

# Programação em Lógica

## IRIS 4

---

Leonor Martins de Sousa 201705377

Sílvia Jorge Moreira da Rocha  
201704684

MIEIC 3 - Turma 1

Prof. Nuno Fonseca

Prof. Daniel Castro Silva

Prof. Rui Camacho

# Índice

Introdução	2
O Jogo	3
História	3
Material	3
Regras	3
Pontuação e Vencedor	4
A Lógica	5
Representação do Estado do Jogo	5
Visualização do Tabuleiro	7
Lista de Jogadas Válidas	8
Execução de Jogadas	8
Final do Jogo	9
Avaliação do Tabuleiro	10
Jogada do Computador	11
Conclusões	13
Bibliografia	14

## Introdução

O primeiro projeto da unidade curricular Programação em Lógica (PLOG) tem âmbito o desenvolvimento de um jogo na linguagem Prolog. O nosso grupo selecionou, de entre as opções disponíveis, o jogo Iris, descrito na secção seguinte.

Este jogo foi implementado na sua totalidade, tendo várias funcionalidades: visualização do seu estado, movimentação das peças de cada jogador, verificação do estado final do jogo e cálculo do seu vencedor.

O jogo foi implementado segundo 3 métodos: humano contra humano, humano contra computador e computador contra computador. Os jogadores simulados por computadores são simulados de acordo com 4 níveis (fácil, médio, difícil e *hardcore*).

# O Jogo

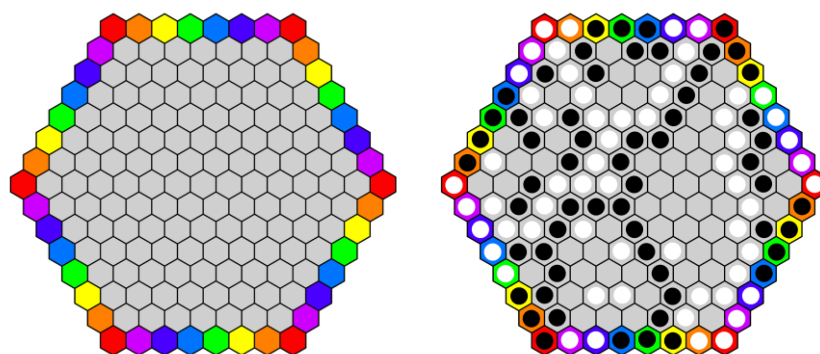
## História

Iris é um jogo que existe desde 2019 e que foi criado por Craig Duncan. Este jogo de tabuleiro enquadra-se na categoria “Estratégia Abstrata” segue um mecanismo de “Construção de Padrões” e é da família “Combinatória”. Deste modo, é um jogo sem tema/enredo, cujo resultado não é dependente da sorte / do acaso. Para além disso, é um jogo desenvolvido de forma a ter 2 jogadores, em que os jogadores alternam os turnos.

## Material

Para jogar este jogo, é necessário um tabuleiro “hexhex”, ou seja, um tabuleiro hexagonal com células hexagonais. As células que compõem o perímetro do tabuleiro são coloridas (formando as cores do arco-íris) e as restantes células são de cor cinzenta.

Para além do tabuleiro, são ainda necessárias peças de tamanho inferior ou igual às células do tabuleiro, sendo que algumas tem que ser pretas e outras brancas (pretas para o jogador 1 e brancas para o jogador 2).



**Fig. 1** - Aspeto do tabuleiro de Iris, sem e com peças.

## Regras

Na primeira jogada, o jogador 1 deverá colocar uma única peça preta numa célula cinzenta à escolha. A partir daí, começando o jogador 2, cada jogador deverá colocar 2 peças em cada jogada, seguindo as duas seguintes regras:

1. Se um jogador coloca a primeira peça numa célula colorida, a segunda peça deverá ser colocada na célula com a mesma cor que se encontra no lado oposto do tabuleiro;

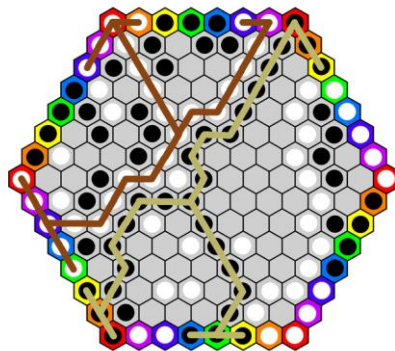
2. Se a primeira peça é colocada numa célula cinzenta, então a segunda peça deverá ser colocada numa célula cinzenta não adjacente à primeira célula. Se já todas as células cinzentas não adjacentes estiverem ocupadas, a segunda peça deverá ser ignorada.

Quando todas as células estão preenchidas ou ambos os jogadores “passam” a sua vez, o jogo termina.

## Pontuação e Vencedor

Cada jogador deverá dividir as suas peças em grupos, sendo que cada grupo é constituído por peças adjacentes. Ganha o jogador que tiver o grupo com maior pontuação. Se os grupos de maior pontuação dos dois jogadores possuírem a mesma pontuação, então comparam-se os grupos com 2ª maior pontuação, e assim consecutivamente, até ocorrer o desempate.

A pontuação de cada grupo é equivalente ao número de células coloridas que ocupa.



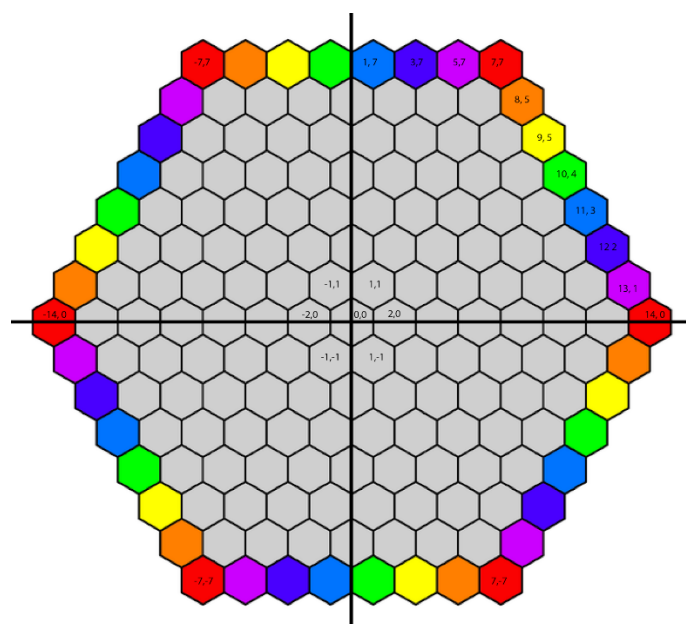
**Fig. 2** - Exemplo de grupos com maior pontuação no final de um jogo.

# A Lógica

## Representação do Estado do Jogo

A representação interna do jogo consiste numa lista de listas de listas. Deste modo o Tabuleiro é representado por uma lista de Linhas. Cada linha é uma lista, cujo *header* é o número da linha e cujos restantes elementos representam as várias células que compõem a linha. Cada célula é representada por uma lista composta por dois elementos: o número da coluna da célula e o estado atual da célula. O estado atual da célula poderá ter 3 valores: 'B' se a célula estiver vazia, 1 se a célula estiver preenchida por uma peça do jogador 1 (preta) ou 2 se a célula estiver preenchida por uma peça do jogador 2 (branca).

O sistema de coordenadas utilizado para um tabuleiro 15x15 é o apresentado abaixo.



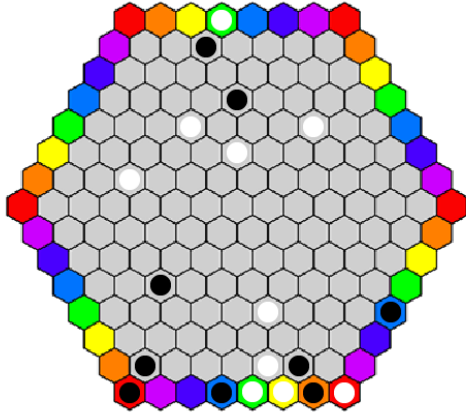


Fig. 4 – Estado Intermédio do Jogo

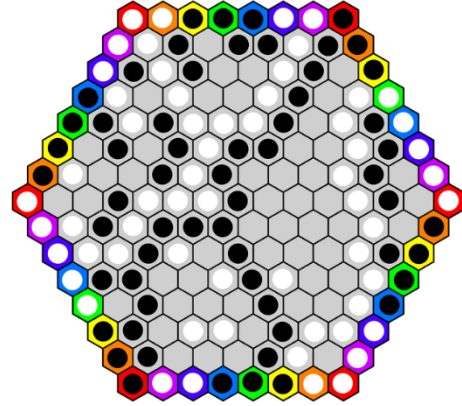


Fig. 5 – Estado Final do Jogo.

### Estado Intermédio

```
[[7, [-7, 'B'], [-5, 'B'], [-3, 'B'], [-1, '2'], [1, 'B'], [3, 'B'], [5, 'B'], [7, 'B']],
[6, [-8, 'B'], [-6, 'B'], [-4, 'B'], [-2, '1'], [0, 'B'], [2, 'B'], [4, 'B'], [6, 'B'], [8, 'B']],
[5, [-9, 'B'], [-7, 'B'], [-5, 'B'], [-3, 'B'], [1, 'B'], [1, 'B'], [3, 'B'], [5, 'B'], [7, 'B'], [9, 'B']],
[4, [-10, 'B'], [-8, 'B'], [-6, 'B'], [-4, 'B'], [-2, 'B'], [0, '1'], [2, 'B'], [4, 'B'], [6, 'B'], [8, 'B'], [10, 'B']],
[3, [-11, 'B'], [-9, 'B'], [-7, 'B'], [-5, 'B'], [-3, '2'], [-1, 'B'], [1, 'B'], [3, 'B'], [5, '2'], [7, 'B'], [9, 'B'], [11, 'B']],
[2, [-12, 'B'], [-10, 'B'], [-8, 'B'], [-6, 'B'], [-4, 'B'], [-2, 'B'], [0, '2'], [2, 'B'], [4, 'B'], [6, 'B'], [8, 'B'], [10, 'B'], [12, 'B']],
[1, [-13, 'B'], [-11, 'B'], [-9, 'B'], [-7, '2'], [-5, 'B'], [-3, 'B'], [-1, 'B'], [1, 'B'], [3, 'B'], [5, 'B'], [7, 'B'], [9, 'B'], [11, 'B'], [13, 'B']],
[0, [-14, 'B'], [-12, 'B'], [-10, 'B'], [-8, 'B'], [-6, 'B'], [-4, 'B'], [-2, 'B'], [0, 'B'], [2, 'B'], [4, 'B'], [6, 'B'], [8, 'B'], [10, 'B'], [12, 'B'], [14, 'B']],
[-1, [-13, 'B'], [-11, 'B'], [-9, 'B'], [-7, 'B'], [-5, 'B'], [-3, 'B'], [-1, 'B'], [1, 'B'], [3, 'B'], [5, 'B'], [7, 'B'], [9, 'B'], [11, 'B'], [13, 'B']],
[-2, [-12, 'B'], [-10, 'B'], [-8, 'B'], [-6, 'B'], [-4, 'B'], [-2, 'B'], [0, 'B'], [2, 'B'], [4, 'B'], [6, 'B'], [8, 'B'], [10, 'B'], [12, 'B']],
[-3, [-11, 'B'], [-9, 'B'], [-7, 'B'], [-5, '1'], [-3, 'B'], [-1, 'B'], [1, 'B'], [3, 'B'], [5, 'B'], [7, 'B'], [9, 'B'], [11, 'B']],
[-4, [-10, 'B'], [-8, 'B'], [-6, 'B'], [-4, 'B'], [-2, 'B'], [0, 'B'], [2, '2'], [4, 'B'], [6, 'B'], [8, 'B'], [10, '1']],
[-5, [-9, 'B'], [-7, 'B'], [-5, 'B'], [-3, 'B'], [-1, 'B'], [1, 'B'], [3, 'B'], [5, 'B'], [7, 'B'], [9, 'B']],
[-6, [-8, 'B'], [-6, '1'], [-4, 'B'], [-2, 'B'], [0, 'B'], [2, '2'], [4, '1'], [6, 'B'], [8, 'B']],
[-7, [-7, '1'], [-5, 'B'], [-3, 'B'], [-1, '1'], [1, '2'], [3, '2'], [5, '1'], [7, '2']]]
```

### Estado Final

```
[[7, [-7, '2'], [-5, '2'], [-3, '1'], [-1, '1'], [1, '1'], [3, '2'], [5, '2'], [7, '1']],
[6, [-8, '2'], [-6, '2'], [-4, '1'], [-2, 'B'], [0, '1'], [2, '1'], [4, '2'], [6, '1'], [8, '1']],
[5, [-9, '2'], [-7, '1'], [-5, '2'], [-3, '1'], [1, 'B'], [1, 'B'], [3, '2'], [5, '1'], [7, 'B'], [9, '1']],
[4, [-10, '1'], [-8, '2'], [-6, 'B'], [-4, '2'], [-2, 'B'], [0, 'B'], [2, '2'], [4, '1'], [6, 'B'], [8, '2'], [10, '2']],
[3, [-11, '1'], [-9, '1'], [-7, '2'], [-5, '1'], [-3, '2'], [-1, '2'], [1, '2'], [3, '1'], [5, 'B'], [7, '2'], [9, '1'], [11, '2']],
[2, [-12, '1'], [-10, 'B'], [-8, '1'], [-6, 'B'], [-4, '1'], [-2, '2'], [0, '1'], [2, '1'], [4, 'B'], [6, 'B'], [8, '2'], [10, '1'], [12, '2']],
[1, [-13, '1'], [-11, '2'], [-9, 'B'], [-7, 'B'], [-5, '1'], [-3, '2'], [-1, '1'], [1, 'B'], [3, 'B'], [5, 'B'], [7, '2'], [9, '1'], [11, 'B'], [13, '2']],
[0, [-14, '2'], [-12, 'B'], [-10, 'B'], [-8, '1'], [-6, '2'], [-4, '2'], [-2, '2'], [0, '1'], [2, 'B'], [4, 'B'], [6, 'B'], [8, '2'], [10, '1'], [12, 'B'], [14, '2']],
[-1, [-13, '2'], [-11, '2'], [-9, '1'], [-7, '2'], [-5, '1'], [-3, '1'], [-1, '1'], [1, 'B'], [3, 'B'], [5, 'B'], [7, 'B'], [9, '2'], [11, 'B'], [13, '1']],
[-2, [-12, '2'], [-10, '2'], [-8, '2'], [-6, '1'], [-4, '2'], [-2, 'B'], [0, '1'], [2, 'B'], [4, 'B'], [6, 'B'], [8, '2'], [10, '1'], [12, '1']],
[-3, [-11, '2'], [-9, '1'], [-7, '1'], [-5, 'B'], [-3, 'B'], [-1, '2'], [1, '1'], [3, '2'], [5, 'B'], [7, 'B'], [9, '2'], [11, '1']],
[-4, [-10, '2'], [-8, 'B'], [-6, '1'], [-4, 'B'], [-2, 'B'], [0, 'B'], [2, '1'], [4, 'B'], [6, 'B'], [8, '2'], [10, '1']],
[-5, [-9, '1'], [-7, '1'], [-5, 'B'], [-3, '2'], [-1, '2'], [1, 'B'], [3, '1'], [5, '2'], [7, '2'], [9, '2']],
[-6, [-8, '1'], [-6, '1'], [-4, 'B'], [-2, 'B'], [0, 'B'], [2, '1'], [4, '2'], [6, 'B'], [8, '2']],
[-7, [-7, '1'], [-5, '2'], [-3, '2'], [-1, '1'], [1, '1'], [3, '1'], [5, '2'], [7, '2']]]
```

## Visualização do Tabuleiro

A visualização do tabuleiro *hexhex* de dimensões 15x15 é feita com recurso ao código abaixo apresentado:

```
drawSpace(0).

drawSpace(N) :- write(' '), N1 is N-1, drawSpace(N1).

displayCell([_ | [P]]) :- write(P), write(' ').

displayLineCells([]).

displayLineCells([H | T]) :- displayCell(H), displayLineCells(T).

displayLine([H | T]) :- (H<0 -> write(H); write(' '), write(H)), (H>0 -> N1 is H+1, drawSpace(N1); N1 is -H+1, drawSpace(N1)), (H<0 -> write('\n '); (H>0 -> write('/ '); write(' '))), displayLineCells(T), (H<0 -> write('/ \n'); (H>0 -> write('\n \n'); write(' ' \n'))).

displayBoard([]).

displayBoard([H | T]) :- displayLine(H), displayBoard(T).

display_game(Board, _) :- displayBoard(Board).
```

Desta forma, para se poder visualizar o tabuleiro deve-se ser usado o predicado `display_game(+Board,+Player)`.

As peças do jogador 1 (peças brancas) são representadas pelo símbolo '1', as peças do jogador 2 (peças pretas) são representadas pelo símbolo '2' e as células vazias (sem nenhuma peça) são representadas pelo símbolo 'B'.

Usando a lista de listas correspondente ao tabuleiro no estado inicial, no predicado `display_game`, já mencionado acima, o resultado obtido é o visível na imagem:

```

7      / B B B B B B B B \
6      / B B B B B B B B \
5      / B B B B B B B B \
4      / B B B B B B B B \
3      / B B B B B B B B \
2      / B B B B B B B B \
1      / B B B B B B B B \
0 | B B B B B B B B B B B B B |
-1 \ B B B B B B B B B B B B /
-2 \ B B B B B B B B B B B B /
-3 \ B B B B B B B B B B B B /
-4 \ B B B B B B B B B B B B /
-5 \ B B B B B B B B B B B B /
-6 \ B B B B B B B B B B B B /
-7 \ B B B B B B B B B B B B /
```

**Fig. 6** – Visualização do Tabuleiro em Modo de Texto



## Lista de Jogadas Válidas

Para obter uma lista com todas as jogadas possíveis para um jogador, dado um determinado estado do jogo, deve ser usado o predicado `valid_moves(+Board, +Player, -ListOfMoves)`. Este predicado é implementado através do seguinte código:

```
generateMovesFromLine(Board, [[CellBoard | _] | T],[Cell | Value], LineBoard, Line, Player, ValidMoves) :-
generateMovesFromLine(Board, T, [Cell, Value], LineBoard, Line, Player, ValidMovesAux), (verifyMove(Board, Line, Cell,
LineBoard, CellBoard), (Line == LineBoard, Cell ==CellBoard -> fail;!) -> append(ValidMovesAux, [ [Line,Cell, LineBoard,
CellBoard] ], ValidMoves); ValidMoves = ValidMovesAux, !).
```

```
checkCellForIsolatedMove(Board, Line, [Cell | _], Move):- (cellEmpty(Board,Line, Cell), \+cellColor(Line, Cell) -> Move = [[Line,
Cell, [], []]]; Move = []).
```

```
valid_moves(Board, Player, ValidMoves) :- generateValidMoves(Board, Board, Player, ValidMoves1),
generateIsolatedMove(Board, Board, ValidMoves2), append(ValidMoves1, ValidMoves2, ValidMoves).
```

Os predicados `generateValidMoves` e `generateValidMovesLine`, em conjunto, percorrem cada célula do Board. Os predicados `generateValidMovesCell` e `generateMovesFromLine`, percorrem todas as outras células do tabuleiro e verificam se o conjunto das duas células constitui uma jogada válida. Estes dois predicados gerem jogadas válidas constituídas por duas peças.

Os predicados `generateIsolatedMove`, `checkLineForIsolatedMove` percorrem todas as células do Board. O predicado `generateCellForIsolatedMove` verifica se é possível uma jogada em que se só se coloque uma peça na célula e questão, de acordo com as regras do jogo.

O predicado `valid_moves` gera todas as jogadas constituídas por 2 peças (`generateValidMoves`) e todas as jogadas constituídas por 1 peça (`generateIsolatedMove`).

## Execução de Jogadas

De forma a executar uma jogada, deve-se utilizar o predicado `move(+Move, +Board, -NewBoard)`. O código seguinte implementa uma parte deste mesmo predicado:

```
verifyMove(Board, Line1, Column1, [], []) :- cellEmpty(Board, Line1, Column1), \+cellColor(Line1, Column1).
verifyMove(Board, Line1, Column1, Line2, Column2) :- cellEmpty(Board, Line1, Column1), cellEmpty(Board, Line2, Column2),
cellColor(Line1, Column1), Line2 == -Line1, Column2 == -Column1.
verifyMove(Board, Line1, Column1, Line2, Column2) :- cellEmpty(Board, Line1, Column1), cellEmpty(Board, Line2, Column2),
\+cellColor(Line1, Column1), \+cellColor(Line2, Column2), \+adjacentPieces(Line1, Column1, Line2, Column2).
```

```
move([Player,Line1,Column1,Line2,Column2], Board, NewBoard) :- verifyMove(Board, Line1, Column1, Line2, Column2),
implement_moves([Player, Line1, Column1, Line2, Column2], Board, NewBoard).
```

O predicado `cellValue` permite obter qual o valor atual de uma determinada célula, dada a sua linha e coluna. O predicado `cellEmpty` permite descobrir se uma célula está vazia. O predicado `adjacentPieces` permite descobrir se duas células são adjacentes. O predicado `cellColor` permite saber se uma célula é colorida. Todos estes predicados são predicados auxiliares mas essenciais para o predicado `move` (essencialmente para o `verifyMove`).

O predicado `verifyMove` permite saber se um determinado movimento é válido, de acordo com as regras do jogo. O predicado `implemente_move` implementa um determinado movimento, usando o predicado `changeCell` para efetivamente mudar o valor da célula em questão. O predicado `move` é responsável por verificar se uma jogada é possível (`verifyMove`) e efetuar a jogada (`implemente_moves`).

## Final do Jogo

A verificação do estado final do jogo, assim como o cálculo das pontuações e a identificação do vencedor são efetuados pelo predicado `game_over(+Board, -Winner)`. Este predicado é implementado, em parte, pelo código seguinte:

```
calculateScore([[Line,Column] | T], GroupPoints):- calculateScore(T, GroupPointsAux), (cellColor(Line, Column)-> GroupPoints is
GroupPointsAux+1; GroupPoints is GroupPointsAux, !).
```

```
calculateGroup(PlayerCells, [[Line, Column]|T], [ColoredLine, ColoredColumn], Igroup, Fgroup, IusedCells, FusedCells) :-
(adjacentPieces(Line, Column, ColoredLine, ColoredColumn), \+ member([Line, Column], IusedCells) ->
    append(IusedCells, [[Line, Column]], UsedCells1),
    append(Igroup, [[Line, Column]], Group1),
    calculateGroup(PlayerCells, T, [ColoredLine, ColoredColumn], Group1, Group2, UsedCells1, UsedCells2),
    calculateGroup(PlayerCells, PlayerCells, [Line, Column], Group2, Fgroup, UsedCells2, FusedCells);
calculateGroup(PlayerCells, T, [ColoredLine, ColoredColumn], Igroup, Fgroup, IusedCells, FusedCells)).
```

```
calculateGroups(PlayerCells, [[Line, Column]|T], Igroups, Fgroups, IusedCells, FusedCells) :-
    (member([Line,Column], IusedCells) ->
        calculateGroups(PlayerCells, T, Igroups, Fgroups, IusedCells, FusedCells);
    append(IusedCells, [[Line, Column]], UsedCells1),
    calculateGroup(PlayerCells, PlayerCells, [Line, Column], [[Line, Column]], Group1, UsedCells1, UsedCells2),
    append(Igroups, [Group1], Group2),
    calculateGroups(PlayerCells, T, Group2, Fgroups, UsedCells2, FusedCells)).
```

```
calculatePoints(Board, Player, Points) :-
    calculateCellsPlayerLines(Board, Player, PlayerCells),
    calculateColored(PlayerCells, ColoredCells),
    calculateGroups(PlayerCells, ColoredCells, [], FinalGroups, [], _),
    calculateGroupsScore(FinalGroups, Points).
```

```
calculateWinner([], [], 3).
```

```
calculateWinner([], _, 1).
```

```
calculateWinner(_, [], 2).
```

```
calculateWinner(PointsP1, PointsP2, Winner):- maxlist(PointsP1, MaxP1), maxlist(PointsP2, MaxP2),
    (MaxP1 == MaxP2 ->
        deleteElement(PointsP1, MaxP1, [], NewPointsP1),
        deleteElement(PointsP2, MaxP2, [], NewPointsP2),
        calculateWinner(NewPointsP1, NewPointsP2, Winner);
    (MaxP2 > MaxP1 -> Winner = 2; Winner = 1)).
```

```
game_over_sure(Board, Winner) :- calculatePoints(Board,1,PointsP1), calculatePoints(Board,2,PointsP2),
calculateWinner(PointsP1, PointsP2, Winner).
```

```
game_over(Board, Winner) :- (boardFull(Board) -> calculatePoints(Board,1,PointsP1), calculatePoints(Board,2,PointsP2),
calculateWinner(PointsP1, PointsP2, Winner); Winner = 0) .
```

Para verificar o estado final do jogo usa-se o predicado BoardFull, que verifica se o tabuleiro está cheio.

O predicado calculateCellsPlayerLines permite calcular as células que estão ocupadas por peças de um determinado jogador. O predicado calculateColored permite calcular, para um dado conjunto de células, quais delas são coloridas e é utilizado, sobretudo, para calcular as células coloridas de um jogador, sendo chamado com o resultado de calculateCellsPlayerLines.

O predicado calculateGroup é responsável por calcular um grupo a partir de uma célula inicial (colorida) que faz parte desse grupo. Este predicado é utilizado no predicado calculateGroups que calcula todos os grupos de um determinado jogador. O predicado calculateScore calcula os pontos de um determinado grupo de células, sendo chamada por calculateGroupsScore que calcula os pontos de todos os grupos de um jogador. O predicado calculatePoints calcula um vetor com as pontuações de todos os grupos de um jogador.

O predicado calculateWinner verifica qual o vencedor, analisando consecutivamente os vários grupos com maiores pontuações, de acordo com as regras do jogo.

O predicado game\_over, após verificar se o board está cheio (BoardFull), calcula as pontuações de cada jogador (calculatePoints) e decide qual o vencedor (calculateWinner).

É ainda de notar o predicado game\_over\_sure que é chamado no caso de o final do jogo ser detetado através da interface do jogo (quando os dois jogadores passam consecutivamente) e não através do estado do jogo.

## Avaliação do Tabuleiro

O predicado value(+Board, +Player, -Value) permite obter o valor de um determinado Board para um determinado jogador, permitindo a comparação entre Boards para decisão de melhores jogadas.

O cálculo do valor de um tabuleiro é calculado com base nas células ocupadas por peças do jogador em questão. O valor/peso de uma célula é calculado com base na sua distância ao centro do tabuleiro/às células coloridas. Desta forma, quando mais longe uma célula estiver do centro (quanto mais perto estiver de uma célula colorida) maior é a sua utilidade, visto que permitirá no futuro construir grupos melhores que, por sua vez, providenciarão uma melhor pontuação. Assim, este predicado é implementado da seguinte forma:

calculateCellWeight(Line, Column, Weight):- Weight is 2\*abs(Line)+abs(Column).

value(Board, Player, Value) :- calculateCellsPlayerLines(Board, Player, PlayerCells), calculateCellsWeight(PlayerCells, Value).

O predicado `calculateCellWeight` calcula o peso/valor de uma determinada célula. O predicado `calculateCellsWeight`, por sua vez, calcula o peso de um determinado conjunto de células. Por fim, o predicado `value`, calcula as células ocupadas por um determinado jogador (`calculateCellsPlayerLines`) e, de seguida, calcula o peso total dessas células (`calculateCellsWeight`).

## Jogada do Computador

Para poder implementar modos de jogo que permitam que um ou dois dos jogadores sejam “simulados” por computadores é necessário que exista um predicado que escolha a melhor jogada possível: `choose_move(+Board, +Level, +Player, -Move)`.

Este predicado contém uma pequena diferença relativamente ao solicitado `choose_move(+Board, +Level, -Move)`. Esta diferença deve-se ao facto de, no jogo Iris ser necessário saber qual o jogador para o qual deve ser efetuado a jogada de forma a efetuar a melhor jogada possível.

```
applyEveryMove(Board, [Move|Moves], Player, Boards) :- applyEveryMove(Board, Moves, Player, BoardsAux), append([Player], Move, MoveComplete), (move(MoveComplete, Board, NewBoard1) -> append(BoardsAux, [NewBoard1], Boards); Boards = BoardsAux).
```

```
generateAllBoardsForAll([Board | Boards], NumberPlays, Player, BestBoards):- generateAllBoardsForAll(Boards,NumberPlays, Player, BestBoardsAux),
    generateAllMoves(Board, Player, ValidMoves),
    applyEveryMove(Board, ValidMoves, Player, NewBoards1),
    NumberPlays1 is NumberPlays-1, (Player==1 -> Player1 = 2; Player1 = 1),
    (NumberPlays1 > 0 ->
        generatesBestBoard(NewBoards1, NumberPlays1, Player1, BestBoard); BestBoard= []),
    append(BestBoardsAux, [BestBoard], BestBoards).
```


```
chooseBestBoard(Board, NumberPlays, Player, BestMove):- generateAllMoves(Board, Player, ValidMoves),
    applyEveryMove(Board, ValidMoves, Player, NewBoards1),
    NumberPlays1 is NumberPlays-1, (Player==1 -> Player1 is 2; Player1 is 1), generateAllBoardsForAll(NewBoards1,
    NumberPlays1, Player1, BestBoards),
    calculateBoardsWeight(BestBoards, Player, Weights), calculateBestBoard(BestBoards, Weights, ValidMoves,
    BestBoard, BestWeight, BestMove).
```

```
choose_move(Board, 1, Player, Move):- generateAllMoves(Board, Player, ValidMoves), calculateBestMove(ValidMoves, Move).
choose_move(Board, 2, Player, Move):- chooseBestBoard(Board, 3, Player, Move).
choose_move(Board, 3, Player, Move):- chooseBestBoard(Board, 5, Player, Move).
```

O predicado `applyEveryMove` aplica uma série de `Moves` a um determinado `Board`, devolvendo os `Boards` resultado.

O predicado `calculateBestBoard`, calcula o `Board` com melhor peso.

O predicado `generateAllBoardsForAll` gera todas as jogadas possíveis, aplica-as ao `Board` e, a partir dos `Boards` resultado, escolhe o melhor `Board` para aquele nível de `NumberPlays`.

 O predicado `chooseBestBoard` escolhe a melhor jogada possível naquele momento para um determinado jogador, de acordo com o melhor Board possível que possa ser gerado para um determinado número de jogadas.

Para um nível de dificuldade 1, apenas é necessário gerar 1 jogada. Para um nível de dificuldade 2, é necessário gerar 3 jogadas, 2 para o próprio jogador e 1 para o outro jogador. Para um nível de dificuldade 3, é necessário jogar 5 jogadas, 3 para o próprio jogador e 2 para o outro.

## Conclusões

Finalizando o projeto e analisando o trabalho desenvolvido, pensamos que conseguimos construir um bom projeto e chegar a um bom resultado, sendo que atingimos os principais requisitos propostos pelos professores da unidade curricular.

Gostaríamos de ressaltar que todo o código foi desenvolvido de forma a que o *Board* possa facilmente ser alterado para tabuleiros hexhex de dimensões diferentes. Para isso basta alterar o *Board* definido no ficheiro 'display.pl' na linha 1 para um *board* válido (o programa não faz deteção de *Boards* errados). De seguida é necessário alterar no ficheiro 'game.pl', linha 11, e ficheiro 'interface.pl', linhas 20, 31 e 50, o valor 7 para um valor correspondente a  $(\text{length}(\text{Board})-1)/2$ , e alterar no ficheiro 'game.pl', linha 11, e ficheiro 'interface.pl', linhas 22, 33 e 52, o valor 14 para um valor correspondente a  $\text{length}(\text{Board})-1$ .

No entanto, é de salientar que alguns melhoramentos poderiam ser aplicados. Um dos aspetos que poderia ser aperfeiçoado é a *interface* do jogo. Contudo, este não nos parece um ponto muito relevante, visto que a mesma vai ser trabalhada ao longo da cadeira de LAIG.

A utilização de jogadores simulados por computadores apesar de funcional, é bastante lenta quando se utiliza níveis de dificuldade que não o fácil. No entanto, isto só acontece visto que o tabuleiro hexhex 15\*15 é um tabuleiro de dimensão muito elevada. De forma a poder testar-se o correto funcionamento desta funcionalidade, pode usar-se tabuleiros de menor dimensão.

Concluindo, pensamos que, apesar das dificuldades encontradas durante o desenvolvimento do projeto, conseguimos fazer um bom trabalho e cumprir todos os requisitos solicitados.

## **Bibliografia**

[Board Game Geek](#)

[Wikipedia](#)