HT++, AIT-F08

Synopsis

|  |
| --- |
|  |

|  |  |
| --- | --- |
| **Vejleder(e):** | **Bjørn Klint Christensen (BJC)** |

|  |  |  |
| --- | --- | --- |
| **Deltager** | **Studienr.** | **Underskrift** |
| Lau Maack-Krommes | 5709 |  |
| Magnus Hemmer Pihl | 5452 |  |
| Mikkel Brøndsholm Nielsen | 5896 |  |
| Troels Hagmann-Hansen | 5711 |  |

Indholdsfortegnelse

[Forord 2](#_Toc198464106)

[1. Indledning 2](#_Toc198464107)

[2. Teori 2](#_Toc198464108)

[2.1 Neurale Netværk 2](#_Toc198464109)

[3. Implementation 2](#_Toc198464110)

[4. Test 2](#_Toc198464111)

[5. Konklusion 3](#_Toc198464112)

[5.1 Den produktorienterede del 3](#_Toc198464113)

[5.2 Den procesorienterede del 4](#_Toc198464114)

[Litteraturfortegnelse 4](#_Toc198464115)

[Ordforklaring 4](#_Toc198464116)

[Bilag 5](#_Toc198464117)

[Bilag 1. Projektoplæg 5](#_Toc198464118)

[Bilag 2. Detaljeret tidsplan 5](#_Toc198464119)

[Bilag 3. Beregning af ... (eksempel på bilag) 5](#_Toc198464120)

[Bilag 4. Måleopstilling (eksempel på bilag) 5](#_Toc198464121)

[Bilag 5. Dokumentation af processen (eksempel på bilag) 5](#_Toc198464122)

Forord

Denne rapportskabelon er udviklet til brug for Stærkstrøms-, IT- og Elektronik ingeniørstuderende på Ingeniørhøjskolen i København.  
Formålet er at støtte studerende i at skrive velstrukturerede og læsevenlige rapporter.

# Indledning

Denne rapport skal dokumentere vores arbejde med at lave et skakspil der kan give en menneskelig spiller modstand. Spillet er programmeret i C++ .

I rapporten vil vi forklare hvordan vi har implementeret spilintelligensen, hvilken teori der ligger bag og hvilke tanker vi har gjort os. Til sidst vil rapporten ud fra en række målinger med forskellige kompileringer af programmet konkludere hvilke metoder der optimerer søgningen i spiltræet bedst.

# Problemformulering

# Teori

## Spillet skak

## Minimax

MiniMax er en veludviklet algoritme, hvis formål det er at bestemme et fordelagtigt ”træk” i et spil. Algoritmen er, som algoritmer når de er bedst, ekstremt generel og kræver kun, at spillet er nulsum, at spillet har et endeligt antal stadier og at der er to spillere.

Strategien bag MiniMax er ikke nødvendigvis at lave de mest værdifulde træk, men derimod at forhindre modstanderen i at have gode muligheder på sigt. Med andre ord antager MiniMax algoritmen at modstanderen altid vil lave sit bedste træk og søger derfor at lave sit træk, sådan at modstanderen har færrest muligt gode muligheder.

Denne strategi vil altid vinde et hvilket som helst spil, hvis det er muligt og evalueringerne af spilstadierne er fornuftige.

Dette har naturligvis en pris. For at MiniMax kan være sikker på at lave de bedste træk, er den nødt til at tænke flere træk frem, sådan at den kan vinde på sigt. Det vil sige, at den bliver nødt til at gennemsøge samtlige træk fra alle stadier og denne proces kan, afhængigt af spillet, være alt fra banal til umulig. Problemet med for store spiltræer kan gøres bedre med AlphaBeta afskæring, som det beskrives i afsnit 3.5, men vil altid sætte grænser for de mere komplicerede spil, som skak.

Et andet element som indgår, er den førnævnte evalueringsfunktion. Denne er ikke integral for MiniMax algoritmen, men spiller alligevel en altafgørende rolle for dennes funktionalitet. Denne beskrives nærmere i afsnit 3.4.

## Udregning af træk

Udregning af træk er normalt ikke særligt interessant, i skak er der dog så mange forskellige brikker og et par special træk som gør udregningen lidt mere besværlig, et andet aspekt er at man ikke må stille sig selv skak.

Netop det at sætte sig selv skak, kan man enten lade træk generatoren sortere disse illegale træk fra, eller man kan ”håbe” på at ens MiniMax algoritme sortere disse fra.

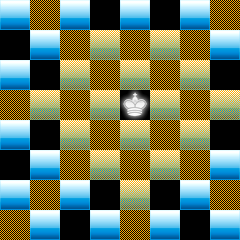
Hvis MiniMax algoritmen skal sortere de illegale træk fra er det vigtigt at evalueringsfunktionen fungere korrekt og at der regnes en del ply frem. Hvis der regnes meget få ply frem bringer man nemt sig selv i skak fordi algoritme først opdager sent at den er ved at blive trængt op i en krog.

Vores spil kan både generere legale træk og alle træk. Vi startede ud med at lave spillet så der kun blev udregnet legale træk. Senere lavede spillet om så det kunne vælges til og fra, med det håb at det ville sætte farten op på udregningen af trækkene. Det gjorde det også, men desværre har vi måttet konstatere at spillet ikke opfører sig pænt når vi bruge alle træk.

### Sådan udregnes træk

Udregningen af træk sker således

* Hvert enkelt felt på brættet løbes igennem
* Hvis et felt holder en af den nuværende spilleres brik udregnes dennes mulige træk.
* Hvis der kun må udføres legale træk tjekkes det om kongen er truet efter hver enkelt træk er udført.

Om kongen er truet udregnes ved at lade en dronning og en springer stå på kongens position. Lad dronning bevæge sig ud i alle retninger og stop hvis der en brik. Hvis brikken kan bevæge sig i den retning vi kom fra, er kongen truet.

Hvis der er springere på en af de otte springer positioner er kongen også truet.

## Statisk evaluering

## Alphabeta afskæring

AlphaBeta er en videreudvikling af MiniMax, der kraftigt forkorter udførelsestiden.

MiniMax antager altid, at modstanderen vil lave sit bedst mulige træk. Det vil sige, at hvis vi i et træ finder bare en enkelt mulighed for modstanderen, som er bedre for ham end det hidtil optimale vi har fundet, så kan vi være sikre på, at hele den gren nødvendigvis vil være dårligere, end hvad vi allerede har afsøgt.

På Figur 1 ses et eksempel på et spiltræ, hvor vi forsøger at opnå en maksimal værdi. Vi afsøger først det venstre træ, hvor vi får værdien 10, fordi vi antager at modstanderen (minimizer) vil vælge den mulighed der er bedst for ham. Begynder vi derefter at afsøge det højre træ, finder vi først værdien 7. Dette er en endnu bedre værdi for vores modstander end noget vi har fundet før; derfor ved vi med sikkerhed, at dette træ vil returnere en værdi der er 7 eller lavere – altså er der ingen grund til at afsøge resten af træet (i dette eksempel ville feltet med værdien 15 aldrig blive undersøgt), da vi ved at vi alligevel hellere vil vælge det venstre træ.

Figur : Eksempel på AlphaBeta afskærning

I dette simple eksempel er der ikke meget at vinde, men i større træer kan forgreningsfaktoren i bedste fald reduceres til kvadratroden af det normale.

## Sortering af træk

AlphaBeta afskæring er en utrolig stor optimering af MiniMax, men kun under de rette omstændigheder. Det optimale ville være at den fandt det bedste træk i første forsøg (da den derfor ville kunne skære flest muligt grene af træet), men hvis det var muligt, ville afsøgningen selvfølgelig være redundant til at starte med.

I stedet kan vi gøre det, at vi fodrer AlphaBeta’en med de mulige træk, som vi har en forventning om vil være bedre end gennemsnittet. Vi rammer sandsynligvis ikke rigtigt i første træk, men vi får under alle omstændigheder rigtig mange dårlige træk skåret fra med det samme.

Der er en række muligheder for, hvad der kunne være gode træk. Vi bruger tre tilfælde: Når man laver en rokade, når man forfremmer en bonde og når man kan tage en af modstanderens brikker.

I stedet for at lave en gammeldags sortering hvor vi sammenligner alle træk med hinanden har vi valgt at lave en lagdelt stak som lader os tilføje forskellige træk i de forskellige lag. Når trækkene skal evalueres kan de løbes igennem som en almindelig liste. Dette gør det muligt at tilføje god træk i starten af listen uden at skulle rykke samtlige træk foran.

## Caching af evalueringer

Den statiske evaluering af et spilstadie er bevidst lavet så den er effektiv, men det betyder på ingen måde at den er gratis. Hvad værre er, så skal den køres utroligt mange gange, på utroligt mange stadier. Interessant er det, at den også skal køres mange, mange gange på de samme stadier – dels på grund af, at vi altid søger mere end ét lag ned, dels fordi vi laver iterativ afsøgning.

Denne evaluering vil altid returnere det samme resultat for det samme stadie, med undtagelse af slutspillet, hvor kongens værdi kan mindskes hvis man sættes skak eller mat. Derfor vil vi, bortset fra disse situationer i slutspillet, kunne genbruge den samme beregning for hver gang et spilstadie dukker op.

Selve cachingen foregår helt traditionelt: En hash værdi for et board beregnes meget hurtigt. Ud fra denne slår vi op i en hash map, om værdien allerede er beregnet: Er den det, returnerer vi den. Ellers beregnes den som normalt.

Vi har opnået en lille ekstra optimering ved at lave vores eget hash map, som bruger vektore istedet for linkede lister til at lave den interne struktur. Det er ikke et map som er generelt anvendeligt, men til vores formål har det givet os lidt bedre ydelse.

Vores cache struktur rehasher som udgangspunkt ikke da det tager for lang tid. I stedet initialiseres den til en størrelse som bør understøtte udregningen af et par træk uden at cachen cleares. Når cachen når en hvis størrelse nulstilles den. Dette er endnu en grund til at vi ikke bruger linkede lister, da disse bliver lagt på heapen og skal allokeres og deallokeres hver gang der caches og cleares.

Den egentlige udfordring i at cache værdier er at finde en unik nøgle (hash værdi) for hvert spilstadie. Antallet af tænkelige stadier i skak løber op i fuldstændig uoverskuelige tal, som på ingen måde kan numereres individuelt. Albert Zobrist fandt i 1970 på den mulighed, at man lavede to sæt tilfældige værdier. Ud fra disse dannes to forskellige hash nøgler for et board. Når en gemt evaluering skal hentes fra cachen skal begge disse hash nøgler altså passe.

## Yderligere optimeringsmuligheder

I dette afsnit finder i en række ideer til forbedringer som desværre ikke har fundet deres vej ind i programmet. Dette er især beklageligt da det faktisk er da disse optimering ville gøre spillet væsentligt bedre.

I Kaare Danielsens slides om skak, beskriver han i hvilken rækkefølge træk bør udregnes for at lade gøre afskæringen i AlphaBeta algoritmen mest effektiv. I toppen af hans liste finder vi bedste træk fra sidste iteration, træk der tager den sidst rykkede brik og træk som lader modstanderen true dine brikker uden fare for selv at blive taget.

At genbruge det bedste træk fra sidste tur/iteration og at evaluere træk der tager det sidst rykkede brik, burde være nemme at implementere og det ville være en naturlig udvidelse.

### Bedste træk fra sidste tur

Det er nemt at se at alle disse tre ting ville kunne gøre afskæringen væsentligt bedre. Ser vi f.eks. på at genbruge det bedste træk fra sidste udregning, må vi regne med at få skåret store dele af spil træet af, fordi alle undtagen det sidste ply (hvis der bruges fast dybde) allerede blev evalueret i sidste iteration. Ser vi på et lille eksempel hvor vi først lader vores maximizer vælge sit træk og derefter gemmer det bedste træk som minimizeren efterfølgende kan tage. Det valgte træk er markeret med grønt.

Figur - Spiltræ

I næste iteration sker der naturligvis en ny evaluering i de yderste blade så værdierne i de enkelte noder er ændret lidt. Vi sørger for at det valgte træk bliver evalueret først, og kan således skære store dele af træet, væk. Hvis vi ikke havde sorteret evalueringen så dette træk var blevet analyseret til sidst havde det betydet at stort set hele træet ville blive evalueret.

Figur - Spiltræ med bedste træk først

Hvor meget der bliver evalueret vil naturligvis variere. At bruge det bedste træk fra sidste tur, minder meget om at bruge iterativ afsøgning som også gør afsøgning bedre.

Figur - Spiltræ med bedste træk til sidst

### Truende træk

En ting der er langt mere kompliceret at implementere er at evaluere træk som fjerner trusler først. Først skal det identificeres hvilke brikker der er truet, dette kan gøres på samme måde som når vi tjekker om kongen er truet. Når brikkerne er identificeret kan det så tjekkes hvorvidt de enkelte træk afhjælper nogen af truslerne.

Det at udregne trusler er ret krævende og logikken rimelig kompliceret. Vi har implementeret logikken til at tjekke efter trusler fordi det er krævet når der skal tjekkes for skak og skakmat, men vi regner ikke med at vi på et senere tidspunkt ville implementere dette for resten af brikkerne.

# Test

I den endelige version af vores AI har vi lavet tre primære optimeringer: AlphaBeta afskæring, sortering af træk, caching af evalueringer. Vi er naturligvis interesseret i, at teste i hvilken grad hver af disse har hjulpet algoritmen.

For at lave disse tests er vores program designet til, at kunne slå diverse optimeringer til og fra ved preprocessering. Desuden kan programmet udskrive data om spillets udførsel til en fil i MatLab format, sådan at vi kan tegne grafer direkte fra genereret data.

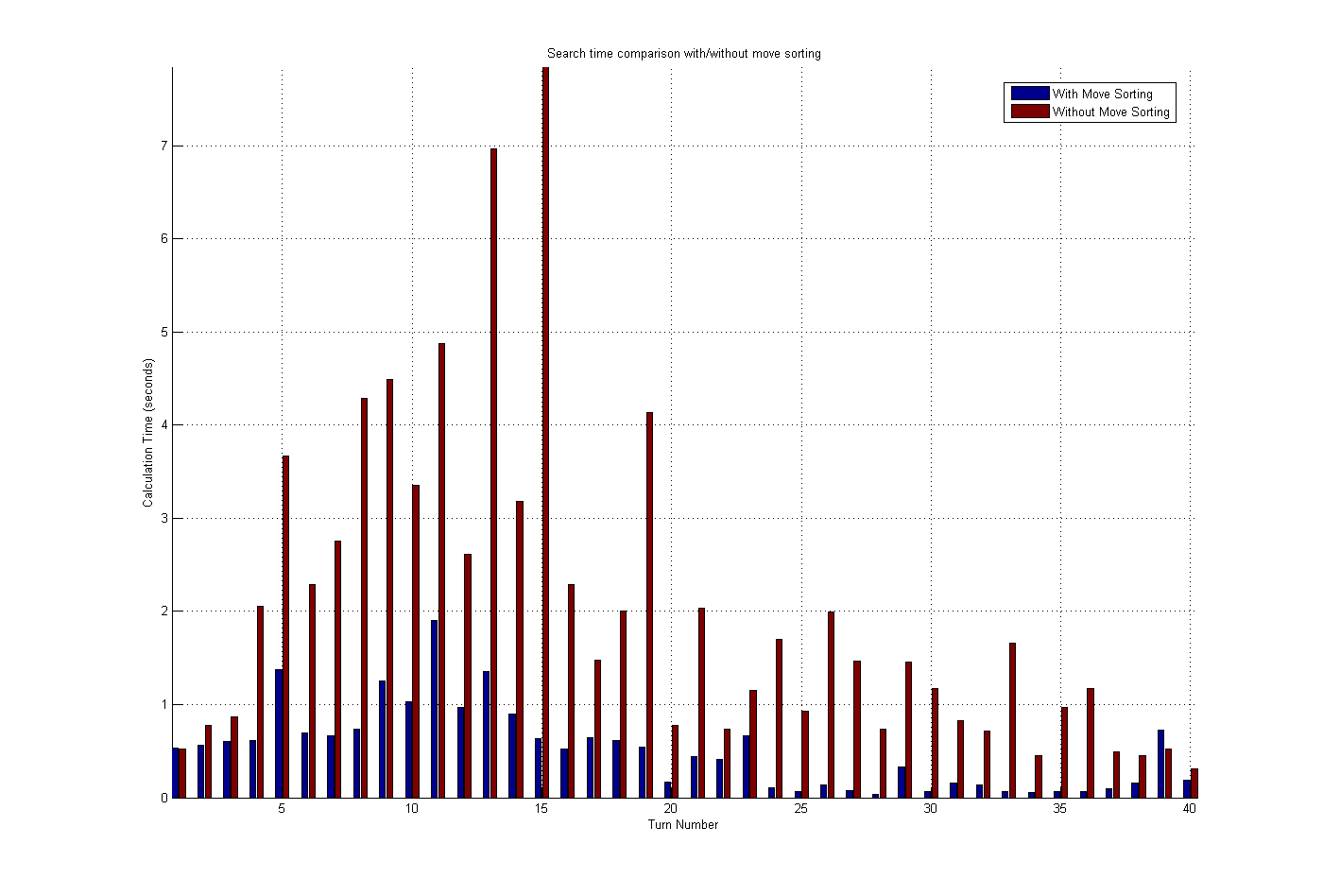
Hver af disse test tager de første 40 træk af et spil med to AI-kontrollerede spillere, der arbejder med den samme konfiguration. For hver test ændrer vi et enkelt parameter, såsom at slå caching fra, og sammenligner data, primært tidsmålinger, med et spil, der har kørt de samme træk, med vores standardkonfiguration.

Under normale forhold ville vi køre mange tests med de samme parametre og finde gennemsnitstiderne, i et forsøg på at undgå inteferens fra operativ systemet, men grundet den lange beregningstid for hvert spil, har vi besluttet at stole på, at elementer som time-slicing osv. ikke spiller en større rolle.

Alle de følgende grafer er blevet formindsket for at passe til dokumentets formatering. De kan findes i fuld størrelse som Bilag 1.

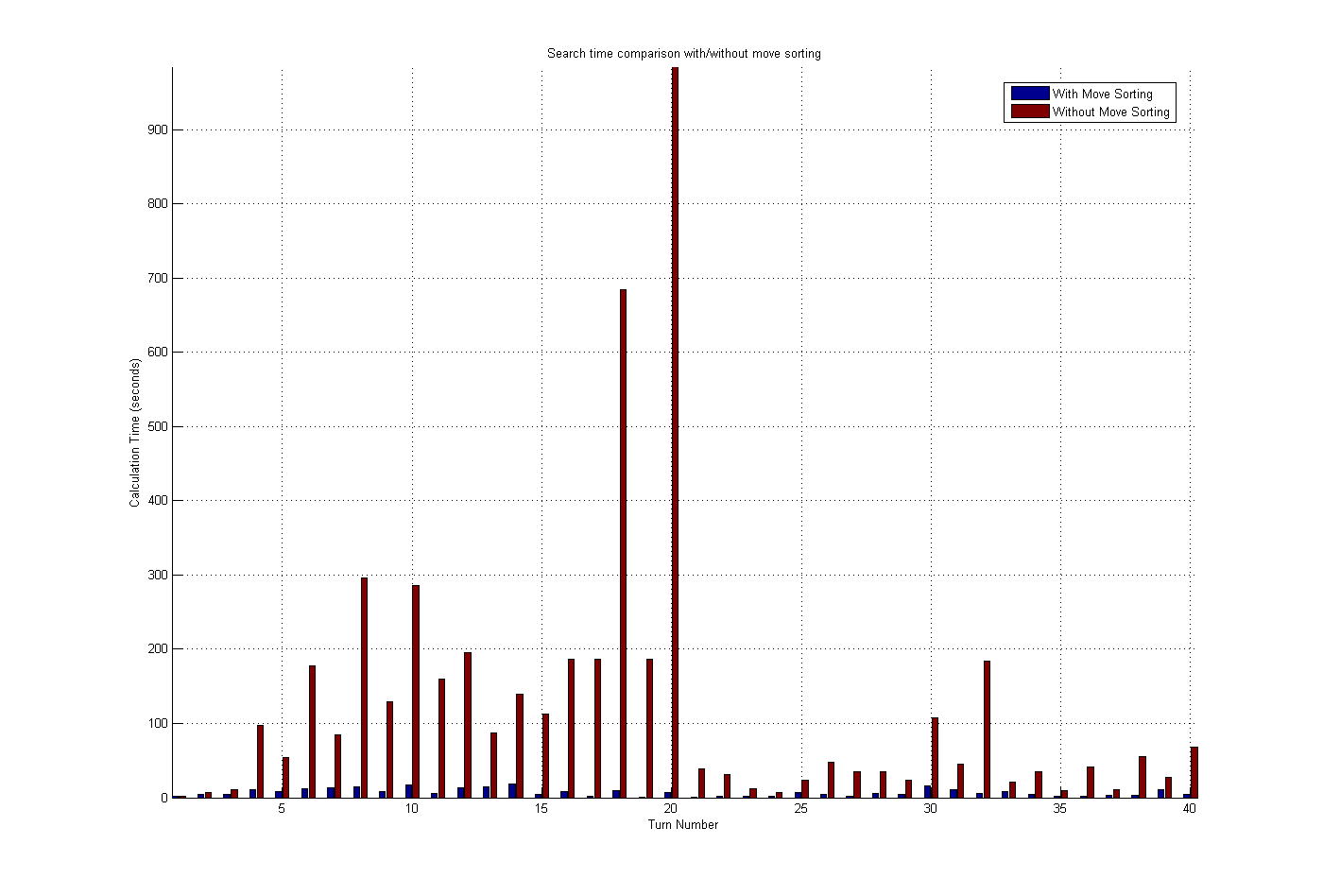
## AlphaBeta afskæring

## Sortering af træk

Som tidligere nævnt findes AlphaBeta afskæringens rigtige styrke i, hvor tidligt den finder den bedste mulighed. Her vil vi teste i hvor høj en grad det giver sig til kende i vores projekt.

Den første graf viser os beregningstiden per træk, når der sorteres med blåt, og når der ikke sorteres med rød (træk afsøges i den rækkefølge, som generatoren opdagede dem), med en søgedybde på fire ply.

Det er tydeligt at det i stort set alle tilfælde er en markant forbedring, når der sorteres. Kun i et enkelt træk (nr. 39) var det hurtigere uden sorteringen, hvilket skyldes at det bedste træk var et af dem der blev genereret først og at det ikke blev prioriteret af sorteringen. Desuden må vi antage, at denne situation kun vil opstå på grund af den meget hurtige beregningstid i dette eksempel.

Forsøger vi igen med fem ply, er billedet endnu mere overvældende.

Her bliver spillet ubrugeligt i mange af turene hvis der ikke sorteres, mens der er universelt lave søgtider med sortering.

Det er åbenlyst ud fra disse resultater, at denne optimering ikke alene er arbejdet værd – den er fuldstændig uundværlig når der spilles på højere niveauer.

## cache.pngCaching af evalueringer

I modsætning til de andre optimeringer er caching noget, der ikke er så åbenlys en forbedring. Hver gang noget skal caches, eller vi skal slå op i cachen, skal der laves hash-værdier for et board, og der skal slås op en i struktur, som kan involvere dynamisk memory. Det er ofre, som tager tid – det interessante er, om de sparer noget i sidste ende.

Denne graf er en direkte sammenligning af når cachen er slået til eller fra. Der er ikke den samme markante forskel, som vi har set tidligere. Mange af turene tager cirka lige lang tid, hvad enten vi bruger caching eller ej. Til gengæld bemærker vi, at når der endelig er en forskel, så er den klart i cachens favør. Det er desuden de ture, som tog markant længere tid at beregne end gennemsnittet. Altså tyder det på, at caching hjælper, når der virkelig er brug for det – hvilket giver mening: På de ture hvor der skal processeres mange træk (de ture der tager lang tid at beregne), vil de samme spilstadier også optræde oftere. Derfor vil cachen tilgås oftere og til slut give en større optimering.

# Konklusion

## Den produktorienterede del

## Den procesorienterede del

Litteraturfortegnelse

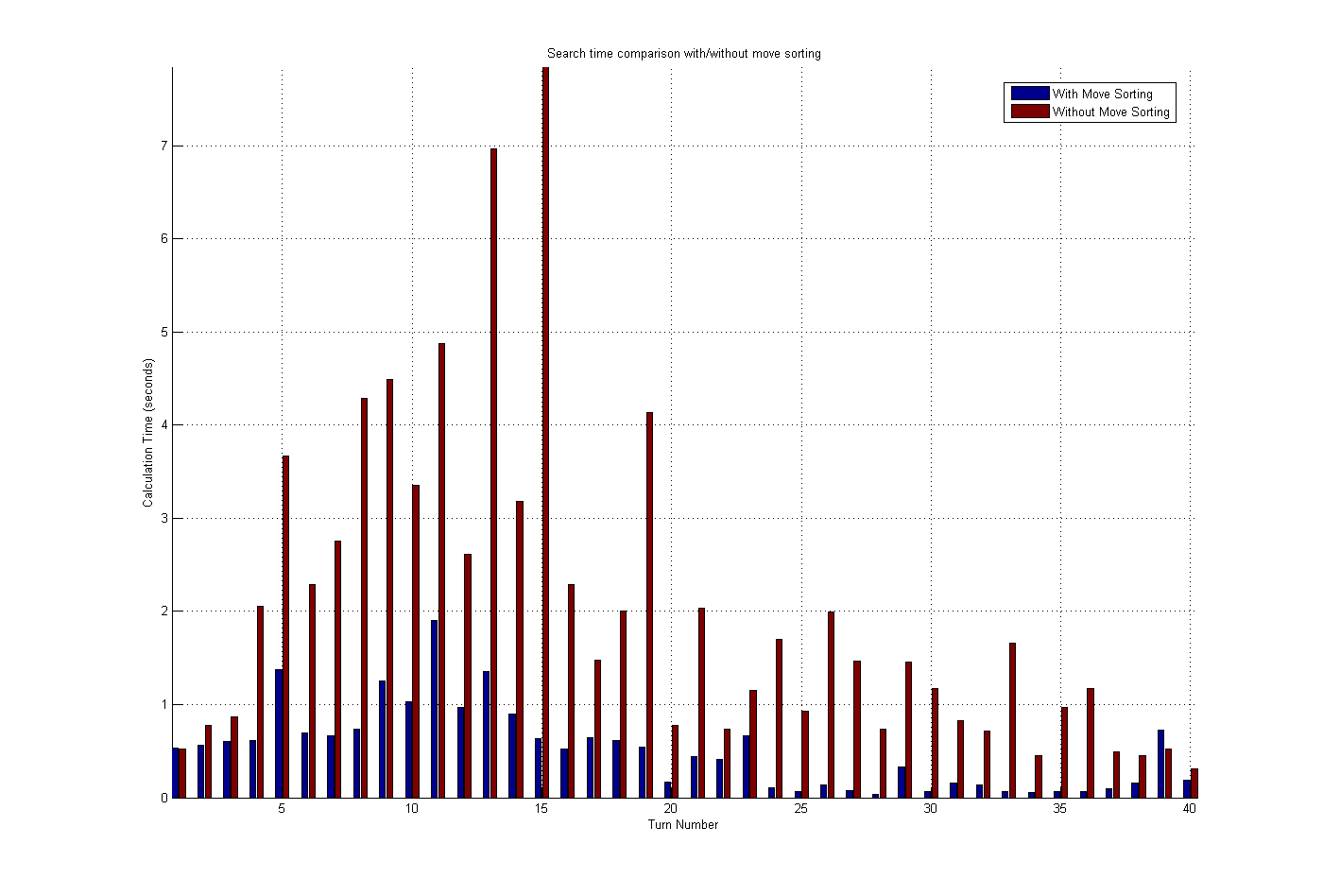
http://en.wikipedia.org/wiki/Zobrist\_hashing

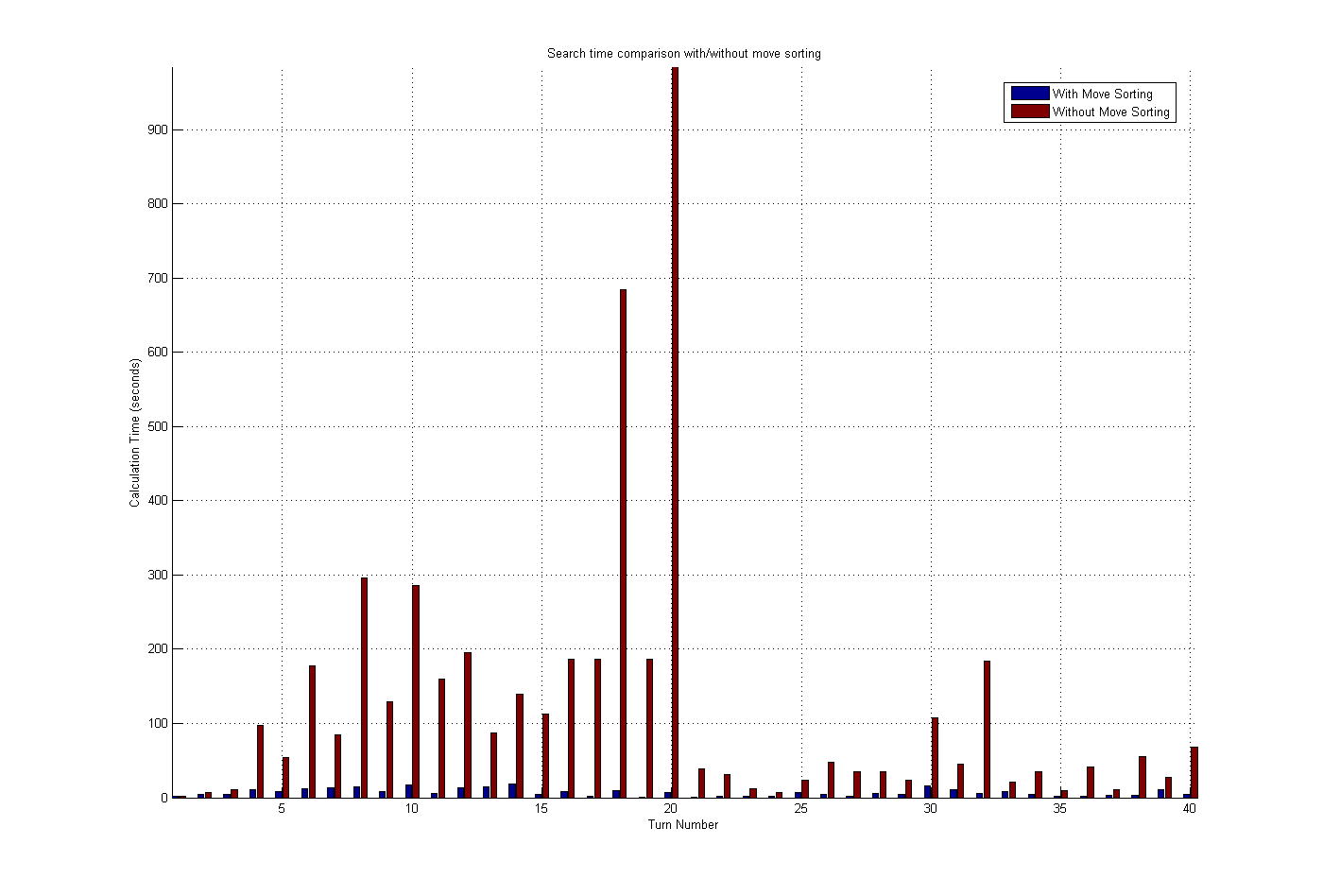
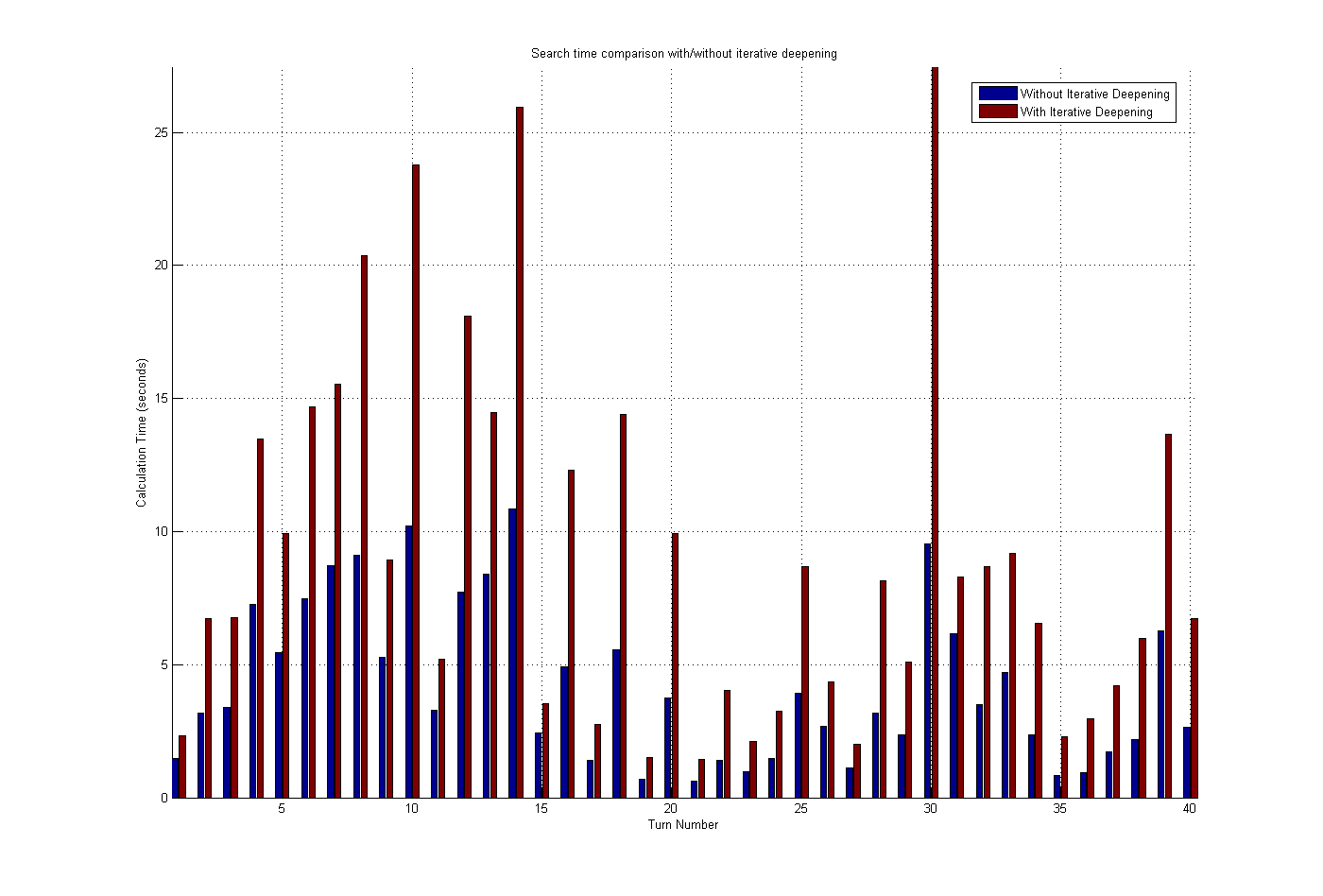
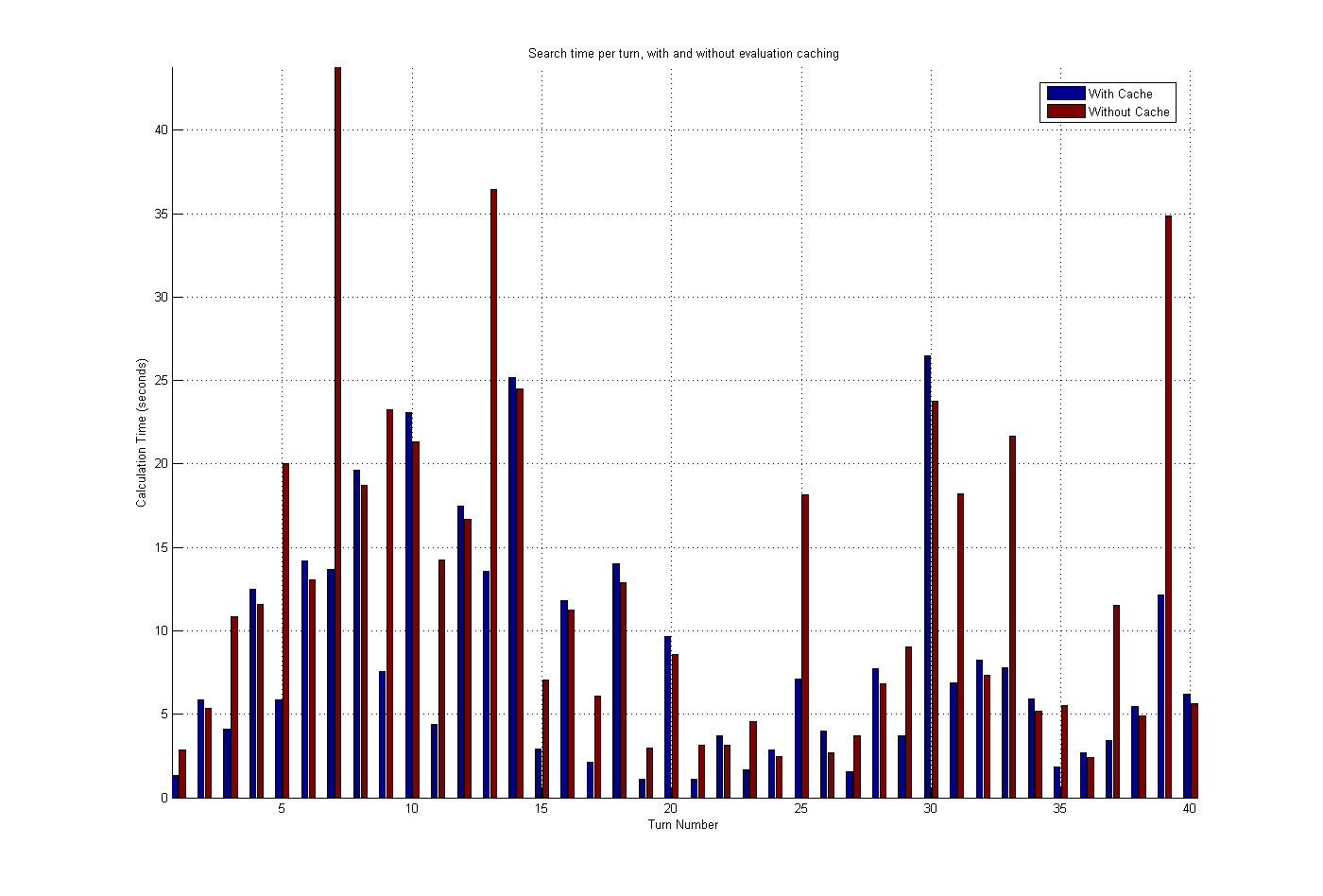
Ordforklaring

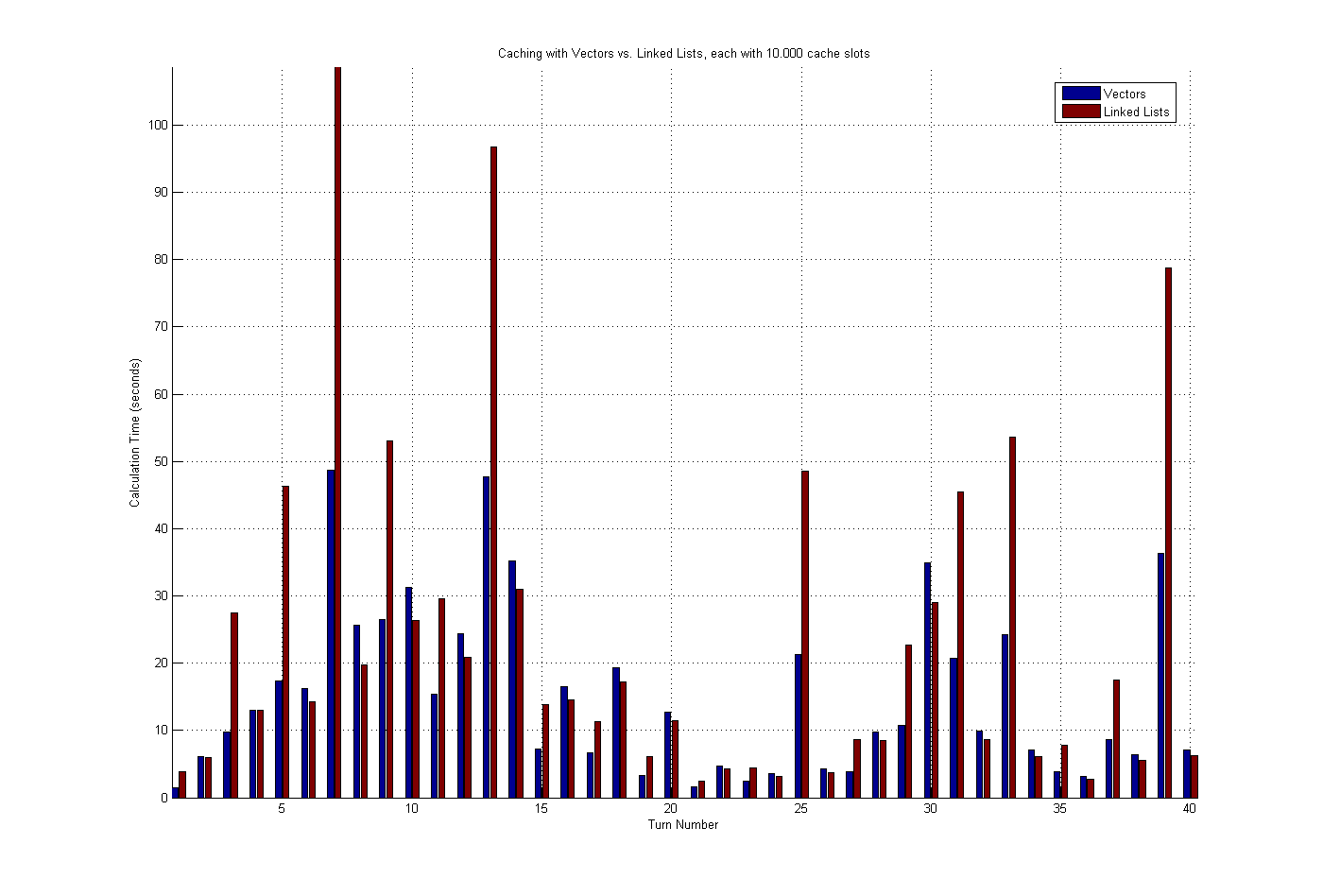
...

Bilag

1. Grafer i fuld størrelse







1. Detaljeret tidsplan

...

1. Beregning af ... (eksempel på bilag)

...

1. Måleopstilling (eksempel på bilag)

...

1. Dokumentation af processen (eksempel på bilag)