Faculdade de Engenharia da
Universidade do Porto

Mestrado Integrado em Engenharia Informática
e Computação

Programação Lógica - 3º ano

Neutreeko

Grupo : Neutreeko_2

Realizado por:

Ricardo Araújo Boia up201505244
Tiago Araújo Castro up201606186

Contents

1 Introdução

Este projeto foi desenvolvido no âmbito da unidade curricular de Programção Lógica, inserida no 3º ano do Mestrado Integrado em Engenharia Informática e Computação.

Tinhamos como objetivo neste projeto o desenvolvimento, em linguagem *Prolog*, do jogo de tabuleiro *Neutreeko* com três modos de utilização - Humano/Humano, Humano/Computador e Computador/Computador - e com dois niveis de dificuldade para o computador - fácil e difícil.

Para uma representação correta do tabuleiro na consola, deverá ser usada a font **Courier** com tamanho **12**, devido a sua propriedade monospaced.

2 História e regras do jogo

2.1 História

O *Neutreeko* é jogado par dois jogadores num tabuleiro 5x5. Este jogo foi inventado em 2001 por Jan Kristian Haugland, um professor de matemática e física norueguês, com um doutoramento em matemática realizado em Oxford. O nome do jogo resulta de uma amálgama entre Neutron e Teeko, os dois jogos nos quais é baseado.

2.2 Inicio do jogo

Cada jogador possui três peças de uma só cor (preto ou branco). O jogo inicia-se sempre com o turno das peças pretas e com a seguinte disposição do tabuleiro:

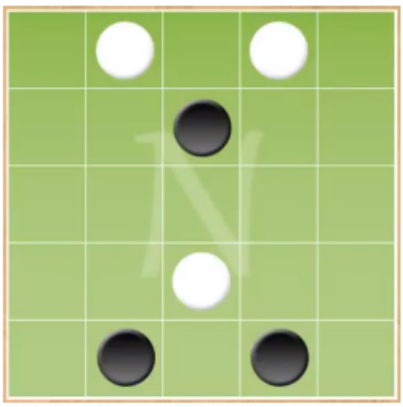

Figure 1: Disposição inicial do tabuleiro

2.3 Turno do jogador

Como foi referido anterioramente, as peças pretas iniciam o jogo. Cada turno consiste no movimento, ortogonal ou diagonal, de uma só peça do jogador. A peça irá deslizar numa determinada direção até encontrar a borda do tabuleiro ou outra peça. O movimento só é considerado válido se a posição final da peça for diferente da inicial.

2.4 Final do jogo

Um jogador é declarado vencedor se conseguir alinhar, ortogonal ou diagonalmente, as suas peça em quadrados adjacentes. Já o empate acontece se a mesma disposição de tabuleiro ocurrer três vezes. No caso de uma situação onde se verifica a vitória de um jogador e o empate, a prioridade é dada à vitória do jogador.

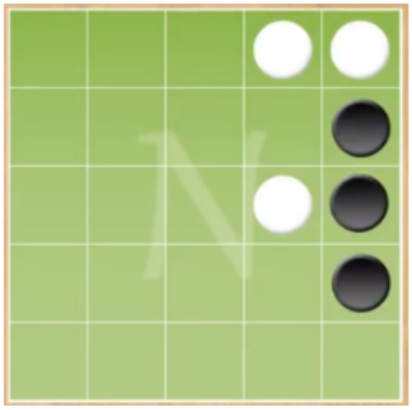

Figure 2: Disposição final do tabuleiro onde se registou a vitória das pretas

3 Lógica do jogo

3.1 Representação do Estado do Jogo

Para a representação do tabuleiro de jogo foi utilizada uma lista de listas 5x5, que vai ser alterada durante o jogo com o movimento das peças. O espaço vazio é representado por 'x', a peça branca por 'w' e a peça preta por 'b'.

3.1.1 Estado inicial do tabuleiro

```
[['x', 'w', 'x', 'w', 'x'],
['x', 'x', 'b', 'x', 'x'],
['x', 'x', 'x', 'x', 'x'],
['x', 'x', 'w', 'x', 'x'],
['x', 'b', 'x', 'b', 'x']]
```

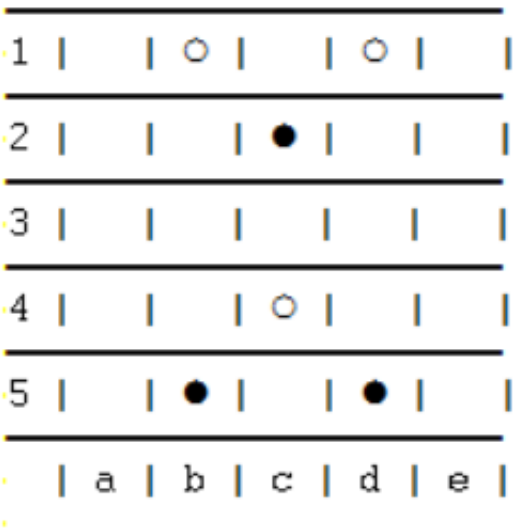

Figure 3: Disposição inicial do tabuleiro na consola

3.1.2 Estado intermédio do tabuleiro

```
[['x', 'x', 'x', 'w', 'x'],
['b', 'x', 'x', 'x', 'x'],
['x', 'x', 'x', 'x', 'x'],
['b', 'w', 'x', 'x', 'x'],
['x', 'w', 'b', 'x', 'x']]
```

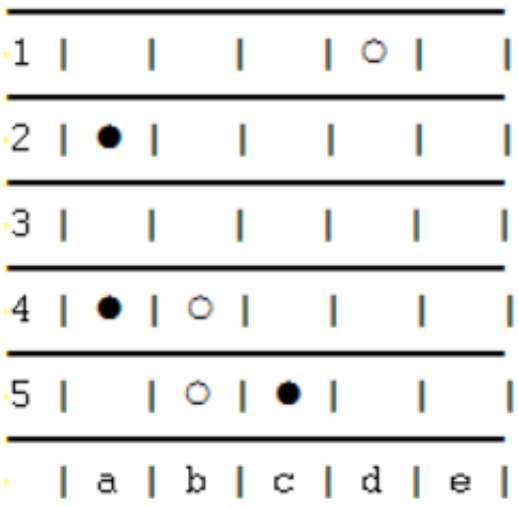

Figure 4: Disposição intermedia do tabuleiro na consola

3.1.3 Estado final do tabuleiro

```
[['x', 'x', 'x', 'w', 'w'],
 ['x', 'x', 'x', 'x', 'b'],
 ['x', 'x', 'x', 'w', 'b'],
 ['x', 'x', 'x', 'x', 'b'],
 ['x', 'x', 'x', 'x', 'x']]
```

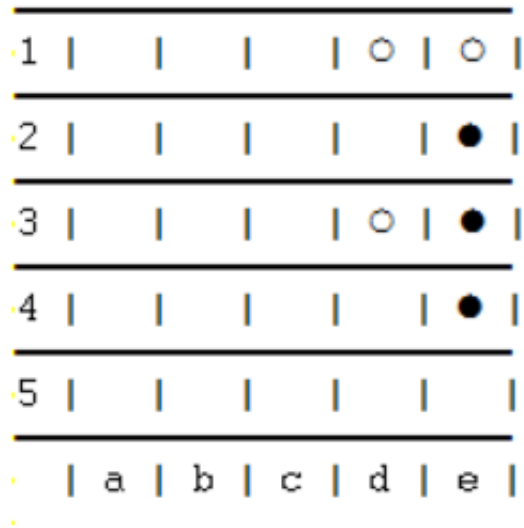

Figure 5: Disposição final do tabuleiro na consola

3.2 Visualização do Tabuleiro

Para a representação do tabuleiro foi utilizado *unicode* de forma a utilizar círculos pretos e brancos para as peças. O desenho é feito pelo predicado recursivo **display_game** e pelas funções auxiliares:

- **display_line** - desenha as cinco posições de cada linha e a separação entre elas.
- **display_columns** - desenha as letras (a-e) utilizadas como coordenadas das colunas.
- **display_border** - desenha as linhas utilizadas no topo e base do tabuleiro, e também na separação das linhas.
- **write_char** - converte cada um dos carateres utilizados na representação do estado de jogo para o *unicode* correspondente.

O código dos carateres de *unicode* usados são os seguintes:

- 0x25CF - peça preta.
- 0x25CB - peça branca.
- 0x2503 - barras verticais utilizadas na separação de posições.
- 0x2501 - barras horizentais utilizadas na separação de posições.

Algumas imagens do desenho do tabuleiro na consola encontram-se na secção anterior.

3.3 Lista de Jogadas Válidas

A obtenção da lista de jogadas válidas está ao cargo de **valid_moves**, que irá colocar numa lista todas as direções válidas para o movimento da peça recebida como argumento, recorrendo a **isMoveValid** para verificar se a posição adjacente é um espaço vazio para cada uma das oito direções. Neste caso, a direção é adicionada à lista de jogadas válidas, caso contrário isto não acontecerá e a direção será considerada não válida.

```
%Gets the list of valid moves for a piece
valid_moves(Board, Row, Column, Player, ListOfMoves) :-
    RowDown is Row + 1,
    RowUp is Row - 1,
    ColumnRight is Column + 1,
    ColumnLeft is Column - 1,
    isMoveValid(Board, RowUp, Column, 1, [], OutList1),
    isMoveValid(Board, Row, ColumnLeft, 2, OutList1, OutList2),
    isMoveValid(Board, Row, ColumnRight, 3, OutList2, OutList3),
    isMoveValid(Board, RowDown, Column, 4, OutList3, OutList4),
    isMoveValid(Board, RowUp, ColumnRight, 5, OutList4, OutList5),
    isMoveValid(Board, RowUp, ColumnLeft, 6, OutList5, OutList6),
    isMoveValid(Board, RowDown, ColumnRight, 7, OutList6, OutList7),
    isMoveValid(Board, RowDown, ColumnLeft, 8, OutList7, ListOfMoves).

%If the movement in a certain direction is valid, that direction is added to the
%list of valid moves
isMoveValid(Board, Row, Column, Dir, InList, OutList) :-
    (getPiece(Row, Column, Board, Piece),
    Piece = 'x', append(InList, [Dir], OutList));
    append(InList, [], OutList).
```

3.4 Execução de Jogadas

A execução das jogadas é realizada pelo predicado **move** que irá "perguntar" ao jogador a posição da peça que pretende mover e a direção em que se realizar o movimento, validar o *input* e atualizar o tabuleiro.

Os predicados **getMovingPiece** e **readDirection** são as responsáveis pela leitura e validação dos *inputs*. O primeiro recebe os valores da linha e da coluna onde se encontra a peça que vai realizar o movimento e irá confirmar se existe uma peça do jogador na posição indicada. Caso não haja ou a peça não tiver movimentos válidos, irão voltar a ser pedidas as coordenadas ao jogador até que a validação seja realizada com sucesso. Já o segundo irá receber a lista de movimentos válidos para a peça indicada pelo jogador e imprimir todas as direções possíveis, indicando se esta é válida ou não. Tal como no predicado anterior, caso o *input* seja inválido, este processo irá repetir-se até a direção escolhida ser válida

```
Enter the row of the piece you want to move (1-5): d

Row is invalid. Try again.
Enter the row of the piece you want to move (1-5): 1
Enter the column of the piece you want to move (a-e): 1

Column is invalid. Try again.
Enter the column of the piece you want to move (a-e): a

Piece selected is invalid. Try again.

Enter the row of the piece you want to move (1-5): 2
Enter the column of the piece you want to move (a-e): c

Piece selected

DIRECTION

1 - North   2 - West   3 - East   4 - South
Northeast not valid   Northwest not valid   7 - Southeast   8 - Southwest
Enter the desired direction: 5

Direction is invalid. Try again.
Enter the desired direction: 1
```

Figure 6: Exemplo do processo realizado para a obtenção dos *inputs*

Após obtidos os *inputs* irá ser executado **findNewPosition** que irá devolver a linha e coluna da posição da peça após efetuado o movimento. Esta é um predicado recursiva que irá simular o movimento da peça uma posição de cada vez na direção indicada até encontrar outra peça ou a borda do tabuleiro. Encontrada a posição final da peça, será chamada o predicado **changePiece** duas vezes, uma para colocar a posição inicial da peça vazia, e outra para colocar a peça na posição final.

3.5 Final do Jogo

O predicado que verifica o final do jogo é o **game_over**. Existem cinco situações que levam ao final do jogo, sendo que em quatro delas um jogador é declarado vencedor e na outra o jogo termina num empate. As quatro situações em que existe um vencedor são quando um jogador alinha as suas peças na horizontal, na vertical, numa diagonal com orientação de nordeste para sudueste ou numa diagonal com orientação de noroeste para sudeste. Cada um destes casos tem um predicado próprio:

- **checkRow** - verifica se as peças estão alinhadas na horizontal, verificando se uma lista com as três peças é sublista de alguma das linhas do tabuleiro.
- **checkColumn** - verifica se as peças estão alinhadas na vertical, obtendo a posição da primeira peça, procurando de cima para baixo e da esquerda para a direita, e verificando se as duas posições imediatamente abaixo possuem uma peça da mesma cor.
- **checkDiagonalNESW** e **checkDiagonalNWSE** - verificam se as peças estão alinhadas diagonalmente utilizando uma estratégia semelhante à utilizada por **checkColumn**, verificando também as posições imediatamente abaixo, mas diagonalmente e não ortogonalmente.

```
% Checks if any row has a game ending condition
checkRow([H|T], Player) :-
    sublist([Player, Player, Player], H);
    checkRow(T, Player).

% Checks if any column has a game ending condition
checkColumn(Board, Player) :-
    getNthPiecePos(Board, Player, Nrow, Ncolumn, 1),
    Nrow2 is Nrow + 1, getPiece(Nrow2, Ncolumn, Board, Piece2), Piece2 = Player,
    Nrow3 is Nrow + 2, getPiece(Nrow3, Ncolumn, Board, Piece3), Piece3 = Player.

% Checks if any diagonal has a game ending condition (NW-SE orientation)
checkDiagonalNWSE(Board, Player) :-
    getNthPiecePos(Board, Player, Nrow, Ncolumn, 1),
    Nrow2 is Nrow + 1, Ncolumn2 is Ncolumn + 1,
    getPiece(Nrow2, Ncolumn2, Board, Piece2), Piece2 = Player,
    Nrow3 is Nrow + 2, Ncolumn3 is Ncolumn + 2,
    getPiece(Nrow3, Ncolumn3, Board, Piece3), Piece3 = Player.
```

```
% Checks if any diagonal has a game ending condition (NE-SW orientation)
checkDiagonalNESW(Board, Player) :-
    getNthPiecePos(Board, Player, Nrow, Ncolumn, 1),
    Nrow2 is Nrow + 1, Ncolumn2 is Ncolumn - 1,
    getPiece(Nrow2, Ncolumn2, Board, Piece2), Piece2 = Player,
    Nrow3 is Nrow + 2, Ncolumn3 is Ncolumn - 2,
    getPiece(Nrow3, Ncolumn3, Board, Piece3), Piece3 = Player.
```

Para ser possível a verificação do empate, o tabuleiro é guardado no final de cada turno pelo predicado **saveBoard** em **previousBoards**. O empate vai acontecer quando a disposição do tabuleiro ocorrer pela terceira vez, ou seja quando existirem dois tabuleiros guardados em **previousBoards** que têm a mesma disposição que o tabuleiro atual. Esta verificação é realizada por **checkDraw**.

```
% Checks if the same board occurred for the third time. If it did, then
% the game is declared a draw.
checkDraw(Board) :-
    previousBoards(N1, Board),
    previousBoards(N2, Board),
    N1 \= N2.

saveBoard(N, Board) :-
    assert(previousBoards(N, Board)).

previousBoards(N, Board).
```

Os predicados utilizados para a verificação do empate não se encontram todos no mesmo ficheiro. **saveBoard** e **previousBoards** encontram-se em *board.pl* enquanto está declarado en *game.pl*.

Terminado o jogo, os quadros guardados serão apagados e o programa voltará ao menu principal.

3.6 Avaliação do Tabuleiro

O tabuleiro é avaliado por **value**. Esta avaliação pode ter 5 valores distintos:

- 3: O tabuleiro contém uma vitoria do jogador;
- 2: O tabuleiro contém uma vitoria eminente do jogador na jogada seguinte;
- 1: O tabuleiro contém duas peças do jogador seguidas. Desta forma, jogadas que tentem juntar a terceira peça a outras 2 que estejam juntas são priorizadas e a separação das peças é evitada;
- 0: O tabuleiro é neutro, não ocorre nenhum dos outros casos;
- -1: O tabuleiro contém uma vitoria do adversário. Este estado permite dar prioridade a jogadas defensivas que possibilitem o bloqueio a vitoria do adversário.

Os casos são avaliados por uma ordem predefinida, consoante a prioridade de cada um. A ordem de avaliação é a seguinte:

3 -1 2 1 0

```
value(Board, 'b', Value) :-
    (checkWin(Board, 'b'), Value is 3);
    (simulateBotWin(Board, 2, 'w'), Value is -1);
    (simulateBotWin(Board, 2, 'b'), Value is 2);
    (check2(Board, 'b'), Value is 1);
    Value is 0.

value(Board, 'w', Value) :-
    (checkWin(Board, 'w'), Value is 3);
    (simulateBotWin(Board, 2, 'b'), Value is -1);
    (simulateBotWin(Board, 2, 'w'), Value is 2);
    (check2(Board, 'w'), Value is 1);
    Value is 0.
```

3.7 Jogada do Computador

O computador pode ter dois niveis de dificuldade, *Easy* e *Hard*. Para executar a jogada do computador é usado o predicado **choose_move** que recebe como parametros o tabuleiro, o tabuleiro com as alterações, a dificuldade (1 para *Easy*, 2 para *Hard*) e a cor do jogador (branco('w') ou preto('b')).

O computador em modo *Easy* seleciona uma peça e uma direção aleatória, tendo sempre em atenção se este movimento é válido. Este modo não tem qualquer tipo de inteligência fazendo todas as suas jogadas sem qualquer critério.

```
choose_move(InBoard, OutBoard, 1, Color) :-
    random(1, 4, Piece),
    getNthPiecePos(InBoard, Color, Nrow, Ncolumn, Piece),
    valid_moves(InBoard, Nrow, Ncolumn, Color, ListOfMoves),
    randomDirection(ListOfMoves, Direction),
    findNewPosition(Direction, InBoard, Nrow, Ncolumn, OutRow, OutColumn),
    changePiece(InBoard, Ncolumn, Nrow, 'x', IntBoard),
    changePiece(IntBoard, OutColumn, OutRow, Color, OutBoard).
```

O computador em modo Hard testa todas as direções validas para cada peça, obtendo a direção com maior valor para cada uma delas (no caso de haver várias direções com o mesmo valor é sorteado qual delas será escolhida). Após a obtenção dos valores mais altos de cada peça, estes serão comparados para selecionar a peça tem o movimento mais valioso (tal como nas direções, no caso de haver duas ou até três peças com valores máximos iguais, a peça a ser movida é escolhida a sorte).

```
choose_move(InBoard, OutBoard, 2, Color) :-
    getBestPlay(1, InBoard, Color, Row1, Column1, Direction1, Value1),
    getBestPlay(2, InBoard, Color, Row2, Column2, Direction2, Value2),
    getBestPlay(3, InBoard, Color, Row3, Column3, Direction3, Value3),
    checkBestPiece(Value1, Value2, Value3, Piece),
    parse(Piece, Row, Row1, Row2, Row3, Column, Column1, Column2, Column3,
    Direction, Direction1, Direction2, Direction3),
    findNewPosition(Direction, InBoard, Row, Column, OutRow, OutColumn),
    changePiece(InBoard, Column, Row, 'x', IntBoard),
    changePiece(IntBoard, OutColumn, OutRow, Color, OutBoard).
```

4 Conclusões

Com a realização deste projeto, pudemos aprender a trabalhar com *Prolog*, visto que esta foi a primeira vez que tivemos contacto com esta linguagem. Esta é uma linguagem bastanta única, o que levou a que passássemos muito tempo a discutir qual seria a maneira mais adequada de implementar alguns predicados, que seriam mais facéis de desenvolver em linguagens com as quais estamos mais familiarizados.

Em relação ao trabalho, pensamos que os objetivos foram cumpridos. Os três modos de jogo estão funcionais, sendo que o computador tem duas dificuldades tal como referido no enunciado. Devido à falta de tempo, houve alguns aspetos que talvez não tenham ficado tão bem como pretendíamos, nomeadamente a eficiência de alguns predicados e também a avaliação do estado do tabuleiro.

Concluindo, com o desenvolvimento deste trabalho tivemos a oportunidade de aprender esta linguagem única e complexa que nos obrigou a abordar os problemas que nos foram surgindo de maneira diferente e que nos pode vir a ser muito útil na nossa carreira professional.

References

[1] Jan Kristian Haugland. `https://boardgamegeek.com/boardgamedesigner/1364/jan-kristian-haugland`.

[2] Neutreeko. `http://www.neutreeko.net/neutreeko.htm`.