NYILATKOZAT

Név: Birszki Levente

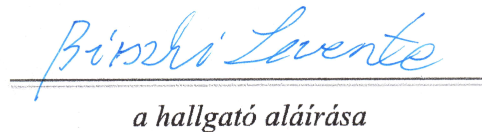
ELTE Természettudományi Kar, szak: Matematika BSc

NEPTUN azonosító: U3J6YK

Szakdolgozat címe:
Adatstruktúrák összehasonlító elemzése

A **szakdolgozat** szerzőjeként fegyelmi felelősségem tudatában kijelentem, hogy a dolgozatom önálló szellemi alkotásom, abban a hivatkozások és idézések standard szabályait következetesen alkalmaztam, mások által írt részeket a megfelelő idézés nélkül nem használtam fel.

Budapest, 2023.06.01.



a hallgató aláírása

Adatstruktúrák összehasonlító elemzése

— SZAKDOLGOZAT —

Készítette:

Birszki Levente

Matematika BSc

Alkalmazott matematikus szakirány

Témavezetők:

Király Zoltán
Egyetemi docens
Számítógéptudományi Tanszék

Schwarcz Tamás Bence
Doktorandusz
Operációkutatási Tanszék

Eötvös Loránd Tudományegyetem
Természettudományi Kar
Budapest, 2023.

Tartalomjegyzék

Bevezetés **1**

1. Kupacok **4**

1.1. Bináris kupac 5
1.2. d-edfokú kupac 7
1.3. Vödrös kupac 8
1.4. r-kupac 9
1.4.1. Az r-kupac javításai 11
1.5. Fibonacci-kupac 14
1.6. Párosítós kupac 16

2. Tesztelés C++ nyelven **18**

2.1. Véletlenszerűen generált gráfok 19
2.1.1. Kis élsúlyú gráfok 19
2.1.2. Ritka gráfok 21
2.1.3. Középsűrű gráfok 29
2.2. Worst-case gráfok 37
2.2.1. cn élű gráfok 37
2.2.2. Ritka gráfok 37
2.2.3. Középsűrű gráfok 44

3. Összegzés **49**

3.1. Mérési eredmények 49
3.2. Milyen kupacot használjunk 50

Hivatkozások **51**

Bevezetés

A szakdolgozat célja és felépítése

Szakdolgozatomban a prioritásos sor, vagy más néven kupac adatstruktúrák elméletével és gyakorlati megvalósításával foglalkozom. Az 1. fejezetben a kupacok különböző verzióit mutatom be, és műveleteinek lépésszámát elemzem. Az r-kupacnak három különböző javítását is kidolgoztam, melyek szintén ebben a fejezetben kerülnek bemutatásra. A műveletek lépésszáma mellett továbbá Dijkstra [1] és Prim [2] algoritmusának lépésszámát is vizsgálom.

A 2. fejezetben a korábban bemutatott adatstruktúrák implementációival végzek teszteléseket C++ programozási nyelven. A tesztelés során a kupacok Dijkstra-algoritmusbeli alkalmazásának hatékonyságát mérem, különböző gráfokon. Az elméleti lépésszámokat a gyakorlati futásidőkkel összevetve a kupacok viselkedését részletesen vizsgáltam, ezzel nem csak arra kaptunk választ, hogy mely esetekben mely kupac a leggyorsabb, hanem arra is, hogy miért.

Végül a 3. fejezetben összegzem a mérési eredményeket, és általuk egy heurisztikát adok arra, hogy mikor milyen kupacot válasszunk a Dijkstra-algoritmushoz, hogy a lehető legjobb futásidőket érjük el.

Alkalmazások

Mielőtt rátérnék a kupacok elméletének tárgyalására, fontosnak tartom, hogy a gyakorlati alkalmazásokról is ejtsek néhány szót. A fent említett két algoritmuson kívül számos más területen is jól használható ez az adatstruktúra.

- Huffman-kódolás [3]: A kódolás során a fa megalkotásához használhatunk kupacokat. A mindenkori gyökerek lesznek a kupac elemei, a gyökerekhez tartozó relatív gyakoriságok pedig a kulcsok. Ekkor két Mintörléssel megkapjuk a két legkisebb értékű gyökeret, majd az új gyökeret, melynek a korábbi két törölt elem lesz a két gyereke, Beszúrjuk a kupacba.

- DES (Discrete-event simulation) modell: a folyamatosan működő rendszereknek, mint amilyen egy operációs rendszer is, egy modellje a Diszkrét Esemény Szimuláció. Ebben a modellben minden végrehajtandó eseménynek van egy prioritása, és

mindig az éppen legnagyobb prioritású eseményt hajtjuk végre. Ekkor az eseményeket tárolhatjuk egy kupacban, ahol a hozzájuk tartozó prioritások a kulcsok.

- Rendezési algoritmusok [4]: A Kupacok rendezési algoritmusok megvalósítására is alkalmasak. A rendezendő elemeket beszúrjuk a kupacba, majd Mintörléseket hajtunk végre, amíg a kupac ki nem ürül. Ezzel az elemeket kulcsuk szerint növekvő sorrendben kapjuk vissza. Az ilyen típusú rendezést kupacos rendezésnek nevezzük.

- ROAM (Real-time optimally adapting mesh) [5]: Ez a számítógépes grafikában használt háromszögelési algoritmus, ami két kupacon alapul. A működési elve az, hogy minden megjelenített háromszöghöz fenntartunk egy prioritást, ami azt jelzi, hogy mennyire szükséges a jelenleginél részletesebben megjeleníteni az adott felületet. Ebben az esetben a kupac elemei a háromszögek lesznek. Amikor javítani szeretnénk a megjelenített kép minőségén, Maxtörlést hajtunk végre a kupacban, így megkapva, hogy melyik háromszöget célszerű több kisebb háromszögre bontani.

- QoS (Quality of Service): Az internetes forgalomirányítók is prioritással látják el a továbbítandó csomagokat. Ennek célja, hogy néhány szolgáltatás (mint például a VoIP, vagy IPTV), ahol fontos a minél alacsonyabb késleltetés, előnyt élvezzen a többivel szemben. A router kupac használatával meghatározhatja, hogy melyik csomagot továbbítsa következőnek, illetve melyik az, amelyik még várhat.

1. fejezet

Kupacok

Ebben a fejezetben a kupac absztrakt adatstruktúrának néhány megvalósításával foglalkozom. Ezekben minden tárolt elemhez tartozik egy kulcs, azaz prioritás is. Egy kupacnak legalább a következő műveleteket kell támogatnia:

- *Beszúrás*: Egy új elem hozzáadása a kupachoz, a hozzá tartozó prioritással.
- *Mintörlés* vagy *Maxtörlés*: A legkisebb vagy legnagyobb prioritású elemnek a törlése.
- *Kulcs-csökkentés* vagy *Kulcs-növelés*: Egy adott elem prioritását változtatja meg.

Megvalósítható kupacműveletek továbbá az *Egyesítés*, amivel kettő vagy több kupacot tudunk egyesíteni, a *Törlés*, amivel egy tetszőleges elem törölhető egy rá mutató pointer segítségével, és a *Minolvasás*, amivel a legkisebb elem kulcsát kaphatjuk meg, anélkül, hogy törölnénk azt. Ezek a gyakorlatban kevésbé fontos műveletek, így elemzésüket nem érintem.

Lehetséges, hogy egy kupacnak művelete a Min- és Maxtörlés is, ezeket hívjuk kétoldalú prioritásos sornak. Dolgozatomnak nem célja ezek bemutatása, mivel a legtöbb alkalmazásban, köztük Dijkstra és Prim algoritmusában elég, ha egy kupac az egyik művelettel rendelkezik. A Kétoldalú prioritásos sorok a gyakorlatban nem is olyan hatékonyak, mint az egyoldalú változataik. Lehetséges továbbá, hogy egy kupac a Kulcs-csökkentés és Kulcs-növelés műveletet egyszerre támogatja. Ez néhány kupac megvalósítás esetén könnyedén kivitelezhető, de az általam alább tárgyalt alkalmazásokban erre sincs szükség.

A kupacok felhasználási módjai közül Dijkstrának illetve Primnek az algoritmusát választottam ki, ezeknek a lépésszámait elemzem. Egy n csúcsú és m élű gráf esetén, ha szomszédsági mátrixban tároljuk, akkor mindkét algoritmus egyaránt $O(n^2)$ lépést követel. Elsősorban ezt szeretnénk javítani úgy, hogy a gráfot éllistában tároljuk, és mellé kupacot használunk. Mindkét algoritmusban legfeljebb n db Beszúrás és Mintörlés, illetve m db Kulcs-csökkentés műveletre van szükség. Néhány kupacnál a műveletek amortizációs idejét vizsgálom. Ezt a módszert először Tarjan használta 1985-ben [6]. Ehhez definiálni kell a potenciálfüggvényt, ami megtalálható az említett cikkben. Ez a fejezet Király Zoltán jegyzete [7] alapján készült.

Ahhoz, hogy lépésszámokról beszélhessünk, szükség van egy számítási modellre. Ez esetünkben a word RAM modell [8], melyben egy szó hossza w, azaz a memóriába legfeljebb w bites számok férnek. Feltételezzük, hogy az alábbi adatok elférnek egy szóban:

- a csúcsok száma, $w \geq \log n$,
- az élek száma, $w \geq \log m$,
- egy tetszőleges út hossza, $w \geq \log((n-1)C)$, ahol C a legnagyobb élköltséget jelenti.

Ekkor a következő műveleteket tudjuk elvégezni a szavakon konstans időben. Aritmetikai műveletek $(+, -, \cdot, /)$, összehasonlítások $(<, \leq, =, \geq, >)$ és bitműveletek (aritmetikai és logikai eltolás, AND, OR, XOR).

1.1. Bináris kupac

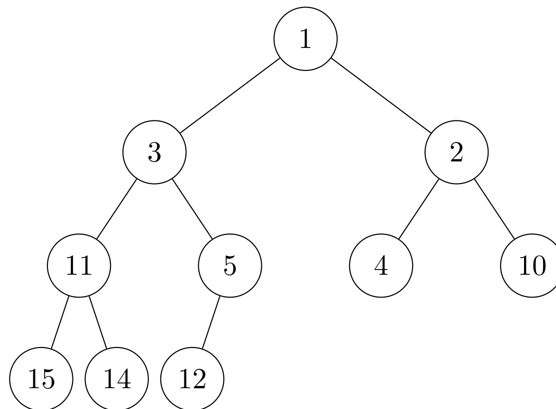
Ezt az adatstruktúrát először J.W.J. Williams mutatta be 1964-ben a kupacrendezési algoritmus részeként [4].

Definíció. Egy fa *kupacrendezett*, ha:

- van egy kitüntetett gyökér;
- a csúcsaiban rekordok vannak, minden rekordnak van egy kitüntetett kulcs mezője;
- a gyökéren kívül minden csúcsra igaz, hogy a benne szereplő rekord kulcsa nem kisebb, mint a szülője kulcsa.

1.1.1. Észrevétel. *A definícióból következik, hogy a minimális kulcsú rekord a gyökérben található. Az is igaz, hogy egy kupacrendezett fának minden gyökeres részfája is kupacrendezett.*

Definíció. Egy gyökeres fa *erősen kiegyensúlyozott*, ha legfeljebb az alsó szint telítetlen, az összes többi szinten maximális számú csúcs van, és az alsó szinten a csúcsok balra zárva helyezkednek el.



1.1. ábra. Bináris kupac

Definíció. A *(bináris) kupac* egy olyan fa, amelyben:

- minden csúcsnak legfeljebb 2 gyereke van,
- kupacrendezett,
- erősen kiegyensúlyozott.

A fenti két definícióból következik, hogy egy n csúcsot tartalmazó kupac mélysége $\lfloor \log n \rfloor$.

A kupac megvalósításához szükség lesz két tömbre, legyenek ezek A és B, és egy C szótárra. Az A tömbben tároljuk a kulcsokat, melynek első eleme a gyökér, azaz $A(1)$ a gyökér kulcsát tartalmazza. Az i. csúcs bal gyermekének kulcsa $A(2i)$-ben, míg jobb gyermekének kulcsa $A(2i+1)$-ben található. Így a j. csúcs szülője a $\lfloor j/2 \rfloor$. Ezen felül használunk egy VÉGE pointert is, mely a kupac aktuális elemszámát adja meg.

A B tömbben az i. csúcshoz tartozó rekord azonosítóját tartjuk, a C szótárban pedig a rekord azonosítójához az A és B tömbbeli helye (indexe) tartozik.

A továbbiakban egy kupac alatt a kulcsokat tartalmazó tömböt értjük. A műveleteket erre adjuk meg, a másik tömb kezelése ehhez hasonló. A szótárat is értelemszerűen frissítenünk kell, amikor egy elem tömbbeli helye megváltozik. A kupacnak a következő műveletei vannak:

- *Újkupac*(A, n): Egy új kupacot hoz létre, azaz egy n méretű tömböt. Itt n az egy időben a kupacban szereplő elemek maximális száma. Lépésszáma $O(1)$.
- *Beszúrás*$(A, elem)$: Egy új elemet szúr be a kupacba. Az új elemet a tömb következő üres helyére szúrja be, ezáltal az erősen kiegyensúlyozottság megmarad, de a kupacrendezettség elromolhat. Ennek korrigálására szükség van egy *Felbillegtet(A, i)* belső műveletre, ami az i. elemet addig „úsztatja felfelé” kicserélve a szülőjével, amíg a kupacrendezettség helyre nem áll. Mivel a fa mélysége $\lfloor \log n \rfloor$, így ennyi összehasonlításra illetve cserére lehet szükség. Tehát ennek a műveletnek a lépésszáma $O(\log n)$.

- *Mintörlés*(A): Kitörli a minimális kulcsú elemet a kupacból, és visszaadja a nevét és kulcsát. Ezt úgy teszi, hogy a gyökérben szereplő elemet kicseréli a tömb utolsó nem üres elemével, majd a VÉGE pointert csökkenti. Ezzel az erősen kiegyensúlyozottság most sem romolhat el, ellenben a kupacrendezettség igen. Itt egy *Lebillegtet*(A, i) belső műveletre van szükség, amit meghívunk a gyökérre, és az i. csúcsban szereplő rekordot most lefelé úsztatjuk, mindig a kisebb kulcsú gyerekével kicserélve, egészen addig, amíg a kupacrendezettség helyre nem áll. A Felbillegtetéssel ellentétben itt ki kell választanunk a kettő közül a kisebb gyermeket, így két összehasonlítást is végeznünk kell szintenként. Ez legfeljebb $2 \cdot \log n$ összehasonlítás és $O(\log n)$ lépés.

- *Kulcs-csökkentés*$(A, elem, \Delta)$: A megadott *elem* kulcsát csökkenti $\Delta \geq 0$ értékkel. A kupacrendezettség fenntartásához a kulcs-csökkentett elemet fel kell billegtetni, így ennek a műveletnek a lépésszáma is $O(\log n)$.

Ha ezt használjuk a Dijkstra-, illetve Prim-algoritmus megvalósításához, akkor összesen n db Beszúrásra, n db Mintörlésre, és m darab Kulcs-csökkkentésre lesz szükség. Így a bináris kupaccal a teljes futási idő mindkét algoritmus esetén $O(m \cdot \log n)$.

1.2. d-edfokú kupac

A bináris kupacnak közvetlen általánosítása a d-edfokú kupac, melyet Johnson írt le először [9].

Definíció. Egy gyökeres fa *d-edfokú kupac*, ha:

- minden csúcsának legfeljebb d gyereke van,
- kupacrendezett,
- erősen kiegyensúlyozott.

A kulcsokat itt is egy tömbben tároljuk a bináris kupachoz hasonlóan, ám a tömb indexelését most az egyszerűség kedvéért 0-tól kezdjük. Tehát a gyökér a 0. csúcs. Az i. csúcs gyermekei a $d \cdot i + 1, d \cdot i + 2, \ldots, d \cdot i + d$, az i. csúcs szülője pedig az $\lfloor (i-1)/d \rfloor$ indexű csúcs. A műveletek lépésszámai a következőképpen alakulnak:

A Beszúrás illetve Kulcs-csökkentés lépésszáma a Felbillegtetés belső művelettől függött. Mivel a fa mélysége $\lfloor \log_d n \rfloor$, így ezeknek a műveleteknek a lépésszáma $O(\log_d n)$.

A Mintörlés esetében a Lebillegtetést kell elemeznünk. Itt d gyerek közül kell a legkisebb kulcsút kiválasztani, majd megnézni, hogy ennek a kulcsa kisebb-e, mint a szülőé. Ez d darab összehasonlítás szintenként, a fa mélysége $\lfloor \log_d n \rfloor$, így ez összesen $O(d \cdot \log_d n)$ lépés.

1.2.1. Észrevétel. *Tekintsük a* $d = 4$ *esetet, és hasonlítsuk össze a bináris kupaccal. Míg utóbbiban a Felbillegtetésnél az összehasonlítások száma* $\log n$*, addig a negyedfokúnál ez* $\log_4 n = (\log n)/2$*, tehát jobb. A Lebillegtetés esetén a bináris kupac* $2 \cdot \log n$ *összehasonlítást végez, a negyedfokú pedig* $4 \cdot \log_4 n = 2 \cdot \log n$*-et, azaz ugyanannyit. Így elmondtató, hogy a negyedfokú kupac egyértelműen jobb, mint a másodfokú.*

1.2.2. Észrevétel. *Vessük most össze a harmadfokú kupacot a negyedfokúval. A legrosszabb esetben $n+m$ darab Felbillegtetés történik, és n darab Lebillegtetés. A harmadfokú kupacnál a Felbillegtetés során $\frac{\log n}{\log 3}$, míg Lebillegtetésnél $3 \cdot \frac{\log n}{\log 3}$ összehasonlítás történik. Ahhoz, hogy megkapjuk, mikor lesz több összehasonlítás a $d = 3$ esetben, mint a $d = 4$-ben, a következő egyenlőtlenséget kell megoldanunk:*

$$(m+n) \cdot \frac{\log n}{\log 3} + n \cdot 3 \cdot \frac{\log n}{\log 3} > (m+n) \cdot \frac{\log n}{2} + n \cdot 2 \log n$$
$$\frac{m}{n} > \frac{5 \log 3 - 8}{2 - \log 3} \approx -0,18$$

Ez azt jelenti, hogy a harmadfokú kupaccal megvalósított Dijkstra-algoritmus legrosszabb esetben több összehasonlítást tesz, mint ha negyedfokút használnánk.

d-edfokú kupac esetén Dijkstra és Prim algoritmusának lépésszáma $O(n \cdot d \cdot \log_d n + m \cdot \log_d n)$. Felmerül a kérdés, hogy mely d választása esetén lesz ez a legkisebb. Egy kéttagú összeg aszimptotikusan a nagyobb taggal egyenlő. Az első tag $d \geq 3$ esetén szigorúan növő, a második tag pedig szigorúan csökkenő, tehát a nagyobbik akkor a legkisebb, ha egyenlőek.

$$\begin{aligned} n \cdot d \cdot \log_d n &= m \cdot \log_d n \\ d &= \frac{m}{n} \\ d^* &:= \begin{cases} \lfloor \frac{m}{n} \rfloor, & \text{ha } \frac{m}{n} > 4 \\ 4, & \text{különben} \end{cases} \end{aligned} \tag{1.1}$$

Azért érdemes ezt a d^* értéket választani, mert egyrészt egész számot szeretnénk, másrészt pedig az 1.2.1 és 1.2.2 észrevételek alapján érdemes másodfokú vagy harmadfokú helyett negyedfokú kupacot használni. Így a lépésszám $O(m \cdot \log_{d^*} n)$.

1.3. Vödrös kupac

Amennyiben a kulcsok nem túl nagy egész számok, tudunk jobb kupacot is csinálni. Tegyük fel, hogy a kulcsok a $[0, C]$ intervallumból kikerülő egész számok. Ekkor felveszünk $C + 1$ vödröt, minden lehetséges kulcshoz egyet. A vödröket a gyakorlatban kétszeresen láncolt listával valósítjuk meg. Így a műveletek:

- *Beszúrás*$(A, elem)$: A műveletet meghívva, az új elemet beletesszük a kulcsának megfelelő vödörbe, azaz ha a kulcsa i, hozzáfűzzük az i. listánk elejére. Ennek lépésszáma $O(1)$.
- *Kulcs-csökkentés*$(A, elem, \Delta)$: A megadott elemet kifűzzük, csökkentjük a kulcsát, és belerakjuk az új vödörbe. Mivel kétszeresen láncolt listánk van, ez is $O(1)$ lépés alatt végrehajtható, ha híváskor rámutatunk az elemre.

- *Mintörlés*(A): Először meg kell találnunk a legkisebb kulcsú nem üres vödröt, ami akár C lépés is lehet. Ebből a vödörből az első elemet kitörölni, és adatait visszaadni már megy konstans időben, így összességében ennek a műveletnek a lépésszáma $O(C)$.

Ez a kupac jól működik a Prim-algoritmusra: ha az élsúlyok a $[0, C]$ intervallumból kerülnek ki, a lépésszáma $O(m + n \cdot C)$. A Dijkstra-algoritmusnál már nem ilyen jó a helyzet. Bár az élsúlyok nem túl nagyok, attól még a kulcsok akár $(n-1) \cdot C$ nagyságúak is lehetnek, így a lépésszáma $O(m + n^2 \cdot C)$.

Definíció. Egy kupacot *monoton kupacnak* nevezünk, amennyiben:

- a Mintörlések során visszaadott kulcsok monoton növő sorrendben követik egymást,
- ha az utolsó Mintörléssel kapott kulcs $d_{\min}$, úgy az összes aktuális kulcs eleme a $[d_{\min}, d_{\min} + C]$ tartománynak, valamilyen C konstansra.

Dial [10] ötlete alapján, amennyiben a Mintörléssel kapott kulcsok monoton növő sorrendben követik egymást, úgy érdemes megjegyezni az aktuális vödör sorszámát, amiből töröltünk, és nem mindig elölről kezdeni a keresést. Jelölje K a maximális kulcsértéket, ekkor n darab Mintörlés összideje $O(nK)$ helyett $O(n + K)$.

A Dijkstra-algoritmus a monoton kupacoknál leírt második tulajdonságot is teljesíti, amit Dial a következőképpen használt ki. $C := \max\{c(e)\}$, azaz a legdrágább él költsége. Ekkor $K \leq (n-1) \cdot C$. Az előző ötlettel együtt a vödröket modulo $C + 1$ tekintjük, és mindig az az első, amiből éppen a Mintörléssel szedtünk ki elemet. Így n darab Mintörlés összideje $O(nC)$.

1.4. r-kupac

Az r- vagy radix-kupac egy monoton kupac, mely csak akkor használható, ha a kulcsok egész számok, viszont ezekben az esetekben hatékonyabb a korábban tárgyalt, d-edfokú és vödrös kupacoknál, ha C elég kicsi. Ezt a kupacot először 1990-ben publikálta Ahuja, Mehlhorn, Orlin és Tarjan [11].

Itt is vödröket fogunk készíteni, de most nem minden kulcshoz egy vödröt, hanem a kulcsok lehetséges értékeit intervallumokra bontjuk, és minden intervallumhoz fog tartozni egy vödör. Ezen intervallumok hossza exponenciálisan növekszik. Egy vödör megvalósítása továbbra is egy kétszeresen láncolt listával történik.

Legyen $c := \lceil \log(C + 1) \rceil + 2$. Vödröket készítünk, $B_1, B_2, \ldots B_c$ névvel. Az i. vödörbe azokat az elemeket rakjuk, amiknek a kulcsa eleme a $\text{range}(B_i)$-nek.

$\text{range}(B_i) = [u_{i-1} + 1, u_i]$. Kezdetben:

$$u_i := \begin{cases} -1, & \text{ha } i = 0 \\ 2^{i-1} - 1, & \text{ha } 0 < i < c \\ n \cdot C, & \text{ha } i = c \end{cases}$$

A későbbiekben ezen u_i értékeket változtatni fogjuk, de fenntartjuk, hogy $|\text{range}(B_1)| = 1$ és $|\text{range}(B_i)| \leq 2^{i-2}$, ha $i = 2, 3, \ldots, c-1$. Továbbá az u_i sorozat monoton növő lesz, de nem szigorúan, így előfordulhat, hogy $\text{range}(B_i) = \emptyset$ valamilyen i-re.

A kupacműveletek amortizációs elemzéséhez definiálnunk kell egy P potenciált.

$$P := \sum_{v \in \text{kupac}} b(v), \text{ ahol } b(v) \text{ a } v\text{-t tartalmazó vödör sorszáma.} \tag{1.2}$$

Továbbá vezessük be az alábbi jelöléseket:

- TI jelölje egy művelet tényleges idejét, azaz, hogy az algoritmus hány lépést tesz a művelet végrehajtása közben. Feltételezzük, hogy egy lépés alatt képesek vagyunk egy vödörből kivenni egy elemet, megvizsgálni, hogy az egy adott vödörbe való-e, és ha igen, akkor bele is rakni.
- A potenciálváltozás legyen $\Delta P := P_{\text{művelet után}} - P_{\text{művelet előtt}}$.
- Egy művelet amortizációs ideje pedig definíció szerint $AI := TI + \Delta P$.

Műveletek:

Beszúrás(A, v): Hátulról indulva megkeressük azt a vödröt, amelyikbe a beszúrandó v való, és hozzáfűzzük a lista elejére.

- $TI = c - b(v) + 1$
- $\Delta P = b(v)$
- $AI = c + 1 = O(\log C)$

Kulcs-csökkentés(A, v, Δ): v kulcsát csökkentjük Δ-val, majd a vödrökön balra végiglépkedve megkeressük azt, amibe az új kulcs alapján való.

- $TI = b_{\text{régi}}(v) - b_{\text{új}}(v) + 1$
- $\Delta P = b_{\text{új}}(v) - b_{\text{régi}}(v)$
- $AI = 1$

Mintörlés(A): Amennyiben az első vödörben (B_1-ben) van elem, kiveszünk egyet, és visszaadjuk. Különben a dolgunk kicsit bonyolultabb, de ebben rejlik a kupac hatékonysága. Megkeressük az első nem üres vödröt, ezen belül is egy minimális kulcsú elemet. A vödör indexét jelölje j, az elemet v, kulcsát pedig $d_{\min}$. Átcímkézzük u_i-ket úgy, hogy a $d_{\min}$ az első vödörbe kerüljön, és továbbra is teljesüljön, hogy a vödrök mérete exponenciálisan nő, és $|\text{range}(B_i)| \leq 2^{i-2}$.

$$u_i := \begin{cases} d_{\min} - 1, & \text{ha } i = 0 \\ d_{\min}, & \text{ha } i = 1 \\ \min\{u_{i-1} + 2^{i-2}, u_j\}, & \text{ha } i = 2, 3, \ldots, j-1 \\ \text{egyébként maradjon változatlan.} \end{cases}$$

Ezzel $\text{range}(B_j) = \emptyset$, így minden elemét átrakjuk a jó vödörbe, hasonlóan a Kulcs-csökkentéshez, egyesével balra lépkedve. Ahhoz, hogy az alábbi amortizációs elemzés megállja a helyét, most egy lépésben kétszer annyi elemi műveletet hajtunk végre. Ezzel a Kulcs-csökkentés és Mintörlés amortizációs ideje is javul. A részletes elemzés megtalálható Király Zoltán [7] jegyzetében.

- $TI = \frac{1}{2} \cdot (2j + |B_j| + 1 + \sum\limits_{u \in B_j, u \neq v} (j - b_{\text{új}}(u)))$
- $\Delta P = -j - \sum\limits_{u \in B_j, u \neq v} (j - b_{\text{új}}(u))$
- $AI \leq 1$

Mivel a Dijkstra-algoritmusban n Beszúrást és Mintörlést, illetve m Kulcs-csökkentést végzünk, a lépésszám r-kupaccal $O(m + n \cdot \log C)$.

1.4.1. Az r-kupac javításai

1. **javítás**

A Beszúrás művelet tényleges ideje $O(c)$-ről $O(\log c)$-re javítható. Jelölje K a beszúrandó elem kulcsát. Ha $u_{c-\log c} < K$, akkor úgy ahogy eddig, hátulról megkeressük a megfelelő vödröt és az elemet beszúrjuk. Így legfeljebb $\log c$ vödröt vizsgálunk. Ha $u_{c-\log c} \geq K$, akkor pedig az első $c - \log c$ vödörben felező kereséssel keressük meg az elem helyét. Így legrosszabb esetben $2 + \log c$ lépést teszünk, míg az amortizációs idő $c + 1$ marad.

2. **javítás**

A legkisebb nem üres vödör megtalálásának tényleges ideje is levihető $O(1)$-re. Az egyszerűség kedvéért a továbbiakban a vödröket 0-tól $c-1$-ig indexeljük. Legyen $x := \sum_{i=0}^{c-1} x_i \cdot 2^i$ egy c bites szám, ahol $x_i = 0$, ha az i. vödör üres, és $x_i = 1$ ha nem. Ezt a kupacműveletek meghívásakor $O(1)$ extra lépéssel tudjuk frissíteni. Amikor a legkisebb indexű nem üres vödröt keressük, a legkisebb i-t kell megtalálnunk, amire $x_i = 1$. Ezt a $\log(x \textit{ AND NOT } (x-1))$ művelettel kaphatjuk meg. Fredman és Willard [12] eredménye alapján a word RAM modellben egy szám kettes alapú logaritmusának egészrészét is megkaphatjuk $O(1)$ lépésben, így a fenti műveletnek a kiszámítása valóban konstans sok lépés.

3. **javítás**

A Beszúrás tényleges ideje $O(1)$-re javítható, de ehhez a kupac felépítését alapjaiban kell megváltoztatni. Ugyanúgy c vödröt készítünk, és a kulcsok bináris felírásával dolgozunk. Mivel a legnagyobb kulcs, ami a kupacba kerülhet $(n-1)C$, ezért a kulcsokat $k := \lceil \log((n-1)C+1) \rceil$ bitesnek tekintjük. Ha valamelyik kulcs ennél kevesebb számjegyből áll, akkor a rövidebbek elejére 0-kat írunk. Így egy kulcs $K = \sum_{i=0}^{k-1} K_i \cdot 2^i$, a legutóbb törölt elem kulcsa pedig $d_{\min} = \sum_{i=0}^{k-1} (d_{\min})_i \cdot 2^i$ alakban írható fel.

Most nem tartunk fent u_i korlátokat, de ugyanúgy igaz lesz, hogy egy nagyobb indexű vödörben nagyobb elemek vannak. Teljesül továbbá, hogy $|range(B_0)| = 1$, illetve vagy $range(B_i) = 2^i$, vagy pedig $range(B_i) = \emptyset$. Egy K kulcsú elem helye:

$$\begin{cases} 0, & \text{ha } K = d_{\min} \\ i, & \text{ha } 1 \leq i \leq c-2, \quad K_{i-1} \neq (d_{\min})_{i-1} \text{ és } K_j = (d_{\min})_j \quad \forall j \geq i \\ c-1, & \text{különben.} \end{cases}$$

Ezt a tulajdonságot fenntartjuk minden elemre, miközben $d_{\min}$ változik. Egy elem helye meghatározható $O(1)$ lépésben az 1.3 képlettel:

$$\min\{c-1, \lceil \log((K \ XOR \ d_{\min}) + 1) \rceil\} \tag{1.3}$$

Műveletek:

Itt is az 1.2 képletben bevezetett potenciált használjuk. Feltesszük, hogy egy lépésbe belefér az, hogy egy vödörből kivegyünk egy elemet, az 1.3 képlettel meghatározzuk, hogy melyik vödörbe való, és belerakjuk.

Beszúrás(A, v): Megkeressük a megfelelő vödröt (1.3), és az elejére befűzzük v-t.

- $TI = 1$
- $\Delta P = b(v)$
- $AI = b(v) + 1 = O(\log C)$

Kulcs-csökkentés(A, v, Δ): v kulcsát csökkentjük Δ-val, majd meghatározzuk a vödröt amibe való. Ha ez nem az a vödör, amiben benne van, akkor átrakjuk.

- $TI = 1$
- $\Delta P = b_{\text{új}}(v) - b_{\text{régi}}(v)$
- $AI = 1 + b_{\text{új}}(v) - b_{\text{régi}}(v) \leq 1$

Mintörlés(A): Először megkeressük az első nem üres vödröt, amit a korábban bemutatott javítás szerint megtehetünk $O(1)$-ben, így feltehetjük, hogy ez önmagában belefér egy lépésbe. Ha ez a 0 indexű vödör, akkor az első elemét töröljük. Különben keresünk benne egy minimális kulcsú elemet, amit törlünk és frissítjük $d_{\min}$-t, végül a vödör minden elemét átrakjuk az új helyére, a következő 1.4.1 állítás miatt más elemeket pedig nem kell átrakni.

1.4.1. Állítás. *Ha Mintörléskor a j. vödörből töröltünk, akkor az $i > j$ vödrökben lévő elemeket nem kell átrakni.*

Bizonyítás. A korábbi minimumot jelölje $d_{\min}$, az újat pedig $d'_{\min}$. Mivel $d'_{\min}$ a j. vödörből került ki, ezért a $k-1,\ldots,j$ bitekben megegyezik $d_{\min}$-nel. Így minden $i > j$ vödörben lévő elemre elmondható, hogy ha $K_{i-1} \neq (d_{\min})_{i-1}$, de minden $l > i$ esetén $K_l = (d_{\min})_l$, akkor ez a $d_{\min}$ megváltoztatása után is igaz lesz. □

Most pedig nézzük meg a Mintörlés művelet amortizációs idejét.

- $TI = 1 + |B_j|$
- $\Delta P = -j - \sum_{u \in B_j, u \neq v} (j - b_{\text{új}}(u))$
- $AI = 1 + |B_j| - j - \sum_{u \in B_j, u \neq v} (j - b_{\text{új}}(u)) \leq 0$ (Az 1.4.3 állítást használva.)

1.4.2. Állítás. *A $c-1$. vödörre igaz, hogy a benne lévő elemek kulcsai az $i \geq c-2$ bitekben megegyeznek.*

Bizonyítás. Monoton kupac esetén egy beszúrt elem kulcsa a $[d_{\min}, d_{\min}+C]$ tartományból kerül ki. Mivel C egy $c-2$ bites szám, ezért minden K kulcsra a $\overline{K_{k-1}K_{k-2}\ldots K_{c-2}}$ szám vagy megegyezik a $\overline{(d_{\min})_{k-1}(d_{\min})_{k-2}\ldots(d_{\min})_{c-2}}$ számmal, azaz a K kulcsú elem a $0,\ldots,c-2$ vödrök valamelyikébe kerül, vagy pedig

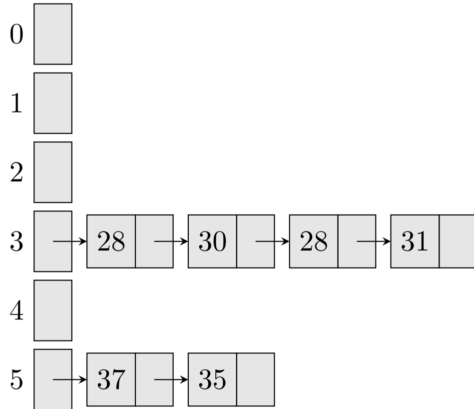
$$\overline{K_{k-1}K_{k-2}\ldots K_{c-2}} = \overline{(d_{\min})_{k-1}(d_{\min})_{k-2}\ldots(d_{\min})_{c-2}} + 1,$$

tehát az elem a $c-1$. vödörbe kerül. □

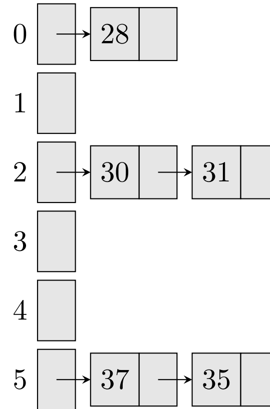
1.4.3. Állítás. *A Mintörlés művelet meghívásakor ha a $j > 0$ vödörből törlünk, akkor a vödör kiürül, és minden elem kisebb vödörbe kerül.*

Bizonyítás. A legutóbb törölt elem kulcsát jelölje $d_{\min}$, a mostani Mintörlés utániét pedig $d'_{\min}$, továbbá tegyük fel, hogy $j \leq c-2$. Ekkor egy K kulcsú elem azért volt a j. vödörben, mert $(d_{\min})_{j-1} \neq K_{j-1}$, de minden $i \geq j$-re $(d_{\min})_i = K_i$. Ez csak úgy lehetséges, ha $(d_{\min})_{j-1} = 0$ és $K_{j-1} = 1$, hiszen különben $K < d_{\min}$ lenne. Mintörléskor az új $d'_{\min}$ a j. vödör elemei közül kerül ki, ami azt jelenti, hogy $(d'_{\min})_{j-1} = K_{j-1} = 1$. Így minden $i \geq j-1$ esetén $K_i = (d'_{\min})_i$, tehát a K kulcsú elem valamely j-nél kisebb vödörbe kerül át.

Ha $j = c-1$, akkor pedig az 1.4.2 állítást használva tudjuk, hogy egy K kulcsú elem esetén $d'_{\min}$ és K megegyeznek a $k-1,\ldots,c-2$ bitekben. Így a K kulcsú elem egy $i < c-1$ vödörbe kerül át. □



1.2. ábra. Mintörlés előtt
$d_{\text{min}} = 27$.



1.3. ábra. Mintörlés után
$d_{\text{min}} = 28$.

Példa. Legyen $C = 10$, ekkor $c = 6$ vödör van. Tegyük fel, hogy az algoritmus futása közben az 1.2 ábrán látható helyzetben van a kupac. Helytakarékosság céljából csak a kulcsokat tüntettem fel, tízes számrendszerben ábrázolva.

Egy Mintörlés műveletet szeretnénk elvégezni. Ehhez megkeressük az első nem üres vödröt, ez esetben ez a 3., és benne egy legkisebb kulcsú elemet, ami a $28 = 11100_2$. Ezután d_{min}-t megváltoztatjuk, és egy ilyen kulcsú elemet törlünk.

Végül a 3. vödör elemeit szét kell osztani a kisebb vödrök között. Tekintsük az $a := 30 = 11110_2$ kulcsot. A fent leírtak szerint ez a 2. vödörbe kerül, hiszen $a_1 \neq (d_{\text{min}})_1$, de minden $i > 1$-re $a_i = (d_{\text{min}})_i$. Hasonló a helyzet a 31-gyel is. Az 5. vödör elemei ugyanott maradnak, hiszen például a $b := 37 = 100101_2$ esetén továbbra is a b_5 a legnagyobb helyiértékű számjegy, amiben d_{min}-től eltér.

1.5. Fibonacci-kupac

A Fibonacci-kupacot Fredman és Tarjan alkották meg 1987-ben [13]. A vödrös reprezentáció helyett ismét fában, pontosabban fákban tároljuk a kupac elemeit. A fák kupacrendezettek, de a korábbiakkal ellentétben nem erősen kiegyensúlyozottak, és egy csúcs gyerekeinek a számára sem adunk explicit felső korlátot. Az alább részletezett tulajdonságokból viszont következni fog, hogy egy csúcsnak legfeljebb $\log_\varphi n \approx 1,44 \cdot \log n$ gyereke lehet. Az előző két kupaccal ellentétben itt a kulcsok ismét tetszőleges valós számok lehetnek.

Tárolás:

- A kupacot egy H pointerrel azonosítjuk, mely egy minimális kulcsú gyökérre mutat,
- egy csúcshoz 4 pointert tartunk fent: szülő, bal testvér, jobb testvér és gyerek, utóbbi egy tetszőleges gyerekre mutat,

- a testvérek, illetve a fák gyökerei ciklikusan két irányban vannak körbeláncolva a testvér pointerek segítségével,
- minden csúcshoz tartozik két további érték: $r(v)$ a v csúcs rangja, azaz a gyerekeinek száma, és $m(v) \in \{0, 1\}$ jelölő-bit.

Egy csúcs megjelölt, azaz $m(v) = 1$, ha v-nek levágtuk egy gyerekét, és nem gyökér, különben $m(v) = 0$. Minden csúcs kezdetben jelöletlen, a gyökerek mindig azok. A kupacműveleteknek ismét az amortizációs idejét tekintjük, amihez vezessük be a következő potenciált.

$$P := (\text{gyökerek száma}) + 2 \cdot (\text{jelölt csúcsok száma})$$

Műveletek:

- Beszúrás$(H, elem)$: A gyökerek közé beszúrjuk az új elemet, mint egy csúcsú fa, és ha ennek kulcsa kisebb, mint az eddigi minimum, akkor H-t frissítjük. Amortizációs ideje $O(1)$.
- Mintörlés(H): Első lépésként megjegyezzük a H csúcsot, a végén ezt fogjuk visszaadni. Ezután H-t töröljük, és a helyére befűzzük a gyerekeinek listáját, a benne szereplő csúcsok szülő-pointereit null pointerre állítjuk, és levesszük róluk a jelölést. Majd amíg létezik két azonos rangú gyökér, egyesítjük a két gyökér fáját a következőképpen: ha u és v két azonos rangú gyökér, és $K(u) < K(v)$, akkor v-t bekötjük u alá, és u rangját növeljük egyel. Amikor már minden gyökér rangja különböző, végigmegyünk a gyökér listán, és H-t a minimális kulcsú gyökérre állítjuk. Amortizációs ideje $O(\log n)$.
- Kulcs-csökkentés$(H, elem, \Delta)$: Az *elemet* kivágjuk a fából, majd hozzáfűzzük a gyökér listához, és a kulcsát Δ-val csökkentjük, továbbá ha meg volt jelölve, jelöletlenné tesszük. Ekkor azt is ellenőrizni kell, hogy a kulcsa kisebb lett-e, mint H kulcsa, ha igen, akkor H-t erre a csúcsra állítjuk. Ezután az *elem* korábbi szülőjét megjelöljük, illetve ha a szülő már meg volt jelölve, úgy őt is kivágjuk, és hozzáfűzzük a gyökér listához, szintén jelöletlenné téve. Ezt addig folytatjuk a szülőkön felfelé haladva, amíg elérünk egy még jelöletlen szülőhöz, amit aztán megjelölünk, vagy egy gyökérhez. A művelet amortizációs ideje $O(1)$.

Így a Dijkstra- és Prim-algoritmus lépésszáma Fibonacci-kupaccal $O(n \log n + m)$. Ez mindkét algoritmus esetén az elérhető legjobb idő, hiszen megvalósítható velük a rendezés a következőképpen. Adott n szám, amihez egy $n + 1$ csúcsú csillaggráfot konstruálunk úgy, hogy gráf éleire az adott értékeket írjuk. Ekkor mindkét algoritmusban a Mintörlésekkel visszakapott kulcsokat sorban letárolva az eredeti n számot kapjuk vissza növekvő sorrendben.

Megjegyzés. 2012-ben Brodal, Lagogiannis és Tarjan bemutatták a Szigorú Fibonacci-kupacot [14]. Ebben a Mintörlés művelet lépésszáma legrosszabb esetben $O(\log n)$, a Beszúrás és Kulcs-csökkentés művelet lépésszáma pedig legrosszabb esetben $O(1)$.

1.6. Párosítós kupac

A Fibonacci-kupac elméletben gyors, de gyakorlatban csak ritkán, mivel egy-egy művelet megvalósítása sok elemi lépésből áll. Alternatíva lehet a gyakorlatban gyorsabb párosítós kupac, melyet Fredman és Tarjan alkotott meg 1986-ban [15]. Forrásként a fejezet elején említett [7] jegyzeten felül Stako és Vitter [16] cikkét is használtam.

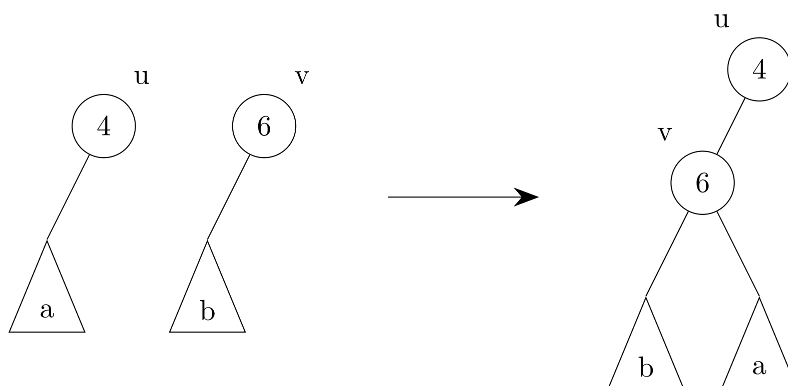
Az itt bemutatott „lusta” változat műveleteinek amortizációs ideje semmilyen potenciál mellett nem érhetik el a Fibonacci-kupacnál feltüntetett időket [17]. Bizonyítható, hogy a $P = \sum_v \log(\#v \text{ leszármazottjai})$ potenciállal mindhárom művelet amortizációs ideje $O(\log n)$. A 2009-ben bemutatott rangpárosítós verzióval [18] ugyanazok az amortizációs idők érhetők el, mint a Fibonacci-kupacban, megtartva azt az előnyt, hogy a gyakorlatban is gyors.

Definíció. Egy kupacot *párosítós kupacnak* nevezünk, amennyiben:

- egy bináris fa,
- a gyökér csúcsnak, amit H-val azonosítunk nincs jobb gyereke,
- félig kupacrendezett, azaz minden v csúcsra teljesül, hogy $K(v) \leq K(u)$, v bal gyerekének minden u leszármazottjára.

A kupacon kívül szükségünk lesz egy segédterületre, ahol több párosítós kupacot tudunk tárolni. A Beszúrás(H, v) művelet során a beszúrandó csúcs erre a segédterületre kerül, mint egy csúcsú párosítós kupac. Ez elvégezhető $O(1)$ lépés alatt.

A Kulcs-csökkentés(H, v, Δ) műveletnél a v csúcs kulcsát lecsökkentjük Δ-val, majd kivágjuk a fából a bal részfájával együtt, és a Beszúráshoz hasonlóan a segédterülethez adjuk az így kapott v gyökerű fát. A jobb részfájának szülője v korábbi szülője lesz. Lépésszáma $O(1)$.



1.4. ábra. Link művelet, ha $K(u) < K(v)$

A Mintörléshez először definiálnunk kell a Link(u, v) műveletet, ami az u és v gyökerű kupacokat egyesíti az 1.4 ábrán látható módon. A Linkeléssel kapott fa is párosítós kupac lesz.

A munka nagyrészét a Mintörlés(H) művelet meghívásakor végezzük el. Először a segédterületen található fákat egyesítjük úgy, hogy párosával Linkeljük őket keletkezési sorrendben. Annak a kupacnak a gyökerét, ami először került a segédterületre, jelölje v_1, a másodikét v_2, és így tovább. Ekkor a (v_1, v_2), $(v_3, v_4), \ldots$ csúcsokat Linkeljük egymással, és az így keletkezett új gyökeret is hozzáadjuk a segédterülethez, az utolsó Linkelendő faként.

Amikor a segédterületen már csak egy kupac maradt, akkor összeLinkeljük H-val, így egy párosítós kupacot és egy üres segédterületet kapunk. A kupacban a minimális kulcsú csúcs H, ezt levágjuk, és a Mintörlés végén visszaadjuk.

Ezután az új gyökértől jobbra lefelé haladva minden élet elvágunk, így sok kisebb párosítós kupacunk keletkezik. Ezekből két fázisban csinálunk egy kupacot. Először a segédterület javításához hasonlóan elindulunk keletkezési sorrendben (azaz fentről lefelé) ezeken a kupacokon és párosával Linkeljük őket, de most csak egyszer. A második fázisban az így kapott fák közül a két utolsót Linkeljük, és ezt addig ismételjük, amíg a végén csak egy fa marad, ezzel helyreállítva a kupacot.

2. fejezet

Tesztelés C++ nyelven

Ebben a fejezetben a C++ programozási nyelven megvalósított kupacokat és ezek viselkedését hasonlítom össze, mérési eredmények alapján. Az összehasonlítás alapját Dijkstra algoritmusa adja. Egy adott gráf esetén bináris kupacot használva futtattam az algoritmust és lementettem a kupacműveleteket, illetve a paramétereiket amikkel meghívtuk őket. Ezután már nem kellett a Dijkstra-algoritmust futtatni, csupán a mentett műveleteken keresztül szimuláltam a futását különböző kupacokat használva. Ezzel a tesztek futtatása gyorsabb volt, hiszen például a Dijkstra-algoritmusbeli összehasonlításokat nem kellett minden alkalommal elvégezni.

A Teszteléshez használt hardware:

- Processzor: Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz 2.30 GHz
- Memória: 16GB DDR4 RAM, 2133MHz
- Videókártya: Intel(R) UHD Graphics és Nvidia GeForce GTX 1050 with Max-Q Design
- Alaplap: ASUSTek COMPUTER INC. UX563FD 1.0

Az operációsrendszer Microsoft Windows 11 Home (verziószám: 10.0.22624 build 22624). Ezen keresztül a Windows Subsystem for Linux program 1.2.5.0 verzióját használva Ubuntu 20.04.5 LTS rendszeren végeztem a méréseket. Azért ezt a megoldást választottam, mert a d-edfokú kupacoknak és a Fibonacci-kupacnak a LEMON könyvtárban megvalósított verzióját használtam, amit egy linux disztribúcióra egyszerűbb volt telepíteni. A LEMON a `https://lemon.cs.elte.hu` oldalon érhető el, az általam használt verziószám: lemon-main-a278d16bd2d0. A C++ kódok fordításához a g++-9 (verziószám: 9.4.0-1ubuntu1 20.04.1) fordítóprogramot használtam. A mérésekhez és a tesztesetek generálásához használt programok, továbbá a mérések és a generált tesztek elérhetőek a `https://github.com/beer-sky/BSc-Szakdolgozat.git` címen.

2.1. Véletlenszerűen generált gráfok

2.1.1. Kis élsúlyú gráfok

Különböző csúcsszámok mellett először $m = 10n$ élszámmal generáltam irányítatlan, összefüggő véletlen gráfokat egyenletes eloszlással, az Erdős–Rényi modellt használva [19]. Adott n és m paraméterek mellett 10 gráfot generáltam, melyek éleit a $[1, 1000]$ tartományból súlyoztam, szintén véletlenszerűen, egyenletes eloszlással. A továbbiakban a vödrös kupacnak a Dial-féle megvalósítását használtam, és az egyszerűség kedvéért az eredeti radix-kupacra r_0-kupacként hivatkozok, a javításokra pedig rendre r_1-, r_2-, és r_3-kupac néven.

Egy gráfból generált műveletsoron 100-szor futtattam a tesztelést, és ennek a 100 futásidőnek a minimumát vettem, ezzel mérsékelve a számítógép által futtatott háttérfolyamatok torzítását. Így a korábban generált 10 gráf alapján 10 minimumot kaptam, amiket átlagoltam. Az eredmények a 2.1 táblázatban találhatók, milliszekundumban mérve.

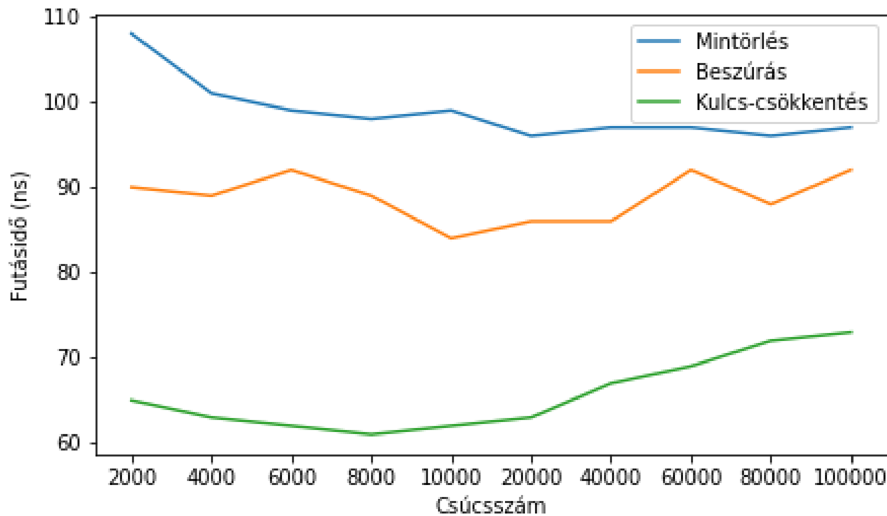
n	Bináris	$d = 4$	$d = 10$	Párosítós	Fibonacci	Vödrös	Radix (javítás nélkül)
2000	1,35	1,12	1,16	1,16	2,09	0,66	1,36
4000	2,80	2,33	2,47	3,17	4,57	1,30	2,67
6000	4,29	3,63	3,87	4,86	7,09	1,95	4,04
8000	5,90	4,97	5,18	6,64	9,70	2,56	5,24
10000	7,36	6,19	6,43	8,59	12,57	3,08	6,50
20000	15,46	12,91	13,55	18,47	27,01	6,27	13,39
40000	32,97	27,27	28,56	39,67	58,36	12,65	27,46
60000	51,87	42,54	44,42	60,71	89,99	19,50	42,05
80000	70,72	57,42	60,16	84,99	123,81	25,99	57,08
100000	91,00	73,64	77,09	121,07	158,23	33,97	71,82

2.1. táblázat. A szimulációk futásideje milliszekundumban mérve.

Ilyen kis élszám esetén a Kulcs-csökkentés művelet lépésszámának a javításától nem várunk sok gyorsítást, hiszen legrosszabb esetben is $O(n)$ Kulcs-csökkentés történik. Azt azonban láthatjuk, hogy a legjobban a vödrös kupac teljesített, így vizsgáljuk meg ezt részletesebben.

Vödrös kupac

Tekintsük a kupacműveletek átlagos futásidejét, ami a 2.1 ábrán látható. Az elemzés szerint a Kulcs-csökkentés és Beszúrás műveletek lépésszáma $O(1)$. A Beszúrás futásideje a gyakorlatban konstans, de a Kulcs-csökkentésnél azt látjuk, hogy nagyobb csúcsszámnál több ideig tart egy Kulcs-csökkentés, mint alacsony csúcsszám esetén. Ez nem mond ellent az $O(1)$-es lépésszámnak, és több dolog is magyarázhatja. Például amikor kevesebb memóriát használunk, akkor a processzor jobban tud optimalizálni, vagy éppen nagyobb



2.1. ábra. A vödrös kupacműveletek átlagos futásideje.

csúcsszámnál több lépést teszünk, mint alacsony csúcsszámnál, de egyik esetben sem többet, mint a megállapított $O(1)$-hez tartozó konstans.

Amikor a kivett elem a vödör egyetlen eleme, akkor a kivétel során 5 adattag művelet helyett csak 4-et kell elvégezni. Hasonlóan, amikor egy elemet üres vödörbe rakunk bele, akkor hat helyett csak ötször végzünk adattagon műveletet. $n = 2000$ csúcs esetén az esetek 56%-ában kell kevesebb adattaggal dolgozni amikor kivesszük a csúcsot és 15%-ában amikor beteszünk. Ezek az arányok $n = 100000$ csúcsnál 3%-ra, és közel 0%-ra módosulnak, ami magyarázza a lassulás egy részét.

A Beszúrás művelet lassabb a Kulcs-csökkentésnél. Ekkor ugyan vödörből nem kell kivenni elemet, de példányosítani kell egy objektumot, és ezt hozzáfűzni egy *std::vector*-hoz, ami a mérési eredmények alapján több időt vesz igénybe. Ezután itt is bele kell tenni az elemet egy vödörbe, ez $n = 2000$ csúcsnál még az esetek felében üres vödör, de $n = 100000$ csúcsnál már csak a 2%-ában.

A harmadik művelet, a Mintörlés esetében a legkisebb kulcsú vödör megtalálása nem minden alkalommal C lépés, mint a legrosszabb esetben lenne. $n = 2000$ esetén a 10 tesztelés során Mintörléssel kapott átlagos legnagyobb kulcs 798, míg $n = 100000$ esetén 1190. Ez azt jelenti, hogy nagyobb csúcsszámnál többször fordul elő, hogy ugyanabból a vödörből törlünk. 2000 csúcsnál egy elem törléséhez átlagosan $0,4$ vödröt kell előre lépni, míg 100000 csúcsnál csak $0,01$-et, ami megmagyarázza, hogy a csúcsszám növekedésével átlagosan miért egyre gyorsabb egy Mintörlés.

Ezt egy kicsit árnyalja a Kulcs-csökkentésnél is tárgyalt jelenség, hiszen amikor a vödörnek az utolsó elemét vesszük ki, öt helyett csak négyszer végzünk műveletet adattagon. $n = 2000$ esetén a Mintörlések 21%-ában, míg $n = 100000$ esetén csupán az 1%-ában kell csak négy műveletet végezni. Ez a lassulás elhanyagolható mértékű a nemüres vödör keresésénél tapasztalt gyorsuláshoz képest.

2.1.2. Ritka gráfok

Láthattuk, hogy ha az élsúlyok kisebbek, mint egy nem túl nagy konstans, akkor a vödrös kupacnál nincs jobb. Most az éleket az $[1, 2n]$ intervallumból súlyozzuk, az élszám pedig legyen $m := n \log n$. Az ezekből generált műveleteken a mért futásidők a 2.2 táblázatban láthatók. A radix-kupacok közül a leggyorsabb megvalósítás szerepel.

n	Bináris	$d = 4$	$d = \log n$	Párosítós	Fibonacci	Vödrös	Radix (3. javítás)
2000	1,13	0,95	1,07	1,48	2,15	0,62	1,14
4000	2,39	2,01	2,34	3,20	4,58	1,30	2,42
6000	3,82	3,52	3,81	5,07	7,15	1,96	3,69
8000	5,12	4,45	5,16	6,98	9,91	2,57	5,16
10000	6,67	5,73	6,96	9,03	12,91	3,38	6,44
20000	14,31	12,25	15,17	19,72	27,61	7,10	13,98
40000	30,72	26,09	33,03	43,08	60,38	15,00	36,48
60000	47,55	40,71	51,07	66,90	93,75	23,02	48,03
80000	66,43	56,44	71,46	93,53	131,24	32,42	63,02
100000	89,95	71,06	95,02	118,61	165,71	40,33	80,14

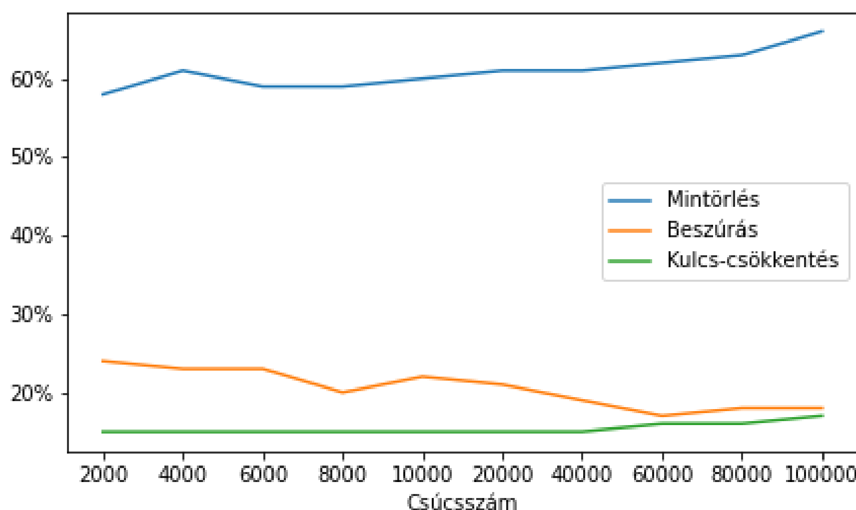
2.2. táblázat. A szimulációk futásideje milliszekundumban mérve.

d-edfokú kupacok

Látható, hogy a negyedfokú kupac gyorsabb a binárisnál, A $\log n$ fokú viszont, ami az elemzés szerint az optimális m/n fokú, csak kis csúcsszámnál gyorsabb. Ahhoz, hogy megértsük miért, vizsgáljuk az egyes kupacműveletek összidejének arányát. A Beszúrás és Kulcs-csökkentés ideje a $\log n$ fokú kupacot használva a negyedfokúnak nagyjából a 83%-a, illetve 64%-a. A Mintörlés művelet $n = 2000$ csúcsnál a 108%-a, $n = 100000$ csúcsnál pedig a 134%-a. Tehát a kupacműveletek a várakozásnak megfelelően javulnak. Az azonban érdekes, hogy például egy Beszúrás átlagos futásidejének aránya 83% marad, de a két kupac mélységének az aránya 58%-ról 50%-ra változik. Később kiderül, hogy ez azzal van összefüggésben, hogy egy Beszúrásnál (és Kulcs-csökkentésnél) átlagosan kevés szintet kell Felbillegtetni, így ilyen kis változás a kupacok mélységének arányában a futásidőben is elhanyagolhatóan kevés változást eredményez.

A 2.2 ábra azt mutatja, hogy az algoritmus mivel mennyi időt tölt a teljes futásidőhöz képest, ha a negyedfokú kupacot használjuk. Az, hogy az érétékek kicsit kevesebb, mint 100%-ra összegződnek, annak köszönhető, hogy a vezérlési szerkezetek külön nem mérhetők. Így a maradék idő az elágazások és ciklusok végrehajtásával telik.

Meglepő lehet, hogy a futás során több időt tölt az algoritmus Mintörléssel, amiből n darab van, mint Kulcs-csökkentéssel, amiből legrosszabb esetben m. Ez magyarázattal szolgál arra, hogy miért lassabb a $\log n$ fokú, hiszen pont a Mintörlés művelet az, amihez több lépésre van szükség nagyobb fokszámnál. A véletlen gráfok esetén a [20] cikk szerint



2.2. ábra. Negyedfokú kupacműveletek aránya az összidőhöz képest.

a Kulcs-csökkentések várható darabszáma $O(\min\{m, n\log(\frac{2m}{n})\})$, ami $O(n \cdot \log\log n)$. A mérések ezt megerősítik, nagyjából $n\ln(\frac{m}{2n})$ darab Kulcs-csökkentés történik.

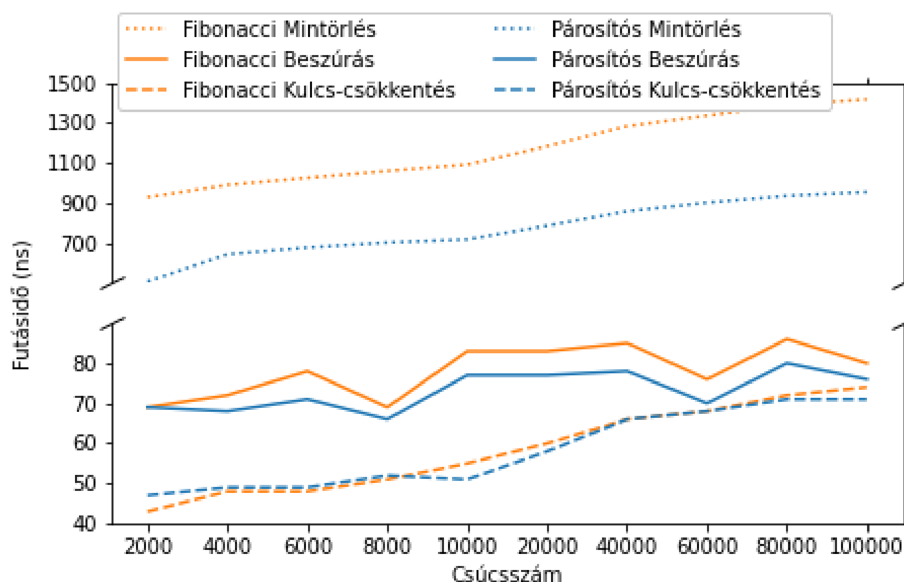
Szintén a [20] cikkben adják meg a Kulcs-csökkentések várható darabszámára kapott felső becslés alapján a kupac optimális fokszámát, $d^* := \lfloor \ln(\frac{2m}{n}) \rfloor$, ha $\frac{2m}{n} \geq 8$. Ez esetünkben $n = 2000$ és $n = 100000$ csúcsszámnál is $d^* = 3$, de az 1.2.2 megjegyzésben láttuk, hogy a harmadfokú kupac helyett mindig érdemesebb negyedfokút használni.

A mérésekből az is kiderül, hogy a negyedfokú kupacnál a Kulcs-csökkentés műveletek összideje a Beszúrásnál is kevesebb, pedig előbbiből több van. Ennek oka, hogy random gráfon dolgozunk, így egy Beszúrás során a negyedfokú kupacban átlagosan $0,4$ szintet billegtetünk fel egy csúcsot, míg egy Kulcs-csökkentés alkalmával $0,32$ szintet. Ezen felül a Beszúrásnál több adminisztratív lépést kell elvégezni, ami egy új elem hozzáadásával jár. Mintörléskor pedig 2000 csúcsnál átlagosan $4,42$, míg 100000 csúcsnál $7,25$ szintet billegtetünk le, ami megmagyarázza, hogy a Mintörlések futásidejének aránya miért nő a csúcsszám növelésével.

Fibonacci- és párosítós kupac

A Fibonacci-kupacnál azt tapasztalhatjuk, hogy ez jóval lassabb, mint bármelyik másik verzió. Ennek oka a Mintörlés művelet, amelynek amortizációs ideje $O(\log n)$, de a hozzá tartozó konstans szorzó sokkal nagyobb, mint például a d-edfokú kupacok esetén. A párosítós kupac már jobban teljesített, de még mindig lassabb, mint a korábban vizsgált adatstruktúrák.

A Beszúrás művelet mindkét kupacban $O(1)$ lépés, és az implementációban a hozzájuk tartozó konstans szorzó is megközelítőleg megegyezik, ami a 2.3 ábrán jól látszik. A Kulcs-



2.3. ábra. A párosítós és a Fibonacci-kupac műveleteinek átlagos futásideje.

csökkentés tényleges ideje a párosítós kupacban $O(1)$, a Fibonacci-kupacban pedig az amortizációs ideje $O(1)$. A mérések azt mutatják, hogy a gyakorlatban mindkét kupacot használva nagyjából ugyanannyi idő a Kulcs-csökkentés is.

A párosítós kupac nagy előnye a Fibonaccival szemben a Mintörlés, aminek futásideje az utóbbiban átlagosan másfélszer lassabb. És mivel ez a művelet teszi ki a Fibonacci-kupac futásidejének $85-87\%$-át, ezt gyorsítva úgy, hogy a Beszúrás és Kulcs-csökkentés nem lassult, sokat javítottunk a teljes futásidőn.

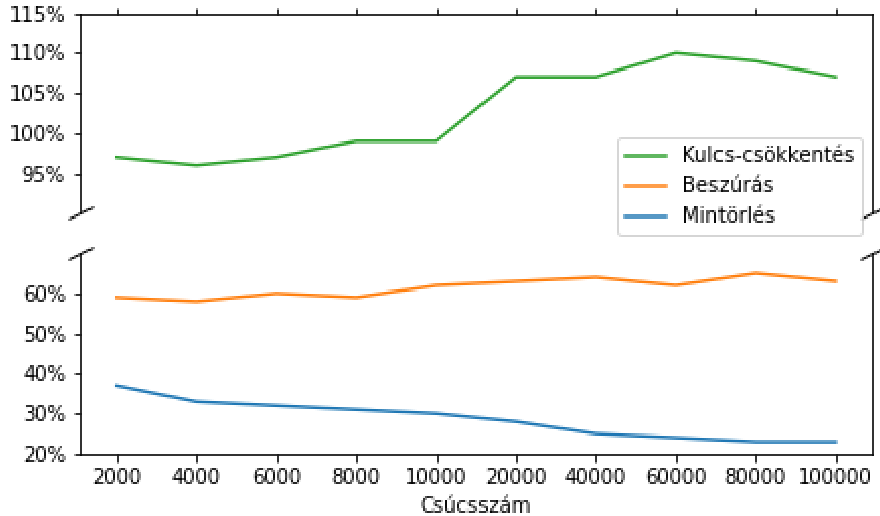
Vödrös kupac

Továbblépve a vödrös kupacra láthatjuk, hogy mindközül ez teljesít a legjobban. A Beszúrás és Kulcs-csökkentés műveletek lépésszáma $O(1)$, de a Mintörlés műveleté $O(C)$. Mivel most $C = 2n$, ez $n = 100000$ csúcsnál a bináris kupachoz viszonyítva (amiben egy Mintörlés lépésszáma $O(\log n)$) 4 nagyságrenddel több.

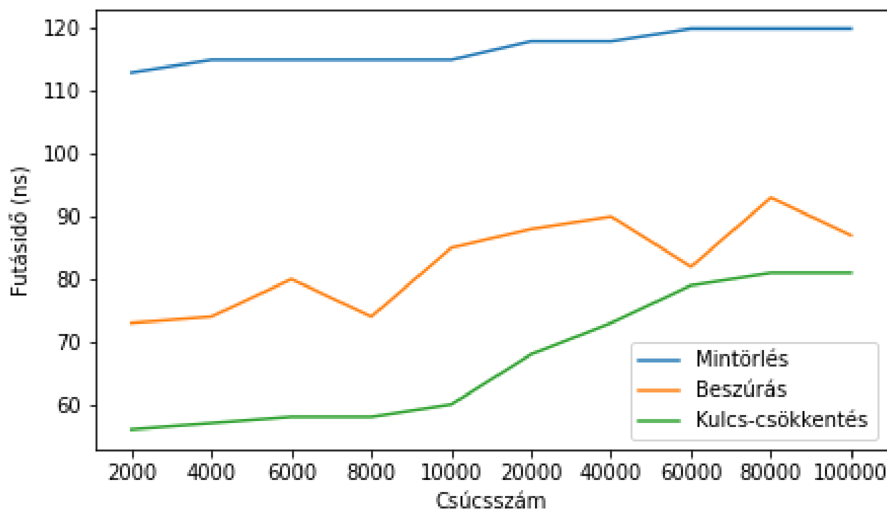
A 2.4 ábrán látható mérési eredmény ennek fényében elsőre meglepő. A Beszúrás műveletnél például elmarad a csúcsszám növekedésével várt gyorsulás. Ezt az magyarázza, hogy a mérések alapján olyan random gráf esetén, ahol $m = n \log n$, a bináris kupacban 2000 csúcsnál átlagosan $0,93$, míg 100000 csúcsnál $0,97$ szintet billegtetünk fel, azaz nem lassul lényegesen.

Bináris kupacot használva a Kulcs-csökkentésnél átlagosan $0,85$ szintet billegtetünk fel. így azt, hogy ez nagy csúcsszámnál arányaiban miért lassabb, a vödrös kupacnál kell keresni. Egy Felbillegtetés a gyakorlatban nagyjából annyi művelet elvégzését jelenti, mint a vödrös kupacban egy Kulcs-csökkentés, ahol $O(1)$ lépés megtalálni, hogy egy Kulcs-csökkentett elem melyik vödörbe kerül. Ezt az is alátámasztja, hogy kis csúcsszámnál

nagyjából ugyanannyi idő ez a művelet mindkét kupacot használva.



2.4. ábra. Vödrös és bináris kupacműveletek futásidejének aránya.



2.5. ábra. A vödrös kupacműveletek átlagos futásideje.

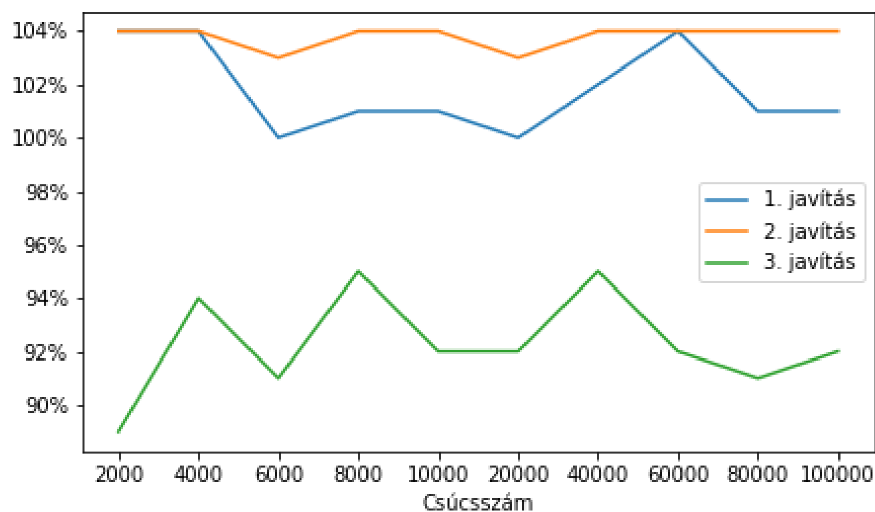
Vizsgáljuk azt, hogy a vödrös kupacban mennyi ideig tart egy-egy művelet átlagos futásideje. Ezek az adatok a 2.5 ábrán láthatók. Azt látjuk, hogy alacsonyabb csúcsszámnál kevesebb ideig tart egy Kulcs-csökkentés elvégzése, hasonlóan a kis élsúlyú gráfoknál látottakhoz. 2000 csúcs esetén a Kulcs-csökkentések 82%-ánál fordul elő, hogy üres vödörből

veszünk ki elemet, és 49%-ánál, hogy üres vödörbe tesszük bele, míg 100000 csúcsnál ezek a számok 78% és 43%, tehát a lassulásra most nem ez a magyarázat.

A Mintörlés egy kicsit lassul, ahogyan a csúcsszám nő, ellentétben a kis élsúlyozással. Mivel most a maximális élsúly a csúcsszámmal együtt skálázódik, ezért a Mintörléssel megkapott legnagyobb távolságok is nagyobbak lesznek, ha a csúcsszám nagyobb. Így több vödröt kell előre lépni, amíg megtaláljuk a következőt, ami nem üres.

r-kupac és javításai

A 2.6 ábrán az látható, hogy az r-kupac javításainak futásideje hogyan viszonyul az r_0-kupachoz. Az első két bemutatott javítás a futásidőn nem javított, de nem is állítható biztosan, hogy rontott. Az ábrán látható nagyjából 5%-os különbség a futásidőben a mérések természetes ingadozásából is fakadhat. A kupacműveletek átlagos futásidejét nézve sem látunk egyértelmű különbséget a két javítás, és az eredeti között.

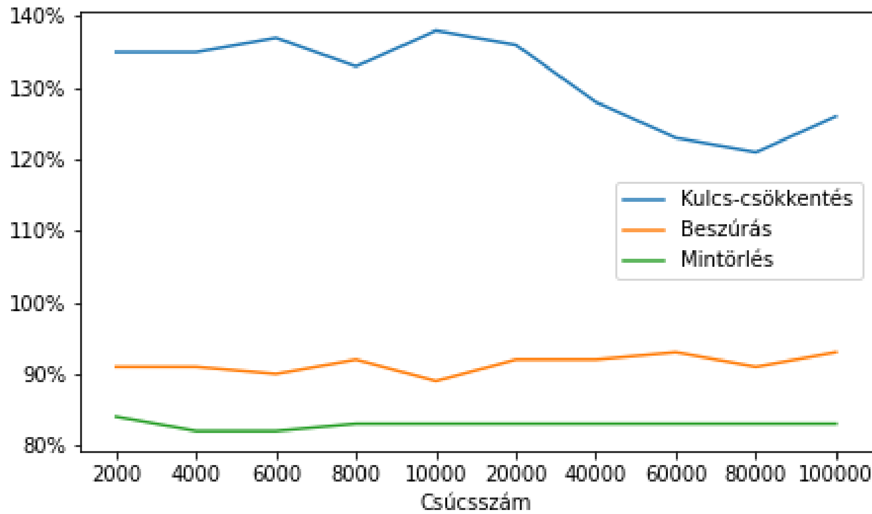


2.6. ábra. r-kupac javításainak összideje az eredetihez képest.

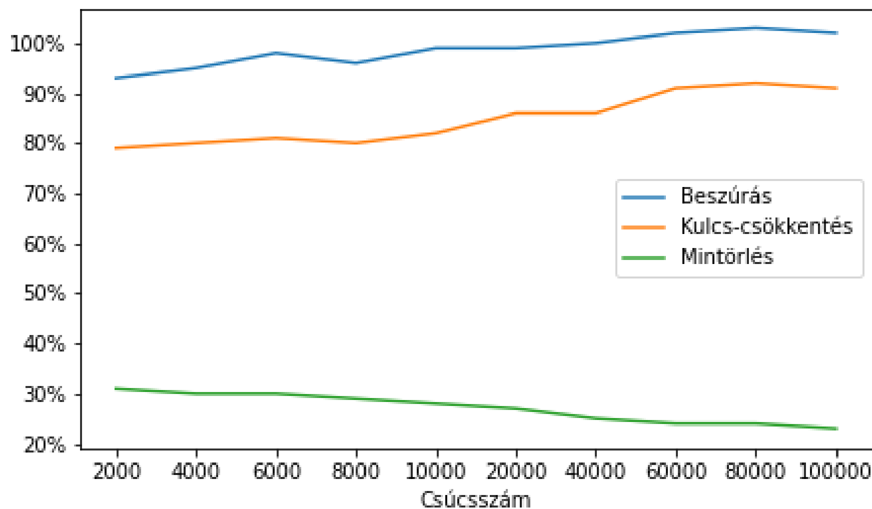
A harmadik javításnál már kicsit jobb a helyzet. Itt az implementációnál nem a logaritmus műveletet használtam a korábbi az 1.3 képletben, hanem a programozási nyelv által biztosított műveletet, amely képes a gyakorlatban gyorsabban kiszámítani azt, hogy melyik a legnagyobb helyiértékű bit, aminek értéke 1. Ez már a kupacműveletek futásideje szempontjából is érdekesebb.

A 2.7 ábrán az látható, hogy a Beszúrás és Mintörlés műveleten gyorsítottunk, de a Kulcs-csökkentés jóval lassabb lett. Nagyobb csúcsszámnál ez az arány javul, hiszen az eredeti implementációban ennek a műveletnek csak az amortizációs ideje $O(1)$, míg jelen esetben a tényleges ideje is. A lassulás oka az, hogy az eredeti implementációban, és az első két javításban Kulcs-csökkentések felénél fordul az elő, hogy az elem ugyanabban a vödörben marad. Ezt megmérve a harmadik javításra, amikor a Kulcs-csökkentés legrosszabb

esetben $O(1)$ lépés, azt tapasztaljuk, hogy szinte minden esetben új vödörbe kell rakni az elemet.



2.7. ábra. Az r_3-kupac műveleteinek futásidejének aránya az r_0-kupachoz képest.



2.8. ábra. Vödrös kupacműveletek összidejének aránya az r_3-kupacéhoz képest.

Végül hasonlítsuk össze a vödrös és az r_3-kupacot. A 2.2 táblázatban szereplő futásidőkből kiderül, hogy $n = 100000$ csúcsnál a vödrös kupac kétszer olyan gyors, mint az r_3-kupac. A 2.8 ábrán a két kupac műveleteinek összidejének az aránya látható.

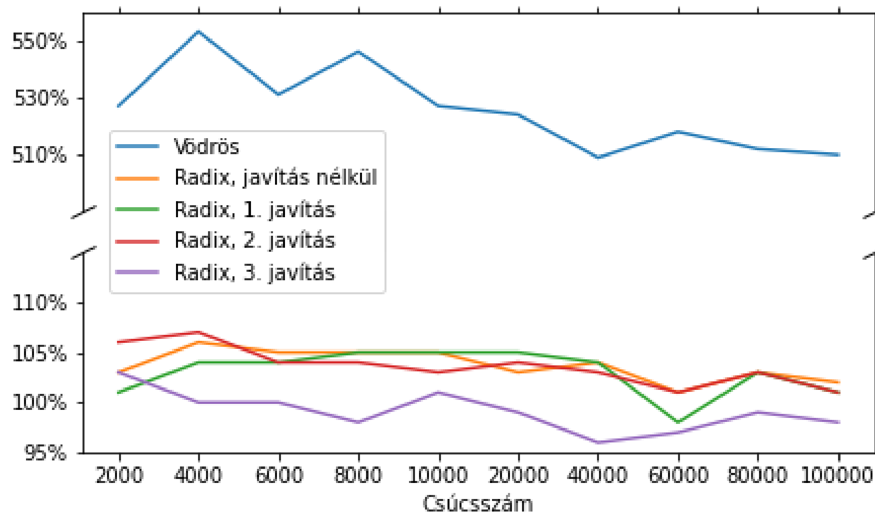
A Beszúrás nagyjából ugyanannyi idő a két kupacban. A Kulcs-csökkentésnél sincs nagy különbség, de a vödrös kupac kicsit jobban teljesít, $n = 2000$ csúcsnál még 20%-kal

gyorsabb, de $n = 100000$-nél már csak 10%-kal. Ennek oka, hogy egy Kulcs-csökkentés átlagos futásideje kis csúcsszámnál még gyorsabb a vödrös kupac esetén, ahogyan ez a 2.5 ábrán is láthattuk.

A nagy különbséget azonban a Mintörlés okozza. Ennek átlagos futásideje a vödrös kupacot használva $n = 2000$ csúcsnál $113ns$, míg $n = 100000$ csúcs esetén $120ns$. Ezzel szemben az r_3-kupacban 2000 csúcsnál $362ns$, és 100000 csúcs esetén $524ns$.

Nagyobb élsúlyok

A kulcsmanipulációs kupacoknál, azaz a vödrös és r-kupacnál érdemes azt is vizsgálni, ha az élsúlyok nagyobbak. A meglévő gráfokon az élsúlyokat a 100-szorosára változtattam, így ezek most a $[100, 200n]$ tartományból kerülnek ki 100-as lépésközzel.

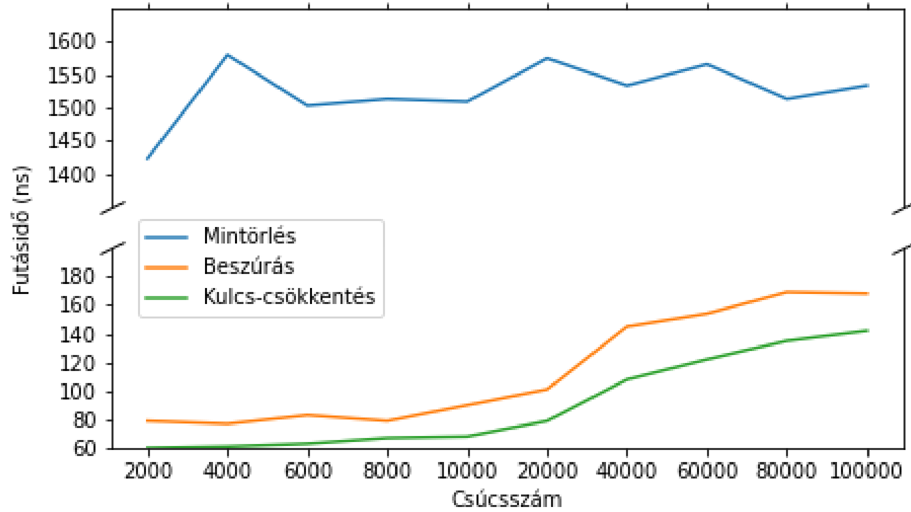


2.9. ábra. Kulcsmanipulációs kupacok futásidejének változása, ha $C = 2n$ helyett $200n$.

A 2.9 ábrán az látható, hogy hogyan aránylik a $[100, 200n]$ intervallumból vett súlyozáson mért futásidő az $[1, 2n]$-en mérthez képest. A vödrös kupac ötször lassabb lett, ezzel a leglassabb kupaccá válva. Műveleteinek átlagos futásidejét a 2.10 ábra tartalmazza.

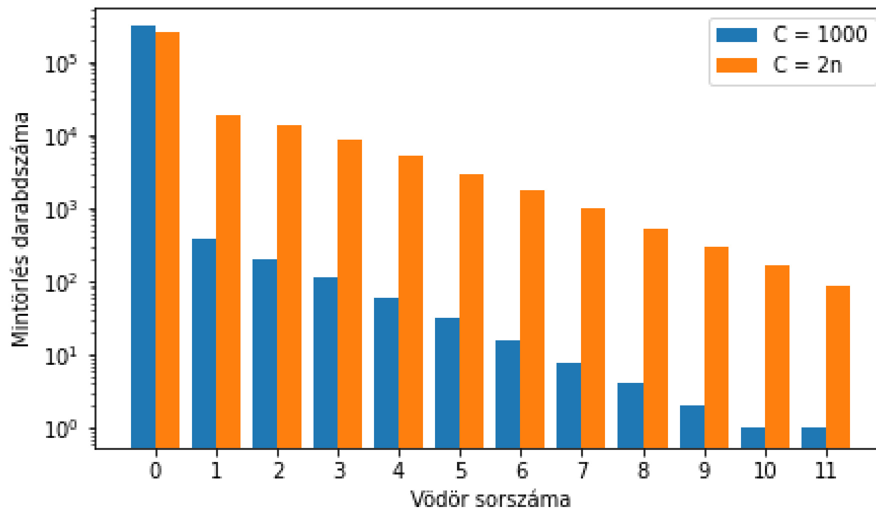
A Mintörlés jóval lassabb, mint a $C = 2n$ esetben, ahol 110 - $120ns$ volt. Ennek oka, hogy most minden alkalommal, amikor az első vödör üres, és a következő nemüres vödröt keressük, legalább 100-at kell előre lépni. A Beszúrás és Kulcs-csökkentés futásideje kis csúcsszámnál hasonló volt, mint a $C = 2n$ estben, de $n = 20000$-től nőni kezdett. Ugyan $O(1)$ a lépésszáma ennek a két műveletnek, de ha a gyakorlatban túl nagy méretű tömbökön dolgozunk, a processzor már nem tudja hatékonyan kezelni a cache-ben, így az $O(1)$-hez tartozó konstans szorzó megnő. Hogy mi számít túl nagynak, az a processzor konkrét típusától függ.

A radix kupacok sokat lassultak, amikor a maximális $C = 1000$ élsúlyt felemeltük $C = 2n$-re, de a $C = 200n$-re változtatása már nem befolyásolta a futásidőket. A lassulás



2.10. ábra. Vödrös kupacműveletek átlagos futásideje, $C = 200n$.

oka szintén a Mintörlés művelet, hiszen nagyobb tartományból véletlenszerűen súlyozva többször fordul elő, hogy a minimumot nagyobb vödörből töröljük. A 2.11 ábrán látható, hogy melyik vödörből hányszor törlünk minimumot, az r_0 kupacot használva. Azt is megfigyelhetjük, hogy a harmadik javításban a Mintörlés művelet nem csak gyorsabb a többinél, de nagyobb élköltségek mellett kevésbé is lassul.



2.11. ábra. Az r_3-kupacban Mintörlés során ürített vödrök darabszáma logaritmikus skálán, $n = 100000$.

2.1.3. Középsűrű gráfok

Vizsgáljuk a Dijkstra-algoritmus által generált műveletsor lefutását az előzőekhez hasonló gráfokon, de most $m := n^{3/2}$ élszámmal ahol az élsúlyok szintén az $[1, 2n]$ intervallumból kerülnek ki. A különböző kupacok használatával mért futásidőket a 2.3 táblázat tartalmazza. A radix-kupacból a leggyorsabb megvalósítás szerepel a táblázatban.

n	Bináris	$d = 4$	$d = \ln(2\sqrt{n})$	$d = \sqrt{n}$	Párosítós	Fibonacci	Vödrös	Radix (3. javítás)
2000	1,28	1,15	1,18	1,80	1,66	2,28	0,78	1,23
4000	2,81	2,50	2,51	4,68	3,58	5,09	1,69	2,56
6000	4,43	3,84	4,02	8,34	5,72	7,93	2,54	4,06
8000	6,05	5,19	5,97	12,57	7,92	10,94	3,47	5,38
10000	8,01	6,82	7,62	17,28	10,17	14,24	4,51	7,09
20000	16,92	14,21	16,05	47,19	22,28	30,81	9,93	15,18
40000	37,04	30,99	38,15	129,85	48,65	67,48	21,77	32,75
60000	58,51	48,04	56,89	234,36	76,14	105,15	34,53	50,56
80000	81,01	66,98	68,97	359,97	105,59	146,22	48,33	69,86
100000	103,48	85,05	88,34	501,24	134,98	186,90	60,94	88,89

2.3. táblázat. Futásidők milliszekundumban mérve, $m = n^{3/2}$.

d-edfokú kupacok

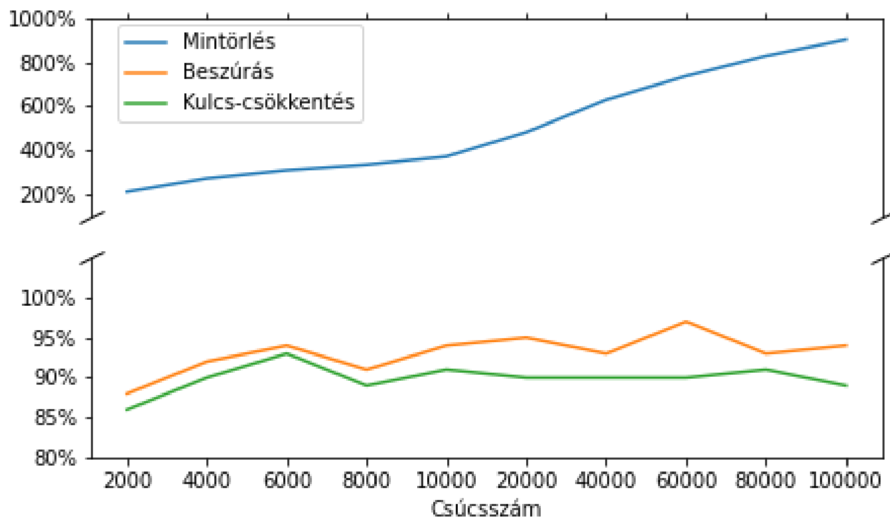
A táblázatból kiderül, hogy random gráf esetén a legrosszabb esetben optimális d^* fokú kupac (itt $d^* = \lfloor\sqrt{n}\rfloor$), amit az 1.1 egyenletben határoztunk meg, sokkal lassabb is lehet, mint a bináris. A két kupac műveleteinek aránya a 2.12 ábrán látható. Ugyan a Beszúrás és Kulcs-csökkentés műveletek gyorsabbak a $\sqrt{n}$-edfokúban, a Mintörlés sokszor lassabb. A Bináris kupacot használva a futásidő 50%-át teszi ki a Mintörlés, csúcsszámtól függetlenül. Ehhez képest a $\sqrt{n}$-edfokú kupacban a $80 - 90$%-át.

Az $m = n \log n$ esethez hasonlóan, mivel véletlenszerűen generált gráfon dolgozunk, a Kulcs-csökkentések várható darabszáma $O(\min\{m, n\log(\frac{2m}{n})\}) = O(n \log n)$. Például 100000 csúcsnál 500000 Kulcs-csökkentés történik, ezért nem éri meg ilyen nagy fokú kupacot használni ebben az esetben sem.

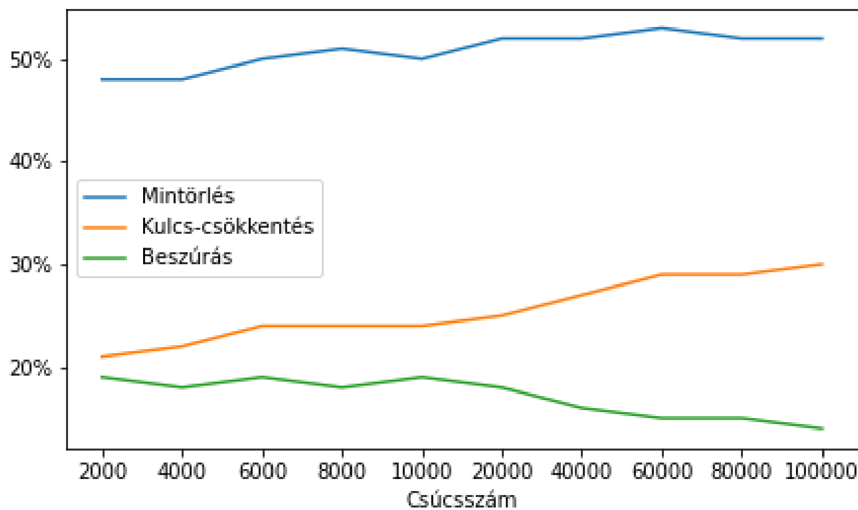
A kulcs-csökkentések várható darabszámára kapott felső becslés alapján, a [20] cikk szerint az optimális kupac foka $d^* = \lfloor\ln(\frac{2m}{n})\rfloor = \lfloor\ln(2\sqrt{n})\rfloor$. Ez $n = 2000$ csúcs esetén a negyedfokú, $n = 100000$ csúcsnál pedig már a hatodfokú lenne. A 2.3 táblázatban látható mérési eredmények alapján azonban ezutóbbi egy kicsit lassabb a negyedfokúnál.

Érdemes még megvizsgálni a negyedfokú kupacot használva a kupacműveletek összidejének arányát, ami a 2.13 ábrán látható. Vessük össze ezt a 2.2 ábrával, amely szintén a kupacműveletek arányát tartalmazza, de az $m = n \log n$ esetben.

Az első szembetűnő dolog, hogy a ritka gráfok alapján generált tesztekben a Kulcs-csökkentések a futásidőnek mindössze 15%-át tették ki, de az $m = n^{3/2}$ élszám esetén ez az arány 20%-ról 30%-ra nő, ahogy a csúcsszámot 2000-ről 100000-re növeljük. Ez



2.12. ábra. Bináris és $\sqrt{n}$-edfokú kupacműveletek futásidejének aránya, $m = n^{3/2}$.

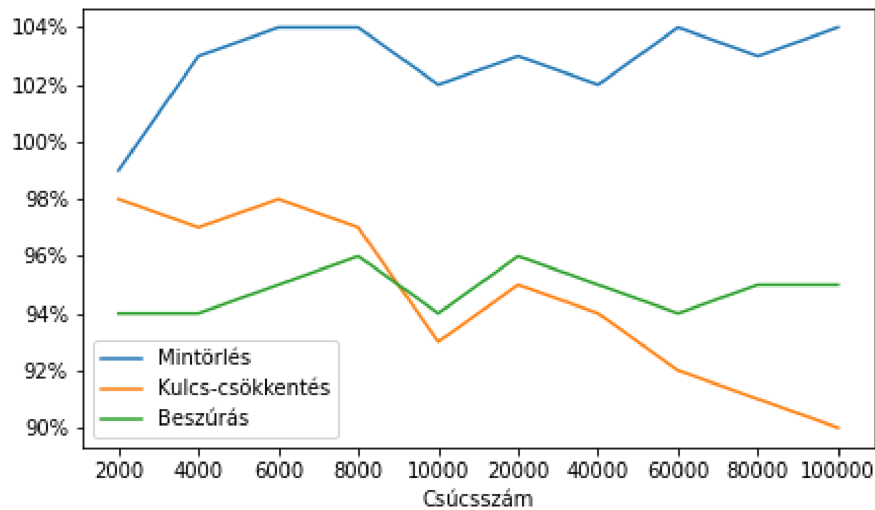


2.13. ábra. Negyedfokú kupacműveletek összidejének aránya az teljes futásidőhöz képest, $m = n^{3/2}$.

nem meglepő, hiszen most várhatóan $O(n \log \log n)$ helyett $O(n \log n)$ Kulcs-csökkentés történik.

A Kulcs-csökkentések már a futásidő nagyobb részét teszik ki, mint a Beszúrások, ellentétben a ritka gráfokkal, de a Beszúrások aránya az összidőhöz képest ugyanúgy csökken. A

Felbillegtetett szintek átlagos száma szinte megegyezik a ritka gráfoknál mért számokkal. Egy Beszúrás során átlagosan $0,43$, míg egy Kulcs-csökkentés során $0,29$ szintet Billegtetünk fel. A Futásidő nagyrészét most is a Mintörlés teszi ki, amiben $n = 2000$ csúcsnál átlagosan $4,46$ szintet, $n = 100000$ csúcsnál pedig $7,28$ szintet kell Lebillegtetni. Ezek a számok is megegyeznek az $m = n \log n$ élszámnál mért értékekkel.



2.14. ábra. Fibonacci-kupacműveletek átlagos idejének aránya a ritka gráfokhoz képest.

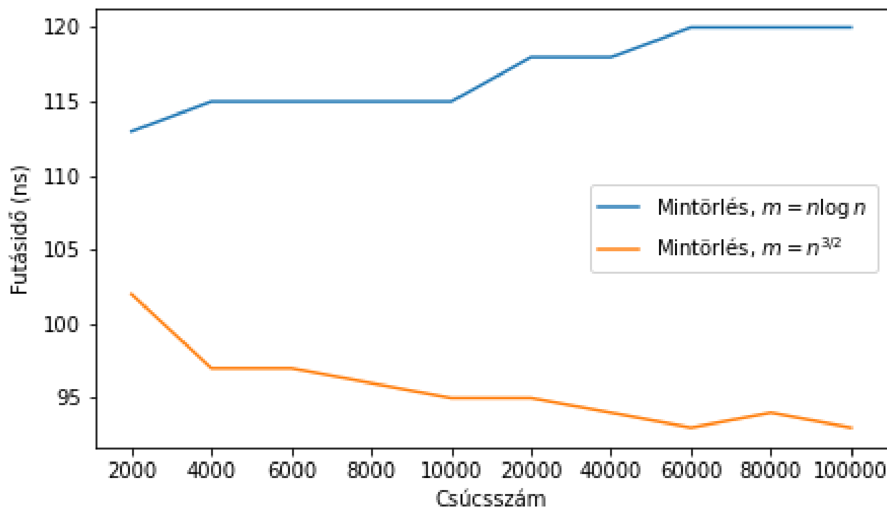
Fibonacci- és párosítós kupac

A 2.3 táblázatból az is látható, hogy a párosítós és a Fibonacci-kupacok továbbra sem gyorsabbak a többi kupacnál (kivétel a $d = \sqrt{n}$). Mindkét kupac esetében a Beszúrás és Mintörlés kupacművelet átlagos futásidejét elemezve majdnem ugyanazokat az eredményeket kapjuk, mint a ritka gráfok esetén. A 2.14 ábrán a Fibonacci kupacműveletek aránya látható a ritka gráfokhoz viszonyítva. A Kulcs-csökkentések egy kicsit változnak, $n = 100000$ csúcsnál a párosítós kupacban 8%-kal, míg a Fibonacci-kupacban 10%-kal gyorsabbak a sűrű gráfok alapján generált tesztelésekben, mint az $m = n \log n$ élszám esetén. Ennek oka, hogy amikor két Mintörlés között többször csökkentjük ugyanannak az elemnek a kulcsát, csak először kell kivágni azt a fából. Később már a gyökér listában, illetve a segédterületen található elem Kulcs-csökkentése gyorsabban elvégezhető.

Kulcsmanipulációs kupacok

A vödrös és radix-kupacoknál a Beszúrás és Kulcs-csökkentés műveletek átlagos futásideje ugyanannyi a középsűrű, és a ritka gráfnál, ahogyan az az elemzés alapján várható volt. Nézzük a Mintörlést először a vödrös kupac esetében, ami a 2.15 ábrán látható. Most

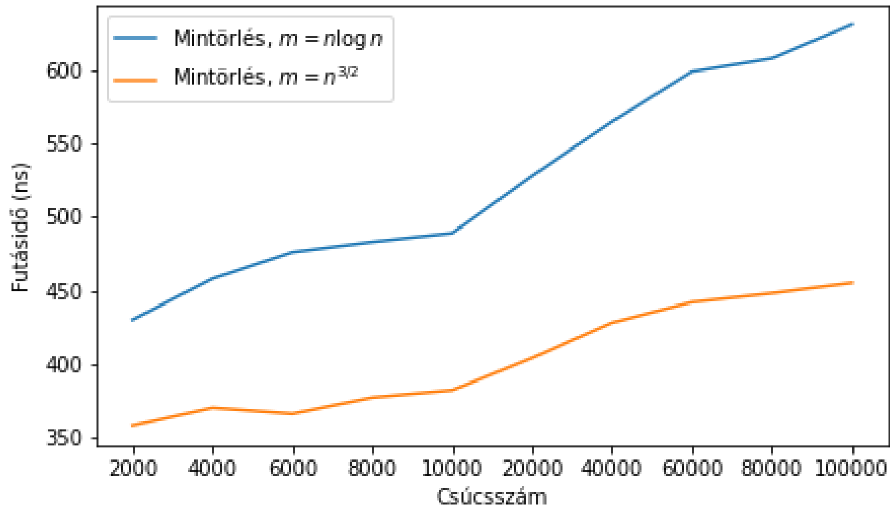
egy Mintörlés átlagos ideje a csúcsszám növelésével nem nő ahogy korábban, hanem csökken. Ennek oka, hogy egy ritka gráf alapján generált műveletsorban egy Mintörlés során csúcsszámtól függetlenül átlagosan $1,47$ vödröt kell előre lépni, mire megtaláljuk az elsőt, ami nem üres. Ha az élszám $m = n^{3/2}$, akkor 2000 csúcsnál még átlagosan $0,4$ vödröt lépünk előre, de 100000 csúcsnál már csak $0,08$-at.



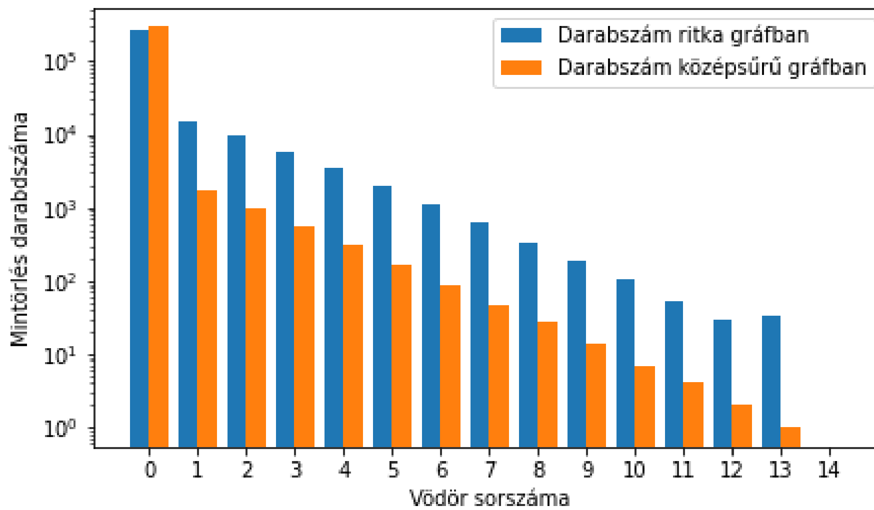
2.15. ábra. Egy Mintörlés átlagos futásideje vödrös kupacot használva.

Ahogyan az a 2.16 ábrán látható, az r_0-kupacban a Mintörlés futásideje ugyan mindkét esetben nő, de a sűrű gráfból generált teszten egy Mintörlés átlagosan jóval gyorsabb. Ennek oka hasonló, mint a vödrös kupacnál. Mivel több véletlenszerűen súlyozott él van, a legrövidebb utak is rövidebbek lesznek. Ez azt jelenti, hogy több olyan csúcs van, amik hasonlóan távol vannak a kiinduló csúcstól. Így amikor Mintörlést hajtunk végre, sokszor a minimumot kisebb kulcsokat tartalmazó vödörben találjuk meg, és azokat a vödröket, amik nagy kulcsú elemeket tartalmaznak, gyakran Kulcs-csökkentés során ürítjük ki. Ezt alátámasztja a 2.17 ábra, amelyen azt szemléltettem, hogy melyik vödörből hányszor törlünk ritka, illetve sűrű gráf esetén, ha $n = 100000$.

A radix-kupacot a javításaival összevetve ismét azt láthatjuk, hogy alig van különbség a futásidőben, és ez alól a harmadik javítás sem kivétel. Ezutóbbi esetben a műveletek futásidejének arányai hasonlítanak a 2.7 ábrán látotthoz, de most a Kulcs-csökkentés 130%-ról 115%-ra csökken, ahogyan a csúcsszám nő. Mivel a Kulcs-csökkentés a teljes futásidőnek csak a 20%-át teszi ki, ezért ez a futásidőn nem észrevehető javulás.



2.16. ábra. Egy Mintörlés átlagos futásideje az r_0-kupacot használva.

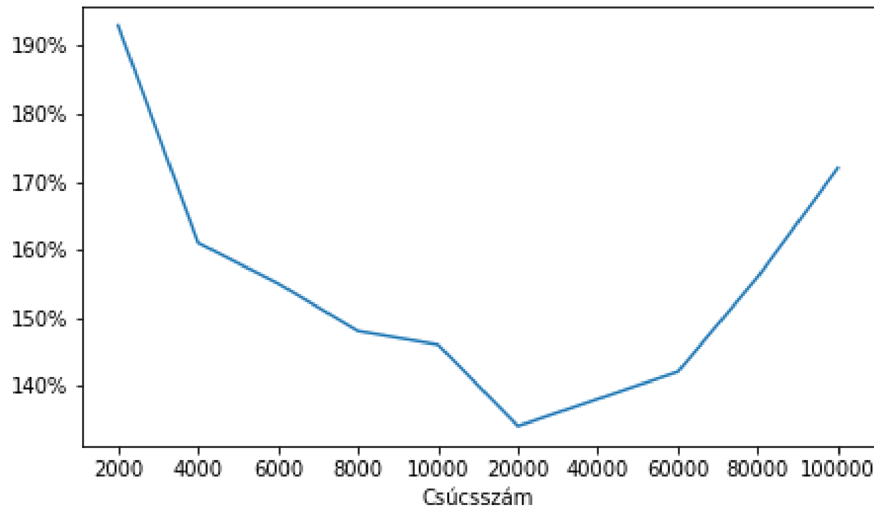


2.17. ábra. Mintörlés során ürített vödrök r_0-kupacot használva, logaritmikus skálán, $n =$ 100000.

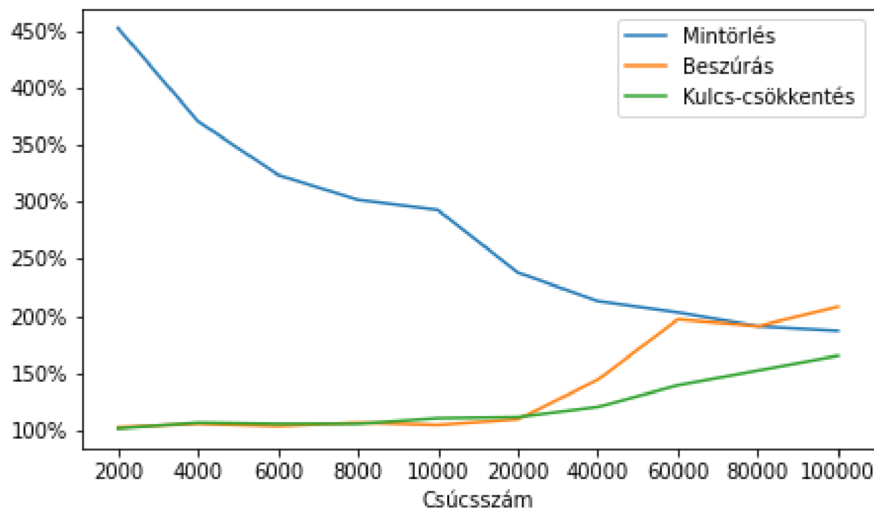
Nagyobb élsúlyok

Ismét vizsgáljuk a kulcsmanipulációs kupacokat úgy, hogy az élsúlyokat felszorozzuk 100-zal. A 2.18 ábrán látható, hogyan aránylik a $C = 200n$ maximális élsúlyú gráfból

generált teszteset a $C = 2n$-hez képest vödrös kupacot használva ($m = n^{3/2}$). Ahhoz hogy jobban megértsük a grafikont, hasonlítsuk össze a műveletek futásidejének arányát is, amik a 2.19 ábrán találhatók.



2.18. ábra. Vödrös kupac futásidejének aránya a $C = 2n$-hez képest, ha $C = 200n$.

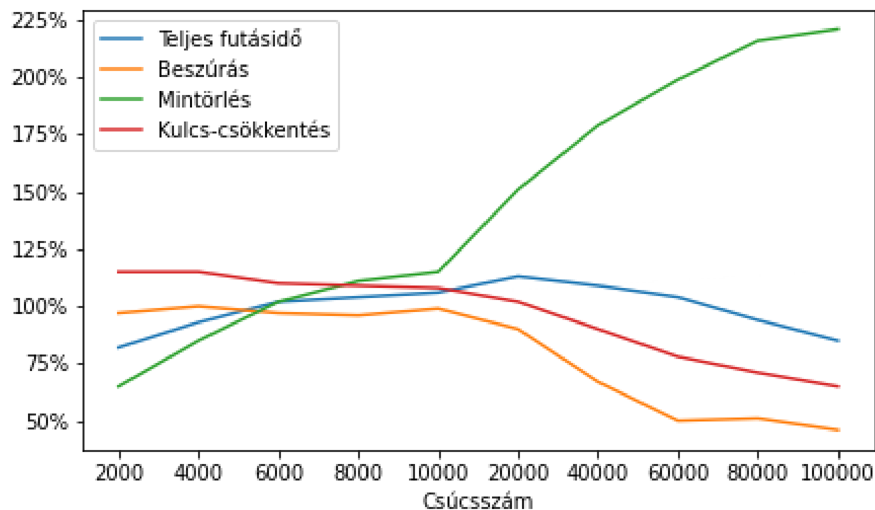


2.19. ábra. Vödrös kupacműveletek futásidejének változása a $C = 2n$-hez képest, ha $C = 200n$.

Kis csúcsszám esetén a Beszúrás és Kulcs-csökkentés műveletek ugyanannyi ideig tar-

tanak a kis és nagy élsúlyok esetén is, és a ritka gráfokon mért eredményekhez hasonlóan lassulnak. A Mintörlések futásideje ellenben javul a csúcsszám növelésével. Ezt az magyarázza, hogy a $C = 2n$ esetben 2000 csúcsnál még átlagosan $0,4$ vödröt lépünk előre, de 100000 csúcsnál már csak $0,08$-at. Ez úgy lehet, hogy sokszor törlünk ugyanabból a vödörből, és minél nagyobb a csúcsszám, arányaiban annál többször fordul ez elő. $C = 200n$ esetén minden alkalommal, amikor az aktuális első vödör üres, nem egy, hanem 100 lépést kell tenni. Mivel kis csúcsszámnál ez gyakrabban fordul elő, ezért ott ez nagyobb lassulást eredményez.

A 2.18 ábrán $n = 20000$-nél kezdődő növekedést két dolog együttes hatása befolyásolja. Az egyik, hogy a Mintörlés 2000 csúcs esetén még a futásidő 62%-át teszi ki, de 100000 csúcsnál már csak a 17%-át. A másik pedig, hogy a Beszúrás és Kulcs-csökkentés műveletek $n = 20000$ csúcsszám felett kezdenek lassulni a $C = 2n$ esethez képest, így ezek a teljes futásidőt is nagy mértékben növelik.

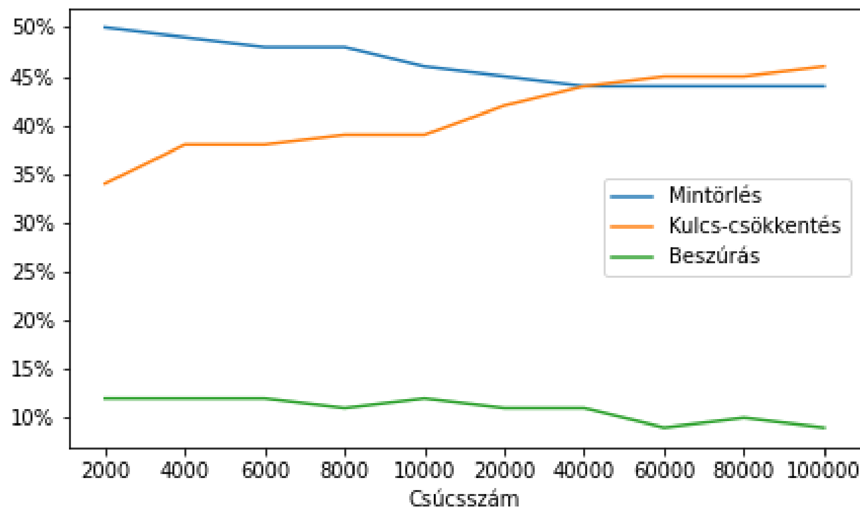


2.20. ábra. r_3-kupacműveletek futásidejének aránya a vödrös kupachoz képest.

A radix-kupacok esetében ugyanazt a megállapítást tehetjük, mint korábban, a ritka gráfoknál; a futásidőt nem befolyásolta az, hogy az élsúlyokat felszoroztuk 100-zal. A 2.20 ábrán az látható, hogy az $m = n^{3/2}$ esetben leggyorsabb r-kupac műveletei, (ami a 3. javítás) hogyan aránylanak a vödrös kupac műveleteinek átlagos idejéhez, illetve a teljes futásidők aránya.

A Beszúrás kis csúcsszámnál mindkét kupac esetén ugyanannyi idő, és a Kulcs-csökkentés is csak 15%-kal lassabb a radix kupacban. Nagy csúcsszámnál a vödrös kupacnak ezen műveletei lassulni kezdenek, így az r_3-kupac ezekhez képest gyorsul. A Mintörlés az r_3-kupac esetén 100000 csúcsnál 27%-kal lassabb, mint 2000 csúcsnál, a vödrös kupacot használva pedig 67%-kal gyorsabb. Így ennek a műveletnek az átlagos futásideje nagy csúcsszám esetén jóval lassabb az előbbi kupacban.

Láthatjuk azt is, hogy a teljes futásidő 2000 és 4000 csúcsnál még a radix kupacban jobb, aztán 60000 csúcsig a vödrös kupac a gyorsabb. A 2.21 ábra segít megérteni, hogy miért, amelyen az látható, hogy a futásidőnek mekkora részét teszik ki az egyes műveletek az r_3-kupacot használva. A Kulcs-csökkentés csak nagyobb csúcsszámnál lesz hangsúlyosabb a Mintörlésnél, ahogyan a vödrös kupacnál is. Így az, hogy ez a művelet a radix kupacnál gyorsabb, elég ahhoz, hogy ellensúlyozza a Mintörlés lassulását.



2.21. ábra. Az r_3-kupacműveletek összidejének aránya a teljes futásidőhöz képest.

2.2. Worst-case gráfok

Mivel az 1. fejezetben az aszimptotikus elemzések során a legrosszabb esetet vizsgáltuk, érdemes olyan gráfokon is tesztelni, melyben a Dijkstra-algoritmus maximális számú Kulcs-csökkentés műveletet végez, pontosan $m - n + 1$-et. Ehhez a LEDA programcsomagot bemutató könyvben [21] leírt módszert vettem alapul. Összefoglalva:

- Az $(i, i+1),\ 0 \leq i < n-1$ élek költsége $c := 100$. Ez tetszőleges más érték is lehetne.
- A maradék $m-(n-1)$ él a $(0,2),(0,3),\ldots,(0,n-1),(1,3),(1,4),\ldots,(1,n-1),(2,4),\ldots$ élek lesznek. Ezek költsége $c_{i,j}$, ahol

$$c_{i,j} = \begin{cases} (n-i)c+1, & \text{ha ez az utolsó él} \\ c_{i,j+1}+1, & \text{ha } j < n-1 \\ c_{i+1,i+3}+c+1, & \text{ha } j = n-1 \end{cases}$$

- A fenti súlyozás használata mellett még azt is meg kell követelnünk, hogy az i. csúcs éllistájában szereplő élek sorrendje $(i, i+2), \ldots, (i, n-1), (i, i+1)$ legyen.

Az eredeti módszerben az utolsó élnek $c + 1$ költséget adnak, de ez csak akkor működik, ha $c = 0$. Ez a konfiguráció nem csak azt garantálja, hogy maximális számú Kulcs-csökkentést végzünk, hanem azt is, hogy egy Kulcs-csökkentéskor mindig a korábban legdrágább élből válik az új legolcsóbb él. Ugyanakkor az összes Beszúrás az algoritmus elején történik, hiszen a 0. csúcsból minden más csúcs látható, amennyiben $m > 2n - 2$.

2.2.1. cn élű gráfok

A fent bemutatott módszerrel 10 különböző csúcsszámú gráfot generáltam, $m = 10n$ élszámmal. A random gráfokhoz hasonlóan itt is egy bináris kupacot használtam a Dijkstra-algoritmus lefuttatásához, és a kupacműveleteket lementettem. Minden műveletsoron 100-szor futtattam a méréseket és ezek minimumát vettem. Az így kapott eredmények a 2.4 táblázatban találhatók, milliszekundumban mérve. A Radix-kupacnak továbbra is a leggyorsabb megvalósítása szerepel a táblázatban.

Mivel az élszám $O(n)$ ezért a Kulcs-csökkentések darabszáma is ennyi. Ennek ellenére látszik a táblázatból, hogy a legrosszabb esetben ennek a műveletnek a javítása még így is lényeges javulást eredményez.

2.2.2. Ritka gráfok

A cn élszámnál a gyakorlatban fontosabb esetek a ritka, azaz $m = n \log n$ élszámú gráfok alapján generált műveletsorozatok, tekintsük most azokat. A mérési eredmények a 2.5 táblázatban szerepelnek. A Radix kupacnak a leggyorsabb verziója szerepel.

n	Bináris	$d = 4$	$d = 10$	Párosítós	Fibonacci	Vödrös	Radix (javítás nélkül)
2000	5,10	3,47	2,90	1,62	2,63	3,12	1,08
4000	11,32	7,44	6,24	3,27	5,35	6,25	2,19
6000	18,01	11,93	9,62	4,94	8,13	9,33	3,31
8000	24,48	15,97	12,92	6,58	10,82	12,52	4,50
10000	31,71	20,59	16,69	8,45	13,67	16,05	6,30
20000	69,98	43,58	36,61	16,84	28,15	32,22	12,84
40000	150,17	94,33	79,58	34,65	57,45	64,71	25,86
60000	238,67	147,41	123,17	50,91	85,88	97,39	35,32
80000	319,71	200,25	166,75	69,68	117,40	143,40	51,94
100000	409,59	253,68	210,60	86,00	145,30	192,37	58,31

2.4. táblázat. A szimulációk futásideje milliszekundumban mérve, $m = 10n$.

n	Bináris	$d = 4$	$d = \log n$	Párosítós	Fibonacci	Vödrös	Radix (javítás nélkül)
2000	5,77	3,74	3,08	1,93	2,96	3,15	1,02
4000	13,27	8,96	7,25	4,07	6,23	6,51	2,12
6000	21,84	13,94	11,61	6,45	9,85	10,07	3,26
8000	31,18	19,69	16,07	9,00	13,59	13,68	4,52
10000	40,94	26,33	21,02	11,54	17,37	17,27	6,41
20000	95,32	59,52	46,58	24,87	38,00	36,26	14,07
40000	219,02	135,45	101,18	54,26	80,96	74,53	29,46
60000	354,41	216,47	160,29	84,28	127,63	114,01	41,17
80000	495,70	301,84	223,61	117,55	178,40	157,01	63,80
100000	645,16	390,59	295,23	148,81	223,33	197,56	73,34

2.5. táblázat. A szimulációk futásideje milliszekundumban mérve, $m = n \log n$.

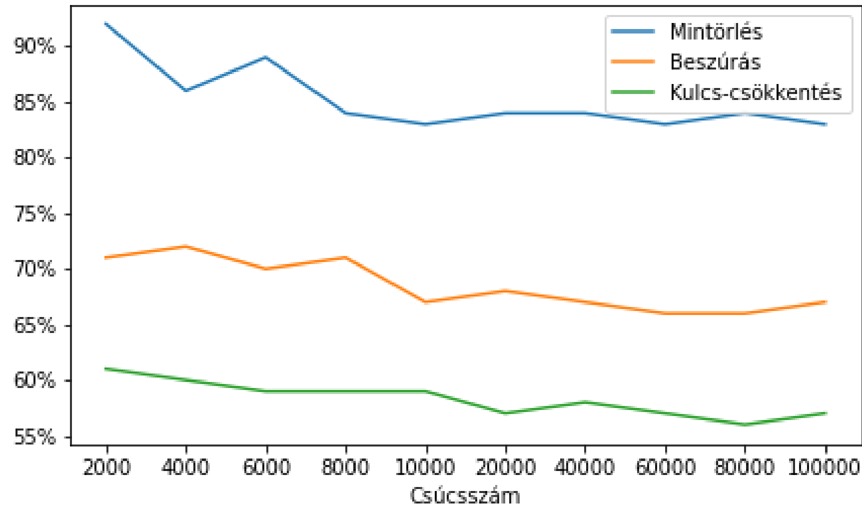
d-edfokú kupacok

Az elemzésnek megfelelően a bináris kupac a leglassabb. Mivel most minden Kulcs-csökkentésnél a legnagyobb kulcsból lesz a legkisebb, ezért minden Kulcs-csökkentés alkalmával a Felbillegtetések darabszáma is $\log_d$(kupac aktuális mérete).

A 2.22 ábrán látható a bináris és negyedfokú kupacműveletek összidejének aránya. A Mintörlés során elméletben ugyanannyi lépést tesz az algoritmus, a gyakorlatban a negyedfokú mégis gyorsabb. Ez azért van, mert Lebillegtetéskor n-szer két elemből kiválasztani a minimumot lassabb, mint $n/2$-ször négy elemből. A processzor a cache hatékony kihasználásával, gyorsabban tud egymás utáni tömbelemekkel dolgozni, mint több, egymástól a tömbben távol elhelyezkedő elemmel. Ez a cache locality tulajdonság, aminek részletes tárgyalása a d-edfokú kupacok vonatkozásában megtalálható a [22] cikkben.

A Beszúrás és Kulcs-csökkentés műveleteknél feleannyi összehasonlítás történik a Fel-

billegtetések során, de van $O(1)$ lépés, amit mindkettőben el kell végezni, így a gyakorlatban elért gyorsulás nem kétszeres.



2.22. ábra. Negyedfokú és bináris kupacműveletek összidejének aránya.

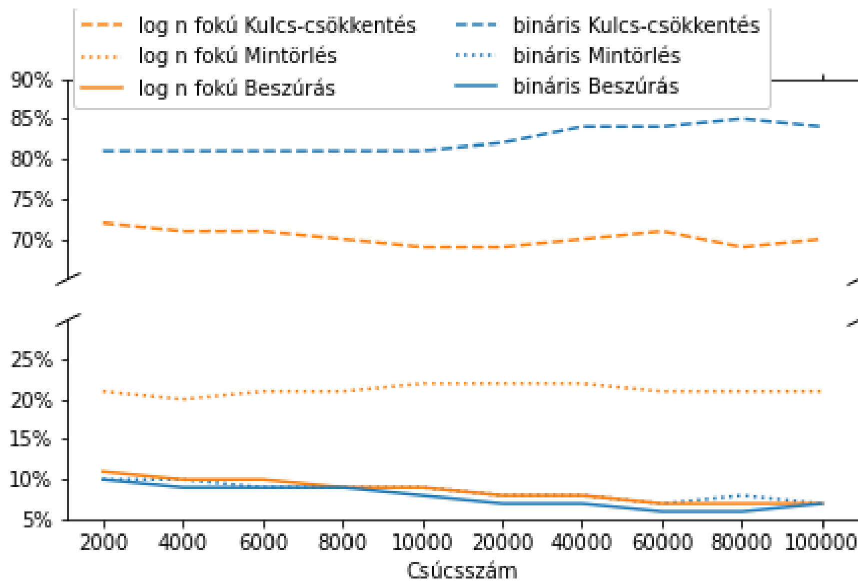
A random generált gráfoknál látottakkal ellentétben a d-edfokú kupacok közül az elméletben is optimális $\log n$ fokú a leggyorsabb. a 2.23 ábrán a bináris és $\log n$ fokú kupacok műveleteinek aránya látható az összidejükhöz képest. Megfigyelhetjük, hogy még a $\log n$-fokú kupacnál is a futásidő nagyrészét a Kulcs-csökkentés teszi ki, de ebben az esetben már a Mintörlés sem elhanyagolható idő.

Fibonacci- és párosítós kupac

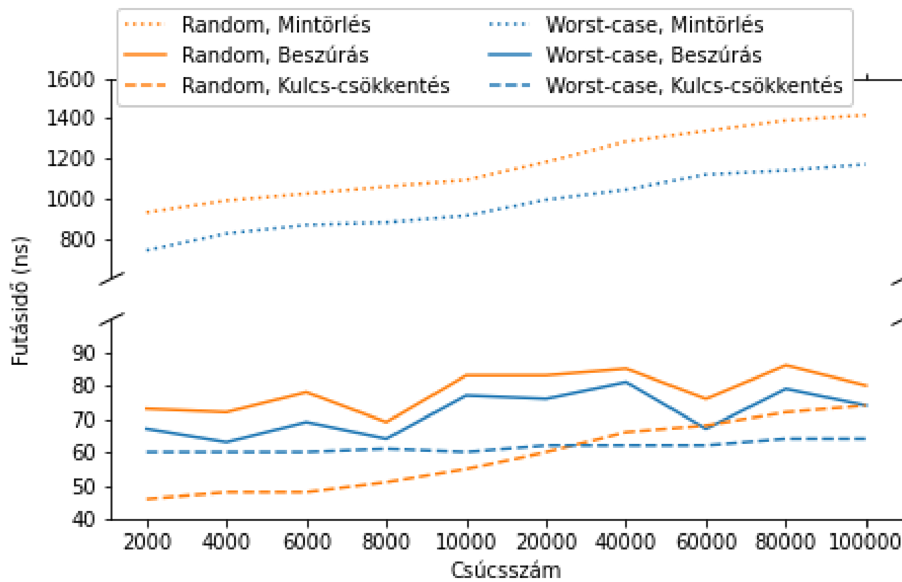
A párosítós és a Fibonacci-kupacban a Kulcs-csökkentés művelet futásidején javítottunk a Mintörlés kárára. Ez azért fontos, mert a worst-case gráfokból generált teszteken a bináris kupacot használva a futásidő $80-85\%$-át töltjük Kulcs-csökkentéssel, így ennek a javítása nagy hatással van a kupac teljesítményére.

A Fibonacci-kupacot használva a Kulcs-csökkentések során lentről felfelé haladva az összes csúcsot kivágjuk a fákból, és egyesével a gyökér listához fűzzük, így ennek a Műveletnek az átlagos futásideje független a csúcsszámtól. Emiatt a Mintörléskor is csak eltávolítunk a gyökér-listából egy elemet, és egycsúcsú fákat egyesítünk. Így ez a művelet gyorsabb a worst-case gráfok esetén, mint a random gráfoknál. A műveletek futásideje a 2.24 ábrán látható.

A Worst-case gráfoknál 2000 csúcsnál a futásidőnek 40%-át, 100000 csúcsnál pedig 44%-át teszik ki a Kulcs-csökkentések, a Mintörlések pedig csúcsszámtól függetlenül 36%-át. A 2.1 táblázattal összevetve az eredményeket azt is láthatjuk, hogy például $n = 100000$



2.23. ábra. Kupacműveletek aránya az összidőhöz képest, $d = \log n$ és $d = 2$ esetén.



2.24. ábra. Fibonacci-kupac műveleteinek átlagos ideje random és worst-case gráfokban, $m = n \log n$.

csúcsnál $5,7$-szer annyi Kulcs-csökkentés történt, mint a véletlen generált gráfoknál, a Mintörlés gyorsulása miatt a teljes futásidő mégis csak 34%-kal több.

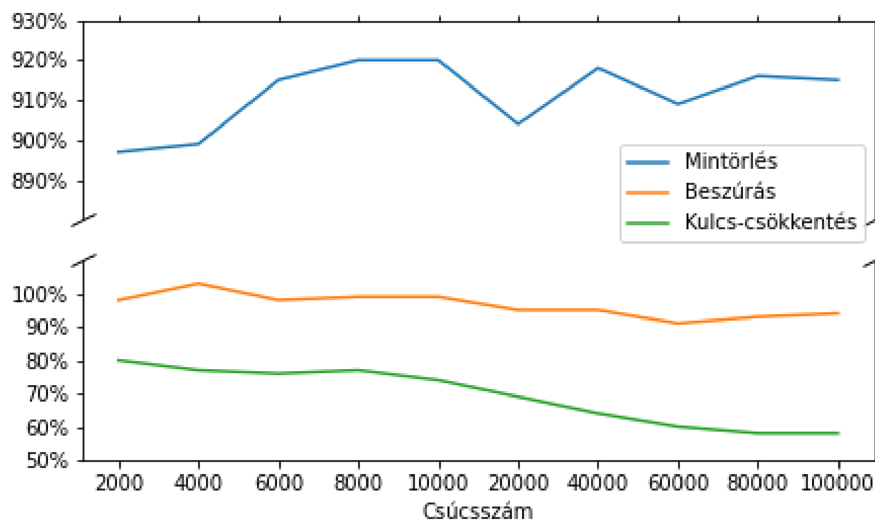
A párosítós kupacban is hasonló a helyzet, a Mintörlés kezdetekor a segédterületen csak egy csúcsú fák vannak. Ekkor a kupac gyökerének a kulcsa a legkisebb, mert mindig

a Legnagyobb kulcsot csökkentettük úgy, hogy a legkisebb legyen, és miután minden csúcs a segédterületre került, a gyökeret is Kulcs-csökkentjük. A segédterületen található fák Linkelése után az így kapott fát a gyökér alá kötjük (hiszen a gyökér kulcsa kisebb, mint a kapott fa gyökerének a kulcsa), majd a gyökeret töröljük.

Mivel a segédterület Linkelésével kapott fa gyökerének nincs jobb gyereke, ezért a kupac gyökerének törlése után nem lesz szükség további Linkelésre. A random gráfnál egy Mintörlés átlagos ideje a csúcsszám növekedésével 576 ns-ról 871 ns-ra romlik, a worst-case gráfnál pedig 344 ns-ról 540 ns-ra.

Vödrös kupac

A vödrös kupac most csak a d-edfokú és Fibonacci-kupacoknál gyorsabb. A 2.25 ábrán a kupacműveletek átlagos futásidejét hasonlítottam össze a random generált gráfokból készült szimulációkon mért időkkel ($m = n \log n, C = 2n$). A Beszúrás művelet nagyjából ugyanannyi idő a random és a worst-case gráfoknál is. Ez elsőre meglepő annak fényében, hogy amikor a maximális élsúly $C = 200n$ volt, a Beszúrás nagy csúcsszámnál lassabb volt (2.10 ábra). Most a maximális élsúly nagyjából $110n$, mégsem lassul ez a művelet. Ennek oka, hogy most az összes Beszúrás az algoritmus futásának az elején történik, egymás után, ráadásul egymást követő vödrökbe szúrjuk be az elemeket. Így a [22] cikkben is tárgyalt cache locality tulajdonság miatt nem okoz lassulást a tömb nagy mérete.



2.25. ábra. A vödrös kupacműveletek átlagos futásidejének aránya a random gráfhoz viszonyítva.

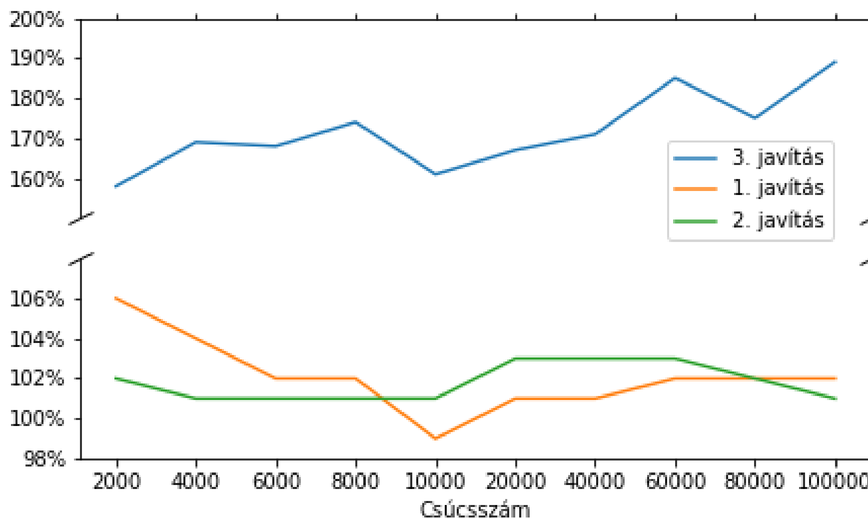
Egy Kulcs-csökkentés művelet átlagos futásideje már kis csúcsszámnál is kevesebb, mint a random gráfoknál, de ahogyan a csúcsszámot növeljük, a különbség is egyre nagyobb

lesz. Ennek oka, hogy a Kulcs-csökkentések során most mindig egyelemű vödörből veszük ki, és üres vödörbe rakunk be elemet, így mindig a lehető legkevesebb adattagot kell módosítanunk. A worst-case gráfoknál a Kulcs-csökkentések ideje nem is nő, ahogyan a csúcsszámot növeljük, mindig $45ns$ marad.

A Mintörlés során most minden alkalommal 100 vödröt kell előre lépni, hogy megtaláljuk az első nem üreset. Ez jóval több, mint a random esetben, így nem meglepő, hogy egy Mintörlés átlagos futásideje ennyivel lassabb most, mint a random gráfoknál.

r-kupac és javításai

Az r_0-kupac teljesített a legjobban ezen a gráftípuson. Az eredeti és az első két javítás között a korábbi tapasztalatoknak megfelelően alig volt különbség, ez a 2.26 ábrán is látszik. A javítás nélküli implementációban $n = 2000$ csúcsnál a futásidő 45%-át teszi ki a Mintörlés és a Kulcs-csökkentés is, ezek az arányok $n = 100000$ csúcs esetén már 35%-ra, és 50%-ra módosulnak.



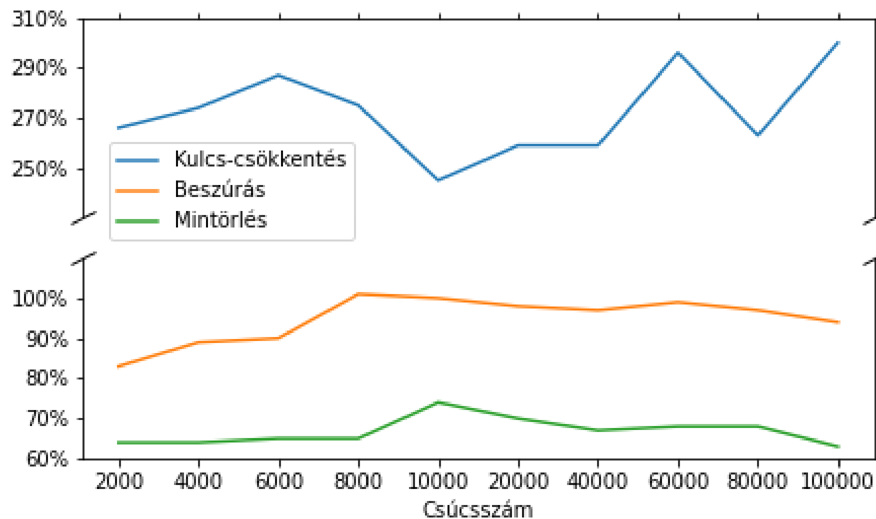
2.26. ábra. Az r-kupac javításainak összideje az eredetihez képest.

Az első javítás, amivel a Beszúrás lépésszáma legrosszabb esetben is $O(\log n)$ most nem javíthat, hiszen az elején beszúrunk minden csúcsot nagy kulccsal, ez a javítás pedig azon alapult, hogy ha kis sorszámú vödörbe való egy újonnan beszúrt kulcs, azt gyorsabban megtaláljuk.

A második javítás elméletben lehet gyorsabb, hiszen ebben a Mintörlés során a nemüres vödör keresésének a lépésszámát javítottuk $O(1)$-re. Mivel most minden Mintörléskor a következő legkisebb elem kulcsa 100-zal nagyobb, mint a legutóbb törölt elemé, ezért minden alkalommal 6 vödröt kell megvizsgálni, amíg megtaláljuk a következő nem üreset. Ez a gyakorlatban egy Mintörlés átlagos idejében nagyjából 5% javulást jelent az eredeti

megvalósításhoz képest. Emellett az első nem üres vödör $O(1)$-ben történő megkereséséhez a másik két művelet során is egy-egy extra műveletet kell elvégezni, ami a Beszúrást átlagosan 1%-kal, a Kulcs-csökkentést pedig 2%-kal lassítja. Fontos megjegyezni, hogy ezek a számok olyan kicsik (nem több, mint 5%), hogy akár a mérések természetes ingadozásából is fakadhatnak.

A harmadik javítás jóval lassabb, hiszen ahogy korábban is láthattuk, a Beszúrás és Mintörlés műveletek gyorsabbak lettek, de a Kulcs-csökkentés kárára. Az itt mért eredmények a 2.27 ábrán szerepelnek.



2.27. ábra. Az r-kupac harmadik javítása az eredetihez képest.

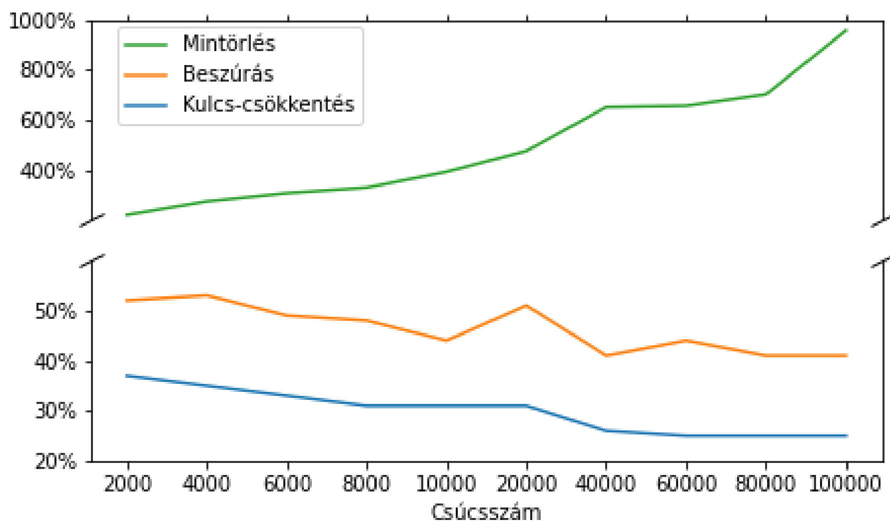
2.2.3. Középsűrű gráfok

Tekintsük most ismét a worst-case teszteseteket, de $m = n^{3/2}$ élszámmal. A különböző kupacok használatával mért futásidőket a 2.6 táblázat tartalmazza.

n	Bináris	$d = 4$	$d = \sqrt{n}$	Párosítós	Fibonacci	Vödrös	Radix (javítás nélkül)
2000	21,38	13,57	8,69	7,76	10,41	6,42	2,59
4000	66,86	40,33	25,73	22,16	28,82	16,58	6,73
6000	128,84	76,81	46,35	40,82	53,20	29,08	11,97
8000	203,59	124,68	70,37	63,38	83,06	44,09	18,37
10000	293,42	177,68	102,48	89,46	117,12	60,50	24,74
20000	885,22	526,53	316,95	257,47	330,68	163,66	67,38
40000	2691,79	1586,61	817,06	732,03	933,30	450,09	185,77
60000	5109,10	3001,20	1470,14	1345,54	1715,54	813,57	339,11
80000	8103,58	4758,69	2221,14	2077,97	2644,46	1242,80	518,14
100000	11673,71	6731,48	3213,93	2910,44	3751,52	1728,66	713,65

2.6. táblázat. A szimulációk futásideje milliszekundumban mérve, $m = n \log n$.

d-edfokú kupacok



2.28. ábra. $\sqrt{n}$ fokú és Bináris kupacok műveleteinek aránya.

A bináris kupacban a futásidőnek 96%-át, míg a negyedfokúban a 93%-át teszi ki a Kulcs-csökkentés. A d-edfokú kupacok közül optimális $\sqrt{n}$ fokú kupacban, amiben ezt és

a Beszúrást gyorsítottuk már csak 86%-át. A $\sqrt{n}$ fokú és Bináris kupacműveletek összidejének aránya a 2.28 ábrán látható.

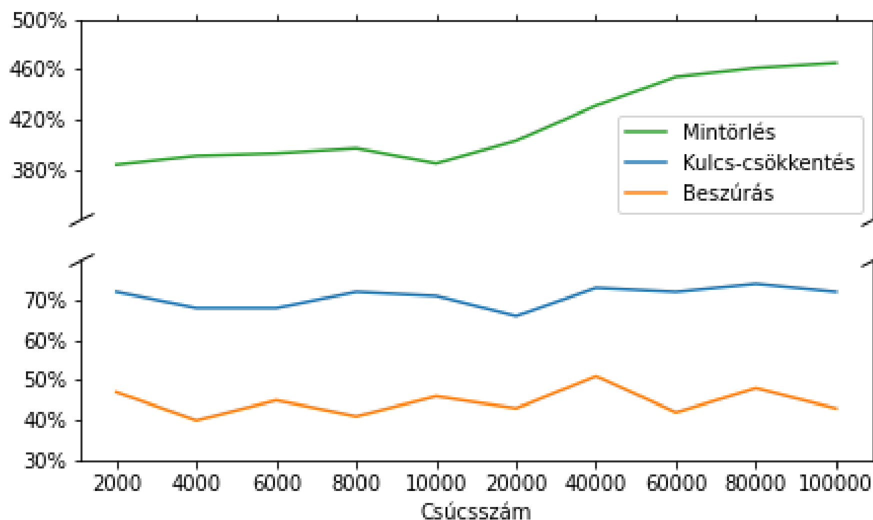
$\sqrt{n}$ fokú kupacot használva 2000 csúcsnál $2,7$-szer gyorsabb a Kulcs-csökkentés, mint a bináris kupacban, 100000 csúcsnál pedig 4-szer. A fent említettek szerint ez adja a futásidő nagyrészét, ezért például 100000 csúcsnál a $\sqrt{n}$ fokú kupac $3,6$-szor olyan gyors, mint a bináris. A mérésekből az is kiderült, hogy egy Kulcs-csökkentés átlagos futásideje a $\sqrt{n}$ fokú kupacot használva a gyakorlatban $88ns$. Ez nem meglepő, hiszen a kupac mélysége három, 2000 és 100000 csúcsnál is.

Éppen ezért a Beszúrás ideje is konstansnak mondható, de itt a az átlagos futásidők ingadozása kicsit nagyobb volt a csúcsszám változásával, a legnagyobb és legkisebb mért eredmény közötti különbség 10%.

A Mintörlés művelet jóval lassabb a $\sqrt{n}$ fokú kupacban, hiszen itt minden szinten $\sqrt{n}$ elemből kell kiválasztani a minimumot, és nem csak kettőből. Bináris kupacot használva 2000 csúcsnál a futásidőnek 3%-át tette ki a Mintörlés, 100000 csúcsnál pedig kevesebb, mint 1%-át. $\sqrt{n}$ fokú kupacot használva ezek a számok 15% és 12%, ami már nem elhanyagolható, de a Kulcs-csökkentés gyorsításával lényegesen többet nyerünk, így nem baj ha a Mintörlés ennyivel lassabb.

Fibonacci- és párosítós kupac

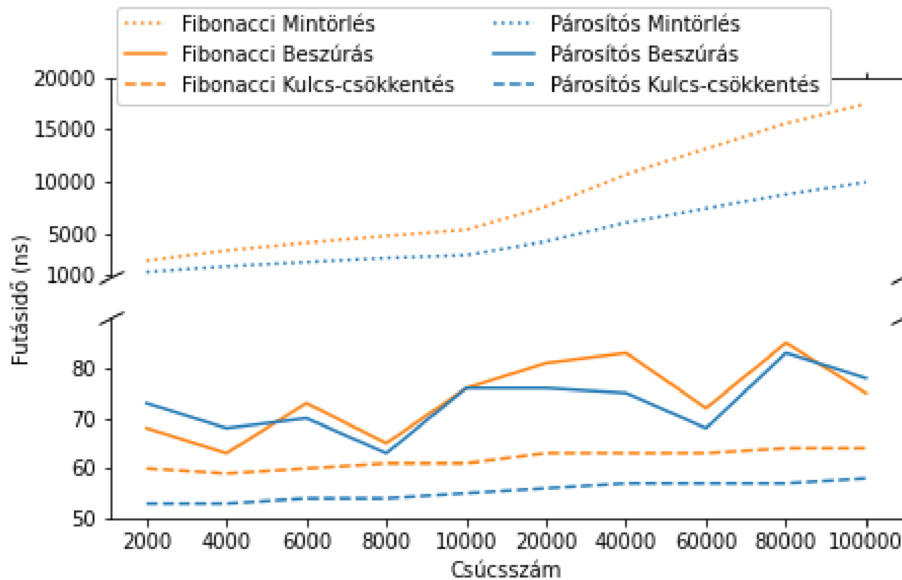
A Fibonacci-kupac most is gyorsabb a másod- és negyedfokú kupacoknál, akárcsak a ritka gráfoknál, de a $\sqrt{n}$ fokú gyorsabb nála. Hasonlítsuk hát össze ezzel. A 2.29 ábrán látható a Fibonacci-kupac műveleteinek futásidejének aránya a $\sqrt{n}$ fokú kupachoz képest.



2.29. ábra. Fibonacci és $\sqrt{n}$ fokú kupacok műveleteinek aránya.

A Kulcs-csökkentés és Beszúrás műveletek aránya konstans volt, mert ezeknek a mű-

veleteknek az átlagos ideje a Fibonacci-kupacban is csúcsszámtól független. A Mintörlés esetében láthatjuk, hogy ez a kupac jóval rosszabbul teljesített, mint a $\sqrt{n}$ fokú. Mivel a Fibonacci-kupacban a futásidő 46%-át teszi ki a Mintörlés, és $50 - 54$%-át a Kulcs-csökkentés, ezért az, hogy a Mintörlés ilyen lassú (2.30 ábra), azt eredményezi, hogy még a $\sqrt{n}$ fokú kupac is gyorsabb a Fibonacci-kupacnál.



2.30. ábra. A párosítós és a Fibonacci-kupac műveleteinek átlagos futásideje.

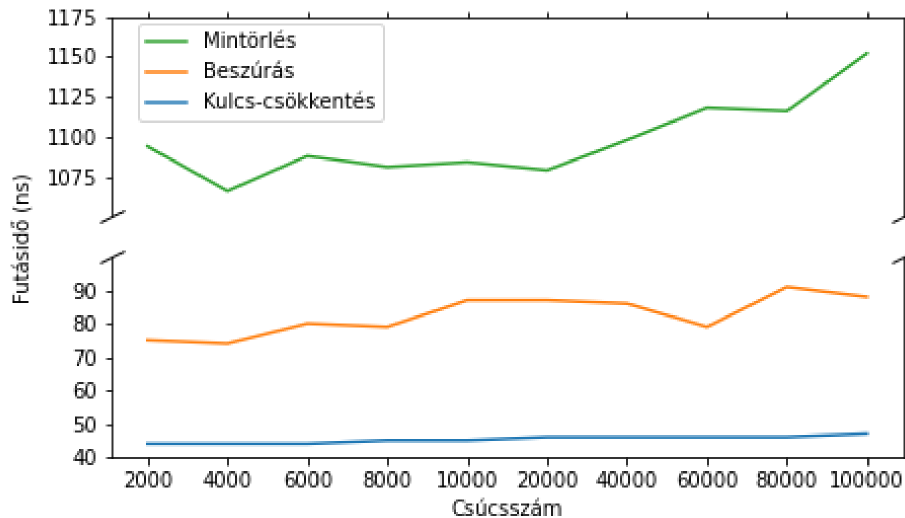
Hasonlítsuk össze ismét a Fibonacci- és a párosítós kupacot. A kupacműveletek átlagos ideje a 2.30 ábrán látható mindkét kupac esetében. A Beszúrások ideje szinte megegyezik a két kupacban, ahogyan ezt vártuk. A Kulcs-csökkentés a Fibonacci-kupacot használva kicsit lassabb, mint a párosítós kupacban, nem úgy, mint a véletlen generált gráfok esetében. Ennek oka, hogy a Párosítós kupacban kettővel kevesebb adattagot kell karbantartani az kupacban tárolt elemekhez. Az utolsó művelet, a Mintörlés majdnem kétszer olyan gyors a párosítós kupacban, mint a Fibonacciban, emiatt a teljes futásidő is 25%-kal gyorsabb az előbbiben. Így ezen a műveletsorozaton a párosítós kupac a leggyorsabb, tetszőleges kulcs értékekhez használható kupac.

Kulcsmanipulációs kupacok

A gráfokban, amelyből a teszteléshez használt műveletsort előállítottuk, a maximális élsúlyok szinte megegyeznek az $m = n \log n$ élszámú gráfoknál kapottakkal, átlagosan $113n$. A vödrös kupac most ismét kiemelkedően teljesített, a párosítós kupacnál is jobb futásidőket ért el. A három kupacművelet átlagos futásideje megegyezik a 2.2.2 fejezetben tárgyalt, ritka gráfoknál mért időkkel. Ezek a 2.31 ábrán láthatók.

Mivel most több Kulcs-csökkentés történik, mint a ritka gráfból generált műveleteknél, ezért itt többet nyerünk a vödrös kupac használatával, hiszen ebben a leggyorsabb ez

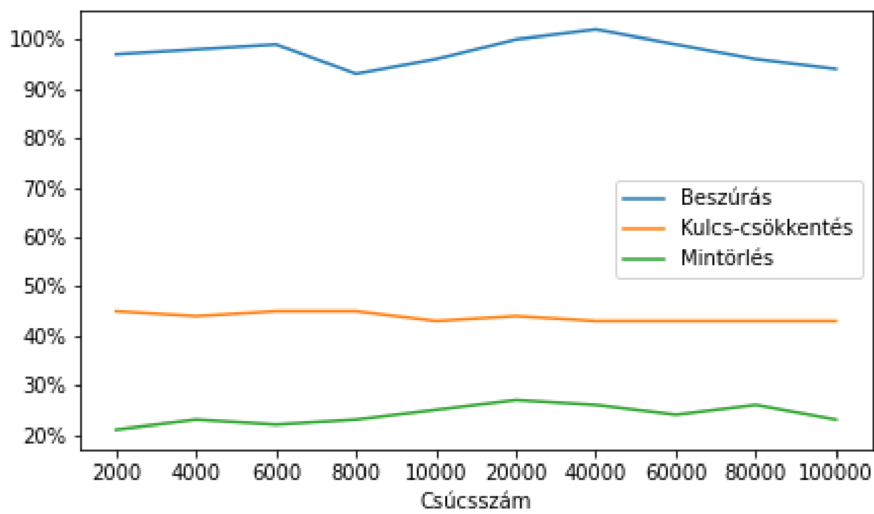
a művelet. Ez önmagában még nem eredményezi, hogy $n = 100000$ csúcsnál $1,68$-szor gyorsabb legyen, mint a párosítós kupac. Szükség van hozzá arra is, hogy a Mintörlés töredéke a 2.30 ábrán látható futásidőnek.



2.31. ábra. Vödrös kupacműveletek átlagos ideje.

Az r_0-kupac teljesített a legjobban ezen a tesztelésen. Az első két javításnál a futásidők ismét $2-3\%$-kal lassabbak. Az r_3-kupac ismét jóval lassabb, hiszen a Kulcs-csökkentés művelet több ideig tart benne, mint a javítás nélküli verzióban. $n = 2000$ csúcs esetén az r_0-kupac teljes futásideje 40%-a az r_3-kupac idejének, $n = 100000$ csúcsszám esetén pedig 31%-a.

Az utolsó, a 2.32 ábrán az látható, hogyan aránylik a kupacműveletek futásideje az r_0-kupacot használva a vödrös kupacban mértekhez képest. Egy Beszúrás ideje ugyanúgy $80ns$ mindkét kupacban, a Kulcs-csökkentés pedig kevesebb, mint a vödrös kupacban mért futásidők fele, $20ns$. A Mintörlés futásideje az r_0-kupacban 2000 csúcs esetén $229ns$, 100000 csúcsnál pedig $262ns$.



2.32. ábra. Kupacműveletek futásidejének aránya az r_0-kupacot használva a vödrös kupachoz képest.

3. fejezet

Összegzés

Szakdolgozatom zárásaként először összefoglalom, hogy a 2. fejezetben milyen gráfokat vizsgáltam, és hogy ezeken melyik kupac használatával értem el a legjobb futásidőket. A második részben ezek alapján javaslatot adok arra, hogy ha adott egy gráf, amin Dijkstra-algoritmust szeretnénk futtatni, akkor a csúcsszám, élszám és élsúlyok ismeretében melyik kupacot használnám.

3.1. Mérési eredmények

A 2. fejezetben kettő fő gráftípust vizsgáltam. Az első az Erdős–Rényi módszerrel véletlenszerűen generált gráfok, melyben a lehetséges élek közül minden él ugyanakkora valószínűséggel van benne a gráfban. A második pedig a d-edfokú kupacokra worst-case gráfok. Itt maximális számú Kulcs-csökkentés történik, és ezek olyanok, hogy mindig a legnagyobb kulcsból lesz a legkisebb. Mindkettő gráftípust $m = n \log n$ és $m = n^{3/2}$ élszámmal is teszteltem, különböző élsúlyokra.

Random gráfok

Először tekintsük azt az esetet, amikor az élsúlyok egész számok, egy $[1, C]$ intervallumból egyenletes eloszlással súlyozva. Ekkor, ha a maximális élsúly nem túl nagy, azaz nagyságrendileg n-nel egyezik meg, akkor a vödrös kupac használatával értük el a legjobb futásidőt.

Ha azonban az élsúlyokat a $[100, 200n]$ intervallumból választjuk, 100-as lépésközzel, akkor már a negyedfokú kupac a leggyorsabb ritka, és sűrű gráfok esetében is. Ebből következik, hogy nem egész élsúlyoknál is a negyedfokú kupacot érdemes választani.

Worst-case gráfok

Ezeknél a gráfoknál, ha az élsúlyok egészek, az r_0-kupac a legjobb választás. Amennyiben az élsúlyok tetszőleges (nemnegatív) számok lehetnek, a párosítós kupacot használva érhetjük el a legjobb eredményeket mind a ritka, mind a sűrű gráfok esetén.

3.2. Milyen kupacot használjunk

Most a feladat az, hogy adott egy gráf, amin Dijkstra algoritmusát szeretnénk futtatni, amihez a legjobb kupacot szeretnénk kiválasztani. Ha az élsúlyok nem egész számok, akkor szerintem a negyedfokú, és a párosítós kupac közül érdemes választani, hiszen az elvégzett tesztek során minden alkalommal e kettő közül volt valamelyik a leggyorsabb.

Amennyiben az élsúlyok egész számok, és a legnagyobb élsúly nagyságrendileg megegyezik a csúcsszámmal, akkor a vödrös kupac lehet a jó választás. Ha ennél nagyobb élsúlyokkal dolgozunk, akkor vagy az r_0-, vagy a negyedfokú kupac.

Hogyan válasszunk két kupac közül

Mindkét élsúlyozás esetében két választási lehetőségre szűkítettük a lehetséges kupacokat, amiből az egyik a negyedfokú kupac. A d-edfokú kupacok közül azért ezt javaslom, mert ennél nagyobb fokú kupac csak akkor volt gyorsabb, amikor úgy konstruáltuk meg a gráfot, hogy biztosan az legyen. A negyedfokú kupac akkor hatékony, ha viszonylag kevés Kulcs-csökkentés történik, és a Kulcs-csökkentett elemeket nem kell sok szintet Felbillegtetni. Ám ezekre csupán a gráf csúcsszáma és élszáma alapján nincs rálátásunk.

Megtehetjük, hogy párhuzamosan, vagy felváltva futtatunk két Dijkstra-algoritmust, amelyek különböző kupacot használnak, és figyeljük, hogy melyik a gyorsabb. Ezt elérhetjük úgy, hogy megadunk egy N küszöbszámot, és mindkettőben számoljuk, hogy hány csúcs van véglegesítve. Amikor az egyik algoritmusban a véglegesített csúcsok száma eléri ezt a küszöbszámot, akkor az általa használt kupacot választjuk, és a másik algoritmus futását megszakítjuk. Ez a küszöbszám lehet például $N = \sqrt{n}$.

Élsúlyok felskálázása

Lehet, hogy az élsúlyok nem egész számok, de megpróbálhatjuk őket azzá alakítani. Ha az élsúlyok nem túl nagyok, és csak néhány tizedesjegy pontossággal vannak megadva, akkor azokat egy elég nagy számmal felszorozva már egész számokat kapunk. Az így kapott gráfon futtathatunk akár radix-kupacot is.

Legyen adott egy G gráf, és egy c élsúlyozás. Legyen K olyan, hogy minden $c(e)$ élsúlyra $K \cdot c(e)$ egész szám, az így kapott élsúlyozást pedig jelölje c'. Ekkor a Dijkstra-algoritmus futtatásával kapott s gyökerű legrövidebb utak feszítőfája ugyanaz mindkét élsúlyozás esetén. Ebben a fában egy csúcs c' szerinti távolsága s-től megegyezik a c szerinti távolság K-szorosával. Így ha c' súlyozással meg tudjuk oldani gyorsan a feladatot, akkor abból azonnal megkapjuk a c súlyozás szerinti legrövidebb utakat is.

Ehhez persze a Dijkstra-algoritmus futásidejéhez képest elhanyagolható időben meg kell tudnunk határozni K-t. Amennyiben az élsúlyozásról rendelkezünk ilyen előismerettel, ezzel a módszerrel tovább gyorsíthatjuk az algoritmus futását.

Hivatkozások

1. Dijkstra, E. W. A Note on Two Problems in Connexion with Graphs. *Numer. Math.* 1. köt., 269–271. old. ISSN: 0029-599X. https://doi.org/10.1007/BF01386390 (1959. dec.).
2. Prim, R. C. Shortest connection networks and some generalizations. *The Bell System Technical Journal* 36. köt., 1389–1401. old. (1957).
3. Huffman, D. A. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* 40. köt., 1098–1101. old. (1952).
4. Williams, J. W. J. Algorithm 232: Heapsort. *Communications of the ACM* 7. köt., 347–348. old. (1964).
5. Duchaineau, M. *és tsai.* ROAMing terrain: Real-time Optimally Adapting Meshes. *Proceedings IEEE Visualization '97,* 81–88. old. (1997. nov.).
6. Tarjan, R. E. Amortized Computational Complexity. *SIAM Journal on Algebraic Discrete Methods* 6. köt., 306–318. old. https://doi.org/10.1137/0606031 (1985).
7. Király, Z. *Adatstruktúrák* https://zkiraly.web.elte.hu//Adatstrukturak.pdf (2022).
8. Fredman, M. L. & Willard, D. E. *BLASTING through the Information Theoretic Barrier with FUSION TREES Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA* (szerk. Ortiz, H.) (ACM, 1990), 1–7. old. https://doi.org/10.1145/100216.100217.
9. Johnson, D. B. Priority queues with update and finding minimum spanning trees. *Information Processing Letters* 4. köt., 53–57. old. ISSN: 0020-0190. https://www.sciencedirect.com/science/article/pii/0020019075900010 (1975).
10. Dial, R. B. Algorithm 360: Shortest-Path Forest with Topological Ordering [H]. *Commun. ACM* 12. köt., 632–633. old. ISSN: 0001-0782. https://doi.org/10.1145/363269.363610 (1969. nov.).
11. Ahuja, R. K., Mehlhorn, K., Orlin, J. & Tarjan, R. E. Faster Algorithms for the Shortest Path Problem. *J. ACM* 37. köt., 213–223. old. ISSN: 0004-5411. https://doi.org/10.1145/77600.77615 (1990. ápr.).
12. Knuth, D. E. *The Art of Computer Programming: Combinatorial Algorithms, Part 1* 1st, 154. old. (Addison-Wesley Professional, 2011). ISBN: 0201038048.

13. Fredman, M. L. & Tarjan, R. E. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *J. ACM* 34. köt., 596–615. old. ISSN: 0004-5411. https://doi.org/10.1145/28869.28874 (1987. júl.).

14. Brodal, G. S., Lagogiannis, G. & Tarjan, R. E. *Strict Fibonacci Heaps Proceedings of the Forty-Fourth Annual ACM Symposium on Theory of Computing* (Association for Computing Machinery, New York, New York, USA, 2012), 1177–1184. old. ISBN: 9781450312455. https://doi.org/10.1145/2213977.2214082.

15. Fredman, M., Sedgewick, R., Sleator, D. & Tarjan, R. The Pairing Heap: A New Form of Self-Adjusting Heap. *Algorithmica* 1. köt., 111–129. old. (1986. nov.).

16. Stasko, J. T. & Vitter, J. S. Pairing Heaps: Experiments and Analysis. *Commun. ACM* 30. köt., 234–249. old. ISSN: 0001-0782. https://doi.org/10.1145/214748.214759 (1987. márc.).

17. Fredman, M. L. On the Efficiency of Pairing Heaps and Related Data Structures. *J. ACM* 46. köt., 473–501. old. ISSN: 0004-5411. https://doi.org/10.1145/320211.320214 (1999. júl.).

18. Haeupler, B., Sen, S. & Tarjan, R. E. Rank-Pairing Heaps. *SIAM Journal on Computing* 40. köt., 1463–1485. old. https://doi.org/10.1137/100785351 (2011).

19. Erdős, P. & Rényi, A. On Random Graphs I. *Publicationes Mathematicae Debrecen* 6. köt., 290. old. (1959).

20. Noshita, K. A theorem on the expected complexity of Dijkstra's shortest path algorithm. *Journal of Algorithms* 6. köt., 400–408. old. ISSN: 0196-6774. https://www.sciencedirect.com/science/article/pii/0196677485900094 (1985).

21. Mehlhorn, K. & Näher, S. *LEDA: A Platform for Combinatorial and Geometric Computing* 34–36. old. (Cambridge University Press, 1999. jan.). https://www.microsoft.com/en-us/research/publication/leda-a-platform-for-combinatorial-and-geometric-computing/.

22. Naor, D., Martel, C. U. & Matloff, N. S. Performance of Priority Queue Structures in a Virtual Memory Environment. *The Computer Journal* 34. köt., 428–437. old. ISSN: 0010-4620. https://doi.org/10.1093/comjnl/34.5.428 (1991. jan.).