



Faculdade de Engenharia da Universidade do  
Porto

# Barca

## Trabalho Prático 2

**Programação Funcional e em Lógica**  
**Turma 5 - Barca 5**

Fábio Araújo de Sá ([up202007658@fe.up.pt](mailto:up202007658@fe.up.pt)).  
Lourenço Alexandre Correia Gonçalves ([up202004816@fe.up.pt](mailto:up202004816@fe.up.pt)).

Contribuição: 50%  
Contribuição: 50%

Porto, 30 de Dezembro de 2022

# Índice

<b>Instalação e Execução</b>	<b>3</b>
<b>Descrição do Jogo</b>	<b>3</b>
<b>Lógica do Jogo</b>	<b>4</b>
Representação interna do estado do jogo	4
Estado do jogo inicial	5
Estado do jogo intermédio	5
Estado do jogo final	6
Visualização do estado do jogo	7
Execução das jogadas	10
Lista de jogadas válidas	11
Final do jogo	12
Avaliação do tabuleiro	12
Jogada do computador	13
<b>Conclusões</b>	<b>15</b>
<b>Bibliografia</b>	<b>15</b>

# Instalação e Execução

Para instalar o jogo Barca primeiro é necessário fazer o *download* dos ficheiros presentes em PFL\_TP2\_T05\_Barca5.zip e descompactá-los. Dentro do diretório *src* consulte o ficheiro *main.pl* através da linha de comandos ou pela própria UI do Sicstus Prolog 4.7.1. O jogo está disponível em ambientes Windows e Linux.

O jogo inicia-se com o predicado *play/0*:

---

```
? - play.
```

---

## Descrição do Jogo

Barca é um jogo de tabuleiros para dois jogadores. Na sua versão tradicional cada jogador possui dois animais de cada tipo (rato, leão, elefante) e o tabuleiro 10x10 contém quatro casas centrais: as águas. Ganha quem colocar três dos seus animais nas águas. Existem algumas restrições de movimento:

1. Os ratos só se movem ortogonalmente, os leões diagonalmente e o elefante em ambas direções;
2. Os ratos têm medo dos leões, os leões têm medo dos elefantes e os elefantes dos ratos;
3. Um animal não se pode mover para perto (célula adjacente) de um animal do qual ele tenha medo, se este pertencer ao oponente;
4. Quando um animal está com medo, o jogador é forçado a movê-lo. Nesta situação, o movimento pode ignorar a regra 3;
5. Um animal não pode mover-se por cima de outro;

Em testes realizados, embora muito raramente, reparou-se que é possível barrar todos os movimentos de animais com medo do jogador adversário. Assim, nessa situação, para que haja progressão do jogo, é também admitido um movimento com um animal que não tenha medo.

A implementação em Prolog que se segue rege-se através destas regras. No entanto também existem modos com diferentes tamanhos de tabuleiro, o que provoca um aumento do número de animais, de águas disponíveis e uma dificuldade acrescida.

As regras e funcionamento do jogo foram consultadas de dois sites: [BoardGameGeek](#) e [Wikipédia](#).

# Lógica do Jogo

## Representação interna do estado do jogo

O estado do jogo, *GameState*, é um argumento essencial de todos os predicados principais da implementação. É formado por uma lista de quatro elementos:

- **Board**, uma matriz quadrada de tamanho indicado pelo utilizador no momento da configuração. Contém átomos a representar animais (*elephant1*, *mouse2*...) nas suas posições iniciais e átomos *empty* em todas as posições não ocupadas por estes;
- **Player**, o jogador que irá iniciar a partida (átomo *player1* ou *player2*);
- **FearList/ForcedMoves**, contém as coordenadas (pares coluna-linha) de todos os animais com medo que pertencem ao Player atual. É importante para a aplicação da quarta regra do tópico anterior em *runtime*;
- **TotalMoves**, acumulador do número total de movimentos durante o jogo. Com esse valor é possível calcular o número de jogadas do vencedor e verificar a assertividade do Bot Greedy contra outro jogador;

Note-se que na representação do tabuleiro não estão presentes as águas por motivos de eficiência temporal. São de facto as únicas peças com localização contante em toda a duração do jogo e delas dependem diretamente operações muito frequentes: os valores associados à avaliação de cada tabuleiro (no caso do *Bot Greedy*) e a avaliação do término do jogo em cada ciclo, com o predicado **game\_over(+GameState, -Winner)**.

Encontrar as localizações das águas inseridas em Board seria na ordem de  $O(N^2)$ , com **N** diretamente proporcional ao tamanho do tabuleiro. Tratando as águas como factos inseridos apenas no momento da escolha do tamanho do tabuleiro, a operação é apenas de ordem  $O(C)$ , com **C** constante e equivalente ao número total de águas no jogo que é sempre muito inferior a qualquer **N** permitido. De forma semelhante, a quantidade de águas necessárias para ganhar o jogo é inserida no facto **n\_waters\_to\_win/1** para ser acedida facilmente. No fundo é uma pequena mudança que traz um considerável ganho na *performance*.

Aqui é possível observar a operação em *runtime* da atribuição das coordenadas a águas quando o tabuleiro escolhido é 10x10:

---

```
% water(+Coordinate)
:- dynamic water/1.

% fill_water(+BoardSize)
fill_water(10):-
    asserta((water(4-4))),
    asserta((water(4-7))),
    asserta((water(7-4))),
    asserta((water(7-7))),
    asserta((n_waters_to_win(3))), !.
```

---

board.pl

## Estado do jogo inicial

```
GameState([[[empty, empty, empty, empty, lion1, lion1, empty, empty, empty, empty],
[empty, empty, empty, elephant1, mouse1, mouse1, elephant1, empty, empty, empty],
[empty, empty, empty, empty, empty, empty, empty, empty, empty, empty],
[empty, empty, empty, empty, empty, empty, empty, empty, empty, empty],
[empty, empty, empty, empty, empty, empty, empty, empty, empty, empty],
[empty, empty, empty, empty, empty, empty, empty, empty, empty, empty],
[empty, empty, empty, empty, empty, empty, empty, empty, empty, empty],
[empty, empty, empty, empty, elephant2, mouse2, elephant2, empty, empty, empty],
[empty, empty, empty, empty, empty, lion2, lion2, empty, empty, empty]], %Board
player1, % Player
[], % FearList
0 % TotalMoves
```

	1	2	3	4	5	6	7	8	9	10
1					L	L				
2				E	M	M	E			
3										
4				#			#			
5										
6										
7				#			#			
8										
9				e	m	m	e			
10					l	l				

## Estado do jogo intermédio

```
GameState([[[empty, empty, empty, empty, lion1, lion1, empty, empty, empty, empty],
[empty, empty, empty, empty, mouse1, mouse1, elephant1, empty, empty, empty],
[empty, empty, empty, empty, empty, empty, empty, empty, empty, empty],
[empty, empty, empty, elephant1, empty, empty, empty, empty, empty, empty],
[empty, empty, empty, empty, mouse2, empty, empty, empty, empty, empty],
[empty, empty, empty, empty, empty, empty, empty, empty, empty, empty],
[empty, empty, empty, empty, empty, empty, empty, empty, empty, empty],
[empty, empty, empty, empty, empty, empty, empty, empty, empty, empty],
[empty, empty, empty, elephant2, empty, mouse2, elephant2, empty, empty, empty],
[empty, empty, empty, empty, lion2, lion2, empty, empty, empty, empty]], %Board
player1, % Player
[4-4], % FearList
2 % TotalMoves
```

	1	2	3	4	5	6	7	8	9	10
1					L	L				
2					M	M	E			
3										
4				E			#			
5					m					
6										
7				#			#			
8										
9				e		m	e			
10					l	l				

## Estado do jogo final

```
GameState([[[empty, empty, empty, empty, lion1, lion1, empty, empty, empty, empty],
[empty, empty, empty, elephant1, empty, mouse1, elephant1, empty, empty, empty],
[empty, empty, empty, empty, empty, empty, empty, empty, empty, empty],
[empty, empty, empty, mouse1, empty, empty, elephant1, empty, empty, empty],
[empty, empty, empty, empty, mouse2, empty, empty, empty, empty, empty],
[empty, empty, empty, empty, empty, empty, empty, empty, empty, empty],
[empty, empty, empty, empty, empty, empty, empty, empty, empty, empty],
[empty, elephant2, empty, mouse2, empty, empty, elephant1, empty, empty, empty],
[empty, empty, empty, empty, empty, empty, empty, empty, empty, empty],
[empty, empty, empty, empty, lion2, lion2, empty, empty, empty, empty]], %Board
[], % FearList
11 % TotalMoves
```

	1	2	3	4	5	6	7	8	9	10
1					L	L				
2						M				
3										
4				M			E			
5					m					
6										
7		e		m			E			
8								e		
9										
10					l	l				

## Visualização do estado do jogo

Antes de iniciar o jogo, o(s) utilizador(s) são convidados a configurar a partida. Os parâmetros são:

A - Modo (Humano / Humano, Humano / Computador, Computador / Computador);

B - Nome dos jogadores;

C - Nível de dificuldade do Bot;

D - Qual o jogador que inicia a partida;

E - Tamanho do tabuleiro;

Em qualquer caso a validação de input e todas as combinações de jogadores possíveis são asseguradas. Uma possível interação seria a seguinte:

---

```
Please select game mode:
1 - Human vs. Human
2 - Human vs. Bot
3 - Bot vs. Bot
Mode between 1 and 3: 2          A

Human vs. Bot
Hello player1, what is your name? User B

Please select player2 status:
1 - Random
2 - Greedy
Difficulty between 1 and 2: 2    C

Who starts playing?
1 - User with UPPERCASE animals
2 - bot with lowercase animals
Select between 1 and 2: 1       D

Board size: 10, 13 or 16? 10    E
```

---

configuration.pl

A validação dos pontos A e C é garantida com o predicado genérico **get\_option/4**, moldável através do contexto e que também é utilizado para validar as coordenadas dos animais a movimentar e parcialmente o input E:

---

```
% get_option(+Min,+Max,+Context,-Value)
get_option(Min,Max,Context,Value):-
    format('~a between ~d and ~d: ', [Context, Min, Max]),
    repeat,
    read_number(Value),
    between(Min, Max, Value), !.
```

---

utils.pl

No caso concreto de jogadores Bot, a dificuldade, **difficulty/2**, é colocada dinamicamente na base de factos para poder ser acedida em todos os predicados:

---

```
% choose_difficulty(+Bot)
choose_difficulty(Bot) :-
    format('Please select ~a status:\n', [Bot]),
    write('1 - Random\n'),
    write('2 - Greedy\n'),
    get_option(1, 2, 'Difficulty', Option), !,
    asserta((difficulty(Bot, Option))).
```

---

configurations.pl

Por outro lado, em B utilizou-se uma implementação mais *user-friendly* do `read/1`, admitindo caracteres até ao *endline*. O nome de cada jogador também é colocado dinamicamente na base de factos através do predicado **name\_of/2**:

---

```
% get_line(-Result, +Acc):-
get_line(Result, Acc):-
    get_char(Char),
    Char \= '\n',
    append(Acc, [Char], Acc1),
    get_line(Result, Acc1).
get_line(Result, Acc):-
    atom_chars(Result, Acc).

% get_name(+Player)
get_name(Player):-
    format('Hello ~a, what is your name? ', [Player]),
    get_line(Name, []),
    asserta(name_of(Player, Name)).
```

---

configurations.pl

Após a escolha do tamanho do tabuleiro em E o próprio é inicializado com **init\_state/2**. Por motivos de simetria admitem-se valores fixos para o tamanho, variando de 10x10 (o modo tradicional) a 16x16 (com mais animais e águas, logo mais complexo).

---

```
% configurations(-GameState)
configurations([Board,Player,[],0]):-
    barca,
    set_mode,
    init_random_state,
    choose_player(Player),
    choose_board(Size),
    init_state(Size, Board).
```

---

configurations.pl



Com o GameState inicializado completa-se a parte das configurações. Em seguida o tabuleiro de jogo é mostrado. Em cada interação, essa ação é desencadeada pelo predicado **display\_game/1**:

---

```
% display_game(+GameState)
display_game([Board,_,_,_]) :-
    clear_console,                % Limpa dados visíveis na consola
    length(Board, Size),          % Tamanho do Tabuleiro a apresentar
    display_header(1, Size),      % 1  2  3  4  5  6 ... Size
    display_bar(Size),            % |---|---|---| ...
    display_rows(Board, 1, Size). % | E | L | m | ...
```

---

main.pl

O predicado **display\_rows/3** é responsável por imprimir os animais nas coordenadas corretas. Para generalizar as relações, considera-se animal do jogador playerX todos os animaisX (lion1 pertence ao player1, lion2 ao player2). Foi necessário elaborar os predicados **piece\_info/3**, **fears/2** e **symbol/2**, que traduzem o contexto do estado do jogo sem dependências do número de animais que cada jogador possui:

---

```
% piece_info(-Type,?Player,+Animal)
piece_info(lion, player1, lion1).
piece_info(elephant, player2, elephant2). %...

% fears(+Type1,+Type2)
fears(lion,elephant).
fears(mouse,lion).
fears(elephant,mouse).

% symbol(+Animal,-Symbol)
symbol(lion1, 'L') :- !.
symbol(lion2, 'l') :- !.
symbol(elephant1, 'E') :- !. %...
```

---

data.pl

## Execução das jogadas

O jogo funciona com base num ciclo cujo único caso de paragem é a vitória de um dos jogadores:

---

```
game_cycle(GameState):-
    game_over(GameState, Winner), !, % caso de paragem, game over
    display_game(GameState),          % mostra o tabuleiro final
    show_winner(GameState, Winner).    % anuncia o vencedor

game_cycle(GameState):-                % loop principal
    display_game(GameState),           % mostra o estado do tabuleiro
    choose_move(GameState, Move),      % escolhe um movimento válido
    move(GameState, Move, NewGameState), !, % Move. Novo GameState
    game_cycle(NewGameState).
```

---

main.pl

Em **choose\_move/2** o jogador é convidado a inserir duas coordenadas, que correspondem à posição inicial e final de um animal no tabuleiro. As coordenadas resultantes são avaliadas através do predicado **validate\_move/3**:

---

```
validate_move(GameState, ColI-RowI, ColF-RowF) :-
    [Board, Player, FearList, _] = GameState,
    in_bounds(Board, ColI-RowI), in_bounds(Board, ColF-RowF),           I
    (member(ColI-RowI, FearList); length(FearList, 0)),                II
    position(Board, ColI-RowI, PieceI),
    position(Board, ColF-RowF, PieceF),
    \+(piece_info(PieceI, neutral)), piece_info(PieceF, neutral),      III
    piece_info(PieceType, Player, PieceI),                             IV
    valid_direction(PieceType, ColI-RowI, ColF-RowF),                   V
    \+path_obstructed(Board, ColI-RowI, ColF-RowF),                     VI
    (\+fears_close(Board, PieceI, ColF-RowF);                          VII
    member(ColI-RowI, FearList)).
```

---

main.pl

Considera-se um movimento válido quando:

- I. As coordenadas estão dentro dos limites do tabuleiro escolhido;
- II. A coordenada inicial seja de um animal com medo (membro da lista *FearList*) ou de qualquer outro animal caso nenhum esteja com medo naquele momento;
- III. A coordenada inicial corresponda a um animal e a coordenada final esteja livre (*empty* ou *water*);
- IV. A coordenada inicial corresponda a um animal de *Player*;
- V. A translação da coordenada inicial para a coordenada final corresponda a um movimento numa direção válida para aquele tipo de peça;

- VI. Durante o movimento, o caminho percorrido não está obstruído por outro animal;
- VII. A coordenada final corresponde a uma zona livre de peças que possam causar medo ao animal movido ou a coordenada final corresponde a um qualquer local caso o animal movido já esteja com medo (membro da lista FearList);

Após selecionar corretamente o movimento, há troca de peças no tabuleiro. O predicado **move/3** também atualiza o valor de GameState para estar apto à próxima jogada:

---

```
% move(+GameState, +Move, -NewGameState)
move(GameState, ColI-RowI-ColF-RowF, NewGameState):-
    [Board,Player,_,TotalMoves] = GameState,
    position(Board,ColI-RowI,Piece),
    put_piece(Board, ColI-RowI, empty, NewBoard1),
    put_piece(NewBoard1, ColF-RowF, Piece, NewBoard),
    other_player(Player, NewPlayer),
    forced_moves(NewBoard, NewPlayer, NewForcedMoves),
    NewTotalMoves is TotalMoves + 1,
    NewGameState = [NewBoard,NewPlayer,NewForcedMoves,NewTotalMoves].
```

---

main.pl

## Lista de jogadas válidas

A lista de jogadas válidas é obtida através do predicado **findall/3**, com o predicado objetivo **validate\_move/3**. Em testes realizados, embora muito raramente, reparou-se que é possível barrar todos os movimentos de animais com medo do jogador adversário, ficando ListOfMoves vazia (A). Assim, para que haja progressão do jogo, é também admitido um movimento com um animal que não tenha medo (B).

---

```
valid_moves(GameState, _, ListOfMoves):-
    findall(ColI-RowI-ColF-RowF,
            validate_move(GameState,ColI-RowI,ColF-RowF),
            ListOfMoves),
    \+length(ListOfMoves, 0), !.                                     A

valid_moves(GameState, Player, ListOfMoves):-                      B
    [Board,Player,_,TotalMoves] = GameState,
    findall(ColI-RowI-ColF-RowF,
            validate_move([Board,Player,[],TotalMoves],ColI-RowI,ColF-RowF),
            ListOfMoves).
```

---

main.pl

Por motivos de eficiência temporal, considerou-se apenas usar este predicado para avaliar a movimentação dos Bots. De facto, para um jogador humano, é mais vantajoso usar diretamente o predicado **validate\_move/3**, em vez de calcular todos os movimentos possíveis.

## Final do jogo

No *gameloop* é realizada a contagem do número de peças dentro de água que pertencem ao jogador no final de cada jogada. Caso o número conquistado seja inferior a uma unidade em relação ao número total de águas no tabuleiro, o jogo acaba e é declarado o vencedor da partida.

---

```
% game_over(+GameState, -Winner)
game_over([Board,OtherPlayer,_, _], Winner):-
    n_waters_to_win(NWatersToWin),
    other_player(OtherPlayer, Winner),
    count_waters(Board, Winner, NWatersToWin).
```

---

main.pl

## Avaliação do tabuleiro

Cada tabuleiro é avaliado de acordo com o posicionamento de cada peça através do predicado **value/3**, tendo em conta as águas capturadas e medos causados aos animais do adversário. A função de avaliação é utilizada no algoritmo *greedy* do modo de jogo contra um *bot* para descobrir qual será o melhor movimento a realizar.

A expressão correspondente ao cálculo do valor da avaliação do tabuleiro resulta da soma de quatro termos:

$$A*100 + B*(95 - 30*A) + C + D$$

**A** - Diferença entre o número de águas conquistadas pelo jogador atual e o número de águas conquistadas pelo adversário. A maximização deste parâmetro é prioritária e essencial à vitória.

**B** - Número de animais do adversário que, após a jogada, ficaram com medo e estão na água. Este parâmetro assegura pontos consideráveis pois obriga o oponente a mudar de posição um animal que estava na água, retirando-lhe a vantagem. Ao mesmo tempo garante que o animal que causou o medo fica adjacente a uma água. O valor total deste termo na equação também depende do valor de A.

**C** - Número de animais do adversário que, após a jogada, ficam com medo mas não estão na água. Mesmo que a jogada não consiga alcançar uma água (A) ou retirar um animal do oponente da água (B), é sempre vantajoso restringir os movimentos seguintes do adversário, uma vez que este é obrigado a mover primeiro os seus animais com medo.

**D** - Número de animais do jogador atual que estão na direção de uma água não ocupada. Traz vantagem numa próxima jogada.

Note-se globalmente a influência de A nas seguintes três situações:

- O jogador atual está em vantagem ( $A > 0$ ). Nesse caso o algoritmo irá priorizar a conquista de mais águas para mais rapidamente atingir a vitória, ou seja, maximiza o termo  $A*100$  em relação a  $B*(95 - 30*A)$ ;
- Os jogadores estão empatados ( $A = 0$ ). Nesse caso, se houver águas livres, o algoritmo irá priorizar capturar novas águas (100 pontos). Se todas as águas estiverem ocupadas, então o algoritmo irá retirar da água um animal do oponente

para obter vantagem numa próxima jogada (95 pontos, ligeiramente inferior para dar sempre prioridade ao primeiro caso);

- O jogador atual está em desvantagem ( $A < 0$ ). Nesse caso é imperativo obrigar o oponente a mover um dos seus animais que estão na água, através da maximização do termo  $B*(95 - 30*A)$ ;

---

```
% value(+GameState,+Player,-Value)
value([Board,OtherPlayer,ForcedMoves,_], Player, Value):-
    count_waters(Board, Player, Waters),
    count_waters(Board, OtherPlayer, EnemyWaters),
    WaterDiff is Waters-EnemyWaters,
    value_forced_moves(WaterDiff, ForcedMoves, 0, ForcedMovesVal),
    check_directions(Board,Player,WatersReachable),
    Value is 100*WaterDiff + ForcedMovesVal + WatersReachable.    Fórmula

% value_forced_moves(+WaterDiff,+ForcedMoves,+Acc,-Number)
value_forced_moves(_, [], Acc, Acc).
value_forced_moves(WaterDiff,[H|T], Acc, Number) :-
    water(H),
    Acc1 is Acc + 95 - WaterDiff*30,
    value_forced_moves(WaterDiff, T, Acc1, Number), !.
value_forced_moves(WaterDiff, [_|T], Acc, Number):-
    Acc1 is Acc + 1,
    value_forced_moves(WaterDiff, T, Acc1, Number), !.

% check_directions(+Board,+Player,-Result)
check_directions(Board,Player,Result):-
    findall(1,( piece_info(Type1,Player,Piece1),
                in_bounds(Board,ColI-RowI),
                position(Board,ColI-RowI,Piece1),
                water(ColF-RowF),
                position(Board,ColF-RowF,water),
                valid_direction(Type1,ColI-RowI,ColF-RowF),
                \+path_obstructed(Board,ColI-RowI,ColF-RowF)),
            List),
    length(List, Result).
```

---

main.pl

## Jogada do computador

Para os *bots* decidirem qual movimento realizar, foram realizados dois métodos: *random* e *greedy*.

O método *random*, como o nome indica, apenas escolhe de forma aleatória um movimento da lista de movimentos válidos:

---

```
% choose_move(+GameState, +Player, +Level, -Move)
choose_move(GameState, Player, 1, ColI-RowI-ColF-RowF):-
    valid_moves(GameState, Player, ListOfMoves),
    random_member(ColI-RowI-ColF-RowF, ListOfMoves).
```

---

main.pl

O método *greedy* utiliza o predicado de avaliação do tabuleiro **value/3** juntamente com o algoritmo **minimax** para avaliar todos os movimentos futuros possíveis e escolher o que é mais vantajoso.

Para a implementação do minimax, predicado **minimax/5**, tivemos em conta a ordem dos predicados constituintes para que o *backtracking* do Prolog fosse o mais eficiente possível. Mesmo assim, dada a dimensão do tabuleiro, o número de movimentos possíveis para cada animal e as inúmeras verificações necessárias para cada jogada, uma profundidade de pesquisa de três níveis mostrou-se muito demorada. Optou-se assim por avaliar apenas dois níveis, que relacionam um futuro movimento do jogador atual em função de uma seguinte possível jogada do oponente.

Em cada nível a avaliação da jogada resulta recursivamente do valor atual do tabuleiro com a soma do menor resultado do nível imediatamente abaixo. No fundo, trata-se da maximização da seguinte expressão

$$\text{value(MyNextMove)} - \text{value(NextOpponentMove)}$$

em que cada termo resulta da avaliação de níveis 1 (maximização da jogada atual) e 2 (minimização da jogada seguinte do oponente), respectivamente.

---

```
% choose_move(+GameState,+Player,+Level,-Move)
choose_move(GameState, Player, 2, ColI-RowI-ColF-RowF):-
    valid_moves(GameState, Player, ListOfMoves),
    other_player(Player, NewPlayer),
    findall(Value-Coordinate,
        (member(Coordinate, ListOfMoves),
         move(GameState, Coordinate, NewGameState),
         value(NewGameState, Player, Value1),
         minimax(NewGameState, NewPlayer, min, 1, Value2),
         Value is Value1 + Value2), Pairs),
    sort(Pairs, SortedPairs),
    last(SortedPairs, Max-_),
    findall(Coordinates, member(Max-Coordinates, SortedPairs), MaxCoordinates),
    random_member(ColI-RowI-ColF-RowF, MaxCoordinates).
```

```
% minimax(+GameState, +Player, +Type, +Level, -Value)
minimax(_, _, _, 2, 0):- !.
minimax(GameState, Player, Type, Level, Value):-
    other_player(Player, NewPlayer),
```

```
swap_minimax(Type, NewType),
NextLevel is Level + 1,
valid_moves(GameState, Player, ListOfMoves),
setof(Val, ( member(Coordinate, ListOfMoves),
              move(GameState, Coordinate, NewGameState),
              value(NewGameState, Player, Value1),
              minimax(NewGameState, NewPlayer, NewType, NextLevel, Value2),
              Val is Value1 + Value2), Values),
eval(Type, Values, Value).
```

---

main.pl

Caso o valor de dois ou mais movimentos seja o mesmo, é escolhido um deles aleatoriamente de modo a não tendenciar jogadas.

## Conclusões

O jogo Barca foi implementado com sucesso em Prolog. Pode ser jogado em modo *Player vs Player*, *Player vs Bot* e *Bot vs Bot*, tendo os bots 2 dificuldades possíveis. Com 3 tabuleiros de tamanhos diferentes e com diferentes números de peças e águas para conquistar. Todas as interações são robustas e garantem a validade do estado de jogo a todo momento.

A parte mais desafiante do projeto foi de facto a implementação do modo *greedy* do *bot* com recurso à avaliação de cada jogada possível e ao algoritmo minimax. Com isto consolidamos os conhecimentos adquiridos durante as aulas teóricas e práticas.

## Bibliografia

As regras e funcionamento do jogo foram consultadas de dois sites:

- <https://boardgamegeek.com/image/1395936/barca>
- [https://en.wikipedia.org/wiki/Barca\\_\(board\\_game\)](https://en.wikipedia.org/wiki/Barca_(board_game))