Faculdade de Engenharia da Universidade Do Porto

Mestrado Integrado em Engenharia Informática e Computação

Programação em Lógica – 3º ano

T1 – Avaliação Final - Knight Line_1

Alunos:

João Miguel Vaz Tello da Gama Amaral – up201708805

João Nuno Rodrigues Ferreira – up201605330

18 de Novembro, 2018

Índice:

1. Introdução

Este projecto foi desenvolvido no âmbito da unidade curricular de Programação em Lógica, do curso Mestrado Integrado em Engenharia Informática e Computação. Para podermos realizar este projecto usámos o sistema SICStus Prolog e escrevemos o código na linguagem PROLOG.

O tema foi escolhido por nós e é o jogo de tabuleiro Knight Line. É um jogo bastante fácil de compreender, mas que ao mesmo tempo envolve um exigente raciocínio.

A realização deste jogo tinha como objetivo desenvolver a nossa programação lógica, estender o nosso conhecimento e experiência no paradigma da Programação Declarativa. Também tinha como propósito desenvolver a nossa perícia para raciocínio abstracto.

2. O Jogo: Knight Line

2.1 Descrição

Knight Line é um jogo abstracto de estratégia e de conexão minimalista para dois jogadores. Cada jogo tem uma duração média de 10 minutos. A origem do jogo é desconhecida.

No início do jogo, cada jogador tem 20 azulejos quadrados de cor preta ou branca. Estas peças são colocadas em duas pilhas adjacentes.

Cada jogada consiste em mover uma parte da pilha (deve ser deixado para trás, no mínimo, um tijolo) para um espaço vazio fazendo um "knight's move" (movimento baseado na peça cavalo do xadrez). Todas as peças têm de estar conectadas, pelo menos diagonalmente, durante o decorrer do jogo. O primeiro jogador a fazer uma linha de 4 ortogonalmente ou na diagonal ganha.

Ao contrário de jogos como xadrez, Knight Line não é jogado num tabuleiro, mas sim numa grelha imaginária e cujos limites não estão estabelecidos. O tamanho da grelha na qual devemos pensar é, portanto, o espaço ocupado pelas peças, juntamente com as casas adjacentes, nas quais ainda não há peças, mas que são possíveis jogadas.

Nota: Optámos por colocar as peças de cor preta representadas por um X e as peças brancas por O.

/////	0	1	2	3	4	5	6
A							
B		O,3	X,3	O,5			
C		O,3	X,3	X,2	O,1		
D		X,4	X,4	O,4	X,4	O,4	
E							

Fig. 1 – Exemplo de uma situação intermédia do jogo.

2.2. Início do jogo

Para o jogo começar devem ser colocadas as duas pilhas de 20 azulejos em duas posições adjacentes, horizontalmente. Tendo em conta que o jogador que possui as peças pretas será o primeiro a jogar, o que lhe dá alguma vantagem sobre o adversário, na sua primeira jogada apenas poderá mover 1 peça para a casa na qual a pretende colocar.

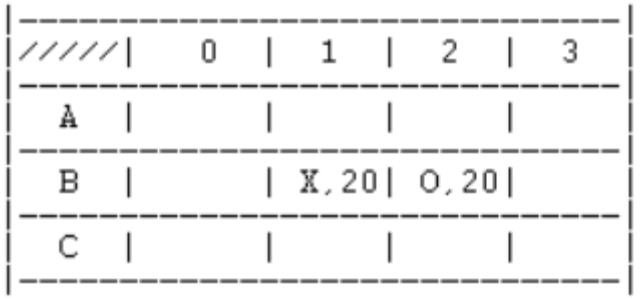

Fig. 2 – Situação inicial do jogo

2.3. Objetivo

O objetivo é conseguir fazer uma linha de 4, podendo cada elemento da linha ser uma única peça ou uma pilha com mais de uma.

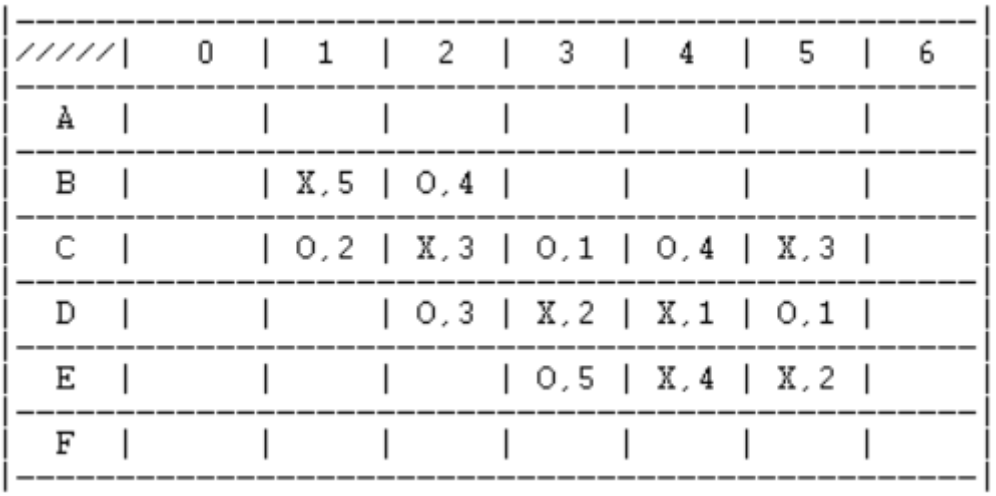

Fig. 3 – Exemplo de uma situação final do jogo (4 pretas em linha na diagonal).

2.4. Desenvolvimento

Em cada jogada, cada jogador pode movimentar uma porção das suas pilhas, deixando na posição inicial, pelo menos, uma peça. As jogadas têm de respeitar o movimento "knight's move". Porém a jogada só pode ser feita para uma posição adjacente a outra peça, quer seja uma peça do jogador ou do adversário.

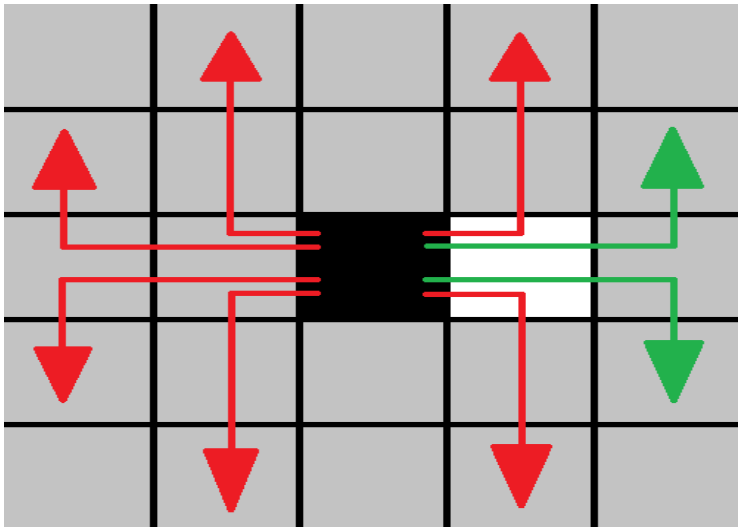

Fig. 4 – Possíveis jogadas iniciais da peça preta a verde e não possíveis a vermelho

2.5. Final

O fim do jogo acontece quando um jogador conseguir fazer uma linha de 4 com as suas respectivas peças, respeitando sempre o "knight's move" e nunca deixando espaços vazios em sítios onde já jogou.

3. Lógica do Jogo

3.1. Representação do Estado do Jogo

Para este jogo, optámos por definir o tabuleiro com sendo uma lista de listas de listas. A lista mais geral engloba todo o tabuleiro. Se considerarmos cada uma das listas dentro desta, estamos a falar de uma linha. Por fim, temos as listas que são constituídas sempre por um par de átomos, em que o primeiro representa a cor e o segundo o número de peças.

- Estado inicial (Fig. 2)

[[[empty, 0], [empty, 0], [empty, 0], [empty, 0]],

[[empty, 0], [black, 20], [white, 20], [empty, 0]],

[[empty, 0], [empty, 0], [empty, 0], [empty, 0]]].

- Exemplo de estado intermédio (Fig. 1)

[[[empty, 0], [empty, 0], [empty, 0], [empty, 0], [empty, 0], [empty, 0], [empty, 0]],

[[empty, 0], [white, 3], [black, 3], [white, 5], [empty, 0], [empty, 0], [empty, 0]],

[[empty, 0], [white, 3], [black, 3], [black, 2], [white, 1], [empty, 0], [empty, 0]],

[[empty, 0], [black, 4], [black, 4], [white, 4], [black, 4], [white, 4], [empty, 0]],

[[empty, 0], [empty, 0], [empty, 0], [empty, 0], [empty, 0], [empty, 0], [empty, 0]]]

- Exemplo de estado final (Fig. 3)

[[[empty, 0], [empty, 0], [empty, 0], [empty, 0], [empty, 0], [empty, 0], [empty, 0]],

[[empty, 0], [black, 5], [white, 4], [empty, 0], [empty, 0], [empty, 0], [empty, 0]],

[[empty, 0], [white, 2], [black, 3], [white, 1], [white, 4], [black, 3], [empty, 0]],

[[empty, 0], [empty, 0], [white, 3], [black, 2], [black, 1], [white, 1], [empty, 0]],

[[empty, 0], [empty, 0], [empty, 0], [white, 5], [black, 4], [black, 2], [empty, 0]],

[[empty, 0], [empty, 0], [empty, 0], [empty, 0], [empty, 0], [empty, 0], [empty, 0]]].

3.2. Visualização do Tabuleiro

Para ajudar na representação das peças e do número da linha, usamos 2 funções:

symbol(white, S) :- S='O'.

symbol(black, S) :- S='X'.

letter(1, S) :- S='A'.

…

letter (26, S) :- S='Z'.

Já para dar o display em si, usamos a função **display_game**, que recebe como argumento o tabuleiro e que chama algumas funções auxiliares:

```
display_game([H|T]) :-

    arrayLength(H, Columns),

    nl, write('|'), separation(Columns), write('|'), nl,
    write('|/////|   '), tableTop(Columns, 0), nl,
    write('|'), separation(Columns), write('|'),
    printBoard([H|T], Columns, 0, 1), nl, nl.
```

Fig. 5 – Predicado display_game

Este **display_game** irá chamar funções que, por sua vez, também irão chamar outras funções. São estas:

- **separation** – Imprime as divisórias entre linhas diferentes
- **tableTop** – Imprime o topo do tabuleiro, onde se encontram as referências, para o jogador, do número da coluna
- **printBoard** – Imprime todas as linhas do tabuleiro, juntamente com as letras, que são referências para o jogador, do número da linha
- **printRow** – Imprime toda uma linha do tabuleiro
- **printPair** – Imprime um par de átomos
- **printNumber** – Imprime o número de um par de átomos

3.3. Lista de Jogadas Válidas

Para obter jogadas válidas, não utilizamos qualquer tipo de predicado que devolvesse qualquer tipo de listas com jogadas válidas. Enquanto o jogo decorre, cada jogador tem a possibilidade de escolher a stack que quer jogar. Esta ação é feita recorrendo ao predicado **chooseStack**(Board, C, R, Player) onde Board é o estado atual do jogo, C e R são as coordenadas da stack e Player é o jogador atual.

```
chooseStack(Board, C, R, Player):-
    write('To play'), nl,
    write('Column : '), read(Ctest),
    write('Row    : '), read(Rtest),
    checkStackConditions(Board, Ctest, Rtest, Player, F1),
    chooseStackAux(Board, C, R, Player, F1, Ctest, Rtest).
chooseStackAux(Board, C, R, Player, 1, _, _) :-
    chooseStack(Board, C, R, Player).
chooseStackAux(_, C, R, _, 0, Ctest, Rtest) :-
    C is Ctest,
    R is Rtest.
```

Fig. 6 – Predicado chooseStack

Este predicado recorre ao predicado **checkStackConditions,** que é responsável pela verificação da peça estar dentro do tabuleiro, da stack pertencer ao jogador e se a stack contém mais do que uma peça. Se todas estas condições forem verificadas, então, o jogador pode efetuar uma jogada com esta stack.

```
checkStackConditions([H|T], Col, Row, Player, Flag) :-
    checkInsideBoard([H|T], Col, Row, F1),
    (F1 =:= 1 , write('The tile must belong to the board!'), nl, Flag is 1;
        checkPiece([H|T], Row, Col, F2),
        (F2 =\= Player , write('The tile is empty or belongs to another player!'), nl, Flag is 1;
            checkNumber([H|T], Row, Col, Number),
            (Number = 1 , write('The tile only contains one piece, it cant be moved!'), nl, Flag is 1;
            Flag is 0
            )
        )
    ).
```

Fig. 7 – Predicado checkStackConditions

Após ser feita a verificação da stack que se pretende jogar, recorremos ao predicado **chooseWhereToMove**(Board, C1, R1, C2, R2, Player), que é responsável pela verificação das condições da posição para onde se vai jogar, recebendo o estado atual do jogo, o jogador atual, a posição a partir da qual se vai mover a stack e C2 e R2 que vão ser usadas para guardar a posição válida para onde se quer jogar.

```
chooseWhereToMove(Board, C1, R1, C2, R2, Player):-
    write('Where to play'), nl,
    write('Column : '), read(Ctest),
    write('Row    : '), read(Rtest),
    checkTileConditions(Board, C1, R1, Ctest, Rtest, F1),
    chooseWhereToMoveAux(Board, C1, R1, C2, R2, Player, F1, Ctest, Rtest).
chooseWhereToMoveAux(Board, C1, R1, C2, R2, Player, 1, _, _) :-
    chooseWhereToMove(Board, C1, R1, C2, R2, Player).
chooseWhereToMoveAux(_, _, _, C2, R2, _, 0, Ctest, Rtest) :-
    C2 is Ctest,
    R2 is Rtest.
```

Fig. 8 – Predicado chooseWhereToMove

O predicado **chooseWhereToMove** recorre ao predicado **checkTileConditions**, que verifica se a posição escolhida para onde o movimento será efetuado pertence ao tabuleiro, se está vazia, se corresponde a um movimento em 'L' e tem peças adjacentes.

```
checkTileConditions([H|T], C1, R1, C2, R2, Flag) :-
    checkInsideBoard([H|T], C2, R2, F1),
    (F1 =:= 1 , write('The tile must belong to the board'), nl, Flag is 1;
        checkPiece([H|T], R2, C2, F2),
        (F2 =\= 2 , write('The tile must be empty'), nl, Flag is 1;
            checkLPlay([H|T], C1, R1, C2, R2, F3),
            (F3 =:= 0 , write('The play must be in an L shape'), nl, Flag is 1;
                checkAdjacents([H|T], R2, C2, F4),
                (F4 =:= 0 , write('The tile must have adjacent pieces'), nl, Flag is 1;
                    Flag is 0
                )
            )
        )
    ).
```

Fig. 9 – Predicado checkTileConditions

No caso dos bots, cada algoritmo tem de respeitar estas condições de jogada referidas em cima. O bot mais inteligente verifica todas as posições livres, fazendo filtragem por nível (quantas mais peças em linha estiverem perto da casa vazia melhor) para depois verificar se alguma peça consegue efetuar um movimento em 'L' para essa casa e fazer o movimento.

Em conclusão, não usamos qualquer tipo de técnica de armazenamento de jogadas possíveis numa lista. Em vez disso, verificamos sempre a jogada escolhida utilizando predicados que verificam as condições de jogo.

3.4. Execução de jogadas

Para efetuar a execução de jogadas, é utilizado o predicado **move**(Board, Player, C1, R1, C2, R2, Np, Board1). Neste predicado é recebido o estado atual do jogo (Board), o jogador que está a efetuar a jogada (Player), coordenadas da stack que foi selecionada para ser movida (C1, R1), coordenadas da posição para onde se vai efetuar a jogada (C2, R2), número de peças que vão ser movidas (Np) e Board1 vai ser o tabuleiro de jogo alterado.

```
move(Board, Player, C1, R1, C2, R2, Np, Board1) :-
    (Player \= 0,
        makeMoveB(Board, C1, R1, C2, R2, Np, Board1);
        makeMoveW(Board, C1, R1, C2, R2, Np, Board1)
    ).
```

Fig. 10 – Predicado move

O predicado **move** recorre a outros dois predicados, sendo eles: **makeMoveB** e **makeMoveW**. B e W representam o tipo de peças a ser movidas, caso o **Player** em move seja 0 é utilizada a **makeMoveW**, caso o Player em move seja 1 é utilizada a **makeMoveB**.

Os predicados **makeMove** tratam de atualizar a stack escolhida para se mover conforme o número de peças selecionado (**updatePiece**), mover o número de peças selecionado para a casa selecionada (**movePiece**) e por fim verificar se é necessário acrescentar colunas e linhas no tabuleiro (**boardResize**).

```
updatePiece([H|T], New, 0, IndCol, Number, Player, Final):-
    updatePieceAux(H, IndCol, Number, Z),
    replace(H, IndCol, [Player,Z], R),
    append([R], T, Q),
    append(New, Q, Final).
updatePiece([H|T], New, IndRow, IndCol, Number, Player, Final):-
    IndRow1 is IndRow - 1,
    append(New, [H], NewT),
    updatePiece(T, NewT, IndRow1, IndCol, Number, Player, Final).
updatePieceAux([[_|T1]|_], 0, Number, Z):-
    Z is T1 - Number.
updatePieceAux([_|T], IndCol, Number, Z):-
    IndCol1 is IndCol - 1,
    updatePieceAux(T, IndCol1, Number, Z).
```

Fig. 11 – Predicado updatePiece

```
movePiece([H|T], New, 0, IndCol, Number, Player, Final):-
    replace(H, IndCol, [Player,Number], R),
    append([R], T, Q),
    append(New, Q, Final).
movePiece([H|T], New, IndRow, IndCol, Number, Player, Final):-
    IndRow1 is IndRow-1,
    append(New, [H], NewT),
    movePiece(T, NewT, IndRow1, IndCol, Number, Player, Final).
```

Fig. 12 – Predicado movePiece

```
boardResize([H|T], IndC, IndR, [H4|T4]) :-
    arrayLength(H, Columns),
    arrayLength([H|T], Rows),
    Col is Columns - 1,
    Row is Rows - 1,
    ((IndC =:= 0 ; IndR =:= 0 ; IndC =:= Col ; IndR =:= Row) ,
        addColumnEnd([H|T], [H1|T1]),
        addRowEnd([H1|T1], [H2|T2]),
        addRowStart([H2|T2], [H3|T3]),
        addColumnStart([H3|T3], [H4|T4]);
        append([], [H|T], [H4|T4])).
```

Fig. 13 – Predicado boardResize

3.5. Final do Jogo

Para efetuar a verificação de fim de jogo, é utilizado o predicado **game_over**(X, Player, MaxRow, MaxCol, Win, Lose). Neste predicado, é recebido o estado atual do jogo (X), o último jogador a efetuar uma jogada (Player), o número de linhas do tabuleiro atual (MaxRow) e o número de colunas do tabuleiro atual (MaxCol). Win e Lose são argumentos que vão receber um certo valor dependendo da vitória de um jogador (conseguindo um quatro em linha) ou da derrota do mesmo (por não ter mais peças para jogar: todas as stacks com uma peça).

O predicado **game_over** recorre a mais dois predicados: **checkWin** e **checkIfPossible**. O predicado **checkWin** percorre o tabuleiro atual e verifica, peça a peça, se esta tem na sua horizontal, vertical ou diagonais mais três peças do mesmo tipo. Se tal acontecer, então a vitória é atribuída ao jogador que acabou de jogar, mudando o argumento Win para 1. O predicado **checkIfPossible** percorre o tabuleiro todo procurando todas as peças que correspondam ao argumento Player e verifica se todas as stacks contêm apenas uma peça. Se esta condição se verificar, então o jogador não pode efetuar mais jogadas. O argumento Lose muda para 1 e o jogador perde o jogo.

```
game_over(X, Player, MaxRow, MaxCol, Win, Lose) :-
    checkWin(X, Player, MaxRow, MaxCol, 0, 0, Win),
    checkIfPossible(X, Player, MaxRow, MaxCol, 0, 0, Lose).
```

Fig. 14 – Predicado game_over

```
checkWin(_, _, MaxRow, MaxCol, MaxRow, MaxCol, Win):-
    Win is 0.
checkWin(X, Player, MaxRow, MaxCol, Row, Col, Win):-
    checkPiece(X, Row, Col, Flag),
    Col1 is Col+1,
    (Flag =:= Player ,
        check(X, Row, Col, Player, W1),
        (W1 =:= 1 ,
            Win is 1;
            (Col =:= MaxCol ,
                Row1 is Row +1,
                checkWin(X, Player, MaxRow, MaxCol, Row1, 0, Win);
                checkWin(X, Player, MaxRow, MaxCol, Row, Col1, Win)
            )
        );
        (Col = MaxCol ,
            Row1 is Row + 1,
            checkWin(X, Player, MaxRow, MaxCol, Row1, 0, Win);
            checkWin(X, Player, MaxRow, MaxCol, Row, Col1, Win)
        )
    ).
```

Fig. 15 – Predicado checkWin

```
checkIfPossible(_, _, MaxRow, MaxCol, MaxRow, MaxCol, Lose):-
    Lose is 1.
checkIfPossible(X, Player, MaxRow, MaxCol, Row, Col, Lose):-
    checkPiece(X, Row, Col, Flag),
    Col1 is Col+1,
    (Flag =:= Player ,
        checkNumber(X, Row, Col, Num),
        (Num \= 1 ,
            Lose is 0;
            (Col =:= MaxCol ,
                Row1 is Row +1,
                checkIfPossible(X, Player, MaxRow, MaxCol, Row1, 0, Lose);
                checkIfPossible(X, Player, MaxRow, MaxCol, Row, Col1, Lose)
            )
        );
        (Col = MaxCol ,
            Row1 is Row + 1,
            checkIfPossible(X, Player, MaxRow, MaxCol, Row1, 0, Lose);
            checkIfPossible(X, Player, MaxRow, MaxCol, Row, Col1, Lose)
        )
    ).
```

Fig. 16 – Predicado checkIfPossible

3.6. Avaliação do Tabuleiro

Em relação à avaliação do tabuleiro, não possuímos um predicado como o value, que nos retorne um determinado valor específico, dentro de um certo intervalo. Em compensação, possuímos os predicados **checkNumberHorizontal**, **checkNumberVertical**, **checkNumberDiagonal1** e **checkNumberDiagonal2**, que recebem todos os mesmos parâmetros. São estes (X, Player, Col, Row, Number), onde X representa o tabuleiro, Player o jogador cujas peças iremos avaliar, Col e Row as coordenadas da posição empty a avaliar e Number será a variável cujo valor será alterado e que nos irá fornecer informação sobre a posição em questão e o jogador em questão. Cada um destes predicados irá chamar outros 2, diferentes para cada uma das também diferentes direcções, que serão responsáveis por auxiliar aquelas 4 principais. Assim sendo, estes predicados irão receber uma posição que corresponde a um empty e irão determinar quantas peças do jogador Player há para cada um dos lados. Este valor estará sempre contido num intervalo que vai de 0 (não há peças do jogador para nenhum dos lados) até 3 (há no mínimo 3 peças do jogador, juntas a este emtpy, para qualquer um dos lados, ou até somadas). Desta forma, se, por exemplo, houver uma parte do tabuleiro como a que se segue:

[white, 3], [empty, 0], [white, 4], [white, 1]

O predicado **checkNumberHorizontal** quando chamado para a posição empty, irá retornar o valor 3 na variável Number. Assim, uma jogada white para esta posição será a ideal, pois resultará na vitória do jogador que possui as peças brancas. Ainda em relação a este tópico, o predicado **checkAll**(X, Player, Col, Row, Max, Flag) é responsável por chamar os 4 predicados acima descritos, recebendo como parâmetros, para além do tabuleiro, do jogador e da posição empty, um valor Max e uma Flag. O valor Max será sempre um valor compreendido entre 0 e 3, que será usado no predicado **pcMove** e a Flag irá tomar o valor de 1 ou 0, caso não haja esse número de peças ou caso haja.

```
checkAll(X, Player, Col, Row, Max, Flag):-
    checkNumberHorizontal(X, Player, Col, Row, N1),
    checkNumberVertical(X, Player, Col, Row, N2),
    checkNumberDiagonal1(X, Player, Col, Row, N3),
    checkNumberDiagonal2(X, Player, Col, Row, N4),
    ((N1 = Max ; N2 = Max; N3 = Max; N4 = Max),
        Flag is 0;
        Flag is 1
    ).
```

Fig. 17 – Predicado checkAll

Por fim, o predicado **pcMove** (que será melhor explicado no tópico seguinte, 3.7. Jogada do Computador) recebe vários argumentos, nomeadamente aquele que nos interessa para este caso, que é o Max. Esta variável irá ser inicializada a 3 e irá sendo decrementada até 0. Representa o número de peças em linha, de determinado jogador, que uma posição empty poderá ter nos seus espaços adjacentes. Começando a 3, e usando o predicado **checkAll**, o algoritmo irá procurar emptys com 3 peças em linha. Em caso de insucesso, o valor de Max será decrementado para 2, e o algoritmo irá, desta vez, procurar 2 peças em linha, e assim sucessivamente. Assim que, com o atual valor Max, se encontrar uma peça que seja capaz de se mover para a posição empty, que é a melhor opção para se jogar, o programa deixará de correr o predicado **pcMove**, deixando assim, o valor de Max como a forma de avaliação do estado do jogo para cada jogador.

3.7. Jogada do Computador

O predicado c**hoose_move**(Board, Player, Level, C1, R1, C2, R2, Np, Counter) é usado para efetuar a jogada do computador. Neste predicado, é recebido o estado atual do jogo (Board), o jogador que vai efetuar a jogada (Player), o nível de dificuldade do bot (Level) e a variável utilizada para alternar entre jogadores na recursiva de jogo (Counter). No fim da função, as coordenadas da stack escolhida para mover são colocadas em C1 e R1, as coordenadas da posição para onde se vai mover peças são colocadas em C2 e R2 e, por fim, o número de peças a jogar é colocado em Np. Caso a jogada seja a primeira do jogo todo, o parâmetro Np é sempre 1.

Este predicado recorre a mais dois predicados: **pcMoveRandom** e **pcMove**. O predicado **pcMoveRandom** é utilizado se o nível de dificuldade (Level) for igual a 1, caso este parâmetro seja 2 é utilizado o predicado **pcMove**.

O predicado **pcMoveRandom** calcula as dimensões atuais do tabuleiro recorrendo a checkLengths(X, RowSize, ColSize). Após colocar as dimensões em RowSize e ColSize, escolhe uma peça random, de acordo com o jogador, e um número de peças para jogar recorrendo a chooseRandomPieceNumber(X, Player, ColSize, RowSize, C1, R1, Np). Finalmente, escolhe um movimento em L random, recorrendo a chooseRandomLMove(X, Player, ColSize, RowSize, C1, R1, C2, R2).

```
pcMoveRandom(X, Player, C1, R1, C2, R2, Np) :-
    checkLengths(X, RowSize, ColSize),
    chooseRandomPieceNumber(X, Player, ColSize, RowSize, C1, R1, Np),
    chooseRandomLMove(X, Player, ColSize, RowSize, C1, R1, C2, R2).
```

Fig. 18 – Predicado pcMoveRandom

```
chooseRandomPieceNumber(X, Player, ColSize, RowSize, Col, Row, Number) :-
    random(0, ColSize, C1),
    random(0, RowSize, R1),
    checkPiece(X, R1, C1, F1),
    (F1 =\= Player ,
        chooseRandomPieceNumber(X, Player, ColSize, RowSize, Col, Row, Number);
        checkNumber(X, R1, C1, MaxNum),
        (MaxNum =:= 1 ,
            chooseRandomPieceNumber(X, Player, ColSize, RowSize, Col, Row, Number);
            random(1, MaxNum, Number),
            Col is C1,
            Row is R1
        )
    ).
```

Fig. 19 – Predicado chooseRandomPieceNumber

```
chooseRandomLMove(X, Player, ColSize, RowSize, PieceC, PieceR, TileC, TileR) :-
    random(0, ColSize, C2),
    random(0, RowSize, R2),
    checkPiece(X, R2, C2, F1),
    (F1 =\= 2 ,
        chooseRandomLMove(X, Player, ColSize, RowSize, PieceC, PieceR, TileC, TileR);
        checkLPlay(X, PieceC, PieceR, C2, R2, F2),
        (F2 =\= 1 ,
            chooseRandomLMove(X, Player, ColSize, RowSize, PieceC, PieceR, TileC, TileR);
            checkAdjacents(X, R2, C2, F3),
            (F3 =\= 1 ,
                chooseRandomLMove(X, Player, ColSize, RowSize, PieceC, PieceR, TileC, TileR);
                TileC is C2,
                TileR is R2
            )
        )
    ).
```

Fig. 20 – Predicado chooseRandomLMove

O predicado **pcMove** recebe o estado atual do jogo, o jogador que vai efetuar a jogada, e um argumento Max, que representa o máximo de peças daquele jogador que uma posição desocupada pode ter nas redondezas. Por outras palavras, é um argumento que é inicializado a 3, fazendo com que o algoritmo procure posições vazias com três peças em linha. Caso não tenha sucesso, o argumento é decrementado recursivamente. Finalmente, os argumentos From e To são listas que vão conter a stack que se vai jogar e a posição para onde se vai jogar de acordo com o algoritmo ganancioso implementado.

O predicado recorre ainda a **botCoordinates.** Este predicado recebe a lista que contém as posições das peças do jogador atual e todas as posições vazias que respeitem o argumento Max. Este predicado itera a lista de posições vazias totalmente para cada peça da lista das peças do jogador. Assim que encontra uma casa vazia que respeite as condições do movimento, seleciona a peça e a casa vazia para efetuar a jogada.

```
pcMove(_, _, -1, _, _).
pcMove(X, 0, Max, From, To):-

    checkEmpty(X, X, 0, 0, Max, [], Jog, 0),
    getWhites(X, 0, 0, [], Coords),

    botCoordinates(X, Jog, Coords, Coords, Piece, Tile),
    ((Piece = [], Tile = []) ,
        Max1 is Max-1,
        pcMove(X, 0, Max1, From, To);
        From = Piece,
        To = Tile
    ).
pcMove(X, 1, Max, From, To):-

    checkEmpty(X, X, 0, 0, Max, [], Jog, 1),
    getBlacks(X, 0, 0, [], Coords),

    botCoordinates(X, Jog, Coords, Coords, Piece, Tile),
    ((Piece = [], Tile = []) ,
        Max1 is Max-1,
        pcMove(X, 1, Max1, From, To);
        From = Piece,
        To = Tile
    ).
```

Fig. 21 – Predicado pcMove

```
botCoordinates(_, [], _, _, Piece, Tile):-
    Tile = [],
    Piece = [].
botCoordinates(X, [_|T1], [], Y, Piece, Tile):-
    botCoordinates(X, T1, Y, Y, Piece, Tile).
botCoordinates(X, [H1|T1], [H2|T2], Y, Piece, Tile):-
    botCoordinatesAux(X, H1, H2, F),
    (F = 1 ,
        Tile = H1,
        Piece = H2
    ;
        botCoordinates(X, [H1|T1], T2, Y, Piece, Tile)
    ).
botCoordinatesAux(X, [H|T], [H1|T1], Flag):-
    checkLPlay(X, H, T, H1, T1, F),
    checkAdjacents(X, H, T, F1),
    ((F = 1, F1 = 1) , Flag is 1 ; Flag is 0).
```

Fig. 22 – Predicado botCoordinates

4. Conclusões

O principal objetivo deste projeto foi aplicar o conhecimento adquirido nas aulas práticas e teóricas da UC Programação em Lógica (PLOG). Ao longo do desenvolvimento do trabalho fomos passando por várias dificuldades que envolviam a recursividade de prolog. No entanto, foram todas superadas com sucesso.

Existem aspetos no trabalho que podiam ser melhorados, tal como o uso de inteligência artificial. O algoritmo implementado é bom, mas podia recorrer a mais pormenores para ser ainda mais eficiente. Isto, porque o nosso bot utiliza um algoritmo ganancioso, que apenas tem em conta a jogada seguinte, quando podíamos ter visualizado o jogo para além da próxima jogada.

Concluindo, todo o trabalho foi realizado com sucesso e devido a este mesmo conseguimos desenvolver o nosso conhecimento sobre a linguagem prolog.

5. Bibliografia

- https://nestorgames.com/#knightline_detail
- https://boardgamegeek.com/boardgame/146989/knight-line
- https://boardgamegeek.com/thread/1594753/clever-little-game-which-deserves-more-attention