Inhoudsopgave

Inleiding

Rond 1200 kwam schaken naar Europa. Het 'Spel der Koningen' fascineerde mensen, op een manier die nooit eerder voor mogelijk was gehouden. Al snel veranderde het in een sport, maar ook daar hield het niet op. 800 jaar later, de mensheid is schaakt nog steeds, maar nu met een vergroot speelveld: de computer.

Bij het nadenken over een onderwerp voor mijn Profielwerkstuk (PWS) wist ik al meteen dat het over wiskunde moest gaan. Abstracte omgevingen, grote berekeningen, schaken en programma's fascineren me altijd. Daarom heb ik ervoor gekozen om een schaakprogramma te bouwen, dat door middel van *Artificial Intelligence* zetten bedenkt voor de computer. Ik wilde antwoord op de volgende vragen:

- Is het mogelijk om in minder dan drie maanden in mijn eentje een schaakprogramma te bouwen?
- Kan ik de computer zelf een zet laten bedenken?
- Is het mogelijk om een zoek-algorime helemaal te doorgronden?
- Kan ik de computer op verschillende niveaus laten spelen?

Dit PWS is een verslag van de zoektocht naar de antwoorden, en de problemen die ik onderweg ben tegengekomen. Het resultaat van het programma dat ik heb gebouwd staat op internet: http://hidde.webatu.com. U kunt het programma (met bijbehorende source code) daar downloaden en zelf uitproberen.

Ten slotte wil ik Hendrik Blauwendraat bedanken voor alle begeleiding tijdens de laatste drie maanden. Van het begin tot het einde stond hij klaar met ideeën en commentaar. Een betere begeleider had ik niet kunnen wensen!

Hidde Wieringa, december 2011

1: De geschiedenis van de schaakcomputer

In 1770 bouwde de diplomaat Wolfgang von Tempelen een machine die kon schaken. Hij verwonderde er mensen mee door heel Europa, en beroemde personen als Napoleon Bonaparte en Benjamin Franklin hebben ertegen geschaakt. Het geheim, dat pas werd ontdekt nadat de machine 85 jaar zijn werk had gedaan, was de menselijke schaakspeler die erin zat.

1.1 Schaakprogramma

Het eerste schaakprogramma is wonderlijk genoeg geschreven toen er nog geen computers bestonden. Alan Turing (1912 – 1954)[1], een wereldberoemd wiskundige, schreef instructies voor een machine om te schaken. Omdat zo'n machine nog niet bestond speelde hij zelf voor computer. Het duurde wel meer dan een half uur om een zet te bedenken.

In diezelfde periode bedacht Claude Shannon dat hij een computer zou kunnen leren schaken. Shannon bedacht de *A-Strategy* en de *B-Strategy*. Hij bedoelde met de *A-Strategy* de *brute-force* techniek: alle zetten uitproberen, en daaruit de beste kiezen. De *B-Strategy* was slimmer, omdat deze een aantal takken van alle mogelijke posities niet analyseerde, waardoor de analyse veel sneller werd.

In 1946 werd in Amerika een enorme computer (de *MANIAC I*)[2] gebouwd, die meer dan 10.000 instructies per seconde kon ontvangen en geprogrammeerd kon worden. Deze machine werd oorspronkelijk gebouwd om de correcte vorm van de implosie te berekenen die nodig was om de kettingreactie van de atoombom op gang te krijgen. Toen de computer af was, ging het team

[1] Alan Turing: http://en.wikipedia.org/wiki/Alan_Turing
[2] MANIAC I: http://chessprogramming.wikispaces.com/MANIAC+I

van wetenschappers niet aan de slag met de bom, maar programmeerde het eerst een schaakprogramma. Het ging hier niet om een bord van 8x8 maar om een bord van 6x6 zonder lopers, zodat de berekeningen minder groot waren. Met deze computer duurde het ongeveer twaalf minuten om vier zetten vooruit te denken. Met een 8x8 bord zou dit minimaal drie uur duren.
De *MANIAC I* heeft drie keer geschaakt: de eerste keer tegen zichzelf (wit won), de tweede keer tegen een grootmeester (de meester won) en de derde keer tegen een vrouw die twee weken eerder schaken had geleerd. Ze werd in 23 zetten mat gezet. Dit was de eerste keer dat een computer een mens in een intellectueel spel versloeg!

Omdat wetenschappers wilden weten of het mogelijk was om een schaakcomputer te laten winnen van een grootmeester. Ze zijn gaan rekenen aan de brute-force techniek. Als er per positie gemiddeld 30 mogelijke zetten zijn, en een potje gemiddeld 40 zetten duurt (van beide partijen), zijn er dus $(30^2)^{40} = 14780882941434592331608321020638329760100000000000000000000000000000000000000000000000000000000000000000000000000000000 \approx 1{,}5 * 10^{118}$ verschillende posities zijn voor een potje schaken. Aangezien er ongeveer 10^{80} atomen in het universum zijn[3] kan een computer nooit alle mogelijke zetten berekenen.
Computers uit het begin van de 20e eeuw konden ongeveer 500 posities per seconde berekenen, dus ongeveer 90.000 posities in de drie minuten per zet. Dat betekent dat er $\log_{30}(90000) \approx 3{,}35$ posities vooruit kan worden gedacht, dat is ongeveer anderhalve zet. Een beginnende schaker kan daarvan winnen.
Als de computer vier posities vooruit zou moeten kunnen denken, zou dat ${30^4}/_{180} = 4500$ posities per seconde betekenen, 9 keer zoveel. Nóg een positie vooruit zou nogmaals 30 keer meer rekenkracht kosten. Het doel om een grootmeester te verslaan leek in de 50-er jaren onhaalbaar.

1.2 'Alpha-Beta'

In 1958 kwam er een grote doorbraak in de zoektocht naar een goed schaakprogramma. Drie wetenschappers van de universiteit in Pittsburgh bedachten het 'Alpha-Beta' algoritme. Hiermee kon een groot aantal takken van de analyse van de mogelijke posities weggestreept te worden, wat de analyse erg versnelt. Opeens konden schaakprogramma's drie zetten (zes of zeven posities) vooruit denken. Toch was er nog steeds een exponentiële toename aan rekenkracht nodig om een positie verder te kunnen analyseren.

In 1968 sloot David Levy, een Engelse grootmeester een weddenschap af[4]. Hij stelde dat hij de komende tien jaar niet verslagen zou worden door een schaakcomputer. Hij won de weddenschap door in 1978 Chess 4.7 te verslaan[5]. Acht jaar later, werden de schaakcomputers *ChipTest* en *Deep Thought* gebouwd. *Deep thought* kon 500.000 posities per seconde berekenen, en versloeg David Levy. Toch haalde deze computer nog niet het niveau van een wereldkampioen.

De makers van *Deep Thought* gingen werken bij IBM en maakten daar de computer *Deep Blue*. Deze bevatte chips die speciaal waren gemaakt voor de schaakcomputer, en tussen de twee en de drie miljoen posities per seconde konden berekenen. Door tweehonderd van deze chips te gebruiken, was het mogelijk ongeveer 200 miljoen posities per seconde te berekenen. *Deep Blue* versloeg in 1996 één keer Gary Kasparov, de toenmalige wereldkampioen schaken. Kasparov

---

[3] Atomen in het universum: http://www.wolframalpha.com/input/?i=atoms+in+the+universe
[4] De Levy Weddenschap: http://chessprogramming.wikispaces.com/David+Levy#The Gambler-The Levy Bet
[5] Chess 4.7: http://www.geocities.com/SiliconValley/Lab/7378/levy.htm

won daarna wel drie keer en speelde één keer gelijk tegen *Deep Blue*, maar het was wel de eerste keer ooit dat een wereldkampioen schaken werd verslagen door een computer.

Hoewel de wereldkampioen een keer verslagen was, was *Deep Blue* nog niet onoverwinnelijk. Ken Thompson zocht in de jaren '80 uit hoe diep een computer moest zoeken om op het niveau van een wereldkampioenschap te komen.[6] Meestal wordt dit berekend in *Elo punten*, een internationale *rating*[7] om schaakniveaus te kunnen vergelijken. Als Thompson zijn schaakcomputer tegen zichzelf liet spelen met één positie minder ver zoeken verschil, kwam dat uit op ongeveer 200 *Elo punten* per positie. Dat betekende dat een computer op wereldkampioen niveau (2800 *Elo punten)* veertien posities (of zeven zetten) diep zou moeten kunnen zoeken. Dat komt neer op een zoeksnelheid van 1 miljard posities per seconde.

Door de jaren heen zijn wetenschappers steeds bezig gebleven met het ontwerpen van schaakcomputers. Het blijft een goede uitdaging, en vormt een goede maatstaf voor het verschil in rekenkracht tussen die van een mens en die van een computer!

[6] Ken Thompson: http://www.chessbase.com/newsdetail.asp?newsid=6970
[7] Elo punten systeem: http://en.wikipedia.org/wiki/Elo_rating_system

2: Het schaakprogramma

Een schaakprogramma bestaat grofweg uit drie onderdelen:

- het hoofddeel van het programma, dat de andere twee delen aanstuurt, de berekende zetten uitvoert en de input van de gebruiker verwerkt.
- de *Graphical User Interface* (de *GUI*) regelt de in- en output van het schaakprogramma: het scherm met het schaakbord en alle stukken erop, de plek in het scherm waar de gebruiker klikt en alle tekst die in de statusbalk verschijnt.
- de Engine, die de beste zet voor de computer berekent, en die zet doorgeeft aan het programma. De Engine staat beschreven in hoofdstuk 3, aangezien dat het belangrijkste (en moeilijkste) deel is van een schaakcomputer.

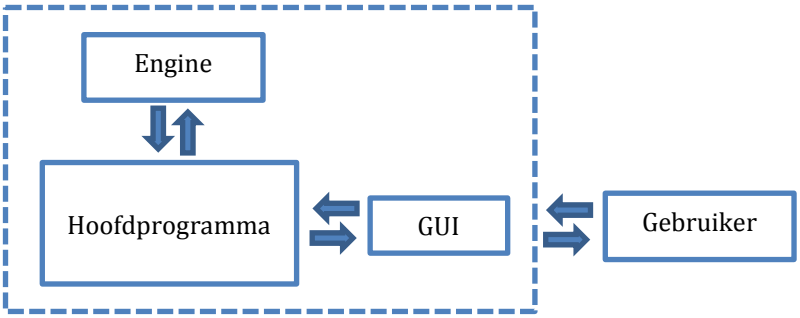
De schaakcomputer die ik zelf, volgens deze indeling, heb gemaakt heet Onyx. Tijdens het uitleggen van de onderdelen van een schaakprogramma zal ik vaak kort toelichten hoe ik dat onderdeel heb gemaakt binnen Onyx. De uitleg over alle problemen waar ik tegenaan ben gelopen, en alle onderdelen van Onyx die niet in dit hoofdstuk behandel komen aan bod in hoofdstuk 4.

2.1 De GUI

De *Graphical User Interface* van een schaakcomputer laat zien aan de gebruiker wat er gebeurt in het programma. Ook stelt de GUI de gebruiker in staat om het programma instructies te geven. De GUI van Onyx bestaat uit twee hoofdonderdelen. Allereerst is er het optieschermpje waar de gebruiker de eerste keuzes kan maken. Daarnaast is er het hoofdscherm dat zichtbaar is

zolang het programma loopt, met daarop het schaakbord, twee informatiepanelen en een aantal knoppen.

In het optieschermpje worden de beginopties geselecteerd en doorgegeven aan het hoofdprogramma. De beginopties zijn: de speelkleur, de bordgrootte en de moeilijkheidsgraad. Het programma start pas op als het optieschermpje wordt weg geklikt. Dan wordt het schaakbord met alle stukken erop getekend. Nu kan de GUI wachten tot er een klik wordt gegeven op het bord, of totdat de beste zet voor de computer is berekend als de speler heeft aangegeven met zwart te willen spelen.



Figuur 1: Een schaakprogramma

De GUI van Onyx heeft een menubalk en een statusbalk, twee kleine hulpmiddelen die helpen om een programma toegankelijk te maken, en die niet in de weg te zitten. Vanuit de menubalk kun je een nieuw spel maken, het huidige spel opslaan of laden en natuurlijk het programma afsluiten. Ook kun je met de helpknop een internet pagina openen, waarop de helponderwerpen van Onyx verschijnen.

Het spel opslaan werkt met tekstbestanden (.onyx), die door middel van letters en cijfers met alle data van het spel wordt gevuld. Als er op 'Spel laden' wordt gedrukt, kan zo'n bestand weer worden ingelezen, om weer verder te gaan met het vorige spel.

2.2 Het hoofdprogramma

Het hoofdprogramma is het belangrijkste onderdeel van een schaakcomputer, want dit deel regelt alles, van de in- en output tot de berekening van een zet. Het hoofdprogramma werkt niet alleen als medium voor informatie, het verwerkt ook een deel van de informatie, en stuurt zo andere onderdelen van het geheel aan.

Zodra de gebruiker op een stuk klikt dat hij wil verplaatsen, rekent het hoofdprogramma alle mogelijke zetten voor dat stuk uit. Dat doet hij door alle mogelijke plekken langs te gaan waar het stuk vanuit zijn positie naar toe kan en dan te kijken of dit een geldige zou zijn. Het resultaat geeft het hoofdprogramma weer door aan de GUI, die in op het schaakbord laat zien waar het aangeklikte stuk naartoe kan.

Zodra de gebruiker klikt op het veld waarnaar hij het stuk wil verplaatsen geeft de computer opdracht aan het geselecteerde stuk om zich te verplaatsen en vraagt de GUI om het stuk op zijn nieuwe plek neer te zetten. Zodra het stuk verplaatst is start de Engine. Deze krijgt de opdracht om een goede zet uit te rekenen voor de computer (zie paragraaf 2.3: De Engine). Als er een zet is uitgerekend wordt deze geanalyseerd: er wordt door het hoofdprogramma gekeken welk stuk er moet worden verplaatst, waar naartoe en hoelang de berekeningen hebben geduurd. De benodigde tijd is handig om te weten, om zo de prestaties van de Engine te kunnen vergelijken, bij een aanpassing van de code.

Ondertussen wordt aan de GUI verteld dat het stuk zich heeft verplaatst en hoelang daarover gerekend is. De GUI zal de zet als tekst weergeven en een bericht plaatsen met alle informatie van de berekeningen van de computer.

2.3: De Engine

De engine is het belangrijkste deel van een schaakprogramma: de computer kan er goede zetten mee berekenen, de Engine neemt het meeste rekentijd en processorkracht van de hele schaakcomputer en de Engine beslaat veruit de meeste regels code.

De Engine is opgebouwd uit meerdere delen:

- de start-functie
- het 'Alpha-Beta' algoritme met alle uitbreidingen
- alle ondersteunende functies

2.3.1 De start-functie

Wanneer de Engine wordt aangeroepen wordt de startfunctie als eerste opgestart. De start-functie krijgt alle data van het hoofdprogramma binnen die de computer nodig heeft om een goede zet te bedenken. Die data bestaat onder andere uit het schaakbord met alle stukken, zoals dat er op dit moment uitziet, de speler aan zet en of de computer met wit speelt. Vanaf hier worden alle benodigde functies aangeroepen, zoals de Alpha-Beta functie en de functie die controleert of een zet geldig is. De Alpha-Beta functie bevat het algoritme dat op basis van alle mogelijke zetten de beste bepaalt.

2.3.2 De Alpha-Beta functie

In de Alpha-Beta functie wordt de beste zet gezocht, uit alle mogelijke zetten. De functie heeft een bijzondere eigenschap, namelijk het aanroepen van zichzelf. Het is daarom erg moeilijk om hem te begrijpen (zelfs als je je erin verdiept). De functie is gelimiteerd door de maximale zoekdiepte, die de hij ontvangen heeft van de start-functie van de Engine. De zoekdiepte is het aantal posities dat de functie vooruit moet kijken, in de zoektocht naar de beste zet. De limitering is er om ervoor te zorgen dat de functie niet oneindig keer zichzelf blijft aanroepen, en voor altijd doorgaat.

De Alpha-Beta functie doet de volgende dingen in de genoemde volgorde (in hoofdstuk 3 wordt het Alpha-Beta algoritme uitgelegd):

1. Er wordt gekeken of de maximale zoekdiepte is bereikt.
2. Alle mogelijke zetten voor deze positie voor de speler aan zet worden berekend.
3. Eén voor één worden alle mogelijke zetten gemaakt
    a. Alle aanvallende zetten worden geanalyseerd.
    b. Alle verdedigende/normale zetten worden geanalyseerd.
4. Alle zetten worden weer ongedaan gemaakt.
5. Er wordt gekeken of er zetten zijn geanalyseerd.
6. De beste zet wordt teruggegeven aan de start-functie.

Stap voor stap zullen we de functie uitpluizen. De uitleg is niet volledig, om het begrijpelijk te maken voor iedereen die nog niks van dit onderwerp afweet.

1. Het eerste wat de Alpha-Beta functie doet is een check of de maximale zoekdiepte is bereikt. Als dat het geval is moet de functie niet de positie nog dieper analyseren, maar de waarde van de positie zoals deze nu staat teruggeven. Deze wordt gemaakt door de evaluatie functie (paragraaf 2.3.3. (In de realiteit gebeurt die niet direct: eerst wordt de Quiescence Search (uitgelegd in paragraaf 3.6) aangeroepen, die zal zorgen voor de

uiteindelijke waarde van de positie.)
De reden dat er een maximale zoekdiepte op de Alpha-Beta functie zit, is het enorme aantal mogelijke posities. Vanaf het begin van het spel kunnen nooit alle posities worden onderzocht, omdat dat miljoenen jaren zou duren. Met een maximale diepte wordt de functie beperkt, en zal de zoektocht korter duren. Het is wel zo, dat het resultaat van het de Alpha-Beta functie beter wordt naarmate dieper wordt gezocht, omdat meer posities worden geanalyseerd.

2. Alle mogelijke zetten voor de speler aan zet worden berekend. Voor elk stuk (dat aan zet is) wordt gekeken waar het zich naar toe kan bewegen en of dat mag volgens de spelregels. Dit houdt in dat rekening wordt gehouden met rokeren, en-passant, promoveren van pionnen en een positie waarin de speler zichzelf schaak zet. Eerst worden alle zetten gesorteerd op waarde. Deze waarde hangt ervan af of de zet de andere koning schaak of mat zet, of er gerokeerd wordt, of de zet een stuk slaat met een hoge waarde en of de positie van het stuk op het bord goed is. Ten slotte worden alle zetten opgeslagen, zodat de Engine ze later weer kan gebruiken.
3. Eén voor één worden alle zetten gemaakt. Dit houdt in dat het stuk (tijdelijk) verplaatst wordt, en er dus steeds weer een nieuwe positie ontstaat. Binnen deze 'loop' wordt de nieuwe positie geanalyseerd.
   Alle aanvallende zetten, waarbij een ander stuk wordt geslagen of de andere koning wordt schaak gezet worden als eerste geanalyseerd. Hierbij wordt simpelweg de Alpha-Beta functie opnieuw aangeroepen met de huidige zoekdiepte - 1. De zoekdiepte wordt dus steeds kleiner naarmate dieper gezocht wordt. Als de zoekdiepte 0 is, dan stopt de Alpha-Beta functie, volgens de check in stap 1.
   Op dezelfde manier worden alle verdedigende/normale zetten geanalyseerd. De normale zetten worden na de aanvallende zetten gesorteerd, omdat de aanvallende zetten vaker een beter resultaat opleveren dan de verdedigende zetten. Door de (verwachte) betere zetten als eerste te sorteren grijpt Alpha-Beta Pruning (paragraaf 3.2.1) eerder in en doet beter zijn werk (kortere analyse tijd met dezelfde resultaten). Van elke positie komt een waarde terug (de waarde die de Alpha-Beta functie die net is aangeroepen en zijn werk heeft gedaan heeft teruggegeven). Als de waarde groter is dan de tot nu toe grootste waarde, wordt de gevonden waarde de beste zet.
4. Alle zetten worden nu weer ongedaan gemaakt, zodat de volgende zet onderzocht kan worden.
5. Tijdens de analyse in de Alpha-Beta functie wordt in een variabele bijgehouden hoeveel zetten er geanalyseerd zijn. Als er in totaal nul zetten zijn geanalyseerd (omdat er nul mogelijke zetten waren) kan dat twee dingen betekenen: De speler staat schaakmat, of de speler staat pad. Het enige verschil tussen deze twee gevallen is dat de koning schaak staat of niet. Als de speler mat of pad staat, hoeft er geen beste zet teruggegeven te worden aan het programma: het spel is klaar.
6. Als laatste wordt de beste zet (de zet die de hoogste waarde heeft) teruggegeven aan de start-functie, die hem op zijn beurt weer teruggeeft aan het hoofdprogramma.

2.3.3 De ondersteunende functies

De ondersteunende functies van de Engine bestaan uit losse *methods*, die samen twee taken uitvoeren:

- De generatie van zetten binnen de Engine
- De verplaatsing van de stukken over het bord binnen de Engine
- De evaluatie van een bord

De functies bestaan uit meer dan één *method* om optimale prestaties te behalen. Deze functies werken alleen binnen de Engine, tijdens de zoektocht naar de beste zet: ze zijn geoptimaliseerd voor de snelheid, niet om het gebruikersgemak of de leesbaarheid.

De snelheid van het verplaatsen van stukken over het bord is ook belangrijk, omdat dat miljoenen keren gebeurt tijdens één analyse. Er wordt binnen de Engine met een *array* gewerkt (een tweedimensionale deze keer), om het bord te maken, in plaats van *objects*. Dit levert een hele kleine snelheidsverbetering op, maar bij miljoenen herhalingen is het erg rendabel.

De evaluatie van een positie gebeurt in een aparte functie. Het Alpha-Beta algoritme heeft voor de laatste posities die geanalyseerd worden een waarde nodig. Deze waarde wordt gebruikt voor die positie, waarmee het algoritme de beste zet kan berekenen. Deze waarde wordt bepaald door een aantal verschillende factoren:

- De stukken van de speler aan zet en de stukken van de tegenstander
- Op welke plek staan de stukken van de speler aan zet?
- Staat de koning van de tegenstander schaak?
- Is er een loper- of paardenpaar?

Het makkelijkste onderdeel is het aantal stukken van de speler aan zet en die van de tegenstander. Elk stuk heeft een standaard waarde toegewezen gekregen, ik heb gekozen voor de onderstaande lijst:

- Koningin: 850
- Toren: 525
- Loper: 333
- Paard: 305
- Pion: 100

Hierbij heb ik drie opmerkingen. Ten eerste is er sinds de eerste schaakcomputer discussie over de waardes van stukken, die kunnen verschillen per schaakprogramma. Na het testen van Onyx gaven deze waardes het beste resultaat voor de computer. Ten tweede is het goed om op te merken dat een toren en een loper samen meer waard zijn dan een koningin, en dat alle andere twee stukken samen minder waard zijn. Ook hiervoor heb ik bewust gekozen. Ten slotte is er de koning, die heeft geen waarde, omdat dit stuk nooit geslagen kan worden. Er staan altijd twee koningen op het bord, dus er is geen vergelijking nodig met andere stukken.

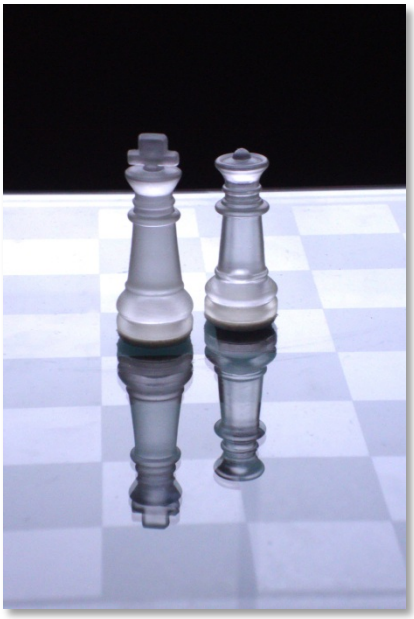
De plek van de stukken op het bord maakt uit voor de kracht van een positie. Elke positie van elke stuk op het bord heeft zijn eigen waarde, en die waarde wordt gebruikt in de berekening. De tabel die al deze waardes bevat heet een *Position*-Table. Als een toren in het midden van het bord staat, kan hij snel geslagen worden, als hij helemaal in de hoek staat heeft hij weinig macht over het bord. Een toren op de zevende rij is meestal erg sterk, en daarom krijgen deze posities een hoge waarde (zie figuur 2).

```
 0,  0,  0,  0,  0,  0,  0,  0
 10, 15, 15, 15, 15, 15, 15, 5
-5,  0,  0,  0,  0,  0,  0, -5
-5,  0,  0,  0,  0,  0,  0, -5
-5,  0,  0,  0,  0,  0,  0, -5
-5,  0,  0,  0,  0,  0,  0, -5
-5,  0,  0,  0,  0,  0,  0, -5
 0,  0,  5,  15, 15, 5,  0,  0
```

Figuur 2: Position-Table waardes van de toren

Als de koning van de tegenstander schaak staat is dat voordelig voor de speler aan zet. De tegenstander kan dan veel minder zetten doen, en de eerste keer schaak kan het begin zijn van het einde van het spel voor de tegenstander. De eigen koning wordt niet meegenomen in de berekening: het resultaat daarvan is al te merken in de mogelijke zetten, die drastisch minder worden.

Als laatste wordt gekeken naar loper- en paardenparen. Deze twee paren verschillen een beetje van elkaar. Loperparen zijn twee lopers die naast elkaar staan, zonder vakjes ertussen. Als twee lopers zo staan, vallen ze gezamenlijk veel vakjes aan, die dicht bij elkaar liggen. In het midden van het bord kan een loperpaar betekenen dat deze speler het bord controleert: de tegenstander kan zijn stukken niet goed plaatsen.
Een paardenpaar zijn twee paarden die elkaar dekken (figuurlijk). Omdat paarden kunnen springen hebben ze geen last van andere stukken. Twee paarden die elkaar dekken maken een groot gebied onbereikbaar voor de tegenstander, en zijn niet makkelijk te slaan. Elk stuk dat een paard slaat heeft meestal een hogere waarde dan het paard, dus het is voordelig voor de speler aan zet om dat stuk dan terug te slaan. Dit maakt het net als bij de lopers moeilijk voor de tegenstander om zijn stukken op een goede plek te krijgen.
Een dergelijke opzet van de twee lopers of paarden wordt beloond met een bonus, zodat de positie een hogere waarde krijgt dan een positie waar geen paren aanwezig zijn.



3: Het Alpha-Beta algoritme

Het Alpha-Beta Algoritme (een voorbeeld van een Min-Max algoritme) is een algoritme dat de optimale situatie berekent bij een totaal transparant spel met twee spelers. Dit betekent dat voor beide spelers alle informatie over het spel op tafel ligt (in tegenstelling tot de meeste kaartspelen) en dat alle mogelijke zetten dus te vinden zijn. Het Alpha-Beta algoritme kan het beste en het makkelijkste gebruikt worden voor *Zero-sum* spellen: een verlies van de ene speler is gelijk aan de winst van de andere speler, en andersom. Voorbeelden van spellen waarbij het algoritme goed werkt, zijn de *Zero-sum* spellen schaken, Boter, Kaas en Eieren, Vier op een Rij en het Japanse spel Go.

In dit hoofdstuk zal ik de lezer laten kennismaken met het Alpha-Beta algoritme, uiteenzetten hoe het werkt, vertellen welke uitbreidingen er zijn en uitleggen waarom ik ervoor heb gekozen.

3.1 Het algoritme

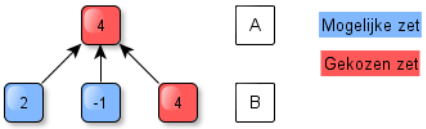
Om dit algoritme te analyseren werken we terug vanuit de doelstellingen van de twee spelers bij een *Zero-sum* spel:

- A wil de optimale situatie voor zichzelf en de slechtste situatie voor B
- B wil de optimale situatie voor zichzelf en de slechtste situatie voor A

Wie dit doel het beste bereikt wint. Als A aan zet is en helemaal niet vooruit kijkt naar de zetten van B, dan ziet zijn keuze-diagram er uit als in figuur 1 (We nemen hier aan dat er steeds drie mogelijke zetten zijn per positie, dit kan in de realiteit minder of meer zijn):

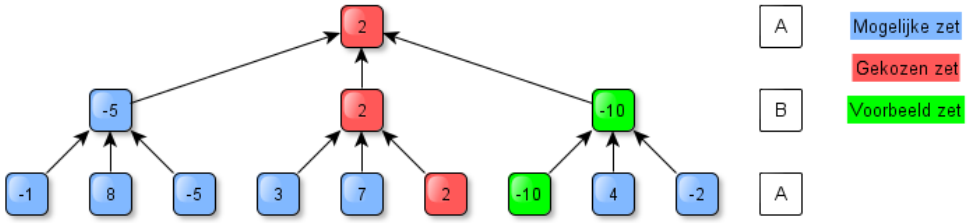
In de diagrammen in dit hoofdstuk stelt het eerste blokje de huidige positie voor. Alle lijnen daaruit zijn mogelijke zetten, die de blokjes eronder opleveren (de nieuwe mogelijke posities). De cijfers in de blokjes staan voor de waarde van de positie, zoals A het ziet. Naast elke rij blokjes staat wie er aan de beurt is tijdens die positie. Deze speler mag kiezen welke van de onderliggende blokjes hij kiest als nieuwe positie (hij mag zelf kiezen welke zet hij doet). Bij de laatste positie van elke tak van zetten is de waarde bekend, die is uitgerekend door de evaluatie functie. Bij de evaluatie van de hele *analyse-boom*, wordt van beneden naar boven gewerkt, omdat er keuzes worden gemaakt op basis van de bekende informatie. Om die reden staan de pijlen naar het bovenliggende blokje gericht.

De notatie in dit hoofdstuk is anders voor de verschillende functies van de tekst. Tekst die afkomstig is uit een functie- of codevoorbeeld, of de naam is van een variabele, wordt genoteerd met het lettertype Courier New, een monospace lettertype dat standaard is bij programmeren. Code kan zo makkelijk herkend worden.



Figuur 1

Voor A is het in figuur 1. niet moeilijk om de beste zet te kiezen. De zet waarbij de positie de waarde van 4 zou krijgen is de beste zet, omdat A daar de optimale situatie behaalt (het hoogste getal). Waar A niet op let is dat B misschien na die zet een significant betere zet kan doen dan A, waardoor A opeens aan de verliezende hand is. In figuur 2 kijkt A één beurt vooruit, en houdt daar rekening met B, die ervoor wil zorgen dat A niet op de optimale situatie uitkomt.



Figuur 2

Wat zien we hier: elke keer als B aan de beurt is kiest hij voor de beste positie voor hem. Dat is voor A de slechtste positie, uitgedrukt in een lage waarde.
De zet die A doet uiteindelijk doet komt uit op een waarde van 2 (figuur 2) in plaats van 4 (figuur 1). Dit is minder gunstig voor A, maar als A de zet had gekozen die mogelijk uit zou komen bij een waarde van 4, dan had B de zet gedaan die uitkomt bij een waarde van -10, wat helemaal niet goed is voor A (in figuur 2 met groen aangegeven).

Omdat het Alpha-Beta algoritme makkelijk te implementeren is, wordt het voor veel spellen vaak gebruikt. Per spel worden kleine aanpassingen gemaakt om ervoor te zorgen dat het algoritme optimaal werkt en de berekeningen zo snel mogelijk maakt. Ik heb het Alpha-Beta algoritme gekozen omdat er veel mogelijkheden zijn om mijn schaakprogramma te optimaliseren. Bovendien is er veel informatie over bekend, wat mij kan helpen om zelf een schaakprogramma te bouwen.

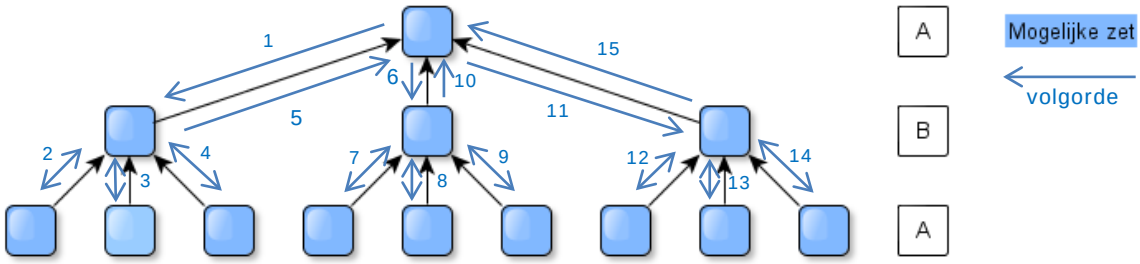
3.2 Optimalisatie

Voor schaken geldt dat er gemiddeld 30 mogelijke zetten zijn per positie, en dat betekent $30^6 = 729$ miljoen te analyseren posities bij al drie zetten vooruit kijken. Om de gemiddelde computer dit te laten berekenen binnen een aantal seconden moeten er optimalisaties worden toegepast. Hiermee wordt de zoektocht naar de beste zet versneld. Die optimalisaties zijn onder andere Alpha-Beta Pruning, Aspiration Search, en Quiescence Search. Deze begrippen leg ik in de komende paragrafen uit.

3.2.1 Alpha-Beta Pruning

Het Engelse woord 'pruning' betekent snoeien. Dat is precies wat Alpha-Beta Pruning met het algoritme doet: de boom van posities sterk bijsnoeien. Als er van bepaalde zetten een waarde bekend is, kan al van te voren, vóór het analyseren van de positie worden vastgesteld dat deze positie nooit een betere zet kan opleveren dan de huidige beste zet. In figuur 4 wordt Alpha-Beta Pruning uitgelegd aan de hand van pseudo-code. In de praktijk is de analyse van een zet veel gecompliceerder. In het voorbeeld maak ik gebruik van een versimpelde versie om het proces begrijpelijker te maken.

Als het commando wordt gegeven om de beste zet te vinden voor A (volgens de code van figuur 3: `alphaBeta(2, -∞, ∞, A)`[8] maakt het programma twee variabelen aan: α en β met als begin waardes –∞ en ∞. α en β zijn twee waarden die de marges, de boven- en ondergrens, zijn van de analyse zijn. Deze twee waardes worden gebruikt om te bepalen of een zet de moeite waard is om te analyseren. β zal worden geminimaliseerd, α zal worden gemaximaliseerd. In de diagrammen van deze paragraaf zal met twee getallen naast het blokje van de positie blokje de α en β van die positie aangegeven worden. Een verschil met de voorgaande schema's, is dat nu vanuit de eerste positie naar beneden wordt gewerkt, en vanuit de *leafs* van de analyse weer terug (figuur 3).



Figuur 3

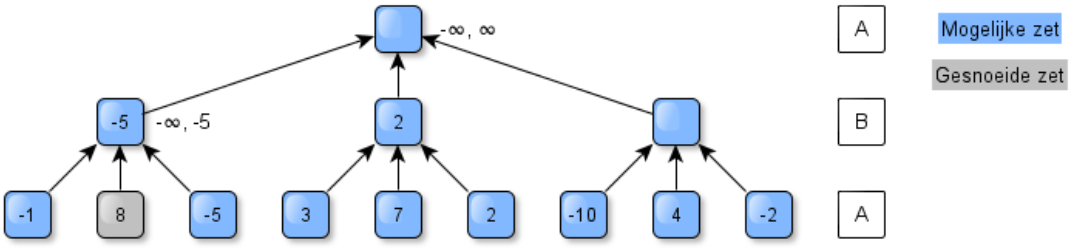
[8] Alpha-Beta Puning pseudo-code: http://en.wikipedia.org/wiki/Alpha-beta_pruning

```
function alphaBeta(depth, α, β, Player)
      if depth == 0
            return value_of_position
      if Player == A
            for each possible_move
                  α = max(α, alphaBeta(depth-1, α, β, B ))
                  if β ≤ α
                        break
            return α
      else
            for each possible_move
                  β = min(β, alphaBeta(depth-1, α, β, A ))
                  if β ≤ α
                        break
            return β
```

Figuur 4: het Alpha-Beta algoritme in pseudo-code

Bij de eerste analyse van een positie voor B krijgt de β van die positie een nieuwe waarde als er een lagere waarde wordt gevonden dan de huidige β (figuur 5). Eerst had β een waarde van ∞, die nu -1 wordt. De waarde van de volgende positie is 8. Deze wordt meteen al genegeerd omdat 8 hoger is dan -1. Als een positie genegeerd wordt, dan wordt deze niet geanalyseerd, en doet het programma dus minder berekeningen. Hierdoor wordt de analyse sneller. De waarde van de derde positie van B na de eerste mogelijke zet van A wordt niet genegeerd. -5 is lager dan β, dus β wordt -5 en -5 wordt teruggegeven aan de bovenliggende zet.

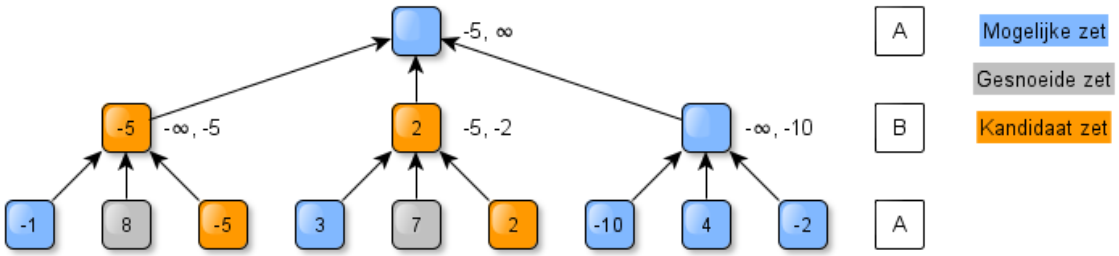
Bij de analyse van de eerste mogelijke zet van A krijgt A -5 teruggegeven als kandidaat-zet, wat ervoor zorgt dat α een nieuwe waarde krijgt van -5, omdat -5 groter is dan -∞.



Figuur 5

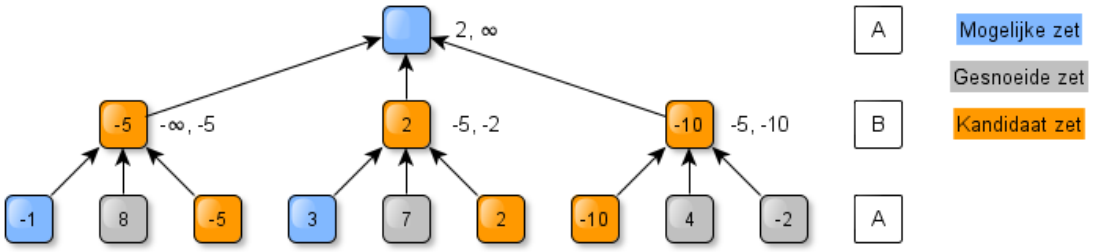
De volgende positie krijgt nu de α en β mee van respectievelijk -5 en ∞ (dit zijn de α en β van de bovenliggende zet). De waarde van 3 wordt geanalyseerd (β wordt 3 want 3 is lager dan ∞), de waarde van 7 genegeerd (7 is hoger dan β), en de waarde van 2 wordt geanalyseerd (β wordt 2 want 2 is lager dan 3). 2 wordt teruggegeven naar boven als kandidaat-zet (Figuur 6).

Na de analyse van de tweede mogelijke zet van A worden de α en β van A respectievelijk 2 en ∞.



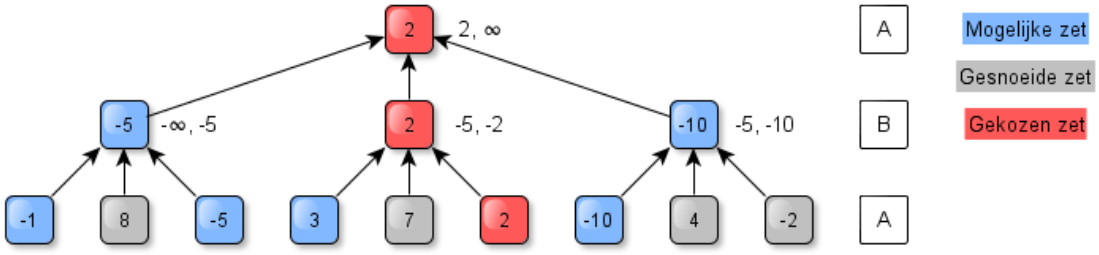
Figuur 6

Bij de laatste tak van de analyse zijn de α en β van B 2 en ∞, beide doorgegeven van A. Bij deze tak gebeurt nu precies datgene waardoor Alpha-Beta Pruning zo goed optimaliseert: De waarde van -10 wordt geanalyseerd en is lager dan de huidige β. β wordt daarom -10, terwijl α 2 is. Omdat β altijd hoger moet zijn dan α wordt -10 gelijk teruggegeven aan A, zonder naar een andere zet te kijken (zie Figuur 7). Als de β van een te analyseren mogelijke zet hoger is dan de α, kan deze zet nooit meer als beste zet uit de analyse komen.



Figuur 7

Als laatste analyseert A zijn eigen tak met de laatste drie waardes die over zijn (Figuur 8). A selecteert de hoogste waarde (de beste zet) die in dit geval 2 is. Deze zet is de beste zet voor A als er één beurt vooruit wordt gekeken.



Figuur 8

3.2.2 Het resultaat van Alpha-Beta Pruning

In de praktijk zijn er duizenden takken van de analyse van de beste zet, met daaraan duizenden te analyseren posities. Als er helemaal geen Pruning wordt toegepast zijn er $b * b * b * b * b \ldots = b^d$ te analyseren posities, waarbij b het aantal mogelijke zetten is, en d de maximale diepte tot waar we zoeken. In de optimale situatie waarbij Pruning zijn werk het beste doet, geldt dat er

$b * 1 * b * 1 * b * 1 \ldots = b^{d/2} = \sqrt{b^d}$ te analyseren zetten zijn. Dit betekent dat er bij optimale werking twee keer zo diep gezocht kan worden. Tijdens een echte analyse ligt de optimalisatie ergens hier tussen, dit hangt af van de volgorde waarin de zetten geanalyseerd worden.

3.3 De Hash-Table

Tijdens het onderzoeken van alle zetten wordt een Hash-Table bijgehouden, van de waardes van alle zetten. Een hash tabel is een niet volledig gevulde tabel met lange getallen, waaruit een waarde af te leiden is, en de positie van die waarde in te tabel.

3.3.1 De werking van een Hash-Table

Bij elke analyse van een positie wordt een de positie vertaald in een binair getal. Een binair getal is een getal zoals iedereen die kent, maar opgeschreven in een tweetallig stelsel (alleen 1'en en 0'en), in plaats van een tientallig stelsel, waarin wij leven. De vertaling van de positie naar het binaire getal gebeurt door de hash-functie. Deze functie zorgt dat alle elementen die een positie uniek maken een bit beheren in de hash-waarde. Een bit is een `1` of een `0` in een binair getal. Een voorbeeld van een simpele hash functie voor Boter, Kaas en Eieren is figuur 9.

```
function hash ()
      for each square_of_the_board
            hash_val = hash_val | (square_value * pow(2, square_index*2))
      return hash_val
```

Figuur 9: Een hash functie voor Boter, Kaas en Eieren

	O	X
O	O	X
X	X	

	O	X	O	O	X	X	X	
00	01	10	01	01	10	10	10	00

Figuur 10: De hash van een Boter, Kaas en Eieren positie

Aan de hand van figuur 9 zal ik uitleggen hoe de hash functie werkt, en hoe als resulataat de hash waarde van een positie tot stand komt. Voor elk vakje van het bord wordt de waarde van dat vakje (`square_value` (`00` voor leeg, `01` voor een rondje en `10` voor een kruisje)) berekend. Die waarde wordt gemultipliceerd met 2 tot de macht van de van de positie van het vakje (`square_index`) op het bord keer 2. Het resultaat wordt geORd (symbool `|`) met de huidige waarde van `hash_val`. Met de OR operatie kan een waarde aan een getal worden toegevoegd (figuur 11).

De hash waarde van figuur 10 is `0010101001011001000`. Elke twee bits stellen een waarde van een vakje voor, van achteren naar voren. Het laatste vakje (`00`) is leeg, het op een na laatste vakje (`10`) bevat een kruisje. Daarna komen nog twee kruisjes, een rondje, een kruisje, een rondje, een kruisje, een rondje, en het eerste vakje is weer leeg. In de praktijk zal hier nog een waarde aan toegevoegd worden, die de positie krijgt.

```
011001110
100010110 |
111011110
```

Figuur 11: De OR operatie

Met deze hash waarde gebeuren twee dingen, ten eerste wordt de waarde van de positie toegevoegd, en ten tweede wordt de hash waarde opgeslagen in een op de plek van de hash (de

rij van enen en nullen vormt een getal). De waarde van de positie hangt er dus aan als extra data, en heeft niks met de opslagplek te maken.

Als het programma later een positie tegenkomt, waarvan op de plek van de hash van de te analyseren positie in de tabel al een getal staat, kan van dat getal dat in de tabel staat de waarde gehaald worden van de (blijkbaar al eerder geanalyseerde) positie, zonder dat deze opnieuw geanalyseerd hoeft te worden.

In een schaakprogramma gaat dit op bijna precies dezelfde manier. Het enige verschil is dat er in plaats van de waardes van een vakje een groot getal gebruikt wordt (een uniek getal van 64 bits, voor elke combinatie van een stuk op een vakje), die geXORd in plaats van geORd wordt met de huidige hash waarde. De XOR operatie (symbool ^) geeft een `1` als resultaat dan en slechts dan als er één enkele `1` in de twee bits die geXORd worden aanwezig is. Bij dat enorme getal wordt de data (in dit geval de waarde van de positie) opgeslagen. Later kan bij de analyse van dezelfde positie de al geanalyseerde waarde opnieuw gebruikt worden.

```
011001110
100010110 ^
111011000
```

Figuur 12: De XOR operatie

3.3.1 Problemen van de Hash Table

Het enige probleem is dat bij zo'n groot getal niet zomaar de hash waarde kan worden gebruikt als positie in de hash tabel. Het getal wordt dus gemoduleerd, met een zo groot mogelijk priemgetal zodat de tabel net niet te veel ruimte verbruikt. Dit kan voor conflicten zorgen, want twee unieke hash waardes kunnen nu wel de zelfde opslagpositie in de hash tabel geven (bijvoorbeeld $15\ mod\ 7 = 1, 22\ mod\ 7 = 1$).

Omdat er priemgetallen worden gebruikt om te moduleren komen conflicten niet vaak voor. Toch worden er soms hash waardes vervangen in de tabel, omdat de hash waarde in de tabel een andere positie bevat dan de positie die nu geanalyseerd wordt. Het is dus voordelig om een zo groot mogelijke hash tabel te maken, om te veel conflicten te voorkomen.

3.3.2 De optimalisatie van de Hash Table

De hash table (er is geen goede Nederlandse vertaling) optimaliseert Alpha-Beta Pruning wanneer een positie al eerder is geanalyseerd. Als dat zo is dan geeft de hash table op de plek van de hash waarde van de te analyseren positie een getal weer. Als er een getal staat, betekent dit deze positie al eerder is geanalyseerd, en dus dat de waarde van die positie gebruikt kan worden voor de analyse van nu. Dit kan heel veel analyse schelen als er een eerder geanalyseerde positie gevonden wordt aan het begin van een tak van de analyse. Het kan echter ook nutteloos zijn, als er een al bekende hash waarde wordt gevonden aan het eind van een analyse tak (een zogenaamde *leaf*, net als het blaadje aan het eind van de tak bij een boom).

3.4 Move Ordering

Om Alpha-Beta Pruning op zijn beurt te optimaliseren wordt er Move Ordering toegepast. Move ordering is niets anders dan de te analyseren zetten zo sorteren dat de 'waarschijnlijk beste' zetten als eerste worden geanalyseerd. Dit heeft als resultaat dat de marges (de α en β) van de analyse veel kleiner worden en er dus eerder zetten zullen worden gesnoeid.

Dit werkt als volgt. Als eerste worden de beste zetten geanalyseerd, in de volgende volgorde[9] (dit kan anders zijn per schaakprogramma):

1. De Hash Table zet
2. Mate-killer zetten

[9] Move Ordering - http://www.top-5000.nl/authors/rebel/chess840.htm

3. Winning-capture zetten
4. Good-capture zetten
5. Killer zetten
6. Rokeren en Promoveren

In de volgende paragrafen wordt uitgelegd wat deze zetten inhouden, en waarom het goed is wanneer deze zetten zo snel mogelijk worden geanalyseerd.

3.4.1 De Hash-Table zet

De Hash-Table zet is eigenlijk geen zet, maar de tak van analyse waaruit de vorige beste zet is gekozen. Er is een hele grote kans dat deze zet zeer goed uit de analyse komt, omdat de enige verandering ten opzichte van de vorige analyse de toegenomen zoekdiepte met twee is. Dit komt omdat de vorige zet nu gedaan is en al vast staat. Er wordt met dezelfde diepte gezocht, dus totaal komt de analyse twee zetten dieper. Dit kan leiden tot een verandering van de waarde van de analyse, waardoor de Hash Table zet misschien niet de beste zet is. In dat geval is de zoek marge nog zo groot, dat de volgende zetten voldoende ruimte krijgen om gekozen te worden.

3.4.2 Mate-Killer zetten

De *Mate-Killer* (*MK*) zetten zijn normale *Killer* zetten, maar ze zetten de andere speler schaak. Omdat dit het einde van een spel kan betekenen, en dus een waarde van ∞, worden de *MK* zetten als eerste onderzocht, na de *Hash-Table* zet. Zie *Killer* zetten voor meer informatie over deze zetten.

3.4.3 Winning-Capture zetten

Als een zet materiaal wint is het een *Winning-Capture* (*WC*) zet. Hiermee slaat deze zet, of een serie van zetten betere stukken van de andere speler, dan de andere speler van eigen stukken slaat. Je krijgt een voorsprong in het spel: na deze zetten is de balans van de totale waarde van je eigen stukken tegenover de totale waarde van de stukken van de andere speler is na deze zetten sterk verbeterd. Elke mogelijke zet na deze zetten krijgt dus een hoge score, en geeft een goede kans om als beste zet gekozen te worden.

3.4.4 De Good-Capture zetten

De Good-Capture (*GC*) zetten zijn zetten die een stuk slaan van precies dezelfde waarde. Dit kan een goede zet zijn, als de positie van je eigen stukken beter is dan de positie van de stukken van de tegenstander. De analyse van deze zetten, die niet vaak voorkomen, wordt dus als vierde gedaan. Binnen de *GC* zetten geldt nog een volgorde, om de betere *GC* van de minder goede te onderscheiden:

- Dame voor dame
- Toren voor toren
- Paard voor loper
- Loper voor loper
- Loper voor paard
- Pion voor pion

3.4.5 De Killer zetten

Vaak kan de tegenstander een goede zet doen die hem een voorsprong geeft. Er moet dus worden voorkomen dat de tegenstander die zet doet. Zetten die een goede zet van de tegenstander voorkomen noemen we Killer zetten. Ze zijn in het algemeen goed en zorgen ervoor dat de tegenstander niet voor komt te staan. Daarom leveren ze een hoge waarde op, en worden ze vaak gekozen als beste zet.

Meestal worden er twee *Killer* zetten bijgehouden. Het is een belangrijke zaak om ervoor te zorgen dat er geen van de zetten die in boven- of onderstaande paragrafen beschreven staan terecht komen bij de *Killer* zetten. Als dat het geval is wordt een andere zet uitgesloten om snel te worden geanalyseerd, en verslechterd de optimalisatie van Move Ordering.

3.4.6 Rokeren en promoveren

De twee speciale soorten zetten, rokeren en promoveren, krijgen ook een plekje bovenaan de lijst van te analyseren zetten. Rokeren zorgt in het algemeen voor een goede positie van de toren in kwestie en de koning: de koning komt goed verdedigd achter de pionnen komt te staan en de toren krijgt een groter bereik, soms zelfs een hele vrije kolom.

Promoveren kan een winnende situatie geven, omdat een simpele pion niet alleen tot 8 keer in waarde stijgt, maar ook nog een positie kan geven waarin de tegenstander in een aantal zetten mat staat.

Het maakt bij rokeren en promoveren niet uit in welke volgorde de analyse plaatsvindt: Rokeren gebeurt bijna altijd aan het begin van het spel of niet (de koning en de toren in kwestie mogen allebei nog niet hebben bewogen). Promoveren kan alleen aan het einde van het spel, wanneer er nog maar weinig stukken zijn die de pion kunnen slaan; natuurlijk moet de pion zelf helemaal naar de overkant van het bord. Mochten op een bepaald punt in het spel beide zetten mogelijk zijn, dan worden beide geanalyseerd en wordt de beste waarde gekozen.

3.5 Aspiration Search

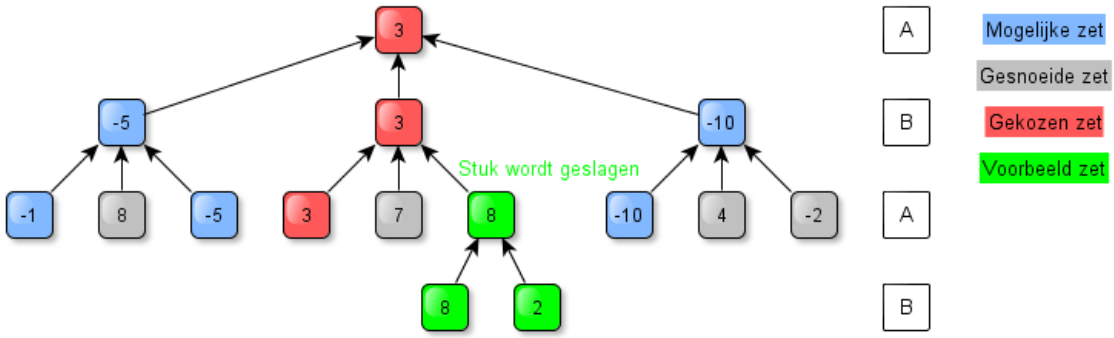
Aspiration Search is geen optimalisatie maar een uitbreiding op Alpha-Beta Pruning. Omdat in de eerste takken van de analyse αeen waarde heeft van -∞ en een β een waarde heeft van ∞ worden in die tak alle zetten onderzocht. Om dat te voorkomen, en dus rekenkracht te winnen kan er een schatting worden gedaan van de mogelijke waarde van de uiteindelijke zet. Bij deze schatting wordt bij het begin van de analyse een 'window' toegevoegd: een marge die ervoor zorgt dat de Aspiration Search niet alle zetten afsnoeit. Als de Alpha-Beta functie nu wordt aangeroepen zal het eruitzien als `alphaBeta(depth, schatting-marge, schatting+marge, A)` in plaats van `alphaBeta(depth, -∞, ∞, A).`

Omdat er marges worden gesteld, naast α en β, kan een Aspiration Search erg negatief zijn voor de zoektocht naar de beste zet. Als zich een kans voordoet om een hele goede zet te doen, en de waarde die terug wordt gegeven is extreem hoog, passeert die zet de marge aan de bovenkant, en wordt niet verder onderzocht. Het is dus de kunst om een goede marge in te stellen, die wel optimaliseert, maar niet de meest waardevolle zetten laat staan.

3.6 Quiescence Search

Een schaakprogramma kan niet oneindig diep zoeken naar alle mogelijke zetten. Door het zogenaamde horizon effect kan de computer een erge fout begaan. De beste zet die is gevonden heeft een hoge waarde, bijvoorbeeld omdat een stuk van de tegenstander is geslagen of omdat de tegenstander schaak staat. Waar de computer niet op let is dat zijn tegenstander misschien de volgende beurt het aanvallende stuk weer terug kan slaan met een stuk van lagere waarde. Dit kan leiden tot de overwinning van je tegenstander.

Om dit te voorkomen is er de Quiescence Search (figuur x). Als er in de laatste geanalyseerde zet een stuk geslagen is zal de computer daar niet stoppen met zoeken, maar doorgaan met analyseren. Vanaf die positie worden niet meer alle mogelijke zetten onderzocht, maar alleen de zetten die een ander stuk slaan. Nu kan de computer er zeker van zijn dat de laatste zet die geanalyseerd wordt geen significante negatieve gevolgen meer heeft, en de waarde ervan gebruiken zoals in de rest van de zoektocht naar de beste zet.



Figuur 13

In figuur 13 wordt bij de groene positie een stuk geslagen. Nu wordt niet de waarde twee teruggegeven, maar de computer zoekt verder, naar alle verdere zetten waarin stukken geslagen worden. A kan in deze positie een goede zet doen, wat het resultaat oplevert van 8 in plaats van twee. Bij de analyse van de zet van B wordt dan ook de zet met de waarde van 3 gekozen (en de waarde van 8 wordt verder overgeslagen, β is nu 3 en 8 is groter dan 3). Het uiteindelijke resultaat is een betere positie voor A (nu met een waarde van 3 in plaats van 2 in figuur 13).

4: Onyx

Hoe kun je ooit een computer laten schaken? Dat was de eerste vraag die in mijn hoofd opkwam toen mij gevraagd werd een Profielwerkstuk onderwerp te bedenken. Natuurlijk bestaan er al veel schaak programma's, maar om er zelf een te maken is iets totaal anders. Omdat ik kan schaken, én programmeren erg leuk vind, was een schaakprogramma de perfecte combinatie om veel tijd aan te besteden en me erin te verdiepen. Het zou niet makkelijk worden, maar het moest lukken!

Ik zal in paragraaf 4.1 uitleggen wat programmeren precies inhoudt, en de paragrafen daarna hoe Onyx van nul tot mat is gekomen.

4.1 Programmeren

Voordat ik zal uitleggen hoe ik Onyx in elkaar heb gezet moet eerst wat verteld worden over programmeren. Programmeren is het schrijven van (voor mensen leesbare) code, die instructies geeft aan de computer.[10] Bij het compileren wordt deze code vertaald naar enen en nullen, die voor de computer leesbaar zijn. Dit is ook precies waarin de verschillende programmeertalen van elkaar verschillen: de code die geprogrammeerd wordt is anders voor elke taal, maar de uiteindelijke enen en nullen voor de computer kunnen resulteren in precies hetzelfde programma.

[10] Programmeren: http://nl.wikipedia.org/wiki/Programmeren_(computer)

Een programma schrijven (instructies geven aan een computer) is in vele opzichten anders dan instructies geven aan een mens. De computer doet precies hetzelfde, elke keer dat je hem dezelfde instructies laat uitvoeren. Of je dat één keer doet, of 100 miljoen keer, altijd zal de zelfde uitkomst berekend worden met dezelfde instructies.

Programma's die iets berekenen zonder dat de gebruiker er aan te pas komt zijn saai, want ze doen altijd hetzelfde. Daarom is het leuk als de gebruiker iets kan toevoegen, door zelf iets in te voeren. Het programma wordt gelijk wel een stuk moeilijker, een gebruiker kan heel veel verschillende dingen invoeren, waar allemaal rekening mee moet worden gehouden. De gebruiker wordt dus altijd beperkt in wat hij kan invoeren, dat is ook zo bij Onyx.

Het schrijven van een programma is niet iets dat zomaar gebeurt. Het kost moeite, om een werkend programma te maken. Het schrijven van code gaat nooit in één keer goed, het grootste deel van de programmeertijd ben je bezig je eigen fouten uit de code te halen.

4.2 De start

Gelijk nadat ik het onderwerp van mijn Profielwerkstuk had gekozen, ben ik gaan programmeren. Ik heb de meeste ervaring met de programmeertaal *Java*, een taal die vaak wordt gebruikt voor kleinere programma's. Een minpunt van Java is de snelheid, die een factor tien kan schelen met *C++* of *C#*. Omdat ik te weinig ervaring heb met de laatste twee talen, heb ik Java gekozen. Ik heb me onder andere verdiept in de optimalisatie van Java programma's, zodat de grote zoektocht naar een goede zet voor de computer bijna even snel zou verlopen als in andere talen.

De *Graphical User Interface* is het eerste onderdeel van het schaakprogramma dat ik gemaakt heb. Nog diezelfde dag draaide de *GUI*, een lege huls voor het uiteindelijke programma. Wat er nu mogelijk was, was het gerichte testen van de dingen die ik bouwde voor het uiteindelijke programma. Alles wat ik maakte was nu te zien, en dat helpt bij het analyseren van de fouten die ik onvermijdelijk zou gaan maken.

Na het opzetten van de *GUI* ben ik begonnen aan het programma. De Classes zijn het begin van een programma en daarom heb ik de nieuwe bestanden waar de Classes in staan aangemaakt, met een aantal variabelen erin. De variabelen werden nog niet gebruikt, maar ze schelen werk voor later. Ook heb ik al snel een aantal *Methods* gemaakt, die later vol met code gestopt zouden worden.

4.3 Stukken op het bord

Het begin van de uiteindelijke productie was de denkbeeldige draadjes van de GUI verbinden met het programma. Dit was nodig, omdat een klik op het scherm van de gebruiker geregistreerd en verwerkt moet worden. Dit gebeurt in de vorm van *Listeners*, in het geval van Onyx alleen een *MouseListener*. Zodra de gebruiker ergens binnen het programma klikt, wordt er een 'bericht' gestuurd naar een van de functies binnen het programma. In dat bericht staan de knop waarmee geklikt is en de coördinaten van het punt waar geklikt is. De functie die wordt aangeroepen zorgt verder voor de taken die moeten worden uitgevoerd.

Daarna moesten er stukken op het bord komen. Met hulp van de Piece Class heb ik 32 stukken aangemaakt. Aan de stukken werd een positie op het bord gegeven en een passend plaatje toegewezen. Nu kan de speler een stuk op het bord herkennen.
Vervolgens moet elke klik van de gebruiker worden verwerk. Elke keer als de gebruiker op het bord klikt kijkt het programma of er een stuk van de eigen kleur staat op de positie van de klik. Als dat het geval is wordt dat stuk geselecteerd en kon dat stuk bewogen worden. Als de gebruiker daarna op een leeg vakje klikt dat voor het geselecteerde stuk een geldige zet is, beweegt het stuk daar naartoe.

Om dit te doen waren twee nieuwe functies nodig. De eerste functie kijkt of de zet een geldige zet is voor het geselecteerde stuk: voor alle mogelijke zetten tussen de beginpositie en de eindpositie wordt gekeken of er een stuk staat van de eigen kleur, en of de eigen koning niet schaak wordt gezet. Dit was moeilijk, omdat het veel rekenkracht kostte om elke mogelijke zet te vinden en te analyseren.

Het probleem waar ik snel daarna tegenaan liep, was moeilijk te vinden. Er verschenen geen fouten in het programma, maar er was wel tijdverlies tijdens de berekeningen. Het probleem was, dat de zetten van de dame en de lopers niet alleen binnen het bord werden gegenereerd, maar acht vakjes naar rechts, links, boven en onder vanuit de start positie. Een deel van die vakjes ligt buiten het bord, en werd dus niet meegenomen in de geldige zetten. Ze kostten wel onnodig veel tijd om te berekenen. Figuur 1 laat de toename van de prestatie zien bij een zoekdiepte van 2 en 4 (één en twee zetten vooruit kijken).

Zoekdiepte 2:

Voor:		*Na:*		
T (ms)	**Posities**	**T (ms)**	**Posities**	**Afname tijd**
229	3779	235	3371	*-3%*
60	4797	53	4059	*12%*
46	9649	47	5799	*-2%*
102	17907	31	6907	*70%*
157	32396	73	6570	*54%*
139	37423	48	7063	*65%*
156	31368	33	6082	*79%*
148	35233	30	6229	*80%*
120	30810	44	5621	*63%*

Zoekdiepte 4:

Voor:		*Na:*		
T (ms)	**Posities**	**T (ms)**	**Posities**	**Afname tijd**
4158	1026675	2720	628309	*35%*
5125	1441745	2870	828901	*44%*
14374	4187831	4585	1355626	*68%*
35303	10502299	5699	1711904	*84%*

Figuur 1: Toename prestatie bij zetten generatie binnen bord

De andere nieuwe functie zorgde voor het bewegen van de stukken. Het stuk dat moest bewegen veranderde zijn coördinaten, en binnen het programma werd het bord geactualiseerd, zodat het stuk daar ook op de goede plaats kwam te staan. Als laatste kreeg de GUI de opdracht om het bord met de stukken opnieuw te tekenen, zodat de gebruiker kan zien dat het stuk op zijn nieuwe plek staat.

4.4 Alpha-Beta implementeren

Als er stukken zijn die bewogen kunnen worden en er zijn twee spelers binnen het programma, kan er een schaakspel gespeeld worden. Het is alleen erg saai om alleen te spelen, dus de computer moet zelf ook zetten kunnen bedenken. De *Artificial Intelligence* van de computer moest gemaakt worden, en dat kostte veel onderzoek. Vragen die ik me daarbij stelde waren: waar begint een computer met zoeken? Wordt elke mogelijke zet onderzocht? Hoe diep wordt er gezocht? Kan de computer alle rekenkracht van de processor benutten?

Ik ben begonnen met een simpele Alpha-Beta functie, met hulp van pseudo-code van het internet. Dit mislukte totaal, omdat het einde van de functie niet gedefinieerd was. In paragraaf 3.2 wordt beschreven hoe de functie zichzelf aanroept, en dat gebeurde nu ook, zonder einde. Het programma loopt dan vast, en werkt niet meer.

Er moest dus eerst een evaluatie functie worden gemaakt. De eerste versie van de functie keek alleen naar de stukken op het bord, en naar de positie van de stukken, waarbij elk vakje voor elk stuk een vaste waarde heeft in een tabel. Deze evaluatie functie werkte, en kon verschillende posities (al was de manier waarop niet optimaal) toch van elkaar onderscheiden. Opnieuw heb ik de Alpha-Beta functie gebouwd, en na een aantal dagen debuggen kon de computer een correcte, geldige zet bedenken!

Door de implementatie van de Alpha-Beta functie gebeurde er toch af en toe iets raars. Alle stukken verdwenen systematisch van het bord, zodra de derde zet van de computer was geweest. Het kostte me een aantal dagen om dit op te lossen. Het probleem werd veroorzaakt door twee regels code, die verkeerd om stonden. De twee regels code stonden in de functie die een gemaakte zet weer ongedaan maakte zodat een volgende zet geanalyseerd kon worden. In plaats van eerst een check te doen of de zet die al gedaan was geldig was, werd altijd de zet terug gezet. Dit resulteerde in verplaatsingen van bijna alle stukken op het bord, die nooit verplaatst hadden moeten worden.

4.5 Versnelling en Alpha-Beta Pruning

De Alpha-Beta functie was langzaam, veel te langzaam. Dit kwam door het aanmaken van veel objects, die niet nodig waren. Objects nemen geheugen in, en zodra je er eentje aanmaakt moet dat stukje geheugen gereserveerd worden. Voor één object maakt dat niet uit, maar voor 10 miljoen objects wel. De *Garbage Collect*, een deel van *de Java Virtual Machine* (de omgeving waar Java programma's in draaien) dat alle objects opruimt zodra ze overbodig zijn is niet efficiënt, en kostte te veel tijd.

De oplossing was, om een groot deel van de functies die alle mogelijke zetten bedachten opnieuw schrijven, zodat die *arrays* in plaats van objects gebruikten. Arrays zijn series van getallen, die ontzettend snel gemaakt en gelezen kunnen worden. Door arrays te gebruiken versnelde het proces van de beste zet aanzienlijk, maar het was nog niet genoeg.

Via fora op internet kwam ik uitbreidingen op de Alpha-Beta functie op het spoor. Een daarvan was Alpha-Beta Pruning, uitgelegd in paragraaf 3.2.1. Alpha-Beta Pruning was niet makkelijk te implementeren, door de twee variabelen α en β. Elke keer als de Alpha-Beta functie zichzelf opriep, moesten de twee waardes van de variablen genegativeerd worden, omdat de speler dan ook wisselt. Dit gaf moeilijkheden met de waardebepaling van de positie in de evaluatiefunctie, omdat soms de waarde daar ook genegativeerd moest worden (afhankelijk van de speler die aan zet is). Pas later heb ik daar veel code kunnen verwijderen toen ik door had waar het probleem zat. Dit probleem zorgde er ook juist voor, dat de beste zetten soms als slechtste werden aangezien, en andersom.

Voor:		*Na:*		
T (ms)	**Posities**	**T (ms)**	**Posities**	**Afname tijd**
1653	628309	359	36037	*78%*
2137	1032560	78	18902	*96%*
2652	1280852	78	27625	*97%*
1903	907890	109	32053	*94%*
3729	1757717	140	47717	*96%*
4524	2162031	172	55697	*96%*
6568	3289810	187	72891	*97%*
2995	1486275	141	49871	*95%*
3354	1627666	156	52987	*95%*

Figuur 2: Toename prestatie na implementatie Alpha-Beta Pruning (zoekdiepte 4)

Uiteindelijk lukte het, om α en β onder controle te krijgen, en de Alpha-Beta Pruning werkend te krijgen. In figuur 2 staat de ongelofelijke versnelling van de uitbreiding op het algoritme. Onyx werd steeds speelbaarder.

4.6 Debugging en schaak

Op school heb ik ook een aantal keer aan Onyx gewerkt. Dit werkte niet goed, omdat de resolutie van de school computers zo laag was, dat Onyx groter was dan het hele scherm. Ik heb dus een keuze mogelijk gemaakt in het optieschermpje bij het opstarten, voor de grote of de kleine versie. Zo kan elke computer Onyx goed weergeven.

Ook het *debuggen* (regel voor regel meelezen met de computer tijdens de uitvoering van Onyx om problemen in het programma op te sporen) begon echt. Steeds vaker ontdekte ik kleine foutjes, die zetten veroorzaakten die niet mogelijk moesten zijn. Zo waren pionnen vaak een

groot probleem: soms mogen ze twee stappen vooruit en ze slaan schuin. Welke kant de pionnen op mogen schuiven en slaan hangt af van welke kleur ze zijn en aan welke kant van het bord die kleur staat. Een klein foutje veroorzaakte pionnen die andere stukken recht vooruit sloegen, over andere stukken heen sprongen, of stukken van hun eigen kleur sloegen. Om dit op te lossen heb ik de code veranderd die de zetten genereerde voor de pionnen. Het maakte bij dit probleem uit of de pion van de computer was of niet, om uit te vinden welke kant hij op mocht schuiven.

Het implementeren van 'schaak zetten' ging makkelijker dan verwacht. De enige check die nodig was, is of de koning werd aangevallen, door welk stuk dan ook. De problemen kwamen pas echt toen ik probeerde zetten te verhinderen die de eigen koning schaak zetten. Voor elke mogelijke zet moest de eigen koning eerst gevonden worden op het bord, de zet gedaan worden, en dan gekeken of de koning schaak stond. Zo ja, dan was de zet ongeldig.

4.7 C#

De PWS dag van de tweede activiteiten week heb ik besteed aan het porten (omzetten van programmeertaal in een andere taal) van Onyx in *C#*. Soms kan C# tot 20 keer sneller zijn dan Java, en ik wilde graag proberen of dat bij mijn programma ook het geval was. Ik heb artikelen gelezen over Visual Basic, een programmeeromgeving voor C#, waarmee het makkelijk is om een GUI te maken, en een programma met een Engine daaromheen te breien. Ik heb me die ochtend verdiept in C# en alle vervangingen van bijzondere Classes die ik in Java gebruikte om hetzelfde doel te bereiken in C# als me in Java gelukt was. Ondanks de weinige ervaring die ik heb met C# was dit een kwestie van veel kopiëren en plakken, samen met zoeken en vervangen. Zo kon ik toch enigszins het programma draaiend krijgen.

Het is gelukt om Onyx te laten spelen, al is het geen heel spel. Omdat ik maximaal één ochtend aan C# wilde besteden heb ik nauwelijks tijd gehad om te debuggen, en heb ik alleen geanalyseerd of C# sneller was bij het berekenen van een zet. De resultaten die uit de Engine kwamen waren enigszins teleurstellend: de rekentijden waren bijna precies gelijk (figuur 3).

Java:		*C#:*		
T (ms)	**Posities**	**T (ms)**	**Posities**	**Afname tijd**
412	4593	468	4593	*-14%*
897	21118	982	21118	*-9%*
1401	33022	1205	33022	*14%*
234	4003	259	4003	*-11%*

Figuur 3: Toename prestatie na porten naar C# (zoekdiepte 4)

Toch heb ik voordeel gehad van C#, door de layout van de GUI die ik in had gemaakt. In de C# GUI was er een menu, een statusbalk en mooie Windows knoppen in plaats van de standaard Java knoppen. Natuurlijk is dat ook in Java mogelijk, het kostte wat onderzoek om te ontdekken hoe. De status- en menubalk laten Onyx er mooier uitzien, en het gebruikersgemak wordt vergroot.

4.8 Opslaan en laden

Het is handig als een schaakspel opgeslagen en weer geladen kan worden. Om dit te doen heb ik twee nieuwe functies geschreven, die een positie vertalen naar een reeks letters en cijfers. Deze notatie van een schaakpositie wordt de *FEN-notatie* genoemd. In plaats van een bord weer te geven, wordt voor elke rij de stukken erop weergegeven. Voor lege vakjes worden geen spaties gebruikt, maar het aantal totaal aan elkaar grenzende lege vakjes. De beginpositie van een schaakspel ziet eruit als in figuur 4[11]:

[11] Fen Notation: http://mediocrechess.blogspot.com/2006/12/guide-fen-notation.html

```
rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq –
```

Figuur 10: de FEN notatie

Met kleine letters worden de zwarte stukken aangegeven, met hoofdletters de witte stukken. De enkele letter bepaalt de kleur aan zet, de vier letters de rechten om te rokeren voor beide spelers. Het laatste symbool is de kolom waar een en-passant zet mogelijk is. Onyx slaat een bord op met gedeeltelijke andere notatie. Omdat de gebruiker kan kiezen of hij met wit of zwart speelt, is er een extra element dat wordt toegevoegd aan het begin van het bestand. Ook wordt het versienummer van Onyx meegenomen in het bestand, zodat er geen problemen ontstaat wanneer een gebruiker een bestand van een andere versie probeert te openen.

Conclusie

Een citaat van Johnny Bigert dat ik perfect bij Onyx vind passen: "*Chess computer programming is not the art of programming, it is the art of debugging."*[12] Ik heb ontdekt dat programmeren soms behoorlijk lastig is, en dat ik een groot deel van alle programmeertijd fouten maak. Omdat een schaakprogramma zo'n groot project is, kostte het debuggen relatief veel meer tijd. Dat was nieuw voor mij, en erg leerzaam.

Wat ik nooit van te voren had kunnen bedenken is de complexiteit van simpele dingen, zoals een zet op een schaakbord. Wat voor een mens vanzelfsprekend is, is voor een computer bijna onmogelijk. Een omweg bedenken, zodat een computer iets kan berekenen dat voor een mens als 'normaal' beschouwd wordt, was één van de dingen die mij veel plezier hebben gegeven.

Het Alpha-Beta algoritme was helemaal nieuw voor mij, en ik vind het fascinerend hoeveel diepgang een klein stuk code kan hebben en het complexe resultaat dat ik ermee heb kunnen bereiken. Het zoek-algoritme is waarschijnlijk het onderdeel waar ik het langste aan geprogrammeerd heb. Ook is het gelukt om Onyx een moeilijkheidsgraad te geven, door de Engine dieper en minder diep te laten zoeken.

In totaal heb ik 68.002 tekens geprogrammeerd (zonder spaties), in 3249 regels code, die in 12 classes zijn ondergebracht.

Na drie maanden werk kan ik met plezier terugkijken op de uren die ik met Onyx heb doorgebracht. Ik ben tegen problemen aangelopen, waar ik zonder literatuur, websites en originele ideeën van anderen nooit uitgekomen was. Ik ben tevreden met het resultaat: een goed draaiende schaakcomputer, die een mens een goede pot schaak kan bezorgen. Toch zal Onyx nooit af zijn: er zal altijd uitbreiding, versnelling en verbetering mogelijk zijn…

[12] Rainman chess: http://www.johnnybigert.se/chess_computer.html

Begrippenlijst

Boom (van analyse): Alle mogelijke vervolg-zetten van een bepaalde positie tot een bepaalde diepte. Meestal weergegeven met punt-lijn diagram, met de beginpositie als oorsprong.

Class: Een beschrijving van een aantal eigenschappen, die bij elk Object van die class hetzelfde zijn. Zo zou de Class 'Auto' de eigenschappen 'merk', 'kleur' en 'ruimte' bevatten.

Debuggen: Regel voor regel meelezen met de computer tijdens de uitvoering van een programma om problemen en fouten in het programma op te sporen.

Functie: Zie *Method*.

Java: Een platformonafhankelijke, object georiënteerde programmeertaal met een uitgebreide Class-library. Java is grotendeels gebaseerd op C++, met veel gelijke syntax.

Leaf (van analyse): De laatste positie in de boom, verder wordt er niet geanalyseerd. *Leaf* is afgeleid van een blaadje aan het einde van een tak aan een boom.

Method: Een deel van een *Class* in een Java programma, dat nul of meer variabelen als input krijgt, er iets mee doet, en zo nodig een variabele terug geeft. *Methods* worden gebruikt voor stukken code die erg vaak worden herhaald, zodat dezelfde code niet herschreven hoeft te worden.

Positie: Een bepaalde situatie van het spel. Een positie is onder andere van het aantal stukken op het bord, de plek van de stukken en de speler aan zet afhankelijk.

Pseudo-code: Programmeer taal die voor iedereen leesbaar is, en niet als een echt programma kan draaien. Met pseudo-code wordt de werking van een stuk code uitgelegd.

Tak (van analyse): Alle mogelijke zetten van een bepaalde positie, die niet de eerste positie van analyse is.

Object: Een voorbeeld van een Class. Zoals een Object een voorbeeld is van een Class is 'Rood' een voorbeeld is van een 'Kleur'.

Zet: Een verplaatsing van een schaakstuk door een speler. Na een zet krijgt de andere speler de beurt.

Bronnen

Boeken

E.A. Heinz, 'AEL Pruning', **in**: ICGA Journal March 2000, p. 21 – 32, Cambridge (VS), 2000
P. Pinto, 'Introducing the Min-Max Algorithm', **in**: AI Depot Contest, 2002
P. Sestoft, *Java performance, Reducing time and space consumption* , Copenhagen (Denmark), 2005
M. Watson, *Practical Artificial Intelligence Programming With Java* , 2008
J. Wang, S. Li & X. Xu, *Minors Hash Table in Computer Chinese Chess,* ShenYang (China)

Internet: artikelen[*]

F. Friedel, *A short history of computer chess*, http://www.chessbase.com/columns/column.asp?pid=102
C. Frayn, *Computer Chess Programming Theory* , http://www.frayn.net/beowulf/theory.html#killer, 2007
Y. Lin, *Game Trees,* http://www.ocf.berkeley.edu/~yosenl/extras/alphabeta/alphabeta.html
J. Pettersson, *Aspiration Windows, Killer moves and PVS*,
http://mediocrechess.varten.org/guides/aspirationwindows.html
E. Schröder, *The Inner Work Of A Chess Program*, http://www.top-5000.nl/authors/rebel/chess830.htm
L.C. Wee, *Introduction To Combination Game Theory,* http://sps.nus.edu.sg/~limchuwe/cgt/
Writing a chess program , http://fchess2.blogspot.com/2009/10/piece-graphics-and-internal-position.html, 2009

Internet: algemene informatie[*]

Alpha-Beta pruning, http://en.wikipedia.org/wiki/Alpha-beta_pruning
Chess history, http://www.computerhistory.org/chess/
Chess rules, http://en.wikipedia.org/wiki/Chess
Endgame table, http://en.wikipedia.org/wiki/Tablebase
Endgame, http://chessprogramming.wikispaces.com/Endgame
Min-Max, http://www.stanford.edu/~msirota/soco/minimax.html
Min-Max, http://www.stanford.edu/~msirota/soco/alphabeta.html
Min-Max, http://www.ocf.berkeley.edu/~yosenl/extras/alphabeta/alphabeta.html
Move ordering, http://www.top-5000.nl/authors/rebel/chess840.htm
Move ordering, http://vicki-chess.blogspot.com/2007/05/move-ordering.html
NegaMax, http://en.wikipedia.org/wiki/Negamax
Quiescence Search, http://en.wikipedia.org/wiki/Quiescence_search
Chess AI Logic, http://www.dreamincode.net/forums/topic/100664-chess-ai-logic/
Chess Tree Search, http://verhelst.home.xs4all.nl/chess/search.html
Programmeren: http://nl.wikipedia.org/wiki/Programmeren_(computer)
Programming Chess Wiki, http://chessprogramming.wikispaces.com/
Rainman chess: http://www.johnnybigert.se/chess_computer.html
Schaken, http://en.wikipedia.org/wiki/Chess
Schaken, http://mathworld.wolfram.com/Chess.html
Wolfram Alpha, http://www.wolframalpha.com/

Foto's

H. Wieringa, *Reflecting chesspieces*

Diagrammen

H. Wieringa, gemaakt met het programma *yEd Graph Editor*

---

[*] De gebruikte informatie van alle websites komt van de websites in de staat zoals ze op 8 december 2011 waren.