FACULTEIT WETENSCHAPPEN

BACHELOR OF SCIENCE IN INFORMATICA

2020-2021

LOGISCH PROGRAMMEREN - PROJECT
SCHAAKCOMPUTER

Maarten Van Neyghem

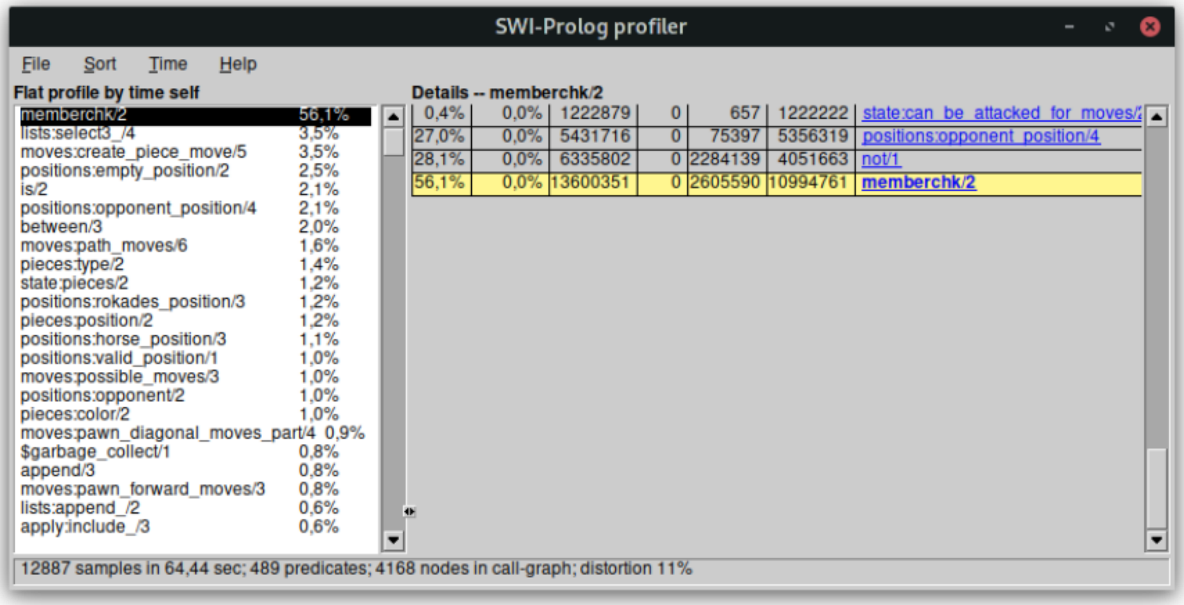
UNIVERSITEIT GENT

1 Interne Bord voorstelling

De interne bord voorstelling heeft een reeks van aanpassing gehad doorheen het ontwikkelen van het project. Deze aanpassingen werden gedaan voor performantie of om een zeer grote ariteit te voorkomen van de verschillende predicaten.

1.1 Oudere bord voorstellingen

In de oudere versies van de schaakcomputer werd gebruik gemaakt van een lijst van stukken. Dit leek initieel een goed idee, aangezien er gemakkelijk een stuk aan het bord kon toegevoegd of verwijderd worden. Deze voorstelling zorgde echter voor enkele performantie problemen. In een lijst hebben stukken geen vaste positie en moet er dus gezocht worden door de lijst om een stuk te vinden op een bepaalde positie. Dit zorgde voor een grote bottleneck bij alpha-beta snoeien, zoals te zien is in de profiler:



Figuur 1: Trace verkregen via de SwiPl profiler: `profile(predicate)`.

Op deze trace is duidelijk te zien dat de bottleneck het zoeken in de lijst is. Om die reden werd deze bordlayout vervangen door een systeem met compound-terms.

1.2 Huidige bord voorstelling

De huidige bordvoorstelling maakt gebruik van een structure die verschillende componenten heeft met informatie over de huidige state van het spel. Deze structure komt vaak voor in de code onder de naam `State`. Een state structure ziet er uit als volgt:

```
state(Board, CurrentPlayer, Rokades, Passant)
```

- `Board`: Dit bevat een ander structure die het huidige bord voorstelt. Het bord bestaat uit 8 substructures die elk een rij voorstellen en die elk 8 argumenten bevatten, wat de posities voorstelt. De positie in de eerste structure stelt dus de rij voor en de positie in de 2de structure de kolom. Indien er geen stuk staat op een bepaalde positie wordt `none` gebruikt.

Voorbeeld:

```
rows(
    row(none, none, none, none, none, none, none, piece(white, king, 8/8)),
    row(none, none, none, none, none, none, none, none),
    row(none, none, none, none, none, none, none, none),
    row(none, none, none, none, none, none, none, none),
    row(none, none, none, none, none, none, none, none),
    row(none, none, none, none, none, none, none, none),
    row(none, none, none, none, none, none, none, none),
    row(piece(black, king, 1/1), none, none, none, none, none, none, none)
)
```

Een stuk is opnieuw zelf een structure en heeft volgende vorm: `piece(Color, Type, X/Y)`. De positie van het stuk wordt herhaald in de structure zodat het mogelijk is om stukken door te geven aan andere predicaten als een geheel, zonder nog eens apart de positie mee te moeten geven.

- `CurrentPlayer`: Dit is de kleur van de speler die de volgende zet mag doen. De waarde kan het atom `white` of `black` zijn.
- `Rokades`: Dit is een lijst met mogelijke rokades. Een rokade is opnieuw zelf een structure en is van de vorm: `rokade(Color, long|short)`, waarbij de atoms `long` en `short` respectievelijk voor de lange of korte rokade staan.
- `Passant`: Dit is de en-passant mogelijkheid die kan gebruikt worden om de pion van de tegenstander aan te vallen met een en-passant zet. Dit is opnieuw zelf een structure: `passant(Color, X/Y)`. Indien er geen en-passant mogelijkheid is wordt `none` gebruikt.

In de `state.pl` module worden predicaten voorzien om makkelijk de verschillende structures uit de state te halen. Op die manier kan de hele state van een spel in 1 variabele doorgegeven worden aan de verschillende predicaten, wat een te grote ariteit kan voorkomen.

2 Algoritme

Om de volgende beste zet te bepalen wordt gebruik gemaakt van alpha-beta snoeien met een beperkte diepte. Elke top in de spelboom stelt een bepaalde state voor, die bekomen wordt door een van de mogelijks volgende zetten uit te voeren op de vorige state. Uitwerken van de spelboom tot diepte van 3 is mogelijk met het huidige algoritme tijdens het spelen tegen de random speler binnen de opgegeven tijdslimiet.

2.1 Alpha-Beta Snoeien

Om de best mogelijke volgende zet te gebruiken wordt alpha-beta snoeien gebruikt met een depth-first search zoekfunctie met een beperkte diepte.

Het predicaat `alphabeta` implementeert dit algoritme *(src/alphabeta.pl op lijn 13-48)*. Alle mogelijke volgende states vanuit een huidige state in de spelboom worden geëvalueerd en het predicaat `best` wordt geëvalueerd om de best mogelijke volgende state te bepalen vanuit die top/state. Een blad in de spelboom kan ofwel het einde van een spel voorstellen, waarbij een van de spelers dus schaakmat of in patstelling staat ofwel wanneer de opgegeven diepte bereikt wordt. Elke blad krijgt een score die gebruikt wordt om de best mogelijke volgende zet te bepalen.

De lijst met mogelijk volgende states wordt geëvalueerd door het predicaat `all_possible_states` *(src/state.pl op lijn 75-105)*. Dit predicaat vraagt alle mogelijke moves op voor elk stuk van de huidige speler, via het predicaat `all_possible_moves` *(src/move.pl op lijn 79-93)* en zal voor alle moves , die niet tot schaak leiden, de corresponderende state en het resultaat unificeren in een lijst.

2.2 Bepalen van de best mogelijke volgende state

Het bepalen van de best mogelijke volgende state gebeurt in het `best` predicaat *(src/alphabeta.pl op lijn 54-65)*. Dit predicaat neemt een lijst met states en gaat deze één voor één uitwerken om zo de state met de relatief beste score eruit halen. De states worden verder uitgewerkt door `alphabeta` op te roepen voor de huidige state.

Indien er meerdere states zijn, wordt het `cut` predicaat geëvalueerd *(src/alphabeta.pl op lijn 71-90)*. Dit predicaat gaat takken die buiten de reeds gedefinieerde boven- en ondergrens vallen wegsnoeien en dus voorkomen dat deze verder uitgewerkt worden. Een tak wordt weggesnoeid indien:

- **Huidige speler is aan de beurt**: het algoritme moet maximaliseren, indien de waarde groter is of gelijk aan de bovengrens wordt de tak weggesnoeid.
- **Tegenstander is aan de beurt**: het algoritme moet minimaliseren, indien de waarde kleiner of gelijk aan de ondergrens wordt de tak weggesnoeid.

Verder wordt recursief de beste state uitgewerkt door telkens 2 states te vergelijken. Deze vergelijking gebeurt door het `best_of` predicaat te evalueren*(src/alphabeta.pl op lijn 122-175)*. Dit predicaat gaat 2 states vergelijken en de state met de hoogste score unificeren met `BestState` (en de respectievelijke score met `BestScore`).

`cut` gaat naast snoeien ook nog de bounds updaten, door het evalueren van het `update_bounds` predicaat *(src/alphabeta.pl op lijn 100-116)*.

2.3 Scorigsfunctie

</> Code: De code van deze functie is te vinden in src/alphabeta.pl op lijn 204-360.

Aangezien er gewerkt wordt met een dieptebeperking, waardoor een top in de spelboom niet altijd het einde van een spel voorstelt, is dus nood aan een scoringsfunctie die elke state een score kan geven.

De gebruikte scoringsfunctie is een *symmetric evaluation function*. Dit wil zeggen dat de huidige state onafhankelijk van de vorige states geëvalueerd wordt. De functie is dus symmetrisch aangezien 2 dezelfde states altijd dezelfde score zullen krijgen.

De scoring van een state gebeurt als volgt:

- **Bij schaakmat:**
  De hoogst/laagst mogelijke score wordt gegeven: 10000 wanneer de tegenstander schaakmat staat, -10000 wanneer de speler zelf schaakmat staat.

- **Bij patstelling:**
  Score 0 wordt gegeven, aangezien er een gelijkstand is.

- **Bij alle andere states:**
  Elk stuk krijgt een vaste waarde, afhankelijk van hoe belangrijk het stuk is om een spel te winnen. De scores zijn gebaseerd op het systeem van *Hans Berliner* [2], een wereldkampioen computerschaak. De score van een bepaalde speler is dan de som van de waardes van alle beschikbare stukken.

Stuk	Score
Koningin	8.8
Toren	5.5
Loper	3.33
Paard	3.2
Pion	afhankelijk van positie

Tabel 1: Waarde van elk stuk, gebaseerd op het systeem van *Hans Berliner* [2].

De score van een pion is afhankelijk van de positie op het bord. Initieel kregen pionnen een vaste waarde, waardoor deze vaak in het midden van het veld bleven staan en andere stukken niet de mogelijkheid kregen om zich meer naar voor te bewegen, om zo de stukken van de tegenstander aan te vallen. Stukken die zich meer naar voor bewegen krijgen een hogere score, met grotere scores dichter bij de middelste kolommen. Op die manier gaan de pionnen het spel openspelen, waardoor de andere stukken dus ook meer bewegingsmogelijkheid hebben. De specifieke waardes zijn te vinden in de code *(src/alphabeta.pl op lijn 353-360)*.

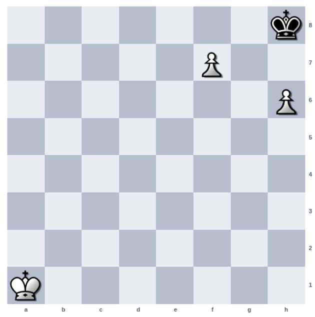
Voor beide spelers wordt de totale score berekend door een som te nemen van de stukken en hun corresponderende waardes. Deze 2 scores worden vervolgens van elkaar afgetrokken om de totaalscore van de state te krijgen:

$$score(State) = score(Maximaliserende\ Speler) - score(Minimaliserende\ Speler)$$

Indien de maximaliserende of minimaliserende speler de hoogste score heeft zal de score van de state respectievelijk positief of negatief zijn. *(src/alphabeta.pl op lijn 211)*

2.4 Voorbeeld

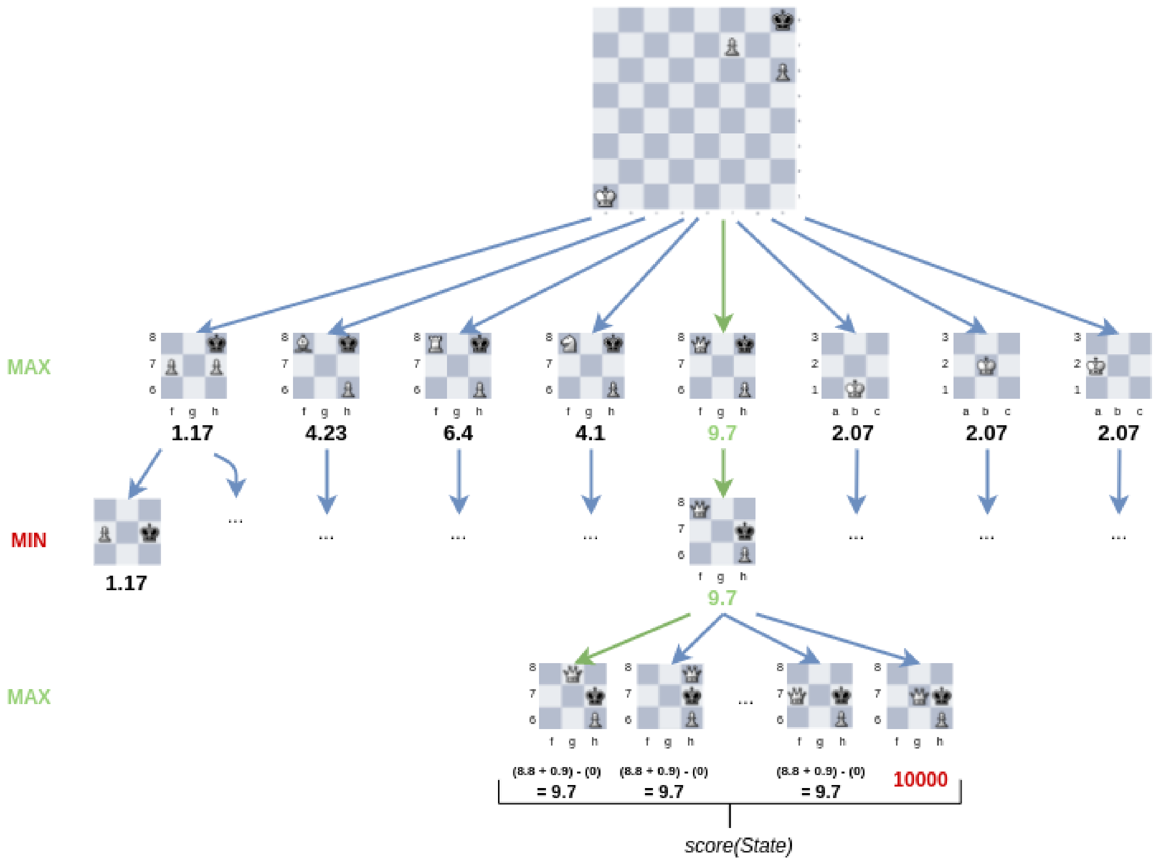
Met bord voorstelling in figuur 2 is het mogelijk om de zwarte koning in 2 zetten schaakmat te plaatsen. Dit kan bereikt worden door de witte pion op F7 naar F8 te verplaatsen en deze vervolgens te promoveren tot een koningin. De tegenstander heeft dan nog slechts 1 mogelijkheid om niet schaak te staan, door zich te verplaatsen van H8 naar H7. Vervolgens kan de witte speler zijn koningin verplaatsen van F8 naar G7 waardoor de zwarte speler schaakmat staat.



Figuur 2: Bord voorstelling voor het voorbeeld. De witte speler moet de volgende zet plaatsen.

De schaakcomputer kiest, bij een diepte van 3 als best mogelijke zet, om de de witte pion op F7 naar F8 te verplaatsen en deze vervolgens te promoveren tot een koningin.

De bekomen spelboom, met een diepte van 3, ziet er uit als volgt:



Figuur 3: Spelboom voor het voorbeeld

De schaakcomputer kiest voor de promotie van een pion naar een dame, omdat dit de hoogst mogelijke score geeft van alle mogelijke volgende states. Aangezien de 2de rij van de spelboom moet minimaliseren, wordt de score van de schaakmat op de 3de rij niet meegenomen als score van de bovenliggende top op de 2de rij. De schaakcomputer kiest echter nog steeds de tak waarin de snelst mogelijke schaakmat zich bevindt. Dit komt omdat de promotie tot een dame de hoogste score geeft en bij gevolg dus de grootste kans heeft om de zwarte koning schaakmat te zetten. Dit verklaart meteen de keuze van de voorgedefinieerde scores van de verschillende stukken: hoe meer mogelijke bewegingen een stuk kan hebben, hoe hoger de score.

3 Benchmarks

3.1 Tegen de random speler

Volgende benchmarks werden uitgevoerd tegen de random speler in `opgave/vsRandom`. Elke diepte werd 10 keer uitgevoerd en de gemiddeldes worden in volgende tabel genoteerd:

	Diepte 1	Diepte 2	Diepte 3
Aantal keren winst	10/10	10/10	10/10
Gemiddeld aantal zetten	28.8	15.2	10.1
Gemiddelde uitvoeringstijd	0.25s	0.74s	7.78s

Tabel 2: Aantal keren winst, aantal zetten en gemiddelde uitvoeringstijd voor elke diepte. Gemiddelde van 10 uitvoeringen.

Er is een duidelijke afname in gemiddeld aantal zetten tot het einde van het spel. Dit komt doordat de schaakcomputer slimmere beslissingen kan maken bij het dieper uitwerken van de spelboom. Dit maakt het mogelijk om zetten van de tegenstander te anticiperen.

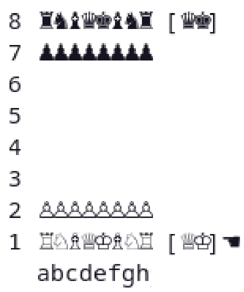
Een diepte van 4 was niet uitvoerbaar tegen de random speler wegens timeouts na bepaalde zetten.

3.2 Specifieke bord configuraties

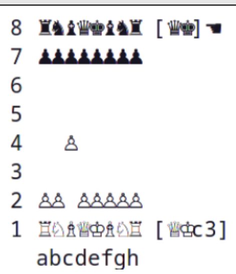
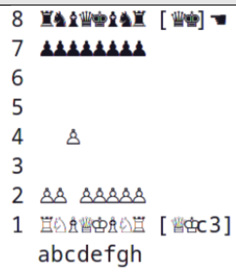
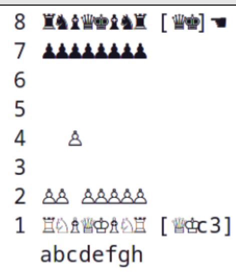
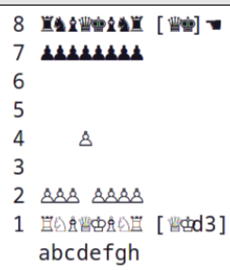
Specifieke bord voorstellingen werden geanalyseerd op uitvoeringstijd, geheugengebruik en de bekomen volgende zet.

3.2.1 Vanuit startpositie

Volgende benchmarks werden uitgevoerd vanuit de startpositie van het spel (figuur 4). Elk stuk is hierbij op de initiële positie en de witte speler is aan de beurt. De benchmark resultaten zijn te vinden in tabel 3.



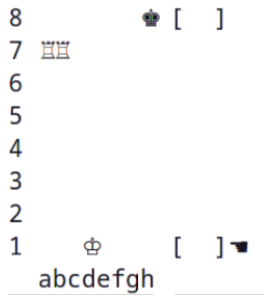
Figuur 4: Bord layout

	Diepte 1	Diepte 2	Diepte 3	Diepte 4
Geplaatste zet				
Uitvoeringstijd	0.23s	0.51s	2.78s	25.1s
Geheugengebruik	9.21 mb	9.34 mb	10.23 mb	10.27 mb

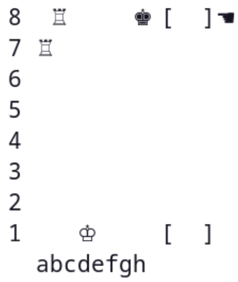
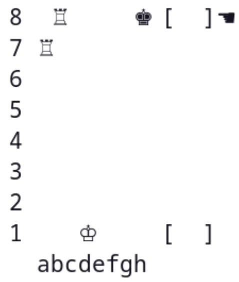
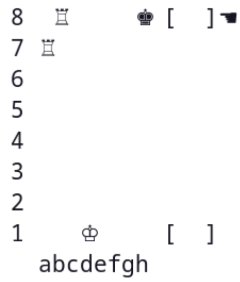
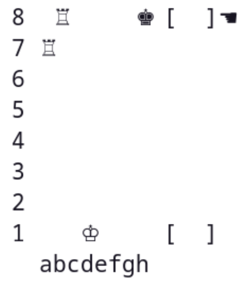
Tabel 3: Volgende zet, uitvoeringstijd en geheugengebruik voor elke diepte. Gemiddelde van 10 uitvoeringen.

3.2.2 Endgame

Volgende benchmarks werden uitgevoerd op een spelbord waarbij de tegenstander in 1 zet schaakmat geplaatst kan worden (figuur 5). De witte speler is momenteel aan de beurt en kan de koning van de tegenstander schaak zetten door 1 van zijn torens met 1 stap vooruit te zetten. De benchmark resultaten zijn te vinden in tabel 4.



Figuur 5: Bord layout

	Diepte 1	Diepte 2	Diepte 3	Diepte 4
Geplaatste zet				
Uitvoeringstijd	0.23s	0.24s	0.81s	1.35s
Geheugengebruik	9.05 mb	9.09 mb	9.30 mb	9.31 mb

Tabel 4: Volgende zet, uitvoeringstijd en geheugengebruik voor elke diepte. Gemiddelde van 10 uitvoeringen.

3.2.3 Willekeurige zet

Volgende benchmarks werden uitgevoerd op een willekeurig spelbord (figuur 6), gegenereerd door `FEN-generator` [1], waarbij de witte speler de volgende zet moet plaatsen. Volgens `nextchessmove.com` [3] is de best volgende zet het verplaatsen van het witte paard van d1 naar b2 en dus de toren op die positie te slaan. De benchmark resultaten zijn te vinden in tabel 5. Vanaf een diepte van 3 wordt de best mogelijke volgende zet bereikt door het algoritme.

Figuur 6: Bord layout

	Diepte 1	Diepte 2	Diepte 3	Diepte 4
Geplaatste zet				
Tijd	0.23s	0.69s	5.41s	56.7s
Geheugen	9.2 mb	9.42 mb	9.62 mb	11.15 mb

Tabel 5: Volgende zet, uitvoeringstijd en geheugengebruik voor elke diepte. Gemiddelde van 10 uitvoeringen.

4 Conclusie

De uiteindelijke schaakcomputer voldoet aan de opgegeven functionaliteitsnormen. Alle schaakbewegingen zijn ondersteund en de schaakcomputer kan tot een diepte van 3 snoeien via een alpha-beta algoritme binnen de opgelegde tijdslimieten van de random speler.

Elk predicaat heeft documentatie die volgende informatie bevat:

- **Een interface** met de mogelijke argumenten en hun initialisatie.

    +<variabele>: een variabele die reeds geïnitialiseerd moet zijn.

    -<variabele>: een variabele die nog niet geïnitialiseerd moet zijn en geünificeerd zal worden.

- **Uitleg over de functionaliteit** van het predicaat of de specifieke clause van dat predicaat.

De parser ondersteunt alle mogelijke borden en wordt voor het grootste deel hergebruikt voor het schrijven van de uitvoer (src/io/writer.pl op lijn 24).

Elke mogelijke zet heeft een test in de `tests` folder, samen met een aantal andere testen voor de verschillende functies van de verschillende modules.

4.1 Mogelijke verbeteringen

Hoewel de schaakcomputer aan alle functionele eisen voldoet, zijn er echter nog een aantal aanpassingen in de code mogelijk:

- **Geen append voor mogelijke moves:** In het move systeem worden mogelijke moves voor stukken geëvalueerd in een lijst. Wanneer er meerdere lijsten van moves gecombineerd worden is daarvoor een `append`-operatie nodig, wat negatief is voor de performantie. Indien de moves voor de verschillende stukken gebruik zouden maken van predicaten die een enkele move behandelen, in plaats van een lijst, zou van een enkele `findall` kunnen gebruikt worden om in 1 bewerking de lijst met moves voor een stuk te unificeren.

- **Opslaan van koningen in bord voorstelling:** Om alle mogelijke volgende states te bepalen moeten moves die tot schaak leiden weggefilterd worden. Om de koning op het huidige bord te vinden moeten alle stukken overlopen worden in $O(n)$-tijd. Wanneer beide koningen apart opgeslagen zouden worden kan dit in $O(1)$-tijd, wat voor een kortere uitvoeringstijd zal zorgen.

Doorvoeren van deze verbeteringen zou het mogelijk kunnen maken om tot een diepte van 4 te kunnen werken bij het alpha-beta snoeien. Ze vragen echter revisie van een groot deel van de code en het herschrijven van een groot deel van de testen. Om deze reden werd beslist om deze verbeteringen niet door te voeren en de huidige code in te dienen.

Referenties

[1] *Bernd's Random-FEN-Generator*. adres: http://bernd.bplaced.net/fengenerator/fengenerator.html (bezocht op 11-06-2021).

[2] *Chess piece relative value*. adres: https://en.wikipedia.org/wiki/Chess_piece_relative_value#Hans_Berliner's_system (bezocht op 12-06-2021).

[3] *Next Chess Move: The strongest online chess calculator*. adres: https://nextchessmove.com/?fen=rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR%5C%20w%5C%20KQkq%5C%20-%5C%200%5C%201 (bezocht op 11-06-2021).

Code

Listing 1: src/alphabeta.pl

```prolog
:- module(alphabeta, []).

:- use_module("move").
:- use_module("state").
:- use_module("position").
:- use_module("piece").


%! alphabeta(+Player, +CurrentState, +TraversedDepth, +MaxDepth, +LowerBound, +UpperBound, -BestState, -BestScore)
%
%  Alpha/beta pruning to determin the next best move.
%  Depth will specify the max recursion depth.
alphabeta(Player, CurrentState, MaxDepth, MaxDepth, _, _, CurrentState, BestScore) :- % Leaf: maximum depth is reached

    % Calculate the score for the current state
    score(Player, CurrentState, MaxDepth, MaxDepth, BestScore), !.

alphabeta(Player, CurrentState, TraversedDepth, MaxDepth, LowerBound, UpperBound, BestState, BestScore) :-  % Continue building the game tree

    % Determin all possible next states for the current state
    state:all_possible_states(CurrentState, NextStates),

    % Next States must not be empty (otherwise there is a checkmate or a stalemate)
    NextStates \= [],

    % Decrement the depth
    NextTraversedDepth is TraversedDepth + 1,

    % Find the best possible move
    best(Player, NextStates, NextTraversedDepth, MaxDepth, LowerBound, UpperBound, BestState, BestScore), !.

% This branch will only be reached if the above 2 variants of the predicate fail.
% This is only the case when all_possible_states is empty,
% which is checkmate for the current player.
alphabeta(Player, CurrentState, TraversedDepth, MaxDepth, _, _, CurrentState, BestScore) :- % Leaf: a player is checkmate
    state:currentcolor(CurrentState, CheckmatePlayer),

    % Make sure the king is check (otherwise a stalemate is reached)
    state:check(CurrentState, CheckmatePlayer),

    % Calculate the score fot the current state.
    score_checkmate(Player, CheckmatePlayer, TraversedDepth, MaxDepth, BestScore), !.

```

```prolog
alphabeta(Player, CurrentState, _, _, _, _, none, BestScore) :- % Leaf: a player is stalemate
    state:currentcolor(CurrentState, StalematePlayer),

    % Calculate the score fot the current state.
    score_stalemate(Player, StalematePlayer, BestScore), !.


%! best(+Player, +States, +TraversedDepth, +MaxDepth, +LowerBound, +UpperBound, -BestState, -BestScore)
%
%  Best state in a given list of states.
best(_, [], _, _, _, _, _, _). % Base case
best(Player, [State], TraversedDepth, MaxDepth, LowerBound, UpperBound, State, Score) :- % Single state, return state

    % Do minimax for the current state
    alphabeta(Player, State, TraversedDepth, MaxDepth, LowerBound, UpperBound, _, Score), !.
best(Player, [State | OtherStates], TraversedDepth, MaxDepth, LowerBound, UpperBound, BestState, BestScore) :-    % Multiple states, determin best state

    % Do minimax for the current state
    alphabeta(Player, State, TraversedDepth, MaxDepth, LowerBound, UpperBound, _, Score),

    % Cut or continue evaluation
    cut(Player, State, Score, OtherStates, TraversedDepth, MaxDepth, LowerBound, UpperBound, BestState, BestScore).


%! cut(+Player, +State, +Score, +OtherStates, +TraversedDepth, +MaxDepth, +LowerBound, +UpperBound, -BestState, -BestScore)
%
%  Cut of branches that will never lead to a result (Alpha/Beta-pruning)
cut(Player, State, Score, _, _, _, _, UpperBound, State, Score) :- % Maximizing player, cut-off
    max(State, Player),

    % Cut-off the branch if the score is larger than the upperbound
    Score >= UpperBound, !.
cut(Player, State, Score, _, _, _, LowerBound, _, State, Score) :- % Minimizing player, cut-off
    min(State, Player),

    % Cut-off the branch if the score is larger than the upperbound
    Score =< LowerBound, !.
cut(Player, State1, Score1, OtherStates, TraversedDepth, MaxDepth, LowerBound, UpperBound, BestState, BestScore) :-  % Continue evaluation

    % Update upper/lower bound
    update_bounds(Player, State1, Score1, LowerBound, UpperBound, NewLowerBound, NewUpperBound),
```

```

    % Continue evaluation of the other states
    best(Player, OtherStates, TraversedDepth, MaxDepth, NewLowerBound, NewUpperBound, State2, Score2),

    % Determin the best state of the 2 states
    best_of(Player, State1, State2, Score1, Score2, BestState, BestScore), !.


%! update_bounds(+Player, +State, +Score, +LowerBound, +UpperBound, -NewLowerBound, -NewUpperBound)
%
%  Update the upper & lowerbound, if necessary

% Update the lowerbound to the current score if:
% * Current player is maximizing player
% * Score is larger than the lowerbound
update_bounds(Player, State, Score, LowerBound, UpperBound, Score, UpperBound) :-
    max(State, Player),

    % Score must be larger than the lowerbound.
    Score > LowerBound, !.

% Update the upperbound to the current score if:
% * Current player is minimizing player
% * Score is smaller than the upperbound
update_bounds(Player, State, Score, LowerBound, UpperBound, LowerBound, Score) :-
    min(State, Player),

    % Score must be larger than the lowerbound.
    Score < UpperBound, !.

% Base case
update_bounds(_, _, _, LowerBound, UpperBound, LowerBound, UpperBound).


%! best_of(+Player, +State1, +State2, +Score1, +Score2, -BestState, -BestScore)
%
%  Best state between 2 states, based on their scores.
best_of(Player, State1, _, Score1, Score2, BestState, BestScore) :- % Maximizing player (Score 1 is largest)
    max(State1, Player),

    % Score 1 is largest
    Score1 >= Score2,

    % Update best state
    BestState = State1,
    BestScore = Score1.
best_of(Player, _, State2, Score1, Score2, BestState, BestScore) :- % Maximizing player (Score 2 is largest)
    max(State2, Player),

```

```
    % Score 2 is largest
    Score1 < Score2,

    % Update best state
    BestState = State2,
    BestScore = Score2.
best_of(Player, State1, _, Score1, Score2, BestState, BestScore) :- % Minimizing player (Score 1 is smallest)
    min(State1, Player),

    % Score 1 is smallest
    Score1 =< Score2,

    % Update best state
    BestState = State1,
    BestScore = Score1.
best_of(Player, _, State2, Score1, Score2, BestState, BestScore) :- % Minimizing player (Score 2 is smallest)
    min(State2, Player),

    % Score 2 is smallest
    Score1 > Score2,

    % Update best state
    BestState = State2,
    BestScore = Score2.


%! max(+State, +Player)
%
%  If the state is for the maximizing player
%  Since the state contains the player that can do the next move, the currentcolor must be different from the player.
max(State, Player) :-
    state:currentcolor(State, CurrentPlayer),
    CurrentPlayer \== Player.


%! min(+State, +Player)
%
%  If the state is for the minimizing player
%  Since the state contains the player that can do the next move, the currentcolor must be the same as the player.
min(State, Player) :-
    state:currentcolor(State, CurrentPlayer),
    CurrentPlayer == Player.


%! score(+Player, +State, +TraversedDepth, +MaxDepth, -Score)
%
%  Score for a given state.
%
%  This scoring predicate is a symmetric evaluation function that will score the current state of the board.
```

```
%  It does not keep track of previous states or boards and just evaluates the current state, as is.
%
%  - Each piece will receive a value based on it's importance in the game
%  - Pawns will be encouraged to advance. If pawns keep stuck on the central row, they will protect the king, but block all other pieces from advancing.
%
%  To score a state we subtract the player's score with the opponent's score
%  This way boards with a large difference will receive a higher score
%  => If the player has a higher score, the overal score will be positive
%  => If the opponent has a higher score, the overal score will be negative

score(Player, State, TraversedDepth, MaxDepth, Score) :- % Checkmate or stalemate

    % Player cannot do any more moves
    state:checkmate_or_stalemate(State),

    % Checkmate or stalemate
    score_checkmate_or_stalemate(Player, State, TraversedDepth, MaxDepth, Score).

score(Player, State, _, _, Score) :-                     % Symmetric evaluation scoring function

    % Score for the player
    score_player(Player, State, PlayerScore),

    % Score for the opponent
    piece:opponent(Player, OpponentPlayer),
    score_player(OpponentPlayer, State, OpponentScore),

    % Calculate the state score
    Score is PlayerScore - OpponentScore.

%! score_player(+Player, +State, -Score)
%
%  Score for a given state for a given player.
score_player(Player, State, Score) :-
    % Get the pieces for the given player
    state:color_pieces(State, Player, ColorPieces),

    % Evaluate every piece and add it's score to the scores
    score_pieces(ColorPieces, PiecesScore),

    % Add all the scores together
    Score = PiecesScore.


%! score_checkmate_or_stalemate(+Player, +State, +TraversedDepth, +MaxDepth, -Score)
%
%  Score when a given player is either checkmate or stalemate.
%  If the player is checkmate, return the checkmate score.
%  If the player is stalemate, return the stalemate score.
score_checkmate_or_stalemate(Player, State, TraversedDepth, MaxDepth, Score) :-   % Checkmate
```

```
    state:currentcolor(State, CurrentPlayer),

    % Make sure the king is check (otherwise a stalemate is reached)
    state:check(State, CurrentPlayer),

    % Checkmate score
    score_checkmate(Player, CurrentPlayer, TraversedDepth, MaxDepth, Score).

score_checkmate_or_stalemate(Player, State, _, _, Score) :-                       % Stalemate
    state:currentcolor(State, CurrentPlayer),

    % Checkmate score
    score_stalemate(Player, CurrentPlayer, Score).


%! score_checkmate(+Player, +CheckmatePlayer, +TraversedDepth, +MaxDepth, -Score)
%
%  Score when a given state is checkmate.
score_checkmate(Player, CheckmatePlayer, TraversedDepth, MaxDepth, Score) :-
    (
        Player == CheckmatePlayer, PartialScore = -10000
        ;
        PartialScore = 10000
    ),

    % Apply a depth penalty to the checkmate.
    % This will make sure the chosen nextmove will be based on the branch with the quickest possible checkmate.
    ScorePenalty is MaxDepth - TraversedDepth, % This value will become smaller when the traversed depth will increase
    Score is PartialScore + ScorePenalty.


%! score_stalemate(+Player, +StalematePlayer, -Score)
%
%  Score when a given state is stalemate.
score_stalemate(_, _, 0).


%! score_pieces(+Pieces, -Score)
%
%  Based on Hans Berliner's System.
%  Score a set of pieces.
score_pieces([Piece | Pieces], Score) :-

    % Score for the current piece
    score_piece(Piece, PieceScore),

    % Recursive call
    score_pieces(Pieces, PiecesScore),

    % Add the scores together
```

```
    Score is PieceScore + PiecesScore, !.
score_pieces([], 0).


%! score_piece(+Piece, -Score)
%
%  Score a single piece.
%  Scores are based on the values recommended by Hans Berliner's system (World
↪  Correspondence Chess Champion)

% Queen
score_piece(piece(_, queen, _), 8.8).

% Tower
score_piece(piece(_, tower, _), 5.1).

% Bishop
score_piece(piece(_, bishop, _), 3.33).

% Horse
score_piece(piece(_, horse, _), 3.2).

% Pawn
score_piece(piece(white, pawn, X/Y), Score) :-    % white: Get score from pawn table

    % Scoring table
    score_pawn_table(ScoringTable),

    % Row
    nth0(Y, ScoringTable, Row),

    % Score
    nth0(X, Row, Score).

score_piece(piece(black, pawn, X/Y), Score) :-    % black: Get score from pawn table
    XRev is 8 - X,
    YRev is 8 - Y,

    % Scoring table
    score_pawn_table(ScoringTable),

    % Row
    nth0(YRev, ScoringTable, Row),

    % Score
    nth0(XRev, Row, Score).

score_piece(piece(_, pawn, _), 1).                % Default value

% Default
score_piece(piece(_, _, _), 0).
```

```
%! score_pawn_table(+ScoringTable)
%
%  Scoring table for scoring pawns (from the perspective of the white player)
%
%  Pawns will receive a higher score as they advance.
%  This will prevent them from staying center, blocking other pieces to move forward.
%  Based on Hans Berliner's System.
score_pawn_table([
    [0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00],
    [0.90, 0.95, 1.05, 1.10, 1.10, 1.05, 0.95, 0.90],
    [0.90, 0.95, 1.05, 1.15, 1.15, 1.05, 0.95, 0.90],
    [0.90, 0.95, 1.10, 1.20, 1.20, 1.10, 0.95, 0.90],
    [0.97, 1.03, 1.17, 1.27, 1.27, 1.17, 1.03, 0.97],
    [1.06, 1.12, 1.25, 1.40, 1.40, 1.25, 1.12, 1.06]
]).
```

Listing 2: src/state.pl

```prolog
:- module(state, []).

:- use_module("move").
:- use_module("piece").
:- use_module("position").


%! board(+State, -Board)
%
%  Extract the board from the given state.
board(state(Board, _, _, _), Board).


%! currentcolor(+State, -CurrentColor)
%
%  Extract the current color (the player that can do the current move) from the given state.
currentcolor(state(_, CurrentColor, _, _), CurrentColor).


%! nextcolor(+State, -NextColor)
%
%  Extract the next color (the player that can do a move after the current player did a move) from the given state.
nextcolor(state(_, white, _, _), black).
nextcolor(state(_, black, _, _), white).


%! rokades(+State, -Rokades)
%
%  Extract the remaining rokades from the given state.
rokades(state(_, _, Rokades, _), Rokades).


%! passant(+State, -Passant)
%
%  Extract the en-passant possibility from the given state.
passant(state(_, _, _, Passant), Passant).


%! empty_state(+CurrentColor, +Rokades, +Passant, -State)
%
%  Create a state with an empty board.
empty_state(CurrentColor, Rokades, Passant, State) :-

    % Create an empty board
    Board = rows(
        row(none, none, none, none, none, none, none, none),
        row(none, none, none, none, none, none, none, none),
        row(none, none, none, none, none, none, none, none),
```

```
        row(none, none, none, none, none, none, none, none),
        row(none, none, none, none, none, none, none, none),
        row(none, none, none, none, none, none, none, none),
        row(none, none, none, none, none, none, none, none),
        row(none, none, none, none, none, none, none, none)
    ),

    % Create the state
    State = state(Board, CurrentColor, Rokades, Passant).


%! create_state(+Pieces, +CurrentColor, +Rokades, +Passant, -State)
%
%  Create a state from a list of pieces
create_state(Pieces, CurrentColor, Rokades, Passant, State) :-

    % Create an empty state
    empty_state(CurrentColor, Rokades, Passant, EmptyState),

    % Set the pieces
    set_pieces(EmptyState, Pieces, State).


%! all_possible_states/2(+CurrentState, -NextStates)
%
%  Generate all possible next states for a given state
all_possible_states(CurrentState, NextStates) :-
    state:currentcolor(CurrentState, CurrentColor),

    % All pseudo-possible moves for the next player
    move:all_possible_moves(CurrentColor, CurrentState, NextMoves),

    % Helper predicate
    all_possible_states(CurrentState, NextMoves, NextStates).


%! all_possible_states/3(+CurrentState, +Moves, ?NextStates)
%
%  Generate a new state for every possible move and append it to a list.
all_possible_states(CurrentState, [Move | Moves], NextStates) :-  % Valid pseudo-move
    state:currentcolor(CurrentState, CurrentColor),

    % Do the move and retrieve the new state
    move:do_move(Move, CurrentState, NextState),

    (
        % State is check and should not be included
        state:check(NextState, CurrentColor),
        OtherStates = NextStates
        ;
        % State is not check and should be included
        [NextState | OtherStates] = NextStates
    ),
```

```
    % Recursive call
    all_possible_states(CurrentState, Moves, OtherStates), !.
all_possible_states(_, [], []).                                     % Base case


%! pieces/2(+State, -Pieces)
%
%  List of pieces that are currently on the board
pieces(State, Pieces) :-

    % Find all possible positions on the board
    position:valid_positions(Positions),

    % Helper predicate
    pieces(State, Positions, Pieces).


%! pieces/3(+State, +Positions, -Pieces)
%
%  Helper predicate for pieces/3
pieces(State, [Position | Positions], [Piece | Pieces]) :- % Piece at current position is
↪   not "none"

    % Piece at the current position
    piece_at_position(State, Position, Piece),

    % Piece must not be none
    Piece \== none,

    % Recursive call
    pieces(State, Positions, Pieces), !.
pieces(State, [_ | Positions], Pieces) :-                  % Piece at current position is
↪   "none"

    % Recursive call
    pieces(State, Positions, Pieces), !.
pieces(_, [], []).                                          % Base case


%! color_pieces/3(+State, +Color, -ColorPieces)
%
%  Unify all pieces for a given Color from the given state.
color_pieces(State, Color, ColorPieces) :-
    % Find all possible positions on the board
    position:valid_positions(Positions),

    % Helper predicate
    color_pieces(State, Color, Positions, ColorPieces).


%! color_pieces/4(+State, +Color, +Positions, -ColorPieces)
%
```

```
%  Helper predicate for color_pieces/3
color_pieces(State, Color, [Position | Positions], [ColorPiece | ColorPieces]) :- % Piece at current position is of the given color

    % Piece at the current position
    piece_at_position(State, Position, ColorPiece),

    % Color must match
    piece:color(ColorPiece, Color),

    % Recursive call
    color_pieces(State, Color, Positions, ColorPieces), !.
color_pieces(State, Color, [_ | Positions], ColorPieces) :-                      % Piece at current position is "none" or not of the given color

    % Recursive call
    color_pieces(State, Color, Positions, ColorPieces), !.
color_pieces(_, _, [], []).                                                    % Base case

%! king/3(+State, +Color, -King).
%
%  Get the king piece for a given color
king(State, Color, King) :-
    % Find all possible positions on the board
    position:valid_positions(Positions),

    % Helper predicate
    king(Positions, State, Color, King).

%! king/4(+Positions, +State, +Color, -King).
%
%  Helper predicate for king/3
king([Position | _], State, Color, King) :-
    % King Piece
    King = piece(Color, king, Position),

    % Piece at the current position is the king
    piece_at_position(State, Position, King).
king([_ | Positions], State, Color, King) :-
    % Recursive call
    king(Positions, State, Color, King).


%! piece_at_position(+State, +X/+Y, -Piece)
%
%  Piece at a given position in a given state.
piece_at_position(State, X/Y, Piece) :-
    state:board(State, Board),

    % Requested row
    arg(Y, Board, Row),

    % Requested piece
```

```
    arg(X, Row, Piece).


%! set_piece_at_position(+State, +Piece, +X/+Y, -NewState)
%
%  Set a piece at a given position on the board of the given state.
set_piece_at_position(State, Piece, X/Y, NewState) :-
    state:board(State, Board),

    % Get the row of the given position
    arg(Y, Board, Row),

    % Duplicate board & row to prevent setarg from altering other states.
    duplicate_term(Board, NewBoard),
    duplicate_term(Row, NewRow),

    % Update the position in the row
    setarg(X, NewRow, Piece),

    % Update the board
    setarg(Y, NewBoard, NewRow),

    % Update the state
    set_board(State, NewBoard, NewState), !.


%! set_piece(+State, +Piece, -NewState)
%
%  Set a piece on the board in a given state.
set_piece(State, Piece, NewState) :-
    piece:position(Piece, X/Y),

    % Set the piece at the given position
    set_piece_at_position(State, Piece, X/Y, NewState).


%! set_pieces(+State, +Pieces, +NewState)
%
%  Set a list of pieces on the board in a given state.
set_pieces(State, [], State).
set_pieces(State, [Piece | Pieces], NewState) :-

    % Set the piece
    set_piece(State, Piece, PartialState),

    % Recursive call
    set_pieces(PartialState, Pieces, NewState), !.


%! remove_piece(+State, +Piece, -NewState)
%
%  Remove a piece from the board in a given state.
remove_piece(State, Piece, NewState) :-
```

```
    piece:position(Piece, X/Y),

    % Set the piece position to none
    set_piece_at_position(State, none, X/Y, NewState).


%! remove_pieces(+State, +Pieces, -NewState)
%
%  Remove a list of pieces from the board
remove_pieces(State, [], State).
remove_pieces(State, [Piece | Pieces], NewState) :-

    % Remove the piece
    remove_piece(State, Piece, PartialState),

    % Recursive call
    remove_pieces(PartialState, Pieces, NewState), !.


%! remove_rokades(+State, +Rokade, -NewState)
%
%  Remove a rokade from the state
remove_rokade(State, Rokade, NewState) :-
    state:board(State, Board),
    state:currentcolor(State, CurrentColor),
    state:rokades(State, Rokades),
    state:passant(State, Passant),

    % Remove the rokade
    delete(Rokades, Rokade, NewRokades),

    % Create the new state
    NewState = state(Board, CurrentColor, NewRokades, Passant).


%! remove_rokades(+State, +Rokades, -NewState)
%
%  Remove a list of rokades from the state
remove_rokades(State, [], State).
remove_rokades(State, [Rokade | Rokades], NewState) :-

    % Remove the rokade
    remove_rokade(State, Rokade, PartialState),

    % Recursive call
    remove_rokades(PartialState, Rokades, NewState), !.


%! set_board(+State, +Board, -NewState)
%
%  Set new board value
set_board(State, Board, NewState) :-
    state:currentcolor(State, CurrentColor),
```

```
    state:rokades(State, Rokades),
    state:passant(State, Passant),

    % Create the new state
    NewState = state(Board, CurrentColor, Rokades, Passant).


%! set_passant(+State, +Passant, -NewState)
%
%  Set new passant value
set_passant(State, Passant, NewState) :-
    state:board(State, Board),
    state:currentcolor(State, CurrentColor),
    state:rokades(State, Rokades),

    % Create the new state
    NewState = state(Board, CurrentColor, Rokades, Passant).


%! set_color(+State, +Color, -NewState)
%
%  Set new color value
set_color(State, Color, NewState) :-
    state:board(State, Board),
    state:passant(State, Passant),
    state:rokades(State, Rokades),

    % Create the new state
    NewState = state(Board, Color, Rokades, Passant).


%! check(+State, +Color)
%
% If the king of the given color in-check for the given state.
% The king is now in range of attack by the opponent player.
check(State, Color) :-

    % Retrieve the king from the board
    state:king(State, Color, KingPiece), !,

    % Check if any opponent piece can attack the king of the given color
    state:can_be_attacked(KingPiece, State).


%! can_be_attacked/2(+Piece, +State)
%
%  If a given piece can be attacked in the given state.
can_be_attacked(Piece, State) :-
    piece:color(Piece, Color),

    % Opponent Color
    piece:opponent(Color, OpponentColor),

```

```
    % Opponent Pieces
    state:color_pieces(State, OpponentColor, OpponentPieces),

    % Helper predicate
    can_be_attacked_by_pieces(Piece, State, OpponentPieces).

%! can_be_attacked_by_pieces(+Piece, +State, +OpponentPieces)
%
%  If a given piece can be attacked by a list of opponent pieces.
can_be_attacked_by_pieces(Piece, State, [OpponentPiece | _]) :-             % Can be attacked by moves of current piece

    % Possible moves
    move:possible_moves(OpponentPiece, State, OpponentMoves),

    % Piece can be attacked by received list of moves
    can_be_attacked_by_moves(Piece, OpponentMoves).
can_be_attacked_by_pieces(Piece, State, [_ | OpponentPieces]) :- % Cannot be attacked by moves of current piece
    % Recursive call
    can_be_attacked_by_pieces(Piece, State, OpponentPieces).

%! can_be_attacked_by_moves(+Piece, +Moves)
%
%  If a given piece can be attacked in a given list of moves.
can_be_attacked_by_moves(Piece, [Move | _]) :-  % Can be attacked
    move:delete_pieces(Move, DeletePieces),

    % Piece is present inside the move
    memberchk(Piece, DeletePieces), !.
can_be_attacked_by_moves(Piece, [_ | Moves]) :- % Cannot be attacked

    % Recursive call
    can_be_attacked_by_moves(Piece, Moves), !.


%! checkmate_or_stalemate/1(+State)
%
%  If a given state is checkmate or stalemate for the current player.
%  Will check if the current player cannot do any more moves.
checkmate_or_stalemate(State) :-
    state:currentcolor(State, Color),

    % Find all possible positions on the board
    position:valid_positions(Positions),

    % Helper predicate
    checkmate_or_stalemate(State, Color, Positions).

%! checkmate_or_stalemate/3(+State, +Color +Positions)
%
%  Helper predicate for checkmate_or_stalemate/1
checkmate_or_stalemate(_, _, []).
```

```
checkmate_or_stalemate(State, Color, [Position | Positions]) :-     % Piece at position
↪ is of given color
    Piece = piece(Color, _, Position),

    % Piece at the current position
    piece_at_position(State, Position, Piece),

    % All possible moves for the current piece.
    move:possible_moves(Piece, State, PseudoMoves),

    % All possible valid moves for the current piece should be empty.
    move:valid_moves(State, PseudoMoves, []),

    % Recursive call.
    checkmate_or_stalemate(State, Color, Positions).
checkmate_or_stalemate(State, Color, [Position | Positions]) :-              % Piece at
↪ position is not of given color or none

    % Position is of opponent or empty
    position:empty_or_opponent_position(Position, Color, State),

    % Recursive call.
    checkmate_or_stalemate(State, Color, Positions).
```

Listing 3: src/move.pl

```prolog
:- module(move, []).

:- use_module("state").
:- use_module("position").
:- use_module("piece").


%! delete_pieces(+Move, -DeletePieces)
%
%  List of pieces to delete from the board for a given move
delete_pieces(move(DeletePieces, _), DeletePieces).


%! append_pieces(+Move, -AppendPieces)
%
%  List of pieces to append to the board for a given move
append_pieces(move(_, AppendPieces), AppendPieces).


%! delete_rokades(+Move, -DeleteRokades)
%
%  List of rokades to delete for a given move
delete_rokades(Move, DeleteRokades) :-
    move:delete_pieces(Move, DeletePieces),

    % Convert a list of pieces into a list of rokades
    maplist([Piece, Rokades] >> (piece:rokades_piece(Piece, Rokades)), DeletePieces,
    ↪ PossibleRokades),

    % Merge all possible rokades
    append(PossibleRokades, DeleteRokades).


%! new_passant(+Move, -NewPassant)
%
%  Get the new en-passant possibility for a given move
new_passant(Move, NewPassant) :-
    % When moving a piece 2 steps forward (and creating an en-passant possibility)
    % the piece will never attack an opponent
    move:delete_pieces(Move, [piece(Color, pawn, OldPosition)]),
    move:append_pieces(Move, [piece(Color, pawn, NewPosition)]),

    % Find the en-passant possibility
    position:pawn_start_position(OldPosition, Color),
    piece:passant_piece(piece(Color, pawn, NewPosition), NewPassant), !.
new_passant(_, none).


%! do_move(+Move, +CurrentState, -NewState)
%
```

```
% Update the state with a given move for a given piece.
do_move(Move, CurrentState, NewState) :-
    move:delete_pieces(Move, DeletePieces),
    move:append_pieces(Move, AppendPieces),
    move:delete_rokades(Move, DeleteRokades),
    move:new_passant(Move, NewPassant),

    % Next color
    state:nextcolor(CurrentState, NewColor),

    % Delete pieces
    state:remove_pieces(CurrentState, DeletePieces, PartialState1),

    % Add pieces
    state:set_pieces(PartialState1, AppendPieces, PartialState2),

    % Delete Rokades
    state:remove_rokades(PartialState2, DeleteRokades, PartialState3),

    % Update passant
    state:set_passant(PartialState3, NewPassant, PartialState4),

    % Update color
    state:set_color(PartialState4, NewColor, NewState).


%! all_possible_moves/2(+State, -Moves)
%
%  All pseudo-possible moves for the current state.
all_possible_moves(State, Moves) :-
    state:currentcolor(State, CurrentColor),
    all_possible_moves(CurrentColor, State, Moves).


%! all_possible_moves/3(+Color, +State, -Moves)
%
%  All pseudo-possible moves for the current state for a given color.
all_possible_moves(Color, State, Moves) :-

    % Get the pieces for the given color
    state:color_pieces(State, Color, ColorPieces),

    % Get all possible moves for the pieces
    all_possible_moves_for_pieces(ColorPieces, State, Moves).


%! all_possible_moves_for_pieces(+Pieces, +State, -Moves)
%
%  All pseudo-possible moves for all given pieces in the current state.
all_possible_moves_for_pieces([Piece | Pieces], State, Moves) :-

    % All possible moves for the current piece
    possible_moves(Piece, State, PieceMoves),
```

```

    % Recursive call
    all_possible_moves_for_pieces(Pieces, State, RestMoves),

    % Merge the moves into the moves list
    append(PieceMoves, RestMoves, Moves), !.
all_possible_moves_for_pieces([], _, []).

%! valid_move(+State, +Move)
%
%  If a given move is valid.
%  A move is considered invalid if it causes a check.
valid_move(State, Move) :-
    state:currentcolor(State, CurrentColor),

    % Do the move and retrieve the new state
    move:do_move(Move, State, NextState),

    % State is check
    not(state:check(NextState, CurrentColor)).


%! valid_moves(+State, +Moves, ?ValidMoves)
%
%  Filter a given list of moves by removing all moves that lead to a check of the current color.
valid_moves(_, [], []).
valid_moves(State, [Move | Moves], ValidMoves) :-  % Move is valid

    % Check if move is valid
    % This cut operator is to prevent having to use "not(valid_move(...))" in the
    % next predicate. This is for performance reasons.
    valid_move(State, Move), !,

    % Assign the valid move
    ValidMoves = [Move | ValidMovesRest],

    % Recursive call
    valid_moves(State, Moves, ValidMovesRest).
valid_moves(State, [_ | Moves], ValidMoves) :-              % Move is invalid
    % Recursive call
    valid_moves(State, Moves, ValidMoves).


%! possible_moves(+Piece, +State, -Moves)
%
%  All psuedo-possible moves for a specific piece in the given state.
%  This predicate will also include moves that cause a potential in-check situation.

% King
possible_moves(Piece, State, Moves) :-
    piece:type(Piece, king),

```

```
    % Rokades
    findall(Move, rokades_move(Piece, State, Move), RokadesMoves),

    % King can move in a square
    square_moves(Piece, State, SquareMoves),

    % Merge possible moves
    append([RokadesMoves, SquareMoves], Moves), !.

% Queen
possible_moves(Piece, State, Moves) :-
    piece:type(Piece, queen),

    % Queen can move diagonally or in a cross
    cross_moves(Piece, State, CrossMoves),
    diagonal_moves(Piece, State, DiagonalMoves),

    % Merge possible moves
    append([CrossMoves, DiagonalMoves], Moves), !.

% Tower
possible_moves(Piece, State, Moves) :-
    piece:type(Piece, tower),

    % Tower can move in a cross
    cross_moves(Piece, State, Moves), !.

% Bishop
possible_moves(Piece, State, Moves) :-
    piece:type(Piece, bishop),

    % Bishop can move in diagonally.
    diagonal_moves(Piece, State, Moves), !.

% Horse
possible_moves(Piece, State, Moves) :-
    piece:type(Piece, horse),

   % Horse positions
   findall(Position, position:horse_position(Piece, State, Position), Positions),

   % Convert positions into moves
   positions_to_moves(Piece, State, Positions, Moves), !.

% Pawn
possible_moves(Piece, State, Moves) :-
    piece:type(Piece, pawn),

    % Possible moves
    pawn_forward_moves(Piece, State, ForwardMoves),
    pawn_diagonal_moves(Piece, State, DiagonalMoves),
    pawn_passant_moves(Piece, State, PassantMoves),

```

```
    % Merge possible moves
    append([ForwardMoves, DiagonalMoves, PassantMoves], MergedMoves),

    % Handle potential pawn promotional moves
    convert_promotion_moves(Piece, MergedMoves, Moves), !.


%! convert_promotion_moves(+Piece, +Moves, +PromotionMoves)
%
%  Convert a list of moves to a list of promotion moves.
%  Will scan every move, check if the piece can be promoted, and create the correct promotions
convert_promotion_moves(Piece, [Move | Moves], PromotionMoves) :-    % Current move is promotion move
    piece:color(Piece, Color),
    Move = move(DeletePieces, AppendPieces),

    % Select the pawn
    select(piece(Color, pawn, NewPosition), AppendPieces, _),

    % Check if the pawn position is a promotion position
    position:pawn_promotion_position(NewPosition, Color),

    % Recursive call
    convert_promotion_moves(Piece, Moves, PromotionMovesRest),

    % Create the promotion moves
    PromotionMovesCurrent = [
        move(DeletePieces, [piece(Color, queen, NewPosition)]),
        move(DeletePieces, [piece(Color, horse, NewPosition)]),
        move(DeletePieces, [piece(Color, tower, NewPosition)]),
        move(DeletePieces, [piece(Color, bishop, NewPosition)])
    ],

    % Merge
    append([PromotionMovesCurrent, PromotionMovesRest], PromotionMoves), !.

convert_promotion_moves(Piece, [Move | Moves], [PromotionMove | PromotionMoves]) :-    % Current move is not a promotion move

    % Add the old move to the promotion moves
    % since the original moves, that are no promotions, must be included as well
    PromotionMove = Move,

    % Recursive call
    convert_promotion_moves(Piece, Moves, PromotionMoves), !.

convert_promotion_moves(_, [], []) :- !. % Base Case


%! pawn_moves(+Piece, +State, -Moves)
%
%  Moves for the pawn going forward
```

```
pawn_forward_moves(Piece, State, [Move1, Move2]) :- % Pawn on start position (can move 2 steps forward)
    piece:type(Piece, pawn),
    piece:color(Piece, Color),
    piece:position(Piece, CurrentPosition),

    % Pawn must be on start position
    position:pawn_start_position(CurrentPosition, Color),

    % First position must be valid & empty (otherwise the pawn is not able to move 2 steps forward)
    position:forward_position(CurrentPosition, Color, NewPosition1),
    position:valid_position(NewPosition1),
    position:empty_position(NewPosition1, State),

    % Second position must be valid & empty
    position:forward_position(NewPosition1, Color, NewPosition2),
    position:valid_position(NewPosition2),
    position:empty_position(NewPosition2, State),

    % Create the moves
    create_move(CurrentPosition, NewPosition1, State, Move1),
    create_move(CurrentPosition, NewPosition2, State, Move2), !.

pawn_forward_moves(Piece, State, [Move1]) :-        % Pawn (can move max 1 step forward)
    piece:type(Piece, pawn),
    piece:position(Piece, CurrentPosition),
    piece:color(Piece, Color),

    % Forward position must be valid & empty
    position:forward_position(CurrentPosition, Color, NewPosition1),
    position:valid_position(NewPosition1),
    position:empty_position(NewPosition1, State),

    % Create the moves
    create_move(CurrentPosition, NewPosition1, State, Move1), !.

pawn_forward_moves(Piece, _, []) :-             % Pawn cannot move forward
    piece:type(Piece, pawn),
    !.

%! pawn_diagonal_moves(+Piece, +State, -Moves)
%
%  Moves for the given pawn moving diagonally
pawn_diagonal_moves(Piece, State, Moves) :-

    % Moves for both diagonal parts
    pawn_diagonal_moves_part(Piece, State, -1, LeftMoves),
    pawn_diagonal_moves_part(Piece, State, 1, RightMoves),

    % Merge the 2 lists
    append([LeftMoves, RightMoves], Moves).

```

```prolog

%! pawn_diagonal_moves_part(+Piece, +State, +XDifference, -Moves)
%
%  Moves for the given pawn moving diagonally either left or right.
%  XDifference = 1: right diagonal move
%  XDifference = -1: left diagonal move
pawn_diagonal_moves_part(Piece, State, XDifference, [Move]) :- % Left diagonal
    piece:type(Piece, pawn),
    piece:position(Piece, X/Y),
    piece:color(Piece, Color),

    % New position
    XNew is X + XDifference,
    position:forward_position(XNew/Y, Color, XNew/YNew),

    % New position must be valid
    position:valid_position(XNew/YNew),

    % New position must be taken by an opponent piece
    position:opponent_position(XNew/YNew, Color, State),

    % Create the move
    create_move(X/Y, XNew/YNew, State, Move), !.

pawn_diagonal_moves_part(Piece, _, _, []) :-
    piece:type(Piece, pawn), !.


%! pawn_passant_moves(+Piece, +State, -Moves)
%
%  Move for the given pawn if an en-passant move is possible
pawn_passant_moves(Piece, State, Moves) :-
    piece:type(Piece, pawn),
    piece:color(Piece, PieceColor),
    state:passant(State, Passant),

    Passant = passant(PassantColor, _),

    % En-passant position must be for the opponent
    piece:opponent(PieceColor, PassantColor),

    % En-pasant for both directions
    pawn_passant_moves_part(Piece, State, -1, LeftMoves),
    pawn_passant_moves_part(Piece, State, 1, RightMoves),

    % Merge the 2 lists
    append([LeftMoves, RightMoves], Moves), !.

pawn_passant_moves(Piece, _, []) :-
    piece:type(Piece, pawn), !.


%! pawn_passant_moves_part(+Piece, +Passant, +XDifference, -Moves)
```

```
%
%  Moves for the given pawn doing en-passant either left or right
%  XDifference = 1: right en-passant move
%  XDifference = -1: left en-passant move
pawn_passant_moves_part(Piece, State, XDifference, [Move]) :-
    piece:type(Piece, pawn),
    piece:color(Piece, PieceColor),
    piece:position(Piece, X/Y),
    state:passant(State, Passant),

    Passant = passant(PassantColor, XPassant/YPassant),

    % Check if the passant possibility is next to piece.
    XPassant is X + XDifference,
    position:forward_position(XPassant/Y, PieceColor, XPassant/YPassant),

    % New position of the pawn after en-passant
    position:forward_position(XPassant/Y, PieceColor, XNew/YNew),

    % Piece to remove by doing the en-passant move
    OpponentPiece = piece(PassantColor, pawn, XPassant/Y),

    % Create the move
    Move = move([Piece, OpponentPiece], [piece(PieceColor, pawn, XNew/YNew)]), !.

pawn_passant_moves_part(Piece, _, _, []) :-
    piece:type(Piece, pawn), !.


%! rokades_move(+King, +State, -Moves)
%
%  Rokades move for the king
rokades_move(King, State, Move) :-   % Short rokade
    piece:color(King, Color),
    piece:position(King, KingPosition),
    state:rokades(State, Rokades),
    _/Y = KingPosition,

    % Short rokade
    ShortRokade = rokade(Color, short),
    memberchk(ShortRokade, Rokades),

    % Tower for rokade
    Tower = piece(Color, tower, TowerPosition),
    piece:rokades_piece(Tower, [ShortRokade]),

    % Check if the pieces between the tower and king are empty
    position:empty_between_positions(KingPosition, TowerPosition, State),

    % New pieces
    NewKing  = piece(Color, king,  7/Y),
    NewTower = piece(Color, tower, 6/Y),

```

```
    % Create the move
    Move = move([King, Tower], [NewKing, NewTower]).

rokades_move(King, State, Move) :-   % Long rokade
    piece:color(King, Color),
    piece:position(King, KingPosition),
    state:rokades(State, Rokades),
    _/Y = KingPosition,

    % Long rokade
    LongRokade = rokade(Color, long),
    memberchk(LongRokade, Rokades),

    % Tower for rokade
    Tower = piece(Color, tower, TowerPosition),
    piece:rokades_piece(Tower, [LongRokade]),

    % Check if the pieces between the tower and king are empty
    position:empty_between_positions(TowerPosition, KingPosition, State),

    % New pieces
    NewKing  = piece(Color, king,  3/Y),
    NewTower = piece(Color, tower, 4/Y),

    % Create the move
    Move = move([King, Tower], [NewKing, NewTower]).


%! square_moves(+Piece, +State, -Moves)
%
%  Moves in a square around a given piece
square_moves(Piece, State, Moves) :-

    % Square positions
    findall(Position, position:square_position(Piece, State, Position), Positions),

    % Convert positions into moves
    positions_to_moves(Piece, State, Positions, Moves).


%! cross_moves(+Piece, +State, -Moves)
%
%  Moves in a cross starting from a given piece
cross_moves(Piece, State, Moves) :-

    path_moves(Piece, State, 1, 0, RightMoves),   % Right row part
    path_moves(Piece, State, -1, 0, LeftMoves),   % Left row part
    path_moves(Piece, State, 0, 1, TopMoves),     % Top column part
    path_moves(Piece, State, 0, -1, BottomMoves), % Bottom column part

    % Merge lists
    append([RightMoves, LeftMoves, TopMoves, BottomMoves], Moves).

```

```prolog

%! diagonal_moves(+Piece, +State, -Moves)
%
%  Moves on the diagonals starting from a given piece
diagonal_moves(Piece, State, Moves) :-
    path_moves(Piece, State, 1, 1, TopRightMoves),       % Top-right diagonal
    path_moves(Piece, State, -1, 1, TopLeftMoves),       % Top-right diagonal
    path_moves(Piece, State, 1, -1, BottomRightMoves),   % Bottom-right diagonal
    path_moves(Piece, State, -1, -1, BottomLeftMoves),   % Bottom-left diagonal

    % Merge lists
    append([TopRightMoves, TopLeftMoves, BottomRightMoves, BottomLeftMoves], Moves).

%! path_moves/5(+Piece, +State, +XDirection, +YDirection, -Moves)
%
%  Moves on a given path starting from a piece and with incremental addition of
↪  (XDirection, YDirection)
%  Will stop the path when a new position is either invalid or blocked by another piece
path_moves(Piece, State, XDirection, YDirection, Moves) :-
    piece:position(Piece, X/Y),

    path_moves(Piece, X/Y, State, XDirection, YDirection, Moves).


%! path_moves/6(+StartPiece, +PreviousPosition, +State, +XDirection, +YDirection, -Moves)
%
%  Helper function for path_moves/5.
%  Uses a StartPiece to correctly form the moves.
path_moves(StartPiece, X/Y, State, XDirection, YDirection, [Move | Moves]) :-

    % Unify the new position
    XNew is X + XDirection,
    YNew is Y + YDirection,

    % Create the move
    create_piece_move(StartPiece, XNew/YNew, State, Move),

    % New position must be valid
    position:valid_position(XNew/YNew),

    % New position must be empty
    position:empty_position(XNew/YNew, State), !,

    % Recursivly extend the diagonal
    path_moves(StartPiece, XNew/YNew, State, XDirection, YDirection, Moves).
path_moves(StartPiece, X/Y, State, XDirection, YDirection, [Move]) :-
    piece:color(StartPiece, Color),

    % Unify the new position
    XNew is X + XDirection,
    YNew is Y + YDirection,

    % Create the move
```

```
    create_piece_move(StartPiece, XNew/YNew, State, Move),

    % New position must be valid
    position:valid_position(XNew/YNew),

    % New position must be taken by the opponent
    position:opponent_position(XNew/YNew, Color, State), !.
path_moves(_, _, _, _, _, []).

%! positions_to_moves(+Piece, +Stat, +Positions, -Moves)
%
%  Corresponding moves for a given set of positions
positions_to_moves(Piece, State, [NextPosition | NextPositions], [Move | Moves]) :-

    % Construct the move
    create_piece_move(Piece, NextPosition, State, Move),

    % Recursive Call
    positions_to_moves(Piece, State, NextPositions, Moves), !.
positions_to_moves(_, _, [], []).


%! create_move/4(+CurrentPosition, +NewPosition, +State, -Move)
%
%  Create a move from a given position to a new position.
create_move(CurrentPosition, NewPosition, State, Move) :-
    state:piece_at_position(State, CurrentPosition, CurrentPiece),
    create_piece_move(CurrentPiece, NewPosition, State, Move).


%! create_move/4(+CurrentPiece, +NewPosition, +State, -Move)
%
%  Create a move for a given piece, position and en-passant possability
create_piece_move(CurrentPiece, NewPosition, State, Move) :-  % Opponent on new position
    piece:color(CurrentPiece, Color),
    piece:type(CurrentPiece, Type),

    % Opponent at the new position
    position:opponent_position(NewPosition, Color, State, OpponentPiece),

    % Create the new piece
    NewPiece = piece(Color, Type, NewPosition),

    % Unify the move
    Move = move([CurrentPiece, OpponentPiece], [NewPiece]), !.
create_piece_move(CurrentPiece, NewPosition, State, Move) :-  % No piece on new position
    piece:color(CurrentPiece, Color),
    piece:type(CurrentPiece, Type),

    % Empty new position
    position:empty_position(NewPosition, State),

    % Create the new piece
```

```
    NewPiece = piece(Color, Type, NewPosition),

    % Unify the move
    Move = move([CurrentPiece], [NewPiece]), !.
```

Listing 4: src/piece.pl

```
:- module(piece, []).

:- use_module("position").
:- use_module("move").


%! position(+Piece, -Position)
%
%  Extract the position from the given piece.
position(piece(_, _, Position), Position).


%! type(+Piece, -Type)
%
%  Extract the type from the given piece.
type(piece(_, Type, _), Type).


%! type(+Color, -Type)
%
%  Extract the color from the given piece.
color(piece(Color, _, _), Color).


%! rokades_piece(+Piece, -Rokades)
%
%  List of rokades for a given piece
rokades_piece(piece(white, tower, 1/1), [rokade(white, long)]).                           % Tower
rokades_piece(piece(white, tower, 8/1), [rokade(white, short)]).                          % Tower
rokades_piece(piece(white, king, 5/1), [rokade(white, long), rokade(white, short)]). % King

rokades_piece(piece(black, tower, 1/8), [rokade(black, long)]).                           % Tower
rokades_piece(piece(black, tower, 8/8), [rokade(black, short)]).                          % Tower
rokades_piece(piece(black, king, 5/8), [rokade(black, long), rokade(black, short)]). % King

rokades_piece(_, []). % Base case


%! passant_piece(+Piece, -Passant)
%
%  En-passant possibility for a given piece
passant_piece(piece(white, pawn, X/4), passant(white, X/3)).
passant_piece(piece(black, pawn, X/5), passant(black, X/6)).
passant_piece(_, none).
```

```
%! row_pieces(+Y, +Pieces, -RowPieces)
%
%  List of pieces for a given row.
row_pieces(Y, [Piece | Pieces], [RowPiece | RowPieces]) :- % Match
    position(Piece, _/PieceY),

    % Row numbers must match
    PieceY == Y,

    % Append to the list
    RowPiece = Piece, !,

    % Recursive call
    row_pieces(Y, Pieces, RowPieces), !.
row_pieces(Y, [Piece | Pieces], RowPieces) :- % No match
    position(Piece, _/PieceY),

    % Row numbers must not match
    PieceY \== Y,

    % Recursive call
    row_pieces(Y, Pieces, RowPieces), !.
row_pieces(_, [], []).


%! sorted_pieces/2(+Pieces, -SortedPieces)
%
%  Sorted list for a given list of pieces by X-coordinate.
sorted_pieces(Pieces, SortedPieces) :-
    sorted_pieces(Pieces, 1, SortedPieces).

%!  sorted_pieces/3(+Pieces, +X, -SortedPieces)
%
%   Helper predicate for sorted_pieces/2.
sorted_pieces(Pieces, X, [SortedPiece | SortedPieces]) :-

    % X must be valid
    between(1, 8, X),

    % Select the piece with current X coordinate, if any
    select(piece(Color, Type, X/Y), Pieces, _),

    % Add the piece
    SortedPiece = piece(Color, Type, X/Y),

    % Recursive call
    XNext is X + 1,
    sorted_pieces(Pieces, XNext, SortedPieces), !.
sorted_pieces(Pieces, X, SortedPieces) :-

    % X must be valid
```

```
    between(1, 8, X),

    % Recursive call
    XNext is X + 1,
    sorted_pieces(Pieces, XNext, SortedPieces), !.
sorted_pieces(_, _, []).


%! opponent(+Color, -OpponentColor)
%
%  Opponent color for a given color
opponent(white, black).
opponent(black, white).
```

Listing 5: src/position.pl

```
:- module(position, []).

:- use_module("state").
:- use_module("piece").
:- use_module("util/utils").


%! pawn_start_position(+X/+Y, +Color)
%
%  Pawn is on it's starting position.
pawn_start_position(_/2, white).
pawn_start_position(_/7, black).


%! pawn_promotion_position(+X/+Y, +Color)
%
%  Pawn is on it's promotion position.
pawn_promotion_position(_/8, white).
pawn_promotion_position(_/1, black).


%! forward_position(+X/+Y, +Color, +X/-Y)
%
%  Forward for a given piece
%  For white piece: +1
%  For black piece: -1
forward_position(X/Y, white, X/YNew) :- YNew is Y + 1.
forward_position(X/Y, black, X/YNew) :- YNew is Y - 1.


%! horse_position(+Piece, +State, -Position)
%
%  Move that could be done by the horse from a given piece
horse_position(piece(Color, _, X/Y), State, XPos/YPos) :-

    % Positions in a square the current position
    % (X/Y) will also be unified
    utils:between2(X, XPos),
    utils:between2(Y, YPos),

    % New position must be valid
    valid_position(XPos/YPos),

    % Position must be empty or taken by an opponent piece
    empty_or_opponent_position(XPos/YPos, Color, State),

    % Difference in positions
    XDiff is X - XPos,
    YDiff is Y - YPos,

```

```
    % Possible differences for the move
    PossibleDifferences = [
        (-1, 2),
        (-2, 1),
        (1, 2),
        (2, 1),
        (-2, -1),
        (-1, -2),
        (2, -1),
        (1, -2)
    ],

    % Difference must be a member of the possible differences
    memberchk((XDiff, YDiff), PossibleDifferences).


%! square_position(+Piece, +State, -XPos/-YPos)
%
%  Position in a square around a given piece
square_position(Piece, State, XPos/YPos) :-
    piece:color(Piece, Color),
    piece:position(Piece, X/Y),

    % Positions in a square the current position
    % (X/Y) will also be unified
    utils:between1(X, XPos),
    utils:between1(Y, YPos),

    % Position must not be (X/Y)
    XPos/YPos \== X/Y,

    % New position must be valid
    valid_position(XPos/YPos),

    % New position must be empty or taken by an opponent piece
    empty_or_opponent_position(XPos/YPos, Color, State).


%! valid_position(+X/+Y)
%
%  Check if a give coordinate is a valid position on the board for a piece to move to.
%  Will check if the position is on the board (not outside).
%
%  WARNING: This predicate will not check if the position is allowed for the particular piece type!
:- table valid_position/1. % Memoization
valid_position(X/Y) :-

    % X must be inside the board
    between(1, 8, X),

    % Y must be inside the board
    between(1, 8, Y).
```

```
%! valid_positions(-Positions)
%
%  List of all possible positions on the board
%
%  This could also be done using "findall", but hard-coding this makes it significantly faster when alpha-beta pruning.
:- table valid_positions/1. % Memoization
valid_positions(Positions) :-
    findall(X/Y, valid_position(X/Y), Positions).


%! empty_position(+X/+Y, +State)
%
%  Check if a given position is not taken by a piece.
empty_position(X/Y, State) :-

    % Piece at the given position must be none
    state:piece_at_position(State, X/Y, none).


%! opponent_position/3(+X/+Y, +Color, +State)
%
%  Check if a given position is taken by a piece of the opponent player.
opponent_position(X/Y, Color, State) :-

    % Opponent color
    piece:opponent(Color, OpponentColor),

    % Piece at the given position must be of the opponents color
    state:piece_at_position(State, X/Y, piece(OpponentColor, _, _)).


%! opponent_position/4(+X/+Y, +Color, +State, -OpponentPiece)
%
%  Check if a given position is taken by a piece of the opponent player.
%  Unify the piece with OpponentPiece.
opponent_position(X/Y, Color, State, OpponentPiece) :-

    % Opponent color
    piece:opponent(Color, OpponentColor),

    % Piece at the given position must be as described above
    state:piece_at_position(State, X/Y, piece(PieceColor, PieceType, _)),

    % Piece color must match opponent color
    PieceColor == OpponentColor,

    % Opponent Piece
    OpponentPiece = piece(OpponentColor, PieceType, X/Y).
```

```
%! empty_or_opponent_position(+X/+Y, +Color, +State)
%
%  Check if a position is empty or taken by a piece of the opponent player.
empty_or_opponent_position(X/Y, _, State) :- empty_position(X/Y, State).
empty_or_opponent_position(X/Y, Color, State) :- opponent_position(X/Y, Color, State).


%! empty_between_positions(+X1/+Y, +X2/+Y, +State)
%
%  If the positions between 2 coordinates (in a row line) are empty
empty_between_positions(X/Y, X/Y, _).                % Base Case
empty_between_positions(X1/Y, X2/Y, _) :-           % No positions between the given positions
    XPlus is X1 + 1,

    % There are no positions between the X1 & X2
    XPlus == X2.
empty_between_positions(X1/Y, X2/Y, State) :-      % Recursion Case
    % X1 must be smaller than X2
    X1 < X2,

    % Position must be between X1 and X2
    XPos is X1 + 1,

    % New position must be empty
    empty_position(XPos/Y, State),

    % Recursive call
    empty_between_positions(XPos/Y, X2/Y, State).
```

Listing 6: src/io/parser.pl

```
:- module(parser, []).

:- use_module(library(pio)).
:- use_module(library(dcg/basics)).
:- use_module("../state").

% Interpret quoted strings as ASCII character codes.
:- set_prolog_flag(double_quotes, codes).


%! parse_state(-State, StartColor)
%
%  Parse a chess game state.
parse_state(State) -->
    parse_rows(8, PiecesList, RokadesList, Passant, StartColor),
    parse_final_row,
    {
        % Create a flat list of pieces
        append(PiecesList, Pieces),

        % Create a flat list of rokades
        append(RokadesList, Rokades),

        % Create state
        state:create_state(Pieces, StartColor, Rokades, Passant, State),

        % Set passant to "none" if no en-passant move was unified
        (Passant = none, ! ; true)
    }.


%! parse_rows(+Y, -Pieces, -Rokades, -StartColor)
%
%  Parse all rows in a flat list of board positions.
parse_rows(8, [Pieces | PiecesRest], [Rokades | RokadesRest], Passant, StartColor) --> %
↪ Last row
    % Parse the row
    parse_border_row(8, black, Pieces, Rokades, Passant, StartColor),

    % Recursive call
    parse_rows(7, PiecesRest, RokadesRest, Passant, StartColor).

parse_rows(1, [Pieces], [Rokades], Passant, StartColor) --> % First row
    % Parse the row
    parse_border_row(1, white, Pieces, Rokades, Passant, StartColor).

parse_rows(Y, [Pieces | PiecesRest], Rokades, Passant, StartColor) --> % Rows in between
↪ first & last row
    {
        YNext is Y - 1
```

```
    },

    % Parse the row
    parse_row(Y, Pieces),

    % Recursive call
    parse_rows(YNext, PiecesRest, Rokades, Passant, StartColor).


%! parse_row(-Y, -Pieces)
%
%  Parse a row of the chess board.
%  Will not parse the first/last row
parse_row(Y, Pieces) -->
    parse_row_number(Y),
    parse_space,
    parse_pieces(1/Y, Pieces),
    parse_newline.


%! parse_border_row(+Y, +Color, -Pieces, -Rokades, -StartColor)
%
%  Parse the first/last row of the chess board.
parse_border_row(Y, Color, Pieces, Rokades, Passant, StartColor) -->
    parse_row_number(Y),
    parse_space,
    parse_pieces(1/Y, Pieces),
    parse_space,
    parse_metadata(Color, Rokades, Passant),
    parse_current_player(Y, StartColor),
    parse_newline.


%! parse_metadata(+Color, -Rokades, -Passant)
%
%  Parse metadata (possible rokades & passant posibility)
parse_metadata(Color, Rokades, Passant) -->
    "[",
    parse_rokades(Color, Rokades),
    parse_passant(Color, Passant),
    "]".


%! parse_rokades(+Color, -Rokades)
%
%  Parse a rokade notation.
%
%  This is verbose on purpose to allow re-use of the parser for generating output.
parse_rokades(Color, [LongRokade, ShortRokade]) --> % Both rokades
    parse_rokade_piece(Color, long, LongRokade),
    parse_rokade_piece(Color, short, ShortRokade).

parse_rokades(Color, [LongRokade]) --> % Only first rokade
```

```
    parse_rokade_piece(Color, long, LongRokade),
    parse_space.

parse_rokades(Color, [ShortRokade]) --> % Only second rokade
    parse_space,
    parse_rokade_piece(Color, short, ShortRokade).

parse_rokades(_, []) --> % No rokades
    parse_space,
    parse_space.

%! parse_rokade_piece(+Color, +RokadeType, -Rokades)
%
%  Parse a single piece of the rokade notation
parse_rokade_piece(Color, long, Rokade) --> % Large
    parse_piece(Color, queen),

    % Create the rokade
    {
        Rokade = rokade(Color, long)
    }.

parse_rokade_piece(Color, short, Rokade) --> % Short
    parse_piece(Color, king),

    % Create the rokade
    {
        Rokade = rokade(Color, short)
    }.


%! parse_passant(+Color, -Passant)
%
%  Parse passant possibility
parse_passant(Color, passant(Color, X/Y)) --> % En-passant possible
    parse_column_number(X),
    parse_row_number(Y).
parse_passant(_, _) --> [].                    % En-passant not possible


%! parse_passant_position(-X/-Y)
%
%  Parse passant possibility position
parse_passant_position(X/Y) -->          % En-passant possible
    parse_column_number(X),
    parse_row_number(Y).

parse_passant_position(_) --> [].        % En-passant not possible


%! parse_current_player(+Y, -StartColor)
%
%  Parse a current player symbol or nothing
```

```
parse_current_player(8, black) --> "".
parse_current_player(1, white) --> "".
parse_current_player(_, _)     --> "".


%! parse_final_row()
%
%  Parse the final row of the board.
%  This row does not contain any extra information.
parse_final_row -->
    parse_space,
    parse_space,
    "abcdefgh",
    parse_newline_or_nothing.


%! parse_row_number(+RowNumber)
%
%  Parse a row number between 1 and 8.
parse_row_number(RowNumber) -->
    {
        % RowNumber must be a valid Y-value
        % (this is here to allow for re-using the parser as output writer)
        between(1, 8, RowNumber)
    },
    integer(RowNumber).


%! parse_column_number(+ColumnNumber)
%
%  Parse a column number between 1 and 8.
parse_column_number(1) --> "a".
parse_column_number(2) --> "b".
parse_column_number(3) --> "c".
parse_column_number(4) --> "d".
parse_column_number(5) --> "e".
parse_column_number(6) --> "f".
parse_column_number(7) --> "g".
parse_column_number(8) --> "h".


%! parse_space()
%
%  Parse a single space.
parse_space --> " ".


%! parse_newline()
%
% Parse a single newline.
parse_newline --> "\n".
```

```
%! parse_newline_or_nothing()
%
% Parse a single newline or nothing
parse_newline_or_nothing --> parse_newline.
parse_newline_or_nothing --> "".


%! parse_pieces(-Pieces)
%
%  Parse a row of pieces.
%  Will stop parsing when a newline is detected

% Piece: taken position
parse_pieces(X/Y, [Piece | Pieces]) -->
    {
        XNext is X + 1,

        % Create the piece
        Piece = piece(Color, Type, X/Y),

        % X must be a valid X-value
        % (this is here to allow for re-using the parser as output writer)
        between(1, 8, X)
    },

    parse_piece(Color, Type),
    parse_pieces(XNext/Y, Pieces).

% Space: empty position
parse_pieces(X/Y, Pieces) -->
    {
        XNext is X + 1,

        % X must be a X-value
        % (this is here to allow for re-using the parser as output writer)
        between(1, 8, X)
    },

    parse_space,
    parse_pieces(XNext/Y, Pieces).

parse_pieces(_, []) --> [].


%! parse_piece(-Color, -Type)
%
%  Parse a single piece.
parse_piece(white, king)   --> "\u2654". % White king
parse_piece(white, queen)  --> "\u2655". % White queen
parse_piece(white, tower)  --> "\u2656". % White tower
parse_piece(white, bishop) --> "\u2657". % White tower
parse_piece(white, horse)  --> "\u2658". % White horse
parse_piece(white, pawn)   --> "\u2659". % White pawn
```

```
parse_piece(black, king)   --> "\u265A". % Black king
parse_piece(black, queen)  --> "\u265B". % Black queen
parse_piece(black, tower)  --> "\u265C". % Black tower
parse_piece(black, bishop) --> "\u265D". % Black tower
parse_piece(black, horse)  --> "\u265E". % Black horse
parse_piece(black, pawn)   --> "\u265F". % Black pawn
```

Listing 7: src/io/writer.pl

```
:- module(writer, []).

:- use_module("parser").
:- use_module("../state").
:- use_module("../position").
:- use_module("../move").
:- use_module("../piece").


%! write_state(+State)
%
%  Write a chess game state to stdout.
write_state(State) :-
    state:pieces(State, Pieces),
    state:currentcolor(State, StartColor),
    state:rokades(State, Rokades),
    state:passant(State, Passant),

    % Convert board & rokades in a format that could be used by the parser
    extract_rows(8, Pieces, PiecesList),
    extract_rokades(Rokades, RokadesList),

    % Parse the rows output
    parser:parse_rows(8, PiecesList, RokadesList, Passant, StartColor, OutRows, []),

    % Parse the final row output
    parser:parse_final_row(OutFinalRow, []),

    % Write the output
    write_codes(OutRows),
    write_codes(OutFinalRow).


%! write_states(+States)
%
%  Write a given list of states to stdout.
write_states([]).
write_states([State]) :-

    % Write the board to stdout
    write_state(State), !.
write_states([State | States]) :-

    % Write the board to stdout
    write_state(State),
    write("\n~\n"),

    % Recursive call
    write_states(States), !.

```

```
%! write_draw()
%
%  Write "DRAW" to stdout.
write_draw() :- write("DRAW").


%! write_states_or_draw(+CurrentState, -NextStates)
%
%  Write all the states to stdout, or write "DRAW" in case of a stalemate

% Stalemate
write_states_or_draw(CurrentState, []) :-
    state:currentcolor(CurrentState, CurrentColor),

    % Check if the king is not check
    not(state:check(CurrentState, CurrentColor)),

    % Write "DRAW"
    write_draw, !.

% Write all states
write_states_or_draw(_, NextStates) :-
    write_states(NextStates), !.


%! write_codes(+Codes)
%
%  Write a given list of codes to stdout.
write_codes(Codes) :-
    atom_codes(String, Codes),
    write(String).


%! extract_rows(+Y, +Pieces, -Rows)
%
%  List of pieces for a given board, with each list representing the pieces for that row.
extract_rows(Y, Pieces, [Row | Rows]) :-

    % Y must be valid
    between(1, 8, Y), !,

    % Extract the row
    piece:row_pieces(Y, Pieces, UnsortedRow),

    % Sort the row
    piece:sorted_pieces(UnsortedRow, Row),

    % Next row
    YNext is Y - 1,

    % Recursive call
    extract_rows(YNext, Pieces, Rows), !.
```

```
extract_rows(_, _, []).


%! extract_rokades(+Rokades, +RokadesList)
extract_rokades(Rokades, [BlackRokades, WhiteRokades]) :-
    extract_rokades_color(black, Rokades, BlackRokades),
    extract_rokades_color(white, Rokades, WhiteRokades).


%! extract_rokades_color(+Color, +Rokades, -ColorRokades)
%
%  Extract all rokades for a given color
extract_rokades_color(Color, [Rokade | Rokades], [ColorRokade | ColorRokades]) :- % Match
    Rokade = rokade(RokadeColor, _),

    % Color must match
    Color == RokadeColor,

    % Append to the list
    ColorRokade = Rokade, !,

    % Recursive call
    extract_rokades_color(Color, Rokades, ColorRokades), !.
extract_rokades_color(Color, [_ | Rokades], ColorRokades) :-                 % No match
    % Recursive call
    extract_rokades_color(Color, Rokades, ColorRokades), !.
extract_rokades_color(_, [], []).
```

Listing 8: src/main.pl

```
:- initialization(main, main).

:- use_module("io/parser").
:- use_module("io/writer").
:- use_module("move").
:- use_module("alphabeta").
:- use_module("state").


main :-
    current_prolog_flag(argv, Args),
    handle_main(Args),
    halt(0).

handle_main([_]) :- % Test Mode

    % Load the data from the stdin stream and parse it.
    phrase_from_stream(parser:parse_state(State), current_input),

    % Get all possible states for the current state
    state:all_possible_states(State, NextStates),

    % Print all possible states
    writer:write_states_or_draw(State, NextStates).

handle_main([]) :- % Move Mode

    % Load the data from the stdin stream and parse it.
    phrase_from_stream(parser:parse_state(State), current_input),

    % Extract the player from the state
    state:currentcolor(State, Player),

    % Determin the next best move
    alphabeta:alphabeta(Player, State, 0, 4, -100000, 100000, BestState, _),

    % Write the next state to stdout, or write "DRAW" in case of a stalemate.
    % There is a stalemate when the best state is equal to none
    (
        % Print "DRAW"
        BestState == none, writer:write_draw()
        ;
        % Print the best state
        writer:write_state(BestState)
    ).
```

Listing 9: src/util/utils.pl

```
:- module(utils, []).
```

```


%! between1(+Start, ?Value)
%
%  Value is between Start - 1 and Start + 1
:- table between1/2.
between1(Start, Value) :-
    Minus is Start - 1,
    Plus  is Start + 1,
    between(Minus, Plus, Value).


%! between2(+Start, ?Value)
%
%  Value is between Start - 2 and Start + 2
:- table between2/2.
between2(Start, Value) :-
    Minus is Start - 2,
    Plus  is Start + 2,
    between(Minus, Plus, Value).

%! list_equals(+List1, +List2)
%
%  If 2 lists match, ignoring the order
list_equals([], []).
list_equals([H1 | T1], List2) :-
    member(H1, List2),
    delete(List2, H1, List3),
    list_equals(T1, List3).
```