Mélységi gráfkeresés, élek osztályozása, alkalmazásai

- 2020. ősz – Szita B.

1. Mélységi gráfkeresés

Az alábbiakban a „mélységi gráfkeresés" algoritmust ismertetjük. Beszélünk az alapfeladatról, megadjuk az algoritmusát műveletigénnyel és bemutatjuk, hogyan lehet illusztrálni az algoritmus lejátszását.

1.1. Feladat

Járjuk be egy gráf **összes csúcsát** úgy, hogy egy tetszőlegesen kiválasztott csúcsból elindulva abból egy tetszőleges úton a „falig" megyünk, majd egy csúcsnyit visszafordulva egy másik úton ismét olyan mélyre megyünk, ahogy csak lehetséges, és ezt ismételjük egész addig, amíg nem marad hova visszafordulni.

Ekkor összefüggő esetben épp végeztünk, nem összefüggő esetben keresünk egy tetszőleges újabb, még érintetlen csúcsot és folytatjuk tovább.

Előbb-utóbb nem marad már „érintetlen" – tehát a bejárás által még nem látogatott – csúcs, ezzel az algoritmus terminál. Ez a nem összefüggő esetre is vonatkozik.

Az első mondatban említett „falig" is azt jelenti: annak a csúcsnak már nincs több érintetlen szomszédja.

1.1.1. Feltételek

A gráf *irányítottság*, *élsúlyozottság*, sőt *összefüggőség* szempontjából is **tetszőleges** lehet.

Hangsúlyozzuk, hogy nem összefüggő esetben is úgy hangzik a feladat, hogy "látogassuk a gráf összes csúcsát". Ezt a „feature"-t már a szélességi bejárásnál is felvetettük, a mélységi esetben pedig alapértelmezésben így működik az algoritmus.

Itt *komponens*enként van kezdőcsúcs, de az alábbiakban ismertetett változatban ez nem előre rögzített, így az algoritmus *üres gráf*ra is működik (és értelemszerűen nem csinál semmit).

Kört (akár *hurokélt*) is tartalmazó gráfra is működik, de kördetektáló algoritmust is könnyedén építhetünk rá – lásd később.

Amennyiben a gráfok a *bináris fák* általánosításai, úgy a mélységi bejárás a *preorder* bejárás általánosításaként fogható fel.

1.1.2. A nevéről

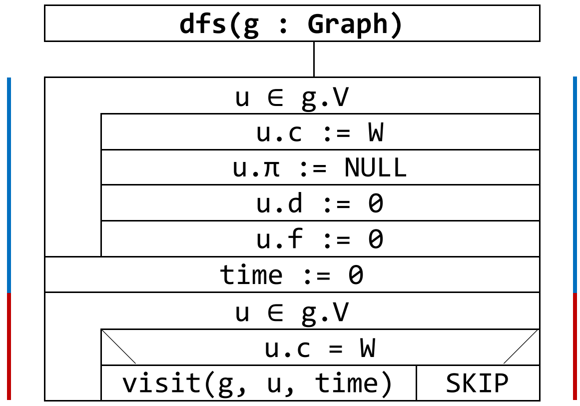
"*Mélységi*", hiszen a gráf csúcsait a kezdőcsúcshoz képest egyre "mélyebbre" hatolva látogatjuk – ellentétben a szélességi bejárással, ahol először a legközelebbieket, majd az eggyel távolabbiakat, stb. vesszük sorra.

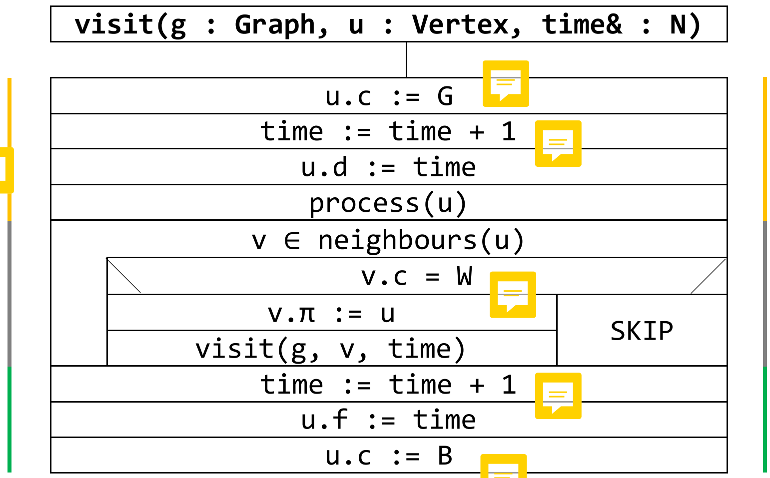
"*Keresés*", hiszen az algoritmus egy felsorolást biztosít a gráf csúcsaira, amit egy megállási feltétel megadásával keresésre használhatunk.

Emellett "*mélységi bejárásként*" is nevezhetjük, hiszen hacsak nem adunk meg egy megállási feltételt, garantáltan az összes csúcsot bejárjuk.

Angolul "*Depth-first Search*" néven ismert, ezért a továbbiakban **DFS** rövidítéssel is fogunk rá utalni.

1.2. ALGORITMUS





Az algoritmus ADT (absztrakt) szinten van megfogalmazva, tehát itt most `Graph` a gráf és `Vertex` a csúcs típusa. Továbbá, feltételezzük, hogy rendelkezésre áll egy `neighbours(Vertex)` függvény, ami a paraméterként átadott csúcs szomszédjait adja vissza, ezzel megragadván a gráf éleit. Az alkalmazott reprezentáció itt most nem lényeges, de nyilván elvárjuk, hogy minden használt funkció legyen hatékony.

1.2.1. Magyarázat & szemléltetés

Maga a gondolatmenet, így az algoritmus is *rekurzív*.

A `dfs()` eljárás egyetlen paramétere a gráf, amit be kell járni. Ebben az alapváltozatban semmivel nem térünk vissza, de minden csúcson elvégzünk egy ezen az absztrakt szinten közelebbről nem meghatározott eljárást. Ez a konkrétabb alkalmazásokban eltérhet, akár visszatérési érték is lehet, amit pl. egy új, *akkumulált* paraméter bevezetésével tudunk kezelni, kb. úgy, mint itt a `time` változót.

Tekintsük az első 6 sort – **kékkel** jelölve látható. Ez az algoritmus inicializálása, egyszer fut le. Itt vesszük sorra az összes csúcsokat, azok attribútumait kezdeti értékre állítjuk:

- Minden csúcs színe fehér (`W`) – azaz *még* nem derítettük fel
- Minden csúcs szülője `NULL` – hiszen *még* nem tudjuk, mi is lesz ez
- Minden csúcs felfedezése és befejezése `0` – azaz *még* nem történt meg egyik aktus se

A *globális* idő számlálót is 0-ra állítjuk, ami egyébként nem lesz „valid" értéke egyik `d`-nek/`f`-nek sem, ez csupán inicializáció.

Most, sorra vesszük a csúcsokat – lásd **vörössel** jelölt 3 sort –, `for each` ciklust használva, tehát nem definiált sorrendben – praktikusan, a nevezetes reprezentációkban „ábécésorrendben". Megnézzük az adott csúcs fehér-e – ez kezdetben rögtön az első csúcsra igaz lesz, majd meghívjuk a rekurzív függvényt. Ha ez a rekurzív függvény visszatér ide, akkor tudjuk, hogy az első csúcsból kiinduló (irányított) *komponens*sel végeztük. Ekkor végigjárja a maradék csúcsokat, ha csak egy komponens volt, ezek eddigre már feketék lesznek, ha pedig több komponens volt, megtalálja az újabb még *érintetlen*, aza fehér színű komponenst, bejárja azt is, és így tovább végig.

A `visit()` segédeljárás megkapja tehát a gráfot és az aktuális kiinduló csúcsot (olvasásra) és a globális *timestamp* értéket (olvasásra és írásra).

Tekintsük a **naranccsal** jelölt 4 sort. Ez az `u` csúcs (első) látogatását adja meg:

A színét szürkére (`G`) állítjuk – a fehér az érintetlent, a szürke az épp *látogatás alatt* levő csúcsot jelenti – egész pontosan, ha egy csúcs szürke, akkor azt a csúcsot, vagy annak egy a végső *mélységi fá*ban vett leszármazottját látogatjuk.

Léptetjük a globális időt eggyel – ne feledjük, úgy léptünk ide elsőre be, hogy ez az érték `0` volt, de tudjuk, hogy `1`-től `2n`-ig akarunk időpillanatokat rendelni a csúcsokat ért eseményekhez.

Most a *mélységi*, avagy *felfedezési szám*ot beállítjuk erre az új, *egyedi* értékre (kezdetben `1`).

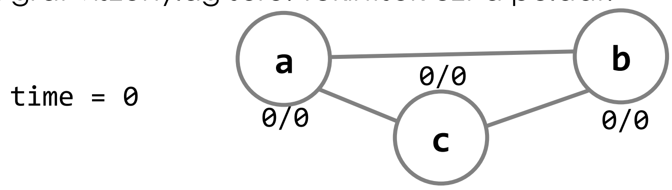
Lefuttatunk valamilyen algoritmust, amit minden csúcsra szeretnénk (ekképpen ez a bejárás tekinthető egy „*enumerátor*nak", a `process()` pedig maga a tevékenység, amit elemenként el akarunk végezni).

Utána – lásd **szürkével** –, végigjárjuk az *aktuális vizsgált csúcs* szomszédjait (a sorrend mindegy), mindre ellenőrizzük, hogy az az adott csúcs még nem *látogatott*-e, ha ez igaz, beállítjuk *szülő*jének az aktuális csúcsot, és meghívjuk ugyanezt a függvényt rá rekurzívan. A szülő beállítás azért nem került bele a rekurzív hívásba, mert így csak ezért a parancsért nem kell plusz egy paramétert átadni, illetve – láthatjuk – ez a rész nagyon hasonlít a **vörössel** jelölt részlethez, csak épp ott nincs szülő pointer állítás: ott is a `visit()` függvényt hívjuk, viszont ott tudatosan nem állítunk szülő pointert, ott nincs annak értelme; azok a csúcsok a mélységi fák *gyökere*i, nincs szülőjük.

Itt tehát potenciálisan annyi rekurzív hívást végzünk, ahány szomszédja van az aktuális csúcsnak – azért csak potenciálisan, mert lehet, hogy a szomszédok között valaki már be lett járva, sőt, olyan is lehet, akit egy másik szomszéd fog előbb érinteni – azaz végső soron, mire sorra kerül, már be lesz járva.

Miután ezek a szomszédokból adódó rekurzív hívások – és azoknak a rekurzív továbbhívásai is – visszatértek, a csúcsot meglátogatottnak tekintjük. Ennek kódja látható a **zöld** részben. Növeljük az idő-számlálót, a *befejezési szám*ot erre a globálisan egyedi számra módosítjuk, és feketének (`B`), azaz befejezettnek színezzük a csúcsot.

Érdemes végiggondolni, hogy a színezések, és a globális timestamp növekedése hogy is néz ki olyan esetben, amikor a gráf viszonylag sűrű. Tekintsük ezt a példát:



Most az egyszerűség kedvéért tegyük fel, hogy olyan megvalósítást használunk, ahol a `for each` ciklusok a csúcsokat *ábécésorrendben* látogatják.

Ekkor tehát az `a`-val hívjuk meg először a `visit()`-et, hiszen az az első *fehér* csúcs. A `visit()` függvény alapján `a` színe szürke lesz, a `d`-je `1`. Utána megnézzük a szomszédait, előbb `b`-t (majd utána `c`-t). Mivel `b` fehér, ezért a π-jét `a`-ra állítjuk, majd meghívjuk vele a `visit()`-et. Miután ez *return*öl, ne feledjük, hogy innen folytatjuk, azaz még `c`-t is meg fogjuk nézni. Közben `a` marad szürke, a `d`-je `1`, az `f`-je `0`, a π-je pedig `NULL`.

Most `b`-ben vagyunk, szürkére állítjuk, a `d`-je `2` lesz. A globális számláló most `2`. Megnézzük a szomszédokat, `a` szürke, de `c` fehér, ezért ennek a π-jét `b`-re állítjuk és meghívjuk vele a `visit()`-et.

Tehát `c`-ben vagyunk, szürkére állítjuk, a `d`-t pedig `3`-ra. Persze a `time` is `3`. Nézzük a gyerekeket. Előbb `a`-t, majd `b`-t, mind a kettő szürke, tehát `SKIP`-eljük mindkettőt. `time`-ot `4`-re állítjuk, a `c` befejezési számát szintén, `c`-t feketére színezzük és ez a rekurzív szint visszatér a szülő kontextusba, ami a `visit()` függvény a `b` csúcson.

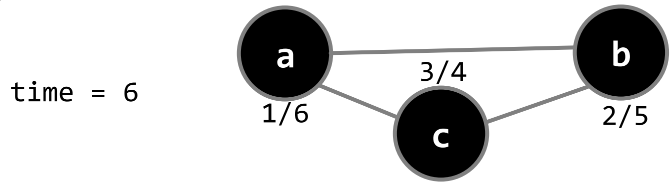
`b` csúcsnak az `a` szomszédját `SKIP`-eltük, a `c`-ből most jövünk vissza, tehát az algoritmus **zöld** soraihoz ugrunk: `b` befejezési számot kap, ami a `time` új értéke: `5`; majd feketére színezzük és visszatérünk a hívó helyre, ami a `visit()` függvény az `a` csúcsra, annak is csak az első gyerekét vizsgáltuk: `b`-t.

Most tehát a második gyerekét vizsgáljuk, ami `c`, és *eddigre* már fekete. Az egy dolog, hogy amikor ezt az egészet meghívtuk még fehér volt. Egy *szál*on dolgozunk, a rekurzió a fentiekben ismertetett elven fog visszabomlani, tehát mire ez a ciklus `c`-hez ér, már `b` révén egyszer elértük ezt a csúcsot, így most újra már nem látogatjuk, nem dolgozzuk fel. Az ilyen élekre majd később az „*előreél*" kifejezést fogjuk bevezetni.

Mivel tehát `c`-t `SKIP`-eltük, a legkülsőbb `visit()` *szekvencia* is eljut a **zöld** blokkig, tehát növeli `time`-ot `6`-ra, ezt megkapja `a.f`, és az `a` csúcs be is feketedik.

Kilépünk `dfs()`-be, itt még a ciklus végigmegy `b`-n és `c`-n, de azt látja, egyik sem fehér, tehát nem lép be újra a rekurzióba.

Most így néz ki a gráf:



A csúcsoknál az első szám legyen a *mélységi szám*, azaz a `d` attribútum, míg a második a *befejezési szám*, azaz az `f`-fel jelölt, az algoritmus által a csúcsról számon tartott adat. A `time` az algoritmus végére pont `2n` értéket vesz fel, de ez csak egy *segédváltozó*, az algoritmus *terminálás*ával kikerül a *scope*-ból.

Tehát `time` `0`-ról indul, *szigorúan monoton nő*, a végén *pont* `2n` lesz. Minden pozitív `time` érték bekerül valamelyik csúcs `d`-jébe, `f`-jébe. A végére minden `d` és `f` egyedi; minden csúcsra `d` és `f` is `[1..2n]` intervallumbeli; minden csúcsra a `d`-je kisebb, mint az `f`-je. Minden csúcs esetén `d` és `f` is `0`-ra inicializálódik majd *pontosan* egyszer kap új értéket, *előbb* `d`, *majd* `f`. Emiatt kisebb `d`. Mivel `n` csúcs van, minden csúcsra `2` szám, ez összesen pont `2n` különböző érték. Az elején minden csúcs *fehér*, a végére biztosan mind *fekete* lesz. De ennél még sokkal többet is tudunk. Ezeket lásd később.

1.2.2. Összefoglaló/jelmagyarázat

- `dfs` – Az algoritmus neve – *Depth-first Search*
- `Graph` – A gráf típusa *absztrakt* szinten
- `Vertex` – A csúcs típusa *absztrakt* szinten
- `g.V` – Ha `g` egy `Graph` – A csúcsok *halmaza*, azaz ez egy `Set<Vertex>` típusú *objektum*
- `n` – A csúcsok száma, azaz `|g.V|`
- `v.c` – Ha `v` egy csúcs – A csúcs „színe" (color)
- `W`, `G`, `B` – *Felsorolási típus*, a `c` adattag lehetséges értékei – Rendre: *fehér* (white), *szürke* (gray), *fekete* (black)
- `v.π` – Ha `v` egy csúcs – Ez egy `Vertex*`, azaz egy csúcsra mutató *pointer*, potenciálisan `NULL` értékkel. `v` szülő csúcsát jelenti, azaz annak az élnek a *kezdőcsúcsát*, amiből a mélységi fában `v`-t elértük – ez `NULL`, ha `v` a *gyökér*

- `time` – Egy természetes szám – Ezzel mérjük, hogy *globálisan* az algoritmus hányadik „látogatást" illetve „távozást" végzi, a végén épp `2n` lesz az értéke
- `v.d`, `v.f` – Ha `v` egy csúcs – Rendre a *belépési/felderítési/mélységi* (depth, discovery) és a *véglegesítési/távozási* (finalization) számérték, ami azt mondja meg, az adott csúcsot globálisan hányadiknak látogattuk meg; illetve hányadiknak hagytuk el az ő általa meghatározott részfát. `1`-től számozzuk, a `0` érték jelenti a *még* nem látogatott/ *még* nem elhagyott csúcsokat. A `d`-t azért hívjuk mélységi számnak, mert az algoritmus logikája miatt, a kiinduló csúcstól számolva az attól vett *mélyéget*, élszámban kifejezett *távolságot* mutatja (persze *egy lehetséges* – nem feltétlen optimális – úton, ez nem valódi távolság, arra van a *BFS*, illetve élsúlyozott esetben a *Dijkstra* és további speciális esetei)
- `process(Vertex)` – A bejárás mint általában, feltételezi, hogy valamilyen érdemi eljárást meghívunk a csúcsokon, ez kerülhet eme függvény helyére a gyakorlati alkalmazásokban
- `visit(Graph, Vertex, &N)` – A rekurzív segédfüggvény – A 3. paraméter az idő, ezt módosítjuk is, ezért a *referencia* szerinti paraméterátadás. A gráfot pedig, tekintve, hogy nem *primitív* típus, eleve referencia szerint adjuk át, de se ez, se a csúcs nem kap itt új értéket

1.3. IMPLEMENTÁCIÓ ÉS MŰVELETIGÉNY

Mint láttuk, egy *globális* segédváltozót, valamint minden csúcshoz 4 plusz adatot tartunk számon (`time`; `d`, `f`, `c`, `π`). Ezeket felvehetjük *segédtömb*ként (segédtömbökként) vagy az adott fa reprezentációt az *OOP* vívmányait használva pl. *származtatás*sal kiegészíthetjük. Érdemes lehet az egész algoritmusra egy külön *osztályt* készíteni, így a `time` annak *adattag*ja lehet, nem kell átadni *paraméter*ként.

Inicializálásnál (**kék** rész) egyszer végigmegyünk a csúcsokon és a fentieket *alapértelmezett*re állítjuk. Nyilván bizonyos nyelveknél pl. az explicit `NULL`-ra állítást meg se kell tenni, ha a típus *default* értéke az. A lényeg, akár *mátrixos*, akár *láncolt* gráfábrázolást is használunk, ez egy egyszerű, a csúcsok száma szerinti *lineáris* művelet, azaz `Θ(n)` *műveletigényű*.

A **piros** részben újra végigmegyünk a csúcsokon és feltételesen meghívjuk a rekurzív függvényt. A csúcsokon végigiterálni megint csak *lineáris*, bármely implementációt is válasszuk.

A `visit()` függvény sorai általában egyszerű *értékadás*ok, a `process()`-ről most tegyük fel, hogy *konstans*, van viszont a `neighbours()` függvény kiszámolása és az az által visszaadott *gyűjtemény bejárása* a rekurzív hívással. Ezzel mi a helyzet? Egy adott csúcs szomszédainak megállapítása mindkét nevezetes reprezentációnál *konstans* idejű feladat, ám az ezeken való iterálás *lineáris* komplexitású. Milyen hosszú a szomszédok listája? Amennyi a `visit()` által aktuálisan tekintett csúcs *kifoka*.

Tudjuk, hogy a `visit()` akár a `dfs()`-ből, akár a *rekurzív* hívások mentén minden csúcsba *pontosan egyszer* megy bele, hiszen a `visit()` meghívásának *őrfeltétele*, hogy a csúcs színe *fehér* legyen, de rögtön első parancsaként átszínezi *szürkére*, s ebből *fehér* már nem lesz soha. Viszont minden csúcsot megnézünk és addig nem nyugszunk, míg van *fehér* csúcs, így tehát a `visit()` mindösszesen *pontosan* `n`-szer fog meghívódni.

A `visit()`-ben tehát mindig az adott csúcs *szomszéd*ait nézzük sorra. Mivel minden csúcsra ez történik, a `neighbours()`-bejárások össz-lépészáma az *élek száma*, `e` lesz (illetve *irányítatlan* esetben `2e`, de ez *nagyságrend*i különbséget nem okoz).

Összesítve:

- Inicializálás: `n`
- `visit()`: összesen `n`
- `visit()`-en belüli ciklus: összesen `e` vagy `2e`

Összesen `Θ(n+e)` a műveletigény – feltéve, hogy a `process()` `Θ(1)`-es.

A fenti algoritmus egy *sablon*, amit a konkrét esetekben, alkalmazásokban személyre szabhatunk. Ez azt is jelenti, hogy bizonyos részeit el lehet hagyni. Ez nagyságrendi javulást (vagy rontást) nem fog okozni, de jó tudni róla.

Pl. a π-t, ha nem vagyunk kíváncsiak a mélységi bejárás által generált *feszítőerdő*re, a „*mélységi erdő*re". Ez eddig viszonylag egyszerű, mert a π-t az algoritmus nem „használta" a működéséhez.

De emellett a *színek* plusz adattagjait (vagy tömbjét) is úgy ahogy vannak, kidobhatjuk. Mikor *fehér* egy csúcs? Inicializálástól, amíg rá nem hívjuk a `visit()`-et. Azaz addig, amíg mind a `d`-je, mind az `f`-je `0`. Az, hogy a π-je is `NULL`, az már más kérdés, az semmit nem jelent. A `visit()` elején lesz *szürke*, egészen a `visit()` végéig. Tehát akkor *szürke*, amikor `d`-je már van, de `f`-je még nincs. És akkor *fekete*, amikor `d`-je is és `f`-je is van. Ez alapján az algoritmus feltételei is átfogalmazhatók, bár a megértés ezzel kevésbé lesz intuitív.

1.4. FELADATOK

1.4.1. Néhány ellenőrző villám-keresztkérdés

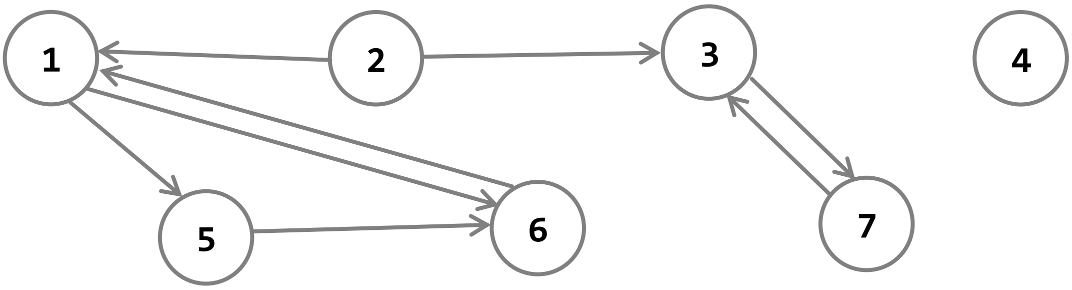
Ahol lehetett, a válaszokat indoklás nélkül a kérdések mögé írtam fehérrel – kimásolással ellenőrizhetők. Konzultáción beszéljük át részletesebben.

- A `dfs()` algoritmus **kék** és **vörös** részében is egy-egy azonos *gyűjtemény*t bejáró ciklust látunk egymással szekvenciában. Nem lehetne-e ezeket összevonni?
- Irányítatlan esetben tudjuk, ha `a` szomszédja `b`, akkor `b` szomszédja is `a`. Ekkor hogy lehet, hogy mégis minden csúcsot csak egyszer jár be?
- Lehet-e olyan adattag, aminek az értéke végig az marad, amit *inicializálás*kor megkap?
- Miért adjuk át `time`-ot *referencia* szerint?
- Ha egy `v` csúcsra az algoritmus végére `v.f = v.d + 4`, ez szemléletesen mit jelent?
- Egy adott pillanatban hány *szürke* csúcs van?
- Képzeljünk el egy gráfot, ami egy labirintust ír le (az élek az utak, a csúcsok a kereszteződések/sarkok, az egyik csúcs meg van *címkézve* az *EXIT* felirattal, ide kéne eljutni). Valahol a labirintus közepén állsz és nem látsz rá a teljes labirintusra mindig csak az aktuális csúcsra (szóval mintha ott lennél a valóságban). A *szélességi* vagy a *mélységi* bejárás stratégiájával jutsz ki várhatóan könnyebben/hamarabb a labirintusból? Miért? Egyáltalán biztos-e hogy kijutsz ezen stratégiákkal (feltéve, hogy van *EXIT* és van a *start*ból *út* is oda)?

1.4.2. Komplexebb feladatok

- Írd át az algoritmust úgy, hogy a `c` és π adattagok frissítését és használatát kihagyod belőle
- Írd át az algoritmust úgy, hogy a *mélységi* és *befejezési szám*okat külön *szekvencia* szerint számolja, tehát annak a *csúcs*nak, amibe először lépünk be, a mélységi száma `1`, amibe k-adiknak, annak `k`, amelyikbe utolsónak, annak pedig `n` legyen; és ugyanez legyen igaz a *befejezési számra* is. Mindkettő legyen egyedi és `[1..n]`-beli. Ezzel vajon vesztünk információt a hivatalos változathoz képest?
- Írjuk meg az algoritmust tömbös reprezentációra

- Végezd el az algoritmust ezen a példán. Írd a csúcsok mellé `d/f` alakban a *mélységi* és *befejezési* számaikat, húzd át vastagon azokat az éleket, amik a π szerint a *feszítőerdő* részei. Dicsérd meg magad, hogy jó munkát végeztél, de ne aggódj, most jön a neheze.

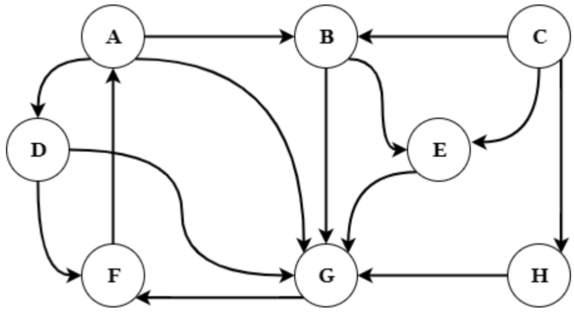


2. ÉLEK OSZTÁLYOZÁSA

Most végignézünk egy nagyobb példát, ahol átismételjük az első fejezetben tanultakat. Azután *osztályozzuk* az éleket a mélységi bejárás során betöltött szerepük alapján. Ennek segítségével hasznos megállapítást tudunk tenni a gráfról.

2.1. PÉLDA

Járjuk be ezt a gráfot *mélységien*:

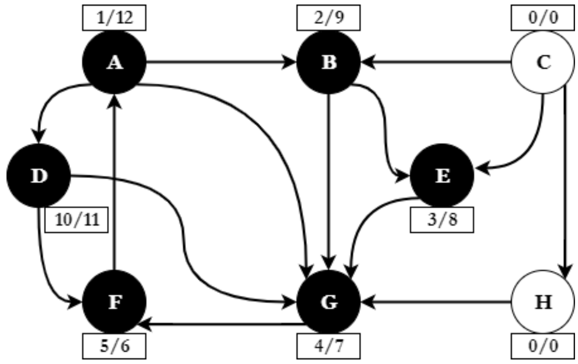


Bár ez nincs eleve elrendelve, de most hozzuk meg azt a döntést, ahol több csúcs közül is választhatunk, ott ábécésorrend szerint fogjuk ezt megtenni. Ez alapján az `A` csúcs lesz az első, amit bejárunk. Tehát ennek a *mélységi száma* `1`, a *befejezési* egyelőre `0`. A *szomszédai* következnek (*irányított* eset, tehát az `F` nem), sorban `B`, `D` és `G`.

`B` mélységi száma tehát `2`. Megyünk tovább ennek a gyerekeivel, előbb `E`-vel, majd `G`-vel…, persze ebből is látszik, hogy mire `A`-ból eljutnánk `G`-be, már mindenképp érintettük ezt a csúcsot.

Amikor `F`-nél tartunk, annak `A` lenne az egyetlen gyerek, de ezt a csúcsot már látogattuk – egész pontosan `A`-ból kiindulva jutottunk el `F`-ig (itt tulajdonképpen egy *kört* találtunk). `F` lesz tehát az első csúcs, amit befejezünk, megkapja a `6`-ost, s a *rekurzió*ban visszalépünk `G`-re. Amikor `B`-nél tartunk, akkor azt látjuk, innen is el tudnánk jutni `G`-be (ráadásul „rövidebben”), de ezt kihagyjuk, mert `E`-n keresztül jártunk már ott. Visszatérünk `A`-ba, innen még `D` érintetlen, elmegyünk hát oda, onnan csupa *fekete* csúcsba jutnánk, visszatérünk ezért `A`-ba, ahonnan `G` az utolsó vizsgálandó gyerek, de ahogy a fentiekben előrevetítettük eddigre már az is *fekete*.

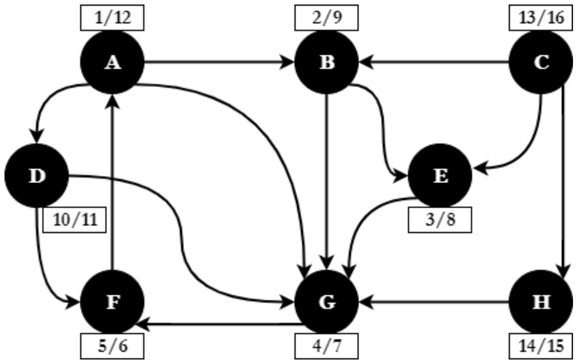
Kilépünk tehát a `dfs()` ciklusáig, most így néz ki az ábra – érdemes papíron megcsinálni, s ellenőrizni, ezen a ponton Ti is itt tartotok-e:



Most a `dfs() for each`ének következő csúcsát néznénk, ez `B`, de mivel *fekete*, nem hívjuk meg a `visit()` függvényt. `C` következik, erre belépünk, hiszen *fehér*.

Megkapja a soron következő *mélységi szám*ot, ami a `13`, `B`-be és `E`-be nem mehet, mert azok *feketék*, de `H`-t innen fedezzük fel `14`-essel. Most már minden csúcs *fekete*, de ezt az algoritmus az általunk tanult változatában nem figyeli, tehát még pörög tovább: `H`-ból `G`-be nem mehetünk, s ez volt az utolsó ellenőrizendő *gyerek*, `H` megkapja a *befejezési számát*, majd visszalép `C`-be. Annak sincs már több gyereke, ezzel megkapja `C` is a *befejezési szám*ot, ami `2n`, azaz `16`.

A főprogram `for each` ciklusa még végignézi a többi csúcsot `D`-től kezdve, de érdemi dolog már nem fog történni, íme a végeredmény:



Illetve a *végeredmény* a π értékekkel teljes, amit megadhatunk egy táblázatban – arra utalva, hogy π implementálható egy tömbbel. Ennek a táblázatnak két sort készítettem, mégpedig azért mert minden *csúcs* `π` értéke *legfeljebb kétszer* kap értéket: *inicializálás*kor, és a csúcs *bejárás*akor, amennyiben a `visit()` függvényből hívtuk és nem a `dfs()`-ből.

π	A	B	C	D	E	F	G	H
Init	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
Vége	NULL	A	NULL	A	B	G	E	C

Mit látunk a táblázatban? Hogy az *eredeti* gráf mely *él*ei adják meg a *mélységi erdő*t. A *kiinduló csúcs*ok rendre a „*Vége*" sorban levő csúcsok, a *célcsúcs*ok pedig a fejlécek. Az `A` és `C` alatt `NULL` értékek maradtak: ezek a *feszítőerdő fá*inak a *gyöker*ei.

Ha nem ábécésorrendben járjuk be a csúcsokat, akkor a fák gyökerei persze más csúcsok is lehetnek.

2.2. ÉLEK OSZTÁLYOZÁSA – KONKRÉTAN ÉS A PÉLDÁN KERESZTÜL

Az eredeti gráf éleit a mélységi bejárás algoritmusa *4*, egymástól *diszjunkt* halmazba osztja be – osztályozza. Ezek:

- *Faélek*
- *Előreélek*
- *Visszaélek*
- *Keresztélek*

Az algoritmust kibővíthetjük egy *élcímkézés*sel, ahol a megfelelő helyen hozzárendeljük a feldolgozás alatt álló élhez a hozzáillő kategóriát. Előbb nézzük szemléletesen, hogy lehet besorolni az éleket a kategóriákba.

2.2.1. Faélek – F

*Faél*nek nevezzük azokat az eredeti gráfbeli éleket, amik az algoritmus futásának végére kialakuló *mélységi erdő* éleit képezik.

Ha az algoritmussal számon tartjuk a π értékeket (egyébként akár elhagyható is lenne ez), akkor könnyű ezeket az éleket beazonosítani: ha `a.π = b` egy adott `a`–`b` csúcspárra, akkor a `b`-ből `a`-ba vezető él *faél* (*irányított* esetben fontos, hogy `b`-ből `a`-ba és nem fordítva).

Futás közben azok az élek bizonyulnak faélnek, amik olyan csúcsba vezetnek, amiket az algoritmus azzal a lépéssel *fedez fel* magának, ami mentén tovább hívja a `visit()` függvényt.

Kicsit megfoghatóbban: ha épp az `(a,b)` élt vizsgáljuk (azáltal, hogy a `visit()`-ben az `u` szerepében `a` van, a `neighbours()` halmazból pedig épp `b`-t érintjük), *akkor és csak akkor* lesz faél ez, ha ekkor `b.c = W`, avagy `b.d = 0`. `a`-ra nincs elvárásunk, de egyébként tudjuk, hogy `a.c = G` és `a.d > 0`, valamint hogy `b.f` és `a.f` is `0`. Mert máshogy nem vizsgálhatjuk ezt az adott élt.

2.2.2. Előreélek – E

Előreél az, amikor a feldolgozás alatt álló csúcs egy olyan gyerekébe vezető élet vizsgálunk, amibe *már* ugyanebből a csúcsból kiindulva, több lépcsőben korábban találtunk egy *utat* (és annak éleit faélekként fel is vettük). Ez most egy közvetlen él, azaz 1 hosszú út, úgymond optimálisabb, de mégis ignoráljuk a *feszítőerdő* szempontjából, hiszen annak lényege, hogy minden csúcsot csak egy úton érjünk el. A mélységi bejárásnak nem tisztje az ilyen értelemben legrövidebb utakat megtalálni.

Formálisabban, ha épp `(a,b)` élet vizsgáljuk, ez előreél lesz *a. cs. a.* ha…

- `b.d > 0` és `b.f > 0`, avagy `b.c = B`
  ÉS
- `a.d < b.d`

Megj.: tulajdonképpen a `b.d > 0` feltételt nem is kell vizsgálni, mert egyrészt ez `b.f > 0`-ból következik (nem fejezhetünk be egy még meg se kezdett csúcsot); másrészt ha `a.d < b.d`, akkor ebből is következik, hogy `b.d > 0`, hiszen `a.d` nem lehet negatív – ami azt illeti, 0 se, mert ha `a`-ból kiinduló élet vizsgálunk, akkor `a.c = G`, azaz `a.d > 0`.

2.2.3. Visszaélek – V

Akkor találtunk *visszaélt*, ha a feldolgozás alatt álló csúcs egy olyan szomszédjába vezető élt vizsgálunk, amire igaz, hogy a feldolgozás alatt álló csúcsot ebből a szomszédból fedeztük fel (akár *közvetlenül*, akár *több lépésben*).

Azaz, ha találtunk egy *kört*, ami akár egy oda-vissza élpár is lehet a két csúcs között.

Számszerűen: ha épp az `(a,b)` élt vizsgáljuk, ahol `b.d > 0`, de `b.f = 0`, azaz `b.c = G`.

2.2.4. Keresztélek – K

*Keresztél*ről akkor beszélünk, ha az adott él egy másik *mélységi fá*ba (vagy *mélységi részfá*ba) vezet.

Szabatosabban: ha `(a,b)` élt vizsgáljuk és …

- `b.d > 0` és `b.f > 0`, azaz `b.c = B`
  ÉS
- `a.d > b.d` (azaz nem `a`-ból jutottunk áttételesen `b`-be eredetileg se)

Megj.: ez hasonló az *előreél* esetéhez, viszont itt az `a.d > b.d` feltétel akkor is igaz, ha `b.c = W`, de ez nekünk nem jó, így a `b.d > 0` legalábbis ebből nem következik – `b.f > 0`-ból persze továbbra is.

2.2.5. Összefoglalás

Egy csúcs lehetséges színei/számai:

Zárójelbe írtam, ami következik a vele konjunkcióban levő másik állításból.

Egyéb kombináció `d`-re és `f`-re lehetetlen.

`a.c = W`	`a.d = 0` (és `a.f = 0`)
`a.c = G`	`a.d > 0` és `a.f = 0`
`a.c = B`	(`a.d > 0`) és `a.f > 0`

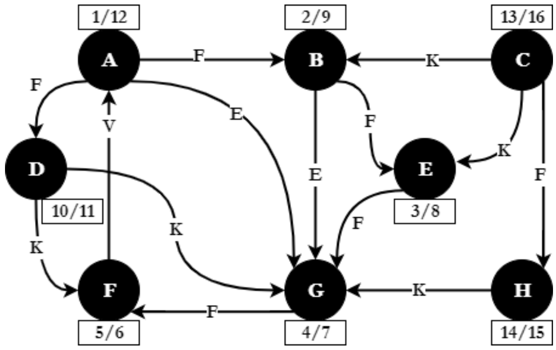
Amikor `(a,b)` élt vizsgáljuk az alábbi és csak az alábbi esetek lehetségesek:

Először is, ez mindenképp igaz: `a.c = G` és `a.d > 0` és `a.f = 0`.

`b`-re (illetve `a` és `b` viszonyára pedig ezeket kell ellenőrizni):

(a,b) F	b.c = W	
(a,b) E	b.c = B	a.d < b.d
(a,b) V	b.c = G	
(a,b) K	b.c = B	a.d > b.d

A példa befejezése:



Jól látszik, ebben van mindenféle él. A `K`-k a részfák között, a `V` a körben, az `E` ahol le tudtuk vágni az utat. `F`-fekkel pedig kirajzolódik két fa.

2.3. MIRE JÓ EZ?

*Kör*figyelésre.

Ha *faél*t találtunk, az a fa része, a fában nincs *kör*.

Ha *előreél*t leltünk (*irányított* esetben), az se kör, csak találtunk egy gyorsabb utat két pont között.

Ha *keresztél*be botlottunk, akkor azt látjuk, hogy a gráf két „*komponense*" között mégiscsak van kapcsolat, bár eddig nem gondoltuk – ha a gráf egy *folyamat* menetét írja le, akkor két függetlennek tűnő részfolyamat között bizonyos ponton lehet az egyik előfeltétele a másiknak. De kört az ilyen élek se jelentenek.

Ha *visszaél*re bukkantunk, akkor egy olyan utat találtunk két csúcs között, ami (akár hosszabb úton) már fordítva jelen volt. Megint, ha arról beszélünk, hogy egy folyamatban bizonyos lépés előfeltétele egy másiknak, akkor ilyenkor azzal szembesülünk, hogy A is előfeltétele B-nek, de egyben B is A-nak. Tehát a folyamat nem teljesíthető.

Tehát egy ilyen gráfban *akkor és csak akkor* van *kör*, ha egy *mélységi bejárás*ában van *visszaél*.

Ha a gráf egy folyamat lépéseit és a köztük levő előfeltételi rendszert írja le, akkor csak *körmentes gráf*nak van értelme. Erre jó ez.

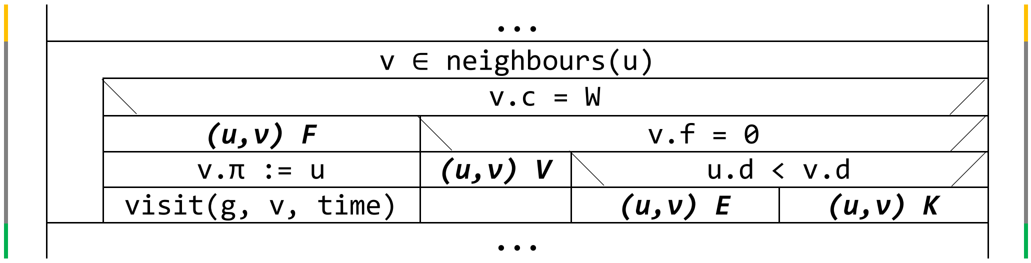
2.4. DAG-TULAJDONSÁG

DAG: *directed acyclic graph*. Tehát egy olyan *irányított* gráf, ami *körmentes*. Azaz, nincs benne *visszaél*.

Hogy dönthetjük ezt el?

Tekintsük a `visit()` függvényt. A **szürke** részében találunk egy elágazást, ami azt dönti el, hogy hívjuk-e tovább rekurzívan a `visit()`-et. Azaz, hogy az adott, feldolgozás alatt álló csúcs adott szomszédját az aktuális élen keresztül fedeztük-e fel. Ez épp a „*faél*" definíciója.

Álljon itt most a szürke rész alternatív megvalósítása, immár az *élek osztályozásá*val kiegészítve. Ha bármikor is *visszaél*t találunk, a gráf nem DAG, ha nem találunk, akkor az.



A `v.f = 0` elágazás igaz ága azt jelenti, `v.c = G`, mivel ugyan ezzel csak a `v.c = B`-t zárjuk ki, de a `v.c = W` feltétel hamis ágában vagyunk eleve.

2.5. FELADATOK

2.5.1. Villámkérdések

Ahol lehetett, a válaszokat indoklás nélkül a kérdések mögé írtam fehérrel – kimásolással ellenőrizhetők. Konzultáción beszéljük át részletesebben.

- Azt tudjuk, ha nem ábécésorrendben vesszük a csúcsokat, mások lehetnek a fák *gyökerei*. De lehetnek mások a *feszítőerdő* fáiban résztvevő csúcsok is? Lehet-e olyan, hogy egyik bejárással *feszítőfát* kapunk, a másik sorrend szerinti bejárással több fából álló *erdőt* ugyanarra a gráfra?

- Azt tudjuk, hogy a *színezés* elhagyható, mert ami információt tartalmaz, azt a `d/f` számokból ki tudjuk következtetni. De vajon megírható-e úgy az algoritmus – körfigyeléssel! –, hogy a `d/f` számokat hagyjuk el, és a színezést tartjuk meg?

- Igaz-e: a *faél* egy fa éle?

- Igaz-e: ha a bejárás során azonos *gyökérből* két utat is találunk egy adott csúcshoz, és abból az egyik kevesebb élből áll, akkor itt legalább az egyik él *előreél*?

- Igaz-e: a *hurokél visszaél*?

- Mi van, ha *irányítatlan* a gráf és *visszaélt* találunk? (Megj. A DAG eleve irányított gráfot ír elő)

2.5.2. Komplexebb feladat

- Osztályozd az előző fejezet utolsó feladatában levő gráf éleit a mélységi bejárás után. Döntsd el, ez a gráf DAG-e

3. A DFS ALKALMAZÁSAI

Álljon itt néhány a *körfigyelés*en túlmutató gyakorlati alkalmazás.

3.1. TOPOLOGIKUS RENDEZÉS

A *topologikus rendezés*, vagy *topologikus sorrend* egy *DAG*-okra értelmezett fogalom.

Egy gráf topologikus rendezése a csúcsok egy olyan nem feltétlen egyértelmű felsorolása, amire minden a gráfbeli `(u,v)` élre teljesül, hogy a topologikus sorrendben `u` megelőzi `v`-t.

Azaz ha egy ilyen sorbarendezésben `u` megelőzi `v`-t, akkor a gráfban nem lehetett `v`-ből `u`-ba menő él (se *út* – hiszen DAG, azaz *körmentes*). Az, hogy `u`-ból `v`-be menjen, az nem elvárás, csak lehetőség.

Ahogy korábban említettük, egy érvényes alkalmazás, hogy a gráf *csúcs*ait egy munka részeinek, a köztük levő *él*eket a munkarészek közötti előfeltétel-viszonyoknak tekintjük. Ilyenkor a *topologikus rendezés* a munka elvégzéséhez szükséges lépések egy *lehetséges* sorrendjét adja meg.

Ha `A`-ból és `B`-ből is vezet él `C`-be, akkor egy ilyen alkalmazásban mondhatjuk, hogy `A` és `B` is *előfeltétele* `C`-nek, azaz `C` csak akkor végezhető el, ha `A` és `B` is kész. Ekkor érvényes topologikus sorrend az `<A, B, C>` és a `<B, A, C>` is, hiszen `A` és `B` egymáshoz képest levő sorrendjére nem tettünk megkötést. Ha `A`-ból `B`-be is vezet él, akkor csak az `<A, B, C>` sorrend lehet a helyes. Ha pedig `C`-ből `A`-ba vezet él a fentiek mellett, akkor nem DAG, nincs topologikus sorrend.

3.1.1. Meghatározása befokokkal

Az alábbi algoritmust használhatjuk a *topologikus sorrend* előállítására:

1. Határozzuk meg a csúcsok *befok*ait (tehát, hogy hány él vezet beléjük)
2. Keressünk egy 0 befokút, és írjuk ki – ilyen csúcs mindig lesz, a későbbi lépésekben is, feltéve, hogy a gráf DAG
3. *Töröljük* ezt a csúcsot, az éleivel együtt – *módosítsuk* ennek alapján a többi csúcs befokát
4. GOTO 2.

Ez egy egyszerű elv, de az *implementáció*jához fogadjunk el egy-két tanácsot:

Nem igazán szerencsés, ha *ténylegesen* törlünk az eredeti gráfból. Ezt nem mindig tehetjük meg, vagy nem lesz hatékony, átlátható, stb.

Az elején határozzuk meg a *befok*okat (pl. egy *segédtömb*be) és csak a 0 befokú csúcsokat gyűjtsük egy sorba (`Queue`). Járjuk végig ezt a sort. Az aktuálisan feldolgozott csúcsra vegyük az összes kimenő élét, a cél-csúcsok tárolt befokait csökkentsük, de se csúcsot, se élt ne töröljünk sehonnan. Ha ilyenkor egy *befok* eléri a 0-t, a hozzá tartozó csúcsot rakjuk a sorba. És így tovább, míg el nem fogy.

Mivel csak *DAG*-okra várhatunk topologikus rendezést, szerencsés ha az algoritmusban *körfigyelés* is van. Ez egyszerű, és összességében nem plusz művelet: csak azt kell megnéznünk, amikor végeztünk az algoritmussal, azaz kifogyott a sor, akkor minden csúcsot bejártunk-e. Ehhez egy számlálót kell növelnünk minden körben, s a cél: ennek *pontosan* `n` legyen az értéke a végére.

3.1.2. Meghatározása DFS-sel

Futtassuk le a *mélységi bejárás*t a gráfon, közben a tanult módon figyeljük a *visszaél*eket, hogy eldönthessük, egyáltalán *DAG*-e – azaz létezik-e *topologikus sorrend*je.

A *DFS* közben jegyezzük fel minden csúcshoz a *befejezési szám*okat.

Egy lehetséges *topologikus rendezés*t kapunk, ha a befejezési számok szerinti *csökkenő* sorrendben felsoroljuk a csúcsokat.

Implementációhoz tipp: nem is kell „feljegyezni” a befejezési számokat, elég ha befejezéskor egy *verem*be (`Stack`) rakjuk az érintett csúcsot, s a végén ezen verem elemeit írjuk ki.

3.1.3. Példa

Álljon itt egy „*szöveges feladat*”:

A következőkben megadom, hogyan kell sajttortát készíteni. Néhol egyszerűsítek és a konkrét mennyiségeket is elhagyom, mert nem ez a lényeg:

Törjünk össze egy adag kekszet. Olvasszunk vajat, adjuk hozzá. Nyomkodjuk ezt az alapot egy sütőformába. Süssük meg. Közben egy keverőtálban a krém rész hozzávalóit keverjük össze. A kihűlt alapra öntsük ezt a masszát, majd süssük tovább. Fogyasztás előtt hűtsük le.

A feladat, hogy adjunk meg egy listát, *lépésről lépésre* melyik teendő mit kövessen, amennyiben nem akarunk párhuzamosan több részen dolgozni.

Mik itt az egyes részfolyamatok?

Az „alap” esetén:

- Keksz összetörése (`AK`)
- Vaj olvasztása (`AV`)
- Keksz és vaj összekeverése (`AÖ`)
- Formába nyomkodása (`AF`)
- Sütés (`AS`) – itt persze lehetne szó az előmelegítésről, illetve nem mindegy mi mennyi ideig tart, de ettől is tekintsünk most el

Itt az `AK` és `AV` folyamatok függetlenek, de meg kell előzzék `AÖ`-t, ami megelőzi `AF`-et, az pedig megelőzi `AS`-t.

A krém esetén:

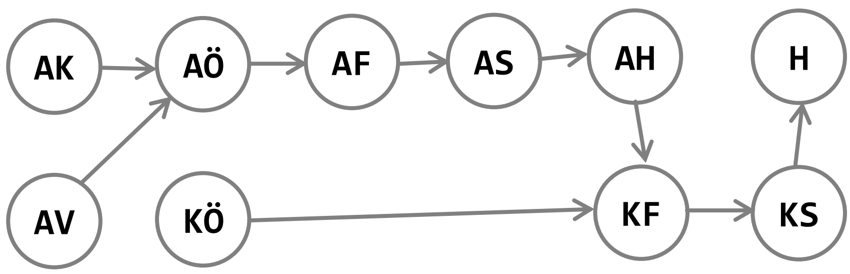
- Hozzávalók összekeverése (`KÖ`)
- Sütőformába öntés (`KF`)
- Sütés (`KS`)

Itt `KÖ` biztosan `KF` előtt van, végül pedig jöhet `KS`.

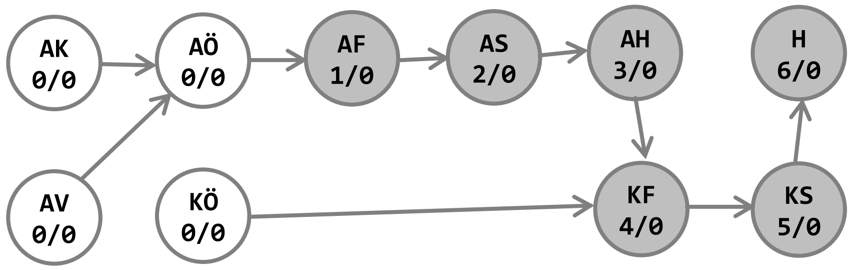
Az alapot és a krémet valamilyen szinten lehet *párhuzamosan* készíteni (és mivel az első sütés ideje egyébként nagyon rövid, ennek a gyakorlatban lehet is értelme). A `KF` lépés az, aminek előfeltétele nemhogy csak meg legyen sütve az alap, hanem hogy ki is legyen hűlve. Ehhez vegyünk fel még egy lépést (`AH`).

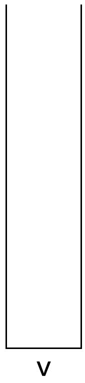
A végső sütés után még le kell hűteni a tortát (`H`).

Ezek alapján az alábbi *irányított gráf*ot rajzolhatjuk fel:

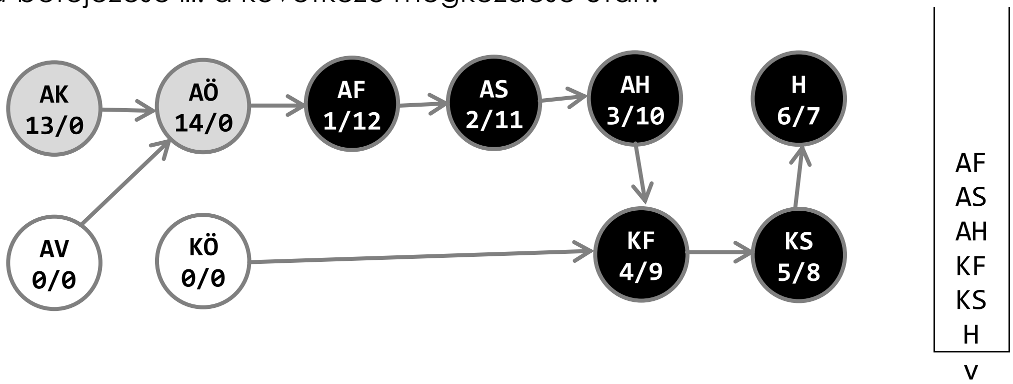


Járjuk be *mélységi bejárás*sal, befejezéskor gyűjtsük egy *verem*be a csúcsokat. Ha *kör*t találunk, szakítsuk meg a ciklust és termináljuk az algoritmust. Mint eddig, most is címkék szerinti ábécésorrendben fogok haladni:

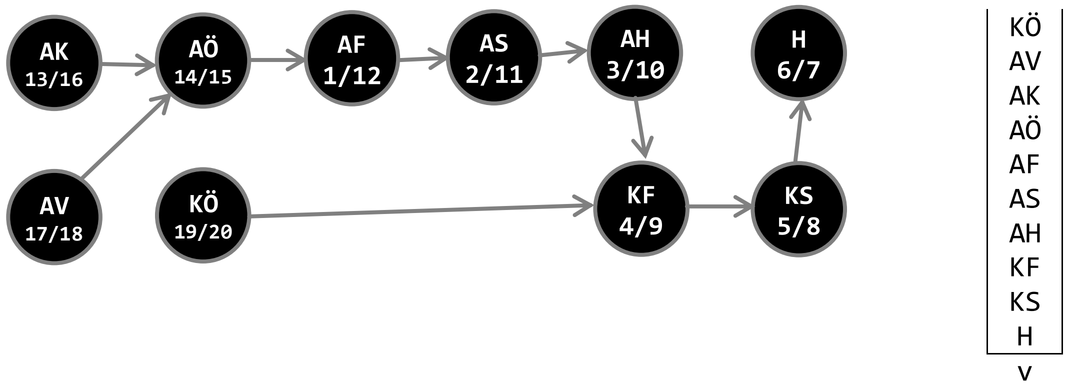




Az első *mélységi fa* befejezése ill. a következő megkezdése után:



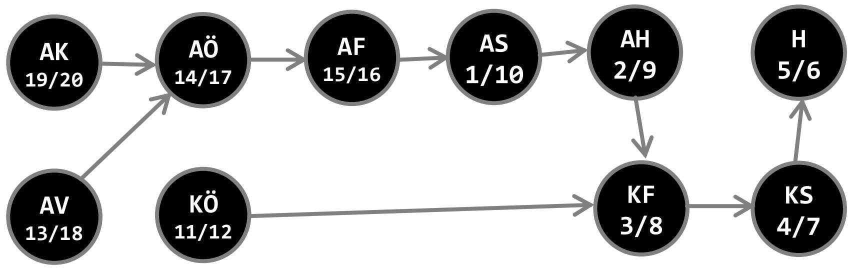
Végeredmény. A *topologikus sorrend* a verem tartalma „felülről lefelé” azaz a kivétel sorrendjében:



Tehát az algoritmus szerint a krém összekeverésével kezdünk és csak ezután készítjük el a tortaalapot. Ez elfogadható, hiszen az a lényeg, hogy addig ne *öntsük rá* a krémet az alapra, amíg az nem készült el. Ez pedig teljesül.

Mivel itt nincs olyan csúcs, amiből több él is kivezetne, azaz nincs olyan lépés, ami *több* másik lépésnek is *előfeltétel*e volna, ezért egy-egy részfa esetén a következő csúcs (és így a következő lépés) meghatározásában nem volt mozgásterünk, de ahol több előfeltétel volt, ott más mélységi bejárással más féle sorrend jött volna ki ezen lépések között (pl. az AK megelőzhette volna az AV-t, amíg mindkettő az AÖ előtt marad; ill. az AH megelőzhette volna a KÖ-t, amíg mindkettő a KF előtt marad).

Természetesen olyan *topologikus sorrend* is lehetséges, ahol az alapkészítés *szekvenciá*ja közé ékeljük a krém összekeverését, azaz pl. ez: AK, AV, AÖ, AF, KÖ, AS, AH, KF, KS, H. Talán elsőre nem is nyilvánvaló, de ez a konkrét sorrend is előállhat a mélységi bejárásos módszerrel, amennyiben így járjuk be a gráfot (a fák gyökerei rendre: AS, KÖ, AV, AK) – érdemes végiggondolni, hogy lehet tudatosan pont ilyen sorrendet előállítani:



3.2. ERŐSEN ÖSSZEFÜGGŐ KOMPONENSEK

Azt tudjuk, hogy egy gráf lehet *összefüggő* és *nem összefüggő*. Az összefüggőségnek lehet egyfajta „mértéke" is, illetve egy gráfon belül lehet értelme összefüggő részgráfokról beszélni.

3.2.1. Definíció

Egy *irányított gráf* (vagy annak egy *komponense*) *erősen összefüggő*, ha minden `u`, `v` csúcsára igaz, hogy van irányított út `u`-ból `v`-be **és** `v`-ből `u`-ba is.

Azaz egy olyan irányított gráf, vagy annak részgráfja, ahol *minden csúcs* minden másik csúcsból elérhető, noha nem feltétlen *teljes gráf*ról beszélünk.

Angolul: *Strongly Connected Components* (SCC).

3.2.2. Meghatározása (*Kosaraju*-algoritmus)

1. Futtassuk le a *mélységi bejárás*t tetszőleges sorrendben. A π tömböt nem kell meghatározni, de tegyük *verem*be a csúcsokat befejezéskor – hasonlóan a topologikus sorrend meghatározásának algoritmusához

2. Készítsünk egy olyan gráfot, ami az eredeti *transzponált*ja (azaz ugyanazon csúcsokkal, de fordított irányú élekkel)

3. Futtassuk le újra a mélységi bejárást, de most a transzponált gráfon. A `dfs()` ciklusában a csúcsokat kifejezetten a verembeli sorrend alapján válasszuk meg. Azaz kezdjük a *veremtető* csúcsával, és amikor a `visit()` függvény teljesen visszatér, folytassuk a verem legfelső még *fehér* csúcsával, stb.

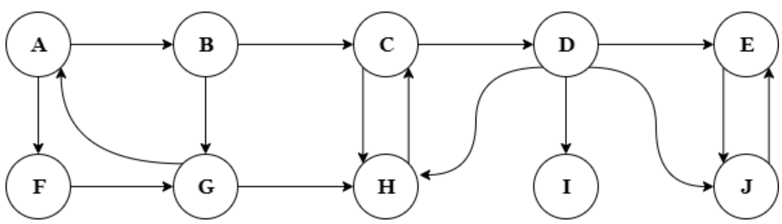
4. A második bejárás szerint egy *mélységi fá*ba tartozó csúcsok definiálnak egy *erősen összefüggő komponens*t. Akkor „kezdünk" új mélységi fát, amikor a rekurzió visszatér a `dfs()` ciklusába és új csúcsot húzunk a veremből

Vigyázat, bár a *transzponált* gráf szerint végezzük a második DFS-t, de nyilván a kapott komponensek az eredeti gráfra vonatkoznak.

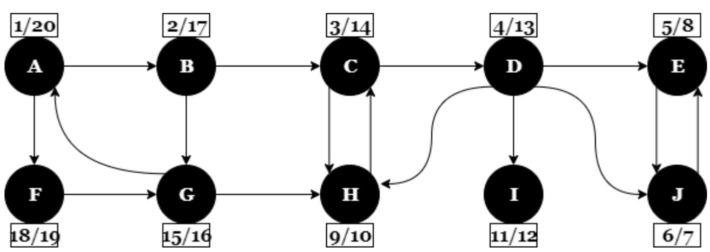
Műveletigény: a két DFS: `2(n+e)`, a transzponálás: `e`. Összesen `Θ(n+e)`.

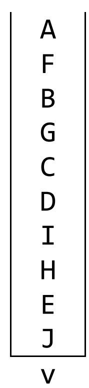
3.2.3. Példa

Határozzuk meg az *erősen összefüggő komponens*eket ebben a gráfban:



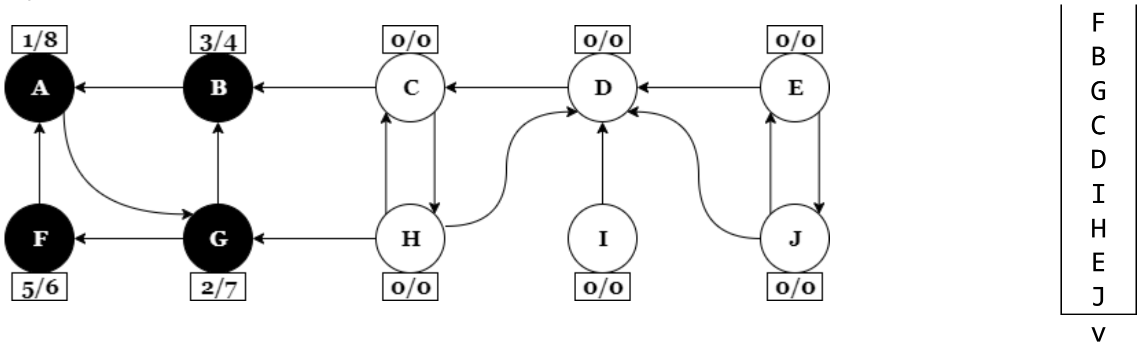
Az első *mélységi bejárás* után, a feltöltött veremmel (döntéshelyzetben ábécésorrendben választottam a következő csúcsot):



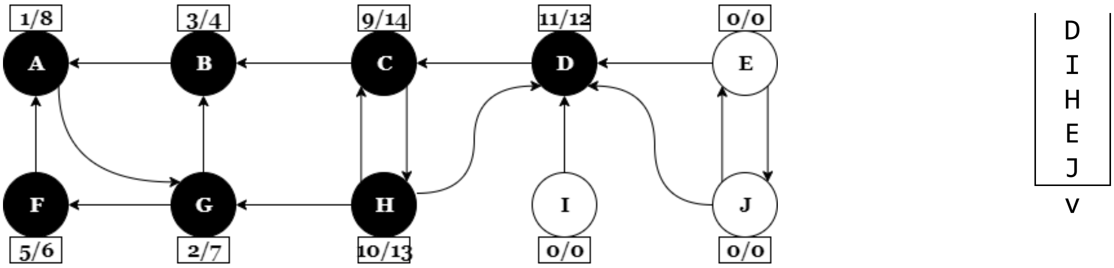


Most a *transzponált gráf*on kezdek mélységi bejárást. Az A csúccsal fogom kezdeni, de nem az ábécésorrend miatt, hanem mert az a *verem* teteje. Amennyiben a *rekurzió*ban egy csúcsnak több gyereke is van, maradok az ábécésorrendnél, csak a `dfs()` ciklusát táplálom a verem diktálta sorrendben.

A következő ábrán a *transzponált gráf* bejárása látható az első „elakadásig", azaz az első olyan helyzetig, amikor elakad a rekurzív `visit()`-hívássorozat. Ezzel tehát a transzponált gráfon előállt a *mélységi erdő* első *fá*ja. A résztvevő csúcsok – A, B, F, G – fogják alkotni az **eredeti** gráf egyik *erősen összefüggő komponens*ét.



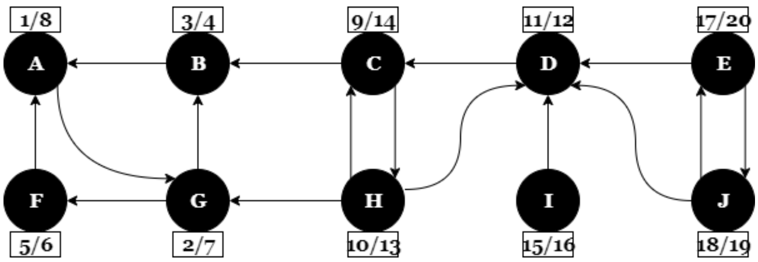
Most menjünk tovább. A *veremtető*vel kéne folytatnunk. F a veremtető, de mivel ez a csúcs már *fekete*, csak eldobjuk. B és G is így jár, tehát végül C-ből indítjuk a következő rekurziót.



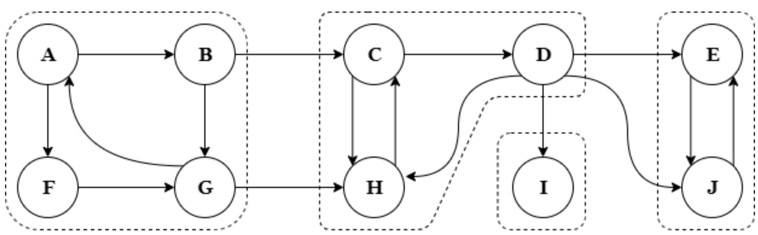
Ez alapján a második *komponens* csúcsai: C, D, H.

D-vel nem tudjuk folytatni, mert már *fekete*, tehát most I-ből indulunk ki. I-ből egy darab *él* vezet ki, az is egy *fekete* csúcsba, tehát itt a rekurzió csak egy lépéses volt: I önmagában alkot egy *komponens*t. Ez logikus is. Az eredeti gráfban I kifoka 0, tehát hiába érhető el esetleg I más csúcsokból, de I-ből egy csúcs sem, azaz *kölcsönösen* nem elérhető senkivel – és épp ez kellene az egy erősen összefüggő komponenshez tartozáshoz.

Végül, a H jönne, ami már *fekete*, ezért az E-vel folytatjuk, az egy újabb kétcsúcsú komponenst definiál (E, J), és ezzel végeztünk is a bejárással.



Az alábbi *komponens*eket határoztuk meg:



3.3. FÉLIG ÖSSZEFÜGGŐSÉG

Ahogyan a *topologikus rendezés* a *mélységi bejárás*ra épített, valamint az *erősen összefüggő komponensek* algoritmusa a topologikus sorrendre, úgy fog ez az algoritmus az utóbbin nyugodni.

3.3.1. Definíció

Egy *irányított gráf félig összefüggő*, ha minden `u`, `v` csúcsára igaz, hogy van irányított út `u`-ból `v`-be **vagy** `v`-ből `u`-ba.

Ez „*megengedő vagy*”, azaz ha `u` és `v` között van oda-vissza út, az is elfogadható, de nem elvárás. Az viszont elvárás, hogy minden csúcspárra legalább az egyikből el lehessen jutni a másikba.

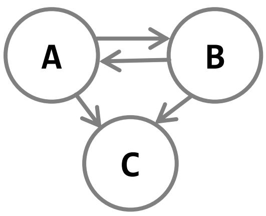
Ebből az is következik, hogy egy *erősen összefüggő gráf* egyben *félig összefüggő* is, illetve minden erősen összefüggő *komponens* egy félig összefüggő *részgráf* is.

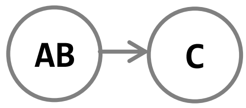
Angolul: *Semi-connectivity*.

3.3.2. Algoritmusa

1. Határozzuk meg a gráf *erősen összefüggő komponens*eit

2. Ebből készítsük el a *komponensgráf*ot
   - Azaz az *erősen összefüggő komponens*eket „vonjuk össze” egy csúcsba, a köztük levő éleket hagyjuk el, a belőlük kimenő/bemenő éleket tartsuk meg egyszeres számossággal
   - Példa: bal oldalon az *eredeti*, jobb oldalon a *komponensgráf*. Itt `A` és `B` egy komponens, `C` egy másik. Ha akár `A`-ba, akár `B`-be ment volna él `C`-ből is, akkor ez az egész egy komponens lenne és minden élt elhagytunk volna. Ha az `(A,C)` vagy `(A,B)` élek közül az egyik nem lenne, a komponensgráf akkor is ugyanez lenne





3. Ha most van két olyan komponens, ami között nincs *út*, biztosan nem lesz félig összefüggő a gráf

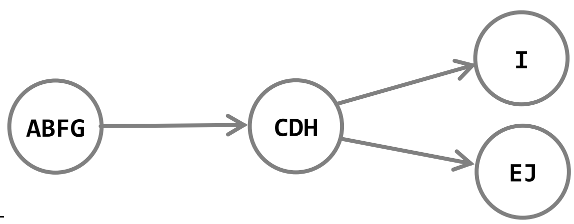
4. Tudjuk, hogy a komponensgráf *DAG*[1]. Emiatt van értelme rá *topologikus sorrend*et venni. Nézzünk egy ilyet

5. Akkor *félig összefüggő* a gráf, ha `u` és `v` a topologikus sorrendben két tetszőleges egymást közvetlenül követő csúcs esetén van `(u,v)` él a komponensgráfban
   - Megj.: ha ez igaz, akkor egyébként biztosan csak egy féle topologikus sorrend lesz

A műveletigény így $\Theta(n+e)$ tud lenni. Floyd-Warshall algoritmussal is megoldható ez a kérdés, de annak $\Theta(n^3)$ lenne a műveletigénye.

3.3.3. Példa

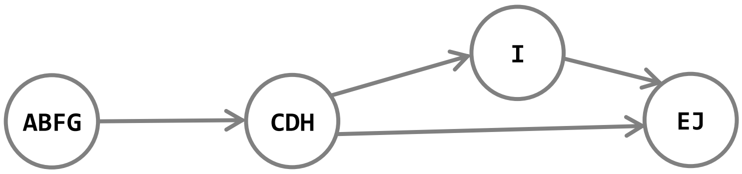
Folytassuk az előbbi példát. Az előző oldalon láthatjuk a komponenseket, ebből a *komponensgráf*:



[1] Ha nem így lenne, akkor lenne irányított kör, de akkor az azt jelenti, hogy a körben részt vevő eredeti csúcsok között is volt kör…, azaz azok egy erősen összefüggő komponenst alkotnának. De akkor a komponensgráfban nem tartozhatnak különböző csúcsokhoz, tehát mégse lenne kör. Ellentmondás

`I` és `EJ` között nincs út, tehát `I` komponens csúcsai és `EJ` komponens csúcsai között nincs semmilyen irányú út az eredeti gráfban. Ez ránézésre is könnyen ellenőrizhető. Ez a gráf tehát nem *félig összefüggő*. És ebből következően nem is *erősen összefüggő*.

Húzzunk be az eredeti gráfban egy `(I,E)` élt. Ellenőrizhető, ez nem okoz semmilyen különbséget az előző lépések folyamán, viszont a komponensgráf így fog kinézni:



Így már minden két csúcs között van út legalább (és legfeljebb) egyik irányba.

Vegyük a *topologikus sorrend*et. Az `ABFG` befoka 0, ezért ez lesz az első csúcs. Most, ha ezt elvennénk, a `CDH` befoka lenne 0, tehát ez lesz a következő. Most az `I` jönne, végül az `EJ`. Minden csúcs befoka 0 lenne egy ponton, és minden csúcs sorra kerülne. És mivel mindig pontosan egy szóba jövő csúcs volt, a topologikus sorrend egyértelmű: `ABFG`, `CDH`, `I`, `EJ`.

Most azt kell megnézni, megvannak-e az alábbi élek a komponensgráfban:

- `(ABFG, CDH)`
- `(CDH, I)`
- `(I, EJ)`

Megvannak. Tehát a gráf így már *félig összefüggő*.

3.4. Feladatok

3.4.1. Villámkérdések

Ahol lehetett, a válaszokat indoklás nélkül a kérdések mögé írtam fehérrel – kimásolással ellenőrizhetők. Konzultáción beszéljük át részletesebben.

- Milyen gráfra lesz *egyértelmű* a topologikus sorrend?

- Milyen gráfra érvényes topologikus sorrend a csúcsok bármely *permutáció*ja?

- *Irányítatlan* gráfra van-e értelme a topologikus sorrendnek?

- Igaz-e, hogy mindkét tanult módszer *helyes* és *teljes* – azaz minden topologikus sorrend előállítható velük – és csak azok?

- Mennyi a topologikus sorrend meghatározásának *műveletigény*e? Múlik-e a reprezentáción, ill. a tanult algoritmus megválasztásán? Rosszabb-e *kördetektálás*sal? (Mondjunk *minimális*, *maximális*, *átlagos* műveletigényt)

- Mennyi az *összes lehetséges* topologikus sorrend kiírásának műveletigénye?

- Mit jelent, ha egy gráfban egy darab *erősen összefüggő komponens* van? És ha `n`?

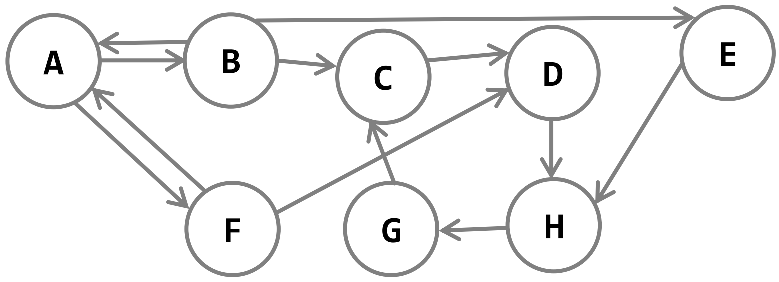
3.4.2. Komplexebb feladatok

- Ábrázoljuk irányított gráfként az alábbi folyamat lépéseit a köztük levő előfeltétel-viszonyokkal. Járjuk be mélységien, s erre építve határozzunk meg egy topologikus sorrendet:

A félév elején jelentkezni kell egy kurzusra. A félév során a jegyszerzéshez két zh-t kell megírni (előbb az elsőt, majd a másodikat), és be kell adni két házi feladatot (mindegy mikor, csak a szorgalmi időszak alatt legyen). Ezek után vagy gyakuv-re megy a hallgató, vagy vizsgázni. A gyakuv után is elérhető a vizsga.

❖ Írjuk meg a mélységi bejárásra alapozott topologikus sorrendet megállapító algoritmust éllistás reprezentációra

❖ Döntsük el az alábbi gráfról, hogy félig összefüggő-e



TARTALOM