Rekonstruktion af evolutionære træer vha. eksperimenter

Casper Lund Thomsen <casper@daimi.au.dk> *19993647*
Mads Laursen <dossen@daimi.au.dk> *19993727*

Vejleder:
Christian Nørgaard Storm Pedersen
Datalogisk Institut
Aarhus Universitet

6. juni 2005

Abstract

Inden for bioinformatik er evolutionære træer af stor interesse. Da træerne ikke er givet, må de konstrueres - desværre kræver mange konstruktionsmetoder meget tid. Der findes imidlertid en familie af algoritmer, der baserer sig på såkaldte eksperimenter og er væsentligt mere effektive, målt i beregningstid. Vi fremlægger her en implementation af en algoritme til rekonstruktion af evolutionære træer i tid $O(nd \log_d n)$, hvor d er graden af træet og n er antallet af arter. Algoritmen benytter højst $n\lceil d/2 \rceil (\log_{2\lceil d/2 \rceil - 1} n + O(1))$ eksperimenter for $d > 2$ og $n(\log n + O(1))$ for $d = 2$. Algoritmen blev udviklet af Brodal et. al. og publiceret i "The Complexity of Construction Evoluotionary Trees Using Experiments". Den resulterende implementation, fremstillet i Java, er udviklet med henblik på at kunne indgå i andre projekter baseret på eksperimentmodellen.

Indhold

Figurer

Kapitel 1

Indledning

Der foregår hele tiden en naturlig evolution blandt de forskellige arter af organismer på Jorden. Det er af stor interesse for biologer m.fl. at vide, hvorledes disse arter relaterer til hinanden. Dette gøres typisk ved hjælp af et evolutionært træ; i et sådant træ repræsenterer bladene arterne, mens de interne knuder repræsenterer artsdannelser og roden er den seneste fælles forfader til arterne.

Den fuldstændige evolutionære historie er sjældent kendt, hvilket betyder, at man er nødt til at udlede denne fra den mængde af arter, man har tilrådighed. Hvilke informationer, der skal udledes, og hvordan selve træet skal rekonstrueres, beskrives i den model man arbejder med for at genskabe det evolutionære træ. I denne opgave har vi arbejdet med eksperiment-modellen, der er beskrevet i [KLW90].

Et eksperiment giver information om, hvorledes tre arter er relateret; hvis vi i et evolutionært træ udvælger tre arter, skal det deltræ, disse tre giver anledning til, have den samme topologi som det træ, man får ud fra et eksperiment på disse tre. Se afsnit 2.2 for yderligere informationer om eksperimenter.

Eksperimentmodellen antager, at eksperimenterne er perfekte, altså at de aldrig modsiger hinanden. Hermed menes det, at havde vi givet et sandt træ for de arter, der eksperimenteres på, så ville alle resultaterne af eksperimenterne stemme overens med det sande træ. Hvis dette er tilfældet, så konstruerer algoritmen det sande evolutionære træ. I modsat fald vil algoritmen stadig resultere i et træ, der er konsistent med eksperimenterne, men ikke nødvendigvis med det sande træ. Vi har i kapitel 6 overvejet, hvorledes man kan slække på kravet om, at eksperimenter skal være perfekte.

[KLW90] præsenterer en række algoritmer med forskellige udførelsesti-

der og forskellige antal benyttede eksperimenter. Overordnet set er der en afvejning mellem antallet af udførte eksperimenter og den algoritmiske kompleksitet, således at de hurtige algoritmer benytter flere eksperimenter.[1] I forhold til algoritmen i [LOÖ99] præsenterer [BFPÖ01] en algoritme, der har samme udførelsestid, men er en faktor $\Theta(\log d)$, hvor d er udgraden af træet, bedre mht. antallet af eksperimenter, der foretages i forhold til resultatet fra [LOÖ99]. Dette er vigtigt, da eksperimenter, per [KLW90], kan være dyre at udføre i praksis.

I [BFPÖ01] opnår den omtalte forbedring ved at videreudvikle algoritmen fra [LOÖ99]. Det er den videreudviklede algoritme, vi har implementeret og beskrevet i flere detaljer i dette speciale. Hvor det ikke er specificeret, vil "artiklen" i denne rapport referere til [BFPÖ01].

Forløbet af rapporten er som følger: I kapitel 2 vil algoritmen til rekonstruktion af træer via eksperimenter blive præsenteret og skitseret. Kapitel 3 gennemgår algoritmens dele i detaljer, efterfulgt af en dybdere gennemgang af hvorledes de kombineres til den fulde algoritme. Hvordan man benytter vores implementation bliver gennemgået i kapitel 4. Dernæst gennemgår vi, i kapitel 5, implementationens performance i praksis, ved at undersøge udførelsestiden og forbruget af eksperimenter. Herefter følger kapitel 6 med vores idéer til fremtidigt arbejde. Endeligt konkluderer vi i kapitel 7 på projektet.

1.1 Hjemmeside

Vi har opbygget en hjemmeside (`http://speciale.dossen.dk/`), hvor vi har placeret diverse informationer relevant til vores speciale. På denne side er det muligt at hente vores implementation i form af en JAR-fil. Der forefindes også en vejledning i, hvorledes denne JAR-fil benyttes til at rekonstruere træer. Det er endvidere muligt at downloade kildekoden til vores implementation fra hjemmesiden. I bilag F ses skærmdumps af hjemmesiden.

Vi har ydermere placeret en række testdata på hjemmesiden, således at man hurtigt kan komme i gang med at benytte programmet.

[1] En algoritme med udførelsestid $O(n \log n)$ kan naturligvis ikke udføre mere end $O(n \log n)$ eksperimenter, men modellen udtaler sig også om konstanterne for antallet af eksperimenter.

Kapitel 2

Overblik

I det følgende vil vi kort skitsere algoritmen fra [BFPÖ01] til konstruktion af evolutionære træer, vi har implementeret. De nærmere detaljer om algoritmen følger i afsnit 3.5.

Algoritmen tillader rekonstruktion af træer med arbitrær udgrad i tid $O(nd \log_d n)$, hvor n er antallet af blade i træet, og d er dets højeste udgrad. Under konstruktionen foretages maksimalt $n\lceil d/2 \rceil (\log_{2\lceil d/2 \rceil - 1} n + O(1))$ eksperimenter for $d > 2$ og $n(\log n + O(1))$ for $d = 2$.

I tilfælde af binære træer er det en forbedring i forhold til [KLW90]s algoritme med $O(n \log n)$ udførelsestid, der benytter $4 \cdot n \log n$ eksperimenter.

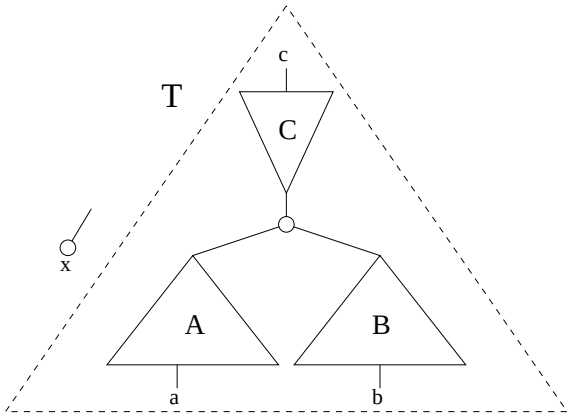
2.1 Fremgangsmåde

Algoritmen tager som input en mængde af arter. Den opbygger så det evolutionære træ, T, ved at tilføje arterne en for en til træet over de arter, der allerede er blevet tilføjet.

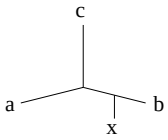
Figur 2.1 viser et trin i søgningen efter indsættelsespunktet for arten x. Ved at foretage eksperimenter mellem x og arter a, b og c fra undertræerne A, B og C, afgør vi i hvilket undertræ søgningen skal fortsættes. Hvis vi eksempelvis finder, at x er mest relateret til b, som figur 2.2 viser, fortsætter vi søgningen i undertræet B. Hvis x er lige beslægtet til alle undertræerne, så indsættes x som et nyt blad på den aktuelle knude.

Det initielle træ konstrueres med den første art som det eneste blad. Den næste art indsættes så som søskende til den første art. Dette er de eneste mulige topologier, se figur 2.3.

Med den tredje art er der de fire muligheder vist på figur 2.4. Hvilken af dem, vi skal vælge, afgøres som i det generelle tilfælde ovenfor, med den mo-



Figur 2.1: I træet T søger vi efter indsættelsespunktet for arten x.

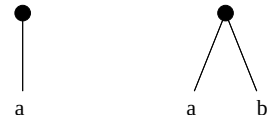


Figur 2.2: Ved eksperimenter finder vi, at x skal indsættes på stien mellem den aktuelle knude og b.

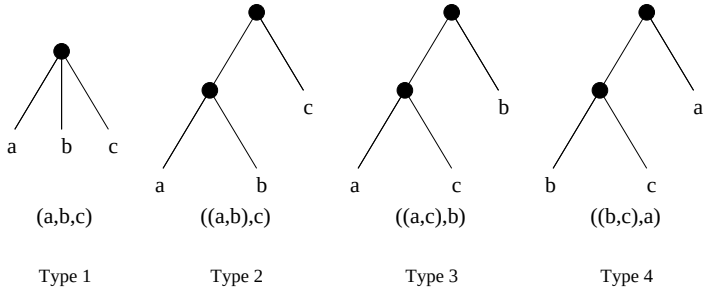
difikation, at C er et tomt træ. Hermed behøver vi kun det ene eksperiment på $\{x, a, b\}$.

Denne fremgangsmåde lader os konstruere det korrekte træ for en vilkårlig mængde af arter. Da vi ikke har nogen garantier om balanceringen i det evolutionære træ, kan antallet af eksperimenter, der skal udføres for hver art, i værste fald blive lineært i det eksisterende træs størrelse. Antag, at træet T fra figur 2.1 er binært, så lader vi C være de knuder, vi har søgt igennem, mens A og B er de undertræer der fortsat er i betragtning. Hvis A så kun består af bladet a, vil et eksperiment på a, b og x kun reducere søgerummet med 1 blad, nemlig a, medmindre vi finder at x skal indsættes her.

For at sikre at eksperimenterne reducerer søgerummet effektivt, vil vi i afsnit 3.2 introducere et balanceret rodfæstet træ, *separatortræet* S_T, og en bijektion, $s^{(T)}$, mellem knuder i T og i S_T. Vi benytter notationen s_v for $s^{(T)}(v), v \in T$. Figur 2.5 viser hvorledes knuden u i T svarer til en knude s_u i S_T og undertræerne A, B og C svarer til undertræerne S_A, S_B og S_C. De

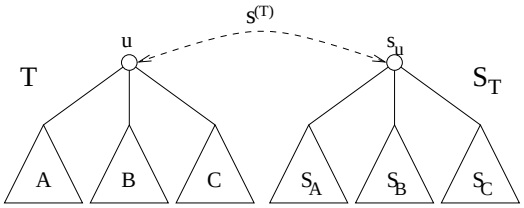


Figur 2.3: For de første to arter er der ikke noget valg med hensyn til træets topologi.



Figur 2.4: De fire mulige udfald af et eksperiment med arterne a, b og c.

præcise egenskaber ved bijektionen vil blive defineret i afsnit 3.2.

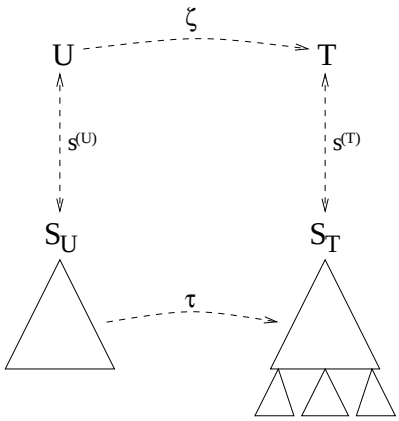


Figur 2.5: Separatortræet S_T har en knude s_u for hver knude u i T

Separatortræet konstrueres med logaritmisk højde og således, at vi kan udføre rekonstruktionen af det evolutionære træ ved at søge fra roden af S_T. Herved begrænser vi det udførte antal eksperimenter.

Tidsforbruget på at vedligeholde S_T under indsættelser i T vises i [BFPÖ01] at være $O(\log^2 n)$ amortiseret per indsættelse. Ved at introducere konstruktionen *sample*, som vi vil beskrive i detaljer i afsnit 3.3, kan dette reduceres til $O(\log n)$ amortiseret per indsættelse. Samplen består af et træ, U, og en

morfi, $\zeta : U \rightarrow T$, der via bijektionerne $s^{(U)}$ og $s^{(T)}$ giver os en træmorfi, $\tau : S_U \rightarrow S_T$, der lader os simulere S_T ved hjælp af S_U. Figur 2.6 viser relationerne mellem træerne.



Figur 2.6: Givet et træ, så har vi en bijektion til dets separatortræ. τ afbilleder fra S_U til S_T, og fra U til T har vi en lignende morfi.

I afsnit 3.3 beskriver vi, hvorledes samplen skal konstrueres, så vi med dens separatortræ kan simulere separatortræet for T, uden eksplicit at konstruere hele S_T.

2.2 Eksperimenter

For hver art, der indsættes i det evolutionære træ, foretager algoritmen som nævnt eksperimenter, der leder frem til det rette indsættelsespunkt. Disse eksperimenter giver os med tre arter, $\{a, b, c\}$, som input en af de i figur 2.4 viste topologier. Som nævnt i kapitel 1, antages det, at alle de udførte eksperimenter er konsistente med det træ, der skal konstrueres. Givet inkonsistente eksperimenter vil algoritmen stadig konstruere et træ, og dette træ vil være konsistent med de udførte eksperimenter.

Der er ikke fra algoritmens side gjort andre antagelser om eksperimenterne, end at det er muligt at sammenligne den nye art med vilkårlige par af arter, der allerede er i træet. En mulig metode er DNA-DNA hybridisering, der, som fortalt i [BFPÖ01], baserer sig på en sammenhæng mellem beslægtedhed og den temperatur, hvor DNA molekyler binder sig til hinanden. Ved

en sådan laboratorieprocess er algoritmens tidsforbrug imellem hvert eksperiment af interesse, da vi ikke kan forudsige, hvilke eksperimenter algoritmen mangler.

Man kan også benytte træer. Givet det fulde træ, er den nemmeste løsning naturligvis blot at gengive det. Vi vil dog senere benytte kendte træer som orakler til afprøvning af algoritmen. En anden mulighed er, at der er flere træer, der hver især kun indeholder en delmængde af de arter, træet ønskes bygget for. I denne situation vil nogle eksperimenter stadig kunne udføres, enten fordi alle tre arter er i samme deltræ, eller fordi overlap mellem træerne tillader os at udlede topologien. Dette er dog ikke garanteret, at kunne lade sig gøre for alle de eksperimenter vi måtte ønske at udføre. Udvidelse af algoritmen i denne retning ligger uden for dette projekts område.

2.2.1 Afstandsmatricer

En anden mulighed for at udføre eksperimenterne er afstandsmatricer eller andre metoder baseret på afstande. Det er naturligvis en forudsætning, at vi fra matricerne kan give konsistente svar på topologierne. Det kan man ikke for generelle matricer, men [KLW90] definerer matricer, hvor vi kan få konsistente svar på eksperimenter som *støjende ultrametriske*, jfr. følgende definition 1.

Definition 1 *En matrice er støjende ultrametrisk, hvis der findes et rodfæstet evolutionært træ, hvor det gælder for alle tripletter af arter a, b og c, at* $M_{ab} < min\{M_{ac}, M_{bc}\}$ *hvis og kun hvis den laveste fælles forfader for a og b sidder under den laveste fælles forfader for a og c i det evolutionære træ.*

Givet en støjende ultrametrisk afstandsmatrice kan vi besvare eksperimenterne, således at alle svar er konsistente med et veldefineret evolutionært træ.

Et konkret problem ved afstandsmatricer er størrelsen. Da afstandsmatricerne optager $O(n^2)$ plads, følger det, at indlæsningen af en matrice tager $O(n^2)$ tid. Dette kan naturligvis laves med en lille konstant, men for tilstrækkeligt store træer er den implementerede algoritme faktisk hurtigere end indlæsningen af dens input. Et andet argument mod afstandsmatricer er, at de repræsenterer flere informationer, indsamlet på forhånd, end algoritmen kan drage udbytte af. I afsnit 6.2.1 vil vi overveje hvorledes dette problem kan reduceres.

Kapitel 3

Rekonstruktion af evolutionære træer vha. eksperimenter

Arbejdet med at implementere algoritmen fra artiklen har givet os en forståelse af, hvorledes de forskellige datastrukturer og metoder arbejder sammen. Det er denne forståelse for sammenhængen vi vil forsøge at videregive i dette kapitel.

Overordnet set følger vi strukturen fra artiklen, dvs. vi præsenterer de lemmaer og teoremer,[1] der optræder i artiklen. Det samme gør sig dog ikke gældende for definitionerne. Ved de definitioner, der stammer fra artiklen, vil vi nævne det, samt skrive, hvad definitionen hedder i [BFPÖ01].

Vi gennemarbejder altså alle de skridt, der er i [BFPÖ01], dvs. at vi også gennemgår de algoritmer, lemmaerne giver anledning til, der ikke direkte skal bruges til at realisere trærekonstruktionsalgoritmen, men blot er der for at lede en ind på den rette tankegang. Vi arbejder med en større detaljeringsgrad end den der bliver præsenteret i artiklen.

Efter at have læst [BFPÖ01] og diskuteret hvorledes vi skulle gribe opgaven an, skulle det afgøres, hvilket programmeringssprog vi ønskede at benytte. Diskussionen gik på om vi skulle benytte C, C++ eller Java; de to første sprog giver større frihed til at udnytte hukommelsen og har også en marginalt hurtigere udførelsestid, hvorimod Java har den fordel, at det har hukommelsesstyring, fx en Garbage collector. Dette, sammenholdt med at vi begge har erfaring med at udvikle i Java, gjorde at vi valgte Java, version 1.4.2, til vores implementation. Vi har benyttet det integrerede udviklingsværktøj Eclipse, version 3. For at administrere projektets kompleksitet har vi endvidere benyttet Javas pakke-mekanisme til at opdele vores implemen-

[1] Vi har dog oversat dem til dansk og indført vores notation.

tation i moduler.

Forløbet i dette kapitel er som følger: i afsnit 3.1 gennemgåes datastrukturen for det evolutionære træ, i afsnit 3.2 kommer en beskrivelse af datastrukturen for et separatortræ, samt hvorledes dette konstrueres og vedligeholdes under indsættelse af nye knuder. I afsnit 3.3 beskriver vi hvorledes vi konstruerer og vedligeholder en forbedret datastruktur for separatortræer, samplen. Det næste skridt er at sortere separatortræet, hvilket gøres i afsnit 3.4.

Med datastrukturerne for træer og separatortræer på plads, beskriver vi i afsnit 3.5 algoritmen til rekonstruktion af det evolutionære træ i større detaljer.

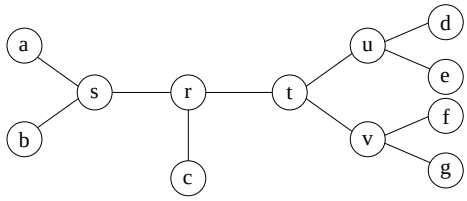
I afsnit 3.6 har vi en gennemgang af de øvrige datastrukturer, som er nødvendige for at kunne konstruere separatortræet og samplen effektivt. Afsnit 3.7 indeholder en gennemgang af de metoder, vi har benyttet til at teste vores implementation under udviklingsprocessen.

Når vi i de følgende afsnit fremfører et lemma eller teorem vil følgende struktur blive benyttet i behandlingen af dette: Et afsnit kaldet beskrivelse, som gennemgår hvorledes man konstruktivt opfylder kravene for det pågældende lemma/teorem, dette afsnit bliver efterfulgt af et afsnit om udførelsestiden og der afsluttes med et diskussionsafsnit.

Vi benytter følgende notation i opgaven: alle metoder står med kursiv (fx *insertNode*), mens alle variabler angives med understregninger.

3.1 Evolutionært træ

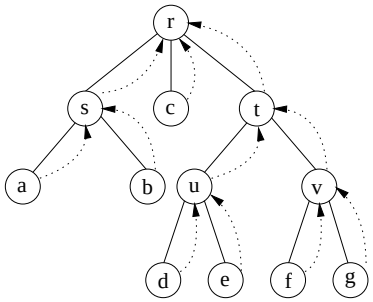
Den grundlæggende datastruktur, som repræsenterer algoritmens resultat, er det evolutionære træ, T. T behøver ikke at være rodfæstet og derfor repræsenteres det med nabolister, hvilket giver os et træ med ikke-orienterede kanter som illustreret på figur 3.1.



Figur 3.1: Et ikke-rodfæstet træ.

Da algoritmen til konstruktion af separatortræet for det evolutionære træ

er afhængig af at træet har en rod, er træet udstyret med operationen *makeRooted*, der vælger en arbitrær knude som rod.[2] Fra denne knude udføres et dybde-først-gennemløb af træet, hvor vi i hver knude vi besøger placerer en pegepind til den knude, vi kom fra. Hermed opnår vi en rodfæstning af træet med pegepinde fra en knude til dens forældre. Figur 3.2 viser, hvordan træet fra figur 3.1 kommer til at se ud efter denne ændring, hvor den arbitrært valgte rod er r. Pegepindene er repræsenteret med de stiplede linier.



Figur 3.2: Træet er nu blevet rodfæstet.

Endelig har hver knude i det evolutionære træ en, muligvis tom, mængde af blade. Foreningen af bladmængderne indeholder alle de arter, der er blevet indsat i træet. Bladene selv behøver blot at have en label, eller en anden form for reference til de arter, de repræsenterer.

3.2 Separatortræ

Under søgningen på indsættelsespunktet for en ny art i det evolutionære træ T, ønsker vi, som nævnt i afsnit 2.1, at minimere søgerummet mest muligt efter hvert eksperiment. Men da T ikke er garanteret at være begrænset af logaritmisk højde kan vi risikere at lave et lineært gennemløb af knuderne i T for at finde indsættelsespunktet. Dette forbedrer vi, ved at benytte et separatortræ.

Først giver vi en abstrakt definition, efterfulgt af en beskrivelse af den nødvendige datastruktur. Dernæst beskriver vi konstruktionen af separatortræet og til sidst vedligeholdelsen under indsættelse af knuder i det tilhørende evolutionære træ og dermed i separatortræet.

[2] Vi vælger altid den første knude i den struktur, hvorved vi repræsenterer træets knuder, men dette er ikke et krav.

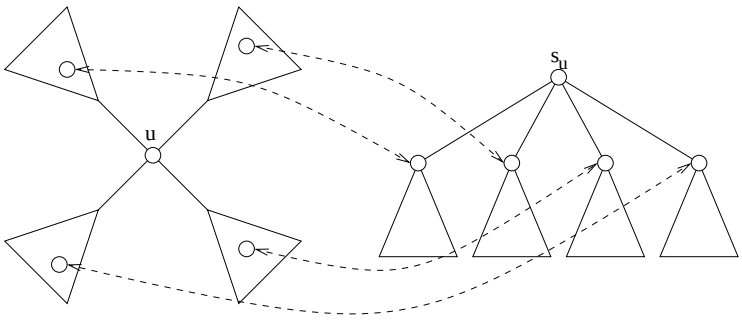
I [BFPÖ01] skelnes der i notationen ikke mellem knuden v og dens separatorknude – for at synliggøre denne forskel bruger vi s_v for separatorknuden for v, ligeledes benytter vi S_T for separatortræet for det evolutionære træ T.

3.2.1 Definition

Den centrale egenskab ved et separatortræ er, at det repræsenterer en rekursiv opdeling af det tilhørende evolutionære træ i undertræer. Følgende er den formelle definition, fra [BFPÖ01] (definition 1). Vi har omskrevet til vores notation, hvor $s_v = s^{(T)}(v), v \in T$:

Definition 2 *Lad T være et ikke-rodfæstet træ med n knuder. Et separatortræ S_T for T er et rodfæstet træ med samme størrelse som T. Den rekursive definition er som følger:*
Rodknuden, s_u, for S_T hører til knuden u i T, kaldet separatorknuden for u. Hvis man fjerner u fra T opdeles T i disjunkte træer T_1,..,T_k, hvor k er antallet af kanter ud af u i T. Børnene af s_u i S_T er rodknuderne af separatortræerne S_{T_1},..,S_{T_k}.

Figur 3.3 viser hvorledes den ovenstående definition giver os den lovede bijektion, $s^{(T)}$, mellem knuder i T og S_T.



Figur 3.3: Her ses hvorledes børnene af s_u har deres tilhørende knuder i undertræer af u. For knuderne i undertræerne er $s^{(T)}$ indikeret med de stiplede pile.

Alene ud fra definition 2 kan der konstrueres flere forskellige separatortræer for et givet træ. For at opnå den ønskede udførelsestid benyttes separatortræet i konstruktionen af det evolutionære træ har vi behov for, at det er balanceret. Det opnår vi ved at benytte 1/2-separatortræer, også kaldet centroidetræer. Disse træer opfylder definition 3 (artiklens definition

2), hvor T_i er undertræerne T_1,..,T_k i definition 2 og $|T|$ er antallet af knuder i T.

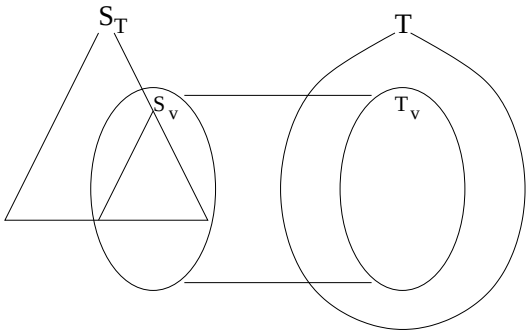
Definition 3 *Et separatortræ er et t-separatortræ, for en værdi $t \in [1/2, 1]$, hvis $|T_i| \leq t|T|$ for hvert T_i og separatortræet for hvert T_i også er et t-separatortræ.*

Ud fra definitionen ses det, at hvert niveau i et 1/2-separatortræ vil svare til mindst en halvering af antallet af knuder. Det gælder uanset strukturen af det tilhørende træ. Herved er højden på 1/2-separatortræet begrænset af $\lceil \log n \rceil$, hvor n er antallet af kanter i træet.

Til nedenstående faktum skal vi bruge følgende definition, der ikke er udtrykt eksplicit i artiklen:

Definition 4 *For et givet separatorundertræ, S_{s_v}, rodfæstet ved s_v, definerer vi den inducerede undergraf, T_v, af T, som mængden af knuder i T, hvor de tilhørende separatorknuder ligger i S_{s_v}, og alle kanter i T, hvor begge endepunkter er med i den inducerede graf.*

Figur 3.4 viser hvordan T_v er indlejret i T.



Figur 3.4: Undertræet S_{s_v} inducerer en undergraf T_v af T.

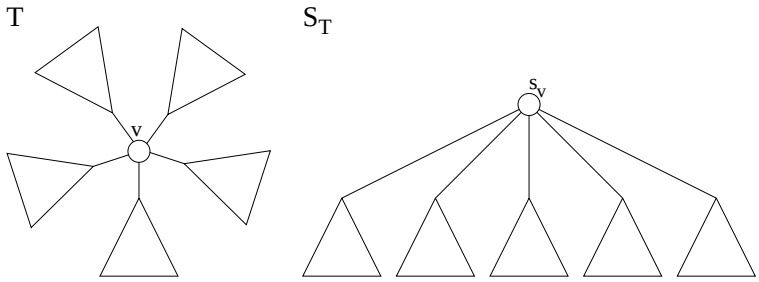
Med denne definition kan vi til vedligeholdelsen af separatortræet bruge følgende faktum:

Faktum 1 *Givet et separatortræ S_T for det evolutionære træ T, lader vi v være en knude i T og s_v være dens separatorknude. Vi lader nu S_{s_v} være undertræet af S_T, der er rodfæstet ved s_v. Så gælder følgende:*

1. *Undergrafen T_v af T, der induceres af S_{s_v}, er et træ, og S_{s_v} er et gyldigt separatortræ for T_v.*
2. *Givet en kant i T med præcist et endepunkt i T_v, er det andet endepunkt en knude u, hvis separatorknude, s_u, er en forfader til s_v. Yderligere kan hver knude w, hvis separatorknude, s_w er forfader til s_v, højest være endepunkt for en sådan kant.*

Dette faktum fremsættes i [BFPÖ01] uden bevis – vi vil her bevise det ved induktion på undertræer af T.

I basistilfældet, $T_v = T$, er den første egenskab trivielt opfyldt, da T_v er hele træet og $S_{s_v} = S_T$. Af figur 3.5 ses også, at der oplagt ikke er nogen kanter med præcist et endepunkt i T, hvorfor den anden egenskab er opfyldt.

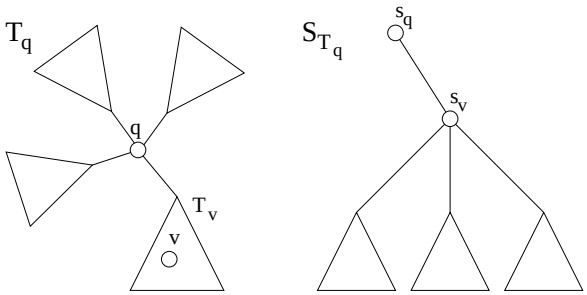


Figur 3.5: I basistilfældet for faktum 1 ses det, at den inducerede graf er hele T, og at der ikke er nogen kanter med præcis et endepunkt i T.

For de øvrige tilfælde betragter vi figur 3.6. Her ser vi, hvorledes T_v er det undertræ, der opstår ved at fjerne q fra T_q. Hermed følger at den første egenskab er opfyldt for alle T_v. For at vise den anden egenskab i faktum 1 observerer vi, at der er én kant ud af T_v. Denne kant ender i q. Da kanter med endepunkt i $T \setminus T_q$ også går ud af T_q følger egenskab 2 per hypotese.

3.2.2 Datastruktur

Separatortræet repræsenteres ved et træobjekt, der indeholder rodknuden og plads til en række yderligere oplysninger, vi beskriver senere. Ved at have et



Figur 3.6: For det induktive skridt ser vi at kun en kant ud af T_v ender i q.

separat objekt for træet kan vi under vedligeholdelsesprocedurerne, som vi beskriver i sektion 3.2.4, udskifte knuderne i træet uden at andre objekter skal berøres.

Ydermere har separatortræet en metode til tilføjelse af et nyt barn, hvorfra de nødvendige vedligeholdelsesprocedurer kan kaldes. Disse vil vi vende tilbage til senere.

Endelig har knuderne en pegepind til de tilsvarende knuder i det evolutionære træ, og tilsvarende pegepinde tilføjes til dettes knuder. Herved kan vi i konstant tid få den evolutionære knude, u, givet dens separatorknude, s_u, og omvendt.

3.2.3 Konstruktion

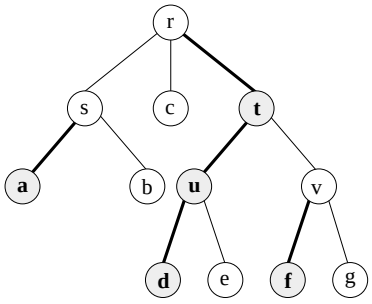
Her vil vi gennemgå to algoritmer til konstruktion af separatortræer. Først en simpel, $O(n \log n)$ algoritme, som følger lemma 1 i [BFPÖ01]. Efterfølgende kommer en forbedret algoritme, der benytter effektive datastrukturer og herved reducerer udførelsestiden til $O(n)$, per artiklens lemma 2.

Inden vi går igang med at gennemgå de to måder at bygge separatortræer på, får vi brug for følgende definitioner af begreberne *Tungt barn* og *Tung sti*, disse er defineret på side 6 i [BFPÖ01]:

Definition 5

- *Størrelsen af en knude v ($|v|$): Givet en rodfæstning af det evolutionære træ, defineres $|v|$ til at være antallet af knuder i undertræer rodfæstet ved v. Bemærk, at v tælles med i størrelsen af sit undertræ, dvs. $|rod_T| = |T|$.*
- *Tungt barn: Det barn af en knude, der er af maksimal størrelse. I tilfælde af at to eller flere knuder er lige store vælges en arbitrær som tungt barn.*
- *Tung sti: Kanterne mellem tunge børn og deres forældre definerer en dekomposition af træet i disjunkte tunge stier.*

På figur 3.7 er træet fra figur 3.2 dekoreret med fremhævning af tunge børn og stier.



Figur 3.7: Træets tunge børn og stier.

Den første algoritme er den mest naive metode, hvor vi kun anvender definitionen af tunge børn og stier. For hvert skridt i konstruktionen genberegnes en knudes tunge barn og dermed dens tunge sti. Resultatet af algoritmen er følgende lemma:

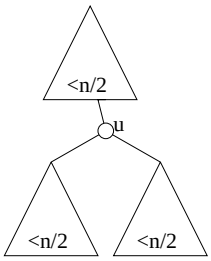
Lemma 1 *Givet et træ T med n knuder, så kan et 1/2-separatortræ S_T bygges for T i $O(n \log n)$ tid.*

Beskrivelse

Først udfører vi *makeRooted* på træet, således at det rodfæstes, og i hver knude registreres, hvem der er dennes forældre. Dernæst foretager vi et dybde-først gennemløb af T, hvor vi beregner størrelsen af alle undertræer og

denne beregnede størrelse lagres i rodknuden for det pågældende undertræ. Yderligere registreres det største barn mht. størrelsen som knudens tunge barn, hvilket giver os de tunge stier for T, disse afsluttes med bladene i T.

For at finde den knude, der skal være separatorknude, tages udgangspunkt i rodknuden r i T. Fra r følges den tunge sti indtil vi finder den knude u, der er længst nede i træet, hvorom det gælder at $|u| \geq n/2$. Ved hjælp af figur 3.8 ser vi at u er en 1/2-separatorknude. Da $|u| \geq n/2$ er træet over u mindre end $n/2$, og da u er valgt så langt nede i træet som muligt, er alle undertræer af u mindre end $n/2$.



Figur 3.8: Separatorknude u vælges, så den deler træet i deltræer af størrelse $< n/2$.

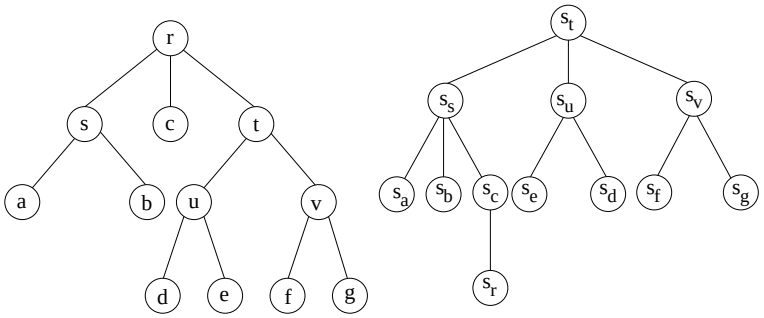
Vi kæder nu separatorknuden s_u sammen med u, og lader s_u blive rod i separatortræet S_T. u fjernes nu fra T, hvilket opdeler T i disjunkte træer $T_1, .., T_k$. Derefter gentages proceduren rekursivt på de enkelte T_i'er og roden af de separatortræer, der kommer fra T_i'erne, bliver børn af s_u i S_T. Ved hvert rekursivt kald gennemløber vi træet T_i for at reetablere tunge børn og stier.

På figur 3.9 ser vi resultatet af ovennævnte algoritme på træet fra figur 3.1.

Under konstruktionen af separatortræet repræsenterer vi fjernelsen af u fra træet ved at fjerne den fra dens naboers naboliste. Hermed kan vi sikre, at rekursionen kun fortsætter i dele af træet, som ikke allerede er blevet bearbejdet. Da datastrukturen for T efter konstruktionen stadig skal repræsentere alle kanter ved dobbelte pegepinde, reetablerer vi dem på tilbagevejen gennem de rekursive skridt i konstruktionen.

Udførelsestider

[BFPÖ01] beviser for lemma 1 en udførelsestid på $O(n \log n)$. Argumentet hænger på balanceringen af separatortræet, som følger af, at hvert niveau i



Figur 3.9: Træet og et separatortræ for dette.

separatortræet repræsenterer mindst en halvering af det største separatortræ for de underliggende træer. Det giver os, at separatortræet er $O(\log n)$ højt. Hvert niveau i separatortræet kræver en konstruktion af tunge stier og efterfølgende et gennemløb af dem for at finde separatorknuderne. For hvert niveau er dette lineært i antallet af knuder i træet, hvilket er $O(n)$. Følgeligt bliver hele konstruktionen $O(n \log n)$.

Vi har verificeret denne udførelsestid eksperimentelt, se afsnit 5.6.1 for de præcise resultater.

Diskussion

Lemma 1 indeholdt ingen tvetydigheder eller problematikker og var derfor meget ligefremt at få realiseret.

Med det i hånden kan vi konstruere separatortræer, men $O(n \log n)$ er ikke hurtigt nok til separatortræernes tiltænkte anvendelse.

For at opnå den udførelsestid for rekonstruktion af det evolutionære træ, som artiklen præsenterer, er det et krav at vi kan konstruere et separatortræ hurtigere end resultatet fra lemma 1. Det fører os videre til lemma 2, der siger følgende:

Lemma 2 *Givet et træ T med n knuder, så kan et 1/2 separatortræ S_T bygges for T i $O(n)$ tid.*

Beskrivelse

Algoritmen, der realiserer lemma 2, følger grundlægende den samme fremgangsmåde som lemma 1, blot udnytter vi, at informationer om tunge børn

og stier kan vedligeholdes med væsentligt mindre arbejde end en fuldstændig genberegning. Ydermere udnytter vi, at knudernes størrelse er monotont faldende, til at finde separatorknuden på en tung sti hurtigere end ved at gennemløbe stien.

Ligesom i den foregående metode rodfæstes træet T og derefter beregnes størrelsen af alle knuderne i T. Når dette er gjort, konstrueres der et søgetræ for hver af de fundne tunge stier. Dette kan vi gøre i lineær tid ved at placere stien i en eksplicit liste under den rekursive konstruktion og konstruere søgetræet for denne liste, i tid proportional med størrelsen af den tunge sti, når vi kommer til den øverste knude i stien. Her gemmer vi så en pegepind til søgetræet. Knuderne er initielt placeret i søgetræet med deres størrelse som nøgle. I det forrige lemma genberegnede vi størrelserne af knuderne, efter at en knude var blevet udvalgt til separatorknude. Da dette tager tid, benytter vi i stedet, at vi kan justere knudernes nøgle i søgetræet effektivt. For mere information om det brugte søgetræ henvises til afsnit 3.6.1, hvor de nødvendige metoder gennemgås.

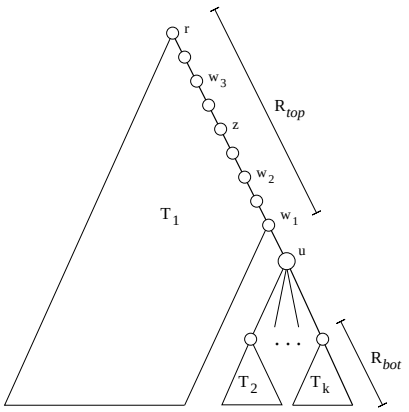
I hver knude opretholder vi en prioritetskø over de børn, der ikke er det tunge barn. Til konstruktionen af denne prioritetskø benyttes en lineær tids konstruktionsmetode. Herved kan vi senere, i logaritmisk tid, tilgå det næsttungeste barn af en knude for at opdatere denne knudes tunge barn. Se afsnit 3.6.2 for mere information om prioritetskøen.

Separatortræet konstrueres nu rekursivt som før. Dog benyttes søgetræet, der indeholde den tunge sti, startende med roden for det træ vi bygger for, til at lokalisere separatorknuden, fremfor at søge lineært gennem den tunge sti.

Separatorknuden findes således: Roden r har den maksimale nøgle i søgetræet, da den er den største knude i træet. Hermed kan vi med forespørgslen *successor(*$|r|/2$*)*, der giver den knude u, længst nede i T, hvor $|u| \geq |r|/2$, finde en gyldig knude u til at blive rodknuden s_u i separatortræet S_T. Størrelsen, $|r|$, findes ved at benytte metoden *key(r)*. Gyldigheden indses ved, at alle børn v af u i T opfylder at $|v| \leq |r|/2$. Så fjernes u fra T på samme måde som i den foregående algoritme. Dette opdeler T i k disjunkte undertræer $T_1, .., T_k$. Størrelsen n_i af hvert undertræ T_i er per konstruktion maksimalt $n/2$, altså $n_i \leq n/2$. Den ovennævnte procedure foretages rekursivt på T_i'erne, således at vi får opbygget 1/2-separatortræer for hvert T_i, igen benytter vi tricket med at fjerne u fra nabolisterne for rødderne af T_i'erne, således at vi styrer rekursionen den rigtige vej. Rødderne i separatortræerne for T_i'erne bliver børn af s_u i S_T.

For hvert skridt i rekursionen skal vi, efter at have fundet knuden u, opdatere søgetræet R for den tunge sti indeholdende u. Dette gøres ved

at benytte metoden *split(u)*, der deler søgetræet ved u i to nye søgetræer med henholdsvis de elementer med større nøgler end u og de elementer med mindre nøgler end u samt u. Ved at benytte *split(u)* to gange får man tre dele: knuden u og søgetræerne R_{top} og R_{bot}. R_{top} indeholder den del af den tunge sti, der er over u i T, og R_{bot} indeholder den del, der ligger under u i T. Dette skyldes, at den tunge sti er monotont faldende, dvs. at den mindste nøgle i R_{top} er større end den største nøgle i R_{bot}. På figur 3.10 ses denne opsplitning af søgetræet.



Figur 3.10: Separatorknuden u på den tunge sti $R = R_{top} \cup \{u\} \cup R_{bot}$, og knuderne $w_1, ..., w_k$. Bemærk at T_1 er stort på figuren, på trods af at det i lighed med de øvrige undertræer har størrelse $< n/2$.

R_{bot} kan let opdateres; det tunge barn hc for u i T skal have en reference til R_{bot}, da hc er det maksimale element i R_{bot}, og dermed roden på den tunge sti, som R_{bot} indeholder. Herfra følger, at alle træerne $T_2, .., T_k$, der var rodfæstet på u i T, har de korrekte søgetræer til deres tunge stier.

I modsætning til opdateringen af R_{bot} kræver det lidt mere arbejde at få R_{top} opdateret. For at opdatere denne, skal alle søgetræer og tunge stier i træet T_1 opdateres. Dette indebærer to ting: Alle nøglerne, dvs. størrelserne, i R_{top} skal opdateres, da disse er blevet mindre med fjernelse af u og de dertil rodfæstede undertræer. Dette gøres ved hjælp af metoden *addpathcost(-key(u))*, der modificerer alle nøgler i søgetræet, hvor nøglen hentes fra søgetræet R_T for T. Det andet der skal opdateres er, at de nye tunge børn skal findes og arrangeres korrekt i søgetræer.

Til denne opdatering defineres knuderne $w_1, ..w_l$, hvor $l \leq \log n$, som

følger:

Definition 6

- w_1 *er forælderen til* u *i* T*.*
- w_{i+1} *er den forfader til* w_i *i* R_{top}*, der er givet ved successor(*$2 \cdot |w_i|$*), hvor* $|w_i| = key(w_i)$*.*

Indholdet af definition 6 er at finde i [BFPÖ01] på side 9, definitionene er ikke navngivet i artiklen.

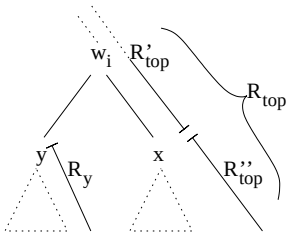
På figur 3.10 er der som eksempel markeret knuderne w_1, w_2, w_3. Per konstruktion er $|w_{i+1}| \geq 2 \cdot |w_i|$ og da $|T_1| \leq n/2$, ved valget af u som separatorknude, følger at $|w_i| \geq 2^{i-1}$ og at $l \leq \log n$. Det skal nu indses, at $w_1, .., w_l$ er de eneste knuder i R_{top}, hvis tunge barn muligvis ikke længere er det største barn og derfor finder vi det nye tunge barn. Betragt nu knuden z fra R_{top}, der er placeret på en sti mellem w_i og w_{i+1}. Argumentet går på at vise, at z's børn er korrekt placeret, dvs. at det barn, der er i R_{top}, stadigvæk er z's tunge barn i T_1. Dette indses ved at betragte $|w_i| < |z| < 2 \cdot |w_i|$, hvorfra det ses at $|w_i| > |z|/2$. Dette betyder, at der ikke eksisterer et andet undertræ rodfæstet ved et barn af z, der er stort nok til at kunne indeholde et nyt tungt barn for z.

Selve opdateringen foregår således: For at opdatere de enkelte w_i'er kigges på x, som var det tungeste barn af w_i i T og det næsttungeste barn y, som findes i prioritetskøen, Q_{w_i}, over de andre børn af w_i. Hvis Q_{w_i} er tom, dvs. w_i ingen andre børn har, skal w_i ikke opdateres. Ellers er der to tilfælde, der skal undersøges: hvis $i = 1$ og hvis $i \geq 2$.

Hvis $i = 1$, så er $x = u$, hvilket gør at y bliver det nye tunge barn for w_1 og y skal fjernes fra Q_{w_1}. Så flettes R_{top} sammen med søgetræet, der indeholder den tunge sti, der starter ved y, vha. *join*-metoden, der kombinerer to søgetræer til et, og det resulterende søgetræ gemmes i w_1.

Hvis $i \geq 2$, er der igen to muligheder: Hvis $|x| \geq |y|$ i T_1, dvs. at x er mindst ligeså tung som y, så er x også tungt barn for w_i i T_1. Ellers er x ikke tungt barn for w_i i T_1, og derfor skal den tunge opdateres. Dette gøres ved at splitte R_{top} ved x, hvilket giver to søgetræer R'_{top}, som indeholder alle de knuder, der er på stien mellem roden af T_1 og w_i og R''_{top}, der indeholder knuderne på den tunge sti, der starter ved x. Da y skal være det nye tunge barn for w_i fjernes denne fra Q_{w_i} og x tilføjes til Q_{w_i}, da R''_{top} indeholdte den tunge sti, der startede med x, så får x en reference til denne. Så skal søgetræerne R'_{top} og søgetræet R_y, der indeholder den tunge sti, startende ved y, sættes sammen ved hjælp af *join* operationen og det resulterende træ

gemmes i y. Når w_i er blevet opdateret, kaldes rekursivt på w_{i+1}. Hvorledes R_{top}, R'_{top}, R''_{top} og R_y er placeret i forhold til hinanden kan ses på figur 3.11.



Figur 3.11: Hvis w_i's barn y er tungere end x, skal y gøres til tungt barn af w_i.

Når vi når til w_l, repræsenterer alle søgetræer i T_1 til T_k de korrekte tunge stier.

Udførelsestider

Med konstruktionsalgoritmer i lineær tid til søgetræer og prioritetskøer, er forarbejdet til denne algoritme klart $O(n)$. Da vi konstruerer det samme separatortræ som i den foregående algoritme, er der fortsat $O(\log n)$ niveauer. På hvert af de niveauer benytter vi nu, i stedet for et lineært gennemløb, søgetræerne og prioritetskøerne. Alle søgetræsoperationerne tager højst logaritmisk tid, se afsnit 3.6.1, og alle operationer udført på prioritetskøen tager højst logaritmisk tid, se afsnit 3.6.2. Det, sammenholdt med, at der maksimalt er $\log(n)$ skridt at opdatere R_{top} i, giver at hvert niveau kommer til at tage $O(\log^2 n)$ tid. Herved domineres den samlede udførelsestid af $O(n)$ for det indledende gennemløb.

Igen har vi ved eksperimenter verificeret at implementationen overholder den forventede udførelsestid. Se afsnit 5.6.1.

Diskussion

I sammenligning med den algoritme lemma 1 gav anledning til, så var den algoritme som lemma 2 gav anledning til væsentligt mere omfattende og kompleks.

Specielt viste det sig hurtigt, at vi ikke kunne finde standardpakker med de datastrukturer, der opfylder alle kravene i [BFPÖ01]. Derfor var vi nødt til selv at implementere de forskellige datastrukturer, der bliver beskrevet i afsnit 3.6. Specielt viste referencen til søgetræer sig at være upræcis, da

de træer, der beskrives i [Tar83] kapitel 5, ikke er dem, der skal bruges. Derimod kan de ønskede operationer bygges på splaytræer som beskrevet i [Tar83] kapitel 4, med en konstruktion, der ligger tæt på datastrukturen **Path**.

Ligeledes skulle den heapbaserede prioritetskøs datastruktur implementeres. I modsætning til søgetræet var denne mere velkendte datastruktur ikke svær at konstruere, men ikke desto mindre krævede det en del gennemtestning at sikre korrekt opførsel i alle specialtilfælde.

Nu kan vi effektivt konstruere det separatortræ, vi omtalte i afsnit 2.1, som skal bruges til at reducere søgerummet effektivt i forbindelse med søgningen efter et indsættelsespunkt.

Vi har i dette afsnit først vist en algoritme til at konstruere separatortræet S_T for det evolutionære træ T i $O(n \log n)$ tid og efterfølgende har vi reduceret konstruktionstiden til $O(n)$, hvor n er antallet af knuder i T. Dette resultat får vi behov for i det næste afsnit, hvor vi vedligeholder det konstruerede separatortræ under indsættelser af knuder i T og S_T.

3.2.4 Vedligeholdelse

Da det er muligt at indsætte ekstra knuder i træet T, hvilket kræver at de tilhørende knuder indsættes i S_T, skal to aspekter håndteres: hvordan indsættes nye knuder, og hvordan vedligeholdes højden af separatortræet under disse indsættelser.

En ny knude kan indsættes i træet T på en af to måder:

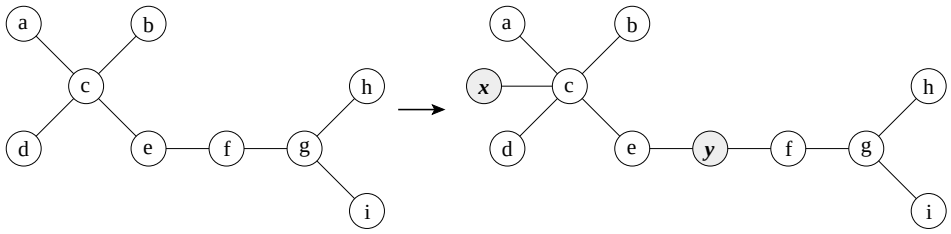
- Type 1: Tilføjelse af en bladknude, der bliver forbundet via en ny kant til en knude u allerede i T.

- Type 2: Indsættelse af en ny knude mellem to knuder allerede i T, dvs. at en eksisterende kant brydes op i to kanter.

På figur 3.12 har vi eksemplificeret disse to typer af indsættelser. Knuden x bliver indsat ved en Type 1 indsættelse på knuden c og knuden y bliver indsat vha. af en Type 2 indsættelse mellem knuderne e og f.

Efter indsættelse af knuden, u, i T skal den nye knudes separatorknude, s_u, indsættes i separatortræet S_T. De to typer håndteres således:

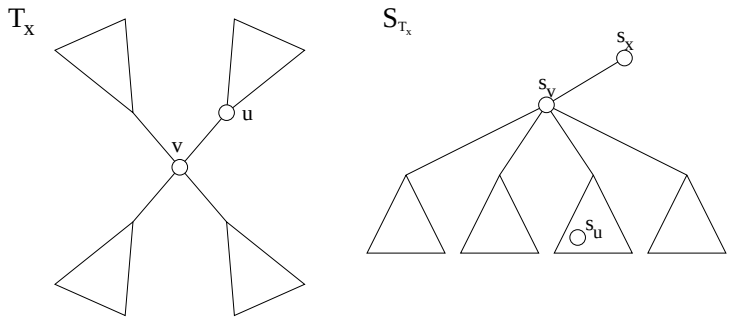
- Type 1: Knuden s_u indsætttes som barn på separatorknuden for us forfader i T.



Figur 3.12: De to typer af indsættelser, der kan forekomme i et træ.

- Type 2: Knuden s_u indsættes som barn til den af de to separatorknuder, som svarer til dens naboer i T, der sidder længst nede i separatortræet. Bemærk, at de to oprindelige knuder i T sidder på den samme sti fra roden til et blad i separatortræet S_T jvf. faktum 1.

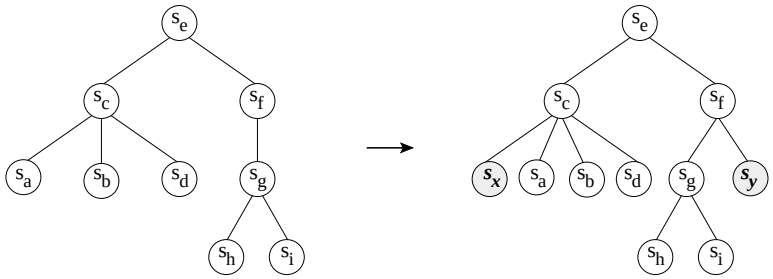
Den ovenstående bemærkning kan også indses på følgende måde, der ikke er nævnt i [BFPÖ01]: Hvis vi i et træ T_x, har et kant-par (v, u). Så vil en af disse knuder blive valgt før den anden til at blive repræsenteret i separatortræet. Per definition af et separatortræ så må s_u tilhøre S_{s_v} ellers må s_v tilhøre S_{s_u}. Da der altid er en sti fra en knude til roden i den træ, er det ønskede opnået. På figur 3.13 betragter vi kun det første tilfælde, da det andet er symmetrisk.



Figur 3.13: u og v er et kant-par i T_x og det ses, at stien fra s_u til s_x går igennem s_v i S_{T_x}.

Figur 3.14 viser indsættelsen af de tilhørende separatorknuder, s_x og s_y til de knuder, x og y, der blev indsat på figur 3.12. s_x er blevet indsat som

barn af separatorknuden s_c. Det ses endvidere, at s_y er placeret som barn af s_f, da det er denne, der sidder længst nede i separatortræet af s_e og s_f.



Figur 3.14: Håndtering af de to typer indsættelser i separatortræet, svarende til indsættelserne i figur 3.12.

Nu når vi ved, hvordan de to mulige typer indsættelser skal håndteres, bemærkes det, at disse indsættelser i separatortræet kan medføre at dette ikke længere er balanceret. I det følgende gennemgås to metoder, hvorved man kan vedligeholde separatortræet under indsættelser af knuder og dermed begrænse højden af separatortræet.

Den generelle strategi for begge metoder er at genbygge hele separatortræet efter der er foretaget et vist antal indsættelser.

Det nedenstående lemma beskriver udførelsestiden for vedligeholdelse af et separatortræ per indsættelse.

Lemma 3 *For ethvert $0 < \epsilon < 1/4$, kan et $(1/2 + \epsilon)$-separatortræ vedligeholdes i amortiseret $O((\log n)/\epsilon)$ tid per indsættelse, givet at starttræet er et 1/2-separatortræ.*

Beskrivelse

Her bruger vi variable for størrelse og dybde i separatorknuderne, hvorfor de skal tilføjes.

Disse informationer holdes opdateret når der indsættes en ny knude i træet. Dette gøres ved gå langs stien fra den indsatte knude til roden og på denne tur opdatere informationerne. På turen finder vi endvidere eventuelle knuder, der overtræder 1/2-separatorknudeegenskaben, altså at størrelsen af separatorknuden er større end halvdelen af dens umiddelbare forfaders størrelse. Sådanne knuder kan opstå efter et vist antal indsættelser i træet. Hvis der er mere end en knude, der overtræder denne egenskab, vælger vi den knude s_v, der er tættest på roden af S_T og genbygger separatortræet

for de træer T_{v_i}, som er rodfæstet ved v. Til denne genbygning benyttes algoritmen fra lemma 2, så undertræet genbygges i lineær tid af undertræets størrelse. Når S_{T_v} er blevet genbygget, gennemløbes det rekursivt og dybde- og størrelse-variablerne opdateres.

Udførelsestider

Den ønskede udførelsestid opnås ved at betragte følgende amortiseringsargument. Lad s_u være det største barn af s_v før træet skal genbygges, hvilket giver at $|s_u| > (1/2 + \epsilon)|s_v|$. Umiddelbart efter den sidste gang vi genbyggede det træ, der er rodfæstet ved s_v, var s_u enten ikke til stede eller vi har at $|s_u|_{\text{før}} \leq |s_v|_{\text{efter}}/2 \leq |s_v|_{\text{efter}}$. Da $|s_u|_{\text{efter}} > (1/2 + \epsilon)|s_v|_{\text{efter}}$, er der mindst foregået $\epsilon|s_v|_{\text{efter}}$ indsættelser under s_v siden sidste genbygning. Ved at opkræve $O(1/\epsilon)$ tid per indsættelse kan vi dække rekonstruktionen ved s_v, og da separatortræet er $O(\log n)$ højt, får vi i alt $O((\log n)/\epsilon)$ per indsættelse, amortiseret.

Diskussion

De problemer, der opstod med at implementere algoritmen som lemma 3 giver anledning til, skyldtes det design, vi havde benyttet i programmet. Specielt var der endnu ikke taget højde for at separatortræet kunne ændre sig. For at løse dette foretog vi store refaktoreringer af koden på dette tidspunkt. Udover at inkludere passende referencer, så udskiftningen af knuder i separatortræet kan ske transparent, set fra resten af koden, var det også her, vi fik behov for at implementere vedligeholdelsesmekanismen i separatortræet. Heldigvis har Eclipse gode refaktoreringsværktøjer, hvilket lettede arbejdsbyrden. Efter refaktoreringen gik det let med at få implementeret algoritmen.

Med lemma 3 i hånden kan vi vedligeholde separatortræet under indsættelser af knuder. Vha. af lemma 2 og lemma 3 får vi følgende følgende lemma:

Lemma 4 *Et* $(\frac{1}{2} + \frac{1}{3 \cdot \lceil \log n \rceil})$*-separatortræ kan vedligeholdes med en højde begrænset af* $\lceil \log n \rceil$ *i amortiseret tid* $O(\log^2 n)$ *per indsættelse, givet at det separatortræ, der startes med, er et 1/2-separatortræ*

Beskrivelse

Udover at vedligeholde separatortræet, lover lemma 4 også at højden på separatortræet bliver begrænset af $\lceil \log n \rceil$. Måden dette gøres på bygger

videre på lemma 3, med $\epsilon = \frac{1}{3 \cdot \lceil \log N \rceil}$, hvor N er en 2-potens, som er større end eller lig med n. N initialiseres til den mindste 2-potens, der større end eller lig med $2 \cdot n$, dvs. $N = 2^{\lceil \log n \rceil + 1}$. Når n bliver større end N fordobler vi N. Dette betyder at ϵ ændrer sig, og når dette sker genbygger vi hele separatortræet vha. algoritmen fra lemma 2 og på den måde får vi et helt nyt 1/2-separatortræ, der opfylder kravene fremstillet i lemmaet.

Udførelsestider

Bemærk at separatortræets størrelse n skal fordobles før der skal laves en total genbygning af separatortræet, så amortiseret kan de $n/2$ indsættelser, der forekommer efter den foregående genbygning, finansiere $O(n)$ tid til genbygningen med et konstant bidrag. Så fra lemma 3 fås at den amortiserede udførelsestid for indsættelse bliver $O((\log n)/\epsilon) = O((\log n)/(\frac{1}{3\lceil \log N \rceil})) = O(\log^2 n)$.

Beviset for højdebegrænsningen kan ses på side 12 i [BFPÖ01].

Diskussion

Der var ikke de helt store problemer med at implementere den algoritme som lemmaet giver anledning til, specielt fordi vi under arbejdet med algoritmen for lemma 3 havde foretaget de restruktureringer af koden, der var nødvendige for at realisere lemma 3 og 4.

Med de resultater, vi har beskrevet indtil nu, kan vi konstruere et separatortræ i $O(n)$ tid samt vedligeholde dette under indsættelse i tid $O(\log^2 n)$ per indsat knude. Vha. lemma 4 er vi også garanteret at højden af træet er begrænset af $\lceil \log n \rceil$.

3.3 Sample

Her vil vi præsentere datastrukturen *sample*, der, som skitseret i afsnit 2.1, gør det muligt at vedligeholde et separatortræ hurtigere end $O(\log^2 n)$ tid per indsættelse.

Samplen er et træ, hvis knuder med en passende morfi, som vi vil definere i afsnit 3.3.1, relaterer til en delmængde af knuderne i T. Vi vil også definere, hvordan separatortræet for samplen kan benyttes til at simulere S_T.

Dette afsnit forløber som følger: I afsnit 3.3.1 definerer vi en sample og diskuterer separatortræet for denne. I det efterfølgende afsnit 3.3.2 gennem-

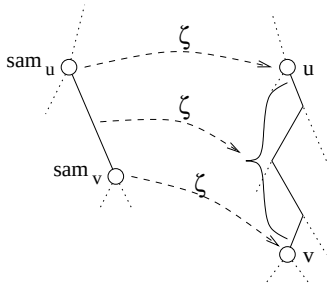
går vi den nødvendige datastruktur og vi slutter af med i afsnit 3.3.3 at beskrive, hvorledes en sample skal realiseres.

3.3.1 Definition

Vi vil nu formelt definere samplen. Definition, der bliver præsenteret her, adskiller sig fra beskrivelsen af samplen i [BFPÖ01] ved at betragte U og T som to træer med passende afbildninger imellem – i stedet for et meget dekoreret træ.

Definition 7 *Samplen for træet T består af et træ, U, og en morfi, $\zeta : U \to T$, således at ζ er en injektiv afbildning af knuderne i U over i en delmængde af knuderne i T. Vi benytter notationen sam_v, hvor $\zeta(sam_v) = v, v \in T$. Kanterne i U skal ζ afbillede over i stier i T, således at hvis sam_u og sam_v er forbundet med en kant i U, så er der en sti mellem u og v i T.*

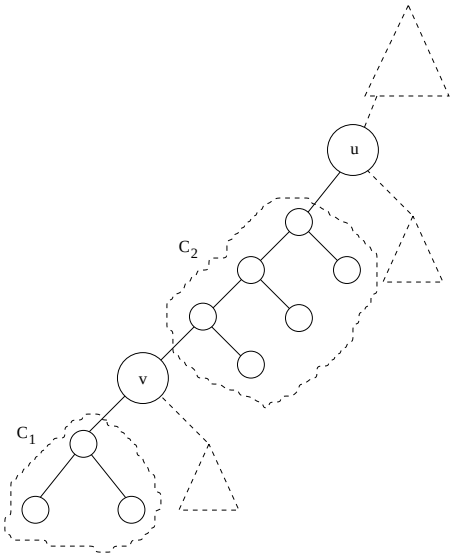
På figur 3.15 viser vi, hvordan morfien ζ virker på to knuder i U og kanten mellem dem. Da U og T er træer, følger det, at hvis stien mellem u og v er ikke-tom og lukkes med alle de knuder i T, som kan nås fra knuder på stien, så har vi en forbundet komponent i T, som har kontakt til præcis to knuder i $\zeta(U)$.



Figur 3.15: ζ afbilleder kanten (sam_u, sam_v) i U over i stien $(u, ..., v)$ i T.

Hvis delmængden $\zeta(U)$ af T fjernes fra T, så står vi med en samling af forbundne komponenter i T. De komponenter, der opstår ved lukningen af en sti mellem to knuder i T betegner vi som kantkomponenter, mens komponenter, der kun er forbundet til en knude i $\zeta(U)$, betegnes som bladkomponenter. Figur 3.16 illustrerer forskellen mellem kant- og bladkomponenter.

Når vi taler om en knude i træet T vil vi sige at "v ligger i samplen", hvis $v \in \zeta(U)$. Mængden af komponenter vil vi betegne med "komponenterne



Figur 3.16: Her ser vi et udsnit af et træ. Knuderne u og v er i $\zeta(U)$. Komponenten C_2 er en kantkomponent og C_1 er en bladkomponent.

induceret af U". Komponenter vil vi generelt betegne med C, og om nødvendigt annotere med den eller de knuder i T, de grænser op til, således at $C_{u,v}$ er en kantkomponent i T, hvor $u, v \in \zeta(U)$.

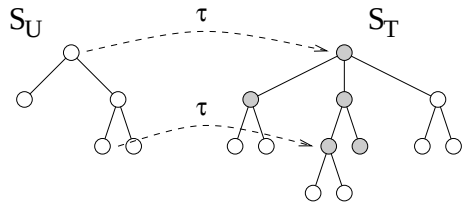
Separatortræ for samplen

Da U er et træ, kan der konstrueres et separatortræ for det. Vi vil nu gennemgå, hvorledes det skal bygges og benyttes, for at vi kan simulere S_T med det.

Vi definerer først afbildningen $\tau = s^{(T)} \circ \zeta \circ (s^{(U)})^{-1}$. Da $s^{(T)}$ og $s^{(U)}$ er bijektioner mellem knuderne i T og S_T henholdsvis U og S_U, afbilleder τ knuderne i S_U injektivt på S_T.

Vi vil nu begrænse os til de tilfælde, hvor τ bliver en rodfæstet træmorfi. Det vil sige at, hvis $rod(S_T T)$ betegner roden i S_T, så $\tau(rod(S_U)) = rod(S_T)$. Og hvis $s_u \to s_v$ betegner kanten mellem s_u og s_v, så $\tau(s_{sam_u} \to s_{sam_v}) = s_u \to s_v$. I disse tilfælde bliver $\tau(S_U)$ toppen af S_T.

Figur 3.17 skitserer, hvorledes τ afbilleder fra S_U til S_T. Vi bemærker her, at flere af knuderne i S_T har børn, som ikke er i $\tau(S_U)$, eftersom kun en delmængde af knuderne i T ligger i samplen.



Figur 3.17: Træmorfien τ på S_U rammer de med gråt markerede knuder i S_T.

Vi kan nu observere, at de resterende knuder i S_T, dem der ikke rammes af $\tau(S_U)$, præcist udgør de komponenter, som U inducerer. Da en sammenhængende komponent i et træ selv er et træ, kan vi konstruere separatortræer for komponenterne. Vi vil benævne disse separatortræer med S_C, hvor C er komponenten. S_T er således $\tau(S_U)$ udvidet med separatortræerne for de komponenter U inducerer.

Ideen er nu, at da $\tau(S_U)$ og komponenternes separatortræer tilsammen udgør S_T, så undlader vi helt at konstruere S_T, og konstruerer i stedet U og S_U. Ved at benytte disse sammen med komponenternes separatortræer, opnår vi at en indsættelse i T kan håndteres ved en opdatering af S_U eller den relevante komponents separatortræ. I afsnit 3.3.3 viser vi, hvorledes U konstrueres, så det simulerede separatortræ S_T bliver balanceret.

For bladkomponenter, C_u, er det oplagt, at S_{C_u} bliver et undertræ af $\tau(s_{sam_u})$ i S_T. For en kantkomponent, $C_{u,v}$ er der to muligheder, men da $\tau(S_U)$ udgør toppen af S_T, skal $S_{C_{u,v}}$ være undertræet for den af $\tau(s_{sam_u})$ og $\tau(s_{sam_v})$, der sidder længst nede i S_T.

3.3.2 Datastruktur

Da vi igen har et træ, som vi skal bygge et separatortræ for, vil vi repræsentere U med underklasser af den datastruktur (for træ og knuder), vi konstruerede til at repræsentere T. Hermed bliver kanterne i U repræsenteret ved kantlister, ligesom kanterne i T.

Træ- og knudeobjekterne skal udvides med referencer til det underliggende træ henholdsvis de underliggende knuder, for at kunne repræsentere ζ. Ligeledes udvider vi datastrukturen for T med referencer den anden vej – til U.

Knuderne i U skal også have en liste over de tilhørende komponenter. Komponenterne repræsenterer vi med en liste for hver komponent over

de knuder, som den indeholder. Separatortræet for komponenten skal også kunne tilgås herfra. Desuden skal hver komponent have en reference tilbage til den knude i U, den tilhører – og i tilfælde af kantkomponenter også den anden naboknude.

3.3.3 Realisering

Med samplen defineret vil vi gennemgå, hvorledes vi konstruerer den, således at træmorfien τ får de ønskede egenskaber.

Da U er et træ, skal vi under konstruktion og vedligeholdelse opretholde som invariant, at hver komponent højest er forbundet til to knuder i $\zeta(U)$. I modsat fald kan komponentens separatortræ ikke entydigt kædes sammen med en knude i S_U. Denne invariant vil vi referere til som invariant A.

For senere at kunne vise nedenstående teorem om den amortiserede tidskompleksitet af at vedligeholde samplen som en simulering af S_T, skal vi yderligere forlange som invariant, at der for alle komponenter gælder at $|C| < \Delta$, hvor $\Delta = 4\lceil(\log n)/4\rceil$. n er her ikke træets aktuelle størrelse, men størrelsen da samplen senest blev konstrueret. Dette vil blive benævnt invariant B.

Teorem 1 *Lad T være et ikke-rodfæstet træ, der indeholder n knuder. Efter lineær tids preprocessering, kan et separatortræ S_T for T med en højde begrænset af* $\log(n+m)+5$ *vedligeholdes under m indsættelser i tid* $O(m \cdot \log(n+m))$.

Beskrivelse

For at opnå resultatet fra teorem 1 vil vi benytte sample strukturen fra de foregående afsnit. Først vil vi gennemgå, hvorledes samplen kan konstrueres, så den opfylder betingelserne fra definition 7 og morfien τ, som defineret via s^U, ζ og s^T, bliver således, at teorem 1 kan opfyldes. Undervejs vil vi skulle opretholde invariant A, for at den konstruerede sample bliver et træ. For senere at kunne overholde det lovede tidsforbrug for vedligeholdelse, skal invariant B overholdes. Den første sample bliver bygget med den modifikation, at $|C| < \Delta/2$. Vi konstruerer altså initielt samplen, således at komponenterne er den halve størrelse af det vi senere vil vedligeholde.

Efter at have konstrueret samplen, vil vi gennemgå vedligeholdelsen af denne. Den basale ide er, at samplen vedligeholdes gennem $m < n$ indsættelser. Hvis $m = n$ konstruerer vi en ny sample for træet, der nu har dobbelt så mange knuder. Herved kan konstruktionen af en ny sample finanseres i konstant tid per indsættelse amortiseret. For hver af de m indsættelser vil vi

opretholde invariant B, som skrevet. Dette gør vi ved at opdele overtrædende komponenter.

Konstruktionen forløber således: Først rodfæster vi træet som for konstruktionen af separatortræet. Det rodfæstede træ gennemløber vi derefter dybdeførst. Under gennemløbet opbygger vi træet U og vedligeholder de komponenter, som det foreløbige U inducerer i den sete del af T. Knuder tilføjes til U, når en af de inducerede knuder bliver større en $\Delta/2$ (Δ hvis ikke det er den første konstruktion af en sample). Roden i T, r, er den første knude, der tilføjes til samplen, således at U består af knude sam_r.

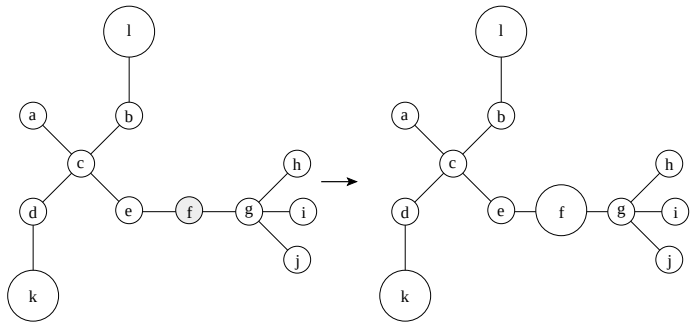
For alle de efterfølgende knuder vil vi benævne den nye knude med v og dens forgænger i gennemløbet w. Knuden v bliver altså besøgt via kanten $w \rightarrow v$. Der er nu to muligheder: Hvis knuden sam_w eksisterer i U, så er vi netop kommet ind i en ny komponent, og v tilføjes som den første knude. Komponenten er en bladkomponent og tilknyttes sam_w. Hvis der ikke findes en knude sam_w i U, så findes der en komponent, C, således at $w \in C$. Til denne komponent tilføjes v.

Det ses at denne fremgangsmåde alene vil konstruere en sample med én knude i U og en bladkomponent for hver kant ud af den valgte rod i T. Med mindre roden deler T i undertræer, der er mindre end $\Delta/2$ store, vil vi undervejs komme til at overtræde invariant B, der så skal genoprettes. Dette kan ske hver gang vi indsætter v i en eksisterende komponent, hvorfor vi sammenligner komponentens størrelse med $\Delta/2$ (eller Δ hvis vi tidligere har bygget en sample).

Hvis invarianten er brudt, så genoprettes den ved at vælge en knude f i C og tilføje sam_f til U. Knuden f vælges så den deler C i nye komponenter med størrelse højest $\Delta/4$; f udvælges efter samme fremgangsmåde som lokalisering af separatorknuden i lemma 1 (side 16), altså beregning af størrelser og den tunge sti, der følges til den rigtige knude. Figur 3.18 viser hvorledes komponenten $\{a, b, c, d, e, f, g, h, i, j\}$ opsplittes ved at knuden f tildeles en sampleknude, sam_f, der tilføjes til samplen. De knuder, der er i samplen, initielt k og l, indikeres på figuren med større cirkler.

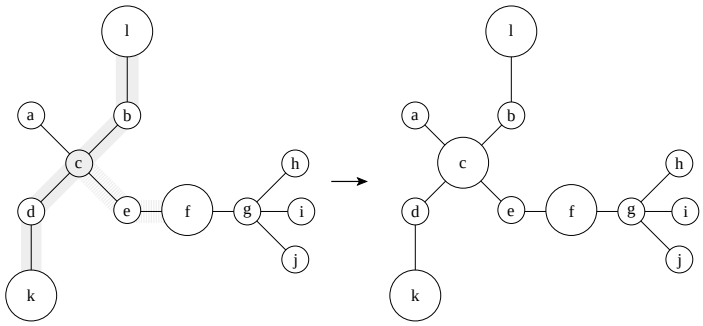
Et split kan, som det gør i figur 3.18, give anledning til, at der opstår komponenter, der er forbundet til mere end to knuder i $\zeta(U)$. I dette tilfælde er det $\{a, b, c, d, e\}$, der er forbundet til f, k og l. Dermed overtrædes den første invariant om samplen. Dette sker, når den nye knude i $\zeta(U)$ ikke ligger på den direkte sti i T mellem de to eksisterende knuder i $\zeta(U)$.

Figur 3.19 viser stien mellem l og k, der består af b, c og d. Løsningen på problemet er at finde den unikke knude, i dette tilfælde c, der ligger hvor stierne mellem de to knuder med eksisterende sampleknuder, k og l, og f,



Figur 3.18: Opsplitningen af kantkomponenten $\{a, b, c, d, e, f, g, h, i, j\}$ resulterer i en kantkomponent med tre naboer

der netop har fået sin sampleknude, mødes. sam_c tilføjes til U på samme måde som sam_f blev det, og invarianten er nu igen opfyldt.



Figur 3.19: Bruddet påinvariant A fra figur 3.18 løses ved at finde frem til c, der også tilføjes til samplen. Herved genoprettes invarianten.

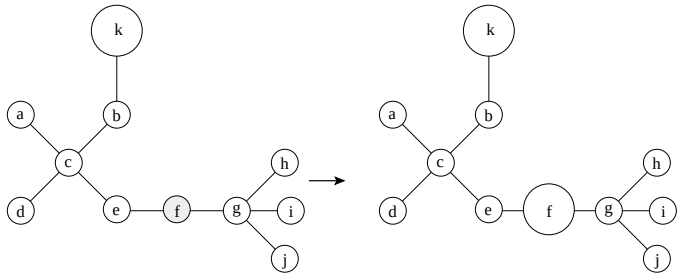
Når samplen og de komponenter, U inducerer, er blevet konstrueret, så bygges der separatortræer for U og samtlige komponenter med algoritmen fra lemma 2. Efter konstruktionen bliver komponenterne hægtet på de rette knuder i U, som tidligere defineret.

Da vi, hver gang en komponent deles, vælger den knude, som er roden i komponentens separatortræ, ses det, at separatortræet S_U bliver toppen af et separatortræ for T. Hermed giver sammensætningen af bijektionerne, $s^{(T)}$ og $s^{(U)}$, og morfien ζ os den lovede τ.

Med konstruktionen på plads skal samplen vil vi i det efterfølgende gennemgå hvorledes strukturen vedligeholdes under indsættelser af knuder. Ideen er som nævnt, at vi rekonstruerer samplen fra bunden, hver gang det underliggende træs størrelse fordobles. Hermed kan vi nøjes med at vedligeholde strukturen under $m < n$ indsættelser.

Ved en indsættelse som et blad (type 1, defineret på side 23), kan det ske, at den knude u, som den nye knude bliver indsat som nabo til, allerede er i samplen. Hvis dette er tilfældet, bliver der dannet en ny bladkomponent indeholdende den nye knude og denne komponent bliver hægtet på sam_u. Alle andre indsættelser bliver håndteret ens, dvs. at ved de andre indsættelser er der en eksisterende komponent C (både blad- og kantkomponent), hvis størrelse vokser med præcis 1.

Efter knuden er blevet indsat i komponenten C, bliver separatortræet for C rekonstrueret vha. algoritmen fra lemma 2, og størrelsen af C tjekkes. Hvis denne er blevet større end Δ, så skal C splittes. Da der er blevet bygget et separatortræ for C, splittes der nu ved roden af separatortræet for C. Ligesom ved konstruktion kan en opsplitning af en kantkomponent give anledning til, at invarianten om maksimalt to naboer ikke overholdes længere, hvilket løses på samme måde som ved den initielle konstruktion. Hvis det er en bladkomponent, der bliver for stor, så kan opsplitningen ikke overtræde invarianten, der opstår derimod altid en kantkomponent. Figur 3.20 illustrerer opsplitningen af en bladkomponent. Efter opsplitningen bygges der separatortræer for alle de nye komponenter.



Figur 3.20: Opsplitning af bladkomponenten $\{a, b, c, d, e, f, g, h, i, j\}$ ved tilføjelse af sam_f til U.

En opsplitning af en komponent giver anledning til mindst én indsættelse i U. Ved balanceringsmekanismen fra lemma 4 bliver S_U holdt balanceret

under disse indsættelser. Når antallet af indsættelser i U, m, når n, lader vi $n = |T|$ og genberegner Δ. Herefter bygger vi en ny sample, som beskrevet ovenfor.

Ved en indsættelse af en knude i U, der ikke giver anledning til en genbygning af separatortræet S_U, kan der opstå en genbygning af et undertræ, $S_{s_{sam_v}}$, af S_U, i henhold til algoritmen fra lemma 3. Dette betyder, at $S_{s_{sam_v}}$ bliver rebalanceret, hvorfor dybden på de enkelte knuder i undertræet kan have ændret sig. Derfor er det nødvendigt at gennemløbe de kantkomponenter, der er koblet på mindst en af knuderne fra $S_{s_{sam_v}}$ og tjekke, at de stadigvæk sidder forbundet til den knude, af de to, der sidder dybest i separatortræet S_U.

Udførelsestider

Først vil vi gennemgå, hvorledes konstruktionen af samplen kan holdes i lineær tid.

Den første observation er, at så længe vi blot tilføjer de besøgte knuder til komponenter, så udfører vi et konstant arbejde per knude. Hermed bliver den interessante del af konstruktionen opbygningen af U. Da vi splitter komponenterne op ved en størrelse på $\Delta/2$ i dele med højest $\Delta/4$ knuder, ved vi, at en opsplitning sker efter mindst $\Delta/4$ tilføjelser til komponenten. Hermed følger, at det samlede antal opsplitninger er begrænset af $4|T|/\Delta$.

Hver opsplitning kræver et gennemløb af komponenten for at finde knuden, der skal deles ved, efterfulgt af et gennemløb for at opbygge de nye komponenter. Hvis invarianten om antallet af naboer overtrædes, så skal vi yderligere splitte en af de nye komponenter op, men da denne opsplitning ikke kan overtræde invarianten, er det også blot et lineært gennemløb af en komponent, der er mindre end den oprindelige komponent til opsplitning. Samlet giver det os, at en opsplitning kan gennemføres i lineær tid i Δ. Hermed bliver den samlede tid for konstruktionen af samplen $O(|T|)$, plus den tid vi benytter til konstruktionen af separatortræerne. Da hver knude i T bliver repræsenteret i præcist et af de separatortræer vi konstruerer, giver lemma 2 os, at separatortræerne også kan bygges i $O(|T|)$.

For vedligeholdelsen behøver vi kun bekymre os om $m < n$ indsættelser, da vi, som nævnt, rekonstruerer samplen hver gang det underliggende træ fordobles i størrelse. Ved det sædvanlige amortiseringsargument er en fordobling nok til at finansere den lineære rekonstruktion.

Hver knude, der indsættes, vil blive placeret i en komponent. Denne kom-

ponent vil i værste fald skulle have sit separatortræ rekonstrueret, hvilket per lemma 2 tager lineær tid i Δ – altså $O(\log n)$ tid. I tilfælde af at komponenten bliver for stor, vil opdelingen i delkomponenter også kunne gennemføres i lineær tid af komponentens størrelse, $O(\log n)$. Ved hver opsplitning bliver en eller to knuder føjet til U. De komponenter, der opstår under vedligeholdelse, enten ved opdeling eller nye komponenter, er aldrig større end $\Delta/2$. Hermed forekommer der højest en overtrædelse for hver $\Delta/2$ indsættelser og dermed højest $2m/\Delta$ overtrædelser for m indsættelser. Hver af disse overtrædelser kan som sagt højest resultere i to indsættelser i samplen U. Disse indsættelser tager per lemma 4 $O(\log^2 |U|)$. Da U er en delmængde af T bliver dette $O(\log^2 n)$.

For m indsættelser kommer vedligeholdelse af samplen U altså til at tage:

$$(2m/\Delta) \cdot \sum_{i=0}^{m-1} O(\log^2(n+i)) \leq O(\frac{m \cdot \log^2(n+m)}{\Delta}) \leq$$

$$O(\frac{m \cdot \log(n+m) \cdot \log(2 \cdot n)}{\Delta}) = O(\frac{m \cdot \log(n+m) \cdot \log(2 \cdot n)}{\log n}) \leq$$

$$O(m \cdot \log(n+m))$$

Det andet ulighedstegn gælder da $m < n$. Ovenstående viser, at vedligeholdelsen kan håndteres i tid $O(m \log(n+m))$.

Teorem 1 lover også, at den samlede højde for det simulerede separatortræ højst er $\log(n+m)+5$.

Først kigger vi på S_U. Her startede vi, jfr. beviset for konstruktionsskridtet, med $4n/\Delta$ splittede komponenter, så der maksimalt kan være $(8n/\Delta)+1$ knuder i U. For $m = 0$ følger det nu, at højden af s_U er $\log(8n/\Delta)+1 = \log(n+m)+4-\log\Delta$. For $m > 0$ observerer vi, at der under vedligeholdelsen kommer højest $2m/\Delta$ opsplitninger, der hver kan tilføje en eller to knuder til U. I værste fald er $|U|$ begrænset af $8(n+m)/\Delta$ og tilsvarende til det ovenstående har vi igen en maksimal højde på $\log(n+m)+4-\log\Delta$.

Da vi har som invariant, at ingen komponenter har mere end Δ knuder, følger det at deres separatortræer har højde $\log\Delta$, da de holdes som 1/2-separatortræer. Når vi tæller de kanter, som ville forbinde separatortræerne for komponenterne med S_U, hvis vi faktisk konstruerede det simulerede S_T, får vi at den samlede højde bliver $\log(n+m)+5$.

Vi har, ligesom ved de andre separatortræskonstruktionsmetoder, også testet om vores implementation af teorem 1 opfylder den lovede udførelsestid. Dette er, som det kan ses i afsnit 5.6.3, tilfældet.

Diskussion

Som nævnt har vi implementeret samplen som en udvidelse af vores datastruktur for det evolutionære træ. Dette sparede os for en del arbejde med at tilpasse konstruktionen af separatortræer. Til gengæld var vi nødt til at håndtere vedligeholdelsen af to forskellige datastrukturer fra indsættelsesmetoderne på den nye og forbedrede træ-datastruktur. Dels skulle det evolutionære træ nu have vedligeholdt sin sample, og dels skulle U have vedligeholdt sit separatortræ.

Konstruktion af separatortræet for U blev hermed løst uden yderligere indsats. Konstruktion af separatortræer for komponenterne er vanskeligere, da vi nu har en yderligere betingelse for de forskellige rekursive gennemløb af datastrukturen, der benyttes ved konstruktionen af separatortræer, nemlig at de skal blive i en komponent. Løsningen er at lade knuderne i det evolutionære træ kende deres komponent og give metoderne i konstruktionen et yderligere argument, der, hvis det er tilstede, repræsenterer den aktuelle komponent. Nu er betingelsen for at fortsætte rekursionen i disse funktioner, at en nabo ikke er blevet besøgt og ligger i den aktuelle komponent, hvor den før blot var, at naboen ikke var blevet besøgt.

Generelt betød introduktionen af samplen, at vi måtte refaktorisere store dele af koden, så den kunne håndtere komponenter. Her var vi meget godt tilfredse med vores valg af Eclipse, der er et solidt udviklingsmiljø, hvor man nemt kan finde de funktioner, der var berørt af eksempelvis en udvidet argumentliste til en metode.

Endelig er opsplitningen af komponenterne ganske udfordrende, da der er mange tilfælde, der nemt kan blandes sammen. Også metoden til at finde den ekstra knude, der skal tilføjes til U ved overtrædelse af invariant 2, krævede en del overvejelser, før vi fandt en fremgangsmåde, der løste problemet i ét gennemløb af komponenten

Den største udfordring ved at implementere samplen, er faktisk at overskue strukturen. Til det formål gennemgik vi en række eksempler i hånden, for at forstå sammenhængene.

Med samplen defineret og konstrueret har vi nu den effektive implementation af separatortræer med logaritmisk højde, som vi i kapitel 2 forklarede behovet for.

3.4 Ordnede separatortræer

I algoritmen til rekonstruktion af det evolutionære træ ønsker vi at kunne gennemløbe børnene af en separatorknude ordnet efter faldende størrelse. Herved kan vi, når algoritmen arbejder sig ned gennem separatortræet, eliminere de størst mulige undertræer først, hvis der skal undersøges mange kandidater før det rette undertræ findes.

3.4.1 Sortering af separatortræet

I dette afsnit beskriver vi hvorledes et almindeligt separatortræ, som beskrevet i afsnit 3.2, sorteres. Ideen er at sortere hele træet på en gang, og derved kunne udnytte bucket-sort algoritmen til at gøre det effektivt.

Lemma 5 *Et separatortræ med n knuder kan i tid $O(n)$ bearbejdes således, at børn af knuderne bliver sorteret i aftagende orden*

Beskrivelse

Først gennemløbes separatortræet og størrelserne på knuderne beregnes. Knuderne opsamles undervejs i en liste.

Dernæst sorteres hele listen ved hjælp af bucket-sort, denne er beskrevet i [GT98].

Nu skal vi have etableret ordningen på en knudes børn, dette gøres ved, at vi i hver knude har en hægtet liste, der indeholder børnene i faldende orden. Denne liste bliver bygget ved at gennemløbe den sorterede liste i voksende størrelse og gøre den knude, man er nået til, til det første barn i forælderens hægtede liste. Når gennemløbet af den sorterede liste er færdigt, er det ønskede opnået.

Udførelsestider

At beregne størrelsen af knuderne i træet kræver et lineært gennemløb af knuderne i træet.

Bucket-sort tager lineær tid og plads – i maksimum af størrelsen på input og den største nøgle i inputtet. Da nøglerne er størrelser på undertræer, er den største nøgle imidlertid lig størrelsen på input, hvorved denne anvendelse af bucket-sort kan gøres indenfor $O(n)$.

For at opbygge de hægtede lister laves et lineært gennemløb af de sorterede knuder, hvor der for hvert enkelt element laves en indsættelse i en hægtet

liste, hvilket er en $O(1)$ operation, eftersom vi altid indsætter forrest. Hvilket samlet giver en udførelsestid på $O(n)$ for at bygge de hægtede lister.

Dermed er den ønskede udførelsestid opnået.

Diskussion

Vi havde ikke de store problemer med at implementere algoritmen der realiserer lemma 5, selve sorteringen blev indsat de steder, hvor vi bygger et separatortræ. Da det er en lineær operation ændrer det ikke på den samlede udførelsestid.

3.4.2 Vedligeholdelse af sortering ved sample

Den ordning af børnene, vi nu har indført, skal også kunne opretholdes i den to-lags struktur, som vi indførte med samplen. Hermed bliver vi nødt til at genoverveje resultaterne fra teorem 1, og tilpasse algoritmen, så de stadig kan opnås.

Teorem 2 *Lad T være et ikke-rodfæstet træ indeholdende n knuder. Efter lineær tids preprocessering, så kan et separatortræ med en ordning af børnene for T vedligeholdes under m indsættelser i tid $O(m \cdot \log(n+m))$, således at højden af træet er begrænset af $\log(n+m)+5$ og at der for enhver sti $(v_1, \ldots, v_\ell)$ fra roden v_1 til en knude v_ℓ i separatortræet gælder at*

$$\Big(\prod_{d_i \leq 2} 2\Big) \cdot \Big(\prod_{d_i > 2} d_i\Big) < 16 \cdot d \cdot (n+m),$$

hvor d_i er det nummer som v_{i+1} har i nummereringen af børnene til v_i, for $1 \leq i < \ell$, og hvor d er $max\{d_1, \ldots, d_{\ell-1}\}$

Beskrivelse

Dette opnås ved en videreudbygning af den måde, hvorpå samplen og dens komponenter bliver konstrueret som beskrevet for teorem 1.

For at opnå en ordning af børnene i to-lags strukturen skal både knuder i og udenfor samplen håndteres. Hvis v ligger i samplen U, så kan den have to slags børn: Dem der også er i U og dem der ikke er i U. Børnene, der er i U, kommer først i ordningen af vs børn efterfulgt af børnene udenfor U. De børn, som også er i U, bliver ordnet efter størrelsen af deres undertræ i separatortræet for U og de andre børn optræder i tilfældig orden. Hvis v ikke ligger i U, betyder det at v er placeret i en komponent: I dette tilfælde

bliver v's børn også ordnet i faldende størrelse, hvor størrelsen er givet ved størrelsen af barnets undertræ i separatortræet for komponenten. Denne ordning kan opnås ved at udføre den sortering, som beskrives af lemma 5, på separatortræet for samplen og separatortræerne for dens komponenter.

Den beskrevne ordning, der ønskes på børnene til en knude, skal vedligeholdes under indsættelser af knuder i træet T, samt ved genbalanceringen af separatortræerne for komponenterne eller separatortræet for U.

Når en ny knude bliver indsat i en komponent, bliver komponentens separatortræ genbygget vha. af algoritmen fra lemma 2. Denne algoritme bliver ligeledes benyttet, hvis der sker en indsættelse af en knude i separatortræet for U, uden at dette giver anledning til en komplet genbygningen af samplen. I disse tilfælde kan ordningen genetableres ved at benytte algoritmen fra lemma 5 i forbindelse med opbygningen af separatortræet for komponenterne henholdsvis samplen. Hvis indsættelsen af en knude i U sker på grund af at en komponent er blevet delt, kan det være, at ordningen af børnene fra hvor den nye knude er blevet indsat i separatortræet for U og op til roden i U's separatortræ skal opdateres.

Denne opdatering skal foretages i alle knuder på stien fra indsættelsesknuden til roden. Ved hver knude skal dens nøgle i den hægtede liste blot forøges med 1, hvilket kan implementeres i konstant tid. Hermed kan denne opdatering udføres i tid proportionalt med højden af træet.

For at fuldføre beviset for teorem 2, mangler vi at bevise at uligheden gælder. Dette er gjort på side 16-18 i [BFPÖ01] og beviset vil derfor ikke blive gengivet her.

Udførelsestider

Da sorteringen per lemma 5 kan gøres i $O(n)$ tid kan vi, ved at benytte den overstående beskrivelse, overholde den tidsgrænse teorem 1 giver. Og da udvidelsen kun består i at lave et ekstra gennemløb af selve træet, uden at ændre på højden af denne, har vi at højdebegrænsningen fra teorem 1 stadigvæk holder.

Udførelsestiden er endnu en gang verificeret eksperimentielt, se afsnit 5.6.2 for detaljer.

Diskussion

Vi oplevede ingen problemer med at få implementeret en ordning på børnene, selv når vi bruger to-lags strukturen. Den kodemæssige udvidelse bestod i at indsætte kaldet til at få bygget disse hægtede lister af børnene de rigtige

steder, dvs. når der blev bygget separatortræer for enten en komponent eller et nyt separatortræ for samplen, samt at håndtere når en indsættelse skete ved en opdeling af en komponent.

Med det ovenstående har vi nu konstrueret en effektiv datastruktur for separatortræer, med børn ordnet efter faldende størrelse. I den efterfølgende algoritme til rekonstruktion af det evolutionære træ vil vi vedligeholde sådan et separatortræ for det evolutionære træ, og benytte det til at gennemløbe træet effektivt.

3.5 Rekonstruktion og vedligeholdelse af evolutionære træer

I dette afsnit vil vi gennemgå selve algoritmen for at rekonstruere et evolutionært træ vha. eksperiment-modellen, samt hvorledes et evolutionært træ kan vedligeholdes under indsættelse af nye knuder.

Først vil vi gennemgå algoritmen til at rekonstruere et evolutionært træ vha. brugen af eksperimenter. Derefter vil vi kort redegøre for udførelsestiden samt forbruget af eksperimenter for algoritmen, både i forbindelse med rekonstruktionen af det evolutionære træ samt ved vedligeholdelse af det evolutionære træ under indsættelse af nye arter.

I det følgende vil vi generelt tale om separatortræet, S_T, selvom det er repræsenteret gennem $\tau(S_U)$ og separatortræerne for de inducerede komponenter. Hvor det er nødvendigt vil vi eksplicit kigge på de enkelte separatortræer for U og dens komponenter.

Den væsentlige egenskab ved knuderne i det evolutionære træ er, i denne sammenhæng, om de er tilstødende til indsættelsespunktet for den nye art, vi ønsker at indsætte, jfr. nedenstående definition, der genfindes på artiklens side 19.

Definition 8 *Et indsættelsespunkt for a siges at være tilstødende på en knude v, hvis et af følgende gælder:*

- *a skal indsættes direkte under v*
- *a skal dele en allerede eksisterende kant, som er tilstødende på v, dvs. at der bliver indsat en ny intern knude på kanten og a indsættes som et blad under denne nye knude*
- *Hvis v er roden i T, så skal der laves en ny rod for T, hvorpå v og a indsættes som børn, henholdsvis som intern knude og blad*

3.5.1 Algoritmen

I dette afsnit vil vi gennemgå selve algoritmen for rekonstruktion af det evolutionære træ vha. eksperimenter i væsentligt flere detaljer end i [BFPÖ01]. Vi har blandt andet gjort algoritmen eksplicit ved at udtrykke den i pseudokode.

For at finde frem til indsættelsespunktet for den nye art a, benyttes separatortræet S_T for de interne knuder i det evolutionære træ T til at søge gennem T for at finde en knude, der er tilstødende til indsættelsespunktet.

Det ovenstående giver anledning til følgende invariant for søgningen efter indsættelsespunktet: Antag, at vi har fundet en knude v i separatortræet S_T for de interne knuder i T og lad S_v være de knuder i T, der er placeret i det undertræ af S_T, som er rodfæstet ved v, inklusiv v. Så er indsættelsespunktet for arten a tilstødende til en af knuderne i S_v.

Søgningen starter i roden af separatortræet. Ved eksperimenter vælges der så for hver besøgt knude et undertræ i det evolutionære træ, hvori søgningen efter indsættelsespunktet skal fortsætte. I undertræet starter vi igen i roden af dets separatortræ. Ved at gentage det ovenstående indtil vi enten møder en knude, hvor eksperimenterne direkte fortæller os, at bladet for den nye art skal indsættes, eller vi når til et blad, til hvilket bladet for den nye art indsættes som nabo. Da vi altid bevæger os nedad i separatortræet, vil antallet af rekursive skridt være begrænset af separatortræets højde.

Mere detaljeret foregår søgningen efter indsættelsespunktet som følger:

Lad s_v være den knude i separatortræet S_T, vi er kommet til. Vi ønsker nu at afgøre, hvor i træet T arten a skal indsættes, relativt til knude v. Indsættelsespunktet kan sidde i deltræet over v, i et undertræ rodfæstet i et af vs børn eller ved v selv. I det sidste tilfælde er søgningen afsluttet, og a indsættes som et blad på v.

Inden søgningen startes, skal vi have etableret en ordning på v's børn

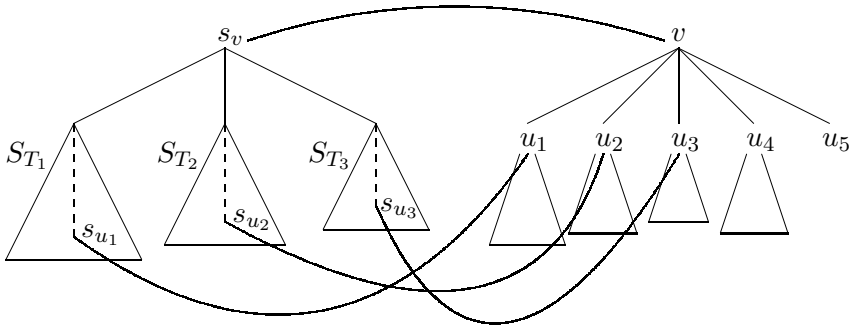
i det evolutionære træ T, således at vi undersøger det tungeste barn først. Lad $u_1, \ldots, u_k$, være naboer til v i T. Lad $u_1, \ldots, u_i$, $i \leq k$ være de naboer til v, hvor de tilhørende separatorknuder $s_{u_1}, \ldots, s_{u_i}$ er placeret i forskellige undertræer $S_{T_1}, \ldots, S_{T_i}$ under s_v i separatortræet S_T. De resterende naboer $u_{i+1}, \ldots, u_k$ er enten blade i T eller har separatorknuder, der er placeret over s_v i S_T. Ordningen på $u_1, \ldots, u_i$ i T kommer fra hvorledes undertræerne $S_{T_1}, \ldots, S_{T_i}$ til s_v er ordnet i S_T. Da børnene til s_v, kender deres plads i ordningen på s_v's børn, skal vi blot for hver nabo u_j til v finde roden på det separatorundertræ rodfæstet ved s_v, hvori s_{u_j} er placeret. Dette gøres ved at løbe fra s_{u_j} og opad i separatorundertræet indtil vi møder et barn af s_v, dvs. roden af separatorundertræet, og benytte dennes placering i ordningen af s_v's børn til at afgøre u_j's placering i ordningen af v's naboer. Hvis dette gøres for samtlige $u_1, \ldots, u_i$ har vi fået etableret den ønskede ordning; de resterende naboer $u_{i+1}, \ldots, u_k$ optræder i en arbitrær ordning.

Figur 3.21 viser hvorledes børnene af knuden v sorteres efter ordningen for børnene af s_v. v's børn er markeret u_1 til u_5 efter den rækkefølge de får, når de er blevet sorteret. I dette eksempel har tre af v's fem børn et tilsvarende separatorundertræ under s_v. Separatorknuden $s_{v_{forfader}}$, for v's forfader er i S_T barn af en separatorknude, der sidder højere oppe i separatortræet end s_v. Havde $s_{v_{forfader}}$ været placeret i et separatortræ rodfæstet ved s_v, så skulle dette separatorundertræ identificeres, således at rekursionen kan fortsætte ned i det, hvis eksperimenterne viser, at vi skal højere op i T. Det vigtige at bemærke her er, at vi i dette tilfælde, som altid, bevæger os nedad i separatortræet, mens søgningen i T bevæger sig opad og afskærer et eller flere dybere undertræer i processen.

For at kunne sortere knuderne i træet efter deres orden i separatortræet, skal vi først kunne sammenholde de to mængder af knuder. Per konstruktion kan separatorknuderne for knuden v's børn jo både ligge over og under s_v. Til dette har vi en rekursiv funktion, algoritme 1, der, ved at starte nede i separatortræet og søge opad, sikrer at vi højest bruger tid proportionalt med separatortræets højde. Ved at anvende dybderne på separatorknuderne undgås søgninger, der aldrig kunne give et resultat. Ydermere kan to-lags strukturen udnyttes til at skyde genvej, ved at hoppe direkte til roden af komponenten, hvis s_v (i algoritmen) ligger i en komponent, og s_p er i samplen.

Med afbildningen mellem børn og separatorbørn på plads er sorteringen blot et spørgsmål om at oprette afbildningen for alle v's børn, sortere dem efter s_v's børn (der er ordnede) og tilføje de resterende børn af v. I algoritme 2 gennemgås dette.

Efter at vi har fået lavet en ordning på børnene til v i T, begynder vi at udføre de eksperimenter, der skal finde indsættelsespunktet. Der foretages



Figur 3.21: Sortering af knuder ved hjælp af separatorknuder. Bemærk at sorteringen ikke tager hensyn til størrelsen på de eventuelle undertræer af u_4 og u_5, da deres separatorknuder ikke er i undertræer af s_v.

Algoritme 1: separatorRoot(s_v,s_p)

Input : To knuder, s_v og s_p, i separatortræet.
Output: En knude, s_u, der er barn af s_p, hvis s_v er i undertræet rodfæstet ved s_u.

```
if depth(s_v) ≤ depth(s_p) then
    return null;
else
    if parent(s_v) = s_p then
        return s_v;
    else
        separatorRoot(parent(s_v),s_p);
    end
end
```

højst $\lceil k/2 \rceil$ eksperimenter ved knuden v. Det i'te eksperiment bliver foretaget på arterne a, b og c, hvor a er arten, vi er ved at finde et indsættelsespunkt til, og arterne b og c er blade i T, der er placeret under henholdsvis u_{2i-1} og u_{2i},

Algoritme 2: sortBySeparatorOrder(v)

Input : En knude, v, i det evolutionære træ.
Output: C, et array af knuder, der er børn af v og SC, rødderne i de separatorundertræer under s_v, som børnene af v ligger i.
Data: En afbildning, M, fra separatorknuder under s_v, til børn af v i deres undertræer.

```
foreach u ∈ children(v) do
    s_temp ← separatorRoot(s_u,s_v);
    if s_temp ≠ null then
        M[s_temp] ← u;
    end
end
i ← 1;
foreach s_u ∈children(s_v) do
    C[i] ← M[s_u];
    SC[i] ← s_u;
    i ← i + 1;
end
foreach u ∈ children(v) do
    if u ∉ C then
        C[i] ← u;
        i ← i + 1;
    end
end
```

dvs. at b og c's nærmeste fælles forfader er v. Hvis k er ulige, så udføres der et eksperiment, hvor b og c er blade fra T, der er placeret under henholdsvis u_k og u_1. Bemærk, at per definition er disse blade forskellige, eftersom vi altid konstruerer knuder, så de har to børn, enten blade eller knuder, og alle knuder, der får fjernet et barn, får det erstattet med et nyt. For at vi hurtigt kan få fat i bladene b og c, givet at vi har u_{2i-1} og u_{2i}, har vi at hver intern knude i T har en pegepind til et arbitrært blad i dens undertræ. Når der indsættes en ny knude i T, så sættes pegepinden til at pege på det blad, der gav anledning til indsættelsen af den nye knude. Eksperiment i, mellem a, b og c, kan give følgende 4 forskellige udfald:

1. (a, b, c) angiver at indsættelsespunktet for a er tilstødende til enten en efterkommer til u_j, hvor hverken b eller c er efterkommere af u_j eller er et nyt blad under v.

2. $((a, b), c)$ angiver at indsættelsespunktet for a er tilstødende til en efterkommer af u_{2i-1}, da den nærmeste fælles forfader for a og b ligger under v i T.

3. $((a, c), b)$ angiver at indsættelsespunktet for a er tilstødende til en efterkommer af u_{2i}, da den nærmeste fælles forfader for a og c ligger under v i T. Hvis k er ulige, så er a tilstødende til en efterkommer af u_1.

4. $((b, c), a)$ angiver at indsættelsespunktet for a sidder i et undertræ over v, da den nærmeste fælles forfader mellem a og b (og mellem a og c) er over v. Hvis v er roden i træet, så skal der konstrueres en ny rod med v og a som børn.

Vi starter med $i = 1$ og udfører så eksperimenter, hvor vi øger i, op til$\lceil k/2 \rceil$. Hvis vi undervejs møder et i, der giver et andet udfald end tilfælde 1, fortæller det os, hvor vi skal fortsætte. Hvis alle eksperimenter gav topologien (a, b, c), bliver a indsat som et blad direkte under v i T. I de andre tilfælde ved vi nu, i hvilket undertræ forbundet til v, a skal indsættes. Hvis der ikke er et tilhørende undertræ i S_T, har vi fundet et tilfælde, hvor en tilstødende kant skal deles. Ellers har vi fundet et undertræ, som vi kan fortsætte søgningen i. For at kunne fortsætte søgningen identificerer vi det barn af s_v, der er rod i undertræets separatortræ. Denne separatorknude bliver så udgangspunkt for den næste omgang eksperimenter.

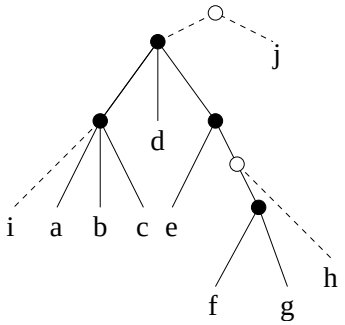
Når vi har fundet frem til, hvor arten a skal indsættes, kan indsættelsen give anledning til en af tre former for ændringer af det evolutionære træ, som illustreret på figur 3.22.

- Arten i bliver indsat som et blad på en eksisterende knude.

- Indsættelsen af arten j giver anledning til, at der bliver dannet en ny rodknude i træet, hvor den eksisterende rodknude bliver barn, samt at j bliver indsat som blad til den nye rodknude.

- Indsættelsen af arten h i træet giver anledning til at en eksisterende kant mellem to interne knuder i træet bliver delt, og der bliver indsat en ny knude, hvorpå h hænger som et blad.

I algoritme 3, *addSpecies*, indsætter vi en art i det evolutionære træ. Som det ses, håndteres de to første arter som særtilfælde, mens de resterende arter indsættes ved den rekursive procedure *recursivelyAddSpecies*, algoritme 4, der håndterer de resterende indsættelser.



Figur 3.22: De tre måder en nyt art, i, j eller h, kan indsættes på.

For bedre at kunne præsentere algoritmen, er indsættelser på en kant i det evolutionære træ beskrevet separat, som proceduren *splitEdge*, algoritme 5.

Vi benytter en operation, *performExperiment*, der repræsenterer udførelsen af et eksperiment på tre givne arter.

Generelt benytter vi en række operationer på det evolutionære træ og separatortræet. De inkluderer *removeLeaf*, *addLeaf*, *insertNode* og *insertNodeBetween*, der tillader os at fjerne og tilføje blade og indsætte nye interne knuder, enten under en eksisterende knude eller mellem to.

Algoritme 3: addSpecies(a)

Input : Et evolutionært træ, T, med separatortræ S_T og den art a, der skal indsættes.

Output: Træerne T og S_T, med a indsat det rette sted.

```
if size(T) = 0 then
    root(T) ← newNode ;
    addLeaf(root(T),a);
else if size(T) = 1 then
    addLeaf(root(T),a);
else
    recursivelyAddSpecies(root(S_T),a);
end
```

Algoritme 4: recursivelyAddSpecies(a,s_v)

Input : En art, a, der skal indsættes i det evolutionære træ, og en separatorknude, s_v, hvor søgningen starter.

Data: Array C af knuder, array SC af separatorknuder, knuden v, der hører til s_v og antallet af børn til v, k.

```
(C, SC) ← sortBySeparatorOrder(children(v));
for i ∈ [1, ⌈k/2⌉] do
    i_b ← 2i − 1;
    if 2i ≤ k) then
        i_c ← 2i;
    else
        i_c ← 1;
    end
    topologi t ← performExperiment(a,C[i_b],C[i_c]);
    if t = Type 2 then
        if SC[i_b] ≠ null then
            recursivelyAddSpecies(SC[i_b],a);
        else
            splitEdge(v,a,C[i_b]);
        end
    else if t = Type 3 then
        if SC[i_c] ≠ null then
            recursivelyAddSpecies(SC[i_c],a);
        else
            splitEdge(v,a,C[i_c]);
        end
    else if t = Type 4 then
        if separatorRoot(s_parent(v),s_v) ≠ null then
            recursivelyAddSpecies(separatorRoot(s_parent(v),s_v),a);
        else
            node n ← newNode();
            insertNode(v,n);
            addLeaf(n,a);
        end
    end
end
if allType1 then
    addLeaf(v,a);
end
```

Algoritme 5: splitEdge(v,a,u)

Input : En knude, v, i det evolutionære træ, et barn af v, u, og en art, a, der skal kobles på træet på kanten mellem v og u.

Data: En knude, n.

```
n ← newNode();
if leaf(u) then
    removeLeaf(v,u);
    insertNode(v,n);
    addLeaf(n,a);
    addLeaf(n,u);
else
    insertNodeBetween(v,n,u);
    addLeaf(n,a);
end
```

Af algoritmen følger, at når indsættelsespunktet er blevet fundet, så bliver der altid indsat et blad og højst en intern knude i T. Efter dette er gjort, opdateres separatortræet S_T ved hjælp af algoritmen fra teorem 2.

3.5.2 Realisering

Den største forskel mellem algoritmen og dens implementation er håndteringen af samplen. Da vi ikke har designet en fuldstændig abstraktion, der ville få samplen med tilhørende strukturer til at ligne et separatortræ, er det nødvendigt eksplicit at håndtere forskellen mellem separatorknuder for sampleknuder og for knuder i komponenter. Specielt gør det sig gældende i begyndelsen af vores rekursive metode, *insertLeaf*(), der udgør kernen i algoritmen. Denne metode er implementationen af *recursivelyAddSpecies*, algoritme 4.

Yderligere komplikationer opstår omkring etableringen af listerne over børn. Det primære problem skyldes, at pegepindene mellem knuder og separatorknuder går gennem sampleknuderne. Dette kunne naturligvis håndteres i de underliggende datastrukturer, men vi har placeret det her.

Det næste skridt er at forbinde de rigtige knuder med deres separatorknuder. Her opnår vi en lille gevinst ved eksplicit kendskab til samplen. Det betyder nemlig at søgninger op i separatortræet vil kunne afsluttes tidligt. Konkret gælder det, at en forældreknude i separatortræet for en komponent aldrig kan have et barn i separatortræet for samplen. En yderligere optimering opnås, når søgningen starter i en komponent, og forældreknuden er i

separatortræet for samplen. Så kan der hoppes direkte til roden af komponentens separatortræ.

Endeligt er der indsat en række kald til rutiner, der opsamler data om tidsforbrug og antallet af udførte eksperimenter, og en række tests for fejlsituationer.

Udover disse tilføjelser, i forhold til algoritmen, som præsenteret i det foregående afsnit, er opsplitningen i funktioner ikke helt som tidligere beskrevet. Specielt er opsplitning af kanter og sortering af knuder ikke splittet ud som separate metoder.

3.5.3 Kompleksitet

Der er to interessante mål for kompleksiteten af algoritmen: Udførelsestiden, som for enhver anden algoritme, og antallet af udførte eksperimenter. Antallet af eksperimenter er interessant, eftersom denne metode antager dem som fundamentale operationer, og [KLW90] fortæller os, at deres praktiske omkostning kan være høj.

Dette er indeholdt i lemma 6 og teorem 3 fra [BFPÖ01].

Lemma 6 *Givet et evolutionært træ T med n arter og en grad d, og et separatortræ S_T for T i henhold til teorem 2, så kan en ny art indsættes i T og S_T i amortiseret tid $O(d \cdot \log_d n)$, ved anvendelse af højest $\lceil d/2 \rceil (\log_{2\lceil d/2 \rceil - 1} n + O(1))$ eksperimenter for $d > 2$ og højest $\log n + O(1)$ eksperimenter for $d = 2$.*

Lemma 6 fortæller om udførelsestiden, samt hvor mange eksperimenter, der skal bruges for at indsætte én ny art i det evolutionære træ samt i det tilhørende separatortræ. Dette resultat sammen med teorem 2 giver os følgende grænser for tidsforbruget samt antallet af eksperimenter for at konstruere og vedligeholde et evolutionært træ under indsættelser af nye arter vha. eksperiment-modellen. Beviset for lemma 6 findes i [BFPÖ01] og vil ikke blive uddybet her.

Teorem 3 *Efter $O(n)$ preprocesseringstid kan et evolutionært træ for n arter vedligeholdes gennem m indsættelser i tid $O(dm \log_d(n+m))$ ved anvendelse af højest $m\lceil d/2 \rceil (\log_{2\lceil d/2 \rceil - 1}(n+m) + O(1))$ eksperimenter for $d > 2$ og højest $m(\log(n+m) + O(1))$ eksperimenter for $d = 2$, hvor d er den højeste udgrad under indsættelserne.*

Teoremerne 1 og 2 giver os direkte preprocesseringen, da denne blot består i at konstruere og sortere samplen for det givne træ.

For at vise udførelsestiden tager vi resultatet fra lemma 6 og summerer for alle m indsættelser:

$$\sum_{i=0}^{m-1} O(d \log_d(n+i)) \leq O(dm \log_d(n+m))$$

Da vi direkte laver træet for de første to arter, hvorefter vi tilføjer de øvrige arter en for en til T, følger det heraf, at konstruktion af et træ med n blade er $O(dn \log_d n)$.

Tilsvarende får vi:

$$\sum_{i=0}^{m-1} \lceil d/2 \rceil (\log_{2\lceil d/2 \rceil - 1}(n+i) + O(1)) \leq m \lceil d/2 \rceil (\log_{2\lceil d/2 \rceil - 1}(n+m) + O(1))$$

Igen følger det at konstruktionen kræver højest $n\lceil d/2 \rceil (\log_{2\lceil d/2 \rceil - 1} n + O(1))$ eksperimenter.

Hermed er artiklens primære resultat vist.

3.6 Andre datastrukturer

Udover de ovennævnte centrale datastrukturer og algoritmer så har vi som nævnt implementeret nogle simplere datastrukturer, hvor vi ikke har kunnet fremskaffe en brugbar implementation.

3.6.1 Søgetræ

I den første algoritme, fra lemma 1, til konstruktion af separatortræer repræsenterer vi tunge stier ved referencer mellem de knuder, der udgør dem. Denne repræsentation er ækvivalent med en hægtet liste, med det deraf følgende tidsforbrug.

For at kunne reducere udførelsestiden for konstruktionen fra $O(n \log n)$ til $O(n)$, benytter vi søgetræer til at repræsentere tunge stier, således at vi effektivt kan finde separatorknuderne.

I det følgende vil vi kort gennemgå de nødvendige faciliteter i søgetræerne, og hvordan disse opnås.

Datastruktur

Søgetræerne, vi benytter, er baseret på splaytræer, beskrevet i [Tar83] kapitel 4, [ST85] og [Sky00]. Det er en type selvbalancerende træer, der garanterer $O(\log n)$ højde. Ovenpå splaytræet har vi konstrueret søgetræ-datastrukturen med følgende funktioner: *key*, *join*, *split*, *successor* og *addpathcost*, der alle skal køre i amortiseret $O(\log n)$ tid.

Herefter følger en mere detaljeret beskrivelse af, hvad funktionerne skal kunne. Alle funktioner har et underforstået argument – det søgetræ de kaldes på.

- *key*(element) - Givet et element returneres dennes nøgle.
- *join*(søgetræ) - Samler to søgetræer givet at nøglerne i det ene søgetræ er mindre end den mindste nøgle i det andet søgetræ.
- *split*(element) - Deler et søgetræ ved elementet og returnerer to nye søgetræer.
- *successor*(nøgle) - Returnerer det element med den mindste nøgle, der er større end eller lig med den givne nøgle.
- *addpathcost*(cost) - Adderer den samme værdi til alle nøgler i søgetræet,

Nøglerne lagres i splayknuderne som differencer, således at $key(n) = \underline{\Delta \text{key}}(n) + key(\underline{\text{parent}}(n))$, hvis knuden n ikke sidder i roden af træet. Dette tillader, som vi vender tilbage til om lidt, at nøglerne kan opdateres i hele træet ved kun at ændre i roden.

Realisering

Implementationen kan betragtes i to dele. Først er der et splaytræ, der er i stand til at balancere sig selv ved hjælp af metoden *splay*(). Den udfører fra en given knude de i [Tar83] beskrevne rotationer, således at den valgte knude gøres til rod i træet, uden at det ændrer på den ordning, træet repræsenterer. Herved balanceres træet samtidigt.

Under rotationerne skal de nævnte $\underline{\Delta \text{key}}$-variable opdateres i de berørte knuder, men dette kan udføres lokalt, så med hensyn til udførselstiden ændrer det ikke noget.

For at kunne implementere de følgende funktioner effektivt er elementerne, som søgetræet kan indeholde, begrænset til objekter, der implementerer et interface, **Searchable**. Dette interface sikrer, at elementerne kan indeholde en pegepind til splayknuderne, der ligeledes har en pegepind til deres

element. Yderligere er der krævet en metode til at få den nøgle, som elementet skal indsættes under. Dette er nødvendigt, da søgetræet bygges fra en liste af elementer.

Oven på splaytræet er selve søgetræet så bygget. De enkelte metoder er implementeret som følger:

- *key*(element) - Ved at placere en reference til den tilhørende knude i elementet er denne metode et simpelt opslag.

- *join*(søgetræ) - Her lokaliseres den mindste knude i det *store* træ (træet med de største knudenøgler). På denne knude kaldes *splay*(), således at den er rod i sit træ. Da den er den mindste knude i sit træ følger det, at den ikke har noget venstre barn. Og da alle knuderne i det andet træ har mindre nøgle, så kan det indsættes som venstre barn. Så mangler der blot at justere nøglerne i det indsatte træ, men det kan gøres lokalt i dets rod, ved at justere $\underline{\Delta \text{key}}$, således at den afspejler, at roden nu er barn af den minimale knude i det *store* træ.

- *split*(element) - Her benyttes igen relationen mellem elementer og knuder til at finde den relevante knude. På denne udføres så et *splay*() og det højre undertræ kan nu pilles ud og returneres som det store søgetræ, mens resterne af det oprindelige træ udgør det lille træ. En alternativ model, som vi har implementeret af praktiske årsager, er at returnere de to undertræer og smide det givne element væk. Dette benytter vi os af for at simplificere koden, hvor vi ellers skulle splitte to gange.

- *successor*(nøgle) - For at finde den mindste knude med nøgle størrelig den givne, søger vi rekursivt fra roden. Ved at bære forælderens nøgle med ned som et argument til de rekursive kald, kan vi udregne de besøgte knuders nøgler fra deres $\underline{\Delta \text{key}}$, uden at skulle kalde tilbage op i træet. Der resterer nu blot, at sammenligne den givne nøgle med den beregnede nøgle for den aktuelle knude og fortsætte rekursionen til højre eller venstre, indtil resultatet er fundet.

- *addpathcost*(cost) - Da nøglerne er gemt som forskelle, er det simpelt at opdatere dem ved blot at justere rodens $\underline{\Delta \text{key}}$.

Herudover har vi implementeret metoder til at finde det største og mindste element, størrelsen af et træ (ved at gennemløbe det, da vi kun benytter det til testning), iteration over elementerne, og en konstruktionsmetode med lineær udførelsestid. Givet en **Collection** så konverteres dette til et array og

deles ved det midterste element og der fortsættes rekursivt på de to mindre arrays. I bunden af rekursionen kan der så returneres blade med et element, der så længere oppe kombineres med interne knuder. Således konstrueres træet så det fra starten er balanceret.

Udførelsestider

Som det fremgår af ovenstående har *key* og *addpathcost* konstant udførelsestid. På grund af balanceringen kan *join*, *split*, og *successor* udføres i $O(\log n)$ tid, amortiseret, som ønsket. Og som nævnt er konstruktion af et balanceret træ fra sorterede elementer simpelt i lineær tid.

Diskussion

Et væsentligt problem i [BFPÖ01]'s reference til søgetræer er, at der specifikt henvises til [Tar83], kapitel 5. Der beskrives ganske vist en type træer med blandt andet *link*-, *cut*-, og *addcost*-operationer, der ved første øjekast ligner de ønskede operationer *join*, *split*, og *addpathcost*. Imidlertid er de beskrevne træer ikke søgetræer, men derimod "rigtige" træer. Men det viste sig at være kapitel 4 i [Tar83], der beskrev hvorledes man kan implementere de ønskede metoder ovenpå et splaytræ. Vi har dog brugt princippet fra hjælpedatastrukturen **Path**, der bliver beskrevet i kapitel 5, om at gemme nøglerne som forskelle i stedet for værdier i knuderne.

Udover ovennævnte upræcise reference er det ikke specielt problematisk at implementere søgetræer ved splaytræer. Specielt gør det implementationen simplere, at der ikke er behov for andre opdatering af datastrukturen end *join* og *split*.

3.6.2 Prioritetskø

Med de tunge børn placeret i søgetræer skal vi, som nævnt, også kunne tilgå de øvrige børn af knuderne i det evolutionære træ ordnet efter deres undertræers størrelse. For at gøre det effektivt behøvede vi en prioritetskø.

Datastruktur

Prioritetskøen skal understøtte følgende funktioner, der alle skal have højest logaritmisk udførelsestid.

- *insertItem*(prioritet, element) - Indsættelse af et element med en arbitrær prioritet

- *removeElement*() - Fjerner og returnerer det element med højest prioritet

Desuden skal køen kunne konstrueres i lineær tid, givet en liste med elementer, der, i modsætning til konstruktionen af søgetræet i sektion 3.6.1, ikke antages at være sorteret.

Realisering

Vi forsøgte at finde en allerede implementeret prioritetskø på nettet, der understøttede de krav, vi havde, såsom en lineær konstruktionstid. Dette viste sig at være sværere end forventet. Vi kunne simpelthen ikke finde en, hvis struktur var generel nok til at vi kunne benytte den.

Den implementerede prioritetskø er en heapbaseret prioritetskø, som beskrevet i [GT98]. Denne bygger på et linkbaseret binært træ og har de ønskede egenskaber.

3.6.3 Hægtet liste

Til opbevaring af de sorterede separatorknuder behøver vi en hægtet liste. Igen var der ikke en færdig implementation, med et egnet interface, til rådighed, hvorfor vi har implementeret denne simple datastruktur.

Specielt havde vi behov for at listen effektivt kunne håndtere den ofte forekommende situation, hvor et element får sin nøgle forøget med 1, hvorved den skifter plads i listen.

Datastruktur

Listen er opbygget med dobbelt-kædede listeknuder, der hver indeholder en ikke-tom mængde af elementer og en nøgle.

Realisering

For at sikre passende afkobling mellem listen og elementerne, har vi indført et ekstra lag af indirektion, i form af **NodeObject**-klassen, som er indeholdt i listeknudernes mængde af elementer. Disse peger så videre på de faktiske elementer, separatorknuderne, der også peger tilbage på deres **NodeObject**. Herved kan vi let komme fra et element til dets plads i den kædede liste, uden at bekymre os unødigt om opdateringerne i listen eller i elementerne.

Håndtering af nøgle-forøgningen er også lavet ved hjælp af **NodeObject** afkoblingen. Elementet flyttes simpelthen til den næste listeknude i rækken,

efter at vi har sikret os, at den har den ønskede nøgle (listeknuden med den rette nøgle oprettes og indsættes om nødvendigt).

3.6.4 Øvrige

Udover de ovennævnte datastrukturer benyttes en række mindre komplicerede datastrukturer. Hvorledes disse er blevet realiseret vil ikke blive gennemgået.

- Binært træ - Vi benytter et binært træ til at lave den heapbaserede prioritetskø.
- Newick Træer - Selve programmet returnerer et evolutionært træ i NEWICK-format og derfor har vi lavet en datastruktur til håndtering af dette. Yderligere er der bygget en simpel parser til formatet, som er beskrevet i bilag B.1 og praktiske hjælpefaciliteter såsom beregning af maksimal udgrad og håndtering af navnesammenfald.[3]
- Distance Matrix - Inputtet til programmet er en matrice. Vi har lavet en wrapperstruktur til en sådan matrice for at lave et pænt interface til tilgangen af matricen. Matricerne kan efter behov konstrueres udfra filer i PHYLIP-format, se bilag B.2, eller fra allerede indlæste Newick-træer.

3.7 Testning

Vi har ønsket at udvikle et korrekt, robust og stabilt stykke software. For at opnå disse tre ting har vi underlagt programmet en stor mængde aftestning. Vi har valgt at basere vores aftestning på Unit-testing strategien, og da det hele er skrevet i Java benytter vi JUnit, et unit-test framework til Java[4]. Da JUnit er de-facto standarden på området, er der glimrende understøttelse i vores valgte udviklingsværktøj, Eclipse, via plugins, der præsenterer resultaterne af testkørsler grafisk og gør det nemt at navigere mellem tests og deres resultater.

Vi har helt fra starten lavet testcases til de klasser og metoder, vi har skrevet. Dog har vi ikke benyttet strategien med at skrive testkoden, før vi skrev selve koden.

[3] Da noget af vores data indeholdt trunkerede navne, havde vi behov for at gøre dem unikke i et givent træ.

[4] For mere information om JUnit henviser vi til `http://junit.org/`, den officielle hjemmeside.

Da vi har relativt få basale klasser, hvorpå vi bygger nogle avancerede klasser, opnår vi, i praksis, en form for integrationstest ved at unit-teste disse avancerede klasser.

Mere præcist har vi struktureret vores testcases således, at for hver package vi har i koden, så er der en tilhørende testpackage. I hver af disse testpackages er der en række testklasser, der er designet til at teste den tilhørende klasse og dens metoder. Testklasserne er opbygget med et par af metoder, *setUp*() og *tearDown*(), der bliver kaldt før hver testcase (metoder navngivet *testCase1*(), *testCase2*(),... bliver automatisk kørt af JUnit). Hermed er det let at sætte data op, således at testcases kan køres både individuelt og sammen.

Figur 3.24 viser som eksempel den ottende testcase til prioritetskø-koden. Den benytter en skeletklasse for input, **TestKeyable**, der er inkluderet som figur 3.23.

Det første interessante sker på linie 10, hvor den nævnte *setUp*()-metode defineres. Her sætter vi vores testdata op, så vi har en sorteret liste af **TestKeyable**-objekter. *tearDown*()-metoden er ikke nødvendig, da der ikke er noget, vi behøver at rydde op.

Selve testmetoden, der starter på linie 19, itererer fra 0 til størrelsen af vores liste af testdata. For hver størrelse i det interval bygger vi så, på linie 21-24, et **HashSet** af det ønskede antal elementer fra listen af testdata. Dette **HashSet** sikrer, at elementerne ikke er i en specifik orden. Så konstruerer vi køen, hvilket ikke kan gå galt i dette tilfælde (andre testcases tester blandt andet at den definerede **Exception** kan kastes). Dernæst udtrækkes alle elementerne i køen (linie 31), og hvert element testes (linie 32) mod det korrekte element i listen med testdata. Endelig testes at køen er tom (linie 37).

```
private class TestKeyable implements Keyable {
 int key;
 String data;
 public TestKeyable(String data, int key) {
  this.key = key;
  this.data = data;
 }

 public int getKey() {
  return key;
 }

 public String getData() {
  return data;
 }
}
```

Figur 3.23: TestKeyable.

```
public class PriorityQueueTests extends TestCase {
 PriorityQueue pq;
 HashSet hs;
 Vector v;

 public PriorityQueueTests(String arg0) {
  super(arg0);
 }

 protected void setUp() throws Exception {
  super.setUp();
  this.v = new Vector();
  int end = 100;
  for (int i = 0; i < end; i++) {
   this.v.add(new TestKeyable("k" + i, i));
  }
 }

 public void testCase8() {
  for(int bound=0; bound<this.v.size(); bound++) {
   this.hs = new HashSet();
   for (int i = 0; i < bound; i++) {
    this.hs.add(this.v.get(i));
   }
   try {
    this.pq = new HeapPriorityQueue(hs);
   } catch (NotKeyableException nke) { }
   assertEquals(bound, this.pq.size());
   for (int i = 0; i < bound; i++) {
    try {
     TestKeyable temp = (TestKeyable) this.pq.removeMinElement();
     assertEquals(this.v.get(i), temp);
    } catch (Exception e) {
     assertTrue(false);
    }
   }
   assertEquals(0, this.pq.size());
  }
 }
}
```


Figur 3.24: *testCase8*() fra PriorityQueueTests.java.

I de tilfælde, hvor vi tester fx konstruktion af separatortræer eller samplen, benytter vi nogle træer, vi har lavet i et simpelt XML-format. Vi har så lavet en testmetode for hvert valg af rod. Dette har vi gjort, da der jo i algoritmen bliver valgt en arbitrær knude som rod. Dette er ikke brugbart i en testsituation, da vi ønsker at garantere 100% reproducerbarhed. Denne strategi giver anledning til rigtig mange linjer testkode og derfor har vi også en ret omfattende testsuite. Selve suiten indeholder omkring 2200 testcases. Dette antal kunne reduceres væsentligt ved at lade testcases iterere over mulighederne. Dette ville imidlertid gøre det vanskeligere at undersøge præcist det fejlende tilfælde, hvorfor vi fremstillede de mange testcases. En stor del af de simple tilfælde er genereret vha. scripts, for at sikre at ingen tilfælde blev glemt – og for at spare arbejde.

En anden nødvendig ændring, der sikrer, at vi altid kan reproducere de fejl, som opstod under udviklingen, består i at overskrive Javas indbyggede metode, *hashCode*(), der benyttes når objekter placeres i Javas indbyggede, hash-baserede, datastrukturer. Ved at erstatte denne metode med en, der blot udleverede unikke numre, startende ved nul, når hver test startes.[5] Herved sikres det, at den rækkefølge, objekter returneres i fra de hash-baserede datastrukturer, er konsistent og uafhængig af eventuelle foregående testcases. I modsat fald kan man risikere, at en fejl, der er afhængig af rækkefølgen på input, kan detekteres ved kørsel af hele testsuiten men ikke ved kørsel af den individuelle testcase alene. Da dette oplagt er utilfredsstillende, lavede vi den beskrevne ændring.

Idéen med at have ca. 2200 testcases er jo ikke god, hvis man ikke undersøger om disse blot tester det samme, derfor har vi undersøgt dækningsgraden af vores testcases. For at gøre dette, har vi benyttet et plugin til Eclipse kaldet dJUnit[6] til at lave disse undersøgelser med. Dette plugin, kan udover at fortælle os dækningsgraden, også vise hvilke metoder der kaldes oftest. Denne analyse har vi brugt til at optimere på de mest brugte metoder. dJUnit fungerer ved at indlæse JUnit testcases og dekorere den indlæste kode med passende instruktioner til at registrere om og hvor ofte hver eneste del af koden udføres. Dette gør naturligvis, at testkørslerne tager meget længere tid, end når der blot afvikles umodificeret kode. I vores tilfælde tog et par minutters test under JUnit over en time at afvikle under dJUnit.

Denne analyse gav anledning til, at vi ændrede i nogle af vores testcases

[5] Dette er i overensstemmelse med specifikationen af *hashCode* i java, `http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Object.html\#hashCode()`, da vi ikke har ikke-trivielle *equals* på objekter, der benytter dette trick.

[6] Tilgængeligt fra `http://works.dgic.co.jp/djunit/`. En del af dokumentationen er ikke oversat til engelsk, men til vores formål gav det ikke problemer.

for at ramme flere tilfælde. Med disse rettelser har vi en dækningsgrad på ca. 83%. Hvilket jo ved første øjekast ikke virker imponerende, da det betyder at 17% af vores kode ikke bliver testet. Men de 17% består hovedsageligt af korrekthedstest (sanity checks), håndtering af I/O-metoder og andre hjælpe- og debug-metoder, fx *toString()*. Så vi mener, at en dækningsgrad på 83%, med hensyntagen til hvad de sidste 17% dækker over, er et rigtigt godt resultat og at det underbygger stabiliteten og robustheden af vores program.

Et af de steder, hvor dJUnit hjalp os utroligt meget, udover at hjælpe os med at sikre tilstrækkelig testning af alle ender af koden, var da vi opdagede at vores implementation kørte for langsomt. Det viste sig, at nogle af de mest kaldte metoder var *toString*() på en række objekter. Da disse metoder ikke burde blive kaldt, undersøgte vi det, og fandt at vi ikke havde fået fjernet tilstrækkeligt meget af vores debugkode. Da nogle af disse metoder involverede gennemløb af datastrukturerne, forårsagede de den observerede forøgelse af udførelsestiden.

3.7.1 Test af datastrukturer

En vigtig forudsætning for at kunne gennemføre automatiske teste er en metode til automatisk validering af resultaterne af disse tests. I mange tilfælde er det tilstrækkeligt at teste returværdien af de kald, testkoden foretager, men når man som i vores tilfælde benytter flere lag af datastrukturer, med adskillige indbyrdes relationer, så kan det være svært at se årsagen til en fejl ud fra et forkert output. For at løse dette problem har vi implementeret et par klasser, der har som deres eneste formål at validere datastrukturerne.

Validering af separatortræer

Klassen **SeparatorChecker** er en af disse valideringsklasser. Den tager et separatortræ og udfører en række simple tests, såsom at størrelsen er som forventet. Dernæst benytter den træets metoder til at gennemføre et gennemløb. Under dette gennemløb bliver der for hver knude testet en række af de egenskaber, der skal gælde for et separatortræ. Specielt testes det, at 1/2-separatoregenskaben (definition 2) er overholdt ved at beregne størrelsen på den aktuelle knude og sammenligne med størrelserne på dens børn. Tilsvarende testes det, at børnene kommer i den rette rækkefølge, såfremt der er tale om et sorteret træ.

Validering af sample

Klassen **SampleChecker** er opbygget efter samme ide som **SeparatorChecker**. Under gennemløbet checkes det, at relationerne mellem samplen og det underliggende træ er korrekte. For at kunne validere alle de egenskaber, som vi under udviklingsprocessen enten har haft problemer med at opretholde eller har mistænkt for at blive overtrådt, opsamler den et par lister af objekter under gennemløbet af datastrukturen. En af de opsamlede lister indeholder alle de knuder, der udgør samplen. Den benyttes i endnu et gennemløb, hvor alle de komponenter, der kan nås fra knuder i samplen, tælles. Herved valideres direkte, at samplen deler det underliggende træ i komponenter af størrelse maksimalt Δ. Den samme liste benyttes også til den næste test, hvor det valideres, at alle naborelationer i samplen er i overensstemmelse med de inducerede kantkomponenter eller svarer til knuder, der er naboer i det underliggende træ[7], og at relationerne er symmetriske.

Den anden liste indeholder alle komponenterne, og gennemløbes for at validere at deres egenskaber er korrekte. For at fange referencer, der burde have været slettet, testes alle komponenterne for et flag, der sættes når de bliver delt. Via et gennemløb af komponenternes underliggende træknuder validerer vi, at antallet af naboer i samplen er præcist 1 for bladkomponenter, eller 2 for kantkomponenter. Det valideres også, at kantkomponenter er tilknyttet den dybeste af deres naboer i samplen.

Endelig testes det at både samplen og de tilhørende komponenter har gyldige separatortræer ved hjælp af SeparatorChecker.

Anvendelse i testning

Checker-klasserne anvendes i vores testcases således, at den testede datastruktur valideres så ofte som muligt. I praksis betyder det, at de kaldes hver gang der er lavet en operation på datastrukturen. Specielt har vi testcases, der indsætter så mange knuder i et lille træ at dets størrelse fordobles flere gange. Derved sikres det, at de indbyggede rekonstruktioner forekommer. Ved at variere denne fremgangsmåde sikrer vi os også, at alle opdateringsmetoderne benyttes, og at alle deres betingede dele afvikles. Specielt den sidste egenskab er besværlig at opnå uden de førnævnte værktøjer til analyse af dækningsgraden.

[7] I dette tilfælde er den tomme kantkomponent ikke repræsenteret.

Diskussion

Det største problem ved denne type test er naturligvis at finde den mængde af egenskaber, det giver mening at teste for, og specielt at sikre, at alle interessante egenskaber bliver undersøgt.

Med disse klasser har vi fanget en lang række logiske fejl i datastrukturerne. Fejl, der gav en række af symptomer fra små fejl i resultaterne til forskellige exceptions på visse input. Da fejlene ofte er opstået et helt andet sted end deres symptomer, har det været en stor hjælp at kunne fange de tilfælde, hvor datastrukturen ikke har været korrekt.

Kapitel 4

Anvendelse

For at muliggøre anvendelse af vores implementation har vi inkluderet en række eksempelprogrammer, der pakker datastrukturerne ind i et anvendeligt interface. Programmerne er pakket i et Java-arkiv, en JAR-fil.

I slutningen af kapitlet vil vi gennemgå, hvorledes implementationen kan udvides og forbindes til anden software.

4.1 Trækonstruktion

Den væsentligste anvendelse er naturligvis afvikling af trækonstruktionen.

Implementationen indeholder tre eksempelprogrammer, der kan drive algoritmen. Det første program tager en afstandsmatrice i PHYLIP-format og producerer et træ i NEWICK-format, som beskrevet i bilag B. Da vores matricer er konstrueret fra kendte træer, har vi adgang til træerne, der er tilladt (men ikke påkrævet) som andet argument til programmet. Hvis der specificeres et træ, bliver dette benyttet til at tjekke om det rekonstruerede træ er korrekt. Herved kan vi direkte konstatere, at algoritmen giver det rette resultat.

I de nedenstående eksempler er de rekonstruerede træer ombrudt for at kunne være på siden, i praksis udgør det en linie i outputtet, af hensyn til videre bearbejdelse af de producerede træer. Den første linie af outputtet viser resultaterne af vores tidsmåling samt optællingen af antal brugte eksperimenter.

Det følgende eksempel viser en rekonstruktion af et træ ud fra matricen `foo.phylip`.

Eksperiment med matrix

```
dossen@pc # java -jar treeconstruction.jar \
            speciale.treeconstruction.MatrixExperimenter \
            foo.phylip
Total: 1102 ms (Algorithm: 323 ms, Experiments: #707, 207 ms)
Constructed tree:
((t10,t95),((((((t61,t63),t41),(t87,t3),t42,t20,t89,t68,t28,
t29),t1,t76,t49,t4),(((t52,t91),(t72,t98),t96,t66,t50),(((t47,
t92),t30),t80,t34),t51),t25,t53),(((((t48,t83),t6),t39),((t55,
t27,t43,t12,t86),t23,t17,t24,t73),t57),((t84,t14),(t70,t26,
t33),((t99,t32),t65,t5,t9,t54),t11,t78,t16,t15,t85),((t75,t18,
t88),t67,t100),(t8,t13,t19,t22),(((t58,t81),t71,t46),t37),
((t59,t35,t60,t94),(t56,t40),(((t97,t77),t38),((t31,t36),t93,
t64,t74),((((t90,t45),t21,t69),t7),t2)),t82),t44)),t79,t62);
```

Hvis man, som vi i vores test-situation, har adgang til det sande træ, så er det som sagt muligt at få valideret det konstruerede træ. Det kendte træ, som der valideres med i det følgende eksempel, er `foo.newick`. Vi benytter herfra og frem forkortelser for eksperimentklasserne. **MatrixExperimenter** kan kaldes med navnet "PHYLIP".

Eksperiment med matrix og træ

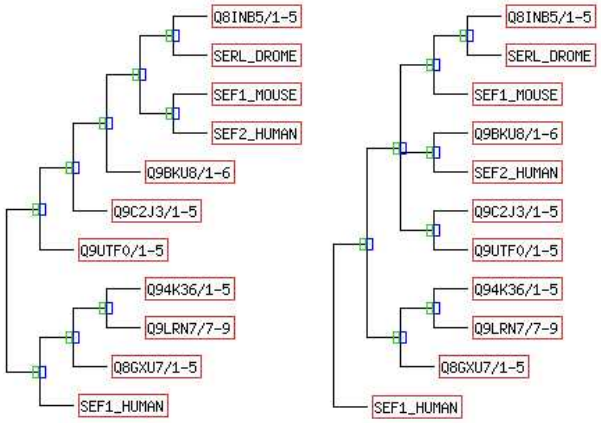
```
dossen@pc # java -jar treeconstruction.jar PHYLIP foo.phylip \
            foo.newick
Total: 1143 ms (Algorithm: 358 ms, Experiments: #707, 194 ms)
Constructed tree:
((t10,t95),((((((t61,t63),t41),(t87,t3),t42,t20,t89,t68,t28,
t29),t1,t76,t49,t4),(((t52,t91),(t72,t98),t96,t66,t50),(((t47,
t92),t30),t80,t34),t51),t25,t53),(((((t48,t83),t6),t39),((t55,
t27,t43,t12,t86),t23,t17,t24,t73),t57),((t84,t14),(t70,t26,
t33),((t99,t32),t65,t5,t9,t54),t11,t78,t16,t15,t85),((t75,t18,
t88),t67,t100),(t8,t13,t19,t22),(((t58,t81),t71,t46),t37),
((t59,t35,t60,t94),(t56,t40),(((t97,t77),t38),((t31,t36),t93,
t64,t74),((((t90,t45),t21,t69),t7),t2)),t82),t44)),t79,t62);
```

I tilfælde af data, der ikke er støjende ultrametriske (jfr. definition 1), kan det naturligvis ikke garanteres at det sande underliggende træ konstrueres. Med et andet træ som andet argument sammenlignes dette med resultatet, og uoverensstemmelser rapporteres som fejl, som det ses i det næste eksempel.

Eksperiment med matrix og træ der ikke matcher

```
dossen@pc # java -jar treeconstruction.jar PHYLIP 4F5.phylip \
            4F5.newick
Total: 284 ms (Algorithm: 202 ms, Experiments: #21, 7 ms)
Constructed tree:
((((Q9LRN7/7-9,Q94K36/1-5),Q8GXU7/1-5),(((Q8INB5/1-5,
SERL_DROME),SEF1_MOUSE),(Q9BKU8/1-6,SEF2_HUMAN),(Q9UTF0/1-5,
Q9C2J3/1-5))),SEF1_HUMAN);
Trees do not agree!!!
  ((((((Q8INB5/1-5,SERL_DROME),(SEF1_MOUSE,SEF2_HUMAN)),
  Q9BKU8/1-6),Q9C2J3/1-5),Q9UTF0/1-5),(((Q94K36/1-5,
  Q9LRN7/7-9),Q8GXU7/1-5),SEF1_HUMAN));
  (((((Q8INB5/1-5,SERL_DROME),SEF1_MOUSE),(Q9BKU8/1-6,
  SEF2_HUMAN),(Q9C2J3/1-5,Q9UTF0/1-5)),((Q94K36/1-5,
  Q9LRN7/7-9),Q8GXU7/1-5)),SEF1_HUMAN);
```

Som det ses, vises der ved uoverensstemmelser begge de involverede træer, i et format der kan sammenlignes som strenge.[1] Figur 4.1 viser de to træer[2].



Figur 4.1: Det højre træ er rekonstrueret ved vores algoritme, mens det venstre er lavet ved neighbour-joining.

[1] Alle knuder er sorteret alfabetisk. Interne knuder er, for så vidt angår sortering, navngivet ved deres deltræ.

[2] Tegnet vha. `http://www.proweb.org/treeviewer/`.

Til at afvikle vores eksperimenter direkte fra kendte træer har vi benyttet den anden implementation af eksperimentudførelsen, **NewickExperimenter**, der kan kaldes med navnet "NEWICK".

Eksperiment med træ

```
dossen@pc # java -jar treeconstruction.jar NEWICK 4F5.newick
Total: 238 ms (Algorithm: 189 ms, Experiments: #21, 7 ms)
Constructed tree:
((((Q94K36/1-5,Q9LRN7/7-9),Q8GXU7/1-5),SEF1_HUMAN),
(((((SERL_DROME,Q8INB5/1-5),(SEF1_MOUSE,SEF2_HUMAN)),
Q9BKU8/1-6),Q9C2J3/1-5),Q9UTF0/1-5));
```

Vores tredje eksperimentklasse er en interaktiv udgave til demonstrationsformål. Den er implementeret i **CommandLineExperimenter**, med det korte navn "CMDLINE".

De tre nævnte brugergrænseflader til koden er ikke udstyret med så mange features, som man kunne ønske sig i et værktøj til slutbrugere. Givet det overkommelige interface burde det dog ikke være vanskeligt at binde en bedre brugergrænseflade på implementationen, eller pakke den ind i et passende modul, så den kan benyttes fra en eksisterende programpakke.

Interaktivt eksperiment

```
dossen@pc # java -jar treeconstruction.jar CMDLINE
Add new species:
> a
Current tree:
(a);
Add new species:
> b
Current tree:
(a,b);
Add new species:
> c
Enter topologi for
 c
 a
 b
> (a,b,c)
Current tree:
(a,c,b);
Add new species:
> d
Enter topologi for
 d
 a
 c
> (d,(a,c))
Current tree:
((a,c,b),d);
Add new species:
> quit
Exiting program
Final tree:
((a,c,b),d);
```

4.2 Andre funktioner

Da vi ønskede at køre vores implementation på afstandsmatricer, hvor træerne var kendte, genererede vi dem direkte fra NEWICK-træer, som vist i

det næste eksempel:

Generering af Matrice

```
dossen@pc # java -classpath treeconstruction.jar \
            speciale.distancematrix.DistanceMatrixFromNewick \
            4F5.newick
        11
Q8GXU7/1-5   0.00000   1.09087   0.17801   0.92684   0.79063...
Q8INB5/1-5   1.09087   0.00000   1.10873   0.43644   0.60891...
Q94K36/1-5   0.17801   1.10873   0.00000   0.94470   0.80849...
Q9BKU8/1-6   0.92684   0.43644   0.94470   0.00000   0.44488...
Q9C2J3/1-5   0.79063   0.60891   0.80849   0.44488   0.00000...
Q9LRN7/7-9   0.17801   1.10873   0.03571   0.94470   0.80849...
Q9UTF0/1-5   0.63630   0.68580   0.65415   0.52177   0.38557...
SEF1_HUMAN   0.68272   1.31146   0.70057   1.14743   1.01123...
SEF1_MOUSE   1.22617   0.46336   1.24403   0.57174   0.74421...
SEF2_HUMAN   1.22617   0.46336   1.24403   0.57174   0.74421...
SERL_DROME   1.09087   0.03448   1.10873   0.43644   0.60891...
```

Her har vi igen beskåret outputtet, det fulde output indeholder naturligvis alle de tolv kolonner i matricen (labels plus elleve afstandskolonner).

4.3 Udvidelse

Hvis implementationen ønskes anvendt med en anden metode til udførelse af eksperimenter, så er det første trin at skrive en ny implementation af den abstrakte klasse **Experimenter**. I den konkrete klasse skal metoden *performExperiment*() implementeres. Den tager som argument tre arter og returnerer en af de fire mulige topologier, som defineret i **Experimenter**. Det første argument til *performExperiment*(), a, er per konvention den art, der ønskes indsat. Figur 4.2 viser interfacet.

Metoden *constructTree*() skal der også skrives en implementation af. Implementationen skal tilføje arter til træet ved at kalde *addLeaf*()-metoden i det givne **TreeConstructor**-objekt, tc.

Metoden *parseTopologi*() oversætter fra vilkårlige NEWICK-træer for tre blade til de fire faste strenge, der repræsenterer topologierne. Denne metode stilles til rådighed for at samle oversættelsen mellem generelle NEWICK-træer for tre arter til topologier som **TreeConstructor** kan forstå. Imple-

mentationer af **Experimenter** kan frit benytte denne metode eller direkte returnere en af de fire konstanter.

Det konstruerede træ kan til enhver tid hentes fra trækonstruktionsobjektet med metoden *getNewickFormat*(), der som navnet angiver returnerer træet i en NEWICK-formateret streng.

```
package speciale.treeconstruction;

public abstract class Experimenter {
  public static final String type1 = "(a,b,c)";

  public static final String type2 = "((a,b),c)";

  public static final String type3 = "((a,c),b)";

  public static final String type4 = "((b,c),a)";

  public abstract int constructTree(String[] args);

  public abstract String performExperiment(TreeLeaf a,
                                           TreeLeaf b,
                                           TreeLeaf c);

  public String parseTopologi(TreeLeaf a, TreeLeaf b,
                              TreeLeaf c, String topologi)
        throws TopologiParseException {
    //kode fjernet
  }
}
```

Figur 4.2: Klassen **Experimenter**, kun metodesignature med *public* adgang.

Figur 4.3 viser en simpel implementation, der indsætter et blad for hvert kommandolinie-argument, og altid besvarer eksperimenter med type2. Dette illustrerer den simple adgang til at udvide implementationen.

Her viser vi hvorledes **MyExperimenter** kompileres og anvendes, i dette tilfælde i en linux-shell. Bemærk, at det af hensyn til Javas klasseindlæsning

er nødvendigt at pakke klassen i en JAR-fil. Argumentet "-Djava.ext.dirs=." angiver at JAR-filen ligger i det aktuelle directory.

Kompilering af MyExperimenter

```
dossen@pc # javac -classpath treeconstruction.jar -d . \
            myexp/MyExperimenter.java

dossen@pc # ls myexp
MyExperimenter.class  MyExperimenter.java

dossen@pc # jar -cvf myexp.jar myexp/MyExperimenter.class
added manifest
adding: myexp/MyExperimenter.class(in = 980)
(out= 503)(deflated 48%)

dossen@pc # jar -tvf myexp.jar
     0 Sat May 28 15:07:02 CEST 2005 META-INF/
    68 Sat May 28 15:07:02 CEST 2005 META-INF/MANIFEST.MF
   980 Sat May 28 15:06:40 CEST 2005 myexp/MyExperimenter.class

dossen@pc # java -Djava.ext.dirs=. -jar treeconstruction.jar \
            myexp.MyExperimenter a b c d
(((a,d),c),b);
```

Ved denne fremgangsmåde kan implementationen udvides og benyttes uden at modificere den eksisterende kode. I kapitel 6 beskriver vi en række foreslag til udvidelser, hvoraf nogle også kræver modifikation af den eksisterende kode.

```
//MyExperimenter skal placeres i pakken myexp
package myexp;

//Importer klasser
import speciale.treeconstruction.Experimenter;
import speciale.treeconstruction.TreeConstructor;
import speciale.treeconstruction.TreeLeaf;

public class MyExperimenter extends Experimenter {

  //Konstruerer et træ for de givne argumenter
  public int constructTree(String[] args) {
    for(int i = 0; i < args.length; i++) {
      //Indsæt et blad for det i'te argument
      this.tc.addLeaf(new TreeLeaf(args[i]));
    }
    //Udskriv det konstruerede træ
    System.out.println(this.tc.getNewickFormat());
    //Rapporter at der ikke opstod fejl
    return 0;
  }

  //Udfør eksperiment for arterne a, b og c
  public String performExperiment(TreeLeaf a, TreeLeaf b,
                                  TreeLeaf c) {
    //Rapporter at topologien er ((a,b),c)
    return Experimenter.type2;
  }
}
```

Figur 4.3: Klassen **MyExperimenter**.

Kapitel 5

Resultater

Vi har eksperimenteret med flere aspekter af koden, primært udførelsestiderne og det udførte antal eksperimenter, for at validere at implementationen lever op til algoritmernes lovede, teoretiske resultat.

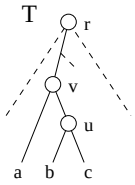
Desuden har vi undersøgt kvaliteten af de konstruerede træer. På støjende ultrametriske data er kvalitetskravet simpelt: Det konstruerede træ skal have samme topologi som det sande træ. Givet data, der ikke opfylder kravet om støjende ultrametrik, har vi ikke nogen garantier for kvaliteten. For fuldstændighedens skyld har vi konstrueret en række træer og undersøgt, hvor meget de har tilfælles med træer konstrueret med en anden algoritme, neighbour-joining (se [DEKM98]), der kan konstruere træer ud fra data, der ikke er støjende ultrametriske.

5.1 Testdata

Til vores eksperimenter har vi brugt data fra to kilder: Den første samling træer er genereret med et program fra kurset "Algorithms in Bioinformatics, Efterår 2003"[1], på baggrund af Pfam databasen[2], der i den udgave, vi har benyttet, indeholder matricer for familier af proteiner med op til 3000 arter. Programmet anvender neighbour-joining-algoritmen, fra [DEKM98]. De genererede træer har vi anvendt, hvor vi ikke behøvede eller ønskede ultrametriske træer. Den anden samling består af tilfældigt genererede, ultrametriske træer. Disse er lavet med `r8s`[3], der ved at starte med et givet antal blade og tildele dem tilfældige forfædre kan opbygge ultrametriske træer med præ-

[1] Beskrevet på `http://daimi.au.dk/~cstorm/courses/bioinf_e03/project3.html`
[2] Tilgængelig fra `http://www.sanger.ac.uk/Software/Pfam/`
[3] Kan findes på `http://ginger.ucdavis.edu/r8s/index.html`

cis det antal blade, vi ønsker. I denne sammenhæng skal det bemærkes, at afstandsmatricen for et ultrametrisk træ også er støjende ultrametrisk, eftersom ultrametrisk er et stærkere krav. At et træ er ultrametrisk betyder, at alle bladene har samme afstand til roden af træet. Dvs. at i en matrice for et ultrametrisk træ skal gælde de krav opskrevet i definition 1. Dette indses ved at betragte figur 5.1, hvor T er et ultrametrisk træ, i dette udvælges et undertræ rodfæstet ved knuden v. Matricen for dette undertræ T_v er også ultrametrisk, da afstanden fra bladene a, b og c er den samme til roden r af T. Vi skriver M_{ac} for afstanden mellem arterne a og c. Vi kan fraregne M_{vr}, da denne er ens for de tre arter, hvilket giver at a, b og c har lige langt til v og dermed er T_v et ultrametrisk undertræ af T. For at matricen for træet T_v er støjende ultrametrisk, skal M_{bc} være mindre end M_{ab} og M_{ac}, hvis b og c har laveste fælles forfader, der er lavere end for a og b (eller c). Fra figuren ses at $M_{bc} = |(b,u)| + |(c,u)|$ og $M_{ab} = |(b,u)| + |(u,v)| + |(v,a)|$ (og tilsvarende for c). Da $|(b,u)| = |(c,u)| < |(v,a)|$, får vi at $M_{bc} < min(M_{ab}, M_{ac})$, og da u sidder lavere end v i T_v følger det, at matricen for det ultrametriske træ også er støjende ultrametrisk.



Figur 5.1: På figuren ses et ultrametrisk træ T

5.1.1 Matricer

Vi har lavet forsøg med matricer, selvom de ikke repræsenterer en effektiv metode til at udføre eksperimenter, medmindre omkostningen ved indlæsning kan undgås. Dette ville kunne opnås ved at benytte et format, hvor indgangene nemt kan adresseres direkte, eller ved kun at beregne de nødvendige afstande, som vi kommer ind på i afsnit 6.2.1. For at kunne verificere resultaterne, har vi selv konstrueret matricerne på baggrund af de træer, vi har fremstillet til vores eksperimenter. Da matricerne er fremstillet ved at måle afstande i ultrametriske træer, er de selv ultrametriske, hvorfor vi forventer at kunne rekonstruere de korrekte træer.

5.1.2 Højere udgrad

For at få træer med en højere udgrad, tog vi udgangspunkt i vores genererede ultrametriske træer. Vi foretog et dybde-først-gennemløb, hvor vi slog så mange børn som muligt sammen, op til en given øvre grænse for udgraden, med justering af kantlængderne, så træerne forblev ultrametriske.

5.1.3 Problemer

Et væsentligt problem ved den metode, hvorpå vi laver træer med højere udgrad, er bevarelse af ultrametrik egenskaben. Da vi benyttede Javas **double**-type, altså en flydende-komma-type, var det ikke muligt at garantere, at afstandene i træerne forblev eksakt de samme, når vi lagde kantlængder sammen. For at løse det har vi skiftet repræsentationen af længder i **NewickTree**- og **DistanceMatrix**-klasserne ud med skalerede **long**s. Da **long** er en heltalstype har addition den normale, ønskede opførsel, så længe størrelserne ikke overskrider grænserne for det, der kan repræsenteres.

Det var imidlertid ikke tilstrækkeligt at skifte repræsentation, da de resulterende træer havde stadig afvigelser på sidste decimal. Da dette ikke kan undgås,[4] løber vi alle blade i de genererede træer igennem efter sammenlægningen af kanterne og justerer deres kantlængder, således at alle blade er lige langt fra roden. Herved er træerne ultrametriske igen, og deres udgrad er nu hævet til den ønskede størrelse.

5.2 Eksperimenter på afstandsmatricer

De følgende eksperimenter er udført på afstandsmatricer, der er udledt fra vores konstruerede, ultrametriske træer. Dermed havde vi, udover input til algoritmen, også en facitliste. Det giver os, at det rekonstruerede træ kan valideres for korrekthed.

Selve algoritmen kan håndtere træer med arbitrær udgrad, men vi vil i det følgende præsentere udførelsestider samt forbruget af eksperimenter i to tilfælde: Når udgraden er 2, dvs. et binært træ og når udgraden er større end 2, dvs. træer med arbitrær udgrad.

[4] r8s repræsenterer kantlængder ved flydende-komma tal, så når vi begynder at lægge dem sammen, er det umuligt at undgå afvigelser. Da vi ændrer på topologien af træerne kan dette medføre, at et træ ikke længere er ultrametrisk.

5.2.1 Binære træer

Først kigger vi på binære træer fra vores sæt af testdata.

Fremgangsmåde

Eksperimenterne blev udført på en dedikeret maskine, der ikke blev benyttet til andre formål under eksperimenterne. Dataopsamling er sket via Javas indbyggede ur, der leverer tidsstempler med millisekundpræcision. Ved at tage tid ved hvert kald til algoritmens hovedfunktion, *addLeaf*(), og ved alle kald videre derfra til eksperimentimplementationen, er det muligt at give såvel det samlede tidsforbrug som den del af tiden, der er brugt på henholdsvis selve algoritmen samt udførslen af de eksperimenter, der foretages af algoritmen. Vi har yderligere instrumenteret koden, således at vi kan tælle, hvor mange eksperimenter, der bliver udført.

Da vi måler rigtig tid, og ikke CPU-tid, er det muligt for eksterne faktorer at påvirke vores resultater, selvom maskinen ikke havde andre opgaver. For at mindske det problem er alle kørsler gentaget 5 gange, og gennemsnittet over disse kørsler er benyttet til graferne (i nogle tilfælde er der flere træer med samme størrelse, så der er datapunkter med så mange som 15 kørsler).

Med både træer og matricer ved hånden kan vi også validere, at de genererede træer er korrekte. Samtlige konstruerede træer er i fuld overensstemmelse med de kendte.

5.2.2 Antal eksperimenter

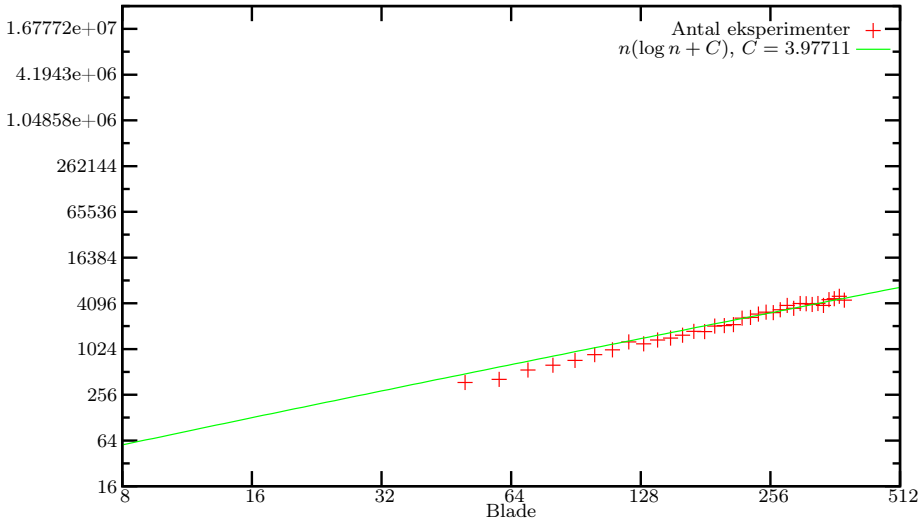
På figur 5.2 (større format kan ses i appendiks C.1) har vi plottet det anvendte antal eksperimenter for forskellige matrice-størrelser. Som reference har vi også plottet det teoretiske antal eksperimenter, $n(\log n + C)$, hvor C er en konstant fundet ved hjælp af *curve-fitting*[5] i `gnuplot`.

Det udførte antal eksperimenter ser tydeligvis ud til at have den forventede størrelse.

5.2.3 Tidsforbrug

Ligeledes har vi, på figur 5.3 (større format kan ses i appendiks C.1), plottet tidsforbruget. Igen har vi medtaget en referencekurve, der repræsenterer den teoretiske udførelsestid, $C \cdot n \log n$, som viser at tidsforbruget ser fornuftigt ud. Problemstørrelsen er imidlertid så lille, at det ikke er rimeligt at

[5]Curve-fitting funktionen tager en funktion med en eller flere konstanter og optimerer konstanterne, så funktionen afviger mindst muligt fra et givet sæt af datapunkter.



Figur 5.2: Antal eksperimenter for $d = 2$

drage konklusioner på dette grundlag, da træerne for de største af matricerne rekonstrueres med under 1 sekund anvendt i algoritmen.

5.2.4 Problemer

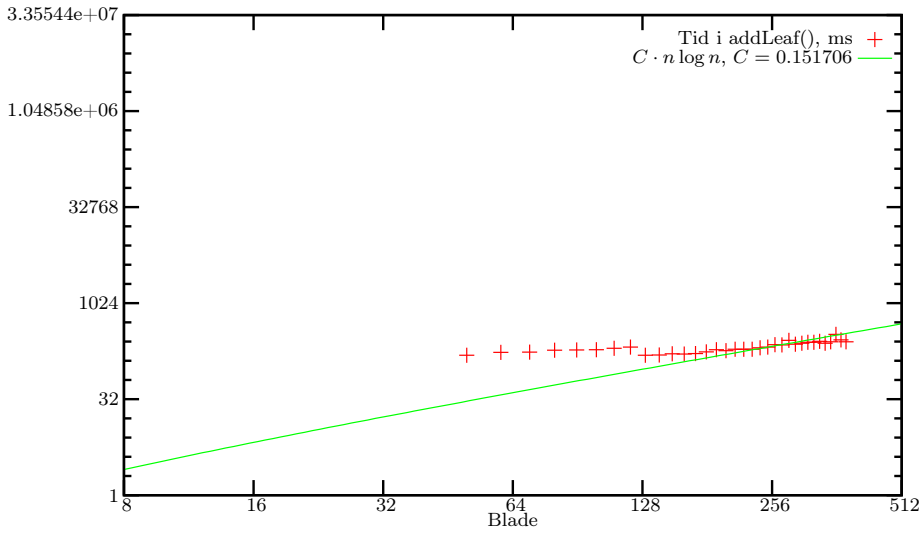
Som nævnt er de byggede træer for små til at kunne konkludere noget om algoritmen. Dette problem skyldes anvendelsen af afstandsmatricer, der med kvadratisk kompleksitet i tid og plads, gør udførelsen af vores algoritme til en mindre del af den tid vores eksperimenter tager.

5.3 Træer med højere udgrad

Da algoritmen er defineret for træer med arbitrær udgrad, har vi også testet vores algoritme på sådanne træer.

5.3.1 Fremgangsmåde

Der er ikke ændret noget i fremgangsmetoden i forhold til træer med udgrad 2. Der er blot benyttet de træ-sæt vi genererede ved at slå knuder sammen, som beskrevet i afsnit 5.1.2. Konkret har vi lavet træer med en række udgrader mellem 3 og 42, som vi mener at have valgt repræsentativt.



Figur 5.3: Tidsforbrug for $d = 2$

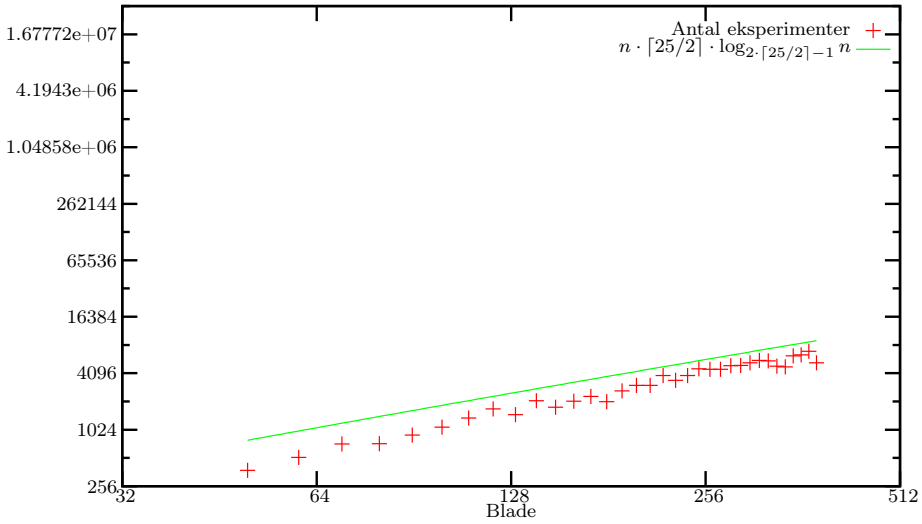
5.3.2 Antal eksperimenter

Som repræsentativt eksempel har vi her, på figur 5.4, plottet det anvendte antal eksperimenter for træer med udgrad 25. Resultaterne af vores øvrige eksperimenter med træer af højere udgrad end 2 er i bilag C.2.1. Igen ser vi tydeligt, ved at sammenligne med den indlagte referencekurve, at algoritmen ved disse problemstørrelser opfører sig som ventet.

5.3.3 Tidsforbrug

Vi har ligeledes plottet tidsforbruget for $d = 25$, på figur 5.5. Vores implementation overholder igen den teoretiske lovede udførelsestid, $O(25 \cdot n \log_{25} n)$, jvf. referencekurven.

Udførelsestiderne for de øvrige træer med varierende udgrad kan ses i bilag C.2.2. Generelt ses det at implementationen følger den specificerede udførelsestid. Ligesom i det binære tilfælde mangler vi at køre på nogle større data for med rimelig sikkerhed at kunne påstå at udførelsestiden lever op til den teoretiske forventning.



Figur 5.4: Antal eksperimenter for $d = 25$

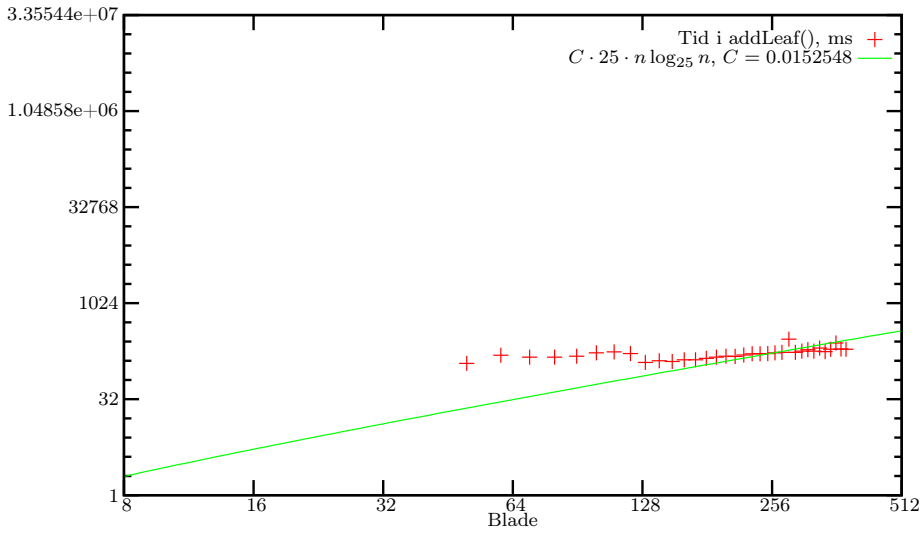
5.3.4 Problemer

Problemet er det samme som for de binære træer: Matricer er ikke effektive, hvis man, som vi gør, indlæser dem i deres helhed.

5.4 Eksperimenter direkte på træer

Selvom det fra et anvendelsessynspunkt er fuldstændigt uinteressant at rekon struere træer for allerede kendte træer, så mener vi, der er flere gode grunde til at foretage sådanne eksperimenter i denne sammenhæng. For det første er tidsmålingerne, som vi foretager dem, uafhængige af hvilken metode vi benytter til at besvare eksperimenterne.[6] Yderligere er vi i alle tilfælde nødt til at kende de sande træer for at kunne udtale os om korrektheden af det givne resultat, så fra det synspunkt er træbaserede eksperimenter også en god ide. Endeligt kan vi i praksis udføre flere og større eksperimenter med træer end med matricer, så hermed får vi mulighed for at undgå den mangel på data for store input, som gjorde, at vi ikke kunne afgøre performance, udfra de netop gennemgåede eksperimenter på matricer.

[6] Den samlede tid bliver selvfølgeligt påvirket, men eftersom eksperiment-modellen tager "udfør eksperiment" som en fundamental operation, så giver det god mening at fraregne tiden anvendt på eksperimenter. Denne dækkes af *antallet* af udførte eksperimenter.



Figur 5.5: Tidsforbrug for $d = 25$

5.4.1 Binære træer

Her er anvendt samme fremgangsmåde og data som ved afstandsmatricer for de binære træer. Blot besvares eksperimenterne ikke ved at slå op i matricerne, men i stedet ved at undersøge topologien i træerne.

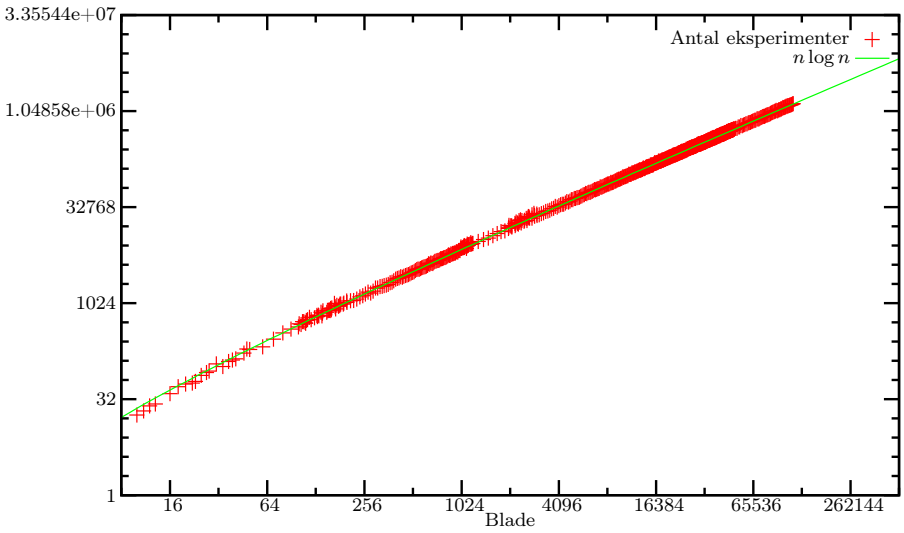
Antal eksperimenter

Som det fremgår af figur 5.6 (større format kan ses i appendiks D.1) er det anvendte antal eksperimenter i fuldstændig overensstemmelse med forventningerne, $n \log n$.

Tidsforbrug

På figur 5.7 (større format kan ses i appendiks D.1) har vi plottet den anvendte tid til indsættelse af blade i træet, altså den tid, der sammenlagt er tilbragt i *addLeaf*()-metoden, fraregnet den tid, der benyttes til at udføre eksperimenter.

Igen har vi fundet en brugbar konstant og plottet det teoretiske tidsforbrug med denne konstant. Heraf ses det, at algoritmen asymptotisk bruger tid som forventet. Ydermere ses der ved store data en svag tendens til at udførelsestiden med mellemrum, der svarer til en fordobling i input, pludseligt



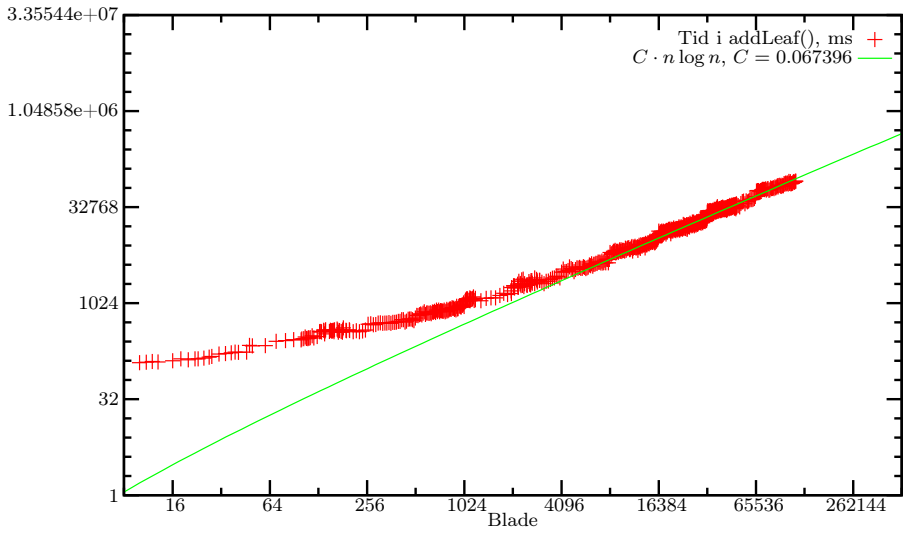
Figur 5.6: Antal eksperimenter for $d = 2$

springer lidt. Dette stemmer godt overens med de underliggende datastrukturers design, hvor rekonstruktion foretages ved fordobling af størrelsen. Hermed vil der blive foretaget det samme antal, x, rekonstruktioner for inputstørrelser i intervallet $[n : 2n-1]$, medens der for $2n$ knuder i datastrukturen vil forekomme $x+1$ rekonstruktioner.

For at vurdere præcisionen af vores udspecificerede tidsmålinger har vi på figur 5.8 plottet afvigelsen mellem de udspecificerede katagorier (besvarelse af eksperimenter og indsættelse af bladet) og det samlede forbrug af tid (på udførsel af algoritmen, indlæsning af input og generering af output er stadig ikke medregnet). Det ses, at vores tidsmålinger tager højde for over 95% af den anvendte tid, hvor en væsentlig del af resten er anvendt til at finde det næste blad og klargøre dette til indsættelse.

Problemer

Vi kræver at bladene har unikke navne, dette gav os umiddelbart det problem, at de træer, vi havde fremstillet ved neighbour-joining ikke havde unikt navngivne blade. De oprindelige afstandsmatricer havde navnene trunkeret til 10 tegn, og i nogle tilfælde gav det adskillige knuder med de samme navne. For at undgå dette problem tilføjer vores Newick-parser, som nævnt i afsnit 3.6.4, en unik streng til de knuder, der ikke i forvejen har et unikt navn.



Figur 5.7: Tidsforbrug for $d = 2$

5.4.2 Træer med højere udgrad

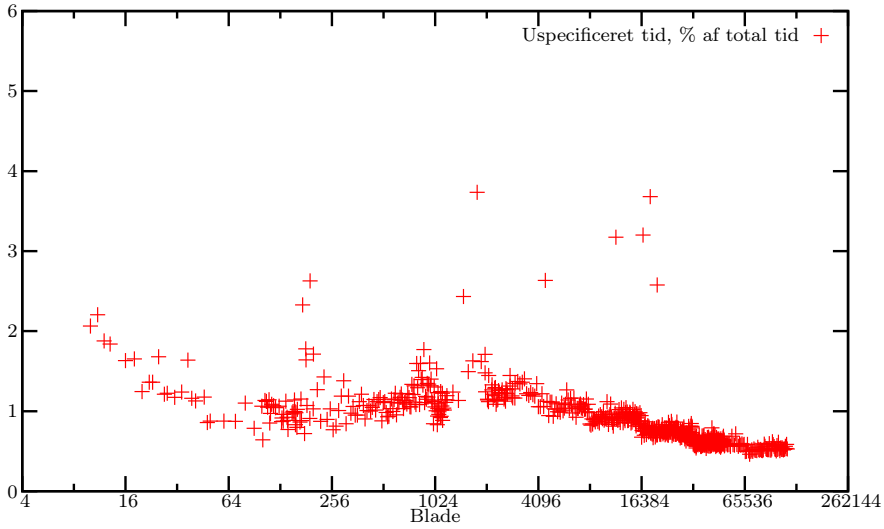
Igen adskiller disse eksperimenter sig kun fra de matricebaserede ved eksperimentmetoden.

Antal eksperimenter

Figur 5.9 viser det anvendte antal eksperimenter for udgrad 25, valgt som et repræsentativt eksempel for træer med arbitrær udgrad. Graferne for de andre udgrader kan ses i bilag D.2.1, figur D.4 – D.19. Der er også her plottet referencekurver.

Tidsforbrug

Den næste figur, 5.10 viser tidsforbruget, igen for udgrad 25. Graferne for alle de kørte udgrader er figur D.20 – D.35 i bilag D.2.2. Igen har vi plottet udførelsestiden og fundet konstanten. En interessant tendens, som kan ses på figurerne i bilaget, er at konstanten ikke er konstant, men svagt aftagende for voksende d.



Figur 5.8: Uspecificeret tidsforbrug

Problemer

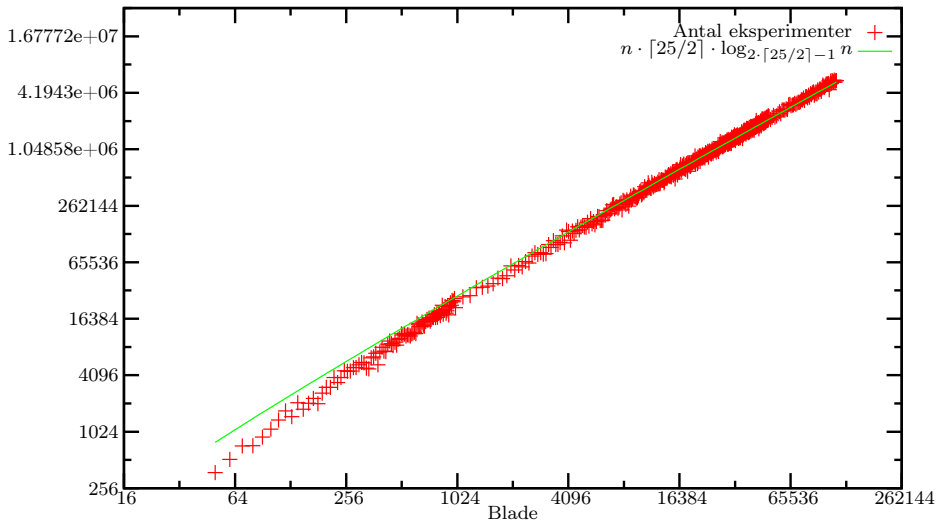
Det eneste problem, vi har haft med træer af arbitrær udgrad, er at det tog lang tid at lave selve dataopsamlingen. Vi har haft en maskine til at arbejde på vores datasæt i 3 uger for at lave den dataopsamling, som bliver præsenteret i de ovenstående grafer. Udover tidsfaktoren oplevede vi ikke andre problemer med træer af arbitrær udgrad.

5.5 Mere realistiske data

Her har vi forsøgt os med konstruktion af træer for de data, vi har fra Pfam-databasen, som nævnt i 5.1. Da disse data ikke er støjende ultrametriske matricer, har vi som nævnt ikke nogen forventning til kvaliteten af de konstruerede træer.

5.5.1 Fremgangsmåde

For at have træer at sammenligne med, har vi som nævnt anvendt neighbourjoining-algoritmen til at konstruere træer. Vi har dernæst kørt vores algoritme på de samme data. Dette giver os to samlinger af træer, der ikke kan forventes at være ens, så det næste spørgsmål er: Hvordan sammenligner



Figur 5.9: Antal eksperimenter for $d = 25$

vi dem? Da dette eksperiment mest tjener til at stille vores nysgerrighed, har vi valgt blot at kigge på et simpelt mål for, hvor meget de genererede træer har tilfælles.

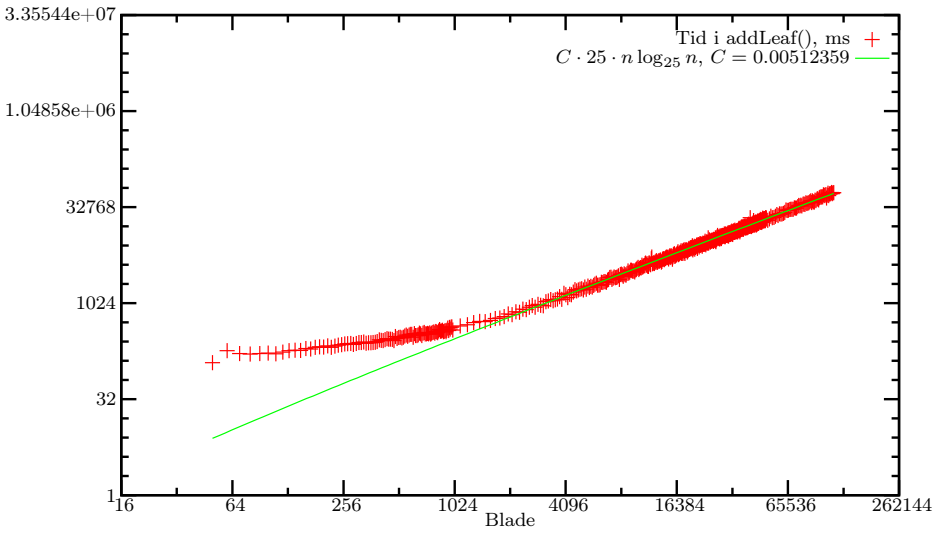
Et sådant mål er, hvor mange kanter træerne har tilfælles. Dette kan vi beregne med Split-Dist[7]. Programmet tæller antallet af ikke-trivielle kanter (kanter, der ikke har et blad i en af enderne), og afgør, hvor mange af disse deles mellem de sammenlignede træer. For at kunne vurdere resultaterne, har vi omregnet antallet af delte kanter til procent af det samlede antal kanter.

5.5.2 Resultat

I bilag E har vi inkluderet resultaterne af dette eksperiment i tabelform. Som det tydeligt fremgår, så er der for langt de fleste træer tale om meget få overensstemmelser. Og hvor der er flere kanter til fælles, er der tale om meget små træer.

Dette resultat siger naturligvis ikke meget om algoritmens anvendelighed, uden antagelser om, hvor godt neighbour-joining gør det. Givet mere tid og flere datasæt ville det være interessant at lave en grundigere undersøgelse af, præcist hvordan de træer, vores algoritme kan producere, relaterer til dem, andre algoritmer kan fremstille.

[7]Kan findes på `http://www.daimi.au.dk/~mailund/split-dist.html`.



Figur 5.10: Tidsforbrug for $d = 25$

5.6 Eksperimenter på datastrukturerne

Udover eksperimenter med den fulde algoritme har vi også udført en række eksperimenter med de underliggende datastrukturer. For at verificere, at vores implementationer lever op til de lovede udførelsestider, har vi afviklet datastrukturernes konstruktionsalgoritmer på vores samling af træer. Igen er kørslerne foretaget på en maskine, der ikke havde andre opgaver imens. Tidsmåling er også her opnået ved hjælp af Javas indbyggede ur.

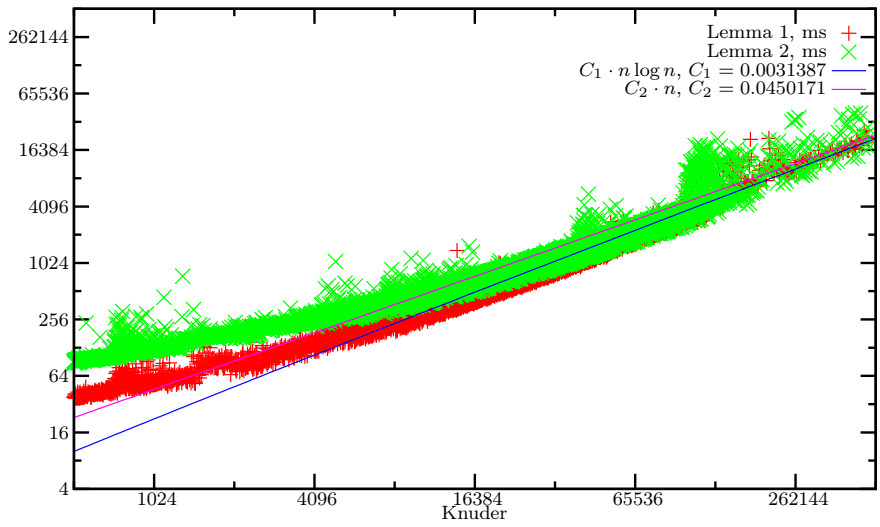
Som det ses i de følgende grafer, så er vi igen nede og måle meget små tidsrum, men i modsætning til tidsmålingerne på algoritmen, så er der her tale om et ubrudt tidsrum, hvorved afrundingsfejl undgås. Det er dog stadig nemt for udefra kommende forstyrrelser at påvirke data, som flere af datapunkterne klart indikerer.

5.6.1 Konstruktion af separatortræer

Den første benchmark, vi har udført, er konstruktion af separatortræer. Performance af denne algoritme er afgørende for hele implementationens udførelsestid, da konstruktion af separatortræer ligger til grund for alt, hvad vi gør.

På figur 5.11 ses de målte udførelsestider for konstruktion af separa-

tortræer, med begge de to implementerede algoritmer. Som det ses, er algoritmen fra lemma 2 langsommere for træer af de størrelser, vi har benyttet, mens den ved større træer vil vinde på at have den laveste asymptotiske tidskompleksitet. Dog er usikkerheden i målingerne så stor, at der i praksis er sammenfald for træer med cirka 30000 knuder eller mere.



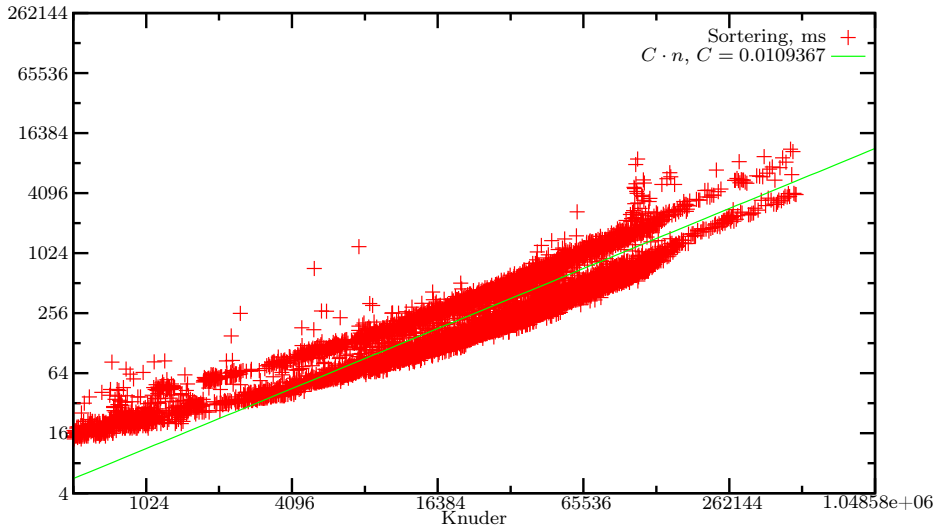
Figur 5.11: Tidsforbruget for de to algoritmer til konstruktion af separatortræer.

Ud fra dette eksperiment er det klart, at den simple algoritme, som lemma 1 giver anledning til, er mere effektiv for mange træer af en størrelse med praktisk interesse. I afsnit 6.2.4 kigger vi på en strategi for at drage fordel af dette.

Det skal også bemærkes, at lemma 2 giver anledning til en mere kompliceret algoritme, hvor der er større mulighed for optimeringer end i den algoritme, der kommer af lemma 1.

5.6.2 Sortering af separatortræer

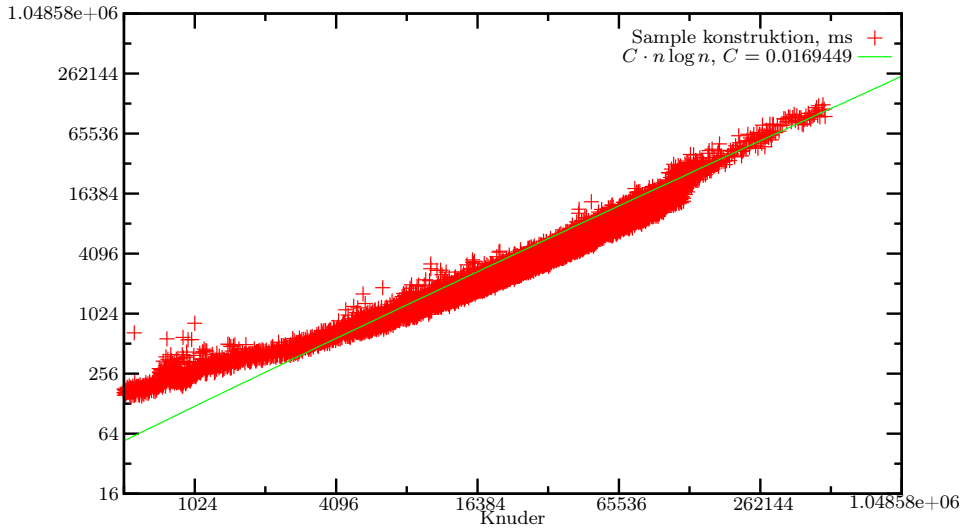
Det næste eksperiment, vist på figur 5.12, viser hvorledes udførelsestiden for sortering af separatortræet vha. lemma 5 afhænger af størrelsen på input. På figur 5.12 ses det, at der er opstået en to deling af datapunkterne, men at de begge vokser lineært i størrelsen af input. Vi har ingen forklaring på, hvorfor grafen er udformet som tilfældet er.



Figur 5.12: Tidsforbruget for sortering af separatortræer.

5.6.3 Konstruktion af samplen

Endelig har vi taget tid på konstruktion af samplen. Her skal det bemærkes, at vi konstruerer den fulde datastruktur, inklusiv sorterede separatortræer. Som vi ved fra teorem 2, har dette imidlertid ikke nogen betydning for den asymptotiske tidskompleksitet. Også her ses det at implementationen leverer det ønskede resultat.



Figur 5.13: Tidsforbruget for konstruktion af samplen.

Kapitel 6

Udvidelser og fremtidigt arbejde

I det følgende vil vi beskrive en række udvidelser og forbedringer, som ikke lod sig implementere indenfor den tidsramme, vores speciale udgjorde.

Først vil vi beskrive, hvordan algoritmen forventes at kunne gøres anvendelig på mere generelle data. Efterfølgende vil vi kigge på de områder, hvor implementationen kan gøres mere anvendelig for andre end os selv. Dette omfatter både anvendelse af selve den implementerede algoritme og anvendelse af dele af projektet til andre formål.

6.1 Heuristikker for virkelige data

Som vi tidligere har været inde på, kræver algoritmen, at resultaterne af alle eksperimenter er konsistente med det sande træ, for at kunne rekonstruere dette korrekt. Som det fremgår af afsnit 5.5 så håndterer algoritmen – som forventet – det ikke særlig godt, hvis matricerne ikke er støjende ultrametriske jvf. definition 1. Dvs. at svarene fra eksperimenterne i dette tilfælde ikke er konsistente med det sande træ, som model kan eksperimentmodellen ikke detektere disse inkonsistenser. Vi behøver altså en metode til at detektere disse inkonsistenser, samt efterfølgende en heuristik til at løse problemet.

En måde at detektere disse inkonsistenser på kunne være, at lave en kontrol på om resultatet af eksperimentet nu er korrekt. Dette kunne fx gøres ved i stedet for at spørge om, hvorledes arten a er relateret til arterne b og c, så kunne man spørge om, hvorledes a er relateret til en mængde af de arter, der sidder i henholdsvis samme undertræ som b og c. Dette betyder selvfølgeligt, at man udfører flere eksperimenter. Man kunne måske

nøjes med at lave et tilpas antal forespørgsler for at sikre, at der ikke opstår inkonsistenser.

Det kunne være interessant at implementere nogle heuristikker til at håndtere disse inkonsistenser og undersøge deres evner til at rekonstruere det sande træ.

Et par alternative heuristikker kunne fx være:

- "Vælg tilfældigt" – I tilfælde af at der ikke er et par af blade med mindst afstand, så vælges der et tilfældigt blad som det mindst beslægtede. Afhængigt af den underliggende metode, kan det også tænkes at tilfældigheden ikke behøver at være uniform over alle de tilladte topologier. Hvis eksempelvis type 2 kan udelukkes, så vil tilfældigt valg mellem type 3 og 4 naturligvis være oplagt.

- "Kræv lighed ved type 1 topologi" – Hvis eksperimentmetoden benytter et afstandsmål og resultatet af eksperimentet for tre arter ikke giver en type 2, 3 eller 4 topologi, så kan vi kræve at der skal være lighed mellem de tre parvise afstande, før der returneres type 1. Hvis der ikke er lighed må man deducere, hvilken af de tre andre topologier, man returnerer.

- "Byg et træ" – For tripletter, der ikke giver et svar, kan der bygges et træ ved andre metoder, eksempelvis neighbour-joining eller lignende, der kan bygge et passende træ uden at stille de samme krav til input. Hvis dette deltræ kan vedligeholdes og konsulteres når matricen ikke direkte kan besvare et eksperiment, så vil algoritmen i værste fald reduceres til udførselstiden på den valgte træbygningsalgoritme[1]. Hvis de hjælpetræer, der bliver bygget undervejs er små og der er tilstrækkeligt få af dem, er det imidlertid ikke utænkeligt, at heuristikken vil være hurtigere end den valgte hjælpe-algoritme ville være for hele træet.

En anden mulighed, der specielt kunne være nyttig i forbindelse med metoder, der direkte involverer fysiske eksperimenter, er simpelthen at lade brugeren løse inkonsistenser. Hvis der er tilstrækkeligt få, vil en manuel løsning måske være at foretrække.

[1] Vi antager her at hjælpealgoritmen har dårligere (asymptotisk) udførelsestid end vores.

6.2 Forbedringer af implementationen

Selvom vores implementation er væsentligt hurtigere end vi havde turdet håbe på, så er der oplagte muligheder for effektivisering og forbedring på implementationsområdet.

Den mest oplagte mulighed er at forsøge at minimere størrelsen af de centrale datastrukturer. En mulighed er for eksempel at erstatte instanser af Javas indbyggede datastrukturer med minimale implementationer. Ligeledes er der en lang række steder, hvor vi, for at sikre os at koden er robust, tester for korruption af datastrukturen. Disse tests kan naturligvis fjernes med en, formodentligt ikke helt ubetragtelig, tidsgevinst til følge.

Det er også oplagt at forsøge at optimere indstillingerne på den virtuelle maskine. Eksempelvis har vi allerede justeret allokationen af heapspace[2] til 512 MB, altså hele hukommelsen på den maskine vi benytter. Da maskinen også benytter hukommelse til andre formål betyder det, at vi risikerer at løbe tør for fysisk hukommelse og begynde at benytte langsom virtuel hukommelse, men i praksis ser det ikke ud til at det har haft indflydelse på vores resultater.

En anden justerbar parameter på den virtuelle maskine er størrelsen på kaldstakken. Under normal anvendelse har vi ikke observeret problemer fra den kant. Under udarbejdelsen af testkoden har vi imidlertid observeret enkelte forekomster af stakoverløb. Eksempelvis er konstruktion af separatortræer for træer, hvor den længste sti er over 1700 knuder lang, ikke muligt uden at modificere størrelsen på kaldstakken. Som sagt har vi ikke observeret problemet ved anvendelse af hele algoritmen, men det er naturligvis en begrænsning, der bør undersøges. Specielt når algoritmens evne til at håndtere meget store træer tages i betragtning.

Da vi allerede har konstateret, at profiling giver et glimrende grundlag for at optimere koden, vil det naturlige første skridt mod optimering være indsamling af flere data om algoritmens *hotspots*, altså steder hvor der tilbringes lang tid. Optimering vil naturligvis have størst virkning der. For at kunne undersøge det, er det imidlertid en forudsætning, at kunne indsamle data på større kørsler, hvilket vores aktuelle værktøjer ikke er velegnede til, da vores setup til at udføre mange kørsler ikke er integreret med Eclipse. Det burde dog ikke være et stort problem at etablere et profiling-setup, der er egnet til dette formål.

[2] Hvori objekter skabes.

6.2.1 Effektiv håndtering af matricer

Som vi tidligere har nævnt, så er indlæsningen af matricer tidsmæssigt krævende, og deres indhold af data er væsentligt større end det vi behøver for at kunne konstruere et træ.

Algoritmisk er der naturligvis ikke nogen måde at undgå problemet, indlæsning af n^2 indgange i en matrice tager naturligvis $O(n^2)$ tid. Hvis det er muligt at styre den underliggende metode, der benyttes til opbygning af afstandsmatricen, så kan vi imidlertid nøjes med at finde de afstande, der faktisk benyttes. En implementation af **Experimenter** vil således kunne konstruere de $O(n \log n)$ indgange i matricen, som behøves undervejs. Den konstruerede matrice kan selvfølgelig gemmes til senere brug. Dog skal et passende format benyttes, så vi ikke ender med igen at bruge kvadratisk tid til det.

6.2.2 Andet sprog

En anden mulighed for at forbedre implementationen er at omskrive den helt eller delvist i eksempelvis C eller C++. Vi har valgt at holde implementationen i Java for at kunne koncentrere indsatsen om at lave en korrekt implementation – men med en fungerende implementation og det tilhørende sæt af tests bør det være muligt at skrive en implementation i et andet sprog. Vi kan naturligvis ikke udtale os om gevinsten ved sådan en omskrivning, men alene muligheden for at få en hukommelsesmæssig optimal implementation bør give en gevinst, om ikke andet så i form af muligheden for at kunne konstruere endnu større træer.

6.2.3 Generalisering

Udover implementationen af algoritmen til trækonstruktion, så er der i projektet implementeret en række underordnede datastrukturer. Specielt kunne implementationen af separatortræer være interessant, isoleret set. Derfor bør dennes interface forbedres og generaliseres, så den kan anvendes i andre projekter. Specielt bør der designes et abstraktionslag, så datastrukturen kan benyttes uden at kende til **Sample**/**Component** systemet og andre optimeringer.

6.2.4 Hybridalgoritme

I afsnit 5.6.1 så vi, at ved endog ganske store træer er algoritmen fra lemma 1 faktisk den hurtigste til konstruktion af separatortræer. Hvis ikke optime-

ringer ændrer det billede væsentligt, vil en interessant optimering være at kombinere algoritmerne fra lemma 1 og lemma 2.

Da begge algoritmer producerer det samme resultat, kan de kombineres ved, at der for hver gang vi konstruerer et separatortræ, vælges den mest egnede algoritme. Dette kan opnås ved at vælge en passende størrelse, hvor vi forventer, at træet er stort nok til at strategien fra lemma 2 er mest effektiv. Alternativt kan der udføres en test i starten af programmet, som afgør, hvad den bedste størrelse at skifte strategi ved er.

Det skal naturligvis sikres, at begge algoritmer er i stand til at fungere, uden at blive forvirrede over hinandens efterladenskaber. Specielt skal det sikres, at der ikke gøres fejlagtige antagelser om datastrukturens udseende, i den tro at det var den samme algoritme, der behandlede træet i den foregående kørsel. Derudover vil det formodentligt være mest passende, at sikre at skiftet af algoritme altid falder sammen med en fuldstændig rekonstruktion af separatortræet. Vedligeholdelse med en anden algoritme end den, der byggede træet vil muligvis være vanskeligt. Da vedligeholdelse opnås ved at rekonstruere mindre dele af træet, vil det i hvert fald være passende at anvende lemma 1 algoritmen til vedligeholdelse af et separatortræ konstrueret med lemma 1 algoritmen. For lemma 2 algoritmen kan der muligvis være en fordel i at benytte lemma 1 algoritmen til vedligeholdelse, når genbygningen kun omfatter små undertræer.

Denne form for optimering påvirker naturligvis ikke algoritmens asymptotiske performance, men i praktiske tilfælde vil mange træer være så små, at det vil kunne betyde en reel gevinst. Specielt gør anvendelsen af samplen, at endog meget store træer vil involvere separatortræer i en størrelsesklasse, hvor lemma 1 har potentiale til at være hurtigere.

6.2.5 Kantlængder

Det er ikke oplagt hvordan, men da algoritmen alene producerer træer uden kantlængder, vil det være interessant, om man kan finde en metode, der kan sætte længder på kanterne. Muligvis kan man undervejs opsamle data, der i et senere skridt kan benyttes til effektivt at beregne længderne af træets kanter.

Kapitel 7

Konklusion

Det er af stor interesse for biologer m.fl. at rekonstruere den evolutionære historie for en mængde arter. Det er netop denne problematik, vi har arbejdet med i dette speciale i form af at realisere algoritmen fra [BFPÖ01].

Under hele udviklingsprocessen har vi haft megen fokus på udvikle veltestet og robust kode. Vi har således brugt meget energi på at både lave unit-test samt undersøge om disse test havde en god dækning af koden. Med dette arbejde mener vi at have skabt en stabil og korrekt implementation af algoritmen fra [BFPÖ01].

Korrektheden kommer af to ting: At implementationen givet et træ som input kan rekonstruere det selvsamme træ vha. eksperimenter, samt at det er lykkedes os at opfylde den teoretiske beskrevne udførelsestid for algoritmen, som det kan ses i kapitel 5.

Selvom vi ikke har foretaget videre optimeringer af koden, så er algoritmen, asymptotisk, så hurtig, at implementationen måske kan have en vis praktisk anvendelse.

Endelig så mener vi at have lagt et solidt grundlag, hvorpå udvikling af en mere optimeret implementation, interessante udvidelser eller andre forbedringer kan baseres. Specielt er eksistensen af en fungerende implementation samt testsuite nyttig, hvis det ønskes at implementere algoritmen i et andet sprog/miljø.

Den nuværende udgave af koden kan håndtere træer med ca. 115.000 arter, hvor inputtet til programmet er af lineær størrelse, på en ganske almindelig pc med 512Mb ram og med en P4 1.8Ghz processor på under 30 minutter. Som beskrevet i afsnit 6.2.1 skal der laves noget smart for at nedbringe indlæsningstiden i forbindelse med en matrice (kvadratisk størrelse), hvis man ønsker at rekonstruere store træer.

Litteratur

[BFPÖ01] Gerth Stølting Brodal, Rolf Fagerberg, Christian Nørgaard Storm Pedersen, and Anna Östlin. The complexity of constructing evolutionary trees using experiments. rs RS-01-1, Brics, Daimi, January 2001.

[DEKM98] Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mitchinson. *Biological sequence analysis*, chapter 7.1-7.3: Building phylogenetic trees. Cambridge University press, Cambridge, UK, 1998.

[GT98] Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java*. John Wiley and Sons, New York, NY, USA; London, UK; Sydney, Australia, 1998.

[KLW90] Sampath Kannan, Eugene Lawler, and Tandy Warnow. Determining the evolutionary tree. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 475–484. Society for Industrial and Applied Mathematics, 1990.

[LOÖ99] Andrzej Lingas, Hans Olsson, and Anna Östlin. Efficient merging, construction, and maintenance of evolutionary trees. In *Proceedings of the 26th International Colloquium on Automata, Languages and Programming*, pages 544–553. Springer-Verlag, 1999.

[Sky00] Sven Skyum. *Course notes for Algorithms*, chapter 1: A self-adjusting data structure - Splay Trees. Daimi, 2000.

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[Tar83] Robert Endre Tarjan. *Data Structures and Network Algorithms.* Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics, 1983.

Onlineressourser

`http://java.sun.com/j2se/1.4.2/docs/api/` er den officielle referencedokumentation til Javas klassebibliotek.

`http://junit.org/` er den officielle side for JUnit projektet.

`http://works.dgic.co.jp/djunit/` er hjemmesiden for dJUnit.

`http://evolution.genetics.washington.edu/phylip.html` beskriver filformaterne – NEWICK og PHYLIP.

`http://daimi.au.dk/~cstorm/courses/bioinf_e03/project3.html` beskriver det projekt, hvorunder neighbour-joining programmet blev skrevet.

`http://www.sanger.ac.uk/Software/Pfam/` er en database over proteinsekvenser, hvorfra vi har benytte afstandsdata.

`http://ginger.ucdavis.edu/r8s/index.html` er et værktøj, der kan konstruere ultrametriske træer.

`http://www.daimi.au.dk/~mailund/split-dist.html` er et værktøj til sammenligning af træers topologier.

`http://www.proweb.org/treeviewer/` er en webservice, der kan tegne træer, ud fra filer i NEWICK-format.

Bilag A

Abstract in english

Abstract

In the world of bioinformatics, evolutionary trees are of great interest. Because these trees don't exist, they have to be reconstructed. Unfortunately, the methods for doing this often demand a great amount of time. Nevertheless there is a family of algorithms which uses so-called "experiments" and are considerably more time-efficient. In this thesis we present an implementation of an algorithm for reconstructing evolutionary trees in $O(nd \log_d n)$ time, where d is the degree of the tree and n is the number of species in it. The algorithm requires $n\lceil d/2 \rceil (\log_{2\lceil d/2 \rceil - 1} n + O(1))$ experiments for $d > 2$ og $n(\log n + O(1))$ for $d = 2$. The algorithm was developed by Brodal et. al. and it was published in "The Complexity of Constructing Evolutionary Trees Using Experiments". Our implementation, written in Java, is developed with the intent to be incorporated in other projects based on the experiment model.

Bilag B

Formater

I det følgende vil vi kort gennemgå de anvendte filformater. Både NEWICK og PHYLIP (afstandsmatricer) er simple tekstformater, der blandt andet anvendes af programpakken `Phylip`. De to formater er beskrevet på henholdsvis `http://evolution.genetics.washington.edu/phylip/newicktree.html` og `http://evolution.genetics.washington.edu/phylip/doc/distance.html`.

B.1 NEWICK-format

NEWICK-formatet enkoder et træ ved hjælp af parentetisering. Formatet understøtter angivelse af kantlængder.

Nedenstående grammatik viser det, vi har fundet nødvendigt at kunne parse.

⟨*leaf*⟩ ::= ⟨*name*⟩
  | ⟨*name*⟩ ':' ⟨*length*⟩

⟨*node*⟩ ::= ⟨*leaf*⟩
  | '(' ⟨*nodelist*⟩ ')'
  | '(' ⟨*nodelist*⟩ ')' ':' ⟨*length*⟩

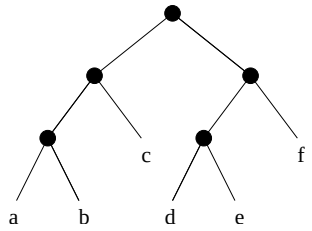
⟨*nodelist*⟩ ::= ⟨*node*⟩ ',' ⟨*nodelist*⟩
  | ⟨*node*⟩

⟨*tree*⟩ ::= '(' ⟨*nodelist*⟩ ')' ';'

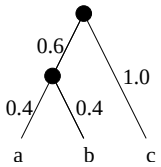
⟨*name*⟩ er navnet på et blad, som streng, og ⟨*length*⟩ er en endelig decimalbrøk.

Eksempelvis repræsenterer $(((a, b), c), ((d, e), f))$; træet på figur B.1. Her er træet antaget at være rodfæstet ved knuden repræsenteret ved det yderste sæt parenteser.

På figur B.2 er vist træet $((a : 0.4, b : 0.4) : 0.6, c : 1.0);$, med kantlængder.



Figur B.1: Træet $(((a, b), c), ((d, e), f));$.



Figur B.2: Træet $((a : 0.4, b : 0.4) : 0.6, c : 1.0);$.

B.2 PHYLIP-format

PHYLIP-formatet repræsenterer en afstandsmatrice i tekstformat.

Filen indledes med en tabulatorkarakter, efterfulgt af antallet arter i matricen. Dernæst følger en linie for hver art. Hver linie begynder med artens navn. Dernæst kommer en endelig decimalbrøk for hver art i matricen, der angiver afstanden mellem den aktuelle art og de øvrige.

Eksempelvis repræsenterer følgende matrice afstandene i træet på figur B.2:

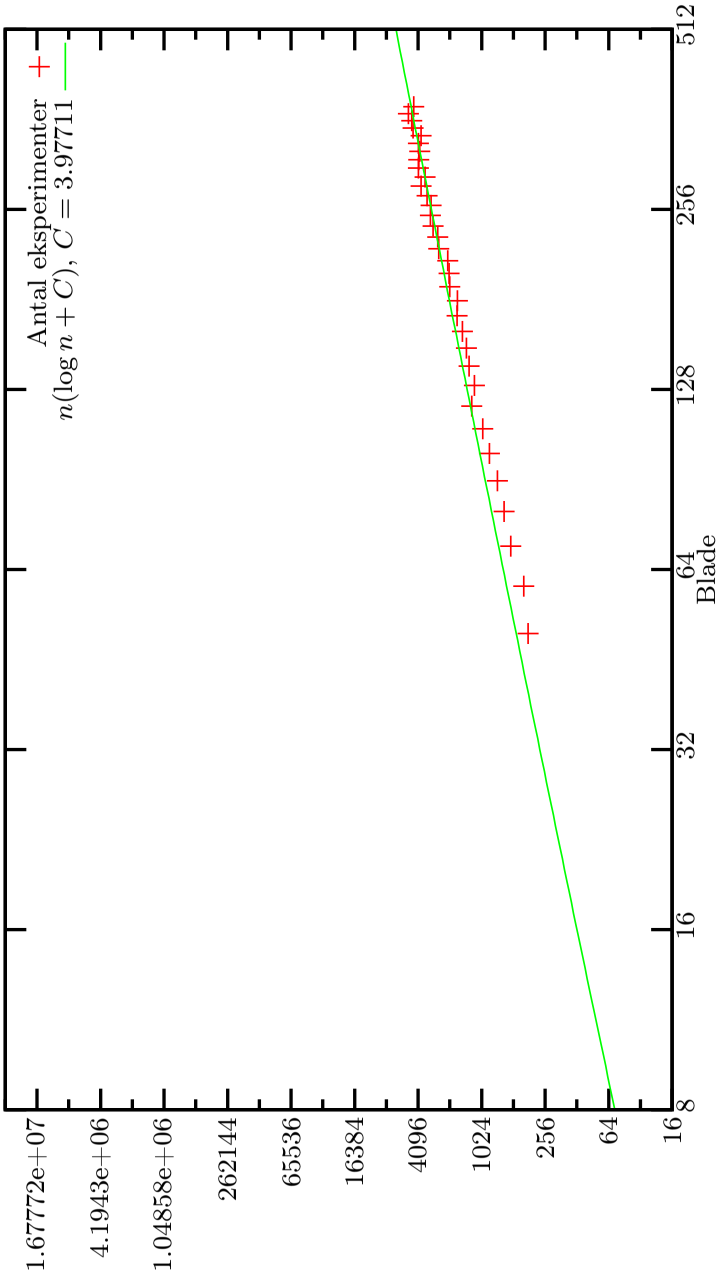
abc.phylip

```
        3
a   0.00000   0.80000   2.00000
b   0.80000   0.00000   2.00000
c   2.00000   2.00000   0.00000
```

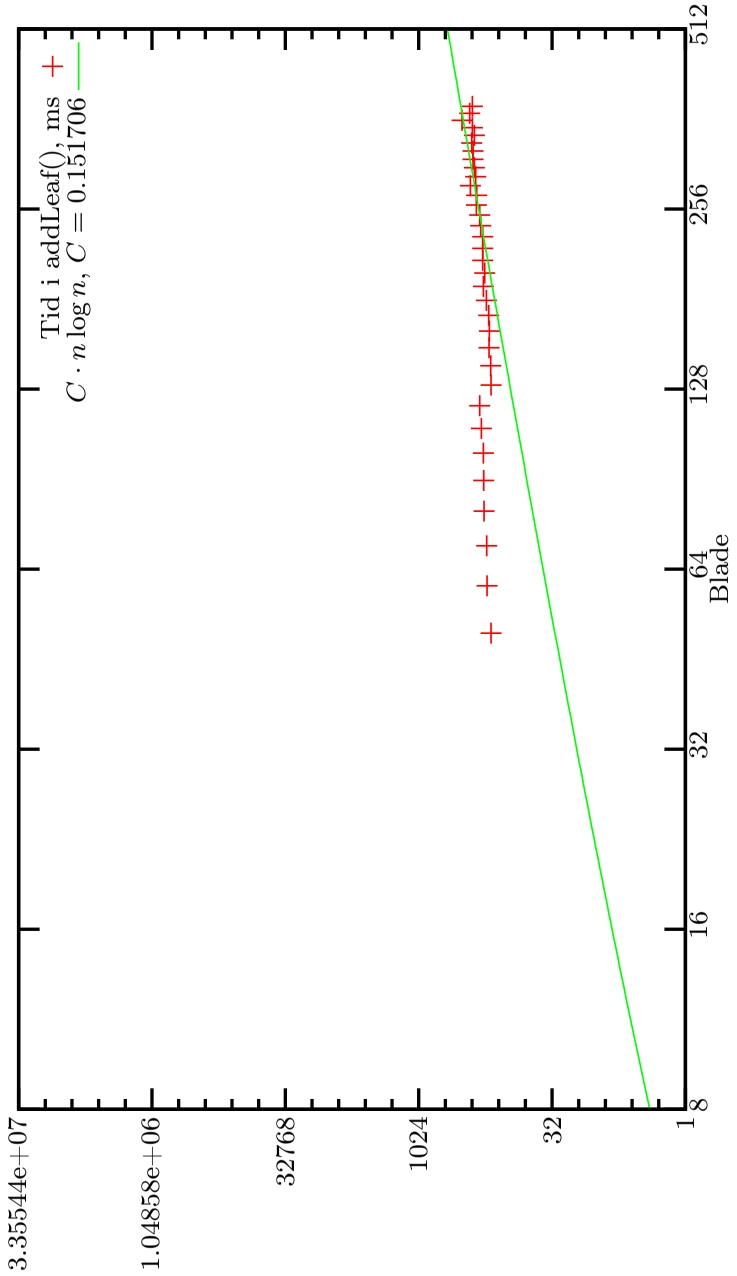
Bilag C

Eksperimenter på matricer

C.1 Binære træer



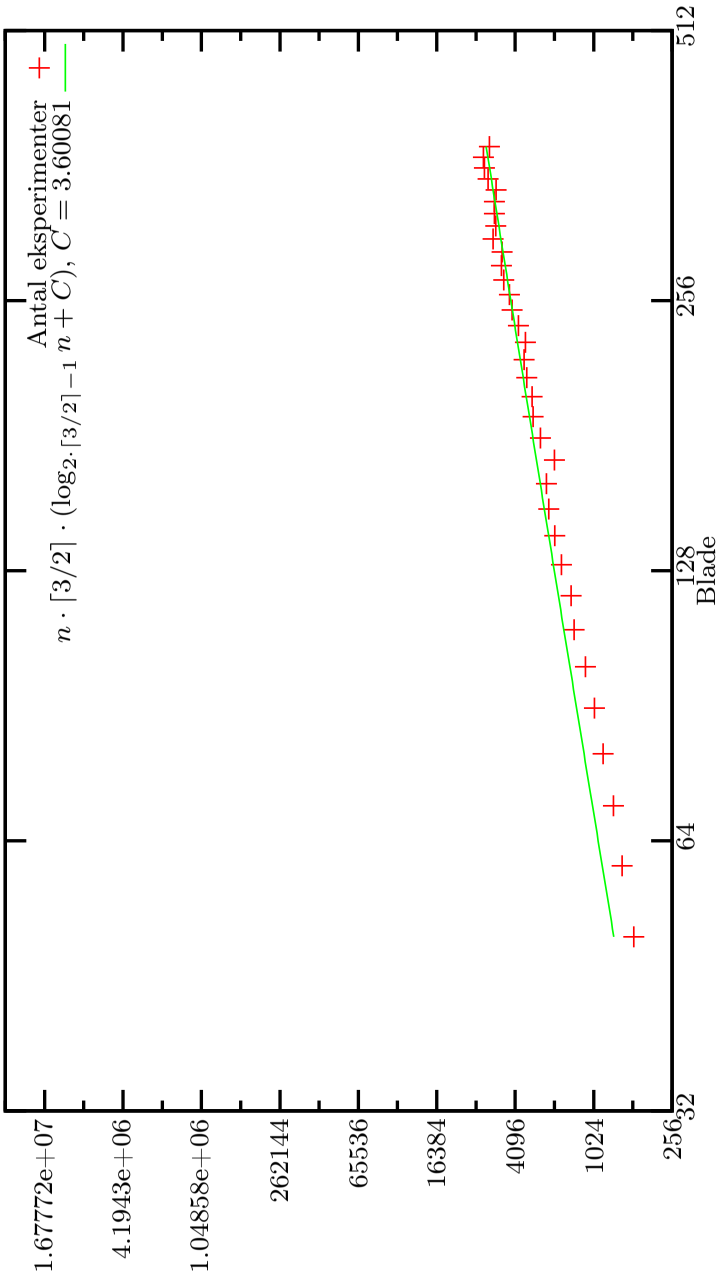
Figur C.1: Matricer, antal eksperimenter for $d = 2$



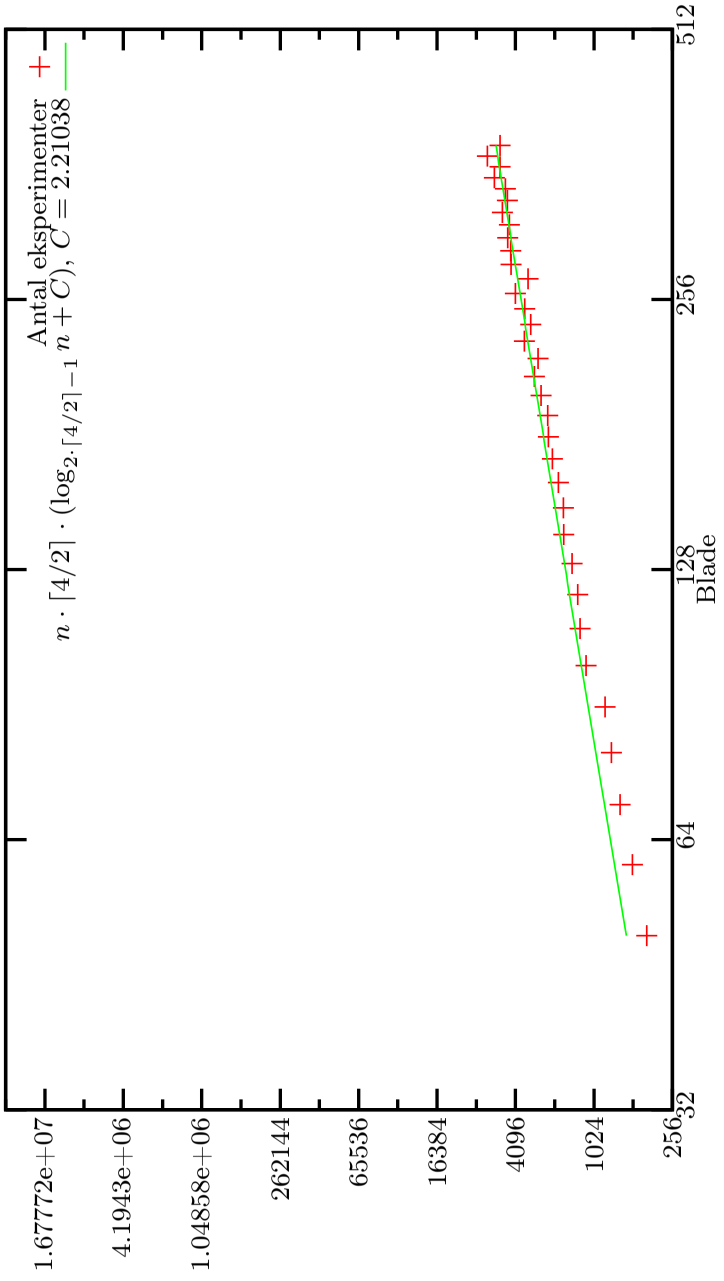
Figur C.2: Matricer, tidsforbrug for $d = 2$

C.2 Træer med højere udgrad

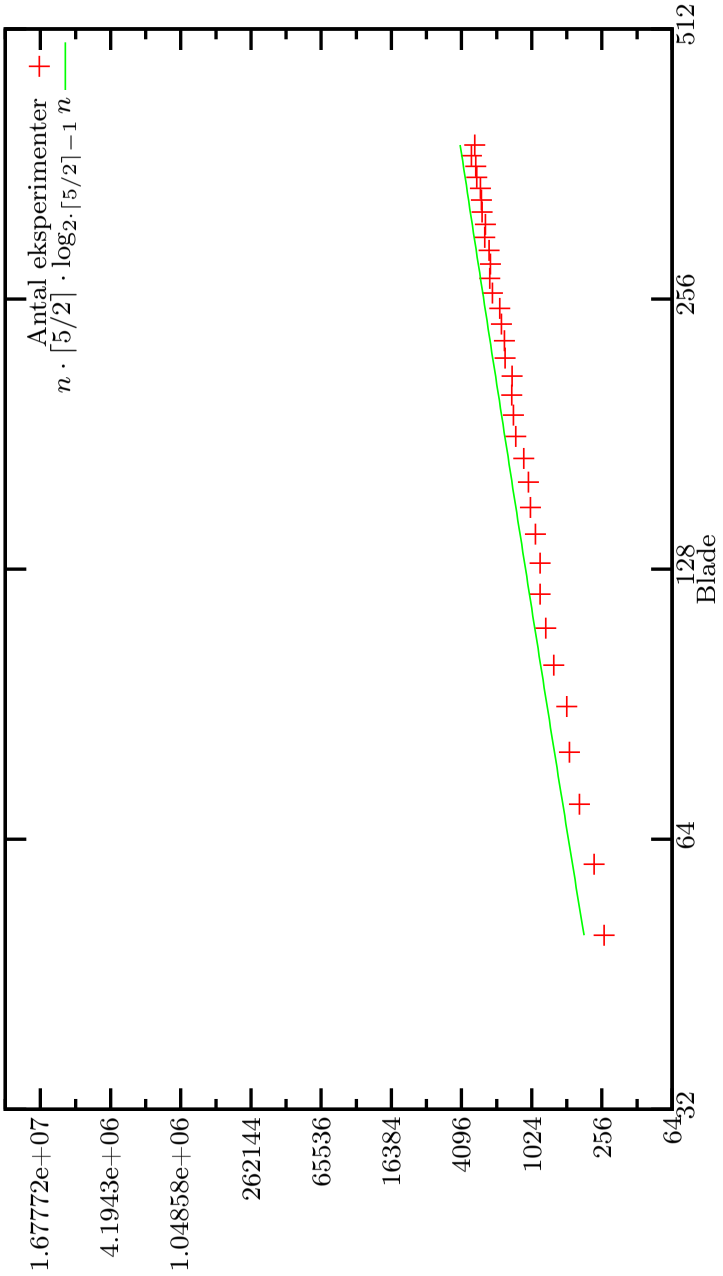
C.2.1 Antal eksperimenter



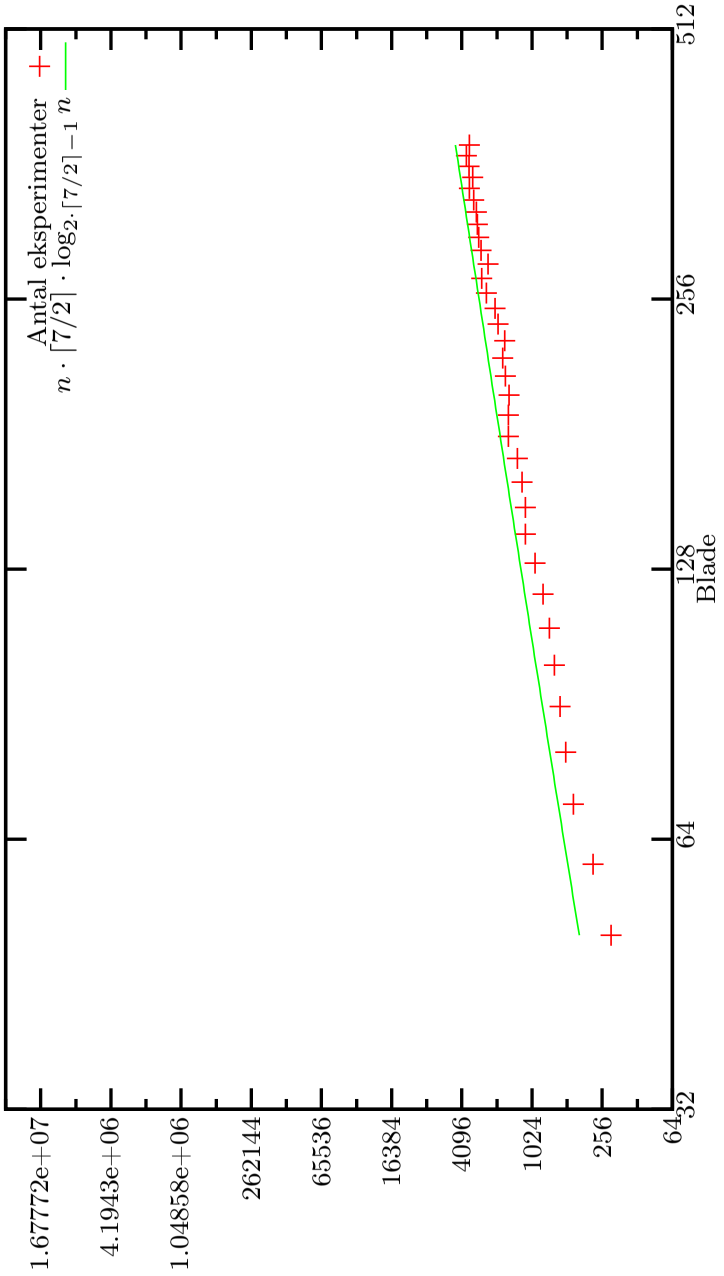
Figur C.3: Matricer, antal eksperimenter for $d = 3$



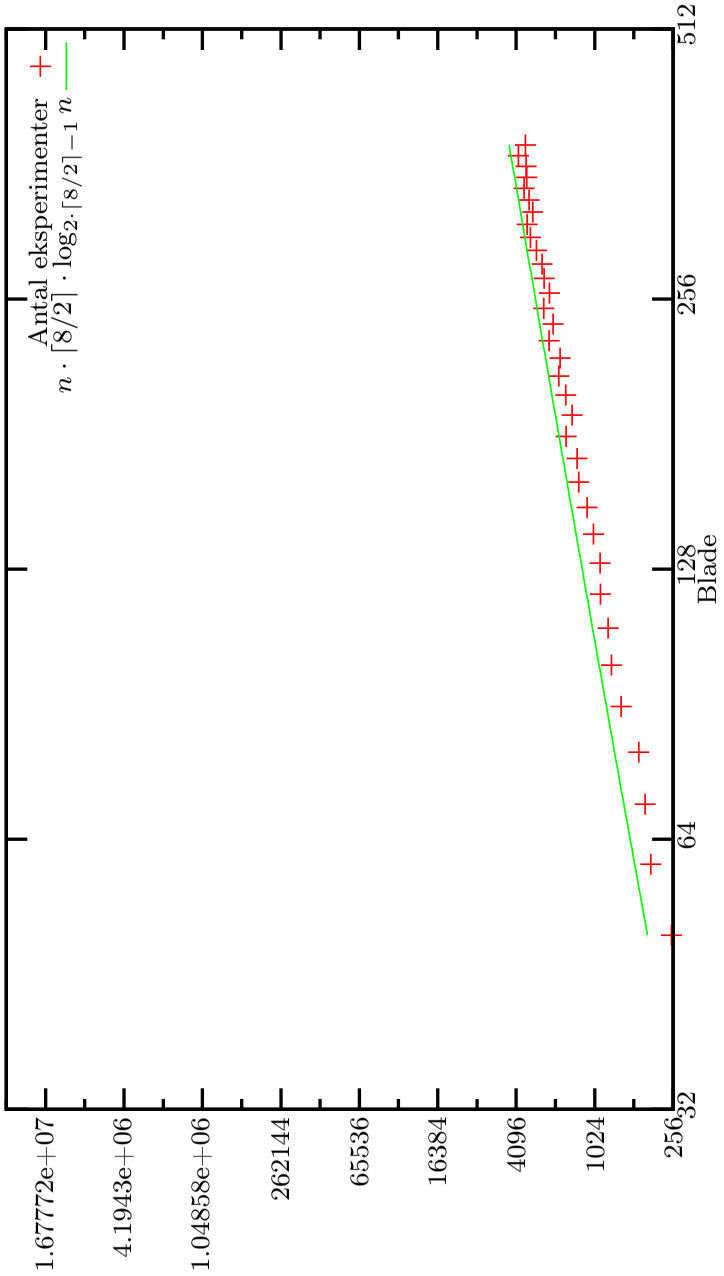
Figur C.4: Matricer, antal eksperimenter for $d = 4$



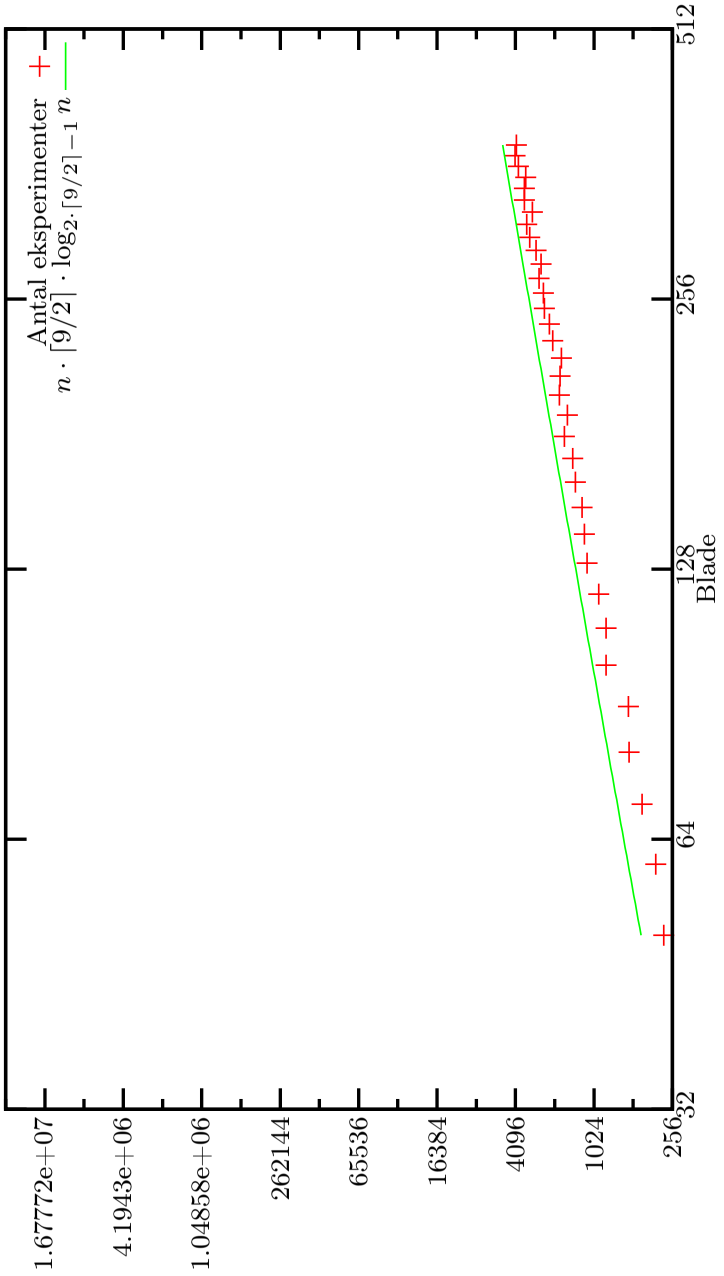
Figur C.5: Matricer, antal eksperimenter for $d = 5$



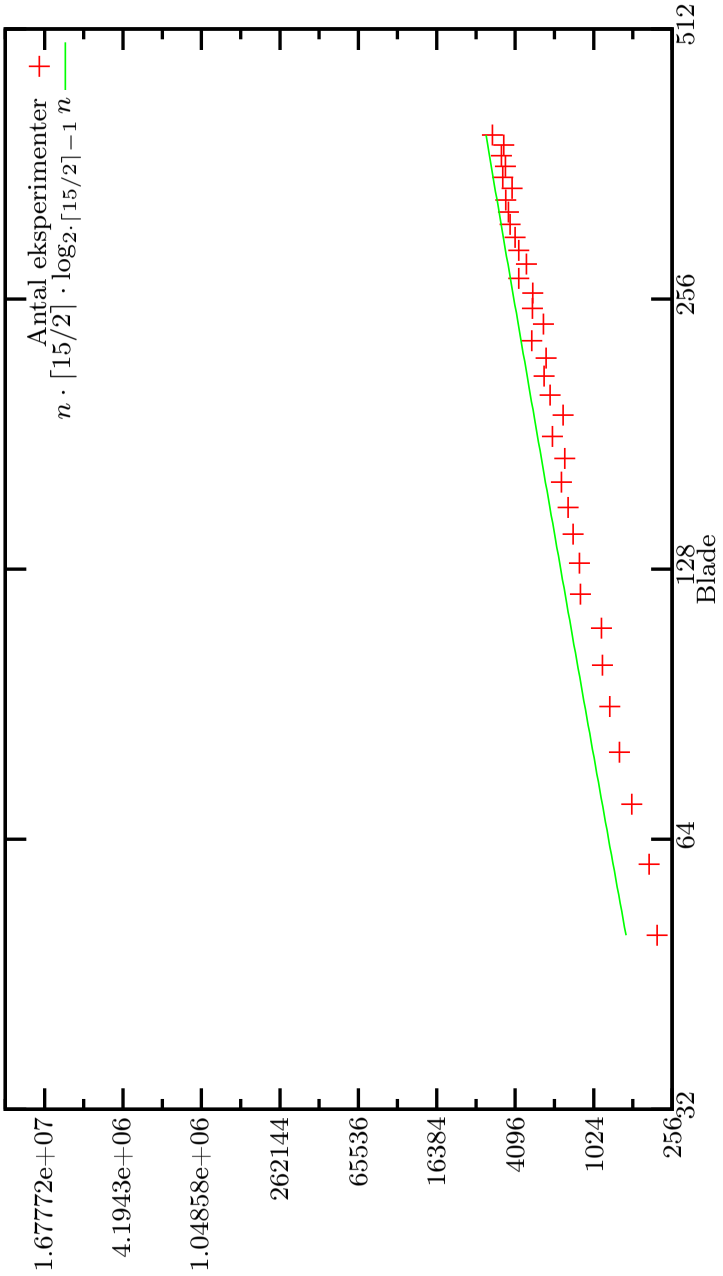
Figur C.6: Matricer, antal eksperimenter for $d = 7$



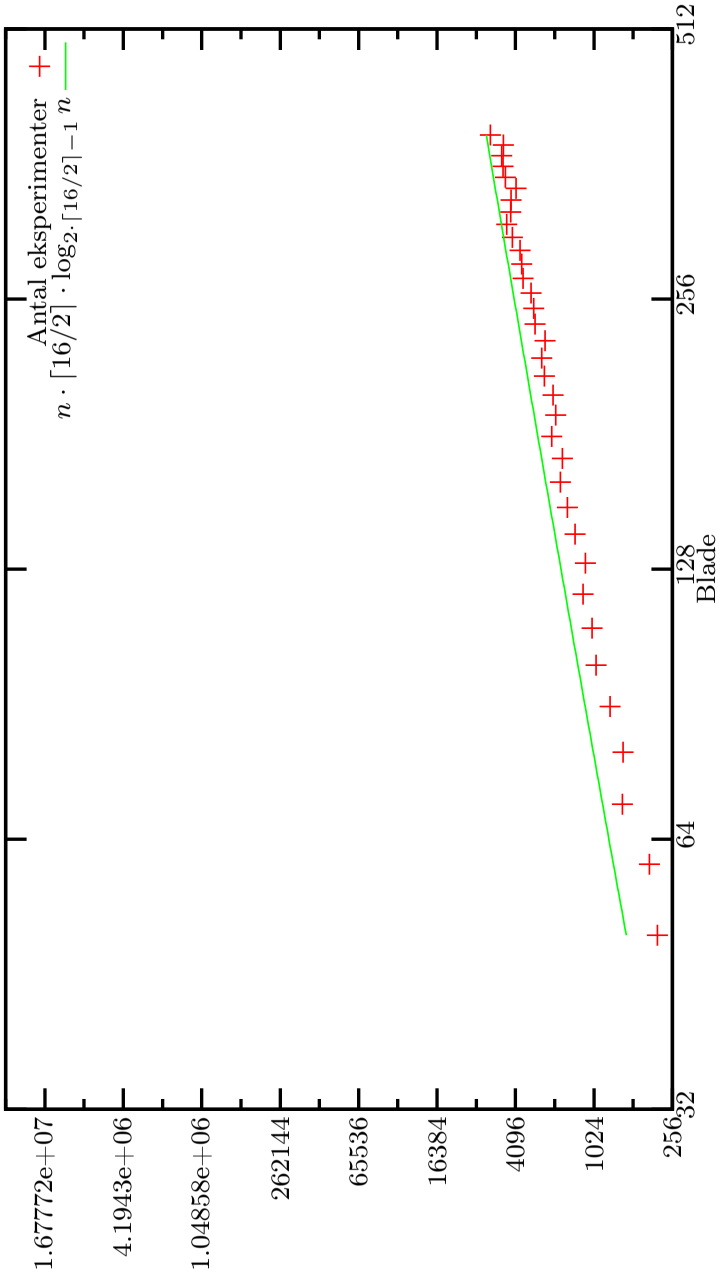
Figur C.7: Matricer, antal eksperimenter for $d = 8$



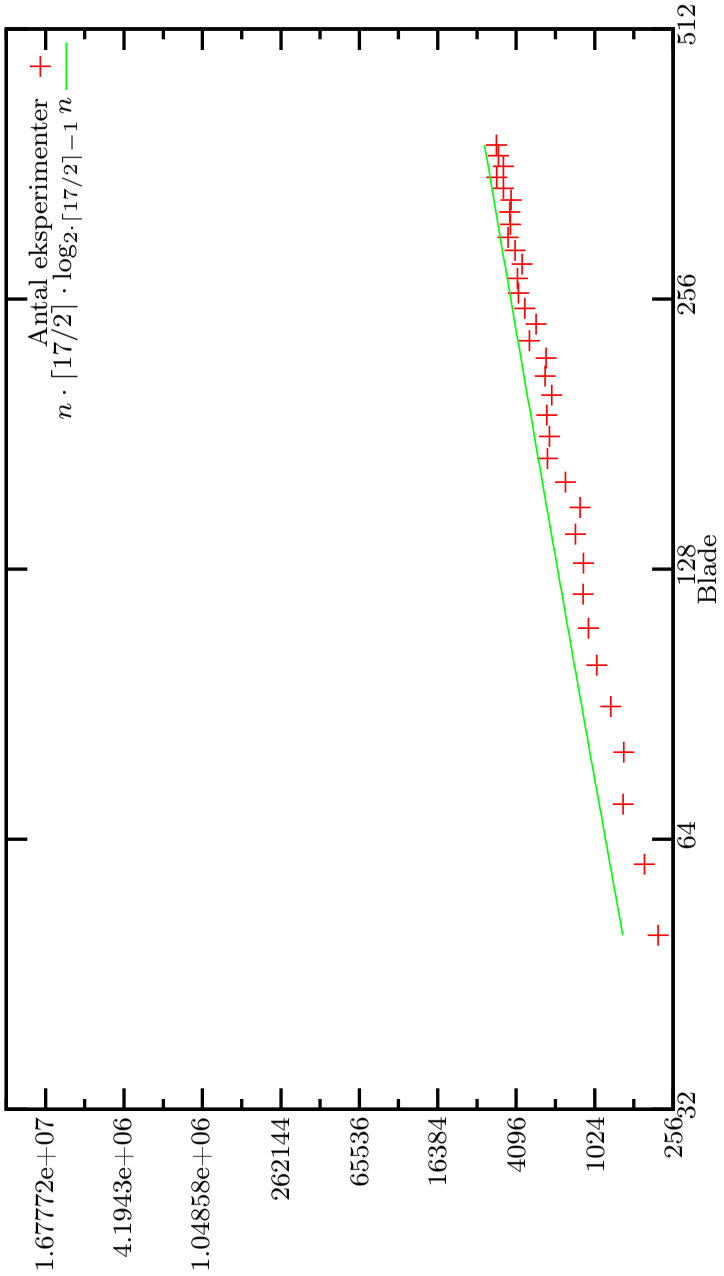
Figur C.8: Matricer, antal eksperimenter for $d = 9$



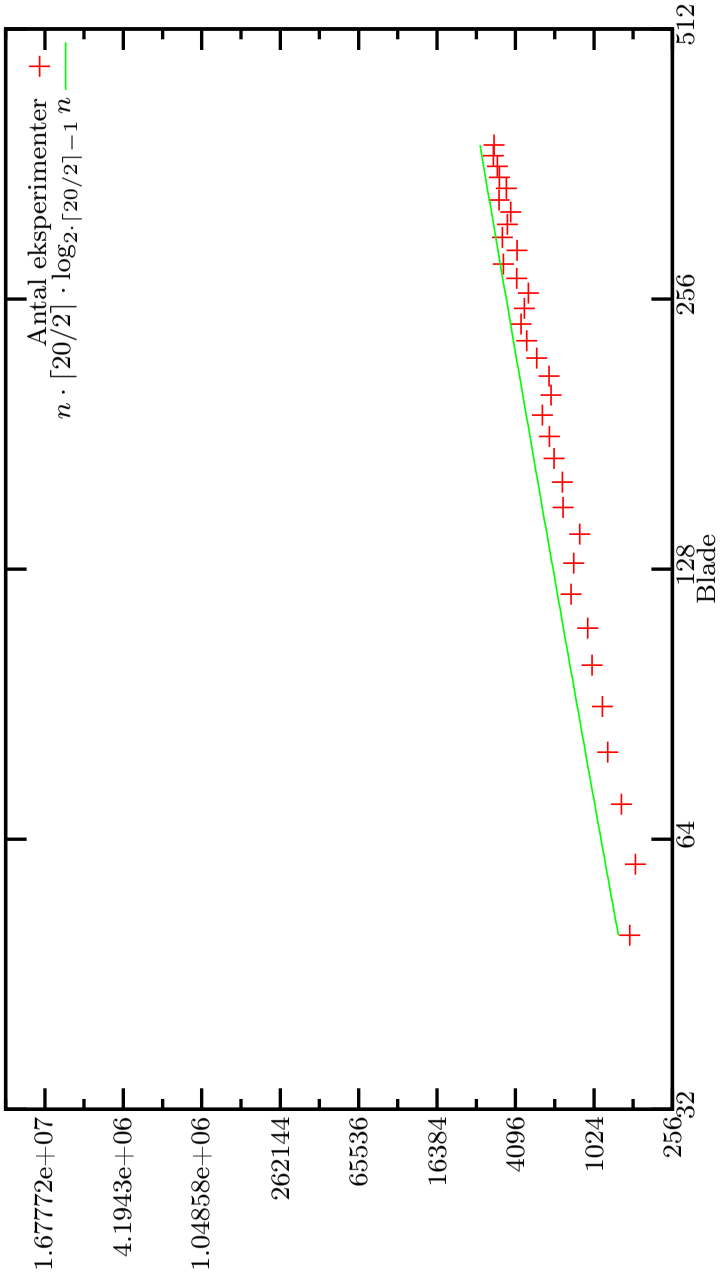
Figur C.9: Matricer, antal eksperimenter for $d = 15$



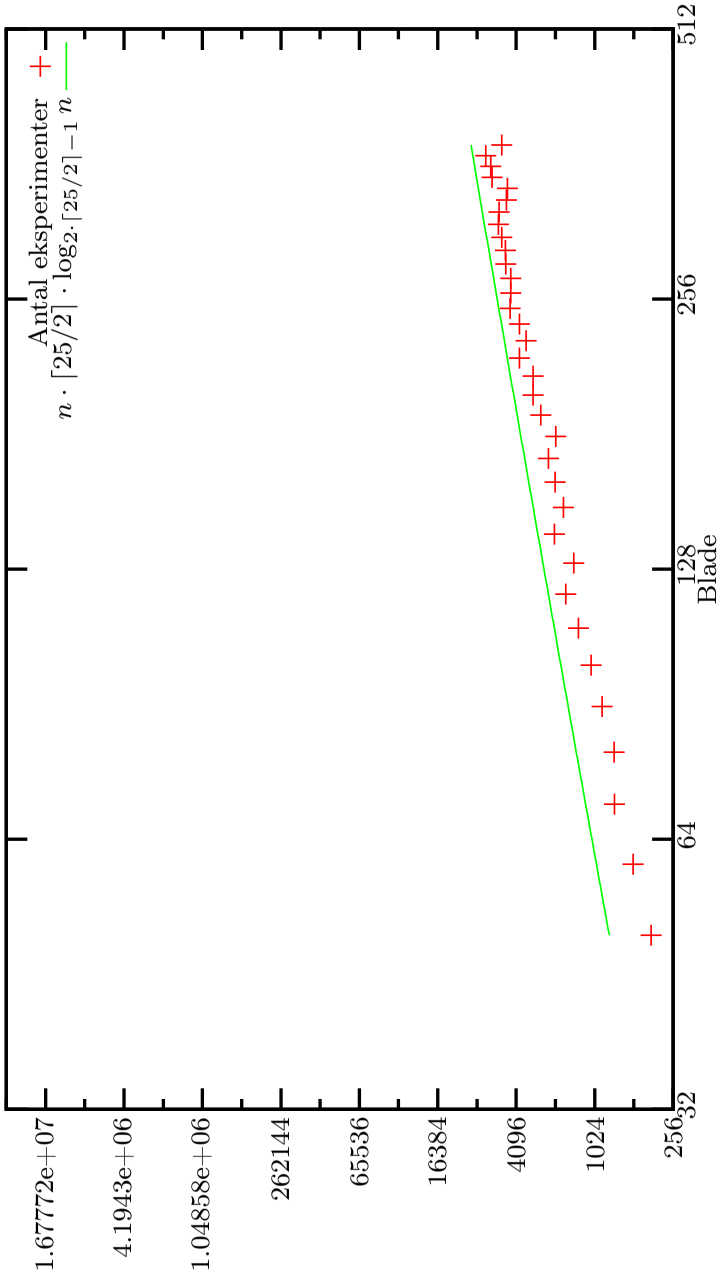
Figur C.10: Matricer, antal eksperimenter for $d = 16$



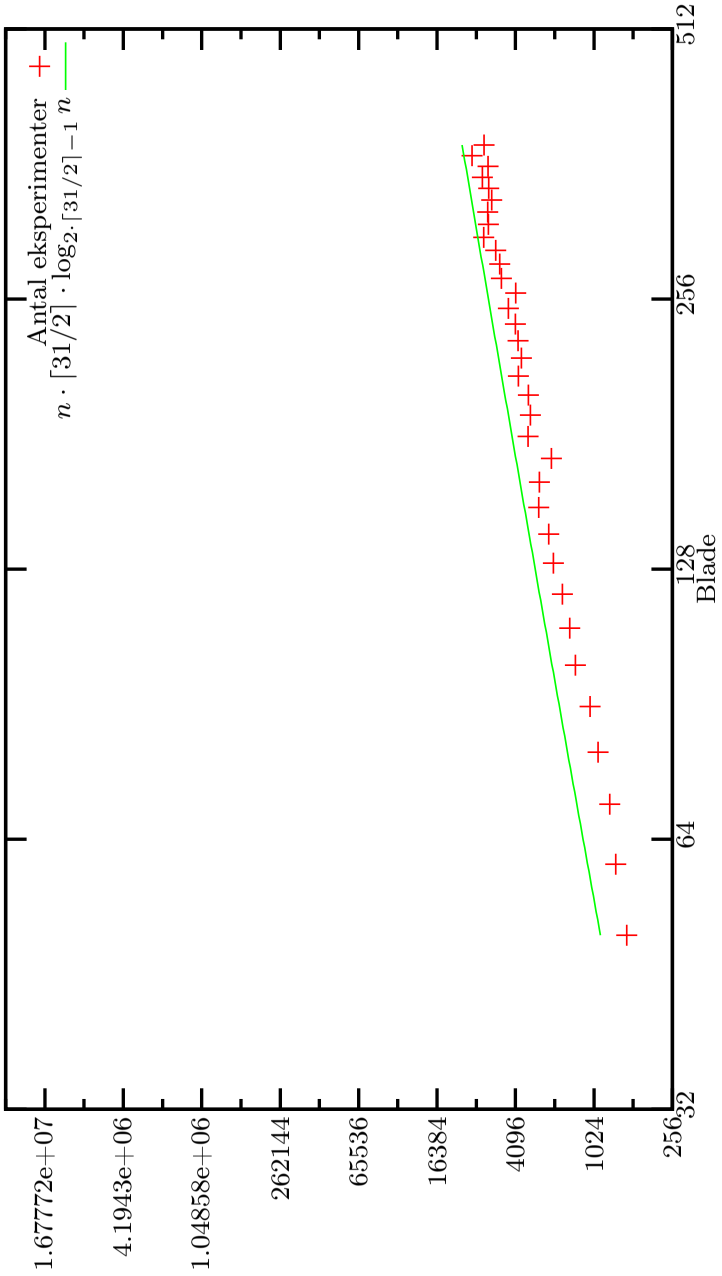
Figur C.11: Matricer, antal eksperimenter for $d = 17$



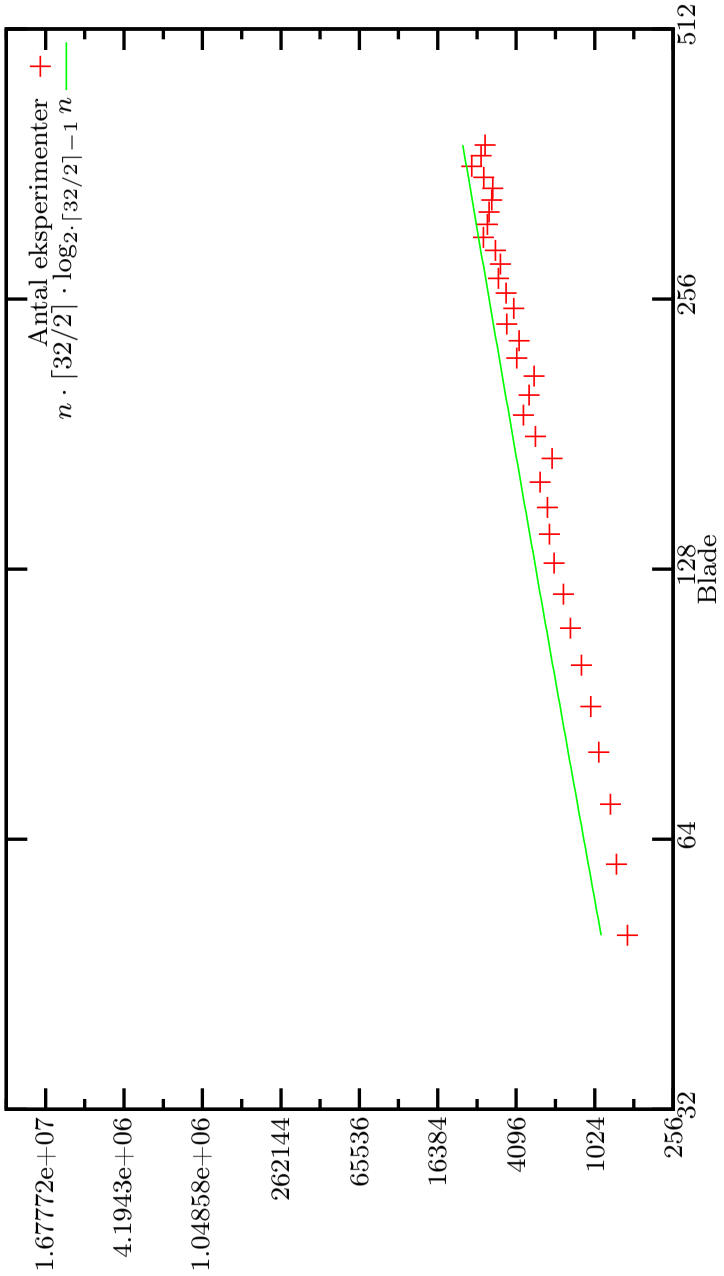
Figur C.12: Matricer, antal eksperimenter for $d = 20$



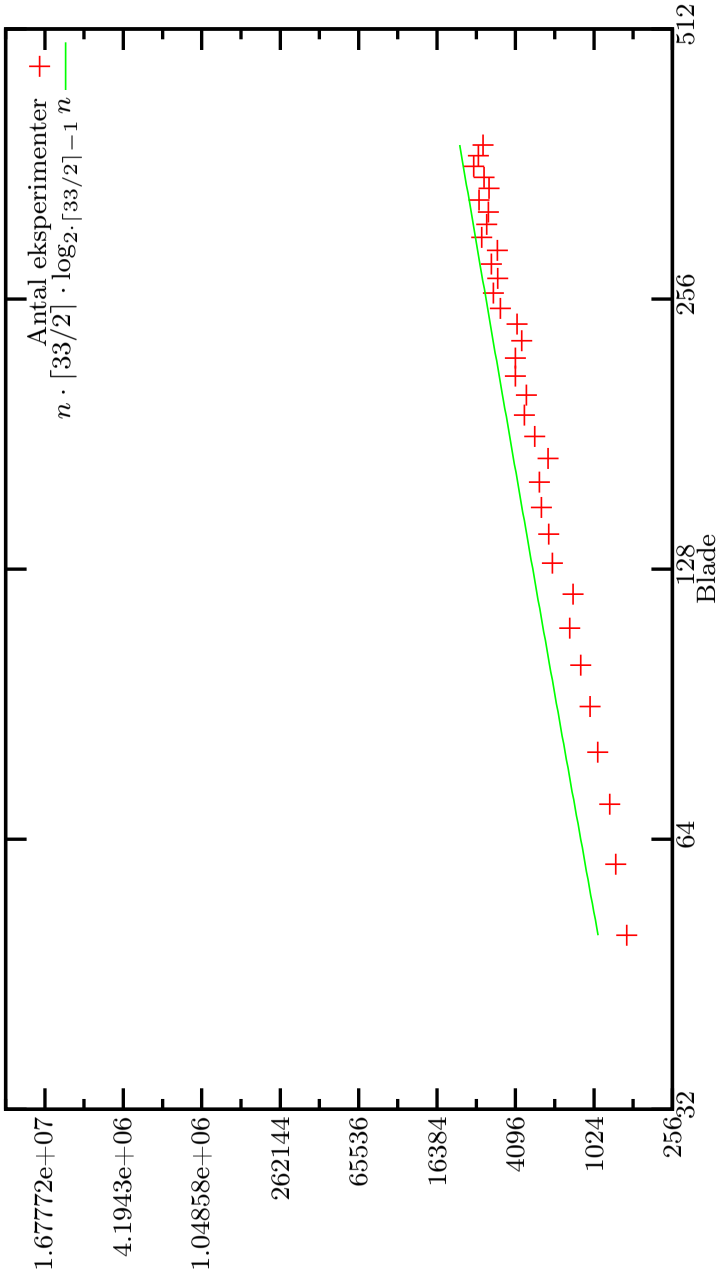
Figur C.13: Matricer, antal eksperimenter for $d = 25$



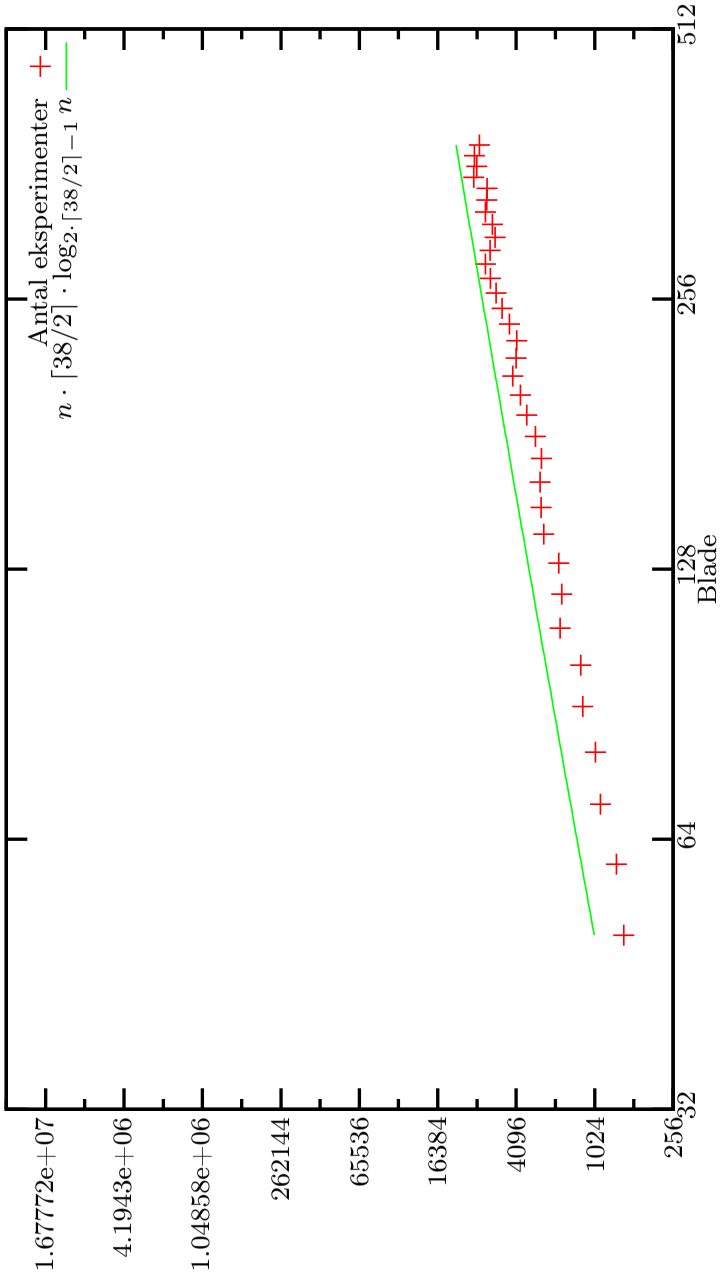
Figur C.14: Matricer, antal eksperimenter for $d = 31$



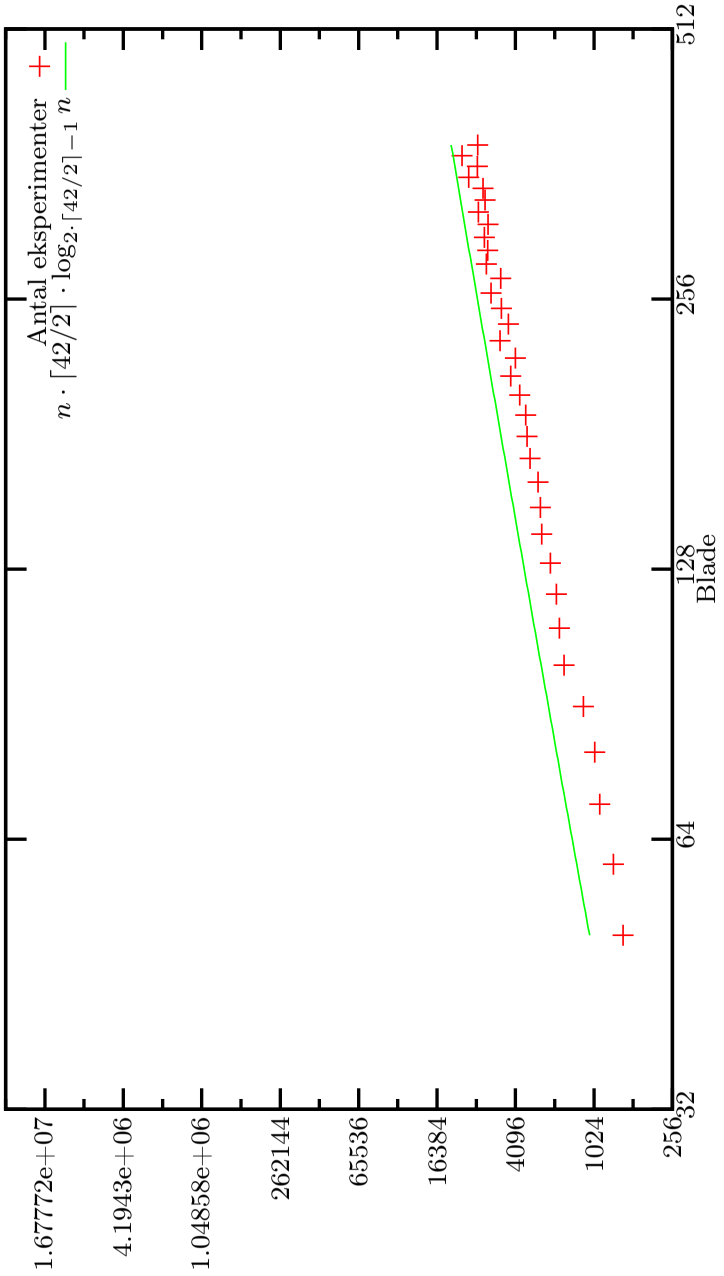
Figur C.15: Matricer, antal eksperimenter for $d = 32$



Figur C.16: Matricer, antal eksperimenter for $d = 33$

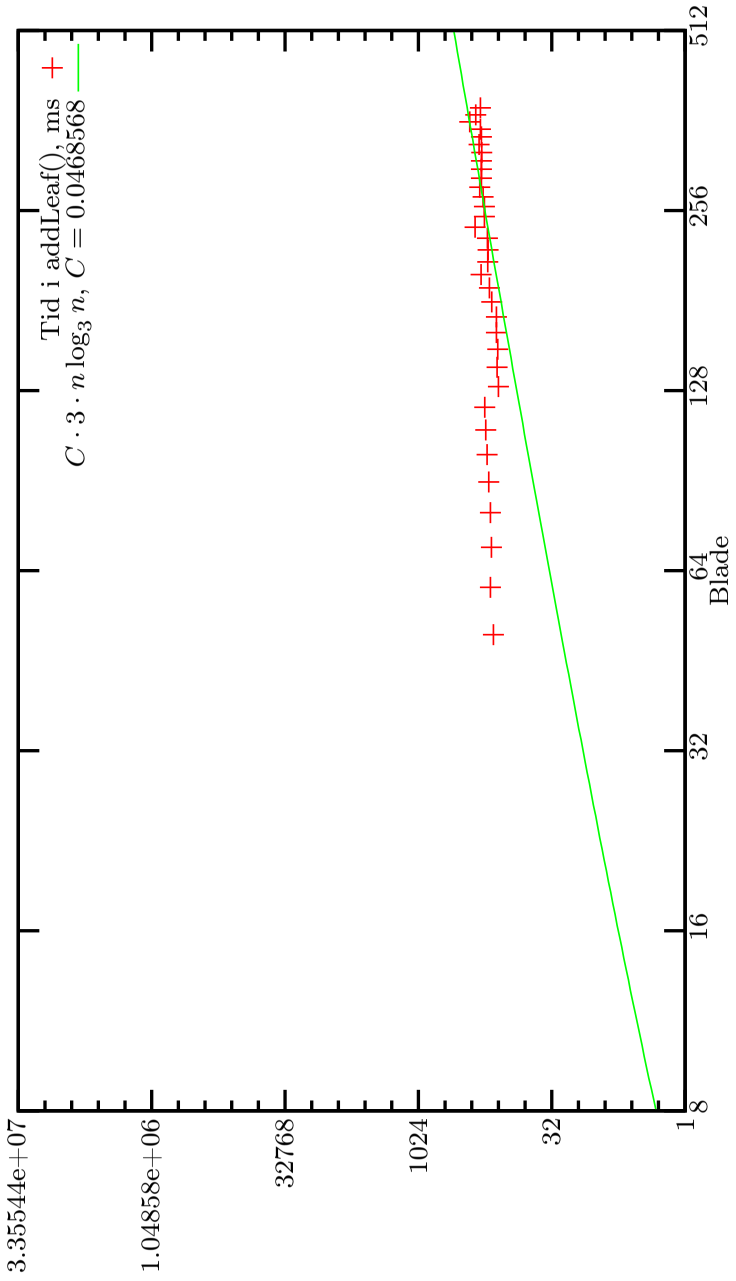


Figur C.17: Matricer, antal eksperimenter for $d = 38$

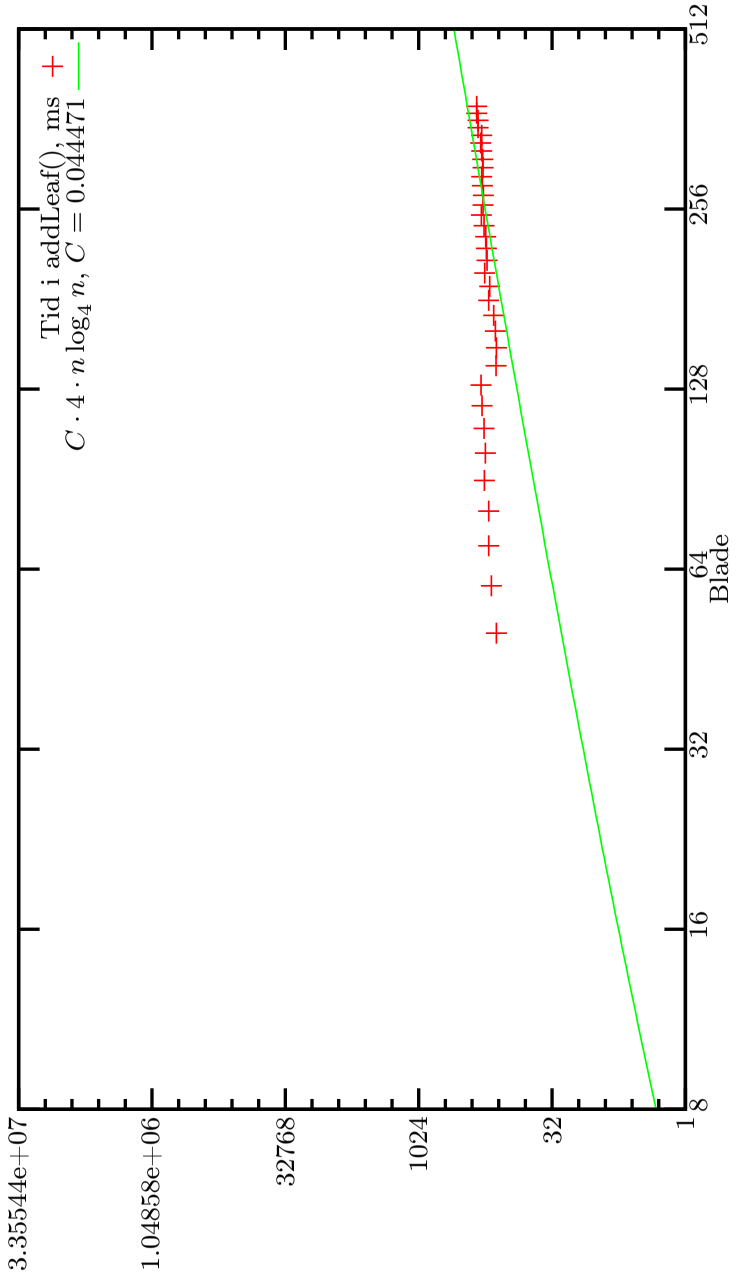


Figur C.18: Matricer, antal eksperimenter for $d = 42$

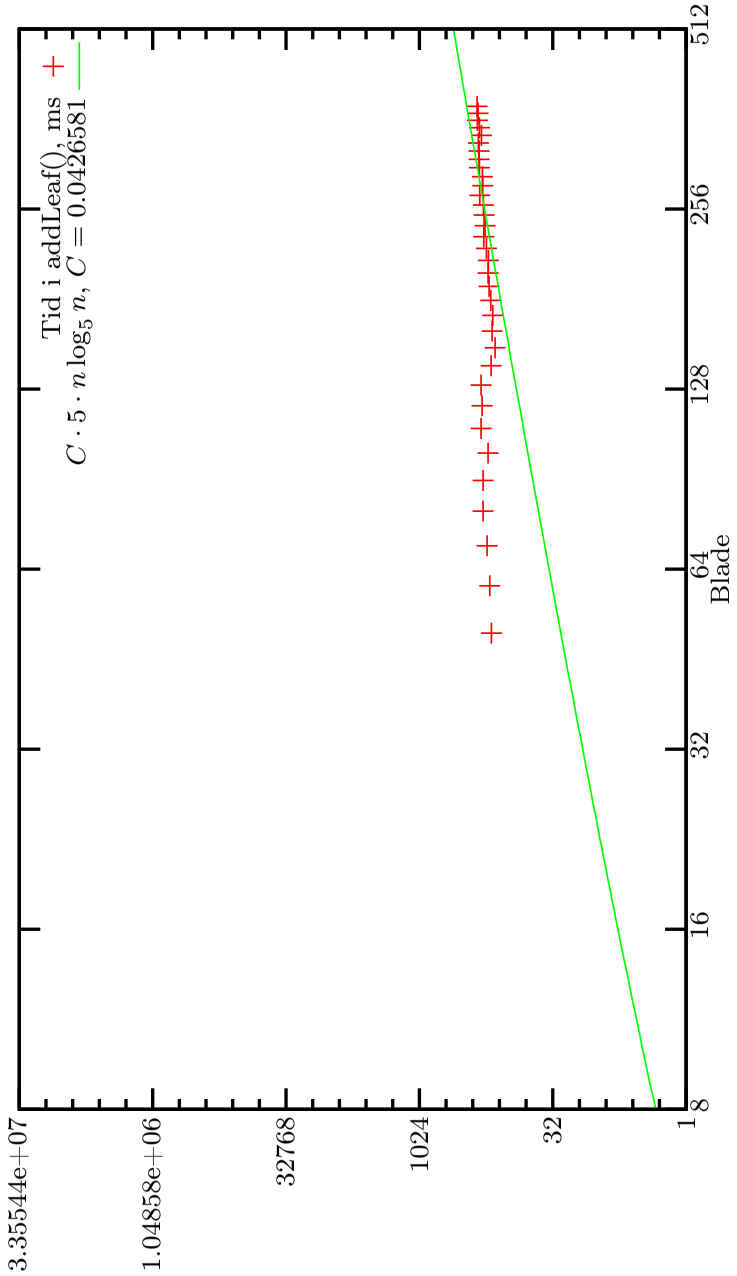
C.2.2 Tidsforbrug



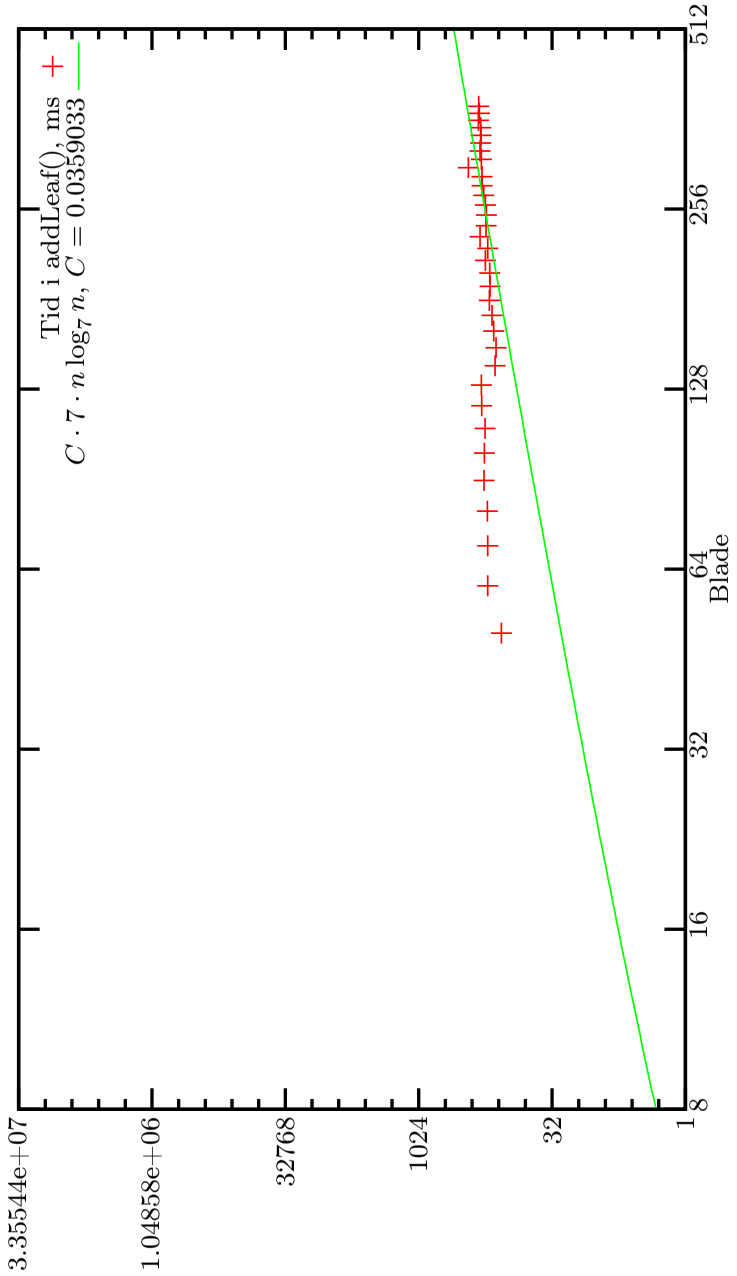
Figur C.19: Matricer, tidsforbrug for $d = 3$



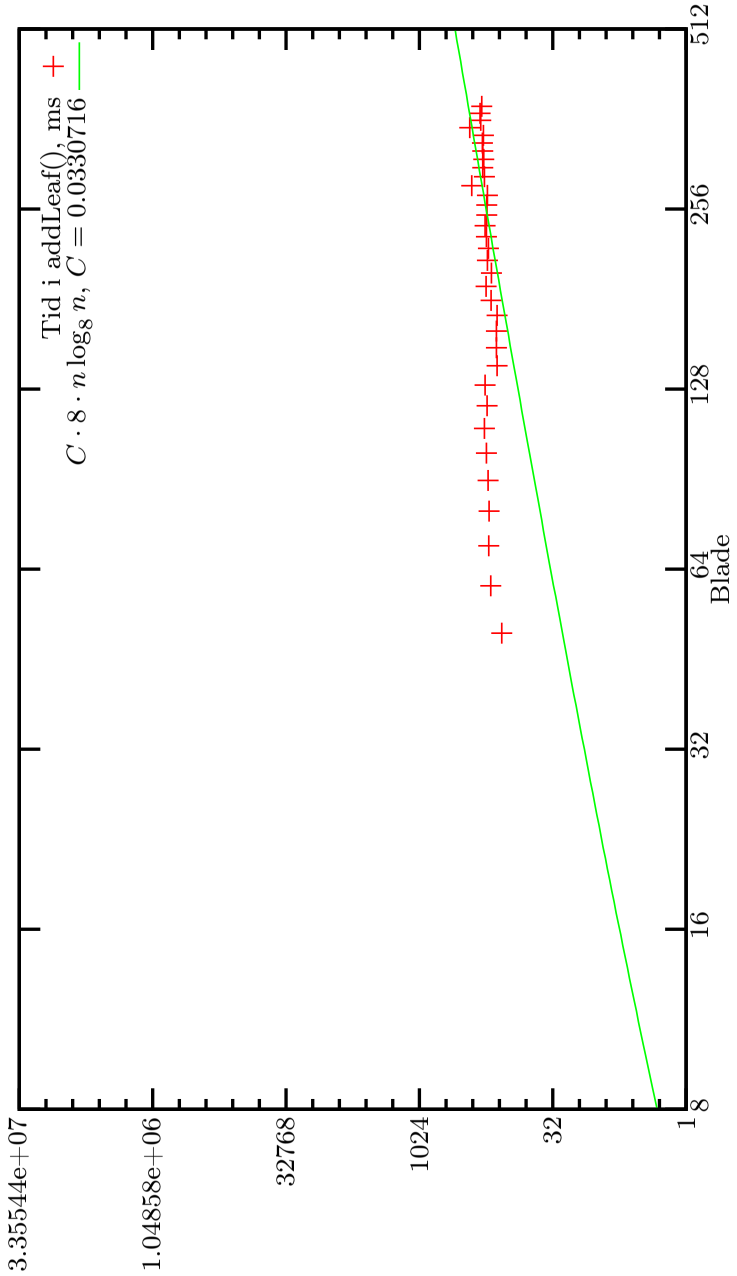
Figur C.20: Matricer, tidsforbrug for $d = 4$



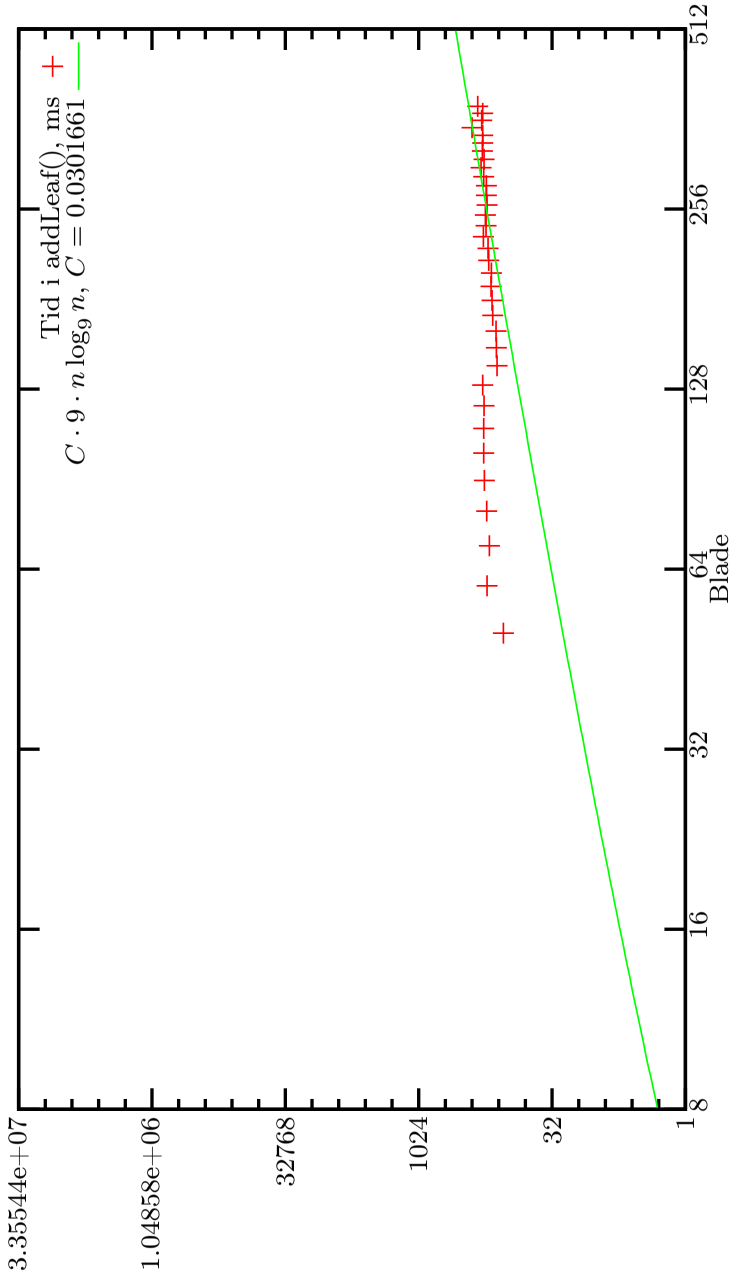
Figur C.21: Matricer, tidsforbrug for $d = 5$



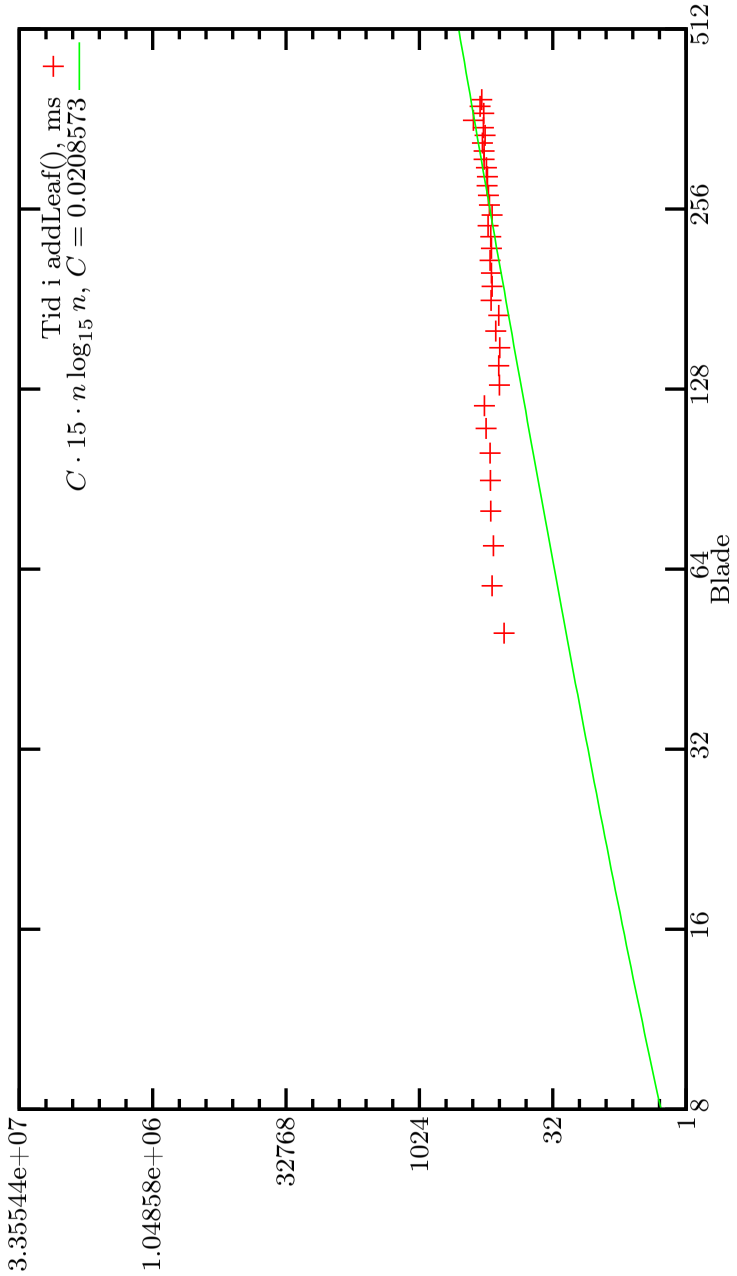
Figur C.22: Matricer, tidsforbrug for $d = 7$



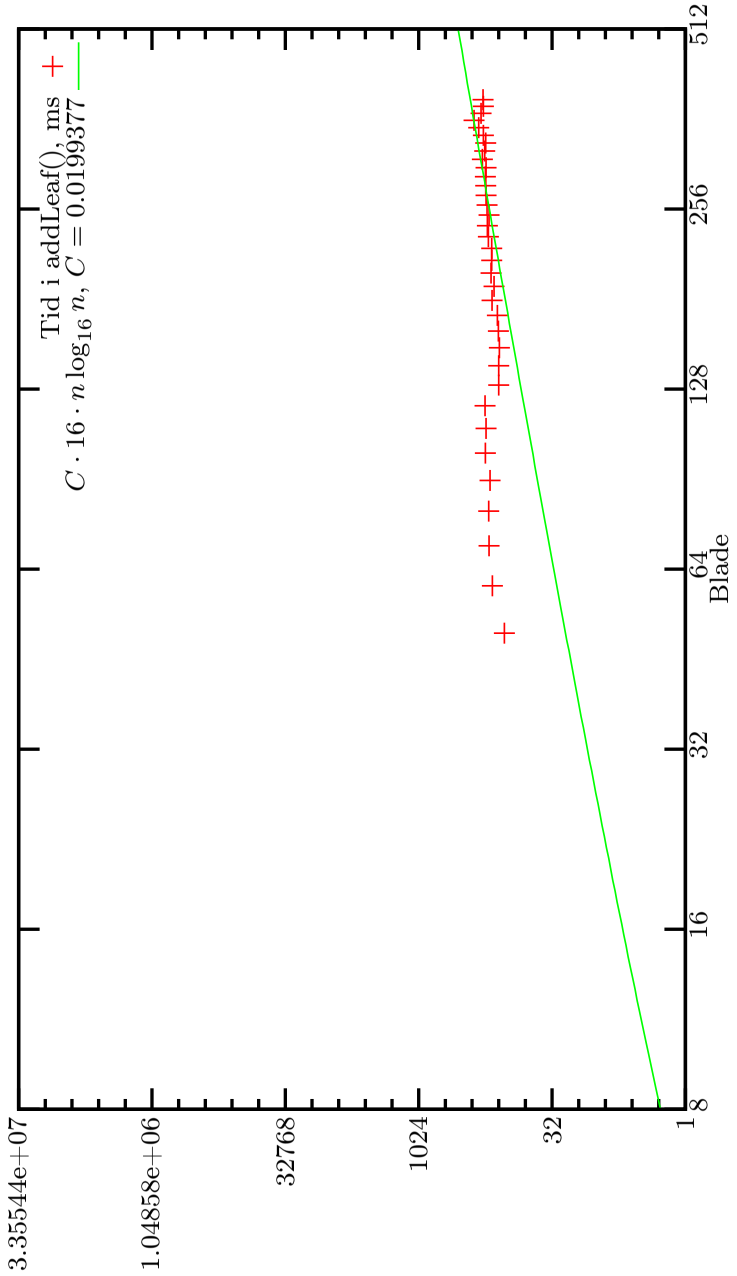
Figur C.23: Matricer, tidsforbrug for $d = 8$



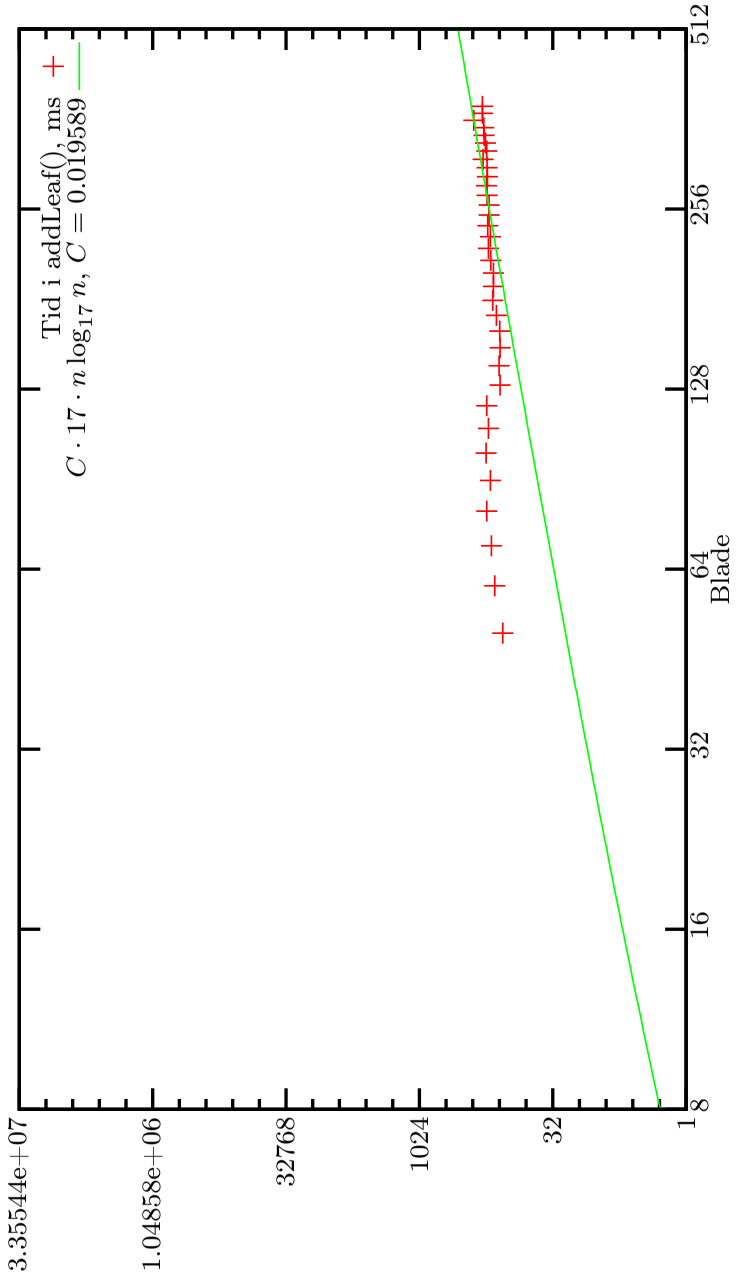
Figur C.24: Matricer, tidsforbrug for $d = 9$



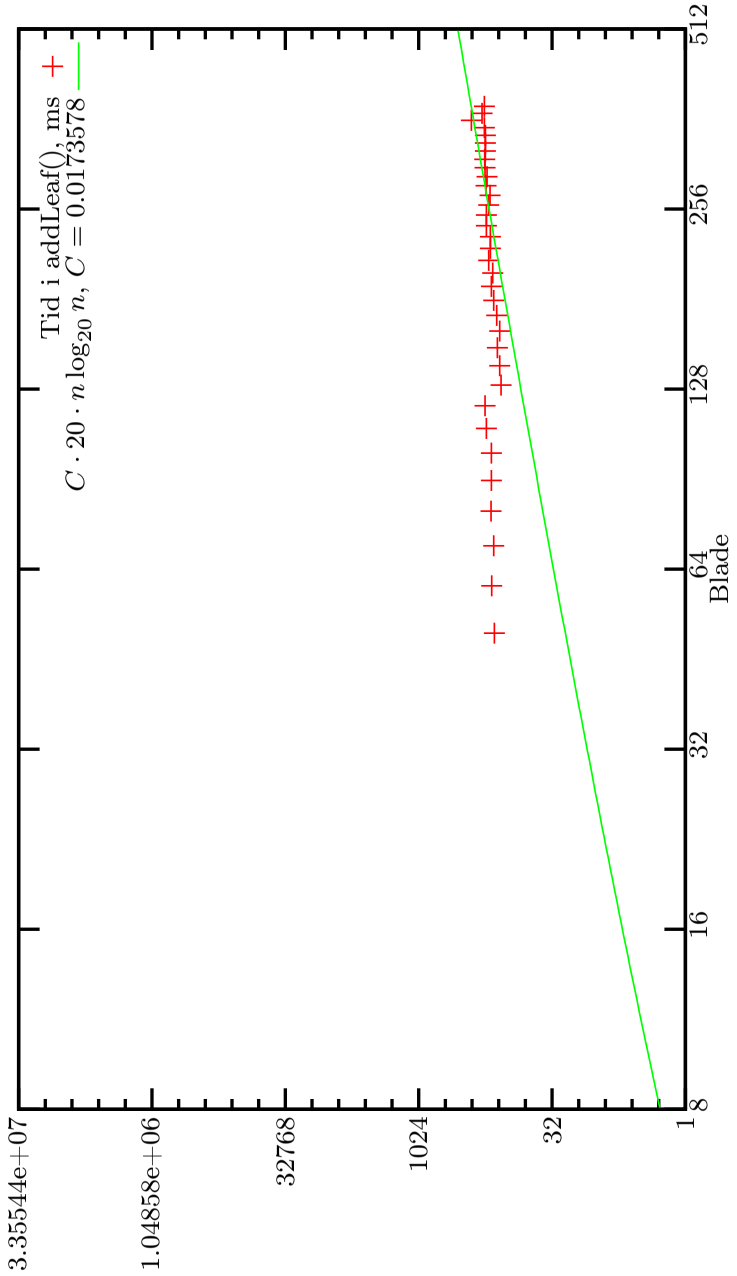
Figur C.25: Matricer, tidsforbrug for $d = 15$



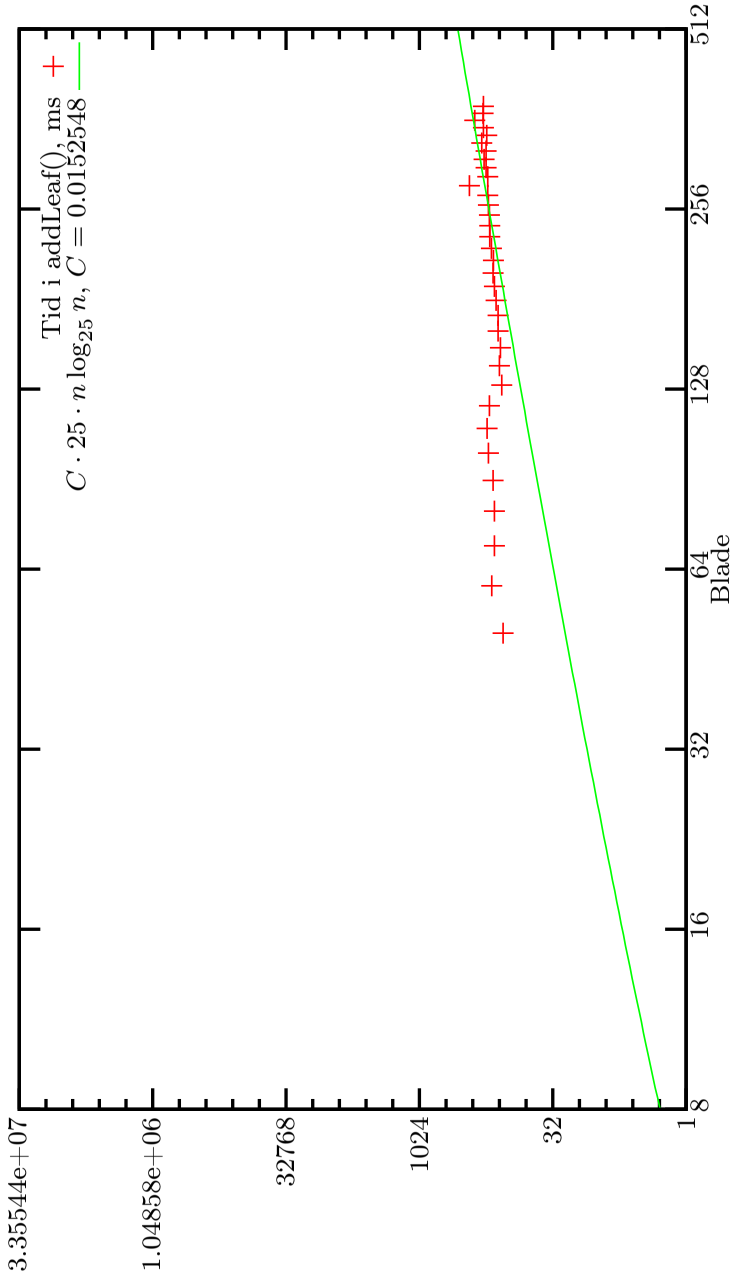
Figur C.26: Matricer, tidsforbrug for $d = 16$



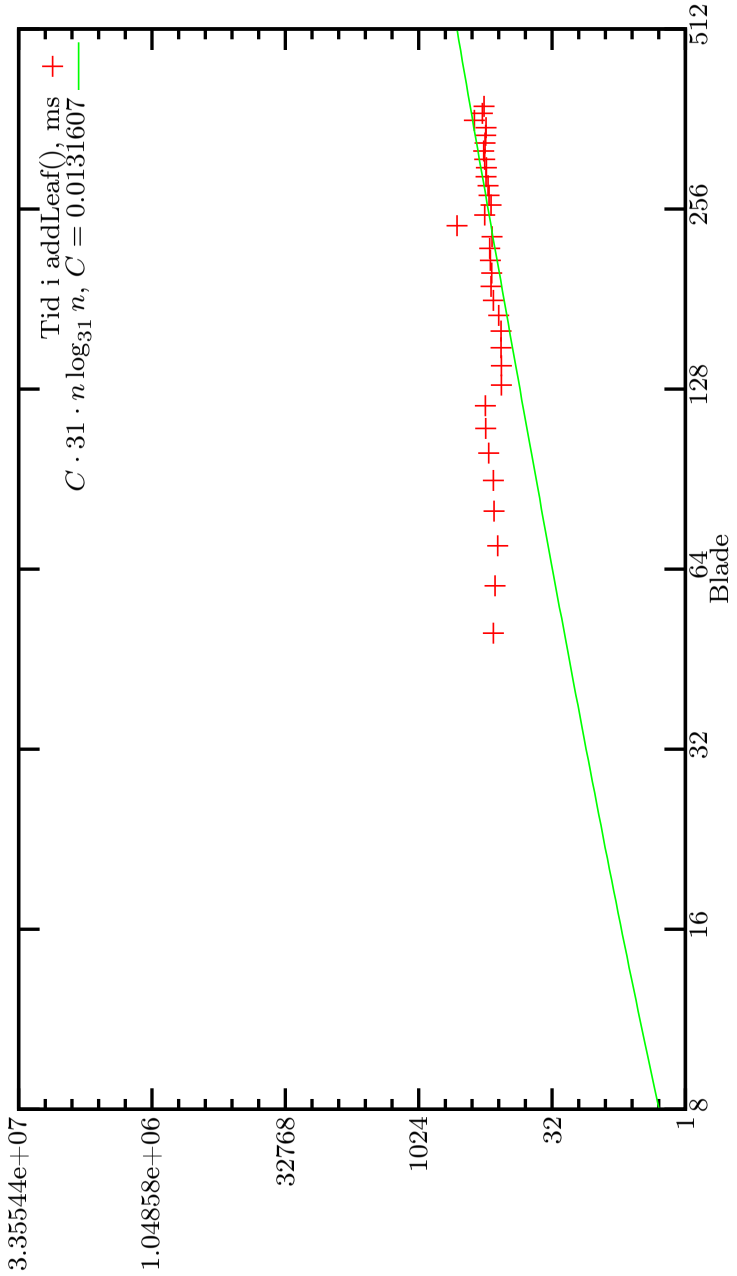
Figur C.27: Matricer, tidsforbrug for $d = 17$



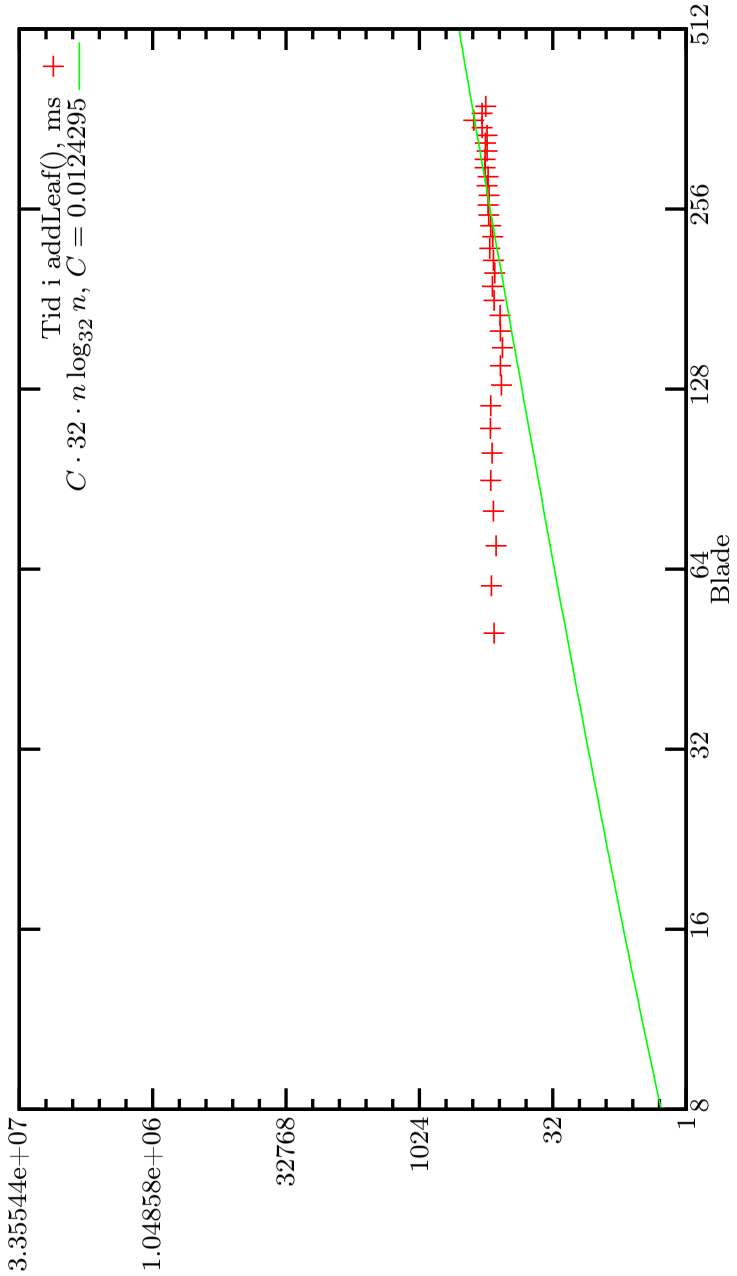
Figur C.28: Matricer, tidsforbrug for $d = 20$



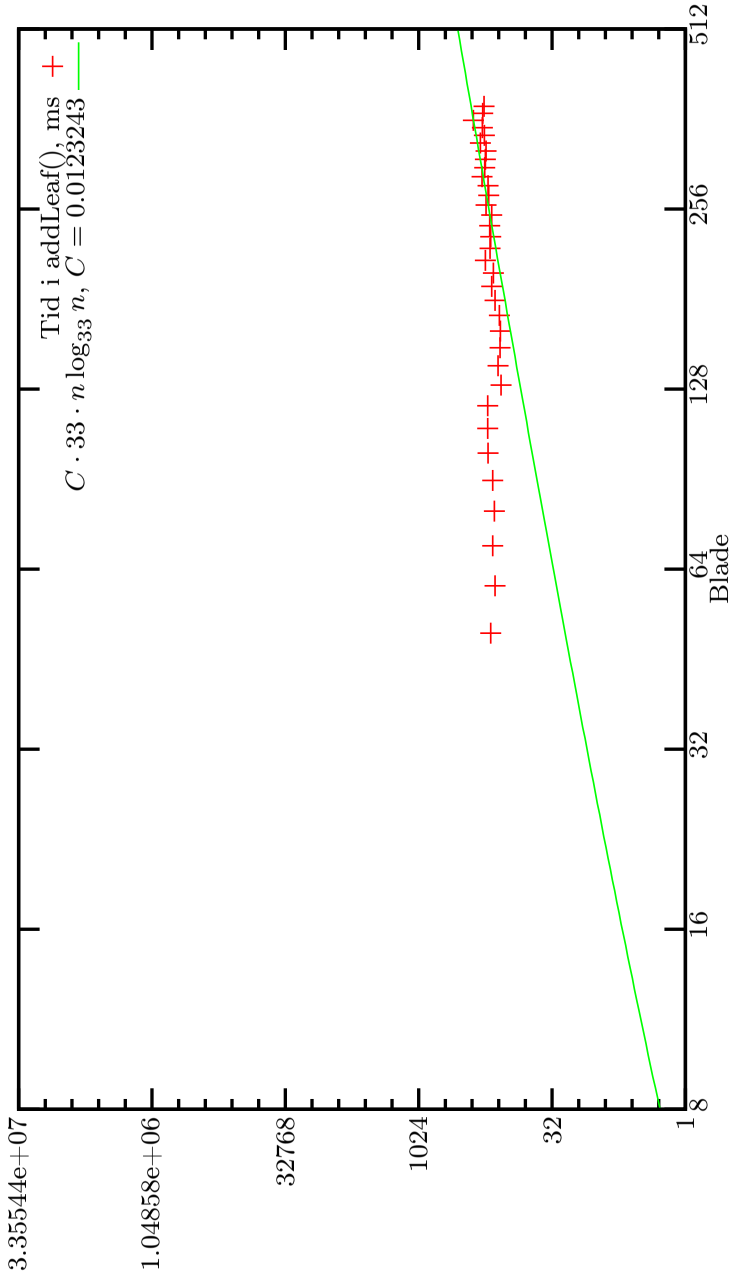
Figur C.29: Matricer, tidsforbrug for $d = 25$



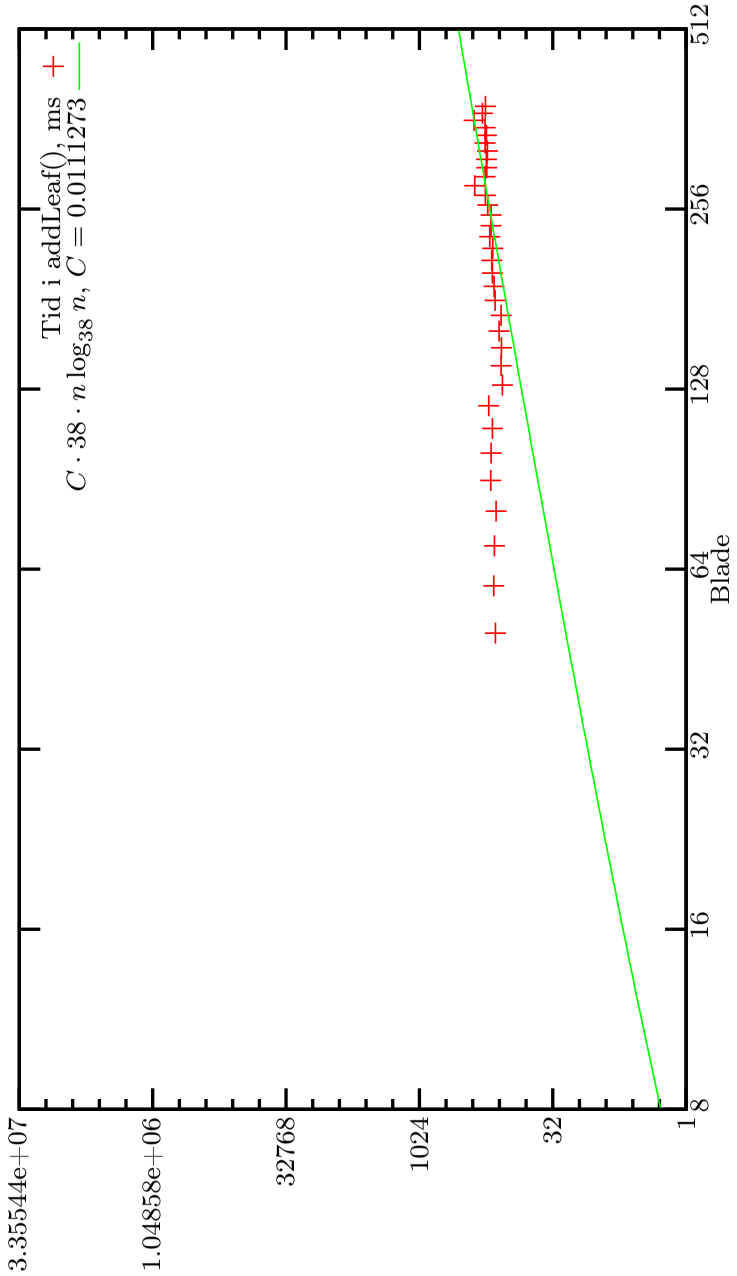
Figur C.30: Matricer, tidsforbrug for $d = 31$



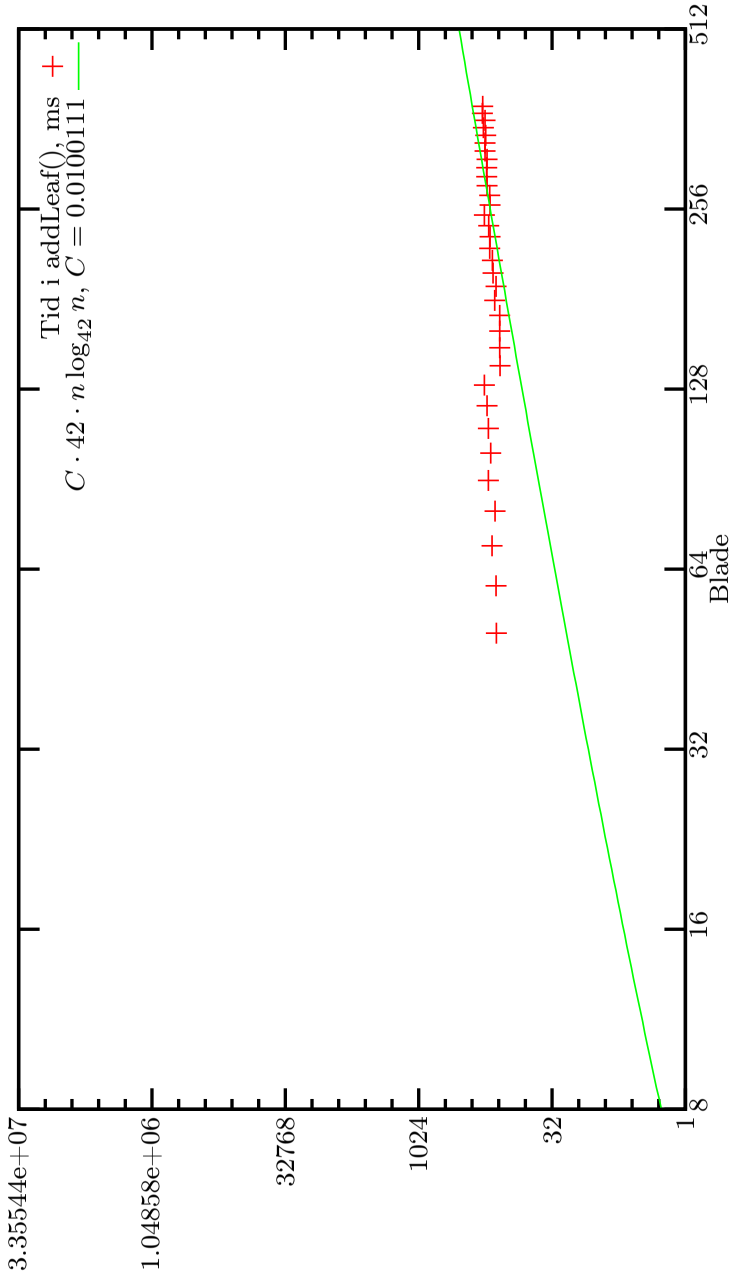
Figur C.31: Matricer, tidsforbrug for $d = 32$



Figur C.32: Matricer, tidsforbrug for $d = 33$



Figur C.33: Matricer, tidsforbrug for $d = 38$

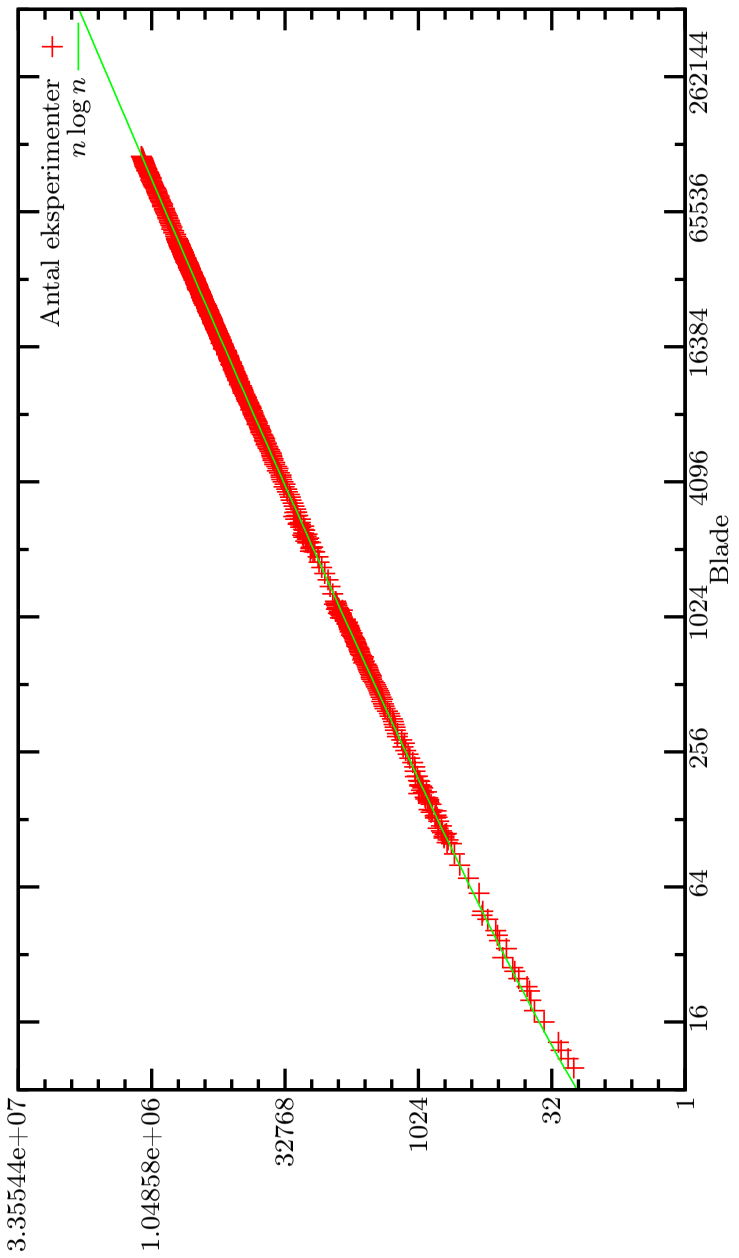


Figur C.34: Matricer, tidsforbrug for $d = 42$

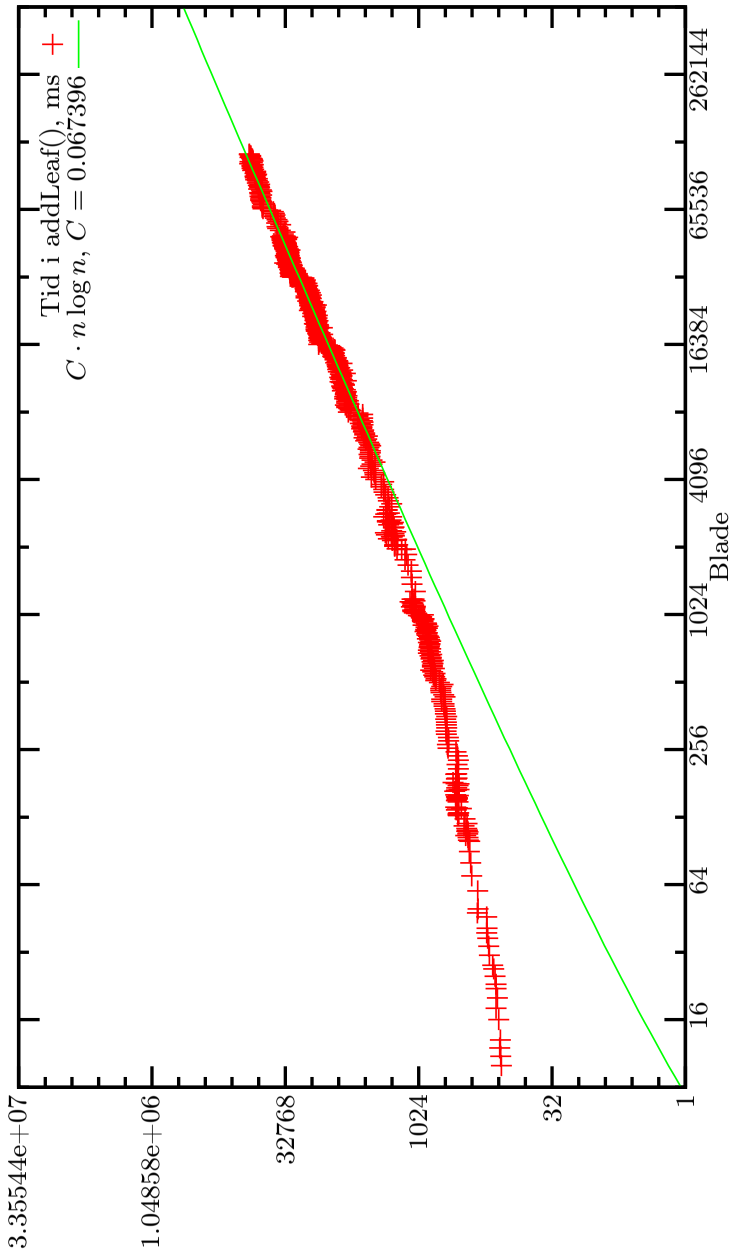
Bilag D

Eksperimenter på træer

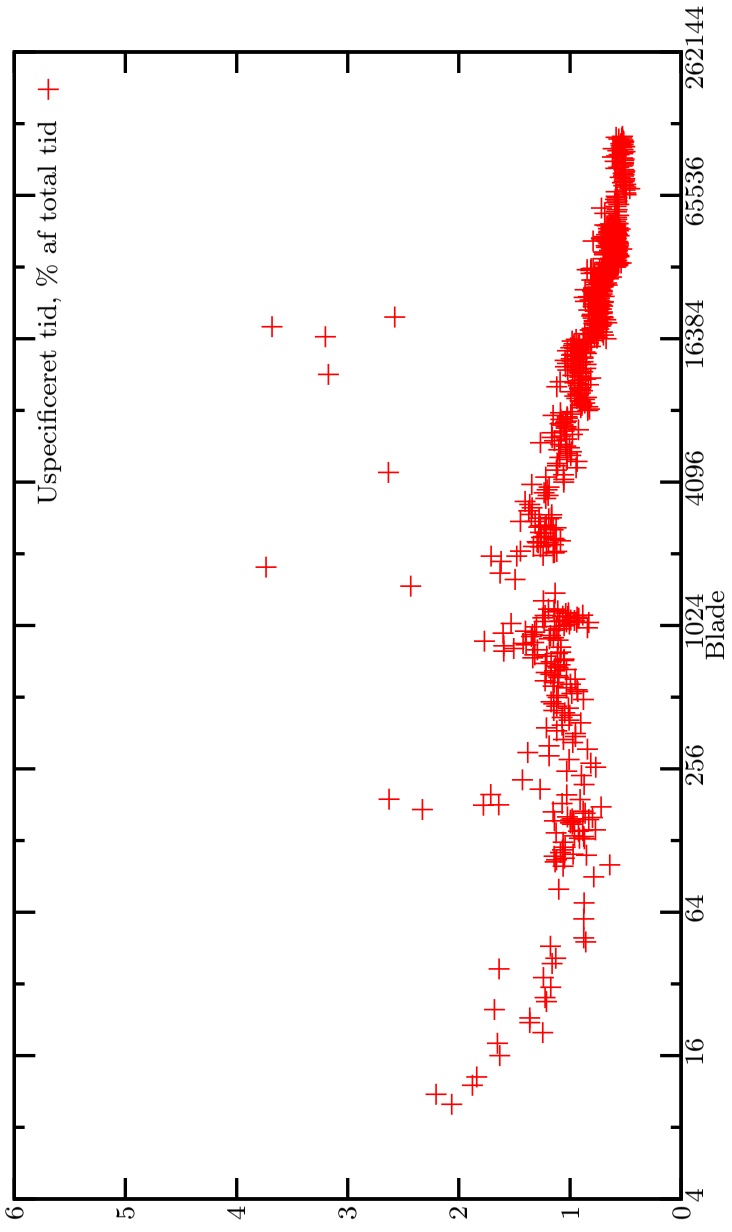
D.1 Binære træer



Figur D.1: Antal eksperimenter for $d = 2$



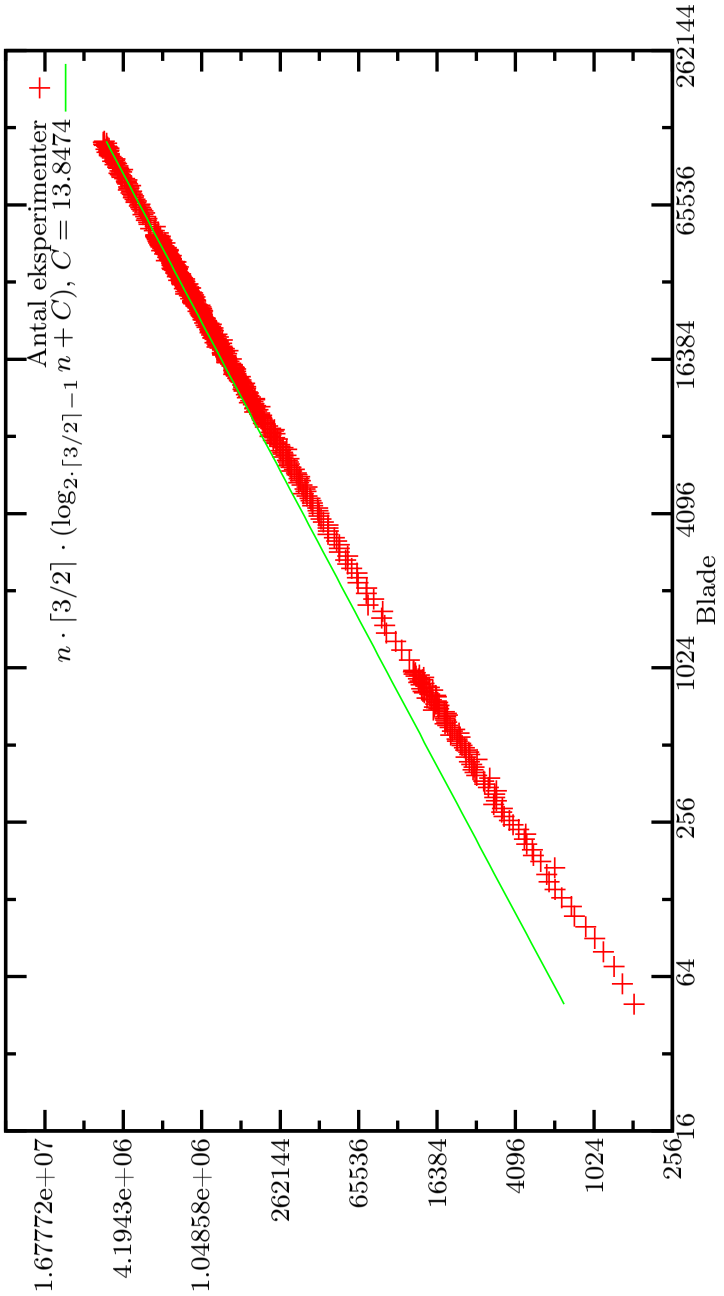
Figur D.2: Tidsforbrug for $d = 2$



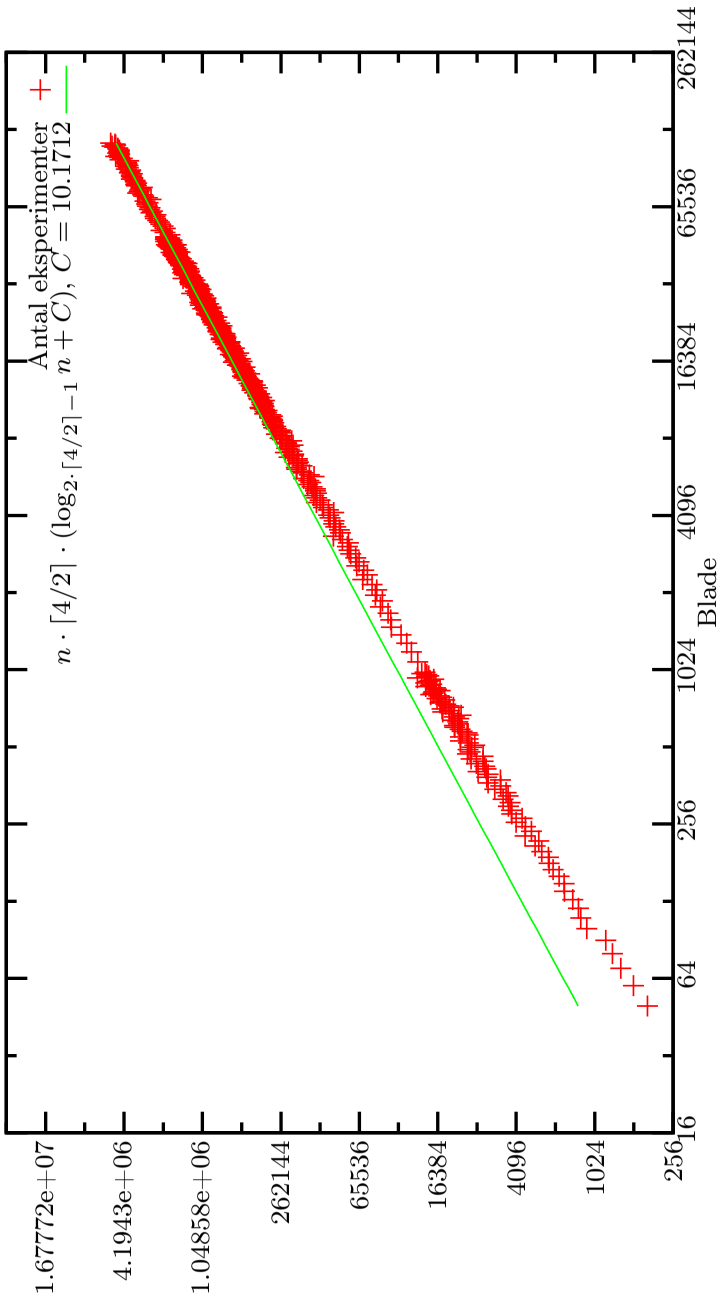
Figur D.3: Uspecificeret tidsforbrug, $d = 2$

D.2 Træer med højere udgrad

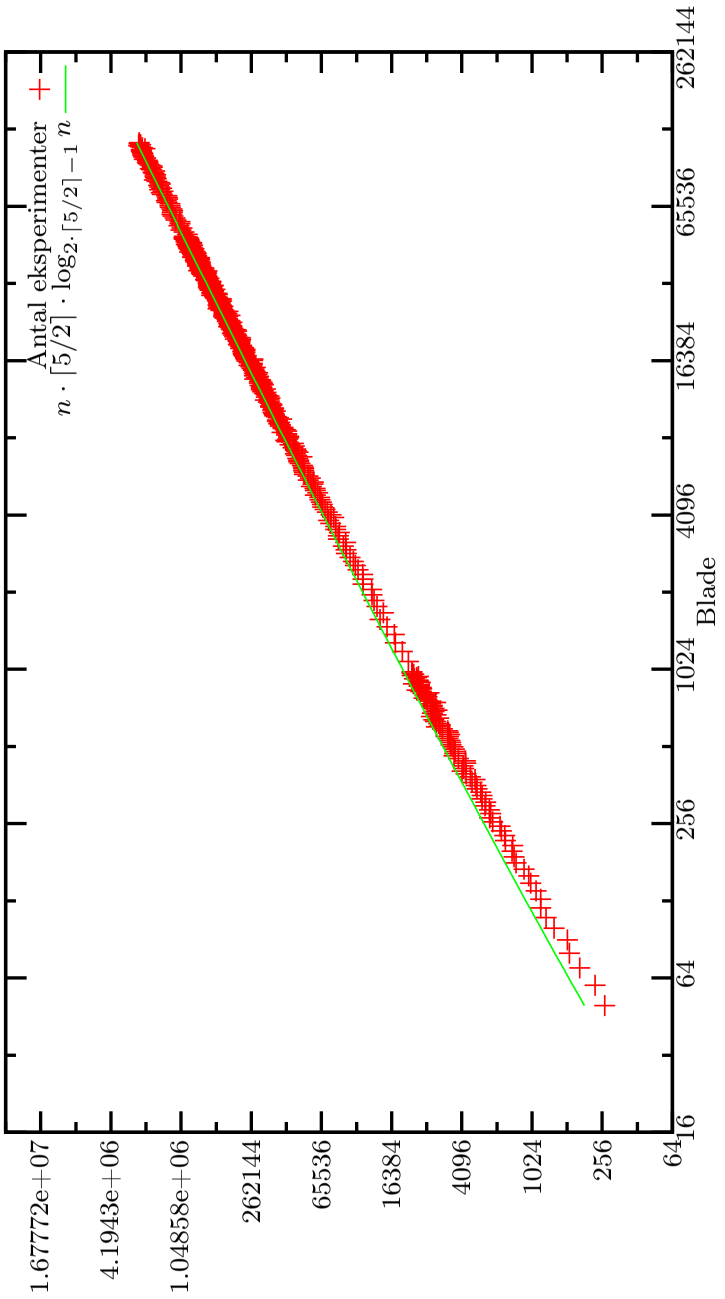
D.2.1 Antal eksperimenter



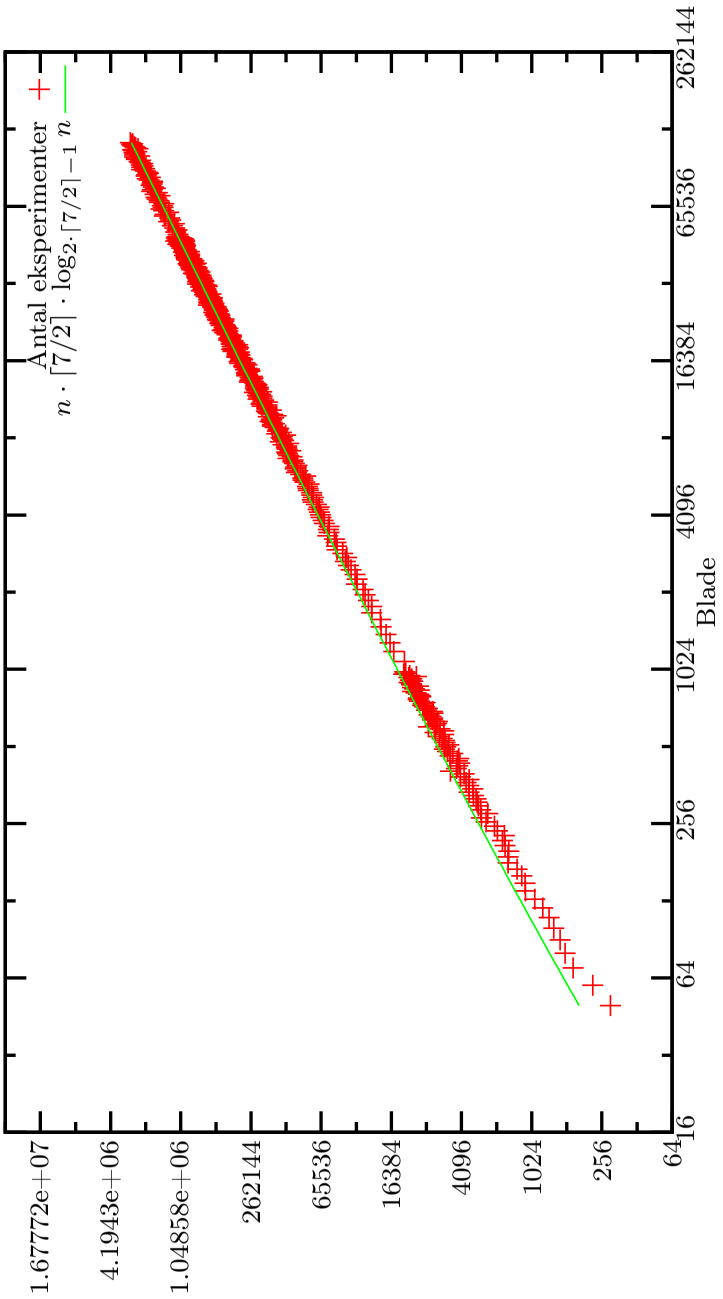
Figur D.4: Træer, antal eksperimenter for $d = 3$



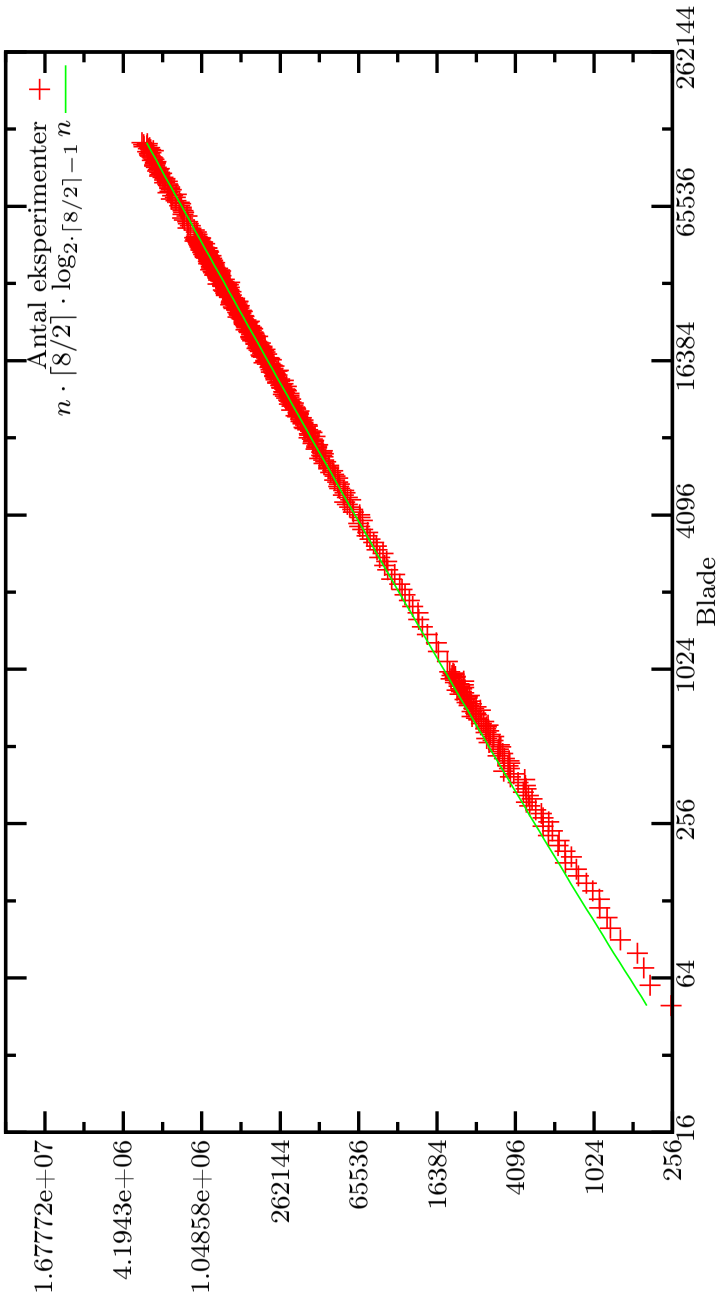
Figur D.5: Træer, antal eksperimenter for $d = 4$



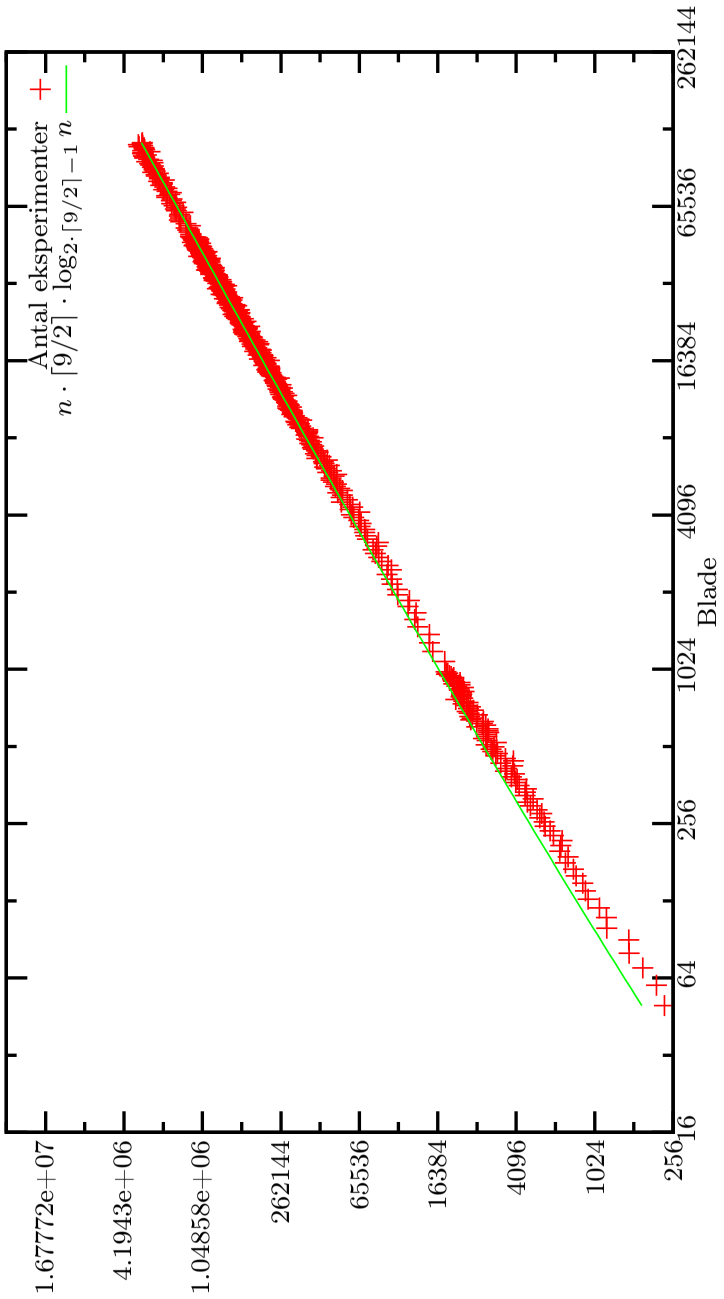
Figur D.6: Træer, antal eksperimenter for $d = 5$



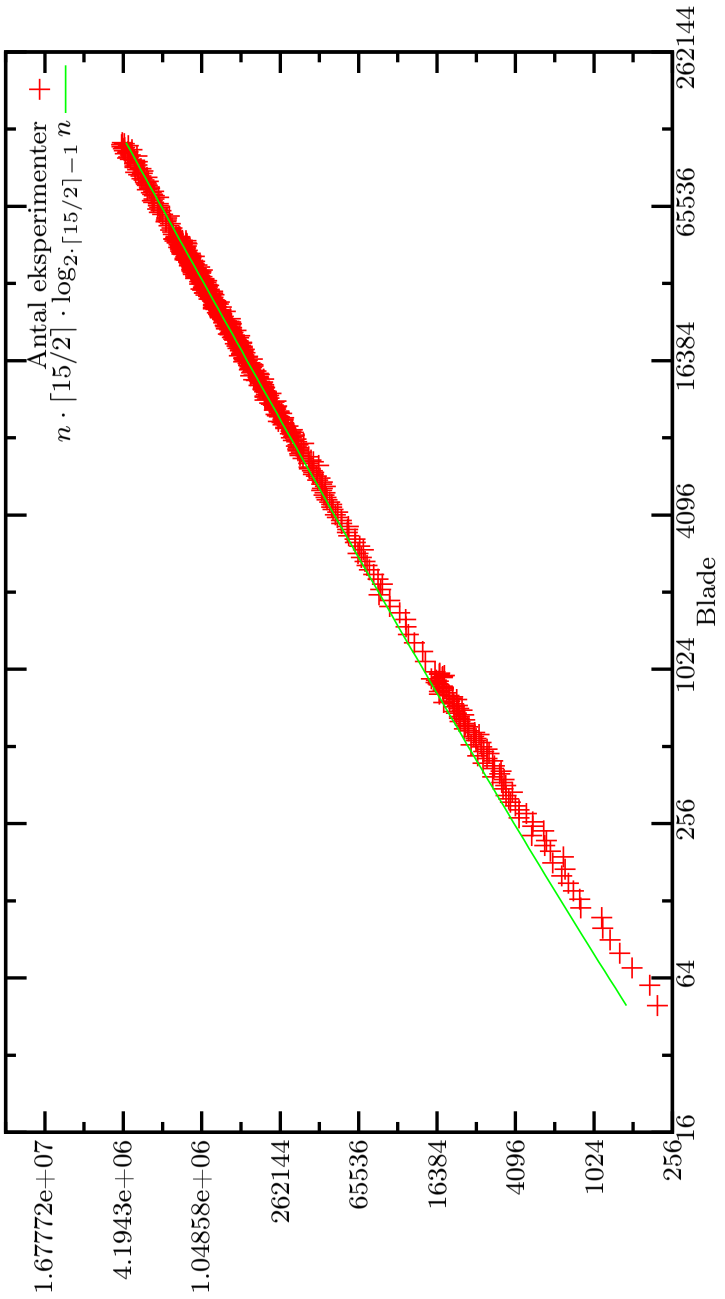
Figur D.7: Træer, antal eksperimenter for $d = 7$



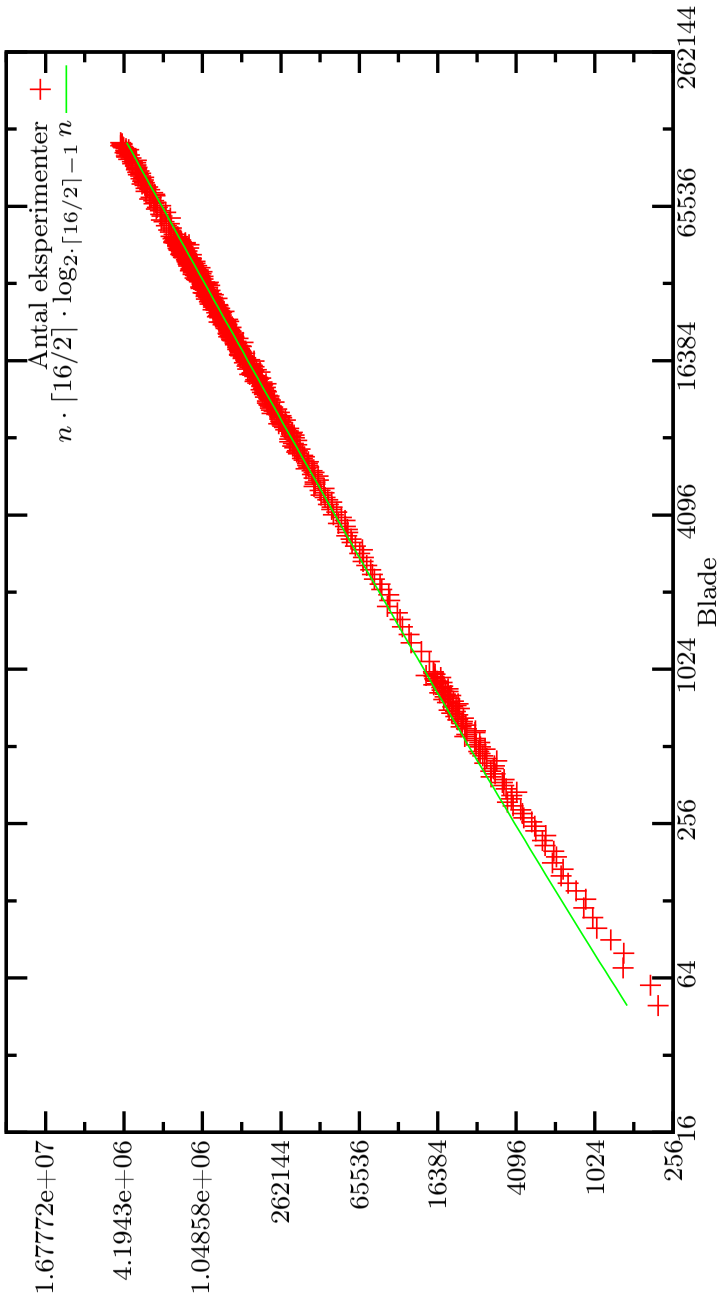
Figur D.8: Træer, antal eksperimenter for $d = 8$



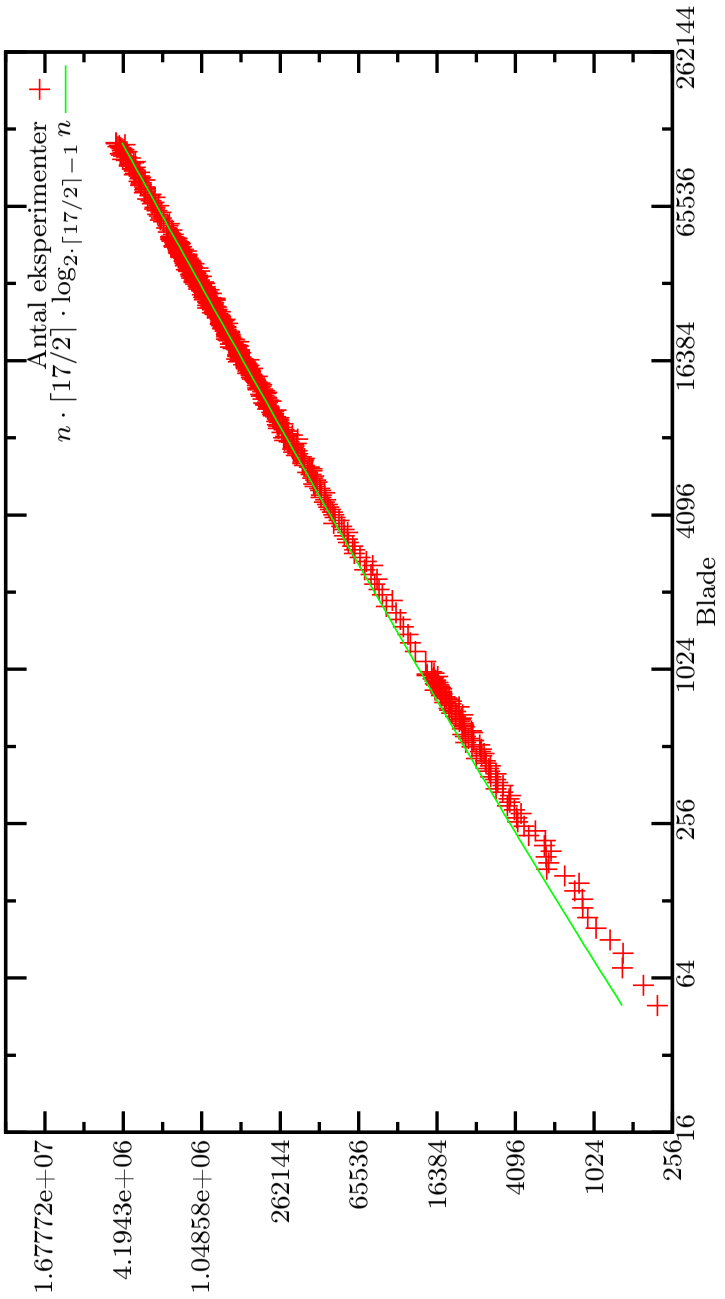
Figur D.9: Træer, antal eksperimenter for $d = 9$



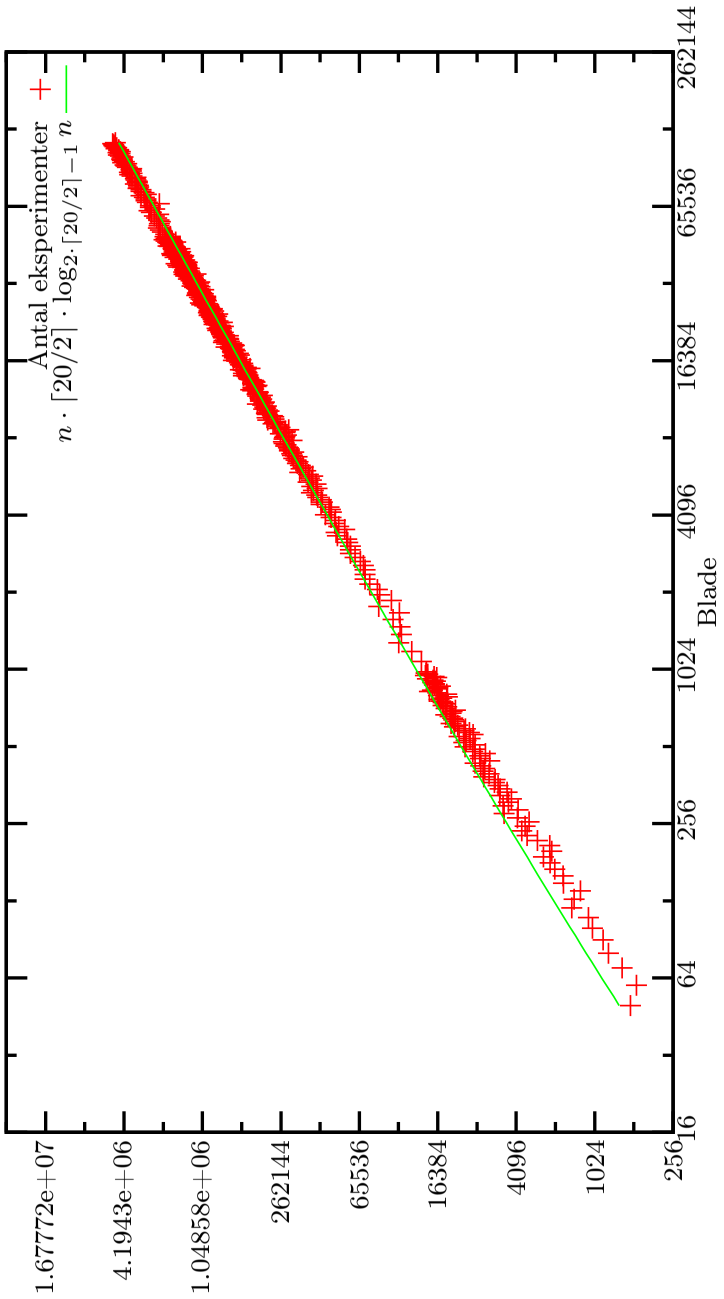
Figur D.10: Træer, antal eksperimenter for $d = 15$



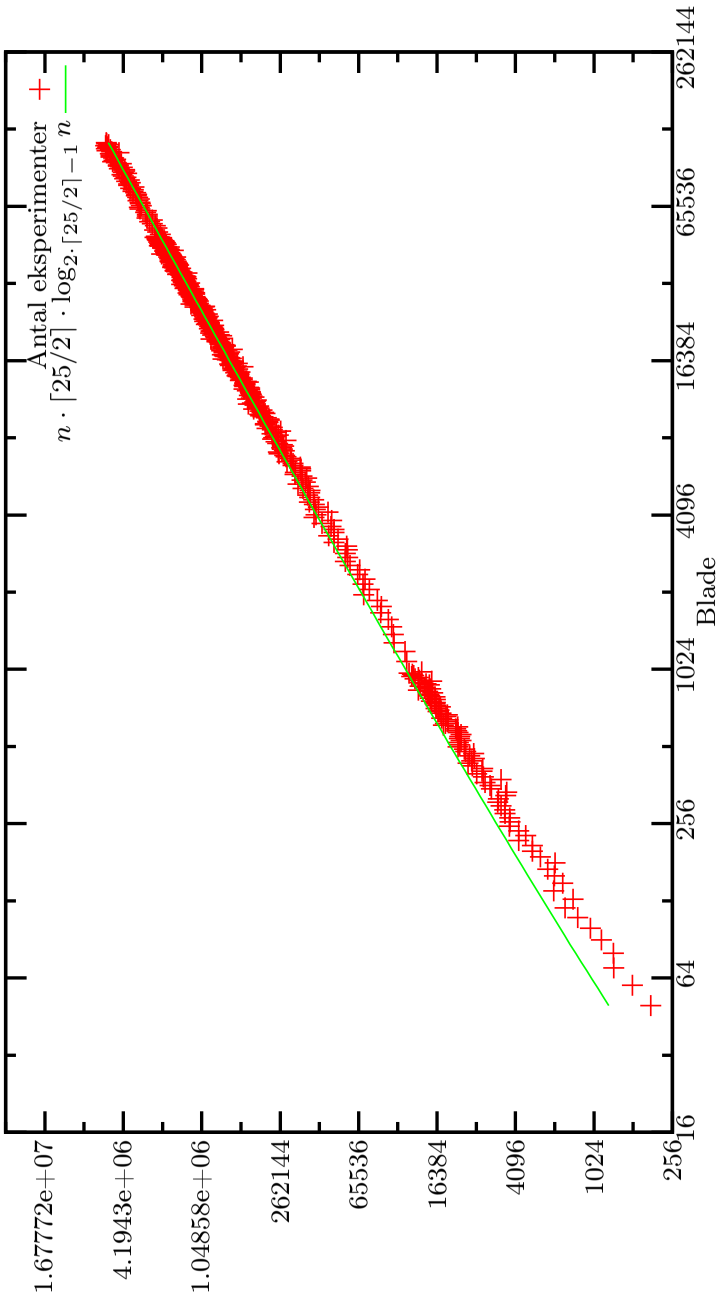
Figur D.11: Træer, antal eksperimenter for $d = 16$



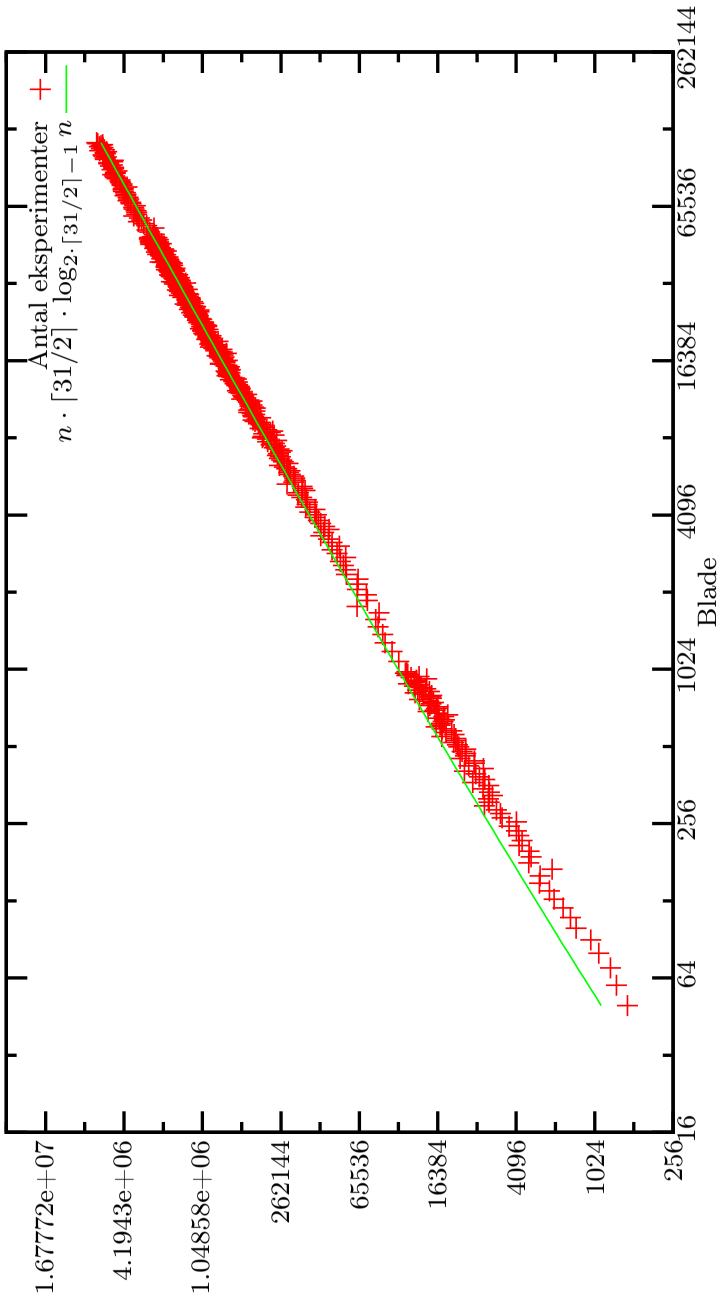
Figur D.12: Træer, antal eksperimenter for $d = 17$



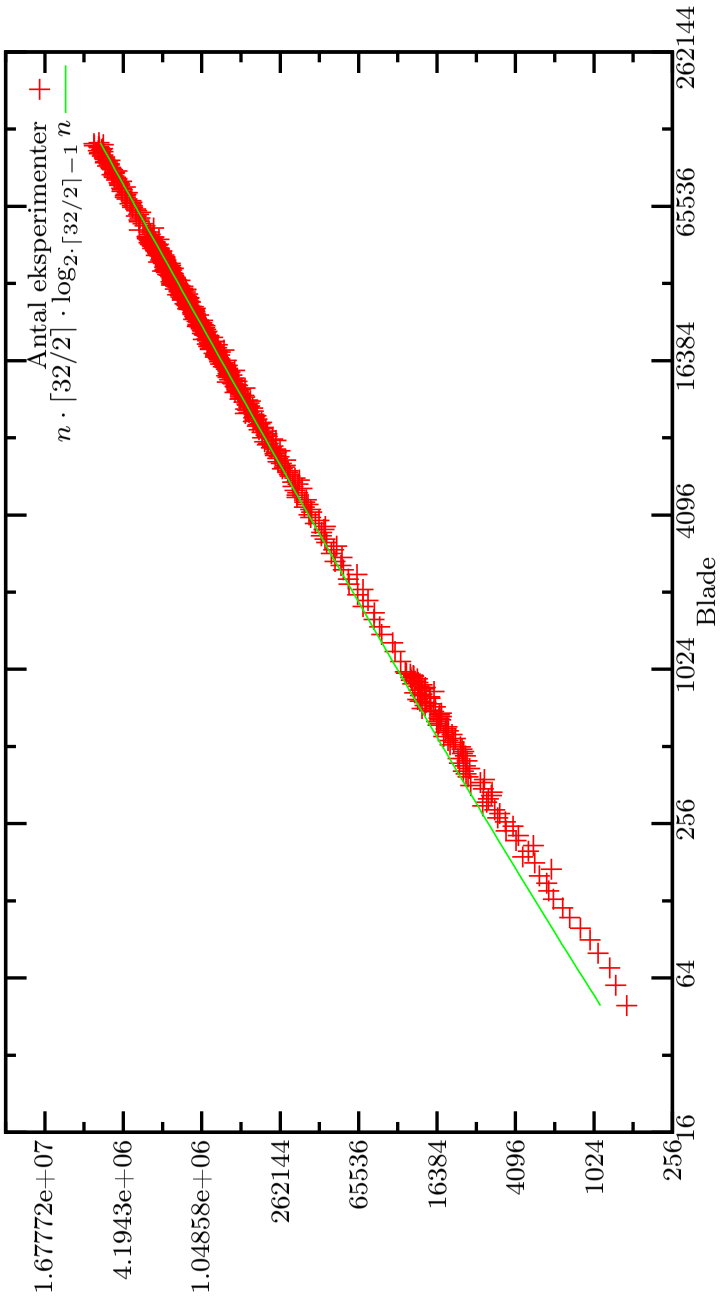
Figur D.13: Træer, antal eksperimenter for $d = 20$



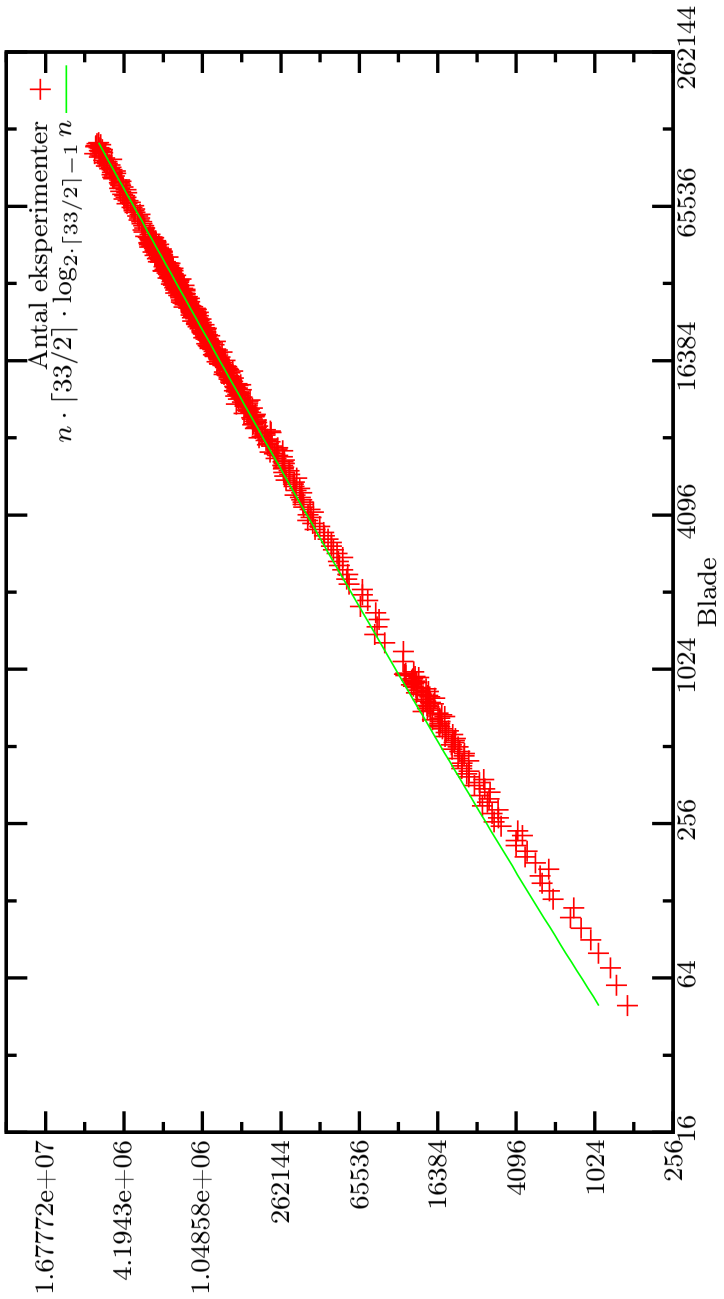
Figur D.14: Træer, antal eksperimenter for $d = 25$



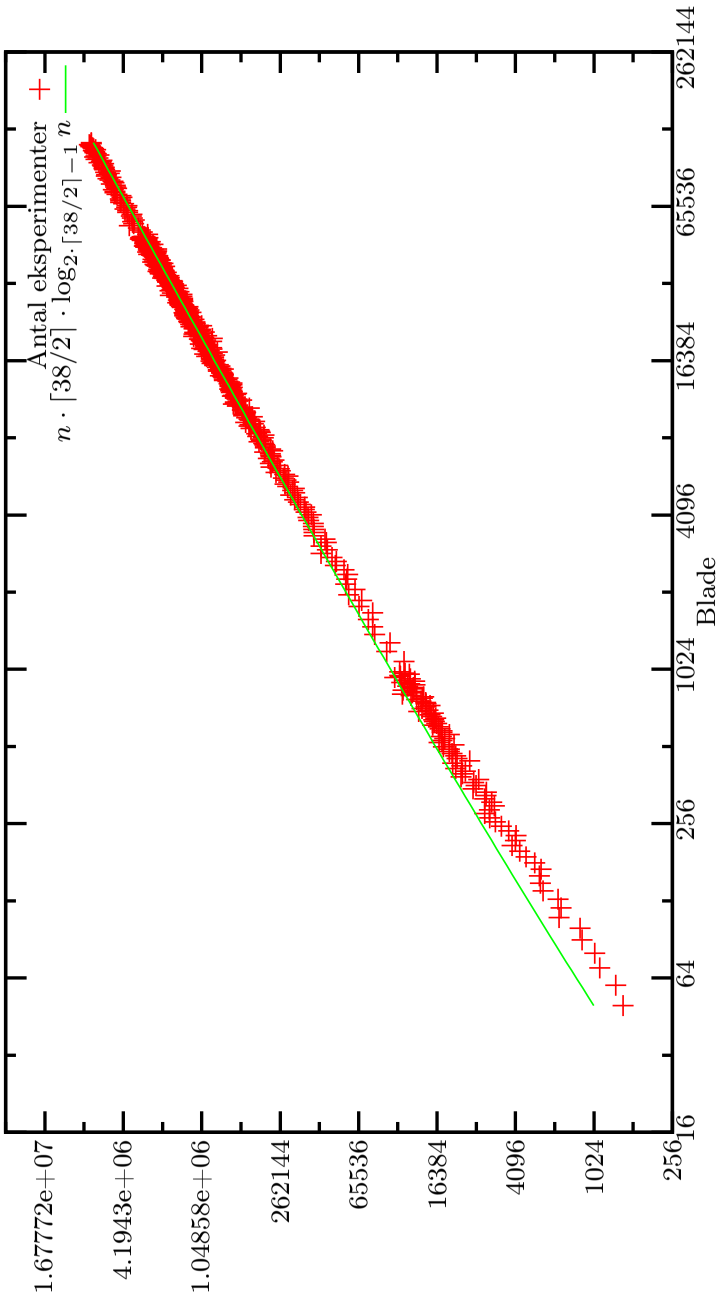
Figur D.15: Træer, antal eksperimenter for $d = 31$



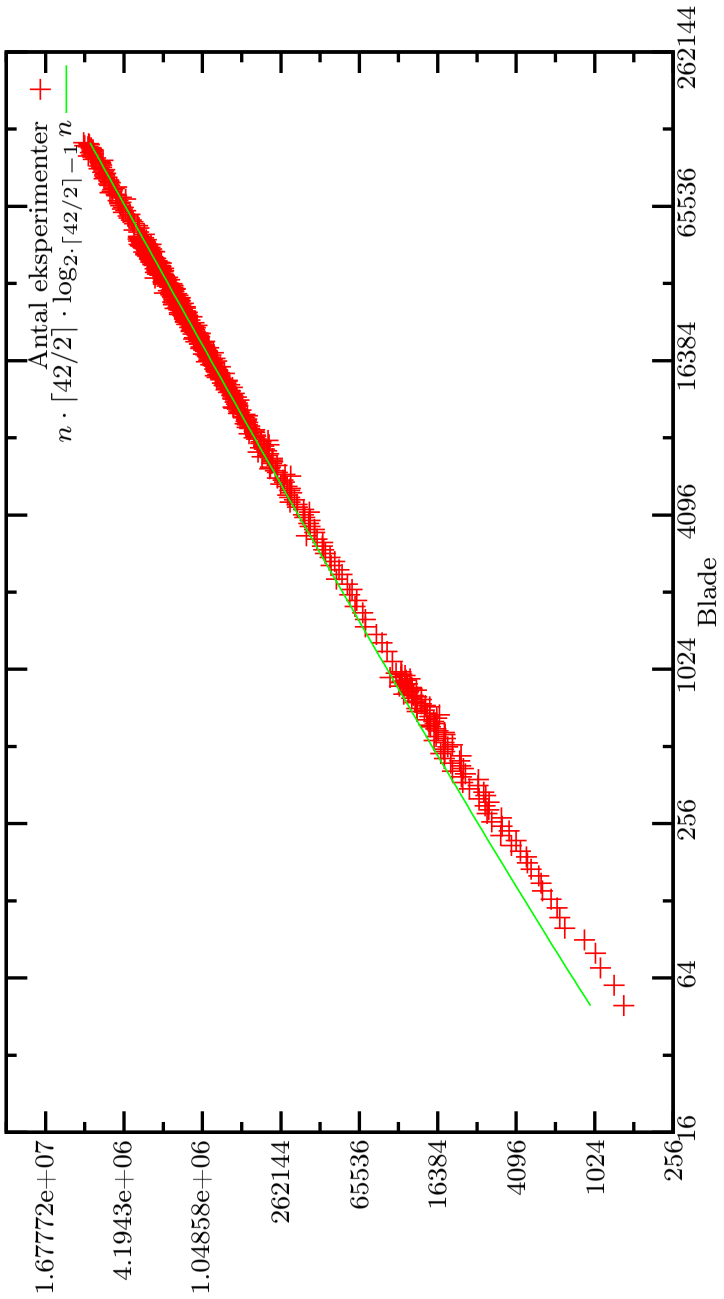
Figur D.16: Træer, antal eksperimenter for $d = 32$



Figur D.17: Træer, antal eksperimenter for $d = 33$

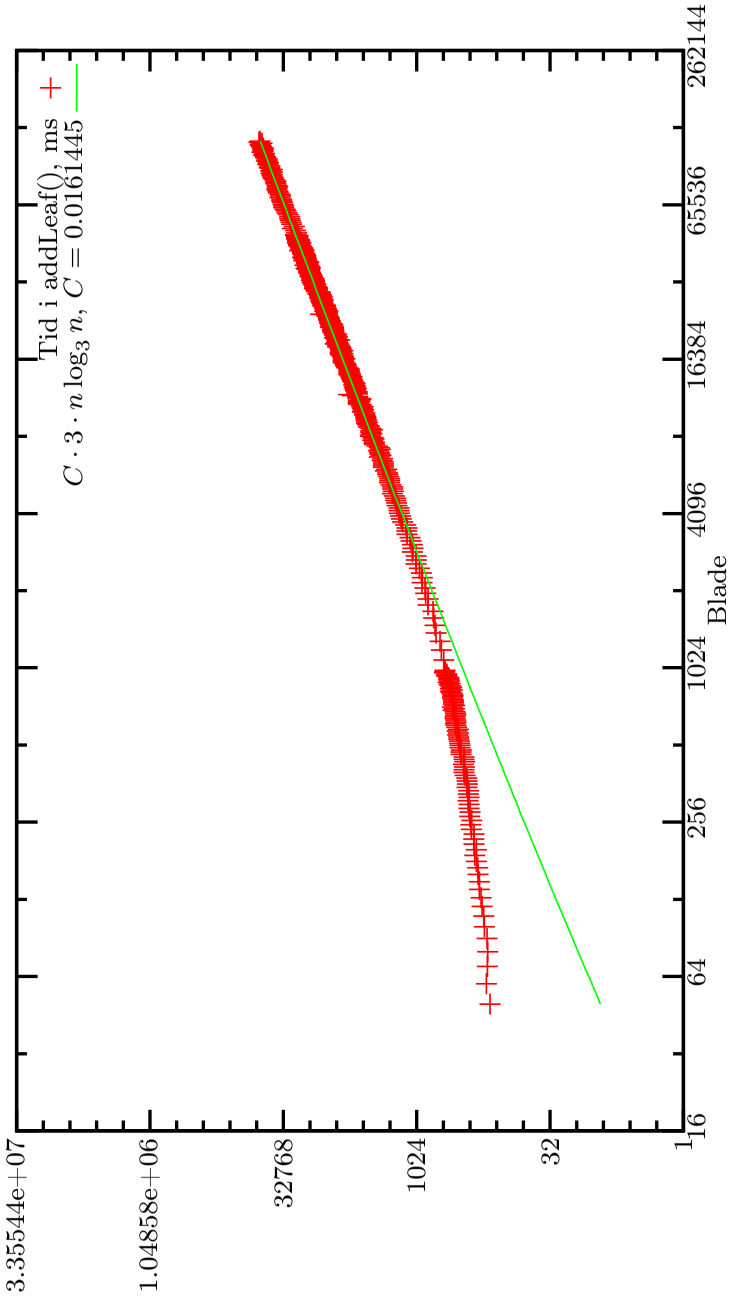


Figur D.18: Træer, antal eksperimenter for $d = 38$

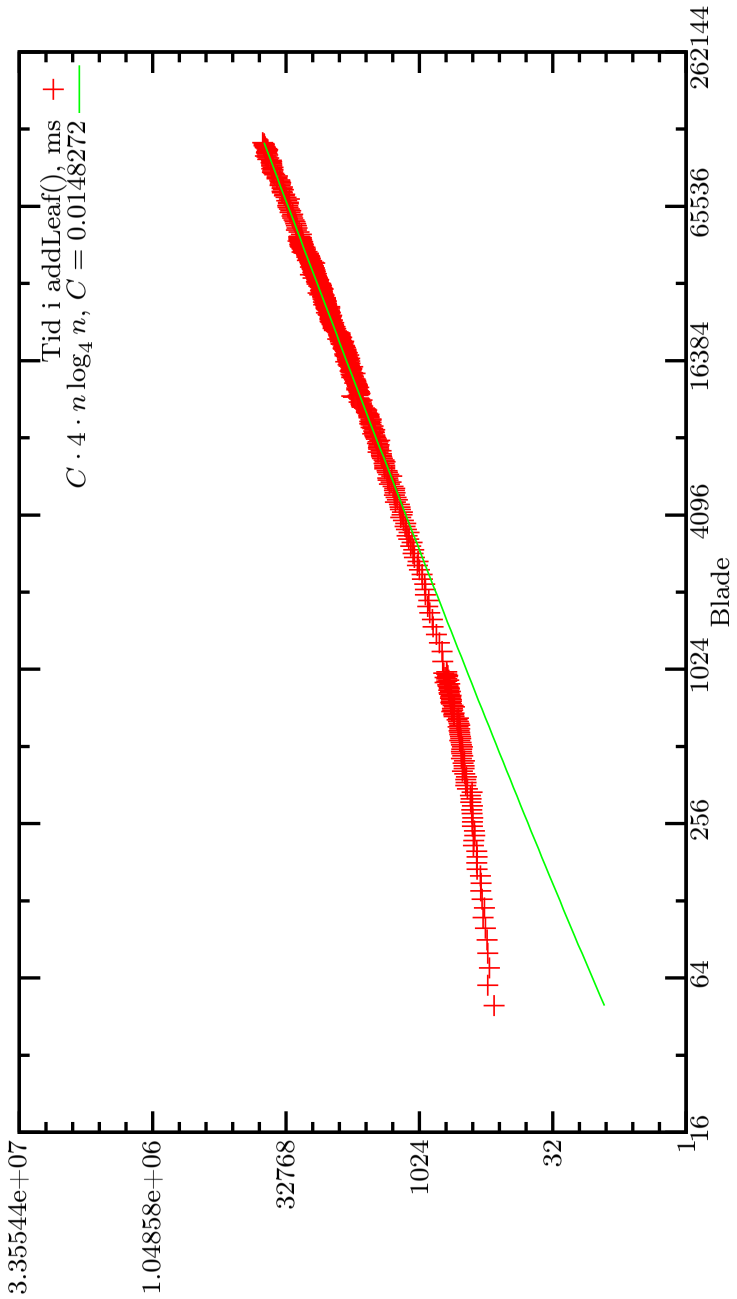


Figur D.19: Træer, antal eksperimenter for $d = 42$

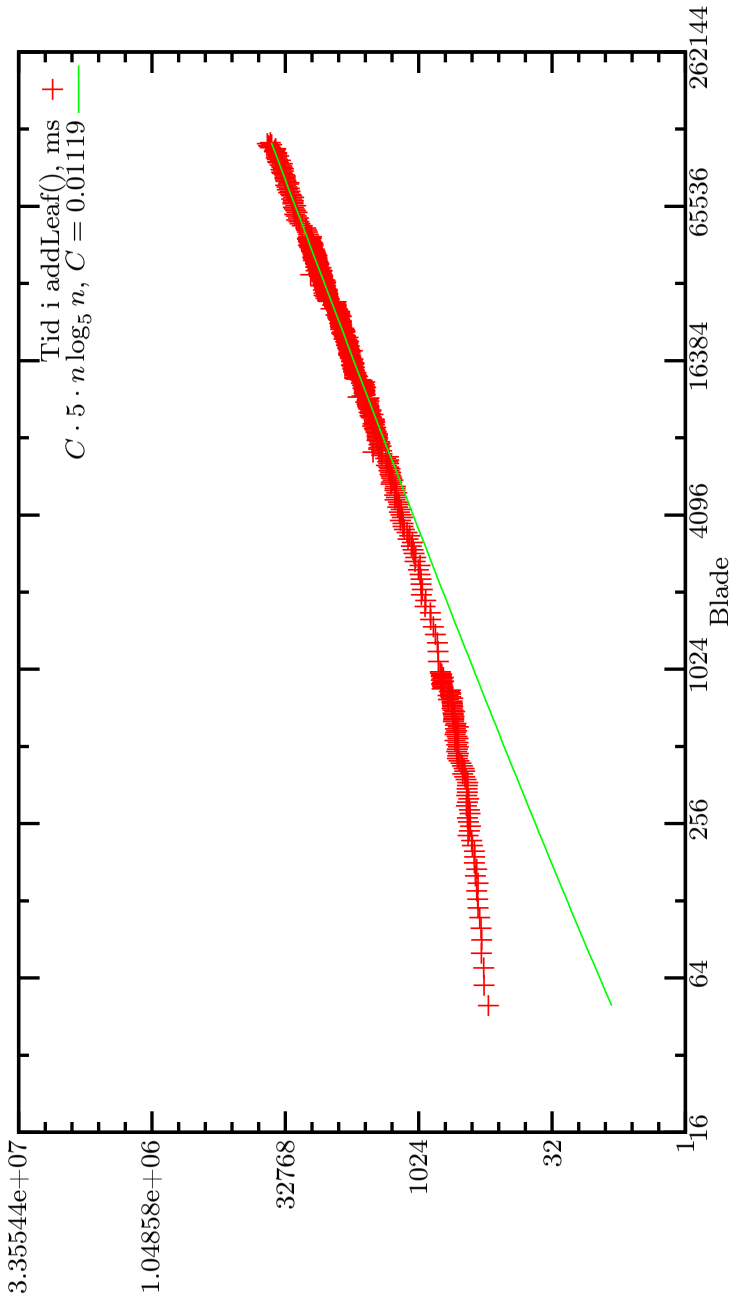
D.2.2 Tidsforbrug



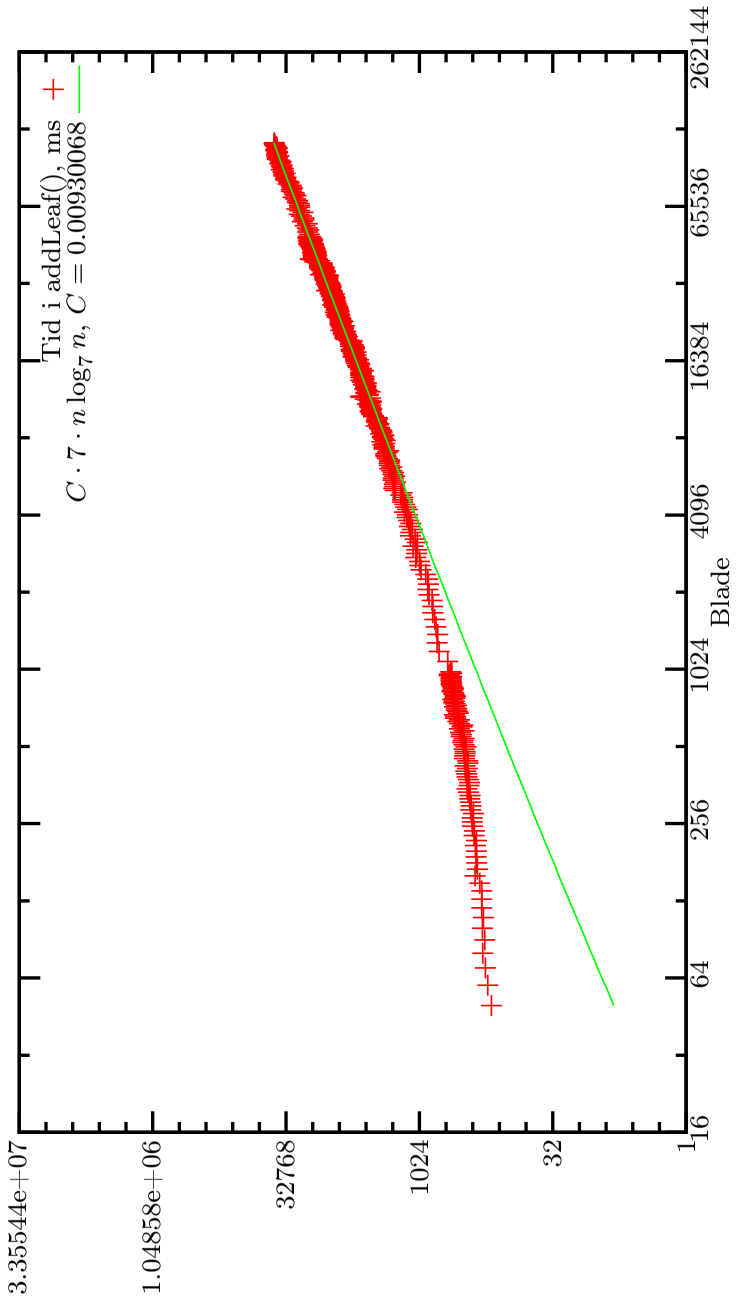
Figur D.20: Træer, tidsforbrug for $d = 3$



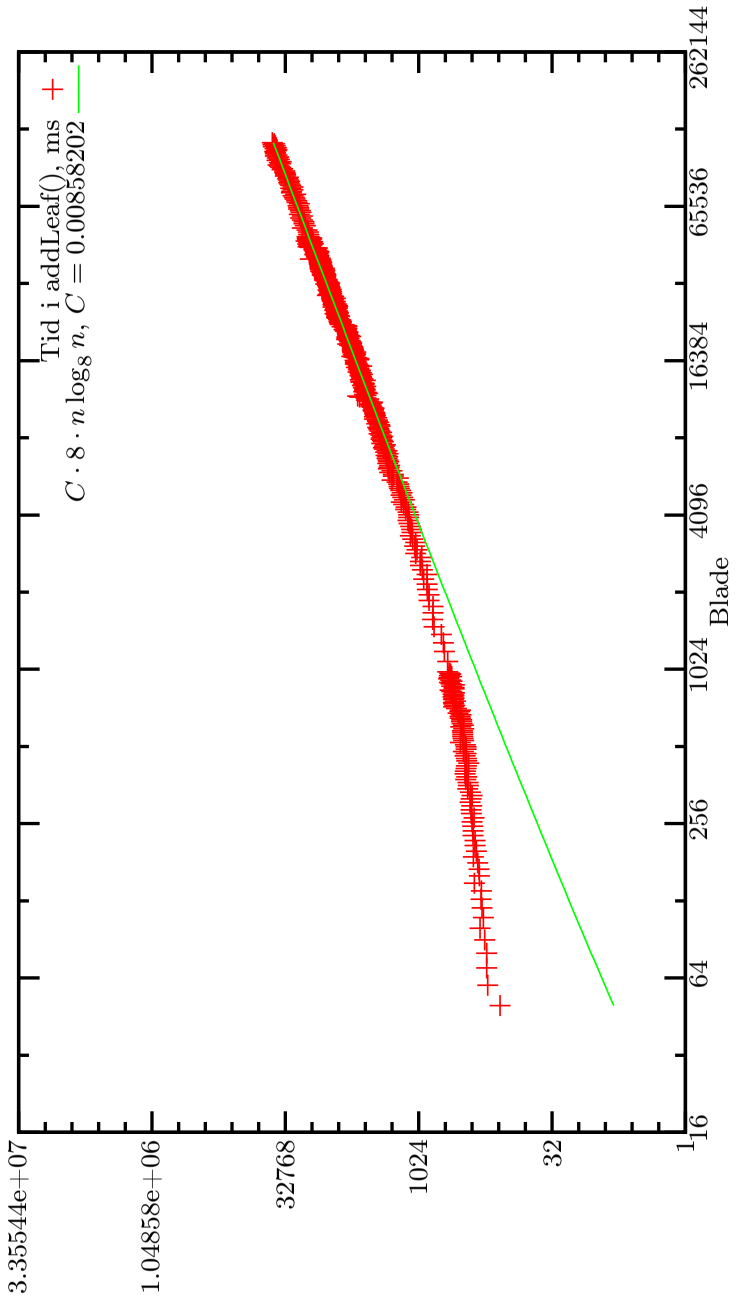
Figur D.21: Træer, tidsforbrug for $d = 4$



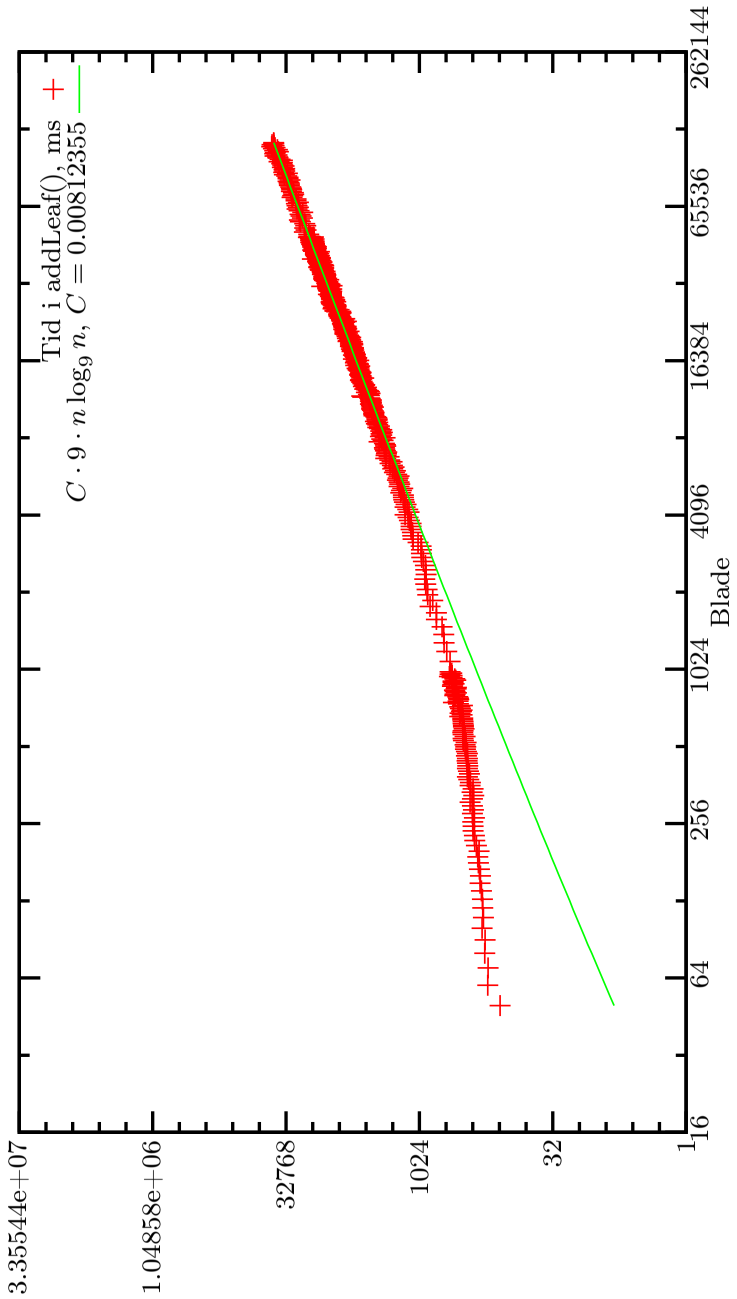
Figur D.22: Træer, tidsforbrug for $d = 5$



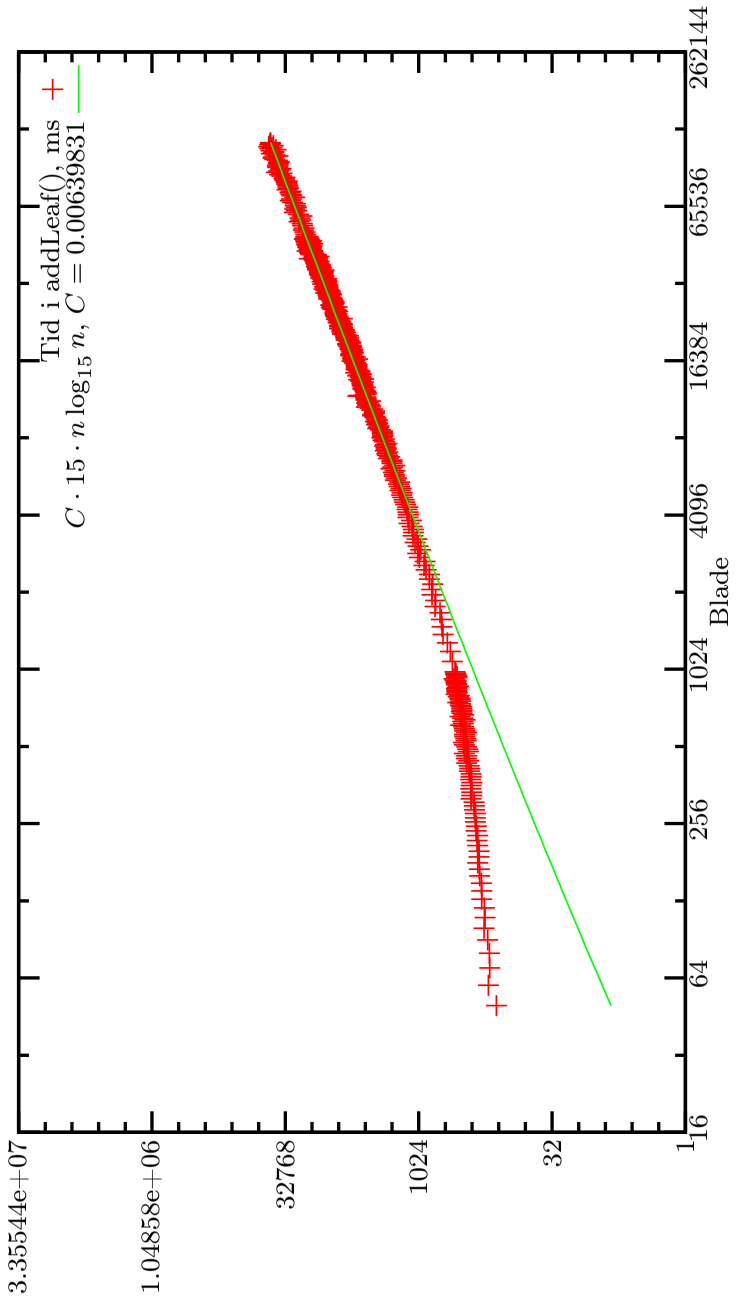
Figur D.23: Træer, tidsforbrug for $d = 7$



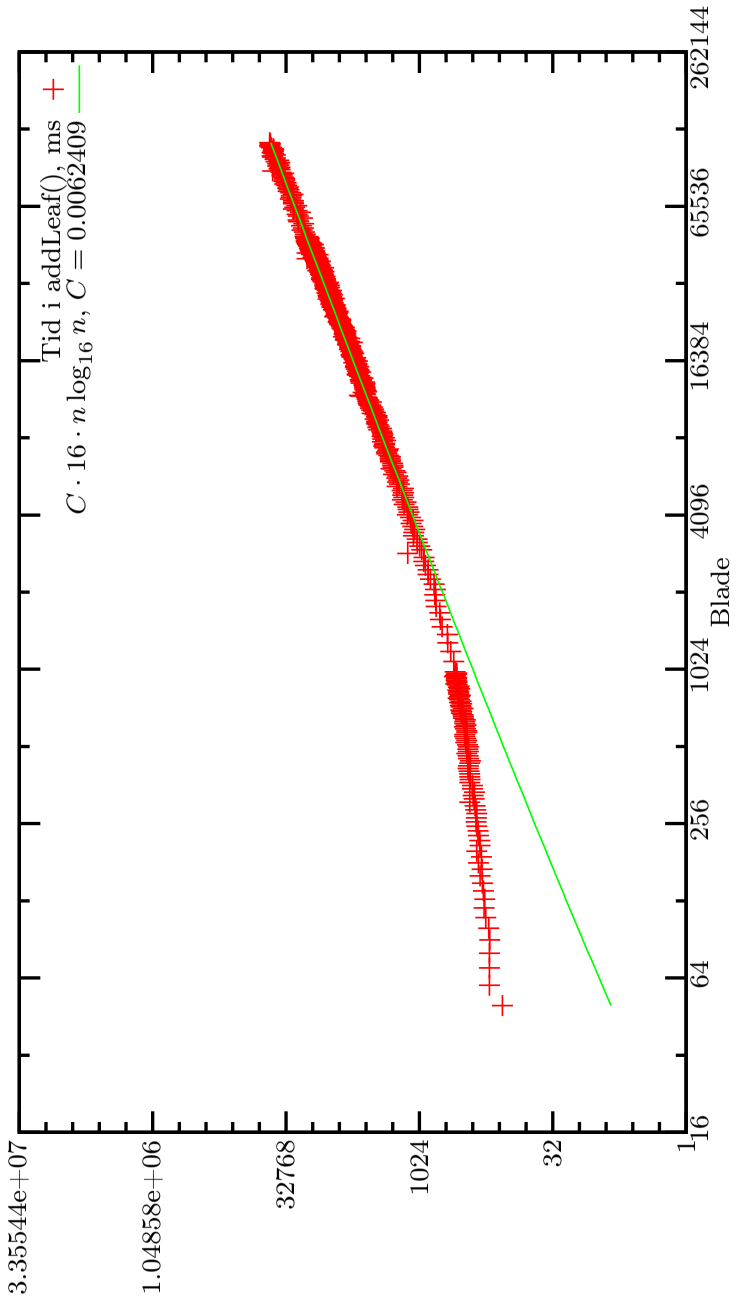
Figur D.24: Træer, tidsforbrug for $d = 8$



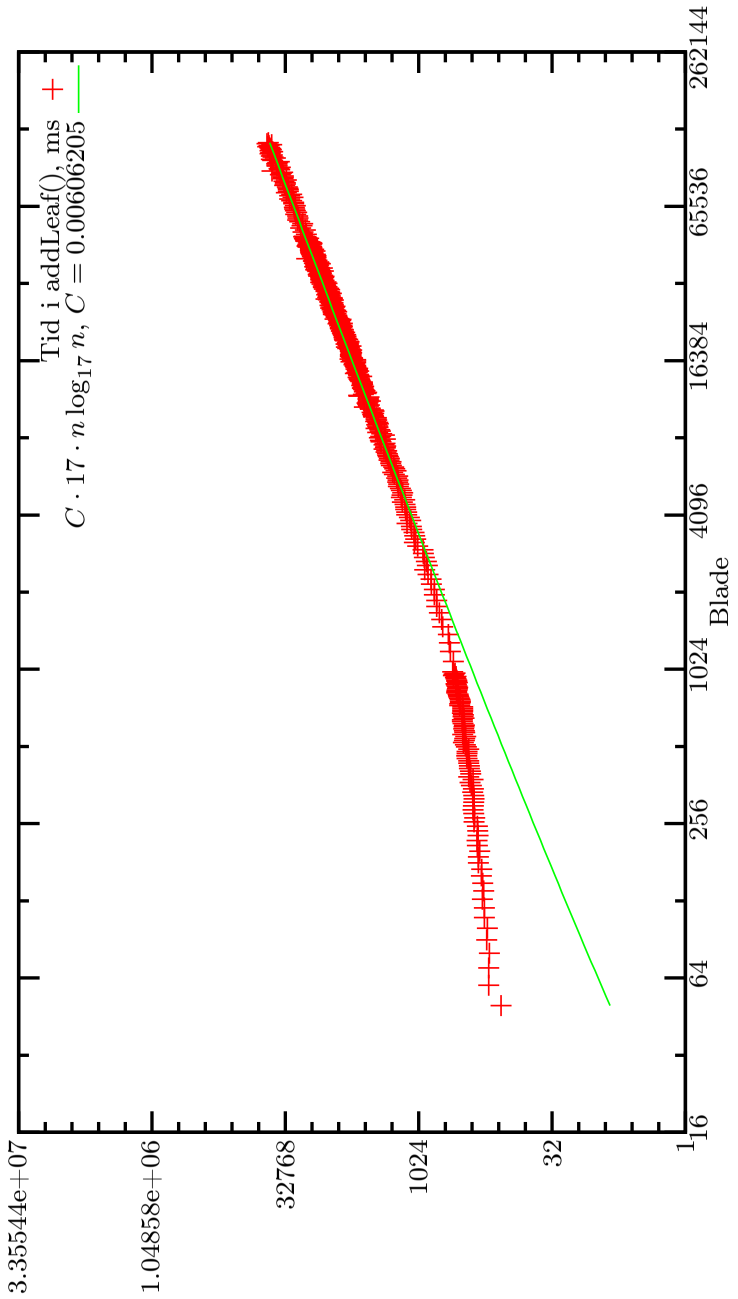
Figur D.25: Træer, tidsforbrug for $d = 9$



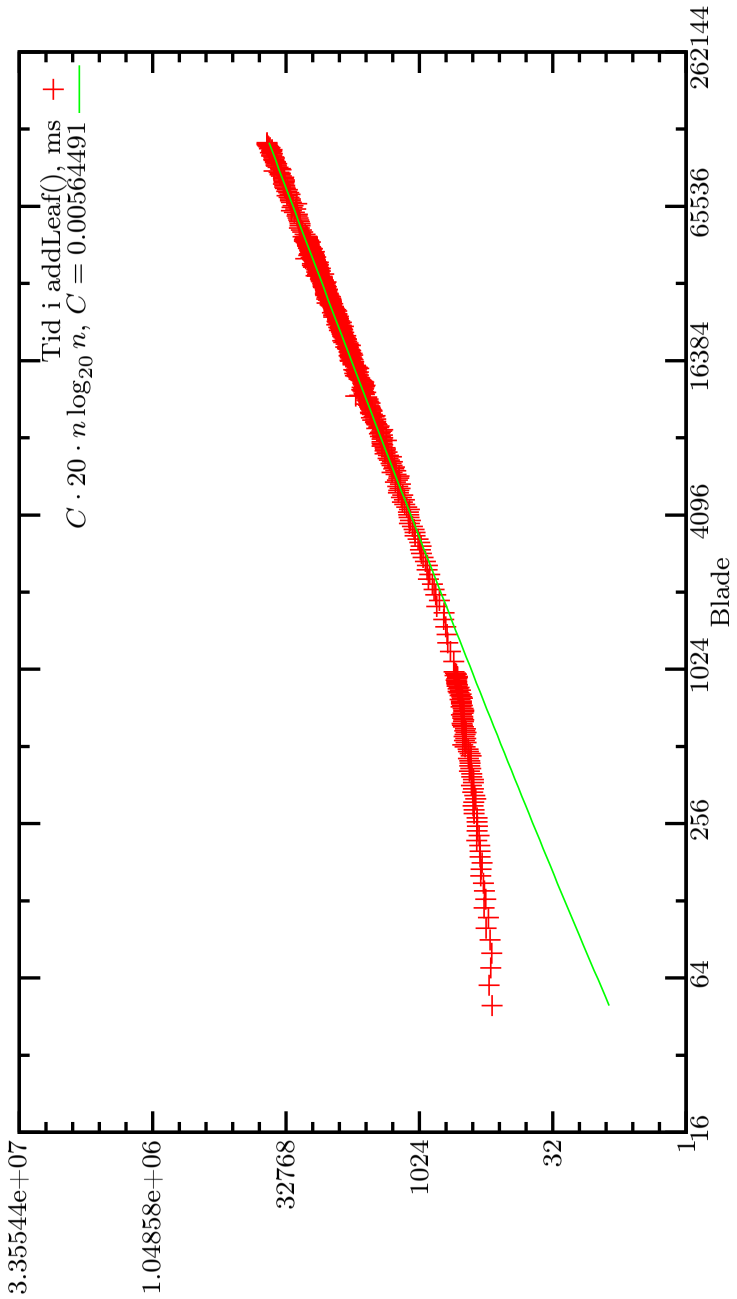
Figur D.26: Træer, tidsforbrug for $d = 15$



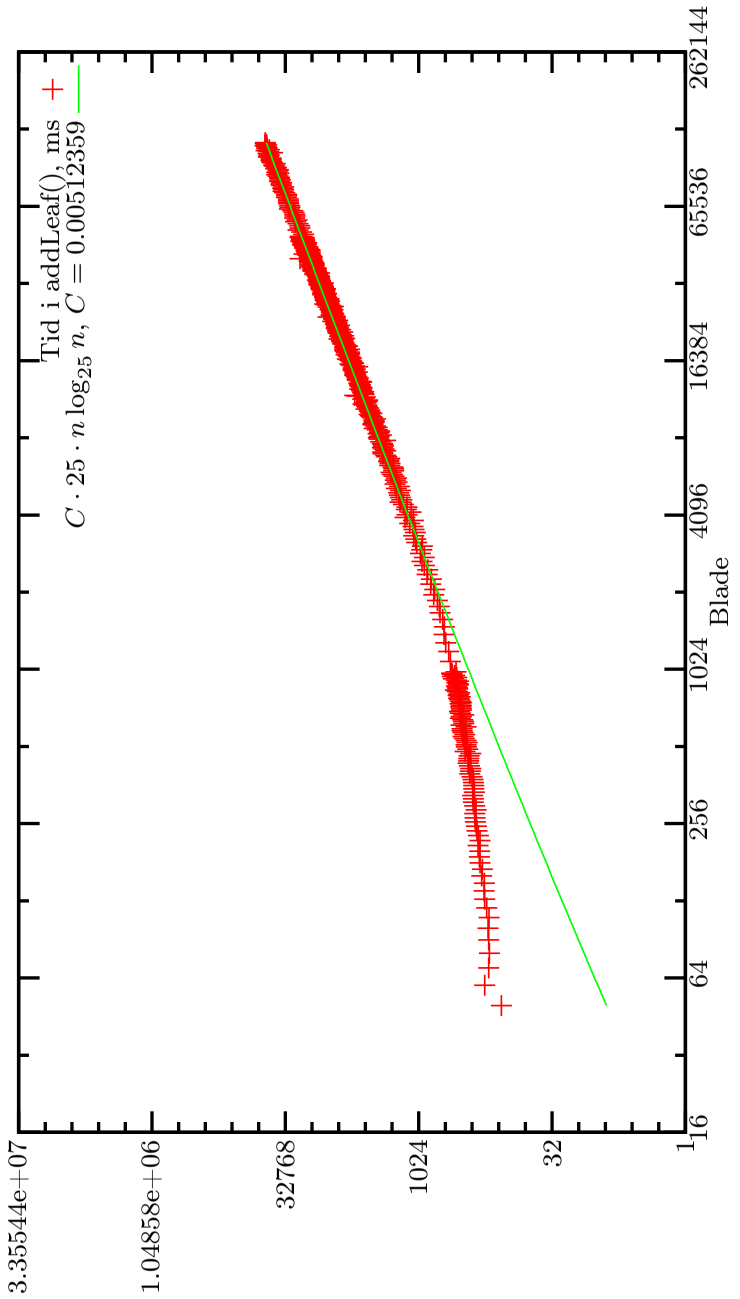
Figur D.27: Træer, tidsforbrug for $d = 16$



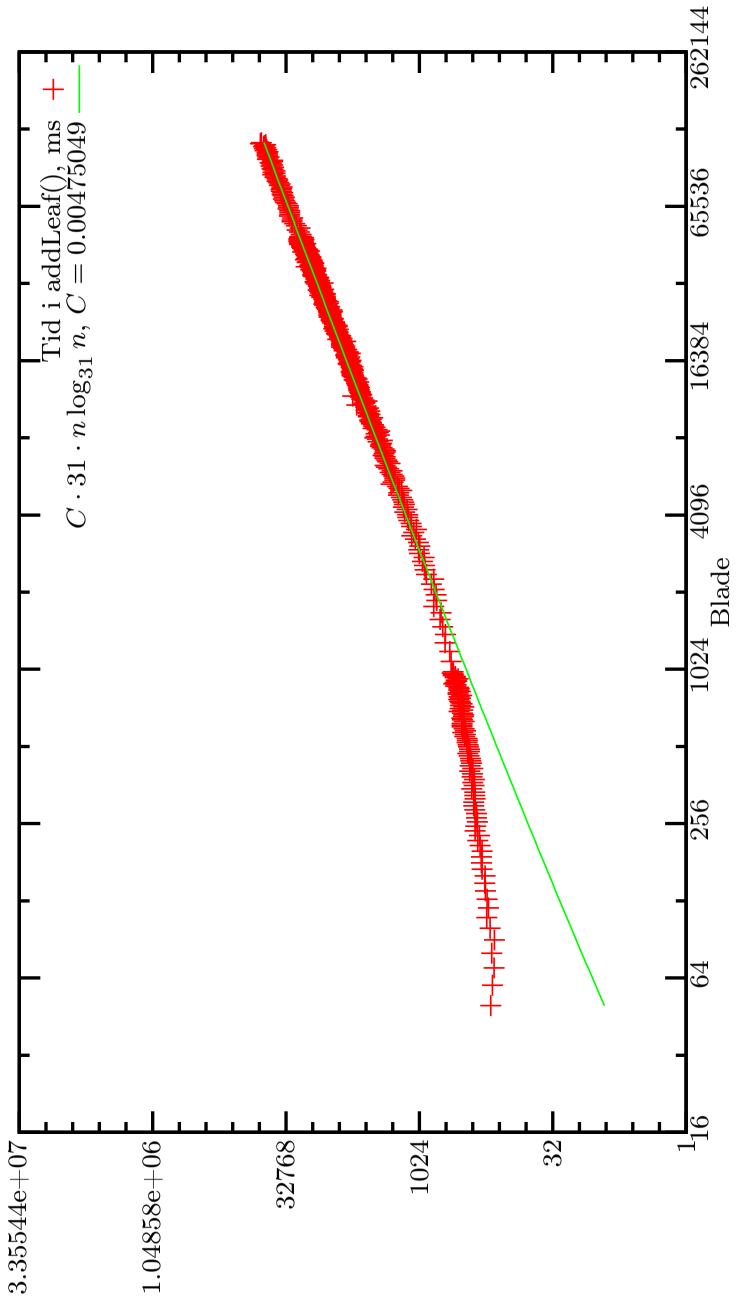
Figur D.28: Træer, tidsforbrug for $d = 17$



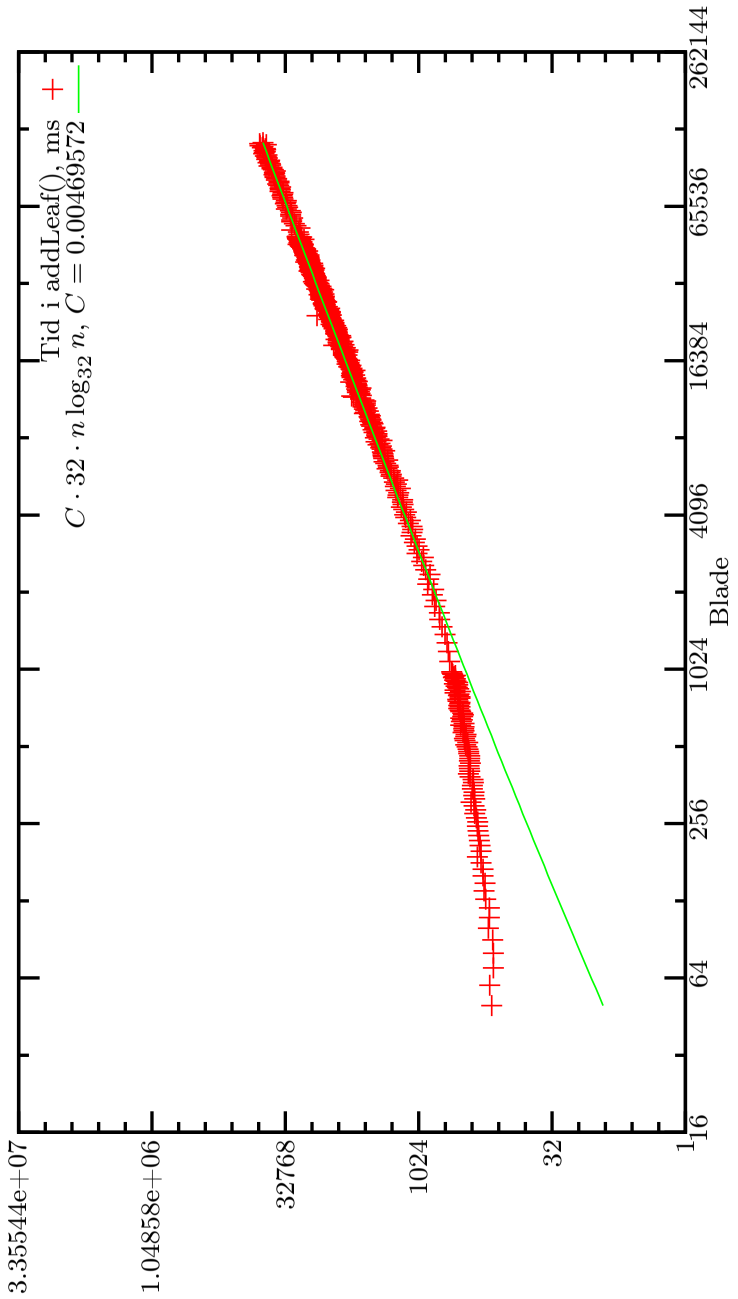
Figur D.29: Træer, tidsforbrug for $d = 20$



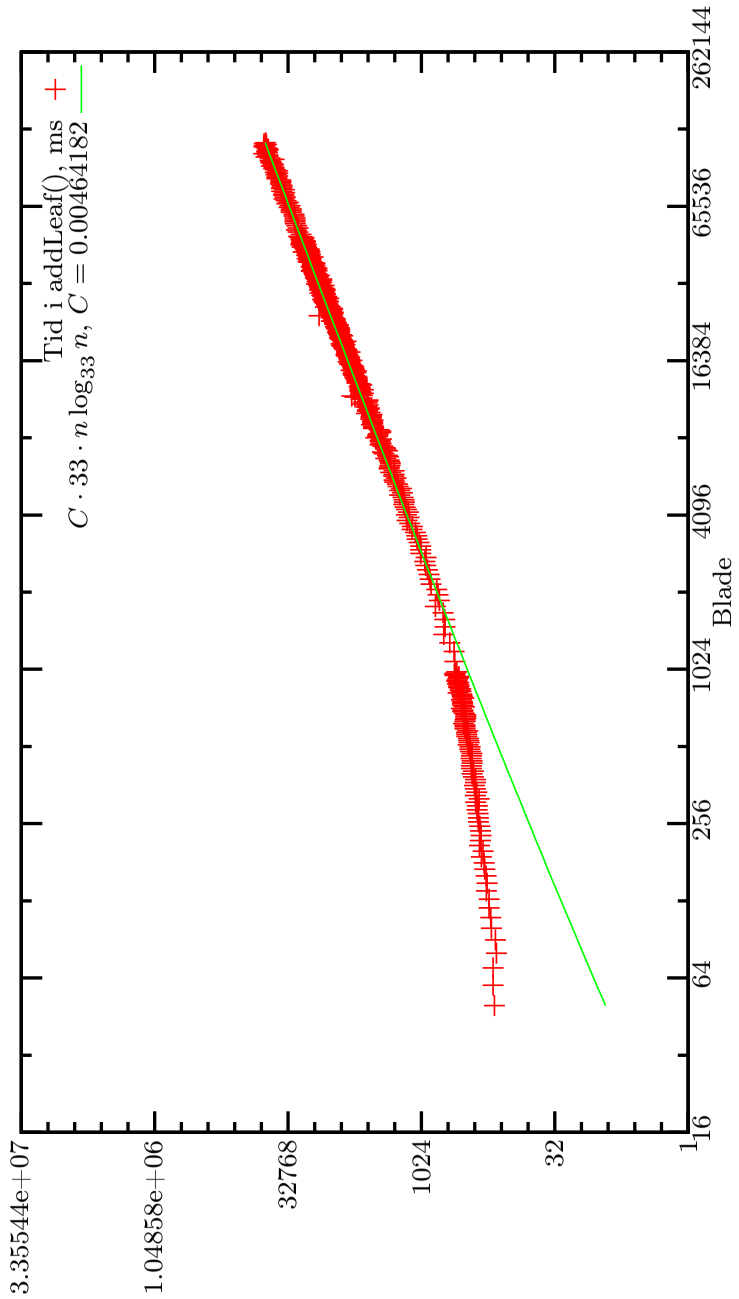
Figur D.30: Træer, tidsforbrug for $d = 25$



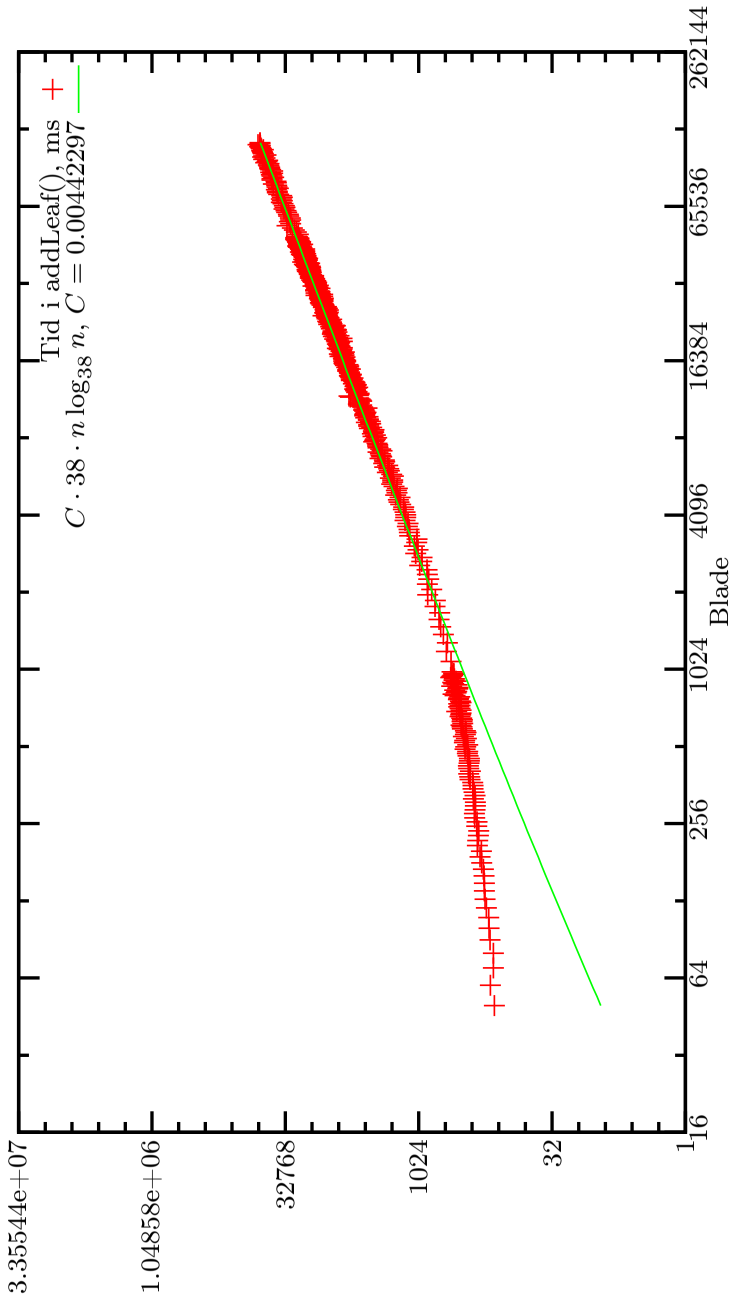
Figur D.31: Træer, tidsforbrug for $d = 31$



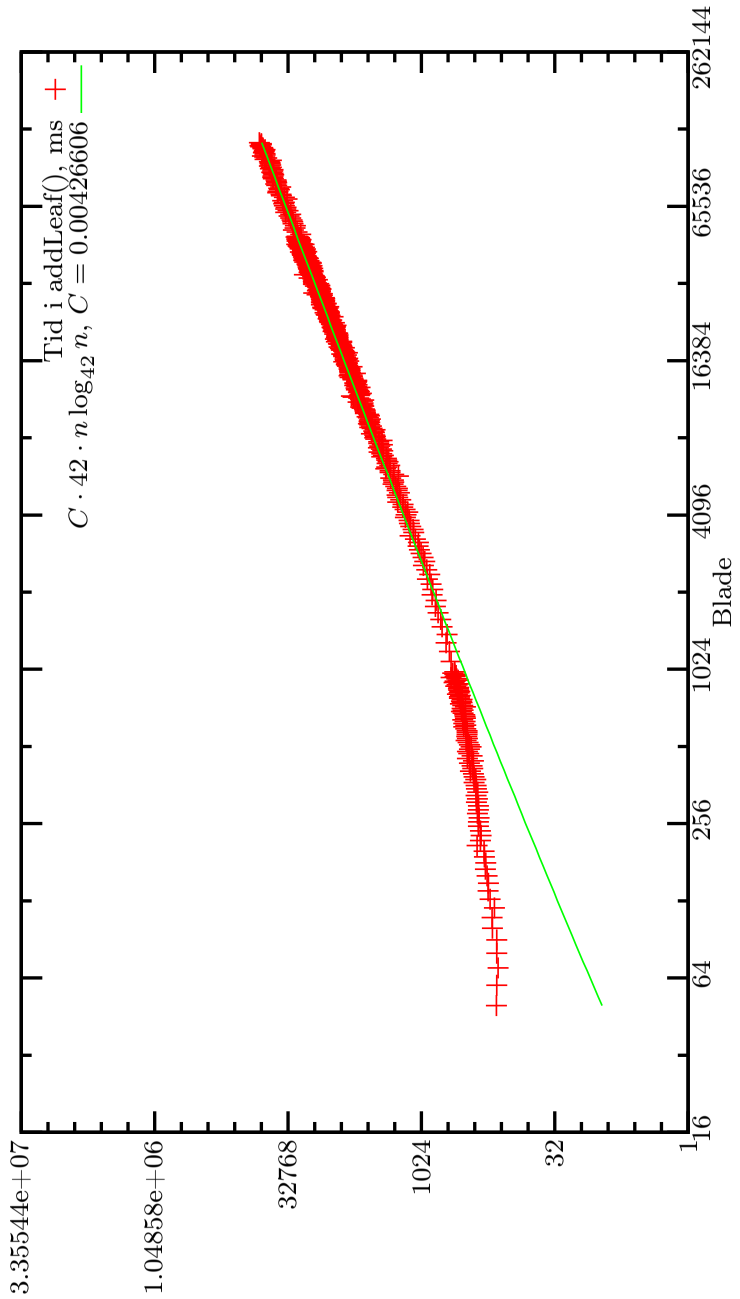
Figur D.32: Træer, tidsforbrug for $d = 32$



Figur D.33: Træer, tidsforbrug for $d = 33$



Figur D.34: Træer, tidsforbrug for $d = 38$



Figur D.35: Træer, tidsforbrug for $d = 42$

Bilag E

Split-Dist resultater for Pfam

% delte kanter	antal kanter	matrice
0	12	Acp26Ab
0	29	2-ph_phosp
1	2033	trans_reg_C
1	219	Archaeal_ATPase
2	1918	sigma70_r2
2	2108	GST_N
2	2138	Phage_integrase
2	2182	GST_C
2	347	AraC_binding
3	2262	aldedh
3	345	ABM
4	1849	VPR
4	204	5-FTHF_cyc-lig
4	276	A2M
4	281	ABC1
5	1669	AMO
5	1727	ATP-synt_8
5	2053	Pribosyltran
5	288	AFP
5	289	5_3_exonuclease
5	294	Amidase_3
5	372	Aldose_epim
6	1850	Vif
6	288	APH
6	288	arginase
6	315	ArfGap
7	199	AdoHcyase

Split-Dist resultater for Pfam

% delte kanter	antal kanter	matrice
7	210	ADK_lid
7	243	Amidase_4
7	280	alk_phosphatase
7	310	AP_endonuc_2
7	364	aminotran_4
8	181	Amidohydro_2
8	228	Acylphosphatase
8	248	ALAD
8	342	ACPS
8	46	3H
8	58	Acetylald_DH
9	187	A2M_N
9	199	A_deaminase
9	275	An_peroxidase
9	278	ABC-3
9	309	Adaptin_N
10	223	AlaDh_PNT_N
10	263	5_3_exonuc_N
10	303	acid_phosphat
10	66	ABG_transport
10	66	Ada_Zn_binding
10	80	AcetylCoA_hydro
11	220	Alpha_E3_glycop
11	229	AlaDh_PNT_C
12	183	AdoHcyase_NAD
12	194	Amidinotransf
12	202	Ala_racemase_C
12	75	AAA_div
13	196	Aerolysin
13	236	arrestin_C
14	147	Alpha_core
14	186	AICARFT_IMPCHas
14	214	Adenylsucc_synt
16	186	Arginosuc_synth
16	210	Acetate_kinase
17	184	Activin_recp
17	211	Arena_glycoprot
18	224	arrestin
18	27	3-alpha
21	37	ABA_WDS
22	74	Acetyltransf2

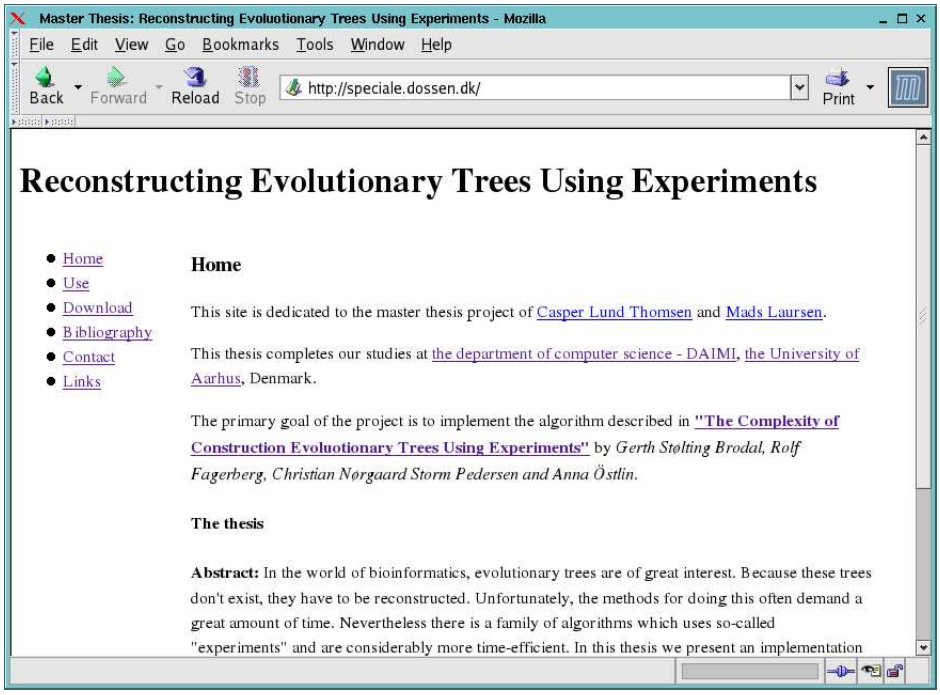
% delte kanter	antal kanter	matrice
24	62	Adeno_E1A
27	11	4F5
27	22	ADC
27	33	Adeno_52K
27	54	AAL_decarboxy
28	35	adeno_100
40	25	5-nucleotidase
41	24	Adeno_E1B_55K_N
42	35	Adeno_E1B_55K
48	39	A4_EXTRA
50	30	Adeno_E1B_19K
50	8	Acyl_transf_2
54	11	AbfB
64	28	Adeno_E3_14_5
70	10	AceK
77	9	AAR2
87	8	ACN9

Bilag F

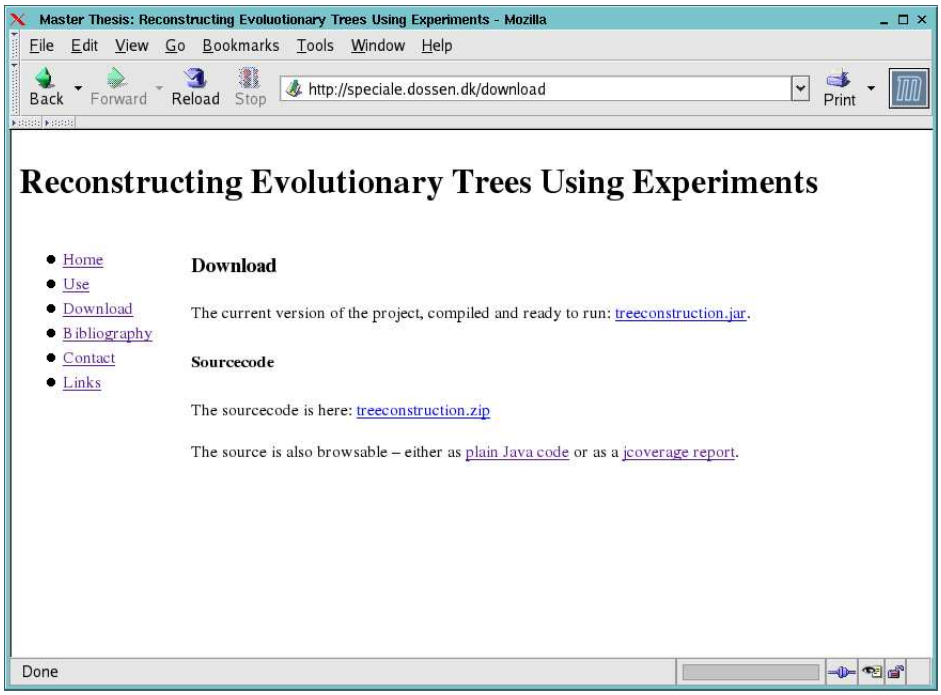
Hjemmeside – skærmdumps

Som nævnt i afsnit 1.1 har vi til specialet oprettet en hjemmeside, hvor koden til vores implementation kan gennemgås og downloades. Siden findes på adressen `http://speciale.dossen.dk/`.

De følgende figurer er skærmdumps af siden. På F.1 ser vi forsiden, der kort fortæller hvad det går ud på og peger på onlineudgaven af denne rapport. Figur F.2 viser den side, hvorfra koden kan downloades.

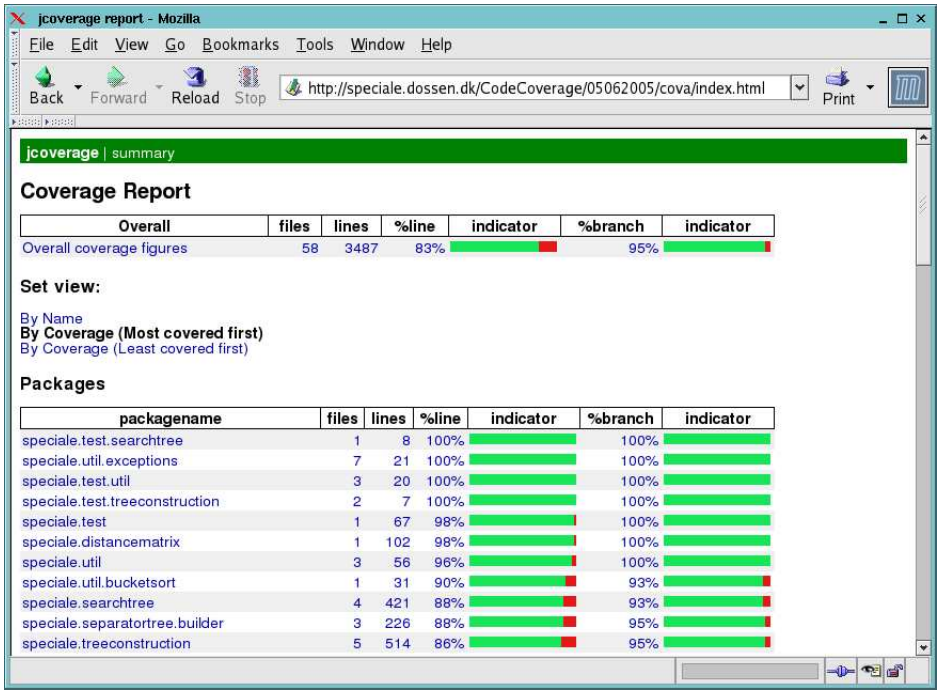


Figur F.1: Hovedsiden.



Figur F.2: Downloadsiden.

Den detaljerede rapport om vores testdækning (se afsnit 3.7) er også gjort tilgængelig på projektsiden, og forsiden på denne rapport kan ses på figur F.3.



Figur F.3: Forsiden af rapporten om dækningsgrad.