UNIVERSITEIT
GENT

Patbot 2000

Lissens Tobiah

2018-06-25

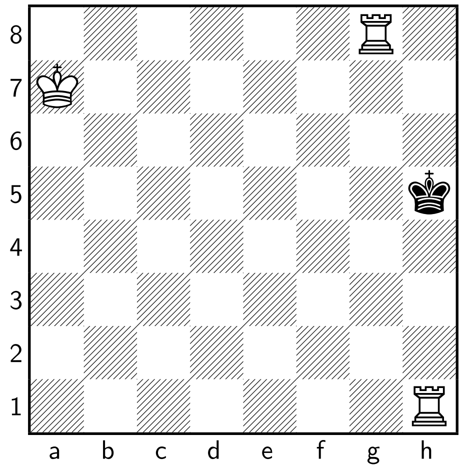
Inhoudsopgave

1 Inleiding

Schaken is een heel bekende denksport. Het evalueren van een schaakpositie behoort echter tot de complexiteitsklasse EXPTIME. Hierdoor is een optimale manier van schaken nog steeds een raadsel. Dit heeft programmeurs echter niet tegengehouden de uitdaging aan te gaan om schaakcomputers te schrijven die beter zijn dan de menselijke meesters. Enkele bekende schaakcomputers zijn Stockfish 9, Houdini6 en Alpha Zero. De bedoeling van dit project is een schaakcomputer te schrijven die gelijkspel kan spelen tegen de schaakcomputer Stockfish 9 met 1 seconde denktijd. Daarom heeft de schaakcomputer de naam Patbot 2000 toegewezen gekregen.

2 Fen Invoer en uitvoer

Het parsen van de FEN[1] gebeurt aan de hand van DCG[2]. Deze manier van parsen is handig omdat je eenvoudig bidirectioneel kan converteren van FEN-string naar een interne representatie en van een interne representatie terug naar FEN.



FEN 6R1/K7/8/7k/8/8/8/7R w KQkq - 10 50

De bovenstaande matconfiguratie wordt intern met de volgende Prologterm voorgesteld.

```
fen_config(
    board(
        row(nil,nil,nil,nil,nil,nil,piece(w,rook),nil),
        row(piece(w,king),nil,nil,nil,nil,nil,nil,nil),
        row(nil,nil,nil,nil,nil,nil,nil,nil),
        row(nil,nil,nil,nil,nil,,nil,piece(b, king)),
        row(nil,nil,nil,nil,nil,nil,nil,nil),
        row(nil,nil,nil,nil,nil,nil,nil,nil),
        row(nil,nil,nil,nil,nil,nil,nil,nil),
        row(nil,nil,nil,nil,nil,nil,nil,piece(w,rook))
    ),w,castle(false,false,false,false),nil,10,50
)
```

De argumenten van fen_config hebben de volgende betekenis.

[1] Forsyth-Edwards Notation
[2] Definite clause grammar

Tabel 1:

1 Bord	Bord bestaande uit 8 row termen bestaande uit 8 nil of piece(Kleur, Type) termen.
2 Kleur	Kleur die op dit moment aan zet is.
3 Rokade	Castle term bestaande uit booleans white kingside, white queenside, black kingside, black queenside respectievelijk.
4 Enpassant	Veld in de voorgaande beurt.
5 Halve-zettenteller	telt het aantal zetten sinds het slaan van een stuk of het verzetten van een pion.
6 Volle-zettenteller	telt het aantal zetten dat zwart heeft gespeelt sinds de start van het spel.

3 Genereren Zetten

Het genereren van zetten gebeurt voor de Loper, Toren, Koningin en Koning allemaal op dezelde manier. Dit gebeurt aan de hand van de `keep_moving_start` regel. Deze kan gevonden worden in file chess_rules.pl(sectie 11.1.4) lijn 88. In deze regel wordt een lijst van richtingen opgezocht behorende tot het stuktype. Er wordt ook een bereik[3] meegegeven. De richtingen die behoren tot de stukken zijn de volgende:

Tabel 2: richtingen

Stuk	Richtingen	Bereik
♗	`-1/1, 1/1, -1/-1, 1/-1`	8
♖	`-1/0, 1/0, 0/-1, 0/-1`	8
♕	♖, ♗	8
♔	♕	1

Verder kan het paard geïmplementeerd worden door de huidige posite met een positie uit de lijst `[-2/-1, -1/-2, 1/-2, 2/-1, -2/1, -1/2, 1/2, 2/1]` op te tellen. De pion bestaat uit heel veel uitzonderingen waarvoor elk een apparte regel is gemaakt deze zijn te vinden in de file chess_rules.pl (sectie 11.1.4) vanaf lijn 130. Verder word het controleren op schaak staan na een zet gedaan door alle volgende borden te genereren en te controleren of de koning van de huidige kleur nog op het veld staat. Deze manier van controleren op schaak is alles behalve efficiënt, maar dit wordt bij het opbouwen van de spelboom bij minimax toch maar uitzonderlijk gebruik van gemaakt.

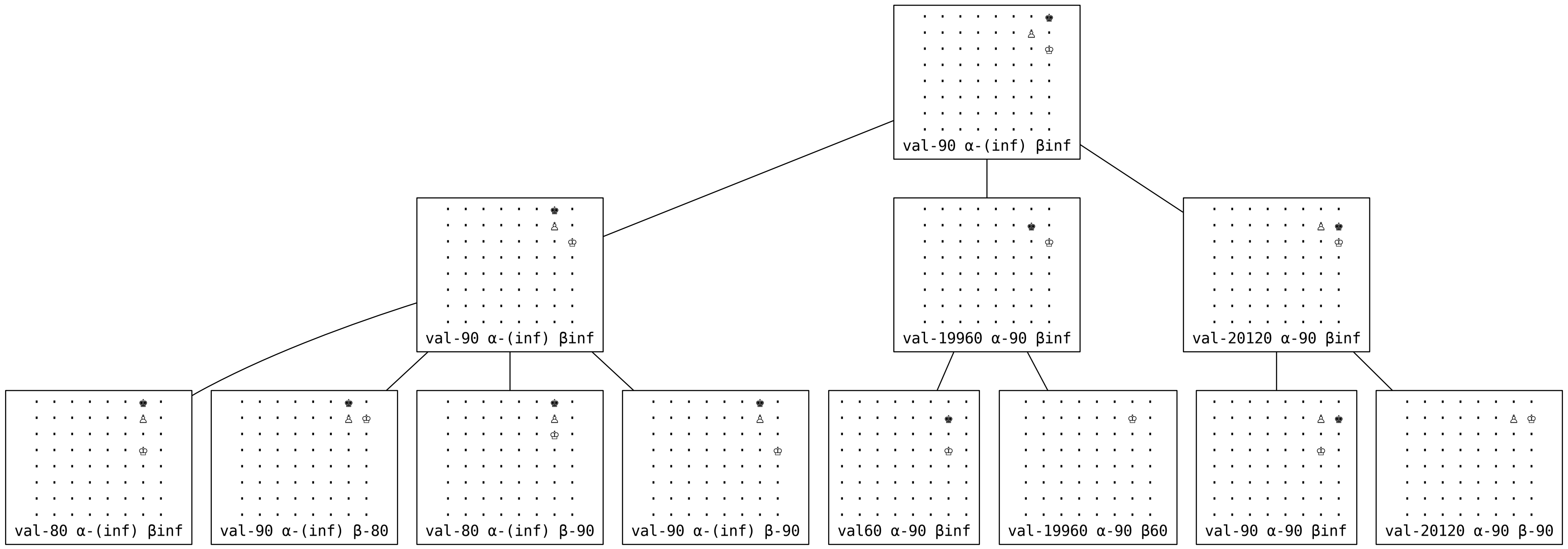
4 Minimax

Een implementatie van het minimax algoritme in Prolog wordt uitgelegd in (*Artificial Intelligence - Implementing Minimax with Prolog*). Het probleem met gewone minimax toepassen bij schaken is dat deze boom veel te groot zal worden. We moeten dit dus een klein beetje aanpassen door de spelboom maar tot een bepaalde diepte d op te bouwen. Wanneer deze diepte d wordt bereikt moeten we een zo goed mogelijke inschatting van de positie kunnen maken. Dit doen we aan de hand van een heuristische evaluatiefunctie die bepaalt hoe goed een bepaald bord voor de huidige speler is. Een voorbeeld van een heel slechte maar eenvoudige evaluatie functie is het $\#stukken\ huidige\ speler - \#stukken\ andere\ speler$.

5 Alpha-Beta

Alpha beta snoeien is een verbetering op het minimax algoritme. Het algoritme werkt door een onder- en bovengrens bij te houden waaraan het beste pad door de spelboom moet voldoen. Tijdens het algoritme worden deze grenzen geleidelijk aangepast en indien we zeker zijn dat een bepaalde deelbomen niet bezocht moeten worden, kunnen we deze overslaan. Het toepassen van het alpha-beta algoritme kan gezien worden in Figuur. 1.

[3] het maximaal aantal vakjes dat ze kunnen opschuiven



Figuur 1: alpha-beta

Het alpha-beta algoritme in Prolog kan gevonden worden in bestand chess_alpha_beta.pl 11.1.6 gebaseerd op (*Programmierkurs Prolog*).

6 Evaluatie

Bij het afkappen van een zoekboom op een zeker diepte moeten we een positie kunnen evalueren. Het kiezen van een goede evaluatie functie is enorm belangrijk. In deze implementatie hebben we gekozen voor de simplified chess evaluatie functie die in (*Simplified evaluation function*) staat beschreven. Het komt er op neer dat elk stuk de waarde zoals in Tabel 3 wordt toegekend. De meeste waarden zijn vanzelfsprekend behalve deze voor het paard en de loper. De meeste schaakboeken kennen deze elk een score van 300 toe, maar om te voorkomen dat deze stukken worden geruild voor 3 pionen. En om er voor te zorgen dat een loper paar meer waard is dan een paarden paar krijgen deze een iets hogere score waarde. Ook wordt de score van de koning in de tabel opgenomen. Hierdoor moeten we tijdens het berekenen van de volgende stukken niet meer expleciet op schaak controleren wat een relatief dure operatie is. De bovenstaande manier van werken wordt semi-legale-zettengeneratie genoemd.

stuk	waarde
♙	100
♘	320
♗	330
♙	500
♕	900
♔	20000

Tabel 3: Waarden van stukken

Verder is bij het evalueren ook de positie van de stukken belangrijk: het aantal velden dat ze bedreigen en het samenhangen van pionnen enzovoort. Hiervoor wordt er gebruik gemaakt van positie tabellen. Dit zijn tabellen die voor elke coördinaat een bonus waarde of penalty toekennen. Zoals we bijvoorbeeld in 11.1.7 lijn 90 kunnen zien, krijgt een paard een grote bonus wanneer hij in het midden staat. Terwijl hij een grote panalty krijgt wanneer hij in de hoek staat. Dit komt doordat een paard in het midden 8 velden kan bereiken terwijl een paard in een hoek er maar 2 kan bereiken.

7 Resultaten

In Tabel.4 zien we Patbot 2000 die zowel tegen stockfish 9 als tegen pychess(bot van pychess) 6 games speelt (3 keer zwart 3 keer wit). Zowel pychess als stockfish kregen alletwee maar 1 seconde denktijd voor een zet. Zoals verwacht presteren zowel stockfish als pychess zelfs met de beperkte denktijd veel beter.

Tabel 4: Resultaten op 5 games

match	win eerste	win tweede	draw
pychess vs Patbot 2000	4	0	2
stockfish 9 vs Patbot 2000	6	0	0
stockfish 9 vs pychess	6	0	0

8 Bespreking

De huidig Patbot heeft echter nog heel wat gebreken. Enkele mogelijke verberteringen zijn de volgende. Patbot kan op dit moment geen onderscheid maken tussen verschillende gamefases zoals het begin, midden of einde. We zouden een heuristiek kunnen schrijven die de huidige gamefase bepaalt door bijvoorbeeld het aantal aanwezige stukken te tellen. Aan de hand van de gamefase zouden we dan onze pos/stuk-score kunnen aanpassen. Een koning in de begin-en middenfase aan de rand is goed, maar in het eindspel is het meestal voordelig deze meer in het spel te betrekken. Verder maakt de huidige schaakcomputer ook nog gebruik van semi-legale-zettengeneratie. Dit wil zeggen dat we bij het genereren van de volgende zetten in de spelboom niet checken of de koning schaak staat. Het probleem hiermee is dat wanneer de schaakbot zijn tegenstander mat wilt zetten hij het verschill tussen mat en pat niet kan opmerken. Door het gebruik van de gamefases zouden we in het eindspel kunnen overschakelen naar legale move generatie. Controleren op schaak staan is een grote kost maar in het eindspel is het aantal mogelijke zetten ook een stuk kleiner. Er zijn echter nog vele andere mogelijke technieken uit de computerschaakwereld die kunnen toegepast worden maar de bovenstaande zijn kleine modificaties aan de huidige Patbot die het programma toch al een stuk beter kunnen maken.

9 Conclusie

Zoals verwacht is het schrijven van een goede schaakcomputer een hele grote uitdaging die buiten de scope van dit vak ligt. Het is dus ook logisch dat deze schaakcomputer niet heel goed presteert. Desalnietemin is deze schaakcomputer een mooi proof of concept en heel handig om verschillende concepten uit de schaakcomputerwereld snel uit te proberen.

10 Bedanking

Graag zou ik Ruben Maes willen bedanken, die een fen2uci wrapper heeft geschreven, hiermee was het eenvoudig de bot te koppelen aan GUI's of andere bots.

Referenties

Michniewski, Tomasz. *Simplified evaluation function.* URL: `https://chessprogramming.wikispaces.com/Simplified+evaluation+function` (bezocht op 18-06-2018).

Picard, Gauthier. *Artificial Intelligence - Implementing Minimax with Prolog.* URL: `https://www.emse.fr/~picard/cours/ai/minimax/` (bezocht op 18-06-2018).

— *Programmierkurs Prolog.* URL: `http://www-ai.cs.uni-dortmund.de/LEHRE/PROLOG/FOLIEN/Folien_Suchprobleme.pdf` (bezocht op 18-06-2018).

11 Appendix Broncode

11.1 Src

11.1.1 main.pl

```prolog
#!/usr/bin/env swipl
:- initialization(main, main).
:- use_module(chess_io).
:- use_module(chess_operations).
:- use_module(chess_rules).
:- use_module(chess_engine).
:- use_module(chess_debug).

main(Args) :-
    length(Args, 6),
    fen_io(Args, Game),
    engine(Game, NewConfig),
    fen_io(Next, NewConfig),
    write(Next).

main(Args) :-
    length(Args, 7),
    append(Arg, ['TEST'], Args),
    fen_io(Arg, Game),
    forall(
        (
            options(Game, NewConfig),
            fen_io(FenString, NewConfig)
        ), (
            write(FenString),
            nl
        )
    ).

% vim: set sw=4 ts=4 ft=prolog et :
```

11.1.2 chess_io.pl

```prolog
:- module(chess_io, [fen_io/2]).
:- set_prolog_flag(double_quotes, chars).
:- use_module(library(dcg/basics)).

/**
* Convert input arguments to a internal prolog term or back.
* @arg In The fen input arguments.
* @arg Out The internal prolog term.
*/
fen_io('DRAW', 'DRAW') :- !.
fen_io(In, Out) :-
    var(Out),
    arg_to_fen(In, Chars),
    phrase(fen(Out), Chars).

fen_io(In, Out) :-
    nonvar(Out),
    phrase(fen(Out), Chars),
    arg_to_fen(In, Chars).

/**
```

```
* Parse FEN using DGC.
*
* Usage phrase(fen_config(X), "FENSTRING").
* The inverse operation and generation are also supported.
*
*/
fen(fen_config(Board, Turn, Castle, Passant, Half, Full)) -->
    board(Board), space,
    turn(Turn), space,
    castle(Castle), space,
    en_passant(Passant), space,
    [ Half ], space, [ Full ].

% Parse the prolog board.
board(board(R1, R2, R3, R4, R5, R6, R7, R8)) --> col(8, [ R8, R7, R6, R5, R4, R3, R2, R1 ]).

% Parse prolog columns.
col(1, [ H ]) --> row(H).
col(L , [ H|T ]) --> {succ(L2, L)}, row(H), forwardslash, col(L2, T).

% Parse a prolog row.
row(row(C1, C2, C3, C4, C5, C6, C7, C8)) --> pieces([ C1, C2, C3, C4, C5, C6, C7, C8 ] - []).

/**
* Parse prolog pieces or empty squares.
* 2 numbers in a FEN representation should never follow eachother!!!.
*/
pieces(Front - Back) --> empty(Front - Back).
pieces(Front - Back) --> empty(Front - Temp1), piece(Temp1 - Temp2), pieces(Temp2 - Back).

/**
* Build an empty list (list containing nil elements).
* @arg Length The length of the list to build.
* @arg List The list containing nil elements.
*/
build_empty(1, [ nil | X ] - X).
build_empty(L1, [ nil | Front ] - Back) :-  L1 < 9, succ(L2, L1), build_empty(L2, Front-Back).

/**
* Parse empty squares.
*/
empty(X - X) --> "".
empty(X) --> [ N ], { nth1(L, "12345678", N), build_empty(L, X) }.

/**
* Parse a single piece.
*/
piece([ P | E ] - E) --> [ C ], { is_piece(C, P) }.

/**
* Parse the turn.
*/
turn(b) --> "b".
turn(w) --> "w".

/**
* Parse column id's
*/
```

```
column_id(Alpha, Num) :- nth1(Num, "abcdefgh", Alpha).

/**
* Parse enpassant options.
*/
en_passant(nil) --> "-".
en_passant(3/C) --> [ Alpha ], "3", {column_id(Alpha, C)}.
en_passant(6/C) --> [ Alpha ], "6", {column_id(Alpha, C)}.

/**
* Parse castling options.
*/
castle(castle(false, false, false, false)) --> "-", !.
castle(castle(WK, WQ, BK, BQ)) -->
    castle('K', WK),
    castle('Q', WQ),
    castle('k', BK),
    castle('q', BQ).

castle(Char, true) --> [Char].
castle(_, false) --> "".

/**
* Find the piece corresponding to a letter.
*
* @arg Char The Char representing the piece.
* @arg Piece The term representing the piece.
*/
is_piece('k', piece(b, king)).
is_piece('q', piece(b, queen)).
is_piece('r', piece(b, rook)).
is_piece('b', piece(b, bishop)).
is_piece('n', piece(b, knight)).
is_piece('p', piece(b, pawn)).
is_piece('K', piece(w, king)).
is_piece('Q', piece(w, queen)).
is_piece('R', piece(w, rook)).
is_piece('B', piece(w, bishop)).
is_piece('N', piece(w, knight)).
is_piece('P', piece(w, pawn)).

forwardslash --> "/".
space --> " ".

% UGLY CONVERSIONS yuk, ieuw, you're fired!
% DO NOT READ THIS !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!.
arg_to_fen([ H, F ], [ HN, ' ', FN ]) :- atom_number(H, HN), atom_number(F, FN).

arg_to_fen([ H|T ], L) :-
    var(L), !,
    atom_chars(H, C),
    arg_to_fen(T, T2),
    append(C, [ ' '| T2 ], L).

arg_to_fen(A, L) :-
    nonvar(L),
    append(L2, [ H, ' ', F ], L),
    atom_chars(Temp, L2),
    atomic_list_concat([H, ' ', F], Temp2),
    atomic_concat(Temp, Temp2, A).

```

```
% vim: set sw=4 ts=4 et :
```

11.1.3 chess_operations.pl

```
:- module(chess_operations, [get_board/2, get_turn/2, get_castle/2, set_castle/3,
   ↪ get_enpassant/2, set_enpassant/3, get_half_count/2, get_full_count/2, set_square/4,
   ↪  get_square/3, is_empty/2, is_mine/2, set_turn/3, other_player/2 , all_coordinates
   ↪ /1, add_positions/3, update_half_count/4, update_full_count/2, set_half_count/3]).


/**
* Get the current board
*
* @arg Config The game configuration.
* @arg Board The board in the configuration.
*/
get_board(fen_config(Board, _, _, _, _, _), Board).

/**
* Set a new board in the config.
*
* @arg Config The current game configuration.
* @arg Board The new board.
* @arg NewConfig The new game configuration.
*/
set_board(fen_config(_, T, C, E, H, F), B, fen_config(B, T, C, E, H, F)).

/**
* Get the turn from the config.
*
* @arg Config The current game configuration.
* @arg Turn The turn in the configuration.
*/
get_turn(fen_config(_, Turn, _, _, _, _), Turn).

/**
* Set the turn in the config.
*
* @arg Config The current game configuration.
* @arg Turn The new Turn.
* @arg NewConfig The new game configuration.
*/
set_turn(fen_config(B, _, C, E, H, F), T, fen_config(B, T, C, E, H, F)).

/**
* Get the castle options from the game configuration.
*
* @arg Config The current game configuration.
* @arg Castle The castle options in the configuration.
*/
get_castle(fen_config(_, _, Castle, _, _, _), Castle).

/**
* Set the castling options in the config.
*
* @arg Config The current game configuration.
* @arg Castle The new Castle options.
* @arg NewConfig The new game configuration.
*/
set_castle(fen_config(B, T, _, E, H, F), C, fen_config(B, T, C, E, H, F)).
```

```

/**
* Get the enpasant options from the config.
*
* @arg Config The current game configuration.
* @arg EnPassant The enpassant options in the configuration.
*/
get_enpassant(fen_config(_, _, _, Passant, _, _), Passant).

/**
* Set the enpassant options in the config.
*
* @arg Config The current game configuration.
* @arg Turn The new Turn.
* @arg NewConfig The new game configuration.
*/
set_enpassant(fen_config(B, T, C, _, H, F), E, fen_config(B, T, C, E, H, F)).

/**
* Get the half count from the config.
*
* @arg Config The current game configuration.
* @arg HalfCount The half count in the configuration.
*/
get_half_count(fen_config(_, _, _, _, Half, _), Half).

/**
* Set the halfcount in the config.
*
* @arg Config The current game configuration.
* @arg HalfCount The new halfcount.
* @arg NewConfig The new game configuration.
*/
set_half_count(fen_config(B, T, C, E, _, F), H, fen_config(B, T, C, E, H, F)).

/**
* Get the full count from the config.
*
* @arg Config The current game configuration.
* @arg FullCount The full count in the configuration.
*/
get_full_count(fen_config(_, _, _, _, _, Full), Full).

/**
* Set the fullcount in the config.
*
* @arg Config The current game configuration.
* @arg FullCount The new Castle options.
* @arg NewConfig The new game configuration.
*/
set_full_count(fen_config(B, T, C, E, H, _), F, fen_config(B, T, C, E, H, F)).

/**
* Set the row in a board.
*
* @arg Board The current boardstate.
* @arg Row The row index.
* @arg Row The new row in a board.
* @arg NewBoard The new boardstate.
*/
```

```
set_row(board(_, _2, _3, _4, _5, _6, _7, _8), 1 , _1, board(_1, _2, _3, _4, _5, _6, _7, _8)).
set_row(board(_1, _, _3, _4, _5, _6, _7, _8), 2 , _2, board(_1, _2, _3, _4, _5, _6, _7, _8)).
set_row(board(_1, _2, _, _4, _5, _6, _7, _8), 3 , _3, board(_1, _2, _3, _4, _5, _6, _7, _8)).
set_row(board(_1, _2, _3, _, _5, _6, _7, _8), 4 , _4, board(_1, _2, _3, _4, _5, _6, _7, _8)).
set_row(board(_1, _2, _3, _4, _, _6, _7, _8), 5 , _5, board(_1, _2, _3, _4, _5, _6, _7, _8)).
set_row(board(_1, _2, _3, _4, _5, _, _7, _8), 6 , _6, board(_1, _2, _3, _4, _5, _6, _7, _8)).
set_row(board(_1, _2, _3, _4, _5, _6, _, _8), 7 , _7, board(_1, _2, _3, _4, _5, _6, _7, _8)).
set_row(board(_1, _2, _3, _4, _5, _6, _7, _), 8 , _8, board(_1, _2, _3, _4, _5, _6, _7, _8)).

/**
* Set a square in a row.
*
* @arg Row The current game configuration.
* @arg Col The column index.
* @arg Square The new square in a row.
* @arg NewRow The new game configuration.
*/
set_col(row(_, _2, _3, _4, _5, _6, _7, _8), 1 , _1, row(_1, _2, _3, _4, _5, _6, _7, _8)).
set_col(row(_1, _, _3, _4, _5, _6, _7, _8), 2 , _2, row(_1, _2, _3, _4, _5, _6, _7, _8)).
set_col(row(_1, _2, _, _4, _5, _6, _7, _8), 3 , _3, row(_1, _2, _3, _4, _5, _6, _7, _8)).
set_col(row(_1, _2, _3, _, _5, _6, _7, _8), 4 , _4, row(_1, _2, _3, _4, _5, _6, _7, _8)).
set_col(row(_1, _2, _3, _4, _, _6, _7, _8), 5 , _5, row(_1, _2, _3, _4, _5, _6, _7, _8)).
set_col(row(_1, _2, _3, _4, _5, _, _7, _8), 6 , _6, row(_1, _2, _3, _4, _5, _6, _7, _8)).
set_col(row(_1, _2, _3, _4, _5, _6, _, _8), 7 , _7, row(_1, _2, _3, _4, _5, _6, _7, _8)).
set_col(row(_1, _2, _3, _4, _5, _6, _7, _), 8 , _8, row(_1, _2, _3, _4, _5, _6, _7, _8)).


/**
* Set a square in a game configuration at a certain position.
*
* @arg Config The current game configuration.
* @arg Position containing row and column index
* @arg Square The new square (either nil or a piece).
* @arg NewConfig The new game configuration.
*/
set_square(Config, R/C, Square, NewConfig) :-
    get_board(Config, Board),
    arg(R, Board, Row),
    set_col(Row, C, Square, NewRow),
    set_row(Board, R, NewRow, NewBoard),
    set_board(Config, NewBoard, NewConfig).

/**
* Get a square in the game configuration.
*
* @arg Config The current game configuration.
* @arg Position containing row and column index
* @arg Square The new square in the game configuration.
*/
get_square(Config, R/C, Square) :-
    get_board(Config, Board),
    arg(R, Board, Row),
    arg(C, Row, Square).
```

```
/**
* Check if the square on a certain position belongs to the current player.
*
* @arg Config The current game configuration.
* @arg Position The position to examine.
*/
is_mine(Config, R/C) :- get_turn(Config, Color), get_square(Config, R/C, piece(Color, _)).

/**
* Check if a square on a certain position is empty (contains nil).
* @arg Config The curreng game configuration.
* @arg Position The position to examine.
*/
is_empty(Config, R/C) :- get_square(Config, R/C, nil).

/**
* The oposite color.
*
* @arg Color The color.
* @arg OpositeColor The oposite color.
*/
other_player(b, w).
other_player(w, b).

/**
* Update the fullcount after black has played a move.
* fullcount = fullcount + 1
*
* @arg Config The game configuration without the fullcount updated.
* @arg NewConfig The configuration with the fullcount updated.
*/
update_full_count(Config, Config) :- get_turn(Config, w).
update_full_count(Config, NewConfig) :-
    get_turn(Config, b),
    get_full_count(Config, Count),
    NewCount is Count + 1,
    set_full_count(Config, NewCount, NewConfig).
/**
* Rest the half count to 0.
*
* @arg Config The current game state.
* @arg NewConfig The game state with the halfcount set to 0.
*/
reset_half_count(Config, NewConfig) :- set_half_count(Config, 0, NewConfig).

/**
* Update the half count.
*
* @arg Config The current game configuration.
* @arg NewConfig The next game configuration.
*/
update_half_count(Config, NewConfig) :-
    get_half_count(Config, Count),
    NewCount is Count + 1,
    set_half_count(Config, NewCount, NewConfig).

/**
* Smart update of the halfcount.
* halfcount = halfcount +1 if no capture or pawn move.
* otherwise halfcount = 0
* @arg Config The old configuration before the move.
* @arg Square1 Old square.
* @arg Square2 New Square.
* @arg NewConfig The configuration with the half count intelligently updated.
*/
update_half_count(Config, Square, nil, NewConfig) :-
    Square \= piece(_, pawn), !,
    update_half_count(Config, NewConfig).

update_half_count(Config, _, _, NewConfig) :-
    reset_half_count(Config, NewConfig).

/**
* Valid coordinates in a chess game.
* Can be used to check or generate.
* R in 1..8
* C in 1..8
*
* @arg Pos The coordinate to check or generate.
*/
all_coordinates(R/C) :- between(1, 8, R), between(1, 8, C).

/**
* Add to coordinates
* @arg Coordinate1 The first coordinate.
* @arg Coordinate2 The second coordinate.
* @arg NewCoordinate The addition of the 2 coordinates.
*/
add_positions(R1/C1, R2/C2, R3/C3) :- R3 is R1 + R2, C3 is C1 + C2.

% vim: set sw=4 ts=4 et :
```

11.1.4 chess_rules.pl

```
:- module(chess_rules, [options/2, options_no_check/2, dir/2, is_not_check/2]).
:- use_module(chess_operations).
:- use_module(chess_debug).


/**
* Generate the next possible configurations and also check if the king is in check.
* @arg Config The current configuration.
* @arg NewConfig The next configuration.
*/
options(Config, NewConfig) :-
    get_turn(Config, Color),
    options_no_check(Config, NewConfig),
    is_not_check(NewConfig, Color).

/**
* Generate the next possible congfigurations but do not check if the king is in check.
*
* @arg Config The current configuration.
* @arg NewConfig The next configuration.
*/
options_no_check(Config, NewConfig) :-
    get_turn(Config, Color), % Get the current turn.
    other_player(Color, Color2), % Get the other player.
    get_square(Config, Pos1, piece(Color, T)), % Iterate over all squares of the current player.
    update_board(Config, Pos1, piece(Color, T), Pos2, Temp0), % update the board.
    get_square(Config, Pos2, Square2), % Get the square of the updated position.
    update_castle(Temp0, Temp3), % Update the castling options
    update_half_count(Temp3, piece(Color, T), Square2, Temp4), % update the halfcount intellently.
    update_full_count(Temp4, Temp5), % Update the full count.
    set_turn(Temp5, Color2, NewConfig). % switch the turns.

/**
* Move a piece to a certain valid position.
*
* @arg Config The current configuration.
* @arg Pos1 The position of the current piece.
* @arg Pos2 The position the piece should move to.
* @arg Piece The piece on Pos1.
* @arg NewConfig The new configuration with the piece moved.
*/
move(Config, Pos1, Pos2, Piece, NewConfig) :-
    all_coordinates(Pos2),
    \+ is_mine(Config, Pos2),
    set_square(Config, Pos1, nil, Temp),
    set_square(Temp, Pos2, Piece, NewConfig).

dir(knight, Pos) :- member(Pos, [(-2)/(-1), (-1)/(-2), 1/(-2), 2/(-1), (-2)/1, (-1)/2, 1/2, 2/1]).
dir(bishop, Pos) :- member(Pos, [(-1)/1, 1/1, (-1)/(-1), 1/(-1)]).
dir(rook, Pos) :- member(Pos, [0/1, 1/0, 0/(-1), (-1)/0]).
dir(queen, Pos) :- dir(bishop, Pos); dir(rook, Pos).
dir(king, Pos) :- dir(queen, Pos).

/**
* Generate all possible moves with there new configuration per piece.
*
* @arg Config The current configuration
* @arg Pos1 The coordinate of the piece to move.
* @arg Piece The piece on the coordinate.
* @arg Pos2 The position that was moved to.
* @arg Config2 The new configuration with the piece moved.
*/

% update pawn movement.
update_board(Config, Pos1, piece(C, pawn), Pos2, Config2) :-
    !,
    (pawn(w, 2, 8, 1/0, [1/1, 1/(-1)], Config, Pos1, C, Pos2, Config2)
    ;
    pawn(b, 7, 1, (-1)/0, [(-1)/1, (-1)/(-1)], Config, Pos1, C, Pos2, Config2)).

% update knight movement
update_board(Config, Pos1, piece(C, knight), Pos3, Config2) :-
    !,
    dir(knight, Pos2),
    add_positions(Pos1, Pos2, Pos3),
    move(Config, Pos1, Pos3, piece(C, knight), Temp),
    set_enpassant(Temp, nil, Config2).

% update king movement.
update_board(Config, Pos1, piece(C, king), Pos2, Config2) :-
    !,
    (movement(Config, 1, piece(C, king), Pos1, Pos2, Config2) % normall 1 step movement.
    ;
    castle_options(Config, Pos1, C, Pos2, Temp), % castling.
    set_enpassant(Temp, nil, Config2)).
```

```prolog

% update bishop rook and queen movement.
update_board(Config, Pos1, Piece, Pos2, Config2) :-
    movement(Config, 8, Piece, Pos1, Pos2, Config2).

movement(_, _, 0, _, []) :- !.
movement(Config, _, _, Pos, [Pos]) :- \+ is_empty(Config, Pos), !.

movement(Config, Dir, Range, Pos1, [ Pos1 | Moves1]) :-
    add_positions(Pos1, Dir, Pos2),
    NewRange is Range - 1,
    movement(Config, Dir, NewRange, Pos2, Moves1).

movement(Config, Range , piece(C, Piece), Pos1, Pos3, NewConfig) :-
    dir(Piece, Dir),
    add_positions(Pos1, Dir, Pos2),
    movement(Config, Dir, Range , Pos2, Options),
    member(Pos3, Options),
    move(Config, Pos1, Pos3, piece(C, Piece), Temp),
    set_enpassant(Temp, nil, NewConfig).

/**
* Generate all possible moves for a pawn in an overengineered fashion :).
*
* Note every pawn rule has a different prolog rule!
* The pawn rules are
* 1) normall
* 2) attack
* 3) skip row
* 4) enpassant
*
* @arg Color The color of the pawn for which these rules apply.
* @arg BaseRow The starting row of the pawn.
* @arg PromoteRow The row at which the pawn promotes.
* @arg MoveDir Direction the pawn moves in.
* @arg TakeDirs List of directions the pawn can capture.
* @arg Config The current configuration
* @arg Pos1 The current position of the pawn.
* @arg Color The color of the pawn.
* @arg Pos2 The new position of the pawn.
* @arg Config2 The new game configuration with the pawn moved.
*/

% Pawn normall movement.
pawn(Color, _, PromoteRow, MoveDir, _ , Config, Pos1, Color, Pos2, Config2) :-
    add_positions(Pos1, MoveDir, Pos2),
    is_empty(Config, Pos2),
    promote(PromoteRow, Pos2, Type),
    move(Config, Pos1, Pos2, piece(Color, Type), Temp),
    set_enpassant(Temp, nil, Config2).

% Pawn attack movement.
pawn(Color, _, PromoteRow, _ , TakeDirs , Config, Pos1, Color, Pos2, Config2) :-
    member(Dir, TakeDirs),
    add_positions(Pos1, Dir, Pos2),
    \+ is_empty(Config, Pos2),
    promote(PromoteRow, Pos2, Type),
    move(Config, Pos1, Pos2, piece(Color, Type), Temp),
    set_enpassant(Temp, nil, Config2).

% Pawn skip row at base position.
```

```
pawn(Color, BaseRow, _, MoveDir , _ , Config, BaseRow/C, Color, Pos3, Config2) :-
    add_positions(BaseRow/C, MoveDir, Pos2),
    is_empty(Config, Pos2),
    add_positions(Pos2, MoveDir, Pos3),
    is_empty(Config, Pos3),
    move(Config, BaseRow/C, Pos3, piece(Color, pawn), Temp),
    set_enpassant(Temp, Pos2, Config2).

% Pawn enpassant.
pawn(Color, _, _, MoveDir , TakeDirs , Config, Pos1, Color, EnpassantPos, Config2) :-
    get_enpassant(Config, EnpassantPos),
    member(Dir, TakeDirs),
    add_positions(Pos1, Dir, EnpassantPos),
    is_empty(Config, EnpassantPos),
    reverseDir(MoveDir, OtherMoveDir),
    add_positions(EnpassantPos, OtherMoveDir, SlayPawnPos),
    move(Config, Pos1, EnpassantPos, piece(Color, pawn), Temp1),
    set_square(Temp1, SlayPawnPos, nil, Temp2),
    set_enpassant(Temp2, nil, Config2).

/**
* promote a piece to a knight, bishop, rook or queen if the piece has reached it's promotion row.
*
* @arg PromoteRow The row on which to promote a piece.
* @arg Position The position of the piece.
* @arg Type The type of piece to which the pawn can promote.
*/
promote(PromoteRow, Row/_, pawn) :- PromoteRow \= Row.
promote(Row, Row/_, Piece) :- member(Piece, [knight, bishop, rook, queen]).

/**
* Get the castle opstions for a given color.
*
* @arg Color The player color.
* @arg Config The game configuration.
* @arg Castle The castle options for the given color.
*/
get_current_castle(w, Config, (WK, WQ)) :- get_castle(Config, castle(WK, WQ, _, _)).
get_current_castle(b, Config, (BK, BQ)) :- get_castle(Config, castle(_, _, BK, BQ)).

/**
* Generate the castle opstions for the 2 colors.
* @arg Config The current configuration
* @arg Pos1 The position of the current players king.
* @arg Color The color of the current configuration.
* @arg Pos2 The new Position of the king.
* @arg Config2 The new Configuration with the king and rook moved.
*/
castle_options(Config, Pos1, Color, Pos2, Config2) :-
    get_current_castle(Color, Config, CastleOptions),
    (
        castle_options([6, 7], [5, 6], 5, 7, 8, 6, (true, _ ),
                       Config, Pos1, piece(Color, king), CastleOptions, Pos2, Config2)
    ;
        castle_options([2, 3, 4], [5, 4], 5, 3, 1, 4, ( _, true ),
                       Config, Pos1, piece(Color, king), CastleOptions, Pos2, Config2)
    ).

castle_options(Empty, NotAttacked, OldK, NewK, OldR, NewR, Sides,
               Config, R/OldK, piece(C, king), Sides, R/NewK, NewConfig) :-
    forall(member(C2, Empty), is_empty(Config, R/C2)),
    set_square(Config, R/OldK, nil, Temp1),
    forall(
        (
            member(C2, NotAttacked),
            other_player(C, OtherColor),
            set_square(Config, R/C2, piece(king, C), T1),
            set_turn(T1, OtherColor, Temp),
            options_no_check(Temp, Temp2)
        ),
        get_square(Temp2, R/C2, piece(king, C))
    ),
    set_square(Temp1, R/OldR, nil, Temp2),
    set_square(Temp2, R/NewK, piece(C, king), Temp3),
    set_square(Temp3, R/NewR, piece(C, rook), NewConfig).

/**
* Reverse the signs of a coordinate.
*
* @arg Coordinate The initial coordinate.
* @arg ReversedCoordinate The coordinate with the signs flipped.
*/
reverseDir(R/C, R2/C2) :- R2 is R * (-1), C2 is C * (-1).


% THIS IS JUST MOVED IN TEST MODE, because this is really slow and in a real engine you should never test this explicitly.
is_not_check(Config, Color) :- pos_not_attacked(Config, _, piece(Color, king)).

/**
* Check if the piece on the given position is not attacked.
*
* @arg Config The current configuration.
* @arg The current position.
* @arg The piece on the position.
*/
pos_not_attacked(Config, Pos, Piece) :-
    forall(options_no_check(Config, NewConfig), get_square(NewConfig, Pos, Piece)).

/**
* Update the castle options.
*
* @arg Config The current configuration.
* @arg NewConfig The configuration with the castle options updated.
*/
update_castle(Config, NewConfig) :-
    get_castle(Config, castle(A1, B1, C1, D1)),
    check_castling(Config, 1/8, 1/5, w, A1, A2),
    check_castling(Config, 1/1, 1/5, w, B1, B2),
    check_castling(Config, 8/8, 8/5, b, C1, C2),
    check_castling(Config, 8/1, 8/5, b, D1, D2),
    set_castle(Config, castle(A2, B2, C2, D2), NewConfig).

/**
* Check if castling is still possible.
* Check if rook and king ar moved and if castling was possible in the previous configuration.
*
* @arg Config The previous configuration with the piece already moved.
* @arg RookPosition Position the rook should be in for castling to be valid.
* @arg KingPosition.Position the king should be in for castling to be valid.
* @arg Color The color for which we are currently checking a castling option.
* @arg OldCastle Old castling options (most be true for the new castlig options to be true).
* @arg NewCastle New options (true if castling is still possible else false).
*/
check_castling(Config, RookPosition, KingPosition, Color, true, true) :-
    get_square(Config, RookPosition, piece(Color, rook)),
    get_square(Config, KingPosition, piece(Color, king)), !.

check_castling(_, _, _, _, _, false).

% vim: set sw=4 ts=4 et :
```

11.1.5 chess_engine.pl

```
:- module(chess_engine, [engine/2]).
:- use_module(chess_alpha_beta).
:- use_module(chess_rules).
:- use_module(chess_operations).



/**
* Try to find the best next board with alpha-beta pruning.
*
* @arg Config The board configuration.
* @arg NewConfig The next board configuration.
*/

% force draw on the 150th move.
% I don't care if I can put my opponent in mate at this time.
engine(Config, 'DRAW') :- get_half_count(Config, L), L >= 149, !.

engine(Config, NewConfig) :-
    % Get the current Player and let the engine optimize moves for this color.
    alpha_beta(Config, NewConfig, _), ! . % Start alpha-beta pruning.

engine(_, 'DRAW').
% vim: set sw=4 ts=4 et :
```

11.1.6 chess_alpha_beta.pl

```
:- module(chess_alpha_beta, [alpha_beta/3]).
:- use_module(chess_evaluation).
:- use_module(chess_rules).
:- use_module(chess_operations).

/**
* Find the next move with alpha beta pruning.
*
* @arg Me The color of the player to optimize.
* @arg Config The current configuration.
* @arg BestConfig The best next configuration according to the engine.
*/
alpha_beta(Config, BestConfig, BestVal) :-
    get_turn(Config, Me),
    bagof(NextConfig, options(Config, NextConfig), NextConfigList),
    random_permutation(NextConfigList, RandomNextConfigList),
    find_depth(RandomNextConfigList, Depth),
    boundedbest(Me, Depth, RandomNextConfigList, -(inf), inf, BestConfig, BestVal), !.
```

```
/**
* Recursive alpha beta pruning entry point.
*
* @arg Me The color of the player to optimize.
* @arg Depth The depth decrement count.
* @arg Config The current configuration.
* @arg Alpha The Lower bound in alpha-beta pruning.
* @arg Beta The upper bound in alpha-beta pruning.
* @arg BestConfig The Best config the engine could find.
* @arg BestVal The estimated value of the best board.
*/
alpha_beta(Me, Depth, Config, Alpha, Beta, BestConfig, BestVal) :-
    succ(NewDepth, Depth), % this fails silently on -1 so we don't go below 0

    % don't seek past checkmates
    get_square(Config, _, piece(w, king)), get_square(Config, _, piece(b, king)),
    % this may include positions where we put ourself in check but that's catched by the king's high value
    bagof(NextConfig, options_no_check(Config, NextConfig), NextConfigList),
    % calculate best next move
    boundedbest(Me, NewDepth, NextConfigList, Alpha, Beta, BestConfig, BestVal), !.

alpha_beta(Me, _, Config, _, _, _, BestVal) :- evaluate(Me, Config, BestVal). %estimate the evaluation.

/**
* Recursively call alpha beta on a new depth and bound the game tree if possible.
*
* @arg Me The current player to optimize.
* @arg Depth The depth decrement count of the tree.
* @arg ConfigurationList The list of all configurations at the current depth.
* @arg Alpha The lower bound.
* @arg Beta The upper bound.
* @arg BestConfig The best config found from the subtrees.
* @arg Bestval The estimated value of the bestConfig.
*/
boundedbest(Me, Depth, [Config|ConfigList], Alpha, Beta, BestConfig, BestVal) :-
    alpha_beta(Me, Depth, Config, Alpha, Beta, _, Val),
    goodenough(Me, Depth, ConfigList, Alpha, Beta, Config, Val, BestConfig, BestVal).

/**
* Check if bounding is possible or update the bounds en recursively call boundedbest.
*
* @arg Me The current player to optimize.
* @arg Depth The depth decrement count of the tree.
* @arg ConfigurationList The list of all configurations at the current depth.
* @arg Alpha The lower bound.
* @arg Beta The upper bound.
* @arg Val The value of the current node.
* @arg BestConfig The best config found from the subtrees.
* @arg Bestval The estimated value of the bestConfig.
*/
goodenough(_, _, [], _, _, Config, Val, Config, Val).
goodenough(Me, _, _, Alpha, Beta, Config, Val, Config, Val) :-
    \+ get_turn(Config, Me), Val > Beta, !
    ;
    get_turn(Config, Me), Val < Alpha, !.

goodenough(Me, Depth, ConfigList, Alpha, Beta, Config, Val, BestConfig, BestVal) :-
    newbounds(Me, Alpha, Beta, Config, Val, NewAlpha, NewBeta),
    boundedbest(Me, Depth, ConfigList, NewAlpha, NewBeta, Config1, Val1),
    betterOf(Me, Config, Val, Config1, Val1, BestConfig, BestVal).


/**
* Changes the bound according to the alpha-beta pruning scheme.
*
* @arg Me The color of the player to optimize.
* @arg Alpha The Lower bound in alpha-beta pruning.
* @arg Beta The upper bound in alpha-beta pruning.
* @arg Config The current configuration.
* @arg Val The current estimated board value.
* @arg NewAlpha The new alpha bound.
* @arg NewBeta The new beta bound.
*/
newbounds(Me, Alpha, Beta, Config, Val, Val, Beta) :- \+ get_turn(Config, Me), Val > Alpha, !.
newbounds(Me, Alpha, Beta, Config, Val, Alpha, Val) :- get_turn(Config, Me), Val < Beta, !.
newbounds(_, Alpha, Beta, _, _, Alpha, Beta).

/**
* Maximize or minimize the game configuration value according to which player has to move.
*
* @arg Me The color of the player to optimize.
* @arg Config0 The first game configuration.
* @arg Val0 The value estimated for game configuration 0.
* @arg Config1 The second game configuration.
* @arg Val1 The value estimated for game configuration 1.
* @arg BestConfig The best game configuration in the current node in the game tree.
* @arg BestVal The best value for the player in the current node in the game tree.
*/
betterOf(Me, Config0, Val0, _, Val1, Config0, Val0) :-
    % If after the move we want to judge it's the other's player's turn. that means the move is
    % ours. we want to maximize value
    get_turn(Config0, Me), Val0 < Val1, !
    ;
    % the other player wants to maximize value
    \+ get_turn(Config0, Me), Val0 > Val1, !.

betterOf(_, _, _, Config1, Val1, Config1, Val1).

/**
* Estimate the depth according to the length of the root's children.
*
* @arg NextConfigList The list of the first configurations.
* @arg Depth The chosen depth.
*/
find_depth(NextConfigList, Depth) :- length(NextConfigList, L), depth(L, Depth).
depth(L, 3) :- L < 15, !.
depth(_, 2).

% vim: set sw=4 ts=4 et :
```

11.1.7 chess_evaluation.pl

```
:- module(chess_evaluation, [evaluate/3]).
:- use_module(chess_operations).
:- use_module(chess_debug).

```

```
/**
* Calculate the value estimation of a bord.
*
* @arg Me The color of the player to optimize.
* @arg GameState The current state of the game.
* @arg Val The estimated value of the GameState.
*/
evaluate(Me, GameState, Val) :-
    findall(Val,
        (
            get_square(GameState, Pos, piece(PieceColor, Type)),
            value(Me, PieceColor, Type, Pos, Val)
        ), List1),
    sum_list(List1, Val).

/**
* Calculate the value estimation of a piece in a certain position.
*
* @arg Me The color of the player to optimize.
* @arg Other The color of the piece to value.
* @arg Type The type of piece eg: pawn, knight...
* @arg Pos The current position of the piece.
* @arg Val The value for the piece in the current gamestate.
*/
value(Me, PieceColor, Type, Pos, Val) :-
    value(Type, Val1),
    translate_table(Pos, PieceColor, Pos2),
    position(Type, Pos2, Val2),
    ProtoVal is Val1 + Val2,
    sign(Me, PieceColor, ProtoVal, Val).

/**
* Calculate the sign of the value.
* The sign is non altered if the color equal to the color to optimize.
* The sign is switched if the color is equal to the other player.
* @arg Me The player to optimize.
* @arg PieceColor The color to whom this piece belongs.
* @arg CurrentVal The currentvalue
* @arg ModifiedVal The AlteredValue
*/
sign(Me, Me, Val, Val) :- !.
sign(_, _, Val, -(Val)).

/**
* Find the default score of a piece.
* This scoring system satisfies the following equation.
* bishop > knight > 3 * pawn
* bishop + knight = rook + 1.5 * pawn
* queen + pawn = 2 * rook
*
* @arg pieceType The type of the piece.
* @arg score The default score of the piece.
*/
value(pawn, 100).
value(knight, 320).
value(bishop, 330).
value(rook, 500).
value(queen, 900).
value(king, 20000).

/**
```

```
* Translate the tables who are defined in function of black pieces.
* @arg coordinate Original board coordinate.
* @arg color Color of the piece.
* @arg tranlatedCoordinate Translated boad coordinate.
*/
translate_table(R/C, b, R/C) :- !.
translate_table(R/C, w, R2/C) :- R2 is 9 - R.

% Pawn's placement scores.
position(pawn, R/C, Val) :-
  nth1(
    R,
    [[0, 0, 0, 0, 0, 0, 0, 0],
    [50, 50, 50, 50, 50, 50, 50, 50],
    [10, 10, 20, 30, 30, 20, 10, 10],
    [ 5, 5, 10, 25, 25, 10, 5, 5],
    [ 0, 0, 0, 20, 20, 0, 0, 0],
    [ 5, -5, -10, 0, 0, -10, -5, 5],
    [ 5, 10, 10, -20, -20, 10, 10, 5],
    [ 0, 0, 0, 0, 0, 0, 0, 0]],
    Row
  ),
  nth1(C, Row, Val), !.

% Knight's placement scores.
position(knight, R/C, Val) :-
  nth1(
    R,
    [[-50, -40, -30, -30, -30, -30, -40, -50],
    [ -40, -20, 0, 0, 0, 0, -20, -40],
    [ -30, 0, 10, 15, 15, 10, 0, -30],
    [ -30, 5, 15, 20, 20, 15, 5, -30],
    [ -30, 0, 15, 20, 20, 15, 0, -30],
    [ -30, 5, 10, 15, 15, 10, 5, -30],
    [ -40, -20, 0, 5, 5, 0, -20, -40],
    [ -50, -40, -30, -30, -30, -30, -40, -50]],
    Row
  ),
  nth1(C, Row, Val), !.

% Bishop's placement scores.
position(bishop, R/C, Val) :-
  nth1(
    R,
    [[-20, -10, -10, -10, -10, -10, -10, -20],
    [ -10, 0, 0, 0, 0, 0, 0, -10],
    [ -10, 0, 5, 10, 10, 5, 0, -10],
    [ -10, 5, 5, 10, 10, 5, 5, -10],
    [ -10, 0, 10, 10, 10, 10, 0, -10],
    [ -10, 10, 10, 10, 10, 10, 10, -10],
    [ -10, 5, 0, 0, 0, 0, 5, -10],
    [ -20, -10, -10, -10, -10, -10, -10, -20]],
    Row
  ),
  nth1(C, Row, Val), !.

% Rook's placement scores.
position(rook, R/C, Val) :-
  nth1(
    R,
    [[0, 0, 0, 0, 0, 0, 0, 0],
    [ 5, 10, 10, 10, 10, 10, 10, 5],
    [-5, 0, 0, 0, 0, 0, 0, -5],
    [-5, 0, 0, 0, 0, 0, 0, -5],
    [-5, 0, 0, 0, 0, 0, 0, -5],
    [-5, 0, 0, 0, 0, 0, 0, -5],
    [-5, 0, 0, 0, 0, 0, 0, -5],
    [ 0, 0, 0, 5, 5, 0, 0, 0]],
    Row
  ),
  nth1(C, Row, Val), !.

%Queen's placement scores.
position(queen, R/C, Val) :-
  nth1(
    R,
    [[-20, -10, -10, -5, -5, -10, -10, -20],
    [ -10, 0, 0, 0, 0, 0, 0, -10],
    [ -10, 0, 5, 5, 5, 5, 0, -10],
    [  -5, 0, 5, 5, 5, 5, 0, -5],
    [   0, 0, 5, 5, 5, 5, 0, -5],
    [ -10, 5, 5, 5, 5, 5, 0, -10],
    [ -10, 0, 5, 0, 0, 0, 0, -10],
    [ -20, -10, -10, -5, -5, -10, -10, -20]],
    Row
  ),
  nth1(C, Row, Val), !.

%King's placement scores.
position(king, R/C, Val) :-
  nth1(
    R,
    [[-30, -40, -40, -50, -50, -40, -40, -30],
    [ -30, -40, -40, -50, -50, -40, -40, -30],
    [ -30, -40, -40, -50, -50, -40, -40, -30],
    [ -30, -40, -40, -50, -50, -40, -40, -30],
    [ -20, -30, -30, -40, -40, -30, -30, -20],
    [ -10, -20, -20, -20, -20, -20, -20, -10],
    [  20, 20, 0, 0, 0, 0, 20, 20],
    [  20, 30, 10, 0, 0, 10, 30, 20]],
    Row
  ),
  nth1(C, Row, Val), !.

% vim: set sw=4 ts=4 et :
```

11.2 Test

11.2.1 test_main.pl

```
#!/usr/bin/env swipl
:- initialization(main, main).

main(_) :-
    TestFiles = ['test_chess_io.pl', 'test_chess_rules.pl'],
    consult(TestFiles),
    load_test_files(TestFiles),
    show_coverage(run_tests),
    halt(0).
```

11.2.2 test_chess_io.pl

```
:- begin_tests(chess_io).
:- set_prolog_flag(double_quotes, chars).
:- use_module(test_chess_util).
:- use_module('../src/chess_operations.pl').
:- use_module('../src/chess_io').


/**
* Test the is piece function mapping a char to a piece.
*/
test(is_piece, [forall((piece_type(T), color(C)))]) :-
    nth1(Index, [pawn/b, knight/b, bishop/b, rook/b, queen/b, king/b,
                 pawn/w, knight/w, bishop/w, rook/w, queen/w, king/w], T/C),
    nth1(Index, "pnbrqkPNBRQK", Letter),
    chess_io:is_piece(Letter, piece(C, T)), !.

/**
* Test the piece parsing function.
*/
test(piece, [forall((piece_type(T), color(C)))]) :-
    nth1(Index, [pawn/b, knight/b, bishop/b, rook/b, queen/b, king/b,
                 pawn/w, knight/w, bishop/w, rook/w, queen/w, king/w], T/C),
    nth1(Index, "pnbrqkPNBRQK", Letter),
    phrase(chess_io:piece([piece(C, T) | X] - X), [Letter]), !.
/**
* Test building a list containing nil's for a certain length.
*/
test(build_empty, [forall(between(1, 8, L))]) :-
    chess_io:build_empty(L, List - []),
    length(List, L),
    forall(member(X, List), X=nil), !.

/**
* Test parse empty squares.
*/
test(empty, [forall(nth1(L, "12345678", Char))]) :-
    phrase(chess_io:empty(List - []), [Char]),
    length(List, L),
    forall(member(X, List), X=nil), !.
/**
* Test parse emtpy squares.
*/
test(enpassant) :-
    phrase(chess_io:en_passant(nil), "-"), !.

test(enpassant, forall((nth1(C, "abcdefgh", CChar), member(R/RChar, [3/'3', 6/'6'])))) :-
    phrase(chess_io:en_passant(R/C), [CChar, RChar]).

to_wk(false, ''). to_wk(true, 'K').
to_wq(false, ''). to_wq(true, 'Q').
to_bk(false, ''). to_bk(true, 'k').
to_bq(false, ''). to_bq(true, 'q').

/**
* Test parsing castling options.
*/
test(castle) :-
    phrase(chess_io:castle(castle(false, false, false, false)), "-"), !.

```

```
test(castle, forall(maplist(bool, [_, _, _, _], [A, B, C, D]))) :-
    Castle =.. [ castle | [A, B, C, D]],
    to_wk(A, A2), to_wq(B, B2),
    to_bk(C, C2), to_bq(D , D2),
    exclude(=(''), [A2, B2, C2, D2], L),
    phrase(chess_io:castle(Castle), L), !.



fenweight(C, Val) :- nth1(Val, "12345678", C), !.
fenweight(C, 1) :- member(C, "pnbrqkPNBRQK").
valid_fen_row(Fen, N) :- maplist(fenweight, Fen, NumberList), sum_list(NumberList, X), X==N .
valid_square(nil).
valid_square(piece(C, P)) :- color(C), piece_type(P).
/**
*
* Test all valid sub rows of length 3.
*/
test(pieces, forall(phrase(chess_io:pieces([A, B, C]-[]), Fen))) :-
    valid_fen_row(Fen, 3),
    forall(member(Square, [A, B, C]), valid_square(Square)),
    !.

:- end_tests(chess_io).

% vim: set sw=4 ts=4 et :
```

11.2.3 test_chess_rules.pl

```
:- begin_tests(chess_rules).
:- use_module('../src/chess_rules.pl').
:- use_module('../src/chess_operations.pl').
:- use_module('../src/chess_debug.pl').
:- use_module('../src/chess_io').
:- use_module(test_fen_db).
:- use_module(test_chess_util).

update_wk(true, Config, NewConfig) :- set_square(Config, 1/8, piece(w, rook), NewConfig).
update_wk(false, Config, Config).
update_wq(true, Config, NewConfig) :- set_square(Config, 1/1, piece(w, rook), NewConfig).
update_wq(false, Config, Config).
update_bk(true, Config, NewConfig) :- set_square(Config, 8/8, piece(b, rook), NewConfig).
update_bk(false, Config, Config).
update_bq(true, Config, NewConfig) :- set_square(Config, 8/1, piece(b, rook), NewConfig).
update_bq(false, Config, Config).

and(false, false, false). and(false, true, false).
and(true, false, false). and(true, true, true).

/**
* Test the updating of the castling options.
*/
test(update_castle, [forall((maplist(and, [ A, B, C, D ], List2, After)))]):-
    CastleBegin =.. [ castle | List2 ],
    CastleAfter =.. [ castle | After ],
    empty_config(Config),
    set_castle(Config, CastleBegin, Config2),
    set_square(Config2, 1/5, piece(w, king), Config2A),
    set_square(Config2A, 8/5, piece(b, king), Config3),
    update_wk(A, Config3, Config4),
    update_wq(B, Config4, Config5),
    update_bk(C, Config5, Config6),
    update_bq(D, Config6, Config7),
    chess_rules:update_castle(Config7, Config8),
    get_castle(Config8, CastleAfter).
/**
* Test the promotion help function
*/
test(promote, [forall((member(Row, [1, 8]), all_coordinates(R/C)))]) :-
    findall(Type, chess_rules:promote(Row, R/C, Type), List),
    (Row = R ->
        member(knight, List),
        member(bishop, List),
        member(rook, List),
        member(queen, List),
        \+ member(pawn, List),
        \+ member(king, List)
    ;
        List = [ pawn ]
    ), !.

/**
* The folowing rules are simplified rules for the pieces
* used to test if the normall movement of the pieces is correct.
* To test edgecases we will do different hardcoded tests.
*/

pawn_pos(Conf, w, R1/C, R2/C) :- succ(R1, R2), chess_operations:is_empty(Conf, R2/C).
pawn_pos(Conf, b, R1/C, R2/C) :- succ(R2, R1), chess_operations:is_empty(Conf, R2/C).
pawn_pos(Conf, w, R1/C1, R2/C2) :- succ(R1, R2), succ(C1, C2), \+ chess_operations:
    ↪ is_empty(Conf, R2/C2).
pawn_pos(Conf, b, R1/C1, R2/C2) :- succ(R2, R1), succ(C1, C2), \+ chess_operations:
    ↪ is_empty(Conf, R2/C2).
pawn_pos(Conf, w, R1/C1, R2/C2) :- succ(R1, R2), succ(C2, C1), \+ chess_operations:
    ↪ is_empty(Conf, R2/C2).
pawn_pos(Conf, b, R1/C1, R2/C2) :- succ(R2, R1), succ(C2, C1), \+ chess_operations:
    ↪ is_empty(Conf, R2/C2).
pawn_pos(Conf, w, 2/C, 4/C) :- chess_operations:is_empty(Conf, 3/C), chess_operations:
    ↪ is_empty(Conf, 4/C).
pawn_pos(Conf, b, 7/C, 5/C) :- chess_operations:is_empty(Conf, 6/C), chess_operations:
    ↪ is_empty(Conf, 5/C).

knight_pos(_, _, Pos1, Pos2) :-
    maplist(
        chess_operations:add_positions(Pos1),
        [ 1/2, (-1)/2, 1/(-2), (-1)/(-2), 2/1, 2/(-1), (-2)/(1), (-2)/(-1)],
        L1
    ), member(Pos2, L1).

king_pos(_, _, Pos1, Pos2) :-
    maplist(
        chess_operations:add_positions(Pos1),
        [ 1/0, 1/1, (-1)/0, (-1)/(-1), (-1)/1, 1/(-1), (0)/(1), (0)/(-1)],
        L1
    ), member(Pos2, L1).

keep_moving(Conf, _, [ Pos ], Pos) :- \+ chess_operations:is_empty(Conf, Pos), ! .

keep_moving(Conf, DR/DC, [ R/C | History ], R/C) :-
    R2 is R + DR, C2 is C + DC,
    keep_moving(Conf, DR/DC, History, R2/C2).

keep_moving_start(Conf, DR/DC, History, R/C) :-
    R2 is R + DR, C2 is C + DC,
    keep_moving(Conf, DR/DC, History, R2/C2).

bishop_pos(Conf, _, Pos1, Pos2) :-
    (
        keep_moving_start(Conf, 1/1, Moves, Pos1);
        keep_moving_start(Conf, 1/(-1), Moves, Pos1);
        keep_moving_start(Conf, (-1)/1, Moves, Pos1);
        keep_moving_start(Conf, (-1)/(-1), Moves, Pos1)
    ), member(Pos2, Moves).

rook_pos(Conf, _, Pos1, Pos2) :-
    (
        keep_moving_start(Conf, 0/1, Moves, Pos1);
        keep_moving_start(Conf, 0/(-1), Moves, Pos1);
        keep_moving_start(Conf, (-1)/0, Moves, Pos1);
        keep_moving_start(Conf, 1/0, Moves, Pos1)
    ), member(Pos2, Moves).
queen_pos(Conf, C, Pos1, Pos2) :- rook_pos(Conf, C, Pos1, Pos2) ; bishop_pos(Conf, C, Pos1, Pos2).

goal_piece((Goal, Piece)) :-
    member(
        (Goal, Piece),
        [
            (pawn_pos, pawn),
            (knight_pos, knight),
            (bishop_pos, bishop),
            (rook_pos, rook),
            (queen_pos, queen),
            (king_pos, king)
        ]
    ).

%%%%%%%%% RANDOM TESTS %%%%%%%%%%

/**
*
* Create an empty bord.
* Place a test piece on the bord.
* Fill the board with other pieces.
* Check if all the moves of the piece are valid.
*/
random_run(Validator, Type) :-
    empty_config(EmptyConfig),
    random_position(Pos), random_color(C),
    set_square(EmptyConfig, Pos, piece(C, Type), Config),
    random(0, 20, Amount), % place a random amount of pieces on the board.
    random_pieces(Config, [w, b], [ pawn, knight, bishop, queen, king ], Amount, NewConfig),
    findall(Pos2, call(Validator, NewConfig, C, Pos, Pos2), T),
    filter_valid_me(T, NewConfig, Coordinates1),
    findall(Pos2, chess_rules:update_board(NewConfig, Pos, piece(C, Type), Pos2, _), Coordinates2),
    equal(Coordinates1, Coordinates2).

/**
```

```
* Generate 1000 random bords for each piece and check if their default movement is correct.
*/
test(update_board, [forall((between(0, 1000, _), goal_piece((Goal, Piece))))]) :- random_run(Goal, Piece).

%%%%%%%%%% HARDCODED TESTS FOR EDGECASES ENPASSANT CHECK ETC %%%%%%%%%%%%

/**
* Convert FenAtom to a board configuration.
*/
fenatom_to_board(FenAtom, Config) :-
    atomic_list_concat(Args, ' ', FenAtom),
    chess_io:fen_io(Args, Config).

/**
* Check if the moves generated by the move generator equal the actual moves in the test database.
*/
test(options, [forall(fens(X))]) :-
    fen(ex, X, FenAtom), fenatom_to_board(FenAtom, Config),
    findall(Option, chess_rules:options(Config, Option), Options1),
    findall(Option, (fen(sol, X, Option)), Temp),
    maplist(fenatom_to_board, Temp, Options2),
    equal(Options1, Options2), !.



:- end_tests(chess_rules).

% vim: set sw=4 ts=4 et :
```

11.2.4 test_chess_util.pl

```
:- module(test_chess_util, [piece_type/1, color/1, bool/2, empty_config/1, random_pieces/5, random_color/1, random_position/1, equal/2, filter_valid_me/3]).
:- use_module('../src/chess_operations.pl').
:- use_module('../src/chess_rules.pl').


%%%%%%%%%%% DISCLAIMER UGLY UTIL FUNCTIONS USED IN TESTS %%%%%%%%%%%%%%

bool(_, false). bool(_, true).
piece_type(X) :- member(X, [pawn, knight, bishop, rook, queen, king]).
color(w).
color(b).


empty_config(fen_config(
    board(
        row(nil, nil, nil, nil, nil, nil, nil, nil),
        row(nil, nil, nil, nil, nil, nil, nil, nil),
        row(nil, nil, nil, nil, nil, nil, nil, nil),
        row(nil, nil, nil, nil, nil, nil, nil, nil),
        row(nil, nil, nil, nil, nil, nil, nil, nil),
        row(nil, nil, nil, nil, nil, nil, nil, nil),
        row(nil, nil, nil, nil, nil, nil, nil, nil),
        row(nil, nil, nil, nil, nil, nil, nil, nil)
), w, castle(false, false, false, false), nil, 0, 1)).

```

```prolog
% Taken from stackoverflow https://stackoverflow.com/questions/27151274/prolog-take-the-first-n-elements-of-a-list
take(Src, N, L) :- findall(E, (nth1(I, Src, E), I =< N), L).


% Hack for lambda I don't know how to fix this elegantly.
square(Config, Pos, Piece, NewConfig) :- chess_operations:set_square(Config, Pos, Piece, NewConfig).

% Put random pieces on the board.
random_pieces(Config, Colors, Pieces, Amount, NewConfig) :-
    findall(Empty, chess_operations:get_square(Config, Empty, nil), EmptySquares),
    random_permutation(EmptySquares, RandomEmptySquares),
    take(RandomEmptySquares, Amount, Squares),
    % yes, I can still use folds. I love them.
    foldl(([Pos, CurrConfig, Out] >>
        (random_select(Color, Colors, _),
        random_select(Piece, Pieces, _),
        square(CurrConfig, Pos, piece(Color, Piece), Out)))
    , Squares, Config, NewConfig).

%select a random position
random_position(R/C) :- random(1, 9, R), random(1, 9, C).

%select a random color
random_color(C) :- random_select(C, [w, b], _).

% Check if the two lists contain the same elements.
equal(List1, List2) :-
    list_to_set(List1, Set1), list_to_set(List2, Set2),
    length(Set1, L),
    length(Set2, L),
    subtract(Set1, Set2, []), subtract(Set2, Set1, []).

% filter out invalid coordinates and coordinates that already have a piece from the current player.
filter_valid_me(List1, Config, List3) :-
    include(chess_operations:all_coordinates, List1, List2),
    exclude(chess_operations:is_mine(Config), List2, List3).

% vim: set sw=4 ts=4 et :
```