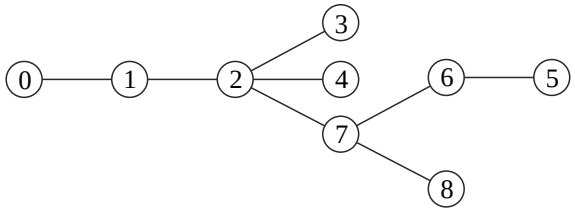
S. 66.
Adrián Patrik, 12. évf.
Baross Gábor Középiskola, Szakiskola és Kollégium, Debrecen

Egy gráfelméleti feladattal van dolgunk. A megoldás során a szokásos módon a házakat *csúcsoknak* vagy *pontoknak* fogjuk nevezni; a csúcsokat összekötő vezetékeket *éleknek*, a két villámcsapás által okozott elektromos jelenséget, melyet a két végpontja és az egyes házakra jutó töltése jellemez, *útvonalnak* hívjuk. Fontos észrevétel, hogy a bemenő irányítatlan gráf, amelyben bármely két csúcsot pontosan egy út köt össze, fagráf.

A gráf ábrázolása

Az első megoldandó probléma a gráf ábrázolása. A csúcsok száma nagy, a szomszédsági mátrixos ábrázolás kizárt. A szomszédsági listák ésszerű alternatívát jelenthetnek, de azok is csak akkor, ha láncolt listaként implementáljuk őket.

A villámok útvonalainak ábrázolását úgy oldjuk meg, hogy az útvonalakat külön sorszámozva tároljuk, majd a gráf minden csúcsán egy láncolt listába gyűjtjük azokat az indexeket, amelyekhez tartozó útvonalak az adott ponton végződnek.



1. ábra. A példa bemenet gráfja

A naiv algoritmus

Az egyik – elég egyszerű – alternatíva a következő: olvassuk be egy villámcsapás adatait, majd az egyik végpontjától a másikig futtassunk egy mélységi keresést, ami meghatározza a két pont közötti útvonalat; ennek mentén minden csúcsnál növeljük meg az összegyűjtött energia mennyiségét a villámcsapás során szerzettel. Sajnos $O(NQ)$ futásidővel esélytelen.

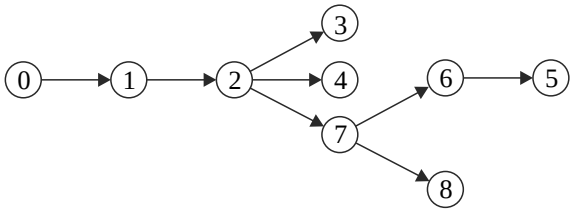
Egy jobb megoldás

Egy újabb megoldási ötlet a következő: a beolvasott villámok útvonalait ne dolgozzuk fel azonnal, hanem tároljuk el őket. Minden lépésben töröljük a gráf egy levelét addig, amíg a gráfnak van csúcsa, de minden törlés előtt hajtsuk végre a következőt: Minden olyan útvonalra, amelyet a törlendő csúcsnál feljegyeztünk (mert az egy végpontja), adjuk hozzá a csúcs össztöltéséhez az útvonal töltését, és ha az útvonal két végpontja nem azonos, akkor azt a végpontját, ami a törlendő csúcs, módosítsuk a csúcs egyetlen szomszédjára és jegyezzük fel, hogy az a most módosított útvonal egy végpontja; egyébként állítsuk az útvonal töltését nullára. (Az utolsó, egyébként-eset nem szükséges, ha a láncolt listák helyett halmazt használunk.) Ennek a megoldásnak az előnye az előzővel szemben, hogy a gráfbejárás nem vizsgál meg feleslegesen csúcsokat a másik végpont keresése közben, de ettől még a komplexitása $O(NQ)$ marad.

Az algoritmus lassúsága elsősorban abból adódott mindkét megoldásban, hogy az egyes házakon összegyűlt töltéseket minden egyes házra újból és újból kiszámoltuk, nem használtuk fel, hogy elegendő a *változásoknál* frissíteni az adatokat. Harmadik megoldásunkat ennek fényében tervezzük meg.

Egy még jobb algoritmus

Az előző két algoritmus fő hibája, hogy a gráfban nem tudtunk *irányokat* kijelölni, egy csomópontban (kettőnél nagyobb fokszámú csúcsban) vagy minden irányban próbálkoztunk vagy észrevettük, hogy meg is állhatunk a feldolgozással, hiszen a fagráfok tulajdonságaiból következően mindig van legalább egy levelük, ahol a feldolgozás iránya egyértelmű. Jó esély van arra, hogy ha a gráfban sikerülne irányokat megadni, akkor a megoldás is gyorsabbá válhatna.



2. ábra. A példa bemenet 0 gyökerű feszítőfája

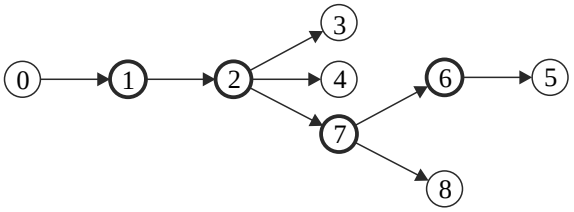
Válasszunk ki önkényesen egy csúcsot a gráfban, legyen ő a fa gyökere. (Tetszőleges csúcsot választhatunk.) Indítsunk el egy mélységi keresést ebből a csúcsból kiindulva, ami a szokásos módon sorra veszi a fa csúcsit, ezáltal egy irányított mélységi feszítőfát építve.

Osztályozzuk az útvonalakat a végpontjainak ebben a keresőfában elfoglalt helye alapján:

1. A két csúcs azonos.
2. Az egyik csúcs a másik leszármazottja a feszítőfában.
3. A két csúcs a feszítőfa két különböző ágán van. Ekkor van legalább egy közös ősük.

Ennek alapján beszélhetünk első, második és harmadik típusú útvonalakról.

Az első eset legfeljebb kellemetlenség lehet, mert egy feltétellel könnyen tesztelhető. Az utolsó kettővel kell inkább foglalkoznunk.

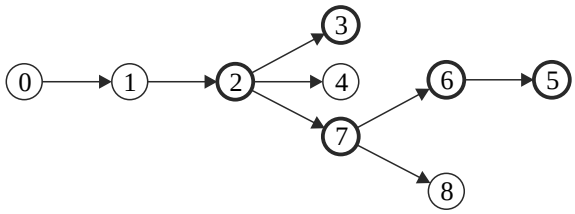


3. ábra. A példa bemenet nem elágazó 1–6 útvonala

Tegyük fel, hogy a fában a p és q csúcsok között 2. típusú útvonal van, mégpedig úgy, hogy p q-nak őse ($p \rightsquigarrow q$). Indítsunk egy mélységi bejárást a fa gyökeréből és színezzük a csúcsokat a szokásos módon fehérre, szürkére vagy

feketére. Az útvonal feldolgozása csak q-ból p-be és nem fordítva történhet, mert p első meglátogatásakor (amikor p szürke lesz) még nem tudjuk, hogy a fában melyik útvonalon kell tovább haladni ahhoz, hogy q-ba jussunk. (Egyáltalán ekkor még az sem biztos, hogy q leszármazottja.) Amikor q-hoz érünk és szürkére színezzük, akkor még mindig nem célszerű az útvonallal foglalkozni, mert q után annak gyermekeit fogjuk bejárni, amit a $p \rightsquigarrow q$ villám nem érint.

Amikor azonban q-t feketére színezzük, az algoritmus már p felé visszalép. A $p \rightsquigarrow q$ útvonal ismert, ezt a mélységi bejáráshoz használt verem tartalmazza. A visszalépési folyamatban a csúcsok mindaddig megkapják a $p \rightsquigarrow q$ útvonal töltését, amíg az algoritmus p-n túl nem lép. Ezért p-ig visszalépve a közbenső csúcsok mind megkapják a gyermekeikből, mint gyökerekből induló részfákból származó töltést. (Itt konkrétan egy útvonalat vettünk figyelembe, de természetesen egy adott csúcson keresztül több is haladhat: ilyenkor azok szuperpozíciójával kell számolni, az egyes mennyiségek egymástól függetlenül kumulatívan összegezhetők.) Azt, hogy p ősei is megkapják a töltést úgy akadályozhatjuk meg, hogy p szülőjének az útvonal töltésével megegyező nagyságú negatív töltést adunk. Ha ehhez hozzáadjuk a gyermekétől – p-től – továbbadott töltést, azok összege 0 lesz, vagyis p szülőjén nem jelenik meg a $p \rightsquigarrow q$ útvonal miatt többlettöltés; mivel a bejárás ezt az összeget lépteti vissza a csúcs felé, ezért ott sem lesz probléma.



4. ábra. A példa bemenet elágazó 3–5 útvonala

Nehezebb boldogulni a harmadik típusú útvonalakkal, vagyis amikor a két végpont nincs egymással leszármazotti viszonyban. Ha p és q egy ilyen útvonal végpontjaiként a gráf valamely csúcsai, akkor létezik olyan u csúcs, hogy $u \rightsquigarrow p$ és $u \rightsquigarrow q$. Vegyük észre, hogy sem p, sem q nem lehet a fa gyökere, mert abból minden csúcsba vezet út, mi pedig feltettük, hogy a két végpont egymásból kölcsönösen elérhetetlen. u ezért mindenképpen létezik, legalább egy, de lehet, hogy több csúcs is megfelel a definíciójának. Legyen v a gráf azon csúcsa, amelyre $v \rightsquigarrow p$ és $v \rightsquigarrow q$ létezik, de ugyanez már egyik gyermekére sem teljesül. v a lehetséges u-k közül a gyökértől legtávolabbi, amit a p és q csúcsok legközelebbi közös ősének (lowest common ancestor, LCA) nevezünk és $LCA(p,q)$ val jelölünk.

A szigorú matematikai bizonyításokat mellőzve azt állítjuk, hogy a $p \rightsquigarrow q$ út felírható a $p \rightsquigarrow LCA(p,q)$ és a $LCA(p,q) \rightsquigarrow q$ utak ebben a sorrendben vett konkatenációjaként, vagyis ahhoz, hogy p-ből q-ba jussunk, át kell haladnunk a legkisebb közös ősükön. (Szemléletesen: ahhoz hogy a fa egyik ágából a másikba jussunk át kell menni azon a ponton, ahol az ágak találkoznak.)

Ezt felhasználva az útvonalak két olyan útra bonthatók, melyeknek egyetlen közös pontjuk van, ez pedig a legközelebbi közös ősük. Ezek viszont olyan utak, melyekre teljesül, hogy az egyik végpontjuk a másik leszármazottja (konkrétan az eredeti végpontok a közös ős leszármazottai), tehát második típusú útvonalakként kezelhetők. Egy valamire kell vigyáznunk: a legközelebbi közös ős lesz az a pont, ahol a két összegzés „összeér”, ezért itt az útvonal töltésének kétszeresét kapjuk, vagyis ezt az értéket egyszer le kell belőle vonni.

A legközelebbi közös ős keresésének feladata sem triviális, mert ha a fa fordított, a csúcsokból a gyökér felé mutató éleit a naiv módszerrel végigkövetjük, egy ilyen művelet $O(N)$ idejű lesz. Mivel ezt minden útvonalra meg kell tennünk, ezért a teljes komplexitás itt is $O(NQ)$-nak adódik, ami semmivel sem jobb az előző megoldásnál.

Adrián Patrik, 12. évf.
Baross Gábor Középiskola, Szakiskola és Kollégium, Debrecen

Egy majdnem optimális megoldás

Az LCA keresését az RMQ-ra (range minimum query) visszavezetve két csúcs LCA-ja egy $O(N)$ idejű előfeldolgozás után $O(1)$ időben megtalálható. Ez azonban az ágyúval verébre tipikus esete lenne, mert az futásidőre adott korlátok nem annyira szorosak, hogy ehhez az – egyébként nem egyszerű – algoritmushoz kelljen folyamodnunk. Tarjan megoldása a mélységi bejárást és a diszjunkthalmaz-erdőket használja az LCA meghatározására. Ennek egy módosított változatát használjuk a feladat megoldására. Az eredeti eljárás megtalálható az *Új algoritmusok*[1] c. könyvben.

A Tarjan-féle LCA algoritmust a következőképpen módosítjuk/egészítjük ki:

```
PROCESS-TREE(i)
 1  i.color ← GRAY
 2  i.ancestor ← i
 3  foreach j in i.neighbours
 4      if j.color = GRAY
 5          continue
 6      j.parent ← i
 7      i.charge ← i.charge + PROCESS-TREE(j)
 8      j.ancestor ← i
 9  i.color ← BLACK
10  r ← i.charge
11  foreach p in i.paths
12      j ← OTHER-ENDPOINT(p, i)
13      if j.color = BLACK
14          k ← FIND-ANCESTOR(j)
15          if k = i
16              if i = j
17                  i.charge ← i.charge + p.charge
18                  continue
19              r ← r − p.charge
20          else
21              r ← r + p.charge
22              i.charge ← i.charge + p.charge
23              k.charge ← k.charge − p.charge
24              if k.parent ≠ NULL
25                  k.parent.charge ← k.parent.charge − p.charge
26      else
27          r ← r + p.charge
28          i.charge ← i.charge + p.charge
29  return r
```

Az első és egyetlen „manuális” függvényhívást a mélységi feszítőfa gyökerére kell végrehajtanunk, amit ebben a feladatban tetszőlegesen választhatunk.

Értelmezzük az algoritmust! Amint a mélységi bejárás eléri a csúcsot, szürkére színezi és saját magát felelteti meg a legközelebbi ősének. Az első **foreach** ciklus végignézi a csúcs összes szomszédját, és ha az már bejárt (ezen a ponton csak a szülője lehet bejárt), akkor továbblép a következő szomszédra; egyébként beállítja saját

[1] T. Cormen, C. Leiserson, R. Rivest, C. Stein: *Új algoritmusok*. Scholar kiadó, Budapest, 2003, 454. oldal

magát a gyermeke szülőjeként a feszítőfában és továbblép a mélységi keresésben. A PROCESS-TREE visszatérési értéke a paraméteréül kapott csúcsból induló részfa befejezetlen útvonalainak töltésösszege, amit hozzá kell adnunk a szülőjének töltéséhez (ha nem, arról a második **foreach** gondoskodik). Pontosabban megfogalmazva PROCESS-TREE(p) azon útvonalak töltéseinek összegét adja, amelyeknek pontosan egy végpontja szerepel a p gyökerű feszítő-részfában vagy egyik végpontjuk p.

Mikor a rekurzió visszalépett, a gyermekből induló részfával végzett az algoritmus (minden olyan végpont-párt, amelynek a legközelebbi közös ősei a részfában voltak, feldolgozott), a fennmaradó, még nem feldolgozott útvonalaknak legalább a szülőn (i) át kell haladniuk, tehát j ősének i-t állítjuk be. Ez a lépés felel meg a diszjunkthalmaz-erdő egyesítés műveletének, hiszen beolvasztjuk az i legkisebb közös ősű csúcsok közé a részfa minden csúcsát, amit eddig a gyermek, j reprezentált.

A csúcsot ezen a ponton feldolgozottnak tekintjük, és színét feketére állítjuk, hiszen többé már nem jövünk ide vissza. A visszatérési értéknek, a szülő felé továbbított töltésnek először a csúcs saját töltését állítjuk be, majd ezt szükség szerint a továbbiakban módosítjuk.

A második **foreach** ciklus feladata, hogy feldolgozza az összes olyan útvonalat, amelynek az éppen vizsgált csúcs valamely végpontja; az éppen feldolgozott útvonalat p-ben tároljuk, a j változóba az útvonal másik végpontja kerül. Itt több esettel kell számolnunk.

1. A másik végpont színe fekete, vagyis ez a másodikként feldolgozott végpont. Ekkor megkeressük k-t, a másik végpont ős-reprezentánsát a FIND-ANCESTOR függvénnyel. Ennek értékétől függően újabb eseteket kapunk:

   (a) Ha $k = i$, az megint kétféleképpen lehet. Vagy úgy, hogy j ős-reprezentánsa az algoritmus 2. sorában került beállításra, és ekkor $j = i$, vagy úgy, hogy ez a 8. sorban történt meg; ekkor j i-nek valamely gyermeke. Az előbbi esetben (első típusú útvonal) i töltését megnöveljük az útvonal töltésével, és készen vagyunk, nézhetjük a következő útvonalat. Utóbbi esetben pedig, ha j gyermeke i-nek, akkor ez egy második típusú útvonal és i a felső végpontja. Ez azt jelenti, hogy az ebből az útvonalból származó töltést nem szabad felfelé továbbadni (mert ott már nem tart az út), de i-n még érvényesíteni kell.

   (b) Egyébként k már fel van dolgozva, de i-nek nem gyermeke. Szülője sem lehet, mert egy csúcs minden gyermekét hamarabb dolgozzuk fel, mint önmagát, ezért j i valamely ősének leszármazottja és k a legközelebbi közös ősük. Ezt indirekten bizonyíthatjuk. k azon gyermekének részfáját, ami j-t tartalmazza, már k alá rendeltük (gyökere ős-reprezentánsának beállítottuk k-t), vagyis k-nak azon gyermeke, ami j-t tartalmazza, i-t nem, különben már mindkét csúcs feldolgozott lenne még azelőtt, hogy a hozzácsatolás megtörtént volna. (A mélységi bejárás előbb dolgozza fel egy csúcs leszármazottait.) Ez ellentmondás, mert i-t még csak most dolgozzuk fel.

   Az újonnan talált útvonal töltését még nem „örököltük meg” egyik gyermektől sem, tehát ezt mind az aktuális csúcs össztöltéséhez, mind a befejezetlen utak össztöltéséhez hozzá kell adni. Ugyanezt az útvonal másik végénél is megtettük, ezért k-nál ezt a töltést kétszer számoljuk, így egyszer csökkenteni kell k össztöltését az útvonal töltésével, majd k szülőjéét még egyszer, hogy az összegében mind a két ágról származó töltést semlegesítse. Ez alól egyetlen kivétel létezik: k a teljes feszítőfa gyökere és nincs szülője. Ekkor ugyanis k feketévé válásával a rekurzió véget ér.

2. Minden egyéb esetben az útvonal másik végpontját még nem dolgoztuk fel, tehát i az első végpont. Ekkor csak elindítjuk a fában felfelé a töltést és ennek likvidálását a második csúcs feldolgozásának idejére (a fentiekben leírtakra) hagyjuk.

A FIND-ANCESTOR őskereső függvény megegyezik a diszjunkthalmaz-erdőkre általánosan használt reprezentáns-kereső algoritmussal.

Adrián Patrik, 12. évf.
Baross Gábor Középiskola, Szakiskola és Kollégium, Debrecen

Elemzés

Elemezzük egy kicsit az algoritmus erőforrásigényeit, mielőtt rátérnénk az implementáció részleteire. A memóriafoglalás teljesen statikus, mert így gyorsabb. Az adatok beolvasása $O(N)+O(Q)$ időt vesz igénybe a szomszédsági- és útlisták építésével együtt. Maga a rekurzió az N csúcs mindegyikét egyszer járja be, azokon összesen $O(Q)$ lépést hajt végre, melyek összesen $O(Q\log N)$[2] időt vehetnek igénybe (amortizált komplexitás), ami minden bizonnyal belefér a 3 másodperces időkorlátba. A diszjunkthalmaz-erdőkre jellemző $O(N\alpha(N))$-es futásidőnél (ahol α az inverz Ackermann-függvény, ami minden gyakorlati esetben ≤ 5) azért kaptunk itt rosszabb eredményt, mert a két halmaz uniójánál (8. sor) egyértelműen kijelöljük, hogy melyik fa gyökerét választjuk a halmaz reprezentánsának, ezáltal a rang szerinti unió kiesik, csak az úttömörítés marad, mint gyorsító tényező.

Az LCA meghatározása optimális esetben egy $O(N)$ idejű előfeldolgozás után $O(1)$ időben végezhető, vagyis a feladat megoldható $O(N)+O(Q)$ időben is. (Ennyi idő minden, a feladatot megoldó algoritmusnak kell.) Ehhez képest az általunk adott algoritmus ugyan lassabb, de „nem olyan sokkal”.

A megoldás memóriaigénye is $O(N)+O(Q)$, hiszen a csúcsokat és az útvonalakat nyilvántartó tömbök mérete N és Q, valamint minden élet (számuk $N-1$) és útvonalat két végpontjukon tartunk számon egy láncolt listában. (A tényleges megvalósításban azonban az előzetes statikus foglalás miatt a memóriaigény elég nagy, de nem függ a bemenettől.) Ha maga a feszítőfa egy láncolt listává torzul, akkor a mélységi keresés rekurziómélysége elérheti a csúcsok számát, vagyis a fentieken felül $O(N)$ veremmemóriára is szükség van.

A megvalósítás

A program az adatok beolvasásával kezdődik. Az élek mindegyikéhez a szomszédsági listák két elemét készítjük el, majd ezeket hozzácsatoljuk a végpontok szomszédsági listáihoz. (Mivel a sorrend nem számít, ezért első elemként.) A listaépítésnél nem használjuk a 0 indexű listaelemet, mert ennek a pozícióját a C sajátosságai miatt hasznosabb, ha a lista végének jelzésére használjuk. Hasonlóan járunk el a villámok útvonalainak beolvasásakor. Itt csak arra kell ügyelni, hogy ha a két végpont megegyezik, akkor az érintett csúcshoz csak egyetlen bejegyzés készüljön.

Ezután nekilátunk a gráf feldolgozásának. A feszítőfa gyökerének a 0 indexű csúcsot választjuk. A programkód lényegében megegyezik a fent leírt pszeudokódos algoritmussal, csupán néhány technikai megoldásban tér el. Ilyen például, hogy a függvény nem közvetlenül a csúcsokat, hanem azok indexeit tárolja a változókban, vagy hogy a **foreach** ciklusokat a megfelelő listaiterációkra cseréltük.

Az egyes programelemeket leíró adatszerkezetek (listaelem, csúcs, útvonal) saját `typedef`-es struktúra típust kaptak. A maximális darabszámokat egy `#define`-os konstans értékű makróval könnyen módosíthatóvá tettük. A tömböket globálisan definiáltuk a globális elérhetőség, a 0-inicializáció és az egyszerűbb helyfoglalás miatt.

Fordítás

Mivel a rekurzió legrosszabb esetben N mélységig is lemehet, ezért könnyen lehet, hogy a program kifut az alapértelmezett veremmemóriából. Lényeges, hogy a fordításnál a maximális veremméretnek *legalább* 32 MB-ot adjunk meg!

A program fordítását a Visual C++ fordítóval a `cl s66.c /F 0x2000000` paranccsal végezhetjük.

A forráskód UTF-8 kódolású.

[2] `http://www.eecs.wsu.edu/~ananth/CptS223/Lectures/UnionFind.pdf`