Minimális költségű feszítőfák

➢ 2020. ősz – Szita B.

1. Bevezetés

Ebben a fejezetben ismertetjük a feladatot és mutatunk rá két (igen hasonló) absztrakt megoldási stratégiát, amit majd a későbbi fejezetekben fogunk konkrétabb, implementációközelibb szinten részletezni.

1.1. Feladat

Szemléletesebben: adott egy hálózat, amiben a csúcsok közötti összeköttetésekre több alternatíva is van. Pl. egy autópálya-hálózat lehetséges nyomvonalai, mindre költségbecsléssel. Válasszuk ki, azokat az összeköttetéseket, amikkel a legkisebb összköltséggel az összes pontot a közös hálózatra tudjuk kapcsolni.

Hivatalosabban: adott egy **összefüggő**, **irányítatlan**, **élsúlyozott** gráf. Keressük olyan *részgráf*ját, aminek a csúcsai *azonosak* az eredeti gráf csúcsaival, a részgráf *fa* (e kettő feltétel jelenti azt, hogy „*feszítőfa*"), és éleinek **összesített súlya minimális**.

- Miért *összefüggő*? – mert nem lehetne feszítőfája, ha nem lenne az
- Miért *irányítatlan*? – mert általában ezen feladatoknak (közmű-vezetékek, autópálya-építés, stb.) így van értelmük; egyébként irányított gráfokra a lenti módszerek nem is működnének!
- Miért *élsúlyozott*? – megint csak: általában így értelmesek a való életből vett feladatok, viszont az ismertetett algoritmusok élsúlyozatlan esetre is működőképesek, viszont ilyenkor a kisebb komplexitású szélességi bejárás is előállítaná a megoldást. Az élsúlyok lehetnek negatívak is

Angolul *Minimum Spanning Tree* (MST).

1.2. Általános algoritmus

Ezt a részt az előadó honlapján megtalálható hivatalos jegyzettel együtt, azt kiegészítve olvassátok. A pontos definíciókat itt nem is közlöm, a szimbólumok és fogalmak magyarázatára szorítkozom.

Az alapalgoritmus a következőképpen szól:

1. Vegyünk egy *üres* élhalmazt
2. `n-1` alkalommal vegyünk ehhez hozzá mindig egy „*biztonságosan hozzávehető*" élt
3. A végén a részgráf *csúcsai* az eredeti gráf csúcsai, az *élei* az így felépített élhalmaz lesznek

Jelölések:

- Az eredeti gráf legyen `G = (V,E)`. `V` a csúcsok (*vertices*), `E` az élek (*edges*) halmaza
- Legyen `n = |V|`
- Az épülő élhalmazt majd `A`-nak hívjuk
- A kész feszítőfát `T`-vel (*tree*) jelöljük, ami egy (V,T_E) pár, azaz a csúcsai az eredeti gráf csúcsai (ezért nincs alsó indexezés), az éleire pedig igaz, hogy T_E részhalmaza `E`-nek (azaz `G` éleinek – ez nem feltétlen valódi részhalmaz-viszony, mert `G` lehet, hogy eredetileg is fa volt)
- A végén $A = T_E$ lesz

- Az `MST` a *minimum spanning tree* kifejezés rövidítése, ez a tulajdonság lesz igaz `T`-re (tulajdonság, mert az ilyen részfa nem feltétlen egyedi)

Két kérdés adódik: miért `n-1` kör van és mi az a „*biztonságosan hozzávehető*”?

- `n-1` élet veszünk fel, mert `n` csúcs van, egy `n` csúcsú *fá*nak biztosan `n-1` éle van

- *Biztonságosan hozzávehető* egy él, ha egy `A`-hoz megfelelően megválasztott *vágás*ban *könnyű* él

Ez persze újabb kérdéseket vet fel… Mi az a *vágás*? Hogy válasszam meg `A`-tól függően? Mi az, hogy egy él egy vágásban *könnyű*?

- A „*vágás*” a gráf *csúcs*ainak *két* **nemüres** részhalmazra való bontása, *osztályozás*a. Minden csúcs pontosan az egyikbe kerül

- A lehetséges vágások közül nekünk egy olyat kell választani, ami *elkerüli* `A`-t

- Ha van egy *vágás*, ami *elkerüli* `A`-t, akkor a vágás *egyik legkisebb súlyú* élét nevezzük *könnyű*nek

Oké…, most már tudunk majdnem mindent, de mi az, hogy a „*vágás elkerüli az élhalmazt*” és mikor a *vágás éle* egy él?

- Akkor *kerüli el* a vágás `A`-t, ha `A` **minden** éle (`A` továbbra is egy *élhalmaz*) olyan, hogy a két *végpont*ja *ugyanabban* a halmazban van (ezek pedig *csúcshalmaz*ok). Azaz a vágás két részhalmaza között nem lehet `A`-beli él

- Ezt úgy mondjuk szépen, hogy `A` egyik éle se *keresztezi a vágást* (a keresztezés jelentené azt, hogy a két halmaz között megy az él)

- Az, hogy a *vágás éle*, az pedig azt jelenti, hogy a vágásban szereplő két csúcshalmaz között megy, azaz a már bevezetett fogalommal élve keresztezi azt

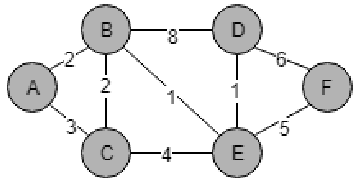
Foglaljuk össze:

- Tegyük `n-1`-szer a következőt:
  - Vágjuk fel két részre a csúcsokat úgy, hogy az eddigi feszítőfa-élek ne menjenek a két halmaz között (ezt az elején triviális, aztán egyre nehezebb megcsinálni, de nyilván minden körben megvalósítható)
  - A két halmaz között futó élek közül válasszuk ki az egyik legolcsóbbat és vegyük a feszítőfához

Ekkor nyilván mindig a *legolcsóbb* olyan élt választjuk, ami egy addig leválasztott részt az épülő feszítőfába integrál, így hát minden *csúcs*ot végül is a legolcsóbb *él*lel értünk el.

1.3. PÉLDA

Tekintsük az alábbi gráfot:



Élsúlyozott, *összefüggő*, *irányítatlan*, tehát a fenti algoritmus alkalmazható.

Lévén 6 csúcsa van, 5 körben fogjuk elkészíteni a fát.

A következőkben a *vágás*okat abszolút „önkényesen” választom, semmilyen sorrend nincs kőbe vésve.

Tehát `A` lesz a minden körben bővülő *élhalmaz*, `S` és `V\S` lesz a *vágás*.

1. kör

- `A = {}`
- `S = {A,B,C}`, következésképp `V\S = {D,E,F}`
- A vágást keresztező élek: `(B,D)`, `(B,E)`, `(C,E)`
- Ebből a minimális, ezt veszük `A`-hoz: **`(B,E)`**

2. kör

- `A = {(B,E)}`
- `S` nem lehet ugyanaz, mert a `(B,E)` él már keresztezne `A`-ra
  → `S = {A, C}`, `V\S = {B, D, E, F}`
- A szóba jöhető élek: `(A,B)`, `(B,C)`, `(C,E)`
- Minimális: **`(A,B)`**

3. kör

- `A = {(A,B), (B,E)}`
- `S = {A,B,E}`, `V\S = {C, D, F}`
- Élek: `(A,C)`, `(B,C)`, `(B,D)`, `(C,E)`, `(D,E)`, `(E,F)`
- Minimális: **`(D,E)`**

4. kör

- `A = {(A,B), (B,E), (D,E)}`
- `S = {A,B,C,D,E}`, `V\S = {F}`
- Élek: `(D,F)`, `(E,F)`
- Minimális: **`(E,F)`**

5. – utolsó – kör

- `A = {(A,B), (B,E), (D,E), (E,F)}`
- `S = {A,B,D,E,F}`, `V\S = {C}`
- Élek: `(A,C)`, `(B,C)`, `(C,E)`
- Minimális: **`(B,C)`**

Ezzel a feszítőfa élei: `{(A,B), (B,C), (B,E), (D,E), (E,F)}`

A példában azt láthattuk, az `A` halmaz folyamatosan szomszédos csúcsokkal „terjeszkedik”. Ez a fajta működés nem törvényszerű, elvileg *szigetszerűen* is létrejöhet a végeredmény (a *Prim-algoritmus* ilyen terjeszkedős lesz, a *Kruskal* pedig inkább szigetes).

1.4. PIROS-KÉK ALGORITMUS [KIEGÉSZÍTŐ ANYAG]

A következőkben ismertetett algoritmus csupán egy másik megfogalmazás, szemléltetés a minimális feszítőfa keresési problémára. Ez semmilyen formában nem lesz visszakérdezve, csak mint érdekesség szerepel itt, illetve talán segít egy más szögből megvilágítani az eddigieket, elmélyíteni a tanultakat.

Vezessük be az élekre a „*piros*” és „*kék*” színezést. Kékek lesznek azok az élek, amik *biztosan* a feszítőfa részei és pirosak, amelyek *biztosan nem*.

Kezdetben minden él *fehér*. Az algoritmus `e` db körből áll, ennek során mindig *egy* élt színezünk be. `e` az élek száma. Ami egyszer be lett színezve valamilyenre, már olyan is marad.

A piros élek meghatározásához használjuk a **piros szabály**t:

- Vegyünk egy *kört*, ami *nem* tartalmaz piros élt (kéket tartalmazhat), ennek a **legnagyobb** költségű élét fessük pirosra

Logikus, hiszen a kör minden éle biztos nem lesz a *feszítőfa* része, és akkor már a legdrágábbat hagyjuk ki (lehet, hogy többet is ki fogunk, ha az másik kör része is), a feladat végső soron a körök eliminálása.

A kék élekhez a **kék szabály**t használjuk:

- Vegyük a csúcsoknak egy `X` *nem üres* részhalmazát, úgy, hogy az `X`-ből ne vezessen ki kék él (nem kell, hogy `X` összefüggő legyen, nem kell, hogy benne kék élek fussanak), a **legkisebb** kivezető élet fessük kékre

Érthető, hiszen ha egy *csúcshalmaz* még nem volt rácsatlakoztatva a rendszerre, érdemes a legolcsóbb él mentén ezt megtenni.

A szabályokat lehet bármilyen sorrendben alkalmazni.

Világos, hogy az előző részben tárgyalt alapalgoritmus gyakorlatilag a *kék szabály*ok ismétlését jelenti.

- Az `A` élhalmaz a *kék él*ek egyre gyarapodó halmaza
- A *vágás* az `X` és a `V\X` mentén jön létre, hiszen nem vezethet `X` és `V\X` között *kék* (`A`-beli) él, azaz a vágásnak *el kell kerülnie* a kék éleket
- Most az `X`-en kívüli *legolcsóbb* élt adjuk hozzá a kék élekhez, azaz `A`-hoz hozzáadjuk az `X` és `V\X` közötti legolcsóbb élt: azaz azt a legolcsóbb élt, ami *keresztezi* a vágást, azaz a *könnyű* élt!

Nézzünk egy lejátszást a fenti példára!

1.

- *Piros* szabály
- `{A,B,C}` egy kör, nincs piros éle, legyen hát a legdrágább, `(A,C)` piros!

2.

- *Kék* szabály
- `{F,B,C}` egy tetszőleges nem üres halmaz, amiből nem vezet ki kék él, a legolcsóbb kivezető él, a `(B,E)` legyen kék

3.

- *Piros* szabály
- `{D,E,F}` egy kör, legyen `(D,F)` piros

4.

- *Kék* szabály
- `{B,E,A}`, legyen `(D,E)` kék

5.

- *Piros* szabály
- `{B, C, E}`, `(C,E)` piros

6.

- *Kék* szabály
- `{A,C}`, pl. `(A,B)` lehet kék

7.

- *Piros* szabály
- `{B,D,E}`, legyen `(B,D)` piros

8.

- *Kék* szabály
- `{C,F}`, legyen `(B,C)` kék

9.

- *Kék* szabály (nincs már kör)
- `{F}`, `(E,F)` legyen kék

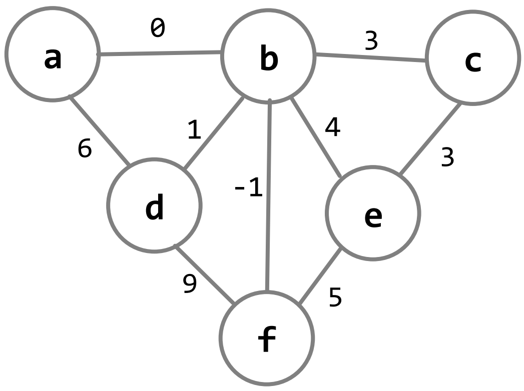
Tehát a kék élek: `{(B,E), (D,E), (A,B), (B,C), (E,F)}`, pontosan mint az előző megoldással.

Megjegyzések: nem kell felváltva alkalmazni a *piros* és *kék* szabályokat; ha elérjük az `n-1` kék élt, akár le is lehet állítani az algoritmust; a körök csúcsszáma persze nem kötelező, hogy mindig 3 legyen.

1.5. KÉRDÉSEK

- Miért nehéz így implementálni ezeket az algoritmusokat?

- Végezd el a fenti algoritmusokat ezen a gráfon:



2. PRIM-ALGORITMUS

Jarník ill. Prim algoritmusa egy könnyen implementálható, *mohó* algoritmus a fenti probléma megoldására. Az alábbiakban ezt ismertetjük.

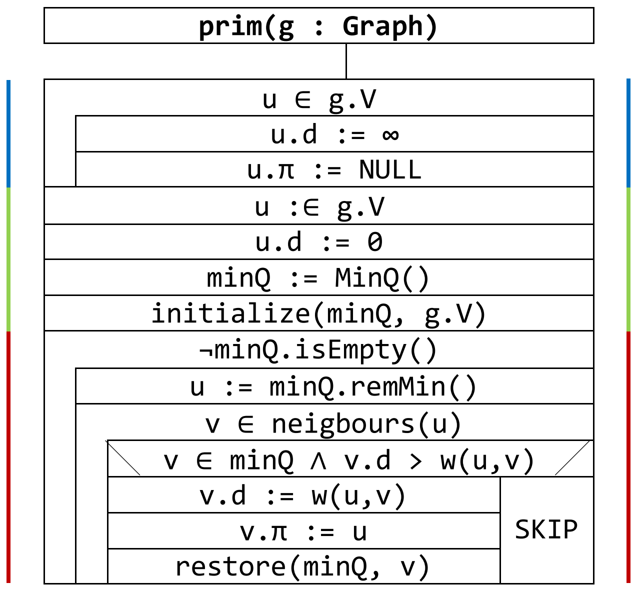
2.1. ALGORITMUS

Véletlenszerűen kijelölünk egy kezdőelemet („`:∈`" – *nemdeterminisztikus értékkiválasztás*), és abból iterálva, *lokális optimum*ok szerint mindig egy-egy új csúcsot húzunk a készülő *fá*ba. Gyakorlatilag a *Dijkstra-algoritmus*sal azonos a módszer, csak itt nem az utat, hanem a „rendszerbe való beépítés" költségét minimalizáljuk, azaz azt, hogy egy adott még nem elérhető csúcsot, milyen **él** felvételével tudjuk a legolcsóbban bevenni. A kezdőcsúcs opcionálisan akár paraméter is lehet.

Ez a *piros-kék algoritmus*ra vonatkozóan csak és kizárólag a *kék* szabályok ismételgetését jelenti, mégpedig olyan `X` halmazzal, ami azokat a csúcsokat tartalmazza, amik nincsenek a `minQ`-ban. Ez a halmaz mindig eggyel bővül, és mindig azt az élt „színezzük kékre", amelyik a frissen `X`-be került, azaz „lezárt" csúcsot az eddigi rendszerre köti. A *Prim-algoritmus*ban `n` kör van, de összesen `n-1` élt húzunk be, hiszen amikor `X` egy elemű (első kör után), azt az egy elemet még nem volt mire rákötni.

A tanult, „hivatalos" elméleti háttéralgoritmusra nézvést pedig azt mondhatjuk, hogy a mindenkori *vágás* a `minQ` és a `V \ minQ` elemei között megy (legalábbis az első kör után, amikor már nem üres az egyik halmaz), azaz a már lezárt csúcshalmaz egyre terjeszkedik. A fentihez hasonlóan, mindig a frissen a `minQ`-ból kikerült elemet a többire rávezető él lesz a *könnyű*, azaz a hozzáveendő.

Fontos tehát érteni, hogy nem akkor történik a „*kék szabály*" vagy a „*könnyű él hozzáadása*" alkalmazás, amikor frissítjük egy csúcs `(d,π)`-párját, hanem amikor „jóváhagyjuk" egy csúcs `(d,π)`-értékeit akkor, amikor kivesszük azt a `minQ`-ból.



A **kék** részben inicializáljuk minden csúcs *szülőj*ét ismeretlenre és *árá*t végtelenre. Csak a *kezdőcsúcs* szülője marad `NULL`, az ára semelyik csúcsnak sem lesz a végére végtelen.

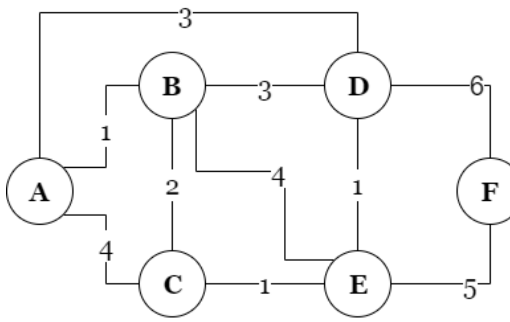
A **zöld** részben kiválasztjuk a *tetszőleges* kezdőcsúcsot, elkönyveljük, hogy ez ingyen a *fa* része, létrehozunk egy *minimum prioritásos sort* és inicializáljuk azt, azaz bepakoljuk a kezdőcsúcsot 0-s és az összes többit végtelenes prioritási értékkel.

A **vörös** rész a ciklus, aminek során a kezdetben `n` elemű prioritásos sorból mindig az aktuálisan legolcsóbb csúcsot kivéve csúcsról csúcsra feldolgozzuk a gráfot. Végignézzük az adott csúcs *kimenő él*eit, és ha úgy tűnik, ebből a csúcsból olcsóbb úton tudunk eljutni egy eddig még nem *véglegesített* (esetleg eleve még nem is látogatott) másik csúcsba, akkor azt frissítjük. Ilyenkor az új prioritások mentén a sort is újra kellhet rendezni.

Implementációjára hasonló megjegyzésekkel élhetünk, mint a *Dijkstra* esetében.

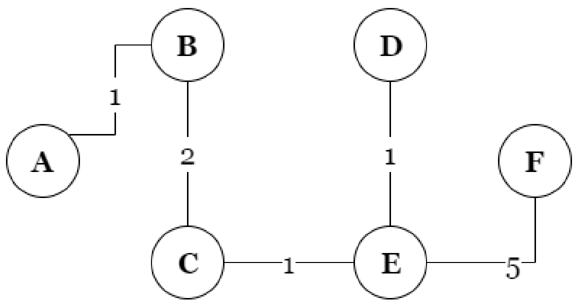
2.2. PÉLDA

Futtassuk le a *Prim-algoritmus*t az alábbi gráfon, minden körben jelöljük `d`-t, π-t és `minQ`-t.



Kör	minQ	(d,π)					
		A	B	C	D	E	F
INIT	<**(A,0)**,(B,∞),(C,∞),(D,∞),(E,∞),(F,∞)>	(0,∅)	(∞,∅)	(∞,∅)	(∞,∅)	(∞,∅)	(∞,∅)
A	<**(B,1)**,(D,3),(C,4),(E,∞),(F,∞)>	KÉSZ	(1,A)	(4,A)	(3,A)		
B	<**(C,2)**,(D,3),(E,4),(F,∞)>		KÉSZ	(2,B)		(4,B)	
C	<**(E,1)**,(D,3),(F,∞)>			KÉSZ		(1,C)	
E	<**(D,1)**,(F,5)>				(1,E)	KÉSZ	(5,E)
D	<**(F,5)**>				KÉSZ		
F	<>						KÉSZ

Íme az *MST*:



2.3. FELADATOK

- Egy `n` csúcsú gráfban egy adott csúcs `d` értéke minimum és maximum hányszor kaphat új értéket a Prim-algoritmus futása során?
- `d` és π közül valamelyik elhagyható-e?
- Beszéljük meg, milyen implementációnál mit és milyen lépésszámmal csinál a `restore()` függvény
- Játszd le a Prim-algoritmust az előző fejezet feladatában megadott gráfra

3. KRUSKAL-ALGORITMUS

A fenti probléma egy másik lehetséges megközelítéssel vett megoldása. A *Prim* esetében egy *csúcshalmaz*hoz vettük hozzá mindig a legalkalmasabb újabb csúcsot, most pedig a legolcsóbb *él*ek szerint fogunk haladni.

3.1. ALGORITMUS

Két verziót is mutatok, a második struktogram bizonyos átnevezések mellett gyakorlatilag megfelel az előadásanyagbeli változatnak – ez az optimalizált, helyes verzió.

A struktogramokban látható ∅ most *üres halmaz* jel, nem *nullpointer*.

Piros rész a lenti struktogramban: adott egy gráf (`g`), amihez egy *élhalmazt*, a leendő *MST* éleit építjük, ezt hívtuk az előadásjegyzetben `A`-nak, itt `mstEdges`nek.

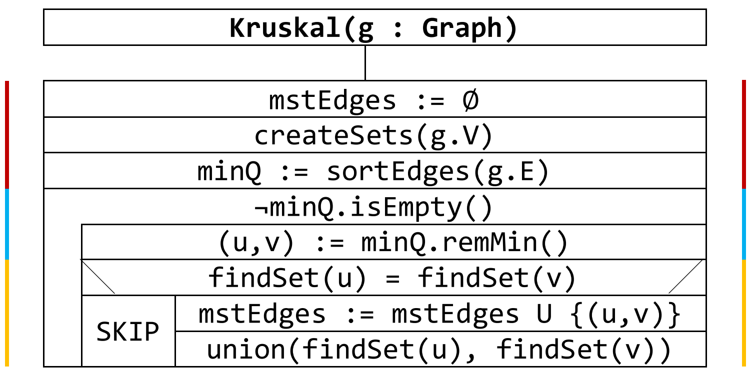
Egyrészt kiindulunk az *üres* élhalmazból, másrészt minden csúcshoz kezdetben egy egyelemű halmazt (*komponenst* rendelünk), így minden csúcs önmagában egy *fa* – ez a `createSets()` függvény, praktikusan minden csúcshoz egy egyre növekvő, egyedi számot rendel pl. egy tömbbel.

Készítünk egy *minimum prioritásos sort* az élek súlyára nézve, majd eszerint vesszük előbb a legolcsóbbat, majd a következő legolcsóbbat, stb., végül a legdrágább *élt*. Ez hasonló a *Prim-algoritmus*ban látottakhoz, tehát ez az algoritmus is *mohó*. Két nagy különbség van: itt élek szerint építjük fel a prioritásos sort, míg ott csúcsok szerint; illetve itt a sorrend az inicializálás után *nem változik*, míg ott időről időre *frissíteni* kellett a prioritásos sort. Ebből következően most a prioritásos sort simán reprezentálhatjuk egy *rendezett tömb*bel is.

Kék rész: egy ciklussal megyünk, amíg van *feldolgozatlan* él. Kivesszük a „következő" legolcsóbb élet.

Sárga rész: megnézzük, ezen legolcsóbb él két *végpontj*ának halmazai azonosak-e. Ha igen, akkor ezt az élet nem vesszük a fába, hiszen olyan két csúcs között menne, amik már eddig is elérhetők voltak egymásból a fa élein át – azaz ez *kört* hozna be a készülő feszítőfába, ami miatt az elveszítené *fa* jellegét.

Ha viszont most találtuk meg két *eddig* egymásból nem elérhető csúcs (*komponens*) közötti legolcsóbb összekötő *élt*, akkor ezt bátran a *feszítőfá*hoz adhatjuk. Ekkor az algoritmus számára tudatni kell azt is, hogy mostantól ez a két csúcs egy komponensbe került – persze az összes egy komponensbeli egyéb csúcsokkal együtt – erről szól a `union()` függvény.



`g` csúcsai és az `mstEdges` együtt minden pillanatban egy *feszítőerdőt* határoznak meg, ez egy *invariáns* az algoritmusra. Kezdetben ez csupa egycsúcsú fa. Az összevonások nyomán a végére eljutunk az egyetlen *komponens*hez, egyetlen *fá*hoz, ami innen nyilvánvalóan *feszítőfa* is. A minimálissága pedig az alábbiakból következik:

Az *absztrakt algoritmus*nak ez úgy felel meg, hogy amikor egy olyan adott `(u,v)` él kerül sorra, amit ténylegesen hozzá is adom az élhalmazhoz, egy olyan *vágást* tekintek, aminek az egyik halmazában `u` és komponense szerepelnek, a másikban a többi csúcs. Világos, hogy nem lehet olyan `mstEdges`beli él, ami keresztezné ezt a vágást, mert akkor `u` komponense nagyobb lenne. Eközött a két komponens között fut az `(u,v)` él. Az ilyenek közül ez a legolcsóbb, azaz a *könnyű* él.

A *piros-kék algoritmus*nak pedig úgy felelünk meg, hogy ha `(u,v)`-t nem vesszük hozzá, akkor legyen a *kör* `u` *komponense* az `(u,v)` éllel kiegészülve, ebben szükségképpen utóbbi a *legdrágább*, hiszen a többi előbb lett kivéve a *prioritásos* sorból; ha az élt hozzávesszük, akkor pedig `u` komponense legyen az a csúcshalmaz, amiből nem vezet ki *kék* él – azaz a *feszítőfa* éle, hiszen ha kivezetne, nagyobb lenne a *komponens*. Most ha épp `(u,v)` van soron, akkor biztosan ez az él a legolcsóbb még színtelen, tehát *kék*re kell színezni, avagy a *fá*ba kell építeni.

Tehát az algoritmus valóban a *minimális költségű feszítőfát* adja így meg, de mennyire *hatékony* vajon?

3.2. Optimalizálás

Piros rész: az *üres halmaz* létrehozása *konstans*. Minden csúcshoz egy *egyedi* szám rendelése a csúcsok számával korrelál, tehát `Θ(n)`-es. Az élek sorbarendezése `Θ(e × log(e))`.

Mivel csak *összefüggő* gráfra van értelme a feladatnak, `e` legalább `n-1`, tehát az kimondható, hogy `n ∈ O(e)`. A fordított irány, s ezzel a `Θ` már nem *feltétlen* lesz igaz (mert lehet *sűrű* a gráf).

De az biztos, hogy emiatt az egész **piros** rész nagyságrendileg `Θ(e × log(e))`.

A **kék** rész *ciklus*ánál nyilván maga a *feltétel* kiszámítása *konstans*, a `remMin()` is ebben az esetben tud konstans lenni, viszont ha ezzel minden körben egy élt veszünk ki a sorból, a *ciklusmag* annak eredeti elemszámaszor, azaz `e`-szer fog lefutni.

Ezen belül (**sárga**) az `mstSet`hez *uniózás* triviálisan *konstans*. A `union()` is lehet *konstans* (ez nem evidens, később indoklom). A `findSet()` pedig `O(log(n))`-es (ezt is később indoklom).

Tehát ott tartunk, hogy a ciklus `O(e × log(n))`, összességében a `Θ(e × log(e))` egy jó nagyságrend.

Tudunk-e ezen hatékonyítani?

Egyrészt, vezessünk be egy *segédváltozót*, ezt én `components`nek fogom hívni, az előadásjegyzetben `k`. Ezzel számoljuk, hány darab *komponens*, hány *részfa* van épp a végső feszítőfa aktuális *részgráf*jában. Kezdetben ez nyilván `n`, és minden olyan körben csökken, amikor összevonunk két fát. Nyilván akkor vagyunk kész, ha `n-1`-szer csökkentettük, azaz, ha az értéke `1` lesz. Összességében így, `n`-ről indulva és csökkentve hatékonyabban használható ez a segédváltozó, mintha egy `0`-ról induló, növő, az *élek számát* számoló változót tartanánk karban.

Előrébb nagyságrendben nem vagyunk, mert lehet olyan gráf, ahol ezek után is ez egy `e`-s ciklus.

A másik javítás a `findSet()` kirakása változóba, hogy csak egyszer fusson le, hiszen ez nem *konstans*.

A harmadik javítás pedig a `minQ` esetében mégis a *kupac* használata. Ott ugyanis bár nem *konstans* a `remMin()`, hanem `O(log(e))`, de az inicializálás is csak `Θ(e)`.

Tehát így ott tartunk, hogy a **piros** rész `Θ(1) + Θ(n) + Θ(e) + Θ(1) = Θ(n + e)`.

A **kék** rész `O(e)`-s szorzót hoz be.

A belseje `O(log(e)) + O(log(n)) + O(log(n)) + 1 + 1 + 1 + 1 = O(log(n)) + O(log(e))`. Az utolsó 4 db konstans a **sárga** rész.

Igen ám, de amúgy `O(log(n)) = O(log(e))`. Ha feltesszük, hogy nincsenek párhuzamos élek, az élek száma legfeljebb négyzete lehet nagyságrendileg a csúcsok számának. Ez pedig *logaritmus* alatt csak *konstans* szorzó.

Ez alapján összességében: `Θ(n + e) + O(e × log(n))` a műveletigény. Ebből a második tag a *domináns*.

Ez pedig a javított algoritmus:

```
Kruskal(g : Graph)
mstEdges := ∅
createSets(g.V)
minQ := sortEdges(g.E)
components := |g.V|
¬minQ.isEmpty() ∧ components > 1
    (u,v) := minQ.remMin()
    uSet := findSet(u)
    vSet := findSet(v)
    uSet = vSet
      SKIP | mstEdges := mstEdges U {(u,v)}
           | union(uSet, vSet)
           | components := components - 1
```

3.3. UNIÓ-HOLVAN (*DISJOINT-SET*) ADATSZERKEZET

Az adatszerkezetre angolul a *union-find* ill. a *merge-find* kifejezéseket is használják.

Feltettük, hogy a `findSet()` és a `union()` is igen *hatékony* műveletek.

Ez fontos kérdés, mivel a `union()` esetében nem csak `u` és `v` komponensét kell egyesíteni, hanem az összes olyan csúcsét is, ami a két elemmel azonos halmazban volt! Ez látszólag egy `n`-es ciklus lenne (ráadásul egy `e`-s ciklusban).

A *komponenshalmaz*okat reprezentáljuk az „*unió-holvan*" adatszerkezettel, ami egy speciális *fa*.

Minden csúcshoz két plusz adatot tárolunk (ez lehet két a csúcsokkal indexelt *tömb*, vagy a csúcsok típusába felvett plusz *adattag* is akár). `π` és `s` ez a két adattag. `π` jelentse a *szülőt* (*parent*), ami nem is teljesen korrekt megfogalmazás, ugyanis ez vagy a szülőt, vagy a fa *gyökerét* fogja jelenteni. `s` pedig az adott csúcsból, mint *gyökérből* kiinduló fa *mérete* (*csúcsszáma*) elcache-elve. Mind `s`, mind `π` szempontjából a gyökércsúcsok értékei érdekelnek csak minket, a többi csúcsra vetített értéknek abban van szerepe, hogy a gyökérben minél optimálisabban tudjuk karban tartani a két attribútum értékét.

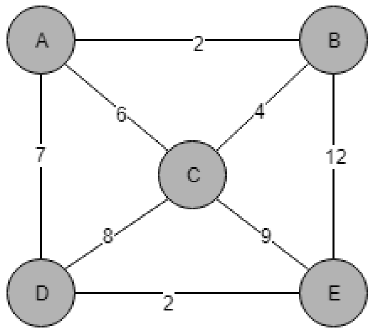
A `createSets()` függvényben tehát minden csúcsnak a `π`-jét önmagára, az `s`-ét `1`-re állítjuk. Ez megfelel a fenti *kontraktus*nak, és ahogy feltételeztük, a *csúcsok* számára véve *lineáris* műveletigényű.

A `findSet()`-ben megnézzük, a csúcs `π`-je önmaga-e. Ha igen, akkor ez egy *fa* gyökere, `return`ölünk vele. Ha nem, akkor *rekurzív*an megnézzük a `π`-jére (*parent*jére) ugyanezt egészen addig, amíg *gyökérre* nem bukkanunk. Mivel cél az, hogy ez a *keresés* minél gyorsabb legyen, a jövőben is, ezért igyekszünk ezt optimalizálni. A keresés során tehát rekurzív módon *famagasság*nyit lépünk felfelé a `π` értékek mentén. Plusz műveletigény nélkül el tudjuk érni, hogy az út során *látogatott* csúcsok `π`-je mind átállítódjon a közös *szülőre*/*gyökérre* (az `s`-ek nem változnak). Ez azt jelenti, hogy a fa gyökerének *gyerekszáma* nő, de a fa *magassága* csökken(het). A gyerekszám növekedése pedig mindegy, hiszen ebben az irányban sose nézelődünk, nem is tartjuk számon a *pointer*eket. Mindegyik `π` a szülő/gyökér felé mutat, más pointer pedig nincs. Ez a művelet tehát alapvetően *logaritmikus*, de ugyanilyen szereposztásban újra meghívva már *konstans* lenne.

A `union()` művelet megnézi melyik fa a *nagyobb* (*elemszám*ra értve, de akár *magasság*ra is nézhetjük – bizonyos jegyzetekben ez így szerepel), és ez alá beszúrja a kisebbiket. Azt tudjuk, hogy a két paraméter két *különböző* fa gyökere (hiszen a `findSet()` ezt kereste, és meg se hívtuk volna a `union()`-t, ha azonos fák lennének érintve). Egyszerűen csak a π mentén beszúrjuk az egyiket a másik alá, valamint az új, *közös* gyökér esetén az `s`-t is értelemszerűen, az *invariáns*t megtartva frissítjük.

A `findSet()`-nél megadott optimalizációt (ti. a *fa* kilapítását) a `union()`-ba is beépíthetnénk. Igaz ehhez kéne tárolni *gyerek* pointereket is és ez nem is lenne *konstans*. Ugyanakkor, így az összes fa 0 vagy 1 magas lenne, ami a másik műveletnél jobb. Mindenesetre a tanult verzióban a `union()` műveletigénye konstans.

3.4. PÉLDA



Kezdetben tehát minden csúcs egy halmaz (*komponens*): {A}, {B}, {C}, {D}, {E}.

- Kiválasztom a legolcsóbb élt: `(A,B)`, mivel `A` és `B` külön halmaz, ezért ezt az élt felveszem és `A`-t és `B`-t összevonom: {A, B}, {C}, {D}, {E}
- Most a második legolcsóbb: `(D,E)`, ezek is külön vannak, összevonom hát őket: {A, B}, {C}, {D, E}
- Most a `(B,C)` él jön, ez is két külön halmazt köt össze, ezért felveszem és összevonom az érintett halmazokat: {A, B, C}, {D, E}
- Az `(A,C)` él a következő, de `A` és `C` halmaza azonos, ezért ez nem lesz a *legolcsóbb feszítőfa* része
- Az `(A,D)` él véglegesíti a feszítőfát: {A, B, C, D, E}

Az elsőként közölt algoritmus esetén még ebben a sorrendben a `(C,D)`, a `(C,E)` és a `(B,E)` éleket is vizsgálnánk, de persze már nem lesz változás. A *komponensszámolós optimalizáció* pedig megóv minket a felesleges köröktől.

Végül a *feszítőfa* élei: `(A,B)`, `(D,E)`, `(B,C)`, `(A,D)`. Ez 5 csúcsra 4 él, rendben van.

3.5. FELADATOK

- Jegyzetből: „*Hogyan tudná kifinomultabban értelmezni azt az esetet, ha a Kruskal-algoritmus* `k` > `1` *(itt:* `components` > `1`*) értékkel tér vissza? Mit jelent a* `k` *értéke általában? Tudna-e értelmet tulajdonítani a Kruskal-algoritmus által kiszámolt* `A` *(itt:* `mstEdges`*) élhalmaznak, ha végül* `k` > `1` *értéket ad vissza?*"

- Miért hatékonyabb így , hogy `components` `n`-ről indul és csökken, mintha `0`-tól indulna és nőne, az *élek számá*t nézné? (kis különbségről van csak szó)

- Milyen gráfban lesz az optimalizáció után továbbra is `e`-s ciklus a `minQ`-n való végigjárás?

- „*Ez azt jelenti, hogy a fa gyökerének gyerekszáma nő, de a fa magassága csökken(het).*" – miért a *feltételes mód*?

- Miért jó/jobb, ha a *nagyobb* fa *alá* szúrjuk a *kisebb*bet?

- Feltételezve, hogy a *Prim-algoritmus* műveletigénye *Fibonacci-kupac*cal `O(e + n × log(n))`, a *Kruskal*é pedig `O(e × log(n))`, melyik a jobb választás és miért?

- B-szakirányra: miért nem jó választás *terminálófüggvény*nek `components`? Mi működhet helyette?

- Futtasd le a *Kruskal-algoritmus*t az első fejezet feladatában megadott gráfra, jelöld minden körben a *komponens*eket!

- Futtasd le a *Kruskal-algoritmus*t a második fejezet példájában megadott gráfra, jelöld minden körben a *komponens*eket!

TARTALOM