú bitsorozatokkal kódoljuk. A gyakrabban előforduló karakterek kódja rövidebb, a ritkábban előfordulóké hosszabb.

Prefix-mentes kód: Egyetlen karakter kódja sem prefixe semelyik másik karakter kódjának sem.

A karakterenként kódoló tömörítések között a Huffman-kód hossza minimális. Ugyanahhoz a szöveghez többféle kódfa és hozzátartozó kódtáblázat építhető, de mindegyik segítségével az input szövegnek ugyanolyan hosszú tömörített kódját kapjuk. Betömörítés a kódtáblával, kitömörítés a kódfával. Ezért a tömörített fájl a kódfát is tartalmazza.

A **tömörítendő fájlt, illetve szöveget kétszer olvassa végig.**
- Először meghatározza a szövegben előforduló karakterek halmazát és az egyes karakterek gyakoriságát, majd ennek alapján kódfát, abból pedig kódtáblázatot épít.
- Másodszorra a kódtábla alapján kiírja az output fájlba sorban a karakterek bináris kódját.

A **kódfa** szigorúan bináris fa. Mindegyik karakterhez tartozik egy-egy levele, amit a karakteren kívül annak gyakorisága, azaz előfordulásainak száma is címkéz. A belső csúcsokat a csúcshoz tartozó részfa leveleit címkéző karakterek gyakoriságainak összegével címkézzük. (Így a kódfa gyökerét a tömörítendő szöveg hossza címkézi.)

A **kódfát** úgy**építjük** fel, hogy először egycsúcsú fák egy minimum-prioritásos sorát határozzuk meg, amelyben mindegyik karakter pontosan egy csúcsot címkéz. A csúcsot a karakteren kívül annak gyakorisága is címkézi. A minimum-prioritásos sort a benne tárolt fák gyökerét címkéző gyakoriságértékek szerint építjük fel. Ezután a következőt csináljuk ciklusban, amíg a kupac még legalább kettő fából áll.

Kiveszünk a kupacból egy olyan fát, amelyeknek gyökerét a legkisebb gyakoriság címkézi. Ezután a maradék kupacra ezt még egyszer megismételjük. Összeadjuk a két gyakoriságot. Az összeggel címkézünk egy új csúcsot, amelynek bal és jobb részfája az előbb kiválasztott két fa lesz. A bal ágat a 0, a jobb ágat az 1 címkézi. Az így képzett új fát visszatesszük a minimum-prioritásos sorba.

A fenti ciklus után a minimum-prioritásos sorban maradó egyetlen bináris fa a Huffman-féle kódfa.

A kódfából ezután kódtáblázatot készítünk. Mindegyik karakterekhez tartozó kódot úgy kapjuk meg, hogy a kódfa gyökerétől elindulva és a karakterhez tartozó levélig lefelé haladva a kódfa éleit címkéző biteket összeolvassuk. (Ezt hatékonyan kivitelezhetjük pl. a kódfa preorder bejárásával, az aktuális csúcshoz vezető bitsorozat folyamatos nyilvántartásával, és levélhez érve, a kódtáblázatba írásával.)

Befejezésül újra végigolvassuk a tömörítendő szöveget, és a kódtáblázat segítségével sorban mindegyik karakter bináris kódját a (kezdetben üres) tömörített bitsorozat végéhez fűzzük. A tömörített fájl a kódfát is tartalmazza,

így a gyakorlatban Huffman-kódolással is csak hosszabb szövegeket érdemes tömöríteni.

A **kitömörítést** is karakterenként végezzük. Mindegyik karakter kinyeréséhez a kódfa gyökerétől indulunk, majd a tömörített kód sorban olvasott bitjeinek hatására 0 esetén balra, 1 esetén jobbra lépünk lefelé a fában, mígnem levélcsúcshoz érünk. Ekkor kiírjuk a levelet címkéző karaktert, majd a Huffman-kódban a következő bittől és újra a kódfa gyökerétől folytatjuk, amíg a tömörített kódon végig nem érünk.

2.2.1. Huffman-kódolás szemléltetése

Pl. az *ABRAKADABRA* szöveget egyszer végigolvasva meghatározhatjuk milyen karakterek fordulnak elő a szövegben, és milyen gyakorisággal. Úgy képzelhetjük, hogy az alábbi táblázat az új betűkkel folyamatosan bővül, ahogy haladunk előre a szövegben.

szöveg:	A	B	R	A	K	A	D	A	B	R	A
A	1			2		3		4			5
B	-	1							2		
D	-	-	-	-	-	-	1				
K	-	-	-	-	1						
R	-	-	1							2	

A fenti számolást (betű/gyakoriság) alakban összegezve:

$$\langle (D/\mathbf{1}), (K/\mathbf{1}), [B/\mathbf{2}], \{R/\mathbf{2}\}, (A/\mathbf{5}) \rangle$$

A fenti öt kifejezést öt egycsúcsú bináris fának tekinthetjük. (A jobb olvashatóság kedvéért többféle zárójelpárt alkalmaztunk.) Mindegyik csúcs egyben levél és gyökér. A levelekhez tartozó két címkét *karakter/gyakoriság* alakban írtuk le. A tömörítés algoritmusa szerint ezeket egy minimum-prioritásos sorba tesszük. A könnyebb érthetőség kedvéért ezt a minimum-prioritásos sort a szokásos minimum-kupacos reprezentáció helyett most a fák gyökerében lévő gyakoriság-értékek (röviden *fa-gyakoriság-értékek*) szerint rendezett fa-sorozattal szemléltetjük. (Azonos gyakoriságok esetén a betűk alfabetikus sorrendje szerint rendezünk. Ez ugyan önkényes, de az algoritmus bemutatása szempontjából hasznos egyértelműsítés. A fák ágait is hasonlóképpen rendezzük sorba.)

Ezután kivesszük a két legkisebb gyakoriság-értékű fát, egy új gyökércsúcs alá tesszük őket bal- és jobboldali részfának, a új gyökércsúcsot pedig a két

fa-gyakoriság-érték összegével címkézzük. Végül visszatesszük az új fát a minimum-prioritásos sorba.

$$\langle [B/\mathbf{2}], [(D/1)\mathbf{2}(K/1)], \{R/\mathbf{2}\}, (A/\mathbf{5})\rangle$$

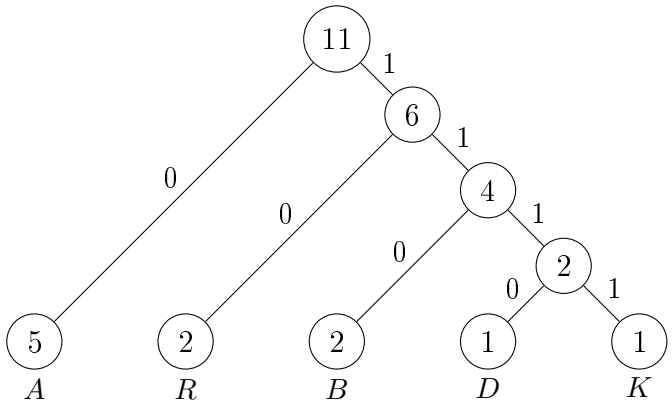
A fenti eljárást addig ismételjük, amíg már csak egy fánk marad. Ezt végül kivesszük a minimum-prioritásos sorból: ez a Huffman-féle kódfa.

$$\langle \{R/\mathbf{2}\}, \{[B/2]\mathbf{4}[(D/1)2(K/1)]\}, (A/\mathbf{5})\rangle$$

$$\langle (A/\mathbf{5}), (\{R/2\}\mathbf{6}\{[B/2]4[(D/1)2(K/1)]\})\rangle$$

$$[(A/5)\mathbf{11}(\{R/2\}6\{[B/2]4[(D/1)2(K/1)]\})]$$

A fent kapott kódfát az 1. ábrán is láthatjuk.



1. ábra. Az *ABRAKADABRA* szövegnek az alfabetikus konvencióval adódó Huffman-féle kódfája

Tekintsünk az 1. ábrán látható kódfában egy tetszőleges egyszerű, azaz körmentes utat, amely a fa gyökerétől lefelé valamelyik leveléig halad! Az út éleit címkéző biteket összeolvasva adódik a levelet címkéző karakter Huffman-kódja. Így a karaktarekre a következő kódtáblázatot kapjuk.

karakter	kód
A	0
B	110
D	1110
K	1111
R	10

A fentiek alapján az ABRAKADABRA szöveg Huffman kódja 23 bit, ami lényegesen rövidebb, mint a fenti naiv tömörítés esetén. A kódtáblázat bináris kódjait az ABRAKADABRA szöveg karakterei szerint sorban egymás után fűzve kapjuk a szöveg Huffman-kódját.

01101001111011100110100

A **kitömörítéshez** az előbbi Huffman-kód és a kódfa alapján a kezdő nulla rögtön az „A" címkéjű levélhez visz. Ezután sorban olvasva a maradékból a biteket, a 110 a B-hez visz, majd a 10 az R-hez, a 0 az A-hoz, a 1111 a K-hoz, a 0 az A-hoz, az 1110 a D-hez, a 0 az A-hoz, a 110 a B-hez, a 10 az R-hez, és végül a 0 az A-hoz. Így visszakaptuk az eredeti, tömörítetelen szöveget.

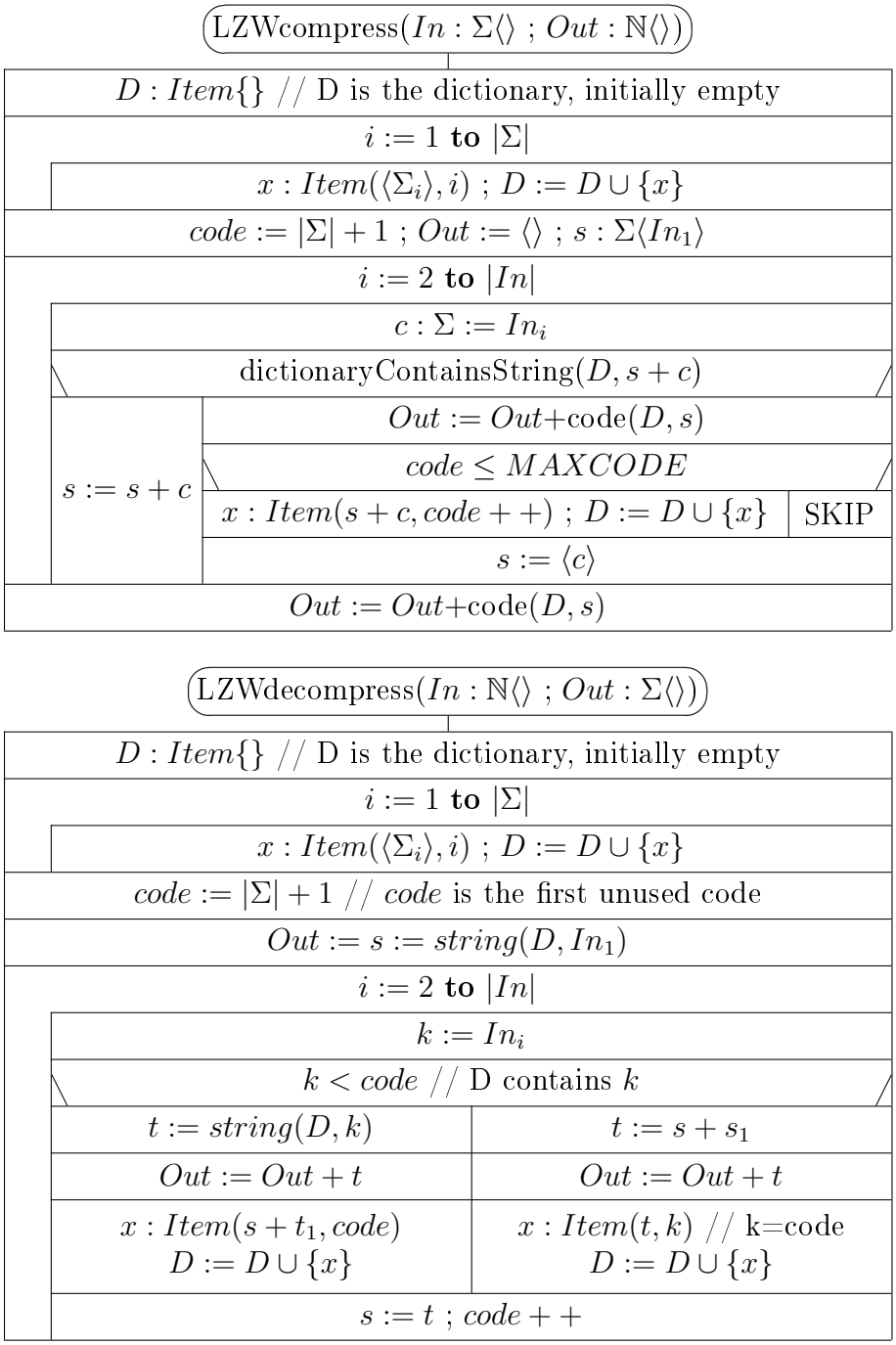
2.1. Feladat. *Próbáljuk ki, hogy ha a Huffman-kódolásban lévő indeterminizmusokat a fenti alfabetikus sorrendtől eltérően oldjuk fel, ugyanarra a tömörítendő szövegre mégis mindig ugyanolyan hosszú Huffman-kódot kapunk! (Ha például a minimum-prioritásos sorból azonos fa-gyakoriság-értekek esetén az alacsonyabb fát vesszük ki előbb – ezt az ad-hoc szabályt az alfabetikus konvenciónál erősebbnek véve –, akkor a fenti példában a kódfát felépítő ciklus második iterációjában a* $[B/\mathbf{2}]$ *és az* $\{R/\mathbf{2}\}$ *fát fogjuk összevonni.)*

2.3. Lempel–Ziv–Welch (LZW) módszer

Az input szöveget ismétlődő mintákra (sztringekre) bontja. Mindegyik mintát ugyanolyan hosszú bináris kóddal helyettesíti. Ezek a minták kódjai. A tömörített fájl a kódtáblázatot nem tartalmazza. Részletes magyarázat olvasható Ivanyos Gábor, Rónyai Lajos és Szabó Réka: *Algoritmusok* c. könyvében [4]. (Online elérhetősége az irodalomjegyzékünkben.)

Jelölések az absztrakt struktogramokhoz:
- Ha a kódok b bitesek, akkor $MAXCODE = 2^b - 1$ globális konstans a kódként használható legnagyobb számérték. Ha pl. $b = 12$, akkor $MAXCODE = 2^{12} - 1 = 4095$.
- A $\Sigma = \langle \sigma_1, \sigma_2, \ldots, \sigma_d \rangle$ sorozat tartalmazza az ábécé karaktereit.
- A tömörítésnél „*In*" a tömörítendő szöveg. „*Out*" a tömörítés eredménye: kódok sorozata. A kitömörítésnél fordítva.
- D a szótár, ami $(string, code)$ rendezett párok, azaz $Item$-ek halmaza. A szótárat a tömörített kód nem tartalmazza. Ehelyett a kitömörítés rekonstruálja az ábécé és a tömörített kód alapján.

Item
$+string : \Sigma\langle\rangle$ $+code : \mathbb{N}$
$+Item(s : \Sigma\langle\rangle \ ; \ k : \mathbb{N})\{string := s \ ; \ code := k\}$


LZWcompress($In : \Sigma\langle\rangle$; $Out : \mathbb{N}\langle\rangle$)
$D : Item\{\}$ // D is the dictionary, initially empty
$i := 1$ **to** $|\Sigma|$
$x : Item(\langle\Sigma_i\rangle, i)$; $D := D \cup \{x\}$
$code := |\Sigma| + 1$; $Out := \langle\rangle$; $s : \Sigma\langle In_1\rangle$
$i := 2$ **to** $|In|$
$c : \Sigma := In_i$
dictionaryContainsString($D, s + c$)
$s := s + c$
$Out := Out$+code(D, s)
$code \leq MAXCODE$
$x : Item(s + c, code + +)$; $D := D \cup \{x\}$
SKIP
$s := \langle c\rangle$
$Out := Out$+code(D, s)

LZWdecompress($In : \mathbb{N}\langle\rangle$; $Out : \Sigma\langle\rangle$)
$D : Item\{\}$ // D is the dictionary, initially empty
$i := 1$ **to** $|\Sigma|$
$x : Item(\langle\Sigma_i\rangle, i)$; $D := D \cup \{x\}$
$code := |\Sigma| + 1$ // $code$ is the first unused code
$Out := s := string(D, In_1)$
$i := 2$ **to** $|In|$
$k := In_i$
$k < code$ // D contains k
$t := string(D, k)$
$t := s + s_1$
$Out := Out + t$
$Out := Out + t$
$x : Item(s + t_1, code)$
$D := D \cup \{x\}$
$x : Item(t, k)$ // k=code
$D := D \cup \{x\}$
$s := t$; $code + +$