Algoritmusok és adatszerkezetek II.

A jegyzetet *dr. Ásványi Tibor* előadásain *Lanka Máté* készítette.

Tartalomjegyzék

Rendezés lineáris időben

Bucket sort (edényrendezés)

Elég komoly előfeltételezéseket kell tennünk a kulcsokról, azaz kulcsok ∈ [0,1). (n db kulcs), n…k, ez benne lesz a [0,n) intervallumban. Az ⌊n … k⌋ ∈ 0 … (n − 1).

Ha a kulcsok vagy számok (egész / valós), vagy könnyű azokat átkonvertálni számokra, akkor könnyű berakni ebbe az intervallumra. Ha tudunk alsó és felső korlátot mondani, akkor ez megoldható.

Ez a rendezés a következőképpen zajlik (példa).

<0,32; 0,57; 0,83; 0,12; 0,93>

Öt darab adat van, azaz 5 db bucket van, egy 0., 1., 2., 3., és egy 4..

Mindegyik kulcsot felszorozzuk az elemek számával. Azaz a 0.32*5. Ennek alsó egészrésze 1. Majd a 0,57*5, aminek alsó egészrésze 2. Ezt rendezgetjük egész addig, míg az utolsó elemig nem érünk.

0.	0,12
1.	0,32
2.	0,57
3.	
4.	0,83; 0.93

Ezek után rendezzük az „edényeket”, majd összefűzzük, és az egész végül rendezve is lesz.

Ennek ez a rendezett sorozat lesz az eredménye: <0,12; 0,32; 0,57; 0,83; 0,93>.

BUCKET_SORT(&L)

n := az L lista hossza
B[0..n−1] létrehozása
B[0..n−1] := ∅
L ≠ ∅
kivesszük L első elemét
Az elemet n kulcsa (k) szerint befűzzük a megfelelő szálú B[(alsó egészrész:)n…k] lista elejére
i := 0..n−1
B[i] rendezése
B[0], B[1],..,[n−1] listákat összefűzzük L−be

Mekkora ennek a költsége, futási ideje?
Rögtön az elején $\theta(n)$ van, a létrehozás $\theta(1)$, a B tömb elemeinek nullázása és az első ciklus egyaránt $\theta(n)$ mindkettő, majd az összefűzés szintén $\theta(n)$. A rendezéssel vannak már bajok. Ha nagyon egyszerű rendezést veszünk (pl. beszúró), akkor ez $\theta(n^2)$ műveletigényű lesz

$$MT_R(n_i) \in \theta(n_i^2)$$

Az egész rendezésnek pedig a műveletigénye:

$$mT_{BS}(n) \in \theta(n)$$

$$MT_{BS}(n) \in \theta(n^2)$$

$$AT_{BS}(n) \in \theta(n)$$

Ennél vannak komplikáltabb esetek is. Vegyünk egy bonyolultabb, speciális problémát.

A[1..n]: T rendezése.

$\varphi: T \to 0..k \;\; (k \in O(n)$

Az egyszerűség kedvéért vegyünk 4-es számrendszer-beli háromjegyű számokat:

332
013
231
332
033

A φ függvény ezek közül a legelső számjegyeket választja ki (azaz a 3,0,2,3,0 számjegyeket). Vegyünk egy C tömböt, amelyet nullázzunk ki:

	C	1	2	3	4	5	Eredm.:	Szum.:	5	4	3	2	1	
0	0		1			2	2	2	1			0		0
1	0						0	2						2
2	0			1			1	3			2			2
3	0	1			2		2	5		4			3	3

Mit is csináltunk? Szépen elemenként végigmegyünk, és megnézzük, hogy milyen elemmel kezdődik, majd ezeket folyamatosan számoljuk, eredménynek végül megkapjuk, hogy 0-ból és 3-ból kettő eset van, míg 2-ből egy eset van.

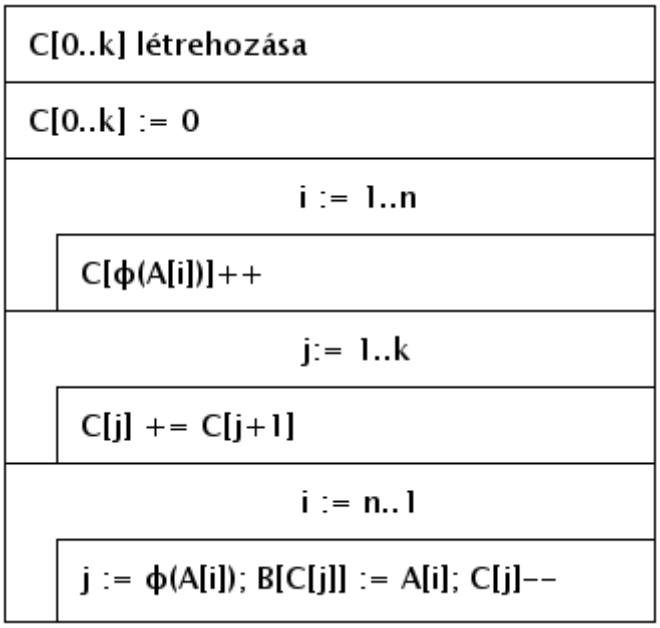
Ezek után felveszünk egy B vektort, amelybe felvesszük a rendezés eredményét. Ez úgy történik, hogy megnézzük a legutolsó elemet. A φ értéke nulla, vagy legfeljebb nulla, ez kétszer fordulhat elő, tehát a kettes helyre tesszük be. Majd jön az utolsó elem feldolgozása, ennek 3 a kulcsa, legfeljebb 3 kulcsú elemből 5 van, ezt folytatjuk, amíg nem fejezzük be az elsővel.

	B
1	013
2	033
3	231
4	312
5	332

Végül látjuk, hogy az első számjegy szerint rendezett lett a sorozat. Vajon miért lett ez rendezett? Ha megnézzük az eredeti tömböt, akkor láthatjuk a második és harmadik számjegyek szerint már rendezve volt a sorozat. Megnéztük, hogy milyen kulcsokkal kezdődtek a számok, ezekből megnéztük, hogy maximum hány darab fordul elő.

Leszámláló rendezés

COUNTING_SORT(A[1..n], k, φ, B[1..n])



Ennek a műveletigénye mekkora? A létrehozás $\theta(1)$, a lenullázás $\theta(k)$, az első ciklus $\theta(n)$, a második ciklus $\theta(k)$, míg az utolsó szintén $\theta(n)$. Tehát ez összesen $\theta(n+k) = \theta(n)$ $\big(k \in \theta(n)\big)$.

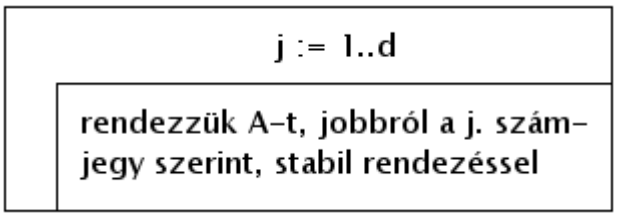
Radix rendezés (számjegypozíciós rendezés)

Van egy összetett kulcs, amely több kulcs kompozíciója. Képzeljük el, hogy lerendezzük az adatainkat a kulcsaink jobbszélső számjegyei szerint, ezt folytatva a jobbról második számjeggyel. Mivel ez stabil rendezés, ezért a jobbszélső számjegy szerint rendezve marad. Ezt folytatjuk a jobbról harmadik, negyedik, …, egészen a bal oldali számjegyig.

A kulcsok tehát d-jegyű számok. (Ez a stabil rendezés lehet leszámláló rendezés is.)

Tegyük fel, hogy ez a stabil rendezés a leszámláló rendezés.

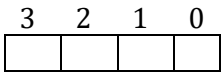
RADIX_SORT(A[1..N],d)



$$T_{RS}(n,k,d) \in \theta\big(d(n+k)\big)\overset{\overset{\text{d konstans}}{\text{k} \in \text{O(n)}}}{\theta(n)}$$

Pl. Tegyük fel, hogy 4 byte-os unsigned a kulcs.

$$\text{d} = 4, \text{k} = 255 \in \text{O(n)}\big(255 \in O(n) \subseteq O(n)\big)$$



i. bájt kinyerése (i=0..3)

$$x \gg 8 * (i - 1)\&255$$

Maradunk a négyes számrendszernél.

<32, 21, 22, 02, 32>

Rendezzük az utolsó számjegy szerint.

0:	
1:	21
2:	32, 22, 02, 32
3:	

Ezek után sorrendben összefűztük az elemeket: <21, 32, 22, 02, 32>

Majd ezt folytatjuk a jobbról a második számjegy szerint:

0:	02
1:	
2:	21, 22
3:	32, 32

Végül ezeket is összefűzzük: <02, 21, 22, 32, 32>

Műveletigény: $d\big(\theta(k) + \theta(n) + \theta(n+k)\big)$. Az első része az inicializálás, a második a szétrakás, a harmadik része pedig az összefűzés műveletigénye, ez pedig függ a számjegyek számától (d). Ez összesen végül:

$$\theta\big(d(n+k)\big) = \theta(n)$$

ahol d egy konstans és $k \in O(n)$.

Hash-táblák

Azaz hasítótáblák. Megpróbálja az átlagos futási időt konstansra lecsökkenteni. Ennek az az ára, hogy a maximális futási idő viszont picit nő, ez a hasítótábla méretétől is függ.

Ez egy vektornak képzelhető valami.

T[0..(m-1)] : R
T[0],…,T[m-1] : rések (slotok)
n
$\alpha = \frac{n}{m}$ – *kitöltési hányados*
U : k, k', $k_i \in U$ – ez az kulcsok univerzuma
h : U ->0..(m-1) – a kulcsokat leképezi egy nekik való helyre.

Direkt címzés

R = rekordokra mutató pointertípus (rekord*)

U = 0..(n-1)

$\neg(m \gg n)$

fv keres(T[0..(m-1], k)

```
return T[k]
```

beszúr(T[0..(m-1)], p)

```
T[p − kulcs] = p
```

töröl(T[0..(m-1)], k)

```
T[k] := ∅
```

init(T[0..(m-1)])

```
T[0..(m−1)] := ∅
```

Ezen műveletek közül az utolsó műveletigénye lesz csak $\theta(m)$, míg az összes többié $\theta(1)$.

$|U| \gg n$

$\frac{1}{2}$ 96 10 15 1872

$2 * 10^2 * 365 * 10^3 = 830 * 10^5 = 73 * 10^6$ állampolgár van a világon.

Ez teljesen tipikus, hogy a lehetséges kulcsok száma sokkal több, mint amit használunk. Például ha csak név, anyja neve, és születési dátum alapján vennénk a kulcsokat, akkor is sok lehetséges jön ki.

Ezt általában úgy szokták megoldani, hogy megpróbálnak egy m számot keresni. Ez $m \approx 1.1n, 1.2n$ körül van, azaz az m-et 10-20%-kal nagyobbra szokták venni, mint ahány adat van. Az m viszont lehet kisebb is, mint n.

Mivel az univerzum elemszáma sokkal nagyobb, mint m, ezért előfordulhat, hogy két kulcs ugyanarra a résre képez le. $h(k_1) = h(k_2)$. $T_{h(k)}(n) \in \theta(1)$. Ezt hívják kulcsütközésnek. Ezt valahogy kezelni kellene. Milyen lehetőségeink vannak erre? A legtriviálisabb módszer az kulcsütközés feloldása láncolással.

Ezt hogy oldjuk meg? Ha két kulcs ugyanarra a slotra képez le, akkor ezt leginkább úgy oldjuk meg, hogy a résből mutatunk a k_1 kulcsra, majd a k_2 kulcsra mutatunk arra. Ezt egészen addig ismételjük, míg az ilyen kulcsok elfogynak, az utolsó így már nem mutat sehova, a pointere *NIL* értéket vesz fel. A törlés is hasonlóan működik, azaz végigmegy a kulcsokon, és ha talált egyezést, akkor azt egyszerűen kitörli. Szélsőséges esetben előfordulhat, hogy az összes elem ugyanazon a sloton lesz. A beszúrás is kényes, főleg, ha egy slotra több kulcs is leképez. A minimális idővel viszont szerencsénk van, ha maximum egy kulcsunk van, akkor legelsőre kész is vagyunk.

$$MT_{\substack{keresés \\ törlés \\ beszúrás}}(n) \in \theta(n)$$

$$mT_{\substack{keresés \\ törlés \\ beszúrás}}(n) \in \theta(1)$$

Az átlagost kiszámítani viszont nem olyan könnyű már. Ha van egy $\alpha = \frac{n}{m}$ kitöltöttségi hányadosunk, akkor igazából meg is vagyunk.

$$AT_{\substack{keresés \\ törlés \\ beszúrás}}(\alpha) = \theta(1 + \alpha)$$

Ezt lehet így is jelölni: $AT_{\substack{keresés \\ törlés \\ beszúrás}}(n,m) = \theta\left(1+\frac{n}{m}\right)$

Bizonyos esetekben szoktak olyat is csinálni, hogy két irányba láncolják a listákat. Ennek az az előnye, hogy ha egy keresés megtalál egy elemet, és visszaadja a pointerét, akkor egy törlésnek már nem kell újra megkeresnie, hanem egyből tudja törölni ténylegesen is. Ezt egyirányú esetben is meg lehet oldani, csak rafináltan kell a programot megoldani (meg elég sok plusz memória is). Körbeláncolni viszont nem érdemes, mert tudni kell, melyik a baloldali elem, fejelem pedig szintén nincs.

Ha az α nem sokkal haladja meg az egyet, akkor azt pedig felülről is lehet becsülni egy konstanssal.

Nyílt címzés

T[0..(m-1)]: R - tábla, ahol $m \geq n$, és az R nem egy pointer, hanem az elemek típusa

$$\boldsymbol{s : R}$$

→ {**"ü","f","t"**} : státuszfüggvény, ahol "ü": üres, "f": foglalt, "t": törölt, a két szélsőt "szabad"-nak is nevezzük

$$\boldsymbol{h : Ux[0..(m-1)] \rightarrow 0..(m-1)}$$: hasítópróba, szintén leképezés.

$\boldsymbol{h(\cdot,0), h(\cdot,1), \ldots, h(\cdot,m-1)}$: ez m darab hasítófüggvény.

Először vegyük az az esetet, hogy **nincs törlés**

Ha nincs törlés, akkor egyszerű a dolgunk, mert egy beszúrás ellenőrzi, hogy a h(k,0) üres-e. Ha nem, akkor először ellenőrzi, hogy ugyanolyan-e a kulcs, mint amit be akarunk szúrni, mert ebben az esetben szintén kulcsütközés van, ha ezt se áll fenn, megy tovább a h(k,1). Ha viszont üres, akkor beszúrja a kulcsot. Ezt h(k,m-1)-ig csinálja.

$$< h(k,0), h(k,1), \ldots, h(k,m-1) >$$

Ezt **potenciális próbasorozat**nak szokás nevezni, mert nem biztos, hogy végig jutunk rajta, mert könnyen előfordulhat, hogy kulcsütközés, vagy üres slot fordul elő. Ennek a prefixe az **aktuális próbasorozat**, azaz amennyit végignézünk a sorozatban. Tehát kulcsütközés esetén sikertelen a kulcs beszúrása, míg üres slotnál sikeres.

A keresés hasonlóan működik, mert az folyamatosan megy előre, és vagy megtalálja az adott kulcsú elemet (valahányadik próbánál), vagy ha üres rést talál, akkor már biztos nem fogjuk megtalálni az adott elemet, mert a beszúrás mindig a legelső üres helyre teszi az adott kulcsot. A harmadik lehetőség, hogy végigmegy az egész sorozaton, és szintén nem találja meg, ez csak akkor fordulhat elő, ha minden rés tartalmaz kulcsot.

Persze nagyon fontos, hogy a potenciális próbasorozat egy permutációja legyen a $< 0, \ldots, (m-1) >$ sorozatnak. Ennek minden esetben fenn kell állnia. Ha ez nem áll fenn egy ilyen hasítópróbára, akkor a *h* függvény egy nem jó függvény.

Kitérő: Hogy néznek ki a hash függvények? Nézzünk vissza a láncolt esetekre. Vegyünk egy egyszerű függvényt.

$h : U \rightarrow 0 \ldots (m-1)$
egyszerű egyenletes hasítás

Az egyik legegyszerűbb hasítófüggvény azt teszi fel, hogy az univerzum, a kulcsok halmaza: $|U| = [0,1)$, ilyenkor egyszerűen lehet használni azt a kulcs függvényt, hogy $h(k) = \lfloor k * m \rfloor$. A másik lehetőség az **osztó módszer**, ami segít, ha egyszerű egyenletes hasításokat szeretnénk definiálni. Ha feltehetjük, hogy $U \subseteq \mathbb{Z}$ akkor $h(k) = k \bmod m$. Ez szintén egy gyakori választás. Ehhez viszont

szintén feltétel, hogy az m egy prímszám legyen, ami nincs 2 hatványainak közelében. Az ilyen hash függvények általában jól szoktak viselkedni és a táblaméretre is elég szoros. Ilyen például a 11, 13, mert $8 = 2^3$ és $16 = 2^4$ között a hatványoktól elég távol vannak, de akár a 41, 43, 47 is jó választások. Nyilván figyelembe kell venni, hogy hány adatot kell tárolni. Ha a kulcs nem egész szám, akkor egész számmá kell konvertálni.

A harmadik ilyen fontos módszert **szorzó módszer**nek nevezzük, ahol a $0 < A < 1$ relációban az A egy konstans, a $h(k) = \lfloor m * \{k * A\} \rfloor$. Ennek a módszernek az osztó módszerrel szemben az a hátránya, hogy ez a való aritmetika jóval lassabb, mint az egész aritmetika. Erre találtak ki egy módszert, hogy ne feltétlen legyen szükséges valósat használni. Ehhez találták ki azt a megoldást, hogy feltesszük, hogy az $U = 0 \dots 2^w - 1$. Feltesszük még, hogy a tábla mérete: $m = 2^p$, ahol $1 < p < w$. Ezek után vesszük még az $\lfloor A * 2^w \rfloor$. Mivel A nagyobb 0-nál, kisebb 1-nél, ezért ez és a k kulcs biztos, hogy w bites számok lesz. Ha ezeket összeszorozzuk, akkor ez már 2w bites számok lesznek. A ___. ___ kinézetű számnál először jön az r_1, odaképzelünk egy tizedesvesszőt, majd az r_0 következik. Azért képzelhetjük oda, mert $r_1 * 2^w + r_0 = k * \lfloor A * 2^w \rfloor \approx 2^w(k * A) = 2^w \lfloor k * A \rfloor * 2^w\{k * A\}$. Az $r_0 \approx 2^w\{k * A\}$.

Tehát tulajdonképpen: $2^p * r_0 = m * r_0 \approx 2^w\{k * A\}m$.

Maga az algoritmus hogy néz ki akkor?

$\left((k * \lfloor A * 2^w \rfloor)\&2^w - 1\right) \gg (w - p)$. A >> jel aritmetikailag a jobbra shiftelést jelenti (magyar jelentése nincs, ezt így mondjuk).

Ezek voltak a leghíresebb hash függvények. Közülük a szorzó módszer bármekkora méretű táblára használható.

Visszatérve a nyílt címzésre újra. Az első ilyen hasító módszer a

Lineáris próba

$$h(k, i) = (h'(k) + i) \bmod m$$

A lineáris próba le fog képezni egy adott sortra, majd tőle eggyel jobbra lép, és így megy végig az egész táblán. Először a $h'(k)$-n próbálkozik, majd így lép tovább jobb, és ha a végére lép, akkor a legelejére lép, egészen addig ismételve a jobbra lépkedést, míg a kiindulóponthoz nem ért. Ennek a hátránya, hogy nagy, összefüggő tábláknál ez sokáig tart. Például, egy adott k_1 kulcs mellett a k_2 ugyanakkora, a táblán pedig több ugyanakkora rész alakul ki. Ha van egy másik kulcs, amely szintén egy hosszú, összefüggő részt alakít ki, ezek összekapcsolódhatnak egymással. Az ilyen összefüggéseket hívjuk **elsődleges csomósodás**nak.

Másik lehetőség, amely szintén előfordulhat, hogy a $h(k_i) = (h'(k) + c_1 i^2 + c_2 i) \bmod m$, és a $h'(k_1) = h'(k_2) \gg$ itt a teljes próbasorozatuk ugyanaz, ezt **másodlagos csomósodás**nak nevezzük.

Felvetül a kérdés, hogy tényleg lefedi-e a táblát az ilyen négyzetes próba? Erre többfélét kitaláltak, az egyik ilyen példa, hogy ha $m = 2^p$, és $c_1 = c_2 = \frac{1}{2}$.

Nyílt címzés

A nyílt címzésnél az U az univerzum, ami a kulcsok halmaza, egy h leképezés az $U * [0..(n-1)] \rightarrow [0..(n-1)]$ intervallumon, egy $T: [0..(n-1)] : R$ a rekordokat tartalmazza, az r : R esetén pedig az r a kulcs. Az univerzum általában sokkal nagyobb, mint az a mennyiség, amiket tárolunk, azaz $|U| \gg m$, illetve $h(k_1, 0) = h(k_2, 0)$. A sorozatot így kell elképzelni:

$$< h(k, 0), \dots, h(k, m-1) >$$

Illetve van még egy s : R -> {'u','f','t'} függvény is.

Ha üres rést talál, oda beszúrja az adott elemet, ha az egész sorozat üres, akkor szintén beszúrja, ha viszont talál üres rést, de az adott kulcs már létezik, a beszúrás meghiúsul. Ugyanez történik, ha a sorozat tele van. A fent említett sorozat permutációja $< 0, \dots, m-1 >$ sorozatnak.

Mindig n-nel jelöltük, hány adatunk van, míg m-mel a hasítótábla méretét. $\alpha = \frac{n}{m}$ pedig a kitöltöttségi hányados lesz. Erről általában el szoktuk várni, hogy, ne érje el az 1-et, míg 1 fölé sose szokott menni (nem is mehet). Ezért kiköthetjük, hogy $0 \leq \alpha \leq 1$. Ha ez a hányados kisebb, mint 1, a beszúrás lehetséges.

Idáig a törlés problémájával még nem foglalkoztunk. Ideális esetben a következő potenciális próbasorozat valósul meg:

$$< h(k,0), \dots, h(k,m-1) >$$

Ilyenkor egyenletes hasítás jön létre.

Amennyiben $0 < \alpha < 1$, nincs törölt slot és egyenletes hasítás is van, akkor sikertelen keresés történt, ez esetben az aktuális próbasorozat hossza a várható értéke $\leq \frac{1}{1-\alpha}$. Ugyanez a helyzet sikeres beszúrás esetén. Ez nyilván teljesülni fog, hiszen α kisebb, mint 1.

Ellenben mi van, ha sikeres a keresés? Ez esetben a próbasorozat hosszának várható értéke $\leq \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$, ugyanez történik sikertelen beszúrás esetén is. Ennek bizonyítása a hivatalos jegyzetben van.

Vegyünk egy példát. Mit jelent ez konkrét esetben? Legyen $\alpha = 50\%$, Akkor a sikertelen keresés várható hossza ≤ 2. Sikeres keresés esetén ez az érték 1,317.

Ha azonban $\alpha = 90\%$, Akkor a sikertelen keresés várható hossza ≤ 10, míg sikeres keresés esetén ez az érték 2,259.

Ha feltesszük, hogy a törlést megengedjük, az nem megoldás, ha üresre állítjuk a slotot. Képzeljük el, hogy van a következő tábla:

$$< \cdots, 'f', 'f', \dots >$$

Ha viszont a két 'f' érték közül az egyiket 'ü'-re (üresre) állítjuk, a keresés az ürestől jobbra lévő, de foglalt elemet már nem fogja látni. Ezért viszont 't'-re (töröltre) kell állítani az adott slotot, mert ilyenkor a keresés átmegy azon a sloton. Viszont a beszúrást is módosítani kell. Az nem úgy működik, hogy a legelső 'ü'-re beszúrja a kulcsot, hanem megjegyzi, hogy melyik slotok töröltek. A beszúrás vagy megtalálja az adott kulcsot és sikertelen, vagy tele van a tábla és sikertelen, vagy a legelső üres vagy törölt slotra beszúrja a kulcsot. A műveletek tehát:

init(T[i]) : s(T[i])='ü'

T[i]:= r : s(T[i])='f'

töröl(T[i]) : s(T[i]) = 't'

Az előző előadáson volt szó a lineáris próbával. Arról van szó, hogy egy olyan próbasorozatot kell definiálni minden lehetséges kulcsra, amely lefedi az egész táblát.

Kell ehhez egy $h(k,i) = (h'(k) + i) \bmod m$ függvény és $k' : U \to 0..(m-1)$ leképezés. Ezek alapján a tábla $< h'(k), \dots m-1, 0, \dots, h'(k)-1 >$ alakot fog felvenni. Tegyük fel, hogy van i darab slot, amelyek foglaltak. Mekkora a valószínűsége, hogy a következő beszúrással a következő slot

is foglaltá válik? Ha egyenletesen halad a h', akkor minden slotnak ugyanannyi a valószínűsége, azaz $\frac{i+1}{m}$. Ezt a jelenséget hívják elsődleges csomósodásnak.

Ennek kiküszöbölésére kitalálták a négyzetes próbát. Ezt a következőképp definiálták:

$h(k,i) = (h'(k) + i^2) mod\ m$

Később ez a következőképp alakították át:

$h(k,i) = (h'(k) + c_1 i^2 + c_2 i) mod\ m$, ahol $c_1 = c_2 = \frac{1}{2}$, illetve $m = 2^p \ \ (p \in \mathbb{N})$.

Mekkora két szomszédos slot különbsége, hogy lehet ezt kiszámolni? Erre jó a következő képlet:

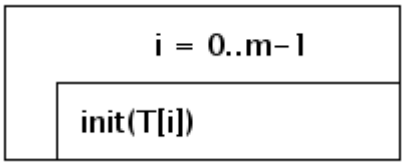
$$\big(h(k,i+1) - h(k,i)\big)\ mod\ m = \left(\frac{(i+1)^2 - (i+1)}{2} - \frac{i^2+i}{2}\right) mod\ m$$
$$= \frac{i^2 + 2i + i + i + 1 - i^2 - i}{2}\ mod\ m = \frac{2i+2}{2}\ mond\ m = (i+1)\ mod\ m$$
$$\gg h(k,i+1) = h(k,i+1) = \big(h(k,i) + (i+1)\big)\ mod\ m$$

$h'(k_1) = h'(k_2) \gg$ az egész próbasorozat ugyanaz, ez másodlagos csomósodás.

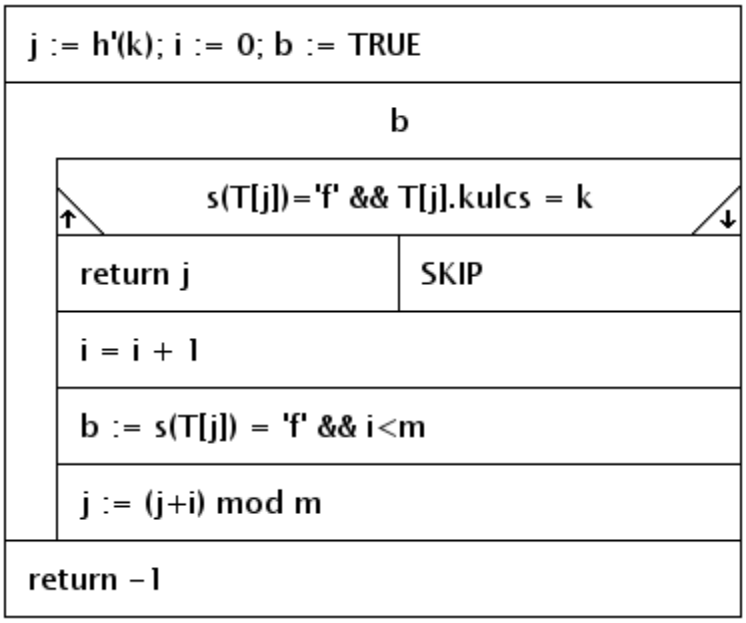
A lineáris próbát akkor szokás használni, ha nagyon kicsi az esélye a kulcsütközésnek.

Hogy lehet leprogramozni ezt a négyzetes próbát?

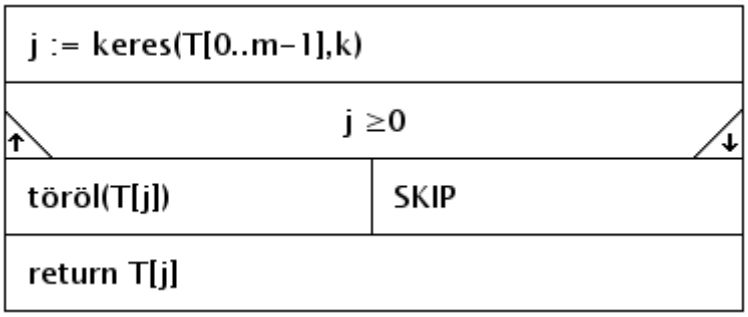
init (T[0..(m-1)])



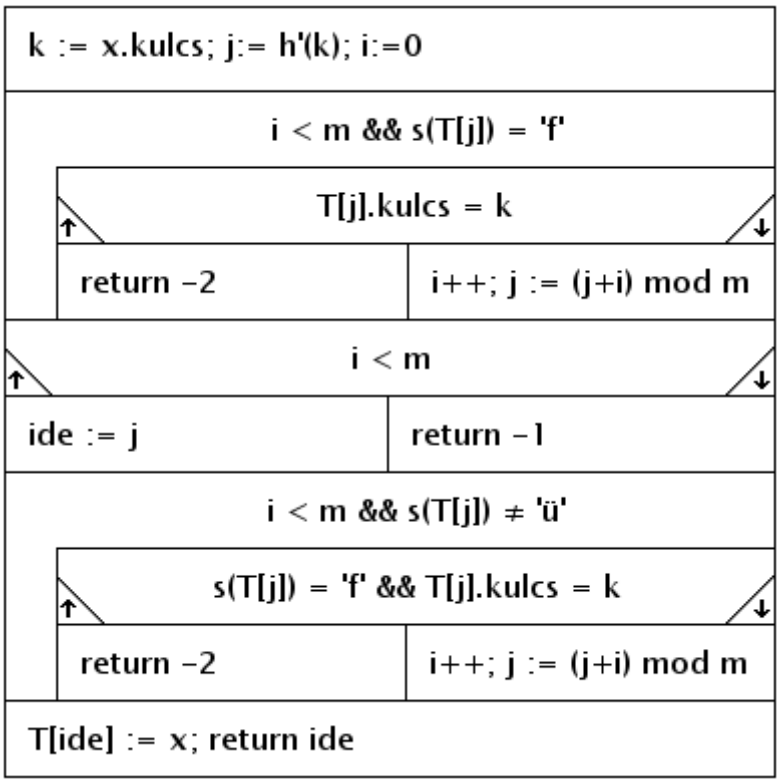
fv keres(T[0..(m-1)],k)



fv töröl(T[0..(m-1)],k)



fv beszúr(T[0..(m-1)],x) (x:R)



Kettős hasítás

$h_1: U \to 0..m-1;\ h_2: U \to 1..m-1$
$h(k,i) = \big(h_1(k) + i * h_2(k)\big)\ mod\ m$
$\theta(n^2)$ próbasorozat

$< h(k,0), \dots, h\big(k,(m-1)\big) >$ permutációja $< 0, \dots, m-1 >$.

Rendezett gráf

A gráf egy kulcsokból és élekből álló halmaz, ahol a csúcsok halmaza véges, az élhalmaz pedig részhalmaza az él*él halmaznak. Képletekkel leírva:

$$G = (V, E)$$
$$0 < |V| \leq +\infty$$
$$E \subseteq V * V$$
$$W: E \to \mathbb{R}$$

A szimmetrikus gráf pedig azt jelenti, ha$(u, v) \in E$, ami annyit jelent, hogy $(v, u) \in E$, azaz $w(u, v) = w(v, u)$. Egy egyszerű gráf egy olyan gráf, amelyben nincsenek hurok- és párhuzamos élek. Természetesen a gráfok lehetnek irányítottak és irányítatlanok is, ezeket lehet grafikusan szemléltetni. A gráfok csúcsait és éleit lehet címkézni, ezekkel könnyebben be lehet őket azonosítani.

Hogyan lehet rendezni egy gráfot? Gráf reprezentációi a következőképpen néznek ki. A csúcsmátrix (szomszédossági mátrix) esetében az ábrázolásnál fel kell tenni, hogy a csúcsok halmaza, $V = \{1, \dots, n\}$ között fel vannak címkézve (ezek részhalmazai $\mathbb{N}$-nek).

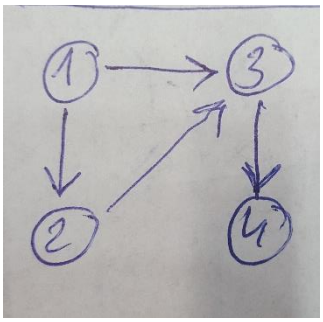
Először nézzük a súlyozatlan eseteket. Ilyenkor azt mondjuk, hogy ha c-vel jelöljük a mátrixot, akkor $c[i,j] = \begin{cases} 1, \text{ha (i,j)} \in \text{E} \\ 0, \text{ha (i,j)} \notin E \end{cases}$. Míg ha súlyozott, akkor a $c[i,j] = \begin{cases} w(i,j), \text{ha } (i,j) \in E \ (i \neq j) \\ 0, \text{ha } i = j \\ +\infty, \text{ha } i \neq j \text{ és } (i,j) \notin E \end{cases}$. A súlyozatlan esetnél tehát a c biteknek egy mátrixa, azaz bitmátrix. Az egész memóriaigénye aszimptotikusan, ha utánaszámolunk, a mátrix méretével arányos, azaz $\theta(n^2)$, amely nagy mátrixok esetén, ha sok csúcsa van a gráf (és ez gyakorlatban sűrűn előfordul), akkor annyival lesz nagyobb a műveletigény is.

Namármost, kérdés, hogyan lehetne csökkenteni a memóriaigényt? Egy természetes csökkentés azonnal eszünkbe juthat, ha szimmetrikus a mátrix, akkor el lehet azzal játszani, hogy úgy ábrázoljuk a mátrixot, hogy ezt egy c'vektorban ábrázoljuk.

	1									
c'	c_{11}	c_{21}	c_{22}	c_{31}	c_{32}	c_{33}	…	c_{n1}	…	c_{nn}

A képlet, amit veszünk: $c[i,j] = c[j,i]$.

A képlet, ahogy ezek viszont kijönnek a vektorban: $c[i,j] = c'\left[\frac{1(i-1)}{2} + j\right]$ $(ha\ i \geq j)$. Így a memória felét meg lehet spórolni. Az a gond, hogy ha hatalmas a gráf és viszonylag kevés él van (pl. csúcsonként van 1-2 él), ez pedig bitekben is rengeteg. Hogyan lehet ezt másképp ábrázolni?



Szomszédossági éllista ← ennek segítségével. A példában egy irányított gráfot vettünk. Mivel van négy csúcsunk, ezért vettük a következő ábrát:

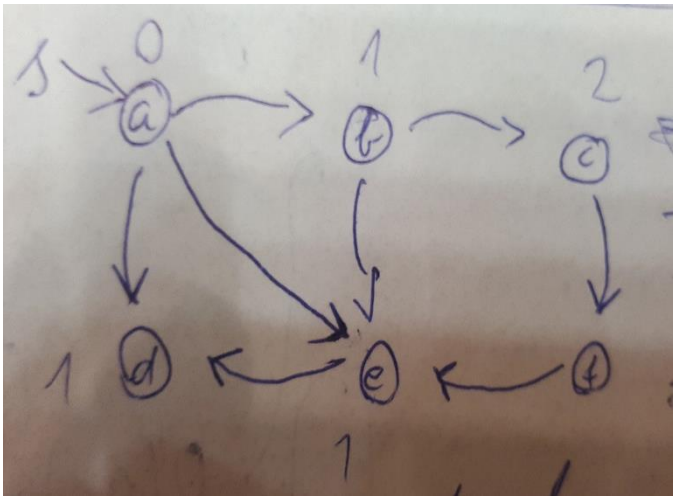
	Adj.					
1		→	2	→	3	X
2		→	3	X		
3		→	4	X		
4	X					

Tehát vettük az alapcsúcsokat, és megnéztük, hogy az adott csúcsból hova mutattak élek, hányas számmal címkézett csúcsokba. Képlettel általánosan: $(u, v) \in E, u(u, v) = W$. Itt a memóriaigény is teljesen más lesz.

A vektorban van n darab elem ($|V| = n, |E| = e$), akkor ennek a memóriaigénye $\theta(n + e)$ lesz, mert a vektor mérete maga az n, míg listaelemek száma maga az e. Ez irányítatlan gráfoknál is így van. Nyilván ha $e \ll n^2$, akkor ez jóval jobb, mint a csúcsmátrixos, vagy szomszédossági mátrixos megoldás. De ha $e \approx \frac{n(n-1)}{2}$ akkor viszont a csúcsmátrixos megoldás a jó, mert akkor nem kell a pointereket tárolni. Ezzel a gráfreprezentációs algoritmusokat le is zárhatjuk.

Gráfalgoritmusok

Szélességi gráfbejárás



Nagyon sok algoritmus kiindulópontja a *szélességi gráfbejárás (breadth first search)*, ami egy igen egyszerű feladatot old meg. Tegyük fel, hogy van egy gráfunk. Jelöljük a csúcsokat a-tól f-ig tartó betűcímkékkel. Kijelöljük az *a* csúcsot (akármelyiket kijelölhetjük), és feltesszük a kérdést, hogy az *a* csúcsból melyik csúcsokba hány lépéssel tudunk eljutni, mi a legrövidebb út. Jelen példában élsúlyozatlan gráfokkal dolgozunk még, következő órán beszélünk majd az élsúlyozott gráfokról. Jelen példában *a* csúcsból az *f* kivételével akármelyik csúcsba el tudunk jutni. Jelen példában ezt igen könnyen meg tudtuk határozni, de ezt egy 1 millió csúcsszámú gráfban hogy lehet meghatározni?

Elindul az algoritmus a start csúcsból, felméri a rá következő csúcsokkal a kapcsolatot (amelyek között is lehet kapcsolat, sőt visszamutathat akár a startra is), majd ezen csúcsok rákövetkezőit is felderíti, majd ezen csúcsok rákövetkezőit (kapcsolat bármelyik csúcsok között lehetnek). Ezt hogy lehet egyszerűen megvalósítani? Úgy hogy sorba rendezi a csúcsokat, amely függ attól, hogy egy adott csúcsot hány lépéstől tudunk elérni. Ilyenkor színekkel jelöljük a csúcsokat:

fehér – ha még nem jártunk ott;
szürke – ha pont ott járunk;
fekete – ha rákövetkezőit is feldolgoztuk;

Ezen csúcsok feldolgozását a sorba rendezetteken könnyű megvalósítani, elindulunk a *start* csúcsból, majd haladunk először az 1 távolságra, majd így tovább a 2. távolságra, a 3. távolságra, a sokadik távolságra, amíg az összes csúcsot fel nem dolgoztuk.

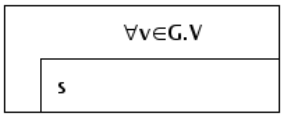
Talán mielőtt felírjuk a stuktogrammot, szemléltessük, hogy a példagráfon ez mit jelent? Vannak tehát az *a, b, c, d, e, f* csúcsok.

								π					
d	a	b	c	d	e	f	Q	a	b	c	d	e	f
	0	∞	∞	∞	∞	∞	<a>	∅	∅	∅	∅	∅	∅
a		1		1	1		<b,d,e>		a		a	a	
b			2				<d,e,c>			b			
d							<e,c>						
e							<c>						
c							<>						
	0	1	2	1	1	∞		∅	a	b	a	a	∅

Mit is csináltunk? A *d* függvény összehasonlítja a szülő csúcsot a rákövetkezőivel. Ha elérjük, akkor leírjuk, hány lépéssel érjük el. A Q oszlopba írtuk le, hogy az adott csúcsnak (amely a legelső oszlopba került) mely csúcsok a gyermekei, míg a Q oszloptól jobbra az adott csúcs oszlopába azt írtuk, hogy melyik a szülője. Az egyes sorokba csak a változásokat írom le, a meglévő értékeket nem szükséges leírni. Ha egy adott csúcs szülőjének értéke marad ∅, akkor nem lehet elérni az adott csúcsot a start csúcsból, nincs oda irányított él. A megtett utak mentén megvastagítjuk (a

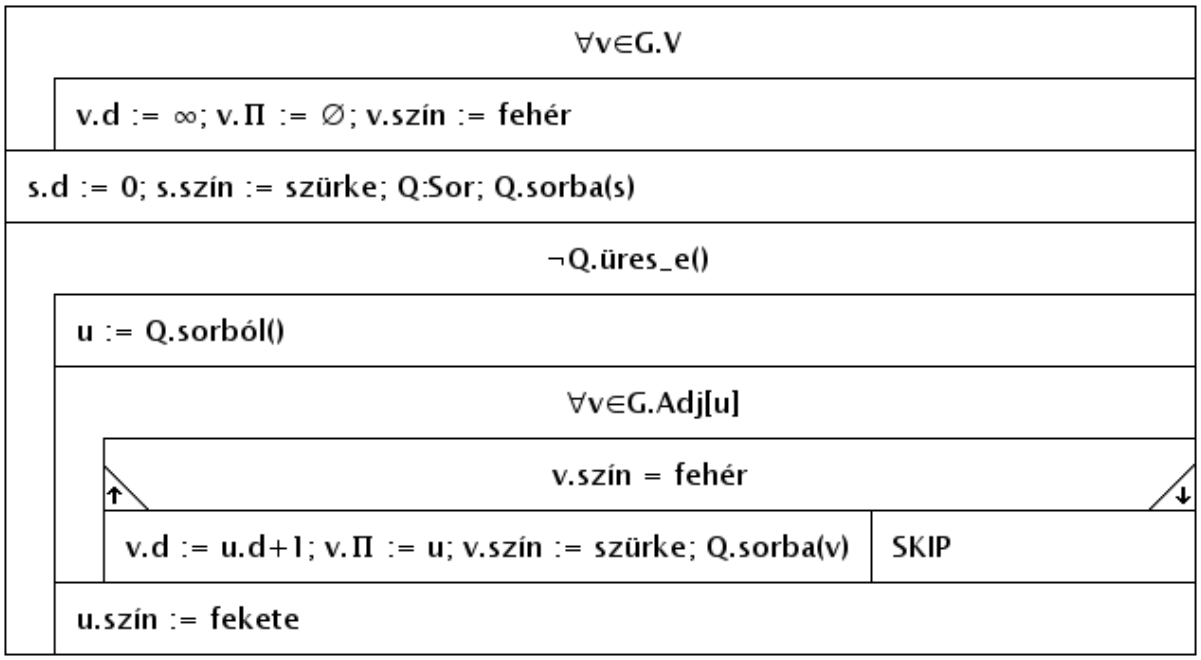
példán duplavonalat húztunk az irányított élhez) az élet, az jelenti az optimális utat és a faszerkezetet. Ebben az ábrában impliciten és expliciten is minden információ benne van.

Mik a G gráf rákövetkezői egy adott csúcsban? $G.Adj[u] = \{v \in G.V \mid (u,v) \in G.E\}$.



Az alábbi algoritmusban a G a gráf, az s pedig a startcsúcs:

BFS(G, s)



Az algoritmus hatékonysága igen egyszerű. Az első ciklus műveletigénye $\theta(n)$, a szürkére színezés sora $\theta(1)$ lesz. Az utolsó nagy ciklus esetében lehet, hogy nem mindegyik csúcs kerül sorra, ezért a belső ciklus nélkül a műveletigénye $O(n)$, míg a belső ciklus annyiszor hajtódik végre, ahány él van a gráfban, vagy esetleg kétszer annyiszor, ha irányítatlan. Mivel maximum kétszer hajtódhat ez végre, ezért $O(e)$ lesz annak a műveletigénye, így maga az egész algoritmus műveletigénye:

$$T_{BFS}(n,e) \in O(n+e)$$
$$MT_{BFS}(n,e) \in \theta(n+e)$$
$$mT_{BFS}(n,e) \in \theta(n)$$

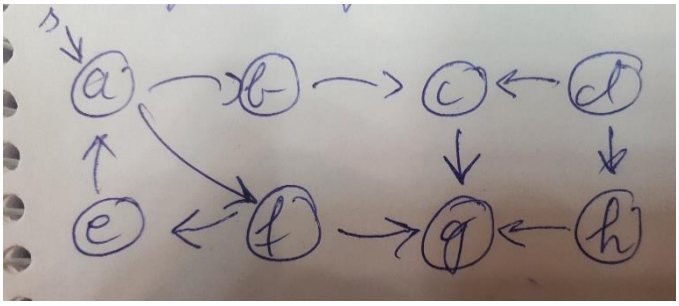
Szomszédossági reprezentációs listánál is ezek a számok érvényesek, de mi van, ha csúcsmátrix van? Ez esetben van három vektorom, amely az attríbútumokat fogja reprezentálni (a *d*, a π, illetve a *szín*). Viszont az a gond, hogy a listán viszont ha végigmegyek, az nem olyan hosszú lépés mint, mint amilyen hosszú a lista, a belső ciklusnál az egy $O(n^2)$ műveletigénnyel fog rendelkezni. Viszont mindenféleképpen n lépés a rákövetkezőket ellenőrizni, tehát a műveletigény csúcsmátrixoknál a következőképpen fog változni:

$$T_{(n)} \in O(n^2)$$
$$MT_{(n)} \in \theta(n^2)$$
$$mT_{(n)} \in \theta(n)$$

A ritkagráf fogalom azt jelenti, hogy $e \in O(n)$, ebből látszik, hogy $T_{BFS}(n) \in O(n)$, míg $T_{BFS_{mátrixreprezentáció}}(n) \in O(n2)$. Sűrűgráfoknál ez annyiban változik, hogy $e \in \theta(n^2)$, $T_{BFS}(n) \in O(n^2)$.

Egy picit belekóstolunk más témába is:

Mélységi gráfbejárás



A szélességi bejárásnál start csúcsból indultunk, majd úgy mentünk a rákövetkezőire mentünk. Egy másik példát vettünk, ahol van 8 csúcsunk, *a*-tól *h*-ig címkézve. Marad az *u.d*, mint *discovery time*, míg bejön az *u.f*, mint *finishing time*, az $u.\pi$ pedig a szülőt határozza meg, míg az *u.szín* a színét.

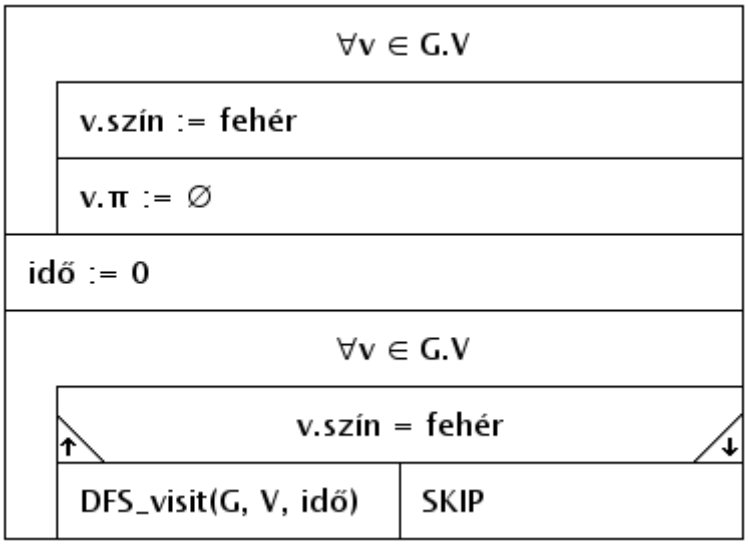
Az algoritmus rekurzív lesz. Elindul a start csúcsból. Első lépésként vegyük azt a konvenciót, hogy ha több rákövetkezője van egy csúcsnak, ábécé sorrendben megyünk rajtuk végig. Így megyünk először a *b* csúcsba, a rá mutató irányított él vonalát pedig duplázzuk, második lépésként pedig megnézzük a *b* csúcs gyermekeit, így jutunk el a *c*-be, ezt érjük el harmadiknak. 4.-nek a *g*-be léptünk, viszont mivel innen nincs több lehetőségünk, ezért az 5. esemény, hogy visszalépünk a szülőbe. Mivel c-ből se tudunk továbbjutni, a 6. lépés az lesz, hogy megint visszalépünk. A *b*-ből viszont van másik rákövetkező, tehát 7. lépésként az *f* csúcsba lépünk, onnan 8. lépésként az e-be lépünk. *b*-ből el tudjuk érni a start csúcsot, ott V-vel jelöljük, hogy *vissza*, majd 9. lépés, vissza az *f*-be. Onnan el tudjuk érni ismét a *g*-t, ezért *K*-val jelöljük, hogy *keresztél*, ezért 10. lépés, visszamegyünk a szülőbe. 11. és 12. lépés, hogy visszamegyünk *b* szülőjébe, majd mivel *a*-ból nem tudunk továbbmenni, ellenőrizzük, melyik csúcsokat nem tudtuk elérni.

Onnan is indítunk egy mélységi bejárást. A *d* a legkisebb, amit nem érintettünk, 13. lépés, onnan a *c*-be tudnánk lépni, de ott voltunk, *K*-val jelöljük. A *h*-ba tudnánk csak lépni, 14. lépés, majd onnan *g* felé *K*-t jelölünk, 15. lépés a szülőbe visszalépés, 16.-kal pedig lezárjuk ismét a mélységi bejárást.

Depth first search

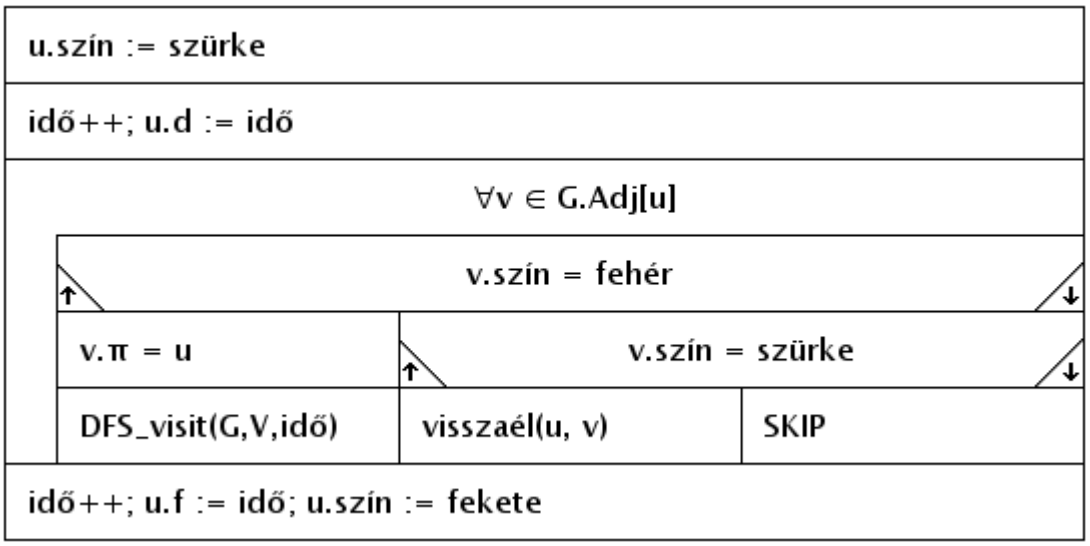
Itt nincs kitüntetett kezdőpont, itt véletlenszerűen kiválasztott pontból kell bejárni. Ha maradt érintetlen csúcs, akkor újabb bejárást csinál. Azaz a teljes bejárás több kisebb bejárásból áll.

DFS(G) (G=(V,E) irányított)



Műveletigény: Az első ciklus nyilván $\theta(n)$ lesz, az idő lenullázása $\theta(1)$, míg a második ciklus szintén $\theta(n)$.

DFS_visit(G, u, &idő) (u.d – discovery time; u.f – finishing time)



Megjegyzés: $G.Adj[u] = \{v \in G.V \mid (u,v) \in G.E\}$

Műveletigény: Az összes meghívásra számoljuk ki a futási időt. Minden csúcsot csak akkor fogjuk beszínezni, amikor fehér csúcsként megtaláljuk. Ezt minden csúcsra egyszer fogjuk végrehajtani, így $\theta(n)$ lesz a műveletigény. Mi a helyzet a ciklussal? Az az élek számától függ, így $\theta(e)$-szer fog végrehajtódni. Az utolsó sor szintén $\theta(n)$-szer történik meg.

$$T(n,e) \in \theta(n+e)$$

a → b ← c
↓ ↔ ↓ ↓
d ↔ e f

Első lépésben az *a* csúcsba lépünk, onnan tovább is tudunk lépni a *b* csúcsba 2. lépésként. 3. lépésként mehetünk az *e*-be, ahonnan vissza tudunk jutni az *a*-ba, de az már szürke csúcs, ezért az oda vezető él *v*, azaz visszél lesz, 4. lépésként befejezzük a *e* csúcsot, 5. lépésként a *b*-t is. 6. lépésként látjuk, hogy *a*-ból még el tudunk jutni a *d*-be, ahonnan megint el tudunk jutni *e*-be, de ott már voltunk, ezért az az él *k*, azaz keresztél lesz. 7. lépésben befejezzük a *d*-t. Utána megint látjuk, hogy ismét el tudnánk érni *a*-ból az *e* csúcsot, azt az élet *e*-vel jelöljük. Ezzel be is fejeztük az *a*-t is, a 8. időpontban. Maradt még két csúcs, a *c* lesz az új gyökere a mélységi fának a 9-es időpontban. Először megnézi a *b*-t, ami fekete csúcs, viszont másik mélységi fában, ezért az is keresztél lesz. Maradt az *f*-be mutató él, a 10. lépésként oda is lép, 11. lépésként be is fejezi, végül 12.-ként a *c*-t is befejezi és ezzel kész a bejárás.

Ezen lehet látni, hogy hogyan is működik az egész bejárás.

Volt négy féle él. Ha $u \to v$ csúcsok vannak, akkor:

- *faél:* Ha egy mélységi fának egy éle van;
- *visszaél:* Ha egy mélységi fában a *v* csúcs az *u* őse;
- *előreél:* Ha egy mélységi fában a *v* csúcs nem közvetlenül az *u* leszármazottja;
- *keresztél:* Vagy egy mélységi fa két különböző csúcsa között van, vagy két különböző mélységi fát köz össze.

TÉTEL.

A DFS során feldolgozott *(u, v)* él:

- faél akkor, és csak akkor, ha v színe még fehér;
- visszaél, akkor és csak akkor, ha v színe szürke;
- előreél, akkor és csak akkor, ha v színe fekete, és $u.d < v.d$;
- keresztél akkor, ha v színe fekete, és $u.d > v.d$.

Ezen tétel segítségével bármely gráfban bármely élet fel tudunk címkézni.

Nézzük meg, mire is tudjuk használni ezt az algoritmust? A szélességi bejárásnál a legrövidebb utat tudjuk szerencsére bejárni.

Topologikus rendezés

Ez azt jelenti, hogy egy gráf csúcsait sorbarakjuk a példán látható módon:

a → b ← c
↓ → ↓ ↓
d ↔ e f

Tehát rakjuk sorba a csúcsokat, mondjuk valahogy úgy, hogy *c, f, a, d, b, e*. Ha ide berajzoljuk az éleket ugyanúgy, ahogy az eredeti gráfban voltak. A sorban leírás azt jelenti, hogy a sorban él csak előrefele, későbbi élhez mutat a sorban. Konkrét példa az életből: nyakkendőt nem vesz fel az ember az ing előtt, vagy cipőt a zokni előtt…

$G = (V, E)$ irányított, DAG ekvivalens azzal, hogy G-ben nincs irányított kör.

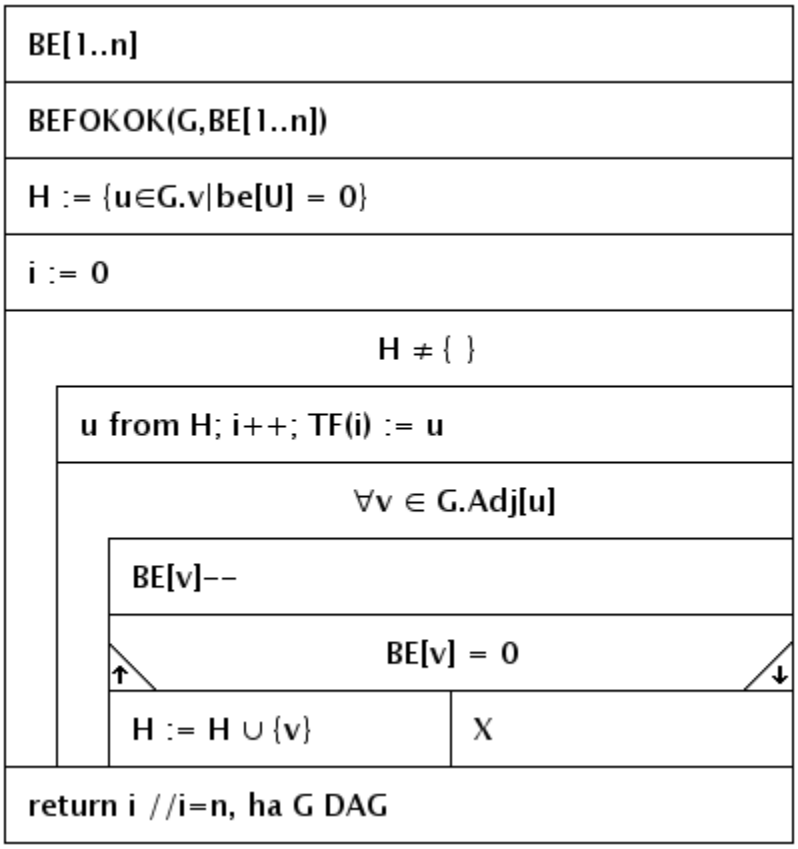
TÉTEL. G-nek ∃ topologikus rendezése, ha G DAG.

BIZONYÍTÁS. ~~Triviális.~~

Ha $G \neg DAG$, akkor van egy $< u_1, u_2, \dots, u_k, u_1 >$ irányított kör. Ha lenne topologikus rendezése, akkor u_1 után jönne u_2, …, u_k, és u_1 végül. Ez viszont ellentmondás, tehát nincs topologikus rendezése, azaz $\nexists G$-nek topologikus rendezése.

Ha viszont G DAG, akkor ∃ topologikus rendezés. Ilyenkor nincs irányított kör, tehát létezik olyan u_1 csúcs, amelynek nincs megelőzője. Ez legyen az első a topologikus rendezésben. Tegyük fel, hogy ezt most töröljük a gráfból, az összes olyan éllel, amelyet kitörlünk belőle. A maradék gráfban sincsen irányított kör. Ezt folytassuk tovább az u_2 csúccsal, azaz töröljük azt is.

fv TF(G,TR[1..n])



Megjegyzés: 2. sorban: $\forall n \in G.V: BE[u] \coloneqq |\{v \in G.V | (v,u) \in G.E\}|$

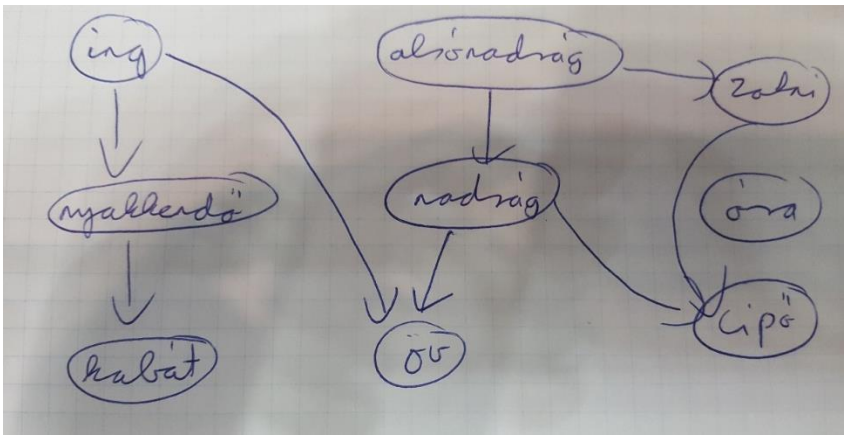
Műveletigény: Az első sor az $\theta(n+e)$, a második sor pedig $\theta(n)$ műveletigényű lesz. Az i lenullázása $\theta(1)$, míg a nagy ciklus legeleje $\theta(n)$. A belső ciklus az élek számától függ, annak műveletigénye tehát $\theta(e)$ lesz. Mekkora lesz tehát az egész algoritmus műveletigénye?

$$T(u,e) \in O(n+e)$$

Megjegyzés: G DAG esetén: $T(n,e) \in \theta(n+e)$

Ez lesz tehát a topologikus rendezés, csak egy apró hiba van benne. Ha véletlenül tartalmaz irányított kört a gráf, azt visszakapjuk, hogy van benne, csak azt nem, hogy hol. De mélységi bejárással az irányított kör helyét is meg lehet adni.

Vegyünk egy konkrét példát:



Első eseményként felvesszük az alsónadrágot, majd utána másodiként a nadrág következik. Harmadikként a cipőt, majd 4. eseményként azt „befejezzük". Nadrág után vesszük fel az övet, majd utána a kabátot, azt 7. lépésként be is fejezzük, 8.-nak befejezzük az övet, 9.-nek a nadrágot. 10.ként a zoknit vesszük fel, mivel cipő már volt, az ott keresztél lesz, 11.-ként a zoknit is befejezzük. 12.-ként az alsónadrággal is végeztünk. 13. eseményként elkezdjük az inget, 14.-ként nyakkendő következik, onnan a kabát jönne, de nem jön, mert már volt, tehát ez is keresztél, így 15.-ként befejezzük a nyakkendőt. Az ingnél találunk még egy övet, ami keresztél, tehát befejezzük az inget is, 17.-ként elkezdjük az órát, és 18.-ként be is fejezzük.

A DFS-re épülő topologikus rendezés:

1. kezdetben a TR üres
2. Mikor egy csúcsot befejezzük, akkor berakjuk a TR elejére
3. Ha egy (u, v) él feldolgozása során v szürke, akkor (u,v) visszél, és G nem DAG.

Minimális feszítőfák

Anno Lengyelországban meg akartak valósítani valami projektet, a példa kedvéért vettünk 6 várost *a*-tól *f*-ig jelölve őket. Ezek között vannak összeköttetések, ahol a vonalaknak vannak költségeik. Rajz híján szöveges magyarázat:

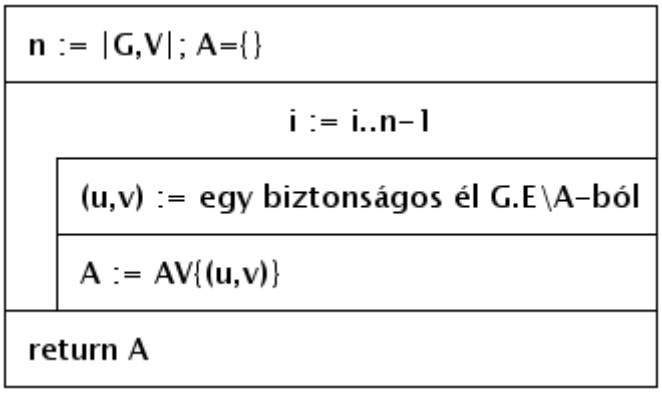
a-b: 2
a-d: 4
b-d: 5
d-e: 1
b-e: 3
b-c: 4
c-e: 1
c-f: 5
e-f: 2

Ezek a lehetséges összeköttetések. A lényeg, hogy olyan összeköttetést válasszunk, amelyek a legkevesebb költséggel járnak. Ebből egy irányítatlan fát szeretnénk kiválasztani. Kört nem célszerű rajzolni, mert az egyik élét ilyenkor mindig el lehet hagyni.

Tehát van ez az összefüggő gráfunk ($G = (V, E), E \subseteq V * V, w: E \to \mathbb{R}$). Mivel irányítatlan, ezért tetszőleges (u, v) él esetén $(u, v) = (v, u)$, illetve $w(u, v) = w(v, u)$. Ennek lesz egy $T = (V, T_E)$, ez egy összefüggő, körmentes gráf lesz, ezt hívjuk **feszítőfá**nak (*spanning tree* angolul). Ehhez a gráfhoz tudunk egy súlyt is rendelni, ez $w(T) = \sum_{(u,v)\in T_e} w(u, v)$. Mikor lesz minimális a feszítőfa? Ha a T: MST a definíció szerint $\forall T'$ feszítőfára $w(T') \geq w(T)$.

Egy általános feszítőfa megtalálására van algoritmusunk is.

fv Gen_MST(G,w)



Megjegyzés: A ciklusban van egy invariáns, ami azt jelenti, hogy $A \subseteq$ valamely MST élhalmazának. Biztonságos egy él, a $AV\{(u,v)\} \subseteq (u,v) \notin A$.

Kérdés, hogyan tudjuk meghatározni ezt a biztonságos élet? Ehhez fogunk megadni pár meghatározást. Hogy ha S nemüres halmaz, amely nem a teljes gráf élhalmaza ($\{\emptyset\} \subset S \subset V$, ez esetén *(S, V*S)*-t vágásnak nevezzük. $(u,v) \in E$ keresztezi a vágást definíció szerint, ha $u \in S$ && $v \in V\backslash S$ vagy $u \in V\backslash S$ && $v \in S$. (u,v) a vágásban könnyű él, definíció szerint, ha (u,v) keresztezi a vágást, és $\forall(x,y)$ él, ami szintén keresztezi a vágást, a $w(x,y) \geq w(u,v)$. Még azt kéne tudni, hogy mit jelent, ha egy vágás elkerüli A-t. $A \subseteq E$-t akkor kerüli el a vágás (definíció szerint), ha A-nak egyetlen éle sem keresztezi a vágást.

TÉTEL. $G = (V,E), u: E \to \mathbb{R}$. Tegyük fel, hogy $A \subseteq$ valamely MST élhalmaznak, illetve $(S, V\backslash S)$ vágás elkerüli A-t, és $(u,v) \in E\backslash A$ könnyű él $(S, V\backslash S)$ vágásban. Ekkor (u,v) biztonságos A-ra nézve. ~~(„Tudom, hogy nem lehet mindent elolvasni, figyeljék amit mondok!")~~

Megjegyzés

Vettük az eredeti gráfot, és vettünk rajta három élet, ahol a rajzban duplavonallal szerepeltek. $A = \{(d,e),(c,e),(e,f)\}, V = \{a,b\}$. A három él dupla vonallal lett megjelölve. Ezek alapján az a és a b csúcsok lettek levágva, ezt jelöli a V halmaz is. Ebből következik, hogy (b,e) könnyű él a $(\{a,b\},\{c,d,e,f\})$ vágásban. Ebből tovább következik, hogy (b,e) biztonságos $A = \{(d,e),(c,e),(e,f)\}$-re nézve. Ebből még tovább következik, hogy $AV\{(b,e)\}$, ez pedig a definíció szerint igaz.

Így $AV\{(b,e)\} = \{(b,e),(c,e),(d,e),(e,f)\}$, ez továbbra is MST egy részhalmaza.

BIZONYÍTÁS.

Legyen $T = (V, T_E), MST: T_E \supset= A$ (//nem tudtam aláírni az $=$-et). $(S, V\backslash S)$ elkerüli A-t, ha $(u,v) \in E\backslash A, (u,v)$ könnyű él $(S, V\backslash S)$-ben. Itt vettünk egy példát, ahol két csúcs, u, és v el voltak vágva. Itt két eset lehetséges:

a) $(u,v) \in T_E$, ekkor $AV\{(u,v)\} \subseteq T_E$. Ez a része igaz, bebizonyítottuk.

b) $(u,v) \notin T_E$. Vettünk a példában még egy *(p,q)* élt is, amelyek szintén el vannak vágva. Ekkor $(p,g) \notin A \Rightarrow A \subseteq T_E\backslash\{(p,q)\}$. Legyen egy $T' = (V, T'_E)$, ahol $T'_E = T_E\backslash\{(p,q)\} \cup \{(u,v)\}$.

$w(T') = w(T) - w(p,q) + w(u,v) \leq w(T)$, ugyanis $w(p,q) \geq w(u,v)$. Ezzel pedig tulajdonképpen be is bizonyítottuk az állítást, mivel $T'_E \supset= A \cup \{(u,v)\}$. ■

Már csak az algoritmusa kellene ennek a műveletnek. Erre már jópár algoritmust kidolgoztak a történelemben, de erre a legversenyképesebb megoldás a Kruskal-algoritmusa (mint Diszkrét matematika 2.-ből).

Itt vettünk szintén egy gráfot, rajz híján az élek súlyát ismét szövegesen tudom csak átadni.

a-b: 2
a-d: 4
b-d: 3
d-e: 1
b-e: 3
b-c: 4
c-e: 6
c-f: 5
e-f: 5

Első körben monoton növekvően rendezzük E-t:

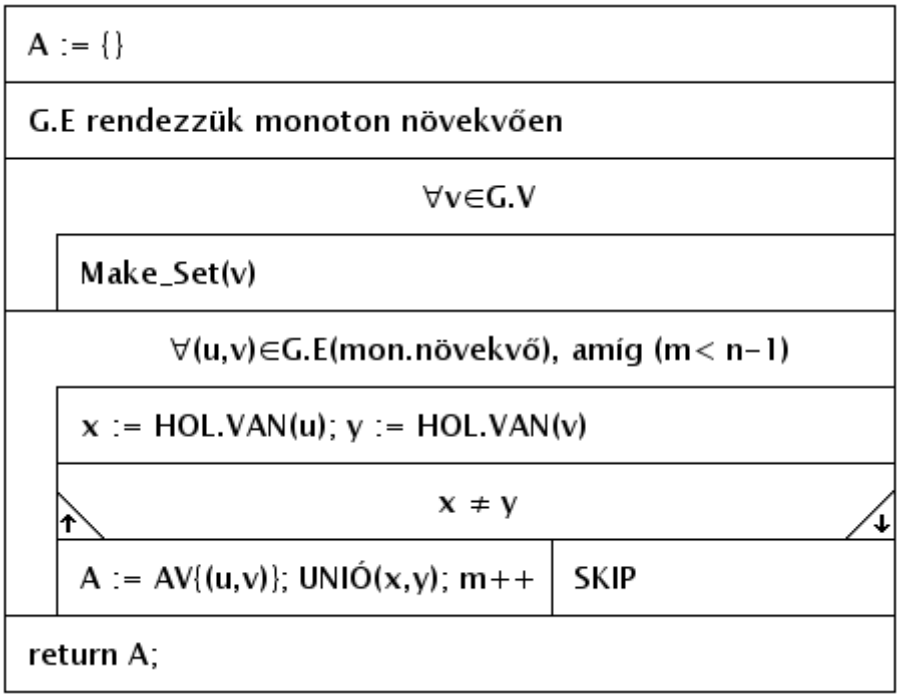
	1	2	3	4	5	6	7	8	9
	$d-e$ 1	$a-b$ 2	$e-f$ 2	$b-e$ 3	$a-d$ 5	$d-b$ 5	$c-f$ 5	$b-c$ 6	$e-c$ 6
a b c d e f	a b c d-e f	a-b c d-e f	a-b c d-e-f	a-b c | d-e-f	a-b c | d-e-f	a-b c | d-e-f	a-b c | | d-e-f		

Mit csináltunk? Az oszlopok mentén haladva, vettük az éleket, és ha a berajzolásával még nem keletkezett kör, akkor berajzoltuk a gráfba. Ha kihagytuk, mindig mentünk tovább. A lényeg, hogy addig menjünk, míg annyi élt rajzoltunk be, míg már többet nem tudunk berajzolni kör létrehozása nélkül.

Ha van két komponens, amelyek csak egy éllel vannak összekötve, akkor vegyünk egy vágást, amely kettészedi ezeket. Ez az egy él tehát könnyű él lesz, mert ennek a legalacsonyabb költségű, ha lenne ennél olcsóbb, az a rendezésben is előrébb lett volna.

Hogy néz ki az algoritmus?

fv KRUSKAL_MST(G,w)



Az előbbi példában, ha irányítjuk a gráfot, a következő jön ki:

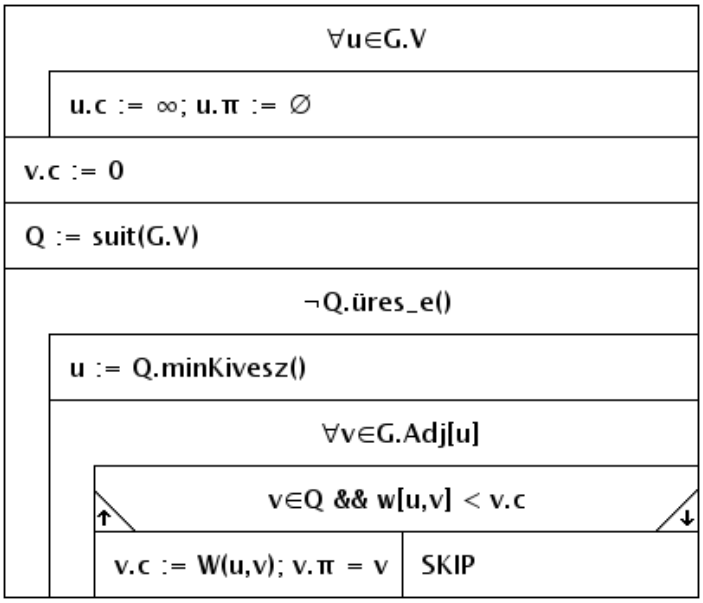
1	2	3	4	5	6	7	8	9
$d-e$ 1	$a-b$ 2	$e-f$ 2	$b-e$ 3	a-d 5	d-b 5	$c-f$ 5	$b-c$ 6	$e-c$ 6
$a \quad b \quad c$ $d \rightarrow \quad e \quad f$	$a \rightarrow \quad b \quad c$ $d \rightarrow \quad e \quad f$	$a \rightarrow \quad b \quad c$ $d \rightarrow \quad e \quad \leftarrow f$	$a \rightarrow \quad b \quad c$ $\uparrow$ $d \rightarrow \quad e \quad \leftarrow f$			$a \rightarrow \quad b \quad \leftarrow c$ $d \rightarrow \quad e \quad \leftarrow f$		

Most már csak azt kellene meghatározni, mennyibe is kerül nekünk ez az algoritmus?

Az első sor $O(e \log e)$, az első ciklus $\theta(n)$, a második ciklusban a ciklus feltétele $O(e)$, az első sora $O(e \log e)$, az *if-else* ág $O(e)$, míg a return $\theta(n)$. Az egész $O\big((n+e)\log n\big)$, az egész pedig $O(e \log n)$ lesz.

Prim-algoritmus

PRIM_MST(G,w,v)



Vettünk egy gráfot, ahol a csúcsoknak voltak élsúlyaik is, ezek a kapcsolódási értékek *(v.c)*. Rajz híján szöveges magyarázat:

a-b: 2
a-d: 4
b-c: 6
b-e: 3
b-d: 5
c-e: 6
c-f: 5
d-e: 1
e-f: 2

Majd vettük a következő számolást:

	c							π					
	a	b	c	d	e	f	Q	a	b	c	d	e	f
	0	∞	∞	∞	∞	∞	a,b,c,d,e,f	∅	∅	∅	∅	∅	∅
a		2		4			b,c,d,e,f		a		a		
b			6		3		c,d,e,f			b		b	
e				1		2	c,d,f				e		e
d							c,f						
f			5				c			f			
c							$<>$						
	0	2	5	1	3	2		∅	a	f	e	b	e

Mit is csináltunk? Elindultunk egy kiindulású csúcsból, majd végignézve a szomszédait, felírtuk, hogy az adott csúcsból egy szomszédos kulcs milyen költséggel érthető el, a Q sorból kivettük, majd felírtuk a szomszédos csúcs szülőjét, ahonnan elértük. Mindig figyeltük a költségeket, a szomszédok figyelembe vételénél ugyanis, ha egy csúcsnak több is volt, először azt vesszük figyelembe, amelyiknek kevesebb a költsége, hogy elérjük. Amelyik csúcsokat vettük, azok feszítőfát fognak alkotni. Ha nem tudtunk venni olyan csúcsot, amelyet már elértünk, de el tudnánk érni, akkor az adott sor üres marad. Nem adjuk össze a korábbi költségeket, mindig csak az aktuálisat tekintettük, ezért is van, hogy a *c* csúcshoz az *f* csúcson át kapcsolódunk, nem a *b*-n, vagy az *e*-n át, mivel *f*-ből csak 5 a költség, míg a másik kettőből 6-6.

Egy dolgot kell már csak meggondolnunk: mennyire hatékony ez az algoritmus? Szokásos jelöléseinkkel ($n = |G.V|, e = |G.E|$). Az algoritmus első ciklusa $\theta(n)$ lesz, a következő ciklusig tartó két sor is $\theta(n)$-ből megúszható. A második ciklus „magja" szintén $\theta(n)$ lesz. A ciklus első sora egy kérdéses eset, mert két lehetőség van: rendezetlen a prioritásos sor, akkor ez n lépés lesz, mivel mindegyiket végig kell ellenőrizni, ezért $n * \theta(n) = \theta(n^2)$ lesz a rendezetlen eset. Ha viszont ez a prioritásos sor egy lineáris kupacban van benne, azaz ez egy minimumkupac, akkor az egész $n * O(\lg n) = O(n * \lg n)$ alatt fog megtörténni. A belső ciklus $2e * \theta(1)$-szer hajtódik végre, azaz $\theta(e)$. És a vége, az utolsó szintén $2e * \theta(1)$-szer hajtódik végre maximum, azaz szintén $\theta(e)$. Kupac esetén ezek $\theta(e)$-re, illetve $O(e)$-re módosulnak.

Rendezetlen sor esetén a műveletigény:

$$T(n) \in \theta(n^2)$$

Míg lineáris minimumkupac esetén fontos, hogy éllistás gráfábrázolás szükséges. Annyi kiegészítés szükséges a stuktogramhoz, hogy az utolsó sorban végrehajtódik egy *HELYREÁLLÍTÁS(Q,v)* metódus is $O(\lg n)$ műveletigénnyel, de maximum $2e$-szer, így maga az egész műveletigénye $O(e * \lg n)$ lesz. A teljes műveletigény tehát ebben az esetben:

$$O(n + e)(\lg n) = O(e * \lg n) \ \ (e \geq n - 1)$$

Végeredményben:

$$T(n, e) \in O(e * \lg n)$$

Ez volt tehát a Prim-algoritmus.

Most pedig témát váltunk.

Dijskra-algortmus

Vettünk egy $G = (V, E)$ gráfot, egy $w: E \to P_0$ leképezést, ahol $P_0 = \{u \in \mathbb{R} | u \geq 0\}$. Vettünk egy irányított gráfot, ahol a csúcsoknak voltak költségei is, itt legyen az *a* a startcsúcs:

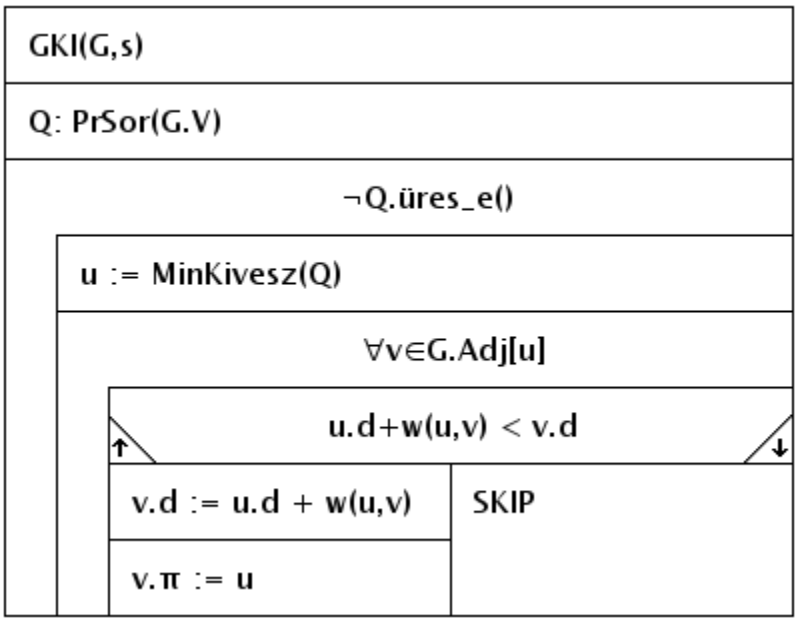
$$a \to d: 1$$
$$a \to b: 2$$
$$a \to d: 4$$
$$b \to c: 0$$
$$d \to c: 2$$
$$d \to b: 1$$
$$c \to d: 1$$
$$e \to d: 3$$
$$e \to c: 2$$

	d						π				
	a	b	c	d	e	Q	a	b	c	d	e
	0	∞	∞	∞	∞	a, b, c, d, e	∅	∅	∅	∅	∅
a		2	4	1		b, c, d, e		a	a	a	
d			3			b, c, e			d		
b			2			c, e			b		
c						e					
e						< >					
	0	2	2	1	0		∅	a	d	a	∅
	a	b	c	d	e		a	b	c	d	e

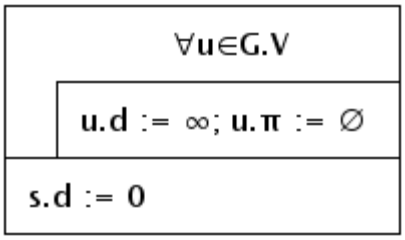
Mit csináltunk? Elindultunk a startcsúcsból, majd vettük a szomszédait, és leírtuk az elérési költségeit. A soron lévő csúcsot kivettük a *Q* sorból, és utána leírtuk az adott csúcs szülőjét, hogy onnan tudjuk elérni. Itt viszont figyeljük az elérési költségeket, ezért 3 a *c* elérési költsége *d*-ből,viszont még mindig kevesebb, mintha *a*-ból közvetlen érnénk el. Az épp soron következő kulcsoknál figyelni kell az költségeket, mert mindig az aktuálisan legalacsonyabbat kell figyelembe venni. Ha egy már járt csúcsot el tudnánk érni máshonnan is, de magasabb költséggel, akkor azt nem írjuk bele a táblázatba, figyelmen kívül hagyjuk.

Vegyük az algoritmusát az egésznek.

Dijskra(G,w,s)



GKI(G,s)



A GKI(G,s) metódus műveletigénye $\theta(n)$ lesz.

Most viszont nézzük a fő metódust. Itt a műveletigénye megint egy kérdéses eset, ugyanis ha a Q sor rendezetlen, akkor:

$$T(n) \in O(n^2)$$

Ellenben ha Q egy lineáris minimumkupac, akkor $T(n,e) \in O\big((n+e)\lg n\big)$.

Megjegyzés: Érdemes megemlíteni, hogy ha nem lineáris minimumkupac, hanem Fibonacci minimumkupacot használunk Q-ra, akkor el lehet érni a $T(n,e) \in O(e + n * \lg n)$-es műveletigényt is.

TÉTEL.

Mikor a Dijkstra algoritmus kivesz egy csúcsot Q-ból, akkor az $u.d$ optimális, és s→u út, amit találtunk (ami $u.d$ költségű), az is optimális.

BIZONYÍTÁS.

s csúcsot kivesszük, 0 a költsége, pipa.

Indirekt módon tegyük fel, hogy van olyan u csúcs, amire nem igaz a tétel. Akkor tekintsük az első ilyet, nézzük az s→u optimális utat. Jelöljük a költségét $\delta(s,u)$-val, és vegyük ezen az úton az első olyan csúcsot (pl. e), ami még benne van a Q-ban. Akkor az $e.d$ nyilván optimális, azaz egyenlő a $\delta(s,e)$-vel, és s→e is optimális.

Ekkor $u.d > \delta(s,u) \geq \delta(s,e) = e.d$, ez pedig ellentmondás!■

Negatív körök

Van egy gráfunk, legyen $G = (V,E), w: E \to \mathbb{R}, |V| =: n$ és $(w(u,v) < 0$ is lehet)

A példa kedvéért vettük a következő gráfot:

$$a \to b: 3$$
$$a \to c: 1$$
$$b \to d: -2$$
$$c \to b: -1$$
$$d \to c: 4$$

Az a lesz a startcsúcs.

Definíció.

$< u_1, u_2, \dots, u_n >, u_1 = u_n$ negatív kör, ha $\sum_{i=1}^{n-1} w(u_i, u_{i+1}) < 0$

Tulajdonság.

1) Ha a startcsúcsból nem érhető el negatív kör, akkor tetszőleges csúcsban létezik optimális út.
2) Ha viszont elérhető negatív kör, akkor a negatív körből elérhető tetszőleges csúcsra nincs optimális út.
3) Ha viszont nem létezik negatív kör, akkor tetszőleges optimális útból a körök elhagyhatók.
4) Egy körmentes út max (n-1) élből áll.

A 2)-esre egy példa, hogy ha elindulunk egy startcsúcsból, ahonnan eljutunk egy körbe, és a körből van egy él, ami a célcsúcsba mutat (ahol az él olyan csúcsból mutat ki, amelyik még nem az utolsó még nem érintett csúcs), akkor az már negatív kör.

Ezek lennének az alapvető tulajdonságai ezeknek az irányított / irányítatlan élsúlyozott gráfoknak. Az irányítatlant úgy kell értelmezni, hogy oda-vissza vannak irányítva a gráfok. De alapvetően csak irányított gráfokkal fogunk foglalkozni.

Sor alapú Bellman-Ford

Az algoritmus, amit tárgyalni fogunk, a szélességi keresés egy változata. Ha elkezdenénk ezt a gráfot szélességi kereséssel bejárni, az élsúlyok tekintetével, akkor mi adódna?

d	a	b	c	d	Q	π	a	b	c	d	(*)
	0	∞	∞	∞	$< a >$		∅	∅	∅	∅	INIT
a		3	1		$< b, c >$			a	a		0. menet
b				1	$< c, d >$					b	1. menet
c		0			$< d, b >$			c			
d			~~5~~		$< b >$						2. menet
b				-2	$< d >$					b	
d			~~2~~		$< >$						3. menet
∑	0	0	1	-2			∅	c	a	b	

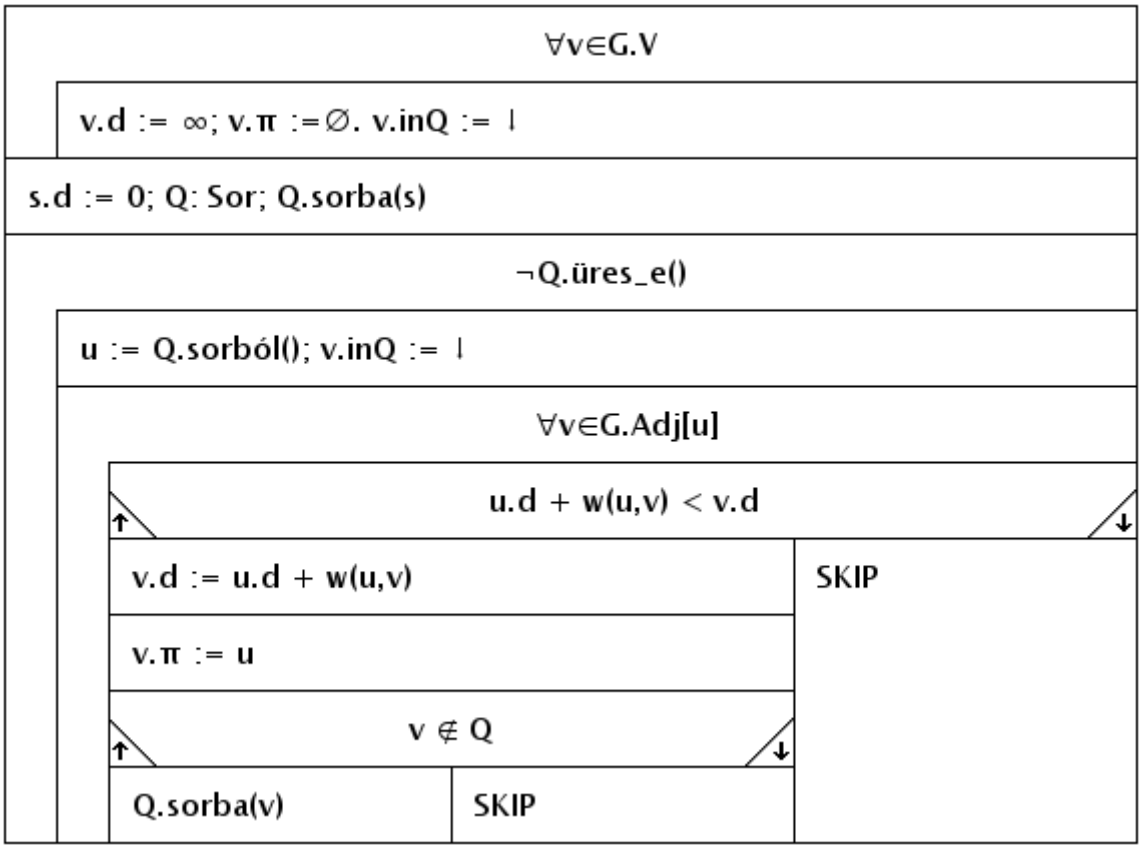
A gráfunk: $(start) \to a: 0, a \to c: 1, c \to b: -1, b \to d: -2$, a csúcsokban az értékek: $a: 0, c: 1, b: 0, d: -2$.

Mit is csináltunk? Sorban haladtunk a csúcsok mentén ábécé sorrendben. Megnéztük, hogy melyik csúcsokat érhetjük el belőle, és beírtuk a táblázatba először is, hogy mekkora az elérés költsége. Majd ezt beírjuk a sorba, kivesszük a sorból azt a csúcsot, amelyiknek a gyermekeit keressük, és leírjuk az adott csúcs szülőjét. Ha sokadik elérésre jutunk el egy csúcsra, akkor az élek költségeit összeadjuk. Ha olyan csúcsba jutunk el, ahol már voltunk, akkor összehasonlítjuk a költségeiket, és az alacsonyabbat vesszük, ugyanezt vesszük a π értékeinél is, azaz hogy az adott csúcsnak melyik a szülője. Ezeket a módosításokat hajtjuk csak végre a szélességi keresésen. Az egész algoritmust addig ismételgetjük, míg üres nem lesz a Q sor.

Az az igazság, hogy ezt az algoritmust egy magyar származású ember, Robert Endre Tarjan készítette, ezt a *Data Structures and Network Algorithms*, amelyet 1983-ban publikált. Ezt az algoritmust **szélességi szkennelés**nek nevezi, angolul *breadth-first scanning*. Azonban ezt csak ő hívja így, a szakirodalomban gyakoribb elnevezés a *Queue_based Bellman-Ford,* ez magyarosítva leginkább **sor_alapú Bellman-Ford** algoritmus.

Nézzük a stuktogramját.

Sor_alapú_BF(G,w,s)



Ez lenne *Tarjan professzor* algoritmusa. A második cikluson belüli ciklust nevezzük az *u* kiterjesztésének.

Vezessük be a **menet** fogalmát, ezek az inicializálás után jönnek, ez a stuktogramunkban az első három sor.

0. menet: a startcsúcs feldolgozása, kivesszük őt a sorból, és kiterjesszük. A teljes második ciklus az az *u* feldolgozása.

i+1. menet: Az i. menet végén a Q-ban lévő csúcsok feldolgozása. Az algoritmusunkat tekintsük meg ehhez, hogy mit jelentettek ezek a menetek (a fenti táblázatban a (*) oszlopban, ami utólag lett beillesztve).

Nézzük meg, hogy ezek után milyen tulajdonságokat tudunk mondani ezen menetek alapján az algoritmusra. Egy alapvető tulajdonság lesz, folytatva a korábbiakkal:

5) Ha az *u* csúcsban létezik olyan *k* élt tartalmazó optimális út, akkor a *k.* menet elején legkésőbb ez már rendelkezésünkre fog állni, azaz már találtunk optimális utat.
6) Ha a startcsúcsból nem érhető el negatív kör, akkor legkésőbb az *(n-1).* menet elejére minden, az *s*-ből elérhető csúcsba optimális utat találunk.
7) Ha a startcsúcsból nem érhető el negatív kör, akkor legkésőbb az *(n-1).* menet végére kiürül a sor.
8) Ha a startcsúcsból nem érhető el negatív kör, akkor az algoritmusunk futási ideje: $T_{SaBF}(n, e) = O(n * e)$.
9) Ha az *(n-1).* menet végére nem ürül ki a Q sor, akkor létezik *s*-ből elérhető negatív kör.
10) Ha az *(n-1).* menet végére nem ürül ki a Q sor, $\Leftrightarrow$ *s*-ből elérhető negatív kör.

11) Ha az *(n-1).* menet végére nem ürül ki a Qsor, akkor a Qsor bármely $w$elemére, ha elindulunk a w-ből a π attribútum mentén, akkor fogunk találni egy ismétlő v csúcsot, azaz $< v, \dots, v.\pi.\pi, v.\pi, v >$ negatív kör, és s-ből ez elérhető.

Az algoritmus $\min(n, e) + e \leq 2e$ csúcsot fog feldolgozni. $\min(n, e) + e$ alkalommal fut le a külső ciklus, és e alkalommal a belső ciklus (ezek felső becslések). Ezért a teljes műveletigény:

$$2e * n + n + 1 \in O(n * e).$$

Ennek az algoritmusnak az a legnagyobb hátránya, hogy igen magas futási ideje is lehet. De mi van, ha van olyan gráfunk, amelyben vannak negatív élek, de nincs benne negatív kör? A Dijkstra algoritmus a maga alacsonyabb műveletigényével ezt hatékonyabban tudta megoldani. De a kérdés az, hogy ha nincs egyáltalán kör a gráfban, ki lehet-e ezt használni? A válasz: igen, mégpedig úgy, hogy a gráf csúcsait topologikusan rendezzük és utána a csúcsokat kiterjesszük.

Nézzünk erre egy példát.

Legrövidebb utak egy forrásból körmentes irányított gráfokhoz

Legyen a következő, topologikusan rendezett gráfunk:

$$a \to b: 2$$
$$a \to e: 2$$
$$a \to d: 3$$
$$b \to e: -1$$
$$b \to f: 2$$
$$b \to c: 1$$
$$e \to f: 2$$
$$e \to d: 1$$
$$f \to c: 1$$

Az algoritmus először is topologikusan rendezi a gráfot, majd végigmegy ezen a rendezésen és minden csúcsot kiterjeszt.

d	a	b	c	d	e	f	π	a	b	c	d	e	f
init	∞	0	∞	∞	∞	∞							
a													
b			1		-1	2				b		b	b
e				0		1					e		e
f													
d													
c													
∑	∞	0	1	0	-1	1		∅	∅	b	e	b	e

Mit is csináltunk? Elindultunk a startcsúcsból, majd utána megnéztük, melyik csúcsokba vezet onnan él, beírjuk a táblázatba, majd a jobb oldalra pedig az adott csúcs szülőjét. Több gyerek esetén a legkisebb költségű csúcsot kell venni. Időhiány miatt nem fejeztük be a kifejtést, csak a végeredményt.

Futási ideje: $\theta(n + e)$ az első fázisnak, és $\theta(n + e)$ a második menetnek, azaz a teljes költség $\theta(n + e)$.

Legrövidebb utak minden csúcspárhoz

Van egy egyszerű gráfunk, amelyben nincs negatív kör. $G = (V, E), w: E \to \mathbb{R}$, az élek száma pedig $V = 1 \dots n \quad (n \in \mathbb{N}_+)$. Ez lesz a Floyd-Warshall algoritmus, ezzel szeretnénk a legrövidebb utakat kiszámolni. Az algoritmusnak van erre szüksége, hogy a csúcsok 1-től n-ig legyenek sorszámozva.

Ha szeretnénk megszámolni az optimális út költségét, akkor erre kell egy $D_{i,j}$, ami az *i* csúcsból a *j* csúcsba vezető út optimális költsége. Nyilván nemcsak ez érdekel, hanem például ezeket az utakat is meghatározhatjuk egy $\pi_{i,j}$-vel, azaz hogy az *i-j* optimális úton mi a *j* szülője ($i, j \in 1..n$).

Ha nincs negatív kör, akkor tudjuk, hogy létezik ez az optimális út. De mi van, ha nincs ez az út? Akkor viszont $D_{i,j} = \infty$ lesz. Persze ilyenkor a $\pi_{i,j} = \emptyset$ lesz értelemszerűen. Ez az, amit szeretnénk meghatározni, ez az algoritmus célja.

Floydnak volt anno az a zseniális ötlete, hogy (bár ez Warshall előzetes terve alapján), hogy tetszőleges csúcsok között van-e út? Erre volt az az ötlet, hogy definiáljuk a $D_{i,j}^{(k)}$ mátrixot, ahol $k \in 0..n; i, j \in 1..n$. Hasonló feltételekkel a $\pi_{i,j}^{(k)}$ lesz még. Az első mátrix az hasonlóan az *i-j* optimális út költsége, de itt nem akármilyen utak közül válogatunk: az a megszorításunk, hogy a közbenső csúcsok csak $[1..k]$-beli csúcsok közül kerülhetnek ki. Ugyanígy a $\pi_{i,j}^{(k)}$ is *i-j* optimális úton *j* szülője is ugyanezen közbenső csúcsok közül kerülhet ki.

A definíciók után nézzük ezen algoritmus **következmény**eit. Úgy tűnik, hogy $n+1$ darab mátrixpárról van hát szó. Milyen tulajdonságaik van ezeknek? $D_{i,j}^{(0)}$: Alapesetben azt jelenti, hogy [1,0] intervallumból kerülhetnének ki a belső csúcsok. Ez nyilván $D_{i,j}^{(0)} = \begin{cases} 0, \text{ha } i = j \\ w(i,j), \text{ha } (i,j) \in E \text{ és } i \neq j \\ \infty, \text{ha i} \neq \text{j, és } (i,j) \notin E \end{cases}$. Ezzel együtt $\pi_{i,j}^{(0)} = \begin{cases} \emptyset, \text{ha } i = j \\ i, \text{ha } (i,j) \in E, \text{és } i \neq j. \\ \emptyset, \text{ha i} \neq \text{j és } (i,j) \notin E \end{cases}$

Tegyük fel, hogy *i* csúcsból el szeretnénk jutni *j* csúcsba. Egyik lehetőség, hogy közvetlenül azt érjük el, közben legfeljebb az 1 … $(k-1)$ csúcsokat érintjük. Másik lehetőség, hogy a *k.* csúcsot is érintjük. Utóbbi esetben *k*-ig (értelemszerűen) érinthetjük a $1..(k-1)$ csúcsokat, ugyanígy a *k*-tól is. Ekkor megmondhatjuk, hogy ha találtunk javító utat, akkor ezen esetekben: $D_{i,j}^{(k-1)} > D_{i,k}^{(k-1)} + D_{k,j}^{(k-1)}$, akkor $D_{i,j}^{(k)} = D_{i,k}^{(k+1)} + D_{k,j}^{(k-1)}$, valamint $\pi_{i,j}^{(k)} = \pi_{k,j}^{(k-1)}$. Különben $D_{i,j}^{(k)} = D_{i,j}^{(k-1)}$, és $\pi_{i,j}^{(k)} = \pi_{i,j}^{(k-1)}$.

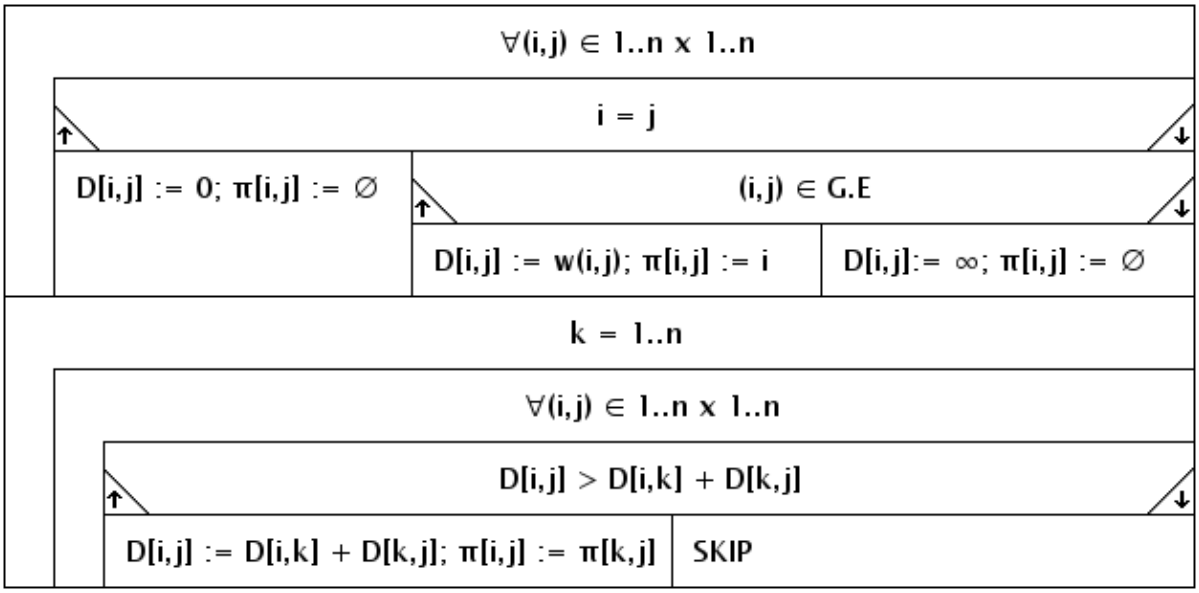
Ezek szerint az algoritmusnak *n+1* darab mátrixpárt kell meghatároznia:

$$< \left(\left(D_{i,j}^{(k)} \right)_{i,j=1..n}, \left(\pi_{i,j}^{(k)} \right)_{i,j=1..n} \right) | k = 1..n >$$

Memóriaigényt nézve ez elég riasztó lesz. Ha mondjuk $n = 1000$, akkor máris ott az aggodalom, hogy elég lesz-e a számítógép memóriája. Azonban meg lehet nyugodni, hogy nincs szükség sok memóriára. Ugyanis $D_{i,k}^{(k)} = D_{i,k}^{(k-1)}$. Ha mondjuk viszont *j* helyére *k*-t írok, akkor $\pi_{i,k}^{(k)} = \pi_{i,k}^{(k-1)}$. Hasonló esetben $D_{k,j}^{(k)} = D_{k,j}^{(k-1)}$, és $\pi_{k,j}^{(k)} = \pi_{k,j}^{(k-1)}$.

Ezek után tekintsük az algoritmust.

FLOYD_WARSHALL$(G, w, D[1..n, 1..n], \pi[1..n, 1..n])$



Az első ciklusban előállítjuk a $\left(D^{(0)}, \pi^{(0)}\right)$ mátrixpárt.
A második ciklusban az az értékadás lesz, hogy $\left(D^{(k)}, \pi^{(k)}\right) \leftarrow \left(D^{(k-1)}, \pi^{(k-1)}\right)$.

Az első inicializáció nagyban függ a gráftól, hogy az milyent vesz fel.

Műveletigény: Az inicializálás műveletigénye $\theta(n^2)$ lesz, míg a második résznél a gráf összes pontján kell végigmenni, ez nyilván $n + \theta(n^2) = \theta(n^3)$ alkalommal történik meg, így a közös műveletigény:

$$T_{FW}(n) \in \theta(n^3)$$

Érdemes lenne meggondolni, ez hogyan is viszonyul a Dijkstra-algoritmushoz? Utóbbi nem tudja azt, mint ez, mert míg a Dijsktra-algoritmus 1 csúcsból határozza meg a legrövidebb utat, bár ha mindegyik csúcsból elindítom, akkor előbb-utóbb megkapom ugyanezeket a mátrixpárokat. Ha az első csúcsból indítom, akkor a mátrixok első sorát számolja ki, ha a másodikból, akkor a második sort számolja ki, és így tovább. De még ebben az esetben is az az alternatív megoldás, hogy n alkalommal elindítjuk a Dijkstra algoritmust. Ekkor kapunk egy bináris kupacot, és egy szomszédossági éllistát, aminek műveletigénye $n * O\big((n + e) \log n\big)$. Ez sűrű gráfnál elég nagy műveletigénnyel fog dolgozni. Ha azonban $e \in O(n)$, akkor ez meg fog egyezni $O(n^2 \log n)$-es műveletigénnyel. Tehát akkor már nem csak bináris kupac, szomszédossági éllista, hanem ritka gráf tulajdonságot is felteszünk, akkor dominálni fog, hogy a Dijkstra ez esetben egyszerűbb (meg hogy kisebbek a konstans szorzók). Azonban ha sűrű a gráf, akkor látjuk, hogy a prioritásos sort rendezetlen vektorban tároljuk, azaz megint vesszük n-szer a Dijkstrát. A szomszédossági éllista, vagy csúcsmátrix itt most lényegtelen. Ekkor $n * O(n^2) = O(n^3)$ lesz a művelet igénye. Azonban ha $e \in \theta(n^2)$, akkor $n * O(n^2 \log n) = O(n^3 \log n)$. Ezt így tehát nem választjuk, mert tl költséget.

Nagyságrendileg látszik, hogy a kettő tehát nem felel meg egymásnak, és hogy a Floyd-Warshall algoritmus mennyivel egyszerűbb. Célszerűbb tehát ezt választani, ha sűrű a gráf, vagy az n nem túl nagy.

Van még a Bellman-Ford algoritmus is, amelyet valamelyik előző előadáson vettünk. Ha ezt n-szer vesszük, akkor ugyanígy gráfoknál $n * O(n * e)$ műveletigénnyel számolhatunk ritkább gráf esetén, azonban megint kitétel, ha $e \in O(n)$, akkor a műveletigény már $O(n^3)$ lesz. Viszont ha $e \in O(n^2)$, akkor $O(n^4)$-nel számolhatunk ismét, ezt nem választjuk!

Most vegyünk egy másik esetet, ahol a $G = (V, E)$ egy tranzitív lezárt, és a $T_{i,j} = \exists i \to j$ út. $(i, j \in 1..n)$. Ez az egész Warshall-algoritmusa, tehát az előzőből fognak visszaköszönni még dolgok. Definiáljuk még a $T_{i,j}^{(k)} = \exists i \to k$ út, ahol érintünk még közben csúcsot $1..k$ között. Kapásból mondhatjuk, hogy következik ebből, hogy $T_{i,j}^{(0)} = (i = j$ vagy $(i, j) \in E)$. Következő következmény, hogy $T_{i,j}^{(k)} = T_{i,j}^{(k-1)}$ vagy $\left(T_{i,k}^{(k-1)}$ és $T_{k,j}^{k-1}\right)$. Azaz *i*-ből *j*-be kétféleképpen tudunk eljutni, vagy érintettünk már csúcsot $1..(k-1)$ között, vagy már egy *k*csúcsot érintettünk (amely előtt és után érinthettünk más egyéb csúcsot. Ekkor elmondhatjuk, hogy $T_{i,k}^{(k)} = T_{i,k}^{(k-1)}$. Hasonlóképpen $T_{k,j}^{(k)} = T_{k,j}^{(k-1)}$.

Nézzük az algoritmusát.

TRANZ_LEZ$(G, T[1..n, 1..n])$

```
∀(i,j) ∈ 1..n x 1..n
    T[i,j] := (i=j ∨ (i,j) ∈ G.E)
k := 1..n
    ∀(i,j) ∈ 1..n x 1..n
        T[i,j] := T[i,j] ∨ (T[i,k] ∧ T[k,j]
```

Az első ciklus előállítja a $T^{(0)}$ mátrixot.
A második ciklus magjától meg azt várjuk, hogy a $T^{(k)} \leftarrow T^{(k-1)}$.

<u>Műveletigény:</u> Az inicializálás nyilván $\theta(n^2)$ lesz, míg a második ciklus $n * \theta(n^2) = \theta(n^3)$, tehát miután ezt összeadjuk, megkapjuk, hogy

$$T_{TL}(n) \in \theta(n^3)$$

Ha a szélességi gráfkeresést lefuttatjuk *n*-szer, azzal az egyszerűsítéssel, hogy nem határozzuk meg az utakat, csak megmondjuk, hogy van-e, akkor nagyságrendileg majdnem ott hagyja az algoritmust, ahol volt. Az $n * BFS$ műveletigénye $T(n, e) = n * O(n + e)$ lesz, ami ritka gráfoknál $\left(e \in O(n)\right)$ azt fogja jelenteni, hogy $T(n) \in O(n^2)$. Sűrű gráfokra viszont $\left(e \in \theta(n^2)\right)$ az *e* fogja lenyelni az *n*-et, ami után azt kapjuk, hogy $T(n) \in O(n^3)$. Ha sűrű a gráf, akkor egy csúcsból túlnyomó részben a többi csúcs is elérhető. A jobb konstans szorzó miatt a tranzitív lezárt algoritmusa tehát szerencsésebb lesz sűrűbb gráf esetén.

Tegyük fel, hogy van a következő gráfunk. Rajz híján ismét szöveges magyarázat:

$$1 \to 2$$
$$2 \to 3$$
$$3 \to 1$$

Warshall-algoritmust alkalmazunk rá. Ekkor a $T^{(0)} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$ $(\uparrow := 1, \downarrow := 0)$. Ehhez képest, ha megnézzük a $T^{(1)}$ mátrixot, azt kapjuk, hogy $T^{(1)} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1^{(*)} & 1 \end{bmatrix}$. Általánosan leírva $T_{i,k}^{(k)} =$

$T_{i,k}^{(k-1)}; T_{k,j}^{(k)} = T_{k,j}^{(k-1)}$. Azaz azt akarjuk meghatározni, hogy mi lesz a mátrix *i.* sorának *j.* oszlopában az adott elem? Nyilván az se mindegy, mi a k értéke. Magyarul, ezt úgy tudjuk könnyen kiszámolni, hogy a hiányzó csúcsot ráállítjuk az első oszlopra, majd az első sorra is, és ha mindkettő érték egyezik, akkor ugyanazt leírjuk. Ezért lett 1-es végül (a mátrixban (*)-gal jelölve, a későbbiekben a változott értéknél is lesz (*) majd). Hasonlóan megy ez tovább: $T^{(2)} = \begin{bmatrix} 1 & 1 & 1^{(*)} \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$. Végül $T^{(3)} = \begin{bmatrix} 1 & 1 & 1 \\ 1^{(*)} & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$.

Mintaillesztések

Brute-force algoritmus

Vannak a következő adataink:

$T[1 \dots n]$: Σ – ez lesz a szöveg
$P[1 \dots m]$: Σ- ez lesz a minta
$\infty > |\epsilon| = d > 0$ és $0 < m \leq n$

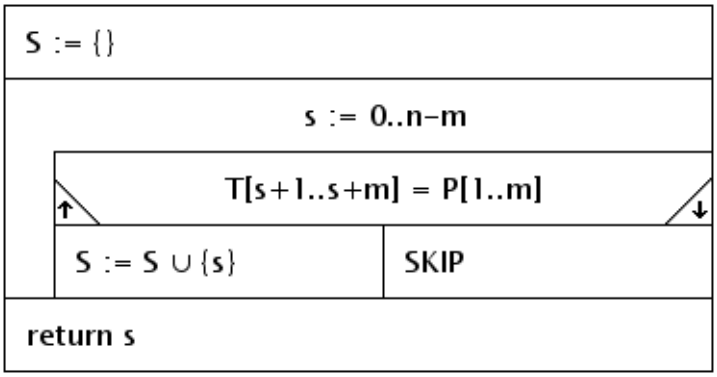
Azaz ne legyen több karakterünk, mint ahol keressük a mintát. Általában az is igaz szokott lenni, hogy a minta sokkal kisebb, mint a szöveg, de mi csak azt tesszük fel, hogy ne legyen hosszabb.
Mi ezt a mintát szeretnénk megkeresni a szövegben, legyen példa:

	1	2	3	4	5	6	7	8	9	10	11	
$T[1 \dots 11] =$	A	B	A	B	B	A	B	A	B	A	B	
$P[1 \dots 4] =$	B(X)	A	B	A								
		B	A	B	A(X)							
			B(X)	A	B	A						
				B	A(X)	B	A					
					B	A	B	A(T)				
						B(X)	A	B	A			
							B	A	B	A(T)		
								B(X)	A	B	A	
									B	A	B	A(V)

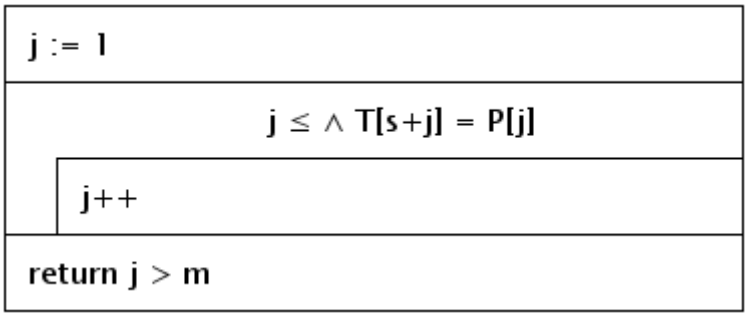
Mit is csináltunk? Elkezdtük nézni a mintánkat, hogy illeszkedik-e a szövegre? Ha nem illeszkedik (a legelső ilyen nem-illeszkedést (X)-szel jelöltük), akkor eggyel jobbra csúsztattuk a mintánkat, és arra ellenőriztük. Ha találtunk egyezést, akkor bekeretezzük (nekem a szövegszerkesztő határai miatt ez nem volt lehetséges, ezért (T)-vel (találat) jelöltem). Ha a mintával kicsúsztunk a szövegből, akkor az ellentmondás lesz (amit én (V)-vel jelöltem). A megoldásunk pedig ezek után: $S = \{4,6\}$. Ezt a megoldást **brute force algoritmus**nak nevezzük, de lehet emellett egyszerű mintaillesztő algoritmusnak is nevezni. Ez az S általánosítva: $S = \{s \in 0..n - m \mid T[s + 1 \dots s + m] = P[1..m]\}$.

Ha már algoritmus, nézzük a stuktogramot:

fv $Egyszerű\_mintaillesztő\_alg(T[1..n], P[1..m])$



$T[s+1 \dots s+m] = P[1..m]$



Műveletigény. A második stuktogram műveletigényei:

$$mT(m) \in \theta(1)$$
$$MT(m) \in O(m)$$

Az elsőnél az első és utolsó sorok értelemszerűen $\theta(1)$-szer fordulnak le. Viszont a ciklus kérdéses. Akkor fut le a legtöbbször, ha minden esetre igazat dob vissza. Ezért meghatározunk minimális és maximális futási időt:

$$mT(n,m) \in \theta(n-m+1)$$
$$MT(n,m) \in \theta\big(m*(n-m+1)\big)$$

A teljes algoritmus futási ideje ezek alapján az lesz, hogy:

$$mT(n,m) \in \theta(n-m+1)$$
$$MT(n,m) \in \theta\big(m*(n-m+1)\big)$$

Elég gyakran fel tudjuk tenni, hogy az n határozottan nagyobb, mint az m, ez esetén pedig mondhatjuk, hogy

$$mT(n) \in \theta(n)$$
$$MT(n) \in \theta(n*m)$$

Sajnos az egyszerű mintaillesztés be tud lassulni, ha viszonylag hosszú szöveget kell keresni. Ezért próbáltak gyorsabb megoldásokat is találni. Az egyik ilyen megoldás a **quick search** lesz.

Quick search (gyorskeresés)

Mindjárt nézünk erre is egy példát.

$$\Sigma = \{A, B, C, D\}$$

$T =$	A	D	A	B	A	D	C	A	D	A	B	C	A	B	A	D	A	C	A	D	A	D	A
$P =$	C_x	A	D	A																			
		$\boldsymbol{C_x}$	**A**	**D**	**A**																		
				$\boldsymbol{C_x}$	**A**	**D**	**A**																	
					$\boldsymbol{C_x}$	**A**	**D**	**A**																
$s = 6$							**C**	**A**	**D**	$\boldsymbol{A_T}$													
												C	**A**	$\boldsymbol{D_X}$	**A**								
														$\boldsymbol{C_X}$	**A**	**D**	**A**						
$s = 17$																		**C**	**A**	**D**	$\boldsymbol{A_T}$		
																				$\boldsymbol{C_x}$	**A**	**D**	**A**

$$snit(P[1..m], shift[\Sigma])$$

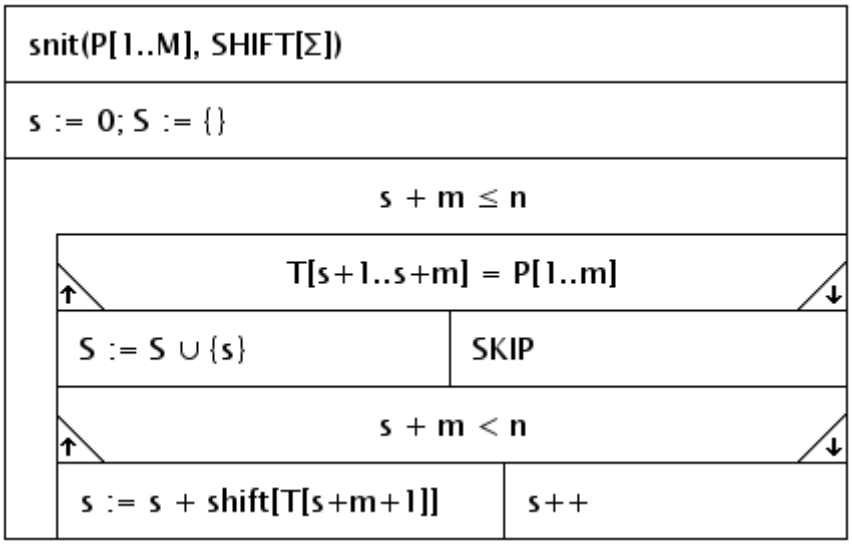
```
∀c ∈ Σ
    shift[c] := m+1
j := 1..m
    shift[p[j]] := m+1−j
```

Műveletigény: Az első ciklus $\theta(d)$, a második $\theta(m)$, a kettő együtt $\theta(m + d)$.

	shift	A	B	C	D
		5	5	5	5
C	1			4	
A	2	3			
D	3				2
A	4	1			
	shift	1	5	4	2

És akkor vissza tudunk térni a fenti táblázathoz (az ezután beszúrt értékek **vastag megjelenésben** lesznek bent). Ebben a táblázatban néztük, hogy a mintánk alapján milyen karakter jött, és az adott karakter oszlopában lévő értéket 1-gyel csökkentettük, ahhoz képest csökkentettük a többi értéket is. Így kaptuk meg az utolsó sorban lévő számokat. A fenti táblázathoz ez úgy kötődik, hogy ha a szövegben egy adott karakterhez érünk, akkor annyit csúsztatunk, amekkora szám van az adott karakter alatt. Helyhiány miatt, a nincs találat, illetve találat jelét alsó indexbe rakom.

$Quick\_Search(T[1..n], P[1..m], S)$



Műveletigény: Első sor nyilván $\theta(n+d)$, második sor $\theta(1)$. A főciklus a kérdés. Ez $(n-m+1)$-szer fog végrehajtódni, maximum. Ezért a ciklus minimum futási ideje:

$$mT(n,m) \in \theta\left(\frac{n}{m+1}\right), \text{pl. ha } T \text{ és } P \text{ diszjunkt.}$$
$$MT(n,m) \in \theta\big(m*(n-m+1)\big) =^{(\text{ha } n \gg m)} \theta(m*n)$$

Végül mekkora lesz az egész ciklus műveletigénye?

$$mT(n,m) \in \theta\left(\frac{n}{m+1}+m\right)$$
$$MT(n,m) \in \theta\big(m*(n-m+1)\big)$$

Haladjunk tovább a tananyagban.

Rabin-karp

Itt másra alapszik az algoritmus. Képzeljük el, hogy az ábécének van egy elemszáma ($|\Sigma| = d$), akkor tudunk csinálni egy olyan leképezést, hogy $\phi: \Sigma \to 0..d-1$. Ha van egy szövegünk, hogy $P[1..m] \sim P_0$, akkor tudunk csinálni egy olyan m-számrendszerbeli számot, hogy:

$$\sum_{j=1}^{m} \phi(P[j]) * d^{m-j}$$

Lesz még egy mintánk, $T[s+1..s+m] \sim T_s$, ez az a szám lesz definíció alapján, hogy

$$\sum_{j=1}^{m} \phi(T[s+j]) d^{m-j}$$

$$S = \{s \in 0..n-m \mid T_s = P_0\}$$

Hogy fog működni ez az egész?

Világos, hogy a P_0, T_0 számokat *Horner-elrendezéssel* kell kiszámolni. Ez $\theta(m)$ időben számolható. Viszont T_s-t hogy kapjuk meg T_{s-1}-ből? Erre kell annak a képlete is:

$$T_{s-1} = \sum_{j=1}^{m} \phi(T[s-1+j])d^{m-j} = \sum_{j=0}^{m-1} \phi(T[s+j]) + d^{m-j-1}$$
$$T_s = (T_{s-1} - \phi(T[s])d^{m-1})d + \phi(T[s+m])$$

Az egész algoritmust szerencsére $\theta(n)$ időben le tud futni. A gond az, hogy ezek viszont hatalmas számok lesznek. Amíg a 64 bites integer aritmetikába belefférnek, addig még jó, de azután viszont csak gondok lesznek. („Persze az alma belefér, de egy arab vagy spanyol nemes neve már nem.").

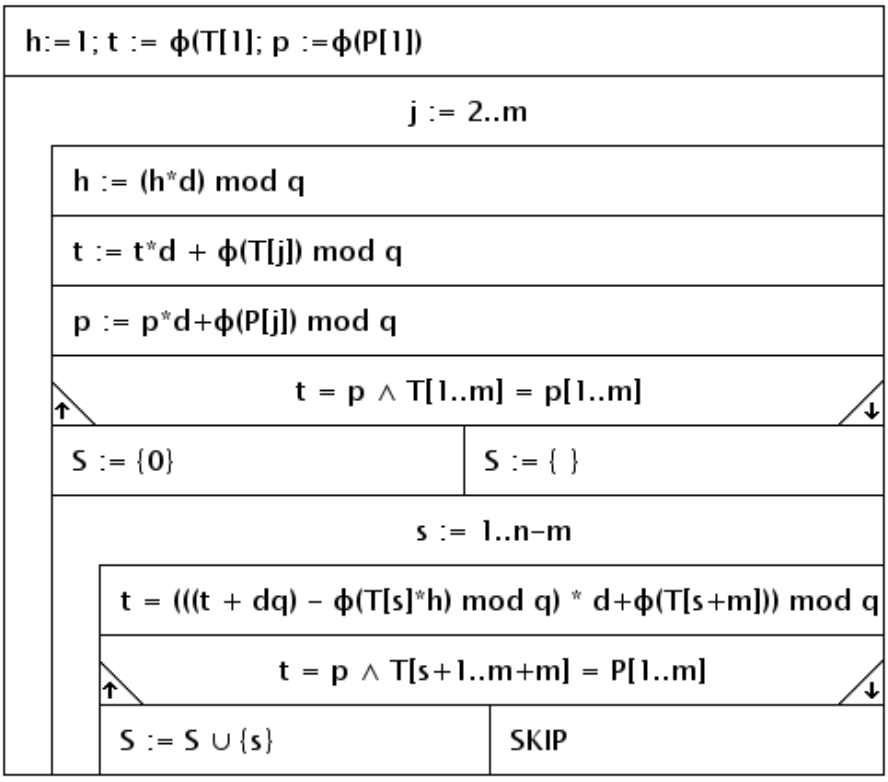
Fogunk definiálni ehhez egy $p \coloneqq P_0 \; mod \; q$. Ehhez jön még egy $t_s \coloneqq T_s \; mod \; q$, valamint $h \coloneqq d^{m-1} mod \; q$. Itt a q egy nagy prímszám. A $t_s \coloneqq \big((t_{s-1} - \phi(T[s]) * h) * d + \phi(T[s+m])\big) \; mod \; q$.

$$t_s = \Big(\big((t_{s-1} + dq - \phi(T[s])) \; mod \; q\big) * d + \phi(T[s+m])\Big) mod \; q$$

Tehát a P mintához asszociálunk. Bijektív módon lesz definiálva egy $P[1..m]{\sim}p = \big(\sum_{j=1}^{m} \phi(P[j])d^{m-j}\big) \; mod \; q$. Ez a ϕ függvény bijektív módon megfelelteti az ábécét ezen a módon: $\phi: \sigma \to 0..d-1$. A q itt egy prímszám lesz, valamint még az a feltétel is teljesül rá, hogy $1 < q \leq \frac{\max N}{d+1}$. Persze a q a lehető legnagyobb prímszám kell, hogy legyen. Minél nagyobb ez a q, annál kevesebb string lesz leképezve arra a P-re. Lesz még a $T[s+1..s+m]$, ez akár illeszkedhet is a p-re. Ez így egy lehetséges megoldás. Ehhez is társítunk egy számértéket, ez pedig a $T[s+1..s+m]{\sim}t_s = \big(\sum_{j=1}^{m} \phi(T[s+j]) * d^{m-j}\big) \; mod \; q$. Erre azt mondhatjuk, hogy azt az S halmazt akarjuk kiszámolni, hogy $S = \{s \in 0..n-m \mid p = t_s \text{ és } T[s+1 \ldots s+m] = P[1 \ldots m]\}$. Jogos a kérdés, hogy az első feltétel miért szükséges? Ugyanis ha $p \neq t_s$, akkor nem lesz egyenlőség a második feltételben. Míg ha egyenlő a kettő szám, akkor lesz meg az egyenlőség a másik oldalon is.

Nézzük meg az algoritmusát is.

$RABIN\_KARP(T[1 \ldots n], P[1 \ldots m], S)$



Első ránézésre bárki azt mondaná, hogy ez semmiben sem jobb, mint a *Brute force* algoritmus. A valóság az, hogy ez leginkább rövid kiértékelésnek mondható. Ezért a legtöbb esetben a string összehasonlításig el se jut a program. Ha véletlenül egyenlőek, akkor kiírja, hogy van találat, de sok esetben nem jut el odáig.

Műveletigény: A kezdő ciklus az $\theta(n)$, az *igaz-hamis* része $O(n)$, a második, belső ciklus teljes műveletigénye jó esetben, ha egyszer se történik string összehasonlítás, akkor $\theta(n-m)$ minimális esetben. Maximális esetben viszont $\theta\big(n(n-m)\big)$. Ha a teljes algoritmusra nézzük, akkor összesen:

$$mT_{RK}(n,m) \in \theta(n)$$
$$MT_{RK}(n,m) \in \theta\big(m(n-m+1)\big)$$
$$AT_{RK}(n,m) \approx mT_{RK}(n)$$

Persze ez csak általában van így, lehet rá mondani ellenpéldát. A maximális mindig fog tudni realizálódni. Ha például a szöveg csupa *A* betűből áll, a mintánk is csupa *A*, csak kevesebb, akkor a maximális futási idő fog előfordulni. Ha viszont a mintánk és a szövegünk diszjunktak, akkor a minimális futási idő fog előfordulni. A minimális tehát csak *P, T* diszjunkt szövegek esetén fog előfordulni.

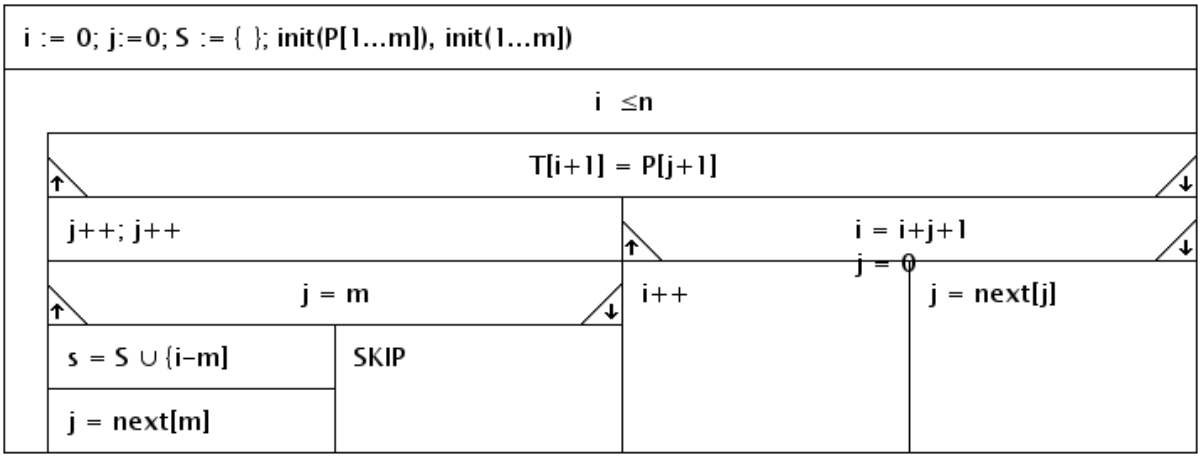
Most pedig nézzük a következő algoritmust.

Knuth-Morris-Pratt algoritmus

$P[1\ldots8]=$	B	A	B	A	B	B	A	B										
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
T[1…18]=	A	B	A	B	A	B	A	B	B	A	B	A	B	A	B	B	A	B
	B_X																	
		B	A	B	A	B	B_X											
$s=3$				B	A	B	A	B	B	A	B_T							
									B	A	B	A	B	B_X				

Mit is csináltunk? A *P* mintánkat a *T* string alá próbáljuk beszúrni, közben figyeljük, hogy van-e egyezés a kettő között. Ha nincs, akkor egy sorral lejjebb megyünk, és a rá következő karaktertől csináljuk ugyanezt (ahol nem volt egyezés, ott alsó indexbe *X*-et írtunk). Közben azt is figyeljük, hogy az első karakterünk hol fordult elő legközelebb a mintánkban, és onnantól folytatjuk ebben az esetben. Amennyiben volt találatunk, ott ezt alsó indexben *T*-vel jelöltem. Ezután hogy határozzuk meg a mintánk prefixét. Ilyenkor meg kell határozni egy minimális eltolást. Mindenesetre ilyenkor meg kell nézni, melyik prefixre illeszkedik.

$$KMP(T[1..n], P[1..m], S)$$



Lesz majd még egy olyan leképezésünk is, hogy $next: 1..m \to 0..m-1$, illetve $next(j) = \max\{h \in [0..j-1] | P[1..n] = T[1..i]_n\}$, ahol $T[1..i]_n = P[1..j]_n$, ezekből pedig következik, hogy $next(j) \in j-1, 0 \leq next(j) < j$. Ezekből tovább következik, hogy $next[1..m] = next(1..m)$. Némi átalakítás után pedig $next(j) = \max\{h \in [0..j-1] | P[1..n] = = P[1..j]_n\}$. Ezek alapján próbáljuk meg kitölteni a következő táblázatot.

$P =$	B	A	B	A	B	B	A	B
j	1	2	3	4	5	6	7	8
$next(j)$	0	0	1	2	3	1	2	3
			B	A	B	A		
				B	A	B		
					B	A		
						B	A	B

Már csak az algoritmus műveletigénye van hátra.

Műveletigény:

$$T(n,m) \in \theta(m) + \omega(m) = \omega(m)$$
$$t(i,j) = 2i - j \in 0..2n, \text{szigorúan monoton növő}$$
$$T(n,m) \in O(2n) = O(n) + \theta(m)$$
$$T(n) \in \theta(n)$$

Karaktertömörítések

$\sigma = \{\delta_1, \dots, \delta_d\}, \sigma^*, P[1 \dots m], T[1..n]$, ezek voltak ugyebár az adataink. Ha volt egy *x*, ami prefixe *y*-nak, akkor definíció alapján $\exists z: xz = y$. Hasonló a szuffixnél, csak ott annyi az eltérés, hogy $\exists z: zx = y$. Az *x* string hosszát így jelöljük: $|x|$. Üres stringnek értelemszerűen 0 a hossza, míg $|abc| = 3$. Érdemes megnézni néhány tulajdonságát is megnézni ezeknek a szuffixeknek, prefixeknek:

1. Ha van egy *x* prefixe *z*-nek, illetve *y* prefixe *z*-nek is, akkor három eset van:
   a. Ha $|x| < |y|$, akkor *x* az *y* prefixe is egyben.
   b. Ha $|x| > |y|$, akkor *y* az *x* prefixe is egyben.
   c. Ha $|x| = |y|$, akkor a két prefix megegyezik.
2. **Definíció.** Tegyük fel, hogy $H \subseteq \mathbb{Z}$, ami véges, akkor van értelme beszélni a $\max_k(H)$-ról, ami azt jelenti, hogy *H* k-adik legnagyobb eleme.
   Definíció. Tegyük fel, hogy $P[1..m]: \Sigma$. Ekkor definíció szerint:
   $$H(j) = \{h \in 0..j-1 \mid P[1..h] \text{ szuffixe } P[1..j]\}$$
   $$next(j) = \max H(j)$$
   $$next_k(j) = \max_k H(j)$$
   $$next^k(j) = next\big(next(\dots next(j) \dots)\big), \text{a jobb oldalnál } k - \text{szor alkalmazva}$$
   Definíció. *x* prefix-szuffix párosa *y*-nak, ha *x* prefixe és szuffixe is *y*-nak, viszont $x \neq y$.
3. Egy $P[1..k]$ string prefix-szuffix párosa a $P[1..j]$ stringnek, ha
   a. $h < j$ és;
   b. $P[1..h]$ szuffixe $P[1..j]$-nek.
4. Mindig igaz lesz, hogy $0 \leq next(j) < j$. Csak akkor lesz 0, ha üres a string.
5. Egy $P[1..h+1]$ akkor lesz szuffixe $P[1..j+1]$ stringnek (feltettük ,hogy $j < m$), ha
   a. $P[1..h]$ szuffixe $P[1..j]$ és;
   b. $P[j+1] = P[h+1]$.
6. $0 \leq next(j+1) \leq next(j) + 1$. A *next()* függvény ugyanis sose ugrik 1-nél többet. Az, hogy csökkenhet akár 0-ra is, azt könnyű belátni. Íme egy példa:
   a. $P = ABAC, next(3) = 1, next(4) = 0$.

b. $P = ABABABC, next(6) = 4, next(7) = 0.$

BIZONYÍTÁS.

Tegyük fel, hogy $next(j+1) > next(j)+1$, ekkor $i \coloneqq next(j+1)-1$, azaz $i+1 > next(j)+1$. Nézzük meg, mi a helyzet a $P[1..i+1]$ stringgel. Ez pont a $P[1..next(j+1)]$ string lesz szükségszerűen, ami szuffixe $P[1..j+1]$-nek. Erre rajzoltunk fel pár példát is, ezeket nem fogom tudni sajnos lerajzolni, de a végeredményben az jött ki, hogy $P[1..i]$ szuffixe $P[1...j]$, azaz $i > next(j)$. Viszont láthatjuk, hogy ez ellentmondás. A *next* definíciójából az jön ki, hogy $i \leq next(j)$, ugyanis utóbbi a leghosszabb $P[1..h]$ alakú szuffix hossza, *i*-nek pedig (~~asszem)~~ mindenféleképpen kisebbnek kell lennie.

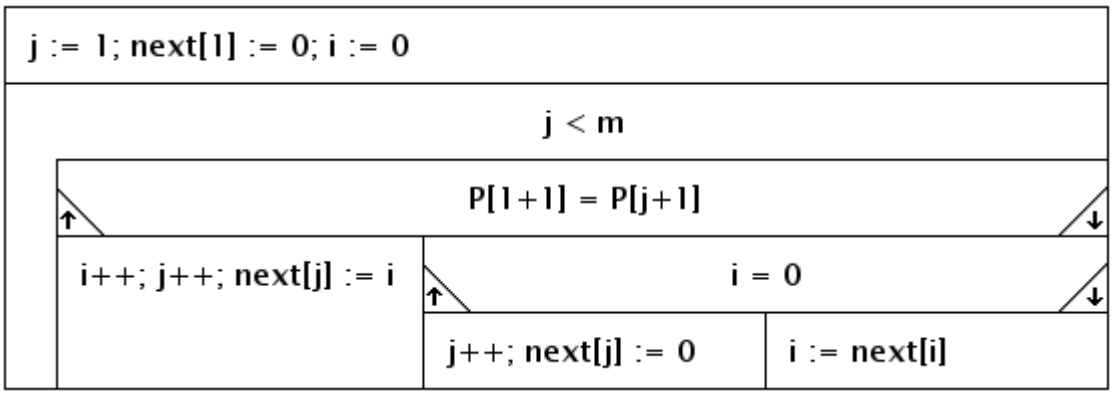
7. (NB.) Ha a $next_k(j)$ értelmezve van, akkor igaz, hogy:

$$next^k(j) = ^{(j>next(j)>next^2(j)>\cdots>next^{k_0}(j)=0)} = next_k(j).$$

Bizonyítás véges teljes indukcióval történik, k szerint, amelyet most nem teszünk meg. $(k \in 1..k_0)$

Miért is mondtuk ki ezt a sok tulajdonságot? Az a célunk, hogy szeretnénk meghatározni egy tömböt, ahol $next[1..m] = next(1..m)$, azaz a *next*() függvény értékeit bemásolom egy *next[]* tömbbe. Ezen tömb értékei sorban a függvény értékei lesznek .Erre a tömbre lesz szüksége az algoritmusunknak. Egy ilyen tömböt ki lehetne tölteni a definíció szerint, de a műveletigénye akkor $\theta(n^3)$-ös lenne. Ennél hatékonyabb, lineáris algoritmusok is léteznek. Ehhez kell majd az *init()* nevű eljárás.

$$init(P[1..m], next[1..m])$$



Megjegyzések: Első sorban ugyebár teljesülnie kell $0 \leq next(1) < 1$ feltételnek. Második sorban pedig egy invariáns műveletnek kell történnie, ahol $0 \leq i < j \leq m$, és $next[i..j] = next(i..j)$, emellett még $P[1..i]$ szuffixe $P[1..j]$-nek. Ez utóbbi általában is igaz lesz. Nyilván ha $P[1..i]$ szuffixe $P[1..j]$-nek, illetve $P[i+1] = P[j+1]$, akkor teljesül, hogy $P[1..i+1]$ szuffixe lesz $P[j+1]$-nek. Ezért kell egy 4. tulajdonságnak is teljesülnie, ami úgy szól, hogy:

$$\forall h\text{: ha } i < h < j \text{ és } P[1..k] \text{ szuffixe } P[1..j] - \text{nek, akkor } P[h+1] \neq P[j+1].$$

Nézzünk akkor most már konkrét példát is, és jöhet mellé a gyönyörűséges táblázat is:

			1	2	3	4	5	6	7	8	m = 8
i	*j*	*next(j)*	A	B	A	B	B	A	B	A	
0	1	0		A_X							
0	2	0			A						
1	3	1				B					
2	4	2					A_X				
0	4	2					A_x				
0	5	0						A			
1	6	1							B		
2	7	2								A	
3	8	3									j = m

Magyarázat nem szükséges szerintem, mert a stuktogram egyszerűsége miatt lehet követni. (Ha valami mégsem tiszta, kommentben jelezzétek nyugodtan!)

Ezek után megkapjuk a következő táblázatot:

j	1	2	3	4	5	6	7	8
P[j]	A	B	A	B	B	A	B	A
next[j]	0	0	1	2	0	1	2	3

A főciklus legalább $m-1$-szer lefut, ebből az következik, hogy

$$T(m) \in \Omega(m)$$

Vegyük figyelembe, hogy $t(j,i) \coloneqq 2j-1$, ekkor $t: [1..m]x[0..m-1] \to 2..2m$, ekkor pedig $\forall$ciklusmag végrehajtással szigorúan monoton növekvő. Így viszont az algoritmus maximum $(2m-2)$-szer hajtódik végre:

$$T(m) \in O(m)$$

A teljes műveletigény ezek alapján az lett, hogy:

$$T(m) \in \theta(n)$$

A hátralévő időben vezessük be a következő előadás anyagát. Vannak a *zip, unzip* fájlok, azaz az alapfájlok állnak egy hosszú-hosszú bitsorozatok, ezeket kellene tömöríteni. Ehhez az adataink:

$T[1..m]: \Sigma$, ahol $\Sigma = \{\delta_1, \dots, \delta_d\}$. Ha van $k$darabszámú bitünk, akkor 2^kkülönböző jel ábrázolható. Ekkor $2^k \geq d$, azaz $k \geq \lceil \lg d \rceil$, optimálisan $k \coloneqq \lceil \lg d \rceil$.

Például $|\Sigma| = 26 = d$, ekkor $\lceil \lg d \rceil = \lceil \lg 26 \rceil = 5$. Tegyük fel, hogy 8 bit volt, ekkor $\frac{5}{8}$ a tömörítési arány.

Huffman-kód

Egy klasszikus az úgynevezett **Huffman-kód**. Erre nézzünk egy példát:

ABRAKADABRA. Ezt szeretnénk tömöríteni. Először is számoljuk meg, hogy a különböző karakterek hányszor fordulnak elő.

A	5
B	2
R	2
K	1
D	1

Ez épp rendezve is van, mármint az előfordulásokat tekintve. Ha nem lettek volna, rendezni kellett volna. Ezután kell egy kódfát építeni a következő módon: $A-5, B-2, R-2, K-1, D-1$. Ezek lesznek a kódfa levelei. Vesszük a két legkisebb előfordulási sűrűséget, és azt mondja, hogy azok

együtt ketten kétszer fordulnak elő. Aztán vesszük megint a két legkisebbet, amiből pont három eset is van, ezért az *R* és a *B* betűk összesen 4-szer fordulnak elő. Megint a két legkisebb a 2, és a 4, azok együtt 6-szor fordulnak elő. Már csak két értékünk van a kettő együtt 11-szer fordul elő. Ezek alapján ki is jött a kódfa. Így az történt, hogy a gyakrabban előforduló esetekhez alacsonyabb fa társul, míg a ritkábban előfordulókhoz magasabb. Természetesen jelen esetben lett volna más megoldás is.

Ezek után *selector*okat társítunk a részfákhoz. Ilyenkor a gyökérből indulva a bal részfához 0, míg a jobb oldalihoz 1 van. Belül ugyanezt csináljuk, a bal oldali részfa megint 0, a jobb oldali részfához 1 van. Ugyanez a továbbiaknál, ugyanezekkel a szabályokkal. Míg minden karakterhez van egy saját kód. Ez egy úgynevezett prefix kód, ebben nem fordul elő olyan, hogy egy kód egy másik kód prefixe legyen. Ez azért jó, mert visszafejtésnél ha megvan a kódfa, amelyet úgy hozunk létre, hogy nézzük ezeket a selectorokat:

A	B	R	A	K	A	D	A	B	R	A
0	100	101	0	110	0	111	0	100	101	1

Azaz 0 100 101 0 110 0 111 0 100 101 1 lesz a kódfánk. Ennek a bitszáma pedig $5 + 6 * 3 = 23$ bit. Ha naiv kódolással csináltuk volna, akkor $11 * 3 = 33$ bit lenne a bitszám, azaz máris jobban tömörítettünk a Huffmannal.

Akkor most már nézzük, hogy csináltuk meg a visszatömörítést. Ugyebár megvan ez a bitsorozat. Egyszerűen úgy, hogy elkezdjük olvasni a kódjainkat. A *0* lesz az *A*, a *100* a *B*, és így tovább. Van azonban egy megszorításunk, hogy akkor optimális a Huffman-kód, hogy ha beszorítjuk a kódot, hogy fix számú biteket tömörít egymás után.

Lempel-Ziv-Welch algoritmus

Ha jobb tömörítést akarunk, akkor más elvet kell nézni. A másik tömörítési mód, ami egyébként a *ZIP, UNZIP* algoritmus lényegét tartalmazza, azt három neves matematikusnak köszönhető. Ezért lesz az algoritmusunk a **Lempel-Ziv-Welch algoritmus**, azaz röviden **LZW**. A *ZIP* tömörítés alapelveit fektették le ezzel az algoritmussal. Ezért szakítania kellett a Huffman-féle karakterenkénti tömörítéssel. Lényege, hogy változó hosszú stringeket tömörít be fix hosszúságú sorozattá. Mivel mi 1-től fogunk kódolni, ezért $2^{12} - 1 = 4095$. Tehát ennyiféle kódot fogunk tudni kreálni, mivel kétbites kódokkal fogunk dolgozni. Fontos, hogy ez fix hosszú kódokat képez.

Nézzünk rá egy példát. Ha *k* bites a kódunk, akkor a $MAXKÓD = 2^k - 1$, ez egy *globális konstans*. Nézzük most már a konkrét példát. Van egy inputom és egy outputom.

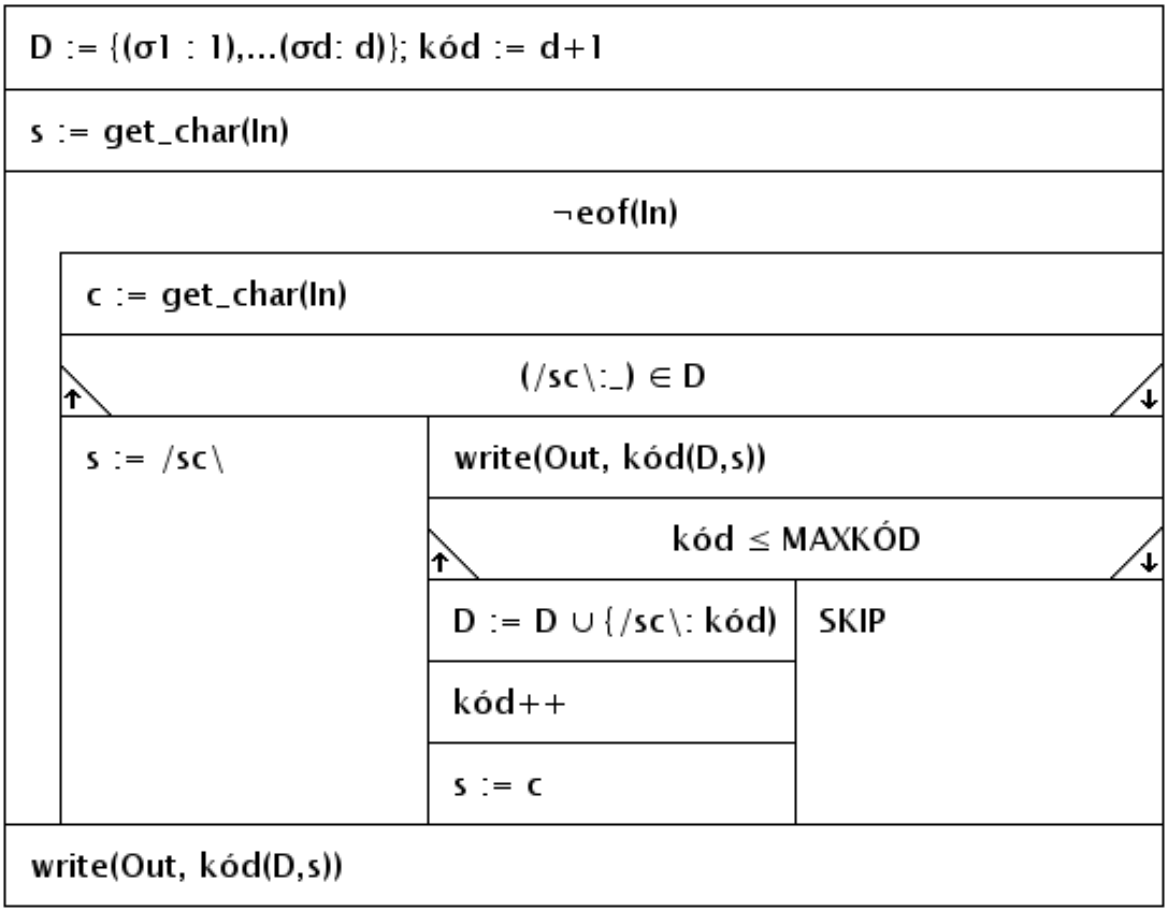
input:	a	b	a	b	c	b	a	b	a	b	a	a	a	a	a	a	a
output:	1	2	4		3	5		8			1	10		11			1

szó	kód
a	1
b	2
c	3
ab	4
ba	5
abc	6
cb	7
bab	8
baba	9
aa	10
aaa	11
aaaa	12

Mit is csináltunk? A kódoláshoz kell tudni, hogy mi is az ábécénk. Ez a $\Sigma = \{a, b, c\}$. Azt mondja az algoritmus, hogy felveszünk még egy szótárat is. A szótárban vannak szavak, és kódok. Mivel három karakterünk már volt, azoknak meg is van adva a kódja. Látjuk, hogy az *a* kódja 1, utána megpróbál hosszabb stringeket megtalálni a szótárban. Az *ab* kódja nincs meg, ezért azt felveszi. Kiírjuk a *b* kódját. Miért is? Mert a *ba* kódját se találta meg a szótárban, ezért hozzáveszi. Megyünk tovább, *a*-t megnézi, majd továbbnézi, *ab*-t megtalálta, *abc*-t nem, ezért előbbinek leírja a kódját, utóbbit hozzáveszi a szótárba. Így megyünk tovább. A *c* benne van, a *cb* nincs, előbbi kódját leírja, utóbbinak kódját felveszi. Ezt így ismételjük tovább, amíg meg nem kapjuk a végleges kódot.

Nézzük, hogy néz ki az algoritmusunk:

$LZW\_COMRESS(In, Out, \Sigma)$ //MAXKÓD konstansunk is van



Megjegyzés: Az ábécénk általános jelölése: $\Sigma = \{\sigma_1, \dots, \sigma_d)\}$, a $\underline{\sigma_i}$ jelölésünk is van, illetve a szótár $D \subseteq \{(s: k) \mid s \in \Sigma^+ \text{ és } k \in 1..MAXKÓD\}$. A program során $t, s \in \Sigma^+$, az s_1 pedig egy s string első karaktere lesz. Lesz még egy kód = első szabad kód képletünk is. Σ^+ annyit jelent, hogy ez egy nemüres string. Még egy jelölésünk lesz, hogy $c \in \Sigma$, valamint $\widehat{sc}$ a két szó konkatenáltját jelöli a továbbiakban, amelyet a stuktogramban én /sc\-vel fogok jelölni. Az inputnál feltételezzük, hogy nem üres a string. Az $eof(In)$ függvény akkor dob igazat, ha beolvasta az utolsó karaktert, és több beolvasható karakter már nincs.

LZW-algoritmus visszafelé

Nézzük akkor, hogy működik a visszatömörítés. Szótár nem áll rendelkezésünkre, csak az ábécé: $\Sigma = \{a, b, c\}$.

In:	1	2	4	3	5	8	1	10	11	1
Out:	a	b	ab	c	ba	bab	a	aa	aaa	a

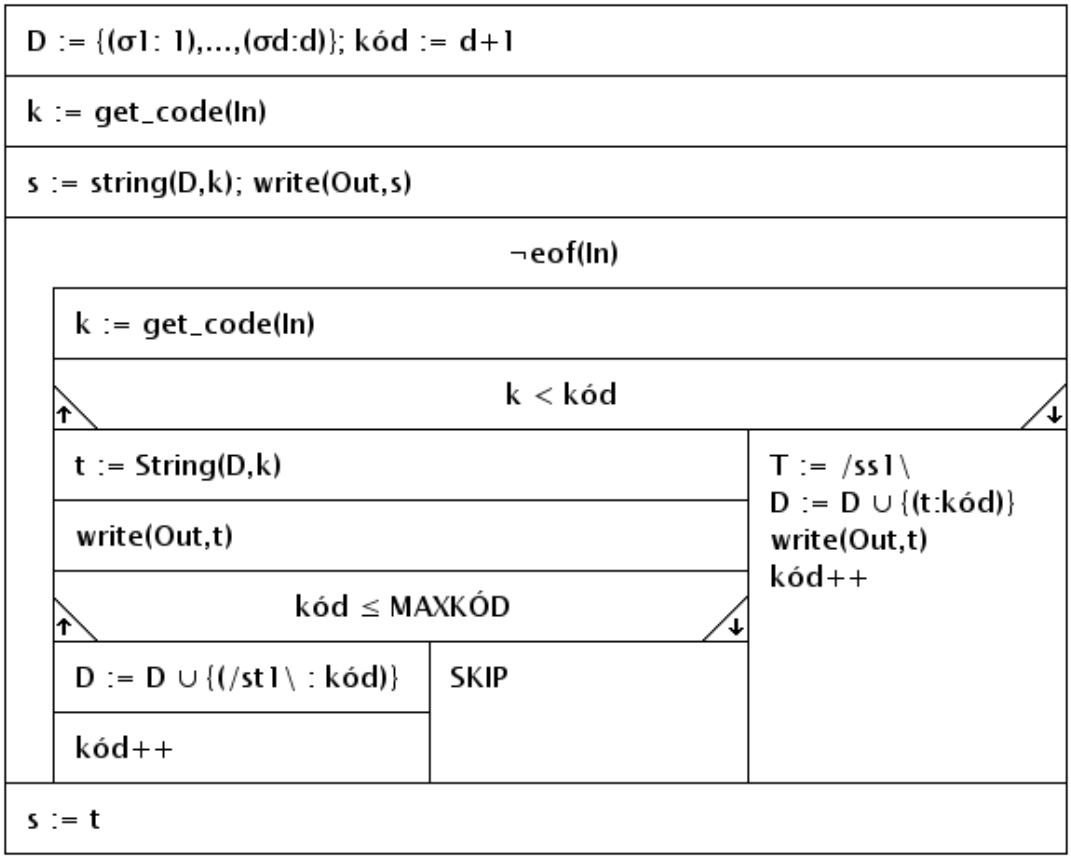
szótár:

szó:	kód:
a	1
b	2
c	3
ab	4
ba	5
abc	6
cb	7
bab	8
baba	9
aa	10
aaa	11

Mit is csináltunk? Az ábécénk kódjait fel tudjuk írni. Ezek alapján az első két kód megvan. Majd látjuk, hogy egy 4-es kód jön, de olyan nincs a szótárban, úgyhogy megnéztük, hogy előtte milyen karakterek vannak. Ez az *ab*, úgyhogy felírjuk. Igenám, de amikor a tömörítő megtalálta az *ab*-t, akkor utána megtalálta a *ba*-t. Ezt megtesszük mi is. Megnézzük a 3-ast, az benne van, beírjuk. De előtte *abc*-t is beleírta a szótárba, tegyük meg mi is. Most jön a feketeleves: olyan kód jön, ami nincs. Megnézzük a korábbiakat. Azt tudjuk, hogy *ba*-val kezdődött. De mi a vége? Nem tudjuk. Úgyhogy megismételjük a stringet, kijön hogy *baba?*, ennek az első három karakterét vehetjük is. Ezután a *baba* karaktert is megtaláltuk, azt is felvesszük a szótárba. Ezután megint olyan kód jön, ami nincs a szótárban. De rutinosak vagyunk. Az *a*? a 10-es kód. De mi van a *?* helyén? De ezután látjuk, hogy ha összekötjük az előtte levővel, kijön hogy *aa*, és ezt rögtön fel is használta. Ezután újra ismeretlen kód jön, ugyanez az ismétlés van. Azt mondjuk, a 11-es (kód) úgy néz ki, hogy *aa*?, mivel nem tudjuk mi jön. De összehasonlítva az előzővel, megjön, hogy *aaaa*?, ebből csak az első három karaktert vesszük, mert az még úgy nem volt, és felvesszük a szótárba. De akkor már fel tudunk venni egy 12-es kódú szót is, ami az *aaaa* lesz. A végén az utolsó kód már könnyű.

Nézzük akkor még meg az algoritmusát, a következő oldalon:

$LZW\_DECOMPRESS(In, Out, \Sigma)$



Megjegyzés: Az inputról feltételezzük, hogy nemüres.