FUSE

Relatório Final

Mestrado Integrado em Engenharia Informática e Computação

Programação em Lógica

Grupo 4
Eduardo João Santana Macedo - 201703658
Francisco José Paiva Gonçalves - 201704790

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

15 de Novembro de 2019

Índice

Resumo

Foi-nos proposto desenvolver um jogo de tabuleiro numa linguagem de programação em lógica, Prolog. Escolhemos o Fuse, um jogo recentemente criado, para dois jogadores, que consiste em mover peças brancas e pretas das suas posições iniciais de maneira a no final do jogo criar o maior grupo consecutivo de peças dispostas ortogonalmente.

Este trabalho tinha como objetivo aprofundar e consolidar conceitos da linguagem Prolog, abordados nas aulas de Programação em Lógica. Foi benéfico ter um projeto como este, de maneira a ter um contacto aprofundado com a linguagem e a encontrar obstáculos que nos fazem evoluir.

Assim, concebemos um jogo de tabuleiro com 3 modos de jogo (Jogador v Jogador, CPU v CPU e Jogador v CPU).

As posições iniciais não podem conter peças nenhumas após a ficarem sem a peça inicial. Os movimentos das peças na vertical.

1. Introdução

O objetivo deste trabalho foi implementar um jogo de tabuleiro (Fuse), que tem interesse do ponto de vista da aprendizagem mais profunda e experimental da linguagem Prolog, já que há a necessidade de criar condições e regras para movimentar peças e também de determinar o resultado do jogo usando esta linguagem.

O relatório segue a seguinte estrutura:

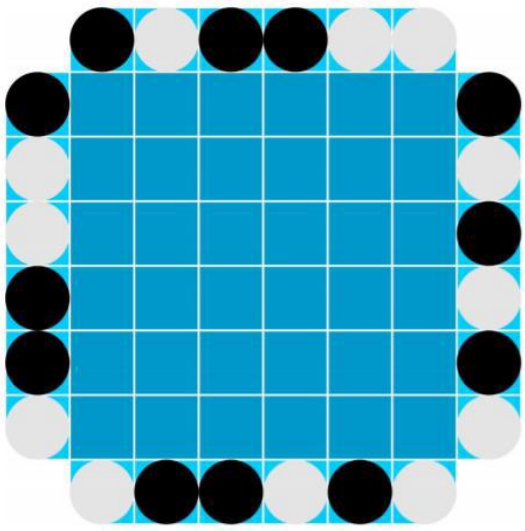
- O Jogo - Fuse: Descrição do jogo e suas regras
- Lógica do Jogo: Descrição da lógica do jogo, mas desta vez da perspetiva da implementação. Esta secção subdivide-se em:
  - Representação do estado do jogo: Exemplos visuais de estados de jogo (inicial, intermédio e final).
  - Visualização do Tabuleiro: Descrição da lógica por trás dos predicados da disposição do tabuleiro
  - Lista de Jogadas Válidas: Descrição da lógica por trás dos predicados que avaliam e geram jogadas válidas.
  - Execução e jogadas: Descrição da evolução do jogo e de como é avaliada cada jogada.
  - Avaliação do tabuleiro: Descrição dos predicados que analisam o estado do tabuleiro.
  - Final do Jogo: Descrição dos predicados usados para verificar se o jogo já chegou ao fim.
  - Jogada do Computador: Descrição dos predicados que geram jogadas automaticamente e jogando uma.
- Interface disponível: Descrição do interface criado que permite navegar nos menus do jogo.
- Conclusões: Ilações finais, comentários sobre o que foi implementado e reflexão.

2. O Jogo - Fuse

FUSE é um jogo de tabuleiro espanhol criado apenas em 2019 que iremos desenvolver usando a linguagem de Prolog. Trata-se de um jogo em que cada jogador joga alternadamente para no final obter a melhor pontuação.

Preparação do Jogo:

O jogo começa com as peças brancas e pretas distribuídas nas casas do perímetro aleatoriamente, porém nunca poderão estar mais que duas casas da mesma cor seguidas. Na imagem em baixo está representada uma possível configuração inicial do tabuleiro.



Exemplo de um tabuleiro no estado inicial

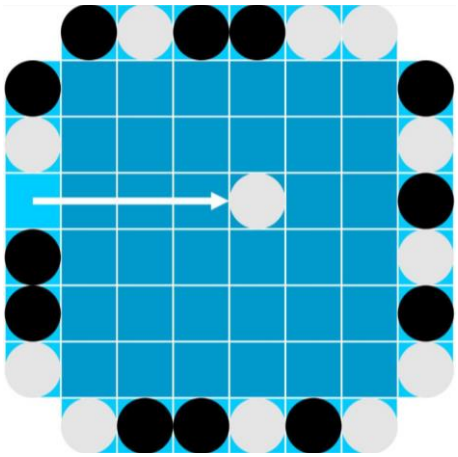
O material necessário para o jogo é um tabuleiro quadrado 7x7 (sendo que as casas dos cantos nunca serão usadas nem terão relevância no jogo), 12 peças brancas, 12 peças pretas e, para um modo de jogo de jogo extra, não implementado por nós, duas peças pequenas azuis.

Como jogar:

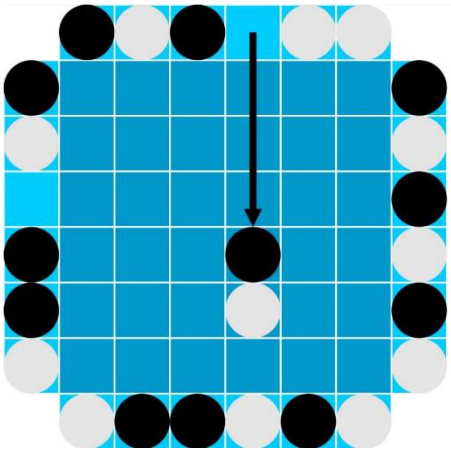
Começando com as peças brancas, os jogadores vão alternando jogadas, arrastando os discos do perímetro para a área do tabuleiro 6x6 (azul escura na imagem). O jogador é obrigado a usar um disco, exceto quando não tem alternativa sendo que, nesse caso, é obrigado a passar a vez. Os discos apenas se podem mover na coluna ou linha em que começaram e podem se mover tantas casas sendo que, dependendo de estarem inicialmente em baixo ou cima (movimento vertical) ou à esquerda ou direita (movimento horizontal):

Tópicos e notas importantes do jogo:
- Um ou mais discos que estejam no caminho podem ser empurrados como resultado da jogada.

- Todos os discos no fim da jogada têm de estar dentro da área jogável (6x6), ou seja, não é possível fazer uma jogada em que sejam empurrados discos para fora da área azul escura. Isto implica que, uma vez que a casa azul clara deixe de ter uma peça, nunca mais pode voltar a ter uma peça.



Exemplo de primeira jogada do jogo



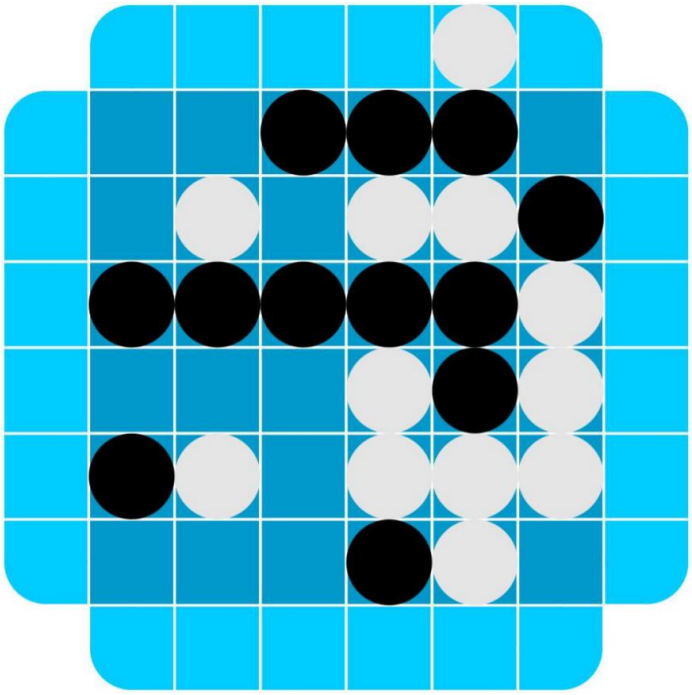
Exemplo de segunda jogada do jogo

Note-se que a peça branca jogada na primeira jogada foi empurrada pela peça preta, já que o segundo jogador decidiu empurrar a sua peça 4 casas.

Fim do jogo:

O jogo termina quando mais nenhum jogador já não tiver mais jogadas possíveis a fazer. O jogador que tiver mais peças conectadas ortogonalmente da sua cor ganha sendo que empates são possíveis.

Devido à rapidez do jogo, é recomendado este ser jogado a melhor de 3 ou de 5, funcionalidade que seria feita por conta própria, já que não existe nenhuma implementação nossa com um contador desse género ou modo de jogo nesses moldes.



Exemplo de vitória do jogador de peças brancas (7 – 6)

Note-se que o jogo terminou mesmo com a peça branca no perímetro pois já não são possíveis mais nenhumas jogadas (jogar a peça branca implicaria empurrar a outra peça branca da extremidade para fora da área jogável)

3. Lógica do Jogo

3.1 Representação do Estado do Jogo

Para armazenar a informação do jogo foi utilizada uma lista de listas com diferentes átomos (white, black, null e empty) para representarem as peças no tabuleiro ou ausência destas mesmas. Em baixo encontra-se a definição usada em código para dois exemplos de tabuleiros, inicial e final.

```
tabuleiroInicial([
    [null, black, white, black, black, white, white, null],
    [black, empty, empty, empty, empty, empty, empty, black],
    [white, empty, empty, empty, empty, empty, empty, white],
    [white, empty, empty, empty, empty, empty, empty, black],
    [black, empty, empty, empty, empty, empty, empty, white],
    [black, empty, empty, empty, empty, empty, empty, black],
    [white, empty, empty, empty, empty, empty, empty, white],
    [null, white, black, black, white, black, white, null]
    ]).

tabuleiroFinal([
    [null, empty, empty, empty, empty, white, empty, null],
    [empty, empty, empty, black, black, black, empty, empty],
    [empty, empty, white, empty, white, white, black, empty],
    [empty, black, black, black, black, black, white, empty],
    [empty, empty, empty, empty, white, black, white, empty],
    [empty, black, white, empty, white, white, white, empty],
    [empty, empty, empty, empty, black, white, empty, empty],
    [null, empty, empty, empty, empty, empty, empty, null]
    ]).
```

	1	2	3	4	5	6	7	8
a		B	W	B	B	W	W	
b	B	.	.	.	.	.	.	B
c	W	.	.	.	.	.	.	W
d	W	.	.	.	.	.	.	B
e	B	.	.	.	.	.	.	W
f	B	.	.	.	.	.	.	B
g	W	.	.	.	.	.	.	W
h		W	B	B	W	B	W	

Tabuleiro Inicial

	1	2	3	4	5	6	7	8
a		.	.	.	.	.	.	
b	.	.	.	B	B	B	W	.
c	.	.	W	.	W	W	B	.
d	.	B	B	B	B	B	W	.
e	.	.	.	.	W	B	W	.
f	.	B	W	.	W	W	W	.
g	.	.	.	.	B	W	.	.
h		.	.	.	.	.	.	

Tabuleiro Final

3.2 Visualização do Tabuleiro

Na entrega intermédia os predicados de display continham apenas tabuleiros constantes. Para esta entrega temos predicados que geram um tabuleiro aleatório no seu formato inicial, isto é, tendo 12 peças de cada cor e seguindo a regra de ter no máximo duas peças da mesma cor juntas.

```
pos(0,'a').
pos(1,'b').
pos(2,'c').
pos(3,'d').
pos(4,'e').
pos(5,'f').
pos(6,'g').
pos(7,'h').


symbol(empty,'.').
symbol(null,' ').
symbol(black,'B').
symbol(white,'W').

display_game(Board):-
    nl,
    write('   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |\n'),
    write('---|---|---|---|---|---|---|---|---|\n'),
    imprimeTabuleiro(Board, 0).
```

```
imprimeTabuleiro([], 8).

imprimeTabuleiro([Head|Tail], Number) :-
    pos(Number, L),
    write(' '),
    write(L),
    Number1 is Number + 1,
    write(' |'),
    imprimeLinha(Head),
    write('\n---|---|---|---|---|---|---|---|---|\n'),
    imprimeTabuleiro(Tail, Number1).

imprimeLinha([]).

imprimeLinha([Head|Tail]) :-
    symbol(Head,S),
    format(' ~s |', [S]),
    imprimeLinha(Tail).
```

As figuras acima representam o código de display de um tabuleiro (deve se lida primeiro a imagem da esquerda e depois a da direita).

3.3 Lista de Jogadas Válidas

Antes de referirmos como é obtida a lista de jogadas válidas iremos brevemente explicar um predicado muito importante para essa sua obtenção.

Uma jogada é considerada válida se obedecer aos seguintes critérios:

A cor da peça escolhida for da mesma cor atribuída ao jogador

O número de casas que se pretende avançar a peça não pode implicar que esta mesma saia fora do tabuleiro ou então também não poderá implicar que sejam empurradas outras peças para fora da área jogável.

	1	2	3	4	5	6	7	8
a		W	B	B	W	W	B	
b	B	.	.	.	.	.	.	B
c	W	.	.	.	.	.	.	W
d	W	.	.	.	.	.	.	W
e	B	.	.	.	.	.	.	B
f	W	.	.	.	.	.	.	B
g	W	.	.	.	.	.	.	W
h		B	W	B	W	B	W	

Cada peça só pode andar num sentido dependendo da sua posição inicial:

Na Coluna 1 as peças deslocam-se para a direita;

Na Coluna 8 as peças deslocam-se para a esquerda;

Na Linha 'a' as peças deslocam-se para baixo;

Na Linha 'h' as peças deslocam-se para cima;

Para a validação das jogadas são usados os seguintes predicados encontrados no ficheiro logic.pl:

checkValidPlay - É chamado para validar a jogada e consiste na chamada de dois predicados correspondentes aos dois critérios acima definidos.

```
checkValidPlay(Player, Board, Row, Column, Number)
checkCoord(Player, Row, Column, Board, Peca)
checkValidStepRight(Row, 1, Board, Number, 0)
checkValidStepLeft(Row, 8, Board, Number, 0)
checkValidStepDown(1, Column, Board, Number, 0)
checkValidStepUp(8, Column, Board, Number, 0)
```

O predicado que verifica o primeiro critério é o checkCoord e o que verifica o segundo é o checkValidStepXXXX, sendo o 'XXXX' dependendo da direção que a peça se irá mover.
Este predicado é também usado para validar as jogadas do CPU obtidas pelo predicado findall. (mais no capítulo 3.7)

checkCoord - Serve para verificar se a peça que o utilizador (Row, Column) escolheu é efetivamente da cor que lhe foi atribuída.

checkValidStepXXXX - Verifica se o número (Number) de blocos que o utilizador escolheu é válido. A validade é obtida contando o número de células 'empty' que o tabuleiro tem na linha/coluna que a peça se irá movimentar. A peça só poderá mover-se quantas casas 'empty' houver na sua respetiva linha/coluna de jogada, o que é verificada nesta seguinte condição após a contagem ter sido feita na segunda definição do predicado:

```
checkValidStepRight(Row, Column, Board, Steps, Counter) :-
    Column < 7,
    NextColumn is Column + 1,
    getPeca(Row, NextColumn, Board, Peca),
    (
        Peca == empty,
        NewCounter is Counter + 1;
        NewCounter is Counter
    ),
    checkValidStepRight(Row, NextColumn, Board, Steps, NewCounter).

checkValidStepRight(_, _, _, Steps, Counter) :-
    \+ (Steps > Counter).
```

Para a obtenção de uma lista de jogadas possíveis foram usados os seguintes predicados:

```
valid_moves(Board, Player, ListOfMoves):-
   findall([R,C,N], findall_aux(Board, R, C, N, Player), ListOfMoves).
findall_aux(Board, RowOut, ColOut, Number, Player) :-
    findall_aux_right(Board, 2, 8, 1, Player, RowOut, ColOut, Number);
    findall_aux_left(Board, 2, 1, 1, Player, RowOut, ColOut, Number);
    findall_aux_top(Board, 1, 2, 1, Player, RowOut, ColOut, Number);
    findall_aux_down(Board, 8, 2, 1, Player, RowOut, ColOut, Number).
```

findall - É usado este predicado que retorna na terceira variável (ListOfOutputs) uma lista de listas com todas as jogadas possíveis para as peças de uma determinada cor. Cada elemento da lista é uma lista com 3 elementos (ex. [1, 2, 3]) em que o primeiro elemento determina a linha, o segundo elemento a coluna e o terceiro o número de casas a avançar na jogada. Para o segundo argumento do predicado foi desenvolvida a função auxiliar findall_aux.

findall_aux - Predicado que irá chamar outros 4 predicados (findall_aux_XXXX) para verificar todas as jogadas possíveis que se podem fazer. Por exemplo, findall_aux_right irá ver procurar por todas as peças da cor do Player na coluna do lado direito e ver quantas blocos é que podem avançar cada uma no máximo. É também nos predicados findall_aux_XXXX que é usado o predicado acima mencionado checkValidPlay para dar fail ao predicado caso esteja a analisar uma jogada inválida não a colocando, portanto, na ListOfOutputs. Em baixo encontra-se a definição do predicado findall_aux_right:

```
findall_aux_right(Board, R, C, N, Player, RowOut, ColOut, NumOut) :-
    !, R < 8,
    ((
       checkValidPlay(Player, Board, R, C, N),
       RowOut is R,
       ColOut is C,
       NumOut is N
     );
     ((N > 5,
       NewR is R + 1, !,
       findall_aux_right(Board, NewR, C, 1, Player, RowOut, ColOut, NumOut));
      (NewN is N + 1, !,
       findall_aux_right(Board, R, C, NewN, Player, RowOut, ColOut, NumOut)
       ))).
```

3.4 Execução de Jogadas

O jogo tem um ciclo principal chamado mainLoop que está encarregue de ver se um jogador pode realizar jogadas ou terá de passar a vez, efetuar as jogadas e verificar se o jogo acabou.

Dependendo se é PvP, PvCPU ou CPUvCPU o nosso programa tem mainLoops diferentes apesar de serem semelhantes entre si. Aqui encontra-se o predicado mainLoop2 (PvCPU) que acaba por ser uma junção dos restantes dois modos.

```
mainLoop2(Player1, CPU, Board, Level):-
     game_over(Board);
     valid_moves(Board, Player1, ListOfMoves),
        length(ListOfMoves, Size1),
         (  Size1 \= 0 ->
            write('> White Player\'s turn...\n'),
            askCoordsWhite(Player1, Board, NewBoard)
            ; write('No valid plays for white player...\n'),
            copy(Board, NewBoard)),
    valid_moves(NewBoard, CPU, ListOfMoves2),
         length(ListOfMoves2, Size2),
          ( Size2 \= 0 ->
            write('> Black Player\'s turn...\n'),
    choose_move(CPU, Level, ListOfMoves2, NewBoard, FinalBoard),
            display_game(FinalBoard)
            ; write('No valid plays...\n'),
                copy(NewBoard, FinalBoard)
         ),
      mainLoop2(Player1, CPU, FinalBoard, Level).
```

O ciclo começa com a verificação se o jogo acabou. Caso esta verificação falhe o ciclo continua iniciando a jogada do Player1. Aqui, para evitar a situação de pedir a jogada ao utilizador e este não ter jogadas possíveis, é chamado o valid_moves para obtermos a lista de todas as jogadas válidas. Caso o tamanho desta lista for zero, passamos a jogada à frente. Caso contrário é invocado o predicado askCoordsWhite que pede a jogada ao utilizador e, caso seja válida, chama o predicado move para a efetuar (explicado com mais rigor em baixo).

```
askCoordsWhite(Player, Board, NewBoard):-
    askRow(NewRow), nl,
    askColumn(NewColumn),
    askBlocks(Number),
    move(Player, Board, NewBoard, NewRow, NewColumn, Number).
```

A segunda metade do ciclo segue a mesma estruture, no entanto, em vez de pedir a jogada ao utlizador chama o predicado choose_move para obter uma jogada da lista de jogadas válidas possíveis dependendo da dificuldadade (Level). No fim o predicado é chamado recursivamente com o novo tabuleiro obtido.

3.5 Final do Jogo

Os predicados de verificação de final do jogo (game_over(Board)) estão separados dos predicados de determinar o vencedor. O raciocínio por trás do predicado game_over(Board) foi dividi-lo, por assim dizer, em 4 predicados distintos: um para o topo (linha), outro para a direita (coluna), outro para baixo (linha) e outro para a esquerda (coluna). Como estes predicados são iguais entre si, explicar a lógica de um, explica a lógica dos quatro. O predicado checkGameOverTop(Board, Row, Col) recebe as 3 variáveis indicadas, sem devolver nenhuma. O objetivo dele é verificar peça a peça o conteúdo da linha (neste caso). Caso a linha esteja completamente vazia, passa para o checkGameOverRight(Board, Row, Col) que fará o mesmo, mas para a coluna da direita e assim sucessivamente. Se a linha não estiver vazia será feita a verificação da linha ou coluna (dependendo do predicado atual – top, right, bottom ou left) e em caso de não estar é porque o jogo ainda pode continuar. Se a linha estiver cheia, os predicados devem continuar a verificação até terminar o predicado checkGameOverLeft.

```
game_over(Board) :-
    checkGameOverTop(Board, 1, 1),
    checkGameOverRight(Board, 1, 8),
    checkGameOverBottom(Board, 8, 1),
    checkGameOverLeft(Board, 1, 1),
```

3.6 Avaliação do Tabuleiro

No contexto do nosso jogo faz sentido fazer uma avaliação do tabuleiro apenas no final do jogo, portanto se o jogo estiver terminado (determinado com predicados descritos na secção anterior) é chamado duas vezes o predicado traverseBoard(Board, Row, Col, Player, Score AuxScore). Este predicado invoca investigaPeca(Board, Row, Col, Player, Score, NewScore, BoardV). Nas variáveis BScore e WScore serão retornados os pontos do jogador com peças pretas e peças brancas, respetivamente. O valor BoardV deve ser interpretado como o tabuleiro que vai sofrendo consecutivas alterações (valores de *visited* a serem escritos por cima dos valores originais do tabuleiro que representava o fim do jogo). Com o predicado printWinner(B, W) é possível imprimir se houve um empate ou vitória das peças brancas ou vitórias das peças pretas.

A ideia por trás destes predicados é usar o conceito de visitado e ir alterando o Board. A cada peça que fazemos "get" verificamos se já foi visitada ou não. Caso não tenha sido visitada, faz-se "set" dessa peça como visitada (escrevendo por cima do valor que lá estava), verificar se é do tipo Player e se for incrementar 1. Logo a seguir deve ser feita uma verificação para cima, baixo, esquerda e direita alterando o resultado (score) atual. Caso uma destas condições falhe deve haver um "else" por assim dizer que permita "abortar a verificação e avançar para a próxima peça. Caso haja uma interrupção na ligação destas peças e outra sequência com mais peças ligadas a função traverseBoard responsabiliza-se de detetar qual é a máxima pontuação. Abaixo está o predicado traverseBoard() e também investigaPeca(). BoardV corresponde a Board Visitado, tabuleiro com alterações. Em anexo segue a implementação dos predicados checkBottom, checkRight, checkLeft e checkTop.

```
investigaPeca(Board, Row, Col, _, Score, NewScore, BoardV) :-
    getPeca(Row, Col, Board, T),
    \+(checkVisited(Board, T, Row, Col, BoardV)),
    copy(Board, BoardV), NewScore is Score.

investigaPeca(Board, Row, Col, Player, Score, NewScore, BoardV) :-
    getPeca(Row, Col, Board, T),
    checkVisited(Board, T, Row, Col, BoardV),
    Player == T,
    AuxScore is Score + 1,
    checkLeft(BoardV, Row, Col, Player, AuxScore, AuxLeft),
    checkRight(BoardV, Row, Col, Player, AuxLeft, AuxRight),
    checkTop(BoardV, Row, Col, Player, AuxRight, AuxTop),
    checkBottom(BoardV, Row, Col, Player, AuxTop, NewScore).
```

```
investigaPeca(Board, Row, Col, Player, Score, NewScore, BoardV) :-
    getPeca(Row, Col, Board, T),
    checkVisited(Board, T, Row, Col, BoardV),
    Player \= T,
    NewScore is Score.

traverseBoard(_, 9, _, _, Score, ScoreAux) :-
    ScoreAux is Score.

traverseBoard(Board, Row, Col, Player, Score, ScoreAux) :-
    % trace,
    investigaPeca(Board, Row, Col, Player, Score, ScoreAuxInvest, BoardV),
    ((ScoreAuxInvest > Score, ScoreAuxTraverse is ScoreAuxInvest)
        ; ScoreAuxTraverse is Score),
    NewCol is Col + 1,
    ((      NewCol > 8,
            NewRow is Row + 1, %se ultrapassar 8 next row
            traverseBoard(BoardV, NewRow, 1, Player, ScoreAuxTraverse, S)
        ); traverseBoard(BoardV, Row, NewCol, Player, ScoreAuxTraverse, S)),
    ScoreAux is S.
```

3.7 Jogada do Computador

Para a geração de jogadas do computador, após no mainLoop obtermos a lista de jogadas possíveis e concluirmos que o jogador não pode passar a vez (length of ListOfMoves diferente de 0) é evocado o predicado choose_move que escolhe a jogada do computador com base na dificuldade escolhida(Level). A dificuldade do modo 4 CPUvCPU é por predefinição aleatória que é a recomendada.

```
choose_move(CPU, Level, ListOfOutputs, Board, NewBoard)
move(CPU, Board, NewBoard, Row, Column, Number)
```

choose_move - Funciona de forma diferente dependendo do valor de Level. Se este for 0, irá buscar o primeiro elemento da lista de jogadas possíveis e chamar o predicado move para realizar a jogada. Caso tenha o valor 1, funcionará de uma forma semelhante, mas, no entanto, o índice da jogada que irá ser extraída da lista será agora aleatório.
No fim da jogada estar concluída é imprimida no ecrã a Linha, Coluna e número de casas que a peça avançou para informar o Utilizador.
move - Faz mover a peça no tabuleiro com base nos valores de Row, Column e Number nas variáveis de entrada.

O uso do predicado findall (já acima explicado) foi criado inicialmente para obter as jogadas possíveis do CPU. No entanto o seu uso e importância foi aumentado pois é sempre chamado no mainLoop antes de uma jogada para verificar se é necessário dar skip à vez do jogador pois pode não haver jogadas válidas para este mesmo.

(Excerto de mainLoop2)

```
mainLoop2(Player1, CPU, Board, Level):-
     game_over(Board);
     valid_moves(Board, Player1, ListOfMoves),
        length(ListOfMoves, Size1),

         (  Size1 \= 0 ->

                write('> White Player\'s turn...\n'),
                askCoordsWhite(Player1, Board, NewBoard)
                ;
                write('No valid plays for white player...\n'),
                copy(Board, NewBoard)
         ),
```

Interface

A interface é curta, simples e auto explicativa. A opção 1 leva ao modo jogador contra jogador (humanos), a opção 2 ao modo jogador contra computador, a opção 3 ao modo anterior numa dificuldade mais elevada e a quarta opção computador contra computador. A opção 0 permite sair do programa. Quando o jogo termina também há uma opção de introduzir 0 para voltar ao menu principal, acompanhado de um *print* no ecrã a indicar para o fazer. É importante sublinhar que todas as *inputs* devem seguir a regra de terminar com um ponto final.

```
 ____________________________________________________
|                                                    |
|        ########  ##     ##   ######   ########     |
|        ##        ##     ##  ##    ##  ##           |
|        ##        ##     ##  ##        ##           |
|        ######    ##     ##   ######   ######       |
|        ##        ##     ##        ##  ##           |
|        ##        ##     ##  ##    ##  ##           |
|        ##         #######    ######   ########     |
|____________________________________________________|
|                                                    |
|                  [1]  P  vs  P                     |
|                  [2]  P  vs CPU (Very Easy)        |
|                  [3]  P  vs CPU (Recommended)      |
|                  [4] CPU vs CPU                    |
|                  [0] Exit                          |
|                                                    |
|____________________________________________________|

> Choose an option
|:
```

```
    | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
---|---|---|---|---|---|---|---|---|---|
 a |   | . | . | . | . | . | . |   |
---|---|---|---|---|---|---|---|---|---|
 b | . | W | W | B | B | W | W | . |
---|---|---|---|---|---|---|---|---|---|
 c | . | B | . | . | . | . | B | . |
---|---|---|---|---|---|---|---|---|---|
 d | . | B | W | W | W | . | W | . |
---|---|---|---|---|---|---|---|---|---|
 e | . | . | W | . | B | B | B | . |
---|---|---|---|---|---|---|---|---|---|
 f | . | . | . | B | . | B | W | . |
---|---|---|---|---|---|---|---|---|---|
 g | . | W | W | . | . | W | W | . |
---|---|---|---|---|---|---|---|---|---|
 h |   | . | . | . | . | . | . |   |
---|---|---|---|---|---|---|---|---|---|
============================
====     GAME OVER      ====
==== WHITE PLAYER WINS ====
============================

Press [0] to go back to MAIN MENU.
```

4. Conclusões

O projeto desenvolvido tinha como objetivo desenvolver as capacidades e conhecimentos em Prolog e nesse sentido o projeto foi muito benéfico.

Ao longo da realização do projeto fomos desenvolvendo o raciocínio necessário para entender esta linguagem que requer uma esquematização e conceção diferente da maioria das linguagens com que já trabalhamos até agora no curso (MIEIC).

Pensamos que a nossa implementação cumpre os requisitos, já que com sucesso conseguimos desenvolver o jogo, apesar de poderem ser feitas melhorias ao código. Por exemplo, os predicados de verificação de fim de jogo foram feitos antes dos predicados que geram listas de jogadas válidas e nesse sentido a implementação podia ser melhor, pois podia usar listas de jogadas válidas, poupávamos linhas de código e provavelmente a execução, conceção e leitura do programa seria mais simples.

Para concluir, o trabalho foi completo com sucesso, e contribuiu para a melhor integração nesta maneira de pensar e conceber código, em Prolog.

Bibliografia

Regras Fuse: *https://nestorgames.com/rulebooks/FUSE_EN.pdf*
Site principal: *https://nestorgames.com/#fuse_detail*

Anexo I

Abaixo segue a implementação dos métodos de check necessários para avaliação de tabuleiro:

```
checkVisited(Board, Peca, Row, Col, RetBoard):-
    Peca \= visited,
    setPeca(Row, Col, visited, Board, RetBoard).


checkLeft(Board, Row, Col, Player, AuxScore, NewScore):-
    (
        Col \= 1,
        NCol is Col-1,
        getPeca(Row, NCol, Board, Type),
        Type == Player,
        NewScore is AuxScore + 1
    ); NewScore is AuxScore.

checkRight(Board, Row, Col, Player, AuxScore, NewScore):-
    (
        Col \= 8,
        NCol is Col+1,
        getPeca(Row, NCol, Board, Type),
        Type == Player,
        NewScore is AuxScore + 1
    ); NewScore is AuxScore.


checkTop(Board, Row, Col, Player, AuxScore, NewScore):-
    (
        Row \= 1,
        NRow is Row+1,
        getPeca(NRow, Col, Board,Type),
        Type == Player,
        NewScore is AuxScore + 1
    ); NewScore is AuxScore.

checkBottom(Board, Row, Col, Player, AuxScore, NewScore):-
    (
        Row \= 8,
        NRow is Row-1,
        getPeca(NRow, Col, Board, Type),
        Type == Player,
        NewScore is AuxScore + 1
    ); NewScore.
```