U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

›

Faculdade de Engenharia da Universidade do Porto

Programação em Lógica

3º ANO - **MIEIC**

Nudge

Estudantes & Autores

Diogo Silva, up201706892

up201706892@fe.up.pt

Ana Mafalda Santos, up201706791

up201706791@fe.up.pt

15 de **novembro** de **2019**

Nudge_02

Índice

Tabela de Figuras

A. Introdução

No âmbito da unidade curricular de PLOG foi-nos proposta a realização de um jogo de tabuleiro em ***Prolog***.

Depois de alguma análise e deliberação, das distintas opções, o jogo escolhido pelo nosso grupo foi o **Nudge**, por ser um jogo simples que não deixa de ser desafiante e interessante.

O objetivo deste relatório passa por apresentar o jogo e explicitar o modo de implementação do mesmo, assim como a introdução de inteligência artificial às jogadas feitas pelo computador.

B. Nudge: O Jogo

Consiste num jogo para duas pessoas, com dois conjuntos de peças de cores diferentes – **brancas** e **pretas**. Inicialmente, estas estarão dispostas como mostra a **figura 1**.

O jogador que começa é escolhido aleatoriamente. Depois, cada um desta joga alternadamente e pode fazer até dois movimentos (nunca diagonais) com as suas peças.

Estes dois movimentos podem ser feitos sobre a mesma peça, em duas peças diferentes (individualmente) ou ainda num grupo de peças (caso estejam alinhadas) na direção da linha ou coluna que formam em conjunto. Se este conjunto superar em número as peças na sua direção, é permitido "empurrar" (**nudge**) a(s) peças adversárias.

Para ganhar basta "empurrar" com sucesso uma peça adversária para fora do tabuleiro, respeitando as regras estabelecidas (como mostra a **figura 2**).

Figura **1**: Tabuleiro Inicial

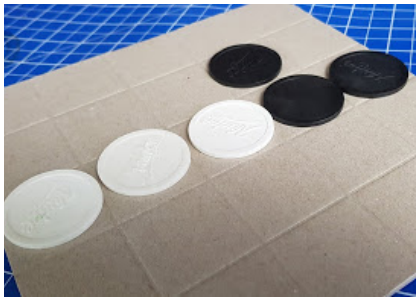

Figura **2**: Tabuleiro Final

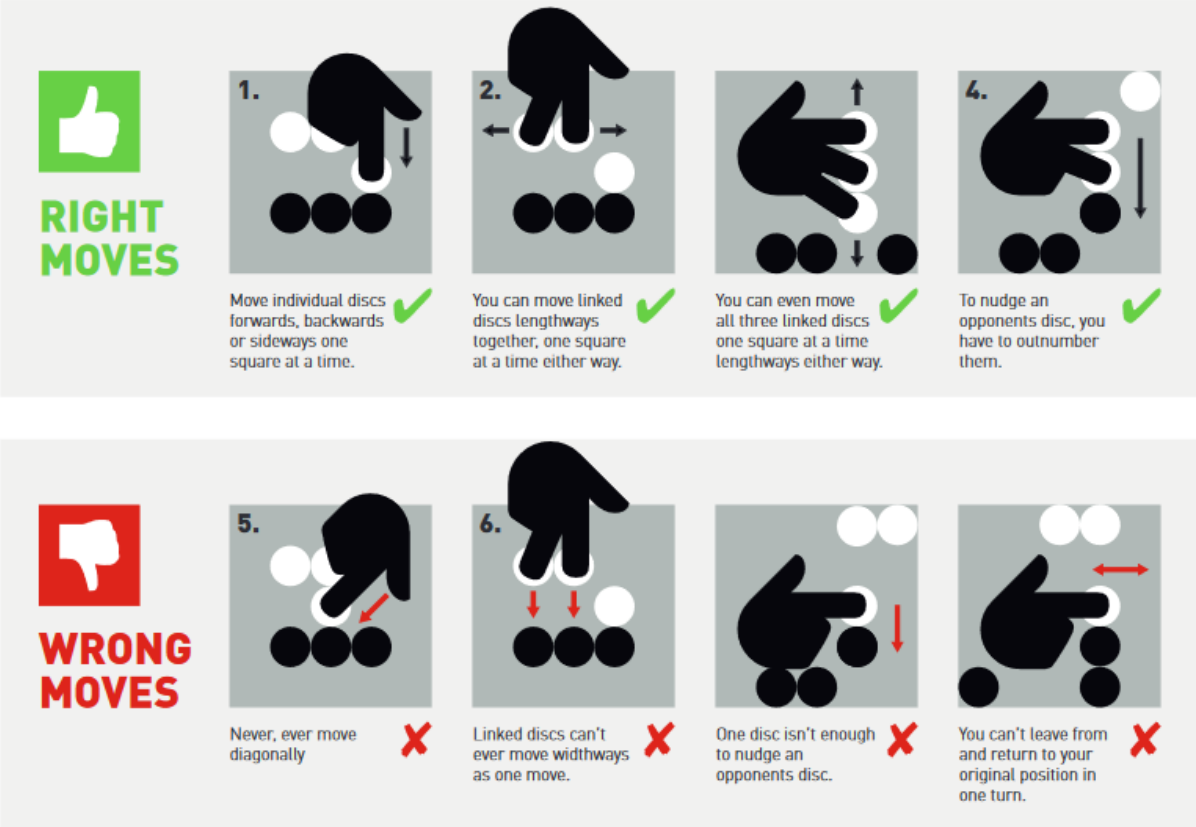

Figura **3**: Algumas jogadas válidas e inválidas

Fonte: https://nudge.greenwich-design.co.uk/

C. Lógica do Jogo

1. Representação Interna do Estado do Jogo

A representação interna do tabuleiro é feita através de uma lista de listas, com tamanho fixo de **5x5**.

As peças brancas são identificadas por "**white**", as pretas por "**black**" e os espaços livres do tabuleiro por uma *string* vazia " ".

Temos ainda três tabuleiros: um que exemplifica o início do jogo, um que mostra um estado intermédio e outro com um possível término deste.

```
board_beg(
[
    ['     ', '     ', '     ', '     ', '     '],
    ['     ', white, white, white, '     '],
    ['     ', '     ', '     ', '     ', '     '],
    ['     ', black, black, black, '     '],
    ['     ', '     ', '     ', '     ', '     ']
]).
```

Figura **4**: Tabuleiro Inicial

```
board_mid(
[
    ['     ', '     ', '     ', '     ', '     '],
    ['     ', white, '     ', black, '     '],
    ['     ', white, white, '     ', '     '],
    ['     ', black, black, '     ', '     '],
    ['     ', '     ', '     ', '     ', '     ']
]).
```

Figura **5**: Tabuleiro Intermédio

```
board_end(
[
    ['     ', '     ', '     ', '     ', '     '],
    ['     ', white, '     ', '     ', '     '],
    ['     ', '     ', '     ', black, white],
    ['     ', black, '     ', '     ', white],
    ['     ', '     ', '     ', '     ', black]
]).
```

Figura **6**: Tabuleiro Final

2. Visualização do Tabuleiro em Modo de Texto

Para a visualização do tabuleiro na consola foi implementado um predicado que, recursivamente, imprime todos os elementos do mesmo.

De modo a permitir a melhor visualização, foram criadas fronteiras para cada célula. As células livres estão vazias, as "**white**" têm uma peça branca e as "**black**" uma peça preta. Por baixo do tabuleiro é indicado o jogador que deverá jogar e a cor das suas peças. As seguintes figuras mostram o tabuleiro em vários estados do jogo:

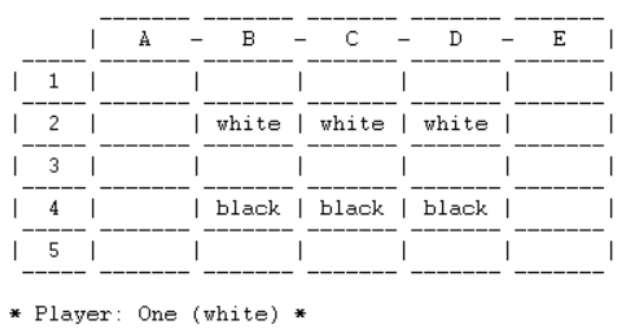

Figura 7: Tabuleiro Inicial

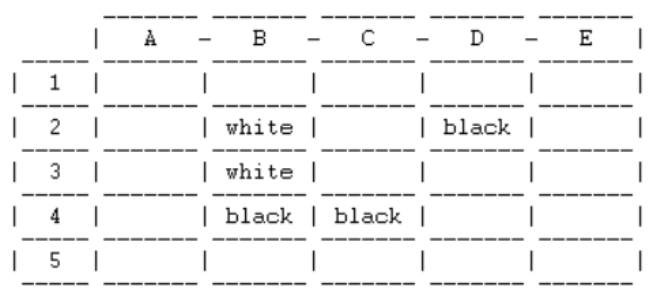

Figura 8: Tabuleiro Intermédio

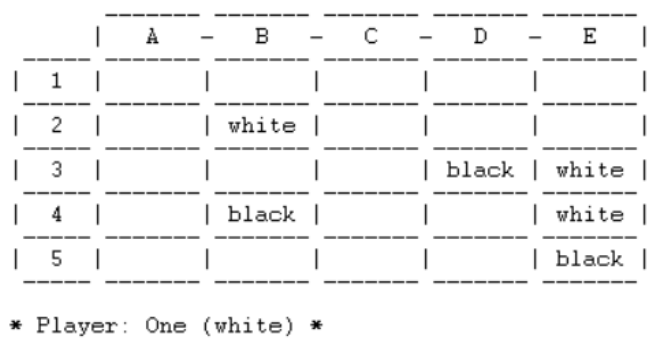

Figura 9: Tabuleiro Final

3. Lista de Jogadas Válidas

Sempre que o utilizador mexe uma peça, as seguintes condições são verificadas para determinar se é possível a sua execução:

- se é uma posição válida (de acordo com os limites do tabuleiro);
- se a posição de destino está vazia;
- se é movimentada apenas uma casa na vertical ou na horizontal (não diagonalmente);
- se na posição de destino se encontra uma peça da mesma cor (para fazer *nudge*);
- se a posição de origem e destino não é a mesma;
- se é possível fazer *nudge* de uma peça da cor contrária;
- se o jogo já terminou.

Assim, em vez de retornarmos uma lista com os possíveis movimentos, testamos se estes são válidos, usando predicados responsáveis por verificar o explicitado acima.

No caso da jogada do computador, criamos uma lista com todas as possibilidades de jogada tendo sempre em conta quais as válidas.

Alguns exemplos dos predicados presentes no ficheiro `validateInpute.pl`, assim como o predicado `valid_moves(…)` presente no `ai.pl`:

```
% verifies if a given move (CRow, CColumn) -> (NRow, NColumn) is valid
checkMove(CRow, CColumn, NRow, NColumn, Color, Adversary, Board, FinalBoard, PreviousBoard, GameStatus) :-
    (
        once(checkSamePosition(CRow, NRow, CColumn, NColumn)),
        once(checkNudge(CRow, CColumn, CRow, CColumn, NRow, NColumn, Color, Adversary, Board, FinalBoard, 0, GameStatus, 1)),
        once(checkDiagonal(CRow, CColumn, NRow, NColumn)),
        once(checkReturnPosition(PreviousBoard, FinalBoard, 1))
        ;
        fail
).
```

Figura **10**: Predicado de validação principal

```
% checks if the player is trying to move to the same place
checkSamePosition(CRow, NRow, CColumn, NColumn) :-
    (
        CRow \= NRow
        ;
        CColumn \= NColumn
        ;
        write('You are already there!\n\n'),
        fail
).
```

Figura **11**: Verifica se a posição inicial é igual á final após a jogada

```
% verifies if a given coordinate is inside the board
checkLimits(Row, Column) :-
  (
    Row > 0,
    Row < 6,
    Column > 0,
    Column < 6
    ;
    fail
).
```

Figura **12**: Verifica os limites do Tabuleiro

```
% checks if a Piece is in that place (Row, Column)
% ------------------------
checkPosition(Row, Column, Piece, Board) :-
    getRow(Row, Column, Piece, Board).

getRow(1, Column, Piece, [Row|_More]) :-
    getColumn(Column, Row, Piece).

getRow(N, Column, Piece, [_Row|Tail]) :-
    N > 1,
    Next is N - 1,
    getRow(Next, Column, Piece, Tail).

% ------------
getColumn(1, [Current|_More], Piece) :-
    (
        Piece == Current
        ;
        fail
).
getColumn(N, [_X|Tail], Piece) :-
    N > 1,
    Next is N - 1,
    getColumn(Next, Tail, Piece).
```

Figura **13**: Verifica se a peça que pretende mover está naquele local

```
% checks if the player is trying to move diagonally
checkDiagonal(CRow, CColumn, NRow, NColumn) :-
    (
        % just moving columns
        abs(CColumn - NColumn) =:= 1,
        CRow - NRow =:= 0
        ;
        % just moving rows
        abs(CRow - NRow) =:= 1,
        CColumn - NColumn =:= 0
        ;
        % moving more than a cell vertically
        abs(CColumn - NColumn) =\= 1,
        CRow - NRow =:= 0,
        write('Can\'t move more than a cell at a time!\n'),
        fail
        ;
        % moving more than a cell horizontally
        abs(CRow - NRow) =\= 1,
        CColumn - NColumn =:= 0,
        write('Can\'t move more than a cell at a time!\n'),
        fail
        ;
        % both movements -> diagonally
        abs(CColumn - NColumn) =:= 1,
        abs(CRow - NRow) =:= 1,
        write('Diagonals are not allowed!\n'),
        fail
).
```

Figura **14**: Verifica se o apenas se moveu uma casa e não diagonalmente

```
% finds all possible moves with two plays
valid_moves([], _, _, _, _, Board, Board) :- !.
valid_moves([Head|Tail], PreviousBoard, Color, Adversary, GameStatus, AllBoards, FinalistBoard) :-
    findall(NewFinalBoard, moveAI(PreviousBoard, Head, NewFinalBoard, Color, Adversary, GameStatus), NewAllBoards),
    join_lists(AllBoards, NewAllBoards, FinalAllBoards),
    valid_moves(Tail, PreviousBoard, Color, Adversary, GameStatus, FinalAllBoards, FinalistBoard).
```

Figura **15**: Predicado que trata da lista de jogadas válidas

4. Execução de Jogadas

Atendendo que cada jogador pode mover peças duas vezes, no *game loop* é, para cada jogador, chamado o predicado `whiteTurn(…)` ou `blackTurn(…)`, dependendo do jogador. A única função destas é chamar o predicado `move(…)` (**Figura 16**) com os argumentos respetivos. Aqui são pedidas as coordenadas do local anterior da peça e do novo, sendo estas posteriormente validadas pelo predicado `checkPosition(…)` e `checkMove(…)`, respetivamente.

Após o movimento ser aceite, é chamado o predicado `setPiece(…)` que recebendo o tabuleiro inicial (TabIn), a peça (Piece) e a posição da mesma (Row, Column), constrói o tabuleiro final (TabOut). Este é chamado no predicado `checkNudge(…)` que, por sua vez, é chamado no predicado `checkMove(…)`.

```
% standard white piece turn
whiteTurn(PreviousBoard, Board, FinalBoard, Player, GameStatus, DisplayColor) :-
    once(move(PreviousBoard, Board, FinalBoard, white, black, Player, GameStatus, DisplayColor)).

% standard black piece turn
blackTurn(PreviousBoard, Board, FinalBoard, Player, GameStatus, DisplayColor) :-
    once(move(PreviousBoard, Board, FinalBoard, black, white, Player, GameStatus, DisplayColor)).
```

Figura **16**: Predicados do jogador

```
% reads coordinates and if everything succeeds, the move is made
move(PreviousBoard, Board, FinalBoard, Color, Adversary, Player, GameStatus, DisplayColor) :-
    repeat,
    (
        readCoordinates(CRow, CColumn, 'Current'),
        checkPosition(CRow, CColumn, Color, Board),
        readCoordinates(NRow, NColumn, 'New'),
        checkMove(CRow, CColumn, NRow, NColumn, Color, Adversary, Board, FinalBoard, PreviousBoard, GameStatus),
        display_game(FinalBoard, Player, DisplayColor)
        ;
        write('* Try again! *\n\n'),
        fail
).
```

Figura **17**: Predicado encarregado de mover as peças

Quando a jogada é feita pelo computador, é chamado o predicado `aiTurn(…)` que trata de escolher entre os tabuleiros possíveis e o nível escolhido, a jogada mais indicada.

```
% ai's double turn
aiTurn(PreviousBoard, Board, _, Player2, Color, Adversary, GameStatus, BoardAI, GameLevel) :-
    % finds all possible plays with just one movement
    once(findall(FinalBoard, moveAI(PreviousBoard, Board, FinalBoard, Color, Adversary, GameStatus), AllBoards1)),
    % if there's any chance to win, those boards are put on NewWin1
    once(value(AllBoards1, _, NewWin1, _, _, Adversary)),
    length(NewWin1, L),
    (
        % if on hard level, it makes a win move if possible
        GameLevel == 2,
        L \== 0,
        random_member(BoardAI, NewWin1)
        ;
        % if on easy level or no winning moves found with just one movement, it finds all possible plays with a second movement
        once(valid_moves(AllBoards1, PreviousBoard, Color, Adversary, GameStatus, _, AllBoards)),
        once(value(AllBoards, _, NewWin, _, NewOther, Adversary)),
        % according to the gameLevel it either chooses randomnly or a winning movement
        once(getBoard(NewWin, NewOther, BoardAI, GameLevel))
    ),
```

Figura **18**: Predicado que executa a jogada do computador

5. Final do Jogo

O jogo termina quando um dos jogadores consegue, na mesma fila, horizontal ou vertical e com peças consecutivas e não alternadas, ter um número maior de peças da sua cor do que da do

adversário, que lhe permite "empurrar" (*nudge*) uma dessas peças para fora do tabuleiro.

Sempre que um dos jogadores efetua um *nudge* vencedor, o predicado `checkNudge(…)` (responsável por validar a respetiva jogada) muda o argumento GameStatus para o valor 1. Por sua vez, após cada jogada, o predicado `game_over(…)` trata de verificar se GameStatus foi alterado, significando que existe um vencedor e o jogo deve ser terminado, procedendo assim ao seu término e à indicação do jogador que venceu.

```
% checks whether the game is over (GameStatus \== 1) and prints the winner (Winner)
game_over(GameStatus, Winner) :-
    (
        GameStatus \== 1
        ;
        Winner == '1',
        playerOneWins
        ;
        Winner == '2',
        playerTwoWins
).
```

Figura **19**: Predicado usado para terminar o jogo

6. Avaliação do Tabuleiro

Para determinar qual das possíveis jogadas o computador deve fazer, usamos o predicado `value(…)`.

Como o computador pode efetuar até duas jogadas, são calculadas todas as combinações de jogadas possíveis que este pode fazer. Por sua vez, o predicado supracitado, recebe a lista com todos os tabuleiros e, dependendo do tabuleiro que recebe, está encarregado de verificar se o computador pode vencer o jogo. Se esse for o caso, este coloca na lista NewWin todas os tabuleiros vencedores, sendo que todos os outros serão colocados numa outra lista – NewOther.

Para fazer esta verificação, todos os tabuleiros são percorridos e o predicado `gameOverAI(…)` vê se em algum destes estão menos de três peças do seu adversário presentes, ou seja, uma peça foi arrastada para fora do tabuleiro e o jogo acabou.

```
% winning moves -> NewWin, other moves -> NewOther
value([], Board, Board, Board1, Board1, _) :- !.
value([Head|Tail], Win, NewWin, Other, NewOther, Adversary) :-
    (
        gameOverAI(Head, Adversary, 0),
        join_lists(Other, [Head], Other2),
        join_lists(Win, [], Win2)
        ;
        join_lists(Win, [Head], Win2),
        join_lists(Other, [], Other2)
    ),
    value(Tail, Win2, NewWin, Other2, NewOther, Adversary).
```

Figura **20**: Predicado usado para avaliar os melhores tabuleiros

7. Jogada do Computador

Dependendo do nível (*Easy* ou *Hard*), na função `choose_move(…)` vão ocorrer duas escolhas distintas.

Este predicado recebe duas listas: uma com os tabuleiros vencedores e outra com as restantes. Caso o nível seja "fácil", estas listas são concatenadas e o tabuleiro com a jogada final será escolhido aleatoriamente. Caso contrário, no nível mais "difícil", é escolhido um tabuleiro da lista de tabuleiros vencedores (gerados pelo predicado `value(…)`) aleatoriamente, ou, se este estiver vazio, um aleatório da lista NewOther.

```
% randomly selects one of the possible boards
choose_move(Win, Other, BoardAI, GameLevel) :-
  length(Win, X),
  (
    % easy level - random board chosen
    GameLevel == 1,
    join_lists(Win, Other, Boards),
    random_select(BoardAI, Boards, _)
    ;
    % hard level - random winning board chosen and if there are none, another random board is chosen
    (
      X \== 0,
      random_select(BoardAI, Win, _)
      ;
      random_select(BoardAI, Other, _)
    )
).
```

Figura **21**: Escolha do tabuleiro com a jogada final

D. Conclusões

Tendo este trabalho como principal objetivo aplicar a matéria lecionada nas aulas práticas e teóricas da cadeira, considero que apesar de termos encontrado algumas dificuldades iniciais na implementação da inteligência artificial nas jogadas executadas pelo computador, este foi concluído com sucesso, tendo superado as nossas expectativas.

Consideramos que conseguimos aplicar tudo o que nos era pedido, inclusive inteligência artificial capaz de antever até duas jogadas.

Em suma, com este trabalho conseguimos não só aprimorar o nosso conhecimento da linguagem ***Prolog***, mas também a nossa adaptabilidade a novos conceitos e a novas linguagens.