# *Crab Stack*

**Relatório Final**


Mestrado Integrado em Engenharia Informática e Computação

Programação em Lógica

**Grupo Crab_Stack_1:**
Inês de Sousa Caldas - up200904082
Maria Teresa Chaves - up201306842

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

14 de Novembro de 2016

# Resumo

No âmbito da unidade curricular de Programação em Lógica foi desenvolvido o jogo *Crab Stack* em ambiente de *Sicstus Prolog*. Neste jogo, cada jogador tem um conjunto de caranguejos que pode mover no tabuleiro de acordo com um conjunto de regras. O objetivo é fazer com que o adversário fique sem jogadas possíveis.

Para implementar o jogo dividiu-se o jogo nas suas diversas fases. O desenvolvimento da aplicação consistiu na representação de um tabuleiro hexagonal, composto por células que representam rochas, onde os caranguejos estão posicionados, podendo estes apenas mover-se para cima de rochas habitadas por outros caranguejos. Implementou-se inicialmente um menu de forma a que fosse intuitivo para o utilizador jogar o *Crab Stack* em modo texto. Depois partiu-se para a implementação da mecânica do jogo, começando pelo início do jogo em que as peças são distribuídas pelo tabuleiro de forma aleatória. Posteriormente, implementou-se o movimento de uma peça no tabuleiro, verificando sempre se após o movimento, este provocava uma onda (descrita nas regras de jogo). Por último, foi implementado o final do jogo para verificar se um dos jogadores teria ganho.

Assim, o resultado do presente trabalho é uma aplicação que representa e implementa o jogo *Crab Stack* e que mostra ao utilizador uma interface em modo texto intuitiva. O jogo permite também escolher o modo de jogo que inclui Humano vs Humano, Humano vs Máquina ou mesmo Máquina vs Máquina.

# Conteúdo

# 1 Introdução

Este trabalho teve como objetivo a implementação, em linguagem Prolog, de um jogo de tabuleiro para dois jogadores. Um jogo de tabuleiro caracteriza-se pelo tipo de tabuleiro e de peças, pelas regras de movimentação das peças (jogadas possíveis) e pelas condições de terminação do jogo com derrota, vitória ou empate. Também teve como objetivo a implementação da aplicação de forma a que fosse possível jogar em três modos de utilização: Humano vs Humano, Humano vs Computador e Computador vs Computador. No caso de algum dos modos escolhidos incluir o computador, é possível escolher qual o nível de dificuldade do mesmo. Por último, teve também como objetivo a implementação de uma interface adequada com o utilizador, em modo de texto. [1]

Este trabalho, teve como motivação uma aprendizagem didática da linguagem Prolog e um aprofundamento e maturação dos conhecimentos transmitidos na unidade curricular de Programação em Lógica. Outra grande motivação, foi a aprendizagem de conceitos de inteligência artificial em ambiente de jogo, como os algoritmos de *minimax* e *alphabeta*.

O presente relatório, para além da introdução, encontra-se dividido em quatro outras grandes secções:

- **O Jogo *Crab Stack*** - descrição do jogo, da sua história e das suas regras;
- **Lógica do Jogo** - descrição do projeto e da implementação da lógica do jogo em Prolog, que inclui a forma de representação do estado do tabuleiro e sua visualização, execução de movimentos, verificação do cumprimento das regras do jogo, determinação do final do jogo e cálculo das jogadas a realizar pelo computador utilizando diversos níveis de jogo;
- **Interface com o Utilizador** - descrição do módulo de interface com o utilizador em modo de texto;
- **Conclusões**.

# 2 O Jogo *Crab Stack*

O jogo *Crab Stack* foi publicado pela *Blue Orange Games* em 2015. Este foi conceptualizado por *Henri Kermarrec* que contou com a colaboração da artista *Stéphane Escapa*. Enquadra-se nas categorias de jogos abstratos e familiares, sendo aconselhado a jogadores com idade superior a 8 anos. [2]

Este jogo pode ser jogado por dois a quatro jogadores. Dependendo do número de jogadores apenas uma parte do tabuleiro é utilizado. No caso de serem dois jogadores utiliza-se apenas as rochas amarelas (figura 1); no caso de serem três jogadores, para além das rochas amarelas também são utilizadas as rochas pretas; e por último no caso de serem quatro jogadores, joga-se em todo o tabuleiro. [4] No contexto da unidade curricular de Programação em Lógica, é pretendido o desenvolvimento de um jogo de tabuleiro para dois jogadores. Desta forma apenas serão enunciadas as regras [3] relacionadas com o modo de dois jogadores.

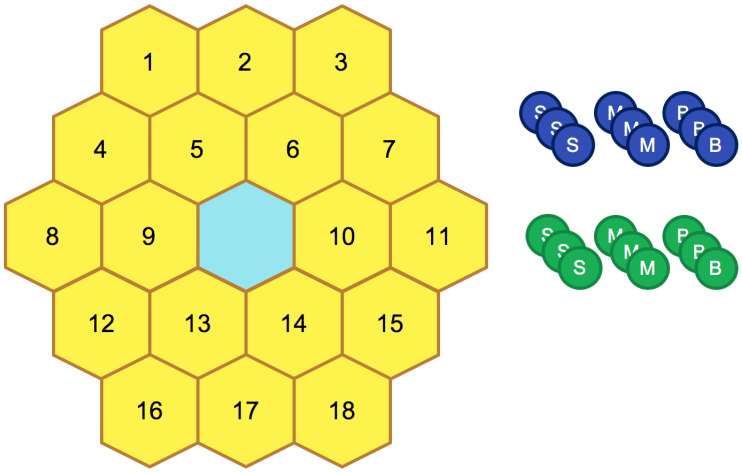



Figura 1: Tabuleiro de jogo

## 2.1 Preparação do tabuleiro

Com um formato hexagonal, o tabuleiro é formado por 18 posições, como é possível observar na figura 1 nas rochas amarelas.
A cada jogador é atribuído um conjunto de 9 caranguejos: 3 grandes, 3 médios e 3 pequenos, como é possível observar no lado direito da figura 1.
Antes do jogo começar, os caranguejos dos jogadores são distribuídos aleatoriamente pelo tabuleiro obedecendo às regras de empilhamento que serão explicitadas na secção 2.2.

## 2.2 Movimentos possíveis

No turno de um jogador, este pode mover um dos seus caranguejos de forma a terminar o seu movimento numa rocha ocupada por outro caranguejo. Dependendo do seu tamanho, um caranguejo move-se, obrigatoriamente, um determinado número de rochas:

- Pequeno - move-se três rochas;
- Médio - move-se duas rochas;
- Grande - move-se uma rocha.

No final do movimento, um caranguejo não pode regressar à rocha inicial.
Apenas os caranguejos que estejam no topo da pilha podem ser movidos durante o turno de um jogador.
O empilhamento dos caranguejos tem de obedecer às seguintes regras:

- um grande pode ficar em cima de qualquer outro caranguejo;
- um médio pode ficar em cima de um caranguejo médio ou pequeno;
- um pequeno pode apenas ficar em cima de outro caranguejo pequeno.

## 2.3 Regra da Onda

Os caranguejos gostam de estar em grupos grandes e não gostam de ser separados. Numa situação em que os caranguejos fiquem separados em dois grupos, por uma linha de rochas vazias, um destes grupos irá ser eliminado do jogo por uma onda. O grupo a ser eliminado é selecionado pela seguinte ordem de prioridades:

1. O grupo de caranguejos que ocupar menos espaço no tabuleiro é removido do jogo;
2. Se os dois grupos ocupam o mesmo número de casas no tabuleiro, então o grupo com o menor número total de caranguejos é removido do jogo;
3. Se o número de caranguejos for o mesmo, o jogador do turno em que ocorreu a separação, decide qual o grupo que irá ser removido do jogo.

## 2.4 Fim do jogo

O *Crab Stack* pode terminar nos seguintes estados do jogo:

- Se todas as peças de um jogador forem removidas do jogo, através de uma onda, o jogador perde o jogo;
- Se um jogador, no inicio do seu turno, não conseguir mover nenhum dos seus caranguejos, perde o jogo.

# 3 Lógica do Jogo

Nesta secção é feita uma descrição do projeto e da implementação da lógica do jogo em Prolog, incluindo a forma de representação do estado do tabuleiro e sua visualização, execução de movimentos, verificação do cumprimento das regras do jogo, determinação do final do jogo e cálculo das jogadas a realizar pelo computador utilizando diversos níveis de dificuldade de jogo.

## 3.1 Representação do Estado do Jogo

### 3.1.1 Representação do estado do tabuleiro

O tabuleiro é representado por uma lista de listas, em que cada elemento da lista representa uma rocha. Os caranguejos de cada jogador serão representados por Sx, Mx, Bx, em que x representa o número do jogador e S, M, B o tamanho do caranguejo, respetivamente pequeno, médio e grande. Uma rocha, i.e. uma célula da lista, tem dois estados diferentes possíveis: estar vazia ou ter uma pilha de caranguejos. Uma pilha de caranguejos, pode ser por exemplo [B1, S1, S2], em que o elemento B1 representa o topo da pilha.
O código para a criação de um tabuleiro é dado por:

```
% Initializes Board
init_board(Board):-
        Empty_Board = [ [[],[],[]],
                        [[],[],[],[]],
                        [[],[],[],[]],
                        [[],[],[],[]],
                        [[],[],[]]
         ],
        crabs(CrabsList),
        add_crab_init_board(Empty_Board, CrabsList, Board).

% Adds the crabs to empty board cells
add_crab_init_board(Board, [], Board).

add_crab_init_board(Board, [HCrabs | TCrabs], FinalBoard):-
        aux_board(Board, HCrabs, Tmp_FinalBoard),
        add_crab_init_board(Tmp_FinalBoard, TCrabs, FinalBoard),!.

% Auxiliary function to put a crab in an empty cell with random
aux_board(Board, Crab, FinalBoard):-
        repeat,
        random(1, 19, Rock),
        \+ (get_rock(Rock, Board, _)),
        add_crab_board(Board, Rock, Crab, FinalBoard).
```

### 3.1.2 Posições iniciais do jogo

No início do jogo, as peças de cada jogador são colocadas aleatoriamente no tabuleiro, garantindo que fica com todas as rochas ocupadas. Abaixo segue-se o predicado que inicia o jogo e um exemplo de um tabuleiro no estado inicial do jogo.

```
Board =  [ %initial board example
           [['S1'],['M2'],['M1']],
           [['B2'],['B1'],['M1'],['S2']],
           [['S2'],['S1'],['M2'],['B2']],
           [['M2'],['B2'],['S1'],['B1']],
           [['M1'],['S2'],['B1']]]
```

Para uma melhor visualização do estado inicial representado acima, segue-se uma figura ilustrativa.

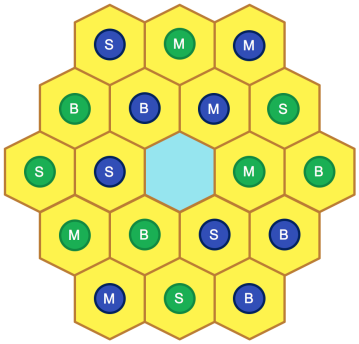


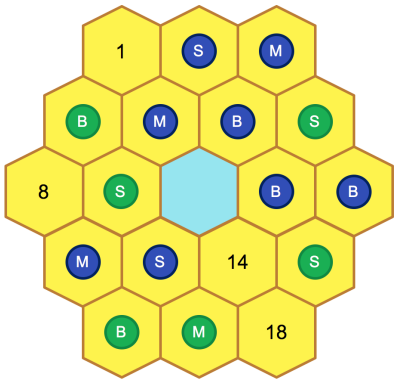

Figura 2: Estado inicial do jogo

### 3.1.3 Posições intermédias do jogo

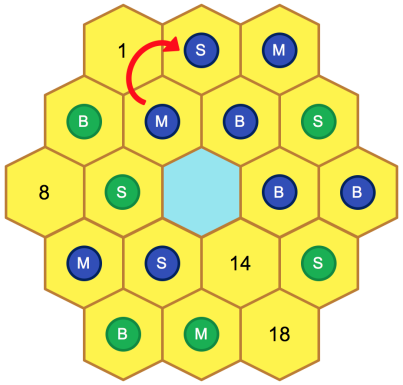
Abaixo seguem-se exemplos de tabuleiros num estado intermédio do jogo.

```
Board = [%General Intermediate state board
        [['M1','M2','S1'],[],[]],
        [[],['B1'],['M1'],['S2']],
        [['B2','S2'],['S1'],['M2'],['B1','B2']],
        [['B2','M2'],[],['B1','S1'],[]],
        [[],['M1','S2'],[]]]

Board = [% Intermediate state board With a possibility for a wave
        [[],['S1'],['M1']],
        [['B2'],['M1'],['B1'],['S2']],
        [[],['S2'],['B1','M2'],['B1','B2']],
        [['M1','M2'],['S1'],[],['S2','S1']],
        [['B2'],['M2'],[]]]

Board = [% Intermediate state board ready for the Wave
        [[],['M1','S1'],['M1']],
        [['B2'],[],['B1'],['S2']],
        [[],['S2'],['B1','M2'],['B1','B2']],
        [['M1','M2'],['S1'],[],['S2','S1']],
        [['B2'],['M2'],[]]]

Board = [% Intermediate state board after the Wave
        [[],['M1','S1'],['M1']],
        [[],[],['B1'],['S2']],
        [[],[],['B1','M2'],['B1','B2']],
        [[],[],[],['S2','S1']],
        [[],[],[]]]
```

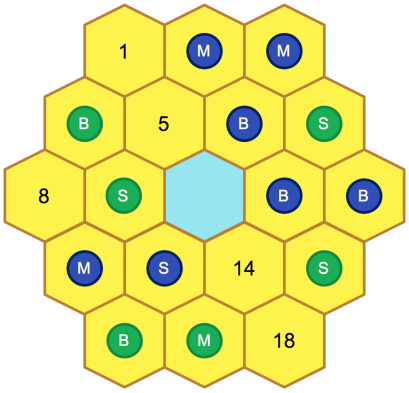
Para uma melhor visualização da regra da onda, seguem-se imagens ilustrativas do exemplo apresentado acima, com os diferentes estados intermédios.



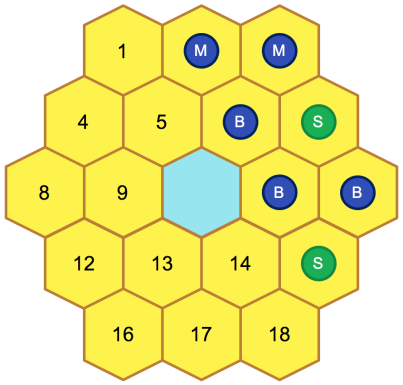

(a) Possibilidade de ocorrer uma onda.




(b) Jogador azul move peça para ocorrer uma onda.




(c) Ocorre uma onda.




(d) Peças do grupo esquerdo são removidas do jogo.


(e) O jogador azul ganha o jogo.

Figura 3: Exemplo da ocorrência de uma onda durante o turno do jogador azul.

### 3.1.4 Posições finais do jogo

No exemplo da onda, mencionada na secção anterior, verifica-se que após a onda, o jogador verde apenas pode mover um dos seus caranguejos pequenos para cima do seu outro caranguejo pequeno, deixando de ter movimentos possíveis, e portanto o jogador azul ganha o jogo.

```
Board = [% Green's turn after the wave
        [[],['M1','S1'],['M1']],
        [[],[],['B1'],['S2']],
        [[],[],['B1','M2'],['B1','B2']],
        [[],[],[],['S2','S1']],
        [[],[],[]]]

Board = [% Final state board after the green's move
        [[],['M1','S1'],['M1']],
        [[],[],['B1'],['S2','S2']],
        [[],[],['B1','M2'],['B1','B2']],
        [[],[],[],['S1']],
        [[],[],[]]]
```

## 3.2 Visualização do Tabuleiro

Para a visualização do tabuleiro, irão ser implementados os seguintes predicados:

- Cabeçalho do predicado para a visualização do tabuleiro:

  ```
  display_board(+Board).
  ```

- Cabeçalho do predicado para a visualização da pilha:

  ```
  display_stack(+Rock, +Stack).
  ```

Segue-se a visualização do output do predicado *display_board* para o primeiro exemplo de um estado intermédio de jogo apresentado na secção 3.1.3.

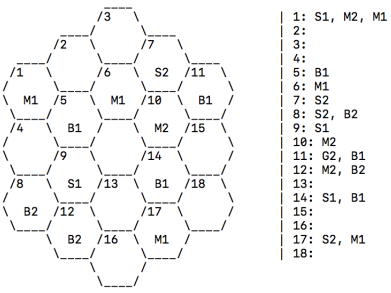



Figura 4: Representação do output no estado intermédio do jogo.

## 3.3 Lista de Jogadas Válidas

Para a obtenção de uma lista de jogadas possíveis implementou-se o seguinte predicado `check_moves(+Board, +Init_Pos, +Init_Crab_Size, +Final_Pos, +Player_Color, +Moves, +ListMoves, -Moves_Rock, -List_Moves_Rock)` que, de forma recursiva, determina se os movimentos são válidos para um dado jogador e para uma dada posição inicial e posição final do tabuleiro. No final, é obtida uma listagem com os movimentos possíveis para uma mesma posição inicial, por exemplo, List_Moves_Rock = [[1,2], [1,4]], ou seja, a partir da posição 1 é possível mover um caranguejo para as posições 2 ou 4. O predicado `moves_left(+Board, +Player_Color, +Moves, +List_Moves_Counter, +Count, -Final_Moves, -List_Final_Moves)`, chama o `check_moves` para as várias posições iniciais e cria uma lista com todas as jogadas possíveis em `List_Final_Moves`.

## 3.4 Execução de Jogadas

O predicado `make_move(+Board, +Player, +Player_Color, -FinalBoard)`, permite ao jogador realizar um movimento. Este predicado faz uma chamada a `choose_tile(+Board, +Player, +Player_Color, -FinalBoard)`, que dependendo se o `Player` for *human* ou *computer*, questiona ao jogador a posição do caranguejo a mover e a sua posição final, ou determina as posições inicial e final com base na inteligência artificial implementada, respetivamente.

Os predicados seguintes, foram implementados para validar a jogada introduzida por um jogador *human*. O predicado `legalMove(+Board, +Rock_Init, +Rock, +Player_Color, -FinalBoard)` faz as verificações gerais sobre a legalidade do movimento, tal como, se as posições introduzidas estão dentro do intervalo de rochas do tabuleiro (numeradas de 1 a 18), se existe um caranguejo nessa rocha e se este pertence ao jogador. Para a validação das regras de jogo, é invocado o predicado `valid_crab_movement(+Board, +Rock_Init, +Rock, +Crab, +Crab_Size, -FinalBoard)`. Este predicado verifica se a deslocação do caranguejo é válida de acordo com as regras de movimento de um caranguejo dependendo do seu tamanho e se na posição final existe outra caranguejo, dado que estes não se podem deslocar para rochas vazias. Finalmente, se a jogada passar por todos os testes de validação é chamado o predicado `update_board(+Board, +Rock, +Rock_Init, +Crab, -FinalBoard)` que faz a atualização do tabuleiro.

## 3.5 Avaliação do Tabuleiro

O predicado `evaluate_board(+Board, +Player, -Value)` permite avaliar um tabuleiro, para um dado jogador, fazendo uso do algoritmo *alphabeta*. Para o *computer* escolher a sua jogada, é invocado o predicado `evaluate_and_choose(+Board, +Player, +Moves, +Depth, +Alpha, +Beta, +Record, -BestMove)`, que para uma listagem dos movimentos possíveis para um jogador, determina a melhor jogada, a partir da avaliação feita no `evaluate_board`. Para avaliar um estado de jogo, fizeram-se as seguintes considerações:

- Por cada movimento possível, atribui-se um ponto;
- Por cada caranguejo *Big* do jogador no topo de uma pilha, atribui-se 3 pontos;

- Por cada caranguejo *Medium* do jogador no topo de uma pilha, atribui-se 2 pontos;
- Finalmente, por cada caranguejo *Small* do jogador no topo de uma pilha, atribui-se 1 pontos.

### 3.6 Final do Jogo

A cada jogada é necessário verificar se o jogo terminou. Para o efeito, criou-se o predicado `game_over(+Board, +Winner)` que verifica se a condição de final de jogo foi atingida. O jogo termina, se um jogador ficar sem movimentos possíveis. O predicado `moves_left(+Board, +Player_Color, +Moves, +Count, -Final_Moves)` calcula o número de jogadas restantes no estado atual de jogo e é invocado pelo predicado `game_over`. No caso de `Final_Moves` for igual a zero para algum dos jogadores, o outro jogador é declarado vencedor.

### 3.7 Jogada do Computador

A jogada de um computer é feita a partir do predicado move_computer(+Board, +Computer_Color, +Depth, -Computer_FinalBoard), em que a profundidade indica o nível de dificuldade do computador, que equivale à profundidade de pesquisa no algoritmo *alphabeta*. Os predicados que influenciam na escolha do movimento do computador, são os indicados na secção de Avaliação do Tabuleiro e que são invocados a partir do predicado `alpha_beta(+Board, +Player, +Depth, +Alpha, +Beta, -Move, -Value)`.

## 4 Interface com o Utilizador

O jogo arranca com a chamada do predicado *crabStack*. É apresentado ao jogador o menu principal 5, onde este pode escolher se pretende jogar (*1. Play*), sendo encaminhado para o sub-menu de jogo, ler as regras de jogo (*2. How to play*), conhecer os criadores do jogo (*3. About*) ou sair da aplicação (*4. Exit*). Para uma melhor compreensão dos diferentes sub-menus, estes podem ser consultados no anexo A.

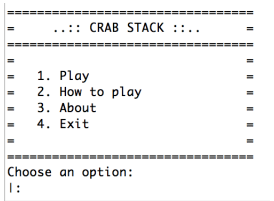



Figura 5: Representação do menu principal.

Caso o jogador tenha escolhido a opção 1, é encaminhado para o sub-menu de modo de jogo onde pode escolher entre 3 opções de jogo:

- 1. Jogador vs. Jogador
- 2. Jogador vs. Computador
- 3. Computador vs. Computador

Se o jogador escolher algum dos modos de computador, terá de introduzir o nível de dificuldade do mesmo.

# 5 Conclusões

Durante este projeto verificou-se um aprofundamento dos conhecimentos em Prolog anteriormente adquiridos durante as aulas e um melhor domínio sobre os mesmos. Este projeto permitiu também, uma melhor compreensão do novo paradigma de programação que a aprendizagem do Prolog exigiu.

O jogo implementado cumpriu as várias regras de jogo descritas e o modo AI foi implementado com sucesso para diferentes níveis de dificuldade, facultados pelo nível de profundidade de pesquisa do algoritmo *alphabeta*. Um aspeto a melhorar, é a eficiência de avaliação e escolha do movimento a efetuar pelo computador.

No geral, concluí-se que o trabalho desenvolvido foi produtivo e positivo, cumprindo todas as especificações exigidas.

# Bibliografia

[1] Henrique Lopes Cardoso Daniel Silva, Rui Camacho. Enunciado: Aplicação em Prolog para um Jogo de Tabuleiro, 2016.

[2] Eric Mortensen. Crab Stack Review and Instructions, 2015.

[3] Blue Orange. Crab Stack — Board Game — BoardGameGeek, 2016.

[4] Crab Stack. Blue Orange Games, 2010.

# A Interface com o utilizador

## A.1 How to play

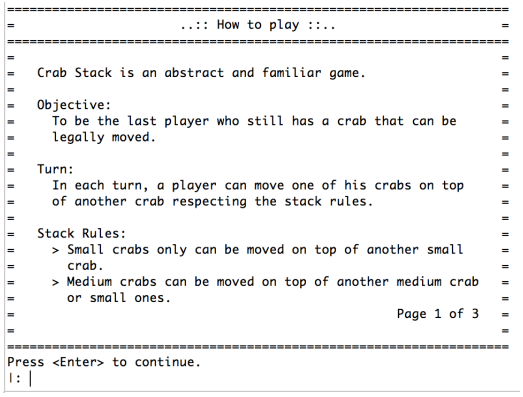



Figura 6: Representação do sub-menu *How to play* página 1.

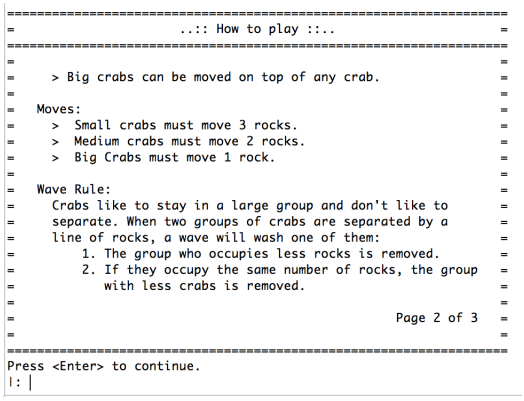



Figura 7: Representação do sub-menu *How to play* página 2.

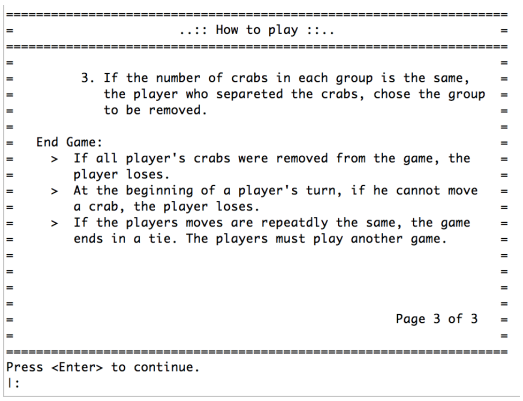



Figura 8: Representação do sub-menu *How to play* página 3.

## A.2 About

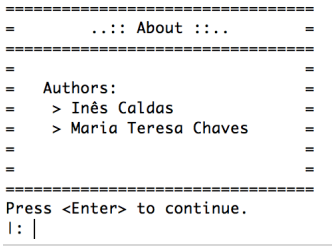



Figura 9: Representação do sub-menu *About*.

## A.3 Play

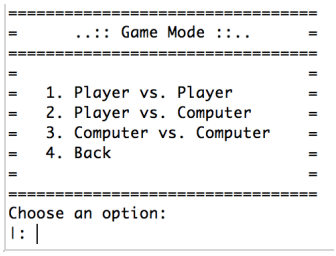



Figura 10: Representação do sub-menu *Play*.

## A.4 Player vs Player

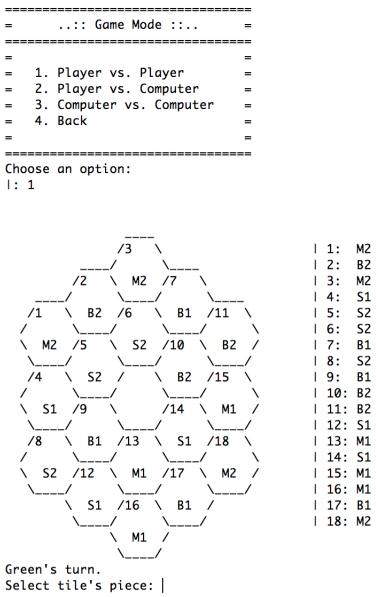



Figura 11: Representação do sub-menu *player vs player*.

## A.5 Player vs Computer

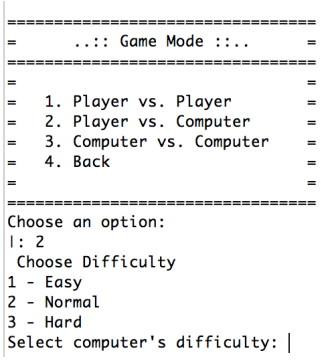



Figura 12: Representação do sub-menu *player vs computer*.

## A.6 Computer vs Computer

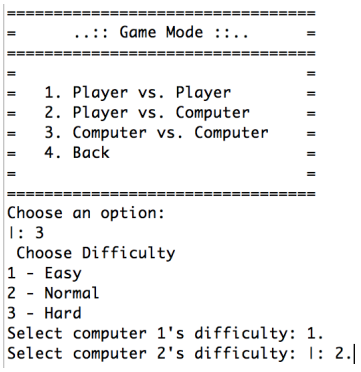



Figura 13: Representação do sub-menu *computer vs computer*.

# B Código Fonte

## B.1 crab_stack.pl

```
/* ********************************************* */
/*                                               */
/*                    CRABSTACK                  */
/*                                               */
/*  To play, run the query crabStack/0.          */
/*  Available game modes include:                */
/*     - Human vs Human                          */
/*     - Human vs Computer                       */
/*     - Computer vs Computer                    */
/*                                               */
/* ********************************************* */

:- [cs_board, cs_menus, cs_utilities, cs_ai].
:- use_module(library(random)).

/* ********************************************* */
/*                                               */
/*                   Game Launcher               */
/*                                               */
/* ********************************************* */

crabStack:-
        mainMenu.

/* ********************************************* */
/*                                               */
/*                Main Game Logic                */
/*                                               */
/* ********************************************* */

playGamePvP :-
        play('human', g, 'human', b).

playGamePvB :-
        play('human', g, 'computer', b).

playGameBvB :-
        play('computer', g, 'computer', b).


/* ********************************************* */
/*                                               */
/*   play/{5, 6, 7}                              */
/*      +Arg 1:   the board                      */
/*      +Arg 1,2: player 1                       */
/*      +Arg 2,3: player 1's color               */
/*      +Arg 4: player 1's difficulty            */
/*      +Arg 4,5: player 2                       */
/*      +Arg 5,6: player 2's color               */
/*      +Arg 7: player 2's difficulty            */
/*   Summary: This function starts the game.     */
/*                                               */
/* ********************************************* */

%% 1 - calling rule
play('human', P1_Color, 'human', P2_Color) :-
        init_board(Board), !,
        play(Board, 'human', P1_Color, 'human', P2_Color).

%% 1 - calling rule
play('human', P1_Color, 'computer', P2_Color) :-
        init_board(Board), !,
        ask_level(Depth),
        play(Board, 'human', P1_Color, 'computer', P2_Color, Depth).

%% 1 - calling rule
play('computer', P1_Color, 'computer', P2_Color) :-
        init_board(Board), !,
        ask_level(Depth1, Depth2),
        play(Board, 'computer', P1_Color, Depth1, 'computer', P2_Color, Depth2).

```

```
72  %% 1 - base case: game has been won
73  play(Board, _, _, _, _) :-
74          game_over(Board, Winner),
75          print_board(Board),
76          print_winner(Winner).
77
78  %% 2 - recursive: game not over
79  play(Board, 'human', P1_Color, 'human', P2_Color) :-
80          \+ game_over(Board, _),
81          print_board(Board),
82          make_move(Board, 'human', P1_Color, FinalBoard), !,
83          play(FinalBoard, 'human', P2_Color, 'human', P1_Color).
84
85  %% 1 - base case: game has been won
86  play(Board, _, _, _, _, _) :-
87          game_over(Board, Winner),
88          print_board(Board),
89          print_winner(Winner).
90
91  %% 2 - recursive: game not over
92  play(Board, 'human', P1_Color,'computer', P2_Color, Depth) :-
93          \+ game_over(Board, _),
94          print_board(Board),
95          make_move(Board, 'human', P1_Color, FinalBoard), !,
96          display_board(FinalBoard),
97          move_computer(FinalBoard, P2_Color, Depth, Computer_FinalBoard),
98          play(Computer_FinalBoard, 'human', P1_Color, 'computer', P2_Color, Depth).
99
100 %% 1 - base case: game has been won
101 play(Board, _, _, _, _, _, _) :-
102         game_over(Board, Winner),
103         print_board(Board),
104         print_winner(Winner).
105
106 %% 2 - recursive: game not over
107 play(Board, 'computer', P1_Color, Depth1, 'computer', P2_Color, Depth2) :-
108         \+ game_over(Board, _),
109         display_board(Board),
110         move_computer(Board, P1_Color, Depth1, Computer1_FinalBoard),
111         play(Computer1_FinalBoard, 'computer', P2_Color, Depth2, 'computer', P1_Color,
            ↪   Depth1).
112
113
114 /* ********************************************** */
115 /*                                                */
116 /*   make_move/4                                  */
117 /*      +Arg 1: the hex board                     */
118 /*      +Arg 2: the player                        */
119 /*      +Arg 3: the player's color                */
120 /*      -Arg 4: final board                       */
121 /*   Summary: Puts a stone of the specified color */
122 /*            on the specified tile of the game   */
123 /*            board.                              */
124 /*                                                */
125 /* ********************************************** */
126
127 make_move(Board, Player, Player_Color, FinalBoard) :-
128         choose_tile(Board, Player, Player_Color, FinalBoard).
129
130
131 /* ********************************************** */
132 /*                                                */
133 /*   choose_tile/4                                */
134 /*      +Arg 1: the hex board                     */
135 /*      +Arg 2: the player                        */
136 /*      +Arg 3: the player's color                */
137 /*      -Arg 4: final board                       */
138 /*   Summary: Allows the player to choose a tile to */
139 /*            play.                               */
140 /*                                                */
141 /* ********************************************** */
142
143 choose_tile(Board, human, Player_Color, FinalBoard) :-
144         repeat,
145         color(Player_Color, Color_Text),
146         format('~s''s turn.', Color_Text),
```

```
147         nl,
148         print('Select tile''s piece: '),
149         read(Rock_Init),
150         print('Select tile''s destination: '),
151         read(Rock),
152         legalMove(Board, Rock_Init, Rock, Player_Color, FinalBoard).
153
154
155 /* ********************************************* */
156 /*                                               */
157 /*   legalMove/5                                 */
158 /*      +Arg 1: actual board                     */
159 /*      +Arg 2: initial rock                     */
160 /*      +Arg 3: destination rock                 */
161 /*      +Arg 4: player color                     */
162 /*      -Arg 5: final board                      */
163 /*   Summary: Determines if a tile can legally be */
164 /*            played.                            */
165 /*                                               */
166 /* ********************************************* */
167
168 legalMove(Board, Rock_Init, Rock, Player_Color, FinalBoard) :-
169         Rock_Init \= Rock,
170         in_range(Rock_Init),
171         in_range(Rock),
172         get_rock(Rock_Init, Board, Crab),
173         crab_stats(Crab, Crab_Size, Crab_Color),
174         Crab_Color == Player_Color, %% Checks if the crab belongs to the player
175         valid_crab_movement(Board, Rock_Init, Rock, Crab, Crab_Size, FinalBoard).
176
177
178 /* ********************************************* */
179 /*                                               */
180 /*   valid_crab_movement/6                       */
181 /*      +Arg 1: actual board                     */
182 /*      +Arg 2: initial rock                     */
183 /*      +Arg 3: destination rock                 */
184 /*      +Arg 4: the crab (eg. M1)                */
185 /*      +Arg 5: crab size                        */
186 /*      -Arg 6: final board                      */
187 /*   Summary: Determines if a crab movement is   */
188 /*            valid.                             */
189 /*                                               */
190 /* ********************************************* */
191
192 valid_crab_movement(Board, Rock_Init, Rock, Crab, Crab_Size, FinalBoard):-
193         dist(Crab_Size, Rock_Init, Valid_Moves),
194         member(Rock, Valid_Moves), %% valid distance for crab size
195         get_rock(Rock, Board, Crab_Top),
196         crab_stats(Crab_Top, Crab_Top_Size, _),
197         valid_pile_crab(Crab_Size, Crab_Top_Size),
198         update_board(Board, Rock, Rock_Init, Crab, FinalBoard).
199
200
201 /* ********************************************* */
202 /*                                               */
203 /*   update_board/5                              */
204 /*      +Arg 1: actual board                     */
205 /*      +Arg 2: destination rock                 */
206 /*      +Arg 3: initial rock                     */
207 /*      +Arg 4: the crab (eg. M1)                */
208 /*      -Arg 5: final board                      */
209 /*   Summary: Updates the borad with the crab    */
210 /*            movement.                          */
211 /*                                               */
212 /* ********************************************* */
213
214 update_board(Board, Rock, Rock_Init, Crab, FinalBoard):-
215         add_crab_board(Board, Rock, Crab, FinalTemp),
216         remove_crab_board(FinalTemp, Rock_Init, FinalBoard),
217         \+ (wave(FinalBoard, Rock_Init, _)).
218
219 update_board(Board, Rock, Rock_Init, Crab, FinalBoard):-
220         add_crab_board(Board, Rock, Crab, FinalTemp),
221         remove_crab_board(FinalTemp, Rock_Init, Tmp_Board),
222         wave(Tmp_Board, Rock_Init, FinalBoard).
```

```
223
224
225 /* ********************************************** */
226 /*                                                */
227 /*   add_crab_board/4                             */
228 /*      +Arg 1: actual board                      */
229 /*      +Arg 2: rock position                     */
230 /*      +Arg 3: the crab (eg. M1)                 */
231 /*      -Arg 4: final board                       */
232 /*   Summary: Adds the crab to the rock position. */
233 /*                                                */
234 /* ********************************************** */
235
236 add_crab_board(Board, Rock, Crab, FinalBoard):-
237         (Rock == 1; Rock == 2; Rock == 3),
238         addCrab(Board, 1, Rock, Crab, FinalBoard).
239
240 add_crab_board(Board, Rock, Crab, FinalBoard):-
241         (Rock == 4; Rock == 5; Rock == 6; Rock == 7),
242         Col is Rock - 3,
243         addCrab(Board, 2, Col, Crab, FinalBoard).
244
245 add_crab_board(Board, Rock, Crab, FinalBoard):-
246         (Rock == 8; Rock == 9; Rock == 10; Rock == 11),
247         Col is Rock - 7,
248         addCrab(Board, 3, Col, Crab, FinalBoard).
249
250 add_crab_board(Board, Rock, Crab, FinalBoard):-
251         (Rock == 12; Rock == 13; Rock == 14; Rock == 15),
252         Col is Rock - 11,
253         addCrab(Board, 4, Col, Crab, FinalBoard).
254
255 add_crab_board(Board, Rock, Crab, FinalBoard):-
256         (Rock == 16; Rock == 17; Rock == 18),
257         Col is Rock - 15,
258         addCrab(Board, 5, Col, Crab, FinalBoard).
259
260
261 /* ********************************************** */
262 /*                                                */
263 /*   remove_stack_board/3                         */
264 /*      +Arg 1: actual board                      */
265 /*      +Arg 2: rock position                     */
266 /*      -Arg 3: final board                       */
267 /*   Summary: Removes all crabs of the a rock     */
268 /*            position.                           */
269 /*                                                */
270 /* ********************************************** */
271
272 remove_stack_board(Board, Rock, FinalBoard):-
273         (Rock == 1; Rock == 2; Rock == 3),
274         removeCrabStack(Board, 1, Rock, FinalBoard).
275
276 remove_stack_board(Board, Rock, FinalBoard):-
277         (Rock == 4; Rock == 5; Rock == 6; Rock == 7),
278         Col is Rock - 3,
279         removeCrabStack(Board, 2, Col, FinalBoard).
280
281 remove_stack_board(Board, Rock, FinalBoard):-
282         (Rock == 8; Rock == 9; Rock == 10; Rock == 11),
283         Col is Rock - 7,
284         removeCrabStack(Board, 3, Col, FinalBoard).
285
286 remove_stack_board(Board, Rock, FinalBoard):-
287         (Rock == 12; Rock == 13; Rock == 14; Rock == 15),
288         Col is Rock - 11,
289         removeCrabStack(Board, 4, Col, FinalBoard).
290
291 remove_stack_board(Board, Rock, FinalBoard):-
292         (Rock == 16; Rock == 17; Rock == 18),
293         Col is Rock - 15,
294         removeCrabStack(Board, 5, Col, FinalBoard).
295
296
297 /* ********************************************** */
298 /*                                                */
```

```
299 /*   remove_crab_board/3                          */
300 /*      +Arg 1: actual board                      */
301 /*      +Arg 2: rock position                     */
302 /*      -Arg 3: final board                       */
303 /*   Summary: Removes the top crab of the rock    */
304 /*            position.                           */
305 /*                                                */
306 /* ********************************************** */
307
308 remove_crab_board(Board, Rock, FinalBoard):-
309         (Rock == 1; Rock == 2; Rock == 3),
310         removeCrab(Board, 1, Rock, FinalBoard).
311
312 remove_crab_board(Board, Rock, FinalBoard):-
313         (Rock == 4; Rock == 5; Rock == 6; Rock == 7),
314         Col is Rock - 3,
315         removeCrab(Board, 2, Col, FinalBoard).
316
317 remove_crab_board(Board, Rock, FinalBoard):-
318         (Rock == 8; Rock == 9; Rock == 10; Rock == 11),
319         Col is Rock - 7,
320         removeCrab(Board, 3, Col, FinalBoard).
321
322 remove_crab_board(Board, Rock, FinalBoard):-
323         (Rock == 12; Rock == 13; Rock == 14; Rock == 15),
324         Col is Rock - 11,
325         removeCrab(Board, 4, Col, FinalBoard).
326
327 remove_crab_board(Board, Rock, FinalBoard):-
328         (Rock == 16; Rock == 17; Rock == 18),
329         Col is Rock - 15,
330         removeCrab(Board, 5, Col, FinalBoard).
331
332
333 /* ********************************************** */
334 /*                                                */
335 /*   game_over/2                                  */
336 /*      +Arg 1: the board                         */
337 /*      -Arg 2: winner                            */
338 /*   Summary: Determines if the game is over and  */
339 /*            if so returns the winner.           */
340 /*                                                */
341 /* ********************************************** */
342
343 game_over(Board, b) :-
344         moves_left(Board, g, 0, [], 1, Moves, _List_Moves),
345         Moves == 0.
346
347 game_over(Board, g) :-
348         moves_left(Board, b, 0, [], 1, Moves, _List_Moves),
349         Moves == 0.
350
351
352 /* ********************************************** */
353 /*                                                */
354 /*   moves_left/7                                 */
355 /*      +Arg 1: the board                         */
356 /*      +Arg 2: player color                      */
357 /*      +Arg 3: number of left moves (counter)    */
358 /*      +Arg 4: list of left moves (counter)      */
359 /*      +Arg 5: initial position                  */
360 /*      -Arg 6: number of movements               */
361 /*      +Arg 7: list of left moves                */
362 /*   Summary: Determines the number of left moves */
363 /*            the player can make.                */
364 /*                                                */
365 /* ********************************************** */
366
367 moves_left(_, _, Moves, List_Moves, 19, Moves, List_Moves). % final state
368
369 moves_left(Board, Player_Color, Moves, List_Moves_Counter, Count, Final_Moves,
    ↪   List_Final_Moves):-
370         Count \= 1,
371         \+ (get_rock(Count, Board, _)),
372         New_Count is Count + 1,
```

```
373         moves_left(Board, Player_Color, Moves, List_Moves_Counter, New_Count,
             ↪  Final_Moves, List_Final_Moves), !.
374
375 moves_left(Board, Player_Color, Moves, List_Moves_Counter, Count, Final_Moves,
     ↪  List_Final_Moves):-
376         Count \= 1,
377         get_rock(Count, Board, Crab),
378         crab_stats(Crab, Crab_Size, Crab_Color),
379         Crab_Color == Player_Color, %% Checks if the crab belongs to the player
380         check_moves(Board, Count, Crab_Size, 1, Player_Color, Moves, [], Moves_Rock,
             ↪  New_List_Moves),
381         New_Count is Count + 1,
382                 append(New_List_Moves, List_Moves_Counter, NNew_List_Moves),
383         moves_left(Board, Player_Color, Moves_Rock, NNew_List_Moves, New_Count,
             ↪  Final_Moves, List_Final_Moves), !.
384
385 moves_left(Board, Player_Color, Moves, List_Moves_Counter, Count, Final_Moves,
     ↪  List_Final_Moves):-
386         Count \= 1,
387         get_rock(Count, Board, Crab),
388         crab_stats(Crab, _, Crab_Color),
389         Crab_Color \= Player_Color, %% Checks if the crab belongs to the player
390         New_Count is Count + 1,
391         moves_left(Board, Player_Color, Moves, List_Moves_Counter, New_Count,
             ↪  Final_Moves, List_Final_Moves), !.
392
393 moves_left(Board, Player_Color, Moves, List_Moves_Counter, Count, Final_Moves,
     ↪  List_Final_Moves):-
394         Count == 1,
395         \+ (get_rock(Count, Board, _)),
396         New_Count is Count + 1,
397         moves_left(Board, Player_Color, Moves, List_Moves_Counter, New_Count,
             ↪  Final_Moves, List_Final_Moves), !.
398
399 moves_left(Board, Player_Color, Moves, List_Moves_Counter, Count, Final_Moves,
     ↪  List_Final_Moves):-
400         Count == 1,
401         get_rock(Count, Board, Crab),
402         crab_stats(Crab, Crab_Size, Crab_Color),
403         Crab_Color == Player_Color, %% Checks if the crab belongs to the player
404         check_moves(Board, Count, Crab_Size, 2, Player_Color, Moves, [], Moves_Rock,
             ↪  New_List_Moves),
405         New_Count is Count + 1,
406                 append(New_List_Moves, List_Moves_Counter, NNew_List_Moves),
407         moves_left(Board, Player_Color, Moves_Rock, NNew_List_Moves, New_Count,
             ↪  Final_Moves, List_Final_Moves), !.
408
409 moves_left(Board, Player_Color, Moves, List_Moves_Counter, Count, Final_Moves,
     ↪  List_Final_Moves):-
410         Count == 1,
411         get_rock(Count, Board, Crab),
412         crab_stats(Crab, _, Crab_Color),
413         Crab_Color \= Player_Color, %% Checks if the crab belongs to the player
414         New_Count is Count + 1,
415         moves_left(Board, Player_Color, Moves, List_Moves_Counter, New_Count,
             ↪  Final_Moves, List_Final_Moves), !.
416
417
418 /* ********************************************** */
419 /*                                                */
420 /*   check_moves/9                                */
421 /*      +Arg 1: the board                         */
422 /*      +Arg 2: initial position                  */
423 /*      +Arg 3: size of initial position crab     */
424 /*      +Arg 4: final position                    */
425 /*      +Arg 5: player color                      */
426 /*      +Arg 6: number of movements (counter)     */
427 /*      +Arg 7: list of movements (counter)       */
428 /*      -Arg 8: number of movements               */
429 /*      -Arg 9: list of movements                 */
430 /*   Summary: Determines the number of left moves */
431 /*            the player can make on the          */
432 /*            initial position.                   */
433 /*                                                */
434 /* ********************************************** */
435
```

```
436 check_moves(_, 17, _, 19, _, Moves, ListMoves, Moves, ListMoves).
437
438 check_moves(_, _, _, 19, _, Moves, ListMoves, Moves, ListMoves).
439
440 check_moves(Board, Init_Pos, Init_Crab_Size, Final_Pos, Player_Color, Moves, ListMoves,
    ↪  Moves_Rock, List_Moves_Rock):-
441         Init_Pos \= Final_Pos,
442         get_rock(Final_Pos, Board, Final_Crab),
443         crab_stats(Final_Crab, Final_Crab_Size, _),
444         valid_pile_crab(Init_Crab_Size, Final_Crab_Size),
445         dist(Init_Crab_Size, Init_Pos, Valid_Moves),
446         member(Final_Pos, Valid_Moves), %% valid distance for crab size
447         New_Moves is Moves + 1,
448         New_Final_Pos is Final_Pos + 1,
449         check_moves(Board, Init_Pos, Init_Crab_Size, New_Final_Pos, Player_Color,
          ↪  New_Moves, [[Init_Pos, Final_Pos] | ListMoves], Moves_Rock,
          ↪  List_Moves_Rock), !.
450
451 check_moves(Board, Init_Pos, Init_Crab_Size, Final_Pos, Player_Color, Moves, ListMoves,
    ↪  Moves_Rock, List_Moves_Rock):-
452         Init_Pos \= Final_Pos,
453         \+ (get_rock(Final_Pos, Board, _)),
454         New_Final_Pos is Final_Pos + 1,
455         check_moves(Board, Init_Pos, Init_Crab_Size, New_Final_Pos, Player_Color,
          ↪  Moves, ListMoves, Moves_Rock, List_Moves_Rock), !.
456
457 check_moves(Board, Init_Pos, Init_Crab_Size, Final_Pos, Player_Color, Moves, ListMoves,
    ↪  Moves_Rock, List_Moves_Rock):-
458         Init_Pos == Final_Pos,
459         New_Final_Pos is Final_Pos + 1,
460         check_moves(Board, Init_Pos, Init_Crab_Size, New_Final_Pos, Player_Color,
          ↪  Moves, ListMoves, Moves_Rock, List_Moves_Rock).
461
462 check_moves(Board, Init_Pos, Init_Crab_Size, Final_Pos, Player_Color, Moves, ListMoves,
    ↪  Moves_Rock, List_Moves_Rock):-
463         Init_Pos \= Final_Pos,
464         get_rock(Final_Pos, Board, Final_Crab),
465         crab_stats(Final_Crab, Final_Crab_Size, _),
466         \+ (valid_pile_crab(Init_Crab_Size, Final_Crab_Size)),
467         New_Final_Pos is Final_Pos + 1,
468         check_moves(Board, Init_Pos, Init_Crab_Size, New_Final_Pos, Player_Color,
          ↪  Moves, ListMoves, Moves_Rock, List_Moves_Rock), !.
469
470 check_moves(Board, Init_Pos, Init_Crab_Size, Final_Pos, Player_Color, Moves, ListMoves,
    ↪  Moves_Rock, List_Moves_Rock):-
471         Init_Pos \= Final_Pos,
472         get_rock(Final_Pos, Board, Final_Crab),
473         crab_stats(Final_Crab, Final_Crab_Size, _),
474         valid_pile_crab(Init_Crab_Size, Final_Crab_Size),
475         dist(Init_Crab_Size, Init_Pos, Valid_Moves),
476         \+ (member(Final_Pos, Valid_Moves)), %% valid distance for crab size
477         New_Final_Pos is Final_Pos + 1,
478         check_moves(Board, Init_Pos, Init_Crab_Size, New_Final_Pos, Player_Color,
          ↪  Moves, ListMoves, Moves_Rock, List_Moves_Rock), !.
479
480
481 /* ********************************************** */
482 /*                                                */
483 /*   in_range/1                                   */
484 /*      +Arg 1: the tile                          */
485 /*   Summary: Determines if the tile is in the    */
486 /*            playable range of tiles.            */
487 /*                                                */
488 /* ********************************************** */
489
490 in_range(Tile) :-
491         integer(Tile),
492         Tile > 0,
493         Tile < 19.
494
495
496 /* ********************************************** */
497 /*                                                */
498 /*                  Game Rules                    */
499 /*                                                */
500 /* ********************************************** */
```

```
501
502 /* ********************************************** */
503 /*                                                */
504 /*   dist/3                                       */
505 /*      +Arg 1: crab size                         */
506 /*      +Arg 2: crab position                     */
507 /*      -Arg 3: list of possible moves            */
508 /*   Summary: Determines the positions that a crab */
509 /*            can go given it's size and position. */
510 /*                                                */
511 /* ********************************************** */
512
513 dist('B', 1, [2, 4, 5]).
514 dist('M', 1, [2, 3, 4, 5, 6, 8, 9]).
515 dist('S', 1, [2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13]).
516
517 dist('B', 2, [1, 3, 5, 6]).
518 dist('M', 2, [1, 3, 4, 5, 6, 7, 9, 10]).
519 dist('S', 2, [1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]).
520
521 dist('B', 3, [2, 6, 7]).
522 dist('M', 3, [1, 2, 5, 6, 7, 10, 11]).
523 dist('S', 3, [1, 2, 4, 5, 6, 7, 9, 10, 11, 14, 15]).
524
525 dist('B', 4, [1, 5, 8, 9]).
526 dist('M', 4, [1, 2, 5, 6, 8, 9, 12, 13]).
527 dist('S', 4, [1, 2, 3, 5, 6, 7, 8, 9, 10, 12, 13, 14, 16, 17]).
528
529 dist('B', 5, [1, 2, 4, 6, 9]).
530 dist('M', 5, [1, 2, 3, 4, 6, 7, 8, 9, 10, 12, 13]).
531 dist('S', 5, [1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]).
532
533 dist('B', 6, [2, 3, 5, 7, 10]).
534 dist('M', 6, [1, 2, 3, 4, 5, 7, 9, 10, 11, 14, 15]).
535 dist('S', 6, [1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 17, 18]).
536
537 dist('B', 7, [3, 6, 10, 11]).
538 dist('M', 7, [2, 3, 5, 6, 10, 11, 14, 15]).
539 dist('S', 7, [1, 2, 3, 4, 5, 6, 9, 10, 11, 13, 14, 15, 17, 18]).
540
541 dist('B', 8, [4, 9, 12]).
542 dist('M', 8, [1, 4, 5, 9, 12, 13, 16]).
543 dist('S', 8, [1, 2, 4, 5, 6, 9, 12, 13, 14, 16, 17]).
544
545 dist('B', 9, [4, 5, 8, 12, 13]).
546 dist('M', 9, [1, 2, 4, 5, 6, 8, 12, 13, 14, 16, 17]).
547 dist('S', 9, [1, 2, 3, 4, 5, 6, 7, 8, 10, 12, 13, 14, 15, 16, 17, 18]).
548
549 dist('B', 10, [6, 7, 11, 14, 15]).
550 dist('M', 10, [2, 3, 5, 6, 7, 11, 13, 14, 15, 17, 18]).
551 dist('S', 10, [1, 2, 3, 4, 5, 6, 7, 9, 11, 12, 13, 14, 15, 16, 17, 18]).
552
553 dist('B', 11,  [7, 10, 15]).
554 dist('M', 11,  [3, 6, 7, 10, 14, 15, 17, 18]).
555 dist('S', 11,  [2, 3, 5, 6, 7, 10, 13, 14, 15, 17, 18]).
556
557 dist('B', 12,  [8, 9, 13, 16]).
558 dist('M', 12,  [4, 5, 8, 9, 13, 14, 16, 17]).
559 dist('S', 12,  [1, 2, 4, 5, 6, 8, 9, 10, 13, 14, 15, 16, 17, 18]).
560
561 dist('B', 13,  [9, 12, 14, 16, 17]).
562 dist('M', 13,  [4, 5, 8, 9, 10, 12, 14, 15, 16, 17, 18]).
563 dist('S', 13,  [1, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 15, 16, 17, 18]).
564
565 dist('B', 14,  [10, 13, 15, 17, 18]).
566 dist('M', 14,  [6, 7, 9, 10, 11, 12, 13, 15, 16, 17, 18]).
567 dist('S', 14,  [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15, 16, 17, 18]).
568
569 dist('B', 15,  [10, 11, 14, 18]).
570 dist('M', 15,  [6, 7, 10, 11, 13, 14, 17, 18]).
571 dist('S', 15,  [2, 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 16, 17, 18]).
572
573 dist('B', 16,  [12, 13, 17]).
574 dist('M', 16,  [8, 9, 12, 13, 14, 17, 18]).
575 dist('S', 16,  [4, 5, 8, 9, 10, 12, 13, 14, 15, 17, 18]).
576
```

```
577   dist('B', 17,  [13, 14, 16, 18]).
578   dist('M', 17,  [9, 10, 12, 13, 14, 15, 16, 18]).
579   dist('S', 17,  [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 18]).
580
581   dist('B', 18,  [14, 15, 17]).
582   dist('M', 18,  [10, 11, 13, 14, 15, 16, 17]).
583   dist('S', 18,  [6, 7, 9, 10, 11, 12, 13, 14, 15, 16, 17]).
584
585
586   /* ********************************************** */
587   /*                                                */
588   /*   water_limiit/1                               */
589   /*      ?Arg 1: rocks with water as neighbor      */
590   /*   Summary: Defines the rocks that are neighbors */
591   /*            of the water.                       */
592   /*                                                */
593   /* ********************************************** */
594
595   water_limit([5, 6, 9, 10, 13, 14]).
596
597
598   /* ********************************************** */
599   /*                                                */
600   /*   limiit/1                                     */
601   /*      ?Arg 1: rocks in the limit of the board   */
602   /*   Summary: Defines the rocks that are in the   */
603   /*            limit of the board.                 */
604   /*                                                */
605   /* ********************************************** */
606
607   limit([1, 2, 3, 4, 7, 8, 11, 12, 15, 16, 17, 18]).
608
609
610   /* ********************************************** */
611   /*                                                */
612   /*   line/1                                       */
613   /*      ?Arg 1: line of the board                 */
614   /*   Summary: Defines a horizontal line of the    */
615   /*            board.                              */
616   /*                                                */
617   /* ********************************************** */
618
619   line([1, 2, 3]).
620   line([4, 5, 6, 7]).
621   line([8, 9, 10, 11]).
622   line([12, 13, 14, 15]).
623   line([16, 17, 18]).
624
625
626   /* ********************************************** */
627   /*                                                */
628   /*   valid_pile_crab/2                            */
629   /*      +Arg 1: Crab to be moved size             */
630   /*      +Arg 2: Crab on top of rock size          */
631   /*   Summary: Determines if a tile can legally be */
632   /*            played.                             */
633   /*                                                */
634   /* ********************************************** */
635
636   valid_pile_crab('S', 'S').
637   valid_pile_crab('M', 'S').
638   valid_pile_crab('M', 'M').
639   valid_pile_crab('B', 'S').
640   valid_pile_crab('B', 'M').
641   valid_pile_crab('B', 'B').
642
643
644   /* ********************************************** */
645   /*                                                */
646   /*   get_empty_board_limits/3                     */
647   /*      +Arg 1: board                             */
648   /*      +Arg 2: position                          */
649   /*      -Arg 3: empty position neighbors board    */
650   /*              limits                            */
651   /*   Summary: Determines the empty position       */
652   /*            neighbors board limits.             */
```

```
653 /*                                                 */
654 /* *********************************************** */
655
656 get_empty_board_limits(Board, Position, EmptyLimits):-
657         dist('B', Position, Big_Limits),
658         empty_neighbors(Board, Big_Limits, [], Empty_Big_Limits),
659         length(Empty_Big_Limits, Num_Empties),
660         Num_Empties \= 0,
661         limit(Board_Limits),
662         intersection(Empty_Big_Limits, Board_Limits, EmptyLimits).
663
664
665 /* *********************************************** */
666 /*                                                 */
667 /*   get_empty_water_limits/3                      */
668 /*      +Arg 1: board                              */
669 /*      +Arg 2: position                           */
670 /*      -Arg 3: empty position neighbors water     */
671 /*              limits                             */
672 /*   Summary: Determines the empty position        */
673 /*            neighbors water limits.              */
674 /*                                                 */
675 /* *********************************************** */
676
677 get_empty_water_limits(Board, Position, EmptyLimits):-
678         dist('B', Position, Big_Limits),
679         empty_neighbors(Board, Big_Limits, [], Empty_Big_Limits),
680         length(Empty_Big_Limits, Num_Empties),
681         Num_Empties \= 0,
682         water_limit(WLimits),
683         intersection(Empty_Big_Limits, WLimits, EmptyLimits).
684
685
686 /* *********************************************** */
687 /*                                                 */
688 /*   get_final_position_limits/4                   */
689 /*      +Arg 1: board                              */
690 /*      +Arg 2: initial position empty limits      */
691 /*      +Arg 3: water limits                       */
692 /*      -Arg 4: empty positions of water neighbors */
693 /*              board limits                       */
694 /*   Summary: Determines the empty positions of    */
695 /*            water neighbors board limits.        */
696 /*                                                 */
697 /* *********************************************** */
698
699 get_final_position_limits(_, _, [], []).
700
701 get_final_position_limits(Board, Init_EmptyLimits, [HWLimits|TWLimits],
    ↪   FinalPositions):-
702         get_rock(HWLimits, Board, _),
703         get_final_position_limits(Board, Init_EmptyLimits, TWLimits, FinalPositions),
          ↪   !.
704
705 get_final_position_limits(Board, Init_EmptyLimits, [HWLimits|TWLimits],
    ↪   FinalPositions):-
706         \+ (get_rock(HWLimits, Board, _)),
707         get_empty_board_limits(Board, HWLimits, Final_EmptyLimits),
708         append(Init_EmptyLimits, Final_EmptyLimits, EmptyLimits),
709         intersection(Init_EmptyLimits, Final_EmptyLimits, Duplicates),
710         delete_all_list(Duplicates, EmptyLimits, L),
711         length(L, Num_FinalPositions),
712         Num_FinalPositions < 2,
713         get_final_position_limits(Board, Init_EmptyLimits, TWLimits, FinalPositions),
          ↪   !.
714
715 get_final_position_limits(Board, Init_EmptyLimits, [HWLimits|_TWLimits],
    ↪   FinalPositions):-
716         \+ (get_rock(HWLimits, Board, _)),
717         get_empty_board_limits(Board, HWLimits, Final_EmptyLimits),
718         append(Init_EmptyLimits, Final_EmptyLimits, EmptyLimits),
719         intersection(Init_EmptyLimits, Final_EmptyLimits, Duplicates),
720         delete_all_list(Duplicates, EmptyLimits, Tmp_FinalPositions),
721         length(Tmp_FinalPositions, Num_FinalPositions),
722         Num_FinalPositions >= 2,
723         append(Tmp_FinalPositions, [HWLimits], FinalPositions).
```

```
724
725
726   /* ********************************************* */
727   /*                                               */
728   /*   wave/3                                      */
729   /*      +Arg 1: board                            */
730   /*      +Arg 2: initial position                 */
731   /*      +Arg 3: final board                      */
732   /*   Summary: Determines if a tile can legally be   */
733   /*            played.                            */
734   /*                                               */
735   /* ********************************************* */
736
737   % initial position is a water limit
738   wave(Board, Init_Pos, FinalBoard):-
739           \+ (get_rock(Init_Pos, Board, _)), % empty pile
740           water_limit(WLimits),
741           member(Init_Pos, WLimits),
742           get_empty_board_limits(Board, Init_Pos, Init_EmptyLimits),
743           length(Init_EmptyLimits, Num_Init_EmptyLimits),
744           Num_Init_EmptyLimits \= 0,
745           delete(WLimits, Init_Pos, New_WLimits),
746           get_final_position_limits(Board, Init_EmptyLimits, New_WLimits,
              ↪  FinalPositions),
747           length(FinalPositions, Num_FinalPositions),
748           Num_FinalPositions >= 2,
749           wave_wash_crabs(Board, FinalBoard).
750
751   % initial position is a board limit
752   wave(Board, Init_Pos, FinalBoard):-
753           \+ (get_rock(Init_Pos, Board, _)), % empty pile
754           limit(Board_Limits),
755           member(Init_Pos, Board_Limits),
756           get_empty_water_limits(Board, Init_Pos, WLimit),
757           water_limit(WLimits),
758           delete(WLimits, WLimit, New_WLimits),
759           get_final_position_limits(Board, [Init_Pos], New_WLimits, FinalPositions),
760           length(FinalPositions, Num_FinalPositions),
761           Num_FinalPositions >= 2,
762           wave_wash_crabs(Board, FinalBoard).
763
764
765   /* ********************************************* */
766   /*                                               */
767   /*   get_occupied_rocks/4                        */
768   /*      +Arg 1: board                            */
769   /*      +Arg 2: counter                          */
770   /*      +Arg 3: occupied rocks counter           */
771   /*      -Arg 4: occupied rocks                   */
772   /*   Summary: Gets the rocks with crabs.         */
773   /*                                               */
774   /* ********************************************* */
775
776   get_occupied_rocks(_, 19, RocksCounter, OccupiedRocks):-
777           reverse(RocksCounter, OccupiedRocks).
778
779   get_occupied_rocks(Board, Counter, RocksCounter, OccupiedRocks):-
780           get_rock(Counter, Board, _),
781           New_Counter is Counter + 1,
782           get_occupied_rocks(Board, New_Counter, [Counter | RocksCounter],
              ↪  OccupiedRocks), !.
783
784   get_occupied_rocks(Board, Counter, RocksCounter, OccupiedRocks):-
785           \+(get_rock(Counter, Board, _)),
786           New_Counter is Counter + 1,
787           get_occupied_rocks(Board, New_Counter, RocksCounter, OccupiedRocks), !.
788
789
790   /* ********************************************* */
791   /*                                               */
792   /*   all_neighbors/3                             */
793   /*      +Arg 1: group of rocks                   */
794   /*      +Arg 2: counter for neighbors of the rocks  */
795   /*      -Arg 3: neighbors                        */
796   /*   Summary: Determines all the neighbors of a     */
797   /*            group of rocks.                    */
```

```
798  /*                                            */
799  /* ****************************************** */
800
801  all_neighbors([], Neighbors, Neighbors).
802
803  all_neighbors([HGroup|TGroup], Neighbors_Counter, Neighbors):-
804          dist('B', HGroup, NeighborsGroup),
805          append(NeighborsGroup, Neighbors_Counter, New_Neighbors_Counter),
806          all_neighbors(TGroup, New_Neighbors_Counter, Neighbors),!.
807
808
809  /* ****************************************** */
810  /*                                            */
811  /*   create_groups/5                          */
812  /*      +Arg 1: occupied rocks                */
813  /*      +Arg 2: counter for group 1           */
814  /*      +Arg 3: counter for group 2           */
815  /*      -Arg 4: crab group 1                  */
816  /*      -Arg 5: crab group 2                  */
817  /*   Summary: Creates board 2 groups of crabs */
818  /*            given a split line.             */
819  /*                                            */
820  /* ****************************************** */
821
822  create_groups([], Group1, Group2, Group1, Group2).
823
824  create_groups([HOccupiedRocks|TOccupiedRocks], Counter1, Counter2, Group1, Group2):-
825          all_neighbors(Counter1, [], GroupNeighbors),
826          member(HOccupiedRocks, GroupNeighbors),
827          append(Counter1, [HOccupiedRocks], New_Counter1),
828          create_groups(TOccupiedRocks, New_Counter1, Counter2, Group1, Group2), !.
829
830  create_groups([HOccupiedRocks|TOccupiedRocks], Counter1, Counter2, Group1, Group2):-
831          all_neighbors(Counter2, [], GroupNeighbors),
832          length(Counter2, N),
833          (member(HOccupiedRocks, GroupNeighbors); N == 0),
834          append(Counter2, [HOccupiedRocks], New_Counter2),
835          create_groups(TOccupiedRocks, Counter1, New_Counter2, Group1, Group2), !.
836
837  create_groups([HOccupiedRocks|TOccupiedRocks], Counter1, Counter2, Group1, Group2):-
838          all_neighbors(Counter1, [], GroupNeighbors1),
839          all_neighbors(Counter2, [], GroupNeighbors2),
840          \+ member(HOccupiedRocks, GroupNeighbors1),
841          \+ member(HOccupiedRocks, GroupNeighbors2),
842          append(TOccupiedRocks, [HOccupiedRocks], ToLastOccupied),
843          create_groups(ToLastOccupied, Counter1, Counter2, Group1, Group2), !.
844
845
846  /* ****************************************** */
847  /*                                            */
848  /*   split_groups/3                           */
849  /*      +Arg 1: board                         */
850  /*      -Arg 2: crab group 1                  */
851  /*      -Arg 3: crab group 2                  */
852  /*   Summary: Splits the board into 2 groups of */
853  /*            crabs given a split line.       */
854  /*                                            */
855  /* ****************************************** */
856
857  split_groups(Board, Group1, Group2):-
858          get_occupied_rocks(Board, 1, [], [FirstRock|OccupiedRocks]),
859          create_groups(OccupiedRocks, [FirstRock], [], Group1, Group2).
860
861
862  /* ****************************************** */
863  /*                                            */
864  /*   count_group_crabs/4                      */
865  /*      +Arg 1: board                         */
866  /*      +Arg 2: crab group                    */
867  /*      +Arg 3: counter                       */
868  /*      -Arg 4: number of crabs in the group  */
869  /*   Summary: Counts the number of crabs in the */
870  /*            group.                          */
871  /*                                            */
872  /* ****************************************** */
873
```

```
874  count_group_crabs(_, [], Num, Num).
875
876  count_group_crabs(Board, [HGroup | TGroup], Counter, Num):-
877          count_stack(HGroup, Board, NumCrabs),
878          New_Counter is Counter + NumCrabs,
879          count_group_crabs(Board, TGroup, New_Counter, Num).
880
881
882  /* ********************************************** */
883  /*                                                */
884  /*   ask_wave/3                                   */
885  /*      +Arg 1: crab group 1                      */
886  /*      +Arg 2: crab group 2                      */
887  /*      -Arg 3: selected crab group               */
888  /*   Summary: The player selects crab group.      */
889  /*                                                */
890  /* ********************************************** */
891
892  ask_wave(Group1, Group2, Selected_Group):-
893          repeat,
894          print('1 - '), print(Group1), nl,
895          print('2 - '), print(Group2), nl,
896          print('Select group to wash away: '),
897          read(Group),
898          integer(Group),
899          (Group == 1, Selected_Group = Group1; Group == 2, Selected_Group = Group2).
900
901
902  /* ********************************************** */
903  /*                                                */
904  /*   weakest_group/4                              */
905  /*      +Arg 1: board                             */
906  /*      +Arg 2: crab group 1                      */
907  /*      +Arg 3: crab group 2                      */
908  /*      -Arg 4: weakest crab group                */
909  /*   Summary: selects the weakest crab group.     */
910  /*                                                */
911  /* ********************************************** */
912
913  weakest_group(_Board, Group1, Group2, Group2):-
914          length(Group1, Num_Group1),
915          length(Group2, Num_Group2),
916          Num_Group1 > Num_Group2.
917
918  weakest_group(_Board, Group1, Group2, Group1):-
919          length(Group1, Num_Group1),
920          length(Group2, Num_Group2),
921          Num_Group1 < Num_Group2.
922
923  weakest_group(Board, Group1, Group2, Group2):-
924          length(Group1, Num_Group1),
925          length(Group2, Num_Group2),
926          Num_Group1 == Num_Group2,
927          count_group_crabs(Board, Group1, 0, Num_Crabs1),
928          count_group_crabs(Board, Group2, 0, Num_Crabs2),
929          Num_Crabs1 > Num_Crabs2.
930
931  weakest_group(Board, Group1, Group2, Group1):-
932          length(Group1, Num_Group1),
933          length(Group2, Num_Group2),
934          Num_Group1 == Num_Group2,
935          count_group_crabs(Board, Group1, 0, Num_Crabs1),
936          count_group_crabs(Board, Group2, 0, Num_Crabs2),
937          Num_Crabs1 < Num_Crabs2.
938
939  weakest_group(Board, Group1, Group2, Selected_Group):-
940          length(Group1, Num_Group1),
941          length(Group2, Num_Group2),
942          Num_Group1 == Num_Group2,
943          count_group_crabs(Board, Group1, 0, Num_Crabs1),
944          count_group_crabs(Board, Group2, 0, Num_Crabs2),
945          Num_Crabs1 == Num_Crabs2,
946          ask_wave(Group1, Group2, Selected_Group).
947
948
949  /* ********************************************** */
```

```
950  /*                                                  */
951  /*   remove_wave/3                                  */
952  /*      +Arg 1: actual board                        */
953  /*      +Arg 2: group to remove                     */
954  /*      -Arg 3: final board                         */
955  /*   Summary: Removes a group of crabs of a given   */
956  /*            board                                 */
957  /*                                                  */
958  /* ************************************************ */
959
960  remove_wave(FinalBoardWave, [], FinalBoardWave).
961
962  remove_wave(Board, [HRock|TRock], FinalBoardWave):-
963          remove_stack_board(Board, HRock, FinalBoard),
964          remove_wave(FinalBoard, TRock, FinalBoardWave).
965
966
967  /* ************************************************ */
968  /*                                                  */
969  /*   wave_wash_crabs/2                              */
970  /*      +Arg 1: board                               */
971  /*      -Arg 2: final board                         */
972  /*   Summary: Washes away the smallest crab group.  */
973  /*                                                  */
974  /* ************************************************ */
975
976  wave_wash_crabs(Board, FinalBoard):-
977          split_groups(Board, Group1, Group2),
978          weakest_group(Board, Group1, Group2, Weakest),
979          remove_wave(Board, Weakest, FinalBoard).
980
981
982  /* ************************************************ */
983  /*                                                  */
984  /*   empty_neighbors/4                              */
985  /*      +Arg 1: board                               */
986  /*      +Arg 2: neighbors positions                 */
987  /*      +Arg 3: counter for empty positions         */
988  /*      -Arg 4: empty neighbor positions            */
989  /*   Summary: Determines the empty neighbor         */
990  /*            positions.                            */
991  /*                                                  */
992  /* ************************************************ */
993
994  empty_neighbors(_, [], Count, Empties):-
995          reverse(Count, Empties).
996
997  empty_neighbors(Board, [HPositions | TPositions], Count, Empties):-
998          \+ (get_rock(HPositions, Board, _)),
999          empty_neighbors(Board, TPositions, [HPositions|Count], Empties), !.
1000
1001 empty_neighbors(Board, [HPositions | TPositions], Count, Empties):-
1002         get_rock(HPositions, Board, _),
1003         empty_neighbors(Board, TPositions, Count, Empties), !.
1004
1005
1006 /* ************************************************ */
1007 /*                                                  */
1008 /*               Auxiliar Functions                 */
1009 /*                                                  */
1010 /* ************************************************ */
1011
1012
1013 /* ************************************************ */
1014 /*   color/2                                        */
1015 /*      +Arg 1: the color (g or b)                  */
1016 /*      -Arg 2: text representation                 */
1017 /*   Summary: Returns a string representation of    */
1018 /*            the color.                            */
1019 /*                                                  */
1020 /* ************************************************ */
1021
1022 color(g, 'Green').
1023 color(b, 'Blue').
1024
1025
```

```
1026  /* ********************************************** */
1027  /*                                                */
1028  /*   print_winner/1                               */
1029  /*      +Arg 1: the winner                        */
1030  /*   Summary: A utility function to print the     */
1031  /*            winner of the game.                 */
1032  /*                                                */
1033  /* ********************************************** */
1034
1035  print_winner(Winner) :-
1036          color(Winner, Winning_Color),
1037          random(1, 7, X),
1038          msg(X, S),
1039          format('~2n~s ~s wins.', [S, Winning_Color]),
1040          nl,
1041          pressEnterToContinue.
1042
1043
1044  /* ********************************************** */
1045  /*                                                */
1046  /*   print_board/{1/2}                            */
1047  /*      +Arg 1: the hex board                     */
1048  /*      +Arg 2: the tile color                    */
1049  /*   Summary: Simply a utility function to otput  */
1050  /*            the board and ask for input.        */
1051  /*            board.                              */
1052  /*                                                */
1053  /* ********************************************** */
1054
1055  print_board(Board) :-
1056          nl, nl,
1057          display_board(Board),
1058          flush_output(user_output).
1059
1060
1061  /* ********************************************** */
1062  /*                                                */
1063  /*   msg/2                                        */
1064  /*      +Arg 1: a number                          */
1065  /*      -Arg 2: end game message                  */
1066  /*   Summary: Used to generate a random end-game  */
1067  /*            message to make the game more fun.  */
1068  /*            board.                              */
1069  /*                                                */
1070  /* ********************************************** */
1071
1072  msg(1, 'Close call, but').
1073  msg(2, 'Well played,').
1074  msg(3, 'Congratulations,').
1075  msg(4, 'Annihalation,').
1076  msg(5, 'That was a slaughter,').
1077  msg(6, 'Pwnt. n00b l0lz.').
1078
1079
1080  /* ********************************************** */
1081  /*                                                */
1082  /*   next_color                                   */
1083  /*      +Arg 1: current color                     */
1084  /*      -Arg 2: next coloressage                  */
1085  /*   Summary: Gives the next color that will play.*/
1086  /*                                                */
1087  /* ********************************************** */
1088
1089  next_color(g, b).
1090  next_color(b, g).
1091
1092
1093  /* ********************************************** */
1094  /*                                                */
1095  /*   crab_stats/3                                 */
1096  /*      +Arg 1: Crab                              */
1097  /*      -Arg 2: Crab size                         */
1098  /*      -Arg 3: Crab color                        */
1099  /*   Summary: determines the crab size and color  */
1100  /*                                                */
1101  /* ********************************************** */
```

```
1102
1103  crab_stats(Crab, 'S', g):-
1104          Crab == 'S1'.
1105
1106  crab_stats(Crab, 'M', g):-
1107          Crab == 'M1'.
1108
1109  crab_stats(Crab, 'B', g):-
1110          Crab == 'B1'.
1111
1112  crab_stats(Crab, 'S', b):-
1113          Crab == 'S2'.
1114
1115  crab_stats(Crab, 'M', b):-
1116          Crab == 'M2'.
1117
1118  crab_stats(Crab, 'B', b):-
1119          Crab == 'B2'.
1120
1121
1122  /* ********************************************** */
1123  /*                  End of program                */
1124  /* ********************************************** */
```

## B.2 cs_ai.pl

```
/* ******************************************** */
/*                                              */
/*              Artificial Inteligence          */
/*                                              */
/* ******************************************** */

:- use_module(library(lists)).
:- [cs_utilities].

/* ******************************************** */
/*                                              */
/*   number_tile_crab/6                         */
/*      +Arg 1: the board                       */
/*      +Arg 2: player                          */
/*      +Arg 3: crab type                       */
/*      +Arg 4: position of the rock            */
/*      +Arg 5: counter for the crabs           */
/*      -Arg 6: number of type crabs on top of the  */
/*              stacks.                         */
/*   Summary: Determines the number of type crabs   */
/*            on the top of the stacks.         */
/*                                              */
/* ******************************************** */

number_tile_crab(_, _, _, 19, Total_Number, Total_Number).

number_tile_crab(Board, Player, Type, Pos, Counter, Total_Number):-
        get_rock(Pos, Board, Crab),
        crab_stats(Crab, Crab_Size, Crab_Color),
        Crab_Color == Player, %% Checks if the crab belongs to the player
        Crab_Size == Type,
        New_Counter is Counter + 1,
        New_Pos is Pos + 1,
        number_tile_crab(Board, Player, Type, New_Pos, New_Counter, Total_Number), !.

number_tile_crab(Board, Player, Type, Pos, Counter, Total_Number):-
        New_Pos is Pos + 1,
        number_tile_crab(Board, Player, Type, New_Pos, Counter, Total_Number), !.


/* ******************************************** */
/*                                              */
/*   evaluate_board/3                           */
/*      +Arg 1: the board                       */
/*      +Arg 2: player                          */
/*      -Arg 3: value of the current board      */
/*   Summary: Determines the value of the current   */
/*            board.                            */
/*                                              */
/* ******************************************** */

evaluate_board(Board, Player, Value):-
        moves_left(Board, Player, 0, [], 1, _Moves, List_Moves),
        length(List_Moves, NumMoves),
        number_tile_crab(Board, Player, 'B', 1, 0, Number_Bigs),
        number_tile_crab(Board, Player, 'M', 1, 0, Number_Mediums),
        number_tile_crab(Board, Player, 'S', 1, 0, Number_Smalls),
        Value is NumMoves + Number_Smalls + 2*Number_Mediums + 3*Number_Bigs.


/* ******************************************** */
/*                                              */
/*   move_computer/4                            */
/*      +Arg 1: the board                       */
/*      +Arg 2: player                          */
/*      +Arg 3: depth                           */
/*      -Arg 4: final board                     */
/*   Summary: Performes a move by the computer.     */
/*                                              */
/* ******************************************** */

move_computer(Board, Player, Depth, Final_Board) :-
        alpha_beta(Board, Player, Depth, -200, 200, [Init_Pos, Final_Pos], _),
        get_rock(Init_Pos, Board, Crab),
```

```
75          update_board(Board, Final_Pos, Init_Pos, Crab, Final_Board),
76          nl,
77          print('Computer moved crab from '),
78          print(Init_Pos),
79          print(' to '),
80          print(Final_Pos), nl, nl.
81
82
83  /* ********************************************** */
84  /*                                                */
85  /*   alpha_beta/7                                 */
86  /*      +Arg 1: the board                         */
87  /*      +Arg 2: player                            */
88  /*      +Arg 3: depth                             */
89  /*      +Arg 4: alpha                             */
90  /*      +Arg 5: beta                              */
91  /*      -Arg 6: movement to be done               */
92  /*      -Arg 7: value of the movement             */
93  /*   Summary: Determines the best movement for a  */
94  /*            player given a certain depth.       */
95  /*                                                */
96  /* ********************************************** */
97
98  alpha_beta(Board, Player, 0, _Alpha, _Beta, _NoMove, Value) :-
99     evaluate_board(Board, Player, Value).
100
101 alpha_beta(Board, Player, Depth, Alpha, Beta, Move, Value) :-
102    Depth > 0,
103    moves_left(Board, Player, 0, [], 1, _Moves, List_Moves),
104    Alpha1 is -Beta, %max/min
105    Beta1 is -Alpha,
106    New_Depth is Depth - 1, %profundidade
107    evaluate_and_choose(Board, Player, List_Moves, New_Depth, Alpha1, Beta1, nil, (Move,
       ↪   Value)).
108
109
110 /* ********************************************** */
111 /*                                                */
112 /*   evaluate_and_choose/8                        */
113 /*      +Arg 1: the board                         */
114 /*      +Arg 2: player                            */
115 /*      +Arg 3: list of moves                     */
116 /*      +Arg 4: depth                             */
117 /*      +Arg 5: alpha                             */
118 /*      +Arg 6: beta                              */
119 /*      +Arg 7: record of the actual best movement */
120 /*      -Arg 8: best movement                     */
121 /*   Summary: Determines the best movement for a   */
122 /*            player given a certain depth and the */
123 /*            possible player's moves.            */
124 /*                                                */
125 /* ********************************************** */
126
127 evaluate_and_choose(Board, Player, [[Pos_Init, Pos_Final] | Moves], Depth, Alpha, Beta,
       ↪   Record, BestMove) :-
128        get_rock(Pos_Init, Board, Crab_Top),
129        update_board(Board, Pos_Final, Pos_Init, Crab_Top, FinalBoard),
130        other_player(Player, OtherPlayer),
131        alpha_beta(FinalBoard, OtherPlayer, Depth, Alpha, Beta, _OtherMove, Value),
132        Value1 is -Value,
133        cutoff(Board, Player, [Pos_Init, Pos_Final], Value1, Depth, Alpha, Beta, Moves,
           ↪   Record, BestMove).
134
135 evaluate_and_choose(_Board, _Player, [], _Depth, Alpha, _Beta, Move, (Move, Alpha)).
136
137
138 /* ********************************************** */
139 /*                                                */
140 /*   cutoff/10                                    */
141 /*      +Arg 1: the board                         */
142 /*      +Arg 2: player                            */
143 /*      +Arg 3: move                              */
144 /*      +Arg 4: value of the move                 */
145 /*      +Arg 5: depth                             */
146 /*      +Arg 6: alpha                             */
147 /*      +Arg 7: beta                              */
```

```
148  /*      +Arg 8: list of moves                        */
149  /*      +Arg 9: record of the actual best movement   */
150  /*      -Arg 10: best movement and it's value        */
151  /*   Summary: Implements the cut off of the minimax */
152  /*            algorithm.                            */
153  /*                                                  */
154  /* ************************************************ */
155
156  cutoff(_Board, _Player, Move, Value, _Depth, _Alpha, Beta, _Moves, _Record,
     ↪   (Move,Value)) :-
157     Value >= Beta, !.
158
159  cutoff(Board, Player, Move, Value, Depth, Alpha, Beta, Moves, _Record, BestMove) :-
160     Alpha < Value, Value < Beta, !,
161     evaluate_and_choose(Board, Player, Moves, Depth, Value, Beta, Move, BestMove).
162
163  cutoff(Board, Player, _Move, Value, Depth, Alpha, Beta, Moves, Record, BestMove) :-
164     Value =< Alpha, !,
165     evaluate_and_choose(Board, Player, Moves, Depth, Alpha, Beta, Record, BestMove).
166
167
168  /* ************************************************ */
169  /*                                                  */
170  /*   other_player/2                                 */
171  /*      +Arg 1: current player                      */
172  /*      +Arg 2: next player                         */
173  /*   Summary: Determines the next player.           */
174  /*                                                  */
175  /* ************************************************ */
176
177  other_player(g, b).
178  other_player(b, g).
179
180
181  /* ************************************************ */
182  /*                                                  */
183  /*   print_level_CvsC/2                             */
184  /*      -Arg 1: difficulty of player 1              */
185  /*      -Arg 2: difficulty of player 2              */
186  /*   Summary: Interface for the players difficulty. */
187  /*                                                  */
188  /* ************************************************ */
189
190  print_level_CvsC(Mode1, Mode2):-
191          repeat,
192          print(' Choose Difficulty '),nl,
193          print('1 - Easy'), nl,
194          print('2 - Normal'), nl,
195          print('3 - Hard'), nl,
196          print('Select computer 1''s difficulty: '),
197          read(Mode1),
198          print('Select computer 2''s difficulty: '),
199          read(Mode2),
200          integer(Mode1),
201          integer(Mode2),
202          Mode1 > 0,
203          Mode1 < 4,
204          Mode2 > 0,
205          Mode2 < 4.
206
207
208  /* ************************************************ */
209  /*                                                  */
210  /*   print_level_HvsC/2                             */
211  /*      -Arg 1: difficulty of player 1              */
212  /*   Summary: Interface for the players difficulty. */
213  /*                                                  */
214  /* ************************************************ */
215
216  print_level_HvsC(Mode):-
217          repeat,
218          print(' Choose Difficulty '),nl,
219          print('1 - Easy'), nl,
220          print('2 - Normal'), nl,
221          print('3 - Hard'), nl,
222          print('Select computer''s difficulty: '),
```

```
223           read(Mode),
224           integer(Mode),
225           Mode > 0,
226           Mode < 4.
227
228
229   /* ******************************************** */
230   /*                                              */
231   /*   ask_level/{1,2}                            */
232   /*      -Arg 1: difficulty of player 1          */
233   /*      -Arg 2: difficulty of player 2          */
234   /*   Summary: Asks for the players difficulty.  */
235   /*                                              */
236   /* ******************************************** */
237
238   ask_level(Depth1, Depth2):-
239           print_level_CvsC(Depth1, Depth2).
240
241   ask_level(Depth):-
242           print_level_HvsC(Depth).
243
```

## B.3 cs_board.pl

```
:- use_module(library(lists)).
:- use_module(library(random)).
:- [cs_utilities].

/* ******************************************** */
/*                                              */
/*   init_board/{1}                             */
/*                                              */
/*   Summary: This function starts the game.    */
/*                                              */
/* ******************************************** */

crabs(['S1', 'S1', 'S1', 'S2', 'S2', 'S2', 'M1', 'M1', 'M1', 'M2', 'M2', 'M2', 'B1',
 ↪   'B1', 'B1', 'B2', 'B2', 'B2']).


aux_board(Board, Crab, FinalBoard):-
        repeat,
        random(1, 19, Rock),
        \+ (get_rock(Rock, Board, _)),
        add_crab_board(Board, Rock, Crab, FinalBoard).


add_crab_init_board(Board, [], Board).

add_crab_init_board(Board, [HCrabs | TCrabs], FinalBoard):-
        aux_board(Board, HCrabs, Tmp_FinalBoard),
        add_crab_init_board(Tmp_FinalBoard, TCrabs, FinalBoard), !.


init_board(Board):-
        Empty_Board = [ [[],[],[]],
                [[],[],[],[]],
                [[],[],[],[]],
                [[],[],[],[]],
                [[],[],[]]
         ],
        crabs(CrabsList),
        add_crab_init_board(Empty_Board, CrabsList, Board).


draw_space(0).
draw_space(N):- N>0,write(' '), N1 is N-1,draw_space(N1).

draw_top:-
        write('____').

/* ******************************************** */
/*                                              */
/*   display_board/1                            */
/*      +Arg 1: board                           */
/*   Summary: Prints a given board.             */
/*                                              */
/* ******************************************** */

display_board(Board):-
        draw_space(16),
        draw_top,
        nl,
        draw_space(15), write('/3   \\'),draw_space(20),write('| 1:
 ↪   '),display_stack(1,Board),
        nl,
        draw_space(10), draw_top, write('/      \\'),draw_top, draw_space(15),write('|
 ↪   2:  '),display_stack(2,Board),
        nl,
        draw_space(9), write('/2   \\'),check_rock(3, Board),write('/7   \\'),
 ↪   draw_space(14),write('| 3:  '),display_stack(3,Board),
        nl,
        draw_space(4),  draw_top, write('/      \\'),draw_top,write('/
 ↪   \\'),draw_top, draw_space(9),write('| 4:  '),display_stack(4,Board),
        nl,
        draw_space(3), write('/1   \\'),check_rock(2,Board),write('/6
 ↪   \\'),check_rock(7, Board),write('/11  \\'),draw_space(8),write('| 5:
 ↪   '),display_stack(5,Board),
```

```
68          nl,
69          draw_space(2), write('/      \\'), draw_top,write('/
             ↪   \\'),draw_top,write('/      \\'),draw_space(7),write('| 6:
             ↪   '),display_stack(6,Board),
70          nl,
71          draw_space(2),write('\\'), check_rock(1, Board), write('/5   \\'),check_rock(6,
             ↪   Board),write('/10  \\'),check_rock(11,
             ↪   Board),write('/'),draw_space(7),write('| 7:  '),display_stack(7,Board),
72          nl,
73          draw_space(3),write('\\'), draw_top,write('/      \\'),draw_top,write('/
             ↪   \\'),draw_top,write('/'),draw_space(8),write('| 8:
             ↪   '),display_stack(8,Board),
74          nl,
75          draw_space(3), write('/4   \\'),check_rock(5, Board),write('/
             ↪   \\'),check_rock(10,Board),write('/15  \\'),draw_space(8),write('| 9:
             ↪   '),display_stack(9,Board),
76          nl,
77          draw_space(2), write('/      \\'), draw_top,write('/
             ↪   \\'),draw_top,write('/      \\'),draw_space(7),write('| 10:
             ↪   '),display_stack(10,Board),
78          nl,
79          draw_space(2),write('\\'), check_rock(4, Board), write('/9   \\
             ↪   '),write('/14  \\'),check_rock(15,
             ↪   Board),write('/'),draw_space(7),write('| 11: '),display_stack(11,Board),
80          nl,
81          draw_space(3),write('\\'), draw_top,write('/      \\'),draw_top,write('/
             ↪   \\'),draw_top,write('/'),draw_space(8),write('| 12:
             ↪   '),display_stack(12,Board),
82          nl,
83          draw_space(3), write('/8   \\'),check_rock(9, Board),write('/13
             ↪   \\'),check_rock(14, Board),write('/18  \\'),draw_space(8),write('| 13:
             ↪   '),display_stack(13,Board),
84          nl,
85          draw_space(2), write('/      \\'), draw_top,write('/
             ↪   \\'),draw_top,write('/      \\'),draw_space(7),write('| 14:
             ↪   '),display_stack(14,Board),
86          nl,
87          draw_space(2),write('\\'), check_rock(8, Board), write('/12
             ↪   \\'),check_rock(13, Board), write('/17  \\'),check_rock(18,
             ↪   Board),write('/'),draw_space(7),write('| 15: '),display_stack(15,Board),
88          nl,
89          draw_space(3), write('\\'),draw_top, write('/      \\'),draw_top,write('/
             ↪   \\'),draw_top,write('/'),draw_space(8),write('| 16:
             ↪   '),display_stack(16,Board),
90          nl,
91          draw_space(8),write('\\'), check_rock(12, Board), write('/16
             ↪   \\'),check_rock(17, Board),write('/'),draw_space(13),write('| 17:
             ↪   '),display_stack(17,Board),
92          nl,
93          draw_space(9),write('\\'), draw_top, write('/
             ↪   \\'),draw_top,write('/'),draw_space(14),write('| 18:
             ↪   '),display_stack(18,Board),
94          nl,
95          draw_space(14),write('\\'),check_rock(16, Board),write('/'),
96          nl,
97          draw_space(15),write('\\'),draw_top,write('/'),
98          nl.
99
100
101 /* ********************************************* */
102 /*                                               */
103 /*   check_rock/2                                */
104 /*      +Arg 1: the rock number                  */
105 /*      +Arg 2: the game board                   */
106 /*   Summary: Print the top of the pile in the rock */
107 /*            if the rock is empty prints spaces */
108 /*                                               */
109 /* ********************************************* */
110
111 check_rock(Rock, Board):-
112         \+ get_rock(Rock, Board, _),
113         write('      '). % empty rock
114
115 check_rock(Rock, Board):-
116         get_rock(Rock, Board, Tile),
117         write('  '),
```

```
118         write(Tile),
119         atom_length(Tile, Size),
120         White is 4 - Size,
121         draw_space(White).
122
123
124 /* ********************************************* */
125 /*                                               */
126 /*   get_rock/3                                  */
127 /*      +Arg 1: the rock number                  */
128 /*      +Arg 2: the game board                   */
129 /*      -Arg 3: the crab                         */
130 /*   Summary: Gets the crab on top of the pile   */
131 /*            on the rock                        */
132 /*                                               */
133 /* ********************************************* */
134
135 get_rock(Rock, [H|_], Tile):-
136         (Rock == 1; Rock == 2; Rock == 3),
137         nth1(Rock, H, E),
138         \+ length(E, 0),
139         nth1(1, E, Tile).
140
141 get_rock(Rock, Board, Tile):-
142         (Rock == 4; Rock == 5; Rock == 6; Rock == 7),
143         nth1(2, Board, H),
144         Index is Rock - 3,
145         nth1(Index, H, E),
146         \+ length(E, 0),
147         nth1(1, E, Tile).
148
149 get_rock(Rock, Board, Tile):-
150         (Rock == 8; Rock == 9; Rock == 10; Rock == 11),
151         nth1(3, Board, H),
152         Index is Rock - 7,
153         nth1(Index, H, E),
154         \+ length(E, 0),
155         nth1(1, E, Tile).
156
157 get_rock(Rock, Board, Tile):-
158         (Rock == 12; Rock == 13; Rock == 14; Rock == 15),
159         nth1(4, Board, H),
160         Index is Rock - 11,
161         nth1(Index, H, E),
162         \+ length(E, 0),
163         nth1(1, E, Tile).
164
165 get_rock(Rock, Board, Tile):-
166         (Rock == 16; Rock == 17; Rock == 18),
167         nth1(5, Board, H),
168         Index is Rock - 15,
169         nth1(Index, H, E),
170         \+ length(E, 0),
171         nth1(1, E, Tile).
172
173
174 /* ********************************************* */
175 /*                                               */
176 /*   display_stack/2                             */
177 /*      +Arg 1: the rock number                  */
178 /*      +Arg 2: the game board                   */
179 /*   Summary: Prints the stack on a given rock.  */
180 /*                                               */
181 /* ********************************************* */
182
183 display_stack(Rock, [H|_]):-
184         (Rock == 1; Rock == 2; Rock == 3),
185         nth1(Rock, H, E),
186         reverse(E, Stack),
187         printlist(Stack).
188
189 display_stack(Rock, Board):-
190         (Rock == 4; Rock == 5; Rock == 6; Rock == 7),
191         nth1(2, Board, H),
192         Index is Rock - 3,
193         nth1(Index, H, E),
```

```
194         reverse(E, Stack),
195         printlist(Stack).
196
197 display_stack(Rock, Board):-
198         (Rock == 8; Rock == 9; Rock == 10; Rock == 11),
199         nth1(3, Board, H),
200         Index is Rock - 7,
201         nth1(Index, H, E),
202         reverse(E, Stack),
203         printlist(Stack).
204
205 display_stack(Rock, Board):-
206         (Rock == 12; Rock == 13; Rock == 14; Rock == 15),
207         nth1(4, Board, H),
208         Index is Rock - 11,
209         nth1(Index, H, E),
210         reverse(E, Stack),
211         printlist(Stack).
212
213 display_stack(Rock, Board):-
214         (Rock == 16; Rock == 17; Rock == 18),
215         nth1(5, Board, H),
216         Index is Rock - 15,
217         nth1(Index, H, E),
218         reverse(E, Stack),
219         printlist(Stack).
220
221
222 /* ********************************************* */
223 /*                                               */
224 /*   count_stack/3                               */
225 /*      +Arg 1: the rock number                  */
226 /*      +Arg 2: the game board                   */
227 /*      +Arg 3: number of crabs on the rock      */
228 /*   Summary: Counts the number of crabs on a given */
229 /*            rock.                              */
230 /*                                               */
231 /* ********************************************* */
232
233 count_stack(Rock, [H|_], Count):-
234         (Rock == 1; Rock == 2; Rock == 3),
235         nth1(Rock, H, E),
236         length(E, Count).
237
238 count_stack(Rock, Board, Count):-
239         (Rock == 4; Rock == 5; Rock == 6; Rock == 7),
240         nth1(2, Board, H),
241         Index is Rock - 3,
242         nth1(Index, H, E),
243         length(E, Count).
244
245 count_stack(Rock, Board, Count):-
246         (Rock == 8; Rock == 9; Rock == 10; Rock == 11),
247         nth1(3, Board, H),
248         Index is Rock - 7,
249         nth1(Index, H, E),
250         length(E, Count).
251
252 count_stack(Rock, Board, Count):-
253         (Rock == 12; Rock == 13; Rock == 14; Rock == 15),
254         nth1(4, Board, H),
255         Index is Rock - 11,
256         nth1(Index, H, E),
257         length(E, Count).
258
259 count_stack(Rock, Board, Count):-
260         (Rock == 16; Rock == 17; Rock == 18),
261         nth1(5, Board, H),
262         Index is Rock - 15,
263         nth1(Index, H, E),
264         length(E, Count).
265
266
267 /* ********************************************* */
268 /*                                               */
269 /*   removeCrab/4                                */
```

```
270 /*      +Arg 1: the game board                      */
271 /*      +Arg 2: row                                 */
272 /*      +Arg 3: column                              */
273 /*      -Arg 4: final board                         */
274 /*   Summary: Removes a crab on the position (R, C).*/
275 /*                                                  */
276 /* ************************************************ */
277
278 removeCrab(Board, R, C, FinalBoard) :-
279         nth1(R, Board, OldRow, RestRows),   % get the row and the rest
280         nth1(C, OldRow, Stack, NewRow),    % we don't care the _Val deleted
281         nth1(1, Stack, _Val,  NewStack),
282         nth1(C, FinalRow, NewStack, NewRow),
283         nth1(R, FinalBoard, FinalRow, RestRows).
284
285
286 /* ************************************************ */
287 /*                                                  */
288 /*   removeCrabStack/4                              */
289 /*      +Arg 1: the game board                      */
290 /*      +Arg 2: row                                 */
291 /*      +Arg 3: column                              */
292 /*      -Arg 4: final board                         */
293 /*   Summary: Removes all crabs on the position     */
294 /*            (R, C).                               */
295 /*                                                  */
296 /* ************************************************ */
297
298 removeCrabStack(Board, R, C, FinalBoard) :-
299         nth1(R, Board, OldRow, RestRows),   % get the row and the rest
300         nth1(C, OldRow, _Stack, NewRow),    % we don't care the _Val deleted
301         nth1(C, FinalRow, [], NewRow),
302         nth1(R, FinalBoard, FinalRow, RestRows).
303
304
305 /* ************************************************ */
306 /*                                                  */
307 /*   addCrab/4                                      */
308 /*      +Arg 1: the game board                      */
309 /*      +Arg 2: row                                 */
310 /*      +Arg 3: column                              */
311 /*      -Arg 4: final board                         */
312 /*   Summary: Adds a crab on the position (R, C).   */
313 /*                                                  */
314 /* ************************************************ */
315
316 addCrab(Board, R, C, Crab, FinalBoard) :-
317         nth1(R, Board, OldRow, RestRows),   % get the row and the rest
318         nth1(C, OldRow, _Val, RestRow),    % we don't care the _Val deleted
319         nth1(C, OldRow, Val),
320         nth1(1, S, Crab, Val),
321         nth1(C, NewRow, S, RestRow),
322         nth1(R, FinalBoard, NewRow, RestRows).   % insert updated row in rest, get Upd
          ↪   matrix
```

## B.4 cs_menus.pl

```
/* ******************************************** */
/*                                              */
/*                    Game Menus                */
/*                                              */
/* ******************************************** */
mainMenu:-
        printMainMenu,
        getChar(Input),
        (
                Input = '1' -> gameModeMenu, mainMenu;
                Input = '2' -> helpMenu, mainMenu;
                Input = '3' -> aboutMenu, mainMenu;
                Input = '4';

                nl,
                write('Error: invalid input.'), nl,
                pressEnterToContinue, nl,
                mainMenu
        ).

printMainMenu:-
        clearConsole,
        write('================================='), nl,
        write('=     ..:: CRAB STACK ::..      ='), nl,
        write('================================='), nl,
        write('=                               ='), nl,
        write('=   1. Play                      ='), nl,
        write('=   2. How to play               ='), nl,
        write('=   3. About                     ='), nl,
        write('=   4. Exit                      ='), nl,
        write('=                               ='), nl,
        write('================================='), nl,
        write('Choose an option:'), nl.

gameModeMenu:-
        printgameModeMenu,
        getChar(Input),
        (
                Input = '1' -> startPvPGame;
                Input = '2' -> startPvBGame;
                Input = '3' -> startBvBGame;
                Input = '4';

                nl,
                write('Error: invalid input.'), nl,
                pressEnterToContinue, nl,
                gameModeMenu
        ).

printgameModeMenu:-
        clearConsole,
        write('================================='), nl,
        write('=      ..:: Game Mode ::..      ='), nl,
        write('================================='), nl,
        write('=                               ='), nl,
        write('=   1. Player vs. Player         ='), nl,
        write('=   2. Player vs. Computer       ='), nl,
        write('=   3. Computer vs. Computer     ='), nl,
        write('=   4. Back                      ='), nl,
        write('=                               ='), nl,
        write('================================='), nl,
        write('Choose an option:'), nl.

startPvPGame:-
        playGamePvP.
startPvBGame:-
        playGamePvB.
startBvBGame:-
        playGameBvB.

helpMenu:-
        clearConsole,
        write('================================================================='),
        ↪  nl,
```

```
74          write('=                    ..:: How to play ::..                    ='),
             ↪  nl,
75          write('======================================================================'),
             ↪  nl,
76          write('=                                                                  ='),
             ↪  nl,
77          write('=   Crab Stack is an abstract and familiar game.                   ='),
             ↪  nl,
78          write('=                                                                  ='),
             ↪  nl,
79          write('=   Objective:                                                     ='),
             ↪  nl,
80          write('=     To be the last player who still has a crab that can be      ='),
             ↪  nl,
81          write('=     legally moved.                                               ='),
             ↪  nl,
82          write('=                                                                  ='),
             ↪  nl,
83          write('=   Turn:                                                          ='),
             ↪  nl,
84          write('=     In each turn, a player can move one of his crabs on top     ='),
             ↪  nl,
85          write('=     of another crab respecting the stack rules.                  ='),
             ↪  nl,
86          write('=                                                                  ='),
             ↪  nl,
87          write('=   Stack Rules:                                                   ='),
             ↪  nl,
88          write('=     > Small crabs only can be moved on top of another small     ='),
             ↪  nl,
89          write('=       crab.                                                      ='),
             ↪  nl,
90          write('=     > Medium crabs can be moved on top of another medium crab   ='),
             ↪  nl,
91          write('=       or small ones.                                             ='),
             ↪  nl,
92          write('=                                                 Page 1 of 3   ='),
             ↪  nl,
93          write('=                                                                  ='),
             ↪  nl,
94          write('======================================================================'),
             ↪  nl,
95          pressEnterToContinue, nl,
96
97          clearConsole,
98          write('======================================================================'),
             ↪  nl,
99          write('=                    ..:: How to play ::..                    ='),
             ↪  nl,
100         write('======================================================================'),
             ↪  nl,
101         write('=                                                                  ='),
             ↪  nl,
102         write('=     > Big crabs can be moved on top of any crab.                 ='),
             ↪  nl,
103         write('=                                                                  ='),
             ↪  nl,
104         write('=   Moves:                                                         ='),
             ↪  nl,
105         write('=     >  Small crabs must move 3 rocks.                            ='),
             ↪  nl,
106         write('=     >  Medium crabs must move 2 rocks.                           ='),
             ↪  nl,
107         write('=     >  Big Crabs must move 1 rock.                               ='),
             ↪  nl,
108         write('=                                                                  ='),
             ↪  nl,
109         write('=   Wave Rule:                                                     ='),
             ↪  nl,
110         write('=     Crabs like to stay in a large group and don\'t like to       ='),
             ↪  nl,
111         write('=     separate. When two groups of crabs are separated by a       ='),
    ↪  nl,
112         write('=     line of rocks, a wave will wash one of them:                 ='),
    ↪  nl,
```

```
113         write('=         1. The group who occupies less rocks is removed.        ='),
    ↪   nl,
114         write('=         2. If they occupy the same number of rocks, the group   ='),
    ↪   nl,
115         write('=            with less crabs is removed.                          ='),
    ↪   nl,
116         write('=                                                                 ='),
    ↪   nl,
117         write('=                                                  Page 2 of 3    ='),
    ↪   nl,
118         write('=                                                                 ='),
    ↪   nl,
119         write('=================================================================='),
    ↪   nl,
120         pressEnterToContinue, nl,
121
122         clearConsole,
123         write('=================================================================='),
    ↪   nl,
124         write('=                      ..:: How to play ::..                      ='),
    ↪   nl,
125         write('=================================================================='),
    ↪   nl,
126         write('=                                                                 ='),
    ↪   nl,
127         write('=         3. If the number of crabs in each group is the same,    ='),
    ↪   nl,
128         write('=            the player who separeted the crabs, chose the group  ='),
    ↪   nl,
129         write('=            to be removed.                                       ='),
    ↪   nl,
130         write('=                                                                 ='),
    ↪   nl,
131         write('=   End Game:                                                     ='),
    ↪   nl,
132         write('=     > If all player\'s crabs were removed from the game, the    ='),
    ↪   nl,
133         write('=       player loses.                                             ='),
            ↪   nl,
134         write('=     > At the beginning of a player\'s turn, if he cannot move   ='),
            ↪   nl,
135         write('=       a crab, the player loses.                                 ='),
    ↪   nl,
136         write('=     > If the players moves are repeatdly the same, the game     ='),
    ↪   nl,
137         write('=       ends in a tie. The players must play another game.        ='),
    ↪   nl,
138         write('=                                                                 ='),
    ↪   nl,
139         write('=                                                                 ='),
    ↪   nl,
140         write('=                                                                 ='),
    ↪   nl,
141         write('=                                                                 ='),
    ↪   nl,
142         write('=                                                  Page 3 of 3    ='),
    ↪   nl,
143         write('=                                                                 ='),
    ↪   nl,
144         write('=================================================================='),
    ↪   nl,
145         pressEnterToContinue, nl.
146
147 aboutMenu:-
148         clearConsole,
149         write('================================='), nl,
150         write('=        ..:: About ::..        ='), nl,
151         write('================================='), nl,
152         write('=                               ='), nl,
153         write('=   Authors:                    ='), nl,
154         write('=    > Inês Caldas              ='), nl,
155         write('=    > Maria Teresa Chaves      ='), nl,
156         write('=                               ='), nl,
157         write('=                               ='), nl,
158         write('================================='), nl,
159         pressEnterToContinue, nl.
```

## B.5 cs_utilities.pl

```
/* ******************************************** */
/*                                              */
/*                   Utilities                  */
/*                                              */
/* ******************************************** */

:- use_module(library(lists)).

pressEnterToContinue:-
        write('Press <Enter> to continue.'), nl,
        waitForEnter, !.


waitForEnter:-
        get_char(_).


clearConsole:-
        clearConsole(40), !.

clearConsole(0).

clearConsole(N):-
        nl,
        N1 is N-1,
        clearConsole(N1).


getChar(Input):-
        get_char(Input),
        get_char(_).

getCode(Input):-
        get_code(TempInput),
        get_code(_),
        Input is TempInput - 48.

getInt(Input):-
        get_code(TempInput),
        Input is TempInput - 48.


discardInputChar:-
        get_code(_).


printlist([]).

printlist([X]):-
       write(X).

printlist([X|List]) :-
        write(X),write(', '),
        printlist(List).


intersection([], _, []).
intersection([Head|L1tail], L2, L3) :-
        memberchk(Head, L2),
        !,
        L3 = [Head|L3tail],
        intersection(L1tail, L2, L3tail).
intersection([_|L1tail], L2, L3) :-
        intersection(L1tail, L2, L3).


delete_all(X, L, L):-
        \+ member(X, L).


delete_all(X, L, L1):-
        member(X, L),
        delete(L, X, L2),
        delete_all(X, L2, L1),!.
```

```
75
76
77  delete_all_list([], L, L).
78
79  delete_all_list([H|T], L, L2):-
80          delete_all(H, L, L3),
81          delete_all_list(T, L3, L2),!.
```