U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Echek

Relatório Final

Trabalho Prático 1

Programação em Lógica

(2019/2020)

Turma **5** - Grupo **Echek_1**:

Carlos Jorge Direito Albuquerque - **up201706735**@fe.up.pt

Tito Alexandre Trindade Griné - **up201706732**@fe.up.pt

Índice

1. Introdução

O objetivo deste primeiro trabalho prático era modelar um jogo de tabuleiro na linguagem de programação Prolog, por forma a introduzir o conceito de Programação em Lógica. Para além de recriar o jogo em modo de texto, foi-nos também pedido que criássemos uma inteligência artificial para ser possível um utilizador jogar contra o computador e até pôr o computador a jogar contra si próprio. Todos os objetivos foram cumpridos e é a seguir explicada toda a nossa abordagem.

2. O Jogo Echek

2.1. História

Echek é um jogo criado por Léandre Proust, um autor de jogos de tabuleiro *Free-to-Play* natural de França. O criador conta já com 9 jogos originais, sendo Echek o primeiro e também o mais velho - foi criado em Fevereiro de 2019.

Segundo Léandre, "a ideia [do Echek] é reproduzir a experiência de um jogo de xadrez apenas com 12 cartaz e sem tabuleiro".

2.2. Regras

Início: Cada jogador fica com as 6 peças de uma cor. Os Reis são colocados frente a frente na zona de jogo, ficando as restantes peças na reserva de cada jogador. Começa a jogar quem tiver as peças brancas.

Figura 1 - Disposição Original do Jogo

Objetivo: Movimentar as peças de modo a que o Rei do adversário fique rodeado, isto é, ter uma peça, do jogador ou do adversário, ou um limite nas quatro casas (Norte, Sul, Este e Oeste) adjacentes ao Rei.

Desenvolvimento: Os jogadores jogam à vez, podendo em cada jogada realizar uma de duas possíveis ações:

- Jogar uma peça: O jogador escolhe uma peça da sua reserva e coloca-a em jogo. A peça tem de ser posicionada por forma a ficar ligada por um canto ou por um lado a uma outra peça do jogador (mesma cor); não é possível jogar uma peça diretamente numa das 4 casas adjacentes ao Rei Inimigo.
- Mover uma peça: O jogador escolhe uma das suas peças em jogo e movimenta-a respeitando as regras do movimento de cada peça; após a jogada, todas as peças em jogo devem estar conectadas umas às outras por um lado ou por um canto, caso contrário a jogada não é válida.

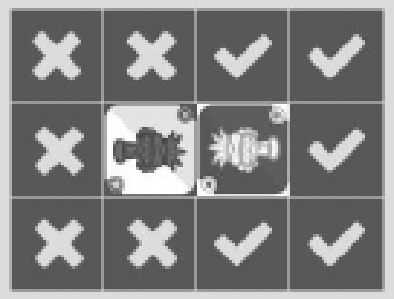

Figura 1 - Possíveis casas de colocação para a primeira peça do jogador branco

Não existe captura de peças, isto é, uma peça não pode ser movimentada para a casa de uma outra peça.

O espaço de jogo está limitado a 4 casas na horizontal e 4 na vertical, as peças não podem ser movimentadas nem jogados para lá desse limite. Os limites só existem quando as peças em jogo assim o forçam, ou seja, quando quatro casas numa direção contêm peças.

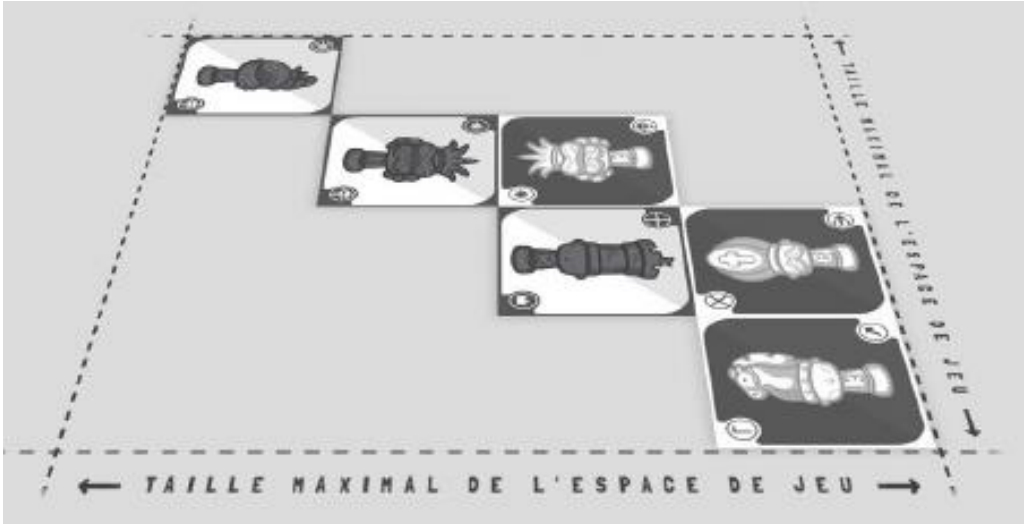

Figura 2 - Exemplo da criação de limites do tabuleiro

Mecânica das Peças: As peças usadas são as mesmas do xadrez, contudo, cada jogador apenas tem uma peça de cada tipo. Existem, portanto, as seguintes peças:

- Rei: Movimenta-se uma casa em linha reta ou na diagonal. Se for rodeado, o jogador perde o jogo.
- Rainha: Pode movimentar-se um número ilimitado de casas em linha reta ou na diagonal. Se for rodeada, é retirada do tabuleiro e não pode ser colocada em jogo outra vez.
- Bispo: Pode movimentar-se um número ilimitado de casa na diagonal. Quando é colocada em jogo, o jogador escolhe uma peça do tabuleiro (excluindo os reis) para ser colocada na reserva do jogador cuja peça pertence.
- Torre: Pode movimentar-se um número ilimitado de casas em linha reta. Apenas uma vez por jogo, se a peça estiver no tabuleiro, o jogador pode optar por não mover nenhuma peça, e trocar a Torre e o Rei de posição.
- Cavalo: Movimenta-se em *L*, ou seja, uma casa em linha reta e outra na diagonal. Pode passar por cima tanto de peças aliadas como inimigas.
- Peão: Movimenta-se uma casa em linha reta. Quando colocada em jogo, pode movimentar-se ainda na mesma jogada.

As peças que se movimentam um número ilimitado de casas (rainha, torre e bispo) não podem passar por casas contendo peças da cor oposta, mas é possível se as peças forem da mesma cor

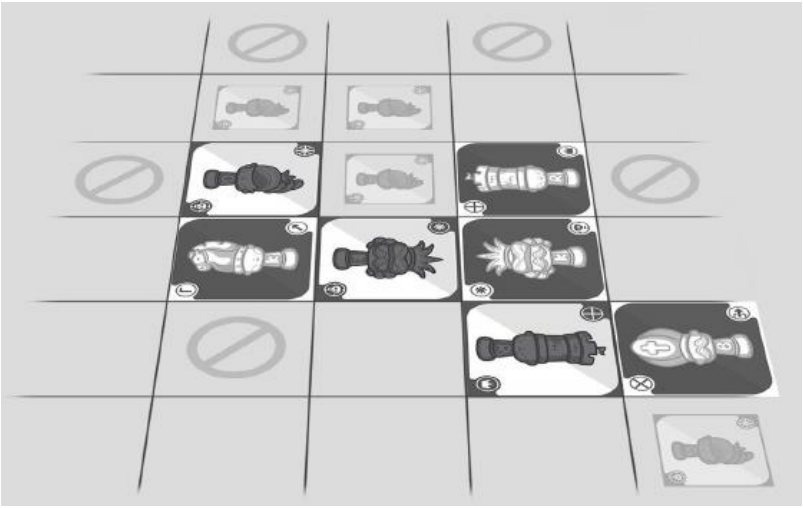

Figura 3 - Exemplo dos possíveis movimentos da rainha preta

Final: O jogo termina quando um dos jogadores cumpre o objetivo, isto é, rodeia o Rei do adversário. Não existe nenhuma outra condição de terminação do jogo, ou seja, até um dos Reis ser rodeado, é sempre possível realizar uma jogada.

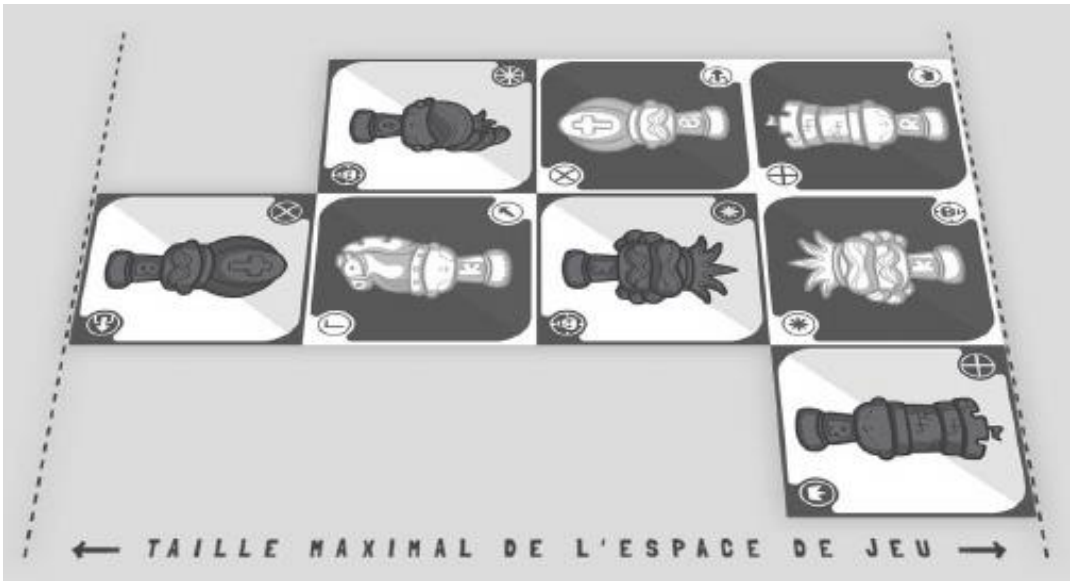

Figura 4 - Exemplo de uma derrota do jogador branco

Fontes:

https://en.tipeee.com/leandreproust (Site Oficial do Autor, consultado a 14/10)

http://www.fabrikludik.fr/jeuxGratuit/echekRegle.pdf (Livro de Regras do Jogo, consultado a 14/10)

3. Lógica do Jogo

3.1. Representação do Estado do Jogo

Uma vez que o jogo difere um bocado dos jogos tradicionais de tabuleiro, visto que não tem um tabuleiro estático, mas sim um relativo, isto é, está dependente da posição das peças em jogo para definir os possíveis limites, a abordagem normal de mapear o tabuleiro numa lista de listas pareceu-nos inadequada ou pelo menos, pouco prática. O facto de ser um tabuleiro dinâmico não é trivial, e considerámos que forçar o tabuleiro a ser estático tiraria uma mecânica do jogo que acrescenta várias possibilidades para estratégias interessantes, o que torna, em último caso, o jogo mais divertido.

Assim, primeiro decidimos representar internamente o jogo através de um conjunto de predicados que ditam as peças e respetivas posições que se encontram em jogo. O jogador vê apenas uma grelha 6x6, que o auxilia a indicar movimentos. Na realidade, internamente, o jogo desenrola até uma das peças sair do quadrado 4x4 no interior do tabuleiro, situação em que as peças são ajustadas por forma a recolocá-las de novo no quadrado central (fazendo *shift* de todas as peças uma casa na direção necessária). Caso a disposição do jogo obrigue a limites, dado o reajuste que é feito, o limite estará sempre no quadrado 4x4 interior. Isto assegura também que nunca há um movimento válido fora do quadrado 6x6, já que as peças têm de estar sempre ligadas umas às outras.

```
% Pieces internal representation.
% These facts associate color and name to the corresponding character.
piece(king, black, '♚').
piece(king, white, '♔').
piece(queen, black, '♛').
piece(queen, white, '♕').
piece(bishop, black, '♝').
piece(bishop, white, '♗').
piece(rook, black, '♜').
piece(rook, white, '♖').
piece(knight, black, '♞').
piece(knight, white, '♘').
piece(pawn, black, '♟').
piece(pawn, white, '♙').
```

Figura 6 - Representação Interna das Peças

```
cell(3, 2, white, king).
cell(2, 2, black, king).
```

Figura 7 - Representação do Tabuleiro Inicial
(as casas em branco não têm predicado)

```
cell(1, 2, white, king).
cell(3, 3, black, king).
cell(3, 4, white, pawn).
cell(3, 2, black, pawn).
cell(2, 1, white, knight).
cell(4, 2, black, knight).
cell(2, 3, white, queen).
cell(4, 3, white, bishop).
cell(3, 1, black, bishop).
cell(1, 3, white, rook).
cell(4, 4, black, rook).
```

Figura 8 - Representação do tabuleiro a meio de um jogo

```
cell(1, 2, white, king).
cell(3, 3, black, king).
cell(3, 4, white, pawn).
cell(2, 2, black, pawn).
cell(2, 1, white, knight).
cell(4, 2, black, knight).
cell(2, 3, white, queen).
cell(1, 3, white, queen).
cell(3, 2, white, bishop).
cell(3, 1, black, bishop).
cell(1, 4, white, rook).
cell(4, 4, black, rook).
```

Figura 9 - Representação do tabuleiro no final do jogo
(Rei preto rodeado)

```
% Predicate called every turn to display the current board state.
display_board :-
    nl, nl, draw_space, draw_title, nl,
    draw_space, draw_top_boundary,
    draw_space, draw_column_coordinates,
    \+display_board([6, 6], 0),             % Actual board display with pieces is made here.
    draw_space, draw_bottom_boundary, nl.

% Displays the whole board (by rows).
% Calls itself recursively until it reaches the X row (usually 0).
display_board([R, C], X) :-
    X < R,
    draw_space, draw_separator(C),
    draw_space, draw_line_coordinate(X), % Drawing the line number so the user can more easily play a piece.
    \+display_row([X, C], 0), nl,
    N is X + 1, !, display_board([R, C], N).

% Displays a row. Only displays the actual content (pieces) of a row.
% Calls itself recursively until it reaches the X column (usually 0).
display_row([R, C], X) :-
    X < C, display_piece(R, X),
    N is X + 1, !, display_row([R, C], N).

% Displays a piece. The piece's character is drawn if the owner's color over a background of the opposite color.
display_piece(R, C) :-
    cell(C, R, Color, Piece),
    piece(Piece, Color, Char),
    opposite(Color, Opposite),
    draw_piece(Char, Color, Opposite).

% If there is no piece on a given possition, then this predicate draws a blank space.
display_piece(R, C) :-
    \+cell(C, R, _, _),
    draw_blank.
```

Figura 10 - Funções principais responsáveis pelo display do tabuleiro.

Nota: Outras funções também são utilizadas para o display do tabuleiro, apenas para manter a aparência consistente.

3.2. Visualização do Tabuleiro

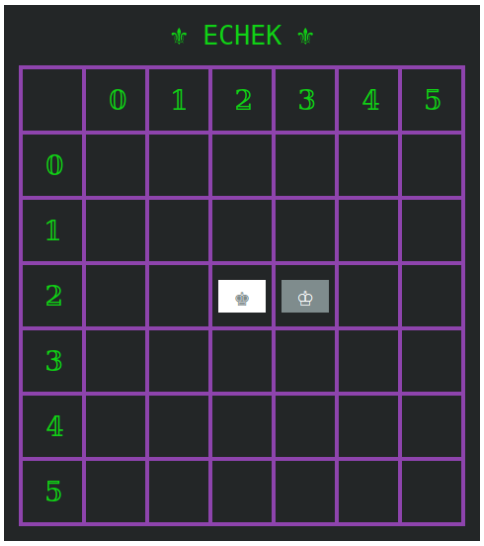

Figura 11 - Configuração Inicial

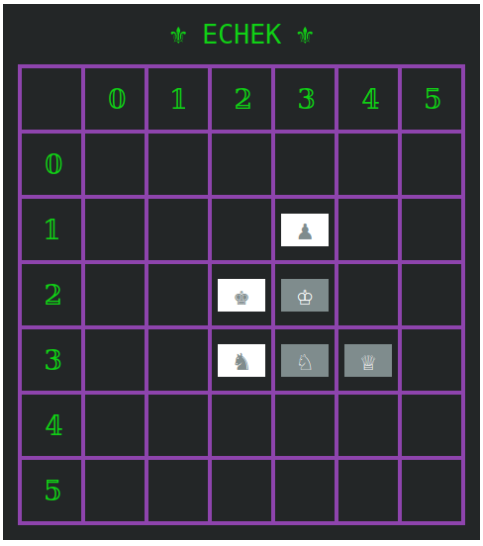

Figura 12 - Configuração Intermédia 1

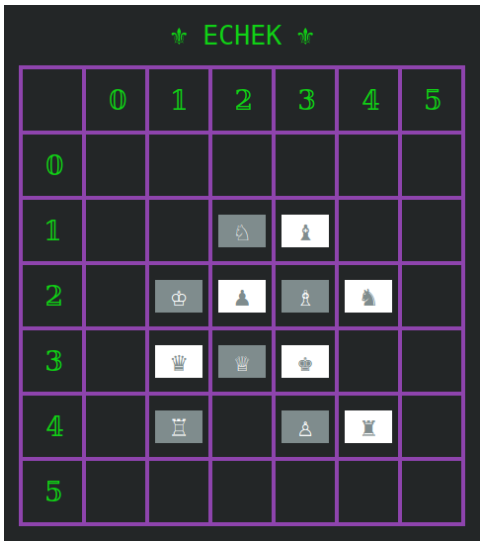

Figura 13 - Configuração Intermédia 2

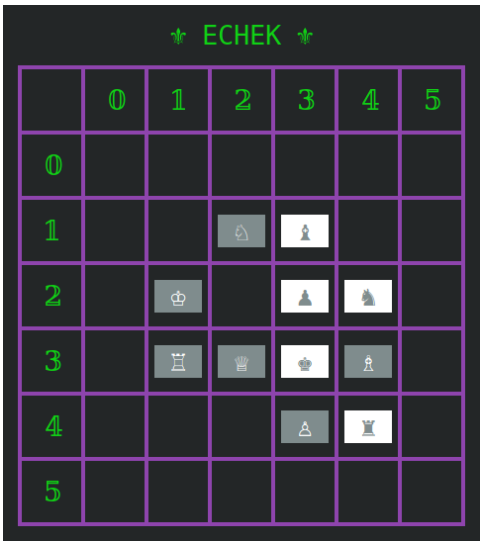

Figura 14 - Configuração Final
(Rei Preto Rodeado)

Nota: De modo a ser possível visualizar as cores e os caracteres de Unicode utilizados, recorreu-se a SWI em vez de o Sictus como sistema de desenvolvimento para Prolog.

3.3. Lista de Jogadas Válidas

A implementação do jogo assume que o jogador conhece as regras, mesmo assim no menu principal é possível selecionar a opção de instruções que indica como interagir com o programa durante o jogo, mais concretamente como realizar as habilidades especiais de algumas peças. Contudo, este menu não contém uma descrição detalhada de todas as regras e possíveis estratégias do jogo. Assim sendo, o predicado ***valid_moves*** não é utilizado numa jogada de um utilizador, apenas é utilizado para uma jogada de um AI (*Artificial Inteligence*).

Portanto, quando é a vez de o AI realizar uma jogada é chamado, primeiramente, o predicado ***valid_moves***:

```
% Finds all the possible moves a player can make (placement and movement), creating a list of type [[Piece0, X0, Y0], [Piece1, X1, Y1], ..., [PieceN, XN, YN]].
valid_moves(Player, PossibleMoves) :-
    findall([Piece, X, Y],(piece(Piece, Player, _), between(0, 5, X),between(0, 5, Y), possible_play(Player, Piece, X, Y)), PossibleMoves).
```

Figura 15 - Predicado valid_moves

O predicado ***valid_moves*** utiliza o predicado *findall* que vai percorrer todas as peças do jogador (*piece(-Piece,+ Player, _)*), iterando sobre todas os pares de coordenadas (x, y) entre 0 e 5 do tabuleiro (*between(0, 5, -X/-Y)*), e manter apenas aquelas que são validadas pelo predicado *possible_play*, que verifica se uma dada peça do jogador pode na próxima jogada estar na célula (x, y), quer através de um *placement* (colocar a peça em jogo), quer através de um *move* (mover a peça). Todas as combinações aceites são guardadas numa lista *PossibleMoves* no qual cada termo terá o formato *[Piece, X, Y]*.

```
% Checks if the Piece belonging to the Player can either be placed or moved to position (X, Y).
possible_play(Player, Piece, X, Y) :-
    check_placement(Player, Piece, X, Y);
    check_movement(Player, Piece, X, Y).
```

Figura 16 - Predicado possible_play

Dado que cada peça tem a sua habilidade especial, para o nosso jogo foi necessário implementar mais dois predicados que obtivessem jogadas validas para estas ocasiões.

Uma dessas situações é a possibilidade do movimento imediato do peão, após a sua colocação no tabuleiro. Para isto foi apenas necessário criar um predicado *valid_special_play* para o peão, que essencialmente funciona de forma análoga ao predicado *valid_moves*, mas em vez de iterar sobre todas as peças do jogador, apenas verifica os possíveis movimentos do peão.

```
% Finds all possible moves for the special ability of the pawn belonging ot the Player, when placed.
% Creates a list of type [[pawn, X0, X0], ..., [pawn, XN, YN]].
valid_special_play(pawn, Player, PossibleMoves) :-
    cell(PX, PY, Player, pawn), append([[PX, PY]], [], Move1),
    valid_piece_moves(Player, pawn, Moves2),
    append(Move1, Moves2, PossibleMoves).
```

Figura 17 - Predicado valid_special_play do peão

A segunda situação diz respeito à colocação do bispo em jogo, em que o jogador é forçado a escolher uma peça que esteja no tabuleiro para ser removida, exceto no caso em que apenas estão os reis no tabuleiro, que não podem ser removidos. Tal como no peão, existe o predicado *valid_special_play* que irá obter todas as posições das peças do tabuleiro que podem ser removidas, recorrendo por sua vez ao predicado *possible_removal*, que avalia se uma peça é possível de remover do tabuleiro.

```
% Finds all possible pieces that the bishop of Player can remove with its special ability, when placed.
% Creates a list of type [[X0, X0], ..., [XN, YN]].
valid_special_play(bishop, Player, PossibleMoves) :-
    findall([X, Y],(cell(X, Y, TargetPlayer, Piece), possible_removable(Player, TargetPlayer, Piece, X, Y)), PossibleMoves).
```

Figura 18 - Predicado valid_special_play para o bispo

Com o auxílio destes predicados é possível então obter uma lista com todas as jogadas possíveis, incluindo casos especiais, para cada momento do jogo.

3.4. Execução de Jogadas

No caso de se tratar de um jogador, o turno é interpretado no predicado *player_turn(+Player)* que pede ao utilizar que escolha uma das peças do jogo e que de seguida tome uma ação. Esta ação é tratada pelo predicado *player_action(+Player, +Piece)* e tanto pode ser uma movimentação ou uma colocação de peça. No caso de a peça ter uma habilidade especial aquando da sua colocação (como no caso do peão e do bispo), ela é executada diretamente após a colocação, ainda na mesma instância do predicado *player_action*. Para essas habilidades serem executadas, existe o predicado *special_ability_on_placement(+Piece, +Player)*, que, de acordo com a peça em causa, efetua a jogada pedindo ao utilizador as coordenadas e verificando se conduzem a uma jogada válida.

```
% Allows the player to choose a piece, redisplays the board and makes the plater take an action.
% If the player cancels the action then the prompt to select the piece is shown once more.
player_turn(Player) :-
    repeat,
    (choose_piece(Player, Piece),
     display_board,
     player_action(Player, Piece)
    ),!,
    readjust_board,             % If the pieces are in the outer ring of the board, it reajusts it by shifting appropriately
    check_queens_death.         % Checks if any of the queens are surrounded (if so they will be removed from the game)

% Depending on the piece selected (if it's on the board or not), the action it takes can either be a move or a placement.
% It always returns true.
player_action(Player, Piece) :-
    move(Player, Piece);
    (place(Player, Piece) , special_ability_on_placement(Piece, Player)). % If its a placement and the piece has a special ability, it will execute it.
```

Figura 19 - Predicados player_turn e player_action

Quer o jogador escolha uma peça para colocar ou para jogar, o funcionamento do programa é idêntico - as coordenadas são lidas do standard input (se elas forem *"-1, -1"* a jogada é cancelada e o jogador pode escolher a peça novamente) e é feita a verificação da validade da jogada. Para uma colocação em jogo, o predicado responsável é *place(+Player, +Piece)* que verifica se a peça escolhida pode ser colocada ou não em jogo. É de notar que caso a peça seja uma rainha a colocação é permitida só se a rainha do jogador ainda não tiver sido morta.

```
% Reads the user input coordinates until it finds a valid position to place the queen, returning true,
% if the queen is dead or the user cancels the piece choice, it returns false.
place(Player, queen) :-
    \+ cell(_, _, Player, queen),!,
    dead(Player, queen, false),         % Check if the players queen has been killed before, in which case the queen can't be replaced on the board.
    repeat,
    read_coords(X, Y, Player, queen, place),
    (cancel(X, Y);
     (valid_cell(Y, X, Player),
      add_to_database(X, Y, Player, queen))),!,
    \+ cancel(X, Y).

% Reads the user input coordinates until it finds a valid position to place the Piece, returnin true,
% or if the user cancels the piece choice, returning false.
place(Player, Piece) :-
    \+ cell(_, _, Player, Piece),
    repeat,
    read_coords(X, Y, Player, Piece, place),
    (cancel(X, Y);
     (valid_cell(Y, X, Player),
      add_to_database(X, Y, Player, Piece))),!,
    \+ cancel(X, Y).
```

Figura 20 - Predicados place

Para uma movimentação é usado o predicado *move(+Player, +Piece)* que por sua vez analisa se a peça pode ou não ser movida.

```
% This predicate is responsible for moving a piece.
% It askes the user where to move the piece to and then checks if it is a valid move and does not break any rule.
move(Player, Piece) :-
    cell(Xinit, Yinit, Player, Piece),
    !,
    repeat,
    read_coords(Xdest, Ydest, Player, Piece, move),
    (cancel(Xdest, Ydest);
    (castling_move(Player, Piece, Xdest, Ydest, Xinit, Yinit));
    (valid_cell(Ydest, Xdest),
    valid_move(Piece, Xinit, Yinit, Xdest, Ydest, Player),
    change_database(Xdest, Ydest, Player, Piece),
    ((check_virtual_limits,
      check_connections); (change_database(Xinit, Yinit, Player, Piece), false)))),!,
    \+ cancel(Xdest, Ydest).
```

Figura 21 - Predicado move

No caso de se tratar de um AI, o turno passa a ser controlado pelo predicado *ai_turn(+Player, +Level)*, que, por sua vez, procede à seleção de jogadas válidas (*valid_moves*), à escolha da jogada adequada de acordo com o nível passad0 como argumento (ver seccção 3.7) e por fim efetua a ação/jogada escolhida.

```
% It calculates the valid moves for the AI, and, depending on the Level choosen, decides what
% move to make and takes action.
ai_turn(Player, Level) :-
    valid_moves(Player, Moves),
    choose_move(Level, Player, Moves, Move),
    ai_action(Player, Level, Move),
    readjust_board,             % If the pieces are in the outer ring of the board, it reajusts it by shifting appropriately.
    check_queens_death.         % Checks if any of the queens are surrounded (if so they will be removed from the game).
```

Figura 22 - Predicado ai_turn

Para tal, invoca o predicado *ai_action(+Player, +Level, +Move)*, que faz essencialmente o mesmo que *player_action*, com algumas diferenças.

```
% Depending on the piece selected (if it's on the board or not), the action it takes can either be a move or a placement.
% It always returns true.
ai_action(Player, Level, [Piece, X, Y]) :-
    move_ai(Player, Piece, X, Y);
    (place_ai(Player, Piece, X, Y),
    ((valid_special_play(Piece, Player, Moves),     % If its a placement and the piece has a special ability, it will get a list of possible moves.
      choose_special_move(Level, Player, Piece, Moves, Move),!,      % Chooses a move from the list, criteria depends on the Level of the AI.
      special_ability_on_placement_ai(Piece, Player, Move)); true)). % Executes the special move.
```

Figura 23 - Predicado ai_action

Estas encontram-se maioritariamente no facto de toda a jogada já ter sido escolhida e de ter sido verificada como jogada válida. Por essa razão, os predicados *place_ai(+Player, +Piece, +X, +Y)* e *move_ai(+Player, +Piece, +X, +Y)* fazem o mesmo que as correspondentes versões para o utilizador, mas sem pedirem input e sem verificarem a validade da jogada recebida.

```
% Checks if the piece is already on the board, in which case it return false, if not it adds it.
% Since the placement is done by the AI that already analysed the move as being valid, no further validation is done.
place_ai(Player, Piece, X, Y) :-
    \+ cell(_, _, Player, Piece),
    add_to_database(X, Y, Player, Piece).
```

Figura 24 - Predicado place_ai

```
% This predicate moves a piece for the AI.
% This one does not need to ask for input or to check if it is a valid move because the AI only generates valid moves.
move_ai(Player, Piece, Xdest, Ydest) :-
    cell(Xinit, Yinit, Player, Piece),!,
    (castling_move(Player, Piece, Xdest, Ydest, Xinit, Yinit);
     change_database(Xdest, Ydest, Player, Piece)).
```

Figura 25 - Predicado move_ai

Todavia, para que um AI consiga executar as habilidades especiais do peão e do bispo, o predicado *ai_action* tem de ser ligeiramente mais complexo que o *player_action*. Isto deve-se ao facto de as possíveis jogadas especiais terem também de ser geradas (*valid_special_plays*), escolhidas (ver secção 3.7) e, por fim, executadas recorrendo ao predicado *special_ability_on_placement_ai(+Piece, +Player, +Move)*.

3.5. Final do Jogo

No final de cada jogada, quer se trata de uma jogada de um jogador ou de uma das AI, é utilizado o predicado ***game_over*** que analisa o estado do tabuleiro e verifica se houve um empate ou uma vitória, caso em que devolve verdadeiro, se nenhum tiver acontecido retorna falso e é dada continuação ao jogo, com a jogada do próximo jogador.

Para que o jogo termine, pelo menos um dos reis tem de estar rodeado por quatro peças nas células imediatamente adjacentes em cada direção (Norte, Este, Sul, Oeste). Assim, a regra começa por obter as coordenadas de ambos os reis (*cell(-WKC/-BKC, -WKR/-BKR , +white/+black, +king)*) e utiliza o predicado *check_surrounded* para saber se algum dos reis foi rodeado e, portanto, morto. De seguida, o predicado verifica se ambos estão rodeados, situação em que ocorre empate, e caso não seja, verifica cada um dos reis. Se o jogo terminar uma mensagem é mostrada indicando quem ganhou, ou que houve empate.

```
% Checks if any of the kings is surrounded. If both are, displays a tie message and returns true.
% If only one of them is, then the opposite color player has won, a dedicated message is displayed and
% return true. If none are surrounded it returns false.
game_over(Winner) :-
    cell(WKC, WKR, white, king),
    cell(BKC, BKR, black, king),
    !,
    ((check_surrounded(WKR, WKC), check_surrounded(BKR, BKC),
     ansi_format([fg(yellow)], 'It\'s a draw!!', []), Winner = black_white), nl;
    (check_surrounded(WKR, WKC),
     ansi_format([fg(yellow)], 'The black player has won!!', []), Winner = black), nl;
    (check_surrounded(BKR, BKC),
     ansi_format([fg(yellow)], 'The white player has won!!', []), Winner = white), nl).
```

Figura 26 - Predicado game_over

3.6. Avaliação do Tabuleiro

A avaliação do tabuleiro tem como finalidade o AI poder escolher a(s) melhor(es) jogada(s), de modo a que possa ter uma estratégia mais inteligente. Como o nosso jogo não tem um sistema de pontuação fixo, tivemos de criar uma heurística de valorização, de raiz.

Existem dois predicados de avaliação: *value(+Player, +Piece, +X, +Y, -Value)*, que avalia as jogadas normais; *value_special(+Player, +Piece, +X, +Y, -Value)*, que avalia as jogadas especiais do peão e do bispo. Na verdade, a avaliação da habilidade especial do peão é feita da mesma forma que as restantes peças. O que difere no peão é a avaliação da sua colocação, que, para ser o mais inteligente possível, tem de ter em conta a movimentação instantânea do peão.

```
% Evaluates the value of a pawn move, that can be slightly different from the other pieces given the fact that it can move immediatly after it's placed.
value(Player, pawn, X, Y, Value) :-
    \+ cell(_, _, Player, pawn),
    value_next_move(Player, V2, pawn, X, Y),
    value_defensive(Player, V3, pawn, X, Y),
    ((V3 =:= -100, Value is V3);
     (V2 >= 1, V3 >= 6, Value is 120); %In this case, it is very likely that the ai can win the game with a special pawn move
      Value is (0 + (0.1 * V2) + V3)).

% Evaluates the value of the given Piece play by the Player, considering 3 factors:
% - Its immediate offensive value;
% - Its usefulness in the next move;
% - It's immediatte defensive value.
value(Player, Piece, X, Y, Value) :-
    value_offensive(Player, V1, Piece, X, Y),
    value_next_move(Player, V2, Piece, X, Y),
    value_defensive(Player, V3, Piece, X, Y),
    ((V3 =:= -100, Value is V3);
      Value is (0 + (1.2 * V1) + (0.05 * V2) + V3)).
```

Figura 27 - Predicados value

A avaliação de uma jogada pelo nosso programa está dividida em três partes, na prática, três predicados:

- *value_offensive(+Player, -Value, +Piece, +X, +Y)* - atribui mais pontos quando uma jogada tenta cercar o rei ou a rainha inimigos; caso a jogada rodeie o rei inimigo então a sua valorização é propositadamente exagerada para que o AI acabe com o jogo sempre que tenha oportunidade.
- *value_defensive(+Player, -Value, +Piece, +X, +Y)* - valoriza negativamente, isto é, desincentiva o AI a realizar jogadas que cerquem o próprio rei ou rainha. Também aqui uma jogada que cerque o próprio rei é exageradamente desvalorizada por forma a evitar que o AI se suicide.
- *value_next_move(+Player, -Value, +Piece, +X, +Y)* - valoriza as possíveis próximas jogadas de uma peça de forma a dar mais valor às peças mais versáteis que consigam

atacar o rei ou a rainha adversários na próxima jogada. Isto não tem em conta, contudo, as possíveis jogadas que o adversário pode tomar.

Usando estes três níveis de avaliação de uma jogada, o AI consegue simular uma estratégia ofensiva, sempre que esta seja possível, e defensiva, quando começa a ser demasiado cercada. Como foi referido anteriormente, a colocação do peão é avaliada de forma diferente. O valor ofensivo é descartado, porque independentemente de onde a peça vai ser colocada ela vai poder mover-se logo de seguida, e é feita uma estimativa para avaliar se o AI consegue ganhar o jogo ao mover logo o peão para cercar o rei inimigo, sendo a jogada bastante valorizada nesse caso.

Quanto à habilidade especial do bispo, o funcionamento é idêntico. Ou seja, existe uma valorização ofensiva, que consiste em dar valor negativo à remoção de peças que cerquem o rei ou a rainha adversários (*value_offensive_remove*), uma valorização defensiva, que valoriza a remoção de peças à volta do próprio rei e rainha (*value_defensive_remove*), e, em vez de dar valor a uma próxima jogada, atribui um valor a cada peça. As peças aliadas têm valor negativo, as peças adversárias valor positivo e o valor absoluto de cada peça obedece à ordem rainha > torre > cavalo > peão > bispo (*value_piece_remove*). Para além disto, os pontos ofensivos e defensivos são proporcionais ao número de peças a rodear o rei adversário e aliado, respetivamente, para dar mais ênfase a essas jogadas em situações mais críticas.

```
% Evaluates the value of a pawns' special ability (it can move immediatly after it's placed).
value_special(Player, pawn, X, Y, Value) :-
    value(Player, pawn, X, Y, Value).

% Evaluates the value of a bishops' special ability (it can remove a piece from the board when placed).
value_special(Player, bishop, X, Y, Value) :-
    opposite(Player, Enemy),
    cell(X, Y, PieceColor, Piece),
    value_offensive_remove(Enemy, V1, X, Y),  % If it is the players king/queen then the removal is strongly encouraged.
    value_defensive_remove(Player, V2, X, Y), % Otherwise it is strongly disencouraged.
    value_piece_remove(Player, PieceColor, Piece, V3),
    count_attackers(Player, king, NFriendlyKing),
    count_attackers(Enemy, king, NEnemyKing),
    Value is 0 + NFriendlyKing * V1 + NEnemyKing * V2 + V3. % The higher the number of pieces around a king, the more important is the removal.
```

Figura 28 - Predicados value_special

Com todas estas cláusulas tentámos recriar o que seria uma estratégia equilibrada com uma conotação ofensiva. Foi preciso adicionar também a possibilidade de o AI optar por uma jogada não ótima, quando as melhores jogadas incluíam apenas movimentos do rei, uma vez que nestas situações, quando a jogar contra si próprio, o AI entrava num ciclo infinito, em se limitava a mover os reis no tabuleiro.

3.7. Jogada do Computador

Para este trabalho implementámos dois tipos de AI, o primeiro, que designámos de "random" AI, que tal como o nome indica, uma vez obtida a lista de jogadas possíveis escolhe uma delas aleatoriamente. O segundo AI, que designámos de "smart" AI (embora isto seja debatível), utiliza o predicado *value*, atribuindo um valor a cada jogada possível, e selecionando aleatoriamente uma jogada das com o maior valor.

O predicado ***choose_move*** recebe, em ambos os casos, o nível de dificuldade, que pode ser o átomo random ou smart, o *Player*, ou seja, a cor das peças do jogador, uma lista das jogadas possíveis, *Moves*, e irá colocar na última variável a jogada a fazer pelo AI.

```
% From a list of possible moves, chooses a random one. Used by random AI.
choose_move(random, _, Moves, Move) :-
    random_member(Move, Moves).

% From a list of possible moves, calculates the value (integer) of each one, creates a list with only the moves with
% the highest value (best play), and randomly picks one. Used by "smart" AI.
choose_move(smart, Player, Moves, BestMove) :-
    calculate_moves_value(Player, Moves, ValuedMoves),
    max_value_list(ValuedMoves, BestValue),
    make_best_moves_list(ValuedMoves, BestValue, BestMoves),
    random_member(BestMove, BestMoves), !.

% From a list of possible special moves, chooses a random one. Used by random AI.
choose_special_move(random, _, _, Moves, Move) :-
    random_member(Move, Moves).

% From a list of possible special moves, calculates the value (integer) of each one, it then creates a list with only
% the highest value moves (best ones), and randomly chooses one. Used by "smart" AI.
choose_special_move(smart, Player, Piece, Moves, BestMove) :-
    calculate_special_moves_value(Player, Piece, Moves, ValuedMoves),
    max_value_list(ValuedMoves, BestValue),
    make_best_special_moves_list(ValuedMoves, BestValue, BestMoves),
    random_member(BestMove, BestMoves), !.
```

Figura 29 - Predicados choose_move

Como já foi referido, tanto no *choose_move* como no *choose_special_move*, se o nível de dificuldade for random, a jogada seguinte é escolhida aleatoriamente das jogadas possíveis existentes.

Caso seja a dificuldade smart, é primeiramente calculado o valor respetivo de cada jogada possível (*calculate_moves_value(+Player, +Moves, -ValuedMoves)*). De seguida, a lista é percorrida para encontrar o valor máximo de uma jogada existente (*max_value_list(+ValuedMoves, -BestValue)*). Obtendo este valor, é criada uma terceira lista, desta vez apenas com as jogadas com valor máximo (*make_best_moves_list(+ValuedMoves,*

+BestValue, -BestMoves)). Finalmente, das melhores jogadas é escolhida uma aleatoriamente como a próxima jogada.

É importante referir que muitas das vezes que o AI inteligente, ao jogar contra si próprio, o jogo entrava num ciclo infinito, dado que as melhores jogadas incluíam apenas o movimento do rei.

Assim, para colmatar este problema, decidimos acrescentar um critério especial de escolha. Quando as jogadas ótimas incluem apenas mover o rei, é adicionada uma jogada aleatória à lista de jogadas ótimas, havendo assim a possibilidade de o AI fazer uma jogada menos aconselhável. Desta forma, com os testes que fomos correndo, o problema foi resolvido, uma vez que o AI pode simular a ocorrência de uma má jogada, ou pelo menos não ótima.

4. Conclusões

Com este projeto foi possível ganharmos uma noção muito mais clara das capacidades do Prolog como linguagem de programação. No início tínhamos dificuldade em perceber como seriamos capazes de implementar o nosso jogo, com todas as suas mecânicas e exceções, recorrendo apenas a átomos, variáveis, listas e predicados. Contudo, após a fase inicial de descobrir como replicar certas funcionalidades, mais comuns em outras linguagens, como ciclos for ou if..else statements, apercebemo-nos de como podíamos utilizar de forma eficiente o modo como Prolog funcionava, para obtermos predicados que, em poucas linhas, eram capazes de realizar operações razoavelmente complexas. Isto tornou o desenvolvimento do jogo bastante divertido (em certas alturas) e definitivamente fez-nos ter uma maior apreciação pela linguagem Prolog.

Apesar de estarmos bastante satisfeitos com a implementação do nosso jogo, como é natural, nem tudo está ao nosso agrado e com mais tempo teríamos feito algumas modificações. Primeiro, podíamos organizar muito melhor o código que temos. Apesar do esforço inicial para manter o código modular e bem estruturado, com o desenvolvimento do jogo e a necessidade de fazer *refactoring* a maiores secções de código, esse objetivo foi-se tornando mais complicado de seguir, o que leva a que tenhamos predicados em ficheiros que talvez estivessem melhor noutros, bem como predicados que não estão otimizados ou muito semelhantes a outros.

Em segundo, o “smart” AI podia ser otimizado. Um dos aspetos é na escolha de uma jogada aleatória quando as jogadas ótimas são apenas do rei (possibilita a ocorrência de ciclos). O ideal seria calcular o segundo melhor valor nas jogadas possíveis e adicionar aleatoriamente uma jogada com esse valor. Outro aspeto concerne o facto de as jogadas especiais do peão e do bispo não serem inteiramente consideradas aquando da escolha de os colocar em jogo, ou seja, apesar de, quando são jogadas, também ser feita a avaliação da melhor jogada para a sua habilidade especial, antes de o serem, esta habilidade não é tida em conta da forma que gostaríamos.

Por fim, gostaríamos de tornar a interface do utilizador mais robusta. Uma vez que usamos o predicado *read* do Prolog para ler o input do utilizador, se o utilizador colocar o input com virgulas ou pontos extra, é lançada uma exceção e o jogo é abortado.