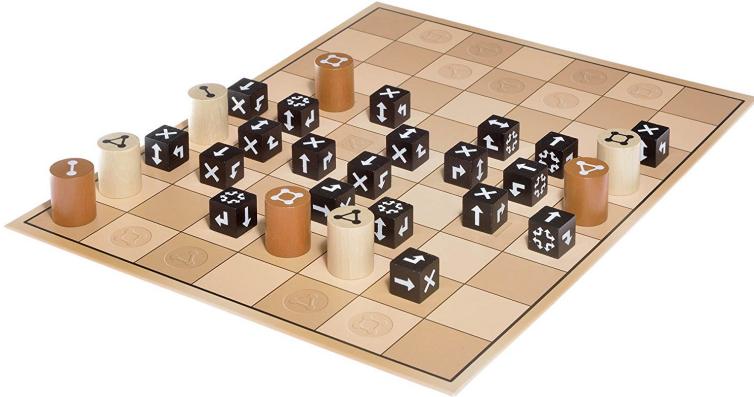


Programação em Lógica

Relatório Final

Barragoon 4



Leonardo Manuel Gomes Teixeira – up201502848
Maria Eduarda Santos Cunha – up201506524



1. Resumo

Este projeto é uma possível abordagem ao jogo Barragoon, descrito em detalhe na secção O Jogo Barragoon, em PROLOG.

Todo o processo de desenvolvimento verificou-se bastante difícil dada a novidade do paradigma da linguagem de programação em questão para nós, aliado ao facto de o próprio jogo ter uma lógica bastante complexa, pelo seu número elevado de regras e restrições.

Conseguimos implementar com sucesso todas as regras do jogo e completar os três modos de jogo – humano contra humano, humano contra computador e computador contra computador, e criamos 2 níveis de dificuldade.

Apesar do volume de regras, pensamos que a versão final é bastante *user friendly* e intuitiva de jogar.

De forma geral, o nosso feedback é bastante positivo. Este trabalho foi fundamental para a consolidação dos conteúdos lecionados nas aulas práticas e teóricas e sentimo-nos capazes de realizar os novos níveis ainda que não tenhamos tido tempo para tal.



Índice

1.	Resumo	2
2.	O Jogo Barragoon	4
2.1.	Tabuleiro e Peças.....	4
2.2.	Objetivo.....	4
2.3.	Movimentos	5
2.4.	Regras.....	5
3.	Lógica do Jogo.....	7
3.1.	Representação do Estado de Jogo	7
3.2.	Visualização do Tabuleiro.....	9
3.3.	Lista de Jogadas Válidas	11
3.4.	Execução de Jogadas.....	11
3.5.	Avaliação do Tabuleiro.....	12
3.6.	Final do Jogo	12
3.7.	Jogada do Computador.....	13
4.	Interface com o Utilizador.....	14
5.	Conclusões.....	16
6.	Bibliografia	16
7.	Anexos.....	17



2. O Jogo Barragoon

O Barragoon é um jogo de estratégia sem qualquer fator de aleatoriedade para 2 jogadores. Foi publicado pela primeira vez a 3 de março de 2014 pela companhia WiWa Spiele UG. As suas regras foram atualizadas pela última vez a 30 de março de 2016.

2.1. Tabuleiro e Peças

O jogo realiza-se num tabuleiro de 9x7 células e os jogadores jogam sempre à vez.

Existem 2 tipos principais de peças: as telhas de cada jogador, brancas ou castanhas, e os barragoons.

- As telhas dos jogadores possuem na sua face um símbolo com 2, 3 ou 4 círculos, relativo ao número de células que podem andar num só movimento (fig.1). Cada jogador começa com 7 telhas: 2 de 2 círculos, 3 de 3 círculos e 2 de 4 círculos.
- O barragoon é a peça central do jogo. É uma peça cúbica, em que cada uma das suas faces possui um símbolo que indica a permissão do jogador de mover a sua peça pela célula em que o barragoon se encontra (fig.2). O jogo começa com 8 barragoons, mas existem 32.



Figura 2: Telhas Brancas ou Castanhas



Figura 1: Faces de um Barragoon

2.2. Objetivo

Ambos os jogadores têm de recorrer às suas aptidões táticas para mover as suas telhas e dispor os barragoons de forma a que lhes seja permitido capturar todas as telhas do outro jogador ou, pelo menos, impedir o seu progresso.



O jogo acaba quando um dos jogadores já não consegue mover telhas, porque não possui nenhuma ou por as que tem se encontrarem limitadas por barragoons. O outro é o vencedor.

2.3. Movimentos

Existem 2 tipos de movimentos: full moves e short moves.

- Os full moves correspondem a percorrer x células, de acordo com o número de círculos na telha do jogador (2, 3 ou 4), respetivamente (fig.3);
- Os short moves correspondem a percorrer $x-1$ células, de acordo com o número de círculos na telha do jogador (2, 3 ou 4, logo movimentos de 1, 2 ou 3 células), respetivamente (fig.4).

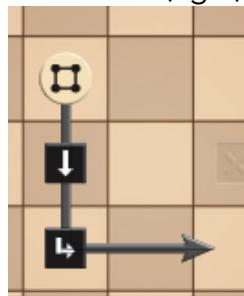


Figura 3: Full Move com Telha de 4 Círculos

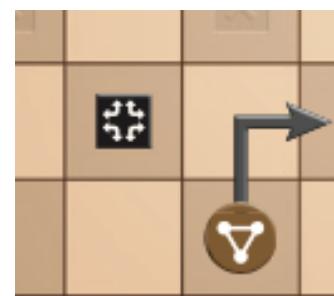


Figura 4: Short Move com Telha de 3 Círculos

2.4. Regras

- Uma peça é capturada se a peça do oponente terminar na mesma célula que ela (fig.5);
- Só é possível capturar uma peça durante um full move;
- As telhas com 2 círculos não podem capturar barragoons com o símbolo “todas as direções” virado para cima (fig.6);
- Se um barragoon for capturado, tem de voltar a ser colocado no tabuleiro, numa posição livre à escolha do jogador, com a face levantada para cima que ele preferir;
- Sempre que uma telha é capturada, são adicionados 2 barragoons novos ao tabuleiro, um por cada jogador, e coloca primeiro no tabuleiro o jogador cuja telha foi capturada;
- Nunca se pode mudar um barragoon de posição uma vez colocado;
- Quando se toca numa telha para a mover, não se pode trocar por outra ou voltar atrás;



- Durante um movimento, só se pode efetuar uma mudança de direção uma vez de 90° (fig.7);
- Os movimentos só podem ser verticais ou horizontais, nunca na diagonal.

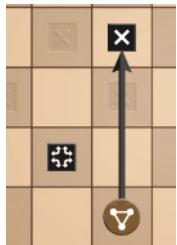


Figura 5: Captura de um Barragoon



Figura 6: Peça de 2 Círculos Não Captura Barragoon com Face “Todas as Direções” Voltada para Cima

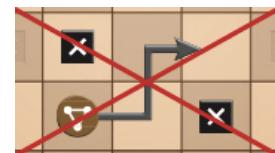


Figura 7: Movimento Impossível com 2 Mudanças de Direção

Ainda que só existam 32 barragoons e seria, por consequência, esperado que houvesse uma regra implementada para limitar a inserção de barragoons tendo em conta este valor, dado que sempre que um barragoon é comido é inserido o mesmo de volta e sempre que uma telha é comida são inseridos 2 e só há 7 telhas por jogador, concluímos que a situação em que se tenta inserir mais do que 32 nunca se vai verificar.

Por exemplo, tendo sido comidos 6 telhas de cada jogador, na situação final em que cada jogador possui 1 telha, quando um comer a do outro, o jogo acaba. E apenas nesse momento é que ocorreria a hipotética inserção do 33º.



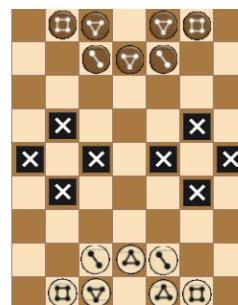
3. Lógica do Jogo

3.1. Representação do Estado de Jogo

Por questões de simplificação, os barragoons encontram-se aqui representados apenas por, por exemplo, barraX, em vez de bg-'barraX'. Na prática e em todos os fragmentos de código, referimo-nos a cada barragoon por um par, cujo primeiro elemento é bg e o segundo é a face voltada para cima.

Estado Inicial:

```
[[empty, b-4, b-3, empty, b-3, b-4, empty],
 [empty, empty, b-2, b-3, b-2, empty, empty],
 [empty, empty, empty, empty, empty, empty, empty],
 [empty, barraX, empty, empty, empty, barraX, empty],
 [barraX, empty, barraX, empty, barraX, empty, barraX],
 [empty, barraX, empty, empty, empty, barraX, empty],
 [empty, empty, empty, empty, empty, empty, empty],
 [empty, empty, w-2, w-3, w-2, empty, empty],
 [empty, w-4, w-3, empty, w-3, w-4, empty]]
```



		b4		b3				b3		b4	
			b2		b3		b2				
		x							x		
	x		x				x		x		
		x						x		x	
			w2		w3		w2				
		w4		w3				w3		w4	

Estados intermédios:

1.

```
[[empty, b-4, b-3, empty, b-3, b-4, empty],
 [empty, empty, b-2, b-3, b-2, empty, empty],
 [empty, empty, empty, empty, empty, empty, empty],
 [empty, barraX, empty, empty, empty, barraX, empty],
 [barraX, empty, barraX, w-3, barraX, empty, barraX],
 [empty, barraX, empty, empty, empty, barraX, empty],
 [empty, empty, empty, empty, empty, empty, empty],
 [empty, empty, w-2, empty, w-2, empty, empty],
 [empty, w-4, w-3, empty, w-3, w-4, empty]]
```

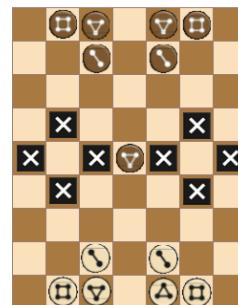


		b4		b3				b3		b4	
			b2		b3		b2				
	x		x		w3		x		x		
	x						x		x		
		x					x		x		
			w2				w2				
	w4		w3				w3		w4		



2.

```
[[empty, b-4, b-3, empty, b-3, b-4, empty],
 [empty, empty, b-2, empty, b-2, empty, empty],
 [empty, empty, empty, empty, empty, empty, empty],
 [empty, barraX, empty, empty, empty, barraX, empty],
 [barraX, empty, barraX, b-3, barraX, empty, barraX],
 [empty, barraX, empty, empty, empty, barraX, empty],
 [empty, empty, empty, empty, empty, empty, empty],
 [empty, empty, w-2, empty, w-2, empty, empty],
 [empty, w-4, w-3, empty, w-3, w-4, empty]]
```



	b4	b3	b3	b4	
	b2	b2	b2		
	X				X
X		X	b3	X	X
	X				X
	w2		w2		
	w4	w3	w3	w4	

3.

```
[[empty, b-4, b-3, empty, b-3, b-4, empty],
 [empty, empty, b-2, empty, empty, empty, empty],
 [empty, empty, empty, empty, empty, empty, empty],
 [empty, barraX, empty, empty, b-2, barraX, empty],
 [barraX, empty, barraX, b-3, barraX, empty, barraX],
 [empty, barraX, empty, allDir, empty, barraX, empty],
 [empty, empty, empty, empty, right, w-2, empty],
 [empty, empty, w-2, empty, empty, empty, empty],
 [empty, w-4, w-3, empty, w-3, w-4, empty]]
```



	b4	b3	b3	b4	
	b2	b2	b2		
	X				X
X		X	b3	X	X
	X		*		X
			>	w2	
	w2				
	w4	w3	w3	w4	



4.

```
[[empty, b-4, b-3, empty, b-3, b-4, empty],
 [empty, empty, b-2, empty, empty, empty, empty],
 [empty, empty, empty, empty, empty, empty, empty],
 [empty, bg-'barraX', empty, empty, b-2, barraX, empty],
 [barraX, empty, barraX, empty, barraX, empty, barraX],
 [empty, b-3, empty, allDir, empty, barraX, empty],
 [empty, empty, empty, empty, right, empty, empty],
 [empty, empty, w-2, w-3, empty, empty, empty],
 [empty, w-4, w-3, empty, w-3, w-4, empty]]
```



I	b4	I	b3	I	I	b3	I	b4	I	
I	I	I	b2	I	I	I	I	I	I	
I	I	I	I	I	I	I	I	I	I	
I	I	X	I	I	I	b2	I	X	I	
I	X	I	I	X	I	I	X	I	I	
I	I	b3	I	I	*	I	I	X	I	
I	I	I	I	I	I	>	I	I	I	
I	I	I	w2	I	w3	I	I	I	I	
I	I	I	w2	I	I	I	I	I	I	
I	I	w4	I	w3	I	I	w3	I	w4	I

Estado final:

```
[[barraX, b-4, empty, b-2, b-3, b-4, barraX],
 [empty, barraX, empty, empty, empty, barraX, empty],
 [empty, barraX, empty, empty, empty, empty, empty],
 [barraX, w-2, barraX, empty, empty, empty, empty],
 [empty, barraX, empty, empty, empty, barraX, barraX],
 [empty, empty, empty, empty, b-2, empty, empty],
 [empty, empty, empty, empty, barraX, empty, empty],
 [empty, empty, barraX, b-3, barraX, empty, empty],
 [empty, barraX, w-3, barraX, w-3, barraX, empty]]
```



I	X	I	b4	I	I	b2	I	b3	I	b4	I	X
I	I	X	I	I	I	I	I	I	I	X	I	I
I	I	X	I	I	I	I	I	I	I	I	I	I
I	X	I	w2	I	X	I	I	I	I	I	I	I
I	I	X	I	I	I	I	I	X	I	X	I	I
I	I	I	I	I	I	I	I	X	I	I	I	I
I	I	I	I	I	X	I	b3	I	X	I	I	I
I	I	I	X	I	w3	I	X	I	w3	I	X	I

3.2. Visualização do Tabuleiro

```

1 initialBoard(
2   [[empty, b-4, b-3, empty, b-3, b-4, empty],
3    [empty, empty, b-2, b-3, b-2, empty, empty],
4    [empty, empty, empty, empty, empty, empty, empty],
5    [empty, bg-'barraX', empty, empty, empty, bg-'barraX', empty],
6    [bg-'barraX', empty, bg-'barraX', empty, bg-'barraX', empty, bg-'barraX'],
7    [empty, bg-'barraX', empty, empty, empty, bg-'barraX', empty],
8    [empty, empty, empty, empty, empty, empty, empty],
9    [empty, empty, w-2, w-3, w-2, empty, empty],
10   [empty, w-4, w-3, empty, w-3, w-4, empty]]).
11
12 displayGame(Game) :-
```



```

13  getBoard(Game, Board),
14  getCurrentPlayer(Game, Player),
15  clearScreen,
16  displayPlayerTurn(Player),
17  lettersAxis, nl,
18  horizontalBorder, nl,
19  numbersAxis(RowNumbers),
20  displayBoard(Board, RowNumbers), nl.
21  displayBoard([], []).
22
23  displayBoard([RowToDisplay|RemainingBoard],
24    [RowToDisplayNumber|RemainingRowNumbers]) :-
25    translate([RowToDisplayNumber]),
26    translate(RowToDisplay), border, nl,
27    horizontalBorder, nl,
28    displayBoard(RemainingBoard, RemainingRowNumbers).
29
30  % -- Board Translation --
31  translate([]).
32  translate(['empty'|R]) :- border, write(' _'), !, translate(R).
33  translate([w-2|R]) :- border, write(' w2 '), !, translate(R).
34  translate([w-3|R]) :- border, write(' w3 '), !, translate(R).
35  translate([w-4|R]) :- border, write(' w4 '), !, translate(R).
36  translate([b-2|R]) :- border, write(' b2 '), !, translate(R).
37  translate([b-3|R]) :- border, write(' b3 '), !, translate(R).
38  translate([b-4|R]) :- border, write(' b4 '), !, translate(R).
39  translate([bg-'barraX'|R]) :- border, write(' X '), !, translate(R).
40  translate([bg-'allDir'|R]) :- border, write(' + '), !, translate(R).
41  translate([bg-'oDirU'|R]) :- border, write(' V '), !, translate(R).
42  translate([bg-'oDirD'|R]) :- border, write(' A '), !, translate(R).
43  translate([bg-'oDirL'|R]) :- border, write(' <= '), !, translate(R).
44  translate([bg-'oDirR'|R]) :- border, write(' => '), !, translate(R).
45  translate([bg-'tDirH'|R]) :- border, write(' X '), !, translate(R).
46  translate([bg-'tDirV'|R]) :- border, write(' X '), !, translate(R).
47  translate([bg-'DtoR'|R]) :- border, write(' .> '), !, translate(R).
48  translate([bg-'DtoL'|R]) :- border, write(' <.'), !, translate(R).
49  translate([bg-'UtoR'|R]) :- border, write(' \">>'), !, translate(R).
50  translate([bg-'UtoL'|R]) :- border, write(' <\''), !, translate(R).
51  translate([bg-'LtoU'|R]) :- border, write(' -^'), !, translate(R).
52  translate([bg-'LtoD'|R]) :- border, write(' -v'), !, translate(R).
53  translate([bg-'RtoU'|R]) :- border, write(' ^-'), !, translate(R).
54  translate([bg-'RtoD'|R]) :- border, write(' v-'), !, translate(R).
55  translate(['um'|R]) :- write('1'), !, translate(R).
56  translate(['dois'|R]) :- write('2'), !, translate(R).
57  translate(['tres'|R]) :- write('3'), !, translate(R).
58  translate(['quatro'|R]) :- write('4'), !, translate(R).
59  translate(['cinco'|R]) :- write('5'), !, translate(R).
60  translate(['seis'|R]) :- write('6'), !, translate(R).
61  translate(['sete'|R]) :- write('7'), !, translate(R).
62  translate(['oito'|R]) :- write('8'), !, translate(R).
63  translate(['nove'|R]) :- write('9'), !, translate(R).
64
65  % -- Board Axis --
66  numbersAxis([um, dois, tres, quatro, cinco, seis, sete, oito, nove]).
67
68  lettersAxis :- write(' A B C D E F G').
69
70  % -- Board Borders --

```



```

71 horizontalBorder :- write('-----').
72 border :- write('|').

```

3.3. Lista de Jogadas Válidas

O predicado **getMovesAvailable(+Game, +Row, +Column, -List)** retorna uma lista com as várias opções de percurso que uma peça pode percorrer, já validadas.

Esta função processa-se através da obtenção da telha a ser jogada com **getCell(+Game, +Row, +Column, -List)**. Assim que sabemos o tipo de telha (2, 3 ou 4 círculos), obtemos em **availableMoves(+NDots, -allMovesAvailable)** todos os percursos existentes para uma telha dessas. Finalmente, com **getMovesAvailableAux(+Game, +Row, +Column, +AllMovesAvailable, -List)** retornamos a lista final com os percursos após serem validados por **validatePath(+Row, +Column, +Path, +false)** e **validateMove(+Game, +Row, +Column, +Path, _, +false)**.

Em **validatePath** é verificado se o movimento não causa o ultrapassar dos limites do tabuleiro e se apenas se dá uma mudança de direção.

Em **validateMove** verifica-se se o caminho corresponde a um tipo de movimento full ou short e se as telhas estão a deslocar-se corretamente por cima dos barragoons dispostos no tabuleiro.

Este cálculo automático da lista de jogadas válidas só tem relevância aquando das jogadas do computador.

3.4. Execução de Jogadas

O ciclo principal **playGame(+Game, -NewGame)** é caracterizado pela execução da jogada **playerTurn(+Game, -UpdatedGame)**, responsável por efetuar todos os pontos relativos à jogada de um jogador, **switchPlayer(+UpdatedGame, -NextPlayerGame)**, que trata de alternar os jogadores a cada jogada, e a chamada recursiva de **playGame**.

O predicado **playerTurn** é constituído por **playerMove(+Game, +RowSrc, +ColSrc, -Path)** ou **botMove(+Game, -Row, -Column, -Path)** dependendo do modo jogo que se trata, **validateMove(+Game, +RowSrc, +ColSrc, +Path, -PieceCaptured)**, **movePiece(+Game, +RowSrc, +ColSrc, +Path, -NewGame1)** e uma verificação de se alguma peça, telha ou barragoon, foi capturada.

Em **playerMove**, é pedido ao jogador que escolha a telha que pretende mover, através da inserção na consola da linha, representada por um número de 1 a 9, e da coluna, representada por uma letra de A a G (**chooseTile**), é validado que essa telha existe e lhe pertence (**validateTile**), pede-se o percurso que a peça vai efetuar,



representado pelas teclas WASD (`choosePath`) e, por fim, verifica-se se esse caminho não ultrapassa os limites do tabuleiro (`validatePath`).

O predicado `botMove`, que existe em paralelo com `playerMove`, ao invés de receber do jogador as coordenadas e caminho, gera-os aleatoriamente (`getRandomElemFromList`) a partir de listas de posições (`getPlayerPieces`) e percursos (`getMovesAvailable`) que já foram selecionados por serem válidos. Isto torna a função seguinte, `validateMove`, algo desnecessária para esta jogada. Com algum tempo, podia haver alguma reestruturação de código de forma a eliminar esta redundância.

Em `validateMove`, trata-se de todas as questões relacionadas com o movimento da peça respeitar as regras do jogo. Verificando através de `getCell(+Board, +RowSrc, +ColSrc, -Piece)` qual é a telha que está a ser movida, com recurso a `isShortMove(+Piece, +Path)` e `isFullMove(+Piece, +Path)` determinamos se o percurso inserido pelo jogador é válido. Ainda, em `validateCrossMovements(+Game, +RowSrc, +ColSrc, +Path, +IsLongMove, +Piece, -PieceCaptured)` calcula-se se o movimento em questão obedece às regras de passagem em cima de barragoons e se não passa por cima de telhas se não for com o objetivo de as comer.

Em `movePiece`, é calculada a célula final em que a telha deve acabar através de `getDestCellFromPath(+RowSrc, +ColSrc, +Path, -RowDest, -ColDest)`, elimina-se o conteúdo dessa posição e procede-se à colocação da telha no novo destino com `moveFromSrcToDest(+Game, +RowSrc, +ColSrc, +RowDest, +ColDest, -NewGame)`.

3.5. Avaliação do Tabuleiro

No primeiro nível de dificuldade, não há qualquer tipo de avaliação das jogadas favoráveis. O computador joga sempre aleatoriamente.

No segundo nível de dificuldade, as jogadas do computador são determinadas de acordo com um sistema de prioridades, sendo que, quando são calculados os movimentos possíveis para a telha selecionada, se resultarem em telhas comidas, são colocados na cabeça da lista. Caso contrário, vão para o fim da lista.

3.6. Final do Jogo

O jogo termina quando um dos jogadores já não tem telhas ou as que possui não se conseguem deslocar.

De forma a lidar com o primeiro ponto, utilizamos o predicado `countPlayerPieces(+Board, +CurrentPlayer, -CountPieces)` que verifica no tabuleiro o



número de telhas do jogador cuja vez for de jogar. Se **CountPieces** retornar a 0, o jogo acaba e esse jogador perde.

Para verificar se, existindo peças, estas possuem movimentos possíveis, recorremos a **countMovesAvailable(+Game, +Row, +Column, -Count)** para contar os movimentos de uma peça. Se **Count** retornar a 0 para todas as peças em tabuleiro do jogador, o jogo acaba e esse jogador perde. Esta solução é um pouco *hardcoded*, pois recorremos a todos os percursos possíveis para cada telha para efetuar esta verificação.

3.7. Jogada do Computador

No primeiro nível de jogo, tanto no modo humano contra computador como computador contra computador, as jogadas são completamente aleatórias. A forma como a escolha de telha a mover e percurso a efetuar se processa está extensivamente descrita nas secções Lista de Jogadas Válidas e Execução de Jogadas.

No segundo nível de jogo, de acordo com diferentes consequências possíveis de um movimento, comer uma telha ou não, define-se uma lista com prioridades. Quando o caminho resulta em comer uma telha, passa para a cabeça da lista, caso contrário, vai para o fim. Em vez de escolher um percurso aleatoriamente da lista, escolhe sempre o primeiro, pois é garantido que resulta numa telha comida.



4. Interface com o Utilizador

Quando iniciado com o comando `start`, é apresentado o menu principal com as opções de jogar em modo humano contra humano, humano contra computador e computador contra computador, apresentar as regras ou sair do jogo.

Se for escolhida a opção 4 - Regras, é apresentado o seguinte menu que enumera as regras relevantes para o jogador.

Press any key to continue...|



Iniciado o jogo em modo humano contra humano, as jogadas apresentam-se com um cabeçalho que indica a vez do jogador – White Player ou Black Player, o tabuleiro de jogo com eixos de números para as linhas e letras para as colunas e, por fim, perguntas relativas ao *input* do utilizador, como coordenadas da peça a mover e caminho que a peça deve seguir. Caso alguma destas informações seja inserida de forma errada, é sempre apresentada uma mensagem de erro e é pedido de novo ao jogador que insira a informação em questão até que a insira corretamente.

Se for o caso de inserir um barragoon no tabuleiro, é pedido ao respetivo jogador que insira a informação relativa a essa colocação (coordenadas de destino do barragoon e a face que se pretende que fique voltada para cima).

```
*****
*          White Player      *
*****
A   B   C   D   E   F   G
-----
1|   | b4 | b3 |   | b3 | b4 |   |
-----  

2|   |   | b2 | b3 | b2 |   |   |
-----  

3|   |   |   |   |   |   |   |
-----  

4|   | X |   |   |   | X |   |
-----  

5| X |   | X |   | X |   | X |  

-----  

6|   | X |   |   |   | X |   |
-----  

7|   |   |   |   |   |   |   |
-----  

8|   |   | w2 | w3 | w2 |   |  

-----  

9|   | w4 | w3 |   | w3 | w4 |  

-----
```

Which tile would you like to move?
Row: 9
Column: b

Please insert the path that you want that piece to follow:
(Use WASD - eg. wwwd + Enter).
l: wwwd

```
*****
*          White Player      *
*****
A   B   C   D   E   F   G
-----
1|   |   |   |   | b3 | b4 |   |
-----  

2| b3 |   | b2 | b3 | b2 |   |  

-----  

3|   |   | b4 |   |   |   |   |
-----  

4|   | X |   |   |   | X |   |
-----  

5| w3 |   | X |   | X |   | X |  

-----  

6|   | X |   |   |   | X |   |
-----  

7|   |   | w4 |   |   |   |  

-----  

8|   |   | w2 | w3 | w2 |   |  

-----  

9|   |   |   |   | w3 | w4 |  

-----
```

Where do you wish to place your barragoon?
Row: 9
Column: a

Which barragoon do you wish to insert?

The options are:

- 1- X
 - 2- + (All Directions)
 - 3- => (One Direction)
 - 4- <=> (Two Directions)
 - 5- '>' (Left Turn)
 - 6- '<' (Right Turn)
- l: 4

In which direction you want to set the barragoon?

The options are:

- 1- - (Horizontal)
 - 2- I (Vertical)
- l:



5. Conclusões

Concluído o projeto, achamos importante ressaltar aquilo que poderia ter sido melhorado e não foi apenas pela ausência de tempo. Destaca-se a existência de soluções ditas *hardcoded* (por exemplo, o cálculo do fim do jogo através da verificação de todos os paths possíveis) e a simplicidade do segundo nível de dificuldade.

Pretendíamos ter criado uma lista paralela à lista de percursos válidos do computador com valores do tipo 0, 1 ou 2 caso resultassem em não comer nenhuma peça, comer um barragoon ou comer uma telha, respectivamente. Acabamos por fazer apenas uma lista organizada com percursos que resultem em telha comida a serem enviados para a cabeça e o resto para o fim da lista.

Ainda, pensamos que a nossa implementação é bastante intuitiva e de fácil compreensão, mas a sua complexidade poderia ter sido sensivelmente reduzida com alguma reutilização de código e *layering*.

Concluindo, consideramos que realizamos com sucesso aquilo a que nos propusemos e com o tempo que nos foi dado. Os conceitos das aulas foram, sem dúvida, consolidados e temos agora uma nova compreensão da utilidade desta linguagem de programação.

6. Bibliografia

- [1] https://www.youtube.com/watch?v=qG1i0_sn_FI
- [2] <https://boardgamegeek.com/boardgame/157779/barragoon>
- [3] http://www.barragoon.de/bsp/BARRAGOON_en.pdf
- [4] <https://stackoverflow.com>
- [5] <http://www.swi-prolog.org>



7. Anexos

main.pl

```

1  :- include('Utilities.pl').
2  :- include('Interface.pl').
3  :- include('Logic.pl').
4  :- use_module(library(system)).
5
6  %-----%
7  %---Barragoon -----%
8  %-----%
9  %---- escreva start -----%
10 %--- na consola para correr ---%
11 %-----%
12 %-----%
13
14 % --- START ---
15 start :-  
16 clearScreen,  
17 mainMenu.
```

Interface.pl

```

1  %-----%
2  %---Interface-----%
3  %-----%
4
5  %
6  %----- MENUS -----
7  %
8
9  % -- Menus --
10 mainMenu :-  
11 displayMainMenu,  
12 getCharThenEnter(Option),  
13 (  
14 Option = '1' -> startGamePvP;  
15 Option = '2';  
16 Option = '3';  
17 Option = '4' -> displayRules;  
18 Option = '5';  
19  
20 clearScreen,  
21 write('ERROR : invalid input...'), spacing(1),  
22 mainMenu  
23 ).  
24
25 displayMainMenu :-  
26 upperFrame,  
27 titleFrame,  
28 write('| *****|'),nl,  
29 write('| * Main Menu * |'),nl,  
30 write('| *****|'),nl,  
31 write('| Play: |'),nl,  
32 write('| 1 - Player vs Player |'),nl,  
33 write('| 2 - Player vs Computer |'),nl,  
34 write('| 3 - Computer vs Computer Easy |'),nl,
```





```

93    write('
94        !'),nl.
95    % -- Frames --
96    upperFrame :-
97    write(' _____'),nl,
98    write('|           |'),nl.
99
100   lowerFrame :-
101      write('
102      |           |'),nl,
103
104      % -----
105      % ----- BOARD -----
106      % -----
107
108      % -- BOARD --
109      initialBoard( [[empty, b-4, b-3, empty, b-3, b-4, empty],
110      [empty, empty, b-2, b-3, b-2, empty, empty],
111      [empty, empty, empty, empty, empty, empty, empty],
112      [empty, bg-barraX, empty, empty, empty, bg-barraX, empty],
113      [bg-barraX, empty, bg-barraX, empty, bg-barraX, empty, bg-barraX],
114      [empty, bg-barraX, empty, empty, empty, bg-barraX, empty],
115      [empty, empty, empty, empty, empty, empty, empty],
116      [empty, empty, w-2, w-3, w-2, empty, empty],
117      [empty, w-4, w-3, empty, w-3, w-4, empty]]).
118
119      displayGame(Game) :-
120          getBoard(Game, Board),
121          getCurrentPlayer(Game, Player),
122
123          clearScreen,
124          displayPlayerTurn(Player),
125          lettersAxis,nl,
126          horizontalBorder, nl,
127          numbersAxis(RowNumbers),
128          displayBoard(Board, RowNumbers), nl.
129
130      displayBoard([], []).
131      displayBoard([RowToDisplay|RemainingBoard],
132      [RowToDisplayNumber|RemainingRowNumbers]) :-
133          translate([RowToDisplayNumber]),
134          translate(RowToDisplay),border, nl,
135          horizontalBorder, nl,
136          displayBoard(RemainingBoard, RemainingRowNumbers).
137
138      % -- Board Translation --
139      translate([]).
140      translate(['empty'|R]) :- border, write('  '), !, translate(R).
141      translate(['w-2|R']) :- border, write(' w2 '), !, translate(R).
142      translate(['w-3|R']) :- border, write(' w3 '), !, translate(R).
143      translate(['w-4|R']) :- border, write(' w4 '), !, translate(R).
144      translate(['b-2|R']) :- border, write(' b2 '), !, translate(R).
145      translate(['b-3|R']) :- border, write(' b3 '), !, translate(R).
146      translate(['b-4|R']) :- border, write(' b4 '), !, translate(R).
147      translate(['bg-'barraX|R]) :- border, write(' X '), !, translate(R).
148      translate(['bg-'allDir|R]) :- border, write(' + '), !, translate(R).
149      translate(['bg-'oDirU|R]) :- border, write(' A '), !, translate(R).
150      translate(['bg-'oDirD|R]) :- border, write(' V '), !, translate(R).

```



```

151  translate([bg-'oDirL'|R]) :- border, write(' <= '), !, translate(R).
152  translate([bg-'oDirR'|R]) :- border, write(' => '), !, translate(R).
153  translate([bg-'tDirH'|R]) :- border, write(' - '), !, translate(R).
154  translate([bg-'tDirV'|R]) :- border, write(' | '), !, translate(R).
155  translate([bg-'DtoR'|R]) :- border, write(' .> '), !, translate(R).
156  translate([bg-'DtoL'|R]) :- border, write(' <.'), !, translate(R).
157  translate([bg-'UtoR'|R]) :- border, write(' \\'>'), !, translate(R).
158  translate([bg-'UtoL'|R]) :- border, write(' <\'' ), !, translate(R).
159  translate([bg-'LtoU'|R]) :- border, write(' -^'), !, translate(R).
160  translate([bg-'LtoD'|R]) :- border, write(' -v'), !, translate(R).
161  translate([bg-'RtoU'|R]) :- border, write(' ^-'), !, translate(R).
162  translate([bg-'RtoD'|R]) :- border, write(' v-'), !, translate(R).
163  translate(['1'|R]) :- write('1'), !, translate(R).
164  translate(['2'|R]) :- write('2'), !, translate(R).
165  translate(['3'|R]) :- write('3'), !, translate(R).
166  translate(['4'|R]) :- write('4'), !, translate(R).
167  translate(['5'|R]) :- write('5'), !, translate(R).
168  translate(['6'|R]) :- write('6'), !, translate(R).
169  translate(['7'|R]) :- write('7'), !, translate(R).
170  translate(['8'|R]) :- write('8'), !, translate(R).
171  translate(['9'|R]) :- write('9'), !, translate(R).
172
173 % -- Board Axis --
174 numbersAxis(['1', '2', '3', '4', '5', '6', '7', '8', '9']).
175
176 lettersAxis :- write(' A B C D E F G').
177
178 % -- Board Borders --
179 horizontalBorder :- write(' -----').
180 border :- write('|').

```

Logic.pl

```

1  %-----%
2  %-----Game Logic-----%
3  %-----%
4  %
5  % ----- GAME -----
6  %
7  startGame(Mode) :-
8  initializeGame(Game, Mode),
9  playGame(Game).
10
11 initializeGame(Game, Mode) :-
12  initialBoard(Board),
13  Game = [Board, w, Mode], !.
14
15  % --- Game Loop ---
16  playGame(Game) :-
17  ifelse(isGameOver(Game),
18    (
19      i. switchPlayer(Game, EndGame),
20      ii. gameOver(EndGame)
21    ),
22    (
23      i. playerTurn(Game, UpdatedGame),

```



```

ii.    switchPlayer(UpdatedGame, NextPlayerGame),
iii.   playGame(NextPlayerGame)
d.    )).

16   isGameOver(Game) :-
17     getBoard(Game, Board),
18     getCurrentPlayer(Game, CurrentPlayer),

19     %check is current player still has any piece
20     countPlayerPieces(Board, CurrentPlayer, CountPieces),
21     !,
22     ifelse(CountPieces = 0,
a.      true,
b.      (
i.        countMovesAvailableAllPieces(Game, CountMoves),
ii.        !,
iii.        ifelse(CountMoves = 0,
iv.          true,
v.          fail
vi.        )
c.      )
23    ).

24   gameOver(Game) :-
25     nl, write('And the Winner is...'), nl, nl,
26     displayGame(Game),
27     nl, write('Bye').

28   playerTurn(Game, NewGame) :-
29     displayGame(Game),

30     %do playerMove while not validateMove:

31     getPlayerType(Game, PlayerType),
32     repeat,
33     (
a.       PlayerType = player -> playerMove(Game, RowSrc, ColSrc, Path);
b.       PlayerType = bot -> botMove(Game, RowSrc, ColSrc, Path)
34     ),
35     validateMove(Game, RowSrc, ColSrc, Path, PieceCaptured),
36     !,
37     movePiece(Game, RowSrc, ColSrc, Path, NewGame1),

38     (
a.       PieceCaptured = barragoonPiece -> barragoonCaptured(NewGame1,NewGame);
b.       PieceCaptured = playerPiece -> playerPieceCaptured(NewGame1, NewGame);
c.       PieceCaptured = empty -> NewGame = NewGame1
39     ).

40   playerMove(Game, RowSrc, ColSrc, Path) :-
41     %do chooseTile while not validateTile:
42     repeat,

```



```

43      (
44      chooseTile(RowSrc, ColSrc, 'Which tile would you like to move?')
45      ),
46      validateTile(Game, RowSrc, ColSrc),
47      !,
48      %do choosePath while not validatePath:
49      repeat,
50      (
51      choosePath(Path, 'Please insert the path that you want that piece to follow:\n(Use
WASD - eg. wwwd + Enter).')
52      ),
53      validatePath(RowSrc, ColSrc, Path),
54      !.
55      botMove(Game, Row, Column, Path):-  

56      repeat,
57      (
58      a. getPlayerPieces(Game, Pieces),
59      b. getRandomElemFromList(Pieces, [Row, Column]),
60      c. getMovesAvailable(Game, Row, Column, MovesAvailable)
61      ),
62      botChooseMove(Game, MovesAvailable, Path).
63  

64      % -----
65      % ----- MOVEMENTS -----
66      % -----  

67      getDestCellFromPath(RowSrc, ColSrc, [], RowDest, ColDest) :-
68      RowDest = RowSrc,
69      ColDest = ColSrc.
70      getDestCellFromPath(RowSrc, ColSrc, [Move|Tail], RowDest, ColDest) :-
71      (
72      Move == 'w' -> (RowDest1 is RowSrc-1,
getDestCellFromPath(RowDest1,ColSrc,Tail,RowDest,ColDest));
73      Move == 's' -> (RowDest1 is RowSrc+1,
getDestCellFromPath(RowDest1,ColSrc,Tail,RowDest,ColDest));
74      Move == 'a' -> (ColDest1 is ColSrc-1,
getDestCellFromPath(RowSrc,ColDest1,Tail,RowDest,ColDest));
```



```

75      Move == 'd' -> (ColDest1 is ColSrc+1,
getDestCellFromPath(RowSrc,ColDest1,Tail,RowDest,ColDest))
76    ).

77  % --- Move piece ---
78  movePiece(Game, RowSrc, ColSrc, Path, NewGame) :-
79    getDestCellFromPath(RowSrc, ColSrc, Path, RowDest, ColDest),
80    moveFromSrcToDest(Game,RowSrc,ColSrc,RowDest,ColDest,NewGame).

81  moveFromSrcToDest(Game, RowSrc, ColSrc, RowDest, ColDest, NewGame) :-
82    clearCell(Game, RowSrc, ColSrc, Value, NewGame1),
83    setCell(NewGame1,RowDest, ColDest, Value, NewGame).

84  % --- Check if it is a short move ---
85  %isShortMove(+Piece, +Path)
86  isShortMove(_Player-NDots, Path) :-
87    length(Path, N),

88  N1 is N+1,
89  N1 == NDots.

90  % --- Check if it is a long move ---
91  %isFullMove(+Piece, +Path)
92  isFullMove(_Player-NDots, Path) :-
93    length(Path, N),
94  N == NDots.

95  % -----
96  % ----- IDENTIFY PIECES -----
97  % -----


98  % --- Check if it is a valid tile ---
99  %validateTile(+Game, +RowSrcPos, +ColSrcPos): make sure the position
100 corresponds to a piece of the player
100 validateTile(Game, RowSrc, ColSrc) :-  

101
101  getCurrentPlayer(Game, CurrentPlayer),
102  getCell(Game, RowSrc, ColSrc, Piece),
103  Piece = CurrentPlayer-_.

104  validateTile(_Game, _RowSrc, _ColSrc) :-
105  write('There\'s not a movable piece in that position.'), nl,
106  write('Please, try another position.'), nl,
107  fail.
108  %go_back to repeat cycle

109  % -----
110  % ----- MOVEMENT RULES -----
111  % -----


112  % --- Check if it is a valid path ---
113  validatePathValues([]).
114  validatePathValues([H|T]) :-
115  member(H,['w','a','s','d']),

```



```

116    validatePathValues(T).

117    validatePath(RowSrc, ColSrc, Path) :-
118        validatePath(RowSrc, ColSrc, Path, true).

119    validatePath(RowSrc, ColSrc, Path, _) :-
120        %verify if it ends inside the board
121        getDestCellFromPath(RowSrc, ColSrc, Path, RowDest, ColDest),
122        RowDest < 9, RowDest >= 0,
123        ColDest < 9, ColDest >= 0,

124    %verify if it turns just once
125    verifyTurnsOnce(Path).

126    validatePath(_, _, _, ErrorMessageFlag) :-
127        (ErrorMessageFlag -> (
128            a. write('That path is not valid!'), nl,
129            b. write('Please, try another path.'), nl
130            ); true),
131        fail.

130    %go_back to repeat cycle

131    validateMove(Game, RowSrc, ColSrc, Path, PieceCaptured) :-
132        validateMove(Game, RowSrc, ColSrc, Path, PieceCaptured, true).

133    validateMove(Game, RowSrc, ColSrc, Path, PieceCaptured, ErrorMessageFlag) :-
134        getCell(Game, RowSrc, ColSrc, Piece),
135        (
136            a. isShortMove(Piece, Path) -> IsFullMove is 0;
137            b. isFullMove(Piece, Path) -> IsFullMove is 1;

138            c. (ErrorMessageFlag -> write('Invalid number of movimentations.\nPlease introduce
another move\n\n'); true),
139            d. fail
140        ),

141    validateCrossMovements(Game, RowSrc, ColSrc, Path, IsFullMove, Piece,
142      PieceCaptured, ErrorMessageFlag) :-
143        getCurrentPlayer(Game, CurrentPlayer),

144        getDestCellFromPath(RowSrc, ColSrc, [LastMove], RowDest, ColDest),
145        getCell(Game, RowDest, ColDest, Piece),

146        ifelse(Piece = 'empty',
147            a. emptyTile(PieceCaptured),
148            b. ifelse(Piece = bg-BgType,
149                i. ifelse(IsFullMove == 1,
150                    ii. ifelse(PieceMoved = _P-2, BgType = 'allDir',
151                        1. (ErrorMessageFlag -> write('You cannot capture an All Directions Barragoon piece

```



```

with a Two Dotted Tile.\n\n'); true), fail),
2. true),
iii. ((ErrorMessageFlag -> write('You cannot capture a piece during a short move.\n\n'); true), fail)),
iv. ifelse(Piece = CurrentPlayer_ ,
v. ((ErrorMessageFlag -> write('You may not capture your own piece.\n\n'); true), fail),
vi. playerPieceCaptured(PieceCaptured))).

144 %validateCrossMovements -> verify cross issues.
145 validateCrossMovements(Game, RowSrc, ColSrc, [FirstMove, SecondMove | PathRest], IsFullMove, PieceMoved, PieceCaptured, ErrorMessageFlag) :-
146     getDestCellFromPath(RowSrc, ColSrc, [FirstMove], RowDest1, ColDest1),
147     getDestCellFromPath(RowDest1, ColDest1, [SecondMove], RowDest2, ColDest2),
148     getCell(Game, RowDest1, ColDest1, Piece1),
149     ifelse(Piece1 = bg-BgType,
150         verifyBarragoonCrossability(RowSrc, ColSrc, RowDest1, ColDest1, RowDest2, ColDest2, BgType, ErrorMessageFlag),
151             a. ifelse(Piece1 = 'empty',
152                 i. true,
153                 ii. ((ErrorMessageFlag -> write('You cannot cross any player piece.\n\n'); true), fail))),
154         validateCrossMovements(Game, RowDest1, ColDest1, [SecondMove | PathRest], IsFullMove, PieceMoved, PieceCaptured, ErrorMessageFlag).

152 % --- Check if it turns just once ---
153 verifyTurnsOnce([H|T]) :-
154     complementary(H,C),
155     verifyTurnsOnceAux(T,H,0,C).
156 verifyTurnsOnceAux([],_,N,_):- N < 2.
157 verifyTurnsOnceAux([H|T], H, N, C) :- verifyTurnsOnceAux(T, H, N, C).
158 verifyTurnsOnceAux([H|T], Z, N, C) :- verifyTurnsOnceAux([H|T], Z, N1, C),
159     N1 is N+1,
160     Z \= H,
161     H \= C,
162     verifyTurnsOnceAux(T, H, N1, C1).
163 complementary(H, C1),
164 N1 is N+1,
165 verifyTurnsOnceAux(T, H, N1, C1).
166 % --- Verify crossability for each barragoon piece ---
167 verifyBarragoonCrossability(RowSrc, ColSrc, RowBg, ColBg, RowDest, ColDest, BgType, ErrorMessageFlag) :-
168     (
169         a. BgType = 'barraX' -> (
170             i. (ErrorMessageFlag -> (nl, write('You cannot cross an X barragoon piece.', nl, nl)));
171             true),
172             ii. fail
173             b. );

```



```

c. BgType = 'allDir' -> (
i.
(
ii. RowSrc \= RowDest,
iii. ColSrc \= ColDest
iv. );

v. (ErrorMessageFlag -> (nl, write('You are crossing an All Direction Barragoon in a
wrong way.'), nl, nl); true),
vi. fail
d. );

e. %One direction
f. BgType = 'oDirU' -> (
i.
(
ii. RowSrc2 is RowSrc-2,
iii. RowSrc2 = RowDest,
iv. ColSrc = ColDest
v. );
vi. (ErrorMessageFlag -> (nl, write('You are crossing a Single Direction Barragoon in a
wrong way.'), nl, nl); true),
vii. fail
g. );
h. BgType = 'oDirD' -> (
i.
(
ii. RowSrc2 is RowSrc+2,
iii. RowSrc2 = RowDest,
iv. ColSrc = ColDest
v. );
vi. (ErrorMessageFlag -> (nl, write('You are crossing a Single Direction Barragoon in a
wrong way.'), nl, nl); true),
vii. fail
i. );
j. BgType = 'oDirR' -> (
i.
(
ii. ColSrc2 is ColSrc+2,
iii. ColSrc2 = ColDest,
iv. RowSrc = RowDest
v. );
vi. (ErrorMessageFlag -> (nl, write('You are crossing a Single Direction Barragoon in a
wrong way.'), nl, nl); true),
vii. fail
k. );
l. BgType = 'oDirL' -> (
i.
(
ii. ColSrc2 is ColSrc-2,
iii. ColSrc2 = ColDest,
iv. RowSrc = RowDest
v. );
vi. (ErrorMessageFlag -> (nl, write('You are crossing a Single Direction Barragoon in a
wrong way.'), nl, nl); true),
vii. fail
m. );

n. %Two directions
o. BgType = 'tDirH' -> (
i.
(
ii. (
1. ColSrc2 is ColSrc-2,
```



```

2. ColSrc2 = ColDest,
3. RowSrc = RowDest
iii. );
iv. (
1. ColSrc2 is ColSrc+2,
2. ColSrc2 = ColDest,
3. RowSrc = RowDest
v. )
vi. );
vii. (ErrorMessageFlag -> (nl, write('You are crossing a Double Direction Barragoon in a
wrong way.'), nl, nl); true),
viii. fail
p. );
q. BgType = 'tDirV' -> (
i. (
ii. (
1. RowSrc2 is RowSrc-2,
2. RowSrc2 = RowDest,
3. ColSrc = ColDest
iii. );
iv. (
1. RowSrc2 is RowSrc+2,
2. RowSrc2 = RowDest,
3. ColSrc = ColDest
v. )
vi. );
vii. (ErrorMessageFlag -> (nl, write('You are crossing a Double Direction Barragoon in a
wrong way.'), nl, nl); true),
viii. fail
r. );

s. %Turn to right
t. BgType = 'DtoR' -> (
i. (
ii. RowSrcU is RowSrc-1,
iii. RowBg = RowSrcU,

iv. ColBgR is ColBg+1,
v. ColDest = ColBgR
vi. );
vii. (ErrorMessageFlag -> (nl, write('You are crossing a Right Turn Barragoon in a wrong
way.'), nl, nl); true),
viii. fail
u. );
v. BgType = 'RtoU' -> (
i. (
ii. ColSrcL is ColSrc-1,
iii. ColBg = ColSrcL,

iv. RowBgU is RowBg-1,
v. RowDest = RowBgU
vi. );
vii. (ErrorMessageFlag -> (nl, write('You are crossing a Right Turn Barragoon in a wrong
way.'), nl, nl); true),
viii. fail
w. );
x. BgType = 'UtoL' -> (
i. (

```



```

ii.    RowSrcD is RowSrc+1,
iii.   RowBg = RowSrcD,

iv.   ColBgL is ColBg-1,
v.    ColDest = ColBgL
vi.   );
vii.  (ErrorMessageFlag -> (nl, write('You are crossing a Right Turn Barragoon in a wrong
way.'), nl, nl); true),
viii. fail
y.    );
z.    BgType = 'LtoD' -> (
i.    (
ii.   ColSrcR is ColSrc+1,
iii.  ColBg = ColSrcR,

iv.   RowBgD is RowBg+1,
v.    RowDest = RowBgD
vi.   );
vii.  (ErrorMessageFlag -> (nl, write('You are crossing a Right Turn Barragoon in a wrong
way.'), nl, nl); true),
viii. fail
aa.   );

bb.   %Turn to left
cc.   BgType = 'DtoL' -> (
i.    (
ii.   RowSrcU is RowSrc-1,
iii.  RowBg = RowSrcU,

iv.   ColBgL is ColBg-1,
v.    ColDest = ColBgL
vi.   );
vii.  (ErrorMessageFlag -> (nl, write('You are crossing a Left Turn Barragoon in a wrong
way.'), nl, nl); true),
viii. fail
dd.   );
ee.   BgType = 'LtoU' -> (
i.    (
ii.   ColSrcR is ColSrc+1,
iii.  ColBg = ColSrcR,

iv.   RowBgU is RowBg-1,
v.    RowDest = RowBgU
vi.   );
vii.  (ErrorMessageFlag -> (nl, write('You are crossing a Left Turn Barragoon in a wrong
way.'), nl, nl); true),
viii. fail
ff.   );
gg.   BgType = 'UtoR' -> (
i.    (
ii.   RowSrcD is RowSrc+1,
iii.  RowBg = RowSrcD,

iv.   ColBgR is ColBg+1,
v.    ColDest = ColBgR
vi.   );
vii.  (ErrorMessageFlag -> (nl, write('You are crossing a Left Turn Barragoon in a wrong
way.'), nl, nl); true),

```



```

viii. fail
    hh. );
    ii. BgType = 'RtoD' -> (
    i.
    ii. ColSrcL is ColSrc-1,
    iii. ColBg = ColSrcL,
    iv. RowBgD is RowBg+1,
    v. RowDest = RowBgD
    vi. );
    vii. (ErrorMessageFlag -> (nl, write('You are crossing a Left Turn Barragoon in a wrong
way.'), nl, nl); true),
    viii. fail
    jj. )
169 ).

170 % -----
171 % ----- CONSEQUENCES -----
172 % -----


173 % --- When a barragoon is captured ---
174 barragoonCaptured(Game, NewGame) :-
175 insertBarragoon(Game, NewGame).

176 % --- When a player piece is captured ---
177 playerPieceCaptured(Game, NewGame) :-
178 switchPlayer(Game, NewGame1),
179 insertBarragoon(NewGame1, FirstBarragoonAddedGame),

180 switchPlayer(FirstBarragoonAddedGame, NewGame2),
181 insertBarragoon(NewGame2, NewGame).

182 % --- Insert new barragoon ---
183 insertBarragoon(Game, NewGame) :-
184 %displayGame(Game),
185 getPlayerType(Game, PlayerType),
186 repeat,
187 (
    a. PlayerType = player -> chooseTile(Row, Column, 'Where do you wish to place your
barragoon?');
    b. PlayerType = bot -> botChooseTile(Game, Row, Column)
188 ),
189 getCell(Game, Row, Column, empty),
190 !,

191 write('Which barragoon do you wish to insert?'),nl,
192 write('The options are:'),nl,
193 write('1- X'),nl,
194 write('2- + (All Directions)'),nl,
195 write('3- => (One Direction)'),nl,
196 write('4- <=> (Two Directions)'),nl,
197 write('5- \\'> (Left Turn)'),nl,
198 write('6- <\\' (Right Turn)'),nl,
199 getPlayerType(Game, PlayerType),

```



```

200    (
201      a. PlayerType = player -> getCharThenEnter(Option);
202      b. PlayerType = bot -> (
203          i. random(1,7, OptionInt),
204          ii. numbersAxis(Numbers),
205          iii. nth1(OptionInt, Numbers, Option)
206      )
207  ),
208  (
209    Option = '1' -> setCell(Game, Row, Column, bg-'barraX', NewGame);
210    Option = '2' -> setCell(Game, Row, Column, bg-'allDir', NewGame);
211    Option = '3' -> insertOneDirectionBg(Game, Row, Column, NewGame);
212    Option = '4' -> insertTwoDirectionsBg(Game, Row, Column, NewGame);
213    Option = '5' -> insertTurnToTheLeft(Game, Row, Column, NewGame);
214    Option = '6' -> insertTurnToTheRight(Game, Row, Column, NewGame)
215  ).
216

217  insertOneDirectionBg(Game, Row, Column, NewGame):-  

218    nl, write('In which diretion you want to set the barragoon?'),nl,  

219    write('The options are:'),nl,  

220    write('1- A (Up)'),nl,  

221    write('2- V (Down)'),nl,  

222    write('3- => (Right)'),nl,  

223    write('4- <= (Left)'),nl,  

224
225    getPlayerType(Game, PlayerType),  

226    (
227      a. PlayerType = player -> getCharThenEnter(Option);
228      b. PlayerType = bot -> (
229          i. random(1,5, OptionInt),
230          ii. numbersAxis(Numbers),
231          iii. nth1(OptionInt, Numbers, Option)
232      )
233  ),
234  (
235    a. Option = '1' -> setCell(Game, Row, Column, bg-'oDirU', NewGame);
236    b. Option = '2' -> setCell(Game, Row, Column, bg-'oDirD', NewGame);
237    c. Option = '3' -> setCell(Game, Row, Column, bg-'oDirR', NewGame);
238    d. Option = '4' -> setCell(Game, Row, Column, bg-'oDirL', NewGame)
239  ).  

240
241  insertTwoDirectionsBg(Game, Row, Column, NewGame):-  

242    nl, write('In which diretion you want to set the barragoon?'),nl,  

243    write('The options are:'),nl,  

244    write('1- <> (Horizontal)'),nl,  

245    write('2- | (Vertical)'),nl,  

246    getPlayerType(Game, PlayerType),  

247    (
248      a. PlayerType = player -> getCharThenEnter(Option);
249      b. PlayerType = bot -> (
250          i. random(1,3, OptionInt),
251          ii. numbersAxis(Numbers),
252          iii. nth1(OptionInt, Numbers, Option)
253      )
254  ),
255  (
256    a. Option = '1' -> setCell(Game, Row, Column, bg-'tDirH', NewGame);
257    b. Option = '2' -> setCell(Game, Row, Column, bg-'tDirV', NewGame)
258  ).  

259

```



```

231     ).
232     insertTurnToTheLeft(Game, Row, Column, NewGame):-  

233     nl, write('In which direction you want to set the barragoon?'),nl,  

234     write('The options are:'),nl,  

235     write('1- < (Down to Left)'),nl,  

236     write('2- -^ (Left to Up)'),nl,  

237     write('3- \'> (Up to Right)'),nl,  

238     write('4- v- (Right to Down)'),nl,  

239     getPlayerType(Game, PlayerType),  

240     (  

241         a. PlayerType = player -> getCharThenEnter(Option);  

242         b. PlayerType = bot -> (  

243             i. random(1,5, OptionInt),  

244                 numbersAxis(Numbers),  

245                 nth1(OptionInt, Numbers, Option)  

246             c. )  

247         ),  

248     (  

249         a. Option = '1' -> setCell(Game, Row, Column, bg-'DtoL', NewGame);  

250         b. Option = '2' -> setCell(Game, Row, Column, bg-'LtoU', NewGame);  

251         c. Option = '3' -> setCell(Game, Row, Column, bg-'UtoR', NewGame);  

252         d. Option = '4' -> setCell(Game, Row, Column, bg-'RtoD', NewGame)  

253     ).  

254     insertTurnToTheRight(Game, Row, Column, NewGame):-  

255     nl, write('In which direction you want to set the barragoon?'),nl,  

256     write('The options are:'),nl,  

257     write('1- .> (Down to Right)'),nl,  

258     write('2- ^- (Right to Up)'),nl,  

259     write('3- <\'' (Up to Left)'),nl,  

260     write('4- -v (Left to Down)'),nl,  

261     getPlayerType(Game, PlayerType),  

262     (  

263         a. PlayerType = player -> getCharThenEnter(Option);  

264         b. PlayerType = bot -> (  

265             i. random(1,5, OptionInt),  

266                 numbersAxis(Numbers),  

267                 nth1(OptionInt, Numbers, Option)  

268             c. )  

269         ),  

270     (  

271         a. Option = '1' -> setCell(Game, Row, Column, bg-'DtoR', NewGame);  

272         b. Option = '2' -> setCell(Game, Row, Column, bg-'RtoU', NewGame);  

273         c. Option = '3' -> setCell(Game, Row, Column, bg-'UtoL', NewGame);  

274         d. Option = '4' -> setCell(Game, Row, Column, bg-'LtoD', NewGame)  

275     ).  

276     % --- Valid coordinates ---  

277     validColumns(['a','b','c','d','e','f','g','A','B','C','D','E','F','G']).  

278     validRow(Y):- Y > 48, Y < 60.  

279     validBarragoons([bg-barraX, bg-allDir, bg-oDirU, bg-oDirD, bg-oDirL, bg-oDirR, bg-oDirH, bg-oDirV, bg-left, bg-right]).  

280     complementary('w','s').

```



```

261    complementary('s','w').
262    complementary('a','d').
263    complementary('d','a').

264    barragoonCaptured(barragoonPiece).
265    playerPieceCaptured(playerPiece).
266    emptyTile(empty).

267    % --- Get Player Pieces ---

268    getPlayerPieces(Game, List) :-
269        getCurrentPlayer(Game, Player),
270        getPlayerPiecesAux(Game, Player, 0, 0, List).

271    getPlayerPiecesAux(_, _, 9, _, []).
272    getPlayerPiecesAux(Game, CurrentPlayer, Row, 7, List) :-
273        Row1 is Row+1,
274        getPlayerPiecesAux(Game, CurrentPlayer, Row1, 0, List).
275    getPlayerPiecesAux(Game, CurrentPlayer, Row, Column, List) :-  

276        Column1 is Column+1,  

277        ifelse( getCell(Game, Row, Column, CurrentPlayer-_),  

278            a.  

279            % --- Count player pieces ---
280            getPlayerPiecesAux(Game, CurrentPlayer, Row, Column1, List1),  

281            List = [[Row, Column] | List1]  

282            b.,  

283            getPlayerPiecesAux(Game, CurrentPlayer, Row, Column1, List)  

284            ).  

285  

286    countPlayerPieces([], _, 0).
287    countPlayerPieces([[] | BoardRest], CurrentPlayer, Count) :-  

288        countPlayerPieces(BoardRest, CurrentPlayer, Count).
289  

290    countPlayerPieces([[CurrentPlayer-_| RowRest] | BoardRest], CurrentPlayer, Count) :-  

291        countPlayerPieces([RowRest | BoardRest], CurrentPlayer, N),
292        Count is N+1 .
293  

294    countPlayerPieces([[_Piece | RowRest] | BoardRest], CurrentPlayer, Count) :-  

295        countPlayerPieces([RowRest | BoardRest], CurrentPlayer, Count).

296    % --- Count Moves Available ---

297    countMovesAvailable(Game, Row, Column, Count) :-
298        getMovesAvailable(Game, Row, Column, List),
299        length(List, Count).

300  

301    getMovesAvailable(Game, Row, Column, List) :-
302        getCell(Game, Row, Column, _Player-NDots),

```



```

294     availableMoves(NDots, AllMovesAvailable),
295     getMovesAvailableAux(Game, Row, Column, AllMovesAvailable, List).
296     getMovesAvailableAux(_, _, _, [], []).
297     getMovesAvailableAux(Game, Row, Column, [Path | Tail], List) :-
298       ifelse( validatePath(Row, Column, Path, false), validateMove(Game, Row, Column,
299       Path, PieceCaptured, false)),
300       a. (
301         i. getMovesAvailableAux(Game, Row, Column, Tail, List1),
302         ii. ifelse( PieceCaptured = playerPiece,
303           List = [Path | List1],
304           append(List1, [Path], List)
305         v. )
306       b. ),
307       c. (
308         i. getMovesAvailableAux(Game, Row, Column, Tail, List)
309       d. )
309     ).  

300 % --- Count Moves Available for All Pieces of Player ---
301 countMovesAvailableAllPieces(Game, Count) :-
302   getPlayerPieces(Game, PlayerPieces),
303   countMovesAvailableAllPiecesAux(Game, PlayerPieces, Count).
304   countMovesAvailableAllPiecesAux(_, [], 0).
305   countMovesAvailableAllPiecesAux(Game, [[Row, Column] | RemainingPlayerPieces],
306   Count) :-
306     countMovesAvailable(Game, Row, Column, Count1),
307     countMovesAvailableAllPiecesAux(Game, RemainingPlayerPieces, Count2),
308     Count is Count1+Count2.

```

Utilities.pl

```

1  %-----%
2  %----Useful functions----%
3  %-----%
4
5  %
6  % ----- VISUALIZATION -----
7  %
8
9  % -- Visualization --
10 spacing(Lines) :-
11   spacing(0, Lines).
12   spacing(Line, Limit) :-
13     Line < Limit,
14     LineInc is Line + 1,
15     nl,
16     spacing(LineInc, Limit).
17   spacing(_, _).
18
19 clearScreen :- spacing(50), !.

```



```

20      % -----
21      % ----- RECEIVE INPUT -----
22      % -----
23
24      % --- Receive input ---
25      next :-  
        write('Press any key to continue...'),  
        get_code(_), clearScreen.  
28
29      getCharThenEnter(X) :-  
30          get_char(X),  
31          get_char(_, !).  
32
33      getCodeThenEnter(X) :-  
34          get_code(X),  
35          get_char(_, !).  
36
37      % --- Get columns and rows ---
38      readCharFromUser(X, Possibilities, Message, _Error) :-  
39          write(Message),  
40          getCharThenEnter(X),  
41          member(X, Possibilities).  
42      readCharFromUser(X, Possibilities, Message, Error) :-  
43          !,  
44          write(Error),  
45          readCharFromUser(X, Possibilities, Message, Error).  
46
47      readCodeFromUser(X, Condition, Message, _Error) :-  
48          write(Message),  
49          getCodeThenEnter(X),  
50          Condition.  
51      readCodeFromUser(X, Condition, Message, Error) :-  
52          !,  
53          write(Error),  
54          readCodeFromUser(X, Condition, Message, Error).  
55
56      % --- Get position ---
57      getPositionFromUser(Row, Column) :-  
58          readCodeFromUser(Row, validRow(Row), 'Row: ', 'Invalid Row! Rows must be  
between 1 and 9.\n'),  
59          !,  
60          validColumns(ValidCols),  
61          readCharFromUser(Column, ValidCols, 'Column: ', 'Invalid Column! Columns must be  
between A and G.\n'),  
62          !.  
63
64      chooseTile(Row, Column, Message) :-  
65          write(Message), nl,  
66          getPositionFromUser(RowCode, ColumnChar),  
67          validColumns(Possibilities),  
68
69          nth0(ColPosInPossibilities, Possibilities, ColumnChar),  
70          Column is ColPosInPossibilities mod 7,  
71          Row is RowCode-49.  
72
73      choosePath(Path, Message) :-  
74          nl, write(Message), nl,  
75          readCharUntilEnter(Path),

```



```

76 validatePathValues(Path).
77
78
79 choosePath(_,_) :-
80 nl, write('Wrong path values. Please try again.'), nl,
81 fail.
82
83 botChooseTile(Game, Row, Column) :-
84 getEmptyPlaces(Game, EmptyPlaces),
85 getRandomElemFromList(EmptyPlaces, [Row, Column]).
86
87 % --- Get path ---
88 readCharUntilEnter(List) :-
89 get_char(Char),
90 readCharUntilEnterAux([Char|Tail], Char),
91 List = [Char|Tail],
92 !.
93
94 readCharUntilEnterAux([], '\n').
95 readCharUntilEnterAux([Char|Tail], Char) :-
96 get_char(NewChar),
97 readCharUntilEnterAux(Tail, NewChar).
98
99 % -----
100 % ----- CONDITIONS -----
101 % -----
102
103 % --- operators ---
104 ifelse(Condition, If, _Else) :- Condition, !, If.
105 ifelse(_, _, Else) :- Else.
106
107 not(X) :- X, !, fail.
108 not(_X).
109
110 % -----
111 % ----- READ BOARD -----
112 % -----
113
114 % --- Get cell value ---
115 getCell(Game, Row, Column, Value) :-
116
117 getBoard(Game, Board),
118 nth0(Row, Board, RowList),
119 nth0(Column, RowList, Value).
120
121 % --- Get board ---
122 getBoard(Game, GameBoard) :-
123 nth0(0, Game, GameBoard).
124
125 % --- Get player ---
126 getCurrentPlayer(Game, CurrentPlayer) :-
127 nth0(1, Game, CurrentPlayer).
128
129 % --- Get game mode ---
130 getMode(Game, Mode) :-
131 nth0(2, Game, Mode).

```



```

132    % --- Get Difficulty ---
133    getDifficulty(Game, Difficulty) :-
134        nth0(2, Game, _-Difficulty).
135
136    % -- Get Empty Cells
137    getEmptyPlaces(Game, List) :-
138        getEmptyPlacesAux(Game, 0, 0, List).
139
140        getEmptyPlacesAux(_, 9, []).
141        getEmptyPlacesAux(Game, Row, 7, List) :-
142            Row1 is Row+1,
143            getEmptyPlacesAux(Game, Row1, 0, List).
144            getEmptyPlacesAux(Game, Row, Column, List) :-
145                Column1 is Column+1,
146
147                ifelse(getCell(Game, Row, Column, empty),
148                (
149                    getEmptyPlacesAux(Game, Row, Column1, List1),
150                    List = [[Row, Column] | List1]
151                ),
152                    getEmptyPlacesAux(Game, Row, Column1, List)
153                ).
154
155
156    getPlayerType(Game, PlayerType) :-
157
158        getCurrentPlayer(Game, CurrentPlayer),
159        getMode(Game, Mode),
160
161        (
162            Mode = pvp -> PlayerType = player;
163            Mode = pvc_- -> (
164                CurrentPlayer = w -> PlayerType = player;
165                CurrentPlayer = b -> PlayerType = bot
166            );
167            Mode = cvc_- -> PlayerType = bot
168        ).
169
170    % -----
171    % ----- CHANGE BOARD -----
172    % -----
173
174    % --- Replace ---
175    replaceInList([_|T], 0, X, [X|T]).
176    replaceInList([H|T], I, X, [H|R]) :- I > -1, NI is I-1, replaceInList(T, NI, X, R), !.
177    replaceInList(L, _, _, L).
178
179    % --- Get random element from a List ---
180    getRandomElemFromList(List, Value) :-
181        length(List, ListLength),
182
183        random(0,ListLength, ElemlIndex),
184
185        nth0(ElemlIndex, List, Value).
186
187    % --- Clear a cell ---
188    clearCell(Game, NRow, NColumn, Value, NewGame) :-

```



```

190    getBoard(Game, Board),
191    /*Get Value*/
192    nth0(NRow, Board, Row),
193    nth0(ColumnName, Row, Value),
194    /*Clear cell*/
195    replaceInList(Row, NColumn, empty, NewRow),
196    replaceInList(Board, NRow, NewRow, NewBoard),
197    setBoard(Game, NewBoard, NewGame).
198
199    % --- Change cell value ---
200    setCell(Game, NRow, NColumn, Value, NewGame) :-
201
202    getBoard(Game, Board),
203
204    nth0(NRow, Board, Row),
205    replaceInList(Row, NColumn, Value, NewRow),
206    replaceInList(Board, NRow, NewRow, NewBoard),
207
208    setBoard(Game, NewBoard, NewGame).
209
210    % --- Change board ---
211    setBoard(Game, NewBoard, NewGame) :-
212    replaceInList(Game, 0, NewBoard, NewGame).
213
214    setCurrentPlayer(Game, NewPlayer, NewGame) :-
215    replaceInList(Game, 1, NewPlayer, NewGame).
216
217    setMode(Game, NewMode, NewGame) :-
218    replaceInList(Game, 2, NewMode, NewGame).
219
220    % --- Change player ---
221    switchPlayer(Game, NextPlayerGame) :-
222    getBoard(Game, Board),
223    getCurrentPlayer(Game, CurrentPlayer),
224    getMode(Game, Mode),
225    ifelse( CurrentPlayer == w ,
226           NextPlayer = b,
227           NextPlayer = w),
228    NextPlayerGame = [Board, NextPlayer, Mode].
229
230    availableMoves(2, [    %fullMove
231    ['w','w'], ['s','s'], ['d','d'], ['a','a'],
232    ['w','d'], ['d','w'],
233    ['d','s'], ['s','d'],
234    ['s','a'], ['a','s'],
235    ['a','w'], ['w','a'],
236    %shortMove
237    ['w'], ['d'], ['a'], ['s']
238    ]).
239
240    availableMoves(3, [    %fullMove
241    ['w','w','w'], ['s','s','s'], ['d','d','d'], ['a','a','a'],
242    ['w','w','d'], ['d','w','w'],
243    ['w','d','d'], ['d','d','w'],
244    ['d','d','s'], ['s','d','d'],
245    ['d','s','s'], ['s','s','d'],
246    ['s','s','a'], ['a','s','s'],

```



```

247      ['s','a','a'], ['a','a','s'],
248      ['a','a','w'], ['w','a','a'],
249      ['a','w','w'], ['w','w','a'],
250      %shortMove
251      ['w','w'], ['s','s'], ['d','d'], ['a','a'],
252      ['w','d'], ['d','w'],
253      ['d','s'], ['s','d'],
254      ['s','a'], ['a','s'],
255      ['a','w'], ['w','a']
256      ]).
257      availableMoves(4, [    %fullMove
258      ['w','w','w','w'], ['s','s','s','s'],
259      ['d','d','d','d'], ['a','a','a','a'],
260      ['w','w','w','d'], ['d','w','w','w'],
261      ['w','w','d','d'], ['d','d','w','w'],
262      ['w','d','d','d'], ['d','d','d','w'],
263      ['d','d','d','s'], ['s','d','d','d'],
264      ['d','d','s','s'], ['s','s','d','d'],
265      ['d','s','s','s'], ['s','s','s','d'],
266      ['s','s','s','a'], ['a','s','s','s'],
267      ['s','s','a','a'], ['a','a','s','s'],
268      ['s','a','a','a'], ['a','a','a','s'],
269      ['a','a','a','w'], ['w','a','a','a'],
270      ['a','a','w','w'], ['w','w','a','a'],
271      ['a','w','w','w'], ['w','w','w','a'],
272      %shortMove
273      ['w','w','w'], ['s','s','s'], ['d','d','d'], ['a','a','a'],
274      ['w','w','d'], ['d','w','w'],
275      ['w','d','d'], ['d','d','w'],
276      ['d','d','s'], ['s','d','d'],
277      ['d','s','s'], ['s','s','d'],
278      ['s','s','a'], ['a','s','s'],
279      ['s','a','a'], ['a','a','s'],
280      ['a','a','w'], ['w','a','a'],
281      ['a','w','w'], ['w','w','a']
282      ]).

```