Speciale

Implementering og udvikling af maksimum delsum algoritmer

Datalogisk Institut, Aarhus Universitet
Krzysztof Piatkowski 20033003

Engineering Maximum Delsum Algorithms

Under vejledning af: Gerth Stølting Brodal

Master's Thesis 01/09

Specielt tak til:

Gerth Stølting Brodal
For alle de mange gode ideer og opbakningen.

Jannie Tolstrup
For at gennemlæse og rette stavefejl.

Jonathan Fromer
For at gennemlæse og rette stavefejl

Abstract

The Maximum delsum problem, is about finding the sub-vector with the largest sum in a sequence of n numbers. A variant of this problem is finding the k sub-vectors with the largest sums. This variant is known as the k maximum delsum problem. In this thesis a number of algorithms are implemented and compared to solve the problem. The algorithms are evaluated from a practical point of view, specifically on how well they do in their execution time.

To find the algorithms two different approaches have been applied. The first is a naive approach. The $\binom{n}{2} + n$ possible sub-vectors is found and from these the k with the largest sums are chosen. A part of these algoritms are based on selection and therefore a number of varying selection algorithms are described.

The other approach is based on an article[1] which solve the k maksimum delsum problem in the optimal $O(n + k)$ time. This is done by structuring the sums in a specific way and inserting them into a special persistent heap. To obtain a better practical execution time, the heap is replaced by a persistent searchtree and later by a priority queue.

From these two approaches there is a total of eight different algorithms, who's practical and theoretical execution time is documented, whereafter the described algorithms are compared.

Resumé

Maksimum delsum problemet handler om at finde den delvektor, som har den største sum ud fra en sekvens af n tal. En variant af dette problem, går ud på at finde de k delvektor som de største summer. Denne variant kaldes for k maksimum delsum. I dette speciale implementeres og sammenlignes der forskellige algoritmer til at løse denne variant af problemet. Algoritmerne bedømmes udfra et praktisk synspunkt, altså hvor god deres udførelsestid er.

For at finde frem til algoritmerne er der benyttet to forskellige tilgange. Den første er en naiv tilgang. De $\binom{n}{2} + n$ mulige delvektorer findes og udfra disse vælges de k, der giver den største sum. En del af algoritmerne er baseret på selektion og af den grund beskrives en række forskellige selektionsalgoritmer.

Den anden tilgang er baseret på en artikel [1] som løser k maksimum delsum problemet i den optimale $O(n+k)$ tid. Dette gøres ved at strukturere summerne på en bestemt måde og indsætte dem i en speciel persistent heap. For at opnå en bedre praktisk udførelsestid, erstattes heapen med et persistent søgetræ og senere med en prioritetskø.

Udfra disse to tilgange findes der i alt otte forskellige algoritmer, hvis praktiske og teoretiske udførelsestid dokumenteres, hvorefter de beskrevne algoritmer sammenlignes.

Indhold

1 Introduktion **2**

2 Opsætning af forsøg og repræsentation **8**
2.1 Repræsentation og valg af data . . . 8
2.1.1 Valg af input . . . 8
2.1.2 Valg af output . . . 9
2.1.3 Tilfældige tal . . . 10
2.2 Tests af algoritmer . . . 10
2.2.1 Ratio test . . . 11
2.2.2 Power test . . . 11
2.3 Opsætning . . . 11

3 n^2-familien **13**
3.1 n^2 priority . . . 13
3.2 n^2 select . . . 15
3.3 n^2 select m . . . 17
3.4 n^2 historik . . . 21
3.5 n^2 block . . . 23
3.6 Konklusion på n^2-familien . . . 27

4 n-familien **29**
4.1 Suffiks mængder . . . 29
4.2 n priority . . . 32
4.3 n searchtree . . . 35
4.4 n matrix . . . 37
4.5 Konklusion på n-familien . . . 39

5 Selektion **41**
5.1 Find . . . 41
5.2 Partition . . . 42
5.3 Quickselect . . . 45
5.4 Median of medians . . . 48
5.5 Sammenligning af selektionsalgoritmer . . . 50

6 Persistente datastrukturer **51**
6.1 Persistens . . . 51
6.1.1 Naivt . . . 52
6.1.2 Fat node metoden . . . 52

6.1.3 Node-copy metoden 53
6.2 Persistent Insert heap 53
6.2.1 Insert heap 53
6.2.2 Persistering af Insert heap 55
6.3 Persistente Søgetræer 58
6.3.1 Binært søgetræ 58
6.3.2 Persistering af Binært søgetræ 60
6.3.3 Rød sorte træer 61
6.3.4 Persistering af Rød sorte træer 64

7 Konklusion **66**
7.1 Fremtidig arbejde 66
7.2 Konklusion 67

A Appendix **69**
A.1 Eksperiment setup - uddybning 69
A.2 Profiler udskrifter 72

Litteratur **74**

Kapitel 1

Introduktion

Inden for de sidste mange år, er biomedicin og molekylær biologi blevet mere og mere interessant. Specielt er det interessant, at finde områder i DNA sekvenser, som biologisk set er interessante ud fra forskellige kriterier. Et kriterie kan f.eks. være, at finde identiske eller ens områder i forskellige arter. Et andet kriterie kan være, at finde områder, som er rige på G/C basepar.

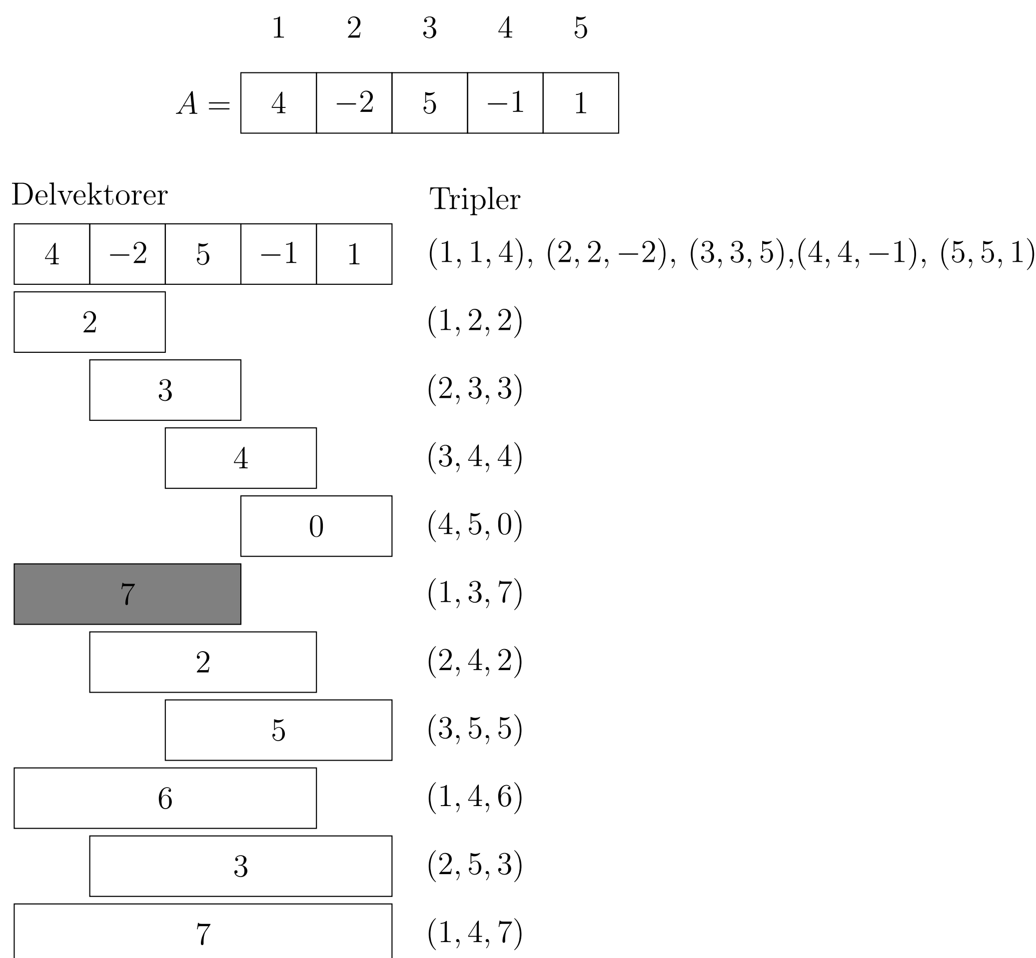
En typisk metode til, at identificere områder, som opfylder et kriterie, er ved dele sekvensen op i delsekvenser og give disse en score. For at finde et interessant område udregnes scoren for delsekvenserne af et område. Hvis kriteriet for et godt område, er et område med en høj score, kaldes problemet at finde et området for "maksimum delsekvense score". Hvis scoren regnes ud som summen af delsekvenserne kaldes problemet for "maksimum delsum". Formelt kan problemet defineres som følger:

Givet en vektor $A[1,\ldots,n]$, findes der for hver sammenhængende delvektor $A[i,\ldots,j]$ en sum (score), hvor $sum = \sum_{t=i}^{j} A[t]$. En vektor af længde n har $\binom{n}{2} + n$ forskellige delvektorer, valget af to forskellige indices, samt alle valg, hvor de to indicer er ens. En sum og de to indicer udgøre et triple (i, j, sum). I dette speciale vil løsningen til et maksimum delsum problem være, at finde det største triple og ikke bare den største sum. I nogle af de relaterede artikler er løsningen på et maksimum delsum problem kun summen, men det er ofte lige så interessant at vide, hvor denne sum kan findes.

Et eksempel kan ses i figur 1.1, hvor alle 10 delvektorer og dertilhørende tripler vises, den ene af løsningerne er skraveret. Bemærk at løsningen ikke er entydigt bestemt, da der findes to mulige løsninger til det givne problem: $(1, 3, 7)$ og $(1, 5, 7)$, der begge har den samme sum.

Problemet og en algoritme, som løser problemet i den optimale $O(n)$ tid, beskrives i [2]. Algoritmen tager inputtet A og gennemløber det fra ende til anden. Hovedideen er, at antage at problemet er løst for $A[1,\ldots,i-1]$, hvor $1 \leq i < n$ og det største triple t_{max} er fundet. Det største triple i $A[1,\ldots,i]$ er enten det samme som i $A[1,\ldots,i-1]$, eller også er det største triple der slutter i i betegnet t_i. Triplet t_i kan enten være triplet $(i, i, A[i])$ såfremt t_{max} har en negativ sum, eller også er det t_{max} hvor $A[i]$ er lagt til summen, og j indekset er talt op. Det største af de tre tripler sættes til at være t_{max} for $A[1,\ldots,i]$.

Løsningen til $A[1,\ldots,i]$ kan altså findes ud fra $A[1,\ldots,i-1]$ i konstant tid. Ved at gennemløbe hele inputtet på denne måde, kan maksimum delsum problemet løses i linær



Figur 1.1: et maksimum delsum problem

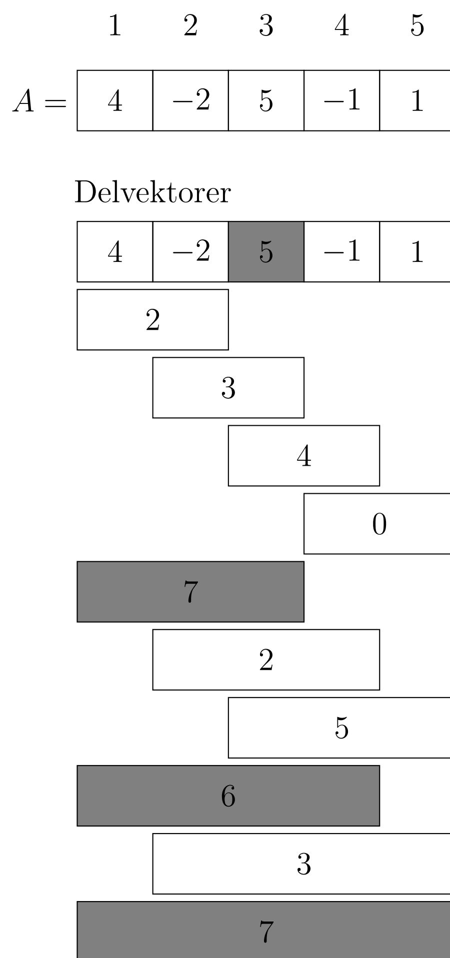
tid.

Ved at generalisere maksimum delsekvens score problemet, er der kommet forskellige varianter af problemet. En retning er at kræve, at scoren for en sekvens, kan udregnes som en funktion og ikke bare summen af delsekvenserne. I [3] beskrives maksimum-densitet sekvens problemet, hvor inputtet er en sekvens af par (a_j, w_j). Densiteten af en delsekvens fra i til j beregnes ved $d(i, j) = (ai + ai + 1 + \cdots + aj)/(wi + wi + 1 + \cdots + wj)$ og løsningen til maksimum-densitet sekvens, er den delsekvens som har den største densitet.

En anden retning er at finde en dækning af en sekvens. En dækning er en mængde af delsekvenser, som er disjointe. Problemet maksimum disjointe dækning beskrives i [4, 5], og går ud på at finde en mængde med k delsekvenser, som laver en dækning, men hvor summen af scorene for sekvenserne er størst.

I dette speciale undersøges en variant af problemet, som blev introduceret i [6]. I varianten findes de k største delsummer og de dertilhørende tripler. I modsætning til [4, 5] må løsningerne gerne overlappe hinanden. Varianten kaldes for k maksimum delsum. Løsningen til problemet er en mængde af k tripler, hvor $1 \leq k \leq \binom{n}{2} + n$.

Et eksempel på k maksimum delsum problemet samt en løsning, ses i figur 1.2, den ene af løsningerne er skraveret. Ligesom i maksimum delsum problemet er løsningen ikke entydig



Figur 1.2: Et k maksimum delsum problem, $k = 4$

bestemt, da der findes to løsninger til det givne problem:

$$\{(1,5,7),(1,3,7),(1,4,6),(3,3,5)\}$$
$$\{(1,5,7),(1,3,7),(1,4,6),(3,5,5)\}$$

En nedre grænse for udførelsestiden for k maksimum delsum problemet er $\Omega(n+k)$ tid, idet alle indgange n i A skal undersøges og outputtet består af k tripler.

Relateret arbejde

Et af de første forsøg på, at finde en god k maksimum delsum algoritme beskrives i [6]. Ideen med algoritmen er, at bygge alle præfiks summer for A på følgende måde:

$$sum_{prefix}[i] = \begin{cases} 0 & \text{for } i = 1 \\ A[i] + sum_{prefix}[i-1] & \text{for } i > 1 \end{cases}$$

Summen fra $A[i]$ til $A[j]$ kan findes ved $sum_{prefix}[j] - sum_{prefix}[i-1]$. Den største sum kan findes ved at gennemløbe A og holde styr på den største og den mindste præfiks sum.

Tabel 1.1: Relaterede algoritmer

Forfatter	Tid
Bae & Takaoka[6]	$O(k \cdot n)$
Bengtsson & Chen[7]	$O(\min\{k + n\log^2 n, n\sqrt{k}\})$
Bae & Takaoka[8]	$O(n \log k + k^2)$
Lin & Lee[9]	$O(n \log n + k)$
Bae & Takaoka[10]	$O((n + k) \log n)$
Brodal & Jørgensen[1]	$O(n + k)$

Ideen ekspanderes nu til k summer. For indgang i i A, opbygges der en sorteret liste med de k mindste præfiks summer. Udfra denne udregnes en liste med k kandidater ved, at gennemløbe listen med de mindste præfiks summer og trække dem fra den $i$te præfiks sum. Listen med kandidater for den $i$te indgang flettes nu sammen med kandidatlisten for $i - 1$, så summerne stadig er sorteret. Kun de k største summer bevares til næste iteration. Til sidst vil listen indholde de k største summer. Alle skridtene til hver indgang i A tager højest $O(k)$ tid, hvilket giver en udførelsestid på $O(n \cdot k)$ og et pladsforbrug på $O(k)$.

Problemet med denne algoritme er, at selvom den er forholdsvis simpel og effektiv ved lille k, vil både tids- og pladsforbruget gøre algoritmen meget langsom, når k vokser.

I [7] videreføres ideen. Først opbygges to matricer B og C, udfra præfiks summerne. Hver af indgangene i B består af elementerne fra mængden $\{sum_{prefix}[1], \ldots, sum_{prefix}[n]\}$ og C af elementerne $\{-sum_{prefix}[1], \ldots, -sum_{prefix}[n]\}$. Den j, ite indgang i matricen $B + C$ svarer til $sum_{prefix}[j] - sum_{prefix}[i-1]$. De k største tripler findes ved at konstruere matricen $B + C$ trinvist og lokalisere de største tripler. Algoritmen har en udførelsestid på $O(\min\{k + n\log^2 n, n\sqrt{k}\})$.

I [8] optimeres udførelsestiden på den første algortime. Ved at bruge et 2–3 træ[11, s. 476] til opbevaringen af de mindste præfiks summer. Derved optimeres udførelsestiden fra $O(\log k)$ til $O(1)$, plus $O(k)$ til at generere kandidaterne. Indsættelsen i 2–3 træet koster $O(\log k)$ tid og skal fortages n gange. Den forbedrede udførelsestid bliver derfor $O(n \log k + k^2)$, hvilket er hurtigere end $O(\min\{k + n\log^2 n, n\sqrt{k}\})$ for $k \leq \sqrt{n} \log n$. Igen vil algoritmen få problemer når k vokser.

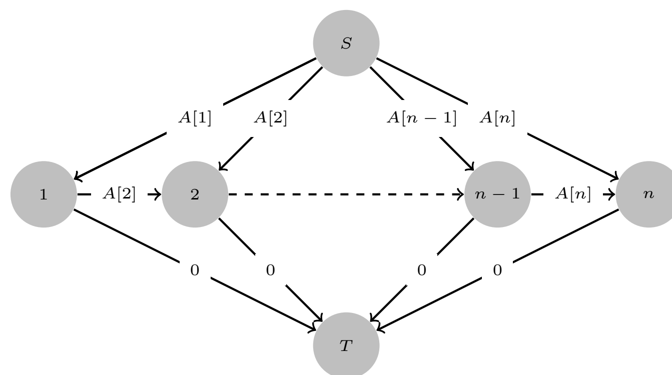
I [9] beskrives en randomiseret algoritme, der kan finde den største sum i forventet $O(n \log n)$ tid og ud fra denne udledes en algoritme, som forventes at kunne finde de k største summer i $O(n \log n + k)$ tid.

Den næste idé blev præsenteret i [10]. Hvis man ser på en triple til en sum (i, j, sum) er det nemt at se, at man kan bygge en tabel, der er $n \times n$ stor med alle $\binom{n}{2} + n$ mulige summer. En søgning efter de k største summer foregår ved at finde den $k$te største sum og inkludere alle summer, som er større end denne. I artiklen oprettes og udfyldes denne tabel partielt og løbende, hvilket giver en algoritme, der bruger $O((n + k) \log n)$ tid på, at finde de k største summer.

I [1] beskrives en algoritme, som opbygger en persistent heap i lineær tid, hvorefter [12] bruges til at udvælge de k største tripler fra heapen i $O(k)$ tid. Algoritmen beskrives mere

detaljeret i kapitel 4. Udførelsestiden for algoritmen er $O(n + k)$, altså det optimale.

Den sidste algoritme[13] er en reduktion fra et andet problem. Ideen er at opbygge en speciel graf som ses i figur 1.3. Udfra grafen ses det, at de k største stier fra S til T, giver de k største summer. Ved at modificere [14], som kan finde de k korteste stier i en orienteret graf til istedet at finde de længste stier, kan et k maksimum problem løses i samme tid som [14]. Udførelsestiden for [14] er $O(m+n+k)$, men fordi antallet af kanter er $O(n)$ vil udførelsestiden for k maksimum delsums algoritmen være den optimale $O(n+k)$.



Figur 1.3: Graf repræsentation af k maksimum delsum problem

Formål og struktur

Formålet med dette speciale er, at finde en algoritme, der løser k maksimum delsum problemet hurtigt i praksis. De fleste af overstående løsninger klarer sig dårligt, når k bliver stor. Det er derfor også et krav, at algoritmerne der findes også kan løse problemer for stort k.

For at finde en hurtig algoritme bruges der en praktisk tilgang. Hvergang en algoritme er implementeret, argumenteres der for teortisk udførelsestid og korrekthed. Algoritmen testes først for korrekthed ved, at sammenligne de tripler den producerer med en i forvejen virkende algoritme. Dernæst undersøges algoritmens praktiske udførelsestid og der argumenteres, for at denne følger den teoretiske. Til sidst evalueres algoritmen og der findes forbedringer. Disse forbedringer bruges til, at udlede næste algoritme. I kapitel 2 uddybes fremgangsmåden yderligere.

Algoritmerne i dette speciale følger to hovedideer: Den første idé er at udvælge de k største tripler udfra alle de $\binom{n}{2} + n$ mulige tripler ved, at producere dem først.

Den første algoritme n^2 priority, indsætter et triple ad gangen i en prioritetskø, når der er k tripler i prioritetskøen, fjernes det mindste triple fra køen. Fordi prioritetskøen hele tiden skal holde triplerne sorteret i denne algoritme, bruges der en masse tid på at justere køen. Dette forsøger n^2 select at forbedre ved, at reducere antallet af sorteringer. Algoritmen kører en selektion på samtlige tripler, hvilket viser sig at være effektivt, men har den ulempe, at algoritmen bruger $O(n^2)$ arbejdsplads. I n^2 select m reduceres arbejdspladsen til $O(k)$ ved at køre selektionsalgoritmen flere gange. For yderligere, at reducere antallet af selektioner undgår n^2 historik, at køre selektion på de tripler, som alligevel ikke kan være blandt de k største. Til sidst forsøger n^2 block, at gøre gennemløbet af de $\binom{n}{2} + n$ tripler I/O effektivt.

Tabel 1.2: Implementeret algoritmer

Navn	Tid	Plads
n^2 priority	$O(n^2 \cdot \log k)$	$O(k)$
n^2 select	$O(n^2)$	$O(n^2)$
n^2 select m	$O(n^2)$	$O(k)$
n^2 historik	$O(n^2)$	$O(k)$
n^2 block	$O(n^2)$	$O(k)$
n priority	$O(n + k\log(n))$	$O(k)$
n searchtree	$O((n + k)\log n)$	$O(n \log n)$
n matrix	$O(n^2)$	$O(n)$

Den anden idé er baseret på [1]. I n priority bruges der en prioritetskø til, at holde styr på, hvor de største tripler findes i en persistent heap. I n searchtree erstattes heapen med et persistent søgetræ, da dette reducerer størrelsen på prioritetskøen og pladsforbruget. I n matrix bruges en prioritetskø som underliggende datastruktur.

De to ovenstående ideer er valgt, fordi den første idé repræsenterer den mest naive måde at løse k maksimum delsum problemet på. Det er med stor sandsynlighed denne algoritme, som vil blive implementeret, hvis man ikke på forhånd har kendskab til problemet og de løsninger der er foreslået i ovenstående tekst. Den anden idé repræsenterer den optimale måde, at løse k maksimum delsum problemet i teoretisk forstand. Det er derfor interessant at se, hvor disse to ender mødes og hvordan de klarer sig i forhold til hinanden.

Specialet er opbygget på følgende måde:

- **Kapitel 2** Alle de praktiske detaljer omkring tests samt hvordan inputtet og outputtet er repræsenteret.
- **Kapitel 3** Gennemgår forskellige k maksimum delsums algoritmer, der er baseret på den første idé.
- **Kapitel 4** Gennemgår forskellige k maksimum delsums algoritmer, der er baseret på den anden idé.
- **Kapitel 5** Beskriver forskellige varianter af en selektionsalgoritme. Disse bruges som dele i k maksimum delsum algoritmerne.
- **Kapitel 6** Beskriver persistens samt en persistent heap og to forskellige persistente søgetræer.
- **Kapitel 7** Konklusion samt fremtidigt arbejde.

Kildekode til algoritmerne kan hentes på:
`http://daimi.au.dk/~kpi/thesis`
eller i pakket udgave på:
`http://daimi.au.dk/~kpi/thesis/code.tar`

Kapitel 2

Opsætning af forsøg og repræsentation

I dette kapitel beskrives de forskellige tekniske aspekter ved specialet. Da udviklingen af algoritmerne har en praktisk tilgang, er det vigtigt at kende til de forskellige tekniske aspekter, der kommer til at præge algoritmernes udførelsestid.

Først diskuteres valg af input og output samt, hvordan disse repræsenteres. Derefter gennemgåes de forskellige metoder, der benyttes til at afgøre om algoritmerne er korrekte. Til sidst diskuteres det kort, hvordan målingerne er blevet foretaget samt computeres specifikationer. En uddybet specifikation af parametre, samt opsætning af systemet ses i appendiks A.1.

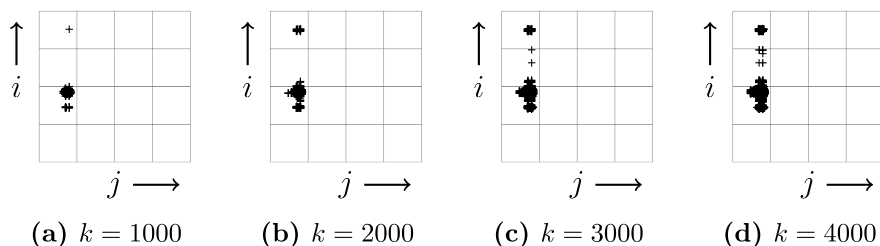
2.1 Repræsentation og valg af data

2.1.1 Valg af input

Maksimum delsum varianterne bliver først interessante, når A kan optage en negativ værdi. Ved maksimum delsum er løsningen triviel, når inputtet er positivt. Ved k maksimum delsum vil løsningerne altid være hele A, foruden en del af en af enderne. Kravet til repræsentationen er derfor, at elementerne i A skal kunne være negative.

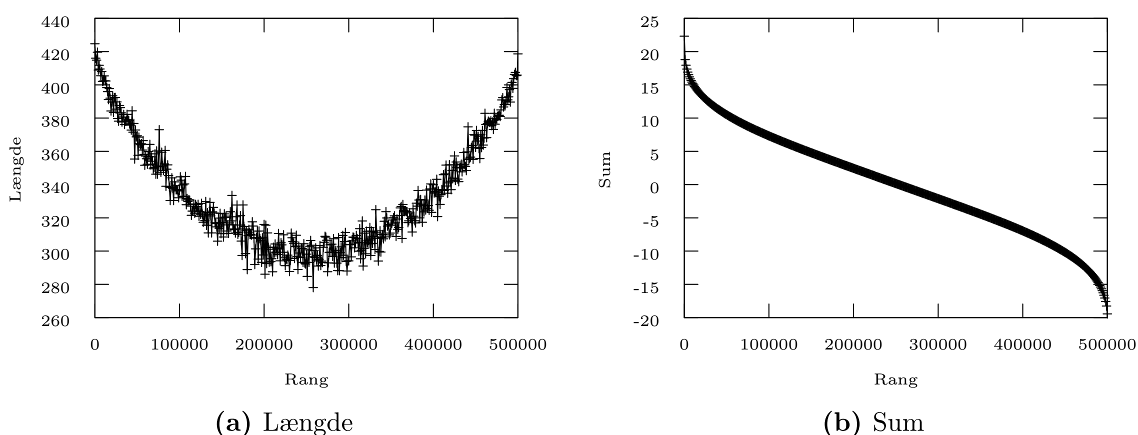
For at undgå overflow og minimere antallet af ens tripler, er A valgt til at bestå af reel tal i intervallet $[-1, 1)$. I figur 2.1 vises fordelingen af triplerne ved dette valg. Figuren viser, hvad de k største triplers i og j indicer er, når $n = 1000$. De fire figurer viser den samme løsning, når k går fra 1000 til 4000. Det interessante er at løsningerne ligger tæt på hinanden og er grupperet. Grupperingen skyldes, at et triples indicer ikke skal flyttes ret meget, før det er forventet, at ramme et triple med tilsvarende sum. Grunden til, at triplerne ligger tæt på kanten, er fordi den forventede sum over hele A er nul. Det vil sige, det forventes at en løsning kan udvides/indsnævres til en af siderne, uden at løsningen bliver dårligere.

I figur2.2a vises en graf med alle løsningerne for et problem med $n = 1000$. I eksperimentet er hvert målepunkt gennemsnittet af 1000 kørsler. Det ses tydeligt, at de længeste tripler både har den mindste og den største rang. I Figur2.2b ses fordelingen af summerne, grafen



Figur 2.1: Fordeling af triple ved uniform data $[-1, 1)$

viser at der er få tripler, som er markant større/mindre end resten. For resten af triplerne, falder summen næsten lineært.



Figur 2.2: Tripler sorteret efter rang, hvor $n = 1000$

Algoritmerne er implementeret i *C++*, i appendiks A.1 ses version, parametre til compileren, samt de resterende typer.

2.1.2 Valg af output

Fordi k kan blive $\binom{n}{2} + n$ stor, er det vigtigt at vælge en god repræsentation af outputtet.

Ved at lade de to indicer være den mindste mulige type *unsigned short* og *sum* være af typen *double*, kommer et triple til at fylde $12B$ ($2B$ for et indeks, $8B$ for *sum*). Det maksimale antal af tripler, der kan være i $2GB$ hukommelsen bliver derfor:

$$\begin{aligned}
\frac{n \cdot (n-1)}{2} + n &= 2GB/12B \\
\frac{n \cdot (n+1)}{2} &= 2GB/12B \\
\frac{n \cdot (n+1)}{2} &= 1,8 \cdot 10^8 \\
n &= \sqrt{1,8 \cdot 10^8 \cdot 2} \\
n &= 19000
\end{aligned}$$

Det vil sige at for $n > 19000$ er det ikke muligt, at finde alle k tripler, hvis de skal gemmes i hukommelsen. Istedet for at returnere hele løsningen, "streames" denne. Det vil sige, at algoritmerne returnerer et triple ad gangen og i vilkårlig rækkefølge. På den måde er det muligt, at repræsentere en løsning uden at bruge mere plads, end hvad der bruges internt i algoritmen. Det er stadig en fordel, at have en så lille repræsentation som muligt, da det tillader algoritmerne at gemme flere tripler i hukommelsen og derved løse større problemer. Selvom løsningerne reelt "streames", beskrives algoritmerne som om de returnerer en vektor af tripler, hvilket simplificerer beskrivelsen og er reelt uden betydning.

Fordi sum-delen af triplerne er summen af en række indgange i A, vil den ikke være unik. Det kan potentielt blive et problem at finde en absolut orden af en mængde tripler. En løsning er, at ordne triplerne efter deres sum først og dernæst efter deres indicer. Selvom dette giver en absolut orden, gør det også algoritmer, der sammenligner tripler, langsommere. Af den grund afgøres ordenen af tripler kun ud fra sum-delen. De steder, hvor manglen på en absolut orden er et problem, tages der stilling til dette.

2.1.3 Tilfældige tal

I C standarden er der indbygget en tilfældighedsgenerator std::rand [15, side 312]. Desværre er specifikationerne omkring generatoren meget vage og det antages generelt, at generatoren er dårlig og ikke specielt tilfældig. Da nogle af algoritmerne er baseret på tilfældighed, er det derfor vigtigt, at have en god generator. I stedet bruges en implementation af *Mersenne Twister* [16], som returnerer reelle tal i intervallet $[0, 1)$. Implementationen har en periode på mellem 2^{607} og 2^{216091}. I eksperimenterne er perioden sat til 2^{19937}, som er langt større end hvad der skal bruges. Generatoren bliver *seeded* med tiden i computeren.

I nogle af algoritmerne skal der bruges tilfældige heltal. Et tilfældigt heltal i intervallet $[0, N)$ findes ved først, at trække en *double* fra generatoren, hvorefter den ganges med N og rundes ned. Et heltal i intervallet $[a, b)$ findes ved først, at finde et heltal i intervallet $[0, b - a)$, hvorefter der lægges a til.

2.2 Tests af algoritmer

For at sikre, at algoritmerne returnerer den korrekte løsning, testes de først mod reference algoritmen n^2 priority. Denne algoritme er meget simpel og det er ikke svært, at se om den fungerer udfra små eksempler. Dernæst er de andre algoritmer blevet implementeret og testet løbende. Først op imod referencen og senere imod hinanden. Dette reducerer sandsynligheden for fejl, jo større mængden af algoritmer bliver.

Der er dog et problem: Hvis der findes flere løsninger, er det svært at afgøre, hvorvidt en løsning er gyldig, som konsekvens af, at løsningerne ikke er entydigt bestemt og der ikke er en absolut orden. Det er naturligvis muligt, at lave en algoritme, der returnerer alle mulige løsninger, men algoritmen vil ikke være praktisk anvendelig for store datasæt. For at afgøre om en algoritme returnerer et gyldigt løsning, køres følgende proces for hver implementeret algoritme:

- $2 \leq n < \infty$

- $k = (\frac{n \cdot (n-1)}{2})$ vælges, sorteres efter sum og sammenlignes med enhver anden implementeret algoritme.
- $k = r \mod \frac{n \cdot (n-1)}{2}$ vælges, hvor r er et tilfældigt heltal. Løsningerne sorteres og triplernes sum og indicer sammenlignes parvis.

Selvom processen ikke er en garanti for, at algoritmerne er korrekte, bliver sandsynligheden for korrekthed større, jo længere processen kører og jo flere algoritmer, der bliver implementeret. Processen har været med til at mindske mængden af fejl og har sikret, at algoritmerne har fungeret efter omskrivning og rettelser.

2.2.1 Ratio test

I [17, s.45] præsenteres to forskellige måder at analysere og visualisere data på. Den første metode kaldes for ratio testen og bruges til at finde ud af om en algoritmes udførelse er den forventede. Hvis $f(n)$ er den forventede udførelsestid for algoritmen på en instans af størrelse n, er det muligt, at finde ud af om algoritmens reelle udførelsestid $t(n)$ passer med den forventede, ved at plotte $r(n)$ hvor $r(n) = \frac{t(n)}{f(n)}$. Hvis kurven ikke konvergerer, er $f(n)$ underestimeret, hvis kurven derimod konvergerer mod nul, er $f(n)$ overestimeret. Ved et korrekt estimat af $f(n)$ vil kurven konvergere mod en konstant. Når der ved et eksperiment står at grafen er plottet med ratio test, henvises der til denne test. Problemet med testen er, at den ikke er speciel præcis, når udførelsestiden er større end $O(n^c)$, hvor $c > 1$.

2.2.2 Power test

Den anden test bruges til, at undersøge en algoritmes udførelsestid, hvis udførelsestiden ikke er kendt på forhånd. Ideen er at transformere dataene fra udførelsen, så $(x, y) \rightarrow (x', y')$ hvor $x' = \log x$ og $y' = \log y$. Hvis algoritmens udførelsestid er $t(n) = b \cdot n^c$, så vil log-log transformationen give at $y' = cx' + b$. Hvis grafen for log-log transformationen fittes til en linje, er det ud fra linjens hældning muligt, at afgøre hvad konstanterne b og c er for algoritmen. Med andre ord er det muligt at afgøre, hvad eksponenten for algoritmen er. Power testen kræver at algoritmen har en polynomiel udførelsestid.

2.3 Opsætning

Computeren, algoritmerne bliver kørt på, har $2GB$ ram, $32KB$ $L1$ cache samt $4MB$ $L2$ cache. Computeren er installeret med et $32bit$ styresystem, hvilket vil sige at en pointer fylder $4B$.

Alle målepunkter er gennemsnittet af fire kørsler. For at sikre, at alle unødvendige programmer er swappet ud af hukommelsen, er der først fortaget en ekstra måling, som kasseres. Maskinen har lov til, at bruge $0,2$ sekunder på at rydde op imellem hver måling.

Til selve målingen er *PAPI* (Performance Application Programming Interface) frameworket [18] brugt. Frameworket bruger *perfctr* [19], som er en driver til de low-level performancetællere, der er indbygget i processoren. Ved at patche linux-kernen med*perfctr* kan man igennem *PAPI* tilgå antallet af cache misses samt den reelle tid, der er brugt i en program. Dette bevirker også, at tiden for at swappe ind og ud af hukommelsen, ikke bliver vist på graferne.

Kapitel 3

n^2-familien

Den første hovedidé til at løse k maksimum delsum problemet er, at gennemløbe alle $\binom{n}{2}+n$ delvektorer og beregne triplerne udfra disse. Ud af triplerne udvælges så de k med den største sum. Den bedste udførelsestid en algoritme i n^2-familien derfor kan opnå er $\Omega(n^2+k)$.

I de efterfølgende afsnit beskrives der forskellige algoritmer, hvor implementationen af hver algoritme er forsøgt optimeret med hensyn til den praktiske udførelsestid ved, at minimere konstanter. Den første algoritme n^2 priority indsætter et triple ad gangen i en prioritetskø, når der er k tripler i prioritetskøen fjernes det mindste triple fra køen. Denne algoritme er forholdsvis langsom. I n^2 select forbedres den praktiske udførelsestid ved, at reducere antallet af sorteringer. Algoritmen kører en selektion på samtlige tripler, hvilket viser sig at være effektivt, men har den ulempe, at algoritmen bruger $O(n^2)$ plads. I n^2 select m reduceres arbejdspladsen til $O(k)$ ved at køre selektionsalgoritmen flere gange. For yderligere at reducere antallet af selektioner undgår n^2 historik, at køre selektion på tripler, som ikke kan være kandidiater til de k største. Til sidst optimerer n^2 block gennemløbet af de $\binom{n}{2}+n$ tripler ved at gøre det I/O effektivt.

3.1 n^2 priority

Den første løsning gennemløber de $\binom{n}{2}+n$ mulige tripler. Mængden med alle $\binom{n}{2}+n$ tripler kan beskrives på følgende måde: $\{(i,j,sum) \mid 1 \leq i \leq j \wedge sum = \sum_{s=i}^{j} A[s]\}$. Det vil sige, at triplerne kan findes ved at lade i løbe fra 1 til $|A|$ og lade j løbe fra i til $|A|$. Summen til et triple kan opbygges iterativt fra et tidligere triple på følgende måde:

$$(i,j,sum) = \begin{cases} (j,j,A[j]) & \text{for } i=j \\ (i,j,s+A[i]) & \text{for } i<j \text{ hvor } (i,j-1,s) \end{cases}$$

Hvergang der produceres et triple i gennemløbet, indsættes dette i en prioritetskø [20] q, hvor triplerne bliver prioriteret efter deres sum. Når størrelsen af køen bliver større end k fjernes det mindste triple fra køen. Det vil sige, at kun de k største tripler, genereret indtil videre, bevares i køen. Til sidst returneres triplerne fra køen.

Algorithm 1: n^2 priority

```
   input  : A, k
   output: Priorty Queue q
1  for i ← 1 to n do
2  |  s ← 0;
3  |  for j ← i to n do
4  |  |  s ← s + A[j];
5  |  |  q.insert(i, j, s);
6  |  |  if q.size > k then
7  |  |  |  q.removeMin();
8  |  |  end
9  |  end
10 end
11 return q
```

Korrekthed

Alle $\binom{n}{2}+n$ mulige delvektorer gennemløbes og derfor må de k største tripler nødvendigvis også blive fundet. Hvergang et triple er fundet indsættes det i køen og det mindste fjernes. Korrekthed følger derfor af korrekthed for prioritetskøen.

Kompleksitet

De $\binom{n}{2} + n$ tripler findes i $O(n^2)$ tid, hvor der indsættes et triple i prioritetskøen hver gang og muligvis slettes der et. Den underliggende datastruktur for prioritetskøen er en implicet binærheap[20], af den grund tager operationerne på prioritetskøen $O(\log |Q|)$ tid, hvor $|Q|$ er antallet af elementer i køen. Sammenlagt giver det en udførelsestid på $O(n^2 \cdot \log k)$. Fordi køen er impliciet prioritetskø er pladsforbruget $O(|Q|)$. Antallet af tripler i køen er altid højest $k + 1$, så det samlede pladsforbrug er $O(k)$.

Implementation

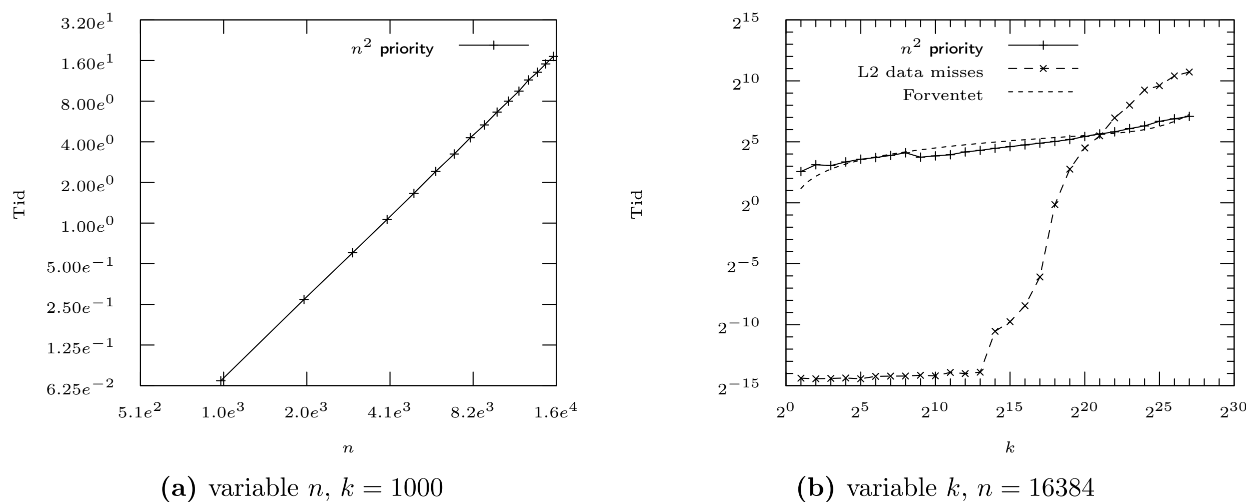
Prioritetskøens insert operation er implementeret med $C{+}{+}$ kaldet push_heap, som jævnfør [21] bruger højest $\log |Q|$ sammenligninger. Tilsvarende er removeMin implementeret med pop_heap, som bruger $2\log(|Q|)$ sammenligninger.

Algoritmen bruger $n^2 \cdot \log k$ sammenligninger på at indsætte triplerne i køen og $2(n^2 - k) \cdot \log k$ sammenligninger på at fjerne triplerne igen. Det er derfor forventet, at antallet af sammenligninger er $n^2 \cdot \log k + 2(n^2 - k) \log k$.

Eksperimenter

Først undersøges n parameteren ved at fastholde k. Algoritmen har en forventet udførelsestid på $O(n^2)$ og af den grund bruges power-testen. Grafen for dette kan ses i figur 3.1a.

For at undersøge k parameteren, varieres k og n fastholdes. Grafen ses i figur 3.1b. På grafen ses også den forventede udførelsestid ud fra antallet af sammenligninger. Ved omkring $k = 2^{19}$ begynder kurven at stige. Denne stigning kan skyldes, at prioritetskøen er begyndt at blive så stor, at den ikke kan være i $L2$ cachen og algoritmen begynder, at lave cache misses. På grafen ses også en kurve for antal $L2$ cache misses, hvor hvert målepunkt er divideret med $4 \cdot 10^7$. De 2^{19} tripler udfylder de $4MB$ cache, som der er tilrådighed, hvilket medfører en stigning i tidsforbruget.



(a) variable n, $k = 1000$

(b) variable k, $n = 16384$

Figur 3.1: n^2 priority: grafer

Selvom algoritmen fungerer, har den et meget stort tidsforbrug. Når $n = 16384$ og $k = 2^{27}$ bruger algoritmen over to minutter på at finde løsningen. Det er derfor interessant at finde ud af, hvor al tiden bliver brugt. Et åbenlyst bud er, at algoritmen bruger tiden på at vedligeholde den heap, der bruges i prioritetskøen. Dette bekræftes af en udskrift fra en profiler figur A.2 i appendiks.

De tre største poster fra udskriften viser, at algoritmen bruger størstedelen af tiden på justere heapen og indsætte tripler. Dette er ikke specielt mærkeligt, da dette bliver gjort $O(n^2)$ gange. Den trejdestørste post er sammenligninger af tripler, selvom en sammenligning i sig selv er hurtig, er tidsforbruget højt, fordi der bliver foretaget $O(\log k)$ af dem for hver indsættelse. En åbenlys optimering af algoritmen er, at reducere antallet af sorteringer i køen. Den efterfølgende algoritme forsøger, at reducere antallet af sorteringer.

3.2 n^2 select

Den forrige algoritme brugte en masse energi på, at sortere triplerne løbende, hvilket viste sig ikke at være specielt effektivt. Hvis de $\binom{n}{2} + n$ triple i stedet sorteres en enkelt gang til sidst, vil den hurtigste sorteringsalgoritme, bruge mindst $O(n^2 \cdot \log n)$ sammenligninger på dette.

I stedet for at sortere, er det nok at identificere det $k$te største triple og vælge de tripler som er større end dette. Til dette formål bruges selektion, som bliver diskuteret i afsnit 5.1.

Algorithm 2: n^2 select

```
   input  : A
   output: vector v
 1 for i ← 1 to n do
 2     s ← 0;
 3     for j ← i to n do
 4         s ← s + A[j];
 5         v.insert(i, j, s);
 6     end
 7 end
 8 selection(0, |v|, v, k);
 9 prune(v, 0, k − 1);
10 return v;
```

Korrekthed

Alle $\binom{n}{2} + n$ mulige tripler gennemløbes. Til sidst udvælges de k største via selektion. Korrekthed følger derfor af korrektheden for selektionsalgoritmen.

Kompleksitet

Gennemløbet af de $\binom{n}{2} + n$ mulige delvektorer tager $O(n^2)$ tid. Efter gennemløbet, køres der en selektion, som bruger $O(s)$, på at finde det $k$te største triple, hvor s er antallet af elementer der køres selektion på. Idet antallet af tripler der køres selektion på er $O(n^2)$, bliver den samlede udførelsestid $O(n^2)$. Pladsforbruget er tilsvarende $O(n^2)$.

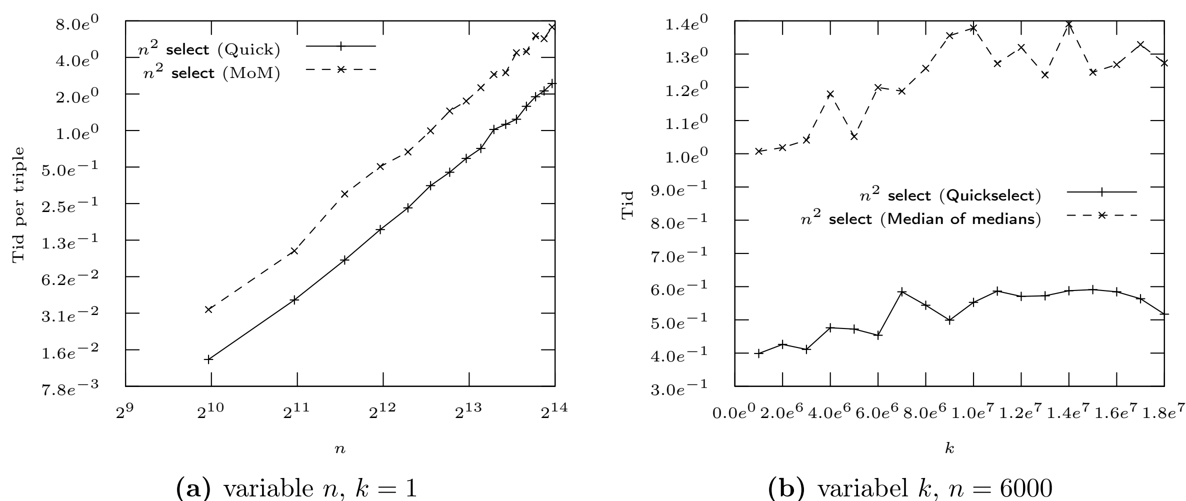
Eksperimenter

Til selektionen vælges Quickselect med P3D strategien afsnit 5.3 og Median of medians fra afsnit 5.4. Det er forventet, at Quickselect klarer sig bedre end Median of medians, da det også er tilfældet i afsnit 5.5.

Figur 3.2a viser kurverne for algoritmen med variabel n. Det er ud fra grafen tydeligt at Quickselect klarer sig bedst for alle værdier af n. Efter $n = 16384$ løber algoritmerne tør for hukommelse, fordi algoritmen skal opbevare alle triplerne i hukommelsen, før selektionen laves. Et triple fylder $12B$ og antallet af tripler i hukommelsen er $\binom{n}{2} + n$, hvilket giver et pladsforbrug på $12B \cdot \binom{16384}{2} \approx 1,5GB$.

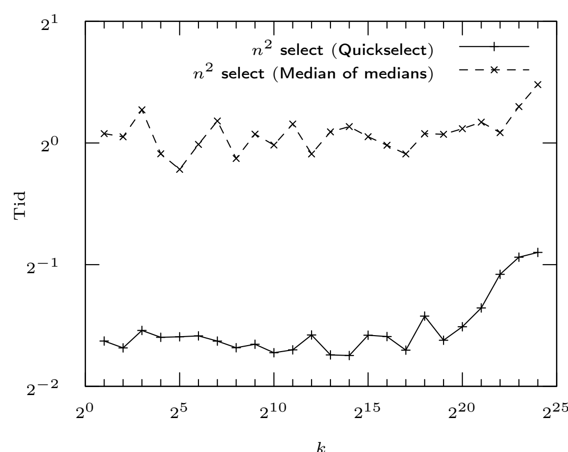
Figur 3.2b viser en variabel k og n fastholdes. Det ses igen tydeligt, at Quickselect er den hurtigste selektionsalgoritme. Tendensen fra selektionsalgoritmerne ses også tydeligt. Kurven for Median of medians viser en stigning, mens Quickselect viser en buet kurve.

Disse tendenser vil højst sandsynligt gå igen ved alle algoritmer, der er baseret på selektionsalgoritmerne. I stedet for at vise grafen med en lineær x-akse, vil k graferne for fremtiden være med logaritmisk akse. Figur 3.3 viser figur 3.2b med logaritmisk x-akse. Det næstsidste målepunkt er midten og buen ses nu som et hak i slutningen af grafen.



(a) variable n, $k = 1$ **(b)** variabel k, $n = 6000$

Figur 3.2: n^2 select: grafer for n og k



Figur 3.3: variable k, $n = 6000$

Algoritmens store problem er, at det ikke er muligt, at have et ret stort n på grund af $O(n^2)$ pladsforbruget. Når n bliver større end 16384 løber computeren tør for hukommelse. Den næste algoritme forsøger at løse dette problem ved, at lave flere selektioner, men med reduceret pladsforbrug.

3.3 n^2 select m

For at imødekomme problemet med pladsforbruget på $O(n^2)$ forsøger denne algoritme, at reducere pladsforbruget. Hovedideen er at køre selektion på samme måde som n^2 select. I stedet for at køre selektionen på $\binom{n}{2} + n$ tripler, køres den på m tripler ad gangen, hvor $k \leq m \leq \binom{n}{2} + n$. Algoritmen er en hybrid imellem n^2 priority og n^2 select. I n^2 priority er $m = k + 1$ og der køres en partiel sortering i stedet for en selektion. I n^2 select er $m = \binom{n}{2} + n$, hvilket medfører, at antallet af selektioner bliver én. Algoritmen n^2 select m forsøger, at finde en værdi af m, som kører godt i praktis, men på samme tid giver et mindre pladsforbrug.

Den første opgave er, at finde en god størrelse for m. Idet m sammen med k afgør, hvor mange gange selektioner der bliver kørt, er det vigtigt, at finde et godt forhold mellem de to parametre. Der er dog en restriktion, idet m bliver nødt til at være en funktion af k. Hvis m bliver mindre end k, vil en selektion efter det $k$te triple ikke give menning. Et åbenlyst bud på m er $m = c \cdot k$, hvor $c > 0$. Dette valg viser sig at være dårligt. Hvis et lille c vælges, vil en lille værdi af k betyde, at antallet af selektioner bliver stort. Selvom selektionsalgoritmen har en lineær udførelsestid, er der stadig forbundet overhead med at køre den. Hvis c derimod er stor og et stort k vælges, bliver problemet med pladsforbruget ikke løst og i værste fald vil det blive værre.

Ideen er, at lade c være forholdsvis lille og lade m have en minimumsværdi. I eksperiment afsnittet forsøges det, at finde et godt m for algoritmen.

Algorithm 3: n^2 select m

```
   input  : A
   output: vector v
 1 for i ← 1 to n do
 2     s ← 0  for j ← i to n do
 3         s ← s + A[j];
 4         v.insert(i, j, s);
 5         if v.size = m then
 6             v ←selection(v,k);
 7             prune(v, 0, k − 1);
 8         end
 9     end
10 end
11 if v.size > k then
12     selection (v,k);
13     prune(v, 0, k − 1);
14 end
15 return v;
```

Korrekthed

Alle $\binom{n}{2} + n$ mulige delvektorer gennemløbes. Hver gang m tripler er fundet, reduceres sættet til k. Til sidst reduceres sættet til k, som giver de k største tripler. Korrekthed følger derfor af selektionsalgoritmen.

Kompleksitet

Gennemløbet af de $\binom{n}{2} + n$ mulige tripler tager $O(n^2)$ tid. Når de første k tripler er udregnet, vil algoritmen lave en selektion $O\big(\frac{n^2}{m-k}\big)$ gange, som hver gang tager $O(m)$ tid. Dette giver en samlet udførelsestid på $O\big(n^2 + \frac{n^2}{m-k} \cdot m\big)$ eller $O(\frac{n^2}{m-k} \cdot m)$, hvor $m > k$. Pladsforbruget er $O(m)$.

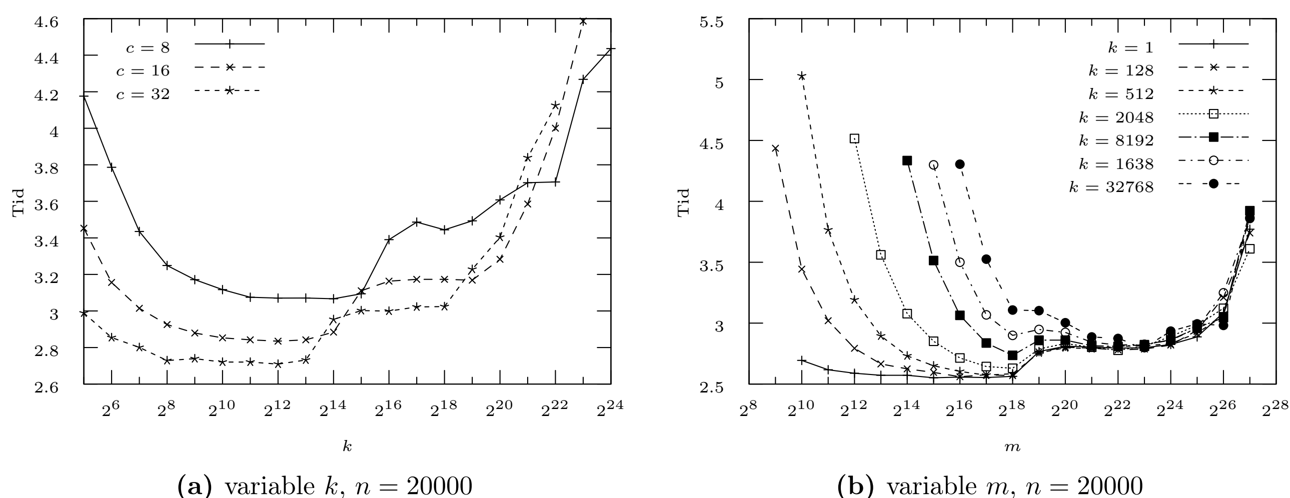
Eksperimenter

Selektionsalgoritmen der bruges i denne algoritme er Quickselect med P3 strategien, selvom P3D er testet i afsnit 5.3 til at være den hurtigeste selektionsalgoritme. Problemet ved P3D er, at de k største tripler ligger i den højre side af m efter en selektion har kørt og de nye tripler indsættes fra venste mod højre. Det vil sige, at det højremeste triple potentielt også er blandt de k største for hele inputtet. For hver gang der vælges et pivot i selektionen, hvis rang ligger mellem k og slutningen af m, vil partitioneringsalgoritmen sørge for, at alle tripler, der har en større rang vil blive flyttet mod højre. Dette betyder, at triplerne fra k til slutningen af m langsomt vil blive sorteret fra største til mindste jo længere algoritmen kører. Som diskuteret i afsnittet 5.3, vil et deterministisk dårligt valg af et pivot, hvor inputtet er sorteret, give en selektionsalgoritme, der bruger $O(s^2)$ tid, hvor s er antallet af elementer der skal laves selektion på. Udførelsestiden for n^2 select m med P3D bliver derfor $O(\frac{n^2}{m-k} \cdot m^2)$. Algoritmen er faktisk så langsom, at det ikke er muligt at teste $k \simeq 10000$ med $n = 20000$. Algoritmen Quickselect med P3 strategien er den selektionsalgoritme som klare sig næstbedst og har den fordel, at den vælger pivotet tilfældigt, så ovenstående undgåes.

Selvom $m = c \cdot k$ er et dårligt valg for enten store eller små værdier af k, er det måske stadig et godt valg, hvis disse ekstremer håndteres. Figur 3.4a viser, hvad der sker ved forskellige valg af c og ved at variere k. I starten er m så lille, at der hele tiden bliver lavet en selektion og det dominerer udførelsestiden. Kurven for $c = 32$ begynder at stige efter $k = 2^{12}$, som svarer til 131072 tripler eller cirka $1.5MB$, hvilket ligger lige under grænsen for $L2$ cachen. De to andre værdier af c viser den samme stigning efter 2^{13} og 2^{14}.

Eksperimentet viser, at jo større m er, jo bedre er det, så længe algoritmen holder sig inde for $L2$ cachen. Når k bliver stor nok, bliver forskellen mellem de forskellige værdier af c mindre.

Denne påstand understøttes af figur 3.4b, der viser en varierende m med forskellige k værdier. Ved $m = 2^{18}$ knækker kurverne som svarer til $3MB$ tripler. Ved $k = 32768$ kan det ikke længere betale sig at have et lille m.

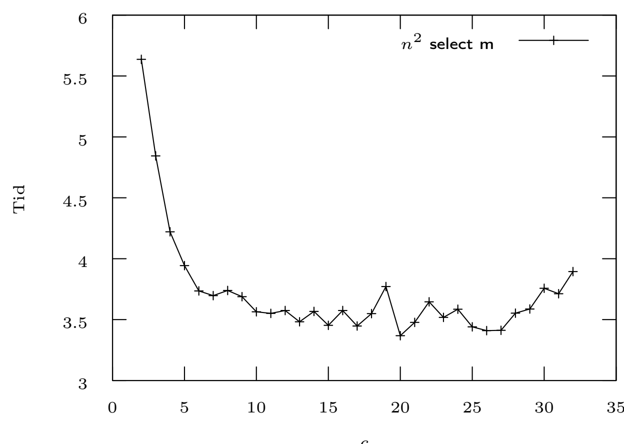


(a) variable k, $n = 20000$

(b) variable m, $n = 20000$

Figur 3.4: n^2 select m: grafer for k og m

Figur 3.5 viser forskellige c-værdier. Ved $c > 16$ er der næsten ingen forskel. For at få et så lille pladsforbrug som muligt, er det en fordel at have et lille c.

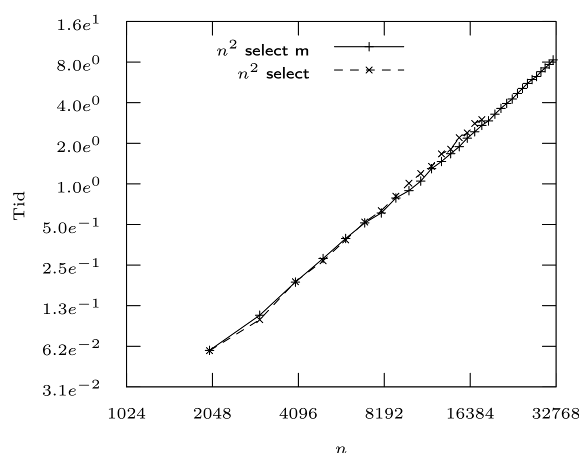


Figur 3.5: n^2 select m: variable c, $n = 20000$, $k = 2^{21}$

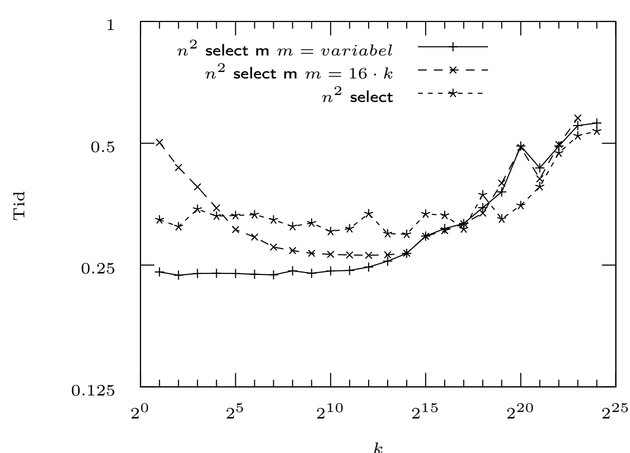
På grundlag af disse resultater, vælges m på følgende måde:

$$m = \begin{cases} 2^{18} & \text{for } 16 \cdot k < 2^{18} \\ 16 \cdot k & \text{for } 2^{18} < 16 \cdot k < \binom{n}{2} + n \\ \binom{n}{2} + n & \text{for } 16 \cdot k \geq \binom{n}{2} + n \end{cases}$$

Ideen er, at have en konstant m når k er lille. De forrige eksperimenter viser, at så længe algoritmen befinder sig i $L2$ cachen er det optimalt at have en m, der er så stor som muligt. Værdien for c vælges til at være 16. Når k er 2^{18} fylder triplerne mere end $L2$ cachen, hvorefter det ikke kan betale sig, at have en konstant m og m sættes til $16 \cdot k$. Når $c \cdot k$ bliver større end $\binom{n}{2} + n$, sættes m til $\binom{n}{2} + n$, da der ikke er nogen grund til at allokere hukommelse til flere tripler end hvad der er muligt at producere. Figur 3.6a viser



(a) variable n, $k = 524288$



(b) variable k, $n = 6000$

Figur 3.6: n^2 select m: grafer for n og k

forskellen mellem n^2 select og n^2 select m. Værdien af k er valgt så m er $16 \cdot 524288$.

Når n bliver stor begynder n^2 select m at overhale n^2 select. Selv om n^2 select m klarer sig bedre end n^2 select, er den praktiske udførelsestid for algoritmen så stor, at det tager for lang tid at teste flere målepunkter.

Figur 3.6b viser grafen for variabel k. Foruden n^2 select og n^2 select m med den beskrevne strategi ($m = variable$), vises algoritmen også med en statisk $m = 16 \cdot k$ strategi for at tydeliggøre forskellen.

I starten er den statiske strategi meget langsommere end den variable. Ved $k = 2^{20}$ er n^2 select marginalt hurtigere end n^2 select m. Dette skyldes at n^2 select m bruger en smule resourcer på at vedligeholde de m tripler. Det interessante er, at selvom n^2 select kun laver en selektion, er n^2 select m meget hurtigere for små k, hvor den skal lave flere selektioner. Det vil sige, at det bedre kan betale sig, at lave mange selektioner på mindre dele af inputtet, end en enkelt på hele inputtet. Dette skyldes, at når k er lille kan hele selektionen foregå i $L2$ cachen. Udfra forsøgene kan det konstateres, at n^2 select m generelt klarer sig bedre end n^2 select.

Fordi n^2 select m laver mange selektioner når k, er lille vil det måske være en fordel, at reducere antallet af disse. Figur A.4 i appendiks viser, at de mange partitioneringer er en væsenlig faktor i udførelsestiden. Ved at reducere antallet af selektioner, vil antallet af partitioneringer også falde. Hvilket giver anledning til næste algoritme.

3.4 n^2 historik

For at reducere antallet af selektioner, reduceres antallet af tripler, der skal køres selektion på. Observationen er, at når de første k tripler er fundet, vil de efterfølgende tripler kun være kandidater til at være blandt de k største, såfremt deres sum er større end eller lig med det triple med den mindste sum af de hidtil fundet. Det er derfor muligt at smide de tripler væk, som alligevel ikke er mulige kandidater, før der laves en selektion på dem. Dermed reduceres antallet af selektioner.

Ved hele tiden at holde styr på, hvad det mindste triple er, kan algoritmen løbende kassere tripler, som har en mindre sum end dette. De første k tripler skal gemmes, da der ikke er nogen garanti for, at de ikke er blandt de k største. Efterfølgende indsættes et triple kun i m, hvis det er større end eller lig med det hidtil mindste. Efter en selektion har kørt, vil det mindste triple være i den $k$te indgang i m og alle de næste tripler sammenlignes med dette.

Korrekthed

Algoritmen starter med at finde k tripler. Efter disse er fundet undlader algoritmen, at indsætte de tripler, som har en mindre sum end det mindste hidtil fundet. Disse tripler kan ikke være kandidater til de k største, da der allerede findes k større tripler. Når der er fundet m tripler laves der en selektion og de $m - k$ mindste tripler kasseres. Den mindste kandidat må være det $k$te største triple i m og være på den $k$te plads. Når alle $\binom{n}{2} + n$ tripler er gennemløbet ryddes der op, så kun k tripler bliver tilbage. Algoritmen kasserer med andre ord kun et triple, hvis og kun hvis, der findes mindst k tripler, som er større.

Algorithm 4: n^2 historik

```
   input  : A, k
   output: v
 1 minSum = −∞;
 2 for i ← 1 to n do
 3 |  s ← 0;
 4 |  for j ← i to n do
 5 |  |  s ← s + A[j];
 6 |  |  if v.size < k or sum > minSum then
 7 |  |  |  v.insert(i, j, sum);
 8 |  |  |  if s < minSum then
 9 |  |  |  |  minSum ← s;
10 |  |  |  end
11 |  |  |  if v.size = m then
12 |  |  |  |  v =selection(v,k);
13 |  |  |  |  prune(v, 0, k − 1);
14 |  |  |  |  minSum ← v[k];
15 |  |  |  end
16 |  |  end
17 |  end
18 end
19 if v.size > k then
20 |  v ←selection(v,k);
21 |  prune(v, 0, k − 1);
22 end
23 return v[k];
```

Kompleksitet

Det viser sig at analysen for n^2 historik ikke er så enkel. Problemet med analysen er, at det er svært at beregne en nedre grænse for, hvor meget der rent faktisk bliver sparet ved at undlade triplerne. I tidligere forsøg vises det, at selektionerne er en dominerende faktor i udførelsestiden. Jo færre tripler der indsættes i m, jo færre selektioner laves der.

Grunden til at det er svært, at beregne hvor mange selektioner der fortages er, at algoritmen kun holder styr på det mindste triple og ikke de k største tripler. Der køres altså selektioner på tripler blandt de m største, men disse kan godt være mindre end de k største.

Hvis man i stedet betragter algoritmen, som en stor prioritetskø, bliver analysen lidt mere håndgribelig. Hver triple bliver indsat, hvis det er større end eller lig med det mindste i køen og det mindste fjernes på samme tid.

Sandsynligheden for at et triple skal indsættes i prioritetskøen er $\frac{k}{s}$, hvor s er antallet af tidligere inspicerede tripler, da det antages at fordelingen af inputtet er uniform. I en

prioritetskø med plads til k tripler er det forventede antal indsættelser i køen derfor:

$$\begin{aligned} & k + \sum_{s=k+1}^{n^2} \frac{k}{s} \\ = \; & k + k(\sum_{s=1}^{n^2} \frac{1}{s} - \sum_{s=1}^{k} \frac{1}{s}) \\ = \; & k + k(\ln(n^2) - \ln(k)) \\ = \; & O(k \ln n) \quad \square \end{aligned}$$

Forskellen mellem prioritetskøen og n^2 historik er, at køen kun behandler k tripler, mens der er plads til m tripler i n^2 historik. Det vil sige, at der er $m - k$ indgange i m som der ikke redegøres for. Det reelle antal indsættelser ligger et sted mellem $O(k \ln n)$ og $O(m \ln n)$. Hvilket er en stor besparelse i forhold til de $O(n^2)$ for n^2 select m.

I værste tilfælde burde den praktiske udførelsestid for n^2 historik, være enten den samme eller en smule dårligere end n^2 select m. Hvis alle tripler er sorteret, så vil de blive indsat fra mindste til største og n^2 historik vil blive nødt til lave selektion på alle tripler præcist som i n^2 select m, men n^2 historik bruger stadig ressourcer, på at sammenligne et givent triple med den hidtil mindste, hvilket bevirker at algoritmen bliver en smule langsommere.

Når k kommer tæt på n^2 vil antallet af besparelser også falde og udførelsestiden vil stige. Når k bliver stor nok vil antallet af tripler, som er mindre end den mindste svinde. Når k er n^2 er der ingen tripler at kassere og udførelsestiden vil igen blive en smule langsommere end n^2 select m.

Eksperimenter

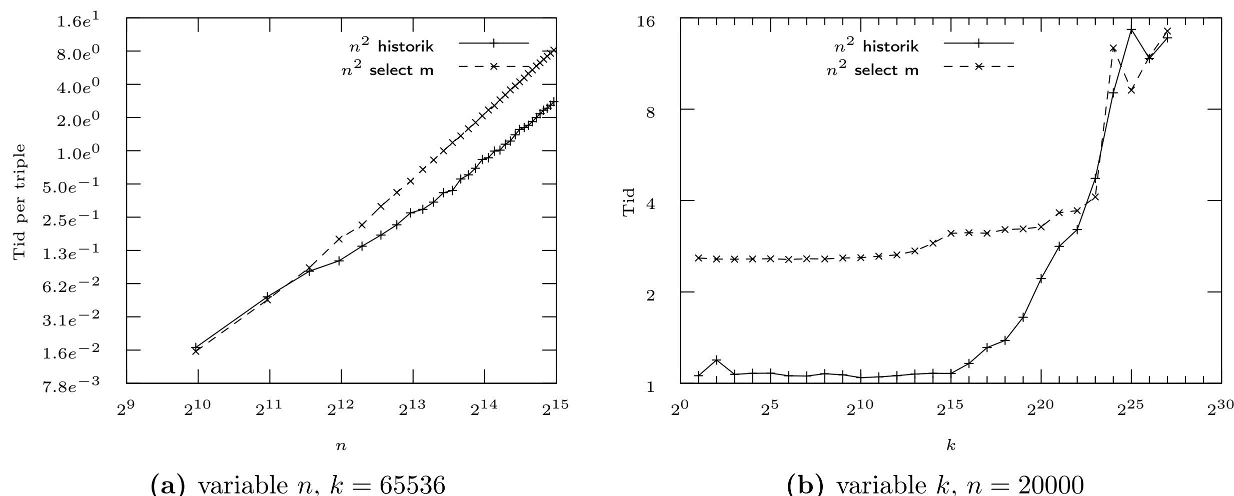
Figur 3.7a viser udførelsestiden for n^2 historik og n^2 select m med variable n. Besparelsen har en tydelig indvirkning på udførelsestiden.

Figur 3.7b viser som forventet at udførelsestiden bliver dårligere for n^2 historik jo større k er. I slutningen af kurven klarer n^2 select m sig marginalt bedre end n^2 historik, men forskellen indtil da er meget stort.

Profiler udskriften, som ses i figur A.5 appendiks, viser nu at antallet af selektioner er reduceret som forventet. Udfra udskriften ses det også, at det meste af tiden nu bliver brugt på at gennemløbe input og beregne tripler. Det kan derfor være en fordel at optimere gennemløbet af inputtet. Den næste algoritme forsøger at gøre gennemløbet mere I/O effektivt.

3.5 n^2 block

Traditionelt set beregnes den teoretisk udførelsestid for en algortime i antallet af primitive operationer, f.eks. antallet sammenligninger eller additioner. I den I/O baserede model beregnes udførelsestiden for en algoritme i antallet af læsninger eller skrivninger mellem to typer lager f.eks. mellem $L1$ cachen og hukommelsen. Når et lager skal læse eller skrive



(a) variable n, $k = 65536$

(b) variable k, $n = 20000$

Figur 3.7: n^2 historik: grafer for n og k

fra et andet lager, foregår det typisk, ved at læse en blok bytes ind ad gangen. Størrelsen af denne blok betegnes med B. Størrelsen på et lager er M.

Måden algoritmerne indtil nu har gennemløbet inputtet, er fra et I/O synspunkt yderst ineffektivt. Gennemløbet konstruerer et triple ved at finde summen mellem i og j, hvor $i \leq j$. For hver værdi af i, går j fra i til $|A|$ og $|A| - i$ inputs bliver læst ind i lageret. For hver Bte input skal der læses en ny blok af elementer ind. Antallet af læsninger for gennemløbet bliver derfor $O(\frac{n^2}{B})$.

Formålet med denne algoritme er, at reducere antallet af læsninger der skal bruges. Problemet med måden at gennemløbe inputtet på er, at alle de inputs j er med til at indlæse kun bruges en gang og derefter kasseres de, for at gøre plads til de næste. For at optimere antallet af læsninger, opdeles inputtet i blokke af størrelse M. Inputtet gennemløbes nu på samme måde som før, men i stedet for at lave et triple ud af den $i$te og den $j$te indgang, udregnes alle tripler, der starter i den $i_b$te blok og slutter i den $j_b$te blok.

Algorithm 5: n^2 block

input : A, k
output: v

1 **for** $i_b \leftarrow 0$ **to** $\frac{n-1}{M}$ **do**
2 | **for** $j_b \leftarrow +$ **to** $\frac{n-1}{M}$ **do**
3 | | **for** $i \leftarrow i_b \cdot (M+1)$ **to** $(i_b + 1) \cdot M$ **do**
4 | | | **for** $j \leftarrow j_b \cdot (M+1)$ **to** $(j_b + 1) \cdot M$ **do**
5 | | | | Beregn sum, og gem bedste tripler.
6 | | | **end**
7 | | **end**
8 | **end**
9 **end**

Kompleksitet

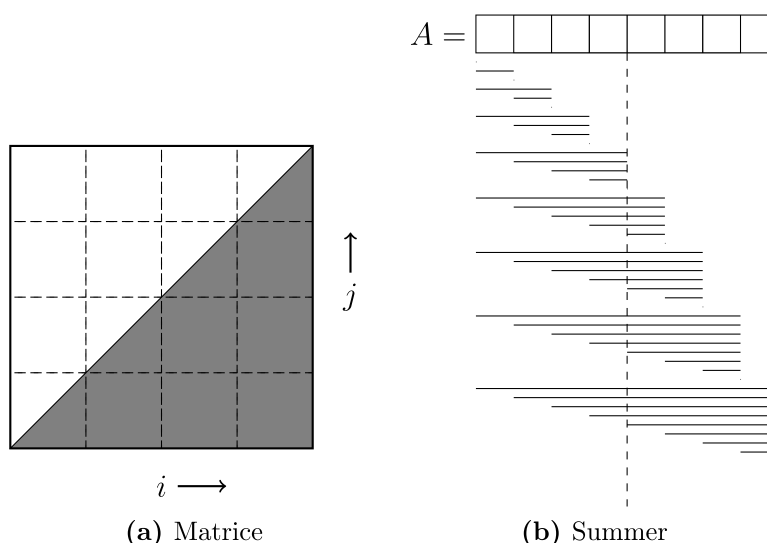
For hver blok af størrelse M bruger algoritmen $\frac{M}{B}$ indlæsninger. Der er i alt $\frac{n}{M}$ blokke, hvilket giver $O\big(\frac{M}{B} \cdot (\frac{n}{M})^2\big) = O\big(\frac{n^2}{B \cdot M}\big)$ læsninger, altså en drastisk forbedring i forhold til $O\big(\frac{n^2}{B}\big)$.

Implementation

Selvom algoritme 5 er korrekt, vil en naiv implementation give en langsom algoritme. Der bruges $O(\frac{n^2}{M})$ divisioners operationer på at holde styr på blokkene, samt $O(n^2)$ multiplikationer på, at finde de rigtige offsets i A. Desuden skal der løbende holdes øje med om i_b og j_b er de sidste blokke. Det næste problem er, at udregne summerne for triplerne. I det almindelige gennemløb er dette en forholdvis nem opgave, idet summen fra i til j kan opbygges trinvist. Problemet ved det I/O effektive gennemløb er, at summen for et givent triple består af summen af tre dele.

- Summen fra i til $i_b \cdot M$.
- Summen af blokkene $i_b + 1$ til $j_b - 1$.
- Summen fra $j_b \cdot M$ til j.

Disse tre summer kræver også en del bogføring at realisere, som igen er dyrt. Løsningen på disse problemer er, at omskrive måden at gennemløbe A på. Den første observation er, at gennemløbet kan ses som en matrice, som opdeles, som ses i figur 3.8a. Den nedereste grå halvdel svarer til $j > i$ som ikke er interessant. De stiplede linjer viser de forskellige blokke. Hvergang en linje passeres enten i i eller j aksen, svarer det til at algoritmen skifter blok.



(a) Matrice **(b)** Summer

Figur 3.8: n^2 block: gennemløb

Algoritmen starter nederst til venstre. Hvor i løber fra venstre mod højre. Hvis i enten møder en vertikallinje eller det grå område tælles j op. Når j rammer toppen af matricen, starter algoritmen i den næste kolonne. Gennemløbet parrer alle blokke med alle andre

præcist som før, men fordelen er, at det ikke længere er nødvendigt med fire for-løkker og udregningen af offsets er lige til. Det viser sig, at det på samme tid er meget nemt, at finde summerne for triplerne løbende. Figur 3.8b viser den rækkefølge de to første blokkes summer kommer i når M er 4. Den stiplede linje viser blokkene. Hver horisontale linje svarer til en sum. Udfra tegningen ses det, at summen kan beregnes som følger: Summen er summen fra starten af den $i_b$te blok til j. Hver gang i tælles op svarer det til, at der skal trækkes $A[i]$ fra summen. Hver gang j tælles op svarer det til, at der bliver lagt $A[j]$ til alle efterfølgende tripler. Når $j = |A|$, skal i starte i den næste kolonne, som realiseres ved, at rykke is startpunkt med M. Koden for algoritmen ses som algoritme 6.

Algorithm 6: n^2 block: detalieret

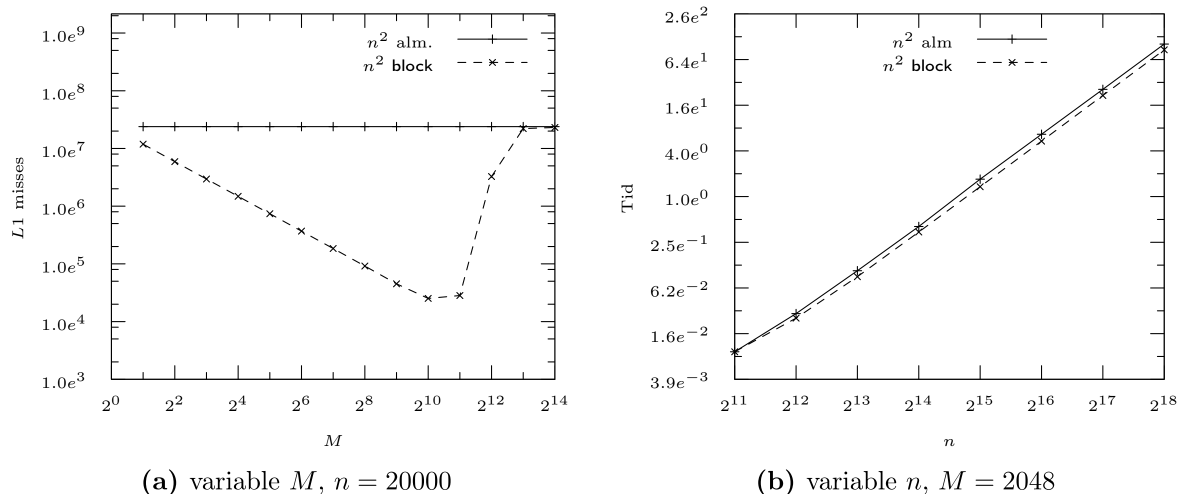
```
   input  : A,k
   output: v
1  while i_blockStart < |A| do
2      j_blockSum ← 0;
3      for j ← i_blockStart to |A| do
4          j_blockSum ← j_blockSum + A[j];
5          for i ← i_blockStart to i_blockEnd and i ≤ j do
6              triplet kan findes her som: (i, j, i_blockSum)
7              i_blockSum ← i_blockSum − A[i];
8          end
9      end
10     i_blockStart ← i_blockEnd;
11     i_blockEnd ← min(i_blockEnd + M, |A|);
12 end
```

Eksperimenter

Det første eksperiment skal afgøre, hvad størrelsen på M skal være for at minimere antallet af cache misses. For ikke at få for mange forstyrrelser fra n^2 historik, er det kun gennemløbet der testes. For at sørge for, at compileren ikke optimerer gennemløbet væk, returneres den sidste sum. Inputtet i forsøget har typen double og det er derfor forventet, at der kan være 4096 inputs i $L1$ cachen, før der kommer misses. Fordi den $i_b$te blok parres med den $j_b$te er det forventet, at den optimale størrelse for M, er halvdelen af hvad der kan være i $L1$ cachen, altså 2048.

Figur 3.9a viser grafen for dette forsøg. Ved at vælge den rigtige M burde algoritmen spare n cache misses, da gennemløbet fastholder M inputs og finder alle mulige summer ud fra dette. Eksperimentet viser at $M = 2048$ er det optimale, men antallet af cache misses er langt under det forventede. Dette skyldes sandsynligvis, at det er meget nemt for processoren at gætte hvilket input, der skal bruges næste gang og indlæser dette, før der kommer en cache miss.

Figur 3.9b viser udførelsestiden for de to forskellige måder at gennemløbe inputtet på. Forskellen mellem den normale og den I/O effektive måde at gennemløbe inputtet på, er minimal. Dette skyldes, at selvom der er en besparelse på faktor M, bruges der en masse energi på, at gennemløbe inputtet mere I/O effektivt. Hvis inputtet havde været

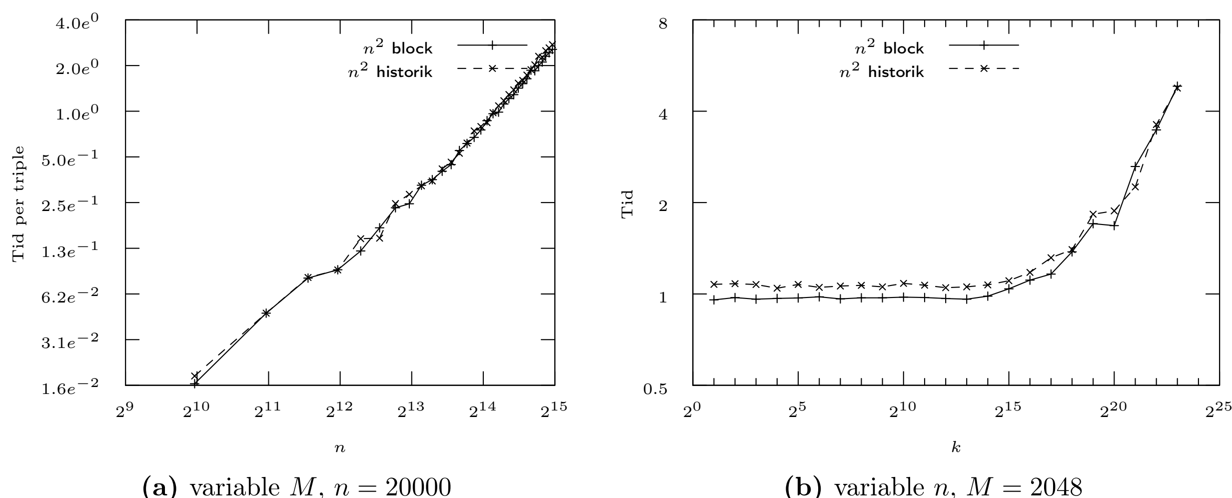


(a) variable M, $n = 20000$

(b) variable n, $M = 2048$

Figur 3.9: n^2 block: grafer for gennemløb med variable M og n

større, ville der være en reduktionen af læsninger mellem harddisken og hukommelsen, eller mellem hukommelsen og $L2$ cachen og derved ville reduktionen i tidsforbruget nok have været en del større. For at algoritmen skal komme ud af $L2$ cachen kræver det at n er større end $\approx 18 \cdot 10^4$. Til sidst vises forskellen mellem de to måder at gennemløbe inputtet på, ved at udvælge triplerne som i n^2 historik. Forskellen ses i figur 3.10 og er minimal.



(a) variable M, $n = 20000$

(b) variable n, $M = 2048$

Figur 3.10: n^2 block: grafer for gennemløb og selektion med variable n og k

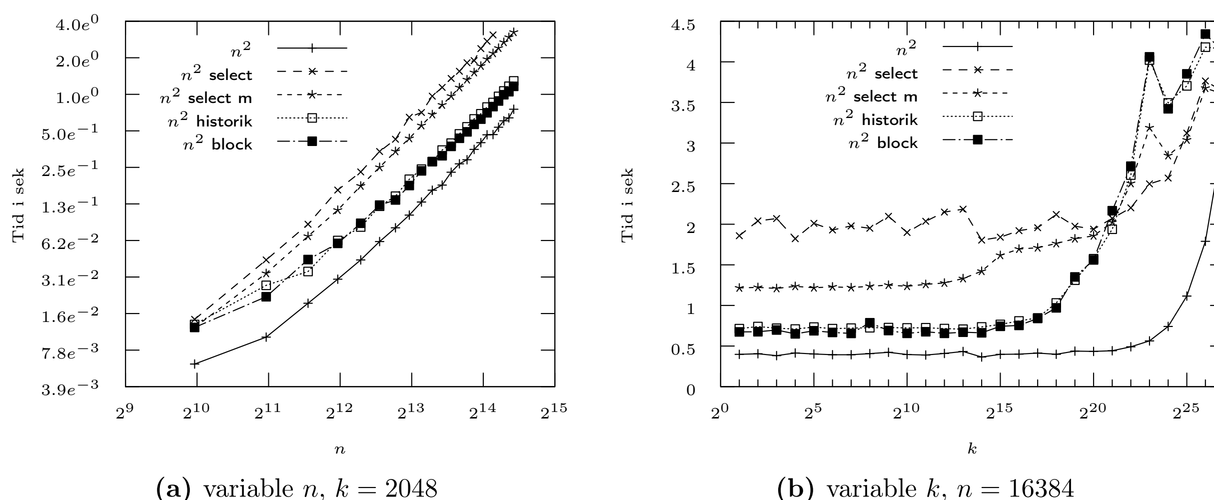
3.6 Konklusion på n^2-familien

I de forrige afsnit er der gennemgået forskellige algoritmer til at løse k maksimum delsum problemet. Fælles for dem alle er, at de gennemløber inputtet og finder de k største udfra de $\binom{n}{2} + n$ tripler inputtet genererer. I figur 3.11 ses de forskellige algoritmer holdt op mod hinanden. På grafen ses der også et et ekstra plot, kaldt n^2. Kurven for n^2 er tiden

for at gennemløbe inputtet, samt tiden for at outputte de k første tripler. Med andre ord det optimale en algoritme i n^2-familien kan opnå.

Figur 3.11b viser kurven for variabel k. I figuren er n^2 priority undladt, da den er for langsom. Når k er lille er n^2 block meget tæt på det optimale. Der er to hovedårsager til, at algoritmerne klarer sig dårligt ved stort k. Den første er, at når k er stor bliver algoritmerne nødt til, at holde styr på mindst k tripler. En mulig løsning kunne være, at når $k > \frac{\binom{n}{2}+n}{2}$, så findes de k mindste tripler i stedet og værdien af det største af de mindste tripler gemmes. Derefter gennemløbes inputtet igen. Alle tripler, som har en værdi, der er større end eller lig med det gemte, er kandidater til at være de k største. Til sidst laves en selektion for at finde de k største. Problemet med denne løsning er, at selvom udførelsestiden muligvis forbedres for den øverste halvdel af spektret, så vil den høje udførelsestid stadig optræde omkring midten, som ses på figur 3.11b. Figur 3.11a, viser kurverne for variable n. Kurven for n^2 block er ret tæt på det optimale. Forskellen mellem n^2 select og n^2 historik på det største punkt er cirka faktor 12 og forskellen mellem n^2 historik og det optimale er cirka faktor 1,3.

Af de implementerede er n^2 block den bedste og vil fremover repræsentere n^2-familien.



(a) variable n, $k = 2048$

(b) variable k, $n = 16384$

Figur 3.11: n^2-familien

Kapitel 4

n-familien

Den store ulempe ved n^2-familien er, at lige meget, hvor mange optimeringer der laves, er det kun konstanter der bliver reduceret. I alle de forrige algoritmer gennemløbes $\binom{n}{2} + n$ delvektorer og der beregnes lige så mange tripler. Som nævnt i kapitel 1, er der en række algoritmer der viser, at man faktisk ikke behøver at beregne alle delvektorerenes summer eksplicit for at finde de k største tripler.

I [1] løses k maksimum delsum i $O(n+k)$ tid og $O(k)$ plads. Denne løsning er optimal, i den forstand, at enhver algoritme vil bruge mindst $\Omega(n)$ tid på, at læse inputtet og returnere outputtet som er k stort. Algoritmen beskrevet i artiklen er den, som algoritmerne i dette kapitel er baseret på.

4.1 Suffiks mængder

Hovedideen i algoritmen er, at det ikke er nødvendigt at gennemløbe alle $\binom{n}{2} + n$ delvektorer for at opbygge de k største tripler. I stedet opbygges der en heap i $O(n)$ tid, der indeholder alle $\binom{n}{2} + n$ tripler, hvorefter de k tripler med den største sum hentes ud fra denne.

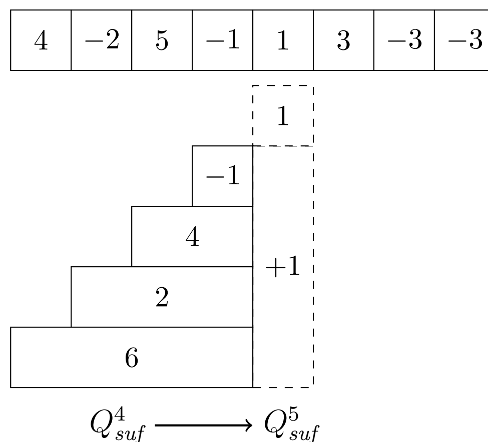
De $\binom{n}{2} + n$ mulige tripler, kan grupperes efter deres slut indeks i mængder. Den mængde med tripler der slutter på j, kalder vi for suffiks mængden Q^j_{suf} og kan beskrives på følgende måde:

$$Q^j_{suf} = \{(i, j, sum_{i,j}) \mid 1 \leq i \leq j \wedge sum_{i,j} = \sum_{s=i}^{j} A[s]\}$$

Mængden Q^j_{suf} kan konstrueres ud fra Q^{j-1}_{suf} som følger:

$$Q^j_{suf} = \begin{cases} \{(1, 1, A[1])\} & \text{for } j = 1 \\ \{(j, j, A[j])\} \cup \{(i, j, sum_{i,j-1} + A[j]) \mid (i, j-1, sum_{i,j-1}) \in Q^{j-1}_{suf}\} & \text{for } j > 1 \end{cases} \quad (4.1)$$

Mængden Q^j_{suf} er altså en forening af de tripler, som starter og slutter i j samt alle tripler i Q^{j-1}_{suf}, hvor $A[j]$ er lagt til deres sum. Figur 4.1 illustrere opbygningen af Q^5_{suf} fra Q^4_{suf}. Først lægges der 1 til alle triplerne fra Q^4_{suf} som sammen med triplet $(5, 5, 1)$ kommer til at udgøre Q^5_{suf}.



Figur 4.1: Q^5_{suf} konstrueres fra Q^4_{suf}

Udfra suffiks mængderne kan der nu konstrueres en forening af disses mængder:

$$Q = \bigcup_{j=1}^{n} Q^j_{suf}$$

En søgning efter de k største tripler, besvares ved at finde de k største tripler i Q.

Det er dog ikke oplagt, hvordan denne søgning skal foretages og det er heller ikke oplagt hvordan konstruktion af Q kan opnå den ønskede tidskompleksitet.

Konstruktion af H

Ideen er at opbygge en heap H på samme måde som Q. Hvis H skal opbygges som Q, skal suffiksmængderne $Q^1_{suf}, \ldots, Q^n_{suf}$ repræsenteres som undertræerne $H^1_{suf}, \ldots, H^n_{suf}$ i heapen. En eksplict konstruktion af disse undertræer vil tage $O(n^2)$ tid, men kan ved passende valg af datastruktur gøres i $O(n)$ tid.

Den første observation er, at ligning 4.1 kan omskrives.

Først ekspanderes $sum_{i,j}$:

$$\begin{aligned} sum_{i,j} &= \sum_{s=i}^{j} A[s] \\ &= \Big(\sum_{s=1}^{j} A[s]\Big) - \Big(\sum_{s=1}^{i} A[s]\Big) \end{aligned}$$

og en ny konstant δ_j defineres:

$$\delta_j = \begin{cases} 0 & \text{for } j = 0 \\ \sum_{i=1}^{j} A[i] & \text{for } j > 0 \end{cases}$$

Ligning 4.1 omskrives:

$$Q^j_{suf} = \{(j, j, A[j])\} \cup \{(i, j, sum_{i,j-1} + A[j]) \mid (i, j-1, sum_{i,j-1}) \in Q^{j-1}_{suf}\}$$

$$= \bigcup_{i=1}^{j} \{(i, j, \delta_{j-1} - \delta_{i-1} + A[j])\}$$

Ideen er nu at trække δ_j fra alle triple i Q^j_{suf}.

$$Q'^j_{suf} = \bigcup_{i=1}^{j} \{(i, j, -\delta_{i-1})\} \tag{4.2}$$

da

$$\delta_j = \delta_{j-1} + A[j]$$

Udfra ligning 4.2 ses det, at Q^j_{suf} kan repræsenteres af par $\langle \delta_j, H^j_{suf} \rangle$, hvor H^j_{suf} er en heap, hvor mængden $\{-\delta_0, \ldots, -\delta_{j-1}\}$ er indsat.

Konstruktionen af H^j_{suf} kræver j indsættelser, hvis H^j_{suf} skal konstrueres fra bunden, hvilket tager $O(n^2)$.

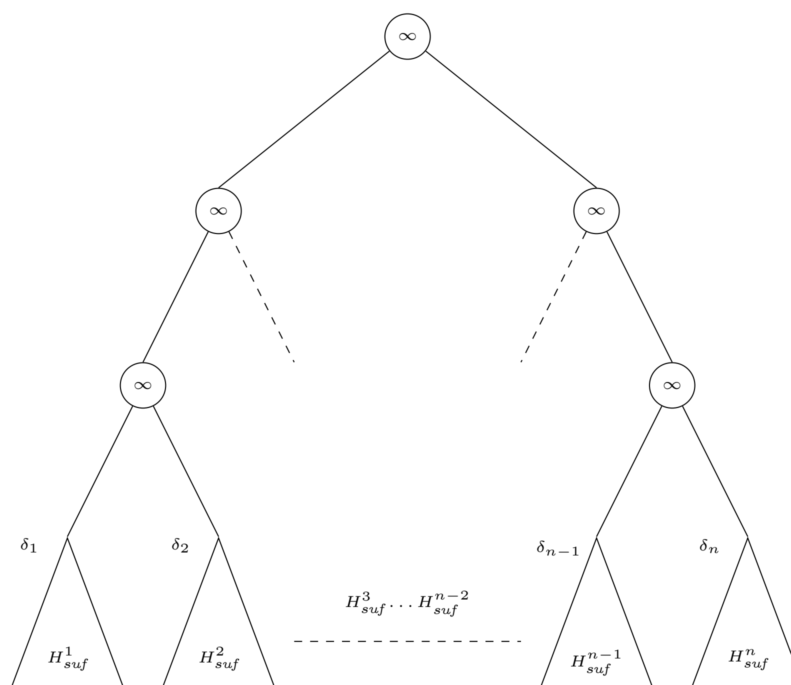
Ideen er nu at bruge Persistent lHeap fra afsnit 6.2.2 til at repræsentere undertræerne $H^1_{suf}, \ldots, H^n_{suf}$. Undertræet H^j_{suf} repræsenteres med den $j$te version af en Persistent lHeap. Når et element indsættes i den $i$te version, kommer den til at repræsentere alle de tripler der starter i i. Fordi dette element også kan tilgåes fra senere versioner, er det vigtigt at gemme den version elementet blev indsat i, hvis et triple skal kunne genskabes. F.eks. triplet $(3, 9, 5)$ findes i H^9_{suf} som elementet $\langle 3, -\delta_8 \rangle$. Den trinvise konstruktion af parret $\langle \delta_j, H^j_{suf} \rangle$ er som følger:

$$\langle \delta_0, H^0_{suf} \rangle = \langle 0, \emptyset \rangle \tag{4.3}$$

$$\langle \delta_j, H^j_{suf} \rangle = \langle \delta_{j-1} + A[j], H^{j-1}_{suf} \cup \{\langle -\delta_{j-1}, j \rangle\} \rangle \tag{4.4}$$

Konstruktionen af parrene $\langle \delta_0, H^0_{suf} \rangle, \langle \delta_1, H^1_{suf} \rangle, \ldots, \langle \delta_n, H^n_{suf} \rangle$, tager derfor $O(n)$, da en indsættelse i Persistent lHeap, ifølge afsnit 6.2.2 tager $O(1)$ og konstanterne $\delta^0, \delta^1, \ldots, \delta^n$ kan findes ved at gennemløbe inputtet A.Til sidst konstrueres H som illusteres i figur 4.2.

Heapene $H^1_{suf}, \ldots, H^n_{suf}$ indsættes som undertræer og forbindes med $n - 1$ knuder, som har værdien $-\infty$.



Figur 4.2: Heapen H

Søgning i H

Til søgning i H bruges Fredriksons binærheap selektionsalgoritme [12]. Udførelsestiden for selektionen er $O(s)$, hvor s er antal elementer i heapen der skal findes. For at finde de k største tripler, søges der efter de $(n-1)+k$ største elementer i H. Når Fredriksons algoritme kommer ind i undertræet H^j_{suf} svarer det til, at algoritmen kommer ind i den $j$te version af Persistent lHeap. Hver gang en knude i den $j$te version skal bruges, beregnes knudens sum ved at lægge δ^j til knuden. Når selektionen har returneret de $(n-1)+k$ elementer vil de $n-1$ af dem være knuderne med ∞, som kasserses.

Den samlede udførelsestid for algoritmen bliver derfor på $O(n+k)$.

I de næste afsnit implementeres tre forskellige algoritmer, der i hovedtræk følger denne ide, fælles for dem er, at de har en dårligere teoretisk udførelsestid, men undgår Fredriksons konstruktion. Problemet med Fredrikson er at algoritmen er forholdsvis kompliceret. Den første algoritme n priority bruger, i stedet for Fredrikson, en prioritetskø for at finde de k største tripler. Den anden algoritme n searchtree skifter Persistent lHeap ud med de to forskellige versioner af persistente søgetræer fra afsnit 6.3. Til sidst implementeres der en udgave, hvor den persistente heap er erstattet med en prioritetskø.

4.2 n priority

Den første algoritme i n-familien bruger en prioritetskø til at finde de k største tripler. Ideen er, at konstruere parrene $\langle \delta_1, H^1_{suf} \rangle, \ldots, \langle \delta_n, H^n_{suf} \rangle$ som i forrige afsnit. I stedet for at konstruere H indsættes rødderne fra de forskellige versioner af Persistent lHeap og de

tilhørende delta-værdier i en prioritetskø. Husk på, at knuderne i heapen er par $\langle -\delta^{i-1}, i \rangle$. En triple kan konstrueres som $(i, j, -\delta_{i-1} + \delta_j)$ og ordenen i prioritetskøen afgøres af *sum*-delen i de tripler der kan produceres. For at finde de k største tripler, tages den største rod, r, ud af køen. Det triple, roden kan producere, er det største triple i en suffiksmængde, men på samme tid, er det også større end alle de andre rødder i køen. Af den grund må det triple være det største af alle. Det næststørste triple findes som et undertræ til r eller også findes det i køen. De to undertræer left og right indsættes derfor i køen og det næste triple trækkes ud. På den måde findes de k største tripler.

Algorithm 7: n priority

```
   input  : A, k
   output: v
 1 H^1_suf, ..., H^n_suf ← constructHeaps(A);
 2 Priorty Queue q;
 3 for i ← 1 to n do
 4 │  q.insert(H^i_suf.getRoot());
 5 end
 6 for i ← 1 to k do
 7 │  r ← q.removeMax();
 8 │  v.insert(r);
 9 │  q.insert(r.left);
10 │  q.insert(r.right);
11 end
12 return v
```

Kompleksitet

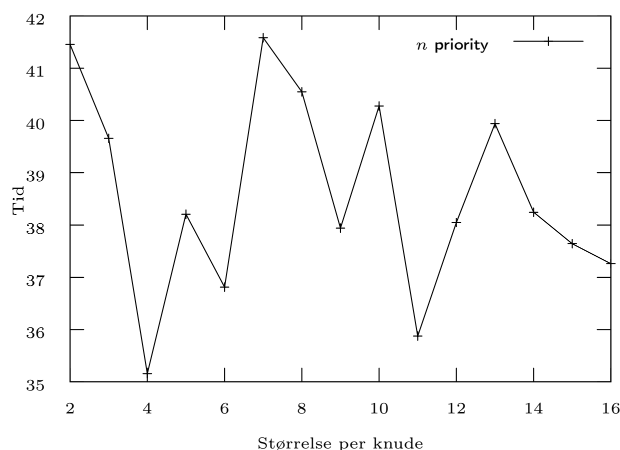
Først bruger algoritmen $O(n)$ på at konstruere de heapen. Hver gang køen trækker en rod ud, indsættes der muligvis to nye rødder i køen. Antallet af rødder i køen stiger derfor med en hver gang et af de k største tripler findes. Når det sidste triple er fundet vil køen bestå af $n + k$ rødder. Udførelsestiden for den sidste udvælgelse bliver derfor $O(\log k)$, som er det samme som $O(\log n)$, da $k = O(n^2)$, hvilket giver en total udførelsestid på $O(n + k \log n)$.

Konstruktionen af heapen bruger $O(n)$ plads og prioritetskøen bruger $O(k)$ plads efter det sidste triple er fundet. Dette giver tilsammen et pladsforbrug på $O(n + k)$

Eksperimenter

I afsnit 6.2.2 argumenteres der for, at størrelsen på right kan have en indflydelse på udførelsestiden. Det er forventet, at en lille knudestørrelse giver flere kopieringer, hvilket giver en længere udførelsestid for konstruktionen af heapen. På samme tid er det forventet at udførelsestiden for et gennemløb af heapen tager kortere tid. Da grafen er forholdsvis lille i forhold til hvor stor k kan blive er det forventet, at en lille knudestørrelse er bedre end en stor.

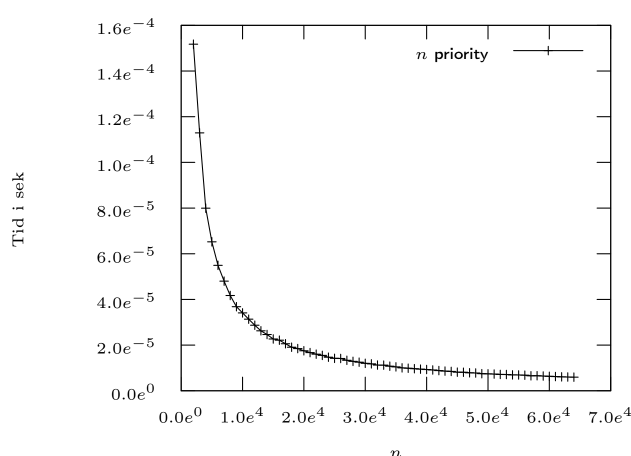
Figur 4.3 viser udførelsestiden med en fast n og k, men hvor størrelsen af right variablerne varieres. Kurven viser, at der ikke er nogen speciel fordel ved at have en stor knudestørrelse. Når right er 4 er udførelsestiden mindst og derfor vælges denne størrelse som den bedste.



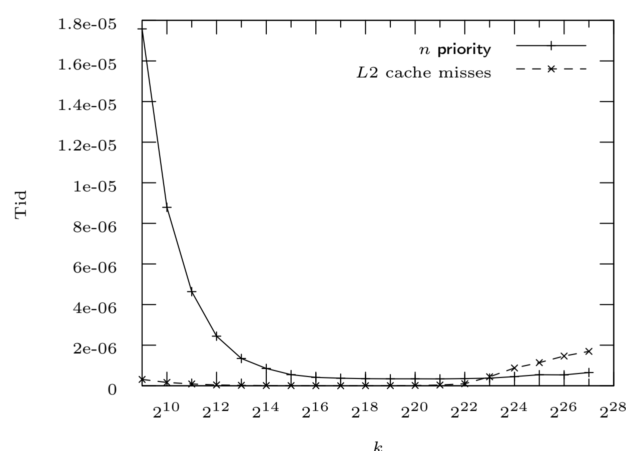
Figur 4.3: n priority: graf knude størrelse, $n = 20000$ og $k = 67108864$

Figur 4.4a viser udførelsestiden for n^2 priority, hvor n varieres. Da den forventede udførelsestid er $O(n + k \log n)$ og k er konstant, undersøges algoritmens udførelsestid med ratio test $f(n) = n + \log n$. Selvom kurven ikke konvergerer helt mod en konstant, er det tæt på. Det er ikke muligt at komme længere ud, da n er 64000 og triplernes indicer er begrænset af størrelsen på en *unsigned short*.

Figur 4.4b viser udførelsestiden for variable k med ratio test $f(k) = k$. Kurven konvergerer ikke mod en konstant som forventet, i slutningen er der en svag stigning. Dette kan skyldes, at prioritetskøen bliver for stor til at kunne være i $L2$ cachen. Kurven for $L2$ cache misses ses også på grafen, hvor hvert punkt er divideret med $k \cdot 10^8$.



(a) variable n, $k = 1048576$



(b) variable k, $n = 20000$

Figur 4.4: n priority: grafer

Problemet med n priority er, at prioritetskøens størrelse stiger jo større k er. I takt med at køen vokser, stiger udførelsestiden og pladsforbruget, det er derfor begrænset hvor

store værdier af k, algoritmen n priority kan finde. I næste algoritme minimeres heapen og pladsforbruget reduceres.

4.3 n searchtree

Denne algoritme prøver at reducere mængden af tripler, der er i prioritetskøen i forhold til n priority. I n priority bliver algoritmen nødt til at indsætte begge undertræer til roden r i prioritetskøen, da begge disse muligvis kan producere det næste triple. Det n priority gør ved at indsætte rødder i køen, er at simulere et inorder gennemløb af heapen. Ideen med den næste algoritme er, at udskifte datastrukturen til en struktur, hvor gennemløbet er nemmere at realisere og derved reducere mængden af potentielle efterkommere til et givent triple.

Ideen er, at udskifte Persistent lHeap med en persistent version af et søgetræ, hvor det højremeste barn fra roden er den største knude i træet. I et søgetræ er det trivielt at lave et inorder gennemløb.

Algorithm 8: n searchtree

input : A, k
output: v

```
1  T^1_suf, ..., T^n_suf ← constructSearchTree(A);
2  Priorty Queue q;
3  for i ← 1 to n do
4  |  q.insert(T^i_suf.getMax());
5  end
6  for i ← 1 to k do
7  |  t ← q.removeMax();
8  |  if t.hasNext() then
9  |  |  p.insert(t)
10 |  end
11 end
12 return v
```

I stedet for, at indsætte roden af heapsene som i forrige algoritme, indsættes den største knude fra hver version af de persistente søgetræer $T^1_{suf}, \ldots, T^n_{suf}$. Denne knude vil altid ligge som den højremeste i den pågældende version af træet. Dernæst vælges den største knude ud fra prioritetskøen og returneres, hvorefter den næste knude, fra samme træ, i inorder rækkefølgen indsættes i køen.

Hvis elementerne i søgetræet er ordnet absolut, er det muligt at finde den næste knude n_{next} i et inorder gennemløb fra en given knude n, ved at søge fra roden. Dette er ikke muligt her, da det er valgt, at triplerne ikke har en absolute orden, som diskuteret i afsnit 2.1. I stedet for at søge fra roden, findes n_{next}, udfra ns position i træet. Knude n_{next} kan findes i træet som:

- Som forældre til n, men kun hvis denne har en mindre end n.
- Som højremeste barn til det undertræ som ligger til venstre for n.

- Den først knude på stien fra n til roden af træet, der er mindre end n.

Kompleksistet

Konstruktionen af $T^1_{suf}, \ldots, T^2_{suf}$ tager $O(n \cdot \log n)$ tid, da en indsættelse i de implementerede søgetræer tager $O(\log n)$ tid. Dernæst tager det $O(\log n)$ tid, at finde det største par i køen og $O(\log n)$ tid, i værste tilfælde, at finde en n_{next} givet n.

Den samlede udførelsestid for algoritmen er $O((n + k) \log n)$.

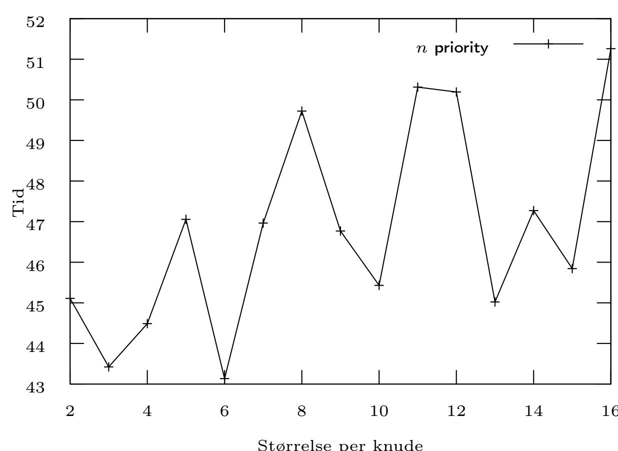
Pladsforbruget er $O(n)$ for søgetræet, samt $O(n)$ til prioritetskøen. Hvilket giver et samlet pladsforbrug på $O(n)$.

Ved at bruge den persistente udgave af søgetræ PRBT fra afsnit 6.3.4, vil knuderne ikke have en pointer til deres forældre. Dette betyder at der yderligere skal gemmes en sti fra roden til knuden n, for hver af de n versioner af træet. Dette medfører et øget pladsforbrug på $O(\log n)$. Ved at implementere PRBT med pointere til forældre, er det naturligvis muligt at undgå denne forøgelse, men tidsforbruget til konstruktionen og pladsforbruget vil stige med en konstant faktor.

Eksperimenter

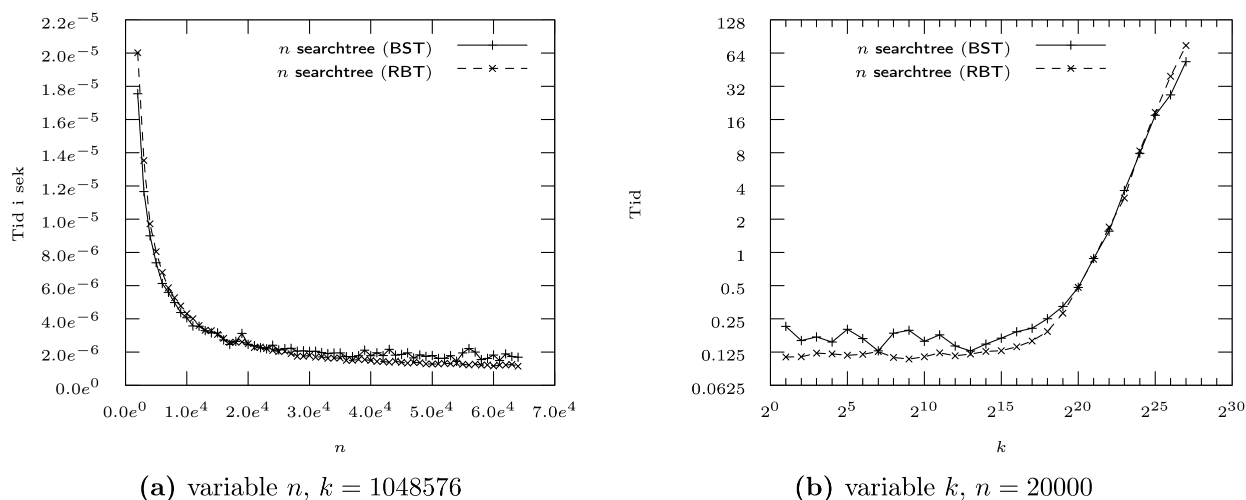
I afsnit 6.3.2 og afsnit 6.3.4 vises to forskellige udgaver af et persistent søgetræ. Det er disse to søgetræer, der bruges som underliggende datastrukturer i n searchtree.

Den første opgave er, at finde en god knudestørrelse for n searchtree med PRBT. Figur 4.5 viser tiden for et forholdvis stort problem, med forskellige knudestørrelser. Ligesom i n priority tilfældet er der ikke nogen speciel stor fordel, at have en stor knudestørrelse. Når størrelsen er 3 ser det ud som om udførelsestiden er kortest og derfor vælges denne størrelse.



Figur 4.5: n searchtree: graf knude størrelse for PRBT, $n = 20000$, $k = 67108864$

I figur 4.6a ses udførelsestiden for variabel n, den forventede udførelsestid for algoritmen er $O((n + k) \log n)$, af den grund laves en ratio test med $f(n) = n \cdot \log n + \log n$, begge



(a) variable n, $k = 1048576$

(b) variable k, $n = 20000$

Figur 4.6: n searchtree: grafer

varianter konvergerer mod en konstant, selvom kurven (den øverste) for PBST er en del mere ustabil.

Selvom n searchtree bruger mindre plads og har en mindre prioritetskø end n priority, bruges der en del tid på at traversere søgetræet. Den næste algoritme forsøger at koge n priority og n searchtree ned til en mere enkel algoritme ved, at erstatte den persistente datastruktur med en prioritetskø.

4.4 n matrix

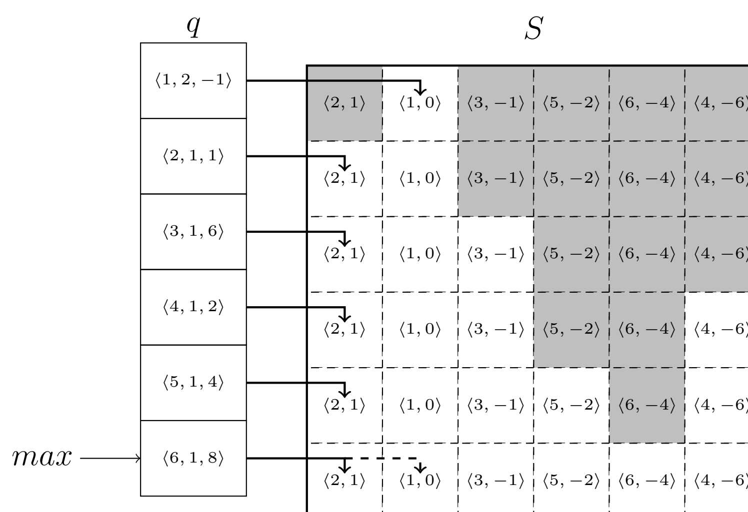
Ideen til n priority og n searchtree kommer oprindelig fra, at den $j$te version af en af underliggende datastruktur, implicit indeholder den suffiksmængde, hvor triplerne slutter på j.

Forskellen mellem version j og $j-1$ er et enkelt par $\langle j, -\delta_{j-1}\rangle$. En anden måde at se de forrige algoritmer på er, at prioritetskøen finder en version af en datastruktur samt en tilhørende δ-værdi. Dernæst lægges *delta*-værdien til den knude, der returneres fra datastrukturen. Datastrukturen har to opgaver: Den første er, at sortere og returnere $\langle i, -\delta_{i-1}\rangle$ parrene i rækkefølge fra største til mindste. Den anden opgave er, at sørge for at den $j$te suffiksmængde kun indholder tripler som slutter i j.

Ideen er, at bruge en enkelt vektor S i stedet for Persistent IHeap som beskrevet i introduktionen. Vektoren S indeholder de samme par som datastrukturen ellers ville indholde: $\langle i, -\delta_{i-1}\rangle$ for $1 \leq i \leq n$. Disse par sorteres nu efter deres δ-del, fra største til mindste. Dernæst laves der en prioritetskø med elementerne $\langle 1, \delta_1, i_S\rangle, \ldots, \langle j, \delta_j, i_S\rangle$ på samme måde som i de forrige algoritmer, men hvor i_S er et indeks i S, begyndende med det største par. Ordenen på elementerne i prioritetskøen er som før, summen af de to delta værdier.

Ideen er nu, at emulere datastrukturen med S. Først trækkes det største $\langle j, \delta_j, i_S\rangle$ ud af køen, dernæst slås der op i S med i_s og et triple produceres som $(i, j, \delta_j - (-\delta_{i-1}))$. Til sidst tælles i_S ned.

Fordi S indeholder alle mulige i indekser, er det muligt at producere tripler hvor $i > j$, hvilket ikke giver menning. For at løse dette problem, springer algoritmen alle indgange over i S hvor $i > j$.



Figur 4.7: n matrix: med $A = (-1, 2, 5, -4, 2, 4)$
.

Figur 4.7 viser start tilstanden for n matrix. Det næste element i prioritetskøen er $\langle 6, 1, 8\rangle$, fordi triplet $(2, 6, 9)$ er det størst mulige. Efterfølgende flyttes pointeren til parret $\langle 1, 0\rangle$. De grå felter vil producere ugyldige triple og er dem, som skal springes over.

Kompleksistet

Algoritmen bruger $O(n \cdot \log n)$ på at konstruere og sortere S. Dernæst bruges der $O(\log n)$ på, at trække ud af køen. Hver gang et par fra køen er fundet, tælles i_S ned. I værste tilfælde vil elementerne i S være sorteret så indekserne er i faldende orden. Hvis dette sker, vil de $n - j$ første par i den $j$te række i matricen være ugyldige indgange, hvilket vil sige der skal springes $O(n^2)$ indgange over, før der først triple kan findes. Dette giver algoritmen en udførelsestid på $O(n^2)$. Da algoritmerne bliver kørt på tilfældige data, betyder det, at det er forventet at algoritmen klarer sig bedre end værste tilfælde. Det er dog svært at give et teoretisk argument for hvor meget bedre den klarer sig.

Pladsforbruget for algoritmen er $O(n)$.

Eksperimenter

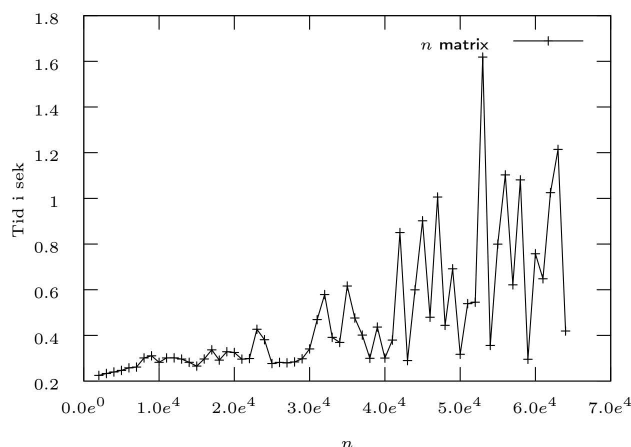
Figur 4.8a viser udførelsestiden for n matrix med variabel n. Den svingende udførelsestid skyldes, at jo større n er, jo større bliver den sortede vektor S og jo længere bliver rækkerne i matricen. Det betyder at, der potentielt er flere ugyldige indgange i hver række. En løsning på dette problem kunne være at bruge et søgetræ, til at finde ud af, hvor den næste gyldige indgang i S er, hvilket vil give en worstcase tid på $O((n+k) \log n)$.

Figur 4.8b viser udførelsestiden for variabel k. Igen er udførelsestiden meget svingende.

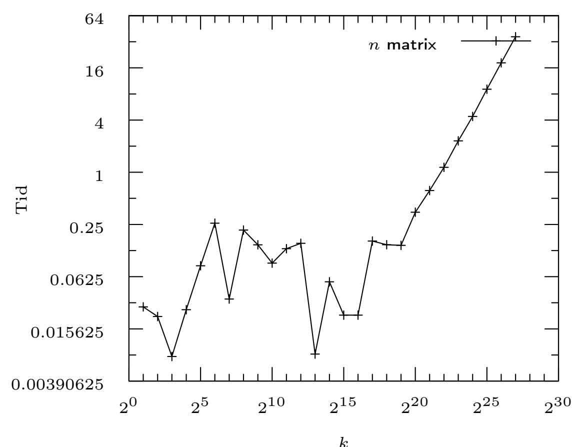
Algorithm 9: n matrix

input : A, k
output: v

1 Vector S;
2 S.insert($\langle 0, 0\rangle$);
3 $sum = 0$;
4 **for** $i \leftarrow 2$ **to** n **do**
5 $\quad$ S.insert($\langle i, sum - A[i]\rangle$);
6 $\quad$ $sum = sum - A[i]$
7 **end**
8 Priority Queue q;
9 **for** $j \leftarrow 1$ **to** n **do**
10 $\quad$ $sum = sum + A[i]$;
11 $\quad$ *Find næste gyldige indgang i S og gem indeks i* i_S;
12 $\quad$ q.insert($\langle j, i_S, sum\rangle$);
13 **end**
14 **for** $l \leftarrow 1$ **to** k **do**
15 $\quad$ $\langle j, i_S, \delta_j\rangle \leftarrow q$.removeMax();
16 $\quad$ $\langle i, -\delta_{i-1}\rangle \leftarrow S[i_S]$;
17 $\quad$ v.insert($(i, j, \delta_j - \delta_{i-1})$);
18 $\quad$ *Find næste gyldige indeks i S og gem i* i_S;
19 $\quad$ **if** $i_S < |S|$ **then**
20 $\quad\quad$ *Hvis der der findes flere indgange i S*;
21 $\quad\quad$ q.insert($\langle j, i_S, \delta_j\rangle$);
22 $\quad$ **end**
23 **end**
24 **return** v



(a) variable n, $k = 1048576$



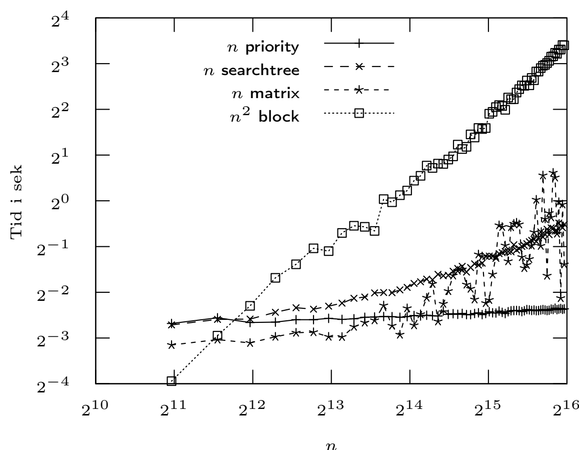
(b) variable k, $n = 20000$

Figur 4.8: n matrix: grafer

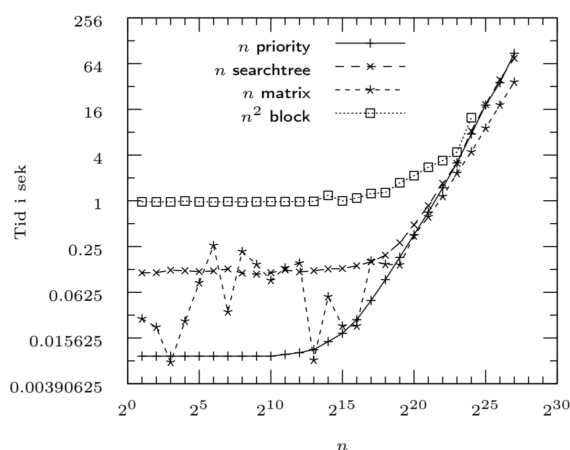
4.5 Konklusion på n-familien

De sidste tre algoritmer har alle været baseret på en $O(n + k)$ algoritme, men opnår ikke denne worstcase tid. Figur 4.9 viser de tre forskellige algoritmer sammenlignet med

hinanden samt n^2 block, som er den hurtigste algoritme fra n^2-familien. Figur 4.9a viser udførelsestiderne med variabel n. Det er tydeligt at se, at alle n-familie algoritmerne klarer sig markant bedre end n^2 block. Ved små værdier af n klarer n matrix sig bedst, hvilket kan have noget at gøre med det forholdvis lille tidsforbrug, der er ved at sætte datastrukturen op. Når n begynder at stige, begynder n matrix at klare sig dårligere. Figur 4.9b viser de fire algoritmer med variabel k. Igen er n^2 block en del langsommere end de andre algoritmer, ved 2^{24} løber algoritmen tør for hukommelse. Det meste af tiden er n priority den hurtigeste algoritme. Når k bliver stor nok begynder n matrix at overhale. I det største målepunkt er n matrix cirka dobbelt så hurtig som n priority.



(a) variabel n, $k = 65536$



(b) variabel k, $n = 20000$

Figur 4.9: n^2-familien

Kapitel 5

Selektion

I dette kapitel forsøges det, at finde en god selektionsalgoritme. Et lignende studie kan findes i [22], hvor der argumenteres for, hvordan et pivot skal vælges for, at få bedre branchprediction. I [23] vises forskellige teknikker til, at gøre implementeringen af quicksort mere effektiv, men artiklen er gammel og både processorer samt compilerer er blevet betydeligt bedre.

I afsnit 5.1 gennemgås den overordnede selektionsalgoritme. Efterfølgende, i afsnit 5.2, undersøges to forskellige måder at lave en partitionering på. I afsnit 5.3 sammenlignes fem varianter af en randomiseret selektionsalgoritme, som forventes at klare sig bedre i praksis end den sidste algoritme i afsnit 5.4, som er deterministisk.

I afsnit 5.5 testes de forskellige selektionsalgoritmer op mod hinanden og den hurtigeste kåres som vinder.

Selektionsalgoritmerne anvendes i n^2 select, n^2 select m, n^2 historik og n^2 block.

5.1 Find

Den mest kendte selektionsalgoritme er *C.A.R. Hoares* algoritme Find [24] (vil fremover blive kaldt Selection).

Selektionsalgoritmens opgave er, givet en vektor A og en rang k, at finde det element i A, som har rang k. Når algoritmen har kørt er rangen af elementerne $A[1] \ldots A[k-1]$ mindre end $A[k]$ og $A[k+1] \ldots A[n]$ er større end $A[k]$.

Algoritmen Find er i bund og grund en simplificeret udgave af sorteringsalgoritmen Quicksort[25]. I Quicksort vælges et pivot, hvis endelige plads findes. Herefter deles, inputtet så alle elementer, der har større rang, placeres til højre for pivotet og alle der er har mindre rang til venstre for. Efterfølgende køres Quicksort rekursivt på de to dele. Forskellen mellem Quicksort og Selection er, at Selection er en *prune-and-search* [17, 245] algoritme. I stedet for at undersøge begge dele af rekursionen, køres der kun rekursivt på den del, som kan indholde det $k$te element, alle andre elementer negligeres.

I [24] beskrives Find som en rekursiv algoritme, men den kan nemt, som i algoritme 10, omskrives så den kører iterativt. Fordelen er, at den iterative version både er hurtigere

$$\begin{aligned} &\text{Problem:}\\ A &= (1, 4, 3, -2, -5, 6, 2)\\ k &= 4\\ \\ &\text{Løsning:}\\ &(-5, 1, -2, \mathbf{2}, 6, 4, 3) \end{aligned}$$

Figur 5.1: Selektion med $k = 4$

Algorithm 10: Selection

```
    input : A, k
 1  l ← 1
 2  r ← |A|
 3  while true do
 4  |   p ←findPivot(A,l,r);
 5  |   p_i ←partition(A,l,r,p);
 6  |   if p = k then
 7  |   |   return;
 8  |   else if k < p_i then
 9  |   |   r ← p_i − 1;
10  |   else if k > p_i then
11  |   |   l ← p_i + 1;
12  |   end
13  end
```

og bruger mindre plads, på grund af en mindre stack.

De to værdier l og r definerer det område i A, som indeholder elementet med den $k$te rang. Før algoritmen har kørt, kan elementet være i hele inputtet, så derfor sættes l til starten og r til slutningen af A.

Et pivot, p, vælges derefter (findPivot) enten tilfældigt (Quickselect) eller deterministisk (Median of medians). Når et pivot er fundet, partitioneres (partition) A. Partitioneringen returnerer ps plads p_i i A. Hvis $k < p_i$ må det $k$te største element ligge til venstre for p_i og r sættes til $p_i - 1$ og omvendt. Herefter påbegyndes en ny iteration. Algoritmen reducerer, for hver iteration, mængden af elementer, som kan være det $k$te. Når $p_i = k$ er det $k$te største element fundet og ligger på den $k$te plads.

5.2 Partition

En partitionsalgoritmes opgave er, at dele A på følgende måde: Elementet e, som starter på den $k$te plads, vil efter partitioneringen ligge på den $j$te plads og $A[1] \ldots A[j-1]$ vil

have en mindre rang end e og $A[j+1]\dots A[n]$ vil have en rang, der er større end eller lig med e. Til sidst returneres es endelige position j.

Den første partitioneringsalgoritme flytter e over på den sidste plads i A, dernæst undersøges alle andre elementer og deres rang sammenlignes med es. Pointeren m holder styr på, hvor e skal ende op til slut. Hvis et element er mindre end e, flyttes det over på den $m$te plads og m tælles op. Det vil sige, at alle elementer, der har en rang, der er mindre end e vil ligge til venstre for m, til sidst byttes e med elementet på den $m$te plads.

Algorithm 11: Partition

```
   input  : A, l, r, p
   output: p's endelige position
1  swap(A[p], A[r]);
2  m ← l;
3  for i ← l to r do
4  |  if A[i] < A[r] then
5  |  |  swap(A[i], A[m]);
6  |  |  m ← m + 1;
7  |  end
8  end
9  swap(A[r], A[m]);
10 return m;
```

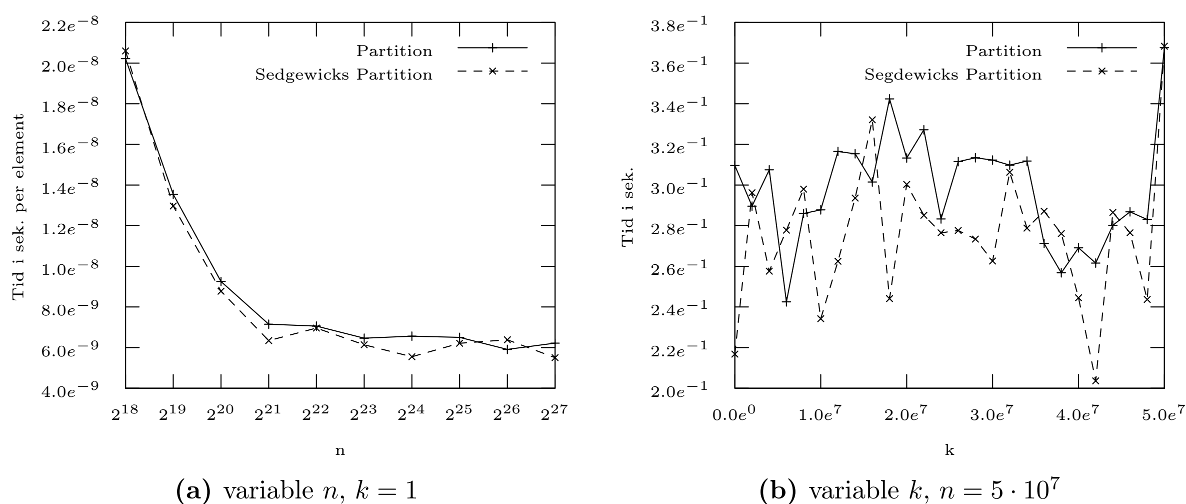
I [23] beskrives forskellige metoder til at optimere Quicksort. Idet Quicksort og Quickselect ligner hinanden meget, er det muligt, at nogle af metoderne kan bruges til at optimere Quickselect. I artiklen diskuteres der forskellige måder, at partitionere på, den næste er ifølge artiklen den mest effektive.

Algorithm 12: Sedgewick Partition

```
   input  : A, l, r, p
   output: p's endelige position
1  swap(A[p], A[r]);
2  i ← l − 1;
3  j ← r;
4  while true do
5  |  while A[i] < A[r] do
6  |  |  i ← i + 1;
7  |  end
8  |  while j > l and A[j] ≥ A[r] do
9  |  |  j ← j − 1;
10 |  end
11 |  if i ≥ j then
12 |  |  swap (A[i], A[j]);
13 |  |  return i
14 |  end
15 end
```

Sedgewicks partitionering undersøger inputtet fra begge sider. Først flyttes det $k$te element til slutningen af A. Nu findes der elementer fra venstre, som er større end e med i pointeren og elementer fra højre, som er mindre end eller lig med e med j pointeren. Hver gang to elementer er fundet, byttes disse rundt. Når de to ender mødes, vil alle elementer, som er mindre være på venstre side af r og alle der er større end eller lig med, vil være på højre side af r. Til sidst flyttes e til r. Algoritmen beskrevet i artiklen antager absolut orden. I den sammenhæng selektionen skal bruges, er dette ikke tilfældet. For at få algoritmen til at fungere på inputs med samme værdi, undersøges der løbende om j løber forbi l. I artiklen optimeres partitioneringen yderligere ved at lave "loop-unrolling" og omskrive maskinekode. Dette undlades i denne implementation. Det er forventet at Sedgewicks partitioneringen klarer sig bedre end den første algoritme, da den kun bytter rundt på elementer, når to elementer bryder ordenen.

Begge algoritmer har en tidskompleksitet på $O(n)$, da algoritmerne sammenligner alle elementer med e.



(a) variable n, $k = 1$

(b) variable k, $n = 5 \cdot 10^7$

Figur 5.2: partitionering: Grafer for n og k

Figur 5.2a viser udførelsestiden for de to algoritmer, når n varieres og k fastholdes. Ved at lave en ratio test med $f(n) = n$ kan man se, at kurverne konvergerer mod en konstant som forventet. Forskellen mellem de to partitioneringsalgoritmer er ikke specielt stor, men det ser ud som om Sedgewicks variant klarer sig marginalt bedre.

For en god ordens skyld testes også forskellige værdier af k, som ses i figur 5.2b. Da A er tilfældig er det naturligt, at begge algoritmer svinger meget i deres udførelsestider. Udførelsestiden for et hvilket som helst k, burde være den samme, hvilket den også er.

Selvom de to algoritmer ligger forholdsvis tæt på hinanden, klarer Sedgewick sig generelt lidt bedre.

Algoritmerne er også begge *in-place*, hvilket vil sige, at der ikke bruges mere plads end input vektoren. Denne egenskab gør at algoritmerne, der senere bruger partitioneringen, ikke får et øget pladsforbrug.

5.3 Quickselect

De første varianter af selektion, vælger pivotet tilfældigt. Ideen er, at et tilfældigt valgt pivots rang med stor sandsynlighed vil ligge tæt på det optimale pivot. I denne sektion implementers der fem varianter af Quickselect:

- **Standard (STD)** : Pivot vælges ved at tilfældigt.
- **Deterministisk (DET)**: Pivotet vælges så indekset bliver det mindst mulige.
- **Median-of-3 (M3)**: Tre elementer vælges og medianen af dem vælges som pivot.
- **Proportion-of-3 (P3)**: Tre elementer vælges tilfældigt, pivotet vælges afhængigt af k.
- **Deterministisk Proportion-of-3 (P3D)**: Tre elementer vælges deterministisk, pivotet vælges afhængigt af k.

Først redegøres der for, at Quickselect bruger tid lineært i input. Analysen er simplificeret, en mere præcis analyse kan ses i [26, 27]. Den tilfældige variabel $T(n)$ repræsenterer udførelsestiden for algoritmen med input n. Lad os antage, at et godt pivot, er et pivot, der deler elementerne, så antallet af elementer der har en større eller mindre rang er $\frac{3n}{4}$.

Hvis $G(n)$ er den tilfældige variabel, der repræsenterer antallet af rekursive kald, før vi rammer et godt pivot, så gælder det at:

$$T(n) \leq \alpha n \cdot G(n) + T(3n/4)$$

Med andre ord, så bruges der αn for hver gang vi rammer et dårligt pivot, samt tiden for at løse det reducerede problem $T(\frac{3n}{4})$.

Ved at bruge *linearity of expectation*

$$E(X + Y) = E(X) + E(Y)$$

samt at $E(G(n))$ er 2, da sandsynligheden for at ramme et godt pivot er $\frac{1}{2}$, fåes der:

$$\begin{aligned} E(T(n)) &\leq E(\alpha n \cdot G(n) + T(3n/4)) \\ &\leq \alpha n \cdot E(G(n)) + E(T(3n/4)) \\ &\leq 2\alpha n \cdot \sum_{i=0}^{\log_{4/3} n} (3/4)^i \leq cn \quad \square \end{aligned}$$

Det sidste led, er hvad man kalder en geometrisk sum og den giver følgende:

$$\sum_{i=0}^{n} a^i = \frac{1 - a^{n+1}}{1 - a}$$

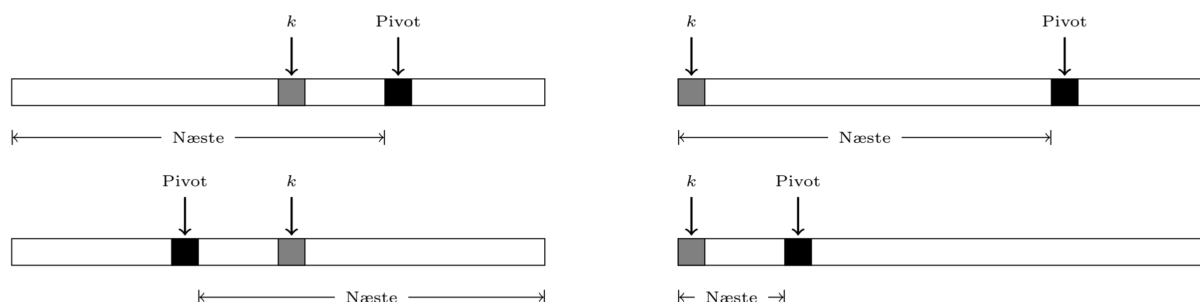
$$\sum_{i=0}^{\log_{4/3} n} (3/4)^i = \frac{1 - 3/4^{\log_{4/3} n + 1}}{1/4}$$

Leddet vokser langsommere end $O(n)$. Tilsammen med $2\alpha n$, er den forventede udførelsestid derfor $O(n)$.

I [27], beskrivers der en ligning for udførelsestiden, som også kan ses som ligning 5.1. Ligningen er desuden med på graferne for, at undersøge om den forventede udførelsestid passer med de implementerede Quickselect algoritmer.

$$\mathbb{E}\{T_{n,k}\} = 2(n + 3 + (n + 1)H_n - (k + 2)H_k - (n + 3 - k)H_{n+1-k}) \tag{5.1}$$

Det er ikke åbenlyst hvorfor Quickselect klarer sig bedre ved ekstremerne end ved midten. Figur 5.3 viser to forskellige valg af k, hvor algoritmen har valgt to tilfældige pivoter, næste iteration er vist underneden. Intuitionen er, at hvis k vælges til midten, vil alle andre valg af pivot end k, resultere i at algoritmen skal køre rekursivt på mindst halvdelen af elementerne. Hvis k derimod ligger til en af siderne, vil valget af et pivot fra k til midten resultere i et kortere gennemløb i næste iteration.



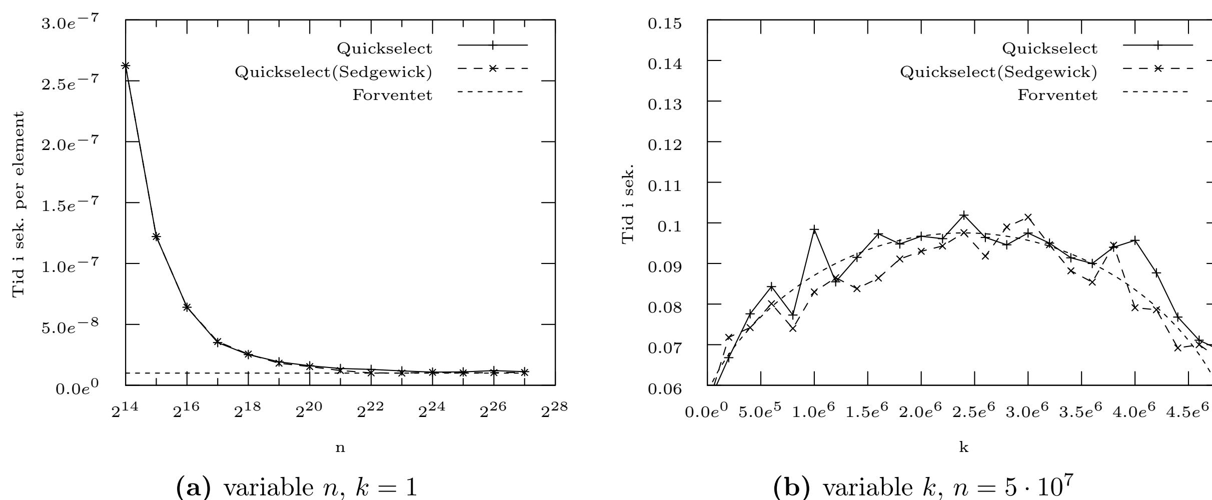
Figur 5.3: Quickselect: Forskellige valg af k

Figur 5.4a viser udførelsestiderne for Quickselect med variabel n og ratio test $f(n) = n$. Algoritmen konvergerer mod en konstant, som forventet. Figur 5.4b viser grafen for variabel k. Kurven buer og algoritmen klarer sig bedre ved store og små værdier af k.

Fordelen ved at bruge Sedgewicks partitionering er ikke specielt tydelig. Kurverne viser dog en minimal forbedring. Af den grund bruges Sedgewick fremover til at partitionere i Quickselect.

I Sedgewick [28] diskuteres forskellen mellem et tilfældig valgt pivot og at vælge en bestemt indgang i inputtet hver gang. Ved at vælge l som pivot, i stedet for at vælge et tilfældigt element, er det forventet at, algoritmen kommer til at køre hurtigere, da det er langsomt at generere tilfældige tal. Problemet er, at hvis valget af pivotet på nogen måde er deterministisk, så vil der findes et input, som tvinger algoritmen til konsekvent, at vælge et dårligt pivot. F.eks, hvis valget er det venstremeste og inputtet er sorteret fra største til mindste, vil pivotet i hver iteration, kun reducere problemet med et enkelt element. Som resultat vil algoritmen køre i $O(n^2)$ tid.

Hvis man ønsker en variant af Quickselect, som kan lave en selektion i garanteret $O(n)$ tid, men som kører ligeså hurtigt som Quickselect, er Introselect[29] et bud. Når inputtet er tilpas "dårligt" skiftes der fra Quickselect til Median of medians.



(a) variable n, $k = 1$

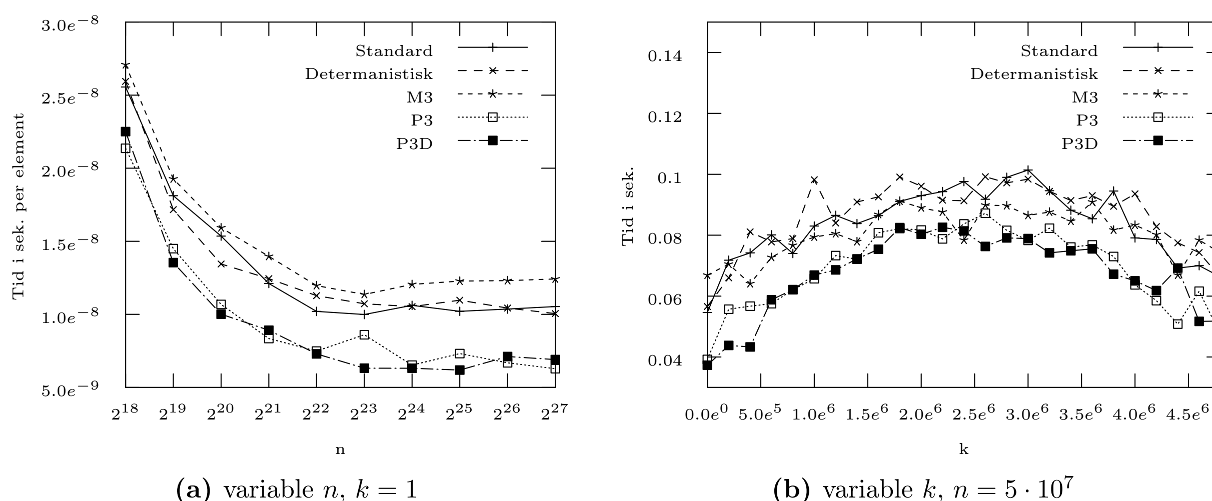
(b) variable k, $n = 5 \cdot 10^7$

Figur 5.4: Quickselect: grafer

En anden optimerings strategi, som også diskuteres i Sedgewick er, at vælge en prøve af elementerne og lade medianen af dem være pivotet. Dette giver en større sandsynlighed for, at finde et element, som er tættere på inputtets median, hvilket igen medfører en reduceret rekursionsdybde. Hvis denne prøve består af tre elementer, kaldes strategien for *median-of-3* [28] eller M3. De tre elementer der vælges er l, r og $\lceil \frac{r-1}{2} \rceil$.

I Quickselect er det ikke medianen, der er interessant, men det at finde et pivot, som er så tæt på k som muligt. Hvis k er 10 i et input på 1000, vil det måske være bedre at vælge det mindste af de 3 valgte elementer. Denne strategi kaldes for *Proportion-from-3* [30] eller P3. Strategien vælger 3 tilfældige elementer som sorteres. Dernæst deles inputtet i tre dele, hvis k ligger i den øverste trejdedel, bliver det største af de sorterede elementer pivotet. Hvis k ligger i midten bliver det næststørste element pivotet og ellers bliver det mindste det nye pivot.

Den sidste strategi *Deterministisk Proportion-from-3* eller P3D er *Proportion-from-3* hvor de tre valgte elementer er l, r og $\lceil \frac{r-1}{2} \rceil$ i stedet for tilfældigt valgte.



(a) variable n, $k = 1$

(b) variable k, $n = 5 \cdot 10^7$

Figur 5.5: Quickselect: grafer for strategier

Figur 5.5 viser graferne for de forskellige strategier. Inputtet til algoritmerne er uniformt fordelt. Som forventet er det hurtigere at bruge l som pivot end et tilfældigt element. M3 strategien klarer sig på ligefod med standard, hvilket ikke er så underligt, da der bliver brugt en del operationer på finde et ikke optimalt pivot. P3 og P3D klare sig markant bedre end de andre strategier, hvor P3D er en smule hurtigere end P3.

5.4 Median of medians

For at finde et pivot deterministisk bruges Median of medians [31]. Ideen er, at dele inputtet op i grupper af en fast størrelse, hvorefter der findes en median af grupperne. Når medianen af alle grupperne er fundet, køres der rekursivt på disse medianer. Det endelige pivot er den median, der bliver fundet rekursivt.

Først gives der er kort argument, for hvorfor selektion med Median of medians bruger tid lineært i input. Analysen er simplificeret.

I [31] argumenteres der for, hvorfor størrelsen af grupperne mindst skal være 5.

Det pivot, der findes, er per definition større og mindre end halvdelen af de medianer der findes. På samme tid er det også større og mindre end 3 af de 5 elementer i hver gruppe. Derfor er medianen større og mindre end $\frac{n}{5} \cdot \frac{1}{2} \cdot 3 = \frac{3n}{10}$ elementer. Udførelsestiden for selektion med Median of medians bliver derfor:

$$T(n) \leq T(n/5) + T(7n/10) + \alpha n \tag{5.2}$$

hvor:

- $T(n/5)$ er tiden for at løse problemet rekursivt, da $\frac{4}{5}$ dele af elementerne bliver kasseret ved hver rekursion.
- $T(7n/10)$ er resten af problemet, idet problemet bliver reduceret med mindst $\frac{3n}{10}$ af elementerne.
- αn er udførelsestiden for partitioneringen, samt udførelsestiden for, at sortere de $n/5$ grupper.

Hvis det antages, at n er stor og at udførelsestiden er $O(n)$ eller cn, kan (5.2) reduceres som følger:

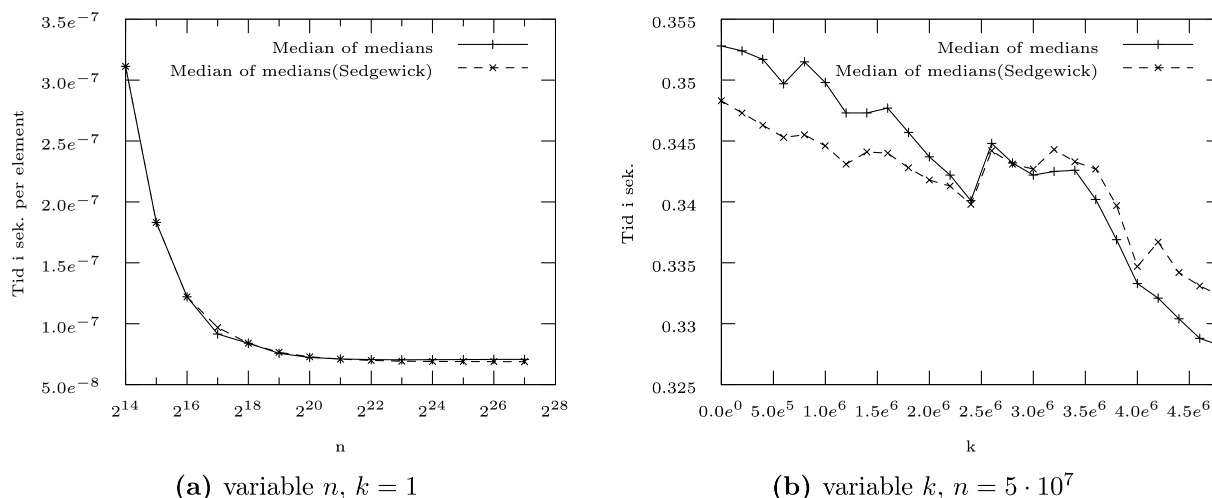
$$\begin{aligned} T(n) &\leq T(n/5) + T(7n/10) + \alpha n \\ cn &\leq cn/5 + 7cn/10 + \alpha n \\ cn &\leq n(c/5 + 7c/10 + \alpha) \\ c &\leq c/5 + 7c/10 + \alpha \\ c &\leq \alpha 10 \quad \square \end{aligned}$$

Med andre ord, så findes der en konstant c, så $T(n)$ er $O(n)$.

Hvis gruppernes størrelse er 5, vil algoritmen finde et element, som har en større rang end $3n/10$ af alle elementer og mindre rang end $3n/10$ af elementerne.

Ligesom partitioneringsalgoritmerne implementeres Median of medians *in-place*. Til sorteringen af grupperne bruges Insertion sort [11, 80]. Sorteringsalgoritmen har en tidskompleksistet på $O(n^2)$, men da grupperne alle har en konstant størrelse, er udførelsestiden ligegyldig og en hvilken som helst sorteringsalgoritme vil kunne bruges. Sortering med insertion sort, har den fordel, at algoritmen er hurtig på små input.

Når en median er fundet, byttes denne med et element i slutningen af inputtet, hvor der står et tilfældigt element. Det tilfældige element skal også indgå i udvælgelsen og derfor flyttes det hen i slutningen af gruppen, hvor det indgår som en del af næste gruppe. Når alle grupper er sorteret, starter en ny runde på medianerne. Når algoritmen har kørt, vil medianen af medianerne ligge som det sidste element i inputtet.



(a) variable n, $k = 1$ **(b)** variable k, $n = 5 \cdot 10^7$

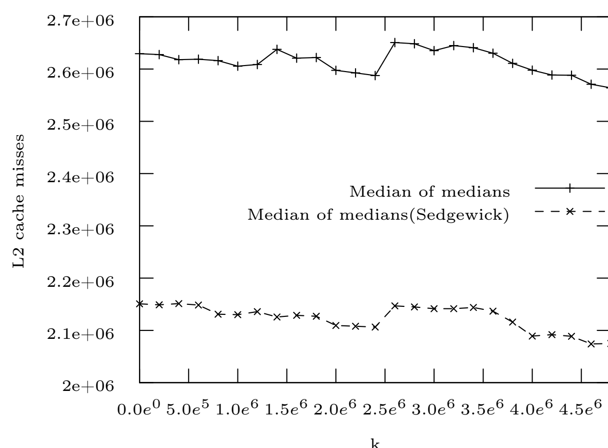
Figur 5.6: Median of medians: grafer

Figur 5.6a viser udførelsestiden for selektion med Median of medians, med variende n og ratio test med $f(n) = n$. Kurven konvergerer mod en konstant, som forventet.

Sedgewicks partitionering ser ikke ud til, at have den helt store fordel, faktisk er den en smule langsommere, når n bliver stor.

Figur 5.6b viser tilgengæld en adfærd, som ikke er forventet. Igennem hele testområdet falder algoritmens udførelsestid. Fordi det pivot, der findes, er både større og mindre end $3n/10$ af elementerne, burde det som i Quickselect tilfældet være hurtigere, at finde et k i ekstremerne og kurverne burde bue. For at finde ud om den uventede udførelsestid skyldes cachen, undersøges antallet af $L2$ cache misses. Resultatet ses som figur 5.7.

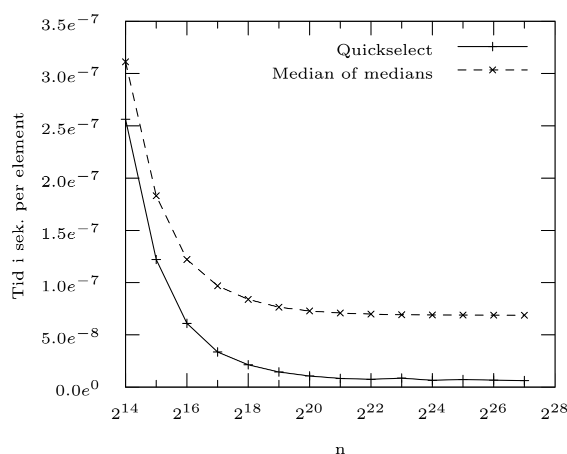
Grafen viser, at den faldende tendens også kan ses i antallet af $L2$ cache misses. Idet Median of medians gennemløber inputtet deterministisk, må dette fald i $L2$ cache misses betyde, at der enten er en forskel på rekursionsdybden i selve udvælgelsen af pivotet eller også betyder det, at det pivot, der bliver fundet, ikke er det rigtige. Det har ikke været muligt at finde fejlen.



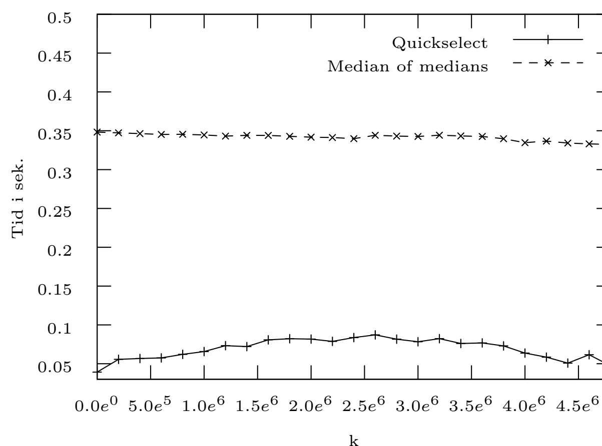
Figur 5.7: variable k, $n = 5 \cdot 10^7$

5.5 Sammenligning af selektionsalgoritmer

Figur 5.8a viser algoritmerne kørt med variabel n med ratio test $f(n) = n$. Forskellen mellem de to selektionsalgoritmer er cirka faktor 6.



(a) variable n, $k = 1$



(b) variable k, $n = 5 \cdot 10^7$

Figur 5.8: Sammenligninger af de forskellige varianter af selektion

Figur 5.8b tester algoritmerne med forskellige værdier af k. Det er tydeligt udfra de to grafer, at Quickselect er hurtigst af de to algoritmer.

Kapitel 6

Persistente datastrukturer

I dette kapitels første afsnit 6.1 defineres persistens og forskellige teknikker til at gøre datastrukturer persistente.

I det følgende afsnit 6.2.1 implementeres IHeap, som er en maksordnet binær heap, hvorefter denne gøres persistent i afsnit 6.2.2. Den persistente udgave af IHeap bruges i n priority.

I afsnit 6.3 beskrives to forskellige søgetræer. I afsnit 6.3.1 beskrives et standard søgetræ, som gøres persistent i afsnit 6.3.2.

I afsnit 6.3.3 beskrives et "rød sort"-træ, som er en simpel måde at balancere et søgetræ på. I afsnit 6.3.4 gøres træet persistent.

De to persistente søgetræer bruges i algoritmen n searchtree.

6.1 Persistens

I dette afsnit gennemgås persistens og forskellige teknikker til at opnå persistens af datastrukturer. I de efterfølgende afsnit gennemgås forskellige datastrukturer, som i persistent form skal bruges i k maksimum delsums algoritmerne samt, hvordan disse gøres persistente.

Normalt er datastrukturer flygtige. Det vil sige, at det kun er den sidste version af datastrukturen, som kan tilgåes. Enhver ændring af datastrukturen opretter en ny version og den gamle ødelægges.

Et eksempel kan være en binær heap. Normalt kan man indsætte et element i heapen ved at bruge insert operationen. For at udføre en insert findes først stedet, hvor elementet skal indsættes og dernæst skal nogle elementer muligvis flyttes op igennem heapen. Efter en *insert* operation ændres strukturen på heapen. Der oprettes altså en ny version af heapen.

En datastruktur er persistent, hvis den tillader adgang til flere versioner. I artiklen [32] diskuteres der forskellige teknikker, til at gøre datastrukturer persistente. Artiklen definerer en datastruktur, som en *kædet datastruktur*. En kædet datastruktur, består af en

endelig mængde knuder, der hver har et konstant antal variabler. Hver variabel kan enten være en værdi eller en pointer til en anden knude. En delmængde af disse knuder fungerer som *adgangsknuder*. I tilfældet med den binære heap, indeholder mængden af adgangsknuder roden af heapen til hver version. En kædet datastruktur tillader to typer af operationer. Den første kaldes for *tilgående operation*. Denne konstruerer en mængde af *tilgåede knuder*. Operationen består af en sekvens af *tilgangsskridt*. Hvert skridt tilføjer en knude til de tilgåede knuder. For at tilføje en knude skal den enten være en adgangsknude eller blive peget på af en knude, der allerede er i mængden af adgangsknuder. Den anden type operation er en *opdaterende operation*, som er en blanding af tilgangsskridt og opdaterskridt. Et *opdaterskridt* ændrer på datastrukturen, ved enten at konstruere en ny knude og tilføje den til de tilgåede knuder eller ved at ændre på en af de i forvejen tilgåede knuder. I binær heap tilfældet er insert en opdaterende operation. Først skal der bruges nogle tilgangsskridt på, at finde den knude, der skal indsættes under, dernæste bruges der en række opdaterskridt på, at flytte knuden op igennem træet. Hver gang en opdaterende operation har kørt, skifter datastrukturen version. Den $i$te version er datastrukturen efter i operaterende operationer. En flygtig datastruktur tillader kun tilgående- og opdaterende operationer på den nyeste version. En partiel persistent datastruktur tillader, at tilgående operationer køres på alle versioner, men kun på den sidste, kan der køres opdaterende operationer. En fuld persistent datastruktur tillader både tilgående- og opdaterende operationer på alle versioner.

6.1.1 Naivt

Den nemmeste måde at gøre en datastruktur persistent på, er ved at kopiere hele strukturen ved hver opdaterende operation. Ved at gøre strukturen persistent på den måde, vil enhver opdaterende operation få et overhead på $O(i)$ tid og $O(i^2)$ plads, hvor i er antallet af tidligere opdaterende operationer. En anden strategi er, at gemme alle opdaterende operationer og genskabe den version, der skal bruges. Denne fremgangsmåde bruger $O(i)$ ekstra plads og tid, for en operation anvendt på den $i$te version. Problemet med begge disse fremgangsmåder, er at de kører meget langsomt på store datastrukturer.

6.1.2 Fat node metoden

I modsætning til den naive tilgang, hvor hele strukturen gøres persistent på en gang, gemmes samtlige variabler, for samtlige versioner, i hver knude. En “fed” knude repræsenterer alle de tilstande en knude har optrådt i. Fordi en datastruktur kan have arbitrært mange forskellige tilstande, kan en knude ligeledes blive arbitrært “fed”. Hver variabel repræsenteres af et antal par, hvor et par (i, v) består af et versionsnummer i og en værdi/pointer v. Hver knude har desuden også en ekstra variabel, som indholder et versionnummer fra det tidspunkt, hvor knuden blev oprettet. Et opdaterende skridt kan, som før nævnt, være konstruktionen af en ny knude. I Fat node metoden sættes den nye knudes variabler og dens versionnummer sættes til den nuværende version af datastrukturen. Et opdaterende skridt kan også være en ændring af en variabel. I det tilfælde tilføjes der et nyt par til variablen, med den nye værdi/pointer og det nuværende versionnummer. Desuden opbevares der en struktur med alle adgangsknuderne. Hvis denne struktur er en vektor, hvor den $i$te indgang indholder alle adgangsknuder der findes til version i, er det muligt

at vedligeholde adgangsknuderne i amortiseret konstant tid og finde adgangsknuderne til den $i$te version i konstant tid. For at tilgå strukturen til en given version, bruges først vektoren med adgangsknuderne til, at finde den rigtige version. Dernæst traverseres strukturen fra den ønskede adgangsknude. Hver gang en knudes variabler skal tilgås, vælges det par, som har det største versionnummer, men som stadig er mindre end eller lig med den version der traverseres med. Problemet med fat node metoden er, at en knude kan blive arbitrært stor. Det vil sige at givet en knude, kan det potentielt tage lang tid, at finde det rigtige par. Hvis en variabel repræsenteres med en linked liste af par, hvor hvert nyt par sættes i enden af listen, vil tidsforbruget i værste tilfælde for at tilgå en knude fra en anden være $O(i)$. Dette kan optimeres ved at repræsentere variablen med en heap, hvilket reducerer tidsforbruget til $O(\log i)$.

6.1.3 Node-copy metoden

I modsætning til *Fat node metoden* tillader *Node-copy* kun et endeligt antal par per variabel. Hvis der ikke er plads til et par i en knude, så laves der en kopi af knuden og alle knuder der pegede på den gamle knude i den aktuelle version, skal opdatere deres pointere til den nye knude med et nyt versionnummer. Hvis nogle af disse heller ikke har plads til opdateringen, kopieres de på samme måde. Ideen er, at knuderne nu har et konstant antal par per variabel og derfor vil et tilgående skridt nu tage konstant tid. Problemet er nu, at en opdaterende operation kan komme til at tage $O(i)$ fordi i værste tilfælde vil alle knuder i hele datastrukturen skulle kopiers.

6.2 Persistent Insert heap

6.2.1 Insert heap

I dette afsnit gennemgåes en maksimum ordnet binærheap. Det specialle ved denne heap er, at den ikke understøtter de traditionelle operationer: delete-max, increase-key og merge. Disse lempelser gør det muligt at implementere insert operationen, så den bruger amortiseret[33] konstant tid.

De traditionelle heaps f.eks. Skew heap[34] eller Leftist heap[35] bruger begge $O(\log n)$ på en insert. Både Leftist heap og Skew heap har begge det problem, at tiden for indsættelse blandt andet bruges til, at ophobe potentiale, der reducerer udførelsestiden for deres øvrige operationer. I [1] gennemgåes IHeap (insert heap), som er en simpel heap der udnytter lempelsen omkring de manglende operationer til at konstruere en effektiv heap. Datastrukturen IHeap er baseret på Skew.

En knude i en IHeap har følgende medlemmer:

- parent: dens forældre.
- left: dens venstre barn.
- right: dens højre barn.
- value: det indsatte element, som bestemmer rangen af knuden.

Når der indsættes et nyt element i Skew heap, klippes den højremeste sti fra roden i stykker, så alle knuderne på stien, bliver rødder i hver deres heap. Knuderne sorteres og sættes sammen igen.

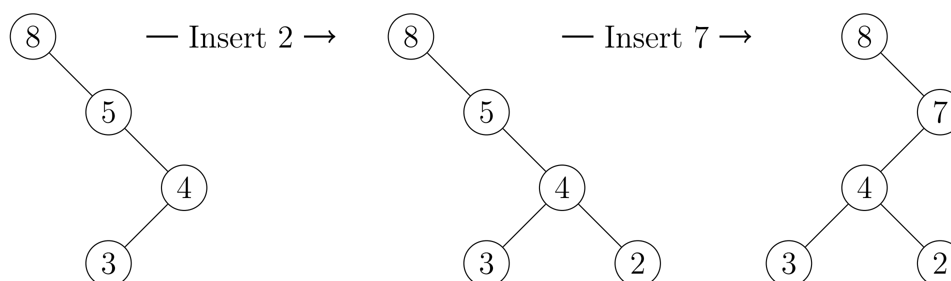
I stedet for at splitte den højremeste sti op, finder IHeap den første knude, p, på stien via parent pointerne, hvor en indsættelse under denne vil opfylde heapordenen. Den nye knude, n, bliver indsat som p.right og dens tidligere højre barn n_{old} bliver indsat som n.left. Hvis der ikke bliver fundet en større knude, bliver n roden af heapen og den gamle rod bliver dens left.

Algorithm 13: IHeap.insert

input : e

```
1 p ←rightMost;
2 while n < p do
3 |   p ← p.parent;
4 end
5 n.right ← p.right;
6 p.right ← n;
7 n.parent ← p;
8 rightMost ← n;
```

Figur 6.1 viser et par indsættelser i heapen. Først indsættes 2. Knuderne på den højremeste sti gennemløbes. Den nye knude med value 2 bryder heapordenen med det samme og bliver indsat som højre barn til 4. Dernæst indsættes 7, som bryder ordenen lige under 8 og 4 bliver dens venstre barn.



Figur 6.1: Indsættelse af 2 og 7 i en eksisterene IHeap

Korrekthed

Heapordenen er altid opfyldt, fordi der kun indsættes under knuder med større rang. Medmindre disse ikke findes og i det tilfælde indsættes der som rod. Derved vil ordenen i en IHeap altid være korrekt efter en indsættelse.

Kompleksitet

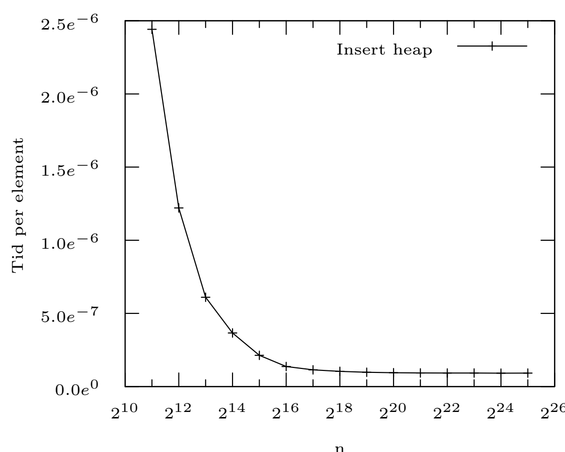
I værste tilfælde kan der konstrueres en højremest sti, der består af $n-1$ knuder, ved at indsætte dem i faldende orden. Hvis den $n$te knude er større end de $n-1$ tidligere

indsatte, vil en indsættelse medføre, at algoritmen bruger $O(n)$ tid på at gennemløbe den højremeste sti. Når man kigger på værste tilfælde er IHeap ikke specielt interessant, da den klarer sig væsentligt dårligere end den normalt brugte balancerede binær heap. Der hvor IHeap viser sin styrke er, hvis man vælger at se på heapen i amortiseret[33] forstand. Ved at analysere den amortiserede udførelsestid for algoritmen, kan vi sige noget om, hvordan algoritmen klarer sig i værste tilfælde, over en række operationer. En klassisk fremgangsmåde til, at lave en amortiseret analyse af en algoritme er ved, at lade operationer betale for noget der potentielt sker senere. I analysen af IHeap er det gennemløbet af den højremeste sti, der giver algoritmen dens høje udførelsestid. I analysen betaler hvert indsat element for, at det kan blive gennemløbet på følgende måde:

Hvis l er antallet af knuder, der bliver passeret på den højremeste sti før der findes en knude, som bryder ordnen, vil tidsforbruget blive $O(l)$ og den højremeste sti bliver $l - 1$ elementer kortere efter indsættelsen. Hver element på stien til roden bliver kun passeret en gang, hvorefter knuden bliver flyttet over som venstre undertræ af en knude. Efterfølgende kan en knude ikke bidrage til udførelsestiden, da det kun er den højremeste sti der gennemløbes ved indsættelse. Hvis hvert element betaler $O(1)$ ekstra tid for at blive passeret en gang på stien, vil der være nok tid til at gennemløbe knuderne og den amortiserede udførelsestid bliver $O(1)$ per indsættelse.

Eksperimenter

Figur 6.2 viser grafen for indsættelse i IHeap. Udførelsestiden vises med ratio test $f(n) = n$, da udførelsestiden for n indsættelser er $O(n)$. Kurven konvergerer mod en konstant, hvilket vil sige, at insert operationen bruger konstant tid som forventes.



Figur 6.2: IHeap: indsættelser

6.2.2 Persistering af Insert heap

Det er tydeligt udfra beskrivelsen at IHeap er flygtig og ikke persistent. Ved at bruge de teknikker der er beskrevet i afsnit 6.1 gøres IHeap persistent.

I den sammenhæng IHeap skal bruges, indsættes der nogle elementer i heapen, hvorefter den til sidst traverseres. Det er derfor ikke nødvendigt at understøtte fuld persistens, men

kun partiel persistens. Et krav til datastrukturen er, at traversering fra en knude til en anden foregår i konstant tid.

Persistering med Fat node metoden

For at gøre IHeap persistent med Fat node metoden, er det kun nødvendigt, at right pointeren gøres persistent. Pointeren parent, der bruges til at finde p, er kun relevant i den sidste version. Pointeren left kan kun ændres på, når n bliver indsat. Der er derfor ingen grund til, at disse gøres persistente.

Vektoren med adgangsknuderne vil til hver version, kun indholde roden af heapen. Indsættelse af et element i version i foregår ved, at gå op igennem heapen fra den højremeste knude. Den højremeste knude behøver ikke, at være i sættet med adgangsknuderne, da der kun kan indsættes i den sidste version og den kan derfor gemmes eksternt. Når p er fundet, indsættes der et par (i, n) i p.right. Pointeren n.left sættes til n_{old} præcist som i den flygtige udgave.

Gennemløbet af IHeap, givet en version i, vil foregå ved først at bruge konstant tid på at finde den rigtige rod i adgangsknuderne.

Adgangen til left foregår præcist som i den flygtige udgave.

Adgang til en right, viser sig at være et problem. Det tager ikke konstant tid, at finde det rigtige par med et givent versionnummer. Ved at indsætte det største element først og de resterende ordnet fra den mindste til den største, vil den største knudes right komme, til at indeholde par for pointere til alle knuder med undtagelse af en knude. Det vil sige, at det kan tage op til $O(\log n)$ tid at, finde det rigtige par. Hvilket bryder kravet for tidskompleksiteten.

Persistering med Node-copy metoden

Ligesom i Fat node er det nok at, gøre right persistent. Ved at sætte antallet af mulige par i right til, at være konstant, men større end to, viser det sig, at den ønskede kompleksitet opnås. Problemet fra Fat node er trivielt løst, da udførelsestiden for, at finde det rigtige par nu er konstant. Traversering i heapen er derfor konstant per besøgte knude. Det eneste der mangler er, at argumentere for kompleksiteten. Antallet af variabler i en knude er konstant, så selve kopieringen af en knude tager konstant tid. Det eneste tidspunkt der skal kopiers en knude på, er hvis right variablen er fuld. Ved at lade indsættelsen af en knude betale for kopieringen af dens forældre, får vi følgende:

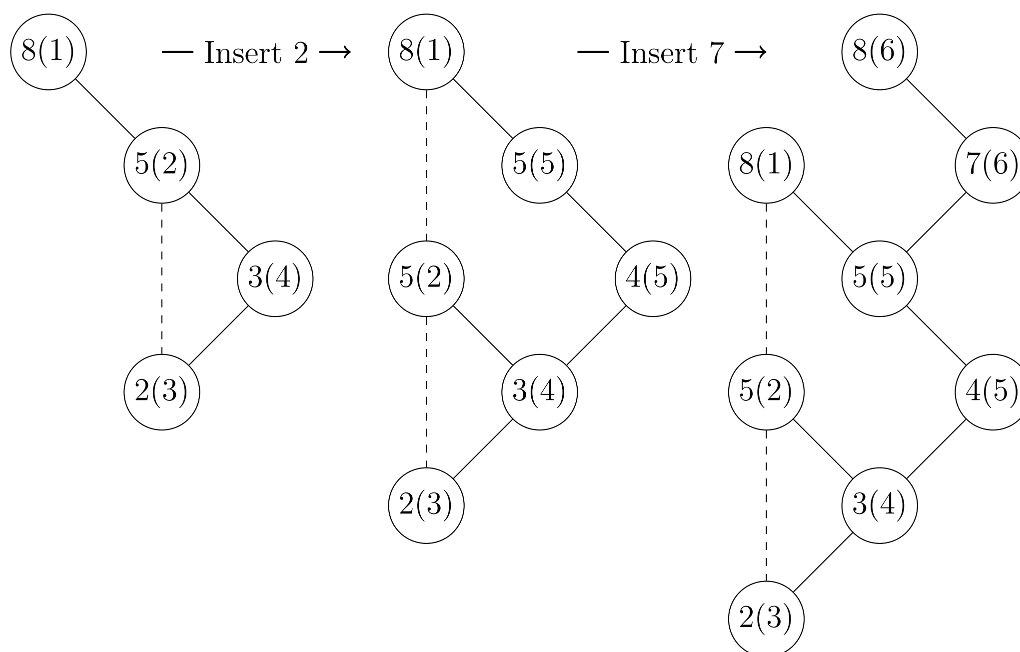
- Hvis en knude indsættes som roden i træet og den gamle rod bliver left, er der ikke nogen forældre at betale for, dette er dog ikke noget problem.
- Hvis en knude indsættes et vilkårligt sted på den højremeste sti, uden at der sker en kopiering, betaler den indsatte for, at dens forældre kan kopieres senere.
- Hvis en knude indsættes og der ikke er plads i forælderen, bruges den ophobede betaling til lave en ny forælder. Den nye forælder tilføjes som højrebarn af den gamle knudes forælder, hvilket igen kan udløse en kopiering. Potentielt set, vil hele

den højremeste sti blive kopieret, men dette er ikke noget problem, da hver knude har betalt for, at dens forældre kan kopieres.

Efter en kopiering, vil der ikke blive indsat i den gamle forælder og den nye forælder vil have en ekstra plads i right. Det er derfor altid muligt, at ophobe betaling til fremtidig kopiering.

Ved at benytte Node-copy metoden er det derfor muligt, at lave en persistent IHeap, som understøtter indsættelse og traversering i konstant tid.

Figur 6.3 viser indsættelser i en IHeap med *Node-copy*, hvor størrelsen på right er to. Efter hver indsættelse vises heapen i den nyeste version. Den stiplede linje viser pointere, som ikke er aktuelle i den nuværende version og tallet i parenteserne viser knudens versionsnummer. Ved indsættelse af 4 er der ikke plads til flere par i 5 og der skal laves en kopi. Pointeren fra 8 skal derfor opdateres, som tilgengæld har plads til den nye 5 knude. Ligeledes laves der en kopi af 8 når 7 indsættes.



Figur 6.3: Indsættelse af 4 og 7 i eksisterene Persistent IHeap

Implementation

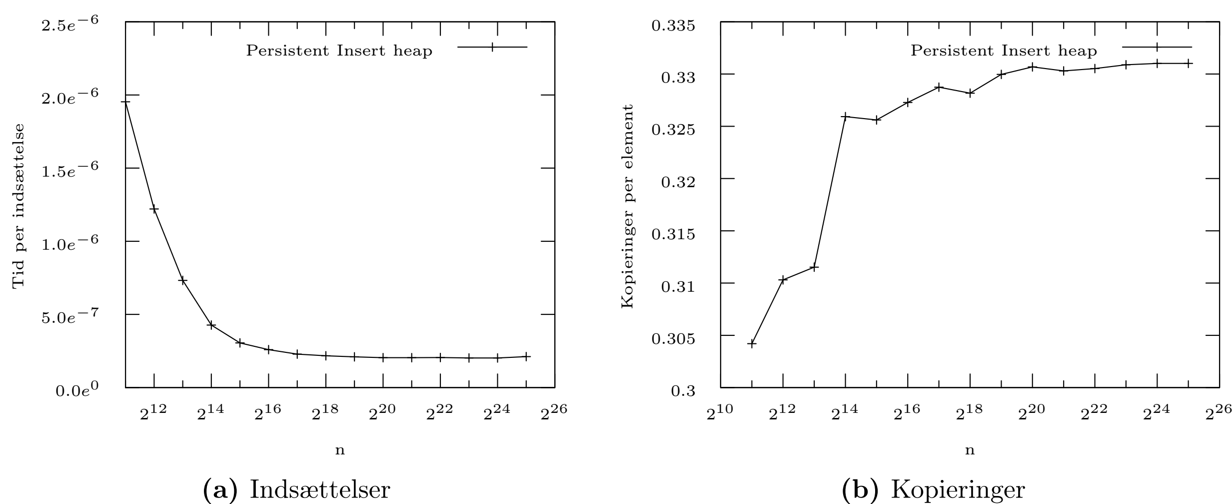
I IHeap har hver knude en pointer til forælderen. Denne knude kan spares væk ved i stedet, at holde den højremeste sti i en separat liste. Hver gang en knude på den stien besøges, fjernes den på samme tid fra listen. Når p er fundet indsættes den nyindsatte knude i listen. Problemet er nu, at hvis en knude skal kopieres, skal den erstatte den gamle i listen. Heldigvis vil en kopiering altid foregå fra p imod roden og en udskiftning kan foretages ved at forsætte i listen, så længe der sker kopieringer.

For at spare yderligere på pladsforbruget er det nok at right repræsenteres med en vektor af pointere til knuder. For at finde en pointers versionsnummer følges pointeren til en knude og pointerens version svarer til knudens versionsnummer. En knudes versionsnummer vil

altid svare til det tidspunkt knuden blev konstrueret og en knude kun kan tilføjes til right når den bliver indsat eller når den bliver kopieret.

Eksperimenter

For finde ud af om udførelsestiden for indsættelsen er korrekt, indsættes et variabelt antal uniformt fordelt tilfældige *double* værdier. Figur 6.4a viser udførelsestiden med ratiotest $f(n) = n$, kurven konvergerer som forventet, mod en konstant. Figur 6.4b viser antallet af kopieringer per indsat element. Grafen viser at der cirka bliver kopieret en knude ved hver tredje indsættelse. Det ser til at sandsynligheden for en kopiering stiger, jo flere elementer der indsættes. En kopiering sker kun, hvis der indsættes en værdi, som hverken er større end den største (indsættelse som ny rod) eller mindre end den mindste(ny højremeste). Jo flere elementer der indsættes, jo mindre er chancen for, at et element er større eller mindre end alle de forrige. Grafen viser, at antallet af knuder i grafen er cirka $n + \frac{n}{3}$. Ved at vælge at right variablen kan have plads til flere knuder, vil antallet af kopieringer falde, men tidsforbruget på at traversere heapen vil stige. Det er svært at teste, hvor mange pladser der skal være i right uden at have en algoritme, der bruger heapen. Denne test gemmes derfor til når heapen anvendes.



(a) Indsættelser **(b)** Kopieringer

Figur 6.4: Persistent IHeap: Grafer

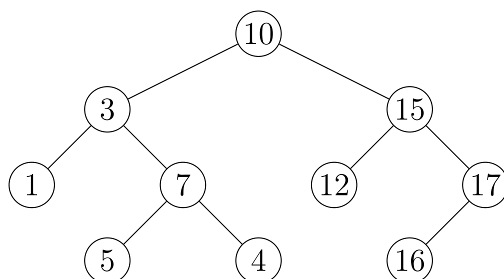
6.3 Persistente Søgetræer

6.3.1 Binært søgetræ

Dette afsnit handler om beskrivelsen og testen af et binært søgetræ [11, s. 426], som fremover vil blive betegnet BST. Ligesom i tilfældet med IHeap er det kun insert operationen der er interessant.

Et BST er et specielt træ, hvor enhver knudes venstre barn har en værdi, der er mindre end knuden og højre barn vil have en større værdi. Træets højde h svare til den længeste sti fra roden til et blad.

En søgning i træet foregår typisk fra roden efter en given værdi v. Hvis v er mindre end en knudes værdi forsættes søgningen fra venstre barn, ellers søges der mod højre. En søgning stopper når en knude med samme værdi som v er fundet.

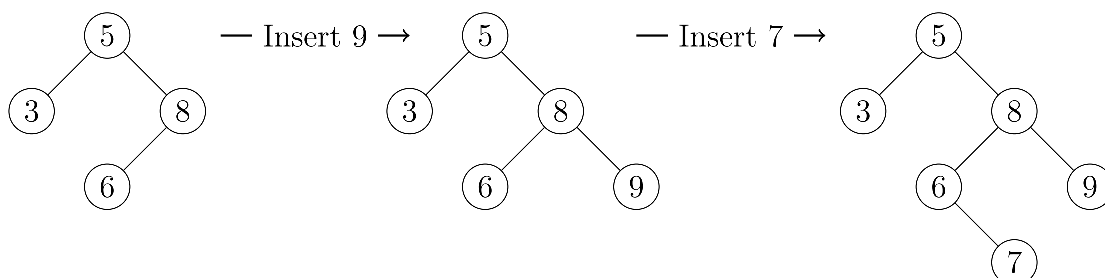


Figur 6.5: Eksempel på BST

I figur 6.5 ses et søgetræ, hvor h er fire. En søgning i træet efter værdien 12, foregår ved at søge mod højre fra roden 10, da $12 > 10$. I 15 vælges det venstre barn 12, som returneres.

Indsættelse

En indsættelse af værdien v i et binært søgetræ foregår ved at søge efter den værdi der skal indsættes, indtil der findes en knude, hvor v er mindre end værdien af knuden og knuden har venstre barn fri, eller en knude, hvor v er større end har højre barn fri.



Figur 6.6: Indsættelse af 9 og 7 i et eksisterene BST

Figur 6.6 viser indsættelse i et eksisterende søgetræ. Først søges der efter 9 i træet, der bliver indsat som højre barn til 8, da 8 har højre barn frit og $9 > 8$. Efterfølgende søges der efter 7 som indsættes som højre barn til 6.

Kompleksitet

Udførelsestiden for insert operationen er den samme som for søgning i træet. Da træet har højde h, vil en søgning i værste tilfælde tage $O(h)$ tid.

I bedste tilfælde vil enhver knude med undtagelse af bladene have præcist to børn og $h = \log n$. I værste tilfælde vil stien fra roden til et blad indeholde alle n knuder og $h = n$. I [36] argumenteres der for, at den forventede højde med en tilfældigt permuterede sekvens er $O(\log n)$.

Algorithm 14: BST.insert

```
   input : e
1  n ← root;
2  while true do
3  |  if e < n.value then
4  |  |  if n.hasLeftChild then
5  |  |  |  n = n.right;
6  |  |  else
7  |  |  |  n.right = e;
8  |  |  |  return;
9  |  |  end
10 |  else
11 |  |  if n.hasRightChild then
12 |  |  |  n = n.right;
13 |  |  else
14 |  |  |  n.right = e;
15 |  |  |  return;
16 |  |  end
17 |  end
18 end
```

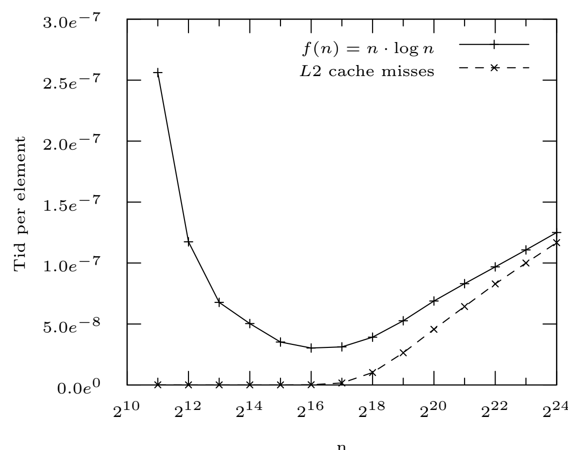
Eksperimenter

Figur 6.7 viser grafen for udførelsestiden ved indsættelse i BST. Grafen er vist med ratiotest $f(n) = n \log n$, da n indsættelser tager $O(n \log n)$. Kurven konvergerer ikke mod en konstant, som forventet, men stiger, hvilket vil sige at insert operationen bruger mere end $O(\log n)$ per insert. Analysen som ses i [36] er baseret på, at de indsatte elementer er en tilfældig permutation af tallene fra 1 til n, hvilket vil sige at elementerne er unikke. Dette er ikke gældende her, hvilket kan have en indflydelse på udførelsestiden. På grafen ses også $L2$ cache misses, hvor hvert målepunkt er divideret med $3 \cdot 10^7 \cdot n \log n$. Det ses tydeligt, at der er en stigning i $L2$ cache misses ved 2^{16}. En knude med *double* som værdi fylder $16B$, hvilket betyder, at der kan være cirka 2^{17} knuder i $L2$ cachen.

6.3.2 Persistering af Binært søgetræ

I dette afsnit gøres det binære søgetræ fra forrige afsnit partielt persistent. Den persistente version af BST, betegnet PBST, vil have de samme egenskaber som BST og det vil være muligt at søge og indsætte i samme kompleksistet som BST. Datastrukturen PBST bliver brugt i n searchtree.

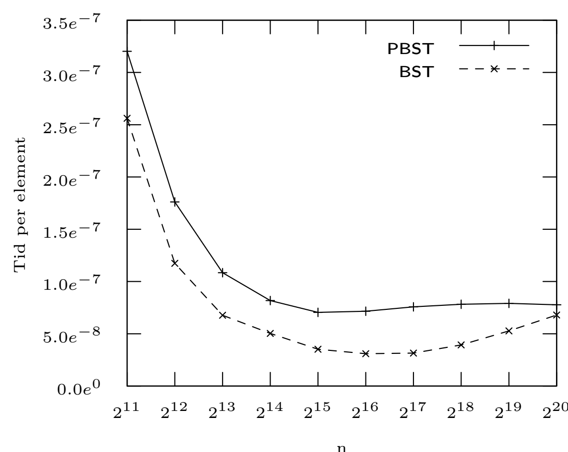
Fordi et BST ikke restruktureres efter en indsættelse, viser det sig at persisteringen er triviel. Et undertræ til knude n vil i BST kun eksistere i den $i$te version, såfremt dets rod eksistere. Det er derfor kun nødvendigt, at sætte versionsnumre på knuderne, for at gøre BST persistent.



Figur 6.7: BST: grafer for n

Eksperimenter

Figur 6.8 viser udførelsestiderne for n indsættelser i BST og PBST med ratio test $f(n) = n \log n$. Udfra figuren ses det, at kurven for PBST, konvergerer mod en konstant, hvilket betyder at udførelsestiden er $O(n \log n)$ som forventet.



Figur 6.8: PBST: grafer for n

6.3.3 Rød sorte træer

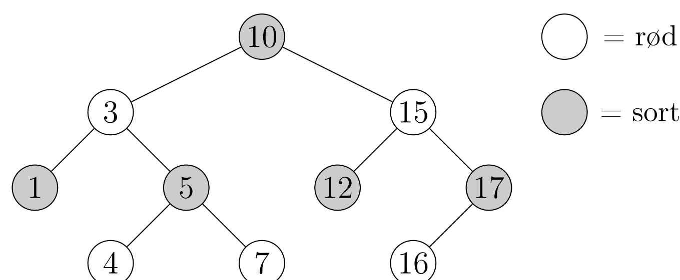
Selvom den forventede højde for et BST er $O(h)$, er tiden i værste tilfælde stadig $O(n^2)$. For at undgå dette, findes der forskellige måder, at restrukturere søgetræer på, så træerne bliver balancerede og højden bliver $O(\log n)$. I et *AVL-træ*[37] restruktureres træet hver gang der findes en knude, hvor højde forskellen mellem dens børn bliver større end én, hvorved højden bliver $O(\log n)$. I *splay træer*[38] opnås en amortiseret udførelsestid på $O(\log n)$ ved, at bruge en speciel *splay* operation hvergang en søgning, indsættelse eller slettelse sker i træet.

En restrukturering der ligner AVL træets, ses i *Rød sorte træer*[39], disse vil fremover blive betegnet RBT. I dette afsnit beskrives og testes en implementation af RBT.

Hovedideen i datastrukturen er, at farvelægge træets knuder, så hver knude enten er sort eller rød. Dernæst opstilles, der nogle egenskaber som træet skal opfylde efter hver operation.

- Rod Roden er altid sort.
- Rød-Sort Enhver rød knude har kun sorte børn.
- Sort højde Enhver sti fra roden til et blad, har det samme antal sorte knuder.

Udfra ovenstående egenskaber, kan det udledes, at højden på et RBT er $O(\log n)$, et bevis ses i[39]. I figur 6.9 ses træet fra forrige afsnit med BST, som er blevet restruktureret med RBT.

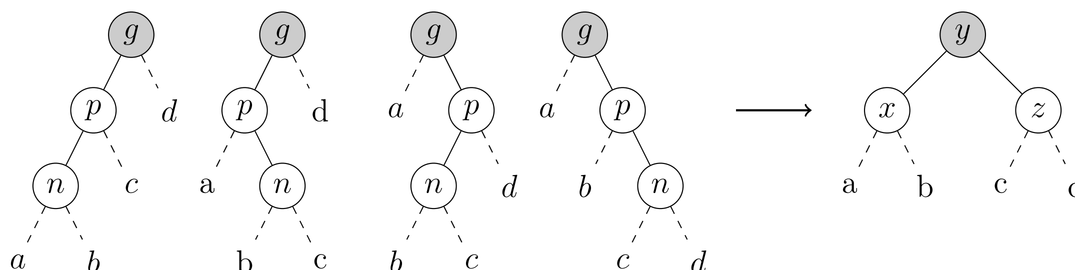


Figur 6.9: Eksempel på RBT

Indsættelse

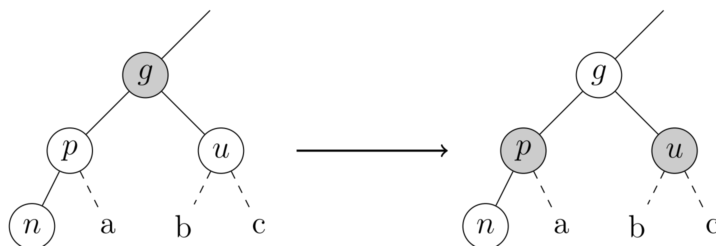
Til at starte med fungerer et RBT præcist som BST. Nye knuder bliver farvet røde til at starte med. Når en knuden indsættes, kan der derfor komme to røde knuder under hinanden, som bryder Rød-Sort egenskaben for træet. For at opretholde egenskaben restruktureres træet på to forskellige måder.

Type 1 Den indsatte n, dens forælder p og dens bedsteforælder g, sorteres efter deres værdi og gennavngives som x, y og z, hvor x er den mindste og z er den største af de tre. Disse sammensættes igen som ses i figur 6.10. Efter en type 1 restrukturering vil træet fra y og op overholde Rød-Sort egenskaben, da roden til de restrukturerede knuder er sort.



Figur 6.10: RBT: Type 1 restrukturering

Type 2 Den anden type restrukturing er blot en omfarvning af knuder. Hvis n har en onkel, u, og denne er rød, farves p og u sorte og g rød, med mindre g er roden af træet. Dette kan give et problem længere oppe i træet. Hvis gs forælder er rød skal dette også løses, igen som enten en type 1 ellet type 2 restrukturering.



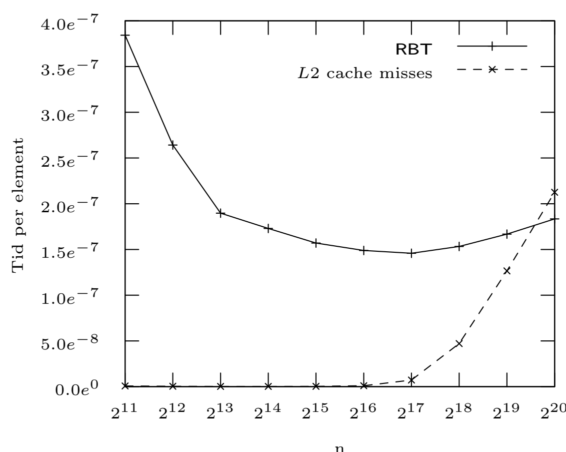
Figur 6.11: RBT: Type 2 restrukturering

Kompleksitet

Ligesom i BST tager det $O(h)$ at finde, den rigtige knude at indsætte under. Fordi træet opfylder de krævede egenskaber før indsættelsen, er højden $O(\log n)$. Dernæst laves der type 2 restruktureringer fra et blad til roden eller til den første type 1 restrukturering. Hver af disse tager konstant tid og da højden er $\log n$, vil disse restruktureringer tage $O(\log n)$. Den samlede udførelsestid for en indsættelse er derfor $O(\log n)$ tid.

Eksperimenter

Figur 6.12 viser indsættelser i et PRBT med ratio test $f(n) = n \log n$. Kurven konvergerer ikke, hvilket med stor sandsynlighed skyldes, at når n bliver større kommer der flere $L2$ cache misses. På grafen ses $L2$ cache misses, hvor hver målepunkt er divideret med $9 \cdot 10^6 n \log n$



Figur 6.12: RBT: grafer for n

6.3.4 Persistering af Rød sorte træer

I dette afsnit gøres RBT fra forrige afsnit partielt persistent. Den persistente udgave af RBT betegnet PRBT bruges i algoritmen n searchtree.

Til persisteringen af RBT bruges *Node-copy* metoden. I afsnit 6.2.2 diskuteres problemet ved at bruge *fat node* metoden. Denne samme problematik opstår her derfor undlades diskussionen af denne.

Ved indsættelse af et element i et RBT bruges knudernes left og right pointere til at finde det sted i træet, hvor den nye knude skal være. Fordi RBT restrukturerer datastrukturen løbende, skal begge disse pointere gøres persistente. Farven color af en knude bruges kun i den sidste udgave til, at afgøre hvordan der skal restruktureres, det er derfor ikke nødvendigt at gøres denne variabel persistent. Det viser sig, at det er bedst at lade være med at gøre parent pointeren persistent.

Implementation

Først gennemløbes træet for, at finde den rigtige knude at indsætte under, dernæst indsættes en rød knude og et muligt brud af Rød-sort egenskab opstår. I forrige afsnit blev de to forskellige typer af restrukturering beskrevet. De to typer kræver, at en given knude, n, kender den forælder, p, og dens bedsteforælder g. Udfra knuden n alene, er det ikke muligt at finde p og g. Men fordi en restrukturering kun sker fra n og op mod roden er det muligt, at opretholde en sti fra roden til n, mens der ledes efter den knude der skal indsættes under. Knuderne p og g til n, kan nemt findes ud fra denne sti.

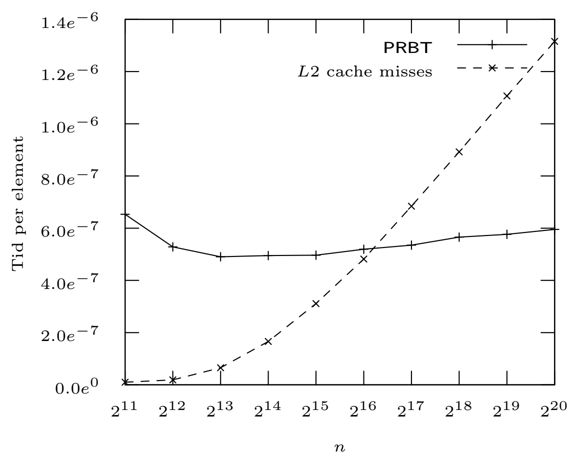
Fordelen ved at undlade, at gøre parent persistent er, at de eneste knuder der muligvis skal kopieres ligger på stien fra n til roden. Hvis hver knude derimod havde en parent pointer, så ville der opstå tilfælde, hvor store dele af grafen kopieres og en masse special tilfælde skulle håndteres. Risikoen for, at en knude skal kopieres reduceres desuden jo færre variabler der er persistente, hvilket igen medfører reduceret pladsforbrug. Det eneste problem der er ved at undlade parent pointeren er, at en inorder traversering af grafen også kræver, at man vedligerholder en sti.

Kompleksitet

Udførelsestiden for, at finde den knude, der skal indsættes er den samme som i RBT. Fordi det kun er knuder fra et blad til roden der kan kopieres, tager kopieringen og restrukturingen af træet $O(\log n)$ per indsættelse.

Eksperimenter

Figur 6.13 viser udførelsestiden ved indsættelser i et PRBT med ratio test $f(n) = n \log n$. Kurven konvergere ikke hvilket med stor sandsynlighed skyldes, at i takt med at grafen bliver større kommer der flere $L2$ cache misses. På grafen ses $L2$ cache misses hvor hver målepunkt er divideret med $4 \cdot 10^6 n \log n$



Figur 6.13: PRBT: grafer for n

Kapitel 7

Konklusion

7.1 Fremtidig arbejde

I [1] beskrives den algoritme som n-familien er baseret på. I artiklen bruges Fredriksons selektionsalgoritme [12] til at finde de k største knuder i Persistent IHeap. Et naturligt valg ville have været, at implementere og teste den algoritme, der er beskrevet i artiklen. I sin artikel beskriver Fredrikson to algoritmer; den ene bruger $O(k)$ tid, hvor den anden bruger $O(k \log \log k)$ tid. Selvom algoritmerne har en god teoretisk udførelsestid, er de forholdvis komplekse. Sammenligner man udførelsestiden med n priority er forskellen kun en faktor $\log k$ og ved n searchtree er forskellen en faktor $\log n$. Det er derfor tvivlsomt om Fredriksons algoritme rent faktisk klarer sig bedre i praksis. Dette er dog værd at undersøge.

Af de implementerede algoritmer må det siges, at n matrix er den algoritme, der klarer sig bedst i praksis når k er stor. Problemet med algoritmen er, at den ikke er specielt god, når n vokser. Dette problem beskrives i afsnit 4.4 der handler om n matrix og en mulig løsning foreslås. Denne foreslåede løsning består i, at bruge et søgetræ, i stedet for at iterere over ugyldige løsninger. Problemet ved løsningen er, at den trækker algoritmen i retning af n searchtree, dog med en flygtig datastruktur. Algoritmen n searchtree der klarer sig markant dårligere end n matrix. Det ville være interessant at konstruere en algoritme, der kun benytter sig af søgetræer, når algoritmen klarer sig tilpas dårligt.

I introduktionen til dette speciale blev der beskrevet en reduktion fra et graf problem [13] til k maksimum delsum problemet. En ide til en algoritme kunne være at finde en k korteste stier algoritme og se om denne vil kunne løse k maksimum problemet hurtigere end de implementerede. Selvom problemet er en reduktion og der sikkert er forbundet et overhead ved at gå en indirekte vej, er der en chance for en forbedring. Den k korteste sti algoritme[14] der bruges i [13], bruger Fredriksons selektionsalgoritme[12] til at finde stierne, så en direkte implementation af ideen, vil med stor sandsynlighed være dårligere end en direkte anvendelse af Fredrikson på Persistent IHeap.

Et væsenligt problem ved k maksimum delsum er, at det er svært at finde et scenarie eller en direkte applikation af problemet. Selvom der i introduktionen beskrives nogle områder, hvor det kan anvendes, er det stadig svært at afgøre, hvad n og k i praksis vil være. Det vil derfor være interessant, at implementere en større del af en applikation for at finde ud

af, hvilken af de forskellige algoritmer, der rent faktisk kan bruges i praktis. Et aspekt af denne implementation ville være at finde ud af, hvordan dataene er fordelt. Mange af de implementerede algoritmers udførelsestid er direkte afhængig af, både i analysen og i den praktisk udførelsestid, at dataen er uniformt fordelt. Dette er måske slet ikke er tilfældet i et virkeligt scenarie.

7.2 Konklusion

Formålet har været at finde en algoritme, der kan løse k maksimum problemet effektivt i praksis. Af den grund er der blevet implementeret en række forskellige algoritmer baseret på to forskellige ideer.

Den første ide, n^2-familien, er en meget naiv ide. Dog sikkert den man normalt ville bruge, hvis man skulle implementere en k maksimum delsums algoritme. Ideen går ud på at gennemløbe alle $\binom{n}{2} + n$ tripler og finde de k største iblandt disse. For at forbedre udførelsestiden er der anvendt forskellige teknikker og derved er algoritmen blevet forbedret trinvist. Den første algoritme n^2 priority, er en meget simpel algoritme, som nok er den algoritme de fleste ville implementere til at starte med. Problemet med denne algoritme er, at der indsættes $O(n^2)$ tripler i en prioritetskø og hver gang laves der en partiel sortering af $O(k)$ tripler. Denne sortering er dyr og bliver i n^2 select erstattet af en selektionsalgoritme, som køres på alle $\binom{n}{2} + n$ tripler. For at finde en god selektionsalgoritme er der implementeret seks forskellige varianter og den bedste er blevet udvalgt. Fordi der skal laves en selektion på $O(n^2)$ tripler, kræver algoritmen $O(n^2)$ plads. I n^2 select m reduceres pladsforbruget og derved forøges størrelsen af de problemer, der kan håndteres. Som en sidegevinst er algoritmen også blevet hurtigere, da den for et lille k kan udnytte $L2$ cachen mere effektivt. I n^2 historik reduceres antallet af selektioner og i n^2 block reduceres antallet af I/O operationer. Algoritmen n^2 block er ret tæt på det optimale, som kan forventes af en algoritme, der følger den første ide.

Den anden ide er baseret på en artikel [1], som løser k maksimum delsum problemet i den optimale udførelsestid. Udfra denne algoritme er der implementeret tre andre algoritmer, som klarer sig markant bedre end n^2 block. Til den første algoritme n priority, er der implementeret en speciel heap, som understøtter indsættelse i $O(1)$. Efterfølgende er denne blevet gjort partielt persistent ved at bruge en teknik fra[32]. I n priority erstattes den forholdvis komplekse selektionsalgoritme fra artiklen, med en prioritetskø. Dette gøres på bekostning af både tid og pladsforbrug. I den efterfølgende algoritme n searchtree reduceres pladsforbruget ved at bruge et partielt persistent søgetræ, men desværre klarer algoritmen sig ikke specielt godt i praksis. Til sidst koges ideen fra de to algoritmer ned til en simpel algoritme, som generelt klarer sig godt for store k værdier, men har det problem, at den praktiske udførelsestid svinger meget, for stort n.

Et af de mål der blev sat fra start, har været at algoritmerne skal kunne klare sig godt for alle værdier af k. Det største problem, som især er gældende ved n^2 familien, er at når k vokser, vokser pladsforbruget tilsvarende. Det har derfor ikke været muligt, at løse ret store problemer uden at algoritmene løber tør for hukommelse. Konsekvensen af dette problem har medført en balancegang imellem tids- og pladsforbrug. Problemet løses først seriøst i n-familien, hvor pladsforbruget reduceres til $O(n)$. I de fleste af de tests, der er blevet fortaget er n forholdvis lille og det har efterfølgende ikke været et problem.

Det kan konkluderes, at ud af de implementede algoritmer, er de bedste dem, som ligger i n-familien. De tre algoritmer har hver der styrker og svagheder. Algoritmen n priority er hurtig for næsten alle værdier af n og k, men da den har et pladsforbrug på $O(k)$, betyder dette, at hvis problemerne bliver ret meget større end de testede, vil algoritmen løbe tør for hukommelse. Algoritmen n searchtree har den fordel at den bruger $O(n)$ plads, men den er markant langsommere end både n priority og n matrix. Den sidste algoritme n matrix er hurtigere end de to forrige, men den opfører sig ustabilt og dens udførelsestid er desuden $O(n^2)$ worstcase, hvilket betyder, at der findes inputs, som får algoritmen til at køre markant langsommere end de to forrige. Der er derfor ikke fundet nogen entydigt optimal algoritme til, at løse k maksimum delsum problemet udfra de i specialet implementerede algoritmer.

Bilag A

Appendix

A.1 Eksperiment setup - uddybning

Cpuen er en Intel cpu med serie nummeret E6600, bemærk at der er to kerner, som deler den samme cache. Programmerne bruger dog kun den ene kerne.

```
>papi_avail

Vendor string and code   : GenuineIntel (1)
Model string and code    : Intel Core 2 (18)
CPU Revision             : 6.000000
CPU Megahertz            : 2401.965088
CPU Clock Megahertz      : 2401
CPU Clock Ticks  / sec   : 100
CPU's in this Node       : 2
Nodes in this System     : 1
Total CPU's              : 2
Number Hardware Counters : 5
Max Multiplex Counters   : 32
...

> papi_mem_info

L1 Instruction Cache:
  Total size: 32KB
  Line size: 64B
  Number of Lines: 512
  Associativity: 8

L1 Data Cache:
  Total size: 32KB
  Line size: 64B
  Number of Lines: 512
  Associativity: 8
```

```
L2 Unified Cache:
  Total size: 128KB
  Line size: 64B
  Number of Lines: 2048
  Associativity: 4

L3 Unified Cache:
  Total size: 4096KB
  Line size: 64B
  Number of Lines: 65536
  Associativity: 16
```

Til kompilering af programmerne er der brugt Gnus g++ compiler med følgende opsætning:

```
>g++ -v

Using built-in specs.
Target: i486-linux-gnu Configured with:
../src/configure -v --with-pkgversion='Debian 4.3.1-2'
--with-bugurl=file:///usr/share/doc/gcc-4.3/README.Bugs
--enable-languages=c,c++,fortran,objc,obj-c++ --prefix=/usr
--enable-shared --with-system-zlib --libexecdir=/usr/lib
--without-included-gettext --enable-threads=posix --enable-nls
--with-gxx-include-dir=/usr/include/c++/4.3 --program-suffix=-4.3
--enable-clocale=gnu --enable-libstdcxx-debug --enable-objc-gc
--enable-mpfr --enable-targets=all --enable-cld
--enable-checking=release --build=i486-linux-gnu --host=i486-linux-gnu
--target=i486-linux-gnu
Thread model: posix gcc version 4.3.1 (Debian 4.3.1-2)
```

Programmerne er compileret med følgende flag:

```
>g++ -Wall -O3 -DDSFMT_MEXP=19937 -std=gnu++98 -Wno-long-long -funroll-loops
 -Wextra -pedantic -I /usr/local/include/ -lperfctr -lpapi src/dSFMT/dSFMT.c
```

OSet eksperimenterne er kørt på en maskine konfigureret med *Debian 4.3.1-2* med *Linux version 2.6.24 (2.6.24-7)* Kernel.

Kernel er patchet med *perfctr-2.6.32* og målingerne er taget med *papi-3.6.0.*

Type	Størrelse	Går fra	Går til
short	$2B$	-32768	32767
unsigned short	$2B$	0	65535
int	$4B$	-2147483648	2147483647
unsigned int	$4B$	0	4294967295
long	$4B$	-2147483648	2147483647
unsigned long	$4B$	0	4294967295
float	$4B$	$1.17549e + 38$	$1.17549e - 38$
double	$8B$	$2.22507e + 308$	$2.22507e - 308$

Figur A.1: Typer i valgte udgave af $C++$

A.2 Profiler udskrifter

I dette afsnit vises profiler udskrifter fra de forskellige implementerede algoritmer. Profileren der er brugt er GNU's profiler "gprof", version 2.18.93.20081009. Bemærk at navnene på funktionerne i alle profiler udskrifterne er simplificerede.

```
Each sample counts as 0.01 seconds.
%    cumulative  self
time    seconds  seconds    calls     self    total  name
33.50      3.73     3.73 18002999     0.00     0.00  adjust_heap
28.25      6.88     3.15 36005999     0.00     0.00  push_heap
20.45      9.16     2.28 582223570    0.00     0.00  Triple::operator<
```

Figur A.2: n^2 priority: $n = 6000$ og $k = 100000$

```
Each sample counts as 0.01 seconds.
%    cumulative  self
time    seconds  seconds    calls     self    total  name
51.97      0.79     0.79        1     0.79     1.52  N2Select
25.66      1.18     0.39       27     0.01     0.03  Partition
12.50      1.37     0.19 50002322     0.00     0.00  Triple::operator<
```

Figur A.3: n^2 select: $n = 10000$ og $k = 100000$

```
Each sample counts as 0.01 seconds.
%    cumulative  self
time    seconds  seconds    calls     self    total  name
40.36      0.67     0.67        1     0.67     1.65  N2SelectM
27.11      1.12     0.45      618     0.00     0.00  Partition
15.06      1.37     0.25 69786616     0.00     0.00  Triple::operator<
```

Figur A.4: n^2 select m: $n = 10000$ og $k = 100000$

```
Each sample counts as 0.01 seconds.
%    cumulative  self
time    seconds  seconds    calls     self    total  name
62.50      0.35     0.35        1   350.00   555.00  N2Historik
16.07      0.44     0.09  5134817     0.00     0.00  Triple::operator<
14.29      0.52     0.08      104     0.77     1.97  Partition
```

Figur A.5: n^2 historik: $n = 10000$ og $k = 100000$

```
Each sample counts as 0.01 seconds.
%     cumulative   self
time     seconds   seconds     calls      self     total   name
65.52       0.38      0.38         1    380.00    570.00   N2Block
17.24       0.48      0.10       118      0.85      1.61   Partition
 6.90       0.52      0.04  12851887      0.00      0.00   Triple::operator<
```

Figur A.6: n^2 block: $n = 10000$ og $k = 100000$

```
Each sample counts as 0.01 seconds.
%     cumulative   self
time     seconds    seconds     calls      self     total   name
25.69       2.06       2.06 347263526      0.00      0.00   vector::iterator
22.07       3.83       1.77   5000000      0.00      0.00   adjust_heap
11.22       5.71       0.90  93110925      0.00      0.00   Delta::operator<
```

Figur A.7: n priority: $n = 10000$ og $k = 5000000$

```
Each sample counts as 0.01 seconds.
%     cumulative   self
time     seconds   seconds     calls      self     total   name
19.39       0.72      0.72   5005000      0.00      0.00   adjust_heap
13.64       1.24      0.51  84308081      0.00      0.00   Delta::operator<
 7.89       1.53      0.29   5010000      0.00      0.00   InorderWalk::next()
```

Figur A.8: n searchtree med PRBT: $n = 10000$ og $k = 5000000$

```
Each sample counts as 0.01 seconds.
%     cumulative   self
time     seconds   seconds     calls      self     total   name
38.01       1.01      1.01   5005000      0.00      0.00   adjust_heap
16.85       1.94      0.45  10005000      0.00      0.00   push_heap
 3.37       2.42      0.09   5000000      0.00      0.00   NMatrix::nextToken
```

Figur A.9: n matrix: $n = 10000$ og $k = 5000000$

Litteratur

[1] G. S. Brodal and A. G. Jørgensen. A linear time algorithm for the k maximal sums problem. In *Proc. 32nd International Symposium on Mathematical Foundations of Computer Science*, volume 4708 of *Lecture Notes in Computer Science*, pages 442–453. Springer Verlag, Berlin, 2007.

[2] J. Bentley. Programming pearls: algorithm design techniques. *Commun. ACM*, 27(9):865–873, 1984.

[3] K.M. Chung and H.I. Lu. An optimal algorithm for the maximum-density segment problem. *SIAM J. Comput.*, 34(2):373–387, 2005.

[4] M. Csuros. Maximum-scoring segment sets. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 1(4):139–150, 2004.

[5] F. Bengtsson and J. Chen. Computing maximum-scoring segments in almost linear time. In *In Proceedings of the 12th Annual International Computing and Combinatorics Conference, volume 4112 of Lecture Notes in Computer Science*, pages 255–264, 2006.

[6] S. E. Bae and T. Takaoka. Algorithms for the problem of k maximum sums and a vlsi algorithm for the k maximum subarrays problem. *Parallel Architectures, Algorithms and Networks, 2004. Proceedings. 7th International Symposium on*, pages 247–253, May 2004.

[7] F. Bengtsson and J. Chen. Efficient algorithms for k maximum sums. *Algorithmica*, 46(1):27–41, 2006.

[8] S. E. Bae and T. Takaoka. Improved algorithms for the k-maximum subarray problem for small k. *In Proceedings of the 11th Annual International Conference on Computing and Combinatorics, volume 3595 of LNCS*, 49:621–631, 2005.

[9] T.C. Lin and D. T. Lee. Randomized algorithm for the sum selection problem. *Theor. Comput. Sci.*, 377(1-3):151–156, 2007.

[10] S. E. Bae and T. Takaoka. Improved algorithms for the k-maximum subarray problem. *Comput. J.*, 49(3):358–374, 2006.

[11] D. E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[12] G. N. Frederickson. An optimal algorithm for selection in a min-heap. *Inf. Comput.*, 104(2):197–214, 1993.

[13] G. S. Brodal, K.M. Chao, and H.F Liu. Personal communication, 2007.

[14] D. Eppstein. Finding the k shortest paths. *SIAM J. Comput.*, 28(2):652–673, 1999.

[15] ISO/IEC. C - approved standards - iso/iec 9899. http://www.open-std.org/jtc1/sc22/wg14/www/standards.

[16] M. Saito and M. Matsumoto. Simd-oriented fast mersenne twister (sfmt), 2008. http://www.math.sci.hiroshima-u.ac.jp/ m-mat/MT/SFMT/index.html.

[17] M. T. Goodrich and R. Tamassia. *Algorithm Design: Foundations, Analysis, and Internet Examples.* John Wiley and sons, 2001.

[18] P. Mucci and ICL group. Papi (performance application programming interface), 2008. http://icl.cs.utk.edu/papi.

[19] M. Pettersson. Linux performance counters driver, 2007. http://user.it.uu.se/m̃ikpe.

[20] R. W. Floyd. Algorithm 245: Treesort. *Commun. ACM*, 7(12):701, 1964.

[21] ISO/IEC. Programming languages - C++ - international standard 14882. http://www.open-std.org/JTC1/SC22/WG21/docs/standards.

[22] K. Kaligosi and P. Sanders. How branch mispredictions affect quicksort. In *ESA'06: Proceedings of the 14th conference on Annual European Symposium*, pages 780–791, London, UK, 2006. Springer-Verlag.

[23] R. Sedgewick. Implementing quicksort programs. *Commun. ACM*, 21(10):847–857, 1978.

[24] C. A. R. Hoare. Algorithm 65: find. *Commun. ACM*, 4(7):321–322, 1961.

[25] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321, 1961.

[26] P. Kirschenhofer and H. Prodinger. Comparisons in hoare's find algorithm. *Comb. Probab. Comput.*, 7(1):111–120, 1998.

[27] L. Devroye. On the probablistic worst-case time of "find". *Algorithmica*, 31:291 – 303, 07 2001.

[28] R. Sedgewick. The analysis of quicksort programs. *Acta Informatica*, pages 327–355, 1977.

[29] D. R. Musser. Introspective sorting and selection algorithms. *Softw. Pract. Exper.*, 27(8):983–993, 1997.

[30] C. Martínez, D. Panario, and A. Viola. Adaptive sampling for quickselect. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 447–455, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[31] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E Tarjan. Two papers on the selection problem: Time bounds for selection [by manual blum, robert w. floyd, vaughan pratt, ronald l. rivest, and robert e. tarjan] and expected time bounds for selection [by robert w. floyd and ronald l. rivest]. Technical report, Stanford, CA, USA, 1973.

[32] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. In *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 109–121, New York, NY, USA, 1986. ACM.

[33] R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, April 1985.

[34] D. D. Sleator and R. E. Tarjan. Self adjusting heaps. *SIAM J. Comput.*, 15(1):52–69, 1986.

[35] C. A. Crane. *Linear lists and priority queues as balanced binary trees*. PhD thesis, Stanford, CA, USA, 1972.

[36] L. Devroye. A note on the height of binary search trees. *J. ACM*, 33(3):489–498, 1986.

[37] G. M. Adel'son-Vel'skii and Y. M. Landis. An algorithm for the organization of information. doklady akademii nauk sssr. In *English translation in Soviet Math. Dokl*, pages 146–263, 1962.

[38] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[39] R. Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Inf.*, 1:290–306, 1972.