UNIVERSITETET I OSLO
Institutt for informatikk

Optimalisert indeksering av spatiale data med R-trær

Masteroppgave

Niels Petter
Rasch-Olsen

25. juli 2007

Forord

Denne oppgaven omhandler indeksering av spatiale data og er utført som ledd i prosjektet Diligent i regi av FAST. FAST utvikler søketeknologi for bedriftsmarkedet. Oppgaven er veiledet av førsteamanuensis II Knut Omang.

Takk til alle som har hjulpet meg. Takk til Knut Omang for god veiledning samt tolerering av mine arbeidsvaner. Takk til Stefan Debald for god oppfølging. Takk til Mads Sibeko for kaffeslaberas og generelt kodeprat. Og sist, men ikke minst takk til min samboer Mari.

Innhold

Figurer

Tabeller

Kapittel 1

Innledning

Behovet for å kunne jobbe med og lagre spatiale data har økt kontinuerlig de siste tyve årene. Mye forskning har foregått innen dette feltet, og områder som Geografiske informasjonssystemer (GIS), Computer Aided Design (CAD) og robotikk har ledet an. I navnet spatiale data ligger det at dette har med avstands- og romforhold å gjøre. Spatiale data er altså alle data som har informasjon som kan knyttes til rom.

Indeksering av spatiale data handler både om rask tilgang på data samt å kunne utføre spatiale operasjoner på dataene. Data indekseres forskjellig avhengig av dataenes natur og operasjonene man ønsker å utføre på disse. I klassiske databasesystemer benyttes i all hovedsak B-treet for raske oppslag. Dette har tidligere blitt omtalt som det "allstedsnærværende B-Treet"[7] grunnet dets ekstensive bruk.

Indeksering av spatiale data skiller seg i to grupper:

- punkter i et flerdimensjonalt rom
- n-dimensjonale rektangler

Den første gruppen indekserer punktene ved å partisjonere opp det underliggende rommet i ikke-overlappende regioner. Innenfor matematikken kalles dette for rom-partisjonering. En slik partisjonering kan fortsette rekursivt til man har oppnådd ønsket presisjon. Kd-trær er et eksempel på en slik struktur[3]. Når det gjelder mer komplekse spatiale objekter må strukturen også kunne håndtere f.eks polygoner og polylinjer. Dette problemet kan reduseres til å håndtere rektangler, såkalte Minimal Bounding Rectangles(MBR), som approksimasjoner på de komplekse objektene, jf. avsnitt 4.1 på side 24. Det er sistnevne denne oppgaven omhandler da spatiale data er høyst heterogene i natur. Både når det gjelder dimensjonene de opptar og ekstensjonen i disse. I avsnitt 2.3 på side 5 presenteres en oversikt over arbeidet som har blitt gjort på dette området.

For indeksering av spatiale data, ble R-treet foreslått i 1984 av Antonin Guttman[10]. De fleste store databaseleverandører har i dag implementert R-trær som en del av sitt spatiale indekseringstilbud[17, 12]. R-trær er en struktur som minner mye om B-trær. Men der B-trær indekserer punkter i en 1-dimensjonal sammenheng (f.eks heltall), kan R-trær arbeide med det n-dimensjonale. Originalt ble to algoritmer for nodedelinger beskrevet av Gutman, i henholdsvis lineær og kvadratisk kompleksitet. Denne oppgaven presenterer en måte å forbedre disse på ved å tillate temporære skjevheter i treet.

1.1 Problemstilling og avgrensning

Denne oppgaven omhandler indeksering av spatiale data. Arbeidet er fokusert på problemene rundt nodedelinger i R-trær, og søker å minimere overlapp mellom noder, noe som er et resultat av dårlige nodedelinger. Dette sammenfalt bra med FAST sine ønsker om en egen implementasjon av R-treet. Selve implementasjonen av R-treet, som tok en betydelig andel av tiden tilgjengelig, førte til en økt forståelse av problematikken rundt nodedelinger. Således er mastergraden todelt. En stor del av arbeidet har blitt lagt ned i undersøkelser av spatial indeksering, mens resten av tiden har blitt benyttet til å utvikle en egen implementasjon samt utvikle en egen variasjon på nodedelinger. Denne oppgaven tar ikke for seg GIS eller Diligent i detalj, men gir en kort presentasjon av disse for å sette oppgaven i en større sammenheng.

1.2 Disposisjon

I kapittel 2 blir bakgrunnen for prosjektet, fagområdet og litteraturen presentert. Flere eksempler på datastrukturer som har drevet utviklingen fremover blir også gitt. Kapittel 3 på side 15 presenterer grunnforståelsen av det spatiale. Både for data og tilhørende operasjoner. I kapittel 4 følger en detaljert gjennomgang av R-treets struktur og funksjon. I kapittel 5 presenteres min implementasjon og en ny måte å håndtere nodedelinger på. Hensikten er å håndtere problemene rundt overlapp mellom noder, noe som er et resultat av dårlige delinger. Kapittel 6 presenterer resultatet av testene utført på treet, mens kapittel 7 konkluderer og presenterer forslag til videre arbeid.

Kapittel 2

Bakgrunn

Denne oppgaven ble påbegynt i forbindelse med det europeiske prosjektet Diligent. I utgangspunktet skulle Diligent legge til rette for å finne et interessant tema for oppgaven. Særlig spatial indeksering fremstod som spennende. Dette er et fagfelt som har hatt høy aktivitet de siste 20 årene, men som fremdeles er høyst aktivt. Originalt var tanken fra FAST sin side å benytte javabiblioteket GeoTools. GeoTools ble originalt utviklet ved Universitetet i Leeds i 1996 som en del av en masteroppgave. Dette biblioteket har vokst seg stort siden den tid, og implementerer standardene fra OGC, Open Geospatial Consortium (se `http://www.opengis.org`), fortløpende. GeoTools sin implementasjon har noen svakheter som bør nevnes. For det første er det en ren implementasjon av R-treet. Den implementerer ikke heurestikkene til R*-treet (som blir omtalt senere). Samtidig har de gjort noen implementasjonsmessige valg som fører til dårlig utnyttelse av minnet, blant annet bruken av foreldrepekere.

Det ble etterhvert klart at man ønsket en egen implementasjon. Det sammenfallt derfor godt med min implementasjon som et ledd i et Diligent sitt behov for et R-tre, og denne oppgaven om nodedelinger.

2.1 Diligent

Her følger en kort redegjørelse om prosjektet Diligent. Min implementasjon kommer til å bli benyttet i dette systemet. Det er derfor interessant å kort oppsumere hva dette systemet består i.

Diligent, akronym for "DIgital Library Infrastructure on Grid ENabled Technology", er et forskningsprosjekt innenfor den europeiske unionen. Det består av organisasjoner, bedrifter og universiteter. Prosjektets formål er å kombinere digitale biblioteker med grid-teknologi. Sluttproduktet blir et

distribuert system som tilbyr opprettelse av virtuelle "on-demand" digitale biblioteker. Man kan da spesifisere hvilke tjenester og behov man har. Et av scenariene systemet initielt skal støtte er impect-scenariet. Dette ligger i domenet de kaller "environmental e-science". Det er dette scenariet som behøver støtte for indeksering av spatiale data, og min implementasjon av R-treet.

2.2 GIS - Geografiske informasjonssystemer

Indeksering av spatiale data sees ofte i sammenheng med geografiske informasjonssystemer. Under følger en redegjørelse om slike systemer.

GIS, akronym for geografiske informasjonssystemer, blir av AGI (`http://www.geo.ed.ac.uk/agidexe/term?271`) beskrevet som et datasystem til å innhente, lagre, sjekke, integrere, manipulere, analysere og fremvise data relatert til posisjoner på jordens overflate. Dette kan også beskrives som den høyteknologiske versjonen av kart.

Vi har gode eksempler på utnyttelse av geografiske informasjonssystemer i Norge. Det norske selskapet Geodata som er ledende leverandør av slike systemer i Norge, hjelper blant annet politiet med spesialiserte GIS-applikasjoner for å kartlegge voldshendelser. På denne måten kan politiet ekstrapolere hendelser i tid og utplassere patruljer i forkant på steder man antar at et økt behov vil oppstå. Geografiske informasjonssystemer viste seg viktig i arbeidet myndighetene hadde i Thailand i forbindelse med tsunamien julen 2004.

Diligent kommer ikke til å representere et generellt GIS-verktøy, men en spesifikk GIS-applikasjon for scenariet gitt ved prosjektstart. Her ser man for seg en ulykke i Atlanterhavet, der en oljetanker velter og oljen renner ut. Ved hjelp av SPOT-satelitter som kontinuerlig mater inn bilder av jorden vil man kunne søke i en spatial temporal sammenheng for å se oljeflakets utvikling.

I et geografisk informasjonssystem kan det ligge mye informasjon. Eksempler på typisk informasjon vil være byer, hus, elver, boligfelt, bedrifter, vannnettverk, strømnettverk og lignende. Så lenge noe kan knyttes til lokalitet, vil det kunne være i et geografisk informasjonssystem. Et geografisk informasjonsysstem er avhengig av databasesystemer som kan håndtere spatiale data[14]. Nøyaktig hva som menes med databasesystem varierer, i noen tilfeller kan det bety en samling flate filer, mens i andre en fullverdig DBMS.

2.3 Indeksering av spatiale data

Effektive søk i databaser er avhengig av støtte på et lavt nivå. Dette gjelder like mye for spatiale databaser som for vanlige databaser. I litteraturen blir teknikkene som understøtter denne effektiviteten omtalt som flerdimensjonale indekser, spatiale indekser, spatiale aksessmetoder eller flerdimensjonale aksessmetoder. Vi kan enkelt si at slike spesifikke aksessmetoder beror på en underliggende datastruktur kalt en indeks. Formålet til indeksene er å redusere mengden objekter et søk må traversere for å utføre en spørring. Hvilke spørringer en struktur understøtter er også av interesse. Det er derfor viktig at dataene organiseres på en slik at måte at den typen spørringer vi interesserer oss for, er det som effektiviseres. De vanligste typene spørringer i en spatial sammenheng er:

- Områdespørringer
- Punktspørringer
- "Nærmeste nabo"-spørringer

R-treets struktur, som blir gjennomgått senere, støtter alle disse formene for spørringer.

Det er en rekke kriterier spatiale aksessmetoder skal dekke[9]:

1 *Dynamisk.* Ettersom objekter blir satt inn og slettet fra databasen i en hvilken som helst rekkefølge, må aksessmetodene kontinuerlig kunne holde tritt med forandringene.

2 *Sekundær lagringshåndtering.* Til tross for økende minne, er det ikke alltid mulig å holde hele databasen i minnet. Derfor må aksessmetoder integrere sekundær lagring sømløst.

3 *Bred støtte for operasjoner.* Aksessmetoder bør ikke støtte kun én type operasjon (f.eks søk) på bekostning av andre operasjoner (f.eks sletting).

4 *Uavhengighet av data og innsettingsrekkefølge.* Aksessmetoder skal ivareta effektiviteten uavhengig av data som er irregulære eller at innsettingsrekkefølgen forandres. Dette er spesielt viktig for data som er distribuert forskjellig langs de forskjellige dimensjonene.

5 *Enkelhet.* Intrikate aksessmetoder med mange spesielle løsninger er ofte utsatt for feil ved implementasjon og er derfor ikke tilstrekkelig robuste for applikasjoner med stor utbredelse.

6 *Skalerbarhet.* Aksessmetoder bør kunne tilpasse seg voksende databaser.

7 *Tidseffektivitet.* Spatiale søk skal være raske. Et primært mål må være å tangere det én-dimensjonale B-treet; Aksessmetodene burde garantere logaritmisk "worst-case" søk for alle mulige datadistribusjoner uavhengig av innsettingsrekkefølgen.

8 *Plasseffektivitet.* En indeks skal være liten i størrelse sammenlignet med dataene de beskriver og derfor kunne garantere en viss plassutnyttelse.

9 *Samtidighet og gjenopprettelse.* I moderne databaser hvor mange brukere samtidig oppdaterer, henter og setter inn data, må aksessmetodene kunne tilby robuste teknikker for transaksjonshåndtering uten særlig tap av effektivitet.

10 *Minimal påvirkning.* Integrasjonen av en aksessmetode i et databasesystem burde ha minimal påvirkning på eksisterende deler av systemet.

Generelt kan man dele inn forskjellige spatiale indekser i to hovedkategorier[5]. Data-organiserende strukturer og rom-organiserende strukturer. Eksempel på den førstenevnte gruppen er R-treet[10, 20, 2] mens den sistnevnte gruppen blant annet har GRID-files og kd-tre baserte metoder. Det er ikke alltid en slik inndeling er så enkel grunnet mengden komplekse og hetrogene struktur som er utviklet. I [9] velges en kategorisering basert på dataenes natur, henholdsvis PAM (Point Access Method) og SAM (Spatial Access Method). Et viktig poeng er at disse abstraksjonene ikke nødvendigvis medfører fysisk korrekthet. Eksempelvis indekserer R-treet også punkter så vel som objekter av utstrekning, mens objekter av utstrekning kan bli sett på som punkter i et høyere dimensjonalt rom. Det er altså intensjonen bak strukturen det skilles etter.

- *Rom-organiserende strukturer:* Baserer seg på å partisjonere det underliggende rommet i celler, uavhengig av distribusjonen av objekter (punkter eller MBR) i planet. Objekter blir koplet til en gitt celle.

- *Data-organiserende strukturer:* Partisjonerer mengden objekter fremover det underliggende rommet. Partisjoneringen tilpasser seg distribusjonen av objekter i rommet.

I 1998 gjorde Gaede og Günther[9] en utvidet undersøkelse og oppsummering av flerdimensjonale aksessmetoder. Selv om det har kommet en ny teori siden den gang er dette fortsatt en av de bredeste undersøkelser man

Navn	Type	Sekundær lagring
k-d-Tree	Rom-organiserende	Ram-basert
BSP-Tree	Rom-organiserende	Ram-basert
BD-Tree	Rom-organiserende	Ram-basert
Quad-Tree	Rom-organiserende	Ram-basert
k-d-B-Tree	Rom-organiserende	Sekundær lagring
Linear Hashing	Rom-organiserende	Sekundær lagring
Grid files	Rom-organiserende	Sekundær lagring
LSD-Tree	Rom-organiserende	Sekundær lagring
Buddy Tree	Rom-organiserende	Sekundær lagring
BANG file	Rom-organiserende	Sekundær lagring
hB-Tree	Rom-organiserende	Sekundær lagring
BV-Tree	Rom-organiserende	Sekundær lagring
R-Tree	Data-oganiserende	Sekundær lagring
R*-Tree	Data-oganiserende	Sekundær lagring
R+-Tree	Data-oganiserende	Sekundær lagring
SP-Tree	Data-oganiserende	Sekundær lagring
JP-Tree	Data-oganiserende	Sekundær lagring
X-Tree	Data-oganiserende	Sekundær lagring
SKD-Tree	Hybrid	Sekundær lagring
GBD-Tree	Data-oganiserende	Sekundær lagring

Tabell 2.1: Taksonomi av en rekke strukturer

har. I tabell 2.1 på forrige side kan man se et utvalg av strukturene artikkelen tar for seg som viktige.

Listen i 2.1 på forrige side er på langt nær komplett. Det er lagt ned mye arbeid i finne forbedrede strukturer til indeksering av spatiale data. Det er også komplisert å sammenligne alle for å velge en optimal struktur. Selv om mange komparative studier har blitt utført, så viser det seg vanskelig å utnevne en struktur som overlegen når man ikke lett kan definere *optimal*. En gitt struktur kan være god på et område, men dårlig på et annet. Det viser seg også at noen studier får gode resultater når de undersøker en struktur mens en senere studie ikke klarer å bekrefte disse funnene. En mulig forklaring kan være bruk av ulike datasett, testmetodologi og lignende.

For denne oppgaven er spesielt R-treet og X-treet viktige strukturer. Disse vil derfor bli gjennomgått nærmere nedenfor.

2.3.1 Rom-organiserende strukturer

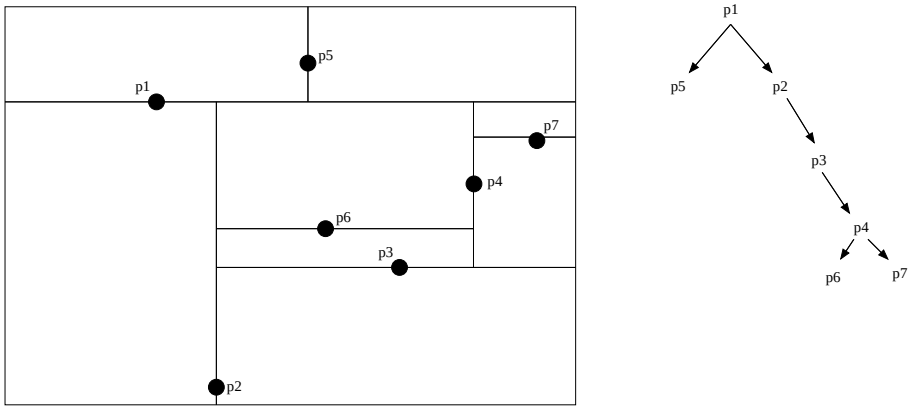
I dette avsnittet blir Grid files og k-d-treet gjennomgått. Dette er to av de mer fremtredene strukturene, og er gode representanter for klassen rom-organiserende strukturer. Som vi kommer inn på senere står ikke disse strukturene i direkte motsetning til de data-organiserende strukturende, men det er interessant med et overblikk over hvordan disse strukturene søker å indeksere spatiale data i forhold til de data-organiserende strukturene.

Grid files

Denne strukturen er kjennetegnet ved at søkerommet deles inn i rektangulære celler. Det resulterende rutenettet er en $n_x * n_y$ array av celler med lik størrelse. Hver celle c er asossiert med en disk page. Punktet P blir plassert i celle c hvis rektangelet asosiert med celle c inneholder P. Alle objekter som koples til celle c blir lagret sekvensielt i pagen asosiert med c. Indeksen krever en 2-dimensjonal array $DIR[1 : n_x, 1 : n_y]$ som oppslagskatalog. Hvert element DIR[i,j] i katalogen inneholder adressen PageID til pagen som lagrer punktene til celle $c_{i,j}$.

k-d-treet

k-d-treet ble først beskrevet av Bentley[3]. Treet er et binært søketre som rekursivt deler opp rommet ved hjelp av $(d-1)$-dimensjonale hyperplan. Retningen på hyperplanene alternerer mellom de d mulighetene. For et eksempel med $d = 3$ vil delende hyperplan alternere perpendikulært til x, y og z-aksen.



Figur 2.1: k-d-tre med $d = 2$

Hvert hyperplan må inneholde minst ett datapunkt som blir brukt som representasjon i treet. I figur 2.1 kan vi se et k-d-tre der $d = 2$, lastet med 7 punkter.

Punktene har blitt lastet i numerisk rekkefølge (p1 - p7). Vi kan her se hvordan første punktet, p1, delte planet i 2 langs x-aksen. Mens p2 deretter delte det nye planet (det definert av p1 sin deling) i to langs y-aksen. Mer enkelt er hvert hyperplan en representant for sitt punkt. En betydelig sårbarhet med k-d-treet er at det er sensitivt overfor rekkefølgen punktene blir satt inn.

2.3.2 Data-organiserende strukturer

Nedenfor gis en en kort introduksjon til R-treet, samt medlemmer i den utvidede familien, og X-treet. R-treet blir gjennomgått i detalj senere, men det er greit med en kort introduksjon før vi ser på X-treet, siden dette er basert på førstnevnte. Oppgaven i helhet baserer seg på R-treet med en utvidelse inspirert av X-treet, det er derfor nødvendig med en grundig gjennomgang av disse.

R-treet

Data-organiserende strukturer har blitt synonymt med R-tre familien. Det har blitt laget så mange derivater av R-treet at vi kan snakke om en hel skog. R-treet er en hierarkisk høydebalansert struktur som partisjonerer opp dataene ut i fra spatiale egenskaper. Dets struktur vil bli nøye gjennomgått i kapittel 4 på side 23. I all hovedsak er strukturen lik B-treet, men

nøklene dekker ikke kun en 1-dimensjonal rekkevidde. R-treet har vokst seg veldig populært og dets variant R*-treet har blitt en de-facto standard i bransjen[13].

Det er utenfor denne oppgaven å ta for seg alle variantene, men interesserte lesere henvises til *R-Trees: Theory and Applications (Advanced Information and Knowledge Processing)*[13] for en bred gjennomgang.

Hilbert R-treet

Hilbert R-treet[11] er en hybrid struktur basert på R-treet og B+-treet. Mer spesifikt er det et B+-tre med geometriske objekter karakterisert av Hilbert-verdien til centroiden. Strukturen er basert på "Hilbert space-filling curve". Oppføringer i trenoder blir tildelt den største Hilbert verdien av sine barn. En gitt oppføring blir derfor representert av tupelet (mbr,H,p) hvor mbr er en vanlig mbr som omfatter barna til noden, H er den største Hilbert verdien til barna og p er pekeren nedover i treet. Oppføringer i bladnoder er akkurat som i R-treet.

Algoritmen for områdespørringer er identisk med R-treet. Man starter fra roten og synker ned i treet ved å sjekke om søke-MBR overlapper med oppføringenes MBR-er. Det som skiller seg ut er hvordan innsettingen foregår. Ved innsetting av objekt E blir Hilbert-verdien H av centroiden til rektangelet regnet ut. Deretter brukes H til å styre oppføringen nedover i treet. For hvert nivå sjekker man H mot de inneværende oppføringenes H. Den minste verdien som er større enn H blir valgt.

Som vi nå ser blir det en rangering av oppføringer i nodene på alle nivåer. Resultatet av dette er at treet kan utnytte B+-treets egenskaper. Alle søsken på et gitt nivå er sortert etter Hilbert-verdien. Bare det å kunne definere søsken er annerledes enn R-treet. Dette fører til at når en node overflyter kan man utsette delingen og prøve å sette inn noen av oppføringene i de nærmeste søskenene på dette nivået. En deling blir kun utført når alle søsken er fulle. Dette gjør at Hilbert R-treet maksimerer utnyttelsen av plassen og unngår unødvendige delinger.

Selv om forfatterne selv hevder at Hilbert R-treet beviselig er den beste dynamiske versjonen av R-treet (på tidspunktet artikkelen ble skrevet) påpeker Manolopoulos og Nanopoulos[13] at denne varianten er sårbar for store objekter. Og ved å øke til flere dimensjoner blir ikke nærhet bevart tilstrekkelig av Hilbert-kurven, noe som fører til økende overlap i interne trenoder.

X-treet

X-treet ble presentert i 1996 som en løsning på problemene rundt indeksering av data med et høyt antall dimensjoner. Selve strukturen er inspirert av de R-tre baserte strukturene, hovedsaklig fordi man anerkjenner behovet for å kunne indeksere både punkter og objekter med ekstensjon i et høyere antall dimensjoner. Spesielt nevnes det økende behovet for å kunne indeksere egenskapvektorer. En egenskapvektor er når man abstraherer et objekt for å trekke ut visse egenskaper fra det. Eksempelvis for bilder kan vi trekke ut fargehistogrammet. Et måte å representere et bilde på i en datamaskin er fargemodellen RGB (Red, Green, Blue). Et gitt piksel beskrives da som et nivå mellom 0 og 255 av disse tre fargene. Tuppelet (128,128,128) vil f.eks være fargen grå. Dette kan også tolkes som et punkt i et tredimensjonalt rom. Punkter i nærheten vil nødvendigvis ha likheter i sitt fargehistogram.

X-treets fortrinn er å unngå overlappende MBR-er i katalognoder (noder som ikke er bladnoder). Fremfor å tillate delinger som fører til stort overlapp utvides katalognodene til supernoder som er dobbelt så store. Det blir påpekt at selv om den lineære søkingen i en så stor node kan virke som et problem, så er alternativet (stort overlapp) mye verre. Den viktigtste metoden i et X-tre er innsetting. Det er denne metoden som bestemmer formen på treet. Innsettingen foregår på følgende måte:

- 1. [Finn korrekt bladnode] Metoden jobber seg rekursivt nedover i treet for å finne riktig bladnode. Hvis det er plass, settes objektet inn og avslutter.

- 2. [Topologisk deling] Hvis innsettingen fører til at noden må deles prøver man først å finne en løsning basert på topologiske og geometriske egenskaper. R*-treets heurestikk er et godt eksempel på deling man kan benytte i dette steget. Hvis man finner en god deling utfører man denne og avslutter.

- 3. [Deling basert på historikk] Hvis den topologiske delingen fører til et stort overlapp, vil man forsøke å finne en minimalt overlappende deling basert på delingshistorikk (hvilke dimensjoner det tidligere har blitt delt over). Denne partisjoneringen kan føre til underfyllte noder, noe som er uakseptabelt. Hvis partisjoneringen er akseptabel, avslutter man.

- 4. [Lag supernode] Om alt annet feiler, utvider man noden til dobbel standard blokkstørrelse. Hvis noden allerede er en supernode utvider man med én blokkstørrelse.

Empiriske forsøk viser at X-treet er overlegent i forhold til R*-treet på et høyere antall dimensjoner, og like godt eller bedre på et lavt antall dimensjoner[4].

2.3.3 Dimensjonale problemer

I databaseverdenen benytter man uttrykket ”curse of dimensionality“ om indeksering av flerdimensjonale data. Dette henspeiler på en rekke problemer som oppstår når dataene overstiger et visst antall dimensjoner. Generelt kan man si at problemet består av at en rekke viktige egenskaper for et objekt, slik som volum og område er eksponensielt avhengig av antall dimensjoner. Dette fører til at de fleste indeksstrukturer, er effektive kun ved et lavt antall dimensjoner[5]. Berchtold og Bohm presenterer følgende problemer[5]:

- Rent geometriske effekter ved overflaten og volumet til (hyper)kuber og (hyper)kuler:

  Volumet til en kube vokser eksponensielt med økende dimensjoner (og konstant kantlengde).

  Volumet til en kule vokser eksponensielt med økende dimensjoner

  Mesteparten av volumet til en kube er veldig nært det $(d-1)$-dimensjonale overflaten til kuben.

- Effekter på formen og plasseringen til indekspartisjoner

  En typisk indekspartisjon i et høyt antall dimensjoner dekker mesteparten av datarommet i de fleste dimensjoner og vil bare være delt i noen få dimensjoner.

  En typisk indekspartisjon er ikke kube-formet, men vil se mer ut som et rektangel.

  En typisk indekspartisjon rører yttergrensene av dataene i de fleste dimensjoner

  Delingen av rommet blir grovere desto fler dimensjoner vi har.

For R-treet betyr dette at man får veldig stort overlapp over tid, som resultat av veldig mange dårlige delinger. For å bøte på problematikken rundt mange dimensjoner ble X-treet[4] presentert. I korte ord presenterer Berchtold med fler her noe de kaller for supernoder som kan være dobbelt så store som vanlige noder. Når en node blir full, gjøres det tester for å finne ut om man kan gjøre en tilfredstillende deling. Når dette ikke er mulig, blir noden konvertert til en supernode slik at man kan tillate flere oppføringer.

Forhåpentligvis kan man da - neste gang den blir full - gjøre en god deling. I tillegg presenteres noe forfatterne kaller for delingshistorie. Ut i fra historiske data om tidligere delinger kan man finne den dimensjonen som fører til minst overlapp med andre partisjoner. Dette er helt essensielt for de gode empiriske resultatene treet har fått. X-treet var en av de største inspirasjonskildene til arbeidet med "Lazy Split", jf. avsnitt 5.2.2 på side 44.

Området spatial indeksering er nå et stort felt, der mange prøver å løse forskjellige problemer. I all hovedsak består feltet av tre forskjellige områder. De tidligste strukturene fokuserte på å håndtere lavere dimensjonalitet, være seg punkter eller objekter med ekstensjon. Vi finner en stor mengde strukturer og forandringer på disse som utelukkende søker å forbedre eksisterende strukturer, slik mitt arbeide gjør. De siste årene har flertallet av strukturer presentert søkt å løse problematikken rundt høyere dimensjonalitet (grunnet tilveksten av applikasjoner som behøver å utføre avanserte søk blant bilde, video og andre medier).

2.4 Sammenfatning

Som vi har sett fra Hilbert R-treet og X-treet, er det et betydelig fokus på å minimere overlapp blant nodene i disse indeksstrukturene. Det fremstår som klart at for å kunne tilfredstille kravene om skalerbarhet og tidseffektivitet kan man ikke tillate en struktur som i verste fall lar deg stige ned i alle noder i treet uten å få noen treff.

Der X-treet velger å følge en linje som ligger nærmere R-treet, ved blant annet å utvide nodene, tenker Hilbert R-treet utenfor boksen. Med R-treet som utgangspunkt har disse to strukturene (blant mange andre) sett på teknikker rundt innsetting og nodedeling med hensikt å konservere en struktur som ikke forringes over tid.

Det er spesielt strukturer som prøver å løse problemene rundt høyere dimensjonalitet, som fokuserer på dette. Grunnen er at ved eksempelvis 2 dimensjoner er problemet nesten ikke merkbart sammenlignet med eksempelvis 30 dimensjoner. Vi skal se en mulig måte å optimalisere R-tre strukturen i kapittel 5 basert på X-treet. Her vil alle ytelsestester bli utført på 2-dimensjonale data, siden det er dette som er relevant i forhold til bruken av treet.

Kapittel 3

Generelle spatiale konsepter

> *The introduction of suitable abstractions is our only mental aid to organize and master complexity.*
> (E. W. Dijkstra)

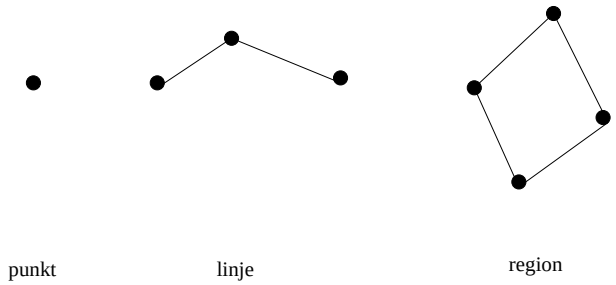
Dette kapittelet tar for seg spatiale data og spatiale operatorer. Det er viktig med en grunnforståelse av disse konseptene før man beveger seg videre. Oppgaven tar for seg overlapp mellom noder i R-treet. Dette overlappet må sees i en spatial sammenheng.

3.1 Spatiale data

Man kan dele inn spatiale data i to hovedabstraksjoner. Objekter / entiteter og kontinuerlige områder. Begge disse abstraksjonene kan igjen modelleres med vektor- eller rasterdata.

Spatiale data er data med romlige egenskaper. Spatiale data handler om hvordan man best modellerer den virkelige verden. Spatiale datatyper (SDT) kan deles i to sett, raster- og vektordata. Rasterdata representeres ved bruk av kvadratiske celler som har en verdi utifra egenskapene til dette området. Ofte ser vi Rasterdata i form av bilder. Slike data egner seg godt til å modellere kontinuerlige data som stigninger, økende konsentrasjoner og lignende, men også til å bruke opp diskplass.

Vektor-data modellerer objekter med datatyper som punkt, linje og region, se figur 3.1 på neste side. Et punkt befinner seg i et flerdimensjonalt rom, men har ingen ekstensjon, og kun dets lokasjon er viktig. En linje er en sammenkopling av flere slike punkter og brukes typisk til å modellere elver og veier. Region er en abstraksjon for objekter hvor deres romlige ekstensjon er viktig, slik som byer, skoger og innsjøer.



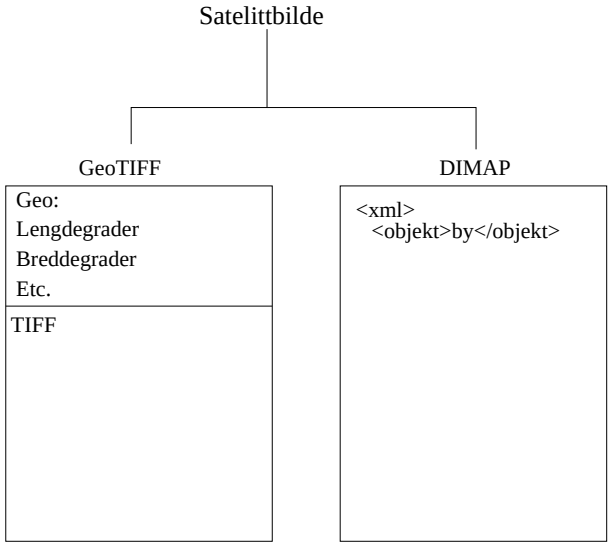
Figur 3.1: Vektor datatyper: Punkt, linje og region

I Diligent-prosjektet kommer disse dataene i form av satelittbilder (Raster-data), noe som gir en optimal approksimasjon i forhold til det å benytte MBR. Man kan lese mer om denne approksimasjonen i avsnitt 4.1 på side 24. Bildene vil være representert som et par; bildefil og metadata, se figur 3.2 på neste side. Bildefilen baseres på geoTiff-formatet som igjen inneholder en bildedel og en ekstra geo-del. Denne geo-delen kan beskrive ting som geodetiske egenskaper (lengdegrader og breddegrader), projeksjon og lignende. Satelittbildene kan igjen inneholde objekter som byer, elver og broer. Slike egenskaper blir da beskrevet i xml ved hjelp av formatet DIMAP, Digital Image Map, som er utviklet av Spot Images i samarbeid med CNES (Centre National d'Eudes Spatiales). Disse dataene er vektor-data som punkt, linje og region. Det er her behov for å utføre en transformasjon som beskriver objektets MBR.

3.2 Spatiale topologiske relasjoner

Relasjoner mellom spatiale objekter er et felt mange har bidratt i, fra det intuitive til det formelle. Predikater slik som "under" og "møter" er ikke nødvendigvis opplagte. Det er derfor et behov for en formell definisjon for spatiale relasjoner. Topologiske relasjoner er bare en delmengde av uttallige spatiale relasjoner, og blir ivaretatt under transformasjoner som rotasjon og skalering[1]. Videre diskusjon baserer seg derfor på topologiske relasjoner mellom to objekter i planet. En intuitiv og vanlig formalisme baserer seg på ren mengdelære. Med bruk av funksjon $point(x)$, som beskriver mengden punkter som hører til en spatial region x, og samlingen mengdeoperatorer $=$, $\neq$, $\subseteq$ og $\cap$ kan vi beskrive følgende forenklede definisjon:

- $x = y$ tilsvarer $point(x) = point(y)$
- $x \neq y$ tilsvarer $point(x) \neq point(y)$



Figur 3.2: Organisering av DIMAP

- x *inni* y tilsvarer $point(x) \subseteq point(y)$
- x *utenfor* y tilsvarer $point(x) \cap point(y) = \emptyset$
- x *snitt* y tilsvarer $point(x) \cap point(y) \neq \emptyset$

Som vi ser er denne definisjonen verken minimal eller komplett. Predikatene *lik* og *inni* er begge inkludert i snitt-definisjonen. *Overlapper* og *møter* dekkes også av snitt-definisjonen. Det er disse problemene som førte til at man utvidet definisjonen med indre og kant av spatiale objekter slik at *overlapper* (ikke-tomt snitt av kant og ikke-tomt snitt av indre) og *møter*(ikke-tomt snitt av kant og tomt snitt av indre) kan skilles. I 1991 gav Egenhofer[16] ut sin artikkel om punktmengder, og det har blitt den mest innflytelsesrike modellen. Begrepene for å definere topologiske relasjoner er altså: indre, kant og ytre.

- Indre av mengden A skrives A^o
- Kant av mengden A skrives δA
- En punktmengde S i $\mathbb{R}^2$ er *åpen* hvis det for hvert av punktene p i S eksisterer en $\epsilon \in \mathbb{R}, \epsilon > 0$, slik at platen med radius ϵ og sentrum p er inneholdt i S.
- S er lukket hvis $\mathbb{R}^2 - S$ er åpen.

Tabell 3.1: De åtte mulige relasjonene mellom indre og kant av 2 spatiale regioner

$\delta A \cap \delta B$	$A^o \cap B^o$	$\delta A \cap B^o$	$A^o \cap \delta B$	Relasjonsnavn
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	*A ikke lik B*
$\neg\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	*A møter B*
$\neg\emptyset$	$\neg\emptyset$	$\emptyset$	$\emptyset$	*A lik B*
$\emptyset$	$\neg\emptyset$	$\neg\emptyset$	$\emptyset$	*A inni B*
$\neg\emptyset$	$\neg\emptyset$	$\neg\emptyset$	$\emptyset$	*B dekker A*
$\emptyset$	$\neg\emptyset$	$\emptyset$	$\neg\emptyset$	*B inni A*
$\neg\emptyset$	$\neg\emptyset$	$\emptyset$	$\neg\emptyset$	*A dekker B*
$\neg\emptyset$	$\neg\emptyset$	$\neg\emptyset$	$\neg\emptyset$	*A overlapper B*

Med disse definisjonene har vi verktøyene vi behøver for å finne ut hvilke relasjoner som finnes mellom to spatiale regioner. Gitt at vi vet kant og indre av objektene behøver vi bare prøve ut om forskjellige snitt er tomme eller ikke. Det er seksten forskjellige kombinasjoner, men i $\mathbb{R}^2$ er det bare åtte som kan finne sted gitt forutsetningen om at regioner ikke kan ha hull. I tabell 3.1 kan man se de åtte relasjonene basert på indre og kant. Videre ser vi relasjonene visualisert i figur 3.3 på neste side.

3.2.1 Spatiale operatorer

Det er mange forskjellige spatiale operatorer som er interessante i forhold til spørringer, under følger en kort oversikt over de mest vanlige, kategorisert etter operator type (unær, binær) og resultat.
Unære spatiale operatorer:

- Unære operatorer med bolsk resultat

  Tester et spatialt objekt for en gitt egenskap, f.eks konveksitet

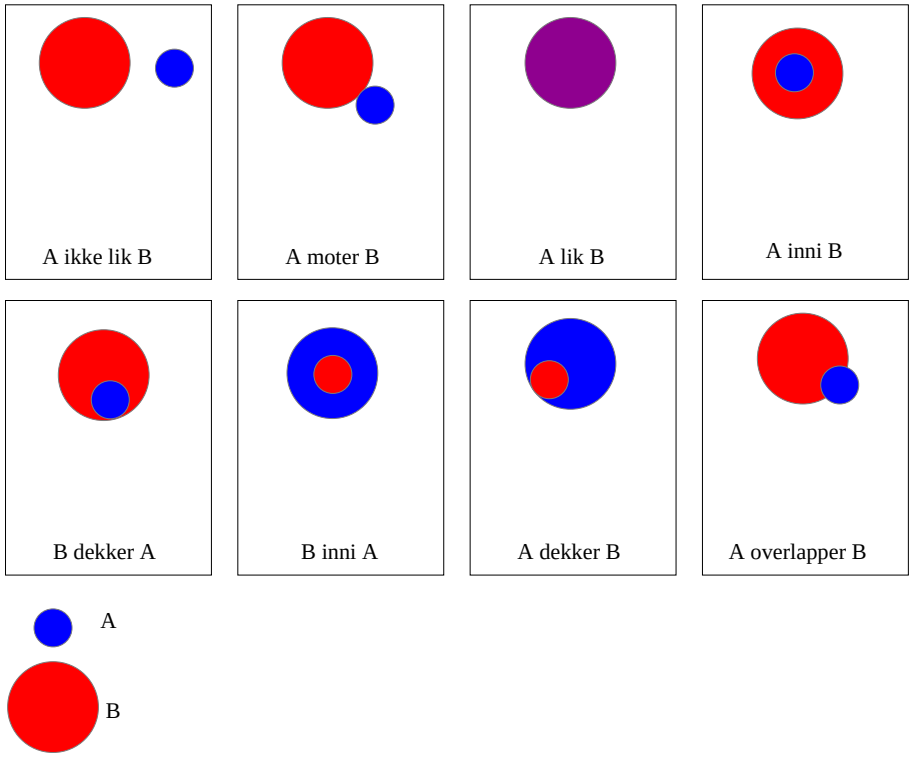
- Unære operatorer med skalart resultat

  Areal av et gitt objekt

- Unære operatorer med spatialt resultat

  Mange typer, f.eks transformasjoner (rotasjon), retnings predikater (nord for, vest for)

Binære spatiale operatorer:



Figur 3.3: De 8 mulige relasjonene realisert av indre og kant

Tabell 3.2: Kopling mellom MBR relasjon og mulig eksakt relasjon

MBR relasjon	Mulige geometrisk relasjoner
ikke lik	ikke lik
møter	ikke lik $\vee$ møter
overlapper	ikke lik $\vee$ møter $\vee$ overlapper
inni	ikke lik $\vee$ møter $\vee$ overlapper $\vee$ inni $\vee$ dekket av
dekket av	ikke lik $\vee$ møter $\vee$ overlapper $\vee$ dekket av
inneholder	ikke lik $\vee$ møter $\vee$ overlapper $\vee$ inneholder $\vee$ dekker
dekker	ikke lik $\vee$ møter $\vee$ overlapper $\vee$ inneholder $\vee$ dekker
lik	lik $\vee$ overlapper $\vee$ dekket av $\vee$ dekker $\vee$ møter

- Binære operatorer med bolsk resultat

  x isWithin y

- Binære operatorer med spatialt resultat

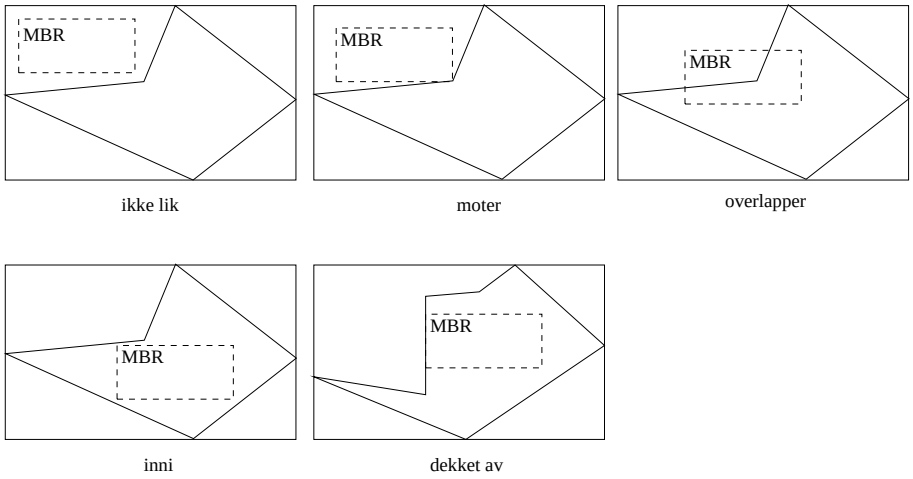
  x Intersection y

- Binære operatorer med skalart resultat

  x Distance y

Alle disse spatiale operatorene har et forhold til approksimasjonen MBR, jf. avsnitt 4.1 på side 24. Siden et MBR kun approksimerer objekter er det mange mulige eksakte geometriske forhold for et gitt MBR forhold. I tabell 3.2[18] kan man se hvordan relasjonene korresponderer til hverandre.

For å lettere forstå denne sammenhengen kan man i figur 3.4 på neste side se forholdene for relasjonen *inni* forholder seg til det eksakte spatiale objektet. Vi ser hvordan de fem mulighetene kan tenkes å se ut. For å forenkle tegningen vises den ene MBR'en uten sitt spatiale objekt. Dette ville bare ha komplisert forklaringen, uten å ha tilført noe mer.



Figur 3.4: Mulige eksakte geometriske relasjoner for MBR relasjon *inni*

Kapittel 4

Beskrivelse og analyse av R-Treet

Dette kapittelet tar for seg R-treet. Vi får se hvordan strukturen er bygget opp og operasjonene utføres. Dette vil legge platformen for implementasjonen.

Egenskaper fra Guttmans originale artikkel om R-treet[10]:

La M være maks antall barn en node kan ha, og $m \leq M/2$ være minste antall barn. Da tilfredstiller et R-tree følgende egenskaper:

1 Hver blad-node har mellom m og M oppføringer med mindre det er rotnoden.

2 For hver oppføring (I, $tuppel - identifikator$) i en bladnode beskriver I det minste omsluttende rektangelet for det n-dimensjonale objektet pekeren referer til.

3 Hver node som ikke er en en blad-node har mellom m og M oppføringer med mindre det er rotnoden.

4 For hver oppføring (I, $barne-peker$) i en node som ikke er en blad-node er I det minste rektangelet som omfatter rektanglene i dets barn.

5 Rot-noden har minst 2 oppføringer med mindre den også er en blad-node.

6 Alle blad-noder er på samme nivå.

Fra dette kan vi konkludere følgende:

- m må velges slik at 2≤m≤ $\lfloor M/2 \rfloor$

- Høyden til et tre med N oppføringer vil på det meste være $|\log_m N|$-1
- Høyden til et tre med N oppføringer vil på det minste være $|\log_M N|$-1

Et R-tre organiserer spatiale data dynamisk i sin hierarkiske struktur. Hver node i treet representerer rektangelet som omfatter alle dets barn. Noden er normalt tenkt å passe en disk page, men dette kommer helt an på bruk. Figur 4.1 på neste side viser noen spatiale objekter $A-1$, $A-2$, $A-3$, $A-4$, $A-5$, $B-1$, $B-2$, $B-3$, $B-4$, $B-5$, $B-6$, $C-1$, $C-2$, $C-3$, $C-4$ og $C-5$ strukturert i treet. Vi ser de tre interne nodene A, B og C som har delt inn dataene i passende rektangler.

I figur 4.2 på side 26 kan man se en visualisering av R-treet. Dette R-treet har blitt lastet med data fra det tyske veinettet. Alle blå bokser indikerer noder. Rød bokser indikerer data. Det oppmuntrende er selvfølgelig å se at dette faktisk ser ut som et ekte kart over Tyskland. I figur 4.3 på side 27 kan vi se det visualiserte R-treet projisert på Tyskland gjennom Google Earth. Google Earth har her tatt høyde for korrekt projisering (kurvingen av jordklodens overflate, i motsetning til et plan). Denne visualisering ble laget med visualiseringspakken laget for implementasjonen. Man kan lese mer om denne i avsnitt 5.1.1 på side 41.

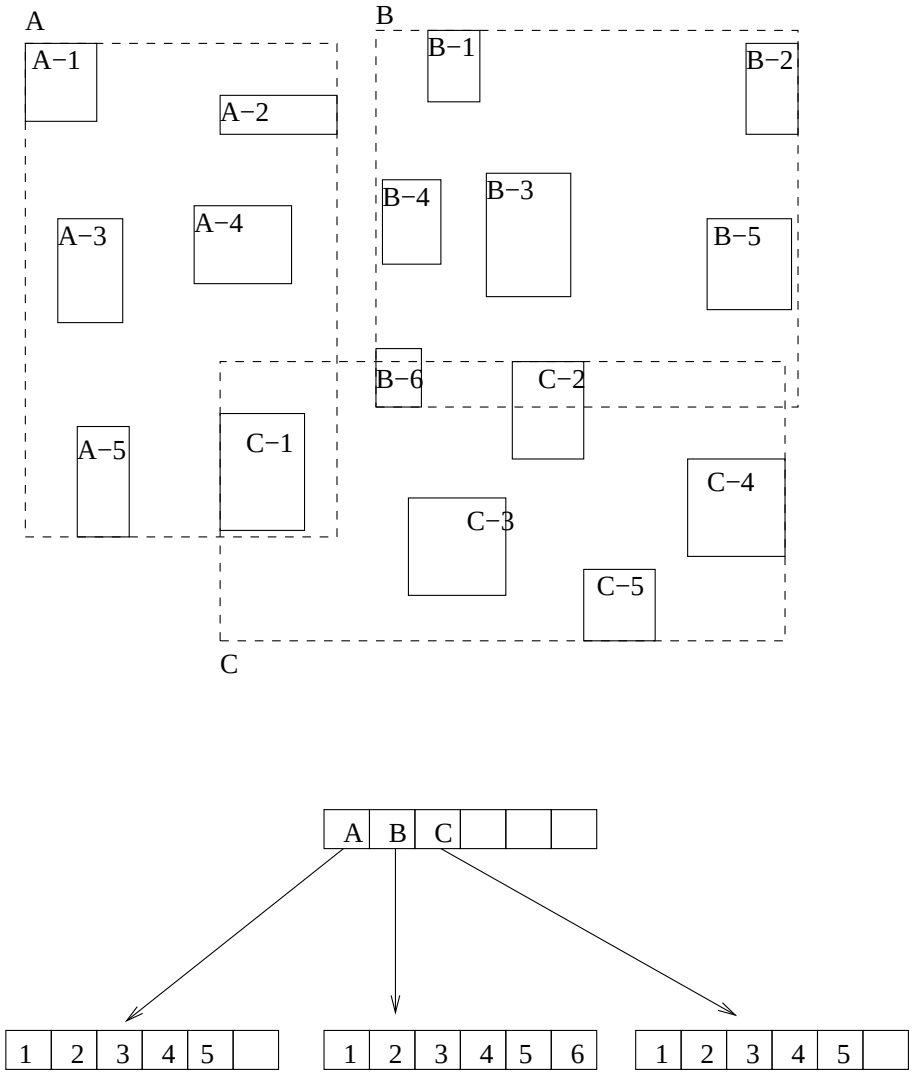
4.1 MBR - Minimal Bounding Rectangle

Generelt uttrykt er det n-dimensjonale rektangelet, n-rektangelet, det minste omsluttende n-rektangelet for et gitt n-dimensjonalt objekt. Et MBR er altså en god approksimasjon på det virkelige spatiale objektet. I figur 4.4 på side 27 kan man se hvorledes rektangelet omslutter det spatiale objektet, her en region. Bruken av MBR er altså en del av en "filter og refine"-strategi hvor man først finner objekter som tilsynelatende passer kriteriene for deretter å kun gå igjennom denne delmengden med mer kostbare operatorer.

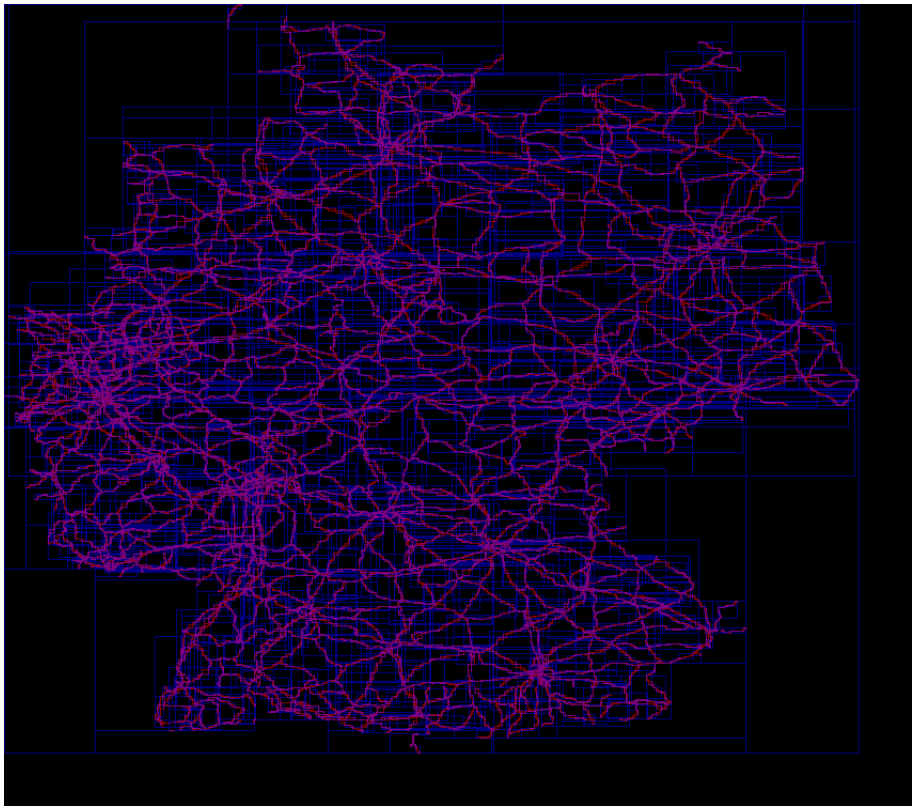
4.2 Generell struktur

R-treets struktur kan minne mye om et B-tre, men de skiller seg ut på noen punkter.

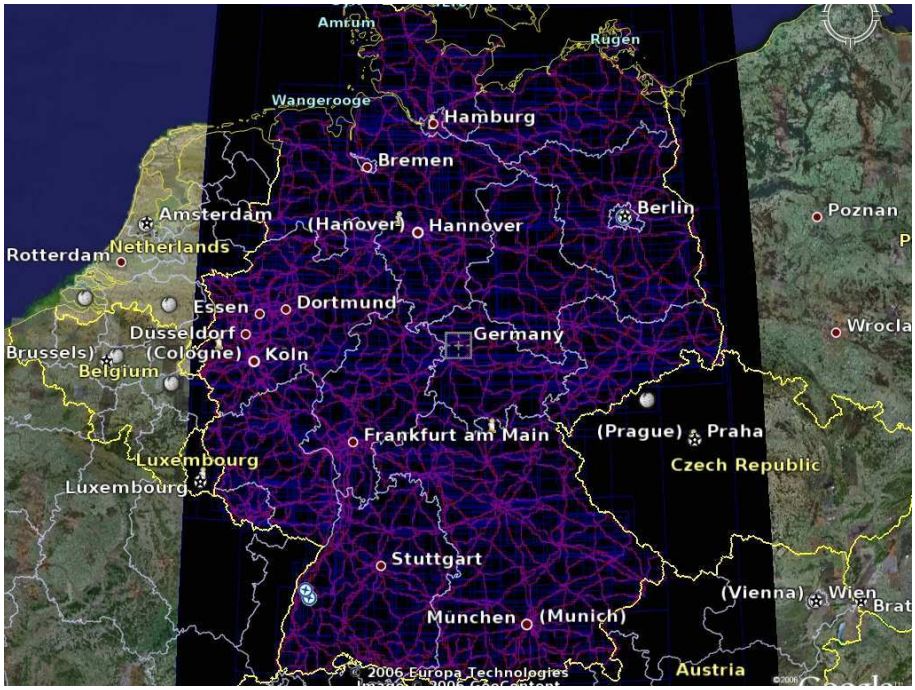
- Nøkkelkompleksitet
  I B-trær er nøkkelen som regel heltallig, noe som ville tilsvart et punkt i en 1-dimensjonal sammenheng. Nøkler i R-trær representerer en MBR for alle sine barn. Ved eksempelvis geodetiske data vil en typisk nøkkel



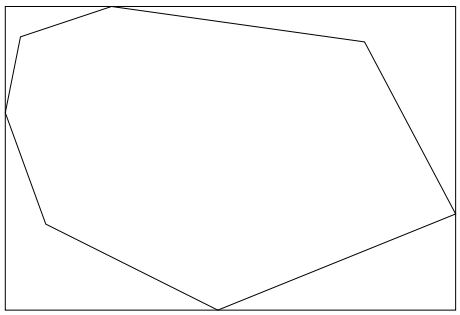
Figur 4.1: Eksempel på data MBR-er og de tilhørende MBR-ene, sammen med det tilsvarende treet.



Figur 4.2: R-tre lastet med data om det tyske veinettet



Figur 4.3: R-tre lastet med data om det tyske veinettet projisert på kartet gjennom google-earth



Figur 4.4: 2-dimensjonal MBR som omslutter en region

bestå av fire koordinater breddegrader start og slutt, lengdegrader start og slutt, mens f.eks for et kretskort vil man typisk benytte nøkler med kartesiske koordinater.

- Node-deling
  Deling av noder i R-trær må ses i en spatial sammenheng, derfor skiller prosedyren seg ganske mye ut fra tilsvarende algoritme i B-trær. Guttman foreslo originalt to algoritmer; lineær og kvadratisk kompleksitet.

- Nabo-pekere
  R-trær holder ingen pekere til nabo bladnoder slik som B-trær.

4.2.1 Noder

En node behøver tilgang til følgende informasjon:

- Maksimal størrelse
- Minimal størrelse
- Foreldre
- Barn, enten som peker til noder eller referanser til eksterne data
- Omfanget til denne noden[1]
- Om den er rot
- Om den er bladnode

For bladnoder behøves en samling tupler av typen $\{MBR, datareferanse\}$. For alle andre noder blir tuplene av typen $\{MBR, barnepeker\}$. Rent teknisk kan man løse dette på mange måter, noe jeg redegjør for i kapittel 5.

For enkelhets skyld vil jeg i resten av dette avsnittet diskutere 2-dimensjonale R-trær, mens eksemplene godt kan generaliseres til det n-dimensjonale. R-trær blir fort dyre for høyere dimensjoner, og det er utviklet mange spesielle varianter som håndterer dette, blant annet delta-treet[8].

[1] Det minste omsluttende n-rektangelet for alle barna

4.3 Oppdatering

Når det gjelder oppdateringer av R-trær, er det to typer design som skiller seg ut. De statiske trærne som ikke kan oppdateres etter at de er bygget, og de dynamiske som fortløpende kan oppdateres. Formålet med de statiske trærne er å utføre det som kalles "bulk-loading", hvor man har alle data på forhånd og kan lage et optimalt tre basert på disse. Mens de dynamiske trærne blir nødt til å inngå et kompromiss, som fører til at selve strukturen kan bli sub-optimal. Hva som benyttes kommer som regel an på behov, men de fleste dynamiske r-trær har også algoritmer for å bli bulk-loadet fra start av, slik at man kan begynne med et godt utgangspunkt. Videre i kapittelet vil kun det originale R-treet bli beskrevet.

Under er gangen i de forskjellige stegene satt opp punktvis, med oppføring E, som beskrevet i [10]. Her beskrives MBR-delen av en oppføring E som EI og $barne - pekeren$ eller $tuppel - identifikatoren$ som EP.

4.3.1 Innsetting

1. [Finn bladnode] Start "Velg blad-node" for å velge riktig node L å sette inn E.

2. [Legg til E] Hvis L har plass, sett inn E, hvis ikke start "Del Node" for å få L og LL som inneholder E og alle de gamle oppføringene.

3. [Propager forandringene] start "Tilpass Tre" på L og eventuelt LL hvis "Splitt Node" ble kjørt.

4. [Voks treet høyere] Hvis splitt-propagering førte til at rot-noden ble splittet, lag en ny rot-node hvis barn er de to nye nodene.

Velg blad-node

1. [Initialisering] Sett N til å være rot-noden

2. [Blad sjekk] Hvis N er blad-node returner N

3. [Velg sub-tre] Hvis N ikke er en blad-node, la F være oppføringen i N hvis areal FI øker minst. Hvis flere har tilsvarende økning, velg den med minst areal.

4. [Nedstigning til blad-node] Sett N til å være barn-noden pekt på av oppføring F og gjenta fra 2.

Tilpass tre

1. [Initialisering] Sett N = L, hvis L ble delt tidligere, sett NN til å være den andre noden.

2. [Sjekk om ferdig] Hvis N er roten, stopp.

3. [Tilpass omsluttende rektangel i foreldre-noden] La P være foreldre-noden til N, og la E_N være N sin oppføring i P. Tilpass $E_N I$ sånn at den omslutter alle oppføringer i N.

4. [Propager node-splitt oppover] Hvis N har en partner NN fra en tidligere deling, lag en ny oppføring E_{NN} med $E_{NN}P$ pekende til NN og $E_{NN}I$ som omslutter alle rektangler i NN. Legg E_{NN} til P hvis det er plass, ellers kall SplittNode for å produsere P og PP som inneholder E_{NN} og alle P sine gamle oppføringer.

5. [Beveg oppover] Sett $N = P$ og $NN = PP$ hvis en deling har skjedd. Gjenta fra 2.

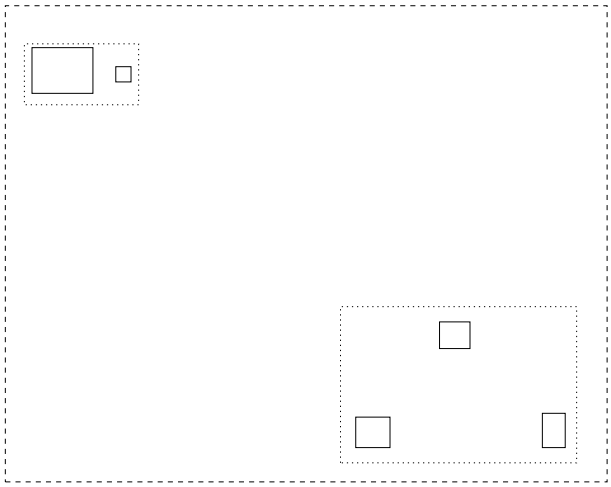
4.3.2 Del node

Deling av noder, også kalt bipartisjonering (for de tilfeller der noder deles i to), søker å dele en mengde rektangler i to grupper med gitte restriksjoner. Guttman foreslo originalt to forskjellige delingsalgoritmer til sitt R-tre. De to skiller seg ved at den ene er i O(n) kompleksitet mens den andre er i $O(n^2)$ kompleksitet, dvs henholdvis *lineær* og *kvadratisk*. Som ved andre datastrukturer må man veie opp kosten for å finne noe mot kosten ved å gjøre forandringer. Ofte fører dyr kost ved innsetting og oppdatering til lav kost ved søk. De to algoritmene som beskrives under har til hensikt å partisjonere settet $M + 1$ i to sett som minimaliserer overlapp og areal. Det som i Guttmans artikkel beskrives som "seeds" vil fra nå av bli omtalt som frø. Gutman skriver at han prøvde å benytte seg av en algoritme som prøvde ut alle mulig kombinasjoner, the exhaustive algorithm", men ved økende nodestørrelser fungerer de ikke.

Dynamisk programmering

En løsning som virker forlokkende til å løse bipartisjoneringsproblemet på er å benytte dynamisk programmering. Tidlig i prosjektet undersøkte jeg dette. Men det viste seg at dette ikke var en gangbar løsning.

Man kan tenke seg at man finner alle partisjoner med 2 noder i, for deretter å benytte seg av den optimale 2-node partisjonen til å bygge en



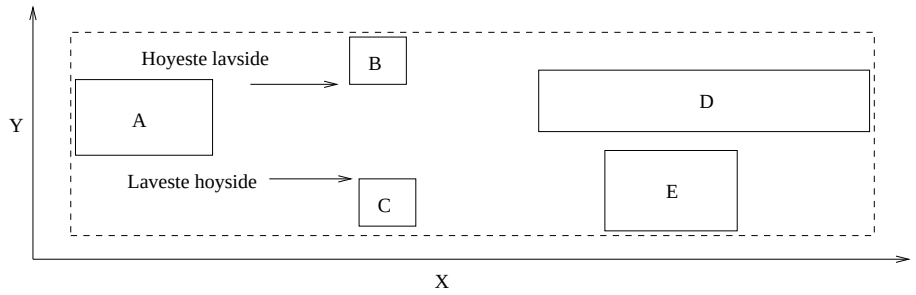
Figur 4.5: Situasjon der beste 2-node løsning ikke er en del av den beste 3-node løsningen

3-node partisjon. Men det er viktig å se her at en optimal del-løsning ikke behøver å være en del av den optimale løsningen, noe man kan se i figur 4.5, derfor kan vi ikke benytte oss av denne fremgangsmåten. Antall iterasjoner algoritmen ville ha utført kan man se i ligning 4.1. Ved $M = 50$ og $m = 2$ tilsvarer dette 626155256640137 iterasjoner, dette er ikke gjennomførbart.

$$\binom{N}{m} + \binom{N}{m+1} + ... + \binom{N}{\lfloor N/2 \rfloor} = \sum_{k=m}^{\lfloor N/2 \rfloor} \binom{N}{k} \tag{4.1}$$

Lineær deling

Første steget i den lineære splitten er å velge det som beskrives som frø. De to frøene skal være utganspunktet til de to nye nodene. Dette gjøres ved en lineær versjon av pickSeeds algoritmen. Her vil man gå igjennom alle oppføringene i noden som skal deles, og finne den laveste høysiden og den høyeste lavsiden for alle dimensjoner. Deretter normaliserer man for størrelsen på dimensjonen og velger det paret som er lengst fra hverandre. Se tegning 4.6 på neste side for en enklere visualisering av dette. I tegningen kan man se for Y-dimensjonen hvilke MBR som har høyeste lavside og laveste høyside, henholdsvis B og C. Den stiplete linjen rundt beskriver her størrelsen for dimensjonene X og Y. Vi kan se av tegningen at A og E vil representere det valgte paret for X-dimensjonen (står ikke eksplisitt i tegningen), og det kan være lett å tenke at avstanden mellom disse to er så stor at de vil bli



Figur 4.6: Valg av frø ved lineær splitt

valgt til frø. Men vi må huske at man normaliserer med hensyn på størrelsen til dimensjonene også, og X er her større enn Y. Etter valg av frø vil man fortløpende gå igjennom den overfyllte noden og distribuere oppføringene til de to nye nodene. Hvilken node en oppføring havner i er basert på hvor mye man må øke arealet til noden for å plassere oppføringen der. Man søker altså å minimere areal-kosten. For å ikke risikere at en node ender opp med alle oppføringene, må man hele tiden sjekke hvor mange oppføringer som er igjen i forhold til hvor mange som er i de respektive nodene. Om en situasjon skulle oppstå der en node trenger to oppføringer for å oppfylle kravet til m, og det er to oppføringer igjen å fordele, vil disse to havne i denne noden. Det er akkurat dette punktet *LazySplit* utnytter, noe vi får se senere.

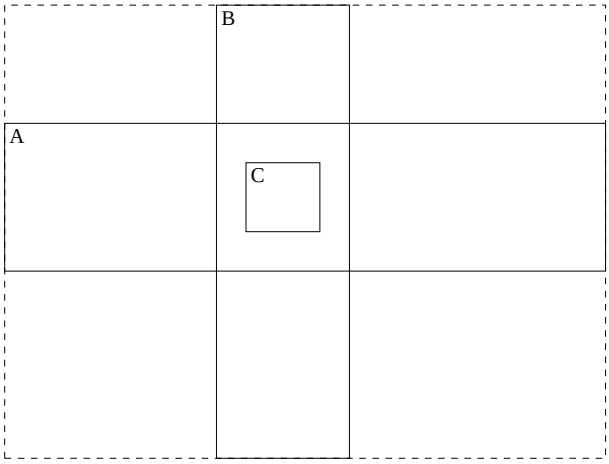
I artikkelen er det en del problemer som ikke nevnes. I noen spesielle situasjoner der en node innehar MBR-er som kun er inni hverandre, vil de valgte frøene være de samme. Det er viktig å ta grep her for å forhindre dette. Grunnen til at det ikke nevnes i artikkelen er nok mest sannsynlig fordi situasjonen sjelden vil oppstå, men det er like fullt noe man må kunne håndtere. Problemet kan utfolde seg på tre måter:

- Alle rektangler inni hverandre over alle dimensjoner
- Alle rektangler inni hverandre over X-dimensjonen
- Alle rekltangler inni hverandre over Y-dimensjonen

Figur 4.7 på neste side viser at oppføring C har den laveste høysiden og den høyeste lavsiden for begge dimensjonene.

Kvadratisk deling

I den kvadratiske delingen velger man frø annerledes. I motsetning til å gå igjennom oppføringene en gang, prøver man hver oppføring opp mot



Figur 4.7: Situasjon der oppføring C dominerer alle dimensjoner

hverandre, noe som gir $\binom{N}{2}$ distinkte muligheter, O(N²) kompleksitet. Det man søker å finne er det paret som ved å koples sammen danner en MBR med det største arealet, disse to vil deretter bli brukt som frø for hver sin node. Se figur 4.3.2 på neste side for detaljert gjennomgang. Som beskrevet i den lineære delingen ble de gjenværende oppføringene plukket ut og satt inn i noden de passet best til, så fremt ikke en node trengte resten av oppføringene. I kvadratisk deling skisserer Guttman en litt annen måte å velge neste oppføring for plassering. For hver gang man skal velge en ny oppføring må man gå igjennom alle de resterende. Dette gjøres ved at man for hver oppføring E som enda ikke er plassert ut, regner ut hvor mye arealet øker hvis du plasserer oppføringen i gruppe 1, kall denne d_1. Tilsvarende regner man ut d_2 for gruppe 2. Etter å ha gått gjennom alle gjenværende oppføringer velger man den E som har størst differanse mellom d_1 og d_2, deretter plasserer man oppføringen i gruppen som øker minst i areal. Ved tilfeller der man har to oppføringer med lik differanse, løser man dette ved først å velge den som vil øke sin gruppe minst, deretter til gruppen med minst oppføringer. Problemene nevnt om den lineære delingen vil ikke gjelde her, da man ikke tillater en oppføring å bli prøvd mot seg selv.

4.4 Søking

Et søk i treet må ses i lys av de spatiale operatorene. Vanlig fremgangsmåte er å utføre en range query"(“range” her henviser til at man søker innenfor et område med ekstensjon for hver dimensjon), også kalt ”window query“. Dette

```
1  private IndexRecord[] QuadraticPickSeeds() {
2      IndexRecord[] Seeds = new IndexRecord[2];
3      double LargestArea = Double.NEGATIVE_INFINITY;
4      for (int x = 0; x < this.Count-1; x++) {
5          for (int y = x+1; y < this.Count; y++) {
6              MBR tmpMBR = new MBR(this.getEntry(x).getBounds());
7              tmpMBR.expandBy(new MBR(this.getEntry(y).getBounds()
8              if(tmpMBR.getArea() > LargestArea) {
9                  LargestArea = tmpMBR.getArea();
10                 Seeds[0] = this.getEntry(x);
11                 Seeds[1] = this.getEntry(y);
12                 }
13              }
14         }
15     return Seeds;
16 }
```

Figur 4.8: Valg av frø, kvadratisk tid algoritme

viser til at man har et n-dimensjonalt søkerektangel, MBR, og ved hjelp av en spatial operator finner treffene innenfor denne. Den vanligste operasjonen er å benytte en binær operator med spatialt resultat.[2] Resultatmengden plukkes ut ved bruk av bolske predikater for å bestemme hvor man skal nedstige og hvilke oppføringer man skal plukke ut fra treet. Første del av søket er nedstigningen i treet hvor man benytter seg av ønsket spatialt predikat, eksempelvis "overlapp". For hver oppføring X i noden, og søkerektangel Y spør man "X overlapper Y", det bolske resultatet bestemmer om du fortsetter nedstigningen i denne oppføringen sine barn. Andre del er når man befinner seg i en bladnode, hver oppføring som overlapper legges til resultatmengden. Se pseudokode i figur 4.4 på neste side.

4.5 Generelle varianter

Siden 1984, da Guttman ga ut sin artikkel om R-treet, har det kommet mange varianter. Både spesialiserte og generelle. Det de har til felles er tilrettelegging for spatial aksess av data. For informasjon om andre strukturer som også er basert på R-treet, men som skiller seg mer fra dette, jf. avsnitt 2.3.2 på side 9. Slik B-treet har varianter kalt B+-tre og B*-tre, har R-treet også dette. Dette

[2]Man kan også se for seg situasjoner der man ønsker å vite om noe finnes; bolsk resultat.

```
1 Search(MBR searchRectangle, Stack resultStack)
2         for each index in this node:
3                 if index.MBR intersects with searchRectangle
4                         if(this.isLeaf)
5                                 resultStack.push(index)
6                         else
7                                 index.Search(searchRectangle)
```

Figur 4.9: Søk med binær spatial operator

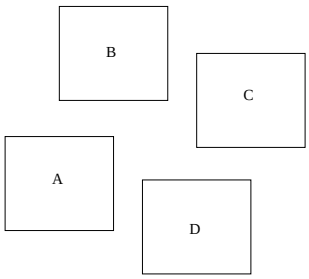
avsnittet vil ta for seg forskjellene mellom disse.

4.5.1 R+-treet

R+-treet ble originalt beskrevet av T. Sellis med fler[20]. De tar for seg hvorledes et R-tre over tid forfaller med hensyn på node-overlapp. Med node-overlapp menes overlappet mellom de omsluttende rektanglene til noder på samme nivå i treet. Dette fører også til at noder høyere i treet overlapper noder lenger ned i treet som ikke er etterkommere av denne. Årsaken til det økende overlappet er to-delt. Den fremste årsaken er den kvadratiske delingsalgoritmen. Siden dette kun er en heuristisk tilnærming til problemet vil den ikke kunne gjøre optimale delinger. Det andre er problemet med at man ikke alltid kan dele en partisjon i to uten å lage et overlapp. Figur 4.10 på neste side representerer en node med 4 barn. Med et minstekrav, ved deling, på to barn i de nye partisjonene ser vi at ingen permutasjoner vil kunne gjøre et rent kutt.

1. A og B vil overlappe D
2. A og C vil overlappe B og D
3. A og D vil overlappe C
4. B og C vil overlappe A
5. B og D vil overlappe A og C
6. C og D vil overlappe B

For å løse problemet ble det valgt å kreve at man ikke tillater overlapp. Dette håndheves ved at man klipper opp et rektangel slik at det tilpasses rektanglene det overlapper. Dette fører til at treet ikke forringes over tid



Figur 4.10: Overlappingsproblem ved partisjonering

på samme måte som i R-treet og antallet veier som må søkes blir mindre. Samtidig får man imidlertid samme objekt flere steder i treet. Dette vil øke minneforbruket og søketiden da man må eliminere duplikater. Dette er ikke ønskelig i f.eks. en RAM-basert indeks. En slik bestemmelse vil føre til forandringer i flere av algoritmene til treet. Sletting av noder må forsikre seg om at alle versjoner av et objekt fjernes fra treet. Innsetting må kunne sette inn et objekt flere steder og del node må kunne, i motsetning til vanlig r-tre, propagerer forandringer nedover. Hvis vi ser tilbake på figur 4.1 på side 25 vil datambr $C1$ befinne seg i både A og C, $C2$ i B og C, $B6$ i B og C

4.5.2 R*-treet

Som tidligere beskrevet er kriteriet for R-treet områdeminimering for MBR. I 1990 ble R*-treet[2] foreslått. En variant som går utover R-treets kriterier. Følgende kriterier ligger til grunn for R*-treet[13]:

- Minimering av området dekket av hver MBR
  Har til hensikt å minimere dødområdet, området som er dekket av MBR-er, men ikke de inneværende rektangler. Dette reduserer antall grener man må følge under spørringer.

- Minimering av overlapp mellom MBR-er
  Samme formål som forrige punkt, begrenser antall grener man må følge under spørringer.

- Minimering av MBR-marginer
  Prøver å forme mer kvadratiske rektangler for å øke ytelsen på spørringer med store kvadratiske former. Siden kvadratiske objekter lettere blir pakket, blir arealet på MBRer lenger opp i treet mindre, arealminimering indirekte oppnådd.

- Maksimering av lagringsutnyttelse
  Når utnyttelsen av treet er lav - underfyllte noder - har søket tendenser til å gjøre oppslag i flere noder. Dette holder spesielt for større spørringer. Når nodeutnyttelsen er lav øker trehøyden.

R*-treet prøver å finne den best mulige kombinasjonen av de overnevnte kriterer. Imidlertid kan enkelte av disse stå i motstrid til hverandre slik at oppfyllelse av ett kriterium hindrer oppfyllelse av et annet kriterium. Eksempelvis kan man tillate et mindre antall oppføringer i en node for å sikre minimalt overlapp av noder, men dette kan føre til dårligere utnyttelse av nodene som strider mot det siste kriteriet.

Kapittel 5

R-tre implementasjon

I forrige kapittel ble R-treets struktur og dets operasjoner gjennomgått. Vi fikk så på noen av problemene rundt nodedelinger og hvordan blant annet R*-treet løser disse med en "ingeniør-tilnærming". Dette kapittelet beskriver først min implementasjon av R-treet for deretter å presentere en ny måte å dele noder på kallt *LazySplit*. I neste kapittel skal vi se hvordan den nye delingsalgoritmen presterer sammenlignet med vanlig nodedeling.

I utgangspunktet benyttet FAST Geotools (`http://geotools.codehaus.org/`) sin implementasjon. Dette er en java-implementasjon basert på den originale artikkelen fra 1984. Denne er lisensiert under LGPL (GNU Lesser General Public License) og man ønsket å gjøre en del forandringer i koden, det ble derfor bestemt at man skulle lage en egen implementasjon. I min grunnimplementasjonen la jeg ikke mye vekt på forandringer i forhold til artikkelen, men gjorde et lettere forsøk på å minimalisere minnebruk uten at dette førte til for uleselig kode. Man finner av den grunn ikke noen bruk av bit-operasjoner for å få komprimert informasjonen. Det er laget et par varianter[15] som virker motivert av CSB+-treet (Cache Sensitive B+-tre[19]), der man optimaliserer node-størrelse i forhold til størrelsen på cache-linjene i CPU slik at en node får plass. Slik komprimering blir som regel pålagt de n-dimensjonale nøklene (MBRene) kombinert med at man ikke lagrer rektangelsider som er like foreldrenes sider. Årsaken til at man ikke kan kjøre hard komprimering på nøklene er blant annet at sluttresultatet er en funksjon av tiden man bruker på dekomprimering, samt tiden man vinner på å unngå cache-misses. En annen negativ side av et cache-optimaliserte r-tre blir nødvendigvis høyden. Selv i et 2-dimensjonalt tilfelle vil hver nøkkel ha 4 distinkte koordinater, noe som tar plass. Nodestørrelsen blir derfor begrenset og høyden øker.

```
1 npro.lazy.NodeLazyLinear
2 npro.lazy.NodeLazyQuadratic
3 npro.normal.NodeLazy
4 npro.normal.NodeQuadratic
5 npro.utils.MBR
6 npro.utils.Node
7 npro.utils.Rtree
8 npro.utils.IndexEntry
```

Figur 5.1: Kodehierarki

5.1 Implementasjonsdetaljer

Min implementasjon, som er skrevet i java og baserer seg på 1.5 spesifikasjonene, ligger på rundt 2 000 linjer kode. Koden er bygget rundt en pakkestruktur som er inndelt slik man ser i figur 5.1. Bortsett fra selve R-treet er noe kode skrevet i Python. Dette er hovedsaklig hjelpeverktøy for generering av syntetiske datasett.

npro.utils.Node
For å unngå mye redundans ble en abstrakt Nodeklasse opprettet. Denne inneholder og håndterer alle generiske variable og metoder. Samtidig definerer den de abstrakte metodene slik at disse blir implementert i klassene som arver fra denne. Hver node inneholder en array av typen IndexRecord. Det er denne arrayen som beskriver alle barna til noden eller dataene avhengig av om noden er en bladnode eller ikke. Hver node har også en MBR som beskriver det omsluttende rektangelet for alle oppføringene til noden.

npro.utils.IndexRecord
Objekter av typen IndexRecord er veldig enkle. De inneholder en MBR som forteller om omfanget til barnet denne IndexRecorden peker på, eller selve dataene. Samtidig er det en generisk peker av typen *Object* som må kastes til riktig type ved bruk. Denne pekeren peker på det aktuelle barnet eller eksempelvis en databasereferanse. I Diligent vil denne kun peke på en ID.

npro.utils.MBR
MBR-klassen er en abstrakt representasjon av et n-dimensjonalt rektangel som beskrevet i avsnitt 4.1 på side 24. I denne spesifikke implementasjonen fungerer den utelukkende som en representasjon av et to-dimensjonalt rektangel. MBR-klassen håndterer også de spatiale operatorene som skal

```
1 boolean intersects(MBR other) {
2         return !(x2 < other.x1 || x1 > other.x2 ||
3                  y1 > other.y2 || y2 < other.y1);
4 }
```

Figur 5.2: *Overlapper* operatoren

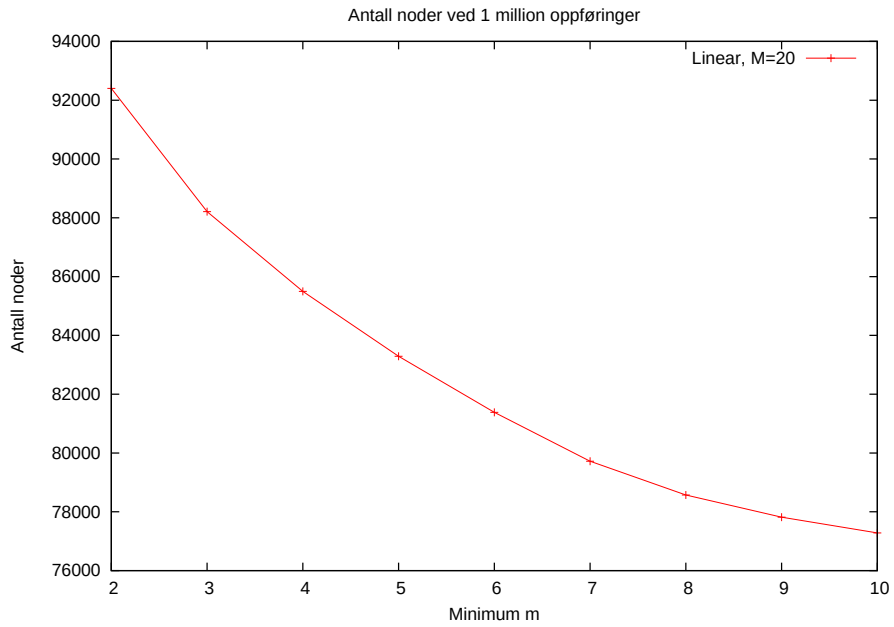
benyttes. I denne sammenhengen er det binære predikatet *overlapper* implementert. Det er viktig å merke seg at slike operatorer er avhengige av å være raske, da de blir utført nedover på alle steg i treet. I figur 5.1 ser vi pseudo-koden for operatoren. Uttrykket evalueres til falskt med en gang noen av de individuelle testene er sanne. Testen er lagt opp på denne måten fordi vi antar at et søk som regel vil falsifisere flere rektangler enn det aksepterer. Sett i et bredere perspektiv enn selve oppgaven vil det være mer korrekt å implementere operatorene for seg selv, slik at man åpner muligheten for utviklere til selv å implementere ønskede operatorer. Dette gjelder spesielt i raffinerings-steget da økt kompleksitet er vanlig, og det et stort antall forskjellige søk som er ønskelige.

5.1.1 Visualisering

For lettere å kunne identifisere algoritmiske problemer ved nodedelinger samt kunne visualisere et helt tre, har jeg implementert en visualiseringspakke i R-treet. I store trekk visualiserer den enten hele treet, eller en stakk med MBR-er. Visualisering av bare en stakk lar deg eksempelvis visualisere kun resultatet av et søk, eller innholdet i en spesifikk node på forskjellige tidspunkter. Dette er veldig gunstig for å overvåke delingene. Man kan selv spesifisere detaljnivået på bildene som blir presentert, men koden påtvinger et 4:3 forhold der du selv spesifiserer bredden. I kapittel 4 på side 23 kan man se flere bilder som er generert ved hjelp av denne pakken.

5.1.2 Redusert minnebruk

Som tidligere beskrevet propagerer nodedelinger seg oppover i treet. Dette medfører at man behøver en måte å spore veien tilbake opp treet igjen. Geotools har løst dette ved foreldrepekere i alle noder, noe som øker størrelsen per node med 4 Bytes. Dette er avhengig av arkitektur: For et 64 bit system kan en peker eksempelvis være 8 Byte. Ved én million oppføringer, et optimalt tre med alle noder fulle og en M på 10 bruker man da opp om lag 400 kB plass. I realiteten har man mange fler noder og plassbruken



Figur 5.3: Antall noder i treet ved forskjellige minimumskrav m for M=20, lineær deling

kan mangedobles. I figur 5.3 kan man se hvordan et høyere minimumskrav fører til et mindre antall noder. Sett i lys av at man ønsker, både i en situasjon der deler av treet befinner seg på disk og der man har en ren RAM-basert indeks, å benytte så lite minne som mulig, ser vi at en høyere m er ønskelig. En høyere m kan lett provosere frem et tre med mange dårlige delinger og medfølgende høyt overlapp. I min løsning har jeg valgt å fjerne foreldrepekere fullstendig. Nodedelinger og oppover-propageringen er hendelser som etterfølger en oppdatering, og veien man har gått ned i treet kan enkelt spores tilbake ved å benytte en stakk som historisk spor. Gangen i en oppdatering vil da være å legge inneværende node på stakken på veien ned i treet. Når man senere skal propagere forandringene oppover, det vil si kjøre adjustTree metoden, popper man bare av node etter node for hver iterasjon.

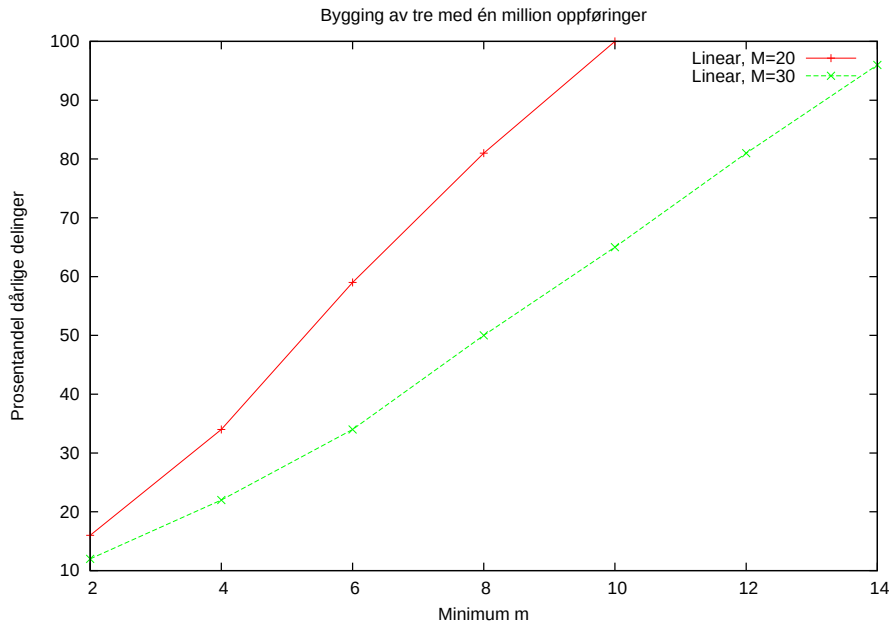
5.2 En ny plan for deling av noder

Det er flere som har tatt for seg problemet med bipartisjonering av noder. I 1994 kom hybridstrukturen Hilbert R-treet [11]. Selve treet ligger utenfor oppgaven, men måten de håndterer nodedelinger på er interessant. Her deler

man ikke en node før alle søsken (noder på samme nivå) er fulle. Istedenfor kan overflytende noder dele ut oppføringer til sine søsken. I 2002 foreslo Brakatsoulas med fler cR-treet[6]. Her forandres helt forutsetningen om at en node nødvendigvis må partisjoneres i to. Videre presenteres en metode for å finne den optimale k-node fordelingen. Det var blant annet disse artiklene som fikk meg til å revurdere måten noder deles på. Mest fremtredende er X-treet, en struktur originalt tiltenkt å løse problematikken rundt data av høy dimensjonalitet. Her presenteres et konsept om super-noder, der man kan vente med å dele en node gitt at den fører til en dårlig deling. Måten dette løses på er å gjøre om en node til en super-node. Dette gjøres ved å effektivt doble plassen i noden, for deretter å fortsette innsettinger i denne noden til den er full igjen. Som vi skal i avsnittet om *LazySplit* benytter jeg det samme prinsippet om hetrogene noder. Men fremfor å tillate en node å bli så stor, noe som gjør søk mindre effektive hvis et stort antall av nodene i treet er i en "utvidet" situasjon, vil jeg kun tillate treet å vokse litt før jeg setter den foten. Grunnen til at jeg kan tillate meg å operere mer dynamisk, er fordi min implementasjon kun skal benyttes med geospatiale data og det er lettere å få utført en deling relativt fort ved utvidelse av plassen i noden.

5.2.1 Problemer rundt nodedeling

Et av de store problemene ved nodedelinger er at man har motstridende ønsker. På den ene side ønsker man et balansert tre, noe man garanterer ved å kreve et minimum m av oppføringer i en node. Balansert i denne sammenhengen må ikke forveksles med skjevheten på treet. Man kan få en skjevhet av oppføringer, men ikke noder da treet er selvbalanserende av natur. Her tolkes skjevhet som fyllingsgraden på treet. Det er ønskelig med et minimum av underfulle noder for å unngå stor høyde og skjevhet i oppføringer. Samtidig fører mange underfulle noder til en økning av det totale antallet noder i treet, noe som leder til mindre overlapp, men mer minnebruk. På den annen side ønsker man en best mulig deling basert på heuristikken man benytter. Dette gir spesielt stort utslag for den lineære delingen, som Guttman foreslo, da man ikke analyserer hele settet til en node, men fortløpende fordeler oppføringene. Når man har underskudd på oppføringer i en node og krever at resterende oppføringer skal inn i denne, er sannsynligheten for at delingen er dårlig. For å finne ut av dette, tok jeg målinger av hvor ofte man havner i situasjoner der man tvinger oppføringer inn i en node for å oppfylle kravet til m. Resultatet viste hvordan antall antatt dårlige delinger skyter i været når man setter et sterkere krav til m. I figur 5.4 på neste side ser man dette tydelig. Det er min antagelse at slike dårlige delinger, spesielt høyt oppe i treet, vil føre til unødig stort overlapp.



Figur 5.4: Antall dårlige delinger ved forskjellige minimumskrav m for $M = 20$ og $M = 30$

Dette fører igjen til at man må traversere mange flere grener enn nødvendig.

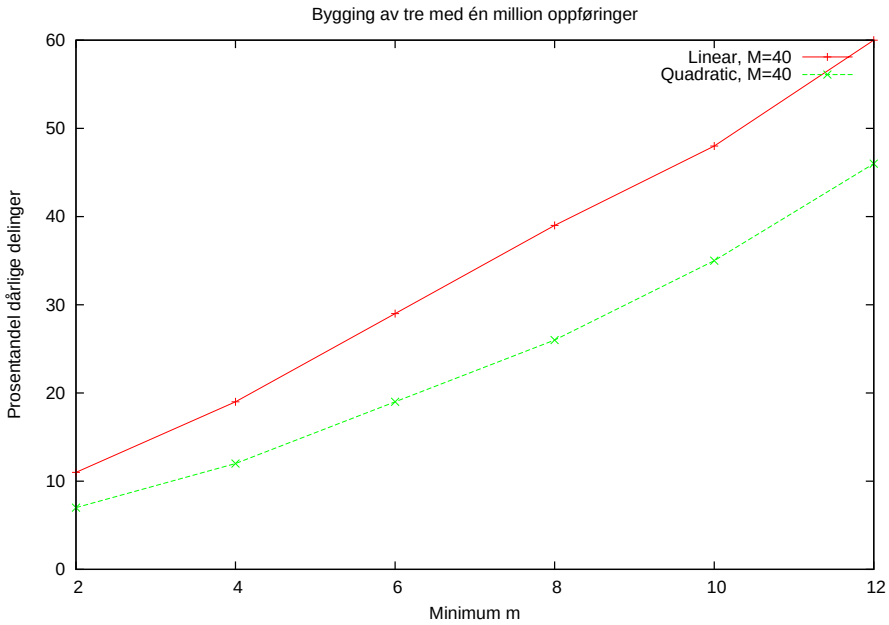
Til sammenligning er det interessant å se hvor mange delinger som er dårlige for den kvadratiske algoritmen. Dette for å se om dårlige delinger korrelerer med resultatet vi ser under søk. I figur 5.5 på neste side kan man se prosentandelen dårlige delinger for den kvadratiske algoritmen plottet sammen med den lineære. Man kan her se at den kvadratiske konsekvent er noe bedre enn den lineære, men alikevel ikke nok til å forklare den store forbedringen man får på søk. Det er mest sannsynlig en kombinasjon av færre dårlige delinger samt en bedre distribusjon av dataene som fører til dette. Tabell 5.1 på neste side viser tallmaterialet.

5.2.2 Lazy Split

Poenget med en hierarkisk struktur er å unngå sekvensiell lesing. De unødige nodene ignoreres og kun den relevante delmengden må gjennomgås. X-treet velger å doble sin nodestørrelse. Vi kan anta at dette gjøres for å optimalisere i forhold til sekundærlager, slik at en node eksempelvis kan oppta to diskpager når den blir en supernode. Som forfatterne poengterer fører dette til dobbelt så mye sekvensiell lesing for en gitt node, men alternativet (dårlige nodedelinger) er verre. Situasjonen for sekvensiell lesing blir verre desto fler

m	Linear $M = 20$	Linear $M = 30$	Quadratic $M = 40$
2	16	12	7
4	34	22	12
6	59	34	19
8	81	50	26
10	100	65	35
12		81	46
14		100	

Tabell 5.1: Antall dårlige delinger ved forskjellige minimumskrav m for $M = 20$, $M = 30$ og $M = 40$



Figur 5.5: Antall dårlige delinger ved forskjellige minimumskrav m for M=40 i den kvadratiske algoritmen

noder i treet som er i en slik tilstand. Muligheten for å både kunne nyte godt av utvidede noder og samtidig minimere sekvensiell lesing, er viktig å se nærmere på. Det er nettopp dette Lazy Split gjør. Vi antar at man ikke nødvendigvis behøver å tilpasse en node for mange ekstra oppføringer, men at det holder med noen. Jeg har to argumenter for at dette er fornuftig i forhold til sekundær-lager: For det første ser fremtiden lys ut for RAM-baserte indekser. 64-bits arkitektur er kanskje bare i startgropen for øyeblikket, men det er kun et spørsmål om tid før dette blir standard, noe som åpner for et mye større adresse-rom (som igjen fører til muligheten for mer RAM). For det andre kan man se for seg at en node ikke nødvendigvis må utnytte en hel disk-page. Om man reserverer noe plass til overflytende oppføringer er det mulig at utbyttet man får i mindre overlapp, vil overskygge tapet man får i mindre oppføringer (høyere tre på disk). Vi skal nå se på hvordan *LazySplit* er implementert i praksis. Det er viktig å huske at på dette stadiet er strukturen kun et "proof of concept" og eksempelvis sekundærlager er ikke tatt med i utviklingen.

Tanken bak *LazySplit*-algoritmen er at man tillater temporære skjevheter i treet for å kunne nyte godt av potensielt bedre delinger. Ved å tillate disse temporære skjevhetene utnytter man det faktum at en deling gjøres basert på lokale data, man tillater hver node å prøve ut fremtidige data hvis nåtids data er for dårlige. I implementasjonen av slike noder er det en ekstra IndexRecord-peker som opptar 4 bytes. Hvis og når noden overflyter, oppretter man en ekstra array i forhåndsbestemt størrelse. Den konseptuelle gangen beskrives under. Størrelsen på denne ekstra arrayen beskrives heretter som λ, mens M og m er som før.

Innsetting

1. [Finn bladnode] Start "Velg blad-node" for å velge riktig node L å sette inn E.

2. [Legg til E] Hvis L har plass, sett inn E, hvis ikke start "Del Node" for å få L og LL som inneholder E og alle de gamle oppføringene.

3. [Propager forandringene] start "Tilpass Tre" på L og eventuelt LL hvis "Splitt Node" ble kjørt.

4. [Voks treet høyere] Hvis splitt-propagering førte til at rot-noden ble splittet, lag en ny rot-node hvis barn er de to nye nodene.

Del Node

1. Hvis en nodedeling ikke fører til en dårlig partisjonering, eller $N + 1 > M + \lambda$, gå videre, ellers hopp til punkt 3.

```
1 public IndexRecord getEntry(int x) {
2         return OF(x) ?
3         this.OverFlowEntries[OFI(x)] : this.Entries[x];
4         }
```

Figur 5.6: Metoden for å håndtere all aksessering av arrayer

2. kall på gamle "del node" og returner dennes L og LL.

3. Legg til den overflytende oppføringen i den ekstra plassen og returner L og LL hvor LL er en null-peker.

Som vi ser over, legges det inn et ekstra steg i del-node-metoden. Her prøver vi først å finne ut om delingen er dårlig. Kriteriene som benyttes for å identifisere en dårlig deling, er om oppføringer tvinges inn i en node for å oppfylle m, uten mulighet til å finne ut om de faktisk hører til denne. Et "worst-case scenario" er hvis $N > M + \lambda$ og delingen fortsatt er dårlig, da har vi potensielt kastet bort $\lambda * N$ tid.

I og med at veldig mange av metodene i treet jobber med arrayene er det viktig å skjule dette arbeidet slik at man ikke får unødvendig komplisert og redundant kode. Dette er viktig både for det vanlige tilfellet der man kun forholder seg til én array, men spesielt når man har en hetrogen behandling av én eller to arrayer. I figur 5.2.2 ser vi den enkle metoden som skjuler dette fra resten. Her er $OF(x)$ en funksjon som regner ut om x er større enn M, og $OFI(x)$ en funksjon som gir tilbake $xmoduloM$.

Kapittel 6

Ytelsestester

I dette kapittelet blir formålet, metodene og resultatene av ytelsestestene på implementasjonen gjennomgått. Det blir vist hvordan *LazySplit* anvendt sammen med den lineære delingsalgoritmen både gir et bedre tre med mindre overlapp samt liten økning i byggetid. Det blir også vist hvordan metoden kan hjelpe et tre bygget med den kvadratiske delingsalgoritmen til å bli enda bedre.

Testen som er gjennomført er todelt. På den ene siden ønsker vi å se hvilken økning i søketid man kan forvente, implisert av antall noder man må besøke i treet. Samtidig må man ta tidskosten for oppbygging av strukturen i betraktning. Det vil ikke bli testet for sletting av data i treet da dette ikke blir påvirket av *LazySplit*.

Mange spatiale indekser som har blitt utviklet, måler sine resultater mot R*-treet. Siden R*-treet er mottatt som den generisk beste varianten (det finnes bedre varianter, men ingen har fått samme tyngde som R*-treet, og er heller ikke nødvendigvis like generiske) faller dette naturlig. I mine tester får jeg ikke gjort dette da heuristikkene til R*-treet ikke har blitt implementert enda.

6.1 Koordinatgenerering

For enkelt å få tilgang på syntetiske datasett, benyttes et script til å generere disse. Scriptet genererer en binær-fil med et spesifisert antall rektangler innen et spesifisert område. Størrelse og form kan spesifiseres. Disse rektanglene er ikke forbeholdt brukt utelukkende som data, men kan også benyttes som søkerektangler. Grunnet prosjektets behov for en struktur som utelukkende håndterer 2-dimensjonale data, er scriptet også kun tilrettelagt for dette. Scriptet er laget i Python og er relativt enkelt.

Det er fire variable som styrer genereringen:

- Størrelse på dimensjonen x
- Størrelse på objektene innenfor dimensjon x
- Størrelse på dimensjonen y
- Størrelse på objektene innenfor dimensjon y

6.2 Ytelsestest av oppdatering

6.2.1 Metode

For å finne ut hvor dyrt vi må betale (i form av tid) for bruk av den nye delingsalgoritmen, er det viktig å få prøvd ut tre-konstruksjon mot vanlige delinger. Det viktige her er ikke å finne ut hva en oppdatering på et fullt tre koster, men å sammenligne den komplette byggingen av et tre med *LazySplit* algoritmen mot vanlig deling. Samtidig ser vi på tiden til den vanlige kvadratiske delingen mot *LazySplit* lineær. Antageligvis kunne vi, for å finne et bedre mål enn tid, ha telt gjennomsnittlige iterasjoner per deling og multiplisert dette med antall delinger. Totalt antall delingsiterasjoner ville ha gitt en indikator på tidsbruken. Dette blir ikke gjort. Det er derfor viktig å få med hva slags maskinvare testene har blitt utført på, slik at tidsbruken får et referansepunkt. Serveren, som ikke har hatt andre oppgaver gående (cron-jobber o.l), har hatt 1 GB RAM og AMD Athlon 64 3500+ CPU. Operativsystemet er linux med 2.6 kjerne.

6.2.2 Formål

Formålet for tidtakingene er for å finne ut hvor stor ekstra kostnad som påløper ved å bruke *LazySplit*. Det er ønskelig at *LazySplit* ikke legger på for mye overhead. (Hvis overheadet blir alt for stort vil man ikke tjene stort.) Hvis tiden begynner å konvergere mot algoritmer av høyere kompleksitet i stor O notasjon, har en påført for stor kostnad. Selv om man bare får en ekstra koeffisient å forholde seg til, noe som ikke forandrer kompleksiteten teoretisk i stor-O notasjon vil det gi praktiske utslag. Formålet blir å identifisere hvor stort utslag.

6.2.3 Datasett

Det er vanskelig å få tak i store datasett i form av MBR-er uten å betale for dette. Geodetiske data kommer ofte i form av TIGER lines eller Shape-formatet, og det finnes noen gratis sett, men selv om man skulle transformert disse til MBR-er vil man ofte sitte igjen med for få objekter i forhold til det som er ønskelig for testene. Nettstedet GeoCommunity (`http://www.geocomm.com/`) tilbyr sine medlemmer datasett spredt rundt i verden. Dette er ofte lett tilgjengelig informasjon som politiske linjer, kontinenter og lignende. R-tre portalen (`www.rtreeportal.org`) har også enkelte datasett. Blant annet datasettet brukt til visualiseringen av Tyskland kom fra denne siden.

For mine tester valgte jeg å generere syntetiske sett. Datasettet jeg genererte var i en størrelsesorden av 1 000 000 2-dimensjonale rektangler. Koordinatene er uniformt fordelt i et diskret plan og trekkes ved en pesudo-random algoritme. Som tidligere beskrevet består en slik MBR av fire koordinater, start og slutt for hver dimensjon. Mer formelt ble koordinatene trukket fra settet COORD.
COORD $= \{x \in \mathbb{Z} \mid 0 \leq x \leq 1100000\}$.
For å gjennomføre selve genereringen utarbeidet jeg et python-script som skriver ut disse MBR'ene binært til en fil. Dette scriptet er omtalt i avsnitt 6.1 på side 49. Det er viktige motforestillinger mot å benytte syntetiske sett. De sammenfaller i liten grad med den virkeligheten som indeksene ofte benyttes til, og dette kan skape falske inntrykk av effektivitet. Problemene i tilknytning til å få tak i store datasett har vært så betydelige, at det forsvarer bruken av syntetiske sett. Strukturen det testes mot benytter også de samme syntetiske dataene. Det er med andre ord ingen forskjellsbehandling.

6.2.4 Resultater

Som implisert i *LazySplit*-algoritmen vil delinger kunne føre til alt mellom N og $\lambda * N$ iterasjoner, altså fortsatt O(N) kompleksitet (N her referer til $M \leq N \leq M + \lambda$). Men det er å anta at tiden en tre-konstruksjon tar, blir noe større grunnet denne faktoren. Man kan se i figur 6.1 på side 53 og tabell 6.1 på neste side at den lineære algoritmen ligger et sted mellom 24 og 25 sekunder avhengig av m. Den kvadratiske ligger mellom 43 og 45 sekunder. Byggingen utført med Lazy Split faller mellom 28 og 32 sekunder. Det er to ting å merke seg her. For det første øker ikke tiden mye fra den vanlige lineære delingen mye når man benytter *LazySplit*. For det andre når man ser at man får en økning på rundt 28 pst. i forhold til den normale lineære, må vi kunne konkludere med at selve delingsprosessen sett relativt

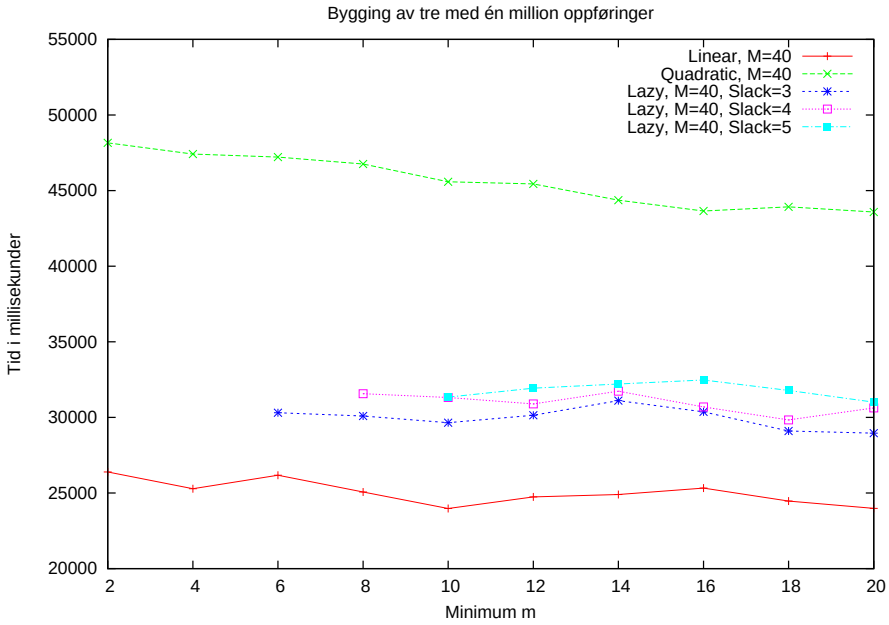
m	Linear	Quadratic	Lazy:Slack=3	Lazy: Slack=4	Lazy: Slack=5
10	23974	45587	29649	31320	31340
12	24739	45443	30145	30895	31931
14	24901	44363	31112	31731	32211
16	25327	43647	30364	30695	32475
18	24463	43925	29096	29834	31784
20	23980	43580	28961	30624	31009

Tabell 6.1: Tidsbruk ved tre-konstruksjon for M=40

til hele tre-konstruksjonen spiller en mindre rolle enn det som virker intutivt. Vi vet at delingen vil øke i tid med en faktor på mellom 1 og λ, og vi har tidligere i figur 5.4 på side 44 sett at man ofte vil utnytte den ekstra arrayen grunnet den høye prosentandelen dårlige delinger. Man kan også konkludere det samme basert på at den kvadratiske delingen kun dobler tidsbruken. Ligning 6.1 viser tiden brukt av delingene. Vi ser her, ut i fra tallene som er hentet inn, at selve delingen står for ca 17 pst. av tidsbruken.

Det viktigste er at tidsbruken ikke øker så mye som det er intuitivt å anta. Ofte vil det være nok å tillate én ekstra oppføring for at en node kan deles fornuftig. Dette understøtter *LazySplit* i den forstand at selv om vi ville ha spart mye tid på å kunne sjekke for mulig deling én gang ekstra, slik som i X-treet, ser vi at den ekstra kosten ikke blir så veldig stor.

$$
\begin{aligned}
&\text{Gitt antagelsen} \\
&iotid + delingstid = totaltid \\
&\text{Ved innsetting} \\
&iotid + x = 23.97 \\
&iotid + x^2 = 45.58 \\
&\text{Gir oss} \\
&-x^2 + x + 21.61 = 0 \\
&x = 4.18
\end{aligned}
\tag{6.1}
$$



Figur 6.1: Tid brukt for innsetting av én million oppføringer

6.3 Ytelsestest av spørringer

6.3.1 Metode

For å finne ut om ideen bak *LazySplit* fungerer slik det er tenkt, må man få prøvd den ut. Det faller naturlig å prøve den ut på den lineære delingsalgoritmen, samtidig som man ser hva slags kvalitet den kvadratiske algoritmen får. Det er interessant å se om *LazySplit* kan styrke den lineære algoritmen i så stor grad at det resulterende treet kan sidestilles med et tre basert på den kvadratiske algoritmen.

Jeg begynte jeg med tidtakninger ved hjelp fra java-APIet. Dette for å finne tiden for bygging av treet, samt å ta tiden på 10 000 søk. Siden spesifikasjonen til Sun ikke gir deg en mulighet til å få reel cpu-tid, kommer tidene til å variere avhengig av hva som kjører på maskinen. Det finnes teknikker der man utnytter JNI (Java Native Interface) med egenlaget C / C++ kode for å kunne gjøre bedre tidtakninger, men så ikke dette som interessant da det ville ha medført for mye ekstra arbeid i forhold til hva man får ut av det. Med utgangspunkt i disse problemene valgte jeg å kjøre de 10 000 spørringene 50 ganger, notere tiden for hver av de 50 kjøringene, beregne et gjennomsnitt og et standardavvik. Hvis testen ikke ga et tilfredstillende standardavvik, ble

```
for(max = 10; max <= 80; max += 10) {
    for(min = 2; min <= floor(max/2); min += 2) {
        Tree = buildNormalTree(max,min);
        runTests(Tree)
        for(slack = 2; slack <= floor(min/2); slack++) {
            Tree = buildLazyTree(max,min,slack);
            runTests(Tree);
        }
    }
}
```

Figur 6.2: Testkjøringer på vanlig og lazy trær

den simpelthen kjørt på nytt. Dette viste seg veldig kostbart med henhold til tiden jeg hadde til rådighet for testing. Det var et stort antall tester å kjøre. Til slutt falt valget på å registrere antall noder som blir berørt gjennom søkene. På denne måten får man en direkte indikasjon hvor gode delingene er, desto flere noder man er innom, desto flere noder overlapper sannsynligvis på flere nivåer i treet. Samtidig slipper man å bekymre seg for ekstreme uteliggere i testene grunnet cronjobber og lignende inteferens.

For det normale treet med lineær delingsalgoritme gjelder følgende verdier:

- $10 \leq M \leq 80$ med en inkrementering på 10
- $2 \leq m \leq \lfloor M/2 \rfloor$ med en inkrementering på 2

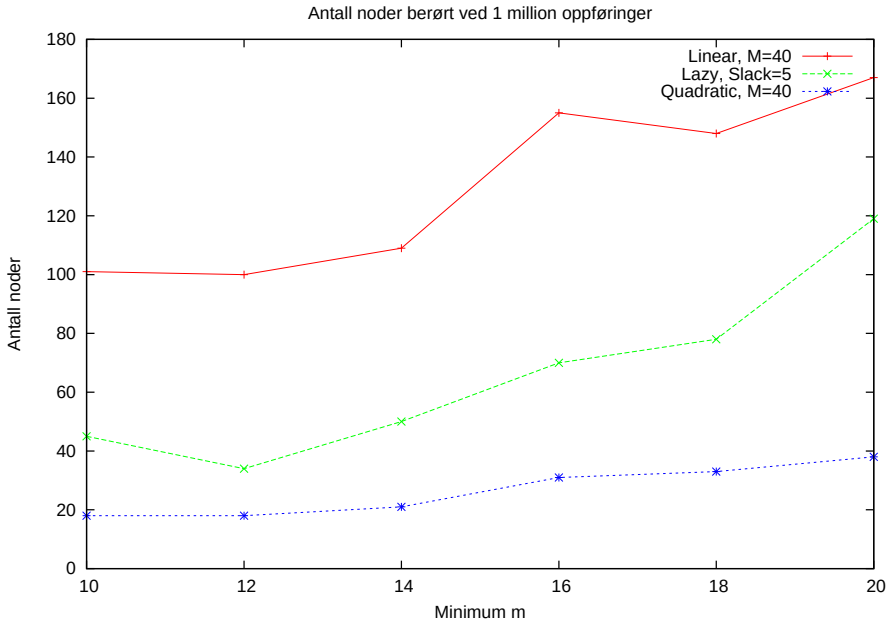
For hver av disse iterasjonene ble også *LazySplit*-treet testet med:

- $2 \leq \lambda \leq \lfloor m/2 \rfloor$ med inkrementering på 1

Se pseduo-koden i figur 6.3.1 for en programmatisk forklaring.

Testdata

Til å utføre søk på treet genererte jeg 10 000 uniformt fordelte søkerektangler. Disse varierte i størrelse fra 0.001 pst. til 1 pst. av det totale dataplanet. Rektanglene har også forskjellig form. Det er viktig å få prøvd lange og tynne rektangler, så vel som fullstendig kvadratiske rektangler, da datasettet også er høyst hetrogent i den sammenheng. Årsaken til at jeg ønsket såpass mange søk var for å få et representativt gjennomsnitt statistisk.



Figur 6.3: Noder berørt ved én million oppføringer

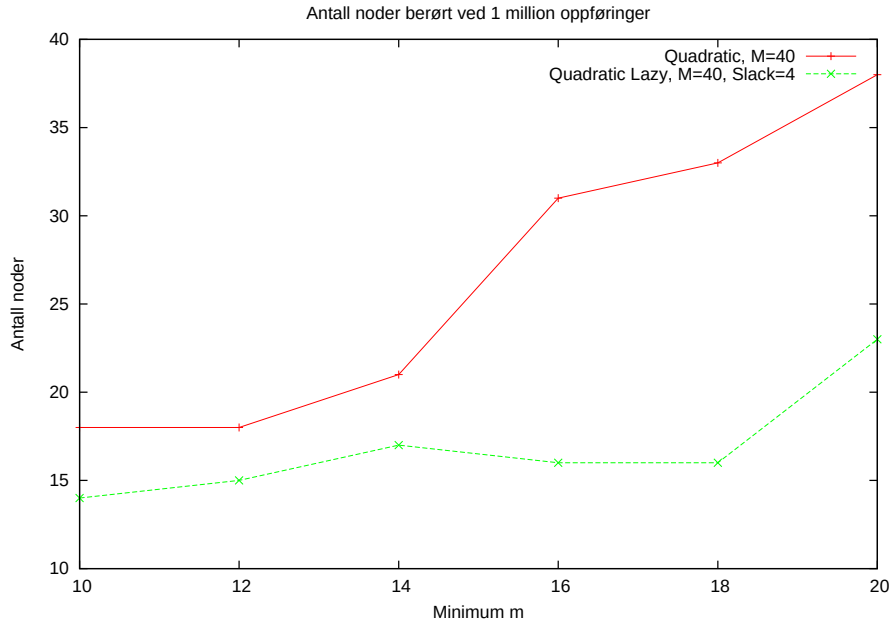
6.3.2 Formål

Formålet er å finne ut om treets struktur blir bedret av å tillate temporære skjevheter. Det er ønskelig å få et tre som kan måle seg mot den kvadratiske algoritmen i kvalitet, og den lineære algoritmen i indekseringstid. I forrige avsnitt så vi hvordan *LazySplit* førte til et mindre overhead enn det man intuitivt vil anta.

6.3.3 Resultater

I figur 6.3 kan man se søk i et tre med $M = 40$. De tre grafene representerer den kvadratiske, den lineære og den lineære med bruk av *LazySplit*. Vi ser her hvordan *LazySplit* for forskjellige verdier av m omtrentlig halverer antall noder man berører i søkene. Dette er en direkte indikator på at overlappet har blitt redusert drastisk.

Det er tydelig at den kvadratiske algoritmen bygger et tre av høyere kvalitet, derfor er det spennende å se hva slags forbedring man kan få ved å bruke *LazySplit* på denne. Vi kunne anta at den lineære delingen som er mye dårligere, fikk godt utnytte av *LazySplit*, det samme kan ikke nødvendigvis sies om den kvadratiske. Som vi så i figur 5.5 på side 45, unngikk den kvadratiske algoritmen en del av de dårlige delingene. Dette faller naturlig



Figur 6.4: Noder berørt ved én million oppføringer

siden den prøver ut alle oppføringene mot hverandre. Som grafene i figur 6.4 viser, muliggjør *LazySplit* for en enda bedre struktur i kombinasjon med den kvadratiske algoritmen. Men dette koster igjen i form av enda lengere oppdateringstid.

Kapittel 7

Konklusjon

I starten av oppgaven ble problemstillingen rundt min implementasjon av R-treet beskrevet. Det ble påpekt at det var ønske om å kunne indeksere data fortere og oppnå bedre spørretid. Oppgaven har tatt for seg problemene rundt overlappende noder. Min implementasjonen forandret noen av forutsetningene for hvordan noder deles basert på X-treets håndtering av et høyt antall dimensjoner. I kapittel 5 ble *LazySplit*, som er et dynamiskt perspektiv på denne teknikken beskrevet, samt noe av problematikken som denne teknikken prøvde å redusere. Samtidig ble data som ikke hadde et behov for eksplisitivitet i nodene fjernet for å lage et mindre minne-fotavtrykk. Kapittel 6 viser at resultatene til *LazySplit* er lovende. Tidskostnaden ved bygging viste seg å være mindre enn det som er intuitivt. Treets struktur ble også markant bedret for søk ved at overlappet mellom nodene sank. *LazySplit* er dermed et viktig skritt mot mindre overlapp.

Fremover er det interessant å se på nytten man får av en slik dynamisk utsettelse av delinger som *LazySplit* representerer. Spesielt tester mot X-treet for å fastslå den faktiske differansen mellom de mulige store sekvensielle lesingene dette har, mot *LazySplit* sitt dynamiske perspektiv.

Selve implementasjonen som skal benyttes i produksjon, er ikke helt ferdig. Siden den ble utviklet mer som et "proof of concept", er ting som håndtering av samtidig aksess, bulk-loading og persistens ikke implementert. Dette er videre arbeid som må komme på plass. Innenfor feltet spatial indeksering er det et viktig utviklingspotensial å se på muligheten for fremtidige rene RAM-baserte indekser. Dette gjelder også for store datamengder. Flerdimensjonale data tar mye plass å beskrive, spesielt ved et høyt antall dimensjoner. Et marked for 64-bits arkitekturer vil imidlertid legge til rette for mye mer RAM enn det vi ser i dag.

Bibliografi

[1] *Spatial databases with application to GIS*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[2] Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B. The r*-tree: an efficient and robust access method for points and rectangles. In *SIGMOD '90: Proceedings of the 1990 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1990), ACM Press, pp. 322–331.

[3] Bentley, J. L. Multidimensional binary search trees used for associative searching. *Commun. ACM 18*, 9 (1975), 509–517.

[4] Berchtold, S., Keim, D. A., and Kriegel, H.-P. The x-tree: An index structure for high-dimensional data. In *Proceedings of the 22nd International Conference on Very Large Databases* (San Francisco, U.S.A., 1996), T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, Eds., Morgan Kaufmann Publishers, pp. 28–39.

[5] Böhm, C., Berchtold, S., and Keim, D. A. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv. 33*, 3 (2001), 322–373.

[6] Brakatsoulas, S., Pfoser, D., and Theodoridis, Y. Revisiting r-tree construction principles, 2002.

[7] Comer, D. Ubiquitous b-tree. *ACM Comput. Surv. 11*, 2 (1979), 121–137.

[8] Cui, B., Ooi, B. C., Su, J., and Tan, K.-L. Indexing high-dimensional data for efficient in-memory similarity search. *IEEE Transactions on Knowledge and Data Engineering 17*, 3 (2005), 339–353.

[9] Gaede, V., and Günther, O. Multidimensional access methods. *ACM Comput. Surv. 30*, 2 (1998), 170–231.

[10] Guttman, A. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1984), ACM Press, pp. 47–57.

[11] Kamel, I., and Faloutsos, C. Hilbert r-tree: An improved r-tree using fractals. In *Proceedings of the Twentieth International Conference on Very Large Databases* (Santiago, Chile, 1994), pp. 500–509.

[12] Kothuri, R. K. V., Ravada, S., and Abugov, D. Quadtree and r-tree indexes in oracle spatial: a comparison using gis data. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2002), ACM Press, pp. 546–557.

[13] Manolopoulos, Y., Nanopoulos, A., Papadopoulos, A. N., and Theodoridis, Y. *R-Trees: Theory and Applications (Advanced Information and Knowledge Processing).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[14] Medeiros, C. B., and Pires, F. Databases for gis. *SIGMOD Rec. 23*, 1 (1994), 107–115.

[15] Min, Y. S., Yang, C. Y., Yoo, J. S., Shim, J. M., and Song, S. I. Pcr-tree: An enhanced cache conscious multi-dimensional index structures. In *DEXA* (2004), pp. 212–221.

[16] M.J. Egenhofer, R. F. Point-set topological relations. *Int. Journal for Geographical Information Systems* (1991).

[17] MySQL. Creating spatial indexes. http://dev.mysql.com/doc/refman/5.1/en/creating-spatial-indexes.html, March 2007.

[18] Popescu, A. R. A study of r-tree based spatial access methods. Master's thesis, UNIVERSITY OF HELSINKI, 2003.

[19] Rao, J., and Ross, K. A. Making b+- trees cache conscious in main memory. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2000), ACM Press, pp. 475–486.

[20] Sellis, T. K., Roussopoulos, N., and Faloutsos, C. The r -tree: A dynamic index for multi-dimensional objects. In *The VLDB Journal* (1987), pp. 507–518.

Testdata

I dette tillegget er tabeller med data fra alle kjøringene med *LazySplit* på den lineære algoritmen. I de tilfeller der en λ ikke er oppgitt må dette tolkes som at *LazySplit* ikke er i bruk. Disse tallene kan da sees komparativt til de eterfølgende.

M	m	λ	Noder berørt	Prosent dårlige delinger	Byggetid
10	2		180	37.0	17805
10	4		306	87.0	16969
10	4	2	280	59.0	20247

Tabell 1: Data ved forskjellige oppsett

M	m	λ	Noder berørt	Prosent dårlige delinger	Byggetid
20	2		66	16.0	20914
20	4		151	34.0	19809
20	4	2	86	22.0	22870
20	6		153	59.0	20547
20	6	2	186	40.0	22624
20	6	3	96	32.0	24413
20	8		245	81.0	19962
20	8	2	132	64.0	23049
20	8	3	148	55.0	24557
20	8	4	144	45.0	24602
20	10		337	100.0	19422
20	10	2	262	94.0	22639
20	10	3	218	87.0	23749
20	10	4	199	78.0	24837
20	10	5	138	71.0	27245

M	m	λ	Noder berørt	Prosent dårlige delinger	Byggetid
30	2		40	12.0	23485
30	4		97	22.0	21052
30	4	2	65	14.0	23303
30	6		133	34.0	21032
30	6	2	66	24.0	24609
30	6	3	44	18.0	26731
30	8		95	50.0	23051
30	8	2	70	37.0	25081
30	8	3	48	30.0	26903
30	8	4	43	24.0	27838
30	10		128	65.0	22176
30	10	2	120	51.0	24790
30	10	3	110	42.0	25391
30	10	4	61	35.0	27438
30	10	5	57	30.0	28695
30	12		179	81.0	22463
30	12	2	119	67.0	25561
30	12	3	95	60.0	27048
30	12	4	82	51.0	27994
30	12	5	76	45.0	28849
30	12	6	67	39.0	30576
30	14		246	96.0	21636
30	14	2	190	85.0	24151
30	14	3	141	78.0	25976
30	14	4	153	71.0	27186
30	14	5	98	64.0	29509
30	14	6	98	58.0	30328
30	14	7	115	52.0	30836

M	m	λ	Noder berørt	Prosent dårlige delinger	Byggetid
40	2		34	11.0	26393
40	4		47	19.0	25287
40	4	2	29	13.0	29731
40	6		54	29.0	26175
40	6	2	34	20.0	29309
40	6	3	30	14.0	30312
40	8		62	39.0	25064
40	8	2	40	28.0	29510
40	8	3	50	20.0	30089
40	8	4	31	16.0	31564
40	10		101	48.0	23974
40	10	2	59	36.0	28727
40	10	3	65	28.0	29649
40	10	4	36	22.0	31320
40	10	5	45	19.0	31340
40	12		100	60.0	24739
40	12	2	64	46.0	29103
40	12	3	60	37.0	30145
40	12	4	61	30.0	30895
40	12	5	34	26.0	31931
40	12	6	36	23.0	32920
40	14		109	70.0	24901
40	14	2	90	57.0	29224
40	14	3	57	49.0	31112
40	14	4	51	43.0	31731
40	14	5	50	37.0	32211
40	14	6	40	32.0	33643

M	m	λ	Noder berørt	Prosent dårlige delinger	Byggetid
40	14	7	40	28.0	33468
40	16		155	80.0	25327
40	16	2	132	67.0	28383
40	16	3	90	60.0	30364
40	16	4	99	52.0	30695
40	16	5	70	46.0	32475
40	16	6	56	41.0	33243
40	16	7	64	37.0	32091
40	16	8	59	32.0	32173
40	18		148	92.0	24463
40	18	2	124	82.0	27212
40	18	3	127	75.0	29096
40	18	4	115	69.0	29834
40	18	5	78	64.0	31784
40	18	6	78	58.0	32320
40	18	7	84	52.0	33063
40	18	8	91	47.0	33300
40	18	9	65	43.0	34306
40	20		167	100.0	23980
40	20	2	235	97.0	26993
40	20	3	135	93.0	28961
40	20	4	143	89.0	30624
40	20	5	119	83.0	31009
40	20	6	121	79.0	32211
40	20	7	106	72.0	33441
40	20	8	78	68.0	34459
40	20	9	108	63.0	34878
40	20	10	66	58.0	36430

M	m	λ	Noder berørt	Prosent dårlige delinger	Byggetid
50	2		28	9.0	29471
50	4		33	16.0	27601
50	4	2	26	10.0	29895
50	6		56	21.0	25335
50	6	2	35	15.0	29113
50	6	3	34	11.0	29820
50	8		71	30.0	24494
50	8	2	33	20.0	29635
50	8	3	42	14.0	28806
50	8	4	26	11.0	29932
50	10		66	39.0	25476
50	10	2	42	26.0	29122
50	10	3	41	21.0	29307
50	10	4	32	16.0	30636
50	10	5	27	13.0	30564
50	12		78	45.0	24398
50	12	2	43	35.0	28738
50	12	3	38	27.0	29888
50	12	4	37	22.0	30291
50	12	5	30	18.0	30948
50	12	6	28	14.0	30754
50	14		82	55.0	25765
50	14	2	55	43.0	29180
50	14	3	46	35.0	30199
50	14	4	44	28.0	29735
50	14	5	35	23.0	31036
50	14	6	33	20.0	31279
50	14	7	31	17.0	32109
50	16		79	63.0	25915
50	16	2	77	49.0	28109

M	m	λ	Noder berørt	Prosent dårlige delinger	Byggetid
50	16	3	68	43.0	29317
50	16	4	71	36.0	29966
50	16	5	48	32.0	31241
50	16	6	39	28.0	32504
50	16	7	58	22.0	30334
50	16	8	30	21.0	33207
50	18		76	71.0	26647
50	18	2	87	59.0	28638
50	18	3	61	52.0	30116
50	18	4	62	44.0	29857
50	18	5	64	39.0	31260
50	18	6	41	36.0	32515
50	18	7	48	31.0	32637
50	18	8	37	28.0	33688
50	18	9	37	24.0	32683
50	20		114	79.0	25897
50	20	2	112	69.0	27728
50	20	3	77	62.0	29502
50	20	4	74	55.0	31941
50	20	5	56	52.0	31666
50	20	6	59	45.0	32183
50	20	7	50	40.0	33254
50	20	8	39	37.0	34832
50	20	9	43	34.0	35439
50	20	10	40	30.0	34949
50	22		108	89.0	25950
50	22	2	127	79.0	27864
50	22	3	112	73.0	30231
50	22	4	83	67.0	31210
50	22	5	102	62.0	30757

M	m	λ	Noder berørt	Prosent dårlige delinger	Byggetid
50	22	6	69	58.0	33294
50	22	7	79	51.0	32715
50	22	8	68	47.0	33826
50	22	9	62	44.0	34972
50	22	10	68	40.0	35096
50	22	11	51	36.0	36190
50	24		156	97.0	25484
50	24	2	148	91.0	27703
50	24	3	114	87.0	30032
50	24	4	104	82.0	30579
50	24	5	105	78.0	31881
50	24	6	76	74.0	33653
50	24	7	72	68.0	34300
50	24	8	75	63.0	35474
50	24	9	79	59.0	35182
50	24	10	84	54.0	35324
50	24	11	67	51.0	37055
50	24	12	48	47.0	39147
60	2		27	7.0	29568
60	4		34	13.0	27068
60	4	2	31	9.0	29620
60	6		50	19.0	26956
60	6	2	22	14.0	31512
60	6	3	23	10.0	31624
60	8		91	22.0	23037
60	8	2	27	19.0	30844
60	8	3	23	14.0	31491
60	8	4	21	11.0	32501
60	10		67	30.0	26628
60	10	2	40	22.0	28613
60	10	3	28	19.0	31634
60	10	4	31	14.0	31886
60	10	5	25	11.0	31921

M	m	λ	Noder berørt	Prosent dårlige delinger	Byggetid
60	12		67	38.0	26758
60	12	2	31	31.0	30574
60	12	3	38	22.0	30861
60	12	4	42	17.0	30136
60	12	5	41	14.0	31962
60	12	6	42	12.0	31659
60	14		94	46.0	25295
60	14	2	42	35.0	29474
60	14	3	32	28.0	31869
60	14	4	41	21.0	31121
60	14	5	26	19.0	33672
60	14	6	26	16.0	33293
60	14	7	21	14.0	33873
60	16		85	50.0	26389
60	16	2	59	42.0	30423
60	16	3	49	34.0	31268
60	16	4	40	27.0	30800
60	16	5	30	23.0	32023
60	16	6	30	21.0	33742
60	16	7	27	17.0	33012
60	16	8	30	15.0	34170
60	18		96	59.0	26551
60	18	2	58	50.0	30211
60	18	3	38	41.0	32284
60	18	4	32	36.0	33564
60	18	5	35	30.0	33515
60	18	6	33	26.0	32793
60	18	7	27	23.0	34731
60	18	8	28	20.0	35616
60	18	9	29	18.0	35031
60	20		102	66.0	26850
60	20	2	80	55.0	29270
60	20	3	49	48.0	31382
60	20	4	42	42.0	33644

M	m	λ	Noder berørt	Prosent dårlige delinger	Byggetid
60	20	5	51	36.0	32836
60	20	6	34	32.0	35180
60	20	7	37	29.0	34736
60	20	8	34	25.0	34695
60	20	9	32	21.0	35061
60	20	10	26	21.0	36669
60	22		111	72.0	26878
60	22	2	70	62.0	30532
60	22	3	61	56.0	32517
60	22	4	56	50.0	32547
60	22	5	48	44.0	34227
60	22	6	41	39.0	35235
60	22	7	38	35.0	35247
60	22	8	42	32.0	35130
60	22	9	30	29.0	36946
60	22	10	35	27.0	36764
60	22	11	31	23.0	37199
60	24		104	79.0	27589
60	24	2	82	70.0	30555
60	24	3	83	65.0	31605
60	24	4	54	57.0	33019
60	24	5	64	51.0	33244
60	24	6	61	47.0	33637
60	24	7	49	42.0	34575
60	24	8	63	39.0	35857
60	24	9	48	36.0	35600
60	24	10	44	32.0	37165
60	24	11	44	30.0	36909
60	24	12	37	26.0	38015
60	26		102	86.0	27637
60	26	2	102	79.0	29601
60	26	3	91	73.0	32124
60	26	4	83	66.0	31458

M	m	λ	Noder berørt	Prosent dårlige delinger	Byggetid
60	26	5	75	63.0	33571
60	26	6	69	56.0	33733
60	26	7	70	52.0	34705
60	26	8	71	49.0	36802
60	26	9	59	44.0	35952
60	26	10	52	40.0	37061
60	26	11	45	38.0	39351
60	26	12	39	34.0	39313
60	26	13	39	32.0	40192
60	28		129	94.0	27089
60	28	2	115	88.0	29809
60	28	3	87	83.0	31251
60	28	4	75	79.0	33445
60	28	5	87	74.0	33298
60	28	6	84	69.0	33684
60	28	7	80	65.0	35410
60	28	8	73	60.0	36149
60	28	9	78	56.0	37883
60	28	10	56	53.0	38125
60	28	11	51	50.0	38826
60	28	12	57	44.0	39315
60	28	13	52	43.0	41009
60	28	14	64	38.0	40548
60	30		180	100.0	26769
60	30	2	112	98.0	29356
60	30	3	108	95.0	30872
60	30	4	102	92.0	32833
60	30	5	130	88.0	32809
60	30	6	97	85.0	35472
60	30	7	90	81.0	36059
60	30	8	72	76.0	37613

M	m	λ	Noder berørt	Prosent dårlige delinger	Byggetid
60	30	9	84	73.0	38869
60	30	10	70	69.0	39975
60	30	11	71	64.0	41179
60	30	12	57	61.0	41805
60	30	13	65	58.0	43254
60	30	14	68	52.0	41790
60	30	15	47	49.0	45072
70	2		20	8.0	31886
70	4		38	11.0	28287
70	4	2	23	8.0	32731
70	6		39	16.0	27972
70	6	2	22	12.0	33330
70	6	3	20	8.0	33461
70	8		57	21.0	25433
70	8	2	30	15.0	31727
70	8	3	27	11.0	32712
70	8	4	20	9.0	34407
70	10		60	26.0	27357
70	10	2	27	19.0	32321
70	10	3	29	14.0	32331
70	10	4	21	11.0	34383
70	10	5	26	9.0	33843
70	12		48	33.0	29455
70	12	2	29	24.0	31728
70	12	3	28	18.0	33597
70	12	4	21	14.0	34499
70	12	5	19	11.0	35620
70	12	6	18	10.0	35668
70	14		47	38.0	29106
70	14	2	33	28.0	32508
70	14	3	28	21.0	33877
70	14	4	21	18.0	35486
70	14	5	31	14.0	33179

M	m	λ	Noder berørt	Prosent dårlige delinger	Byggetid
70	14	6	18	12.0	35307
70	14	7	28	10.0	34108
70	16		65	44.0	27889
70	16	2	52	35.0	31396
70	16	3	35	27.0	33619
70	16	4	34	21.0	33218
70	16	5	28	17.0	34532
70	16	6	31	15.0	33094
70	16	7	23	13.0	35561
70	16	8	22	11.0	36718
70	18		76	49.0	26870
70	18	2	61	39.0	31872
70	18	3	47	31.0	33139
70	18	4	31	26.0	35203
70	18	5	39	22.0	33843
70	18	6	23	19.0	35168
70	18	7	31	16.0	34560
70	18	8	32	13.0	34960
70	18	9	25	13.0	37095
70	20		74	54.0	28323
70	20	2	62	42.0	31372
70	20	3	53	37.0	33070
70	20	4	36	31.0	34453
70	20	5	43	25.0	33977
70	20	6	42	23.0	34028
70	20	7	24	20.0	36616
70	20	8	21	18.0	37645
70	20	9	22	15.0	36593
70	20	10	21	14.0	37730
70	22		55	62.0	30416
70	22	2	49	51.0	33415
70	22	3	44	42.0	33360

M	m	λ	Noder berørt	Prosent dårlige delinger	Byggetid
70	22	4	47	36.0	33698
70	22	5	36	32.0	36674
70	22	6	33	29.0	37171
70	22	7	39	24.0	35008
70	22	8	30	22.0	37279
70	22	9	29	20.0	36796
70	22	10	22	18.0	38467
70	22	11	24	15.0	37503
70	24		72	67.0	28613
70	24	2	59	57.0	33129
70	24	3	57	48.0	34381
70	24	4	42	43.0	35245
70	24	5	47	38.0	34943
70	24	6	42	34.0	35694
70	24	7	32	30.0	36476
70	24	8	32	28.0	37413
70	24	9	34	25.0	37431
70	24	10	30	22.0	39030
70	24	11	28	20.0	39531
70	24	12	24	18.0	38865
70	26		104	72.0	28465
70	26	2	74	62.0	32038
70	26	3	57	57.0	35052
70	26	4	56	51.0	34345
70	26	5	43	45.0	36380
70	26	6	45	41.0	36782
70	26	7	43	38.0	37695
70	26	8	44	32.0	37708
70	26	9	28	30.0	39531
70	26	10	40	28.0	39041
70	26	11	33	25.0	39662
70	26	12	28	24.0	40468

M	m	λ	Noder berørt	Prosent dårlige delinger	Byggetid
70	26	13	32	21.0	40439
70	28		123	80.0	28144
70	28	2	78	71.0	32907
70	28	3	72	62.0	33469
70	28	4	60	57.0	35308
70	28	5	58	53.0	36573
70	28	6	54	48.0	36233
70	28	7	52	43.0	36310
70	28	8	40	40.0	38247
70	28	9	56	37.0	37454
70	28	10	55	35.0	40206
70	28	11	44	31.0	38353
70	28	12	40	28.0	40795
70	28	13	33	27.0	41592
70	28	14	39	23.0	41123
70	30		97	86.0	29922
70	30	2	94	76.0	31030
70	30	3	91	72.0	33474
70	30	4	55	66.0	35837
70	30	5	72	62.0	35985
70	30	6	53	55.0	37349
70	30	7	75	53.0	36383
70	30	8	50	48.0	38210
70	30	9	45	45.0	39758
70	30	10	50	41.0	40430
70	30	11	40	39.0	41814
70	30	12	57	37.0	40385
70	30	13	42	34.0	42738
70	30	14	44	31.0	41660
70	30	15	38	29.0	44743

M	m	λ	Noder berørt	Prosent dårlige delinger	Byggetid
70	32		129	91.0	28505
70	32	2	103	85.0	31386
70	32	3	109	80.0	32910
70	32	4	72	75.0	34952
70	32	5	72	72.0	35649
70	32	6	65	67.0	37905
70	32	7	63	64.0	38504
70	32	8	69	58.0	38025
70	32	9	64	56.0	41164
70	32	10	51	51.0	41499
70	32	11	53	49.0	41816
70	32	12	51	44.0	42269
70	32	13	49	43.0	43145
70	32	14	48	39.0	43797
70	32	15	34	38.0	46427
70	32	16	33	35.0	47291
70	34		120	98.0	28348
70	34	2	109	93.0	31499
70	34	3	103	91.0	34036
70	34	4	107	87.0	33722
70	34	5	71	84.0	37221
70	34	6	81	79.0	37532
70	34	7	75	76.0	38578
70	34	8	78	71.0	39757
70	34	9	81	67.0	41104
70	34	10	80	65.0	42128
70	34	11	64	59.0	42128
70	34	12	58	56.0	43692
70	34	13	51	54.0	44782
70	34	14	46	50.0	46838
70	34	15	46	48.0	48354
70	34	16	54	45.0	45429

M	m	λ	Noder berørt	Prosent dårlige delinger	Byggetid
70	34	17	39	42.0	47771
80	2		19	7.0	32517
80	4		23	13.0	33067
80	4	2	21	7.0	34715
80	6		29	17.0	31331
80	6	2	22	11.0	34672
80	6	3	18	7.0	34850
80	8		32	22.0	32468
80	8	2	24	14.0	35255
80	8	3	29	9.0	34057
80	8	4	30	7.0	33758
80	10		31	26.0	32862
80	10	2	28	17.0	35728
80	10	3	21	12.0	35356
80	10	4	24	9.0	34620
80	10	5	17	7.0	37014
80	12		27	31.0	32363
80	12	2	28	22.0	35010
80	12	3	22	16.0	37123
80	12	4	27	12.0	36168
80	12	5	18	10.0	37102
80	12	6	19	8.0	37449
80	14		50	31.0	29653
80	14	2	53	23.0	31895
80	14	3	31	19.0	34176
80	14	4	26	15.0	36736
80	14	5	21	13.0	37548
80	14	6	21	10.0	38574

M	m	λ	Noder berørt	Prosent dårlige delinger	Byggetid
80	14	7	19	8.0	38621
80	16		81	37.0	27245
80	16	2	40	28.0	33754
80	16	3	30	23.0	35773
80	16	4	25	17.0	37596
80	16	5	23	15.0	38161
80	16	6	26	12.0	35998
80	16	7	19	11.0	38200
80	16	8	23	9.0	36814
80	18		58	43.0	30744
80	18	2	46	33.0	34111
80	18	3	31	26.0	35203
80	18	4	26	21.0	36974
80	18	5	27	18.0	36428
80	18	6	35	15.0	36157
80	18	7	21	14.0	39084
80	18	8	21	11.0	37865
80	18	9	17	10.0	38721
80	20		84	47.0	28489
80	20	2	44	36.0	34898
80	20	3	36	31.0	36397
80	20	4	29	25.0	36828
80	20	5	23	23.0	38419
80	20	6	24	19.0	38698
80	20	7	29	16.0	37619
80	20	8	22	14.0	39875
80	20	9	23	13.0	38699
80	20	10	20	11.0	40209

M	m	λ	Noder berørt	Prosent dårlige delinger	Byggetid
80	22		54	53.0	31937
80	22	2	36	42.0	36777
80	22	3	32	36.0	37921
80	22	4	29	30.0	38142
80	22	5	28	26.0	38008
80	22	6	27	23.0	39189
80	22	7	23	21.0	40426
80	22	8	22	17.0	40441
80	22	9	19	15.0	40609
80	22	10	28	13.0	38026
80	22	11	22	12.0	40314
80	24		55	59.0	31873
80	24	2	57	48.0	35798
80	24	3	47	42.0	35518
80	24	4	40	34.0	35998
80	24	5	30	30.0	38871
80	24	6	33	27.0	38626
80	24	7	26	23.0	39914
80	24	8	27	20.0	39552
80	24	9	31	17.0	40477
80	24	10	23	17.0	40643
80	24	11	22	15.0	42304
80	24	12	22	14.0	41733
80	26		57	64.0	32028
80	26	2	56	54.0	35174
80	26	3	50	46.0	36389
80	26	4	43	40.0	37590
80	26	5	31	35.0	40038
80	26	6	35	31.0	38979

M	m	λ	Noder berørt	Prosent dårlige delinger	Byggetid
80	26	7	30	28.0	40338
80	26	8	25	25.0	40290
80	26	9	23	23.0	41240
80	26	10	25	20.0	41271
80	26	11	24	18.0	42721
80	26	12	26	17.0	40843
80	26	13	24	15.0	42057
80	28		82	69.0	30604
80	28	2	67	58.0	35846
80	28	3	62	51.0	34913
80	28	4	45	46.0	38042
80	28	5	45	41.0	37748
80	28	6	31	38.0	39952
80	28	7	31	34.0	41249
80	28	8	34	31.0	40859
80	28	9	30	27.0	40625
80	28	10	39	25.0	40335
80	28	11	24	24.0	43042
80	28	12	27	22.0	43300
80	28	13	32	19.0	41488
80	28	14	32	19.0	42187
80	30		84	76.0	32950
80	30	2	72	63.0	34297
80	30	3	62	58.0	37020
80	30	4	49	51.0	37558
80	30	5	43	47.0	39230
80	30	6	39	43.0	40036
80	30	7	55	38.0	38246
80	30	8	40	36.0	41736
80	30	9	38	33.0	40268

M	m	λ	Noder berørt	Prosent dårlige delinger	Byggetid
80	30	10	48	31.0	40349
80	30	11	30	29.0	42977
80	30	12	35	26.0	42578
80	30	13	40	24.0	43070
80	30	14	28	22.0	42982
80	30	15	22	21.0	45189
80	32		128	79.0	30513
80	32	2	77	70.0	35261
80	32	3	60	64.0	38365
80	32	4	52	59.0	37634
80	32	5	63	53.0	37283
80	32	6	46	50.0	40004
80	32	7	39	45.0	41117
80	32	8	40	42.0	41388
80	32	9	35	40.0	42053
80	32	10	43	35.0	40581
80	32	11	36	34.0	44428
80	32	12	40	30.0	44814
80	32	13	33	29.0	45176
80	32	14	36	26.0	44547
80	32	15	32	24.0	44284
80	32	16	25	24.0	47210
80	34		85	84.0	32095
80	34	2	69	76.0	36788
80	34	3	63	70.0	36542
80	34	4	64	65.0	36808
80	34	5	57	61.0	39333
80	34	6	50	57.0	39678
80	34	7	54	53.0	40691
80	34	8	58	48.0	40374
80	34	9	37	45.0	41747
80	34	10	33	41.0	43466

M	m	λ	Noder berørt	Prosent dårlige delinger	Byggetid
80	34	11	33	39.0	45163
80	34	12	31	35.0	44881
80	34	13	27	33.0	45958
80	34	14	32	31.0	46666
80	34	15	48	29.0	43539
80	34	16	29	27.0	47821
80	34	17	25	25.0	47835
80	36		102	89.0	30721
80	36	2	96	83.0	33776
80	36	3	70	78.0	35840
80	36	4	69	74.0	38252
80	36	5	77	69.0	38528
80	36	6	66	66.0	38795
80	36	7	57	63.0	41593
80	36	8	61	59.0	42074
80	36	9	51	52.0	42425
80	36	10	48	49.0	43596
80	36	11	42	48.0	44279
80	36	12	40	47.0	44943
80	36	13	40	43.0	46327
80	36	14	38	39.0	47036
80	36	15	40	38.0	47826
80	36	16	31	35.0	47868
80	36	17	41	32.0	48464
80	36	18	32	31.0	50151
80	38		103	95.0	30870
80	38	2	92	91.0	34427
80	38	3	116	87.0	34791
80	38	4	80	83.0	37917
80	38	5	86	80.0	37931
80	38	6	83	77.0	39627

M	m	λ	Noder berørt	Prosent dårlige delinger	Byggetid
80	38	7	96	73.0	40220
80	38	8	66	68.0	42339
80	38	9	59	64.0	42560
80	38	10	55	61.0	44068
80	38	11	57	59.0	45811
80	38	12	82	56.0	44233
80	38	13	55	52.0	46358
80	38	14	46	49.0	46975
80	38	15	40	46.0	49522
80	38	16	54	43.0	48084
80	38	17	54	40.0	51649
80	38	18	44	38.0	50965
80	38	19	35	37.0	51147
80	40		120	100.0	31255
80	40	2	91	98.0	35269
80	40	3	105	96.0	35128
80	40	4	115	94.0	35976
80	40	5	130	92.0	37004
80	40	6	107	88.0	39355
80	40	7	87	85.0	41887
80	40	8	82	82.0	42502
80	40	9	88	79.0	42352
80	40	10	73	75.0	43582
80	40	11	61	72.0	46458
80	40	12	77	69.0	45168
80	40	13	73	66.0	47743
80	40	14	50	60.0	48926
80	40	15	58	59.0	49885
80	40	16	69	56.0	49591
80	40	17	64	52.0	51641
80	40	18	57	48.0	50944
80	40	19	61	46.0	51046
80	40	20	68	44.0	52101