14. Haladó algoritmusok

Gráfalgoritmusok

Gráf ábrázolás

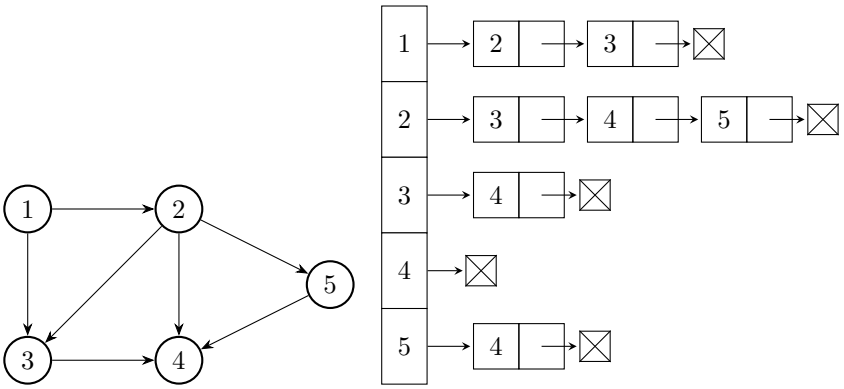
Legyen a továbbiakban $G = (V, E)$ gráf, ahol

- V a csúcsok halmaza ($|V| = n$),
- E pedig a csúcsok közti élek halmaza.

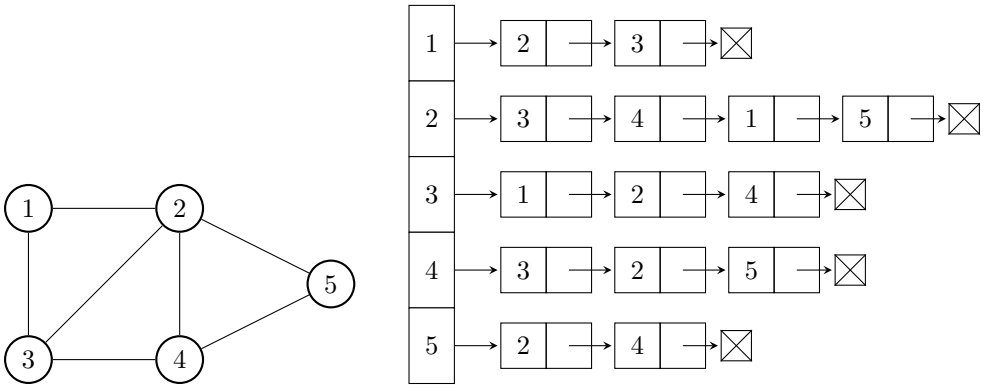
Éllistás ábrázolás

Legyen G véges gráf. Vegyünk fel egy mutatókat tartalmazó $Adj[1...n]$ tömböt (a csúcsokkal indexeljük a tömböt). A tömbben lévő mutatók mutatnak az éllistákra (más néven a szomszédsági listákra). Az éllisták lehetnek egy- vagy kétirányú, fejelemes vagy fejelem nélküli listák, ez most nem lényeges a gráf szempontjából.

- Irányított gráf esetén, az éllisták listaelemei reprezentálják az éleket. Az élnek megfelelő listaelemet abban a listában tároljuk, amelyik csúcsból kiindul az él, és a célcsúcs indexét eltároljuk a listaelemben. Tehát az él megvalósítása az i-edik listában egy olyan listaelem, amelyben eltároltuk j-t, mint az él célcsúcsát.



- Irányítatlan gráf esetén, egy élnek két listaelemet feleltetünk meg, azaz egy irányított élt egy oda-vissza mutató, irányított élpárral valósítunk meg a korábban említett módon. Élsúlyozott gráf esetén, az él súlyát is a listaelemben fogjuk tárolni.



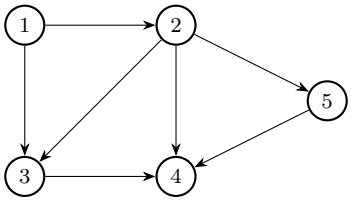
Az éllistás ábrázolás helyfoglalása irányítatlan gráfok esetén a csúcsok számával (Adj tömb), illetve az élek számával (éllista-elemek száma) arányos. Az elfoglalt memória méretének nagyságrendje $(n + e)$. Irányított gráfok esetén az élek számának duplájával kell számolnunk, így $(n + 2e)$-vel arányos helyfoglaláshoz jutunk.
Mivel a memóriaigény az élek számával arányos, ezért az éllistás ábrázolást ritka, illetve nem-sűrű (mondhatnánk 'normál') gráfok ($e \ll n^2$) esetén szokták használni, ugyanis sűrű gráf esetén a szomszédsági mátrixhoz képest itt jelentkezik a listák láncolásából származó helyfoglalás is, a mutatók tárolása révén.

Szomszédsági mátrix

Legyen $G = (V, E)$ véges gráf, és n a csúcsok száma. Ekkor a gráfot egy $n \times n$-es mátrixban ábrázoljuk, ahol az oszlopokat és a sorokat rendre a csúcsokkal indexeljük (ez leggyakrabban $1, \ldots, n$). Egy mezőben akkor van 1-es, ha a hozzá tartozó oszlop által meghatározott csúcs szomszédja a sor által meghatározott csúcsnak.

$$C[i,j] = \begin{cases} 1, & \text{ha } (i,j) \in E \\ 0, & \text{ha } (i,j) \notin E \end{cases}$$

Példa:



$$A^{5\times5} = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}^{5\times5} \Leftrightarrow \begin{bmatrix} \cdot & 1\to2 & 1\to3 & \cdot & \cdot \\ \cdot & \cdot & 2\to3 & 2\to4 & 2\to5 \\ \cdot & \cdot & \cdot & 3\to4 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 5\to4 & \cdot \end{bmatrix}^{5\times5}$$

Ha súlyozott a gráf, akkor az élsúlyokat (élköltségeket) is el kell tárolni. Ezt is a mátrixon belül oldjuk meg. A súly valós számokat vehet fel. Természetesen adódik, hogy ahol előzőleg 1-est írtunk, azaz létezett az illető él, oda most írjuk be az él költségét.

Két további eset maradt, a mátrix főátlója, és az olyan mezők, amelyek esetén nem létezik él. Vezessük be a végtelen (∞) élsúlyt, és a nem létező élek esetén a mátrix megfelelő helyére írjunk ∞-t. Egy ilyen "élen" csak végtelen nagy költséggel tudunk végighaladni (tehát nem tudunk).

A mátrix főátlójába kerülnének a hurokélek költségei, de ilyen értékeket nem alkalmazunk, mivel a legtöbb gyakorlati probléma leírására alkalmas egyszerű gráfokra korlátozzuk. Az egyszerű gráfok nem tartalmaznak hurokéleket (valamint többszörös éleket sem).

Élsúlyozott gráf esetén a szomszédsági mátrix kitöltését a következő megállapodás szerint végezzük:

$$C[i,j] = \begin{cases} 0, & \text{ha } i = j \\ c(i,j), & \text{ha } (i,j) \in E \\ \infty, & \text{ha } (i,j) \notin E \end{cases}$$

A mátrixos ábrázolás helyfoglalása mindig ugyanakkora, független az élek számától, a mátrix méretével n^2-tel arányos. (Az n pontú teljes gráfnak is ekkora a helyfoglalása.) A mátrixos reprezentációt sűrű gráfok esetén érdemes használni, hogy ne legyen túl nagy a helypazarlás.

Szélességi bejárás

A G gráf (irányított/irányítatlan) s startcsúcsából a távolság sorrendjében érjük el a csúcsokat. A legrövidebb utak feszítőfáját adja meg, így csak a távolság számít, a súly nem. A nyilvántartott csúcsokat egy sor adatszerkezetben tároljuk, az aktuális csúcs gyerekeit a sorba tesszük. A következő csúcs pedig a sor legelső eleme lesz. A csúcsok távolságát egy d, szüleiket egy π tömbbe írjuk, és ∞ illetve 0 értékekkel inicializáljuk.
Az algoritmus:

1. Az s startcsúcsot betesszük a sorba
2. A következő lépéseket addig ismételjük, míg a sor üres nem lesz
   - Kivesszük a sor legelső (u) elemét
   - Azokat a gyerekcsúcsokat, melyeknek a távolsága nem ∞ figyelmen kívül hagyjuk (ezeken már jártunk)
   - A többi gyerekre (v): beállítjuk a szülőjét ($\pi[v] = u$), a távolságát ($d[v] = d[u] + 1$), majd berakjuk a sorba.

Minimális költségű utak keresése

Dijkstra algoritmus

Egy G irányított, pozitív élsúlyokkal rendelkező gráfban keres s startcsúcsból minimális költségű utakat minden csúcshoz.

Az algoritmus a szélességi bejárás módosított változata. Mivel itt egy hosszabb útnak lehet kisebb a költsége, mint egy rövidebbnek, egy már megtalált csúcsot nem szabad figyelmen kívül hagyni. Ezért minden csúcs rendelkezik három állapottal (nem elért, elért, kész). A d és π tömböket a szélességi bejáráshoz hasonlóan kezeljük.

A még nem kész csúcsokat egy prioritásos sorba helyezzük, vagy minden esetben minimumkeresést alkalmazunk.

Az algoritmus:

1. Az s startcsúcs súlyát 0-ra állítjuk eltároljuk
2. A többi csúcs súlyát ∞-re állítjuk
3. A következő lépéseket addig ismételjük, míg a konténerünk üres nem lesz
   - Kivesszük a sor legjobb (u) elemét, és "kész"-re állítjuk
   - Ha egy gyerekcsúcs (v) nem kész, és a jelenleg hozzávezető út súlya kisebb, mint az eddigi, akkor: a szülőjét u-ra állítjuk ($\pi[v] = u$), és a súlyát frissítjük ($d[v] = d[u] + d(u, v)$).
   - A többi csúcsot kihagyjuk.

A minimális költségű utak problémája (negatív élekkel)

A kezdőcsúcsból elérhető negatív összköltségű körön nem léteznének legkisebb költségű utak, mivel az illető körön tetszőlegesen sokszor végig menve az utak költsége mindig tovább csökkenthető lenne.

Irányítatlan gráf esetén, egy (u, v) negatív súlyú irányítatlan élen oda-vissza haladva az út költsége szintén korlátlanul csökkenthető lenne, azaz úgy viselkedne, mint egy negatív összköltségű kör. Tekintsük az irányítás nélküli élte tehát negatív összköltségű, két élből álló irányított körnek. Ez egybevág az ábrázolás szintjén megvalósított irányítatlan gráffal, ahol egy irányítatlan élt, egy oda-vissza irányított élpárral valósítunk meg. Tehát irányítatlan gráf esetén a megszorításunk az, hogy egyáltalán ne tartalmazzon negatív súlyú élt, mert az negatív irányított körnek tekinthető.

Bellman-Ford algoritmus

Minden csúcsra, ha létezik legrövidebb út, akkor létezik egyszerű legrövidebb út is, mivel a körök összköltsége nem negatív, így a kört elhagyva az út költsége nem nőhet. Egy n pontú gráfban, a legnagyobb élszámú egyszerű út hossza legfeljebb n - 1 lehet.

A Bellman-Ford-algoritmus a Dijkstra algoritmusnál megismert közelítés műveletét végzi, azaz egy csúcson át a szomszédba vezető él mentén vizsgálja, hogy az illető él része-e a legrövidebb útnak, javító él-e. Egy menetben az összes élre megvizsgálja, hogy javító él-e vagy sem. Összesen n - 1 menetet végez.

Vizsgáljunk meg egy $p^* = s \rightsquigarrow u$ legrövidebb utat. Minden menetben a p^* minden élén végzünk közelítést. Legyen $v \to w$ él része p^*-nak. Miután p^* v-ig tartó részútja p_v^* ismertté válik, a következő menetben a p_v^* is ismert lesz, mivel az (v, w) éllel is végzünk közelítést. Azonban az élek feldolgozásának (közelítésének) sorrendjére nem tettünk semmilyen megkötést, így csak azt tudjuk garantálni, hogy az első lépés után az 1 élszámú legrövidebb utak, a második lépés után a 2 élszámú legrövidebb utak, és így tovább, válnak ismertté. Mivel a leghosszabb egyszerű út $n - 1$ élszámú, ezért szükséges lehet az $n - 1$ menet.

Egy G élsúlyozott (akár negatív) irányított gráf s startcsúcsából keres minden élhez minimális költségű utakat, illetve felismeri, ha negatív költségű kör van a gráfban. A d és π tömböket az előzőekhez hasonlóan kezeljük.

Az algoritmus:

1. A startcsúcs súlyát állítsuk be 0-ra.
2. $n-1$ iterációban menjünk végig az összes csúcson, és minden csúcsot (u) vessünk össze minden csúccsal (v). Ha olcsóbb utat találtunk akkor v-be felülírjuk a súlyát ($d[v] = d[u] + d(u,v)$), és a szülőjét ($\pi[v] = u$).
3. Ha az n-edik iterációban is történt módosítás, negatív kör van a gráfban

Minimális költségű feszítőfa keresése

A probléma megjelenése egy időszakban, a villamosítás éveiben elég gyakori volt. Ha egy terület villamosítását kell megoldani a lehető legkisebb költséggel, akkor a feladat minimális összköltségű vezetékrendszer tervezése megadott helységek között.

A modellünk legyen irányítás nélküli, súlyozott gráf, ahol a városoknak megfeleltetjük a gráf pontjait, az éleknek pedig a tervezett, két várost összekötő villamos vezetéket. Az élek irányítás nélküliek az elektromos áram irányítatlan tulajdonsága miatt, és súlyozottak, ahol az élek költségei legyenek a becsült építési költségek.

Legyen $G(V,E)$ irányítatlan gráf.

- Részgráf: A $G' = (V', E')$ gráf, G ***részgráfja***, ha $V' \subseteq V$ és $E' \subseteq E$ és $\forall (u,v) \in E' : u, v \in V'$.
- Feszítőfa: $F = (V, E')$ összefüggő, körmentes irányítatlangráfot G ***feszítőfájának*** nevezzük. (F és G csúcshalmaza azonos)
- Minimális feszítőfa: $F = (V, E')$ feszítőfa minimális, ha G feszítőfái között az élek költsége

$$c(H) = \sum_{(u,v)\in E'} c(u,v)$$

minimális, azaz

$$C(F) = \left\{ c(H) \;\middle|\; H \text{ feszítőfája } G\text{-nek} \right\}$$

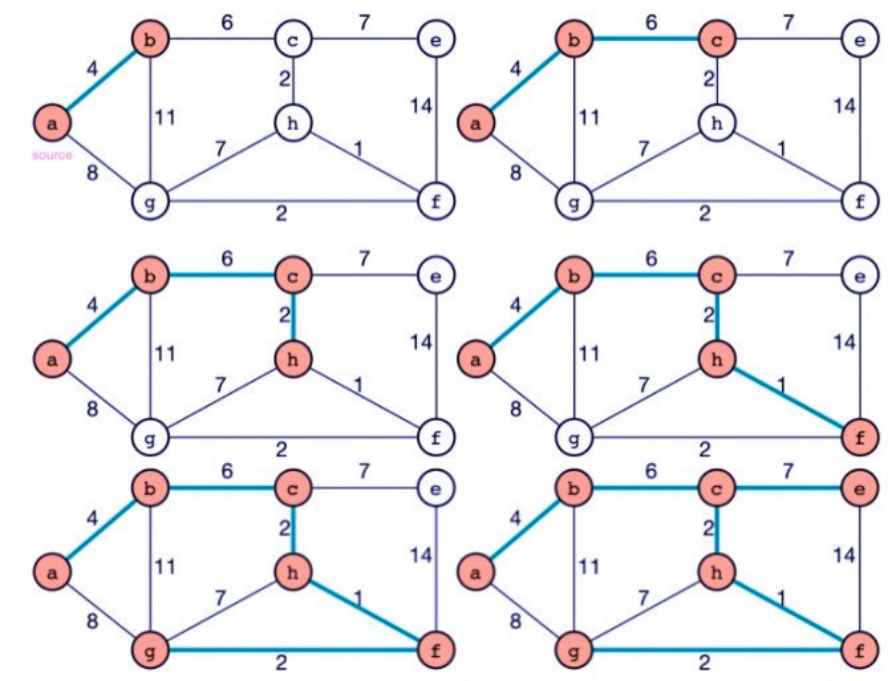
Prim-algoritmus

Egy speciális piros-kék algoritmus, amely minden lépésben a kék szabályt alkalmazza.

Kék szabály: Válasszunk ki egy olyan $\emptyset \neq X \subset V$ csúcshalmazt, amiből nem vezet ki kék él.
Ez után egy legkisebb súlyú X-ből kimenő szintelen élt fessünk kékre.

Az algoritmus:

- Válasszuk ki a gráf egy tetszőleges csúcsát, legyen ez egy egy csúcsú fa. (kezdő X)
- Ameddig van a gráfnak olyan csúcsa, amelyik nincs benne a fában, végezzük el a következőket:
  - Válasszuk ki a fa csúcsai és a gráf többi csúcsa között futó élek közül a legkisebb súlyút. $min\left\{(u,v) \in E \;\middle|\; u \in X,\ v \notin X,\ (u,v) \notin X\right\}$
  - A kiválasztott él nem fabeli csúcsát tegyük át a fába az éllel együtt. (kékre festés)



1. ábra. Prim algoritmus

Mélységi bejárás

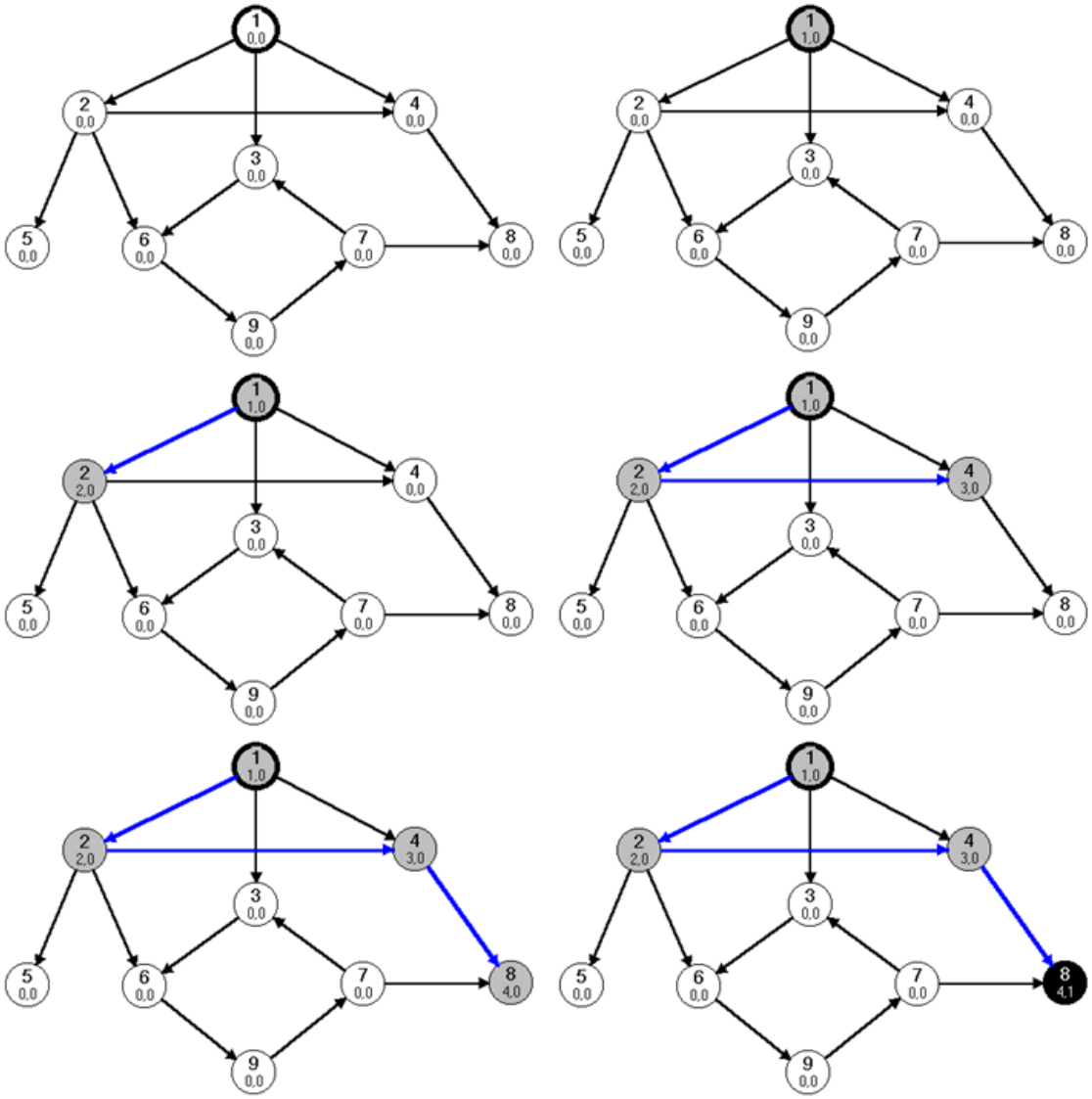
A mélységi bejárás kidolgozása felé lépve, megkülönböztetjük a csúcsok státuszát. Erre több, szemléletében különböző módszerrel találkozhatunk, amelyek végső ugyanazt a célt érik el. Itt a színezés eszközével élünk és három színt használunk. Attól függően színezzük a csúcsokat, hogy az illető csúcsot illetően a bejárás milyen fázisban van.

Egy csúcs legyen fehér, ha még nem jutottunk el hozzá a bejárás során (kezdetben minden csúcs fehér).
Egy csúcs legyen szürke, ha a bejárás során már elértük a csúcsot, de még nem állíthatjuk, hogy az illető csúcsból elérhető összes csúcsot meglátogattuk.
A csúcs legyen fekete, ha azt mondhatjuk, hogy az illető csúcsból elérhető összes csúcsot már meglátogattuk és visszamehetünk (vagy már visszamentünk) az idevezető út megelőző csúcsára

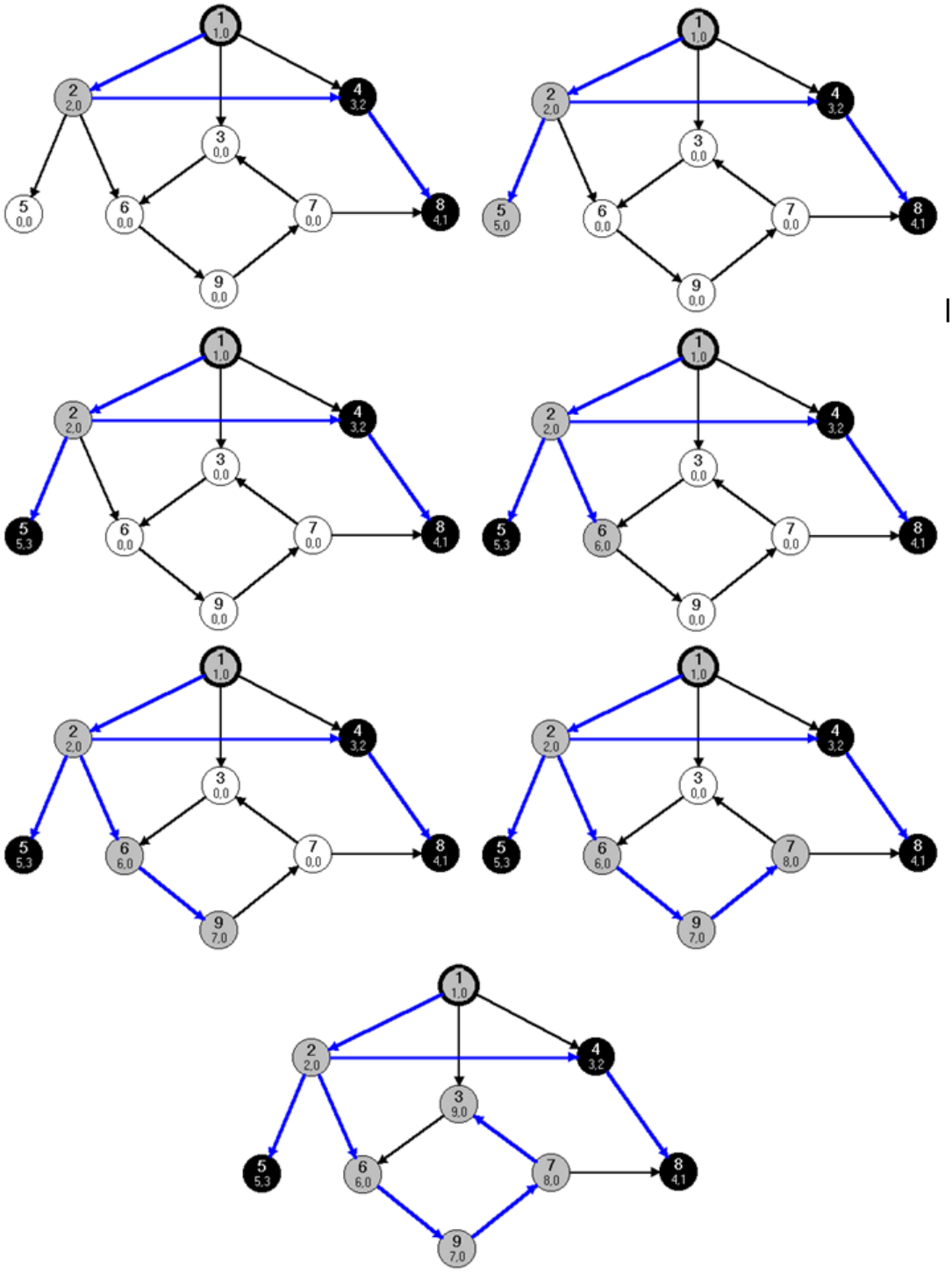
A bejárás során tároljuk el, hogy egy csúcsot hányadikként értünk el, azaz hányadikként lett szürke és tároljuk el azt is, hogy hányadikként fejeztük be a csúcs, és a belőle elérhető csúcsok bejárását, azaz a csúcs hányadikként lett fekete. Az említett számokat nevezzük mélységi, illetve befejezési számnak, amelyeket az ábrákon a csúcsok címkéi alatt jelenítünk meg. (Alternatív szóhasználat: belépési, illetve kilépési számok.) Utalunk arra, hogy ezeknek a számoknak lényeges szerep jut majd az élek osztályozásánál.

A következő ábrákon egy gráf mélységi bejárása látható. A példában szereplő gráfon a csúcsokból kimenő élek feldolgozási sorrendje legyen a rákövetkező csúcsok címkéje szerint növekedően, vagyis alfabetikusan rendezett (például a láncolt ábrázolásnál az éllista a csúcsok címkéje szerint rendezett).

Nézzük az első ábrát, amelyben a kezdőcsúcs legyen az 1-es csúcs. Legyen kezdetben minden csúcs fehér, és a mélységi és befejezési számuk is legyen az extremális 0. A kezdőcsúcsot érjük el elsőként, tehát színezzük szürkére, és a mélységi számát állítsuk be 1-re.



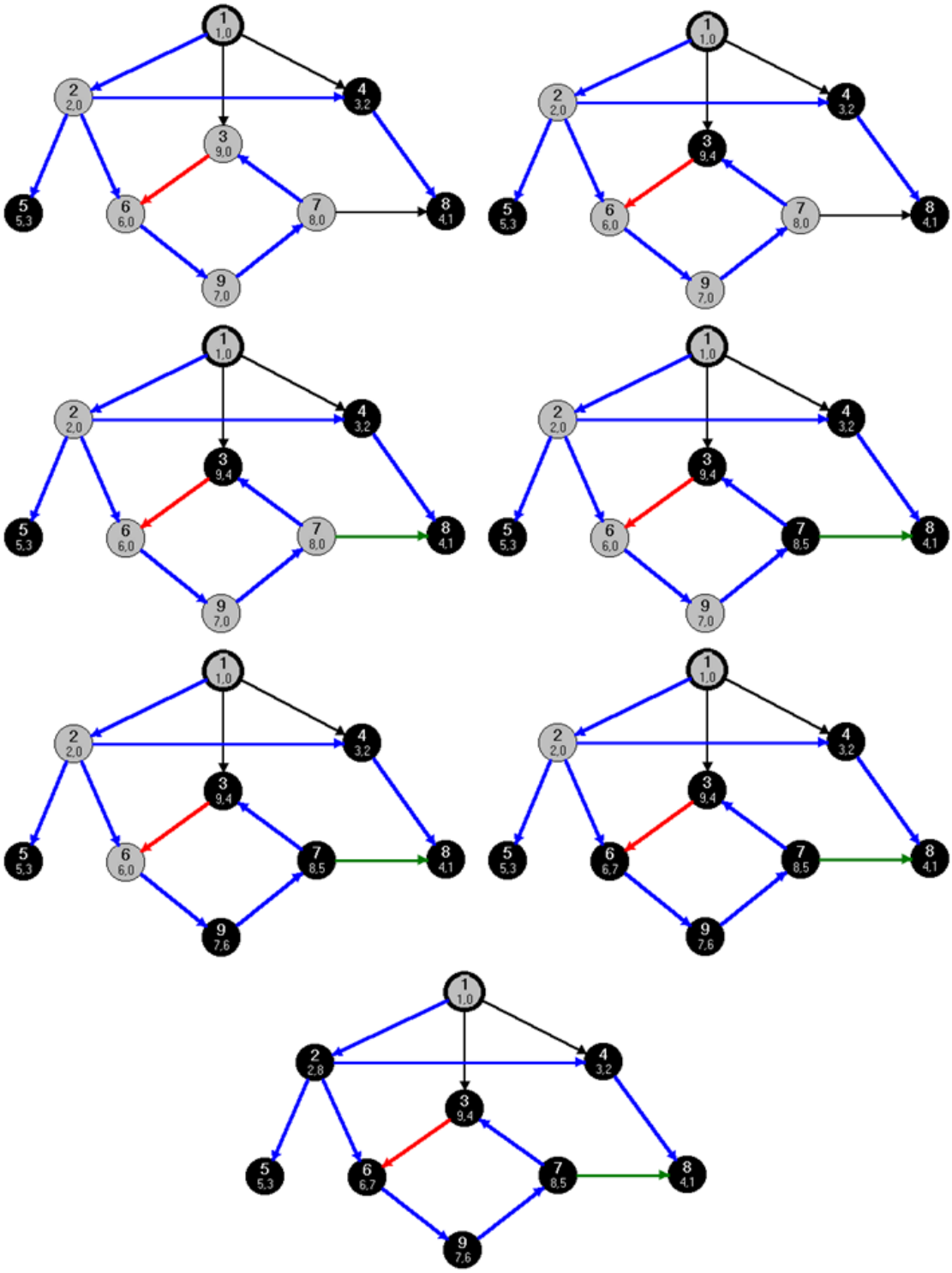
Az 1-es csúcsból három él vezet ki, de a kikötöttük, hogy az élek feldolgozási sorrendje legyen a szomszéd csúcsok címkéje szerint növekedően rendezett. Ekkor a 2-es csúcsot érjük el másodikként. Ezután harmadikként a 4-es csúcs, majd negyedikként a 8-as csúcs következik. Mivel a 8-as csúcsnak egyáltalán nincs szomszédja, bejárását befejeztük és a csúcsot feketére színezzük. Mivel a bejárás során a 8-as csúcs lett elsőként fekete, a befejezési száma 1-es lesz.



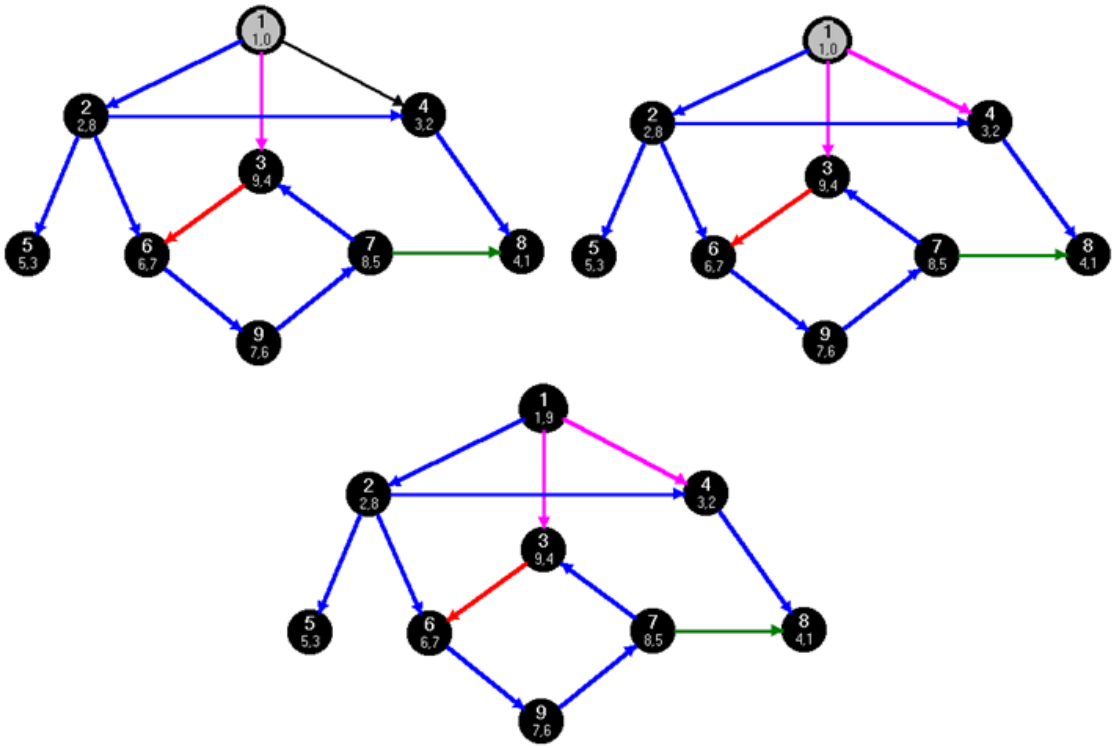
A bejárás során eddig megtett utunk $< 1, 2, 4, 8 >$. Most menjünk vissza az utolsó előtti 4-es csúcsra. Mivel a 4-es csúcsnak sincs meg nem látogatott szomszédja, így ennek csúcs bejárását is befejeztük; színezzük a csúcsot feketére, és a bejárási számát állítsuk 2-re.

Menjünk vissza a 2-es csúcshoz. A 2-es csúcsnak két olyan szomszédja is van, amelyet még nem látogattunk meg. Lépjünk a kisebb címkéjű csúcsba.

Az 5-ös csúcs bejárását harmadikként fejezzük be. A 2-es csúcsból a bejárást a 6-os csúcs irányába folytatjuk. Tovább haladva, hetedikként elérjük a 9-es csúcsot. Nyolcadikként következik a 7-es csúcs. Mivel a 3-as csúcs még fehér, azt érjük el kilencedikként



A 3-as csúcsból a 6-os csúcsba vezet él, azonban a 6-os csúcsot már bejártuk, a színe már nem fehér, erre már nem folytatjuk a bejárást.
Mivel a 3-as csúcsból már nem vezet él fehér csúcsba, így a 3-as csúcs bejárását is befejeztük.
Visszamegyünk a 7-es csúcsba, ahol a sorrendben következő él, a $(7, 8)$ mentén látjuk, hogy a 8-as csúcs színe már fekete. Mivel a 7-es csúcsnak nincs fehér szomszédja, így ötödikként befejeztük a bejárását.
A 9-es csúcsnak a bejárását is befejeztük. Az úton ismét egy csúccsal visszamegyünk és befejezzük a 6-os csúcsot.
A 2-es csúcsra lépve, látható, hogy minden kimenő éle mentén már próbálkoztunk, így nyolcadikként azt is elhagyjuk.



A mélységi bejárás algoritmusa

Az előző példában úgy kezdtük a bejárást, hogy kijelöltünk egy kezdőcsúcsot, amelyből kiindulva történetesen az összes csúcs elérhető volt mélységi bejárással. Egy másik példában előfordulhatna, hogy lennének olyan csúcsok, amelyeket egyáltalán nem tudnánk elérni egy kiválasztott startcsúcsból. A későbbi alkalmazások érdekében a mélységi bejárást úgy definiáljuk, hogy az a gráf minden pontjához eljusson.

Az algoritmus működésének nem feltétele egy kezdőcsúcs megadása, azt az eljárás keretében magunk tetszőlegesen választjuk meg, kívülről nézve véletlen jelleggel. Miután minden, ebből elérhető csúcsot bejártunk, visszajutottunk az említett kezdőpontba. Ha maradt olyan csúcs, amelyet a bejárás nem ért el, azaz színe fehér maradt, akkor választunk közülük egy következőt és abból kiindulva újra elvégezzük a mélységi bejárást.

Ezt az eljárást addig folytatjuk, amíg van fehér csúcsunk. Nyilván minden ilyen menetben legalább egy csúcsot átszínezünk feketére, tehát véges számú menet után elfogynak a fehér csúcsok. Elegendő a csúcsok halmazán egyszer végigmenni (a gyakorlatban a csúcsok címkéje szerinti növekedően), és ha egy csúcs színe fehér, akkor onnan indítsunk egy bejárást.

A gráf éleit a mélységi bejárás közben osztályozhatjuk. (Inicializáláskor minden értéket 0-ra állítottunk)

- Faél: A következő csúcs mélységi száma = 0
- Visszaél: A következő csúcs mélységi száma > 0, és befejezési száma $= 0$
  (Tehát az aktuális út egy előző csúcsára kanyarodunk vissza)
- Keresztél: A következő csúcs mélységi száma > 0, és befejezési száma > 0, továbbá az aktuális csúcs mélységi száma $>$ következő csúcs mélységi száma.
  (Ekkor egy az aktuális csúcsot megelőző csúcsból induló, már megtalált útba mutató éllel van dolgunk)
- Előreél: A következő csúcs mélységi száma > 0, és befejezési száma > 0, továbbá az aktuális csúcs mélységi száma $<$ következő csúcs mélységi száma.
  (Ekkor egy az aktuális csúcsból induló, már megtalált útba mutató éllel van dolgunk)

DAG Topologikus rendezése

Alapfogalmak

- Topologikus rendezés Egy $G(V, E)$ gráf topologikus rendezése a csúcsok olyan sorrendje, melyben $\forall(u \to v) \in E$ élre u előbb van a sorrendben, mint v
- DAG - Directed Acyclic Graph Irányított körmentes gráf.
  Legtöbbször munkafolyamatok irányítására illetve függőségek analizálására használják.

  Tulajdonságok:

  - Ha G gráfra a mélységi bejárás visszaélt talál (Azaz kört talált) $\implies$ G nem DAG
  - Ha G nem DAG (van benne kör) $\implies$ Bármely mélységi bejárás talál visszaélt
  - Ha G-nek van topologikus rendezések $\implies$ G DAG
  - Minden DAG topologikusan rendezhető.

DAG topologikus rendezése
Egy G gráf mélységi bejárása során tegyük verembe azokat a csúcsokat, melyekből visszaléptünk. Az algoritmus után a verem tartalmát kiírva megkapjuk a gráf egy topologikus rendezését.

Adattömörítések

Huffman-algoritmus

A Huffman-kódolás karakterek (jelek, betűk, számjegyek) olyan kódolását jelenti, ahol az egyes karakterekhez rendelt kódok nem azonos hosszúságúak (különböző számú bitből állnak), így a belőlük alkotott szöveg hosszában rövidíthetővé válik. Ez a karakterek gyakoriságának figyelembevételével történik. Maga a kódolás egy mohó stratégián alapszik, és az adattömörítésben igen hatékonyan használható.

A kódolás során egy speciális adatszerkezetet (lényegében egy bináris fát, amely most egyszerűen csak listaként ábrázolunk) építünk fel lépésről-lépésre, a következő módon:

1. Kiválasztjuk a lista két legkisebb gyakoriságú elemét, amely egy háromcsúcsú bináris fa két levele (olyan csúcs, amelynek nincs gyereke) lesz (amelyeket a gyakorisággal címkézzük meg), majd ezekhez hozzárendelünk egy gyökeret, amelyet a két gyakoriság összegével címkézünk meg.
2. Ezután a két vizsgált elemet kitöröljük a listából, és azok összegét beszúrjuk az érték szerinti megfelelő helyre, hogy a lista rendezettsége megmaradjon.
3. Ezután folytatjuk a műveletet az 1. lépésnél mindaddig, amíg van elem a listában.

Az így felépített adatszerkezetben a levelek az eredeti karaktereknek (illetve azok gyakoriságának) felelnek meg.

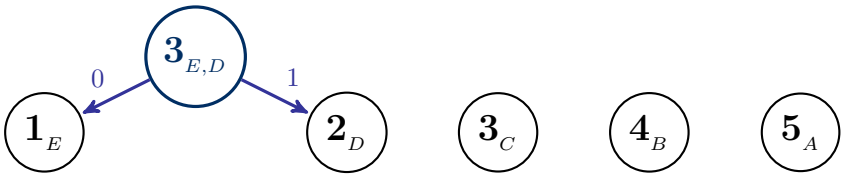
Az eredményül kapott fában, minden csúcs esetében címkézzük meg 0-val a belőle kiinduló bal oldali élt, 1-gyel pedig a jobb oldalit.

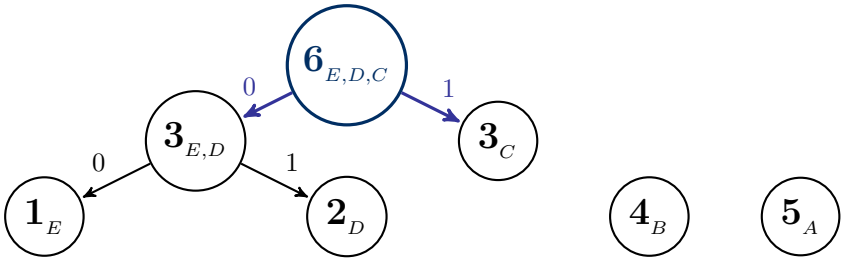
A gyökértől egy adott levélig egyetlen út halad. Ezen út éleihez rendelt 0 és 1 címkéket sorrendben összeolvasva, megkapjuk a levélhez rendelt karakter kódját. Látható, hogy a gyakoribb karakterek kódja rövidebb, míg a kevésbé gyakoribbaké hosszabb lesz.

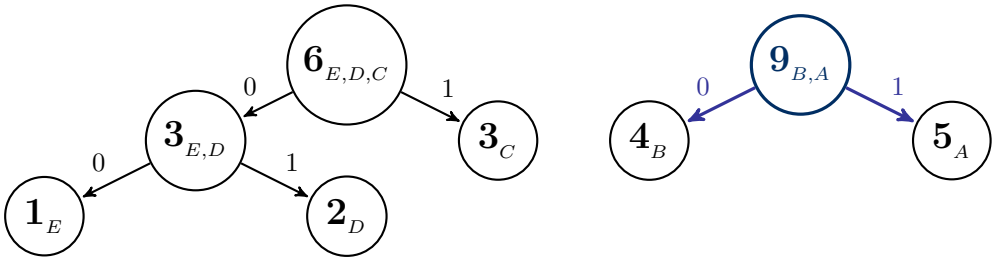
Megjegyzés: Attól függően, hogy a bináris fa felépítésében egy adott lépésben melyik elem kerül balra és melyik jobbra, különböző eredményt kaphatunk, de ez nem befolyásolja a kapott kód hatékonyságát, illetve a megoldás helyességét.

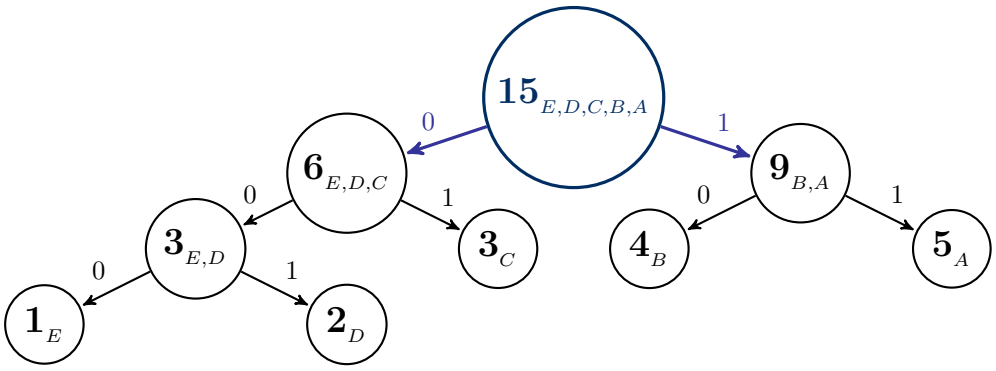
Példa: Legyen a következő öt betű, mely a megadott gyakorisággal fordul elő (már növekvő sorrendben):

Ekkor a redukciós lépések a következők:









Tehát a kódszavak:

A	B	C	D	E
11	10	01	001	000

LZW-algoritmus

Az LZW (Lempel-Ziv-Welch) egy veszteségmentes tömörítési algoritmus. A tömörítésnek a lényege, hogy egy szótárat bővítünk folyamatosan, és az egyes kódolandó szavakhoz szótárindexeket rendelünk.

Kódolás

A kódolás algoritmusa a következő lépésekből áll:

1. A szótárt inicializáljuk az összes 1 hosszú szóval
2. Kikeressük a szótárból a leghosszabb, jelenlegi inputtal összeillő W sztringet
3. W szótárindexét kiadjuk, és W-t eltávolítjuk az inputról
4. A W szó és az input következő szimbólumának konkatenációját felvesszük a szótárba
5. A 2. lépéstől ismételjük

Példa:

Bemenet: wabbawabba

Szótár:

a	b	w	wa	ab	bb	ba	aw	wab	bba
1	2	3	4	5	6	7	8	9	10

Index	**E**lőző $\begin{cases} \textbf{B[I-1]}, \text{ ha } E+J \in S \wedge I \geq 2 \\ \textbf{E}+\textbf{B[I-1]}, \text{ ha } K[I-1] = \varnothing \wedge I \geq 2 \\ \varnothing, \text{ különben} \end{cases}$	**J**elenlegi B[I]	**K**ód $\begin{cases} \textbf{E}+\textbf{J}, \text{ ha } E+J \notin S \\ \varnothing, \text{ ha } E+J \in S \end{cases}$	**K**imenet $\begin{cases} \textbf{E}, \text{ ha K} \neq \varnothing \vee J = \varnothing \\ \varnothing, \text{ ha K} = \varnothing \end{cases}$
1		w		
2	w	a	wa - 4	**3**
3	a	b	ab - 5	**1**
4	b	b	bb - 6	**2**
5	b	a	ba - 7	**2**
6	a	w	aw - 8	**1**
7	w	a		
8	wa	b	wab - 9	**4**
9	b	b		
10	bb	a	bba - 10	**6**
(11)$\varnothing$	a			**1**

Kimenet: 3 1 2 2 1 4 6 1

Dekódolás

A dekódolás során is építenünk kell a szótárat. Ezt már azonban csak a dekódolt szöveg(rész) segítségével tudjuk megtenni, mivel egy megkapott kód dekódolt szava és az utána lévő szó első karakteréből áll össze a szótár következő eleme.

Tehát a dekódolás lépései:

1. Kikeressük a kapott kódhoz tartozó szót a szótárból (u), az output-ra rakjuk
2. Kikeressük a következő szót (v) a szótárból, az első szimbólumát u-hoz konkatenálva a szótárba rakjuk a következő indexszel.
3. Amennyiben már nincs következő szó, dekódolunk, de nem írunk a szótárba.

Megtörténhet az az eset, hogy mégis kapunk olyan kódszót, mely még nincs benne a szótárban. Ez akkor fordulhat elő, ha a kódolásnál az aktuálisan szótárba írt szó következik.

Példa:

Szöveg: AAA
Szótár: A - 1

Ekkor a kódolásnál vesszük az első karaktert, a szótárbeli indexe 1, ezt kiküldjük az outputra. A következő karakter A, így AA-t beírjuk a szótárba 2-es indexszel. Az első karaktert töröljük az inputról. Addig olvasunk, míg szótárbeli egyezést találunk, így AA-t olvassuk (amit pont az előbb raktunk be), ennek indexe 2, tehát ezt küldjük az outputra. AA-t töröljük az inputról, és ezzel végeztünk is. Az output: 1,2

Dekódoljuk az 1,2 inputot! Jelenleg a szótárban csak A van 1-es indexszel. Vegyük az input első karakterét, az 1-et, ennek szótárbeli megfelelője A. Ezt tegyük az outputra. A következő index a 2, de ilyen bejegyzés még nem szerepel a szótárban.

Ebben az esetben a dekódolásnál, egy trükköt vetünk be. A szótárba írás pillanatában még nem ismert a beírandó szó utolsó karaktere (A példában A-t találtuk, de nem volt 2-es bejegyzés). Ekkor ?-et írunk a szótárba írandó szó utolsó karakterének helyére. (Tehát A? - 2 kerül a szótárba). De mostmár tudni lehet az új bejegyzés első betűjét (A? - 2 az új bejegyzés, ennek első betűje A). Cseréljük le a ?-et erre a betűre. (Tehát AA - 2 lesz a szótárban).

Példa:

Bemenet: 3 1 2 2 1 4 6 1

Szótár:

a	b	w
1	2	3

Input	**E**lőző	**K**imenet	**K**ód (Sejtés)	**K**ód (Teljes)
3		w	w? - 4	
1	w	a	a? - 5	wa - 4
2	a	b	b? - 6	ab - 5
2	b	b	b? - 7	bb - 6
1	b	a	a? - 8	ba - 7
4	a	wa	wa? - 9	aw - 8
6	w	bb	bb? - 10	wab - 9
1	bb	a		bba - 10
∅				

Kimenet: wabbawabba

Mintaillesztés

A mintaillesztés feladata az, hogy egy szövegben egy szövegminta (szövegrészlet, string) előfordulását vagy előfordulásait megkeressük. A mintaillesztés elnevezés mellett találkozunk a stringkeresés elnevezéssel is.

A feladat általánosítható: valamely alaptípus feletti sorozatban keressük egy másik (általában jóval rövidebb) sorozat előfordulásait (például egy DNS láncban keresünk egy szakaszt).

A továbbiakban egyszerűsítjük a feladatot a minta első előfordulásának a megkeresésére, amelynek segítségével az összes előfordulás megkapható. (Keressük meg a minta első előfordulását, majd a hátralévő szövegben ismét keressük az első előfordulást stb.)

Vezessük be az alábbi jelöléseket:

- Legyen H egy tetszőleges alaptípus feletti véges halmaz, a szöveg *ábécéje*.
- Legyen a *szöveg*, amelyben a mintát keressük: $S[1 \dots n] \in H^*$, azaz egy n hosszú H feletti véges sorozat.
- Legyen a *minta*, amelyet keresünk a szövegben: $S[1 \dots m] \in H^*$, egy m hosszú szintén a H feletti véges sorozat.

Továbbá, tegyük fel, hogy S-en és M-en megengedett művelet az indexelés, azaz hivatkozhatunk a szöveg vagy a minta egy i-edik elemére $S[i]$ ($i \in [1 \dots n]$), $M[i]$ ($i \in [1 \dots m]$).

A tárgyalt algoritmusok némelyike lényeges módosítás nélkül átírható szekvenciális fájlokra is (ahol az indexelés nem megengedett), míg a más tárgyalt algoritmusok csak puffer használatával alkalmazhatók a csak szekvenciálisan olvasható hosszabb szövegekre.

Az illeszkedés fogalma

Azt mondjuk, hogy

- az M minta a $k+1$-dik pozíción illeszkedik az S szövegre (előfordul a szövegben), vagy
- az M minta k eltolással illeszkedik S-re, illetve
- k érvényes eltolás,

ha

$$S[k+1 \cdots k+m] = M[1 \dots m], \text{ azaz } \quad \forall j \in [1 \dots m] : S[k+j] = M[j].$$

Továbbá, az M mintának a $(k+1)$-edik pozíción való illeszkedése az M első előfordulása az S szövegben, ha

$$\forall i \in N,\ i \in [0 \dots k-1] : S[i+1 \dots i+m] \neq M[1 \dots m].$$

Legyen például a szövegünk S="ABBABCAB", és a keresett minta pedig M="ABC". A fenti definíció szerint az M minta a 4-edik pozíción, k=3 eltolással illeszkedik az S szövegre (ABB**ABC**AB).

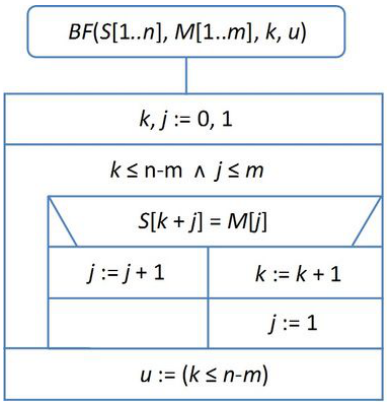
Az egyszerű mintaillesztés algoritmusa

A stringkeresési feladat naiv megoldást nevezzük egyszerű mintaillesztésnek. Ehhez egy a nyers erő (brute force) levén működő algoritmushoz könnyen eljuthatunk a már tanult programozási tételekre való visszavezetéssel.

Tekintsük megengedett műveletnek az $S[k+1 \dots k+m] = M[1 \dots m]$ vizsgálatot. Ennek a kifejezésnek az eredményét megkaphatjuk karakterenkénti összehasonlítással is, amelynek során a minta minden karakterét összehasonlítjuk a szövegdarab megfelelő karakterével; és ha az összes vizsgált karakter egyezik, akkor a kifejezés értéke legyen igaz, különben hamis.

Az $S[k+1 \dots k+m] = M[1 \dots m]$ vizsgálat előbb említett megvalósítása javítható, ha visszavezetjük lineáris keresésre, amelynek során keressük az első olyan $j \in N,\ j \in [1 \dots m]$ pozíciót, amelyre $S[k+j] \neq M[j]$.

Amennyiben nem találunk ilyen j pozíciót, azaz $\forall j \in N,\ j \in [1 \dots m] : S[k+j] = M[j]$, akkor az M illeszkedik S-re k eltolással, tehát $S[k+1 \dots k+m] = M[1 \dots m]$ vizsgálat eredménye legyen igaz, különben pedig hamis. Ezt a megoldást nevezzük az egyszerű mintaillesztés algoritmusának, amely nem más, mint egy lineáris keresésbe ágyazott lineáris keresés.



2. ábra. Az egyszerű mintaillesztés algoritmusa

Műveletigény

A legjobb esetben a minta első karaktere egyáltalán nem szerepel a szövegben, így minden k eltolásnál már $j = 1$ esetben mindig elromlik az illeszkedés. Tehát minden eltolásnál csak egy összehasonlítás történik, így az összehasonlítások száma megegyezik az eltolások számával, $(n-m+1)$-gyel. Azaz $\text{MÖ}(n,m) = n-m+1 = \Theta(n)$

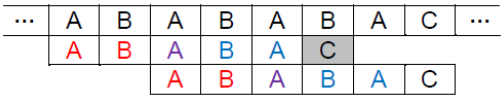
A legkedvezőtlenebb esethez akkor jutunk, ha minden eltolásánál csak a minta utolsó karakterénél romlik el az illeszkedés. Ekkor minden eltolásnál m összehasonlítást végzünk, így a műveletigény az eltolások számának m-szeresével jellemezhető. Azaz $\text{MÖ}(n,m) = (n-m+1) * m = \Theta(n * m)$

Szekvenciális sorozatokra, fájlokra való alkalmazhatóság

A gyakorlatban az általunk szövegnek nevezett sorozat nem egyszer igen nagyméretű is lehet, emiatt csak olyan szekvenciális formában áll rendelkezésünkre, amelyen az indexelés nem megengedett művelet. Hasznos lehet annak vizsgálata, hogy az ismertetett algoritmust mennyire egyszerű átírni szekvenciális sorozatokra, illetve fájlokra. Az egyszerű mintaillesztő algoritmus szekvenciális sorozatokra történő átírásánál kénytelenek vagyunk puffert használni, mivel a szövegben időnként vissza kell "ugrani" (akkor, ha az illeszkedés nem a minta első karakterénél romlik el).

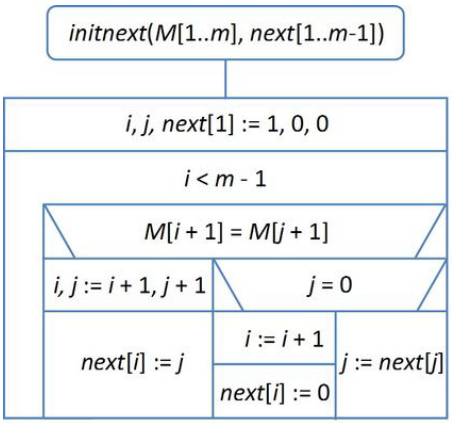
Knuth-Morris-Pratt algoritmus

A nyers erőt használó egyszerű mintaillesztés műveletigénye legrosszabb esetben $n * m$-es volt. A Knuth-Morris-Pratt algoritmus (KMP-vel rövidítjük) egyike azon mintaillesztő eljárásoknak, amelyek ügyes észrevételek és mélyebb megfontolások alapján hatékonyabb módon oldják meg az stringkeresés feladatát.



3. ábra. KMP algoritmus több karakter tolás estén

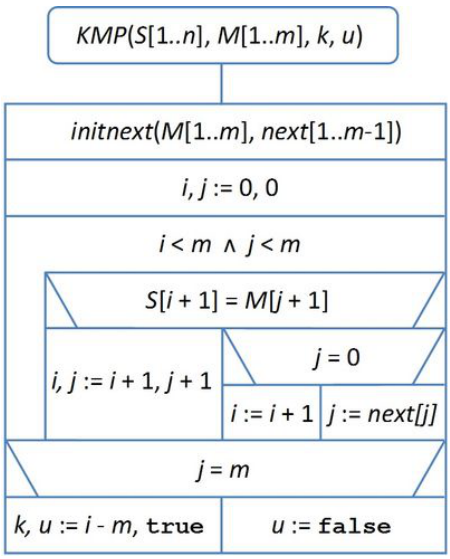
Az ugrás megállapítását a következőképp tesszük: Az eddig megvizsgált egyező mintarész elején (prefix) és végén (suffix) olyan kartersorozatot keresünk, melyek megegyeznek. Ha találunk ilyet, akkor a mintát annyival tolhatjuk, hogy az elején lévő része ráilleszkedjen a végén levőre.
Azt, hogy ez egyes esetekben mekkorát tolhatunk nem kell minden elromlás alkalmával vizsgálni. Ha a mintára önmagával lefuttatjuk az algoritmus egy módosított változatát (4. ábra), kitölthetünk egy tömböt, mely alapján a tolásokat végezni fogjuk.



4. ábra. KMP tolásokat szabályzó tömb kitöltése

Az algoritmus (ld 5. ábra):

- Két indexet i és j futtatunk a szövegen illetve a mintán.
- Ha az $i + 1$-edik és $j + 1$-edik karakterek megegyeznek, akkor léptetjük mind a kettőt.
- Ha nem egyeznek meg, akkor:
  - Ha a minta első elemét vizsgáltuk, akkor egyet tolunk a mintán, magyarul a minta indexe marad az első betűn, és a szövegben lévő indexet növeljük eggyel ($i = i + 1$)
  - Ha nem a minta első elemét vizsgáltuk, akkor annyit tolunk, amennyit szabad. Ez azt jelenti, hogy csak a mintán lévő indexet helyezzük egy kisebb helyre ($j = next[j]$)
- Addig megyünk, míg vagy a minta, vagy a szöveg végére nem érünk. Ha a minta végére értünk, akkor megtaláltuk a mintát a szövegben, ha a szöveg végére értünk, akkor pedig nem.

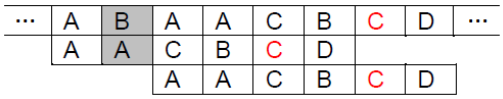


5. ábra. KMP algoritmus

Boyer-Moore — Quick search algoritmus

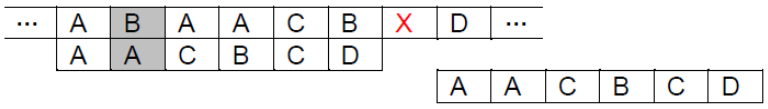
Míg a KMP algoritmus az elromlás helye előtti rész alapján döntött a tolásról, addig a QS a minta utáni karakter alapján. Tehát elromlás esetén:

- Ha a minta utáni karakter benne van a mintában, akkor jobbról az első előfordulására illesztjük. (6. ábra)



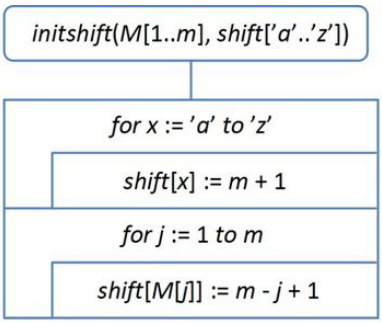
6. ábra. QS - eltolás ha a minta utáni karakter benne van a mintában

- Ha a minta utáni karakter nincs benne a mintában, akkor a mintát ezen karakter után illesztjük. (7. ábra)



7. ábra. QS - eltolás ha a minta utáni karakter nincs benne a mintában

Az eltolás kiszámítását megint elő lehet segíteni egy tömbbel, most azonban, mivel nem a minta az érdekes, és nem tudjuk pontosan mely karakterek szerepelnek a szövegben, így a tömbbe az egész abc-t fel kell vennünk (8. ábra)

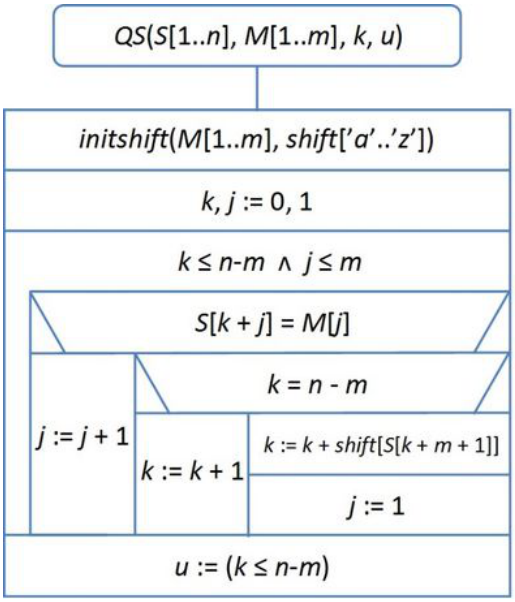


8. ábra. QS - Az eltolást elősegítő tömb ($Shift['a'...'z']$) konstruálása

Az algoritmus (ld. 9. ábra):

- Két indexet k és j futtatunk a szövegen illetve a mintán.

- Ha a szöveg $k + j$-edik eleme megegyezik a minta j-edik karakterével, akkor léptetjük j-t (mivel a szövegben $k + j$-edik elemet nézzük, így elég j-t növelni).
- Ha nem egyeznek meg, akkor:
  - Ha a minta már a szöveg végén van ($k = n - m$), akkor csak növeljük k-t eggyel, ami hamissá teszi a ciklus feltételt.
  - Ha még nem vagyunk a szöveg végén k-t toljuk annyival, amennyivel lehet (ezt az előre beállított $Shift$ tömb határozza meg). És a j-t visszaállítjuk 1-re.
- Addig megyünk, míg vagy a minta végére érünk j-vel, vagy a mintát továbbtoltuk a szöveg végénél. Előbbi esetben egyezést találtunk, míg az utóbbiban nem.



9. ábra. QS

Rabin-Karp algoritmus

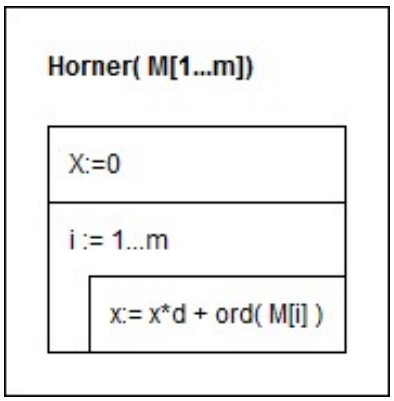
A Rabin-Karp algoritmus lényege, hogy minden betűhöz az ábécéből egy számjegyet rendelünk, és a keresést számok összehasonlításával végezzük. Világos, hogy ehhez egy ábécé méretnek megfelelő számrendszerre lesz szükségünk. A szövegből mindig a minta hosszával egyező részeket szelünk ki, és ezeket hasonlítjuk össze.

Példa:
Minta: BBAC → 1102
Szöveg: DACABBAC → 30201102, amiből a következő számokat állítjuk elő: 3020, 0201, 2011, 0110, 1102

A fent látható szeletek lesznek az s_i-k.

Az algoritmus működéséhez azonban számos apró ötletet alkalmazunk:

1. A minta számokká alakítását Horner-módszer segítségével végezzük.



10. ábra. RK - Horner-módszer

Az $ord()$ függvény az egyes betűknek megfelelő számot adja vissza. A d a számrendszer alapszáma.

2. A szöveg mintával megegyező hosszú szeleteinek (s_i) előállítása:
   s_0-t a Horner-módszerrel ki tudjuk számolni. Ezek után s_{i+1} a következőképp számolandó:

$$s_{i+1} = (s_i - ord(S[i]) \cdot d^{m-1}) \cdot d + ord(S[i+1])$$

   Magyarázat: s_i elejéről levágjuk az első számjegyet ($s_i - ord(S[i]) \cdot d^{m-1}$), majd a maradékot eltoljuk egy helyiértékkel (szorzás d-vel), végül az utolsó helyiértékre beírjuk a következő betűnek megfelelő számjegyet ($+ord(S[i+1])$)

   Példa:

   Az előző példa szövegével és mintájával ($d = 10$ elemű ábécé és $m = 4$ hosszú minta):
   $s_0 = 3020$, ekkor: $s_{0+1} = s_1 = (3020 - ord(D) \cdot 10^3) \cdot 10 + ord(B) = (3020 - 3000) \cdot 10 + 1 = 0201$

3. Felmerülhet a kérdés, hogy az ilyen magas alapszámú számrendszerek nem okoznak-e gondot az ábrázolásnál? A kérdés jogos. Vegyük a következő életszerű példát:

   4 bájton ábrázoljuk a számainkat (2^{32}). Az abc legyen 32 elemű ($d = 32$), a minta 8 hosszú ($m = 8$). Ekkor a d^{m-1} kiszámítása: $32^7 = (2^5)^7 = 2^{35}$, ami már nem ábrázolható 4 bájton.

   Ennek kiküszöbölésére vezessünk be egy nagy p prímet, melyre $d \cdot p$ még ábrázolható. És a műveleteket számoljuk mod p. Ekkor természetesen a kongruencia miatt lesz olyan eset, amikor az algoritmus egyezést mutat, mikor valójában nincs. Ez nem okoz gondot, mivel ilyen esetben karakterenkénti egyezést vizsgálva ezt a problémát kezelni tudjuk. (Fordított eset nem fordul elő tehát nem lesz olyan eset, mikor karakterenkénti egyezés van, de numerikus nincs). [Ha p kellően nagy, a jelenség nagyon ritkán fordul elő.]

4. A mod p számítás egy másik problémát is felvet. Ugyanis a kivonás alkalmával negatív számokat is kaphatunk. Például: Legyen $p = 7$, ekkor, ha $ord(S[i]) = 9$, akkor előző számítás után $s_i = 2...$, de ebből $ord(S[i]) \cdot d^{m-1} = 9 \cdot 10^3 = 9000$-et vonunk ki negatív számot kapunk.

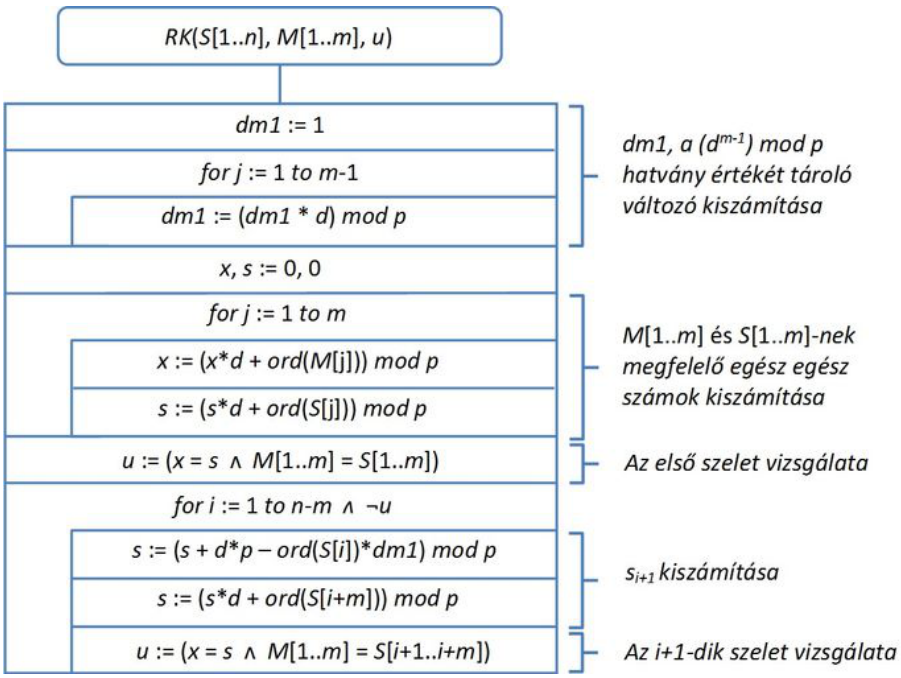
   Megoldásként s_{i+1}-et két lépésben számoljuk:

$$s := (s_i + d \cdot p - ord(S[i]) \cdot d^{m-1}) \mod p$$

$$s_{i+1} := (s \cdot d + ord(S[i+1])) \mod p$$

A fentiek alapján az algoritmus a következő (ld. 11. ábra)

1. Kiszámoljuk d^{m-1}-et ($dm1$)
2. Egy iterációban meghatározzuk Horner-módszerrel a minta számait (x) és s_0-t
3. Ellenőrizzük, hogy egyeznek-e
4. Addig számolgatjuk s_i értékét míg a minta nem egyezik s_i-vel, vagy a minta a szöveg végére nem ért.


RK(S[1..n], M[1..m], u)
dm1 := 1
for j := 1 to m-1
dm1 := (dm1 * d) mod p
dm1, a (d^(m-1)) mod p hatvány értékét tároló változó kiszámítása
x, s := 0, 0
for j := 1 to m
x := (x*d + ord(M[j])) mod p
s := (s*d + ord(S[j])) mod p
M[1..m] és S[1..m]-nek megfelelő egész egész számok kiszámítása
u := (x = s ∧ M[1..m] = S[1..m])
Az első szelet vizsgálata
for i := 1 to n-m ∧ ¬u
s := (s + d*p – ord(S[i])*dm1) mod p
s := (s*d + ord(S[i+m])) mod p
s_(i+1) kiszámítása
u := (x = s ∧ M[1..m] = S[i+1..i+m])
Az i+1-dik szelet vizsgálata

11. ábra. RK