Eigenstate

Ricardo Jorge de Araújo Ferreira - up200305418

Grupo Eigenstate_2

Projecto no github: `https://github.com/ricardojaferreira/Eigenstate`

18 de Novembro de 2018

1 INTRODUÇÃO

Neste trabalho foi desenvolvido o jogo Eigenstate [1]. Eigenstate, é um jogo de tabuleiro para dois jogadores. As regras do jogo e a sua história são apresentadas na secção 2.
Pretende-se com este trabalho, desensenvolver um jogo que, obdecendo às regras, valide a movimentação das peças e verifique condições de terminação do jogo com derrota, vitória ou empate. Devem ainda ser desenvolvidos os seguintes modos de jogo: Humano/Humano, Humano/Computador, Computador/Humano e Computador/Computador. As jogadas do computador devem ser desenvolvidas com dois modos possíveis. Um modo aleatório, em que cada jogada é escolhida de modo aleatório, sem se preocupar se essa será a melhor jogada naquele momento. No entanto, todas as jogadas devem ser válidas, atendendo às regras do jogo. Outro modo mais avançado, em que, antes de cada jogada, é feita uma validação do estado actual do jogo e escolhida a melhor jogada para aquele estado. Neste jogo, uma jogada envolve, mover uma peça e colocar Pegs em duas peças para permitir mais movimentos em jogadas seguintes. Atendendo a isto, o modo avançado, deve determinar não só a melhor movimentação possível, mas também a colocação de Pegs que potenciem uma melhor futura jogada. No caso aleatório, a colocação de Pegs é também aleatória.
O jogo foi implementado utilizando a linguagem Prolog em SWI-Prolog.
O jogo deve ter também uma interface em modo de texto, que permita ao jogador de forma intuitiva interagir com o jogo e verificar o estado do mesmo.
Todos os objectivos foram cumpridos e todas as regras do jogo implementadas. Nas próximas secções será apresentado em detalhe a lógica do jogo.

2 O JOGO EIGENSTATE

2.1 HISTÓRIA

Eigenstate é um jogo de tabuleiro criado por Martin Grider [1]. É um jogo para dois jogadores com regras simples mas que cresce em complexidade à medida que o jogo avança. Este jogo ainda não está disponível no mercado, mas prêve-se a criação de um projecto no kickstarter para isso acontecer.
Eigenstate foi inspirado nos jogos The Duke [2] e Onitama [3]. Durante o seu desenvolvimento o autor escreveu vários cenários de jogabilidade e condições de vitória, tendo pensado até num cenário semelhante ao checkmate do Xadréz. Mas no final, durante uma apresentação do jogo, decidiu manter apenas duas regras básicas. O jogador na sua vez só tem que mover uma peça e colocar dois novos pegs para possibilitar movimentos numa nova jogada. Termina com a eliminação das peças adversárias, no entanto, a condição de vitória ainda pode ser alterada na versão final.
O nome Eigenstate vem de um termo da física quântica que se refere ao possível movimento de uma partícula [4].

2.2 REGRAS

2.2.1 CONFIGURAÇÃO

O jogo joga-se num tabuleiro quadrado com 36 células (6x6). Cada jogador escolhe uma cor e coloca todas as peças dessa cor, 6 peças por jogador, no seu lado do tabuleiro.
Todas as peças começam com dois pegs, um no centro que representa a posição actual da peça e outro peg que permite a peça avançar uma casa.

2.2.2 JOGABILIDADE

Cada jogador no seu turno, se possível, deve:

1. Mover uma das suas peças em conformidade com os pegs (como explicado mais à frente);

2. Colocar dois pegs em qualquer uma das suas peças, na mesma, ou diferentes.

O movimento das peças é regido pelo seguinte. Todos os pegs colocados na peça representam possíveis jogadas relativamente à sua posição actual. A posição actual é marcada pelo peg colocado no centro, tipicamente com uma cor diferente dos outros.
Este movimento está ilustrado na figura 2.2. Na figura é possível ver que o jogador moveu a peça 1 e colocou dois novos pegs, um na peça que moveu e outro na peça 2. Na próxima jogada, este jogador poderá mover a peça 1 para as posições "a" ou "b" e a peça 2 pode ser colocada nas posições "d" ou "c". Estes movimentos correspondem à posição dos pegs em cada peça.
O movimento das peças obdece ainda às seguintes regras:

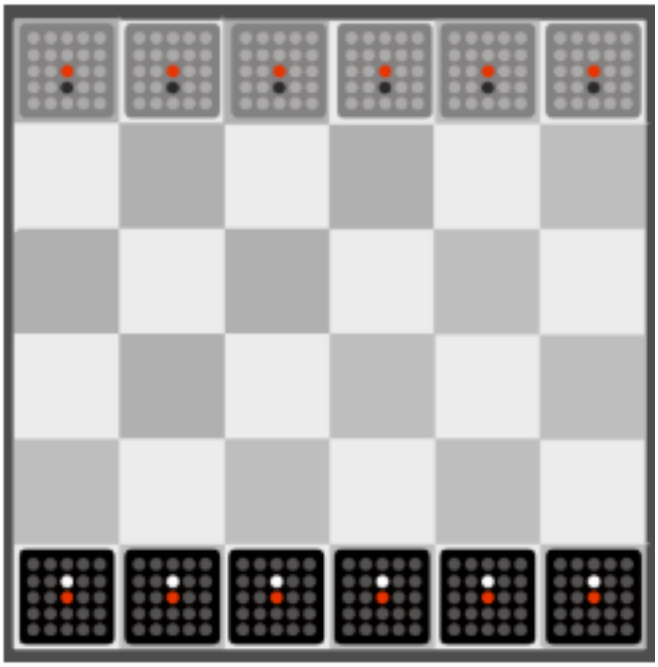

Figure 2.1: Eigenstate: Configuração inicial. [5]

- Os pegs nunca podem ser removidos das peças, garantindo que as peças podem sempre avançar uma casa;
- As peças podem saltar por cima de outras peças;
- As peças não podem sair do tabuleiro;
- As peças não rodam;
- As peças só se podem mover para trás se tiverem pegs atrás do marcador central;
- Uma peça que se mova para a mesma posição de outra peça, faz com que a outra seja removida do tabuleiro;
- Peças do mesmo jogador, podem mover-se para cima uma das outras e as peças têm que ser removidas conforme a regra anterior.

A colocação de pegs, obdece às seguintes regras:

- Só podem ser colocados em espaços livres, das peças do jogador não capturadas;
- Em cada turno podem ser colocados 2 pegs em peças diferentes ou na mesma;
- Não é necessário colocar nenhum peg na peça movida nessa ronda.

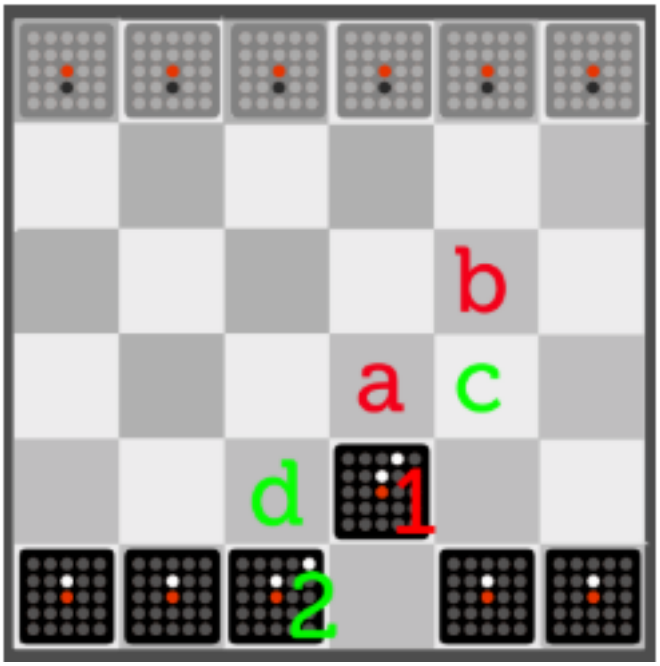

Figure 2.2: Eigenstate: Movimento de peças. [5]

2.2.3 Condição de Vitória

A forma principal de vitória consiste em reduzir o adversário a apenas uma peça. Num caso em que ambos os jogadores tenham duas peças cada um, a vitória pode ser conseguida se um jogador colocar pegs em todas as posições de uma das suas peças.

3 Lógica do Jogo

A implementação do jogo está organizada do seguinte modo:

- **main.pl** - Neste ficheiro está implementado o predicado *play/0* que dá início ao jogo e os predicados relativos ao ciclo de jogo bem como as chamadas para movimentar peças e adicionar pegs. Este ficheiro é o ponto de entrada da aplicação.
- **utils.pl** - Neste ficheiro encontram-se predicados que desempenham funções básicas, como contar o número de elementos numa Lista de Listas (*countElementsOnMatrix/3*) ou obter a posição X e Y de um elemento numa Lista de Listas (*getMatrixElementPosition/4*), entre outras. Estas funções são utilizadas em várias partes da lógica do jogo.
- **Pasta logic**
  - **board.pl** - Predicados relativos a funções e características do tabuleiro de jogo, por exemplo, o tamanho do tabuleiro (*boardSquareSize/1*) ou obter o valor da célula através da posição (*getBoardCellValue/4*).

  - **pieces.pl** - Predicados relativos a funções e características das peças, por exemplo, verificar se peça pertence ao jogador (*checkIfPieceBelongsToPlayer/2*) ou inicializar peças (*initPiecesPlayer/2*).
  - **players.pl** - Predicados relativos a acções e características dos jogadores, por exemplo, definir o tipo do jogador (*setPlayer/2*) obter a lista de peças activas do jogador (*getListOfPiecesForPlayer/3*).
  - **victory.pl** - Predicado para verificar se o jogo terminou ou se deve continuar (*game_over/5*).
  - **Pasta ai**
    * **move.pl** - Lógica para escolha da melhor jogada para o modo avançado do computador, (*choosePieceToMoveComputerAI/6*).
    * **add_pegs.pl** - Lógica para escolha da peça e colocação de peg para o modo avançado do computador, com duas estratégias distintas (*strategyFillPegs/7*), (*strategyGoFurther/7*), orquestradas por (*runPegStrategy/7*). O ciclo de jogo chama esta lógica através do predicado (*addPegToPieceComputerAI/4*).
    * **common.pl** - Implementação do predicado (*getBestChoiceByScore/4*), utilizado pela lógica de movimentação e de Pegs para escolher a melhor peça e posição baseada numa lista de pesos.

- **Pasta cli**
  - **board_cli.pl** - Predicados para impressão de estados relacionados com o tabuleiro, por exemplo, visualizar o tabuleiro (*display_game/2*).
  - **piece_cli.pl** - Predicados para impressão de peças, por exemplo, (*printOnePiece/2*).
  - **player_cli.pl** - Predicados para interação com o jogador, por exemplo, pedir ao jogador para escolher uma célula para mover a peça (*askPlayerToChooseCell/5*).
  - **printutils.pl** - Utilitários para as funções de visualização, por exemplo, cores das peças dos jogadores (*playerColor/2*).

- **Pasta loaders**
  - **load_cli.pl** - Consulta todos os ficheiros relacionados com a visualização e interação com a consola de texto.
  - **load_logic.pl** - Consulta todos os ficheiros relacionados com lógica do jogo.

Alguns ficheiros, no final, têm uma secção de testes, feita para testar predicados isoladamente, utilizando estados pré-definidos. Estes testes funcionam um pouco como testes unitários utilizando mocks.

3.1 Representação do Estado do Jogo

O jogo terá como representações principais o tabuleiro e a peça. Sendo que as peças são também elas tabuleiros para a colocação de pegs, a representação destes dois elementos será

semelhante, embora a forma como se interpretam os resultados seja bastante diferente. Ambos terão uma representação baseada em listas de listas.
A representação inicial do tabuleiro será dada pelo predicado *startBoard/1*.

```
1 startBoard([
2                   [1,2,3,4,5,6],
3                   [0,0,0,0,0,0],
4                   [0,0,0,0,0,0],
5                   [0,0,0,0,0,0],
6                   [0,0,0,0,0,0],
7                   [7,8,9,10,11,12]
8 ])
```

O tabuleiro é uma lista com 6 sub-listas, uma para cada linha do tabuleiro. Com esta estrutura, podemos pesquisar o tabuleiro de uma forma análoga a uma matriz, pesquisando por linha e coluna, tirando partido da divisão das listas em *Head* e *Tail*.
Tendo em conta esta estrutura, para obter o valor de uma célula do tabuleiro, sabendo a sua posição X (Linha) e a sua posição Y (coluna), podemos utilizar o predicado *getBoardCellValue/4*:

```
1 getBoardCellValue(Board,PosX,PosY,Value):-
2         X1 is PosX-1,
3         discardElementsFromList(Board,X1,[Line|_]),
4         Y1 is PosY-1,
5         discardElementsFromList(Line,Y1,[Value|_]).
```

Este predicado elimina linhas do tabuleiro até chegar à linha definida pela posição PosX. Obtendo a linha pretendida, são eliminados os elementos da linha até à PosY, obtendo-se uma lista em que o primeiro elemento (*Head*) é o valor pretendido. Esta particularidade de divisão das listas em *Head* e *Tail* permite utilizar o mesmo predicado de eliminação de elementos, tanto para listas de listas como para listas simples, pois, cada lista dentro de outra lista, pode ser tratada como um elemento simples de uma lista.
O predicado *discardElementsFromList/3* é o seguinte:

```
1 discardElementsFromList(X,0,X).
2
3 discardElementsFromList([_|T],N,Result):-
4         N1 is N-1,
5         discardElementsFromList(T,N1,Result).
```

Este predicado vai recursivamente retirando elementos da lista passada como primeiro argumento e termina quando o valor a obter, passado na variável N, chega a zero, passando para Result os elementos restantes da lista.
Os atómos do tabuleiro têm o seguinte significado:

- 0 - célula vazia;
- 1 até 6 - peças do jogador 1;

- 7 até 12 - peças do jogador 2.

Como cada peça pode ter um estado diferente ao longo do jogo (dependendo do número de pegs que tem) é necessário uma representação individual de cada uma.
A aborgadem para a representação das peças será semelhante à utilizada para o tabuleiro, embora com alguma complexidade acrescida. Cada peça será representada por uma lista com o seguinte formato (peça no estado inicial):

```
1 startPiecePlayer(
2         1, [
3                         [0,0,0,0,0],
4                         [0,0,0,0,0],
5                         [0,0,8,0,0],
6                         [0,0,1,0,0],
7                         [0,0,0,0,0]
8                 ]
9         ).
```

O método para percorrer as peças será o mesmo utilizado para o tabuleiro, pois cada peça, é ela própria um tabuleiro. Cada átomo nas peças terá o seguinte significado:

- 0 - célula vazia, pode ser colocado um peg;
- 1 - célula com peg;
- 8 - célula central que marca a posição da peça no tabuleiro.

Um predicado de particular interesse para obter valores das peças é o *getPieceElementsPositions/5*:

```
1  getPieceElementsPositions(_F,_X,[],P,[P]):-
2          !.
3
4  getPieceElementsPositions(Functor,X,[H|T],[],Pegs):-
5          getByLine(Functor,[X,-2,H,NewPegLine]),
6          X1 is X+1,
7          getPieceElementsPositions(Functor,X1,T,NewPegLine,Pegs).
8
9  getPieceElementsPositions(Functor,X,[H|T],PrevPegs,[PrevPegs|Pegs]):-
10         getByLine(Functor,[X,-2,H,NewPegLine]),
11         X1 is X+1,
12         getPieceElementsPositions(Functor,X1,T,NewPegLine,Pegs).
```

Por cada linha da peça é chamado o predicado *getByLine/2* com um desvio de -2. Este predicado está definido do seguinte modo:

```
1          getByLine(P,LArgs):- G=.. [P|LArgs], G.
```

Define um operador univ que pode ser chamado com um dos seguintes predicados *getPegsByLine/4*, *getFutureMovementByLine/4* ou *getEmptySpaceByLine/4*, que permitem obter o número de células ocupadas com pegs, o número de células onde é possível colocar um novo peg e o número de células vazias, respectivamente. O valor de desvio de -2 serve para traduzir as posições da peça em deslocação da peça, ou seja, um peg na posição (1,1) corresponde a um movimento de (-2,-2). Se a peça estiver no tabuleiro na posição (4,3), significa que uma deslocação de (-2,-2) colocará a peça na posição (2,1). Sendo o ponto central das peças a posição (3,3), isto significa que as peças podem efectuar deslocações no tabuleiro entre (-2,-2) e (2,2).

3.2 Visualização do Tabuleiro

O tabuleiro numa posição inicial está representado na Figura 3.1. Pode também ser visto a representção das peças de cada jogador na posição inicial. As peças verdes pertencem ao jodador 1 e as peças vermelhas ao jogador 2.
O tabuleiro numa fase de jogo intermédia pode ser visto na Figura 3.2 e na fase final na Figura 3.3.
A Figura 3.4 mostra uma peça num estado intermédio do jogo. O símbolo "*" representa Pegs, o símbolo "@" marca a posição actual da peça e o símbolo "O" representa uma célula vazia.

3.3 Lista de Jogadas Válidas

Uma jogada válida neste jogo deve respeitar um conjunto de factores, que são: ser uma peça do jogador, ter um peg que lhe permita ir para uma determinada posição e essa posição ser dentro do tabuleiro.
Tendo em vista as condicionantes anteriores, a validação de uma jogada inicia-se com a escolha da peça a mover. No caso de um jogador Humano, essa escolha é feita com o predicado *choosePieceToMove/3*. Este predicado é definido do seguinte modo:

```
1 choosePieceToMove(Player,Board,PieceToMove):-
2         boardSquareSize(S),
3         askPlayerToChoosePieceToMove(Player,S,PosX,PosY),
4         checkBoardCellChoice(Player,Board,PosX,PosY,PieceToMove),!.
```

Verifica-se o tamanho do tabuleiro e pede-se ao jogador para escolher uma posição X e Y dentro do tabuleiro, depois essa posição é validada, para se verificar se representa uma peça do jogador e caso seja, é devolvida a variável *PieceToMove* com o número da peça a mover.
Com esta informação é então possível obter o número de pegs da peça, traduzidos em movimento, como explicado na secção 3.1 recorrendo ao predicado *getPegsByLine/4*. Pede-se então ao jogador para escolher nova posição X e Y para a colocação da peça e valida-se o movimento com o seguinte predicado:

```
1 checkIfIsAValidMovement(_Player,PegsPositions,Line,Column,PosX,PosY,PosX,PosY):-
2         MovementX is PosX-Line,
3         MovementY is PosY-Column,
4         checkIfExists([MovementY,MovementX],PegsPositions).
```

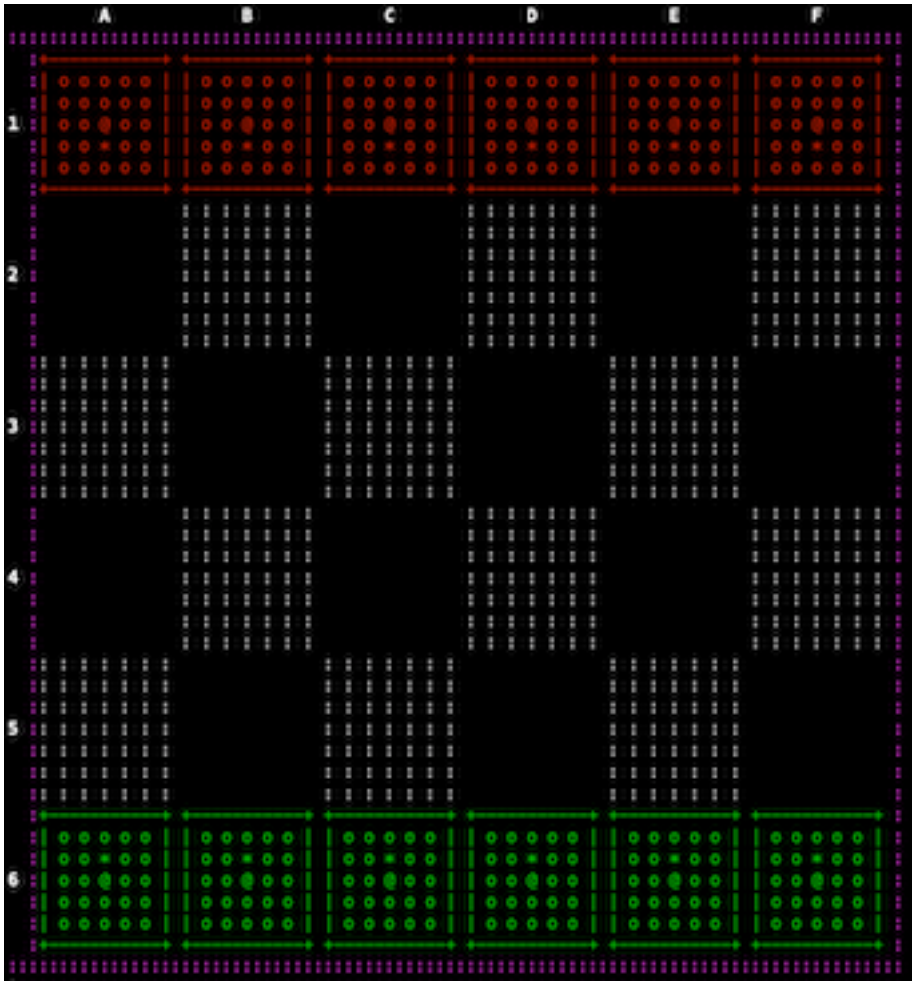

Figure 3.1: Representação do tabuleiro num estado inicial

```
5
6 checkIfIsAValidMovement(Player, PegsPositions, Line, Column, _X, _Y, PosX, PosY):-
7         showErrorPieceCannotBeMovedToThatLocation,
8         getPositionToMove(Player, PegsPositions, Line, Column, PosX, PosY).
```

Este predicado recebe o jogador actual, para, caso a peça e movimento escolhido não correspondam a um movimento válido se possa voltar a pedir ao jogador para escolher uma nova opção. Recebe uma lista com a posição, traduzida em movimento, de todos os pegs da peça. Recebe ainda a posição da peça no tabuleiro (Line e Column) e a Posição X e Y para mover a peça (PosX,PosY). Com esta informação é calculado o movimento necessário em X e em Y para a peça ir para a posição escolhida e verifica-se se esse conjunto de movimento existe na lista de Pegs. Caso exista, o movimento é válido, se não, pede-se uma nova opção ao jogador. De notar, que o jogador apenas consegue escolher posições X e Y dentro do tabuleiro, caso não o faça, o predicado *getPositionToMove/6* estará constantemente a ser repetido, pedindo

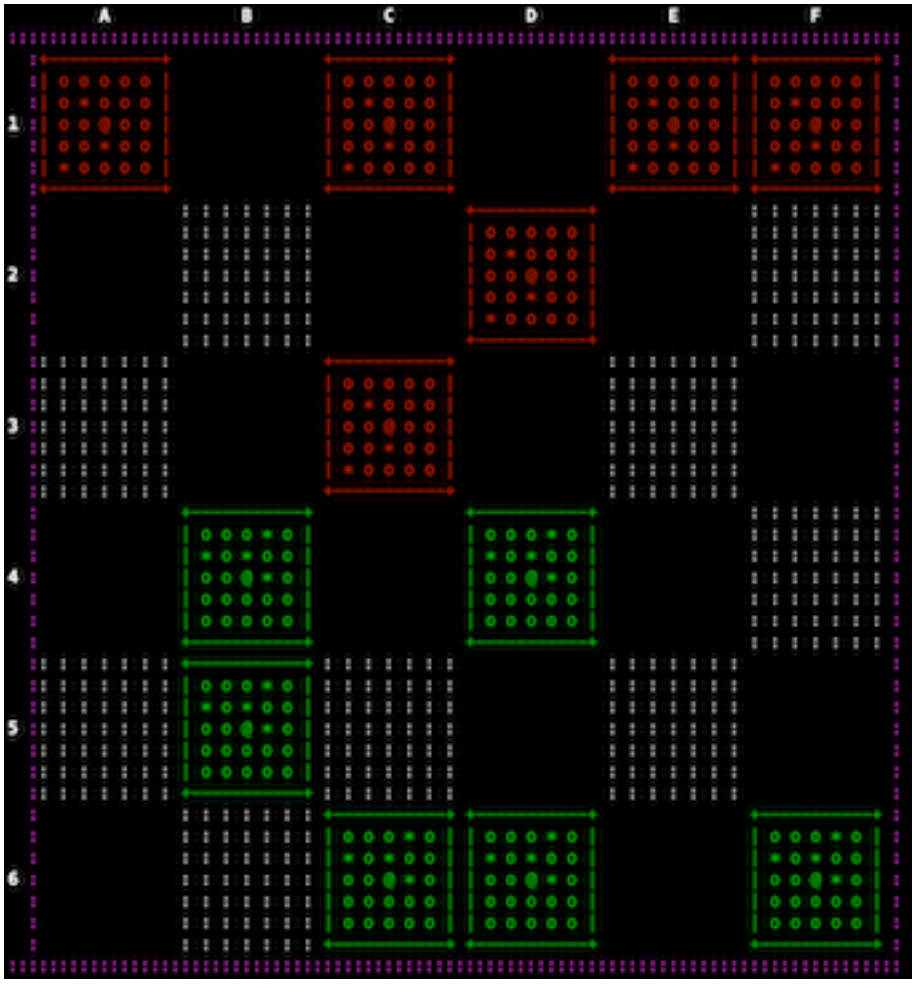

Figure 3.2: Representação tabuleiro num estado intermédio

novas posições ao jogador.

Devido a todo este processo iterativo de escolha e validação, este predicado de validação de jogada é único para o jogador humano. No caso do jogador computador (Aleatório ou Avançado) utiliza-se um predicado *valid_moves/4* mais simples.

```
1 valid_moves(Board,Piece,PegsPositions,ValidMoves):-
2         getBoardCellPosition(Board,Piece,1,1,Line,Column),
3         filterMoves(Line,Column,PegsPositions,ValidMoves),!.
4
5 filterMoves(_Line,_Column,[],[]).
6 filterMoves(Line,Column,[[Y,X]|T],[[PosY,PosX]|FT]):-
7         PosX is Line+X,
8         PosY is Column+Y,
9         boardSquareSize(S),
```

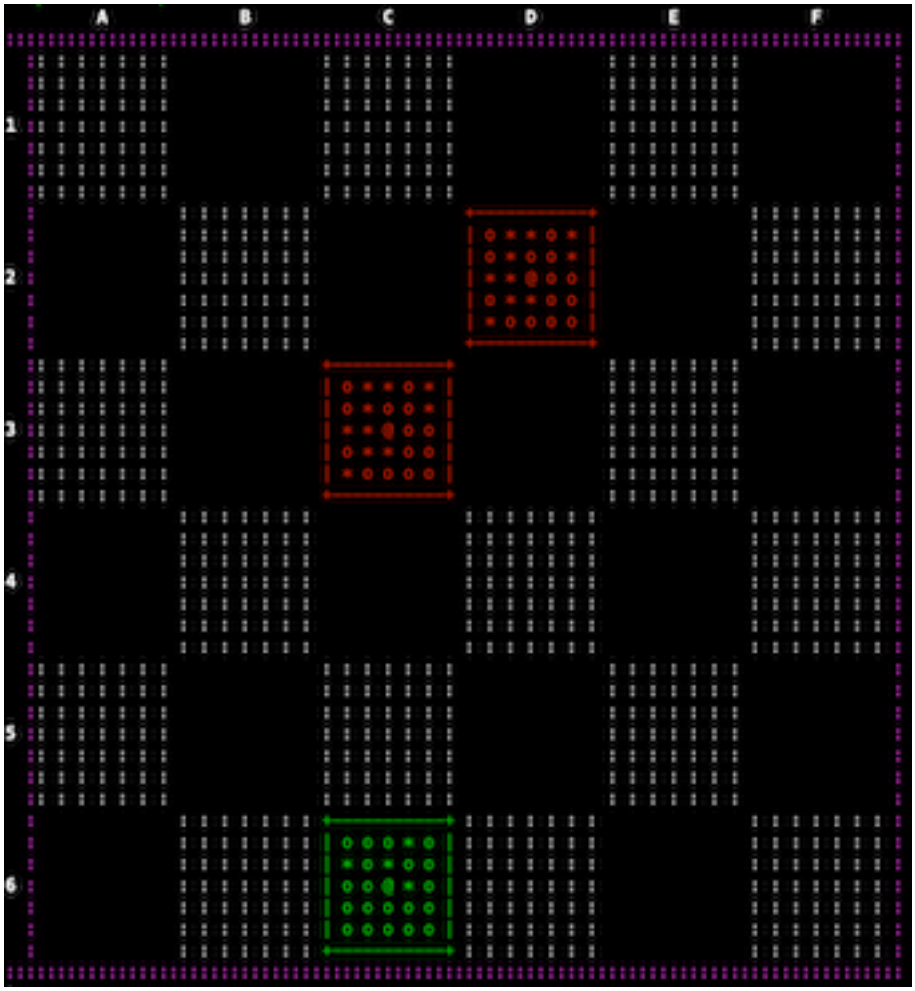

Figure 3.3: Representação do tabuleiro no estado final.

```
10          between(1,S,PosX),
11          between(1,S,PosY),
12          filterMoves(Line,Column,T,FT).
13  filterMoves(Line,Column,[_H|T],ValidMoves):-
14          filterMoves(Line,Column,T,ValidMoves).
```

O computador tem a possibilidade de escolher uma peça a partir de uma lista apenas com peças que pode mover, não havendo necessidade de validar se a peça é do jogador correcto, nem de pedir para escolher nova peça. Do mesmo modo para a escolha do movimento, depois de saber quais os pegs disponíveis, basta filtrar quais os pegs que podem resultar em jogadas válidas e pedir ao computador para escolher uma jogada. O predicado *valid_moves/4* faz exactamente isso. A partir de uma lista com posições de pegs, traduzidos em movimento, é aplicado um filtro que utiliza a posição da peça escolhida (Line e Column) e obtém-se uma lista de movimentos possíveis para aquela peça (ValidMoves).

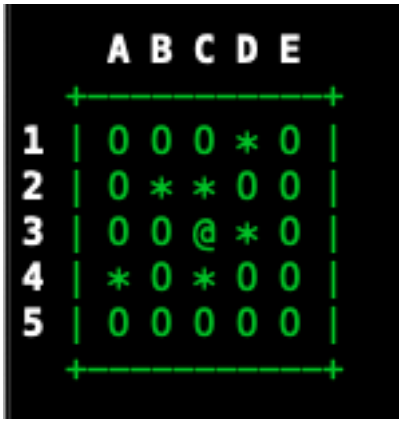

Figure 3.4: Representação de uma peça num estado intermédio.

O computador pode depois pegar nesta lista e escolher um movimento para executar. Esta escolha irá depender do tipo de computador.

3.4 Execução de Jogadas

Como explicado em secções anteriores, uma jogada completa envolve, escolher uma peça para mover, verificar os pegs dessa peça, mover a peça com base nos pegs e escolher peças para colocar dois pegs.
Estes passos são executados no predicado *gameLoop/7* cujo primeiro átomo é um número de 1 a 4 que representa cada um dos estados do movimento:

- 1 - Escolher e mover peça através do predicado *move/6*.
- 2 - Escolher peça e adicionar primeiro Peg através do predicado *playerAddPeg/5*.
- 3 - Escolher peça e adicionar segundo Peg através do predicado *playerAddPeg/5*.
- 4 - Responsável por terminar o loop e mostrar o estado final do jogo.

Dada a complexidade e condicionantes envolvidas com o movimento de uma peça, o predicado *move/6* terá execuções diferentes, em função do tipo de jogador.

```
1  move(Player,human,ListOfPieces,Board,NewBoard,NewListOfPieces):-
2          choosePieceToMove(Player,Board,PieceToMove),
3          movePiece(Player,PieceToMove,ListOfPieces,Board,NewListOfPieces,NewBoard),
4          display_game(NewBoard,NewListOfPieces).
5
6  move(Player,computerRandom,ListOfPieces,Board,NewBoard,NewListOfPieces):-
7          waitForPlayerMove(Player),
8          choosePieceToMoveComputer(Player,Board,PieceToMove),
9          movePieceComputer(PieceToMove,ListOfPieces,Board,NewListOfPieces,NewBoard),
10         display_game(NewBoard,NewListOfPieces).
```

```
11
12 move(Player,computerAI,ListOfPieces,Board,NewBoard,NewListOfPieces):-
13         waitForPlayerMove(Player),
14         choosePieceToMoveComputerAI(Player,Board,ListOfPieces,PieceToMove,PosX,PosY),
15         movePieceOnBoard(Board,PieceToMove,PosX,PosY,NewBoard),
16         updatePieceList(NewBoard,ListOfPieces,NewListOfPieces),
17         display_game(NewBoard,NewListOfPieces).
```

O jogador humano começa por escolher uma peça para mover (*choosePieceToMove/3*), terminando apenas quando a peça escolhido é uma das suas peças. Sabendo qual a peça a mover, inicia-se a escolha de uma nova posição para a peça, utilizando os predicados de validação apresentados na secção 3.3, isto dará origem a um novo tabuleiro (NewBoard), com a posição da peça movida actualizada e uma nova lista de peças (NewListOfPieces), que se poderá manter igual ou ser actualizada, caso tenha sido eliminada alguma peça do tabuleiro. No final mostra-se o tabuleiro resultante.
O jogador computador aleatório, escolhe a peça a mover directamente do tabuleiro, pois sabe quais as peças que lhe pertencem, não precisa de escolher posição X nem Y como o jogador humano. Fazendo então uma escolha aleatório de entre as suas peças, inicia-se o movimento da peça, que consiste na escolha de uma movimentação aleatória escolhida, utilizando o resultado do predicado *valid_moves/4*, já apresentado. No final, é calculado novo tabuleiro e nova lista de peças como no caso anterior e apresentado o tabuleiro resultante.
O jogador computador avançado, executa funções mais avançadas para a escolha da peça a mover. Estas acções estão descritas nas secções 3.6 e 3.7. Após escolha da peça a mover, o predicado *choosePieceToMoveComputerAI/6* retorna a Peça a mover (PieceToMove) e a posição para onde deve ir a peça (PosX e PosY), que são as mesmas variáveis utilizadas pela movimentação do jogador humano, por isso, no final do movimento do computador avançado, tal como no movimento do jogador humando, os predicados *movePieceOnBoard/5* e *updatePieceList/3* são utilizados para se obter um tabuleiro actualizado e uma nova lista de peças.
Para adicionar novos Pegs é utilizado o predicado *playerAddPeg*, também com execuções diferentes para os diferentes tipos de jogador.

```
1  playerAddPeg(Player,human,Board,ListOfPieces,NewListOfPieces):-
2          addPegToPiece(Player,Board,ListOfPieces,NewListOfPieces),
3          display_game(Board,NewListOfPieces).
4
5  playerAddPeg(Player,computerRandom,Board,ListOfPieces,NewListOfPieces):-
6          waitForPlayerToAddPeg(Player),
7          addPegToPieceComputer(Player,Board,ListOfPieces,NewListOfPieces),
8          display_game(Board,NewListOfPieces).
9
10 playerAddPeg(Player,computerAI,Board,ListOfPieces,NewListOfPieces):-
11         waitForPlayerToAddPeg(Player),
12         addPegToPieceComputerAI(Player,Board,ListOfPieces,NewListOfPieces),
13         display_game(Board,NewListOfPieces).
```

A escolha de colocação de pegs será análoga à apresentada para mover uma peça, com a diferença que a posição a escolher será uma posição dentro da peça, não havendo validação de movimentação válida. Deve-se contudo validar que a posição escolhida está vazia e dentro dos limites da peça. O computador aleatório, escolhe uma posição aleatória com base nas posições disponíveis e o computador avançado utiliza um conjunto de regras para decidir qual a melhor posição para colocar o novo peg.

3.5 Final do Jogo

O final do jogo pode acontecer de duas formas distintas, um dos jogadores fica reduzido a uma peça e vence o outro jogador, ou ambos os jogadores têm duas peças e vence o primeiro jogador a completar uma peça com pegs.
Para se verificar estas condições em cada fase da movimentação deve-se verificar a condição de vitória, feita através do predicado *game_over/5*. Este predicado, para além de verificar a condição de vitória, calcula também qual o próximo estado do movimento, como referido na secção 3.3.

```
1 game_over(Loop,Board,_ListOfPieces,NextLoop,Winner):-
2         victoryByEatPieces(Board,Loop,NextLoop,Winner).
3
4 game_over(Loop,_Board,ListOfPieces,NextLoop,Winner):-
5         victoryByPieceWithPegs(ListOfPieces,Loop,NextLoop,Winner).
```

O predicado *game_over/5* utiliza duas estratégias para verificar se o jogo terminou. A primeira é verificar se algum dos jogadores ficou reduzido a apenas uma peça e nesse caso, termina o jogo, devolvendo na variável Winner o vencedor e passando a variável NextLoop para 4.
Caso não se consiga provar a condição anterior, verifica-se se ambos os jogadores têm apenas duas peças e qual deles tem uma peça com todos os pegs preenchidos. Se isto for provado, o jogo termina, com indicação do vencedor ou empate.
No caso da última validação não se provar, a variável NextLoop é actualizada para o ciclo correspondente e o jogo continua.

3.6 Avaliação do Tabuleiro

A avaliação do tabuleiro é feita através da atribuição de pesos a cada célula do tabuleiro e feita em três fases. Por isso, não se utilizou o predicado *value(+Board, +Player, -Value)* tal como sugerido no enunciado.
Esta avaliação é feita através dos predicados *generateWeightBoard/3*, do predicado *evaluatePiecesAround/6* e *getListOfPiecesAndMovesByScore/5*.
O predicado *generateWeightBoard/3* recebe os átomos List, com a lista de peças do jogador, Board, com o estado actual do tabuleiro e devolve WeightBoard, que será um tabuleiro onde as células têm representados valores correspondentes a pesos, atribuidos com base nos seguintes predicados.

```
1 weightOfOpponentCells(72).
2 weightOfEmptyCells(48).
```

```
3 weightOfPlayerCells(0).
```

Um tabuleiro de pesos poderá ter o seguinte valor:

```
1 [
2         [72,0,0,48,48,48],
3         [48,0,72,48,48,48],
4         [48,72,48,48,48,48],
5         [48,48,48,48,48,48],
6         [48,48,48,48,48,48],
7         [48,48,48,48,48,48]
8 ]
```

As células do jogador têm um peso de 0, ou seja, representam uma célula sem interesse de movimentação, uma célula vazia, com o valor 48, representa um interesse médio e uma célula com uma peça do adversário com o valor 72, será a mais interessante para mover a peça. Os valores dos pesos foram obtidos considerando que uma peça, no máximo terá 24 posições circundantes com peças do adversário. Fora deste círculo, uma peça adversária já não poderá chegar à peça do jogador, uma vez que, no máximo cada peça se pode mover apenas 2 células. Com o número 24, desenvolveram-se 3 patamares que serão utilizados para o cálculo dos pesos balanceados.

Os pesos balanceados, representam a possibilidade de uma peça em determinada posição poder ser eliminada pelo adversário. Para este cálculo, utiliza-se o predicado *evaluatePiecesAround/6*, que recebe a lista de peças do jogador, a lista de todas as peças em jogo, o tabuleiro, a posição da primeira linha do tabuleiro (na primeira chamada será sempre 1), a lista de pesos calculada anteriormente e devolve uma lista de pesos balanceados, semelhante à seguinte:

```
1 [
2         [71,1,2,47,48,48],
3         [48,3,71,48,48,48],
4         [47,71,47,47,48,48],
5         [47,47,47,47,48,48],
6         [47,48,47,48,48,48],
7         [48,48,48,48,48,48]
8 ]
```

Utilizando a lista de pesos originais, vai ser avaliada, para cada célula, quantas peças adversárias podem, na próxima jogada ir para essa célula. Por cada célula que se possa mover é removido um valor ao peso original, caso a célula em análise represente uma peça vazia ou uma peça do adversário. Se a célula em análise tiver uma peça do jogador, é adicionado um valor por cada peça adversária que se possa mover para essa localização. Deste modo consegue-se representar o real interesse de mover as peças do jogador para determinada posição e quais as pesas do jogador com maior ameaça.

Neste ponto, consegue-se perceber o interesse de ter 3 patamares diferentes de pesos. Sendo que, no máximo poderiam ser possíveis 24 possíveis ataques de peças adversárias para cada célula, consegue-se garantir que uma célula do adversário nunca terá menor interesse que

uma célula vazia. Este cenário embora exagerado, pois apenas existem 6 peças por jogador, dá uma noção mais quantitativa do interesse de mover uma peça para determinada posição. Por fim, o predicado *getListOfPiecesAndMovesByScore/5*, vai organizar, por peça e por jogada, os pesos obtidos com o predicado anterior. Obtendo-se uma lista com o seguinte formato:

```
[
        [Score1,[Piece1,PosX,PosY]],
        [Score2,[Piece1,PosX,PosY]],
        ...
]
```

Score é a pontuação atribuída a mover a Peça (Piece) para a posição (PosX, PosY). Com esta lista é fácil obter a jogada com a maior pontuação.
Para a escolha das posições para colocar pegs utiliza-se uma estratégia semelhante à de movimentação mas separada por duas estratégias que representam dois estados diferentes do jogo.
A primeira estratégia, verifica se os jogadores têm apenas 2 peças cada um, nesse caso, a vitória é conseguida preenchendo os pegs numa das peças. Para isto, é utilizado o predicado *strategyFillPegs/5* que vai escolher a peça com o maior número de pegs e avaliar as posições apenas nessa peça.
A segunda estratégia, irá avaliar as jogadas disponíveis em todas as peças, para no final escolher o conjunto peça e posição com a melhor pontuação. Isto é realizado no predicado *strategyGoFurther/5*.
A avaliação da colocação de pegs em determinada posição, é feita com base no tabuleiro de pesos balanceados e calculando qual jogada originará uma maior pontuação, assumindo que o tabuleiro estará na mesma posição quando o jogador voltar a jogar. Este pressuposto, apesar de inválido, não está totalmente errado, uma vez que a jogada do adversário não irá alterar muito o tabuleiro e a colocação de pegs será prioritária para peças não ameaçadas, logo a peça onde se colocar o peg ainda estará no tabuleiro na próxima jogada. Para dar um maior grau de liberdade de movimentação às peças, permitindo deslocações maiores, tanto para escapar como para capturar peças, as posições para coloção de pegs que permitam uma movimentação de duas casas têm um acrescimo de pontuação de 1 valor.
Em caso de empate, tanto para mover peças, como para colocar pegs, é escolhida uma combinação aleatória, dos maiores pesos, para evitar um padrão nas escolhas do computador.

3.7 Jogada do Computador

Novamente, dada a complexidade e o processo iterativo de uma jogada, não é possível descrever todo o processo apenas com um predicado *choose_move(+Board, +Level, -Move)* tal como sugerido.
As jogadas do computador são o resultado de todas as validações apresentadas anteriormente. Escolha da peça, movimento, escolha de peças e de posições para colocar pegs. Todas estas movimentações são realizadas pelo predicado *gameLoop/7*, que é chamada logo após as inicializações do jogo e continua em ciclo até o jogo terminar.

```
gameLoop(1,Player,P1Type,P2Type,ListOfPieces,Board,_W):-
```

```
2           getCurrentPlayerType(Player,P1Type,P2Type,Type),
3           move(Player,Type,ListOfPieces,Board,BoardMove,ListOfPiecesMove),
4           game_over(1,BoardMove,ListOfPiecesMove,NextLoop,Winner),
5           gameLoop(NextLoop,Player,P1Type,P2Type,ListOfPiecesMove,BoardMove,Winner).
6
7   gameLoop(2,Player,P1Type,P2Type,ListOfPieces,Board,_W):-
8           getCurrentPlayerType(Player,P1Type,P2Type,Type),
9           playerAddPeg(Player,Type,Board,ListOfPieces,ListOfPiecesPeg),
10          game_over(2,Board,ListOfPiecesPeg,NextLoop,Winner),
11          gameLoop(NextLoop,Player,P1Type,P2Type,ListOfPiecesPeg,Board,Winner).
12
13  gameLoop(3,Player,P1Type,P2Type,ListOfPieces,Board,_W):-
14          getCurrentPlayerType(Player,P1Type,P2Type,Type),
15          playerAddPeg(Player,Type,Board,ListOfPieces,ListOfPiecesLastPeg),
16          game_over(3,Board,ListOfPiecesLastPeg,NextLoop,Winner),
17          nextPlayer(Player,NextPlayer),
18          gameLoop(NextLoop,NextPlayer,P1Type,P2Type,ListOfPiecesLastPeg,Board,Winner).
19
20  gameLoop(4,_P,_T1,_T2,_LP,_B,Winner):-
21          print_gameOver_message(Winner).
```

O Loop 1 encarrega-se de escolher e mover a peça, verifica se após a movimentação o jogo terminou e chama o próximo loop. O Loop 2, adiciona o primeiro peg numa peça, verifica se o jogo terminou e chama o próximo loop. O Loop 3, adiciona o segundo peg numa peça, verifica se o jogo terminou, actualiza o próximo jogador e chama o loop 1 caso o jogo não tenha terminado. Por fim, o Loop 4, é chamado caso o jogo termine e imprime uma mensagem com o jogador que ganhou.

4 CONCLUSÕES

O desenvolvimento deste trabalho, permite concluir que a linguagem Prolog é bastante eficiente para desenvolver sistemas com inteligência artifial. Sendo uma linguagem baseada em lógica, a forma de pensar de um humano é rapidamente traduzida na linguagem.
O desenvolvimento do trabalho foi também bastante enriquecedor, pois permitiu desenvolver uma forma de pensar em programação diferente da tradicional. Foi um desafio no início, mas com o desenrolar do trabalho tornou-se mais natural pensar em lógica e colocar de lado de lado um pensamento mais imperativo de realizar as diferentes accões e obter os objectivos.
É também interessante de avaliar a quantidade de código escrita, que é bastante reduzida quando comparada com outras linguagens.
Em termos de melhorias, pode-se apontar uma maior robustez na forma como os inputs são tratados, uma melhor interface com o utilizador e uma maior quantidade de testes para aumentar a fiabilidade dos predicados e prever mais situações.

REFERÊNCIAS

[1] Board Game Geek. Eigenstate: Board game, 2018. URL: https://boardgamegeek.com/boardgame/250725/eigenstate.

[2] Board Game Geek. The duke: Board game, 2013. URL: https://boardgamegeek.com/boardgame/36235/duke.

[3] Board Game Geek. Onitama: Board game, 2014. URL: https://boardgamegeek.com/boardgame/160477/onitama.

[4] Martin Grider. Eigenstate on bgg, 2018. URL: http://chesstris.com/2018/04/15/eigenstate-on-bgg.

[5] Martin Grider. Eigenstate pnp rules, 2018. URL: https://tinyurl.com/eigenstate.

[6] Swi Prolog. Swi Prolog Reference Manual, 2018. URL: http://www.swi-prolog.org/pldoc/doc_for?object=manual.

[7] Patrick Blackburn, Johan Bos, and Kristina Striegnitz. Learn Prolog Now, 2012. URL: http://www.learnprolognow.org/.