Algoritmusok és adatszerkezetek II. előadásjegyzet

Ásványi Tibor – asvanyi@inf.elte.hu

2026. július 29.

Tartalomjegyzék

1. Bevezetés, Ajánlott irodalom

Kedves Hallgatók!

A vizsgára való készülésben – ezen a nyomtatott jegyzeten kívül – elsősorban az előadásokon és a gyakorlatokon készített jegyzeteikre támaszkodhatnak.

(Ebben a jegyzetben a *-gal jelzett alfejezetek elsősorban a tananyag mélyebb megértését szolgálják, azaz nem képezik a vizsga részét.)

További ajánlott források:

Hivatkozások

[1] Ásványi Tibor, Algoritmusok és adatszerkezetek II. előadásjegyzet

[2] Ásványi Tibor, Algoritmusok II. gyakorló feladatok

[3] 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 (Fourth Edititon),
The MIT Press, 2022.

[4] Fekete István, Algoritmusok jegyzet

[5] Rónyai Lajos – Ivanyos Gábor – Szabó Réka, Algoritmusok,
TypoTEX Kiadó, 1999. ISBN 963 9132 16 0

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

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

[8] Carl Burch, Ásványi Tibor, B+ fák

[9] Ásványi Tibor, Algoritmusok és adatszerkezetek I. előadásjegyzet

[10] 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

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

Ezúton szeretnék köszönetet mondani *Umann Kristófnak* és *Varga Henrik Zoltánnak* az AVL fákról szóló fejezetben található szép, színvonalas szemléltető ábrák elkészítéséért, az ezekre szánt időért és szellemi ráfordításért!

A vizsgákon az elméleti kérdések egy-egy tétel bizonyos részleteire vonatkoznak. Lesznek még megoldandó feladatok, amik részben a tanult algoritmusok működésének szemléltetését, bemutatását, részben a szerzett ismeretek kreatív felhasználását kérik számon. Egy algoritmus, program, művelet bemutatásának mindig része a műveletigény elemzése.

Az előadások elsősorban a CLRS könyv [3] angol eredetijének jelzett kiadását követik, de pl. a piros-fekete fák helyett az AVL fákat tárgyaljuk, a jelöléseket az első féléves jegyzetnek megfelelően módosítva. (A CLRS könyv [3] érintett fejezeteiben a [korábbi kiadás alapján készült] magyar fordítás és a jelzett angol változat között leginkább csak néhány jelölésbeli különbséget találtunk.)

Az egyes struktogramokat általában nem dolgozzuk ki az értékadó utasítások szintjéig. Az olyan implementációs részleteket, mint a listák és egyéb adatszerkezetek, valamint az adattípusok műveleteinek pontos kódja, a dinamikusan allokált objektumok deallokálása stb. általában az Olvasóra hagyjuk, hiszen ezekkel az előző félévben foglalkoztunk. Használni fogunk olyan absztrakt fogalmakat, mint a véges halmazok, a sorozatok és a gráfok. Ezeket, ha valamely eljárás paraméterlistáján szerepelnek – mint strukturált adatokat – minden esetben cím szerint vesszük át. (A skalárok változatlanul érték vagy cím szerint adódnak át, ahol az érték szerinti paraméterátvétel az alapértelmezett.)

2. Jelölések

$$\left.\begin{array}{l} u\,..\,v = \{i \in \mathbb{Z} : u \le i \le v\} \\ [u\,..\,v) = \{i \in \mathbb{Z} : u \le i < v\} \\ (u\,..\,v) = \{i \in \mathbb{Z} : u < i < v\} \end{array}\right\} \text{ahol } u, v \in \mathbb{Z}.$$

A struktogramokban a „for” ciklusok (illetve a „for each” ciklusok) mintájára alkalmazni fogunk a ciklusfeltételek helyén pl. „$\forall x : P(x)$” alakú kifejezéseket, ami azt jelenti, hogy a ciklusmagot a $P(x)$ állítás igazsághalmazának minden x elemére kell végrehajtani, valamint „$\forall v \in V$” alakúakat, ami a

„$\forall v : v \in V$” rövidítése. Ehhez $P(x)$ igazsághalmazának, illetve V-nek végesnek, és hatékonyan felsorolhatónak kell lennie. Ez ideális esetben azt jelenti, hogy a felsorolás műveletigénye az igazsághalmaz, illetve V méretétől lineárisan függ.

Ha a ciklus fejében „$i := u$ to v” alakú kifejezést látunk, akkor a ciklusmag az $u..v$ egész intervallum i elemeire szigorúan monoton növekvő sorrendben hajtódik végre.

Ha pedig a ciklus fejében „$i := u$ downto v” alakú kifejezést látunk, akkor a ciklusmag a $v..u$ egész intervallum i elemeire hajtódik végre, de szigorúan monoton csökkenő sorrendben.

A fentiek szerint az egyszerűbb programrészletek helyén gyakran szerepelnek majd angol vagy magyar nyelvű utasítások, amelyek részletes átgondolását és esetleges kidolgozását a korábban tanultak alapján szintén az Olvasóra bízzuk. A struktogramokban az ilyen, formailag általában felszólító mondatok végéről a felkiáltójelet elhagyjuk (mivel az adott szövegkörnyezetben ez gyakran faktoriális függvényként lenne [félre]érthető).

A fentieken kívül ld. még az előző félévi nyomtatott jegyzetben [9] bevezetett jelöléseket!

3. Tematika

Minden tételhez: Hivatkozások: például a „[3] 8.2, 8.3, 8.4” jelentése: a [3] sorszámú szakirodalom adott (al)fejezetei.

1. Veszteségmentes adattömörítés. Naiv módszer. Huffman kód, kódfa, optimalitás. LZW (Lempel-Ziv-Welch) tömörítés. [1]

2. Általános fák, bináris láncolt reprezentáció, bejárások ([1]; [10] 4.7 bevezetése). Egyéb reprezentációk? (HF)

3. AVL fák és műveleteik: kiegyensúlyozási sémák, programok. Az AVL fa magassága [1, 4, 10].

4. B+ fák és műveleteik [8].

5. Elemi gráfalgoritmusok [1, 3]. Gráfábrázolások (representations of graphs).

A szélességi gráfkeresés (breadth-first search: BFS). A szélességi gráfkeresés futási ideje (the run-time analysis of BFS). A legrövidebb utak (shortest paths). A szélességi feszítőfa (breadth-first tree). HF: A szélességi gráfkeresés megvalósítása a klasszikus gráfábrázolások esetén; hatékonyság.

A mélységi gráfkeresés (depth-first search: DFS). Mélységi feszítő erdő (depth-first forest). A gráf csúcsainak szín- és időpont címkéi (colors and timestamps of vertices). Az élek osztályozása (classification of edges, its connections with the colors and timestamps of the vertexes). A mélységi gráfkeresés futási ideje (the run-time analysis of DFS). Topologikus rendezés (topological sort). HF: A mélységi gráfkeresés és a topologikus rendezés megvalósítása a klasszikus gráfábrázolások esetén; hatékonyság.

6. Minimális feszítőfák (Minimum Spanning Trees: MSTs) [1, 3]. Egy általános algoritmus (A general algorithm). Egy tétel a biztonságos élekről és a minimális feszítőfákról (A theorem on safe edges and MSTs). Prim és Kruskal algoritmusai (The algorithms of Kruskal and Prim). A futási idők elemzése (Their run-time analysis). HF: A Prim algoritmus implementációja a két fő gráfábrázolás és a szükséges prioritásos sor különböző megvalósításai esetén (The implementations of the algorithm of Prim with respect to the main graph and priority queue representations).

7. Legrövidebb utak egy forrásból (Single-Source Shortest Paths) [1, 3, 6, 7, 11]. A legrövidebb utak fája (Shortest-paths tree). Negatív körök (Negative cycles). Közelítés (Relaxation).

A sor-alapú (Queue-based) Bellman-Ford algoritmus. A menet (pass) fogalma. Futási idő elemzése. Helyessége. A legrövidebb út kinyomtatása.

Legrövidebb utak egy forrásból, körmentes irányított gráfokra. (DAG shortest paths.) Futási idő elemzése. Helyessége.

Dijkstra algoritmusa. Helyessége. Fontosabb implementációi a két fő gráfábrázolás és a szükséges prioritásos sor különböző megvalósításai esetén. A futási idők elemzése.

8. Dinamikus programozás [1, 3]. A legrövidebb utak minden csúcspárra (All-Pairs Shortest Paths). A megoldás ábrázolása a (D, Π) mátrix-párral. HF: Adott csúcspárra a legrövidebb út kinyomtatása.

A Floyd-Warshall algoritmus és a $(D^{(k)}, \Pi^{(k)})$ mátrix párok. A futási idő elemzése. Összehasonlítás a különböző legrövidebb utak egy forrásból algoritmusok $|G.V|$-szeri végrehajtásával.

Irányított gráf tranzitív lezártja (Transitive closure of a directed graph) és a $T^{(k)}$ mátrixok. Az algoritmus és futási ideje. Összehasonlítás a szélességi keresés $|G.V|$-szeri végrehajtásával.

9. Mintaillesztés (String Matching) [1, 3]. Egy egyszerű mintaillesztő algoritmus (a naive string-matching algorithm). A futási idő elemzése.

A Quick Search algoritmus. Inicializálása. A futási idő elemzése.

A Knuth-Morris-Pratt algoritmus. Inicializálása. A futási idő elemzése.

4. AVL fák (AVL trees)

Az előző félévben tanultunk a bináris keresőfákról, mint (a hasító táblák mellett) adathalmazok, szótárak reprezentációjára szolgáló adatszerkezetekről. Láttuk, hogy a tipikus szótárműveletek (mint egy kulcs – és a hozzá tartozó adatok – keresése, beszúrása és eltávolítása) átlagos esetben megfelelő hatékonyságúak, mert a futási idő legfeljebb a fa magasságával arányos, ennek várható értéke pedig $\Theta(\log n)$, ahol n a fa mérete, azaz csúcsainak száma, és a $\log n$ függvény nagyon lassan nő.

A legrosszabb esetben azonban a bináris keresőfa (BST) magassága $n-1$, így a szótárműveletek sem elég hatékonyak. Ezen segít a kiegyensúlyozott keresőfák (balanced search trees) alkalmazása, mert ezeknél a fa magassága minden esetben $\Theta(\log n)$, ami – megfelelő implementáció mellett – garantálja a szótárműveletek megfelelő hatékonyságát.

A különböző fajta kiegyensúlyozott bináris keresőfák (balanced BSTs) klasszikus típusai a piros-fekete [3] és az AVL fák. Ez utóbbiakat tárgyaljuk itt. Amennyiben az alkalmazásunkban lényegesen több a szótárban való keresés, mint az azt módosító művelet, az AVL fák alkalmazása előnyösebb, mert a piros-fekete fáknál az AVL fák erősebben kiegyensúlyozottak. Az ideális a majdnem teljes bináris keresőfák alkalmazása lehetne (ui. ezek magassága minimális), de ezek módosítására nem ismerünk hatékony megoldásokat. Az AVL rövidítés mögött *Georgy Adelson-Velsky* és *Evgenii Landis* szovjet matematikusok nevét fedezhetjük fel, akik 1963-ban dolgozták ki koncepciójukat.

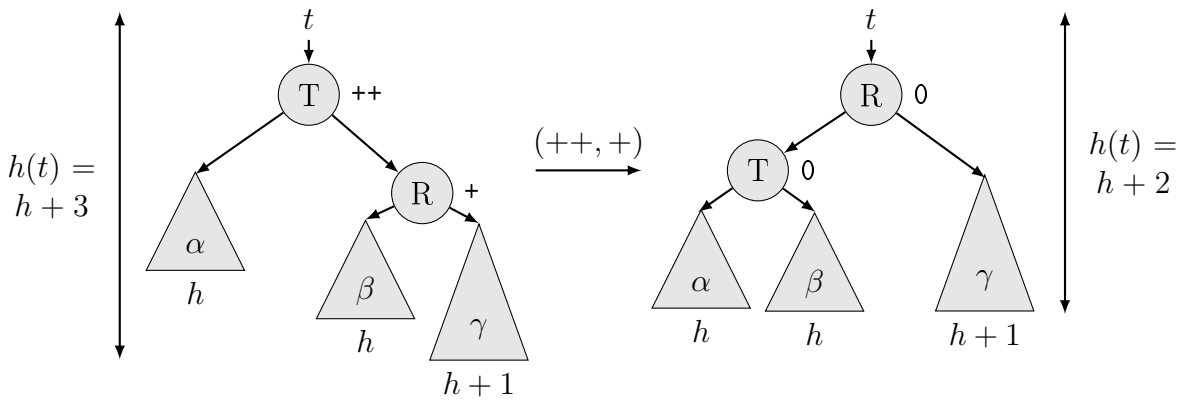
Az AVL fa módosításával járó műveletek, mint pl. a beszúrás és a törlés, elronthatják a kiegyensúlyozottságot, amit – kiegyensúlyozó forgatások (*balancing rotations*) segítségével – még a művelet befejezése előtt helyre kell állítani.

A témával ismerősek kedvéért elöljáróban megjegyezzük, hogy az AVL fák kiegyensúlyozásának szabályai megtalálhatók az 1-6. ábrákon.

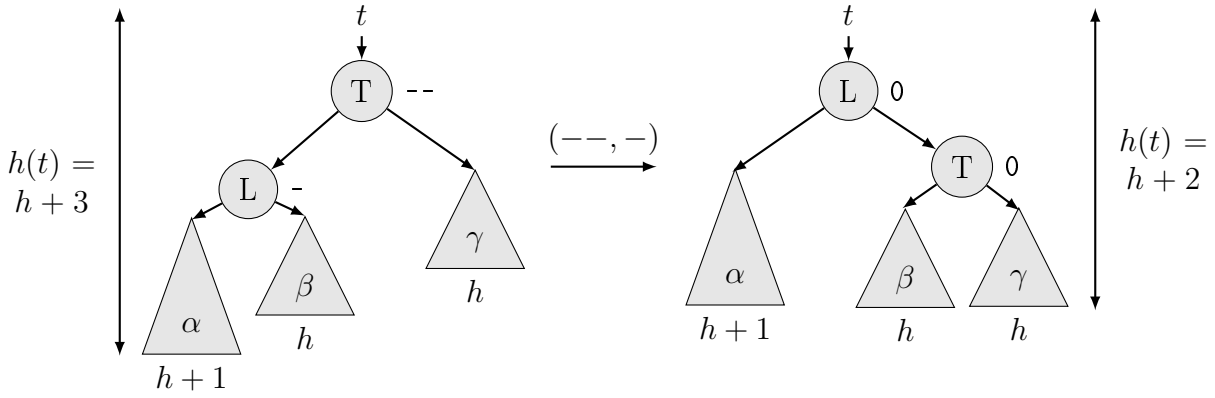
Ebben a fejezetben a továbbiakban részletesen tárgyaljuk az AVL fa fogalmát, tulajdonságait, egy különösen hatékony megvalósítását lehetővé tevő reprezentációját, és ennek megfelelően a műveleteit.

Az *AVL fák* magasság szerint kiegyensúlyozott bináris keresőfák (height-balanced BSTs). Egy bináris fa *magasság szerint kiegyensúlyozott*, ha minden csúcsa kiegyensúlyozott. (Mostantól *kiegyensúlyozott* alatt *magasság szerint kiegyensúlyozottat* értünk.) Egy bináris fa egy $(*p)$ csúcsa *kiegyensúlyozott* (balanced node), ha a csúcs $p \to b$ egyensúlyára $|p \to b| \leq 1$. A $(*p)$ csúcs egyensúlya [the $(*p)$ node's balance] definíció szerint:

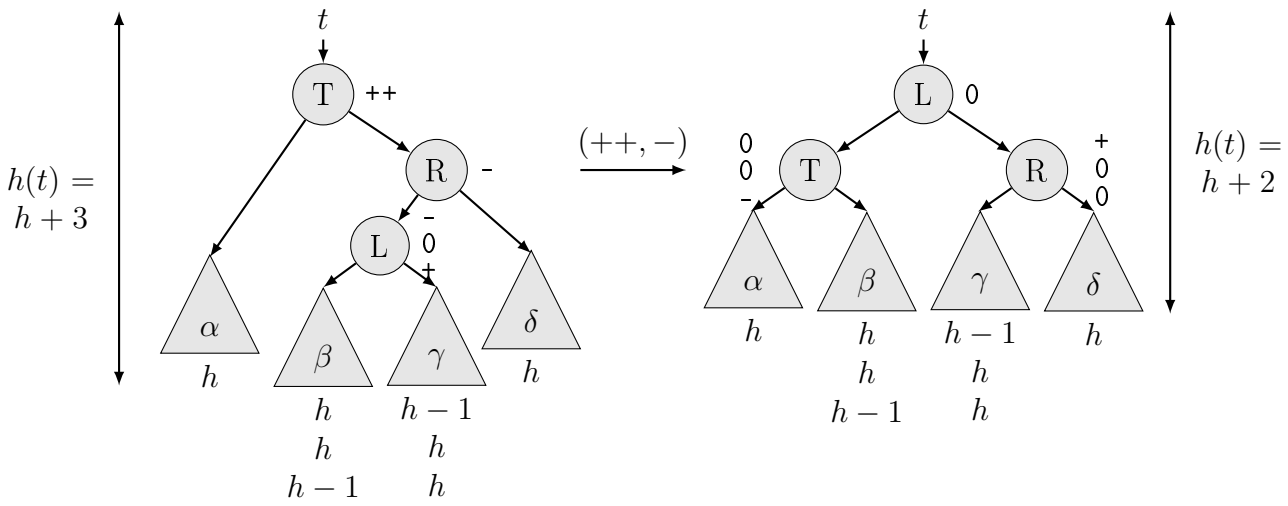
$$p \to b = h(p \to right) - h(p \to left)$$



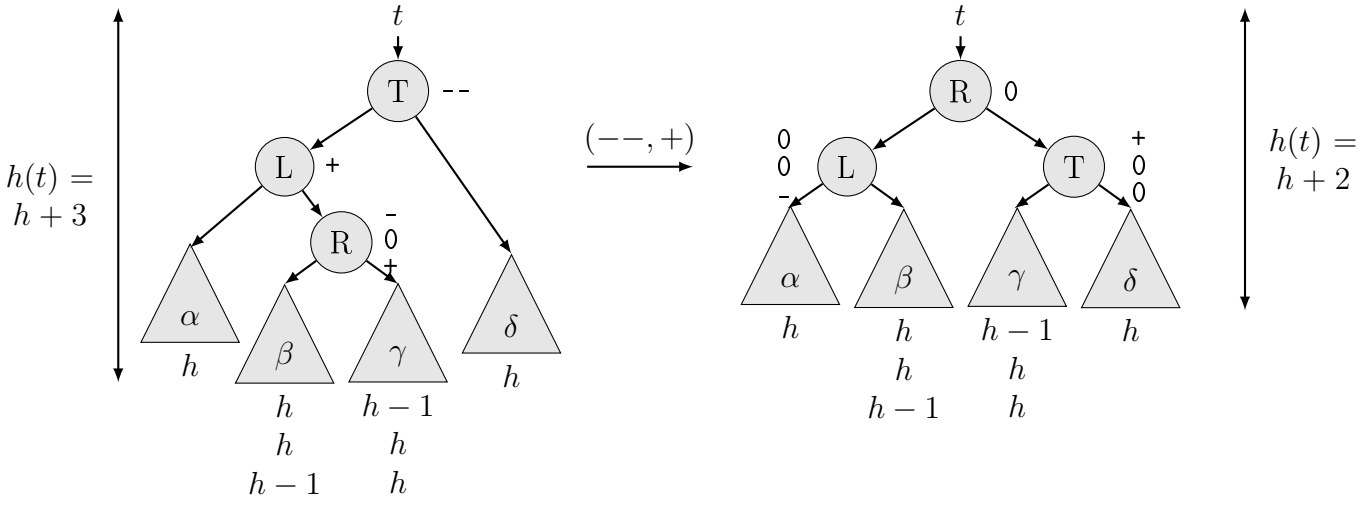
1. ábra. $(++,+)$ forgatás. A kiegyensúlyozatlan (T) csúcsnak és magasabb részfája (R) gyökércsúcsának egyensúlya nem ellentétes előjelű. Így a magasabb részfájához tartozó gyermeke, (R), a kiegyensúlyozó forgatás (*balancing rotation*) után az új gyökércsúcs. A többiek helyét a BST tulajdonság már meghatározza.



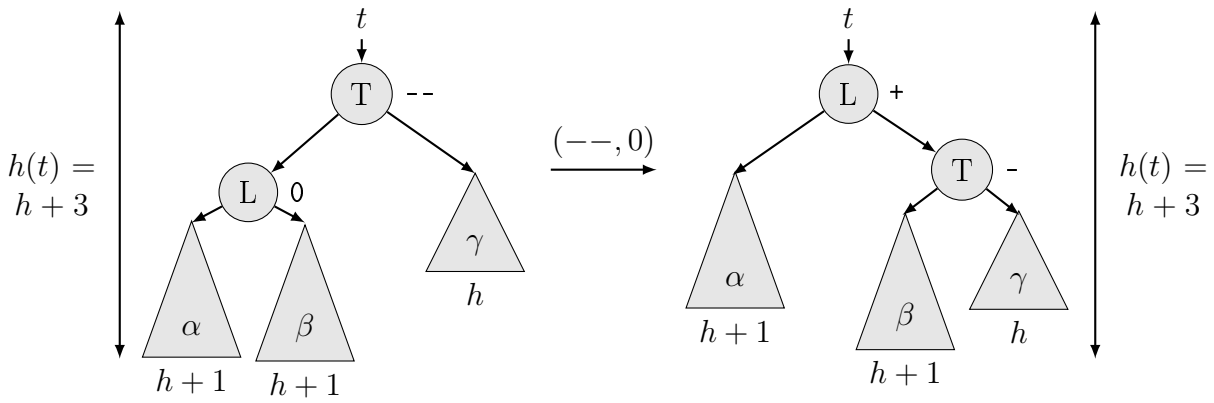
2. ábra. $(--,-)$ forgatás. A kiegyensúlyozatlan (T) csúcsnak és magasabb részfája (L) gyökércsúcsának egyensúlya nem ellentétes előjelű. Így a magasabb részfájához tartozó gyermeke, (L), a kiegyensúlyozó forgatás (*balancing rotation*) után az új gyökércsúcs. A többiek helyét a BST tulajdonság már meghatározza.



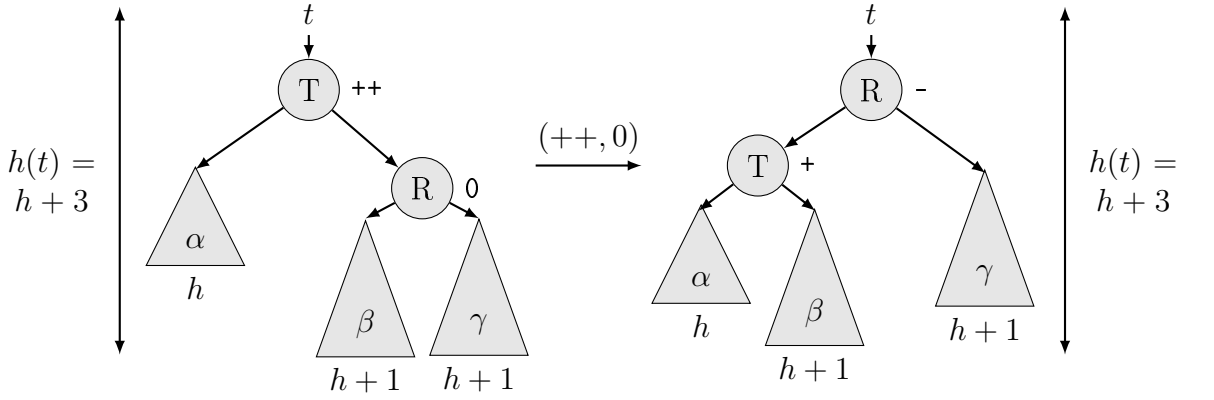
3. ábra. $(++,-)$ forgatás. A kiegyensúlyozatlan (T) csúcsnak és magasabb részfája (R) gyökércsúcsának egyensúlya ellentétes előjelű. Így a magasabb részfájához tartozó unokája, (L), a kiegyensúlyozó forgatás után az új gyökércsúcs. A többiek helyét a BST tulajdonság már meghatározza.



4. ábra. $(--,+)$ forgatás. A kiegyensúlyozatlan (T) csúcsnak és magasabb részfája (L) gyökércsúcsának egyensúlya ellentétes előjelű. Így a magasabb részfájához tartozó unokája, (R), a kiegyensúlyozó forgatás után az új gyökércsúcs. A többiek helyét a BST tulajdonság már meghatározza.



5. ábra. $(--, 0)$ forgatás. A kiegyensúlyozatlan (T) csúcsnak és magasabb részfája (L) gyökércsúcsának egyensúlya nem ellentétes előjelű. Így a magasabb részfájához tartozó gyermeke, (L), a kiegyensúlyozó forgatás (*balancing rotation*) után az új gyökércsúcs. A többiek helyét a BST tulajdonság már meghatározza.



6. ábra. $(++, 0)$ forgatás. A kiegyensúlyozatlan (T) csúcsnak és magasabb részfája (R) gyökércsúcsának egyensúlya nem ellentétes előjelű. Így a magasabb részfájához tartozó gyermeke, (R), a kiegyensúlyozó forgatás (*balancing rotation*) után az új gyökércsúcs. A többiek helyét a BST tulajdonság már meghatározza.

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 $n = |t|$.

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 (rész)fákat üres zárójelpár jelöli. A könnyebb olvashatóság kedvéért [],

{} és esetleg ⟨⟩ zárójeleket is alkalmazunk, a levélcsúcsoknál pedig a két üres részfát nem jelöljük.

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.*

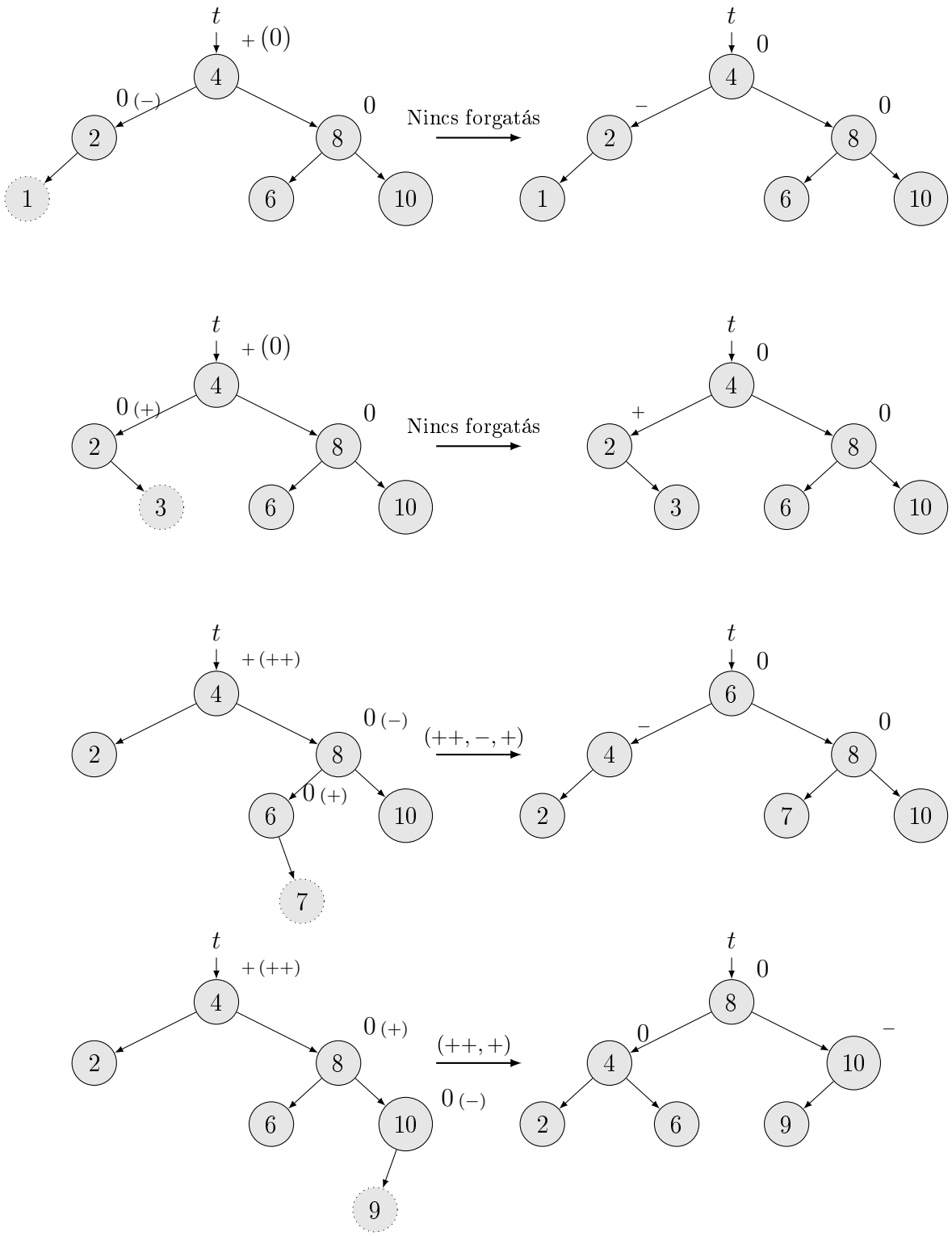
4.1. AVL fák: beszúrás

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. (Ld. a 8. ábra első két sorát!)

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. (Ld. a 8. ábra utolsó két sorát!)

A legutolsó esetben a bináris keresőfa könnyen kiegyensúlyozható, ha meggondoljuk, hogy az a következő sémára illeszkedik, amiben görög kisbetűk



7. ábra. Az AVL fa beszúrás néhány alapesete. A bal oldali ábrákon az új csúcsot halványabban jelöltük. A felette levő csúcsok átsúlyozásának eredményét zárójelbe tettük. A jobb oldalon látható a megfelelő kiegyensúlyozó forgatások (3. sor → 3. ábra 3. alesete; 4. sor → 1. ábra) végeredménye.

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=($\langle$9$\rangle$10−$\langle\rangle$).

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

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

A fenti példán ez a következőt jelenti (8. ábra utolsó sor):
{[2]4++[(6)8+($\langle$9$\rangle$10−$\langle\rangle$)]} → { [(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 kapott
{ [2]4++[($\langle\rangle$6+$\langle$7$\rangle$)8−(10)] } kiegyensúlyozatlan bináris keresőfa kiegyensúlyozásával pedig {[(2)4−()] 6∘ [(7)8∘(10)]} adódik (8. ábra, utolsó előtti sor), amire az

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

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 $\gamma = \langle 7\rangle$ 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 $h(\gamma) = h - 1$ ⇒ az eredmény fában T∘ és R+.
(2) L∘ esetén $h(\beta) = h$ és $h(\gamma) = h$ ⇒ az eredmény fában T∘ és R∘.
(3) L+ esetén $h(\beta) = h - 1$ és $h(\gamma) = h$ ⇒ az eredmény fában T− és R∘.

Mindhárom esetben $h([\alpha\ \mathrm{T}\ \beta]) = h + 1 = h([\gamma\ \mathrm{R}\ \delta])$, így L$\circ$.
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:

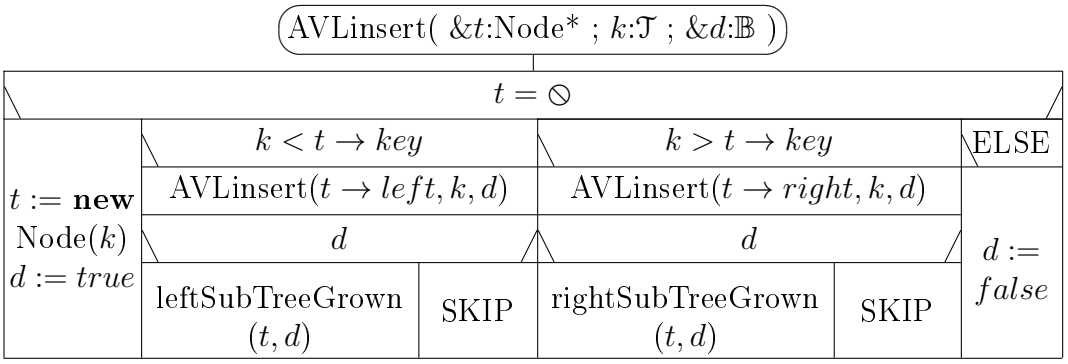
$$s_t = -\lfloor (s+1)/2 \rfloor \quad \text{és} \quad s_r = \lfloor (1-s)/2 \rfloor$$

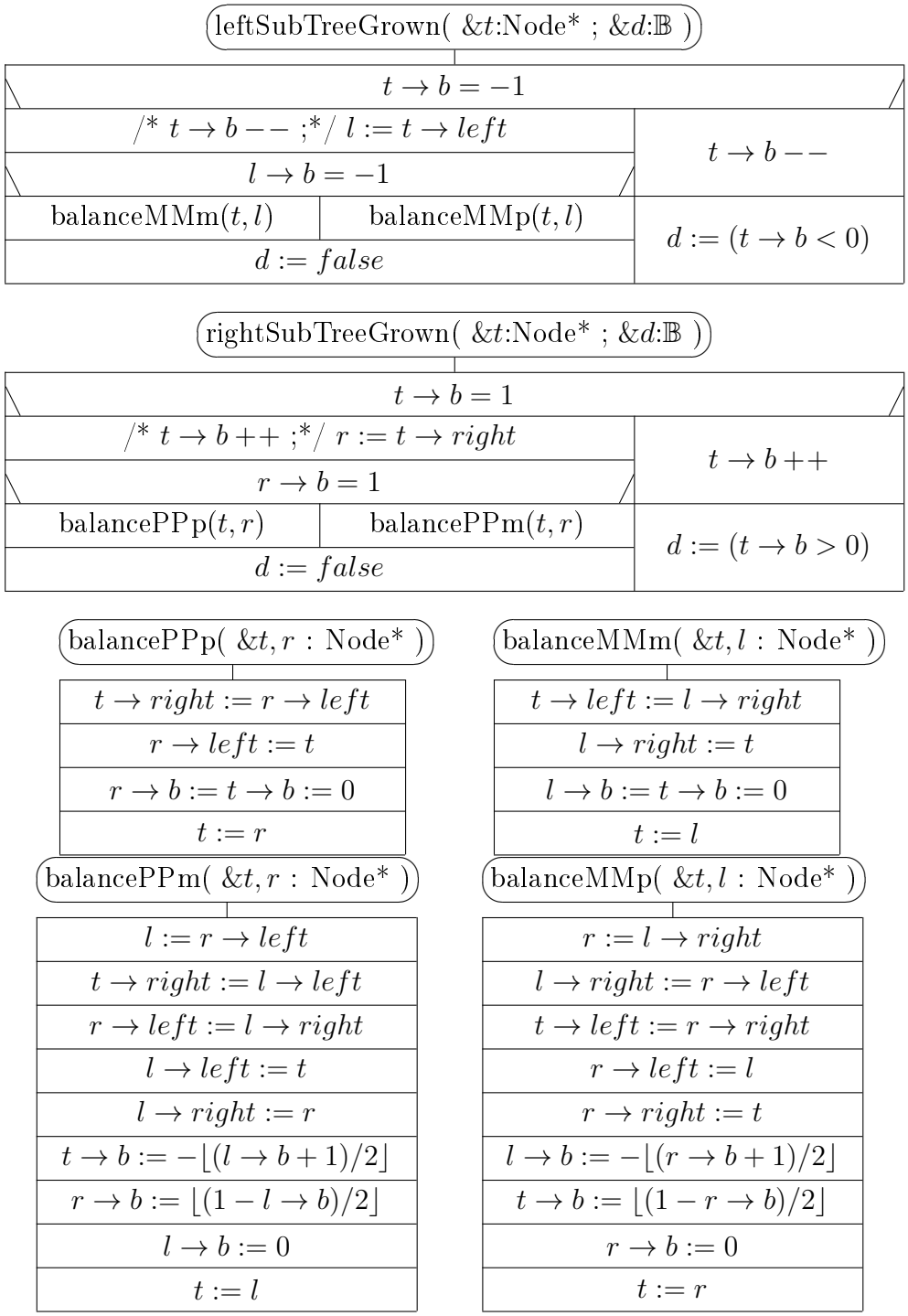
Az eddigi két kiegyensúlyozási séma mellett természetes módon adódnak ezek tükörképei, a forgatások előtt a T$--$ csúccsal a gyökérben. Ezek tehát a következők (2. és 4. ábra):

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

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 ($d = false$).





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

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

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

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

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.

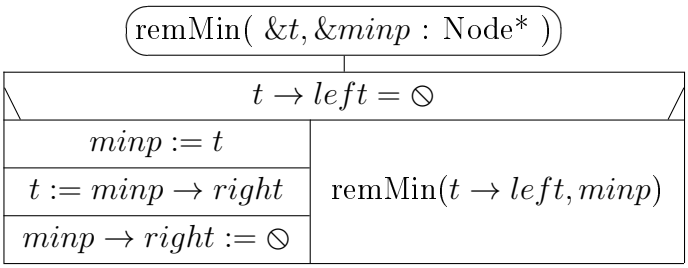
R+ esetén már láttuk, hogy $h(\alpha) = h = h(\beta)$ és $h(\gamma) = h + 1$. Ezért a kiegyensúlyozás után $h([(\alpha\ \mathrm{T}\ \beta)\ \mathrm{R}\ \gamma]) = h + 2$.

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.

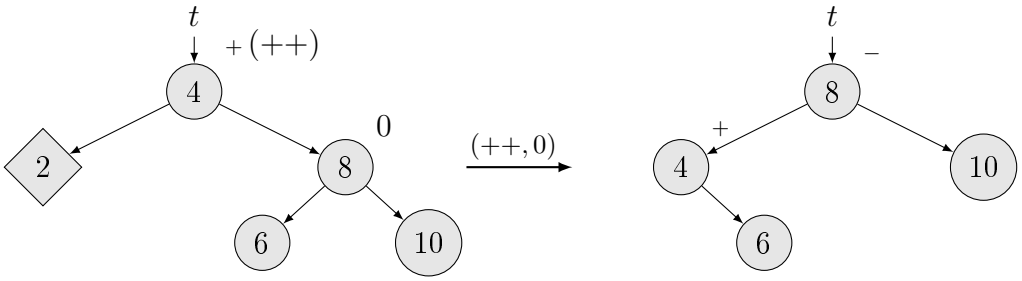
4.2. AVL fák: a remMin($t, minp$) eljárás

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.



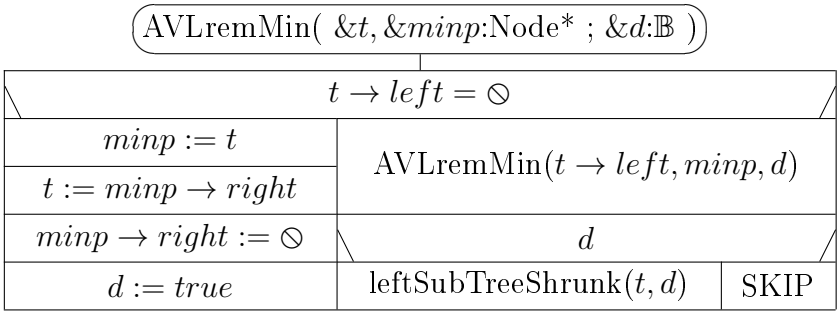
8. ábra. Az AVL fák remMin() műveletének új alapesete. A bal oldali ábrán az eltávolított csúcsot rombusszal jelöltük. A felette levő csúcs átsúlyozásának eredményét zárójelbe tettük. A jobb oldalon látható a megfelelő kiegyensúlyozó forgatás (6. ábra) végeredménye.

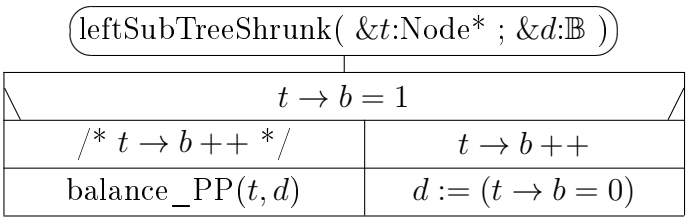
Szerencsére egy balra forgatás, a megfelelő egyensúly beállítások mellett most is megoldja a problémát (6. ábra):

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

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):





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

$r := t \to right$		
$r \to b = -1$	$r \to b = 0$	$r \to b = 1$
balancePPm(t, r)	balancePP0(t, r)	balancePPp(t, r)
	$d := false$	

balancePP0(&t, r : Node*)

$t \to right := r \to left$
$r \to left := t$
$t \to b := 1$
$r \to b := -1$
$t := r$

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.

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

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

4.3. AVL fák: törlés

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

del(&t:Node* ; k:$\mathcal{T}$)

$t \neq \oslash$			
$k < t \to key$	$k > t \to key$	$k = t \to key$	SKIP
del$(t \to left, k)$	del$(t \to right, k)$	delRoot(t)	

delRoot(&t:Node*)

$t \to left = \oslash$	$t \to right = \oslash$	$t \to left \neq \oslash \wedge t \to right \neq \oslash$
$p := t$	$p := t$	remMin$(t \to right, p)$
$t := p \to right$	$t := p \to left$	$p \to left := t \to left$
		$p \to right := t \to right$
delete p	**delete** p	**delete** t ; $t := p$

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.

AVLdel(&t:Node* ; k:$\mathcal{T}$; &d:$\mathbb{B}$)

<table>
<tr><th colspan="5">$t \neq \oslash$</th></tr>
<tr><th>$k < t \to key$</th><th colspan="2">$k > t \to key$</th><th>$k = t \to key$</th><th rowspan="4">$d := false$</th></tr>
<tr><td>AVLdel$(t \to left, k, d)$</td><td colspan="2">AVLdel$(t \to right, k, d)$</td><td rowspan="3">AVLdelRoot (t, d)</td></tr>
<tr><th>d</th><th colspan="2">d</th></tr>
<tr><td>leftSubTreeShrunk (t, d)</td><td>SKIP</td><td>rightSubTreeShrunk (t, d)</td><td>SKIP</td></tr>
</table>

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

<table>
<tr><th>$t \to left = \oslash$</th><th>$t \to right = \oslash$</th><th colspan="2">$t \to left \neq \oslash \wedge t \to right \neq \oslash$</th></tr>
<tr><td>$p := t$</td><td>$p := t$</td><td colspan="2" rowspan="2">rightSubTreeMinToRoot(t, d)</td></tr>
<tr><td>$t := p \to right$</td><td>$t := p \to left$</td></tr>
<tr><td>**delete** p</td><td>**delete** p</td><th colspan="2">d</th></tr>
<tr><td>$d := true$</td><td>$d := true$</td><td>rightSubTreeShrunk(t, d)</td><td>SKIP</td></tr>
</table>

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.

4.3. 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!*

4.4. 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 legnagyobb kulcsú csúcsával helyettesíti!*

4.4. 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 $\langle f_0, f_1, f_2, \dots \rangle$ 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 $(\varphi_{h-1}\ \mathrm{G}\ \varphi_{h-2})$ vagy $(\varphi_{h-2}\ \mathrm{G}\ \varphi_{h-1})$ alakú, ahol φ_{h-1} és φ_{h-2} $h-1$, illetve $h-2$ magasságú Fibonacci fák.

4.5. 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

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

5. Általános fák (General trees)

Az általános fák esetében, a bináris fákkal összehasonlítva, egy csúcsnak tetszőlegesen sok gyereke lehet. A fák most is irányítottak, és az utak a gyökércsúcstól a levelek felé vezetnek. 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 fáról* (*ordered tree*) 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 (*rooted trees*) beszélünk.[2]

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

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.

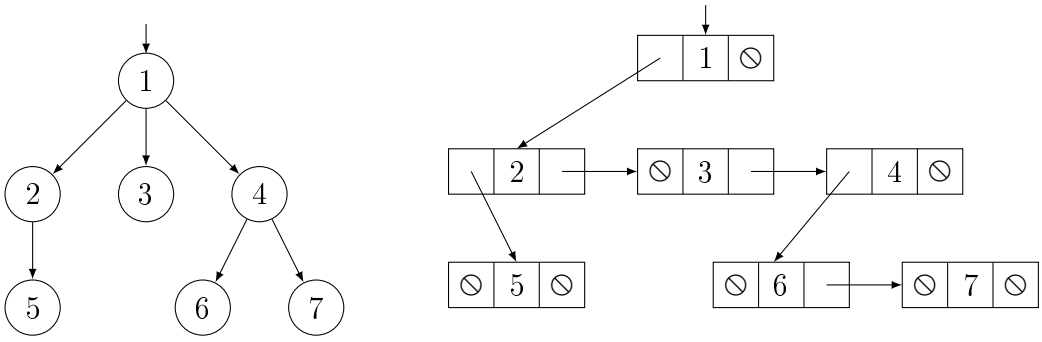
5.1. Feladat. *Próbáljunk megadni az általános fákra másfajta láncolt reprezentációkat is! Hasonlítsuk össze az általunk adott ábrázolásokat és a bináris láncolt reprezentációt, memóriaigény és rugalmasság szempontjából!*

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 (9. ábra).

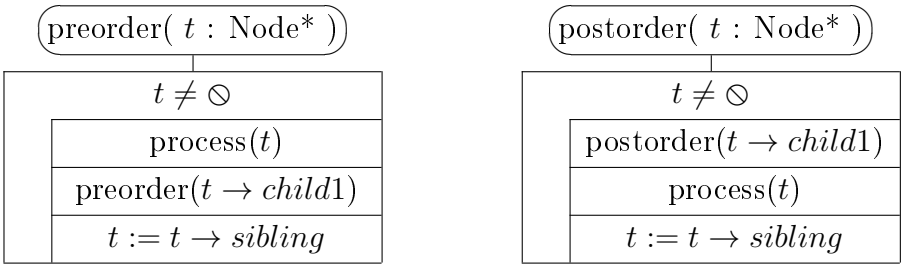
5.2. Feladat. *Írjunk programot, ami kiír egy binárisan láncolt fát a fenti zárójeles alakban! Írjunk olyat is, ami egy szövegfájlból visszaolvassa! Készítsük el a kiíró és a visszaolvasó programokat az általunk kidolgozott alternatív láncolt ábrázolásokra is! (A visszaolvasó programokat általában nehezebb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

Ha a fa csúcsai 1-től n-ig (vagy általában, az egész számok egy folytonos intervallumával) indexelhetők, mint a 9. ábrán látható példában, ahol $n = 7$; a fa *fordított reprezentációja* a lehető legtömörebb ábrázolás. Pl. egy $Parent/1 : \mathbb{N}[n]$ tömböt használhatunk, ahol $Parent[i]$ az i indexű csúcs szülőjének indexe. Ha i a gyökércsúcsra hivatkozik, akkor $Parent[i] = 0$. Egy másik fajta fordított reprezentáció szerint a gyökérnél $Parent[i] = i$. Az előbbi tradíció szerint a 9. ábrán látható fa fordított ábrázolása a $\langle 0, 1, 1, 1, 2, 4, 4 \rangle$ tömb, az utóbbi szokás szerint pedig az $\langle 1, 1, 1, 1, 2, 4, 4 \rangle$ tömb. A fordított faábrázolásokkal a gráfalgoritmusoknál fogunk találkozni.



9. 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]



5.3. 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!*

[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

5.4. 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!*

6. B+ fák és műveleteik

Ld. B+ fa.pdf a Canvas-ben!

7. Egyszerű gráfok és ábrázolásaik ([3] 20)

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

7.1. Gráfelméleti alapfogalmak

7.1. 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, azaz a továbbiakban gráf alatt *egyszerű gráfot* értünk.

Nincs ugyanis semmilyen eszközünk arra, hogy két (u, v) élet megkülönböztessünk (párhuzamos élek), ráadásul az ilyen kapcsolatok az (u, u) alakú, ún. hurokélekkel együtt az általunk tárgyalt algoritmusoknál feleslegesek.

7.2. Definíció. *A* $G = (V, E)$ *gráf* irányítatlan*, ha tetszőleges* $(u, v) \in E$ *élre, az* (u, v) *és a* (v, u) *éleket azonosnak tekintjük. (Megjegyzés: Ebből a megállapodásból persze* $(u, v) \in E \iff (v, u) \in E$ *következik.)*

7.3. Definíció. *A* $G = (V, E)$ *gráf* irányított*, ha tetszőleges* $(u, v), (v, u) \in E$ *élpárra* $(u, v) \neq (v, u)$ *(nem tekintjük őket egyenlőknek). 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.

7.4. 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* $i \in 1 \ldots n$*-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.*

7.5. 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.

7.6. 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.

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

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

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

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

7.10. 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.*

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

7.12. 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.*

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

7.14. 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.*

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

7.16. 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.*

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

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

7.19. 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.

7.2. Bevezetés a gráfábrázolásokhoz

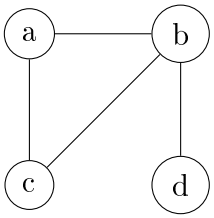
A gráfábrázolásoknál a $G = (V, E)$ gráfról általában föltesszük, hogy $V = \{v_1, \cdots, v_n\}$, ahol $n \in \mathbb{N}$, azaz hogy a gráf csúcsait egyértelműen azonosítják az $1 \ldots n$ sorszámok.

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.)

7.3. Grafikus ábrázolás

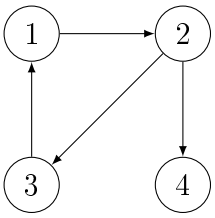
A gráfoknál megszokott módon a csúcsokat kis körök jelölik, az éleket irányított gráfoknál a körök közti nyilak, irányítatlan esetben a köröket összekötő vonalak reprezentálják. A csúcsok sorszámát (illetve az azt reprezentáló betűt) általában a körökbe írjuk.
A 10. ábrán egy egyszerű, irányítatlan gráfot láthatunk, grafikus és szöveges reprezentációban, a csúcsokat kisbetűkkel azonosítva.

A 11. 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.

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



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

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

7.4. Szöveges ábrázolás

Az irányítatatlan 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. a 10. 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 11. ábrát!)

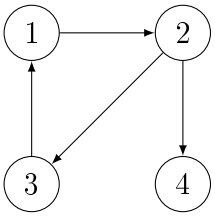
7.5. Szomszédossági mátrixos (adjacency matrix), más néven csúcsmátrixos reprezentáció

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 12. ábrán látható irányított gráfot például a mellette lévő bitmátrix reprezentálja.



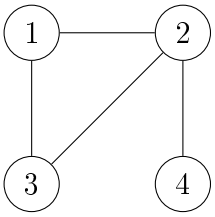
A	1	2	3	4
1	0	1	0	0
2	0	0	1	1
3	1	0	0	0
4	0	0	0	0

12. 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 $(v_i, v_j) \in E \iff (v_j, v_i) \in E$.

A 10. ábráról már ismerős irányítatlan gráf csúcsmátrixos ábrázolása a szokásos $a = 1, b = 2, c = 3, d = 4$ megfeleltetéssel a 13. 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

13. 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.

7.6. Szomszédossági listás (adjacency list) reprezentáció

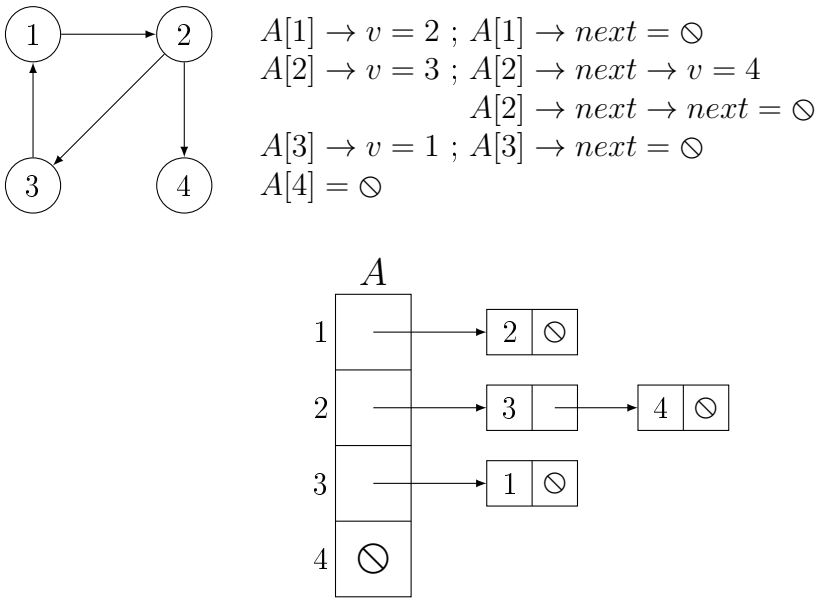
A szomszédossági listás ábrázolás hasonlít a szöveges reprezentációhoz. A $G = (V, E)$ gráfot ($V = \{v_1, \cdots, v_n\}$) az $A/1 : Edge^*[n]$ pointertömb

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 a 14. ábrán.



14. 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.

7.7. 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.

7.7.1. 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 $\Theta(n^2)$.

7.7.2. 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.

8. 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 konkatenáltjukat 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 élettartamuk 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 attributumhivatkozások esetén a szomszédossági listás gráfreprezentáció az alapértelmezett (ami azért adott esetben megváltoztatható), míg az E attributumhivatkozások esetében ez további megfontolás tárgya.

9. Elemi gráfalgoritmusok ([3] 21)

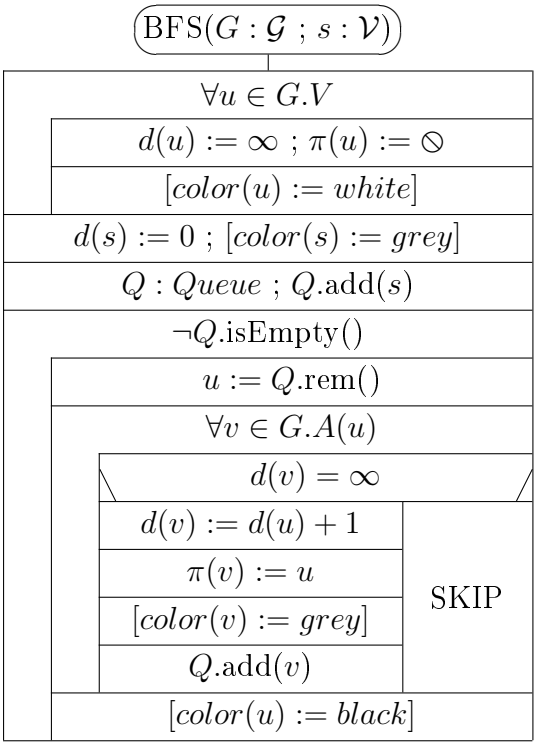
Elemi gráfalgoritmusok alatt élsúlyozatlan gráfokon értelmezett algoritmusokat értünk. Élsúlyozatlan gráfokban tetszőleges út hossza egyszerűen az út mentén érintett élek száma. Az *algoritmusok* témakörben szokásos fogalmak szerint az út tartalmazhat kört. Ha a gráfban tetszőleges u és v csúcsokra az (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 (a szélességi keresést és a mélységi bejárást), valamint ezek néhány alkalmazását tárgyaljuk.

9.1. 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 végrehajtva 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[8].

Az s start (más néven source, azaz forrás) csúcsra azonban tudjuk, hogy $d(s) = 0$ az optimális $s \rightsquigarrow s$ út hossza[9]. 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,[10] és bekerülnek a sorba.[11]

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

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

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

[10] [szürkére színezi őket]

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

$\pi(v) = u$ értékeket a szülőjüknek megfelelően,[12] és bekerülnek a sor végére[13]. Ha valamelyik v gyerekcsúcsra az (u, v) él feldolgozásakor már $d(v) \neq \infty$, akkor ez a csúcs már korábbról ismert[14], 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.[15]

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

9.1.1. 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-

[12] [szürkére színezi őket]

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

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

[15] [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.

9.2. 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* $d = d(v)$.

9.3. 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* $d = d(v)$.

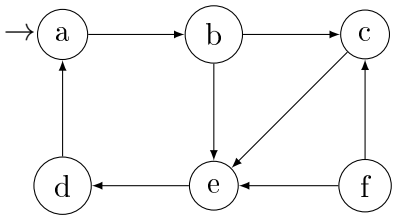
9.4. Feladat. *Tegyük fel, hogy a G gráfra már lefutott a szélességi BFS(G, s) algoritmus. Írja meg a printShortestPath(s, v) eljárást, ami kiírja az s-ből v-be vezető (a BFS által kiszámolt) legrövidebb utat, ha s-ből v elérhető! Különben a kiírás azt adja meg, hogy melyik csúcsból melyik nem érhető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

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

9.1.2. A szélességi keresés szemléltetése

A 15. 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,

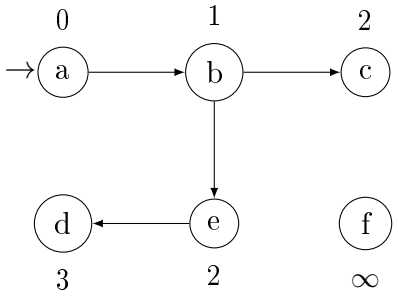


a → b.
b → c ; e.
c → e.
d → a.
e → d.
f → c ; e.

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

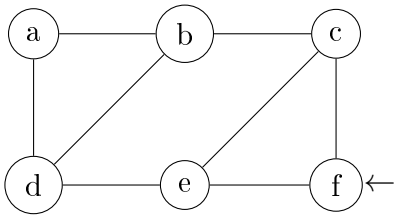
vizsgákon is elvárjuk, hogy tartsák be.) Alább pl. ez akkor érvényesül, amikor a „b” csúcs gyerekeit „c,e” sorrendben tesszük be a sorba.

ex-	changes of d and π						Q :
pand	a	b	c	d	e	f	Queue
$d\,\mathcal{V}$	0 ⊘	∞ ⊘	∞ ⊘	∞ ⊘	∞ ⊘	∞ ⊘	⟨ a ⟩
0 a		1 a					⟨ b ⟩
1 b			2 b		2 b		⟨ c, e ⟩
2 c							⟨ e ⟩
2 e				3 e			⟨ d ⟩
3 d							⟨ ⟩



16. ábra. A 15. ábráról ismerős gráf szélességi fája, ha s = a.

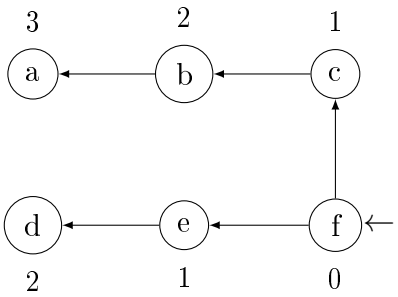
Most pedig a 17. ábrán látható gráfra szemléltetjük a BFS működését, az alábbi táblázat segítségével, ahol most „f” a start csúcs.



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

17. ábra. Ugyanaz az irányítatlan gráf grafikus (balra) és szöveges (jobbra) ábrázolással (s = f).

ex-	changes of d and π						Q :
pand	a	b	c	d	e	f	Queue
$d\,\mathcal{V}$	∞ ⊘	∞ ⊘	∞ ⊘	∞ ⊘	∞ ⊘	0 ⊘	⟨ f ⟩
0 f			1 f		1 f		⟨ c, e ⟩
1 c		2 c					⟨ e, b ⟩
1 e				2 e			⟨ b, d ⟩
2 b	3 b						⟨ d, a ⟩
2 d							⟨ a ⟩
3 a							⟨ ⟩



18. ábra. A 17. ábráról ismerős gráf szélességi fája, ha s = f.

9.1.3. 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 $mT(n,m) \in \Theta(n)$.

9.1.4. 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.

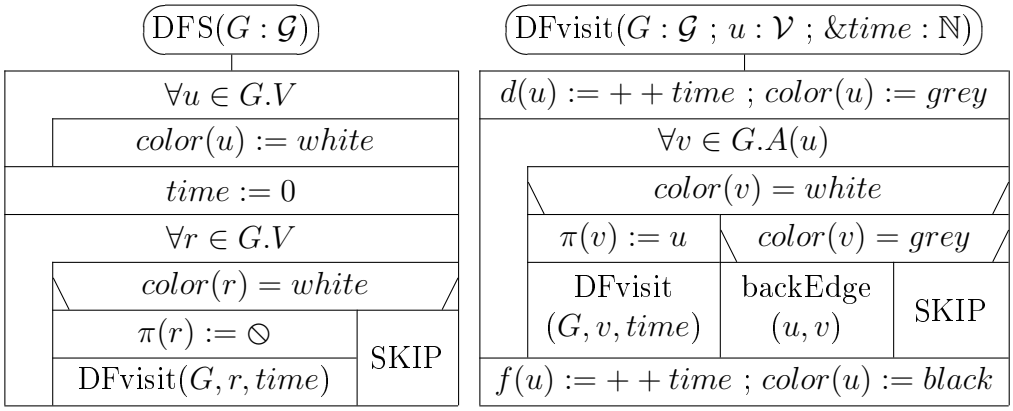
9.5. Feladat. *Írja meg a BFS($A/1 : Edge*[n]* ; $s : 1\,..\,n$; $d/1, \pi/1 : \mathbb{N}[n]$) eljárást, ami a szélességi keresés implementációja szomszédossági listás (10.4) gráfábrázolás esetére. Ügyeljen arra, hogy az absztrakt algoritmus hatékonyságának aszimptotikus nagyságrendje megmaradjon!*

9.6. 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 (10.3) gráfábrázolás esetére. Tartható-e az absztrakt algoritmus hatékonyságának aszimptotikus nagyságrendje? Ha nem, hogyan változik?*

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

Mélységi bejárásnak (Depth-first Traversal) vagy mélységi gráfbejárásnak is nevezzük, 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)

9.2.1. 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 $\iff u = \pi(v)$

9.2.2. Az élek osztályozása (Classification of edges)

9.7. 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.*

9.8. 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* $\wedge\ d(u) < d(v)$.

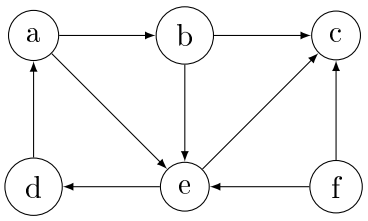
(u, v) kereszt-él (cross edge) $\iff$ *a v csúcs már fekete* $\wedge\ d(u) > d(v)$.

9.9. 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 9.7. definíciót úgy, hogy a 9.8. tétel megfogalmazásán ne kelljen módosítani?*

9.2.3. A mélységi bejárás szemléltetése

A 19. 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.

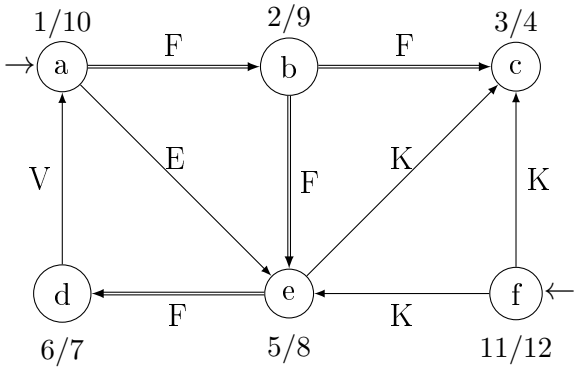


a → b ; e.
b → c ; e.
d → a.
e → c ; d.
f → c ; e.

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

A 20. 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.]



20. ábra. A 19. 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 9.8. 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 9.8. 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 9.8. tételt!) [Vegyük észre, hogy ha vissza-élt találunk, egyúttal irányított kört is találtunk a gráfban! (Ld. a 9.7. 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 9.8. 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.

9.2.4. 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 $MT(n), mT(n) \in \Theta(n+m)$.

9.2.5. A DAG tulajdonság eldöntése

9.10. 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 algoritmus DAG-ot vár a bemenetén, így az input ellenőrzése is szükséges lehet. (Ld. pl. a *topologikus rendezést* (9.2.6) 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 [3]. [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.]

9.11. 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.

9.12. 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.)

9.13. 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!*

9.14. 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!*

9.15. 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?*

9.2.6. Topologikus rendezés

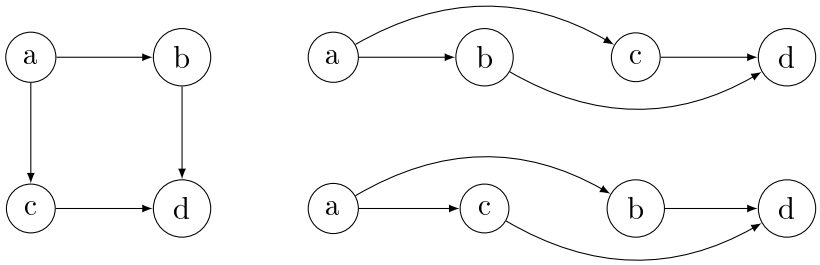
9.16. Definíció. *Irányított gráf topologikus rendezése alatt a gráf csúcsainak olyan sorba rendezését értjük, amelyben minden él egy-egy később jövő csúcsba (szemléletesen: balról jobbra) mutat.*

Ugyanannak a gráfnak lehet egynél több topologikus rendezése, mint a 21. ábra mutatja.

9.17. 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).



21. 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:

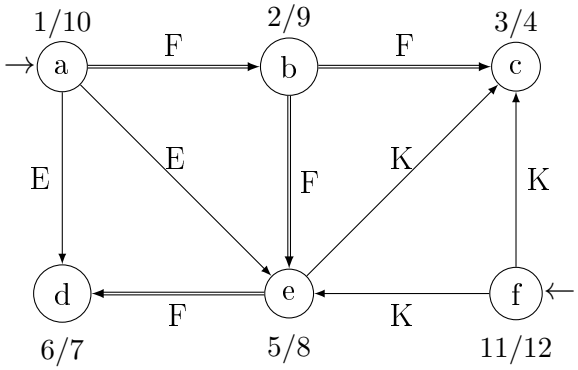
Először létrehozunk egy üres vermet, majd végrehajtjuk a gráf mélységi bejárását úgy, hogy valahányszor befejezünk egy csúcsot, a verem tetejére tesszük. Végül a verem tartalmát kiolvasva megkapjuk a gráf csúcsainak topologikus rendezését.

Vegyük észre, hogy ez az algoritmus képes ellenőrizni 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 22. 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.

9.18. 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?*



22. ábra. A DAG topologikus rendezésében 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!

9.19. Feladat. *Vegyük észre, hogy a 9.17. 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?*

10. Élsúlyozott gráfok és ábrázolásaik

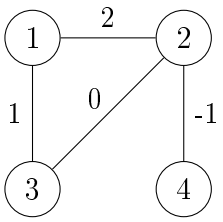
10.1. Definíció. Élsúlyozott gráf *alatt egy* $G = (V, E)$ *gráfot értünk a* $w : E \to \mathbb{R}$ *súlyfüggvénnyel, ahol* V *a csúcsok (vertices) tetszőleges, véges halmaza,* $E \subseteq V \times V \setminus \{(u, u) : u \in V\}$ *az élek (edges) halmaza.*

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

10.2. 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.*

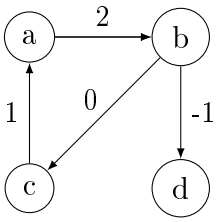
10.1. Grafikus ábrázolás

Az éleket súlyukkal címkézzük.



1 – 2, 2 ; 3, 1.
2 – 3, 0 ; 4, -1.

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



a → b, 2.
b → c, 0 ; d, -1.
c → a, 1.

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

10.2. Szöveges ábrázolás

Az irányítatlan 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. a 23. 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 24. ábrát!)

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

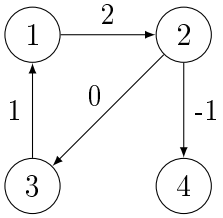
A szomszédossági mátrixos, vagy más néven csúcsmátrixos ábrázolásnál a $G = (V, E)$ gráfot a $w : E \to \mathbb{R}$ súlyfüggvénnyel ($V = \{v_1, \cdots , v_n\}$) egy $A/1 : \mathbb{R}_\infty[n, n]$ mátrix reprezentálja, ahol $n = |V|$ a csúcsok száma, $1\,..\,n$ a csúcsok sorszámai, és tetszőleges $i, j \in 1\,..\,n$ 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 25. ábrán látható irányított gráfot például a mellette lévő szomszédossági mátrix reprezentálja.



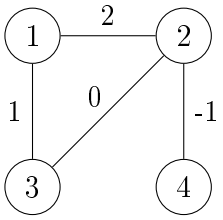
A	1	2	3	4
1	0	2	∞	∞
2	∞	0	0	-1
3	1	∞	0	∞
4	∞	∞	∞	0

25. 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 $(v_j, v_i) = (v_i, v_j) \in E$.

A 23. ábráról már ismerős irányítatlan gráf csúcsmátrixos ábrázolása a 26. ábrán látható.



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

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

10.4. 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.

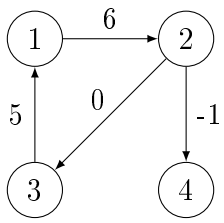
$Edge$
$+v : \mathbb{N}$
$+w : \mathbb{R}$
$+next : Edge*$

A $next$ és a v attribútumok 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 a 27. ábrán.

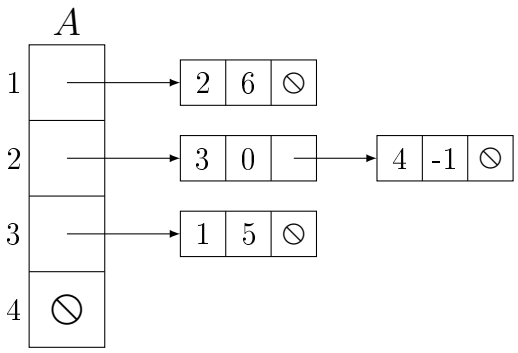
10.5. Élsúlyozott gráfábrázolások tárigénye

10.5.1. 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 $\Theta(n^2)$.



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



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

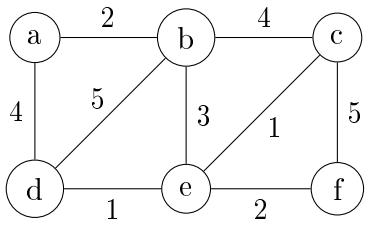
10.5.2. Szomszédossági listák

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

10.6. 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

11. Minimális feszítőfák ([3] 21)



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

28. 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 28. 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.)

11.1. 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).*

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

11.3. 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:*

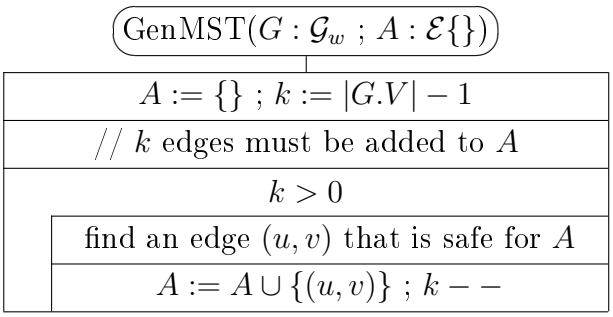
$$w(G) = \sum_{e \in E} w(e)$$

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

11.1. 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.



11.5. 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.*

11.6. 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.

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

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

11.9. 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 $w(u, v) \leq w(p, q)$.*

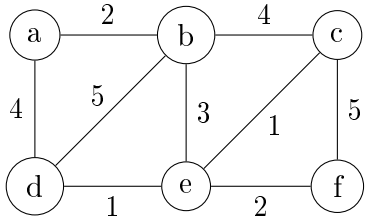
11.10. Definíció. *A $G = (V, E)$ gráfban az $A \subseteq E$ élhalmazt* elkerüli *az $(S, V \setminus S)$* vágás, *ha az A egyetlen éle sem keresztezi a vágást.*

11.11. 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.

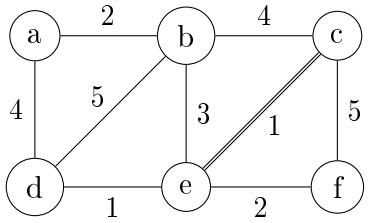
Bizonyítás. $(u,v) \notin A$, ui. (u,v) keresztezi az A-t elkerülő vágást.
Legyen $T = (V, T_E)$ olyan MST, amire $A \subseteq T_E$
a, $(u,v) \in T_E \Rightarrow$ készen vagyunk.
b, $(u,v) \notin T_E$ esetén megjegyezzük, hogy T feszítőfa, tehát T-ben el lehet jutni u-ból v-be, és a T-ben az u-ból a 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. Jelöljük a gráfban dupla vonallal az A élhalmaz elemeit!

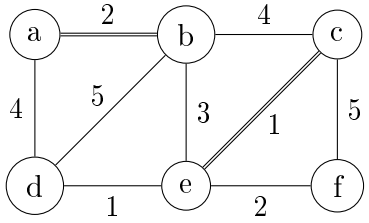
Az alábbi példában minden lépésben választunk egy A-t elkerülő vágást, majd ebben egy könnyű élt, amit hozzáveszünk A-hoz. A hozzávétel eredménye a következő lépés kiinduló pontja.



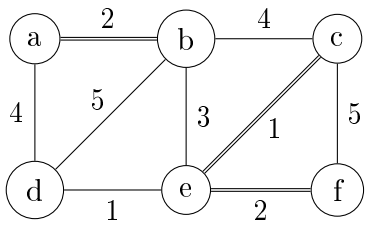
Kezdetben $A = \{\}$, így bármelyik vágás jó. Válasszuk pl. az $(\{a,b,c\}, \{d,e,f\})$ vágást! Ebben (c,e) a könnyű, tehát biztonságos él. Vegyük hozzá A-hoz!



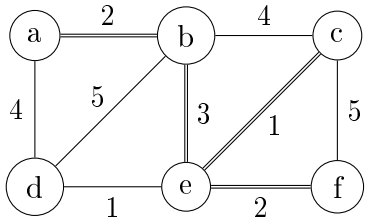
Most $A = \{(c,e)\}$. Ezt elkerüli pl. az $(\{a,f\}, \{b,c,d,e\})$ vágás, amiben (a,b) és (e,f) a könnyű, tehát biztonságos élek. Válasszuk pl. (a,b)-t!



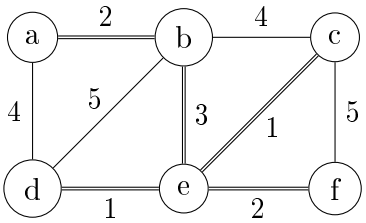
Most $A = \{(a,b),(c,e)\}$. Ezt elkerüli pl. az $(\{a,b,c,d,e\}, \{f\})$ vágás, amiben (e,f) a könnyű, tehát biztonságos él.



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-Jarník** 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 11.11. tétel szerint viszont a mohóság most kifizetődik, azaz mindkét algoritmus minimális feszítőfát számol.

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

11.2. 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.

11.13. 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.

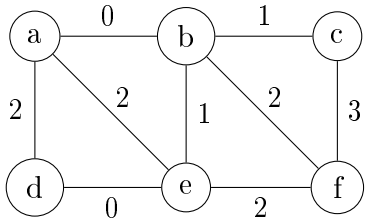
Kérdés még, hogy a fenti invariáns miért igaz. Kezdetben a (V, A) úgy feszítő erdő, hogy $A = \{\}$, tehát A részhalmaza a G bármelyik minimális feszítőfája élhalmazának, azaz az invariáns igaz.

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

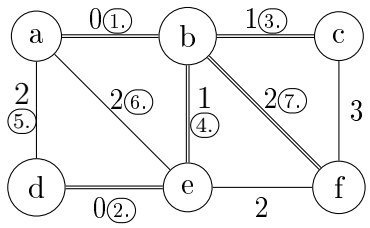
korábban feldolgoztuk, így ezek vagy benne vannak A-ban, vagy nincsenek benne A-ban, de a (V, A) feszítő erdő egyik fájának két csúcsát kötik össze, ezért eldobtuk őket]. Teljesülnek tehát a 11.11. tétel feltételei. Ennélfogva e biztonságosan hozzávehető A-hoz, és hozzá is vesszük. A fentiek szerint 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.

29. ábra. Összefüggő, élsúlyozott, irányítatlan gráf

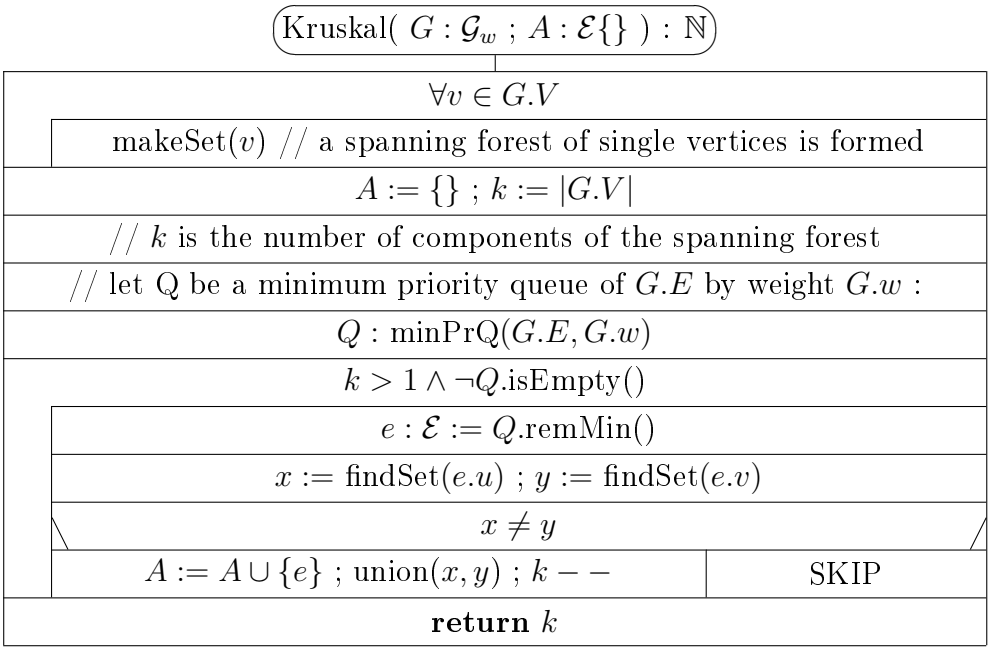
11.14. Példa. *Szemléltessük Kruskal algoritmusát a 29. ábrán látható gráfon!*



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

30. ábra. A 29. 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 hozzá 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 11.2.1. 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.

11.15. Feladat. *Hogyan tudná kifinomultabban értelmezni azt az esetet, ha a Kruskal algoritmus $k > 1$ értékkel tér vissza? Mit jelent a k értéke általában? Tudna-e értelmet tulajdonítani a Kruskal algoritmus által kiszámolt A élhalmaznak, ha végül $k > 1$ értéket ad vissza?*

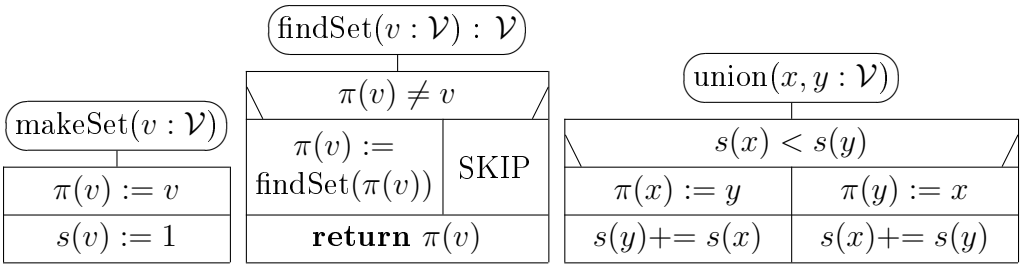
11.16. Feladat. *Implementálja a Kruskal algoritmust az elméleti $O(m*\lg n)$ maximális műveletigény megtartásával!*

11.2.1. 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 végrehajtása után az eredetileg nagyobb komponens reprezentáns eleme lesz az egyesített halmazt reprezentáló csúcs.

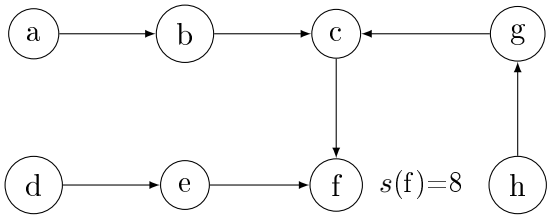
A csúcshalmazokat a reprezentáns elemük felé irányított fákkal ábrázoljuk. Ezért mindegyik csúcsnak van egy π címkéje, aminek értéke az ő szülője, kivéve az irányított fák gyökércsúcsait; tetszőleges r gyökércsúcsra viszont $\pi(r) = r$, $s(r)$ pedig az r-hez tartozó fa mérete. (A fákban a gyökércsúcstól 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.



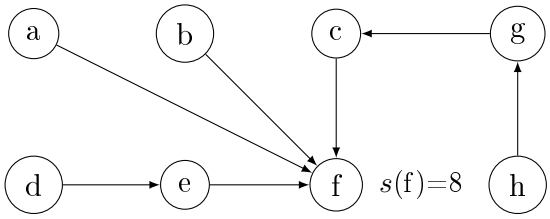
Mint látható, a makeSet(v) művelet a gráf v csúcsából egyelemű irányított fát képez. (Ezt a Kruskal algoritmus kezdetén a gráf mindegyik csúcsára meghívjuk.)

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.
Pl. tegyük fel, hogy adott a 31. 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" [3] könyvben részletes elemzés olvasható. Az alábbi hatékonyságelemzésnél eltekintünk az *útrövidítésből* eredő *hatékonyságnövekedéstől*. Célunkat az így adódó pontatlanabb felső becslésekkel is elérjük.

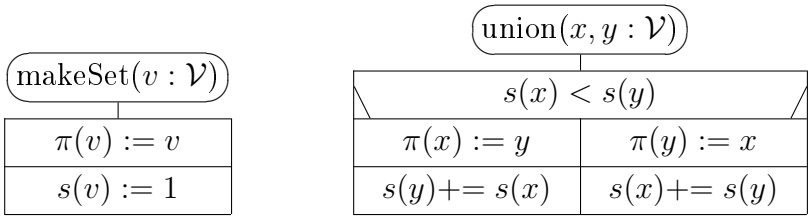


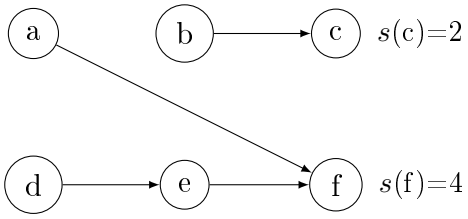
31. ábra. Gyökere (f) felé irányított fa. Pl. π(d)=e, π(e)=f, π(f)=f. Fent a findSet(a) függvényhívás egy lehetséges inputja, lent ugyanennek a függvényhívásnak a mellékhatása. (A hívás az f csúccsal tér vissza.)



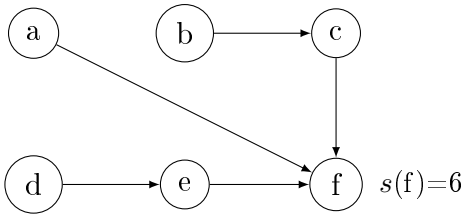
Annak megfelelően, hogy a Kruskal algoritmusban az „x := findSet($e.u$) ; y := findSet($e.v$)” utasítások az e él két végpontjához tartozó gyökércsúcsokat határozzák meg, a union(x, y) művelet később a megfelelő irányított fák gyökércsúcsait köti össze, feltéve, hogy $x \neq y$. A fenti módszerrel a 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 11.17. tulajdonságot!) Ebből pedig közvetlenül következik az, hogy a findSet(v) függvény műveletigénye $O(\log n)$, ahol $n = |G.V|$. A union() művelet hatására a 32. ábrán láthatunk példát.

11.17. 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* $\log s(r) \geq h(r)$.





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



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.

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

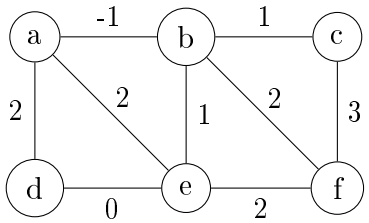
□

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 11.17. 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.

11.2.2. A Kruskal algoritmus grafikus szemléltetése a halmazműveletekkel együtt



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

33. ábra. Összefüggő, élsúlyozott, irányítatlan gráf

A 33. ábrán látható gráfra szemléltetjük a 30. ábránál részletesebb módon a Kruskal algoritmus működését, az alábbi illusztráció segítségével. Az algoritmus által kezelt feszítő erdő fáinak csúcshalmazait a megelőző, 11.2.1. alfejezetben ismertetett módon, a gyökerük felé irányított fák segítségével reprezentáljuk, ahol az r gyökércsúcsra $s(r)$ a megfelelő fa mérete. Az alábbi illusztráció sorai az algoritmus fő ciklusa iterációinak felelnek meg. Az utolsót kivéve, mindegyik sor négy részből áll.

1. Az első az irányított erdő állapota a ciklusmag végrehajtásának kezdetén. A csúcsok s címkéit csak a reprezentáns elemük (gyökerük) felé irányított fák gyökércsúcsainál tüntettük fel, mert a többi csúcsnál ezek értéke érdektelen. Mivel mindegyeik fa r gyökércsúcsánál $r = \pi(r)$, ezeknél az irányított hurokélek ábrázolásától eltekintettünk.

2. A sor második eleme az adott ciklusiterációban megvizsgált $e = u \overset{w}{—} v$ él. (Ez a legkisebb súlyú, korábban megvizsgált él. Ha több ilyen, egyenlő súlyú él van, akkor a Kruskal algoritmus prezentációjában az alfabetikusan legkisebbet vesszük sorba, az élek csúcsait is alfabetikus sorrendbe írva, pl. nem b$\overset{-1}{—}$a, hanem a$\overset{-1}{—}$b él.)

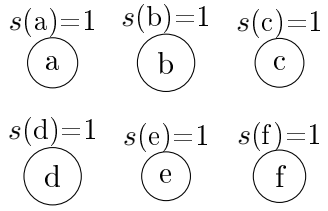
3. A sor harmadik része az irányított erdő állapota az $x :=$ findSet($e.u$); $y :=$ findSet($e.v$) utasítások végrehajtása után, az x és az y változók értékével együtt (ahol $e.u$ és $e.v$ az e él két végpontja). Mivel itt az irányított fáknak sem a gyökércsúcsa, sem a mérete nem változik, ezeknél az ábráknál a gyökércsúcsok s-értékeitől eltekintettünk.

4. A sor negyedik eleménél megállapítjuk, hogy az aktuális e él biztonságos-e, azaz, hogy éle lesz-e az algoritmusunk által meghatározott MST-nek. Ez attól függ, hogy az e két végpontjához tartozó irányított fák gyökércsúcsai (azaz x és y, amik a megfelelő csúcshalmazok reprezentáns elemei) különböznek-e, ami megfelel annak, hogy az e él a (V, A) feszítő erdő (ld. a 11.2. alfejezetet) két különböző fáját köti-e össze.

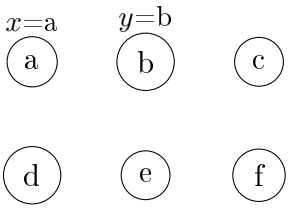
   Ha igen, akkor (V, A) az $A := A \cup \{e\}$ utasítás után is feszítő erdő marad, és a két irányított fát a union(x, y) struktogram szerint egyesítve az irányított és az irányítatlan erdő fáinak csúcshalmazai továbbra is megfelelnek egymásnak. (A union(x, y) struktogram szerint, a kisebbik fa gyökerét a nagyobbik alá csatoljuk. Ha pedig a méretek egyenlők, akkor x lesz az új gyökércsúcs, azaz az e él bal végpontjához tartozó irányított fa gyökere.)

   Ha $x = y$, akkor az e él két végpontja ugyanabban az irányított fában, azaz az eddigi biztonságos élek által kialakított feszítő erdőnek is ugyanabban a fájában van. Az e él tehát ez utóbbi, irányítatlan fában is kört képezne, tehát nem biztonságos. Ezért ezt az élt nem vesszük hozzá az irányítatlan feszítő erdő A élhalmazához, és a megfelelő irányított fákat sem egyesítjük.

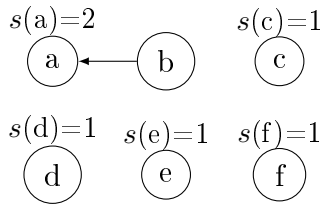
5. Az e él feldolgozása után adódó irányított erdőt a következő sor első ábráján láthatjuk.



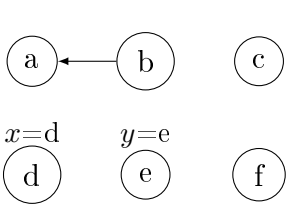
Is $\mathrm{a} \overset{-1}{—} \mathrm{b}$ safe?



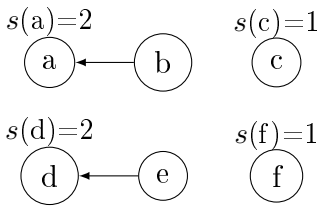
$\mathrm{a} \overset{-1}{—} \mathrm{b}$ is safe, invoke union(a,b).



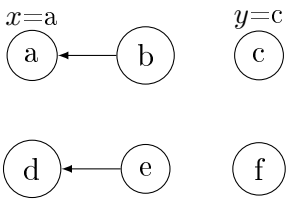
Is $\mathrm{d} \overset{0}{—} \mathrm{e}$ safe?



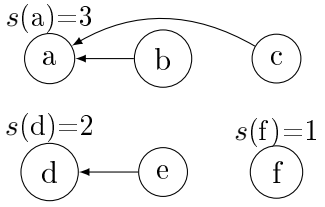
$\mathrm{d} \overset{0}{—} \mathrm{e}$ is safe, invoke union(d,e).



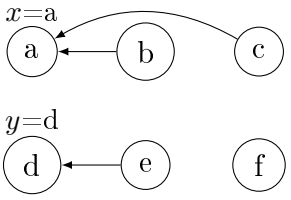
Is $b \overset{1}{-} c$ safe?



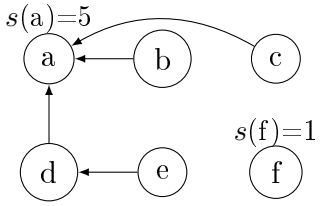
$b \overset{1}{-} c$ is safe, invoke union(a,c).



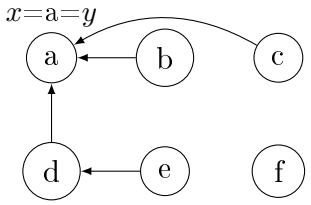
Is $b \overset{1}{-} e$ safe?



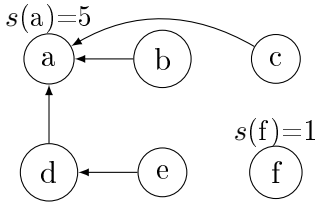
$b \overset{1}{-} e$ is safe, invoke union(a,d).



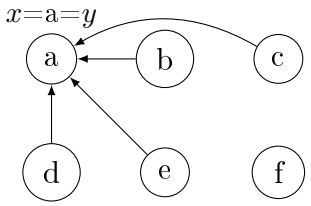
Is $a \overset{2}{-} d$ safe?



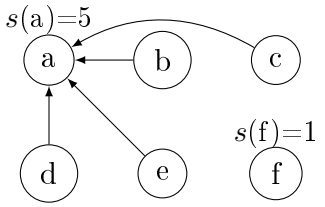
$a \overset{2}{-} d$ is unsafe, it is not an edge of the MST.



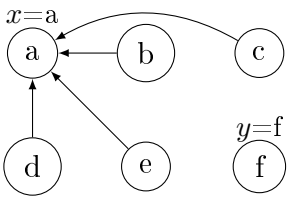
Is $a \overset{2}{-} e$ safe?



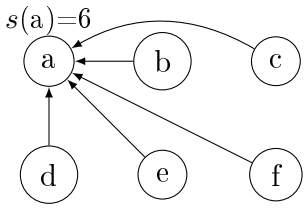
$a \overset{2}{-} e$ is unsafe, it is not an edge of the MST.



Is $b \overset{2}{-} f$ safe?



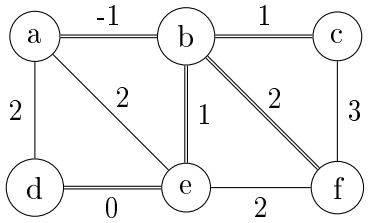
$b \overset{2}{-} f$ is safe, invoke union(a,f).



Each vertex is in the same vertex set.
Thus, the spanning forest is a single tree:
The Kruskal algorithm stops.
The edge set of the MST consists of the safe edges.

A fenti illusztrációban csak az utolsó előtti, az a$\overset{2}{—}$e él feldolgozása alatt, a findSet(e) függvényhívás végrehajtása során történik tényleges útrövidítés: az "e→d→a" irányított út rövidül "e→a"-ra.

Az algoritmus eredménye a biztonságosnak talált élek halmaza. A kapott MST pedig a gráf csúcshalmazából és ezen élekből áll össze. Ld. a 34. 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 }.

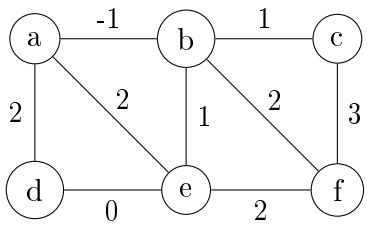
34. ábra. A 33. ábra gráfjából az utána következő táblázattal kapott biztonságos élhalmaz (jobbra), és a hozzá tartozó MST a gráfban (balra). A gráfnak a minimális feszítőfához tartozó éleit a grafikus ábrázolásban dupla vonallal jelöltük. Látható, hogy a b$\overset{2}{—}$f helyett az e$\overset{2}{—}$f 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.

11.2.3. A Kruskal algoritmus táblázatos szemléltetése a halmazműveletekkel együtt*

(Ez az alfejezet nem része a hivatalos tananyagnak.)

A 35. ábrán látható gráfra szemléltetjük a 30. ábránál részletesebb módon a Kruskal algoritmus működését, az alábbi táblázat segítségével. Az algoritmus által kezelt feszítő erdő fáinak csúcshalmazait a 11.2.1. 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.

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, a 11.2.1. alfejezetben ismertetett megfontolásoknak megfelelő módon.



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

35. ábra. Összefüggő, élsúlyozott, irányítatlan gráf

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	a2	a					d $\overset{0}{—}$ e	d1	e1	+
ab, c, de, f				d2	d		b $\overset{1}{—}$ c	ba2	c1	+
abc, de, f	a3		a				b $\overset{1}{—}$ e	ba3	ed2	+
abcde, f	a5			a			a $\overset{2}{—}$ d	a5	da5	−
abcde, f							a $\overset{2}{—}$ e	a5	eda5	−
abcde, f					a		b $\overset{2}{—}$ f	ba5	f1	+
abcdef	a6					a	-			

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.

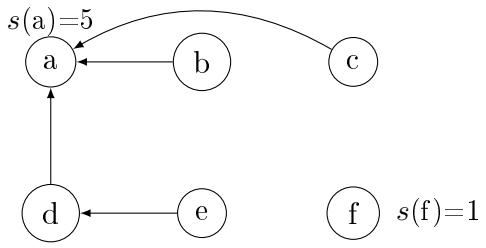
A következő hat oszlop soraiban a gráf csúcsai aktuális π és s értékeinek változásait mutatjuk meg. Ezen hat oszlop első sorában az inicializálás utáni állapotot láthatjuk: Minden $v \in G.V$ csúcsra végrehajtottuk a makeSet(v) műveletet. A további sorokban mindig a relatíve előző sorban elvégzett findSet() és union() műveleteknek a π és s értékekre gyakorolt hatását illusztráltuk. Az s-értékek csak az irányított fák gyökércsúcsainál érdekesek.

Az $u \overset{w}{—} v$ oszlopban a feszítő erdő aktuális állapotának kialakulása után feldolgozásra 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 $\overset{-1}{—}$ b; az egyenlő súlyú éleket pedig lexikografikusan növekvő sorrendben választjuk ki feldolgozásra.)

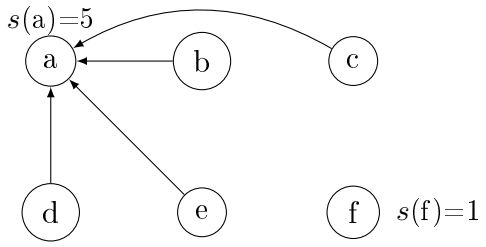
Az $u\pi^* s$ oszlopban azt a sztringet ábrázoljuk, amelynek első betűje az előbb kiválasztott él bal végpontja; a sztringben tetszőleges két, egymást követő betű közül a második az első π értéke; a szám előtt a sztring utolsó betűje a hozzá és az őt megelőző betűkhöz tartozó irányított fa gyökere; a 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 mellékhatása.

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. A hatodik sorban találjuk a eda5 sztringet, ami szerint az e már jobbról a harmadik betű, és ennek megfelelően a következő sorban már π(e)=a. A mi példánkban ez az egyetlen tényleges útrövidítés (a megfelelő irányított fában). Ld. a 36. ábrát!



36. ábra. A táblázat hatodik sorában, a gráf a$\overset{2}{-}$e élének feldolgozása alatt végrehajtódik a findSet(e) hívás. Ennek során az a gyökerű irányított fában az "eda" út (ld. fent) "ea"-ra rövidül, azaz π(e)=a lesz (ld. lent). Ez a mellékhatás a táblázat következő sorában tükröződik. (Mivel az a$\overset{2}{-}$e él feldolgozása során azt találjuk, hogy findSet(a)=findSet(e), 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.)



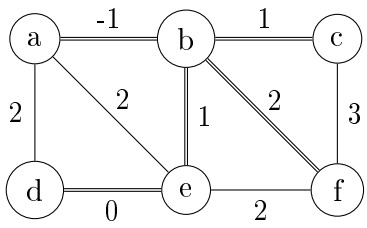
Tekintsük most azt az esetet, amikor az $u\pi^*s$ és a $v\pi^*s$ oszlopban levő sztringnek a számok előtti utolsó betűi egyenlők! Világos, hogy tetszőleges

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 $u \underline{\overset{w}{—}} v$ 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: az "eda" irányított út rövidül "ea"-ra. Ld. a 36. ábrát!)

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 irányított 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 union(x, y) struktogram szerint x lesz az új gyökércsúcs, azaz a táblázat adott sorában feldolgozott irányítatlan él bal végpontjához tartozó irányított fa gyökere.)

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 37. 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$\}$.

37. ábra. A 35. 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.

11.3. A Prim-Jarník algoritmus

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 $(N, V\backslash N)$ vágásban. (Ld. a 11.11. tételt!)

A megfelelő könnyű él hatékony meghatározása céljából egy Q min-prioritásos sorban tartjuk nyilván a $V\backslash N$ csúcshalmazt. Mindegyik $u \in V\backslash N$ 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 $(N, V\backslash N)$ 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 11.11. 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.

Prim($G : \mathcal{G}_w$; $r : \mathcal{V}$)

$\forall v \in G.V$
$\quad c(v) := \infty$ // costs are still infinite
$\quad$ // edge $(p(v), v)$ will be in the MST where $c(v) = G.w(p(v), v)$
$c(r) := 0$; $p(r) := \oslash$ // except for the root: r is the root of the MST
// let Q be a minimum priority queue of $G.V \setminus \{r\}$ by label values $c(v)$:
Q : minPrQ($G.V \setminus \{r\}, c$) // $c(v)$ = cost of light edge to (partial) MST
$u := r$ // vertex $u = r$ has become the first node of the (partial) MST
$\neg Q$.isEmpty()
$\quad$ // neighbors of u may have come closer to the partial MST
$\quad \forall v \in G.A(u) : v \in Q \wedge c(v) > G.w(u, v)$
$\quad\quad p(v) := u$; $c(v) := G.w(u, v)$; Q.adjust(v)
$\quad u := Q$.remMin() // $(p(u), u)$ is a new edge of the MST

A Prim-Jarník 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 $\forall v \in G.A(u)$ 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.

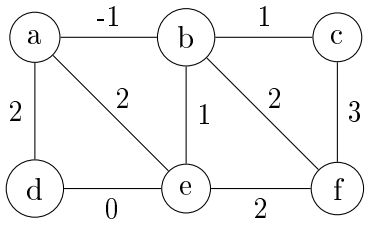
A fenti részeredményeket összeadva a Prim-Jarník algoritmus teljes futási idejére $\Theta(n)+\Theta(n)+O(n*\log n)+O(m*\log n) = O((n+m)*\log n)$ adódik. Mivel a gráf összefüggő, $m \geq n-1$, így végül

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

11.18. Feladat. *Implementálja a Prim-Jarník 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?*

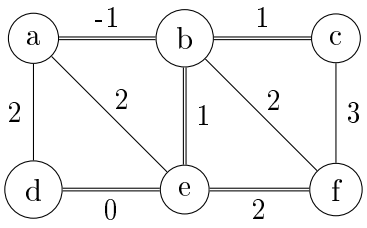
11.19. Feladat. *Implementálja a Prim-Jarník algoritmust szomszédossági listás gráfábrázolás esetén! Ügyeljen arra, hogy a prioritásos sor megfelelő megvalósításával biztosítani kell az elméleti $O(m * \lg n)$ műveletigényt!*

Az alábbi, már ismerős gráfon szemléltetjük a Prim-Jarník algoritmus működését, a **d** csúcsból indítva. Az inicializálás után mindegyik sorban azt a (még benne nem lévő) csúcsot vesszük hozzá a részleges feszítőfához, amelyik a legközelebb van hozzá. Először tehát a **d** csúcsot vesszük hozzá. Ezzel az **a** és az **e** csúcsok a pillanatnyi (egy csúcsú) fa közvetlen szomszédai lesznek, 2, illetve 0 távolsággal. Most az **e** csúcs van legközelebb a fához. Hozzávesszük tehát a fához az ahhoz kapcsoló e$\overset{0}{—}$d éllel együtt. Emiatt az **e** csúcs **b** és **f** szomszédai kerülnek közelebb a fához stb.



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

in-	changes of c and p labels					
to	a	b	c	d	e	f
MST	$\infty \oslash$	$\infty \oslash$	$\infty \oslash$	$0 \oslash$	$\infty \oslash$	$\infty \oslash$
d	2 d			|	0 d	
e $\overset{0}{-}$ d		1 e		|	|	2 e
b $\overset{1}{-}$ e	-1 b	|	1 b	|	|	
a $\overset{-1}{-}$ b	|	|		|	|	
c $\overset{1}{-}$ b	|	|	|	|	|	
f $\overset{2}{-}$ e	|	|	|	|	|	|



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

12. Legrövidebb utak egy forrásból ([3] 22 ; [6, 11])

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 $s \rightsquigarrow u \rightarrow v$ 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.)

12.1. 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. Erre

Dijkstra($G : \mathcal{G}_w$; $s : \mathcal{V}$)

$\forall v \in G.V$
$d(v) := \infty$; $\pi(v) := \oslash$ // distances are still ∞, no parent
// Later: $\pi(v) =$ parent of v on $s \rightsquigarrow v$ where $d(v) = w(s \rightsquigarrow v)$
$d(s) := 0$ // but s is the root of the shortest-path tree, s has no parent
// let Q be a minimum priority queue of $G.V \backslash \{s\}$ by label values $d(v)$:
Q : minPrQ($G.V \backslash \{s\}, d$)
$u := s$ // going to calculate shortest paths form s to other vertices
$d(u) < \infty \wedge \neg Q$.isEmpty()
// check, if $s \rightsquigarrow u \rightarrow v$ is shorter than $s \rightsquigarrow v$ before
$\forall v \in G.A(u) : d(v) > d(u) + G.w(u, v)$
$\pi(v) := u$; $d(v) := d(u) + G.w(u, v)$; Q.adjust(v)
$u := Q$.remMin() // $s \rightsquigarrow u$ is optimal now, if it exists

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

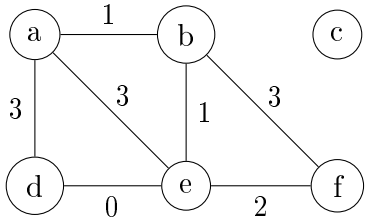
Másrészt $mT_{\mathrm{Dijkstra}}(n, m) \in \Theta(n)$, mivel a 2. ciklus előtti rész műveletigénye már $\Theta(n)$, amihez csak a második ciklus egyetlen iterációja és a belső ciklus nullaszori iterációja adódik hozzá, amennyiben nincs a gráfban s-nek rákövetkezője. Ebben az esetben a második ciklus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

12.1. Feladat. *Implementálja a Dijkstra algoritmust szomszédossági mátrixos gráfábrázolás esetén! A prioritásos sort rendezetlen tömbbel reprezentál-*

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

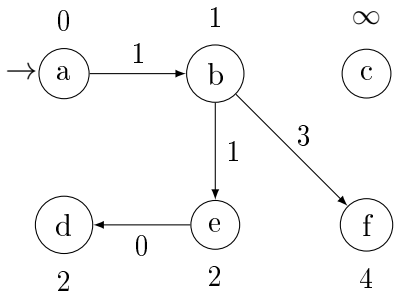
12.2. Feladat. *Implementálja a Dijkstra algoritmust szomszédossági listás gráfábrázolás esetén! Ügyeljen arra, hogy a prioritásos sor megfelelő megvalósításával biztosítani lehet az elméleti $O((m+n)*\lg n)$ műveletigényt!*

Az alábbi gráfon szemléltetjük a Dijkstra algoritmus működését, az **a** csúcsból indítva. A Dijkstra algoritmusnál a *ki nem terjesztett* csúcsokat röviden *nyílt* csúcsoknak nevezzük. Az algoritmus mindig a nyílt csúcsok küzül választja ki kiterjesztésre a(z egyik) legkisebb d-értékűt. Ha a minimális d-érték ∞, vagy a kiválasztás után már nem marad nyílt csúcs, az algoritmus megáll.



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

ex-	changes of d and π labels					
pand	a	b	c	d	e	f
$d\,\mathcal{V}$	$0\,\oslash$	$\infty\,\oslash$	$\infty\,\oslash$	$\infty\,\oslash$	$\infty\,\oslash$	$\infty\,\oslash$
0 a	|	1 a		3 a	3 a	
1 b	|	|			2 b	4 b
2 e	|	|		2 e	|	
2 d	|	|		|	|	
4 f	|	|		|	|	|



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

12.3. 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. □

12.4. 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. □

12.5. 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.

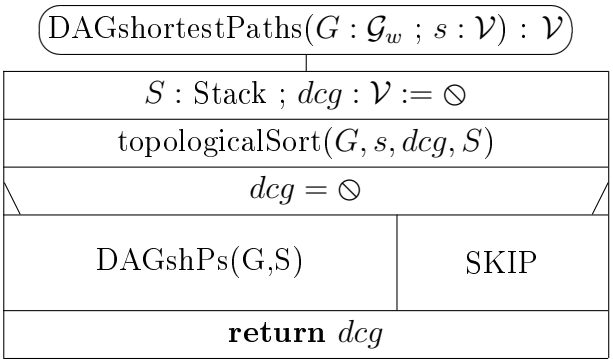
Ha pedig egyszer a kiterjesztésre kiválasztott u csúcsra már $d(u) = \infty$, akkor $Q \cup \{u\}$ egyetlen eleme sem érhető már el az s csúcsból, és megállhatunk: mindegyik d-értéke végtelen, π-érteke pedig $\oslash$, míg a korábban, véges d-értékkel kiterjesztett csúcsok éppen azok, amelyek elérhetők s-ből, és ezekbe találtunk is optimális utat.

12.2. DAG legrövidebb utak egy forrásból (DAGshP)

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

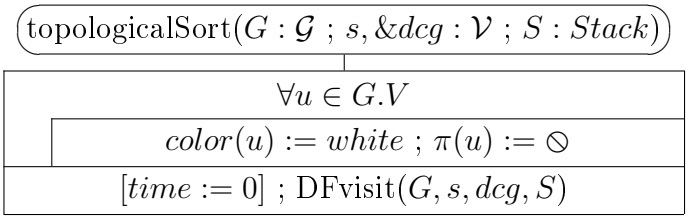
Ebben az esetben nincs a gráfban s-ből elérhető negatív kör sem, így a *legrövidebb utak egy forrásból* feladat megoldható.

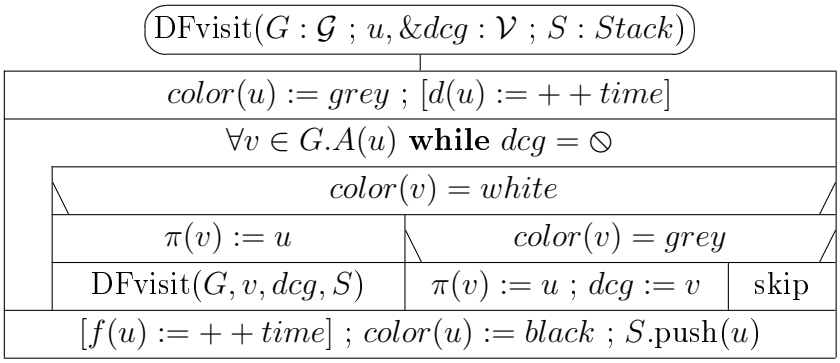
Ez az algoritmus ellenőrzi az előfeltételét. Ha teljesül, a DAGshortestPaths() függvény $\oslash$ értékkel tér vissza. Különben megtalál egy irányított kört, aminek egyik csúcsával tér vissza. Ebből indulva a π-címkék mentén a kör fordított irányban bejárható.

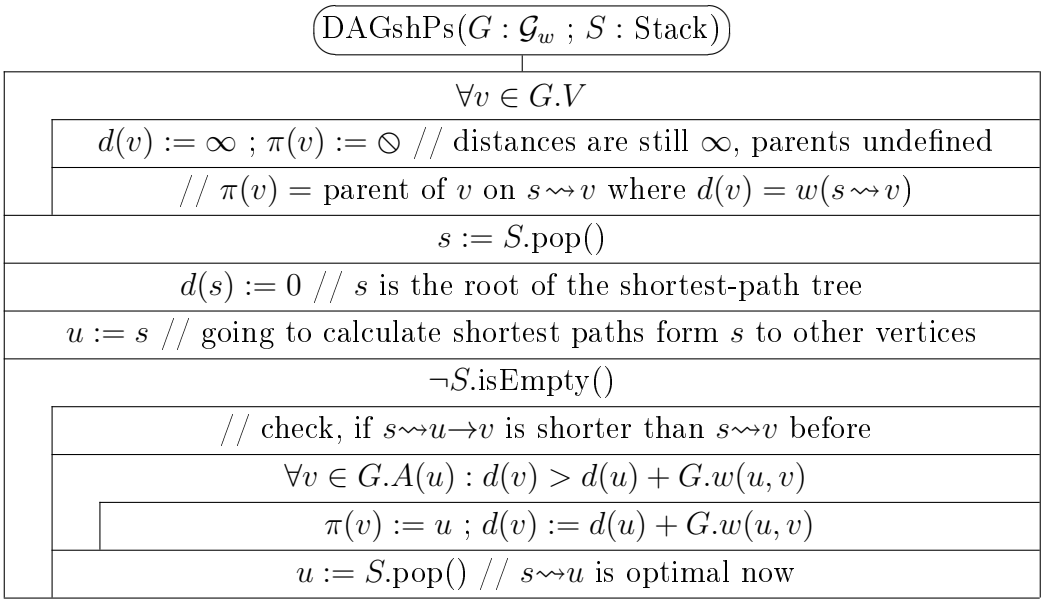


A topologikus rendezés csak az s-ből elérhető részgráfot próbálja topologikusan rendezni.

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.







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

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

12.7. 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. □

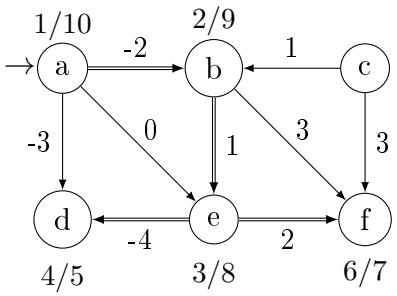
12.8. Feladat. *Implementálja a* DAG legrövidebb utak egy forrásból *algoritmust szomszédossági mátrixos gráfábrázolás esetén!*

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

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

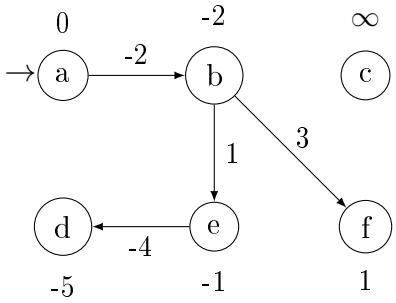
Mekkora lesz a műveletigény? Tartható-e az elméleti maximális, illetve minimális műveletigény ezzel az ábrázolással?

A következő oldalon látható gráfon szemléltetjük a *DAG legrövidebb utak egy forrásból* algoritmust, ahol $s =$ 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. Az utolsó csúcs nem kerül kiterjesztésre, ennek ui. már nem lehet a gráfban sem rákövetkezője.



a → b, -2 ; d, -3 ; e, 0.
b → e, 1 ; f, 3.
c → b, 1 ; f, 3.
e → d, -4 ; f, 2.
$S = \langle$ a, b, e, f, d $\rangle$

ex-	changes of d and π labels					
pand	a	b	c	d	e	f
$d\,\mathcal{V}$	0 ⊘	∞ ⊘	∞ ⊘	∞ ⊘	∞ ⊘	∞ ⊘
0 a		-2 a		-3 a	0 a	
-2 b					-1 b	1 b
-1 e				-5 e		
1 f						



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

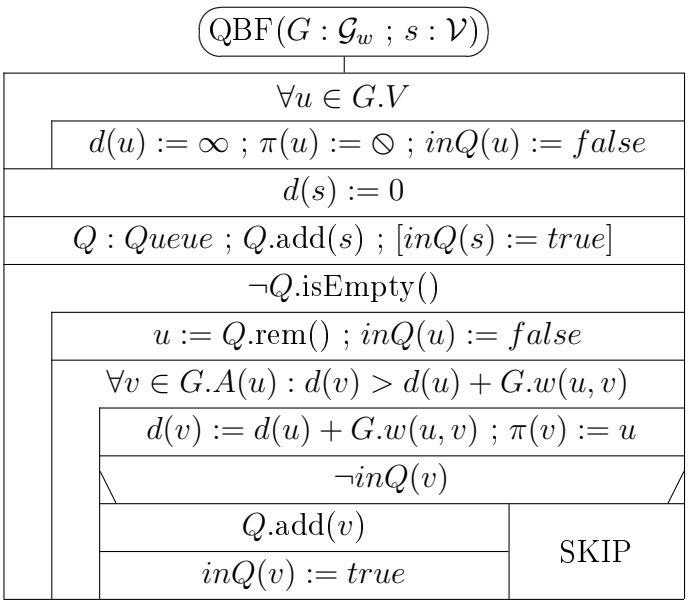
12.3. A QBF algoritmus

A *QBF* a *Queue-based Bellman-Ford* rövidítése. Magyarul, teljes nevén *Sor-alapú Bellman-Ford algoritmus*. A *Sor-alapú* vagy *Queue-based* jelzői kifejezés elhagyása hiba, mert létezik több *Bellman-Ford* algoritmus is, amelyek a *QBF*-től lényegesen különböző módon oldják meg a *legrövidebb utak egy forrásból* problémát. A továbbiakban a *QBF* algoritmust tárgyaljuk.

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

A *QBF algoritmus* (tudományos elemzés és negatív kör vizsgálat nélkül) az informatikai folklórból származik. Legjobb tudásunk szerint először Tarján Róbert Endre amerikai matematikus[16] elemezte és publikálta, *Breadth-first Scanning* néven [6].

A QBF hasonlít a szélességi kereséshez, de az utak hosszát a Dijkstra, a DAGshP (és a Bellman-Ford [3, 4]) 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.



[16] Tarján professzor úr szülei magyar származásúak.

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 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.

12.3.1. 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 [6]. Itt e jegyzet szerzőjének egy viszonylag egyszerű, de könnyen ellenőrizhető kritériumát fogjuk használni [11].

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, legfeljebb n–1 lépésben (azt a szélsőséges esetet sem kizárva, amikor ezt a kört egyetlen negatív hurokél alkotja).

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.

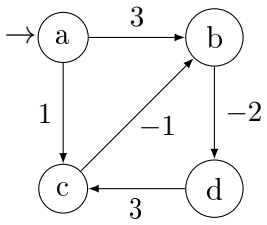
Ha valamely (u, v) él mentén végzett közelítés során úgy találjuk, hogy $e(v) \geq n$, akkor a fentiek szerint meghatározzuk valamelyik negatív kör egyik csúcsát, és az algoritmus ezzel tér vissza.

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 kiegészített QBF $O(n*m)$ idő alatt minden esetben befejeződik.

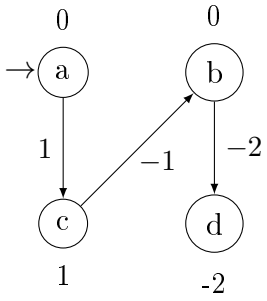
12.3.2. A legrövidebb utak fája meghatározásának szemléltetése

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



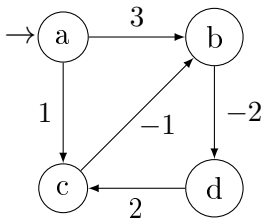
a → b, 3 ; c, 1.
b → d, −2.
c → b, −1.
d → c, 3.

ex-	changes of $d\,\pi\,e$				Q :
pand	a	b	c	d	Queue
$d\,\mathcal{V}\,e$	$0 \oslash 0$	$\infty \oslash$	$\infty \oslash$	$\infty \oslash$	⟨ a ⟩
0 a 0		3 a 1	1 a 1		⟨ b, c ⟩
3 b 1				1 b 2	⟨ c, d ⟩
1 c 1		0 c 2			⟨ d, b ⟩
1 d 2					⟨ b ⟩
0 b 2				−2 b 3	⟨ d ⟩
−2 d 3					⟨ ⟩



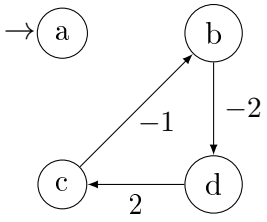
A legrövidebb utak fája s = a esetén. A csúcsoknál csak a d-értékeket tüntettük föl.

12.3.3. A negatív körök kezelésének szemléltetése



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

ex-	changes of $d\,\pi\,e$				Q :
pand	a	b	c	d	Queue
$d\,\mathcal{V}\,e$	$0 \oslash 0$	$\infty \oslash$	$\infty \oslash$	$\infty \oslash$	⟨ a ⟩
0 a 0		3 a 1	1 a 1		⟨ b, c ⟩
3 b 1				1 b 2	⟨ c, d ⟩
1 c 1		0 c 2			⟨ d, b ⟩
1 d 2					⟨ b ⟩
0 b 2				−2 b 3	⟨ d ⟩
−2 d 3			0 d ④		⟨ ⟩

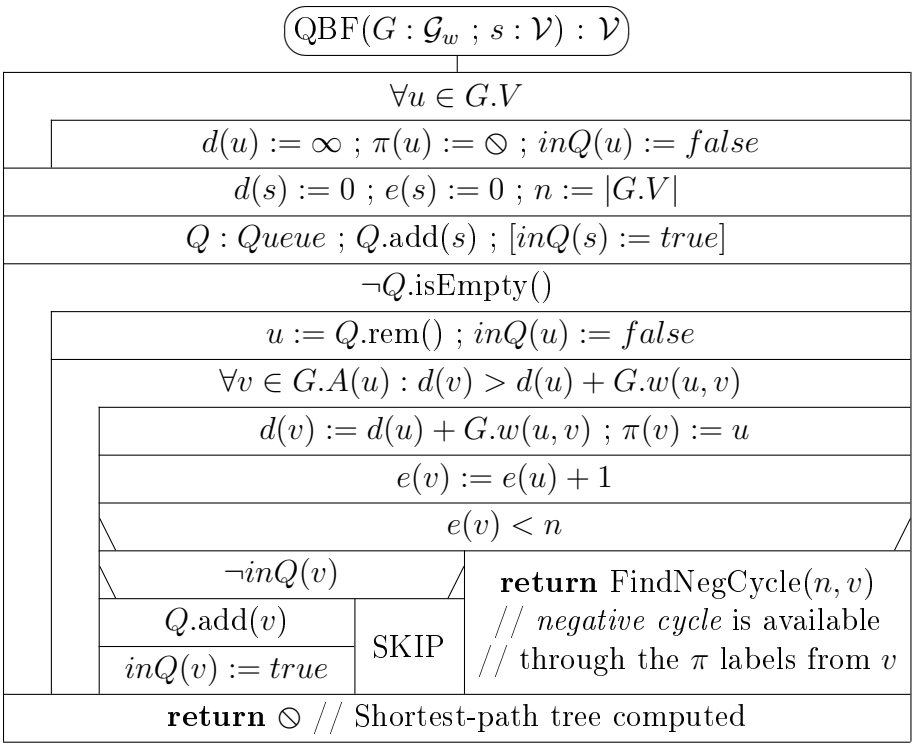


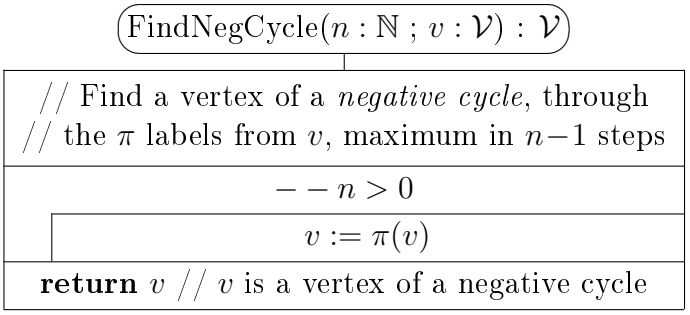
Itt megállunk, mert $e(\mathrm{c}) = 4 = n$, tehát negatív kör található a „c” csúcsból a π értékek szerint visszafelé haladva.

12.3.4. 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 fentebb megadott változata alkalmazható.





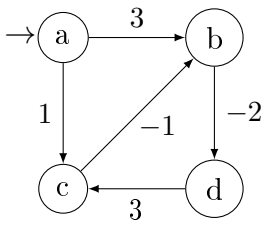
12.3.5. A QBF algoritmus elemzése

Az alábbi elemzésben arra az esetre szorítkozunk, amikor a gráfban nincs s-ből elérhető negatív kör. (A negatív körök esetét ld. a [11] cikkben!)

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

12.10. 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.

ex-	changes of $d\,\pi\,e$				Q :	
pand	a	b	c	d	Queue	Pass
$d\,\mathcal{V}\,e$	0 ⊘ 0	∞ ⊘	∞ ⊘	∞ ⊘	⟨ a ⟩	
0 a 0		3 a 1	1 a 1		⟨ b, c ⟩	0.
3 b 1				1 b 2	⟨ c, d ⟩	1.
1 c 1		0 c 2			⟨ d, b ⟩	1.
1 d 2					⟨ b ⟩	2.
0 b 2				−2 b 3	⟨ d ⟩	2.
−2 d 3					⟨ ⟩	3.

12.11. 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. □

12.12. 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. □

12.13. 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

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

A fenti műveletigény bizonyításához elég meggondolni, hogy mindegyik menet $O(m)$ műveletigényű, legfeljebb n menet van ugyanis. Az egyes menetek $O(m)$ műveletigényének kulcsa pedig az, hogy egyrészt a Q sorban (a $\neg inQ(v)$ ellenőrzés miatt) nincsenek duplikált csúcsok, másrészt mindegyik menet elején $|Q| \leq m$.

Ez utóbbi állítás bizonyításánál feltehető, hogy $m > 0$. A nulladik menet elején csak a start csúcs van a sorban, így $|Q| \leq m$ teljesül. Ha pedig az i-edik menet elején legfeljebb m csúcs van a sorban, amelyek páronként különböznek, akkor az ezekből kimenő élek is különbözők: legfeljebb m db (hiszen irányított gráfokról beszélünk). Így az i-edik menetben legfeljebb m csúcsot kell a sorból eltávolítani és kiterjeszteni, valamint legfeljebb m élet feldolgozni. Továbbá az i-edik menetben a sorba kerülő csúcsok a feldolgozott élek végpontjai, amelyek száma szintén $\leq m$. Ezért egyrészt az i-edik menet végén is teljesül a $|Q| \leq m$ invariáns, másrészt az i-edik menet műveletigénye $O(m)$ $(i \in [0\,..\,n))$.

A fenti, elméleti műveletigény becslés meglehetősen rossz hatékonyságot sugall. Ezzel szemben, gyakorlati tesztek, pozitív élsúlyú, véletlenszerűen generált, nagyméretű, s-ből összefüggő ritka gráfokra[17] $(m \in O(n))$ 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.)

[17] 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 $k \leq 4$.

13. Utak minden csúcspárra ([3] 23)

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

13.1. Bevezetés a dinamikus programozásba

A *dinamikus programozás (dynamic programming)* hasonló az *oszd meg és uralkodj* programozási elvhez (*divide-and-conquer* method), amit az összefésülő és a gyorsrendezésnél alkalmaztunk. Van néhány triviális esetünk, azaz olyan esetek, amelyek közvetlenül megoldhatók. A bonyolultabb vagy nagyobb problémákat kisebb részfeladatokra bontjuk, ezeket (rekurzívan) megoldjuk, majd a megoldásaikat összevonva oldjuk meg az eredeti problémát.

A fő különbség az, hogy az *oszd meg és uralkodj* módszer a részfeladatokat egymástól függetlenül oldja meg, mint például a fenti rendezések esetében. Ezért akkor hatékony, ha a részfeladatok általában egymástól függetlenek. Hatékonysága elvész, ha egy nagyobb probléma rekurzív módon tartalmaz közös részfeladatokat és rész-részfeladatokat. A *dinamikus programozás* során minden részfeladatot csak egyszer oldunk meg, megjegyezzük a megoldását, és ezt hasznosítjuk, ha később újra találkozunk vele.

Vegyük példának a Fibonacci függvény definícióját!
$F : \mathbb{N} \to \mathbb{N}$
$F_n = F_{n-1} + F_{n-2} \quad$ if $n > 1$
$F_1 = 1 \quad$ and $\quad F_0 = 0.$

Amennyiben ezt a definíciót közvetlenül kódoljuk, akkor $n > 1$ esetén az F_n kiszámítását az F_{n-1} és az F_{n-2} kiszámítására bontjuk, majd összeadjuk a két részeredményt. (A két rekurzív hívás eredményének összeadásával „uralkodunk”.) Például:
$F_9 = F_8 + F_7 = (F_7 + F_6) + F_7 = ([F_6 + F_5] + F_6) + (F_6 + F_5) =$
$([\{F_5 + F_4\} + F_5] + \{F_5 + F_4\}) + (\{F_5 + F_4\} + F_5) = \ldots$

Így az F_9-et és az F_8-at csak egyszer számoljuk ki, az F_7-et kétszer, az F_6-ot háromszor, F_5-öt ötször, és így tovább. Általában az F_{n-k}, F_{k+1} alkalommal

kerül kiszámításra, amiből az következik, hogy az F_n értékének kiszámítása exponenciálisan függ n-től.

Ezzel ellentétben, az F_n (továbbra is csak egész aritmetikát használva) lineáris időben is kiszámítható, ha a számítást az F_2, F_3, F_4 stb. kiszámításával kezdjük, és ugyanebben a sorrendben folytatjuk, miközben végig az utolsó két részeredményre kell emlékeznünk.

Fent a dinamikus programozás egy triviális esetét láttuk. A most következő két algoritmusnál majd kicsit bonyolultabb módon alkalmazzuk. Figyeljük meg, hogy a rekurzív képletek ellenére a mátrixsorozatok elemeit sorban számoljuk ki, és mindegyik részeredmény mátrixot, illetve márixpárt az előzőből számoljuk ki! (Az csak a hab a tortán, hogy az előző részeredményre való emlékezésnek nem lesz majd extra tárigénye.) A Tisztelt Olvasó könnyen utánaszámolhat, hogy a rekurzív képletek (13.9. és 13.14. Tulajdonság) közvetlen programozása a most következő algoritmusok esetében a futási időnek még a fenti példánál is meredekebb emelkedését idézné elő.

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

13.1. 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.

13.2. 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 $k \in 0\,..\,n$).*

13.3. 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.

13.4. 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:* $i \underset{opt}{\overset{k}{\rightsquigarrow}} j = i \underset{opt}{\overset{k-1}{\rightsquigarrow}} j \quad \vee \quad i \underset{opt}{\overset{k}{\rightsquigarrow}} j = i \underset{opt}{\overset{k-1}{\rightsquigarrow}} k \underset{opt}{\overset{k-1}{\rightsquigarrow}} j.$

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

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

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

13.8. 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.

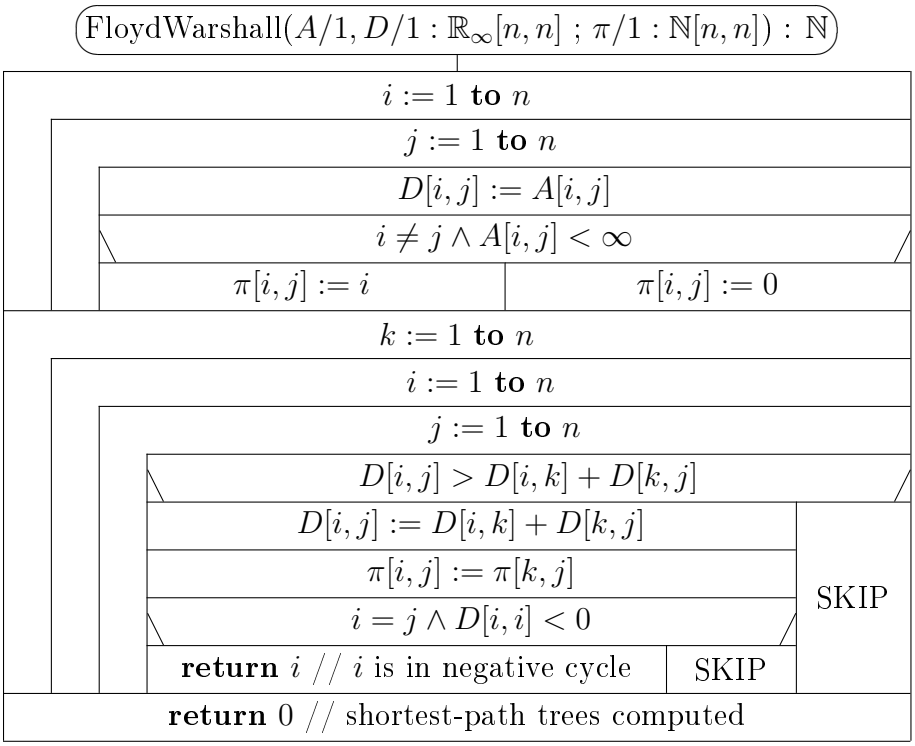
13.9. Tulajdonság. *A 13.4. megjegyzés alapján, $k \in 1\,..\,n$ esetén:*
Ha $D_{ij}^{(k-1)} > D_{ik}^{(k-1)} + D_{kj}^{(k-1)}$
akkor $D_{ij}^{(k)} = D_{ik}^{(k-1)} + D_{kj}^{(k-1)} \wedge \pi_{ij}^{(k)} = \pi_{kj}^{(k-1)}$
különben $D_{ij}^{(k)} = D_{ij}^{(k-1)} \wedge \pi_{ij}^{(k)} = \pi_{ij}^{(k-1)}$

13.10. 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 $mT(n) \in \Theta(n^2)$.

13.2.1. 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 nagyságrenddel 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.

13.3. 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.

13.11. 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.*

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

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

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

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

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

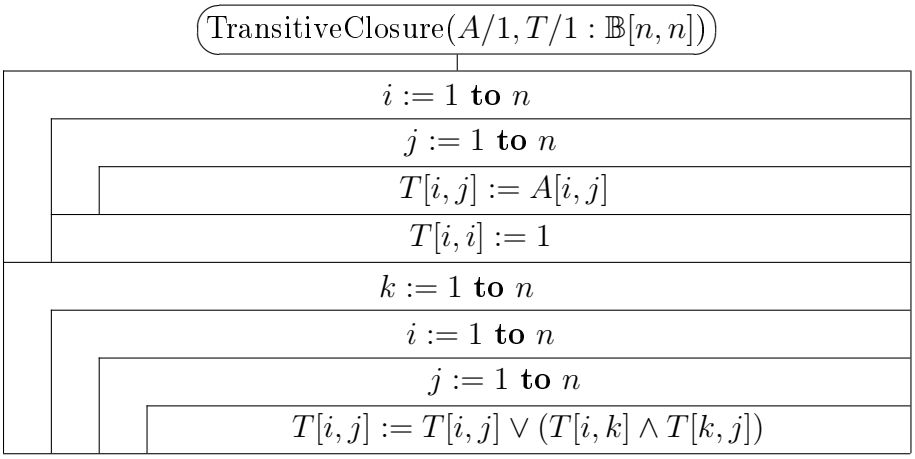
13.15. Következmény.

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

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.

13.3.1. 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 maximális műveletigény, ami a gyakorlatban hosszabb futási időt eredményez, a nagyon egyszerű TC algoritmussal összehasonlítva.

14. Mintaillesztés (String-matching [3] 32)

14.1. Jelölések. *(Notations)*
$\mathbb{N} = \{\, i \in \mathbb{Z} \mid i \geq 0 \,\}$ $\quad i\,..\,k = \{\, j \in \mathbb{N} \mid i \leq j \leq k \,\}$
$[i\,..\,k) = \{\, j \in \mathbb{N} \mid i \leq j < k \,\}$ $\quad (i\,..\,k) = \{\, j \in \mathbb{N} \mid i < j < k \,\}$
$\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_d\}$ *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.

14.2. 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).

14.3. 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*

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

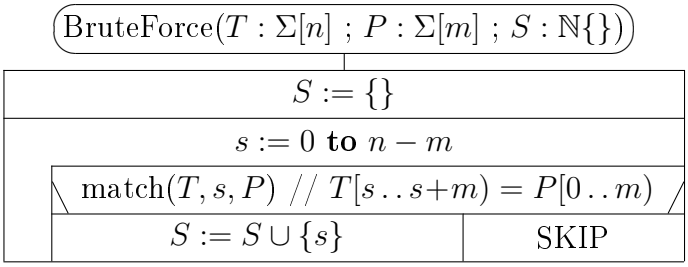
14.1. Egyszerű mintaillesztés (Naive string-matching)

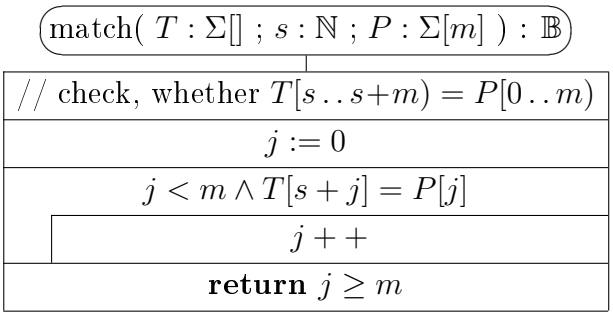
Vegyük bevezetésként a következő példát! A $P[0\,..\,4) = BABA$ mintát keressük a $T[0\,..\,11) = ABABBABABAB$ szövegben. ($\underline{B}$: B-t sikeresen illesztette a szöveg megfelelő betűjére; $\not{B}$: sikertelenül illesztette.)

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

$i=$	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

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





A match(T, s, P) függvényre: $MT_{\text{match}}(m) \in \Theta(m)$
$mT_{\text{match}}(m) \in \Theta(1)$
Innét a teljes algoritmusra: $MT_{\text{BF}}(n, m) \in \Theta((n - m + 1) * m)$
$mT_{\text{BF}}(n, m) \in \Theta(n - m + 1)$
$m \approx \lfloor n/2 \rfloor$ esetén: $MT_{\text{BF}}(n) \in \Theta(n^2)$
$mT_{\text{BF}}(n) \in \Theta(n)$

A naiv mintaillesztés műveletigényének részletesebb elemzése*

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:

14.4. Következmény. *Ha egy feladatosztályon valamely k konstansra* $0 < k < 1 \wedge m \leq k * n \Rightarrow mT_{BF}(n, m) \in \Theta(n) \wedge MT_{BF}(n, m) \in \Theta(n * m)$

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 a 14.4 következménnyel adódik az alábbi eredmény:

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

14.2. 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.

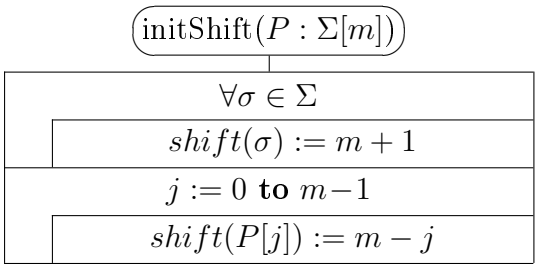
Tegyük fel most, hogy $\sigma = T[s + m]$. Ekkor a $shift(\sigma)$ érték megmondja, hogy a $T[s\,..\,s{+}m) = P[0\,..\,m)$ összehasonlítás után legalább mennyivel kell (jobbra) eltolni a P mintát a szövegen ahhoz, hogy a $T[s{+}m]$ alapján legyen esély a mintának a megfelelő szövegrészhez való illeszkedésére.
– $\sigma \in P[0\,..\,m)$ 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 $shift$ értékeket a következő táblázat mutatja.

σ	A	B	C	D
$shift(\sigma)$	1	5	4	2



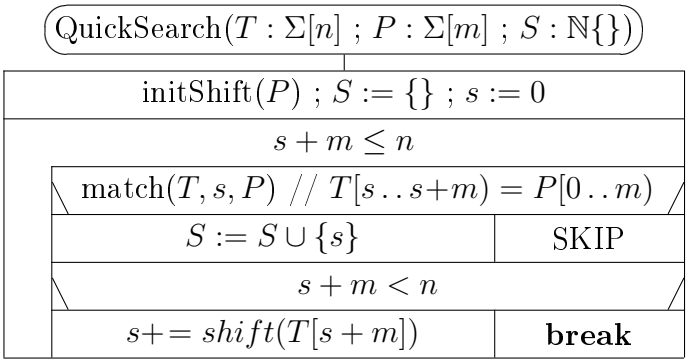
Az ábécé méretét konstansnak tekintve $T_{\text{initShift}}(m) \in \Theta(m)$ adódik.

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																		
$s = 6$							$\underline{C}$	$\underline{A}$	$\underline{D}$	$\underline{A}$													
												$\underline{C}$	$\underline{A}$	$\not B$	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.

14.3. 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, alapfogalmakat fogjuk használni.

14.6. 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 $z + x = y$. ($ABBA, BBA, BA, A, \varepsilon \sqsupseteq ABBA$)*

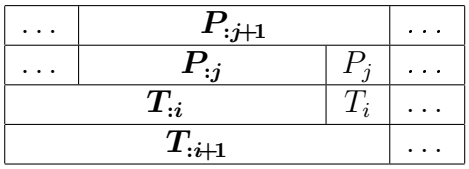
- *Ha x és y két sztring, akkor $x \sqsupset y$ (x az y* valódi szuffixe [proper suffix]*) azt jelenti, hogy $x \sqsupseteq y \wedge x \neq y$. ($BBA, BA, A, \varepsilon \sqsupset ABBA$)*

- *Ha x és y két sztring, akkor x* az y prefix-szuffixe *azt jelenti, hogy $x \sqsubset y \wedge x \sqsupset y$ (x* az y valódi prefixe és szuffixe*).*

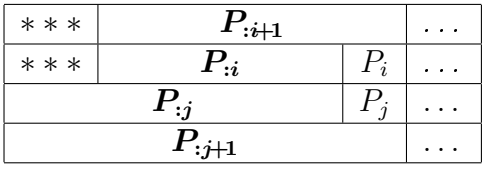
- *$P_{:j} = P[0\,..\,j) = P[0\,..\,j-1]$ (csak ebben az alfejezetben) $P_{:j}$ a P sztring j hosszúságú prefixe, azaz kezdőszelete. $P_{:0} = \varepsilon$ az üres prefixe. Hasonlóan $T_{:i} = T[0\,..\,i)$.*
  [Eszerint minden sztringnek szuffixe *az üres sztring, azaz $P_{:0} \sqsupseteq P_{:j}$ $(j \in 0\,..\,m)$, és minden nemüres sztringnek* valódi szuffixe *az üres sztring, azaz $P_{:0} \sqsupset P_{:j} \quad (j \in 1\,..\,m)$.]*

A következő lemma hasznos lesz [3]. Mindkét állítást az azt követő ábrával szemléltetjük.

14.7. Lemma. *Szuffixkiterjesztési (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!

14.8. 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 $P = P_{:6} = BABABB$. Az aktuális eltolás $i-5$. A sikeresen illesztett betűket aláhúztuk. A sikertelenül illesztett karaktert pedig áthúztuk.*

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

A fenti 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 tehát a P minta egy, az eddiginél $(i-5)$ nagyobb eltolását keressük, ahol van esély arra, hogy ez egy érvényes eltolás lesz, és közben biztosan nem ugrunk át érvényes eltolást. Ennek az ígéretes eltolásnak a keresésénél a KMP algoritmusnál csak a T szöveg már korábban megvizsgált, első i betűjét, azaz $T_{:i}$ kezdőszeletét vesszük figyelembe.

$i-4$ eltolásnál a $T_{i-4} = A$ betűvel a $P_0 = B$ betűt szeretnénk illeszteni, ami sikertelen lenne, és a $i-2$ és az i eltolásnál is hasonló a helyzet $(T_{i-2} \neq P_0)$. Ezek tehát érvénytelen eltolások.

$i-3$ eltolásnál azonban, mint az alábbi táblázat utolsó előtti sorából látható, $T[i-3\,..\,i) = BAB = P_{:3}$, és hasonlóan, $i-1$ eltolásnál, mint a táblázat utolsó sora szemlélteti, $T[i-1\,..\,i) = B = P_{:1}$. Ezek tehát, a szöveg $T_{:i}$ prefixét figyelembe véve hasonlóan ígéretesek. Ha viszont a kettő közül a nagyobbik eltolást választanánk, átugornánk egy érvényes eltolást, mint a táblázat utolsó előtti sora mutatja.

$\ldots$	T_{i-5}	T_{i-4}	T_{i-3}	T_{i-2}	T_{i-1}	T_i	T_{i+1}	T_{i+2}
$\ldots$	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	$\underline{A}$	$\underline{B}$	$\underline{B}$
					B	A	B	$\ldots$

A fentiek szerint tehát a P mintának legkisebb, az aktuálishoz képest további eltolását *keressük úgy, hogy a minta illeszkedő, itt $P_{:5}$ kezdőszeletének az a $P_{:k}$ $(0 \leq k < 5)$ prefixe, ami még a szöveg $T[i-k\,..\,i)$ része alatt van, illeszkedjen arra, azaz $P_{:k} = T[i-k\,..\,i)$, vagy ami ugyanaz, $P_{:k} \sqsupset T_{:i}$ teljesüljön. (Lásd a fenti táblázat utolsó előtti sorát!)*

Ez a $P_{:k}$-t meghatározó legkisebb további eltolás *legfeljebb 5, vagy általában a minta aktuálisan illeszkedő prefixének a hossza, 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 $0 \leq h < j \wedge P_{:h} \sqsupset T_{:i}$ teljesül: Ezt a leghosszabb $P_{:h}$-t keressük.

Továbbá $P_{:h} \sqsupset T_{:i}$ ekvivalens a $P_{:h} \sqsupset P_{:j}$ relációval, mert $P_{:j} \sqsupseteq T_{:i} \wedge 0 \leq h < j$, mint az alábbi ábra mutatja. (Más szavakkal, $P[0\,..\,h) = T[i-h\,..\,i)$ ekvivalens a $P[0\,..\,h) = P[j-h\,..\,j)$ relációval, mert $P[0\,..\,j) = T[i-j\,..\,i) \wedge 0 \leq h < j$.) Tehát azt a leghosszabb $P_{:h}$-t keressük, amire $0 \leq h < j \wedge P_{:h} \sqsupset P_{:j}$.

$\ldots$	$\boldsymbol{T_{i-j}}$	$\ldots$	$\boldsymbol{T_{i-h}}$	$\ldots$	$\boldsymbol{T_{i-1}}$	T_i	$\ldots$
$P_{:j} =$	$\boldsymbol{P_0}$	$\ldots$	$\boldsymbol{P_{j-h}}$	$\ldots$	$\boldsymbol{P_{j-1}}$	$\ldots$	$\ldots$
			P_0	$\ldots$	P_{h-1}	$\ldots$	$\ldots$

Ennek a hosszát a π prefix-függvény határozza meg.

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

Más szavakkal, a leghosszabb $P_{:h}$-t keressük, amire $P_{:h} \sqsubset P_{:j} \wedge P_{:h} \sqsupset P_{:j}$ (azaz $P_{:h}$ a $P_{:j}$ *prefix-szuffixe*).

Ennek a függvénynek a hatékony kiszámítását később, 14.3.2-ben részletezzük. Előbb azonban megnézzük a KMP algoritmus még egy konkrét esetét, majd 14.3.1-ben megadjuk és elemezzük a fő eljárás struktogramját. Mindenekelőtt azonban megadjuk a π prefix-függvény két tulajdonságát, amelyek segítségével, a definició alapján való kiszámításánál, csökkenthetjük a megvizsgálandó esetek számát.

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

Bizonyítás. A 14.9. definíció szerint $\pi(j)$ a $0\,..\,j-1$ egy részhalmazának a maximuma. □

14.11. 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 $\pi(j) \geq 0$ a 14.10. tulajdonság szerint.
– Ha $\pi(j+1) > 0 \Rightarrow$ a prefix-függvény definíciója (14.9) szerint $P_{:(\pi(j+1)-1)+1} = P_{:\pi(j+1)} \sqsupset P_{:j+1} \Rightarrow$ a 14.7. lemmával $P_{:\pi(j+1)-1} \sqsupset P_{:j} \Rightarrow$ újra az 14.9. definícióval $\pi(j+1) - 1 \leq \pi(j) \Rightarrow \pi(j+1) \leq \pi(j) + 1$. □

14.12. Példa. *Kiszámítjuk a π függvényt a 14.9. definíció, a 14.10. tulajdonság és a 14.11. 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*

A 14.12. példa magyarázata:

Minden esetben $\pi(1) = 0$, ui. 14.10. szerint $\pi(1) \in 0\,..\,1-1$

14.11. szerint a π függvény legfeljebb egyesével nő. Ezért $\pi(2) \leq 1$, de $\pi(2) \neq 1$ ui. a π prefix-függvény definíciója (14.9) szerint $\pi(2) = 1$ -ből $P_{:1} \sqsupset P_{:2}$ azaz $B \sqsupset BA$ következne. Következésképp $\pi(2) = 0$.

Hasonlóan, $\pi(2) = 0$ és 14.11. szerint $\pi(3) \leq 1$. Mivel $P_{:1} = B \sqsupset BAB = P_{:3}$, ezért a 14.9. definíció szerint $\pi(3) \geq 1$, végül $\pi(3) = 1$.

Ebből $\pi(4) \leq 2$, és mivel $P_{:2} = BA \sqsupset BABA = P_{:4}$, $\pi(4) = 2$. Hasonlóan $\pi(5) = 3$.

Innét $\pi(6) \leq 4$, de $P_{:4} = BABA \not\sqsupset BABABB = P_{:6}$, $P_{:3} = BAB \not\sqsupset BABABB = P_{:6}$, $P_{:2} = BA \not\sqsupset BABABB = P_{:6}$, ellenben $P_{:1} = B \sqsupset BABABB = P_{:6}$. Tehát $P_{:1}$ a $P_{:6}$ *leghosszabb prefix-szuffixe*, amiből $\pi(6) = 1$ adódik.

A fentiekhez hasonlóan kapjuk, hogy $\pi(7) = 2$ és $\pi(8) = 3$.

14.13. Példa. *Alkalmazzuk az eddig tanultakat a következő esetben! A 14.12. Példából már ismert $P_{:8} = P[0\,..\,8) = BABABBAB$ minta előfordulásait keressük a $T_{:18} = T[0\,..\,18) = ABABABABBBABABABBBAB$ szövegben.* (Az algoritmus, a jelöletlen betűkről *illesztés nélkül is tudja,* 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

A táblázatról: $T[i]$ az aktuális betű a $T_{:n}$ szövegben (most $n = 18$). Ha $j < m$, akkor $P[j]$ az aktuális betű a $P_{:m}$ mintában (most $m = 8$). Ha $j = m$, akkor megtaláltuk a P egy előfordulását T-ben: az algoritmusunk invariánsa ui., hogy $s = i–j$ a minta aktuális eltolása $\wedge\, P_{:j} \sqsupseteq T_{:i} \wedge 0 \leq j \leq i \leq n \wedge j \leq m$.

A táblázat 3. sorában $i = j = 0$ -val, azaz $s = i - j = 0$ aktuális eltolással megpróbáltuk a $P[j] = T[i]$ illesztést, ami meghiúsult. Mint most is, $j = 0$ esetén i-t eggyel megnöveljük, hogy a következő lehetséges eltolással próbálkozzunk.

$s = i - j = 1$ aktuális eltolással (a 4. sorban) öt sikeres illesztést hajtottunk végre, közben i-t és j-t párhuzamosan inkrementálva. Ezután $i = 6 \wedge j = 5$, következésképp $P_{:5} \sqsupseteq T_{:6}$. Ekkor a $P[j] = T[i]$, azaz a $P[5] = T[6]$ illesztés meghiúsul, és ezért az $s = i - j = 1$ érvénytelen eltolás. Most a $\pi(5) = 3$ értékre van szükségünk, ami azt jelenti, úgy kell eltolnunk a mintát, hogy a minta új helyzetében (5. sor) az első három betűje maradjon a mostani helyzete ($s = 1$) szerinti $P_{:5}$ alatt, és ezzel együtt $T_{:6}$ alatt. Mivel a minta $s = 1$ eltolása szerint $P_{:5}$ ugyanaz, mint $T_{:6}$ utolsó 5 betűje, és $\pi(5) = 3$ miatt $P_{:3}$ a $P_{:5}$ leghosszabb prefix-szuffixe, $P_{:3} \sqsupseteq T_{:6}$ is automatikusan teljesül, és egyben $s = 3$ a legkisebb eltolás, amivel a mintának a $T_{:6}$ alatt maradó prefixe illeszkedik $T_{:6}$ lehető leghosszabb szuffixére. Ezt az eltolást a $j := \pi(j)$ értékadással tudjuk elérni, hogy utána a $P_{:j} \sqsupseteq T_{:i}$ és a többi invariáns igaz maradjon. Ezután $j = 3$ lesz, $i = 6$ marad, és valóban $s = i - j = 3$ a P minta aktuális eltolása.

$i =$	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

$s = i - j = 3$ aktuális eltolással tehát a minta első 3 betűje automatikusan illeszkedik, így a mintaillesztést a $P[3] = T[6]$ feltételvizsgálattal kezdjük. Most 5 sikeres illesztés következik, közben i-t és j-t párhuzamosan inkrementálva. Ezután $i = 11 \wedge j = 8$, következésképp $P_{:8} \sqsupseteq T_{:11}$, és $s = i - j = 3$ az első érvényes eltolás. Ezt feljegyezzük. Most $P_{:8}$ leghosszabb prefix-szuffixe $P_{:3}$, hiszen $\pi(8) = 3$. Újra $j := \pi(j)$ adja a legkisebb eltolást, ami mellett $P_{:j} \sqsupseteq T_{:i}$ és a többi invariáns igaz marad. Ezután $j = 3$ lesz, $i = 11$ marad, és $s = i - j = 8$ a P minta aktuális eltolása.

$s = i - j = 8$ aktuális eltolással tehát a minta első 3 betűje automatikusan illeszkedik, így a mintaillesztést a $P[3] = T[11]$ feltételvizsgálattal kezdjük. Most 2 sikeres illesztés következik, közben i-t és j-t párhuzamosan inkrementálva. Ezután $i = 13 \wedge j = 5$, következésképp $P_{:5} \sqsupseteq T_{:13}$. Ekkor a $P[j] = T[i]$, azaz a $P[5] = T[13]$ illesztés meghiúsul, és ezért az $s = i - j = 8$ érvénytelen

eltolás. Ismét $j := \pi(j)$ adja a legkisebb eltolást, ami mellett $P_{:j} \sqsupseteq T_{:i}$ és a többi invariáns igaz marad. Ezután $j = 3$ lesz, $i = 13$ marad, és $s = i{-}j = 10$ a P minta aktuális eltolása.

$s = i{-}j = 10$ aktuális eltolással tehát a minta első 3 betűje automatikusan illeszkedik, így a mintaillesztést a $P[3] = T[13]$ feltételvizsgálattal kezdjük. Most 5 sikeres illesztés következik, közben i-t és j-t párhuzamosan inkrementálva. Ezután $i = 18 \wedge j = 8$, következésképp $P_{:8} \sqsupseteq T_{:18}$, és $s = i{-}j = 10$ a második érvényes eltolás. Ezt is feljegyezzük. Most $P_{:8}$ leghosszabb prefix-szuffixe $P_{:3}$, hiszen $\pi(8) = 3$. Újra $j := \pi(j)$ adja a legkisebb eltolást, ami mellett $P_{:j} \sqsupseteq T_{:i}$ és a többi invariáns igaz marad. Ezután $j = 3$ lesz, $i = 18$ marad, és $s = i{-}j = 15$ a P minta aktuális eltolása.

$s = i{-}j = 15$ aktuális eltolással tehát $j = 3 \wedge i = 18$. Most i a $T_{:n} = T_{:18}$ szöveget túlindexelné, vagy ami ugyanaz, $P[j]$-t nincs mivel összehasonlítani, így az algoritmus leáll.

$i =$	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

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

14.3.1. A KMP algoritmus fő eljárása

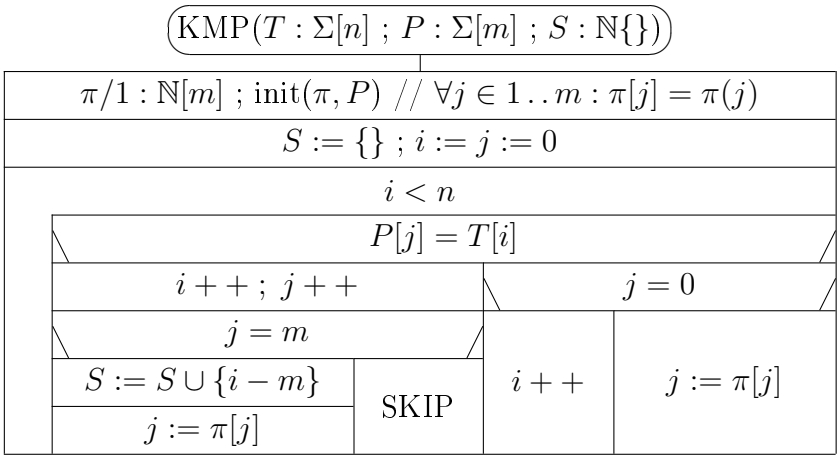
A fenti bevezetésben ismertetett megfontolások alapján megadjuk a KMP algoritmus fő eljárásának struktogramját. Az algoritmust tehát az ott ismertetett invariáns szerint írjuk fel, mindegyik ciklusiterációban egy $P[j] = T[i]$ feltételvizsgálattal és a kapcsolódó műveletekkel. A ciklus feltétele $i < n$, a leállás biztosítása és a T szöveg túlindexelésének elkerülése céljából. A struktogramban a szemléltetésnél használt s eltolást a mindenkori $i{-}j$ különbség reprezentálja.

Vegyük észre, hogy az alábbi struktogramban $j = m$ csak a $j{+}{+}$ utasítás után lehetséges, de ebben az esetben a $j := \pi[j]$ utasítás újra biztosítja, hogy $0 \leq j < m$ legyen. Így a P mintát is mindig helyesen indexeljük.

Először deklarálunk egy $\pi/1 : \mathbb{N}[m]$ tömböt, amit hatékonysági okokból az $\mathrm{init}(\pi, P)$ eljárás segítségével előre feltöltünk a P minta szerinti π függvény értékeivel, és a továbbiakban a tömbben tárolt függvényértékeket használjuk. A ciklus előtt inicializáljuk még a megtalált érvényes eltolások halmazát,

és biztosítjuk *a ciklus globális invariánsát*, ami ezután a ciklus tetszőleges pontján teljesül, kivéve az „i++ ; j++” utasítások között.

A ciklus globális invariánsa:
$i-j$ a minta aktuális eltolása $\wedge\, P_{:j} \sqsupseteq T_{:i} \wedge\, 0 \leq j \leq i \leq n \wedge\, j \leq m$.



A 14.3.2 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 KMP(T, P, S) eljárás megállása és hatékonysága: A 14.3.2. alfejezetben be fogjuk látni, hogy az init(π, P) eljárás $\Theta(m)$ idő alatt lefut. Erre is alapozva itt azt igazoljuk, hogy a KMP(T, P, S) eljárás szintén megáll és műveletigénye $\Theta(n)$.

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 (14.3.2) $\Theta(m)$ és az egyéb inicializálások $\Theta(1)$ 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 invariáns tulajdonsága a fentiek szerint: $0 \leq j \leq i \leq n$.

- 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 az $2i - j$ kifejezést! Az $0 \leq j \leq i \leq n$ invariáns tulajdonság miatt $2i - j = i + (i-j) \in 0+0\,..\,n+n = 0\,..\,2n$. Innét $2i - j \in 0\,..\,2n$.

Az első iteráció előtt $2i - j = 0$, ami mindegyik iterációban (mind a négy programágon) szigorúan monoton nő, és végig $2i - j \leq 2n$, így a ciklus legfeljebb $2n$ iteráció után megáll.

14.3.2. A π prefix tömb inicializálása

A π prefix tömböt az init(π, P) eljárás a π függvény megfelelő értékeivel tölti fel. Utófeltétele tehát: $[\,\forall k \in 1\,..\,m : \pi[k] = \pi(k)\,]$. A π függvény definíciója (14.9) szerint $\pi(j) = \max\{\, h \in 0\,..\,j-1 \mid P_{:h} \sqsupset P_{:j} \,\}$ $\quad (j \in 1\,..\,m)$. Ezért a mintától függetlenül, $\pi(1) = 0$ mindig teljesül. $\pi[1]$ értékét ennek megfelelően inicializáljuk.

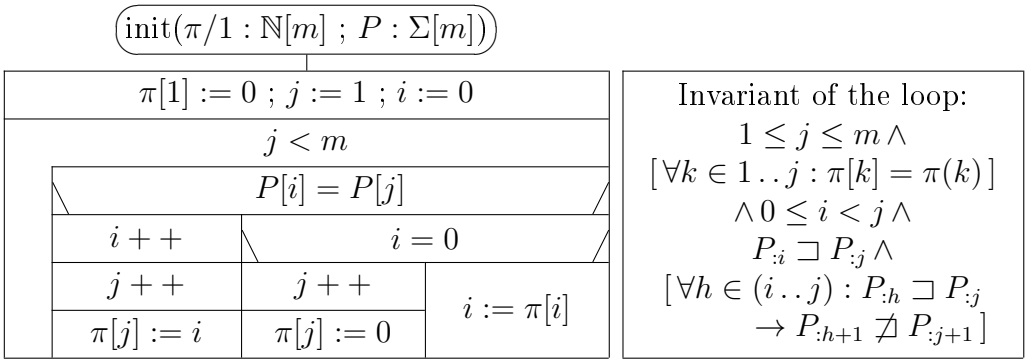
A ciklust az $1 \leq j \leq m \wedge [\,\forall k \in 1\,..\,j : \pi[k] = \pi(k)\,]$ invariáns szerint kívánjuk szervezni, ahol, ha j eléri m-et, teljesül az utófeltétel. Ezért a ciklus előtt elvégezzük még a $j := 1$ inicializálást, amivel ez az invariáns $j = 1$-re már teljesül (ui. már $\pi[1] = 0$); a ciklusfeltétel pedig $j < m$, ami addig igaz, amíg a π tömböt még nem töltöttük fel.

Ha tehát a fenti invariáns adott $j \in [1\,..\,m)$ értékre már teljesül, akkor a következő feladat $\pi(j+1)$ meghatározása, amely majd $\pi[j+1]$ értéke lesz.

Mivel $\pi(j+1) = \max\{\, h \in 0\,..\,j \mid P_{:h} \sqsupset P_{:j+1} \,\}$, két lehetőség van: ez az érték pozitív, vagy nulla. Tegyük fel először, hogy pozitív! Ebben az esetben $\pi(j+1) = \max\{\, i+1 \in 1\,..\,j \mid P_{:i+1} \sqsupset P_{:j+1} \,\}$. Ez azt jelenti, hogy ezzel a feltételezéssel a $P_{:j+1}$ leghosszabb $P_{:i+1}$ prefix-szuffixét keressük.

A **14.7. Lemma** szerint $P_{:i+1} \sqsupset P_{:j+1} \iff P_{:i} \sqsupset P_{:j} \wedge P[i] = P[j]$ (szuffixkiterjesztési lemma). Ez alapján a $P_{:j}$ leghosszabb $P_{:i}$ prefix-szuffixét keressük, amelyre $P[i] = P[j]$. Ha ilyet nem találunk, akkor $\pi(j+1) = 0$.

A keresést praktikusan a $P_{:j}$ leghosszabb $P_{:i}$ prefix-szuffixével kezdjük, ahogy az alábbi kódból is látható, és sorban az egyre rövidebbekkel próbálkozunk.



Ha egy prefix-szuffixnél $P[i] = P[j]$ igaznak bizonyul, akkor a 14.7. Lemma és a π prefix-függvény definíciója alapján $\pi(j+1) = i+1$, amit a bal oldali

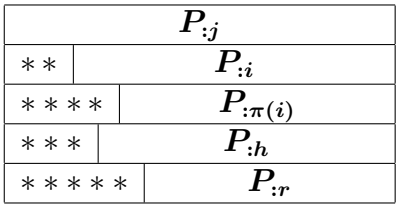
programágon könyvelünk. Ennek hatására a megnövelt i és j értékekre $i = \pi(j)$ teljesül. Így a következő ciklusiterációban a további keresést az új $P_{:j}$ leghosszabb $P_{:i}$ prefix-szuffixével kezdjük.

Ha egy prefix-szuffixnél $P[i] \neq P[j]$, és még $i > 0$, azaz az aktuális $P_{:i}$ prefix-szuffix még nemüres, akkor a jobb oldali programág szerint a $P_{:j}$ következő leghosszabb $P_{:i}$ prefix-szuffixével próbálkozunk. Ha pedig már $i = 0$, akkor már kipróbáltuk a $P_{:j}$ összes $P_{:i}$ prefix-szuffixét, de egyikre sem teljesült $P[i] = P[j]$, ami azt jelenti, hogy $\pi(j+1) = 0$. Ezt a középső programágon könyveljük. Így a következő ciklusiterációban a további keresést ismét az új $P_{:j}$ leghosszabb $P_{:i}$ prefix-szuffixével kezdjük, ami most az üres sztring.

Még hátra van, hogy $i > 0$ esetén a jobb oldali programágon az $i := \pi[i]$ értékadás pontosan adja-e a $P_{:j}$ következő leghosszabb $P_{:i}$ prefix-szuffixének a hosszát. Erre vonatkozik a következő lemma.

14.14. Lemma. *Ha $P_{:i}$ a $P_{:j}$-nek egy nemüres prefix-szuffixe (tehát $0 < i < j$), akkor $P_{:\pi(i)}$ a $P_{:j}$-nek a $P_{:i}$ után következő leghosszabb prefix-szuffixe.*

Bizonyítás. Egyrészt, a π függvény definíciója szerint $P_{:\pi(i)}$ a $P_{:i}$ leghosszabb prefix-szuffixe. Másrészt, az aktuális $P_{:i}$ a $P_{:j}$ prefix-szuffixe. Így $P_{:\pi(i)}$ a $P_{:j}$-nek is szuffixe, mint az alábbi ábra mutatja. Mivel $\pi(i) < i < j$, $P_{:\pi(i)}$ a $P_{:j}$-nek a $P_{:i}$-nél rövidebb prefix-szuffixe. Továbbá a $P_{:j}$-nek a $P_{:i}$ után következő leghosszabb prefix-szuffixe a $P_{:i}$-nek is prefix-szuffixe, így nem lehet $P_{:\pi(i)}$-nél hosszabb, amint az ábrán $P_{:h}$ lenne, és persze rövidebb sem, amint az ábrán $P_{:r}$ lenne. □



A 14.14. Lemma szerint tehát az $i := \pi[i]$ értékadás éppen a $P_{:j}$ következő leghosszabb prefix-szuffixének hosszát adja, feltéve, hogy a kérdéses $\pi[i]$ érték az adott pillanatban már rendelkezésünkre áll, azaz $\pi[i] = \pi(i)$ teljesül. Ez viszont init() eljárás $[\forall k \in 1..j : \pi[k] = \pi(k)] \wedge 0 \leq i < j$ invariáns tulajdonságából és a jobb oldali programág $i \neq 0$ feltételéből azonnal következik.

Az init(π, P) eljárás megállása és hatékonysága: Az alábbiakban igazoljuk, hogy az eljárás műveletigénye $\Theta(m)$.

Mivel a ciklus inicializálása és minden egyes iterációja is $\Theta(1)$ 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. $2j-i = j + (j-i) \wedge j \geq 1 \wedge j-i \geq 1 \Rightarrow 2j-i \geq 2$; 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.

14.3.3. A KMP algoritmus összegzése

Láttuk, hogy a KMP algoritmusnál a 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 algoritmussal 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. Következésképpen, mivel a $T[0\,..\,n)$ szövegre csak a $T[i]$ kifejezésen keresztül hivatkozunk, a szövegben 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 szövegben 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.

14.3.4. 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: ($S = \{\,2; 7\,\} = V$ adódik)

$i = 0$		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	$\underline{A}$

14.3.5. A KMP algoritmus KMP(T, P, S) eljárása parciális helyességének bizonyítása az invariánsok módszerével*

Ebben az alfejezetben matematikai igényességű bizonyítást adunk a KMP(T, P, S) eljárás parciális helyességére.

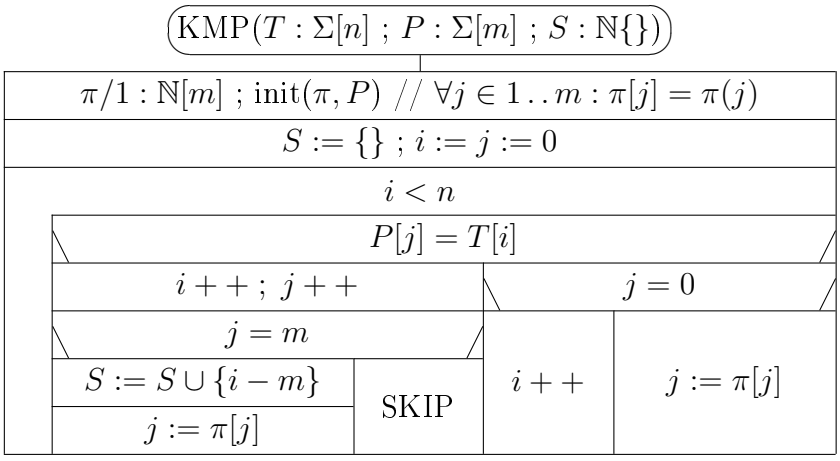
Az eljárás elemzése a ciklusa alábbi 14.18. invariánsán alapszik, ahol $i-j$ a P minta aktuális eltolása a T szövegen. Az invariáns helyességének bizonyításához hasznosak lesznek a következő lemmák. (Az első kettő a sztring szuffix és a valódi szuffix fogalmak közvetlen következménye. A harmadikat bebizonyítjuk.)

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

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

14.17. 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 (14.16) 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 (14.9) alapján. □



14.18. Tétel. *Az* (Inv) *állítás a* KMP(T, P, S) *eljárás ciklusának invariánsa (ami a ciklusfeltétel kiértékelése előtt teljesül).*
(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 init(π, 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 szuffixkiterjesztési lemma (14.7) 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 $j \in 1\,..\,m \wedge P_{:j} \sqsupseteq T_{:i}$, a 14.17. 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 (14.15. 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 szuffixkiterjesztési lemma (14.7) 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) $\quad P_{:j} \sqsupseteq T_{:i} \wedge 0 \leq j \leq i < n \wedge j < m \ \wedge S = V \cap [0\,..\,i{-}j]$ 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) $\quad P_{:j} \sqsupseteq T_{:i} \wedge 0 \leq j \leq i \leq n \wedge j < m \ \wedge S = V \cap [0\,..\,i{-}j)$.
  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) $\quad P_{:j} \sqsupseteq T_{:i} \wedge 0 < j \leq i \leq n \wedge j \leq m \ \wedge S = V \cap [0\,..\,i{-}j]$.
  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*:

14.19. Tétel. *Ha a KMP algoritmus megáll, akkor megoldja a 14.3. (mintaillesztési) problémát, azaz $S = V$ teljesül, amikor a* KMP(T, P, S) *eljárás befejeződik.*

Bizonyítás. A KMP(T, P, S) eljárás ciklusának (Inv) invariánsa (14.18), 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. □

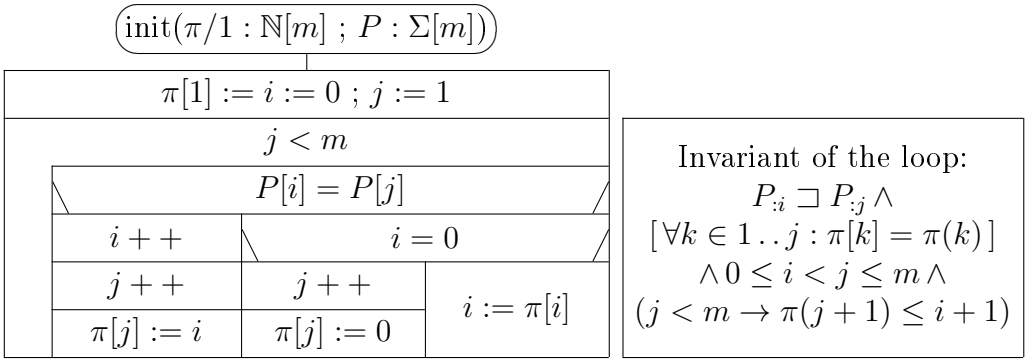
14.3.6. Az init(π, P) eljárás parciális helyességének bizonyítása az invariánsok módszerével*

Ebben az alfejezetben matematikai igényességű bizonyítást adunk az init(π, P) eljárás parciális helyességére.

A következő, a π prefix *függvényre* vonatkozó lemma hasznos lesz.

14.20. 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 az alábbi, 14.21. tételben található ciklusinvariánsán alapszik.

14.21. Tétel. *Az* (inv) *állítás az* init(π, P) *eljárás ciklusának invariánsa.*

(inv) $P_{:i} \sqsupset P_{:j} \wedge (\,\forall k \in 1\,..\,j : \pi[k] = \pi(k)\,) \wedge$
$0 \le i < j \le m \wedge (\,j < m \to \pi(j+1) \le i+1\,).$

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 nemüres sztringnek valódi szuffixe, továbbá a 14.10. tulajdonság szerint $\pi(1) = 0$, valamint a P minta m mérete nem nulla, és végül a 14.11. 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) $P_{:i} \sqsupset P_{:j} \wedge (\,\forall k \in 1\,..\,j : \pi[k] = \pi(k)\,) \wedge$
$0 \le i < j < m \wedge \pi(j+1) \le i+1.$

1. Ha $P[i] = P[j]$, akkor a a 14.7. lemma és (inv1)-ből $P_{:i} \sqsupset P_{:j}$ alapján $P_{:i+1} \sqsupset P_{:j+1}$. Innét a π prefix-függvény definíciója (14.9) 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 $j < m$, a 14.11. 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 a 14.7. lemma alapján $P_{:i+1} \not\sqsupset P_{:j+1}$. Innét a π prefix-függvény definíciója (14.9) 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) $P_{:i} \sqsupset P_{:j} \wedge (\,\forall k \in 1\,..\,j : \pi[k] = \pi(k)\,) \wedge$
$0 \le i < j < m \wedge \pi(j+1) \le i.$

   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 $j < m$, a 14.11. 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) $P_{:i} \sqsupset P_{:j} \wedge (\,\forall k \in 1\,..\,j : \pi[k] = \pi(k)\,) \wedge$
$0 < i < j < m \wedge \pi(j+1) \leq i.$
Mivel $i > 0$, a 14.20. 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 (14.9) $P_{:\pi[i]} \sqsupset P_{:i}$ adódik. Ebből az (inv3) a $P_{:i} \sqsupset P_{:j}$ állításával együtt, a 14.15. 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 utolsó ágának a végén is teljesül az (inv) állítás.

□

Az init(π, P) eljárás parciális helyessége*:

14.22. 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 a 14.21. 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. □

A KMP algoritmus parciális helyessége bizonyításának összegzése*:
A fentiekben egy *viszonylag* egyszerű és rövid bizonyítást sikerült adnunk a lineáris műveletigényű KMP algoritmus helyességére és hatékonyságára. (Érdemes összehasonlítani a CLRS könyvben [3] 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.

15. Információtömörítés ([5])

Ebben a jegyzetben csak *veszteségmentes* (lossless) tömörítésekkel foglakozunk, ahol a tömörített fájlból pontosan rekonstruálható az eredeti bemenet. (Ilyen pl. a ZIP, ARJ, PNG, FLAC, RAR és 7z tömörítés.) Meg kell itt jegyeznünk, hogy pl. egy kép- vagy hangfájl esetében már a digitalizálás is óhatatlanul információvesztéssel jár.

Nem foglakozunk tehát a veszteséges (lossy) tömörítésekkel, mint pl. az MP3, MP4, JPG stb. Ez utóbbiak elhagyják azokat a részleteket, amiket a tervezők szerint az átlagember már valószínűleg nem érzékel (kivéve például, amikor egy képet nagyítunk); és ezen az áron drasztikusan csökkentik a fájlméretet.

A tágyalt módszereknél mindenütt feltesszük, hogy a bemenet egy véges, nemüres $\Sigma = \langle \sigma_1, \sigma_2, \ldots, \sigma_d \rangle$ ábécé feletti szöveg. Ez nem szűkíti le a lehetséges alkalmazások körét, hiszen betűknek akár egy bináris input bájtjait vagy vagy egyéb, fix számú bitből álló szakaszait is tekinthetjük.

15.1. Naiv módszer

A tömörítendő szöveget betűnként, fix hosszúságú bitsorozatokkal kódoljuk.

$\Sigma = \langle \sigma_1, \sigma_2, \ldots, \sigma_d \rangle$ az ábécé $(0 < d < \infty)$.

Egy-egy betű $\lceil \lg d \rceil$ bitet igényel, 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 d-féle különböző kód $\lceil \lg d \rceil$ biten ábrázolható, de eggyel kevesebb biten már nem.

T: $\Sigma\langle\rangle$ a tömörítendő szöveg. $n = |T|$ jelöléssel $n * \lceil \lg d \rceil$ bitben 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-öt kioszthatunk az 5 betűnek.) A tömörített fájl a kódtáblát is tartalmazza.

A fenti ABRAKADABRA szöveg kódtáblája lehet pl. a következő:

betű	kód
A	000
B	001
D	010
K	011
R	100

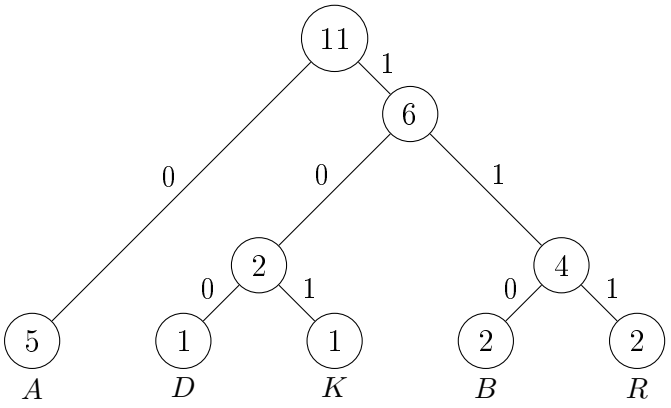
A fenti kódtáblával a tömörített kód a következő lesz:
00000110000001100001000000011000000.

Ez a tömörített fájlba foglalt kódtábla alapján könnyedén 3 bites szakaszokra bontható és kitömöríthető. A kódtábla mérete miatt a gyakorlatban csak hosszabb szövegeket érdemes így tömöríteni.

15.2. Huffman-kód

A tömörítendő szöveget betűnként, változó hosszúságú bitsorozatokkal kódoljuk. A gyakrabban előforduló betűk kódja rövidebb, a ritkábban előfordulóké pedig hosszabb. A betűk kódjait a *kódfa*, ill. a *kódtábla* tartalmazza.

A **kódfa** szigorúan bináris fa. Mindegyik betűhöz tartozik egy-egy levele, amit a betűn kívül annak gyakorisága, azaz előfordulásainak száma is címkéz. A belső csúcsokat a csúcshoz tartozó részfa leveleit címkéző betűk gyakoriságainak összegével címkézzük. (Így a kódfa gyökerét a tömörítendő szöveg hossza címkézi.) Mindegyik bal gyerekbe vezető élt 0-val címkézzük, a jobb gyerekbe vezetőt pedig 1-gyel.



Tetszőleges betű kódja a betűhöz vezető utat címkéző bitek sorozata. A fenti ábra szerinti kódok a következők. 0:A, 100:D, 101:K, 110:B, 111:R.

A Huffman-kód *prefixmentes*. Ez azt jelenti, hogy egyetlen betű kódja sem valamelyik másik betű kódjának prefixe (mivel a betűk a kódfa leveleit címkézik). Ez biztosítja a Huffman-kód egyértelmű kitömörítését.

A betűnkénti kódolású tömörítések között a Huffman-kód hossza minimális. Ugyanahhoz a szöveghez többféle kódfa és hozzá tartozó kódtáblázat építhető, de mindegyik segítségével az input szövegnek ugyanolyan hosszú tömörített kódját kapjuk. A betömörítés a kódtáblával, a kicsomagolás pedig a kódfával történik. Ezért a tömörített fájl a kódfát is tartalmazza.

A **tömörítés a bemenetét kétszer olvassa végig.**

Először meghatározza a szövegben előforduló betűk halmazát és az egyes betűk gyakoriságát.

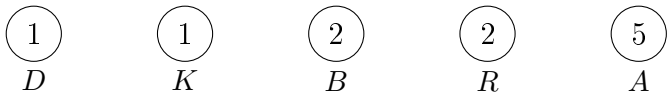
Pl. az *ABRAKADABRA* szöveget egyszer végigolvasva meghatározhatjuk milyen betűk fordulnak elő a szövegben, és milyen gyakorisággal. Úgy képzelhetjük, hogy az alábbi táblázat az új betűkkel folyamatosan bővül, ahogy előrehaladunk a szövegben.

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	

Ennek alapján a tömörítő algoritmus *kódfát*, abból pedig *kódtáblát* épít.

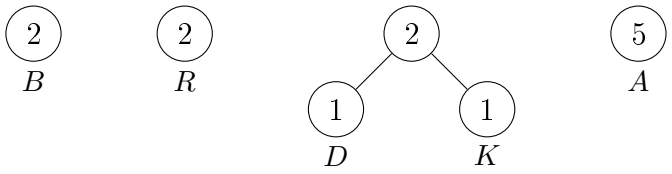
Ezután a tömörítés másodszor is végigolvassa a bemenetet, és a kódtábla alapján sorban kiírja az output fájlba a betűk bináris kódját.

A **kódfát** úgy **építjük** fel, hogy először egycsúcsú fák egy minimum-prioritásos sorát határozzuk meg, amelyben mindegyik betű pontosan egy csúcsot címkéz. A csúcsot a betűn kívül annak gyakorisága is címkézi.



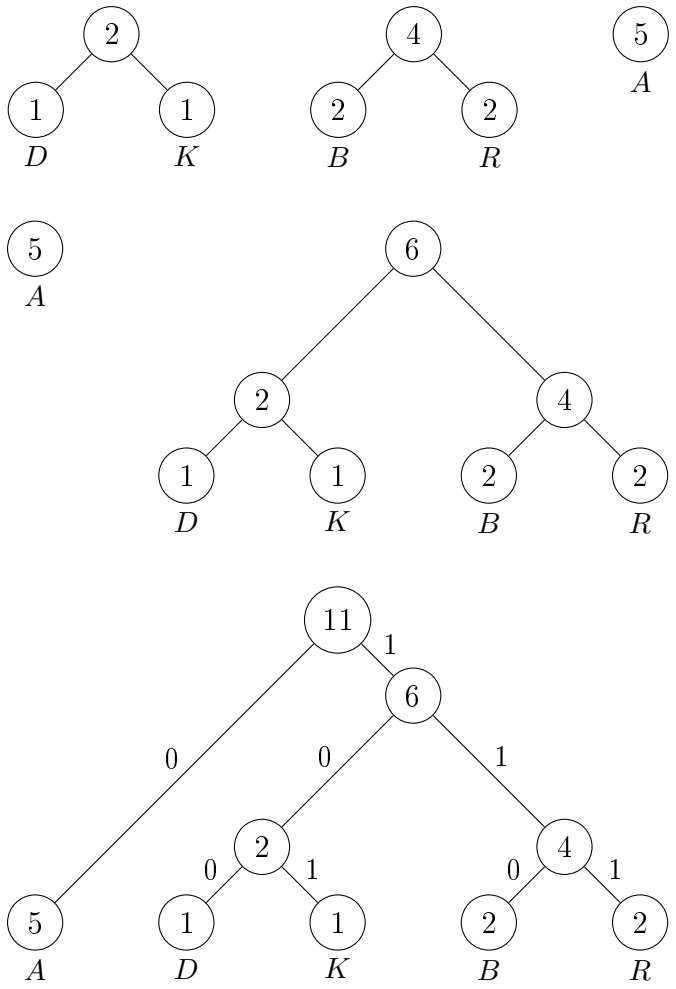
A minimum-prioritásos sort a benne tárolt fák gyökerét címkéző gyakoriság-értékek szerint szervezzük. A prioritásos sornak az egycsúcsú fákkal való feltöltése után a következőt csináljuk ciklusban, amíg a kupac még legalább kettő fából áll.

Kiveszünk a kupacból egy olyan fát, amelynek a 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 sorban az előbb kiválasztott két fa lesz. Az így képzett új fát visszatesszük a minimum-prioritásos sorba.



A kurzuson alkalmazott konvenció szerint, azonos gyakoriság esetén a kisebb magasságú fát részesítjük előnyben. Ha két vagy több fánál ez is egyenlő, akkor ezeket a fák frontját címkéző szavak alfabetikus sorrendje szerint rendezzük.[18] Ez a konvenció ugyan önkényes, de az algoritmus bemutatása szempontjából hasznos egyértelműsítés. A fák ágait is hasonlóan rendezzük sorba.

A ciklust addig ismételjük, amíg már csak egy fánk marad.



A fenti ciklus után a minimum-prioritásos sorban maradó egyetlen bináris fa, az élek fenti címkézésével a Huffman-féle kódfa.

A kódfából ezután kódtáblát készítünk. Mindegyik betűhöz tartozó kódot úgy kapjuk meg, hogy a kódfa gyökerétől elindulva, a betűhöz tartozó levélig

[18] A fa frontja balról jobbra a levelei sorozata. A fa frontját címkéző szót, a leveleket címkéző betűket a levelek sorrendje szerint, balról jobbra összeolvasva kapjuk.

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.)

betű	kód
A	0
B	110
D	100
K	101
R	111

Befejezésül újra végigolvassuk a tömörítendő $ABRAKADABRA$ szöveget, és a kódtáblázat segítségével sorban mindegyik betű bináris kódját a (kezdetben üres) tömörített bitsorozat végéhez fűzzük.

01101110101010001101110

Az $ABRAKADABRA$ szöveg Huffman kódja tehát 23 bit, ami lényegesen rövidebb, mint az előző alfejezetben ismertetett naiv tömörítés esetén.

A tömörített fájl a kódfát is tartalmazza, így a gyakorlatban Huffman-kódolással csak hosszabb szövegeket érdemes tömöríteni.

A **kicsomagolást** is betűnként végezzük. Mindegyik betű kinyeréséhez a kódfa gyökerétől indulunk, majd a tömörített kód sorban olvasott bitjeinek hatására 0 esetén balra, 1 esetén jobbra lépünk lefelé a fában, míg levélcsúcshoz nem érünk. Ekkor kiírjuk a levelet címkéző betűt, majd a Huffman-kódban a következő bittől és újra a kódfa gyökerétől folytatjuk, amíg a tömörített kódon végig nem érünk.

A fenti példánál, 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 111 az R-hez, a 0 az A-hoz, a 101 a K-hoz, a 0 az A-hoz, az 100 a D-hez, a 0 az A-hoz, a 110 a B-hez, a 111 az R-hez, és végül a 0 az A-hoz. Így visszakaptuk az eredeti, tömörítetelen szöveget.

15.1. Feladat. *Próbáljuk ki, hogy ha a Huffman-kódolásban lévő indeterminizmusokat a fenti konvenciótól 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 a magasabb fát vesszük ki előbb – ezt az ad-hoc szabályt az alfabetikus konvenciónál most is erősebbnek véve –, akkor a fenti példában a kódfát felépítő ciklus második iterációjában a* $[\,(D/1)\,\mathbf{2}\,(K/1)\,]$ *és a* $(B/\mathbf{2})$ *fát fogjuk összevonni.)*

15.3. A Lempel–Ziv–Welch (LZW) módszer

Az LZW tömörítés egy heurisztikus algoritmus, különösen mély matematikai háttér nélkül. Ellentétben a Huffman-kódolással, semmiféle optimalitást sem garantál, de a tapasztalatok szerint jól tömörít, átlagosan 30%-kal jobban, mint a betűnkénti prefixmentes kódok között optimális Huffman-kódolás. Ez azért lehetséges, mert a tömörítendő fájlokban hosszú, ismétlődő minták vannak, amiket az LZW algoritmus „megtanul", és rövid kódokkal helyettesít. Ennek ellenére, gyakran a Huffman-kódolás ad jobb eredményt. A közismert ZIP tömörítés az LZW elvén működik.

Összehasonlítva a Huffman-kódolással, ahol az input fix hosszú darabjait változó hosszúságú kódokkal tömörítjük, az LZW algoritmus az input változó hosszúságú darabjait fix hosszúságú kódokkal tömöríti.

Az LZW algoritmus nagy előnye, hogy ismert ábécé esetén, tömörítéskor a bemenetet csak egyszer olvassa végig, így lényegesen gyorsabban tömörít. (A Huffman-kódolásnál, előre adott ábécéhez tartozó szövegek esetén, a tökéletes optimalitás feláldozásával úgy szokták megspórolni az input kétszeri végigolvasását, hogy a be- és kitömörítés a korábbi tömörítésekhez generált kódtáblát, ill. kódfát használja, amiket minden újabb tömörítésnél az aktuális betűgyakoriságok alapján finomítanak.)

Az LZW tömörítés az input szöveget ismétlődő mintákra (változó hosszúságú sztringekre) bontja. Mindegyik mintát ugyanolyan hosszú bináris kóddal helyettesíti. Ezek a minták kódjai. A tömörített fájl csak a kódokat, továbbá a fix kódhosszt és az ábécével kapcsolatos információkat tartalmazza. Ellentétben a naiv és a Huffman-kóddal, nem tartalmazza a kódok és a minták (a másik kettőnél betűk) megfeleltetését. (Igaz, az LZW algoritmus szótára lényegesen nagyobb, mint pl. a Huffman-kódhoz tartozó kódfa.) A kitömörítés rekonstruálja a kódok és a hozzájuk tartozó betű- vagy bájtsztringek közötti megfeleltetést.

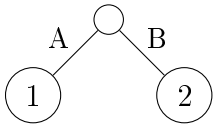
15.3.1. Az LZW (Lempel–Ziv–Welch) tömörítés

Az alábbiakban az LZW tömörítést a $\Sigma = \{A, B\}$ ábácével az $ABAB\,BABA\,BAB\,ABA\,A$ szöveg tömörítésén mutatjuk be, ahol a szóközök nem részei a bemenetnek (csak a könnyebb olvashatóságot szolgálják).

Megjegyezzük, hogy minél kisebb az ábécé, annál könnyebb olyan bemeneteket adni, amelyek elég rövidek a bemutatáshoz, de ismétlődő mintákat tartalmaznak. Ezért az LZW algoritmust gyakran $2-4$ betűs ábécével szokták bemutatni. A gyakorlatban ezzel szemben akár bináris fájlokat is, pl. kép- vagy hangfájlokat is veszteségmentesen tömöríthetünk vele. A betűknek megfeleltethetjük pl. a bájtokat, ahol minden bájt önmagát kódolja, de a

kódok általában $11-16$ bitesek, viszont a kódhosszt még a tömörítés kezdete előtt fixálni kell. A kódhossz megállapításának problémájával itt nem foglalkozunk, csak jelezzük, hogy a hosszabb kódok exponenciálisan több minta megkülönböztetését teszik lehetővé, ami csökkentheti, de persze növelheti is a tömörített fájl méretét.

A tömörítés során a bemenetet változó hosszúságú *szavakra* (sztringekre) bontjuk. (Ezek a „szavak" nem felelnek meg sem a fenti tagolásnak, sem a természetes szavaknak.) Egy szótár segítségével mindegyik szónak, mint kulcsnak, egy pozitív egész számkódot, mint értéket feleltetünk meg[19]. Az alábbiakban a szótárat (a szótárban való keresések elkerülése érdekében) egy ún. *szófa* segítségével reprezentáljuk. A szófa kezdetben az ábécé betűinek kódjait, pontosabban a lehetséges egybetűs szavak kódjait tartalmazza. A kezdeti kódokat az ábécé betűsorrendjének megfelelően osztjuk ki.



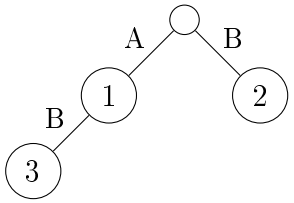
A kezdeti szófa csak a gyökércsúcsból és a gyerekeiből áll. Az éleket az ábécé betűi, a gyökércsúcs gyerekeit pedig a megfelelő kódok címkézik. A későbbiekben a gyökér kivételével mindegyik csúcs alá újabbak is kerülhetnek. Mindegyik újabb csúcsot sorban az első, még szabad kód címkézi. Ez a kód gyökérből a csúcshoz vezető út mentén, az éleket címkéző betűkből összeolvasott szó kódja. Tetszőleges csúcsnak legfeljebb annyi gyereke lehet, amennyi az ábécé mérete. A gyerekeihez vezető élek betűcímkéi egyediek. Az előbbiek szerint a szótárat reprezentáló szófában a tárolt szavak, mint kulcsok, és a kódok, mint értékek, bijektíven megfelelnek egymásnak.

Az alábbiakban leírt lépések addig folytatódnak, amíg a tömörítendő szöveg még tömörítetlen maradéka üres nem lesz.

A tömörítés első lépéseként az $ABAB\,BABA\,BAB\,ABA\,A$ szöveg első betűjéből álló szónak (A), a szótár (szófa) segítségével megfeleltetjük az 1 kódot, de az első két betűből álló AB szó még nincs a szótárban. Ezért az A kódját kiírjuk, az AB kódját pedig felvesszük a szótárba.

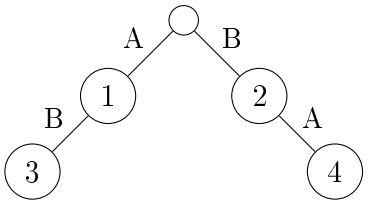
Bemenet:	A	$BAB\,BABA\,BAB\,ABA\,A$
Kimenet:	1	

[19] A kódok nemnegatív egész számok is lehetnének.



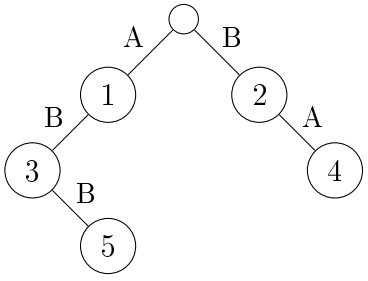
A második lépésben a még tömörítetlen $BAB\,BABA\,BAB\,ABA\,A$ maradék szöveg első betűjéből álló szónak (B), a szótár (szófa) segítségével megfeleltetjük az 2 kódot, de az első két betűből álló BA szó még nincs a szótárban. Ezért a B kódját kiírjuk, a BA kódját pedig felvesszük a szótárba.

Bemenet:	A	B	$AB\,BABA\,BAB\,ABA\,A$
Kimenet:	1	2	



A harmadik lépésben a még tömörítetlen $AB\,BABA\,BAB\,ABA\,A$ maradék szöveg első két betűjéből álló szónak (AB), a szótár (szófa) segítségével megfeleltetjük a 3 kódot, de az első három betűből álló ABB szó még nincs a szótárban. Ezért az AB kódját kiírjuk, az ABB kódját pedig felvesszük a szótárba.

Bemenet:	A	B	AB	$BABA\,BAB\,ABA\,A$
Kimenet:	1	2	3	



Általánosságban minden lépésben a maradék szöveg leghosszabb olyan kezdőszeletét választjuk le, amely megtalálható a szótárban. Mivel a szótárban

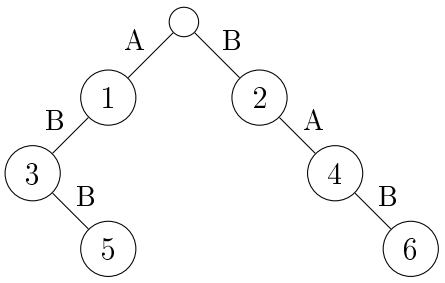
az új szavakat a régiek végéhez egy-egy betűt hozzátoldva kapjuk, minden szótárbeli szónak minden nemüres kezdőszelete (prefixe) is benne van a szótárban. (Ezért a szótárunkat reprezentáló szófát prefix-fának is szokás nevezni.)

Mivel pedig a szótárat szófával reprezentáljuk, a maradék szöveg leghosszabb, a szótárban megtalálható S kezdőszeletét úgy kapjuk, hogy az S sztringet a maradék szöveg első betűjével inicializáljuk, miközben a maradék szöveg elejéről leválasztjuk ezt a betűt, és a fában a gyökérről a betűnek megfelelő gyerekére lépünk; majd ciklusban addig megyünk tovább, amíg a maradék szöveg el nem fogy, és az első betűjére, amit jelöljön most c, a szófa a aktuális csúcsának van olyan g gyereke, hogy az abba vezető élt a c betű címkézi. Amíg a ciklusfeltétel teljesül, a ciklusmagban végrehajtjuk az $S := S{+}c; a := g$ értékadásokat, és a maradék szöveg elejéről leválasztjuk a c betűt.

A ciklus befejezése után pedig kiírjuk az a aktuális csúcsot címkéző kódot. Amennyiben a ciklusunk úgy állt le, hogy a maradék szöveg még nem üres, akkor az első betűjét c-vel jelölve, az a aktuális csúcsnak nincs még olyan gyereke, amelyre az oda vezető élet a c betű címkézné. Ilyenkor a szófában az a aktuális csúcs alá létrehozunk egy új h gyerekcsúcsot, az (a, h) élt a c betűvel, a h csúcsot pedig a legkisebb, még fel nem használt kóddal címkézzük. Ilyen módon, a szótárhoz hozzávettük az $S{+}c$ szót, az éppen most felhasznált kóddal együtt.

A negyedik lépésben a még tömörítetlen $BABA\,BAB\,ABA\,A$ maradék szöveg első két betűjéből álló szónak (BA), a szótár (szófa) segítségével megfeleltetjük a 4 kódot, de az első három betűből álló BAB szó még nincs a szótárban. Ezért a BA kódját kiírjuk, a BAB kódját pedig felvesszük a szótárba.

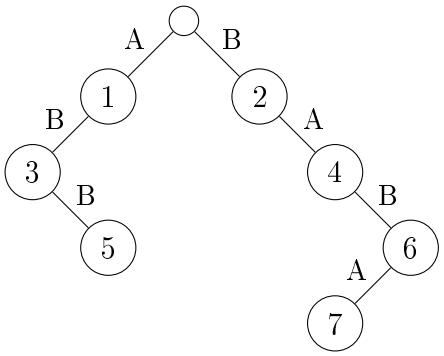
Bemenet:	A	B	AB	BA	$BA\,BAB\,ABA\,A$
Kimenet:	1	2	3	4	



Az ötödik lépésben a még tömörítetlen $BA\,BAB\,ABA\,A$ maradék szöveg első három betűjéből álló szónak (BAB), a szótár (szófa) segítségével meg-

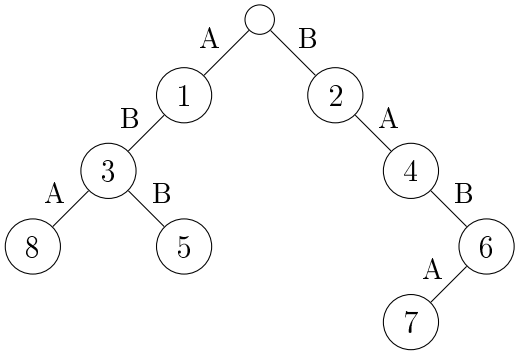
feleltetjük a 6 kódot, de az első négy betűből álló $BABA$ szó még nincs a szótárban. Ezért a BAB kódját kiírjuk, a $BABA$ kódját pedig felvesszük a szótárba.

Bemenet:	A	B	AB	BA	BAB	$AB\,ABA\,A$
Kimenet:	1	2	3	4	6	



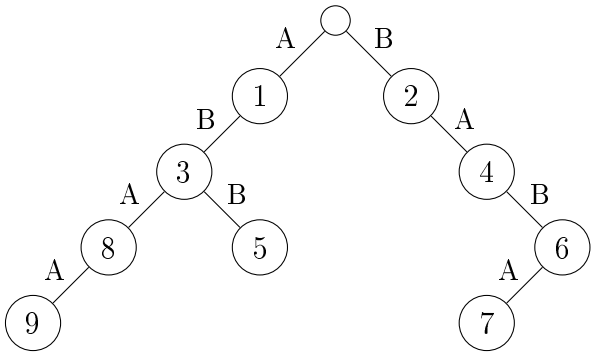
A hatodik lépésben a még tömörítetlen $AB\,ABA\,A$ maradék szöveg első két betűjéből álló szónak (AB), a szótár (szófa) segítségével megfeleltetjük a 3 kódot, de az első három betűből álló ABA szó még nincs a szótárban. Ezért az AB kódját kiírjuk, a ABA kódját pedig felvesszük a szótárba.

Bemenet:	A	B	AB	BA	BAB	AB	$ABA\,A$
Kimenet:	1	2	3	4	6	3	



A hetedik lépésben a még tömörítetlen $ABA\,A$ maradék szöveg első három betűjéből álló szónak (ABA), a szótár (szófa) segítségével megfeleltetjük a 8 kódot, de az első négy betűből álló $ABAA$ szó még nincs a szótárban. Ezért az ABA kódját kiírjuk, az $ABAA$ kódját pedig felvesszük a szótárba.

Bemenet:	A	B	AB	BA	BAB	AB	ABA	A
Kimenet:	1	2	3	4	6	3	8	



A nyolcadik lépésben a még tömörítetlen A maradék szöveg egyetlen betűjéből álló szónak (A), a szótár (szófa) segítségével megfeleltetjük a 1 kódot, és ezzel a maradék szöveg üres. Ezért az A kódját kiírjuk, és befejezzük a tönörítést.

Bemenet:	A	B	AB	BA	BAB	AB	ABA	A
Kimenet:	1	2	3	4	6	3	8	1

A fenti példában a legnagyobb kód a 9. Mivel a kódok 4 biten 15-ig ábrázolhatók, az LZW algoritmus fenti letjátszása legalább 4 bites kódokkal változatlanul működik. 4 bites kódokkal a kimenet 8*4=32 bitet igényel.

Érdekes lehet megfontolni, mi történne, ha csak 3 bites kódokat használnánk. Ekkor a 8-as és a 9-es kódot már nem tudnánk generálni, ezért az új kódok generálása, és ezzel együtt a szótár bővítése a 7-es kód után leállna, azaz az ötödik lépésben kialakított szótárral dolgoznánk tovább. Ezután a maradék $AB\,ABA\,A$ szöveget a $3, 3, 1, 1$ kódokkal tudnánk tömöríteni, és mivel az ötödik lépéssel bezárólag 5 kódot generáltunk, a kimenet 9*3=27 bitet igényelne.

Ha viszont a kódokat nem 1-től, hanem 0-tól generáltuk volna, mindegyik kód eggyel kisebb lett volna, és a kódok generálása, ezzel együtt a szótár (szófa) építése 3 bites kódokkal is csak a hatodik lépés után állt volna le. A hetedik lépésben tehát az LZW tömörítés ABA-nak a 7 kódot találta volna, és nem építette volna tovább a szótárat, majd a nyolcadik lépésben a maradék A szöveget a 0 kóddal tömörítettük volna. Ebben az esetben tehát a kimenet 8*3=24 bitet igényelt volna.

A tömörített fájlt tekintve mindegyik most kiszámolt teljes kódhosszhoz hozzájön az az információ, amely az ábécére, valamint az egyes kódok fix hosszára vonatkozik. A szótárat viszont a kitömörítés rekonstruálja, így azt a tömörített kód nem tartalmazza.

15.3.2. Az LZW (Lempel–Ziv–Welch) kitömörítés

A kitömörítés vagy kicsomagolás algoritmusa nagy vonalakban a betömörítést követi. Itt azonban a szótár kulcsai a kódok, így a szótár szerkezete sokkal egyszerűbb: sztringek vektorának tekinthető, amit a kódokkal indexelünk.

Az algoritmust az alábbi táblázatok szemléltetik, ahol az első sor a bemenet, a tömörített kódok sorozata. A második a kimenet, a kitömörített szöveg, a bemenet szerint tagolva, a harmadik és a negyedik a D szótár (Dictionary). A harmadik sor szigorúan monoton növekvő sorrendben tartalmazza az összes (adott pillanatig) generált kódot (a szótár kulcsait, azaz indexeit), a negyedik pedig a kódokhoz tartozó szavakat.

A kezdőállapot a következő:

Bemenet:				1, 2, 3, 4, 6, 3, 8, 1
Kimenet:				
Szótár	kód	1	2	
	szó	A	B	

Ezután elkezdhetjük feldolgozni a bemenetet. Már a kezdeti szótár tartalmazza az első két kódhoz tartozó szavakat, így megvan a kimenet első két betűje. Tudjuk, hogy a betömörítés az A-t megtalálta a szótárban, de az AB-t nem, ezért az 1-es kódot írta ki, majd felvette a szótárba az AB-t 3-as kóddal.

Bemenet:				1	2	3, 4, 6, 3, 8, 1
Kimenet:				A	B	
Szótár	kód	1	2	3		
	szó	A	B	AB		

Hasonlóan mehetünk tovább egészen a négyes kódhoz tartozó szó kiírásáig. A fenti szótárból már ki tudjuk írni a bemeneten következő 3-as kódhoz tartozó AB kimenetet. Tudjuk, hogy a betömörítés a B-t megtalálta a szótárban, de a BA szót nem, ezért a 2-es kódot írta ki, majd felvette a szótárba a BA szót 4-es kóddal.

Ezután már ki tudjuk írni a bemeneten következő 4-es kódhoz tartozó BA kimenetet. Tudjuk, hogy a betömörítés az AB szót megtalálta a szótárban, de a ABB szót nem, ezért a 3-as kódot írta ki, majd felvette a szótárba a ABB szót 5-ös kóddal.

Bemenet:				1	2	3	4	6, 3, 8, 1
Kimenet:				A	B	AB	BA	
Szó-	kód	1	2	3	4	5		
tár	szó	A	B	AB	BA	ABB		

Most viszont gondban vagyunk, mert a bemeneten a 6-os kód következik, ami még nem szerepel a szótárban. A betömörítésnél ui. a 4-es kód kiírásával egy lépésben már felvettük a szótárba a 6-os kódot, amit rögtön utána a kimeneten is felhasználtunk. A kitömörítésnél viszont a szótárgenerálás egy lépéssel a kimenet generálása után kullog. Ezért, ha a bemeneten ismeretlen kód jön, az mindig eggyel nagyobb, mint az utolsó, már ismert kód.

Bemenet:				1	2	3	4	6	3, 8, 1
Kimenet:				A	B	AB	BA	?	
Szó-	kód	1	2	3	4	5	6		
tár	szó	A	B	AB	BA	ABB	?		

Szerencsére tudjuk, hogy a betömörítés az BA szót megtalálta a szótárban, de a BAu szót nem, ahol u (unknown) jelöli a BA szó után a betömörítés bemenetén következő ismeretlen betűt. Ezért a 4-es kódot írta ki, majd felvette a szótárba a BAu szót 6-os kóddal. A kitömörítés bemenetén viszont most a 6-os kód következik, amihez a BAu szó szartozik, a végén az ismeretlen u betűvel.

Bemenet:				1	2	3	4	6	3, 8, 1
Kimenet:				A	B	AB	BA	BAu	
Szó-	kód	1	2	3	4	5	6		
tár	szó	A	B	AB	BA	ABB	BAu		

Már csak a fenti táblázatunkra kell ránéznünk. Tudjuk, hogy a betömörítés a BA szót megtalálta a szótárban, de a BAu, azaz a BAB szót nem, ezért $u = B$.

Bemenet:				1	2	3	4	6	3, 8, 1
Kimenet:				A	B	AB	BA	BAB	
Szó-	kód	1	2	3	4	5	6		
tár	szó	A	B	AB	BA	ABB	BAB		

A fentieket általánosítva, amikor a betömörítés egy szót megtalál a szótárban, a hozzá tartozó k kódot vagy az előző lépésben generálta, vagy korábban. Ha

korábban generálta, akkor a k kódhoz tartozó szó már a kitömörítés aktuális lépéséhez is rendelkezésre áll.

Ha viszont a betömörítés a k kódot az előző lépésben generálta, akkor a kitömörítés aktuális lépése idején a k kódhoz tartozó szó még nem áll rendelkezésünkre, hiszen éppen ekkor kellene az előző lépésben meghatározott teljes szó és az aktuális lépésben kikeresett szó első betűjének konkatenálásával kiszámítani. Ez utóbbi esetben azonban, ha a kitömörítéskor az előző lépésben kiírt S szót tekintjük, akkor a betömörítés előző lépésében generált k kódhoz tartozó szó Su alakú, és a kitömörítés aktuális lépése idején, a k kódhoz tartozó szó szintén Su. De akkor az u betű éppen az Su szó első betűje, vagy ami ugyanaz, a nemüres S szó első betűje. Eszerint a k kódhoz tartozó szó az $S + S_1$, ahol az S_1 a kitömörítés előző lépése által kiírt S szó első betűje.

A kitömörítés bemenetén a következő kód a 3-as, amihez az AB szó tartozik (ezt írjuk ki), így pedig a 7-es kódhoz a $BABA$ kerül a szótárba.

Bemenet:				1	2	3	4	6	3	8, 1
Kimenet:				A	B	AB	BA	BAB	AB	
Szó-	kód	1	2	3	4	5	6	7		
tár	szó	A	B	AB	BA	ABB	BAB	$BABA$		

A kitömörítés bemenetén a következő kód a 8-as, amihez a szótárban még nincs szó. Az előzőleg kiírt szó az AB, ezért a 8-as kódhoz az ABA tartozik a szótárban, és a kimeneten is.

Bemenet:				1	2	3	4	6	3	8	1
Kimenet:				A	B	AB	BA	BAB	AB	ABA	
Szó-	kód	1	2	3	4	5	6	7	8		
tár	szó	A	B	AB	BA	ABB	BAB	$BABA$	ABA		

A bemeneten az utolsó kód az 1-es. Ehhez a szótár alapján az A egyebetűs szót írjuk ki. Következésképpen a szótárba a 9-es kódhoz az $ABAA$ szó kerül.

Bemenet:				1	2	3	4	6	3	8	1
Kimenet:				A	B	AB	BA	BAB	AB	ABA	A
Szó-	kód	1	2	3	4	5	6	7	8	9	
tár	szó	A	B	AB	BA	ABB	BAB	$BABA$	ABA	$ABAA$	

A bemenet ezzel elfogyott, a kimeneten pedig megjelent az eredeti, tömörítetlen szöveg. A kitömörítést befejeztük.

A szótár a kitömörítésnél is ugyanannyi bejegyzést tartalmaz, mint a betömörítésnél, csak itt a kódok a kulcsok. A fix kódhossz miatt, amennyiben a kódok b bitesek, akkor $MAXCODE = 2^b - 1$ a kódként használható legnagyobb számérték. Ha pl. $b = 12$, akkor $MAXCODE = 2^{12} - 1 = 4095$. A $MAXCODE$ elérése után a szótárépítés megszűnik, a be- és a kitömörítés is az addig generált szótárral dolgozik tovább.[20]

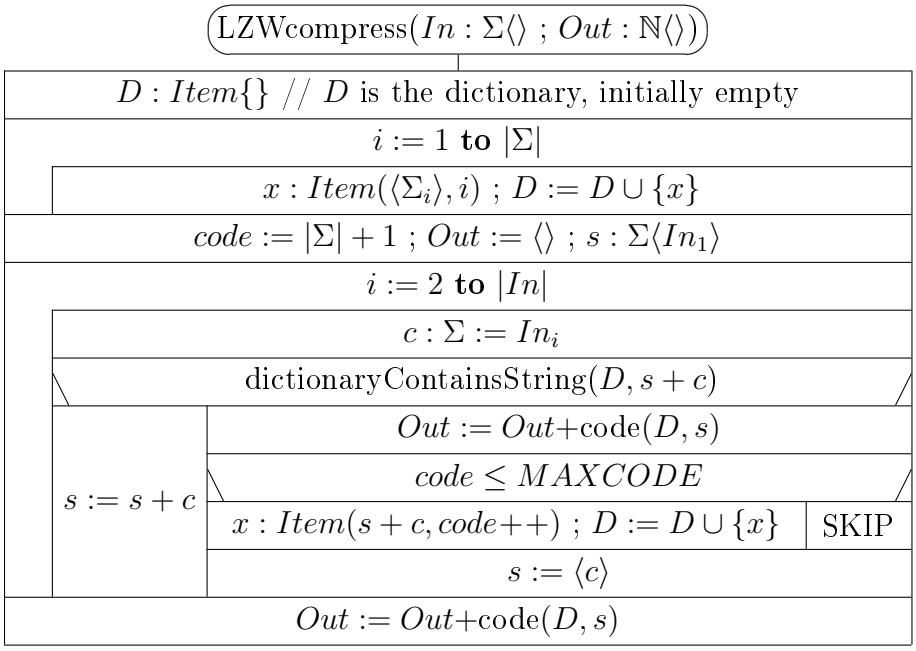
15.3.3. Az LZW algoritmus struktogramjai*

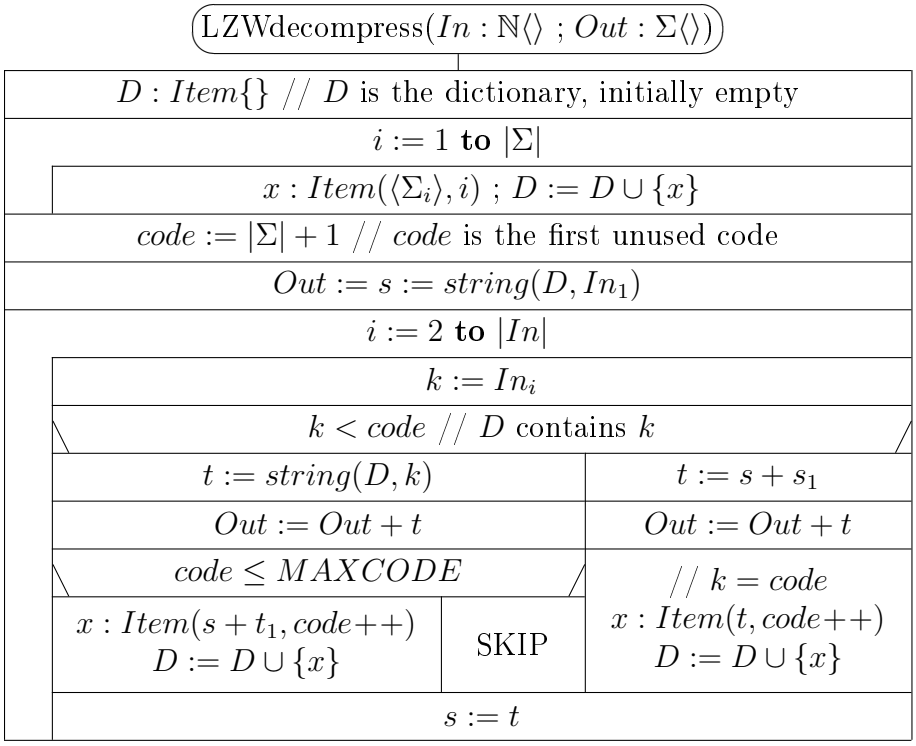
Az alábbiakban az LZW algoritmus összefoglalásaként megadjuk a be- és a kitömörítés *absztrakt struktogramját*. A szótárnak a fentiekben vázolt, a be- és a kitömörítésnél különböző megvalósítását melőzzük. Egységesen rendezett párok halmazának tekintjük, amire adottnak vesszük az eljárásokban használt, beszélő nevekkel ellátott műveleteket. A kódokat egytől sorszámozzuk és a sorozatokat is egytől indexeljük. Használjuk a 8. fejezetben kialakított jelöléseket.

Jelölések az absztrakt struktogramokhoz:
- $MAXCODE = 2^b - 1$ (globális konstans) a kódként használható legnagyobb számérték.
- A $\Sigma = \langle \sigma_1, \sigma_2, \ldots, \sigma_d \rangle$ sorozat tartalmazza az ábécé betűit.
- 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.

$Item$
$+string : \Sigma\langle\rangle$ $+code : \mathbb{N}$
$+Item(s : \Sigma\langle\rangle \ ; \ k : \mathbb{N})\{string := s \ ; \ code := k\}$

[20] Alternatív stratégia pl., hogy ilyenkor (az ábécé alapján) újrainicializálják szótárat, és a továbbiakban ezzel dolgoznak tovább.


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
$t := string(D, k)$
$t := s + s_1$
$Out := Out + t$
$Out := Out + t$
$code \leq MAXCODE$
$x : Item(s + t_1, code{+}{+})$
$D := D \cup \{x\}$
SKIP
// $k = code$
$x : Item(t, code{+}{+})$
$D := D \cup \{x\}$
$s := t$