Dinamikus gráfok III

az irányítás visszatér

Iván Szabolcs

2016 ősz

Detour: amortizált időigény

Egy-egy törlés költsége több lehet, mint $\log N^2$ (pl. ha sok k szintű él van a kisebb fán belül).
Erre a ,,hagyományos" legrosszabbeset-analízis (pl. mindig N-nel becsülve a végigjárt élek számát) egy nagyon rossz korlátot adna.

Potenciálmódszer

Az elv: egy művelet költségét (vagy annak egy részét) ,,megelőlegezve", már korábban ,,kifizetjük".

Nézzünk egy-két egyszerűbb példát, aztán visszatérünk a feszítőerdőből törlés amortizált költségére.

ArrayList (Java-ban; C++ban `vector`)

Az ArrayList egy dinamikusan növő tömb adatszerkezet (vektor), támogatja a push(a) és a get(i) (direkt címzés, tömbcímzés) műveleteket.

- Inicializálás: foglalunk egy konstans méretű T tömböt (Java: 32), mérete 0, kapacitása ez a konstans.
- get(i): ha $0 \leq i <$ size, visszaadjuk $T[i]$-t, egyébként AIOOBE.
- push(a): ha size $<$ capacity, akkor T[size++] $= a$.
  Különben capacity-t duplázzuk, foglaljunk egy ekkora méretű új tömböt, másoljuk bele az egész eddigi T-t az elejére és ekkor már beszúrhatjuk mint az előbb.

Időigények?
A retrievalé $O(1)$.

ArrayList

Ha ,,hagyományos'' módon számoljuk, akkor egy `push` időigénye akár $O(n)$ is lehet, ahol n az aktuális mérete az ArrayListnek.
De érezhető, hogy ez nagyon ritkán történik meg.
Ötlet: két duplázás közti költséget ,,terheljünk rá'' a beszúrt elemekre közben egyenletesen, előre kifizetve a duplázást!

Egy nem-duplázó beszúrás költsége legyen 1 helyett 3.

Ez ugyanúgy konstans, és beszúrásonként képez kettő ,,tartalékot''.

Egy n méretű tömb duplázásának a költsége (mondjuk) n.
Mire a duplázásig eljutunk, az előző duplázás óta ,,gyűjtöttünk'' $2\frac{n}{2} = n$ tartalékot, ezt ,,költsük el'' a duplázásra!
Így magának a duplázásnak nincs költsége, kifizettük előlegként. Az ezutáni beszúrásé megint 3, stb.

Ekkor `push`onként $O(1)$ amortizált költségünk van (minden `push`ra hármat írunk, ekkor a teljes műveleti költséget végig felülről becsüljük).

Sor két veremből

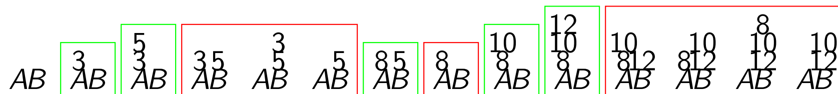
Funkcionális programozási nyelvekben a(z immutable) lista az alap adatszerkezet, ami tkp egy verem, konstant idejű push, pop és isEmpty utasításokkal.

Sor

Viszünk két vermet: A és B.

- enqueue(a): A.push(a).
- dequeue: if B.isEmpty while !A.isEmpty B.push(A.pop); return B.pop

push(3), push(5), pop, push(8), pop, push(10), push(12), pop



Sor két veremből

Itt is: a `push`-nál az adott elemhez előre befizetjük azt a későbbi költséget, mikor majd kivesszük A-ból és betesszük B-be.
(Egy A.`push`t, egy A.`isEmpty`t, egy A.popot és egy B.`push`t fizetünk ki, az utóbbi három műveletet ,,előre".)
A `push` költsége továbbra is amortizált $O(1)$.
Most már a popé is (csak a B.popot kell fizessük akkor)

ACM feladatban...

Implementáljunk olyan sor adatszerkezetet, amiben az enqueue, dequeue és min műveletek időigénye (amortizált) $O(1)$!

Az előző szerint

Elég, ha olyan vermet tudunk csinálni, aminek a minimumát is konstans időben le tudjuk kérdezni.

Viszünk két vermet, egyet a minimumoknak

Mivel $\min\{\texttt{a} :: \texttt{A}\} = \min(a, \min\{\texttt{A}\})$ (azaz: a minimum kiszámítható a verem tetejéből és a verem maradék részén a minimum értékéből), ezért: vigyünk egy B vermet a minimumoknak is

`push(a)`: A.push(a); B.push(min(a, B.top))

`pop`: A.pop; B.pop

`min`: B.top

Konstans időigény. Működik minden olyan f aggregált függvényre, melyre $f(a, S) = g(a, f(S))$ valamilyen g kiszámítható függvényre. (pl. maximum, összeg, elemszám – így az átlagot is le lehet kérdezni pl)

Recap: probléma

A megoldandó problémák alakja általában:

- kiindulunk egy G gráfból, erre kapjuk utasítások egy (hosszú) sorozatát
- egy utasítás lehet insert(u, v), erase(u, v) vagy query(u, v)
- az insert beszúrja, az erase törli az (u, v) élt, ők az update utasítások
- most: a query(u, v) azt kell megválaszolja, hogy u-ból v elérhető-e
- most: a gráfok irányítatlanok (még pár másodpercig)
- a csúcsok száma n, az éleké m, a parancsoké q
- ha a ,,szokásos" módon felépítjük a gráfot és mindig ,,from scratch" újraszámoljuk a queryt, akkor az időigény updatenként $O(1)$, querynként $O(n+m)$
- ha minden update-nél kiszámítjuk és eltároljuk a tranzitív lezártat, akkor updatenként $O(n \cdot m)$, querynként $O(1)$
- be akarjuk lőni a kettő közé

Törlés feszítő erdő-hierarchiából

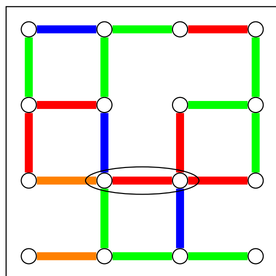
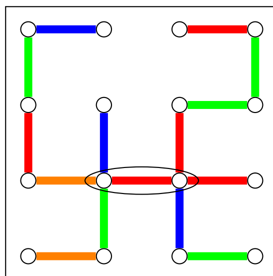
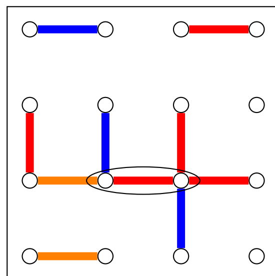
- A G_i gráfok éllistáit is tároljuk. (Így be tudjuk járni hatékonyan az egy csúcsra illeszkedő, adott szintű éleket.)
- insert(i,j): beszúrjuk az (i,j) élt, $\ell(i,j) := \log N$.
  Ha i és j az $F_{\log N}$-nek különböző fáiban vannak, akkor egybe mergeljük a két fát és behúzzuk $F_{\log N}$-be az (i,j) élt. Ez $O(\log N)$ idő.
- query(i,j): a két csúcs akkor van egy komponensben, ha $F_{\log N}$-nek egy fájában vannak. Ez $O(\log n)$ idő.
- delete(i,j):
  - Kivesszük az (i,j) élt (G szomszédsági listáiból és a G_i-k szomszédsági mátrixaiból).
  - Ha (i,j) nincs benne $F_{\log N}$-ben, nincs gond, megyünk tovább.
  - Ha igen, akkor benne van az összes F_k-ben, amire $k \geq \ell(i,j)$. Töröljük ki belőlük.
  - Ezek után keresünk egy élt, amivel össze tudjuk kötni az i és j fáját.

Csere-él keresése

- Mivel minden F_k egy minimális feszítőerdő, így $\ell(i,j)$-nél kisebb szintű éllel nem tudjuk összekötni (i,j) helyett a széteső fákat.
- Tehát a $k = \ell(i,j), \ldots, \log N$ szintű élek közt keresünk összekötő élt, ilyen sorrendben.
- Legyen T_i és T_j az i-t és j-t tartalmazó fa F_k-ban úgy, hogy $|T_i| \leq |T_j|$.
- Mivel eredetileg az invariáns miatt $|T_i| + |T_j| \leq 2^k$, így $|T_i| \leq 2^{k-1}$.
- Járjuk be az összes olyan (x,y) élt, amire $x \in T_i$ és $\ell(x,y) = k$.
- Ha $y \in T_j$, akkor adjuk hozzá az (x,y) élt a $T_k, \ldots, T_{\log N}$ fákhoz és végeztünk.
- Különben csökkentsük $\ell(x,y)$-t eggyel.

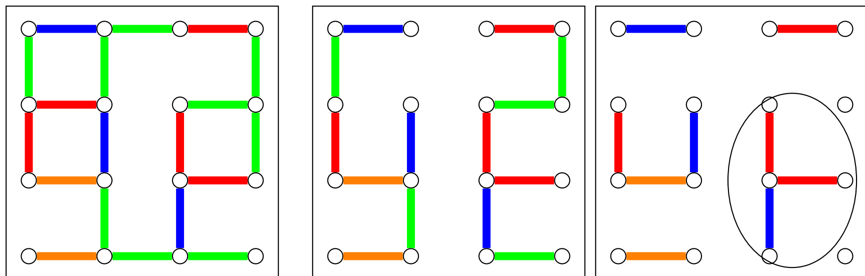
Csere-él példa

$n = 16$ csúcs, az élek szintjei egy, kettő, három vagy négy.
A kék komponensek mérete legfeljebb 2, a kék-narancs komponenseké legfeljebb 4, a kék-narancs-piros komponenseké legfeljebb 8, stb.

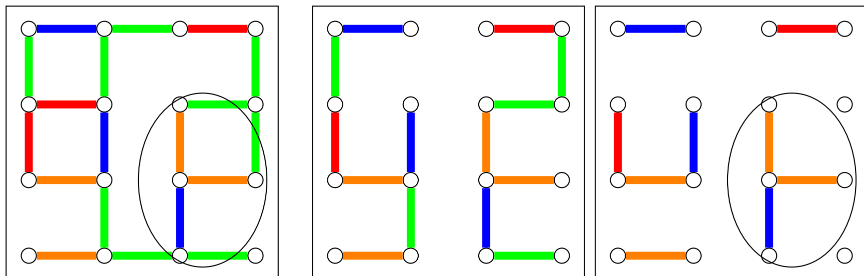
Csere-él példa

$n = 16$ csúcs, az élek szintjei egy, kettő, három vagy négy.
A kék komponensek mérete legfeljebb 2, a kék-narancs komponenseké legfeljebb 4, a kék-narancs-piros komponenseké legfeljebb 8, stb.



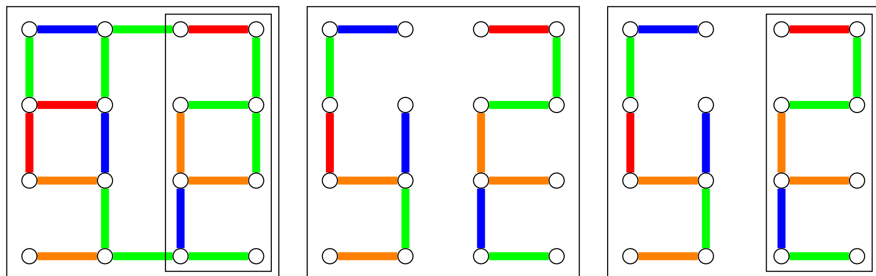
Csere-él példa

$n = 16$ csúcs, az élek szintjei egy, kettő, három vagy négy.
A kék komponensek mérete legfeljebb 2, a kék-narancs komponenseké legfeljebb 4, a kék-narancs-piros komponenseké legfeljebb 8, stb.



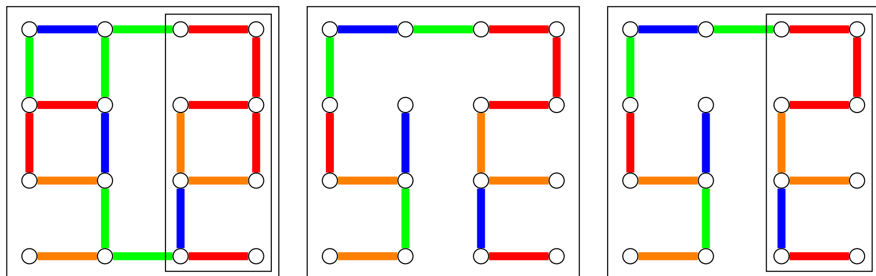
Csere-él példa

$n = 16$ csúcs, az élek szintjei egy, kettő, három vagy négy.
A kék komponensek mérete legfeljebb 2, a kék-narancs komponenseké legfeljebb 4, a kék-narancs-piros komponenseké legfeljebb 8, stb.



Csere-él példa

$n = 16$ csúcs, az élek szintjei egy, kettő, három vagy négy.
A kék komponensek mérete legfeljebb 2, a kék-narancs komponenseké legfeljebb 4, a kék-narancs-piros komponenseké legfeljebb 8, stb.



Vissza az erdőkből törlésre

Tehát: a (kisebb) T_1 fában végignézzük az összes k szintű élt.
Ha az él a fán belül megy: csökkentjük a szintjét. (Ez élenként $\log N$ idő ET forestben).
Egyébként: behúzzuk az ET forest két fája közé és végeztünk a törléssel.
Hagyományosan számolva: egy a törlés költsége a k. szinten lehet $O(e \cdot \log N)$, ahol e a fában scannelt élek száma. (Ez soknak látszik, nem jó felső korlát.)

Ehelyett:

Az él szintjének csökkenésével járó scanneléseket előre kifizetjük az él beszúrásakor!

Az e élt $\log N$ szintűként szúrjuk be $\Rightarrow$ kifizetünk a $\log N$ költségű beszúráson felül pluszban még $\log^2 N$ költséget. Így minden beszúrt él $\log^2 N$,,tartalékkal" indul.

Vissza az erdőkből törlésre

Törléskor: a scannelt élek közül aminek csökkentjük a szintjét, azt törléskor nem számoljuk, hanem az él által a beszúráskor felhalmozott tartalékból vonjuk le.
Így szint csökkenéskor ebből a tartalékból vonunk le $\log N$-t.
Az eredeti $\log N$ szintet 1-ig tudjuk csak csökkenteni, annál tovább nem fogjuk, ezért a kezdeti tartalék fedezi az ilyen költségeket végig.
Ha nem csökken a szint, azt a törléshez számoljuk. Abból viszont csak egy lesz legfeljebb törlésenként, így a beszúrás amortizált költsége $O(\log^2 N)$, a törlésé pedig $O(\log N)$, összesen az update amortizált költsége $(\log^2 N)$.

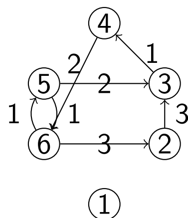
Irányított gráfok

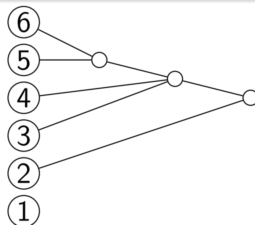
- Ha az input gráf irányított, többé nem elég feszítő erdőt tárolni.
- Első lépésben az erősen összefüggő komponenseket fogjuk tárolni, dinamikusan.
- insert(u, v) esetén komponensek összeolvadhatnak.

Inkrementális SCC

Ötlet

- Minden (u, v) élhez tároljuk azt a t időpontot, amikor (ahanyadik lépésben) hozzáadtuk a gráfhoz.
- A G_i gráf: a $\leq t$ időpontban hozzáadott élek alkotta gráf.
- A query(u, v) lekérdezés vissza fogja adni azt a t-t, amióta u-ból elérhető v (és mondjuk ∞-t, ha nem érhető el).
- Ezúttal az insert és a delete utasítások egy lépésben élek egy halmazát is hozzáadhatják / elvehetik egyszerre.





Inkrementális SCC

Komponens-erdő

- Nyilvántartunk egy komponenserdő struktúrát (ld előző ábra)
- Minden létrejövő C SCC-re felírjuk azt a $\mathrm{T}[C]$ időpontot, amikor létrejött
- Minden összeolvadó SCC-re felírjuk azt a $\mathrm{parent}(C)$ komponenst, akibe összeolvadt
- Inicializálás: minden komponens egyelemű, parent üres, létrehozási időpontjuk 0

$\mathrm{query}(u, v)$ annak a legkisebb időpontú SCC-nek az időpontja kell legyen, amelyikbe már u és v is beletartozik

Inkrementális SCC

Számok az éleken

- Minden (u, v) élhez nyilvántartjuk, hogy mikor hoztuk létre: $\text{time}(u, v)$
- Továbbá azt is, hogy mikor került u és v ugyanabba a komponensbe: $\text{query}(u, v)$.
- A kettő maximuma: $\text{time}'(u, v)$.
- G_i: azok az (u, v) élek, akiknek $\text{time}(u, v) \leq i$.
- F_i: azok az (u, v) élek, akiknek $\text{time}'(u, v) = i$.

Akkor

- F_i és F_j diszjunkt, ha $i \neq j$ (mert más a time' érték).
- $F_i \subseteq G_i$ (mert $\text{time}(u, v) \leq \text{time}'(u, v)$).
- F_i éleit G_i belső éleinek, a többit külső éleinek hívjuk.

Belső élek, külső élek

Tehát G_i belső élei a G_i komponensein belül mennek, a külső élei pedig különböző komponenseket kötnek össze.

Union-Find strikes again

Az aktuális SCC-inket egy Union-Find adatszerkezetben is tároljuk.

Inicializálás

- $t := 0$
- $F_1 := \emptyset$
- minden v csúcsra legyen parent(v) := null, t$[v] := 0$
- UnionFind inicializálása

Beszúrás

insert(E)

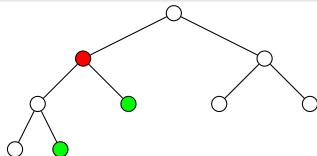
- $t++$ (telik az idő)
- $F_t := F_t \cup E$ (az eddigi külső élek és E élei együtt adják F_t-t)
- Számítsuk ki az $F' = \{(\text{find}(u), \text{find}(v)) : (u, v) \in F_t\}$ élhalmazt (a komponensek reprezentánsai közt menjenek az élek)
- Számítsuk ki ennek a gráfnak az SCC-it.
- Ha van több csúcsú SCC-je ennek a gráfnak, akkor ennek hozzunk létre egy új SCC csúcsot a komponensfában, kössük alá az őt alkotó komponenseket és mergeljük össze őket az Union-Find struktúránkban.
- Az új F_{t+1} legyen a mostani F_t-ből a (Union-Find szerinti) különböző komponensek közt menő élek halmaza.
- Időigény: $O(m\alpha(n))$.

Mikor kerültek egy komponensbe?

Ha a lekérdezés az, hogy az u és v csúcsok hanyadik lépésben kerültek egy SCC-be, akkor egy lowest common ancestor problémával állunk szemben.

Lowest Common Ancestor

- Input: egy T (irányított, gyökeres) fa.
- Lekérdezések: (u, v) csúcspárok, adjuk vissza azt a w-t, aki őse u-nak is, v-nek is, de w-nél lejjebb már nincs ilyen csúcs.



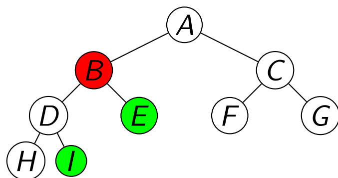
LCA

Preprocess time, query time

Preprocess nélkül egy LCA queryt $O(h)$ időben tudunk megválaszolni, ahol h a mélyebben levő csúcs mélysége.

Preprocess

- Futtassunk a fában egy DFS-t és ahányszor érintünk egy csúcsot (alulról vagy felülről), írjuk ki a szintjét.
- Minden csúcshoz jegyezzük fel a (fentről) elérési és elhagyási idejét.



012323212101212 10

$A : [0, 17]$

$B : [1, 9]$

$E : [8, 8]$

$I : [5, 5]$

LCA

Ha a két intervallum tartalmazza egymást, akkor a közös ős: a csúcs a nagyobb intervallummal.

Ellenkező esetben diszjunktak és a közös ős a két intervallum közt érintett legkisebb szintű csúcs lesz.

Range Minimum Query, RMQ

- Input: egy $A[1,\ldots,n]$ tömb.
- Queryk: $[i,j]$ intervallumok.
- Output: az intervallum egy legkisebb eleme.

RMQ

Preprocess

- Minden i-re és 2^j-re, amire $i + 2^j \leq n$, számítsuk ki előre az $A[i \ldots i + 2^j - 1]$ tömb minimumértékét.
- Alapeset: $A[i, i]$ minimuma $A[i]$.
- Indukció: $A[i, i + 2^j - 1]$ minimuma $A[i, i + 2^{j-1} - 1]$ és $A[i + 2^{j-1}, i + 2^j - 1]$ minimuma, az $O(1)$ idő.
- Ilyen intervallumból van $O(n \log n)$.

Query

Az $[i \ldots j]$ intervallum minumum eleme:

- vegyük a legnagyobb 2^k kettő-hatványt, ami belefér az intervallumba;
- a válasz $A[i, i + 2^k - 1]$ és $A[j + 1 - 2^k, j]$ minimuma lesz – konstans idő.

Tudunk jobbat?