Self-adjusting hálózatok

Fraknói Ádám
Témavezetők: Vass Balázs, Rétvári Gábor

2022. december 16.

1. Bevezetés

A hálózati rendszereknél fontos, hogy úgy alakítsuk ki őket, hogy rugalmasak legyenek és könnyen újra konfigurálhatóak. Gyorsan változhatnak a feltételek, amihez alkalmazkodnia kell a rendszernek. Ugyanakkor ez kompromisszummal is jár: a gyakoribb módosítások a hálózaton javítják a teljesítményt, azonban magasabb újra konfigurálási költségekkel is járnak.

Még ma sem ismerjük elég jól az önbeálló (self-adjusting) hálózatok algoritmikus problémáit. Mivel a következő bemeneti adatról nincs információnk, így nem tudjuk előre, hogy milyen műveleteket kell végrehajtani a gráfon legközelebb, ezért az online algoritmusoknak jól kell reagálniuk a változásokra. Ideális esetben azt tudjuk mondani, hogy egy online algoritmus c-versenyképes, azaz tetszőleges bemenet esetén az algoritmus által adott megoldás költsége nem nagyobb, mint egy optimális offline algoritmus költségének c-szerese.

A self-adjusting hálózatokra nagyon hasonlítanak a self-adjusting adatstruktúrák, mint például a self-adjusting listák és splay fák [3]. A lista hozzáférési problémát (list update problem) Sleator és Tarjan mutatták be az 1980-as években [2], melyről a 2. fejezetben számolok be. Ez arról szól, hogy olyan online algoritmust keresünk, amely egy listából kell lekérdeznie elemeket, miközben az éppen kiválasztott elemet költség nélkül a listában előrébb viheti. Közismert algoritmus erre a *move-to-front* (MTF) algoritmus, amely azt csinálja, hogy a legutoljára lekérdezett elemet mindig a lista elejére teszi. Belátható, hogy ez egy 2-versenyképes algoritmus.

Az [1] cikk általánosítja ezt a problémát, míg a a lista hozzáférési problémánál semmilyen költség nem volt, itt már tetszőleges lista átrendezésre bevezet egy költségfüggvényt. Az itt lévő fogalmakat és eredményeket a 3. fejezetben mutatom be.

Hogy mik is azok splay fák és miért jók, azt a 4. fejezetben mutatom be.

2. Lista hozzáférési probléma

A lista hozzáférési problémában adott egy láncolt lista, és a következő műveleteket hajthatjuk végre rajta:

- Keres(x): Az x elem megkeresése
- Beszúr(x): Az x elem beszúrása.
- Töröl(x): Az x elem törlése.

Folyamatosan beérkeznek hozzánk ezeknek a kéréseknek egy sorozata, amit nekünk teljesítenünk kell. Jelölje n a lista maximális elemszámát valaha, és m a kérések számát.

Feltesszük, hogy az i. elem megkereséséhez vagy törléséhez i darab műveletre van szükségünk, ugyanis a mutatónk kezdetben az első elemen szerepel, és az elejétől kezdve végig kell néznünk, hogy egyenlő-e az adott érték x-szel. A beszúráshoz $n+1$ művelet szükséges, ugyanis végig kell mennünk az egész listán megnézni, hogy valóban nincs-e benne az adott elem, majd utána beszúrni. A lista karbantartására az alábbi műveleteket engedjük meg:

- A megkeresett x elemet tetszőleges hellyel *ingyen* előrébb lehet mozgatni a listában.
- Bármely két szomszédos elemet felcserélhetünk *1 költséggel*.

Jelenleg azzal az esettel foglalkozunk, amikor csak a *keres* műveletünk van. Ilyenkor van egy $x_1, \ldots, x_n$ bemenetünk, majd sorban a $\sigma = (\sigma_1, \ldots, \sigma_m)$ kérések sorozata, ahol $\sigma_i \in \{x_1, \ldots, x_n\}$.

Több online algoritmust is alaposan megvizsgáltak erre a problémára, ebből hármat is bemutatok:

Move-to-front (MTF) algoritmus: Egy elem keresése (vagy beszúrása) során az elemet a lista legelejére visszük.

Transpose (T): Egy elem keresése (vagy beszúrása) során az elemet cseréljük ki az előtte lévővel.

Frequence count (FC): Minden elemre tartsuk számon, hogy hányszor volt eddig megkeresve. A lista ennek az értékében monoton csökkenjen.

Mindegyik algoritmusra igaz, hogy csak olyan karbantartást alkalmaz, ami ingyenes. Azonban a három algoritmus közül csak az MTF algoritmus lesz megfelelő.

2.1. Állítás. *Sem a T, sem az FC algoritmus nem versenyképes.*

2.2. Tétel. *Az MTF algoritmus 2-versenyképes.*

A tételt például potenciálfüggvényekkel lehetséges belátni. A tétel fő ötlete az, hogy minden x, y elempárra tekintsük az ő sorrendjüket és hogy mikor cserélnek sorrendet.

A Move-to-front "optimalitását" a lista hozzáférési problémában az alábbi tétel mutatja:

2.3. Tétel. *Ha egy online algoritmus c-versenyképes, akkor $c \geq 2$.*

3. Self-adjusting lineáris hálózatok

Az előző fejezetben a kurzor mindig visszaugrott a lista elejére és így kellett kiválasztani a következő elemeket. Most azonban feltesszük egyrészről, hogy a kérések egy $\sigma_i = (s_i, t_i)$ párt tartalmaznak, ahol a kurzor kezdetben az s_i elemen van és a t_i elemet keressük. Így ennek a kérésnek a költsége $|h(s_i) - h(t_i)|$, ha a $h : V \to \mathbb{N}$ függvény az elemek helyét definiálja. Most csak azt a karbantartási műveletet engedjük meg, amikor két szomszédos elemet felcserélhetünk 1 költséggel.

Ez a probléma már jóval nehezebb, ami az alábbi példán is észlelhető: Vegyünk egy $V = \{c, v_1, \ldots, v_n\}$ gráfot, majd legyen $\sigma_i := (c, v_i)$. Ekkor egy offline algoritmus ezt könnyedén meg tudja meg tudja oldani, ugyanis a c csúcsot folyamatosan körbe mozgatja. Azonban egy online algoritmusnál ezt is figyelembe kell venni, hogy egy ilyen központi csúcsot mozgatni kell.

Rossz hír, hogy az eredeti lista hozzáférési problémával ellentétben ebben a modellben nem létezik konstans közelítő algoritmus, ugyanis megadható egy $\Omega(\log n)$-es alsóbecslés a versenyképességi hányadosra egy determinisztikus online algoritmus esetén.

3.1. Tétel. *Jelölje $cost(ON(\sigma))$ (és $cost(OFF(\sigma))$) egy online (és egy offline) algoritmus költségét a σ kéréssorozaton. Ekkor $|\sigma| = \Omega(n^2)$ esetén a versenyképességi hányadosra $\max_\sigma \frac{cost(ON(\sigma))}{cost(OFF(\sigma))}$ igaz az, hogy legalább $\Omega(\log n)$.*

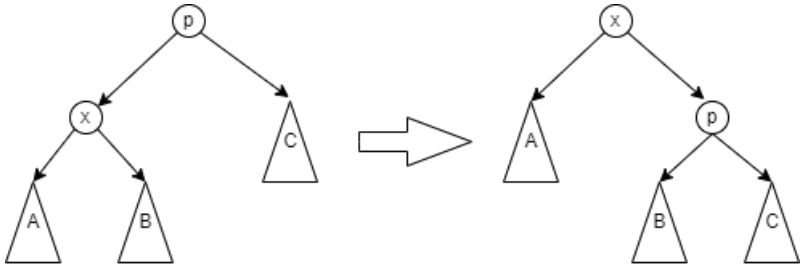
Szerencsére megadható egy felső becslés is: létezik egy olyan algoritmus, amely $\mathcal{O}(\log(n))$-versenyképes, ha a kéréssorozat megfelelően hosszú.

4. Splay fák

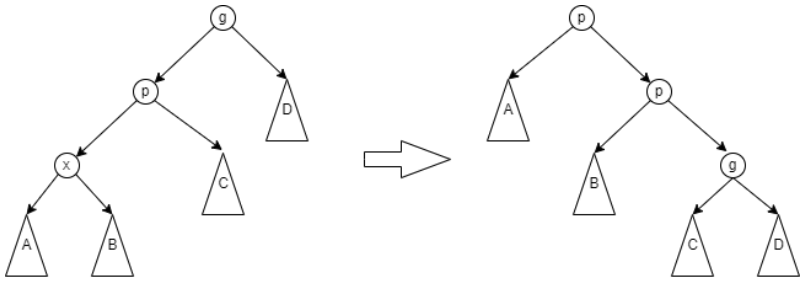
A splay fa (splay tree) egy self-adjusting bináris kereső fa, azaz a keresett elemek mindig a gyökérbe mozgatja, így a gyakran keresett elemek a bináris fa tetejéhez közel fognak elhelyezkedni, így gyorsan elérhetőek lesznek a későbbiekben.

A megkeresett x elem gyökérbe való felvitelét úgynevezett splay lépések sorozatával tesszük meg. Jelöljük x szülőjét p-vel, és p szülőjét g-vel, ha létezik. A splay lépések a következők:

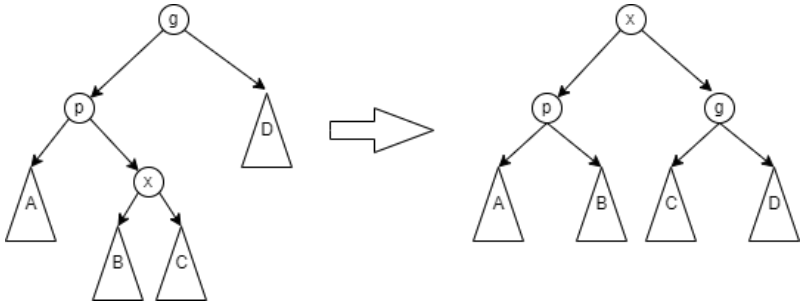
Cikk lépés: tegyük fel, hogy p nem a gyökér. Ekkor forgassuk meg a fát az xp él mentén.



Cikk-cikk lépés: tegyük fel, p nem a gyökér, továbbá x és p ugyanolyan oldalú gyerekek. Ekkor forgassuk meg a fát pg, majd xp él mentén.



Cikk-cakk lépés: tegyük fel, hogy p nem a gyökér, továbbá x és p különböző oldalú gyerekek. Ekkor forgassuk meg a fát px, majd xg élek mentén.



Egy csúcs splay-elése azt jelenti, hogy a csúcsot ilyen splay lépések sorozatával a gyökérig felvisszük. A splay fában a műveletek (keres(x), beszúr(x), töröl(x)), ugyanúgy néznek ki, mint a bináris kereső fában, annyi különbséggel, hogy a művelet végén most splay-eljük a csúcsot.

A karbantartás költsége legyen az, hogy hány forgatást végeztünk el a splayelés során. Ezek alapján a hatékonysága egy splay fának a következő lesz:

4.1. Tétel. *Bármilyen m műveletből álló sorozatnak a splay fán a futási ideje $\mathcal{O}(m \log n)$, ahol n a csúcsok maximális száma valaha a fában.*

A bizonyítás kulcsa, hogy definiálunk egy ϕ potenciál függvényt. Jelölje $size(x)$ az x csúcs alatt lévő csúcsok számát x-et beleértve, majd $rank(x) := \log_2(size(x))$. Ekkor legyen $\phi := \sum_x rank(x)$, azaz a potenciálfüggvény az összes csúcs rangja. A bizonyítás egy fontos eszköze még az alábbi lemma:

4.2. Tétel. *A karbantartási ideje az x csúcs splayelésének az r gyökér esetén legfeljebb $3(rank(r) - rank(x)) + 1$.*

Mivel egy csúcs rangja legfeljebb $\log n$, ezért a karbantartási ideje egy műveletnek legfeljebb $\mathcal{O}(\log n)$ lesz.

Hivatkozások

[1] Chen Avin, Ingo van Duijn, and Stefan Schmid. Self-adjusting linear networks. In Mohsen Ghaffari, Mikhail Nesterenko, Sébastien Tixeuil, Sara Tucci, and Yukiko Yamauchi, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 368–382, Cham, 2019. Springer International Publishing.

[2] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, feb 1985.

[3] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, jul 1985.