HT++, AIT-F08

|  |
| --- |
|  |

|  |  |
| --- | --- |
| **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

[1. Figurliste 2](#_Toc199586602)

[2. Ordforklaring 3](#_Toc199586603)

[3. Indledning 3](#_Toc199586604)

[4. Problemformulering 3](#_Toc199586605)

[4.1 Regler 4](#_Toc199586606)

[4.1.1 Sådan vindes spillet 6](#_Toc199586607)

[4.1.2 Uafgjort 6](#_Toc199586608)

[4.1.3 Interface 6](#_Toc199586609)

[4.1.4 Yderligere krav 7](#_Toc199586610)

[5. Spillet skak 7](#_Toc199586611)

[6. Teori 7](#_Toc199586612)

[6.1 Minimax 7](#_Toc199586613)

[6.2 Udregning af træk 8](#_Toc199586614)

[6.2.1 Sådan udregnes træk 9](#_Toc199586615)

[6.3 Statisk evaluering 9](#_Toc199586616)

[6.4 AlphaBeta afskæring 10](#_Toc199586617)

[6.5 Sortering af træk 11](#_Toc199586618)

[6.6 Caching af evalueringer 11](#_Toc199586619)

[6.7 Yderligere optimeringsmuligheder 12](#_Toc199586620)

[6.7.1 Bedste træk fra sidste tur og iterativ afsøgning 13](#_Toc199586621)

[6.7.2 Truende træk 15](#_Toc199586622)

[7. Implementering 15](#_Toc199586623)

[7.1 Systemets opbygning 15](#_Toc199586624)

[7.2 Skak 16](#_Toc199586625)

[7.3 AI 17](#_Toc199586626)

[8. Test 18](#_Toc199586627)

[8.1 AlphaBeta afskæring 19](#_Toc199586628)

[8.2 Sortering af træk 20](#_Toc199586629)

[8.3 Caching af evalueringer 21](#_Toc199586630)

[9. Konklusion 24](#_Toc199586631)

[Litteraturfortegnelse 25](#_Toc199586632)

[Bilag 25](#_Toc199586633)

[Bilag 1. Indhold af CD’en 25](#_Toc199586634)

# Figurliste

[Figur 1 - Skakbræt start tilstand 4](#_Toc199580370)

[Figur 2 - En-passant 5](#_Toc199580371)

[Figur 3 - Før og efter rokade 6](#_Toc199580372)

[Figur 4 - Trusler 9](file:///D:\IHK\6sem\AIT\ht-chess\AI%20Rapport.docx#_Toc199580373)

[Figur 5: Eksempel på AlphaBeta afskærning 10](file:///D:\IHK\6sem\AIT\ht-chess\AI%20Rapport.docx#_Toc199580374)

[Figur 6 - Spiltræ 14](#_Toc199580375)

[Figur 7 - Spiltræ med bedste træk først 14](#_Toc199580376)

[Figur 8 - Spiltræ med bedste træk til sidst 14](#_Toc199580377)

[Figur 9 - Klassediagram skak 16](#_Toc199580378)

[Figur 10 - 0x88 positionering 16](#_Toc199580379)

[Figur 11 - Klassediagram AI 17](#_Toc199580380)

[Figur 12 - Udregningstid af træk med og uden sortering AlphaBeta og MiniMax, 4 ply 19](#_Toc199580381)

[Figur 13 - Udregningstid med og uden sortering af træk, 4 ply 20](#_Toc199580382)

[Figur 14 - Udregningstid med og uden sortering af træk, 5 ply 21](#_Toc199580383)

[Figur 15 - Udregningstid med og uden caching, 5 ply, iterativ afsøgning 22](#_Toc199580384)

[Figur 16 - Udregningstid med og uden caching, 5 ply, normal søgning 23](#_Toc199580385)

# Ordforklaring

|  |  |
| --- | --- |
| Ord | Forklaring |
| Blad | En node i et træ som ikke har nogen børn. |
| Cache og caching | En cache er et lager. Dvs. at når vi cacher, gemmer vi information så det kan hives frem igen senere. |
| Forgrennings-faktor | Hvor mange nye træk der i snit opstår efter hvert træk. |
| Msb og lsb | Most significant bit og least significant bit. Det henholdsvis mest og mindst betydende bit i et binært tal eller struktur. |
| Ply | Et ply er en halv tur, dvs. et træk. Når både hvid og sort har udført deres træk, altså to ply, er der gået en tur. |
| Spiltilstand eller -stadie | En spiltilstand er en kombination af brikkernes position på brættet og om en-passant og rokade er muligt mm. |
| Spiltræ | Træ der indeholder de mulige handlinger spillerne kan tage. |

# 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

Vi ønsker at lave et skakspil som forhåbentligt kan slå Kaare Danielsens skakmaskine og give de andre spillere i turneringen modstand.

## Regler

Reglerne til skak kender de fleste. Her følger dog alligevel en kort opsummering af hvilke regler vi ønsker at implementere.

Spillets startposition er altid som på Figur 1. (Tegninger taget fra wikipedia)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | A | B | c | d | E | f | g | h |  |
| 8 | [a8 rd](http://en.wikipedia.org/wiki/Image:Chess_rdl44.png) | [b8 nd](http://en.wikipedia.org/wiki/Image:Chess_ndd44.png) | [c8 bd](http://en.wikipedia.org/wiki/Image:Chess_bdl44.png) | [d8 qd](http://en.wikipedia.org/wiki/Image:Chess_qdd44.png) | [e8 kd](http://en.wikipedia.org/wiki/Image:Chess_kdl44.png) | [f8 bd](http://en.wikipedia.org/wiki/Image:Chess_bdd44.png) | [g8 nd](http://en.wikipedia.org/wiki/Image:Chess_ndl44.png) | [h8 rd](http://en.wikipedia.org/wiki/Image:Chess_rdd44.png) | 8 |
| 7 | [a7 pd](http://en.wikipedia.org/wiki/Image:Chess_pdd44.png) | [b7 pd](http://en.wikipedia.org/wiki/Image:Chess_pdl44.png) | [a7 pd](http://en.wikipedia.org/wiki/Image:Chess_pdd44.png) | [b7 pd](http://en.wikipedia.org/wiki/Image:Chess_pdl44.png) | [a7 pd](http://en.wikipedia.org/wiki/Image:Chess_pdd44.png) | [b7 pd](http://en.wikipedia.org/wiki/Image:Chess_pdl44.png) | [a7 pd](http://en.wikipedia.org/wiki/Image:Chess_pdd44.png) | [b7 pd](http://en.wikipedia.org/wiki/Image:Chess_pdl44.png) | 7 |
| 6 | [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) | [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) | [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) | [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) | [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) | [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) | [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) | [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) | 6 |
| 5 | [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) | [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) | [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) | [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) | [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) | [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) | [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) | [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) | 5 |
| 4 | [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) | [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) | [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) | [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) | [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) | [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) | [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) | [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) | 4 |
| 3 | [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) | [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) | [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) | [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) | [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) | [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) | [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) | [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) | 3 |
| 2 | [a2 pl](http://en.wikipedia.org/wiki/Image:Chess_pll44.png) | [b2 pl](http://en.wikipedia.org/wiki/Image:Chess_pld44.png) | [a2 pl](http://en.wikipedia.org/wiki/Image:Chess_pll44.png) | [b2 pl](http://en.wikipedia.org/wiki/Image:Chess_pld44.png) | [a2 pl](http://en.wikipedia.org/wiki/Image:Chess_pll44.png) | [b2 pl](http://en.wikipedia.org/wiki/Image:Chess_pld44.png) | [a2 pl](http://en.wikipedia.org/wiki/Image:Chess_pll44.png) | [b2 pl](http://en.wikipedia.org/wiki/Image:Chess_pld44.png) | 2 |
| 1 | [a1 rl](http://en.wikipedia.org/wiki/Image:Chess_rld44.png) | [b1 nl](http://en.wikipedia.org/wiki/Image:Chess_nll44.png) | [c1 bl](http://en.wikipedia.org/wiki/Image:Chess_bld44.png) | [d1 ql](http://en.wikipedia.org/wiki/Image:Chess_qll44.png) | [e1 kl](http://en.wikipedia.org/wiki/Image:Chess_kld44.png) | [f1 bl](http://en.wikipedia.org/wiki/Image:Chess_bll44.png) | [g1 nl](http://en.wikipedia.org/wiki/Image:Chess_nld44.png) | [h1 rl](http://en.wikipedia.org/wiki/Image:Chess_rll44.png) | 1 |
|  | a | B | C | d | E | f | g | h |  |

Figur - Skakbræt start tilstand

Et felt er under angreb hvis en af modstanderens brikker kan rykke dertil.

[a2 pl](http://en.wikipedia.org/wiki/Image:Chess_pll44.png) Bønder kan rykke et felt lige frem, hvis der på destinationen ikke allerede står en brik. Hvis bonden ikke har rykket før kan han rykke to felter frem, hvis begge felter er frie. Bonden kan tage brikker der står et diagonalt skridt foran dem. Hvide bønder kan kun rykke mod række 8, altså væk fra deres udgangspunkt, de sorte kan kun rykke den modsatte vej.

Hvis en bonde rykker to felter frem, og modstanderen har en bonde i feltet direkte til højre eller venstre for destination, så har modstanderen ret til at udføre en-passant. Når man udfører en-passant tager man modstanderens bonde på halvvejen som vist på Figur 2.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) | [b7 pd](http://en.wikipedia.org/wiki/Image:Chess_pdl44.png) | [b2 pl](http://en.wikipedia.org/wiki/Image:Chess_pld44.png) | [b7 pd](http://en.wikipedia.org/wiki/Image:Chess_pdl44.png) | [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) | [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) |
| [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) | [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) | [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) | [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) | [b7 pd](http://en.wikipedia.org/wiki/Image:Chess_pdl44.png) | [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) |
| [b2 pl](http://en.wikipedia.org/wiki/Image:Chess_pld44.png) | [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) | [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) | [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) | [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) | [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) |

Figur - En-passant

Hvis en hvid bonde når til række 8 eller en sort når til række 1 kan spilleren i samme tur forfremme bonden til en valgfri brik (undtagen kongen). I vores spil vil bønder automatisk blive forfremmet til dronninger.

[a1 rl](http://en.wikipedia.org/wiki/Image:Chess_rld44.png) Tårnet kan rykke vertikalt og horisontalt så langt den ønsker, så længe en brik ikke står i vejen.

[b1 nl](http://en.wikipedia.org/wiki/Image:Chess_nll44.png) Hesten kan rykke i L form uanset om der er brikker mellem den og destinationen. Hesten kan fra sin position rykke: (2,1), (2,-1), (-2, 1), (-2,-1), (1, 2), (1, -2), (-1, 2), (-1, -2).

[c1 bl](http://en.wikipedia.org/wiki/Image:Chess_bld44.png) Løberen kan rykke diagonalt på felter af samme farve som det felt den startede på. Løberen kan rykke så langt den ønsker så længe der ikke er brikker i vejen.

[d1 ql](http://en.wikipedia.org/wiki/Image:Chess_qll44.png) Dronning kan rykke både som tårnet og løberen.

[e1 kl](http://en.wikipedia.org/wiki/Image:Chess_kld44.png) Kongen kan rykke til alle de otte tilstødende felter. Kongen må ikke rykke til felter der er under angreb. Kongen kan udføre rokade med et af de to tårne, hvis der ikke står nogen brikker mellem kongen og det valgte tårn. Hverken kongen eller tårnet har rykket, og kongen ikke er under angreb i nogle af de felter der passeres. En rokade startes ved at rykke kongen.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Damesiden | | | | Kongesiden | | | |
| [a8 rd](http://en.wikipedia.org/wiki/Image:Chess_rdl44.png) | [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) | [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) | [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) | [e8 kd](http://en.wikipedia.org/wiki/Image:Chess_kdl44.png) | [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) | [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) | [h8 rd](http://en.wikipedia.org/wiki/Image:Chess_rdd44.png) |
| [a1 rl](http://en.wikipedia.org/wiki/Image:Chess_rld44.png) | [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) | [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) | [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) | [e1 kl](http://en.wikipedia.org/wiki/Image:Chess_kld44.png) | [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) | [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) | [h1 rl](http://en.wikipedia.org/wiki/Image:Chess_rll44.png) |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) | [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) | [e8 kd](http://en.wikipedia.org/wiki/Image:Chess_kdl44.png) | [h8 rd](http://en.wikipedia.org/wiki/Image:Chess_rdd44.png) | [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) | [h8 rd](http://en.wikipedia.org/wiki/Image:Chess_rdd44.png) | [e8 kd](http://en.wikipedia.org/wiki/Image:Chess_kdl44.png) | [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) |
| [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) | [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) | [e1 kl](http://en.wikipedia.org/wiki/Image:Chess_kld44.png) | [h1 rl](http://en.wikipedia.org/wiki/Image:Chess_rll44.png) | [d6 __](http://en.wikipedia.org/wiki/Image:Chess_d44.png) | [h1 rl](http://en.wikipedia.org/wiki/Image:Chess_rll44.png) | [e1 kl](http://en.wikipedia.org/wiki/Image:Chess_kld44.png) | [a6 __](http://en.wikipedia.org/wiki/Image:Chess_l44.png) |

Figur - Før og efter rokade

### Sådan vindes spillet

Når modstanderen er skakmat er spillet vundet. Man er skakmat hvis kongen er under angreb uden at kunne flytte væk eller fjerne truslen på anden vis.

### Uafgjort

Spillet er uafgjort hvis en spiller ikke har nogen lovlige træk tilbage, dvs. at han ikke kan lave et træk, uden at sætte sin egen konge skak eller mat.

Hvis der er gået 50 træk siden sidste ryk med en bonde eller sidste gang en brik er blevet taget, er spillet også uafgjort.

### Interface

Som minimum skal programmet have et tekstbaseret interface. Vi vil dog meget gerne have et grafisk interface hvor musen også kan bruges til at rykke brikkerne med.

### Yderligere krav

Spillet skal kunne få sat en tidsbegrænsning som sørger for at et træk altid udregnes indenfor x antal sekunder – til turneringen blev denne grænse sat til 30 sekunder.

# Spillet skak

Spillet skak er interessant i AI sammenhæng fordi det er et nulsum spil med et stort tilstandsrum. Nulsum spil er spil, der ikke baserer sig på tilfældigheder som terningkast. Dvs. at man ved en logisk undersøgelse kan bestemme hvilket træk der må være det bedste. Udtrykket nulsum knytter sig til den logiske evaluering som fortæller om en tilstand er til ens fordel eller ej.

En tilstand evalueres for hver spiller og hvis summen er nul betyder det at tilstanden er lige god eller dårlig for begge spillere. Hvis summen er større eller mindre end nul vil den ene spiller ”føre”.

At skak er et nulsum spil gør det muligt at lade en MiniMax, med eller uden AlphaBeta afskæring, løbe spiltræet igennem og ved hjælp af en statisk evaluering vurdere hvorvidt et træk på længere sigt vil være til ens fordel.

I spil med små tilstandsrum, som for eksempel tre på stribe, er afsøgning af spiltræet ikke særligt tidskrævende og det er nemt at lave smarte træk.

Men da skak har et meget stort tilstandsrum ca. 2.99\*1041 (jf. <http://www.chesshub.com/faq/cchess/>), er det ikke så simpelt at beregne. Dette tal er så stort at det ikke kan gemmes i en computer, hvilket gør det svært at udregne gode træk på en fornuftig tid. Vi kan ikke blot kigge i en tabel og finde svaret – vi bliver nødt til at udregne og estimere værdien af de gode træk. Pga. kompleksiteten er der altså incitament og mulighed for optimeringer.

# Teori

## 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 sørger derfor for, at lave sit træk, sådan at modstanderens bedste træk er dårligst muligt.

Denne strategi vil altid vinde et hvilket som helst spil, hvis det er muligt og evalueringerne af spilstadierne er fornuftige, forudsat at algoritmen søger til bunden af spiltræet.

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 afhjælpes med AlphaBeta afskæring, som det beskrives i afsnit 6.4, 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 6.2.

## Udregning af træk

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

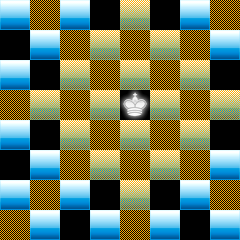
Netop det at sætte sig selv skak, kan man enten lade træk generatoren om at sortere fra, eller man kan ”håbe” på at ens MiniMax algoritme sorterer disse fra, en strategi der giver fart men som er svær at sikre mod fejl.

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

Vores spil kan enten udelukkende generere legale træk, eller alle træk, alt efter opsætning. Vi startede ud med at lave spillet så der kun blev udregnet legale træk. Senere lavede vi 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 bruger alle træk. Det skyldes at evalueringsfunktionen blot kontrollerer, om vi er skak eller skakmat og ikke om trækket er lovligt.

### Sådan udregnes træk

Udregningen af træk sker således

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

Figur - Trusler

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 der står på feltet 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

Evaluatoren beskriver stillingen mellem de to spillere som en numerisk værdi ud fra brættets tilstand. Det er derfor evalueringsfunktionen der giver AI'en viden om, hvilke tilstande der er fordelagtige. Vi har i vores implementering taget højde for hver enkelt brik og givet disse point ud fra hvor vigtige de er og ligeledes for deres placering. F.eks. er det ikke godt at have 2 bønder foran hinanden, fordi de spærrer for hinanden.

Vi har således taget udgangspunkt i Kaare Danielsens noter. Evalueringsfunktionen er ikke optimal for netop vores spil, fordi vi både har kort og lang rokade, hvorimod Kaares evalueringsfunktion kun er rettet mod kort rokade. Det vil sige, at bønderne ikke placerer sig fornuftigt for at lave lang rokade.

Evalueringsfunktionen ændrer adfærd alt efter hvor mange brikker der er tilbage i spillet. I slutspillet er det lettere at undgå at få sin konge fanget, hvis denne er tæt på midten, hvorimod det i starten er smartest at holde kongen godt beskyttet. Derfor giver tilstande, hvor kongen er i midten, højere værdi i slutspillet.

Vores evalueringsfunktion dækker som sagt alle disse ting. Yderligere optimeringer kunne være, at genkende specielle tilstande som starten på finter eller at kunne genkende mønstre. Dette vil især udmærke sig i starten og i slutspillet, hvor man kan udvælge komplicerede finter til at slå modstanderen.

## 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 træk, 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 forsimplet spiltræ, hvor vi forsøger at opnå en maksimal værdi. Vi afsøger først den venstre del af 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 modstanderen, hvis han spiller optimalt, altid vil forhindre de bedre træk i den del af træet.

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 almindelig 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 gode 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, at 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.

I iterativ afsøgning øges søgedybden per iteration så de første ply vil blive regnet igen og igen.

Evalueringen vil altid returnere det samme resultat for det samme stadie da den er statisk, med undtagelse af de stadier hvor kongen er sat skakmat eller spillet er uafgjort, da disse afhænger af dybden. Derfor vil vi, bortset fra disse situationer, 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 et 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 i cachen. Det er ikke et map som er generelt anvendeligt, men til vores formål har det givet os lidt bedre ydelse i nogle situationer.

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 flyder over. Hvis cachen flyder over slettes alt indholdet. Dette er endnu en grund til at vi ikke bruger linkede lister, da hver enkelt node i listerne 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 nummereres 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 spilstadie. 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 alle har fundet deres vej ind i programmet.

I Kaare Danielsens slides om skak, beskriver han i hvilken rækkefølge træk bør udregnes for at gøre afskæringen i AlphaBeta algoritmen mest effektiv. I toppen af hans liste finder vi bedste træk fra sidste iteration. Da vi ved at han har implementeret iterativ afsøgning er vi i tvivl om det knytter sig til det, eller om det i virkeligheden er bedste træk fra sidste tur. Begge dele vil kunne hjælpe. Kaare vil også gerne have flyttet træk der tager sig af trusler flyttet op i listen. Derudover vil det nok også kunne betale sig at flytte træk der tager den sidst flyttede brik op i sorteringen.

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 af spillet, som vi desværre ikke har nået.

### Bedste træk fra sidste tur og iterativ afsøgning

Ser vi på at genbruge det bedste træk fra sidste udregning, må vi regne med at få skåret store dele af spiltræet af. Fordi alle undtagen det sidste ply (hvis der bruges fast dybde) allerede er blevet evalueret i sidste iteration.

Vi har desværre ikke iplementeret at bruge det bedste træk fra sidste tur. Til gengæld bruger vi idéen til at lave iterativ afsøgning. Dvs. at vi i stedet for at søge hele vejen ned igennem spiltræet først afsøger de to først ply, så sorterer vi det første ply efter hvilke træk der ser bedst ud lige nu. Vi afsøger så et ply længere ned, sorterer, og gentager ned til den dybde vi ønsker. Iterativ afsøgning ser i vores implementation ud til at være lidt langsommere end en normal afsøgning, men den har den fordel at den tidligere kan give et praj om hvilket træk der er godt.

Her ser vi på et 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

Iterativ afsøgning er ikke særligt givtigt ved små dybder, men når vi søger mange ply ned og forgrenningsfaktoren er høj bliver afskæringen meget højere.

### 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.

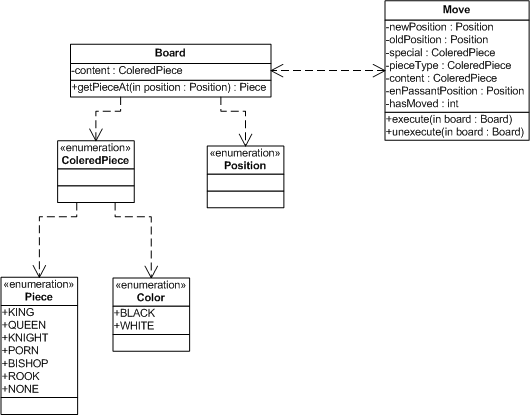
Desværre er det kostbart at tjekke for trusler og logikken er rimelig kompliceret, så hvis det skal være mere effektivt skal der findes mere effektive metoder, evt. med bitboards.Vi har implementeret logikken til at kontrollere efter trusler, fordi det er krævet når der skal kontrolleres for skak og skakmat, men vi regner ikke med at vi på et senere tidspunkt ville implementere dette for resten af brikkerne.

# Implementering

## Systemets opbygning

For at forstå opbygningen af spillet har vi delt det op i flere dele: en skak, AI og interface del. Interfacet er ikke interessant for faget og vil derfor ikke blive gennemgået i dybden her. Først vil vi kigge på skak logikken.

## Skak



Figur - Klassediagram skak

På Figur 9 ser vi et simplificeret klassediagram for skak logikken i vores spil. Navnene skulle gerne være umiddelbart forståelige.

**Board** er klassen der holder vores repræsentation af skakbrættet. Brættet gemmes som en vector<ColoredPiece> men bruges i praksis som et almindeligt array. I stedet for blot at gemme de 64 felter lineært bruger vi 0x88 repræsentation, dvs. Række-koordinatet gemmes i de laveste tre bit og kolonne-koordinatet i bit 4-6.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 (msb) | 6 | 5 | 4 | 3 | 2 | 1 | 0 (lsb) |
| - | x | x | x | - | y | y | y |

Figur - 0x88 positionering

Det smarte ved 0x88 repræsentation er at begge koordinater kan gemmes i en variabel og at det er nemt at tjekke for overflow, ved blot at tjekke om bit 7 eller 3 er sat.

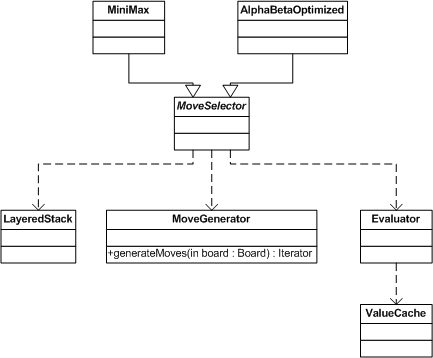
Board er en meget stor klasse og indeholder også logikken til at bestemme om en brik er under angreb eller om en spiller er skakmat. Board kan også finde alle mulige træk fra en specifik position eller for bestemte brikker.

**ColoredPiece** er en enumeration som indeholder de forskellige **Piece**s OR’ed med **Color**.

**Position** indeholder alle de lovlige positioner på brættet, samt INVALID\_POSITION som er 0x88.

**Move** repræsenterer overgang mellem to spilstadier, altså et træk. Move indeholder al data nødvendig for at ændre brættets tilstand og de data der er nødvendig for at ophæve ændringen igen.

## AI



Figur - Klassediagram AI

Når et træk skal udregnes beder spillet en **MoveSelector** klasse om at finde det bedste træk. MoveSelector er navnet på den h-fil som vores MiniMax algoritmer ligger i. Vi har blot for forståelsens skyld taget den med her for at vise at **MiniMax** og **AlphaBetaOptimized** fungerer som klasser der følger et fælles interface.

De to MoveSelector klasser bruger så **MoveGenerator** som løber brættets felter igennem og beder Board generere de mulige træk fra hver af dem. Træk bliver placeret i en **LayeredStack**. LayeredStack lader os placere de genererede træk i forskellige lag og løbe dem igennem som om de var på en lang liste. Dette sparer os for at lave en langsommelig sortering eller ombytning af træk i listen. LayeredStack gør det også muligt at låse de træk der er genereret i et ply sådan så vi i det næste kun ser de nye vi genererer. Når vi vender tilbage til det foregående ply kan listen automatisk fjerne/poppe alt det nye indhold.

Hvorvidt det i praksis var nødvendigt at lave LayeredStack kan diskuteres. Hvis vi regner med at der i hver node genereres 30 - 40 træk betyder det at en almindelig sortering ikke ville være fuldstændig usandsynlig at gøre brug af. Vi har desværre ikke implementeret sortering af de genererede træk i andet end det første ply, da det var nødvendigt til iterativ afsøgning, og kan derfor ikke sammenligne hvilken metode der er hurtigst.

Til den statiske evaluering af træk har vi **Evaluator.** Hvis caching er slået til tjekker Evaluator om brættet er i en stilling der kan caches og forsøger så at finde en eventuelt gemt evaluering fra **ValueCache**. Hvis en gemt evaluering ikke findes genererer Evaluator den og beder ValueCache om at gemme den.

# Test

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

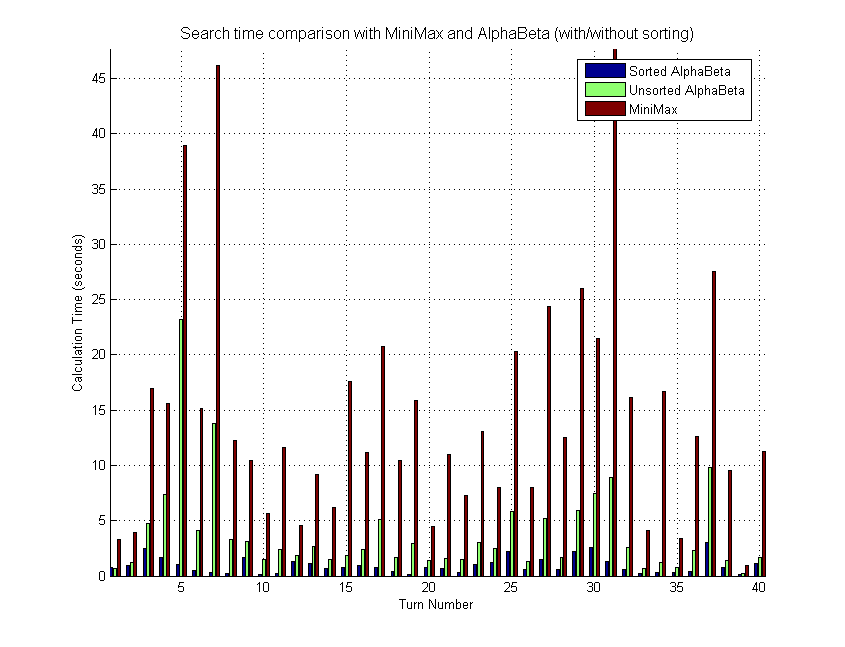
For at lave disse tests er vores program designet til, at kunne slå diverse optimeringer til og fra ved preprocessering. Desuden kan programmet udskrive data om spillets gang til en MatLab-fil, så vi kan tegne grafer direkte fra de genererede data.

Hver af disse tests tager de første 40 træk af et spil med to computer 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 interferens fra operativsystemet, 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.

## AlphaBeta afskæring

Som udgangspunkt vil vi gerne vise at AlphaBeta i praksis giver en væsentlig bedre ydelse end MiniMax.



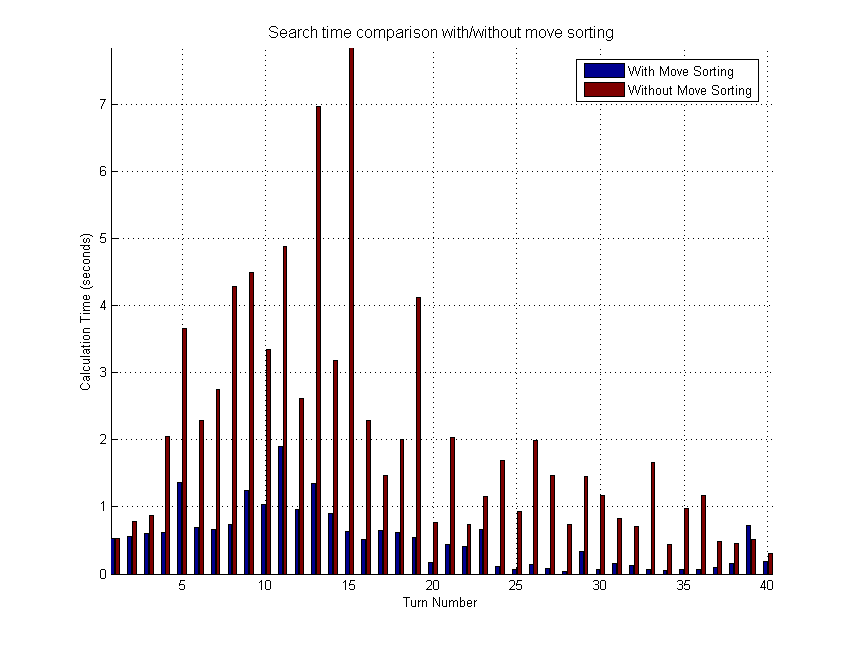
Figur - Udregningstid af træk med og uden sortering AlphaBeta og MiniMax, 4 ply

På Figur 12 kan ses en sorteret og en usorteret AlphaBeta holdt op imod en almindelig MiniMax. Grafen viser 4 plys afsøgning; vi har ikke lavet udregningerne for 5 ply da det ville tage alt for lang tid. Det kan meget nemt ses at AlphaBeta, selv usorteret, er væsentligt bedre end MiniMax.

## Sortering af træk

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

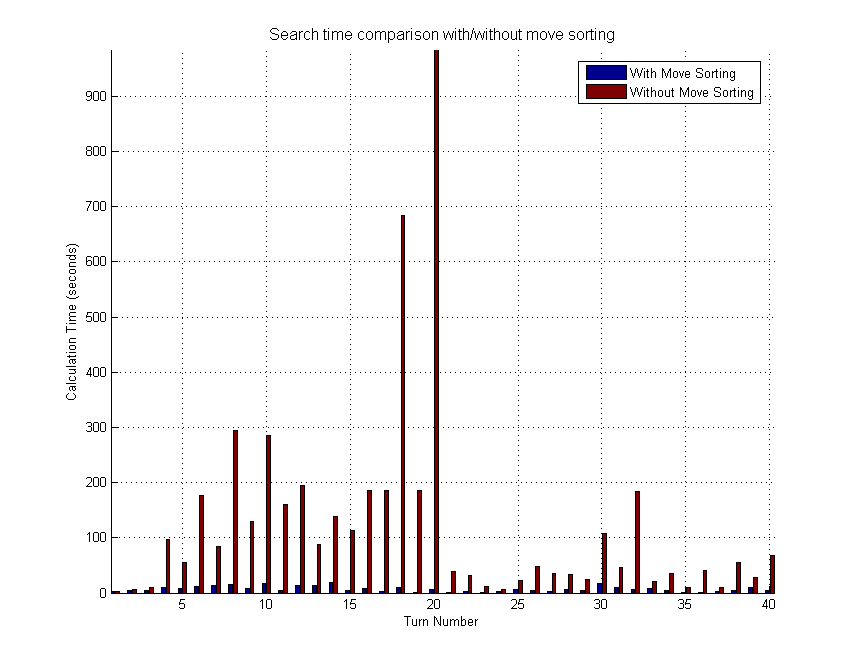
Den første graf viser os ber­egningstiden per træk, når der sorteres med blåt, og når der ikke sorteres med rødt (træk afsøges i den rækkefølge, som generatoren opdager dem), med en søgedybde på fire ply.



Figur - Udregningstid med og uden sortering af træk, 4 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, og kun i sjældne tilfælde.

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



Figur - Udregningstid med og uden sortering af træk, 5 ply

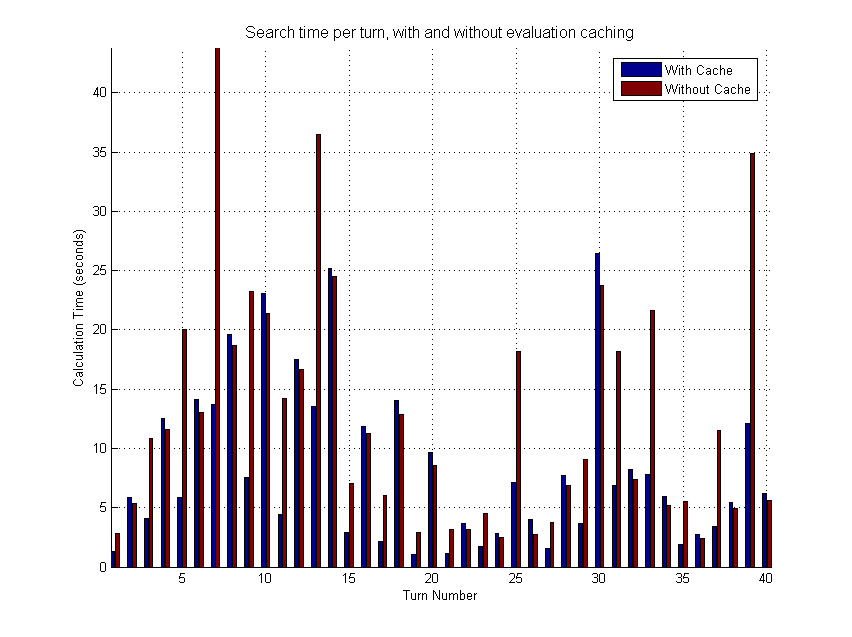
Her bliver spillet ubrugeligt i mange af turene hvis der ikke sorteres, mens der er universelt lave søgetider med sortering.

Det er åbenlyst ud fra disse resultater, at denne optimering ikke alene er arbejdet værd – den er fuldstændig uundværlig når der spilles med høj dybde.

## Caching 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 i en struktur, som kan involvere dynamisk memory. Det er ting, som tager tid – det interessante er, om de sparer noget i sidste ende.

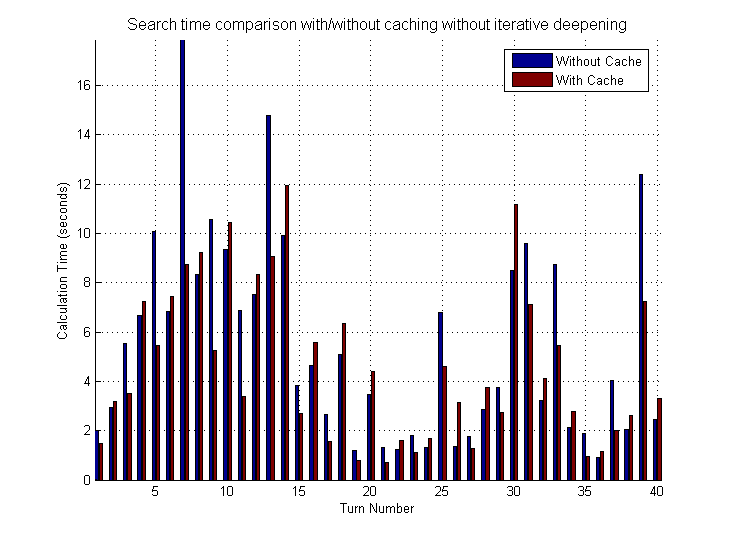
Denne graf er en direkte sammenligning af når cachen er slået til eller fra, når iterativ afsøgning er slået til. Her er ikke den samme markante forskel, som vi har set tidligere.



Figur - Udregningstid med og uden caching, 5 ply, iterativ afsøgning

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.

Ser vi derimod på effekten af caching når der *ikke* benyttes iterativ afsøgning, er resultaterne mindre positive.



Figur - Udregningstid med og uden caching, 5 ply, normal søgning

Under iterativ afsøgning støder vi automatisk på de samme spilstadier, hver gang vi starter forfra på afsøgningen. Uden iterativ afsøgning er der væsentlig mindre gentagelse, hvilket betyder at cachen ikke bliver benyttet nær så ofte. I dette tilfælde ligger søgetiderne rimelig tæt det meste af tiden, men typisk lidt langsommere med caching. Ud fra dette kan vi konkludere at den ekstra udregningstid involveret i cachingen overgår det, vi vinder på det, når der er få gentagelser i afsøgningen. Caching bør kun benyttes i forbindelse med iterativ afsøgning.

# Konklusion

Ud fra vore tests kan vi se at AlphaBeta er væsentligt hurtigere end en almindelig MiniMax især når dybden er høj. Vi kan også se at vores optimeringer med at sortere træk virkelig har givet pote. Vores endelige produkt er ikke perfekt – der er stadig mulighed for en del optimeringer. Vi kunne også godt have tænkt os et mere gennemført slutspil, hvor algoritmen mere direkte forsøge at sætte modstanderen mat, men alt i alt mener vi at have en god forståelse for teorien og implementeringen af kunstig intelligens.

Vi mener at vi kunne have lavet programmet mere effektivt hvis vi havde skrevet det i et andet sprog fordi det havde sparet os for en del tekniske problemer, der ikke er direkte relevante for kunstig intelligens. Vi valgte C++ for at få mere erfaring med sproget og vi synes også, at vi har lært meget som programmører, på trods af den negative effekt det har haft for projektet.

Vi kan også se at vi i vores iver efter at optimere har gjort programmet dårligere. Vi besluttede ikke kun at generere tilladte træk, hvilket har resulteret i et mindre stabilt og mere kompliceret program. Det var en dårlig beslutning, som i sidste ende gav et dårligere spil for en meget beskeden optimering.

Bortset fra dette synes vi, at vi har lavet et glimrende skakprogram, som implementerer alle regler og kører ved ganske udmærkede tider på fem ply. Selvom vi har lavet mindre heldige beslutninger undervejs, er vi ikke i tvivl om, at vi har en gennemgående forståelse for faget og er i stand til at benytte denne viden i fremtidige projekter.

Litteraturfortegnelse

<http://en.wikipedia.org/wiki/Zobrist_hashing>

<http://www.chesshub.com/faq/cchess/>

<http://en.wikipedia.org/wiki/Chess>

Compendium for Artificial Intelligence (8. edition) - Bjørn Christensen

Bilag

1. Indhold af CD’en

På den vedlagte CD finder du al kildekoden til vores projekt. Bemærk at hvis du ønsker at compile projektet selv, skal du bruge Boost biblioteket og SDL biblioteket. Vi har i projekt folderen vedlagt ”SDL og Boost setup.txt” som forklarer hvordan disse skal sættes op i Visual Studio.

|  |  |
| --- | --- |
| Mappe | Forklaring |
| MatLab | MatLab data og udregninger. |
| Projekt | Visual Studio projekt. |
| Rapport | Rapporten i pdf og docx format. |
| Release | Kompilet program |
| Source | Kilde kode uden projekt. |

Den konfiguration af programmet som vi har compilet og vedlagt køre 5 ply med iterativ afsøgning og caching. Der genereres kun legale træk. Alle regler er med, inklusiv en-passant, som ikke var med i turneringen. Vær opmærksom på at programmet først opdaterer dit træk på skærmen efter computeren har udregnet sit træk!