Quantum Leap

Relatório Final

Mestrado Integrado em Engenharia Informática e Computação

Programação em Lógica

Grupo T3_Quantum_Leap_4:

Márcio Filipe Vilela Fontes - ei12183@fe.up.pt
Rui Pedro Alves de Sousa e Costa Andrade - ei12010@fe.up.pt

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, s/n, 4200-465 Porto, Portugal

11 de Novembro de 2014

Resumo

O desenvolvimento de jogos de tabuleiro é uma atividade constante, utilizando linguagens de programação em lógica, muito úteis para implementação das potencialidades da inteligência artificial. As diferentes regras dos variadíssimos jogos de tabuleiro existentes, tornam desafiante a implementação da sua inteligência artificial.

Este projeto incide sobre o jogo Quantum Leap, um dos propostos pelos docentes da unidade curricular de Programação em Lógica. As funcionalidades da aplicação desenvolvida centram-se na representação de um tabuleiro hexagonal, composto por células que serão populadas pelos dois jogadores (isto no modo Jogador vs. Jogador), a interface de movimentos que cada jogador poderá realizar (quer no modo Jogador vs. Jogador ou Jogador vs. Computador), e por último no modo Computador vs. Computador, o nível de "inteligência"que a máquina deverá ter em consideração ao realizar uma jogada.

Assim, o resultado deste trabalho é uma aplicação baseada no jogo Quantum Leap, capaz de fornecer uma interface simples ao utilizador e, no modo de jogo Máquina vs. Máquina, já utilizando uma pequena parte de inteligência artificial, a aplicação é capaz também de realizar, definindo o nível de "inteligência", jogadas mais idênticas às que um ser humano pudesse realizar, tendo em conta a sua capacidade de análise e previsão de acontecimentos.

Palavras-Chave: Tabuleiro; Peça; Prolog; Inteligência Artificial

Abstract

The development of board games is a constant activity, using logic programming languages, which are very useful for implementing the outstanding qualities of artificial intelligence. The different rules of the widely varying board games, makes it challenging to implement its artificial intelligence.

This project focuses on the game Quantum Leap, proposed by the teachers of the course of Programação em Lógica. The application's features developed focus on the representation of an hexagonal board, composed of cells that are populated by the two players (Player vs. Player mode), the interface moves each player may do (either Player vs. Player or Player vs. Computer), and lastly in Computer vs. Computer mode, the level of "intelligence"that the machine should consider when making a move.

So the result of this work is an application based on the game Quantum Leap, able to provide a simple interface to the user, and in game mode Machine vs. Machine, only using a small piece of artificial intelligence, the application is also able to perform, defining the level of "intelligence", more similar moves that a human being could accomplish, given their capacity for analysis and prediction of events.

Keywords: Board; Piece; Prolog; Artificial Intelligence

Conteúdo

Lista de Figuras

1 Introdução

Este relatório está associado e pretende demonstrar o desenvolvimento do projeto *Quantum Leap* para a unidade curricular de Programação em Lógica onde será caracterizado, definido e analisado todos os aspetos referentes ao mesmo, sendo nele descrito todo o processo de implementação da aplicação.

1.1 Motivação

A escolha deste projeto teve como principal motivação, não só a aparente dificuldade de representação e execução de movimentos por parte de cada jogador, mas também a implementação da "inteligência"necessária para aproximar a realidade de que a máquina possa realizar jogadas tão "perfeitas"como um ser humano.

1.2 Objetivos

Com o desenvolvimento deste projeto pretende-se adquirir conhecimentos mais coesos de programação em lógica, novas técnicas de implementação, uma aplicação (ainda que diminuta) de inteligência artificial e facilidade de abordagem do problema em questão em futuros projetos. Não obstante, pretende-se no final deste projeto que se cumpra aquilo que foi estipulado, mantendo acima de tudo a qualidade exigida, quer pelos próprios alunos que desenvolveram a aplicação, quer pelos docentes da unidade curricular.

1.3 Estrutura do Relatório

O presente relatório está dividido nas seguintes secções:

- **O Jogo *Quantum Leap*** - descrição sucinta do jogo, a sua história e as suas regras.
- **Lógica do Jogo** - descrição do projeto e implementação da sua lógica de jogo em Prolog.
- **Interface com o Utilizador** - descrição do módulo de interface com o utilizador em modo de texto.
- **Considerações Finais** - descrição das conclusões e desenvolvimentos futuros.

2 O Jogo *Quantum Leap*

2.1 Introdução

O *Quantum Leap*[1] é um jogo de tabuleiro de forma hexagonal, para dois jogadores, podendo este ser estruturado de qualquer forma ou tamanho.[2]

No *Quantum Leap*, as pedras de cada jogador começam dispersas no tabuleiro. O objetivo do jogo é ser o último jogador a conseguir fazer uma jogada válida. As pedras podem unicamente mover-se para capturar as pedras inimigas, saltando para a posição da pedra inimiga escolhida. Estas ganham o seu "poder de salto"pelo número de pedras "amigas"que estão imediatamente à sua volta.

2.2 Preparação do Jogo

De forma aleatória deve-se distribuir todas as pedras (neste caso, 30 de cada cor) nas respetivas células livres (61), de forma que cada célula contenha uma única pedra e exista um único espaço livre. Este espaço livre pode localizar-se em qualquer lugar do tabuleiro, exceto no centro deste (para evitar a possibilidade rara de haver simetrias no tabuleiro).

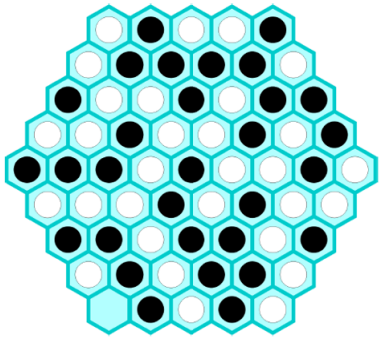


Figura 1: Exemplo de Tabuleiro Inicial

[1] Informação obtida em http://www.boardgamegeek.com/filepage/89482/official-rules-english

[2] Para efeitos de demonstração, utilizaremos um tabuleiro de 5 hexágonos de lado, 30 pedras brancas e 30 pedras pretas.

2.3 Como Jogar

Cada jogador tem uma respetiva cor (BRANCO ou PRETO).

Antes do jogo começar, o jogador que tiver escolhido a cor PRETA pode trocar as posições de quaisquer duas pedras presentes no tabuleiro.

O outro jogador (cor BRANCA) começa o jogo. Os jogadores alternam de turnos durante o jogo até que haja alguém que não consiga efetuar uma jogada válida, perdendo assim o jogo.

Em cada turno terá de haver uma captura de uma pedra inimiga. Uma pedra efetua uma captura ao saltar em linha, em quaisquer das 6 direções possíveis, exatamente tantos espaços quantas as pedras "amigas"estiverem à sua volta, posicionando-se na célula da pedra inimiga que é removida do jogo. Qualquer pedra pode saltar sobre outras pedras. Abaixo segue-se uma figura ilustrativa de um movimento de jogo.

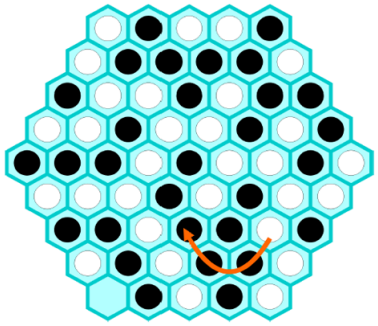


Figura 2: Exemplo de captura. A pedra BRANCA tem à sua volta 2 pedras "amigas", portanto terá de saltar exatamente duas células para capturar a pedra PRETA.

Caso a um jogador não seja possível fazer a captura num dado turno, este perde o jogo. Abaixo segue-se uma figura que demonstra um estado possível de final de jogo.

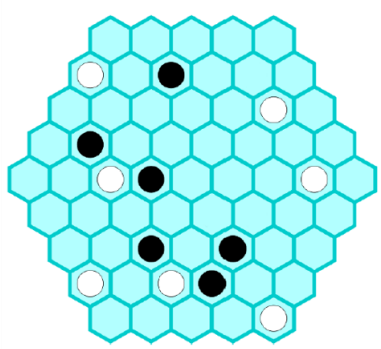


Figura 3: Exemplo de fim de jogo. O jogador das pedras BRANCAS não consegue fazer uma jogada válida e assim perde o jogo.

3 Lógica do Jogo

3.1 Representação do Estado do Jogo

O tabuleiro é representado segundo uma lista de listas $L = \{L_1, L_2, ..., L_N\}$, em que N é o número de níveis[3] do hexágono.

No exemplo utilizado, o tabuleiro tem comprimento de lado igual a 5, o que implica que este possui 9 níveis ou, dito de outra forma, uma lista de 9 listas em Prolog. Abaixo segue-se um código exemplo da criação de um novo tabuleiro.

```
makeBoard( Board, Spaces ) :-
  Size is 5,
  makeDinBoard( Board, Spaces, 0, Size ).
```

A figura acima demonstra o estado do tabuleiro na fase inicial, sem nenhuma célula preenchida. De uma forma mais visual, segue-se abaixo uma figura demonstrativa do tabuleiro (no seu estado inicial).

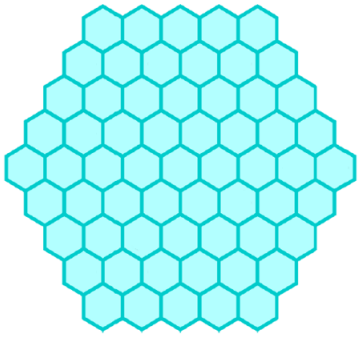


Figura 4: Exemplo de um tabuleiro no seu estado inicial, sem nenhuma célula preenchida.

Como já foi referido na secção 2.2, as pedras devem ser distribuídas aleatoriamente pelo tabuleiro, de forma que o jogador que tiver escolhido as pedras PRETAS possa proceder à troca de quaisquer duas pedras presentes no tabuleiro (ver secção 2.3). Abaixo segue-se um possível exemplo, da respetiva definição em Prolog, de um tabuleiro preenchido aleatoriamente (número 1 representa as pedras PRETAS e número 2 as pedras BRANCAS).

[3] O número de níveis é dado como $N = 2k - 1$, em que k é o comprimento de cada lado do hexágono.

```
makeBoard( Board, Spaces ) :-
  Size is 5,
  makeDinBoard( NewBoard, Spaces, 0, Size ),
  numPieces( Num, Size ),
  fillBoard( NewBoard, Board, Num, Num ).
```

De uma forma visual, o tabuleiro apresentaria o aspeto a seguir:

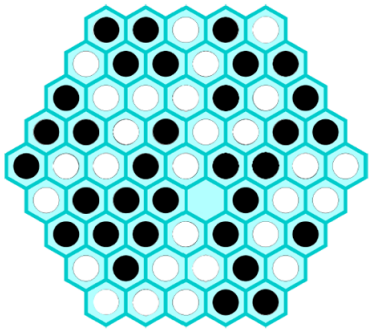


Figura 5: Tabuleiro preenchido aleatoriamente a partir da execução do código anterior.

Após o jogador ter trocado a posição de algumas pedras (caso deseje, utilizando o predicado `edit_board(Board, Spaces, ResultBoard)`), o outro jogador (pedras BRANCAS) deverá começar o jogo. De forma hipotética, uma possível jogada poderia ser a apresentada abaixo:

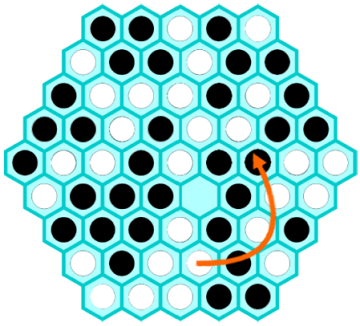


Figura 6: Exemplo de 1ª jogada pelas pedras BRANCAS.

Após uma dada série de jogadas, alternadamente pelos dois jogadores, utilizando o predicado `movePiece(Board, ResultBoard, Line, Col, DLine, DCol)`, um possível estado final do tabuleiro poderá ser o representado abaixo:

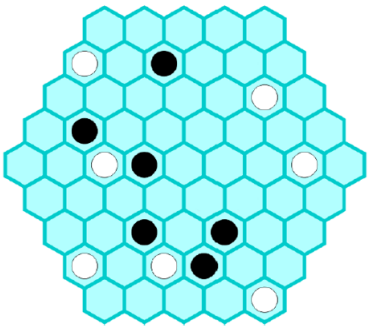


Figura 7: Exemplo do estado final do tabuleiro num jogo.

Após o final do jogo, a aplicação termina e, caso seja vontade dos jogadores jogarem novamente, então terão executar o predicado `start` especificando *a priori* o respetivo tamanho do tabuleiro.

3.2 Visualização do Tabuleiro

Tal como está descrito na secção 3.1, o predicado `makeBoard(Board, Spaces)` permite gerar um novo tabuleiro. Assumindo que, ao início, este tem todas as suas células vazias, utilizando o predicado `printBoard(Board, Spaces)`, o *output* esperado será seguinte (ignorando para já a numeração das linhas e colunas):

```
     [ _ _ _ _ _ ]
    [ _ _ _ _ _ _ ]
   [ _ _ _ _ _ _ _ ]
  [ _ _ _ _ _ _ _ _ ]
 [ _ _ _ _ _ _ _ _ _ ]
  [ _ _ _ _ _ _ _ _ ]
   [ _ _ _ _ _ _ _ ]
    [ _ _ _ _ _ _ ]
     [ _ _ _ _ _ ]
```

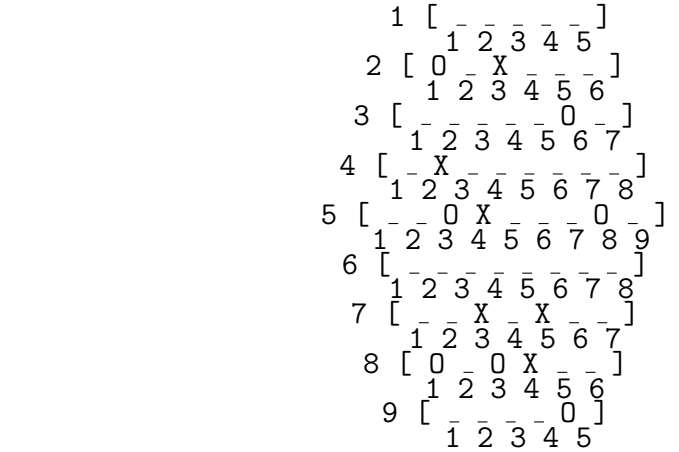
De seguida, e utilizando o seguimento de execução da secção 3.1, neste caso relativamente à Figura 5, recorrendo novamente ao predicado para imprimir no ecrã o tabuleiro, obtém-se:

```
    1 [ X X O X O ]
        1 2 3 4 5
  2 [ O X X O X X ]
      1 2 3 4 5 6
 3 [ X O O O X O X ]
    1 2 3 4 5 6 7
4 [ X X O X O O X X ]
  1 2 3 4 5 6 7 8
5 [ X O O X O X X O O ]
  1 2 3 4 5 6 7 8 9
 6 [ O X X X _ X O O ]
    1 2 3 4 5 6 7 8
 7 [ X X X O X O X ]
    1 2 3 4 5 6 7
  8 [ O X O O X O ]
      1 2 3 4 5 6
   9 [ O O O X X ]
       1 2 3 4 5
```

É de notar que a declaração do tabuleiro é realizada apenas com os números 0 (indicando célula vazia, com o carácter _), 1 (para pedras PRETAS, com o carácter `X`) e 2 (para pedras BRANCAS, com o carácter `O`). A tradução desses números para os caracteres, é realizada pelo predicado `translate(Number, Character)` que possui os seguintes três factos:

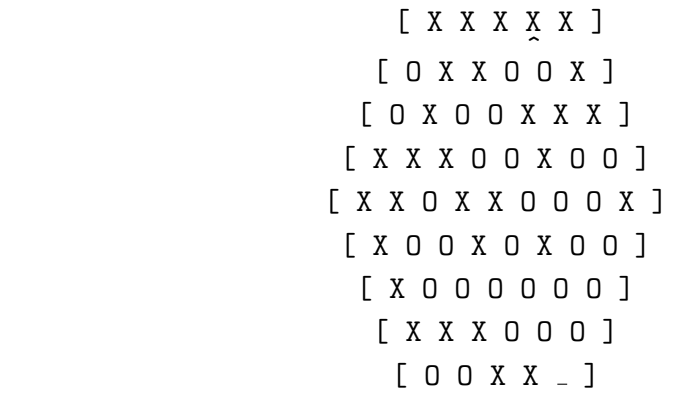
```
translate(0, '_').
translate(1, 'X').
translate(2, 'O').
```

Como tal, no final de um jogo, a visualização do tabuleiro terá o seguinte aspeto (baseado na Figura 7):



Como é possível verificar, não é possível mover qualquer pedra, tendo em conta as regras do jogo, previamente explicitadas na secção 2.1.

O predicado que irá gerir o início de jogo e consequente computação do tabuleiro, visualização, etc. será o `startGame(Size).` que receberá como argumento o tamanho do tabuleiro, tendo este como único objetivo iniciar um novo jogo e gerir o *loop* deste. Abaixo segue-se uma representação do tabuleiro no ato de escolha da jogada/posição para onde mover (já depois de ter sido escolhida a peça a mover-se).



Esta representação não mostra a numeração das linhas e colunas, no entanto, a sua única funcionalidade é dar ênfase à peça que irá ser movida (colocando um símbolo por baixo desta). Mais informação será detalhada na secção 4 sobre a interface com o utilizador.

3.3 Lista de Jogadas Válidas

A Lista de Jogadas Válidas pode ser apresentada no ecrã, utilizando o predicado `listMoves(Board, Line Col, MovesList)`. Esta funcionalidade é particularmente útil, de modo a mostrar ao utilizador os movimentos que uma dada peça localizada no ponto `(Line, Col)` de `Board` pode realizar.

O resultado pretendido é retornado na lista `MovesList` e apresentado no ecrã sob a forma de uma lista de listas, em que cada elemento é uma posição `(l,c)` possível.

Utilizando o tabuleiro já definido na secção 3.2, se pretendêssemos mover a peça localizada em `(3,4)` utilizando o predicado

```
listMoves(Board, 3, 4, MV),
```

então a resultado esperado em `MV` seria:

```
MV = [[3|1],[3|7],[6|3],[6|6]].
```

Como se pode concluir através da figura representativa do tabuleiro, a peça localizada em `(3,4)` tem três peças da mesma cor à sua volta. Como tal, esta só poderá mover-se para uma dada posição (que contenha uma peça de cor adversária) numa distância de três unidades.

As únicas posições possíveis seriam (tendo apenas em conta a distância d):

$$MV_d = \{(0,1),(0,4),(3,1),(3,7),(6,3),(6,6)\}$$

.

Como as posições do tabuleiro começam em `(1,1)`, seria matematicamente impossível uma peça mover-se para a posição `(0,1)` ou `(0,4)`.

Aliás, estes casos nem sequer são testados porque falham imediatamente uma condição crucial do algoritmo, que é o teste dos limites do tabuleiro.

Assim, resta-nos apenas as células `(3,1),(3,7),(6,3),(6,6)`. Como não basta verificar apenas a distância entre as peças, é necessário prosseguir para o teste da cor da célula para onde se pretende mover. Neste caso particular, todas as células restantes contêm peças de cor diferente àquela que pretendemos mover. Assim sendo, consegue-se concluir que efetivamente as células esperadas são iguais à lista de células retornada pelo predicado `listMoves`.

Para terminar, se a peça fosse movida para `(3,1)`, então o tabuleiro esperado seria:

```
    1 [ X X O X O ]
        1 2 3 4 5
  2 [ O X X O X X ]
      1 2 3 4 5 6
  3 [ O O O _ X O X ]
    1 2 3 4 5 6 7
 4 [ X X O X O O X X ]
     1 2 3 4 5 6 7 8
5 [ X O O X O X X O O ]
   1 2 3 4 5 6 7 8 9
 6 [ O X X X _ X O O ]
     1 2 3 4 5 6 7 8
  7 [ X X X O X O X ]
      1 2 3 4 5 6 7
   8 [ O X O O X O ]
       1 2 3 4 5 6
    9 [ O O O X X ]
        1 2 3 4 5
```

3.4 Execução de Jogadas

3.4.1 1ª Fase

Tal como indicam as regras do jogo (ver secção 2.3), é dado ao jogador das pedras PRETAS a escolha de poder trocar quaisquer duas pedras no tabuleiro. Nesta fase, o predicado `edit_board(Board, Spaces, ResultBoard)`, terá como objetivo trocar duas pedras:

$$S_1 = (l_1, c_1)$$

e

$$S_2 = (l_2, c_2)$$

em que l_i indica a linha e c_i a respetiva coluna.

A pedra S_1 tomará o lugar de S_2, sendo esta última removida do `Board` atual e, no final da execução do predicado, será retornado um novo tabuleiro `ResultBoard`, já com a pedra S_2 removida.

Com isto, termina a fase de *switching pieces* e inicia-se a fase de jogadas (começando o jogador das pedras BRANCAS).

3.4.2 2ª Fase

Após a fase de troca de pedras por parte do jogador das pedras PRETAS (ver secção 3.4.1), segue-se realmente para o início do jogo.

Já referido anteriormente na secção 2.3, o jogador das pedras BRANCAS faz a primeira jogada e assim se vai alternando, até que haja um jogador que não consiga fazer uma jogada válida. Isso é controlado por um predicado denominado:

```
movePiece(Board, ResultBoard, Line, Col, DLine, DCol).
```

O objetivo deste predicado é realizar uma jogada - saltar para uma pedra adversária - tendo em conta a `DLine` e `DCol` pretendidas. Como uma pedra poderá saltar várias posições e, mais importante ainda, por cima de outras pedras, é necessário indicar a direção para onde deverá mover-se. Para efeitos de controlo de erros, também será necessário saber qual a pedra `Piece` que irá mover-se (dada por `Line` e `Col`), de forma a garantir que um dado jogador ao selecionar uma pedra não selecione alguma que não é sua. De uma forma mais matemática, assumindo que o conjunto das pedras do jogador 1 (pedras PRETAS) é igual ao seguinte:

$$J_1 = \{P_0, P_1, ..., P_n\}$$

e que o conjunto das pedras do jogador 2 (pedras BRANCAS) é igual a

$$J_2 = \{B_0, B_1, ..., B_n\}$$

então deverá garantir-se que, caso seja o jogador 1 a realizar a jogada, então a pedra $S_i \in J_1$, e no caso do jogador 2, $S_j \in J_2$, sendo $S_i \neq S_j$ obrigatoriamente visto que $J_1 \cap J_2 = \emptyset$.

O predicado que permite obter a pedra numa dada posição (l, c) é o que se segue:

```
getPiece(Board, Line, Col, Piece)
```

ou seja, a pedra p numa dada posição (l, c) irá ser retornada em `Piece`, caso esta exista, visto que a posição escolhida pode não existir (l ou c fora dos limites do tabuleiro) ou poderá ser uma célula vazia.

Após garantir-se que a jogada é válida, procede-se à remoção da pedra adversária e retorno do novo tabuleiro, sendo este armazenado em `ResultBoard`.

3.5 Avaliação do Tabuleiro

Para efeitos de interface, facultamos automaticamente as jogadas possíveis da peça escolhida pelo utilizador. Ou seja, esta funcionalidade não é algo que o utilizador possa fazer, mas sim que já lhe é fornecido, visto que, após análise do que seria prioritário aparecer no ecrã, chegou-se à conclusão que a avaliação do tabuleiro deveria ser efetuada logo após a escolha de uma peça, e o respetivo resultado ser mostrado.

Caso o resultado seja nulo, i.e, se não houver jogadas possíveis para uma certa peça, então o utilizador é informado sobre essa situação e a tentativa de jogada é considerada inválida.

Prosseguindo para um exemplo prático, se o tabuleiro gerado no modo `Human vs Human` (ver secção 4.1) for o representado abaixo (assume-se neste momento que será a turno das pedras PRETAS):

```
    1 [ X X X X X ]
        1 2 3 4 5
  2 [ O X X O O X ]
      1 2 3 4 5 6
3 [ O X O O X X X ]
    1 2 3 4 5 6 7
 4 [ X X X O O X O O ]
    1 2 3 4 5 6 7 8
5 [ X X O X X O O O X ]
  1 2 3 4 5 6 7 8 9
 6 [ X O O X O X O O ]
    1 2 3 4 5 6 7 8
  7 [ X O O O O O O ]
      1 2 3 4 5 6 7
    8 [ X X X O O O ]
        1 2 3 4 5 6
      9 [ O O X X _ ]
          1 2 3 4 5
```

e, depois ser escolhida a peça (6,6), então o utilizador será informado que não poderá realizar nenhuma jogada com essa peça, e poderá escolher novamente outra para movimentar.

```
That piece has no valid moves!  Please pick a different piece.
```

No caso de o utilizador tentar escolher uma peça adversária, então deverá ser informado que não poderá realizar nenhuma jogada, sob a seguinte mensagem no ecrã:

```
Pick one of your own pieces!
```

Já no caso de o utilizador escolher acertadamente uma peça sua e que consiga ser movimentada, então será mostrado no ecrã as várias que o jogador poderá realizar, que será descrito com mais detalhe numa subsecção de Interface com o Utilizador (ver secção 4.1).

Toda esta análise do tabuleiro é realizada pelo predicado que permite listar todos os movimentos que uma peça localizada em `(Line, Col)` poderá realizar

```
listMoves(Board, Line, Col, MovesList)
```

e o predicado que testa para cada direção e distância, se é possível mover a peça

```
canMove(Board, Line, Col, Dir, N, NewLine, NewCol).
```

O retorno de `listMoves` será posteriormente utilizado pelo predicado `printMoves` para mostrar ao jogador as respetivas jogadas possíveis de cada peça. Nesse momento será pedido ao utilizador para continuar o processo de execução de uma jogada, escolhendo para quer mover a sua peça (utilizando claramente a lista devolvida por `listMoves`).

3.6 Final do Jogo

Nesta secção será descrito como é testado o final de jogo. Adicionalmente aos que já foram descritos, teremos predicados auxiliares tais como:

```
playerMoves(Board, Piece, Valid)
numNearPiece(Board, Line, Col, Num)
```

Relativamente ao primeiro predicado, a sua funcionalidade será testar se um jogador pode mover uma certa pedra `Piece` para uma dada posição (útil para testar o final de jogo, assumindo que já se testou previamente todas as outras pedras, visto que caso este predicado retorne *no*, implica que o jogador não pode fazer nenhuma jogada válida, perdendo assim o jogo).

Já no segundo predicado, o seu objetivo será retornar em `Num` o número de peças adjacentes à peça escolhida. Como tal, será necessário testar, para cada uma das seis direções possíveis[4], se a pedra que foi selecionada poderá mover-se para uma dada célula do tabuleiro.

Num exemplo concreto da figura abaixo, assume-se que seja o turno das pedras BRANCAS. O predicado `numNearPiece` irá retornar em `Num` as três posições para as quais se pode mover (indicadas pela seta):

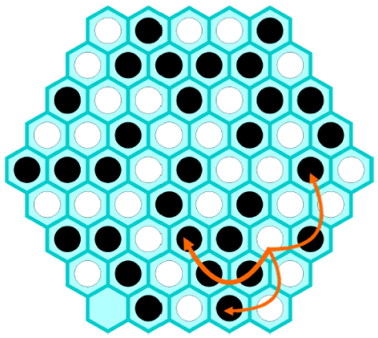


Figura 8: Representação gráfica do que será retornado em `Result`. As setas indicam as posições para as quais a pedra BRANCA se pode mover.

[4]Considera-se como direções possíveis as seguintes: Cima, Baixo, Cima-Esquerda, Cima-Direita, Baixo-Esquerda e Baixo-Direita. No entanto, caso estejamos a testar alguma pedra nas extremidades do tabuleiro, o número de direções possíveis reduz para três.

De uma forma mais matemática, seja $P = \{p_1, p_2, ..., p_n\}$ o conjunto de todas as pedras atuais do jogador, p a pedra que se pretende mover, k o número de pedras "amigas"[5] na sua vizinhança e $C = \{c_1, c_2, ..., c_m\}$ o conjunto das pedras que distam de k posições linearmente, num caso em que a pedra p não esteja nas extremidades do tabuleiro. O conjunto retornado pelo predicado será então: $R = C \backslash P$.

Generalizando o conceito, se $\forall p_i \in P \Rightarrow R_i = \emptyset$, então podemos concluir que o jogador não consegue realizar nenhuma jogada válida, terminando assim o jogo. Abaixo segue-se uma representação de final de jogo, descrita mais em detalhe na secção 4.

```
    1 [ _ _ _ _ _ ]
      1 2 3 4 5
  2 [ O _ X _ _ _ ]
      1 2 3 4 5 6
  3 [ _ _ _ _ _ O _ ]
      1 2 3 4 5 6 7
  4 [ _ X _ _ _ _ _ ]
      1 2 3 4 5 6 7 8
5 [ _ _ O X _ _ _ O _ ]
    1 2 3 4 5 6 7 8 9
  6 [ _ _ _ _ _ _ _ ]
      1 2 3 4 5 6 7 8
  7 [ _ _ X _ X _ _ ]
      1 2 3 4 5 6 7
  8 [ O _ O X _ _ ]
      1 2 3 4 5 6
    9 [ _ _ _ _ O ]
        1 2 3 4 5
Player X can't make any more moves!  Player O wins!
```

[5] da mesma cor

3.7 Jogada do Computador

Tal como foi sugerido pelos docentes da unidade curricular, implementou-se dois tipos de inteligência/dificuldade no jogo. Uma poderá denominar-se de fácil (menos inteligente) e outra de difícil (mais inteligente).

O mais prioritário da jogada do computador terá a ver as diferenças de implementação dos algoritmos. De uma forma simples, sem entrar em detalhes programáticos, segue-se abaixo uma breve descrição sobre o modo de funcionamento das respetivas inteligências:

- **Easy Mode** - neste modo, o objetivo do computador é encontrar a primeira jogada possível de ser realizada e executá-la imediatamente, sem dar importância ou testar quaisquer outras soluções.

- **Hard Mode** - neste modo, o objetivo do computador é procurar uma jogada que garanta imediatamente a vitória do jogo, i.e, terá de testar todas as possíveis soluções e verificar se alguma delas garante a vitória. Caso não a encontre, utiliza-se o mesmo método que no **Easy Mode**.

Antes de implementarmos estes algoritmos, especulamos se seria viável desenvolver uma inteligência de tal forma que o computador fosse "prevendo"ou "aprendendo"com as jogadas do utilizador. No entanto, para tornar essa ideia real, seria necessário bases de dados para armazenar as jogadas e os respetivos resultados ou caminhos a que levaram, e aumentaria de forma exponencial a complexidade do tema em questão.

Como tal, foi decidido que a jogada do computador teria uma vertente mais programática (mais simples de implementar) e não tanto uma vertente mais matemática (mais simples de formular, no entanto, mais difícil de implementar).

Na secção de Anexos (ver secção 6), é presente o código fonte de cada uma das inteligências utilizadas no jogo, para melhorar entendimento do paradigma utilizado.

4 Interface com o Utilizador

O objetivo principal da interface desenvolvida foi tornar uma interação simples, fácil de manipular as peças do tabuleiro e uma visualização que seja rápida de entender pelo utilizador, quer para efeitos das suas peças e das do adversário, quer para efeitos de movimentação das mesmas. Não obstante, teve-se o cuidado da própria leitura de dados a partir do teclado (movimentos das peças) ser realizada de forma quase imediata.

Para iniciar a aplicação é necessário chamar o predicado `start(Size)`. Depois de ser escolhido o tamanho do tabuleiro é disponibilizado ao utilizador um menu de apresentação e escolha do modo de jogo.

```
Please pick a game mode by writing one of the options.
1.  Human vs Human
2.  Human vs Computer
3.  Computer vs Computer
```

4.1 Human vs. Human

Neste modo, tal como as regras ditam, o primeiro jogador será o das pedras PRETAS que começará por, caso queira, trocar peças do tabuleiro gerado aleatoriamente. Segue-se abaixo um exemplo ilustrativo do início de um jogo.

```
Player O (2) may now edit the Board at will.
          1 [ X X X X X ]
              1 2 3 4 5
        2 [ O X X O O X ]
            1 2 3 4 5 6
      3 [ O X O O X X X ]
          1 2 3 4 5 6 7
    4 [ X X X O O X O O ]
        1 2 3 4 5 6 7 8
  5 [ X X O X X O O O X ]
      1 2 3 4 5 6 7 8 9
    6 [ X O O X O X O O ]
        1 2 3 4 5 6 7 8
      7 [ X O O O O O O ]
          1 2 3 4 5 6 7
        8 [ X X X O O O ]
            1 2 3 4 5 6
          9 [ O O X X _ ]
              1 2 3 4 5

Pick two pieces to swap.
If both pieces are out of the board (like 0-0 for example)
the swapping will end.
```

Depois da fase de *swapping pieces*, passar-se-á realmente para o início do jogo. Abaixo segue-se o que será apresentado ao utilizador (assumindo que nenhuma peça foi movida na fase anterior).

```
    1 [ X X X X X ]
        1 2 3 4 5
  2 [ O X X O O X ]
      1 2 3 4 5 6
 3 [ O X O O X X X ]
     1 2 3 4 5 6 7
4 [ X X X O O X O O ]
    1 2 3 4 5 6 7 8
5 [ X X O X X O O O X ]
   1 2 3 4 5 6 7 8 9
 6 [ X O O X O X O O ]
    1 2 3 4 5 6 7 8
  7 [ X O O O O O O ]
      1 2 3 4 5 6 7
   8 [ X X X O O O ]
       1 2 3 4 5 6
    9 [ O O X X _ ]
        1 2 3 4 5

Turn to play:  X
Please pick a piece to move.
```

De seguida, é dado ao utilizador o conhecimento (em cada jogada) sobre como deverá ser o *input* de escolha da peça a mover.

```
Please enter coordinates in the format X-Y.
X is the position along the line, and Y is the line.
```

Se o utilizador pretender, por exemplo, mover a peça na posição `(1,3)` então deverá introduzir `3-1` como *input*. Matematicamente, para uma célula (l, c) deverá introduzir como *input* o valor $c - l$.

Ao introduzir o valor `3-1.` como *input*, irá ser mostrado no ecrã as possíveis jogadas/células para onde o jogador poderá mover a peça escolhida anteriormente. Adicionalmente, é colocado um símbolo por baixo da célula que se pretende mover (neste ponto não se mostra ao utilizador as posições ao lado do tabuleiro por questões visuais que, experimentalmente, não eram benéficas para a interface).

A seguir, é disponibilizado uma figura representativa do estado do tabuleiro após a escolha da peça pretendida.

```
[ X X X X X ]
[ O X X O O X ]
[ O X O O X X X ]
[ X X X O O X O O ]
[ X X O X X O O O X ]
[ X O O X O X O O ]
[ X O O O O O O ]
[ X X X O O O ]
[ O O X X _ ]

Please pick one of the following possible moves:
[4, 4]
[4, 7]
```

Depois de ser disponibilizado ao utilizador as possíveis jogadas que este pode fazer com a peça referenciada anteriormente, este deverá introduzir uma das jogadas que lhe foram apresentadas. Hipoteticamente, se o utilizador escolher a posição `(4,7)`, então mais uma vez deverá introduzir como *input* o valor `7-4`.

A aplicação irá interpretar o valor introduzido e irá mover a peça para a posição escolhida, removendo dessa forma a peça adversária, e colocando na posição de onde a peça se moveu, uma célula vazia.

```
1 [ X X X _ X ]
    1 2 3 4 5
2 [ O X X O O X ]
    1 2 3 4 5 6
3 [ O X O O X X X ]
    1 2 3 4 5 6 7
4 [ X X X O O X X O ]
    1 2 3 4 5 6 7 8
5 [ X X O X X O O O X ]
    1 2 3 4 5 6 7 8 9
6 [ X O O X O X O O ]
    1 2 3 4 5 6 7 8
7 [ X O O O O O O ]
    1 2 3 4 5 6 7
8 [ X X X O O O ]
    1 2 3 4 5 6
9 [ O O X X _ ]
    1 2 3 4 5

Turn to play:  O
Please pick a piece to move.
```

Este processo repetir-se-á até que finalmente um dos jogadores seja incapaz de realizar qualquer jogada. Quando tal acontecer, é mostrado ao utilizador o estado final do tabuleiro e o respetivo vencedor, terminando assim a aplicação.

```
    1 [ _ _ _ _ _ ]
      1 2 3 4 5
  2 [ O _ X _ _ _ ]
      1 2 3 4 5 6
  3 [ _ _ _ _ _ O _ ]
      1 2 3 4 5 6 7
  4 [ _ X _ _ _ _ _ _ ]
      1 2 3 4 5 6 7 8
5 [ _ _ O X _ _ _ O _ ]
    1 2 3 4 5 6 7 8 9
  6 [ _ _ _ _ _ _ _ _ ]
      1 2 3 4 5 6 7 8
  7 [ _ _ X _ X _ _ ]
      1 2 3 4 5 6 7
  8 [ O _ O X _ _ ]
      1 2 3 4 5 6
    9 [ _ _ _ _ O ]
      1 2 3 4 5
Player X can't make any more moves!  Player O wins!
```

No caso de o utilizador escolher a opção de `Human vs Computer`, então a interface será muito idêntica, como será descrita a seguir.

4.2 Human vs. Computer

Relativamente a esta interface, aquilo que é mostrado ao utilizador e os próprios que este pode executar, não são muito diferentes do modo `Human vs Human`.

Ao contrário do que é realizado no modo anterior, não haverá fase de *swapping pieces*, ou seja, passar-se-á logo à fase de escolha da peça por parte do jogador. Quando este decidir qual deve mover e para onde moverá, então o computador irá decidir qual será a melhor opção na sua jogada, e este processo repetir-se-á, tal como no modo `Human vs Human` até que haja um vencedor.

Também, ao contrário do modo anterior, após a definição do tamanho do tabuleiro e escolha deste modo, é necessário o utilizador escolher uma das seguintes "inteligências":

1. `Computer doesn't think before playing.`
2. `Computer thinks minimally before acting.`

A primeira "inteligência"é considerada como fácil e a segunda como difícil, embora, tal como foi sugerido pelos docentes da unidade curricular, a capacidade de "inteligência"e previsão de acontecimentos não estará segundo as "normas"ou técnicas que serão lecionadas na unidade curricular de Inteligência Artificial, visto que ainda não tivemos oportunidade de a frequentar.

A título de exemplo, utilizar-se-á um tabuleiro com comprimento de lado de 3 unidades para tornar mais fácil de visualizar, visto que será muito idêntico ao que já foi referenciado no outro modo de jogo. O tabuleiro será apresentado da mesma forma e a escolha de peça e movimentação será exatamente igual, como é demonstrado abaixo.

```
  1 [ X O X ]
      1 2 3
  2 [ X O O X ]
      1 2 3 4
3 [ O X X O X ]
    1 2 3 4 5
  4 [ O X O O ]
      1 2 3 4
    5 [ O X _ ]
        1 2 3

Turn to play:  X
Please pick a piece to move.
```

Se o jogador mover a peça `(2,4)` para a posição `(2,2)`, então o tabuleiro resultante será

```
  1 [ X _ X ]
      1 2 3
  2 [ X X O _ ]
      1 2 3 4
3 [ O X X O X ]
    1 2 3 4 5
  4 [ O X O O ]
      1 2 3 4
    5 [ O X _ ]
        1 2 3
```

Se fosse no modo `Human vs Human`, então seria a vez das pedras BRANCAS jogarem. No entanto, como o modo de jogo é contra o computador, então esse deverá assumir a jogada a ser realizada.

Pelo algoritmo "inteligente"que foi implementado, a jogada realizada pelo computador foi a representada pelo tabuleiro que se segue:

```
Turn to play:  O
  1 [ X _ X ]
      1 2 3
 2 [ X O O _ ]
     1 2 3 4
3 [ O X X O X ]
    1 2 3 4 5
 4 [ O X O O ]
     1 2 3 4
  5 [ O X _ ]
      1 2 3
```

Tal como já referido anteriormente, o processo repetir-se-á da mesma forma até que haja vencedor, mostrando a mesma mensagem que no modo `Human vs Human`.

4.3 Computer vs. Computer

Neste modo, é pedido que o utilizador defina primeiramente a inteligência de cada um dos "jogadores"artificiais (será mostrado duas vezes o mesmo menu).

```
1.  Computer doesn't think before playing.
2.  Computer thinks minimally before acting.
```

Depois de feitas as escolhas, cada jogada será separada por um *Enter* do teclado, tal como é demonstrado a seguir. Como as diferenças entre as inteligências utilizadas já foram referidas e explicitadas na secção 3.7, torna-se então prioritário mostrar realmente jogadas hipotéticas que o computador poderá tomar.

```
  1 [ X X X ]
      1 2 3
 2 [ X O O O ]
     1 2 3 4
3 [ X X O X X ]
    1 2 3 4 5
 4 [ O O O O ]
     1 2 3 4
  5 [ X O _ ]
      1 2 3
```

Tendo em conta o tabuleiro acima como sendo o inicial, então prossegue-se imediatamente para a execução de jogadas por parte de ambos os jogadores "artificiais".

```
Turn to play:  X
  1 [ X X _ ]
      1 2 3
 2 [ X O X O ]
     1 2 3 4
3 [ X X O X X ]
    1 2 3 4 5
 4 [ O O O O ]
     1 2 3 4
  5 [ X O _ ]
      1 2 3
Press Enter to continue.

Turn to play:  O
  1 [ X X _ ]
      1 2 3
 2 [ O _ X O ]
     1 2 3 4
3 [ X X O X X ]
    1 2 3 4 5
 4 [ O O O O ]
     1 2 3 4
  5 [ X O _ ]
      1 2 3
Press Enter to continue.
```

O utilizador terá de continuamente premir a tecla *Enter* até que alguns dos "jogadores"fique sem jogadas possíveis. Nesse momento é mostrada a mensagem habitual do vencedor.

Por mera curiosidade, o tabuleiro final neste jogo e respetivo vencedor foram

```
  1 [ _ X _ ]
      1 2 3
 2 [ _ _ _ O ]
     1 2 3 4
3 [ O X _ _ X ]
    1 2 3 4 5
 4 [ _ _ _ O ]
     1 2 3 4
  5 [ X O _ ]
      1 2 3
Player X can't take any more moves!  Player O wins!
```

5 Considerações Finais

5.1 Dificuldades

A nossa maior dificuldade residiu no desenvolvimento de testes para efetuar uma jogada, especialmente no funcionamento do predicado `movePiece` e auxiliares.

Especialmente no desenvolvimento da inteligência artificial do jogo, sentimos alguma dificuldade em encarar uma nova realidade que, para todo o ser humano é algo quase instantâneo, já no ponto de vista da máquina temos que seguir a lógica de *tell everything you want the machine to do*. De certa forma, foi uma motivação adicional para superarmos a dificuldade, no entanto, é sensato esclarecer que não é uma tarefa fácil tornar explícito à máquina aquilo que ela deve fazer.

A somar a isto, não esquecendo que a *interface* gráfica deste motor irá ser implementada na unidade curricular de Laboratório de Aplicações com Interface Gráfica, tentamos obter as melhores soluções, de modo a que não haja problemas futuros.

5.2 Desenvolvimentos Futuros

Um dos aspetos a salientar em desenvolvimentos futuros, seria melhorar claramente a inteligência artificial, no entanto, para esta tornar-se mais simples de implementar, seria necessário tornar o código fonte o mais claro e eficiente possível. A inteligência artificial, como requere um elevado perfecionismo implica um maior tempo de computação de testes ou até mesmo de otimização do resultado final, portanto seria extremamente necessário rever o código fonte e otimizá-lo antes de prosseguir-se para mais funcionalidades de inteligência artificial.

Tal como é descrito na secção da Jogada do Computador, seria deveras desafiante guardar todos os movimentos realizados pelo jogador e computador e, através disso, gerar conhecimento e a própria máquina aprender com ele.

Por último, e mais ligado à *interface* gráfica, através do `movePiece` que pretendemos desenvolver, seria agradável poder mostrar ao utilizador, quando este clica numa peça sua, as diferentes peças para onde poderá movê-la (assinalando com uma cor diferente as peças inimigas com, por exemplo, *outer glow*).

5.3 Conclusões

A parte realmente crucial a reter do desenvolvimento deste projeto é mesmo a complexidade da própria inteligência artificial que se pretende ou se espera obter. Um simples desvio de objetivos pode levar a crescimento exponencial da complexidade do problema, da sua resolução e implementação. Esse é um dos fatores que tornou este projeto desafiante, motivador e bastante didático, permitindo-nos adquirir técnicas e, mais importante ainda, *saber por onde não devemos ir*, i.e, caminhos ou resoluções que aparentemente poderão ser benéficas para a solução final mas, a longo prazo, ao incluir funcionalidades mais custosas em termos computacionais, podem tornar-se muito pouco eficientes.

Referências

[1] "Wolfram Alpha", http://www.wolframalpha.com/input/?i=1%2C7%2C19%2C37 (acedido em 5 de Novembro de 2014)

[2] "SWI Prolog", http://www.swi-prolog.org/pldoc/man?predicate=consult/1 (acedido em 1 de Novembro de 2014)

[3] "SWI Prolog IO", http://www.swi-prolog.org/pldoc/man?section=IO (acedido em 2 de Novembro de 2014)

[4] "Introdução à Inteligência Artificial - SCE-5774", http://www.icmc.usp.br/pessoas/sandra/18/trabalho01.html (acedido em 8 de Novembro de 2014)

[5] "Tactical and Strategic AI", http://imada.sdu.dk/ marco/Teaching/AY2012-2013/DM810/Slides/dm810-lec13.pdf (acedido em 8 de Novembro de 2014)

[6] "Foundations of Artificial Intelligence", http://ais.informatik.uni-freiburg.de/teaching/ss11/ki/slides/ai06_board_games_handout_4up.pdf (acedido em 8 de Novembro de 2014)

6 Anexos

Nesta secção segue-se o código-fonte desenvolvido para cada um dos seguintes predicados.

6.1 printBoard

Descrição: para impressão do tabuleiro.

```
printBoard( [], [], _ ).

printBoard( [Head|Tail], [SpaceNum|SpaceList], N ) :-
  write( N ),
  tab( SpaceNum ),
  write( '[␣' ),
  printLine( Head, M ),
  write( ']' ),
  nl,
  tab( SpaceNum ) ,
  write( '␣␣␣' ),
  printNums( M ),
  nl ,
  NN is N+1,
  printBoard( Tail , SpaceList, NN ).

printBoard( Board, Spaces ) :-
  printBoard( Board, Spaces, 1 ).
```

6.2 printLine

Descrição: para impressão de uma linha do tabuleiro no ecrã.

```
printLine( [], 0 ).

printLine( [E|List], M ) :-
  translate( E, C ),
  write( C ),
  write( '␣' ),
  printLine( List, MM ),
  M is MM+1.
```

6.3 translate

Descrição: "Converte"um dado número no símbolo correspondente (célula vazia, pedra PRETA ou pedra BRANCA).

```
translate( 0, '_' ).
translate( 1, 'X' ).
translate( 2, 'O' ).
```

6.4 printNums

Descrição: imprime no ecrã o número da linha e coluna no ato de impressão do tabuleiro (`printBoard`).

```
printNums( 0 ).

printNums( M ) :-
  MM is M-1,
  printNums( MM ),
  write( M ),
  write( '␣' ).
```

6.5 makeDinBoard

Descrição: retorna um novo tabuleiro dinâmico, ou seja, o tamanho `Size` é especificado no predicado e o tabuleiro correspondente é devolvido (inicialmente com células todas vazias).

```
makeDinBoard( [Board], [0], Counter, Size ) :-
  ListSize is Counter+Size,
  Counter is Size-1,
  makeDinList( Board, ListSize ).

makeDinBoard( Board, Spaces, Counter, Size ) :-
  ListSize is Counter+Size,
  NewCounter is Counter+1,
  NumSpaces is Size-NewCounter,
  makeDinList( CurrentLine, ListSize ),
  makeDinBoard( InBoard, InSpaces, NewCounter, Size ),
  append( [CurrentLine], InBoard, TmpBoard ),
  append( TmpBoard, [CurrentLine], Board ),
  append( [NumSpaces], InSpaces, TmpSpaces ),
  append( TmpSpaces, [NumSpaces], Spaces ).
```

6.6 makeDinList

Descrição: retorna uma lista apenas composta por zeros (útil para criação de cada linha do tabuleiro).

```
makeDinList( [], 0 ).

makeDinList( [0|LT], Size ) :-
  NSize is Size-1,
  makeDinList( LT, NSize ).
```

6.7 fillLine

Descrição: preenche uma linha com um número aleatório de X's e O's.

```
%Fills a line with a random number of X's and O's.
fillLine( [], [], X, O, X, O ). %End of line reached.
fillLine( [O], [O], 0, 0, 0, 0 ). %End reached.

fillLine([_|LT], [RLH|RLT], NumCross, 0, RCross, 0):-
  RLH is 1, %Current element is an X (1).
  NewCross is NumCross-1, %Decrement number of X's.
  fillLine( LT, RLT, NewCross, 0, RCross, 0 ).

fillLine([_|LT], [RLH|RLT], 0, NumCircle, 0, RCircle):-
  RLH is 2, %Current element is an O (2).
  NewCircle is NumCircle-1, %Decrement number of O's.
  fillLine( LT, RLT, 0, NewCircle, 0, RCircle ).

fillLine( [_|LT], [RLH|RLT], NX, NO, RX, RO ):-
  random( 0, 2, R ), %R = [0,1]; ( [0,2[ )
  NewCross is NX-(1-R),
  NewCirc is NO-R,
  RLH is 1+R,
  fillLine( LT, RLT, NewCross, NewCirc, RX, RO ).
```

6.8 fillBoard

Descrição: retorna o tabuleiro preenchido com o número aleatório de X's e O's.

```
fillBoard( [], [], 0, 0 ).

fillBoard( [BH|BT], [RBH|RBT], NX, NO ) :-
  fillLine( BH, RBH, NX, NO, NewCross, NewCircle ),
  fillBoard( BT, RBT, NewCross, NewCircle ).
```

6.9 numCells

Descrição: retorna o número de células de um hexágono, tendo em conta o seu tamanho atual.[6]

```
numCells( Num, Size ) :-
  Num is 3*Size*Size-3*Size+1.
```

[6] O número de células é dado como $C = 3k^2 - 3k + 1$, em que k é o comprimento de cada lado do hexágono.

6.10 numPlayerPieces

Descrição: retorna o número de células de cada jogador, tendo em conta o tamanho do tabuleiro.[7]

```
numPlayerPieces( Num, Size ) :-
  numCells( TotalNum, Size ),
  Num is round((TotalNum-1)/2).
```

6.11 playerMoves

Descrição: testa se um jogador pode mover uma peça qualquer.

```
playerMovesLine( _, [], _, _, _, _ ).

playerMovesLine(Board,[_|LT],Line,Col,Piece,Valid):-
  (getPiece( Board, Line, Col, Piece ),
    listMoves( Board, Line, Col, ML );
    append( [], [], ML ) ),
  (append( [], [], ML ),
    NCol is Col+1,
    playerMovesLine(Board,LT,Line,NCol,Piece,Valid);
      Valid is 1).

playerMoves( _, [], _, _, _ ).

playerMoves( Board, [BH|BT], Line, Piece, Valid ) :-
  playerMovesLine( Board, BH, Line, 1, Piece, Valid ),
  NLine is Line+1,
  playerMoves( Board, BT, NLine, Piece, Valid ).

playerMoves( Board, Piece, Valid ) :-
  playerMoves( Board, Board, 1, Piece, Valid ), !,
  (integer( Valid ), !;
  Valid is 0).
```

6.12 getSize

Descrição: retorna o comprimento de lado de um tabuleiro.

```
getSize( [BH|_], Size ) :-
  length( BH, Size ).
```

[7] O número de células de cada jogador é dado como $C_p = \frac{C-1}{2}$, em que C é o número total de células.

6.13 movePiece

Descrição: move uma peça para uma dada posição do tabuleiro. Utiliza como predicados auxiliares `replacePiece` que, por sua vez, utiliza o predicado `replaceInLine`.

```
replaceInLine( [_|LT], [Piece|LT], 1, Piece ).

replaceInLine( [LH|LT], [LH|RLT], Col, Piece ) :-
  NewCol is Col-1,
  replaceInLine( LT, RLT, NewCol, Piece ).

replacePiece([BH|BT], [RBH|BT], 1, Col, Piece) :-
  replaceInLine( BH, RBH, Col, Piece ).

replacePiece([BH|BT], [BH|RBT], Line, Col, Piece) :-
  NewLine is Line-1,
  replacePiece( BT, RBT, NewLine, Col, Piece ).

movePiece(Board, ResultBoard, Line, Col, DLine, DCol):-
  getPiece( Board, Line, Col, Piece ),
  replacePiece( Board, TmpBoard, DLine, DLine, Piece ),
  replacePiece( TmpBoard, ResultBoard, Line, Col, 0 ).
```

6.14 Computer Easy

Descrição: inteligência artificial utilizada no modo fácil. Utiliza o predicado `easyMoves` que, por sua vez, utiliza o predicado `easyLines`.

```
computer_easy( Board, Spaces, ResultBoard, Piece ) :-
  %Print whose turn it is to play.
  translate( Piece, C ),
  write( 'Turn to play: ' ), write( C ), nl,
  %Get the move to be made.
  easyMoves( Board, Piece, Line, Col ),
  listMoves( Board, Line, Col, [[DLine|DCol]|_] ),
  %Make the move.
  movePiece(Board,ResultBoard,Line,Col,DLine,DCol),
  %Print the Board and end.
  printBoard( ResultBoard, Spaces ),
  write( 'Press Enter to continue.' ), nl,
  get_char( _ ).
```

6.15 Computer Hard

Descrição: inteligência artificial utilizada no modo difícil. Utiliza o predicado `easyMoves` que, por sua vez, utiliza o predicado `easyLines`.

```
computer_easy( Board, Spaces, ResultBoard, Piece ) :-
  %Print whose turn it is to play.
  translate( Piece, C ),
  write( 'Turn to play: ' ), write( C ), nl,
  %Get the move to be made.
  hardMoves( Board, Piece, Line, Col ),
  listMoves( Board, Line, Col, [[DLine|DCol]|_] ),
  %Make the move.
  movePiece(Board,ResultBoard,Line,Col,DLine,DCol),
  %Print the Board and end.
  printBoard( ResultBoard, Spaces ),
  write( 'Press Enter to continue.' ), nl,
  get_char( _ ).
```

Nota: Não estão presentes no relatório certos predicados por causa da sua extensão de casos particulares, sendo recomendado ver o código fonte pelos respetivos ficheiros, devidamente comentados.