Universidade do Porto

FEUP Faculdade de Engenharia

Mestrado Integrado em Engenharia Informática e Computação

Programação em Lógica

Trabalho Prático nº1

Fields of Action

Relatório Final

3MIEIC05 – Fields of Action_1

Ângelo Daniel Pereira Mendes Moura …up201303828@fe.up.pt

César Alexandre de Costa Pinho …up201604039@fe.up.pt

15 de novembro de 2018

Fields Of Action

Introdução

Este projeto tem como objetivo a implementação do jogo de tabuleiro "*Fields of Action*" usando a linguagem de programação PROLOG (Programação Lógica).

O código desenvolvido concentrasse unicamente na vertente logica do jogo, como tal a sua *interface* é bastante simplificada, usando texto imprimido na consola como forma de comunicar o estado do tabuleiro e do jogo ao jogador.

Sendo este projeto realizado unicamente usando PROLOG, apresenta um desafio único, pois não é muito comum o uso de uma linguagem puramente logica para implementar um jogo de tabuleiro.

Descrição do jogo

História

"*Fields of Action*" é um jogo de tabuleiro de estratégia abstrata criado por Sid Sackson em 1982.

Sid Sackson foi um *designer* e colecionador Americano mais conhecido por ser o criador do jogo "*Acquire*", que desde 2011 foi introduzido, juntamente com o seu criador, na "*Academy of Adventure Gaming Arts & Design's Hall of Fame*".

"*Fields of Action*" foi inspirado num jogo da autoria de Claude Soucie chamado "*Lines of Action*", jogo este que Sid Sackson publicitou no seu livro "*A Gamut of Games*" (1969).

Detalhes do Jogo

Número de Jogadores

"*Fields of Action*" é jogado por dois jogadores que competem entre si.

Tabuleiro

"*Fields of Action*" joga-se num tabuleiro 8x8.

Neste tabuleiro há um conjunto de 12 peças para cada jogador, com cores diferentes para distinguir as peças de cada um (por exemplo: Preto e Branco). As peças de cada conjunto estão numeradas de 1 a 12.

O jogo começa com o tabuleiro no estado representado na Figura 1.

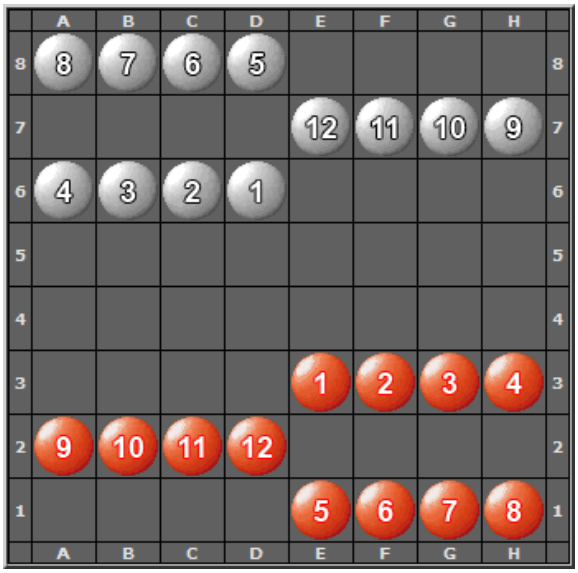


Figura 1 - Tabuleiro inicial

Objetivo

O objetivo do jogo é capturar cinco peças inimigas que estão numeradas sequencialmente, e.g. 7-8-9-10-11 (a ordem da captura não é valorizada). Um jogador também ganha se o seu oponente não conseguir realizar nenhum movimento considerado legal.

Ritmo de Jogo

Começando com o jogador que controla as peças de cor Preta, os jogadores alternam turnos até um completar o objetivo, sendo esse jogador considerado o vencedor. Ou até um jogador não conseguir realizar um movimento válido, sendo considerado o oponente desse jogador o vencedor.

Regras

1. Apenas uma peça pode ser movida por turno.
2. As peças podem mover-se horizontalmente, verticalmente ou diagonalmente.
3. Uma peça pode ser movida o número de casas igual ao número de peças, de qualquer cor, adjacentes a essa peça, antes do movimento (Figura 2).
4. A peça escolhida pode saltar por cima de peças de qualquer cor.
5. A peça escolhida não pode pousar sobre uma peça da mesma cor.
6. A peça escolhida pousar sobre uma peça de cor inimiga. A peça inimiga é então removida do tabuleiro. Isto é referido como uma captura.
7. Uma peça sem outras adjacentes não tem número limite de casas de movimento (3ª regra), no entanto tem que poisar numa casa onde fique adjacente a pelo menos duas peças de qualquer cor (Figura 3).

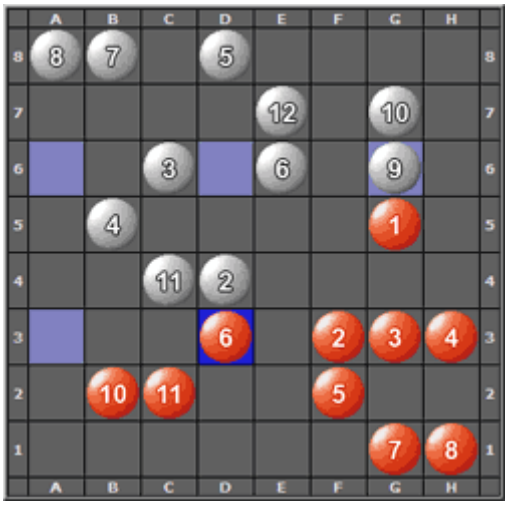
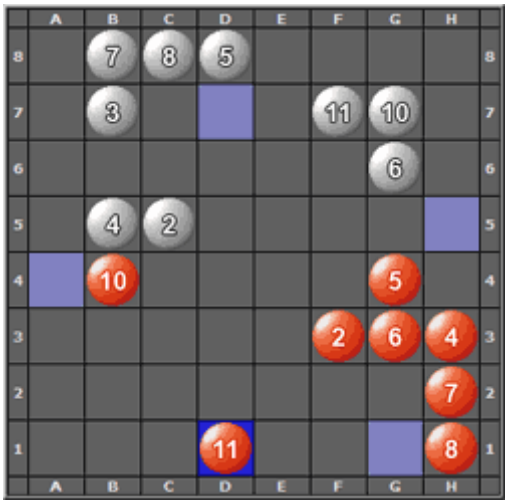


Figura 2 - O "6" Vermelho tem 3 peças adjacentes ("2" branco, "11" branco e "11" preto") por isso pode mover exatamente 3 quadrados em qualquer direção desde que não poise numa peça da mesma cor. Atenção que uma das opções nesta posição é a captura do "9" branco".

Figura 3 - O "11" Preto está isolado. Devido a isso pode mover-se em qualquer direção se poisar numa casa vazia com pelo menos duas peças adjacentes. Todas estas casas são destacadas.



Lógica do Jogo

O jogo é, em termos de código, um ciclo.

Cada iteração do ciclo representa uma jogada, ou seja, o ciclo refere-se alternadamente aos jogadores.

O ciclo começa por verificar se a condição de vitória foi alcançada, ou seja, se jogo acabou. De seguida calcula o tamanho do tabuleiro, e representa-o no ecrã. Com o tabuleiro visível, o ciclo passa a faze onde é determinada a jogada a realizar, isto pode ser introduzido, caso seja o turno do utilizador, ou calculado caso seja o turno do computador. Com a próxima jogada estabelecida, o ciclo entra na fase onde de facto realiza o movimento. Com o movimento do turno realizado, o ciclo troca de jogador e recomeça.

O ciclo descrito repete-se até a condição de vitória ser alcançada.

Representação do Estado do Jogo

Para a implementação do jogo em PROLOG, foi necessário escolher a representação de dados a utilizar para o tabuleiro e peças. Assim, foi escolhida uma representação matricial, isto é uma lista de listas, de dimensão 8x8, cujos elementos são elementos átomos e termos compostos. As células vazias do tabuleiro são representadas por um átomo (0), as peças são representadas por termos compostos onde o lado esquerdo indica o número do jogador, 1 ou 2, e o lado direito indica o número da peça em questão, entre 1 e 12, por exemplo 1-6 é a peça número 6 do jogador 1.

O tabuleiro inicial começa com as 12 peças de cada jogador dispostas, como já foi referido.

```
tab([ [1-8,1-7 ,1-6 ,1-5 , 0  , 0  , 0  , 0 ],
      [ 0 , 0  , 0  , 0  ,1-12,1-11,1-10,1-9],
      [1-4,1-3 ,1-2 ,1-1 , 0  , 0  , 0  , 0 ],
      [ 0 , 0  , 0  , 0  , 0  , 0  , 0  , 0 ],
      [ 0 , 0  , 0  , 0  , 0  , 0  , 0  , 0 ],
      [ 0 , 0  , 0  , 0  ,2-1 ,2-2 ,2-3 ,2-4],
      [2-9,2-10,2-11,2-12, 0  , 0  , 0  , 0 ],
      [ 0 , 0  , 0  , 0  ,2-5 ,2-6 ,2-7 ,2-8]]).
```

Figura 4 - Representação do tabuleiro inicial

```
tab([ [ 0 ,1-7 ,1-8 ,1-5 , 0  , 0  , 0  , 0 ],
      [ 0 ,1-3 , 0  , 0  , 0  ,1-11,1-10, 0 ],
      [ 0 , 0  , 0  , 0  , 0  , 0  ,1-6 , 0 ],
      [ 0 ,1-4 ,1-2 , 0  , 0  , 0  , 0  , 0 ],
      [ 0 ,2-10, 0  , 0  , 0  , 0  ,2-5 , 0 ],
      [ 0 , 0  , 0  , 0  , 0  ,2-2 ,2-6 ,2-4],
      [ 0 , 0  , 0  , 0  , 0  , 0  , 0  ,2-7],
      [ 0 , 0  , 0  ,2-11, 0  , 0  , 0  ,2-8]]).
```

Figura 5 - Representação de um possível tabuleiro intermédio

```
tab([ [1-8,2-12,1-6 , 0  , 0  , 0  , 0  , 0 ],
      [ 0 , 0  ,1-5 , 0  , 0  , 0  , 0  , 0 ],
      [ 0 , 0  , 0  , 0  ,1-1 , 0  , 0  , 0 ],
      [1-4, 0  , 0  , 0  , 0  , 0  ,1-11, 0 ],
      [ 0 , 0  ,2-9 , 0  ,2-8 , 0  , 0  , 0 ],
      [ 0 , 0  , 0  , 0  , 0  , 0  , 0  , 0 ],
      [ 0 , 0  ,2-11,2-10, 0  , 0  , 0  , 0 ],
      [ 0 ,1-9 , 0  , 0  , 0  ,2-6 ,2-7 , 0 ]]).
```

Figura 6 - Representação de um possível tabuleiro final

Visualização do tabuleiro

O *display* do tabuleiro de jogo na consola é da responsabilidade de um predicado que percorre a estrutura de dados que armazena o tabuleiro e imprime-o. Esse predicado, baseado na recursividade, percorre cada linha do tabuleiro invocando outro predicado que percorrerá todas as células de uma linha imprimindo um símbolo de jogo por célula, dependendo do valor da mesma.

```
    A     B     C     D     E     F     G     H
  ---------------------------------------------
8 | B8  | B7  | B6  | B5  |     |     |     |     | 8
  ---------------------------------------------
7 |     |     |     |     | B12 | B11 | B10 | B9  | 7
  ---------------------------------------------
6 | B4  | B3  | B2  | B1  |     |     |     |     | 6
  ---------------------------------------------
5 |     |     |     |     |     |     |     |     | 5
  ---------------------------------------------
4 |     |     |     |     |     |     |     |     | 4
  ---------------------------------------------
3 |     |     |     |     | W1  | W2  | W3  | W4  | 3
  ---------------------------------------------
2 | W9  | W10 | W11 | W12 |     |     |     |     | 2
  ---------------------------------------------
1 |     |     |     |     | W5  | W6  | W7  | W8  | 1
  ---------------------------------------------
    A     B     C     D     E     F     G     H
```

Figura 7 - Tabuleiro inicial

```
    A     B     C     D     E     F     G     H
  ---------------------------------------------
8 |     | B7  | B8  | B5  |     |     |     |     | 8
  ---------------------------------------------
7 |     | B3  |     |     |     | B11 | B10 |     | 7
  ---------------------------------------------
6 |     |     |     |     |     |     | B6  |     | 6
  ---------------------------------------------
5 |     | B4  | B2  |     |     |     |     |     | 5
  ---------------------------------------------
4 |     | W10 |     |     |     |     | W5  |     | 4
  ---------------------------------------------
3 |     |     |     |     |     | W2  | W6  | W4  | 3
  ---------------------------------------------
2 |     |     |     |     |     |     |     | W7  | 2
  ---------------------------------------------
1 |     |     |     | W11 |     |     |     | W8  | 1
  ---------------------------------------------
    A     B     C     D     E     F     G     H
```

Figura 8 - Tabuleiro Intermédio

```
    A     B     C     D     E     F     G     H
  ---------------------------------------------
8 | B8  | W12 | B6  |     |     |     |     |     | 8
  ---------------------------------------------
7 |     |     | B5  |     |     |     |     |     | 7
  ---------------------------------------------
6 |     |     |     |     | B1  |     |     |     | 6
  ---------------------------------------------
5 | B4  |     |     |     |     |     | B11 |     | 5
  ---------------------------------------------
4 |     |     | W9  |     | W8  |     |     |     | 4
  ---------------------------------------------
3 |     |     |     |     |     |     |     |     | 3
  ---------------------------------------------
2 |     |     | W11 | W10 |     |     |     |     | 2
  ---------------------------------------------
1 |     | B9  |     |     |     | W6  | W7  |     | 1
  ---------------------------------------------
    A     B     C     D     E     F     G     H
```

Figura 9 – Tabuleiro Final

Lista de Jogadas Válidas

O cálculo da lista das jogadas válidas é ligeiramente diferente se for a vez do jogador ou a vez do computador.

Quando é o turno do jogador, é pedido previamente a peça que se deseja mover, calculando-se apenas os movimentos válidos que essa peça pode realizar e apresenta-se a lista ao jogador para este prosseguir a seleção do movimento.

Quando é a vez de o computador realizar um movimento, são calculadas todas as jogadas possíveis de serrem realizadas nesse momento.

Outra possível divergência no cálculo de jogadas válidas acontece quando a peça a ser avaliada se encontra isolada no tabuleiro. Como neste caso o movimento das peças é diferente do normal é necessário calcular as jogadas validas para esta peça de maneira diferente também.

Predicados:

- valid_moves(Board, Player, Lines, Columns, ListOfMoves) :-
  valid_moves(Board, Player, 1, Lines, Columns, ListOfLists),
  append(ListOfLists, ListOfMoves).
- valid_moves(_, _, 13, _, _, []).
- valid_moves(Board, Player, Piece, Lines, Columns, [X|ListOfMoves]) :-
  exist_piece(Player, Board, Piece),
  possible_plays(Player, Board, Piece, Lines, Columns, X),
  NewPiece is Piece + 1,
  valid_moves(Board, Player, NewPiece, Lines, Columns, ListOfMoves).
- valid_moves(Board, Player, Piece, Lines, Columns, ListOfMoves) :-
  Piece =< 12,
  NewPiece is Piece + 1,
  valid_moves(Board, Player, NewPiece, Lines, Columns, ListOfMoves).

Execução de Jogadas

A execução das jogadas é realizada depois de ter sido selecionada uma peça valida e um movimento valido que esta pode realizar. Neste ponto da execução do jogo, é conhecido o valor da peça, onde ela se encontra inicialmente e onde a peça ficará. Sabendo isto, a célula do tabuleiro onde ela se encontra é preenchida de maneira a simbolizar uma célula vazia e a célula onde a peça acaba é preenchida com a informação revelante da peça, nomeadamente o seu controlador e o seu valor.

Caso exista outra peça na posição final, é realizada uma captura. Isto é feito colocando a informação da peça capturada numa lista.

Predicados:

- move(Jog, Board, Lines, Columns, L-C-NewL-NewC-Play, NewBoard) :-
  piece_position(Board, _-Peca, L, C, Columns),
  replace(Board, Jog-Peca, NewL, NewC, Board1),
  empty_cel(V),
  replace(Board1, V, L, C, NewBoard),
  print_move(L-C-NewL-NewC-Play, Lines),
  is_catch(Jog, Board, Columns, L-C-NewL-NewC-Play).

Final do Jogo

O jogo "*Fields of Action*" acaba quando se encontra uma de duas situações: um dos jogadores capturou cinco peças de valor sequencial ou não existem mais jogadas possíveis.

O primeiro caso e identificado verificando uma lista que contem a informação das peças capturadas até esse ponto do jogo. Caso se verifique que um jogador capturou cinco peças de valor sequencial, o jogo acaba e esse jogador é considerado o vencedor.

O segundo é verificado vendo se existem movimentos validos no estado de jogo atual, caso não existam o jogo acaba e o jogador que realizou o último movimento é considerado o vencedor.

Predicados:

- game_over(Player, Pieces) :-
  abolish(catch/1),
  new_line(1),
  write('***\n'),
  write('** VENCEDOR :: JOGADOR '),
  write(Player),
  write(' **\n'),
  write('***\n'),
  write(' PECAS CAPTURADAS :\n'),
  display_catched_pieces(Player,Pieces),
  asserta(is_game_over(true)).
- game_over(1) :-
  write('\nJOGADOR '),
  write(1),
  write(' SEM MOVIMENTOS POSSIVEIS\n'),
  pecasCapturadas([], Pieces, 2),
  sort(Pieces, OrderedPieces),
  game_over(2, OrderedPieces).
- game_over(2) :-
  write('\nJOGADOR '),
  write(2),
  write(' SEM MOVIMENTOS POSSIVEIS\n'),
  pecasCapturadas([], Pieces, 2),
  sort(Pieces, OrderedPieces),
  game_over(1, OrderedPieces).

Avaliação do Tabuleiro

A avaliação do tabuleiro é obtida avaliando o número de peças necessárias para obter uma sequência de cinco peças.

Cada possível movimento é passível de ser atribuído um valor, esse valor depende do quanto esse movimento aproxima o jogo de um estado de vitoria para o jogador atual. Ou seja, o valor de um movimento é máximo quando esse movimento completa uma sequência de cinco peças capturadas.

Predicados:

- value(Board, Player, FirstSeqPiece-LastSeqPiece, Value) :-
  pieces_out_of_game(Board, Player, Pieces),
  sort(Pieces, OrderedPieces),

```
check_sequence(OrderedPieces, FirstSeqPiece-LastSeqPiece, Value).
```

Jogada do Computador

O computador pode seguir dois diferentes métodos de escolha de movimentos.

O primeiro envolve a escolha de um movimento aleatório dos possíveis no momento.

O segundo é uma escolha informada. É atribuído um valor a cada movimento possível, e o computador realiza o melhor movimento nesse momento, havendo um empate o computador escolhe o primeiro a ser encontrado.

Predicados:

- ```
  choose_move(_, _, computer1, ListOfMoves, Move) :-
      random_member(Move, ListOfMoves).
  ```

- ```
  choose_move(Board, Player, computer2, ListOfMoves, Move) :-
      board_size(Board, Lines, Columns),
      choose_best_move(Board, Player, Lines, Columns, ListOfMoves, _, L-C-NewL-NewC-Play),
      Play =:= 0,
       random_member(Move, ListOfMoves).
  ```

- ```
  choose_move(Board, Player, computer2, ListOfMoves, Move) :-
      board_size(Board, Lines, Columns),
      choose_best_move(Board, Player, Lines, Columns, ListOfMoves, _, Move).
  ```
  ```

Conclusões

Sabíamos logo de início que a dificuldade deste projeto não originaria da complexidade do jogo em si, mas sim da linguagem usada para a implementação. PROLOG é uma linguagem muito diferente do que estamos habituados a trabalhar e certas noções e técnicas tiveram de ser reaprendidas, ao mesmo tempo novas precisaram de ser aprendidas.

Um dos maiores contribuidores para a duração do tempo de desenvolvimento do trabalho foi o facto de ser necessário a programação individual de cada caso e situação possível, isto tornou o código bastante extenso e um pouco repetitivo de escrever.

Avaliado o código desenvolvido, sentimo-nos confiantes no nosso trabalho, no entanto temos noção que é possível que haja maneiras melhores de implementar um jogo em PROLOG, maneiras que devido á nossa inexperiência com a linguagem não são visíveis para nos de momento.

No geral, programar em PROLOG mostrou-se um desafio maior do que estávamos á espera inicialmente, mas acreditamos que no geral realizamos um bom trabalho.

Bibliografia

- http://www.iggamecenter.com/info/en/foa.html
- https://boardgamegeek.com/boardgame/18352/fields-action

Anexo-Código

fields_of_action.pl

```
:- include('Interface.pl').
:- include('Logica.pl').
:- include('Menus.pl').
:- include('Utilitarios.pl').
:- include('Logica_computer.pl').

:- use_module(library(lists)).
:- use_module(library(between)).
:- dynamic is_game_over/1.
:- dynamic catch/1.

:- use_module(library(lists)).
:- use_module(library(between)).
:- dynamic is_game_over/1.
:- dynamic catch/1.

board([ [1-8 ,1-7, 1-6 ,1-5, 0-0, 0-0, 0-0, 0-0],
        [0-0, 0-0, 0-0, 0-0,1-12,1-11,1-10, 1-9],
        [1-4, 1-3, 1-2, 1-1, 0-0, 0-0, 0-0, 0-0],
        [0-0, 0-0, 0-0, 0-0, 0-0, 0-0, 0-0, 0-0],
        [0-0, 0-0, 0-0, 0-0, 0-0, 0-0, 0-0, 0-0],
        [0-0, 0-0, 0-0, 0-0, 2-1, 2-2, 2-3, 2-4],
        [2-9,2-10,2-11,2-12, 0-0, 0-0, 0-0, 0-0],
        [0-0, 0-0, 0-0, 0-0, 2-5, 2-6, 2-7, 2-8]]).

play :-
    start_menu.
```

Interface.pl

```
/* Imprime as jogadas possiveis */
display_plays(_, [], Indice, Indice).
display_plays(QuantLin, [_-_-NewL-NewC-Play|Plays], Indice,
TotalJog) :-
    Lin is QuantLin - NewL,
    Col is NewC + 65,
    space(2),
    write(Indice),
    write(' -> '),
    put_code(Col),
    write(Lin),
    (Play =\= 0 -> write(' *Captura* ') ; true),
    new_line(1),
    NextI is Indice + 1,
    display_plays(QuantLin, Plays, NextI, TotalJog).

/* Imprime um movimento efetuado pelo computador */
print_move(L-C-NewL-NewC-_, Lines) :-
    OldLin is Lines - L,
    OldCol is C + 65,
    NewLin is Lines - NewL,
    NewCol is NewC + 65,
    write('Move : '),
    put_code(OldCol),
    write(OldLin),
    write(' -> '),
    put_code(NewCol),
    write(NewLin),
    new_line(2).

/* Imprme as pecas que foram capturads pelo vencedor deo jogo */
display_catched_pieces(1, [P|Pieces]) :-
    space(3),
    display_player(2-_),
    display_piece(_-P),
    new_line(1),
    display_catched_pieces(1, Pieces).
display_catched_pieces(2, [P|Pieces]) :-
    space(3),
    display_player(1-_),
    display_piece(_-P),
    new_line(1),
    display_catched_pieces(2, Pieces).
display_catched_pieces(_, []).

/* Imprime o tabuleiro de jogo e a numeração de linhas e colunas */
display_game(Board, Player, Lin, Col) :-
    clr,
    display_col_num(Col),
    new_line(1),
```

```
    display_board(Board, Lin, Col),
    display_col_num(Col),
    new_line(2),
    write('--> Jogador '),
    write(Player),
    new_line(2).

/* Imprime a numeração das colunas */
display_col_num(N) :-
    space(6),
    Letter is 65,
    put_code(Letter),
    space(2),
    N1 is N - 1,
    Next is Letter + 1,
    display_col_num(Next, N1).

display_col_num(_,0).
display_col_num(Letter, N) :-
    space(2),
    put_code(Letter),
    space(2),
    N1 is N - 1,
    Next is Letter + 1,
    display_col_num(Next, N1).

/* Imprime a numeração das linhas */
display_lin_num(N) :-
    write(N).

/* Imprime o separador de linhas */
display_lin_separ(Col) :-
    space(3),
    write('------'),
    N1 is Col - 1,
    display_lin_separ(N1, Col).
display_lin_separ(N, Col) :-
    N > 0, !,
    write('-----'),
    N1 is N - 1,
    display_lin_separ(N1, Col).
display_lin_separ(0, _) :- nl.

/* Imprime o separador de colunas */
display_col_separ :-
    write(' | ').

/* Imprime o separador de colunas e o numero da linha */
display_lin_num_and_col_separ(NumL) :-
    NumL > 0,
    space(1),
    display_lin_num(NumL),
```

```
    display_col_separ.

display_lin_num_and_col_separ(0) :-
    space(2),
    display_col_separ.

/* Imprime o tabuleiro */
display_board([L|T], NumL, Col) :-
    display_lin_separ(Col),
    display_lin_num_and_col_separ(NumL),

    display_player_line(L),

    display_lin_num(NumL), nl,
    display_lin_num_and_col_separ(0),

    display_piece_line(L),

    new_line(1),
    NextL is NumL - 1,
    display_board(T, NextL, Col).

display_board([], _, Col) :- display_lin_separ(Col).

/* Imprime a Linha com os jogadores a que pertence cada peca */
display_player_line([]).
display_player_line([C|L]) :-
    display_player(C),
    display_col_separ,
    display_player_line(L).

/* Imprime a linha com o numero da pecas */
display_piece_line([]).
display_piece_line([C|L]) :-
    display_piece(C),
    display_col_separ,
    display_piece_line(L).

/* Imprime um jogador */
display_player(0-0) :- space(2).
display_player(1-_) :- write('B'), space(1).
display_player(2-_) :- write('W'), space(1).

/* Imprime uma peca */
display_piece(0-0) :- space(2).
display_piece(_-X) :- X < 10, space(1), write(X).
display_piece(_-X) :- X >= 10, write(X).
```

Menus.pl

```
display_start_menu :-

write('*********************************************************\n'),
    write('****
****\n'),
    write('****                FIELDS OF ACTION
****\n'),
    write('****
****\n'),

write('*********************************************************\n'),
    write('****
****\n'),
    write('****
****\n'),
    write('****                    1 - Jogar
****\n'),
    write('****                    2 - About
****\n'),
    write('****                    3 - Sair
****\n'),
    write('****
****\n'),
    write('****
****\n'),

write('*********************************************************\n').

start_menu :-
    clr,
    display_start_menu,
    get_number(Choice),
    (
        Choice =:= 1 -> choose_game_menu;
        Choice =:= 2 -> about_menu;
        Choice =:= 3;

        start_menu
    ).

display_about_menu :-

write('*********************************************************\n'),
    write('****
****\n'),
    write('****                FIELDS OF ACTION
****\n'),
    write('****
****\n'),
```

```
write('**************************************************\n'),
    write('****
****\n'),
    write('****                    Autores:
****\n'),
    write('****                - Angelo Moura
****\n'),
    write('****                - Cesar Pinho
****\n'),
    write('****
****\n'),
    write('****                   3 - Voltar
****\n'),
    write('****
****\n'),

write('**************************************************\n').

about_menu :-
    clr,
    display_about_menu,
    get_code(Code),
    skip_line,
    Choice is Code - 48,
    (
        Choice =:= 3 -> start_menu;

        about_menu
    ).

display_choose_game_menu :-

write('**************************************************\n'),
    write('****
****\n'),
    write('****                FIELDS OF ACTION
****\n'),
    write('****
****\n'),

write('**************************************************\n'),
    write('****
****\n'),
    write('****            1 - Jogador vs Jogador
****\n'),
    write('****            2 - Jogador vs Computador
****\n'),
    write('****            3 - Computador vs Computador
****\n'),
    write('****
****\n'),
```

```
    write('****                    4 - Voltar
****\n'),
    write('****
****\n'),

write('*****************************************************\n').

choose_game_menu :-
    clr,
    display_choose_game_menu,
    get_number(Choice),
    (
        Choice =:= 1 -> start_game;
        Choice =:= 2 -> choose_level(human);
        Choice =:= 3 -> choose_level(computer);
        Choice =:= 4 -> start_menu;

        choose_game_menu
    ).

display_choose_level_menu :-

write('*****************************************************\n'),
    write('****
****\n'),
    write('****                FIELDS OF ACTION
****\n'),
    write('****
****\n'),

write('*****************************************************\n'),
    write('****
****\n'),
    write('****                    1 - Facil
****\n'),
    write('****                    2 - Dificil
****\n'),
    write('****
****\n'),
    write('****                    3 - Voltar
****\n'),
    write('****
****\n'),

write('*****************************************************\n').

choose_level(human) :-
    clr,
    display_choose_level_menu,
    get_number(Choice),
    (
        Choice =:= 1 -> start_game(human1);
```

```
        Choice =:= 2 -> start_game(human2);
        Choice =:= 3 -> choose_game_menu;

        choose_level(human)
    ).

choose_level(computer) :-
    clr,
    display_choose_level_menu,
    get_number(Choice),
    (
        Choice =:= 1 -> start_game(computer1);
        Choice =:= 2 -> start_game(computer2);
        Choice =:= 3 -> choose_game_menu;

        choose_level(computer)
    ).

start_game(human1) :-
    asserta(is_game_over(false)),
    board(Board),
    choose_player(J),
    (
        J =:= 1 -> play(Board,1, human, computer1);
        J =:= 2 -> play(Board,1, computer1, human);
        J =:= 3 -> choose_game_menu
    ).

start_game(human2) :-
    asserta(is_game_over(false)),
    board(Board),
    choose_player(J),
    (
        J =:= 1 -> play(Board,1, human, computer2);
        J =:= 2 -> play(Board,1, computer2, human);
        J =:= 3 -> choose_game_menu
    ).

start_game(computer1) :-
    asserta(is_game_over(false)),
    board(Board),
    play(Board,1, computer1, computer1).

start_game(computer2) :-
    asserta(is_game_over(false)),
    board(Board),
    play(Board,1, computer2, computer2).

start_game :-
    asserta(is_game_over(false)),
    board(Board),
    choose_player(J),
```

```
    (
        J =:= 1 -> play(Board,1, human, human);
        J =:= 2 -> play(Board,1, human, human);
        J =:= 3 -> choose_game_menu
    ).

choose_player(Choice) :-
    write('\n O jogador com as pecas pretas e o primeiro a
jogar.\n'),
    write(' Qual as pecas que prefere?\n'),
    write('   1 - Pretas  (Jogador 1)\n'),
    write('   2 - Brancas (Jogador 2)\n'),
    write('   3 - Voltar\n Escolha: '),
    get_number(Choice),
    check_choice_player(Choice).

check_choice_player(1).
check_choice_player(2).
check_choice_player(3).
check_choice_player(_) :-
    write('Erro: Escolha invalida.\n'),
    start_game.

game_over(Player, Pieces) :-
    abolish(catch/1),

    new_line(1),
    write('*********************************************\n'),
    write('**            VENCEDOR :: JOGADOR '), write(Player),
write('            **\n'),
    write('*********************************************\n'),
    write('  PECAS CAPTURADAS :\n'),
    display_catched_pieces(Player,Pieces),
    asserta(is_game_over(true)).

game_over(1) :-
    write('\nJOGADOR '),
    write(1),
    write(' SEM MOVIMENTOS POSSIVEIS\n'),
    pecasCapturadas([], Pieces, 2),
    sort(Pieces, OrderedPieces),
    game_over(2, OrderedPieces).

game_over(2) :-
    write('\nJOGADOR '),
    write(2),
    write(' SEM MOVIMENTOS POSSIVEIS\n'),
    pecasCapturadas([], Pieces, 2),
    sort(Pieces, OrderedPieces),
    game_over(1, OrderedPieces).
```

Logica.pl

```
/* Predicado principal de Jogo */
play(_, _, _, _) :-
    is_game_over(GameOver), GameOver == true,
    abolish(is_game_over/1).

play(Board, Player, human, Type) :-
    is_game_over(GameOver), GameOver == false,
    board_size(Board, Lines, Columns),
    display_game(Board, Player, Lines, Columns), repeat,
    choose_piece(Player, Board, Piece),
    possible_plays(Player, Board, Piece, Lines, Columns, Plays),
    check_quant_plays(Player, Board, Piece, Plays),
    choose_move(Player, Board, human, Type, Lines, Plays, Move),
    make_move(Player, Board, Piece, Columns, Move, NewBoard),
    change_player(Player, NewPlayer),
    play(NewBoard, NewPlayer, Type, human).

play(Board, Player, computer1, Type) :-
    is_game_over(GameOver), GameOver == false,
    board_size(Board, Lines, Columns),
    display_game(Board, Player, Lines, Columns),
    valid_moves(Board, Player, Lines, Columns, ListOfMoves),
    choose_move(Board, Player, computer1, ListOfMoves, Move),
    move(Player, Board, Lines, Columns, Move, NewBoard),
    wait_enter,
    change_player(Player, NewJ),
    play(NewBoard, NewJ, Type, computer1).

play(Board, Player, computer2, Type) :-
    is_game_over(GameOver), GameOver == false,
    board_size(Board, Lines, Columns),
    display_game(Board, Player, Lines, Columns),
    valid_moves(Board, Player, Lines, Columns, ListOfMoves),
    choose_move(Board, Player, computer2, ListOfMoves, Move),
    move(Player, Board, Lines, Columns, Move, NewBoard),
    wait_enter,
    change_player(Player, NewJ),
    play(NewBoard, NewJ, Type, computer2).

/* Pede uma peca ao jogador, verifica se existe no tabuleiro e
retorna-a*/
choose_piece(Player, Board, Piece) :-
    write('Escolha o numero da peca que pretende mover : '),
    get_number(Choice),
    new_line(1),
    exist_piece(Player,Board, Choice),
    Piece = Choice.
choose_piece(_, _, _) :-
    write('Erro: Peca nao existente.'),
    new_line(2),
```

```
    !,fail.

/* Verifica se a Piece existe no tabuleiro */
exist_piece(Player, Board, Piece) :-
    append(Board, List),
    member(Player-Piece, List).

/* Calcula as jogadas possiveis para a Piece */
possible_plays(Player, Board, Piece, Lines, Columns, Plays) :-
    piece_position(Board, Player-Piece, Lin, Col, Columns),
    adjacent_pieces(Board, Lin, Col, Lines, Columns, NumAdj),
    NumAdj =\= 0, !,
    move_horizontal(Player, Board, Lin, Col, Columns, NumAdj,
MovH),
    move_vertical(Player, Board, Lin, Col, Lines, NumAdj, MovV),
    move_diagonal(Player, Board, Lin, Col, Lines, Columns, NumAdj,
MovD),
    append(MovH, MovV, MovHV),
    append(MovHV, MovD, Plays).

possible_plays(Player, Board, Piece, Lines, Columns, ValidPlays) :-
    piece_position(Board, Player-Piece, Lin, Col, Columns),
    possible_plays_without_adjacents(Player, Board, Lin, Col,
Lines, Columns, 1, Plays),
    replace(Board, 0-0, Lin, Col, Board1),
    filter_plays(Board1, Lines, Columns, Plays, ValidPlays).

/* Calcula as jogadas possiveis para uma peca da posicao (Lin,Col)
quando esta nao tem pecas adjacentes */
possible_plays_without_adjacents(Player, Board, Lin, Col, Lines,
Columns, NumAdj, Plays) :-
    NumAdj < Lines,
    NumAdj < Columns,
    move_horizontal(Player, Board, Lin, Col, Columns, NumAdj,
MovH),
    move_vertical(Player, Board, Lin, Col, Lines, NumAdj, MovV),
    move_diagonal(Player, Board, Lin, Col, Lines, Columns, NumAdj,
MovD),
    NewNumAdj is NumAdj + 1,
    possible_plays_without_adjacents(Player, Board, Lin, Col,
Lines, Columns, NewNumAdj, Plays2),
    append(MovH, MovV, MovHV),
    append(MovHV, MovD, Plays1),
    append(Plays1, Plays2, Plays).

possible_plays_without_adjacents(_, _, _, _, _, _, _, []).

/* Retorna apenas as jogadas que resultam com pelo menos 2 pecas
adjacentes.
Isto é aplicado quando uma peca nao tem pecas adjacentes */
filter_plays(Board, Lines, Columns, [L-C-NewL-NewC-Play|Plays],
[X|List]) :-
```

```
    Play =:= 0,
    adjacent_pieces(Board, NewL, NewC, Lines, Columns, NumAdj),
    NumAdj >= 2,
    X = L-C-NewL-NewC-Play,
    filter_plays(Board, Lines, Columns, Plays, List).

filter_plays(Board, Lines, Columns, [_|Plays], List) :-
    filter_plays(Board, Lines, Columns, Plays, List).

filter_plays(_, _, _, [], []).

/* Verifica se a lista tem jogadas, caso contrario
verifica se outras pecas ainda têm movimentos possiveis */
check_quant_plays(Player, Board, []) :-
    Piece = 1,
    check_other_pieces(Player, Board, Piece).
check_quant_plays(_, _, _).

check_quant_plays(Player, Board, Piece, []) :-
    P is Piece + 1,
    check_other_pieces(Player, Board, P),
    write('A peca escolhida nao tem movimentos possiveis. \n'),
    !, fail.
check_quant_plays(_, _, _, L) :- tail(L, _).

/* Se nao exitir jogadas possiveis para nenhuma peca, é game over
*/
check_other_pieces(Player, _, 13) :- !, game_over(Player).
check_other_pieces(Player, Board, Piece):-
    exist_piece(Player, Board, Piece),
    board_size(Board, Lines, Columns),
    possible_plays(Player, Board, Piece, Lines, Columns, Plays),
    check_quant_plays(Player, Board, Piece, Plays).

/* Apresenta os movimentos possiveis e pede ao jogador para
escolher um */
choose_move(_, _, _, _, Lines, Plays, Move) :-
    write(' Jogadas possiveis :\n'),
    display_plays(Lines, Plays, 1, TotalJog),
    new_line(1),
    space(2),
    write('0 -> Voltar\n'),
    write('Escolha : '),
    peek_code(Code),
    Code =\= 48,
    get_number(Choice),
    Choice > 0,
    Choice < TotalJog,
    nth1(Choice, Plays, Move), !.

choose_move(Player, Board, Type1, Type2, _, _, _) :-
    get_number(Code),
```

```
    Code =:= 48, !,
    play(Board, Player, Type1, Type2).

choose_move(Player, Board, Type1, Type2, Lines, Plays, Move) :-
    write('Erro: Escolha invalida.\n\n'),
    choose_move(Player, Board, Type1, Type2, Lines, Plays, Move).

/* Move a peca no tabuleiro e verifica se foi capturada alguma peca
*/
make_move(Player, Board, Piece, Columns, L-C-NewL-NewC-Play,
NewBoard) :-
    replace(Board, Player-Piece, NewL, NewC, Board1),
    empty_cel(V),
    replace(Board1, V, L, C, NewBoard),
    is_catch(Player, Board, Columns, L-C-NewL-NewC-Play).

/* Verifica se foi capturada alguma peca, adiciona-a á base de
dados e verifica se foi vitoria */
is_catch(_, _, _, _-_-_-_-0).
is_catch(Player, Board, Columns, _-_-NewL-NewC-Play) :-
    Play =\= 0, Play =\= Player,
    piece_position(Board, Play-Piece, NewL, NewC, Columns),
    write(' * Peca '),
    write(Piece),
    write(' capturada *\n'),
    asserta(catch(Player-Piece)),
    check_vitory(Player).

/* Verifica se é vitoria do jogador */
check_vitory(Player) :-
    catched_pieces(Player, Pieces),
    sort(Pieces, OrderedPieces),
    check_sequence(OrderedPieces, _, Value),
    Value >= 5,
    game_over(Player, OrderedPieces).

check_vitory(_).

/* Verifica se existe alguma sequencia de numeros, e retorna a
maior sequencia existente */
check_sequence([X|Pieces], FirstGreatSeqPiece-LastGreatSeqPiece,
GreaterSeq) :-
    check_sequence([X|Pieces], 1, X-X, FirstGreatSeqPiece-
LastGreatSeqPiece, GreaterSeq).

check_sequence([], _, _, 0-0, 0).
check_sequence([X|Pieces], Quant, FirstP-LastP, FirstGreatSeqPiece-
LastGreatSeqPiece, GreaterSeq) :-
    NextPiece is LastP + 1,
    NextPiece =:= X,
    NextQ is Quant + 1,
```

```
    check_sequence(Pieces, NextQ, FirstP-X, FirstSeqPiece-
LastSeqPiece, Seq),
    save_sequence(NextQ, FirstP-X, FirstSeqPiece-LastSeqPiece, Seq,
FirstGreatSeqPiece-LastGreatSeqPiece, GreaterSeq).

check_sequence([X|Pieces], _, _-LastP, FirstGreatSeqPiece-
LastGreatSeqPiece, GreaterSeq) :-
    NextPiece is LastP + 1,
    NextPiece =\= X,
    NextQ is 1,
    check_sequence(Pieces, NextQ, X-X, FirstSeqPiece-LastSeqPiece,
Seq),
    save_sequence(NextQ, X-X, FirstSeqPiece-LastSeqPiece, Seq,
FirstGreatSeqPiece-LastGreatSeqPiece, GreaterSeq).

save_sequence(Quant, FirstP-LastP, _, SeqOrder, FirstGreatSeqPiece-
LastGreatSeqPiece, GreaterSeq) :-
    Quant > SeqOrder,
    GreaterSeq = Quant,
    FirstGreatSeqPiece = FirstP,
    LastGreatSeqPiece = LastP.
save_sequence(_, _, OldFirstPiece-OldLastPiece, SeqOrder,
OldFirstPiece-OldLastPiece, SeqOrder).

/* Retorna todas as pecas capturadas pelo Player */
catched_pieces(Player, CatchedPieces) :-
    catched_pieces([], CatchedPieces, Player).

catched_pieces(L1,L,Player) :-
    catch(Player-X),
    not(member(X, L1)),
    append(L1,[X],List),
    catched_pieces(List,L,Player).

catched_pieces(L,L,_).

%%%-------------------------------------------------------%%%
%%%   Calcula o numero de pecas adjacentes a uma peca   %%%
%%%-------------------------------------------------------%%%
/* Retorna a posicao de uma peca.
Pode obter-se a peca numa posicao (L,C) ou a posicao (L,C) de uma
peca */
piece_position(Board, Play-Piece, L, C, Col) :-
    append(Board, BoardList),
    nth0(Num, BoardList, Play-Piece),
    L is div(Num, Col),
    C is mod(Num, Col),!.

/* Calcula o numero de pecas adjacentes a uma peca na posicao (L,C)
Nos comentarios de cada predicado, é mostrada a situação da peca qe
é analisada.
```

```
P representa a posicao da peca, e os numero em redor sao as casas
do tabuleiro em redor da peca */
%        C1  C2  C3  %
%  L1    1   2   3   %
%  L2    4   P   5   %
%  L3    6   7   8   %
adjacent_pieces(Board, L, C, Lines, Columns, NumAdj) :-
    L > 0, C > 0, L < Lines-1, C < Columns-1, !,
    L1 is L - 1,
    L2 is L,
    L3 is L + 1,
    C1 is C - 1,
    C2 is C,
    C3 is C + 1,
    %%  L1  %%
    is_ocuppied_position(Board,L1,C1,Columns,N1),
    is_ocuppied_position(Board,L1,C2,Columns,N2),
    is_ocuppied_position(Board,L1,C3,Columns,N3),
    %%  L2  %%
    is_ocuppied_position(Board,L2,C1,Columns,N4),
    is_ocuppied_position(Board,L2,C3,Columns,N5),
    %%  L3  %%
    is_ocuppied_position(Board,L3,C1,Columns,N6),
    is_ocuppied_position(Board,L3,C2,Columns,N7),
    is_ocuppied_position(Board,L3,C3,Columns,N8),
    NumAdj is N1 + N2 + N3 + N4 + N5 + N6 + N7 + N8.

%        C1  C2 %
%  L1    P   1  %
%  L2    2   3  %
adjacent_pieces(Board, L, C, _, Columns, NumAdj) :-
    L =:= 0, C =:= 0, !,
    L1 is L,
    L2 is L + 1,
    C1 is C,
    C2 is C + 1,
    %%  L1  %%
    is_ocuppied_position(Board,L1,C2,Columns,N1),
    %%  L2  %%
    is_ocuppied_position(Board,L2,C1,Columns,N2),
    is_ocuppied_position(Board,L2,C2,Columns,N3),
    NumAdj is N1 + N2 + N3.

%        C1  C2  %
%  L1    1   P   %
%  L2    2   3   %
adjacent_pieces(Board, L, C, _, Columns, NumAdj) :-
    L =:= 0 , C =:= Columns-1, !,
    L1 is L,
    L2 is L + 1,
    C1 is C - 1,
    C2 is C,
```

```
    %%  L1  %%
    is_ocuppied_position(Board,L1,C1,Columns,N1),
    %%  L2  %%
    is_ocuppied_position(Board,L2,C1,Columns,N2),
    is_ocuppied_position(Board,L2,C2,Columns,N3),
    NumAdj is N1 + N2 + N3.

%        C1  C2  %
%  L1    1   2   %
%  L2    P   3   %
adjacent_pieces(Board, L, C, Lines, Columns, NumAdj) :-
    L =:= Lines-1 , C =:= 0, !,
    L1 is L - 1,
    L2 is L,
    C1 is C,
    C2 is C + 1,
    %%  L1  %%
    is_ocuppied_position(Board,L1,C1,Columns,N1),
    is_ocuppied_position(Board,L1,C2,Columns,N2),
    %%  L2  %%
    is_ocuppied_position(Board,L2,C2,Columns,N3),
    NumAdj is N1 + N2 + N3.

%        C1  C2  %
%  L1    1   2   %
%  L2    3   P   %
adjacent_pieces(Board, L, C, Lines, Columns, NumAdj) :-
    L =:= Lines-1 , C =:= Columns-1, !,
    L1 is L - 1,
    L2 is L,
    C1 is C - 1,
    C2 is C,
    %%  L1  %%
    is_ocuppied_position(Board,L1,C1,Columns,N1),
    is_ocuppied_position(Board,L1,C2,Columns,N2),
    %%  L2  %%
    is_ocuppied_position(Board,L2,C1,Columns,N3),
    NumAdj is N1 + N2 + N3.

%        C1  C2  C3  %
%  L1    1   P   2   %
%  L2    3   4   5   %
adjacent_pieces(Board, L, C, _, Columns, NumAdj) :-
    L =:= 0, C > 0, C < Columns-1, !,
    L1 is L,
    L2 is L + 1,
    C1 is C - 1,
    C2 is C,
    C3 is C + 1,
    %%  L1  %%
    is_ocuppied_position(Board,L1,C1,Columns,N1),
    is_ocuppied_position(Board,L1,C3,Columns,N2),
```

```
    %%  L2  %%
    is_ocuppied_position(Board,L2,C1,Columns,N3),
    is_ocuppied_position(Board,L2,C2,Columns,N4),
    is_ocuppied_position(Board,L2,C3,Columns,N5),
    NumAdj is N1 + N2 + N3 + N4 + N5.

%        C1  C2  C3  %
%  L1    1   2   3   %
%  L2    4   P   5   %
adjacent_pieces(Board, L, C, Lines, Columns, NumAdj) :-
    L =:= Lines-1, C > 0, C < Columns-1, !,
    L1 is L - 1,
    L2 is L,
    C1 is C - 1,
    C2 is C,
    C3 is C + 1,
    %%  L1  %%
    is_ocuppied_position(Board,L1,C1,Columns,N1),
    is_ocuppied_position(Board,L1,C2,Columns,N2),
    is_ocuppied_position(Board,L1,C3,Columns,N3),
    %%  L2  %%
    is_ocuppied_position(Board,L2,C1,Columns,N4),
    is_ocuppied_position(Board,L2,C3,Columns,N5),
    NumAdj is N1 + N2 + N3 + N4 + N5.

%        C1  C2  %
%  L1    1   2   %
%  L2    P   3   %
%  L3    4   5   %
adjacent_pieces(Board, L, C, Lines, Columns, NumAdj) :-
    L > 0, L < Lines-1, C =:= 0, !,
    L1 is L - 1,
    L2 is L,
    L3 is L + 1,
    C1 is C,
    C2 is C + 1,
    %%  L1  %%
    is_ocuppied_position(Board,L1,C1,Columns,N1),
    is_ocuppied_position(Board,L1,C2,Columns,N2),
    %%  L2  %%
    is_ocuppied_position(Board,L2,C2,Columns,N3),
    %%  L3  %%
    is_ocuppied_position(Board,L3,C1,Columns,N4),
    is_ocuppied_position(Board,L3,C2,Columns,N5),
    NumAdj is N1 + N2 + N3 + N4 + N5.

%        C1  C2  %
%  L1    1   2   %
%  L2    3   P   %
%  L3    4   5   %
adjacent_pieces(Board, L, C, Lines, Columns, NumAdj) :-
    L > 0, L < Lines-1, C =:= Columns-1, !,
```

```
    L1 is L - 1,
    L2 is L,
    L3 is L + 1,
    C1 is C - 1,
    C2 is C,
    %%  L1  %%
    is_ocuppied_position(Board,L1,C1,Columns,N1),
    is_ocuppied_position(Board,L1,C2,Columns,N2),
    %%  L2  %%
    is_ocuppied_position(Board,L2,C1,Columns,N3),
    %%  L3  %%
    is_ocuppied_position(Board,L3,C1,Columns,N4),
    is_ocuppied_position(Board,L3,C2,Columns,N5),
    NumAdj is N1 + N2 + N3 + N4 + N5.

/* Verifica se uma posicao (L,C) do tabuleiro está ocupada ou nao.
Em caso afirmativo , Occupied = 1, senao, Occupied = 0 */
is_ocuppied_position(Board, L, C, Columns, Occupied) :-
    append(Board, BoardList),
    Num is (L * Columns) + C,
    nth0(Num, BoardList, Piece),
    empty_cel(Piece), !,
    Occupied = 0.
is_ocuppied_position(_, _, _, _, 1).

%%%-----------------------------------------------------%%%
%%%     Calcula os movimentos possiveis de uma peca   %%%
%%%-----------------------------------------------------%%%
/* Horizontal Movement */
move_horizontal(Player, Board, LinP, ColP, Columns, NumAdj, MovH)
:-
    nth0(LinP, Board, Line),
    move_right(Player, LinP, ColP, Line, Columns, NumAdj, MovR),
    move_left(Player, LinP, ColP, Line, NumAdj, MovL),
    append(MovL,MovR, MovH).

move_right(Player, LinP, ColP, Line, Columns, NumAdj, MovR) :-
    NewC is ColP + NumAdj,
    NewC < Columns,
    nth0(NewC, Line, NewPlayer-NewPiece),
    (empty_cel(NewPlayer-NewPiece) ; NewPlayer =\= Player),
    MovR = [LinP-ColP-LinP-NewC-NewPlayer].
move_right(_, _, _, _, _, _, []).

move_left(Player, LinP, ColP, Line, NumAdj, MovL) :-
    NewC is ColP - NumAdj,
    NewC >= 0,
    nth0(NewC, Line, NewPlayer-NewPiece),
    (empty_cel(NewPlayer-NewPiece) ; NewPlayer =\= Player),
    MovL = [LinP-ColP-LinP-NewC-NewPlayer].
move_left(_, _, _, _, _, []).
```

```
/* Vertical Movement */
move_vertical(Player, Board, LinP, ColP, Lines, NumAdj, MovV) :-
    move_up(Player, Board, LinP, ColP, NumAdj, MovU),
    move_down(Player, Board, LinP, ColP, Lines, NumAdj, MovD),
    append(MovU, MovD, MovV).

move_up(Player, Board, LinP, ColP, NumAdj, MovU) :-
    NewL is LinP - NumAdj,
    NewL >= 0,
    nth0(NewL, Board, NewLine),
    nth0(ColP, NewLine, NewPlayer-NewPiece),
    (empty_cel(NewPlayer-NewPiece) ; NewPlayer =\= Player),
    MovU = [LinP-ColP-NewL-ColP-NewPlayer].
move_up(_, _, _, _, _, []).

move_down(Player, Board, LinP, ColP, Lines, NumAdj, MovD) :-
    NewL is LinP + NumAdj,
    NewL < Lines,
    nth0(NewL, Board, NewLine),
    nth0(ColP, NewLine, NewPlayer-NewPiece),
    (empty_cel(NewPlayer-NewPiece) ; NewPlayer =\= Player),
    MovD = [LinP-ColP-NewL-ColP-NewPlayer].
move_down(_, _, _, _, _, _, []).

/* Diagonal Movement */
move_diagonal(Player, Board, LinP, ColP, Lines, Columns, NumAdj,
MovD) :-
    move_up_left(Player, Board, LinP, ColP, NumAdj, MovUL),
    move_up_right(Player, Board, LinP, ColP, Columns, NumAdj,
MovUR),
    move_down_left(Player, Board, LinP, ColP, Lines, NumAdj,
MovDL),
    move_down_right(Player, Board, LinP, ColP, Lines, Columns,
NumAdj, MovDR),
    append(MovUL, MovUR, Mov1),
    append(Mov1, MovDL, Mov2),
    append(Mov2, MovDR, MovD).

move_up_left(Player, Board, LinP, ColP, NumAdj, MovUL) :-
    NewL is LinP - NumAdj,
    NewL >= 0,
    NewC is ColP - NumAdj,
    NewC >= 0,
    nth0(NewL, Board, NewLine),
    nth0(NewC, NewLine, NewPlayer-NewPiece),
    (empty_cel(NewPlayer-NewPiece) ; NewPlayer =\= Player),
    MovUL = [LinP-ColP-NewL-NewC-NewPlayer].
move_up_left(_, _, _, _, _, []).

move_up_right(Player, Board, LinP, ColP, Columns, NumAdj, MovUR) :-
    NewL is LinP - NumAdj,
    NewL >= 0,
```

```
    NewC is ColP + NumAdj,
    NewC < Columns,
    nth0(NewL, Board, NewLine),
    nth0(NewC, NewLine, NewPlayer-NewPiece),
    (empty_cel(NewPlayer-NewPiece) ; NewPlayer =\= Player),
    MovUR = [LinP-ColP-NewL-NewC-NewPlayer].
move_up_right(_, _, _, _, _, _, []).

move_down_left(Player, Board, LinP, ColP, Lines, NumAdj, MovDL) :-
    NewL is LinP + NumAdj,
    NewL < Lines,
    NewC is ColP - NumAdj,
    NewC >= 0,
    nth0(NewL, Board, NewLine),
    nth0(NewC, NewLine, NewPlayer-NewPiece),
    (empty_cel(NewPlayer-NewPiece) ; NewPlayer =\= Player),
    MovDL = [LinP-ColP-NewL-NewC-NewPlayer].
move_down_left(_, _, _, _, _, _, []).

move_down_right(Player, Board, LinP, ColP, Lines, Columns, NumAdj,
MovDR) :-
    NewL is LinP + NumAdj,
    NewL < Lines,
    NewC is ColP + NumAdj,
    NewC < Columns,
    nth0(NewL, Board, NewLine),
    nth0(NewC, NewLine, NewPlayer-NewPiece),
    (empty_cel(NewPlayer-NewPiece) ; NewPlayer =\= Player),
    MovDR = [LinP-ColP-NewL-NewC-NewPlayer].
move_down_right(_, _, _, _, _, _, _, []).
```

Logica_computer.pl

```
:- use_module(library(random)).

pieces([1,2,3,4,5,6,7,8,9,10,11,12]).

/* Obtem uma lista de todas as jogadas possiveis */
valid_moves(Board, Player, Lines, Columns, ListOfMoves) :-
    valid_moves(Board, Player, 1, Lines, Columns, ListOfLists),
    append(ListOfLists, ListOfMoves).

valid_moves(_, _, 13, _, _, []).

valid_moves(Board, Player, Piece, Lines, Columns, [X|ListOfMoves])
:-
    exist_piece(Player, Board, Piece),
    possible_plays(Player, Board, Piece, Lines, Columns, X),
    NewPiece is Piece + 1,
    valid_moves(Board, Player, NewPiece, Lines, Columns,
ListOfMoves).

valid_moves(Board, Player, Piece, Lines, Columns, ListOfMoves) :-
    Piece =< 12,
    NewPiece is Piece + 1,
    valid_moves(Board, Player, NewPiece, Lines, Columns,
ListOfMoves).

/* Escolhe a jogada a efetuar pelo computador */
choose_move(_, _, computer1, ListOfMoves, Move) :-
    random_member(Move, ListOfMoves).

choose_move(Board, Player, computer2, ListOfMoves, Move) :-
    board_size(Board, Lines, Columns),
    choose_best_move(Board, Player, Lines, Columns, ListOfMoves, _,
L-C-NewL-NewC-Play),
    Play =:= 0,
    random_member(Move, ListOfMoves).

choose_move(Board, Player, computer2, ListOfMoves, Move) :-
    board_size(Board, Lines, Columns),
    choose_best_move(Board, Player, Lines, Columns, ListOfMoves, _,
Move).

/* Escolhe a melhor jogada tenod em conta apenas as possiveis */
choose_best_move(_, _, _, _, [], -1, _).
choose_best_move(Board, Player, Lines, Columns, [L-C-NewL-NewC-
Play|ListOfMoves], Value, Move) :-
    Play =\= 0,
    piece_position(Board, _-Peca, NewL, NewC, Columns),
    value(Board, Player, FirstSeqPiece-LastSeqPiece, Value1),
    Value1 > 0,
    (
```

```
        Peca =:= FirstSeqPiece - 1;
        Peca =:= LastSeqPiece + 1
    ),
    Value2 is Value1 + 5,
    Move1 = L-C-NewL-NewC-Play,
    choose_best_move(Board, Player, Lines, Columns, ListOfMoves,
Value3, Move2),
    save_move(Value2, Move1, Value3, Move2, Value, Move).

choose_best_move(Board, Player, Lines, Columns, [L-C-NewL-NewC-
Play|ListOfMoves], Value, Move) :-
    Play =\= 0,
    piece_position(Board, _-Peca, NewL, NewC, Columns),
    value(Board, Player, FirstSeqPiece-LastSeqPiece, Value1),
    Value1 > 0,
    (
        Peca =:= FirstSeqPiece - 2;
        Peca =:= LastSeqPiece + 2
    ),
    Value2 is Value1 + 4,
    Move1 = L-C-NewL-NewC-Play,
    choose_best_move(Board, Player, Lines, Columns, ListOfMoves,
Value3, Move2),
    save_move(Value2, Move1, Value3, Move2, Value, Move).

choose_best_move(Board, Player, Lines, Columns, [L-C-NewL-NewC-
Play|ListOfMoves], Value, Move) :-
    Play =\= 0,
    piece_position(Board, _-PecaToMove, L, C, Columns),
    replace(Board, Player-PecaToMove, NewL, NewC, Board1),
    empty_cel(V),
    replace(Board1, V, L, C, NewBoard),
    value(NewBoard, Player, _, Value1),
    Move1 = L-C-NewL-NewC-Play,
    choose_best_move(Board, Player, Lines, Columns, ListOfMoves,
Value2, Move2),
    save_move(Value1, Move1, Value2, Move2, Value, Move).

choose_best_move(Board, Player, Lines, Columns, [L-C-NewL-NewC-
Play|ListOfMoves], Value, Move) :-
    Play == 0,
    Value1 = 0,
    Move1 = L-C-NewL-NewC-Play,
    choose_best_move(Board, Player, Lines, Columns, ListOfMoves,
Value2, Move2),
    save_move(Value1, Move1, Value2, Move2, Value, Move).

save_move(Value1, Move1, Value2, _, Value, Move) :-
    Value2 < Value1,
    Value = Value1,
    Move = Move1.
```

```
save_move(Value1, _, Value2, Move2, Value, Move) :-
    Value2 >= Value1,
    Value = Value2,
    Move = Move2.

/* Executa a jogada no tabuleiro e retorna o novo tabuleiro */
move(Player, Board, Lines, Columns, L-C-NewL-NewC-Play, NewBoard)
:-
    piece_position(Board, _-Peca, L, C, Columns),
    replace(Board, Player-Peca, NewL, NewC, Board1),
    empty_cel(V),
    replace(Board1, V, L, C, NewBoard),
    print_move(L-C-NewL-NewC-Play, Lines),
    is_catch(Player, Board, Columns, L-C-NewL-NewC-Play).

/* Avalia o estado do jogo. O Value é o maoir numero de pecas em
sequencia que já foram capturadas */
value(Board, Player, FirstSeqPiece-LastSeqPiece, Value) :-
    pieces_out_of_game(Board, Player, Pieces),
    sort(Pieces, OrderedPieces),
    check_sequence(OrderedPieces, FirstSeqPiece-LastSeqPiece,
Value).

/* Retorna as pecas do adversario que já nao estao no tabuleiro */
pieces_out_of_game(Board, Player, CatchedPieces) :-
    append(Board, BoardList),
    pieces_playing(BoardList, Player, PiecesInGame),
    pieces(TotalPieces),
    subseq(TotalPieces, PiecesInGame, CatchedPieces).

/* Retorna as pecas do adversario que ainda estao no tabuleiro */
pieces_playing([], _, []).
pieces_playing([Play-Piece|BoardList], 1, [P|PiecesInGame]) :-
    Play == 2,
    P = Piece,
    pieces_playing(BoardList, 1, PiecesInGame).

pieces_playing([Play-Piece|BoardList], 2, [P|PiecesInGame]) :-
    Play == 1,
    P = Piece,
    pieces_playing(BoardList, 2, PiecesInGame).

pieces_playing([_|BoardList], Player, PiecesInGame) :-
    pieces_playing(BoardList, Player, PiecesInGame).
```

Utilitarios.pl

```
empty_cel(0-0).

clr :- write('\33\[2J').

wait_enter :-
    write('Press any key to continue...'),
    new_line(2),
    get_char(_).

change_player(1, 2).
change_player(2, 1).

isEmpty([]).
isEmpty([_|_]) :- !, fail.

/* Conta os elementos de uma lista */
count_lines([_|L], NumL) :-
    count_lines(L,N),
    NumL is N + 1.
count_lines([],0).

/* Calcula e retorna o tamanho do tabuleiro (Linhas e Colunas) */
board_size([H | T], Lin, Col) :-
    count_lines(H, Col),
    count_lines(T, X),
    Lin is X + 1.

space(1) :- write(' ').
space(N) :-
    N > 1,
    write(' '),
    Next is N - 1,
    space(Next).

new_line(1) :- nl.
new_line(N) :-
    N > 1,
    nl,
    Next is N - 1,
    new_line(Next).

not(X) :- X ,! ,fail.
not(_).

/* Substitui um o elemento da posicao (Lin,Col) por Elem */
replace([X|L], Elem, 0, Col, [Y|L]) :-
    replace(X, Elem, Col, Y).
replace([X|L], Elem, Lin, Col, [X|NewL]) :-
    Lin > 0,
    Lin1 is Lin - 1,
```

```
    replace(L, Elem, Lin1, Col, NewL).

replace([], _, _, _).
replace([_|L], Elem, 0, [Elem|L]).
replace([X|L], Elem, Col, [X|N]) :-
    Col > 0,
    Col1 is Col - 1,
    replace(L, Elem, Col1, N).

/* Le inputs da consola e retorna se for um numero válido */
:- dynamic choice/1.
get_number(Choice) :-
    asserta((choice(10):-!)),
    get_code(Code1),
    between(48, 57, Code1),
    Num1 is Code1 - 48,
    asserta((choice(Num1):-!)),
    peek_code(Code2),
    between(48, 57, Code2),
    Num2 is Code2 - 48,
    Choice is Num1 * 10 + Num2,
    skip_line, !,
    abolish(choice/1).

get_number(Choice) :-
    choice(Choice),
    Choice == 10,
    abolish(choice/1),
    !, fail.

get_number(Choice) :-
    choice(Choice),
    skip_line,
    abolish(choice/1).
```