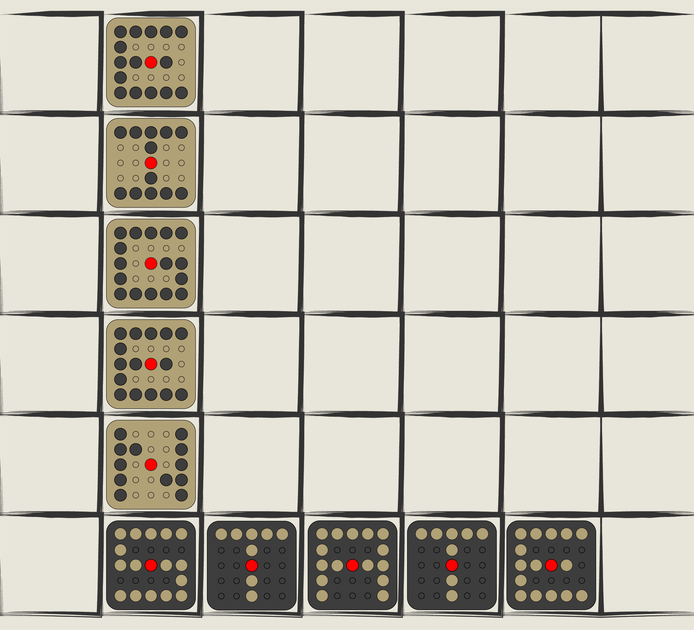


**Relatório**

Programação em Lógica



André Filipe Pinto Esteves [up201606673@fe.up.pt](mailto:up201606673@fe.up.pt)

Luís Diogo dos Santos Teixeira da Silva [up201503730@fe.up.pt](mailto:up201503730@fe.up.pt)

**1. Introdução**

Objetivos do trabalho.

**2. O jogo Eigenstate**

**2.1 História**

Eigenstate foi criado em 2016 por Martin Grider, um desenvolvedor de software e criador de jogos amador americano. O jogo resulta de uma mistura entre as mecânicas de movimento de outros dois jogos abstratos mais aclamados: The Duke (criado por Jeremy Holcomb e Stephen McLaughlin) e Onitama (criado por Shimpei Sato). Um protótipo do jogo foi criado em 2017 e a sua primeira publicação está agendada para o final de 2018.

O termo Eigenstate refere-se ao possível movimento de uma partícula na física quântica. O autor do jogo refere que, apesar de tangencial, poder-se-ia atribuir ao jogo este tema, sendo as peças partículas num sistema e o objetivo de cada jogador é observar o sistema adversário primeiro.

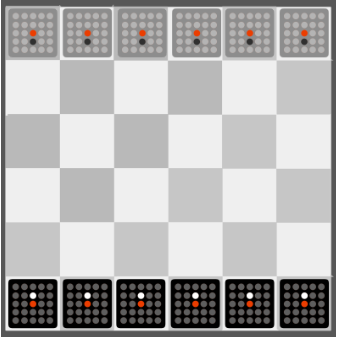
**2.2 Regras**

Eigenstate é um jogo de estratégia abstrata para duas pessoas com regras simples de complexidade crescente com o decorrer do jogo.

Inicialmente cada jogador seleciona a cor com que pretende jogar.

*Peça*

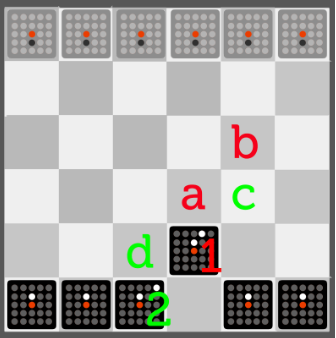
*Pino central*

Cada peça começa com dois pinos, o pino no centro representa a posição no tabuleiro e um pino adicional permitindo a peça ser movida um espaço para a frente. (figura 1)

*Pino dos movimentos possíveis*

Em cada jogada o jogador move uma das suas peças e acrescenta duas eigenstates (pinos) em qualquer uma das suas peças aumentando assim as possibilidades de movimento das peças nas seguintes jogadas. Todos os pinos numa peça exceto o centro representam os movimentos possíveis relativos à posição da peça no tabuleiro.

Figura 1

 Por exemplo (figura 2), o jogador com as peças pretas, na sua jogada, moveu a peça 1 para a frente, e, de seguida, adicionou um pino na peça 1 e na peça 2. Quando o jogador com as peças pretas voltar a jogar pode mudar a peça 1 para as posições a e b (pinos a branco representando os movimentos relativos possíveis) e a peça 2 para as posições c e d.

- Os pinos colocados nas peças nunca podem ser removidos. Assim, cada peça poderá sempre pelo menos ser movida para a frente.

Figura 2

- Peças podem saltar por cima de outras peças (movimento de 2 para c).

- Peças não podem mover-se para fora do tabuleiro.

- Peças não podem ser rodadas.

- Uma peça não pode mover-se para trás a não ser que tenha um pino atrás do pino central.

- Quando uma peça é movida para uma posição onde existe outra peça (do próprio jogador ou do adversário), essa peça é removida do jogo (não sendo, obviamente, uma boa ideia remover as próprias peças).

**Posicionamento dos pinos**

Os pinos têm de ser colocados nos espaços disponíveis das próprias peças que ainda estão em jogo. Em cada jogada os dois pinos podem ser colocados em diferentes peças.

**Objetivo**

O objetivo principal é reduzir o oponente a apenas uma peça restante.

Como objetivo secundário, usado quando os dois jogadores têm exatamente duas peças restantes, o jogador deve, em vez de seguir o objetivo principal, (o qual seria impossível de alcançar, a não ser que seja propositado), preencher todos os pinos de uma das peças restantes. Assim, o jogo nunca acaba em empate.

**3. Lógica do Jogo**

**3.1 Representação do Estado do Jogo**

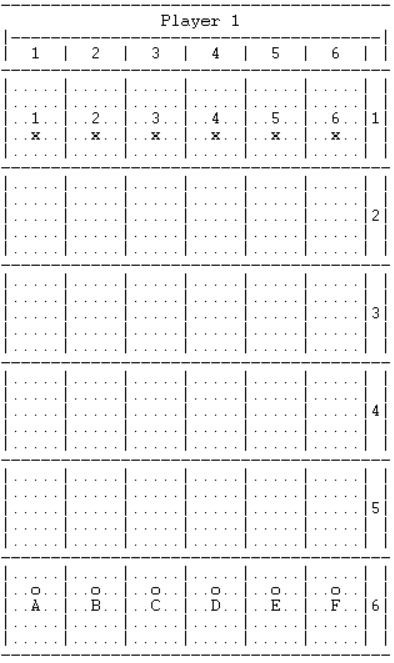
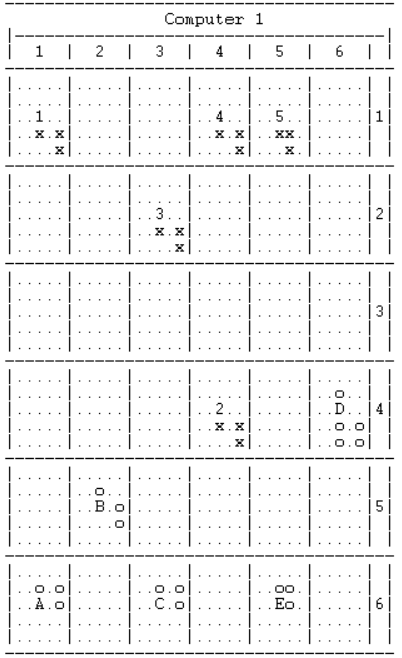
 No estado inicial o tabuleiro é disposto com 6 peças para cada jogador, em linha no topo e no fundo do mesmo, tal como mostra a seguinte figura 3. Cada peça é iniciada com um movimento possível, mover-se uma posição em frente.

Figura - Estado Inicial



Num estado intermédio percebe-se que as várias peças já se movimentaram para possíveis espaços e que já existem vários pinos definidos distribuídos pelas peças. Também é possível dizer que neste momento do jogo ambos os jogadores perderam uma peça, peça F do jogador 1 e a peça 6 do jogador 2. Print retirado do modo de jogo Computer vs Computer (Opção 3), dificuldade fácil, figura 4.

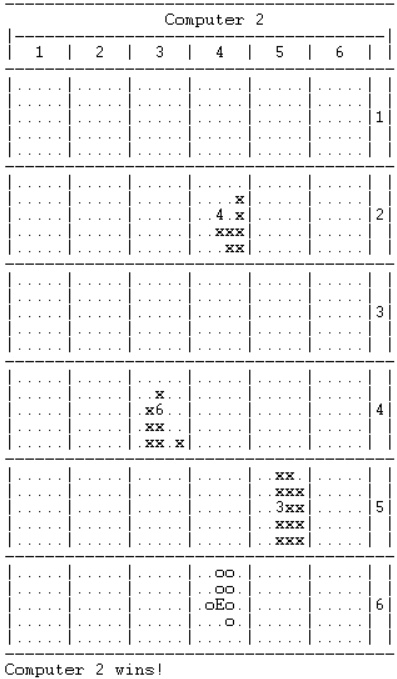


Figura - Estado Intermédio - Computer vs Computer

Figura - Estado Final - Computer 2 ganhou

Por fim num estado final, apenas um jogador é vitorioso, devendo reduzir o oponente a uma peça apenas ou completar na sua totalidade os pinos de uma peça. Na figura apresentada a seguir é possível ver o estado final em que o Computer 2 reduziu o adversário a uma peça.

Figura - Estado Intermédio - Computer vs Computer

**3.2 Visualização do Tabuleiro**

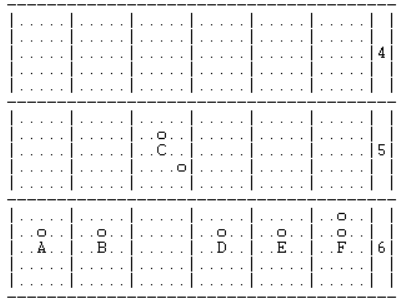
|  |
| --- |
| /\* Displays top of the board view \*/  display\_horizontal:- |
|  |
| /\* Displays name of the last player to make a move |
|  |
| Name - the name of the player that made the last move |
| \*/ |
| display\_head(Name) :- |
|  |
| /\* Displays board |
|  |
| Displays board head and name followed by the board itself |
|  |
| [Head|Tail] - board to be displayed |
| Player - the player that made the last move |
| \*/ |
| display\_board([Head|Tail],Player1) :- |
|  |
| /\* Displays board itself |
|  |
| Displays all rows of the board |
|  |
| Usage: display\_board\_aux(+Board, \_, 6) |
|  |
| [Head|Tail] - the portion of the board yet to be displayed |
| X\_val - the line currently being displayed |
| \*/ |
| display\_board\_aux([], \_, \_) :- |
| display\_board\_aux([Head|Tail], \_, X\_val) :- |
| /\* Displays a line of the board \*/ |
| display\_line([], \_, 1, \_) :- write('| |'). |
| display\_line([Head|Tail], Original, N\_line, X\_val) :- |
| display\_line([], Original, 3, X\_val) :- |
| display\_line([], Original, N\_line, X\_val) :- |
| /\* Displays a Piece line \*/ |
| display\_piece\_line(N\_line, [\_|Tail], Current\_line) :- |
| display\_piece\_line(N\_line, [Head|\_], N\_line) :- |
| /\* Displays each pin element \*/ |
| display\_piece\_element([]). |
| display\_piece\_element([Head|Tail]) :- |

**3.3 Lista de Jogadas Válidas**

Função que retorna todas as jogadas possíveis para movimentação.

Para cada peça em *Pieces* é verificada se está no atual tabuleiro também dado como argumento. Se sim, então é chamada a função *getAvailableEvalMoves* que retorna a lista de movimentos possíveis para e peça em questão analisando os seus pinos de movimento possível, já verificando os eventuais movimentos que leva a peça para fora do tabuleiro. Caso a peça não esteja no tabuleiro atual é criada uma lista vazia. Seguindo isto, essa lista, *PieceMoves*, é adicionada à lista principal *Moves,* que fica em *NextMoves*. Sendo este processo recursivo feito para todas as peças no argumento *Pieces.*

|  |
| --- |
| valid\_moves(\_, [], Moves, Moves). |
| valid\_moves(Board, [P|Pieces], Moves, Return) :- |
| if\_then\_else(has\_element\_matrix(P, Board), |
| getAvailableEvalMoves(Board, P, [], PieceMoves, 5, 5), |
| black(PieceMoves)), |
| append(Moves, PieceMoves, NextMoves), |
| valid\_moves(Board, Pieces, NextMoves, Return). |
|  |
| black([]). |



Exemplo da lista de movimentos possíveis quando o Computer 1 está para jogar, figura 6.

Figura 6 - Estado do jogo para print da lista de movimentos

**3.4 Execução de Jogadas – Interface com o Utilizador**

|  |
| --- |
| p1player :-  chooseMovePiece([pA, pB, pC, pD, pE, pF]), |
| \+choosePinPiece([pA, pB, pC, pD, pE, pF]), |
| \+choosePinPiece([pA, pB, pC, pD, pE, pF]). |

Quando é a vez do utilizador (seja jogador 1 ou jogador 2), é seguido os seguintes passos.

|  |
| --- |
| chooseMovePiece(Pieces) :-  write('Choose piece to move (ex. pX): '), |
| read(Piece), nl, |
| if\_then\_else(validPieceLoop(Piece, Pieces), chooseMove(Piece),callback\_chooseMovePiece(Pieces)). |
|  |

|  |
| --- |
| validPieceLoop(Piece, Pieces) :-  has\_element(Piece, Pieces), |
| board(T), |
| has\_element\_matrix(Piece, T). |

|  |
| --- |
| callback\_chooseMovePiece(Pieces) :-  write('Invalid Piece. '), |
| chooseMovePiece(Pieces). |
|  |

Interface com o utilizador que lê uma peça para ser jogada, se jogador 1 [pA, …, pF], se jogador 2 [p1, …, p6]. Caso o input lido pelo utilizador seja inválido, verificado pela função *validPieceLoop*, então é assumido que é uma peça inválida e volta a pedir ao utilizador outra vez por uma peça válida. Caso seja uma peça válida, isto é, que está no tabuleiro atual, a função *chooseMove* é executada.

|  |
| --- |
| chooseMove(Piece) :-  board(T), |
| index(T, Row, Col, Piece), |
| getAvailableMoves(Piece, [], Moves, 5, 5), |
| write('Available moves [row, column]: '), |
| write(Moves), nl, |
| write('Choose target row: '), |
| read(TrgRow), |
| write('Choose target column: '), |
| read(TrgCol), |
| if\_then\_else(validMoveLoop(Moves, TrgRow, TrgCol), move(Piece, Row, Col, TrgRow, TrgCol), callback\_chooseMove(Piece)). |
|  |

|  |
| --- |
| callback\_chooseMove(Piece) :-  write('\nInvalid target. '), |
| chooseMove(Piece). |

|  |
| --- |
| validMoveLoop(Moves, TrgRow, TrgCol) :-  has\_element([TrgRow, TrgCol], Moves). |

Esta função é usada para o utilizador definir qual o movimento a fazer dentro dos possíveis de acordo com os pinos da peça. A função *index*, neste exemplo, devolve a coluna e a linha da peça recebida da função anterior para depois ser usada na função principal que trata do movimento da peça. Imprime na consola todas os movimentos possíveis da peça. Lê do utilizador a linha seguida da coluna validando o mesmo com a função *validMoveLoop* da mesma maneira que a escolha da peça. Quando é recebida um movimento válido é chamada a função *move* com a Peça a ser movida, a linha e coluna da sua posição atual e a linha e coluna da nova posição. Código da função *move* em seguida.

|  |
| --- |
| move(Piece, Row, Col, TrgRow, TrgCol) :-  board(T), |
| setElemMatrix(Row, Col, 'e', T, NewBoard), |
| setElemMatrix(TrgRow, TrgCol, Piece, NewBoard, FinalBoard), |
| retract(board(T)), |
| assert(board(FinalBoard)). |

Da mesma forma descrita em cima (código semelhante) é feito a leitura do input do utilizador para a posição dos pinos. É possível colocar dois pinos em qualquer uma das suas peças escolhendo primeiro a peça a inserir pino, depois é impresso todos os pinos livres para seres colocados escolhendo um para ser colocado, isto é, verificando a sua possibilidade como descrito anteriormente. Este processo é executado duas vezes pois são dois pinos adicionados por jogada.

**3.5 Final do Jogo**

A cada jogada do ciclo do jogo, tanto após p1Turn como p2Turn é verificado o fim do jogo, *game\_over*.

|  |
| --- |
| gameloop(Player1, Player2, Difficulty) :-  p1Turn(Player1, Difficulty), |
| board(T), |
| display\_board(T,Player1), |
| game\_over(Player1, Player2), |
| p2Turn(Player2, Difficulty), |
| board(NewT), |
| display\_board(NewT, Player2), |
| game\_over(Player1, Player2), |
| gameloop(Player1, Player2, Difficulty). |

A função *game\_over* verifica tanto para o Player1 como para o Player2 o fim do jogo, analisando os dois casos de terminação, isto é, se apenas existe uma peça do adversário restante ou se o jogador tem uma peça com os pinos todos preenchidos.

|  |
| --- |
| game\_over(Player1, Player2) :-  winnerP1(Player1), |
| winnerP2(Player2). |

|  |
| --- |
| winnerP1(\_) :-  checkPlayerPieces([p1, p2, p3, p4, p5, p6], 0), |
| checkP1Pins, !. |
|  |
| winnerP1(Player1) :- |
| write(Player1), |
| write(' wins!'), nl, nl, |
| fail. |

|  |
| --- |
| winnerP2(\_) :-  checkPlayerPieces([pA, pB, pC, pD, pE, pF], 0), |
| checkP2Pins, !. |
|  |
| winnerP2(Player2) :- |
| write(Player2), |
| write(' wins!'), nl, nl, |
| fail. |

*checkPlayerPieces* faz a contagem do numero de peças existentes no tabuleiro atualmente do oponente, se esta contagem for diferente de 1 é true, significa que ainda não ganhou, caso seja 1 significa que o oponente só tem uma peça o que significa que o jogo acabou imprimindo na consola o restivo vencedor.

|  |
| --- |
|  |
| checkPlayerPieces([], Cnt) :- Cnt \= 1.  checkPlayerPieces([P|Ps], Cnt) :- |
| Inc is Cnt+1, |
| board(T), |
| if\_then\_else(has\_element\_matrix(P, T), checkPlayerPieces(Ps, Inc), checkPlayerPieces(Ps, Cnt)). |

Para cada peça de jogador a verificar é feita a verificação se todos os pinos da mesma preenchidos retorna falso se a primeira peça que encontrar totalmente preenchida.

|  |
| --- |
| checkP1Pins :-  checkFullPins(pA), |
| checkFullPins(pB), |
| checkFullPins(pC), |
| checkFullPins(pD), |
| checkFullPins(pE), |
| checkFullPins(pF). |

|  |
| --- |
| checkP2Pins :-  checkFullPins(p1), |
| checkFullPins(p2), |
| checkFullPins(p3), |
| checkFullPins(p4), |
| checkFullPins(p5), |
| checkFullPins(p6). |

|  |
| --- |
| checkFullPins(Piece) :-  piece(Piece, Mat), |
| has\_element\_matrix('.', Mat). |

**3.6 Avaliação do Tabuleiro**

De modo a avaliar a melhor jogada foi definido a função de avaliação, ao qual é avaliado quatro métricas:

- Número de peças no tabuleiro.

- Número de pinos no tabuleiro.

- Número de jogadas possíveis.

- Número de peças que estão sobre captura do adversário no momento atual do tabuleiro.

|  |
| --- |
| value(Board, Pieces, Eval) :-  opPieces(Pieces, OpPieces), |
| calculatePieceVal(Board, Pieces, OpPieces, 0, Val1), |
| calculatePinVal(Board, Pieces, OpPieces, 0, Val2), |
| calculateTotalMovesVal(Board, Pieces, OpPieces, Val3), |
| calculatePieceCaptureVal(Board, Pieces, OpPieces, Val4), |
| Eval is Val1 + Val2 + Val3 + Val4. |

Por cada peça do jogador ainda no tabuleiro é somado 10 pontos, e por cada peça do oponente no tabuleiro é decrementado 10 pontos igualmente.

|  |
| --- |
| calculatePieceVal(\_, [], [], Val, Val). |
| calculatePieceVal(Board, [], [P|OpPieces], Val, Return) :- |
| NextVal is Val - 10, |
| if\_then\_else(has\_element\_matrix(P, Board), |
| calculatePieceVal(Board, [], OpPieces, NextVal, Return), |
| calculatePieceVal(Board, [], OpPieces, Val, Return)). |
| calculatePieceVal(Board, [P|Pieces], OpPieces, Val, Return) :- |
| NextVal is Val + 10, |
| if\_then\_else(has\_element\_matrix(P, Board), |
| calculatePieceVal(Board, Pieces, OpPieces, NextVal, Return), |
| calculatePieceVal(Board, Pieces, OpPieces, Val, Return)). |

Por cada pino existente nas peças do jogador do tabuleiro é incrementado 1 ponto e o mesmo acontece por cada pino que o oponente tem que é decrementado 1 ponto.

|  |
| --- |
| calculatePinVal(\_, [], [], Val, Val).  calculatePinVal(Board, [], [P|OpPieces], Val, Return) :- |
| piece(P, Mat), |
| countElemMatrix(Mat, 'o', 0, Cnt1), |
| countElemMatrix(Mat, 'x', 0, Cnt2), |
| TotalCnt is Cnt1 + Cnt2, |
| NextVal is Val - TotalCnt, |
| if\_then\_else(has\_element\_matrix(P, Board), |
| calculatePinVal(Board, [], OpPieces, NextVal, Return), |
| calculatePinVal(Board, [], OpPieces, Val, Return)). |
| calculatePinVal(Board, [P|Pieces], OpPieces, Val, Return) :- |
| piece(P, Mat), |
| countElemMatrix(Mat, 'o', 0, Cnt1), |
| countElemMatrix(Mat, 'x', 0, Cnt2), |
| TotalCnt is Cnt1 + Cnt2, |
| NextVal is Val + TotalCnt, |
| if\_then\_else(has\_element\_matrix(P, Board), |
| calculatePinVal(Board, Pieces, OpPieces, NextVal, Return), |
| calculatePinVal(Board, Pieces, OpPieces, Val, Return)). |

Por cada movimento possível que o jogador tem é incrementado 1 ponto, e por cada movimento possível que o oponente tem é decrementado 1 ponto. Por cada movimento em que o jogador consegue capturar é incrementado 3 pontos e por cada peça que esta sobre captura é decrementado 8 pontos.

|  |
| --- |
| calculateTotalMovesVal(Board, Pieces, OpPieces, Val) :-  valid\_moves(Board, Pieces, [], PlayerMoves), |
| length(PlayerMoves, Val1), |
| valid\_moves(Board, OpPieces, [], OppMoves), |
| length(OppMoves, Val2), |
| Val is Val1 - Val2. |
| calculatePieceCaptureVal(Board, Pieces, OpPieces, Val) :- |
| valid\_moves(Board, Pieces, [], PlayerMoves), |
| valid\_moves(Board, OpPieces, [], OpMoves), |
| moves\_into\_op\_piece(Board, PlayerMoves, OpPieces, 0, Val1), |
| moves\_into\_op\_piece(Board, OpMoves, Pieces, 0, Val2), |
| Val is (3 \* Val1) - (8 \* Val2). |
| moves\_into\_op\_piece(\_, [], \_, Val, Val). |
| moves\_into\_op\_piece(Board, [[\_, TrgRow, TrgCol]|Moves], OpPieces, Val, Return) :- |
| NextVal is Val + 1, |
| index(Board, TrgRow, TrgCol, Elem), |
| if\_then\_else(has\_element(Elem, OpPieces), |
| moves\_into\_op\_piece(Board, Moves, OpPieces, NextVal, Return), |
| moves\_into\_op\_piece(Board, Moves, OpPieces, Val, Return)). |

**3.7 Jogada do Computador**

Na vez do computador é efetuada a seguinte função *p1Computer* ou *p2Computer.*

|  |
| --- |
| p1computer(Depth) :-  random\_permutation([pA, pB, pC, pD, pE, pF], Pieces), |
| removeExtraPieces(Pieces, [], NewPieces), |
| board(T), |
| valid\_moves(T, NewPieces, [], Moves), |
| chooseBestMove(T, Moves, \_, BestMove, -1000, \_, NewPieces, Depth), |
| moveAI(BestMove), |
| pinAI(NewPieces), |
| pinAI(NewPieces). |

Inicialmente é usada a função *random\_permutation* para assim o Computer(bot) nunca jogar da mesma maneira a cada jogo. Com o decorrer do jogo as peças podem ser removidas por foram sobrepostas pelo adversário (perdidas), assim para saber quais ainda estão no tabuleiro é executada a função *removeExtraPieces* retornando assim as peças ainda em jogo, código dessa função abaixo.

|  |
| --- |
| removeExtraPieces([], Ret, Ret).  removeExtraPieces([P|Pieces], NewPieces, Ret) :- |
| board(T), |
| if\_then\_else(has\_element\_matrix(P, T), |
| removeExtraPieces(Pieces, [P|NewPieces], Ret), |
| removeExtraPieces(Pieces, NewPieces, Ret)). |

De seguida é retornado todas as jogadas possíveis no atual tabuleiro T pela função *valid\_moves*, acima enunciada, tópico 3.3. Posto isto é executada a função principal de AI, *chooseBestMove*. Retorna a melhor jogada avaliando a pontuação do jogo usando uma determinada profundidade(depth) de pesquisa (1 ou mais).

Se a depth for igual a 1, recebe o valor da atual posição teste e verifica se é melhor do que a atual melhor posição teste. Se a depth for maior que 1, cria uma posição teste fazendo uma jogada possível. O oponente move-se seguindo a melhor jogada possível, calculada usando a depth – 1. Troca a melhor jogada por aquela que a avaliação de retorno seja melhor que a atual.

Usage: *chooseBestMove(+Board, +Moves, \_, -BestMove, -1000, \_, +Pieces, +Depth)*

|  |
| --- |
| chooseBestMove(\_, [], MoveRet, MoveRet, EvalRet, EvalRet, \_, \_).  chooseBestMove(Board, [M|Moves], BestMove, MoveRet, BestMoveEval, EvalRet, Pieces, 1) :- |
| makeEvalMove(Board, NewBoard, M), |
| value(NewBoard, Pieces, Eval), |
| if\_then\_else(Eval > BestMoveEval, |
| chooseBestMove(Board, Moves, M, MoveRet, Eval, EvalRet, Pieces, 1), |
| chooseBestMove(Board, Moves, BestMove, MoveRet, BestMoveEval, EvalRet, Pieces, 1)). |
|  |
| chooseBestMove(Board, [M|Moves], BestMove, MoveRet, BestMoveEval, EvalRet, Pieces, Depth) :- |
| Depth \= 1, |
| opPieces(Pieces, OpPieces), |
| NewDepth is Depth - 1, |
| makeEvalMove(Board, NewBoard, M), |
| valid\_moves(NewBoard, OpPieces, [], OpMoves), |
| chooseBestMove(NewBoard, OpMoves, \_, BestOpMove, -1000, \_, OpPieces, NewDepth), |
| makeEvalMove(NewBoard, DepthBoard, BestOpMove), |
| valid\_moves(DepthBoard, Pieces, [], DepthMoves), |
| chooseBestMove(DepthBoard, DepthMoves, \_, \_, -1000, Eval, Pieces, NewDepth), |
| if\_then\_else(Eval > BestMoveEval, |
| chooseBestMove(Board, Moves, M, MoveRet, Eval, EvalRet, Pieces, Depth), |
| chooseBestMove(Board, Moves, BestMove, MoveRet, BestMoveEval, EvalRet, Pieces, Depth)). |

Seguido este processo todo, a peça retornada assumida como melhor opção é movida com a função *MoveAI,* no qual apenas usa a função *índex* para ter a linha e coluna da peça atualmente e usa a função *move* também acima descrita que trata do movimento da peça e sua atualização no tabuleiro.

|  |
| --- |
| moveAI([Piece, TrgRow, TrgCol]) :-  board(T), |
| index(T, Row, Col, Piece), |
| move(Piece, Row, Col, TrgRow, TrgCol). |

|  |
| --- |
| pinAI(Pieces) :-  choosePinPiece(Pieces, \_, Piece, 1000), |
| chooseBestPin(Piece, Row, Col), |
| pin(Piece, Row, Col). |

|  |
| --- |
| choosePinPiece([P|Pieces], Piece, PieceRet, PieceMoves) :-  getAvailableMoves(P, [], Moves, 5, 5), |
| length(Moves, NMoves), |
| if\_then\_else(NMoves < PieceMoves, |
| choosePinPiece(Pieces, P, PieceRet, NMoves), |
| choosePinPiece(Pieces, Piece, PieceRet, PieceMoves)). |

Neste momento é necessário selecionar os dois pinos a colocar nas peças do jogador. Num primeiro estado é apenas retornado a peça ao qual vai ser posto o pino, isto é feito da seguinte forma: percorre as peças que se podem mover, isto é têm pinos colocados para a peça se poder mover, e retorna a peça que tem menos movimentos possíveis. Isto é feito de forma a que todas as peças possuem pinos para se poderem movimentar no decorrer do jogo. Selecionada a peça é escolhido o melhor pino, e isto é feito de uma maneira semelhante, o pino selecionado é um que leva a peça a ter mais um movimento possível na posição atual da peça, isto faz com que não seja colocado um pino numa posição onde a peça já

não pode ser movimentada, por exemplo: a peça X está no lado esquerdo e se for colocado um pino que possibilita a peça mover-se uma posição para a esquerda não faz com que aumente as suas jogadas pois a peça continua a ter as mesmas jogadas possíveis antes da colocação do pino. Este processo é realizado duas vezes de forma a serem colocados dois pinos.

|  |
| --- |
| chooseBestPin(Piece, Row, Col) :-  getUnvailableMoves(Piece, [], UnMoves, 5, 5), |
| getAvailableMoves(Piece, [], Moves, 5, 5), |
| length(Moves, NMoves), |
| testBestPin(Piece, UnMoves, NMoves, \_, [Row|Col], 0). |

**4. Conclusões**

Luís, I guess

WARNING: Ver 3.7 a função que diz no enunciado é choose\_move(+Board, +Level, -Move). E nos não temos essa função

**Bibliografia**

**Luís’s brain.**