U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

NEUTREEKO

Relatório Final
Programação em Lógica

Francisco Manuel Canelas Filipe up**201604601**@fe.up.pt

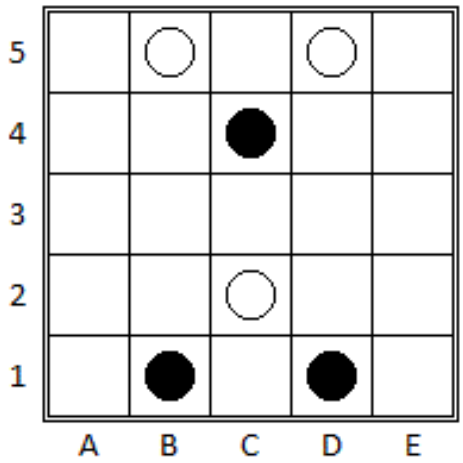
Pedro Miguel Sousa Fernandes up**201603846**@fe.up.pt

Índice

Introdução

No presente relatório descreve-se o funcionamento do jogo de tabuleiro Neutreeko, implementado com a linguagem Prolog. Através deste trabalho procuramos aplicar o conhecimento aprendido em Matemática Discreta e Programação em Lógica de modo a criar um simples jogo, e também aprender algo novo: utilizar algoritmos para desenvolver uma Inteligência Artificial.

Descrição do jogo



História

Neutreeko é um jogo de tabuleiro para dois jogadores criado por Jan Kristian Haugland em 2001. Segundo o mesmo, na altura de lançamento, já existia um jogo similar chamado Dao, mas acreditava não ter a mesma profundidade a nível tático. O nome do jogo deriva de Neutron e Teeko, outros dois jogos no qual este é baseado.

Regras

A configuração do tabuleiro é 5x5, e inicialmente cada jogador tem 3 peças, cujo posicionamento se pode ver na figura. Cada jogador é identificado pela cor das suas peças (branco ou preto).

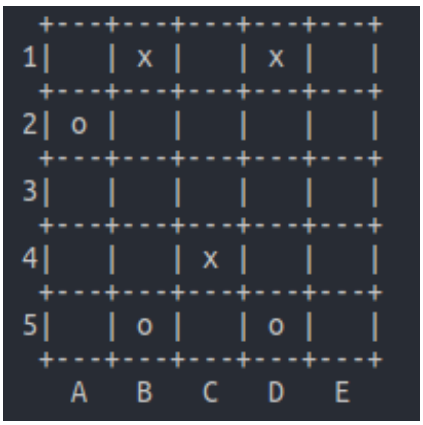
Objetivo: colocar as 3 peças em sequência (sem intervalo entre elas), seja na horizontal, vertical ou diagonal. Se a mesma posição ocorrer três vezes, dá-se um empate.

Movimentos: um movimento de uma peça corresponde a colocá-la, dada uma direção, na posição mais distante possível, mas válida (sem passar por posições já ocupadas ou sair do tabuleiro).

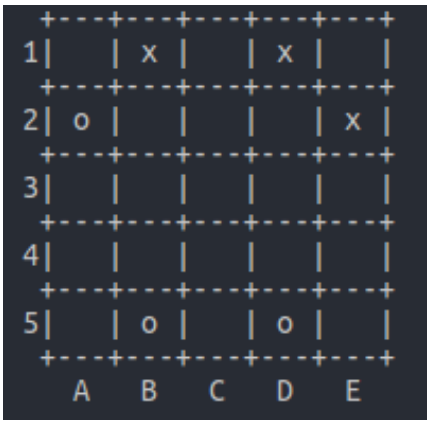
Jogada: os jogadores podem mover as suas peças alternadamente, mas o preto joga sempre primeiro.

Exemplos de Jogadas

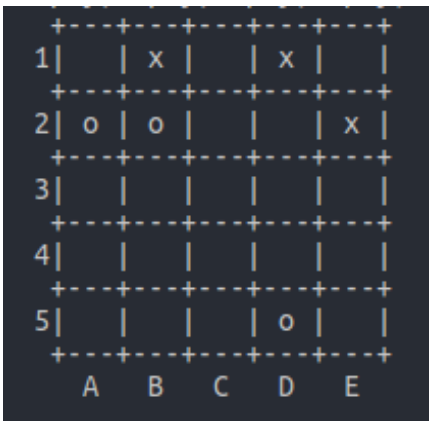
Começando a partir do princípio (como mostrado na imagem acima), as três primeiras jogadas poderiam ser como as seguintes:



4C -> 2E



5B -> 2B



Lógica do Jogo

Representação do Estado do Jogo

O tabuleiro será representado por uma matriz (lista de listas), onde cada elemento representa uma posição deste, sque pode estar vazia (*empty*) ou preenchida por uma peça (*white* ou *black*).

Exemplificação

Seguem-se alguns exemplos de possíveis estados do tabuleiro.

Estado inicial

Em qualquer jogo, o tabuleiro inicial tem sempre a mesma representação.

board([[empty, **white**, empty, **white**, empty],
[empty, empty, **black**, empty, empty],
[empty, empty, empty, empty, empty],
[empty, empty, **white**, empty, empty],
[empty**, black**, empty, **black**, empty]]).

Estados intermédios

board([[empty, **white**, empty, empty, **white**],
[empty, empty, empty, empty, empty],
[empty, empty, empty, empty, empty],
[empty, empty, **white**, empty, **black**],
[empty, empty, **black**, **black**, empty]]).
board([[empty, **white**, empty, **white**, empty],
[**black**, empty, empty, empty, empty],
[empty, empty, empty, empty, empty],
[empty, empty, empty, empty, empty],
[empty, **black**, **black**, **white**, empty]]).
board([[empty, empty, empty, **white**, empty],
[**white**, empty, empty, empty, empty],
[empty, **white**, empty, empty, empty],
[empty, empty, empty, empty, empty],
[empty, **black**, empty, **black**, **black**]]).

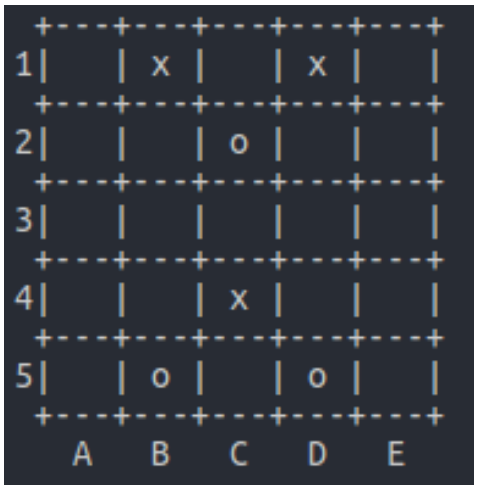
Estados finais

board([[empty, **white**, empty, empty, **white**],
[empty, empty, empty, empty, empty],
[empty, empty, empty, empty, empty],
[empty, empty, empty, empty, empty],
[empty, **white**, **black**, **black**, **black**]]).

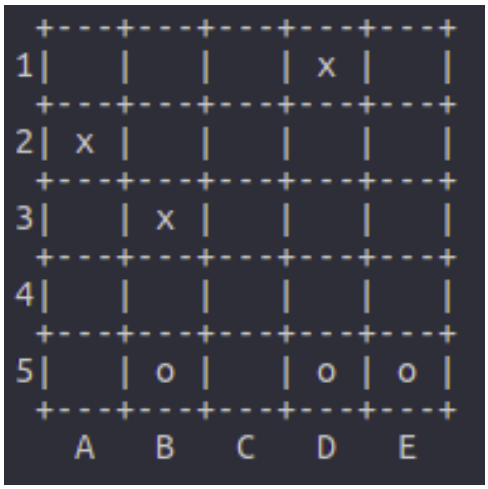
board([[empty, **white**, empty, **white**, empty],
[empty, empty, empty, **white**, empty],
[empty, empty, empty, empty, empty],
[empty, empty, empty, empty, empty],
[empty, **black**, **black**, **black**, empty]]).
board([[empty, empty, empty, empty, empty],
[**white**, empty, empty, empty, empty],
[empty, **white**, empty, empty, empty],
[empty, empty, **white**, empty, empty],
[**black**, **black**, empty, empty, **black**]]).

Visualização do Tabuleiro

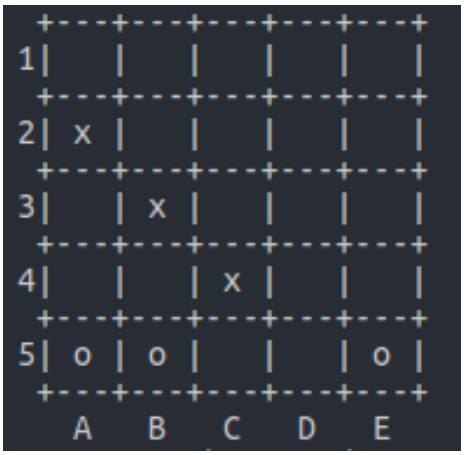
Estado inicial



Estado intermédio



Estado final



Código Desenvolvido

Desde a entrega intermédia, mudamos este código ligeiramente de forma ao utilizador poder identificar melhor as peças, acrescentando índices (números para as linhas e letras para as colunas).

```
display_game(Board, Player) :-
    nl,
    show_board(Board, 1),
    format('~n~nPlayer ~d is playing.', Player).

show_board([Head|Tail], I) :-
    write(' +---+---+---+---+---+'),
    nl,
    write(I),
    write('|'),
    display_line(Head),
    nl,
    NI is I+1,
    show_board(Tail, NI).
show_board([_], _I):-
    write(' +---+---+---+---+---+'),
    nl,
    write('   A   B   C   D   E').

display_line([Head | Tail]):-
    symbol(Head, S),
    write(S),
    write('|'),
    display_line(Tail).
display_line([]).
```

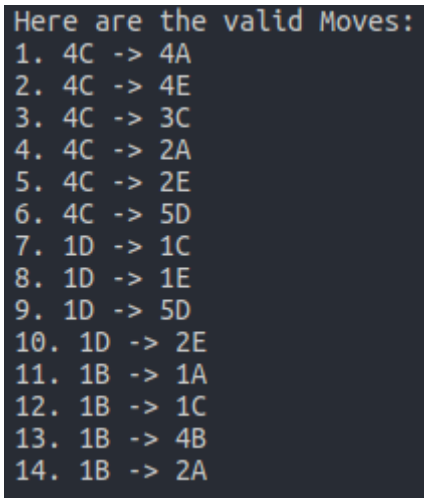
Lista de Jogadas Válidas

valid_moves(+Board, +Player, -ListOfMoves)

Em cada iteração do jogo, é gerada uma lista de jogadas válidas, que pode ser usada para apresentar ao jogador os movimentos que pode executar, ou para o computador poder selecionar o melhor movimento possível. Numa jogada de Neutreeko é possível mover uma peça em qualquer direção. Por esse motivo, a função principal usa três funções secundárias para verificar os movimentos possíveis na horizontal, vertical, e diagonal. Em cada uma delas, o algoritmo é semelhante, e

portanto vamos apenas dar o exemplo da horizontal. Para cada peça, itera-se sobre o índice i (coluna) mantendo o índice j (linha), primeiro para o lado esquerdo da peça e depois para o direito, enquanto não se encontrar um obstáculo (outra peça, ou limite do tabuleiro). Por isso, para cada peça pode haver entre 0 a 2 movimentos na mesma direção.

Pegando no exemplo do modo de jogo Jogador vs. Jogador, em cada iteração as jogadas válidas são mostradas como se pode ver na imagem ao lado. Depois, o jogador deve escolher, recorrendo ao índice, a jogada que pretende efetuar. Caso insira valores absurdos ser-lhe-á pedido um novo input até este inserir um valor válido para que o jogo possa prosseguir.



Código Desenvolvido

```
% Checks if a move is duplicate, meaning that its useless in practice
since the piece doesn't change places
is_duplicate([InitLine,InitCol,DestLine,DestCol]):-
    InitLine = DestLine,
    InitCol = DestCol.

% Discards duplicate moves
discard_duplicate_moves([], NewList, NewList).

discard_duplicate_moves([Head | Tail], TempList, NewList):-
        (is_duplicate(Head),discard_duplicate_moves(Tail, TempList,
NewList));
        (discard_duplicate_moves(Tail, [Head | TempList], NewList)).

% Generates a list of valid moves for each piece of the current player
valid_moves_piece(_Board,[],ListOfMoves,ListOfMoves).

valid_moves_piece(Board, [Head|Tail],List, ListOfMoves):-
    Init = Head,
    Curr = Head,
    valid_horizontal(Board, Curr, Init, List, HorMoves, -1),
    valid_vertical(Board, Curr, Init, HorMoves, HorVertMoves, -1),
    valid_diagonal(Board, Curr, Init, HorVertMoves, AllMoves, -1, -1),
    discard_duplicate_moves(AllMoves, [], NewAllMoves),
    valid_moves_piece(Board, Tail, NewAllMoves, ListOfMoves).
```

```
valid_horizontal(_Board, [_Line,_Col], [_InitLine,_InitCol] , Moves,
Moves , 3).

valid_horizontal(Board, [Line,Col] , [InitLine,InitCol] , List, Moves ,
Inc):-
    NextCol is Col + Inc,
    (
                ((NextCol = 0 ; NextCol = 6),
                 Move = [InitLine, InitCol,Line,Col],
                 NextInc is Inc + 2,
                 Next_Col is InitCol,
                 append(List,[Move],NewList),
                 valid_horizontal(Board, [Line,Next_Col] ,
[InitLine,InitCol] , NewList , Moves, NextInc)
               );
    get_piece(Line,NextCol,Board,Piece),
                ((Piece = black ; Piece = white),
                 Move = [InitLine, InitCol,Line,Col],
                 NextInc is Inc + 2,
                 Next_Col is InitCol,
                 append(List,[Move],NewList),
                 valid_horizontal(Board, [Line,Next_Col] ,
[InitLine,InitCol] , NewList , Moves, NextInc)
                );
             (valid_horizontal(Board, [Line,NextCol], [InitLine,InitCol] ,
List, Moves , Inc))
    ).

% Generates a list of valid moves for a given player
valid_moves(Board, Player, ListOfMoves):-
    get_pieces(Player, Pieces),
    valid_moves_piece(Board,Pieces,[], ListOfMoves).
```

Execução de Jogadas

move(+Move, +Board, -NewBoard)

O predicado **move** recebe como argumentos um movimento, uma lista composta pelos índices da peça inicial (antes do movimento ser efetuado) e da peça final (após o movimento ocorrer), bem como o tabuleiro inicial e final.

Esta função verifica qual o jogador atual, de forma a saber que tipo de peça é que tem que colocar no tabuleiro, e serve-se da função setPiece para, em duas chamadas, colocar uma peça *empty* no local inicial, e a peça desejada no local final.

Antes de terminar, atualiza na base de dados os valores dos átomos que guardam os índices de cada peça. Retorna o tabuleiro final em NewBoard, para ser posteriormente atualizado na base de dados. Este predicado não verifica a validade do argumento Move, pelo que se deve assegurar anteriormente através da chamada valid_moves.

Em contexto de jogo, esta função é chamada ap ós o movimento ser escolhido, seja pelo jogador (como explicado anteriormente), seja pelo computador, com recurso a uma função descrita mais à frente.

Código Desenvolvido

```
% Updates a piece variable
update_piece(InitLine,InitCol,DestLine,DestCol,1):-
    (p1_1(A,B), A = InitLine,B = InitCol, retract(p1_1(A,B)),
      assert(p1_1(DestLine,DestCol)));
    (p1_2(A,B), A = InitLine,B = InitCol, retract(p1_2(A,B)),
      assert(p1_2(DestLine,DestCol)));
    (p1_3(A,B), A = InitLine,B = InitCol, retract(p1_3(A,B)),
      assert(p1_3(DestLine,DestCol))).

update_piece(InitLine,InitCol,DestLine,DestCol,2):-
    (p2_1(A,B), A = InitLine,B = InitCol, retract(p2_1(A,B)),
      assert(p2_1(DestLine,DestCol)));
    (p2_2(A,B), A = InitLine,B = InitCol, retract(p2_2(A,B)),
      assert(p2_2(DestLine,DestCol)));
    (p2_3(A,B), A = InitLine,B = InitCol, retract(p2_3(A,B)),
      assert(p2_3(DestLine,DestCol))).
% Sets a piece on the board list
set_piece(1, 1, [[_El|Rest1]|Rest2], [[Piece|Rest1]|Rest2], Piece).
set_piece(1, N, [[Elem|Rest1]|Rest2], [[Elem|Head]|Rest2], Piece):-
    Next is N-1,
    set_piece(1, Next, [Rest1|Rest2], [Head|Rest2], Piece).
set_piece(N, NCol, [Elem |Rest1], [Elem|Out], Piece):-
    Next is N-1,
    set_piece(Next, NCol, Rest1, Out, Piece).
/**
 * Performs a move, changing the given piece to a new position,
 * and puts an empty piece on the original one.
 */
move([InitLine, InitCol, DestLine, DestCol], Board, NewBoard) :-
    nextPlayer(Player),
    if_then_else(Player = 1, set(black, Piece), set(white, Piece)),
    set_piece(InitLine, InitCol, Board, TempBoard, empty),
    set_piece(DestLine, DestCol, TempBoard, NewBoard, Piece),
    update_piece(InitLine,InitCol,DestLine,DestCol, Player).
```

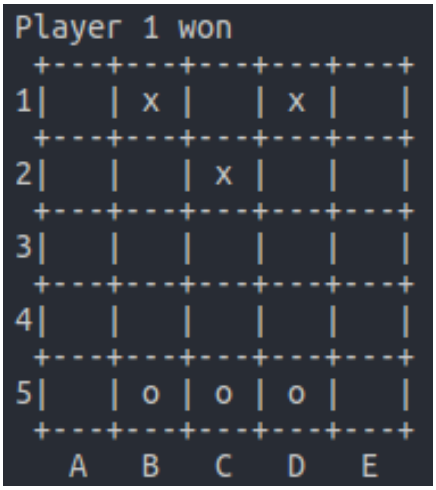
Final do Jogo

game_over(+Board, -Winner)

O jogo pode terminar em vitória (um jogador coloca as suas três peças em posições consecutivas) ou empate (a mesma configuração do tabuleiro ocorre três vezes). Para verificar a vitória, são usados três predicados auxiliares para verificar a vitória numa linha, coluna, ou diagonal. Para isso, existem variáveis que indicam a posição de cada uma das peças (linha e coluna), e após serem recolhidas, é verificado se estão em posições consecutivas, retornado o vencedor (*1* ou *2*). Caso nenhum destes casos se verifique, é retornado zero (o jogo ainda está a decorrer) ou *-1* (o jogo terminou em empate).

Para verificar o empate, é usada uma lista de tabuleiros, e uma lista que contém o número de vezes que o tabuleiro com o mesmo índice já ocorreu. Ou seja, a cada jogada, é verificado se o tabuleiro já pertence à lista. Em caso positivo, não é adicionado à lista de tabuleiros mas, na lista de contagem o tabuleiro respetivo vê o seu número incrementado uma vez. Em caso negativo, é acrescentado à lista de tabuleiros, e na lista de contagem é acrescentado um novo elemento com valor um. Desta forma assegura-se que o tamanho de ambas as listas é igual, e que os elementos que tenham o mesmo índice se relacionam. Assim, no predicado game_over, apenas é necessário saber se a lista de contagem contém o número três.

Na implementação dos modos de jogo, o predicado game_over é chamado no início de cada função, e verifica-se o valor de winner. Se for diferente de zero, o jogo termina, imprime uma mensagem, e retorna ao menu principal.



Código Desenvolvido

```
/*
 Checks if the same board configuration has happened 3 times (three-fold
repetition), in which case a draw occurs.
 */
game_over_draw(-1):-
    countOccurrences(Count),
    member(3, Count).
game_over_draw(0).
/**
 * Checks if a player has three consecutive pieces in a same row, thus
winning the game.
 */
game_over_row(2) :-
    get_white_pieces(Pieces),
    are_consecutive_hor(Pieces).
game_over_row(1) :-
    get_black_pieces(Pieces),
    are_consecutive_hor(Pieces).
/**
 * Checks if a player has three consecutive pieces in a same diagonal,
thus winning the game.
 */
game_over_diag(1) :-
    get_black_pieces(Pieces),
    are_consecutive_diag(Pieces).
game_over_diag(2) :-
    get_white_pieces(Pieces),
    are_consecutive_diag(Pieces).
/**
 * Checks if a player has three consecutive pieces in a same column, thus
winning the game.
 */
game_over_col(1) :-
    get_black_pieces(Pieces),
    are_consecutive_ver(Pieces).
game_over_col(2) :-
    get_white_pieces(Pieces),
    are_consecutive_ver(Pieces).
/**
 * Checks if three given pieces are consecutive in a board line.
 */
are_consecutive_hor([[F1,F2], [S1,S2], [T1, T2]]) :-
    F1=S1,
    S1=T1,
    are_numbers_consecutive([F2, S2, T2]).
/**
 * Checks if three given pieces are consecutive in a board column.
```

```
 */
are_consecutive_ver([[F1,F2], [S1,S2], [T1, T2]]) :-
    F2=S2,
    S2=T2,
    are_numbers_consecutive([F1, S1, T1]).
/**
 * Checks if three given pieces are consecutive in a board diagonal.
 */
are_consecutive_diag([[F1,F2], [S1,S2], [T1, T2]]) :-
    not(duplicate([[F1,F2], [S1,S2], [T1, T2]])),
    sort_by_x([[F1,F2], [S1,S2], [T1, T2]],Sorted),
    nth1(1,Sorted,First),
    nth1(2,Sorted,Middle),
    nth1(3,Sorted,Last),
    check_final_cond(First,Middle,Last).
check_final_cond([X1,MinY],[X2,MiddleY],[X3,MaxY]):-
    (MaxY - MinY) =:= 2,
    MiddleY > MinY,
    MaxY > MiddleY,
    abs(X2 - X1) =:= 1,
    abs(X3 - X2) =:= 1,
    abs(X3 - X1) =:= 2.
```

Avaliação do Tabuleiro

value(+Board, -Value, +Depth)

A função de avaliação do tabuleiro tem como critérios:

- **Avaliar se o jogador atual já ganhou ou perdeu o jogo:** Neste caso é utilizado o predicado ***check_win(Player,Board, Depth,Val).*** A sua função é verificar se o jogador ganhou o jogo, atualizando o valor da variável ***Val*** tendo em atenção a profundidade da árvore (Val = 100*Depth). Desta forma, as vitórias que aparecem mais cedo na árvore de jogadas possíveis vão ter maior valor, sendo consideradas melhores jogadas.
- **Avaliar as posições do jogador atual e do seu adversário verificando se estes possuem ou não peças consecutivas :** Este critério só é avaliado nas "folhas" da árvore de jogadas possíveis. Chama um predicado auxiliar(***value_aux(Player, Board, Val)***), que verifica se um dado jogador possui peças consecutivas. Caso isso aconteça atribui à variável Val um valor de 10 que simboliza a vantagem do jogador em causa. Esta chamada a

value_aux é executada para cada jogador do tabuleiro. A análise final do tabuleiro consiste em subtrair os valores de cada uma das chamadas e atribuindo o seu resultado a Val.

De uma forma geral a função de avaliação, a cada tabuleiro verifica se o jogador que vai efetuar a jogada já ganhou ou perdeu. Caso isso não aconteça, se o tabuleiro for uma "folha" da árvore de pesquisa, avalia-o em função da presença de peças consecutivas.

Código Desenvolvido

```
value(Board,Val, Depth):-
    currPlayer(P),
    nextPlayer(P,NextP),
    (
        (
            check_win(P,Board,Depth,Val)
            ;
            (
                check_win(NextP,Board,Depth,ValL),
                Val is 0 - ValL,
                retract(lost(_)),
                assert(lost(1))
            )
        )
        ;
        (
            Depth =:= 1,
            value_aux(P,Board,Val1),
            value_aux(NextP,Board,Val2),
            Val is Val1 - Val2
        )
        ;
        Val is -2000
    ).

check_win(Player,Board, Depth,Val):- game_over_ai(Board,Player),Val is
100 * Depth.
% Player has advantage (2 consecutive pieces)
value_aux(Player, Board, 10):-
    player_piece(Player,Piece),
    get_pieces(Board, Piece, [[F1,F2], [S1,S2], [T1,T2]]),
    are_consecutive_ai([[F1,F2], [S1,S2]]);
    are_consecutive_ai([[F1,F2], [T1,T2]]);
    are_consecutive_ai([[S1,S2], [T1,T2]]).
value_aux(_,_Board, 0).
```

Jogada do Computador

choose_move(+Board, +Depth, -NextBoard)

Este predicado é responsável por, através de funções auxiliares, gerar a árvore de movimentos possíveis e escolher a melhor jogada a efetuar.

Inicialmente este predicado inicializa os valores das variáveis dinâmicas com valores default, realizando assim um "reset" para repôr os valores inicias.

De seguida executa uma chamada à função ***choose_move_aux(+Pos,+Depth)***, que é responsável por gerar o melhor tabuleiro possível e guardá-lo na variável dinâmica **currMaxBoard**. Esta função gera, para cada tabuleiro, uma lista de jogadas possíveis. A partir desse tabuleiro e da lista de jogadas gerada, são simulados novos tabuleiros para serem avaliados. A primeira geração de tabuleiros vai conter o tabuleiro pretendido (a melhor jogada), no entanto é necessário avaliar a àrvore na sua totalidade, percorrendo todas as gerações. Desta forma usamos uma variável dinâmica (***currRoot***) que guarda o tabuleiro atual da primeira geração de tabuleiros. Como a pesquisa da árvore é feita em profundidade em vez de largura, começando no currRoot e avaliando todas as gerações que dele descendem, conseguimos obter o melhor cenário para esta jogada, tendo em atenção que caso alguma geração descendente resulte na derrota do jogador, esta jogada (***currRoot***) é automaticamente descartada. A derrota é guardada numa variável dinâmica ***lost*** que atua como flag (1 caso aquela jogada resulte em alguma derrota/ zero caso contrário).

Percorrendo a primeira geração e efetuando o mesmo raciocínio é possível obter a melhor jogada a efetuar.

Para o processo de gerar continuamente novas gerações de tabuleiros descendentes uns dos outros é utilizado o predicado ***best(+List, Depth)***, que recebe uma lista de tabuleiros e uma profundidade. Este predicado, atualiza a profundidade e, recorrendo ao ***choose_move_aux(),*** gera para cada tabuleiro da lista, uma nova geração de tabuleiros. Caso a geração atual seja a última a ser gerada, então procede-se à avaliação de cada um dos tabuleiros da lista.

Código Desenvolvido

```
:- dynamic(currPlayer/1).
:- dynamic(currMax/1).
:- dynamic(currMaxBoard/1).
:- dynamic(currRoot/1).
:- dynamic(lost/1).
:- dynamic(previousMax/1).
:- dynamic(previousMaxBoard/1).
currPlayer(0).
currMax(-2000).
currMaxBoard([]).
previousMax(-2000).
previousMaxBoard([]).
currRoot([]).
lost(0).

choose_move(Board, Depth, NextBoard):-
    nextPlayer(Player),
    retract(currPlayer(_)),
    assert(currPlayer(Player)),
    retract(currMax(_)),
    assert(currMax(-2000)),
    retract(currMaxBoard(_)),
    assert(currMaxBoard([])),
    retract(previousMax(_)),
    assert(previousMax(-2000)),
    retract(previousMaxBoard(_)),
    assert(previousMaxBoard([])),
    retract(currRoot(_)),
    assert(currRoot([])),
    retract(lost(_)),
    assert(lost(0)),
    choose_move_aux([Player,play,Board], Depth),
    currMaxBoard(NextBoard),
    update_pieces_ai(Player, NextBoard).

choose_move_aux([_Player,_State,Board], Depth):-
    difficulty(Diff),
    Depth =:= Diff - 1,
    retract(currRoot(_)),
    assert(currRoot(Board)),
    difficulty(Diff),
    lost(L),
    (
        (
            L =:= 1,
            previousMax(PMax),
            previousMaxBoard(PMaxBoard),
```

```
            retract(lost(_)),
            retract(currMax(_)),
            retract(currMaxBoard(_)),
            assert(lost(0)),
            assert(currMax(PMax)),
            assert(currMaxBoard(PMaxBoard))
        )
        ;
        (
            currMax(Max),
            currMaxBoard(MaxBoard),
            retract(previousMax(_)),
            retract(previousMaxBoard(_)),
            assert(previousMax(Max)),
            assert(previousMaxBoard(MaxBoard))
        )
    ),
    fail.
choose_move_aux([_Player,_State,Board], Depth):-
    difficulty(Diff),
    Depth \= Diff,
    value(Board, Val, Depth),
    currMax(Max),
    currRoot(Root),
    (
        (
            Val > Max,
            retract(currMax(_)),
            retract(currMaxBoard(_)),
            assert(currMax(Val)),
            assert(currMaxBoard(Root))
        )
        ;true
    ),fail.
choose_move_aux([Player,State,Board], Depth) :-
    (
        Depth > 1,
        moves([Player,State,Board], PosList),!,
        random_shuffle(PosList,[],NewMoves),
        best(NewMoves, Depth)
    );true.

best([],_Depth).
best([Pos | PosList], Depth):-
    NextDepth is Depth - 1,
    choose_move_aux(Pos, NextDepth),
    best(PosList, Depth).
```

Conclusões

No geral, acreditamos que completamos os objetivos do projeto. No entanto, tivemos alguma dificuldade com a adaptação à linguagem Prolog, tanto pela sua sintaxe como pelo seu paradigma, que é largamente diferente de linguagens como C/C++, Java ou JavaScript.

Bibliografia

http://www.neutreeko.net/neutreeko.htm

http://www.di.fc.ul.pt/~jpn/gv/neutreeko.htm

http://www.iggamecenter.com/info/en/neutreeko.html

https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/

Bratko, I. (2011). Prolog Programming for Artificial Intelligence. 4th Edition. Pearson Education Canada, pp.580-588.