Kunstmatige intelligentie: opdracht 3

Tim van der Meij (1115731) en Simon Klaver (1140760)

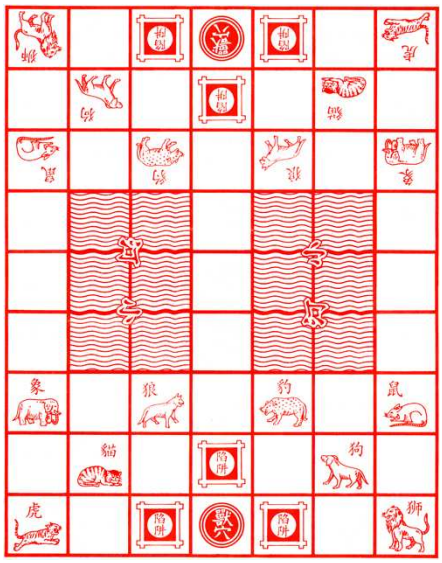
23 april 2013

1 Inleiding

In dit verslag beschrijven we onze uitwerking van de derde programmeeropdracht voor het vak *kunstmatige intelligentie*. De originele opdracht is te vinden op [1]. Het doel van de opdracht is om binnen een gegeven framework voor het Jungle-spel (zie [3]) een zoekalgoritme te schrijven dat, gegeven een bepaalde spelconfiguratie, een zo goed mogelijke zet vindt. We gaan hiertoe twee algoritmen implementeren: negamax en alpha-beta pruning. Uiteindelijk zal de implementatie van deze twee algoritmen in het framework samen met een evaluatiefunctie (die de gegeven spelconfiguraties op diverse relevante criteria beoordeelt) opgeleverd worden.

2 Uitleg probleem

Jungle, ook wel Jungle Chess of *Dou shou qi* (in het Chinees) genaamd, is een bordspel waarin het doel van de speler is om één van zijn stukken op een speciaal vakje (de den) aan de overkant van het rechthoekige spelbord te krijgen (zie figuur 1). Dat kan hij enkel doen door stukken één vakje naar boven, onder, links of rechts te verschuiven. Ieder stuk is een dier en heeft een bepaalde rang. Een stuk van de tegenstander kan geslagen worden als dat stuk een gelijke of lagere rang heeft dan het stuk van de speler. Als een stuk een ander stuk, van gelijke rang, van de tegenstander slaat, blijft het stuk dat slaat staan. De rat is een geval apart in het spel: hoewel hij de laagste rang heeft, kan hij zich (in tegenstelling tot de rest van de stukken) door het water bewegen en kan hij de olifant (met de hoogste rang) slaan. De leeuw en de tijger kunnen springen over het water.



Figuur 1: Het 7×9 spelbord voor Jungle

Rondom de den (de rode cirkels in figuur 1) bevinden zich *traps*. Op deze vakjes wordt een

stuk van de tegenstander verlaagd tot de laagste rang. Het is dan dus voor die tegenstander zaak dat er geen vijandige stukken naast hem staan, want dan kan hij direct geslagen worden.

We willen voor deze opdracht het gegeven Jungle-framework slimmer maken door algoritmen in te bouwen die bepalingen kunnen doen voor de beste zet voor een gegeven spelconfiguratie. Hiertoe gaan we de algoritmen negamax en alpha-beta pruning inbouwen. Deze twee algoritmen worden hieronder in de sectie 'Relevant werk' nader toegelicht. Tevens willen we een evaluatiefunctie maken die de spelconfiguraties gaat beoordelen. De volgende factoren (criteria) zullen in de evaluatiefunctie getoetst worden en hebben dan ook invloed op de berekening van de evaluatiefunctie.

- Het aantal in het spel zijnde stukken van zowel de speler als de tegenstander.
- De totale waarde van de in het spel zijnde stukken van zowel de speler als de tegenstander.
- De minimale afstand van een stuk van zowel de speler als de tegenstander tot de den van respectievelijk de tegenstander en de speler.

Met behulp van deze drie criteria kunnen we een gegeven spelconfiguratie beoordelen op zijn waarde en daarmee de twee zoekalgoritmen helpen met het vinden van de beste zet.

3 Relevant werk

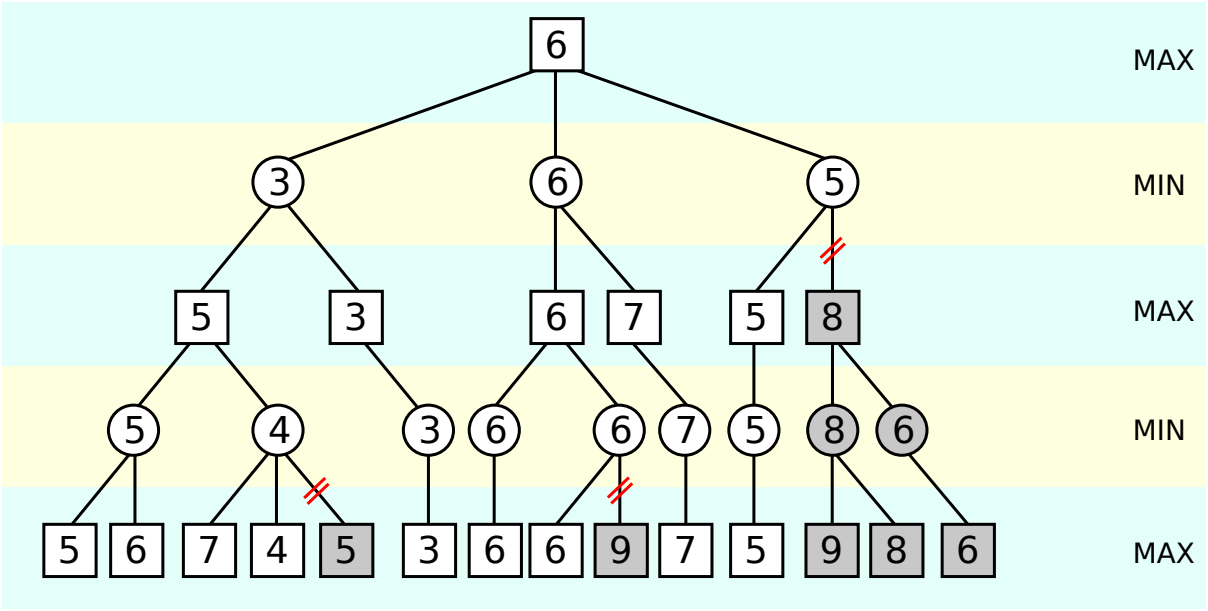
We gebruiken onderstaande werken als inspiratie voor deze opdracht.

- Jungle-framework [2]
- Spelregels Jungle [3]
- Negamax [4]
- Alpha-beta pruning [5]

De twee algoritmen die we voor deze opdracht gaan implementeren in het framework zijn negamax en alpha-beta pruning. Negamax is een variant van het minimax-algoritme. Het minimax-algoritme maakt een boom van alle mogelijke zetten van het programma bij een bepaalde spelconfiguratie, de beste zet die de tegenstander daarop kan doen, de volgende zetten van het programma, enzovoort. De beste zet voor het programma kan vervolgens bepaald worden door aan alle zetten in de boom een score toe te kennen. Dit gebeurt met een speciale evaluatiefunctie. Hierbij geldt over het algemeen dat een hoge score betekent dat een zet goed is voor het programma en een lage score dat de zet nadelig is voor het programma. De bepaling van de beste zet wordt dan gedaan door in iedere knoop van de boom de maximale score van de kinderen te stoppen als men zelf aan de beurt is en de minimale score van de kinderen als de tegenstander aan de beurt is. Negamax is een variant van het minimax-algoritme dat gebruikt maakt van de zero-sumeigenschap. Dit houdt in dat de opbrengsten van de spelers netto op 0 uitkomen: als een speler iets qua score wint, moet de andere speler evenveel qua score verliezen.

Alpha-beta pruning heeft als doel om de zoekruimte van negamax- en minimax-algoritmen te beperken door takken te snoeien (*prunen*) als dat mogelijk is. Een zet wordt dan niet verder geëvalueerd als tijdens de evaluatie blijkt dat de zet zeker slechter zal zijn dan een voorheen bekeken zet. Takken die de uiteindelijke beslissing onmogelijk kunnen beïnvloeden, worden door dit algoritme gesnoeid om zo de zoekruimte vaak aanzienlijk te verkleinen.

Figuur 2 geeft weer hoe alpha-beta pruning werkt. In essentie wordt het minimax-algoritme (of negamax in ons geval) uitgevoerd, maar zodra blijkt dat een vertakking geen betere waarde op kan leveren dan een reeds gevonden waarde wordt die vertakking gesnoeid. In figuur 2 is de winst van dit algoritme goed te zien bij knoop 5 op het tweede niveau. De linkersubboom van die knoop is geëvalueerd en daar is de score 5 als minimum berekend. Hierdoor is het niet meer nodig om de rechtersubboom te bekijken, want ongeacht de score die daar berekend zou worden zou de subboom niets veranderen aan de uiteindelijke beslissing. Immers, als de waarde hoger is dan 5 zou het minimum ervoor zorgen dat de 5 genomen wordt. Als het kleiner dan 5 is maakt het ook niet meer uit, want er is als kandidaat voor de max-knoop op niveau 1 al eerder een score van 6 gedetecteerd. Een kleinere waarde voor min-knoop 5 zal dus geen effect hebben op de uiteindelijke beslissing van het algoritme.



Figuur 2: Grafische weergave van alpha-beta pruning [5]

4 Aanpak

Het eerste deel van de opdracht is om de twee in de sectie 'Relevant werk' genoemde algoritmen te implementeren in het Jungle-framework. We beginnen hiertoe met de meest basale van de twee: negamax. Voor de implementatie van dit algoritme beginnen we met het opvragen van een lijst met alle mogelijke zetten voor de gegeven spelconfiguratie.

Hiervoor bevindt zich een functie in het gegeven framework. Vervolgens gaan we alle zetten voor de gegeven spelconfiguratie langslopen. Hiertoe doen we de zet, gaan we in recursie om tot diepte 0 te komen en de verkregen spelconfiguratie te evalueren en maken we de zet weer ongedaan. Door de recusieve aanroep zullen vanaf de eerste spelconfiguratie alle volgende spelconfiguraties tot een gegeven diepte (aantal zetten vooruitkijken) gemaakt worden. Als we in het laatste niveau van de spelboom terechtgekomen zijn, gaan we de evaluatiefunctie loslaten op de verkregen configuratie. We houden tijdens dit hele proces telkens bij wat de maximale waarde en de daarbij horende beste zet is opdat de functie die aan het einde van het algoritme terug kan geven voor de speler.

Vervolgens bouwen we het algoritme voor alpha-beta pruning in. Aangezien alpha-beta pruning een verbeterde versie van negamax is, nemen we de complete negamax-code als basis voor dit algoritme. Dit vullen we aan met logica om een tak in de boom te snoeien als die tak onmogelijk nog een waarde op kan leveren die de uiteindelijke beslissing van het algoritme kan beïnvloeden. De implementatie hiervan volgt uit de pseudocode in [4].

Ten slotte moet er een evaluatiefunctie gemaakt worden. Deze functie heeft tot doel iedere gegeven spelconfiguratie op waarde te schatten en is dus van essentieel belang voor het succes van de twee zoekalgoritmen. In onze evaluatiefunctie bouwen we de drie criteria in zoals genoemd in de sectie 'Uitleg probleem' hierboven. Als een spelbord winnend is voor de huidige speler, dan wordt er een zeer hoge waarde teruggegeven door de evaluatiefunctie opdat die zet altijd als beste uit de bus komt. Als een spelbord nog niet winnend is, gaan we alle stukken op het bord langslopen en statistieken over het spelbord vergaren.

We tellen allereerst de totale waarde van het aantal zwarte en witte stukken op het bord. Voor de uiteindelijke evaluatie nemen we het verschil van deze twee waarden. Als wit aan de beurt is, wordt de waarde van de aanwezige zwarte stukken afgetrokken van de waarde van de aanwezige witte stukken en vice versa. Als de stukken van zwart dus bij elkaar een grotere waarde hebben dan de stukken van wit en wit is aan de beurt, dan veroorzaakt dit een negatieve waarde en dus een lagere waarde voor de totale evaluatiefunctie. Als de huidige speler dus door een aantal zetten stukken overhoudt met in totaal een kleinere waarde dan de stukken van de tegenstander, wordt dat gezien als een minder goede zet.

Ten tweede bepalen we de minimale afstand van de stukken van zwart en wit tot aan de den van de bijbehorende speler. Dit doen we zodat we een spelconfiguratie waarbij de minimale afstand tot de den klein is als een goede configuratie kunnen bestempelen. Aangezien het framework geen coördinaten van vakjes teruggeeft (maar in plaats daarvan vaknummers) moeten we gebruik maken van de structuur van het bord om de minimale afstand tot de den te berekenen. Omdat het bord uit rijen van 7 vakjes bestaat, kan de afstand tot de den berekend worden met de formule $(\text{abs}(i - den)\ /\ 7) + (\text{abs}(i - den)\ \%\ 7)$. Hierin is i het vaknummer van het stuk zelf en den het vaknummer van de den waar het stuk naartoe moet. Neem bijvoorbeeld i gelijk aan 9 en den gelijk aan 59. Als we de berekening uitvoeren, komen we voor dit voorbeeld uit op een afstand van 8 tot de den, hetgeen op het bord neerkomt op één zet naar rechts en dan zeven zetten naar boven. We gaan dus voor ieder stuk op deze manier de afstand tot het doel bepalen en we houden telkens de minimale afstand bij voor zwart en wit.

De evaluatiefunctie wordt uiteindelijk beïnvloed door de minimale afstand van zwart en wit. We trekken de minimale afstand namelijk af van de maximale afstand (dat is 11: vanaf de linkeronderhoek tot aan de den beneden 3 zetten en dan vanaf die den naar de den bovenaan nog eens 8 zetten) om zo een kleine minimale afstand een zo hoog mogelijke waarde te laten veroorzaken en vice versa. Deze waarde vermenigvuldigen we met de hoeveelheid stukken van de huidige speler om zo configuraties met veel stukken èn een minimale afstand tot de den een betere beoordeling te geven dan configuraties met minder stukken. Het aantal stukken per speler houden we hiertoe ook telkens bij.

Om de evaluatiefunctie slimmer te maken, berekenen we de waarden telkens voor beide spelers. Zo berekenen we als wit aan de beurt is ook de minimale afstand tot de den van zwart. Dat doen we zodat we de twee spelers aan elkaar kunnen relateren. Als de situatie voor de tegenstander namelijk beter is dan die van de huidige speler (bijvoorbeeld als de tegenstander een kleinere minimale afstand tot de den heeft), moet de beoordeling van het bord genuanceerd worden, ook al leek hij in eerste instantie gunstig voor de huidige speler.

Hieruit blijkt dus dat de waarde van de stukken, de hoeveelheid stukken en hun positie op het bord (ten opzichte van het doel) bepalend zijn voor de uitkomst van de evaluatiefunctie.

5 Implementatie

Voor deze opdracht gebruiken we de programmeertaal C++. Bovendien bouwen we voort op een reeds bestaand programma dat ons verstrekt is door Jonathan Vis (zie [2]). Onze implementatie is te vinden in de appendix van dit verslag. In de appendix hebben we enkel de gewijzigde bestanden geplaatst. De overige bestanden van het Jungle-framework zijn ongewijzigd gebleven in onze implementatie en kunnen gevonden worden op [2].

6 Experimenten

We willen nu het verschil tussen het aantal bezochte knopen en bladeren door negamax en alpha-beta pruning onderzoeken. Hiertoe verzamelen we de benodigde gegevens over het aantal bezochte knopen en bladeren tot 7 niveaus diep vanaf de beginconfiguratie. De verzamelde gegevens zijn in de tabellen 1 en 2 hieronder te vinden.

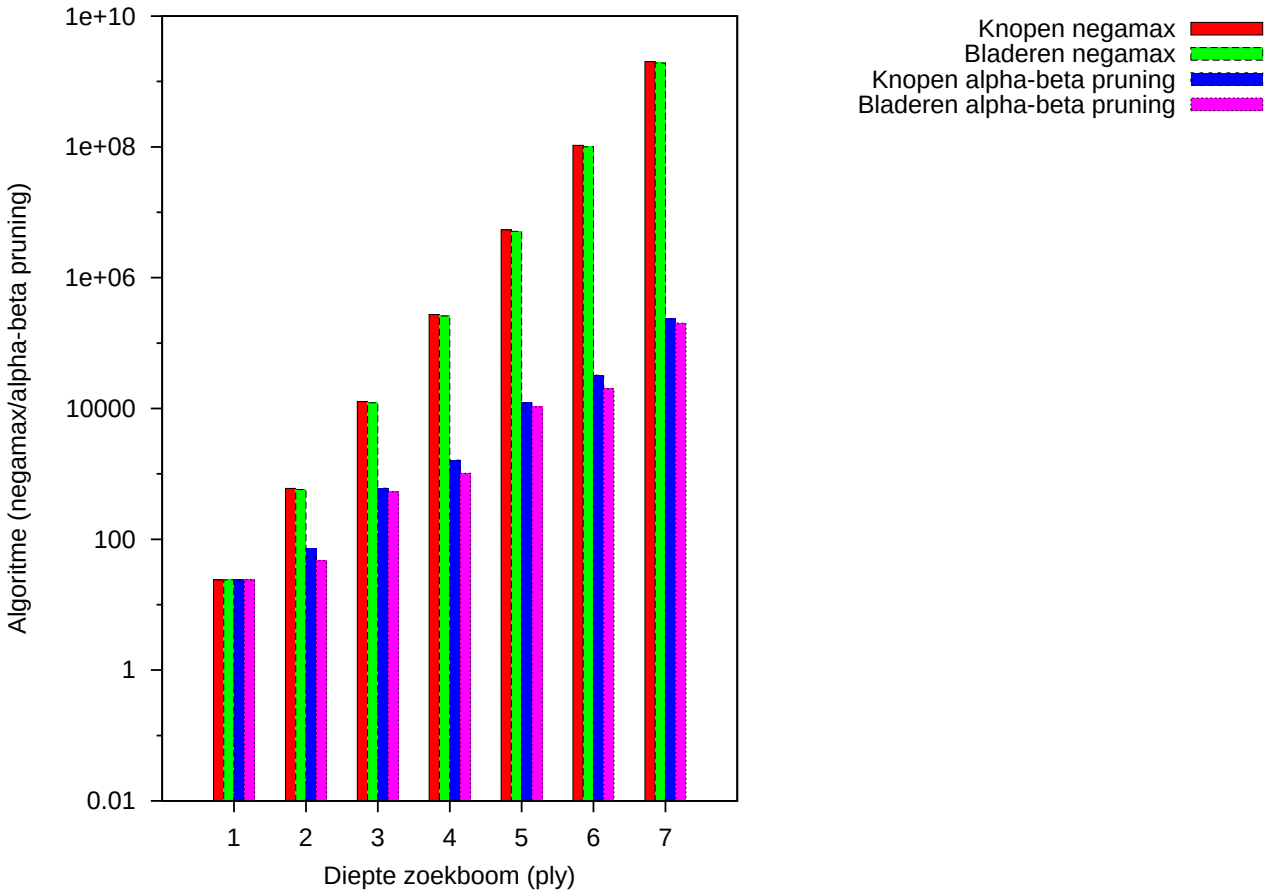
Tabel 1: negamax

Ply	Knopen	Bladeren	Vertakkingsfactor
1	24	24	24
2	600	576	24
3	12840	12240	21.25
4	272940	260100	21.25
5	5384665	5111725	19.65
6	105844654	100459989	19.65
7	2014328075	1908483421	19

Tabel 2: alpha-beta pruning

Ply	Knopen	Bladeren	Vertakkingsfactor
1	24	24	24
2	71	47	1.96
3	604	533	11.34
4	1623	1019	1.91
5	12155	10532	10.33
6	32221	20066	1.91
7	233910	201629	10

Deze gegevens hebben we verkregen door het programma enigszins aan te passen waardoor tellers ingezet worden in de functies voor negamax en alpha-beta pruning. Als we gaan evalueren, hogen we het aantal bladeren met één op. Na de recursieve aanroep hogen we het aantal knopen met één op. Met de verzamelde gegevens hierboven kunnen we de grafiek zoals weergegeven in figuur 3 maken. Merk op dat de y-as een logaritmische schaalverdeling aanhoudt om de waarden goed te kunnen weergeven in de grafiek.



Figuur 3: Ply versus aantal bezochte knopen/bladeren per algoritme

Zeer opvallend in deze grafiek is de drastische verbetering van alpha-beta pruning ten opzichte van negamax. In het bijzonder op ply 7 worden met alpha-beta pruning $\frac{2014328075}{233910} = 8612$ keer zo weinig knopen bekeken als bij negamax; een significante verbetering voor

de zoekcomplexiteit. Merk op dat de verbetering van alpha-beta pruning ten opzichte van negamax afhangt van de evaluatiefunctie. Hoe beter de evaluatiefunctie, hoe beter alpha-beta pruning takken in de boom kan snoeien.

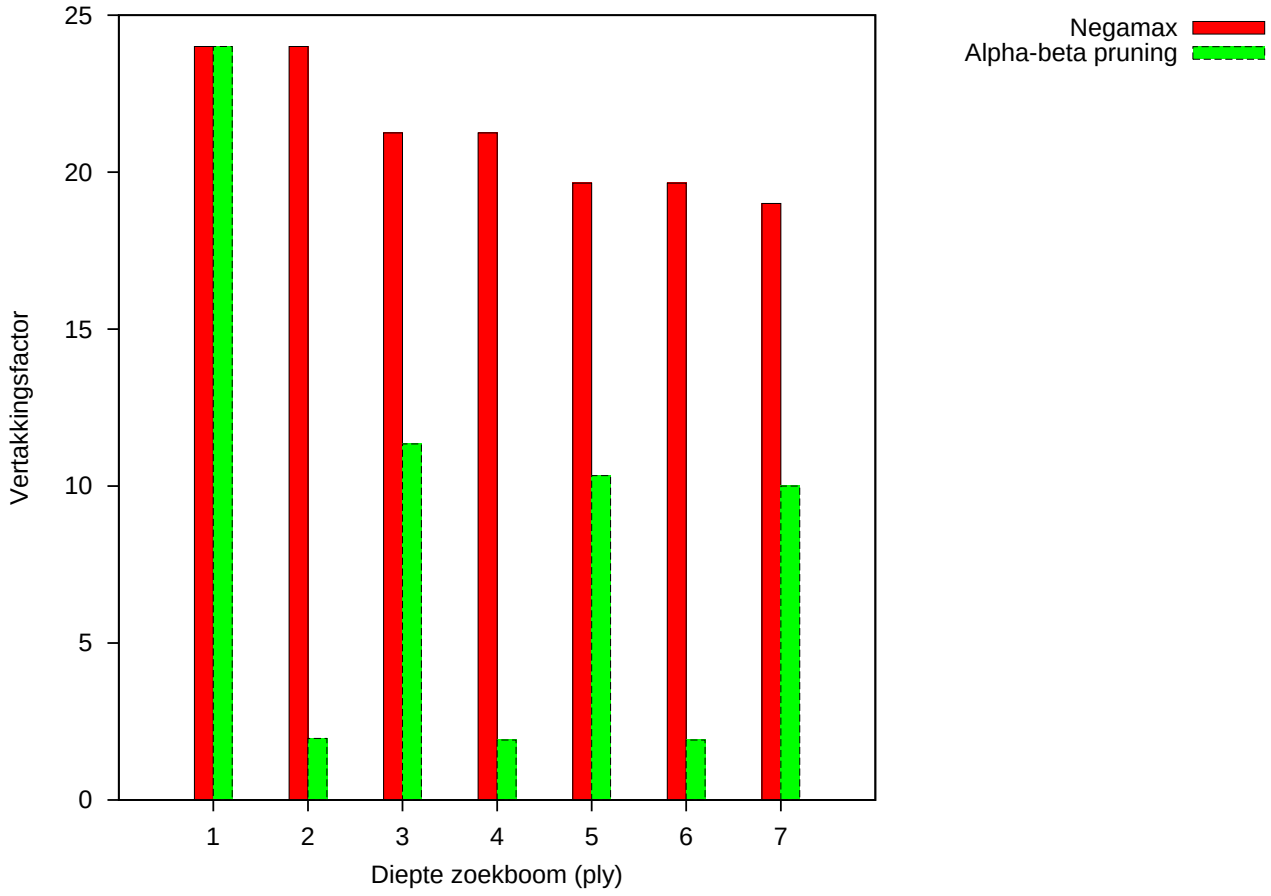
Vervolgens zijn we geïnteresserd in de vertakkingsfactor (*branching factor*) van alpha-beta pruning ten opzichte van negamax. In de speltheorie is de vertakkingsfactor van een boom gelijk aan het aantal kinderen van een knoop. Vaak is het aantal kinderen echter niet hetzelfde voor alle knopen, dus wordt er in dat geval gewerkt met de gemiddelde vertakkingsfactor (*average branching factor*). We gaan de vertakkingsfactor van negamax en alpha-beta pruning per ply berekenen en vervolgens deze waarden relateren aan de theoretische vertakkingsfactor van alpha-beta pruning om zo inzicht te krijgen in de prestaties van onze implementatie.

Om de benodigde vertakkingsfactoren te verkrijgen hoeven we het programma niet aan te passen. We hebben alle benodigde metingen al gedaan voor het onderzoek naar het verschil in bezochte knopen en bladeren van negamax en alpa-beta pruning (zie tabellen 1 en 2 hierboven). Volgens [6] kunnen de vertakkingsfactoren met behulp van de gegevens in de tabellen op verschillende manieren berekend worden. Omdat de verschillende manieren allemaal op dezelfde resultaten uitkomen, kiezen wij voor de methode voor de *effective branching factor*. In feite houdt de methode in dat men de vertakkingsfactor op ply n kan berekenen door het aantal knopen op die ply te delen door het aantal knopen op ply $n-1$. Echter, omdat in onze statistieken het aantal knopen niet alleen het aantal knopen op ply n is, maar het aantal knopen op ply n en alle ply daarvòòr, moeten we in ons geval met het aantal bladeren werken.

Als we de berekeningen uitvoeren, krijgen we de waarden zoals gepresenteerd in de tabellen 1 en 2 hierboven. Een voorbeeld van de berekening van een vertakkingsfactor is bijvoorbeeld die van negamax op ply 4. We delen dus het aantal knopen van ply 4 voor het aantal knopen van ply 3, hetgeen ons 260100 / 12240 = 21.25 oplevert. Merk op dat er op ply 0 maar één knoop en blad is, namelijk de wortel. Dat gegeven hebben we nodig om ook de vertakkingsfactor van ply 1 te kunnen berekenen.

Nu we de vertakkingsfactoren berekend hebben, willen we ze ten slotte relateren aan de theoretische vertakkingsfactor van alpha-beta pruning. Volgens [6] is de theoretische vertakkingsfactor van alpha-beta pruning ongeveer gelijk aan de wortel van de gemiddelde vertakkingsfactor van negamax. Dit geldt echter alleen bij de meest optimale ordening van zetten in de boom.

Om nu een uitspraak te kunnen doen over de prestaties van onze implementatie, willen we kijken of onze implementatie de theoretische ondergrens voor alpha-beta pruning benadert. Zo ja, dan is de zoekboom vrij optimaal. Zo niet, dan kan er niet genoeg gesnoeid worden. We merken als eerste op dat uit onze statistieken in tabel 2 blijkt dat het zogenaamde 'odd-even-effect' [6] speelt: even ply hebben een veel lagere vertakkingsfactor dan oneven ply. Dit effect wordt veroorzaakt door de structuur van de optimale zoekboom van alpha-beta pruning. Als we de gegevens over de vertakkingsfactoren in de tabellen 1 en 2 uitzetten in een grafiek (zie figuur 4), is het odd-even-effect duidelijk zichtbaar.



Figuur 4: Ply versus vertakkingsfactor per algoritme

Om ondanks dit effect toch nuttige uitspraken te kunnen doen over de optimaliteit van de zoekboom, berekenen we voor zowel negamax als alpha-beta pruning het meetkundig gemiddelde [7], hetgeen gedefinieerd is als $\sqrt[n]{a_1 \cdot a_2 \cdot ... \cdot a_n}$. Voor negamax krijgen we dan $\sqrt[7]{24 \cdot 24 \cdot 21.25 \cdot 21.25 \cdot 19.65 \cdot 19.65 \cdot 19} \approx 21.24$ en voor alpha-beta pruning krijgen we dan $\sqrt[7]{24 \cdot 1.96 \cdot 11.34 \cdot 1.91 \cdot 10.33 \cdot 1.91 \cdot 10} \approx 5.72$.

In het optimale geval zou het meetkundig gemiddelde van alpha-beta pruning precies de wortel van het meetkundig gemiddelde van negamax moeten zijn. In ons geval zou er voor alpha-beta pruning dan $\sqrt{21.24} \approx 4.61$ uit moeten komen. We kregen er echter 5.72 uit. Dat betekent dus dat onze zoekboom niet optimaal is en dat er dus gemiddeld meer knopen per expansie van een knoop ontstaan bij onze zoekboom dan in het optimale geval. Ook is nu duidelijk dat negamax èrg ver af ligt van de optimale waarde, hetgeen zich vertaalt in veel langzamere uitvoertijden in vergelijking met alpha-beta pruning.

7 Conclusie

We kunnen concluderen dat onze implementaties van negamax en alpha-beta pruning juist en snel werken. Ook is de evaluatiefunctie voldoende. De spelconfiguraties worden op een logische manier binnen korte tijd beoordeeld. De experimenten tonen aan dat alpha-beta pruning een zeer grote verbetering is ten opzichte van negamax, maar dat het de optimaliteit niet helemaal benadert. Dat is ook niet verwonderlijk, want de optimale situatie wordt in de praktijk nooit bereikt. Er zou immers niet gezocht hoeven worden als de optimale ordening al bekend is. Het hangt dus af van de ordening van de zoekboom hoeveel knopen er bezocht moeten worden en niet van de zoekalgoritmen zelf. Wel kan het zijn dat er minder gesnoeid wordt door de waarde die de evaluatiefunctie teruggeeft. Het is dan ook waarschijnlijk dat de kwaliteit van de evaluatiefunctie een groot deel van het verschil met de theoretische vertakkingsfactor veroorzaakt in onze implementatie.

Deze imperfecties bieden nog ruimte voor verbetering en zouden in een volgende versie aan de kaak gesteld kunnen worden door meer geavanceerde technieken (zoals null-move pruning of transposition tables) in te bouwen en door meer controles aan de evaluatiefunctie toe te voegen. Hierdoor wordt de evaluatiefunctie slimmer en is de kans groot dat er meer gesnoeid kan worden in de boom. Een trade-off hierbij is echter dat meer controles de complexiteit en de uitvoersnelheid van de evaluatiefunctie negatief kunnen beïnvloeden.

Referenties

[1] W.A. Kosters, *Derde programmeeropdracht voorjaar 2013*, 4 april 2013, http://www.liacs.nl/~kosters/AI/jungle2013.html

[2] J.K. Vis, *Jungle-framework*, 4 april 2013, http://www.liacs.nl/~kosters/AI/jungle.tgz

[3] J.K. Vis, *Spelregels Jungle*, 4 april 2013, http://www.liacs.nl/~kosters/AI/jungle.pdf

[4] Wikipedia, *Negamax*, 15 april 2013, https://en.wikipedia.org/wiki/Negamax

[5] Wikipedia, *Alpha-beta pruning*, 15 april 2013, https://en.wikipedia.org/wiki/Alpha-beta_pruning

[6] Chess Programming Wiki, *Branching factor*, 21 april 2013, http://chessprogramming.wikispaces.com/Branching+Factor

[7] Wikipedia, *Meetkundig gemiddelde*, 22 april 2013, https://nl.wikipedia.org/wiki/Meetkundig_gemiddelde

Appendix

Voor deze opdracht bouwen we voort op een reeds bestaand programma dat ons verstrekt is door Jonathan Vis (zie [2]).

main.cc

```
1  // *****************************************************************
2  //   (C) Copyright 2013 Leiden Institute of Advanced Computer Science
3  //   Universiteit Leiden
4  //   All Rights Reserved
5  // *****************************************************************
6  // Kunstmatige Intelligentie --- Jungle
7  // *****************************************************************
8  // FILE INFORMATION:
9  //   File:     main.cc (Depends on: move.h,
10 //                                  move_list.h,
11 //                                  position.h,
12 //                                  search.h,
13 //                                  types.h)
14 //   Author:   Jonathan K. Vis
15 //   Revision: 0.03a
16 //   Date:     2013/01/28
17 // *****************************************************************
18 // DESCRIPTION:
19 //   Hier begint het programma. De gebruiker kan tegen de AI spelen.
20 //   Let op: stdout wordt gebruikt voor message passing tussen twee
21 //   AIs (zie: contest.sh). Gebruik voor communicatie met de gebruiker
22 //   stdlog. En om te debuggen stderr (vaak een goed idee).
23 //   gebruik: ./(executable) white/black depth
24 //   waarbij white of black aangeeft met welke kleur de AI speelt en
25 //   depth aangeeft hoe diep (in plies) de AI moet zoeken.
26 //   2013/01/25 CHANGED: er kan nu zowel als gebruiker tegen de AI
27 //                       worden gespeeld als twee AIs tegen elkaar
28 //                       mbv een named pipe (zie: contest.sh).
29 //   2013/01/28 ADDED: get_move functie om twee gebruikers tegen
30 //                     elkaar te laten spelen.
31 // *****************************************************************
32
33 #include "move.h"
34 #include "move_list.h"
35 #include "position.h"
36 #include "search.h"
37 #include "types.h"
38
39 #include <cstdlib>
40 #include <ctime>
41 #include <climits>
42 #include <iostream>
43 using namespace std;
44
45 // Deze variabele aanpassen naar een naam naar keuze
46 static char const* const ENGINE_NAME = "MyJungleChessEngine v1.0";
47
48 bool parse_args(int const         argc,
49                 char const* const argv[],
50                 bool             &my_turn,
51                 int              &depth)
52 {
53   if (argc < 3)
54   {
55     cerr << "Usage: " << argv[0] << " white/black depth" << endl;
56     return false;
57   } // if
58   if (argv[1][0] == 'b' || argv[1][0] == 'B')
59   {
60     my_turn = false;
61   } // if
62   else if (argv[1][0] == 'w' || argv[1][0] == 'W')
63   {
64     my_turn = true;
65   } // if
66   else
67   {
68     cerr << "Error in first argument: expected 'white' or 'black'"
69          << endl;
70     return false;
71   } // else
72   depth = atoi(argv[2]);
73   if (depth <= 0)
74   {
75     cerr << "Error in second argument: depth is expected to be an "
76          << "integer > 0" << endl;
77     return false;
78   } // if
79   return true;
80 } // parse_args
81
82 Move get_move(void)
83 {
84   Move move(NONE, CAPTURED, CAPTURED, NONE);
85   cin >> move;
86   cin.ignore(1024, '\n');
87   return move;
88 } // get_move
89
90 void random_move(Position &position,
91                  int const depth,
92                  Move     &move)
```

```
93  {
94    Move_list move_list;
95    static_cast<void>(depth); // ignore depth
96    position.generate_moves(move_list);
97    move = move_list.move(rand() % move_list.size());
98  } // random_move
99
100 void play(bool my_turn, int const depth)
101 {
102   Position position;
103   Move move;
104   position.initial();
105   while (!position.is_won() &&
106          !position.is_threefold_repetition() &&
107          !position.no_more_moves() &&
108          !position.last_turn())
109   {
110     if (my_turn)
111     {
112       move = Move(NONE, CAPTURED, CAPTURED, NONE);
113
114       // Change the 'alphabeta' call to one of the
115       // commented calls below to test those features:
116       //   random_move(position, depth, move);
117       //   negamax(position, depth, move);
118       alphabeta(position, depth, -INT_MAX, INT_MAX, move);
119       cout << (position.is_white_turn() ? "white/" : "black/")
120            << position.turn() / 2 + 1 << "> ";
121       cout << move << endl;
122     } // if
123     else
124     {
125       move = Move(NONE, CAPTURED, CAPTURED, NONE);
126       while (!move.is_valid() || !position.validate_move(move))
127       {
128         clog << endl;
129         position.draw_ASCII(clog);
130         clog << (position.is_white_turn() ? "white/" : "black/")
131              << position.turn() / 2 + 1 << "> ";
132         move = get_move();
133       } // while
134     } // else
135     position.do_move(move);
136     my_turn = !my_turn;
137   } // while
138   position.draw_ASCII(clog);
139   clog << "Game ended in a ";
140   if (position.is_won())
141   {
142     if (position.is_white_turn())
143     {
144       clog << "black win." << endl;
145       cout << "white/lost> 0-1" << endl;
146       cout << "black/win> 0-1" << endl;
147     } // if
148     else
149     {
150       clog << "white win." << endl;
151       cout << "white/win> 1-0" << endl;
152       cout << "black/lost> 1-0" << endl;
153     } // else
154   } // if
155   else
156   {
157     clog << "draw." << endl;
158     cout << "white/draw> 1/2-1/2" << endl;
159     cout << "black/draw> 1/2-1/2" << endl;
160   } // if
161   return;
162 } // play
163
164
165 int main(int argc, char* argv[])
166 {
167   bool my_turn;
168   int depth;
169   srand(static_cast<unsigned int>(time(0)));
170   if (!parse_args(argc, argv, my_turn, depth))
171   {
172     return 1;
173   } // if
174   clog << "Engine '" << ENGINE_NAME << "' started as "
175        << (my_turn ? "white." : "black.") << endl;
176
177   cin.rdbuf()->pubsetbuf(0, 0);
178   clog.rdbuf()->pubsetbuf(0, 0);
179   cout.rdbuf()->pubsetbuf(0, 0);
180
181   play(my_turn, depth);
182   return 0;
183 } // main
```

evaluation.cc

```
1  // ***********************************************************************
2  //    (C) Copyright 2013 Leiden Institute of Advanced Computer Science
3  //    Universiteit Leiden
4  //    All Rights Reserved
5  // ***********************************************************************
6  // Kunstmatige Intelligentie --- Jungle
7  // ***********************************************************************
8  // FILE INFORMATION:
9  //    File:     evaluation.cc (Depends on: position.h)
10 //    Author:   Tim van der Meij (1115731) and Simon Klaver (1140760)
11 //    Revision: 1.0
12 //    Date:     2013/04/15
13 // ***********************************************************************
14 // DESCRIPTION:
15 //    Implementatie van de evaluatiefunctie. Deze functie moet een
16 //    geheel getal opleveren op basis van de positie op het bord.
17 //    De evaluatie wordt vanuit het standpunt van de witte speler
18 //    bekeken: een groter (positief) getal geeft een voordeel voor wit.
19 //    Gebruikelijk is dat deze functie symmetrisch is rond 0. Dat wil
20 //    zeggen, indien een positie wordt gewaardeerd voor wit met x,
21 //    wordt deze positie voor zwart met -x gewaardeerd.
22 //    Een aantal factoren kunnen in deze evaluatiefunctie worden
23 //    opgenomen, waarbij direct valt te denken aan de relatieve sterkte
24 //    van de zich nog op het bord bevindende stukken en hun locaties
25 //    op het bord. Bijvoorbeeld een stuk dichtbij de 'den' van de
26 //    tegenstander lijkt gunstig.
27 // ***********************************************************************
28
29 #include "position.h"
30 #include <cmath>
31
32 // ***********************************************************************
33 // MEMBER evaluate: Evaluates a given board.
34 // ***********************************************************************
35 int Position::evaluate() const {
36     uint32_t piece;
37     int sumWhite = 0, sumBlack = 0, distance, minDistanceWhite = 100,
38         minDistanceBlack = 100, i, numPiecesWhite = 0, numPiecesBlack = 0, den;
39
40     if(is_won()) {
41         return 10000;
42     }
43     for(i = 0; i < BOARD_SIZE; i++) {
44         piece = _board[i];
45         if(piece != NONE) {
46             // Calculate the total value of the pieces on the board, the total
47             // number of pieces for each player and the minimal distance to the
48             // den. All these criteria will have an influence on the final result.
49             if(is_white(piece)) {
50                 numPiecesWhite++;
51                 sumWhite += strength(piece);
52                 den = (int)LOCATION_BLACK_DEN;
53                 // Calculation necessary since we do not have (x,y) coordinates,
54                 // but instead the number of a square on the board.
55                 distance = (std::abs(i - den) / 7) + ((int)std::abs(i - den) % 7);
56                 if(distance < minDistanceWhite) {
57                     minDistanceWhite = distance;
58                 }
59             } else {
60                 numPiecesBlack++;
61                 sumBlack += strength(piece);
62                 den = (int)LOCATION_WHITE_DEN;
63                 distance = (std::abs(i - den) / 7) + ((int)std::abs(i - den) % 7);
64                 if(distance < minDistanceBlack) {
65                     minDistanceBlack = distance;
66                 }
67             }
68         }
69     }
70     // The board is 7x9, so a piece in the bottom left corner of the
71     // board can traverse at most (4-1) + (9-1) = 11 squares to get
72     // to the den at the top center of the board.
73     if(is_white_turn()) {
74         return (sumWhite - sumBlack) + ((11 - minDistanceWhite) * numPiecesWhite)
75                - ((11 - minDistanceBlack) * numPiecesBlack);
76     }
77     return (sumBlack - sumWhite) + ((11 - minDistanceBlack) * numPiecesBlack)
78            - ((11 - minDistanceWhite) * numPiecesWhite);
79 }
```

search.cc

```
1  // ***********************************************************************
2  //    (C) Copyright 2013 Leiden Institute of Advanced Computer Science
3  //    Universiteit Leiden
4  //    All Rights Reserved
5  // ***********************************************************************
6  // Kunstmatige Intelligentie --- Jungle
7  // ***********************************************************************
8  // FILE INFORMATION:
9  //    File:     search.cc (Depends on: search.h)
10 //    Author:   Tim van der Meij (1115731) and Simon Klaver (1140760)
11 //    Revision: 1.0
```

```
12  //   Date:     2013/04/15
13  // ********************************************************************
14  // DESCRIPTION:
15  //   Implementatie van de zoekalgoritmen.
16  //   2013/01/28 ADDED: variabelen voor het bijhouden van het aantal
17  //                     bezochte knopen.
18  // ********************************************************************
19
20  #include "search.h"
21  #include <climits>
22
23  uint64_t node_count = 0;
24  uint64_t leaf_count = 0;
25
26  // ********************************************************************
27  // MEMBER negamax: Implementation of the negamax algorithm.
28  // Used to determine the best move for the current player.
29  // ********************************************************************
30  int negamax(Position &position, int const depth, Move &move) {
31      int maxValue = -INT_MAX, i, value;
32      Move_list moveList;
33      Move bestMove, doMove, undoMove;
34
35      if(depth == 0 || position.is_won() || position.last_turn() ||
36         position.no_more_moves() || position.is_threefold_repetition()) {
37          return position.evaluate();
38      }
39      position.generate_moves(moveList);
40      for(i = 0; i < moveList.size(); i++) {
41          doMove = moveList.move(i);
42          position.do_move(doMove);
43          // We need the next line to undo the move from above,
44          // because the recursive 'negamax' call can change the
45          // move argument.
46          undoMove = doMove;
47          value = -negamax(position, depth-1, doMove);
48          if(value > maxValue) {
49              maxValue = value;
50              bestMove = undoMove;
51          }
52          position.undo_move(undoMove);
53      }
54      move = bestMove;
55      return maxValue;
56  }
57
58  // ********************************************************************
59  // MEMBER alphabeta: Implementation of the alpha-beta pruning algorithm.
60  // Used to determine the best move for the current player faster than
61  // negamax by pruning search tree branches if possible.
62  // ********************************************************************
63  int alphabeta(Position &position, int depth, int alpha,
64                int beta, Move &move) {
65      int maxValue = -INT_MAX, i, value;
66      Move_list moveList;
67      Move bestMove, doMove, undoMove;
68
69      if(depth == 0 || position.is_won() || position.last_turn() ||
70         position.no_more_moves() || position.is_threefold_repetition()) {
71          return position.evaluate();
72      }
73      position.generate_moves(moveList);
74      for(i = 0; i < moveList.size(); i++) {
75          doMove = moveList.move(i);
76          position.do_move(doMove);
77          undoMove = doMove;
78          value = -alphabeta(position, depth-1, -beta, -alpha, doMove);
79          if(value > maxValue) {
80              maxValue = value;
81              move = bestMove = undoMove;
82          }
83          position.undo_move(undoMove);
84          if(value >= beta) {
85              return value;
86          } else if(value > alpha) {
87              alpha = value;
88          }
89      }
90      return maxValue;
91  }
```

search.h

```
1   // ********************************************************************
2   //   (C) Copyright 2013 Leiden Institute of Advanced Computer Science
3   //   Universiteit Leiden
4   //   All Rights Reserved
5   // ********************************************************************
6   // Kunstmatige Intelligentie --- Jungle
7   // ********************************************************************
8   // FILE INFORMATION:
9   //   File:     search.h (Implementation file: search.cc)
10  //   Author:   Tim van der Meij (1115731) and Simon Klaver (1140760)
```

```
11  //    Revision: 1.0
12  //    Date:     2013/04/15
13  // ********************************************************************
14  // DESCRIPTION:
15  //    Definitie van de zoekalgoritmen: negamax en alpha-beta. Deze
16  //    moeten in het bestand 'search.cc' worden ge\"implementeerd.
17  //    2013/01/28 ADDED: variabelen voor het bijhouden van het aantal
18  //                      bezochte knopen.
19  // ********************************************************************
20
21  #ifndef __search_h__
22  #define __search_h__
23
24  #include "move.h"
25  #include "move_list.h"
26  #include "position.h"
27
28  #include <cassert>
29  #include <cstdlib>
30  #include <iostream>
31  using namespace std;
32
33  // Gebruik deze variabelen om het aantal bezochte knopen (en bladeren)
34  // bij te houden.
35  extern uint64_t node_count;
36  extern uint64_t leaf_count;
37
38  int negamax(Position &position, int const depth, Move &move);
39
40  int alphabeta(Position &position, int const depth, int alpha,
41                int beta, Move &move);
42
43  #endif // __search_h__
```