Boco

Programação em Lógica

Relatório final

Grupo Boco_1

Carlos Eduardo da Nova Duarte (up201708804) up201708804@fe.up.pt

Sofia de Araújo Lajes (up201704066) up201704066@fe.up.pt

15/11/2019

Índice

1. Introdução

Com este projeto, é pretendida a implementação de um jogo de tabuleiro para dois jogadores, utilizando linguagem Prolog. É também pretendido que seja criado um bot para jogar este jogo, tanto contra um humano quanto outro bot. Duas pessoas devem também poder jogar uma contra a outra.

A interação será realizada, inicialmente, a partir de uma linha de comandos, sendo a representação do jogo feita em ASCII art. Posteriormente será adicionada uma interface gráfica em 3D, que será desenvolvida na UC LAIG.

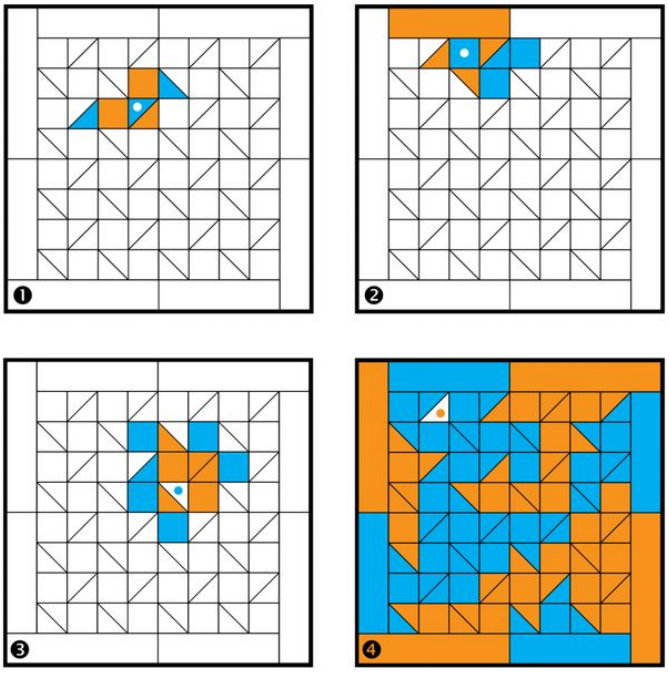
2. O jogo Boco

2.1. História

O jogo foi lançado pela equipa Binary Cocoa em 2019. Devido à recente data de lançamento, ao pouco reconhecimento do jogo e à ausência de mais informações acerca do mesmo, não conseguimos detalhar com maior pormenor as suas origens.

2.2. Tabuleiro de jogo

O tabuleiro é constituído por quadrados, triângulos (resultantes da divisão de um quadrado) e retângulos (dispostos nos limites do tabuleiro, cercando as outras peças).



2.3. Como jogar

Sejam Jogador 1 e Jogador 2 os dois jogadores.

- Inicialmente, o Jogador 1 seleciona qualquer um dos triângulos do tabuleiro de jogo.
- Após o Jogador 1 realizar a sua jogada, o Jogador 2 pode selecionar qualquer peça adjacente à colocada pelo Jogador 1 - podendo ser o triângulo adjacente ao selecionado (e que juntamente com este forma um quadrado), ou outro tipo de peça adjacente a um dos lados do triângulo.
- Os jogadores escolhem, alternadamente, qualquer peça do tabuleiro de jogo, seguindo sempre a regra de partilhar uma aresta com qualquer um dos espaços já selecionados (quer por si, quer pelo oponente). Este processo repete-se até que o jogo termine.

2.4 Como vencer

Um jogador deve rodear completamente pelo menos uma das peças do oponente.

É possível o jogo acabar empatado caso um dos jogadores selecione uma peça em que ambos ganhem e percam.

Um espaço que se encontre na fronteira do tabuleiro não pode ser cercado. Desta forma, todos os espaços de uma cor que estejam conectados a um espaço da mesma cor na fronteira, não podem ser cercados.

3. Lógica do jogo

3.0. SWI-Prolog

Decidimos desenvolver o nosso programa utilizando a implementação “SWI-Prolog”, uma vez que esta dispõe uma interface com cores e símbolos, permitindo uma jogabilidade mais apelativa. A experiência de desenvolvimento em Linux também foi um dos pontos que nos levou a optar por este Prolog.

3.1. Representação do estado do jogo

Foi decidido representar o jogo, internamente, através de uma lista com 10 listas (10 linhas), havendo, em cada, 10 listas (correspondente a 10 colunas) que representam peças.

As peças são representadas da seguinte forma:

- Retângulo: ['R', n, w], tal que 'R' é o átomo identificador de um retângulo, n é o número do retângulo (há 8) e w é o dono deste. O valor w adquire o valor nill se a célula estiver desocupada (se nenhum jogador a tiver selecionado), p1 se ocupada pelo jogador 1 e p2 pelo jogador 2.
- Quadrado: ['Q', w], tal que 'Q’ é o átomo identificador de um quadrado e w é o dono deste, seguindo as mesmas regras explicadas no retângulo.
- Triângulo: ['T', i, w1, w2], tal que 'T’ é o átomo identificador de uma célula com dois triângulos, i é o identificador da orientação dos triângulos (up se for do canto inferior esquerdo para o superior direito, ou down se for do canto superior esquerdo para o

canto inferior direito). Uma vez que dois triângulos no mesmo quadrado podem ter donos diferentes, w1 refere-se ao dono do triângulo esquerdo e w2 ao do triângulo direito.

Note-se que optamos por juntar os pares de triângulos numa única célula, de maneira a mantermos o número de linhas e colunas igual a 10 (de maneira a que quem estiver a analisar o estado de jogo consiga identificar facilmente de que célula se trata, e também para que não haja mais células do que colunas no tabuleiro).

É de realçar também que o valor do "dono" da peça influencia a cor com que será impressa (Player 1 (identificado pelo átomo p1) terá a cor vermelha e Player 2 (identificado pelo átomo p2) a cor azul).

Vejamos um exemplo de um estado inicial do jogo, onde o tabuleiro ainda está completamente vazio:

```
initialBoard([
    [['R',8,nil], ['R',1,nil],['R',1,nil],['R',1,nil],['R',1,nil], ['R',2,nil],['R',2,nil],['R',2,nil],['R',2,nil],['R',2,nil]],
    [['R',8,nil], ['Q',1,nil], ['T',1,dw,nil,nil], ['Q',2,nil], ['T',2,dw,nil,nil], ['Q',3,nil], ['T',3,dw,nil,nil], ['Q',4,nil], ['T',4,dw,nil,nil], ['R',3,nil]],
    [['R',8,nil], ['T',5,up,nil,nil], ['Q',5,nil], ['T',6,up,nil,nil], ['Q',6,nil], ['T',7,up,nil,nil], ['Q',7,nil], ['T',8,up,nil,nil], ['Q',8,nil], ['R',3,nil]],
    [['R',8,nil], ['Q',9,nil], ['T',9,dw,nil,nil], ['Q',10,nil], ['T',10,dw,nil,nil], ['Q',11,nil], ['T',11,dw,nil,nil], ['Q',12,nil], ['T',12,dw,nil,nil], ['R',3,nil]],
    [['R',8,nil], ['T',13,up,nil,nil], ['Q',13,nil], ['T',14,up,nil,nil], ['Q',14,nil], ['T',15,up,nil,nil], ['Q',15,nil], ['T',16,up,nil,nil], ['Q',16,nil], ['R',3,nil]],
    [['R',7,nil], ['Q',17,nil], ['T',17,dw,nil,nil], ['Q',18,nil], ['T',18,dw,nil,nil], ['Q',19,nil], ['T',19,dw,nil,nil], ['Q',20,nil], ['T',20,dw,nil,nil], ['R',4,nil]],
    [['R',7,nil], ['T',21,up,nil,nil], ['Q',21,nil], ['T',22,up,nil,nil], ['Q',22,nil], ['T',23,up,nil,nil], ['Q',23,nil], ['T',24,up,nil,nil], ['Q',24,nil], ['R',4,nil]],
    [['R',7,nil], ['Q',25,nil], ['T',25,dw,nil,nil], ['Q',26,nil], ['T',26,dw,nil,nil], ['Q',27,nil], ['T',27,dw,nil,nil], ['Q',28,nil], ['T',28,dw,nil,nil], ['R',4,nil]],
    [['R',7,nil], ['T',29,up,nil,nil], ['Q',29,nil], ['T',30,up,nil,nil], ['Q',30,nil], ['T',31,up,nil,nil], ['Q',31,nil], ['T',32,up,nil,nil], ['Q',32,nil], ['R',4,nil]],
    [['R',6,nil],['R',6,nil],['R',6,nil],['R',6,nil],['R',6,nil], ['R',5,nil],['R',5,nil],['R',5,nil],['R',5,nil], ['R',4,nil]]
]).
```

Após algumas jogadas, o tabuleiro irá evoluir para o estado seguinte:

```
middleBoard([
    [['R',8,nil], ['R',1,nil],['R',1,nil],['R',1,nil],['R',1,nil], ['R',2,nil],['R',2,nil],['R',2,nil],['R',2,nil],['R',2,nil]],
    [['R',8,nil], ['Q',1,nil], ['T',1,dw,nil,nil], ['Q',2,nil], ['T',2,dw,nil,nil], ['Q',3,nil], ['T',3,dw,nil,nil], ['Q',4,nil], ['T',4,dw,nil,nil], ['R',3,nil]],
    [['R',8,nil], ['T',5,up,nil,nil], ['Q',5,nil], ['T',6,up,nil,nil], ['Q',6,nil], ['T',7,up,nil,nil], ['Q',7,nil], ['T',8,up,nil,nil], ['Q',8,nil], ['R',3,nil]],
    [['R',8,nil], ['Q',9,nil], ['T',9,dw,nil,nil], ['Q',10,p1], ['T',10,dw,p2,p1], ['Q',11,nil], ['T',11,dw,nil,nil], ['Q',12,nil], ['T',12,dw,nil,nil], ['R',3,nil]],
    [['R',8,nil], ['T',13,up,nil,nil], ['Q',13,nil], ['T',14,up,p2,p1], ['Q',14,nil], ['T',15,up,nil,nil], ['Q',15,nil], ['T',16,up,nil,nil], ['Q',16,nil], ['R',3,nil]],
    [['R',7,nil], ['Q',17,nil], ['T',17,dw,nil,nil], ['Q',18,p2], ['T',18,dw,p1,p2], ['Q',19,nil], ['T',19,dw,nil,nil], ['Q',20,nil], ['T',20,dw,nil,nil], ['R',4,nil]],
    [['R',7,nil], ['T',21,up,nil,nil], ['Q',21,nil], ['T',22,up,nil,nil], ['Q',22,p1], ['T',23,up,nil,nil], ['Q',23,nil], ['T',24,up,nil,nil], ['Q',24,nil], ['R',4,nil]],
    [['R',7,nil], ['Q',25,nil], ['T',25,dw,nil,nil], ['Q',26,nil], ['T',26,dw,nil,nil], ['Q',27,nil], ['T',27,dw,nil,nil], ['Q',28,nil], ['T',28,dw,nil,nil], ['R',4,nil]],
    [['R',7,nil], ['T',29,up,nil,nil], ['Q',29,nil], ['T',30,up,nil,nil], ['Q',30,nil], ['T',31,up,nil,nil], ['Q',31,nil], ['T',32,up,nil,nil], ['Q',32,nil], ['R',4,nil]],
    [['R',6,nil],['R',6,nil],['R',6,nil],['R',6,nil],['R',6,nil], ['R',5,nil],['R',5,nil],['R',5,nil],['R',5,nil], ['R',4,nil]]
]).
```

E, por fim, terminará com a vitória do Jogador 2.

```
endBoard([
    [['R',8,nil], ['R',1,nil],['R',1,nil],['R',1,nil],['R',1,nil], ['R',2,nil],['R',2,nil],['R',2,nil],['R',2,nil],['R',2,nil]],
    [['R',8,nil], ['Q',1,nil], ['T',1,dw,nil,nil], ['Q',2,nil], ['T',2,dw,nil,nil], ['Q',3,nil], ['T',3,dw,nil,nil], ['Q',4,nil], ['T',4,dw,nil,nil], ['R',3,nil]],
    [['R',8,nil], ['T',5,up,nil,nil], ['Q',5,nil], ['T',6,up,nil,p1], ['Q',6,p2], ['T',7,up,p1,p2], ['Q',7,p1], ['T',8,up,nil,nil], ['Q',8,nil], ['R',3,nil]],
    [['R',8,nil], ['Q',9,nil], ['T',9,dw,nil,p1], ['Q',10,p1], ['T',10,dw,p2,p1], ['Q',11,nil], ['T',11,dw,p1,p2], ['Q',12,nil], ['T',12,dw,nil,nil], ['R',3,nil]],
    [['R',8,nil], ['T',13,up,nil,nil], ['Q',13,nil], ['T',14,up,p2,p1], ['Q',14,nil], ['T',15,up,nil,nil], ['Q',15,p2], ['T',16,up,p1,p2], ['Q',16,nil], ['R',3,nil]],
    [['R',7,nil], ['Q',17,nil], ['T',17,dw,nil,nil], ['Q',18,p2], ['T',18,dw,p1,p2], ['Q',19,nil], ['T',19,dw,nil,p2], ['Q',20,p1], ['T',20,dw,p2,nil], ['R',4,nil]],
    [['R',7,nil], ['T',21,up,nil,nil], ['Q',21,nil], ['T',22,up,nil,nil], ['Q',22,p1], ['T',23,up,nil,nil], ['Q',23,nil], ['T',24,up,p2,nil], ['Q',24,nil], ['R',4,nil]],
    [['R',7,nil], ['Q',25,nil], ['T',25,dw,nil,nil], ['Q',26,nil], ['T',26,dw,nil,nil], ['Q',27,nil], ['T',27,dw,nil,nil], ['Q',28,nil], ['T',28,dw,nil,nil], ['R',4,nil]],
    [['R',7,nil], ['T',29,up,nil,nil], ['Q',29,nil], ['T',30,up,nil,nil], ['Q',30,nil], ['T',31,up,nil,nil], ['Q',31,nil], ['T',32,up,nil,nil], ['Q',32,nil], ['R',4,nil]],
    [['R',6,nil],['R',6,nil],['R',6,nil],['R',6,nil],['R',6,nil], ['R',5,nil],['R',5,nil],['R',5,nil],['R',5,nil], ['R',4,nil]]
]).
```

Estes estados de jogo serão os mesmos que serão usados, na secção seguinte, para mostrar o tabuleiro ao longo das diversas jogadas de uma partida.

3.2. Visualização do tabuleiro

Desde que o jogo inicia até ao seu fim, o tabuleiro tem que ser atualizado logo após cada jogada. Para isso, o predicado display_game(Board) é executado.

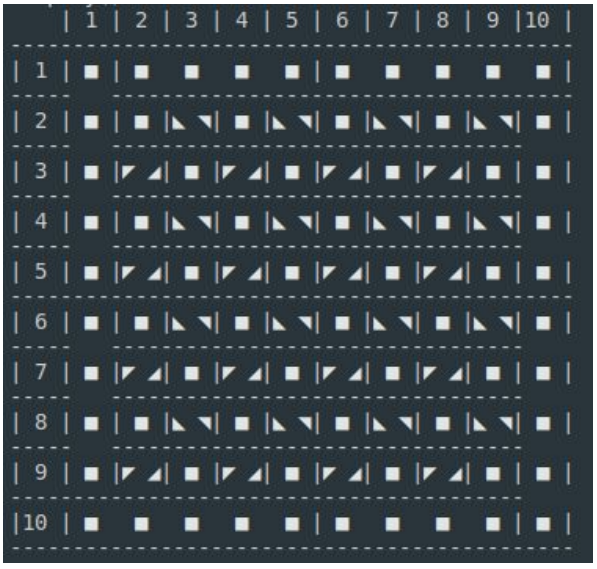
Antes do conteúdo do tabuleiro ser impresso, o jogo escreve na consola uma linha com a indicação de cada coluna do jogo. À medida que as linhas são escritas, o mesmo é feito para cada linha do tabuleiro.

A impressão do tabuleiro é feita linha a linha, coluna a coluna. Inicialmente é impressa uma sequência de hífens que serve para separar visualmente cada linha do tabuleiro. Posteriormente, é impressa a linha correspondente ao índice que ocupa no tabuleiro - de 1 a 10. É de notar que a posição dos retângulos envolventes e a ordem pela

qual aparecem alternadamente os quadrados e os triângulos no tabuleiro influenciam a impressão de cada linha na consola, havendo para isso predicados especiais que são chamados e executados.

Esta função display_game(Board, Player) recebe um estado de jogo e o nome de um jogador, e mostra o tabuleiro correspondente no ecrã.

Na imagem seguinte é possível visualizar o tabuleiro no início do jogo (em que nenhum jogador escolheu uma célula):



As células encontram-se todas a branco, já que ainda nenhuma jogada foi executada.

Após algumas jogadas, o tabuleiro (e consequentemente o estado) do jogo poderá

evoluir para, por exemplo, o que é representado na próxima imagem:

A cor vermelha representa o jogador 1, e a cor azul representa o jogador 2.

Vejamos agora uma continuação do tabuleiro apresentado anteriormente, no qual o Jogador 2 (representado pela cor azul) consegue rodear completamente o Jogador 1, sendo portanto o vencedor da partida:

3.3. Lista de jogadas válidas

De seguida, segue-se o código da função **valid_moves(+Board, -PlayList)**, que permite obter uma lista com todas as jogadas válidas:

```
valid_moves(Board, PlayList) :-
    findall(X1-Y1,
            valid_cell(Board, X1, Y1),
            L1),
    findall(X2-Y2-left,
            valid_cell(Board, X2, Y2, left),
            L2),
    findall(X3-Y3-right,
            valid_cell(Board, X3, Y3, right),
            L3),
    append(L1, L2, AuxL1),
    append(AuxL1, L3, PlayList).
```

A obtenção desta lista é feita recorrendo ao predicado findall. Em cada findall, variamos as coordenadas X e Y (dentro dos valores permitidos - entre 1 e 10, inclusive), e verificamos se a célula localizada nessas mesmas coordenadas é válida ou não. Se for, é adicionada à lista.

Note-se que estamos a executar 3 vezes o predicado findall, já que também temos que contemplar os casos em que a célula selecionada é um triângulo, e esse triângulo pode ser o lado esquerdo ou direito de uma célula.

Para uma célula ser válida, tem que obedecer às seguintes condições:

- A célula tem que estar vazia (ou seja, não pode ter sido selecionada antes por qualquer jogador);

- Tem que existir, adjacente à célula selecionada (cima, baixo, esquerda ou direita), uma outra célula que já tenha sido selecionada por qualquer um dos jogadores.

A verificação das células adjacentes tem que ser subdividida em predicados que detetem se uma célula é um triângulo ou não (já que os triângulos possuem um componente Side, ao contrário das outras células). De seguida, acontece uma nova decomposição dos predicados, os quais verificam individualmente a célula que se encontra em cima, em baixo, à esquerda e à direita. A partir de cada um destes predicados, subdividimos novamente noutros predicados que têm em conta a célula que foi escolhida, já que dependendo dessa seleção, pode estar adjacente à célula um quadrado, um retângulo ou um triângulo do tipo up ou down.

A quantidade considerável de células diferentes que compõem o jogo, bem como a heterogeneidade do tabuleiro, levaram-nos a optar por esta abordagem.

```
valid_cell(Board, Row, Column) :-
    valid_coordinate(Row),
    valid_coordinate(Column),
    cell_empty(Board, Row, Column, _),
    (   check_above(Board, Row, Column)
    ;   check_below(Board, Row, Column)
    ;   check_left(Board, Row, Column)
    ;   check_right(Board, Row, Column)
    ).
```

O funcionamento dos predicados **check_above**, **check_below**, **check_left** e **check_right** é bastante parecido. O seu processamento apenas varia dependendo das possibilidades de células existentes adjacentes à original. Por exemplo: na execução da função **check_above**, se a célula selecionada pelo utilizador for um quadrado, então por cima dessa célula poderá existir um retângulo, um triângulo do tipo up ou um triângulo do tipo down. Por sua vez, se a célula selecionada for um retângulo, a célula acima poderá apenas ser um quadrado, um triângulo do tipo down ou outro retângulo. Sendo os predicados, portanto, bastante similares, analisemos o funcionamento do predicado **check_above**, se a célula selecionada for um quadrado. Caso a célula seja um quadrado, haverão 3 casos possíveis:

- A célula acima é um retângulo;
- A célula acima é um triângulo up;
- A célula acima é um triângulo down.

Analisemos o primeiro caso:

```
%If selected cell is square, checks if above cell is rectangle
check_above(Board, Row, Column) :-
    NewRow is Row-1,
    valid_coordinate(NewRow),
    get_cell(Board, Row, Column, Cell),
    nth1(1, Cell, 'Q'),
    get_cell(Board, NewRow, Column, AboveCell),
    nth1(1, AboveCell, 'R'),
    cell_occupied(Board, NewRow, Column).
```

Sendo que estamos a analisar a célula que se encontra imediatamente acima à célula selecionada, que se encontra na linha Row e na coluna Column, então a célula

adjacente estará na linha Row-1 e coluna Column. Ora, as coordenadas de uma célula têm obrigatoriamente que tomar valores entre 1 e 10, inclusive. Para garantir isso, o predicado **valid_coordinate** faz essa verificação.

Posteriormente, o predicado **get_cell** vai buscar a estrutura interna da célula que ocupa a posição Row-1 e Column, verifica se a célula selecionada pelo utilizador é um quadrado e se a célula acima é um retângulo, e por fim executa verifica se a célula acima está ocupada (é obrigatório que assim seja, para que seja possível jogar na célula escolhida pelo utilizador). No caso específico de a célula acima ser um retângulo, o predicado **cell_occupied** é o seguinte:

```
%Checks if a square cell is occupied
cell_occupied(Board, Row, Column) :-
    valid_coordinate(Row),
    valid_coordinate(Column),
    get_cell(Board, Row, Column, Cell),
    nth1(1, Cell, 'Q'),
    (   nth1(2, Cell, p1)
    ;   nth1(2, Cell, p2)
    ).
```

O seu funcionamento é bastante simples: verifica se a linha e coluna estão dentro dos valores permitidos, e posteriormente verifica se, na estrutura interna de um quadrado, o campo que indica quem é o dono dessa célula está preenchido por p1 ou p2 - que representam o jogador 1 e o jogador 2.

Como já foi referido acima, o algoritmo de funcionamento dos outros predicados de verificação de células adjacentes, bem como os de ocupação de uma célula, são análogos a estes, com as devidas diferenças para acomodar outros tipos de células (triângulos, quadrados e retângulos).

3.4. Execução de jogadas

A execução de uma jogada é realizada pelo predicado **move(+Board, +Player, +GameMode)**, cujos parâmetros são, tal como o nome indica, o tabuleiro de jogo, o jogador que irá executar a jogada (p1 ou p2), e o modo de jogo selecionado.

O modo de jogo pode ser um dos seguintes:

- 1: Humano vs Humano;
- 2: Humano vs PC;
- 3: PC vs Humano;
- 4: PC vs PC.

NOTA: no código, referenciamos um jogador humano como Player, e a inteligência artificial como PC.

Subdividimos os predicados em dois conjuntos:

- O primeiro engloba os predicados **first_move(Board, Player, GameMode)**, que realizam a primeira jogada do jogo por parte de um humano ou do PC. A primeira jogada é diferente das restantes porque não obedece às regras de

validação de uma célula, ou seja, não há qualquer restrição acerca do posicionamento da primeira célula selecionada.

- O segundo é o conjunto dos predicados **move(Board, Player, GameMode)**, que processam todas as restantes jogadas.

Dada a semelhança do processamento destes dois conjuntos, analise-se o funcionamento dos dois predicados **move**:

```
%Player vs Player - lets Player make a move
%Player vs PC - lets Player make a move
%Lets a player make a move if the following scenarios are
%valid:
%-Game mode is Player vs Player
%-Game mode is Player vs PC and it's the Player's turn to play
%-Game mode is PC vs Player and it's the Player's turn to play
move(Board, Player, GameMode) :-
    (   GameMode=1
    ;   GameMode=2,
        Player=p1
    ;   GameMode=3,
        Player=p2
    ),
    valid_play(Board, Player, Row, Column, Side),
    update_board(Player, Row, Column, Side, Board, NewBoard),
    cls(),
    print_board(NewBoard, 1),
    ite(Player==p1, NewPlayer=p2, NewPlayer=p1),
    sleep(1),
    move(NewBoard, NewPlayer, GameMode).
```

O predicado acima realiza vários procedimentos. Primeiro, verifica se é o humano que vai realizar a jogada. Isto apenas acontece se o modo de jogo for o 1 - Humano vs Humano (neste caso, é sempre um humano a realizar uma jogada), - se o modo de jogo for o 2 - Humano vs PC - e for o humano a jogar, ou se o modo de jogo for o 3 - PC vs Humano - e for o humano a jogar.

Caso isto suceda, então a jogada será validada através da execução do predicado **valid_play(Board, Player, Row, Column, Side)**. Este predicado começa por pesquisar quais são as jogadas válidas possíveis no tabuleiro dado o estado do tabuleiro atual, e imprime-as no ecrã. Posteriormente, o predicado pede repetidamente ao utilizador que insira as coordenadas da peça que deseja escolher, até que as células sejam válidas (desta forma garante-se que as coordenadas são válidas, e permite-se ao utilizador reinserir as coordenadas, caso a anterior insersão tenha sido inválida). Após receber as coordenadas, verifica se estas pertencem à lista de jogadas válidas. Caso seja esse o caso, então a jogada é válida. Senão, o utilizador terá que reinserir as coordenadas (tendo estas que existir na lista de jogadas impressas na consola).

```
%Checks if a play inserted by the user is valid.
%To be valid, the coordinates of the cell selected
%by the user need to exist in the list of available plays
%called PlayList.
valid_play(Board, Player, Row, Column, Side) :-
    repeat,
    valid_moves(Board, PlayList),
    writeln("Available plays:"),
    print_valid_moves(PlayList),
    ask_move(Board, [Row, Column, Side], Player),!,
    (member(Row-Column, PlayList);member(Row-Column-Side, PlayList)).
```

Após a jogada ter sido escolhida, é necessário atualizar o tabuleiro de jogo. Para isso, é chamado o predicado **update_board(Player, Row, Column, Side, Board, NewBoard)**, que realizará várias verificações até, por fim, atualizar o tabuleiro.

Vejamos o seu funcionamento:

1. Obtém a célula que tem como coordenadas (Row, Column).
2. Verifica se a célula é um retângulo. Se for, então guarda em List as posições de todos os quadrados que compõe esse retângulo.
3. Se o lado for válido (left ou right), então é porque a célula selecionada é um triângulo, e procede à atualização dessa célula. Caso seja inválido, executa outro predicado que atualiza a célula consoante seja um quadrado ou um retângulo.

Posteriormente, o ecrã é limpo, de maneira a apenas estar visível o tabuleiro mais recente. Segue-se a impressão do novo tabuleiro de jogo, e executa-se novamente a função, mas desta vez o controlo sobre a escolha da jogada recai sobre o oponente do jogador que acabou de selecionar uma célula.

Esse caso é contemplado pelo seguinte predicado:

```
%Player vs PC - lets PC make a move
%PC vs Player - lets PC make a move
%Lets a PC make a move if the following scenarios are
%valid:
%-Game mode is Player vs PC
%-Game mode is PC vs Player and it's the PC's turn to play
%-Game mode is PC vs PC
move(Board, Player, GameMode) :-
    (   GameMode=2,
        Player=p2
    ;   GameMode=3,
        Player=p1
    ;   GameMode=4
    ),
    choose_move(Board, Row, Column, Side),
    update_board(Player, Row, Column, Side, Board, NewBoard),
    cls(),
    print_board(NewBoard, 1),
    ite(Player==p1, NewPlayer=p2, NewPlayer=p1),
    sleep(1),
    move(NewBoard, NewPlayer, GameMode).
```

Como é possível constatar, é bastante similar ao anterior, mas contrariamente a este, apenas executa quando o modo de jogo é o 2 (Humano vs PC) e é o PC a jogar, o 3 (PC vs Humano) e é o PC a jogar, ou o 4 (PC vs PC).

Uma outra diferença é a escolha da célula: como esta escolha é da responsabilidade do PC, é necessária uma maneira de o permitir escolher. O predicado que executa este procedimento (**choose_move**) será abordado na secção **3.7. Jogada do Computador**.

Após a escolha da célula, o predicado é em tudo similar ao apresentado anteriormente.

3.5 Final do Jogo

A condição de finalização do jogo é uma das partes mais complexas do projeto. Todo o processo dedicado a este módulo encontra-se no ficheiro *endgame.pl*.

O predicado que chama esta verificação é o **game_over(Board, Row, Col, Side, Player, Finish)**, tendo como argumentos, por esta ordem, o tabuleiro, as coordenadas Linha, Coluna da peça selecionada e, se nas coordenadas definidas estiver uma célula com dois triângulos, um identificador (left ou right) de qual dos triângulos, o jogador que fez a jogada, , e um sinalizador Finish que indica se o jogo deve parar ou continuar.

Criámos uma estrutura 'Coordinate', que possui a informação de uma peça (estruturas apresentadas no ponto 3.1 deste documento) e ainda a linha e coluna onde se encontra: [*'T', ROW, COLUMN, N, ORIENTATION OF TRIANGLE DIAGONAL, PLAYER]* (para triângulos), *['Q', ROW, COLUMN, PLAYER]* (para quadrados) e *['R', ROW, COLUMN, ID, PLAYER]* (para retângulos); Para definir estas coordenadas é necessário ter predicados com o mesmo propósito mas por métodos diferentes.

```
%CELL WAS PLAYED, LETS CHECK IF GAME IS OVER

%CellInfo contains ['T', ROW, COLUMN, N, ORIENTATION OF TRIANGLE DIAGONAL, PLAYER] for triangles
game_over(Board, ['T'|T], Row, Col, Side, Player, Finish):-
    nth1(1, T, Orientation),
    convertSide(Side,N),
    startVerification(Board, ['T', Row, Col, N, Orientation, Player], Player, [], Finish),
    !.

convertSide(left,1).
convertSide(right,2).

%CellInfo contains ['Q', ROW, COLUMN, PLAYER] for squares
game_over(Board, ['Q'|_], Row, Col, _, Player, Finish):-
    startVerification(Board, ['Q', Row, Col, Player], Player, [], Finish),
    !.

%CellInfo contains ['R', ROW, COLUMN, ID, PLAYER] for squares
game_over(Board, ['R'|T], Row, Col, _, Player, Finish):-
    nth1(1, T, ID),
    startVerification(Board, ['R', Row, Col, ID, Player], Player, [], Finish),
    !.
```

Pode terminar com um dos jogadores a ganhar ou com um empate, portanto, o predicado *startVerification(Board, Coordinates, Player, Finish)*, que recebe um tabuleiro, as coordenadas da peça jogada, o jogador que a jogou e o sinalizador Finish. Este receberá os

resultados dos predicados ‘playerLost’ e ‘playerWon’ através dos últimos parâmetros destes, e decidirá qual o vencedor (ou se há empate) (e por conseguinte, se Finish é verdadeiro (yes) ou falso (no)) e escrever na consola o resultado.

```
26  %START VERIFICATION
27  startVerification(Board, Coordinates, Player, Finish):-
28      playerLost(Board, Coordinates, Player, [], Lost), %VERIFY IF PLAYER LOST
29      playerWon(Board, Coordinates, Player, Won), %VERIFY IF PLAYER WIN
30      getResult(Player, Lost, Won, Finish). %FINAL JUDGEMENT
```

Esta verificação baseia-se na peça colocada, e não no tabuleiro inteiro, uma vez que a diferença entre a iteração atual e a seguinte é obrigatoriamente uma peça apenas. Desta forma, para a verificação de playerLost temos apenas de ter em consideração a peça selecionada e as peças do jogador que a jogou e que à volta desta se encontram, ou seja, temos de verificar se todo o conjunto de peças da mesma cor da jogada se encontram inteiramente rodeadas pelo inimigo; contudo, se esta fôr um retângulo, não pode estar rodeada, portanto, automaticamente, não pode perder.

```
82  playerLost(_, ['R'|_], _, _, notLost).
83
84  playerLost(Board, Coordinates, Player, VerifiedCells, Result):-
85      checkCell(Board, Coordinates, Player, VerifiedCells, Result).
```

Quanto a se ganhou, é mais complicado, para isto temos de verificar se alguma das peças que lhe é adjacente tem a cor do inimigo e, por conseguinte, se está rodeada por peças do jogador ou pertence a um aglomerado rodeado, assim como é feito no playerLost. Para tal, temos de recolher os ‘Neighbours’ consoante o tipo da peça jogada (através de getNeighbours), fazer uma filtragem das peças do jogador oposto (collectEnemyPieces) e chamar o predicado playerLost para cada uma delas, se existirem (areThereEnemies), se não houver, automaticamente, não ganhou.

```
49  %ENDGAME
50  playerWon(Board, Coordinates, Player, Result):-
51      nth1(1, Coordinates, Type),
52      getNeighbours(Board, Type, Coordinates, NeighboursList), %the way we get neighbours depends on the type of piece
53      oppositePlayer(Player, OppositePlayer), %get opposite player
54      collectEnemyPieces(OppositePlayer, NeighboursList, [], EnemyList), %get enemies adjacent to the main piece
55      areThereEnemies(Board, OppositePlayer, EnemyList, Result). % if there are enemies we verify each of them
56
```

```
234 %DISCOVER ENEMY PLAYER NEIGHBOUR PIECES
235 collectEnemyPieces(_, [], AuxiliaryList, AuxiliaryList):- !. %no more neighbour pieces to test, we finish
236
237 collectEnemyPieces(EnemyPlayer, [H|T], AuxiliaryList, EnemyList):-
238     getCellPlayer(H, EnemyPlayer), %confirm if player is enemy
239     append(AuxiliaryList, [H], NewAuxiliaryList),% add it to enemy list
240     collectEnemyPieces(EnemyPlayer, T, NewAuxiliaryList, EnemyList), %recursive call
241     !.
242
243 collectEnemyPieces(EnemyPlayer, [_|T], AuxiliaryList, EnemyList):-
244     collectEnemyPieces(EnemyPlayer, T, AuxiliaryList, EnemyList), %not an enemy piece, recursive call only
245     !.
246
```

Uma das grandes razões para a complexidade do jogo é precisamente a quantidade de peças diferentes e cuidados que temos de ter no seu tratamento. O tratamento de quadrados é constante, no entanto há 4 tipos de triângulos e 8 tipo de retângulos, e todos eles são específicos ao ponto de ser quase incontornável a criação de predicados diferentes para cada um deles. Esta foi uma condição que nos dificultou bastante durante todo o processo construtivo.

Por exemplo, na obtenção dos ‘Neighbours’, um quadrado terá de recolher, naturalmente, a peça acima, abaixo, à esquerda e à direita, mas um retângulo, dependendo do seu ID, terá de recolher 4 peças da coluna seguinte, ou da coluna anterior, ou a linha de cima ou da linha de baixo, bem como um triângulo à esquerda com orientação up tem de ir buscar o triângulo adjacente, a peça acima e à esquerda, originando então um grande número de predicados para um único objetivo.

```
114  % DEALING WITH A CELL'S NEIGHBOURS
115  getNeighbours(Board, 'Q', Coordinates, [CellUp, CellLeft, CellRight, CellDown]):-
116      nth1(2, Coordinates, Row),
117      nth1(3, Coordinates, Col),
118      getCellUp(Board, Row, Col, CellUp),
119      getCellLeft(Board, Row, Col, CellLeft),
120      getCellRight(Board, Row, Col, CellRight),
121      getCellDown(Board, Row, Col, CellDown).
122
123  % GET RECTANGLE NEIGHBOURS
124  getNeighbours(Board, 'R', Coordinates, Neighbours):-
125      nth1(4, Coordinates, ID),
126      getRectangleNeighbours(Board, ID, Neighbours).
127
128  %if the piece is a triangle, it is more complex to fetch its neighbours
129  getNeighbours(Board, 'T', Coordinates, NeighboursList):-
130      nth1(2, Coordinates, Row),
131      nth1(3, Coordinates, Col),
132      nth1(4, Coordinates, N),
133      nth1(5, Coordinates, Orientation),
134      getTriangleNeighbours(Board, Row, Col, Orientation, N, NeighboursList).
```

Outro predicado de extrema importância é *checkNeighbours*, este tem como objetivo a filtragem de da lista de neighbours recebida, inflectindo sobre esta os seguintes parâmetros:

- O player está automaticamente ilibado de perda caso tenha como adjacente uma peça sem dono ou um rectângulo da sua cor;
- A peça em questão está cercada (**surrounded**) caso não cumpra o primeiro ponto, não haja mais vizinhos a analisar e ainda assim a lista de novos vizinhos esteja vazia;
- É devolvida uma lista de novos vizinhos, cumprindo o requisito *searchVisitedCells*, a qual deverá devolver **‘no’**.

```
%CHECK NEIHBOURS ON 3 POSSIBLE OUTCOMES EACH: SAME COLOR, OPPOSITE COLOR OR NONE, ON NONE, WE IMMEDIATELY RETURN FALSE
checkNeighbours(_, _, [], [], _, lost):- !.  %no new neighbours  around nor unoccupied cells

checkNeighbours(_, _, [], AuxiliaryList, AuxiliaryList, continueAnalysing):- !.%list of new neighbours of same color is

checkNeighbours(Player, _, [H|_], _, _, notLost):- %the player is automaticly safe from losing if
    (getCellPlayer(H, nill)  % neighbour cell is not owned by any player
    ; (nth1(1, H, 'R') , getCellPlayer(H, Player)) % OR neighbour cell is a rectangle of the same color
    ),
    !.

checkNeighbours(Player, VerifiedCells, [H|T], AuxiliaryList, NeighboursToAnalyse, Result):-
    getCellPlayer(H, Player), %neighbour cell of same color
    searchVisitedCells(VerifiedCells, H, IsVisited),
    IsVisited == no, % it is not visited yet (not new neighbour)
    append(AuxiliaryList, [H], NewAuxiliaryList), % add to new neighbours list
    checkNeighbours(Player, VerifiedCells, T, NewAuxiliaryList, NeighboursToAnalyse, Result), %recursive call
    !.

checkNeighbours(Player, VerifiedCells, [_|T], AuxiliaryList, NeighboursToAnalyse, Result):- %all cases above fail
    checkNeighbours(Player, VerifiedCells, T, AuxiliaryList, NeighboursToAnalyse, Result), %recursive call
    !.
```

Todo este raciocínio culmina no predicado checkCell, onde são feitos os processos: adiciona peça à lista de peças verificadas; procura todos os vizinhos desta; gera uma lista de todos os vizinhos que devem ser verificados; chama analyseNeighbours, esta por sua vez determina se chegou ao fim ou não, ou seja, percorre a lista de vizinhos (se não fôr vazia), chamando recursivamente a checkCell para cada um destes, verificando os seus vizinhos, podendo originar vários nós-filhos; termina cada chamada recursiva se a procura de vizinhos e vizinhos destes chegar a um limite onde não há mais vizinhos (ou seja, está completamente rodeado) ou quando encontrar, numa das suas ramificações, um vizinho sem dono.

```
%CHECK CELL
checkCell(Board, Coordinates, Player, VerifiedCells, Result):-
    append(VerifiedCells, [Coordinates], NewVerifiedCells), %add coordinates of the cell we are testing to
    nth1(1, Coordinates, Type), %type means Q or T
    getNeighbours(Board, Type, Coordinates, NeighboursList), %get the cells neighbours
    checkNeighbours(Player, NewVerifiedCells, NeighboursList, [], NeighboursToAnalyse, IntermediateResult),
    analyseNeighbours(IntermediateResult, Board, Player, NewVerifiedCells, NeighboursToAnalyse, Result).


% ANALYSE NEIGHBOURS
analyseNeighbours(notLost, _, _, _, _, notLost):- !.

analyseNeighbours(_, _, _, _, [], _) :- !.

analyseNeighbours(_, Board, Player, VerifiedCells, [H|T], Result):-
    checkCell(Board, H, Player, VerifiedCells, IntermediateResult),
    analyseNeighbours(IntermediateResult, Board, Player, VerifiedCells, T, Result),
    !.
```

Contudo, apesar de todo o esforço, não foi possível integrar completamente esta abordagem, pelo que, em alguns casos, o algoritmo falha, e não conseguimos, ainda, detetar qual o erro, pelo que enviamos o código à parte e comentadas as partes onde se deve encontrar a chamada ao predicado no ficheiro game.pl. Esperamos poder demonstrar pessoalmente o algoritmo e tentar compreender as suas falhas.

3.6. Avaliação do tabuleiro

Devido à considerável complexidade da implementação deste jogo em PROLOG, e devido aos prazos estipulados para projetos de outras UCs, não tivemos tempo suficiente para implementar esta parte do trabalho prático. Preferimos utilizar o tempo que nos sobrou para eliminar o máximo de erros e bugs nas outras componentes, para termos o jogo o mais sólido e estável possível.

Com mais tempo, seríamos capazes de realizar esta implementação sem problemas. O algoritmo seria o seguinte:

1. Obter a lista de jogadas possíveis;
2. Para cada jogada possível, verificar quantas células inimigas a estão a rodear, e quantas células do próprio jogador também a rodeiam;
3. Selecionar a célula que maximiza o número de células adjacentes do próprio jogador, e que minimiza o número de células adjacentes do oponente. Caso haja algum empate, selecionar a primeira célula.

3.7. Jogada do computador

Foi implementado, através do predicado **choose_move**, uma inteligência artificial de nível básico.

```
choose_move(Board, Row, Column, Side) :-
    (valid_moves(Board, [Row-Column-Side|_]);valid_moves(Board, [Row-Column|_])).
```

Este predicado executa outro predicado chamado **valid_moves,** que lista todas as jogadas possíveis e seleciona a primeira delas. Repare-se na existência de duas chamadas a este predicado, separadas por um “ou” lógico: isto deve-se ao facto de algumas das células do tabuleiro serem triângulos, e por isso necessitarem do parâmetro “Side”, que faz a distinção entre as duas metades do quadrado que é composto por esses dois triângulos.

Por não termos conseguido ter tempo de implementar a funcionalidade de avaliação do tabuleiro devido à complexidade geral deste jogo, não nos foi possível implementar um nível mais avançado de inteligência artificial.

Contudo, a implementação deste nível mais avançado de AI seria bastante simples de desenvolver, tendo o predicado de valorização completamente funcional. Seria tão simples quanto a execução do seguinte predicado:

setof(V-X, Y, (valid_play(Board, X, Y, NewBoard), value(NewBoard, V)),

sendo que **value** seria a função de avaliação do tabuleiro, que daria uma valorização a cada célula.

4. Conclusões

Ambos achamos que este trabalho foi bastante útil para compreendermos melhor o funcionamento da linguagem PROLOG, e das diferenças que o paradigma declarativo possui comparativamente aos paradigmas imperativo e funcional.

Conseguimos implementar a maior parte dos objetivos que nos foram propostos, com exceção da verificação de final de jogo, avaliação do tabuleiro e nível 2 de inteligência artificial (sendo que o nível 2 de inteligência artificial seria implementado com apenas uma linha de código, tal como demonstrado na secção anterior, tendo funcional o predicado de valorização do tabuleiro).

Achamos o jogo bastante complexo de implementar (embora não o tenhamos achado quando o escolhemos, sendo que nessa altura ainda estávamos no início da UC e não tínhamos bem a perceção do que seria ou não um jogo complexo), e exigiu mesmo muitas horas da nossa parte para conseguirmos chegar ao resultado a que chegamos. Não foi por falta de empenho e dedicação que o jogo não ficou 100% acabado. Apelamos à compreensão do professor relativamente à dificuldade de implementação do jogo.

De qualquer maneira, estamos satisfeitos com o que desenvolvemos, e achamos que o nosso conhecimento da linguagem subiu consideravelmente graças a este projeto.

5. Bibliografia

- Board Game Geek
  https://boardgamegeek.com/boardgame/284633/boco

- Binary Cocoa (Criadores)
  https://binarycocoa.com/bocogame/

- Slides das aulas teóricas da Disciplina e exercícios práticos

- Swi Prolog Libraries and Documentation
  https://www.swi-prolog.org/

- Learn Prolog Now
  http://www.learnprolognow.org/