

Gráfalgoritmusok	3
1. Néhány probléma modellezése gráfokkal	3
1.1 Útkeresés	3
1.2 Minimális költségű út keresése	4
1.3 Minimális költségű feszítőfa keresése	4
1.4 Irányított körmentes gráf (DAG) topologikus rendezése	5
1.5 Erősen összefüggő komponensek meghatározása	6
2. Alapfogalmak, jelölések	7
3. Gráfok ábrázolása	9
3.1 Szomszédsági-mátrix (adjacencia-mátrix , csúcsmátrix)	9
3.2 Szomszédsági lista (éllista)	10
3.3 Ellenőrző kérdések	11
3.4 Gyakorló feladatok	11
3.5 Összefoglalás	13
4. Bejárési stratégiák, szélességi bejárás	15
4.1 Bejárési/keresési stratégiák	15
4.2 Szélességi bejárás/keresés algoritmus	19
4.3 Ellenőrző kérdések	26
4.4 Gyakorló feladatok	26
4.5 Összefoglalás	27
5. Legrövidebb utak egy forrásból	29
5.1 Dijkstra algoritmus	30
5.2 Bellman-Ford algoritmus	36
5.3 Ellenőrző kérdések	41
5.4 Gyakorló feladatok	42
5.5 Összefoglalás	44
6. Legrövidebb utak minden csúcspárra	47
6.1 Floyd algoritmus	47
6.2 Transzitiv lezárt	51
6.3 Warshall algoritmus	52
6.4 Ellenőrző kérdések	53
6.5 Gyakorló feladatok	53
6.6 Összefoglalás	54
7. Minimális költségű feszítőfák	55
7.1 A piros-kék eljárás	57
7.2 Prim algoritmus	62
7.3 Kruskal algoritmus	68
7.4 Ellenőrző kérdések	74
7.5 Gyakorló feladatok	74
7.6 Összefoglalás	75
8. Mélységi bejárás és alkalmazásai	77
8.1 Mélységi bejárás	77
8.2 DAG tulajdonság	93
8.3 DAG topologikus rendezése	96
8.4 Erősen összefüggő komponensek meghatározása	102
8.5 Ellenőrző kérdések	110
8.6 Gyakorló feladatok	111
8.7 Összefoglalás	113
9. Implementáció	115
9.1 Gráfok ábrázolása	115
9.2 Szélességi bejárás	116
9.3 Dijkstra algoritmus	118
9.4 Bellman-Ford algoritmus	120
9.5 Floyd algoritmus	125
9.6 Warshall algoritmus	125
9.7 Prim algoritmus	126
9.8 Kruskal algoritmus	127

9.9	Mélységi bejárás	128
9.10	DAG tulajdonság ellenőrzése	129
9.11	Topologikus sorrend	130
9.12	Erősen összefüggő komponensek meghatározása	131
Mellékletek		133
	Prioritásos sor	133
	Unió-Holvan	139
Irodalomjegyzék		141

Gráfalgoritmusok

A megoldandó feladatok, problémák modellezése során sokszor találkozunk bizonyos "dolgok" (az absztrakt objektumok) közötti kapcsolatokat leíró **bináris relációkkal**. Ezen relációk (kapcsolatok) szemléletes leírásának egyik eszköze a **gráf**. A gráfokkal az ember számára könnyen "emészthető" formában lehet ábrázolni a relációk tulajdonságait (pl.: szimmetria = irányítatlanság vagy kettő hosszú kör, reflexivitás = hurokél stb.). A modell objektumainak megfeleltetjük a **gráf csúcsait** az objektumok közötti kapcsolatokat leírására, pedig a **gráf éleket** használjuk. Ezen szemléletes és könnyen kezelhető tulajdonsága miatt lett a gráf elterjedt modellező eszköz. Mivel egy ilyen általános fogalom, mint a bináris reláció modellezésére használjuk, nagyon sok probléma megfogalmazható, mint gráfelméleti feladat. A gráfalgoritmusok címszó alatt néhány fontos, a gyakorlati életben is gyakran előforduló feladatot, és a feladat megoldására használható algoritmust ismertetünk. A dolgozat megírásának a célja, hogy segítséget nyújtson a felsőoktatásban résztvevő hallgatóknak az alapvető gráfalgoritmusok elsajátításában, azonban haszonnal forgathatják középiskolások is, elsősorban programozás témakör tanulmányozása során, vagy programozás szakkörön.

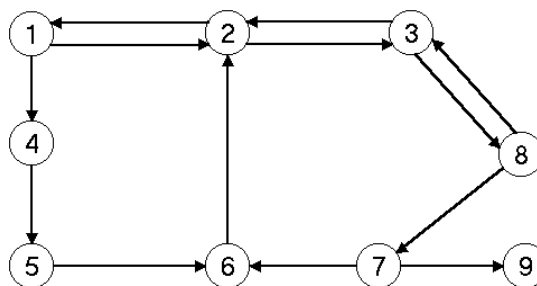
1. Néhány probléma modellezése gráfokkal

A gráfok szerteágazó felhasználásának illusztrálására nézzünk néhány olyan problémát, amelyekre természetes módon adódik a gráf, mint modell felhasználása.

1.1 Útkeresés

Probléma: Szeretnénk eljutni autóval Budapest egyik pontjáról egy másik pontjára. Rendelkezünk egy Budapest térképpel. [1]

Modell: Az útkereszteződéseknek, csomópontoknak megfeleltetjük a gráf csúcsait, az utcáknak, pedig a gráf éleket. Mivel nem minden utcán lehet mindkét irányba haladni az autóval (lehet csak egyirányú az utca), így **irányított éleket** használunk. Ezen kívül az eljutás egyéb jellemzője most közömbös számunkra, tehát eltekintünk az utcák (élek) jellemzésétől (tulajdonságaitól), ezért az éleink legyenek **súlyozatlanok**.



Tehát a **gráf típusa**:

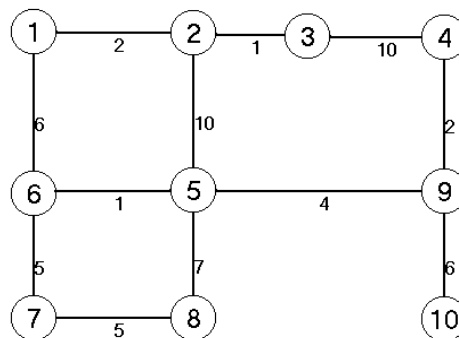
- irányított
- súlyozatlan

Feladat: Keressünk utat az egyik pontból a másikba, azaz keressük az éleknek egy olyan sorozatát, amelyek mentén haladva (szabályosan) eljuthatunk a kezdőcsúcsból a célcsúcsba.

1.2 Minimális költségű út keresése

Probléma: Szeretnénk autóval eljutni az egyik városból egy másik városba. Vegyük figyelembe, az egyes utak költségeit (pl.: benzinköltség, úthasználati díj) és előnyeit (pl.: szép panoráma, nincs dugó stb.).

Modell: A városoknak megfeleltetjük a gráf csúcsait, a városokat összekötő úthálózatnak, pedig a gráf éleit. Általában feltehetjük, hogy két város között, ha van közvetlen országút, akkor azon oda-vissza egyaránt eljuthatunk (**szimmetrikus reláció**), így a gráf élei legyenek **irányítatlanok**. Az országutak jellemzőit megpróbáljuk kvalitatív leírni. Ezen jellemzők lesznek az egyes élek tulajdonságai, amelyet leggyakrabban költségeknek (súlyoknak) tekintünk, ezért a probléma gráfját **élsúlyozottnak** választjuk.



Tehát a **gráf típusa**:

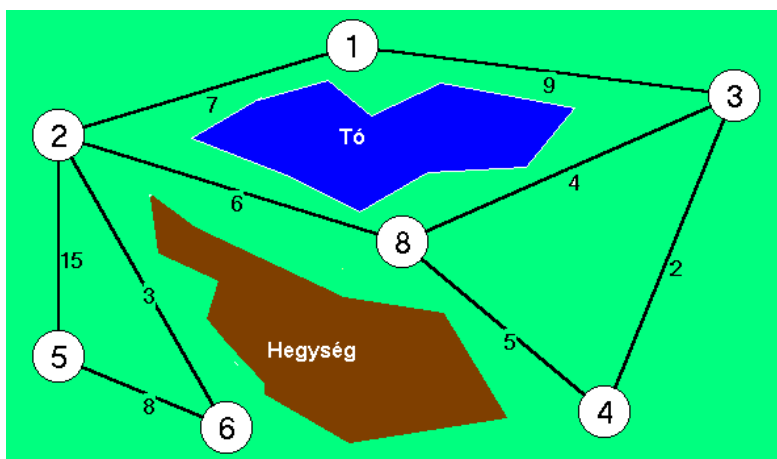
- irányítatlan
- élsúlyozott

Feladat: Keressünk utat az egyik pontból a másikba, de ezen út legyen számunkra a legkevésbé rossz, azaz legyen az egyik pontból a másik pontba vezető utak közül a **legkisebb költségű**.

1.3 Minimális költségű feszítőfa keresése

Probléma: Néhány város elektromos ellátását szeretnénk megoldani, azaz elektromos hálózatot szeretnénk telepíteni a városok között. Adottak a városok közötti elektromos vezetékek kiépítésének költségei. Vannak városok, amelyek között (a földrajzi körülmények miatt) vezeték nem vagy csak "óriási" költséggel telepíthető. [2]

Modell: A városoknak megfeleltetjük a gráf csúcsait, az elektromos vezetékeknek a gráf éleit. Mivel a vezetéken az áram "mindkét irányba folyik" (**szimmetrikus reláció**), ezért a gráfunk legyen **irányítatlan**. A vezeték kiépítésének költsége (a kapcsolat egy tulajdonsága) pedig legyen az él egy tulajdonsága, és nevezzük az **él súlyának** vagy költségének



Tehát a **gráf típusa**:

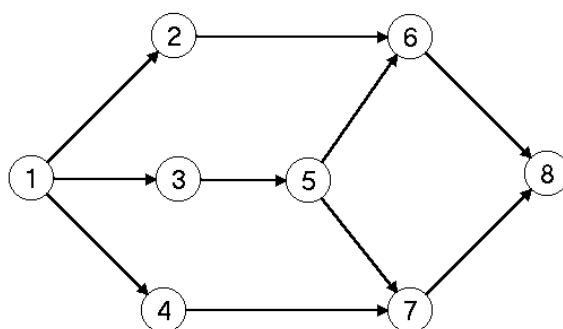
- irányítás nélküli
- élsúlyozott

Feladat: Határozzuk meg a gráffal ábrázolt elektromos hálózatnak egy olyan részhálózatát, amelyre teljesül, hogy bármely két város összeköttetésben van (közvetlen vagy közvetve), és ezen tulajdonságú részhálózatok közül, az illető részhálózat telepítésének összköltsége a legkisebb.

1.4 Irányított körmentes gráf (DAG¹) topologikus rendezése

Probléma: Tegyük fel, hogy egy összetett tevékenység (pl.: egy termék előállítása, a továbbiakban nevezzük gyártásnak) felbontható résztevékenységekre. Szeretnénk készíteni egy gyártási technológiát, azaz a résztevékenységek olyan szekvenciális sorozatát, amelyet mintegy "receptet" végrehajtva, az összetett tevékenység elvégezhető (elkészíthetjük a terméket). Vegyük figyelembe, hogy a **szekvenciális sorrendben** bizonyos tevékenységeknek meg kell előzniük más tevékenységeket, különben fizikailag ellentmondásos lenne a technológiai leírás (pl.: a palacsintát addig nem tölthetjük meg, míg a tésztát meg nem sütöttük). [2]

Modell: A tevékenységeknek megfeleltetjük a gráf csúcsait, az említett megelőzési relációnak pedig a gráf éleit. Mivel a megelőzési reláció nem megfordítható, ezért **irányított gráfot** használunk. Mivel a probléma során nem foglalkozunk az egyes folyamatok súlynak megfeleltethető tulajdonságaival, így az **éleket súly nélkül** tekintjük.



¹ Directed Acyclic Graph

Tehát a **gráf típusa**:

- irányított
- súlyozatlan

Feladat: Írjuk fel a tevékenységek egy olyan szekvenciális sorrendjét, ahol az i -edik tevékenységet nem előzheti meg a j -edik tevékenység ($i \neq j$), ha a gyártás "fizikai" sorrendjében a j -edik tevékenység közvetlenül vagy közvetve az i -edik tevékenységre vagy annak eredményére épül.

Pl.: 1, 2, 3, 4, 5, 6, 7, 8 vagy 1, 3, 5, 2, 6, 4, 7, 8 sorozatok, a feladatban előírt tulajdonsággal rendelkező szekvenciális sorozatok

1.5 Erősen összefüggő komponensek meghatározása

Probléma: Adott Budapest térképe és szeretnénk eldönteni, hogy a város bármely pontjából bármely pontjába eljuthatunk autóval vagy sem. Például az 1.1-es példában látott gráfon, a 9-es pontba el tudunk jutni, de onnan "szabályosan" kijönni már nem tudunk (egyirányú zsákutca). [2]

Modell: Legyen ugyan az, mint az 1.1-es útkeresés problémánál, **irányított** és **súlyozatlan** gráf.

Feladat: Osztályozzuk a gráf pontjait oly módon, hogy egy osztályon belül az egyik pontból a másikba el tudok jutni, és viszont, de ez különböző osztálybeli pontokra ne teljesüljön. Tehát adjuk meg a gráf **erősen összefüggő** komponenseit. Ha a gráf minden pontja egy erősen összefüggő komponensbe esik, akkor bármely pontból, bármely pontba el tudok jutni autóval.

2. Alapfogalmak, jelölések

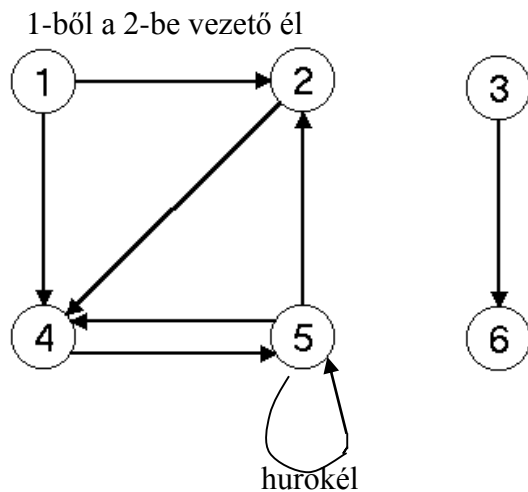
A továbbiakban ismertnek feltételezzük az alapvető gráfelméleti fogalmakat, definíciókat és tételeket. Most nézzünk néhány fontosabb fogalmat kevésbé formálisan, inkább csak a felelevenítés szintjén (fogalmak, jelölések [2]-ből).

Irányított gráf: $G = (V, E)$ pár, ahol V a csúcsok véges halmaza (általában $1, 2, 3, \dots, n$), $E \subseteq V \times V$ pedig az élek halmaza.

él: $e = (u, v) \in E$, ahol $u, v \in V$

csúcsok száma: $n = |V|$

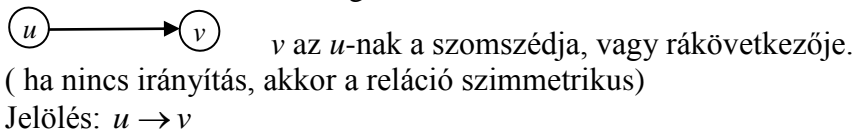
élek száma: $e = |E|$



Az irányított gráf definíciójának kiterjesztése **irányítás nélküli**: $G = (V, E)$, amelyre, ha $(u, v) \in E$, akkor $(v, u) \in E$ is teljesül (E a V elemeiből alkotott rendezetlen párok egy részhalmaza, jelölés: $[u, v]$ rendezetlen pár).

G **egyszerű gráf**, ha nincs benne hurokél és többszörös él.

Szomszéd/rákövetkező fogalma:



Út: $\langle v_0, v_1, \dots, v_k \rangle$ sorozat k hosszúságú út, ha $\forall i \in [1..k]: (v_{i-1}, v_i) \in E$

Jelölés: $v_0 \leadsto v_k$

Kör: egy k hosszúságú út kör, ha $v_0 = v_k$.

A körmentes utat **egyszerű útnak** nevezzük.

A kört **egyszerű körnek** nevezzük, ha nincs benne belső kör, azaz kör, és minden v_1, \dots, v_k esetén páronként különböző csúcsokból áll.

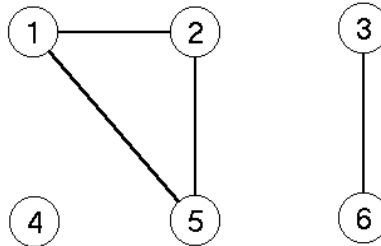
Körmentes gráf: a kört nem tartalmazó gráf.

Fokszám:

- Irányítás nélküli gráf fokszáma a csúcsból kiinduló élek száma
- Irányított gráfnál megkülönböztetjük a **befokot** (bemenő élek száma) és a **kifokot** (kimenő élek száma). Ekkor a fokszámot ezek összege adja.

Összefüggőség:

G Irányítás nélküli gráf összefüggő \Leftrightarrow bármely két csúcs összeköthető úttal \Leftrightarrow a gráf egy darab összefüggő komponensből áll



Pl.: A fenti gráf összefüggő komponensei: $\{1,2,5\} \{3,6\} \{4\}$

G irányított gráf **erősen összefüggő** \Leftrightarrow bármely két csúcs összeköthető úttal (figyelembe véve az irányítást, tehát $u \leadsto v$ és $v \leadsto u$ egyaránt kell, hogy teljesüljön)

Teljes gráf: Olyan irányítás nélküli gráf, amelynek bármely két csúcsa szomszédos

Páros gráf: Olyan irányítás nélküli gráf, amelynek csúcsai két, diszjunkt halmazra bonthatók, és él csak a két különböző halmaz csúcsai között mehet, azonos halmazban lévő csúcsok között nem. (lásd: Arthur király házassági problémái)

Erdő: körmentes, irányítás nélküli gráf.

Fa: Összefüggő, körmentes, irányítás nélküli gráf.

Élsúlyok: $c : E \rightarrow R$ egy olyan valós értékű függvény, amelynek értelmezési tartománya a gráf élhalmaza.

3. Gráfok ábrázolása

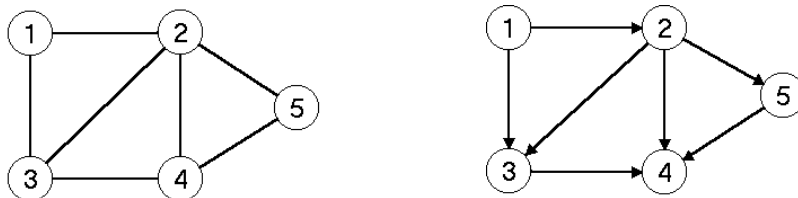
A gráfok ábrázolására két, a gyakorlatban igen elterjedt adatszerkezetet adunk. Az egyik tisztán **aritmetikai ábrázolású** (szomszédsági-mátrix), a másik vegyes, **aritmetikai és láncolt ábrázolású** (éllista).

3.1 Szomszédsági-mátrix (adjacencia-mátrix , csúcsmátrix)

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 $1, \dots, n$). Egy mezőben akkor van 1-es, ha a hozzá tartozó oszlop által meghatározott csúcs szomszédja a sor által meghatározott csúcsnak.

$$, \text{azaz } A[i, j] = \begin{cases} 0 & , ha (i, j) \notin E \\ 1 & , ha (i, j) \in E \end{cases} \quad ([2] \text{ szerinti definíció})$$

Pl.:



A fenti irányított gráf esetén:

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

A fenti irányítatlan gráf esetén:

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix} \quad (\text{szimmetrikus})$$

Ha **súlyozott a gráf**, akkor a súlyokat (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 ∞ 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)” [2]. A mátrix főátlójába kerülnének a **hurokélek költségei**, ami azt takarja, hogy az illető élből önmagába ilyen költséggel tudunk eljutni. Ennek, az elkövetkezendő feladatok esetén, számunkra nincs jelentősége, mivel, ha eljutottunk egy csúcsba, akkor további költség terhe mellett nem akarunk a csúcsba továbbra is "körözni" (mivel az elkövetkezendő algoritmusok jelentős része minimális költségű problémák megoldására törekszik). Tehát, ha már eljutottunk egy csúcsba,

akkor a következő lépésekben az ott tartózkodás költsége legyen 0 ([2]-beli def.). (A gyakorlati életben ritkán előforduló feladatoknál lehet értelme, pl.: negatív kör vagy hurokél egy minimális költségű útkeresésnél, ekkor végtelen nagy haszonra tudunk szert tenni (lásd legrövidebb utak egy forrásból feladatai között található arbitrázs feladat).

Tehát élsúlyozott gráf esetén:

$$C[i, j] = \begin{cases} 0 & , ha i = j \\ c(i, j) & , ha i \neq j \text{ és } (i, j) \in E \\ \infty & , különben \end{cases} \quad ([2] \text{ szerinti definíció, } * \text{ helyett végtelen})$$

Helyfoglalás: A helyfoglalás mindig ugyanakkora, független az élek számától, a mátrix méretével n^2 -tel arányos. (Az n pontú teljes gráfnak is ekkora a helyfoglalása.) Tehát sűrű gráfok esetén érdemes használni, hogy ne legyen túl nagy a helypazarlás. [1]

3.2 Szomszédsági lista (éllista)

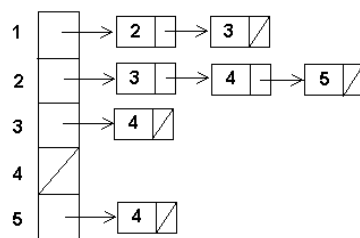
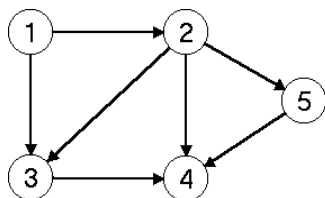
A Gráf minden csúcsához egy listát rendelünk. Ezen listában tartjuk nyilván az adott csúcsból kimenő éleket.

Megvalósítás: Vegyünk fel, egy mutatókat tartalmazó $Adj[1..n]$ tömböt (a csúcsokkal indexeljük a tömböt). A tömbben lévő mutatók mutatnak az éllistákra. (Az éllisták szokásos listák, lehetnek egy vagy kétirányú, fejelemes vagy fejelem nélküli, ez most nem lényeges a gráf szempontjából.) **Írányított gráf** esetén, az éllisták listaelemei reprezentálják az éleket. Az élnek megfelelő listaelemet abban a listában tároljuk, amelyik csúcsból kiindul az él, és a célsúcs indexét eltároljuk a listaelemben. Tehát az $(i, j) \in E$ él megvalósítása: az i -edik listában egy olyan listaelemmel, amelyben eltároltuk a j -t, mint az él célsúcsát. **Írányítatlan gráf** esetén, egy élnek két listaelemet feleltetünk meg, azaz egy irányított élt egy oda-vissza mutató, irányított élpárral valósítunk meg a korábban említett módon. ([1],[2] szerinti megvalósítás)

Élsúlyozott gráf esetén, az él súlyát is a listaelemben fogjuk tárolni. ([1],[2] szerinti megvalósítás)

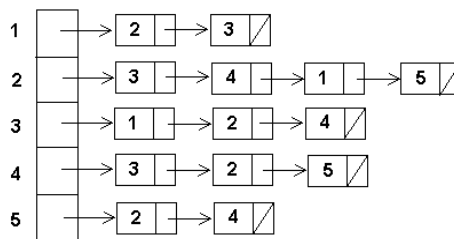
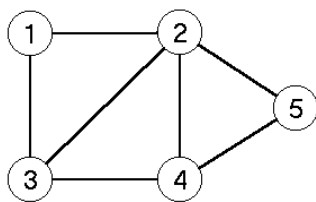
Példa:

Írányított gráf esetén:



Helyfoglalás: $n + e$

Írányítatlan gráf esetén:



Helyfoglalás: $n + 2e$

Helyfoglalás: Mivel a memóriaigény az élek számával arányos, ezért az éllistas ábrázolást ritka gráfok ($|E| \ll |V|^2$) esetén szokták használni. Ugyanis sűrű gráf esetén a szomszédsági mátrixhoz képest, itt még jelentkezik, a listák láncolásából származó (mutatók), plusz helyfoglalás. [1]

3.3 Ellenőrző kérdések

1. A szomszédsági mátrixban hol találhatók az 1-es csúcsból kimenő élek súlyai?
2. A szomszédsági mátrixban hol találhatók az 1-es csúcsba bemenő élek súlyai?
3. A szomszédsági listás ábrázolás esetén hol találhatók az 1-es csúcsból kimenő élek súlyai?
4. A szomszédsági listás ábrázolás esetén hol találhatók az 1-es csúcsba bemenő élek súlyai?
5. A szomszédsági-mátrixban hol találhatók a hurokélek súlyai?
6. A szomszédsági-mátrixban hogyan jeleníthetők meg a párhuzamos élek?
7. Mi mondható el az irányítatlan gráf mátrixáról?
8. Összességében hány listaeleme van a 10 irányítatlan élt tartalmazó gráf éllistáinak?
9. Milyen gyorsan nézhetjük végig egy adott csúcsból kiinduló éleket az egyes ábrázolások esetén?
10. Az egyes ábrázolások esetén milyen gyorsan dönthető el, hogy egy (i,j) pár éle-e a gráfnak?
11. Adott egy irányított gráfot éllistas ábrázolásban. Mennyi "idő" alatt határozható meg egy csúcs kimeneti fokszáma? Mennyi "idő" alatt határozható meg egy csúcs bemeneti fokszáma?

3.4 Gyakorló feladatok

1. Valósítsa meg a gráf típust (élsúlyozott és súlyozatlant is)
 - a csúcsmátrixos ábrázolásban!
 - b Éllistas ábrázolásban!

Műveletek:

 - Él beszúrás, törlés
 - u csúcsnak szomszédja-e v?
 - Egy u csúcs Szomszéd(u) halmazának a megadása.

2. Adjunk meg egy olyan típust/adatszerkezetet (műveletekkel), amely egy gráfot valósít meg csúcsmátrixos ábrázolásban. A szokásos műveletek mellett biztosítson lehetőséget a csúcsok beszúrására és törlésére!
3. Adjon meg egy olyan típust/adatszerkezetet (műveletekkel), amely egy gráfot valósít meg éllistas ábrázolásban. A szokásos műveletek mellett biztosítson lehetőséget a csúcsok beszúrására és törlésére!
4. Adjon meg egy olyan típust/adatszerkezetet (műveletekkel), amely egy gráfot valósít meg teljesen láncolt ábrázolásban, amely az éllistas ábrázolástól annyiban különbözik, hogy a csúcsok „tömbje” is láncoltan van ábrázolva. A szokásos műveletek mellett biztosítson lehetőséget a csúcsok beszúrására és törlésére!
5. Adjon $O(n + e)$ idejű algoritmust, amely meghatározza egy irányított gráf esetén, a gráf csúcsainak kimeneti fokát és bemeneti fokát. A program a kimeneti fok és bemeneti fok értékeket, a *kifok*[1..n] és *befok*[1..n] tömbökbe írja be, amely tömb a csúcsok sorszámaival (címkéjével) van indexelve. [5]
 - a A gráf csúcsmátrixos ábrázolással van megadva.
 - b A gráf éllistas ábrázolással van megadva.
6. Adjon algoritmust, amely egy gráf csúcsmátrixos ábrázolásából előállítja a gráfot éllistas ábrázolásban!
 - a A gráf legyen élsúlyozás nélküli.
 - b A gráf legyen élsúlyozott.
7. Adjon algoritmust, amely egy gráf éllistas ábrázolásából előállítja a gráfot csúcsmátrixos ábrázolásban!
 - a A gráf legyen élsúlyozás nélküli.
 - b A gráf legyen élsúlyozott.
8. Adjon $O(n + e)$ idejű algoritmust egy éllistával ábrázolt $G = (V, E)$ irányítatlan és súlyozatlan gráf, éllistas ábrázolásban megadott, egyszerűsített gráfjának, $G' = (V, E')$ -nek az előállítására. ($G' = (V, E')$ gráf a $G = (V, E)$ egyszerűsített gráfja, ha E' ugyanazokat az éleket tartalmazza, mint E , azzal a különbséggel, hogy E' -ben egy éllel helyettesítjük az E -beli többszörös éleket, és a hurokéleket elhagyjuk.)
9. Adjon algoritmust, amely előállítja egy irányított gráf transzponáltját/fordítottját. ($G = (V, E)$ irányított gráf transzponáltja a $G^T = (V, E')$ irányított gráf, ha $E' = \{(u, v) \in V \times V \mid (v, u) \in E\}$, azaz G^T -t úgy kapjuk G -ből, hogy minden él irányítását megfordítjuk.)
 - a Csúcsmátrixos ábrázolás esetén, konstans mennyiségű többletmemória felhasználásával.
 - b Éllistas ábrázolás esetén, tetszőleges mennyiségű többletmemória felhasználásával.

c Éllistas ábrázolás esetén, konstans mennyiségű többletmemória felhasználásával.

10. Csúcsmátrixos ábrázolás esetén a legtöbb gráf algoritmus futási ideje $\Theta(n^2)$, azonban van kivétel. Egy irányított gráf csúcsa univerzális nyelő, ha bemeneti foka $n-1$ és kimeneti foka 0. Mutassa meg, eldönthető $O(n)$ idő alatt, hogy egy csúcsmátrixszal megadott irányított gráfban van-e univerzális nyelő csúcs! [6]

11. Adott egy n pontú irányítatlan gráf. Adjon algoritmust, amely eldönti, hogy van-e a gráfnak olyan pontja, amely minden más ponttal össze van kötve! Nevezzük ezt a pontot röviden teljes pontnak. [6]

12. Adott $G = (V, E)$ irányított gráf. Állítsa elő $G' = (V, E')$ irányított gráfot, ahol $E' = \{(u, w) \in V \times V \mid \exists v \in V : (u, v) \in E \wedge (v, w) \in E\}$.

a A gráf csúcsmátrixos ábrázolással van megadva.

b A gráf állistas ábrázolással van megadva.

3.5 Összefoglalás

- Gráfok ábrázolása: szomszédsági-mátrix (csúcsmátrix), szomszédsági lista (éllista)
- Szomszédsági mátrix: csúcsokkal indexelt mátrix, az (i, j) eleme reprezentálja az i -ből j -be menő élt, amelyek tartalma:
 - súlyozatlannál 0,1 értékek (igaz, hamis)
 - súlyozottnál az élsúlyok, végtelen extrémális elemmel a nem létező élek esetén
- Szomszédsági lista: csúcsokkal indexelt vektor, melyek az éleket reprezentáló elemek listáját tartalmazza. Egy i -ből j -be menő él megvalósítása: a vektor i -dik listájában j -t tartalmazó listaelem. Súlyozott élek esetén a listaelem az élek súlyát is tartalmazza.
- Helyfoglalás: szomszédsági mátrix esetén n^2 -tel arányos, éllista esetén $n + 2e$ -vel arányos. Helytakarékosági szempontokat figyelembe véve ritka gráfok esetén inkább éllistát, sűrű gráfok esetén inkább csúcsmátrixot használjunk.

4. Bejárési stratégiák, szélességi bejárás

A bejárési/keresési algoritmusok feladata a gráf felderítése, szerkezetének a megismerése vagy adott tulajdonságú csúcs keresése stb. Az ismertetésre kerülő bejárési algoritmusok a csúcsok elérésének stratégiájában különböznek. Két bejárési algoritmust tanulunk a félév során, a szélességi és a mélységi bejárást. Most nézzük a szélességi bejárást, néhány fejezettel később, pedig megvizsgáljuk a mélységi bejárást.

4.1 Bejárési/keresési stratégiák

A bejárési stratégiák **alapötletéhez** tekintsük az alábbi kis példát a *Rónyai-Ivanyos-Szabó: Algoritmusok* c. tankönyvből [2].

Van egy középkori kisváros, ahol az utcai lámpákat egy korosodó lámpagyújtogató ember gyújtja fel. Este az illető egymaga indul munkába. A város főterén felgyújtja az első lámpát, majd a főtérről kivezető egyik utcában gyújtogatja a lámpákat. Amikor egy elágazáshoz ér, valamelyik utcába befordulva folytatja tevékenységét. Elérve a város szélére, egy zsákutcába vagy egy olyan utcába jutva, ahol már égnek a lámpák, az öreg visszamegy az előző útelágazáshoz, és egy olyan utcába fordul, amelyikben még nem égnek a lámpák. Végül visszaérve a főtérre (persze a főtérről kivezető összes utat már bejárta), elvégezve aznap esti munkáját hazatér aludni. Ezt az elvet használja a mélységi keresés.

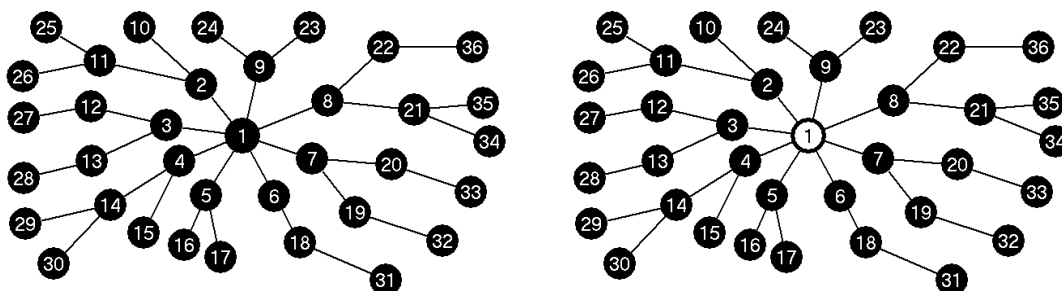
Egy másik este a lámpagyújtogató fáradtnak és gyengének érzi magát, hát szól a barátainak, hogy segítsenek neki az esti munkájában. (Nagyon sok barátja van az illetőnek.) A csapat kimegy este a főtérre, és a következő stratégia szerint gyújtogatja a lámpákat. A főtéren meggyújtják a lámpákat, majd annyi felé oszlik a csapat, ahány főtérről kivezető út van. Mindegyik csapat elindul egy kivezető úton, és út közben felgyújtja a lámpákat. Amikor a csapat egy útelágazáshoz ér, szétoszlik kisebb csapatokra, és mindegyik kisebb csapat tovább indul az elágazás egyik még sötét utcáján. Amikor egy csapat már nem tud tovább menni (város széle, zsákutca, minden lámpa ég a környező utcákban), akkor a csapattagok hazamennek aludni. Ez az elve a szélességi keresésnek.

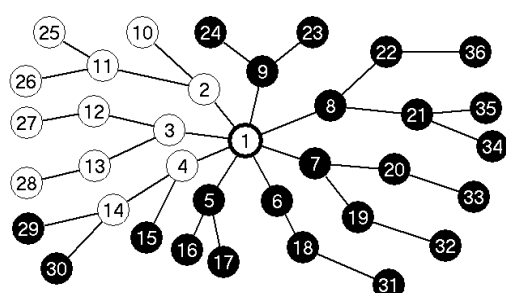
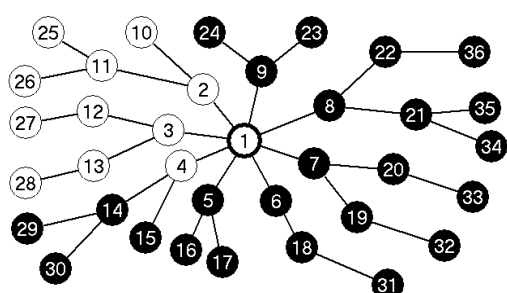
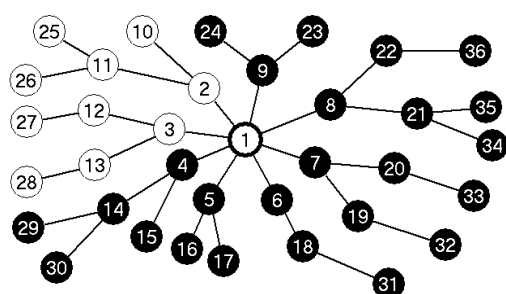
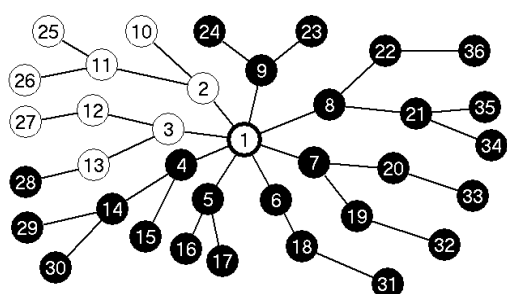
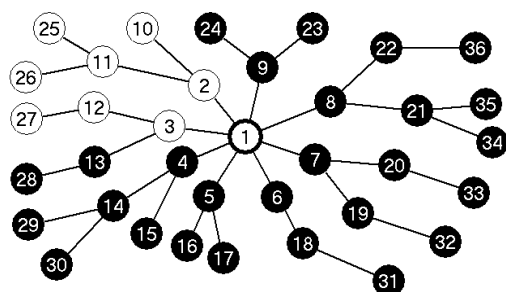
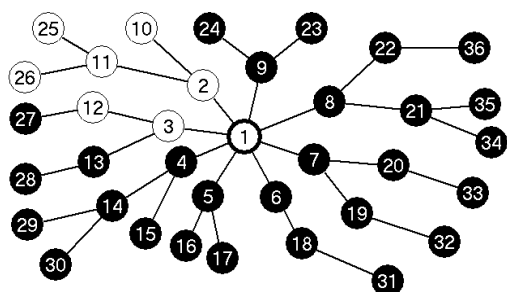
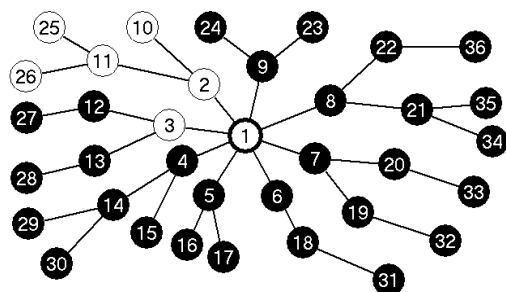
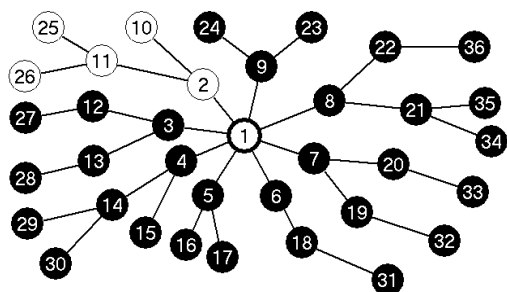
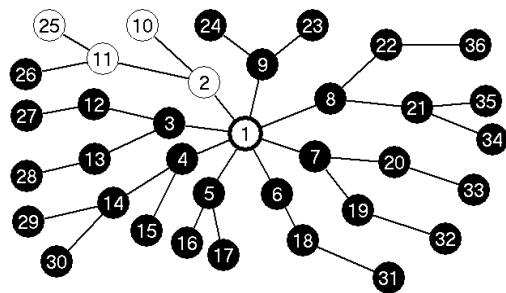
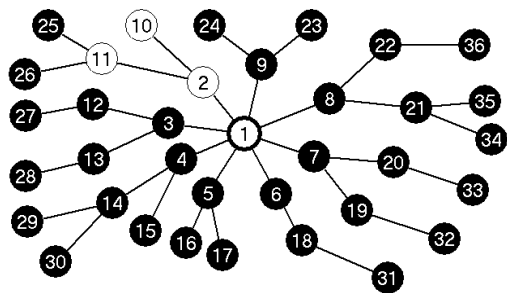
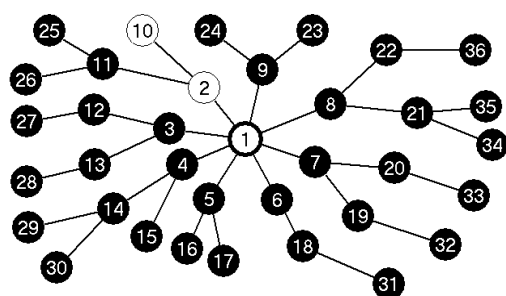
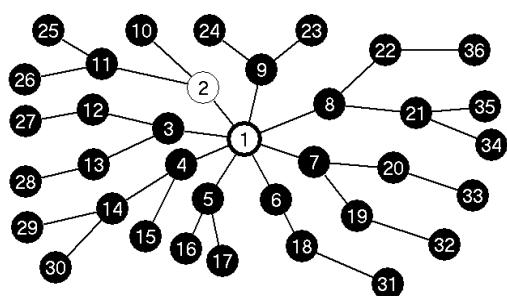
Ha fentről néznénk a várost, ahogy kigyulladnak a lámpák, a következőket látnánk. Az első esetben a főtérről a város széle felé haladó út mentén gyulladnak ki a lámpák, majd a város szélénél egy-egy nagyobb foltok kezdenek kivilágosodni, míg a város közepén még mindig lesznek sötét területek. A második esetben, a középpontból a város széle felé egyre nagyobb sugarú körben terjed a világosság.

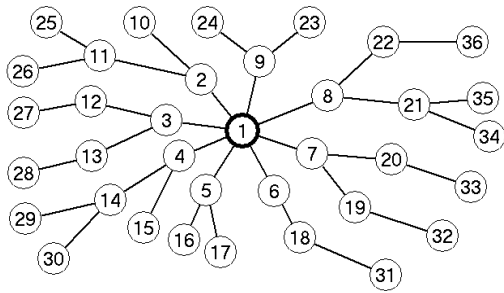
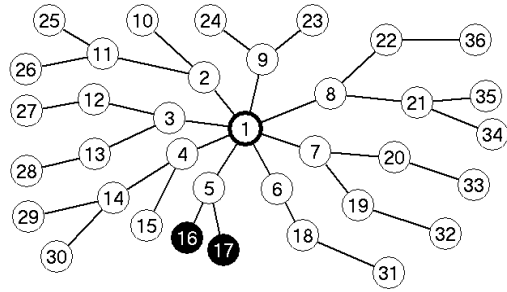
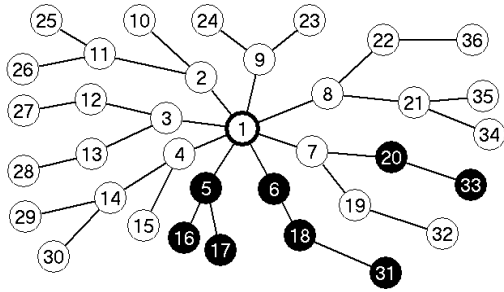
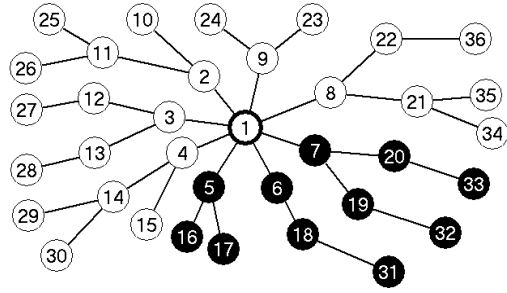
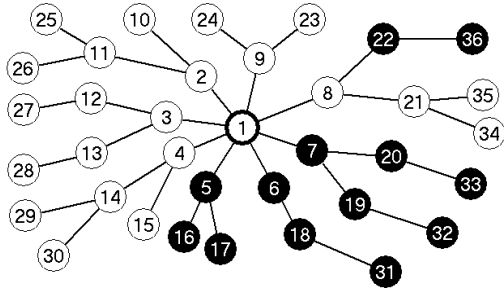
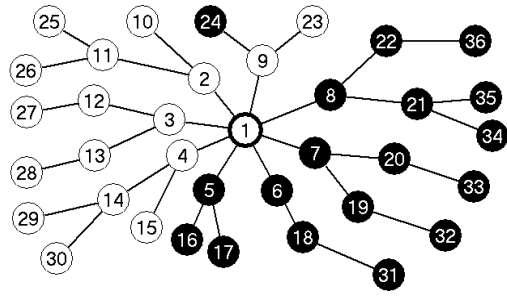
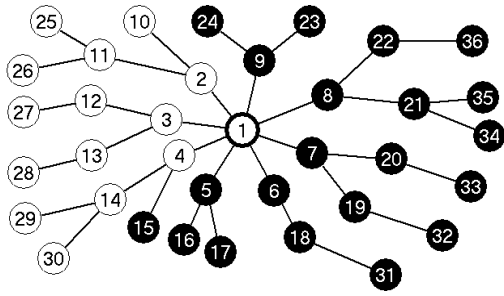
Most nézzük az említett város felülnézetét:

1) Mélységi stratégia:

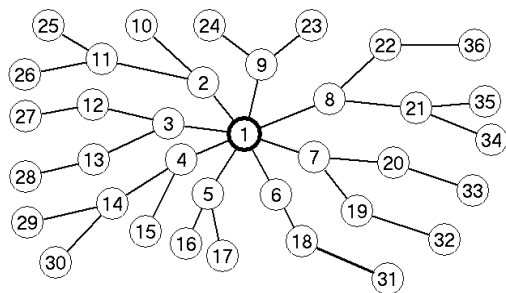
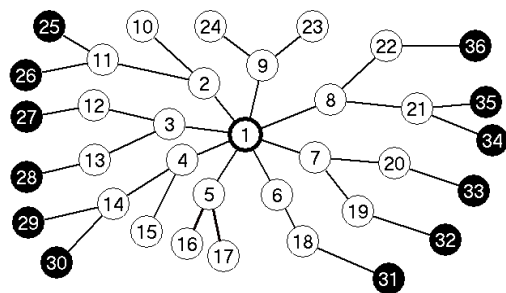
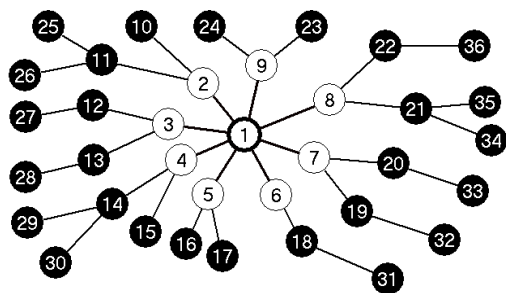
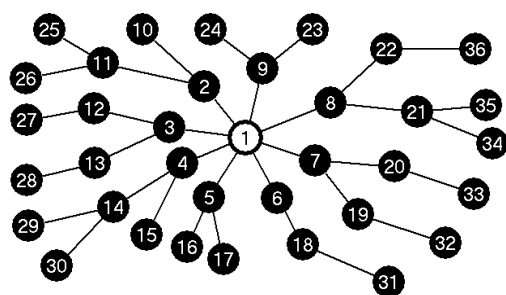
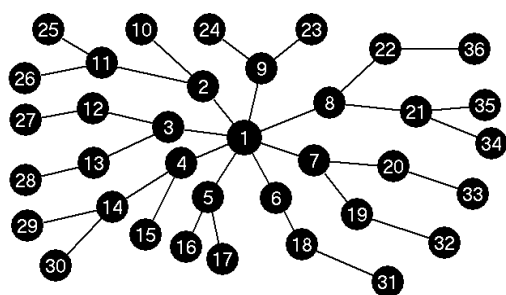
Az első néhány lépés után, már csak a teljesség igénye nélkül néhány pillanatfelvétel.





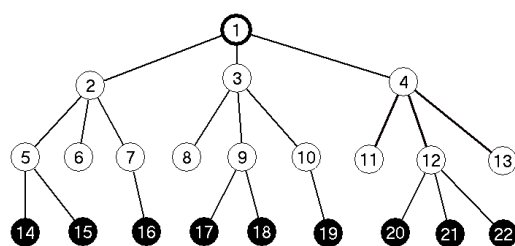
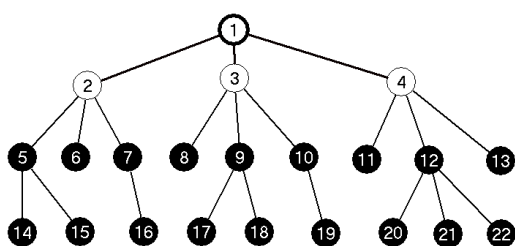
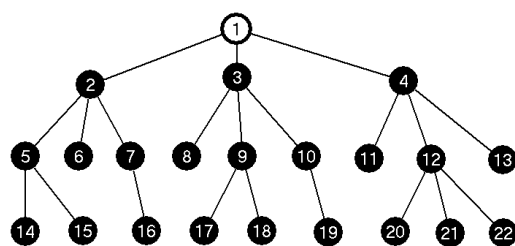
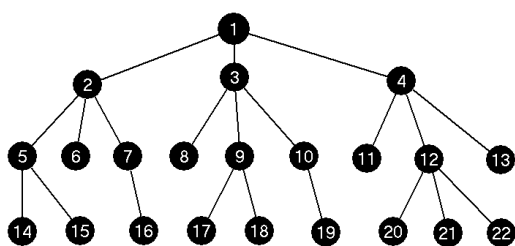


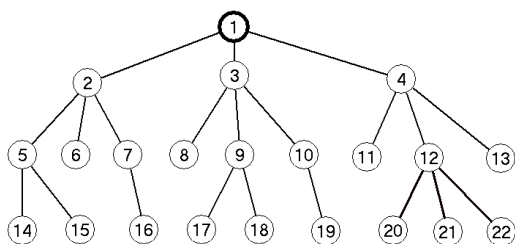
2) Szélességi stratégia:



A példából is kitűnik, hogy a szélességi bejárás könnyen párhuzamosítható.

Mindenki emlékszik még a fáknál tanult **szintfolytonos bejárásra**. Nézzük meg az alábbi példán:





Az ábrákon is jól látható, hogy a szintfolytonos bejárás a szélességi bejárás speciális esete fákra alkalmazva, a fa gyökerét véve kezdőcsúcsnak.

4.2 Szélességi bejárás/keresés algoritmusa

4.2.1 Definíció [1]: Legyen $G=(V,E)$ gráf és $s, u \in V$ csúcsok, és $s \rightsquigarrow u$ út

$\langle v_0, v_1, \dots, v_k \rangle$, ahol $s=v_0$, $u=v_k$.

Az **út hossza** legyen, az út mentén érintetett élek száma, azaz

$$|s \rightsquigarrow u| = |\langle v_0, v_1, \dots, v_k \rangle| - 1 = k$$

Az **u csúcs s-től való távolsága** legyen az $s \rightsquigarrow u$ utak közül a legrövidebb élszáma, azaz

$$d(s, u) = \min \{ |s \rightsquigarrow u| \}$$

Ha nincs $s \rightsquigarrow u$ út a gráfban, akkor legyen $d(s, u) = \infty$

Feladat: Adott egy G irányított vagy irányítás nélküli, véges gráf. Írjuk ki a csúcsokat egy $s \in V$ **kezdőcsúcs**tól való távolságuk növekvő sorrendjében. Minden csúcsra jegyezzük fel a kezdőcsúcs-tól való távolságát, és a hozzá vezető (egyik) legrövidebb úton a megelőző csúcsot. Az azonos távolságú pontok egymás közötti sorrendje, a feladat szempontjából legyen tetszőleges.

Az **algoritmus elvét** az előzőekben már láttuk, most foglaljuk össze röviden:

- 1) Először elérjük a kezdőcsúcsot.
- 2) Majd elérjük a kezdőcsúcs-tól 1 távolságra lévő csúcsokat (a kezdőcsúcs szomszédait)
- 3) Ezután elérjük s -től 2 távolságra lévő csúcsokat (a kezdőcsúcs szomszédainak a szomszédait), és így tovább.
- 4) Ha egy csúcsot már elértünk, akkor a későbbi odajutásoktól el kell tekinteni

Hogyan tudjuk biztosítani a fenti elérési sorrendet? Nézzük meg **ADT** szinten.

Az elérési sorrendnél azt kell figyelembe venni, hogy amíg az összes kezdőcsúcs-tól k (≥ 0) távolságra lévő csúcsot ki nem írtuk, addig nem szabad k -nál kisebb vagy nagyobb távolságú csúcsokat kiírni, sőt mire egy k távolságú csúcsot kiírunk, már az összes k -nál kisebb távolságú csúcsot ki kellett írunk.

Egy $k+1$ távolságú csúcs biztosan egy k távolságú csúcs szomszédja (az egyik legrövidebb úton a megelőző csúcs biztosan k távolságra van a kezdőcsúcs-tól). \Rightarrow A $k+1$ távolságú csúcsokat a k távolságú csúcsok szomszédai között kell keresni (nem biztos, hogy az összes szomszéd $k+1$ távolságú, lehet, hogy egy rövidebb úton már elértük.).

Használjunk **sor** adattípust és biztosítsuk azt az invariáns tulajdonságot, hogy a sorba csak k vagy $k+1$ távolságú csúcsok lehetnek az eléérésük sorrendjében, amely egyben az s -től való távolságuk szerinti (növekedő) sorrendnek is megfelel. Ameddig ki nem ürül a sor, vegyünk ki

az első elemet, írjuk ki és terjesszük ki, azaz a még "meg nem látogatott" szomszédait érjük el és rakjuk be a sorba.

4.2.2 Állítás: Az említett ciklust végrehajtva teljesül a fenti invariáns és a csúcsokat távolságuk sorrendjében érjük el és írjuk ki.

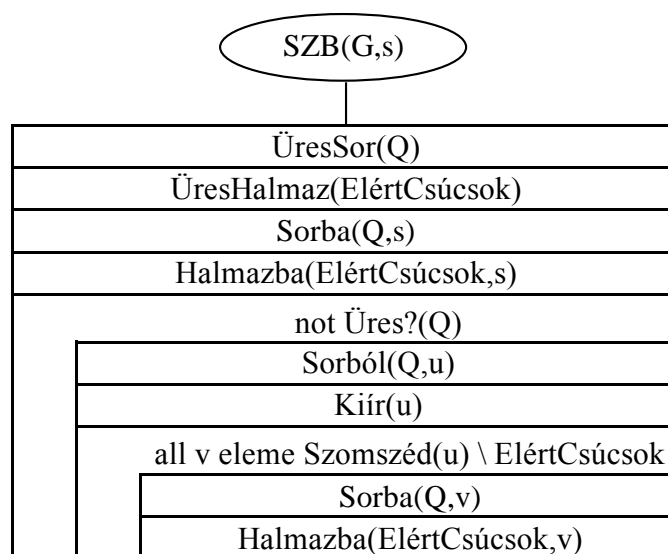
Bizonyítás: Teljes indukcióval: (k távolságú csúcsok a sorban megelőzik a $k+1$ távolságú csúcsokat, és a sorban lévő csúcsok távolságának az eltérése legfeljebb 1)

- $k=0$: Induláskor berakjuk a sorba a kezdőcsúcsot, ami 0 távolságra van. Kivesszük a kezdőcsúcsot a sorból és kiírjuk, majd a szomszédait, az 1 távolságra lévő csúcsokat rakjuk a sorba.
- $k \rightarrow k+1$: Indukciós feltevés szerint a sorba csak k és $k+1$ távolságú csúcsok vannak távolságuk szerint növekvően, és a sor invariánsa, hogy a korábban bekerült csúcsot korábban veszi ki. \Rightarrow Az összes k távolságú csúcsot kiterjesztjük, mielőtt egy $k+1$ távolságú csúcsot kivennénk, és amíg ki nem vettük az összes k távolságú csúcsot, addig csak $k+1$ távolságú csúcsokat rakunk a sorba. Csak az első $k+1$ távolságú csúcs kivételénél kerülhet a sorba $k+2$ távolságú csúcs, de addigra már az összes $k+1$ távolságú csúcs a sorban lesz, mert az összes k távolságú csúcsot kiterjesztettük. $\Rightarrow k+1$ távolságú csúcsok megelőzik a $k+2$ távolságú csúcsokat és a távolság különbség is mindig legfeljebb 1 marad.

Ha egy csúcsot egyszer már elértünk, akkor később nem kell újra elérni. Használjunk egy halmazt, amelybe az **elért csúcsokat** rakjuk, kezdetben csak a kezdőcsúcsot rakjuk bele. Amikor egy csúcsot először elérünk, dobjuk be a halmazba, és minden csúcs kiterjesztésénél csak azokat a szomszédait tekintjük (rakjuk a sorba), amelyeket még nem értünk el, azaz nincsenek benne az elért halmazban \Rightarrow Minden csúcsot csak egyszer érünk el (teszünk a sorba). \Rightarrow Minden csúcsot csak egyszer írunk ki.

Mivel a csúcsok száma véges és minden csúcsot legfeljebb egyszer érünk el és terjesztünk ki.
 \Rightarrow **A bejárás biztosan terminál.**

Tehát az algoritmus **ADT szinten [5]:**



Az algoritmusban használt $Szomszéd(u)$ absztrakt függvény az $u \in V$ csúcs szomszédainak halmazát adja meg. A többi jelölés magától értetődik.

Most nézzük meg egy példán az algoritmus működését **ADS szinten:**

Az ADT szinten tárgyalt halmazt most a csúcsok színezésével valósítjuk meg, sőt a csúcsoknak nem csak két állapotát különböztessük meg, hanem az alábbi három állapotát [1]:

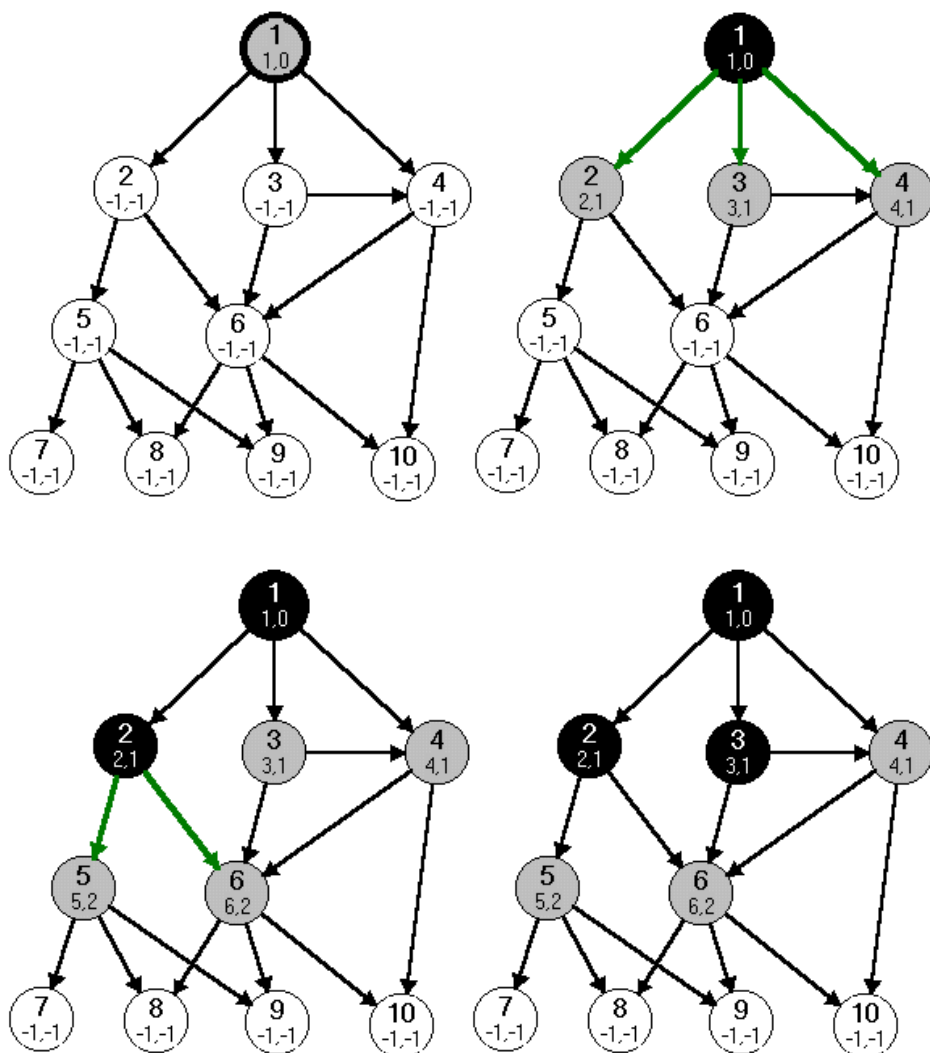
- 1) Amikor egy csúcsot még nem értünk el legyen fehér színű. Induláskor a kezdőcsúcs kivételével minden csúcs ilyen. ($u \notin Q$ és $u \notin ElértCsúcsok$)
- 2) Amikor egy csúcsot elérünk és bedobjuk a sorba, színezzük szürkére. A kezdőcsúcs induláskor ilyen. ($u \in Q$ és $u \in ElértCsúcsok$)
- 3) Amikor egy csúcsot kivettünk a sorból és kiterjesztettük (elértük a szomszédait), a színe legyen fekete. ($u \notin Q$ és $u \in ElértCsúcsok$)

ADT szinten egy csúcs szomszédainak az elérési sorrendjéről (a $Szomszéd(u) \setminus ElértCsúcsok$ feldolgozási sorrendjéről) nem tettünk fel semmit, azaz a szomszéd csúcsok elérése nem egyértelmű, tehát egy nem determinisztikus algoritmust kaptunk. Azonban gyakorlati feladatoknál néha megköveteljük a szomszéd csúcsok elérésének az egyértelműségét, hogy az algoritmus működése egyértelmű, azaz ellenőrizhető legyen (pl.: ZH-ban algoritmus szemléltetése). A következő példában a szomszéd csúcsok feldolgozási sorrendje legyen a csúcsok címkéje szerint növekedően rendezett.

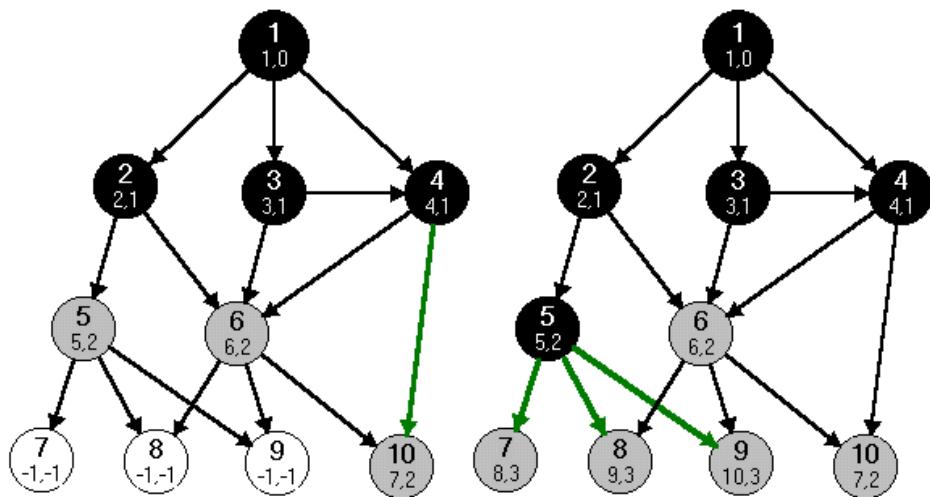
A következő ábra-sorozaton megfigyelhető a szélességi keresés algoritmus lépésenként. A csúcsokra, a címkén kívül, felírunk két pozitív egész számot. Az első szám megadja, hogy az illető csúcsot hányadikként íránk ki, a második szám, pedig a kezdőcsúcsból való távolságot tartalmazza. Kezdetben legyenek -1 extrémális értékűek. A kezdőcsúcs legyen az 1-es címkéjű csúcs.

Kezdetben minden csúcs fehér kivéve a 1-es csúcsot, amelyik szürke. A sorban is kezdetben csak az 1-es csúcs van. Az első lépésben kivesszük a sorból az 1-es csúcsot, majd kiterjesztjük, azaz elérjük az 1-es csúcs még fehér szomszédait (2,3,4), amelyeket szürkére színezzük, és bedobunk a sorba. Az 1-es csúcsot kiterjesztettük, tehát készen vagyunk vele, így feketére színezzük. Figyeljük meg, hogy a sor a szürke csúcsokból áll, az elérési szám (első szám) szerint

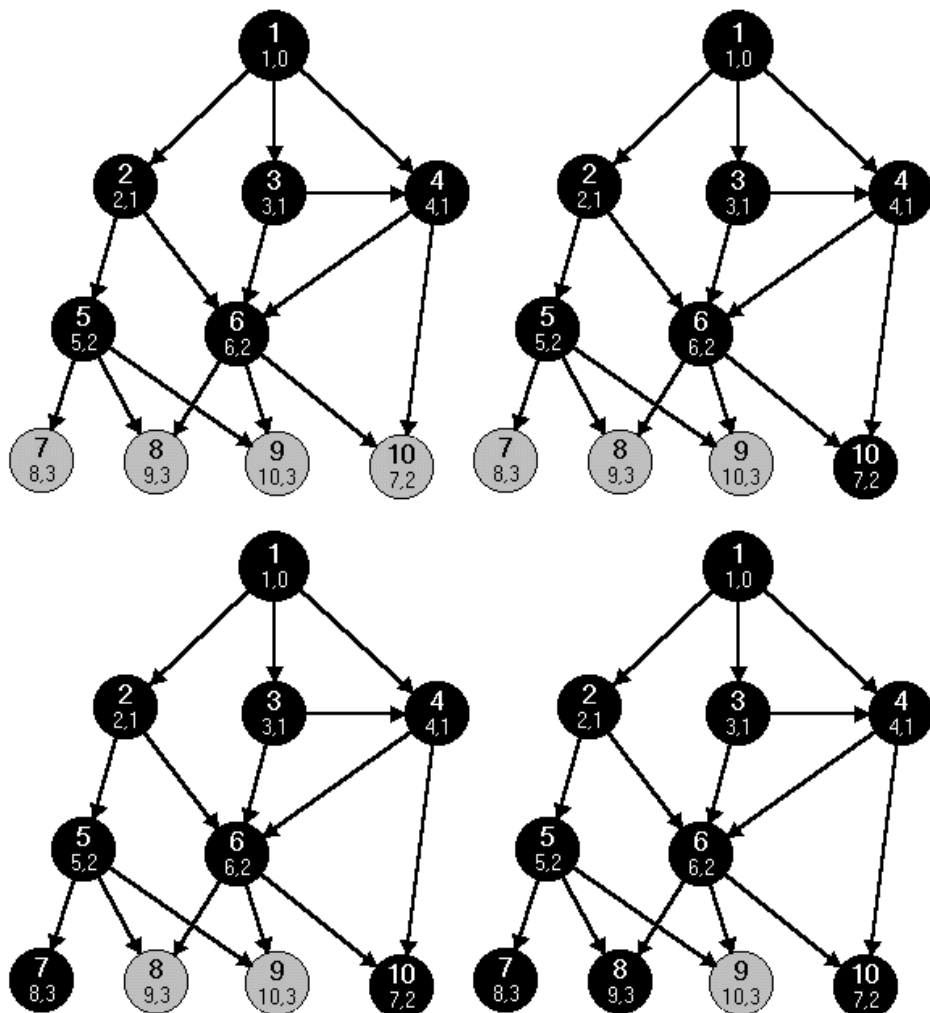
rendezve, azaz mindig azt a szürke csúcsot terjesztjük ki, amelynek az elérési száma a legkisebb, mivel ez a csúcs került be legkorábban a sorba.

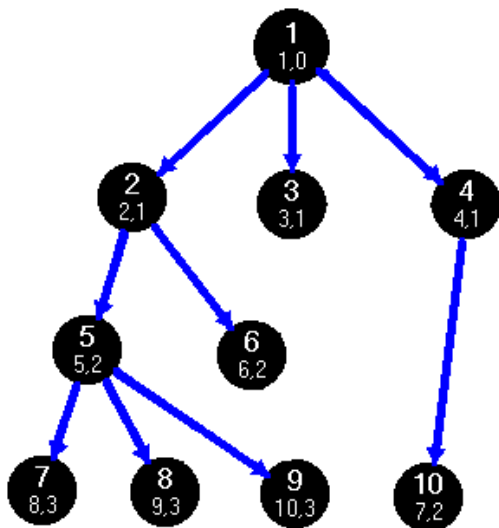


Figyeljük meg, hogy ebben lépésben nem kerül be újabb csúcs a sorba, mivel a 3-as csúcsot terjesztjük ki, de a 3-as minden szomszédját már elértük, azaz nincs fehér színű szomszédja.



A továbbiakban, mivel a sorban lévő csúcsoknak nincsenek szomszédaik, már csak a sorból vesszük ki a csúcsokat és feketére színezzük.





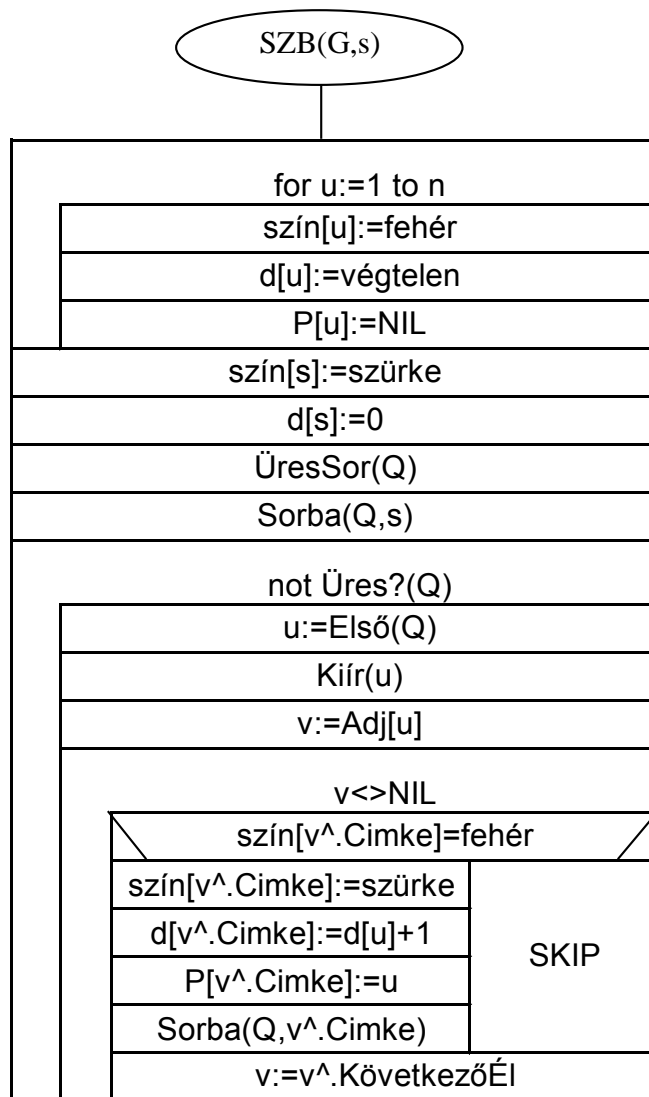
Az utolsó ábrán berajzoltuk a kezdőcsúsból az illető csúcsba vezető, az algoritmus által felderített legrövidebb út mentén az éleket. A csúcsok, és a berajzolt élek alkotta részgráfot jobban megnézve látható, hogy egy kezdőcsúcs gyökerű fát alkot, amely mentén minden csúcs a legrövidebb úton érhető el.

4.2.3 Definíció [1]: $F=(V',E')$ gráf a $G=(V,E)$ gráf szélességi fája, ha V' elemei az s -ből elérhető csúcsok, $E' \subseteq E$ és $\forall v \in V'$ csúcsra pontosan egy egyszerű út vezet s -ből v -be, és ez az út egyike az s -ből v -be vezető legrövidebb G -beli utaknak.

Most nézzük meg az algoritmust az **ábrázolás szintjén:**

- éllistas ábrázolással
- csúcsmátrixos ábrázolással (gyakorló feladatok között)

A csúcsok színét egy $szín[1..n]$ (csúcsokkal indexelt) tömbbe tároljuk. További feladatunk a csúcs távolságának, és a hozzá vezető úton a megelőző csúcsnak (a szélességi fa felrajzolásához) az eltárolása. „Ezt egy $d[1..n]$ és egy $P[1..n]$ tömbben tesszük meg” [2]. Az értékeket akkor ismerjük, amikor elérjük a csúcsot, azaz amikor szürkére színezzük, tehát ekkor fogjuk a tömbbe beírni. Kezdetben legyen minden csúcs végtelen távolságra a kezdőcsúctól, és ha nincs a gráfban $s \rightsquigarrow u$ út, akkor az u távolsága végtelen is marad.



A programot a fenti példán lefuttatva, a következő eredményeket kapjuk: $d[1..10]=[0,1,1,1,2,2,3,3,3,2]$ és $P[1..10]=[NIL,1,1,1,2,2,5,5,5,4]$. Látható, hogy a $P[1..10]$ tömb tartalmából könnyen "előállítható" a szélességi fa, illetve bármely csúcsra kiírható a legrövidebb út (gyakorlaton).

Műveletigény: Az algoritmus az inicializáló lépés során minden csúcsnak beállítja a színét, a d és P tömbbeli értékét. Ez n -el arányos műveletigény: $\Theta(n)$.

Éllezés ábrázolás esetén: minden csúcsot (amibe megy él) **legfeljebb egyszer** teszünk a sorba. Mikor a sorból kivesszük a csúcsot, az éllezésén lévő csúcsokat érjük el. Mivel minden csúcs legfeljebb egyszer kerül a sorba és onnan ki, ezért minden éllezésén legfeljebb egyszer megyünk végig, tehát összességében legfeljebb az összes éllezésén egyszer megyünk végig. Az éllezések együttes hossza e , így a műveletigény $O(e)$ (Ha nem összefüggő a gráf, akkor lehetnek olyan élek, amik mentén nem járunk, ezért nem mondhatunk Θ -t).

$$T(n) = \Theta(n) + O(e) = O(n + e)$$

Csúcsmátrixos ábrázolás esetén: egy csúcs szomszédainak a vizsgálata, a gráf egy n hosszú sorának a végigjárását eredményezi, ezt az összes csúcsra vetítve megkapjuk:

$$T(n) = O(n + n * n) = O(n^2)$$

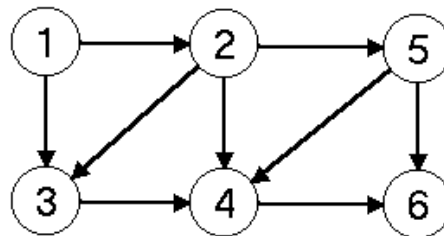
A továbbiakban külön nem hangsúlyozzuk, hogy e -vel arányos műveletigény csúcsmátrixos ábrázolás esetén mindig n^2 -el arányos műveletet jelent.

4.3 Ellenőrző kérdések

1. Adja meg az út hosszának definícióját!
2. Adja meg két csúcs távolságának definícióját!
3. Mekkora lehet két csúcs legnagyobb távolságának!
4. Milyen stratégiát használ a szélességi bejárás?
5. Milyen nevezetes adattípust használ a szélességi bejárás algoritmus?
6. Milyen invariáns tulajdonságot biztosítunk a használt adattípus segítségével?
7. Egy csúcsot hányszor érhetünk el, egyszer vagy többször is?
8. Egy csúcs színe fehér, ha...
9. Egy csúcs színe szürke, ha...
10. Egy csúcs színe fekete, ha...
11. Mi a szélességi fa definíciója?
12. Mi az algoritmus műveletigénye éllistas ábrázolás esetén?
13. Mi az algoritmus műveletigénye csúcsmátrixos ábrázolás esetén?

4.4 Gyakorló feladatok

1. Szemléltesse a szélességi bejárást az alábbi gráfon. Rajzolja le a sor tartalmát, a KÉSZ halmaz tartalmát, a d és a Pi tömb tartalmát lépésenként!



2. Milyen szélességi fát határoz meg az alábbi Pi tömb, melyet egy szélességi bejárás után kaptunk? $Pi = [2, 6, 2, 7, 10, \text{NIL}, 10, 7, 2, 6]$
3. Adjon algoritmust, amely a szélességi bejárás lefuttatása után kiírja egy $v \in V$ csúcsra az egyik legrövidebb $s \rightsquigarrow v$ utat!
4. Adott egy $G = (V, E)$ irányítatlan gráf. Adjon algoritmust, amely eldönti, hogy a páros-e a gráf!
 - a G összefüggő.
 - b G nem biztos, hogy összefüggő.
5. Adott egy $G = (V, E)$ irányítatlan gráf. Adjon algoritmust, amely eldönti, hogy G tartalmaz-e kört!
 - a G összefüggő.

- b G nem biztos, hogy összefüggő.
6. Adott egy $G = (V, E)$ irányított gráf. Adjon algoritmust, amely eldönti, hogy G tartalmaz-e (irányítástól eltekintett) kört!
- a G összefüggő.
- b G nem biztos, hogy összefüggő.
7. Adott egy $G = (V, E)$ irányítatlan gráf. Adjon algoritmust, amely G komponenseit kiszínezi! (Az azonos komponensbe eső csúcsokat azonos, a különböző komponensbe eső csúcsokat különböző színűre.)
8. Adott egy $G = (V, E)$ irányítás nélküli, összefüggő, véges gráf. Adjon $O(n + e)$ futásidőjű algoritmust, amely meghatározza G -nek egy összefüggő, körmentes $G' = (V, E')$ részgráfját (feszítőfáját)!
9. Legyen G egy élsúlyozott gráf, amelynek minden élsúlya természetes szám. A szélességi bejárás algoritmusát átalakítva, adjon algoritmust, amely minden csúcsra megadja, egy $s \in V$ csúcsból az illető csúcsba vezető, legkisebb költségű utat!
10. Adjon algoritmust, amely megoldja az alábbi feladatot! Adott egy $n \times n$ -es sakktabla, egy huszárral (lóval), egy induló mezőről jussunk el egy célmezőre a lehető legkevesebb szabályos lépéssel.
11. Adott egy város tömegközlekedési hálózatának gráfmodellje. A megállóhelyeknek megfeleltettük a gráf csúcsait, egy járat két megálló közé eső szakaszának megfeleltettük a gráf éleit. Az élek egy tulajdonsága legyen a járat azonosítója (pl.: 6-os villamos, 3-as metró stb.). Adjon algoritmust, amely megadja, hogyan tudunk eljutni a gráf egyik csúcsából egy másik csúcsába a legkevesebb „átszállással” (járat váltással)!

4.5 Összefoglalás

- Mélységi stratégia: a kezdőcsúcs egy szomszédját elérjük, majd annak egy szomszédján át folytatva addig megyünk, amíg van még el nem ért szomszéd, majd egy lépést visszalépve az előző csúcsra, annak egy másik még el nem ért csúcsán folytatjuk.
- Szélességi stratégia: először elérjük a kezdőcsúcs közvetlen szomszédait, majd azok szomszédait, mintegy szélesedő „koncentrikus kör”, úgy járjuk be a gráfot.
- A gráf egy útjának a hossza: az út mentén érintett élek száma.
- Két csúcs távolsága: a két csúcs közötti legrövidebb út hossza.
- A szélességi bejárás meghatározza a gráf csúcsainak a kezdőcsúcsból vett távolságát. Miközben felépíti a szélességi fát (kezdőcsúcs gyökerű fa, amely a csúcsokhoz vezető legrövidebb utakat tartalmazza)
- A bejárás elve: egy sort használunk, kezdetbe berakjuk a kezdőcsúcsot. A bejárást addig folytatjuk, míg a sor ki nem ürül. Minden lépésben kivesszünk a sorból egy csúcsot és a még el nem ért (színezéssel jelölhető) szomszédait berakjuk a sorba.

- Műveletigény:

- éllistas ábrázolás esetén $T(n) = \Theta(n) + O(e) = O(n + e)$
- csúcsmátrixos ábrázolás esetén: $T(n) = O(n + n * n) = O(n^2)$

5. Legrövidebb utak egy forrásból

Probléma: Adott egy $G=(V,E)$ élsúlyozott, véges gráf és egy $s \in V$ csúcs ú.n. forrás. Szeretnénk meghatározni, $\forall v \in V$ csúcsra, s -ből v -be vezető **legkisebb költségű** utat.

A gráfalgoritmusok bevezetőjében már felvetettünk egy problémát "minimális költségű út keresése" címen, amely egy csúcspár közötti **legkisebb költségű utat** keresi. Ezt a problémát, tekinthetjük a fent említett **probléma részeként**, azaz a fenti problémát megoldó algoritmus megoldja ezt a problémát is. „Érdekes, hogy **nem ismert** aszimptotikusan hatékonyabb algoritmus s -ből egyetlen v csúcsba vezető legkisebb költségű út megtalálására, mint s -ből minden csúcsba menő ilyen út megtalálásának leggyorsabb algoritmusá” [2]. Ebben a fejezetben **élsúlyozott** gráfokkal fogunk foglalkozni. A **legrövidebb úton**, a szakirodalomban elterjedt módon, a **szélességi keresésnél** tanultaktól eltérően, a **legkisebb költségű utat** fogjuk érteni.

5.0.1 Definíció [1]: Legyen $G=(V,E)$ élsúlyozott, irányított vagy irányítás nélküli gráf $c : E \rightarrow \mathbf{R}$ súlyfüggvénnyel. A $p = \langle v_0, \dots, v_k \rangle$ **út hossza** (súlya, költsége) az utat alkotó élek súlyainak az összege, azaz

$$d(p) = \begin{cases} 0 & , \text{ ha } k = 0 \\ \sum_{i=1}^k c(v_{i-1}, v_i) & , \text{ különben} \end{cases}$$

5.0.2 Definíció [1]: Az u -ból a v -be ($u, v \in V$) vezető **legrövidebb** (legkisebb súlyú, költségű)

út súlya legyen $\delta(u, v) = \begin{cases} \min\{d(u \rightsquigarrow v)\} & , \text{ ha } \exists u \rightsquigarrow v \text{ út a gráfban} \\ \infty & , \text{ különben} \end{cases}$

Az u csúcsból a v -be vezető **legrövidebb úton** a $\delta(u, v)$ súlyú, u -ból v -be vezető utak egyikét értjük.

Egy exponenciális műveletigényű megoldás:

Szeretnénk eljutni a legrövidebb úton Budapestről Szegedre. Rendelkezünk egy autós térképpel, amelyből kiolvashatjuk az útelágazások közti távolságot. A probléma egyik megoldása, ha előállítjuk az **összes egyszerű** (mivel kör esetén végtelen sok út létezik) Budapestről Szegedre menő utat, és egy **minimumkereséssel** kikeressük közülük a legrövidebbet. Előre látható, hogy sok olyan út lesz, amely biztosan nem jöhet számításba (pl.: Sopronon átmenő utak). Ez egy lassú, exponenciális idejű algoritmus. A továbbiakban látni fogunk a probléma megoldására **polinomiális** idejű algoritmust is.

Legrövidebb utak ábrázolása [1]:

A szélességi keresésnél már találkoztunk egy hasonló feladattal, az ottani legrövidebb (legkisebb élszámú) utak nyilvántartásával. Most is ugyanúgy fogunk eljárni, egy $P[1..n]$ tömbben tartjuk nyilván minden csúcsnak, az algoritmus által talált egyik legrövidebb úton, a **megelőzőjét**. Itt is a szélességi fához hasonlóan definiálhatjuk a legrövidebb-utak fát.

5.0.3 Definíció [1]: $F=(V',E')$ gráf a $G=(V,E)$ gráf **legrövidebb-utak fája**, ha V' elemei az s -ből elérhető csúcsok, $E' \subseteq E$ és $\forall v \in V'$ csúcsra pontosan egy egyszerű út vezet s -ből v -be, és ez az út egyike az s -ből v -be vezető legrövidebb G -beli utaknak.

5.1 Dijkstra algoritmus

Feladat: Adott egy $G=(V,E)$ élsúlyozott, irányított vagy irányítás nélküli, **negatív élsúlyokat nem tartalmazó**, véges gráf. Továbbá adott egy $s \in V$ forrás (kezdőcsúcs). Határozzuk meg, $\forall v \in V$ csúcsra, s -ből v -be vezető **legrövidebb utat és annak hosszát!**

Nézzünk az algoritmus helyességének a belátásához szükséges néhány állítást:

5.1.1. Lemma [1]: A $v_0 \in V$ csúcsból a $v_k \in V$ csúcsba vezető, bármely legrövidebb $\langle v_0, \dots, v_{k-1}, v_k \rangle$ ($k > 0$) út olyan, hogy a $\langle v_0, \dots, v_{k-1} \rangle$ út is egyike a v_0 -ból a v_{k-1} -be menő legrövidebb utaknak.

Bizonyítás: Az úthossz definíciójából tudjuk, hogy $d(\langle v_0, \dots, v_{k-1}, v_k \rangle) = d(\langle v_0, \dots, v_{k-1} \rangle) + c(v_{k-1}, v_k)$. Indirekt tegyük fel, hogy létezik $q = \langle v_0, \dots, v_{k-1} \rangle$ útnál rövidebb p út v_{k-1} -be. Ekkor ezen az úton eljutva v_{k-1} -be az úthossz: $d(p) + c(v_{k-1}, v_k) < d(q) + c(v_{k-1}, v_k)$, mivel $d(p) < d(q)$. Tehát találtunk a legrövidebb útnál rövidebb utat, ami ellentmondás.

Az állításból következik, elegendő a legrövidebb úton csak a megelőző csúcsot eltárolni. A lemma könnyen általánosítható "a legrövidebb út részútja is legrövidebb út" állításra.

5.1.2. Lemma: $\forall \langle v_0, \dots, v_k \rangle$ úton, a $d(v_0 \rightsquigarrow v_0), d(v_0 \rightsquigarrow v_1), \dots, d(v_0 \rightsquigarrow v_k)$ részutak költségei **monoton növvő sorozatot** alkotnak.

Bizonyítás: lásd Analízis: Nem negatív tagú végtelen sorok.

Ebben a lemmában használjuk ki, hogy **nincsenek negatív élsúlyok**.

Az algoritmus elve [2]:

Minden lépésben tartsuk nyilván az **összes csúcsra**, a forrástól az illető csúcsba vezető, **eddig talált** legrövidebb utat (a már megismert módon a $d[1..n]$ tömbben a távolságot, és $P[1..n]$ tömbben a megelőző csúcsot).

- 1) Kezdetben a távolság legyen a kezdőcsúcsra 0, a többi csúcsra ∞ .
- 2) Minden lépésben a nem KÉSZ csúcsok közül tekintsük az egyik **legkisebb távolságú** (d_{\min}) csúcsot:
 - a) Azt mondhatjuk, hogy ez a $v \in V$ csúcs már KÉSZ, azaz **ismert** a hozzá vezető legrövidebb út.
 - b) A v -t **terjesszük ki**, azaz v csúcs szomszédaira számítsuk ki a (már ismert) v -be vezető, és onnan egy kimenő éllel meghosszabbított út hosszát. Amennyiben ez jobb (kisebb), mint az illető szomszédba eddig talált legrövidebb út, akkor innentől kezdve ezt az utat tekintsük, az adott szomszédba vezető, eddig talált legrövidebb útnak. Ezt az eljárást szokás **közelítésnek** is nevezni.

5.1.3. Lemma: Az egyes lépések után, $\forall u \in V \setminus \text{KÉSZ}$ csúcs esetén, ha $\exists s \rightsquigarrow u$ út a gráfban, akkor az u csúcshoz vezető legrövidebb úton $\exists q \in \text{KÉSZ}$ csúcs.

Bizonyítás: Az $s \in \text{KÉSZ}$ biztosan teljesül.

5.1.4. Lemma: Minden lépésben, $\forall u \in V \setminus \text{KÉSZ}$ csúcsra: $d_{\min} \leq \delta(s, u)$, azaz s -ből u -ba vezető utak távolsága nem csökkenhet a jelenlegi minimum alá.

Bizonyítás: Indirekt tegyük fel, hogy $\exists u \in V \setminus \text{KÉSZ}$, amelyre az eddig talált legrövidebb p út $d(p) \geq d_{\min}$, de a legrövidebb $p^* = s \rightsquigarrow u$ útra $d_{\min} > d(p^*)$. Tudjuk, hogy van KÉSZ csúcsa a p^* útnak (2. lépéstől kezdve, 5.1.3. lemma). Legyenek $q \in \text{KÉSZ}$ és $w \in \text{Szomszéd}(q) \setminus \text{KÉSZ}$ egymást követő csúcsai p^* -nak (biztos létezik w , mivel $u \in V \setminus \text{KÉSZ}$, legfeljebb $w=u$). Mivel $q \in \text{KÉSZ}$, így már ismert q -ba menő egyik legrövidebb út, ami része az u -ba menő egyik legrövidebb útnak (5.1.1. lemma következménye). Legyen u -ba menő egyik legrövidebb útnak w -ig tartó részútja $p^*_w = s \rightsquigarrow q \rightarrow w$, aminek a hossza már ismert, mivel q -t már KÉSZ-nek választottuk, ezért p^*_q ismert, továbbá q -t kiterjesztettük (lásd 2/b. pontban), így p^*_w is ismert. Azt is tudjuk, hogy $d_{\min} \leq d(p^*_w)$, mivel $w \notin \text{KÉSZ}$. A 3.1.2. monotonitásról szóló lemma miatt, $d(p^*_w) \leq d(p^*) \Rightarrow d_{\min} \leq d(p^*_w) \leq d(p^*)$ ami ellentmond az indirekt feltevésnek.

Az első lépésben triviálisan teljesül az állítás, mivel $d_{\min}=0$ és nincs negatív élsúly.

5.1.5. Lemma: Az egyes lépésekben az algoritmus által KÉSZ-nek kiválasztott $v \in V$ csúcsra valóban ismert az egyik legrövidebb $s \rightsquigarrow v$ út.

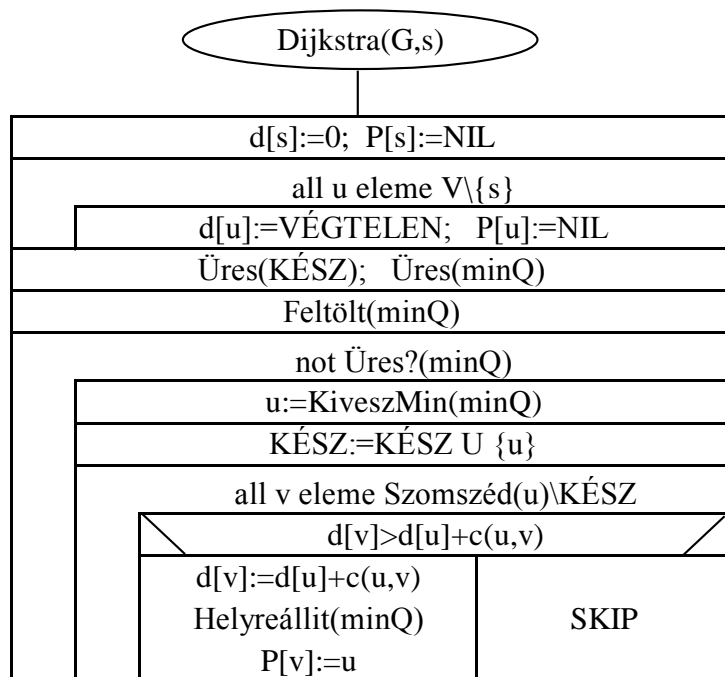
Bizonyítás: Legyen $p = s \rightsquigarrow v$ a jelenleg ismert legrövidebb út (ami lehet ∞ távolságú is), amelyre tudjuk $d(p) = d_{\min}$, és indirekt tegyük fel, hogy $\exists p^* = s \rightsquigarrow v$ út, amelyre $d(p^*) < d(p)$. Tudjuk, hogy $\exists q \in \text{KÉSZ}$ és $w \in \text{Szomszéd}(q) \setminus \text{KÉSZ}$ csúcs a p^* úton. A 5.1.4. lemmában látott módon levezethető, hogy $d(p) = d_{\min} \leq d(p^*_w) \leq d(p^*)$ ami ellentmond az indirekt feltevésnek. Tehát rövidebb $s \rightsquigarrow v$ utat a későbbiekben sem találhatunk.

5.1.6. Tétel: A fenti algoritmus, **negatív élsúlyokat nem tartalmazó** $G=(V,E)$ véges gráf esetén, $s \in V$ forrás (kezdőcsúcs) és $\forall v \in V$ csúcsra, meghatározza s -ből v -be vezető **legrövidebb utat és annak hosszát**.

Bizonyítás: Az algoritmus minden lépésben KÉSZ-nek választ egy csúcsot (5.1.5. lemma). Mivel véges sok csúcsa van a gráfnak, az algoritmus véges időn belül terminál, és $\forall v \in V$ csúcs KÉSZ-en van, azaz ismert a legrövidebb $s \rightsquigarrow v$ út.

Az algoritmus **ADT szintű** leírása [5]:

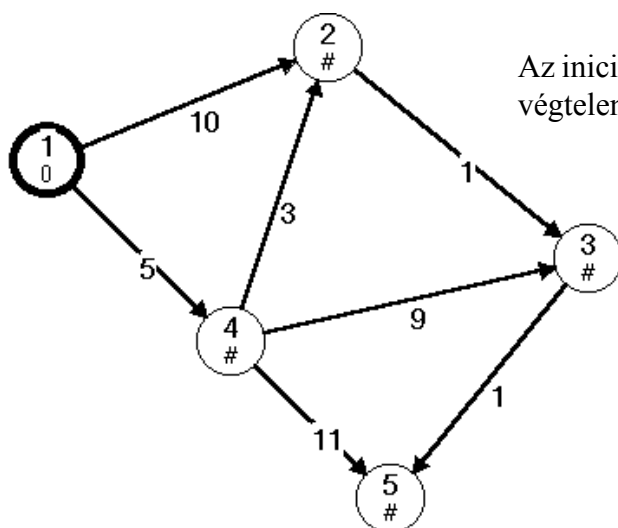
A $d[1..n]$ és $P[1..n]$ tömböket, a korábban ismertetett módon, a távolság és a megelőző csúcs nyilvántartására használjuk. A KÉSZ halmazba rakjuk azokat a csúcsokat, amelyekhez **már ismerjük** az egyik legrövidebb utat. Ezen kívül, használunk egy **minimumválasztó elsőbbségi (prioritásos) sort** (minQ), amelyben a csúcsokat tároljuk a már felfedezett, legrövidebb $d(s \rightsquigarrow u)$ távolsággal, mint **kulcs** értékkel.



Műveletigény	
rendezettlen tömb	kupac
1	
"(n-1)-szer"	
n-1	
1	
0	n
"n-szer"	
n*n	n*logn
n*1	
"e-szer"	
e*1	e*logn

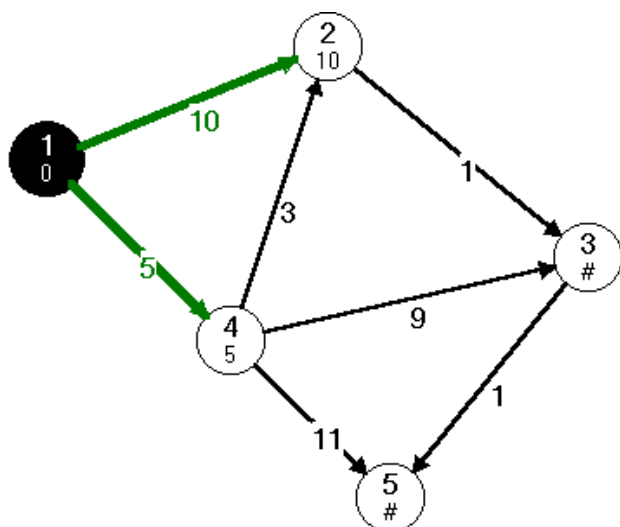
Most nézzük meg egy példán az algoritmus működését **ADS szinten**:

A következő ábra-sorozaton megfigyelhető **Dijkstra algoritmusának működése** lépésenként. A KÉSZ halmazhoz való tartozást színezéssel valósítjuk meg. Legyenek a nem KÉSZ csúcsok fehérek, a KÉSZ csúcsok pedig fekete színűek. A csúcsokra a címkén kívül, felírtuk az eddig talált **legrövidebb út** hosszát is (d tömbbeli értékeket). A **végtelen nagy távolságot** jelöljük '#' jellel. A forrás legyen az 1-es címkéjű csúcs.



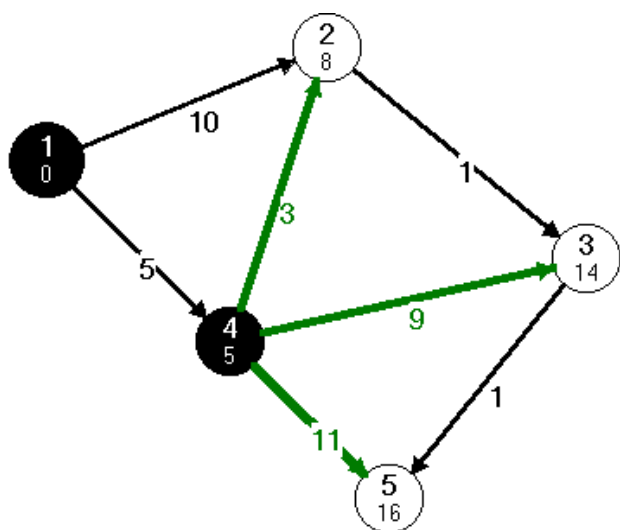
Az inicializáló lépés után a kezdőcsúcs 0, a többi csúcs végtelen súllyal szerepel az elsőbbségi sorban.

inicializálás után



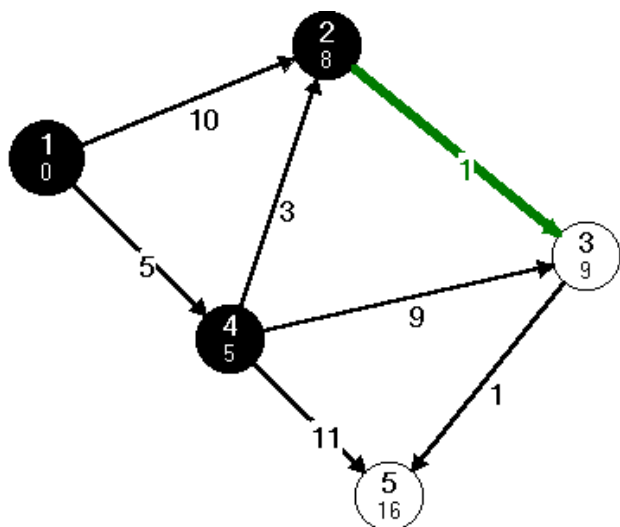
1. lépés

Az első lépésben kivesszük a prioritásos sorból az 1-es csúcsot (mivel az ő prioritása a legkisebb). Az 1-es csúcshoz már ki van számítva a legrövidebb út, tehát ez a csúcs már elkészült, színezzük feketére. Kiterjesztjük az 1-et, azaz a szomszédaira kiszámítjuk az 1-esből kimenő éllel meghosszabbított utat. Ha ez javító él, azaz az 1-esen átmenő út rövidebb, mint az adott szomszédba eddig talált legrövidebb út, akkor a szomszédban ezt feljegyezzük (d és P tömbbe). Az ábrán kiemeltük a javító éleket.



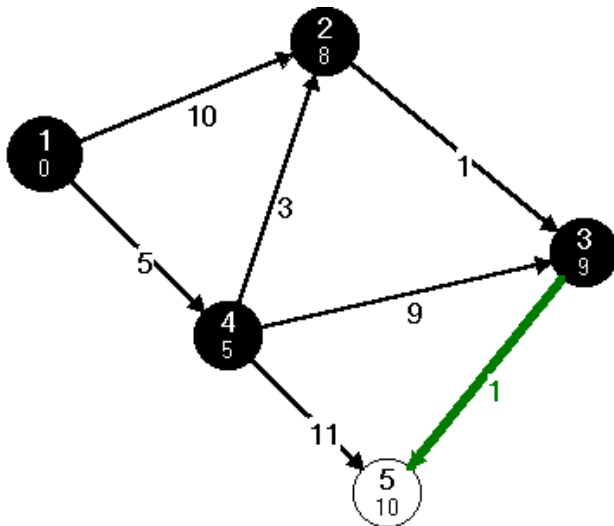
2. lépés

Megfigyelhető, hogy a 2-es csúcsba már korábban is találtunk 10 hosszú utat $\langle 1, 2 \rangle$, de a második lépésben, a 4-es csúcs kiterjesztésekor, találunk, a 4-es csúcson átmenő rövidebb 8 hosszú utat.

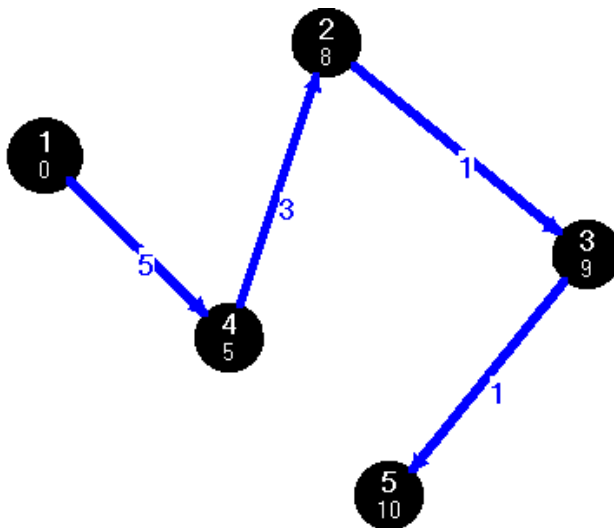


3. lépés

A 2-es csúcs kiterjesztésekor a 3-as csúcsba találtunk egy rövidebb utat.



4. Lépés



5. lépés után kialakult fa

A negyedik lépésben még találunk rövidebb utat az 5-ös csúcsba, majd az utolsó lépésben kivesszük a prioritásos sorból az 5-ös címkéjű csúcsot is. Az utolsó ábrán berajzoltuk a legrövidebb utak fáját alkotó éleket.

Az algoritmus megvalósítása az **ábrázolás szintjén [1]**:

Vizsgáljuk meg a prioritásos sor(*minQ*) megvalósításának két, természetes módon adódó lehetőségét.

- 1) A prioritásos sort valósítsuk meg **rendezetlen tömbbel**, azaz a prioritásos sor legyen maga a $d[1..n]$ tömb. Ekkor a minimum kiválasztására egy **feltételes minimum keresést** kell alkalmazni, amelynek a műveletigénye $\Theta(n)$. A *Feltölt(minQ)* és a *Helyreállít(minQ)* absztrakt műveletek megvalósítása pedig egy SKIP-pel történik.
- 2) **Kupac adatszerkezet** használatával is reprezentálhatjuk a prioritásos sort. Ekkor a *Feltölt(minQ)* eljárás, egy kezdeti kupacot épít, amelynek a műveletigénye **lineáris** (lásd Heap-Sort). Azonban most a $d[1..n]$ tömb változása esetén a kupacot is karban kell tartani, mivel a kulcs érték változik. Ezt a *Helyreállít(minQ)* eljárás teszi meg, amely a csúcsot a gyökér felé "szivárogtatja" fel, ha szükséges (mivel a kulcs értékek csak csökkenhetnek). Ennek a műveletigénye $\log n$ -es.

Megjegyzés: Nem szükséges kezdeti kupacot építeni, felesleges a kupacba rakni a végtelen távolságú elemeket. Kezdetben csak a kezdőcsúcs legyen a kupacban, majd amikor először elérünk egy csúcsot és a távolsága már nem végtelen, elég akkor berakni a kupacba.

Tehát a prioritásos sor fenti két megvalósítása esetén, a következőképpen alakul az **algoritmus műveletigénye [1]:**

A struktogramm mellett feltüntettük az egyes műveletek költségét a két ábrázolás estén. A belső ciklust célszerű globálisan kezelni, ekkor mondható, hogy összesen legfeljebb annyiszor fut le, ahány éle van a gráfnak.

1) Tehát **rendezetlen tömb** esetén:

$$T(n) = O(1 + n - 1 + 1 + 0 + n^2 + n + e) = O(n^2 + e) = O(n^2)$$

2) **Kupac** esetén: $T(n) = O(1 + n - 1 + 1 + n + n * \log n + n + e * \log n) = O((n + e) * \log n)$

Következmény az ábrázolásra [5]:

Rendezetlen tömbbel való ábrázolás műveletigénye **csak a csúcsok számától** függ, míg a kupacos ábrázolás műveletigénye, az **élek számának is a függvénye**. **Sűrű gráfnak** nevezzük az olyan gráfokat, amelyre $e \approx n^2$, **ritka gráfoknak** pedig, amelyre $e \approx n$ (vagy "kevesebb"). Tehát a kupacos ábrázolás műveletigénye ritka gráf esetén $O(n * \log n)$, míg sűrű gráf esetén $O(n^2 \log n)$. Az az érdekes helyzet adódott, hogy a **gráf sűrűsége befolyásolja** milyen **ábrázolást** érdemes választani. A kupac, csak a ritka gráfok esetén hatékonyabb, míg sűrű gráfok esetén a rendezetlen tömbbel való reprezentáció az olcsóbb.

Tehát a reprezentáció szintjén:

Sűrű gráf esetén: csúcsmátrix + rendezetlen tömb.

Ritka gráf esetén: éllista + kupac.

A **mohó algoritmus** mindig az adott lépésben optimálisnak látszó döntést hozza, vagyis a **lokális optimumot** választja abban a reményben, hogy ez **globális optimumhoz** fog majd vezetni. Dijkstra algoritmus is mohó stratégiát követ, amikor minden lépésben KÉSZ-nek választ egy csúcsot. Mivel a legrövidebb út részútja is legrövidebb út, így a lokális optimumok választásával elérhetjük a globális optimumot. [2]

Most vizsgáljuk meg a **szélességi keresés és a Dijkstra algoritmus kapcsolatát**.

Mindkét algoritmusnál, egy kezdőcsúcsból kiinduló legrövidebb utakat állítunk elő, csak a Dijkstra algoritmusnál az **utak hosszának fogalmát általánosítjuk**. Legyen **minden él súlya egységnyi**, ekkor a Dijkstra algoritmus egy szélességi keresést hajt végre. Tehát mondhatjuk, hogy a szélességi keresés **speciális esete** a Dijkstra algoritmusnak, ahol a prioritásos sor helyett, egy egyszerű sort használunk, amellyel javítunk a műveletigényen. Azt kell belátni, hogy a sor használatával is mindig a legkisebb távolságú csúcsot választjuk KÉSZ-nek, de ez következik a szélességi keresésnél megvizsgált invariáns tulajdonságból.

5.2 Bellman-Ford algoritmus

Feladat: Adott egy $G=(V,E)$ élsúlyozott, irányított vagy irányítás nélküli, **negatív összköltségű irányított kört nem tartalmazó** véges gráf, továbbá egy $s \in V$ forrás (kezdőcsúcs). Határozzuk meg, $\forall v \in V$ csúcsra, s -ből v -be vezető **legrövidebb utat és annak hosszát!**

Megjegyzések:

- Kezdőcsúcsból elérhető negatív összköltségű kör esetén, nem léteznek legkisebb költségű utak, mivel az illető körön tetszőlegesen sokszor végig menve az utak költsége mindig csökkenthető (lásd gyakorló feladatok között *arbitrázs* feladat).
- Irányítatlan gráf esetén, egy (u,v) negatív súlyú irányítatlan élen oda-vissza haladva az út költsége végtelenségig csökkenthető, azaz úgy viselkedik, mint egy negatív összköltségű kör. Tekintsük negatív összköltségű, 2 élből álló irányított körnek, amely egybe vág az ábrázolás szintjén megvalósított irányítatlan gráffal, ahol egy irányítatlan élt, egy oda-vissza irányított élpárral valósítunk meg. Tehát irányítatlan gráf esetén a megszorításunk, hogy a gráf ne tartalmazzon negatív súlyú irányítatlan élt, amely negatív összköltségű irányított körnek tekinthető.

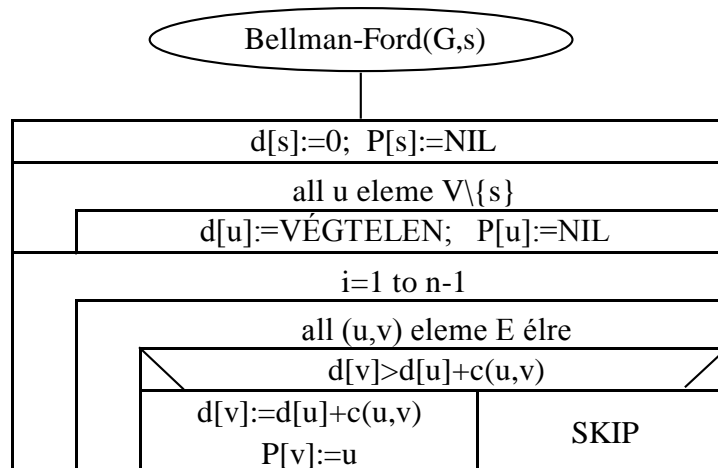
Az algoritmus elve [1]:

„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 élszáma, **legfeljebb $n-1$** lehet.” [2]

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 $p^* = s \rightsquigarrow v$ legrövidebb utat. Minden menetben a p^* minden élen végzünk közelítést. Legyen $x \rightarrow y$ él része p^* -nak. Miután p^* x -ig tartó részútja p^*_x ismerté válik, a következő menetben a p^*_y is ismert lesz, mivel az (x,y) 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 ismerté. Mivel a leghosszabb egyszerű út $n-1$ élszámú, ezért szükséges lehet az $n-1$ menet.

Az algoritmus ADT szinten [5]:



5.2.1. Tétel: Adott egy $G=(V,E)$ élsúlyozott, irányított vagy irányítás nélküli, **negatív összköltségű irányított kört nem tartalmazó** véges gráf, továbbá egy $s \in V$ forrás (kezdőcsúcs). Ekkor a Bellman-Ford algoritmus meghatározza $\forall v \in V$ csúcsra **legrövidebb utat és annak hosszát**.

Bizonyítás:

1) Legyen $p^* = \langle v_0, \dots, v_k \rangle$ egy s -ből v -be vezető, egyszerű legrövidebb út, ahol $s=v_0$ és $v=v_k$.

Teljes indukcióval belátjuk, hogy az i -dik menet után már ismert $\langle v_0, \dots, v_i \rangle$ legrövidebb részút, azaz $d[v_i] = \delta(s, v_i)$ és $P[v_i] = v_{i-1}$, és ez már nem romlik el később sem.

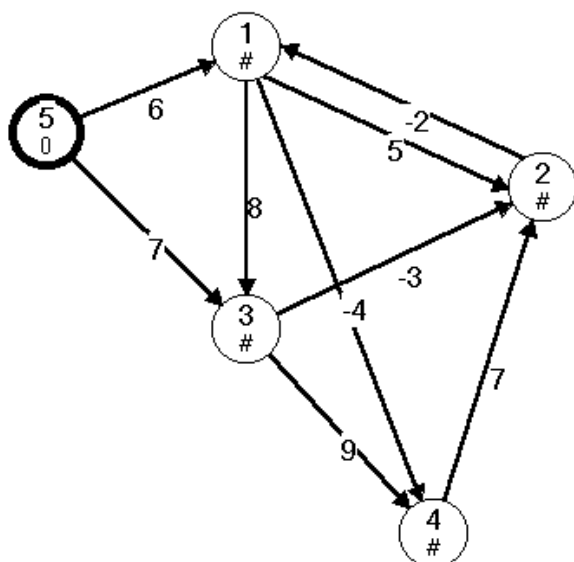
a) Kezdetben az inicializáló lépés után $d[s] = d[v_0] = \delta(s, v_0) = 0$. Ez fennmarad, különben létezne egy olyan u csúcs, hogy $s \rightsquigarrow u \rightarrow s$ és $d(s \rightsquigarrow u) + c(u, s) < 0$, ami azt jelenti, hogy találtunk egy negatív kört.

b) $i-1 \rightarrow i$: Tegyük fel, hogy ismert p^* -nak $s \rightsquigarrow v_{i-1}$ részútja. Az i -dik menetben a (v_{i-1}, v_i) éllal is végzünk közelítést (feljegyezzük $d[v_i] = \delta(s, v_i)$ és $P[v_i] = v_{i-1}$), és ez csak akkor nem történik meg ((v_{i-1}, v_i) nem javító él), ha $\delta(s, v_i)$ már ismert, azaz a v_i -be menő egyik legrövidebb utat már korábban megtaláltuk. A későbbiek során ez már nem változhat, mivel ha ez változna, az azt jelentené, hogy létezik a legrövidebb útnál rövidebb út, mivel p^* is legrövidebb út és annak bármely részútja, így p^* -nak $s \rightsquigarrow v_i$ részútja is legrövidebb út.

2) Mivel a legnagyobb élszámú, egyszerű, legrövidebb út élszáma is legfeljebb $n-1$, ezért a fenti indukciós állításból következik az algoritmus helyessége.

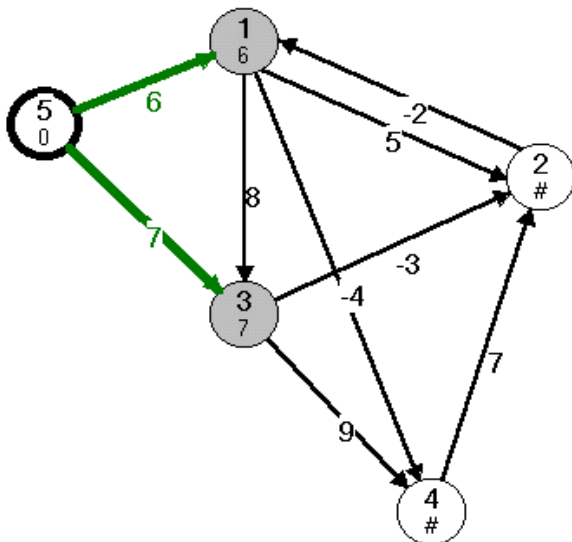
Most pedig nézzük meg egy példán, **ADS szinten** az algoritmus működését:

Tegyük fel, hogy az élek feldolgozási sorrendje a csúcsok címkéje szerint rendezett.



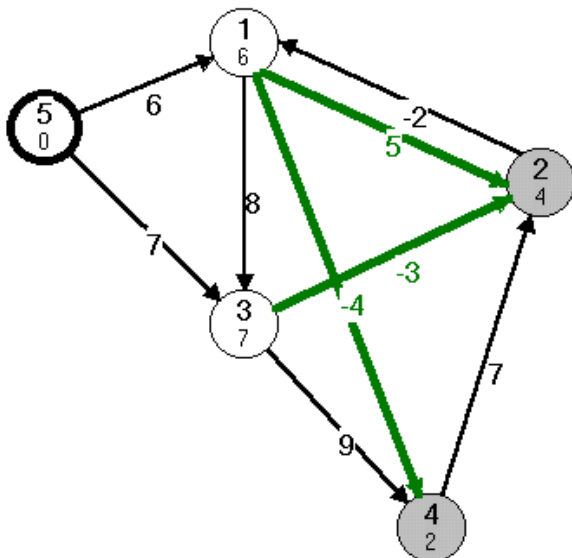
inicializálás után

Az inicializáló lépés során beállítjuk a $d[1..n]$ és $P[1..n]$ tömb értékeit. A végtelen értéket most is '#' jellel jelöljük.



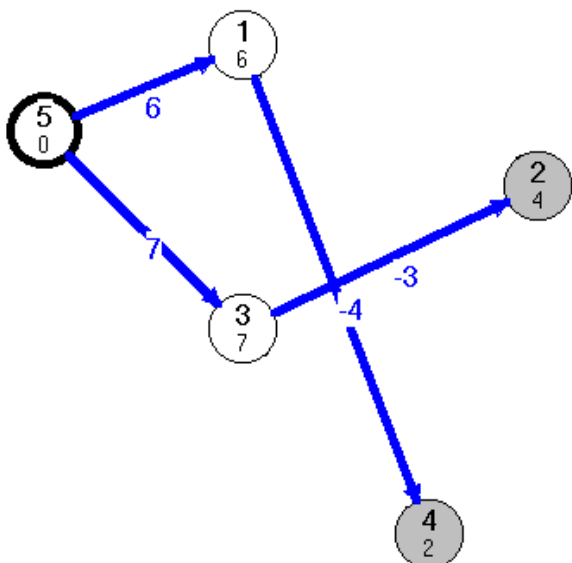
Az első 7 él ((1,2),(1,3),..., (4,2)) közelítésénél nem történik változás, mivel végtelen értékek növelésénél szintén végtelent kapunk, ami nem javít. Csak két javító élt találunk. Most állíthatjuk, hogy minden csúcshoz megtaláltuk az s-ből hozzá vezető, minimális költségű, 1 élszámú utat.

1. lépés



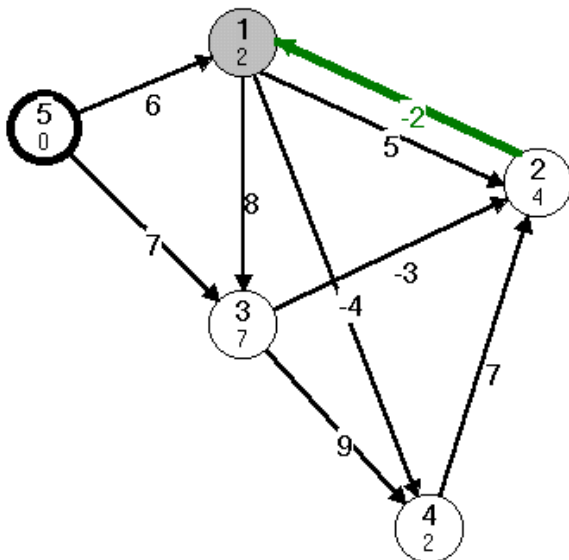
Az ábrán látható, mely csúcsokhoz találtunk javító élt.

2. lépés



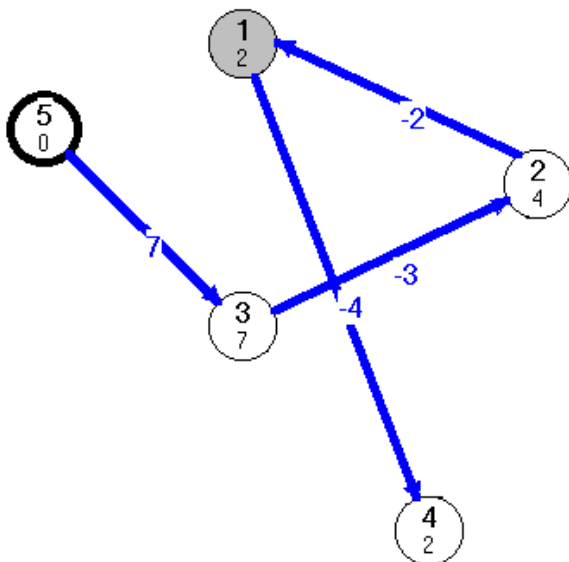
2. lépés fája

Minden csúcshoz meghatároztuk a legkisebb költségű, 1 vagy 2 élszámú utat. Az ábrán látható, az egyes csúcsokba vezető, 1 vagy 2 élszámú legrövidebb utakból kialakult fa. Ez a fa változhat, mivel lehet, hogy egy csúcsba el lehet jutni nagyobb élszámú olcsóbb úton is.



Az 1-be olcsóbb 3 élszámú utat találtunk.

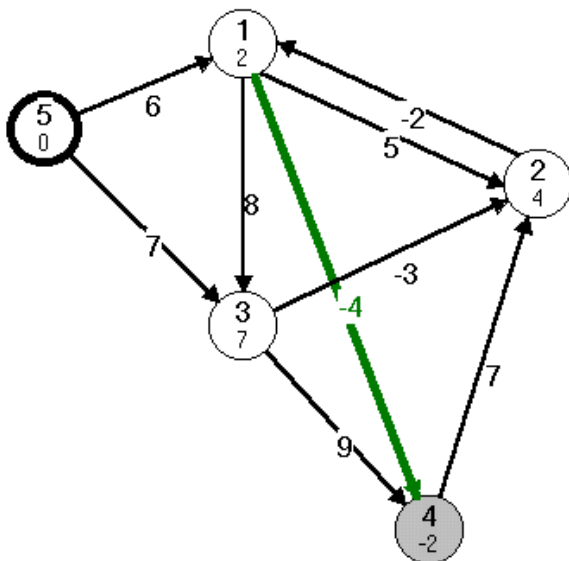
3. lépés



3. lépés fája

A fa változott mivel az 1-be már nem 1 élszámú, hanem 3 élszámú, de rövidebb úton juthatunk el a kezdőcsúcsból.

A 4 megelőzője a korábban talált 1-es, csak most nem 2 élhosszú úttal, hanem 4 élhosszúval $\langle 5,3,2,1,4 \rangle$. Mivel az $(1,4)$ élt korábban dolgoztuk fel, mint $(2,1)$ élt, így a 4-es csúcsnál bejegyzett költség nem konzisztens a fával.



4. lépés

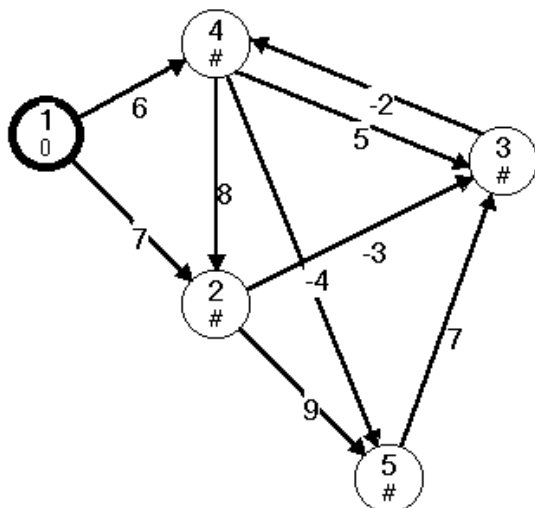
A fa már nem változik, csak 4-es csúcsnál bejegyzett költség veszi fel a helyes értéket.

Műveletigény [1]: Mivel $n-1$ lépés van, és minden lépés során, minden élre végrehajtunk egy közelítést, ezért $T(n) = \Theta((n-1) * e)$.

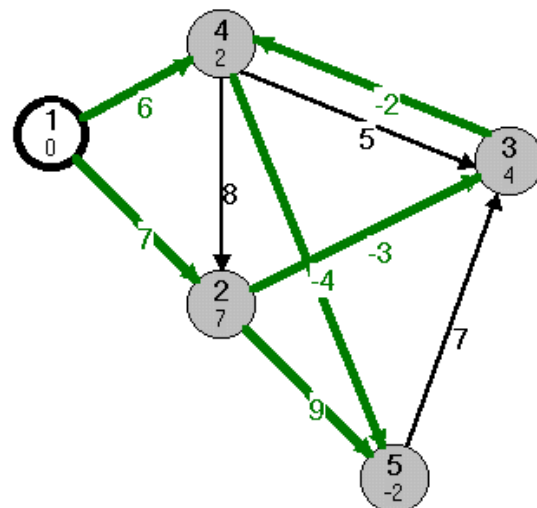
5.2.2. Állítás: (Gyorsítási lehetőség) „ha egy lépés során **nem volt változás** (a közelítések során), akkor **készen vagyunk**, tehát megállhatunk (a buborék rendezésnél már láttunk egy hasonló gyorsítási lehetőséget).” [5]

U.i.: Indirekt tegyük fel, hogy létezik az algoritmus által megadott olyan legrövidebb $p^* = s \rightsquigarrow x \rightarrow y \rightsquigarrow v$ út, hogy az i -dik lépésben az $s \rightsquigarrow x$ részutat már ismerjük, de $x \rightarrow y$ él még nem része a fának, vagy $d[y]$ értéke nem konzisztens, tehát mindkét esetben $d[y] > \delta(s, y)$, továbbá az i -dik lépésben nem történik változás. Azonban $d[x] = \delta(s, x)$ és $\delta(s, y) = \delta(s, x) + c(x, y)$, további az i -dik lépésben az (x, y) él közelítése során $d[y] \leq \delta(s, x) + c(x, y) = \delta(s, y)$, ami ellentmond az indirekt feltevésnek.

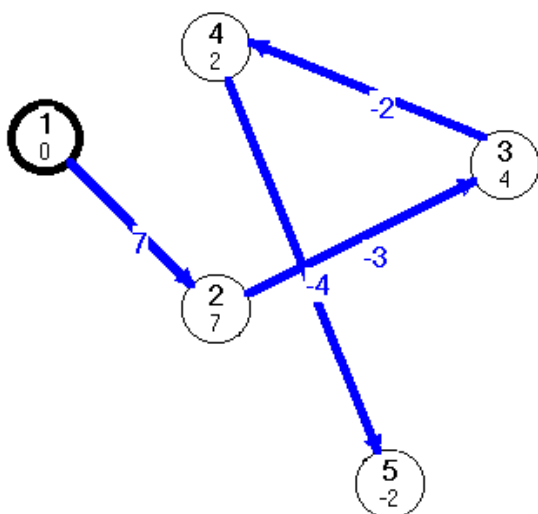
Megjegyzés: Ha a fenti gyorsítási lehetőséget beépítjük az algoritmusba, akkor az **élek feldolgozási sorrendje** befolyásolja az iterációk számát. A fenti példához képest, most átcímkeztük a csúcsokat (az élek feldolgozási sorrendje legyen továbbra is csúcsok címkéje szerint rendezett), így **1 lépésben megkaphatjuk** a megoldást:



inicializálás után



1. lépés



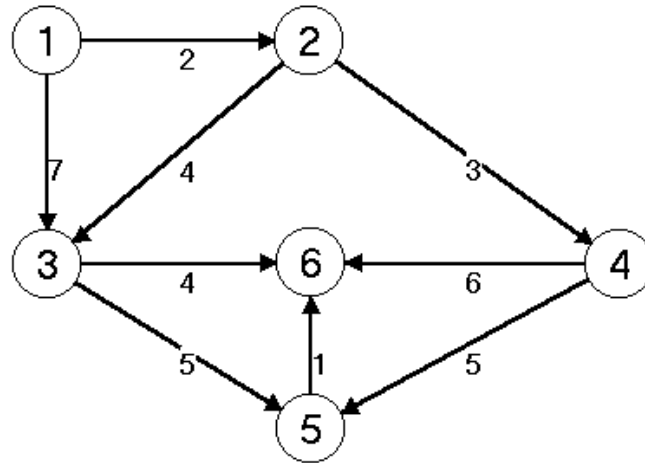
1. lépés után már kialakult a végleges fa

5.3 Ellenőrző kérdések

1. Adja meg az út hosszának definícióját!
2. Adja meg a legrövidebb út definícióját!
3. Mi a legrövidebb utak fájának definíciója?
4. Helyes eredményt ad-e a Dijkstra algoritmus negatív élsúlyokat tartalmazó gráf esetén? Válaszát mutassa be egy egyszerű példán!
5. Mit nevezünk „közelítésnek” az algoritmusban?
6. Milyen nevezetes adattípust használunk a Dijkstra algoritmusban?
7. Az adattípusnak milyen megvalósításait vizsgáltuk?
8. Adja meg a Dijkstra algoritmus műveletigényét különböző ábrázolások esetén!
9. Mit értünk mohó algoritmus alatt?
10. Mivel a Dijkstra algoritmus nem működik negatív élsúlyok esetén, ha a gráfunk tartalmaz negatív élsúlyt, keressük meg a legkisebb élsúlyt, és ennek abszolút értékével növeljük meg az összes él súlyértékét. Ezután a gráf már nem tartalmaz negatív élsúlyt. Ezután lefuttatva a Dijkstra algoritmust, megkapjuk az eredeti gráf legrövidebb útjait, melyeknek valódi költségét az eredeti súlyértékek felhasználásával már könnyen kiszámíthatjuk. Bizonyítsa vagy cáfolja a fenti elgondolást!
11. Tegyük fel, hogy a gráfunk nem tartalmaz negatív súlyú élt. Használhatjuk-e a Dijkstra algoritmust leghosszabb út keresésére? Ha igen, adja meg, hogy mit kell módosítani az algoritmuson!
12. Tegyük fel, hogy a gráfunk csak negatív súlyú éleket tartalmaz, szeretnénk leghosszabb utakat keresni benne. Szorozzuk meg minden él súlyát -1 -el. Ezután már nem fog tartalmazni a gráfunk negatív súlyú élt. Igaz-e, hogy a Dijkstra algoritmust lefuttatva a módosított gráfon, megtalálja az eredeti gráf leghosszabb útjait?
13. Működik-e irányítatlan gráfokon a Dijkstra algoritmus, és ha igen milyen feltételek mellett?
14. Az irányított gráfunk tartalmaz negatív súlyú élt. Milyen feltételek mellett adja meg a legrövidebb utakat a Bellman-Ford algoritmus?
15. Működik-e irányítatlan gráfokon a Bellman-Ford algoritmus, és ha igen milyen feltételek mellett?
16. Hány menetben végez minden élen közelítést a Bellman-Ford algoritmus?
17. Mit állíthatunk az első menet után? Mit állíthatunk az i . menet után?
18. Mit tehattünk, ha egy menetben nem volt változás?

5.4 Gyakorló feladatok

1. Szemléltesse a Dijkstra algoritmus működését az alábbi gráfon! Adja meg menetenként a d és P_i tömbök tartalmát! (Egy menet, egy csúcs prioritásos sorból való kivétele és feldolgozása.)

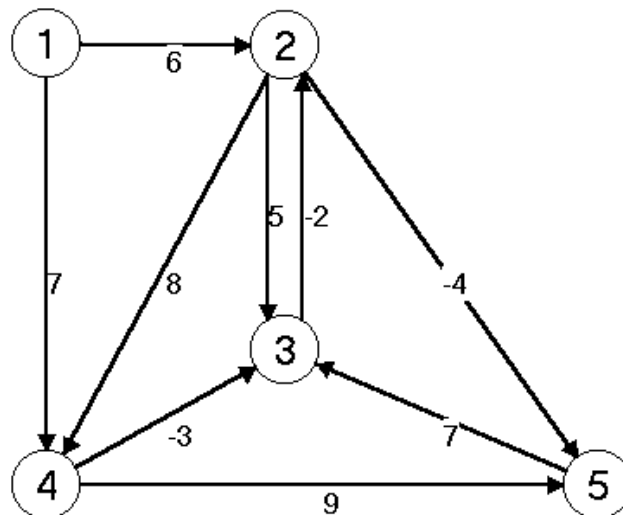


2. A Dijkstra algoritmus lefuttatása után a d tömbök alábbi, menetenkénti sorozatát kaptuk. Mi lehetett a gráf? Próbálja behúzni azokat az éleket, amelyek a d -beli értékekből következnek, és próbálja előállítani a P_i tömbök menetenkénti sorozatát!

0	∞	∞	∞	∞	∞
0	2	∞	∞	3	∞
0	2	6	4	3	∞
0	2	5	4	3	9
0	2	5	4	3	9
0	2	5	4	3	9

3. Adottak repülőjáratok induló és cél állomásokkal, továbbá a repülőjegy árakkal. Jusson el repülővel A városból B városba a legolcsóbb úton!
4. Adott egy olajfűró torony és néhány olajfinomító. Milyen csőhálózatot építsünk ki a torony és a finomítók között, hogy az olaj szállítása minimális költségű legyen. Tegyük fel, hogy a csővezetékeknél és az olajfűró toronynál nem ütközünk kapacitási korlátokba, és tudjuk az egyes pontok közötti leendő olajvezetékek szállítási költségeit.
5. Módosítsa az előző feladatot, k olajfűró torony és m olajfinomító esetére!
6. Egy kommunikációs hálózatot egy irányítatlan gráffal modellezünk, ahol $[u, v] \in E$ él, az u - és a v -t összekötő kommunikációs csatornát szimbolizálja. Az élekhez hozzárendeltük annak valószínűségét, hogy a csatorna működik (nem sérült). Az egyes csatornák működőképessége egymástól független. Adjon algoritmust két pont közötti legmegbízhatóbb út előállítására. [1]

7. Adott egy város tömegközlekedési hálózatának gráfes modellje. A megállóhelyeknek megfeleltettük a gráf csúcsait, egy járat két megálló közé eső szakaszának megfeleltettük a gráf éleit. Az élek tulajdonsága legyen a járat azonosítója (pl.: 6-os villamos, 3-as metró stb.), az élek súlya legyen az illető tömegközlekedési eszköz menetideje a két megálló között. Adjon algoritmust, amely megadja, hogyan tudunk eljutni a gráf egyik csúcsából egy másik csúcsába a legrövidebb idő alatt, ha az átszállásokkal járó idővesztésedet egységesen 10 percnak tekintjük!
8. Adva van egy város, amelyben több mentőállomás és kórház működik. Riasztás esetén a diszpécser értesíti a riasztás helyszínéhez legközelebb eső mentőállomást, megad egy lehetséges legrövidebb útvonalat az állomástól a helyszínig, és megad egy legrövidebb útvonalat a helyszíntől a legközelebbi kórházig. Írjunk programot, amely meghatározza ezeket a legrövidebb útvonalakat!
9. Egy kitűzött időpontban A városból elindulva, menetrendszerinti járatokkal, minél gyorsabban jussunk el B városba. Adottak a használható közlekedési eszközök menetrendjei (repülő, hajók, vonatok stb.). Tekintsünk el egy városban az egyes érkezési-indulási helyszínek (reptér-kikötő stb.) közötti eljutási időktől.
10. Legyen $G = (V, E)$ súlyozott gráf, $c : E \rightarrow \{0, \dots, W-1\}$ egész értékű súlyfüggvénnyel. Módosítsa a Dijkstra algoritmust, hogy a műveletigénye $O(W * n + e)$ legyen! [1]
11. Módosítsa az előző feladatban kapott algoritmust $O((n + e) \log W)$ futási idejűre! [1]
12. Szemléltesse a Bellman-Ford algoritmus működését az alábbi gráfon! Rajzolja fel a d és P_i tömbök tartalmát iterációnként $(1..n-1)$!



13. Gyorsítsa a Bellman-Ford algoritmust, ha egy iterációban nem volt változás, akkor megállhatunk!
14. Gyorsítsa a Bellman-Ford algoritmust, úgy hogy bizonyos éleket nem kell vizsgálni! Írja fel, állítás ábrázolás esetén!

15. Adott egy élsúlyozott, negatív összsúlyú köröket nem tartalmazó, irányított gráf. Tegyük fel, hogy az egyszerű utak élszáma legfeljebb konstans k . Gyorsítson a Bellman-Ford algoritmuson!
16. Egészítsük ki a Bellman-Ford algoritmust úgy, hogy egy logikai változóban visszaadja, hogy van-e a gráfban negatív összsúlyú kör vagy sem, mivel az algoritmus által kiszámított költségek, csak akkor érvényesek, ha ilyen kör nincs a gráfban!
17. Adott egy súlyozott, irányított gráf. Amennyiben tartalmaz negatív összköltségű kört, írassuk ki egy ilyen kör mentén lévő csúcsokat!
18. Adott egy $G=(V,E)$ élsúlyozott, irányított vagy irányítás nélküli, véges gráf. Továbbá adott, egy $s \in V$ forrás (kezdőcsúcs) és $v \in V$ célcsúcs. Adjunk algoritmust, amely meghatározza s -ből v -be vezető leghosszabb utat és annak hosszát!
19. Adottak valuták és valutaárfolyamok, azaz egy olyan irányított gráf, amelyben a csúcsok az egyes valuták, és az élek súlyai az árfolyamok. Az u -ból a v -be menő él súlya megadja, hogy egységnyi u valutáért mennyi v valuta kapható. Váltsa át u valutát v -re, úgy hogy a lehető legtöbb v -t kapjon! [1]
20. Arbitrázs a valutaváltási árfolyamokban rejlő egyenlőtlenségek olyan hasznosítása, amikor egy valuta 1 egységéből elindulva, egy valutaváltási sorozat lefolytatása után, ugyanazon valuta 1 egységénél nagyobb értékére teszünk szert. (Pl.: 1 dollárért veszünk 0,7 fontot, majd a fontot átváltjuk frankra 9,5-es szorzóval, majd a frankot ismét dollárra váltjuk 0,16-os szorzóval, ekkor $0,7 \cdot 9,5 \cdot 0,16 = 1,064$. Tehát 6,4%-os haszonra tettünk szert.) Adjunk algoritmust, amely meghatározza, hogy létezik-e a valutáknak ilyen sorozata, és ha létezik, adjon meg egy ilyet! [1]

5.5 Összefoglalás

- A gráf egy útjának a hossza: az utat alkotó élek súlyainak az összege.
- Két csúcs távolsága: a két csúcs közötti legrövidebb út hossza.
- A Dijkstra algoritmus meghatározza a gráf csúcsainak a kezdőcsúctól vett távolságát, negatív éleket nem tartalmazó gráf esetén.
- Dijkstra algoritmus elve:
Minden lépésben tartsuk nyilván az összes csúcsra, a forrástól az illető csúcsba vezető, eddig talált legrövidebb utat (a már megismert módon a $d[1..n]$ tömbben a távolságot, és $P[1..n]$ tömbben a megelőző csúcsot).
 - Kezdetben a távolság legyen a kezdőcsúcsra 0, a többi csúcsra ∞ .
 - Minden lépésben a nem KÉSZ csúcsok közül tekintsük az egyik legkisebb távolságú (d_{\min}) csúcsot:
 - Azt mondhatjuk, hogy ez a $v \in V$ csúcs már KÉSZ, azaz ismert a hozzá vezető legrövidebb út.
 - A v -t terjesszük ki, azaz v csúcs szomszédaira számítsuk ki a (már ismert) v -be vezető, és onnan egy kimenő éllel meghosszabbított út hosszát. Amennyiben ez jobb (kisebb), mint az illető szomszédba eddig talált legrövidebb út, akkor innentől kezdve ezt az utat tekintsük, az adott szomszédba vezető, eddig talált legrövidebb útnak. Ezt az eljárást szokás közelítésnek is nevezni.

Az algoritmus tulajdonképpen minimumválasztó prioritásos sort használ a már megismert távolságok nyilvántartására. Minden iterációs menetben kivesszük a legkisebb távolságú csúcsot (kész van), és kiterjesztjük

- A prioritásos sort megvalósíthatjuk rendezetlen tömbben (d tömb) feltételes minimumkereséssel, vagy kupaccal.
- Műveletigény:
 - Rendezetlen tömb esetén:

$$T(n) = O(1 + n - 1 + 1 + 0 + n^2 + n + e) = O(n^2 + e) = O(n^2)$$
 - Kupac esetén:

$$T(n) = O(1 + n - 1 + 1 + n + n * \log n + n + e * \log n) = O((n + e) * \log n)$$
- Következmény az ábrázolásra: sűrű gráf esetén: csúcsmátrix + rendezetlen tömb, ritka gráf esetén: éllista + kupac.
- A Bellman-Ford algoritmus meghatározza a gráf csúcsainak a kezdőcsúctól vett távolságát, negatív összköltségű kört nem tartalmazó gráf esetén (irányítatlan gráfnál negatív élt nem tartalmazó).
- 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 élszáma, legfeljebb n-1 lehet.
- A Bellman-Ford algoritmus a Dijkstra algoritmusnál megismert közelítés műveletét végzi. Egy menetben az összes élre megvizsgálja, hogy javító él-e vagy sem. Összesen n-1 menetet végez.
- Műveletigénye $T(n) = \Theta((n - 1) * e)$
- Gyorsítási lehetőség: ha egy menetben nem volt változás, akkor készen vagyunk

6. Legrövidebb utak minden csúcspárra

Probléma: Adott egy $G=(V,E)$ élsúlyozott véges gráf. Szeretnénk meghatározni, $\forall u,v \in V$ csúcsra, u -ból v -be vezető **legkisebb költségű** utat.

Biztos mindenki látott már, az autós térképekben előforduló, a városok egymástól való legkisebb távolságait tartalmazó táblázatot. Ez egy négyzetes táblázat, ahol a sorok és az oszlopok a városok neveivel vannak felcímkézve. A táblázat x címkéjű sorának és y címkéjű oszlopának a metszéspontjában található, az y városnak az x várostól való legkisebb távolsága. Modellezzük az autós térképet egy gráffal (irányított vagy irányítatlan, attól függően, hogy vannak-e egyirányú utak). A csúcsokat megfeleltetjük a városoknak, az élek pedig a városokat összekötő közvetlen utaknak. Az utak hossza legyen az élek súlya, tehát a gráf legyen élsúlyozott. Célunk a fenti táblázat előállítás.

A csúcspárok közötti legkisebb költségű utakat megkereshetjük az előző feladatban tanult algoritmusok segítségével. Minden csúcsot forrásként tekintve futtassuk le a "legrövidebb utak egy forrásból algoritmusok" egyikét [1]:

- 1) Amennyiben az élsúlyok nem negatívak, a **Dijkstra algoritmusát** alkalmazhatjuk. Ekkor, műveletigény:
 - a) Prioritásonként rendezetlen tömböt használva: $T(n)=n*O(n^2)=O(n^3)$
 - b) Prioritásonként kupacot használva: $T(n)=n*O((n+e)*\log n)=O(n^2 \log n + n*e*\log n)$, amit ritka gráfokra alkalmazva $T(n)=O(n^2 \log n)$.
- 2) Ha negatív élsúlyokat is megengedünk, akkor a **Bellman-Ford algoritmust** használhatjuk, amellyel a műveletigény $T(n) = n*(n-1)*e = \Theta(n^2 * e)$. Ez ritka gráfokra $T(n) = O(n^3)$, sűrű gráfokra $T(n) = \Theta(n^4)$.

Ebben a fejezetben egyrészt a negatív élsúlyok esetére hatékonyabb algoritmust adunk, továbbá vizsgáljuk ennek speciális változatát gráfok tranzitív lezártjának a kiszámítására.

6.1 Floyd algoritmus

Feladat: Adott egy $G=(V,E)$ élsúlyozott, irányított vagy irányítás nélküli, **negatív összköltségű irányított kört nem tartalmazó** véges gráf. Határozzuk meg $\forall u,v \in V$ csúcsra, u -ból v -be vezető **legkisebb költségű** utat!

A fejezet további részében az utak hosszán az út mentén szereplő élek költségeinek az összegét értjük (5.0.1. definíció), a csúcspárok távolságán pedig a csúcspár közötti (az 5.0.2. definíció szerinti) egyik legrövidebb út hosszát értjük. Tegyük fel, hogy $V=\{1,2,\dots,n\}$, és hogy a G gráf az C szomszédsági mátrixával adott. A csúcspárok távolságának a kiszámítására egy szintén $n \times n$ -es D mátrixot fogunk használni.

6.1.1. Definíció [1]: Legyen egy $p = \langle v_1, v_2, \dots, v_m \rangle$ egyszerű út **belső csúcsa** p minden v_1 -től és v_m -től különböző csúcsa, azaz $\{v_2, \dots, v_{m-1}\}$ halmaz elemei.

Az algoritmus elve [1]: n iterációs lépés után kapjuk meg a megoldást, mely iterációs lépések során folyamatosan fenntartjuk a $D^{(k)}$ mátrixunkra a következő **invariáns tulajdonságot**: a k -adik iteráció lefutása után $\forall (i, j)$ csúcspárra $D^{(k)}[i, j]$ azon $i \rightsquigarrow j$ utak legrövidebbjeinek a hosszát tartalmazza, amelyek közbülső csúcsai k -nál nem nagyobb sorszámúak. Tehát $k=n$ esetén $\forall (i, j)$ csúcspárra $D^{(n)}[i, j]$ az $i \rightsquigarrow j$ utak legrövidebbjeinek a hosszát, azaz a feladat megoldását tartalmazza.

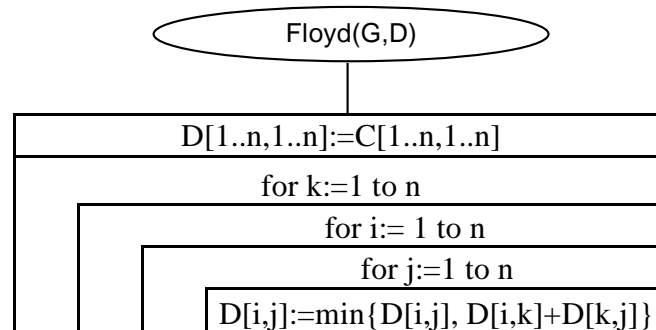
Az invariáns tulajdonság fenntartása: (k szerinti teljes indukció)

- $k=0$ esetben: $\forall (i, j)$ csúcspárra $D^{(0)}[i, j]$ tartalmazza azon $i \rightsquigarrow j$ utak közül a legkisebb költségű utak hosszát, amely belső csúcsainak sorszáma kisebb, mint 1 , azaz nem tartalmaznak belső csúcsot. Ami nem más, mint a C szomszédsági mátrixban szereplő érték. Tehát $D^{(0)}$ mátrix értéke legyen C szomszédsági mátrix.
- $k-1 \rightarrow k$: a $D^{(k)}[i, j]$ értéket szeretnénk kiszámítani a $D^{(k-1)}$ mátrix értékeinek a felhasználásával. Két esetet különböztetünk meg aszerint, hogy $p^{(k)} = i \rightsquigarrow j$ (i -ből j -be vezető, belső csúcsként nem nagyobb, mint k sorszámú csúcsokat tartalmazó) egyik legrövidebb útnak, k belső csúcsa vagy sem. ($p^{(k)}$ út legyen egyszerű út, mert ha tartalmazna kört, és nem lehet negatív összköltségű a kör, akkor a kört "kivágva" a kapott út költsége nem nő, tehát a legrövidebb utak között vannak egyszerűek.)
 - 1) Ha k nem belső csúcsa $p^{(k)}$ -nek, akkor $p^{(k)}$ minden belső csúcsának sorszáma legfeljebb $k-1$, azaz $p^{(k)}$ hossza azonos a legfeljebb $k-1$ belső csúcsokat tartalmazó $i \rightsquigarrow j$ legrövidebb út hosszával $D^{(k-1)}[i, j]$ -vel.
 - 2) Ha k belső csúcs a $p^{(k)}$ úton, akkor felbonthatjuk $p_1^{(k-1)} = i \rightsquigarrow k$ és $p_2^{(k-1)} = k \rightsquigarrow j$ legfeljebb $k-1$ sorszámú belső csúcsokat tartalmazó i -ből k -ba ill. k -ból j -be vezető legrövidebb egyszerű utakra (legrövidebb út részútja is legrövidebb út, 5.1.1. lemma következménye). Tehát $D^{(k)}[i, j] = D^{(k-1)}[i, k] + D^{(k-1)}[k, j]$.
 Tehát a két eset közül az adja a rövidebb utat, ahol kisebb a számított érték, azaz a kérdéses legrövidebb út hossza megkapható az alábbi képlettel:

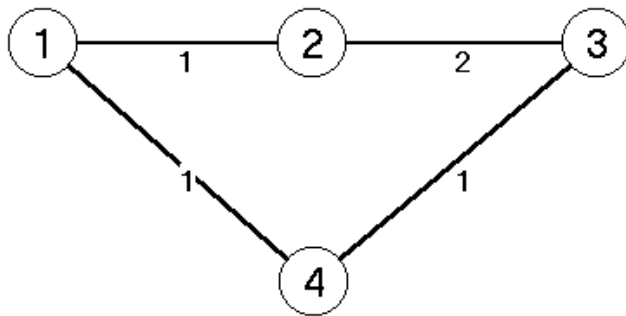
$$D^{(k)}[i, j] = \min\{ D^{(k-1)}[i, j], D^{(k-1)}[i, k] + D^{(k-1)}[k, j] \}$$

„Mivel $D^{(k)}[i, k] = D^{(k-1)}[i, k]$ és $D^{(k)}[k, j] = D^{(k-1)}[k, j]$, így elegendő egyetlen D mátrix az algoritmus végrehajtásához.” [2]

Floyd algoritmus az ábrázolás szintjén, csúcsmátrixos ábrázolással [1]:



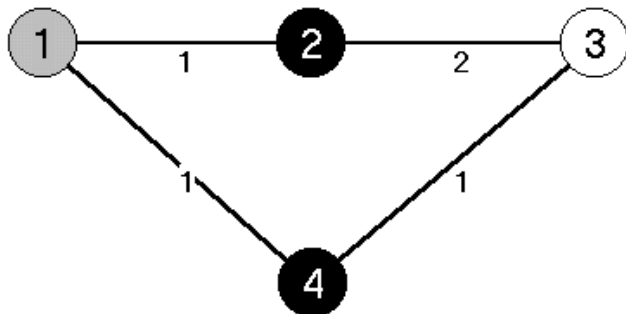
Most nézzük meg egy példán, **ADS szinten** az algoritmus működését:



A kezdeti inicializáló lépés után D mátrix megegyezik a gráf csúcsmátrixának értékével.

$$D^{(0)} = \begin{pmatrix} 0 & 1 & \infty & 1 \\ 1 & 0 & 2 & \infty \\ \infty & 2 & 0 & 1 \\ 1 & \infty & 1 & 0 \end{pmatrix}$$

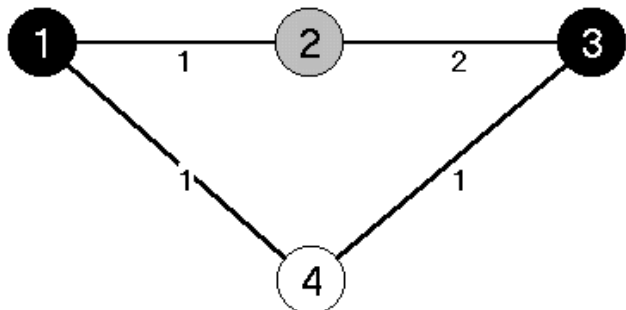
Az **első iteráció során**, az 1-es csúcson átmenő utakkal próbáljuk javítani a mátrix értékeit.



Amikor a 2-es csúcsból a 4-es csúcsba menő utakat vizsgáljuk, találunk az 1-esen átmenő javító utat, $D[2,4]$ értékét 2-re javítjuk. Mivel a gráf irányítatlan, így a szimmetrikus esetben is történik javítás ($D[4,2]$).

Az első iterációs lépés után a következő mátrix alakul ki: $D^{(1)} = \begin{pmatrix} 0 & 1 & \infty & 1 \\ 1 & 0 & 2 & 2 \\ \infty & 2 & 0 & 1 \\ 1 & 2 & 1 & 0 \end{pmatrix}$

A második iterációs lépésben: már olyan javító utakat keresünk, amelyek belső csúcsainak a sorszáma legfeljebb 2. Vizsgáljuk az legfeljebb 1-es sorszámú belső csúcsokat tartalmazó utakat (ill. a még nem létező utakat), és megpróbáljuk közbülső csúcsnak beilleszteni a 2-es csúcsot.



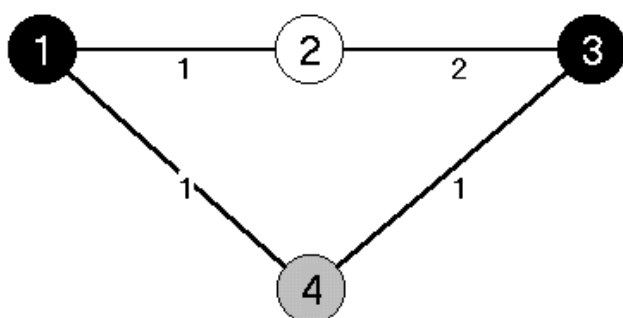
Az 1-ből a 3-ba ill. 3-ból az 1-be találtunk javító utat (az eddig nem létező úthoz, a ∞ hosszú úthoz képest) a 2-esen át.

A második iterációs lépés után a következő mátrix alakul ki: $D^{(2)} = \begin{pmatrix} 0 & 1 & 3 & 1 \\ 1 & 0 & 2 & 2 \\ 3 & 2 & 0 & 1 \\ 1 & 2 & 1 & 0 \end{pmatrix}$.

A harmadik iterációban nem találunk a 3-as csúcson átmenő javító utakat, így a mátrix nem

változik. $D^{(3)} = \begin{pmatrix} 0 & 1 & 3 & 1 \\ 1 & 0 & 2 & 2 \\ 3 & 2 & 0 & 1 \\ 1 & 2 & 1 & 0 \end{pmatrix}.$

A negyedik iterációs lépésben: már olyan javító utakat keresünk, amelyek belső csúcsainak a sorszáma legfeljebb 4. Vizsgáljuk a legfeljebb 3-as sorszámú belső csúcsokat tartalmazó utakat (ill. a még nem létező utakat), és megpróbálunk a 4-es csúcson átmenő, kisebb költségű "elkerülő" utat találni.



Találunk a 4-es csúcson átmenő javító utat. Eddig az 1-esből a 3-mas csúcsba vezető, legfeljebb 3-as sorszámú belső csúcsokat tartalmazó legrövidebb út hossza 3 volt. Ez az út: 1,2,3. Most megengedjük, hogy belső pont sorszáma lehet 4 is, így megvizsgálva az 1,4 ill. 4,3 részutak hosszának összegét, az kevesebb mint, az 1,2,3 út hossza, tehát találtunk egy kisebb költségű elkerülő utat. Természetesen ez a szimmetrikus esetre is igaz.

Végül a negyedik iterációs lépés után megkapjuk a végeredményt: $D^{(4)} = \begin{pmatrix} 0 & 1 & 2 & 1 \\ 1 & 0 & 2 & 2 \\ 2 & 2 & 0 & 1 \\ 1 & 2 & 1 & 0 \end{pmatrix}.$

Az algoritmus műveletigénye [1]: az algoritmus n iterációs lépésben, az n^2 -es mátrix minden elemére konstans számú műveletet végez, így $T(n) = \Theta(n^3)$. Ez egy **stabil algoritmus**, mivel legjobb, legrosszabb és átlagos esetben is azonos a műveletigénye.

Megjegyzések:

1) A műveletigényünk nagyságrendileg ugyanannyi, mintha a **Dijkstra algoritmust csúcsmátrixos** ábrázolású gráfon, prioritásos sorként **rendezetlen tömböt** használva, minden csúcsra, mint forrásra lefuttatnánk. A Dijkstra algoritmusnál már láttuk, hogy ezt a megvalósítást **sűrű gráfok** esetén célszerű alkalmazni. **Ritka gráfok** esetén **éllista** ábrázolást használhatunk, prioritásos sorként pedig **kupacot**, így a műveletigény $T(n) = O(n^2 \log n)$, ami jobb, mint a Floyd algoritmus műveletigénye.

Úgy tűnik, hogy felesleges a Floyd algoritmust használni, mivel a **Dijkstra algoritmus jobb eredményt ad**, azonban a Dijkstra algoritmust csak **nem negatív költségű élek** esetén használhatjuk.

Amennyiben **előfordulhat** a gráfban **negatív súlyú él** (de negatív összköltségű kör nem), a **Bellman-Ford** algoritmus használatával **ritka gráfokra** $T(n) = O(n^3)$, **sűrű gráfokra**

$T(n) = \Theta(n^4)$ műveletigénnyel tudjuk megoldani a feladatot. Látható, hogy sűrű gráfok esetén a **Floyd algoritmus hatékonyabb**.

2) A Floyd algoritmust az **ábrázolás szintjén** adtuk meg mátrixos ábrázolás mellett, mint ahogy a szakirodalomban szokás.

3) Amennyiben a csúcspárok közötti legrövidebb **utakra is kíváncsiak vagyunk** (és nem csak azok hosszára), a korábban már látott módon eltárolhatjuk a megelőző (vagy közbülső k címkéjű) csúcsot. Mivel most csúcspárok közötti utakról van szó minden lehetséges csúcspárra ($n*n$ csúcspár), így érdemes mátrixot használni.

6.2 Tranzitív lezárt

Most azt vizsgáljuk meg, hogy a gráf egy u pontjából el tudunk-e jutni egy v pontjába, azaz létezik-e út u -ból v -be. A fejezet további részében a gráfunk legyen véges, súlyozatlan, irányított vagy irányítatlan, az utak hosszán pedig az út mentén található élek számát értjük (4.2.1. definíció szerint).

6.2.1. Tétel [1]: Legyen a G gráf szomszédsági mátrixa $C \Rightarrow C^k[i, j]$ ($1 \leq i, j \leq n$, $k \in \mathbb{N}$) az i -ből a j -be vezető k hosszúságú utak számát adja meg.

Bizonyítás: **k szerinti teljes indukcióval**

- $k=1$ esetben: 1 hosszú út=él. Tehát $C^1[i, j]$ az i -ből a j -be menő élek számát adja meg, (ami megállapodás szerint legfeljebb 1 lehet), ez pedig pontosan a szomszédsági mátrix definíciója.
- $k-1 \rightarrow k$: Tegyük fel, hogy $k-1$ -ig teljesül az állítás.

Kérdés, hányféleképpen juthatok el k hosszú úton i -ből j -be? $i \rightsquigarrow j$ k hosszúságú utat a következő módon tudjuk **felbontani**: $i \rightsquigarrow s$ $k-1$ hosszú út, majd $s \rightarrow j$ él. Kérdés, **hányféleképpen juthatok el k hosszú úton i -ből j -be úgy, hogy a j -t megelőző csúcs az s ?** Az indukciós feltevés szerint, $C^{k-1}[i, s]$ féle módon juthatok el $k-1$ hosszú úton i -ből s -be, továbbá ha létezik $s \rightarrow j$ él a gráfban ($C[s, j]=1$), akkor azon már csak egyféleképpen tudunk tovább menni j -be, azaz $C^{k-1}[i, s] * C[s, j]$ megadja az i -ből j -be menő k hosszú utak számát, ahol j előtti megelőző csúcs az s . Amennyiben nem létezik $s \rightarrow j$ él, akkor a szorzat értéke nulla, ami kifejezi, hogy nincs ilyen út a gráfban.

Az előzőekben s -en átmenő utakat számláltunk, de s tetszőleges csúcsa lehet a gráfnak, így ezeket **minden $s \in V$ csúcsra összegezzük**:

$$C^k[i, j] = \sum_{s=1}^n (C^{k-1}[i, s] * C[s, j]) \quad \forall i, j - \text{re } (1 \leq i, j \leq n)$$

,ami nem más, mint egy **mátrix szorzat** $C^k = C^{k-1} * C$.

A tétel következménye: $C + C^2 + C^3 + \dots + C^k$ mátrix $[i, j]$ -edik eleme, az i -ből j -be menő legfeljebb k hosszúságú utak számát adja meg.

6.2.2. Definíció [1]: $G=(V,E)$ véges gráf útmátrixa (elérhetőségi mátrixa):

$$\forall i, j - \text{re } (1 \leq i, j \leq n) : \quad U[i, j] = \begin{cases} 1 & , \text{ ha } \exists i \leadsto j \text{ út a gráfban} \\ 0 & , \text{ különben} \end{cases}$$

,ahol $n = |V|$.

Az útmátrix meghatározása [1]: ha létezik i -ből j -be menő út a gráfban, akkor létezik i -ből j -be menő egyszerű út is, továbbá minden egyszerű út legfeljebb $n-1$ hosszú, egy egyszerű kör pedig legfeljebb n hosszú. Tehát számoljuk ki $C + C^2 + C^3 + \dots + C^n$ összeget, és az eredménymátrixban a nullánál nagyobb elemeket írjuk át 1-esre.

6.2.3. Definíció [1]: Egy $G=(V,E)$ gráf **transzitiv lezárása** $G'(V',E')$ gráf ,ahol $V'=V$ és $(u,v) \in E' \Leftrightarrow \exists u \leadsto v$ út a gráfban.

Néhány triviális állítás:

- 1) G útmátrixa G' szomszédsági mátrixa
- 2) G erősen összefüggő $\Leftrightarrow U$ -ban nincs nulla elem $\Leftrightarrow G'$ teljes gráf

6.3 Warshall algoritmus

Az előző fejezetben egy gráf transzitiv lezártját elő tudtuk állítani a szomszédsági mátrix hatványainak az összegeként, amelynek hatékonysága $T(n) = \Theta(n^4)$. Most lássunk egy nagyságrenddel hatékonyabb módszert.

A transzitiv lezárt meghatározására használhatnánk a **Floyd algoritmust is**, hiszen az algoritmus lefutása után, ha $D[i,j]$ véges, akkor létezik $i \leadsto j$ út, ha végtelen, akkor pedig nincs i -ből j -be menő út. Azonban a Floyd algoritmus elvét felhasználva, szép algoritmus adható az **ábrázolás szintjén** a problémára („bár S. Warshall nevéhez fűződő algoritmus megelőzte Floydét” [2]).

Adott a $G=(V,E)$ véges, súlyozatlan, irányított vagy irányítatlan gráf. A **Floyd algoritmushoz képest az alábbi változtatások után** megkapjuk a **Warshall algoritmust [2]:**

- 1) A W mátrix (a Floyd-nál D -vel jelölt mátrix) kezdeti értéke legyen

$$\forall i, j - \text{re } (1 \leq i, j \leq n) : W[i, j] = \begin{cases} 1 & , \text{ ha } i = j \text{ vagy } (i, j) \in E \\ 0 & , \text{ különben} \end{cases}.$$

A mátrixban szereplő értékeket tekintsük logikai értékeknek (hamis=0, igaz=1).

- 2) A ciklusban végzett művelet pedig legyen a következő:
 $W[i, j] := W[i, j] \vee (W[i, k] \wedge W[k, j])$

Az algoritmus helyességének a belátása hasonlóan történhet, mint a Floyd algoritmusnál.

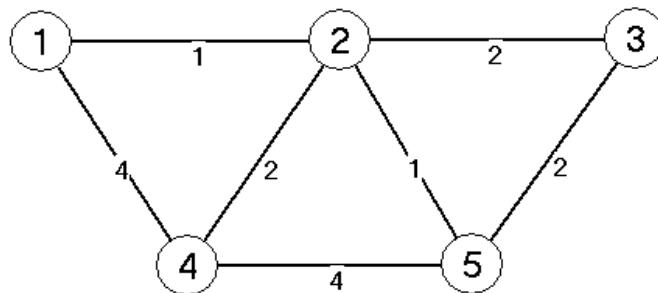
„Természetesen a **műveletigény** is aszimptotikusan megegyezik a Floyd algoritmus műveletigényével, azzal a különbséggel, hogy a Floyd algoritmusnál, a ciklus belsejében konstans műveletnek tekintett összeadás és minimumválasztás helyett, most logikai műveleteket végzünk, ami hatékonyabb lehet.” [1]

6.4 Ellenőrző kérdések

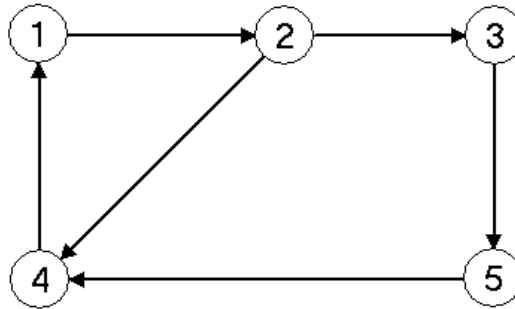
1. Működik-e negatív élsúlyokat tartalmazó gráfon a Floyd algoritmus, ha igen milyen feltételek mellett?
2. Működik-e irányítatlan gráfon a Floyd algoritmus, ha igen milyen feltételek mellett?
3. Mi az algoritmus elve?
4. Milyen adatszerkezetű az algoritmus eredménye?
5. Mi a Floyd algoritmus műveletigénye?
6. Mikor használjuk Floyd algoritmus és mikor minden csúcsból elindított legrövidebb utak meghatározását?
7. Mit ad meg a szomszédsági-mátrix négyzete, k -dik hatványa?
8. Mi az útmátrix (elérhetőségi mátrix) definíciója?
9. Hogyan határozható meg az útmátrix a szomszédsági-mátrix hatványainak segítségével?
10. Mi a tranzitív lezárt definíciója?
11. Mi a kapcsolat egy gráf útmátrixa és a tranzitív lezártja között?
12. Mi a Warshall algoritmus eredménye?
13. Mi a különbség a Warshall és a Floyd algoritmus között?

6.5 Gyakorló feladatok

1. Határozza meg a következő gráf esetén minden csúcspárra a csúcspár közötti legrövidebb út hosszát!



2. Írja fel a Floyd algoritmust a gráf szomszédsági-mátrixos ábrázolása esetén!
3. Írja fel a Floyd algoritmust a gráf éllistas ábrázolása esetén!
4. Módosítsa a Floyd algoritmust, hogy a legrövidebb utakat is ki lehessen írni! Adj meg az algoritmust, amely kiírja egy csúcspár közötti legrövidebb utat!
5. Határozza meg a következő gráf útmátrixát!



6. Írja fel a Warshall algoritmust a gráf szomszédsági-mátrixos ábrázolása esetén!
7. Írja fel a Warshall algoritmust a gráf éllistas ábrázolása esetén!
8. Adja meg egy gráf csúcspáira a legrövidebb olyan utakat, amelyek átmennek egy megadott csúcson!

6.6 Összefoglalás

- A Floyd algoritmus meghatározza minden csúcspárra a legrövidebb utat, negatív összköltségű kört nem tartalmazó gráf esetén.
- Floyd algoritmus elve: n iterációs lépésben futva folyamatosan fenntartva a $D^{(k)}$ mátrixra a következő invariáns tulajdonságot: a k -adik iteráció lefutása után $\forall(i, j)$ csúcspárra $D^{(k)}[i, j]$ azon $i \rightsquigarrow j$ utak legrövidebbjeinek a hosszát tartalmazza, amelyek közbülső csúcsai k -nál nem nagyobb sorszámúak. Tehát $k=n$ esetén $\forall(i, j)$ csúcspárra $D^{(n)}[i, j]$ az $i \rightsquigarrow j$ utak legrövidebbjeinek a hosszát, azaz a feladat megoldását tartalmazza. A megvalósításnál egyetlen D mátrix elegendő. Minden menetben az invariáns fenntartása: $D^{(k)}[i, j] = \min\{ D^{(k-1)}[i, j], D^{(k-1)}[i, k] + D^{(k-1)}[k, j] \}$
- A szomszédsági-mátrix k -dik hatványa a k hosszúságú utak számát adja meg. A $C + C^2 + C^3 + \dots + C^k$ pedig a legfeljebb k hosszúságú utak számát.
- Transitív lezárt: a gráf transitív lezártja olyan gráf, amelyben két csúcs között akkor van él, ha az eredeti gráfban létezik út, csúcsmátrixát az eredeti gráf útmátrixának nevezzük, amely kiszámítható $C + C^2 + C^3 + \dots + C^n$ összegből, ha az eredménymátrixban a nullánál nagyobb elemeket írjuk át 1-esre.
- A Warshall algoritmus meghatározza a gráf útmátrixát (transzitiv lezártját).
- A Warshall algoritmus elve megegyezik a Floyd algoritmusával, az iterációs ciklusban logikai műveleteket végzünk: $W[i, j] := W[i, j] \vee (W[i, k] \wedge W[k, j])$
- Mindkét algoritmus műveletigénye: $T(n) = \Theta(n^3)$

7. Minimális költségű feszítőfák

„A bevezető 1.3. fejezetében már említettük a témakörben klasszikusnak számító példát, miszerint egy terület villamosítását kell megoldani a lehető legkisebb költséggel. Tudomásunk szerint a probléma első érdemi megoldását Otakar Boruvka brnoi professzor közölte 1926-ban, aki Morvaország nyugati részének villamosítása kapcsán találkozott a feladattal, amely szerint minimális összköltségű vezetékrendszert kellett tervezni megadott városok között.” [2]

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.

7.0.1. Definíció [2]: Legyen $G=(V,E)$ irányítatlan gráf. A $G'=(V',E')$ gráfot a G **részgráfjának** nevezzük, ha $V' \subseteq V$ és $E' \subseteq E$, továbbá $\forall [u,v] \in E': u,v \in V'$.

7.0.2. Definíció [2]: Legyen $G=(V,E)$ irányítatlan, összefüggő, véges gráf. A G egy körmentes, összefüggő $F=(V,E')$ részgráfját a G egy **feszítőfájának** nevezzük. (F és G pontjainak halmaza megegyezik)

7.0.3. Definíció [2]: Legyen $G=(V,E)$ irányítatlan, összefüggő, élsúlyozott, véges gráf a $c: E \rightarrow R$ költségfüggvénnyel. Ekkor $F=(V,E')$ feszítőfa a G egy **minimális költségű feszítőfája**, ha költsége $C(F) = \sum_{e \in E'} c(e)$ minimális a G feszítőfái között, azaz

$$C(F) = \min \{ C(H) \mid H \text{ a } G \text{ feszítőfája} \}.$$

Feladat: Adott egy $G=(V,E)$ irányítatlan, összefüggő, élsúlyozott, véges gráf. Határozzuk meg a G egy **minimális költségű feszítőfáját**.

A továbbiakban tekintsünk néhány fákkal kapcsolatos állítást, amelyek a későbbi bizonyítások során hasznosak lehetnek.

7.0.4. Állítás [2]: Minden legalább kétpontú fában van **elsőfokú csúcs**.

Bizonyítás [3]: Tekintsük $u = \langle v_0, v_1, \dots, v_k \rangle$ egyik leghosszabb utat a fában. Ha v_0 -ból menne el egy olyan csúcsba, amely nem eleme $\{v_1, v_2, \dots, v_k\}$ halmaznak, akkor u nem lenne a leghosszabb út, ha v_0 -ból menne el egy olyan csúcsba, amely eleme $\{v_1, v_2, \dots, v_k\}$ halmaznak, akkor az útban lenne kör, tehát nem lenne fa. Így azt kaptuk, v_0 elsőfokú csúcs.

7.0.5. Állítás [2]: Minden összefüggő $G=(V,E)$ gráfnak van **feszítőfája**.

Bizonyítás [3]: Ha a gráfban van kör, elhagyjuk az egyik élet. Ezt véges sokszor ismételve körmentes, összefüggő V csúcshalmazú gráfot kapunk, tehát feszítőfát.

7.0.6. Állítás [2]: Egy n pontú összefüggő gráf fa $\Leftrightarrow n-1$ éle van
Bizonyítás [2]:

\Rightarrow : n pontú fából törölünk egy elsőfokú csúcsot (7.0.4. szerint létezik) és a hozzá tartozó élt, akkor egy $n-1$ pontú fát kapunk. Ezt ismételve, $n-1$ -szer lehet elsőfokú csúcsot elhagyni a hozzá tartozó éllel együtt, mivel a végén már csak egyetlen csúcs marad \Rightarrow az eredeti fának $n-1$ éle volt.

\Leftarrow : Legyen F egy n pontú $n-1$ élű összefüggő gráf, továbbá legyen F' egy feszítőfája F -nek. Az előbb igazoltak szerint F' -nek is $n-1$ éle van $\Rightarrow F=F'$.

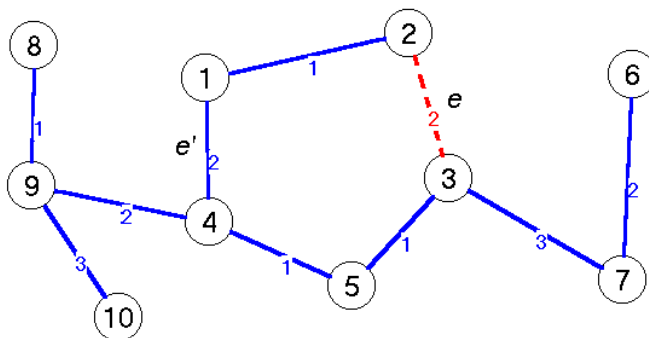
7.0.7. Állítás [2]: Egy fa bármely két pontja között **pontosan egy út** vezet.

Bizonyítás [3]: Indirekt tegyük fel, hogy u -ból v -be két út vezet, ekkor u -ból v -be elmegyek az egyik úton, majd visszajövök a másik úton, akkor legkésőbb u -ba jutva találunk egy olyan csúcsot, amely eleme az első útnak, tehát kört találtam.

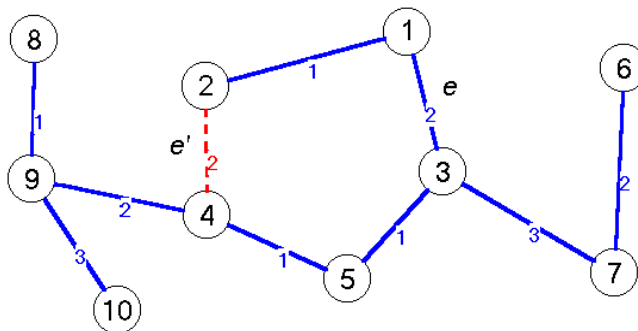
7.0.8. Állítás [2]: Legyen $G=(V,E)$ gráfnak $F=(V,E')$ egy minimális költségű feszítőfája, továbbá, legyen $e=[u,v]$ ($e \in E$) a G -nek egy olyan éle, ami nem éle F -nek ($e \notin E'$), és tegyük fel, hogy az F -beli u -ból v -be vezető úton van olyan e' él ($e' \in E'$), amelyre $c(e) \leq c(e')$. $\Rightarrow F$ -ből az e hozzá vételével és az e' elhagyásával kapott F' gráf is egy minimális költségű feszítőfája G -nek.

Bizonyítás [2]: Vegyük hozzá F -hez e élt, ekkor a kapott gráfban van olyan kör amelynek e' éle. Az e' törlésével kapott gráf tehát összefüggő marad és éleinek a száma is ugyanannyi, mint F éleinek a száma, így 7.0.6. szerint F' is feszítőfája G -nek. Továbbá $C(F') \leq C(F)$, mivel $c(e) \leq c(e')$, azaz egy nem nagyobb költségű éllel cseréltünk le egy élt.

Szemléltessük a 7.0.8. állítást egy példán. Legyenek $u=2$, $v=3$ csúcsok, továbbá $e=[2,3]$, $e'=[1,4]$ az állításban említett élek.



Az állítás szerint, ha e' él helyett e élt vesszük fel a feszítőfa éle közé, akkor áttérünk a G -nek egy másik minimális költségű feszítőfájára.



7.1 A piros-kék eljárás

„A fejezetben tárgyalt algoritmusok közös vonása, hogy valamilyen módszer szerint sorra veszik a gráf éleit, és egyes éleket bevesznek a kialakuló minimális költségű feszítőfába, másokat pedig nem. Ezen algoritmusok általánosításaként Robert E. Tarjan adott egy szép, **nem determinisztikus** eljárást, melyet **piros-kék eljárás**ként emlegetnek. A szemléletes tárgyalás érdekében az éleket szokás beszínezni, innen származik a módszer neve is. A módszer **kékre színezi** a minimális költségű feszítőfába **bekerülő élt**, és **pirosra színezi** azokat az éleket, amelyek már **biztosan nem kerülnek be a fába**. Az élek színezése során két szabályt fogunk alkalmazni a **piros szabályt** és a **kék szabályt**. A két szabályt **tetszőleges sorrendben és tetszőleges helyen** alkalmazhatjuk, akár véletlenített módon.” [2]

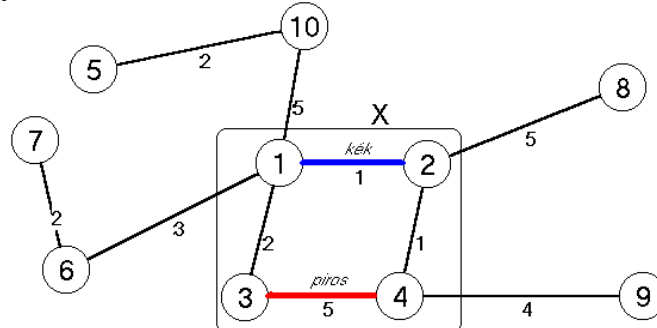
A később ismertetésre kerülő algoritmusokat (Prim, Kruskal) tekinthetjük úgy is, mint a piros-kék eljárás egy-egy specializált változatait.

7.1.1. Definíció [2]: Tekintsük a $G=(V,E)$ irányítatlan, súlyozott véges gráf éleinek egy színezését, amelynél egy él lehet piros, kék vagy színtelen. Ez a **színezés takaros**, ha létezik G -nek olyan minimális költségű feszítőfája, ami **az összes kék élt tartalmazza, de egyetlen piros élt sem tartalmaz.**

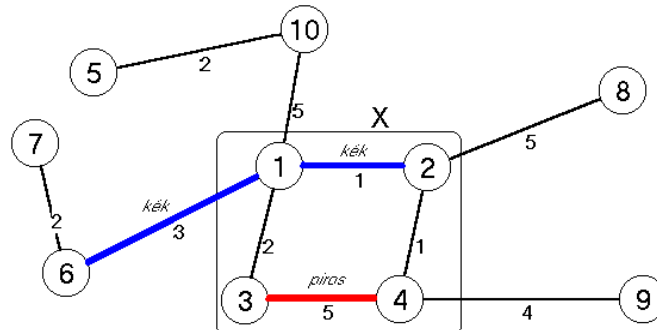
Kék szabály [2]:

Válasszunk ki egy olyan $\emptyset \neq X \subset V$ csúcshalmazt, amiből **nem vezet ki kék él**. Ezután egy **legkisebb súlyú X -ből kimenő színtelen élt** fessünk kékre.

Tekintsük az alábbi ábrán szereplő példán a kék szabály egy alkalmazását. Legyen $X=\{1,2,3,4\}$ halmaz. Látható, hogy X -ből nem vezet ki kék él.



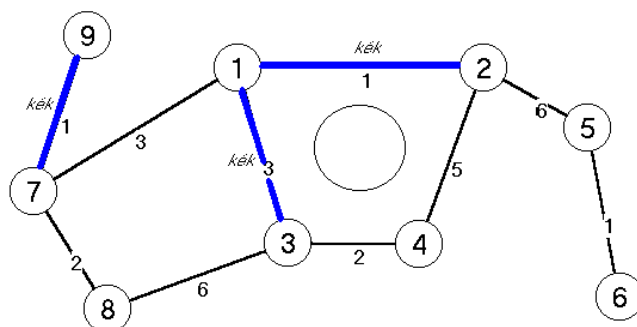
Színezzük kékre az X -ből "kivezető" egyik legkisebb súlyú élt, amely most a 3-as súlyú [1,6] él.



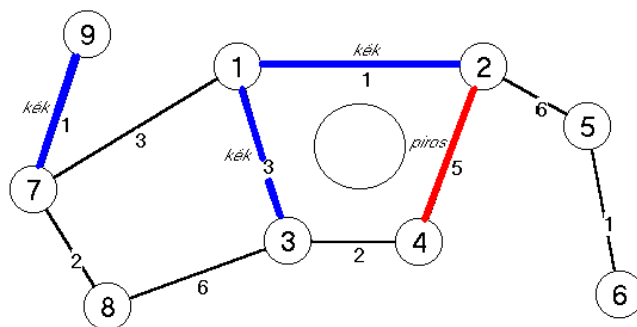
Piros szabály [2]:

Válasszunk G -ben egy olyan egyszerű kört, amiben nincs piros él. A kör egyik legnagyobb súlyú színtelen élet színezzük pirosra.

Most a piros szabály egy alkalmazását illusztráljuk az alábbi ábrán. A szabályban említett kör legyen $\langle 1,2,4,3 \rangle$, amely nem tartalmaz piros élt.



Keressük meg a kör egyik legnagyobb súlyú élet, amely az 5-ös súlyú $[2,4]$ él. Színezzük pirosra.



Piros-kék eljárás [2]:

Legyen kezdetben a $G=(V,E)$ irányítatlan, súlyozott, összefüggő, véges gráf minden éle színtelen. Alkalmazzunk a két szabályt tetszőleges sorrendben és helyen, amíg csak lehetséges.

7.1.2. Tétel [2]: Legyen $G=(V,E)$ irányítatlan, súlyozott, összefüggő, véges gráf, és $n = |V|$.

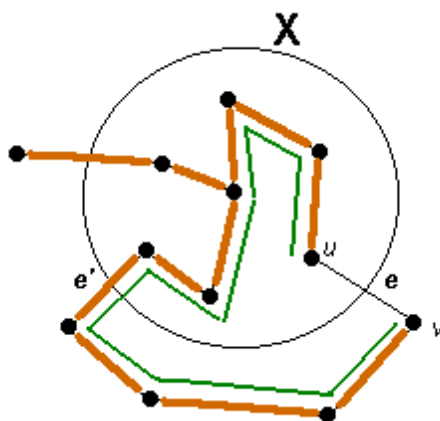
- I. A piros-kék eljárás során **a színezés mindig takaros** marad.
- II. A színezéssel **sosem akadunk el**, ameddig G minden éle színes nem lesz.
- III. Ha beszíneztük G minden élet, akkor a **kék élek** G egy **minimális költségű feszítőfájának éleit** adják, sőt már $n-1$ kékre színezett él után is megkaptuk az említett feszítőfát.

Bizonyítás [2]:

- I. **Teljes indukcióval** lássuk be az állítást. Kezdetben, amikor minden él színtelen nyilván teljesül a takaros színezés. Továbbiakban tegyük fel, hogy egy olyan állapotban vagyunk, amelyre teljesül a takaros színezés. Legyen F a G egy olyan minimális költség feszítőfája, amely az összes, jelenleg kékre színezett élt tartalmazza, és egyetlen, jelenleg pirosra színezett élt sem tartalmaz. Tegyük fel, hogy az eljárás következő lépése során az $e \in E$ élt színeztük be, ahol $e=[u,v]$. **Két eset lehet** attól függően, hogy melyik szabályt alkalmaztuk.

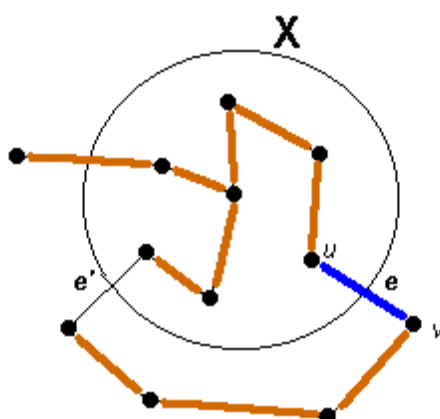
- 1) **A kék szabályt alkalmaztuk:** ekkor nyilván e színe kék lett.

- a) Ha e éle F -nek, akkor F mutatja ,hogy takaros a színezés.
- b) Ha e nem éle F -nek, akkor tekintsük az $X \subset V$ halmazt, amire a kék szabályt alkalmaztuk. Az F -ben $\exists u \rightsquigarrow v$ út, hiszen F feszítőfa (7.0.7. állítás), továbbá ezen az úton van olyan e' él, ami kimegy X -ből (Ugyanis e -t színeztük kékre, tehát a kék szabály értelmében e egyik vége X -en belül, a másik vége X -en kívül van. Továbbá az említett $u \rightsquigarrow v$ F -beli út, egy X -beli és egy X -en kívüli pontot köt össze, tehát valahol ki kell lépnie X -ből.).



Az ábrán vastagabb vonallal jelöltük az F éleket, és segédvonallal az F -beli $u \rightsquigarrow v$ utat.

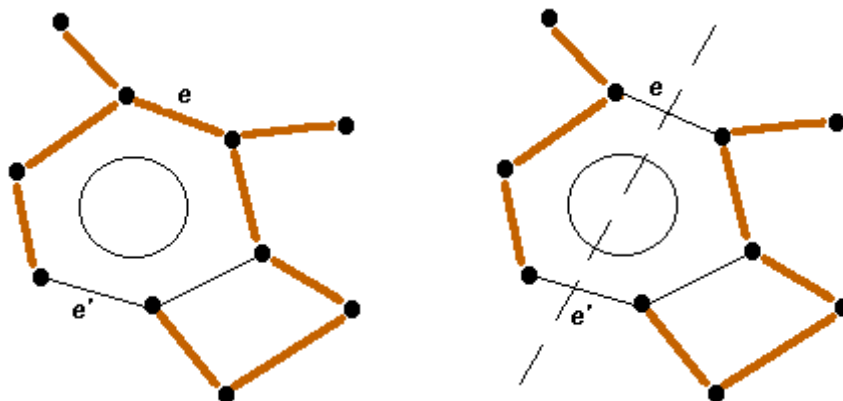
Vizsgáljuk, milyen lehet e' színe! Piros nem lehet, mivel része F -nek, kék sem lehet, mivel a kék szabályt alkalmaztuk, amely szerint X -nek olyannak kell lennie, amiből nem vezet ki kék él. Tehát e' szintelen. Továbbá $c(e) \leq c(e')$, mivel a kék szabály szerint az X -ből kimenő egyik legkisebb súlyú élt kell választani, és mi e -t választottuk. Alkalmazhatjuk a 7.0.8. állítást, mely szerint F -ből e' törlésével és e hozzá vételével kapott új F' gráf is a G egy minimális költségű feszítőfája. Tehát F' igazolja, hogy e kékre színezésével a színezés továbbra is takaros marad.



Az ábrán vastagabb vonallal kiemeltük a másik minimális költségű feszítőfát, F' éleit.

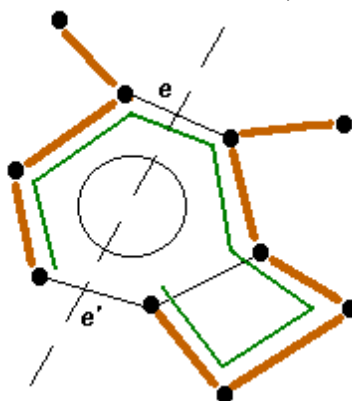
- 2) **A piros szabályt alkalmaztuk:** ekkor nyilván e színe piros lett.
 - a) Ha e nem éle F -nek, akkor F mutatja ,hogy takaros a színezés.
 - b) Ha e éle F -nek, akkor a pirosra színezés azt jelenti, hogy e továbbiakban már nem lehet éle az eljárás során előállítás alatt lévő minimális feszítőfának, tehát a takaros színezés bizonyításához át kell térni egy másik minimális feszítőfára. Az e F -ből való törlésével F két komponensre esik szét. Tekintsük azt a kört, amelyre a piros szabályt alkalmaztuk, ennek van olyan e' éle, amelyik a két

komponenst összeköti és nem éle F -nek (Ugyanis a két komponenst összekötő e -től különböző élnek lennie kell, mivel kör mentén vizsgálódunk, és egy körbeli él elhagyásával az összefüggőség nem szűnhet meg. Továbbá, ha nem lenne ilyen e' él, ami nem éle F -nek, az azt jelenti, hogy a kör minden éle F éle is, tehát kör lenne a fában.).



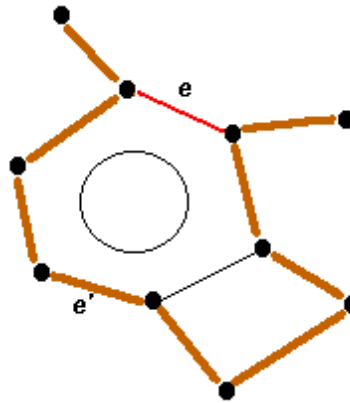
Az ábrákon kiemeltük F éleit, majd illusztráltuk e elhagyása után keletkező két komponenst.

Vizsgáljuk, milyen lehet e' színe! Nem lehet kék, mivel nem éle F -nek és feltettük, hogy a színezés takaros, amit F mutat. Nem lehet piros, mivel a piros szabály értelmében, olyan kört kell választani, amiben nincs piros él. Tehát e' színtelen. Továbbá $c(e') \leq c(e)$, mivel a piros szabály alkalmazása során e -t választottuk színezésre, amely szerint a kör egyik legnagyobb súlyú élet kell pirosra színezni. Az e' végpontjait összekötő F -beli út tartalmazza e élet. (Ugyanis e törlése előtt F feszítőfa volt, és 7.0.7. állítás szerint, bármely két pontja között pontosan egy út vezet. Azonban most két olyan részre esett szét, amelynek egyik komponensében van e' egyik vége, a másik komponensében e' másik vége. F -ben a két komponens között az átjárást éppen az e él biztosította, tehát az említett útnak át kell haladnia az e élen.)



Az ábrán segédvonallal berajzoltuk az e' két végpontját összekötő F -beli utat.

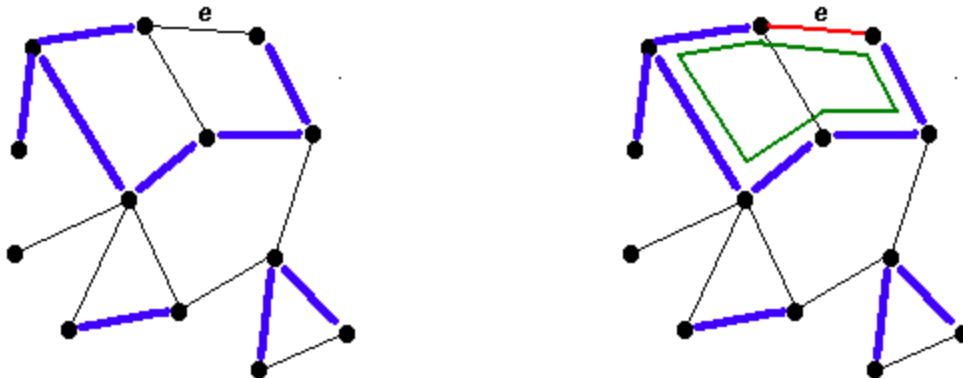
Alkalmazhatjuk a 7.0.8. állítást, mely szerint F -ből e törlésével és e' hozzá vételével kapott új F' gráf is a G egy minimális költségű feszítőfája. Tehát F' igazolja, hogy e pirosra színezésével a színezés továbbra is takaros marad.



Az ábrán vastagabb vonallal kiemeltük a másik minimális költségű feszítőfa, F' éleit.

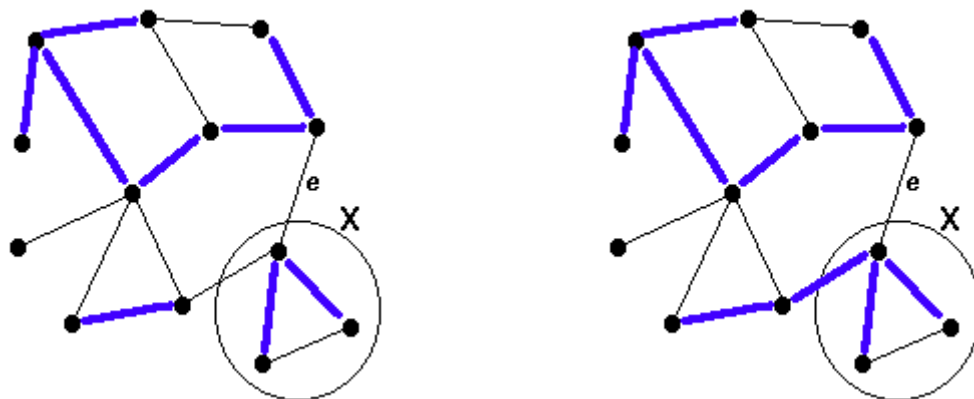
II. Most belátjuk, hogy a színezéssel **sosem akadunk el**, ameddig G minden éle színes nem lesz. Tegyük fel, hogy G -nek még nem minden éle színes. Legyen e egy színtelen él. A színezés takarossága miatt a kék élek egy erdőt alkotnak (de lehet, hogy már egy fát, ekkor az alábbi 1. eset alkalmazható), az **erdő fáit nevezzük kék fáknak**. Két eset lehetséges:

- 1) **Ha e két végpontja ugyanabban a kék fában van.** Ekkor a piros szabályt alkalmazhatjuk arra a körre, aminek az éleit úgy kapjuk, hogy az e két végpontját összekötő egyetlen (7.0.7. állítás) kék úthoz hozzávesszük e -t.



Az ábrákon vastagabb vonallal jelöltük a kék fákat, segédvonallal az említett kört.

- 2) **Ha e két végpontja különböző kék fában van.** Ekkor a kék szabály alkalmazható a következőképpen: X legyen az egyik olyan kék fa csúcsainak halmaza, amelyekben benne van e egyik vége. Ebből a kék fából (X -ből) biztosan megy ki él (legalább e), ezen kimenő élek közül, az egyik legkisebb súlyú (nem biztos, hogy e) kékre színezhető.



Az ábrákon jelöltük a kék fákat, az X halmazt, és végül a két kék fát összekötő, új kék élt.

- III. A harmadik állítás szerint, **végül megkapjuk G egy minimális költségű feszítőfáját.** Ez rögtön következik abból, hogy a végső színezés is takaros. Az állítás második része szerint, az eljárást **elegendő addig folytatni, míg $n-1$ kék él nem lesz.** A 7.0.6. állítás szerint, a feszítőfának összesen $n-1$ éle van, tehát ha már van $n-1$ kék élünk, akkor a továbbiakban több nem is keletkezhet.

Tehát a piros és kék szabályt tetszőleges helyen és sorrendben alkalmazva, végül minimális költségű feszítőfát kapunk, azonban **hatékonysági szempontból** megfontolandó melyik szabályt mikor és hol alkalmazzuk. A következő algoritmusokat a piros-kék eljárás egy-egy speciális esetének is tekinthetjük.

7.2 Prim algoritmus

Az algoritmus elve [2]: A Prim algoritmus **minden lépésben a kék szabályt alkalmazza egy s kezdőcsúcsból kiindulva.** Az algoritmus működése során egyetlen kék fát tartunk nyilván, amely folyamatosan növekszik, míg végül minimális költségű feszítőfa nem lesz. Kezdetben a kék fa egyetlen csúcsból áll, a kezdőcsúcsból, majd minden lépés során, a kék fát tekintve a kék szabályban szereplő X halmaznak, megkeressük az egyik legkisebb súlyú élt (mohó stratégia), amelynek egyik vége eleme a kék fának (X -ben van), a másik vége viszont nem (nem eleme X -nek). Az említett élt hozzá vesszük a kék fához, azaz az élt kékre színezzük, és az él X -en kívüli csúcsát hozzávesszük az X -hez.

Az algoritmus **ADT szintű** leírása [5]:

Az algoritmus megvalósításának a kulcsa az X -ből kimenő egyik legkisebb súlyú él meghatározása. Ehhez használjunk egy **minimum választó elsőbbségi (prioritáson) sort** ($\min Q$), amelyben a fához még nem tartozó (még nem eleme X -nek) csúcsokat tároljuk **az X -től való távolsággal**, mint **kulcs** értékkel. A távolság elnevezéséből adódóan és a korábbi algoritmusokhoz hasonlóan, jelöljük a kulcsot egy $v \in V$ csúcs esetén $d[v]$ -vel.

Egy $v \in V$ csúcs esetén az **X -től való távolság**, azaz a $d[v]$ legyen azon élek közül a minimális súlyú él súlya, amely v és egy X -beli csúcs között halad. Amennyiben nem létezik él v és egy tetszőleges X -beli csúcs között, legyen $d[v] = \infty$.

A korábbi algoritmusokhoz hasonlóan, a $P[1..n]$ tömbbe tároljuk el egy csúcs **feszítőfabeli megelőzőjét (szülőjét)**, amelynek segítségével bejárható a fa.

Az algoritmus elvénél, azt mondtuk, hogy kezdetben a kék fa legyen egyetlen pont, a kezdőcsúcs. Most az X -től való távolság fogalmának bevezetésével, azt mondhatjuk, hogy kezdetben X legyen az üres halmaz, amelytől a kezdőcsúcs nulla távolságra van, az összes többi

csúcs pedig végtelen távolságra. Az algoritmus leírásában az X halmazt explicite nem ábrázoljuk, hanem $X = V \setminus \min Q$.

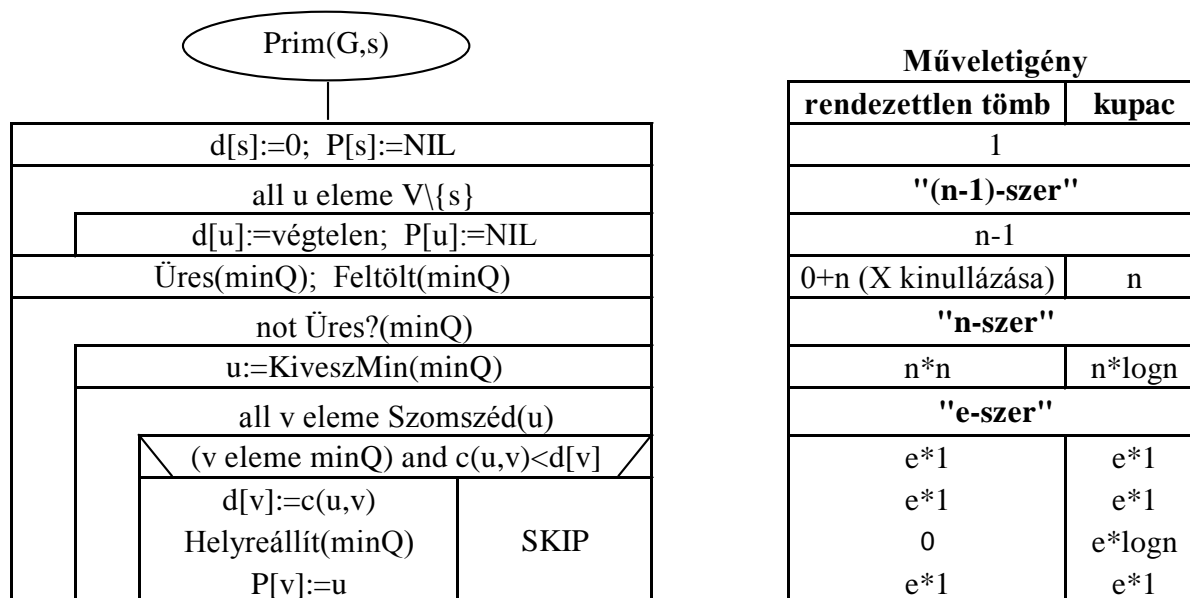
Az **algoritmus minden lépésében** kivesszük a $\min Q$ (egyik) legkisebb kulcsú elemét (az X -ből kimenő egyik legkisebb súlyú él X -en kívüli csúcsát), azaz a készülő feszítőfához, X -hez hozzávesszük az illető csúcsot. Majd az X -en kívüli csúcsok **X -től való távolságát, mint invariáns tulajdonságot karban kell tartani**. Nyilván elegendő az X -be újonnan bekerült csúcs szomszédainak az X -től való távolságát módosítani (ha szükséges), mivel egy v csúcs úgy kerülhet közelebb X -hez, hogy valamelyik u szomszédja bekerül az X -be. Ekkor v távolsága a következőképpen alakul:

- ha $d[v] = \infty$, akkor most legyen $d[v] = c(u, v)$
- ha $d[v] < \infty$, akkor már létezik v -nek olyan w szomszédja, amely eleme X -nek tehát $d[v]$ akkor változik, ha az $[u, v]$ élen keresztül v közelebb van X -hez, mint $[v, w]$ él esetén.

Eközben a $P[1..n]$ **szülőségi tömböt is karban kell tartani**.

Tehát a használt típusok és adatszerkezetek:

- $P[1..n]$ tömb: egy csúcs feszítőfabeli szülőcsúcsának a tárolására.
- $\min Q$: $(d[v], v)$ párokból álló minimumválasztó elsőbbségi sor, ahol $d[v]$ értéke a kulcs.



Az algoritmus megvalósítása az **ábrázolás szintjén [5]**:

Vizsgáljuk meg a prioritásos sor ($\min Q$) megvalósításának két, természetes módon adódó lehetőségét, ahogy a Dijkstra algoritmusnál is már láttuk:

- 1) A prioritásos sort valósítsuk meg **rendezetlen tömbbel**, azaz a prioritásos sor legyen maga a $d[1..n]$ tömb. Ekkor a minimum kiválasztására egy **feltételes minimum keresést** kell alkalmazni, amelynek a műveletigénye $\Theta(n)$. A $Feltölt(\min Q)$ és a $Helyreállít(\min Q)$ absztrakt műveletek megvalósítása pedig egy SKIP-pel történik.

Az algoritmus ADT leírásában az szerepel, hogy a $\min Q$ -ból kiveszünk egy elemet, azonban a $\min Q$ -t egy tömbbel valósítjuk meg, amelynek a mérete nem változik. Tehát osztályozni kell a csúcsokat aszerint, hogy a $\min Q$ -ban vannak-e még, vagy már bekerültek az X halmazba. Legyen egy $X[1..n]$ tömb az alábbi módon definiálva:

$$X[i] = \begin{cases} 0 & , \text{ha } i \notin X \\ 1 & , \text{ha } i \in X \end{cases} . \text{ Az } X \text{ tömböt kezdetben ki kell nullázni, majd menet közben karban}$$

kell tartani. Amint kikerül egy csúcs a $\min Q$ -ból, az X tömbben a csúcsnak megfelelő helyre 1-est kell írni.

- 2) **Kupac adatszerkezet** használatával is reprezentálhatjuk a prioritásos sort. Ekkor a *Feltölt(minQ)* eljárás, egy kezdeti kupacot épít, amelynek a műveletigénye **lineáris** (lásd Heap-Sort). Azonban most a $d[1..n]$ tömb változása esetén a kupacot is karban kell tartani, mivel a kulcs érték változik. Ezt a *Helyreállít(minQ)* eljárás teszi meg, amely a csúcsot a gyökér felé "szivárogtatja" fel, ha szükséges (mivel a kulcs értékek csak csökkenhetnek). Ennek a műveletigénye $\log n$ -es. Ennél az ábrázolásnál is vezessünk be egy segéd tömböt, a $HOL[1..n]$ tömböt, amely megmutatja, hogy egy csúcs hol helyezkedik el a kupacban (a kupacot $[1..2n]$ tömbben valósítjuk meg), illetve legyen 0, ha az illető csúcs már nem eleme a $\min Q$ -nak. A HOL tömb felhasználásával egy csúcs prioritásos sorban való keresésének műveletigényét konstansra csökkenthetjük. A HOL tömböt a $\min Q$ változásakor szintén karban kell tartani.

Megjegyzés: Nem szükséges kezdeti kupacot építeni, felesleges a kupacba rakni a végtelen távolságú elemeket. Kezdetben csak a kezdőcsúcs legyen a kupacban, majd amikor először „elérünk” egy csúcsot és a távolsága már nem végtelen, elég akkor berakni a kupacba.

Tehát a prioritásos sor fenti két megvalósítása esetén, a következőképpen alakul az **algoritmus műveletigénye [1]:**

A struktogramm mellett feltüntettük az egyes műveletek költségét a két ábrázolás esetén. A belső ciklust célszerű globálisan kezelni, ekkor mondható, hogy összesen legfeljebb annyiszor fut le, ahány éle van a gráfnak.

I. Tehát **rendezetlen tömb** esetén:

$$T(n) = O(1 + n - 1 + n + n^2 + 3 * e) = O(n^2 + e) = O(n^2)$$

II. **Kupac** esetén:

$$T(n) = O(1 + n - 1 + n + n * \log n + 3 * e + e * \log n) = O((n + e) * \log n)$$

Következmény az ábrázolásra [5]:

A Dijkstra algoritmusnál már említett következmény itt is érvényes, azaz

Sűrű gráf esetén: csúcsmátrix + rendezetlen tömb.

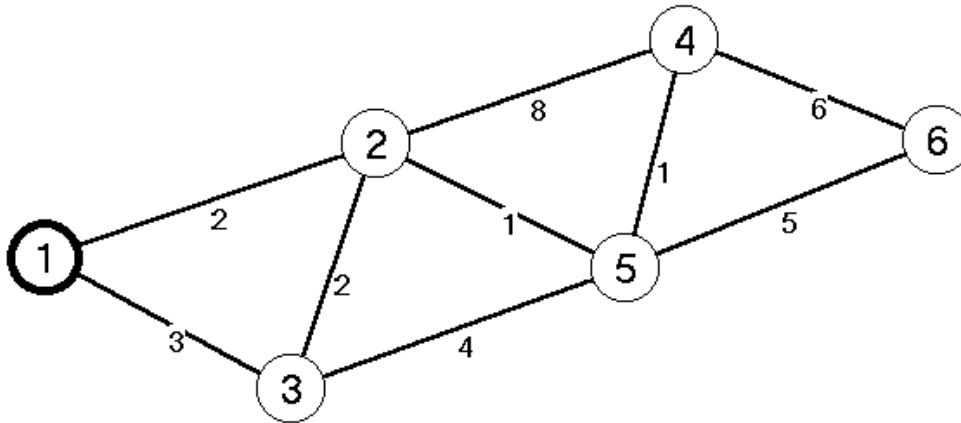
Ritka gráf esetén: éllista + kupac.

Most pedig nézzük meg egy példán, **ADS szinten** az algoritmus működését:

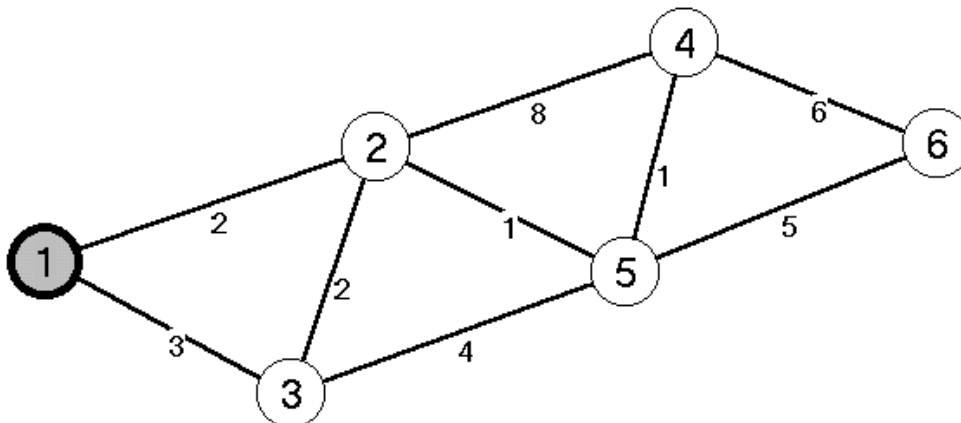
A szemléltetés érdekében színezzük a csúcsokat a következőképpen:

- **Fehér:** a csúcs eleme a $minQ$ -nak és nincs X -beli szomszédja, azaz még nem került "látótávolságba", tehát az X -től való távolsága végtelen.
- **Szürke:** a csúcs eleme a $minQ$ -nak, de létezik X -beli szomszédja, tehát a távolsága már kisebb, mint végtelen.
- **Fekete:** a csúcs kikerült a $minQ$ -ból, azaz bekerült X -be

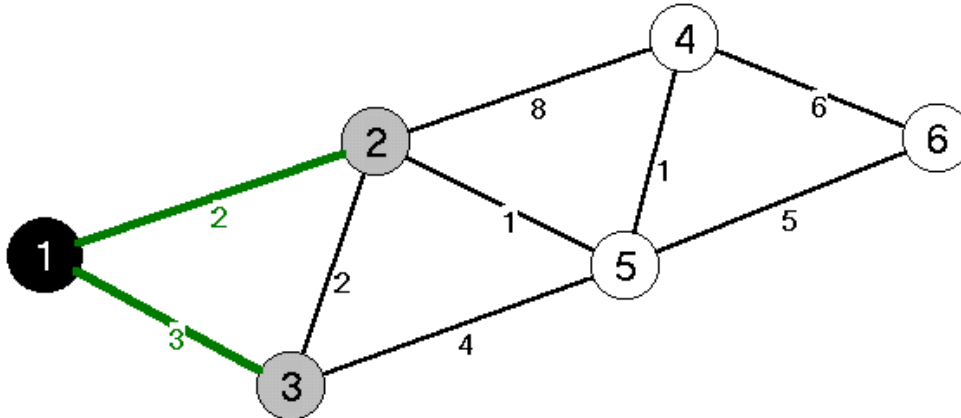
A példában a kezdőcsúcs legyen az 1-es csúcs.



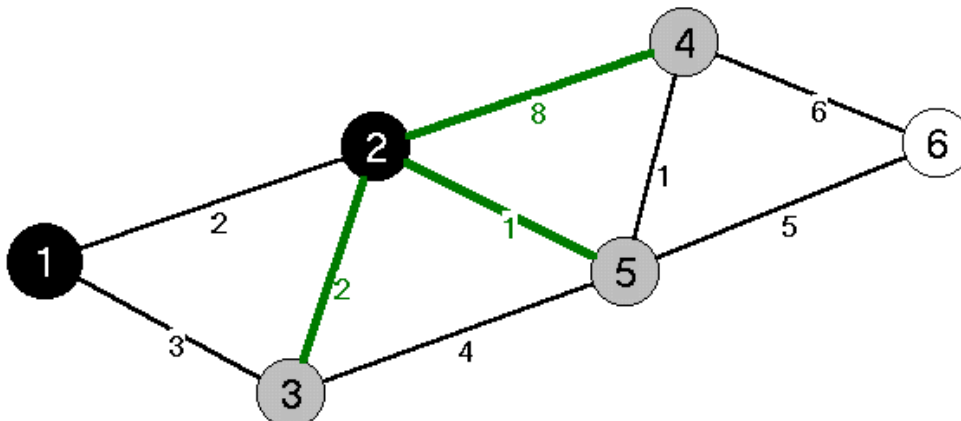
Az inicializáló lépés után az 1-es csúcs kivételével minden csúcs távolsága (az X halmaztól) legyen végtelen, az 1-es csúcs távolsága pedig legyen 0, az X legyen az üres halmaz.



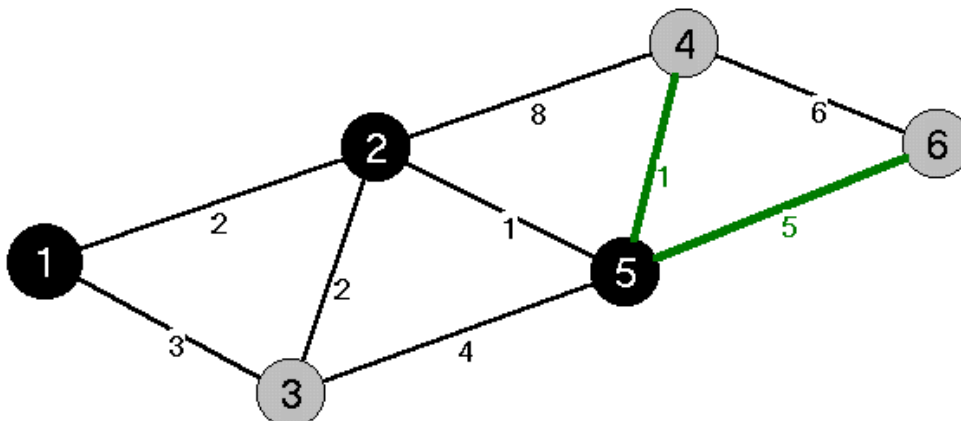
Az első lépésben kivesszük a $\min Q$ -ból az 1-es csúcsot (mivel az 1-es csúcs távolsága a legkisebb az X -től), tehát $X=\{1\}$, majd az 1-es csúcs szomszédai (2 és 3) kerülnek közelebb az X -hez. Ezek távolsága $d[2]=2$ és $d[3]=3$.



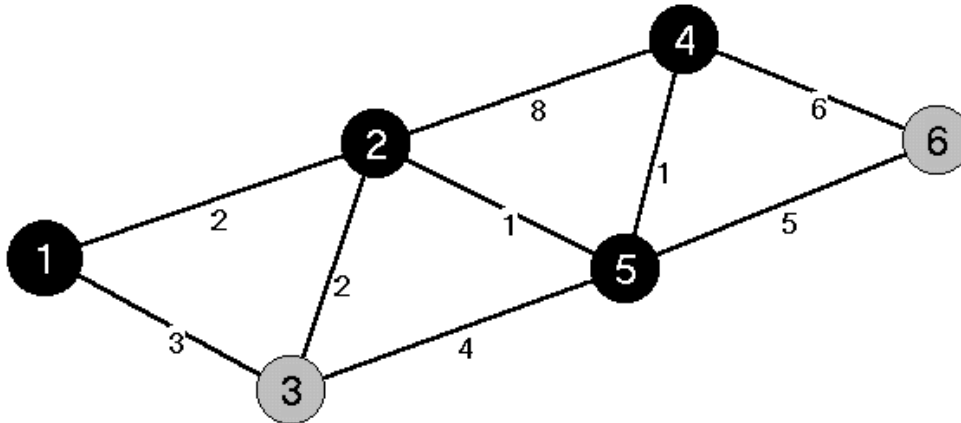
A második lépésben a 2-es csúcs kerül be az X halmazba (feljegyezve az 1-es csúcsot, mint fabeli szülőt), mivel közelebb van X -hez, mint a 3-as csúcs. Ezután a 2-es szomszédai kerülnek "látótávolságba". Megfigyelhető, hogy a 3-as csúcs $d[3]=3$ távolságra volt az X -től, de most közelebb került $d[2]=2$, a $[2,3]$ él figyelembe vételével. Az $X=\{1,2\}$.



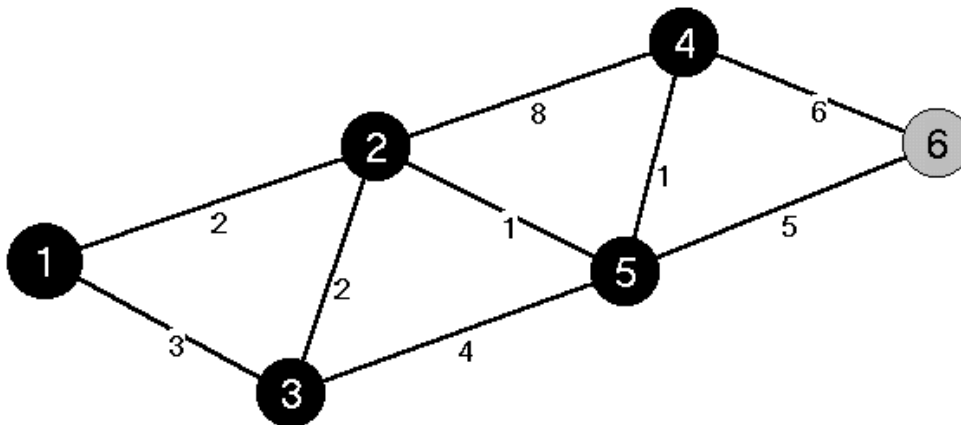
A harmadik lépésben az $X=\{1,2\}$ halmazhoz a legközelebb lévő csúcs ($d[3]=2$, $d[4]=8$, $d[5]=1$), az 5-ös csúcs kerül az X halmazba (feljegyezve szülőként a 2-es csúcsot). Az 5-ös (még X -hez nem tartozó) szomszédai, a 4-es és 6-os csúcsok kerülnek közelebb az X -hez.



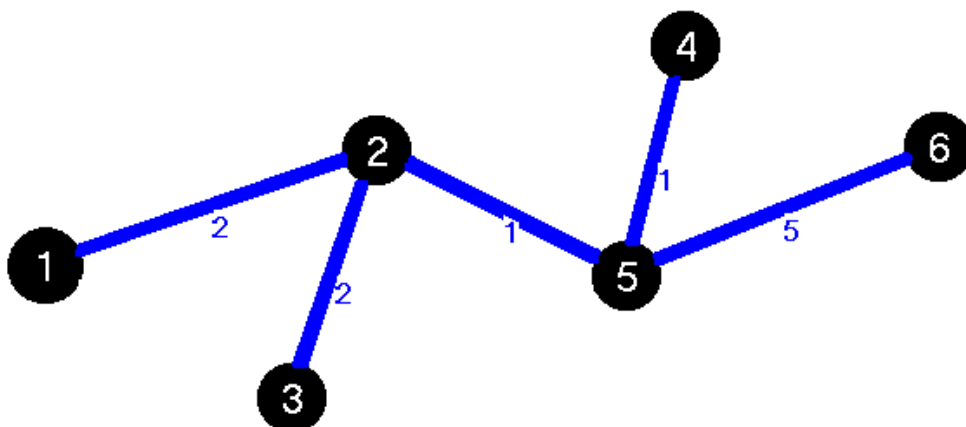
A negyedik lépésben a nem fekete csúcsok közül a legkisebb távolságú, a 4-es csúcs kerül az X -be. $X=\{1,2,4,5\}$. A 4-es szomszédai, a "4-esen keresztül" már nem kerülnek X -hez közelebb.



Az ötödik lépésben a 3-as csúcs kerül az X -be. A 3-asnak már nincs is nem fekete (nem X -beli) szomszédja.



Végül az utolsó lépésben a 6-os csúcs kerül az X halmazba. A menetközben feljegyzett szülőcsúcsok segítségével meghatározható a feszítőfa.



7.3 Kruskal algoritmus

Az algoritmus elve [2]: Kezdetben legyen n db kék fa, azaz a gráf minden csúcsa egy-egy (egy pontból álló) **kék fa**, és legyen minden él színtelen. Minden lépés során *kiválasztjuk az egyik legkisebb súlyú színtelen élt*. Ha a kiválasztott él **két végpontja különböző kék fában van, akkor színezzük kékre, különben** (az él két vége azonos kék fában van, tehát a kék fa éleivel kört alkot) **színezzük pirosra**. A fentiekből kitűnik, hogy a Kruskal algoritmust is tekinthetjük a piros-kék eljárás egy speciális esetének, ahol az élek színezésének a sorrendje egyfajta mohó stratégia szerint történik ("még mohóbb", mint a Prim algoritmusnál).

Ugyanis:

- Amikor egy e **élt pirosra színezzük**, akkor arra az egyszerű körre alkalmazható a piros szabály, amelynek élei az e , és az e két végpontját összekötő kék út élei. Ez egy egyszerű kör, mivel pontosan egy e végpontjait összekötő kék út létezik, továbbá az e kivételével, minden él kék, tehát e színezése előtt nem tartalmazott piros élt. Így teljesülnek a piros szabály feltételei.
- Amikor egy e **élt kékre színezzük**, akkor e két kék fát köt össze, F_1 -et és F_2 -öt. A kék fák definíciójából következik, hogy F_1 csúcsainak halmazából nem vezet ki kék él. Legyen $X = \{F_1 \text{ csúcsainak a halmaza}\}$, ekkor az e él lesz az egyik legkisebb súlyú X -ből kimenő színtelen él, mivel e az egyik legkisebb súlyú (nem csak X -ből kimenő) színtelen él. Tehát teljesülnek a kék szabály feltételei.

Az algoritmusnak egy fontos tulajdonsága, hogy amennyiben a gráf nem összefüggő, úgy egy **minimális költségű feszítőerdőt** határoz meg.

Az algoritmus **ADT szintű** leírása:

Az algoritmus absztrakt szintjén, **diszjunkt halmazokkal való műveleteket** fogunk végezni. Tekintsük a kék fák csúcsainak (diszjunkt) halmazait (ezek a halmazok osztályozzák V -t). Amikor az egyik legkisebb súlyú színtelen élt kiválasztjuk, el kell döntenünk, hogy a két végpontja azonos vagy különböző halmazban vannak-e. Majd a választól függően:

- Ha **azonos halmazban vannak**, akkor a kiválasztott élt színezzük pirosra.
- Ha **különböző halmazban vannak**, akkor a kiválasztott élt színezzük kékre, és a két különböző halmazt vonjuk össze, azaz a két halmaz helyett legyen egy halmaz, amely megegyezik a két halmaz uniójával.

Az algoritmust akkor áll le, ha már nincs színtelen él (leállhatna már akkor is, ha az előbb következne be, hogy beszínezett $n-1$ db kék élt). Mivel véges sok élünk van, és minden lépésben beszínezünk egyet, így $|E|$ lépés után az algoritmus biztosan befejezi a működését.

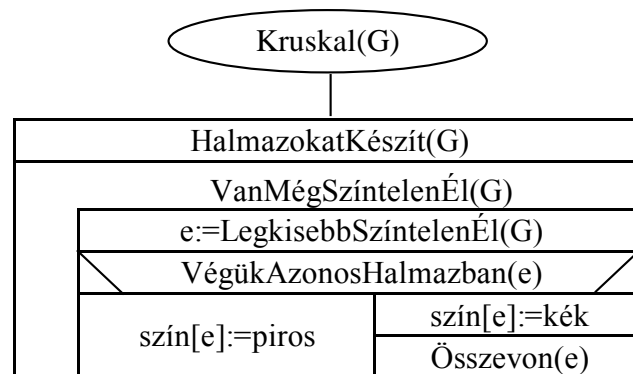
Nézzük, milyen **absztrakt műveleteket** fogunk használni:

Eljárások:

- *HalmazokatKészít(G)*: Elkészíti a kezdeti n db, pontosan egy csúcsot tartalmazó diszjunkt halmazokat.
- *Összevon(e)*: Az e él két végpontja által reprezentált halmazokat összevonja.
- $\text{szín}[e] := \dots$: Az e él színét változtatja meg az értékadás jobb oldalán szereplő színre.

Függvények:

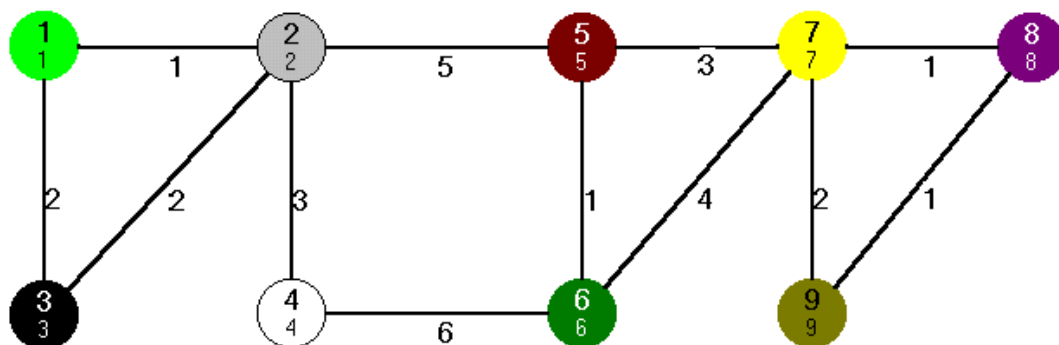
- $\text{VanMégSzingtlenÉl}(G) = \begin{cases} \text{igaz} & , \text{ha } G - \text{nek van még szingtlen él} \\ \text{hamis} & , \text{különben} \end{cases}$
- $\text{VégükAzonosHalmazban}(e) = \begin{cases} \text{igaz} & , \text{ha } e \text{ két végpon tja azonos halmazban van} \\ \text{hamis} & , \text{különben} \end{cases}$
- *LegkisebbSzingtlenÉl(G)*: Visszaadja a legkisebb súlyú szingtlen élt.



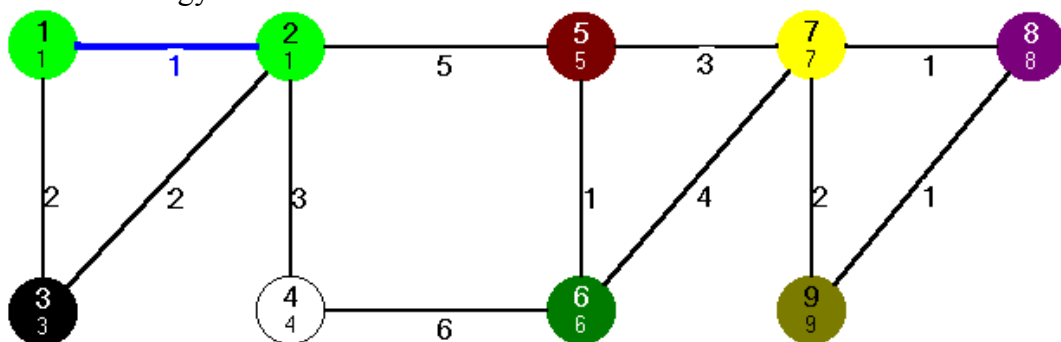
Most pedig nézzük meg egy példán, **ADS szinten** az algoritmus működését:

A példában a csúcsok osztályokhoz (halmazokhoz/kék fához) való tartozását **színezéssel** illetve **címkézéssel** oldottuk meg. Az azonos színű csúcsok, azonos osztályba tartoznak. Tudjuk, hogy az **osztályok reprezentálhatók egy-egy elemükkel**, ezért az ábrán (a csúcs címkéje alatt), feltüntettük azon osztály egy reprezentáló elemének a címkéjét, amelyhez az illető csúcs tartozik. Tehát azok a csúcsok tartoznak egy osztályba (azonos kék fához), amelyeknél a címkéjük alatt megjelenő, méretét tekintve kisebb szám azonos.

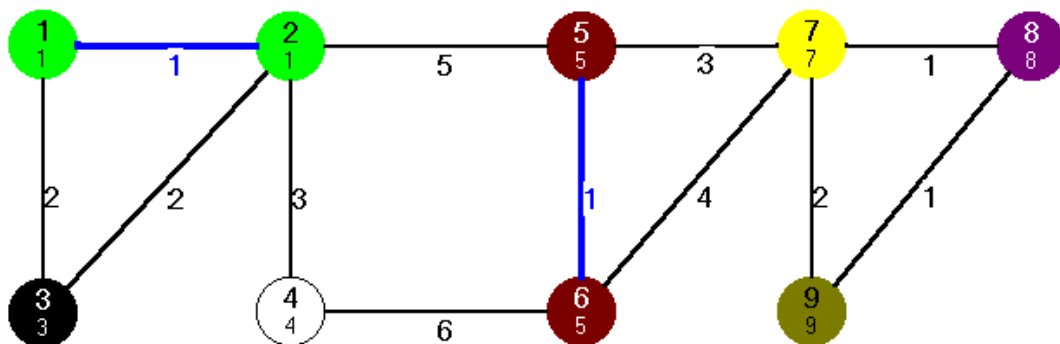
Az inicializáló lépés után, minden él szingtlen és minden csúcs külön osztályt alkot.



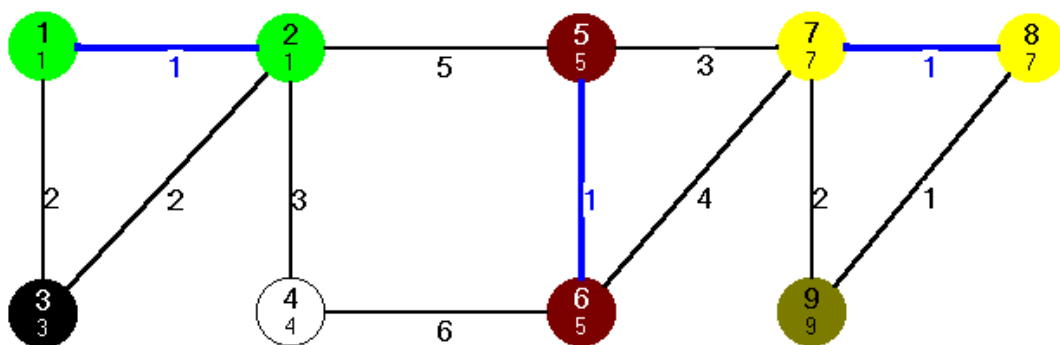
Az első lépésben kiválasztjuk az egyik legkisebb súlyú élt ($[1,2]$), és az 1-es ill. 2-es csúcsokat tartalmazó (egyelemű) halmazokat összevonjuk egyetlen $H=\{1,2\}$ halmazzá. Az új halmaz reprezentáns eleme legyen az 1-es csúcs.



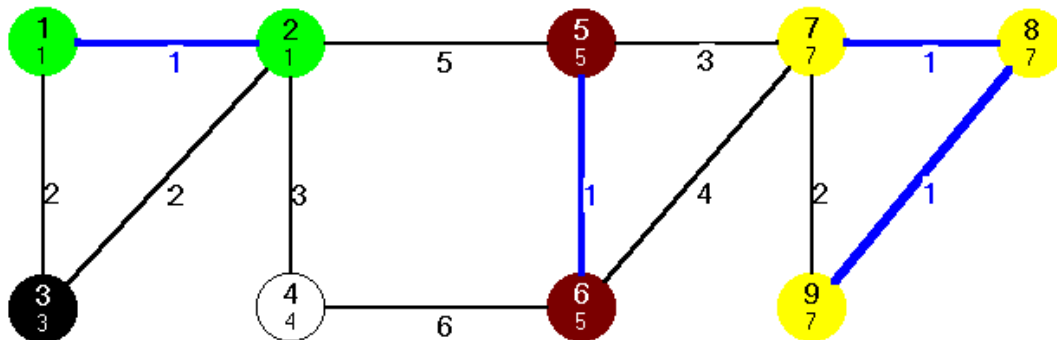
A következő lépésben, az első lépéshez hasonlóan járunk el az 5-ös és 6-os csúcsokkal:



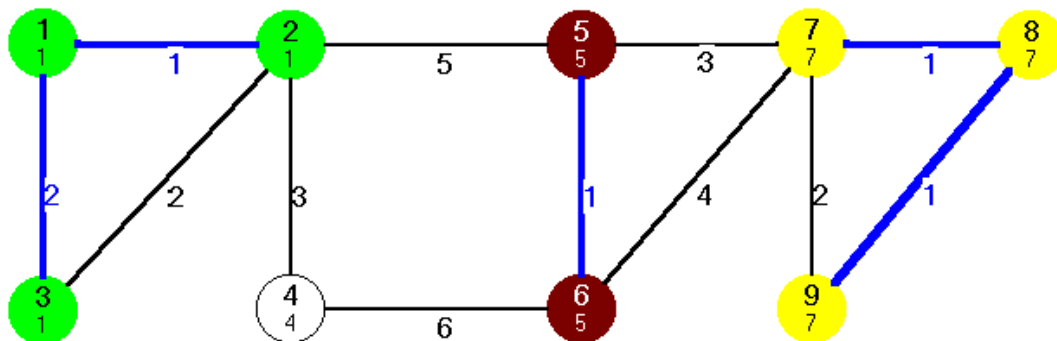
A harmadik lépésben, még mindig egyelemű halmazokat vonunk össze, most a 7-es és 8-as csúcsok osztályait.



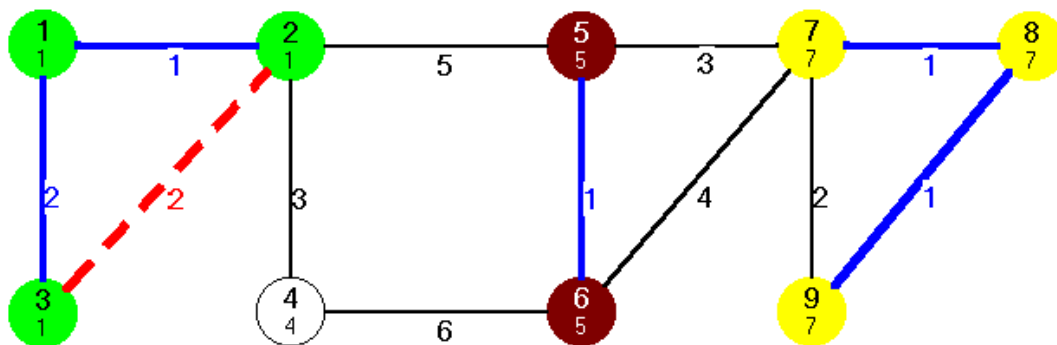
A negyedik lépésben a kiválasztásra kerülő [8,9]-es 1-es súlyú él, még mindig két különböző kék fát köt össze, így kékre kell színezní és a $H_1=\{7,8\}$ és $H_2=\{9\}$ halmazokat össze kell vonni a $H=\{7,8,9\}$ halmazzá.



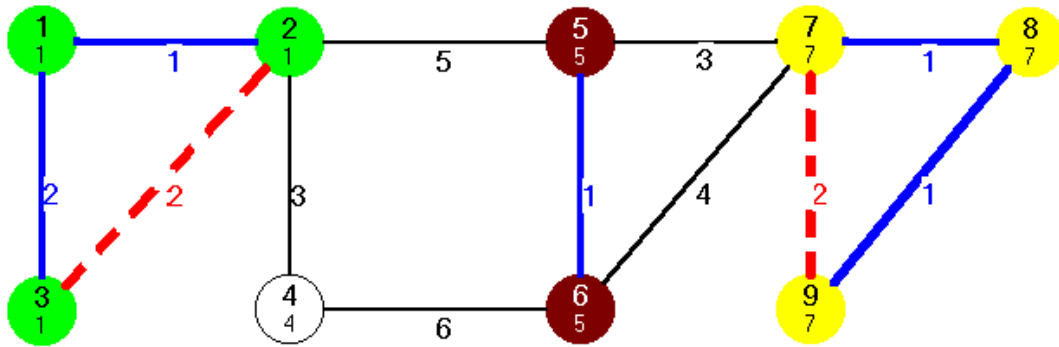
Az ötödik lépésben már nincs 1-es súlyú él. A következő egyik legkisebb súlyú él, valamelyik 2-es súlyú él lesz. Mi most válasszuk az [1,3]-as élt, amelyet kékre színezzük, és a végpontjainak megfelelő halmazokat összevonjuk.



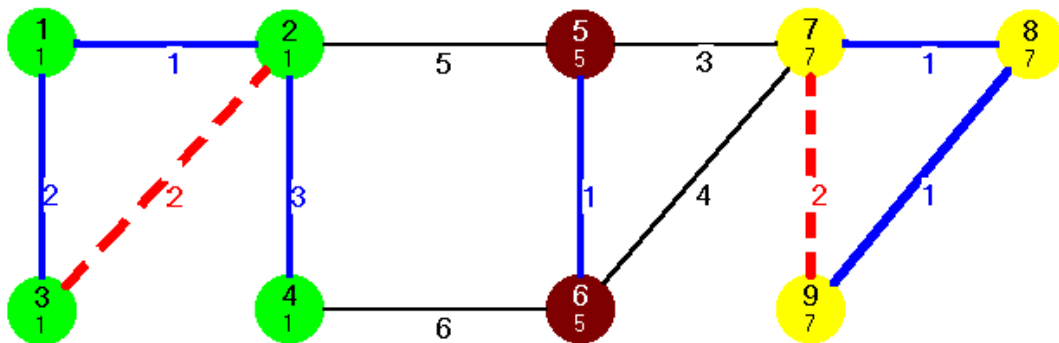
A hatodik lépésben kiválasztott [2,3]-as él két végpontja azonos kék fához tartozik, ezért színezzük pirosra.



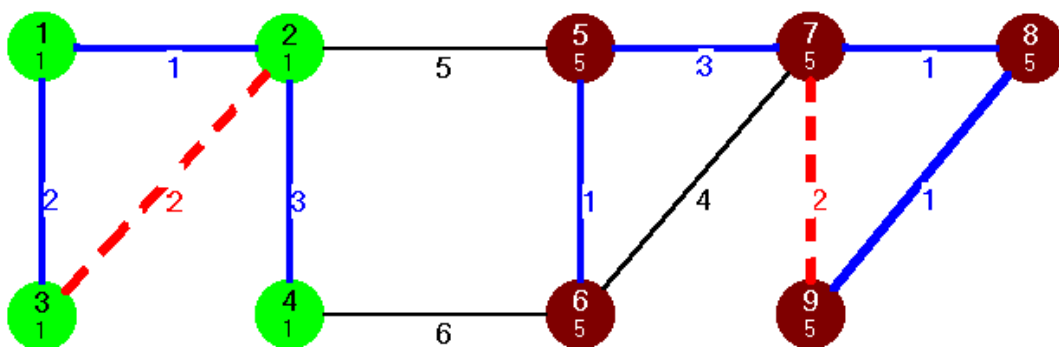
A hetedik lépésben ismét a piros szabályt alkalmazzuk, most a $\langle 7,8,9 \rangle$ körre, amelynek következtében a $[7,9]$ -es él piros lesz.



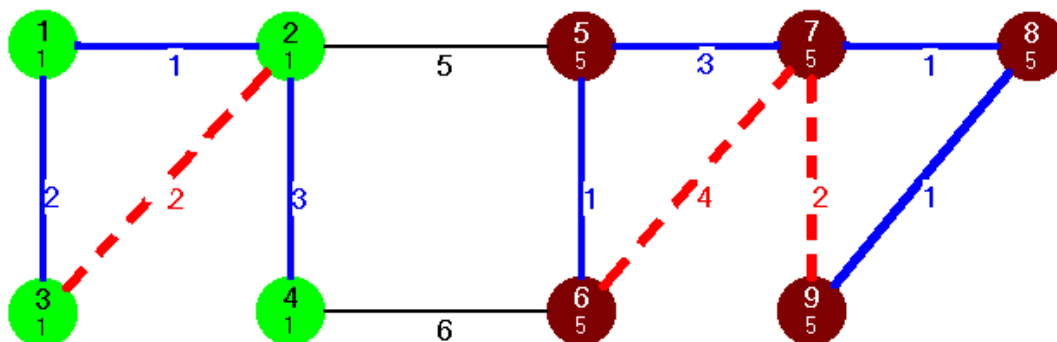
A nyolcadik lépésben a $[2,4]$ -es élt választjuk ki, és a kék szabályt alkalmazhatjuk az $X=\{1,2,3\}$ halmazra. Tehát a $[2,4]$ -es élt kékre színezzük, aminek következtében azonos kék fába kerülnek az $\{1,2,3,4\}$ -es csúcsok. A Kruskal algoritmusnak megfelelően, a kék fák nyilvántartására, vonjuk össze őket egy halmazba!



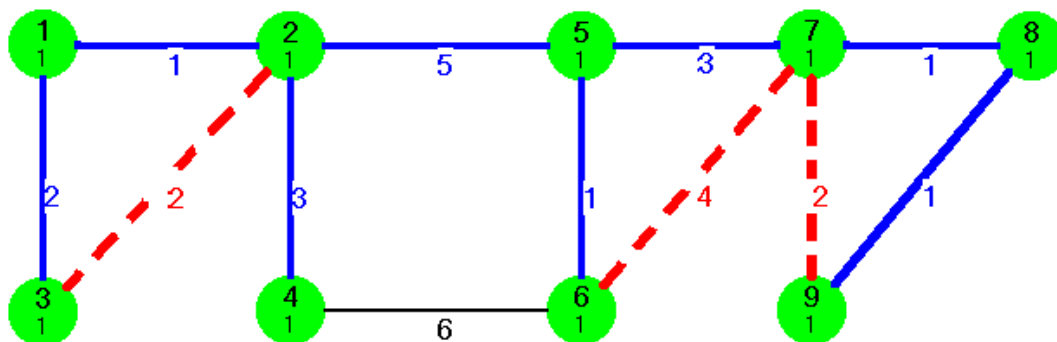
A kilencedik lépésben mindenképpen az $[5,7]$ -es élt kell választanunk, mert ez az egyetlen 3-as súlyú szintelen él. Az élt színezzük kékre, és a $H_1=\{5,6\}$, $H_2=\{7,8,9\}$ halmazokat vonjuk össze!



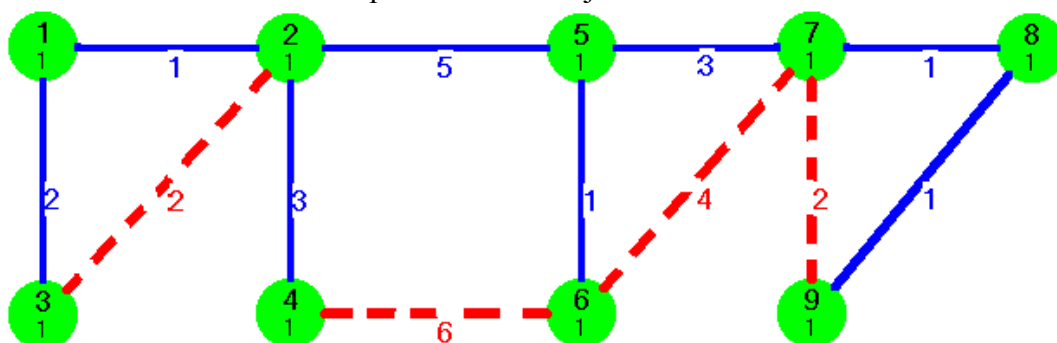
A tizedik lépésben az egyetlen 4-es súlyú szintelen él kerül kiválasztásra, amelynek két végpontja azonos osztályba esik, ezért pirosra színezzük.



A tizenegyedik lépésben a [2,5]-ös él a legkisebb súlyú szintelen él. Mivel a 2-es és 5-ös csúcsok különböző osztályokhoz tartoznak, így az élt színezzük kékre, és a $H_1=\{1,2,3,4\}$, $H_2=\{5,6,7,8,9\}$ halmazokat vonjuk össze! A halmazok összevonása után, már csak egy $H=\{1,2,3,4,5,6,7,8,9\}$ osztályunk (kék fánk) maradt, tehát a továbbiakban már nem alkalmazhatjuk a kék szabályt, azaz **megkaptunk egy minimális költségű feszítőfát**, amelynek élei: [1,3], [1,2], [2,4], [2,5], [5,6], [5,7], [7,8], [8,9].



Az ADT szintű leírás szerint még maradt egy lépés, mivel még van egy szintelen él [4,6]. Természetesen ezt az élt már csak pirosra színezhetjük.



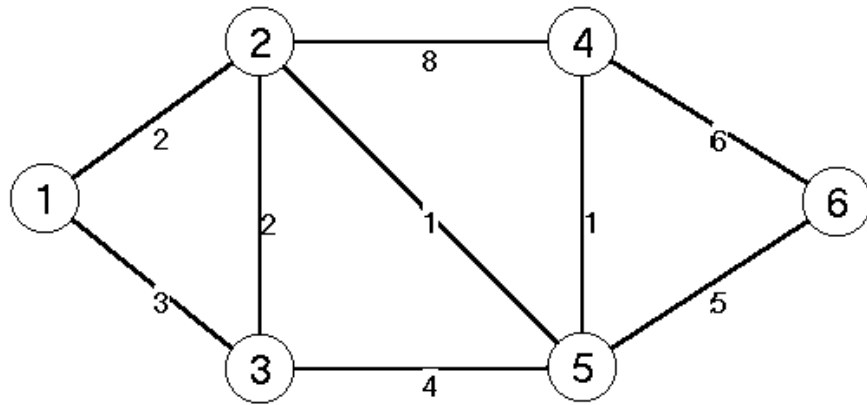
Az ábrázolás szintjén nem tárgyaljuk az algoritmust, mivel az jelenleg nem része a tananyagnak. Jobban szemügyre véve a Kruskal algoritmust, a műveletigénye a diszjunkt halmazok megvalósításától függ. Amennyiben az éleket egy kupac adatszerkezetben tároljuk az élsúlyokkal, mint kulccsal, egy él kivétele $O(\log e)$, e él kivétele $O(e \log e)$. Tehát jó lenne olyan ábrázolást választani a diszjunkt halmazoknak, hogy a teljes algoritmus műveletigénye $O(e \log e)$ maradjon. Ilyen létezik pl. a Rónyai-Iványos-Szabó: *Algoritmusok* c. tankönyvben [2] az UNIÓ-HOLVAN adatszerkezet.

7.4 Ellenőrző kérdések

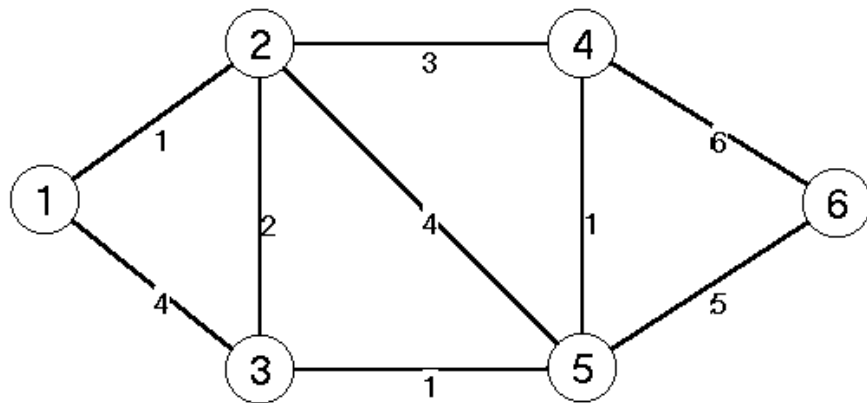
1. Adja meg a részgráf definícióját!
2. Adja meg a feszítőfa definícióját!
3. Adja meg a minimális költségű feszítőfa definícióját!
4. Mi az elsőfokú csúcs és mikor létezik ilyen?
5. Mikor létezik feszítőfa?
6. Hány éle van egy feszítőfának?
7. Már kész feszítőfa esetén, a gráf egy körének mentén hogyan cserélhetem ki (helyettesíthetem egymással) az éleket, hogy továbbra is feszítőfa maradjon?
8. Mit nevezünk takaros színezésnek?
9. Ismertesse a kék szabályt!
10. Ismertesse a piros szabályt!
11. Ismertesse a piros-kék eljárást!
12. Megoldható-e a minimális költségű feszítőfa keresése csak piros szabály alkalmazásával?
13. Megoldható-e a minimális költségű feszítőfa keresése csak kék szabály alkalmazásával?
14. Igaz-e, hogy a Prim algoritmus csak piros szabályt használ?
15. Ismertesse a Prim algoritmus elvét!
16. Milyen nevezetes adattípust használtunk a Prim algoritmusban?
17. Az adattípusnak milyen megvalósításait vizsgáltuk?
18. Milyen invariáns tulajdonságot tartunk fenn a Prim algoritmus során?
19. Mi a közös és mi a különböző a Prim és a Dijkstra algoritmus között?
20. Adja meg a Prim algoritmus műveletigényét különböző ábrázolások esetén!
21. Igaz-e, hogy a Kruskal algoritmus csak kék szabályt használ?
22. Mi a Kruskal algoritmus elve?
23. Milyen halmaz műveleteket használ a Kruskal algoritmus?
24. Mi a Kruskal algoritmus műveletigénye?
25. Mit értünk mohó algoritmus alatt?
26. Tegyük fel, hogy a piros-kék eljárással a G gráf egy minimális feszítőfáját már előállítottuk. Hogyan lehet módosítani ezt a fát, ha G -hez hozzá veszünk egy új csúcsot és az ehhez kapcsolódó éleket?

7.5 Gyakorló feladatok

1. Határozza meg az alábbi gráf minimális költségű feszítőfáját! Alkalmazza a piros, kék szabályokat felváltva, amíg lehet!



2. Szemléltesse a Prim algoritmus működését az alábbi gráfon! Írja fel a minQ és a Pi tartalmát menetenként (minden minQ-ból történő kivételnél)!



3. Írja fel a Prim algoritmust csúcsmátrixos ábrázolás esetén, soros minimumkeresés mellett!
4. Írja fel a Prim algoritmust éllistas ábrázolás esetén, kupac felhasználásával!
5. Tegyük fel, hogy a G irányítatlan, súlyozott gráf élsúlyai $[1..W]$ egészek. Hogyan lehetne gyorsítani a Prim algoritmust? [1]
6. Adjon algoritmust maximális súlyú feszítőfa meghatározására.
7. Legyen a feszítőfa súlya a feszítőfa legnagyobb súlyú élének a súlya. [2]
- a) Adjon algoritmust maximális súlyú feszítőfa meghatározására.
- b) Adjon algoritmust minimális súlyú feszítőfa meghatározására.

7.6 Összefoglalás

- Feszítőfa: az irányítatlan gráf összefüggő, körmentes részgráfja
- Minimális költségű feszítőfa: a feszítőfái közül a legkisebb költségűek egyike, ahol a feszítőfa költsége az élei súlyának összege.

- Egy színezés takaros, ha létezik gráfnak olyan minimális költségű feszítőfája, ami az összes kék élt tartalmazza, de egyetlen piros élt sem tartalmaz.
- Kék szabály: válasszunk ki egy olyan csúcshalmazt, amiből nem vezet ki kék él, majd egy legkisebb súlyú X -ből kimenő szintelen élt fessünk kékre
- Piros szabály: válasszunk a gráfban egy olyan egyszerű kört, amiben nincs piros él és a kör egyik legnagyobb súlyú szintelen élt színezzük pirosra.
- Piros-kék eljárás: legyen kezdetben a gráf minden éle szintelen. Alkalmazzunk a két szabályt tetszőleges sorrendben és helyen, amíg csak lehetséges.
- A Prim algoritmus elve: minden lépésben a kék szabályt alkalmazza egy kezdőcsúcsból kiindulva. Az algoritmus működése során egyetlen kék fát tartunk nyilván, amely folyamatosan növekszik, míg végül minimális költségű feszítőfa nem lesz. Kezdetben a kék fa egyetlen csúcsból áll, a kezdőcsúcsból, majd minden lépés során, a kék fát tekintve a kék szabályban szereplő X halmaznak, megkeressük az egyik legkisebb súlyú élt (mohó stratégia), amelynek egyik vége eleme a kék fának (X -ben van), a másik vége viszont nem (nem eleme X -nek). Az említett élt hozzá vesszük a kék fához, azaz az élt kékre színezzük, és az él X -en kívüli csúcsát hozzávesszük az X -hez. Az X -ből kimenő egyik legkisebb súlyú él meghatározásához használjunk egy minimum választó prioritásos sort ($\min Q$), amelyben a fához még nem tartozó (még nem eleme X -nek) csúcsokat tároljuk az X -től való távolsággal, mint kulcs értékkel. Az algoritmus minden lépésében kivesszük a $\min Q$ (egyik) legkisebb kulcsú elemét, majd a kivett csúcs szomszédai esetén módosítjuk az X -től való távolságot (ha szükséges).
- A prioritásos sort megvalósíthatjuk rendezetlen tömbben (d tömb) feltételes minimumkereséssel, vagy kupaccal.
- Műveletigény:
 - Rendezetlen tömb esetén:

$$T(n) = O(1 + n - 1 + n + n^2 + 3 * e) = O(n^2 + e) = O(n^2)$$
 - Kupac esetén:

$$T(n) = O(1 + n - 1 + n + n * \log n + 3 * e + e * \log n) = O((n + e) * \log n)$$
- Következmény az ábrázolásra: sűrű gráf esetén: csúcsmátrix + rendezetlen tömb, ritka gráf esetén: éllista + kupac.
- A Kruskal algoritmus elve: kezdetben legyen n db kék fa, azaz a gráf minden csúcsa egy-egy (egy pontból álló) kék fa, és legyen minden él szintelen. Minden lépés során kiválasztjuk az egyik legkisebb súlyú szintelen élt. Ha a kiválasztott él két végpontja különböző kék fában van, akkor színezzük kékre, különben (az él két vége azonos kék fában van, tehát a kék fa éleivel kört alkot) színezzük pirosra. Az algoritmust akkor áll le, ha már nincs szintelen él.
- A Kruskal algoritmus kulcsa diszjunkt halmazok kezelés, erre létezik hatékony adatszerkezet: Unió-Holvan [2]

8. Mélységi bejárás és alkalmazásai

8.1 Mélységi bejárás

Az **algoritmus elvét** a *Bejárási/keresési stratégiák* című fejezetben már láttuk, most foglaljuk össze röviden. Egy kezdőpontból kiindulva addig megyünk egy él mentén, ameddig el nem jutunk egy olyan csúcsba, amelyből már nem tudunk tovább menni, mivel nincs már meg nem látogatott szomszédja. Ekkor visszamegyünk az út utolsó előtti csúcsához, és onnan próbálunk egy másik él mentén tovább menni. Ha ezen az ágon is minden csúcsot már bejártunk, ismét visszamegyünk egy csúcsot, és így tovább.

Most vizsgáljuk meg a bejárást **ADS szinten** egy példán:

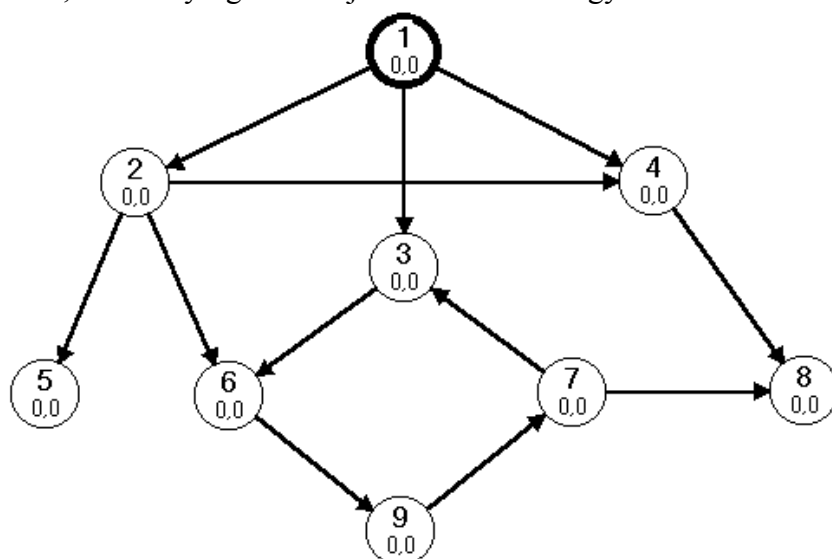
Színezzük a csúcsokat attól függően, hogy az illető csúcsra vonatkozóan a bejárás milyen fázisban van [1]:

- 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

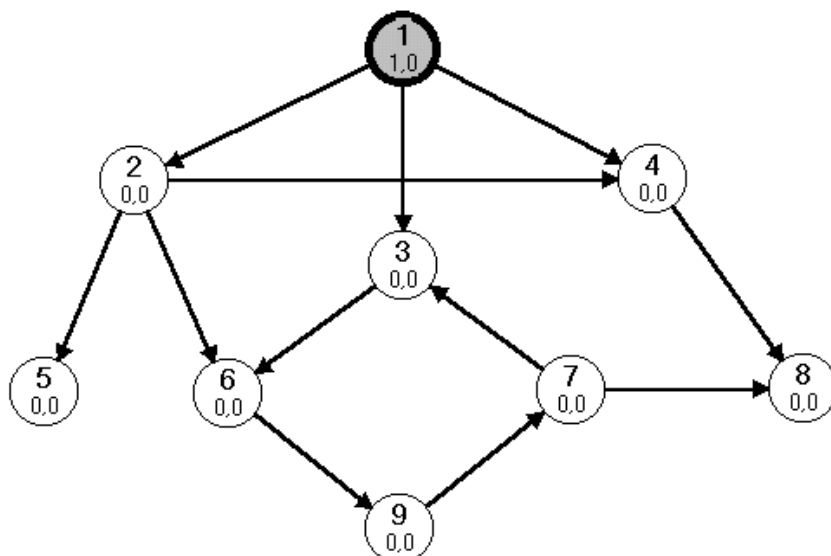
A bejárás során tároljuk el, hogy egy csúcsot hányadikként értünk el, azaz hányadikként lett szürke és tároljuk el, hogy hányadikként fejeztük be a csúcs, és a belőle elérhető csúcsok bejárását, azaz a csúcs hányadikként lett fekete [2]. Az említett számokat nevezzük mélységi, illetve befejezési számnak és az ábrákon a csúcsok címkéi alatt fogjuk megjeleníteni.

A példában egy csúcsból kimenő élek feldolgozási sorrendje legyen a szomszéd csúcsok címkéje szerint növekedően rendezett (pl.: láncolt ábrázolásnál az éllista a csúcsok címkéje szerint rendezett).

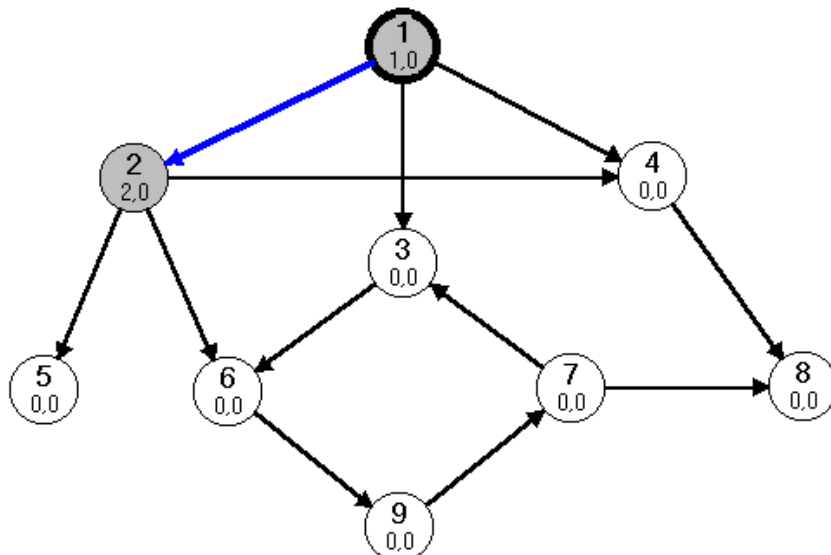
Tehát nézzük az alábbi példát, amelyben a kezdőcsúcs legyen az 1-es csúcs. Legyen kezdetben minden csúcs fehér, és a mélységi és befejezési számuk is legyen az extrémális 0.



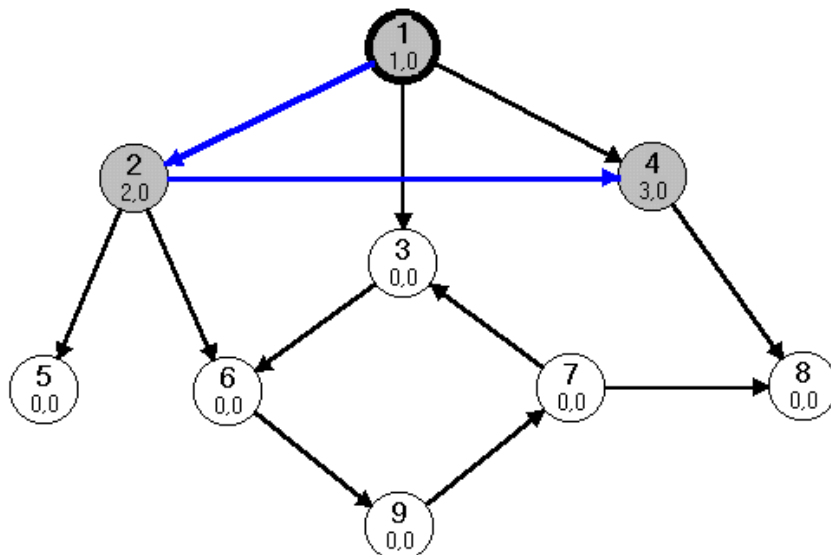
A kezdőcsúcsot érjük el elsőként, tehát színezzük szürkére, és a mélységi számát állítsuk be 1-re



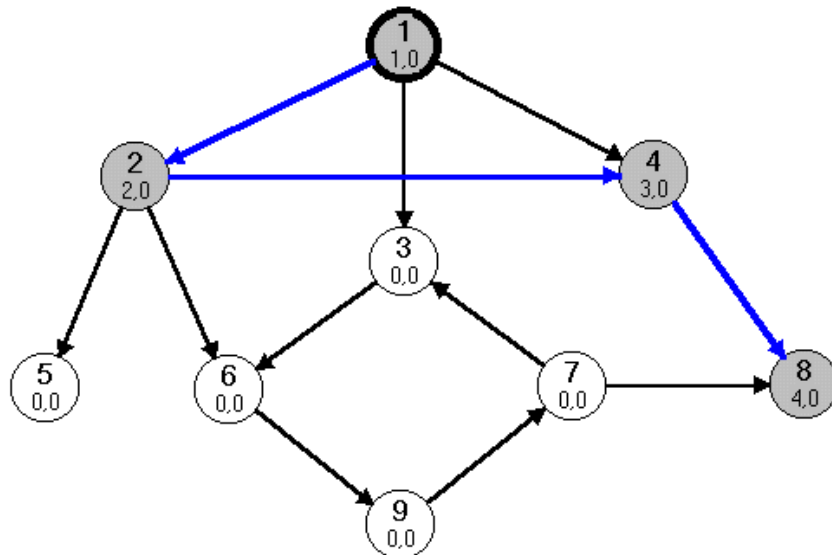
Az 1-es csúcsból három él vezet ki, azaz három él mentén indulhatnánk el, de a kikötöttük feltételként, hogy az élek feldolgozási sorrendje legyen a szomszéd csúcsok címkéje szerint növekedően rendezett. Tehát a 2-es csúcsot érjük el másodikként.



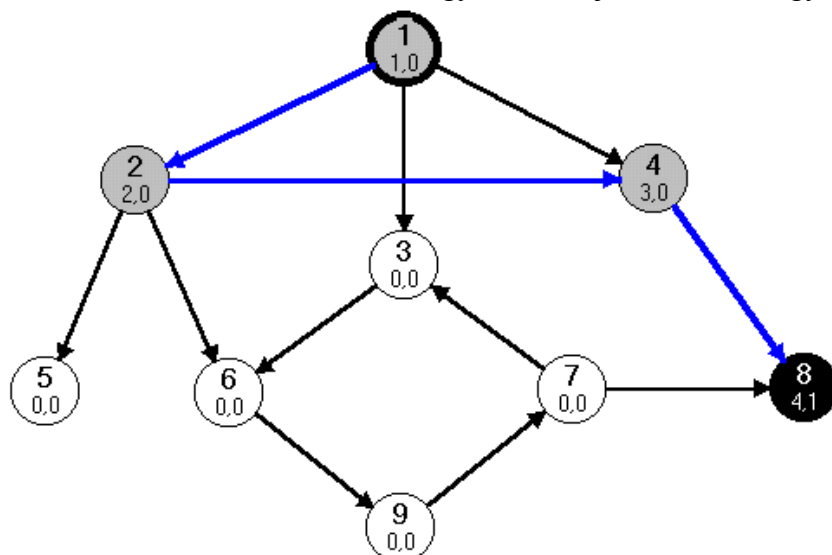
Ezután elérjük harmadikként a 4-es csúcsot.



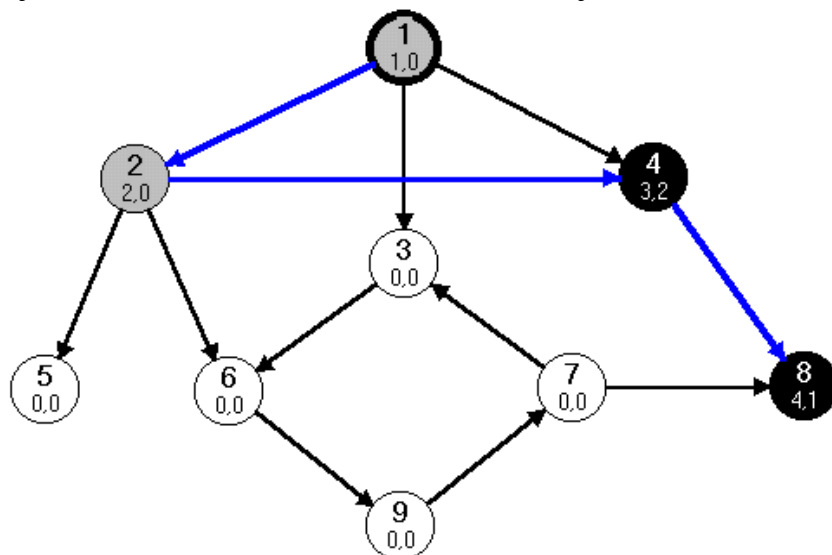
Negyedikként a 8-as csúcsot érjük el.



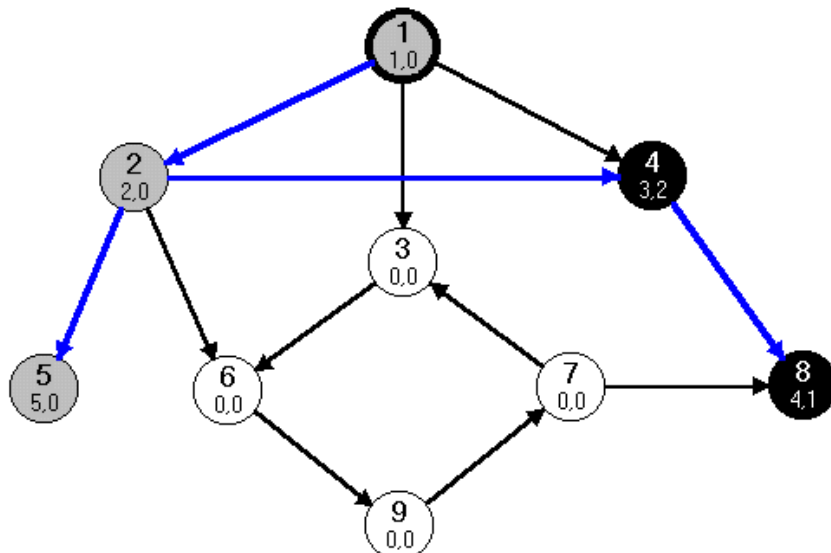
Mivel a 8-as csúcsnak nincs olyan szomszédja, amit még nem látogattunk volna meg (nincs egyáltalán szomszédja), a 8-as csúcs bejárását befejeztük, a csúcsot színezzük feketére. Mivel a bejárás során a 8-as csúcs lett elsőként fekete, így az ő befejezési száma legyen az egyes.



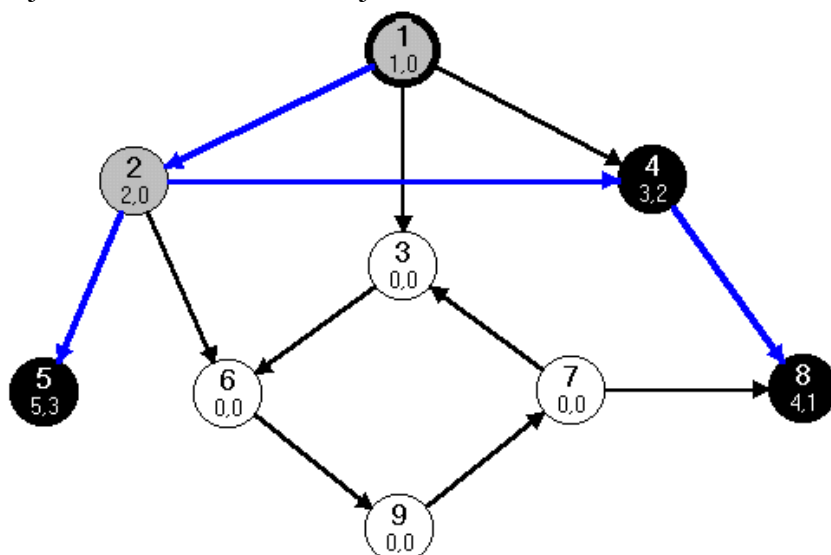
A bejárás során a megtett utunk $\langle 1,2,4,8 \rangle$. Most menjünk vissza az utolsó előtti csúcsra, a 4-es csúcsra. Mivel a 4-es csúcsnak sincs még meg nem látogatott szomszédja, így a 4-es csúcs bejárását is befejeztük, színezzük a csúcsot feketére, és a bejárési számát állítsuk be kettőre.



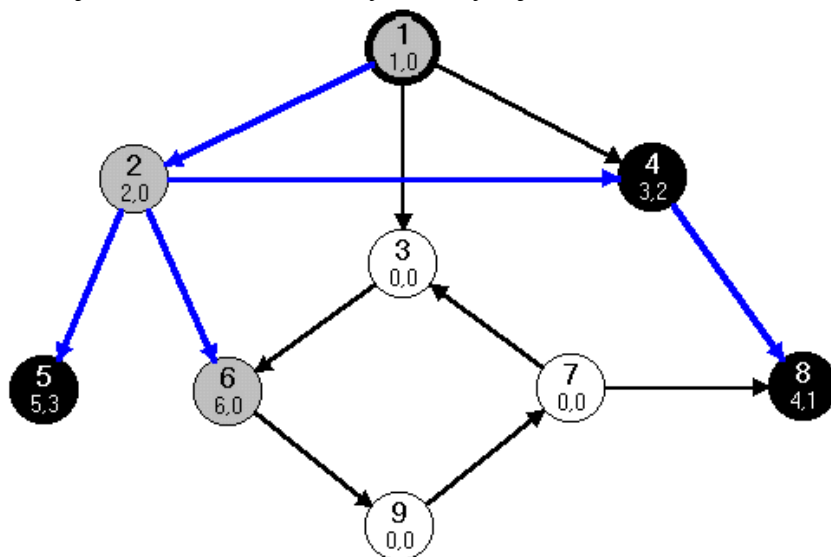
Menjünk vissza a 2-es csúshoz. A 2-es csúsnak két olyan szomszédja is van, amelyet még nem látogattunk meg. Látogassuk meg a kisebb címkéjű csúcsot.



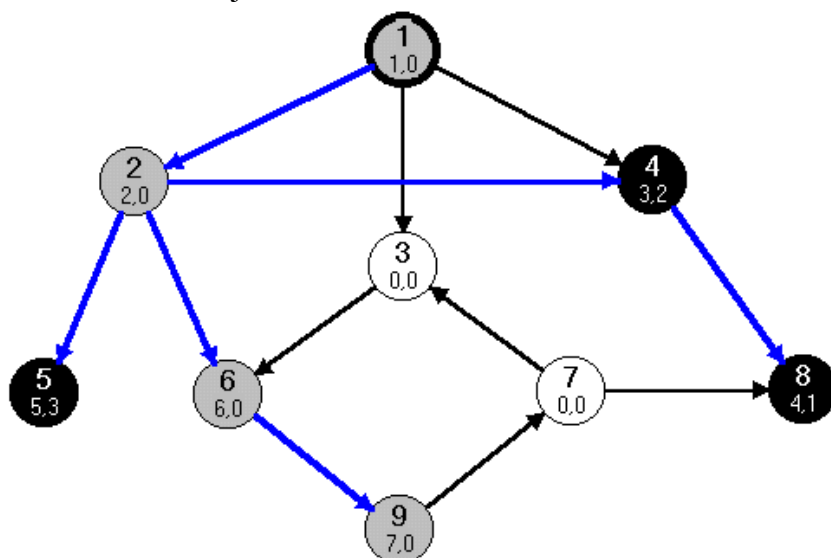
Az 5-ös csúcs bejárását harmadikként befejeztük.



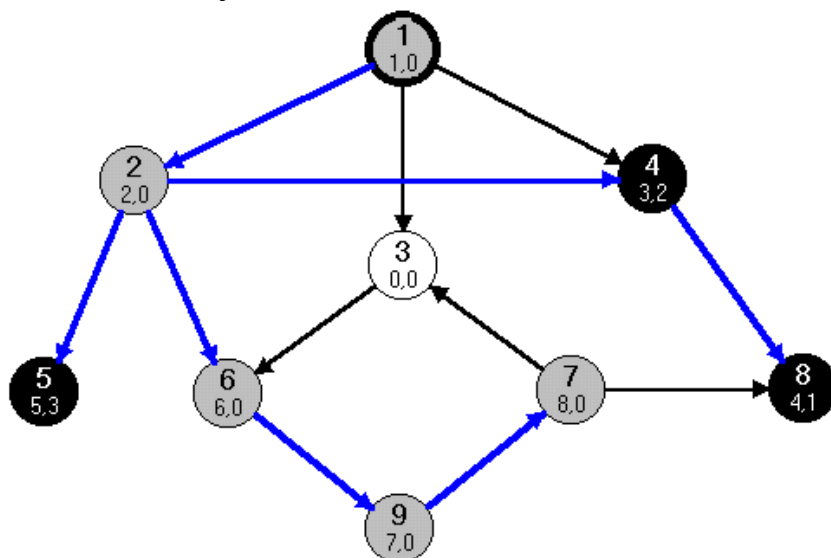
A 2-es csúcsból a bejárást a 6-os csúcs irányába folytatjuk.



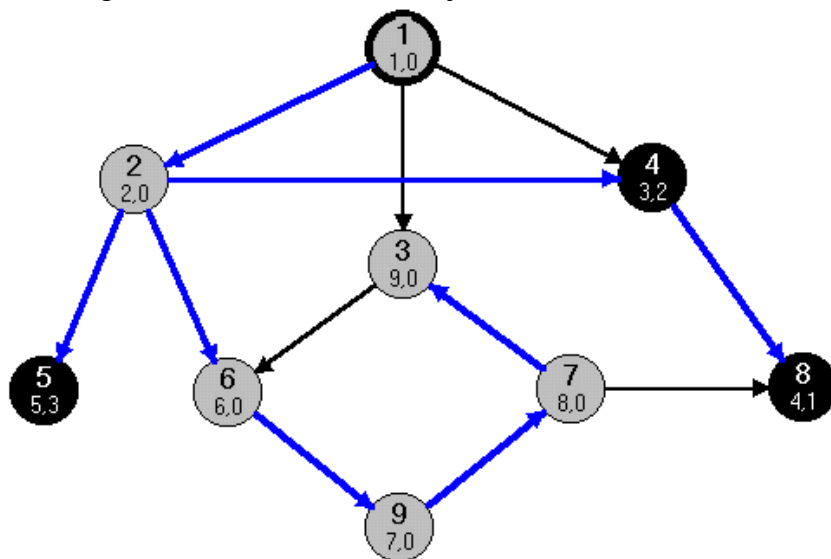
Tovább haladva hetedikként elérjük a 9-es csúcsot.



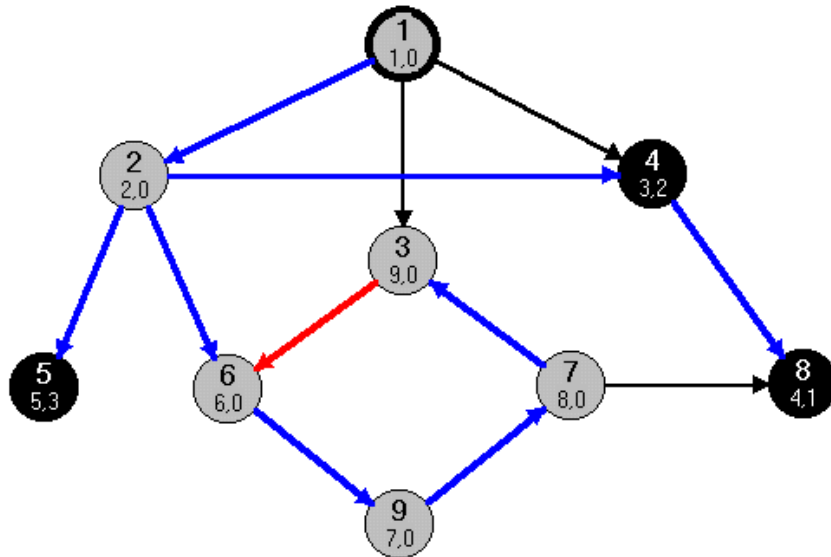
Nyolcadikként a 7-es csúcsot érjük el.



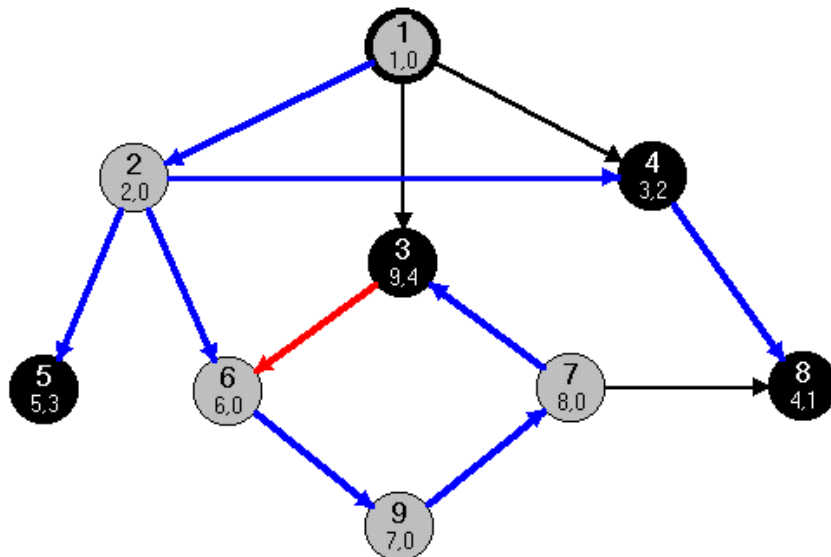
Majd a 7-es csúcsból elsőként megvizsgáljuk a 3-as csúcsba vezető él mentén a lehetőségeket. Mivel a 3-as csúcs még fehér, kilencedikként elérjük a 3-as csúcsot.



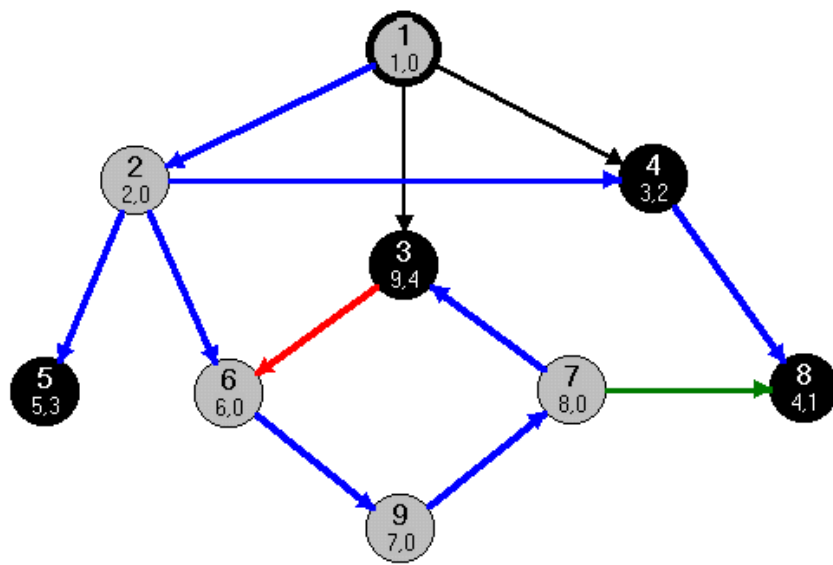
A 3-as csúsból a 6-os csúcsba vezet él, azonban a 6-os csúcsot már meglátogattuk, azaz a színe már nem fehér, azaz erre már nem folytatjuk a bejárást (különben a körön végtelen sokáig keringhetnénk).



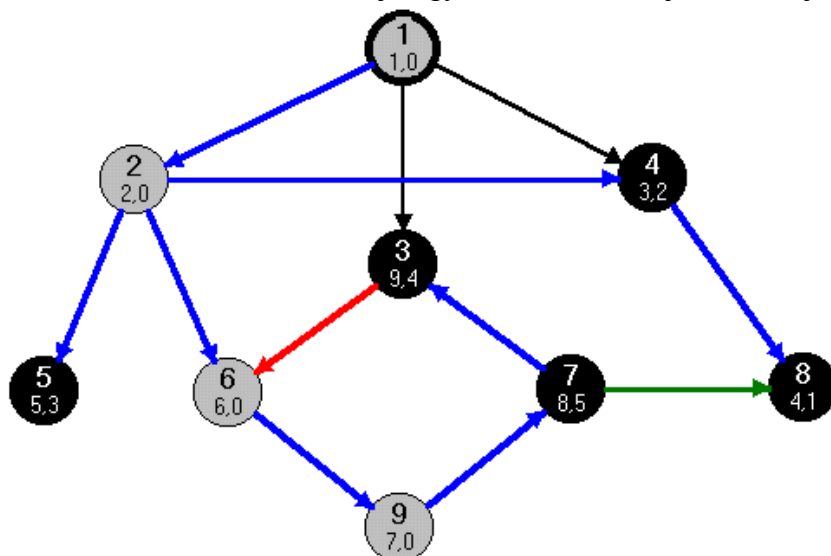
Mivel a 3-as csúsból már nem vezet él még meg nem látogatott csúcsba, így a 3-as csúcs bejárást is befejeztük.



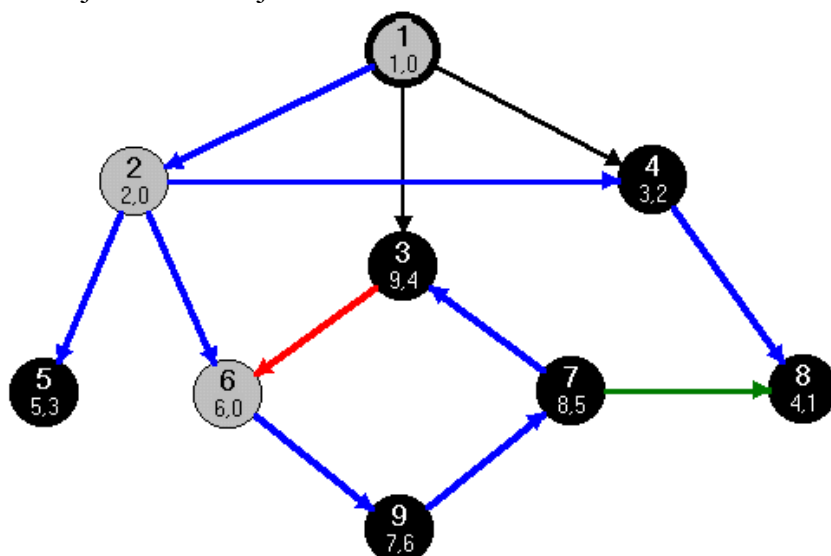
Visszamegyünk a 7-es csúcsba, ahol a sorrendben következő él, a (7,8) mentén vizsgálódunk. Azonban a 8-as csúcs színe már fekete, tehát már befejeztük a bejárást, így erre már felesleges volna mennünk.



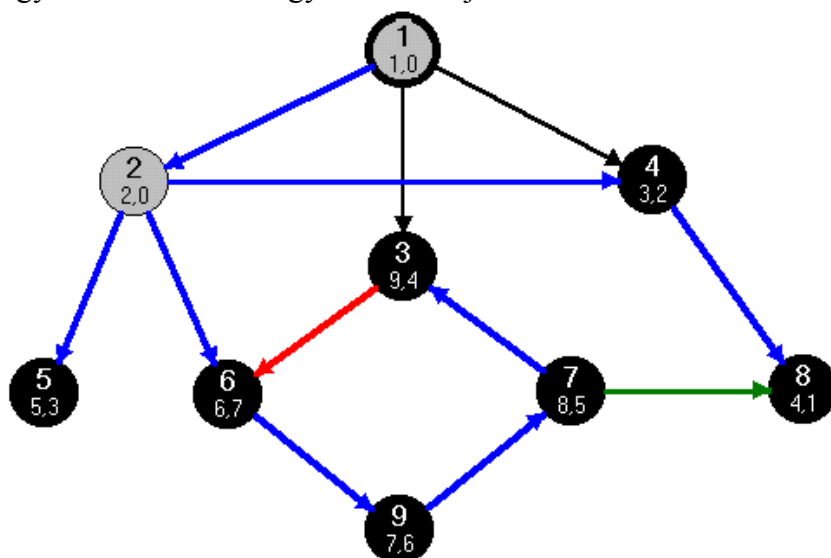
Mivel a 7-es csúcsnak nincs fehér szomszédja, így ötödikként befejeztük a bejárást.



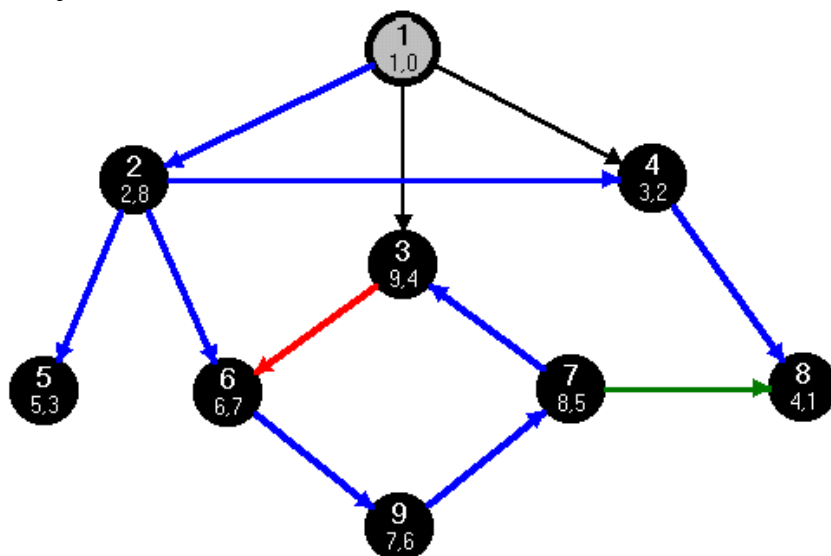
A 9-es csúcsnak a bejárást is befejeztük.



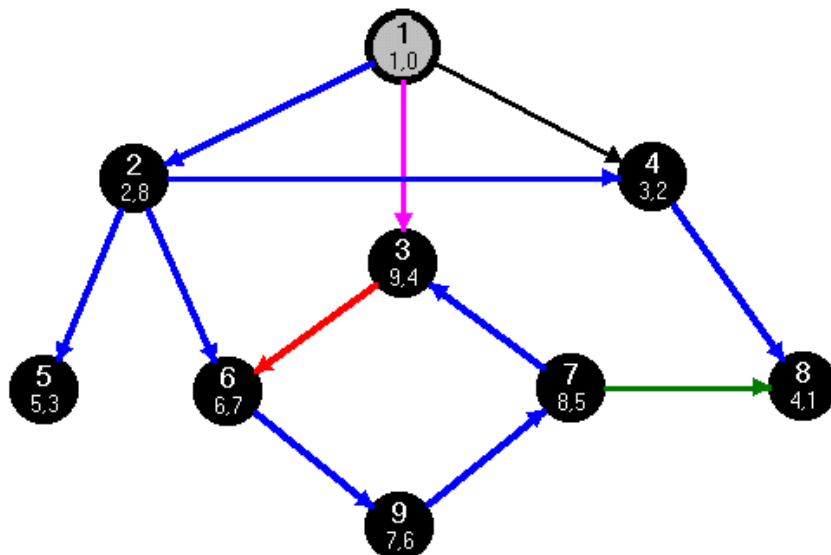
Az úton ismét egy csúccsal visszamegyünk és befejezzük a 6-os csúcsot is.



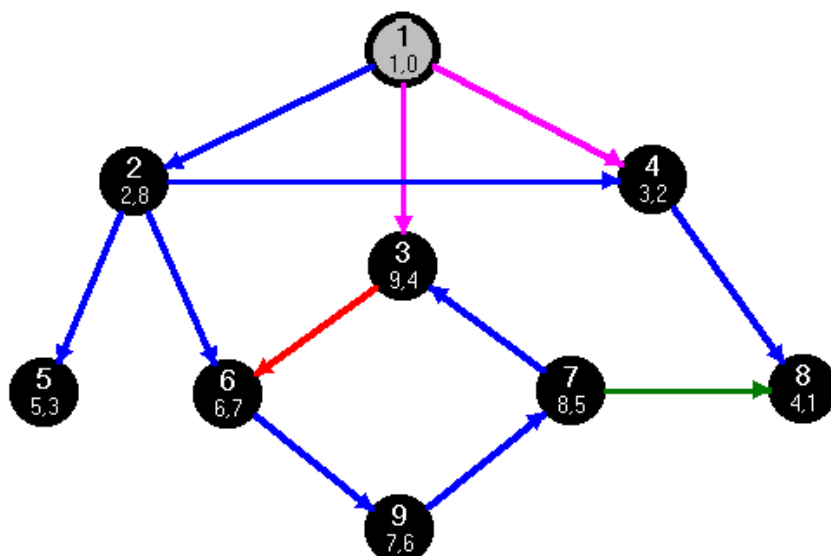
A 2-es csúcsra lépve, látható, hogy minden kimenő éle mentén már próbálkoztunk, így nyolcadikként befejezzük őt is.



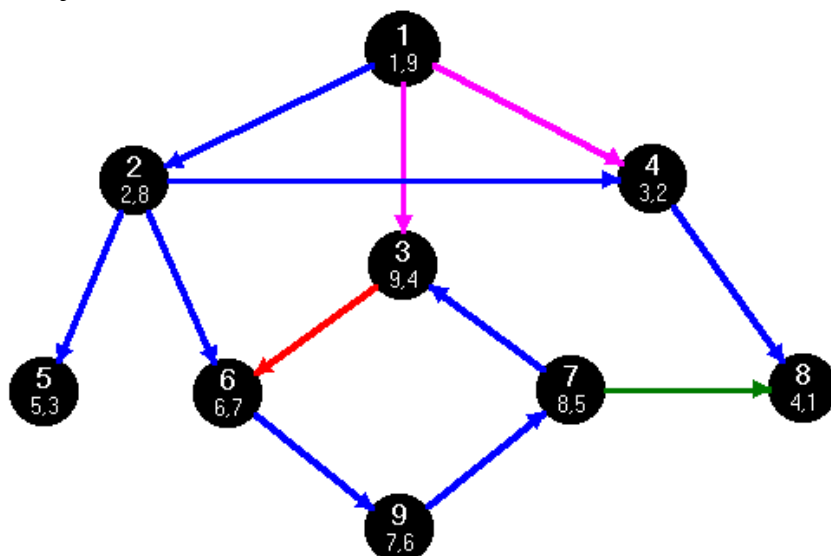
Az 1-es csúcsra lépve megvizsgáljuk a 3-as csúcsot, de látva, hogy színe nem fehér, arra nem megyünk tovább.



Ezután az előzőhöz hasonlóan még megvizsgáljuk a maradék (1,4) él mentén a 4-es csúcsot, de annak színe sem fehér.



Végül az 1-esből kimenő összes él mentén már megvizsgáltuk a bejárési lehetőségeket, így az 1-es csúcsot is befejeztük utolsóként.

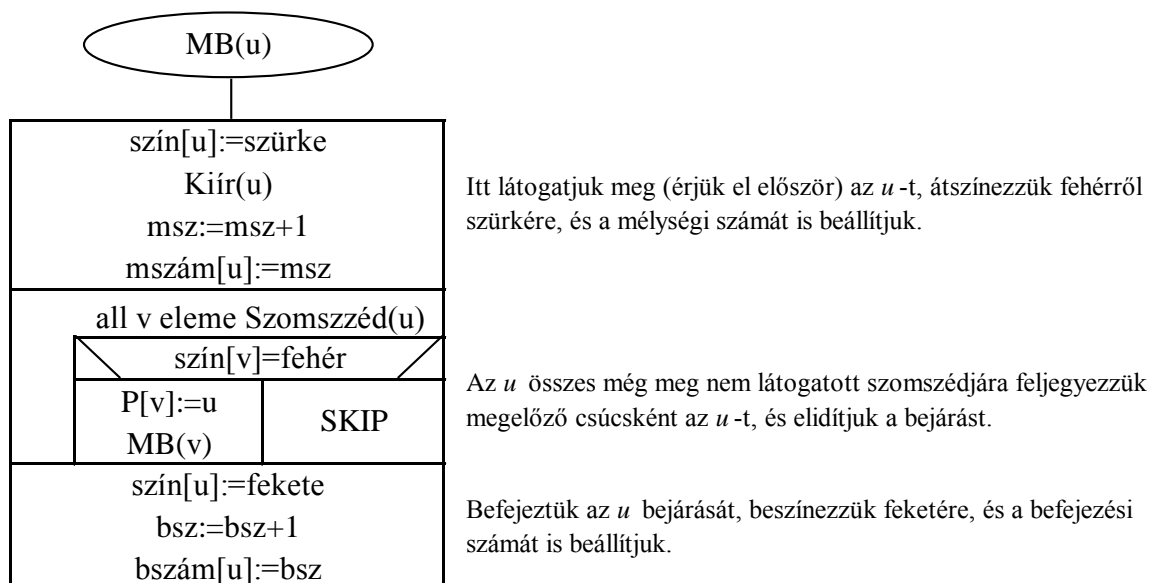
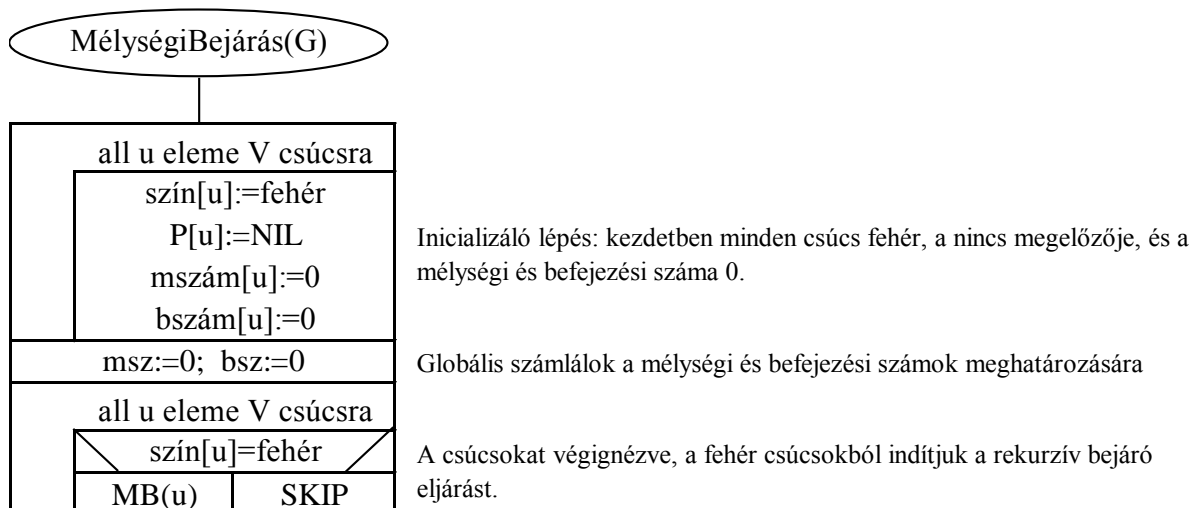


Az algoritmus **ADT szintű** leírása [1][2]:

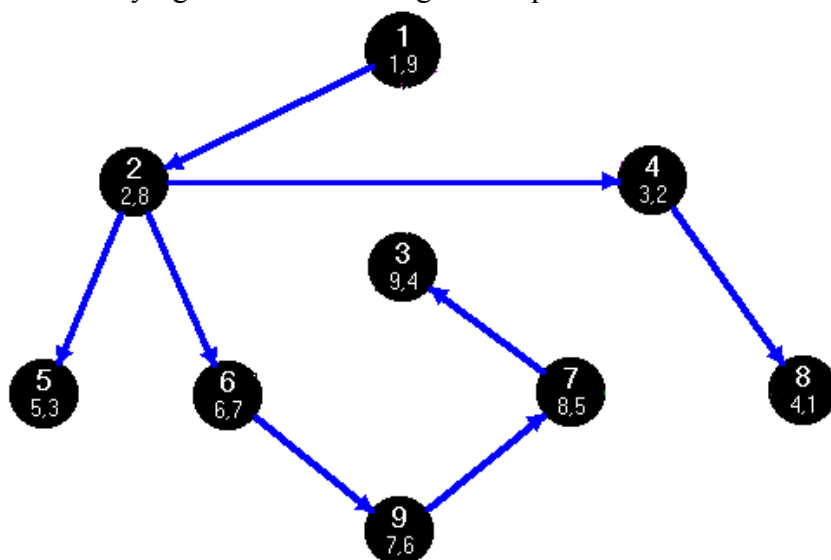
Legyen $G=(V,E)$ irányított vagy irányítatlan, véges gráf, ahol $V=\{1,2,\dots,n\}$. Továbbá definiáljuk az alábbi vektorokat:

- $szín[1..n]$: az ADS szintű színezés megvalósítására
- $mszám[1..n]$ és $bszám[1..n]$: az ADS szinten említett mélységi és befejezési számok nyilvántartására
- $P[1..n]$: a bejárás során, egy csúcs megelőző csúcsának a nyilvántartására (a korábban látottakhoz hasonlóan, pl.: szélességi bejárás vagy Dijkstra algoritmus).

Az előző példában úgy kezdtük a bejárást, hogy kijelöltünk egy kezdőcsúcsot és a kezdőcsúcsból elérhető csúcsokat (a példában az összes csúcs ilyen volt) jártuk be. Egy másik példán előfordulhatna, hogy lennének olyan csúcsok, amelyeket egyáltalán nem látogatnánk meg (a szélességi keresés is így működik). Azonban a későbbi alkalmazások érdekében, a bejárásunk legyen olyan, hogy minden csúcsot meglátogatunk. Tekintsünk egy kezdőcsúcsot, és innen indítsunk el egy bejárást. Miután visszajutottunk az említett kezdőcsúcsba (azaz a kezdőcsúcsból elérhető csúcsokat bejártuk), nem fejezzük be az algoritmust, hanem keresünk egy eddig még meg nem látogatott (azaz fehér) csúcsot és innen újra indítjuk a bejárást. Ezt eddig folytatjuk, amíg van fehér csúcsunk. Nyilván minden ilyen menetben legalább egy csúcsot átszínezzünk 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. Tehát az algoritmust bontsuk két részre. Az egyik része egy kezdőcsúcsból indítja a bejárást, ez lesz az $MB(u)$ eljárás, a másik pedig végigmegy a csúcsok halmazán, és egy fehér csúcsot találva elindítja az előbb említett eljárást. A kezdőcsúcsból induló mélységi bejárásra ($MB(u)$) egyszerű rekurzív definíciót adni, miszerint az u csúcsot akkor jártuk be, ha az összes szomszédját már bejártuk (vagy legalább elértük, a kör miatti végtelen ciklus elkerülése). Tehát az $MB(u)$ eljárást most rekurzíven adjuk meg.



Az $MB(u)$ eljárás futása során feljegyezzük a P tömbbe egy csúcs megelőzőjét, így egy u csúcsból kiinduló, úgynevezett **mélységi fát** kapunk. Az ADS szinten említett példában az 1-es csúcsból kiinduló mélységi faként az alábbi gráfot kaptuk:



Össességében, pedig a P tömbben feljegyzett megelőzési reláció, G egy részgráfját adja, amelyet **mélységi erdőnek** nevezünk.

8.1.1. Definíció [2]: Legyen $G=(V,E)$ irányított, véges gráf. A G mélységi bejárása után a P -ben keletkező megelőzési reláció által ábrázolható irányított T részgráfot, a G egy **mélységi erdőjének** nevezzük.

8.1.2. Definíció [2]: Legyen T a $G=(V,E)$ gráf egy mélységi erdője, és $x, y \in V$ csúcsok. Az y **leszármazottja** x -nek T -ben, ha $\exists x \leadsto y$ T -beli irányított út.

8.1.3. Állítás [2]: (szükséges feltétel) Legyen T a $G=(V,E)$ gráf egy mélységi erdője, $x, y \in V$ csúcsok, $x \neq y$, és y **leszármazottja** x -nek T -ben, akkor $mszám[y] > mszám[x]$.

Bizonyítás: y **leszármazottja** x -nek, tehát $\exists x \leadsto y$ út T -ben. Ez az út úgy keletkezik, hogy az út (u,v) éleinek vizsgálatakor, $szín[v]=fehér$ csúcsok esetén a P tömbbe bejegyzésre kerül a megelőző u csúcs, majd meghívjuk az $MB(v)$ eljárást, ahol a v csúcs legalább egyel megnövelt mélységi szám értéket fog kapni.

8.1.4. Állítás [2]: (szükséges és elégséges feltétel) Legyen T a $G=(V,E)$ gráf egy mélységi erdője, $x, y \in V$ csúcsok, $x \neq y$. Ha $mszám[y] > mszám[x]$ és $bszám[y] < bszám[x] \Leftrightarrow y$ **leszármazottja** x -nek T -ben.

Bizonyítás:

A mélységi és befejezési számok viszonyából következik, hogy előbb meghívtuk az $MB(x)$ eljárást, majd még mielőtt végett ért volna az x szomszédainak rekurzív bejárását végző ciklus, meghívtuk $MB(y)$ -t, amely teljes egészében lefutott, majd csak ez után terminálhatott az említett ciklus. Ez akkor lehetséges, ha létezik egy olyan $z \in V$ csúcs, hogy az $MB(z)$ eljárásban hívtuk meg $MB(y)$ -ot, és $P[y]=z$ bejegyzésre kerül. Továbbá a z csúcs olyan, hogy $z=x$ (azaz y szomszédja x -nek) vagy $MB(z)$ lefutása is teljesül, amit az $MB(y)$ -ről az előbb említettünk. Ezt a gondolatmenetet addig folytathatjuk, míg $z=x$ nem lesz, miközben láthatjuk, hogy a P tömb bejegyzései által reprezentált irányított úton haladunk visszafelé (a rekurzív hívási fán haladunk felfelé). Tehát ez az út igazolja, hogy y **leszármazottja** x -nek T -ben.

8.1.5. Tétel (Fehér út tétel) [1]: Legyen T a $G=(V,E)$ gráf egy mélységi erdője, $x, y \in V$ csúcsok, $x \neq y$. y **leszármazottja** x -nek T -ben. $\Leftrightarrow x$ elérésekor az y elérhető az x -ből, az x kivételével csak fehér csúcsokat tartalmazó úton.

Bizonyítás:

\Rightarrow : y **leszármazottja** x -nek $\Rightarrow \exists x \leadsto y$ T -beli út, amelynek mentén minden csúcs **leszármazottja** x -nek. Ezen fabeli út minden u csúcsára teljesül: $mszám[u] > mszám[x]$ (8.1.3. szükséges feltétel), azaz minden u csúcsot később érünk el, mint az x -et $\Rightarrow x$ elérésekor az u **leszármazott** csúcsok még **fehérek**.

\Leftarrow : Tegyük fel, hogy x elérésekor létezik y -ba vezető fehér csúcsokból álló út, legyen v egy ilyen út első olyan csúcsa, amelyet az x szomszédait felsoroló ciklusban elérünk, azaz meghívjuk az $MB(v)$ eljárást. Mivel x már szürke és v még fehér $\Rightarrow v$ -t később érjük el, mint x -et, tehát $mszám[v] > mszám[x]$. Továbbá $MB(x)$ belsejéből hívtuk meg $MB(v)$ -t tehát $MB(v)$ előbb lefut, mint $MB(x)$, azaz $bszám[v] < bszám[x]$. Tehát a 8.1.4. állítás szerint v **leszármazottja** x -nek. Azonban feltettük, hogy v -t érjük el először az y -hoz vezető úton, tehát az út többi csúcsa v elérésekor még fehér, így a $v \leadsto y$ útra hasonlóan alkalmazható a fenti rekurzív gondolatmenet (míg el nem jutunk az $y \leadsto y$ útig). Miután beláttuk, hogy v **leszármazottja** x -nek és y **leszármazottja** v -nek $\Rightarrow y$ **leszármazottja** x -nek.

Megjegyzés: ha $x=y$, akkor triviálisan teljesül a kölcsönös **leszármazottság**.

8.1.6. Definíció [2]: Az élek osztályozása egy adott mélységi bejárás szerint.

Legyen T a $G=(V,E)$ gráf egy mélységi erdője, és $x, y \in V$ csúcsok. Az $(x, y) \in E$ él

- a) faél, ha $x \rightarrow y$ éle T -nek.
- b) előreél, ha $x \rightarrow y$ nem éle T -nek, de y leszármazottja x -nek T -ben, és $x \neq y$.
- c) visszaél, ha x leszármazottja y -nak T -ben ($x \rightarrow y$ nem éle T -nek; ide tartozik $x=y$ hurokél is).
- d) keresztél, ha x és y nem leszármazottai egymásnak T -ben.

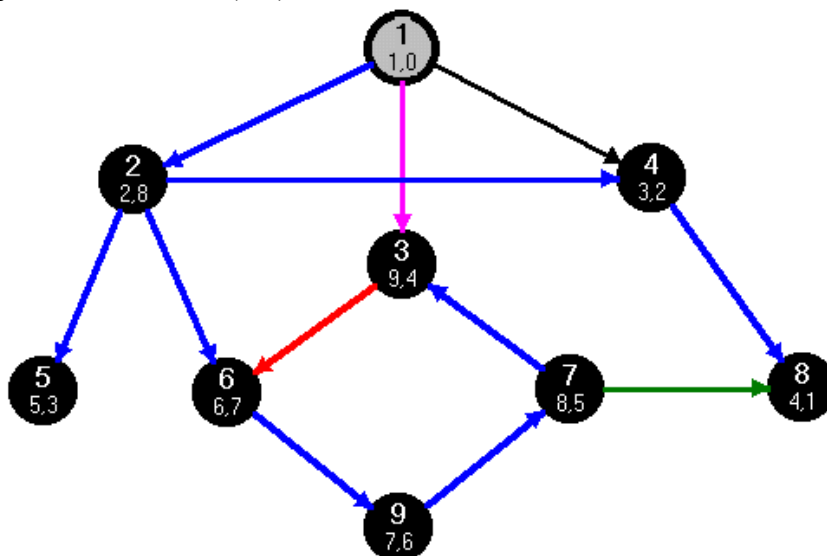
8.1.7. Tétel [2]: Az éltípusok azonosítása a bejárás során.

Tegyük fel, hogy $G=(V,E)$ irányított véges gráf mélység bejárása során éppen az $(x, y) \in E$ vizsgálatánál tartunk. Ekkor (x, y) él

- a) faél, ha $mszám[y] = 0$.
- b) előreél, ha $mszám[y] > mszám[x]$.
- c) visszaél, ha $mszám[y] \leq mszám[x]$ és $bszám[y] = 0$.
- d) keresztél, ha $mszám[y] < mszám[x]$ és $bszám[y] > 0$.

Bizonyítás [2]:

- a) $mszám[y] = 0$ azt jelenti, hogy most érjük el az y csúcsot, azaz a csúcs még fehér. Továbbá az (x, y) él mentén jutunk az y csúcsba, mint x szomszédja, amely az $MB(x)$ x szomszédait felsoroló ciklusában, az elágazás bal oldali ágában lehetséges, ahol valóban a P tömbbe bejegyzésre kerül az él, tehát faél lesz.
- b) $mszám[y] > mszám[x]$ és $mszám[x] > 0$, így $mszám[y] > 0$, tehát y nem fehér (így az él nem lehet faél). Továbbá a mélységi számok viszonyából következik, hogy az $MB(y)$ -t később hívtuk meg, mint az $MB(x)$ -t, de az $MB(x)$ még nem fejeződött be. Így az $MB(y)$ meghívása, az $MB(x)$ eljárás x szomszédait felsoroló ciklusában, valamely szomszédra meghívott MB eljárásban, vagy annak rekurzív leszármazottjában történt, az (x, y) él vizsgálata előtt. Azonban az (x, y) él vizsgálata a ciklus egy későbbi iterációjában történik, tehát az $MB(y)$ -nak mostanra be kellett fejeződnie, de az $MB(x)$ még csak ezek után fog befejeződni $\Rightarrow bszám[y] < bszám[x]$. A 8.1.4. állítást felhasználva beláthatjuk, hogy y leszármazottja x -nek T -ben, és láttuk, hogy nem lehet faél, tehát előreél.
Láthatjuk az ADS szinten vizsgált példán, hogy az 1-es csúcsból a 3-as csúcsba vezető él feldolgozásakor a 3-as csúcsba már eljutottunk az $\langle 1, 2, 6, 9, 7, 3 \rangle$ úton, tehát a 3-as leszármazottja az 1-nek, és az $(1, 3)$ él nem faél, tehát előreél.

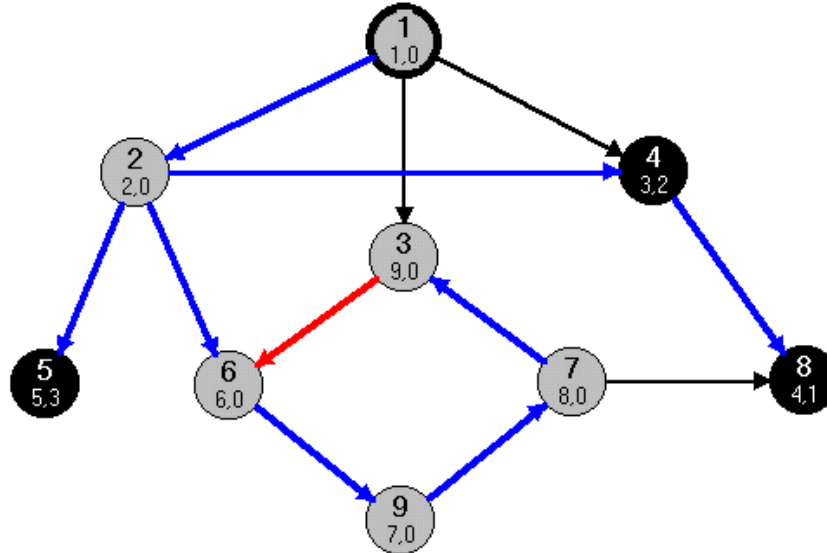


c) $mszám[y] \leq mszám[x]$ és $bszám[y] = 0$

$mszám[y] = mszám[x] \Rightarrow (x,y)$ hurokél, ami triviálisan visszaél

$mszám[y] < mszám[x]$ és $bszám[y] = 0 \Rightarrow$ Az y bejárása elkezdődött, de még nem fejeződött be. Elkezdtuk az x bejárását, amelynek során vizsgáljuk az (x,y) élt. Tehát az y bejárása során jutunk el az x -hez, azaz x leszármazottja y -nak.

(Az $MB(y)$ eljárásban, vagy annak valamely rekurzív leszármazottjában hívtuk meg az $MB(x)$ -et, tehát az $MB(x)$ -nek előbb kell befejeződnie, mint az $MB(y)$ -nak, így a 8.1.4. állítás alkalmazható). Az alábbi ábrán a (3,6) él visszaél.



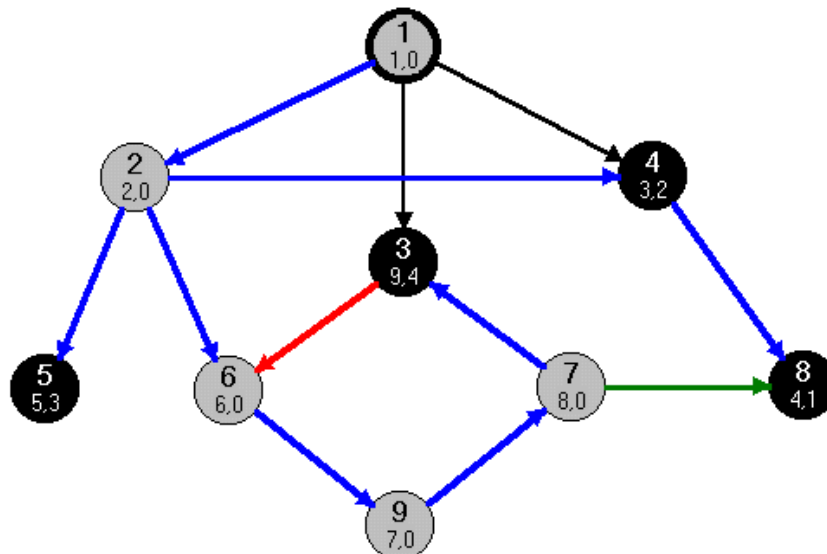
d) $mszám[y] < mszám[x]$ és $bszám[y] > 0$

$mszám[y] < mszám[x] \Rightarrow y$ nem leszármazottja x -nek, mivel a 8.1.3. állítás szerinti szükséges feltételt nem teljesíti.

$mszám[y] < mszám[x]$ szükséges feltétele, hogy x leszármazottja legyen y -nak, de az y bejárása befejeződött ($bszám[y] > 0$), míg az x bejárása még nem, tehát a $bszám[x]$ csak nagyobb lehet $bszám[y]$ -nál, azaz az elégséges feltétel nem teljesül.

Tehát x és y nem leszármazottai egymásnak T -ben, amiből következik, hogy (x,y) él keresztél.

Az alábbi példán a (7,8) él keresztél.



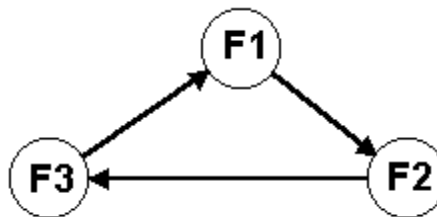
Megjegyzés: Hogyan lehetne azonosítani az (x,y) él típusát az él vizsgálata során az y csúcs színének segítségével (lásd gyakorlaton)?

A mélységi bejárás műveletigénye [2]: Mivel minden csúcsra pontosan egyszer hívjuk meg az *MB* eljárást, az eljárásban minden csúcsnak a szomszédait vizsgáljuk, összesen annyiszor ahány éle van a gráfnak, így $T(n)=O(n+e)$.

8.2 DAG tulajdonság

Probléma: Operációs rendszerekben, adatbázis kezelő rendszerekben előfordulnak olyan esetek, hogy egyes folyamatok más folyamatokra várnak. Ilyen eset lehet egy nyomtatón való nyomtatás várakoztatása, mert a nyomtatót egy másik folyamat használja, vagy egy adattáblán való művelet elvégzésének a várakoztatása, mert egy másik folyamat a táblát zárolta. Építsünk fel egy ú.n. **várakozási gráfot**, ahol a gráf csúcsai a folyamatok, és egy u csúcsból vezessen él egy v csúcsba, ha az u folyamat a v folyamatra várakozik. Az említett várakozási reláció **nem** feltétlen **szimmetrikus**, tehát a gráfunk legyen **irányított**. Probléma akkor van, ha gráfban **irányított kör keletkezik**, ekkor a folyamatok akár végtelen ideig is várakozhatnak egymásra. A jelenséget a szakirodalom **holtpontnak** nevezi (bővebben lásd az operációsrendszerek és adatbázis kezelő rendszerek témakörben megjelent szakirodalomban).

Pl.: F1 folyamat várakozik F2-re, F2 várakozik F3-ra, és F3 várakozik F1-re.



8.2.1. Definíció [2]: (DAG (Directed Acyclic Graph) = Irányított Körmentes Gráf
A $G=(V,E)$ irányított, véges gráf **DAG** tulajdonságú, ha nem tartalmaz irányított kört.

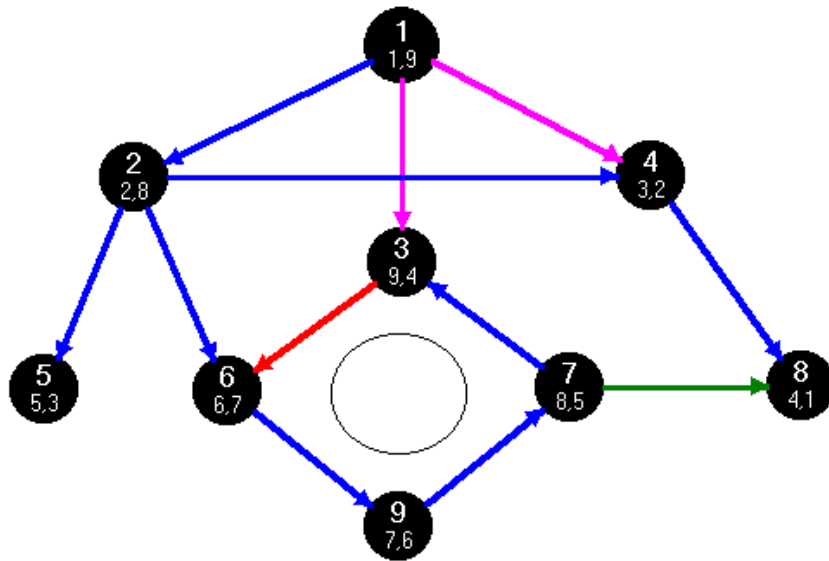
A 8. fejezet további részében az említett $G=(V,E)$ gráf legyen irányított, véges gráf.

DAG tulajdonság ellenőrzése \Leftrightarrow irányított kör felderítése a gráfban

8.2.2. Állítás [2]: Ha a G gráf mélységi bejárása során **találunk visszaélt** $\Rightarrow G$ **nem DAG**

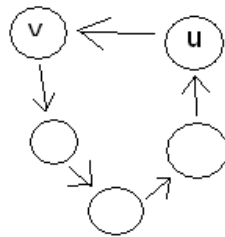
Bizonyítás [2]: Tegyük fel, hogy. $(u,v) \in E$ egy visszaélt $\Rightarrow u$ leszármazottja v -nek a mélységi fában, azaz létezik $v \rightsquigarrow u$ irányított út. Ehhez hozzávéve (u,v) élt $v \rightsquigarrow u \rightarrow v$ egy irányított kör.

Pl.: Az alábbi gráf mélységi bejárása során a (3,6)-os élt visszaélként azonosítottuk, tehát berajzolható egy irányított kör: 6,9,7,3.



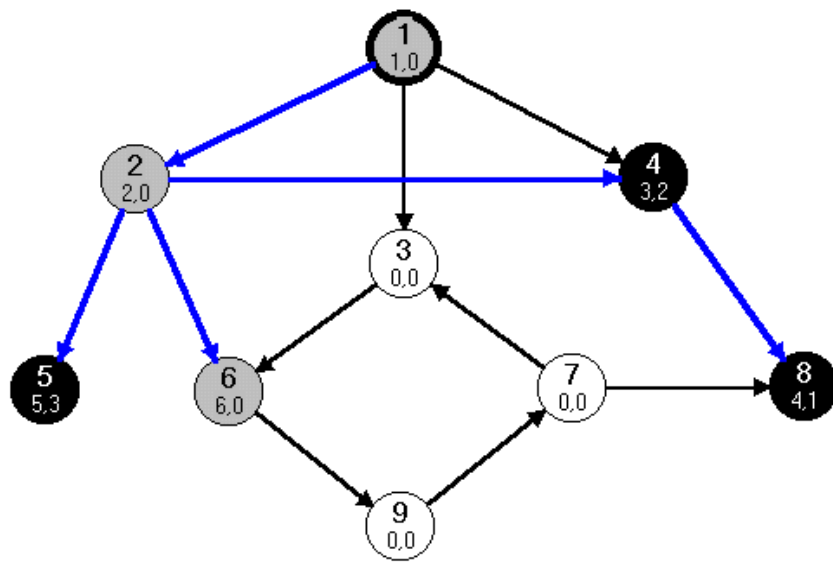
8.2.3. Állítás [2]: G gráf **nem DAG** $\Rightarrow \forall$ mélységi bejárása során **találunk visszaélt**

Bizonyítás: G nem DAG, azaz van benne irányított kör, legyen egy

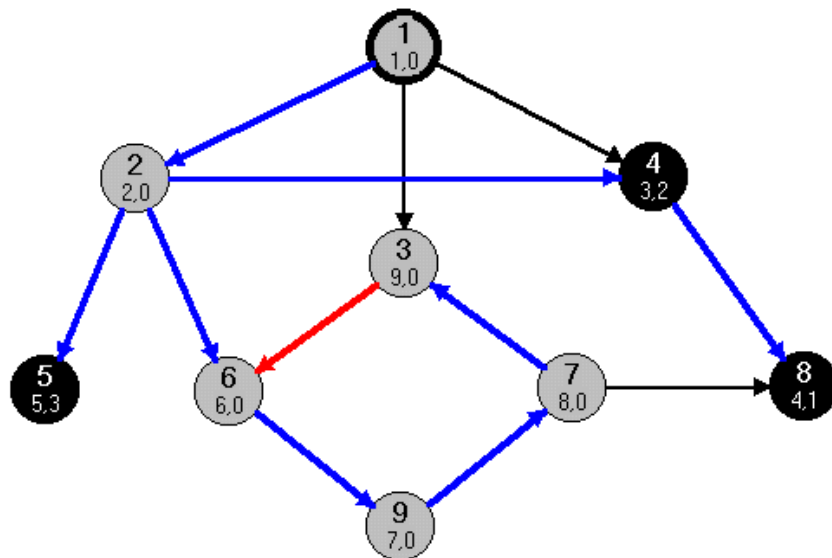


irányított kör, és legyen v a mélységi bejárás során elsőnek elért csúcs \Rightarrow a kör mentén a v kivételével minden csúcs fehér ebben a pillanatban. A kör mentén, vagy kis kerülő úton, de fehér csúcsok mentén eljutunk v -ből u -ba (ha van $v \rightsquigarrow u$ út \Rightarrow van csupa fehér csúcsból álló $v \rightsquigarrow u$ út is) $\Rightarrow (u, v)$ visszaél lesz.

Az alábbi példában a 6-os csúcs játssza a v csúcs szerepét. A kör menti fehér csúcsok: 9,7,3



A körmentén, fehér csúcsokon át, eljutunk a 3-as csúcsba, ahol (3,6)-os élt visszaélként azonosítjuk.



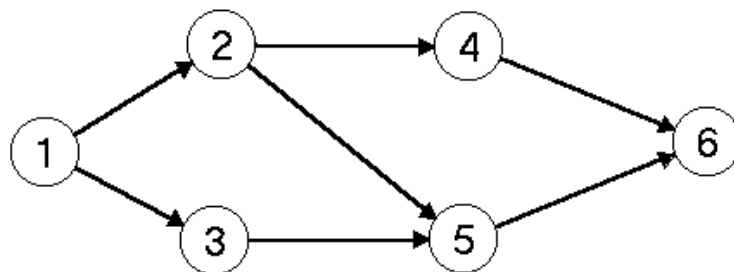
„Tehát egy gráfból $O(n+e)$ idő alatt eldönthető, hogy DAG-e, a mélységi keresés felhasználásával. Nem kell mást tenni, mint a mélységi keresés során figyelni az éltípusokat, ha nem találunk visszaélt, akkor a gráf DAG.” [2]

8.3 DAG topologikus rendezése

Az 1.4. bevezető fejezetben már felvetettünk egy gyártástechnológia előállításának nevezett problémát, amelyre most próbálunk megoldást mutatni.

8.3.1. Definíció [2]: Legyen $G=(V,E)$ irányított, véges gráf, továbbá legyen $n=|V|$. G csúcsainak egy v_1, \dots, v_n felsorolása, G egy **topologikus rendezése**, ha $\forall x \rightarrow y \in E$ él esetén a felsorolásban x előbb áll, mint y , azaz $x=v_i$ és $y=v_j$, akkor $i < j$.

Pl.: Az alábbi gráf egy topologikus rendezése: 1,2,3,4,5,6



8.3.2. Lemma [2]: $G=(V,E)$ irányított gráf DAG $\Rightarrow \exists u \in V$ csúcs, **amelybe nem fut él**.

Bizonyítás [2]: Indirekt tegyük fel, hogy nem létezik ilyen csúcs. Ekkor vegyünk egy csúcsot, befutó élén hátráljunk, azonban minden csúcsnak van befutó éle \Rightarrow végtelen hátrálás \Rightarrow kör (ami ellentmondás)

8.3.3. Tétel [2]: G -nek \exists topologikus rendezése $\Leftrightarrow G$ DAG

Bizonyítás [2]:

\Rightarrow : G -nek \exists topologikus rendezése, és indirekt tegyük fel, hogy G nem DAG, azaz van benne irányított kör, ami ellentmondás, mivel a kör mentén nem lehet alkalmas sorrendet definiálni.

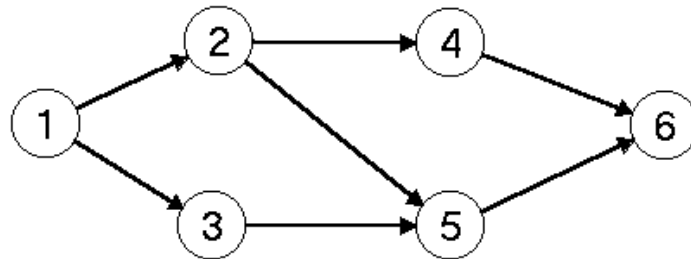
\Leftarrow : Teljes indukcióval

$n=1$ esetén triviális.

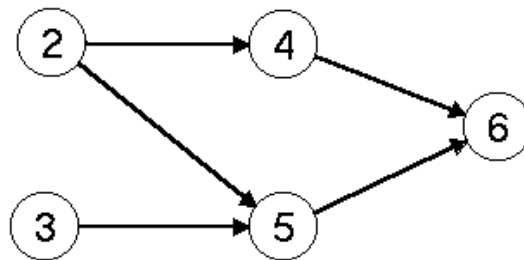
Tegyük fel, hogy $n-1$ pontú DAG-okra igaz az állítás. Legyen $G_n=(V,E)$ $n=|V|$ pontú DAG \Rightarrow

8.3.2. Lemma szerint, $\exists u \in V$ csúcs, amelyben nem megy él. \Rightarrow Töröljük u -t és a belőle kimenő éleket. Így kapjuk G_{n-1} $n-1$ pontú DAG-ot. \Rightarrow Az indukciós feltevés szerint G_{n-1} -nek \exists topologikus rendezése $v_1, \dots, v_{n-1} \Rightarrow u, v_1, \dots, v_{n-1}$ egy topologikus rendezése lesz G_n -nek

Pl.: Legyen G_n gráfunk:



Az G_n gráf 1-es csúcsa olyan, amelybe nem fut él. Töröljük az 1-es csúcsot! Így megkapjuk G_{n-1} gráfot:



ami az indukciós feltétel szerint DAG, így létezik topologikus rendezése: $2,3,4,5,6 \Rightarrow G_n$ -nek topologikus rendezése az $1,2,3,4,5,6$ sorozat, mivel nem megy él az 1-es csúcsba, így az 1-es csúcsot tehetjük a sorozat elejére.

Következmény: Az indukció megad egy algoritmust [2]:

Q : Sor adatszerkezet, kezdetben üres

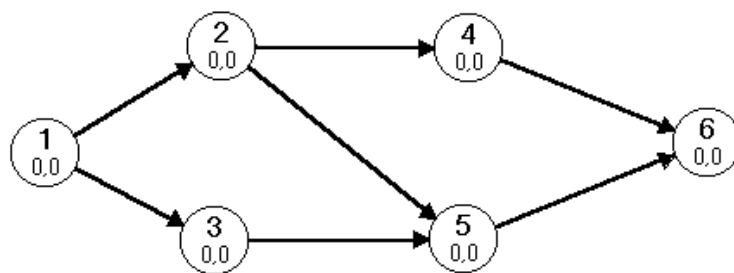
- 1) Q -ba berakjuk azon csúcsokat, amelybe nem megy él
- 2) Ha Q üres \Rightarrow KÉSZ különben $u := \text{First}(Q)$ és $\text{Write}(u)$
- 3) Töröljük G -ből $(u,v) \in E$ éleket. Ha v -be most már nem megy él $\Rightarrow \text{In}(Q,v)$
- 4) GOTO 2

DAG topologikus rendezése mélységi bejárás segítségével [2]:

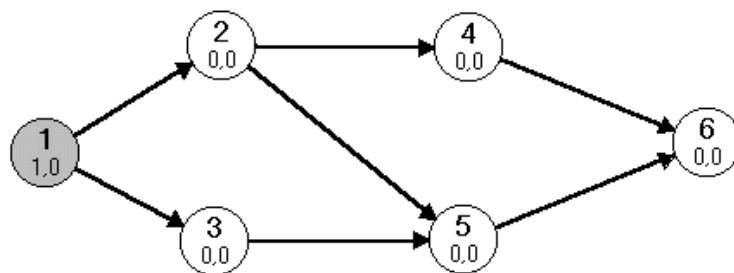
Futassuk le a mélységi bejárást a G DAG-on, majd a csúcsokat írjuk ki a csúcsok befejezési számainak ($\text{bszám}[u]$) csökkenő sorrendjében.

Megvalósítás: G mélységi bejárása során, amikor egy csúcsot elhagyunk, rakjuk a csúcs címkéjét egy veremben, majd a bejárás befejeztével ürítsük ki vermet. A bejárás alatt a DAG tulajdonságot is ellenőrizzük.

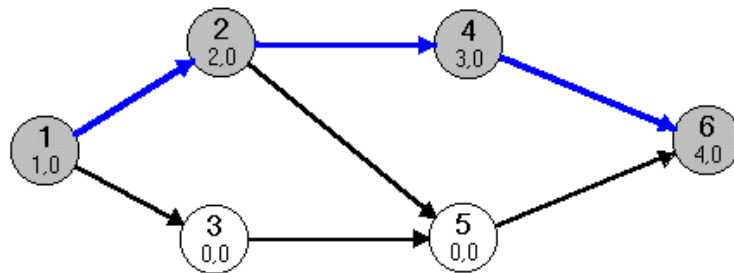
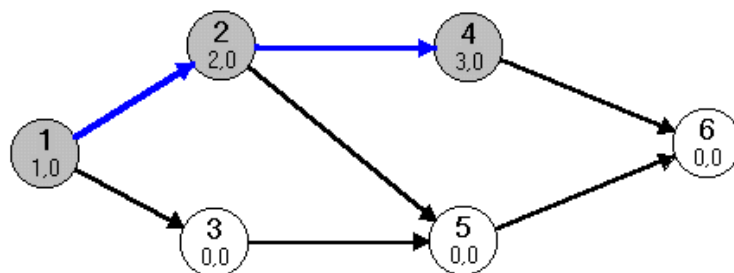
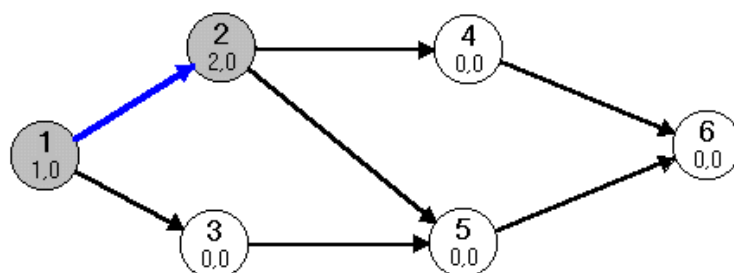
Most vizsgáljuk meg az algoritmus működését **ADS szinten**, egy példán!



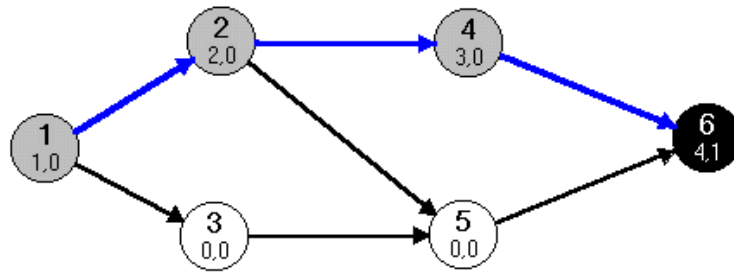
A mélységi bejárással elindulunk az 1-es csúcsból.



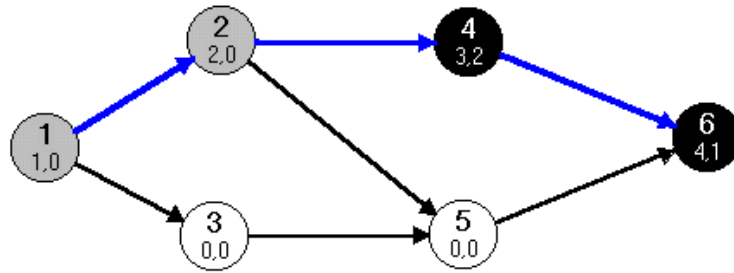
Majd néhány lépés után eljutunk a 6-os csúcsba, miközben csupa faéleket azonosítunk.



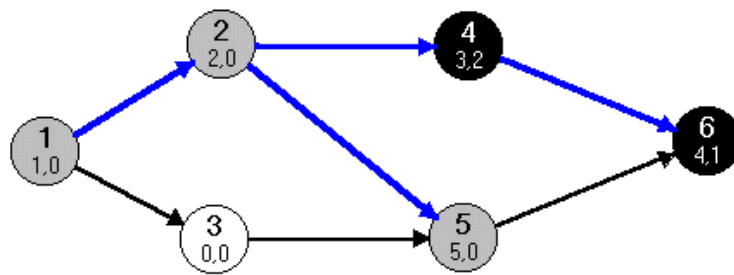
Majd elsőként a 6-os csúcs bejárását fejezzük be, tehát a 6-ost bedobjuk a V verembe: $V=[6$.



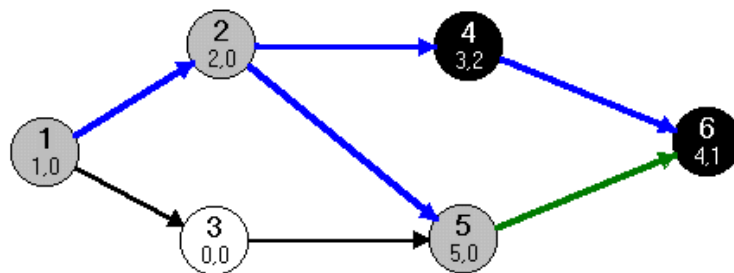
A következő lépésben a 4-es csúcsot fejezzük be. $V=[6,4$.



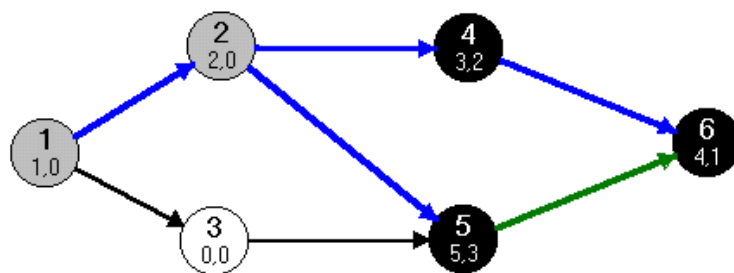
Majd a 2-es csúcsból ellátogatunk az 5-ös csúcsba, miközben a verem változatlan marad.



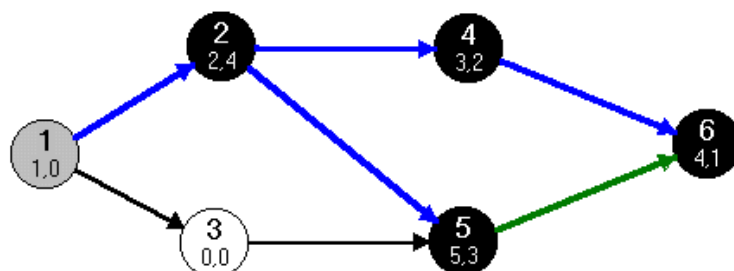
Az 5-ös csúcsból kimenő élt keresztélként azonosítjuk.



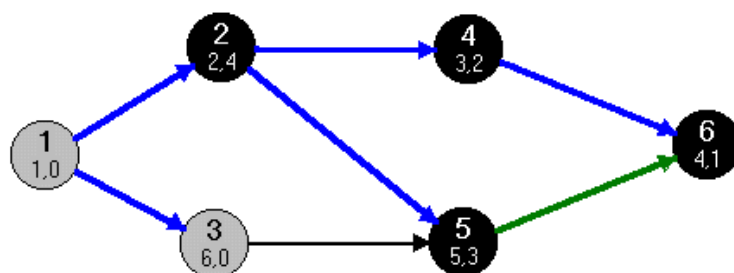
Befejezzük az 5-ös csúcs bejárását is, tehát a verembe dobjuk: $V=[6,4,5]$.



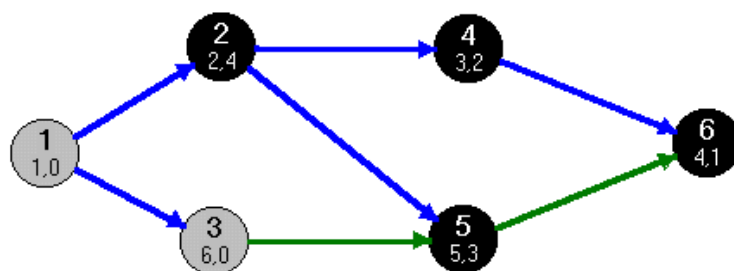
Majd befejezzük a 2-es csúcsot is, és a vermet a 2-essel bővítjük: $V=[6,4,5,2]$.



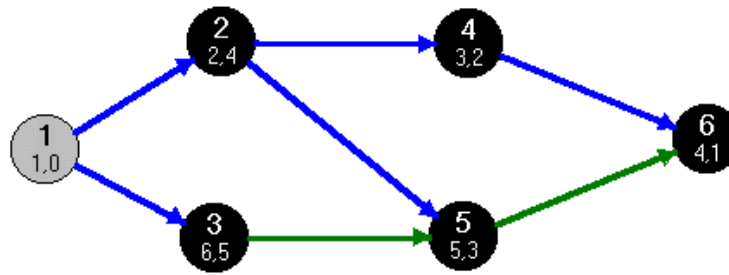
Majd az 1-es csúcsból elérjük a 3-as csúcsot.



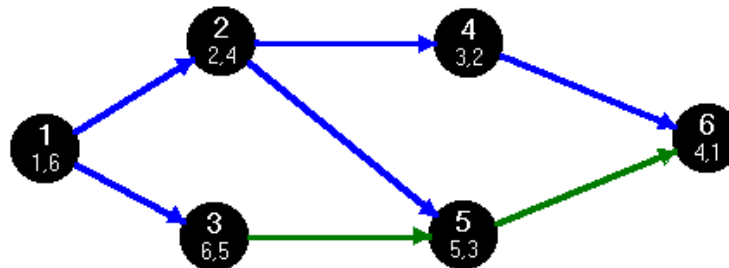
Újabb keresztelt azonosítunk: (3,5)



Befejezzük a 3-as csúcs bejárását is. $V=[6,4,5,2,3]$.



Majd a bejárás utolsó csúcsaként elhagyjuk az 1-es csúcsot is. A veremben van a gráf összes csúcsa a befejezésük szerint: $V=[6,4,5,2,3,1]$.



Menet közben nem találtunk visszaélt, tehát a DAG tulajdonságot rendben találtuk. Végül a verem tartalmát kiírjuk: 1,3,2,5,4,6, amellyel megkapjuk a G egy topologikus rendezését.

8.3.4. Állítás [2]: A fenti eljárás $G=(V,E)$ DAG egy topologikus rendezését állítja elő.

Bizonyítás: Azt kell belátni, hogy $(u,v) \in E \Rightarrow bszám[u] > bszám[v]$

Amikor (u,v) élt vizsgáljuk, akkor v fehér vagy fekete (szürke esetén visszaél lenne, de G DAG)

v fehér $\Rightarrow v$ leszármazottja u -nak $\Rightarrow bszám[u] > bszám[v]$ (8.1.4. Állítás)

v fekete $\Rightarrow v$ -t már megvizsgáltuk és elhagytuk, míg u -t még nem $\Rightarrow bszám[u] > bszám[v]$

Műveletigény [2]:

Mélységi bejárás: $O(n+e)$, a verem műveletek: $\Theta(n) \Rightarrow T(n) = O(n+e)$

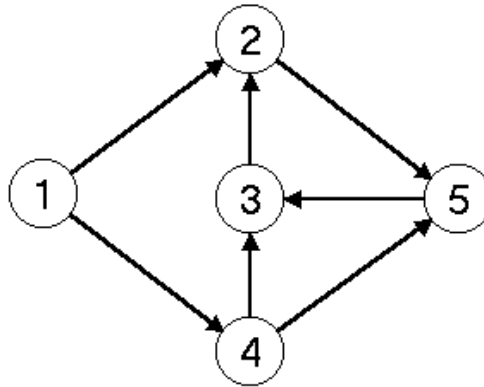
8.4 Erősen összefüggő komponensek meghatározása

Ne feledjük, hogy korábbi megállapodásunknak megfelelően, a fejezet során említett G gráf legyen $G=(V,E)$ irányított, véges gráf.

8.4.1. Definíció [2]: G gráf összefüggő, ha G irányítás nélkül összefüggő.

8.4.2. Definíció [2]: G gráf erősen összefüggő, ha $\forall u, v \in V$ esetén létezik $u \rightsquigarrow v$ út a gráfban.

A definíciók különbségének érzékeltetésére nézzünk egy példát (Rónyai-Ivanyos-Szabó: *Algoritmusok* c. tankönyvből) [2]. Tekintsük egy város úthálózatát, ahol egyirányú utcák is előfordulnak. A gyalogosok számára az egyirányú utca nem jelent megkötést, tehát ők tekinthetik irányítatlannak az utcákat, míg az autósok számára az utcák irányítottak. Egy városrész összefüggő, ha gyalogosan bármely pontjából bármely pontjába eljuthatunk, míg a városrész erősen összefüggő, ha ugyanez megtehető autóval is. Érezhető, hogy az erősen összefüggőség valóban erősebb követelmény, mint az egyszerű összefüggőség. Például az alábbi gráf összefüggő, de nem erősen összefüggő (a 2-es csúcsból nem lehet eljutni az 1-es csúcsba stb.):



8.4.3. Definíció [2]: Bevezetjük \approx relációt V -n. Legyenek $u, v \in V$ csúcsok, és $u \approx v$ reláció teljesül, ha $\exists u \rightsquigarrow v$ és $\exists v \rightsquigarrow u$ utak a gráfban.

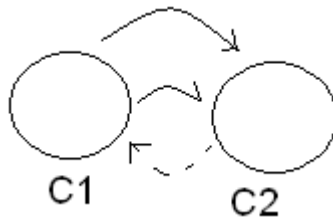
8.4.4. Állítás 2]: $A \approx$ reláció ekvivalencia reláció. (Bizonyítás: triviális)

Következmény [2]: $A \approx$ reláció osztályozza a V csúcshalmazt.

8.4.5. Definíció [2]: $A \approx$ reláció ekvivalencia osztályait a G erős komponenseinek nevezzük.

8.4.6. Állítás [2]: G gráf két erős komponense között az **élek csak egy irányba mehetnek**.

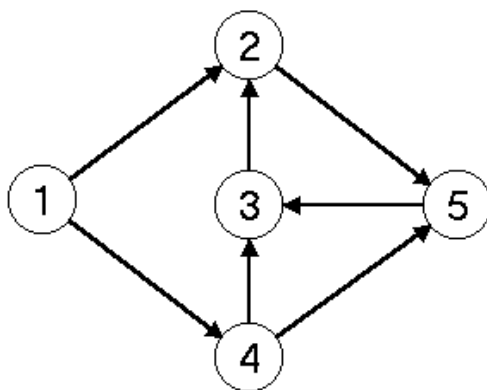
Bizonyítás [2]: Indirekt tegyük fel, hogy két erős komponens között oda-vissza is mehetnek élek.



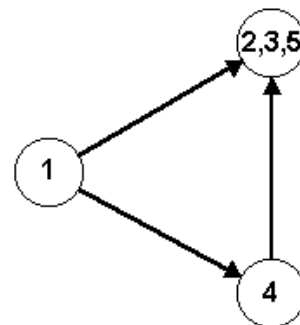
Ekkor $u \in C1$ és $v \in C2$ csúcsok között $\exists u \rightsquigarrow v$ és $\exists v \rightsquigarrow u$ utak, ugyanis az erős komponenseken belül definíció szerint, $C1$ és $C2$ között pedig az indirekt feltevés szerint létezik út $\Rightarrow u \approx v$, ami ellentmondás.

8.4.7. Definíció [2]: G **redukált gráfja** olyan irányított gráf, amelynek csúcsai G erős komponensei, és a redukált gráf két csúcsa között, akkor halad él, ha csúcsoknak megfelelő G erős komponensei között halad él, továbbá az él irányítása megegyezik az erős komponensek között haladó él(ek) irányításával.

Pl.:



G gráf



G redukált gráfja

8.4.8. Állítás [2]: G redukált gráfja DAG.

Bizonyítás [2]: Indirekt tegyük fel, hogy a redukált gráf nem DAG, azaz létezik benne irányított kör. Legyen egy ilyen kör $C1 \rightarrow C2 \rightarrow \dots \rightarrow Ck \rightarrow C1 \Rightarrow$ a kör mentén lévő komponensek kölcsönösen elérhetők $\Rightarrow C1 \approx C2 \approx \dots \approx Ck$ ami ellentmondás.

Algoritmus [2]:

- 1) G -t bejárjuk mélységi bejárással, a csúcsokat a befejezési számok sorrendjében kiírjuk egy listába.
- 2) Transzponáljuk G -t (fordítsuk meg G éleinek irányítását) $\Rightarrow G^T$
- 3) Bejárjuk G^T transzponáltat mélységi bejárással az 1-es pontban elkészült lista csökkenő sorrendjében.

A 3-dik pontban kapott mélységi erdő fái adják a G erős komponenseit.

Megjegyzések:

- a) Ha G DAG, akkor az algoritmus 3-dik pontjában említett bejárési sorrend nem más, mint a G egy topologikus rendezése.

b) A topologikus rendezéshez hasonlóan, az 1-es pontban a lista helyett használhatunk egy vermet a befejezési számok szerinti sorrend fordított előállítására.

8.4.9. Állítás: Bármely mélységi bejárás során, egy erős komponens összes csúcsa **ugyanabba a mélységi fába** kerül.

Bizonyítás: Legyen x az a csúcs, amelyet a bejárás során először érünk el az erős komponens csúcsai közül. \Rightarrow A komponens összes többi csúcsa még fehér. Továbbá erős komponensről van szó, az összes csúcsba vezet út. \Rightarrow Az erős komponens összes többi csúcsába vezet fehér csúcsokból álló út. \Rightarrow (8.1.5. Fehér út tétel) Az erős komponens összes többi csúcsa x leszármazottja lesz a mélységi fában.

8.4.10. Tétel [2]: Futtassuk le a fenti algoritmust G -n. Legyenek $x, y \in V$ G csúcsai, és tekintsük az algoritmus 3-dik pontjában kapott mélységi fákat. Ekkor $x \approx y \Leftrightarrow x$ és y **ugyanabban a mélységi fában vannak.**

Bizonyítás [2]:

\Rightarrow : Az erős komponensek pontjai minden mélységi bejárásnál ugyanabba a fába esnek (8.4.9. Állítás), továbbá G és G^T erős komponensei azonosak $\Rightarrow x$ és y ugyanabba a mélységi fába esnek G^T mélységi bejárása után.

\Leftarrow : Tegyük fel, hogy x és y ugyanabban a fában vannak. Legyen v ennek a fának a gyökere.



Mivel x leszármazottja v -nek $\Rightarrow \exists v \leadsto x$ út G^T -ben $\Rightarrow \exists L = x \leadsto v$ út G -ben

Legyen x' L -nek a legkisebb mélységi számú pontja az algoritmus 1-beli bejárásánál $\Rightarrow v$ leszármazottja x' -nek (Mivel x' elérésekor L minden csúcsa még fehér, alkalmazható a 8.1.5. Fehér út tétel.). $\Rightarrow \exists x' \leadsto v$ G -ben. Az x' gyökerű részében $bszám[x']$ a legnagyobb befejezési szám (8.1.4. Állítás), tehát x' -nek v -nél előbb kell lennie a fordított $bszám$ listában, azonban v mégis előbb van a listában, mivel a G^T -ben való bejárásnál gyökér lett $\Rightarrow x'=v \Rightarrow$ Az L úton $mszám[v]$ a legkisebb mélységi szám, és $bszám[v]$ a legnagyobb befejezési szám (az első bejárás szerint) $\Rightarrow L$ pontjai leszármazottai v -nek $\exists v \leadsto x$ út G -ben.

Mivel $\exists L = x \leadsto v$ és $\exists v \leadsto x$ utak G -ben $\Rightarrow v \approx x$

y -ra hasonlóan belátható, hogy $v \approx y$

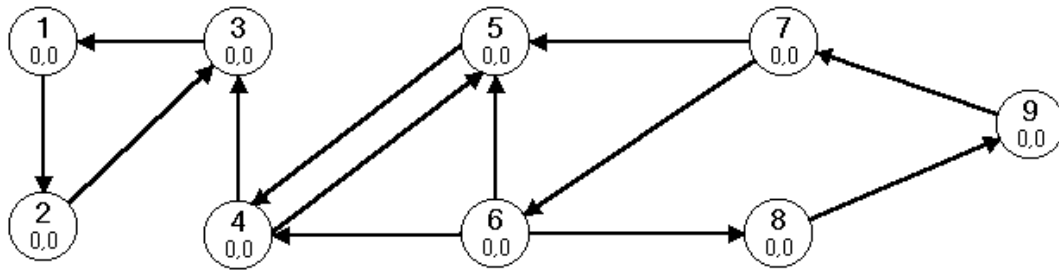
Azt kaptuk, hogy $v \approx x$ és $v \approx y$, továbbá \approx reláció ekvivalencia reláció $\Rightarrow x \approx y$

Műveletigény [2]:

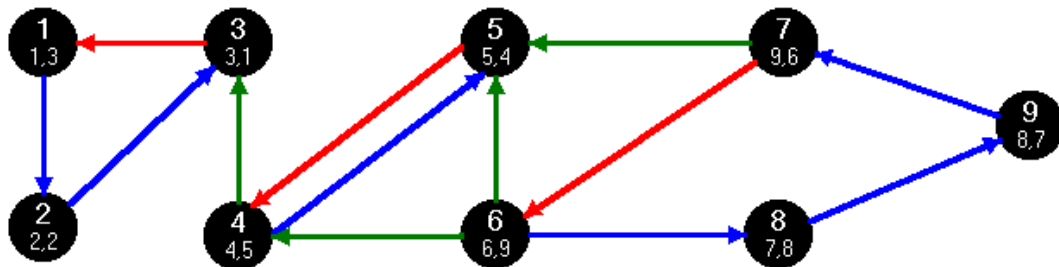
Mélységi bejárás: $O(n+e)$, a gráf megfordítása $O(e)$, a verem műveletek: $\Theta(n) \Rightarrow T(n) = O(n+e)$

Most vizsgáljuk meg az algoritmus működését egy példán ADS szinten:

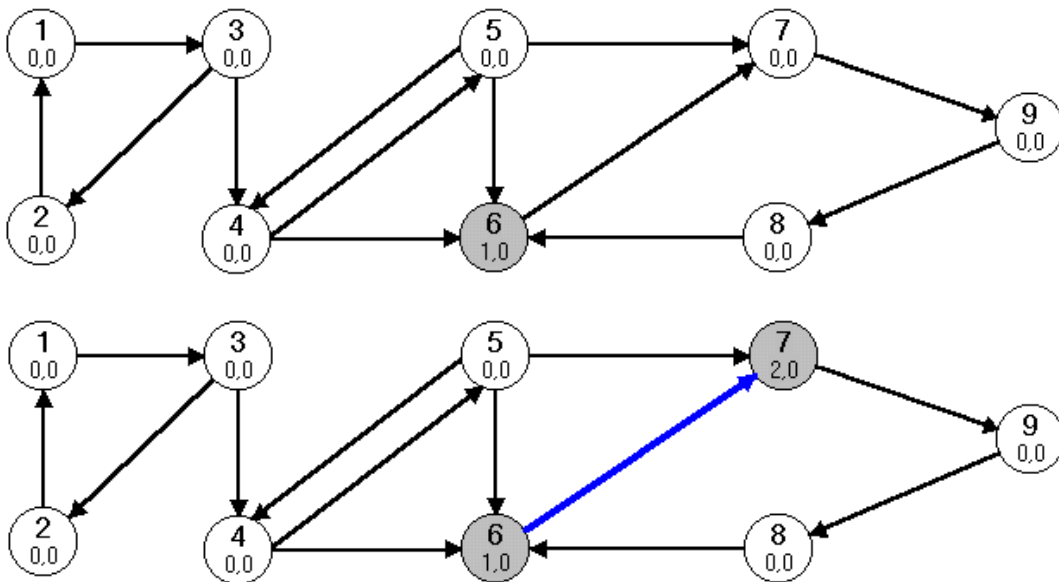
Tekintsük az alábbi gráfot:

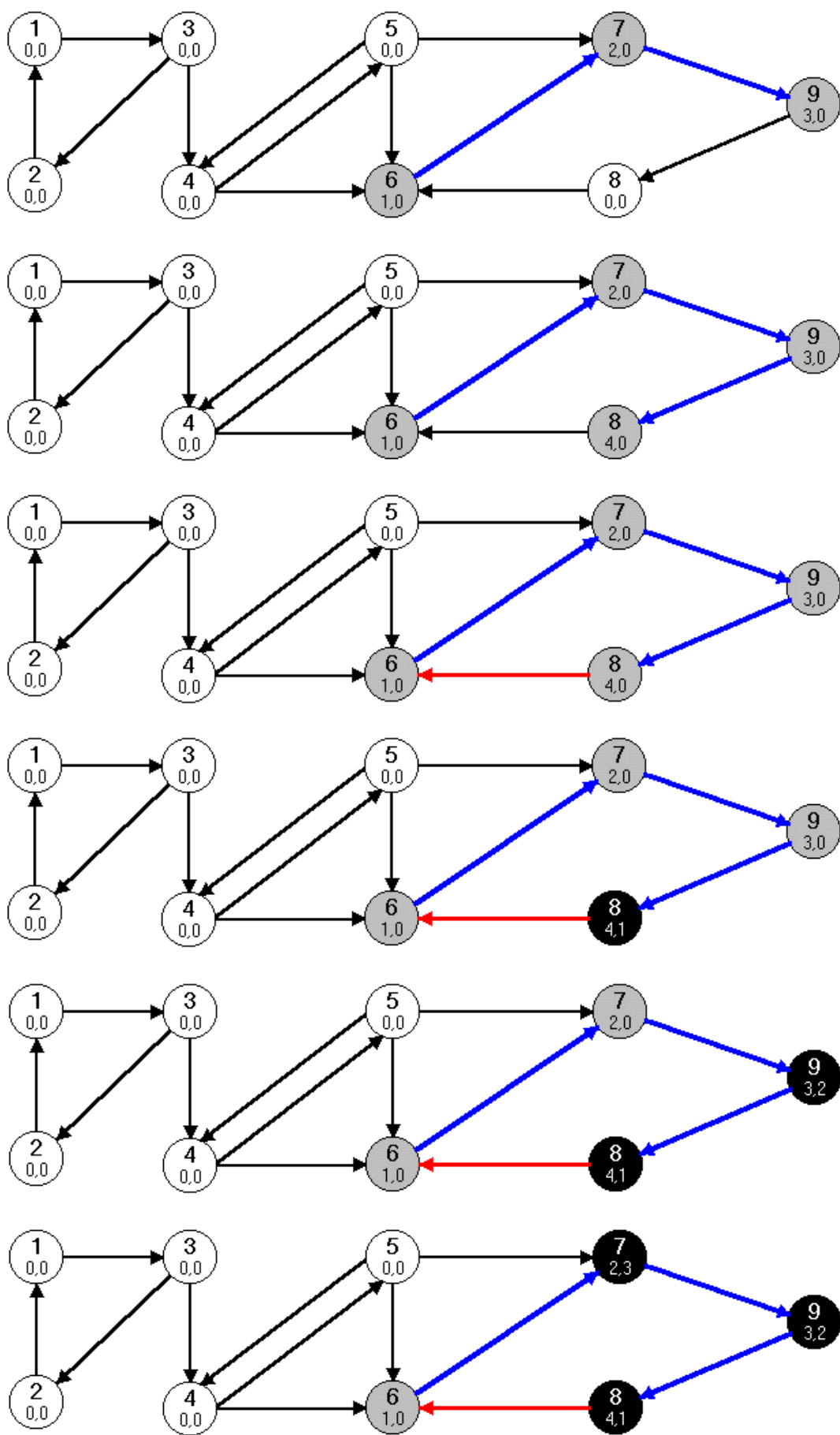


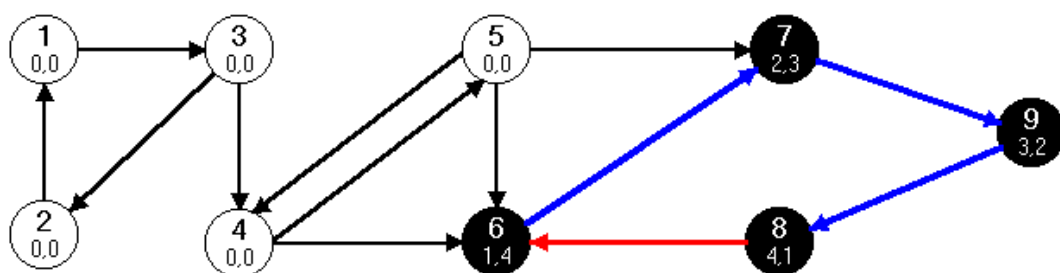
Futtassunk le egy mélységi bejárást a gráfon. A csúcsok feldolgozási sorrendje legyen a csúcsok címkéje szerint rendezett! Az alábbi ábrán látható az első mélységi bejárás eredménye. A befejezési számok szerint csökkenően a csúcsok sorrendje: 6,8,9,7,4,5,1,2,3.



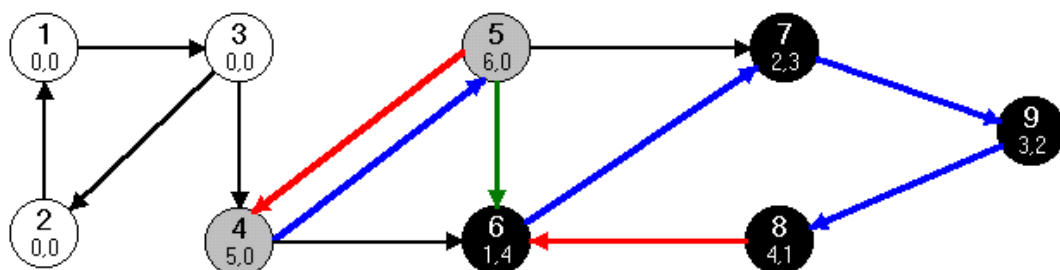
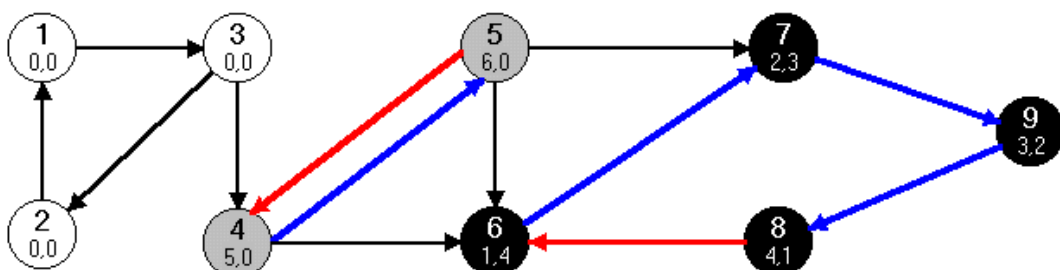
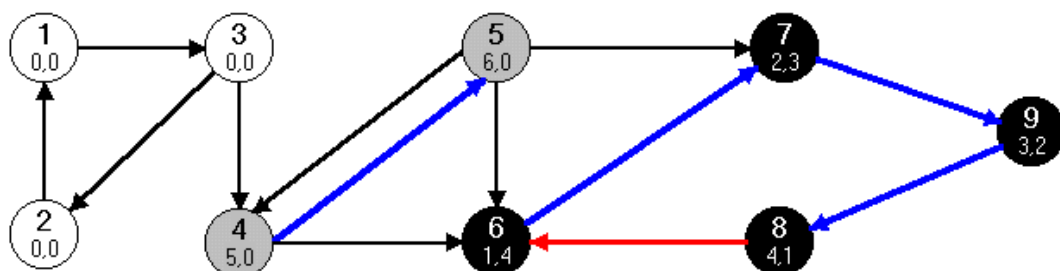
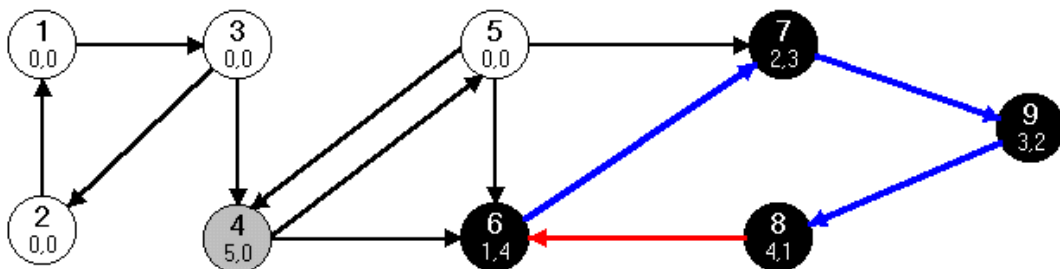
Majd a gráf fordítottján kell a mélységi keresést lefuttatni a csúcsok fent említett sorrendjében. Tehát az eljárás a 6-os csúcsból indul és bejárjuk a 6-osból elérhető csúcsokat.

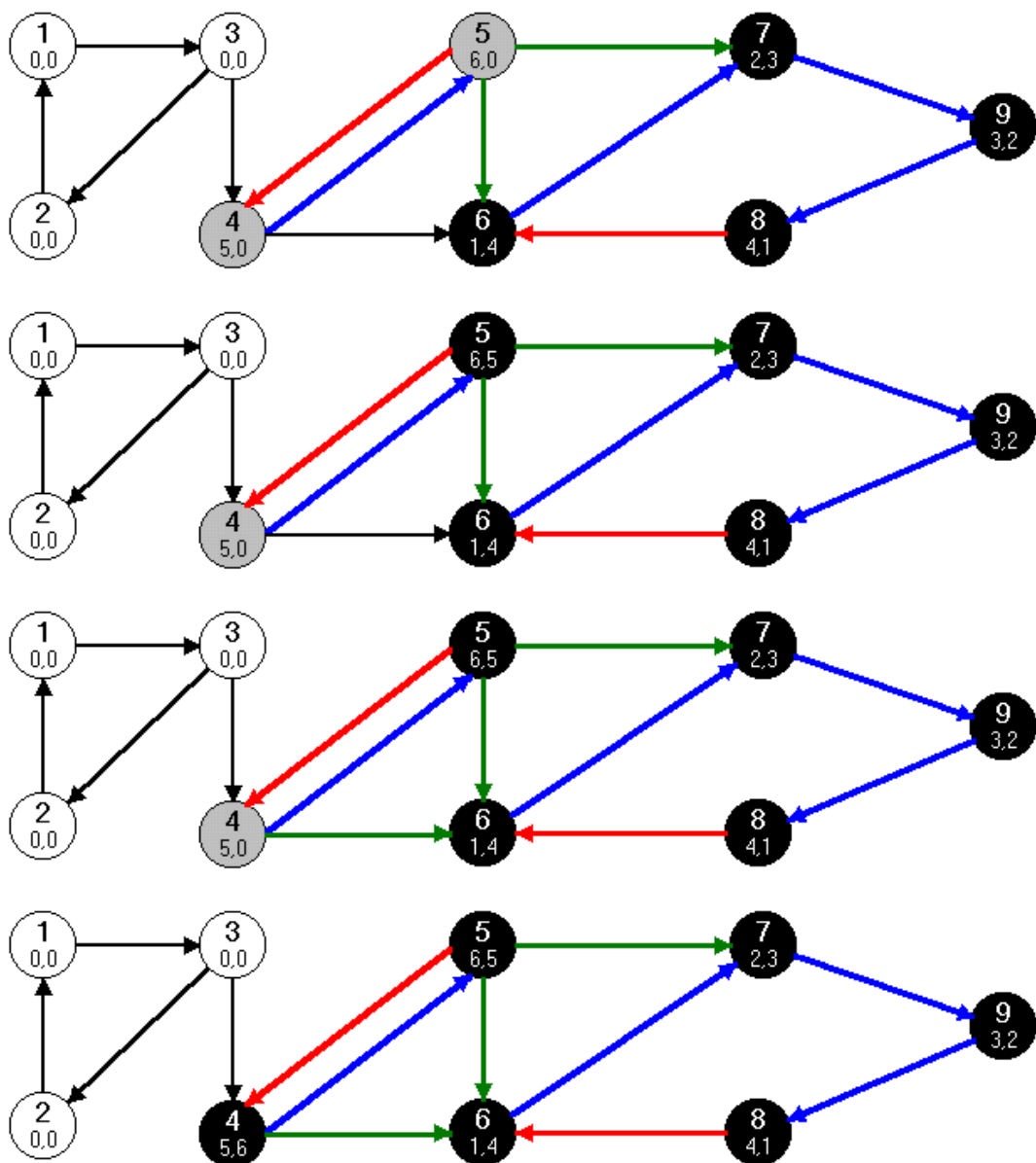




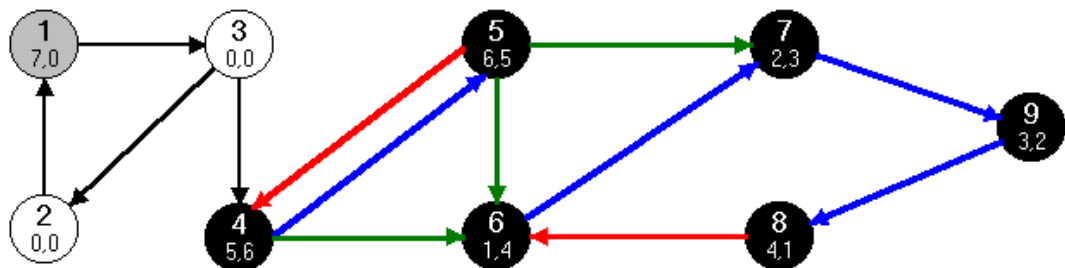


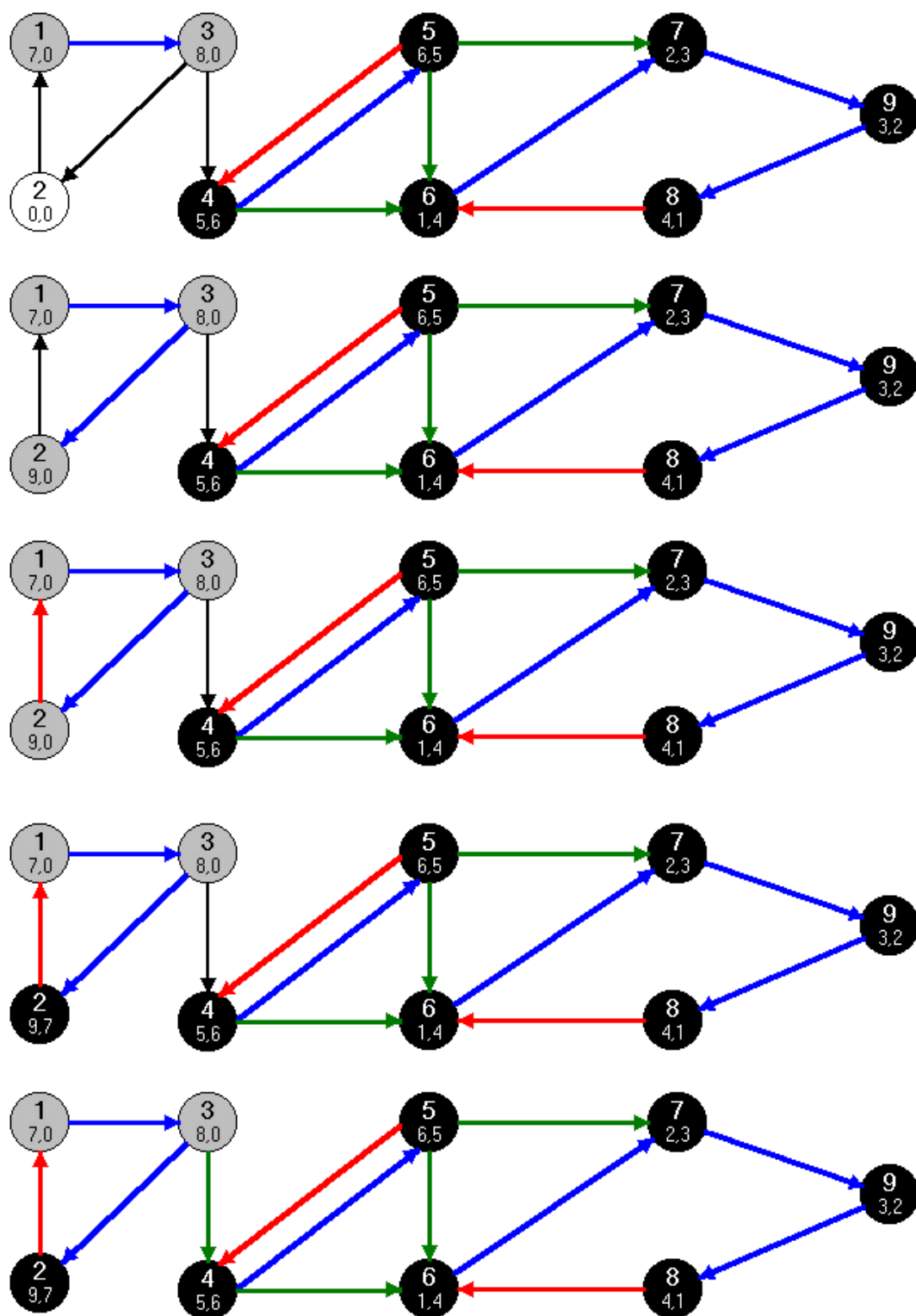
Miután a 6-os csúcs bejárását befejeztük, a bejárési sorrendet meghatározó lista (6,8,9,7,4,5,1,2,3) következő fehér csúcsából (a 4-es csúcsból) folytatjuk a bejárást.

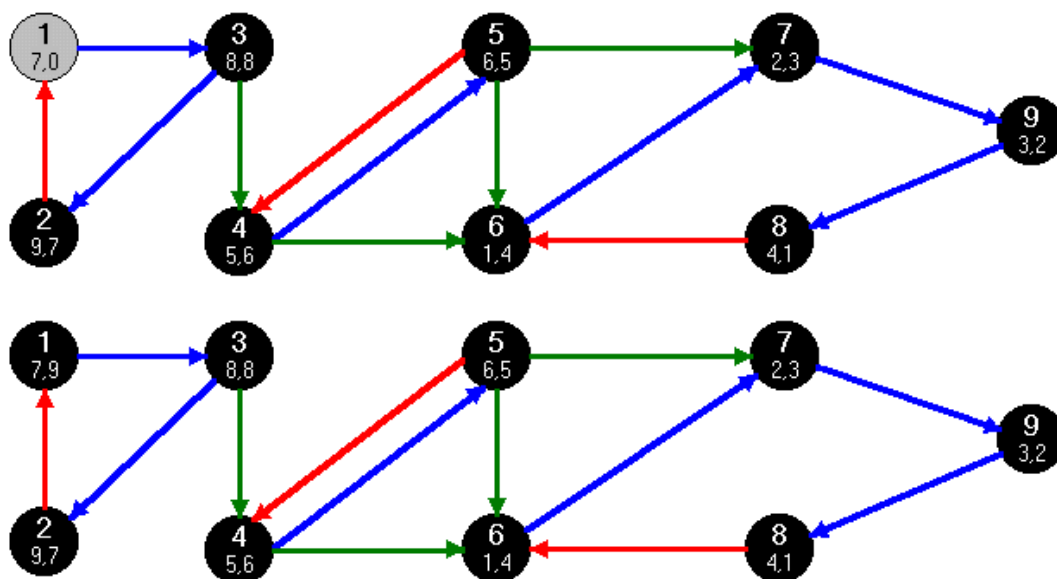




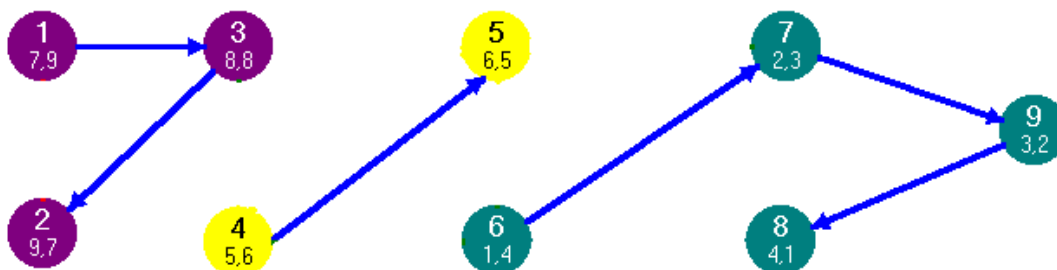
Miután a 4-es csúcs bejárását befejeztük, a bejárési sorrendet meghatározó lista (6,8,9,7,4,5,1,2,3) következő fehér csúcsából (az 1-es csúcsból) folytatjuk a bejárást.







Miután befejeztük a fordított gráf mélységi bejárását is, az erős komponenseket az azonos mélységi fába került csúcsok alkotják. A példában 3 erős komponenst sikerült azonosítani: $C1=\{1,3,2\}$, $C2=\{4,5\}$, $C3=\{6,7,9,8\}$



8.5 Ellenőrző kérdések

1. Adja meg a mélységi bejárás elvét!
2. Hogyan színezzük a csúcsokat a bejárás közben?
3. Mi a mélységi és befejezési szám?
4. Mi a mélységi fa és mélységi erdő?
5. Milyen színű csúcsból indítjuk el a rekurzív eljárást?
6. Mi a leszármazott definíciója?
7. Milyen kapcsolat van a leszármazottság és a mélységi számok között?
8. Mi a szükséges és elégséges feltétele a leszármazottságnak?
9. Adja meg az éltípusok definícióit!
10. Hogyan határozhatjuk meg az éltípusokat?
11. Mi a mélységi bejárás műveletigénye?
12. Adja meg a DAG tulajdonság definícióját!
13. Hogyan határozhatjuk meg a DAG tulajdonságot?

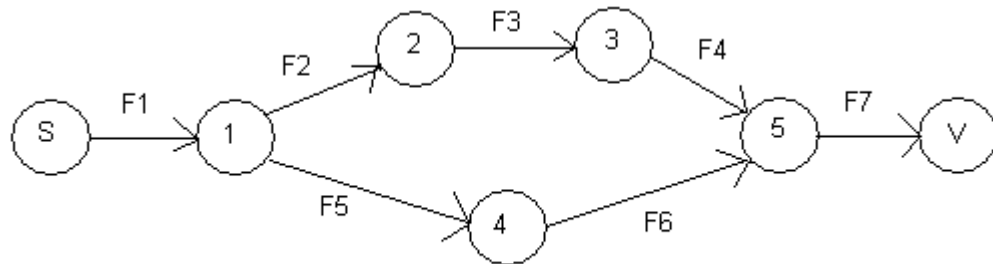
14. Fontos-e, hogy a DAG tulajdonság meghatározásánál melyik csúcsból indítjuk a mélységi bejárást?
15. Adja meg a topologikus rendezés definícióját!
16. Mikor lehet egy gráfnak topologikus rendezése?
17. Hogyan lehet a topologikus sorrendet előállítani?
18. Mikor mondjuk, hogy egy gráf erősen összefüggő?
19. Mi az erős komponensek definíciója?
20. Mi az a redukált gráf?
21. Hogyan határozhatjuk meg egy gráf erős komponenseit?
22. Mi a műveletigénye az erős komponenseket meghatározó algoritmusnak?

8.6 Gyakorló feladatok

1. Írja fel a mélységi bejárást (rekurzív változat)
 - a csúcsmátrixos ábrázolás esetén!
 - b éllistas ábrázolás esetén!
2. Elegendő-e a mélységi bejárás során két szintet használni?
3. Irányított gráf esetén, hogyan lehet a színezés alapján azonosítani az egyes éltípusokat?
4. Irányítatlan gráfoknál milyen éltípusok lehetnek, és mélységi bejárás során hogyan azonosíthatjuk azokat?
5. Lehet-e olyan eset a mélységi bejárás során, hogy egy irányított gráf u csúcsa a keresés után egy olyan mélységi fába kerül, amely csak az u csúcsból áll, annak ellenére, hogy G -ben vannak u -ba bevezető és u -ból kivezető élek?
6. Módosítsa a mélységi bejárást úgy, hogy kiírja az éleket és azok típusát!
 - a Irányított gráf esetén.
 - b Irányítatlan gráf esetén.
7. Írja fel a mélységi bejárás iteratív változatát!
8. Adott egy $G = (V, E)$ irányítatlan gráf. Adjon algoritmust, amely G komponenseit kiszínezi! (Az azonos komponensbe eső csúcsokat azonos, a különböző komponensbe eső csúcsokat különböző színűre.)
9. Adott egy $G = (V, E)$ irányított gráf. Adjon algoritmust, amely G összefüggő (nem az erősen összefüggő) komponenseit kiszínezi! (Az azonos komponensbe eső csúcsokat azonos, a különböző komponensbe eső csúcsokat különböző színűre.)

10. A G gráfon lefuttattuk a mélységi bejárást, de sajnos a gráfot és a Pi tömböt kitöröltük, csak az $mszám$ és a $bszám$ tömbjeink maradtak meg. Adjon algoritmust, amely előállítja a mélységi feszítő erdőt, azaz a Pi tömböt. [10]
11. Adjon algoritmust annak eldöntésére, hogy egy $G=(V,E)$ gráfban bármely $u,v \in V$ csúcsra legfeljebb egy egyszerű út vezet u -ból v -be!
- G irányítatlan gráf esetén.
 - G irányított gráf esetén.
12. Adjon algoritmust, amely egy gyártási folyamat ütemterv grájában megkeres egy kritikus utat! Az előadáson elhangzott algoritmusokat „függvényként” beillesztheti a struktogrammba, ezek kifejtése nem kötelező. Ütemterv gráf: olyan gráf, melynek csúcsai a gyártási folyamat egyes állapotai, a gráf élei, pedig tevékenységek, amelyek a tevékenység időtartamával vannak súlyozva. A gráf tartalmaz két kitüntetett állapotot (csúcsot), a start- (S) és a végállapot (V), amelyek a gyártási folyamat kezdetét illetve végét jelölik. Kritikus út: S -ből V -be vezető maximális időtartamú („maximális költségű”) út. Tehát a „termék” gyártási ideje legalább akkora, mint a kritikus út időtartama („hossza”).

Pl.: Pudingos palacsinta készítésének ütemterve:



- F1: alapanyagok beszerzése (120 perc)
 F2: tészta bekeverése (15 perc)
 F3: serpenyő felmelegítése (3 perc)
 F4: tészta kisütése (60 perc)
 F5: töltelék (puding) megfőzése (20 perc)
 F6: töltelék pihentetése, hűtése (60 perc)
 F7: palacsinták megtöltése (20 perc)

Kritikus út: F1,F5,F6,F7 (220 perc)

13. Adjon hatékony, az erőforrás gráf topologikus rendezésén alapuló, holtpont megelőző eljárást!
14. Adjon $O(n+e)$ futási idejű algoritmust egy irányított $G=(V,E)$ gráf komponens grájának a meghatározására! Ügyeljen arra, hogy legfeljebb egy él kösse össze az algoritmus által előállított komponens gráf csúcsai.
15. Adott gyárak, kereskedelmi és szolgáltató vállalatok halmaza. Azt mondjuk, hogy ha B cég vásárol valamilyen terméket (szolgáltatást). A cégtől, akkor A közvetlen gazdasági befolyással van B -re, mivel A termékei árának változása befolyással van B termékeinek az árára. Ezek után a közvetett gazdasági befolyásolás már könnyen

definiálható. Gazdaságilag kölcsönösen függő csoportosulásnak nevezzük a cégek azon halmazát, ahol bármely két cég közvetlen vagy közvetett módon gazdaságilag befolyásolja egymást. Adott a cégek egy véges halmaza. Határozzuk meg a gazdaságilag kölcsönösen függő csoportosulásokat!

16. Adott egy $G=(V,E)$ irányított, körmentes (DAG) véges gráf. Adjunk $O(n+e)$ futásidejű algoritmust, amely eldönti, hogy létezik-e a gráfban Hamilton út (olyan út, amely a gráf minden csúcsát pontosan egyszer tartalmazza).

8.7 Összefoglalás

- Mélységi stratégia: egy kezdőpontból kiindulva addig megyünk egy él mentén, ameddig el nem jutunk egy olyan csúcsba, amelyből már nem tudunk tovább menni, mivel nincs már meg nem látogatott szomszédja. Ekkor visszamegyünk az út utolsó előtti csúcsához, és onnan próbálunk egy másik él mentén tovább menni. Ha ezen az ágon is minden csúcsot már bejártunk, ismét visszamegyünk egy csúcsot, és így tovább.
- A bejárás során tároljuk el, hogy egy csúcsot hányadikként értünk el és tároljuk el, hogy hányadikként fejeztük be a csúcs, és a belőle elérhető csúcsok bejárását. Az említett számokat nevezzük mélységi (mszám), illetve befejezési számnak (bszám).
- A bejárás felépíti a mélységi fát/erdőt, ahol y csúcs leszármazottja x -nek, ha $\exists x \rightsquigarrow y$ T-beli irányított út a fában.
- Az élek osztályozása egy adott mélységi bejárás szerint, ahol legyen T a gráf egy mélységi erdője
 - a) faél, ha $x \rightarrow y$ éle T-nek.
 - b) előreél, ha $x \rightarrow y$ nem éle T-nek, de y leszármazottja x -nek T-ben, és $x \neq y$.
 - c) visszaél, ha x leszármazottja y -nak T-ben ($x \rightarrow y$ nem éle T-nek; ide tartozik $x=y$ hurokél is).
 - d) keresztél, ha x és y nem leszármazottai egymásnak T-ben.
- Az éltípusok azonosítása a bejárás során, ha a mélység bejárása során éppen az $(x,y) \in E$ vizsgálatánál tartunk:
 - a) faél, ha $mszám[y] = 0$.
 - b) előreél, ha $mszám[y] > mszám[x]$.
 - c) visszaél, ha $mszám[y] \leq mszám[x]$ és $bszám[y] = 0$.
 - d) keresztél, ha $mszám[y] < mszám[x]$ és $bszám[y] > 0$.
- A mélységi bejárás műveletigénye: $T(n)=O(n+e)$.
- DAG = irányított körmentes gráf
- Egy gráfból $O(n+e)$ idő alatt eldönthető, hogy DAG-e, a mélységi keresés felhasználásával. Nem kell mást tenni, mint a mélységi keresés során figyelni az éltípusokat, ha nem találunk visszaélt, akkor a gráf DAG.

- Topologikus rendezés: A gráf csúcsainak egy olyan felsorolása, ahol $\forall x \rightarrow y \in E$ él esetén a felsorolásban x előbb áll, mint y .
- G -nek \exists topologikus rendezése $\Leftrightarrow G$ DAG
- DAG topologikus rendezése mélységi bejárás segítségével: Futassuk le a mélységi bejárást a gráfon, majd a csúcsokat írjuk ki a csúcsok befejezési számainak (bszám[u]) csökkenő sorrendjében.
- G gráf erősen összefüggő, ha $\forall u, v \in V$ esetén létezik $u \rightsquigarrow v$ út a gráfban.
- Bevezetjük \approx relációt V -n. Legyenek $u, v \in V$ csúcsok, és $u \approx v$ reláció teljesül, ha $\exists u \rightsquigarrow v$ és $\exists v \rightsquigarrow u$ utak a gráfban. Ez ekvivalencia reláció, amely osztályozza a csúcsokat. Ezeket az osztályokat nevezzük erős komponenseknek.
- G redukált gráfja olyan irányított gráf, amelynek csúcsai G erős komponensei, és a redukált gráf két csúcsa között, akkor halad él, ha csúcsoknak megfelelő G erős komponensei között halad él, továbbá az él irányítása megegyezik az erős komponensek között haladó él(ek) irányításával.
- Erős komponensek a meghatározása:
 - 1) G -t bejárjuk mélységi bejárással, a csúcsokat a befejezési számok sorrendjében kiírjuk egy listába.
 - 2) Transzponáljuk G -t (fordítsuk meg G éleinek irányítását) $\Rightarrow G^T$
 - 3) Bejárjuk G^T transzponáltat mélységi bejárással az 1-es pontban elkészült lista csökkenő sorrendjében. Az így kapott mélységi erdő fái adják a G erős komponenseit.

9. Implementáció

9.1 Gráfok ábrázolása

A következőkben tekintsük át, hogy lehet C++ programozási nyelven, az STL (Standard Template Library) felhasználásával implementálni a gráfokat. Az implementáció során felhasználjuk az STL vektor, lista, sor és prioritásos sor sablonjait.

Szomszédsági-mátrixot vektorban a vektor sablon példányosítással definiálhatunk. Élsúlyozás nélküli esetben a mátrix elemei legyenek logikai értékek, élsúlyozott esetben pedig egész számok. (Lehetne valós is, de a példáinkban az egyszerűség kedvéért mindig egészeket használunk.). A változók definiálásakor megadhatjuk a csúcsok számát (lehet később is resize függvénnyel), ez lesz a „külső” vektor mérete. A mátrix sorait adó vektoroknak a méretét külön-külön egy ciklusban tudjuk beállítani.

```
#include <vector>

vector< vector<bool> > > g(CSÚCSOK_SZÁMA);
vector< vector<int> > > g1(CSÚCSOK_SZÁMA);

for(int i=0;i<g.size();++i)
    g[i].resize(g.size(),false);

for(int i=0;i<g1.size();++i)
    g1[i].resize(g1.size(),VEGTELEN);
```

Vektorban a vektor.

Itt állítjuk be a sorok méretét és feltöltjük kezdeti értékkel.

A szomszédsági lista megvalósítása vektorban a lista formájában történhet, ahogy az a 3. fejezet ábráin is látszik. Élsúlyozás nélküli esetben a lista elemei legyenek egész értékek (a célcúcs indexe). Élsúlyozott esetben a lista tartalmazza a célcúcs indexét és az él súlyát is. Erre létrehozhatunk egy típust (eltípus néven), és a szomszédsági listák ilyen típusú adatokat tárolhatnak.

```
#include <vector>
#include <list>

vector< list<int> > > g(CSÚCSOK_SZÁMA);

struct eltipus{
    int cel,suly;
    eltipus(int c=0,int s=0): cel(c),suly(s) {}
};

vector< list<eltipus> > > g1(CSÚCSOK_SZÁMA);
```

Vektorban a lista, élsúlyozás nélküli gráf esetén.

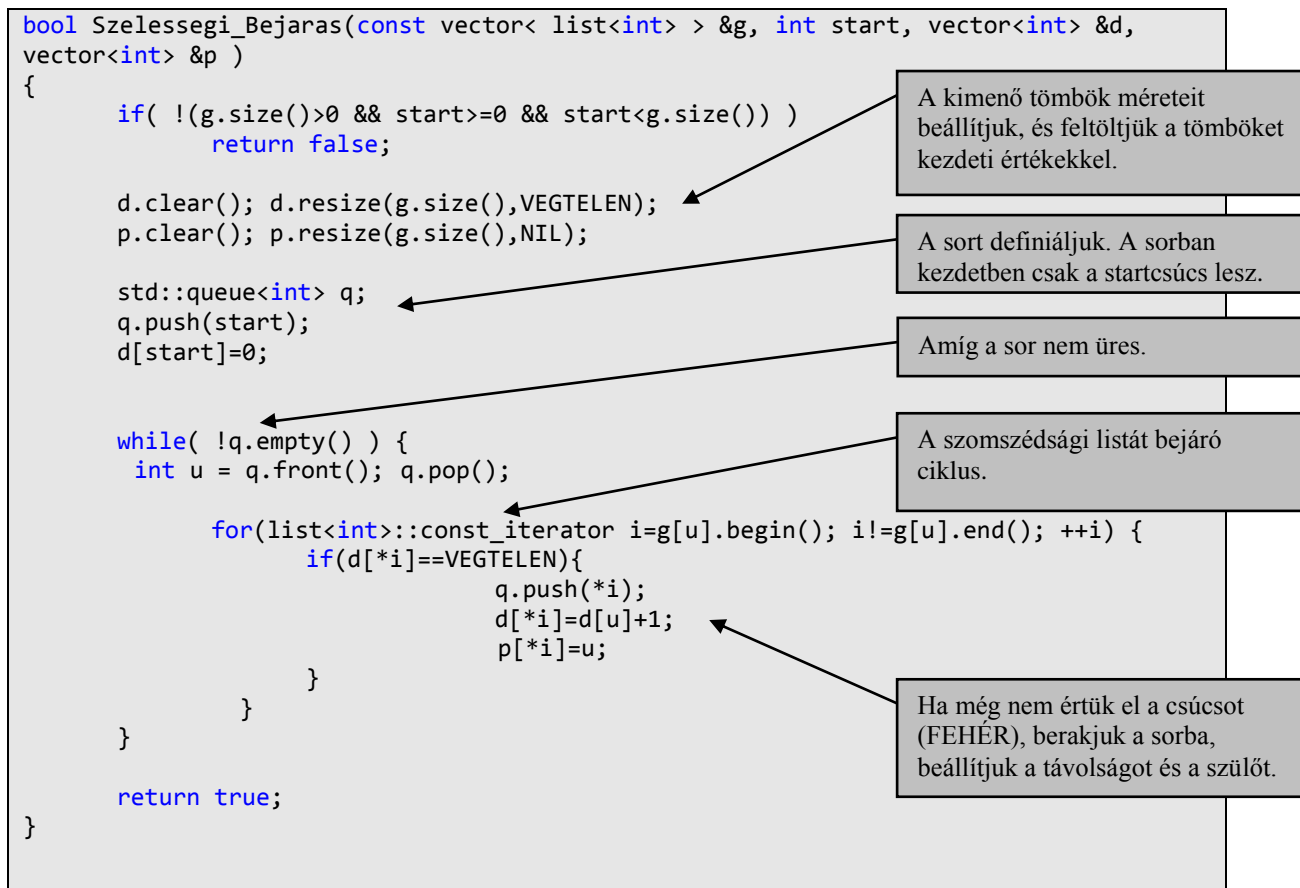
A célcúcsot és súlyt tartalmazó éltípus.

Vektorban a lista, élsúlyozott gráf esetén.

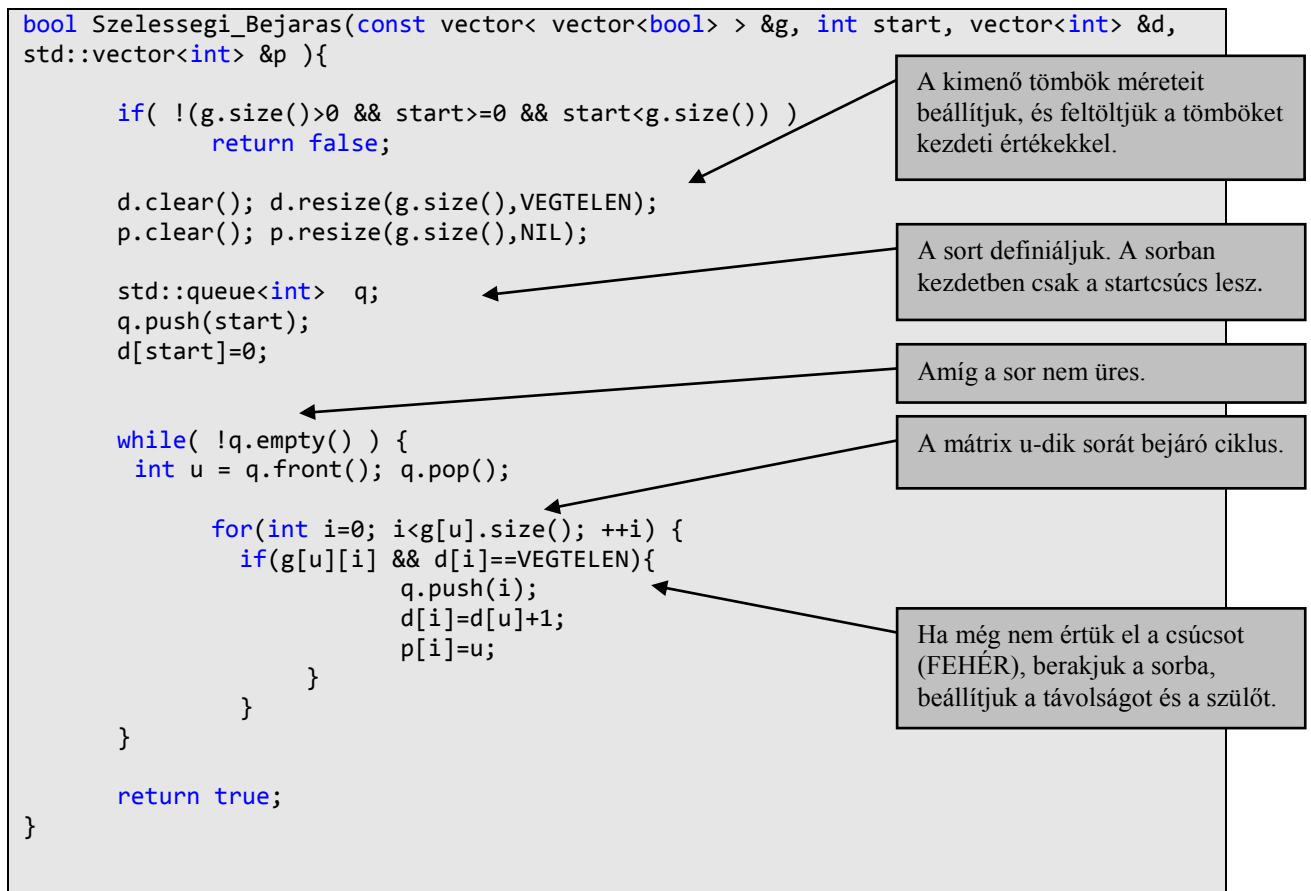
9.2 Szélességi bejárás

A szélességi bejárás implementálása során az STL sor típusát használjuk, amelyben az elért (szürke) csúcsok indexeit tároljuk. Nem használunk színezést, helyette a használhatjuk a csúcsok *d* tömbbeli távolság értékét, amely fehér csúcsok esetén végtelen, színes csúcsok esetén nem végtelen.

Lássuk az algoritmust éllistás ábrázolás esetén:



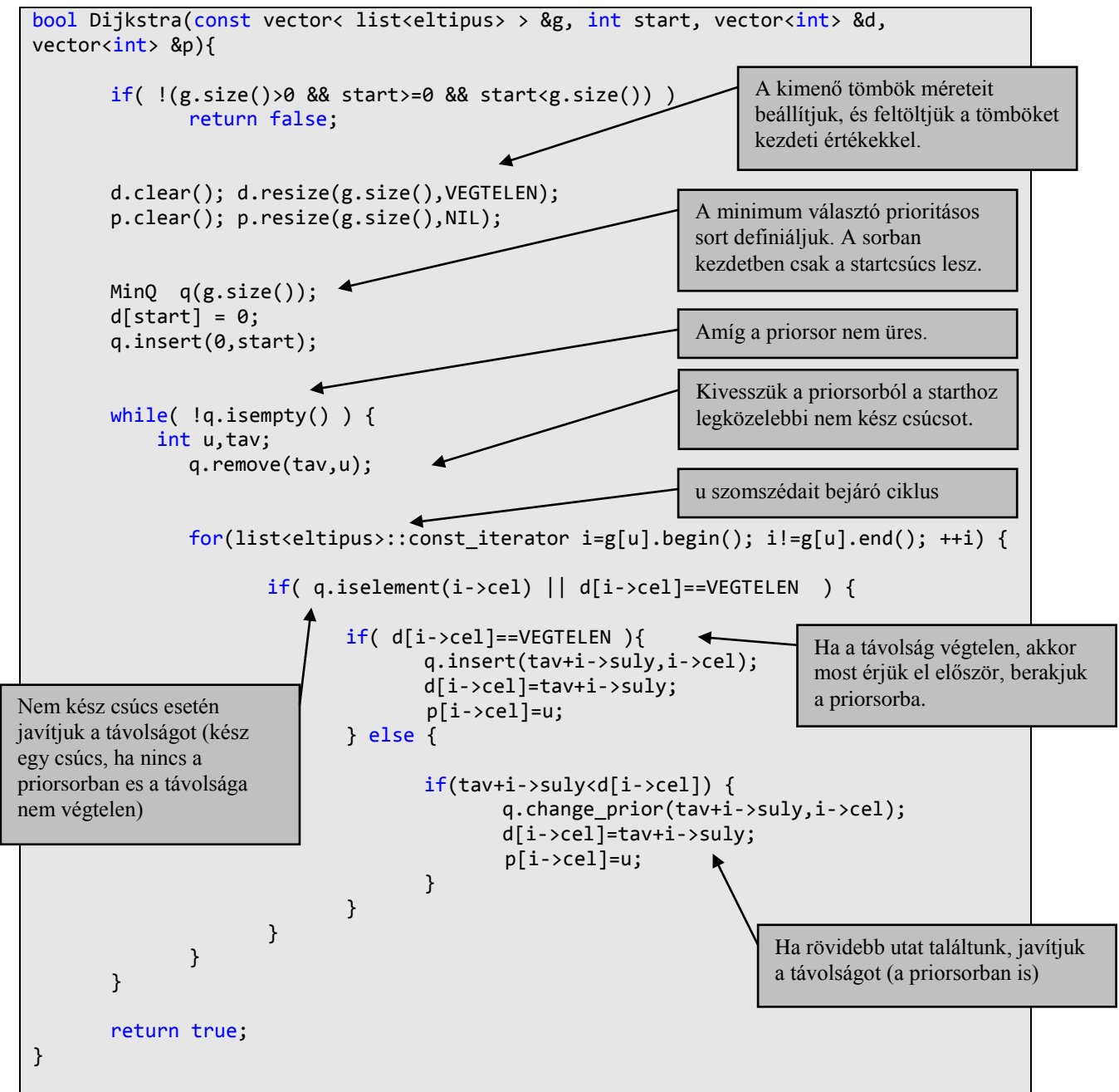
Lássuk az algoritmust szomszédsági-mátrix ábrázolás esetén:



9.3 Dijkstra algoritmus

Ritka gráfok esetén éllistas ábrázolást használtunk, a prioritásos sort pedig kupaccal valósítottuk meg. Az STL-ben található prioritásos sor nem tartalmaz módosító műveletet, pedig rövidebb út felfedezése esetén, módosítani kell egy csúcs prioritását. Készítettünk egy prioritásos sor típus sablont, amelyet a Dijkstra algoritmusban felhasználunk. Ennek implementálását a mellékletekben közöljük.

Éllistas ábrázolás esetén nem használunk színezést, sem KÉSZ halmazt, mivel a prioritásos sor és a távolság vektor segítségével eldönthető, hogy egy csúcs kész vagy nem kész. Egy csúcs kész, ha nincs a prioritásos sorban és a távolsága nem végtelen. Most lássuk az algoritmust:



Sűrű gráfok esetén szomszédsági-mátrix ábrázolást használunk. A prioritásos sort a távolság vektor segítségével valósítjuk meg feltételes minimumkeresést alkalmazva a nem kész csúcsok

távolságértékein. Egy logikai értéket tartalmazó vektort használjunk a KÉSZ halmaz megvalósítására.

```
bool FeltMinKer(const vector<int> &kész,const vector<int> &d,int &minhely){
    bool volt=false;
    for(int i=0; i<d.size(); ++i){
        if(!kész[i]){
            if(volt){
                if(d[i]<d[minhely])
                    minhely=i;
            } else {
                volt=true;
                minhely=i;
            }
        }
    }
    return volt;
}
```

Feltételes minimumkeresés a d vektoron. A kész vektor, mint feltétel felhasználásával.

```
bool Dijkstra(const vector< vector<int> > &g, int start, vector<int> &d, vector<int> &p){
    if( !(g.size()>0 && start>=0 && start<g.size()) )
        return false;

    d.clear(); d.resize(g.size(),VEGTELEN);
    p.clear(); p.resize(g.size(),NIL);

    vector<int> kész(g.size(),false);
    int u;
    d[start] = 0;

    while( FeltMinKer(kész,d,u) ) {
        kész[u] = true;
        //vegyük az u szomszédait, ha van út u-ba
        for(int i=0; i<g[u].size() && d[u]!=VEGTELEN; ++i)
            if( g[u][i]!=VEGTELEN && !kész[i] )
                if(d[u]+g[u][i]<d[i]) {
                    d[i]=d[u]+g[u][i];
                    p[i]=u;
                }
    }

    return true;
}
```

A KÉSZ halmazt megvalósító logikai vektor.

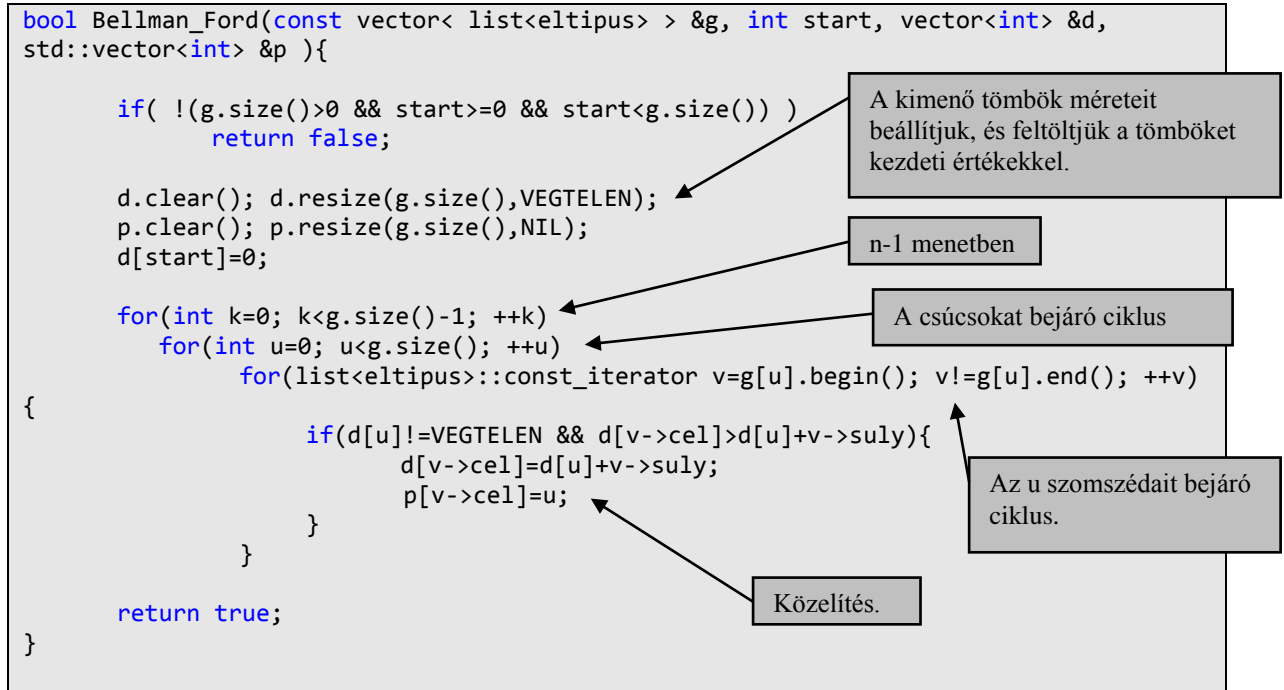
Ciklus amíg van nem kész csúcs.

A mátrix u-dik sorát bejáró ciklus.

Ha van él és még nem kész a csúcs, és rövidebb utat találtunk, beállítjuk a távolságot és a szülőt.

9.4 Bellman-Ford algoritmus

Tegyük fel, hogy a gráf nem tartalmaz negatív összköltségű kört. Az 5.2 fejezetben közölt algoritmus szerint $n-1$ menetben minden élen végezzünk közelítést. Éllistas ábrázolás esetén úgy tudjuk végigjárni az összes élt, hogy az összes csúcs éllistáját bejárjuk. Ezt megtehetjük egy csúcsokat bejáró ciklus belsejébe ágyazott éllistát bejáró ciklussal.



Ha a gráf tartalmaz startcsúcsból elérhető negatív összköltségű kört, akkor nincs legrövidebb út, azaz nincs megoldása a feladatnak. Egészítsük ki függvényt úgy, hogy ekkor hamis értékkel térjen vissza. Ehhez egy n-dik menettel kell kiegészíteni az algoritmust. Ha az n-dik menetben is tudunk „közelíteni”, akkor van a gráfban startcsúcsból elérhető negatív összköltségű kör.

```
bool Bellman_Ford(const vector< list<eltipus> > &g, int start, vector<int> &d,
std::vector<int> &p ){

    if( !(g.size()>0 && start>=0 && start<g.size()) )
        return false;

    d.clear(); d.resize(g.size(),VEGTELEN);
    p.clear(); p.resize(g.size(),NIL);
    d[start]=0;

    for(int k=0; k<g.size()-1; ++k)
        for(int u=0; u<g.size(); ++u)
            for(list<eltipus>::const_iterator v=g[u].begin(); v!=g[u].end(); ++v)
            {
                if(d[u]!=VEGTELEN && d[v->cel]>d[u]+v->suly){
                    d[v->cel]=d[u]+v->suly;
                    p[v->cel]=u;
                }
            }

    bool negativ_kor = false;
    for(int u=0; u<g.size(); ++u)
        for(list<eltipus>::const_iterator v=g[u].begin(); v!=g[u].end(); ++v)
        {
            if(d[u]!=VEGTELEN && d[v->cel]>d[u]+v->suly){
                d[v->cel]=d[u]+v->suly;
                p[v->cel]=u;
                negativ_kor=true;
            }
        }

    return !negativ_kor;
}
```

n-dik menet a negatív kör figyelésére

Az n-dik menetben volt közelítés, tehát találtunk negatív kört.

Szomszédsági-mátrix ábrázolás esetén a mátrix bejárásával tudjuk az összes élt végigjárni.

```
bool Bellman_Ford(const vector< vector<int> > &g, int start, vector<int> &d,  
std::vector<int> &p ){
```

```
    if( !(g.size()>0 && start>=0 && start<g.size()) )  
        return false;
```

```
    d.clear(); d.resize(g.size(),VEGTELEN);  
    p.clear(); p.resize(g.size(),NIL);  
    d[start]=0;
```

A kimenő tömbök méreteit beállítjuk, és feltöltjük a tömböket kezdeti értékekkel.

```
    for(int k=0; k<g.size()-1; ++k)  
        for(int u=0; u<g.size(); ++u)  
            for(int v=0; v<g[u].size(); ++v) {  
                if(g[u][v]!=VEGTELEN && d[u]!=VEGTELEN && d[v]>d[u]+g[u][v])  
                    d[v]=d[u]+g[u][v];  
                    p[v]=u;  
            }  
    }
```

n-dik menet a negatív kör figyelésére

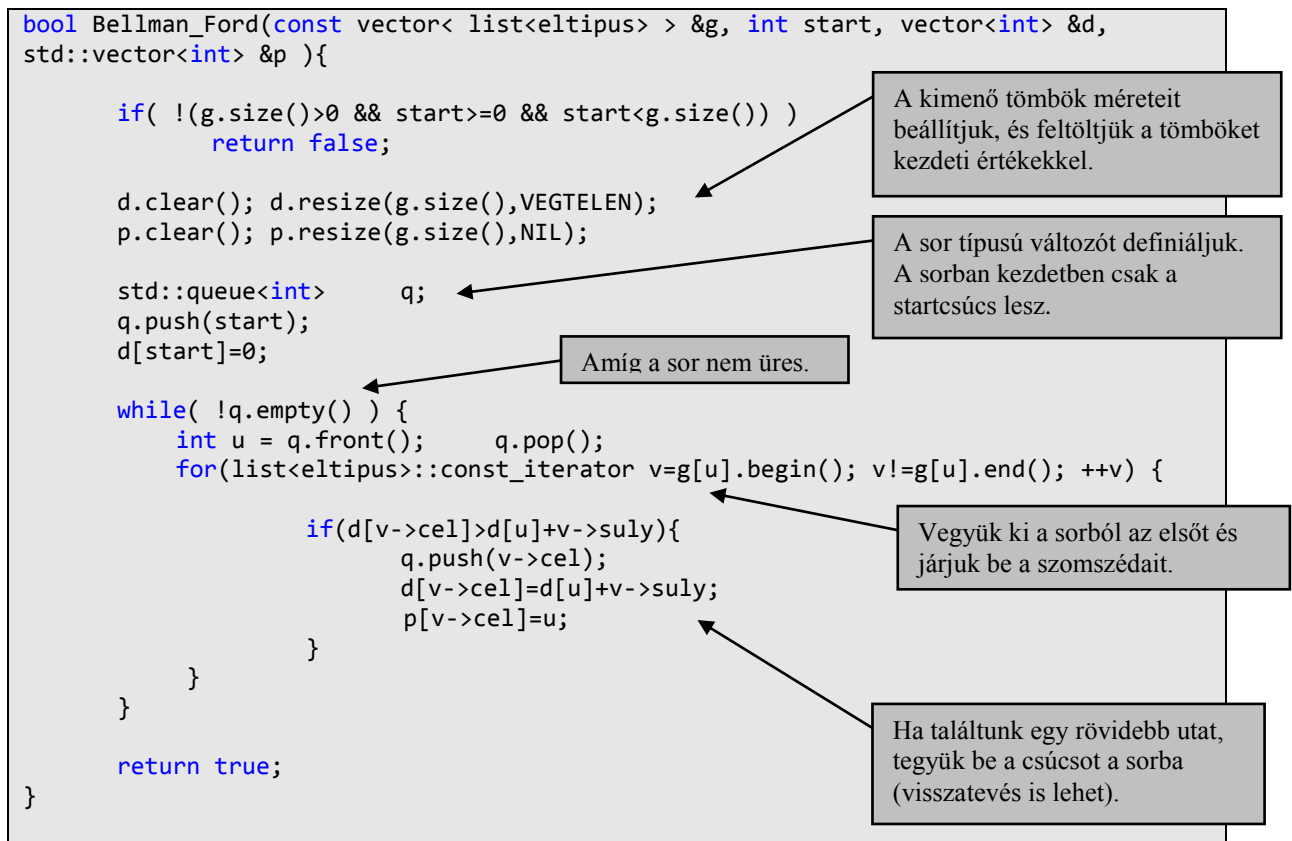
```
    bool negativ_kor = false;  
    for(int u=0; u<g.size(); ++u)  
        for(int v=0; v<g[u].size(); ++v) {  
            if(g[u][v]!=VEGTELEN && d[u]!=VEGTELEN && d[v]>d[u]+g[u][v]){  
                d[v]=d[u]+g[u][v];  
                p[v]=u;  
                negativ_kor=true;  
            }  
        }  
    }
```

Az n-dik menetben volt közelítés, tehát találtunk negatív kört.

```
    return !negativ_kor;
```

```
}
```

Most próbáljunk egy más szemléletű implementációt találni a Bellman-Ford algoritmusra, amely hatékonyabb lehet. Tegyük fel, hogy a gráf nem tartalmaz negatív összköltségű kört. Induljunk ki az invariáns tulajdonságból, miszerint az első menet után az 1 élszámú legrövidebb utak lesznek ismertek, a második menet után a 2 élszámú legrövidebb utak stb. Ez olyan, mint egy szélességi bejárás (csak a távolságértékeket máshogy definiáljuk), ahol a startcsúctól „gyűrűszerűen” tovaterjedő „lökéshullámmal” érjük el a csúcsokat. Arra kell ügyelni, hogy mikor egy csúchhoz kiszámoljuk az i élszámú legrövidebb utat, még nem állíthatjuk, hogy készen vagyunk, mert lehet, hogy van i -nél nagyobb élszámú rövidebb út. Tehát tegyük vissza a csúcsot a sorba, azaz implementálhatjuk a Bellman-Ford algoritmust a szélességi bejáráshoz hasonlóan, sor használatával, csak egy rövidebb út felfedezése esetén rakjuk vissza a csúcsot a sorba (visszatevéses sor használat).



Ha a gráf tartalmaz startcsúsból elérhető negatív összköltségű kört, akkor nincs legrövidebb út, ráadásul az előző megoldás esetén a csúcsokat ismételtlen visszatesszük a sorba, tehát a sor nem ürül ki, végtelen ciklusba kerülünk. Jó lenne, ha ekkor is tudnánk adni egyszerű leállási feltételt. Tudjuk, hogy a legrövidebb egyszerű út legfeljebb $n-1$ élszámú lehet. Ha a legrövidebb utunk eléri az n -dik élszámot, akkor van negatív kör. Tehát számoljuk a legrövidebb utak élszámát, nevezzük mélységnek (szélességi fa mélysége). Ha a mélység értékünk eléri az n -et és még nem terminált a ciklus, akkor negatív körre futottunk.

```

Bool Bellman_Ford(const vector< list<eltipus> > &g, int start, vector<int> &d,
std::vector<int> &p ){

    if( !(g.size()>0 && start>=0 && start<g.size()) )
        return false;

    //kimenő tömbök méreteit beállítjuk, feltöltjük a tömböket kezdeti értékekkel
    d.clear(); d.resize(g.size(),VEGTELEN);
    p.clear(); p.resize(g.size(),NIL);

    //a sorban kezdetben csak a startcsúcs lesz
    std::queue<int> q;
    q.push(start);
    d[start]=0;
    vector<int> melyseg(g.size(),VEGTELEN);
    melyseg[start]=0;

    while( !q.empty() ) {
        int u = q.front(); q.pop();
        for(list<eltipus>::const_iterator v=g[u].begin(); v!=g[u].end(); ++v)
        {
            if(d[v->cel]>d[u]+v->suly){
                q.push(v->cel);
                d[v->cel]=d[u]+v->suly;
                p[v->cel]=u;
                melyseg[v->cel]=melyseg[u]+1;
                if(melyseg[v->cel]>=g.size())
                    return false;
            }
        }
    }

    return true;
}

```

A mélységet számláló vektor (mint szélességi bejárás d tömbje).

A mélység 1-el nagyobb, mint a szülő mélysége (mint a szélességi bejárásnál).

Ha a mélység eléri n -et, kilépünk hamis értékkel, negatív kör, nincs megoldás.

9.5 Floyd algoritmus

A 6.1. fejezetbeli algoritmust könnyedén C++ kód formába önthetjük. Az algoritmus egyetlen kifejtendő részlete a távolság mátrix kezdeti értékének feltöltése. Ez szomszédsági-mátrix ábrázolás esetén mátrix értékadás, éllistas ábrázolás esetén az éllisták súlyainak bemásolása a mátrixba.

```
void Floyd(const vector< list<eltipus> > &g, vector< vector<int> > &D ){

    D.clear();    D.resize(g.size());
    for(int i=0; i<D.size(); ++i)
        D[i].resize(D.size(), VEGTELEN);

    for(int u=0; u<D.size(); ++u)
        for(list<eltipus>::const_iterator v=g[u].begin(); v!=g[u].end(); ++v)
            D[u][v->cel]=v->suly;

    for(int u=0; u<D.size(); ++u)
        D[u][u]=0;

    for(int k=0; k<D.size(); ++k)
        for(int i=0; i<D.size(); ++i)
            for(int j=0; j<D[i].size(); ++j)
                if( D[i][k]!=VEGTELEN && D[k][j]!=VEGTELEN && D[i][k]+D[k][j] < D[i][j] )
                    D[i][j] = D[i][k]+D[k][j];
}
```

Az éllisták súlyainak bemásolása a D távolság mátrixba, az éllisták bejárása közben

A főátló nullázása

A három beágyazott ciklus

Ha találtunk k-n átmenő rövidebb utat.

9.6 Warshall algoritmus

A Warshall algoritmus a Floyd algoritmusnak az egyszerűsített változata, a legbelső ciklus elágazása egyszerűsíthető logikai műveletekkel.

```
void Warshall(const vector< list<int> > &g, vector< vector<bool> > &W ){

    W.clear(); W.resize(g.size());
    for(int i=0; i<W.size(); ++i)
        W[i].resize(W.size(), false);

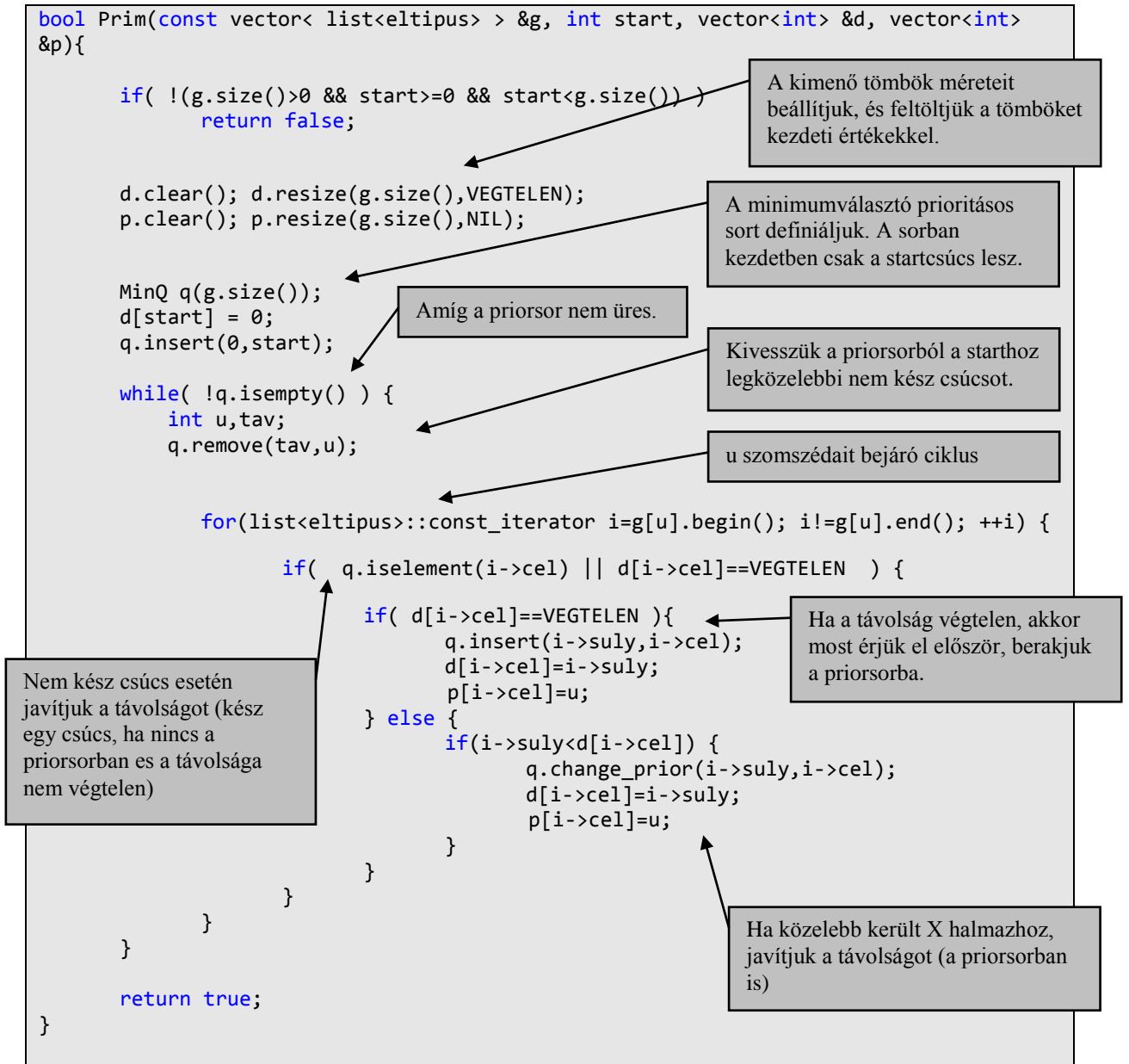
    //W kezdeti beállítása = szomszédsági mátrix
    for(int u=0; u<W.size(); ++u)
        for(list<int>::const_iterator v=g[u].begin(); v!=g[u].end(); ++v)
            W[u][*v]=true;

    //Főátló nullázása
    for(int u=0; u<W.size(); ++u)
        W[u][u]=true;

    for(int k=0; k<W.size(); ++k)
        for(int i=0; i<W.size(); ++i)
            for(int j=0; j<W[i].size(); ++j)
                W[i][j] = W[i][j] || W[i][k] && W[k][j];
}
```

9.7 Prim algoritmus

A Prim algoritmus implementáció tekintetében megegyezik a Dijkstra algoritmussal, azzal a különbséggel, hogy a „közelítésnél” használt távolság nem egy út távolsága (megelőző csúcsig való távolság+élsúly), hanem csak egy él távolsága (X halmaztól való távolság). Tekintsük az algoritmust éllistas ábrázolás esetén:



9.8 Kruskal algoritmus

A mohó algoritmus nem csúcsorientált, hanem él orientált, az éleket veszi sorra súlyuk szerint. Az egyik lehetőség, hogy az éleket rendezzük (akár lineáris rendező algoritmussal) és a rendezett sorozatból vesszük ki az éleket. Egy másik lehetőségünk, hogy prioritásos sorba rakjuk az éleket, majd ebből kivéve rendezett sorrendet kapunk. Ezt a második módszert implementáltuk. Menet közben fákat kezelünk, amelyek csúcsai diszjunkt halmazokat alkotnak. Egy soron következő élnél meg kell tudnunk állapítani, hogy az él két vége azonos fában van-e, azaz el kell tudni dönteni, hogy két csúcs azonos halmazban van-e. Továbbá szükségünk lesz két halmaz (fa) uniójára is. Ezen két műveletet hatékonyan megvalósító adatszerkezet megtalálható Rónyai Lajos, Ivanyos Gábor, Szabó Réka: *Algoritmusok* [2] könyvben. Ennek egy implementációját a mellékletekben közöljük. A függvény lefutása után a minerdo nevű lista fogja tartalmazni a feszítőfa (feszítőerdő) irányítatlan éleit.

```
//irányítatlan, súlyozott él
struct el{
    int u,v,suly;
    el(int x,int y,int s): u(x), v(y), suly(s) {}
};

/*hogya az STL prioritásos sor növekedően rendezzen
meg kell fordítani a < relációt az él típuson*/
bool operator<(el e1, el e2){
    return e1.suly > e2.suly;
}

bool Kruskal(const vector< list<eltipus> > &g, list<el> &minerdo){

    if( g.size()<=0 )
        return false;

    minerdo.clear();
    priority_queue<el> elsorrend;

    for(int u=0; u<g.size(); ++u )
        for(list<eltipus>::const_iterator i=g[u].begin(); i!=g[u].end(); ++i)
            if(i->cel < u)
                elsorrend.push(el(u,i->cel,i->suly));

    UnioHolvan erdo(g.size());

    while( !elsorrend.empty() ){
        el e = elsorrend.top();
        elsorrend.pop();
        if( erdo.holvan(e.u) != erdo.holvan(e.v) ) {
            minerdo.push_back(e);
            erdo.unio(e.u,e.v);
        }
    }

    return true;
}
```

Az élek rendezésére használt prioritásos sor.

Bejárjuk a minden csúcs éllistáját és az éleket a prioritásos sorba rakjuk

A diszjunkt halmazok ábrázolására.

Vesszük sorra az éleket, és ha két végük különböző fában van, akkor berakjuk az eredmény listába és összevonjuk (unio) a fákat.

9.9 Mélységi bejárás

A mélységi bejárás rekurzív algoritmusát implementáltuk C++ nyelven. A bejárás során előállítjuk a csúcsok mélységi és befejezési számát. A csúcsok színezése következik a mélységi és befejezési számok értékéből. A csúcs fehér (még nem értük el), ha még nincs kitöltve a mélységi száma, a csúcs színe szürke (elértük, de nem fejeztük be), ha van mélységi száma, de még nincs befejezési száma, és a csúcs színe fekete, ha mindkét szám ki lett töltve.

Mélységi bejárás éllistas ábrázolás esetén:

```
void MB(int u, const vector< list<int> > &g, vector<int> &mszam, vector<int> &bszam,
vector<int> &p, int &msz, int &bsz){
    mszam[u]=++msz;
    for(list<int>::const_iterator v=g[u].begin(); v!=g[u].end(); ++v)
        if(mszam[*v]==0){
            p[*v]=u;
            MB(*v,g,mszam,bszam,p,msz,bsz);
        }
    bszam[u]=++bsz;
}

void Melysegi_Bejaras(const vector< list<int> > &g, vector<int> &mszam, vector<int>
&bszam, vector<int> &p){
    mszam.clear(); mszam.resize(g.size(),0);
    bszam.clear(); bszam.resize(g.size(),0);
    p.clear(); p.resize(g.size(),NIL);

    int msz=0,bsz=0;

    for(int u=0; u<g.size(); ++u)
        if(mszam[u]==0)
            MB(u,g,mszam,bszam,p,msz,bsz);
}
```

Ha még nem értük el a csúcsot (fehér), rekurzív hívás.

A kimenő tömbök méreteit beállítjuk, és feltöltjük a tömböket kezdeti értékekkel.

Ha még nem értük el a csúcsot (fehér), indítjuk a komponens bejárását.

A csúcsmátrixos ábrázolás csak az éllista végigjárásában különbözik (nem listát kell bejárni, hanem a mátrix sorát), ezért ennek implementálását külön nem adjuk meg, mivel a korábban látott csúcsmátrixos implementációkból már könnyen elkészíthető.

9.10 DAG tulajdonság ellenőrzése

Írjuk át a mélységi bejárás rekurzív változatát. Nincs szükségünk mélységi és befejezési számok könyvelésére, elegendő a csúcsokat színezní. Ha bejárás során szürke színű csúcsot érintünk (elért, de be nem fejezett csúcsot), akkor körre futottunk.

```
bool DG(int u, const vector< list<int> > &g, vector<int> &szin){
    szin[u]=1; //szürke
    for(list<int>::const_iterator v=g[u].begin(); v!=g[u].end(); ++v)
        if(szin[*v]==0) { //Ha még nem értük el a csúcsot (fehér)
            if(!DG(*v,g,szin))
                return false;
        } else if(szin[*v]==1)
            return false;

    szin[u]=2; //fekete
    return true;
}

bool DAG(const vector< list<int> > &g){
    vector<int> szin(g.size(),0);
    for(int u=0; u<g.size(); ++u)
        if(szin[u]==0) //Ha még nem értük el a csúcsot (fehér)
            if(!DG(u,g,szin))
                return false;

    return true;
}
```

Ha a rekurzív hívás hamis értékkel tér vissza, akkor kört találtunk.

Ha szürke a szomszéd, akkor most találtuk a kört.

9.11 Topologikus sorrend

Adjuk meg a gráf csúcsainak topologikus sorrendjét egy listában. Ehhez a mélységi bejárást (DAG tulajdonság ellenőrzését) oly módon kell kiegészíteni, hogy amikor egy csúcsot befejezünk (feketére színezzük) a csúcsot berakjuk a készülő topologikus lista elejére, így a listában csökkenő befejezési szám szerint lesznek a csúcsok. Ha a függvény hamis értékkel tér vissza nincs topologikus sorrend, mert a gráf nem DAG.

```
bool TP(int u, const vector< list<int> > &g, vector<int> &szin, list<int>
&top_sorrend){
    szin[u]=1; //szürke
    for(list<int>::const_iterator v=g[u].begin(); v!=g[u].end(); ++v)
        if(szin[*v]==0) { //Ha még nem értük el a csúcsot (fehér)
            if(!TP(*v,g,szin,top_sorrend))
                return false;
        } else if(szin[*v]==1) //Ha elértük, de nem fejeztük be (szürke)
            return false;

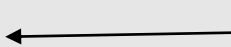
    szin[u]=2; //fekete
    top_sorrend.push_front(u);
    return true;
}

bool Topologikus_Sorrend(const vector< list<int> > &g, list<int> &top_sorrend){

    vector<int> szin(g.size(),0);
    top_sorrend.clear();

    for(int u=0; u<g.size(); ++u)
        if(szin[u]==0) //Ha még nem értük el a csúcsot (fehér)
            if(!TP(u,g,szin,top_sorrend))
                return false;

    return true;
}
```



Berakás a lista elejére, így lesz a lista befejezés szerinti csökkenő.

9.12 Erősen összefüggő komponensek meghatározása

A mélységi bejárást úgy kell módosítani, hogy a rekurzió kezdőcsúcsaink sorrendje megadható legyen. Először lefuttatunk egy mélységi bejárást (a bejárási sorrend tetszőleges lehet, nem adunk bejárási sorrendet) és előállítjuk a csúcsok a befejezési szám szerinti csökkenő listáját (topologikus sorrend meghatározásánál látottaknak megfelelően). Majd „megfordítjuk” a gráfot, és a „fordított” gráfon újra futtatjuk a mélységi bejárást, az első mélységi bejárás kimeneteként kapott listának megfelelő sorrendben választva a rekurzió kezdőcsúcsát. Az erős komponenseket a második mélységi bejárás után kapjuk meg egy vektorban. A vektort a csúcsokkal indexeljük, a vektor értékei pedig a komponens sorszáma lesz. Azok a csúcsok esnek azonos komponensbe, amelyeknek a vektorbeli értéke azonos. A csúcsok színezését a komponens vektor segítségével oldhatjuk meg. Ha még nem töltöttük ki (nulla), akkor a csúcs fehér, ha kitöltöttük, akkor szürke vagy fekete.

```
void EK_MB(int u, const vector< list<int> > &g, list<int> &top_sorrend, vector<int>
&komponensek, int komp_szamlalo){
    komponensek[u]=komp_szamlalo; //szürke
    for(list<int>::const_iterator v=g[u].begin(); v!=g[u].end(); ++v)
        if(komponensek[*v]==0) //Ha még nem értük el a csúcsot (fehér)
            EK_MB(*v,g,top_sorrend,komponensek,komp_szamlalo);

    top_sorrend.push_front(u);
}

void Fordit(const vector< list<int> > &g, vector< list<int> > &gf){
    gf.clear();
    gf.resize(g.size());

    for(int u=0; u<g.size(); ++u)
        for(list<int>::const_iterator v=g[u].begin(); v!=g[u].end(); ++v)
            gf[*v].push_back(u);
}

void EK_Melysegi_Bejaras(const vector< list<int> > &g, const list<int> &sorrend,
list<int> &top_sorrend, vector<int> &komponensek ){

    komponensek.clear();
    komponensek.resize(g.size(),0);
    top_sorrend.clear();
    int komp_szamlalo=0;

    if(sorrend.size()>0){
        for(list<int>::const_iterator u=sorrend.begin(); u!=sorrend.end(); ++u)
            if(komponensek[*u]==0) //Ha még nem értük el a csúcsot
                EK_MB(*u,g,top_sorrend,komponensek,++komp_szamlalo);
    } else //különbön csúcsindex szerint
        for(int u=0; u<g.size(); ++u)
            if(komponensek[u]==0) //Ha még nem értük el a csúcsot
                EK_MB(u,g,top_sorrend,komponensek,++komp_szamlalo);
}
```

Bszám szerinti csökkenő lista lesz, ez lesz a második bejárás sorrendje.

A gráf éleit megfordító eljárás, gf lesz a fordított gráf.

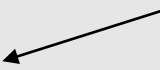
Ha van bejárási sorrend, akkor eszerint kell a rekurzív függvény kiindulópontját meghatározni, különben legyen a sorrend a csúcsok indexe szerinti

```

void Eros_Komponensek(const vector< list<int> > &g, vector<int> &komponensek ){
    list<int> s;
    list<int> top_sorrend;
    vector< list<int> > gf;

    EK_Melysegi_Bejaras(g,s,top_sorrend,komponensek);
    Fordit(g,gf);
    EK_Melysegi_Bejaras(gf,top_sorrend,s,komponensek);
}

```


 Mélységi bejárás a gráfon, a gráf fordítása, mélységi bejárás a fordított gráfon az első bejárás által előállított „topologikus” sorrend szerint.

Mellékletek

Prioritásos sor

```

/*****          PRIORITY QUEUE          *****/

//Prioritásos sor kupaccal megvalósítva.

template< class priority, class data>
class PriorQ {

public:
    PriorQ( int max_size );
    ~PriorQ();
    int size ();           //A sor max mérete.
    int elementcount ();   //A sorban lévő elemek pillanatnyi száma.
    bool isempty ();       //Üres-e a sor?
    bool isfull ();        //Tele van-e a sor?
    bool iselement( const data a ); //Benne van-e a kérdéses adat a sorban?
    bool isprior ( priority& p, const data a ); //Lekérdezzük a kérdéses adat
prioritását. Visszatérési értéke igaz, az adat benne van a sorban, hamis ha nincs.
    bool insert ( const priority p, const data a ); //Egy új adatot szúrunk be,
megadott prioritással. Igaz, ha sikerült beszúrni, es hamis, ha nem, mivel az adat mar szerepel a
sorban vagy megtelt a sor.
    bool remove ( priority& p, data& a ); //Kivesszük a
legnagyobb prioritásu adatot, amelynek adatait a paraméterben adjuk vissza. Igaz, ha sikerült
torolni, hamis, ha nem tudtunk, mert üres volt a sor.
    bool change_prior ( const priority p, const data a );//Megváltoztatjuk az adat
prioritását. Igaz, ha sikerült, hamis, ha nem, mivel az adat nem szerepelt a sorba.

protected:
    //segédfüggvények a hash-fv megírásához
    bool slot_isempty( int hashindex );
    bool slot_isgravestone( int hashindex);
    bool ervenyes_hashindex( int hashindex );

    virtual int hashfv ( data a ) = 0 ; //Leképezés az adatokról a [0..size-1]-ra,
mivel ezekkel a számokkal azonosítjuk az adatokat, az adatok szerinti gyors keres miatt.
Amennyiben az adat benne van a táblába, akkor a táblabeli indexet adja vissza a fv., ha nincs,
akkor egy olyan szabad helynek az indexet adja vissza, ahova beszúrnánk az adatot (mivel
beszúrásnál ide fogjuk majd beszúrnunk).
    virtual bool less( priority p1, priority p2 ) = 0; //igaz, ha a1 kisebb mint a2,
eszerint fogja "rendezni" a prioritásokat a heap

private:
    void csere_a_heapben( int heapindex1, int heapindex2 ); //Két csúcsot megcserél
a heap-ben(a hash-táblát is karban tartja).
    int szulo( int gyoker ); //Egy csúcs szülő jenek az indexe.
    int bal( int gyoker ); //Egy csúcs bal gyerekének az indexe
    int jobb( int gyoker ); //Egy csúcs jobb gyerekének az indexe
    bool uresfa( int gyoker ); //Igaz, ha az adott index üres fa (nincs a
"betöltött" heap-en belül)
    void sullyeszt( int gyoker ); //Egy csúcsot lesüllyeszt a fában, ha szükséges.
    void emel( int gyoker ); //Egy csúcsot felemel a fában, ha szükséges.

//a hash tábla egy slot-ja, ide hash-eljük az adatokat, es ez mutat a heap beli elhelyezkedésre
    struct slot{
        int heapindex; //a prioritasi ertek hol van a heapben
    };
};
```

```

        data adat;          //az adatresz
        slot( int holvan = 0, data a = data() ) : heapindex(holvan), adat(a) {}
        //a heapindex segítségével tartjuk nyilván azt, hogy a slot üres vagy
        törölt (sirkő, gravestone)
        //heapindex: 0 = üres, -1 = törölt
    };

    //a heap egy csúcsa
    struct csucs {
        priority prior;      //egy elem prioritása
        int          hashindex; //mutató a hash táblára, ez mutatja melyik
        elemnek a prioritása van itt
        csucs( priority p = priority(), int index = -1 ) : prior(p),
        hashindex(index) {}
    };

    int          s;          //size
    int          end;         //az alsó legjobb elemre mutat a heapben
    slot*        hashtable;  //ide hash-eljük az adatokat, így gyorsan meg lehet
    mondani egy adatról, hogy mennyi a prioritása ill. módosítani a prioritását
    csucs* heap;             //tömbben taroljuk a heap-et [1..size] részében, azért
    nem 0-tól, mert így gyorsabb a bal ill. jobb gyerek számítása

    //alapértelmezett másolások letiltása
    PriorQ( const PriorQ& q ) {}
    PriorQ& operator=(const PriorQ& q ) {}
};

/*****      public      *****/

template< class priority, class data>
PriorQ<priority,data>::PriorQ( int max_size ) {
    hashtable = new slot[max_size];
    heap      = new csucs[max_size+1]; //mivel [1..size]-ban van a heap
    s         = max_size;
    end       = 0;
}

template< class priority, class data> inline
PriorQ<priority,data>::~PriorQ( ) {
    delete[] hashtable;
    delete[] heap;
}

template< class priority, class data > inline
int PriorQ<priority,data>::size () {
    return s;
}

template< class priority, class data > inline
int PriorQ<priority,data>::elementcount () {
    return end;
}

```

```

template< class priority, class data > inline
bool PriorQ<priority,data>::isempty () {
    return elementcount()==0;
}

template< class priority, class data > inline
bool PriorQ<priority,data>::isfull () {
    return elementcount() == size();
}

template< class priority, class data > inline
bool PriorQ<priority,data>::iselement ( const data a ) {
    return hashtable[ hashfv(a) ].heapindex>0;    //tehát az adott slot nem üres (0) es nem
sírkő (-1)
}

template< class priority, class data >
bool PriorQ<priority,data>::isprior ( priority& p, const data a ) {
    if( iselement(a) ) {
        p = heap[ hashtable[ hashfv(a) ].heapindex ].prior;
        return true;    //Igaz, ha az adat benne van a sorban.
    } else
        return false;    //Hamis, ha az adat nincs a sorban.
}

template< class priority, class data >
bool PriorQ<priority,data>::insert ( const priority p, const data a ) {
    if( isfull() || iselement(a) )
        return false;    //Hamis, ha nem lehet beszúrni, mert az adat már benne van a
sorban vagy megtelt a sor.

    //a sor mérete egyel nő
    ++end;
    //az adatok beírása a hash táblába
    int hashindex = hashfv(a);
    hashtable[hashindex].adat = a;
    hashtable[hashindex].heapindex = end;
    //berakás a heap alsó legjobb helyére
    heap[end].prior = p;
    heap[end].hashindex = hashindex;
    //a beszúrt elem felemelése a helyére
    emel(end);

    return true;    //Igaz, ha sikerült beszúrni.
}

template< class priority, class data >
bool PriorQ<priority,data>::remove ( priority& p, data& a ) {
    if( isempty() )
        return false;    //Hamis, ha nem sikerül törölni, mivel üres sor.

    //a legnagyobb prior adat (heap tetején áll) kimentése visszatérési értéknek
    p = heap[1].prior;
    a = hashtable[heap[1].hashindex].adat;
}

```

```

//a törölt adat helyének a beállítása -1-re, azaz sírkövet rakunk a slotra
hashtable[heap[1].hashindex].heapindex = -1;
//a heap mérete eggyel csökken
--end;

//ha nem ürült ki a sor
if( !isempty() ){
    //a heap alsó legjobb elemének a tetőre hozása
    heap[1] = heap[end+1]; //end+1 mivel az előbb mar --end volt
    hashtable[ heap[1].hashindex ].heapindex = 1;
    //a legfőbb elem lesüllyesztése a "helyére"
    sullyeszt( 1 );
}
return true; //Sikerült beszúrni.
}

template< class priority, class data >
bool PriorQ<priority,data>::change_prior ( const priority p, const data a ) {

    if( !iselement(a) )
        return false; //Ha nem eleme a sornak, nem is lehet változtatni a
prioritását.

    int heapindex = hashtable[ hashfv(a) ].heapindex;
    heap[heapindex].prior = p;

    //lehet, hogy süllyesztetni vagy emelni kell
    sullyeszt(heapindex);
    emel(heapindex);

    return true; //Sikerült változtatni a prioritását.
}

/***** protected *****/

template< class priority, class data > inline
bool PriorQ<priority,data>::ervenyes_hashindex ( int hashindex ) {
    return ( 0<=hashindex && hashindex<size() );
}

template< class priority, class data > inline
bool PriorQ<priority,data>::slot_isempty( int hashindex ){
    return hashtable[hashindex].heapindex == 0;
}

template< class priority, class data > inline
bool PriorQ<priority,data>::slot_isgravestone( int hashindex) {
    return hashtable[hashindex].heapindex == -1;
}

/***** private *****/

template< class priority, class data >
void PriorQ<priority,data>::sullyeszt( int gyoker ) { //a gyökér nem üres fa

```



```

        if( uresfa( bal(gyoker) ) ) //1 elemű fa, nem kell süllyeszteni
            return;

        int irany;
        if( uresfa( jobb(gyoker) ) ) //ha nincs jobbgyerek, akkor balra kell süllyeszteni
            irany = bal(gyoker);
        else {
            //ha létezik mindkét gyerek, akkor a nagyobbik irányába kell süllyeszteni
            if( less( heap[ bal(gyoker) ].prior , heap[ jobb(gyoker) ].prior ) )
                irany = jobb(gyoker);
            else
                irany = bal(gyoker);
        }
        //ha kell süllyeszteni, akkor a meghatározott irányba süllyesztünk
        if( less( heap[gyoker].prior , heap[irany].prior ) ) {
            csere_a_heapben( gyoker, irany );
            sullyeszt(irany);
        }
    }

}

template< class priority, class data >
void PriorQ<priority,data>::emel( int gyoker ) { //a gyökér nem üres fa

    if( uresfa( szulo(gyoker) ) ) //elértem a heap tetejére, nem kell emelni
        return;

    //emelni kell, ha a szülőjének a prioritása kisebb, mint az illető csúcs prioritása
    if( less( heap[ szulo(gyoker) ].prior , heap[gyoker].prior ) ) {
        csere_a_heapben( gyoker, szulo(gyoker) );
        emel( szulo(gyoker) );
    }
}

//A heap-ben két csúcsot megcserél, a hash táblában is karbantartva.
template< class priority, class data >
void PriorQ<priority,data>::csere_a_heapben( int heapindex1, int heapindex2 ) {
    csucs tmpcsucs = heap[ heapindex1 ];
    heap[ heapindex1 ] = heap[ heapindex2 ];
    heap[ heapindex2 ] = tmpcsucs;
    hashtable[ heap[heapindex1].hashindex ].heapindex = heapindex1;
    hashtable[ heap[heapindex2].hashindex ].heapindex = heapindex2;
}

template< class priority, class data > inline
bool PriorQ<priority,data>::uresfa( int gyoker ) {
    return !( 0<gyoker && gyoker<=end );
}

template< class priority, class data > inline
int PriorQ<priority,data>::szulo( int gyoker ) {
    return gyoker/2; //egész osztás, alsó egészre konvertálással
}

template< class priority, class data > inline
int PriorQ<priority,data>::bal( int gyoker ) {
    return gyoker*2;
}

```

```

template< class priority, class data > inline
int   PriorQ<priority,data>::jobb( int gyoker ){
    return gyoker*2+1;
}

/*****                               MinQ                               *****/

//      Minimum választó prioritásos sor származtatva a prioritásos sor sablonból (gráf
algoritmusokhoz)

//A csúcsok és a prioritások is legyenek int típusúak.
class MinQ : public PriorQ<int,int> {
    public:
        MinQ( int max_size ) : PriorQ<int,int>( max_size ) {}
    private:
        //A hash függvény értéke nem más, mint a gráf csúcsának a sorszáma, azaz identitás
fv.
        int   hashfv( int csucs ) { return csucs%size(); }
        //Akkor kisebb egy prioritás, ha nagyobb, mivel minimum választó prior sor kell és
nem maximum választó, azaz a prioritás szerinti "rendezést" megfordítjuk
        bool   less( int p1, int p2 ) { return p1>p2; }
};

```

Unió-Holvan

```
#include <vector>

class UnioHolvan{
public:
    UnioHolvan(const int meret);
    void unio(int x, int y);
    int holvan(int x);
private:
    struct node {
        int szulo;    //fabeli szülője
        int elemszam; //a halmaz elemszáma, ha az illető reprezentáns elem
    };

    std::vector<node> v;
};

UnioHolvan::UnioHolvan(const int meret){
    v.clear(); v.resize(meret);
    for(int i=0; i<meret; ++i){
        v[i].szulo=i;    //gyökérnek a szülője saját maga
        v[i].elemszam=1; //kezdetben minden halmaz 1 elemű
    }
}

/*Megkeresi a halmazt reprezentáló elemet.
A fában felfelé megy, és közben útösszenyomást végez:
az út csúcsait átköti közvetlen a gyökér alá*/
int UnioHolvan::holvan(int x){
    if(v[x].szulo!=x)
        v[x].szulo=holvan(v[x].szulo);
    return v[x].szulo;
}

/*Két halmaz uniója, a nagyobb elemszámú halmaz
gyökere alá köti a kisebb halmaz gyökerét*/
void UnioHolvan::unio(int x, int y){
    int xgyoker = holvan(x);
    int ygyoker = holvan(y);
    if( xgyoker!=ygyoker ){
        if( v[xgyoker].elemszam > v[ygyoker].elemszam ) {
            v[ygyoker].szulo = xgyoker;
            v[xgyoker].elemszam += v[ygyoker].elemszam;
        } else {
            v[xgyoker].szulo = ygyoker;
            v[ygyoker].elemszam += v[xgyoker].elemszam;
        }
    }
}
```


Irodalomjegyzék

- [1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest: *Algoritmusok*. Műszaki Könyvkiadó, 1997, 1999
- [2] Rónyai Lajos, Ivanyos Gábor, Szabó Réka: *Algoritmusok*. TYPOTEX, 1999
- [3] Láng Csabáné: *Bevezető fejezetek a matematikába II*. ELTE Budapest, 1998
- [4] S. Lipschutz: *Adatszerkezetek*. Panem - McGraw-Hill, 1993
- [5] Fekete István: *2002/2003 tanév 2. félévi Algoritmusok és adatszerkezetek című előadásai*.
- [6] Gács Péter, Lovász László: *Algoritmusok*. Tankönyvkiadó, Budapest, 1991
- [7] Hajnal Péter: *Elemi kombinatorikai feladatok*. POLYGON, Szeged, 1997
- [8] *Középiskolai matematikai versenyek 1985-1987*. Tankönyvkiadó, Budapest, 1989
- [9] Bagyinszkiné Orosz Anna, Csörgő Piroska, Gyapjas Ferenc: *Példatár a bevezető fejezetek a matematikába c. tárgyhoz*. Nemzeti Tankönyvkiadó, Budapest, 1999
- [10] Csákány Rita, Csima Judit, Friedl Katalin, Ivanyos Gábor, Küronya Alex, Madas Pál, Pintér Márta, Rónyai Lajos, Sali Attila, Simonyi Gábor, Szabó Réka: *Algoritmusok – gyakorló feladatok* 1999. szeptember 13., <http://www.cs.bme.hu/~friedl/alg/fasor.pdf>