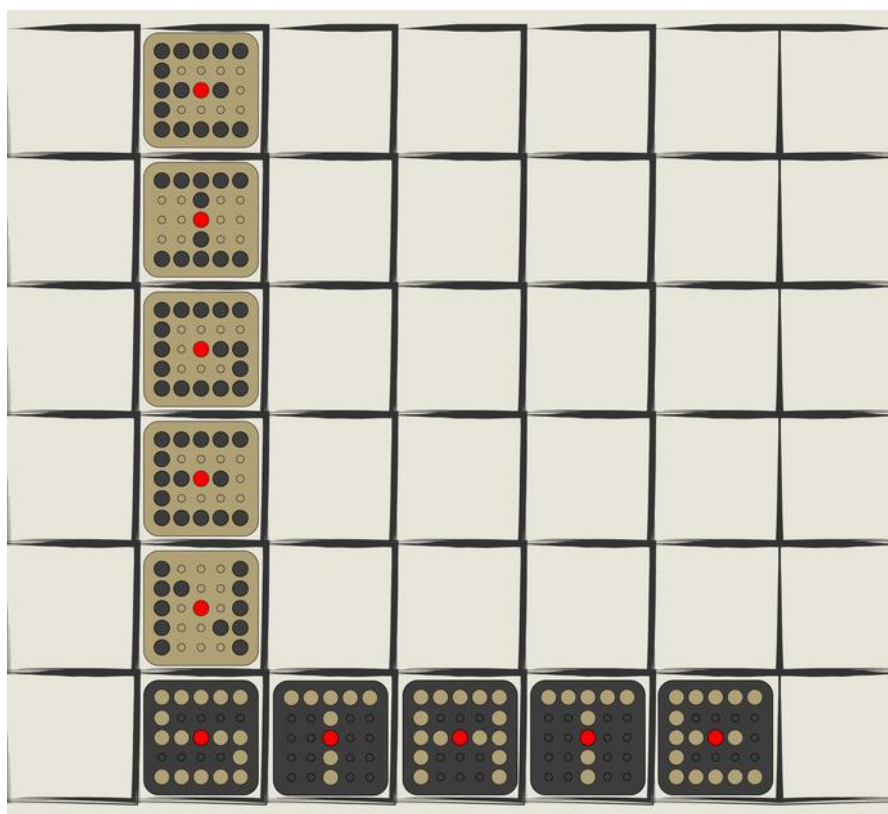


Relatório

Programação em Lógica



FEUP
Programação em Lógica – 2018/2019
Grupo: Eigenstate_1

André Filipe Pinto Esteves

up201606673@fe.up.pt

Luís Diogo dos Santos Teixeira da Silva

up201503730@fe.up.pt

1. Introdução

O objetivo deste trabalho era implementar, em linguagem Prolog, um jogo de tabuleiro, neste caso, o jogo *Eigenstate*. Entende-se pela implementação de um jogo de tabuleiro pela representação do seu tabuleiro e peças, da caracterização dos movimentos possíveis, entre outras regras, e pela verificação de condições de terminação do jogo, sejam estas vitória derrota ou empate – sendo que, no caso do *Eigenstate*, um empate é impossível.

Este projeto visa, portanto, o desenvolvimento de uma aplicação que permita jogar o jogo *Eigenstate* em três modos diferentes: humano contra humano, humano contra computador e computador contra computador. Além disso, o computador deve ser capaz de jogar usando pelo menos dois níveis de dificuldade diferentes. Neste projeto foram implementados três níveis de dificuldade.

A aplicação inclui também uma interface com o utilizador em modo de texto. Mais tarde, no âmbito da unidade curricular de LAIG, será desenvolvida uma interface 3D.

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 da partida.

Inicialmente cada jogador seleciona a cor com que pretende jogar.

Cada peça começa com dois pinos. O pino no centro representa a posição no tabuleiro e o pino adicional permite que a peça seja movida um espaço para a frente. (figura 1)

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

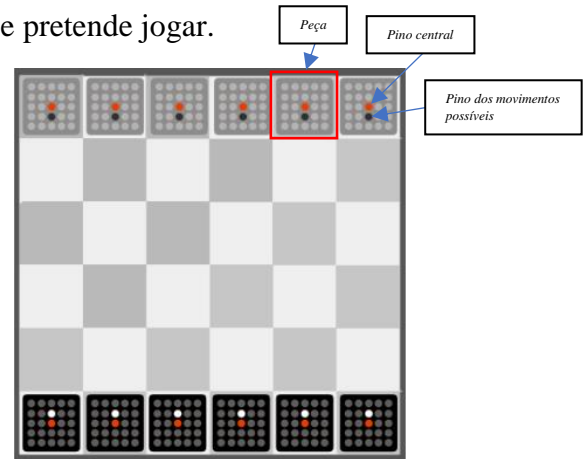


Figura 1

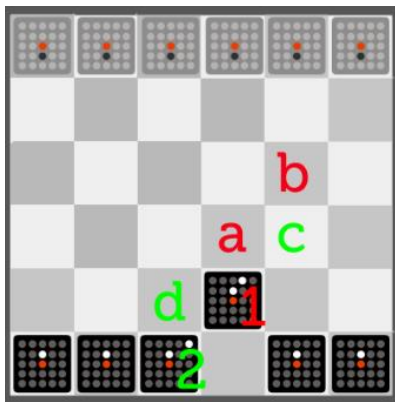


Figura 2

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 mover 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.

- 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, quando os dois jogadores têm exatamente duas peças restantes, cada 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 suas peças restantes. Assim, o jogo nunca acaba em empate.

[illegible]

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 5, é possível ver o estado final em que o Computer 2 reduziu o adversário a uma peça.

Internamente, o tabuleiro e as peças são representados, respetivamente, pelos predicados *board* e *piece*. O primeiro guarda uma lista de listas que representa o posicionamento das peças em relação umas às outras e o segundo faz corresponder cada peça aos pinos que lhe correspondem.

```
board( [[p1, p2, p3, p4, p5, p6],
        [e, e, e, e, e, e],
        [e, e, e, e, e, e],
        [e, e, e, e, e, e],
        [e, e, e, e, e, e],
        [pA, pB, pC, pD, pE, pF]])
```

```
piece(pC, [ [., ., ., ., .],
             [., ., 0, ., .],
             [., ., 'C', ., .],
             [., ., ., ., .],
             [., ., ., ., .] ]).
```

3.2 Visualização do Tabuleiro

Todas as imagens na secção anterior que representam estados de jogo foram criadas chamando a função *display_board*.

```
/* 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 a peça em questão, analisando os seus pinos de movimento possível e 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. De seguida, essa lista, *PieceMoves*, é adicionada à lista principal *Moves*, que fica em *NextMoves*. Este processo recursivo é realizado para todas as peças no argumento *Pieces*. Os movimentos são criados como uma lista de três argumentos, em que o primeiro é a peça em questão, o segundo é a linha do quadrado para o qual a peça se vai mover e o terceiro é a coluna do mesmo quadrado.

```
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 pronto para jogar (figura 6).

```
[pC,6,5],[pC,4,3],[pD,5,4],[pB,5,2],
[pE,5,5],[pA,5,1],[pF,5,6],[pF,4,6]]
```


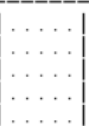



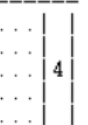

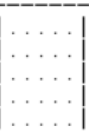
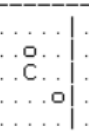


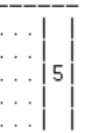
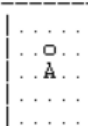
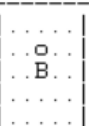
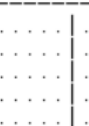

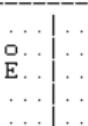
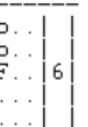
						4
						5
						6

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

3.4 Execução de Jogadas – Interface com o Utilizador

Quando é a vez do utilizador (seja jogador 1 ou jogador 2), são seguidos os seguintes passos:

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

```
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).
```

A interface com o utilizador lê uma peça a 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, de novo, por uma peça. 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)).
```

```
validMoveLoop(Moves, TrgRow, TrgCol) :-  
  has_element([TrgRow, TrgCol], Moves).
```

```
callback_chooseMove(Piece) :-  
  write('\nInvalid target. '),  
  chooseMove(Piece).
```

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 todos os movimentos possíveis da peça e lê do utilizador a linha e a coluna escolhidas, validando o mesmo com a função *validMoveLoop*, de maneira análoga à escolha da peça. Quando é recebido 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.

```
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)).
```

A leitura do input do utilizador para a posição dos pinos é feita de uma forma análoga. É possível colocar dois pinos em qualquer uma das suas peças, escolhendo primeiro a peça a inserir o pino, seguido da impressão de todos os espaços de pinos livres para seres colocado. É realizada a sua verificação, como descrito anteriormente. Este processo é executado duas vezes, pois que são adicionados dois pinos adicionados por jogada.

3.5 Final do Jogo

A seguir a cada jogada que é executada, é verificado se alguma das condições de fim de jogo foram alcançadas, fazendo uso da função *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 número de peças adversárias atualmente existentes no tabuleiro. Caso esta contagem seja diferente de 1, a função retorna *true*, o que significa que o jogo ainda não terminou. Em caso contrário, o oponente só tem uma peça, o que significa que o jogo terminou.

```
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 avaliar é feita a verificação se todos os pinos da mesma preenchidos.

```
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 uma função de avaliação, na qual são avaliadas quatro métricas:

- Número de peças no tabuleiro.
- Número de pinos no tabuleiro.
- Número de jogadas possíveis.
- Número de peças que estão sobre ameaça captura do adversário

```
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 são somados 10 pontos à avaliação e por cada peça do oponente no tabuleiro são subtraídos 10 pontos.

```
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 adicionado 1 ponto à avaliação e por cada pino que o oponente controla é subtraído 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 disponível é adicionado 1 ponto à avaliação e por cada movimento possível que o oponente tem disponível é subtraído 1 ponto. De notar que isto não é equivalente ao número de pinos nas peças, visto que existem pinos que não produzem, num determinado momento, movimentos válidos, visto estarem a apontar para lá da borda do tabuleiro.

```
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.
```

Finalmente, por cada movimento em que o jogador consegue capturar são adicionados 3 pontos à avaliação e por cada peça que esta sob ameaça de captura são subtraídos 8 pontos. Esta diferença de valor deve-se ao facto de termos considerado ser mais importante não expor peças a captura do que ameaçar capturar peças adversárias.

```

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

```

p1computer(Depth) :-
    random_permutation([pA, pB, pC, pD, pE, pF], Pieces),
    removeExtraPieces(Pieces, [], NewPieces),
    board(T),
    valid_moves(T, NewPieces, [], Moves),
    choose_move(T, Moves, _, BestMove, -1000, _, NewPieces, Depth),
    moveAI(BestMove),
    pinAI(NewPieces),
    pinAI(NewPieces).

```

Inicialmente é usada a função *random_permutation*, que dispõem as peças numa ordem aleatória para realizar desempates entre movimentos que a avaliação considere como equivalentes. Com o decorrer do jogo, peças podem ser removidas do tabuleiro. Assim, para saber quais ainda estão em jogo, é executada a função *removeExtraPieces*.

```

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, são retornadas todas as jogadas possíveis no atual tabuleiro T pela função *valid_moves*, acima enunciada (tópico 3.3). Depois é executada a função principal

de escolha de movimento, *choose_move*., que retorna a melhor jogada usando a função de avaliação determinada profundidade de pesquisa (1 ou mais).

Se a profundidade for igual a 1, recebe o valor da atual posição teste e verifica se é melhor do que a atual melhor posição teste. Em caso contrário, cria uma posição teste fazendo uma jogada possível. Assume-se que o oponente se move seguindo a melhor jogada possível, calculada usando a profundidade – 1. Troca a melhor jogada por aquela que a avaliação de retorno se mostre melhor que a atual.

```
choose_move(_, [], MoveRet, MoveRet, EvalRet, EvalRet, _, _).
choose_move(Board, [M|Moves], BestMove, MoveRet, BestMoveEval, EvalRet, Pieces, 1) :-
    makeEvalMove(Board, NewBoard, M),
    value(NewBoard, Pieces, Eval),
    if_then_else(Eval > BestMoveEval,
        choose_move(Board, Moves, M, MoveRet, Eval, EvalRet, Pieces, 1),
        choose_move(Board, Moves, BestMove, MoveRet, BestMoveEval, EvalRet,
Pieces, 1)).

choose_move(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),
    choose_move(NewBoard, OpMoves, _, BestOpMove, -1000, _, OpPieces, NewDepth),
    makeEvalMove(NewBoard, DepthBoard, BestOpMove),
    valid_moves(DepthBoard, Pieces, [], DepthMoves),
    choose_move(DepthBoard, DepthMoves, _, _, -1000, Eval, Pieces, NewDepth),
    if_then_else(Eval > BestMoveEval,
        choose_move(Board, Moves, M, MoveRet, Eval, EvalRet, Pieces, Depth),
        choose_move(Board, Moves, BestMove, MoveRet, BestMoveEval, EvalRet,
Pieces, Depth)).
```

Após todo este processo, o movimento assumido como melhor opção é executado através da função *MoveAI*.

```
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).
```

Neste momento é necessário seleccionar os dois pinos a colocar nas peças do computador. Num primeiro estado é apenas seleccionada a peça na qual vai ser colocado o pino. Este processo é realizado da seguinte forma: são percorridas todas as peças que se podem mover e é seleccionada 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. Seleccionada a peça é escolhido o melhor pino a colocar. Isto é calculado de uma forma semelhante. O pino seleccionado é aquele que leva a peça a ter mais um movimento possível na sua posição atual. Isto faz com que não seja colocado um pino numa posição que a peça não possa utilizar para se movimentar na próxima jogada. Este processo é realizado duas vezes, de forma a serem colocados dois pinos.

```
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)).
```

```
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

Consideramos que, apesar de termos cumprido tudo o que nos foi proposto, existiriam ainda alguns melhoramentos que poderiam ser feitos nesta aplicação.

No que diz respeito à interface de texto, apesar de existir verificação do input do utilizador, isto não evita que o programa termine com erro caso o número de argumentos inseridos em *read* seja o número errado. Não considerámos isto um problema grave, visto que uma nova interface vai ser desenvolvida para este jogo.

No que diz respeito à inteligência artificial, vários aspetos poderiam ser melhorados. No cálculo das jogadas, não são consideradas as colocações de pinos, mas apenas os movimentos das peças. A decisão para esta implementação foi feita devido ao facto de que a colocação de pinos aumenta imenso o número de jogadas possíveis, o que iria aumentar consideravelmente o tempo de processamento da inteligência artificial. Por exemplo, na primeira jogada do jogo, considerando apenas os movimentos das peças, existem 6 jogadas possíveis. Se considerarmos também a colocação de pinos, o número de combinações sobe para 56718. Mesmo assim, o tempo que o computador demora a processar a sua jogada no nível de dificuldade máximo ainda é mais elevado do que seria desejável (cerca de 5 ou 6 segundos). Para além disso, a forma como o computador coloca

os pinos podia ser melhor, tendo em conta que apenas considera se o número de jogadas possíveis no próximo turno aumenta.

Bibliografia

Slides das aulas teóricas

Bratko, Ivan (2012). Prolog programming for artificial intelligence (4th ed.). Harlow, England ; New York: Addison Wesley.