UNIVERSITEIT GENT

LOGISCH PROGRAMMEREN

CHRISTOPHE SCHOLLIERS

Schaakcomputer

Author
BERT COMMEINE

17 augustus 2018

Inhoudsopgave

1 Inleiding

De bedoeling van deze opdracht was om een programma te maken die een fen-notatie van een bordstaat kon inlezen, deze verwerken, en aan de hand van een min-max boom de best mogelijke zet terugsturen. Dit allemaal aan de hand van het logsich programmeerparadigma en in de taal prolog.

Het project die ik nu indien is een verbeterde versie van mijn vorig project. Daarom is een stuk van dit verslag ook onveranderd gebleven. De grootste veranderingen zijn te vinden op het einde van het stuk "Werking", een kleine aanpassing aan de bordvoorstelling en in subparagraaf 4.2. Daarnaast is de conclusie ook volledig herschreven.

2 Werking

Bij het starten van de mainfunctie met de juiste argumenten zullen deze argumenten eerst naar een string worden omgezet, gescheiden door een komma. Dit is omdat de parser begint bij een string. Zo is de omgekeerde bewerking $(Voorstelling \rightarrow String)$ gemakkelijker te implementeren. Na het parsen van de input hebben we dus de bordstaat in de vorm van onze interne voorstelling. Hoe deze er precies uitziet en waarom wordt in het volgend stuk besproken.

Bij het genereren van zetten worden voor alle stukken hun mogelijke zetten gezocht. Wanneer er een zet gevonden is, gaan we na of deze zet "validïs: De koning mag niet schaak staan na deze zet. Wanneer al die zetten gevonden zijn zullen deze een score krijgen gebaseerd op de bordstaat na de zet zelf, als ook de daaropvolgende bordstaten.

In tegenstelling tot vorige keer is het nu ook effectief mogelijk om een schaakspel te spelen tegen de schaakcomputer. Hoewel mijn eerste versie ook zetten kon genereren waren deze niet volledig correct door o.a. niet-functionerende zettentellers. Maar meer over de testen volgt later.

3 Spelvoorstelling

Eerst en vooral wil ik een duidelijk verschil maken tussen mijn spelvoorstelling en mijn bordvoorstelling. Deze laatste is namelijk een deel van mijn spelvoorstelling. Hieronder ziet u de spelvoorstelling, met daarin als eerste stuk de bordvoorstelling. De spelstaat is dus een array van 6 variabelen:

Bord	Beurt	Rokade	Enpassant	Halve	Volle

1. Het bord wordt voorgesteld adhv de relatie rows tussen 8 rijen. Deze rijen op zich worden voorgesteld met lijsten van stukken. Tenslotte worden de stukken voorgesteld met de constante $"piece(rook, black)"$ in het geval van een zwarte toren.
2. De speler die aan de beurt is wordt gewoon bijgehouden als white of black.

Tabel 1: Waarde van stukken

Stuk	Waarde
Pion	10
Paard	30
Toren	50
Loper	30
Koningin	90
Koning	9001

3. De rokade-opties worden voorgesteld met dezelfde characters als in de fen notatie.
4. Een mogelijkheid tot enpassant wordt opnieuw voorgesteld met dezelfde characters als in de fen-notatie
5. De halve-zettenteller is een integer
6. De volle-zettenteller is opnieuw een integer

Voor de rest valt er niet zoveel meer te vertellen over de bordrepresentatie. Deze manier van voorstellen bleek het handigste te zijn om waarden op te zoeken en aan te passen. De rasterstructuur maakte het implementeren ook intuïtiever.

Hier is één ding veranderd sinds vorige verslag, en dat is dat ik de laatste variabele heb laten vallen. Deze hield bij of ik aan een min- of een maxstap zat bij het bepalen van een zet, maar door mijn nieuwe min-max boom is dit niet meer nodig.

4 Algoritme

Het algoritme zullen we opsplitsen in twee stukken: De waardeberekening van een bordstaat en het zoeken naar de meest optimale zet:

4.1 Waardebepaling

Een heel eenvoudige maar effectieve manier om waarden te geven aan bordstaten is door het optellen van de stukken in de kleur van de speler (maal een factor die aangeeft hoe belangrijk het stuk is) en van deze som dezelfde berekening voor de tegestander aftrekken. Logischerwijs streeft men dus naar een zo hoog mogelijk resultaat, aangezien dit betekent dat men meer (waardevolle) stukken heeft dan de tegenstander. De factor per stuksoort staat hieronder vermeld.

4.2 Optimale zet

Het berekenen van een optimale zet gebeurt aan de hand van een min-max boom. Deze datastructuur probeert afwisselend zo goed mogelijke/zo slecht mogelijke zetten te vinden voor de huidige speler. Nu had ik mij initieel gebaseerd op het lesvoorbeeld, maar er zat een fout in mijn implementatie dus ben ik van nul begonnen. Wel heb ik hem aan hetzelfde

interface (= [move/2, value/2]) laten voldoen, maar dit was meer als vangnet bedoeld.

Mijn min-max boom is vrij algemeen geïmplementeerd zodat ik met een variabele diepte kan werken. In mijn main methode heb ik dan (na wat te testen) beslist om mijn diepte op 2 te zetten. Meer diepte gaf niet echt veel betere resulaten maar duurde opmerkelijk veel langer.

5 Testen

Zoals u zult zien wint mijn schaakcomputer in de meerderheid van de spelen tegen de testcomputer. De spellen die hij niet wint eindigen meestal in remise. Nu en dan zit er een spel tussen waarbij mijn programma geen schaakmat kan vinden voor de 50 zetten regen van toepassing is, maar deze situaties zijn vrij zeldzaam.

Daarnaast is er ook een nieuwe test voorzien, en ook hier zijn er geen testen die falen.

6 Conclusie

Dit project is, zoals eerder vermeld, de afgewerkte versie van mijn eerstezit project. In mijn vorige versie zaten enkele stukken die, hoewel ze functioneerden, duidelijk niet de beste oplossing waren. Naast aanvullen van wat ontbrak heb ik ook enkele van deze stukken herschreven. Bijvoorbeeld het parsen van de halve- en vollezettenteller heb ik nu wel correct als een getal (ipv. een lijst van cijfers) geparst.

Daarnaast heb ik ook de spelregels in de computer aangevuld. De eerste versie kon bvb. niet rokeren of enpassant slaan. De huidige versie slaagt voor alle testborden die meegegeven werden, en heeft al talloze spellen gespeeld tegen de testcomputer, zonder een verkeerde zet te doen.

Tenslotte heb ik ook de min-max boom nog eens herbekeken. De vorige versie vertoonde inconsistent gedrag op bepaalde dieptes. Omdat dit enerzijds zeer moeilijk te debuggen valt en anderszijds geadopteerd was van het lesmateriaal besloot ik om gewoon opnieuw te beginnen. De fout zat hem bij het bepalen van waarden voor een speler. Deze werkt momenteel correct en heeft een verstelbare diepte.

Ik zie nog twee dingen die mogelijks beter kunnen: Enerzijds wordt de bordvoorstelling gedraaid bij mij. Dit had ik echt veel te laat door, en heb dan maar beslist om het zo te laten, en van tijd tot tijd een complement te nemen van een coördinaat. Daarnaast heb ik het gevoel dat de schaakcomputer trager is dan andere implementaties, al is dit louter speculatief. Dit zou kunnen liggen aan een cut die ik vergeten ben of een onefficiënte methode om te kijken of ik schaak sta.

Appendix: Code

main.pl

```
#!/usr/bin/env swipl

:- use_module(parser, [parse/2]).
:- use_module(chess, [move/2]).
:- use_module(min_max, [minimax/3]).

:- initialization(main, main).

%Omdat de parser begonnen is van het idee een string om te zetten naar een spelvoorstelling
%zetten we de argumenten eerst om naar een string
main([Board, Turn, Castling, Passant, Half, Full, Test]) :-
    Test  = 'TEST',
    atomic_list_concat([Board, Turn, Castling, Passant, Half, Full],' ',Fen),
    parse(Fen, B),
    findall(X, move(B, X), L),
    parse_list(L),
    halt(0).

main([Board, Turn, Castling, Passant, Half, Full]) :-
    atomic_list_concat([Board, Turn, Castling, Passant, Half, Full],' ',Fen),
    parse(Fen, X),
    minimax(X, B, 2),
    parse(Response, B),
    write(Response),
    nl,
    halt(0).

parse_list([H]) :- parse(Fen, H), write(Fen), nl.
parse_list([H|T]) :- parse(Fen, H), write(Fen), nl, parse_list(T).
```

chess.pl

```
:- module(chess, [move/2, value/2]).
:- use_module(parser, [parse/2]).

% Interface of minimax-module:
% Move

% Deze move voldoet aan het interface, voorzien in de minmax boom
move(Position, NextPosition) :- all_moves(Position, NextPosition),
                                \+(invalid_position(NextPosition)).

% all_moves gaat alle mogelijke zetten, schaak of niet, genereren
```

```
all_moves(Position, NextPosition) :- move_bishop(Position, NextPosition).
all_moves(Position, NextPosition) :- move_king(Position, NextPosition).
all_moves(Position, NextPosition) :- move_knight(Position, NextPosition).
all_moves(Position, NextPosition) :- move_pawn(Position, NextPosition).
all_moves(Position, NextPosition) :- move_queen(Position, NextPosition).
all_moves(Position, NextPosition) :- move_rook(Position, NextPosition).

% Wanneer iemand een zet doet, waarop zijn tegenstander zijn koning
% kan slaan (schaak) is deze zet invalid
invalid_position(Position) :-   get_board(Position, Board),
                                get_turn(Position, Turn),
                                other_color(Turn, Other),
                                find_piece(Board, Column, Row, piece(king, Other)),
                                all_moves(Position, NextPosition),
                                get_board(NextPosition, NextBoard),
                                get_piece(NextBoard, Row, Column, piece(_, Turn)).

% Deze utility functie behoort tot het interface van de minmaxboom
value(Position, Value) :- get_board(Position, Board), get_turn(Position, Turn),
                          board_value(Board, Turn, Value).

% De volgende twee functies ook
wining_to_play(Board, Turn) :- get_turn(Board, Turn).
losing_to_move(Board, Turn) :- get_turn(Board, Other), other_color(Turn, Other).



% Board Value is een functie die een waarde zal geven aan een bord naargelang
% de speler aan beurt nog stukken heeft, en zijn tegenstander geen meer
board_value(Board, Turn, Value) :-  get_row(Board, 1, R1), row_value(R1, V1, Turn),
                                    get_row(Board, 2, R2), row_value(R2, V2, Turn),
                                    get_row(Board, 3, R3), row_value(R3, V3, Turn),
                                    get_row(Board, 4, R4), row_value(R4, V4, Turn),
                                    get_row(Board, 5, R5), row_value(R5, V5, Turn),
                                    get_row(Board, 6, R6), row_value(R6, V6, Turn),
                                    get_row(Board, 7, R7), row_value(R7, V7, Turn),
                                    get_row(Board, 8, R8), row_value(R8, V8, Turn),
                                    Value is V1 + V2 + V3 + V4 + V5 + V6 + V7 + V8, !.

% Row Value (hulpfunctie)
row_value([], 0, _).
row_value([E], V, Turn) :- piece_value(E, Turn, V), !.
row_value([H|T], V, Turn) :- piece_value(H, Turn, V1), row_value(T, V2, Turn), V is V2 + V1.

% Piece Value (hulpfunctie)
piece_value(empty, _, 0).
piece_value(piece(pawn, Turn), Turn, 10).
```

```
piece_value(piece(pawn, Other), Turn, -10) :- other_color(Turn, Other).
piece_value(piece(rook, Turn), Turn, 50).
piece_value(piece(rook, Other), Turn, -50) :- other_color(Turn, Other).
piece_value(piece(knight, Turn), Turn, 30).
piece_value(piece(knight, Other), Turn, -30) :- other_color(Turn, Other).
piece_value(piece(bishop, Turn), Turn, 30).
piece_value(piece(bishop, Other), Turn, -30) :- other_color(Turn, Other).
piece_value(piece(queen, Turn), Turn, 90).
piece_value(piece(queen, Other), Turn, -90) :- other_color(Turn, Other).
piece_value(piece(king, Turn), Turn, 9001).
piece_value(piece(king, Other), Turn, -9001) :- other_color(Turn, Other).

% Utility functions to find data quickly
get_board([Board|_], Board).
get_turn([_, Turn| _], Turn).
get_castling([_, _, Castling|_], Castling).
get_passant([_, _, _, [C, R2]|_], [R, C]) :- R is 9 - R2.
get_half([_, _, _, _, Half|_], Half).
get_full([_, _, _, _, _, Full|_], Full).

% Utility functions to alter gamestate
set_board([_|T], B2,  [B2|T]).
set_castling([Board, Turn, _|T], [Board, Turn, C|T], C).
set_enpassant([Board, Turn, Castling, _|T], [Board, Turn, Castling, Passant|T], Passant).
set_halfmove([Board, Turn, Castling, Passant, _|T],
        [Board, Turn, Castling, Passant, Halfmove|T], Halfmove).

% Swap players en tel eventueel een zet bij na een beurt
swap([Board, white, Castling, Passant, Half, Full],
        [Board, black, Castling, Passant, Half, Full]).
swap([Board, black, Castling, Passant, Half, Full],
        [Board, white, Castling, Passant, Half, Fuller]):- Fuller is Full + 1.

% Find piece on a certain position
get_row(Board, Number, Row) :- arg(Number, Board, Row).
get_piece(Board, Row, Column, Piece) :- get_row(Board, Row, R),
                                        nth0(C, R, Piece),
                                        C is Column - 1.
is_empty(Board, Column, Row) :- get_piece(Board, Row, Column, empty).

% Incrementing the halfmove counter of a gamestate
increment_halfmove(Old, New) :- get_half(Old, OldHalve),
                                NewHalve is OldHalve + 1,
                                set_halfmove(Old, New, NewHalve).

% Find position of a certain piece
valid(X) :- X > 0, X < 9.
```

```
other_color(white, black).
other_color(black, white).
piece_row([Piece |_], 1, Piece).
piece_row([_ | Tail], N, Piece) :- piece_row(Tail, N1, Piece), N is N1 + 1.
find_piece(rows( Row, _, _, _, _, _, _, _), Column, 1, Piece) :- piece_row(Row, Column, Piece)
find_piece(rows(_ , Row, _, _, _, _, _, _), Column, 2, Piece) :- piece_row(Row, Column, Piece)
find_piece(rows(_ , _, Row, _, _, _, _, _), Column, 3, Piece) :- piece_row(Row, Column, Piece)
find_piece(rows(_ , _, _, Row, _, _, _, _), Column, 4, Piece) :- piece_row(Row, Column, Piece)
find_piece(rows(_ , _, _, _, Row, _, _, _), Column, 5, Piece) :- piece_row(Row, Column, Piece)
find_piece(rows(_ , _, _, _, _, Row, _, _), Column, 6, Piece) :- piece_row(Row, Column, Piece)
find_piece(rows(_ , _, _, _, _, _, Row, _), Column, 7, Piece) :- piece_row(Row, Column, Piece)
find_piece(rows(_ , _, _, _, _, _, _, Row), Column, 8, Piece) :- piece_row(Row, Column, Piece)

% Move piece on new board, R1 C1
% set_move(Board, NewBoard, Piece, Column)
set_move([_|T], [Piece|T], Piece, 1).
set_move([H|T1], [H|T2], Piece, N) :- set_move(T1, T2, Piece, N1), N is N1 +1.

% Zet een stuk op een bepaalde plaats op het bord (adhv row/column)
put_piece(Board, Nextboard, Col, Row, Piece) :-
    functor(Board, rows, N),
    functor(Nextboard, rows, N),
    put_piece(N, Board, Nextboard, Col, Row, Piece).

put_piece(Row, Board, Nextboard, Col, Row, Piece):-
    !,
    arg(Row, Board, CurrentRow),
    set_move(CurrentRow, Next, Piece, Col),
    arg(Row, Nextboard, Next),
    N1 is Row-1,
    put_piece(N1,Board,Nextboard,Col, Row,Piece).

put_piece(N, Board, Nextboard, Col, Row, Piece):-
    N > 0,
    arg(N,Board,Arg),
    arg(N,Nextboard,Arg),
    N1 is N-1,
    put_piece(N1,Board,Nextboard,Col, Row,Piece).

put_piece(0, _, _, _, _, _).

take_piece(Board, NewBoard, Castling, Cas, Column, Row, C, R, Piece, Color) :-
  startposition_rook(Board, Other, C, R, T), !,
  delete_castling(T, Castling, Cas),
  other_color(Color, Other),
  put_piece(Board, Next, Column, Row, empty),
  put_piece(Next, NewBoard, C, R, piece(Piece, Color)).
```

```
take_piece(Board, NewBoard, Castling, Castling, Column, Row, C, R, Piece, Color) :-
  find_piece(Board, C, R, piece(_, Other)),
  other_color(Color, Other),
  put_piece(Board, Next, Column, Row, empty),
  put_piece(Next, NewBoard, C, R, piece(Piece, Color)).


% Move knight
dif(A, B, D) :- B is A + D,valid(B).
dif(A, B, D) :- B is A - D,valid(B).
knight_jump(Column, Row, C, R) :- dif(Column, C, 2), dif(Row, R, 1), valid(C), valid(R).
knight_jump(Column, Row, C, R) :- dif(Column, C, 1), dif(Row, R, 2), valid(C), valid(R).
move_knight(Position, NextPosition) :-  get_board(Position, Board),
                                        get_turn(Position, Turn),
                                        find_piece(Board, Column, Row, piece(knight, Turn)),
                                        knight_jump(Column, Row, C, R),
                                        is_empty(Board, C, R),
                                        put_piece(Board, Next, Column, Row, empty),
                                        put_piece(Next, Nextboard, C, R, piece(knight, Turn)),
                                        set_enpassant(Position, Positioner, ['-']),
                                        increment_halfmove(Positioner, Positionest),
                                        set_board(Positionest, Nextboard, NewPosition),
                                        swap(NewPosition, NextPosition).

move_knight(Position, NextPosition) :-  get_board(Position, Board),
                                        get_turn(Position, Turn),
                                        get_castling(Position, Castling),
                                        find_piece(Board, Column, Row, piece(knight, Turn)),
                                        set_halfmove(Position, Positioner, 0),
                                        set_enpassant(Positioner, Positionest, ['-']),
                                        knight_jump(Column, Row, C, R),
                                        take_piece(Board, NewBoard, Castling, Cas, Column, Row
                                        set_board(Positionest, NewBoard, NewPosition),
                                        set_castling(NewPosition, NewerPosition, Cas),
                                        swap(NewerPosition, NextPosition).

% Move Pawn
pawn_jump(Row, R, white) :- R is Row - 1, valid(R).
pawn_jump(Row, R, black) :- R is Row + 1, valid(R).
pawn_double_jump(7, 5, white).
pawn_double_jump(2, 4, black).
pawn_jump_promo(2, 1, white).
pawn_jump_promo(7, 8, black).

% Promotie
move_pawn(Position, NextPosition) :-    get_board(Position, Board),
```

```
                                    get_turn(Position, Turn),
                                    find_piece(Board, Column, Row, piece(pawn, Turn)),
                                    pawn_jump_promo(Row, R, Turn),
                                    is_empty(Board, Column, R), !,
                                    put_piece(Board, Next, Column, Row, empty),
                                put_piece(Next, Nextboard, Column, R, piece(Piece, Turn)),
                                    is_piece(Piece),
                                    set_enpassant(Position, Positioner, ['-']),
                                    set_halfmove(Positioner, Positionest, 0),
                                    set_board(Positionest, Nextboard, NewPosition),
                                    swap(NewPosition, NextPosition).

move_pawn(Position, NextPosition) :-    get_board(Position, Board),
                                    get_turn(Position, Turn),
                                    get_castling(Position, Castling),
                                    find_piece(Board, Column, Row, piece(pawn, Turn)),
                                    pawn_jump_promo(Row, R, Turn),
                                    dif(Column, C, 1),
          take_piece(Board, NewBoard, Castling, Cas, Column, Row, C, R, Piece, Turn),
                                    !,is_piece(Piece),
                                    set_enpassant(Position, Positioner, ['-']),
                                    set_halfmove(Positioner, Positionest, 0),
                                    set_board(Positionest, NewBoard, NewPosition),
                                    set_castling(NewPosition, NewerPosition, Cas),
                                    swap(NewerPosition, NextPosition).

% Dubbele sprong zet
move_pawn(Position, NextPosition) :-    get_board(Position, Board),
                                    get_turn(Position, Turn),
                                    find_piece(Board, Column, Row, piece(pawn, Turn)),
                                    pawn_double_jump(Row, R, Turn),
                                    is_empty(Board, Column, R),
                                    is_empty(Board, Column, Middle),
                                    Middle is (Row + R)/2,
                                    put_piece(Board, Next, Column, Row, empty),
                                 put_piece(Next, Nextboard, Column, R, piece(pawn, Turn)),
                                    set_enpassant(Position, Positioner, [Column, R2]),
                                    R2 is 9 - Middle,
                                    set_halfmove(Positioner, Positionest, 0),
                                    set_board(Positionest, Nextboard, NewPosition),
                                    swap(NewPosition, NextPosition).

% Gewone zet
move_pawn(Position, NextPosition) :-    get_board(Position, Board),
                                    get_turn(Position, Turn),
                                    find_piece(Board, Column, Row, piece(pawn, Turn)),
                                    pawn_jump(Row, R, Turn),
```

```
                                        is_empty(Board, Column, R),
                                        put_piece(Board, Next, Column, Row, empty),
                                    put_piece(Next, Nextboard, Column, R, piece(pawn, Turn)),
                                        set_enpassant(Position, Positioner, ['-']),
                                        set_halfmove(Positioner, Positionest, 0),
                                        set_board(Positionest, Nextboard, NewPosition),
                                        swap(NewPosition, NextPosition).

move_pawn(Position, NextPosition) :-    get_board(Position, Board),
                                        get_turn(Position, Turn),
                                        get_castling(Position, Castling),
                                        find_piece(Board, Column, Row, piece(pawn, Turn)),
                                        pawn_jump(Row, R, Turn),
                                        dif(Column, C, 1),
                    take_piece(Board, NewBoard, Castling, Cas, Column, Row, C, R, pawn, Turn),
                                        set_enpassant(Position, Positioner, ['-']),
                                        set_halfmove(Positioner, Positionest, 0),
                                        set_board(Positionest, NewBoard, NewPosition),
                                        set_castling(NewPosition, NewerPosition, Cas),
                                        swap(NewerPosition, NextPosition).
% Enpassant
move_pawn(Position, NextPosition) :-    get_board(Position, Board),
                                        get_turn(Position, Turn),
                                        get_passant(Position, [R, C]),
                                        find_piece(Board, Column, Row, piece(pawn, Turn)),
                                        pawn_jump(Row, R, Turn),
                                        dif(Column, C, 1),
                                        put_piece(Board, Next, Column, Row, empty),
                                        put_piece(Next, Nextish, C, Row, empty),
                                        put_piece(Nextish, NewBoard, C, R, piece(pawn, Turn)),
                                        set_enpassant(Position, Positioner, ['-']),
                                        set_halfmove(Positioner, Positionest, 0),
                                        set_board(Positionest, NewBoard, NewerPosition),
                                        swap(NewerPosition, NextPosition).

% Dit zijn de stukken die gebruikt mogen worden bij het promoveren
is_piece(queen).
is_piece(rook).
is_piece(knight).
is_piece(bishop).

% Move Rook
valid_rook_move(Board, Row, Column, R2, Column, Turn, Halfmove) :-
  R is Row + 1, valid(R), valid_rook_move_h(Board, R, Column, R2, Column, r, Turn, Halfmove).
valid_rook_move(Board, Row, Column, R2, Column, Turn, Halfmove) :-
  R is Row - 1, valid(R), valid_rook_move_h(Board, R, Column, R2, Column, l, Turn, Halfmove).
valid_rook_move(Board, Row, Column, Row, C2,Turn, Halfmove) :-
```

```
  C is Column + 1, valid(C), valid_rook_move_v(Board, Row, C, Row, C2, u, Turn, Halfmove).
valid_rook_move(Board, Row, Column, Row, C2,Turn, Halfmove) :-
  C is Column - 1, valid(C), valid_rook_move_v(Board, Row, C, Row, C2, d, Turn, Halfmove).

% Wanneer de eerste move horizontaal was blijven we adhv deze relatie horizontale
% vakjes vinden. Er is een variabele voor links/rechts
valid_rook_move_h(Board, Row, Column, Row, Column, _, _, inc) :-
  is_empty(Board, Column, Row).
valid_rook_move_h(Board, Row, Column, Row, Column, _, Turn, reset) :-
  get_piece(Board, Row, Column, piece(_, Other)), other_color(Turn, Other).
valid_rook_move_h(Board, Row, Column, R1, Column, r, Turn, Halfmove) :-
  is_empty(Board, Column, Row), R is Row + 1, valid(R),
  valid_rook_move_h(Board, R, Column, R1, Column, r, Turn, Halfmove).
valid_rook_move_h(Board, Row, Column, R1, Column, l, Turn, Halfmove) :-
  is_empty(Board, Column, Row), R is Row - 1, valid(R),
  valid_rook_move_h(Board, R, Column, R1, Column, l, Turn, Halfmove).

% Wanneer de eerste move verticaal was blijven we adhv deze relatie verticale vakjes
% vinden, er is een variabele voor up/down
valid_rook_move_v(Board, Row, Column, Row, Column, _, _, inc) :-
  is_empty(Board, Column, Row).
valid_rook_move_v(Board, Row, Column, Row, Column, _, Turn, reset) :-
  get_piece(Board, Row, Column, piece(_, Other)), other_color(Turn, Other).
valid_rook_move_v(Board, Row, Column, Row, C1, u, Turn, Halfmove) :-
  is_empty(Board, Column, Row), C is Column + 1, valid(C),
  valid_rook_move_v(Board, Row, C, Row, C1, u, Turn, Halfmove).
valid_rook_move_v(Board, Row, Column, Row, C1, d, Turn, Halfmove) :-
  is_empty(Board, Column, Row), C is Column - 1, valid(C),
  valid_rook_move_v(Board, Row, C, Row, C1, d, Turn, Halfmove).

startposition_rook(Board, Turn, 1, 1, 'q') :- find_piece(Board, 1, 1, piece(rook, Turn)).
startposition_rook(Board, Turn, 8, 1, 'k') :- find_piece(Board, 8, 1, piece(rook, Turn)).
startposition_rook(Board, Turn, 8, 8, 'K') :- find_piece(Board, 8, 8, piece(rook, Turn)).
startposition_rook(Board, Turn, 1, 8, 'Q') :- find_piece(Board, 1, 8, piece(rook, Turn)).

% Hier gebeurt er niets bijzonders (stuk vinden, zet genereren en uitvoeren),
% alleen zal de rokade-aanduider ook aangepast worden
move_rook(Position, NextPosition) :-    get_board(Position, Board),
                                        get_turn(Position, Turn),
                                        startposition_rook(Board, Turn, Column, Row, Del),
                                        get_castling(Position, Castling),
                                        delete_castling(Del, Castling, Cas),
                                        set_castling(Position, NewerPosition, Cas),
                                        valid_rook_move(Board, Row, Column, R, C, Turn, inc),
                                        put_piece(Board, Next, Column, Row, empty),
                                        put_piece(Next, Nextboard, C, R, piece(rook, Turn)),
                                        increment_halfmove(NewerPosition, NewererPosition),
```

```
                                        set_enpassant(NewererPosition, NewerPositioner, ['-'])
                                        set_board(NewerPositioner, Nextboard, NewPosition),
                                        swap(NewPosition, NextPosition).

move_rook(Position, NextPosition) :-    get_board(Position, Board),
                                        get_turn(Position, Turn),
                                        startposition_rook(Board, Turn, Column, Row, Del),
                                        get_castling(Position, Castling),
                                        delete_castling(Del, Castling, Cas),
                                        set_castling(Position, NewerPosition, Cas),
                                        valid_rook_move(Board, Row, Column, R, C, Turn, reset)
                            take_piece(Board, NewBoard, Cas, Cas2, Column, Row, C, R, rook, Turn)
                                        set_halfmove(NewerPosition, NewererPosition, 0),
                                        set_enpassant(NewererPosition, NewerPositioner, ['-'])
                                        set_board(NewerPositioner, NewBoard, NewPosition),
                                        set_castling(NewPosition, Chewbacca, Cas2),
                                        swap(Chewbacca, NextPosition).


move_rook(Position, NextPosition) :-    get_board(Position, Board),
                                        get_turn(Position, Turn),
                                        find_piece(Board, Column, Row, piece(rook, Turn)),
                                        \+(startposition_rook(Board, Turn, Column, Row, _)),
                                        valid_rook_move(Board, Row, Column, R, C, Turn, inc),
                                        put_piece(Board, Next, Column, Row, empty),
                                        put_piece(Next, Nextboard, C, R, piece(rook, Turn)),
                                        increment_halfmove(Position, Poposition),
                                        set_enpassant(Poposition, Positioner, ['-']),
                                        set_board(Positioner, Nextboard, NewPosition),
                                        swap(NewPosition, NextPosition).

move_rook(Position, NextPosition) :-    get_board(Position, Board),
                                        get_turn(Position, Turn),
                                        get_castling(Position, Castling),
                                        find_piece(Board, Column, Row, piece(rook, Turn)),
                                        \+(startposition_rook(Board, Turn, Column, Row, _)),
                                        valid_rook_move(Board, Row, Column, R, C, Turn, reset)
                          take_piece(Board, NewBoard, Castling, Cas, Column, Row, C, R, rook, Turn)
                                        set_halfmove(Position, Poposition, 0),
                                        set_enpassant(Poposition, Positioner, ['-']),
                                        set_board(Positioner, NewBoard, NewPosition),
                                        set_castling(NewPosition, Chewbacca, Cas),
                                        swap(Chewbacca, NextPosition).

% Move Bishop
% De bishop is op dezelfde manier gedefinieerd als de toren, minus het rokeren
valid_bishop_move(Board, Row, Column, R, C, Turn, Halfmove) :-
```

```
  R1 is Row + 1, C1 is Column + 1, valid(R1), valid(C1),
  valid_bishop_move_u(Board, R1, C1, R, C, r, Turn, Halfmove).
valid_bishop_move(Board, Row, Column, R, C, Turn, Halfmove) :-
  R1 is Row + 1, C1 is Column - 1, valid(R1), valid(C1),
  valid_bishop_move_u(Board, R1, C1, R, C, l, Turn, Halfmove).
valid_bishop_move(Board, Row, Column, R, C, Turn, Halfmove) :-
  R1 is Row - 1, C1 is Column + 1, valid(C1), valid(R1),
  valid_bishop_move_d(Board, R1, C1, R, C, r, Turn, Halfmove).
valid_bishop_move(Board, Row, Column, R, C, Turn, Halfmove) :-
  R1 is Row - 1, C1 is Column - 1, valid(C1), valid(R1),
  valid_bishop_move_d(Board, R1, C1, R, C, l, Turn, Halfmove).

valid_bishop_move_u(Board, Row, Column, Row, Column, _, _, inc) :-
  is_empty(Board, Column, Row).
valid_bishop_move_u(Board, Row, Column, Row, Column, _, Turn, reset) :-
  get_piece(Board, Row, Column, piece(_, Other)), other_color(Turn, Other).
valid_bishop_move_u(Board, Row, Column, R, C, r, Turn, Halfmove) :-
  is_empty(Board, Column, Row), R1 is Row + 1, C1 is Column + 1,
    valid(R1), valid(C1), valid_bishop_move_u(Board, R1, C1, R, C, r, Turn, Halfmove).
valid_bishop_move_u(Board, Row, Column, R, C, l, Turn, Halfmove) :-
  is_empty(Board, Column, Row), R1 is Row + 1, C1 is Column - 1,
  valid(R1), valid(C1), valid_bishop_move_u(Board, R1, C1, R, C, l, Turn, Halfmove).

valid_bishop_move_d(Board, Row, Column, Row, Column, _, _, inc) :-
  is_empty(Board, Column, Row).
valid_bishop_move_d(Board, Row, Column, Row, Column, _, Turn, reset) :-
  get_piece(Board, Row, Column, piece(_, Other)), other_color(Turn, Other).
valid_bishop_move_d(Board, Row, Column, R, C, r, Turn, Halfmove) :-
  is_empty(Board, Column, Row), R1 is Row - 1, C1 is Column + 1,
  valid(R1), valid(C1), valid_bishop_move_d(Board, R1, C1, R, C, r, Turn, Halfmove).
valid_bishop_move_d(Board, Row, Column, R, C, l, Turn, Halfmove) :-
  is_empty(Board, Column, Row), R1 is Row - 1, C1 is Column - 1,
  valid(R1), valid(C1), valid_bishop_move_d(Board, R1, C1, R, C, l, Turn, Halfmove).

move_bishop(Position, NextPosition) :-  get_board(Position, Board),
                                        get_turn(Position, Turn),
                                        find_piece(Board, Column, Row, piece(bishop, Turn)),
                                        valid_bishop_move(Board, Row, Column, R, C, Turn, inc)
                                        put_piece(Board, Next, Column, Row, empty),
                                        put_piece(Next, Nextboard, C, R, piece(bishop, Turn)),
                                        increment_halfmove(Position, Poposition),
                                        set_enpassant(Poposition, Positioner, ['-']),
                                        set_board(Positioner, Nextboard, NewPosition),
                                        swap(NewPosition, NextPosition).

move_bishop(Position, NextPosition) :-  get_board(Position, Board),
                                        get_turn(Position, Turn),
```

```
                                    get_castling(Position, Castling),
                                    find_piece(Board, Column, Row, piece(bishop, Turn)),
                                    valid_bishop_move(Board, Row, Column, R, C, Turn, rese
                        take_piece(Board, NewBoard, Castling, Cas, Column, Row, C, R, bishop, Tu
                                    set_halfmove(Position, Poposition, 0),
                                    set_enpassant(Poposition, Positioner, ['-']),
                                    set_board(Positioner, NewBoard, NewPosition),
                                    set_castling(NewPosition, Chewbacca, Cas),
                                    swap(Chewbacca, NextPosition).
% Move Queen
% Om een zet te genereren voor de koningin genereren we alle zetten die zowel torens als loper
move_queen(Position, NextPosition) :-   get_board(Position, Board),
                                    get_turn(Position, Turn),
                                    find_piece(Board, Column, Row, piece(queen, Turn)),
                                    valid_rook_move(Board, Row, Column, R, C, Turn, inc),
                                    put_piece(Board, Next, Column, Row, empty),
                                    put_piece(Next, Nextboard, C, R, piece(queen, Turn)),
                                    increment_halfmove(Position, Poposition),
                                    set_enpassant(Poposition, Positioner, ['-']),
                                    set_board(Positioner, Nextboard, NewPosition),
                                    swap(NewPosition, NextPosition).

move_queen(Position, NextPosition) :-   get_board(Position, Board),
                                    get_turn(Position, Turn),
                                    get_castling(Position, Castling),
                                    find_piece(Board, Column, Row, piece(queen, Turn)),
                                    valid_rook_move(Board, Row, Column, R, C, Turn, reset)
                        take_piece(Board, NewBoard, Castling, Cas, Column, Row, C, R, queen, Turn)
                                    set_halfmove(Position, Poposition, 0),
                                    set_enpassant(Poposition, Positioner, ['-']),
                                    set_board(Positioner, NewBoard, NewPosition),
                                    set_castling(NewPosition, Chewbacca, Cas),
                                    swap(Chewbacca, NextPosition).

move_queen(Position, NextPosition) :-   get_board(Position, Board),
                                    get_turn(Position, Turn),
                                    find_piece(Board, Column, Row, piece(queen, Turn)),
                                    valid_bishop_move(Board, Row, Column, R, C, Turn, inc)
                                    put_piece(Board, Next, Column, Row, empty),
                                    put_piece(Next, Nextboard, C, R, piece(queen, Turn)),
                                    increment_halfmove(Position, Poposition),
                                    set_enpassant(Poposition, Positioner, ['-']),
                                    set_board(Positioner, Nextboard, NewPosition),
                                    swap(NewPosition, NextPosition).

move_queen(Position, NextPosition) :-   get_board(Position, Board),
                                    get_turn(Position, Turn),
```

```
                                        get_castling(Position, Castling),
                                        find_piece(Board, Column, Row, piece(queen, Turn)),
                                        valid_bishop_move(Board, Row, Column, R, C, Turn, rese
                        take_piece(Board, NewBoard, Castling, Cas, Column, Row, C, R, queen, Turn
                                        set_halfmove(Position, Poposition, 0),
                                        set_enpassant(Poposition, Positioner, ['-']),
                                        set_board(Positioner, NewBoard, NewPosition),
                                        set_castling(NewPosition, Chewbacca, Cas),
                                        swap(Chewbacca, NextPosition).

% Move King
filip(Y) :- dirk(X), move_king(X, Y).

% Voor de koning genereren we alle zetten op één plaats afstand. Ook worden de rokade variabel
valid_king_move(Board, Row, Column, R, C, _, inc) :-
  king_move(Row, Column, R, C), is_empty(Board, C, R).
valid_king_move(Board, Row, Column, R, C, Turn, reset) :-
  king_move(Row, Column, R, C), get_piece(Board, R, C, piece(_, Other)), other_color(Turn, Othe

move_king(Position, NextPosition) :-    get_board(Position, Board),
                                        get_turn(Position, Turn),
                                        find_piece(Board, Column, Row, piece(king, Turn)),
                                        valid_king_move(Board, Row, Column, R, C, Turn, inc),
                                        put_piece(Board, Next, Column, Row, empty),
                                        put_piece(Next, Nextboard, C, R, piece(king, Turn)),
                                        increment_halfmove(Position, Poposition),
                                        set_enpassant(Poposition, Positioner, ['-']),
                                        set_board(Positioner, Nextboard, NewPosition),
                                        disable_castling(NewPosition, NewerPosition),
                                        swap(NewerPosition, NextPosition).

move_king(Position, NextPosition) :-    get_board(Position, Board),
                                        get_turn(Position, Turn),
                                        get_castling(Position, Castling),
                                        find_piece(Board, Column, Row, piece(king, Turn)),
                                        valid_king_move(Board, Row, Column, R, C, Turn, reset)
                        take_piece(Board, NewBoard, Castling, Cas, Column, Row, C, R, king, Turn)
                                        set_halfmove(Position, Poposition, 0),
                                        set_enpassant(Poposition, Positioner, ['-']),
                                        set_board(Positioner, NewBoard, NewPosition),
                                        set_castling(NewPosition, NewerPosition , Cas),
                                        disable_castling(NewerPosition, Chewbacca),
                                        swap(Chewbacca, NextPosition).

% Rokeren is de max
move_king(Position, NextPosition) :-  get_board(Position, Board),
                                      get_turn(Position, white),
```

```
                                get_castling(Position, Castling),
                                check_castling(Castling, 'Q'),
                                put_king_for_castlecheck(Position, 5, 8, 4, 8, white),
                                put_piece(Board, Next, 5, 8, empty),
                                put_piece(Next, Nexter, 1, 8, empty),
                                put_piece(Nexter, Nexert, 3, 8, piece(king, white)),
                                put_piece(Nexert, Nextboard, 4, 8, piece(rook, white)),
                                increment_halfmove(Position, Poposition),
                                set_enpassant(Poposition, Positioner, ['-']),
                                set_board(Positioner, Nextboard, NewPosition),
                                disable_castling(NewPosition, NewerPosition),
                                swap(NewerPosition, NextPosition).

move_king(Position, NextPosition) :-  get_board(Position, Board),
                                get_turn(Position, white),
                                get_castling(Position, Castling),
                                check_castling(Castling, 'K'),
                                put_king_for_castlecheck(Position, 5, 8, 6, 8, white),
                                put_piece(Board, Next, 5, 8, empty),
                                put_piece(Next, Nexter, 8, 8, empty),
                                put_piece(Nexter, Nexert, 7, 8, piece(king, white)),
                                put_piece(Nexert, Nextboard, 6, 8, piece(rook, white)),
                                increment_halfmove(Position, Poposition),
                                set_enpassant(Poposition, Positioner, ['-']),
                                set_board(Positioner, Nextboard, NewPosition),
                                disable_castling(NewPosition, NewerPosition),
                                swap(NewerPosition, NextPosition).

move_king(Position, NextPosition) :-  get_board(Position, Board),
                                get_turn(Position, black),
                                get_castling(Position, Castling),
                                check_castling(Castling, 'q'),
                                put_king_for_castlecheck(Position, 5, 1, 4, 1, black),
                                put_piece(Board, Next, 5, 1, empty),
                                put_piece(Next, Nexter, 1, 1, empty),
                                put_piece(Nexter, Nexert, 3, 1, piece(king, black)),
                                put_piece(Nexert, Nextboard, 4, 1, piece(rook, black)),
                                increment_halfmove(Position, Poposition),
                                set_enpassant(Poposition, Positioner, ['-']),
                                set_board(Positioner, Nextboard, NewPosition),
                                disable_castling(NewPosition, NewerPosition),
                                swap(NewerPosition, NextPosition).

move_king(Position, NextPosition) :-  get_board(Position, Board),
                                get_turn(Position, black),
                                get_castling(Position, Castling),
                                check_castling(Castling, 'K'),
```

```
                                    put_king_for_castlecheck(Position, 5, 1, 6, 1, black),
                                    put_piece(Board, Next, 5, 1, empty),
                                    put_piece(Next, Nexter, 8, 1, empty),
                                    put_piece(Nexter, Nexert, 7, 1, piece(king, black)),
                                    put_piece(Nexert, Nextboard, 6, 1, piece(rook, black)),
                                    increment_halfmove(Position, Poposition),
                                    set_enpassant(Poposition, Positioner, ['-']),
                                    set_board(Positioner, Nextboard, NewPosition),
                                    disable_castling(NewPosition, NewerPosition),
                                    swap(NewerPosition, NextPosition).


king_move(Row, Column, R, Column) :- dif(Row, R, 1).
king_move(Row, Column, Row, C) :- dif(Column, C, 1).
king_move(Row, Column, R, C) :- dif(Column, C, 1), dif(Row, R, 1).

put_king_for_castlecheck(Position, C1, R1, C2, R2, Turn) :-   get_board(Position, Board),
                                                              is_empty(Board, C2, R2),
                                                    put_piece(Board, Next, C1, R1, empty),
                                          put_piece(Next, NextBoard, C2, R2, piece(king, Turn)),
                                                  set_board(Position, NextBoard, NextPosition),
                                                              swap(NextPosition, Swap),
                                                              \+(invalid_position(Swap)).

% Disable castling for one particular color, typically when the king moves
disable_castling([Board, white, Castling | Tail], [Board, white, C | Tail]) :-
  delete_castling('K', Castling, C1), delete_castling('Q', C1, C).
disable_castling([Board, black, Castling | Tail], [Board, black, C | Tail]) :-
  delete_castling('k', Castling, C1), delete_castling('q', C1, C).

% Delete one castlingchar from the options
delete_castling(_, '-', '-') :- !.
delete_castling(_, [], []) :- !.
delete_castling(K, [K], '-') :- !.
delete_castling(Term, [Term|Tail], Tail) :- !.
delete_castling(Term, [Head|Tail], [Head|Result]) :- delete_castling_char(Term, Tail, Result).
delete_castling_char(_, [], []) :- !.
delete_castling_char(K, [K], []) :- !.
delete_castling_char(Term, [Term|Tail], Tail) :- !.
delete_castling_char(Term, [Head|Tail], [Head|Result]) :- delete_castling_char(Term, Tail, Resu

% See if castling is still possible
check_castling([A |_], A).
check_castling([_|B], C) :- check_castling(B, C).
```

parser.pl

```
:- module(parser, [parse/2]).
% Deze parse wordt enkel gebruikt om een FEN-string naar een voorstelling om te zetten
parse(Fen, Voorstelling) :- ground(Fen), string_chars(Fen, Chars), position(Voorstelling, Chars
% Deze dan weer enkel om een voorstelling naar een Fen-string om te zetten
parse(Fen, Voorstelling) :- position(Voorstelling, Chars, []), string_chars(F, Chars), atom_st

% De interne voorstelling is een lijst met Board, Turn, ..., adhv een dcg parsen we de string.
position([Board, Turn, Castling, Passant, Half, Full]) -->
  board(Board),
  space,
  turn(Turn),
  space,
  castling_rights(Castling),
  space,
  passant_square(Passant),
  space,
  integer(Half),
  space,
  integer(Full).

% Dit is een dcg om een bord te parsen. Een bord is een relatie rows tussen 8 row-objecten
board(rows(R1, R2, R3, R4, R5, R6, R7, R8)) -->
  row(R1), [/],
  row(R2), [/],
  row(R3), [/],
  row(R4), [/],
  row(R5), [/],
  row(R6), [/],
  row(R7), [/],
  row(R8).

% Hier wordt een row op een vrij eenvoudige, maar snelle, manier geparst.
row([]) --> [].
row([empty, empty, empty, empty, empty, empty, empty, empty| T]) --> ['8'], row(T).
row([empty, empty, empty, empty, empty, empty, empty| T]) --> ['7'], row(T).
row([empty, empty, empty, empty, empty, empty| T]) --> ['6'], row(T).
row([empty, empty, empty, empty, empty| T]) --> ['5'], row(T).
row([empty, empty, empty, empty| T]) --> ['4'], row(T).
row([empty, empty, empty| T]) --> ['3'], row(T).
row([empty, empty| T]) --> ['2'], row(T).
row([empty| T]) --> ['1'], row(T).
row([B|T]) --> piece(B), !, row(T).

% Dit is om de zwarte stukken te kunnen parsen
piece(piece(king, black)) --> [k].
```

```
piece(piece(queen, black)) --> [q].
piece(piece(rook, black)) --> [r].
piece(piece(bishop, black)) --> [b].
piece(piece(knight, black)) --> [n].
piece(piece(pawn, black)) --> [p].

% En dit voor de witte
piece(piece(king, white)) --> ['K'].
piece(piece(queen, white)) --> ['Q'].
piece(piece(rook, white)) --> ['R'].
piece(piece(bishop, white)) --> ['B'].
piece(piece(knight, white)) --> ['N'].
piece(piece(pawn, white)) --> ['P'].

% Hier wordt de turn geparst
turn(white) --> [w].
turn(black) --> [b].

% De rokade mogelijkheden houden we karakter per karakter bij, om aanpassingen gemakkelijker t
castling_rights(-) --> [-], !.
castling_rights([]) --> [].
castling_rights([H|T]) --> !, castling_rights_char(H), castling_rights(T).

castling_rights_char('K') --> ['K'].
castling_rights_char('Q') --> ['Q'].
castling_rights_char(k) --> [k].
castling_rights_char(q) --> [q].

% Hier wordt het vakje voor de enpassantregel bijgehouden
passant_square([-]) --> [-].
passant_square([C, D2]) --> [Char, D], {sub_string('xabcdefgh', C, 1, _, Ch),
                        atom_string(Char, Ch), atom_number(D, D2)} .

%Hulpmiddel voor het parsen van de fen
space --> [' '], !.

% Getallen bestaande uit meer dan een cijfer parsen met een dcg ging niet met
% mijn originele functie, deze kan het wel
integer(N) -->
  {nonvar(N), !,
  atom_chars(N, Codes)},
  Codes.


integer(N, Before, After) :-
  var(N),
  digits(Codes, Before, After),
```

```
number_codes(N, Codes).

digits([D|T]) -->
  digit(D), !,
  digits(T).
digits([]) -->
  [].

digit(D) -->
  [D],
  {
    code_type(D, digit)
  }.
```

min_max.pl

```
:- module(min_max, [minimax/3]).
:- use_module(chess, [move/2, value/2]).

minimax(Pos, Next, Depth) :- minimax(Depth, Pos, _, Next).

% Wanneer diepte 0  bereikt is doen we gewoon verder met de huidige spelwaarde
minimax(0, Position, Value, _) :- !,
      value(Position, Value).

% In het andere geval
minimax(D, Position, Value, Move) :-
      D > 0,
      D1 is D - 1,
      findall(M, move(Position, M), Positions),
      minimax(Positions, Position, D1, -10000, nil, Value, Move).

% Wanneer er geen moves mogelijk zijn is de huidige beste de beste
minimax([], _, _, Value, Best, Value, Best).

% Wanneer er wel mogelijk zijn gaat de beste afhangen van zijn score
minimax([NextPosition|Positions], Position, D, Value0, _, BestValue, BestMove):-
      minimax(D, NextPosition, OppValue, _OppMove),
      Value is -OppValue,
      Value >= Value0,
      minimax(Positions,Position,D, Value ,NextPosition ,BestValue,BestMove), !.

minimax([NextPosition|Positions],Position,D, Value0, Move0, BestValue, BestMove):-
      minimax(D, NextPosition, OppValue, _OppMove),
      Value is -OppValue,
      Value < Value0,
      minimax(Positions,Position,D, Value0,Move0,BestValue,BestMove), !.
```