Project Algoritmen en datastructuren 2, 2023

Sviatoslav Harasymchuk

December 3, 2023

Contents

Chapter 1

Intro

In Chapter 1 zal ik uitleg geven over het test- en benchmarkproces dat ik gedaan heb voor het bouwen van dit project en het versnellen van mijn algoritmen. In volgende secties volgt de uitleg voor elk van de geïmporteerde algoritmen. Sommige paragrafen uit dit verslag zijn door AI-tool (DeepL) behandeld om het aantal (spelling-)schrijffouten te verminderen.

1.1 Benchmarks

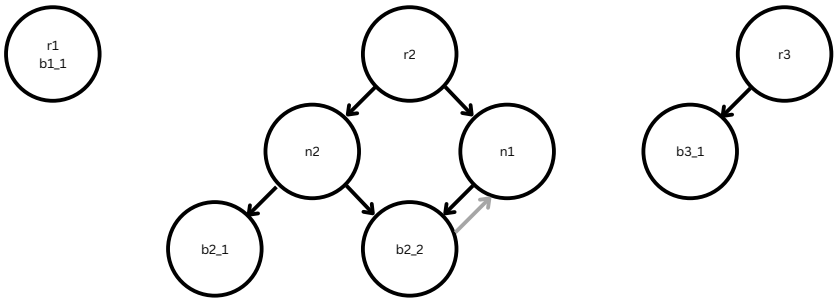
Mijn benchmarks omvatten het 'opwarmen' van de JVM voordat ze de gemiddelde uitvoeringstijden meten. Hierdoor is de standaardafwijking van de dataset lager. De Supplier en Consumer functies zijn bedoeld om specifieke bewerkingen te testen, zoals bijvoorbeeld decreaseKey op een niet-lege heap.

```
1  public <T> long[] sampleTimes(Supplier<T> supplier,
2                                Consumer<T> benchedCons) {
3    int warmupReps = 5;
4    int repetitions = 100;
5    long[] times = new long[repetitions];
6
7    // Warm up:
8    for (int i = 0; i < warmupReps; i++) {
9      benchedConsumer.accept(supplier.get());
10   }
11
12   // Actual run:
13   for (int i = 0; i < repetitions; i++) {
14     T value = supplier.get();
15     times[i] = timeExecution(()->benchedCons.accept(value));
16   }
17   return times;
18 }
```

1.2 Samenhangende grafen

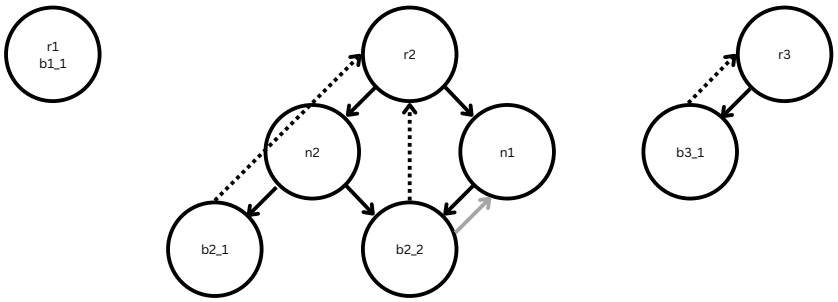
Om onze ShortestPath te testen en te benchmarken, kunnen we een algoritme ontwikkelen dat onze willekeurig gegenereerde graaf samenhangend maakt.

Begin door alle bogen van de graaf te volgen en subgrafen te creëren:

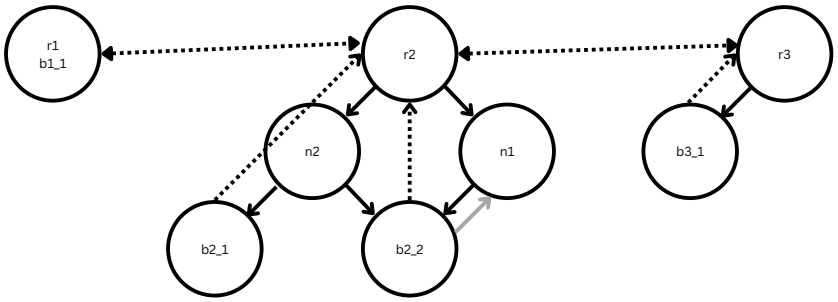


Zij r_i ($i \in \mathbb{N}$) wortel van subgraaf i en b_{ij} ($i \in \mathbb{N}$) blad j van de subgraaf i. Merk op dat de blad b_{22} bidirectioneel verbonden is met node n_1 en nog steeds als blad wordt beschouwd.

Verbind vervolgens alle bladeren van elke subgraaf i met hun wortel:



Blad b_{11} wordt niet met de wortel verbonden omdat die op zich zelf een wortel is. Daarna doorloop je de wortels r_i en maak je bidirectionele bogen (telkens 2 bogen) tussen elk paar wortels:



De resultaat is een samenhangende graaf waarin je uit elke top elke andere top kan bereiken.

Chapter 2

SkewHeap

De meest cruciale bewerking in elke heap is naar mijn mening de *merge*-bewerking. Mijn implementatie van de skew heap maakt gebruik van het algoritme beschreven in het **boek 4.3.1** (p.77). De skew heap zelf is geïmplementeerd met referenties naar twee kinderen (links en rechts), die op zichzelf van het type skew heap zijn. De gehele heap bestaat dus feitelijk uit een node met referenties naar de kinder-skew heaps (recursieve definitie).

```
1  public class SkewHeap<P extends Comparable<P>, V> implements
     PriorityQueue<P, V> {
2    private SkewHeap<P, V> left;
3    private SkewHeap<P, V> right;
4    ...
5  }
```

De decreaseKey wordt normaal opgeroepen op een object die volgende type implementeert: $QueueItem < P\ extends\ Comparable < P >, V >$. Daarom heb ik een implementatie *HeapItem* gemaakt die als veld *item* wordt opgeslagen in de skew heap. HeapItem moet weten welke node (skew deelheap) op de hoogte gebracht moet worden dat de key decreased is om de skew heap eigenschappen in heel de heap te behouden. Eerst had de HeapItem een referentie naar de SkewHeap object (node waarin die item zit) en riep die verder decreaseKey op op een skew heap object. Er was een probleem: de decreaseKey-functie binnen de SkewHeap-klasse moest public zijn, wat resulteerde in een minder goede encapsulatie van de SkewHeap-klasse. Later vond ik echter een betere oplossing om de heap te laten 'fixen' na een decreaseKey van een item: elk item heeft nu een verwijzing naar een *fixheap*-functie uit de SkewHeap-klasse.

```
1  private Runnable decreaseKeyFunction;
```

Als deze functie *null* is, betekent dit dat het item uit de heap is verwijderd. Nu kan de decreaseKey slechts op één specifieke plaats worden

opgeroepen: op het object van HeapItem.

De grootte van de heap kan op drie verschillende manieren worden bepaald:
- Door caching tijdens toevoegen/verwijderen
- Recursief: 1 + (size linker-heap) + (size rechter-heap)
- Recursief-iteratief: gebruik van $Stack < T >$ om recursie te vermijden

Laatste implementatie was een omweg om de $StackOverflowExceptions$ te vermijden voor te diepe (ongebalanceerde) skew heaps.

Om te testen of alle heap-eigenschappen altijd voldaan zijn heb een functie gemaakt die recursief nagaat of elke node beide/geen kinderen heeft, oftewel alleen linker kind.

2.1 Benchmarks

2.1.1 Add item

Het invoegen van een item in een skew heap is een bewerking die over het algemeen weinig kost, zoals te zien is in de benchmarkresultaten hieronder:

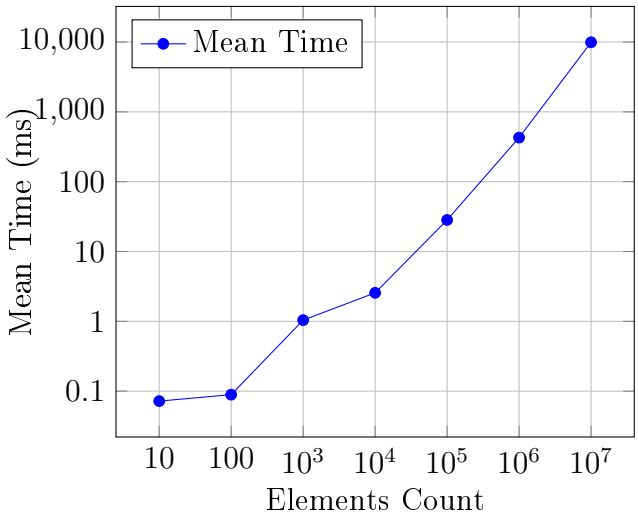


Figure 2.1: SkewHeap elementen toevoegen

Wanneer de invoergrootte wordt verhoogd met een factor van 10, neemt de uitvoeringstijd als volgt toe: (Zie tabel 2.1)

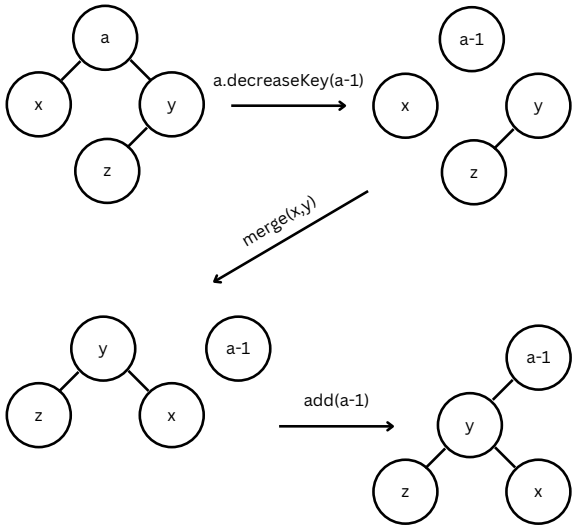
Die toename ziet er logaritmisch (per bewerking) uit, wat in $O(n \log n)$ resulteert op n reeks toevoegingen. Dat is ook te zien op een exponentieel grafiek die voor grote n linear is.

Vergroting van invoergrootte	Toename van uitvoeringstijd
1x naar 1000x	...
1.000x naar 10.000x	x2.5
10.000x naar 100.000x	x11.1
100.000x naar 1.000.000x	x15.1
1.000.000x naar 10.000.000x	x23.2
...	...

Table 2.1: Toename van uitvoeringstijd bij vergroting van de invoergrootte

2.1.2 Decrease key en poll

Tijdens testen merkte ik dat decrease key ook op de root van de heap opgeroepen kan worden, standaart algoritme zou dan de kinderen van de root mergen en dan de resulterende heap de linker kind van root maken. Zie volgende schets met keys: $z \geq x \geq y \geq a$.



Ik vond dat onnodig, vooral als de deelboom van x meerdere elementen zou bevatten. Daarom heb ik een statische variabele toegevoegd waarmee de gebruiker kan kiezen of er al dan niet moet worden samengevoegd voor de decreaseKey bewerking op de root.

```
1  public static boolean reMergeRoot = false;
```

Dan wou ik deze verbetering benchen door eerst 2 willekeurige elementen te decreasen en vervolgens een decreaseKey op de root uit te voeren, gevolgd door poll. Volgende grafieken tonen mijn benchmarks: deze methode is sneller, maar alleen voor kleine heaps (tot 100.000 elementen).

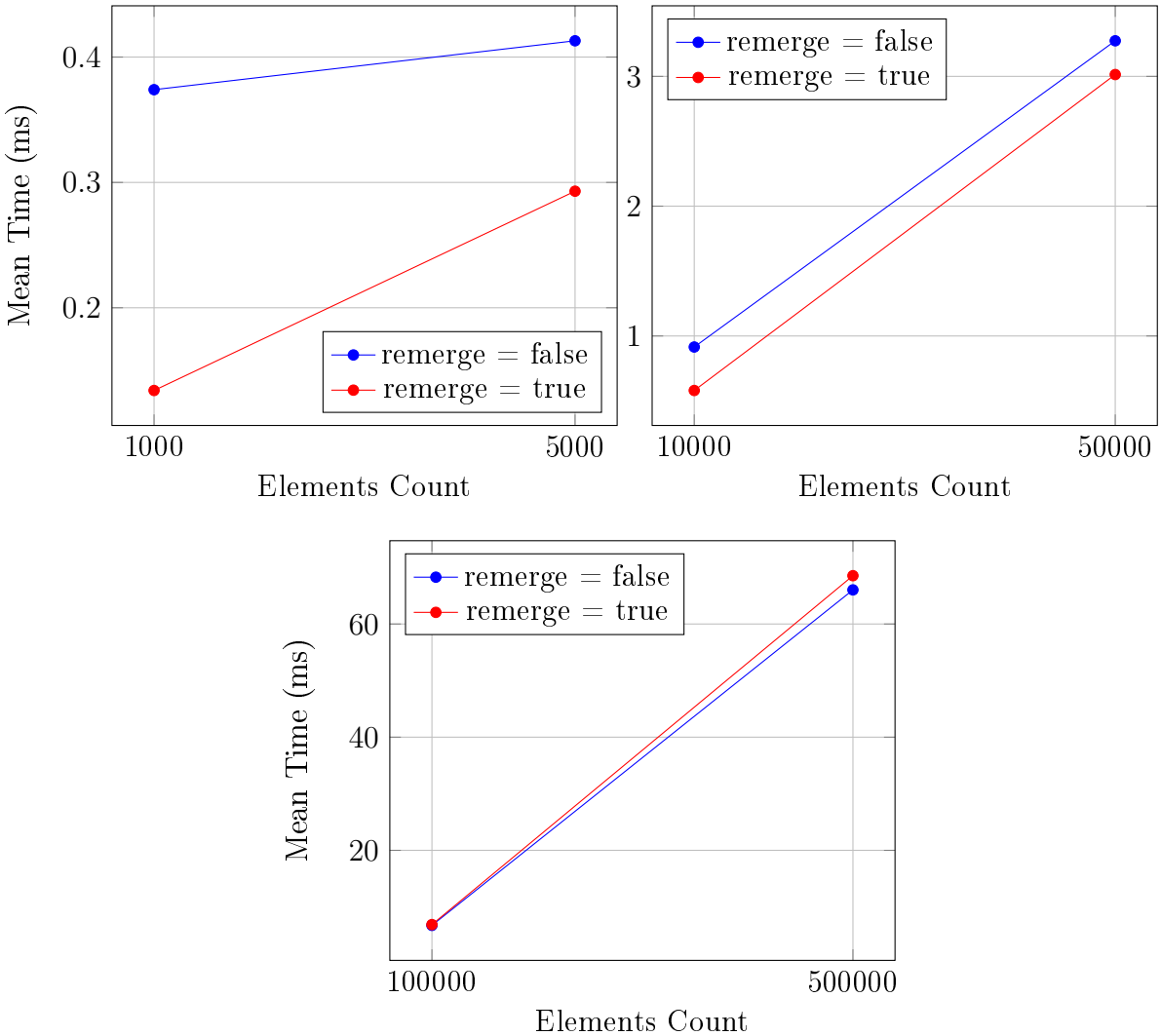


Figure 2.2: **2x** *random decreaseKey* + **1x** *decreaseKey op root met poll*

2.2 Complexiteit van Decrease key

Voor beide bewijzen zullen wij Stelling 16 uit het boek AD2 2023 gebruiken en ook de definities die daar werden geïntroduseerd.

Algoritme 1

> Decrease key op skew heap S:
>
> De deelboom S_v met als wortel v wordt verwijderd uit de heap. Dan wordt het pad van v naar de wortel overlopen. Als een top op dat pad v in zijn linkerdeelboom heeft en een rechterkind heeft, dan worden zijn kinderen gewisseld. Zo blijft het resultaat een skew heap. De wortel v wordt kleiner, en S_v wordt terug gemerged met de originele heap.

Propositie:
De geamortiseerde complexiteit van een reeks van n bewerkingen (toevoegen, wortel verwijderen, decrease key) op een initieel lege skew heap is $O(n \log n)$.

Bewijs/Tegenvoorbeeld:
Deze algoritme voor decrease key maakt gebruik van mergebewerking maar ook een bewerking die op het pad naar de wortel kinderen verwisseld die wij vanaf nu $fixpath$-bewerking zullen noemen.

Uit Stelling 16 weten wij dat complexiteit van een mergebewerking geamortiseerd $O(\log n)$ per bewerking is maar wat is de complexiteit van $fixpath$ eigenlijk?

Na een aantal pogingen vinden wij een reeks bewerkingen die ervoor zorgen dat er veel slechte toppen gemaakt worden tijdens onze $fixpath$.
Aan een lege skew heap S voegen wij eerst n elementen in een gegeven volgorde toe (z.v.v.a. n is even): $n-1$ dan n, dan $n-3$ en dan $n-2$ ... dan 1 en dan 2.

Algemeen: $add\_semi\_descending(i) = n - i + (-1)^i + 1$
Voeg n elementen toe: $\forall\ i \in \{1..n\}\ .\ add\_semi\_descending(i)$

Resultaat van deze toevoegingen zie je op het figuur rechts onderaan. Die toevoegingen worden in een constante tijd uitgevoerd omdat wij telkens hoogstens één vergelijking op de rechter pad uitvoeren (eventueel gevolgd

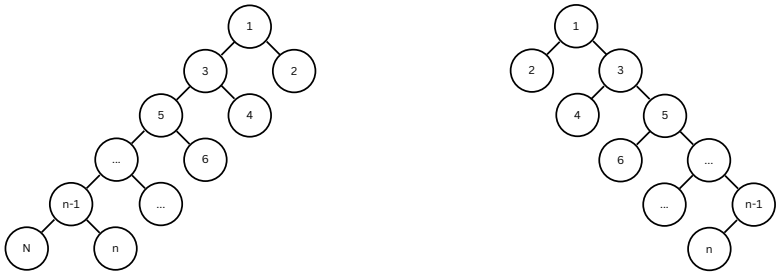


Figure 2.3: Skew heap voor en na het verwijderen van N

door 1 swap van kinderen van de wortel), dus $O(n)$ voor reeks van n bewerkingen.

Stap 1: Wij voegen een key $N > n$ toe ($S_v = S_N$) zodat onze S de linkse heap wordt (op figuur onderaan).

Stap 2: Wij passen de bewerking decrease key op onze key $v = N$ toe dat eerst de S_N uit de heap verwijdert en dan $fixpath$ bewerking op de linkse pad van S uitvoert wat in rechtse heap resulteert. Daarna mergen wij S_N terug met S wat terug in linkse heap resulteert.

Wij passen nu stap 2 n keer toe en zien hier dat wij telkens met elke decrease key $O(n)$ swaps nodig hebben:
1 (verwijderen van S_N) + $\mathbf{\frac{n}{2}}$(links -> rechts) +
1 (toevoegen van S_N) + $\mathbf{\frac{n}{2}}$(rechts -> links)
wat voor reeks van k decrease keys bewerkingen in $O(kn)$ resulteert.

Stel een aantal bewerkingen m met eerst n toevoegingen zoals in definieërd boven en dan $k > n$ decrease keys dan is $m = n + k$ en de complexiteit is: $O(n) + O(kn)$. Aangezien wij alleen kijken voor $k > n$ kunnen wij eerste term laten vallen en 2de term herschrijven als: $\Omega(n^2)$

Dat betekent dat geamortiseerde complexiteit (met algoritme 1 voor decrease key) van een reeks van n bewerkingen (toevoegen, wortel verwijderen, decrease key) op een initieel lege skew heap **niet** $O(n\ \log n)$ kan zijn.

□

Algoritme 2

> Decrease key op skew heap S:
>
> De deelboom S_v met als wortel v wordt verwijderd uit de heap. Dan wordt de wortel (dit is top v) verwijderd uit S_v met de operatie uit de cursus. Het resultaat wordt terug in S geplaatst, met de nieuwe wortel op de plaats waar v zat. Er wordt bij het terugplaatsen niet gemerged en er worden geen kinderen gewisseld. Uiteindelijk wordt de top v met de heap S gemerged.

Propositie:
De geamortiseerde complexiteit van een reeks van n bewerkingen (toevoegen, wortel verwijderen, decrease key) op een initieel lege skew heap is $O(n \log n)$.

Bewijs/Tegenvoorbeeld:
Zoals in de Stelling 16 vermeld staat:

> Het maakt het niet uit of wij de toevoegen en wortel verwijderen met een mergebewerking zouden samenvatten, de kost zou gewoon die van de mergebewerking zijn. (1)

Wij baseren ons op de deelresultaten van Stelling 16 en voegen een extra deelresultaat toe.

Deelresultaat 5: Als op een top S_v in een skew heap S in de verzameling D een decrease key bewerking wordt toegepast en het resultaat D' is dan geldt: $\Phi(D') - \Phi(D) = k$ met k aantal goede toppen die in D' slecht worden.

De echte kost van decrease key zal $O(1)$ zijn wegens punt 1 (bovenaan) die zegt dat wij geen rekening moeten houden met mergen. Hier zal namelijk 2 keer gemerged moeten worden.

	echte kost	gewijzigde kost
één nieuwe skew heap met 1 element toevoegen	1	$1 + \Phi(D') - \Phi(D) = 1$
de wortel van één skew heap verwijderen	1	$1 + \Phi(D') - \Phi(D) \leq 1$
twee skew heaps in de verzameling mergen	g+s	$g + s + \Phi(D') - \Phi(D) \leq g + s + (g - s) = 2g$ $= O(\log m)$ met m aantal bewerkingen
decrease key uitvoeren op één top in S	1	$1 + \Phi(D') - \Phi(D) = 1 + k$ (Deelresultaat 5)

Hoe groot is nu k? Wij maken gebruik van **Lemma 1** die zegt dat $k \leq \log |S|$ en wij weten dat $|S| \leq m$ met m aantal bewerkingen.

Dus is $k = O(\log m)$ een bovengrens voor de gewijzigde kost van decrease key. Dat impliceert dat $O(m \log m)$ een bovengrens voor de geamortiseerde complexiteit van een arbitraire rij van m bewerkingen op een initieel lege verzameling van skew heaps is.

Lemma 1: Als k het aantal goede toppen die na het verwijderen van top v uit de heap S slecht worden dan is $k \leq \log |S|$.

Bewijs:
Om in te zien hoeveel goede toppen na het verwijderen van top v slecht worden kijken wij wanneer één top slecht wordt. Stel S_o is de hoogste top die goed was en slecht is geworden ($|S_o| \leq |S|$). Dat gebeurt namelijk als een top S_o even veel kinderen links: $|S_l| = l$ en rechts: $|S_r| = r$ had ($r = l$), dus na het verwijderen van v aantal kinderen $l = r - 1$.
Wij bewijzen dat $k \leq \log |S|$ met inductie.

Inductiebasis: Bewijs voor $k = 1$:
Met $k = 1$ bestaat er exact één top S_o in S met $|S_r| = |S_l|$ (ook $|S_o| \leq |S|$).
Na het verwijderen van v is $|S'_l| < |S_r|$ of meer specifiek: $|S'_l| = |S_r| - 1$.
Wij weten dat S_r minstens 1 top heeft die in S_l met v correspondeerde (voor de voorwaarde $|S_r| = |S_l|$). Dus $|S_r| \geq 1$.
Deelboom van S_r samen met top S_o geven: $|S_o| \geq 2$ (ook: $1 \leq \log |S_o|$).
Zo geldt: $1 \leq \log |S_o| \leq \log |S|$.

Inductiehypothese: $n \leq \log |S|$.

Inductiestap: Bewijs voor $k = n + 1$:
Wij weten dat $r = l$ dus vervangen wij $|S_o| = |S_l| + |S_r| = 2|S_l|$:
Zo geldt: $n + 1 \leq \log 2|S_l|$.
Dan: $n + 1 \leq \log |S_l| + 1$
Dan: $n \leq \log |S_l|$ wat onze inductie hypothese is omdat S_o de **hoogste** top was en omdat wij dat reduceren tot n.

□

Chapter 3

My Heap

Voor mijn implementatie van de MyHeap heb ik de keuze gemaakt om de Pairing Heap te gebruiken, aangezien deze eerst de snelste leek te zijn tussen de heaps die relatief eenvoudig te implementeren zijn. Net zoals in het geval van de SkewHeap, veroorzaken sommige recursieve bewerkingen een *StackOverflowException*. Om dit te omzeilen, heb ik een aantal iteratieve alternatieven ontwikkeld. Een van deze iteratieve bewerkingen is de *twoPassMerge*, die een cruciale rol speelt in de definitie van PairingHeap. Bovendien is er in de 'oplossing'-map ook *MyPriorityQueueOld* aanwezig, mijn eerste poging om een PairingHeap te implementeren.

MyPriorityQueueOld houdt de lijst van siblings bij in plaats van alleen linker- en rechter-siblings. Deze implementatie zorgt voor een snelle oplossing voor veel randgevallen, waardoor het gemakkelijk te implementeren was, maar het was niet correct. Tijdens decreaseKey wordt de volgende regel uitgevoerd:

```
1     siblings.remove(this);
```

Met behulp van een profiler heb ik waargenomen dat deze regel vaak een zeer dure bewerking is. Dit is gemakkelijk te begrijpen omdat de lijst erg groot kan worden, en remove $O(n)$ is met n elementen in de lijst.

Later heb ik de PairingHeap herschreven om alleen referenties naar hun linker- en rechter-siblings op te slaan. Deze referenties moeten vaak worden bijgewerkt en over het algemeen is er meer 'boekhouding' dan in de 'Old' versie.

3.1 Benchmarks

Benchmarks voor *add* bewerking hebben weinig zin omdat die vast $O(1)$ zijn dus wij zullen direct decreaseKey met poll benchmarken.

3.1.1 Decrease key en poll

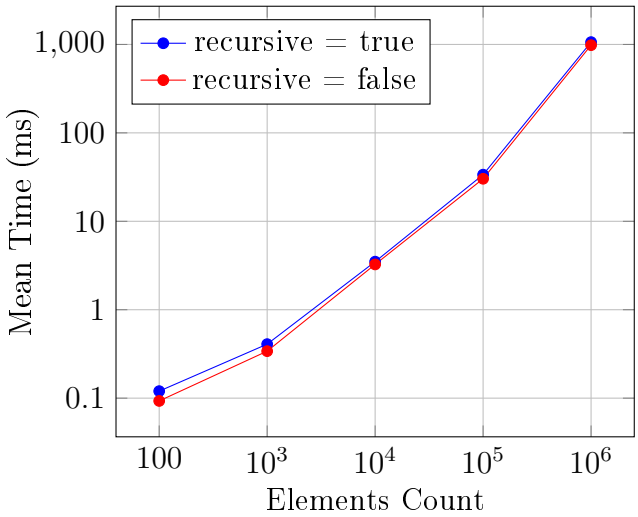


Figure 3.1: Decrease Key en poll

Bovenaan zien we dat de iteratieve methode niet alleen veilig is tegen *StackOverflowException*, maar ook dat deze klein beetje sneller is dan de eenvoudige recursieve twoPassMerge.

Opmerkelijk is dat de functie lijkt te groeien als een exponentiële functie op de exponentiële y-as. Dit suggereert dat de verwachte complexiteit hoger is dan $O(\log n)$ per bewerking.

We kunnen al vaststellen dat na het uitvoeren van n decreaseKey-bewerkingen, de twoPassMerge dan over n elementen moet itereren. Met andere woorden, veel decreaseKey-bewerkingen vertragen onze poll-bewerking.

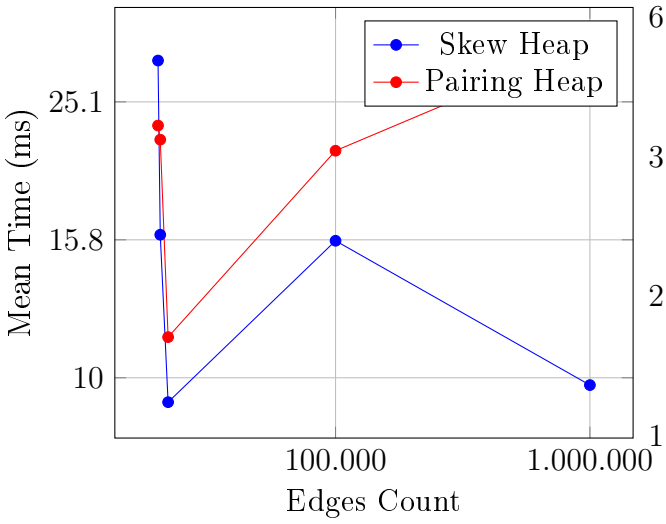
Chapter 4

Shortest Path

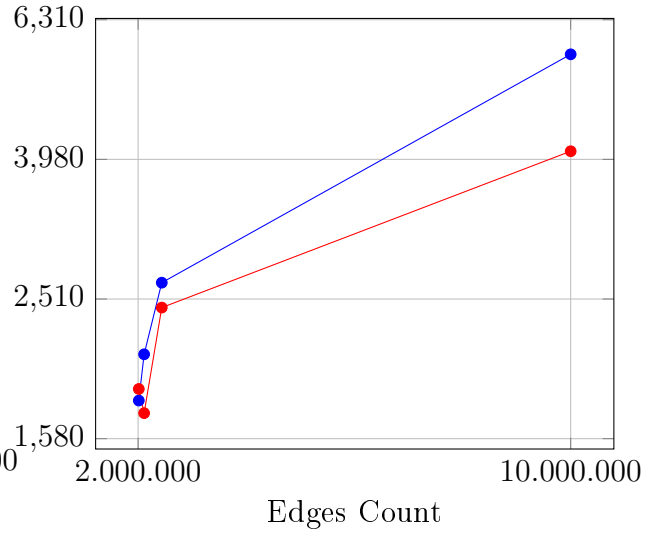
Voor de implementatie van mijn shortest path heb ik A* algoritme gebruikt. De implementatie is eenvoudig en "straight forward".

4.1 Benchmarks

We kijken verder hoe de A* presteert bij verschillende hoeveelheden edges en met verschillende heaps:



(a) 10.000 nodes



(b) 1.000.000 nodes

Het aantal bogen voor elk van de grafen werd bepaald door het aantal random gegenereerde bogen + aantal bogen die werden toegevoegd om de graaf samenhangend te maken(met algoritme beschrijven in sectie **1.2**). Dus

voor grafiek a:
100 random + 19.951 verplicht
1.000 random + 19.459 verplicht
10.000 random + 11.965 verplicht
100.000 random + 0 verplicht
1.000.000 random + 0 verplicht

We zien bij grafiek a dat A* het snelst is als het aantal bogen "dicht" bij het aantal nodes is of wanneer de graaf (bijna) compleet is (wat altijd triviale oplossingen oplevert met padlengte = 1).

Grafiek b ziet er anders uit en ik vermoed dat als het aantal bogen 10^{6^2} zal zal bereiken dat de uitvoeringstijden op dezelfde manier zullen afnemen omdat de graaf bijna compleet zal worden.

In het algemeen is het moeilijk te zeggen welke heap (altijd) beter presteert, maar voor de real world scenario's zou ik kiezen voor pairing heap omdat deze met ordes van miljoenen nodes beter is dan skew heap.