FUSE: Análise do Jogo

Relatório Final

Mestrado Integrado em Engenharia Informática e Computação

Programação em Lógica

Grupo 'FUSE_1':
André Filipe Magalhães Rocha - up201706462@fe.up.pt
Manuel Monge dos Santos Pereira Coutinho - up201704211@fe.up.pt

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

17 de Novembro de 2019

Resumo

O presente relatório serve de suporte de documentação teórica ao desafio lançado na unidade curricular PLOG: implementação na linguagem de programação Prolog de um jogo de tabuleiro - no nosso caso **FUSE**.

Este problema tinha como objetivo, não só a permitir duas pessoas jogar uma contra a outra, mas também de permitir a escolha de jogar contra o computador (ou de ver dois computadores a jogar), segundo diferentes níveis de dificuldade, determinados pelos algoritmos que este(s) usa(m) na escolha da jogada seguinte.

Depois de uma breve contextualização, este documento apresenta uma descrição detalhada de algoritmos de recolha de todas as jogadas válidas, usando o predicado da biblioteca *findall*; da interface feita por meio de menus; da representação interna em listas de listas e da sua visualização; da implementação do movimento de uma peça por algoritmos recursivos que permitem empurrar peças que se encontram no caminho desta; contagem final dos pontos de cada jogador inspirado em algoritmos como o *flood fill*; e explicação dos algoritmos minimax, aleatório e gananciosos que o computador usa para escolher o seu próximo movimento.

Concluimos com uma indagação sobre a enorme influência que uma linguagem diferente tem na forma como pensamos e abordamos um problema, e como uma adaptação a um paradigma de programação novo pode causar alguns contratempos inesperados.

Conteúdo

1 Introdução

Na Unidade Curricular de Programação em Lógica, foi-nos proposta a escrita de um relatório relativamente ao desenvolvimento de um Jogo de Tabuleiro em Prolog. A escolha deste foi feita com base numa lista pré determinada, sendo que o grupo selecionou o FUSE.

Durante o presente documento, será apresentado uma pequena história da criação do jogo, assim como das suas regras. Será ainda feita uma demonstração em Prolog de como o jogo será modelado, apresentando diversos estados do mesmo e a respetiva apresentação no terminal, assim como é feita a respetiva interação com o utilizador.

É ainda apresentado como o jogo em si foi implementado, detalhando diversos pormenores relativamente a aspetos como a obtenção de jogadas válidas e a sua execução, como é que avaliado o final do jogo e realizada a classificação de um tabuleiro.

Por fim é apresentado como são desenvolvidos os diferentes níveis de dificuldade, destacando o funcionamento das diversas heurísticas e algoritmos usados.

É de notar que todo o código apresentado durante este trabalho foi escrito utilizando SWI-Prolog, utilizado maioritariamente pelas suas capacidades de escrever caracteres com formatação com atributos ANSI.

2 FUSE

2.1 História

Néstor Romeral Andrés é um criador de jogos de tabuleiro com elevada experiência. Com mais de 120 jogos de tabuleiro já criados, FUSE inspira-se noutra criação sua, Feed the Ducks do tipo 'push/attract', revelando uma crescente maturidade em relação ao seus jogos passados.

2.2 Regras

Durante um jogo de FUSE, ambos os jogadores introduzem, alternadamente, discos dentro do tabuleiro, onde poderão eventualmente empurrar outros discos no caminho, de modo a criarem o grupo da sua cor de maior tamanho.

2.2.1 Preparação do Tabuleiro

São usados 12 discos brancos e 12 discos pretos, espalhados aleatoriamente pelo tabuleiro, de modo a que não haja mais do que dois discos da mesma cor colocados consecutivamente. Uma possível configuração inicial é apresentada na Figura 1. No caso de ambos os jogadores discordarem da disposição inicial das peças, esta pode ser descartada e será gerada uma nova.

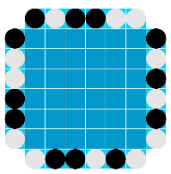
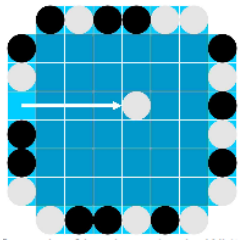


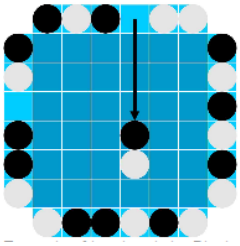
Figura 1: Figura 1 - Exemplo de uma possível disposição inicial do tabuleiro

2.2.2 Desenvolvimento do Jogo

O jogo é iniciado pelo jogador que escolheu os discos brancos, lançando-se alternadamente um novo disco para dentro da área 6x6 interior do tabuleiro. Os jogadores devem lançar um novo disco para dentro do tabuleiro se possível. Caso contrário, o jogador deverá passar a jogada. Os discos podem-se mover ao longo da linha ou coluna onde começaram, sem qualquer tipo de restrições relativamente ao número de casas a avançar, empurrando os discos à sua frente como resultado. Qualquer disco já em jogo, deverá manter-se sempre na área 6x6 do tabuleiro, uma vez que não é possível puxar o disco para a periferia ou fora do mesmo.



(a) Figura 2 - Exemplo de uma primeira jogada



(b) Figura 3 - Exemplo de uma resposta válida

2.2.3 Fim do Jogo

O jogo termina quando ambos os jogadores passam a vez consecutivamente, ou seja, nenhum dos dois consegue trazer uma nova peça da área inicial do tabuleiro de jogo para o quadrado 6x6 respeitando todas as regras previamente explanadas.

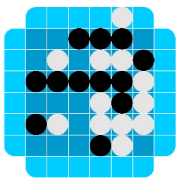


Figura 3: Exemplo de um final de jogo

2.2.4 Vencedor

Existem várias formas de pontuar o jogo e decidir o vencedor. O caso mais simples baseia-se na contagem do maior grupo de peças conectadas ortogonalmente (na imagem acima ganham as brancas 7 vs 6). Um modo mais avançado consiste em multiplicar o tamanho de todos os grupos (ganham as pretas pois 6x3x1x1x1 = 18 vs 7x2x1x1x1 = 14). Existe ainda o modo 'Kings' onde cada rei é assinalado com uma peça azul por cima e vale 2 pontos na contagem final (o número de réis a que cada jogador tem direito é acordado no início).

3 Lógica do Jogo

3.1 Interface com Utilizador

Para uma mais fácil interação com o utilizador do programa, foi criada uma interface de menus simples que lhe permitem escolher de entre as diversas opções que o programa apresenta. A navegação entre menus é feita por meio de números indicando as diferentes opções, com recurso a *read_menu(-Option, +Type)* que, dependendo do Tipo de menu que se lhe apresente, responde e faz parsing do valor introduzido pelo utilizador.

3.1.1 Menu Inicial

O menu inicial, chamado por *display_menu(-Option)* permite escolher o modo de jogo que se deseja (jogador vs jogador; computador vs jogador; computador vs computador), pedir para ver instruções, ou sair do ciclo através da opção *0:Exit*.

Antes de cada um destes menus, é também impresso no ecrã um fabuloso logo em arte do tipo ASCII com recurso a *display_logo*.

Caso a opção jogador vs jogador seja a escolhida, é pedido ao utilizador o número de colunas e de linhas da parte do centro (jogável) do tabuleiro. Apesar de virtualmente não haver limite, foi imposto um máximo de 7 por 7 exclusivé, pois as computações, caso se escolhesse um robot do tipo MiniMax, ficariam excessivamente pesadas.

Qualquer uma das outras opções de modo de jogo leva o utilizador ao menu de escolha do robot explanado na secção seguinte.

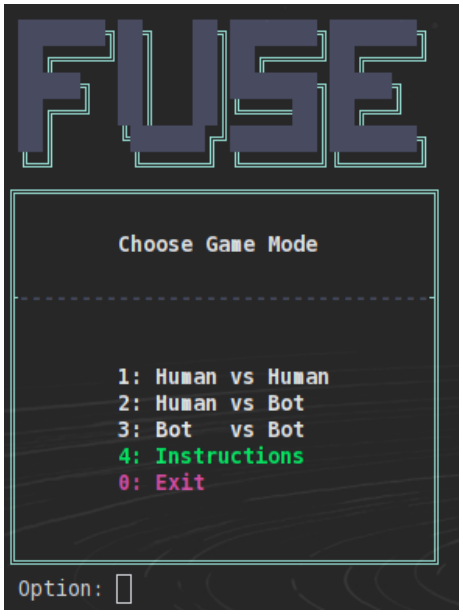


Figura 4: Menu Inicial

3.1.2 Menu Robot

Este menu permite a escolha do nível do(s) robot(s) - plural caso seja o modo robot vs robot. É ainda possível retroceder para o menu inicial a qualquer

momento.

Os níveis discriminados são os seguintes:

1. **Very easy:** O robot efetua uma jogada válida aleatória
2. **Easy:** O robot usa um algoritmo ganancioso que avalia a melhor jogada como aquela que faz a maior ilha
3. **Normal:** O robot usa um algoritmo ganancioso que avalia a melhor jogada como aquela que torna a diferença de tamanhos das duas maiores ilhas o maior possível (positivo para si)
4. **Hard:** O robot usa um algoritmo minimax com profundidade de 1 e avalia a melhor jogada como aquela em que, após a melhor jogada do adversário, a maior ilha dele será o menor possível
5. **Very hard:** O robot usa um algoritmo minimax com 1 de profundidade que admite como melhor jogada aquela que, após a melhor jogada adversário, a diferença entre o tamanho das maiores ilhas é melhor para o seu lado.

Resumidamente, implementaram-se 3 algoritmos - aleatório, ganancioso e minimax - e 2 funções de avaliação - maior ilha e diferença entre os tamanhos das maiores ilhas - e procedeu-se a combinações entre eles.

Caso se tenha escolhido o modo de jogo robot vs robot é necessário fazer a seleção do nível duas vezes (uma por robot) e, de seguida, escolher as dimensões do tabuleiro de forma análoga à descrita anteriormente.

Caso contrário (jogo entre um robot e um humano), após a escolha do grau de dificuldade desejado, o utilizador é enviado para um menu que permite a escolha do primeiro a jogar.

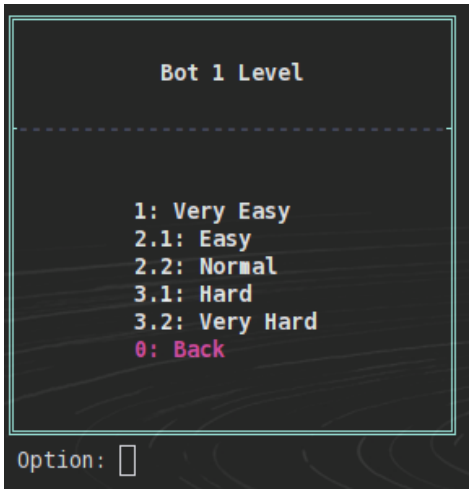


Figura 5: Menu Robot

3.1.3 Menu Ordem de Jogada

Este menu é apenas chamado caso se trate do modo de jogo jogador vs robot e permite ou escolher de entre ambos quem jogará primeiro, ou voltar para o menu inicial.

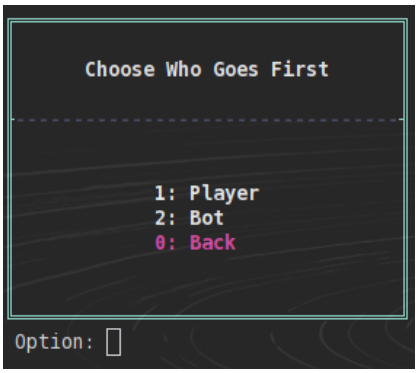


Figura 6: Menu Ordem de Jogada

3.1.4 Instruções

Neste menu encontram-se as instruções muito simplificadas de qual o objetivo do jogo e do formato do input do movimento.

Estas instruções não dispensão a leitura cuidada das regras detalhadas do jogo.

Para voltar para o menu inicial o utilizador apenas precisa de pressionar a tecla ENTER.

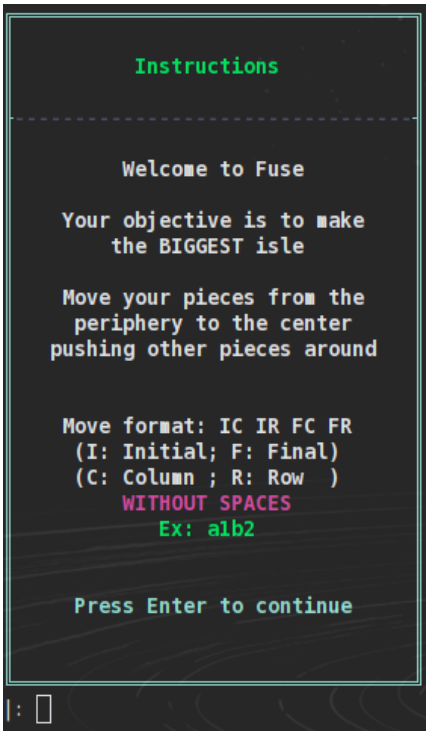


Figura 7: Instruções

3.2 Representação do Estado do Jogo

Uma vez que o nosso tabuleiro principal possui, adjacente a ele, em cada um dos lados uma entrada (linha e/ou coluna) quadrados onde se encontram

os discos, foi necessário procurar uma solução para lidar com estes espaços que mais tarde não poderiam ser ocupados. Deste modo foi criado o átomo null, cuja função será preencher as posições do tabuleiro que não poderão ser mais ocupados.

```
wt:- ansi_format([bold, fg(white)], '~c', [9679]). %
    Disco branco
bl:- ansi_format([bold, fg(black)], '~c', [9679]). %
    Disco preto
null:- write('␣'). % Disco nulo
corner:- write('␣'). % Canto do Tabuleiro
empty:- ansi_format([bold, bg(cyan)], '~s', ['␣']). %
    Lugar vazio
```

Figura 1: Traduções para o terminal

3.2.1 Estado Inicial

Uma vez que o estado do tabuleiro não será sempre o mesmo em cada início de um jogo de FUSE, apenas é possível mostrar um possível exemplo de como seria o estado do tabuleiro caso este fosse aceite. Um tabuleiro apenas é aceite caso este não possua mais que duas peças do mesmo tipo consecutivamente.

```
initial_board([
    [null, wt, bl, wt, bl, bl, wt, null], %1
    [wt, empty, empty, empty, empty, empty, empty, wt], %
        2
    [bl, empty, empty, empty, empty, empty, empty, bl], %
        3
    [wt, empty, empty, empty, empty, empty, empty, wt], %
        4
    [wt, empty, empty, empty, empty, empty, empty, bl], %
        5
    [bl, empty, empty, empty, empty, empty, empty, bl], %
        6
    [wt, empty, empty, empty, empty, empty, empty, wt], %
        7
    [null, bl, wt, wt, bl, wt, wt, null]   %8
]).
```

Figura 2: Exemplo um tabuleiro no seu estado inicial

3.2.2 Estados Intermédios

A cada jogada, cada um dos jogadores coloca um disco em jogo, pelo o que apenas irá abdicar de tal direito caso não seja possível colocar nenhum dos seus discos em jogo.

```
middle_state_board([
    [null, null, null, null,null, wt, null, null], %1
    [null, bl, empty, bl, bl, empty, empty, null], %2
    [null, empty, wt, empty, wt, wt, bl, null], %3
```

```
    [null, bl, bl, bl, bl, bl, wt, null], %4
    [null, empty, bl, empty, wt, bl, wt, null], %5
    [null, bl, wt, empty, wt, wt, wt, null], %6
    [null, empty, empty, empty, bl, wt, empty, null], %7
    [null, null, null, null, null, null, null, null]  %8
]).
```

Figura 3: Exemplo de Estado Intermédio

É de notar que aquando a colocação em jogo de um determinado disco, este nunca mais se poderá mover, excepto quando um outro colidir e empurrar o mesmo.

```
second_middle_board([
[null, wt, bl, null, bl, bl, empty, null], %1
[wt, empty, empty, bl, empty, empty, empty, wt], %2
[bl, empty, empty, empty, bl, empty, wt, null], %3
[wt, empty, empty, empty, empty, empty, empty, wt], %4
[wt, empty, empty, wt, empty, empty, empty, bl], %5
[null, empty, empty, empty, bl, empty, empty, bl], %6
[wt, empty, empty, empty, empty, empty, empty, wt], %7
[null, bl, wt, wt, bl, wt, wt, null]  %8
]).
```

Figura 4: Mais um possível estado intermédio do tabuleiro

3.2.3 Estado final

Um jogo de FUSE termina quando nenhum movimento é possível, pelo que ambos os jogadores passam a vez. Um possível exemplo de um tabuleiro no seu estado final é apresentado em baixo.

```
final_board([
    [null, null, null, null, null, wt, null, null], %1
    [null, empty, empty, bl, bl, bl, empty, null], %2
    [null, empty, wt, empty, wt, wt, bl, null], %3
    [null, bl, bl, bl, bl, bl, wt, null], %4
    [null, empty, empty, empty, wt, bl, wt, null], %5
    [null, bl, wt, empty, wt, wt, wt, null], %6
    [null, empty, empty, empty, bl, wt, empty, null], %7
    [null, null, null, null, null, null, null, null]  %8
]).
```

Figura 5: Exemplo um tabuleiro no seu estado inicial

3.3 Visualização do Tabuleiro

Como descrito no relatório anterior, o tabuleiro é representado por um quadrado vazio, das dimensões desejadas onde os espaços vazios têm cor azul ciano. Em torno deste quadrado há colunas e linhas de onde partem inicialmente as peças e para onde nenhuma pode retornar (por isso, é que não são azuis quando vazias). As peças são representadas por círculos brancos, correspondentes ao jogador 0 (primeiro jogador), e círculos pretos, correspondentes às peças do jogador 1 (segundo jogador).

Em baixo encontra-se uma imagem de um possível estado inicial que recorreu ao predicado *display_game(+Board, +Player)* para que pudessem aparecer no ecrã. Este permite mostrar tabuleiros de qualquer dimensão pretendida da forma descrita na documentação do código da mesma (função presente em *display.pl*).

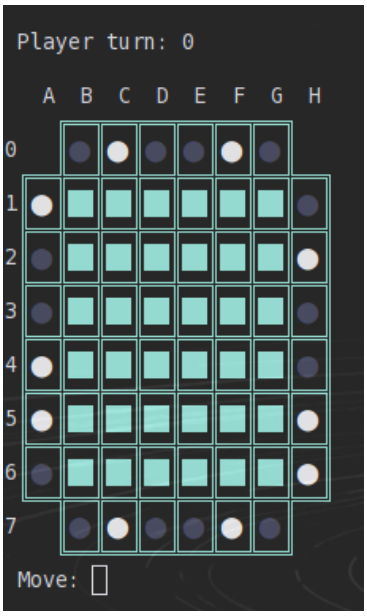


Figura 8: Tabuleiro à espera do *Input* do Utilizador

Como podemos constatar após cada jogada, caso se trate a vez de um jogador humano, é pedido que este introduza o movimento desejado. Usando o predicado *read_move(-Move, +Board)* presente em *interface.pl* é verificado se o Input vem no formato pretendido - ColunaInicial LinhaInicial ColunaFinal LinhaFinal (sem espaços) -, se as posições são válidas (dentro do tabuleiro) e se a inicial é diferente da final. Caso tal não aconteça, é mostrada a mensagem de erro adequada deixando o utilizador reintroduzir o movimento. Caso este digite *'exit'*, procedesse à terminação do jogo por via de um *abort*.

É de notar que, enquanto as linhas são representadas por meio de números, as colunas devem ser letras não importando se se tratam de maiúsculas ou minúsculas: $a1c2 = A1c2 = a1C2 = A1C2$.

3.4 Gerar o Tabuleiro

Uma vez que a geração do nosso tabuleiro é aleatória mas com certas restrições, é relevante que seja explicado tal processo. Uma configuração inicial do FUSE só é considerada válida caso não haja mais do que duas peças da mesma cor seguidas.

Para tornar esta computação mais eficiente, é gerada uma lista de peças que irá contar com todas as peças que serão colocadas nos extremos do tabuleiro, respeitando as seguintes precauções:

- Caso as últimas duas peças geradas sejam da mesma cor, é necessário, naturalmente, escolher uma peça da cor contrária.

- Caso as duas primeiras peças da lista sejam da mesma cor, é necessário que a última peça seja de uma cor diferente, uma vez que a lista deve ser interpretada como circular.
- Caso as duas últimas peças da lista sejam da mesma cor, é necessário aplicar o mesmo referido no caso anterior.
- Caso a primeira e a última peça sejam da mesma cor, a 2ª e penúltima peça do mesmo têm de ser de outra cor.
- Caso a penúltima e última peça do tabuleiro sejam da mesma cor e a segunda e a terceira primeiras peças do tabuleiro também sejam da cor (podendo estes dois grupos ser de cor diferente), o tabuleiro não é realizável.

O predicado de geração *generate_pieces(+Acc, -PiecesList, +NumPieces)*, bem como o de colocação no tabuleiro (conseguido através de rotações do mesmo) *generate_board(+EmptyBoard, -InitializedBoard, +PiecesList)*, encontram-se em *board_generation.pl*.

3.5 Lista de Jogadas Válidas

Para a obtenção de uma lista com todas as jogas válidas, o predicado *valid_moves(+Board, +Player, -ListOfMoves)* é chamado e uma Lista de movimentos é devolvida pelo último argumento.

Esta lista é extremamente importante para a implementação dos *bots*, uma vez que todos eles vão lá buscar o movimento a efetuar dependendo do critério específico ao nível de dificuldade (escolha abordada no capítulo 3.9 com o predicado *choose_move*). Também é importante na passagem de jogada ao próximo jogador, situação de alguma preponderância neste jogo (ver capítulo 3.7 sobre o Final do Jogo).

Esta lista é obtida com o auxílio do *findall* recorrendo ao uso de duas funções diferentes: uma primeira que verifica se uma dada posição está dentro do tabuleiro central (*inside_board(+Board, -Column, -Row)*) que, em conjunto com um *findall*, gera todas as posições do tabuleiro possíveis (usadas como possíveis posições finais); e uma segunda que tenta mover uma peça de uma posição inicial para uma final, determinando se foi bem sucedido ou não (*move(+Board, +Move, -NewBoard, +Player)* (explicada em maior detalhe no capítulo 3.6 sobre Execução de Jogadas).

Por cada posição final obtida, é corrido o predicado *findall* com *move* como parâmetro que retorna todas as posições inicias de onde, para aquele dado jogador e para aquele local, é possível mover uma das suas peças, guardando todos esses movimentos na lista final (os movimentos são um par posição inicial - posição final).

3.6 Execução de Jogadas

O predicado *move(+Board, +Move, -NewBoard, +Player)* não só valida o resto do movimento que lhe é dado como input (algum parsing já era feito aquando da introdução dos dados - ver capítulo 3.2), como também move, caso seja bem sucedido, efetivamente a peça da posição inicial para a final, empurrando todas as que se encontrem no seu caminho.

A ordem de validação vai do mais simples para o mais complexo, para caso falhe, não perder tempo desnecessário em cálculos dispendiosos.

Assim sendo, começamos por verificar se existe alguma peças na posição inicial e, caso exista, se esta corresponde ao jogador que pediu o movimento (ao jogador 0 correspondem as peças brancas e ao jogador 1 as pretas).

De seguida, verificamos se a direção e a posição inicial do movimento são congruentes, ou seja:

- Caso a coluna inicial seja a primeira, o movimento terá de ser horizontal (número da linha inicial igual ao da final) e a posição final terá de ser alguma dentro do tabuleiro quadrado com um número de coluna superior ao inicial;
- Caso a linha inicial seja a primeira, o movimento terá de ser vertical (número da coluna inicial igual ao da final) e a posição final terá de ser alguma dentro do tabuleiro quadrado com um número de linha superior ao inicial;
- Caso a coluna inicial seja a última, o movimento terá de ser horizontal (número da linha inicial igual ao da final) e a posição final terá de ser alguma dentro do tabuleiro quadrado com um número de coluna inferior ao inicial;
- Caso a linha inicial seja a última, o movimento terá de ser vertical (número da coluna inicial igual ao da final) e a posição final terá de ser alguma dentro do tabuleiro quadrado com um número de linha inferior ao inicial;
- Caso contrário o movimento é errado

A verificação de se o movimento de empurrar é válido (se não empurra peças para fora do tabuleiro principal, por exemplo), é feito em "tempo real", uma vez que se falhar o backtracking trata de repor o tabuleiro no seu estado original.

Finalmente, o predicado responsável por empurrar a peça é o *push(+FinalIndex, +Row, -NewRow)*. Este predicado pega na linha desejada e empurra a peça a sua cabeça até ao valor definido por *FinalIndex*.

De notar que devido a rotações do tabuleiro, apenas precisamos de uma função aplicável a todos os movimentos por meio das transformações necessárias.

O *push* é chamado recursivamente, movendo a peça para a frente caso a posição seguinte esteja livre (*empty*), caso contrário, este predicado é chamado na peça que se encontra à sua frente para a obrigar a mover-se de igual forma.

Devido à forma como foram definidas as diferentes células, quando encontra uma das posições iniciais vazias, esta terá tipo *null* e dará erro ao tentarmos mover algo de volta para cima dele.

3.7 Final do Jogo

Não foi desenvolvido um predicado como *game_over(+Board, -Winner)*, devido à pecularidade da maneira como termina o nosso jogo.

Passamos a explicar: o nosso jogo termina quando ambos os jogadores passam a jogada consecutivamente. Inicialmente tínhamos, de facto, desenvolvido um predicado que, com recurso a *valid_moves* descrito no capítulo 3.4 verificava se esta lista se encontrava vazia para ambos os jogadores e retornaria verdadeiro após a contagem das maiores ilhas e determinação do vencedor. Contudo, correr este predicado após cada jogada tornar-se-ia custoso e pouco eficiente.

Assim sendo, abstraímos esta funcionalidade (testar se um jogador deve passar a jogada, pois o retorno do predicado que determina todas as jogadas

válidas foi uma lista vazia) em *pass_move(+Board, +Player)* e, em vez de fazermos todas as verificações a cada jogada que passa, implementamos também um contador do número das jogadas passadas consecutivamente. Este contador é passado como último argumento de *game_loop* que permite que ele pare quando chegar a 2.

Temos então um ciclo com comportamento modular que, caso o contador esteja a 2, termina o jogo determinando o vencedor, caso contrário, testará se a jogada **não** deve ser passada (caso em que um movimento é feito e o contador reposto 0).

Se o predicado *pass_move* foi bem sucedido na chamada anterior (se não havia jogadas para esse determinado jogador), este *game_loop* falha nessa chamada (pois é chamado na negativa) e o turno é passado à frente, incrementando o contador de número de passagens.

De notar que desta forma, por cada ciclo de jogo, temos apenas de, no máximo, verificar se existem jogadas válidas para um jogador, aumentando a eficiência do programa.

Para mais informações à cerca do *game_loop* ver a documentação presente no ficheiro *game.pl*. O predicado *pass_move(+Board, +Player)* encontra-se, junto com a sua documentação no *move.pl*.

A título representativo, a figura seguinte representa o funcionamento de um possível *game_over*

```
game_over(Board, Player, Opponent):-
        valid_moves(Board, Player, []),
        valid_moves(Board, Opponent, []).
```

Figura 6: Predicado exemplificativo de um possível game_over

3.8 Avaliação do Tabuleiro

Uma vez que a avaliação do tabuleiro é feita com base na maior área encontrada de um determinado tipo de peça, foi necessário implementar um algoritmo com a capacidade de detetar a maior área existente no tabuleiro de uma determinada peça.

Para isso, usaram-se algoritmos de *flood fill* com a capacidade de percorrer o tabuleiro em todas as direções de modo a encontrar todas as peças de um determinado tipo em todos os sentidos do tabuleiro. Esta capacidade é implementada pelo predicado *value(+Board, +Disc, -Points)* que tem a capacidade de atravessar o tabuleiro e de fazer *flooding* do mesmo, usando o predicado *board_flood(+Board, -NewBoard,+CurrentDisc, -Points, +Y, +X)* como um predicado auxiliar.

É deixado na seguinte figura uma demonstração do facto principal de *value*

```
value(Visited, Disc, Points, MaxPoints, Y, X):-
    get_element_matrix(Visited, Y, X, Elem),
    Elem \= visited,
    YDown is Y + 1,
    YUp is Y - 1,
    XRight is X + 1,
    XLeft is X - 1,
    replace_matrix(Y, X, visited, Visited, FirstVisited),
```

```
board_flood(FirstVisited, SecondVisited, Disc,
    DownPoints, YDown, X),
board_flood(SecondVisited, ThirdVisited, Disc,
    UpPoints, YUp, X),
board_flood(ThirdVisited, FourthVisited, Disc,
    RightPoints, Y, XRight),
board_flood(FourthVisited, FinalVisited, Disc,
    LeftPoints, Y, XLeft),
NewPoints is DownPoints + UpPoints + LeftPoints +
    RightPoints + 1,
max_points(NewPoints, Points, NewMax),
value(FinalVisited, Disc, NewMax, MaxPoints, YDown, X
    ).
```

Figura 7: Facto Principal do algoritmo de flood fill

3.9 Jogada do Computador

Uma vez que o nosso jogo possui 5 diferentes níveis de dificuldade, será necessário avaliar cada uma destas detalhadamente de forma a compreender o incremento da dificuldade.

3.9.1 Very Easy - Jogada random

Para o nosso primeiro nível foi implementado um algoritmo simples cujo o intuito será apenas encontrar todas as jogadas possíveis para um determinado tabuleiro, escolhendo **aleatoriamente** a jogada para o utilizador.

```
random_move(Board, Player, Move) :-
        valid_moves(Board, Player, Moves),
        length(Moves, NMoves),
        NMoves > 0,
        LastIndex is NMoves - 1,
        random_between(0, LastIndex, RandomIndex),
        nth0(RandomIndex, Moves, Move).
```

Figura 8: Escolha de uma jogada aleatória

3.9.2 Easy - Jogada Greedy de Máxima Pontuação

Na dificuldade Fácil é utilizado um algoritmo ganancioso que irá olhar para todas as jogadas do utilizador e determinar a melhor com base na melhor pontuação obtida pelo atual jogador.

```
greedy_max_move(Player, Opponent, Board, Moves, Move):-
        maplist(generate_move_points(Board, Player,
            Opponent), Moves, Scores, _),
        max_list(Scores, Max),
        nth0(Index, Scores, Max),
        nth0(Index, Moves, Move).
```

Figura 9: Escolha da melhor Jogada com base na Pontuação

3.9.3 Normal - Jogada Greedy de Máxima Diferença de Pontuações

Na dificuldade Normal é utilizado um algoritmo semelhante ao anterior, à excepção do facto que este determina a melhor jogada com base na diferença entre a pontuação obtida pelo Jogador e o seu Oponente.

```
greedy_difference_move(Player, Opponent, Board, Moves,
    Move):-
        maplist(generate_move_points(Board, Player,
            Opponent)
        , Moves, PlayerPoints, OpponentPoints),
        get_best_with_difference(Moves, PlayerPoints,
            OpponentPoints, Move).

get_best_with_difference(Moves, PlayerPoints,
    OpponentPoints, Move):-
        maplist(difference, PlayerPoints, OpponentPoints,
             PointsDifference),
        max_list(PointsDifference, MaxDifference),
        nth0(Index, PointsDifference, MaxDifference),
        nth0(Index, Moves, Move).
```

Figura 10: Escolha da melhor Jogada com base na Diferença de Pontuações

3.9.4 Hard - Jogada Minimax de Mínima Pontuação

Para a seguinte dificuldade foi implementado um algoritmo de Minimax de 1 nível de profundidade, onde este irá obter todas as jogadas possíveis pelo o utilizador, seguido de todas as jogadas do oponente na próxima iteração.

Com base nisto, é necessário aplicar um heurística para determinar a melhor jogada, sendo que é escolhida a jogada com **a pior pontuação para o oponente.**

```
minimax_worst_play(FirstDegreeMoves, _, OpponentPoints,
    Move):-
        min_list(OpponentPoints, WorstPoints),
        nth0(Index, OpponentPoints, WorstPoints),
        nth0(Index, FirstDegreeMoves, Move).

minimax(Board, Player, Func, Move):-
        valid_moves(Board, Player, FirstDegreeMoves),
        get_new_boards(Board, FirstDegreeMoves,
            FirstDegreeBoards, Player),
        NextPlayer is Player + 1,
        Opponent is mod(NextPlayer, 2),
        maplist(generate_board_points(Opponent, Player)
        , FirstDegreeBoards, OpponentPoints, PlayerPoints
            ),
        Pred=..[Func, FirstDegreeMoves, PlayerPoints,
            OpponentPoints, Move],
        Pred.
```

Figura 11: Escolha da melhor Jogada com base na Diferença de Pontuações

3.9.5 Very Hard - Jogada Minimax de Máxima Diferença de Pontuações

À semelhança da dificuldade anterior, esta dificuldade irá alcançar a jogada seguinte do oponente, utilizando uma heurística diferente (assim como a usada no algoritmo de dificuldade *Normal*), da Máxima Diferença de Pontuações.

```
get_best_with_difference(Moves, PlayerPoints,
    OpponentPoints, Move):-
        maplist(difference, PlayerPoints, OpponentPoints,
            PointsDifference),
        max_list(PointsDifference, MaxDifference),
        nth0(Index, PointsDifference, MaxDifference),
        nth0(Index, Moves, Move).

minimax(Board, Player, Func, Move):-
        valid_moves(Board, Player, FirstDegreeMoves),
        get_new_boards(Board, FirstDegreeMoves,
            FirstDegreeBoards, Player),
        NextPlayer is Player + 1,
        Opponent is mod(NextPlayer, 2),
        maplist(generate_board_points(Opponent, Player)
        , FirstDegreeBoards, OpponentPoints, PlayerPoints
            ),
        Pred=..[Func, FirstDegreeMoves, PlayerPoints,
            OpponentPoints, Move],
        Pred.
```

Figura 12: Escolha da melhor Jogada com base na Diferença de Pontuações

4 Conclusões

Em suma, o desenvolvimento deste projeto foi-nos capaz de transmitir as capacidade de pensamento associada a um paradigma completamente diferente do que é habitual em linguagens de programação, aplicado num contexto realista e desafiante.

É de notar de que numa realidade completamente dominada por paradigmas relativamente semelhantes (nomeadamente Imperativo, Orientado a Objetos e Funcional), a introdução do paradigma Declarativo, proporciona um desafio que obriga a uma abordagem completamente diferente, um de extrema relevância para o desenvolvimento das capacidades de qualquer Engenheiro Informático.

Não obstante, tal desafio também se revela como sendo um entrave no desenvolvimento de software, tornando uma tarefa aparentemente simples de realizar, num repto que atrasa taxativamente o desenvolvimento de algo aparentemente simples.

Os próximos passos no desenvolvimento do projeto seria implementar Minimax para vários níveis de profundidade, aplicando também cortes alpha-beta, de modo a conseguir adicionar níveis de maior dificuldade .

Bibliografia

[1] BoardGameGeek Fuse Page : https://boardgamegeek.com/boardgame/286350/fuse

[2] Nestorgames Fuse Page : https://nestorgames.com/fuse$_detail$

[3] Fuse Rulebook : https://nestorgames.com/rulebooks/FUSE$_EN.pdf$

[4] Minimax : https://en.wikipedia.org/wiki/Minimax

[5] SWI-Prolog Documentation - https://www.swi-prolog.org/pldoc