Praktisk brug af dynamisk sampling i data streams

Speciale af Kristian Dorph-Petersen

Januar 2007

Resumé

In the last decade, the growing usage of the Internet and networks in general, has put more focus on how to effectively monitor the data streams such networks consists of. The motivation for wishing to monitor the traffic on a network could be due to security concerns, management, error handling etc. Whatever the reason, the need is there for effective ways to do the monitoring.

In this thesis the practical value of sampling of data streams as a tool in monitoring networks is examined. This is done by first discussing how to define data streams, and what problems are faced when you wish to log the traffic they consist of. Examples of already published ideas of how to do such sampling are then presented.

The focus of the main part of the thesis is on the practical use of the data structure presented in [1], in the context of network monitoring. The data structure is explained in details, although no attempt is made on verifying the theory the data structure is based on. A program implementing the data structure is made and described. This program simulates a monitoring device keeping track of connections being made to a web server. Access logs from the web server running the domain 246.dk are used to make this simulation as realistic as possible. Through these simulations the program is used to evaluate, whether or not the data structure from [1] is able to provide an acceptable amount of information compared to standard lossless logging methods.

Based on the simulations, the resulting evaluation of the data structure is negative. The chosen way of using the data structure does not lead to any good results. The amount of space needed by the data structure, if more than a few samples are needed, is greater than a similar lossless solution. The data structure does work well at gathering a few random samples from data streams but it is not enough in this context. Further more, the way the data structure works internally leads to it being concluded that no good uses of the data structure (in respect to network monitoring) could be found.

Indhold

1 Indledning **5**
1.1 Specialets opbygning 7
1.2 Sprogbruget i specialet 8
1.3 Den vedlagte CD 8
1.4 Den reviderede udgave af [1] 8

2 Data streams **10**
2.1 Hvad er en data stream 10
2.2 Modeller for data streams 11
2.3 Hvorfor logge data streams? 12
2.4 Algoritmisk sampling 14

3 Eksempler på brug af sampling **16**
3.1 To typer af artikler 16
3.2 Heavy hitters sampling 17
3.2.1 Problem 17
3.2.2 Løsning 18
3.3 Reservoir sampling 19
3.3.1 Problem 19
3.3.2 Løsning 19
3.4 Subset-sum sampling 20
3.4.1 Problem 20
3.4.2 Løsning 20
3.5 Sampling med wavelets 21
3.5.1 Problem 21
3.5.2 Løsning 22
3.6 QuickSAND 22
3.6.1 Problem 22
3.6.2 Løsning 23
3.7 Gigascope 23
3.7.1 Problem 24
3.7.2 Løsning 25

4 Den valgte datastruktur **26**
4.1 Matematikken i artiklen 26
4.2 Det grundliggende problem 26
4.3 Problemet formelt 28
4.4 Gennemgang af datastrukturen 29
4.4.1 Overordnet opbygning 29
4.4.2 Unique Element (UE) 30
4.4.3 Destinct Elements (DE) 32
4.4.4 Update() 32

4.4.5 Sample() . . . 33
4.4.6 Korrektheden af datastrukturen . . . 34
4.5 Destinct Elements i detaljer . . . 35
4.5.1 Den grundliggende ide . . . 35
4.5.2 Bedre præcision . . . 37
4.5.3 PCSA algoritmen . . . 38
4.5.4 Den nye datastruktur for DE i [2] . . . 39
4.6 Min anvendelse af datastrukturen . . . 40
4.6.1 Forventninger til datastrukturen . . . 41

5 Min implementation 43
5.1 Målet for implementationen . . . 43
5.2 Valget af programmerings sprog . . . 44
5.3 Fejlhåndtering i programmet . . . 45
5.4 Programmets opbygning . . . 45
5.4.1 Initialisering . . . 46
5.4.2 Kørsel . . . 47
5.4.3 Sliding window i loggen . . . 48
5.5 Kodens opbygning . . . 49
5.5.1 Hovedprogrammet . . . 49
5.5.2 Datastrukturen . . . 52
5.5.3 Visualiseringen . . . 61
5.6 Kode af speciel interesse . . . 63
5.6.1 Sliding window i datastrukturen . . . 63
5.6.2 Hash funktionen . . . 64
5.6.3 Rekursiv sampling . . . 66
5.6.4 DE strukturen og sliding window . . . 68
5.6.5 DE strukturen ved 1 element . . . 69
5.7 Korrektheden af implementationen . . . 71
5.7.1 Handlingsforløbet . . . 71
5.7.2 Overløb og pladsforbrug . . . 72
5.8 Sværhedsgraden af implementeringen . . . 73

6 Kørsler med implementationen 74
6.1 Typer af tests . . . 74
6.2 Fuldstændigheden . . . 75
6.2.1 Opsætning . . . 75
6.2.2 Resultater . . . 76
6.3 Måden der samples . . . 78
6.3.1 Opsætning . . . 79
6.3.2 Resultat . . . 79

7 Er datastrukturen værd at bruge 82
7.1 I min anvendelse . . . 82
7.2 I andre netværks sammenhænge . . . 83

8 Konklusion 85

A Vejledning til mit program 87
A.1 Kompilering . . . 87
A.2 Kørsel af programmet . . . 87
A.3 Brugervejledning . . . 87

B PCSA pseudokode 90

C Kode i *MainForm.cs* **91**

D Kode i *DataTypes.cs* **99**

E Kode i *StatisticControls.cs* **104**

F Kode i *Functions.cs* **111**

G Kode i *DisplayPanel.cs* **112**

H Kode i *TestCode.cs* **115**

Figurer

4.1 Den initielle vektor over mængden af elementer 26
4.2 Vektoren opdateres via en data stream 27
4.3 Fire mulige svar for observanten. Én af værdierne større end 0 eller FAIL. 27
4.4 Flere "parallelle"observanter der alle har de 4 mulige svar 28
4.5 Datastrukturens opbygning i [1] 29
4.6 Datastrukturens opbygning i [2] 30

5.1 Screenshot - Programmet ved start 46
5.2 Screenshot - Programmet efter initialisering 47
5.3 Screenshot af grafikken 62

6.1 Screenshot - Eksempel på et samplet søjlediagram 74
6.2 Screenshot - Sampling efter 3 døgn 79
6.3 Screenshot - Sampling efter 6 døgn 80
6.4 Screenshot - Sampling efter 9 døgn 80
6.5 Screenshot - Sampling efter 12 døgn 81
6.6 Screenshot - Sampling efter 15 døgn 81

A.1 Screenshot - Programmet ved start 88
A.2 Screenshot - Programmet efter tryk på Accept 88

Kapitel 1

Indledning

I efterhånden mange år er computere blevet forbundet i netværk. Lige siden det oprindelige ARPA[1] netværk blev sat i gang og frem til i dag, er et stigende antal computere år for år blevet forbundet i kraftigere og kraftigere netværk. På trods af at de fundamentale principper er de samme, i Internettet vi kender i dag, som i ARPA netværket, så var der nok ingen af de oprindelige stiftere, der havde forestillet sig den udvikling, vi specielt har oplevet de seneste 10 år. I takt med den kraftige vækst af Internettet er der løbende opstået nye behov for redskaber til vedligeholdelse af moderne netværk. Ét af disse behov er bedre logning af netværkstrafikken.

Logning, af hvad der sker på et netværk, er ikke noget nyt og benyttes med rette i mange situationer. Logning er et af de vigtigste redskaber for IT-ansvarlige, hvad enten det drejer sig om logning af aktiviteten på en enkelt server eller på et større firmanetværks backbone-forbindelse til Internettet. Logning kan bruges til at lokalisere fejl, holde øje med kapacitets problemer i netværket, hjælpe med opklaring af misbrug osv. I de senere år med terror debatten i samfundet har logning desuden fået endnu større fokus, da der er talt om ved lov at kræve omfattende registrering af alle borgeres brug af Internettet.

Der kan skrives - og bliver skrevet - meget om, i hvor stor udstrækning vi bør overvåge os selv og andre, men faktum er, at logning allerede nu er et meget brugt redskab, og at det i hvert fald ikke vil blive mindre brugt i fremtiden. Den etiske side af alt dette er dog noget, der fortsat vil blive diskuteret mange år fremover.

Når vi kommer til den praktiske side af logning, er der dog to meget væsentlige ting, der skal tages hånd om, og som har stor indflydelse på, hvor brugbar logning er i praksis. Den første ting er diskussionen om, hvad der egentlig skal logges. Det er meget nemt at sige, at "al aktivitet på Internettet skal registreres", men for programmørerne, der i praksis skal implementere det, er det ikke helt så nemt. Udover naturlige begrænsninger i hvad der kan logges - dataen, der flyder gennem en netforbindelse, viser ikke altid lige godt, hvad der egentlig foregår - så er det også et stort problem, at man ikke kan forudsige alt, hvad der i fremtiden skal kunne ses ud fra loggen. En log er kun så god, som dét man har valgt at registrere. Står man senere og vil undersøge noget, man ikke har valgt at registrere, så kommer man ikke så langt. For at imødegå fremtidige behov er løsningen derfor ofte at logge så meget som overhovedet muligt. Det leder os over til det andet store problem ved logning: Plads.

For at logning kan bruges til noget, er det nødvendigt at gemme det registrerede data et eller andet sted. Typisk vil dette ske på harddiske tilknyttet computeren, der foretager logningen, men afhængig af hvor meget der logges, og hvor længe det skal gemmes, kan andre lagermedier såsom CD'ere, DVD'ere etc. også benyttes. Det

[1] http://en.wikipedia.org/wiki/Defense_Advanced_Research_Projects_Agency

er dog her, vi møder den største begrænsning for logning. Den optimale logning - ud fra et teknisk synspunkt, ikke et etisk - af f.eks. en internet forbindelse er at gemme al datatrafik på ubestemt tid. Den mængde data, der hvert sekund sendes på Internettet i bare Danmark, er dog så enorm, at det i alle praktiske henseender (økonomisk og fysisk) er umuligt.

Man er altså nødt til at vælge at logge bestemte ting, under hensyntagen til hvor meget plads det registrerede kan fylde, og det er her, dette speciale kommer ind i billedet. Der findes mange forslag til, hvad man kan gøre for at logge så effektivt (mht. pladsforbruget) som muligt og en del af disse involverer brug af sampling. Sampling vil sige, at man i stedet for at prøve at registrere alt det, man finder interessant, så nøjes man i stedet med et antal stikprøver, der kan sige noget om det, man er interesseret i.

Som mit speciales overskrift antyder, har jeg valgt sampling af data streams i netværk som fokus for specialet. Mere specifikt har jeg fundet en artikel, der beskriver et interessant forslag, til hvordan en sådan sampling kan foretages. Jeg har så i specialet arbejdet med at undersøge, om jeg synes forslaget er brugbart. Artiklen, der er tale om, er:

- **"Sampling in Dynamic Data Streams and Applications"af Gereon Frahling, Piotr Indyk og Christian Sohler [1].**

Med udgangspunkt i at det fylder for meget at gemme al data, beskriver denne artikel i stedet et teoretiske fundament for en metode til at sample en stream af data og derefter på så begrænset plads som muligt opretholde statistisk information om det observerede.

Valget af præcis [1] som grundlag for specialet er lidt tilfældigt. Da jeg oprindeligt skulle vælge emne for specialet, kiggede jeg på artikler inden for Computational Geometry. Dvs. jeg læste de seneste publicerede artikler inden for det felt for at se, om der var noget, jeg syntes virkede specielt interessant. Jeg fandt denne artikel, da jeg gik igennem *Annual ACM Symposium on Computational Geometry 2005*. Hvad der fangede min opmærksomhed, var forfatternes forslag i artiklens indledning om, at den beskrevne datastruktur kan anvendes i en netværks sammenhæng. Netværk har altid interesseret mig, og det virkede for mig spændende, at en datastruktur fremstillet med et geometrisk udgangspunkt for øje skulle kunne bruges i denne anden sammenhæng.

Efterhånden som jeg fik kigget mere på samplings-feltet, specielt gennem [3], fandt jeg ud af, at algoritmer fra Computational Geometry åbenbart er begyndt at vinde interesse inden for området [3, afsnit 9.4]. Jeg valgte derfor endeligt, at målet for dette speciale skulle være at undersøge den praktiske værdi i netværks sammenhæng af datastrukturen fra [1]. Jeg valgte altså ikke artiklen ud fra en forhånds ide om, at den rent faktisk ville være en god løsning i denne sammenhæng. Jeg kunne principielt set lige så godt have taget udgangspunkt i f.eks. en af de artikler, jeg nævner i kapitel 3, men det var altså [1], der først fangede min interesse.

Med det nævnte fokus for specialet i mente er mine mål følgende:

- At give en introduktion til brugen af sampling til logning i netværks sammenhæng. Blandt andet ved at beskrive eksempler på forslag til og eksisterende brug af sampling af netværks streams.

- Gennemgå og forklare den foreslåede datastruktur fra [1] i detaljer.

- Implementere en udgave af datastrukturen og dermed belyse hvor svært/nemt det beskrevne i [1] er at omsætte til et reelt fungerende program.

- Vurdere datastrukturens praktiske værdi ud fra simuleret brug på test data og det tilhørende pladsforbrug.

For at opnå et så realistisk billede som muligt af algoritmens muligheder har jeg som test data fået adgang til et halvt års log over aktiviteten på webserveren for domænet 246.dk[2]. Denne log indeholder informationer om andre computeres brug af serveren i perioden 31. december 2005 til 29. juni 2006, og har samlet en størrelse på ca. 1,2 GB. Ved at bruge denne log som grundlag for mine tests af datastrukturen fra [1] er det mit håb, at disse tests så realistisk som muligt simulerer en reel logning af en kørende netforbindelse. Ud fra resultaterne af disse tests vil jeg så basere min vurdering af datastrukturen.

Mit grundliggende ønske med specialet har været at opnå en praktisk indsigt i brugen af sampling, specifikt i form at datastrukturen fra [1]. Det betyder, at jeg primært i mit arbejde har fokuseret på praktiske måder at anskue problemet på. Der at derfor steder i specialet, hvor jeg har valgt kun at give en overfladisk gennemgang af tilhørende teori. Det er sket de steder, hvor jeg ikke har følt, at en mere omfattende gennemgang ville bidrage til en bedre forståelse af det praktiske problem.

1.1 Specialets opbygning

Specialet har følgende opbygning:

- Kapitel 2 har til formål at give en formel introduktion til data streams. Desuden kommer jeg nærmere ind på, hvorfor der i det hele taget er interesse for at logge sådanne streams, og hvad brugen af sampling kan bidrage med.

- Kapitel 3 beskriver eksempler på brug af sampling i forbindelse med overvågning af netforbindelser. Målet med kapitlet er, at give en introduktion til forskellige måder dette problem kan gribes an på, og hvor succesfulde disse har været. Der vil dog kun være tale om en overordnet gennemgang, af de artikler der ligger til grund for kapitlet, da jeg ikke forsøger at give en fyldestgørende beskrivelse af teorien bag de beskrevne løsninger.

- Kapitel 4 er en detaljeret gennemgang af datastrukturen beskrevet i [1], da den som sagt er mit primære fokus for specialet. Jeg belyser de punkter, hvor jeg finder at forfatterne overlader lidt for meget til læseren selv, og i det hele taget forsøger jeg at sikre, at der ikke er nogen tvivl om, hvordan den beskrevne datastruktur fungerer og implementeres.

- Kapitel 5 gennemgår min implementation af datastrukturen. Hovedvægten ligger her på at beskrive, i hvilken sammenhæng jeg har valgt at benytte strukturen med loggen, og hvordan jeg har implementeret dette. Desuden kommer jeg ind på, hvordan jeg har løst problemer opstået som følge af evt. mangelfulde oplysninger i [1]. Jeg giver en overordnet gennemgang af min programkode, men jeg antager, at læsere af dette speciale allerede ved noget om programmering. Jeg går derfor ikke i detaljer med de trivielle dele af koden, der - rent liniemæssigt - udgør den største del.

- Kapitel 6 beskriver brugen af min implementation på test dataen og resultaterne heraf. Målet er, at se hvordan datastrukturen fungerer i den valgte sammenhæng, og ud fra dette give en forklaring på evt. uventede resultater.

- Kapitel 7 indeholder min vurdering, af hvor brugbar jeg synes datastrukturen er i den valgte sammenhæng, på baggrund af resultaterne i kapitel 6. Desuden giver jeg en vurdering af, hvor god, jeg tror, den vil være i andre sammenhænge.

[2]En stor tak til Kai Birger Nielsen fra Datalogisk Institut Aarhus Universitet for at stille loggen til rådighed!

- Kapitel 8 er min konklusion på specialet. Her fremhæver jeg de vigtigste erfaringer og resultater, jeg har opnået.

1.2 Sprogbruget i specialet

Jeg har valgt at skrive specialet på dansk, men det giver nogle problemer med de mange engelske udtryk, der findes i denne branche. Nogle af disse kan nemt oversættes til dansk, men for de flestes vedkommende er det de engelske udtryk, der bruges i daglig tale. Ét oplagt eksempel er ordet "harddisk"som oprindeligt hed "pladelager"på dansk, men som der nok ikke er nogen, der bruger længere. Som udgangspunkt bruger jeg derfor de ord (dansk eller engelsk), jeg synes bliver brugt i daglig tale. Hvis der er tilfælde, hvor jeg føler, at der kan være tvivl om, hvordan et givent udtryk skal forstås, vil jeg første gang, det mødes, forklare kort hvad jeg mener.

1.3 Den vedlagte CD

Sammen med specialet er vedlagt en CD indeholdende bl.a. programkoden for min implementation, en eksekverbar fil for programmet, specialet i .pdf format og screenshots af de i kapitel 6 beskrevne test kørsler. Jeg har skrevet specialet med udgangspunkt i, at man ikke behøver kigge på indholdet af CDen, men ønskes det, er det altså muligt.

En brugervejledning til mit program er givet i bilag A. I bilaget er også informationer om hvilke krav, en computer skal opfylde for at kunne køre programmet.

1.4 Den reviderede udgave af [1]

Undervejs i specialet ved arbejdet med min implementering af datastrukturen fra [1] har jeg til tider følt, at artiklens forfattere nogle steder var temmelig vage i deres beskrivelser/forklaringer af, hvordan datastrukturen i praksis skulle implementeres. Det havde naturligvis indflydelse på min implementering, men dog ikke mere end at jeg selv kunne finde på løsninger, de steder hvor det var nødvendigt. Jeg så det som en naturlig del af arbejdet med artiklen og i bund og grund blot som en forlængelse af min undersøgelse af datastrukturens praktiske værdi. Jeg følte, jeg forstod alle de grundliggende ideer i [1], og så længe jeg kunne få min implementation til at stemme overens med disse, så jeg ikke noget galt i de vage steder.

I forbindelse med skrivningen af kapitel 4 blev jeg tvunget til at se nærmere på teorien i artiklen for at kunne beskrive den korrekt. På det tidspunkt var min implementering næsten færdig, og jeg arbejdede derfor på denne skrevne rapport. I den forbindelse blev jeg opmærksom på nogle mærkelige ting i [1], som jeg syntes kun kunne være deciderede fejl. Det bedste eksempel på dette er Lemma 2.1 i [1, afsnit 2]. Det er i dette lemma, at forfatterne sammenfatter teorien for datastrukturen, de konstruerer. Gengivet ordret siger lemmaet:

- Lemma 2.1. *Given a sequence of update operations on a vector* $x : [U] \rightarrow [M]$, *there is a streaming algorithm that with probability* $1 - 1/\delta$ *returns an element* $r \in Supp(x)$ *such that* $Pr[r = i] = 1/\|x\|_0 \pm \delta$ *for every* $i \in Supp(x)$. *The algorithm uses* $O(\log^2(MU/\delta))$ *space.* □

Dét, jeg bemærkede, var brugen af δ på flere forskellige måder. Først bruges δ i konstruktionen $1 - 1/\delta$ og ved læsning af lemmaet lyder det til, at der angives en sandsynlighed mellem 0 og 1. For at det kan lade sig gøre, skal det gælde, at $\delta \geq 1$. Det er der ikke noget galt i, men problemet er, at kort efter i lemmaet optræder δ

igen i konstruktionen $Pr[r = i] = 1/\|x\|_0 \pm \delta$. Denne konstruktion fremstår også som en sandsynlighed mellem 0 og 1, men denne gang lader det til, at δ repræsenterer små værdier, og i hvert fald noget som kræver at $\delta < 1$, for at det skal kunne give mening.

Denne brug af δ på flere måder, som altså ikke sammen giver mening, gjorde mig opmærksom på, at der som et minimum er en trykfejl i [1]. Trykfejl i artikler er selvfølgelig ikke noget nyt, men jeg blev bekymret over, at denne fejl optrådte i det lemma, som skulle sammenfatte dét, man opnår, med datastrukturen jeg havde implementeret. Jeg syntes ikke, jeg bare kunne ignorere en fejl dette centrale sted, og det gav mig to muligheder: Enten måtte jeg selv forsøge at verificere teorien bag datastrukturen, eller også måtte jeg finde en anden artikel, hvor andre havde gjort det. Jeg var ikke vild med tanken om selv at skulle rekonstruere teorien - jeg var trods alt ret langt henne i tidsplanen for specialet - så jeg valgte at starte med at lede efter andre artikler, der rettede fejlen. Det viste sig at være et godt valg med et lidt overraskende resultatet.

Ved simple søgning på Internettet ledte jeg som udgangspunkt efter artikler, der refererede til [1]. Mit håb var, at hvis andre havde brugt artiklens resultater til noget konkret, at de så også havde bemærket samme fejl som mig. Under denne søgning gik jeg tilfældigt ind på den ene forfatter (Christian Sohler) til [1]s hjemmeside [3], hvor der var en liste over det materiale, han havde været med til at publicere. På denne side var der under året 2005 nævnt:

- Gereon Frahling, Piotr Indyk, Christian Sohler

  Sampling in Dynamic Data Streams and Applications (full version)

  In: Proceedings of the 37th ACM Symposium on Theory of Computing (STOC), pp. 209-217, 2005.

Den understregede del var et link til et .pdf dokument. Jeg ved stadig ikke helt, hvorfor jeg trykkede på linket, da jeg også nu vil mene, at ovenstående fremstår som et link til [1], men uanset årsagen trykkede jeg på det og fandt [2].

[2] indeholder, hvad der bedst kan beskrives som en 2. udgave af [1]. Der er ikke angivet nogen forfattere, så udover Christian Sohler ved jeg ikke, om Frahling og/eller Indyk har været involveret i den. Den står anført som værende fra 9. november 2005, hvilket passer med, at den er blevet skrevet efter [1], der blev publiceret juni 2005. Det første, der slog mig, var, at hvor [1] var på 8 sider, så var [2] på 19. Der var altså væsentligt mere indhold i [2]. Dokumentet følger samme udformning som [1], men der er tilføjet en del detaljer og rettet på den bagved liggende teori. Lemma 2.1 fra [1] er f.eks. blevet rettet til noget, der lyder en del mere rigtigt.

Det eneste trælse ved at finde [2] på et så sent tidspunkt er, at der også er blevet ændret i selve måden, den beskrevne datastrukturen skal implementeres på. Dette er lidt - set i bakspejlet - irriterende, eftersom jeg har brugt en del tid på løsninger til problemer i [1], der så viser sig allerede at være løst i [2]. Disse løsninger udgør en væsentlig del af mit arbejde med [1], og derfor vil jeg i specialet stadig bruge den oprindelige artikel som udgangspunkt for mine beskrivelser. Jeg vil dog beskrive den opdaterede teori i [2] de steder, hvor den er mere rigtig. Yderligere vil jeg beskrive de steder, hvor instruktionerne til implementationen er blevet ændret fra [1], og hvad ændringerne i praksis har haft af betydning for mit program.

[3] http://wwwcs.uni-paderborn.de/cs/ag-madh/WWW/english/Sohler/publications.html

Kapitel 2

Data streams

Før end jeg kan undersøge nærmere, hvad det vil sige at sample data streams, er det nødvendigt at få sat på plads, hvad der egentlig menes med "data streams". Dette kapitel beskriver derfor data streams, og i hvilken sammenhæng de er af interesse for specialet.

2.1 Hvad er en data stream

En god definition for data streams er givet i [3, kap. 3]. Data streams repræsenterer input data, der modtages ved høj hastighed. Ved høj hastighed menes det, at streamen belaster det brugte kommunikations og computer udstyr i en sådan grad, at det er svært at:

1. Sende inputtet i sin helhed til modtager programmet.
2. Foretage avancerede realtids beregninger på større dele af inputtet i samme hastighed som det modtages.
3. Lagre inputtet enten midlertidigt eller permanent.

Forkortet kaldes dette TCS (transmit, compute og store). Traditionelt har TCS ikke været noget, man har tænkt på i computer verdenen. Da man oprindeligt begyndte at forbinde computere, var det for at kunne overføre data som f.eks. e-mails, filer til processering osv. Størrelsen af sådanne data var endelig, så begrænsninger i f.eks. transmissions hastigheden havde kun den betydning, at overførslerne tog længere tid. Var man ikke tilfreds med den oplevede performance, kunne det løses ved tilføjelse af ekstra resourcer som f.eks. hurtigere forbindelser til transmissionerne, hurtigere processorer til beregningerne og mere lager plads til at gemme dataen på.

Siden den gang er det dog specielt to udviklinger, der har gjort, at man bør se data streams i form af TCS:

- **Brugen af automatiske data genererende enheder.** I dagens digitaliserede verden findes der et utal af ting, der løbende registrerer et eller andet, og sender dette videre som data til en computer. Det kan både være af den mindre seriøse slags som f.eks. webcams placeret på Rådhuspladsen hvis billeder uploades til en hjemmeside, eller af den mere seriøse som satellitter der løbende tager atmosfæriske billeder til brug for meteorologer.

  Fælles for disse enheder er, at de repræsenterer en potentielt uendelig stor data stream, så længe de er i brug. Muligvis er hastigheden man modtager dataen med ikke større, end at man løbende kan tilføje ny lagerplads, men det ændrer ikke ved, at man er nødt til at gøre noget aktivt, hvis ikke man

skal løbe tør for plads på et eller andet tidspunkt. Om den modtagne data så kræver realtime processering, og om dette er muligt er så ekstra ting, der kan give problemer, men lager problemet er i hvert fald noget, der kræver en løsning.

- **Behovet for realtids analysering og overvågning af eksisterende streams.** Det i mine øjne bedste eksempel på dette er behovet for overvågning af netforbindelser - hvad enten vi taler om store backbone forbindelser på Internettet eller små ADSL opkoblinger. Fælles for disse er, at ser man på aktiviteten på en given forbindelse, så er det en data stream, der normalt er sammensat af mange andre mindre streams. Disse streams vil have transmissions protokollerne tilfælles (f.eks. TCP/IP), men indholdet i de enkelte streams kan være vidt forskellige. I princippet kan alle de enkelte streams repræsentere endelige datamængder i form af f.eks. filoverførsler, men samlet set repræsenterer aktiviteten på en given netforbindelse en - over tid - uendelig stor data stream, der aldrig stopper.

  Uanset typen af overvågning der skal finde sted, og uanset om man ved præcist hvilke typer af data der observeres på den givne forbindelse, så vil man være tvunget til at tænke i termer af TCS. Som ved de automatisk genererende enheder er det fordi man som den observerende part, som minimum skal afgøre, hvad der skal ske med en potentielt uendelig stor datamængde.

Det er behovet for overvågning og analysering af eksisterende streams, der danner grundlaget for sampling og dermed for dette speciale. I denne sammenhæng giver det derfor mening at tænke på data streams i TCS termer.

2.2 Modeller for data streams

I [3, kap. 4] defineres formelt tre modeller for data streams, som forfatteren S. Muthukrishnan mener, dækker over de mest populære modeller, der benyttes inden for feltet i dag. Da jeg ikke har kunne finde noget tegn på, at dette ikke skulle være rigtigt, og da jeg synes, hans definitioner er de bedst skrevne, er det dem jeg gengiver her.

En data stream kan ses som en liste af elementer $a_0, a_1, ...$ der ankommer sekventielt, et efter et. Disse elementer beskriver samlet et underliggende signal A, der kan ses som en 1-dimensional funktion $A : [1 \cdots N] \rightarrow R$. Dét, der adskiller de tre modeller, er, hvordan de enkelte a_i beskriver A.

- **Time Series modellen**. I denne model er $a_i = A[i]$, og de ankommer i streamen i rækkefølgen angivet ved i. Dvs. at $A[j]$ kun bliver påvirket af a_i når $i = j$ og ikke af elementer hverken før eller senere i streamen.

  Et eksempel hvor denne model er brugbar, er hvis man hvert 5 minut modtager de foregående 5 minutters belastning på en router. Informationen om det pågældende tidsintervals belastning ændres ikke, når først den er registret. Hver enkelt modtaget værdi kan derfor ses som værende atomar i forhold til de foregående og efterfølgende værdier, på trods af at de samlet set f.eks. giver informationer om brugsmønstrer af routeren.

- **Cash Register modellen**. I denne model ses et givet a_i som værende en opdatering til en vilkårlig $A[j]$. Dvs. hver enkelt $A[j]$ repræsenterer noget, der holdes styr på, og opdateringerne af disse kan ske i tilfældig rækkefølge. De enkelte a_i kan derfor ses som $a_i = (j, I_i)$ hvor $I_i \geq 0$, som så bevirker opdateringerne $A_i[j] = A_{i-1}[j] + I_i$. Hver enkelt a_i indeholder altså information

om både hvilken indgang i A, der skal opdateres, og hvad opdateringen består af.

Navnet på denne model kommer af, at man kan tænke på denne form for opdateringer som et kasseapparat med x forskellige rum, ét for hver type af mønt der findes. Efterhånden som kasseapparatet bruges, vil der løbende blive tilføjet mønter til de forskellige rum i vilkårlig rækkefølge. Modellen illustrerer dette. I netværks sammenhæng er et eksempel, at man f.eks. overvåger forbindelser fra vilkårlige IPer til en given server. Serveren modtager datapakkerne fra afsenderne i tilfældig rækkefølge (set fra serverens synsvinkel), og opdaterer så sin log med f.eks. information om antallet af modtagne pakker fra hver enkelt IP.

- **Turnstile modellen**. Denne sidste model minder meget om Cash Register modellen med den forskel, at ved opdateringerne $A_i[j] = A_{i-1}[j] + I_i$ kan I_i dække over både positive og negative værdier. Dvs. der kan være tale om, at der både tilføjes og fjernes. Umiddelbart kan det være svært at se, hvad forskellen til Cash Register modellen betyder, men det kan ses på valget af problem område. Cash Register modellen giver mening ved f.eks. overvågningen af datapakkerne på en netværksforbindelse, fordi der i sagens natur altid kommer nye pakker. Der kommer aldrig en negativ pakke i den forstand, at den negerer en tidligere modtaget pakke. Der kan komme en pakke, hvis indhold modvirker indflydelsen fra en tidligere modtaget pakke, men hvis man f.eks. tæller antallet af modtagne pakker ud fra afsendernes IPer, så kan man aldrig modtage en pakke, der bevirker, at tælleren skal tælle ned i stedet for op.

  I nogle tilfælde er der dog brug for negering af ting, f.eks. hvis det, modellen skal beskrive, er en database. I så fald skal man (oftest) kunne slette data igen fra databasen. I sådanne tilfælde giver det derfor mening at snakke om negative opdateringen.

  Inden for Turnstile modellen er der desuden definitionen *strict* Turnstile, som dækker over de tilfælde, hvor $A_i[j] \geq 0$ for alle i på ethvert tidspunkt. Denne definition kommer af, at ofte i det der modelleres, giver det ikke mening at snakke om negative resultater. Hvis f.eks. modellen beskriver antallet af rækker med samme værdi i en given databases tabeller, så vil sletning af rækker aldrig kunne bringe antallet ned på mindre end 0.

Af de tre modeller er Turnstile den mest generelle, derefter kommer Cash Register og til sidst Time Series. Dvs. i princippet ønsker man at designe algoritmer ud fra Turnstile modellen, da de så også kan bruges på de data streams, der beskrives med de to andre modeller. Som altid ved computere kan for voldsom generalisering af et givent problem dog betyde, at det kan blive meget svært eller helt umuligt at lave i praksis. De mindre generelle modeller kan derfor ofte være bedre til at beskrive et givent problem.

2.3 Hvorfor logge data streams?

Et vigtigt spørgsmål i et speciale som dette er, hvorfor det i det hele taget er interessant at foretage logning i netværks sammenhæng. Både fordi det er vigtigt at vide, om det er værd at bruge penge på det, men endnu mere fordi man er nødt til at vide, hvad der er man gerne vil opnå med logning. Det er først, når vi har identificeret, hvad det er vi gerne vil bruge det loggede materiale til, at vi er i stand til at designe et system, der logger det rigtige på en brugbar måde. Uden et klart

mål kan lang tids logning være værdiløs, hvis det viser sig, at første gang man skal bruge det loggede data, at det ikke indeholder svarene, på det man leder efter.

Jeg har ikke fundet noget generelt survey over, hvad det er folk gerne vil opnå med logning, men ved at kigge på publiceret materiale og eksisterende systemer kan jeg se, at de normalt falder inden for tre kategorier:

- **Vedligeholdelse**. Dette dækker over at finde og forebygge fejl/mangler i et fysisk netværk. Fra en administrators synsvinkel begrænser dette sig oftest til at overvåge belastningen af de enkelte knudepunkter (routere) ved hjælp af f.eks. SNMP[1] og belastningen af kritiske netværkstjenester (DNS, web, mail etc). Systemer i denne kategori kendetegnes ved, at de er meget simple, og størrelsen af den loggede data er begrænset. Målet er at holde øje med usædvanlig trafik på netværket. Falder belastningen f.eks. kraftigt på en normalt meget brugt forbindelse, kan det nemlig være tegn på en fejl et sted i netværket.

  Systemer i denne kategori har været brugt længe og er i større eller mindre udstrækning indbygget som standard i backbone produkter til netværk i dag. Udviklingen af disse systemer er derfor mest betinget af producenternes ideer til nye funktioner, der - udover den forhåbentlig praktiske værdi - kan distancere dem fra konkurrenternes produkter.

- **Sikkerhed**. Ved logning med sikkerhed for øje er målet at opdage og/eller sikre beviser om misbrug i et givent system. Misbrug falder normalt ind under to typer af tilfælde: Enten er der tale om en eller flere personer, der forsøger at gøre noget, de ikke burde gøre, eller også er der tale om et program (f.eks. virus), der ligeledes forsøger at gøre noget uønsket. Forskellen mellem de to er, om der er tale om et specifikt angreb på det overvågede system, eller - som i tilfældet med vira - om angrebet er mere tilfældigt, ved at netværket blot rammes sammen med mange andre.

  Man kan diskutere, om beskyttelsen mod vira o.lign. programmer hører under sikkerhed eller blot normal vedligeholdelse, men jeg mener, det bør tages med i sikkerhed, da det i sidste ende drejer sig om programmer skrevet af personer, der ønsker at opnå en eller anden form for uønsket handling i systemet. Disse programmer opstår ikke af sig selv, og det er, hvad der adskiller dem, fra hvad man ellers har at gøre med ved vedligeholdelsen af et system.

  Logningen i sikkerheds øjemed består ofte af to forskellige indfaldsvinkler til, hvad der registreres. Det ene, man prøver at holde øje med, er usædvanlig adfærd på netværket og evt. reagere automatisk på denne adfærd. F.eks. er det meget almindeligt, at systemer, der kræver at folk logger på med brugernavn og kode, automatisk spærrer et givent brugernavn, hvis nogen indtaster koden forkert x gange. Målet i disse tilfælde er derfor at finde ud af, hvordan normal adfærd i systemet ser ud, og så reagere på alt andet.

  Det andet, man normalt ønsker at gøre, er udførligt at registrere, hvad der sker af normal adfærd i systemet. Problemet, i f.eks. systemer hvor brugere kan logge på, er i sidste ende, at man aldrig med 100% sikkerhed kan vide, om det virkelig er den rigtige bruger, der er logget på. Brugeren kan f.eks. have givet sit logon til en anden person af - i brugerens øjne - harmløse årsager. Da handlinger foretaget i systemet ved denne type misbrug normalt ikke kan spottes som unormal adfærd, er det derfor vigtigt, at logge hvad der foretages. Ellers er det næsten umuligt at finde ud af misbrugets omfang, hvis det senere opdages.

[1] http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/snmp.htm

Behovet for sikkerhedsmæssig logning er eksploderet de senere år i takt med specielt Internettets udbredelse. Der findes derfor allerede et hav af produkter, der fokuserer på at analysere logs genereret af eksisterende firewalls, servere etc. eller på selv at observere trafikken på netværket mere direkte.

- **Analyse**. Ved analyse mener jeg logning af data, der løbende bliver genereret af et eller flere systemer, og hvor det er behandlingen af denne data, der er det primære fokus. F.eks. ved modtagelse af meteorologiske observationer fra satellitter er der tale om en stream af data, som netop genereres til efterfølgende analysering. Dvs. der er normalt tale om data, der repræsenterer observationer af noget, vi ikke kontrollerer, og som analyseringen efterfølgende skal forsøge at give en forklaring på.

  Grænsen mellem denne kategori og de to andre er lidt flydende. Det er jo netop gennem analyseringen af f.eks. netværkstrafikken på en given forbindelse, at vi lærer, hvad det er, vi skal holde øje med i vedligeholdelses og sikkerheds øjemed. At analyse er en kategori for sig selv skyldes dog, at jeg mener, der er mange tilfælde, hvor målet for logningen og den efterfølgende analysering ikke falder under de to andre kategorier. Ofte er der nemlig tale om, at analyseringen skal bidrage til, at man kan forudsige fremtidig adfærd og dermed forberede sig. F.eks. bruger man meget i styringen af elevatorer i travle bygninger, at elevatorerne - når de er tomme - placerer sig på den etage, hvor det er størst chance for, der næste gang bliver brug for dem. Dette gæt baseres på analyseringen af tidligere brug af elevatorerne.

  Analysering af observeret data har selvfølgelig fundet sted i mange mange år, men det er først med computerens tilkomst og de senere års massive indførelse af IT overalt at brugen virkelig er eksploderet. Hvor meget det så hjælper, at man realtime analyserer og forudsiger, hvad der vil ske, er et felt i sig selv og uden for dette speciale.

Dette er de tre hovedgrunde, jeg har identificeret, for hvorfor man ønsker at foretage logning. Som nævnt i starten af dette afsnit er den begrænsende faktor dog altid, hvor omfattende den loggede data er, når den skal bruges. Et fælles punkt for alle tre kategorier er derfor, at de ofte benytter de samme grundliggende logs til at arbejde ud fra. Har man flere programmer, der foretager den samme logning inden den efterfølgende brug, er dette spild af resourcer, da det loggede gerne skulle beskrive samme data stream.

2.4 Algoritmisk sampling

Behovet for effektiv logning og behandling af data streams har længe være stort, og vil sandsynligvis kun vokse med tiden. Som nævnt i afsnit 2.1 er der tale om data, som modtages ved "høj hastighed", og dermed sætter begrænsninger for, hvad der kan gøres ved dataen. For at imødegå dette problem har man altid benyttet sig af sampling, men i de senere år er nye ideer til, hvordan det kan gøres, blevet udviklet.

Håbet med sampling er, at man kan udvikle måder, hvorpå man kan løse et givent problem, selvom man ikke har adgang til den fulde oprindelige data stream. Hvis man f.eks. ønsker at overvåge belastningen på en given forbindelse, kan man checke størrelsen på hver eneste datapakke, der passerer forbi. Man kan dog også vælge kun at checke hver 10. pakke og så bruge disse til at give en estimering af belastningen. Denne estimering er sandsynligvis ikke helt præcis, men hvis det blot er præcist nok, i den sammenhæng resultatet skal bruges, er det alligevel anvendeligt. Det kan endda være eneste mulighed for rent faktisk at løse det givne problem. I eksemplet med målingen af belastningen på en forbindelse kan det praktiske problem

være, at datamængden, der flyder gennem forbindelsen, er så stor, at det tilsluttede måleudstyr kan have problemer med at følge med. Eneste mulighed i sådanne tilfælde er derfor at sample og efterfølgende estimere.

Der er altid en pris at betale ved sampling. Prisen, der betales, afhænger af, hvordan man vælger at sample, og det er her de mange algoritmer, der allerede er udtænkt til sampling, kommer ind i billedet (nogle af disse nævnes i kapitel 3). Hvad enten man ignorerer en vis mængde af datapakkerne, man udtænker smarte datastrukturer der skitserer de modtagne pakker eller finder på noget helt tredje, så er prisen, at der går information tabt, i forhold til hvis man kunne bruge den fuldstændige data stream direkte. Hvis ikke der går noget tabt, så er der ikke tale om sampling. Bevarer man den oprindelige stream fuldstændig intakt, er der i stedet tale om lossless lagring af den modtagne data.

At sampling er interessant i mange sammenhænge, på trods af at der betales en pris, skyldes, at man ofte kun er interesseret i at observere specifikke detaljer i den givne data stream. I eksemplet med overvågningen af belastningen er selve det at checke størrelsen på hver enkelt data pakke, der passerer forbi en sampling af streamen. Ved sådan sampling får man den præcise størrelse på belastningen, og er dét, det eneste man er interesseret i, så er prisen, i form af at ingen andre information om pakkerne gemmes, acceptabel.

En form for sampling, som har været brugt i mange år, er registreringen af forudbestemte parametre. Ser man f.eks. på en Apache webservers måde at logge hver enkelt forespørgsel på, kan det være på formen:

```
xxx.xxx.xxx.xxx - - [10/Apr/2006:06:06:14 +0200] "GET /robots.txt HTTP/1.0"
    404 208 "-" "Gigabot/2.0/gigablast.com/spider.html"
```

Serveren foretager i princippet en sampling af de indkomne forespørgsler i form af, at den kun registrerer specifikke oplysninger såsom afsenderens IP, dato osv. Den konkrete data pakke, der blev modtaget med forespørgslen, bliver ikke gemt. Hvilke værdier, der så er mulighed for at gemme, er bestemt af designeren af log systemet.

Det grundliggende ved, at nogen skal vælge, hvad der skal registreres, har ikke ændret sig. Hvad man derimod er begyndt at sætte fokus på, er hvordan det registrerede, bliver behandlet og gemt. I traditionelle log systemer gemmes det registrerede på samme måde som vist ovenfor. Det lagres i f.eks. en tekst fil, som kan læses - mere eller mindre - direkte af et menneske. En nyere indfaldsvinkel til dette er i stedet at gemme det registrerede i datastrukturer, som ved hjælp af algoritmer baseret på matematiske ideer prøver at indeholde samme informationer som en traditionel log med et mindre pladsforbrug. Det mindre pladsforbrug kan medføre, at der også er data i forhold til den traditionelle log, der går tabt, men besparelserne i plads kan i nogle tilfælde opveje et sådant tab.

Det er en sådan datastruktur frembragt ved algoritmisk sampling, der beskrives i [1].

Kapitel 3

Eksempler på brug af sampling

Inden jeg går i gang med at beskrive artiklen, der danner grundlag for min implementation og tilhørende resultater, vil jeg i dette kapitel først præsentere nogle forskellige måder, hvorpå sampling kan bruges i netværks sammenhæng. Det være sig både i form af teoretiske forslag og løsninger der er implementeret i praksis. Fælles for disse ideer er, at de er blevet publiceret i en eller anden form, og at jeg har stødt på dem i forbindelse med mit arbejde med specialet.

Artiklerne jeg har valgt at medtage her, er dem jeg har fundet interessante, og som jeg synes repræsenterer spændende indfaldsvinkler til brugen af sampling. Jeg gennemgår artiklerne i overordnede træk, og jeg har ikke forsøgt at verificere rigtigheden af teorien, der ligger til grund for artiklerne. Målet er blot at give et hurtigt indblik i forskellige former for sampling, ikke at gå for meget i detaljer med dem. Det betyder, at specielt måden hvorpå forfatterne af de valgte artikler løser problemerne, de beskriver, kun beskreves i overordnede træk.

Der er selvfølgelig mange artikler vedrørende sampling jeg ikke nævner her i specialet, både artikler der præsenterer anderledes løsninger til problemerne behandlet i de valgte artikler, og artikler der handler om helt andre måder at udnytte sampling på. Det er ikke mit mål at skabe et fyldestgørende overblik over samplingsområdet i dette speciale, men derimod at fokusere på datastrukturen beskrevet i [1] som resten af specialets kapitler handler om. Artiklerne jeg præsenterer i dette kapitel, skal derfor ikke ses som mit bud på de vigtigste inden for feltet, men som en introduktion til forskellige måder hvorpå brugen af sampling kan udnyttes.

3.1 To typer af artikler

Praktisk talt alle artiklerne, jeg i forbindelse med specialet har kigget på, der har vedrørt brugen af sampling i netværks sammenhæng, har kunnet opdeles i to kategorier:

- **De teoretiske.** Artikler hvori der præsenteres nye ideer til, hvordan man rent teoretisk kan udføre sampling på data streams. Der bliver primært fokuseret på teorien bag forslagene i form af pladsforbrug, beviser for korrektheden af teorien etc. Normalt rummer disse artikler også forslag til og evt. resultater med praktisk brug af ideerne i en konkret sammenhæng, men det primære mål for artiklerne er at informere om de nye teoretiske resultater.

- **De praktiske.** Artikler hvori der primært fokuseres på, hvordan man har benytte sampling i forbindelse med data streams i netværk til at løse allerede

eksisterende problemer. Som jeg har skrevet om i 2, er en stor del af problemet ved f.eks. logning, rent faktisk at vælge hvad der skal logges. Artikler i denne kategori handler derfor ikke så meget om matematisk teori ved sampling, men mere om hvordan man kan bruge sampling i praksis, og hvad man bør sample.

Oprindeligt troede jeg, at de fleste artikler man kunne finde, ville tilhøre begge disse kategorier. Mest af alt håbede jeg på, at finde artikler der omhandlede konkrete implementeringer og brug af sampling med udgangspunkt i eksisterende netværk. Jeg synes dog, at det har vist sig, at der er en stor overvægt af publicerede artikler, tilhørende den første kategori og en del mindre tilhørende den anden. Jeg har samtidigt ikke fundet nogen artikler, jeg reelt synes, tilhører begge kategorier. Mange af dem jeg synes tilhører første kategori, anfører godt nok motiverende problemer taget fra den "virkelige"verden, men der er ikke nogen af dem jeg har set på, som beskriver om de så i praksis også har løst problemerne for de administratorer, der sidder med dem.

Når man får tænkt over det, er det nok egentlig heller ikke så underligt, at der er denne opdeling i artiklerne. Hvor det i de teoretiske artikler er ting så som pladsforbrug, CPU brug, præcision etc. der er det vigtige for forfatterne, er det helt andre ting, der optager folkene bag de praktiske artikler. I disse er det derimod mere overordnede ting, som er vigtige. F.eks. hvor nemt det er at opsætte samplings systemet i netværk og efterfølgende kunne vedligeholde det uden at påvirke driften af netværket. Ud fra artiklerne af dem der har opsat sampling systemer til brug i praksis, virker det altså ikke som om, det er så vigtigt, hvilke algoritmer og datastrukturer der bruges i systemerne, men mere hvordan man designer hele systemet.

I de følgende afsnit beskriver jeg artikler tilhørende begge kategorier, da jeg mener, at både den teoretiske og den praktiske side er vigtig for brugen af sampling. At jeg nævner opdelingen i de to kategorier, skyldes dog, at man ved omtalen af de enkelte artikler skal forberede sig på, at det altså enten kun handler om teoretiske ideer eller kun om praktiske design forslag.

3.2 Heavy hitters sampling

Heavy hitters er betegnelsen for de algoritmer indenfor sampling, der fokuserer på at identificere de elementer i den observerede data stream, der optræder væsentlig flere gange end gennemsnittet. Artiklen jeg kigger på i den sammenhæng er "Approximate Frequency Counts over Data Streams"af G. S. Manku og R. Motwani ([4]).

3.2.1 Problem

Denne artikel handler om de problemer, man står over for i forbindelse med potentielt uendelige data streams, hvis man ønsker at opretholde en eller anden form for statistisk information om de observerede data. I artiklens introduktion ([4, afsnit 1.]) nævnes de to primære problemer, man står over for ved sådanne streams: Nemlig det store - evt. uendelige - pladsforbrug krævet for at kunne lagre hele streamen, og problemet man står over for, hvis man ønsker at udføre en forespørgsel på dataen, uden at det skal tage alt for lang tid at få svar. Disse to problemer nævnes, som den motiverende årsag til, at man bør designe *summary data structures*, hvis formål netop er at være relativt (i forhold til den fulde data stream) små mht. pladsforbrug, og som sætter brugere i stand til at foretage forespørgsler af en eller anden art på dataen.

Hvad det er for en type af forespørgsler, man ønsker at kunne foretage, er selvfølgelig afgørende for, hvordan man bør designe sådanne datastrukturer. I [4]

nævner forfatterne derfor, at man ofte har brug for oplysninger om frekvensen af elementerne i en data stream. Hvilke af elementerne, der så er interessante, er fokuset for artiklen. Specielt nævnes *Iceberg Queries* ([4, afsnit 2.1]) som værende en god repræsentant for typen af forespørgsler, man ønsker at understøtte. Iceberg Queries bliver præsenteret i [5], og er en type af forespørgsler, hvor man spørger efter de elementer, der forekommer med en frekvens større end et bruger defineret niveau. Et eksempel på en sådan forespørgsel er vist i [4, afsnit 2.1] i form af en konstrueret SQL query:

```
SELECT c1, c2, ABABA , ck, COUNT(rest)
FROM R
GROUP BY c1, c2, ABABA , ck
HAVING COUNT(rest) >= tau
```

Parameteren *tau* angiver i denne forespørgsel niveauet fastsat af brugeren.

Der er i artiklen flere motiverende eksempler, men af størst relevans for mit speciale er de netværks relaterede anført i [4, afsnit 2.4]. Her nævner forfatterne, at denne type af forespørgsler kan være meget interessante i forbindelse med overvågning af trafikken på f.eks. backbone linier på Internettet. Man kan se på trafikken i sådanne linier som bestående af flows defineret ved sekvenser af data pakker med samme afsender og modtager adresser. I så fald kan det være interessant, at identificere flows der f.eks. repræsenterer mere end 1% af den samlede trafik, der passerer linien. Både i forhold til almindelig vedligeholdelse og analysering af liniens brug men også i mere kritiske sammenhænge såsom identificeringen af denial-of-service angreb.

På baggrund af disse motiverende eksempler fastsætter forfatterne i [4, afsnit 3] målet for artiklen, nemlig at konstruere en datastruktur, som accepterer to af brugeren definerede parametre: $s \in (0, 1)$ til angivelse af det ønskede niveau og $\epsilon \in (0, 1)$ hvor $\epsilon \ll s$ til angivelse af den tilladte fejl margin. Lader man N angive antallet af elementer, man har observeret ind til videre i den givne data stream, er målet så, at man på ethvert tidspunkt skal kunne bede datastrukturen om at returnere en liste af elementer sammen med deres estimerede frekvens i den observerede data stream. Dette returnerede svar skal opfylde følgende krav:

- Alle elementer hvis virkelige (dvs. ikke estimerede) frekvens overstiger sN returneres.
- Intet element hvis virkelige frekvens er mindre end $(s - \epsilon)N$ returneres.
- De estimerede værdier for frekvenserne er mindre end de virkelige frekvenser med højst ϵN.

Opfylder en datastruktur disse krav, er det klart, at man så vil kunne besvare forespørgslerne i de nævnte eksempler. Forfatterne ser det derfor som deres mål, at lave en datastruktur, der opfylder kravene ved brug af mindst mulig plads.

3.2.2 Løsning

Der præsenteres i [4] to forskellige forslag til algoritmer, der opretholder datastrukturer opfyldende de ønskede krav. Både en non-deterministisk (*Sticky Sampling* [4, afsnit 4.1]) og en deterministisk (*Lossy Counting* [4, afsnit 4.2]). Forskellen på de to løsninger er primært, at *Sticky Sampling* har en ekstra bruger defineret parameter δ, der angiver risikoen for, at datastrukturen opbygget af *Sticky Sampling* helt fejler. Resultaterne af brugen af de to løsninger opsummeres i artiklens sætninger 4.1 og 4.2 til:

- *Sticky Sampling* opfylder de angivne krav med en sandsynlighed på mindst $1 - \delta$ ved brug af maks $\frac{2}{\epsilon} \log(s^{-1}\delta^{-1})$ elementer fra den observerede data stream.

- *Lossy Counting* opfylder de angivne krav ved brug af maks $\frac{1}{\epsilon}\log(\epsilon N)$ elementer fra den observerede data stream.

Umiddelbart ligner det ud fra disse to opsummeringer, at *Sticky Sampling* bruger en konstant mængde plads, og at *Lossy Counting* vokser logaritmisk med længden N af data streamen. De angivne pladsforbrug beskriver dog worst case scenarier, og i [4, afsnit 4.3] argumenterer forfatterne derfor, at man ved de fleste former for data streams, ikke burde havne i den form for pladsforbrug.

Resten af [4] handler om, hvordan de to foreslåede løsninger kan implementeres, og hvilke resultater forfatterne opnåede med disse. Alt dette vil jeg ikke komme ind på, men jeg har selv en enkelt kommentar til artiklen: De angivne parametre givet af brugeren i både *Sticky Sampling* og *Lossy Counting* skal fastsættes inden algoritmerne begynder at analysere den givne data stream, og kan ikke ændres medmindre algoritmerne initialiseres på ny. Det er i hvert fald sådan jeg forstår løsningerne angivet i [4], og jeg kan ikke se nogen steder forfatterne kommenterer på det.

Umiddelbart har min observation ikke meget at gøre med de teoretiske resultater i artiklen, men jeg synes det er en vigtig ting at være opmærksom på, da det altså betyder, at man skal definere parametrene på forespørgslen, når analysen af streamen startes. I situationer hvor den observerede data streams trafik ikke registreres på andre måder, vil det betyde, at man ikke kan gå tilbage og udføre en ny forespørgsel med andre parametre på den allerede observerede del af data streamen. Den mulighed vil være gået tabt.

3.3 Reservoir sampling

Reservoir sampling er en anden samplings form, der blev præsenteret allerede tilbage i 1985 i artiklen "Random sampling with a reservoir"af J. S. Vitter ([6]). Som anført i motivationen for artikelen ([6, afsnit 1]) var det ikke netværk, der blev tænkt på, da artiklen blev skrevet. Det var derimod hvordan man kunne foretage en tilfældig sampling af et ukendt antal elementer på f.eks. et magnet bånd af ukendt længde. Artiklen passer derfor til tiden inden Internettet rigtigt begyndte at blive udbredt. At en data stream på en netforbindelse så svarer til dataen på et bånd af ukendt længde, betyder bare, at artiklen og dermed reservoir sampling stadig er relevant.

3.3.1 Problem

I [6, afsnit 1] formaliseres problemet, nemlig at man står med en mængde af samlet set N elementer, hvorfra man ønsker at udtage et tilfældigt sample i form af n elementer. Man kender ikke størrelsen af N, og man er ikke i stand til på nogen nem måde at bestemme størrelsen af N, inden samplingen skal begynde. Der sættes derfor i [6] den begrænsning, at algoritmen der udvælger de n tilfældige elementer, skal gøre det uden noget kendskab til N, og desuden skal de N elementer præsenteres til algoritmen én efter én uden gentagelser. Dvs. præcist hvad vi i dag vil kalde en data stream.

Hele kernen i dette problem er at sikre, at de n udvalgte elementer også er tilfældigt udvalgt over hele den - ind til da - sete stream uden at have behov for at undersøge tidligere sete elementer.

3.3.2 Løsning

Problemet der ønskes at løse, er altså nemt at overskue. I [6, afsnit 2] går forfatteren derfor direkte videre til grundideen for løsningen til problemet. Denne ide er nemlig, at efterhånden som den valgte data stream observeres opbygges, der en

midlertidig buffer af tilfældige elementer. Denne buffer skal have en størrelse $\geq n$ så der herfra kan udvælges de n tilfældige elementer, når man ikke længere ønsker at observere data streamen. Det er ideen med denne midlertidige buffer, der afføder navnet "reservoir sampling", og er fælles betegnelsen for alle samplings teknikker, der bruger sådanne buffere til at foretage den endelige sampling ud fra.

I [6] beskriver forfatteren hvordan selve samplings algoritmen kan laves og teorien bag hans forslag. Dvs. hvor stor den pågældende buffer skal være, hvornår elementerne i den skal erstattes osv. Jeg vil dog ikke komme ind på teorien her, da jeg ikke føler, den er nødvendig for forståelsen af den grundliggende ide bag reservoir sampling, og for at man ved hvad der er tale om, hvis man møder betegnelsen i andre artikler.

3.4 Subset-sum sampling

Subset-sum sampling repræsenterer en tredje slags sampling, hvor målet er at estimere så nøjagtigt som muligt størrelsen af hver enkelt flow (f.eks. fra IP til IP) gennem en observeret forbindelse. Ideen er beskrevet i artiklen "Learn More, Sample Less: Control of Volume and Variance in Network Measurement"af N. Duffield, C. Lund og M. Thorup ([7]).

3.4.1 Problem

[7] lægger ud med at argumentere for, at i en lang række af sammenhænge er det interessant at vide, hvem der bruger et netværks kapacitet. I [7, afsnit 1] nævnes både driftsmæssige årsager, i form af check på om nogen bruger en uforholdsmæssig stor del af kapaciteten, og behovet for mere basale check af forbrug hvis brugerne f.eks. betaler for den belastning, de påfører netværket. I sådanne situationer er det nødvendigt at have redskaber til rådighed, der fokusere på at måle sådanne belastninger.

På en mere formel måde anføres det i [7], at man kan se på en mængde af flows $i = 1, 2, \ldots, n$ som alle har tilknyttet en størrelse x_i og en nøgle c_i. Nøglen er egentlig den, der identificerer de enkelte flows, og er derfor noget der er konstrueret ud fra headeren i de data pakker, der observeres på netværket. F.eks. afsender og modtager IPerne kombineret til en 64 bits nøgle.

Målet for [7] er, at de gerne vil kunne estimere størrelsen af flows med nøgler tilhørende en given mængde C, dvs. summen $X(C) = \sum_{i:c_i \in C} x_i$. På denne måde vil forfatterne altså gerne have mulighed for effektivt både at få oplyst estimeringer på enkelte flows belastninger og hele grupper af flows. Dette giver god mening i praktisk brug i netværk, hvor man f.eks. er interesseret i at se den samlede belastning genereret af et subnet af computere.

3.4.2 Løsning

Grunden til at jeg finder denne artikel interessant, og har taget den med, er forfatternes indfaldsvinkel til, hvordan det nævnte mål skal opnås. Aktiviteten på den overvågede forbindelse kan være meget voldsom, og det er derfor ikke sikkert, at udstyret der skal måle belastningen, kan følge med, hvis alle observerede data pakker skal undersøges. Af den grund kommer sampling ind i billedet, og i afsnittet om motiveringen for artiklen ([7, afsnit 1]), kommer forfatterne ind på, at kernen i problemet er, hvordan samplingen egentlig foretages. Simple måder til udvælgelse af pakker, der skal indgå i estimeringen, kunne f.eks. være at vælge 1 ud af hver N pakker, så hver data pakke har sandsynligheden $1/N$ for at blive udvalgt. Det

kunne også være at have en deterministisk udvælgelses form, hvor pakker bliver sendt videre til undersøgelse i faste intervaller (N, $2N$ osv.).

Problemet ved netværks trafik er dog ifølge [7], at der kan være meget stor forskel på størrelsen - og dermed betydningen for estimeringen - af hver enkelt data pakke i netværket. Hvis der f.eks. på et netværk optræder mange små data pakker og nogle enkelte virkelig store, så vil en udvælgelse til sampling, der ignorerer størrelsen risikere, at give estimeringer, der ikke er præcise nok. Fokus for [7] er derfor, hvordan man bør foretage samplingen, så de bedst mulige resultater opnås.

Måden forfatterne løser problemet på, er at udvælge pakkerne de ønsker at sample på baggrund af deres størrelse. Ideen er, at hvis alle pakker over en hvis størrelse - bestemt ud fra en bruger defineret parameter - bliver samplet, og man for de mindre sampler en gang i mellem, så opnår man de bedste resultater. Ideen kommer af, at eftersom det er de store pakker, der har størst indflydelse på estimeringen, giver det også mening, at det er dem der primært bliver samplet.

Dette er selvfølgelig en meget forsimplet måde at beskrive løsningen på, og i resten af [7] går forfatterne da også gennem en hel del teori for, hvordan udvælgelsen kan foregå, og hvorfor det giver gode estimeringer. I [7, afsnit 9] nævnes det desuden, at den beskrevne strategi til sampling fra artiklen ”is currently used to collect NetFlow records collected extensively from a large IP backbone”. De skriver ikke, hvilket netværk der er tale om, og det fremgår heller ikke, om der er tale om en test opsætning, eller om den pågældende ejer af netværket bruger det som et redskab i praksis. Uanset brugen af dette opsatte system så lyder det dog som om, at forfatterne af [7] mener, at brugen af sampling på denne måde ved måling af årsagerne til belastninger i netværk er en success.

3.5 Sampling med wavelets

Den næste artikel er ”Surfing Wavelets on Streams: One-Pass Summaries for Approximate Aggregate Queries”af A. C. Gilbert, Y. Kotidis, S. Muthukrishnan og M. J. Strauss ([8]).

3.5.1 Problem

Udgangspunktet for denne artikel minder om de allerede beskrevne motivationer for de andre artikler. Som jeg har skrevet om i kapitel 2, så er det nok egentlig ikke så underligt med Internettets voldsomme fremmarch de sidste 15 år. Behovet for her-og-nu analysering og overvågning er vokset sideløbende med Internettets udbredelse. I [8, afsnit 1] er det derfor også behovet for hurtige og præcise svar om trafikken på en overvåget data stream, der er motivationen for artiklen. Uanset om overvågningen er baseret på en drifts- eller sikkerhedsmæssig baggrund, så gælder det om at konstruere en datastruktur, der er i stand til at besvare en eller anden form for forespørgsler om aktiviteten på den underliggende stream.

Hvor de forrige artikler har fokuseret på hvilke elementer i den observerede stream, der er interessante, og derfor ønskes udvalgt ved sampling, er denne artikels fokus i stedet rettet mod at bevare en *sketch* af streamen i sin helhed. Dvs. at bevare noget, der minder om den oprindelige data stream, og som følge deraf kan bruges til at genskabe den oprindelige stream med en vis præcision. Er den konstruerede *sketch* tæt nok på, hvordan den oprindelige data stream rent faktisk så ud, begynder denne form for sampling at nærme sig, hvad man ville opnå hvis man i stedet havde komprimeret data streamen - blot med brug af meget mindre plads.

I artiklen fokuseres der meget på telefon numre ([8, afsnit 3]), da forfatternes tests af deres ideer blev brugt på opkalds information fra telefonselskabet AT&Ts netværk ([8, afsnit 6]). Ifølge artiklen består elementerne i denne form for netværk

af informationer om kaldte telefon numre, opkaldenes varighed etc. Målet for [8] bliver aldrig rigtigt defineret udover at der i midten af afsnit 3 nævnes, at de to ting forfatterne finder specielt interessante, er tiden det tager at analysere data streamen, og mængden af plads der er nødvendig for at gemme den skabte *sketch*.

3.5.2 Løsning

Til at løse problemet har forfatterne fundet på at bruge wavelets til at beskrive den observerede trafik i data streamen. Resten af [4] handler om, hvordan dette kan gøres, og hvad de fik ud af det med dataen fra AT&Ts netværk. Brugen af wavelets til lossy komprimering, dvs. komprimering hvor man accepterer, at en del af det oprindelige signal går tabt, bruges meget i forbindelse med komprimeringen af musik, billeder og film. Det er derfor en hel videnskab i selv med teorien for disse, og ikke noget jeg ønsker at komme nærmere ind på her. I grove træk drejer det sig, om at når man har en en række af tal, f.eks.

$$A = [1, 3, 5, 11, 12, 13, 0, 1]$$

så kan man i stedet for at gemme tallene selve lagre en række koefficienter, der ud fra et predefineret omregnings system, næsten kan genskabe de oprindelige tal. I [8, table 1] vises koefficienterne, der skal gemmes ved brugen af Haar[1] waveletten på de ovenfor nævnte tal. Ideen er, at koefficienterne normalt fylder meget mindre at gemme end de oprindelige værdier. Især hvis man begynder at eksperimentere med at fjerne de mindst betydende koefficienter i waveletten.

Som sagt, der findes allerede en hel videnskab vedrørende wavelets i forbindelse med digital lyd og billede, da fordelene her er meget store. F.eks. ved billeder på digital form hvor hvert pixel angives ved en integer, udnyttes det, at når disse tal genskabes ud fra wavelets, kan det menneskelige øje normalt ikke se de små farve forskelle, der opstår ved denne lossy rekonstruktion af de enkelte pixels integers. JPEG[2] standarden er et eksempel på et format, der er baseret på udnyttelsen af wavelets til at komprimere billeder med.

Eftersom data streams i sidste ende altid kan ses som tal af en predefineret størrelse, er denne ide i mine øjne faktisk ret interessant. I [8] nævnes der desuden som en fordel ved brugen af wavelets, at har man ekstra CPU kraft og plads til rådighed, kan man gøre en del ud af, at få de gemte tal til at blive mere præcise.

for at hele dene process giver mening, skal det selvfølgelig stadig være muligt at udføre en eller anden form for forespørgsler på den gemte *sketch*, som kan returnerer brugbare svar. I [8, afsnit 5] omtaler forfatterne derfor i dybere detaljer, hvad de kunne gøre det med opkalds dataen fra AT&T.

3.6 QuickSAND

Denne artikel "QuickSAND: Quick Summary and Analysis of Network Data"er som den forrige af A. C. Gilbert, Y. Kotidis, S. Muthukrishnan og M. J. Strauss ([9]).

3.6.1 Problem

Hvor den forrige artikel [8] handlede om brugen af wavelets, som en mulig samplings metode, er denne artikel af meget mere praktisk karakter. En stor del af fokus for artiklen er nemlig, hvordan man rent faktisk bør foretage opsamlingen af data i et netværk. I [9, afsnit 2] gøres der meget ud af at beskrive, hvordan opsamling af trafik

[1] http://en.wikipedia.org/wiki/Haar_wavelet
[2] http://en.wikipedia.org/wiki/Jpeg

data i øjeblikket foregår i eksisterende netværk. Om hvordan det f.eks. kan være netværkets routere, der opsamler data om trafikken de videresender, om sniffer-computere sat "strategiske"steder hvor de kan observere trafik af særlig interesser osv.

En stor del af problemet ved, at trafik data samles så mange steder, er hvordan man skal analysere dem, og hvor evt. forespørgsler til det lagrede skal behandles. En måde at løse det problem på er at opbygge *Data Warehouses* ([9, afsnit 2.4]), hvortil alle enhederne i netværket kan sender den data, de måtte registrere vedrørende trafikken. Fordelen ved sådanne datalagre er f.eks., at man får mulighed for at lave analyser af den samlede trafik på netværket, kigge på trafikkens udvikling over længere tid etc. Ting man ikke ville have mulighed for på de enkelte enheder alene. Ulempen ved at samle dataen skal mest findes i det problem, at den registrerede data skal fra de enkelte enheder og til det centrale lager. Det optager kapacitet på netværket, man måske ikke har råd til at miste i peak hours, samt det er næsten umuligt at sikre synkroniserede timestamps på dataen fra de forskellige enheder.

Da forfatterne til [9] synes, sådanne centrale *Data Warehouses* er en god ide, er fokus for [9] derfor, hvordan man kan minimere størrelsen af dataen, der skal sendes fra de enkelte enheder til det centrale lager. Dvs. så der bruges mindst muligt af netværkets kapacitet på dette.

3.6.2 Løsning

Der er en tæt sammenhæng mellem [9] og [8]. Det er igen AT&Ts netværk, forfatterne har kigget på, og i det hele taget fokuseres der i [9] på, hvordan *sketches* (som omtalt i [8]) kan bruges til at minimere dataen, der skal sendes til det centrale lager. I [9, afsnit 3] gennemgår forfatterne forskellige former for måder sådanne *s-ketches* kan laves på (inkl. ved brug af wavelets), også måder der ikke involverer brug af sampling. Dvs. hvordan man i de forskellige enheder opbygger arrays af data (f.eks. antal sete pakke fra specifikke subnets), hvorefter *sketches* konstrueres til at beskrive disse arrays.

Det grundliggende budskab for artiklen ([9, afsnit 5]) er dog, at et væsentligt problem ved enhver form for trafik registrering, der involvere flere steder i et større netværk, er, at dataen skal samles et eller andet sted, så den kan analyseres. Det er derfor meget vigtigt, at uanset typen af data man vælger at registrere, så skal det der sendes til det centrale lager fylde så lidt som overhovedet muligt. Omend valget af måden *sketchene* laves på, kan betyde, at information går tabt, så er det stadig at foretrække fremfor at sende de oprindelige og potentielt store arrays rundt på netværket.

I princippet "løser"[9] ikke problemet med, hvad der er bedst at gøre ved opsamlingen af data et central sted, da det i sidste ende afhænger helt af netværket, der skal overvåges. Jeg har dog taget artiklen med her, da den i forlængelse af [8] i mine øjne meget godt illustrerer, at hele problematikken omkring sampling er mere end blot at finde på en smart måde at foretage sampling på fra en enkelt enhed i netværket. Det har også stor betydning, hvis det samplede skal integreres med dataen fra andre enheder i et centralt lager. Det er derfor nødvendigt - som præsenteret i [9] - at tænke over hvordan dataen fra én enhed, kan bruges sammen med dataen fra andre enheder til at foretage samlede analyser.

3.7 Gigascope

Udgangspunktet for dette afsnit er artiklen "The Gigascope Stream Database"af C. D. Cranor, T. Johnson, O. Spatscheck and V. Shkapenyuk ([10]). Som ved artiklen om QuickSAND ([9]) handler denne artikel, om de mere praktiske aspekter

af indførelsen af overvågning i kørende netværk. Det er igen AT&Ts netværk, der danner grobund for artiklen, men det specielle ved Gigascope er, at det rent faktisk er et system, der er sat i drift, og blevet brugt af AT&Ts system administratorer ([10, afsnit 4]).

Jeg nævner [10] som udgangspunkt for afsnittet, da det er den, der giver den overordnede beskrivelse af Gigascope, men den er kun én af flere artikler, som omhandler systemet. F.eks. [11] og [12] er artikler, der handler om og uddyber egenskaber ved Gigascope. Hvis man derfor ønsker at stifte grundigt bekendtskab med Gigascope, er det altså ikke nok kun at kigge på [10].

3.7.1 Problem

Ved de tidligere beskrevne artikler, har jeg adskilt problem og løsning, da disse dele har været klart defineret i de pågældende artikler. Det giver især mening i artiklerne hvor forfatterne ønsker at løse et specifikt teoretisk problem, da der i de tilfælde er et klart mål med resultaterne i artiklerne. Jeg bevarer opdelingen her ved beskrivelsen af [10], men denne gang er det ikke så klart defineret, hvad det er forfatterne ønsker at opnå. Dvs. det overordnede mål er selvfølgelig at designe et redskab til brug ved overvågning af trafikken på netværk ([10, afsnit 1]), men der er ikke noget målbart defineret som mål.

Som ved de andre artikler kommer folkene bag [10] med de samme bevæggrunde for i det hele taget at beskæftige sig med dette felt. Fejlsøgning, driftsstyring, sikkerhed etc. er grunde til, at man ønsker at overvåge netværks trafik effektivt. [10] er dog præget af den meget praktiske indfaldsvinkel (i og med de vil designe et brugbart system), og f.eks. i [10, afsnit 1] nævnes eksisterende værktøjer såsom *tcpdump*, der er brugt i tidligere løsninger.

I [10, afsnit 2] kommer forfatterne ind på en form for krav til systemet. Det er ikke tale om konkrete krav fastsat ved starten af projektet, da det mere er en liste over design valg, de har foretaget undervejs i implementeringen. Punkterne giver dog et vist overblik over, hvad forfatterne egentlig har ønsket at opnå. Valgene er:

- **Minimering af data.** Som også forfatterne til QuickSAND [9] hæftede sig ved, så er det helt afgørende for ethvert distribueret system til overvågning, at kommunikationen, mellem enhederne der foretager overvågningen, er minimal. Tilføjelsen af et overvågnings system skal helst ikke være dét, der bliver årsag til kapacitets problemer. I [10, afsnit 2] anføres det, at det gælder om at smide registreret data, der viser sig overflødig, væk så tidligt som muligt. På den måde forhindres det, at det bliver sendt rundt i netværket, og belaster systemet uden grund.

- **Fleksibilitet.** Det anføres, at det er afgørende for brugbarheden af systemet, at det er en hvis grad af fleksibilitet i det designede system, så det også kan løse så mange typer af problemer som muligt. Dog skal det heller ikke være alt for fleksibelt, da det betyder, at det er besværliggør optimering af systemet, og for den sags skyld også gør det sværere at bruge. Det bedste ville være, hvis forespørgsler til systemet kunne gives på en form svarende til SQL ved databaser, da man her har en hvis grad af fleksibilitet i typen af forespørgsler, men samtidigt har et stringent sprog, der kan optimeres.

- **Real time.** Systemet skal designes så det - så vidt muligt - analyserer hver observeret data pakke, så snart denne ses. Kan systemet ikke følge med trafikken i netværket, risikeres det, at der er data der "går tabt"og ikke bliver analyseret, hvilket potentielt kan have stor negativ indflydelse på forespørgslerne til systemet.

- **Fysiske begrænsninger.** Det anføres, at hardwaren der er til rådighed for overvågningen, kan være placeret steder, hvor der f.eks. ikke er megen plads, og sjældent er besøg af en tekniker ([10, afsnit 2.5-7]). Det sætter begrænsninger for typen af hardware, der er til rådighed for overvågningen i form af CPU kraft og lager plads. At der sjældent er besøg af en tekniker, gør også, at enheden måske skal kører i måneder ad gangen uden opsyn. Det er derfor vigtigt, at softwaren er så stabil som muligt, hvilket normalt nemmest opnås bedst med simple programmer.

  Fysiske begrænsninger dækker dog ikke kun den fysiske placering af enhederne men også begrænsninger som følge af prisen på hardwaren. Hvis enhederne til overvågning skal placeres måske tusinder af steder, har prisen på hver enkelt enhed en ikke uvæsentlig betydning for, hvor attraktivt systemet egentlig er i praksis.

Som det kan ses på de nævnte problemstillinger, spiller de fysiske faktorer en meget stor rolle i, hvordan et sådant system skal designes. Omend måden, den egentlige overvågning skal foregå på (e.g. brug af f.eks. sampling), er vigtig, så kommer [10] ikke ind på disse ting, da det ikke er væsentligt for det store perspektiv.

3.7.2 Løsning

Hvordan forfatterne af [10] har endt med at lave Gigascope systemet, er en større afhandling i sig selv, og ikke noget jeg vil gå meget ind i her. Det vigtigste i deres løsning er, at de har endt op med at designe et query sprog kaldet GSQL, hvoraf navnet viser inspirationen hentet fra SQL. [11] og specielt [12] kommer ind på hvordan GSQL er defineret og bruges. I [12] er der en hel række eksempler på, hvordan sådanne queries kan skrives for at besvare konkrete spørgsmål.

I [12, afsnit 4] bliver der sat fokus på, hvordan Gigascope er designet til at være skalerbart og stabilt. Specielt i [12, afsnit 4.3] nævnes det, hvordan sampling er et nødvendigt redskab, for at kunne lave et system der kan følge med de potentielt enorme data mængder, der passerer gennem de overvågede linier. Der bliver dog ikke gået ind i hvilke typer af sampling, der er bedst at benytte, men i stedet nævnes det, at forskellige typer af sampling kan være anvendelige afhængig af typen af forespørgsler, der ønskes besvares.

Om Gigascope er en success, i den forstand at det er værd at bruge i stedet for eksisterende løsninger, er lidt svært at se ud fra artiklerne. Dét, at der i [10, afsnit 1] nævnes, at i marts 2003 havde forfatterne "seven deployments with many more in negotiations", lyder dog til, at det ser ud til at fungere ok i praksis. Om "seven deployments"så dækker syv forskellige netværk eller syv forskellige enheder i f.eks. AT&Ts netværk, er jeg så ikke helt sikker på.

I alle tilfælde nævnes det i [10, afsnit 4], at Gigascope er blevet brugt med - i forfatternes øjne - success, til at kunne analysere det overvågede netværks driftsmæssige tilstand samt som et redskab i forbindelse med fejlsøgning af netværk. Desuden mener forfatterne at Gigascope kan være et værdifuldt værktøj, for forskere der arbejder med netværk i det hele taget.

Omend artiklerne om Gigascope ikke handler helt så meget om sampling direkte, men mere om hvordan man designer et overvågnings system, der bl.a. benytter sampling, så synes jeg som ved [9] stadig, at artiklerne illustrerer godt en række af de praktiske problemer, man står over for ved overvågning af netværk. Derfor har jeg fundet den værd at tage med her, som et eksempel på praktisk brug af sampling.

Kapitel 4

Den valgte datastruktur

Dette kapitels formål er at gennemgå og forklare den valgte datastruktur fra artiklen [1]. [1] er delt op i fire afsnit, hvoraf afsnit 1 er indledningen, og afsnit 4 er konklusionen. Afsnit 2 beskriver selve datastrukturen, og det er den del af artiklen, jeg primært har arbejdet med.

I afsnit 3 beskriver forfatterne en praktisk anvendelse af datastrukturen, hvor de viser, den kan bruges til at estimere størrelsen af det euclidisk mindst udspændende træ for geometriske punkter givet ved en data stream. De beskriver i afsnittet, hvordan man kan opfatte det problem, så det kan løses af deres datastruktur, og hvorfor det - matematisk - er korrekt, hvad de gør. Denne anvendelse af datastrukturen har intet at gøre med netværk, eller måden jeg bruger den på, og jeg vil derfor ikke omtale indholdet af artiklens afsnit 3 yderligere i dette speciale.

Som nævnt i afsnit 1.4 er dele af [1] blevet revideret i [2]. Eftersom [1] er mit oprindelige udgangspunkt for min implementation, er det derfor den, der danner grundlag for dette kapitel. Jeg vil dog løbende drage sammenligninger med det i [2] anførte, og i det hele taget bruge [2] til at beskrive den underliggende teori for datastrukturen.

4.1 Matematikken i artiklen

Forfatterne af [1] fremsætter i artikelen en række påstande om gyldigheden og pladsforbruget af datastrukturen, når den implementeres efter deres anvisninger. De giver derefter matematiske argumenter/beviser for, at disse påstande er korrekte. Da målet for mit speciale er at se på den praktiske anvendelse af algoritmen - dvs. om den giver mening at bruge i netværks sammenhæng - har jeg ikke forsøgt at verificere, om deres påstande er korrekte. Jeg kommer ind på matematikken i det omfang, jeg mener er nødvendigt for forståelsen af selve datastrukturen og for den efterfølgende implementation, men jeg ser det ikke som mit mål at efterprøve teorien i artiklen.

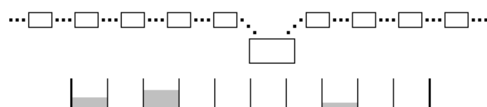
4.2 Det grundliggende problem

Det grundliggende problem, datastrukturen fra [1] forsøger at løse, er følgende:

Figur 4.1: Den initielle vektor over mængden af elementer

Vi har en indekseret mængde af elementer x_i, hvor hvert enkelt element har tilknyttet en værdi. Som udgangspunkt er disse værdier alle 0. Eftersom mængden af elementer er indekseret, kan de repræsenteres af en vektor med én indgang pr. element og med alle indgangenes værdier initieret til 0 (figur 4.1). Det antages yderligere, at værdierne i vektoren ikke kan blive negative.

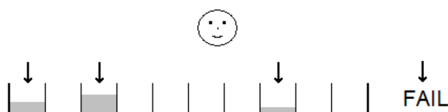
Hvad hver enkelt indgang og tilhørende værdi i vektoren repræsenterer, er i princippet lige gyldigt, men et eksempel kunne være, at hver indgang svarer til en nummereret spand, og den tilhørende værdi er antallet af sten i den pågældende spand. Der er ingen øvre grænse på antallet af sten, der kan være i hver enkelt spand. Derfor de manglende "låg"på indgangene i figur 4.1.



Figur 4.2: Vektoren opdateres via en data stream

Vi har nu en data stream bestående af opdateringer til vektorens indgange. Hver enkelt opdatering angiver derfor en specific indgang i og en værdi a, som x_i skal opdateres med. a må gerne være en negativ opdatering, men indgangen x_i må ikke ende med at blive negativ.

Fortsætter jeg eksemplet fra før, har vi altså nu en lang række personer, som én efter én udvælger en spand, og derefter enten tilføjer eller fjerner sten fra den. Det er selvfølgelig kun muligt at tømme en given spand, så den ikke indeholder flere sten, ikke at få indholdet til at blive negativt. Figur 4.2 viser en situation, hvor den angivne vektor efter et ukendt antal opdateringer har 3 indgange med værdier større end 0, resten af indgangene er 0.

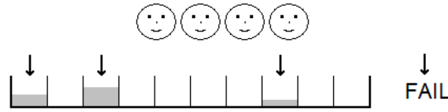


Figur 4.3: Fire mulige svar for observanten. Én af værdierne større end 0 eller FAIL.

Det antages nu, at opdateringen af vektoren vha. data streamen står på i et ukendt tidsrum. Undervejs ser vi hver eneste opdatering, men vi har ikke adgang til selve vektoren. Målet er så på et tilfældigt tidspunkt at være i stand til at oplyse værdien af én af de positive indgange i vektoren, samt hvilken indgang der er tale om. Det skal desuden være tilfældigt, hvilken indgang med positiv værdi der vælges. Er man ikke i stand til at give den oplysning, er svaret FAIL, dog skal sandsynligheden for FAIL være mindre end 1.

Fortsætter jeg igen eksemplet, svarer det til, at en person får bind for øjnene, og derefter får til opgave at holde styr på antallet af sten i alle spandene. Hver gang der ændres på antallet af sten i én af spandene, får han det at vide, men han kan ikke se spandene. På tilfældige tidspunkter vil han så blive bedt om at oplyse antallet af sten i én tilfældig af spandene indeholdende sten og nummeret på spanden. Kan han ikke svare på det, må han sige FAIL. Valget af personen, der får bind for øjnene, skal falde på en, hvis hukommelse altid giver ham en chance for at kunne svare andet end FAIL. Situationen er illustreret af figur 4.3, hvor der altså er 4 mulige svar for personen.

Jeg antager nu, at vi er i stand til at opnå situationen, hvor man altså kan oplyse en tilfældig indgang med positiv værdi i vektoren med en sandsynlighed større end 0.



Figur 4.4: Flere "parallelle"observanter der alle har de 4 mulige svar

Så er det kun tilbage at konstatere, at man så kan reducere risikoen for svaret FAIL vilkårligt ved blot at bruge det nødvendige antal parallelt arbejdende observanter af vektoren. Når vi ønsker en tilfældig indgang i vektoren med en positiv værdi, er det nok, hvis bare én af dem kan give et svar.

I eksemplet svarer det altså bare til, at bruge flere personer som ikke kan se spandene, og som ikke må snakke sammen (figur 4.4).

Målet for datastrukturen fra [1] er at udfylde observantens rolle ved brug af mindst mulig plads. Jævnfør afsnit 2.2 kan man se, at problemet er beskrevet af *strict* Turnstile modellen.

4.3 Problemet formelt

Problemet defineres formelt i [1, afsnit 2] og i [2, afsnit 3]. Det er det samme der står i begge artikler, dog er der små forskelle i den brugte notation. Jeg bruger notationen fra [2], da det gør det nemmere de steder, hvor jeg er tvunget til at gengive teorien i [2] pga. fejlene i [1].

Vi har et indekseret univers U, som kan repræsenteres ved hjælp af vektoren $x = (x_0, \ldots, x_{U-1})$ fra $[M]^U$, hvor M repræsenterer den maksimale værdi, der kan tilknyttes hvert element i U. Vi ønsker nu at konstruere en datastruktur, der ved så minimalt pladsforbrug som muligt tillader visse operationer på vektoren x. Vi ønsker at kunne få et tilfældig sample, blandt de indgange i vektoren der ikke er 0, ud af denne datastruktur. Fordelingen af disse samples bør være næsten uniform.

Ved initialiseringen af datastrukturen antages det, at $x_i = 0$ for alle $i \in U$. Herefter skal datastrukturen kunne tage imod et vilkårligt antal opdateringer i form af Update(i, a), som betyder at der lægges a til x_i. a skal være et tal i $\{-M, \ldots, M\}$ og skal altid medføre, at x_i får en værdi i $\{0, \ldots, M\}$. Dvs. at vi på ethvert tidspunkt har at $0 \leq x_i \leq M$ for $i \in [U]$. Dette krav bliver ikke verificeret af datastrukturen selv. Vi definerer Supp(x) til at være mængden af de indgange i i x, hvor det gælder, at $x_i > 0$, og vi bruger notationen $\|x\|_0 = |\text{Supp}(x)|$ til at angive antallet af disse indgange.

Datastrukturen er parameteriseret ved to tal $\epsilon, \delta > 0$, og har to metoder, der påvirker den underliggende vektor x:

- Update(i, a): Denne metode opdaterer datastrukturen, så $x_i \leftarrow x_i + a$ hvor $i \in [U], a \in \{-M \ldots M\}$. a må som sagt gerne være en negativ værdi, men det antages, at den opdaterede værdi x_i ikke er negativ.
- Sample(): Denne metode returnerer enten et par (i, x_i) for et $i \in [U]$ eller et signal FAIL. Metoden opfylder endvidere to betingelser:
  - Hvis et par (i, x_i) returneres, så er i valgt tilfældigt blandt indgangene med positive værdier Supp(x), sådan at sandsynligheden for at et givent $j \in \text{Supp}(x)$ bliver valgt, er $\Pr[i = j] = \frac{1}{\|x\|_0} \pm \delta$.
  - Sandsynligheden, for at flaget FAIL returneres, er højest δ

Målet er, at designe datastrukturen sådan at $\delta < 1$, så Sample() ikke altid returnerer FAIL. Kan det lade sig gøre, kan sandsynligheden for at der returneres

FAIL gøres vilkårlig lille ved at benytte flere datastrukturer parallelt. Omend det hverken nævnes i [1] eller [2], så medfører brugen af flere datastrkuturer parallelt også, at der potentielt kan samples flere indgange i x. Dette er en egenskab, jeg bruger i forbindelse med min implementation (kapitel 5).

4.4 Gennemgang af datastrukturen

Efter de forrige afsnits beskrivelse af hvad det er, datastrukturen fra [1] prøver at opnå, er tiden kommet til at se, hvordan den er konstrueret. I dette afsnit går jeg igennem dens forskellige dele, så implementationen, jeg præsenterer senere (kapitel 5), forhåbentlig bliver nemmere at gennemskue.

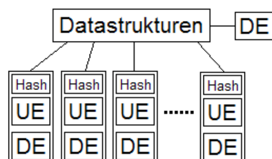
Det er i dette afsnit, at forskellene, mellem [1] og [2] i måden datastrukturen skal laves på, også bliver belyst. [2] indeholder nogle smarte forbedringer, men da min implementation oprindeligt er baseret på [1], er jeg nødt til at beskrive begge artiklers instruktioner.

4.4.1 Overordnet opbygning

Hoved datastrukturen er bygget op omkring en række hash funktioner og to underliggende datastrukturer kaldet Unique Element (UE) og Destinct Elements (DE).

- Hash funktionerne h_j er af typen $h_j : [U] \to [2^j]$. Dvs. hash funktionen h_j hasher indeks i fra vektoren x over i værdier i $\{0, \ldots 2^j - 1\}$. Det er værd at bemærke, at dette betyder, at h_0 sender alle input over i 0. Hash funktionerne antages som udgangspunkt både i [1] og [2] at være helt tilfældige, men begge artikler slækker senere på dette krav.
- UE er en struktur, der bruges til at holde styr på værdierne i vektoren x og dermed sætte datastrukturen i stand til at returnere et sample fra x.
- DE er en struktur, der bruges til at holde styr på antallet af indgange i x med værdi større end 0. Dvs. værdien af $\|x\|_0$.

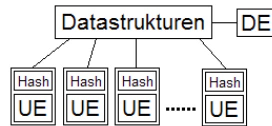
Det er ved hjælp af disse redskaber og de to metoder Update() og Sample(), at datastrukturen bliver i stand til at løse det beskrevne problem. Der er dog forskel på, hvordan henholdsvis [1] og [2] bruger UE og DE.



Figur 4.5: Datastrukturens opbygning i [1]

I den oprindelige artikel [1] er datastrukturen bygget op som vist i figur 4.5. Den benytter én DE struktur til at holde styr på størrelsen af $\|x\|_0$ og så yderligere $\lceil \log U \rceil$ undergrupper med hver deres egen hash funktion, UE og DE. Det er disse undergrupper, der gennem metoderne Update() og Sample() løser problemet (se afsnit 4.4.4 og 4.4.5).

Datastrukturen i den reviderede artikel [2] er næsten identisk med den i [1]. Dog har en smart ændring i UE strukturen (afsnit 4.4.2) gjort det muligt at fjerne DEerne fra undergrupperne. I [2] er datastrukturen derfor bygget op som vist i figur 4.6. Udover at det reducerer datastrukturens samlede pladsforbrug, så løser denne



Figur 4.6: Datastrukturens opbygning i [2]

ændring også et af de store problemer, jeg har haft ved implementeringen. Nemlig at få DE strukturerne til at fungere som forfatterne ønsker det i [1]. Mere om det i afsnit 5.6.5.

4.4.2 Unique Element (UE)

UE er som sagt den struktur, der skal lagre og returnere de faktiske værdier fra vektoren x, der arbejdes på. Det er en simpel struktur, der bliver præsenteret i [1] og efterfølgende forbedret i [2].

Unique Element fra [1]

UE fra [1, afsnit 2] understøtter to metoder Update() og Report() defineret som

- Update(i, a). Er defineret som Update() i afsnit 4.3.
- Report. Hvis $\|x\|_0 = 1$, dvs. hvis vektoren x kun indeholder én indgang $x_i > 0$ så returneres parret (i, x_i). Er $\|x\|_0 \neq 1$ kan ethvert talpar (j, v) returneres. Dette talpar repræsenterer ikke nogen specifik værdi fra vektoren.

For at kunne understøtte de to nævnte metoder holder UE datastrukturen styr på to integers c og C, som begge initialiseres til 0. De to metoder er ved hjælp af disse variabler så givet som:

- Update(i, a): $c = c + a$ og $C = C + (a * i)$
- Report: Konstruer $v = c$ og $i = \frac{C}{v}$. Returner (i, v).

Ifølge [1] er denne datastruktur oplagt korrekt, og den bruger O(log(UM)) bits af lagerplads. At den rent faktisk er korrekt, kommer af, at den definerede Update() sikrer, at

$$c = \sum_{i \in [U]} x_i \qquad \text{og} \qquad C = \sum_{i \in [U]} x_i \cdot i$$

eftersom x er initialiseret til 0. Når $\|x\|_0 = 1$ vil Report() derfor returnere det ene $x_i > 0$, UEen indeholder. Indeholder vektoren flere indgange med værdi større end 0, er det returnerede talpar i princippet ikke helt tilfældigt - det er jo konstrueret ud fra indgangene, der indeholder positive værdier. Det er dog i praksis umuligt at sige noget om specifikke indgange i vektoren på baggrund af talparret.

Unique Element fra [2]

UEen beskrevet i [2, afsnit 3.1] minder meget om UEen fra [1]. Den er dog blevet udvidet, så Report() er lidt smartere. De to metoder Update() og Report() er nu defineret som

- Update(i, a). Er defineret som Update angivet i afsnit 4.3.

- Report: Hvis $\|x\|_0 \neq 1$ returneres FAIL. Hvis $\|x\|_0 = 1$ returneres det unikke par (i, x_i) sådan at $x_i > 0$.

Som det kan ses, er ændringen i Report() i forhold til UEen fra [1], at der kun returneres brugbare værdier eller FAIL. UEen fra [1] returnerer derimod altid et par (i, v), og lader det derfor være op til resten af datastrukturen at holde styr på, at UEen kun indeholder én værdi, og det returnerede derfor giver mening.

Grunden til, at denne ændring kan lade sig gøre i [2], er, at UEen er blevet udvidet til at indeholde tre integers c_0, c_1 og c_2. Update() og Report() implementeres som

- Update(i, a): $c_0 = c_0 + a$, $c_1 = c_1 + (a \cdot i)$ og $c_2 = c_2 + a \cdot i^2$.
- Report: Hvis $c_0 \cdot c_2 - c_1^2 \neq 0$ eller $c_0 = 0$ returneres FAIL. Ellers sættes $i \leftarrow \frac{c_1}{c_0}$ og $x_i \leftarrow c_0$ og der returneres (i, x_i).

c_0 og c_1 svarer altså til c og C i UEen fra [1]. Der er ingen forskel i måden, de opdateres på gennem Update(), og parret (i, x_i) der returneres, beregnes også på samme måde. Forskellen ligger i, at c_2 er tilføjet, så UEen i Report() selv bliver i stand til at afgøre, om $\|x\|_0 = 1$. Ellers returneres der FAIL. Beviset, for at det også er tilfældet, er ([2, afsnit 3.1]): Da vi ved, at $x_i \geq 0$ for alle $i \in [U]$, er det klart, at

$$c_0 = 0 \quad \Leftrightarrow \quad \|x\|_0 = 0$$

eftersom vi ligesom i [1] har, at

$$c_0 = \sum_{i \in [U]} x_i$$

Det afgørende er derfor checket $c_0 \cdot c_2 - c_1^2 \neq 0$, som altså skal sikre, at der returneres FAIL hvis $\|x\|_0 > 1$. Dette vises med følgende omskrivning taget fra [2]:

$$\begin{aligned}
c_0 \cdot c_2 - c_1^2 &= (\sum_{i \in [U]} x_i) \cdot \sum_{i \in [U]} x_i \cdot i^2 - (\sum_{i \in [U]} x_i \cdot i)^2 \\
&= (\sum_{i \in [U]} x_i) \cdot \sum_{i \in [U]} x_i \cdot i^2 - \sum_{i,j \in [U]} x_i \cdot i \cdot x_j \cdot j \\
&= \sum_{i,j \in [U]} x_i x_j \cdot j^2 - \sum_{i,j \in [U]} x_i x_j \cdot i \cdot j \\
&= \frac{1}{2} \sum_{i,j \in [U]} x_i x_j \cdot j^2 + \frac{1}{2} \sum_{i,j \in [U]} x_i x_j \cdot i^2 - \sum_{i,j \in [U]} x_i x_j \cdot i \cdot j \\
&= \frac{1}{2} \sum_{i,j \in [U]} x_i x_j (j^2 - 2ij + i^2) \\
&= \frac{1}{2} \sum_{i,j \in [U]} x_i x_j (j - i)^2
\end{aligned}$$

Ud fra denne omskrivning er det tydeligt, at hvis vi har tilfældet $\|x\|_0 = 1$, er $j - i = 0$, hvilket gør, at hele summen bliver 0. Pga. $x_i \geq 0$ for alle $i \in [U]$ ved vi så, at $c_0 \cdot c_2 - c_1^2$, hvis og kun hvis $\|x\|_0 > 1$. Det er derfor, at det med tilføjelsen af c_2 opnås, at UEen bliver i stand til selv at holde styr på, om der skal returneres FAIL eller ej.

Som i [1] skrives der i [2], at UE bruger $\mathrm{O}(\log(UM))$ bits af plads. Der følger dog denne gang et argument med om hvorfor, nemlig at den maksimale værdi af c_0, c_1, c_2 er $\mathrm{O}(U^3 \cdot M)$, og det giver det nævnte pladsforbrug. Det er selvfølgelig samme type af argument for UEen i [1], men dét, at det er nævnt i [2], viser meget godt forskellen på mængden af detaljer i de to artikler.

4.4.3 Destinct Elements (DE)

Destinct Elements er en struktur, hvis opgave det er at holde styr på hvor mange forskellige elementer en given mængde indeholder. I dette tilfælde hvor mange af indgangene i den beskrevne vektor der har en værdi større end 0. Både [1] og [2] specificerer, at denne struktur på samme måde som UE skal understøtte to metoder Update() og Report() på vektoren $x = (x_0, \ldots, x_{U-1})$, dog her ved DE defineret som:

- Update(i, a). Er defineret som Update angivet i afsnit 4.3.
- Report(). Med sandsynligheden $1 - \delta$ returneres en værdi k sådan så $k \leq \|x\|_0 \leq k(1+\psi)$. Tallene $\delta, \psi > 0$ er parametre. (Variablernes navne er fra [2])

For både [1] og [2] gælder det, at der ikke skrives meget om, hvordan DEen rent faktisk skal laves for at understøtter disse to metoder. De to artikler referer i stedet begge til andre artikler og skriver, at datastrukturer fra disse vil give det ønskede. [1] og [2] henviser dog til to forskellige artikler.

[1], som jo var mit oprindelige udgangspunkt for specialet, refererer til artiklen [13]. Dette er en artikel fra 1985, som handler om, hvordan man kan tælle elementerne i en mængde ud fra opdateringerne i en stream. Umiddelbart lyder det jo meget fornuftigt, men i praksis viser det sig, at datastrukturen beskrevet i [13] ikke er helt så let anvendelig, som man kunne ønske sig, i den sammenhæng [1] vil bruge den i. Jeg går derfor i afsnit 4.5 i detaljer med, hvad det er for en datastruktur for DE [13] beskriver, og i afsnit 5.6 kommer jeg ind på hvilke problemer, [1]s brug af DEen har medført for min implementation.

[2] henviser derimod til [14] fra 2004, som ligeledes opbygger en datastruktur til at tælle elementerne ud fra en stream. [14] bygger i væsentlig grad på ideerne fra [13], men udgangspunktet er, at man vil lave en bedre løsning, og dermed fjerne nogle af de svagheder der er i [13].

Da det først var efter jeg var færdig med min implementation, at jeg fandt [2], og dermed fik kendskab til [14], har jeg valgt ikke at ændre min implementation til at basere DEen på [14]. Et skift til DEen fra [14] burde godt nok give bedre præcision, og havde jeg brugt [14] fra starten, havde nogle af manglerne jeg mødte i [13] ikke været relevante. I bund og grund gør de to typer af DEere dog det samme, og for min endelige vurdering af datastrukturen fra [1] og [2] har det ingen betydning. I afsnit 4.5.4 kommer jeg kort ind på nogle af de forandringer, der er i [14], og hvad de ville have betydet, men jeg prøver ikke på at give en fyldestgørende beskrivelse af DEen i [14].

I [1, afsnit 2] anføres det, at brugen af [13] kan give en ønsket DE med et pladsforbrug på $O(\log^2(MU/\delta)/\psi^2)$. I [2, afsnit 3.2] anføres det, at brugen af [14] kan give et pladsforbrug for DE på $O(1/\psi^2 \cdot \log(1/\psi)\log(1/\delta)\log(U)\log(UM))$. Hverken [1] eller [2] skriver, hvordan de kommer frem til disse størrelser, og da det i sidste ende endte med ikke at have nogen indflydelse på min vurdering af den praktiske anvendelse, har jeg ikke forsøgt at efterprøve, om størrelserne er rigtige.

4.4.4 Update()

Den ene af de to metoder i datastrukturen er Update(), og det er i denne, vi får første halvdel af forklaringen på, hvordan undergrupperne vist i figurerne 4.5 og 4.6 fungerer.

Algorithm 1 Update(i, a) fra [1]

```
for j ∈ [log U] do
  if h_j(i) = 0 then
    UE_j.update(i, a); DE_j.update(i, a)
  end if
end for
DE.update(i, a)
```

Algorithm 2 Update(i, a) fra [2]

```
for j ∈ [log U] do
  if h_j(i) = 0 then
    UE_j.update(i, a)
  end if
end for
DE.update(i, a)
```

Pseudokoden for Update() i henholdsvis [1] og [2] er vist ovenfor. På nær at der i Update() fra [2] ikke er nogen DE_j at opdatere, så er de ellers identiske. Det vigtige at lægge mærke til ved dem begge er checket $h_j(i) = 0$. Som det kan ses, løber begge Update() gennem alle de j undergrupper, og det er så checket, der afgør, hvilke undergrupper der opdateres af den givne indgang. Jævnfør afsnit 4.4.1 betyder måden hash funktionerne er defineret på, at $h_0(i) = 0$ for alle indgange i i vektoren x, $h_1(i) = 0$ for ca. 50% af indgangene, $h_3(i) = 0$ for ca. 25% osv.

Denne måde at fordele indgangene i vektoren x på over undergrupperne er, hvad der sætter datastrukturen i stand til i Sample() at finde en undergruppe hvor UEen kun indeholder én indgang med positiv værdi fra x.

4.4.5 Sample()

Den anden metode i datastrukturen er Sample(), og giver den anden halvdel af forklaringen på, hvordan undergrupperne bruges.

Algorithm 3 Sample fra [1]

```
j = ⌈log(DE.report)⌉
if DE_j.report= 1 then
  return UE_j.report
else
  return FAIL
end if
```

Algorithm 4 Sample fra [2]

```
j = ⌈log(DE.report)⌉
return UE_j.report
```

Som ved Update() er pseudokoden for Sample() også næsten identisk i [1] og [2]. Det er her i Sample(), at fordelingen over undergrupperne, Update() foretager med indgangene i vektoren x, bliver brugt. Linien $j = \lceil \log(\text{DE.report}) \rceil$ bruges nemlig til at identificere den UE_j, der har størst chance for kun at indeholde én indgang med positiv værdi fra x.

Det er også her i Sample(), at ændringen af UE i [2] har en betydning. Hvor Sample() i [1] bruger checket DE$_j$.report= 1 til at sikre, at UE$_j$ indeholder et gyldigt sample, så kan Sample() i [2] derimod bare kalde UE$_j$.report direkte. Udover at den ændrede UE sparer datastrukturen for j DEere, så er det også checket DE$_j$.report= 1, der voldte mig en del problemer under implementeringen. Mere om det i afsnit 5.6.5.

4.4.6 Korrektheden af datastrukturen

I de forrige afsnit har jeg beskrevet datastrukturens opbygning. Inden der dog kan summeres op, hvad der er opnået med teorien fra [1] og [2], er der to områder, der bør berøres:

- **Korrektheden.** Både i [1, afsnit 2] og i [2, afsnit 3.3] argumenteres der for korrektheden af datastrukturen. Som udgangspunkt antages det, at DE strukturen er korrekt i dens estimering. Jævnfør afsnit 4.4.3 er den det med en sandsynlighed på mindst $1 - \delta$. Er det tilfældet, har vi, at UE$_j$.Report $\neq$ FAIL hvis og kun hvis $|\text{Supp}(x) \cap h_j^{-1}(0)| = 1$. Altså i de tilfælde hvor hash funktionen h_j har givet 0 for kun én af indgangene i i vektoren med en værdi større end 0. Under antagelse af, at vi har helt tilfældige hash funktioner til rådighed, vil indgangen returneret af UE$_j$ altså være valgt uniformt tilfældigt fra Supp(x).

  Det skal så vises, hvad den nedre grænse er, for sandsynligheden for at vi har $|\text{Supp}(x) \cap h_j^{-1}(0)| = 1$. I [2] lader de $S_j = h_j^{-1}(0)$ og $\ell = |\text{Supp}(x)|$. Det hævdes derefter, at ud fra korrektheden af DE, følger det at $\ell \leq 2^j \leq 4\ell$. Det giver

  $$\Pr[|S_j \cap \text{ Supp}(x)| = 1] = \ell \cdot 2^{-j} \cdot (1 - 2^{-j})^{\ell-1} \geq 1/4 \cdot (1/4)^4 = 1/4^5$$

- **Tilfældigheden af hash funktionerne**. Ind til nu har teorien i både [1] og [2] været baseret på, at vi har fuldstændigt tilfældige hash funktioner til rådighed. Dette er ikke nogen god antagelse, da en implementation i sidste ende vil være baseret på tilfældige tal genereret i en computer, som ikke har mulighed for at give perfekt tilfældighed. Både [1] og [2] henviser til, at det er nok at have en pseudo tilfældig generator til rådighed, og argumenterer for rigtigheden af dette. Begge henviser til artikler (dog ikke de samme...), der handler om hvordan man laver pseudo tilfældigheds generatorer.

  Udover informationen om at pseudo tilfældigheds generatorer er gode nok, så har det ikke haft den store betydning for mit arbejde med datastrukturen. Som jeg kommer ind på i afsnit 5.6.2, har jeg i min implementation brugt en tilfældigheds generator, der var stillet til rådighed i mit udviklingsværktøj. Da dette viste sig at være rigeligt til min vurdering af datastrukturen, har jeg valgt ikke at bruge ekstra tid på at sætte mig ind i artiklerne om tilfældighed, som [1] og [2] refererer til. Disse artikler fremgår derfor heller ikke af min litteraturliste.

Med disse to områder kort berørt er det tid til at skrive, hvad al denne teori ender ud med. Som jeg skrev i Indledningen til specialet (afsnit 1.4), så er der fejl i det Lemma 2.1 fra [1], der ellers skulle give denne opsummering af det opnåede med teorien. Det var dét, der var årsagen til, at jeg fandt [2], og det tilsvarende lemma her er ændret til noget, der lyder til at passe en del bedre. Altså, fra [2, afsnit 3.3]:

- **Lemma 3.3** *Given a sequence of update operations on a vector* $x = (x_0, \ldots, x_{U-1})$ *with entries from* $[M]$*, there is a data structure that with*

probability $(1/4)^6 - \delta$ *returns a pair* (i, x_i) *with* $i \in Supp(x)$ *such that* $Pr[i = j] = 1/\|x\|_0 \pm \delta$ *for every* $j \in Supp(x)$*. The algorithm uses* $O(\log^2(UM/\delta))$ *space.* □

Dette lemma opsummerer, hvad forfatterne af [1] og [2] når frem til, at man opnår, hvis man følger specielt [2] til punkt og prikke. En lille forskel, der er værd at bemærke mellem det viste lemma og Lemma 2.1 fra [1], er, at det opnåede nu betegnes som "a data structure"i stedet for "a streaming algorithm". Det passer en del bedre sammen med, at man jo netop forsøger at erstatte en stor datastruktur (vektoren x) med noget, der fylder mindre.

4.5 Destinct Elements i detaljer

Før jeg kan runde beskrivelsen af datastrukturen fra [1] helt af, er der et enkelt område, jeg synes bør belyses særskilt. Som jeg skrev i afsnit 4.4.3, så skrives der i [1], at DEerne kan laves ud fra datastrukturer beskrevet i [13], så de opfylder det ønskede. Udover dette nævnes det ikke, hvad [13] rent faktisk fortæller om disse datastrukturer. Eftersom DEerne spiller en væsentlig rolle i datastrukturen, og derfor har stor betydning for min implementering, bruger jeg dette afsnit på at give en gennemgang af DEen foreslået i [13].

[13] består for en stor del af beviser, for hvorfor dét, der gøres, er korrekt, men da jeg som ved [1] vil undersøge den praktiske anvendelse, vælger jeg blot at antage, at teorien er i orden. I dette afsnit gennemgår jeg derfor de ting fra [13], jeg fandt nødvendige for at forstå, hvad der skulle gøres for at implementere DE strukturerne ud fra [13].

Den opdaterede artikel [2] gør heller ikke meget ud af DEerne, men der er dog den forskel, at den vælger at refere til [14] for konstruktionen af disse datastrukturer. [14] er en artikel fra 2004, der bl.a. er baseret på [13], men da jeg som tidligere skrevet først fandt [2], efter at jeg var praktisk talt færdig med implementeringen, vil jeg kun kort beskrive [14] i slutningen af dette afsnit.

[13] er en artikel fra 1985, hvori der beskrives en måde, hvorpå man med begrænset brug af lagerplads, kan estimere antallet af forskellige elementer i en given mængde ud fra et enkelt gennemløb. Motivationen bag [13] anføres til primært at være, at lave et redskab til brug i database systemer hvor man arbejder med store datamængder, og kendskabet til antallet af forskellige elementer i visse situationer er væsentligt. Det afgørende for brugen i forbindelse med [1] er, at datastrukturen der foreslås, altså kun behøver ét gennemløb af datamængden for at kunne give den ønskede estimering. Dette passer perfekt med en data stream, hvor man kun har mulighed for at behandle de modtagne elementer én gang.

4.5.1 Den grundliggende ide

Det antages i [13, afsnit 2], at vi til vores rådighed har en hash funktion af typen:

$$\text{function hash}(x : \text{ records}) \text{: scalar range } [0 \cdots 2^L - 1]$$

som sender en given type af records (elementerne vi ønsker at tælle) over i mængden af binære strenge af længden L med en tilpas uniform fordeling. I så fald kan det observeres, at ved outputet af hash(x) vil bitmønstret $0^k 1 \cdots$ forekomme med sandsynligheden 2^{-k-1} ([13, s. 185]). Ikke noget specielt i det. Ideen er dog, at vi kan bruge disse mønstre, til at give den ønskede estimering, på hvor mange forskellige elementer vi har set.

For at kunne bruge mønstrene på denne måde, skal der først et par definitionen på plads. For ethvert integer $y \geq 0$ definerer vi bit(y, k) til at være bit k i den

binære repræsentation af y. Dvs.

$$y = \sum_{k \geq 0} \text{bit}(y, k) 2^k$$

Samtidigt defineres funktionen $\rho(y)$ til at repræsentere positionen af den mindst betydende 1-bit i repræsentationen af y (mindst betydende bit har position 0). Definitionen er

$$\begin{aligned} \rho(y) &= \min_{k \leq 0} \text{bit}(y, k) \neq 0 && \text{if } y > 0 \\ &= L && \text{if } y = 0 \end{aligned}$$

hvor L altså er længden af de binære strenge, hash funktionen fordeler de givne records over ([13, s. 184-5]). Når vi så får givet en mængde M, vi ønsker at kende antallet af forskellige elementer i, laver vi en vektor af bits $BITMAP[0 \ldots L-1]$, og gør følgende:

```
for i = 0 to L − 1 do
  BITMAP[i] = 0
end for
for all x ∈ M do
  index = ρ(hash(x))

  if BITMAP[index] = 0 then
    BITMAP[index] = 1
  end if
end for
```

Kørslen af disse operationer betyder, at BITMAP[i] $= 1$ hvis og kun hvis vi mindst én gang har mødt mønstret $0^i 1$ blandt de hashede værdier fra elementerne i M. Hvor mange gange, vi har set et givent mønster, kan ikke læses ud fra BITMAP vektoren, kun at vi har set det mindst én gang.

Som [13, s. 185] så efterfølgende gør rede for, er ideen, at hvis n er antallet af forskellige elementer i M, så vil BITMAP[0] blive forsøgt sat til 1 ca. $n/2$ gange, BITMAP[1] ca. $n/4$ gange osv. Efter alle elementerne n har været gennem ovenstående operationer, vil BITMAP[i] med stor sandsynlighed være 0, hvis $i \gg \log_2 n$ og 1 hvis $i \ll \log_2 n$. Mellem disse bits vil der være en blanding af 0 og 1 omkring $i \approx \log_2 n$.

Som eksempel nævnes det ([13, s. 185]), at forfatterne kørte en Unix online manual bestående af 26692 linier (heraf var 16405 forskellige) gennem operationerne med hashingen sat til at ske over 24 bits strenge. Resultatet blev

111111111111001100000000

Det 0 der står længst til venstre, er placeret i position 12 (positionerne tælles fra venstre og starter ved 0), og det 1 der står længst til højre, er placeret i position 15. Det bemærkes af forfatterne til [13], at $2^{14} = 16384$ - dvs. tallet 14 opløfter 2 til den nærmeste værdi til det rigtige antal forskellige linier. Det foreslås derfor, at bruge positionen af det 0 der står længst til venstre som en indikator for $\log_2 n$. Kalder vi denne position for 0-bitten længst til venstre for R, går [13] videre og beregner, at den forventede værdi for R, når de hashede værdier er uniformt fordelt, er tæt på:

$$\mathbf{E}(R) \approx \log_2 \varphi n, \qquad \varphi = 0,77351 \cdots$$

Desuden angiver de, at standard afvigelsen for R er tæt på

$$\sigma(R) \approx 1,12$$

Disse resultater om den forventede værdi for R og den tilhørende korrektionsfaktor φ bliver efterfølgende bevist i [13]. Jeg ønsker dog ikke at gennemgå beviset her, så jeg antager blot, at det er korrekt. Det vigtige i forhold til DE datastrukturen er altså, at man med den nævnte algoritme kan bruge placeringen af den 0-bit, der står længst til venstre til at give en estimeret værdi Ξ for antallet af forskellige elementer n. Dvs.

$$\Xi = \frac{2^{R_n}}{\varphi} \tag{4.1}$$

4.5.2 Bedre præcision

Resultatet fra 4.5.1 er brugbart, men den nævnte standard afvigelse for R på 1,12 gør, at resultaterne stadig kan svinge en del. I [13, afsnit 3] foreslås derfor en måde, hvorpå størrelsen af afvigelserne kan reduceres drastisk.

Ideen er denne gang, at i stedet for kun at benytte én hash-funktion som beskrevet i 4.5.1 så bruges der i stedet m hash-funktioner til at beregne m forskellige BITMAP vektorer. På den måde får man altså m estimeringer $R^{<1>}, R^{<2>}, \ldots, R^{<m>}$, hvorfra man udregner gennemsnittet

$$A = \frac{R^{<1>} + R^{<2>} + \ldots + R^{<m>}}{m} \tag{4.2}$$

Med n forskellige elementer i den undersøgte mængde, konstaterer [13, s. 196], at A har en forventet værdi på

$$\mathbf{E}(A) \approx \log_2 \varphi n$$

og en standard afvigelse der tilfredsstiller

$$\sigma(a) \approx \frac{\sigma_\infty}{\sqrt{m}}$$

Brugen af algoritmer, der udnytter denne form for gennemsnitlig beregning, giver ifølge [13, s. 196] meget gode resultater. Ved f.eks. $m = 64$ er den gennemsnitlige fejl ved den estimerede værdi for de n forskellige elementer ca. 10%. Prisen for dette er dog en m-dobling af CPU behovet pr. element i mængden, fordi alle elementer bliver kørt gennem de m forskellige hash-funktioner. Dvs. valget af ønsket præcision har direkte indflydelse på udførelsestiden af en sådan algoritme.

Et sådant øget CPU forbrug er ikke nødvendigvis acceptabelt (og nok slet ikke i 1985...), så [13] har også en løsning på dette. Løsningen kalder de for *stochastic averaging* ([13, s. 196]). Løsningen er at i stedet for at lade alle elementerne i mængden blive hashed af m hash-funktioner, så bruges der i stedet kun én hash-funktion, hvis resultater i stedet fordeles over m forskellige BITMAP vektorer. Det gøres ved at beregne $\alpha = h(x) \mod m$ og så opdatere BITMAP vektor α med den resterende del af resultatet fra $h(x)$, nemlig $h(x)$ div $m \equiv \lfloor h(x)/m \rfloor$. Efter BITMAP vektorerne er blevet opdateret med alle elementerne, findes $R^{<j>}$erne som tidligere, og gennemsnittet A beregnes som i (4.2). Med håbet om at de hashede værdier af elementerne er blevet fordelt jævnt, så kan det antages, at ca. n/m elementer er blevet brugt i konstruktionen af hver BITMAP vektor. Jævnfør (4.1) betyder det, at A kan bruges til at estimere n/m. Beregningen af den estimerede værdi for det samlede antal forskellige elementer Ξ er derfor

$$\Xi = \frac{m(2^A)}{\varphi}$$

4.5.3 PCSA algoritmen

Resultaterne fra afsnit 4.5.2 bruges i [13] til at konstruere en algoritme kaldet *Probabilistic Counting with Stochastic Averaging* (fremover refereret til som PCSA). Den fulde pseudokode for denne algoritme kan ses i Appendix B. Umiddelbart er den meget ligetil at overskue, men der er nogle enkelte ting, som er specielt værd at bemærke.

- **Pladsforbrug af BITMAP vektorerne**

```
begin
    while not eof(M) do
    begin
        getelement(x); hashedx := hash(x);
        alpha := hashedx mod nmap; index := p(hashedx div nmap);
        if BITMAP[alpha, index] = 0 then BITMAP[alpha, index] := 1;
    end;
```

  Ovenfor er anført et udsnit af pseudokoden for PCSA, som den er vist i [13, s. 197]. Udsnittet viser while-løkke, som tager imod elementerne, vi ønsker af gennemgå, og derefter bruges ideerne beskrevet i 4.5.2 til at bestemme hvilken position i hvilken BITMAP-vektor der skal sættes til 1. Umiddelbart ser det ikke ud som om, der er nogen overraskelser, men der er én ting, som er lidt spøjs, når nu målet er at estimere antallet af forskellige elementer ved brug at så lidt lagerplads som muligt.

  Der oprettes i PSCA et array til BITMAP vektorerne, og det er deres størrelser, der fastsætter algoritmens pladsforbrug. I artiklen [13, s. 205] er de allerede inde på, at ser man på denne matrix af bits indeholdende vektorerne, vil den ofte have et udseende svarende til

$$\begin{array}{ccc} 1111 & 10100 & 0000 \\ 1111 & 11000 & 0000 \\ 1111 & 01011 & 0000 \\ 1111 & 11010 & 0000 \end{array}$$

  Dvs. der vil være en sektion af 1-bits til venstre, en sektion af 0-bits til højre og så en blandet sektion i midten. I [13] skriver de så om, at man kan mindske pladsforbruget, hvis man laver et "scrolling window", der dynamisk kan nøjes med at holde styr på den "interessante"midte af matricen. Noget jeg undrede mig over, da jeg læste dette, er dog, at de ikke nævner, at en del af 0-bittene i højre side helt kan fjernes. Ser vi nemlig på linien

```
alpha := hashedx mod nmap; index := p(hashedx div nmap);
```

  er det denne, der i PCSA afgør hvilken indgang i BITMAP-matricen, der evt. skal sættes til 1. Det spøjse er dog, at *index* aldrig kan antage en værdi, så den fulde længde af de enkelte BITMAP-vektorer udnyttes. I [13] bliver BITMAP-matricen nemlig instantieret som

```
BITMAPS: array[0..nmap-1; 0..maxlength-1] of integer;
```

  hvor der altså ikke tages højde for, at brugen af div i forbindelse med beregningen af *index* sætter en begrænsning på dennes størrelse. Hvis der - som i pseudokoden for PCSA - regnes på integers af størrelsen 32-bit (angivet ved *maxlength*) og nmap = 64, så giver det, at $0 \leq index \leq 25$. Med nmap= 64 ville man altså kunne reducere matricens størrelse med 6/32 allerede ved instantieringen uden at påvirke algoritmens egenskaber.

- **PSCA ved få forskellige elementer**

Der nævnes i [13, s. 203], at PSCA først begynder at estimere antallet af forskellige elementer som forventet, når n/nmap begynder at overstige 10-20. Dette har stor betydning for implementationen af datastrukturen fra [1], da Sample() metoden beskrevet i afsnit 4.4.5 er baseret på, at DE-strukturen kan tælle helt ned til et enkelt element. Faktisk er der det problem, at ser man på estimeringen, der returneres af PSCA

$$\Xi = \frac{m(2^A)}{\varphi} \tag{4.3}$$

og på den tilsvarende pseudokode

```
S = 0;
for i := 0 to nmap-1 do
begin
    R := 0;
    while (BITMAP[i,n]=1) and (R<maxlength) do R := R+1;
    S := S+R;
end;
Xi := trunc((nmap/phi)*2^(S/nmap));
```

så er det næsten umuligt for PSCA at outputte værdien 1. Kun hvis vi accepterer meget dårlig præcision, og lader nmap = 1, og har $S = 0$ i pseudokoden, og dermed $A = 0$ i 4.3, så vil resultatet efter brug af trunc-funktionen være et estimat på 1. Samtidigt er risikoen for, at dette resultat er forkert stor, da $S = 0$ kun indikerer, at ingen hashede værdier har haft en 1-bit som mindst betydende bit. Hvis der altså overhovedet er set nogen elementer endnu. Ved en brug af PSCA med nmap = 64 som [13] foreslår for en præcision på ca. 10%, vil den mindst mulige værdi PSCA kan returnere være trunc$(64/\varphi) = 82$ når $S = 0$.

I [13, s. 203] nævnes det, at man måske kan indføre korrektioner i PSCA algoritmen, hvis man har brug for at kunne estimere meget små værdier. I [1] berøres dette problem slet ikke, der nævnes bare i forbindelse med beskrivelsen af de ønskede DE-strukturer at "one can use data structures from [13] to solve this problem". Vil man implementere datastrukturen fra [1] er man altså nødt til at modificere PSCA for at få fungerende DE-strukturer. Min modifikation er beskrevet i afsnit 5.6.5.

- **Tilfældigheden af de brugte hash funktioner.** Det er faktisk forbavsende lidt, der i artiklen bliver sagt noget om kravet til tilfældighed af de brugte hash funktioner. Da funktionerne første gang bliver nævnt ([13, afsnit 2]) omtales kravet til hash funktionerne blot som: "transforms records into integers sufficiently uniformly distributed". Senere i artiklen nævnes, hvordan forfatterne i deres brug af datastrukturen på tekst filer har brugt en hash funktion, som de - efter et uspecificeret antal kørsler - synes giver en god uniform fordeling. I princippet vil en "dårlig"hash funktion med en mindre god fordeling nok blot betyde en reduktion af datastrukturens præcision, men ud fra [13] er det er altså op til en selv at finde en sådan "god nok"hash funktion.

4.5.4 Den nye datastruktur for DE i [2]

Hvor den oprindelige artikel [1] henviser til, at man bør lave DE datastrukturer på baggrund af [13], er det i [2] blevet ændret til, at man bør lave dem på baggrund af [14]. Som tidligere nævnt var det først sent i arbejdet med specialet, at jeg blev konfronteret med denne ændring, og min implementation bygger derfor på [13]. Da [2] dog på alle områder lader til at være en god revidering af den oprindelige artikel, vil jeg i dette afsnit meget kort komme ind på nogle af de ting i [14], som gør den til et attraktivt grundlag for DE strukturerne.

Den nok mest oplagte grund til at [14] i sidste ende er bedre, er netop at forfatterne af [14] gør meget ud af at fortælle, hvordan de har forsøgt at forbedre PCSAen foreslået i [13]. I indledningen til artiklen ([14, afsnit 1]) nævnes [13] og senere bruger de tid på at beskrive i detaljer, hvordan datastrukturen i [13] fungerer ([14, afsnit 2.2]).

Den grundliggende ide i [14] er den samme som den i [13] med en enkelt større tilføjelse. På samme måde som i [13] benyttes hash funktioner til at opbygge bit mønstre så som

$$1111111111111001100000000$$

Har man et domæne af elementer $[M]$ har vi altså $\log(M)$ bit vi sender elementerne ned i ([14, afsnit 3.1]). Ændringen i forhold til [13] er så, at for hver bit-indgang vi mapper ned i, har vi også et array af $\log(M) + 1$ integers. Den første integer bruges altid til at holde styr på det samlede antal elementer, der er mappet ned i den tilhørende bit-indgang. Resten af integerne bruges til at tælle placeringerne af 1-bits for hvert enkelt indgang i de mappede elementers binære repræsentation. Dvs. når et element mappes til en given bit-position (som i PCSA), så vil det tilhørende array af integers blive talt op med 1 de steder, der passer til 1-bittene i elementets binære repræsentation.

Hele ideen med dette er både, at der bliver holdt styr på antallet af elementer, der er blevet mappet, så man efterfølgende kan fjerne dem igen, og at man ved hjælp af denne opbygning bliver i stand til at tælle elementer fra flere forskellige data streams. En stor del af motiveringen til denne brug af integer arrayet i [14] går netop på, at udover at kunne estimere antallet af forskellige elementer i en stream A så vil man også gerne kunne estimere f.eks. $A \cap B$ og $A \cup B$ hvis man har to seperate data streams A og B. I det hele taget er ideen, at man skal kunne gøre dette for et vilkårligt stort antal data streams.

[14] går ind i en hel del teori for hvorfor deres foreslåede datastruktur gør dem i stand til at give disse estimeringer over forskellige data streams, der i sidste ende ikke er relevant for dette speciale. De kommer også ind på teorien for, hvorfor man kan bruge pseudo tilfældige hash funktioner i deres struktur, da de netop ser det som en svaghed i [13] ([14, afsnit 1]).

Set ud fra en praktisk synsvinkel er datastrukturen fra [14] nok at foretrække, da den altså er en videreudvikling af PCSA fra [13]. I min konkrete brug var der dog ikke stor nok forskel til, at jeg ville ændre min implementation, efter at jeg blev opmærksom på [14]. Som jeg beskriver senere i afsnit 5.6.4, havde jeg dog allerede selv udvidet PCSA med en enkelt tæller for at kunne håndtere fjernelse af elementer. I den sammenhæng er det rart at se, at [14] gør det samme.

4.6 Min anvendelse af datastrukturen

Dét, der oprindeligt fangede min interesse for datastrukturen, var, at der i [1, afsnit 1] blev nævnt lagring af statistiske oplysninger om trafikken på internet routere som et eksempel på motivationen bag datastrukturen. Jeg bestemte mig derfor tidligt i specialet for, at mit mål med [1] skulle være at anvende datastrukturen på en eller anden form for netværks relateret trafik, og så vurdere om denne form for brug af strukturen er brugbar eller ej.

Som nævnt i afsnit 4.3 er det grundliggende krav, for at datastrukturen kan bruges, at man har et problemområde, der kan inddeles i et endeligt antal elementer, hvortil der kan tilknyttes en værdi. Det er så denne værdi, der repræsenterer dét, det ønskes at måle for hvert element og dermed føre statistik over. Pga. det nævnte eksempel i [1] kredsede mine første ideer omkring måling af belastningen på routere. F.eks. ved at lade indgangene i den underliggende vektor repræsentere trafikken fra én gruppe IPer til en anden og så lade værdien repræsentere det antal bytes

der havde passeret routeren. Det praktiske problem i at få adgang til den slags oplysninger fra en kørende router gjorde dog, at jeg i stedet begyndte at overveje, om jeg kunne bruge eksisterende logs fra allerede kørende servere, til at danne grundlag for en simulering af trafikken serveren ser.

Det endte med, at jeg fik adgang til et halvt års access log fra en kørende Apache webserver[1]. Denne log indeholder bl.a. informationer om IPerne på de computere, der har kontaktet serveren, tidspunkt for kontakten og hvilken hjemmeside-fil de anmoder om at få tilsendt. Et eksempel er

```
xxx.xxx.xxx.xxx - - [10/Apr/2006:06:06:14 +0200] "GET /robots.txt HTTP/1.0"
    404 208 "-" "Gigabot/2.0/gigablast.com/spider.html"
```

der angiver IPen på en computer, der anmoder om filen "robots.txt"fra serveren. Som beskrevet i afsnit 2.3 er det en videnskab i sig selv at afgøre, hvilke oplysninger der er værd at gemme, så jeg valgte tidligt at holde den del simpel. Jeg bestemte mig for at lade elementerne i mit problems univers repræsentere IP intervaller (dvs. én eller flere IPere) og så lade den tilhørende registrerede værdi være antallet af connections registreret fra den/de pågældende IPere. På den måde kan man opbygge et simpelt søjlediagram, over IPerne der kontakter serveren, hvor de høje søjler repræsenterer IP intervaller, der har kontaktet serveren mange gange. Har man f.eks. et løbende opdateret diagram over de sidste 24 timers trafik på en server, og ved man, hvordan dette diagram ser ud på en normal dag, kan man nemt spotte usædvanlig trafik (f.eks. som følge af denial-of-service angreb).

Til at undersøge datastrukturen fra [1] har jeg valgt at lade den blive brugt til at generere sådanne søjlediagrammer. Ideen skulle så gerne være, at hvor man på traditionel vis kan gemme og løbende opdatere et nøjagtigt diagram med dertil hørende pladsforbrug, så kan datastrukturen forhåbentlig på vilkårlige tidspunkter generere et "ok"diagram ud fra sin sampling ved et langt mindre pladsforbrug. Hvad jeg mener med "ok"er lidt svært at definere på dette tidspunkt. Det venter jeg med til kapitel 6, hvor jeg tester implementationen af min anvendelse.

4.6.1 Forventninger til datastrukturen

I begyndelsen af mit arbejde med specialet var mit indtryk af datastrukturens egenskaber meget positive, og jeg regnede med at kunne opnå gode resultater med den. På det tidspunkt så jeg den eneste svaghed som værende muligheden for FAILs ved brugen af Sample() metoden. I bund og grund var det endda ikke nogen svaghed, for risikoen for FAILs var, hvad der definerede datastrukturen som værende samplende i stedet for at være en lossless pakkerutine. Kunne Sample() ikke give FAIL, ville man kunne trække alle de sete elementer ud af strukturen, ved rekursivt at kalde skiftevis Sample() og derefter Update() med værdien der fjerner elementet netop fundet af Sample(). På den måde ville alle elementerne kunne blive pillet ud et efter et. FAIL er årsagen til, at man ikke kan regne med at gøre dette.

At jeg så risikoen for FAILs som en svaghed, var mere fordi jeg var bekymret for, hvor stor denne risiko ville være. Som tidligere beskrevet gøres der i [1, afsnit 2] opmærksom på, at så længe sandsynligheden for FAIL er < 1, så kan denne altså gøres vilkårligt lille ved at bruge flere datastrukturer parallelt. Problemet var bare hvor mange parallelle datastrukturer - med der tilhørende pladsforbrug - der ville være påkrævet for at gøre risikoen for FAIL lille nok ved et givent problem.

Da jeg senere i forløbet begyndte at arbejde med min implementation af datastrukturen, begyndte jeg i stedet at blive bekymret over det faktum, at når først datastrukturen var blevet instantieret, så var den fuldstændigt deterministisk. Det gik f.eks. op for mig, at to kald af Sample() uden nogen mellemliggende kald af Update() vil returnere samme "samplede"værdi begge gange. Det er gennem brugen

[1]Nærmere bestemt for domænet www.246.dk

af Update(), der ændrer datastrukturen, der gør, at valget af værdier, der samples, kan ændre sig. Min bekymring gik derfor på dette tidspunkt på en helt specifik type situation, jeg mente kunne opstå i mit valg af anvendelse: At man foretager en sampling af datastrukturen, og ikke får nok værdier ud pga. FAILs til at kunne generere et brugbart histogram. I en sådan situation tænkte jeg, at en bruger ville være nødt til at vente på kald af Update(), førend en ny sampling kunne foretages. Jeg regnede med at i min implementation ville det være noget jeg kunne måle på (hvor lang tid skal man vente i gennemsnit før end x% af diagrammet er genereret), men jeg var bekymret for, hvad dette ville medføre for datastrukturens værdi i praksis. F.eks. ville en netværks administrator, der checker trafikken på en server, nok ikke være tilfreds med at skulle vente i al for lang tid på at få opbygget et brugbart diagram.

På trods af disse konkrete bekymringer havde jeg langt hen i forløbet positive forventninger til den praktiske værdi af datastrukturen. Som det fremgår af min vurdering i kapitel 7, endte det dog en del anderledes, end jeg havde regnet med.

Kapitel 5

Min implementation

Dette kapitel handler om min arbejde med implementeringen af datastrukturen fra [1]. Jeg antager i dette kapitel, at læseren allerede har kendskab til programmering og gængse programmeringssprog såsom C og C++. Jeg beskriver derfor programkoden i overordnede træk, og vil kun gå i detaljer med de steder, hvor jeg synes, at der er detaljer af speciel interesse. Bl.a. de steder, hvor [1] har været uklar, og jeg derfor har været nødt til at improvisere for at få datastrukturen til at fungere.

Som tidligere nævnt lavede jeg implementationen, inden jeg fik kendskab til [2]. Min implementation er derfor baseret på [1], og det eneste, jeg har ændret, efter jeg fik kendskab til [2], er, at jeg har lavet den simple forbedring af UE datastrukturen beskrevet i [2].

5.1 Målet for implementationen

Mit oprindelige mål for implementationen (afsnit 4.6) var løst defineret ud fra, at jeg ville bruge datastrukturen til at holde styr på antallet IPer, der kontakter en Apache webserver. Jeg fik i den forbindelse undervejs i forløbet adgang til et halvt års log over aktiviteten på domænet www.246.dk. Webservere kan konfigureres til at logge, hvad den givne administrator måtte ønske, og i access-loggen for www.246.dk's tilfælde er alle registreringer af formen

```
xxx.xxx.xxx.xxx - - [10/Apr/2006:06:06:14 +0200] "GET /robots.txt HTTP/1.0"
     404 208 "-" "Gigabot/2.0/gigablast.com/spider.html"
```

Access-loggen dækker nemlig over, hvilke IPer, der på hvilke tidspunkter, anmoder om navngivne filer fra serveren. Dertil kommer en række status oplysninger.

Da jeg fik loggen, overvejede jeg, om der var en smart måde, jeg kunne lave en historik over hvilke filer, der tilgås af hvilke IPer. Min tanke var, at har en administrator en historik over access af serverens filer, kan vedkommende både bruge den til hurtigt at se, hvad folk mest henter på serveren. Det ville kunne bruges til at opdage "usædvanlig" adfærd. Hvis der f.eks. kom materiale på serveren, som genererede voldsom trafik (f.eks. pirat kopier), ville det ændrer historikken tilsvarende. Var der desuden mulighed for at se, hvilke IPer der genererede sådan trafik, ville det være nemt at se, om der var tale om trafik over lokalnettet eller ude fra omverdenen.

Problemet ved denne ide var, at jeg ikke umiddelbart kunne se en smart måde at få koblet informationen om filerne med datastrukturen i [1]. Hvad jeg strandede på var, at [1] som tidligere nævnt er baseret på antagelsen om et endeligt univers U, der kan indekseres. Filstrukturen på en webserver er meget sjældent statisk over længere tid, da der konstant ændres og tilføjes hjemmesider, og det matchede ikke datastrukturen særligt godt. Dertil kommer, at [1] opbygger en vektor over

universet, hvor der matches en værdi til hvert element, og selvom jeg legede med tanken om ,hvordan jeg kunne få denne værdi til at sige noget om hvilke IPer, der havde tilgået det givne element, så fandt jeg ikke på nogen nem løsning.

Som jeg allerede har været lidt inde på i afsnit 4.6, endte det med at jeg valgte en simpel løsning: At lade mit univers U være IPerne på Internettet og lade den tilhørende værdi være antallet af gange den pågældende IP optræder i serverens access log. Det gode ved denne løsning er, at den er meget simpel at overskue, og at den falder godt i tråd med den indledende bemærkning i [1], der foreslår, at datastrukturen bruges til at holde statistik over trafikken på en backbone router. Havde jeg lavet implementeringen baseret over trafikken på sådan en router, kunne det f.eks. svare til antallet af gange, de forskellige IPer observeres på et givent netværks interface på routeren. Denne brug matcher perfekt datastrukturens krav om et endeligt univers i form af de $2^3 2$ mulige IP adresser, og kravet om at de tilknyttede værdier ikke må være negative.

Det mindre gode ved anvendelsen er, at det ikke er en helt realistisk måde at logge på. Som jeg skrev om i afsnit 2.3, så er et væsentligt problem ved enhver form for logning rent faktisk at afgøre, hvad der er værd at logge. I en webservers access log, som den jeg har benyttet, er der én indgang pr. fil, en given IP tilgår. Hvis en IP derfor tilgår en hjemmeside, der består af f.eks. 10 filer i form af .html dokumentet og inkluderede billeder, så fungerer det i praksis som 10 forskellige requests af filer fra serveren, og der vil derfor være 10 tilsvarende entries i loggen. Min simple optælling, af antallet af gange IPerne optræder i loggen, er derfor nok ikke det, en administrator ville være interesseret i på en kørende server. Der ville man nok mere være interesseret i f.eks. at registrere antal MB hentet af hver IP, antallet af .html filer hentet etc. Hvad det præcist er, der tælles, har dog ingen betydning for implementation og test af selve datastrukturen fra [1], og derfor synes jeg, at mit simple valg er tilstrækkeligt til mit formål.

5.2 Valget af programmerings sprog

Til implementeringen af datastrukturen valgte jeg at benytte programmerings sproget C# og udviklings værktøjet Visual Studio 2005 fra Microsoft. Jeg benytter normalt en computer med Windows XP og er vant til at bruge Visual Studio 2005 på denne. Valget faldt på C# som programmerings sprog, da jeg har arbejdet med dette i forbindelse med andre opgaver de sidste år på studiet, og derfor er fortrolig med sproget. C# programkode er desuden meget nem at læse for andre, der blot har lidt kendskab til C++, da syntaksen for de to sprog er næsten identisk. C# kan ses som et "højere"niveau sprog end C++, da der er en række ting der håndteres automatisk for programmøren, som ellers ikke bliver taget hånd om i C++. Specielt i forbindelse med garbage collection der foregår automatisk. Hvis man har brugt C++, er der kun nogle få ting, jeg synes, man behøver kende til for at kunne forstå C# kode:

- Garbage collection. Foregår helt automatisk og altså ikke noget programmøren behøver tænke på. Dvs. man skal selvfølgelig stadig være klar over, hvornår garbage collectoren anser f.eks. et objekt for noget, den kan fjerne, men man behøver ikke i samme grad som i C++ huske at deallokere hukommelse hele tiden.

- Ingen pointere. Som udgangspunkt kan man ikke benytte pointere i C#. Dvs. det kan man ikke i "rigtig"C# kode. Hvis man ønsker at benytte pointere, kan man erklære sektioner af sin programkode for værende "unsafe", så compileren alligevel tillader brug af pointere. Hvis man ønsker at bruge pointere i sin kode, er det dog egentlig ikke meningen, at man skal bruge C# som sprog.

> Dét, at der ikke er pointere til rådighed, kommer af, at alle objekter i kald af metoder automatisk sendes som referencer. Det forsøges derfor automatisk i C# at optimere håndteringen af potentielt store instanser af objekter, der sendes rundt i programmet. Det betyder så også, at man skal være klar over, at ændringer i objekter givet som parametre til en metode virkelig også ændrer det pågældende objekt.
>
> Der er undtagelser til dette - f.eks. håndteres de grundliggende strukturer såsom integers, strings etc. anderledes - men som udgangspunkt skal man altså forvente, at et objekt der gives som parameter til en metode, håndteres som en reference fra C++.

En kort introduktion til C# kan evt. læses på Wikipedia[1].

En ekstra grund, til at jeg har valgt C#, er, at konstruktionen af simple visuelle interfaces er meget ligetil. Hvor man tidligere ofte har lavet interfaces i C++ til Windows ved hjælp af Microsoft Foundation Class (MFC) library [2], så introducerede Microsoft sammen med C# og .Net platformen en ny måde at lave interfaces på ved hjælp af "Windows Forms"[3]. Ideen skulle være, at interfaces er simplere at lave med Windows Forms end med MFC, og dette synes jeg er rigtigt for i hvert fald simple interfaces. Eftersom det er minimalt, hvad jeg har brug for ved interfacet til min implementation, er det en ekstra fjer i hatten for C#, at denne del er nem at lave.

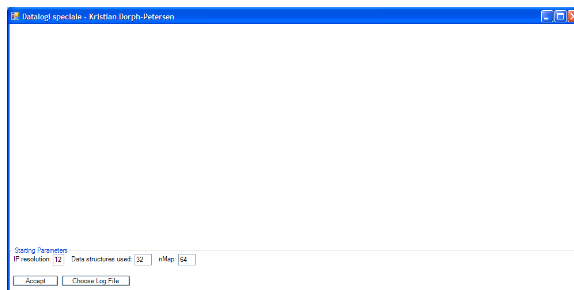
5.3 Fejlhåndtering i programmet

Jeg har ikke forsøgt at lave et "robust"program i den forstand, at enhver type af exception i koden bliver håndteret af programmet. Det vil sige, at hvis brugeren f.eks. prøver at bruge programmet på en fil af et andet format, end de logs programmet er lavet til, så vil der ske uforudsete ting (sandsynligvis crasher programmet). Jeg har taget hånd om nogle af de mere simple ting, såsom hvis brugeren indtaster bogstaver et sted, hvor han skulle angive en parameter i tal, men det er også det hele. I min vejledning til brugen af programmet (bilag A) forklarer jeg, hvad programmet vil acceptere hvor, for at kunne genskabe mine resultater. Man er der udover velkommen til at eksperimentere med programmet, men man skal altså ikke forvente, at programmet prøver at redde en fra forkerte valg.

5.4 Programmets opbygning

Jævnfør udgangspunktet for min implementation har jeg lavet et lille program, der næsten udelukkende består af datastrukturen selv og et interface til visualisering og betjening. Betjeningen består basalt set blot af valget af parametre datastrukturen benyttes med, og så lidt ekstra muligheder til at gøre visualiseringen af datastrukturen mere fleksibel. Min oprindelige ide med programmet var, at man skal kunne simulere den situation, en administrator for webserveren ville benytte datastrukturen i, og det har dannet grundlag for, hvordan jeg har lavet programmet.

Brugen af datastrukturen i praksis består af to dele. Initialiseringen og den efterfølgende brug.



Figur 5.1: Screenshot - Programmet ved start

5.4.1 Initialisering

Som datastrukturen er designet i [1], er det ikke muligt at ændrer på, hvad der sker i datastrukturen, når den først er blevet initialiseret. Problemet er jo netop, at man ikke kan garantere, at man kan få alle de registrerede elementer ud af datastrukturen, og derfor kan man ikke konverterer indholdet, når først kørslen er i gang. Det er derfor ved initialisering, at en administrator skal bestemme, hvad der skal registreres, og hvor nøjagtigt det skal ske. Figur 5.1 viser mit program ved start, når det venter på disse valg af parametre for initialiseringen.

- **Opdeling af IPerne.** Valget af hvad der registreres, er selvfølgelig noget, der normalt er meget vigtigt, for at datastrukturen lagrer data, der senere har en funktionel værdi. Som skrevet tidligere (afsnit 5.1) er valget dog ikke så vigtigt for min vurdering af datastrukturens værdi. Jeg har derfor i mit program simplificeret det til, at man kan vælge en opdeling, af de potentielle IP adresser man kan møde, og så tælle forekomsten af disse i access loggen.

  IPer er, i den nuværende IPv4 standard der benyttes på Internettet, adresser af længden 32 bit. For at gøre IPer mere læselige for mennesker visualiseres IPer normalt som 4 tal mellem 0 og 255 på formen

  xxx.xxx.xxx.xxx

  og det er også på denne måde, IP adresserne optræder i access loggen. Disse 4 tal svarer til de tilsvarende 4 blokke af 8 bits i IP adresserne. Udover at IPerne på denne form er nemmere at læse, så bruges denne opdeling af adresserne i 4 blokke normalt også ved tildelingen af IP adresser til de netværk, der er koblet på Internettet. Det er f.eks. meget normalt, at et mindre netværk får tildelt 256 IP adresser, ved at en myndighed på nettet (som f.eks. RIPE[4] i Europa) fastlægger værdien af de første 3 blokke i de IP adresser, det pågældende netværk må benytte. Den konkrete brug af IP adresser i IP protokollen[5] er et helt område i sig selv, og til min brug er det nok at vide, at ved enhver kontakt til en given webserver er det afsenderes IP, der kan bruges som en form for ID på kontakten.

  Opdelingen af IP adresserne i min implementation foregår ved, at man angiver antallet af n bits i begyndelsen af adresserne, der skal skelnes mellem. Vælger man f.eks. $n = 8$, vil mit program kun bruge de første 8 bits i adresserne fra loggen til at identificere kontakten ud fra. I forhold til datastrukturen fra [1]

[1] http://en.wikipedia.org/wiki/C_Sharp
[2] http://en.wikipedia.org/wiki/Microsoft_Foundation_Classes
[3] http://en.wikipedia.org/wiki/Windows_Forms
[4] http://www.ripe.net/
[5] http://www.ietf.org/rfc/rfc791.txt

betyder det eksempel, at der simuleres en vektor med $2^8 = 256$ indgange. En kontakt fra IPen

$$192.1.1.2$$

vil blive registreret i værdien for $i = 192$ i vektoren sammen med alle andre kontakter fra IPer på formen

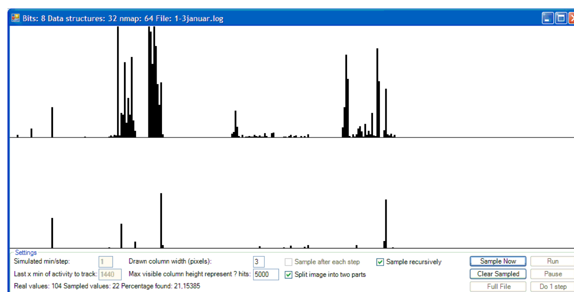
$$192.xxx.xxx.xxx$$

Ønsker man at registrere kontakten fra alle potentielle IPer hver for sig, vil man altså angive $n = 32$. Jeg har valgt denne form for opdeling, da det er en simpel og overskuelig måde, hvormed jeg kan initialisere datastrukturen med forskellige størrelser for universet U.

- **Datastrukturens præcision.** Som angivelse for datastrukturens præcision ved initialiseringen er der to ting, mit program har brug for. Det ene er antallet af parallelle datastrukturer, der skal benyttes, og det andet er værdien for nmap variablen i PCSA algoritmen for DE strukturerne (se afsnit 4.5.2). Som jeg vil komme ind på i kapitel 6, er det mest af alt antallet af parallelle datastrukturer, der er vigtig ved initialiseringen i mit program. nmap og de andre valg der foretages mht. præcisionen af programmet, afhænger mest af mængden af elementer i universet U, som datastrukturen benyttes på. Valget af præcisionen har betydning for datastrukturens pladsforbrug, og har derfor betydning hvis man vil optimere datastrukturen til brug i praksis. For en vurdering af om datastrukturen i det hele taget kan bruges, har pladsforbruget - i hvert fald ikke umiddelbart - ingen indflydelse, og jeg har derfor blot sat disse parametre i koden, så præcision er god nok for min anvendelse af datastrukturen.

Når de ønskede værdier for initialiseringen er valgt er datastrukturen klar til brug.

5.4.2 Kørsel



Figur 5.2: Screenshot - Programmet efter initialisering

Efter at datastrukturen er blevet initialiseret, skifter programmet til simuleringen af praktisk brug (figur 5.2). Ideen er, at i en praktisk anvendelse af datastrukturen, genererer webserveren de sædvanlige logs, og sender disse videre til arkivering, samtidigt med at datastrukturen fodres med de oplysninger, den skal bruge. En administrator, der ønsker at checke den registrerede aktivitet, vil derfor bruge datastrukturen til at lave et "snapshot", af dét serveren har registreret i f.eks. de sidste 5 timer. Dvs. han skal kunne bede programmet om på baggrund af datastrukturens tilstand på det givne tidspunkt at vise et søjlediagram over aktiviteten. Målet

er så, at diagrammet skal vise så meget af den faktiske aktivitet, at administratoren kan nøjes med datastrukturen til at træffe beslutninger ud fra.

Jeg har lavet programmet, så det simuleres, at man har en forbindelse til den observerede webserver, hvorfra der med intervaller på x minutter, kommer opdateringer om aktiviteten i access loggen. Man angiver derfor i programmet, hvor mange simulerede minutter hvert interval skal vare. Da jeg har adgang til en rigtig access log, der strækker sig over 6 måneder, kan programmet rent faktisk lave denne simulering korrekt, ved i hvert interval kun at processere de indgange i loggen der passer til det tilsvarende simulerede tidsinterval.

For at kunne vurdere diagrammerne, der genereres af datastrukturen, har jeg i mit program også implementeret den vektor, som datastrukturen fra [1] sampler. Dét, der holdes styr på, er jo netop en vektor, med i værdier for antallet af gange de enkelte IP intervaller optræder i loggen, og det er derfor meget simpelt (omend pladskrævende) at lade programmet holde styr på denne vektor sideløbende med datastrukturen fra [1]. På denne måde opnår jeg, at jeg hele tiden kan sammenligne de genererede diagrammer med, hvordan de skal se ud, hvis de er perfekte. Mit program er derfor i stand til at vise grafiske repræsentationer af både den reelle aktivitet og diagrammet lavet ud fra datastrukturen, så man løbende kan lave denne sammenligning.

I figur 5.2 vises et screenshot af programmet, hvor der er gereret et diagram ud fra datastrukturen. Øverst er søjlediagrammet, der viser den reelle aktivitet i loggen, nederst er det genererede. Hver søjle repræsenterer et IP interval, og højden af søjlen bestemmes, af antallet af gange IPer fra søjlens interval er observeret i loggen.

5.4.3 Sliding window i loggen

En enkelt detalje i den overordnede opbygning af programmet, jeg synes fortjener seperat omtale, er hvilket udsnit af access loggen fra webserveren, man ønsker at lave diagrammer over. Den mest simple indfaldsvinkel til problemet er blot, at fodre datastrukturen med hver eneste indgang i access loggen efterhånden som de mødes. Dvs. hver gang en IP observeres i loggen, tælles den tilsvarende indgang i i vektoren op med 1, og det er så det. På den måde kommer vektoren altså til at indeholde et billede, af den samlede aktivitet der er observeret hidtil. Det giver desuden en simpel brug af indgangene i access loggen, for når først en indgang er blevet registreret, skal programmet aldrig bruge den igen.

Ulempen ved at køre simuleringen på denne måde er, at i den virkelige verden vil den opbyggede datastruktur hurtigt blive dårlig at bruge, for administratoren der benytter den. Et diagram over den samlede trafik observeret siden serverens start siger ikke noget brugbart om aktiviteten her og nu. Som nævnt tidligere er en potentiel brug af datastrukturen for administratorer, at de genererede histogrammer kan bruges til at spotte usædvanlig trafik på serveren. Kan man f.eks. se en voldsom trafik fra IPer, der ikke normalt kontakter serveren, kan det være tegn på, at man bør kigge nærmere på de ellers arkiverede access logs. Problemet er bare, at fjerner man ikke den usædvanlige trafik fra datastrukturen igen, så kan den optræde i alle fremtidige genererede diagrammer.

Løsningen på dette er i stedet kun at lade datastrukturen indeholde oplysninger om aktiviteten i et begrænset tidsrum. Dvs. at man fastsætter, hvor langt tilbage i tiden man ønsker at huske oplysningerne om aktiviteten på serveren. F.eks. kan det give mening på en kritisk server med meget stor aktivitet, at kun holde styr på hvad der er sket de seneste 5 minutter. En administrator vil kunne bruge diagrammer over sådanne 5 minutters aktivitet til hurtigt at opdage unormal aktivitet. Omvendt vil man på andre servere måske blot være interesseret i at checke aktiviteten for den

seneste uge, hvis man f.eks. kun ønsker at bekræftige, at aktiviteten ser nogenlunde normal ud.

Jeg har set det som en meget væsentlig ting for brugen af datastrukturen, at denne form for "sliding window"over webserverens access log kan lade sig gøre, for at datastrukturen bare kan overvejes som et praktisk redskab. Programmet er derfor lavet med det i tankerne, og ved simuleringerne i programmet kan man derfor angive, hvor mange simulerede minutter man ønsker der skal passerer, før end aktiviteten fra en given IP skal glemmes igen.

Prisen ved at fjerne registreret aktivitet fra datastrukturen igen er så, at der potentielt set kan være uregelmæssig trafik, man normalt ville have reageret på, som aldrig opdages, hvis datastrukturens samplinger uheldigvis ikke viser den del af diagrammet, inden trafikken glemmes igen.

5.5 Kodens opbygning

Ved implementeringen af mit program har jeg forsøgt at opdele programkoden i klart adskilte dele. En god ting ved dette er, at det gør koden nemmere at læse, men hovedårsagen er, at det har gjort det nemmere for mig at udskifte dele undervejs. Specielt ved testningen af om min kode rent faktisk implementerede datastrukturen fra [1] korrekt, var det vigtigt, at jeg nemt kunne bruge simple kodestumper, der gjorde det samme som delene fra datastrukturen. Som udgangspunkt er koden derfor delt op i tre hoveddele: Hovedprogrammet, datastrukturen og visualiseringen af datastrukturen.

5.5.1 Hovedprogrammet

Hovedprogrammet er den del af koden, der står for start og stop af programmet, valg af initialisering, styringen af interfacet etc. Dvs. alt det kode som skal til, for at det "uden om"datastrukturen fungerer. Helt uafhængigt af datastrukturen er det selvfølgelig ikke, da det er denne kode, der initialiserer datastrukturen, og i sidste ende skal sørge for, at brugeren får adgang til resultaterne.

Koden til denne del af programmet er placeret i filen *MainForm.cs*. Den indeholder kun to klasser. Den første klasse er

```
class cProgramStarter
{
    public static void Main()
    {
        Application.EnableVisualStyles();
        Application.DoEvents();
        Application.Run(new cMainForm());
    }
}
```

som er det programkode, der skal til for at starte ethvert form for C# program. Den anden klase er

```
class cMainForm : Form { ... }
```

som er klassen, der styrer det hele. Klassen er nedarvet fra C#s *Form*-klasse, da *cMainForm* også er det visuelle interface for brugeren. En meget stor del af indholdet af *cMainForm*-klassen består derfor af triviel kode, der specificerer hvordan interfacet skal sættes op. Den indeholder dog også interessant kode i de dele, der har direkte at gøre med datastrukturen. Koden i *cMainForm* kan derfor deles op i to dele:

- **Styring af simulationen.** Hovedformålet med *cMainForm* er som sagt at give brugeren mulighed for at styre de simulationer, der foretages med datastrukturen. Visuelt sker det ved, at brugeren kan indtaste parametre i selve

interfacet og trykke på diverse knapper for at fortælle programmet om valgene. I programkoden sker det primært gennem metoderne

```
void btStart_Click(object obj, EventArgs ea) { ... }
void btRun_Click(object sender, EventArgs e) { ... }
void btOneStep_Click(object sender, EventArgs e) { ... }
void RunAnalysis(int MinPrTick, int RemoveAfterMin, int nTicks) { ... }
void btPause_Click(object obj, EventArgs ea) { ... }
void btSampleNow_Click(object sender, EventArgs e) { ... }
```

som alle aktiveres gennem brugerens tryk på interfacets knapper. Der er ikke noget specielt overraskende kode i disse metoder, så jeg nøjes her med en hurtig gennemgang af dem:

- *btStart_Click* kaldes når brugeren har valgt parametrene for initialiseringen af selve datastrukturen. Dvs. har valgt den ønskede opdeling af IPerne, antallet af parallelle datastrukturer der skal anvendes og størrelsen af *nmap* i PSCA algoritmen. I denne metode initialiseres derfor datastrukturen og programmets interface ændres så brugeren får adgang til parametrene for simuleringen på datastrukturen (dvs. skifter til det vist i figur 5.2).
- *btRun_Click* og *btOneStep_Click* bruges begge til at læse data fra den valgte log fil og opdatere datastrukturen. *btRun_Click* kaldes, når brugeren ønsker at lade simuleringen kører indtil enden af log filen, eller til der trykkes "Pause"i interfacet. *btOneStep_Click* kaldes hvis der kun ønskes, at der skal simuleres et enkelt tidsinterval. Dvs. svarende til *btRun_Click* efterfulgt umiddelbart af "Pause".

  Begge metoder foretager simuleringen ved at kalde *RunAnalysis*.
- *RunAnalysis* benyttes af *btRun_Click* og *btOneStep_Click* til at foretage selve simuleringen. Det sker i koden

  ```
  while (Running && NotDone && (nTicks == 0 || i < nTicks))
  {
      NotDone = Analyser.Analyse(MinPrTick, RemoveAfterMin);
      if (cbSample.Checked) { Analyser.SampleData(cbRecursive.Checked); };
      ImagePanel.Invalidate();
      Application.DoEvents();
      lbStatus.Text = "Status: Running! Examined " + Analyser.nAnalysed + "/6078051
                      entries, have met " + Analyser.nErrorEntries + " errors.
                      Simulated time is " + Analyser.SimulatedTime;
      i++;
  };
  ```

  Umiddelbart er dette en meget ligetil *while*-løkke, der kører det antal *nTicks*, der er ønsket, eller hvis *nTicks*= 0 indtil der trykkes "Pause", og *Running* dermed sættes til *false*. *Analyser* objektet er en instans af klassen, der styrer selve datastrukturen, og det er ved kald af denne, at simuleringen foretages, og diverse oplysninger returneres.

  Der er to ting her, der kan virke lidt mærkelige. Den første er de to linier

  ```
  ImagePanel.Invalidate();
  Application.DoEvents();
  ```

  Disse to linier fortæller først Windows, at den del af interfacet, der står for visualiseringen af simulationen, *ImagePanel*, muligvis er ændret, og derfor skal opdateres på skærmen, og derefter at Windows skal gøre dette, inden programmet kører videre. I det hele taget betyder *Application.DoEvents()*, at alle events i programmet (såsom brugerens tryk på knapper etc.) skal udføres, og sikrer dermed, at programmet rent faktisk reagerer på brugerens input, mens simulationen kører.

  Den anden ting er tallet 6078051 i forbindelse med opdateringen af status beskeden på interfacet, om hvor langt simuleringen er kommet. 6078051 er det samlede antal indgange i den fulde log over de ca. 6 måneders

aktivitet på webserveren for www.246.dk. Da programmet jo arbejder på en principielt set uendelig stor data stream, ved programmet aldrig hvor langt det er kommet i behandlingen af log filen. Jeg har derfor selv skrevet dette til i status beskeden, for at man som bruger af mit program, i det mindste ved hvor længe, programmet maksimalt kan fortsætte.

  - *btPause_Click* kaldes, hvis der kører en simulering, og brugeren trykker på "Pause". Simuleringen pauses, og kan senere genoptages ved kald af *btRun_Click* og *btOneStep_Click*.
  - *btSampleNow_Click* kaldes, når simuleringen er pauset, og brugeren beder om at sample datastrukturen for at generere et snapshot af den registrerede aktivitet. Det er altså brugen af denne metode, der svarer til, at en administrator på en webserver ønsker at checke, hvordan aktiviteten ser ud for det valgte tidsinterval.

- **Interface til simulationen.** Den anden del af *cMainForm*s opgave består i, at give brugeren et interface til styring og visualisering af simulationen. Som allerede nævnt er der en stor del triviel programkode i *cMainForm*, hvis formål er at definere placering af knapper, felter til indtastning af simulationens parametre etc. Disse elementer benyttes af metoderne netop beskrevet, til at give brugeren de muligheder han har bruge for.

  Udover den interaktive del af interfacet, skal selve simulationen også vises grafisk. Dvs. når man beder programmet lave et snapshot af aktiviteten registreret i datastrukturen, så skal der helst komme noget frem på skærmen, man kan se på. Dette sker ved, af det meste af programmets visuelle flade, er optaget af objektet *ImagePanel*, der er en instans af klassen *cImagePanel*. Det er dette objekt, der varetager den grafiske visualisering, og *cMainForm* er adskilt fra dette så meget som muligt. Det eneste kode *cMainForm* derfor har, der vedrører *ImagePanel*, er koden, der angiver hvor stor en del af interfacet *ImagePanel* skal bruge, samt

  ```
  ImagePanel.RegisterTrackers(Analyser.NormalTracker, Analyser.SamplerValues);
  ```

  i *btStart_Click*-metoden. Denne linie registrerer de strukturer med dataene, der skal vises grafisk i *ImagePanel*, efter programmet har initialisere hoved datastrukturen. Programkoden for *cImagePanel* beskrives i afsnit 5.5.3.

Hovedprogrammets formål er altså at holde styr på disse to dele. Én ting, som måske kan undre, er, hvorfor jeg ikke har implementeret hovedprogrammet til at benytte flere programtråde. Netop i et program som dette, hvor der under simuleringen bliver lagt en stor belastning på CPUen, er det gode ved f.eks. at lægge simuleringen og interfacet i hver deres tråd, at brugeren ikke vil opleve, at programmet ikke reagerer i perioder. Jeg overvejede det også, men jeg endte med at mene, at det for mit program var tilstrækkeligt, at lade det hele køre i samme tråd, og så som nævnt bruge linien

```
Application.DoEvents();
```

i *while*-løkken for simuleringen til at sikre, at programmet med mellemrum reagerer på brugeren. Eftersom mit program ikke skal bruges til andet end dette speciale, synes jeg ikke det er nødvendigt, at bruge ekstra tid på at få programmet til at blive eksekveret i flere tråde. Det har dog den bivirkning, at i mit program kan der godt gå lidt tid, inden der reageres på brugerens interaktion, hvis der er specielt meget aktivitet registreret i log filen, for det tidsinterval der er ved at blive processeret.

5.5.2 Datastrukturen

Den anden del af programkoden er den del, der tager sig af selve databehandlingen, den implementerede datastruktur fra [1] og den sideløbende vektor med de rigtige værdier som kontrol. Koden for denne del af programmet ligger i filerne *DataTypes.cs*, *StatisticControls.cs* og *Funtions.cs*.

- *DataTypes.cs* er filen, der indeholder koden for den overordnede håndtering af simulationen. I denne fil er koden for, hvordan programmet skal læse webserverens log fil, samt koden for den klasse hovedprogrammet benytter til styringen af hele simulationen. Det er altså koden i denne fil, der ud fra metodekald fra hovedprogrammet, samler selve data behandlingen, og benytter koden i de to andre filer *StatisticControls.cs* og *Funtions.cs* til at foretage simuleringen.

- *StatisticControls.cs* indeholder koden for selve implementeringen af datastrukturen i [1] og for min kontrol struktur.

- *Functions.cs* var ment som filen, der indeholder metoder, der bruges rundt omkring i resten af programmet, og som jeg evt. senere ville lave om. Altså til metoder der implementerer funktionskald, hvor den kaldende kode er ligeglad med hvordan metoden fungerer, bare den får det rigtige resultat tilbage. Når jeg skriver "var ment som", skyldes det, at jeg kun endte ud med at placere koden for programmets hash funktioner i denne fil.

Ingen af de tre filer er særligt store, og man kunne derfor argumentere for, at koden fra alle tre filer skulle være placeret i én og samme fil. Jeg har dog valgt denne opdeling, da koden i de tre filer fungerer som tre separate lag. Koden i *Functions.cs* er uafhængig af de to andre filer, og koden i *StatisticControls.cs* er uafhængig af *DataTypes.cs*.

Størstedelen af koden i de tre filer skulle være ligetil at forstå, når man ved hvad datastrukturen fra [1] handler om, men for fuldstændighedens skyld gennemgår jeg kort hver klasse. De steder i koden jeg synes kræver særlig forklaring, bliver beskrevet i afsnit 5.6.

- *cAnalyseController* i *DataTypes.cs*. Dette er klassen, der bruges af hovedprogrammet til at køre simuleringen. Ideen med klassen er, at den skal stille metoder til rådighed, som starter og pauser simuleringen, håndtere datastrukturerne og i det hele taget sørge for, at hovedprogrammet ikke behøver vide noget, om hvad der sker i koden for simuleringen.

  For at opfylde dette ønske har jeg lavet *cAnalyseController*, så den i sin initialisering modtager parametrene specificeret af brugeren. Det sker i klassens creator-metode med disse linier:

  ```
  public cAnalyseController(string Log, int nBit, int nTrackers, int nmap)
  {
      LogFile = new cLogFileReader(Log);
      NormalTracker = new cNormalTracker(nBit);
      SampCont = new cSamplingController(nBit, nTrackers, nmap);
      SamplerValues = new cSamplerValues(nBit);

      LogEntry = LogFile.GetNextEntry();
      if (LogEntry != null)
          { SimulatedTime = new DateTime(LogEntry.EntryTime.Ticks); };
  }
  ```

  Hvad der er værd, at bemærke her er, at ved skabelsen af et objekt af klassen instantieres *NormalTracker*, vektoren med de rigtige værdier, *SampCont*, en datastruktur der holder styr på *nTrackers* parallelle udgaver af datastrukturen fra [1], og *SamplerValues* som også er en vektor, hvori de samplede værdier

placeres, når der laves et snapshot. Det er disse tre objekter, der bruges til at lagre den data, der genereres under kørslen. De resterende linier sørger for, at log filen åbnes, og er klar til brug.

Metoden *Analyse()* i *cAnalyseController* er hvor det egentlige arbejde laves. Metoden er defineret som

```
public bool Analyse(int MinPrTick, int RemoveTime)
```

og kaldes med henholdsvis *MinPrTick*, der angiver antallet af minutter, der skal behandles i loggen, og *RemoveTime* der angiver hvor mange simulerede minutter, der skal gå, før end allerede behandlede indgange i loggen skal fjernes fra datastrukturen igen. *RemoveTime* er der altså, for at man opnår et sliding window som beskrevet i afsnit 5.4.3. Selve behandlingen af indgangene i loggen foregår i *while*-løkken givet ved:

```
while ((LogEntry != null) && (SimulatedTime > LogEntry.EntryTime))
{
    if (LogEntry.NoError)
    {
        // Force check on any user interactions if 1000 entries has passed
        // with no interface updates
        if (i == 1000)
        {
            Application.DoEvents();
            i = 0;
        };

        if (RemoveTime > 0)
        {
            if ((OldEntry == null) && (Waiting.Count > 0))
                { OldEntry = (cApacheLogEntry)Waiting.Dequeue(); };
            while ((OldEntry != null) && ((OldEntry.EntryTime < LogEntry.EntryTime)))
            {
                NormalTracker.RecordDel(OldEntry.RequesterIP);
                SampCont.RecordDel(OldEntry.RequesterIP);
                if (Waiting.Count > 0)
                    { OldEntry = (cApacheLogEntry)Waiting.Dequeue(); }
                else { OldEntry = null; };
            };
            LogEntry.EntryTime = LogEntry.EntryTime.AddMinutes(RemoveTime);
            Waiting.Enqueue(LogEntry);
        };
        NormalTracker.RecordHit(LogEntry.RequesterIP);
        SampCont.RecordHit(LogEntry.RequesterIP);
    }
    else { nErrorEntries++; };

    nAnalysed++;
    LogEntry = LogFile.GetNextEntry();

    i++;
};
```

Denne *while*-løkke bliver ved med at behandle indgangene fra log filen, ind til den ønskede simulerede tid er passeret, eller til enden af filen er nået. Hver indgang i loggen går igennem følgende skridt:

1. Check om log indgangen er gyldig. I princippet burde alle registrerede kontakter til webserveren i den givne log fil være skrevet på samme måde, men i praksis er det åbenbart ikke tilfældet. I min log fil fra domænet www.246.dk er der nogle enkelte gange, hvor indgange er skrevet med en anden syntaks end den normale. Da der kun er tale om meget få (under 10) i loggen jeg bruger, ignoreres disse indgange i mit program.
2. Check om variablen $i = 1000$ og i så fald kald *Application.DoEvents()*. Som tidligere beskrevet kører programmet kun i én tråd, og dette check sikrer, at ved valg af parametre hvor der er mange indgange i loggen pr. tidsinterval, så checker programmet en gang i mellem for f.eks. resize af programvinduet.

3. Det nok umiddelbart mest mærkelige i løkken er næste skridt, nemlig koden givet ved:

   ```
   if (RemoveTime > 0) { ... }
   ```

   Som det kan ses eksekveres denne blok kode kun, hvis $RemoveTime > 0$, altså hvis brugeren har angivet, at de enkelte indgange i loggen skal fjernes fra datastrukturen efter et antal simulerede minutter. Er $RemoveTime = 0$ bliver de aldrig fjernet. Koden i denne blok håndterer derfor dét, der skal til, hvis data skal kunne pilles ud af datastrukturen igen. Hvad der sker her beskrives i afsnit 5.6.4.

4. Sidste skridt i *while*-løkken er at registrere den behandlede indgang fra loggen i henholdsvis *NormalTracker* og *SampCont*.

Efter endt udførsel af *while*-løkken returnerer metoden til det kaldende hovedprogram med en bool værdi, der indikerer om enden af log filen er nået eller ej.

Den eneste ekstra metode i *cAnalyseController*-klassen er *SampleData()*. Den er givet ved

```
public void SampleData(bool DoRecursively)
{
    SamplerValues.ClearData();
    SampCont.GetSamples(SamplerValues, DoRecursively);
}
```

og sørger blot for, at *SamplerValues* strukturen bliver fyldt op med samplede værdier fra *SampCont*. Dvs. denne metode kaldes, når der skal laves et snapshot.

- *cApacheLogEntry* i *DataTypes.cs*. Jeg har lavet denne klasse, for at kunne håndtere indgangene i log filen så simpelt som muligt, når først de er læst ind. Klassen er defineret som

  ```
  class cApacheLogEntry
  {
      public uint RequesterIP = 0; // Requester address
      public DateTime EntryTime; // Time of request
      public bool NoError = true;

      public cApacheLogEntry(string IP, string Time) { ... }
  }
  ```

  Det eneste jeg bruger fra hver indgang i loggen, er IPen, hvis kontakt er registreret, og tidspunktet det er sket på. I tilfælde af at indgangen i loggen er skrevet med en syntaks programmet ikke forventer, er der desuden et flag *NoError*, som indikerer, om der er noget galt eller ej. Der er en del kode i klassens creator-metode, da jeg skal omdanne IPen og tiden angivet ved tekst i log filen, til numeriske værdier mit program kan bruge, men intet i denne kode er overraskende.

- *cLogFileReader* i *DataTypes.cs*. Denne klasse håndterer selve læsningen fra log filen. Den er defineret som

  ```
  class cLogFileReader
  {
      private FileStream fsLog = null;

      public cLogFileReader(string LogFile) { ... }
      public cApacheLogEntry GetNextEntry() { ... }
  }
  ```

  og koden her er heller ikke særlig spændende. Koden i metoden *GetNextEntry()* er skrevet ud fra syntaksen af log filen og sikrer ved hjælp af *cApacheLogEntry*-klassen, at resten af programmet ikke behøver vide noget om log filen.

Hvis jeg ønskede at udvide programmet til at kunne håndtere log filer med en anden syntaks, er det altså her i *cLogFileReader*, der skal laves ændringer.

- *cNormalTracker* i *StatisticControls.cs*. *cNormalTracker* bruges til at holde styr på min kontrol struktur for simulationen. I bund og grund er det derfor en klasse, der blot skal give et nemt interface til styringen af den underliggende vektor. De væsentlige dele er defineret som

```
class cNormalTracker
{
    public int DetailLevel;
    public uint[] HitTracker;

    public cNormalTracker(int nDetail) { ... }

    public void RecordHit(uint IP) { ... }
    public void RecordDel(uint IP) { ... }
}
```

  HitTracker er vektoren værdierne gemmes i, og *DetailLevel* er antal bits af IPernes adresser, der bruges som opdeling. Det betyder, at *HitTracker* altid vil have en længde på $2^{DetailLevel}$.

  De vigtige metoder i klassen er *RecordHit()* og *RecordDel()*, der henholdsvis registrerer og fjerner aktivitet fra en given IP. Dvs. der lægges enten én til eller trækkes én fra i den indgang i *HitTracker*, der svarer til de første *DetailLevel* bits i IPens adresse.

- *cValueHolder* i *StatisticControls.cs*. Denne klasse indeholder talparret (i, a). Der er intet interessant ved klassen, udover at den er nødt til at være der, for at jeg kan putte sådanne talpar i en C# kø. Den bruges i forbindelse med koden for klassen *cSamplingController*.

- *cSamplingController* i *StatisticControls.cs*. Dette er den første klasse i koden, der har at gøre med den undersøgte datastruktur fra [1]. Klassen er defineret som

```
class cSamplingController
{
    private int nDetailLevel;
    private cSamplingTracker[] Trackers;

    public cSamplingController(int DetailLevel, int nTrackers, int nmap) { ... }
    public void GetSamples(cSamplerValues Values, bool DoRecursively) { ... }
    public void RecordHit(uint IP) { ... }
    public void RecordDel(uint IP) { ... }
}
```

  Klassens formål er at styre et vilkårligt stort antal parallelle datastrukturer fra [1]. Dvs. den tager sig af oprettelsen af dem, søger for at de alle sammen bliver opdateret korrekt og håndtere samplingen af dem. Det meste af koden i klassen er ligetil at forstå, men der er dog nogle enkelte punkter, der fortjener ekstra forklaring.

  cSamplingController-metoden er klassens constructor. Den er defineret som

```
public cSamplingController(int DetailLevel, int nTrackers, int nmap)
{
    nDetailLevel = DetailLevel;
    Trackers = new cSamplingTracker[nTrackers];
    Random Randomizer = new Random();

    for (int i = 0; i < Trackers.Length; i++)
    {
        Trackers[i] = new cSamplingTracker(DetailLevel, Randomizer, nmap);
    };
}
```

I constructoren bliver der oprettet et antal datastrukturer angivet ved variablen *nTrackers*. Dét, der er værd at lægge mærke til her, er, at jeg i forbindelse med oprettelsen, laver en C# pseudo tilfældigheds generator i form af *Randomizer* i koden. Den sendes med til hver enkelt constructor for datastrukturerne, og indgår i dem (som senere beskrevet) som en vigtig del af deres oprettelse. Ifølge dokumentationen til C# instantieres et sådan *Random* objekt ud fra computerens tid, og man kan efterfølgende bruge metoden *Random.Next()* for at få pseudo tilfældige tal. At jeg kun laver ét random-objekt, og bruge det alle steder, i stedet for at lave en for hver datastruktur sparer selvfølgelig resourcer i computeren. Mere vigtigt er det dog, at det også sikrer, at jeg ikke initialiserer to *Random* objekter så hurtigt efter hinanden, at de bliver initialiseret med den samme computer tid. Hele ideen er jo, at datastrukturerne skal være initialiseret forskelligt, og blot risikoen - omend den er lille - for at de skulle blive initialiseret ens, ville være en dårlig ting.

GetSamples-metoden kaldes, når der skal samples værdier fra datastrukturerne. I koden for denne metode, har jeg lavet nogle modifikationer til måden det gøres på i [1], og jeg beskriver derfor dette i detaljer i afsnit 5.6.3.

RecordHit- og *RecordDel*-metoderne kaldes af det overliggende program, når der henholdvis skal indsættes og fjernes værdier i de underliggende datastrukturer. Disse to metoders opgave er derfor at sørge for, at det sker i alle datastrukturerne.

- *cSamplingTracker* i *StatisticControls.cs*. Vi er nu nået til klassen, der repræsenterer datastrukturen fra [1]. Klassen er defineret som

```
class cSamplingTracker
{
    private int DetailLevel;
    private cUniqueElement[] UEs;
    private cDestinctElements DE;
    private cHash[] Hashes;

    public cSamplingTracker(int nDetailLevel, Random rSeed, int nmap) { ... }
    public void Update(int i, int a) { ... }
    public bool Sample(out int i, out int a) { ... }
}
```

Som udgangspunkt følger klassen de overordnede retningslinier beskrevet i kapitel 4. Der er dog en del udkommenteret kode i klassen, både fordi jeg har efterladt kode til evt. test, og fordi det er i denne klasse, jeg har foretaget ændringerne i UE strukturen beskrevet i [2]. Denne ændring betød desuden, at der kun var brug for én DE struktur i klassen.

cSamplingTracker er klassen constructor, og er givet ved:

```
public cSamplingTracker(int nDetailLevel, Random rSeed, int nmap)
{
    DetailLevel = nDetailLevel + 1;
    UEs = new cUniqueElement[DetailLevel];
    DE = new cDestinctElements(rSeed, nmap);
    Hashes = new cHash[DetailLevel];

    for (int i = 0; i < DetailLevel; i++)
    {
        UEs[i] = new cUniqueElement();
        Hashes[i] = new cHash(rSeed.Next(), i);
    };
}
```

De forskellige underliggende datastrukturer (UEerne, DEen og hash funktionerne) initialiseres naturligvis i denne kodeblok. Der er ikke nogen overraskelser, men det er værd at bemærke brugen af objektet *rSeed*. Som jeg beskrev ved klassen *cSamplingController*, får hver constructor af *cSamplingTracker* det samme objekt med pseudo tilfældigheds generatoren. Det er i linien

```
Hashes[i] = new cHash(rSeed.Next(), i);
```

at dette objekt bruges til at sikre, at hver hash funktion initialiseres uafhængigt af de andre.

Update-metoden er givet ved

```
public void Update(int i, int a)
{
    for (int j = 0; j < DetailLevel; j++)
    {
        if (Hashes[j].GetHashValue(i) == 0)
        {
            UEs[j].Update(i, a);
            //DEs[j].Update(i, a);
        };
    };
    DE.Update(i, a);
}
```

og følger direkte anvisningerne beskrevet i kapitel 4. Ændringen i min oprindelige implementation som følge af [2] er blot, at jeg har kunnet droppe linien med opdateringerne af DEerne.

På samme måde fulgte *Sample*-metoden oprindeligt også det beskrevne i kapitel 4. Efter jeg fik kendskab til [2], og ændrede koden til den smartere UE struktur, følger den ikke længere helt anvisningerne. Koden for den er

```
public bool Sample(out int i, out int a)
{
    int nElements = DE.Report();
    if (nElements > 0)
    {
        int j = Convert.ToInt32(Math.Ceiling(Math.Log(nElements, 2)));

        // Original code
        /*
        if (DEs[j].Report() == 1)
        {
            UEs[j].Report(out i, out a);
            return true;
        }
        else
        {
            i = 0;
            a = 0;
            return false;
        };
        */

        // Revised addition
        if (((UEs[j].c * UEs[j].c2) - (UEs[j].C*UEs[j].C)) != 0 || UEs[j].c == 0)
        {
            i = 0;
            a = 0;
            return false;
        }
        else
        {
            UEs[j].Report(out i, out a);
            return true;
        };
    }
    else
    {
        i = 0;
        a = 0;
        return false;
    };
}
```

Den udkommenterede kodeblok er min oprindelige kode for Sample-algoritmen beskrevet i [1]. Forskellen i måden Sample() er lavet på i [1] og [2] er, at i [1] sker checket på, om man kan sample UE strukturen (dvs. om den kun indeholder ét element) her i *Sample*-metoden, hvorimod i [2] sker det nede i

UEen selv, ved at den kan returnerer FAIL. Den simpleste måde for mig at ændre min kode på, var derfor blot at ændre checket af UE strukturen, fra at bruge DEerne til at checke *c*-variablerne i UEen direkte. På denne måde er det blot et spørgsmål om at ændre udkommenteringen for at gå tilbage til at bruge DEerne.

- *cUniqueElement* i *StatisticControls.cs*. Næste klasse i filen er *cUniqueElement*, som implementerer UE datastrukturen. Dens kode er

```
class cUniqueElement
{
    public int c = 0;
    public int C = 0;
    // Revised addition
    public int c2 = 0;

    public void Update(int i, int a)
    {
        c = c + a;
        C = C + (a * i);
        // Revised addition
        c2 = c2 + (a * i * i);
    }

    public void Report(out int i, out int v) { ... }
}
```

  og er helt i tråd med [1]. Pga. måden jeg har ændret checket i *Sample*-metoden i *cSamplingTracker*, betyder opdateringen af UEen i [2] kun, at jeg har behøvet at tilføje de to linier i koden for $c2$ variablen. På denne måde behøver man altså kun ændre den kaldende kode, hvis man ønsker at gå tilbage til at følge anvisningerne i [1].

- *cDestinctElemente* i *StatisticControls.cs*. Denne metode implementere DE strukturen givet ved PCSA algoritmen i [13], da det var den artikel, der blev refereret til i [1]. Dens kode er

```
class cDestinctElements
{
    private ulong nmap;
    private int MaxLength;
    private double phi = 0.77351d;
    private cHash Hasher;

    private int[,] Maps;

    public cDestinctElements(Random Seeder, int SetnMap) { ... }
    public void Update(int i, int a) { ... }
    public int Report() { ... }
    private int p(ulong Value) { ... }
```

  og følger altså som udgangspunkt beskrivelsen af PCSA i afsnit 4.5.3. I to henseender har jeg dog været nødt til at ændre/udvide algoritmen, for at få den til at passe til min brug. Nemlig for at give algoritmen mulighed for at fjerne elementer igen og for at kunne håndtere estimering af et lille antal forskellige elementer. Hvad jeg har gjort i forbindelse med disse to ting, er beskrevet i detaljer i afsnit 5.6.4 og 5.6.5.

  En mindre ting i *cDestinctElements*-klassen, der er værd at lægge mærke til, er, at som i *cSamplingTracker* bruges der også her en hash funktion i form af *cHash*-klassen. Det er derfor *cDestinctElements* i sin constructor også får objektet *Seeder* af klassen *Random*, der kan generere pseudo tilfældige tal, så hash funktionen kan initialiseres unikt.

  p-metoden i klassen finder - som beskrevet i afsnit 4.5.3 - den mindst betydende 1-bit i et givent tal. Den er implementeret ved

```
private int p(ulong Value)
{
    BitArray BA = GlobalFunctions.GetBitsFromValue(MaxLength, Value);

    for (int i = 0; i < BA.Length; i++)
    {
        if (BA[BA.Length - 1 - i])
        {
            return i;
        };
    };

    // Should never reach this
    return MaxLength;
}
```

Den bruger en metode *GlobalFunctions.GetBitsFromValue()* fra *Functions.cs*, som splitter det givne tal op i et array af bits, så det er nemt at finde den rigtige bit. Det lidt mærkelige kald af denne metode med variablen *MaxLength* kommer af, at jeg under implementeringen i en periode troede, at jeg skulle finde den mest betydende 1-bit, og ikke vidste hvor lange integers i C#, jeg ville ende med at bruge. I koden ovenfor er *ulong* f.eks. en 64-bits unsigned repræsentation af tal, og da jeg i hash funktionerne i DEerne kun genererer 32-bits unsigned tal, vil kode, der skal finde indeks for den mest betydende 1-bit (talt fra den mest betydende side), altid finde et indeks, der er 32 for stort. Jeg lavede derfor min kode, så jeg kunne angive, hvor mange bits af *ulong* repræsentationen jeg var interesseret i at få placeret i det viste *BitArray*.

Denne udspecificering af antallet af bits er overflødig, når vi skal finde mindst betydende 1-bit, der jo altid har samme indeks (talt fra mindst betydende side), uanset hvor mange bits der bruges på repræsentationen. Det er dog derfor, funktionskaldet ser lidt spøjst ud.

- *cHash* i *StatisticControls.cs*. Denne klasses opgave er, at stille hash funktioner til rådighed for datastrukturen. Den er implementeret med

```
class cHash
{
    private int SeedValue;
    private int Bits;

    public cHash(int Seed, int nBits)
    {
        SeedValue = Seed;
        Bits = nBits;
    }

    public ulong GetHashValue(int Value)
    {
        return HashFunction.GetHash(SeedValue, Bits, Value);
    }
}
```

hvor selve den kode, der foretager hashningen, er placeret i *Functions.cs* (bliver beskrevet i afsnit 5.6.2). Formålet med denne klasse er, at jeg andre steder i koden kan lave objekter, der repræsenterer hash funktionerne, og som indeholder seedet for den givne hash funktion.

Variablen *Bits* definerer hvor mange bits ,hash funktionen skal sende givne værdier over i. Datastrukturen i [1] benytter nemlig én hash funktion pr. oprettet UE struktur, som hasher de givne værdier over i større og større tal (se afsnit 4.4.1). Ved oprettelsen af et *cHash* objekt definerer man derfor størrelsen i bits på tallene, der skal returneres af hash funktionen.

- *cSamplerValues* i *StatisticControls.cs*. Denne klasse bruges til at gemme de samplede værdier fra datastrukturen, så resten af programmet (især den grafiske del) kan få adgang til resultatet af samplingen. Klassen er defineret som

```
class cSamplerValues
{
    public int DetailLevel;
    public uint[] HitTracker;

    public cSamplerValues(int nDetail) { ... }
    public void SetValue(int i, int a) { ... }
    public void ClearData() { ... }
}
```

På samme måde som *cNormalTracker* indeholder denne klasse en vektor med et antal indgange, der stemmer overens med opdelingen af IP adresserne. Der er intet speciel kode i denne klasse, der er blot de nødvendige metoder til at lave klassen, sætte indgangene til de ønskede værdier, når der samples, og til at nulstille alle indgange i vektoren.

- *cGlobalValues* i *Functions.cs*. Jeg lavede oprindeligt denne klasse til at indeholde de globale funktioner, jeg måtte få brug for rundt omkring i programmet. I C# er sådanne funktioner nemlig nødt til at blive lave som *static* metoder i en klasse. Det endte dog med, at jeg ikke havde meget at placere her:

```
class cGlobalFunctions
{
    public static BitArray GetBitsFromValue(int nBits, ulong Value) { ... }
}
```

  Som allerede beskrevet ved *p*-metoden i *cDestinctElements*-klassen, så tager denne metode et unsigned 64-bit tal, og returnerer de *nBits* mindst betydende bits i et *BitArray*.

- *cHashFunction* i *Functions.cs*. Den anden klasse i *Functions.cs* har at gøre med, hvordan mine hash funktioner fungerer. Klassen er defineret som

```
class cHashFunction
{
    public static ulong GetHash(int Seed, int nBits, int Value) { ... }
    private static BitArray GetBitHash(int Seed, int nBits, int Value) { ... }
    private static ulong ConvertBHtoULong(BitArray Bits) { ... }
}
```

  Når en værdi skal hashes, fungerer det ved, at *GetHash* kaldes med et seed for funktionen, hvor mange bits der skal hashes til, og værdien der skal hashes. Omend koden i disse metoder ikke er særlig omfattende eller kompleks, gør betydningen af hash funktioner for datastrukturen i [1], at jeg beskriver hvordan hash funktionen fungerer i afsnit 5.6.2.

  Det eneste, der er specielt værd at bemærke her, er i koden

```
public static ulong GetHash(int Seed, int nBits, int Value)
{
    if (nBits == 0) { return 0; }
    else { return ConvertBHtoULong(GetBitHash(Seed, nBits, Value)); };
}
```

  Hvis metoden kaldes med *nBits* = 0, dvs. at der altså skal hashes til et tal med en længde på 0 bits, returnerer metoden værdien 0. Jeg har lavet dette, da der for én af hash funktionerne oprettet i datastrukturen gælder, at den skal sende alle værdierne den hasher over i 0. Den vil netop kalde metoden med *nBits* sat til 0.

Dette var en gennemgang af klasserne i de tre filer med koden for håndteringen af datastrukturerne i programmet. Brugen af koden i disse filer håndteres altså af hovedprogrammet, som initialiserer klasserne og kalder metoderne. Koden i disse filer foretager sig altså ikke noget på ”egen hånd”.

5.5.3 Visualiseringen

Den sidste del af programmets kode, ligger i filen *DisplayPanel.cs*, og har at gøre med den grafiske præsentering af simulationen, når programmet kører. Filen indeholder kun én klasse *cImagePanel*, som er en nedarving af C# klassen *Panel*.

Der er flere måder, man kan lave grafik i C# på, men den måde, jeg har valgt (og som jeg finder simplest), er, at man laver en ny klasse baseret på *Panel*-klassen og så implementerer metoden

```
protected override void OnPaint(PaintEventArgs pea) { ... }
```

For alle visuelle objekter, der oprettes i et C# brugerinterface, er der en *Paint*-metode, som kaldes hver gang Windows ønsker, at det pågældende objekt tegnes på skærmen. I praksis sker det hver gang, der bliver kaldt metoden *Invalidate()* på objektet, hvad enten det er af Windows selv eller i programmet.

Alle visuelle objekter i C# har altså denne *OnPaint()*-metode, men jeg har valgt at nedarve min klasse fra *Panel*-klassen, da den netop repræsenterer et simpelt rektangel i brugerinterfacet. For at bruge en sådan nedarvet klasse til at lave grafik med, er det eneste der er nødvendigt ,at hovedprogrammet placerer en instans af den nye klasse et sted på interfacet, og så ellers sørger for, at der bliver kaldt *Invalidate()* på objektet, når man ønsker billedet skal opdateres. Resten håndteres i den nedarvede klasse.

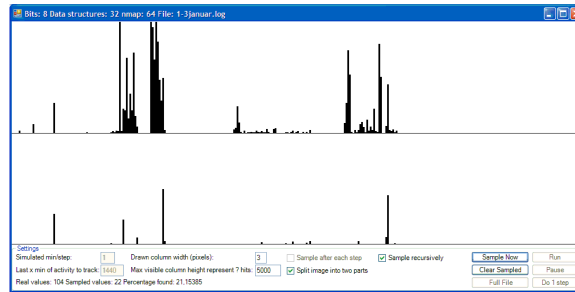
Min *cImagePanel* klasse er defineret som:

```
class cImagePanel : Panel
{
    // Data structures with the values to draw
    private cNormalTracker NormalTracker = null;
    private cSamplerValues SamplerValues = null;

    // Variables controlling the drawing
    private int ColumnWidth = 1;
    private uint FirstTrackShown = 0;
    private uint nTracksShown;
    private int nColumnsShown;
    private int nTracksPrColumn;
    private int nHitsYaxisUp = 1000;
    private bool DrawTwo;

    public cImagePanel() { ... }
    public void RegisterTrackers(cNormalTracker nTracker, cSamplerValues nSampler) { ... }
    public void RegisterVariables(int nColWidth, int YHitsHeight, bool DrawAsTwo) { ... }
    protected override void OnPaint(PaintEventArgs pea) { ... }
    public uint GetColumnFromXY(int x, int y) { ... }
    public string GetColumnData(uint Col) { ... }
    public void SetZoom(uint StartCol, uint EndCol) { ... }
    public void UnZoom() { ... }
}
```

Den første metode, *RegisterTrackers*, bruges til at registrere dataen, der skal tegnes i panelet. Grunden til, at dette ikke sker i klassens constructor, er, at instansen af *cImagePanel* der skal tegnes, oprettes ved programstart, når brugerfladen vises. Det er dog først efter, at brugeren har foretaget de indledende valg med opdelingen af IP adresser etc., at de underliggende datastrukturer kan blive oprettet og derefter registreret i *cImagePanel* objektet.

Den næste metode er *RegisterVariables*, og bruges til at registrere brugerens valg af parametre for hvordan simuleringen skal tegnes. Der er ikke så mange parametre brugeren kan stille på, men de der er, kan sættes både før simulationen startes og undervejs. Man kan som bruger vælge maks værdien for den viste Y-akse, man kan vælge bredden af de tegnede søjler i pixels, og man kan angive om værdierne i de to datastrukturer (med henholdsvis de rigtige og de samplede værdier) skal tegnes på samme x-akse eller på hver sin.

Den tredje metode er *OnPaint*, som indeholder alt koden for, hvordan grafikken skal tegnes. Det er her, hovedvægten af koden for denne klasse er placeret, men da



Figur 5.3: Screenshot af grafikken

det ikke er målet for dette speciale at undersøge, hvordan man laver grafik C#, vil jeg ikke komme nærmere ind på det meste af dette kode. Det eneste i denne metode, jeg synes fortjener lidt omtale, er, hvordan jeg har valgt at vise diagrammer med potentielt 2^{32} søjler på skærme, der i hvert fald ikke har så mange pixels i bredden til rådighed.

Når brugeren vælger en opdeling af IP adresserne, svarer det altså til, at der laves en vektor ud fra denne opdeling. Vælger brugeren f.eks. en opdeling baseret på de første 16 bit i adresserne, giver det en vektor med $2^{16} = 65536$ indgange. Hvis jeg vil vise et søjlediagram over værdierne i disse indgange, betyder det, at jeg i princippet skal bruge lige så mange søjler som indgange. Da det ikke kan lade sig gøre i praksis, har jeg løst det ved, at der i *OnPaint*-metoden løbende udregnes, hvor mange søjler, der er plads til, ud fra den af brugeren valgte bredde på søjlerne. Jeg beregner så

$$\frac{\text{Ønsket antal søjler}}{\text{Muligt antal søjler}}$$

og runder dette resultat op, for at få antallet af indgange i vektoren jeg skal vise i hver søjle på skærmen. Dvs. hver søjle på skærmen kommer derfor til at bestå af summen af en række indgange i vektoren. På denne måde vises den generelle aktivitet over IP adresserne fra 0.0.0.0 som begyndelsen i venstre side af x-aksen til 255.255.255.255 som enden i højre side.

At koden til disse beregninger ligger i *OnPaint* metoden betyder, at disse ting beregnes, hver gang billedet skal tegnes. Umiddelbart er det ikke nødvendigt, da man det meste af tiden ikke ændrer på størrelsen af billedet, men det er så få ekstra beregninger, at det er en simpel måde, til at sikre værdierne altid er rigtige, hvis størrelsen af interfacet ændres.

De resterende metoder i klassen *GetColumnFromXY()*, *GetColumnData()*, *SetZoom()* og *UnZoom()* var i starten af implementeringen tiltænkt som værende metoder, der skulle gøre det muligt at bruge den grafiske repræsentation til f.eks. at zoome ind på interessante områder og få oplysninger om aktiviteten. Efterhånden som implementeringen skred frem, gled disse "for sjov"metoder dog i baggrunden, og de bruges derfor ikke af programmet i øjeblikket. Jeg har dog ladet dem blive i koden, da de illustrerer, at med de underliggende data på plads, er det kun fantasien der sætter grænser for, hvordan dataen kan blive præsenteret på en smart og brugbar måde for brugeren. Den slags overvejelser mht. interfacet har jeg dog valgt ikke at fokusere på. På samme måde som jeg heller ikke er kommet ind på hvilke IP områder, man burde overvåge, typen af indgange i loggen der er interessante etc. Alt dette er godt nok noget, der er interessant, hvis datastrukturen fra [1] skal bruges i praksis - det afhænger dog stadig af, om datastrukturen rent faktisk er brugbar.

5.6 Kode af speciel interesse

I dette afsnit beskriver jeg de steder i programkoden, hvor jeg synes der foregår noget, der kræver en uddybende forklaring. Det drejer sig især om steder, hvor jeg har mødt problemer med implementeringen af datastrukturen, og som jeg ikke synes har været beskrevet i [1] eller [2]. Dvs. steder hvor jeg har været nødsaget til at ændre og/eller udvide koden i datastrukturen for at kunne lave en brugbar implementation.

5.6.1 Sliding window i datastrukturen

Som tidligere beskrevet er det metoden *cAnalyseController.Analyse()*, som i filen *DataTypes.cs* styrer læsningen af log filen og kald af datastrukturerne, så dataen kan blive registreret. I *while*-løkken der gør dette, er der dog en blok kode, som ved første øjekast, kan virke underlig og ret uoverskuelig. Nemlig:

```
if (RemoveTime > 0) {
    if ((OldEntry == null) && (Waiting.Count > 0))
    { OldEntry = (cApacheLogEntry)Waiting.Dequeue(); };
    while ((OldEntry != null) && ((OldEntry.EntryTime < LogEntry.EntryTime)))
    {
        NormalTracker.RecordDel(OldEntry.RequesterIP);
        SampCont.RecordDel(OldEntry.RequesterIP);
        if (Waiting.Count > 0)
            { OldEntry = (cApacheLogEntry)Waiting.Dequeue(); }
        else { OldEntry = null; };
    };
    LogEntry.EntryTime = LogEntry.EntryTime.AddMinutes(RemoveTime);
    Waiting.Enqueue(LogEntry);
};
```

Oprindeligt lavede jeg programmet, så det bare læste fra log filen fra en ende af, og registrerede alt den sete IP aktivitet i datastrukturerne. Som nævnt i afsnit 5.4.3 fandt jeg dog ud af, at det ville være en del mere realistisk, at kunne angive i programmet at datastrukturerne kun skulle indeholde aktiviteten for en specificeret periode. F.eks. de seneste 2 simulerede dage. Jeg tilføjede derfor parameteren *RemoveTime* til metoden, for at man kan angive, hvor mange minutter det registrerede skal huskes og tilføjede desuden ovenstående kodeblok til *while*-løkken. Blokken kaldes kun, hvis $RemoveTime > 0$, da valget $RemoveTime = 0$ bruges til at angive, at programmet aldrig skal smide registreret data væk. Dvs. at loggen skal læses uden brug af et sliding window.

Grunden, til at den blok kode ser ret voldsom ud, skyldes, at for at jeg kan fjerne indgangene fra datastrukturen på et senere tidspunkt, er jeg nødt til at gemme dem et eller andet sted ind til da. Programmet bliver jo præsenteret for loggens elementer som en data stream, hvor de enkelte elementer kun mødes én gang. I en virkelig brug af datastrukturen vil man derfor nok gemme disse tidligere sete indgange fra loggen i en fil på serveren. Hvis man f.eks. ønsker at have et sliding window på 48 timer, skal disse data gemmes i 48 timer. Det vil derfor både være spild af hukommelse, og et unødigt problem ved server nedbrud, at gemme dem i hukommelsen.

I mit simple program til simuleringen hvor den simulerede tid afvikles en hel del hurtigere end "virkelig" tid, er der dog ingen grund til at gemme dem i en fil. Jeg har derfor oprettet en almindelig kø-struktur (en del af C# sproget i form af *Queue*-strukturen), der altså kører efter "først ind, først ud" princippet. Ovenstående kodeblok sørger derfor for, at de undersøgte indgange fra loggen placeres bagerst i køen, og at elementerne i starten af køen bliver sammenlignet med den simulerede tid og så evt. fjernet fra datastrukturen.

Denne måde, jeg har valgt at håndtere lagringen på, betyder, at man skal være opmærksom på, at afhængig af brugerens valg af *RemoveTime*, så kan der komme til at ligge en hel del data i hukommelsen, hvis de skal gemmes i lang tid. Det kan

sløve programmet, men burde ikke have mere betydning end det. Efterhånden som elementerne fjernes fra datastrukturen igen, slettes de, og hukommelsen frigives.

I min egen brug af programmet, til f.eks. kørslerne der ligger til grund for kapitel 6, har det dog ikke haft den store betydning. Da jeg lagde mig fast på den størrelse af loggen, jeg var interesseret i - dvs. størrelsen af mit sliding window - kopierede jeg blot en tilsvarende størrelse ud fra den samlede log fil, og brugte denne som grundlag. Hvis man ønsker et sliding window på f.eks. 48 timer, og man kun har en log fil for aktiviteten i 48 timer, kan man kører analysen af denne fil med $RemoveTime = 0$.

5.6.2 Hash funktionen

Som beskrevet i specielt kapitel 4 spiller hash funktionerne i datastrukturen fra [1] en stor rolle i at sikre, at de matematiske egenskaber er overholdt. Som det også kan ses på selve måden, datastrukturen er lavet på, så er det initialiseringerne af hash funktionerne, der er det eneste non-deterministiske i hele implementationen.

I både [1] og [2] bruges der tid på at argumentere for, hvorfor man kan nøjes med pseudo tilfældigheds generatorer i stedet for perfekt tilfældighed. Dette hænger naturligvis sammen med, at det netop er en computer, der skal stå for genereringen af de tilfældige tal, og det derfor ikke er muligt at have perfekt tilfældighed. Specielt i [2] tages der hånd om dette, ved at man lader DE strukturen være baseret på artiklen [14], som netop gør noget ud af, at den også kan nøjes med en pseudo tilfældigheds generator.

Det korte af det lange er, at der kan bruges meget tid på at vælge og implementere en sådan pseudo tilfældigheds generator, og så derefter gå ind i matematikken for den og checke at den opfylder kravene. Oprindeligt kiggede jeg på Internettet efter sådanne generatorer, som allerede var implementeret og dokumenteret, for at kunne klare denne del af implementationen så enkelt som muligt. Jeg fandt dog ikke nogen, jeg umiddelbart kunne bruge, da dem jeg kiggede på, alle faldt på et specifikt krav: At jeg skulle kunne initialisere et vilkårligt antal hash funktioner forskelligt.

Grunden, til at dette ikke er et trivielt krav, er, at noget af det hash funktioner oftest bruges til - i hvert fald i forbindelse med Internettet - er som en del af sikkerheds løsninger, der har at gøre med kryptering. En af de mere kendte hash funktioner er f.eks. MD5[6], som netop er en funktion, der deterministisk sender en given input streng med vilkårlig størrelse over i en 32 tegns hex-værdi. MD5 kan f.eks. bruges til at kryptere brugeres kodeord på et givent computersystem, så det er de hashede repræsentationer af koderne, der lagres i stedet for koderne selv. På denne måde kan folk der opnår uønsket adgang til de lagrede kodeord ikke læse koderne direkte men kun de hashede værdier.

Sådanne hash funktioner er en videnskab i sig selv, men hele pointen med dem er netop, at de hasher de givne værdier på en kendt og deterministisk måde. Hvorfor? Fordi formålet med denne type hash funktioner netop er, at man skal hashe på samme måde alle steder den givne funktion bruges, så man kan bruge de givne resultater i verifications øjemed. Funktionerne er derfor ikke konstrueret med tilfældig initialisering i tankerne.

Da de konkrete forslag (med udførlige beskrivelser af implementeringen) til hash funktioner jeg umiddelbart kunne finde på Internettet, altså omhandlede hash funktioner af samme type som MD5, fandt jeg i stedet på en helt anden måde at løse problemet på:

For at gøre det så nemt som muligt at ændre i måden hash funktionerne fungerer på, har jeg lavet klassen *cHash* i *StatisticControls.cs*, hvis instanser repræsenterer unikt initialiserede hash funktioner. Som allerede beskrevet i afsnit 5.5.2 er

[6]http://www.faqs.org/rfcs/rfc1321 og http://en.wikipedia.org/wiki/MD5

denne klasses formål blot at holde styr på seedet brugt til initialiseringen af funktionen, og så hvor mange bits givne værdier skal hashes over i. Ved kald af metoden *cHash.GetHashValue()* er det i sidste ende metoden *cHashFuntion.BitArray* i *Functions.cs*, man havner i. Det er nemlig den metode, der repræsenterer den egentlige hashing, der foretages i min implementation. Koden for metoden er:

```
private static BitArray GetBitHash(int Seed, int nBits, int Value)
{
    int tmpValue = Seed + Value;
    if (tmpValue < 0) { tmpValue += int.MaxValue; };

    BitArray Result = new BitArray(nBits);
    Random Randomizer = new Random(tmpValue);

    for (int i = 0; i < nBits; i++)
    {
        if (Randomizer.Next(2) == 1) { Result[i] = true; }
        else { Result[i] = false; };
    };

    return Result;
}
```

Udgangspunktet for dette kode ligger i, at jeg bruger C#s *Random* klasse til at foretage hashingen med. I dokumentationen for C# i "MSDN Library"der følger med Visual Studio 2005, er klassen *Random* beskrevet som: "Represents a pseudo-random number generator, a device that produces a sequence of numbers that meet certain statistical requirements for randomness."Der står ikke mere, og i bund og grund aner jeg derfor egentlig ikke hvor god en pseudo tilfældigheds generator, den egentlig implementerer. Instanser af *Random* skal dog initialiseres af et seed, og da det netop er hvad jeg har brug, har jeg valgt at antage, at tilfældigheden af generatoren er god nok til at opfylde kravene i [1].

Da *Random* egentlig ikke er tiltænkt som udgangspunkt for hash funktioner, har jeg været nødt til at lave et par "hacks"i min implementation, for at få det til at fungere:

- Determinisme. For at have en brugbar hash funktion er det nødvendigt, at den - efter initialisering - hasher alle værdier på samme måde. Dvs. får hash funktionen værdien A to gange, skal den hashe til samme værdi begge gange. *Random*-klassen er bygget omkring ideen, at den initialiseres med et tilfældigt seed, og så returnerer pseudo tilfældige tal ved gentagne kald af metoden *Next()*. Initialiseres to forskellige instanser af *Random* med samme seed, vil de derfor også returnere de samme tal ved efterfølgende kald af *Next()*.

  Jeg udnytter denne egenskab ved netop at initialisere hash funktionerne i min implementation med et seed, som jeg husker gennem kørslen af programmet. I linierne

  ```
  int tmpValue = Seed + Value;
  if (tmpValue < 0) { tmpValue += int.MaxValue; };

  BitArray Result = new BitArray(nBits);
  Random Randomizer = new Random(tmpValue);
  ```

  bruges dette seed sammen med den givne værdi *Value* til at initialisere en instans af *Random*. Kald af metoden med samme seed og værdi vil derfor altid medføre den samme initialisering af *Randomizer*-objektet. Linien

  ```
  if (tmpValue < 0) { tmpValue += int.MaxValue; };
  ```

  er der blot for at sikre, at initialisering sker med et positivt tal, i tilfælde af at $Seed + Value$ resulterer i overløb.

- Hashing til et vilkårligt antal bits. *Random* fungerer ved, at kald af metoden *Next(int x)* returnerer ikke-negative heltal mindre end x. Da x angives ved en

32 bit signed integer, betyder det, at der maksimalt kan returneres værdier af længder på 31 bit. Jeg ønskede ikke at have en sådan begrænsning i min implementation, og jeg bruger derfor *Random* til at generere vilkårligt lange arrays af true og false værdier. I koden sker det i

```
for (int i = 0; i < nBits; i++)
{
    if (Randomizer.Next(2) == 1) { Result[i] = true; }
    else { Result[i] = false; };
};
```

hvor *Randomizer.Next(2)* altså returnerer enten 0 eller 1 ved hvert kald. På denne måde kan der i princippet hashes over i vilkårligt lange værdier. I min implementation er grænsen dog i praksis 64 bit, da jeg i metoden *cHashFunction.ConvertBHtoULong()* laver sådanne arrays om til unsigned 64 bit tal, der efterfølgende bruges i programmet.

Ved denne brug af *Random*-klassen i C# er det altså lykkes mig på en simpel måde at implementere hash funktionerne som ønsket. Prisen ved at gøre det på denne måde er dog, at jeg som sagt egentlig ikke ved hvordan *Random* fungerer, og jeg derfor kun kan stole på, at den er lavet "godt nok". Dertil kommer, at under programkørsel bliver der instantieret et meget stort antal *Random*-objekter, som kun bruges én gang hver især, og derefter skal fjernes af garbage collectoren. Det er med til at sløve programmet ret voldsomt.

5.6.3 Rekursiv sampling

Som den undersøgte datastruktur bliver præsenteret i [1], er forfatternes umiddelbare mål blot at få én samplet værdi ud af datastrukturen, når der samples. Som beskrevet i kapitel 4 nævner forfatterne i forbindelse med deres definering af kravene til datastrukturen, at målet er, at med sandsynligheden $1 - \delta$ returneres et sample, og at man med flere parallelle datastrukturer så kan opnå en vilkårlig sandsynlighed for dette (forudsat at $\delta < 1$).

Med anvendelsen jeg har valgt for datastrukturen, er det bestemt ikke tilstrækkeligt kun at få én værdi ud ved en sampling. Jeg ønsker jo netop at få så mange af de registrerede værdier ud som overhovedet muligt, og en enkelt samplet værdi er derfor ikke nok. Da jeg første gang læste artiklen, havde jeg det indtryk, at en sampling af datastrukturen med sandsynligheden $1 - \delta$ ville returnerer et tilfældigt element. Min oprindelige plan var derfor bare at blive ved med at sample datastrukturen, ind til den returnerede FAIL. Målet skulle så være at have nok parallelle datastrukturer, så δ blev så lille, at en anvendelig procentdel af værdierne ofte ville blive returneret.

Det var først senere i arbejdet med specialet, at jeg fandt ud af, at jeg havde misforstået kravet i [1]. Som tidligere beskrevet så er kravene til datastrukturen i stedet, at den med sandsynligheden $1 - \delta$ kan returnere én registreret værdi, og at denne værdi er næsten uniformt tilfældigt valgt blandt de registrerede værdier. Udover initialiseringen er datastrukturen deterministisk, og samples der flere gange i træk ,uden der registreres nye værdier i datastrukturen, er det den samme værdi der samles hver gang.

For at jeg altså kunne bruge datastrukturen i den sammenhæng jeg ønskede, var det nødvendigt at finde på en måde at bruge samplingen til at få så mange registrerede værdier ud som overhovedet muligt. Dvs. at finde en måde hvorpå gentagne samplinger ikke returnerer samme værdi (udover FAIL). Løsningen til dette gav næsten sig selv, nemlig at når en værdi er blevet returneret ved kald af Sample-metoden, fjernes den fra datastrukturen, så den ikke kan blive returneret igen. Datastrukturen kan nemlig ses som en stak af værdier, hvor indgangene i den simulerede vektor ligger i en tilfældig rækkefølge. Vi kan derfor sample og så

efterfølgende fjerne den returnerede værdi for at arbejde os ned gennem stakken. Det eneste jeg oprindeligt troede ville forhindrer os i at kunne returnerer alle værdier - og dermed adskiller datastrukturen fra en simpel komprimerings metode - var risikoen for FAIL, som stopper vores færd ned gennem stakken (i kapitel 6 forklarer jeg, hvorfor dette ikke virker).

Det er i metoden *GetSamples()* i *StatisticControls.cs*, at jeg har implementeret min rekursive måde at sample på. Jeg har lavet metoden, så man kan vælge at slå rekursiviteten fra, så samplingen i stedet foregår ved, at hver at de oprettede parallelle datastrukturer kun samples én gang, men den funktionalitet bør aldrig bruges i denne sammenhæng. Den interessante del af *GetSamples()* er derfor givet ved:

```
Queue SampledValues = new Queue();

int i;
int a;
bool ValReturned;

for (int j = 0; j < Trackers.Length; j++)
{
    ValReturned = Trackers[j].Sample(out i, out a);
    if (ValReturned && Values.HitTracker[i] == 0)
    {
        Values.HitTracker[i] = Convert.ToUInt32(a);
        cValueHolder tmpVal = new cValueHolder(i, a);
        SampledValues.Enqueue(tmpVal);
    };
}

while (SampledValues.Count > 0)
{
    cValueHolder GetVal = (cValueHolder)SampledValues.Dequeue();
    for (int j = 0; j < Trackers.Length; j++)
    {
        Trackers[j].Update(GetVal.i, -GetVal.a);
        ValReturned = Trackers[j].Sample(out i, out a);
        if (ValReturned && Values.HitTracker[i] == 0)
        {
            Values.HitTracker[i] = Convert.ToUInt32(a);
            cValueHolder tmpVal = new cValueHolder(i, a);
            SampledValues.Enqueue(tmpVal);
        };
    };
};

// Re-insert the removed values
for (int j = 0; j < Values.HitTracker.Length; j++)
{
    if (Values.HitTracker[j] > 0)
    {
        for (int k = 0; k < Trackers.Length; k++)
            { Trackers[k].Update(j, Convert.ToInt32(Values.HitTracker[j])); };
    };
};
```

Den overordnede ide med min kode for den rekursive sampling er, at så mange værdier som overhovedet muligt skal ud af datastrukturen. For at gøre dette benytter jeg en simpel "først ind, først ud"kø givet ved *SampledValues*. I den efterfølgende første kodeblok samples alle de parallelle datastrukturer, de returnerede værdier registreres i resultat vektoren *Values.HitTracker* (programmet antager denne vektor er initialiseret til 0 inden kaldet af metoden), og hver ny returneret værdi lægges i *SampledValues*-køen.

Hvad der er vigtigt, at lægge mærke til her er, at jeg altså ikke fjerner de returnerede værdier fra datastrukturen øjeblikkeligt, når de returneres. Årsagen til dette er, at begynder jeg at fjerne elementerne fra strukturen, inden jeg har samplet alle de parallelle datastrukturer, så risikerer jeg potentielt set, at nogle af disse strukturer ændres fra at returnere en værdi ved sampling til i stedet at returnere FAIL. I praksis burde fjernelsen af nogle enkelte ud af flere tusinde registrerede elementer ikke påvirke de efterfølgende samplinger, men er der mange parallelle

datastrukturer, er det ikke sikkert, antallet er så ubetydeligt endda. Desuden er mit mål jo netop at sample så mange af elementerne som overhovedet muligt, så bare risikoen, for at jeg går glip af nogle værdier, er uønsket.

Når programmet når til den efterfølgende *while*-løkke, er alle de parallelle strukturer altså blevet samplet én gang hver, og medmindre de alle returnerede FAIL, vil *SampledValues*-køen indeholde mindst ét element. I *while*-løkken tages de i køen gemte elementer, og fjernes fra datastrukturen en efter en. Hver gang et element bliver fjernet, samples alle de underliggende parallelle strukturer, og eventuelt nye returnerede elementer placeres i køen. På denne måde sikres det, at hver gang datastrukturen ændres ved fjernelsen af et samplet element, foretages det maksimale antal nye samplinger. Derved bliver så mange elementer som overhovedet muligt samplet fra datastrukturen.

Den sidste kodeblok har til formål at indsætte de fjernede elementer i datastrukturen igen efter endt sampling. Hvis ikke det bliver gjort, vil fjernelsen af de samplede elementer forhindre senere brug af datastrukturen, da der så vil mangle en del af den registrerede data. Her ser vi dog en styrke ved datastrukturen, nemlig dét, at det kun er ved initialiseringen, at der sker noget non-deterministisk i programmet. Fjernelsen og genindsættelsen af de samplede elementer er fuldt deterministisk, og uanset rækkefølgen elementerne genindsættes i, er datastrukturen før og efter kaldet af *GetSamples()*-metoden uforandret. Flest mulige værdier er altså blevet samplet, uden at det kan ses på datastrukturen.

5.6.4 DE strukturen og sliding window

Som beskrevet i afsnit 5.4.3 ønskede jeg at lave mit program, så det understøttede, at aktiviteten fra loggen der blev puttet ind i datastrukturen, også efterfølgende skulle kunne fjernes igen. Udover at dette kunne bruges til at lave et sliding window af loggen, så var det også nødvendigt, hvis jeg skulle kunne lave den rekursive sampling beskrevet i forrige afsnit.

Den overordnede datastruktur fra [1] har ikke umiddelbart nogen problemer med fjernelsen af elementer. Opdateringer af de registrerede værdier må gerne være negative, så længe de registrerede værdier ikke selv bliver mindre end 0. Fjernelsen af et element fra datastrukturen svarer derfor blot til at opdatere den registrerede værdi for elementet, så den netop bliver 0.

I den underliggende DE struktur fra [13] er der derimod ikke som udgangspunkt taget højde for, at man skal kunne fjerne elementerne igen. Ændringen til dette er dog forholdsvis simpel, og også noget der kort berøres i artiklen om DE ([13, slutningen af afsnit 4.]). Som omtalt i afsnit 4.5 er hele DE strukturen baseret på, at der opbygges et antal (*nmap*) arrays af bits, der bruges til at registrere forekomsten af mønstre af typen 0^k1 blandt de hashede værdier. Ændres disse arrays af bits til i stedet være f.eks. arrays af integers, kan man holde styr på antallet af gange, de givne mønstre er observeret, og på den måde tælle ned igen når mønstrene ses igen ved fjernelsen af data.

Prisen for dette er naturligvis et øget pladsforbrug, men selve ændringen er ganske simpel. Som det kan ses i min klasse *cDestinctElements* i *StatisticControls.cs*, har jeg lavet ændringen ved at initialisere et dobbelt array af integers i form af linien:

```
private int[,] Maps;
```

Ændringerne i koden som følge af dette i forhold til PCSA algoritmen (afsnit 4.5.3) fra [13] er så blot, at i *Update*-metoden skal der være mulighed for at kunne tælle op eller ned afhængig af om det er en indsættelse eller fjernelse der registreres. I selve estimerings delen af algoritmens pseudo kode ændres linien

```
while (BITMAP[i,n]=1) and (R<maxlength) do R := R+1;
```

til

```
while (BITMAP[i,n]>0) and (R<maxlength) do R := R+1;
```

så vilkårligt mange registrerede forekomster af et givent mønster blot svarer til "mønstret eksisterer", når der estimeres.

Ændringen af PCSA algoritmen til at understøtte fjernelse af elementer fra DE strukturen igen, er altså ganske simpel, men betyder en pæn relativ forøgelse af DE strukturens pladsforbrug.

5.6.5 DE strukturen ved 1 element

I afsnit 4.5 kom jeg kort ind på, at der i [13, afsnit 4] nævnes, at estimaterne af PCSA algoritmen først begynder at stemme overens med teorien og resultaterne beskrevet i [13], når værdien af $n/nmap$ (n er antallet af sete forskellige elementer) overstiger 10-20. Dette blev et stort problem under min implementering af programmet, da den oprindelige datastruktur beskrevet i [1] er baseret på, at de brugte DE strukturer i forbindelse med samplingen kan give nøjagtige estimeringer af mængder med 1 element. Pseudo koden for datastrukturens Sample() metode er:

Algorithm 5 Sample fra [1]

$j = \lceil \log(\text{DE.report}) \rceil$
if DE_j.report= 1 **then**
  return UE_j.report
else
  return FAIL
end if

Det er netop checket DE_j.report= 1, der viser dette krav til DE strukturerne. Under implementeringen syntes jeg, dette var et meget mærkeligt krav, [1] satte til brugen af DE, specielt når der i [1] overhovedet ikke blev skrevet noget om ,hvad forfatterne ville gøre, for at det rent faktisk skulle være muligt. I den reviderede udgave af artiklen ([2]) er problemet blevet løst ved, at UE strukturen er blevet ændret til selv at kunne foretage dette check, men da jeg ikke kendte til den artikel under implementeringen, beskriver dette afsnit derfor hvordan jeg - mere eller mindre - selv løste problemet.

Under min implementering var mit udgangspunkt, at der i [1] som eneste kommentar til implementeringen af DEerne stod "one can use a data structure from [13] to solve this problem". Jeg fokuserede derfor på hvordan DEerne kunne modificeres, så de også kunne estimere små værdier. Der var dog ikke meget hjælp af hente til dette i [13], da der i artiklens afsnit 4 under kommentarer til evt. implementering af PCSA blot stod: "If very small cardinalities were to be estimated, then based on the characterisation of probability distributions, *corrections* could be computed and introduced in the algorithm."I bund og grund var jeg kun interesseret i at kunne estimere korrekt, når DEen kun indeholdt ét element, og derfor var jeg ikke så vild med ideen om at skulle se på disse *corrections*.

Hvad jeg i stedet gjorde, var at se på, hvad der sker i PCSA algoritmen, når der kun er meget få elementer registreret. Som beskrevet i afsnit 4.5 er PCSA bygget op omkring et antal ($nmap$) arrays af bits, der bruges til at foretage estimeringen. Selve estimeringen sker med linierne:

```
S = 0;
for i := 0 to nmap-1 do
begin
    R := 0;
    while (BITMAP[i,n]=1) and (R<maxlength) do R := R+1;
    S := S+R;
```

```
end;

Xi := trunc((nmap/phi)*2^(S/nmap));
{Result Xi of the PCSA programme that estimates n}
```

Under estimeringen løbes de forskellige bit arrays igennem, og positionerne af de mindst betydende 0-bits bruges til at lave den endelige beregning. Hvad der slog mig var udseendet af denne matrix af bits, når de første elementer bliver registreret. Efter initialisering af DE strukturen men inden der er registreret nogle elementer, er udseendet af matricen ($nmap = 4$):

00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000

Efter registrering af det første element vil én af disse bits være blevet ændret til 1. Ud fra måden sætningen af disse bits foregår på er der 50% chance for, at det er den mindst betydende bit i et af arrayene, der sættes til 1. Dvs. f.eks.:

00000000000000000000000000000000
00000000000000000000000000000001
00000000000000000000000000000000
00000000000000000000000000000000

Ser vi nu på dette eksempel, og så kigger på registreringen af det næste element forskelligt fra det første, så er sandsynligheden for, at PCSA algoritmen vil sætte samme bit til 1 som ved første element ca. $0.5/nmap$. Der er jo netop igen ca. 50% chance for, at det er mindst betydende bit, der skal sættes til 1, og dertil kommer så chancen for, at den rammer samme array i matricen. Under antagelse af brugen af en perfekt tilfældig hash funktion er sandsynligheden for, at de første to registrerede forskellige elementer rammer samme bit i matricen givet ved

$$(\frac{1/2^1}{nmap})^2 + (\frac{1/2^2}{nmap})^2 + \cdots + (\frac{1/2^{32}}{nmap})^2 = \frac{\sum_{i=1}^{32} 2^{-2i}}{nmap^2}$$

hvis der hashes til 32-bit værdier. I min implementation kører jeg som udgangspunkt med $nmap = 64$, og i det tilfælde er sandsynligheden for, at to forskellige elementer i en DE struktur kun er årsag til én 1-bit i matricen altså ca. 0,0081%. Dvs. med meget stor sandsynlighed vil tilstedeværelsen af kun én 1-bit betyde, at der også kun er ét element i DE strukturen.

Denne observation af matricens udseende ved indsættelse af få elementer førte til, at jeg tilføjede følgende blok kode i *Report()*-metoden for *cDestinctElements*-klassen i *StatisticControls.cs*:

```
if ((S / Convert.ToDouble(nmap)) < 0.5d)
{
    int nValues = 0;
    for (int j = 0; j < Convert.ToInt32(nmap); j++)
    {
        for (int k = 0; k < MaxLength; k++)
        {
            if (Maps[j, k] > 0) { nValues++; };
        };
    };

    return nValues;
};
```

Denne kode checker om S (jævnfør det viste udsnit af pseudokoden for PCSA) har en lille relativ værdi i forhold $nmap$. I så fald tæller jeg antallet af indgange i

matricen større end 0, og returnerer denne værdi som estimatet. Dette sikrer altså, at når koden returnerer 1, så er der med stor sandsynlighed kun ét element i DEen.

Denne kode løste mit oprindelige problem og gjorde, at jeg fik datastrukturen fra [1] til at fungere. Jeg var dog ret træt af den lille risiko for, at DEerne ville returnere 1 selvom de f.eks. indeholdt to elementer, da det vil resultere i, at decideret forkerte værdier returneres af samplingen. Desuden var jeg ikke helt sikker på, for hvilke værdier af S det var smart at bruge denne kode ,omend det ikke gjorde så meget, da jeg var mest interesseret i at vide, hvad der skete omkring estimeringen af 1.

I alle tilfælde blev dette problem senere løst, da jeg fandt den reviderede artikel [2]. Løsningen i den med at lade UEerne selv foretaget checket betyder mest af alt, at risikoen for at der returneres forkerte værdier ved samplingen helt forsvinder, og at man altså altid ved, at man får rigtige værdier fra samplingen.

5.7 Korrektheden af implementationen

Når man kigger på korrektheden af enhver form for implementation, er det især to ting man taler: Gør det implementerede det ønskede, og hvilke omstændigheder skal der til, for at det ønskede ikke længere er muligt. Ved mit program drejer det sig derfor om, hvorvidt selve programmet implementerer handlingsforløbet i datastrukturen fra [1] korrekt, og under hvilke omstændigheder - ved f.eks. overløb i variablerne - datastrukturen ikke længere er valid.

5.7.1 Handlingsforløbet

En stor styrke ved datastrukturen fra [1] er, at den overordnet set er ret simpel. Desuden gør dét, at den arbejder på en simpel vektor af tal det nemt at kontrollerer resultaterne. Som noget af det første lavede jeg derfor koden til kontrol-klassen i programmet (*cNormalTracker* i *StatisticControls*), som netop blot holder styr på de rigtige værdier i den underliggende vektor. På den måde var det ligetil under implementeringen, at se på de genererede billeder, at samplingerne fra datastrukturen ikke gave værdier, der slet ikke passede til, hvad der faktisk skete i loggen.

For UE strukturen specifikt var det ligeledes nemt at se, at den gjorde det korrekte. UE er en meget simpel struktur og er meget nemt at overskue i koden.

DE strukturen var derimod ikke helt så simpel at overskue. Dét faktum, at jeg var nødt til at lave modifikationer til DE for at kunne bruge den, gjorde det ikke nemmere at vide, om det rigtige skete i koden. Jeg ventede derfor med at skrive koden for DE til sidst, så jeg var overbevidst om, at resten af koden i programmet fungerede. Eftersom DE blot tæller antallet af forskellige elementer i den underliggende datastruktur, lavede jeg klassen *cDestinctElements_Test* (i *TestCode.cs*) der simulerede DE strukturens adfærd ved simpelt hen altid at returnere det nøjagtige antal elementer. Det foregik blot ved at *cDestintElements_Test* på samme måde som min kontrol-klasse *cNormalTracker* har en vektor med alle de korrekte værdier for det observerede. På denne måde kunne jeg teste resten af programmet uden den rigtige kode for DE, og dermed være så sikker som mulig på at al den kode var færdig da jeg gik i gang med DE.

At jeg lavede en speciel klasse til simulering af DE strukturen, kan virke lidt underligt, hvis man kigger på min programkode nu. Efter ændringerne i min kode som følge af kendskabet til den reviderede artikel [2] bruges der jo kun én DE i hele datastrukturen, til at holde styr på det samlede antal forskellige elementer der er registreret. Det oplagte ville derfor være, at test-klassen *cDestintElements_Test* blot brugte den allerede lagrede kontrolstruktur til at tælle elementerne i vektoren i stedet for selv at oprette en vektor med værdierne. *cDestintElements_Test* skal dog ses i lyset af, at jeg oprindeligt have én DE struktur for hver UE struktur

pga. anvisningerne i [1]. Hver af disse DE strukturer indeholdt en delmængde af værdierne for de observerede elementer, og derfor kunne kontrolstrukturen ikke benyttes.

Ved hjælp af *cDestinctElements_Test* kunne jeg teste resten af programmet. Ved implementeringen af selve DE strukturen i form af *cDestinctElements*-klassen kunne jeg derfor nøjes med at fokusere på, at DEen kunne estimere korrekt, inden jeg lod resten af programmet bruge den. Til det formål lavede jeg endnu en test-klasse, nemlig *cDETester* i *TestCode.cs*. Denne klasse er meget simpel i den forstand, at den sender den registrerede aktivitet i den givne log direkte ind i en DE struktur uden at bruge resten af datastrukturen fra [1]. På den måde kunne DEens estimerings egenskaber testes, uden at den blev påvirket af resten af programmet. Under implementeringen af DEen kunne jeg derfor løbende vurdere kvaliteten af dens estimeringer, og dermed se om de stemte overens med teorien i [13].

På baggrund af resultaterne ved brugen af de to test-klasser, og ud fra de resultater jeg har fået ved brugen af den færdige implementation, vil jeg mene at mit program implementerer handlingsforløbet beskrevet i [1] korrekt. Dog med nogle justeringer jævnfør [2], hvilket specielt drejer sig om ændringerne af UE strukturen, så hver parallel datastruktur kun behøver benytte én DE.

5.7.2 Overløb og pladsforbrug

Et hovedkrav i [1], for at man kan benytte datastrukturen, er, at der kigges på et endeligt univers U, hvis elementer har tilknyttet en endelig værdi M. Mht. universet er der i min brug af datastrukturen ikke nogen problemer, da mængden af IP adresser er en endelig mængde. Ved værdien jeg registrerer forholder det sig dog lidt anderledes.

I min konkrete brug af datastrukturen hvor jeg foretager simuleringen på en allerede given log fra en web server, er der også tale om et endeligt antal gange, de enkelte IPer kan optræde i loggen. Loggen jeg bruger er jo statisk, og jeg ved derfor at ingen IP optræder flere gange end det samlede antal registreringer i loggen (i min log er dette antal 6078051). Ved brug af datastrukturen som et rigtigt redskab i forbindelse med overvågning af en given linie kan man dog ikke være sikker på dette. Man kan gætte på, hvor meget trafik man normalt vil observere, men i sidste ende er det kun den overvågede linies hardware, der sætter grænser for, hvor meget trafik der potentielt kan blive registreret indenfor et givent tidsinterval.

Der er to konkrete steder i koden for datastrukturen, hvor størrelsen af værdierne, der skal gemmes, har betydning:

- UE strukturen. I koden for UE strukturerne er det i integer variablerne c, C og $c2$. Disse variabler opdateres ud fra både størrelsen af universet, og værdierne der tilknyttes elementerne i universet. Sker der overløb i disse variabler, vil efterfølgende samplinger fra den pågældende UE struktur være forkerte.

- DE strukturen. I koden for DE strukturen er det i forbindelse med ændringen, jeg har lavet, for at understøtte sliding window i loggen (se afsnit 5.6.4). Skiftet fra at bruge arrays af bits til arrays af integers betyder, at disse tællere også kan overløbe. Pga. hashingen, der foretages dette sted, kan mange elementers værdier bidrage til den samme tæller i matricen. Specielt den første tæller i hvert array er udsat - hvis hashingen er uniform vil ca. 50% af elementerne ramme første tæller. Er $nmap = 64$ betyder det, at $\frac{1}{128}$ del af elementerne burde blive mappet til den første tæller i hvert array.

Pladsforbruget af datastrukturen afhænger af, hvor store variabler man vælger at bruge disse to steder i koden. I en virkelig brug af datastrukturen til overvågning er det derfor et vigtigt valg, man skal træffe, og man vil desuden være nødt til at

sørge for, at koden håndterer overløb på en eller anden måde. I mit program har jeg valgt blot at lade variablerne være "store nok"til min brug. Dvs. jeg har ikke forsøgt at minimere pladsforbruget, men har i stedet brugt integers af en størrelse, der sikrede, at der ikke skulle være risiko for overløb i mine kørsler. Jeg har valgt at gøre dette, da mit mål jo først og fremmest er, at se om datastrukturen kan bruges i min sammenhæng, ikke hvor lidt plads jeg kan få den ned på. Det er dog stadig en vigtig begrænsning i datastrukturen, som man skal være opmærksom på.

5.8 Sværhedsgraden af implementeringen

Som altid ved implementeringen af et program har jeg undervejs haft min del af irriterende fejl, der har været besværlige at finde. Det er der selvfølgelig ikke noget unormalt i, og som oftest beroede de i simple indtastningsfejl i koden, eller at jeg havde læst noget forkert i artiklerne.

Overordnet set synes jeg, at det var nemt at implementere datastrukturen fra [1]. En kæmpe fordel var nemlig det faktum, at det i sidste ende blot er en vektor med ikke-negative værdier, datastrukturen bygger på. Det var derfor nemt under hele implementeringen, at se om min kode gjorde det rigtige. Dét, at man kan se, om koden virker eller ej, synes jeg, gør det hele meget nemmere, da man så ikke behøver være nervøs for fejl, der blot fordrejer ens resultater.

Når det er sagt, så må jeg dog stadig erkende, at jeg synes, at [1] springer temmelig nemt henover hvordan især DE strukturen skal laves. Dét, at PCSA algoritmen fra [13] ganske simpelt ikke kan bruges uden (i mine øjne) væsentlige modifikationer i forhold til hvad forfatterne af [13] har beskrevet, synes jeg ikke bare kan ignoreres. Jeg sad i hvert fald selv fast ved implementeringen af DE strukturen, ind til jeg fik styr på, hvad der var fejl i min kode, og hvad der var "fejl"som følge af begrænsningerne i den umodificerede PCSA.

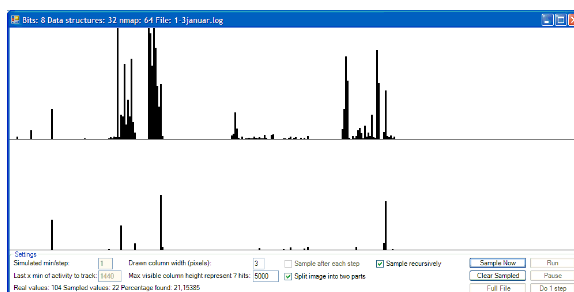
Den reviderede artikel [2] har rettet op på meget af dette ved netop at ændre datastrukturen, så problemerne med DE strukturerne ikke længere har samme indflydelse. Hvis man derfor implementerer datastrukturen med [2] som udgangspunkt, vil jeg derfor mene, at man ikke burde møde de store problemer under implementeringen. Det vil dog som altid afhænge af, hvad man vil bruge datastrukturen til, da det afgør, hvor meget "udenoms"kode det er nødvendigt at pakke datastrukturen ind i.

Kapitel 6

Kørsler med implementationen

Med implementationen af datastrukturen på plads er tiden kommet til at bruge den. Hvis det skal være muligt at vurdere, om datastrukturen er værd at bruge i praksis, er det nødvendigt at have noget at basere en sådan vurdering på. I dette kapitel vil jeg derfor beskrive de test kørsler, jeg har foretaget med mit program, og give en forklaring på hvorfor resultaterne er, som de er. En egentlig vurdering af datastrukturen ud fra resultaterne kommer jeg med i kapitel 7.

6.1 Typer af tests



Figur 6.1: Screenshot - Eksempel på et samplet søjlediagram

Som tidligere beskrevet er hele målet med mit arbejde med datastrukturen, at vurdere hvor anvendelig den er i praksis, når der sættes fokus på overvågning af netværk. I kapitel 5 har jeg beskrevet, hvordan jeg har lavet mit program, så det simulerer en overvågning af aktiviteten på en webserver. Dvs. antallet af connections fra IPer ude i verden til serveren. I programmet opbygges et korrekt søjlediagram over den registrerede aktivitet, og målet for programmet er så at se, om den er i stand til at opbygge et tilsvarende diagram ved hjælp af datastrukturen. Figur 6.1 viser et eksempel på en sådan kørsel af programmet. Øverst er det rigtige søjlediagram, nederst er det samplede.

For at kunne vurdere om det mål er nået, er der to ting, jeg ønsker at kigge nærmere på:

1. **Fuldstændigheden af samplingen.** Den vigtigste parameter, for om datastrukturen overhovedet er noget værd i min anvendelsesform, er hvor gode

diagrammer, den er i stand til at opbygge. Dvs. hvor mange parallelle datastrukturer der skal til - hvis det overhovedet kan lade sig gøre - for at man med samplingen som redskab er i stand til at opbygge diagrammer, der er tæt på det rigtige diagram for aktiviteten. Dette er jeg tvunget til at undersøge, for at jeg i det hele taget har noget at vurdere på. Det vil samtidigt give mig information om antallet af parallelle datastrukturer, der skal bruges. Det vil sætte mig i stand til også at sige noget om pladsforbruget ved denne form for sampling.

2. **Måden der samples.** Én ting er, om samplingen potentielt kan opbygge diagrammer, der minder om det virkelige diagram, noget andet er hvad eventuelle mangler ved det samplede i forhold til det rigtige har af betydning. Hvis samplingen har begrænsninger, som følge af måden datastrukturen laver samplingerne på, er det væsentligt for vurderingen.

Det første punkt er det nemme punkt i den forstand, at det er forholdsvis nemt at måle på. Udgangspunktet for hele simuleringen er jo, at jeg på tidspunktet for samplingen har en vektor x bestående af værdier $x_0, x_1, \ldots, x_i$ $i \in [U]$, som hver især repræsenterer et antal indgange i webservens log for et givent interval af IP adresser. I mit program bruger jeg plads på at opretholde et array med de faktiske værdier, og når der foretages en sampling, opbygger programmet endnu et array, med de samplede værdier. For at måle samplingens fuldstændighed er det derfor en simpel ting i mit program efter endt sampling, at sammenligne disse to arrays og checke hvor mange af de faktiske værdier der er blevet returneret ved samplingen. Dette forhold giver en simpel målbar værdi for samplingen, der fortæller, hvor tæt det samplede diagram er på det rigtige.

Det andet punkt er lidt sværere at forholde sig til. Dét, at der bruges sampling, betyder jo netop, at der vil mangle noget i det opbyggede diagram i forhold til det reelle. Hvis ikke der går data tabt, er det ikke sampling der foretages men derimod lossless lagring. Der bør derfor altid mangle noget i diagrammet genereret af samplingen, og spørgsmålet er, hvad der vil mangle.

En af grundkernerne i [1] er, at datastrukturen er næsten uniform tilfældig i udvælgelsen af de værdier, der samples, så den ikke-samplede del vil så også være tilfældig. Jeg regnede derfor oprindeligt med at bruge dette punkt, til at kommentere på hvilken betydning det har at mangle en vis procentdel af værdierne efter samplingen. Under implementeringen af programmet opdagede jeg dog, at måden datastrukturen foretager samplingen på, kan have en speciel betydning for hvad, der returneres ved samplingen, og hvad der udelades.

Det er altså disse to områder, jeg gerne vil belyse, og jeg har på den baggrund lavet en række test kørsler, der præsenteres i de følgende to afsnit.

6.2 Fuldstændigheden

I dette afsnit kommer jeg ind på, hvordan jeg tester fuldstændigheden af samplingerne. Dvs. det praktiske setup af programmet ved de forskellige tests. Jeg anfører resultaterne og kommenterer på, hvorfor de ser ud, som de gør. Ved hver test har jeg taget et screenshot af resultaterne i programmet, og disse er alle placeret på den medfølgende CD.

6.2.1 Opsætning

Til testen af fuldstændigheden ved samplingerne har jeg valgt at lave kørsler af programmet, hvor jeg varierer på parametrene for opdelingen af IP adresserummet, og antallet af parallelle datastrukturer programmet skal benytte. Til opdelingen af

IP adresserne har jeg i mit program brugt henholdsvis 8, 12 og 16 som parametre, og disse valg giver en underliggende vektor af værdier med henholdsvis 256, 4096 og 65536 indgange. Eftersom den registrerede aktivitet i den undersøgte log bliver fordelt over disse indgange, er min ide at undersøge, hvordan datastrukturen opfører sig, når der både er få og mange indgange større end 0 i vektoren.

For hver af disse tre længder af vektoren har jeg kørt tests for henholdsvis 16, 32, 48 og 64 parallelle datastrukturer. Ideen er her, at se hvordan samplingen klarer sig ved forskellige antal datastrukturer. Der er ingen speciel grund til, at det netop er disse værdier, jeg har valgt, udover at resultaterne de giver, er gode nok til, at jeg kan se noget ud fra dem.

For at kunne lave disse tests skal jeg også vælge et stykke log at gøre det på. Jeg har valgt indgangene i perioden fra kl. 00:00 1. januar 2006 til kl. 24:00 3. januar 2006. Den nævnte periode har jeg skilt ud i en seperat fil (*1-3januar.log* på CDen). Jeg har valgt tre døgn ud fra tanken om, at det er et realistisk tidsinterval, en administrator kunne være interesseret i. Mest af alt har jeg med valget af en periode fra loggen ønsket at sikre, at alle tests kører på det samme tidsinterval, så resultaterne bedre kan sammenlignes.

Ved hver kombination af de to parametre jeg varierer på har jeg lavet fem uafhængige test kørsler. Dvs. jeg har lavet 60 test kørsler i alt. Da initialiseringen af datastrukturen trods alt involverer tilfældighed, håber jeg, at denne måde sikrer mig mod særligt "heldige/uheldige"resultater. Med fem kørsler for hvert valg af parametre burde der være nok til, at sådanne specielle resultater ikke bliver dominerende.

Som en sidste bemærkning til opsætningen af testene vil jeg kort forklare, hvorfor jeg ikke har varieret på den tredje parameter i mit program, *nmap*. I alle test kørslerne har jeg holdt *nmap* fast på 64, da det ifølge [1] og [13] betyder, at præcisionen af DEerne er bedre, end hvad der kræves. At variere på *nmap* ændrer på DEernes præcision ved, at der skrues op og ned for hvor meget plads, de må bruge. Omend det er en vigtig ting, hvis man ønsker at gøre det samlede pladsforbrug så lille som muligt, så har det ingen betydning for testene, jeg foretager her. Jeg har derfor bevidst sat værdien for *nmap* højt, så jeg ikke behøver at bekymre mig om DE strukturens præcision.

6.2.2 Resultater

Jeg har lavet 60 test kørsler, og resultaterne af dem er angivet i de følgende tabeller. Der er én tabel for hver opdeling af IP adresserne i henholdsvis 8, 12 og 16 bits. Rækken "Datastrukturer"angiver parameteren for hvor mange parallelle datastrukturer, testen er kørt med. De resterende rækker angiver antal samplede værdier ved hver enkelt kørsel. I parantes hvor mange procent de samplede udgør af det reelle antal af indgange i vektoren med positiv værdi.

Tabel 6.1: 8 bits opdeling (vektor længde 256, $\|x\|_0 = 104$)

Datastrukturer	16	32	48	64
Kørsel 1	7 (6,7%)	20 (19,2%)	52 (50,0%)	63 (60,6%)
Kørsel 2	11 (10,6%)	22 (21,2%)	53 (51,0%)	104 (100%)
Kørsel 3	4 (3,8%)	18 (17,3%)	10 (9,6%)	104 (100%)
Kørsel 4	4 (3,8%)	11 (10,6%)	50 (48,1%)	104 (100%)
Kørsel 5	7 (6,7%)	13 (12,5%)	26 (25,0%)	104 (100%)

Hvad betyder disse resultater så? Jeg kan i hvert fald udlede følgende to ting:

Tabel 6.2: 12 bits opdeling (vektor længde 4096, $\|x\|_0 = 728$)

Datastrukturer	16	32	48	64
Kørsel 1	4 (0,5%)	11 (1,5%)	21 (2,9%)	28 (3,8%)
Kørsel 2	8 (1,1%)	10 (1,4%)	15 (2,1%)	22 (3,0%)
Kørsel 3	5 (0,7%)	18 (2,5%)	17 (2,3%)	19 (2,6%)
Kørsel 4	9 (1,2%)	11 (1,5%)	20 (2,7%)	19 (2,6%)
Kørsel 5	6 (0,8%)	12 (1,6%)	14 (1,9%)	22 (3,0%)

Tabel 6.3: 16 bits opdeling (vektor længde 65536, $\|x\|_0 = 2541$)

Datastrukturer	16	32	48	64
Kørsel 1	3 (0,1%)	7 (0,3%)	18 (0,7%)	28 (1,1%)
Kørsel 2	5 (0,2%)	7 (0,3%)	17 (0,7%)	20 (0,8%)
Kørsel 3	7 (0,3%)	6 (0,2%)	21 (0,8%)	23 (0,9%)
Kørsel 4	7 (0,3%)	8 (0,3%)	13 (0,5%)	24 (0,9%)
Kørsel 5	6 (0,2%)	7 (0,3%)	13 (0,5%)	22 (0,9%)

1. Jo flere parallelle datastrukturer der benyttes, jo flere værdier samples. Det er dog ikke overraskende, eftersom hver datastruktur har mulighed for at returnere samplede værdier.

Algorithm 6 Sample fra [2]

$j = \lceil \log(\text{DE.report}) \rceil$
return UE_j.report

2. Ved kørslerne for 8 bit er datastrukturerne overvældende succesfulde med at sample værdierne. Specielt ved brugen af 48 og 64 datastrukturer. Kigger man på de tilsvarende kørsler for 12 og 16 bit, er det nogen lunde det samme antal værdier, der begge steder returneres ved brugen af 48 og 64 datastrukturer. Hvorfor er det så voldsomt mange flere, der returneres ved 8 bit kørslerne? Det tog mig lidt tid at indse hvorfor, men svaret på dette spørgsmål er faktisk ret oplagt. Denne lidt specielle adfærd opstår som følge af min rekursive sampling, samt måden datastrukturen er bygget op på.

   Jeg har tidligere forklaret i afsnit 5.6.3, hvordan jeg har implementeret mit program, så det er i stand til at trække så mange samplede værdier ud af de parallelle datastrukturer som overhovedet muligt. Det foregår ved, at hver gang jeg får en værdi ud, så fjernes den midlertidigt for at se, om det kan bevirke, at yderligere samplinger giver nye værdier.

   Denne måde at gøre det på, skal sammenholdes med hvordan samplingen egentlig foregår i programmet. Jeg ser derfor igen på pseudokoden vist ovenfor for Sample() (se afsnit 4.4.5) fra [2]. Som det kan ses, så er det $j = \lceil \log(\text{DE.report}) \rceil$, der afgør hvilken UE_j samplingen forsøges foretaget ud fra i hver enkelt af de parallelle datastrukturer. Jeg vil nu kigge lidt nærmere på, hvad der sker i kørslerne med 16 bits opdeling og 64 datastrukturer.

   Som angivet i tabel 6.3 er $\|x\|_0 = 2541$ for den valgte periode i loggen. Hvis jeg nu antager, at DE strukturerne er i stand til at estimere denne værdi præcist,

så har jeg altså at:

$$j = \lceil \log(2541) \rceil \quad \Rightarrow \quad j = 12$$

I hver enkelt af de 64 parallelle datastrukturer er det altså UE_{12}, der bliver samplet. Jævnfør afsnit 4.4.2 er det kun når UEerne indeholder en enkelt positiv indgang fra vektoren, at de er i stand til at returnere et sample. Antager jeg nu, at vi har den mest optimale situation, hvor hver eneste af de 64 UE_{12} strukturer kun indeholder ét element, så får jeg 64 samplede værdier ud.

I mit program fjerner koden for den rekursive sampling nu disse 64 elementer fra datastrukturerne, for at forsøge at bevirke at andre elementer kan samples. Det betyder så, at ved næste kald af Sample() får jeg:

$$j = \lceil \log(2541 - 64) \rceil \quad \Rightarrow \quad j = 12$$

Det er her årsagen til problemet ligger. Det er nemlig ikke blevet fjernet nok elementer til at ændre værdien af j, og med $j = 12$ igen vil samplingen stoppe her, for alle UE_12 strukturerne er jo blevet samplet og tømt. Det betyder altså, at hvis ikke den første sampling af alle de parallelle datastrukturer returnerer nok værdier til at ændre på j, så kan der maks samples én værdi pr. datastruktur.

Jeg fortsætter nu med at antage, at jeg er heldig, og kan få en samplet værdi ud af hver datastruktur. Det betyder, at jeg skal bruge at antal datastrukturer svarende til halvdelen af $\|x\|_0$, for at være sikker på at værdien af j ændrer sig efter første runde af sampling. Er det tilfældet, vil programmet til gengæld ved næste runde af sampling af UE_{j-1} strukturerne stå utrolig godt rustet. Antallet af datastrukturer vil jo være uforandret, hvilket betyder, at der er mindst lige så mange parallelle datastrukturer som indgange at sample. Når programmet dette punkt, er der derfor en god sandsynlighed for, at j bliver samplet i bund (til $j = 0$), og at alle værdierne i vektoren x dermed bliver samplet.

I kørslerne 2-5 for 8 bits opdeling med 64 datastrukturer er det netop denne situation, der er opstået. Det er derfor, at programmet i disse tilfælde har formået at sample 100% af værdierne.

Har man en situation, hvor dét, man overvåger, ofte antager værdier i næsten alle indgange i den underliggende vektor, kan jeg ud fra disse to observationer (særligt nr. 2) konkludere: Man skal bruge et antal parallelle datastrukturer på mindst 50% af antallet af indgangene i den underliggende vektor, ellers er antallet af ens samplinger begrænset af antallet af parallelle datastrukturer. Kørsler med så mange parallelle datastrukturer vil på baggrund af UEerne alene betyde, at det samlede pladsforbrug vil være større end et simpelt array indeholdende alle værdierne.

Det interessante der er tilbage, er derfor, om datastrukturen er anvendelig, hvis den sampler mindre end 50% af værdierne.

6.3 Måden der samples

Ud fra forrige afsnits resultater er det tydeligt, at skal der være bare den mindste chance for en pladsmæssig gevinst ved brugen af datastrukturen, så er man tvunget til kun at sample en mindre del af de observerede værdier. F.eks. kan man gå efter en sampling på 10% af værdierne, hvilket tilsvarende vil give evt. brugere 10% chance for at se synderen ved eksempelvis et denial-of-service angreb.

Under implementeringen af mit program og de dertil hørende test kørsler lagde jeg dog mærke til noget lidt mærkeligt. Når jeg brugte programmet på den store fil med 6 måneders log fra webserveren for 246.dk, og lavede flere samplinger efter

hinanden (dvs. uden at initialisere en ny datastruktur), så syntes jeg, at der var en del sammenfald i værdierne, der blev samplet hver gang. Oprindeligt havde jeg regnet med, at ændringerne i datastrukturens tilstand som følge af behandlingen af dataen i loggen også ville bevirke, at valget af samplede indgangene ville ændre sig. Det lignede bare ikke, at det var det, der foregik i praksis. Det blev baggrunden for følgende ret simple test.

6.3.1 Opsætning

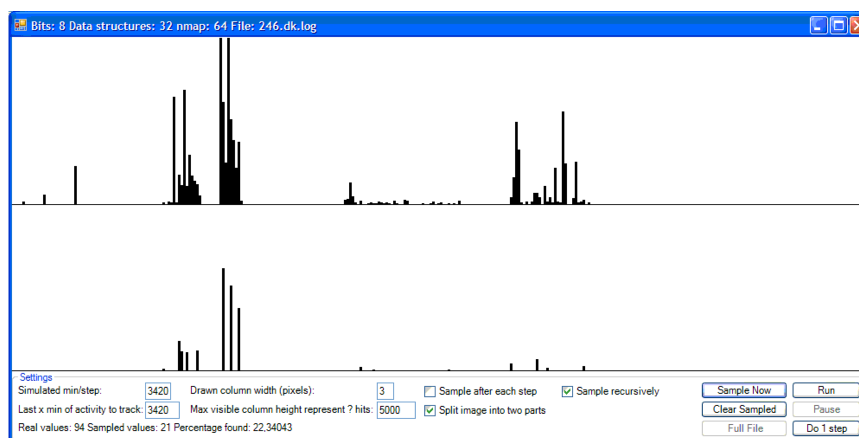
For at kunne bekræftige denne uventede adfærd har jeg kørt fem simulerede tids intervaller i mit program og lavet en sampling efter hver af dem. Hvert tids interval har jeg sat til 3 døgn (3420 minutter), og ligeledes har jeg sat programmet til at fjerne registreret trafik ældre end 3 døgn fra datastrukturen. Jeg har kørt samplingen på en 8 bits opdeling af IP adresserne og med 32 parallelle datastrukturer for at få et synligt antal værdier samplet.

Pointen med disse valg er, at jeg kun initialiserer én datastruktur, som altså bruges på alle fem tidspunkter, der samples. Ved at lade registreret aktivitet blive fjernet efter 3 døgn svarer det til, at jeg kører med et sliding window på 3 døgn. Der samples altså kun på registreret aktivitet for de seneste 3420 minutter i loggen. At jeg så har ladet programmet køre 3 døgn (simuleret tid) mellem hver sampling, betyder sammen med valget for fjernelsen af den registrerede aktivitet, at der ikke er noget overlap i dataen fra loggen, der ligger til grund for hver sampling. Det eneste fælles for samplingerne er altså, at de bruger en identisk initialiseret datastruktur.

Hele ideen med testen er at simulere en reel brug af datastrukturen. En administrator kunne f.eks. initialisere programmet, og sætte det til at overvåge trafikken på en webserver. Valget af samplings tidspunkterne svarer til, at han så en gang imellem i ugerne efter, sætter sig ved computeren, og beder programmet om at give en sampling af den registrerede aktivitet for de seneste 3 døgn.

6.3.2 Resultat

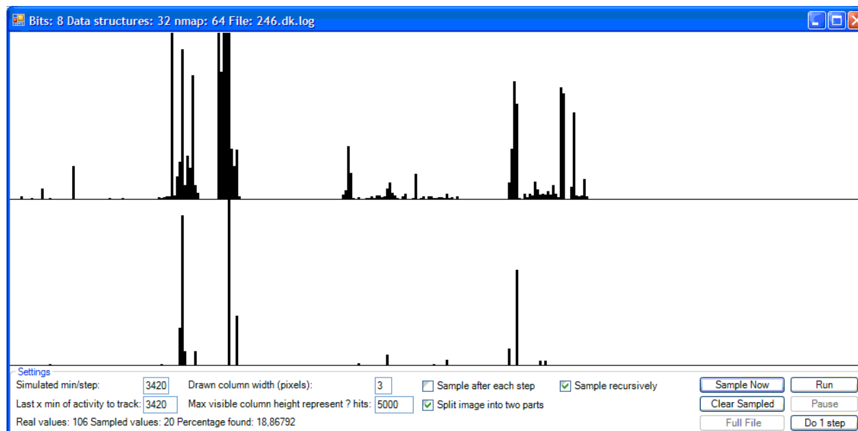
Screenshots af de fem samplinger efter henholdsvis 3, 6, 9, 12 og 15 døgn i loggen er vist i figur 6.2-6.6.



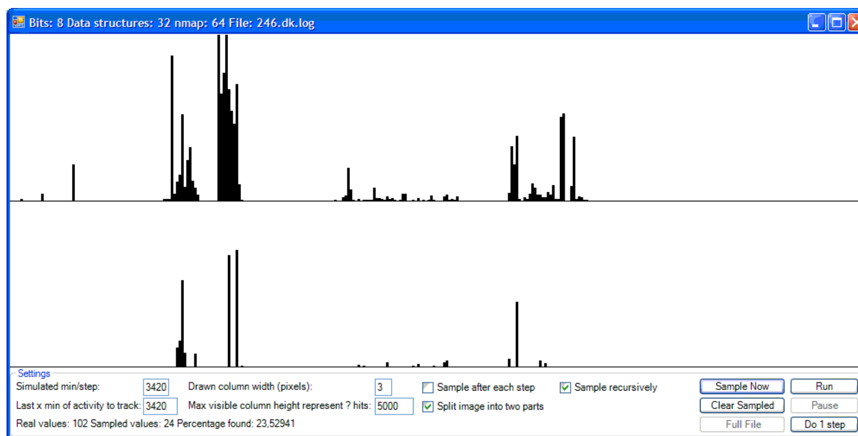
Figur 6.2: Screenshot - Sampling efter 3 døgn

Som tidligere forklaret viser det øverste søjlediagram den faktiske aktivitet i loggen og det nederste samplingen. Dvs. det nederste diagram skal forsøge at gengive det øverste så godt som muligt.

Der er to ting, der er meget tydelige, når man kigger på de fem figurer:



Figur 6.3: Screenshot - Sampling efter 6 døgn

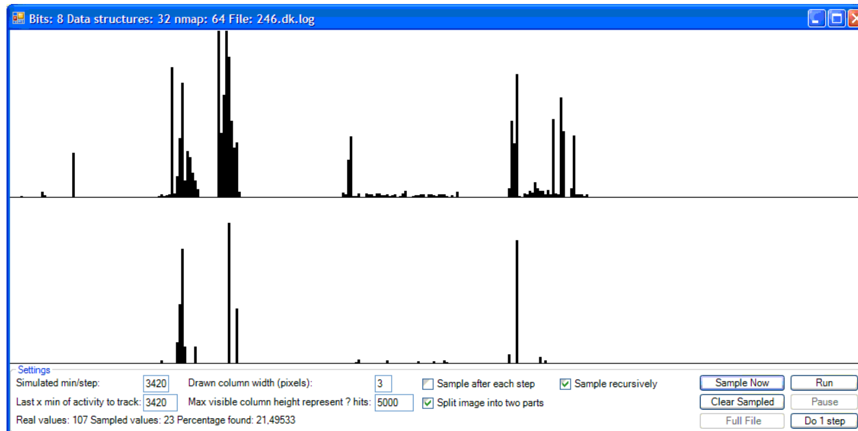


Figur 6.4: Screenshot - Sampling efter 9 døgn

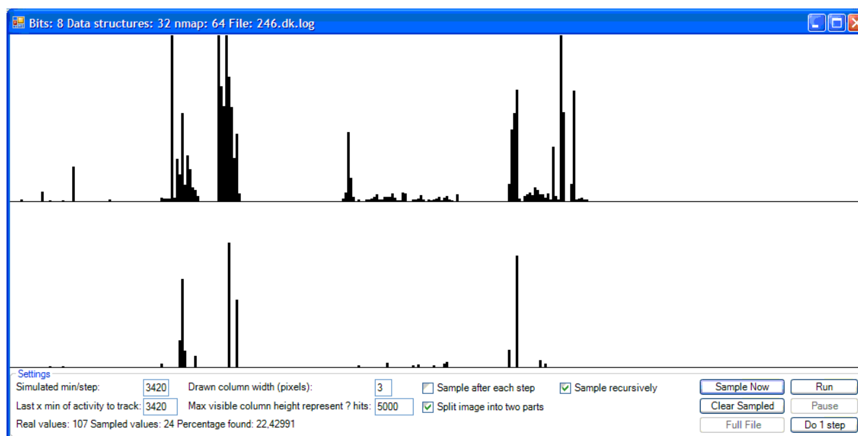
1. Forandringerne i det øverste diagram over den reelle registrerede aktivitet består næsten kun i forskellige søjle højder. Dvs. forskelle i hvor mange gange de enkelte IP intervaller optræder i loggen i det givne tidsrum. Ser man godt efter, er der enkelte små søjler, der kommer og går fra billede til billede (f.eks. i venstre side af diagrammerne), men de fleste søjler er de samme. Blot med forskellige højder.

2. Forandringerne fra billede til billede i det samplede diagram nederst består tilsvarende også mest af forandringer i højderne på de samme søjler. Også her er der enkelte søjler, der kommer og går, men det er kun nogle enkelte, der er tale om. Disse forandringer er dog mere tydelige, da det godt kan være høje søjler, der er blevet samplet i ét billede og udeladt i et andet.

At forandringerne ser ud, som de gør, i diagrammet for den reelle trafik, er ikke mærkeligt. Fordelingen af IP adresser til computere på Internettet er overordnet set geografisk baseret. Bestemte IP intervaller er tildelt specifikke netværk, og disse er oftest geografisk afgrænset. Den fysiske størrelse af netværket afgør så, om afgrænsningen placerer IPerne i et fælles kontinent, land, bygning osv. Denne fordeling af IPerne betyder, at trafik til en given server oftest vil fremstå som grupperinger af IPer, der repræsenterer brugernes globale placering. F.eks. vil de fleste læsninger af en given hjemmeside foregå i de dele af verden, hvor folk taler det sprog, hjemmes-



Figur 6.5: Screenshot - Sampling efter 12 døgn



Figur 6.6: Screenshot - Sampling efter 15 døgn

iden er skrevet i. At denne gruppering af aktiviteten finder sted, er der ikke noget nyt i, og en administrator vil derfor ofte blot bruge diagrammer som de viste til at holde øje med forandringer i den normale aktivitet.

I forhold til den implementerede datastruktur har det dog stor betydning, at almindelig netværks trafik opfører sig på denne måde. Jævnfør gennemgangen af datastrukturen i kapitel 4 så er det antallet af positive indgange fra den underliggende vektor x i hver enkelt UE, der afgør, om UEen er i stand til at returnere et sample. Når der i den observerede trafik for det meste kun ændres op og ned på værdierne i allerede positive indgange, så ændrer det ikke på samplingerne, UEerne kan levere. Kun ændringer i indganges værdier væk fra eller til 0 påvirker UEerne i den sammenhæng.

Den praktiske betydning af dette er tydelig på testens screenshots. Minimale forandringer, i hvilke IP intervaller der genererer den registrerede aktivitet, medfører, at der også kun er minimale forandringer i hvilke søjler, der samples. Endnu vigtigere: Hvis den overvågede server rent faktisk har aktivitet på alle målte indgange, så ingen af disse under normal brug nogen sinde bliver 0, så er valget af indgangene, der samples, låst fast efter initialiseringen af datastrukturen.

Kapitel 7

Er datastrukturen værd at bruge

I dette kapitel vil jeg på baggrund af mit arbejde i specialet op til dette punkt give min vurderingen af den praktiske værdi af datastrukturen fra [1]. Både i sammenhæng med den anvendelse jeg har valgt og i andre typer af anvendelser ved netværk.

7.1 I min anvendelse

Ud fra resultaterne af mine tests i kapitel 6 vil jeg her starter med at komme med en klar vurdering: Synes jeg, at datastrukturen fra [1] er brugbar til sampling af net trafik, svarende til den type jeg simulere i mit program? **Nej**. Grundene til dette meget klare kan summeres op i følgende punkter:

1. **Mangelfuld fleksibilitet.** Det i mine øjne største problem ved datastrukturen i den valgte sammenhæng er den manglende fleksibilitet ved valg af ønsket antal samplinger. Da jeg oprindeligt så på [1], og valgte at basere mit speciale på dens indhold, forstod jeg design kravet til datastrukturen, om at den skulle returnere et sample med sandsynligheden $1 - \delta$, helt anderledes, end det viste sig at være ment. Jeg troede kravet galt gentagne samplinger af datastrukturen, så f.eks. $\delta = 0,2$ ville betyde, at man i gennemsnit kunne sample 5 gange, inden man mødte FAIL. Jævnfør afsnit 6.2 så kan jeg på baggrund af datastrukturens opbygning se, at kravet åbenbart kun gælder det første kald af metoden Sample().

   Det faktum er årsagen til (se afsnit 6.2), at antallet af værdier der potentielt kan samples, er proportionalt med antallet af brugte parallelle datastrukturer. I hvert fald ind til man når "kritisk masse", hvor antallet af datastrukturer er højt nok til, at man går fra at de enkelte datastrukturer kun kan sample én værdi hver, til at de kan sample flere rekursivt.

   At jeg kalder dette for "mangelfuld fleksibilitet", er fordi, jeg ud fra min oprindelige opfattelse af det nævnte krav troede, det ville være praktisk muligt at have en parameter, der kunne skrues op og ned på, som angivelse af det ønskede antal samplinger. F.eks. så en administrator kunne angive, at han gerne ville have ca. 80% af værdierne samplet. Som det har vist sig at fungere, kan man kun det til en vis grad. Det er kun ved at ændre på antallet af brugte datastrukturer, at man kan påvirke antallet af samplinger. Et ønske om en sampling af 50%-100% af værdierne betyder derfor i praksis, at man beder om en sampling på 100%.

2. **Pladsforbruget.** Det kunne være til at leve med, at antallet af samplinger er proportionalt med antallet af parallelle datastrukturer, hvis hver enkelt af disse strukturer brugte meget lidt plads. Det er bare ikke tilfældet. Som jeg allerede har været lidt inde på i afsnit 6.2, så er det nok at se alene på UEerne i datastrukturen for at få en fornemmelse af problemet.

   Datastrukturen skal holdes op imod det simple array, man ellers kunne bruge til at holde styr på alle værdierne. UEerne har tre integers hver, og jeg antager nu, at de blot er af samme størrelse, som de integers man ville have brugt i arrayets indgange. Sættes opdelingen af IP adresserne til 16 bit, betyder det, at hver enkelt af de parallelle datastrukturer indeholder 16 UEere, som dermed svarer til 48 indgange i arrayet. Det betyder, at for at alene UEernes pladsforbrug ikke skal overstige, hvad man ville have brugt på arrayet, så må antallet af parallelle datastrukturer højest være 1/48 del af arrayets længde.

   Dette betyder altså, at hvis UEerne ikke skal fylde mere end arrayet, så er det ca. 2% af indgangene i arrayet, der maksimalt kan blive samplet, ved worst-case tilfældet hvor alle indgangene har en værdi større end 0. Det er vist de færreste administratorer, der kan bruge et diagram baseret på så få værdier til noget... Dertil kommer så pladsforbruget af DEerne i hver datastruktur. Jeg har ikke forsøgt, at optimere DEerne jeg bruger i mit program, men uanset hvor meget der optimeres, gør de i hvert fald ikke pladsforbruget mindre.

3. **De fastlåste samplinger.** Den sidste årsag til mit klare nej er måden, samplingen er låst på (afsnit 6.3). Hvis jeg ser kortvarigt bort fra problemerne i pkt. 1 og 2, så var min oprindelige ide, at en administrator f.eks. skulle kunne bede programmet om at bruge nok diskplads på, at gøre det muligt at opnå en sampling af 50% af de registrerede værdier. På den måde vil man ved problemer, der kan ses i et søjlediagram, have 50% chance for her og nu at kunne spotte årsagen. I perioder uden problem vil en sampling af kun 50% af søjlerne betyde, at administratoren ikke kan se alt, når han vil. Det gør dog ikke så meget, hvis valget af hvilke søjler der samples ændrer sig over tid (f.eks. i løbet af nogle timer). Så kan han da i det mindste over en længere periode opbygge sig et indtryk af den samlede overvågede trafik.

   Hvis dét, der derimod reelt sker i dette eksempel, er, at programmet ved initialiseringen af datastrukturen blot udvælger 50% af søjlerne tilfældigt, og så fremover viser disse til administratoren, så tror jeg, han bedre kunne have valgt dem selv. Når der ikke er problemer, er nogle IPers aktiviteter - f.eks. dem fra ens eget netværk - normalt mere interessante at overvåge end andres, og et uheldigt valg af datastrukturen ved initialisering kan derfor gøre samplingerne ubrugelige i disse perioder.

Disse tre grunde er i mine øjne hver for sig slemme nok, og sammen er de kun endnu værre.

7.2 I andre netværks sammenhænge

Jeg nævnte i kapitel 1, at jeg valgte [1] som udgangspunkt for specialet, da forfatterne i indledningen nævner data streams i netværk som et motiverende eksempel for artiklen. Helt konkret skrev de: ”A prominent example for such a data stream is the internet traffic at a backbone router. Assume we want to maintain some statistics about the routed packets.”([1, afsnit 1]). Denne del af artiklen er endda stadig bevaret i den reviderede udgave [2]. Det var på den baggrund, at jeg valgte at bruge datastrukturen til sampling af aktiviteten i en webservers log.

Når nu jeg ikke mener, at datastrukturen er brugbar i den sammenhæng, jeg har valgt, så har jeg spurgt mig selv, om jeg kan finde på en anden sammenhæng, hvor jeg kan se en god anvendelse. I modsætning til da jeg startede arbejde med [1] i begyndelsen af specialet, ved jeg jo nu, hvordan datastrukturen egentlig fungerer.

Mit svar er faktisk igen nej. Jeg kan ikke finde på noget i forbindelse med netværk, hvor jeg synes, datastrukturen vil være smart at bruge. Problemet er, at som jeg ser datastrukturen nu, så er den kun god i situationer, hvor man ud fra meget få samples, evt. kun én, kan sige noget om helheden af den samplede stream. Ved data streams i netværk er det jo netop dét faktum, at de enkelte data pakker ikke siger noget om helheden, der gør, at andre forslag til sampling som f.eks. [4] og [7] fokuserer på måder, at udvælge de dele af data streams man finder interessante.

Jeg kan selvfølgelig ikke fuldstændigt udelukke datastrukturens anvendelse i forbindelse med overvågning af data streams på netværk, men jeg har som sagt ingen ideer. At forfatterne har valgt at nævne eksemplet med en backbone router, ser jeg derfor nu mere, som at de bare har nævnt forskellige sammenhænge, sampling anvendes i, end at de har haft en konkret brug i tankerne.

En positiv ting, jeg dog har at sige om datastrukturen ved en evt. brug i overvågning på et netværk, er hastigheden, hvormed den kan opdateres. På trods af at mit program på ingen måde er forsøgt optimeret, så tog mine tests med 3 døgns intervaller (afsnit 6.3) kun nogle få minutter pr. interval for min computer at foretage. Omend jeg ikke ved, hvor mange opdateringer man kan foretage med den rette hardware og optimering, så ved jeg i hvert fald, at på en webserver svarende til den for 246.dk, ville den fint kunne følge med ved langt kraftigere belastninger. Dermed er datastrukturen bestemt ikke urealistisk at benytte - hvis man altså kan finde et problem, hvor få samplinger kan give svaret.

Kapitel 8

Konklusion

I dette speciale har jeg arbejdet med brugen af sampling af data streams. Omend sådan sampling kan være relevant i alle sammenhænge, der involverer data streams, har jeg fokuseret på, hvad man kan opnå med sampling af streams i netværk. De væsentligste punkter, jeg har opnået med dette arbejde, er:

- Jeg har undersøgt, hvad begrebet "data streams"egentlig dækker over, og hvordan de formelt kan beskrives. Ud fra dette har jeg set på, hvilken motivation man kan have for at observere sådanne streams i netværk, og hvilke problemer man i så fald står over for. Helt specifikt har jeg beskrevet det mest grundliggende problem: At en data stream potentielt er uendelig stor, og derfor aldrig vil kunne gemmes i sin helhed.

  Dette problem har jeg anført som den primære årsag til, at man er nødt til at sample data streams, så man kun gemmer de detaljer, man ønsker at bevare. Et væsentligt problem er dog, at forudse fremtidige krav til dét, der gemmes, da man ikke senere kan gå tilbage i tiden og se på den oprindelige stream igen. Det gælder derfor om, at det gemte data kan sige så meget som muligt om den oprindelige stream ved brug af så lidt plads som muligt. Det bringer datastrukturer til sampling på banen.

- Jeg har givet eksempler på forskellige anvendelser af sampling i netværks sammenhæng. Både teoretiske og praktiske. For de teoretiske har det mest handlet om, at man identificerer informative matematiske egenskaber ved data streams i netværk, og så laver datastrukturer, der ved minimalt pladsforbrug, er i stand til at estimere disse egenskaber. For de praktiske har det mere handlet om de fysiske problemer, man står over for, når et effektivt system til logning ved brug af sampling skal realiseres i praksis.

  Jeg har kun givet en overordnet beskrivelse af de eksempler, jeg har valgt, da det kun er meningen, at de skal give læseren et løst indblik i forskellige foreslåede ideer og teknikker til sampling.

- Jeg har gennemgået datastrukturen foreslået i [1] og efterfølgende revideret i [2]. Jeg har forsøgt med mine egne ord, at give en beskrivelse af problemet datastrukturen skal løse, ideerne der ligger til grund for den og hvordan forfatterne af [1] har lavet den, så den løser problemet. Jeg har lavet gennemgangen ud fra en praktisk synsvinkel, og har forsøgt at fokusere på de dele af datastrukturen, der volder problemer under en implementering.

- Jeg har valgt en praktisk anvendelse af datastrukturen, løst baseret på hvad der i [1] er foreslået den kan bruges til. Jeg har så implementeret et program, der simulerer denne anvendelse. På baggrund af resultaterne med programmet

har jeg givet min vurdering af datastrukturens praktiske værdi i henholdsvis den valgte anvendelse og i overvågning af netværk generelt.

Sammenholder jeg disse punkter med mine oprindelige mål for specialet (se kapitel 1), vil jeg mene, at jeg har opnået disse mål. Der har dog været en del justeringer undervejs af især min plan for den praktiske anvendelse. Det er en naturlig følge af, at i takt med at jeg har fået større indblik i, hvordan datastrukturen fra [1] fungerer, så har jeg også løbende forsøgt at justere min anvendelse, så jeg syntes, den passede bedst muligt til datastrukturen.

Som det kan ses i kapitel 7, endte jeg med at være ret negativ i min vurdering af datastrukturens potentiale ved overvågningen af data streams i netværk. Her ved slutningen af specialet er jeg på den ene side selvfølgelig lidt ærgelig over, at jeg ikke endte op med en god ny form for anvendelse af datastrukturen. På den anden side gør det mig ikke så meget, da jeg stadig ikke kan se, hvad der kan gøres anderledes, for at datastrukturen kan blive brugbar i netværks sammehæng.

Dét eneste, jeg set i bakspejlet ville ønske, var anderledes, er, at jeg ville ønske, jeg havde fundet den reviderede artikel [2] langt tidligere. Det irriterer mig, at jeg har brugt en stor del af mit arbejde med at få datastrukturen i [1] til at fungere, for blot efterfølgende at finde ud af at [2] gør dette arbejde meget nemmere. Omend det har betydet, at jeg nu ved, hvordan datastrukturen fungerer helt præcist, så har prisen været, at jeg har brugt mindre tid på at kigge på andre forslag til sampling, end jeg oprindeligt havde planlagt.

Når jeg kigger tilbage på de beslutninger, jeg har truffet i arbejdet med specialet, er der ikke nogen, jeg umiddelbart synes, jeg kunne have truffet anderledes, og på den baggrund - og med de opnåede resultater - er jeg tilfreds med resultatet af specialet.

Bilag A

Vejledning til mit program

A.1 Kompilering

På den medfølgende CD er mit programs kildekode placeret i mappen *Source*. Da programmet er skrevet i Microsofts C# kan det vist kun kompileres med Microsoft Visual Stuio 2005 (udgaven jeg har brugt). Jeg har derfor inkluderet projekt filen *SpecialeKode.csproj* der skulle gøre en evt. kompilering mulig uden videre.

A.2 Kørsel af programmet

Programmet er kun testet på en computer med Windows XP SP2 og alle nyeste opdatering. Det er desuden nødvendigt at den nyeste udgave af .Net platformen er installeret for at programmer skrevet i C# i det hele taget kan eksekveres. .Net kan downloades frit via Microsofts side *http://windowsupdate.microsoft.com*.

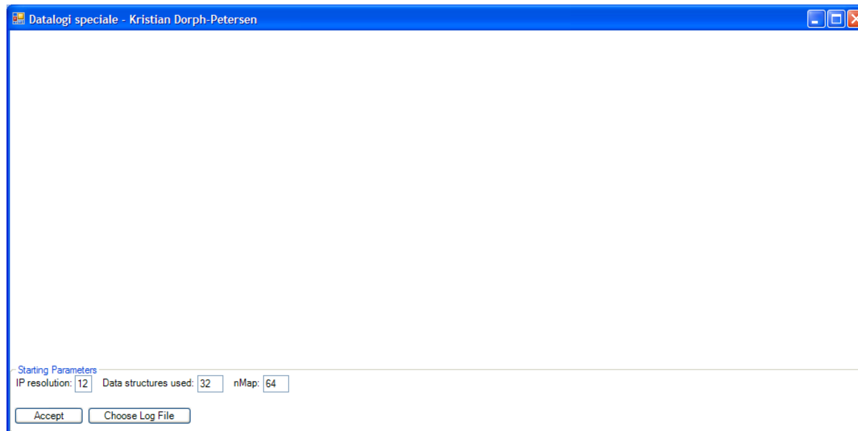
På den vedlagte CD i mappen *Executable* er placeret en fil *SpecialeKode.exe* der kører programmet. Filen er kompileret i Visual Studio som en "Release"udgave. Opfylder computeren man sidder ved de ovenfor nævnte krav skulle denne fil kunne bruges umiddelbart.

A.3 Brugervejledning

Bemærk! Programmet laver kun simple check på om de parametre der angives er brugbare. I dette afsnit angiver jeg hvilke værdier der er testet med. Ved valg af værdier udover dette kan det derfor være programmet crasher uden varsel.

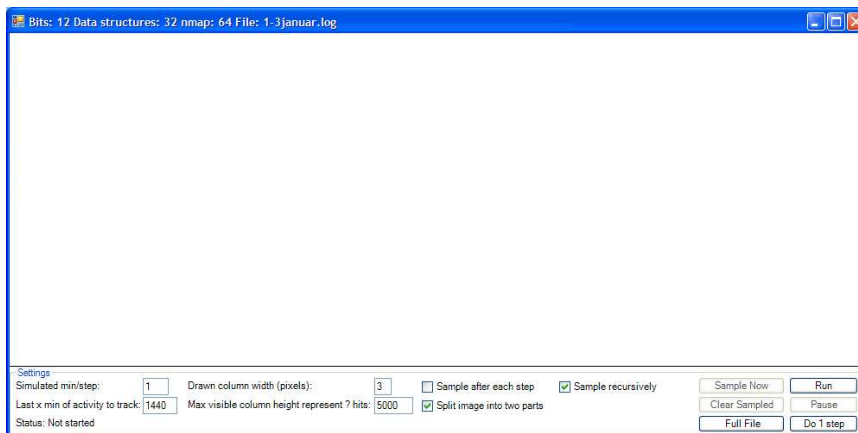
Når programmet startes ser det ud som vist i figur A.1. Her sættes de indledende parametre for en simulering i programmet De forskellige muligheder her er:

- "IP resolution". Sætter antallet af indledende bits i IP adresserne der skal opdeles ud fra. Programmet er testet med valg op til 16.
- "Data structures used". Angiv antallet af parallelle datastrukturer der skal benyttes.
- "nMap". Angiver værdien af nMap parameteren i DE strukturerne. De fleste tests kun kørt med nMap = 64.
- "Choose Log File". Giver mulighed for at vælge log filen der skal arbejdes på. Kun testet på log filerne placeret på CDen i mappen *Logs*.



Figur A.1: Screenshot - Programmet ved start

- "Accept". Accepterer de valgte parametre. Har man ikke specifikt valgt en log fil antages det at simuleringen skal ske på en fil *1-3januar.log* placeret i samme mappe som den eksekverbare fil køres fra.



Figur A.2: Screenshot - Programmet efter tryk på Accept

Efter accept af det indledende billedes parametre ændres programmets interface til det i figur A.2 viste. Mulighederne her er:

- "Simulated min/step". Angiver antallet af simulerede minutter ét "step"skal svare til i loggen. Der er kun mellem disse steps at der er mulighed for at f.eks. pause en simulering så ved store værdier for denne parameter kan programmet i perioder være frossent.
- "Last x min of activity to track". Angiver antallet af simulerede minutter der skal gå førend registreret data fjernes fra datastrukturerne. Dvs. angiver størrelsen af sliding window i loggen. Sæt denne værdi til 0 hvis den registrerede aktivitet aldrig skal fjernes.
- "Drawn column width (pixels)". Angiver bredden af søjlerne der tegnes i pixels.
- "Max visible column height represent ? hits". Angiver hvor stor en værdi den maksimale søjlehøjde der tegnes skal svare til.

- ”Sample after each step”. Angiver om der efter hvert simuleret step automatisk skal foretages en sampling.
- ”Split image into two parts”. Angiver om de to tegnede diagrammer skal tegnes med hver sin x-akse eller to forskellige. Når de tegnes med hver sin vises det reelle diagram øverst og det samplede nederst. Når de tegnes på samme akse er søjlerne der repræsenterer det reelle diagram grønne og de samplede røde.
- ”Sample recursively”. Angiver om programmet skal bruge rekursiv sampling. Er dette slået fra samples hver af de parallelle datastrukturer kun én gang.
- ”Run”. Starter simuleringen. Det reelle diagram tegnes efter hvert step.
- ”Pause”. Pauser en kørende simulering.
- ”Do 1 step”. Kører simuleringen ét step frem og pauser så automatisk igen.
- ”Sample now”. Foretager en sampling og tegner diagrammet dette giver. Hvis en tidligere sampling allerede er vist vil den blive fjernet.
- ”Clear sampled”. Fjerner det samplede diagram.
- ”Full file”. Kører simuleringen på hele den valgte fil med sliding window automatisk slået fra. Er blevet brugt i forbindelse med test kørslerne.

Bilag B

PCSA pseudokode

Fra [13, s. 197].

```
program PCSA;

const nmap = 64; {with nmap = 64, accuracy is typically 10%}
    {nmap corresponds to variable m in the analysis}
    phi = 0.77351 {the magic constant}; maxlength = 32;
    {with maxlength = 32 (==L), one can count up to 10^8}

var M: multiset of data of type records;
    x: records; hashedx, index, alpha, R, S, Xi: integer;
    BITMAPS: array[0..nmap-1; 0..maxlength-1] of integer;

function getelement(var x.records);
    {reads an element x of type records from file M}
function hash(x.records).integer;
    {hashes arecord x into an integer over scalar range [0..2^maxlength -1]}
function p(y.integer).integer;
    {returns the position of the first 1-bit in y; ranks start at 0.}

begin
    while not eof(M) do
    begin
        getelement(x); hashedx := hash(x);
        alpha := hashedx mod nmap; index := p(hashedx div nmap);
        if BITMAP[alpha, index] = 0 then BITMAP[alpha, index] := 1;
    end;

    S = 0;
    for i := 0 to nmap-1 do
    begin
        R := 0;
        while (BITMAP[i,n]=1) and (R<maxlength) do R := R+1;
        S := S+R;
    end;

    Xi := trunc((nmap/phi)*2^(S/nmap));
    {Result Xi of the PCSA programme that estimates n}
end
```

Bilag C

Kode i *MainForm.cs*

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace SpecialeKode {
    // Code to start a C# program
    class cProgramStarter
    {
        [STAThread]
        public static void Main()
        {
            Application.EnableVisualStyles();
            Application.DoEvents();
            Application.Run(new cMainForm());
        }
    }

    // Main Window for project
    class cMainForm : Form
    {
        // Name of the log-file being analysed
        private string LogFile = "1-3januar.log";

        // Variable to check if analysis should be running or not
        private bool Running = false;

        // The elements of the window
        private GroupBox gbInitial = new GroupBox();
        private GroupBox gbSettings = new GroupBox();
        private Label lbIPRes = new Label();
        private Label lbnDataStructs = new Label();
        private Label lbnMap = new Label();
        private Label lbPrTick = new Label();
        private Label lbRemoveMin = new Label();
        private Label lbColumnWidth = new Label();
        private Label lbYaxisUp = new Label();
        private Label lbStatus = new Label();
        private TextBox tbIPRes = new TextBox();
        private TextBox tbnDataStructs = new TextBox();
        private TextBox tbnMap = new TextBox();
        private TextBox tbPrTick = new TextBox();
        private TextBox tbRemoveMin = new TextBox();
        private TextBox tbYaxisUp = new TextBox();
        private TextBox tbColumnWidth = new TextBox();
        private CheckBox cbSample = new CheckBox();
        private CheckBox cbDrawSeperate = new CheckBox();
        private CheckBox cbRecursive = new CheckBox();
        private Button btStart = new Button();
        private Button btLogFile = new Button();
        private Button btRun = new Button();
        private Button btPause = new Button();
        private Button btOneStep = new Button();
        private Button btFullFile = new Button();
        private Button btSampleNow = new Button();
        private Button btClearSampling = new Button();

        // The panel to be drawn upon
        private cImagePanel ImagePanel = new cImagePanel();
```

```
// The analyse controller
private cAnalyseController Analyser = null;

public cMainForm()
{
    // Initial setup of program window
    this.Text = "Datalogi speciale - Kristian Dorph-Petersen";
    this.Width = 1000;
    this.Height = 500;

    // Groupbox and elements for initial setup of analysis
    gbInitial.Parent = this;
    gbInitial.Height = 80;
    gbInitial.Text = "Starting Parameters";
    gbInitial.Dock = DockStyle.Bottom;
    gbInitial.Visible = true;
    gbInitial.BackColor = System.Drawing.Color.White;

    lbIPRes.Parent = gbInitial;
    lbIPRes.Top = 15;
    lbIPRes.Left = 5;
    lbIPRes.AutoSize = true;
    lbIPRes.Text = "IP resolution:";

    tbIPRes.Parent = gbInitial;
    tbIPRes.Top = lbIPRes.Top - 2;
    tbIPRes.Left = lbIPRes.Right;
    tbIPRes.Width = 20;
    tbIPRes.MaxLength = 2;
    tbIPRes.Text = "12";

    lbnDataStructs.Parent = gbInitial;
    lbnDataStructs.Top = lbIPRes.Top;
    lbnDataStructs.Left = tbIPRes.Right + 10;
    lbnDataStructs.AutoSize = true;
    lbnDataStructs.Text = "Data structures used:";

    tbnDataStructs.Parent = gbInitial;
    tbnDataStructs.Top = lbIPRes.Top - 2;
    tbnDataStructs.Left = lbnDataStructs.Right;
    tbnDataStructs.Width = 30;
    tbnDataStructs.MaxLength = 3;
    tbnDataStructs.Text = "32";

    lbnMap.Parent = gbInitial;
    lbnMap.Top = lbIPRes.Top;
    lbnMap.Left = tbnDataStructs.Right + 10;
    lbnMap.AutoSize = true;
    lbnMap.Text = "nMap:";

    tbnMap.Parent = gbInitial;
    tbnMap.Top = lbIPRes.Top - 2;
    tbnMap.Left = lbnMap.Right;
    tbnMap.Width = 30;
    tbnMap.MaxLength = 4;
    tbnMap.Text = "64";

    btStart.Parent = gbInitial;
    btStart.Enabled = true;
    btStart.Top = 50;
    btStart.Left = 5;
    btStart.Width = 80;
    btStart.Height = 20;
    btStart.Text = "Accept";
    btStart.Click += new EventHandler(btStart_Click);

    btLogFile.Parent = gbInitial;
    btLogFile.Enabled = true;
    btLogFile.Top = btStart.Top;
    btLogFile.Left = btStart.Right+5;
    btLogFile.Width = 120;
    btLogFile.Height = 20;
    btLogFile.Text = "Choose Log File";
    btLogFile.Click += new EventHandler(btLogFile_Click);

    // Groupbox and elements for controls available during analysis
    gbSettings.Parent = this;
    gbSettings.Height = gbInitial.Height;
    gbSettings.Text = "Settings";
```

```
gbSettings.Dock = DockStyle.Bottom;
gbSettings.Visible = false;
gbSettings.BackColor = System.Drawing.Color.White;

lbPrTick.Parent = gbSettings;
lbPrTick.Top = 15;
lbPrTick.Left = 5;
lbPrTick.AutoSize = true;
lbPrTick.Text = "Simulated min/step:";

tbPrTick.Parent = gbSettings;
tbPrTick.Top = lbPrTick.Top - 2;
tbPrTick.Width = 30;
tbPrTick.MaxLength = 6;
tbPrTick.Text = "1";

lbRemoveMin.Parent = gbSettings;
lbRemoveMin.Top = lbPrTick.Bottom + 5;
lbRemoveMin.Left = lbPrTick.Left;
lbRemoveMin.AutoSize = true;
lbRemoveMin.Text = "Last x min of activity to track:";

tbRemoveMin.Parent = gbSettings;
tbRemoveMin.Top = lbRemoveMin.Top - 2;
tbRemoveMin.Left = lbRemoveMin.Right;
tbRemoveMin.Width = 40;
tbRemoveMin.MaxLength = 5;
tbRemoveMin.Text = "1440";
tbPrTick.Left = tbRemoveMin.Left;

lbColumnWidth.Parent = gbSettings;
lbColumnWidth.Top = lbPrTick.Top;
lbColumnWidth.Left = tbRemoveMin.Right + 10;
lbColumnWidth.AutoSize = true;
lbColumnWidth.Text = "Drawn column width (pixels):";

tbColumnWidth.Parent = gbSettings;
tbColumnWidth.Top = lbColumnWidth.Top - 2;
tbColumnWidth.Width = 20;
tbColumnWidth.MaxLength = 2;
tbColumnWidth.Text = "3";

lbYaxisUp.Parent = gbSettings;
lbYaxisUp.Top = lbRemoveMin.Top;
lbYaxisUp.Left = lbColumnWidth.Left;
lbYaxisUp.AutoSize = true;
lbYaxisUp.Text = "Max visible column height represent ? hits:";

tbYaxisUp.Parent = gbSettings;
tbYaxisUp.Top = lbYaxisUp.Top - 2;
tbYaxisUp.Left = lbYaxisUp.Right;
tbYaxisUp.Width = 45;
tbYaxisUp.MaxLength = 6;
tbYaxisUp.Text = "5000";
tbColumnWidth.Left = tbYaxisUp.Left;

cbSample.Parent = gbSettings;
cbSample.Top = lbColumnWidth.Top;
cbSample.Left = tbYaxisUp.Right + 10;
cbSample.Checked = false;
cbSample.Text = "Sample after each step";
cbSample.AutoSize = true;

cbDrawSeperate.Parent = gbSettings;
cbDrawSeperate.Top = lbYaxisUp.Top;
cbDrawSeperate.Left = cbSample.Left;
cbDrawSeperate.Checked = true;
cbDrawSeperate.Text = "Split image into two parts";
cbDrawSeperate.AutoSize = true;

cbRecursive.Parent = gbSettings;
cbRecursive.Top = cbSample.Top;
cbRecursive.Left = cbDrawSeperate.Right + 10;
cbRecursive.Checked = true;
cbRecursive.Text = "Sample recursively";
cbRecursive.AutoSize = true;

lbStatus.Parent = gbSettings;
lbStatus.Top = lbRemoveMin.Bottom + 4;
lbStatus.Left = lbRemoveMin.Left;
```

```
lbStatus.AutoSize = true;
lbStatus.Text = "Status: Not started";

// Button to start running analysis
btRun.Parent = gbSettings;
btRun.Enabled = true;
btRun.Top = 12;
btRun.Left = this.ClientRectangle.Width - 90;
btRun.Width = 80;
btRun.Height = 20;
btRun.Text = "Run";
btRun.Anchor = AnchorStyles.Right;
btRun.Click += new EventHandler(btRun_Click);

// Button to pause analysis
btPause.Parent = gbSettings;
btPause.Enabled = false;
btPause.Top = btRun.Top + 22;
btPause.Left = btRun.Left;
btPause.Width = 80;
btPause.Height = 20;
btPause.Text = "Pause";
btPause.Anchor = AnchorStyles.Right;
btPause.Click += new EventHandler(btPause_Click);

// Button to simulate 1 step
btOneStep.Parent = gbSettings;
btOneStep.Enabled = true;
btOneStep.Top = btPause.Top + 22;
btOneStep.Left = btPause.Left;
btOneStep.Width = 80;
btOneStep.Height = 20;
btOneStep.Text = "Do 1 step";
btOneStep.Anchor = AnchorStyles.Right;
btOneStep.Click += new EventHandler(btOneStep_Click);

// Button to save current results to file
btSampleNow.Parent = gbSettings;
btSampleNow.Enabled = false;
btSampleNow.Top = btRun.Top;
btSampleNow.Left = btRun.Left - 105;
btSampleNow.Width = 100;
btSampleNow.Height = 20;
btSampleNow.Text = "Sample Now";
btSampleNow.Anchor = AnchorStyles.Right;
btSampleNow.Click += new EventHandler(btSampleNow_Click);

// Button to save current results to file
btClearSampling.Parent = gbSettings;
btClearSampling.Enabled = false;
btClearSampling.Top = btPause.Top;
btClearSampling.Left = btPause.Left - 105;
btClearSampling.Width = 100;
btClearSampling.Height = 20;
btClearSampling.Text = "Clear Sampled";
btClearSampling.Anchor = AnchorStyles.Right;
btClearSampling.Click += new EventHandler(btClearSampling_Click);

// Button for full file analysis
btFullFile.Parent = gbSettings;
btFullFile.Enabled = true;
btFullFile.Top = btOneStep.Top;
btFullFile.Left = btOneStep.Left - 105;
btFullFile.Width = 100;
btFullFile.Height = 20;
btFullFile.Text = "Full File";
btFullFile.Anchor = AnchorStyles.Right;
btFullFile.Click += new EventHandler(btFullFile_Click);

// The image panel that contains the graphics
ImagePanel.Parent = this;
ImagePanel.Top = 0;
ImagePanel.Left = 0;
ImagePanel.Width = this.ClientRectangle.Width;
ImagePanel.Height = gbSettings.Top;
ImagePanel.Anchor = AnchorStyles.Bottom | AnchorStyles.Left
                    | AnchorStyles.Top | AnchorStyles.Right;

// Code to handle resize of window
this.Resize += new EventHandler(cMainForm_Resize);
```

```
}

void btLogFile_Click(object sender, EventArgs e)
{
    // Display file chooser dialog
    OpenFileDialog OFD = new OpenFileDialog();

    OFD.Title = "Select a log file";
    OFD.Filter = "Log Files|*.log";
    OFD.InitialDirectory = Application.StartupPath;

    if (OFD.ShowDialog() == DialogResult.OK)
    {
        LogFile = OFD.FileName;
    };
}

void btClearSampling_Click(object sender, EventArgs e)
{
    Analyser.ClearSampling();
    ImagePanel.Invalidate();
}

void btSampleNow_Click(object sender, EventArgs e)
{
    // Register display variables in case they've changed
    int MaxYaxis;
    int ColumnWidth;

    try
    {
        MaxYaxis = Convert.ToInt32(tbYaxisUp.Text);
        ColumnWidth = Convert.ToInt32(tbColumnWidth.Text);
    }
    catch { return; };

    // Check values acceptable
    if (MaxYaxis < 1 || ColumnWidth < 1) { return; };

    ImagePanel.RegisterVariables(ColumnWidth, MaxYaxis, cbDrawSeperate.Checked);

    string SampleResult = Analyser.SampleData(cbRecursive.Checked);

    // Display results for sampling
    ImagePanel.Invalidate();
    lbStatus.Text = SampleResult;
}

void btOneStep_Click(object sender, EventArgs e)
{
    // Enable/disable controls as needed (only important if used before Run-button)
    btPause.Enabled = false;
    btFullFile.Enabled = false;

    btRun.Enabled = true;
    btOneStep.Enabled = true;
    btSampleNow.Enabled = true;
    btClearSampling.Enabled = true;
    tbPrTick.Enabled = true;
    tbYaxisUp.Enabled = true;
    tbColumnWidth.Enabled = true;
    cbDrawSeperate.Enabled = true;
    cbSample.Enabled = true;
    cbRecursive.Enabled = true;

    // Check if variables entered correctly
    int MinPrTick;
    int RemoveAfterMin;
    int MaxYaxis;
    int ColumnWidth;

    try
    {
        MinPrTick = Convert.ToInt32(tbPrTick.Text);
        RemoveAfterMin = Convert.ToInt32(tbRemoveMin.Text);
        MaxYaxis = Convert.ToInt32(tbYaxisUp.Text);
        ColumnWidth = Convert.ToInt32(tbColumnWidth.Text);
    }
    catch { return; };
```

```
    // Check values acceptable
    if (MinPrTick < 1 || RemoveAfterMin < 0 || MaxYaxis < 1 || ColumnWidth < 1) { return; };

    // Register drawing values
    ImagePanel.RegisterVariables(ColumnWidth, MaxYaxis, cbDrawSeperate.Checked);

    // Run the simulation once
    RunAnalysis(MinPrTick, RemoveAfterMin, 1);
}

void btRun_Click(object sender, EventArgs e)
{
    // Check if variables entered correctly
    int MinPrTick;
    int RemoveAfterMin;
    int MaxYaxis;
    int ColumnWidth;

    try
    {
        MinPrTick = Convert.ToInt32(tbPrTick.Text);
        RemoveAfterMin = Convert.ToInt32(tbRemoveMin.Text);
        MaxYaxis = Convert.ToInt32(tbYaxisUp.Text);
        ColumnWidth = Convert.ToInt32(tbColumnWidth.Text);
    }
    catch { return; };

    // Check values acceptable
    if (MinPrTick < 1 || RemoveAfterMin < 0 || MaxYaxis < 1 || ColumnWidth < 1) { return; };

    // Enable/disable controls as needed
    btRun.Enabled = false;
    btOneStep.Enabled = false;
    btFullFile.Enabled = false;
    btSampleNow.Enabled = false;
    btClearSampling.Enabled = false;
    tbPrTick.Enabled = false;
    tbRemoveMin.Enabled = false;
    tbYaxisUp.Enabled = false;
    tbColumnWidth.Enabled = false;
    cbDrawSeperate.Enabled = false;
    cbSample.Enabled = false;
    cbRecursive.Enabled = false;

    btPause.Enabled = true;

    // Register drawing values
    ImagePanel.RegisterVariables(ColumnWidth, MaxYaxis, cbDrawSeperate.Checked);

    // Run the simulation
    RunAnalysis(MinPrTick, RemoveAfterMin, 0);
}

void RunAnalysis(int MinPrTick, int RemoveAfterMin, int nTicks)
{
    // Run simulation nTicks times, until pause pressed if nTicks = 0 or EOF in logfile
    bool NotDone = true;
    Running = true;
    int i = 0;

    lbStatus.Text = "Status: Running! Please wait...";
    while (Running && NotDone && (nTicks == 0 || i < nTicks))
    {
        NotDone = Analyser.Analyse(MinPrTick, RemoveAfterMin);
        if (cbSample.Checked) { Analyser.SampleData(cbRecursive.Checked); };
        ImagePanel.Invalidate();
        Application.DoEvents();
        lbStatus.Text = "Status: Running! Examined " + Analyser.nAnalysed
            + "/6078051 entries, have met " + Analyser.nErrorEntries
            + " errors. Simulated time is " + Analyser.SimulatedTime;
        i++;
    };

    if (!NotDone) { lbStatus.Text = "Status: Done! Examined " + Analyser.nAnalysed
                    + "/6078051 entries, have met " + Analyser.nErrorEntries + " errors."; }
    else { lbStatus.Text = "Status: Paused! Examined " + Analyser.nAnalysed
           + "/6078051 entries, have met " + Analyser.nErrorEntries
           + " errors. Simulated time is " + Analyser.SimulatedTime; };
}
```

```
void cMainForm_Resize(object sender, EventArgs e)
{
    ImagePanel.Invalidate();
}

void btStart_Click(object obj, EventArgs ea)
{
    // Check for number in textboxes
    int BitResolution;
    int nDataStructs;
    int nmap;

    try
    {
        BitResolution = Convert.ToInt32(tbIPRes.Text);
        nDataStructs = Convert.ToInt32(tbnDataStructs.Text);
        nmap = Convert.ToInt32(tbnMap.Text);
    }
    catch { return; };

    // Check for usable value of BitResolution
    if (BitResolution < 1 || BitResolution > 31) { return; };

    // Initialise analysis controller and show the simulation controls
    gbInitial.Visible = false;
    gbSettings.Visible = true;
    Analyser = new cAnalyseController(LogFile, BitResolution, nDataStructs, nmap);

    // Register the dataclasses to be drawn
    ImagePanel.RegisterTrackers(Analyser.NormalTracker, Analyser.SamplerValues);

    // Display chosen parameters in title
    this.Text = "Bits: " + BitResolution + " Data structures: " + nDataStructs
        + " nmap: " + nmap+ " File: "+LogFile;
}

void btPause_Click(object obj, EventArgs ea)
{
    Running = false;

    // Enable/disable controls as needed
    btPause.Enabled = false;

    btRun.Enabled = true;
    btOneStep.Enabled = true;
    btSampleNow.Enabled = true;
    btClearSampling.Enabled = true;
    tbPrTick.Enabled = true;
    tbYaxisUp.Enabled = true;
    tbColumnWidth.Enabled = true;
    cbDrawSeperate.Enabled = true;
    cbSample.Enabled = true;
    cbRecursive.Enabled = true;
}

void btFullFile_Click(object obj, EventArgs ea)
{
    // Check if variables entered correctly
    int MaxYaxis;
    int ColumnWidth;

    try
    {
        MaxYaxis = Convert.ToInt32(tbYaxisUp.Text);
        ColumnWidth = Convert.ToInt32(tbColumnWidth.Text);
    }
    catch { return; };

    // Check values acceptable
    if (MaxYaxis < 1 || ColumnWidth < 1) { return; };

    // Enable/disable controls as needed
    btRun.Enabled = false;
    btOneStep.Enabled = false;
    btFullFile.Enabled = false;
    btSampleNow.Enabled = false;
    btClearSampling.Enabled = false;
    tbPrTick.Enabled = false;
    tbRemoveMin.Enabled = false;
    tbYaxisUp.Enabled = false;
```

```
            tbColumnWidth.Enabled = false;
            cbDrawSeperate.Enabled = false;
            cbSample.Enabled = false;
            cbRecursive.Enabled = false;
            btPause.Enabled = false;

            // Register drawing values
            ImagePanel.RegisterVariables(ColumnWidth, MaxYaxis, cbDrawSeperate.Checked);

            // Run the simulation
            lbStatus.Text = "Full file analysis in progress... May take a while depending on file size.";
            Application.DoEvents();
            Analyser.FullFileAnalysis();
            ImagePanel.Invalidate();
            lbStatus.Text = "Full file analysis DONE! Ready for sampling.";

            // Analysis done at this point, enable needed buttons
            tbYaxisUp.Enabled = true;
            tbColumnWidth.Enabled = true;
            btSampleNow.Enabled = true;
            btClearSampling.Enabled = true;
            cbDrawSeperate.Enabled = true;
            cbRecursive.Enabled = true;
        }
    }
}
```

Bilag D

Kode i *DataTypes.cs*

```
using System;
using System.IO;
using System.Windows.Forms;
using System.Collections;

namespace SpecialeKode {
    // Class to contain the needed information from an Apache Access Log entry
    class cApacheLogEntry
    {
        public uint RequesterIP = 0; // Requester address
        public DateTime EntryTime; // Time of request
        public bool NoError = true;

        public cApacheLogEntry(string IP, string Time)
        {
            string tmpStr;
            int i = 0;

            try
            {
                // IP in format xxx.xxx.xxx.xxx - parse to uint
                IP += ".";
                tmpStr = "";
                while (IP[i] != '.')
                {
                    tmpStr += IP[i];
                    i++;
                };
                RequesterIP = Convert.ToUInt32(tmpStr) << 24;

                i++;
                tmpStr = "";
                while (IP[i] != '.')
                {
                    tmpStr += IP[i];
                    i++;
                };
                RequesterIP += Convert.ToUInt32(tmpStr) << 16;

                i++;
                tmpStr = "";
                while (IP[i] != '.')
                {
                    tmpStr += IP[i];
                    i++;
                };
                RequesterIP += Convert.ToUInt32(tmpStr) << 8;

                i++;
                tmpStr = "";
                while (IP[i] != '.')
                {
                    tmpStr += IP[i];
                    i++;
                };
                RequesterIP += Convert.ToUInt32(tmpStr);

                // Entry time in format 31/Dec/2005:20:30:20 - parse to DateTime
```

```
Time += ":";
int Day;
int Month = 1;
int Year;
int Hour;
int Min;
int Sec;

i = 0;
tmpStr = "";
while (Time[i] != '/')
{
    tmpStr += Time[i];
    i++;
};
Day = Convert.ToInt32(tmpStr);

i++;
tmpStr = "";
while (Time[i] != '/')
{
    tmpStr += Time[i];
    i++;
};
switch (tmpStr)
{
    case "Jan":
        Month = 1;
        break;
    case "Feb":
        Month = 2;
        break;
    case "Mar":
        Month = 3;
        break;
    case "Apr":
        Month = 4;
        break;
    case "May":
        Month = 5;
        break;
    case "Jun":
        Month = 6;
        break;
    case "Jul":
        Month = 7;
        break;
    case "Aug":
        Month = 8;
        break;
    case "Sep":
        Month = 9;
        break;
    case "Oct":
        Month = 10;
        break;
    case "Nov":
        Month = 11;
        break;
    case "Dec":
        Month = 12;
        break;
};

i++;
tmpStr = "";
while (Time[i] != ':')
{
    tmpStr += Time[i];
    i++;
};
Year = Convert.ToInt32(tmpStr);

i++;
tmpStr = "";
while (Time[i] != ':')
{
    tmpStr += Time[i];
    i++;
};
```

```
            Hour = Convert.ToInt32(tmpStr);

            i++;
            tmpStr = "";
            while (Time[i] != ':')
            {
                tmpStr += Time[i];
                i++;
            };
            Min = Convert.ToInt32(tmpStr);

            i++;
            tmpStr = "";
            while (Time[i] != ':')
            {
                tmpStr += Time[i];
                i++;
            };
            Sec = Convert.ToInt32(tmpStr);

            EntryTime = new DateTime(Year, Month, Day, Hour, Min, Sec);
        }
        catch
        {
            NoError = false;
        };
    }
}

// Class to handle the reading of the log file
class cLogFileReader
{
    private FileStream fsLog = null;

    public cLogFileReader(string LogFile)
    {
        // Open the log file in readonly mode
        try { fsLog = new FileStream(LogFile, FileMode.Open, FileAccess.Read); }
        catch (FileNotFoundException) { }
        catch { };
    }

    public cApacheLogEntry GetNextEntry()
    {
        string IP = "";
        string EntryTime = "";
        int tmpByte;

        // Move forward to IP of requester
        while ((tmpByte = fsLog.ReadByte()) != -1 && Convert.ToChar(tmpByte) != ' ') { };
        if (-1 == tmpByte) { return null; };

        // Get the IP
        while ((tmpByte = fsLog.ReadByte()) != -1 && Convert.ToChar(tmpByte) != ' ')
        {
            IP += Convert.ToString(Convert.ToChar(tmpByte));
        };
        if (-1 == tmpByte) { return null; };

        // Move forward to date and time
        while ((tmpByte = fsLog.ReadByte()) != -1 && Convert.ToChar(tmpByte) != '[') { };
        if (-1 == tmpByte) { return null; };

        // Get the time
        while ((tmpByte = fsLog.ReadByte()) != -1 && Convert.ToChar(tmpByte) != ' ')
        {
            EntryTime += Convert.ToString(Convert.ToChar(tmpByte));
        };
        if (-1 == tmpByte) { return null; };

        // Move forward to next entry so ready for next request
        while ((tmpByte = fsLog.ReadByte()) != -1 && Convert.ToChar(tmpByte) != '\n') { };

        // Create and return the read entry
        return new cApacheLogEntry(IP, EntryTime);
    }
}

// Class controlling the running of the analysis
class cAnalyseController
```

```
{
    private Queue Waiting = new Queue();
    private cLogFileReader LogFile;
    private cApacheLogEntry LogEntry;
    private cApacheLogEntry OldEntry = null;

    private cSamplingController SampCont;
    public cSamplerValues SamplerValues;
    public cNormalTracker NormalTracker;

    public int nAnalysed = 0;
    public int nErrorEntries = 0;
    public DateTime SimulatedTime;

    // Class creator
    public cAnalyseController(string Log, int nBit, int nTrackers, int nmap)
    {
        LogFile = new cLogFileReader(Log);
        NormalTracker = new cNormalTracker(nBit);
        SampCont = new cSamplingController(nBit, nTrackers, nmap);
        SamplerValues = new cSamplerValues(nBit);

        LogEntry = LogFile.GetNextEntry();
        if (LogEntry != null) { SimulatedTime = new DateTime(LogEntry.EntryTime.Ticks); };
    }

    public bool Analyse(int MinPrTick, int RemoveTime)
    {
        int i = 0;
        SimulatedTime = SimulatedTime.AddMinutes(MinPrTick);

        while ((LogEntry != null) && (SimulatedTime > LogEntry.EntryTime))
        {
            if (LogEntry.NoError)
            {
                // Force check on any user interactions if 1000 entries has passed
                // with no interface updates
                if (i == 1000)
                {
                    Application.DoEvents();
                    i = 0;
                };

                if (RemoveTime > 0)
                {
                    if ((OldEntry == null) && (Waiting.Count > 0))
                        { OldEntry = (cApacheLogEntry)Waiting.Dequeue(); };
                    while ((OldEntry != null) && ((OldEntry.EntryTime < LogEntry.EntryTime)))
                    {
                        NormalTracker.RecordDel(OldEntry.RequesterIP);
                        SampCont.RecordDel(OldEntry.RequesterIP);
                        if (Waiting.Count > 0) { OldEntry = (cApacheLogEntry)Waiting.Dequeue(); }
                        else { OldEntry = null; };
                    };
                    LogEntry.EntryTime = LogEntry.EntryTime.AddMinutes(RemoveTime);
                    Waiting.Enqueue(LogEntry);
                };
                NormalTracker.RecordHit(LogEntry.RequesterIP);
                SampCont.RecordHit(LogEntry.RequesterIP);
            }
            else { nErrorEntries++; };

            nAnalysed++;
            LogEntry = LogFile.GetNextEntry();

            i++;
        };

        if (LogEntry != null) { return true; }
        else { return false; };
    }

    public void FullFileAnalysis()
    {
        // Run as long as there is entries in file
        while (LogEntry != null)
        {
            int i = 0;
            if (LogEntry.NoError)
            {
```

```
                    // Force check on any user interactions if 1000 entries has passed
                    // with no interface updates
                    if (i == 1000)
                    {
                        Application.DoEvents();
                        i = 0;
                    };

                    NormalTracker.RecordHit(LogEntry.RequesterIP);
                }
                else { nErrorEntries++; };

                nAnalysed++;
                LogEntry = LogFile.GetNextEntry();
                i++;
            };

            // File done, put registered data in sampling datastructure
            for (int j = 0; j < NormalTracker.HitTracker.Length; j++)
            {
                if (NormalTracker.HitTracker[j] > 0)
                {
                    SampCont.RecordValue(j, Convert.ToInt32(NormalTracker.HitTracker[j]));
                };
            };
        }

        public string SampleData(bool DoRecursively)
        {
            // Clear previous sampled values and do new sampling
            SamplerValues.ClearData();
            SampCont.GetSamples(SamplerValues, DoRecursively);

            // Construct string with result from sampling
            float RealVals = NormalTracker.nValues();
            float SampledVals = SamplerValues.nValues();
            float Found = 100 * (SampledVals / RealVals);

            return "Real values: " + RealVals + " Sampled values: " + SampledVals
                + " Percentage found: " + Found;
        }

        public void ClearSampling()
        {
            SamplerValues.ClearData();
        }
    }
}
```

Bilag E

Kode i *StatisticControls.cs*

```
using System;
using System.IO;
using System.Collections;

namespace SpecialeKode {
    // Class for tracking of statistic data in the normal way
    class cNormalTracker
    {
        public int DetailLevel;
        public uint[] HitTracker;
        private ulong BiggestValue = 0;

        // Constructer for making new
        public cNormalTracker(int nDetail)
        {
            DetailLevel = nDetail;
            HitTracker = new uint[Convert.ToInt64(Math.Pow(2, DetailLevel))];
        }

        // Constructer when loading old data from file
        public cNormalTracker(string FileName)
        {
            try
            {
                FileStream LoadFile = new FileStream(FileName, FileMode.Open, FileAccess.Read);

                // Get the DetailLevel from first 4 bytes
                byte[] bDetailLevel = new byte[4];
                LoadFile.Read(bDetailLevel, 0, 4);
                DetailLevel = BitConverter.ToInt32(bDetailLevel, 0);

                // Get the BiggestValue from next 8 bytes
                byte[] bBiggestValue = new byte[8];
                LoadFile.Read(bBiggestValue, 0, 8);
                BiggestValue = BitConverter.ToUInt64(bBiggestValue, 0);

                // Get all the values for the HitTracker
                HitTracker = new uint[Convert.ToInt64(Math.Pow(2, DetailLevel))];
                byte[] tmpValue = new byte[4];
                for (int i = 0; i < HitTracker.Length; i++)
                {
                    LoadFile.Read(tmpValue, 0, 4);
                    HitTracker[i] = BitConverter.ToUInt32(tmpValue, 0);
                }
            }
            catch
            {
                // If error in attempt to get data from file just create fresh
                // object with nDetail of 8
                DetailLevel = 8;
                HitTracker = new uint[Convert.ToInt64(Math.Pow(2, DetailLevel))];
            }
        }

        public int SizeOfStoredData()
        {
            return (Convert.ToInt32(Math.Pow(2,DetailLevel))*4);
        }
```

```
    public ulong GetBiggestValue()
    {
        return BiggestValue;
    }

    public void RecordHit(uint IP)
    {
        long tmpValue;
        long HitOn =
            Math.DivRem(IP, Convert.ToInt64(Math.Pow(2,(32-DetailLevel))), out tmpValue);

        if (BiggestValue == Convert.ToUInt64(HitTracker[HitOn])) { BiggestValue++; };
        HitTracker[HitOn]++;
    }

    public void RecordDel(uint IP)
    {
        long tmpValue;
        long DelFrom =
            Math.DivRem(IP, Convert.ToInt64(Math.Pow(2, (32 - DetailLevel))), out tmpValue);

        HitTracker[DelFrom]--;
    }

    public void SaveToFile(string FileName)
    {
        FileStream SaveFile = new FileStream(FileName, FileMode.Create, FileAccess.Write);

        // Save DetailLevel variable in 4 bytes
        byte[] bDetailLevel = BitConverter.GetBytes(DetailLevel);
        SaveFile.Write(bDetailLevel, 0, 4);

        // Save BiggestValue variable in 4 bytes
        byte[] bBiggestValue = BitConverter.GetBytes(BiggestValue);
        SaveFile.Write(bBiggestValue, 0, 8);

        // Save all the entries in the HitTracker
        for (int i = 0; i < HitTracker.Length; i++)
        {
            byte[] iValue = BitConverter.GetBytes(HitTracker[i]);
            SaveFile.Write(iValue, 0, 4);
        };

        SaveFile.Close();
    }

    public int nValues()
    {
        int tmpResult = 0;

        for (int i = 0; i < HitTracker.Length; i++)
        {
            if (HitTracker[i] > 0) { tmpResult++; };
        };

        return tmpResult;
    }
}

class cValueHolder
{
    public int i;
    public int a;

    public cValueHolder(int iSet, int aSet)
    {
        i = iSet;
        a = aSet;
    }
}

// Class for controlling multiple instances of the sampling algoritm class
class cSamplingController
{
    private int nDetailLevel;
    private cSamplingTracker[] Trackers;

    public cSamplingController(int DetailLevel, int nTrackers, int nmap)
    {
```

```
    nDetailLevel = DetailLevel;
    Trackers = new cSamplingTracker[nTrackers];
    Random Randomizer = new Random();

    for (int i = 0; i < Trackers.Length; i++)
    {
        Trackers[i] = new cSamplingTracker(DetailLevel, Randomizer, nmap);
    };
}

public void GetSamples(cSamplerValues Values, bool DoRecursively)
{
    // Check if doing it recursively
    if (DoRecursively)
    {
        Queue SampledValues = new Queue();

        int i;
        int a;
        bool ValReturned;

        for (int j = 0; j < Trackers.Length; j++)
        {
            ValReturned = Trackers[j].Sample(out i, out a);
            if (ValReturned && Values.HitTracker[i] == 0)
            {
                Values.HitTracker[i] = Convert.ToUInt32(a);
                cValueHolder tmpVal = new cValueHolder(i, a);
                SampledValues.Enqueue(tmpVal);
            };
        }

        while (SampledValues.Count > 0)
        {
            cValueHolder GetVal = (cValueHolder)SampledValues.Dequeue();
            for (int j = 0; j < Trackers.Length; j++)
            {
                Trackers[j].Update(GetVal.i, -GetVal.a);
                ValReturned = Trackers[j].Sample(out i, out a);
                if (ValReturned && Values.HitTracker[i] == 0)
                {
                    Values.HitTracker[i] = Convert.ToUInt32(a);
                    cValueHolder tmpVal = new cValueHolder(i, a);
                    SampledValues.Enqueue(tmpVal);
                };
            };
        };

        // Re-insert the removed values
        for (int j = 0; j < Values.HitTracker.Length; j++)
        {
            if (Values.HitTracker[j] > 0)
            {
                for (int k = 0; k < Trackers.Length; k++)
                    { Trackers[k].Update(j, Convert.ToInt32(Values.HitTracker[j])); };
            };
        };
    }
    else
    {
        int i;
        int a;
        bool ValReturned;

        for (int j = 0; j < Trackers.Length; j++)
        {
            ValReturned = Trackers[j].Sample(out i, out a);
            if (ValReturned) { Values.HitTracker[i] = Convert.ToUInt32(a); };
        };
    };
}

public void RecordHit(uint IP)
{
    long tmpValue;
    long HitOn =
        Math.DivRem(IP, Convert.ToInt64(Math.Pow(2, (32 - nDetailLevel))), out tmpValue);

    for (int i = 0; i < Trackers.Length; i++)
    {
```

```
            Trackers[i].Update(Convert.ToInt32(HitOn), 1);
        };
    }

    public void RecordDel(uint IP)
    {
        long tmpValue;
        long DelFrom =
            Math.DivRem(IP, Convert.ToInt64(Math.Pow(2, (32 - nDetailLevel))), out tmpValue);

        for (int i = 0; i < Trackers.Length; i++)
        {
            Trackers[i].Update(Convert.ToInt32(DelFrom), -1);
        };
    }

    // Method for use with the full file analysis
    public void RecordValue(int i, int a)
    {
        for (int j = 0; j < Trackers.Length; j++)
        {
            Trackers[j].Update(i, a);
        };
    }
}

// Class for saving the data using the datastructure from the article
class cSamplingTracker
{
    private int DetailLevel;
    private cUniqueElement[] UEs;
    //private cDestinctElements[] DEs;
    private cDestinctElements DE;
    //private cDestinctElements_Test[] DEs;
    //private cDestinctElements_Test DE;
    private cHash[] Hashes;

    public cSamplingTracker(int nDetailLevel, Random rSeed, int nmap)
    {
        DetailLevel = nDetailLevel + 1;
        UEs = new cUniqueElement[DetailLevel];
        //DEs = new cDestinctElements[DetailLevel];
        DE = new cDestinctElements(rSeed, nmap);
        Hashes = new cHash[DetailLevel];

        // Code for testing purposes
        //DEs = new cDestinctElements_Test[DetailLevel];
        //DE = new cDestinctElements_Test(DetailLevel);

        for (int i = 0; i < DetailLevel; i++)
        {
            UEs[i] = new cUniqueElement();
            //DEs[i] = new cDestinctElements(rSeed, nmap);
            //DEs[i] = new cDestinctElements_Test(DetailLevel);

            //int MaxValue = Convert.ToInt32(Math.Pow(2, i));
            //Hashes[i] = new cHash(rSeed.Next(), i + 1);
            Hashes[i] = new cHash(rSeed.Next(), i);
        };
    }

    public void Update(int i, int a)
    {
        for (int j = 0; j < DetailLevel; j++)
        {
            if (Hashes[j].GetHashValue(i) == 0)
            {
                UEs[j].Update(i, a);
                //DEs[j].Update(i, a);
            };
        };
        DE.Update(i, a);
    }

    public bool Sample(out int i, out int a)
    {
        int nElements = DE.Report();
        if (nElements > 0)
        {
            int j = Convert.ToInt32(Math.Ceiling(Math.Log(nElements, 2)));
```

```
            // Original code
            /*
            if (DEs[j].Report() == 1)
            {
                UEs[j].Report(out i, out a);
                return true;
            }
            else
            {
                i = 0;
                a = 0;
                return false;
            };
            */

            // Revised addition
            if (((UEs[j].c * UEs[j].c2) - (UEs[j].C*UEs[j].C)) != 0 || UEs[j].c == 0)
            {
                i = 0;
                a = 0;
                return false;
            }
            else
            {
                UEs[j].Report(out i, out a);
                return true;
            };
        }
        else
        {
            i = 0;
            a = 0;
            return false;
        };
    }
}

// Class for the Unique Element data structure defined in the algorithm
class cUniqueElement
{
    public int c = 0;
    public int C = 0;

    // Revised addition
    public int c2 = 0;

    public void Update(int i, int a)
    {
        c = c + a;
        C = C + (a * i);

        // Revised addition
        c2 = c2 + (a * i * i);
    }

    public void Report(out int i, out int v)
    {
        v = c;
        i = C / v;
    }
}

// Class for the Destinct Elements data structure defined in the algorithm
class cDestinctElements
{
    private ulong nmap;
    private int MaxLength;
    private double phi = 0.77351d;
    private cHash Hasher;

    private int[,] Maps;

    public cDestinctElements(Random Seeder, int SetnMap)
    {
        nmap = Convert.ToUInt64(SetnMap);
        MaxLength = 32;
        Maps = new int[nmap, MaxLength];
        Hasher = new cHash(Seeder.Next(), MaxLength);
    }
```

```
    public void Update(int i, int a)
    {
        ulong HashedX = Hasher.GetHashValue(i);
        int alpha = Convert.ToInt32(HashedX % nmap);
        int index = p(HashedX / nmap);

        if (index < MaxLength) { Maps[alpha, index] += a; };
    }

    public int Report()
    {
        int S = 0;
        int R;

        for (int i = 0; i < Convert.ToInt32(nmap); i++)
        {
            R = 0;
            while (Maps[i, R] > 0)
            {
                R++;
            };
            S += R;
        };

        if ((S / Convert.ToDouble(nmap)) < 0.5d)
        {
            int nValues = 0;
            for (int j = 0; j < Convert.ToInt32(nmap); j++)
            {
                for (int k = 0; k < MaxLength; k++)
                {
                    if (Maps[j, k] > 0) { nValues++; };
                };
            };

            return nValues;
        };

        double tmp = (nmap / phi) * Math.Pow(2,(S / Convert.ToDouble(nmap)));
        double tmp2 = Math.Floor(tmp);
        return Convert.ToInt32(tmp2);
    }

    private int p(ulong Value)
    {
        BitArray BA = cGlobalFunctions.GetBitsFromValue(MaxLength, Value);

        for (int i = 0; i < BA.Length; i++)
        {
            if (BA[BA.Length - 1 - i])
            {
                return i;
            };
        };

        // Should never reach this
        return MaxLength;
    }
}

// Class for the hash functions
class cHash
{
    private int SeedValue;
    private int Bits;

    public cHash(int Seed, int nBits)
    {
        SeedValue = Seed;
        Bits = nBits;
    }

    public ulong GetHashValue(int Value)
    {
        return cHashFunction.GetHash(SeedValue, Bits, Value);
    }
}

// Class for holding temporary values read from sampling tracker
```

```
    class cSamplerValues
    {
        public int DetailLevel;
        public uint[] HitTracker;

        public cSamplerValues(int nDetail)
        {
            DetailLevel = nDetail;
            HitTracker = new uint[Convert.ToInt64(Math.Pow(2, DetailLevel))];
        }

        public void SetValue(int i, int a)
        {
            HitTracker[i] = Convert.ToUInt32(a);
        }

        public void ClearData()
        {
            for (int i = 0; i < HitTracker.Length; i++) { HitTracker[i] = 0; };
        }

        public int nValues()
        {
            int tmpResult = 0;

            for (int i = 0; i < HitTracker.Length; i++)
            {
                if (HitTracker[i] > 0) { tmpResult++; };
            };

            return tmpResult;
        }
    }
}
```

Bilag F

Kode i *Functions.cs*

```
using System;
using System.Collections;

namespace SpecialeKode {
    class cGlobalFunctions
    {
        public static BitArray GetBitsFromValue(int nBits, ulong Value)
        {
            BitArray Result = new BitArray(nBits);

            ulong Mask = 1ul;
            for (int i = 0; i < nBits; i++)
            {
                Result[nBits - 1 - i] = (Value & Mask) != 0;
                Mask <<= 1;
            };

            return Result;
        }
    }

    class cHashFunction
    {
        public static ulong GetHash(int Seed, int nBits, int Value)
        {
            if (nBits == 0) { return 0; }
            else { return ConvertBHtoULong(GetBitHash(Seed, nBits, Value)); };
        }

        private static BitArray GetBitHash(int Seed, int nBits, int Value)
        {
            int tmpValue = Seed + Value;
            if (tmpValue < 0) { tmpValue += int.MaxValue; };

            BitArray Result = new BitArray(nBits);
            Random Randomizer = new Random(tmpValue);

            // Generate the random bits
            for (int i = 0; i < nBits; i++)
            {
                if (Randomizer.Next(2) == 1) { Result[i] = true; }
                else { Result[i] = false; };
            };
            return Result;
        }

        private static ulong ConvertBHtoULong(BitArray Bits)
        {
            ulong Result = 0;

            for (int i = 0; i < Bits.Length; i++)
            {
                if (Bits[i]) { Result += 1ul << Bits.Length - 1 - i; };
            };
            return Result;
        }
    }
}
```

Bilag G

Kode i *DisplayPanel.cs*

```
using System;
using System.Windows.Forms;
using System.Drawing;
using System.Drawing.Drawing2D;

namespace SpecialeKode {
    class cImagePanel : Panel
    {
        // Data structures with the values to draw
        private cNormalTracker NormalTracker = null;
        private cSamplerValues SamplerValues = null;

        // Variables controlling the drawing
        private int ColumnWidth = 1;
        private uint FirstTrackShown = 0;
        private uint nTracksShown;
        private int nColumnsShown;
        private int nTracksPrColumn;
        private int nHitsYaxisUp = 1000;
        private bool DrawTwo;

        public cImagePanel()
        {
            // Setup parameters for drawing in buffers to avoid flickering
            SetStyle(ControlStyles.UserPaint, true);
            SetStyle(ControlStyles.AllPaintingInWmPaint, true);
            SetStyle(ControlStyles.OptimizedDoubleBuffer, true);
        }

        // Method for registering the tracking data structures
        public void RegisterTrackers(cNormalTracker nTracker, cSamplerValues nSampler)
        {
            NormalTracker = nTracker;
            SamplerValues = nSampler;
            nTracksShown = Convert.ToUInt32(NormalTracker.HitTracker.Length);
        }

        // Method for registering the drawing variables
        public void RegisterVariables(int nColWidth, int YHitsHeight, bool DrawAsTwo)
        {
            ColumnWidth = nColWidth;
            nHitsYaxisUp = YHitsHeight;
            DrawTwo = DrawAsTwo;
        }

        // Overriden paint-method called every time the window want to redraw the panel
        protected override void OnPaint(PaintEventArgs pea)
        {
            // Colors used
            SolidBrush bBackGround = new SolidBrush(Color.White);
            SolidBrush bNormalColumn = new SolidBrush(Color.Green);
            SolidBrush bSampleColumn = new SolidBrush(Color.Red);
            SolidBrush bBlackColumn = new SolidBrush(Color.Black);
            Pen pLines = new Pen(Color.Black);
            // Get the drawable area
            Graphics PaintArea = pea.Graphics;
            // Draw background color
            PaintArea.FillRectangle(bBackGround, 0, 0, this.Width, this.Height);
```

```
// Check if analysis started
if (NormalTracker == null) { return; };

// Check if drawing should be as one or two areas
if (DrawTwo)
{
    // Draw the two x-axis'
    int xAxisPos1 = this.Height - 1;
    int xAxisPos2 = this.Height / 2;
    PaintArea.DrawLine(pLines, 0, xAxisPos1, this.Width, xAxisPos1);
    PaintArea.DrawLine(pLines, 0, xAxisPos2, this.Width, xAxisPos2);

    nColumnsShown =
        Convert.ToInt32(Math.Min((this.Width / (ColumnWidth)), nTracksShown));
    nTracksPrColumn = Convert.ToInt32(Math.Ceiling(Convert.ToDouble(nTracksShown)
        / Convert.ToDouble(nColumnsShown)));

    uint i = FirstTrackShown;
    uint tmpValue;
    uint tmpSampler;

    // Calculate how many hits pr pixel should be shown
    float nHitsPixel = nHitsYaxisUp / (xAxisPos1 - xAxisPos2);

    // Draw the columns
    i = 0;
    while (i < nColumnsShown)
    {
        // Get value of current columns
        int j = 0;
        tmpValue = 0;
        tmpSampler = 0;
        while ((j < nTracksPrColumn) && ((i * nTracksPrColumn + j) < nTracksShown))
        {
            tmpValue +=
                NormalTracker.HitTracker[FirstTrackShown + (i * nTracksPrColumn) + j];
            tmpSampler +=
                SamplerValues.HitTracker[FirstTrackShown + (i * nTracksPrColumn) + j];
            j++;
        };

        // Calculate height of drawn columns
        int nPixelHeight1 = 0;
        if (nHitsPixel > 0) { nPixelHeight1 = Convert.ToInt32(tmpValue / nHitsPixel); };
        nPixelHeight1 = Math.Min(nPixelHeight1, xAxisPos1 - xAxisPos2);
        int nPixelHeight2 = 0;
        if (nHitsPixel > 0) { nPixelHeight2 = Convert.ToInt32(tmpSampler / nHitsPixel); };
        nPixelHeight2 = Math.Min(nPixelHeight2, xAxisPos1 - xAxisPos2);

        // Draw specific columns
        PaintArea.FillRectangle(bBlackColumn, i * ColumnWidth,
            xAxisPos2 - nPixelHeight1, ColumnWidth, nPixelHeight1);
        PaintArea.FillRectangle(bBlackColumn, i * ColumnWidth,
            xAxisPos1 - nPixelHeight2, ColumnWidth, nPixelHeight2);
        i++;
    };
}
else
{
    // Draw x-axis
    int xAxisPos = this.Height - 1;
    PaintArea.DrawLine(pLines, 0, xAxisPos, this.Width, xAxisPos);

    nColumnsShown =
        Convert.ToInt32(Math.Min((this.Width / (ColumnWidth * 2)), nTracksShown));
    nTracksPrColumn = Convert.ToInt32(Math.Ceiling(Convert.ToDouble(nTracksShown)
        / Convert.ToDouble(nColumnsShown)));

    uint i = FirstTrackShown;
    uint tmpValue;
    uint tmpSampler;

    float nHitsPixel = nHitsYaxisUp / ((this.Height - 5.0F) - (this.Height - xAxisPos));

    // Draw the columns
    i = 0;
    while (i < nColumnsShown)
    {
        // Get value of current columns
        int j = 0;
```

```
                tmpValue = 0;
                tmpSampler = 0;
                while ((j < nTracksPrColumn) && ((i * nTracksPrColumn + j) < nTracksShown))
                {
                    tmpValue += NormalTracker.HitTracker[FirstTrackShown + (i * nTracksPrColumn) + j];
                    tmpSampler += SamplerValues.HitTracker[FirstTrackShown + (i * nTracksPrColumn) + j];
                    j++;
                };

                // Calculate height of drawn columns
                int nPixelHeight1 = 0;
                if (nHitsPixel > 0) { nPixelHeight1 = Convert.ToInt32(tmpValue / nHitsPixel); };
                int nPixelHeight2 = 0;
                if (nHitsPixel > 0) { nPixelHeight2 = Convert.ToInt32(tmpSampler / nHitsPixel); };

                // Draw specific columns
                PaintArea.FillRectangle(bNormalColumn, i * (ColumnWidth * 2),
                    xAxisPos - nPixelHeight1, ColumnWidth, nPixelHeight1);
                PaintArea.FillRectangle(bSampleColumn, ColumnWidth + (i * (ColumnWidth * 2)),
                    xAxisPos - nPixelHeight2, ColumnWidth, nPixelHeight2);
                i++;
            };
        };
    }

    public uint GetColumnFromXY(int x, int y)
    {
        return Convert.ToUInt32(x / ColumnWidth);
    }

    public string GetColumnData(uint Col)
    {
        if (Col < nColumnsShown)
        {
            uint Value = 0;
            uint StartTrack = FirstTrackShown + Convert.ToUInt32(Col * nTracksPrColumn);
            for (uint i = StartTrack; i < StartTrack + nTracksPrColumn; i++)
            {
                Value += NormalTracker.HitTracker[i];
            };

            uint StartIP = StartTrack << (32 - NormalTracker.DetailLevel);
            byte[] StartIPBytes = BitConverter.GetBytes(StartIP);

            uint StopIP = uint.MaxValue >> NormalTracker.DetailLevel;
            StopIP += (StartTrack + Convert.ToUInt32(nTracksPrColumn) - 1)
                << (32 - NormalTracker.DetailLevel);
            byte[] StopIPBytes = BitConverter.GetBytes(StopIP);

            return "Hits: " + Value + " Start IP: " + StartIPBytes[3] + "."
                + StartIPBytes[2] + "." + StartIPBytes[1] + "." + StartIPBytes[0]
                +" Stop IP: "+StopIPBytes[3]+"."+StopIPBytes[2]+"."+StopIPBytes[1]
                +"."+StopIPBytes[0];
        };
        return "";
    }

    public void SetZoom(uint StartCol, uint EndCol)
    {
        uint StartTrack = FirstTrackShown + (StartCol * Convert.ToUInt32(nTracksPrColumn));
        uint EndTrack = FirstTrackShown +
            ((EndCol + 1) * Convert.ToUInt32(nTracksPrColumn)) - 1;

        if (!(EndTrack < NormalTracker.HitTracker.Length))
            { EndTrack = Convert.ToUInt32(NormalTracker.HitTracker.Length)-1; };

        if (StartTrack < EndTrack)
        {
            FirstTrackShown = StartTrack;
            nTracksShown = EndTrack - StartTrack + 1;
        };
    }

    public void UnZoom()
    {
        FirstTrackShown = 0;
        nTracksShown = Convert.ToUInt32(NormalTracker.HitTracker.Length);
    }
  }
}
```

Bilag H

Kode i *TestCode.cs*

```
using System;

namespace SpecialeKode {
    class cDETester
    {
        private int nDetailLevel;
        private cNormalTracker NormalTracker;
        private cDestinctElements DE;
        private cLogFileReader LogFile;

        public cDETester(string strLogFile, int nBit)
        {
            nDetailLevel = nBit;
            Random Randomizer = new Random();
            LogFile = new cLogFileReader(strLogFile);
            NormalTracker = new cNormalTracker(nBit);
            DE = new cDestinctElements(Randomizer, 64);
        }

        public string ReadXEntries(int nEntries)
        {
            cApacheLogEntry Entry;
            for (int i = 0; i < nEntries; i++)
            {
                Entry = LogFile.GetNextEntry();
                NormalTracker.RecordHit(Entry.RequesterIP);

                long tmpValue;
                long HitOn = Math.DivRem(Entry.RequesterIP,
                    Convert.ToInt64(Math.Pow(2, (32 - nDetailLevel))), out tmpValue);
                DE.Update(Convert.ToInt32(HitOn), 1);
            };

            int TrueHits = 0;
            for (int j = 0; j < NormalTracker.HitTracker.Length; j++)
            {
                if (NormalTracker.HitTracker[j] > 0) { TrueHits++; };
            };

            int DEHits = DE.Report();

            return "True hits: " + TrueHits + " *** DE hits: " + DEHits;
        }
    }

    // Test code simulating the Destinct Elements data structure
    class cDestinctElements_Test
    {
        private int nDetailLevel;
        private uint[] Elements;

        public cDestinctElements_Test(int DetailLevel)
        {
            nDetailLevel = DetailLevel;
            Elements = new uint[Convert.ToInt64(Math.Pow(2, nDetailLevel))];
        }

        public void Update(int i, int a)
```

```
        {
            Elements[i] = Convert.ToUInt32(Elements[i] + a);
        }

        public int Report()
        {
            int Count = 0;

            for (int i = 0; i < Elements.Length; i++)
            {
                if (Elements[i] > 0) { Count++; };
            };

            return Count;
        }
    }
}
```

Litteratur

[1] Gereon Frahling, Piotr Indyk, and Christian Sohler. Sampling in dynamic data streams and applications. In *SCG '05: Proceedings of the twenty-first annual symposium on Computational geometry*, pages 142–149, New York, NY, USA, 2005. ACM Press.

[2] Sohler og muligvis Frahling og Indyk. Sampling in dynamic data streams and applications (ny udgave). *http://wwwcs.uni-paderborn.de/cs/ag-madh/WWW/english/Sohler/publications.html*, 2005.

[3] S. Muthukrishnan. *Data Streams: Algorithms and Applications*. Publishers Inc., 2005.

[4] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proceedings of the 28th International Conference on Very Large Data Bases, Hong Kong, China*, 2002.

[5] Min Fang, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani, and Jeffrey D. Ullman. Computing iceberg queries efficiently. In *Proc. 24th Int. Conf. Very Large Data Bases, VLDB*, pages 299–310, 24–27 1998.

[6] Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.

[7] N. Duffield, C. Lund, and M. Thorup. Learn more, sample less: Control of volume and variance in network measurement. *http://public.research.att.com/˜duffield/papers/*, 2005.

[8] Anna C. Gilbert, Yannis Kotidis, S. Muthukrishnan, and Martin Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *The VLDB Journal*, pages 79–88, 2001.

[9] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Quicksand: Quick summary and analysis of network data dimacs technical report. *http://www.math.lsa.umich.edu/˜annacg/research.html*, 2001.

[10] O. Spatscheck C. D. Cranor, T. Johnson and V. Shkapenyuk. The gigascope stream database. *IEEE Data Eng. Bull.*, 26(1):27–32, 2003.

[11] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: a stream database for network applications. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 647–651, New York, NY, USA, 2003. ACM Press.

[12] Theodore Johnson Muthukrishnan. Streams, security and scalability. *http://www.research.att.com/˜divesh/papers/*, 2005.

[13] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[14] Sumit Ganguly, Minos Garofalakis, and Rajeev Rastogi. Tracking set-expression cardinalities over continuous update streams. *The VLDB Journal*, 13(4):354–369, 2004.