Faculdade de Engenharia
da Universidade do Porto

Relatório Final

Programação em Lógica

Mestrado Integrado em Engenharia Informática e Computação

Zurero

Autores:
Afonso Jorge Ramos up201506239@fe.up.pt
João Dias Conde Azevedo up201503256@fe.up.pt

Novembro 2018

Conteúdo

1 Introdução

No âmbito da unidade curricular de Programação em Lógica, do curso Mestrado Integrado em Engenharia Informática e Computação, foi-nos sugerido o desenvolvimento de um jogo utilizando a linguagem de programação PROLOG. De entre as várias opções que nos foram disponibilizados pelos docentes foi escolhido o jogo Zurero. Após alguma investigação percebemos que o Zurero é um jogo interessante com características peculiares para o que é um jogo de tabuleiro clássico, como o "deslizar"das pedras.

O objetivo deste trabalho foi a aplicação dos primeiros conceitos interiorizados nas aulas teóricas e desenvolvidos nas aulas práticas da cadeira. Este método de avaliação torna-se importante pois permite-nos avaliar os conhecimentos que adquirimos até então e saber se somos ou não capazes de, com uma linguagem de programação nova e um paradigma completamente diferente do que estamos habituados.

2 Zurero

Zurero é jogado num tabuleiro de 19x19, inicialmente vazio. O jogo foi inventado por Jordan Goldstein em 2009. É uma interessante variação do clássico jogo *Go* onde dois jogadores, com cores distintas, devem "atirar"e fazer "deslizar"pedras ao longo do tabuleiro, desde as bordas do tabuleiro até que atinjam e parem noutra pedra.

O interessante e peculiar neste jogo é o facto de que caso uma pedra deslize e atinja outra que tem um espaço vazio atrás, esta última é empurrada.

O objetivo do jogo é colocar 5 pedras da cor do jogador seguidas, quer numa direção horizontal, vertical ou diagonal.

Por convenção, as peças pretas jogam primeiro e a primeira peça é colocada obrigatoriamente no centro do tabuleiro. Nas jogadas seguintes, alternando entre o jogador preto e o branco, um destes faz deslizar uma pedra de uma borda do tabuleiro para o centro, seguindo as regras de colisão já mencionadas.

Se ao empurrar uma pedra adversária o jogador coloca 5 peças do adversário numa coluna, fila ou diagonal, perde o jogo, a não ser que ele próprio faça uma coluna, fila ou diagonal, caso este em que ganha.

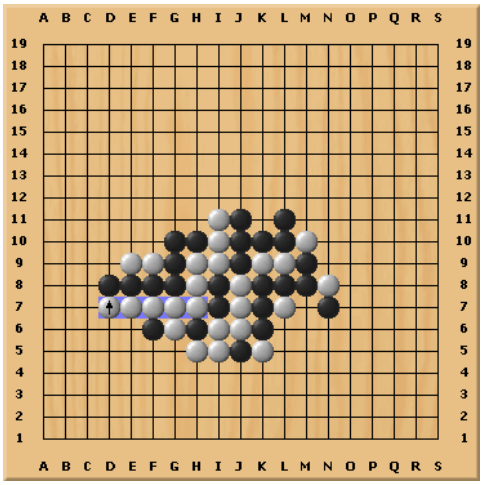

Figura 1: Exemplo de um tabuleiro a meio de jogo

3 Lógica do Jogo

3.1 Representação do Estado do Jogo

O tabuleiro de jogo é representado por uma lista de listas. O tabuleiro é de 19x19 **interseções** e, por isso, a primeira lista conterá outras 19, cada uma dessas com 19 elementos.

Para exemplificação, o código e comentários abaixo representam em linguagem PROLOG o estado inicial do tabuleiro e possíveis estados ao longo do jogo. Cada número de representação interna ao programa é traduzido em um ou mais caracteres na consola do SICStus.

A chave de tradução é também apresentada em baixo.

```
    /* Associates the internal representation with the view*/
        board_element(0,'-+-').
        board_element(1,'-B-'). %Black Pieces
        board_element(2,'-W-'). %White Pieces
```

Listing 1: Tradução da representação interna.

```
/* Starting game board */
initial_board([[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]]).
```

Listing 2: Board inicial.

```
/* Mid game board */
midgame_board([[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
```

```
4                 [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
5                 [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
6                 [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
7                 [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
8                 [0,0,0,0,0,0,0,0,2,0,0,0,0,0,0,0,0,0,0],
9                 [0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0],
10                [0,0,0,0,0,0,0,0,2,1,1,1,2,0,0,0,0,0,0],
11                [0,0,0,0,0,0,0,0,0,0,2,1,0,0,0,0,0,0,0],
12                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
13                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
14                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
15                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
16                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
17                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
18                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
19                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
20                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]]).
```

Listing 3: Board intermédio.

```
1 /* End game board */
2 endgame_board([[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
3                 [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
4                 [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
5                 [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
6                 [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
7                 [0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0],
8                 [0,0,0,0,0,0,0,1,2,0,0,0,0,0,0,0,0,0,0],
9                 [0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0],
10                [0,0,0,0,0,0,0,2,1,1,1,2,0,0,0,0,0,0,0],
11                [0,0,0,0,0,0,0,0,0,2,1,2,0,0,0,0,0,0,0],
12                [0,0,0,0,0,0,0,0,0,2,0,0,0,0,0,0,0,0,0],
13                [0,0,0,0,0,0,0,0,0,2,0,0,0,0,0,0,0,0,0],
14                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
15                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
16                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
17                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
18                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
19                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
20                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]]).
```

Listing 4: Board final.

3.2 Visualização do Tabuleiro

A representação interna do tabuleiro inicial será como a anteriormente apresentada, traduzindo-se no seguinte output de consola no SICStus Prolog:

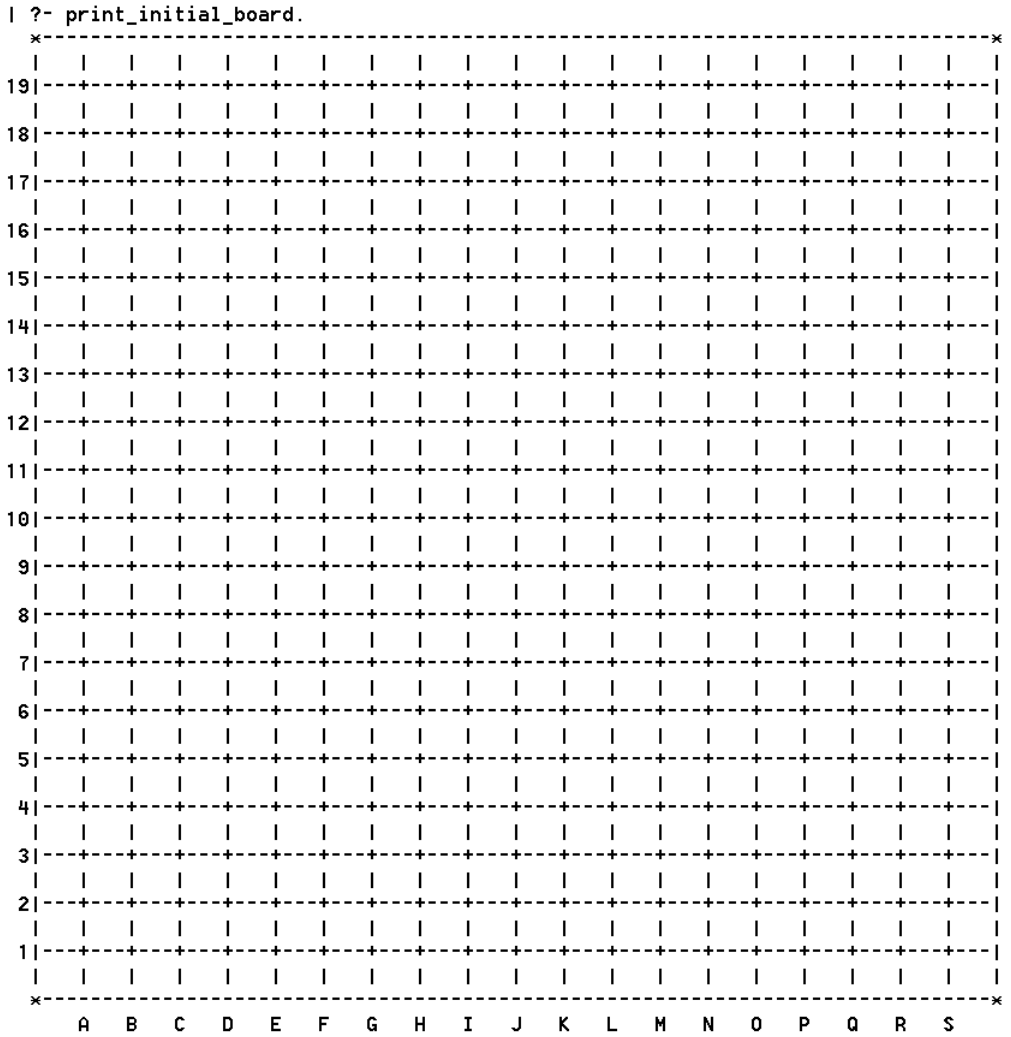

Figura 2: Layout de jogo em estado inicial.

Uma representação interna do tabuleiro a meio de um jogo como a acima apresentada traduz-se no seguinte output de consola no SICStus:

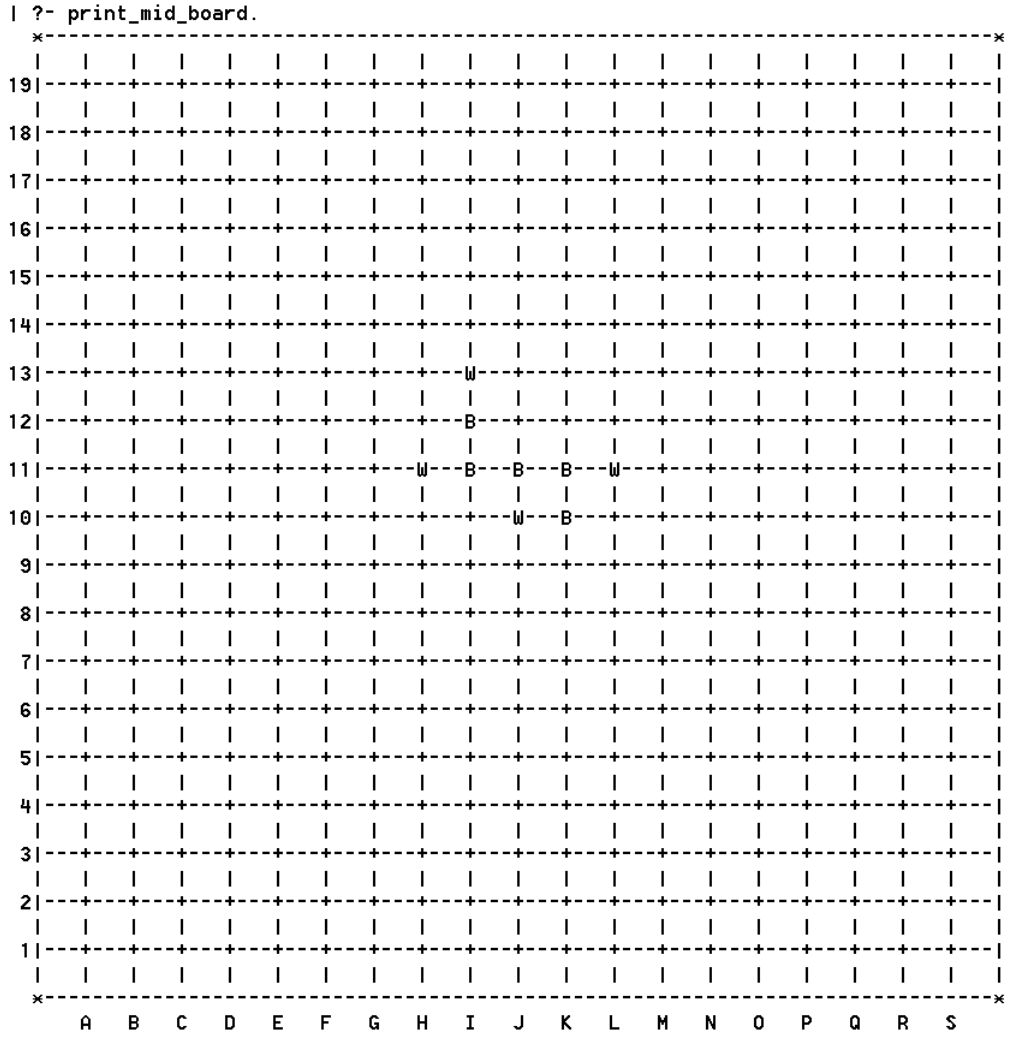

Figura 3: Layout de jogo em estado intermédio.

Ainda, uma possível representação do tabuleiro num estado de jogo final será a seguinte:

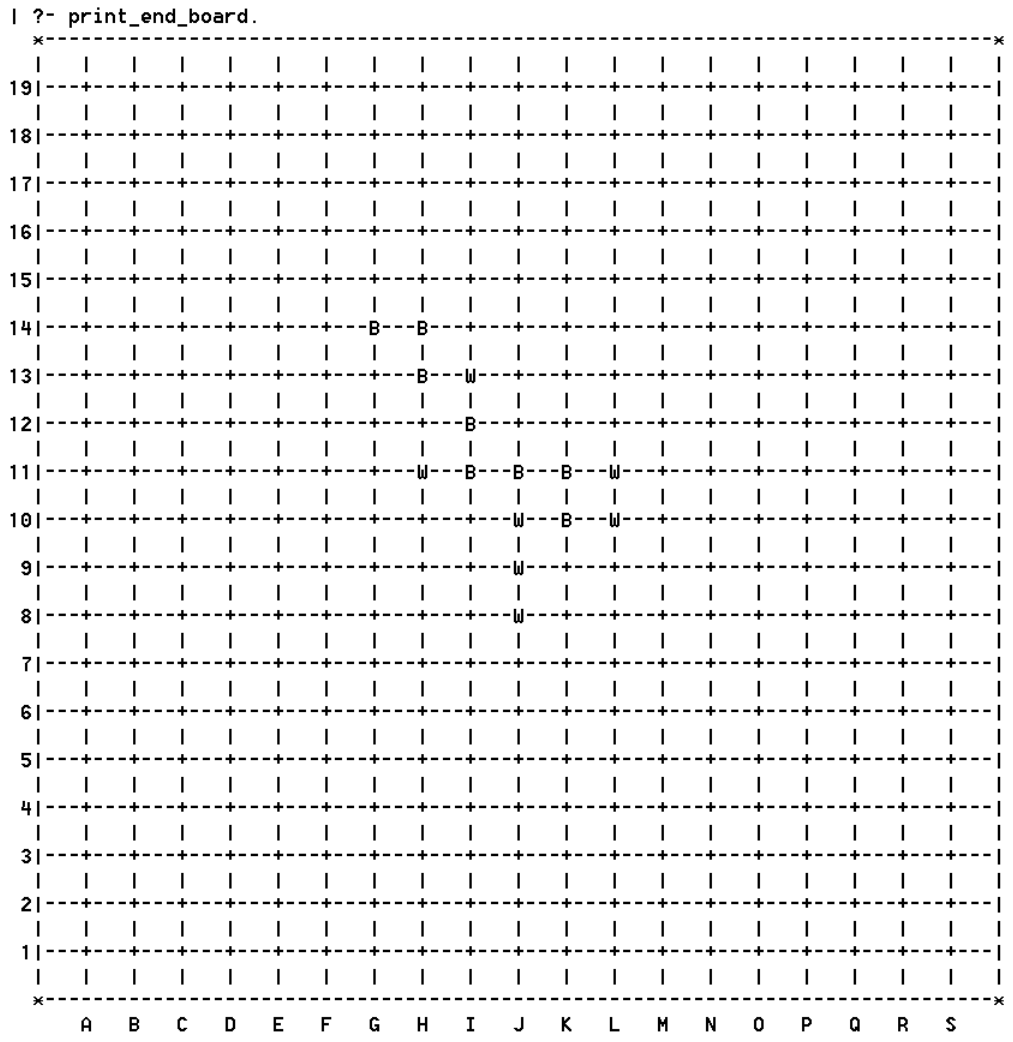

Figura 4: Layout de jogo em estado final.

Face a esta representação, necessária, devido ao facto de um tabuleiro de zurero ser disposto desta forma, com letras (crescentes da esquerda para a direita) para identificar as colunas e números (crescentes de baixo para cima) para identificar as linhas, conseguimos que para o utilizador, não haja dificuldades de adaptação a um tabuleiro diferente do utilizado em outros jogos de tabuleiro.

O tabuleiro inicial de jogo é criado usando o predicado *initial_board(-Board)* em que *Board* contem o tabuleiro inicial.

Para efeitos de apresentação foi construído o predicado *print_board(+Board)* que recebe uma matriz Board (lista de listas) e a imprime na consola, sendo que o tabuleiro atual é sempre representado no final de cada jogada, depois de a jogada ter sido aprovada pelos vários predicados de verificação. É um predicado que chama outro auxiliar, *print_board_aux(+Board,+LineNumbers)* que é chamado de forma recursiva, que se auxilia noutro predicado, também ele recursivo, *print_line(+Line)*.

3.3 Lista de Jogadas Válidas

A listagem de jogadas válidas pode ser obtida através do predicado *valid_moves(+Board, -ValidMoves)* onde recebe o *board* atual e retorna a lista de jogadas possíveis como pode ser observado na imagem a seguir.

Uma jogada apresenta-se como válida se a peça é deslizada numa linha ou coluna, quando feitas deslizar da esquerda/direita ou de cima/baixo, respetivamente, em que exista pelo menos uma outra peça.

```
1  valid_moves(Board, ValidMoves):-
2          findall(Coord-Dir, valid_move(Board, Coord, Dir), ValidMoves)
   .
```

Listing 5: Predicado que devolve a lista de jogadas válidas.

Um *"move"* é essencialmente um par Coordenada-Direção. A Direção pode ser *top/bot* ou *left/right*. Respetivamente, a Coordenada pode ser um índice de coluna ou de linha.

```
| ?-
| ?- initial_board2(_B), valid_moves(_B, ValidMoves).
ValidMoves = [4-top,5-top,6-top,7-top,8-top,9-top,10-top,11-top,
12-top,4-bot,5-bot,6-bot,7-bot,8-bot,9-bot,10-bot,11-bot,12-bot,
10-left,11-left,13-left,14-left,15-left,16-left,17-left,10-right
,11-right,13-right,14-right,15-right,16-right,17-right] ?
```

Figura 5: Exemplo do output do predicado valid_moves.

3.4 Execução de Jogadas

Em cada jogada, é chamado um predicado que engloba todas as ações necessárias dentro de um jogada, *player_move(+Board, +Player, -NewBoard)*, onde, dentro do mesmo é pedido ao jogador, que possui a vez de jogar, de onde pretende lançar a peça, entre baixo, cima, esquerda e direita com o predicado *get_direction(+Direction)*; e, depois, pede ao jogador a coluna ou linha, dependendo da resposta anterior, da qual pretende lançar a peça utilizando o predicado *get_coord(+Coord, -Direction)*.

Após o *input* do jogador da sua jogada, o programa valida a mesma dentro do tabuleiro de jogo, verificando se existe alguma peça na linha ou coluna para o qual o jogador quer lançar utilizando o predicado *valid_move(+Board, +Coord, +Direction)*. No nosso jogo, a validação de jogada por parte do utilizador é bastante acessível, visto que é independente de qual jogador está a jogar e visto que para ser válida apenas tem de existir uma peça na linha ou coluna a ser jogada, a não ser que seja a primeira jogada, pois esta é um caso especial no qual o jogador escolhe exatamente onde colocar a peça. Caso as coordenadas de origem e destino não sejam válidas, o programa retrocede até ao ponto inicial.

Imediatamente a seguir, o predicado *player_stone(?Player, ?Piece)* é chamado para que o programa saiba qual é a peça do jogador, e a seguir, é chamado o predicado *move(+Board, -NewBoard, +Coord, +Direction, +Piece)* para, efetivamente, mover a peça. Sendo que este analisa onde a peça vai parar, isto é, procura na linha ou coluna a peça existente (que já é conhecida a sua existência com o predicado *valid_move* mencionado anteriormente) e a sua posição. Adicionalmente, verifica se esta peça tem outra imediatamente a seguir que impeça o movimento desta, caso contrário a peça lançada pelo jogador ocupa a posição da peça já existente, fazendo a última deslocar-se uma posição na direção que o jogador lançou a segunda peça.

O predicado que chama todos os predicados de jogada mencionados anteriormente, *player_move*, está, por sua vez, dentro do predicado *player_turn(+Board, -NewBoard, +Player, -NextPlayer)*, que trata da chamada dos predicados de impressão de tabuleiro, de mudança de turno e, finalmente, é chamado o predicado *game_over(+NewBoard, +Player)* que se responsabiliza por analisar se o *Player* dado ganhou o jogo. Este predicado é chamado duas vezes para albergar os casos em que com o movimento de uma peça adversária, o oponente ganhe o jogo, no entanto, se essa mesma jogada faça, de alguma forma ambos jogadores ganhar, apenas o jogador que possui a vez é que é, efetivamente, o vencedor, e por isso a chamada do predicado com o jogador atual é a primeira.

Todas as jogadas, de qualquer um dos modos de jogo, com bot ou sem bot, ocorrem dentro do predicado *play_game([...])*, que possui cinco variações do mesmo:

- *play_game([Board, Player—Other])* - Predicado para a primeira jogada;
- *play_game([Board, Player, pvp])* - Predicado para os jogos Player versus Player;

```
1  play_game([Board, Player, pvp]):-
2      player_turn(Board, NewBoard, Player, NextPlayer),
3      play_game([NewBoard, NextPlayer, pvp]).
4
```

Listing 6: Ciclo de jogo Humano vs Humano.

- *play_game([Board, Player, pvb, Diff, bot])* - Predicado para os jogos Player versus Bot em que o Bot joga primeiro;

```
play_game([Board, Player, pvb, Diff, bot]):-
    bot_turn(Board, NewBoard, Player, NextPlayer, Diff),
    request_enter,
    player_turn(NewBoard, NewBoard2, NextPlayer, _),
    play_game([NewBoard2, Player, pvb, Diff, bot]).

```

Listing 7: Ciclo de jogo Computador vs Humano.

- *play_game([Board, Player, pvb, Diff, player])* - Predicado para os jogos Player versus Bot em que o Player joga primeiro;

```

play_game([Board, Player, pvb, Diff, player]):-
    player_turn(Board, NewBoard, Player, NextPlayer),
    bot_turn(NewBoard, NewBoard2, NextPlayer, _, Diff),
    request_enter,
    play_game([NewBoard2, Player, pvb, Diff, player]).

```

Listing 8: Ciclo de jogo Humano vs Computador.

- *play_game([Board, Player, bvb, Bot1Lvl, Bot2Lvl])* - Predicado para os jogos Bot versus Bot com as dificuldades desejadas.

```

play_game([Board, Player, bvb, Bot1Lvl, Bot2Lvl]):-
    bot_turn(Board, NewBoard, Player, NextPlayer, Bot1Lvl),
    request_enter,
    bot_turn(NewBoard, NewBoard2, NextPlayer, _, Bot2Lvl),
    request_enter,
    play_game([NewBoard2, Player, bvb, Bot1Lvl, Bot2Lvl]).

```

Listing 9: Ciclo de jogo Computador vs Computador.

3.5 Final do Jogo

No Zurero fomos presenteados com um problema de verificação de término de jogo interessante, pois se ao empurrar uma pedra adversária o jogador coloca 5 peças do adversário numa coluna, fila ou diagonal, perde o jogo, a não ser que ele próprio faça uma coluna, fila ou diagonal. Devido a este facto, tal como mencionado em 3.4 Execução de Jogadas, tivemos de fazer a chamada de verificação de término de

jogo duas vezes, para verificar, primeiro, se o jogador atual ganhou e, de seguida, se o oponente ganhou.

```
1  game_over(Board, Player):-
2      check_win(Board, Player),
3      clear_console,
4      print_board(Board),
5      print_msg_win_game(Player),
6      request_enter,
7      game_mode_menu.
```

Listing 10: Término de jogo.

O predicado *game_over(+NewBoard, +Player)* é constituído por uma chamada do predicado *check_win(+Board, +Player)*, que se suceder procede ao término do jogo com impressão do board final e após desejo do utilizador, volta ao menu inicial, mas caso contrário, simplesmente falha procedendo à jogada seguinte.

```
1  check_win(Board, Player):-
2      player_stone(Player, Piece),
3      cnt_in_a_row_lines(Board, Piece, LinesCnt),
4      cnt_in_a_row_cols(Board, Piece, ColsCnt),
5      cnt_in_a_row_diags(Board, Piece, DiagsCnt),
6      max_lists(Max, [LinesCnt, ColsCnt, DiagsCnt]),
7      Max >= 5.
```

Listing 11: Verificação de vitória de um jogador.

O predicado *check_win* é constituído, por uma chamada *player_stone(?Player, ?Piece)*, para que possamos saber qual é a peça do jogador atual, para depois procedermos, efetivamente, à verificação da vitória, contando o número de vezes em que a peça do jogador se repete sem intervalos tanto verticalmente, horizontalmente, como diagonalmente, através de um predicado para cada um dos casos, e depois, a averiguação se em algum destes predicados o aparecimento da peça sem intervalos ocorreu 5 vezes ou mais, onde caso se verifique, declaramos o jogador como vencedor.

3.6 Avaliação do Tabuleiro

Para efeitos de avaliação de tabuleiro, o predicado *evaluate_move*, atribui uma pontuação a cada movimento, com base em quantas peças seguidas o movimento consegue obter, seja com peças na horizontal, vertical ou diagonal.

```
1
2  value(Board, Piece, Eval):-
3      cnt_in_a_row_lines(Board, Piece, LinesCnt),
4      cnt_in_a_row_cols(Board, Piece, ColsCnt),
5      cnt_in_a_row_diags(Board, Piece, DiagsCnt),
6      aux_eval([LinesCnt, ColsCnt, DiagsCnt], Eval).
7
8  /* Game winning board, with 5 or more pieces in a row */
9  aux_eval(Cnts, Eval):-
10     max_lists(Max, Cnts),
11     Max >= 5,
12     max_int(MaxInt),
13     sum_lists(AuxEval, Cnts),
14     Eval is MaxInt + AuxEval.
15
16 aux_eval(Cnts, Eval):-
17     sum_lists(Eval, Cnts).
```

Listing 12: Predicado de avaliação de tabuleiro de jogo.

Pelo excerto de código acima vemos claramente como é feita a avaliação do tabuleiro de jogo. Um tabuleiro valerá tanto mais quanto mais peças do jogador em questão tiver em sequência, dado que o objetivo é colocar 5 seguidas. Assim, são contadas todas as linhas, colunas ou diagonais do tabuleiro e é feita uma soma do número de peças seguidas por cada componente.

Para além disso, e para garantir que o computador quando usar este predicado, atribua um valor máximo a um tabuleiro vitorioso, adicionamos um valor extremamente alto ao valor do tabuleiro caso numa das componentes haja 5 ou mais peças seguidas (tabuleiro vitorioso).

3.7 Jogada do Computador

Neste projeto foram implementadas duas dificuldades para as jogadas do computador, a fácil, que da lista de jogadas possíveis escolhe uma jogada aleatória, e a difícil, que, com o auxílio de um predicado de avaliação, escolhe a melhor jogada possível.

Ambas jogadas do computador são determinadas dentro do predicado *bot_move(+Diff, +Board, +Player, -NewBoard)*, que por sua vez está dentro do predicado *bot_turn(+Board, -NewBoard, +Player, -NextPlayer, +Diff)*, sendo que o argumento *Diff* determina a dificuldade do computador.

```
1
2 choose_move(hard, Board, Piece, BestMoveCoord-BestMoveDir):-
3     setof(Eval-Coord-Dir,
4                 (valid_move(Board, Coord, Dir),
5                  once(evaluate_move(Board, Coord, Dir, Piece, Eval))
6                 ),
7             AscValidMoves),
8     reverse(AscValidMoves, [_-BestMoveCoord-BestMoveDir|_]).
```

Listing 13: Predicado de escolha de jogada do computador, no nível difícil.

Dentro do predicado *bot_move*, na dificuldade **hard**, é chamado o predicado acima. O mesmo procura explorar todas as possibilidades de jogada através do *setof* combinado com o predicado *valid_move*. Utilizando o *setof*, procuramos todos os triplos Eval-Coord-Dir que correspondem ao valor de uma determinada jogada em Coord + Dir. O *setof* ordenará por ordem crescente de Eval, pelo que, revertendo a lista de jogadas possíveis, temos à cabeça da mesma a melhor jogada possível.

```
1
2 evaluate_move(Board, Coord, Dir, Piece, Eval):-
3     move(Board, NewBoard, Coord, Dir, Piece),
4     value(NewBoard, Piece, MyEval),
5     player_stone(Player, Piece),
6     enemy_player(Player, Enemy),
7     player_stone(Enemy, EnemyPiece),
8     value(NewBoard, EnemyPiece, EnemyEval),
9     Eval is MyEval - EnemyEval.
```

Listing 14: Predicado avaliação de jogada.

Apesar de ser dada uma penalização ao tabuleiro quanto melhor seja para o inimigo, não é aplicado nenhum procedimento do tipo minimax de forma a garantir que o adversário não ganha.

4 Conclusões

Com a realização do projeto o grupo aprendeu mais sobre a linguagem PROLOG e o paradigma de programação de declarativo, para além dos fundamentos da Programação em Lógica.

Os desafios apresentados foram de dimensão considerável e o grupo considera que o jogo em si era, de entre os apresentados, um dos mais complicados. Contudo, foram arranjadas boas soluções para os mesmos e a qualidade e organização de código foi sempre uma prioridade.

O grupo lamenta ainda não ter podido completar algum do trabalho que decorria de implementação de um algoritmo MiniMax para as jogadas do computador.

Em suma, o projeto era interessante e ambicioso e, com mais ou menos facilidade, o grupo implementou todas as funcionalidades e regras de jogo, procurando fazê-lo de forma organizada e sempre ao máximo declarativa.

5 Bibliografia

[**1**] igGameCenter

[**2**] Board Game Geek