FACULTEIT
WETENSCHAPPEN

BACHELOR OF SCIENCE IN INFORMATICA

2020-2021

LOGISCH PROGRAMMEREN
SCHAAKCOMPUTER IN PROLOG

Freya Van Speybroeck

UNIVERSITEIT
GENT

1 Inleiding

In dit verslag zal besproken worden hoe we een schaakcomputer hebben gemaakt a.d.h.v Prolog. Hiervoor gebruiken we een spelboom, waarop we het minimax algoritme toepassen samen met alpha-beta snoeien om tot een bepaalde diepte te zoeken naar de best mogelijke zet. Gegeven een bepaalde toestand van een schaakbord, zal het prolog programma dus de volgende toestand teruggeven, waarbij de best mogelijke zet wordt weergegeven voor de speler die aan de beurt was.

2 Interne Bord Voorstelling

2.1 Het bord en zijn stukken

Voor de voorstelling van het bord zijn er twee mogelijkheden. Als eerste zou het bord als een lijst van stukken kunnen voorgesteld worden. Een andere optie is om het bord als een matrix, voorgesteld als een compound in prolog, voor te stellen. Dit zijn een aantal voordelen van het bord als een lijst implementeren:

- Het aantal stukken naarmate het spel vordert wordt telkens minder, dus de lijst zal kleiner worden.
- Lijsten kunnen makkelijk gemanipuleerd worden. (In vergelijking met het gebruiken van een compound voor de matrix voorstelling).
- Er moeten telkens maar maximaal 32 stukken overlopen worden, bij een matrix zouden we misschien bij elke zet de volle 64 vakken moeten overlopen.

Er zijn natuurlijk ook enkele nadelen:

- Werken met een compound is sneller dan werken met een lijst, aangezien we aan de componenten kunnen in constante tijd.
- Een lijst daarentegen is een compound, in een compound, enz.

Met deze overwegingen in gedachten, werd er beslist om met een lijst te werken. Zo een lijst bestaat dan uit een aantal stukken die er als volgt uit kunnen zien:

$$piece(white, knight, 2/1), piece(black, pawn, 3/7), ...$$

Dit bord wordt geïnitialiseerd op 6.2 lijn 24.

Daarbij moet niet enkel het bord en zijn stukken voorgesteld worden, maar is het ook belangrijk om na te denken over hoe we de speciale regels, zoals 'en passant' en rokkades, willen implementeren. Om zo transparant mogelijk deze speciale regels voor te stellen, zullen zij speciale stukken op het bord vormen. Op die manier moet het bord zo weinig mogelijk

rekening houden met logica buiten het bord, en moet enkel de speler als extra info worden gegeven. Zo krijgen we voor een spel een voorstelling als $game(Board, Player)$ en niet iets als $game(Board, Castles, EnPassants, Player)$.

2.2 En Passant

Als een pion net 2 stukken naar voor is gegaan, in plaats van 1, dan loopt deze pion het risico om 'en passant' genomen te worden door een andere pion. Om dit op het bord voor te stellen, plaatsen we op het vak dat is overgeslaan een 'en passant' stuk van de kleur die gekozen is. Zo een stuk ziet er bv. als volgt uit:

$en\_passant(white, 3/4)$

Op die manier kan het bord controleren of een pion een schuin links of schuin rechts mag gaan, afhankelijk van of er een gewoon stuk of een $en\_passant$ stuk staat. Bij het nemen van een $en\_passant$ stuk moet natuurlijk ook de bijhorende pion van het bord verwijderd worden.

2.3 Rokkades

Ook de rokkades worden aan de hand van speciale stukken geimplementeerd. De gebruike stukken worden 'castle blockers' genoemd en zien er bv. zo uit:

$castle\_blocker(white, 3/1)$ - voor de lange rokkade aan de witte kant
$castle\_blocker(black, 7/8)$ - voor de korte rokkade aan de zwarte kant

Deze worden o.a. aangemaakt door $get\_blocker$ op 6.2 lijn 342, waar ook de logica rond de rokkades te vinden is.

Als de $castle\_blocker$ aanwezig is, zijn rokkades naar die kant niet meer mogelijk. Als de koning beweegt van zijn initiële positie, dan worden aan beide kanten de blokkades toegevoegd. Bij het eerste keer bewegen van een toren of bij het verliezen van een toren als hij nog op zijn beginpositie stond, wordt aan de kant van de toren een blokkade toegevoegd. Op die manier is de rokkade beweging enkel mogelijk als er geen $castle\_blocker$ staat op de positie waar de koning zou naar bewegen. Indien de koning tijdelijk geen rokkade kan doen, bv. als een vak dat hij passeert aangevallen wordt, wordt er geen blokkade toegevoegd.

2.4 Bewegen

Om de uitleg simpel te houden, zullen we hier verder uitleggen hoe zetten worden gegenereerd voor een bepaald stuk op het bord. Analoog, maar met meer of minder gegeven parameters, kan dan gecontroleerd worden of een bepaalde zet mogelijk is, of kunnen zetten gegenereerd worden voor alle stukken.

Het genereren van de zetten en de bijhorende nieuwe borden, zal gebeuren door de $move$ methode (6.2 lijn 254). Hiervoor moeten 2 van de 4 parameters gegeven zijn, namelijk het bord en en het stuk dat we willen bewegen. Aangezien het stuk zelf zijn kleur bijhoudt, weten we dus welke speler een stuk wilt verzetten.

Vervolgens zal er gezocht worden naar een mogelijke zet, aan de hand van hoe stukken kunnen bewegen, dit gebeurt door $possible\_move$ (6.3 lijn 5). In het geval van een pion, wordt in een apparte $move$ (6.2 lijn 257) ook nog gezocht naar bewegingen die een pion kan doen om een ander stuk te slaan, aangezien bij pionnen de $possible\_move$ enkel de bewegingen naar voor geeft. De methode $possible\_take$ (6.3 lijn 69) geeft alle mogelijke bewegingen die een stuk kan doen om een ander stuk te slaan. Voor alle stukken behalve een pion is dit dus hetzelfde als $possible\_move$.

Daarna wordt gekeken of de nieuwe positie die we hebben gekregen van $possible\_move$ wel een beweging is die kan gebeuren op het bord. Dit wordt gecontroleerd door $valid\_move$ (6.2 lijn 105). De $valid\_move$ methode zal kijken of de nieuwe positie binnen het bord is, of er geen stuk van dezelfde kleur staat (of bij een pion, geen enkel stuk bij het vooruit bewegen), en of er geen stuk tussen de oude en nieuwe positie staan. Enkel bij een paard hoeft deze laatste regel niet gecontroleerd te worden. $move$ zal daarna indien nodig het genomen stuk van het bord verwijderen.

Daarbij zijn er bij $move$ nog een paar uitzonderingen:

- Als een pion een ander stuk neemt, en we dus een beweging vinden a.d.h.v $possible\_take$, is de controle van $valid\_move$ niet nodig, maar is de built-in $select$ genoeg. Deze zal namelijk false geven als er geen stuk verwijderd kon worden.
- Als een pion een andere pion 'en passant' slaat, moet zowel het $en\_passant$ stuk als de genomen pion van het bord verwijderd worden.
- Een koning heeft ook de rokkade mogelijkheid, die wordt gegeven door $possible\_castle$ (6.2 lijn 325). De $move$ methode (6.2 lijn 277) moet hier de juiste toren vinden waarmee de koning de zet doet, moet dan door $valid\_castle$ (6.2 lijn 130) controleren of de rokkade mogelijk is, en dan uiteindelijk de koning en toren op de juiste posities zetten. De $valid\_castle$ controleert of er geen $castle\_blocker$ aanwezig is voor de betreffende rokkade zet, of de koning niet schaak staat, geen aangevallen vakken passeert, niet schaak zou staan na de rokkade en of er geen stuk is tussen de toren en de koning.

Als laatste roept $move$ de $update\_board$ (6.2 lijn 242) methode op. Deze zal eerst $update\_move$ (6.2 lijn 195) oproepen, die het stuk van zijn oude positie verwijderd en het op de nieuwe positie zet. Afhankelijk van welke zet er net is gebeurd, zal $update\_move$ nog een aantal stukken toevoegen aan het bord of ervan wegnemen. De mogelijke gevallen zijn:

- Een pion die een rij overslaat, moet een $en\_passant$ stuk op het bord zetten.

- Een pion die de overkant bereikt, kan worden ingewisseld voor een ander stuk.
- Een koning die verplaatst van zijn initiële positie, moeten 'castle blockers' op het bord zetten, indien die er nog niet waren.
- Zelfde voor een toren die beweegt van zijn initiële positie.

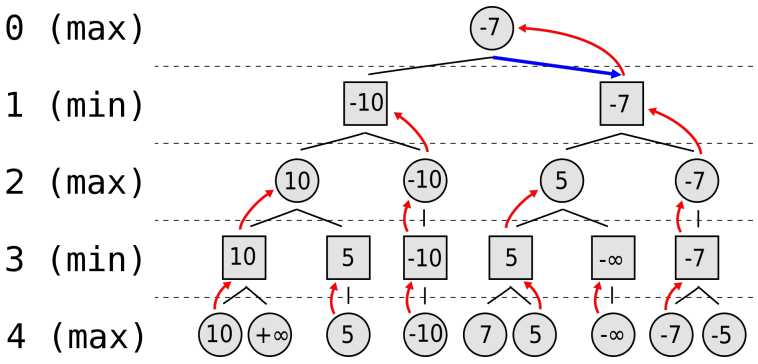
Daarna moet er nog gecontroleerd worden of dit de koning niet schaak zet, en wordt indien nodig een $en\_passant$ stuk van de tegenstander verwijderd. Als een toren genomen werd op zijn initiële positie, moet er ook nog een $castle\_blocker$ toegevoegd worden.

Dit zijn de belangrijkste methoden voor het bewegen van stukken op het bord.

3 Het zoekalgoritme

3.1 Algemene aanpak

Het algoritme om de beste zet te zoeken, maakt gebruik van een zoekboom. Aangezien schaken een 2-player zero-sum game is, kunnen we hiervoor een minimax boom opstellen. De ene speler wilt dus zijn score maximaliseren, terwijl de andere hem wilt minimaliseren. Elke top in de boom stelt een toestand van het spel voor, met een bijhorende score afhankelijk van welke speler aan de beurt is. Zo een boom met scores toegekend aan de toppen kan er dus uit zien zoals hieronder.



Figuur 1: Voorbeeld van een minimax boom

De score van een toestand wordt gebaseerd op de waarden van schaakstukken zoals in echte schaakspellen ook gebruikt wordt, bv. een pion is 1 waard, een paard 3, enz. In deze implementatie, worden deze scores nog eens maal 2 gedaan, om er extra belang aan te hechten. Daarbij wordt ook een hogere score gegeven aan stukken die in het centrum van het bord staan. Bij de koning is dit omgekeerd, die willen we zo ver mogelijk uit het centrum. Er is ook nog een kleine verhoging in score door een stuk te bewegen die nog op zijn initiële positie staat. Als laatste moeten er ook scores aan patstelling of schaakmat gegeven.

Patstelling zal een score van 0 opleveren, aangezien dit gelijkspel is. Schaakmat zal meer opleveren naarmate we minder diep in de zoekboom zitten, zodat er voorkeur wordt gegeven aan snelle schaakmatten. Op diepte 2 zal hij dus bv. 400 opleveren, op diepte 3 maar 200 meer (deze score hangt ook af van tot welke diepte we gaan, in dit voorbeeld gaan we uit van diepte 3).

Aangezien schaken een spel is waarbij heel veel zetten kunnen worden gedaan, zal het vaak zijn dat een top heel veel kinderen (of dus mogelijk volgende toestanden) heeft. Dit betekent dat het minimax algoritme een heel grote boom zal moeten doorzoeken. Om dit te vermijden, zullen we zorgen dat de zoekdiepte beperkt kan worden tot een gegeven diepte. Als die diepte bereikt is, zal de score van de toestand berekend worden, en gepropageerd worden.

Om het algoritme nog te versnellen, kan ook alpha-beta snoeien gebruikt worden. Dit is een veel gebruikte methode die ervoor zorgt dat bepaalde toppen niet meer bezocht moeten worden vanaf we weten dat die geen betere score zal geven dan de scores die we al hebben gevonden tot nu toe. Hiervoor houden we een alpha en een beta bij, die worden gebruikt om eventueel te snoeien en worden aangepast tijdens het doorlopen van de boom.

3.2 Implementatie

Om te werken met een minimax boom, leggen we eerst vast dat wit de maximaliserende speler is, en zwart de minimaliserende. Dit wordt voorgesteld aan de hand van de methodes min_to_move en max_to_move (6.7 lijn 91).

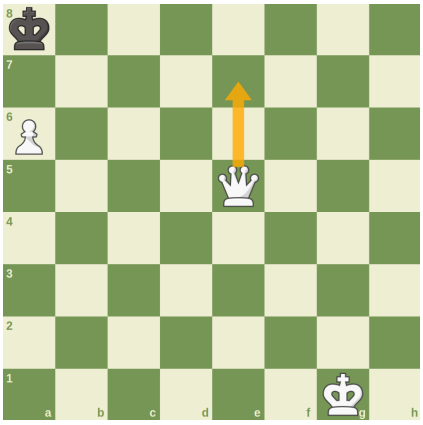
Bij het krijgen van een input bord, wordt get_best_move (6.5 lijn 14) opgeroepen, die $minimax$ (6.6 lijn 6) oproept met een maximale diepte, en een alpha en beta. Alpha moet een lage startwaarde krijgen, beta een hoge.

De methode $minimax$ zal gegeven een $game(Board, Player)$, een diepte en apha-beta waarden, de beste volgende toestand van het spel geven. Dit gebeurt door alle mogelijke volgende toestanden te genereren, en hier uit de beste te vinden met $best$ (6.6 lijn 31). Als echter het aantal mogelijke toestanden gelijk is aan 0, dan kan onmiddelijk de $utility$ (6.7 lijn 4) of score van de toestand worden gegeven. Als de eerste oproep van $minimax$ onmiddelijk 0 volgende toestanden heeft, zal de waarde van de beste volgende toestand gewoon $draw$ zijn.

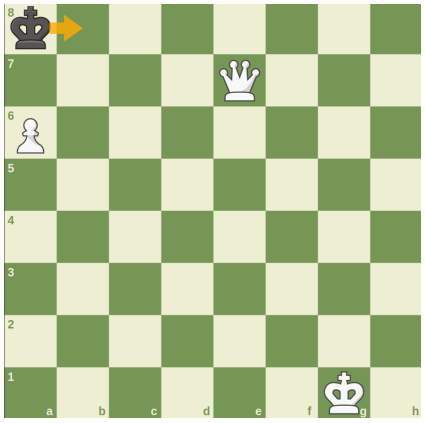
Voordat we $best$ oproepen, verminderen we de meegegeven diepte met 1, zodat bij het bereiken van diepte 0, onmiddelijk de $utility$ wordt teruggeven en er niet verder wordt gezocht in de boom. $best$ zal dan $minimax$ oproepen voor het eerste spel in de lijst, voor dit spel en zijn verkregen score, en voor de rest van de lijst, wordt $best\_pruned$ (6.6 lijn 38) opgeroepen. Deze methode zal de alpha en beta updaten, $best$ oproepen voor de rest van de lijst - met deze nieuwe alpha en beta, en dan het meegegeven spel vergelijken met het

beste spel verkregen uit $best$. De methode $best\_pruned$ zal niet meer recursief $best$ oproepen als er gesnoeid kan worden op basis van de alpha of beta waarde, of als de rest van de spellen een lege lijst is. Er kan gesnoeid worden als we zien dat de min of max speler geen betere score meer zal kunnen halen uit het deel van de boom dat we bezoeken, dit wordt gecontroleerd aan de hand van de alpha en beta waarden. De methode $best\_pruned$ zal dan uiteindelijke het beste spel en zijn score geven, en die dus recursief terug doorgeven naar de eerste oproep van $best$.

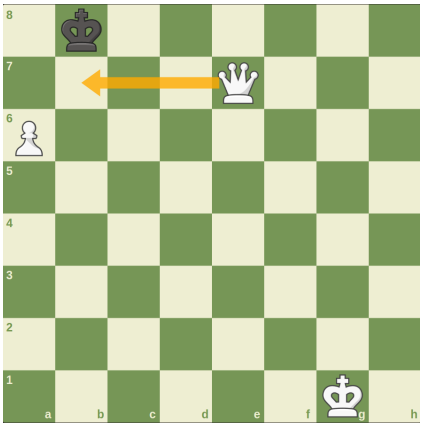
3.3 Voorbeeld



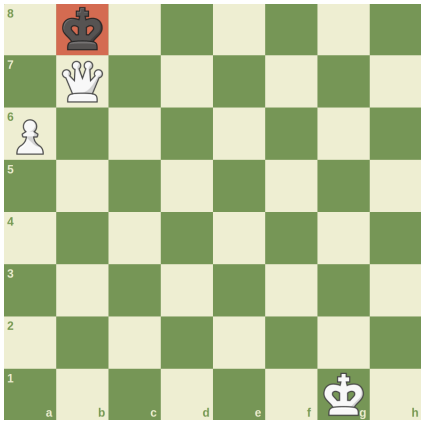
((a)) Eerste zet van wit



((b)) Eerste zet van zwart



((c)) Tweede zet van wit



((d)) Schaakmat

In dit voorbeeld zullen we een schaakmat in 2 zetten bespreken, aangezien de schaakbot op diepte 3 deze zou moeten kunnen vinden.

Het algoritme krijgt dus het bord (a) binnen waarbij de witte speler een zet moet doen, en roept het minimax algoritme op met diepte 3. Alle mogelijke zetten voor wit worden dan

gegenereerd, waaruit we de beste zet moeten vinden. Deze zetten worden gevonden zoals in sectie 2.4 is uitgelegd, voor elk stuk zoeken we dus al zijn bewegingen met behulp van $move$. Het algoritme zal uit deze lijst dan de beste zet proberen vinden, door $best$ op te roepen op de lijst van mogelijke volgende borden.

De schaakbot zal kunnen vinden dat de witte koningin naar e7 zetten, zal leiden tot schaakmat. In de zoekboom zal de top die deze zet voorstelt dus de hoogste score krijgen. Dit komt omdat deze zet schaakmat forceert, aangezien de koning maar 1 mogelijke zet heeft, namelijk naar rechts (b), waarna het schaakmat is in 1 zet. De koningin moet juist nog naar b7 verplaatsen (c) en het spel is gedaan (d).

Andere zetten die overwogen worden zullen niet gegarandeerd leiden tot schaakmat, aangezien ze de koning de kans geven om te ontsnappen, waardoor deze al gesnoeid kunnen worden eens we zien dat er een top is geweest die op z'n minst de schaakmat score heeft.

Het zal dus zo zijn dat voor de eerste zet van wit, het scenario dat zich heeft uitgespeeld dus overlopen wordt in de zoekboom, waardoor we zo een hoge score kregen voor deze zet.

4 Resultaten

We zullen voor de recursiedieptes 1 tot 4 de schaakbot laten spelen tegen de random speler, en kijken hoe die het er tegen opneemt. Dit houdt in dat we o.a. het gemiddelde aantal zetten per spel bekijken, de gemiddelde tijd per zet en hoeveel hij wint of verliest. Het gemiddelde wordt genomen over 20 beurten.

Benchmark resultaten			
Recursiediepte	Seconden per zet	Zetten per spel	Win/Draw/Lose
1	0.25	82.75	14/5/1
2	0.24	46.25	19/1/0
3	3.52	35.85	20/0/0
4	12.92	14.90	20/0/0

Wat er opviel bij de schaakbot op recursie diepte 1, is dat deze snel bij meer zetten komt per spel. Zo kregen we meestal rond de 50 zetten per spel, maar soms ook 170 of meer. Deze zal ook sneller in een lus komen, waardoor er meer dan 1000 zetten worden gedaan, wat we hier niet meerekenen.

De schaakbot op recursiediepte 2, toont geen verschil in tijd met diepte 1, maar we zien wel dat het aantal moves per spel bijna gehalveerd is. Daarbij zal deze ook al minder verliezen (in deze benchmarks verliest die zelfs niet meer).

Op recursiediepte 3 zien we opnieuw een verbetering in het gemiddeld aantal zetten per spel, deze schaakbot zal dus sneller winnen. Daarbij heeft deze schaakbot alle spellen gewonnen, wat weer een verbetering is tegenover het gelijkspel op recursiediepte 2. De gemiddelde tijd per zet stijgt natuurlijk, omdat die nog dieper in de boom moet gaan zoeken.

Ten laatste hebben we nog recursiediepte 4. Deze schaakbot zal opnieuw ervoor zorgen dat het spel in een minder aantal zetten voorbij is. Daarbij wint hij ook altijd. Wat hier opvalt is dat we zouden verwachten dat het aantal seconden per zet veel hoger zou liggen, aangezien het verschil tussen recursiediepte 2 en 3 redelijk groot is. Een verklaring hiervoor kan zijn dat recursiediepte 4 ervoor zorgt dat het spel nooit erg ver geraakt of complex wordt (waardoor de recursieboom veel groter zou worden), aangezien die snel kan winnen.

5 Conclusie

We hebben uiteindelijk een schaakbot kunnen implementeren die al op diepte 3 enkel nog wint van een random speler. Om tot dit eindresultaat te komen, zijn er een aantal beslissingen en ondervindingen geweest die misschien tot een beter of slechter resultaat hebben geleid.

Eerst en vooral was de keuze om 'en passant' en het blokkeren van de rokkades zo transparant mogelijk te doen, achteraf misschien niet het beste idee. Alhoewel het ervoor zorgt dat alle logica op het bord zelf staat, kan het er misschien voor zorgen dat bepaalde zetten iets trager gaan. Deze appart houden kon er bv. voor zorgen dat het controleren op 'en passant' pionnen in 1 stap gevonden werd, in de plaats van over de lijst die het bord voorstelt te moeten lopen. Dit hangt natuurlijk ook af van hoe belangrijk de snelheid is, en hoe diep we moeten kunnen gaan. Tot diepte 4 lijkt snelheid hier nog geen groot probleem te zijn.

Daarnaast was ook de score functie een belangrijk deel van het project. Deze beïnvloedt namelijk welke zetten als goed worden gezien, maar ook het snoeien wordt hierdoor beïnvloedt. Wat hier opviel is dat 1 simpele score functie niet genoeg was om snel te kunnen snoeien. Daarom moeten we die zo variërend mogelijk maken. Een eerste aanzet was om enkel de stukken een score te geven, maar bij nader inzien was er wat extra variatie nodig, zoals bepaalde stukken een hogere score geven naarmate ze verder of dichter in het centrum staan.

Zelfs met het verbeteren van de score functie werkte diepte 3 nog steeds niet snel genoeg. De reden hiervoor was dat bij het controleren van schaakmat, alle mogelijke zetten gegenereerd werden die de koning uit schaakmat halen. Dit is zeer overbodig, aangezien vanaf er 1 zet gevonden is, dit goed genoeg is. Dit was een klein detail met een grote invloed op de snelheid. Door dit op te lossen is de uitvoeringstijd met een factor van 10 verbeterd.

In het algemeen is het dus een goed idee om op voorhand proberen in te zien wat je al-

lemaal nodig zal hebben, en welke implementaties hiervoor het best zullen werken. Zo had misschien het werken met een lijst slecht kunnen aflopen, aangezien die trager zijn dan compounds.

Als laatste is het misschien ook interessant om te bespreken wat de schaakbot nog beter had kunnen maken. Een idee hiervoor was om vanaf er maar een bepaald aantal stukken op het bord staat, de schaakbot dieper te laten zoeken in de boom, aangezien die boom al minder groot is. Alhoewel dit hier dus niet is geïmplementeerd, is het nog iets dat misschien wel een extra verbetering zou kunnen geven.

6 Code

6.1 main.pl

```
:- initialization(main).
:- use_module(input_parser).
:- use_module(game_handler).

% main function of the chess engine
% given an input board, it either outputs all next boards or the best board depending
main:-
    read_string(user_input, _, Str),
    current_prolog_flag(argv, Args),
    init_game(G, Str),
    handle_game(G, Args),
    halt(0).
```

6.2 chess_board.pl

```
:- module(chess_board,
    [   init_board/1,
        opponent/2,
        move/4,
        piece_between/3,
        update_move/4,
        contains/3,
        valid_move/3,
        get_blocker/3,
        castle_side/2,
        color_row/2,
        is_in_checkmate/2,
        is_in_stalemate/2,
        get_result/3,
        get_all_moves/3,
        init_piece/1
    ]).

:- use_module(piece_moves).
:- use_module(utilities).

% -------------------------- initialisation methods ------------------------------

% init_piece(Piece)
```

```
25 % Piece is a piece on an initial chess board, where no pieces have moved yet
26 init_piece(piece(white, rook,   1/1)).
27 init_piece(piece(white, knight, 2/1)).
28 init_piece(piece(white, bishop, 3/1)).
29 init_piece(piece(white, queen,  4/1)).
30 init_piece(piece(white, king,   5/1)).
31 init_piece(piece(white, bishop, 6/1)).
32 init_piece(piece(white, knight, 7/1)).
33 init_piece(piece(white, rook,   8/1)).
34 init_piece(piece(white, pawn,   X/2)) :-
35     between(1, 8, X).
36
37 init_piece(piece(black, rook,   1/8)).
38 init_piece(piece(black, knight, 2/8)).
39 init_piece(piece(black, bishop, 3/8)).
40 init_piece(piece(black, queen,  4/8)).
41 init_piece(piece(black, king,   5/8)).
42 init_piece(piece(black, bishop, 6/8)).
43 init_piece(piece(black, knight, 7/8)).
44 init_piece(piece(black, rook,   8/8)).
45 init_piece(piece(black, pawn,   X/7)) :-
46     between(1, 8, X).
47
48 % init_board(Board)
49 % Board represents an initial chess board, where pieces have not moved yet
50 init_board(Board) :- findall(P, init_piece(P), Board).
51
52 % ------------------------------------ move control/check methods ----------------
53
54 % inside_board(X/Y)
55 % True if X (the column) and Y (the row) are inside the chess board
56 inside_board(X/Y) :-
57     X < 9,
58     X > 0,
59     Y < 9,
60     Y > 0.
61
62 % piece_between(Board, Position, OtherPosition)
63 % Board contains the current pieces on the chess board
64 % True if there is a piece between Position and OtherPosition when Position and Other
65 piece_between(Board, X/Y, X/B) :-
66     !,
67     between_ex(Y, B, TestY),
```

```
68      contains(Board, _, X/TestY),
69      !.
70
71  % piece_between(Board, Position, OtherPosition)
72  % Board contains the current pieces on the chess board
73  % True if there is a piece between Position and OtherPosition when Position and Other
74  piece_between(Board, X/Y, A/Y) :-
75      !,
76      between_ex(X, A, TestX),
77      contains(Board, _, TestX/Y),
78      !.
79
80  % piece_between(Board, Position, OtherPosition)
81  % Board contains the current pieces on the chess board
82  % True if there is a piece between Position and OtherPosition when Position and Other
83  piece_between(Board, X/Y, A/B) :-
84      from_to(X,A,TestX),
85      from_to(Y,B,TestY),
86      (contains(Board, _, TestX/TestY) -> true
87      ;
88      piece_between(Board, TestX/TestY, A/B)).
89
90  from_to(X, A, Test) :- A < X, Test is X-1, A \= Test.
91  from_to(X, A, Test) :- A > X, Test is X+1, A \= Test.
92
93  from_to(Y, B, Test) :- B < Y, Test is Y-1, B \= Test.
94  from_to(Y, B, Test) :- B > Y, Test is Y+1, B \= Test.
95
96  % contains(Board, Color, Position)
97  % True if Board contains a piece having color Color on Position
98  contains(Board, Color, X/Y) :- member(piece(Color, _, X/Y), Board).
99
100 % opponent(Color, OtherColor)
101 % True if Color is the opponent of OtherColor
102 opponent(white, black).
103 opponent(black, white).
104
105 % valid_move(Board, Piece, Position)
106 % True if moving Piece of type pawn to Position is allowed on the chess board represe
107 valid_move(Board, piece(Color, pawn, OldPosition), NewPosition) :-
108     possible_move(piece(Color, pawn, OldPosition), NewPosition),
109     !, % no backtracking when matching on pawn moving forward
110     inside_board(NewPosition),
```

```
111     \+ contains(Board, _, NewPosition),
112     \+ piece_between(Board, OldPosition, NewPosition).
113
114 % valid_move(Board, Piece, Position)
115 % True if moving Piece of type knight to Position is allowed on the chess board repre
116 valid_move(Board, piece(Color, knight, _), NewPosition) :-
117     !, % no backtracking when matching on knight
118     inside_board(NewPosition),
119     \+ contains(Board, Color, NewPosition). % Only opposite colour can be taken
120
121 % valid_move(Board, Piece, Position)
122 % True if moving Piece to Position is allowed on the chess board represented by Board
123 valid_move(Board, piece(Color, _, OldPosition), NewPosition) :-
124     inside_board(NewPosition),
125     \+ contains(Board, Color, NewPosition), % Only opposite colour can be taken
126     \+ piece_between(Board, OldPosition, NewPosition).
127
128 % valid_castle(Board, KingPiece, RookPiece, Position)
129 % True if given a Board, KingPiece can castle to Position, and RookPiece can also rea
130 valid_castle(Board, piece(Color, king, KingPosition), piece(Color, rook, RookPosition
131     \+ member(castle_blocker(Color, NewPosition), Board),
132     \+ piece_between(Board, KingPosition, RookPosition),
133     \+ is_threatened_between(Board, Color, KingPosition, NewPosition).
134
135 % king_is_check(Board, Color)
136 % True if given a Board, king of Color is checked
137 king_is_check(Board, Color) :-
138     member(piece(Color, king, X/Y), Board),
139     is_threatened(Board, piece(Color, king, X/Y)).
140
141 % is_threatened(Board, Piece)
142 % True if a Piece can be taken by the opposite color given a Board
143 is_threatened(Board, piece(Color, _, Position)) :-
144     opponent(Color, Opponent),
145     member(piece(Opponent, OPiece, OPosition), Board),
146     possible_take(piece(Opponent, OPiece, OPosition), Position),
147     valid_move(Board, piece(Opponent, OPiece, OPosition), Position), !.
148
149 % is_threatened_between(Board, Color, Position, OtherPosition)
150 % True if given a Board and Color, either Position or OtherPosition, or some position
151 % helper method for castling, so should only check positions on the same row
152 is_threatened_between(Board, Color, X/Y, A/Y) :-
153     between_in(X, A, R),
```

```
154     TestPiece = piece(Color, pawn, R/Y),
155     is_threatened(Board, TestPiece), !.
156
157 % is_in_checkmate(Board, Color)
158 % True if player of Color is checkmated by opponent given a Board
159 is_in_checkmate(Board, Color) :-
160     king_is_check(Board, Color),
161     findnsols(1, R, get_result(Board, Color, R), ResultBoards), !,
162     \+ length(ResultBoards, 1).
163
164 % is_in_stalemate(Board, Color)
165 % True if player of Color is stalemated by opponent given a Board
166 is_in_stalemate(Board, Color) :-
167     \+ king_is_check(Board, Color),
168     findnsols(1, R, get_result(Board, Color, R), ResultBoards), !,
169     \+ length(ResultBoards, 1).
170
171 % get_all_moves(Board, Player, ResultBoards)
172 % sets ResultBoards to list of all possible Boards given which Player should make a m
173 get_all_moves(Board, Player, Results) :-
174     findall(R, get_result(Board, Player, R), Results).
175
176 % get_result(Board, Player, ResultBoard)
177 % sets ResultBoard to a possible Board given which Player should make a move on Board
178 get_result(Board, Player, Result) :-
179     member(piece(Player, P, X/Y), Board),
180     move(Board, piece(Player, P, X/Y), _, Result).
181
182 % add_castling_blockers(Board, Color, Position, ResultBoard)
183 % sets ResultBoard to Board with castle_blocker for a given Color
184 % only add castle_blocker if Position is an initial rook position
185 add_castling_blockers(Board, Color, X/Y, Result) :-
186     init_piece(piece(Color, rook,X/Y)),!,
187     blocked_column(X, BlockedX),
188     Blocker = castle_blocker(Color, BlockedX/Y),
189     add_if_needed([Blocker], Board, Result).
190
191 add_castling_blockers(Board,_,_,Board).
192
193 % ---------------------------------------- UPDATE THE BOARD BASED ON A MOVE -----
194
195 % update_move(Board, Piece, NewPosition, ResultBoard)
196 % update the Board to ResultBoard, based on a move of Piece to NewPosition
```

```
197 % here: pawn skipping a square generates an en_passant piece on the board
198 update_move(Board, piece(Color, pawn, X/Y), X/B, ResultBoard) :-
199     init_piece(piece(Color,pawn,X/Y)),
200     R is B - Y, Test is abs(R),
201     Test = 2, !,
202     between_ex(Y, B, EnPassantY),
203     select(piece(Color, pawn, X/Y), Board, Board2), % remove old position from the bo
204     ResultBoard = [piece(Color, pawn, X/B), en_passant(Color, X/EnPassantY) | Board2 ]
205
206 % update_move(Board, Piece, NewPosition, ResultBoard)
207 % update the Board to ResultBoard, based on a move of Piece to NewPosition
208 % here: if pawn reaches row of opposite color, it can turn into another piece
209 update_move(Board, piece(Color, pawn, X/Y), A/B, ResultBoard) :-
210     opponent(Color, Opponent),
211     color_row(Opponent, Row),
212     Row = B, !,
213     trade_pawn(NewPiece),
214     select(piece(Color, pawn, X/Y), Board, Board2), % remove old position from the bo
215     ResultBoard = [piece(Color, NewPiece, A/B) | Board2 ]. % add the new position
216
217 % update_move(Board, Piece, NewPosition, ResultBoard)
218 % update the Board to ResultBoard, based on a move of Piece to NewPosition
219 % here: if king moves from its initial position, add castle blockers on the castling
220 update_move(Board, piece(Color, king, X/Y), A/B, ResultBoard) :-
221     init_piece(piece(Color,king,X/Y)), !,
222     findall(Blocker, get_blocker(Color, _ , Blocker), Blockers),
223     select(piece(Color, king, X/Y), Board, Board2),
224     add_if_needed(Blockers, Board2, Board3),
225     ResultBoard = [piece(Color, king, A/B) | Board3 ].
226
227 % update_move(Board, Piece, NewPosition, ResultBoard)
228 % update the Board to ResultBoard, based on a move of Piece to NewPosition
229 % here: if rook moves from its initial position, add castle blockers on the castling
230 update_move(Board, piece(Color, rook, X/Y), A/B, ResultBoard) :-
231     add_castling_blockers(Board, Color, X/Y, Board2),!,
232     select(piece(Color, rook, X/Y), Board2,Board3),
233     ResultBoard = [piece(Color, rook, A/B) | Board3 ].
234
235 % update_move(Board, Piece, NewPosition, ResultBoard)
236 % update the Board to ResultBoard, based on a move of Piece to NewPosition
237 % here: the more general case, move the given piece to a new location without much ex
238 update_move(Board, piece(Color, Piece, X/Y), A/B, ResultBoard) :-
239     select(piece(Color, Piece, X/Y), Board, Board2), % remove old position from the b
```

```
240     ResultBoard = [piece(Color, Piece, A/B) | Board2 ]. % add the new position
241
242 % update_board(Board, Piece, NewPosition, ResultBoard)
243 % update the Board to ResultBoard, based on a move of Piece to NewPosition
244 % then make sure king is not checked, en passant pieces are removed if needed, and/or
245 update_board(Board, piece(Color, Piece, X/Y), A/B, ResultBoard) :-
246     update_move(Board, piece(Color, Piece, X/Y), A/B, Board2),
247     \+ king_is_check(Board2, Color),
248     opponent(Color, Opponent),
249     delete(Board2, en_passant(Opponent, _), Board3), % opponent en passant pieces sho
250     add_castling_blockers(Board3, Opponent, A/B, ResultBoard). %in case tower gets ta
251
252 % ------------------------------------------ MOVE LOGIC -------------------------
253
254 % move(Board, Piece, NewPosition, ResultBoard)
255 % if Piece can go to NewPosition according to the way Piece moves, and it is a valid
256 % here: separate case for pawn takes, no need to check valid move as select will give
257 move(Board, piece(Color, pawn, X/Y), A/B, ResultBoard) :-
258     possible_take(piece(Color, pawn, X/Y), A/B),
259     opponent(Color, Opponent),
260     select(piece(Opponent, _, A/B), Board, Board2), % remove taken piece from the boa
261     update_board(Board2, piece(Color, pawn, X/Y), A/B, ResultBoard).
262
263 % move(Board, Piece, NewPosition, ResultBoard)
264 % if Piece can go to NewPosition according to the way Piece moves, and it is a valid
265 % here: pawn taking en passant
266 move(Board, piece(Color, pawn, X/Y), A/B, ResultBoard) :-
267     possible_take(piece(Color, pawn, X/Y), A/B),
268     opponent(Color, Opponent),
269     select(en_passant(Opponent, A/B), Board, Board2), % take the en_passant piece
270     possible_move(piece(Opponent, pawn, A/B), PX/PY), % get the coordinates of the pi
271     select(piece(Opponent, _, PX/PY), Board2, Board3), % remove taken piece from the
272     update_board(Board3, piece(Color, pawn, X/Y), A/B, ResultBoard).
273
274 % move(Board, Piece, NewPosition, ResultBoard)
275 % if Piece can go to NewPosition according to the way Piece moves, and it is a valid
276 % here: king castling
277 move(Board, piece(Color, king, X/Y), A/B, ResultBoard) :-
278     possible_castle(piece(Color, king, X/Y), A/B),
279     rook_side(A, RookX),
280     Rook = piece(Color, rook, RookX/Y),
281     valid_castle(Board, piece(Color, king, X/Y), Rook, A/B),
282     update_move(Board, piece(Color, king, X/Y), A/B, Board2),
```

```
283     rook_castle(Color, A/B, NewX/NewY),
284     update_board(Board2, Rook, NewX/NewY, ResultBoard).
285 
286 % move(Board, Piece, NewPosition, ResultBoard)
287 % if Piece can go to NewPosition according to the way Piece moves, and it is a valid
288 % here: general case for all pieces besides pawns
289 move(Board, piece(Color, Piece, X/Y), A/B, ResultBoard) :-
290     possible_move(piece(Color, Piece, X/Y), A/B),
291     valid_move(Board, piece(Color, Piece, X/Y), A/B),
292     opponent(Color, Opponent),
293     delete(Board, piece(Opponent, _, A/B), Board2), % delete opponent piece if necess
294     update_board(Board2, piece(Color, Piece, X/Y), A/B, ResultBoard).
295 
296 % ------------------------------------------------ EXTRA HELPER LOGIC ----------------
297 
298 % en_passant_row(Color, Number)
299 % True if Number is the row number of where a pawn could have been moved skipping a s
300 en_passant_row(black, 5).
301 en_passant_row(white, 4).
302 
303 % en_passant_square(Piece, Position)
304 % Given a Piece, sets Position to the coordinates of where an en_passant linked to th
305 en_passant_square(piece(black, pawn, X/Y), X/B) :- B is Y + 1.
306 en_passant_square(piece(white, pawn, X/Y), X/B) :- B is Y - 1.
307 
308 % trade_pawn(PieceType)
309 % True if pawn can be trader for PieceType when pawn reaches opposite side
310 trade_pawn(queen).
311 trade_pawn(knight).
312 trade_pawn(bishop).
313 trade_pawn(rook).
314 
315 % castle_side(Side, ColumnNumber)
316 % True if castling to Side means moving the king to ColumnNumber
317 castle_side(queen, 3).
318 castle_side(king, 7).
319 
320 % color_row(Color, RowNumber)
321 % True if Color's closest row is RowNumber (row where the king starts)
322 color_row(black, 8).
323 color_row(white, 1).
324 
325 % possible_castle(Piece, Position)
```

```
% True if Piece can move to Position, which would be a castling move
possible_castle(piece(Color, king, 5/Y), X/Y) :- color_row(Color, Y), castle_side(_, 

% rook_castle(Color, KingNewPosition, RookNewPosition)
% sets RookNewPosition to the position the rook should take when king castles to King
% Color decides which row the rook is on
rook_castle(Color, 7/Y, 6/Y) :- color_row(Color, Y).
rook_castle(Color, 3/Y, 4/Y) :- color_row(Color, Y).

% rook_side(KingColumn, RookColumn)
% if king castles to KingColumn, we need the rook on RookColumn
rook_side(7,8).
rook_side(3,1).

% get_blocker(Color, Side, Blocker)
% given a Color and castling Side, generate the Blocker needed
get_blocker(Color, Side, Blocker) :-
    castle_side(Side, X),
    color_row(Color, Y),
    Blocker = castle_blocker(Color, X/Y).

% blocked_column(ColumnNumber, BlockedColumnNumber)
% given a ColumnNumber (this shows which side we are on), give which column should be
% castle_blocker can then come on BlockedColumnNumber
blocked_column(X, BlockedX) :- X < 5, castle_side(queen, BlockedX).
blocked_column(X, BlockedX) :- X > 5, castle_side(king, BlockedX).
```

6.3 piece_moves.pl

```
:- module(piece_moves, [possible_move/2, possible_take/2, on_same_diagonal/2]).
:- use_module(chess_board).
:- use_module(utilities).

% possible_move(Piece, NewPosition)
% True if a Piece can do a move to NewPosition
% here Piece is of type rook, and can go horizontal and vertical
possible_move(piece(_, rook, X/Y), A/Y) :-
    between_in(1,8,A),
    A \= X. % piece should not be able to move to the same place

possible_move(piece(_, rook, X/Y), X/B) :-
    between_in(1,8,B),
    B \= Y.
```

```
15
16 % possible_move(Piece, NewPosition)
17 % True if a Piece can do a move to NewPosition
18 % here Piece is of type bishop, and can go along the two diagonals
19 possible_move(piece(_, bishop, X/Y), A/B) :-
20     between_in(1,8,A),
21     between_in(1,8,B),
22     on_same_diagonal(X/Y, A/B),
23     X \= A, Y\= B.
24
25 % possible_move(Piece, NewPosition)
26 % True if a Piece can do a move to NewPosition
27 % here Piece is of type queen, and can do the movements of a bishop or of a rook
28 possible_move(piece(_, queen, X/Y), A/B) :- possible_move(piece(_, bishop, X/Y), A/B).
29 possible_move(piece(_, queen, X/Y), A/B) :- possible_move(piece(_, rook, X/Y), A/B).
30
31 % possible_move(Piece, NewPosition)
32 % True if a Piece can do a move to NewPosition
33 % here Piece is of type king, a king can move one step in each direction
34 possible_move(piece(_, king, X/Y), X/B) :- B is Y - 1.
35 possible_move(piece(_, king, X/Y), X/B) :- B is Y + 1.
36
37 possible_move(piece(_, king, X/Y), A/Y) :- A is X + 1.
38 possible_move(piece(_, king, X/Y), A/Y) :- A is X - 1.
39
40 possible_move(piece(_, king, X/Y), A/B) :- A is X - 1, B is Y - 1.
41 possible_move(piece(_, king, X/Y), A/B) :- A is X + 1, B is Y - 1.
42 possible_move(piece(_, king, X/Y), A/B) :- A is X - 1, B is Y + 1.
43 possible_move(piece(_, king, X/Y), A/B) :- A is X + 1, B is Y + 1.
44
45 % possible_move(Piece, NewPosition)
46 % True if a Piece can do a move to NewPosition
47 % here Piece is of type knight, and can move in the shape of an L
48 possible_move(piece(_, knight, X/Y), A/B)  :- A is X + 1, B is Y + 2.
49 possible_move(piece(_, knight, X/Y), A/B)  :- A is X + 1, B is Y - 2.
50 possible_move(piece(_, knight, X/Y), A/B)  :- A is X - 1, B is Y - 2.
51 possible_move(piece(_, knight, X/Y), A/B)  :- A is X - 1, B is Y + 2.
52 possible_move(piece(_, knight, X/Y), A/B)  :- A is X + 2, B is Y + 1.
53 possible_move(piece(_, knight, X/Y), A/B)  :- A is X + 2, B is Y - 1.
54 possible_move(piece(_, knight, X/Y), A/B)  :- A is X - 2, B is Y + 1.
55 possible_move(piece(_, knight, X/Y), A/B)  :- A is X - 2, B is Y - 1.
56
57 % possible_move(Piece, NewPosition)
```

```
58 % True if a Piece can do a move to NewPosition
59 % here Piece is of type pawn, and can move to the front one step
60 possible_move(piece(white, pawn, X/Y), X/B) :- B is Y+1.
61 possible_move(piece(black, pawn, X/Y), X/B) :- B is Y-1.
62
63 % possible_move(Piece, NewPosition)
64 % True if a Piece can do a move to NewPosition
65 % here Piece is of type pawn, and can move to the front two steps if it's in their in
66 possible_move(piece(white, pawn, X/Y), X/B) :- init_piece(piece(white, pawn, X/Y)), B
67 possible_move(piece(black, pawn, X/Y), X/B) :- init_piece(piece(black, pawn, X/Y)), B
68
69 % possible_take(Piece, NewPosition)
70 % True if a Piece can move to take a piece at NewPosition
71 % here Piece is of type pawn, and can only take sideways
72 possible_take(piece(white, pawn, X/Y), A/B) :- A is X+1, B is Y+1.
73 possible_take(piece(white, pawn, X/Y), A/B) :- A is X-1, B is Y+1.
74
75 possible_take(piece(black, pawn, X/Y), A/B) :- A is X+1, B is Y-1.
76 possible_take(piece(black, pawn, X/Y), A/B) :- A is X-1, B is Y-1.
77
78 % possible_take(Piece, NewPosition)
79 % True if a Piece can move to take a piece at NewPosition
80 % here Piece is of any type besides pawn, and can take any way it can move
81 possible_take(Piece, A/B) :- Piece \= piece(_, pawn, _), possible_move(Piece,A/B).
82
83 % on_same_diagonal(Position, OtherPosition)
84 % True if two squares, Position and OtherPosition, are on the same diagonal
85 on_same_diagonal(X/Y, A/B) :-
86     diagonal(X/Y, Diagonal, Type),
87     diagonal(A/B, OtherDiagonal, Type),
88     Diagonal = OtherDiagonal.
89
90 % diagonal(Position, Diagonal, TypeNumber)
91 % sets Diagonal to the value of the diagonal calculation of a Position
92 % TypeNumber keeps track of which diagonal has been calculated
93 diagonal(X/Y, Diagonal,1) :- Diagonal is X + Y.
94 diagonal(X/Y, Diagonal,2) :- Diagonal is X - Y.
```

6.4 utilities.pl

```
1 :- module(utilities, [add_if_needed/3, between_ex/3, between_in/3]).
2
3 % ------- GENERAL UTILITIES --------
```

```
4
5  % add_if_needed(List, OtherList, ResultList)
6  % same as append, but only append an element of List to OtherList if element is not i
7  % ResultList contains the concatenation of List (not necessarily all elements of List
8  add_if_needed([], R, R) :- !.
9  add_if_needed([X | L], R, Result) :- member(X, R), !, add_if_needed(L, R, Result).
10 add_if_needed([X | L], R, [ X | Result]) :- add_if_needed(L, R, Result).
11
12
13 % between_in(Number1, Number2, Number3)
14 % sets Number3 to a number between Number1 and Number2, including Number1 and Number2
15 between_in(N, M, R) :- between(N,M,R).
16 between_in(N, M, R) :- between(M,N,R).
17
18 % between_ex(Number1, Number2, Number3)
19 % sets Number3 to a number between Number1 and Number2, excluding Number1 and Number2
20 between_ex(N, M, R) :- between(N,M,R), R > N, R < M.
21 between_ex(N, M, R) :- between(M,N,R), N > R, M < R.
```

6.5 game_handler.pl

```
1  :- module(game_handler, [handle_game/2, get_result/3, get_best_move/2]).
2  :- use_module(chess_board).
3  :- use_module(output).
4  :- use_module(minimax).
5
6  % handle_game(Game, ArgumentList)
7  % prints the best game given ArgumentList is empty
8  handle_game(Game, []) :- !, get_best_move(Game, Best), output_game(Best).
9
10 % handle_game(Game, ArgumentList)
11 % prints all possible games given ArgumentList is  not empty (in this case it contain
12 handle_game(Game, _) :- generate_moves(Game).
13
14 % get_best_move(InputGame, BestNextGame)
15 % given InputGame as game(Board, Player), set BestNextGame as game with best possible
16 get_best_move(Game, BestNext) :-
17     minimax(Game, BestNext, _, 3, -1000, 1000), !.
18
19 % generate_moves(Game)
20 % print all possible moves for a given Game
21 generate_moves(game(Board, Player)) :-
22     findall(R, get_result(Board, Player, R), Results),
```

```
23      length(Results, Length),
24      Length > 0, !,
25      opponent(Player, Opponent),
26      print_results(Results, Opponent).
27
28  % generate_moves(Game)
29  % no next moves found for Game, game is drawn
30  generate_moves(_) :- output_game(draw).
31
32  % print_results(BoardList, NextPlayer)
33  % print all games, given all boards in a BoardList and player who is next to move, Ne
34  print_results([], _).
35
36  print_results([R], NextPlayer) :-
37      !,
38      output_game(game(R, NextPlayer)).
39
40  print_results([ R | Results ], NextPlayer) :-
41      output_game(game(R, NextPlayer)),
42      write("~"), nl,
43      print_results(Results, NextPlayer).
```

6.6 minimax.pl

```
1   :- module(minimax, [minimax/6]).
2   :- use_module(chess_board).
3   :- use_module(game_handler).
4   :- use_module(scoring).
5
6   % minimax(InputGame, BestNextGame, Value, Depth, Alpha, Beta)
7   % in this case, we have reached maximum depth (Depth = 0) in the minimax tree, so set
8   % BestNextGame is "draw" for when we have reached this state from the start, else it
9   minimax(Game, draw, Value, 0, _, _) :-
10      !,
11      utility(Game, Value, 0).
12
13  % minimax(InputGame, BestNextGame, Value, Depth, Alpha, Beta)
14  % computes the BestNextGame given an InputGame, where Player is next to move
15  % Value is the value or score of the BestNextGame
16  % Depth is a counter, counting down until 0, representing the depth we have reached i
17  % Alpha and Beta are the min and max values for the alpha-beta pruning
18  minimax(game(Board, Player), BestNext, Value, Depth, Alpha, Beta) :-
19      opponent(Player, Opponent),
```

```
20      NextDepth is Depth - 1,
21      findall(game(R, Opponent), get_result(Board, Player, R), Results),
22      \+ length(Results, 0),!,
23      best(Results, BestNext, Value, NextDepth, Alpha, Beta), !.
24
25  % minimax(InputGame, BestNextGame, Value, Depth, Alpha, Beta)
26  % in this case, Game has no successors, so set its utility as Value
27  % BestNextGame is "draw" for when we have reached this state from the start, else it
28  minimax(Game, draw, Value, Depth, _, _) :-
29      utility(Game, Value, Depth).
30
31  % best(GameList, BestGame, BestValue, Depth, Alpha, Beta)
32  % given a list of games, GameList, compute the BestGame with value BestValue
33  % Depth is de depth counter, Alpha and Beta are the alpha-beta pruning values
34  best([ Game | Games ], BestGame, BestValue, Depth, Alpha, Beta) :-
35      minimax(Game, _, Value, Depth, Alpha, Beta),
36      best_pruned(Games, Game, Value, BestGame, BestValue, Depth, Alpha, Beta).
37
38  % best_pruned(GameList, Game, Value, BestGame, BestValue, Depth, Alpha, Beta)
39  % no more candidates
40  best_pruned([] ,Game, Value, Game, Value,_, _, _) :- !.
41
42  % best_pruned(GameList, Game, Value, BestGame, BestValue, Depth, Alpha, Beta)
43  % either max player reached the upper bound, or min player reached the lower bound
44  best_pruned(_,  Game, Value, Game, Value, _, _, Beta) :-
45      min_to_move(Game), Value > Beta, !.
46
47  best_pruned(_,  Game, Value, Game, Value, _, Alpha, _) :-
48      max_to_move(Game), Value < Alpha, !.
49
50  % best_pruned(GameList, Game, Value, BestGame, BestValue, Depth, Alpha, Beta)
51  % computes the BestGame and its BestValue given a GameList and a Game and its Value
52  % Depth is the depth counter, Alpha and Beta are the alpha-beta pruning values
53  % best_pruned takes Alpha and Beta into consideration, and updates them when needed
54  best_pruned( GameList,  Game, Value, BestGame, BestValue, Depth, Alpha, Beta) :-
55      update_bounds( Alpha, Beta, Game, Value, NewAlpha, NewBeta),
56      best(GameList, OtherGame, OtherValue, Depth, NewAlpha, NewBeta),
57      better_of(Game, Value, OtherGame, OtherValue, BestGame, BestValue).
58
59  % update_bounds(Alpha, Beta, Game, Value, NewAlpha, NewBeta)
60  % update (if necessary) current Alpha and Beta given a Game and its Value, to new val
61  update_bounds(Alpha, Beta, Game, Value, Value, Beta) :-
62      min_to_move(Game), Value > Alpha, !.
```

```
63
64 update_bounds(Alpha, Beta, Game, Value, Alpha, Value):-
65     max_to_move(Game), Value < Beta, !.
66
67 update_bounds(Alpha,Beta, _,_,Alpha, Beta).
68
69 % better_of(Game, Value, OtherGame, OtherValue, BestGame, BestValue)
70 % set BestGame and BestValue as best from two games, namely Game and Value vs. OtherG
71 better_of(Game1, Val1, _, Val2, Game1, Val1) :-
72     min_to_move(Game1),
73     Val1 > Val2, !
74     ;
75     max_to_move(Game1),
76     Val1 < Val2, !.
77
78 better_of(_,_, Game2, Val2, Game2, Val2).
```

6.7 scoring.pl

```
1  :- module(scoring, [utility/3, max_to_move/1, min_to_move/1]).
2  :- use_module(chess_board).
3
4  % utility(Game, Utility)
5  % Game is an input board represented as game(Board, NextPlayer)
6  % NextPlayer is the next player to make a move, the other player just made a move on
7  % utility of a Game calculates the Utility of Game, which represents a score of how g
8  utility(game(Board, NextPlayer), UtilityScore, Depth) :-
9      opponent(NextPlayer, JustPlayed),
10     score(Board, JustPlayed, Utility, Depth),
11     update_score(JustPlayed, Utility, UtilityScore).
12
13
14 % score(Board, Color, Score)
15 % calculates the Score of a Board given a Color to calculate it for
16 % when in end states, give special scores
17 score(Board, Color, 0, _) :- opponent(Color, Opponent), is_in_stalemate(Board,Opponen
18 score(Board, Color, Score, Depth) :- opponent(Color, Opponent), is_in_checkmate(Board
19
20 % score(Board, Color, Score)
21 % calculates the Score of a Board given a Color to calculate it for
22 % when not in an endstate, calculate score based on the placement of the pieces on Bo
23 score(Board, Color, Score, _) :-
24     findall(piece(Color, P, X/Y), member(piece(Color, P,X/Y), Board), Pieces),
```

```
25      board_score(Pieces, BoardScore),
26      center_score(Pieces, CenterScore),
27      moved_score(Pieces, MovedScore),
28      Score1 is BoardScore + CenterScore,
29      Score is Score1 + MovedScore.
30
31  % moved_score(Board, Color, Score)
32  % calculates the Score of Board given a Color, score is based on how many pieces have
33  moved_score(Pieces, Score) :-
34      convlist(piece_moved_score, Pieces, Scores),
35      sum_list(Scores, Score).
36
37  % center_score(Board, Color, Score)
38  % calculates the Score of Board given a Color, score is based on how many pieces are
39  center_score(Pieces, Score) :-
40      convlist(square_score, Pieces, Scores),
41      sum_list(Scores, Score).
42
43  % center_score(Board, Color, Score)
44  % calculates the Score of Board given a Color, score is based on piece scores used in
45  board_score(Pieces, Score) :-
46      convlist(piece_score, Pieces, Scores),
47      sum_list(Scores, Score).
48
49  % piece_score(Piece, Score)
50  % gives the Score of a Piece, Score is a value of a certain piece, based on values us
51  piece_score(piece(_,pawn,_), 2).
52  piece_score(piece(_,knight,_), 4).
53  piece_score(piece(_,bishop,_), 4).
54  piece_score(piece(_,rook,_), 6).
55  piece_score(piece(_,queen,_), 10).
56  piece_score(piece(_,king,_), 0).
57
58  % piece_moved_score(Piece, Score)
59  % gives a Score based on if Piece is still in initial position.
60  % Moving the king or towers should be neutral / worse than other pieces as we want to
61  piece_moved_score(piece(_, king, _), 0):- !.
62  piece_moved_score(piece(_, rook, _), 0):- !.
63
64  % piece_moved_score(Piece, Score)
65  % gives a Score based on if Piece is still in initial position.
66  % Give a negative Score if piece is in its initial position, else Score should be pos
67  piece_moved_score(Piece, -1) :- init_piece(Piece), !.
```

```
68  piece_moved_score(_, 2).
69
70  % square_score(Column/Row, Score)
71  % calculates a Score of a position, represented as Column/Row on the chess board
72  % a square closer to the center of the board gives a higher score, except for the kin
73  square_score(piece(_, king, X/Y), Score) :- !, row_or_column_score(X, RowScore), row_
74  square_score(piece(_, _, X/Y), Score) :- row_or_column_score(X, RowScore), row_or_col
75
76  % row_or_column_score(Number, Score)
77  % Number represents either a row or column of the chess board
78  % given a Number, Score should be higher when row or column is more towards the cente
79  row_or_column_score(1, 0).
80  row_or_column_score(8, 0).
81
82  row_or_column_score(2, 1).
83  row_or_column_score(7, 1).
84
85  row_or_column_score(3, 2).
86  row_or_column_score(6, 2).
87
88  row_or_column_score(4, 3).
89  row_or_column_score(5, 3).
90
91  % max_to_move(Game)
92  % True if white is next to move
93  max_to_move(game(_,white)).
94
95  % min_to_move(Game)
96  % True if black is next to move
97  min_to_move(game(_,black)).
98
99  % update_score(Color, Utility, UpdatedUtility)
100 % Negate the score depending of Color is min or max player
101 update_score(Color, Utility, Utility) :- max_to_move(game(_,Color)), !.
102 update_score(_, Utility, Negated) :- Negated is 0 - Utility.
```

6.8 input_parser.pl

```
1  :- module(input_parser, [init_game/2]).
2  :- use_module(library(pio)).
3  :- use_module(chess_board).
4  :- use_module(mapping).
5
```

```
% ------------------- basic parser functions --------------------------

% parse a number
digit(D) --> [D], { char_type(D, digit) }.

% parse some character that is not a newline or bracket
one_character(D) --> [D], { \+ char_type(D, newline), \+ is_bracket(D) }.

% parse a bracket
bracket(B) --> [B], { is_bracket(B) }.

% True if a character is a bracket
is_bracket(B) :- char_code(']', B).
is_bracket(B) :- char_code('[', B).

% parse whitespace
ws --> [].
ws --> space, ws.

% parse input of type space
space --> [S], { char_type(S, space) }.

% parse an optional non digit, this optional is only needed for the player symbol
optional(black) --> ws.
optional(white) --> one_character(_).

% parse letter(s)
alpha(A) --> [A], { char_type(A, alpha) }.

alphas([L|Ls]) --> alpha(L), alphas(Ls).
alphas([])     --> [].


% ------------------- chess parser functions --------------------------

% parse the last row, "abcdefgh"
last(_) -->
    ws,
    alpha(_),
    alphas(_),
    ws.

% parse the castling and en passant information in [ .. ] given a certain color
```

```
49  info_part(Color, Result) -->
50      bracket(_),
51      { Info = []},
52      castle_piece(Color, queen, Info,Info2),
53      castle_piece(Color, king, Info2, Info3),
54      enpassant_pieces(Color, E),
55      bracket(_),
56      optional(_),
57      { append(E, Info3, Result)}.
58
59  % parse all the pieces on a row of the input board
60  row(Result) -->
61      digit(Number),
62      space,
63      pieces(T, 1/RowNumber),
64      space,
65      info(RowNumber, InfoList),
66      {  char_code(N, Number),
67         atom_number(N, RowNumber),
68         append(InfoList, T, Result)}.
69
70
71  % pass all en passant pawns
72  enpassant_pieces(Color, [P|Ls]) --> enpassant_piece(Color, P), enpassant_pieces(Color
73  enpassant_pieces(_, []) --> [].
74
75  % parse an en passant pawn
76  enpassant_piece(Color, P) -->
77      alpha(Y),
78      digit(X),
79          {
80          char_code(X1,X),
81          char_code(Y1,Y),
82          atom_number(X1, X2),
83          letter_number(Y1, Y2),
84          P = en_passant(Color, X2/Y2) }.
85
86  % parse the info between [ ... ] for a certain color
87  info(1, Result) --> info_part(white, Result).
88  info(8, Result) --> info_part(black, Result).
89  info(_, []) --> ws.
90
91  % parse castling
```

```
92  castle_piece(Color, Side, Info, [ B | Info ]) --> one_character(C), {char_type(C, spa
93  castle_piece(_, _, Info, Info) --> one_character(C), {\+ char_type(C, space)}.
94
95  % parse the pieces with the correct square coordinates (row and column)
96  pieces([P|Ls], X/Y) --> one_character(L), {NewX is X + 1}, pieces(Ls, NewX/Y), { char
97  pieces(Ls, X/Y) --> one_character(L), {NewX is X + 1, char_type(L, space)}, pieces(Ls
98  pieces([],_)     --> [].
99
100 % parse the board
101 read_board([])     --> [].
102 read_board([T|Ts]) --> row(T), ws, read_board(Ts).
103
104 % parse the game
105 read_game(game(R, Player)) --> read_board(T), optional(Player), last(_), {append(T, R
106
107 % init_game(Game, InputString)
108 % set Game to the game read from InputString
109 init_game(Game, Str) :-
110     string_codes(Str,Codes),
111     phrase(read_game(Game), Codes), !.
```

6.9 mapping.pl

```
1   :- module(mapping, [ symbol_to_piece/3, letter_number/2, symbol/3, player/1]).
2
3   % symbol_to_piece( ChessCharacter, Column/Row, Piece)
4   % map between a chess input character ChessCharacter and a Piece
5   % Column/Row is the location of the piece
6   % OPMERKING VOOR VERSLAG: de schaak symbolen in het eerste argument worden niet onder
7   symbol_to_piece(, X/Y, piece(white, pawn, X/Y)).
8   symbol_to_piece(, X/Y, piece(black, pawn, X/Y)).
9
10  symbol_to_piece(, X/Y, piece(white, bishop, X/Y)).
11  symbol_to_piece(, X/Y, piece(black, bishop, X/Y)).
12
13  symbol_to_piece(, X/Y, piece(white, knight, X/Y)).
14  symbol_to_piece(, X/Y, piece(black, knight, X/Y)).
15
16  symbol_to_piece(, X/Y, piece(white, rook, X/Y)).
17  symbol_to_piece(, X/Y, piece(black, rook, X/Y)).
18
19  symbol_to_piece(, X/Y, piece(white, queen, X/Y)).
20  symbol_to_piece(, X/Y, piece(black, queen, X/Y)).
```

```
21
22 symbol_to_piece(_, X/Y, piece(white, king, X/Y)).
23 symbol_to_piece(_, X/Y, piece(black, king, X/Y)).
24
25 % letter_number(Letter, Number)
26 % map between a Letter and a Number of the chess board
27 letter_number('a', 1).
28 letter_number('b', 2).
29 letter_number('c', 3).
30 letter_number('d', 4).
31 letter_number('e', 5).
32 letter_number('f', 6).
33 letter_number('g', 7).
34 letter_number('h', 8).
35
36 % symbol(Color, PieceType, Character)
37 % Character represents the character of a Color and PieceType
38 symbol(white, pawn, "\u2659").
39 symbol(black, pawn, "\u265F").
40
41 symbol(white, bishop, "\u2657").
42 symbol(black, bishop, "\u265D").
43
44 symbol(white, knight, "\u2658").
45 symbol(black, knight, "\u265E").
46
47 symbol(white, rook, "\u2656").
48 symbol(black, rook, "\u265C").
49
50 symbol(white, queen, "\u2655").
51 symbol(black, queen, "\u265B").
52
53 symbol(white, king, "\u2654").
54 symbol(black, king, "\u265A").
55
56 % player(Character)
57 % Character represents the character representing a player's turn
58 player("\u261A").
```

6.10 output.pl

```
1 :- module(output, [ output_game/1 ]).
2 :- use_module(chess_board).
```

```
3  :- use_module(mapping).
4
5  % output_game(Game)
6  % write Game to the terminal
7  output_game(Game) :-
8      write("8 "),
9      output_helper(Game, 1, 8).
10
11 % output_game(Game)
12 % When game is "draw", output should be different from normal output
13 output_game(draw) :- write("DRAW").
14
15 % output_helper(Game, ColumnNumber, RowNumber)
16 % when RowNumber is 0, writing of the game is done, so write a final newline
17 output_helper(_, _, 0) :- !, nl.
18
19 % output_helper(Game, ColumnNumber, RowNumber)
20 % when RowNumber is equal to the row where extra info should be printed and ColumnNum
21 % extract info from Game and write it
22 output_helper(game(Board, Player), 9, Y) :-
23     info_row(Y, Color),
24     !,
25     write(" ["),
26     castle_symbol(Board, Color, queen, Queen),
27     castle_symbol(Board, Color, king, King),
28     write(Queen),
29     write(King),
30     output_enpassant(Board, Color),
31     write("]"),
32     output_player(Player, Color),
33     nl,
34     X = 1,
35     NewY is Y - 1,
36     draw_new_line(NewY),
37     output_helper(game(Board, Player), X, NewY).
38
39 % output_helper(Game, ColumnNumber, RowNumber)
40 % write the piece on the board of Game that's found on ColumnNumber and RowNumber
41 % then continue writing the row
42 output_helper(game(Board, Player), X, Y) :-
43     X < 9,
44     draw(Board, X, Y),
45     NextX is X + 1,
```

```
46      output_helper(game(Board, Player), NextX, Y).
47
48  % output_helper(Game, ColumnNumber, RowNumber)
49  % ColumnNumber is higher than the number of columns on a chess board
50  % reset ColumnNumber and start writing next row
51  output_helper(Game, 9, Y) :-
52      nl,
53      X = 1,
54      NewY is Y - 1,
55      draw_new_line(NewY),
56      output_helper(Game, X, NewY).
57
58  % draw(Board, ColumnNumber, RowNumber)
59  % output the piece that is found on given RowNumber and ColumnNumber
60  draw(Board, X,Y) :-
61      member(piece(Color, Piece, X/Y), Board),
62      !,
63      symbol(Color, Piece, Symbol),
64      write(Symbol).
65
66  % draw(Board, ColumnNumber, RowNumber)
67  % if no piece is found on ColumnNumber and RowNumber, output a space
68  draw(_, _, _) :- write(" ").
69
70  % draw_new_line(RowNumber)
71  % if RowNumber is outside the chess board, output the last line
72  draw_new_line(0) :- !, write("  abcdefgh").
73
74  % draw_new_line(RowNumber)
75  % output the RowNumber and a space for the start of a new line
76  draw_new_line(Y) :-
77      Y =< 8,
78      write(Y),
79      write(" ").
80
81  % output_enpassant(Board, Color) :-
82  % for a given Color, output the en passant pieces on the Board
83  output_enpassant([ Piece | RestBoard ], Color) :-
84      Piece = en_passant(Color, X/Y),!,
85      letter_number(L, X),
86      write(L),
87      write(Y),
88      output_enpassant(RestBoard, Color).
```

```
89
90  % output_enpassant(Board, Color) :-
91  % if head of Board list is not an en passant piece, skip it and check the rest
92  output_enpassant([ _ | RestBoard ], Color) :-
93      output_enpassant(RestBoard, Color).
94
95  % output_enpassant(Board, Color) :-
96  % True if Board is empty
97  output_enpassant([], _).
98
99  % output_player(PlayerColor, InfoColor)
100 % if color of next player, PlayerColor, is the same as the color of the info that nee
101 % output the player symbol
102 output_player(Player, Color) :-
103     Player = Color, !,
104     player(Symbol),
105     write(Symbol).
106
107 % output_player(PlayerColor, InfoColor)
108 % do nothing if PlayerColor and InfoColor are not equal
109 output_player(_, _).
110
111 % info_row(Number, Color)
112 % True if Number is the row where the info of Color is found
113 info_row(1, white).
114 info_row(8, black).
115
116 % castle_symbol(Board, Color, Side, Symbol)
117 % Side represents which side we are castling on, queen or king side
118 % sets Symbol to a space if castling is blocked on the Board, given Color of the play
119 castle_symbol(Board, Color, Side, " ") :-
120     castle_side(Side, X),
121     color_row(Color, Y),
122     member(castle_blocker(Color, X/Y), Board), !.
123
124 % castle_symbol(Board, Color, Side, Symbol)
125 % Side represents which side we are castling on, queen or king side
126 % sets Symbol to a correct symbol if castling is not blocked on the Board, given Colo
127 castle_symbol(_, Color, Side, Symbol) :-
128     symbol(Color, Side, Symbol).
```