

BACHELOR OF SCIENCE IN INFORMATICA

2020-2021

LOGISCH PROGRAMMEREN - PROJECT SCHAAKCOMPUTER

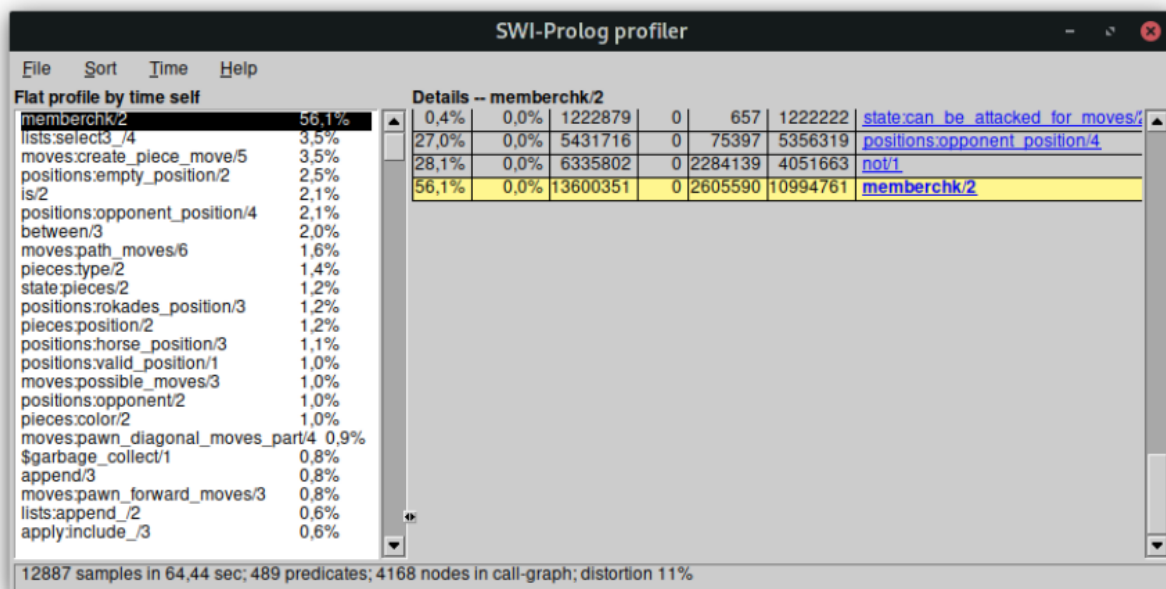
Maarten Van Neyghem

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 sub-structures 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 Scoringsfunctie

🔗 **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 dicht 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:

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

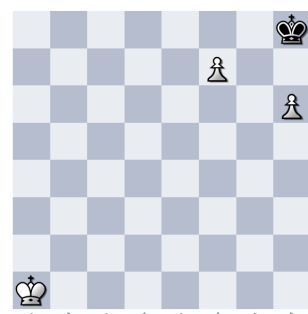
Indien de maximaliserende of minimaliserende speler de hoogste score heeft zal de score van de state respectievelijk positief of negatief zijn. ([src/alphabeta.pl](#) op lijn 211)

2.4 Voorbeeld

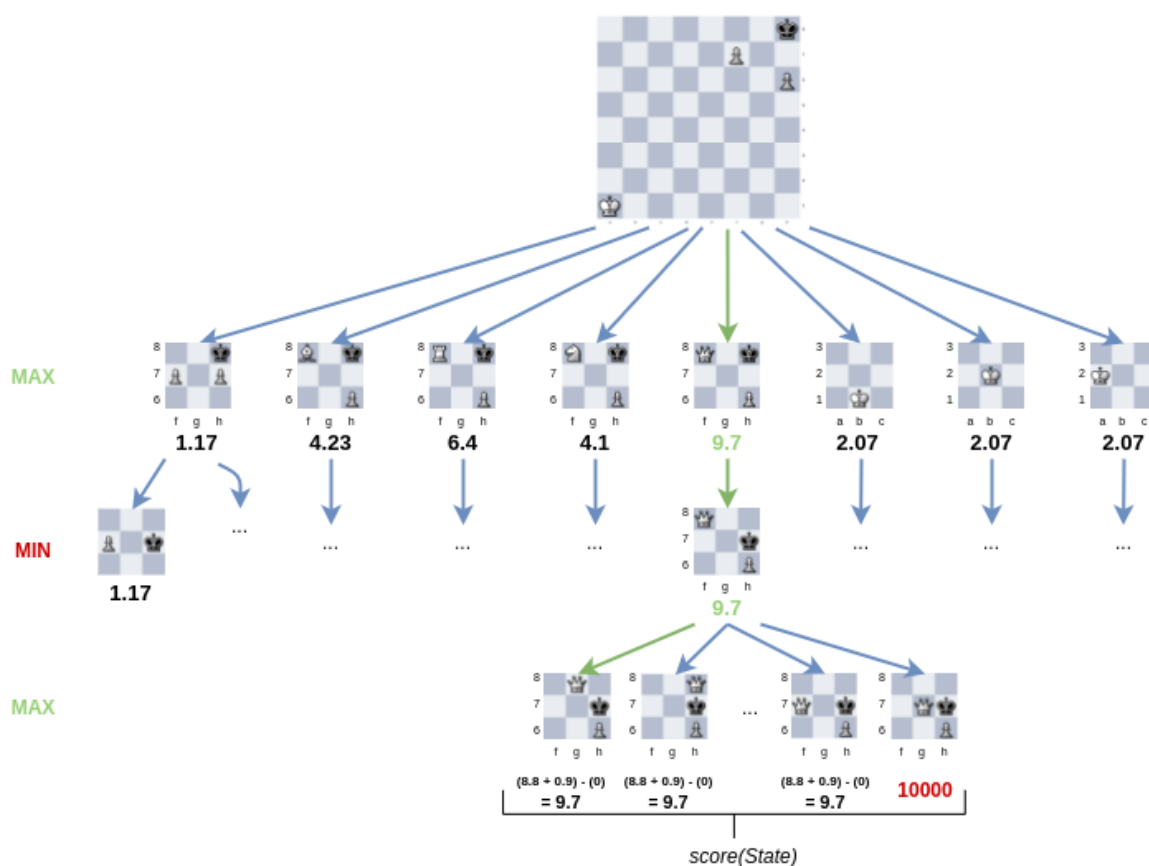
Met bord voorstelling in figuur 2 is het mogelijk om de zwarte koning in 2 zetten schaakmat te plaatsen. Dit kan bereikt worden door de witte pion op F7 naar F8 te verplaatsen en deze vervolgens te promoveren tot een koningin. De tegenstander heeft dan nog slechts 1 mogelijkheid om niet schaak te staan,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

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



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 slimme 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.

```

8 ♔♚♙♜♞♝♞♙ [ ♜♞ ]
7 ♟♟♟♟♟♟♟♟
6
5
4
3
2 ♜♜♜♜♜♜♜♜
1 ♟♟♟♟♟♟♟♟ [ ♜♞♚3 ]
  abcdefgh

```

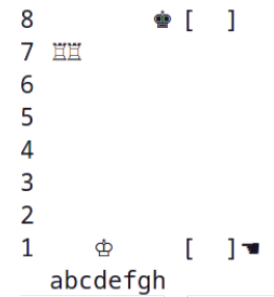
Figuur 4: Bord layout

	Diepte 1	Diepte 2	Diepte 3	Diepte 4
Geplaatste zet	8 ♔♚♙♜♞♝♞♙ [♜♞♚] 7 ♟♟♟♟♟♟♟♟ 6 5 4 ♜ 3 2 ♜♜ ♜♜♜♜♜♜ 1 ♟♟♟♟♟♟♟♟ [♜♞♚3] abcdefgh	8 ♔♚♙♜♞♝♞♙ [♜♞♚] 7 ♟♟♟♟♟♟♟♟ 6 5 4 ♜ 3 2 ♜♜ ♜♜♜♜♜♜ 1 ♟♟♟♟♟♟♟♟ [♜♞♚3] abcdefgh	8 ♔♚♙♜♞♝♞♙ [♜♞♚] 7 ♟♟♟♟♟♟♟♟ 6 5 4 ♜ 3 2 ♜♜ ♜♜♜♜♜♜ 1 ♟♟♟♟♟♟♟♟ [♜♞♚3] abcdefgh	8 ♔♚♙♜♞♝♞♙ [♜♞♚] 7 ♟♟♟♟♟♟♟♟ 6 5 4 ♜ 3 2 ♜♜ ♜♜♜♜♜♜ 1 ♟♟♟♟♟♟♟♟ [♜♞♚3] abcdefgh
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 geïnitieerd moet zijn.
 - <variabele>: een variabele die nog niet geïnitieerd 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/PPPPPPPP/RNBQKBNR%5C%20w%5C%20KQkq%5C%20-%5C%200%5C%201> (bezocht op 11-06-2021).

Code

Listing 1: src/alphabeta.pl

```
1 :- module(alphabeta, []).
2
3 :- use_module("move").
4 :- use_module("state").
5 :- use_module("position").
6 :- use_module("piece").
7
8
9 %! alphabeta(+Player, +CurrentState, +TraversedDepth, +MaxDepth, +LowerBound,
   ↪ +UpperBound, -BestState, -BestScore)
10 %
11 % Alpha/beta pruning to determin the next best move.
12 % Depth will specify the max recursion depth.
13 alphabeta(Player, CurrentState, MaxDepth, MaxDepth, _, _, CurrentState, BestScore) :-
   ↪ % Leaf: maximum depth is reached
14
15     % Calculate the score for the current state
16     score(Player, CurrentState, MaxDepth, MaxDepth, BestScore), !.
17
18 alphabeta(Player, CurrentState, TraversedDepth, MaxDepth, LowerBound, UpperBound,
   ↪ BestState, BestScore) :- % Continue building the game tree
19
20     % Determin all possible next states for the current state
21     state:all_possible_states(CurrentState, NextStates),
22
23     % Next States must not be empty (otherwise there is a checkmate or a stalemate)
24     NextStates \= [],
25
26     % Decrement the depth
27     NextTraversedDepth is TraversedDepth + 1,
28
29     % Find the best possible move
30     best(Player, NextStates, NextTraversedDepth, MaxDepth, LowerBound, UpperBound,
   ↪ BestState, BestScore), !.
31
32 % This branch will only be reached if the above 2 variants of the predicate fail.
33 % This is only the case when all_possible_states is empty,
34 % which is checkmate for the current player.
35 alphabeta(Player, CurrentState, TraversedDepth, MaxDepth, _, _, CurrentState, BestScore)
   ↪ :- % Leaf: a player is checkmate
36     state:currentcolor(CurrentState, CheckmatePlayer),
37
38     % Make sure the king is check (otherwise a stalemate is reached)
39     state:check(CurrentState, CheckmatePlayer),
40
41     % Calculate the score fot the current state.
42     score_checkmate(Player, CheckmatePlayer, TraversedDepth, MaxDepth, BestScore), !.
43
```

```

44 alphabeta(Player, CurrentState, _, _, _, _, none, BestScore) :-
    ↪ % Leaf: a player is stalemate
45     state:currentcolor(CurrentState, StalematePlayer),
46
47     % Calculate the score fot the current state.
48     score_stalemate(Player, StalematePlayer, BestScore), !.
49
50
51 %! best(+Player, +States, +TraversedDepth, +MaxDepth, +LowerBound, +UpperBound,
    ↪ -BestState, -BestScore)
52 %
53 % Best state in a given list of states.
54 best(_, [], _, _, _, _, _).
    ↪ % Base case
55 best(Player, [State], TraversedDepth, MaxDepth, LowerBound, UpperBound, State, Score) :-
    ↪ % Single state, return state
56
57     % Do minimax for the current state
58     alphabeta(Player, State, TraversedDepth, MaxDepth, LowerBound, UpperBound, _, Score)
    ↪ , !.
59 best(Player, [State | OtherStates], TraversedDepth, MaxDepth, LowerBound, UpperBound,
    ↪ BestState, BestScore) :- % Multiple states, determin best state
60
61     % Do minimax for the current state
62     alphabeta(Player, State, TraversedDepth, MaxDepth, LowerBound, UpperBound, _, Score),
63
64     % Cut or continue evaluation
65     cut(Player, State, Score, OtherStates, TraversedDepth, MaxDepth, LowerBound,
    ↪ UpperBound, BestState, BestScore).
66
67
68 %! cut(+Player, +State, +Score, +OtherStates, +TraversedDepth, +MaxDepth, +LowerBound,
    ↪ +UpperBound, -BestState, -BestScore)
69 %
70 % Cut of branches that will never lead to a result (Alpha/Beta-pruning)
71 cut(Player, State, Score, _, _, _, _, UpperBound, State, Score) :-
    ↪ % Maximizing player, cut-off
72     max(State, Player),
73
74     % Cut-off the branch if the score is larger than the upperbound
75     Score >= UpperBound, !.
76 cut(Player, State, Score, _, _, _, LowerBound, _, State, Score) :-
    ↪ % Minimizing player, cut-off
77     min(State, Player),
78
79     % Cut-off the branch if the score is larger than the upperbound
80     Score =< LowerBound, !.
81 cut(Player, State1, Score1, OtherStates, TraversedDepth, MaxDepth, LowerBound,
    ↪ UpperBound, BestState, BestScore) :- % Continue evaluation
82
83     % Update upper/lower bound
84     update_bounds(Player, State1, Score1, LowerBound, UpperBound, NewLowerBound,
    ↪ NewUpperBound),

```

```

85
86     % Continue evaluation of the other states
87     best(Player, OtherStates, TraversedDepth, MaxDepth, NewLowerBound, NewUpperBound,
88         ↪ State2, Score2),
89
90     % Determin the best state of the 2 states
91     best_of(Player, State1, State2, Score1, Score2, BestState, BestScore), !.
92
93 %! update_bounds(+Player, +State, +Score, +LowerBound, +UpperBound, -NewLowerBound,
94 ↪ -NewUpperBound)
95 %
96 % Update the upper & lowerbound, if necessary
97
98 % Update the lowerbound to the current score if:
99 % * Current player is maximizing player
100 % * Score is larger than the lowerbound
101 update_bounds(Player, State, Score, LowerBound, UpperBound, Score, UpperBound) :-
102     max(State, Player),
103
104     % Score must be larger than the lowerbound.
105     Score > LowerBound, !.
106
107 % Update the upperbound to the current score if:
108 % * Current player is minimizing player
109 % * Score is smaller than the upperbound
110 update_bounds(Player, State, Score, LowerBound, UpperBound, LowerBound, Score) :-
111     min(State, Player),
112
113     % Score must be larger than the lowerbound.
114     Score < UpperBound, !.
115
116 % Base case
117 update_bounds(_, _, _, LowerBound, UpperBound, LowerBound, UpperBound).
118
119 %! best_of(+Player, +State1, +State2, +Score1, +Score2, -BestState, -BestScore)
120 %
121 % Best state between 2 states, based on their scores.
122 best_of(Player, State1, _, Score1, Score2, BestState, BestScore) :- % Maximizing player
123     ↪ (Score 1 is largest)
124     max(State1, Player),
125
126     % Score 1 is largest
127     Score1 >= Score2,
128
129     % Update best state
130     BestState = State1,
131     BestScore = Score1.
132 best_of(Player, _, State2, Score1, Score2, BestState, BestScore) :- % Maximizing player
133     ↪ (Score 2 is largest)
134     max(State2, Player),

```

```

134     % Score 2 is largest
135     Score1 < Score2,
136
137     % Update best state
138     BestState = State2,
139     BestScore = Score2.
140 best_of(Player, State1, _, Score1, Score2, BestState, BestScore) :- % Minimizing player
141     ⇨ (Score 1 is smallest)
142     min(State1, Player),
143
144     % Score 1 is smallest
145     Score1 =< Score2,
146
147     % Update best state
148     BestState = State1,
149     BestScore = Score1.
150 best_of(Player, _, State2, Score1, Score2, BestState, BestScore) :- % Minimizing player
151     ⇨ (Score 2 is smallest)
152     min(State2, Player),
153
154     % Score 2 is smallest
155     Score1 > Score2,
156
157     % Update best state
158     BestState = State2,
159     BestScore = Score2.
160
161 %! max(+State, +Player)
162 %
163 % If the state is for the maximizing player
164 % Since the state contains the player that can do the next move, the currentcolor must
165 ⇨ be different from the player.
166 max(State, Player) :-
167     state:currentcolor(State, CurrentPlayer),
168     CurrentPlayer \== Player.
169
170 %! min(+State, +Player)
171 %
172 % If the state is for the minimizing player
173 % Since the state contains the player that can do the next move, the currentcolor must
174 ⇨ be the same as the player.
175 min(State, Player) :-
176     state:currentcolor(State, CurrentPlayer),
177     CurrentPlayer == Player.
178
179 %! score(+Player, +State, +TraversedDepth, +MaxDepth, -Score)
180 %
181 % Score for a given state.
182 %
183 % This scoring predicate is a symmetric evaluation function that will score the current
184 ⇨ state of the board.

```

```

183 % It does not keep track of previous states or boards and just evaluates the current
    ↳ state, as is.
184 %
185 % - Each piece will receive a value based on it's importance in the game
186 % - 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.
187 %
188 % To score a state we subtract the player's score with the opponent's score
189 % This way boards with a large difference will receive a higher score
190 % => If the player has a higher score, the overall score will be positive
191 % => If the opponent has a higher score, the overall score will be negative
192
193 score(Player, State, TraversedDepth, MaxDepth, Score) :- % Checkmate or stalemate
194
195     % Player cannot do any more moves
196     state:checkmate_or_stalemate(State),
197
198     % Checkmate or stalemate
199     score_checkmate_or_stalemate(Player, State, TraversedDepth, MaxDepth, Score).
200
201 score(Player, State, _, _, Score) :- % Symmetric evaluation scoring
    ↳ function
202
203     % Score for the player
204     score_player(Player, State, PlayerScore),
205
206     % Score for the opponent
207     piece:opponent(Player, OpponentPlayer),
208     score_player(OpponentPlayer, State, OpponentScore),
209
210     % Calculate the state score
211     Score is PlayerScore - OpponentScore.
212
213 %! score_player(+Player, +State, -Score)
214 %
215 % Score for a given state for a given player.
216 score_player(Player, State, Score) :-
217     % Get the pieces for the given player
218     state:color_pieces(State, Player, ColorPieces),
219
220     % Evaluate every piece and add it's score to the scores
221     score_pieces(ColorPieces, PiecesScore),
222
223     % Add all the scores together
224     Score = PiecesScore.
225
226
227 %! score_checkmate_or_stalemate(+Player, +State, +TraversedDepth, +MaxDepth, -Score)
228 %
229 % Score when a given player is either checkmate or stalemate.
230 % If the player is checkmate, return the checkmate score.
231 % If the player is stalemate, return the stalemate score.
232 score_checkmate_or_stalemate(Player, State, TraversedDepth, MaxDepth, Score) :- %
    ↳ Checkmate

```

```

233     state:currentcolor(State, CurrentPlayer),
234
235     % Make sure the king is check (otherwise a stalemate is reached)
236     state:check(State, CurrentPlayer),
237
238     % Checkmate score
239     score_checkmate(Player, CurrentPlayer, TraversedDepth, MaxDepth, Score).
240
241 score_checkmate_or_stalemate(Player, State, _, _, Score) :-                                %
242     ⇨ Stalemate
243     state:currentcolor(State, CurrentPlayer),
244
245     % Checkmate score
246     score_stalemate(Player, CurrentPlayer, Score).
247
248 %! score_checkmate(+Player, +CheckmatePlayer, +TraversedDepth, +MaxDepth, -Score)
249 %
250 % Score when a given state is checkmate.
251 score_checkmate(Player, CheckmatePlayer, TraversedDepth, MaxDepth, Score) :-
252     (
253         Player == CheckmatePlayer, PartialScore = -10000
254         ;
255         PartialScore = 10000
256     ),
257
258     % Apply a depth penalty to the checkmate.
259     % This will make sure the chosen nextmove will be based on the branch with the
260     ⇨ quickest possible checkmate.
261     ScorePenalty is MaxDepth - TraversedDepth, % This value will become smaller when the
262     ⇨ traversed depth will increase
263     Score is PartialScore + ScorePenalty.
264
265 %! score_stalemate(+Player, +StalematePlayer, -Score)
266 %
267 % Score when a given state is stalemate.
268 score_stalemate(_, _, 0).
269
270 %! score_pieces(+Pieces, -Score)
271 %
272 % Based on Hans Berliner's System.
273 % Score a set of pieces.
274 score_pieces([Piece | Pieces], Score) :-
275
276     % Score for the current piece
277     score_piece(Piece, PieceScore),
278
279     % Recursive call
280     score_pieces(Pieces, PiecesScore),
281
282     % Add the scores together

```

```

283     Score is PieceScore + PiecesScore, !.
284 score_pieces([], 0).
285
286
287 %! score_piece(+Piece, -Score)
288 %
289 % Score a single piece.
290 % Scores are based on the values recommended by Hans Berliner's system (World
    ↪ Correspondence Chess Champion)
291
292 % Queen
293 score_piece(piece(_, queen, _), 8.8).
294
295 % Tower
296 score_piece(piece(_, tower, _), 5.1).
297
298 % Bishop
299 score_piece(piece(_, bishop, _), 3.33).
300
301 % Horse
302 score_piece(piece(_, horse, _), 3.2).
303
304 % Pawn
305 score_piece(piece(white, pawn, X/Y), Score) :-    % white: Get score from pawn table
306
307     % Scoring table
308     score_pawn_table(ScoringTable),
309
310     % Row
311     nth0(Y, ScoringTable, Row),
312
313     % Score
314     nth0(X, Row, Score).
315
316 score_piece(piece(black, pawn, X/Y), Score) :-    % black: Get score from pawn table
317     XRev is 8 - X,
318     YRev is 8 - Y,
319
320     % Scoring table
321     score_pawn_table(ScoringTable),
322
323     % Row
324     nth0(YRev, ScoringTable, Row),
325
326     % Score
327     nth0(XRev, Row, Score).
328
329 score_piece(piece(_, pawn, _), 1).                % Default value
330
331 % Default
332 score_piece(piece(_, _, _), 0).
333
334

```



```

335  %! score_pawn_table(+ScoringTable)
336  %
337  % Scoring table for scoring pawns (from the perspective of the white player)
338  %
339  % Pawns will receive a higher score as they advance.
340  % This will prevent them from staying center, blocking other pieces to move forward.
341  % Based on Hans Berliner's System.
342  score_pawn_table([
343      [0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00],
344      [0.90, 0.95, 1.05, 1.10, 1.10, 1.05, 0.95, 0.90],
345      [0.90, 0.95, 1.05, 1.15, 1.15, 1.05, 0.95, 0.90],
346      [0.90, 0.95, 1.10, 1.20, 1.20, 1.10, 0.95, 0.90],
347      [0.97, 1.03, 1.17, 1.27, 1.27, 1.17, 1.03, 0.97],
348      [1.06, 1.12, 1.25, 1.40, 1.40, 1.25, 1.12, 1.06]
349  ]).

```

Listing 2: src/state.pl

```

1 :- module(state, []).
2
3 :- use_module("move").
4 :- use_module("piece").
5 :- use_module("position").
6
7
8 %! board(+State, -Board)
9 %
10 % Extract the board from the given state.
11 board(state(Board, _, _, _), Board).
12
13
14 %! currentcolor(+State, -CurrentColor)
15 %
16 % Extract the current color (the player that can do the current move) from the given
    ⇨ state.
17 currentcolor(state(_, CurrentColor, _, _), CurrentColor).
18
19
20 %! nextcolor(+State, -NextColor)
21 %
22 % Extract the next color (the player that can do a move after the current player did a
    ⇨ move) from the given state.
23 nextcolor(state(_, white, _, _), black).
24 nextcolor(state(_, black, _, _), white).
25
26
27 %! rokades(+State, -Rokades)
28 %
29 % Extract the remaining rokades from the given state.
30 rokades(state(_, _, Rokades, _), Rokades).
31
32
33 %! passant(+State, -Passant)
34 %
35 % Extract the en-passant possibility from the given state.
36 passant(state(_, _, _, Passant), Passant).
37
38
39 %! empty_state(+CurrentColor, +Rokades, +Passant, -State)
40 %
41 % Create a state with an empty board.
42 empty_state(CurrentColor, Rokades, Passant, State) :-
43
44     % Create an empty board
45     Board = rows(
46         row(none, none, none, none, none, none, none, none),
47         row(none, none, none, none, none, none, none, none),
48         row(none, none, none, none, none, none, none, none),

```

```

49         row(none, none, none, none, none, none, none, none),
50         row(none, none, none, none, none, none, none, none),
51         row(none, none, none, none, none, none, none, none),
52         row(none, none, none, none, none, none, none, none),
53         row(none, none, none, none, none, none, none, none)
54     ),
55
56     % Create the state
57     State = state(Board, CurrentColor, Rokades, Passant).
58
59
60     %! create_state(+Pieces, +CurrentColor, +Rokades, +Passant, -State)
61     %
62     % Create a state from a list of pieces
63     create_state(Pieces, CurrentColor, Rokades, Passant, State) :-
64
65         % Create an empty state
66         empty_state(CurrentColor, Rokades, Passant, EmptyState),
67
68         % Set the pieces
69         set_pieces(EmptyState, Pieces, State).
70
71
72     %! all_possible_states/2(+CurrentState, -NextStates)
73     %
74     % Generate all possible next states for a given state
75     all_possible_states(CurrentState, NextStates) :-
76         state:currentcolor(CurrentState, CurrentColor),
77
78         % All pseudo-possible moves for the next player
79         move:all_possible_moves(CurrentColor, CurrentState, NextMoves),
80
81         % Helper predicate
82         all_possible_states(CurrentState, NextMoves, NextStates).
83
84
85     %! all_possible_states/3(+CurrentState, +Moves, ?NextStates)
86     %
87     % Generate a new state for every possible move and append it to a list.
88     all_possible_states(CurrentState, [Move | Moves], NextStates) :- % Valid pseudo-move
89         state:currentcolor(CurrentState, CurrentColor),
90
91         % Do the move and retrieve the new state
92         move:do_move(Move, CurrentState, NextState),
93
94         (
95             % State is check and should not be included
96             state:check(NextState, CurrentColor),
97             OtherStates = NextStates
98             ;
99             % State is not check and should be included
100             [NextState | OtherStates] = NextStates
101         ),

```

```

102
103     % Recursive call
104     all_possible_states(CurrentState, Moves, OtherStates), !.
105 all_possible_states(_, [], []). % Base case
106
107
108 %! pieces/2(+State, -Pieces)
109 %
110 % List of pieces that are currently on the board
111 pieces(State, Pieces) :-
112
113     % Find all possible positions on the board
114     position:valid_positions(Positions),
115
116     % Helper predicate
117     pieces(State, Positions, Pieces).
118
119
120 %! pieces/3(+State, +Positions, -Pieces)
121 %
122 % Helper predicate for pieces/3
123 pieces(State, [Position | Positions], [Piece | Pieces]) :- % Piece at current position is
    ↪ not "none"
124
125     % Piece at the current position
126     piece_at_position(State, Position, Piece),
127
128     % Piece must not be none
129     Piece \== none,
130
131     % Recursive call
132     pieces(State, Positions, Pieces), !.
133 pieces(State, [_ | Positions], Pieces) :- % Piece at current position is
    ↪ "none"
134
135     % Recursive call
136     pieces(State, Positions, Pieces), !.
137 pieces(_, [], []). % Base case
138
139
140 %! color_pieces/3(+State, +Color, -ColorPieces)
141 %
142 % Unify all pieces for a given Color from the given state.
143 color_pieces(State, Color, ColorPieces) :-
144     % Find all possible positions on the board
145     position:valid_positions(Positions),
146
147     % Helper predicate
148     color_pieces(State, Color, Positions, ColorPieces).
149
150
151 %! color_pieces/4(+State, +Color, +Positions, -ColorPieces)
152 %

```

```

153 % Helper predicate for color_pieces/3
154 color_pieces(State, Color, [Position | Positions], [ColorPiece | ColorPieces]) :- %
    ↳ Piece at current position is of the given color
155
156 % Piece at the current position
157 piece_at_position(State, Position, ColorPiece),
158
159 % Color must match
160 piece:color(ColorPiece, Color),
161
162 % Recursive call
163 color_pieces(State, Color, Positions, ColorPieces), !.
164 color_pieces(State, Color, [_ | Positions], ColorPieces) :- % Piece
    ↳ at current position is "none" or not of the given color
165
166 % Recursive call
167 color_pieces(State, Color, Positions, ColorPieces), !.
168 color_pieces(_, _, [], []). % Base case
169
170 %! king/3(+State, +Color, -King).
171 %
172 % Get the king piece for a given color
173 king(State, Color, King) :-
174     % Find all possible positions on the board
175     position:valid_positions(Positions),
176
177     % Helper predicate
178     king(Positions, State, Color, King).
179
180 %! king/4(+Positions, +State, +Color, -King).
181 %
182 % Helper predicate for king/3
183 king([Position | _], State, Color, King) :-
184     % King Piece
185     King = piece(Color, king, Position),
186
187     % Piece at the current position is the king
188     piece_at_position(State, Position, King).
189 king(_ | Positions, State, Color, King) :-
190     % Recursive call
191     king(Positions, State, Color, King).
192
193
194 %! piece_at_position(+State, +X/+Y, -Piece)
195 %
196 % Piece at a given position in a given state.
197 piece_at_position(State, X/Y, Piece) :-
198     state:board(State, Board),
199
200     % Requested row
201     arg(Y, Board, Row),
202
203     % Requested piece

```

```

204     arg(X, Row, Piece).
205
206
207     %! set_piece_at_position(+State, +Piece, +X/+Y, -NewState)
208     %
209     % Set a piece at a given position on the board of the given state.
210     set_piece_at_position(State, Piece, X/Y, NewState) :-
211         state:board(State, Board),
212
213         % Get the row of the given position
214         arg(Y, Board, Row),
215
216         % Duplicate board & row to prevent setarg from altering other states.
217         duplicate_term(Board, NewBoard),
218         duplicate_term(Row, NewRow),
219
220         % Update the position in the row
221         setarg(X, NewRow, Piece),
222
223         % Update the board
224         setarg(Y, NewBoard, NewRow),
225
226         % Update the state
227         set_board(State, NewBoard, NewState), !.
228
229
230     %! set_piece(+State, +Piece, -NewState)
231     %
232     % Set a piece on the board in a given state.
233     set_piece(State, Piece, NewState) :-
234         piece:position(Piece, X/Y),
235
236         % Set the piece at the given position
237         set_piece_at_position(State, Piece, X/Y, NewState).
238
239
240     %! set_pieces(+State, +Pieces, +NewState)
241     %
242     % Set a list of pieces on the board in a given state.
243     set_pieces(State, [], State).
244     set_pieces(State, [Piece | Pieces], NewState) :-
245
246         % Set the piece
247         set_piece(State, Piece, PartialState),
248
249         % Recursive call
250         set_pieces(PartialState, Pieces, NewState), !.
251
252
253     %! remove_piece(+State, +Piece, -NewState)
254     %
255     % Remove a piece from the board in a given state.
256     remove_piece(State, Piece, NewState) :-

```

```

257     piece:position(Piece, X/Y),
258
259     % Set the piece position to none
260     set_piece_at_position(State, none, X/Y, NewState).
261
262
263 %! remove_pieces(+State, +Pieces, -NewState)
264 %
265 % Remove a list of pieces from the board
266 remove_pieces(State, [], State).
267 remove_pieces(State, [Piece | Pieces], NewState) :-
268
269     % Remove the piece
270     remove_piece(State, Piece, PartialState),
271
272     % Recursive call
273     remove_pieces(PartialState, Pieces, NewState), !.
274
275
276 %! remove_rokades(+State, +Rokade, -NewState)
277 %
278 % Remove a rokade from the state
279 remove_rokade(State, Rokade, NewState) :-
280     state:board(State, Board),
281     state:currentcolor(State, CurrentColor),
282     state:rokades(State, Rokades),
283     state:passant(State, Passant),
284
285     % Remove the rokade
286     delete(Rokades, Rokade, NewRokades),
287
288     % Create the new state
289     NewState = state(Board, CurrentColor, NewRokades, Passant).
290
291
292 %! remove_rokades(+State, +Rokades, -NewState)
293 %
294 % Remove a list of rokades from the state
295 remove_rokades(State, [], State).
296 remove_rokades(State, [Rokade | Rokades], NewState) :-
297
298     % Remove the rokade
299     remove_rokade(State, Rokade, PartialState),
300
301     % Recursive call
302     remove_rokades(PartialState, Rokades, NewState), !.
303
304
305 %! set_board(+State, +Board, -NewState)
306 %
307 % Set new board value
308 set_board(State, Board, NewState) :-
309     state:currentcolor(State, CurrentColor),

```

```

310     state:rokades(State, Rokades),
311     state:passant(State, Passant),
312
313     % Create the new state
314     NewState = state(Board, CurrentColor, Rokades, Passant).
315
316
317     %! set_passant(+State, +Passant, -NewState)
318     %
319     % Set new passant value
320     set_passant(State, Passant, NewState) :-
321         state:board(State, Board),
322         state:currentcolor(State, CurrentColor),
323         state:rokades(State, Rokades),
324
325         % Create the new state
326         NewState = state(Board, CurrentColor, Rokades, Passant).
327
328
329     %! set_color(+State, +Color, -NewState)
330     %
331     % Set new color value
332     set_color(State, Color, NewState) :-
333         state:board(State, Board),
334         state:passant(State, Passant),
335         state:rokades(State, Rokades),
336
337         % Create the new state
338         NewState = state(Board, Color, Rokades, Passant).
339
340
341     %! check(+State, +Color)
342     %
343     % If the king of the given color in-check for the given state.
344     % The king is now in range of attack by the opponent player.
345     check(State, Color) :-
346
347         % Retrieve the king from the board
348         state:king(State, Color, KingPiece), !,
349
350         % Check if any opponent piece can attack the king of the given color
351         state:can_be_attacked(KingPiece, State).
352
353
354     %! can_be_attacked/2(+Piece, +State)
355     %
356     % If a given piece can be attacked in the given state.
357     can_be_attacked(Piece, State) :-
358         piece:color(Piece, Color),
359
360         % Opponent Color
361         piece:opponent(Color, OpponentColor),
362

```



```

363     % Opponent Pieces
364     state:color_pieces(State, OpponentColor, OpponentPieces),
365
366     % Helper predicate
367     can_be_attacked_by_pieces(Piece, State, OpponentPieces).
368
369     %! can_be_attacked_by_pieces(+Piece, +State, +OpponentPieces)
370     %
371     % If a given piece can be attacked by a list of opponent pieces.
372     can_be_attacked_by_pieces(Piece, State, [OpponentPiece | _]) :-           % Can be
373     ↪ attacked by moves of current piece
374
375     % Possible moves
376     move:possible_moves(OpponentPiece, State, OpponentMoves),
377
378     % Piece can be attacked by received list of moves
379     can_be_attacked_by_moves(Piece, OpponentMoves).
380
381     can_be_attacked_by_pieces(Piece, State, [_ | OpponentPieces]) :- % Cannot be attacked by
382     ↪ moves of current piece
383
384     % Recursive call
385     can_be_attacked_by_pieces(Piece, State, OpponentPieces).
386
387     %! can_be_attacked_by_moves(+Piece, +Moves)
388     %
389     % If a given piece can be attacked in a given list of moves.
390     can_be_attacked_by_moves(Piece, [Move | _]) :- % Can be attacked
391     move:delete_pieces(Move, DeletePieces),
392
393     % Piece is present inside the move
394     memberchk(Piece, DeletePieces), !.
395
396     can_be_attacked_by_moves(Piece, [_ | Moves]) :- % Cannot be attacked
397
398     % Recursive call
399     can_be_attacked_by_moves(Piece, Moves), !.
400
401
402
403
404     %! checkmate_or_stalemate/1(+State)
405     %
406     % If a given state is checkmate or stalemate for the current player.
407     % Will check if the current player cannot do any more moves.
408     checkmate_or_stalemate(State) :-
409     state:currentcolor(State, Color),
410
411     % Find all possible positions on the board
412     position:valid_positions(Positions),
413
414     % Helper predicate
415     checkmate_or_stalemate(State, Color, Positions).
416
417     %! checkmate_or_stalemate/3(+State, +Color +Positions)
418     %
419     % Helper predicate for checkmate_or_stalemate/1
420     checkmate_or_stalemate(_, _, []).

```

```

414 checkmate_or_stalemate(State, Color, [Position | Positions]) :-      % Piece at position
    ⇨ is of given color
415     Piece = piece(Color, _, Position),
416
417     % Piece at the current position
418     piece_at_position(State, Position, Piece),
419
420     % All possible moves for the current piece.
421     move:possible_moves(Piece, State, PseudoMoves),
422
423     % All possible valid moves for the current piece should be empty.
424     move:valid_moves(State, PseudoMoves, []),
425
426     % Recursive call.
427     checkmate_or_stalemate(State, Color, Positions).
428 checkmate_or_stalemate(State, Color, [Position | Positions]) :-      % Piece at
    ⇨ position is not of given color or none
429
430     % Position is of opponent or empty
431     position:empty_or_opponent_position(Position, Color, State),
432
433     % Recursive call.
434     checkmate_or_stalemate(State, Color, Positions).

```

Listing 3: src/move.pl

```

1 :- module(move, []).
2
3 :- use_module("state").
4 :- use_module("position").
5 :- use_module("piece").
6
7
8 %! delete_pieces(+Move, -DeletePieces)
9 %
10 % List of pieces to delete from the board for a given move
11 delete_pieces(move(DeletePieces, _), DeletePieces).
12
13
14 %! append_pieces(+Move, -AppendPieces)
15 %
16 % List of pieces to append to the board for a given move
17 append_pieces(move(_, AppendPieces), AppendPieces).
18
19
20 %! delete_rokades(+Move, -DeleteRokades)
21 %
22 % List of rokades to delete for a given move
23 delete_rokades(Move, DeleteRokades) :-
24     move:delete_pieces(Move, DeletePieces),
25
26     % Convert a list of pieces into a list of rokades
27     maplist([Piece, Rokades] >> (piece:rokades_piece(Piece, Rokades)), DeletePieces,
28         ↪ PossibleRokades),
29
30     % Merge all possible rokades
31     append(PossibleRokades, DeleteRokades).
32
33 %! new_passant(+Move, -NewPassant)
34 %
35 % Get the new en-passant possibility for a given move
36 new_passant(Move, NewPassant) :-
37     % When moving a piece 2 steps forward (and creating an en-passant possibility)
38     % the piece will never attack an opponent
39     move:delete_pieces(Move, [piece(Color, pawn, OldPosition)]),
40     move:append_pieces(Move, [piece(Color, pawn, NewPosition)]),
41
42     % Find the en-passant possibility
43     position:pawn_start_position(OldPosition, Color),
44     piece:passant_piece(piece(Color, pawn, NewPosition), NewPassant), !.
45 new_passant(_, none).
46
47
48 %! do_move(+Move, +CurrentState, -NewState)
49 %

```

```

50 % Update the state with a given move for a given piece.
51 do_move(Move, CurrentState, NewState) :-
52     move:delete_pieces(Move, DeletePieces),
53     move:append_pieces(Move, AppendPieces),
54     move:delete_rokades(Move, DeleteRokades),
55     move:new_passant(Move, NewPassant),
56
57     % Next color
58     state:nextcolor(CurrentState, NewColor),
59
60     % Delete pieces
61     state:remove_pieces(CurrentState, DeletePieces, PartialState1),
62
63     % Add pieces
64     state:set_pieces(PartialState1, AppendPieces, PartialState2),
65
66     % Delete Rokades
67     state:remove_rokades(PartialState2, DeleteRokades, PartialState3),
68
69     % Update passant
70     state:set_passant(PartialState3, NewPassant, PartialState4),
71
72     % Update color
73     state:set_color(PartialState4, NewColor, NewState).
74
75
76 %! all_possible_moves/2(+State, -Moves)
77 %
78 % All pseudo-possible moves for the current state.
79 all_possible_moves(State, Moves) :-
80     state:currentcolor(State, CurrentColor),
81     all_possible_moves(CurrentColor, State, Moves).
82
83
84 %! all_possible_moves/3(+Color, +State, -Moves)
85 %
86 % All pseudo-possible moves for the current state for a given color.
87 all_possible_moves(Color, State, Moves) :-
88
89     % Get the pieces for the given color
90     state:color_pieces(State, Color, ColorPieces),
91
92     % Get all possible moves for the pieces
93     all_possible_moves_for_pieces(ColorPieces, State, Moves).
94
95
96 %! all_possible_moves_for_pieces(+Pieces, +State, -Moves)
97 %
98 % All pseudo-possible moves for all given pieces in the current state.
99 all_possible_moves_for_pieces([Piece | Pieces], State, Moves) :-
100
101     % All possible moves for the current piece
102     possible_moves(Piece, State, PieceMoves),

```

```

103
104     % Recursive call
105     all_possible_moves_for_pieces(Pieces, State, RestMoves),
106
107     % Merge the moves into the moves list
108     append(PieceMoves, RestMoves, Moves), !.
109 all_possible_moves_for_pieces([], _, []).
110
111 %! valid_move(+State, +Move)
112 %
113 % If a given move is valid.
114 % A move is considered invalid if it causes a check.
115 valid_move(State, Move) :-
116     state:currentcolor(State, CurrentColor),
117
118     % Do the move and retrieve the new state
119     move:do_move(Move, State, NextState),
120
121     % State is check
122     not(state:check(NextState, CurrentColor)).
123
124
125 %! valid_moves(+State, +Moves, ?ValidMoves)
126 %
127 % Filter a given list of moves by removing all moves that lead to a check of the current
128 ↪ color.
129 valid_moves(_, [], []).
130 valid_moves(State, [Move | Moves], ValidMoves) :- % Move is valid
131
132     % Check if move is valid
133     % This cut operator is to prevent having to use "not(valid_move(...))" in the
134     % next predicate. This is for performance reasons.
135     valid_move(State, Move), !,
136
137     % Assign the valid move
138     ValidMoves = [Move | ValidMovesRest],
139
140     % Recursive call
141     valid_moves(State, Moves, ValidMovesRest).
142 valid_moves(State, [_ | Moves], ValidMoves) :- % Move is invalid
143     % Recursive call
144     valid_moves(State, Moves, ValidMoves).
145
146
147 %! possible_moves(+Piece, +State, -Moves)
148 %
149 % All psuedo-possible moves for a specific piece in the given state.
150 % This predicate will also include moves that cause a potential in-check situation.
151
152 % King
153 possible_moves(Piece, State, Moves) :-
154     piece:type(Piece, king),

```

```

155     % Rokades
156     findall(Move, rokades_move(Piece, State, Move), RokadesMoves),
157
158     % King can move in a square
159     square_moves(Piece, State, SquareMoves),
160
161     % Merge possible moves
162     append([RokadesMoves, SquareMoves], Moves), !.
163
164 % Queen
165 possible_moves(Piece, State, Moves) :-
166     piece:type(Piece, queen),
167
168     % Queen can move diagonally or in a cross
169     cross_moves(Piece, State, CrossMoves),
170     diagonal_moves(Piece, State, DiagonalMoves),
171
172     % Merge possible moves
173     append([CrossMoves, DiagonalMoves], Moves), !.
174
175 % Tower
176 possible_moves(Piece, State, Moves) :-
177     piece:type(Piece, tower),
178
179     % Tower can move in a cross
180     cross_moves(Piece, State, Moves), !.
181
182 % Bishop
183 possible_moves(Piece, State, Moves) :-
184     piece:type(Piece, bishop),
185
186     % Bishop can move in diagonally.
187     diagonal_moves(Piece, State, Moves), !.
188
189 % Horse
190 possible_moves(Piece, State, Moves) :-
191     piece:type(Piece, horse),
192
193     % Horse positions
194     findall(Position, position:horse_position(Piece, State, Position), Positions),
195
196     % Convert positions into moves
197     positions_to_moves(Piece, State, Positions, Moves), !.
198
199 % Pawn
200 possible_moves(Piece, State, Moves) :-
201     piece:type(Piece, pawn),
202
203     % Possible moves
204     pawn_forward_moves(Piece, State, ForwardMoves),
205     pawn_diagonal_moves(Piece, State, DiagonalMoves),
206     pawn_passant_moves(Piece, State, PassantMoves),
207

```

```

208 % Merge possible moves
209 append([ForwardMoves, DiagonalMoves, PassantMoves], MergedMoves),
210
211 % Handle potential pawn promotional moves
212 convert_promotion_moves(Piece, MergedMoves, Moves), !.
213
214
215 %! convert_promotion_moves(+Piece, +Moves, +PromotionMoves)
216 %
217 % Convert a list of moves to a list of promotion moves.
218 % Will scan every move, check if the piece can be promoted, and create the correct
    ↳ promotions
219 convert_promotion_moves(Piece, [Move | Moves], PromotionMoves) :- % Current move is
    ↳ promotion move
220     piece:color(Piece, Color),
221     Move = move>DeletePieces, AppendPieces),
222
223 % Select the pawn
224 select(piece(Color, pawn, NewPosition), AppendPieces, _),
225
226 % Check if the pawn position is a promotion position
227 position:pawn_promotion_position(NewPosition, Color),
228
229 % Recursive call
230 convert_promotion_moves(Piece, Moves, PromotionMovesRest),
231
232 % Create the promotion moves
233 PromotionMovesCurrent = [
234     move>DeletePieces, [piece(Color, queen, NewPosition)]),
235     move>DeletePieces, [piece(Color, horse, NewPosition)]),
236     move>DeletePieces, [piece(Color, tower, NewPosition)]),
237     move>DeletePieces, [piece(Color, bishop, NewPosition)]),
238 ],
239
240 % Merge
241 append([PromotionMovesCurrent, PromotionMovesRest], PromotionMoves), !.
242
243 convert_promotion_moves(Piece, [Move | Moves], [PromotionMove | PromotionMoves]) :- %
    ↳ Current move is not a promotion move
244
245 % Add the old move to the promotion moves
246 % since the original moves, that are no promotions, must be included as well
247 PromotionMove = Move,
248
249 % Recursive call
250 convert_promotion_moves(Piece, Moves, PromotionMoves), !.
251
252 convert_promotion_moves(_, [], []) :- !. % Base Case
253
254
255 %! pawn_moves(+Piece, +State, -Moves)
256 %
257 % Moves for the pawn going forward

```

```

258 pawn_forward_moves(Piece, State, [Move1, Move2]) :- % Pawn on start position (can move 2
    ↳ steps forward)
259     piece:type(Piece, pawn),
260     piece:color(Piece, Color),
261     piece:position(Piece, CurrentPosition),
262
263     % Pawn must be on start position
264     position:pawn_start_position(CurrentPosition, Color),
265
266     % First position must be valid & empty (otherwise the pawn is not able to move 2
    ↳ steps forward)
267     position:forward_position(CurrentPosition, Color, NewPosition1),
268     position:valid_position(NewPosition1),
269     position:empty_position(NewPosition1, State),
270
271     % Second position must be valid & empty
272     position:forward_position(NewPosition1, Color, NewPosition2),
273     position:valid_position(NewPosition2),
274     position:empty_position(NewPosition2, State),
275
276     % Create the moves
277     create_move(CurrentPosition, NewPosition1, State, Move1),
278     create_move(CurrentPosition, NewPosition2, State, Move2), !.
279
280 pawn_forward_moves(Piece, State, [Move1]) :- % Pawn (can move max 1 step forward)
281     piece:type(Piece, pawn),
282     piece:position(Piece, CurrentPosition),
283     piece:color(Piece, Color),
284
285     % Forward position must be valid & empty
286     position:forward_position(CurrentPosition, Color, NewPosition1),
287     position:valid_position(NewPosition1),
288     position:empty_position(NewPosition1, State),
289
290     % Create the moves
291     create_move(CurrentPosition, NewPosition1, State, Move1), !.
292
293 pawn_forward_moves(Piece, _, []) :- % Pawn cannot move forward
294     piece:type(Piece, pawn),
295     !.
296
297 %! pawn_diagonal_moves(+Piece, +State, -Moves)
298 %
299 % Moves for the given pawn moving diagonally
300 pawn_diagonal_moves(Piece, State, Moves) :-
301
302     % Moves for both diagonal parts
303     pawn_diagonal_moves_part(Piece, State, -1, LeftMoves),
304     pawn_diagonal_moves_part(Piece, State, 1, RightMoves),
305
306     % Merge the 2 lists
307     append([LeftMoves, RightMoves], Moves).
308

```



```

309
310 %! pawn_diagonal_moves_part(+Piece, +State, +XDifference, -Moves)
311 %
312 % Moves for the given pawn moving diagonally either left or right.
313 % XDifference = 1: right diagonal move
314 % XDifference = -1: left diagonal move
315 pawn_diagonal_moves_part(Piece, State, XDifference, [Move]) :- % Left diagonal
316     piece:type(Piece, pawn),
317     piece:position(Piece, X/Y),
318     piece:color(Piece, Color),
319
320     % New position
321     XNew is X + XDifference,
322     position:forward_position(XNew/Y, Color, XNew/YNew),
323
324     % New position must be valid
325     position:valid_position(XNew/YNew),
326
327     % New position must be taken by an opponent piece
328     position:opponent_position(XNew/YNew, Color, State),
329
330     % Create the move
331     create_move(X/Y, XNew/YNew, State, Move), !.
332
333 pawn_diagonal_moves_part(Piece, _, _, []) :-
334     piece:type(Piece, pawn), !.
335
336
337 %! pawn_passant_moves(+Piece, +State, -Moves)
338 %
339 % Move for the given pawn if an en-passant move is possible
340 pawn_passant_moves(Piece, State, Moves) :-
341     piece:type(Piece, pawn),
342     piece:color(Piece, PieceColor),
343     state:passant(State, Passant),
344
345     Passant = passant(PassantColor, _),
346
347     % En-passant position must be for the opponent
348     piece:opponent(PieceColor, PassantColor),
349
350     % En-passant for both directions
351     pawn_passant_moves_part(Piece, State, -1, LeftMoves),
352     pawn_passant_moves_part(Piece, State, 1, RightMoves),
353
354     % Merge the 2 lists
355     append([LeftMoves, RightMoves], Moves), !.
356
357 pawn_passant_moves(Piece, _, []) :-
358     piece:type(Piece, pawn), !.
359
360
361 %! pawn_passant_moves_part(+Piece, +Passant, +XDifference, -Moves)

```

```

362 %
363 % Moves for the given pawn doing en-passant either left or right
364 % XDifference = 1: right en-passant move
365 % XDifference = -1: left en-passant move
366 pawn_passant_moves_part(Piece, State, XDifference, [Move]) :-
367     piece:type(Piece, pawn),
368     piece:color(Piece, PieceColor),
369     piece:position(Piece, X/Y),
370     state:passant(State, Passant),
371
372     Passant = passant(PassantColor, XPassant/YPassant),
373
374     % Check if the passant possibility is next to piece.
375     XPassant is X + XDifference,
376     position:forward_position(XPassant/Y, PieceColor, XPassant/YPassant),
377
378     % New position of the pawn after en-passant
379     position:forward_position(XPassant/Y, PieceColor, XNew/YNew),
380
381     % Piece to remove by doing the en-passant move
382     OpponentPiece = piece(PassantColor, pawn, XPassant/Y),
383
384     % Create the move
385     Move = move([Piece, OpponentPiece], [piece(PieceColor, pawn, XNew/YNew)]), !.
386
387 pawn_passant_moves_part(Piece, _, _, []) :-
388     piece:type(Piece, pawn), !.
389
390
391 %! rokades_move(+King, +State, -Moves)
392 %
393 % Rokades move for the king
394 rokades_move(King, State, Move) :- % Short rokade
395     piece:color(King, Color),
396     piece:position(King, KingPosition),
397     state:rokades(State, Rokades),
398     _/Y = KingPosition,
399
400     % Short rokade
401     ShortRokade = rokade(Color, short),
402     memberchk(ShortRokade, Rokades),
403
404     % Tower for rokade
405     Tower = piece(Color, tower, TowerPosition),
406     piece:rokades_piece(Tower, [ShortRokade]),
407
408     % Check if the pieces between the tower and king are empty
409     position:empty_between_positions(KingPosition, TowerPosition, State),
410
411     % New pieces
412     NewKing = piece(Color, king, 7/Y),
413     NewTower = piece(Color, tower, 6/Y),
414

```

```

415     % Create the move
416     Move = move([King, Tower], [NewKing, NewTower]).
417
418     rokades_move(King, State, Move) :-    % Long rokade
419         piece:color(King, Color),
420         piece:position(King, KingPosition),
421         state:rokades(State, Rokades),
422         _/Y = KingPosition,
423
424     % Long rokade
425     LongRokade = rokade(Color, long),
426     memberchk(LongRokade, Rokades),
427
428     % Tower for rokade
429     Tower = piece(Color, tower, TowerPosition),
430     piece:rokades_piece(Tower, [LongRokade]),
431
432     % Check if the pieces between the tower and king are empty
433     position:empty_between_positions(TowerPosition, KingPosition, State),
434
435     % New pieces
436     NewKing = piece(Color, king, 3/Y),
437     NewTower = piece(Color, tower, 4/Y),
438
439     % Create the move
440     Move = move([King, Tower], [NewKing, NewTower]).
441
442
443     %! square_moves(+Piece, +State, -Moves)
444     %
445     % Moves in a square around a given piece
446     square_moves(Piece, State, Moves) :-
447
448         % Square positions
449         findall(Position, position:square_position(Piece, State, Position), Positions),
450
451         % Convert positions into moves
452         positions_to_moves(Piece, State, Positions, Moves).
453
454
455     %! cross_moves(+Piece, +State, -Moves)
456     %
457     % Moves in a cross starting from a given piece
458     cross_moves(Piece, State, Moves) :-
459
460         path_moves(Piece, State, 1, 0, RightMoves),    % Right row part
461         path_moves(Piece, State, -1, 0, LeftMoves),    % Left row part
462         path_moves(Piece, State, 0, 1, TopMoves),      % Top column part
463         path_moves(Piece, State, 0, -1, BottomMoves),  % Bottom column part
464
465         % Merge lists
466         append([RightMoves, LeftMoves, TopMoves, BottomMoves], Moves).
467

```

```

468
469 %! diagonal_moves(+Piece, +State, -Moves)
470 %
471 % Moves on the diagonals starting from a given piece
472 diagonal_moves(Piece, State, Moves) :-
473     path_moves(Piece, State, 1, 1, TopRightMoves),      % Top-right diagonal
474     path_moves(Piece, State, -1, 1, TopLeftMoves),      % Top-right diagonal
475     path_moves(Piece, State, 1, -1, BottomRightMoves),  % Bottom-right diagonal
476     path_moves(Piece, State, -1, -1, BottomLeftMoves), % Bottom-left diagonal
477
478     % Merge lists
479     append([TopRightMoves, TopLeftMoves, BottomRightMoves, BottomLeftMoves], Moves).
480
481 %! path_moves/5(+Piece, +State, +XDirection, +YDirection, -Moves)
482 %
483 % Moves on a given path starting from a piece and with incremental addition of
484 ↪ (XDirection, YDirection)
485 % Will stop the path when a new position is either invalid or blocked by another piece
486 path_moves(Piece, State, XDirection, YDirection, Moves) :-
487     piece:position(Piece, X/Y),
488
489     path_moves(Piece, X/Y, State, XDirection, YDirection, Moves).
490
491 %! path_moves/6(+StartPiece, +PreviousPosition, +State, +XDirection, +YDirection, -Moves)
492 %
493 % Helper function for path_moves/5.
494 % Uses a StartPiece to correctly form the moves.
495 path_moves(StartPiece, X/Y, State, XDirection, YDirection, [Move | Moves]) :-
496
497     % Unify the new position
498     XNew is X + XDirection,
499     YNew is Y + YDirection,
500
501     % Create the move
502     create_piece_move(StartPiece, XNew/YNew, State, Move),
503
504     % New position must be valid
505     position:valid_position(XNew/YNew),
506
507     % New position must be empty
508     position:empty_position(XNew/YNew, State), !,
509
510     % Recursively extend the diagonal
511     path_moves(StartPiece, XNew/YNew, State, XDirection, YDirection, Moves).
512 path_moves(StartPiece, X/Y, State, XDirection, YDirection, [Move]) :-
513     piece:color(StartPiece, Color),
514
515     % Unify the new position
516     XNew is X + XDirection,
517     YNew is Y + YDirection,
518
519     % Create the move

```

```

520     create_piece_move(StartPiece, XNew/YNew, State, Move),
521
522     % New position must be valid
523     position:valid_position(XNew/YNew),
524
525     % New position must be taken by the opponent
526     position:opponent_position(XNew/YNew, Color, State), !.
527 path_moves(_, _, _, _, _, []).
528
529 %! positions_to_moves(+Piece, +Stat, +Positions, -Moves)
530 %
531 % Corresponding moves for a given set of positions
532 positions_to_moves(Piece, State, [NextPosition | NextPositions], [Move | Moves]) :-
533
534     % Construct the move
535     create_piece_move(Piece, NextPosition, State, Move),
536
537     % Recursive Call
538     positions_to_moves(Piece, State, NextPositions, Moves), !.
539 positions_to_moves(_, _, [], []).
540
541
542 %! create_move/4(+CurrentPosition, +NewPosition, +State, -Move)
543 %
544 % Create a move from a given position to a new position.
545 create_move(CurrentPosition, NewPosition, State, Move) :-
546     state:piece_at_position(State, CurrentPosition, CurrentPiece),
547     create_piece_move(CurrentPiece, NewPosition, State, Move).
548
549
550 %! create_move/4(+CurrentPiece, +NewPosition, +State, -Move)
551 %
552 % Create a move for a given piece, position and en-passant possibility
553 create_piece_move(CurrentPiece, NewPosition, State, Move) :- % Opponent on new position
554     piece:color(CurrentPiece, Color),
555     piece:type(CurrentPiece, Type),
556
557     % Opponent at the new position
558     position:opponent_position(NewPosition, Color, State, OpponentPiece),
559
560     % Create the new piece
561     NewPiece = piece(Color, Type, NewPosition),
562
563     % Unify the move
564     Move = move([CurrentPiece, OpponentPiece], [NewPiece]), !.
565 create_piece_move(CurrentPiece, NewPosition, State, Move) :- % No piece on new position
566     piece:color(CurrentPiece, Color),
567     piece:type(CurrentPiece, Type),
568
569     % Empty new position
570     position:empty_position(NewPosition, State),
571
572     % Create the new piece

```

```
573     NewPiece = piece(Color, Type, NewPosition),
574
575     % Unify the move
576     Move = move([CurrentPiece], [NewPiece]), !.
```

Listing 4: src/piece.pl

```

1 :- module(piece, []).
2
3 :- use_module("position").
4 :- use_module("move").
5
6
7 %! position(+Piece, -Position)
8 %
9 % Extract the position from the given piece.
10 position(piece(_, _, Position), Position).
11
12
13 %! type(+Piece, -Type)
14 %
15 % Extract the type from the given piece.
16 type(piece(_, Type, _), Type).
17
18
19 %! type(+Color, -Type)
20 %
21 % Extract the color from the given piece.
22 color(piece(Color, _, _), Color).
23
24
25 %! rokades_piece(+Piece, -Rokades)
26 %
27 % List of rokades for a given piece
28 rokades_piece(piece(white, tower, 1/1), [rokade(white, long)]). %
29   ⇨ Tower
30 rokades_piece(piece(white, tower, 8/1), [rokade(white, short)]). %
31   ⇨ Tower
32 rokades_piece(piece(white, king, 5/1), [rokade(white, long), rokade(white, short)]). %
33   ⇨ King
34
35 rokades_piece(piece(black, tower, 1/8), [rokade(black, long)]). %
36   ⇨ Tower
37 rokades_piece(piece(black, tower, 8/8), [rokade(black, short)]). %
38   ⇨ Tower
39 rokades_piece(piece(black, king, 5/8), [rokade(black, long), rokade(black, short)]). %
40   ⇨ King
41
42 rokades_piece(_, []). % Base case
43
44
45 %! passant_piece(+Piece, -Passant)
46 %
47 % En-passant possibility for a given piece
48 passant_piece(piece(white, pawn, X/4), passant(white, X/3)).
49 passant_piece(piece(black, pawn, X/5), passant(black, X/6)).
50 passant_piece(_, none).
```

```

45
46
47 %! row_pieces(+Y, +Pieces, -RowPieces)
48 %
49 % List of pieces for a given row.
50 row_pieces(Y, [Piece | Pieces], [RowPiece | RowPieces]) :- % Match
51     position(Piece, _/PieceY),
52
53     % Row numbers must match
54     PieceY == Y,
55
56     % Append to the list
57     RowPiece = Piece, !,
58
59     % Recursive call
60     row_pieces(Y, Pieces, RowPieces), !.
61 row_pieces(Y, [Piece | Pieces], RowPieces) :- % No match
62     position(Piece, _/PieceY),
63
64     % Row numbers must not match
65     PieceY \== Y,
66
67     % Recursive call
68     row_pieces(Y, Pieces, RowPieces), !.
69 row_pieces(_, [], []).
70
71
72 %! sorted_pieces/2(+Pieces, -SortedPieces)
73 %
74 % Sorted list for a given list of pieces by X-coordinate.
75 sorted_pieces(Pieces, SortedPieces) :-
76     sorted_pieces(Pieces, 1, SortedPieces).
77
78 %! sorted_pieces/3(+Pieces, +X, -SortedPieces)
79 %
80 % Helper predicate for sorted_pieces/2.
81 sorted_pieces(Pieces, X, [SortedPiece | SortedPieces]) :-
82
83     % X must be valid
84     between(1, 8, X),
85
86     % Select the piece with current X coordinate, if any
87     select(piece(Color, Type, X/Y), Pieces, _),
88
89     % Add the piece
90     SortedPiece = piece(Color, Type, X/Y),
91
92     % Recursive call
93     XNext is X + 1,
94     sorted_pieces(Pieces, XNext, SortedPieces), !.
95 sorted_pieces(Pieces, X, SortedPieces) :-
96
97     % X must be valid

```



```
98     between(1, 8, X),
99
100     % Recursive call
101     XNext is X + 1,
102     sorted_pieces(Pieces, XNext, SortedPieces), !.
103 sorted_pieces(_, _, []).
104
105
106 %! opponent(+Color, -OpponentColor)
107 %
108 % Opponent color for a given color
109 opponent(white, black).
110 opponent(black, white).
```

Listing 5: src/position.pl

```

1 :- module(position, []).
2
3 :- use_module("state").
4 :- use_module("piece").
5 :- use_module("util/utils").
6
7
8 %! pawn_start_position(+X/+Y, +Color)
9 %
10 % Pawn is on it's starting position.
11 pawn_start_position(_/2, white).
12 pawn_start_position(_/7, black).
13
14
15 %! pawn_promotion_position(+X/+Y, +Color)
16 %
17 % Pawn is on it's promotion position.
18 pawn_promotion_position(_/8, white).
19 pawn_promotion_position(_/1, black).
20
21
22 %! forward_position(+X/+Y, +Color, +X/-Y)
23 %
24 % Forward for a given piece
25 % For white piece: +1
26 % For black piece: -1
27 forward_position(X/Y, white, X/YNew) :- YNew is Y + 1.
28 forward_position(X/Y, black, X/YNew) :- YNew is Y - 1.
29
30
31 %! horse_position(+Piece, +State, -Position)
32 %
33 % Move that could be done by the horse from a given piece
34 horse_position(piece(Color, _, X/Y), State, XPos/YPos) :-
35
36     % Positions in a square the current position
37     % (X/Y) will also be unified
38     utils:between2(X, XPos),
39     utils:between2(Y, YPos),
40
41     % New position must be valid
42     valid_position(XPos/YPos),
43
44     % Position must be empty or taken by an opponent piece
45     empty_or_opponent_position(XPos/YPos, Color, State),
46
47     % Difference in positions
48     XDiff is X - XPos,
49     YDiff is Y - YPos,
50

```

```

51     % Possible differences for the move
52     PossibleDifferences = [
53         (-1, 2),
54         (-2, 1),
55         (1, 2),
56         (2, 1),
57         (-2, -1),
58         (-1, -2),
59         (2, -1),
60         (1, -2)
61     ],
62
63     % Difference must be a member of the possible differences
64     memberchk((XDiff, YDiff), PossibleDifferences).
65
66
67     %! square_position(+Piece, +State, -XPos/-YPos)
68     %
69     % Position in a square around a given piece
70     square_position(Piece, State, XPos/YPos) :-
71         piece:color(Piece, Color),
72         piece:position(Piece, X/Y),
73
74         % Positions in a square the current position
75         % (X/Y) will also be unified
76         utils:between1(X, XPos),
77         utils:between1(Y, YPos),
78
79         % Position must not be (X/Y)
80         XPos/YPos \== X/Y,
81
82         % New position must be valid
83         valid_position(XPos/YPos),
84
85         % New position must be empty or taken by an opponent piece
86         empty_or_opponent_position(XPos/YPos, Color, State).
87
88
89     %! valid_position(+X/+Y)
90     %
91     % Check if a give coordinate is a valid position on the board for a piece to move to.
92     % Will check if the position is on the board (not outside).
93     %
94     % WARNING: This predicate will not check if the position is allowed for the particular
95     % ↪ piece type!
96     :- table valid_position/1. % Memoization
97     valid_position(X/Y) :-
98
99         % X must be inside the board
100         between(1, 8, X),
101
102         % Y must be inside the board
103         between(1, 8, Y).

```

```

103
104
105 %! valid_positions(-Positions)
106 %
107 % List of all possible positions on the board
108 %
109 % This could also be done using "findall", but hard-coding this makes it significantly
    ⇨ faster when alpha-beta pruning.
110 :- table valid_positions/1. % Memoization
111 valid_positions(Positions) :-
112     forall(X/Y, valid_position(X/Y), Positions).
113
114
115 %! empty_position(+X/+Y, +State)
116 %
117 % Check if a given position is not taken by a piece.
118 empty_position(X/Y, State) :-
119
120     % Piece at the given position must be none
121     state:piece_at_position(State, X/Y, none).
122
123
124 %! opponent_position/3(+X/+Y, +Color, +State)
125 %
126 % Check if a given position is taken by a piece of the opponent player.
127 opponent_position(X/Y, Color, State) :-
128
129     % Opponent color
130     piece:opponent(Color, OpponentColor),
131
132     % Piece at the given position must be of the opponents color
133     state:piece_at_position(State, X/Y, piece(OpponentColor, _, _)).
134
135
136 %! opponent_position/4(+X/+Y, +Color, +State, -OpponentPiece)
137 %
138 % Check if a given position is taken by a piece of the opponent player.
139 % Unify the piece with OpponentPiece.
140 opponent_position(X/Y, Color, State, OpponentPiece) :-
141
142     % Opponent color
143     piece:opponent(Color, OpponentColor),
144
145     % Piece at the given position must be as described above
146     state:piece_at_position(State, X/Y, piece(PieceColor, PieceType, _)),
147
148     % Piece color must match opponent color
149     PieceColor == OpponentColor,
150
151     % Opponent Piece
152     OpponentPiece = piece(OpponentColor, PieceType, X/Y).
153
154

```

```

155  %! empty_or_opponent_position(+X/+Y, +Color, +State)
156  %
157  % Check if a position is empty or taken by a piece of the opponent player.
158  empty_or_opponent_position(X/Y, _, State) :- empty_position(X/Y, State).
159  empty_or_opponent_position(X/Y, Color, State) :- opponent_position(X/Y, Color, State).
160
161
162  %! empty_between_positions(+X1/+Y, +X2/+Y, +State)
163  %
164  % If the positions between 2 coordinates (in a row line) are empty
165  empty_between_positions(X/Y, X/Y, _). % Base Case
166  empty_between_positions(X1/Y, X2/Y, _) :- % No positions between the given
    ↪ positions
167      XPlus is X1 + 1,
168
169      % There are no positions between the X1 & X2
170      XPlus == X2.
171  empty_between_positions(X1/Y, X2/Y, State) :- % Recursion Case
172      % X1 must be smaller than X2
173      X1 < X2,
174
175      % Position must be between X1 and X2
176      XPos is X1 + 1,
177
178      % New position must be empty
179      empty_position(XPos/Y, State),
180
181      % Recursive call
182      empty_between_positions(XPos/Y, X2/Y, State).

```

Listing 6: src/io/parser.pl

```

1 :- module(parser, []).
2
3 :- use_module(library(pio)).
4 :- use_module(library(dcg/basics)).
5 :- use_module("../state").
6
7 % Interpret quoted strings as ASCII character codes.
8 :- set_prolog_flag(double_quotes, codes).
9
10
11 %! parse_state(-State, StartColor)
12 %
13 % Parse a chess game state.
14 parse_state(State) -->
15     parse_rows(8, PiecesList, RokadesList, Passant, StartColor),
16     parse_final_row,
17     {
18         % Create a flat list of pieces
19         append(PiecesList, Pieces),
20
21         % Create a flat list of rokades
22         append(RokadesList, Rokades),
23
24         % Create state
25         state:create_state(Pieces, StartColor, Rokades, Passant, State),
26
27         % Set passant to "none" if no en-passant move was unified
28         (Passant = none, ! ; true)
29     }.
30
31
32 %! parse_rows(+Y, -Pieces, -Rokades, -StartColor)
33 %
34 % Parse all rows in a flat list of board positions.
35 parse_rows(8, [Pieces | PiecesRest], [Rokades | RokadesRest], Passant, StartColor) --> %
36     ↪ Last row
37     % Parse the row
38     parse_border_row(8, black, Pieces, Rokades, Passant, StartColor),
39
40     % Recursive call
41     parse_rows(7, PiecesRest, RokadesRest, Passant, StartColor).
42
43 parse_rows(1, [Pieces], [Rokades], Passant, StartColor) --> % First row
44     % Parse the row
45     parse_border_row(1, white, Pieces, Rokades, Passant, StartColor).
46
47 parse_rows(Y, [Pieces | PiecesRest], Rokades, Passant, StartColor) --> % Rows in between
48     ↪ first & last row
49     {
50         YNext is Y - 1

```

```

49     },
50
51     % Parse the row
52     parse_row(Y, Pieces),
53
54     % Recursive call
55     parse_rows(YNext, PiecesRest, Rokades, Passant, StartColor).
56
57
58     %! parse_row(-Y, -Pieces)
59     %
60     % Parse a row of the chess board.
61     % Will not parse the first/last row
62     parse_row(Y, Pieces) -->
63         parse_row_number(Y),
64         parse_space,
65         parse_pieces(1/Y, Pieces),
66         parse_newline.
67
68
69     %! parse_border_row(+Y, +Color, -Pieces, -Rokades, -StartColor)
70     %
71     % Parse the first/last row of the chess board.
72     parse_border_row(Y, Color, Pieces, Rokades, Passant, StartColor) -->
73         parse_row_number(Y),
74         parse_space,
75         parse_pieces(1/Y, Pieces),
76         parse_space,
77         parse_metadata(Color, Rokades, Passant),
78         parse_current_player(Y, StartColor),
79         parse_newline.
80
81
82     %! parse_metadata(+Color, -Rokades, -Passant)
83     %
84     % Parse metadata (possible rokades & passant possibility)
85     parse_metadata(Color, Rokades, Passant) -->
86         "[",
87         parse_rokades(Color, Rokades),
88         parse_passant(Color, Passant),
89         "]".
90
91
92     %! parse_rokades(+Color, -Rokades)
93     %
94     % Parse a rocade notation.
95     %
96     % This is verbose on purpose to allow re-use of the parser for generating output.
97     parse_rokades(Color, [LongRokade, ShortRokade]) --> % Both rokades
98         parse_rocade_piece(Color, long, LongRokade),
99         parse_rocade_piece(Color, short, ShortRokade).
100
101     parse_rokades(Color, [LongRokade]) --> % Only first rocade

```

```

102     parse_rokade_piece(Color, long, LongRokade),
103     parse_space.
104
105 parse_rokades(Color, [ShortRokade]) --> % Only second rokade
106     parse_space,
107     parse_rokade_piece(Color, short, ShortRokade).
108
109 parse_rokades(_, []) --> % No rokades
110     parse_space,
111     parse_space.
112
113 %! parse_rokade_piece(+Color, +RokadeType, -Rokades)
114 %
115 % Parse a single piece of the rokade notation
116 parse_rokade_piece(Color, long, Rokade) --> % Large
117     parse_piece(Color, queen),
118
119     % Create the rokade
120     {
121         Rokade = rokade(Color, long)
122     }.
123
124 parse_rokade_piece(Color, short, Rokade) --> % Short
125     parse_piece(Color, king),
126
127     % Create the rokade
128     {
129         Rokade = rokade(Color, short)
130     }.
131
132
133 %! parse_passant(+Color, -Passant)
134 %
135 % Parse passant possibility
136 parse_passant(Color, passant(Color, X/Y)) --> % En-passant possible
137     parse_column_number(X),
138     parse_row_number(Y).
139 parse_passant(_, _) --> []. % En-passant not possible
140
141
142 %! parse_passant_position(-X/-Y)
143 %
144 % Parse passant possibility position
145 parse_passant_position(X/Y) --> % En-passant possible
146     parse_column_number(X),
147     parse_row_number(Y).
148
149 parse_passant_position(_) --> []. % En-passant not possible
150
151
152 %! parse_current_player(+Y, -StartColor)
153 %
154 % Parse a current player symbol or nothing

```



```

155 parse_current_player(8, black) --> "".
156 parse_current_player(1, white) --> "".
157 parse_current_player(_, _) --> "".
158
159
160 %! parse_final_row()
161 %
162 % Parse the final row of the board.
163 % This row does not contain any extra information.
164 parse_final_row -->
165     parse_space,
166     parse_space,
167     "abcdefgh",
168     parse_newline_or_nothing.
169
170
171 %! parse_row_number(+RowNumber)
172 %
173 % Parse a row number between 1 and 8.
174 parse_row_number(RowNumber) -->
175     {
176         % RowNumber must be a valid Y-value
177         % (this is here to allow for re-using the parser as output writer)
178         between(1, 8, RowNumber)
179     },
180     integer(RowNumber).
181
182
183 %! parse_column_number(+ColumnNumber)
184 %
185 % Parse a column number between 1 and 8.
186 parse_column_number(1) --> "a".
187 parse_column_number(2) --> "b".
188 parse_column_number(3) --> "c".
189 parse_column_number(4) --> "d".
190 parse_column_number(5) --> "e".
191 parse_column_number(6) --> "f".
192 parse_column_number(7) --> "g".
193 parse_column_number(8) --> "h".
194
195
196 %! parse_space()
197 %
198 % Parse a single space.
199 parse_space --> " ".
200
201
202 %! parse_newline()
203 %
204 % Parse a single newline.
205 parse_newline --> "\n".
206
207

```

```

208 %! parse_newline_or_nothing()
209 %
210 % Parse a single newline or nothing
211 parse_newline_or_nothing --> parse_newline.
212 parse_newline_or_nothing --> "".
213
214
215 %! parse_pieces(-Pieces)
216 %
217 % Parse a row of pieces.
218 % Will stop parsing when a newline is detected
219
220 % Piece: taken position
221 parse_pieces(X/Y, [Piece | Pieces]) -->
222     {
223         XNext is X + 1,
224
225         % Create the piece
226         Piece = piece(Color, Type, X/Y),
227
228         % X must be a valid X-value
229         % (this is here to allow for re-using the parser as output writer)
230         between(1, 8, X)
231     },
232
233     parse_piece(Color, Type),
234     parse_pieces(XNext/Y, Pieces).
235
236 % Space: empty position
237 parse_pieces(X/Y, Pieces) -->
238     {
239         XNext is X + 1,
240
241         % X must be a X-value
242         % (this is here to allow for re-using the parser as output writer)
243         between(1, 8, X)
244     },
245
246     parse_space,
247     parse_pieces(XNext/Y, Pieces).
248
249 parse_pieces(_, []) --> [].
250
251
252 %! parse_piece(-Color, -Type)
253 %
254 % Parse a single piece.
255 parse_piece(white, king) --> "\u2654". % White king
256 parse_piece(white, queen) --> "\u2655". % White queen
257 parse_piece(white, tower) --> "\u2656". % White tower
258 parse_piece(white, bishop) --> "\u2657". % White tower
259 parse_piece(white, horse) --> "\u2658". % White horse
260 parse_piece(white, pawn) --> "\u2659". % White pawn

```

```
261
262 parse_piece(black, king) --> "\u265A". % Black king
263 parse_piece(black, queen) --> "\u265B". % Black queen
264 parse_piece(black, tower) --> "\u265C". % Black tower
265 parse_piece(black, bishop) --> "\u265D". % Black tower
266 parse_piece(black, horse) --> "\u265E". % Black horse
267 parse_piece(black, pawn) --> "\u265F". % Black pawn
```

Listing 7: src/io/writer.pl

```

1  :- module(writer, []).
2
3  :- use_module("parser").
4  :- use_module("../state").
5  :- use_module("../position").
6  :- use_module("../move").
7  :- use_module("../piece").
8
9
10 %! write_state(+State)
11 %
12 % Write a chess game state to stdout.
13 write_state(State) :-
14     state:pieces(State, Pieces),
15     state:currentcolor(State, StartColor),
16     state:rokades(State, Rokades),
17     state:passant(State, Passant),
18
19     % Convert board & rokades in a format that could be used by the parser
20     extract_rows(8, Pieces, PiecesList),
21     extract_rokades(Rokades, RokadesList),
22
23     % Parse the rows output
24     parser:parse_rows(8, PiecesList, RokadesList, Passant, StartColor, OutRows, []),
25
26     % Parse the final row output
27     parser:parse_final_row(OutFinalRow, []),
28
29     % Write the output
30     write_codes(OutRows),
31     write_codes(OutFinalRow).
32
33
34 %! write_states(+States)
35 %
36 % Write a given list of states to stdout.
37 write_states([]).
38 write_states([State]) :-
39
40     % Write the board to stdout
41     write_state(State), !.
42 write_states([State | States]) :-
43
44     % Write the board to stdout
45     write_state(State),
46     write("\n~\n"),
47
48     % Recursive call
49     write_states(States), !.
50

```

```

51
52  %! write_draw()
53  %
54  % Write "DRAW" to stdout.
55  write_draw() :- write("DRAW").
56
57
58  %! write_states_or_draw(+CurrentState, -NextStates)
59  %
60  % Write all the states to stdout, or write "DRAW" in case of a stalemate
61
62  % Stalemate
63  write_states_or_draw(CurrentState, []) :-
64      state:currentcolor(CurrentState, CurrentColor),
65
66      % Check if the king is not check
67      not(state:check(CurrentState, CurrentColor)),
68
69      % Write "DRAW"
70      write_draw, !.
71
72  % Write all states
73  write_states_or_draw(_, NextStates) :-
74      write_states(NextStates), !.
75
76
77  %! write_codes(+Codes)
78  %
79  % Write a given list of codes to stdout.
80  write_codes(Codes) :-
81      atom_codes(String, Codes),
82      write(String).
83
84
85  %! extract_rows(+Y, +Pieces, -Rows)
86  %
87  % List of pieces for a given board, with each list representing the pieces for that row.
88  extract_rows(Y, Pieces, [Row | Rows]) :-
89
90      % Y must be valid
91      between(1, 8, Y), !,
92
93      % Extract the row
94      piece:row_pieces(Y, Pieces, UnsortedRow),
95
96      % Sort the row
97      piece:sorted_pieces(UnsortedRow, Row),
98
99      % Next row
100     YNext is Y - 1,
101
102     % Recursive call
103     extract_rows(YNext, Pieces, Rows), !.

```

```

104 extract_rows(_, _, []).
105
106
107 %! extract_rokades(+Rokades, +RokadesList)
108 extract_rokades(Rokades, [BlackRokades, WhiteRokades]) :-
109     extract_rokades_color(black, Rokades, BlackRokades),
110     extract_rokades_color(white, Rokades, WhiteRokades).
111
112
113 %! extract_rokades_color(+Color, +Rokades, -ColorRokades)
114 %
115 % Extract all rokades for a given color
116 extract_rokades_color(Color, [Rokade | Rokades], [ColorRokade | ColorRokades]) :- % Match
117     Rokade = rokade(RokadeColor, _),
118
119     % Color must match
120     Color == RokadeColor,
121
122     % Append to the list
123     ColorRokade = Rokade, !,
124
125     % Recursive call
126     extract_rokades_color(Color, Rokades, ColorRokades), !.
127 extract_rokades_color(Color, [_ | Rokades], ColorRokades) :- % No match
128     % Recursive call
129     extract_rokades_color(Color, Rokades, ColorRokades), !.
130 extract_rokades_color(_, [], []).

```

Listing 8: src/main.pl

```

1 :- initialization(main, main).
2
3 :- use_module("io/parser").
4 :- use_module("io/writer").
5 :- use_module("move").
6 :- use_module("alphabeta").
7 :- use_module("state").
8
9
10 main :-
11     current_prolog_flag(argv, Args),
12     handle_main(Args),
13     halt(0).
14
15 handle_main([_]) :- % Test Mode
16
17     % Load the data from the stdin stream and parse it.
18     phrase_from_stream(parser:parse_state(State), current_input),
19
20     % Get all possible states for the current state
21     state:all_possible_states(State, NextStates),
22
23     % Print all possible states
24     writer:write_states_or_draw(State, NextStates).
25
26 handle_main([]) :- % Move Mode
27
28     % Load the data from the stdin stream and parse it.
29     phrase_from_stream(parser:parse_state(State), current_input),
30
31     % Extract the player from the state
32     state:currentcolor(State, Player),
33
34     % Determin the next best move
35     alphabeta:alphabeta(Player, State, 0, 4, -100000, 100000, BestState, _),
36
37     % Write the next state to stdout, or write "DRAW" in case of a stalemate.
38     % There is a stalemate when the best state is equal to none
39     (
40         % Print "DRAW"
41         BestState == none, writer:write_draw()
42         ;
43         % Print the best state
44         writer:write_state(BestState)
45     ).

```

Listing 9: src/util/utils.pl

```

1 :- module(utils, []).

```

```

2
3
4  %! between1(+Start, ?Value)
5  %
6  % Value is between Start - 1 and Start + 1
7  :- table between1/2.
8  between1(Start, Value) :-
9      Minus is Start - 1,
10     Plus  is Start + 1,
11     between(Minus, Plus, Value).
12
13
14  %! between2(+Start, ?Value)
15  %
16  % Value is between Start - 2 and Start + 2
17  :- table between2/2.
18  between2(Start, Value) :-
19      Minus is Start - 2,
20      Plus  is Start + 2,
21      between(Minus, Plus, Value).
22
23  %! list_equals(+List1, +List2)
24  %
25  % If 2 lists match, ignoring the order
26  list_equals([], []).
27  list_equals([H1 | T1], List2) :-
28      member(H1, List2),
29      delete(List2, H1, List3),
30      list_equals(T1, List3).

```
