Inteligência Artificial – Programação em Lógica 2º Semestre 2021/22

A resolução deste projeto deve ser feita em **grupos de dois alunos** e tem de ser realizada em **Prolog**. O projeto deve ser desenvolvido num ficheiro .pl em que o nome deve ser os números de aluno seguido dos respetivos primeiro e último nome (exemplo: "12345_JoaoSilva_54321_MariaSantos.pl"). Para entregarem devem comprimir este ficheiro usando o formato de compactação **ZIP** (não usar outro formato) e o nome do ficheiro comprimido deve ser por exemplo: "12345_JoaoSilva_54321_MariaSantos.zip".

A entrega é realizada através do e-learning até **23:59 de 24/04/2022**.

Quatro em linha

Pretende-se implementar uma versão alterada do jogo quatro em linha. Nesta versão, dois jogadores jogam num tabuleiro (N x N) e podem colocar alternadamente as peças numa das posições vagas do tabuleiro. Ganha o primeiro jogador que colocar quatro peças consecutivas numa linha, numa coluna ou numa diagonal. É considerado um empate se já não existirem posições livres e nenhum dos jogadores conseguir colocar quatro peças em linha.
No jogo a implementar, o jogador joga contra um agente autónomo que usa o algoritmo do minimax com cortes alfa-beta para decidir as suas jogadas.

Para iniciar o jogo, deve usar-se o predicado **play_game**.

```
play_game():-
    initial_board(6, 6, B0),
    play(1,B0).
```

Este predicado começa por criar um tabuleiro com a dimensão 6x6 e validar a jogada do computador (player 1). O predicado **play** tem como argumentos o jogador que deve jogar e o tabuleiro sobre o qual deve fazer a sua jogada. Este predicado é recursivo de modo a permitir a alternância das jogadas. A sua implementação é constituída pela jogada do computador (play(1,B)) , pela jogada do jogador (play(2,B)) e pela verificação do fim do jogo.

```
play(_,B0):- %GAME OVER
    game_over(B0,T),
    calculate_board_value(T,B0,Value),
    print_board(B0),
    write('Game Over!'),
    nl,
    display(Value),
    !.


play(1,B0):- %AI COMPUTER
    write('Player:'),nl,
    print_board(B0),
    alphabeta(B0,6,-100,100,B1,_,1),
    !,
    play(2,B1).
```

```
play(2,B0):- %USER
    write('Computer:'),nl,
    print_board(B0),
    write('Where to play? (C,L)'),
    read(C),
    read(L),
    valid_move(C,L,B0),
    add_move(2,C,L,B0,B1),
    !,
    play(1,B1).
```

Quando é a vez do computador jogar, é usado o algoritmo do minimax com alfa-beta. O minimax pode ser implementado recursivamente, testando em cada passo todas as possibilidades de jogada de cada jogador. Quando gera as possíveis jogadas do computador, o algoritmo vai escolher a que maximiza as possibilidades do computador vencer (maximização) e quando gera as possíveis jogadas do utilizador vai escolher as que prejudicam mais o computador (minimização).

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
 if depth = 0 or node is a terminal node then
       return the heuristic value of node
 if maximizingPlayer then
       value := −∞
       for each child of node do
             value := max(value,
              alphabeta(child, depth − 1, α, β, FALSE))
             if value ≥ β then
                   break (* β cutoff *)
             α := max(α, value)
       return value
 else
       value := +∞
       for each child of node do
             value := min(value,
                   alphabeta(child, depth − 1, α, β, TRUE))
             if value ≤ α then
                   break (* α cutoff *)
             β := min(β, value)
       return value
```

O predicado que implementa o minimax é chamado alfabeta e tem como argumentos o tabuleiro, o valor de profundidade que ainda é permitido explorar, o alfa, o beta, o tabuleiro resultado, o score da avaliação do resultado na ótica do computador e o jogador que se está a avaliar (minimizar ou a maximizar).

```
alphabeta(Bi, 0, _, _, Bi, Value, P):-
    calculate_board_heuristic(P,Bi,Value),
    !.

alphabeta(Bi, _, _, _, Bi, Value, _):-
    game_over(Bi,T),
    calculate_board_value(T,Bi,Value),
    !.

alphabeta(Bi, D, Alfa, Beta, Bf, Value, Player):-
    possible_moves(Player,Bi,L),
    !,
    next_player(Player,Other),
    evaluate_child(Other, L, D, Alfa, Beta, Bf, Value).
```

O predicado minimax usa o predicado **evaluate_child** para recursivamente avaliar todas as possíveis jogadas de cada jogador.

Usando os predicados fornecidos no ficheiro quatro_em_linha.pl **implemente os predicados necessários para terminar o jogo**. Estes encontram-se listados abaixo, com exemplos do seu funcionamento.

O tabuleiro é representado como uma lista de listas de inteiros. Deve usar o 0 (zero) para representar uma posição vazia, 1 para as peças do computador e 2 para as peças do jogador.

Exemplo de um tabuleiro 6x6 vazio:
[[0,0,0,0,0,0],[0,0,0,0,0,0],[0,0,0,0,0,0],[0,0,0,0,0,0],[0,0,0,0,0,0],[0,0,0,0,0,0]]

```
    0 1 2 3 4 5
  ┌─────────────
0 │ 0 0 0 0 0 0
1 │ 0 0 0 0 0 0
2 │ 0 0 0 0 0 0
3 │ 0 0 0 0 0 0
4 │ 0 0 0 0 0 0
5 │ 0 0 0 0 0 0
```

Exemplo de um tabuleiro 5x5 com uma peça do computador (player 1) na posição (1,2) e uma peça do jogador (player 2) na posição (3,4):
[[0,0,0,0,0],[0,0,**1**,0,0],[0,0,0,0,0],[0,0,0,0,**2**],[0,0,0,0,0]]

```
    0 1 2 3 4
  ┌───────────
0 │ 0 0 0 0 0
1 │ 0 0 1 0 0
2 │ 0 0 0 0 0
3 │ 0 0 0 0 2
4 │ 0 0 0 0 0
```

Os perdicados *play* e o *minimax* dependem dos predicados abaixo. Implemente-os no ficheiro prolog fornecido (.pl):

a) **initial_board(NumRows,NumColumns,Board)** - Board é um tabuleiro vazio com NumRows linhas por NumColumns colunas. O tabuleiro é representado por uma lista de listas de 0.

```
?- initial_board(4, 4, B).
B = [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]].
```

b) **print_board(Board)** – Imprime para a consola o tabuleiro Board.

```
?- print_board([[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]).
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
_________________

true.
```

Nota: para imprimir valores para a consola utilize o predicado *write/1*, se pretender fazer um parágrafo utilize o *nl/0*, como mostra o seguinte exemplo:

```
write_two_names(Name1,Name2):-
       write(Name1), nl, write(Name2).

?- write_two_names("Hello","World").
Hello
World
true.
```

c) **valid_move(X,Y,Board)** - é verdade se a posição (X,Y) do tabuleiro Board estiver vazia (existir um 0).

```
?- valid_move(2,2,[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 1, 0], [0, 0, 0, 0]]).
false.

?- valid_move(2,3,[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 1, 0], [0, 0, 0, 0]]).
true.
```

d) **add_move(Player,X,Y,InitialBoard,FinalBoard)** - O FinalBoard deve ter de ser igual ao InitialBoard com mais uma peça do Player na posição (X,Y)

```
?- add_move(2,1,1,[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 1, 0], [0, 0, 0, 0]],B).
B = [[0, 0, 0, 0], [0, 2, 0, 0], [0, 0, 1, 0], [0, 0, 0, 0]]
true.

?- add_move(2,0,1,[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 1, 0], [0, 0, 0, 0]],B).
B = [[0, 2, 0, 0], [0, 0, 0, 0], [0, 0, 1, 0], [0, 0, 0, 0]]
True.
```

e) **generate_move(Player,InitialBoard,FinalBoard)** - O FinalBoard deve ter de ser igual ao InitialBoard com mais uma peça do jogador Player numa das posições livres do InitialBoard.

```
?- generate_move(2,[[1, 0, 1, 0],[0, 1, 1, 0],[0, 0, 1, 1],[1, 0, 0, 1]],BP).
BP = [[1, 2, 1, 0], [0, 1, 1, 0], [0, 0, 1, 1], [1, 0, 0, 1]] ;
BP = [[1, 0, 1, 2], [0, 1, 1, 0], [0, 0, 1, 1], [1, 0, 0, 1]] ;
BP = [[1, 0, 1, 0], [2, 1, 1, 0], [0, 0, 1, 1], [1, 0, 0, 1]] ;
BP = [[1, 0, 1, 0], [0, 1, 1, 2], [0, 0, 1, 1], [1, 0, 0, 1]] ;
BP = [[1, 0, 1, 0], [0, 1, 1, 0], [2, 0, 1, 1], [1, 0, 0, 1]] ;
BP = [[1, 0, 1, 0], [0, 1, 1, 0], [0, 2, 1, 1], [1, 0, 0, 1]] ;
```

```
BP = [[1, 0, 1, 0], [0, 1, 1, 0], [0, 0, 1, 1], [1, 2, 0, 1]] ;
BP = [[1, 0, 1, 0], [0, 1, 1, 0], [0, 0, 1, 1], [1, 0, 2, 1]] ;
false.
```

f) **game_over(Board,Winner)** - verifica se o jogo terminou e indica o vencedor (Winner: 0- empate, 1 - computador, 2 - jogador). Este predicado deve verificar as diferentes possibilidades de um jogador vencer, bem como verificar se já não existem posições livres no tabuleiro. Seguem-se exemplos de vários possíveis desfechos do jogo.

Computador vence com 4 na diagonal:

```
?- game_over([[1, 0, 1, 0], [0, 1, 1, 0], [0, 0, 1, 1], [1, 0, 0, 1]],W).
W = 1.
```

Computador vence com 4 numa linha:

```
?- game_over([[1, 0, 1, 0], [1, 1, 1, 1], [0, 0, 0, 1], [1, 0, 0, 1]],W).
W = 1.
```

Jogador vence com 4 numa coluna:

```
?- game_over([[1, 0, 2, 0], [1, 1, 2, 1], [0, 0, 2, 1], [1, 0, 2, 1]],W).
W = 2.
```

Empate, pois, o tabuleiro já não tem posições livre:

```
?- game_over([[1, 1, 2, 1], [2, 1, 1, 1], [1, 2, 2, 1], [1, 1, 2, 2]],W).
W = 0
```

O jogo ainda não terminou:

```
?- 0)
false.
```

g) **calculate_board_value(Winner, Score)** - permite saber qual o valor do tabuleiro na perspectiva do computador sabendo quem foi o vencedor (Winner). Por exemplo: -1 se o vencedor for o jogador, 1 se o vencedor foi o computador e 0 se empataram.

```
?- calculate_board_value(1,Value) .
Value = 1.

?- calculate_board_value(2,Value) .
Value = -1.

?- calculate_board_Value(0,Value) .
Value = 0.
```

h) **calculate_board_heuristic(Player,Board,Value)** - permite saber o valor de um tabuleiro não final através de uma heurística. Esta heurística deve avaliar o estado do tabuleiro de modo a valorizar tabuleiros onde o computador tem uma maior hipótese de vencer. Um exemplo de uma heurística seria uma função que valorizaria tabuleiros que tivessem duas ou três peças seguidas, estando assim mais próximas de uma vitória. Se não se conseguir definir uma heurística deve atribuir-se um valor neutro, por exemplo: Value = 0.

Consoante os valores usados na heurística pode fazer sentido ajustar os valores devolvidos no predicado calculate_board_value.

```
calculate_board_heuristic(P,Bi,Value)
```