Shiftago

Relatório Final

Mestrado Integrado em Engenharia Informática e Computação

Programação em Lógica

Grupo Shiftago4:
Daniel Ribeiro de Pinho - 201505302
Francisco Tuna de Andrade - 201503481

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

12 de Novembro de 2017

Resumo

Neste trabalho foi abordada a implementação do jogo de tabuleiro *Shiftago* usando a linguagem de programação Prolog. No programa resultante é possível jogar a variante *Express* do jogo para duas pessoas. Esta implementação permite que pelo menos um dos intervenientes no jogo possa ser o computador, resultando em três situações de jogo: dois jogadores, jogador contra computador, e computador contra computador.

O tabuleiro do jogo é representado internamente através de uma matriz 7×7, análoga ao tabuleiro real, com as jogadas sendo feitas através de operações de manipulação de listas. O utilizador dispõe de mecanismos de interação com o programa, sendo-lhe mostradas opções para navegar menus ou fazer jogadas quando necessário. Quando o computador é um dos jogadores, é usado um de três níveis distintos de dificuldade para fazer a jogada. Este nível é escolhido pelo utilizador ao iniciar o jogo.

Este trabalho permitiu-nos entrar em contacto com um novo paradigma de programação e adaptarmo-nos a maneiras diferentes de resolver problemas. Consideramos assim que o programa resultante deste trabalho foi bem executado.

Conteúdo

1 Introdução

No âmbito da unidade curricular de Programação em Lógica foi-nos proposta a implementação de um jogo de tabuleiro com dois jogadores usando a linguagem de programação Prolog. O jogo que nos foi atribuído foi o jogo Shiftago, criado em 2016.

Este relatório encontra-se dividido nas seguintes partes: *O Jogo Shiftago*, em que falamos da história do jogo de tabuleiro; *Lógica do Jogo*, em que são abordados aspetos mais técnicos, como a representação interna e externa do estado do jogo, execução de jogadas, final do jogo e jogadas do computador; *Interface com o Utilizador*, em que é descrita a maneira como o programa interage com os jogadores; e *Conclusões*, em que falamos das conclusões tiradas e possíveis melhoramentos.

2 O Jogo Shiftago

Criado em 2016 por Frank Warneke e Robert Witter e publicado pela WiWa Spiele [1], o *Shiftago* é um jogo que pode ser jogado por dois a quatro jogadores, com idade recomendada a partir dos oito anos[2]. O jogo pode ser jogado em três modos distintos: *Express*, *Expert* e *Extreme*, sendo que o modo *Express* para duas pessoas será o abordado neste trabalho.

Neste modo existe um tabuleiro 7×7, para além de 22 peças de duas cores diferentes correspondentes a cada jogador. Inicialmente, o tabuleiro está vazio. Os jogadores jogam alternadamente, pondo no seu turno uma peça da sua cor num dos quatro lados do tabuleiro. Se uma peça já estiver na casa que o jogador pretende utilizar ela deve ser simplesmente movida para a frente numa direção perpendicular ao seu lado do tabuleiro. As peças situadas à frente desta devem ser movidas uma casa para frente de modo análogo, tendo no entanto em atenção que não é permitido que alguma peça saia do tabuleiro.

Vence o primeiro jogador que conseguir obter uma linha composta por 5 peças da sua cor[3].

Figura 1: Tabuleiro no seu estado inicial

3 Lógica do Jogo

3.1 Representação do Estado do Jogo

O tabuleiro, sendo uma matriz 7×7, é representado internamente por uma lista de listas. A sua posição inicial será uma estrutura preenchida por apenas zeros, sendo durante o percurso do jogo preenchida com as peças colocadas pelos jogadores.

Durante o jogo, à medida que os jogadores vão fazendo os seus movimentos, os zeros vão sendo substituídos por 1 e 2, significando peças de cada jogador:

- Instância inicial:
  `[[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],`
  `[0,0,0,0,0,0,0]]`.

- Posição intermédia:
  `[[0,0,0,0,1,0,0],[0,0,0,0,1,0,0],[0,0,0,0,2,0,0],`
  `[2,0,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,0,2],`
  `[0,0,0,0,0,0,1]]`.

- Posição final (o jogador 1 venceu porque fez uma linha de 5 na vertical):
  `[[0,0,0,1,1,0,0],[0,0,0,0,1,0,0],[1,0,0,0,1,0,0],`
  `[1,2,0,0,1,0,2],[2,0,0,0,1,0,1],[2,0,0,2,2,2,2],`
  `[0,0,0,2,2,1,1]]`.

Estes exemplos encontram-se representados de seguida:

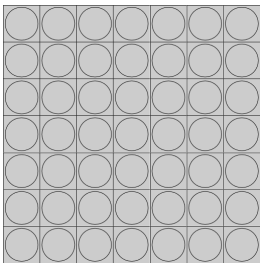

Figura 2: Representação do estado inicial do tabuleiro

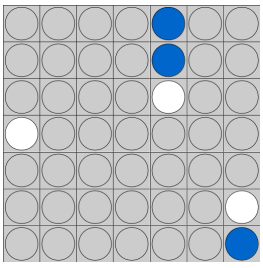

Figura 3: Representação de um estado intermédio do jogo

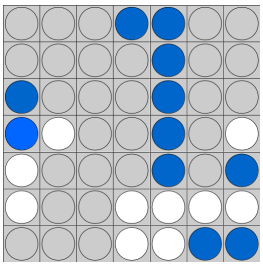

Figura 4: Representação de um estado de fim do jogo

3.2 Visualização do Tabuleiro

Em modo de texto, o tabuleiro pode ser visualizado da seguinte maneira, correspondente o tabuleiro a partir de cima:

```
  1 2 3 4 5 6 7
 +-------------+
1|. . . . . . .|
2|. . . . . . .|
3|. . . . . . .|
4|. . . . . . .|
5|. . . . . . .|
6|. . . . . . .|
7|. . . . . . .|
 +-------------+
```

Para visualizar o tabuleiro desta forma, usamos então os seguintes predicados:

```
display_board(+Matrix).
display_line(+Line, +Row).
display_line(+Line).
display_matrix(+N, +Matrix).
display_top.
display_edge.
```

O predicado `display_board /1` usa `display_top /0` e `display_edge /0` para representar o topo do tabuleiro, seguindo do predicado `display_matrix /2`. Este funciona recursivamente, com a ajuda de um contador (para contar o número certo de linhas e apresentar os respetivos números antes de representar o conteúdo), invocando o predicado `display_line /1 /2` para o efeito. Depois de as sete linhas estarem representadas, o predicado `display_edge /0` é usado novamente para representar o limite inferior do tabuleiro.

Durante o jogo, os caracteres `.` apresentados no ecrã vão sendo substituídos pelos caracteres `X` e `O`, representando as peças inseridas pelo jogador 1 e pelo jogador 2, respetivamente.

É de realçar que não são necessários quaisquer caracteres ASCII específicos para representar o tabuleiro, visto que todos os caracteres podem ser encontrados em qualquer teclado.

3.3 Lista de Jogadas Válidas

Dada a natureza da maneira como as jogadas são feitas (isto é, colocando uma peça numa casa nas arestas do tabuleiro, empurrando quaisquer peças que estejam à sua frente nessa linha ou coluna), um dado tabuleiro tem, no máximo, 28 jogadas possíveis. Este número só varia quando uma dada fila de peças está cheia, já que não é possível empurrar peças para fora do tabuleiro.

Tendo isto em conta, são usados os predicados `valid_moves_make_list /3` e `valid_moves /3` para elaborar uma lista das jogadas possíveis. A diferença principal é que o primeiro cria uma lista com 28 elementos, independentemente da validade das jogadas. Contudo, as jogadas inválidas são substituídas por zeros dentro da lista. O segundo, por outro lado, apenas inclui as jogadas válidas. O predicado `valid_moves /3`, de forma a obter a lista é usado o predicado `valid_moves_make_list /3`. Este, por sua vez, recorre ao predicado auxiliar `valid_moves_make_list_aux /4` para analisar as jogadas possíveis de efetuar em cada aresta, construindo a lista no processo.

O predicado `valid_moves_make_list_aux`, ao percorrer cada aresta do tabuleiro, chama o predicado `valid_move /4` para verificar, com o predicado `possible_move /2`, se é possível colocar uma peça no tabuleiro naquela fila a partir da aresta em questão, calculando também o tabuleiro resultante no processo.

Os argumentos dos predicados enunciados nesta secção encontram-se declarados de seguida.

```
valid_moves(+Board, +Player, -ListOfMoves).
valid_moves_make_list(+Board, +Player, -ListOfMoves_0).
valid_moves_make_list_aux(+Board, +Player, +Move,
↪  -ListOfMoves_0).
valid_move(+Board, +Player, +Move, -NewBoard).
possible_move(+Line).
```

3.4 Execução de Jogadas

A execução de jogadas difere dependendo do tipo de jogador (humano ou computador), mas tem um raciocínio semelhante entre os dois casos. Caso o jogador seja um humano, o jogo usa o predicado `insert_piece /5` para executar a jogada. Este predicado recorre a `get_move /2` para lhe pedir uma aresta do tabuleiro (cima, baixo, esquerda e direita) e uma linha/coluna, contadas a partir do canto superior esquerdo do tabuleiro. É chamado então o predicado `insert_piece /7`, que verifica se a jogada é válida, de acordo com o predicado `possible_move /1`, e então insere uma peça no tabuleiro.

No caso de ser o computador a efetuar a sua jogada, ele usa o predicado `cpu_move /6` para o fazer. Este predicado, por sua vez, recorre ao predicado `valid_moves_make_list /3` e obtêm a lista de jogadas possíveis, de acordo com o descrito na secção anterior. Posteriormente, tenta escolher a melhor jogada existente na lista e executa-a, terminando a sua jogada.

Os argumentos dos predicados enunciados nesta secção encontram-se declarados de seguida.

```
insert_piece(+Board, +CurrentPlayer, -NewBoard, +CurrentPieces,
↪  -NewCurrentPieces).
get_move(-Edge, -Row).
```

```
insert_piece(+Board, +Edge, +Row, +CurrentPlayer, -NewBoard,
↪  +CurrentPieces, -NewCurrentPieces).
possible_move(+Line).
valid_moves_make_list(+Board, +Player, -ListOfMoves_0).
```

3.5 Avaliação do Tabuleiro

A avaliação de um tabuleiro é feita utilizando o predicado `value /3`, que calcula o valor de um determinado tabuleiro para um determinado jogador. Para conseguir chegar a este valor o predicado `value /3` faz uso de um outro predicado, `value_formula /3`, que calcula um valor do tabuleiro intermédio.

Assim, esse valor intermédio utiliza, por sua vez, o valor dado pelo predicado `rate_adj_formula /3` que, para cada linha, coluna e diagonal do tabuleiro, calcula um valor que será necessário para o obtenção de `value_formula`.

`rate_adj_formula` é calculado obtendo, somando para cada linha, diagonal e coluna a potência levada a 5 do número de elementos adjacentes nessa fila. Por exemplo, considerando a linha seguinte:

`X X O O X X X`

O seu `rate_adj_formula` para o jogador 1, cujas peças são representadas por um X, é $2^5 + 3^5 = 275$. Considerando a mesma linha, mas para o jogador 2, cujas peças são representadas por um O, o valor de `rate_adj_formula` é agora $2^5 = 32$.

O valor de `value_formula` é depois calculado como a soma dos vários valores possíveis de `rate_adj_formula` para cada linha, coluna e diagonal do tabuleiro.line

O valor final de `value` é calculado da seguinte forma:

- Se o número de peças máximo de peças adjacentes no tabuleiro fôr maior ou igual a 5, `value` corresponderá a 10^6. O número máximo de peças adjacentes num tabuleiro é calculado utilizando o predicado `max_pieces_adj /4`.
- Caso contrário, se o número de peças adjacentes máximo que o jogador seguinte pode obter na jogada seguinte fôr maior ou igual a 5, `value` corresponderá a -10^6. O número máximo de peças adjacentes que o jogador seguinte pode obter é calculado utilizando os predicados `valid_moves /3` e `max_pieces_adj /4`.
- Caso a condição anterior também não se verifique `value` corresponderá à subtração entre `value_formula` para os tabuleiros e jogadores atuais e o valor máximo de `value_formula` que o jogador seguinte pode obter na jogada seguinte.

Os argumentos dos predicados enunciados nesta secção encontram-se declarados de seguida.

```
value(+Board, +Player, -Value).
value_formula(+Board, +Player,  -Value).
rate_adj_formula(+Elem, +List, -Max).
max_pieces_adj(+Board, +Player, -Position,  -Value)).
```

```
cpu_move(+Board, +Player, +Move, -NewBoard, +CurrentPieces,
↪  -NewCurrentPieces).
valid_moves_make_list(+Board, +Player, -ListOfMoves_0).
```

3.6 Final do Jogo

Após a execução da jogada de um jogador, este (e o jogo, por conseguinte) encontram-se numa de três situações possíveis:

1. O jogador tem no tabuleiro uma fila de cinco peças consecutivas da sua cor, quer na horizontal, vertical ou diagonal;
2. O jogador ficou sem peças e não tem uma fila de cinco peças consecutivas suas;
3. O jogador ainda tem peças para jogar e não tem uma fila de cinco peças consecutivas da sua cor.

O programa verifica, com o predicado `end_move /4`, a ocorrência destes dois primeiros casos, na ordem em que estão aqui mencionados, e depois, caso nenhum deles se verifique, o jogo continua, começando a vez do outro jogador.

O predicado `check_for_win /2 /3` é usado para verificar o caso 1, usando os predicados `check_lines /2`, `check_columns /2` e `check_diagonals /3`, para verificar a existência de filas horizontais, verticais e diagonais, respetivamente.

O predicado `check_lines /2` simplesmente verifica se cada lista que constitui o tabuleiro tem como sublista uma lista com cinco caracteres correspondentes ao caracter do jogador. O predicado `check_columns /2` transpõe a matriz e subsequentemente recorre ao predicado `check_lines /2`, visto que, após a transposição da matriz, os elementos da lista que representa o tabuleiro passaram a ser as colunas. O predicado `check_diagonals /2` percorre a matriz com o predicado `get_diagonal /5`, criando listas que incluem as células da matriz, dispostas diagonalmente, com um comprimento mínimo de cinco caracteres e recorre a `check_lines /2`, verificando as listas criadas.

Para a verificação do caso 2 é usado o predicado `end_move /4` quando o último argumento (o número de peças com que o jogador ficou após a sua jogada) é igual a zero. Esta tentativa de unificação com o predicado só é feita após a unificação com o predicado `check_for_win /3` falhar, o que indica que o jogador ficou sem peças e não tinha uma fila de cinco peças consecutivas. Nesta situação, o oponente ganha imediatamente o jogo.

Caso um destes dois casos se verifique, é chamado o predicado `end_game /1`, que anuncia o vencedor do jogo e pergunta ao utilizador se este quer voltar ao menu principal, com o predicado `return_to_main_menu /1`.

Os argumentos dos predicados enunciados nesta secção encontram-se declarados de seguida.

```
end_move(+Player, +Board, +OpponentPieces, +NewCurrentPieces).
end_game(+Player).
check_for_win(+Pieces, +Player, +Board).
check_for_win(+Player, +Board).
check_lines(+Player, +Board).
check_columns(+Player, +Board).
check_diagonals(+Player, +Board).
```

```
get_diagonal(+Row, +Column, +Board, +Line, -FLine).
return_to_main_menu(+Choice).
```

3.7 Jogada do Computador

O computador tem três níveis de dificuldade implementados: o nível mais fácil, em que é escolhida uma jogada totalmente aleatória; um nível intermédio, em que só se preocupa em maximizar o tamanho da maior fila de peças suas; e um nível mais difícil, em que, para além de maximizar o tamanho das suas filas, tenta reduzir as filas do oponente. Este nível mais difícil é realizado recorrendo à ao predicado `value /3`, descobrindo, entre todas as jogadas possíveis que o cpu pode executar, aquela que lhe deixa com um tabuleiro de *value* maior. A jogada é executada através do predicado `cpu_move /6`, que, de forma análoga ao predicado `insert_piece /5`, descrito na secção 3.4, escolhe uma jogada e aplica-a ao tabuleiro.

O primeiro nível de dificuldade, de forma a escolher uma jogada aleatória, recorre ao predicado `valid_moves_make_list /3` para obter a lista de jogadas válidas. Posteriormente, usa o predicado `cpu_generate_move /3` para escolher uma jogada válida qualquer e aplicá-la ao tabuleiro, usando finalmente `convert_order_Move /2` para ser possível dizer ao utilizador qual foi a jogada feita.

Os outros dois níveis de dificuldade, como tentam escolher a melhor jogada de acordo com os seus critérios, usam o mesmo raciocínio. Tal como o primeiro nível, obtêm a lista de jogadas válidas com `valid_moves_make_list /3`, usando-a depois com o predicado `map_redefined /4 /5`. Este predicado é a única diferença no raciocínio dos dois níveis de dificuldade; o nível intermédio usa este predicado em conjunção com `max_pieces_adj /4`, enquanto que o nível mais avançado usa-o com `value /3`. Cria-se então a lista de valores de jogada para as peças, conforme descrito na secção 3.5. Partindo daqui, o predicado `max_list /3`, com a ajuda de `max_list_aux /6`, é usado para obter a melhor jogada, sendo então aplicada.

Os argumentos dos predicados enunciados nesta secção encontram-se declarados de seguida.

```
cpu_move(+Board, +Player, +Move, -NewBoard, +CurrentPieces,
↪  -NewCurrentPieces).
valid_moves_make_list(+Board, +Player, -ListOfMoves_0).
cpu_generate_move(+ListOfMoves, -NewBoard, -PositionMove).
convert_order_Move(+PositionMove, -Move).
map_redefined(:Pred, +L1, +L2, -L3).
map_redefined(:Pred, +L1, +L2, +L3, -L4).
max_pieces_adj(+Board, +Player, +Position,  -Value).
value(+Board, +Player, -Value).
max_list(+List, -Position, -Value).
max_list_aux(+List, -Position, -Value, +Position_tmp,
↪  +Value_tmp, +Ind).
```

4 Interface com o Utilizador

O módulo de interface é assegurado através de um conjunto de predicados que mostra ao utilizador texto e recebe do teclado os seus *inputs*. O programa dispõe de um menu principal, assegurado pelo predicado `display_main_menu /0`, que usa os predicados `display_title /0` e `display_options /0` para imprimir o título do jogo e as opções do menu, respetivamente.

A impressão do tabuleiro é feita com os predicados mencionados na secção 3.2, principalmente os predicados `display_board /1`e os predicados usados por este, nomeadamente `display_matrix /2`, `display_top /0` e `display_edge /0`. O número de peças que um jogador tem à sua disposição é mostrado pelo predicado `write_pieces /2`.

De forma a que a receção dos *inputs* do utilizador possa ser feita, são usados os predicados `get_boolean /2`, que mostra uma *prompt* ao utilizador e recebe um valor *yes/no*; `get_integer /4`, que também mostra uma *prompt* e recebe um valor inteiro dentro de um intervalo definido; e o predicado `get_move /2`, que por usa vez usa os predicados `get_edge /1` e `get_row /1` para receber a aresta e a fila para fazer uma jogada.

Os argumentos dos predicados enunciados nesta secção encontram-se declarados de seguida.

```
display_main_menu.
display_title.
display_options.
display_board(+Board).
display_matrix(+N, +Matrix).
display_top.
display_edge.
write_pieces(+Player, +Pieces).
get_boolean(+Prompt, -Option)
get_integer(+Prompt, +Min, +Max, -Option).
get_edge(-Edge).
get_row(-Row).
```

5 Conclusões

Com este projeto foi-nos possível entrar em contacto com um novo paradigma de programação bastante díspar aos que estávamos habituados até agora, introduzindo problemas completamente diferentes aos que já tivemos a oportunidade de resolver.

Foi um trabalho exigente, visto que encontrámos alguns problemas com, numa fase inicial, o planeamento da lógica do programa, para além de alguns *bugs* na implementação das jogadas do computador numa fase mais posterior, problemas estes que foram resolvidos eventualmente.

Consideramos que fizemos um bom trabalho, reconhecendo à mesma que este poderia, no entanto, ser melhorado no que toca ao seu desempenho; operações com manipulação, ordenação e comparação de listas conseguem ser dispendiosas, e é algo que se nota quando temos o computador a escolher uma jogada na dificuldade mais difícil.

Os elementos do grupo encaram este trabalho como uma experiência positiva, visto que foi possível alargar os nossos horizontes no que toca ao mundo da programação e ganhar experiência com outras ferramentas.

Bibliografia

[1] BoardGameGeek. Shiftago — board game — boardgamegeek. https://boardgamegeek.com/boardgame/199611/shiftago, 2016. Online em novembro de 2017.

[2] WiWa Spiele. Rules — shiftago - the strategic board game with shifting marbles. http://www.shiftago.com/en/rules.htm, 2016. Online em novembro de 2017.

[3] WiWa Spiele. Rules express — shiftago - the strategic board game with shifting marbles. http://www.shiftago.com/en/rules_express.htm, 2016. Online em novembro de 2017.

A Código-fonte

A.1 main.pl

```
:-use_module(library(lists)).
:-include('user_interface.pl').
:-include('utils.pl').
:-include('make_move.pl').
:-include('end_move.pl').
:-include('cpu.pl').


%---------------------- Loop geral do jogo ----------------------%
%main game function
shiftago:-
    display_main_menu,
    get_integer('Please choose an option: ', 0, 3, GameOption),
    menu_option(GameOption).

%exits game
menu_option(0).

%starts player vs player
menu_option(1):-
    asserta(game_mode(1)), asserta(cpu_level(0)),
    ↪  asserta(cpu_player(0)),
    init(Board, PlayerOnePieces, PlayerTwoPieces),
    player_vs_player(Board, PlayerOnePieces, PlayerTwoPieces, 1).

%starts player vs ai
menu_option(2):-
    asserta(game_mode(2)), once(get_cpu_difficulty),
    ↪  asserta(cpu_player(2)),
    init(Board, PlayerOnePieces, PlayerTwoPieces),
    player_vs_cpu(Board, PlayerOnePieces, PlayerTwoPieces, 1).

%starts ai vs ai
menu_option(3):-
    asserta(game_mode(3)), once(get_cpu_difficulty),
    ↪  asserta(cpu_player(0)),
    init(Board, PlayerOnePieces, PlayerTwoPieces),
    cpu_vs_cpu(Board, PlayerOnePieces, PlayerTwoPieces, 1).

%creates a blank board
init([[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],
↪  [0,0,0,0,0,0,0]], 22, 22).

%---------------------- Player vs Player ----------------------%
%shows the board, asks for the pieces, makes the move and then
↪  ends the current play
```

```
42 player_vs_player(Board, CurrentPieces, OpponentPieces,
   ↪ CurrentPlayer):-
43     display_board(Board),
44     write_pieces(CurrentPlayer, CurrentPieces),
45     insert_piece(Board, CurrentPlayer, NewBoard, CurrentPieces,
       ↪ NewCurrentPieces),
46     end_move(CurrentPlayer, NewBoard, OpponentPieces,
       ↪ NewCurrentPieces).
47
48 %----------------------- Player vs CPU -----------------------%
49 player_vs_cpu(Board, CurrentPieces, OpponentPieces,
   ↪ CurrentPlayer):-
50     cpu_player(CurrentPlayer), !,
51     display_board(Board),
52     cpu_move(Board, CurrentPlayer, Move, NewBoard, CurrentPieces,
       ↪ NewCurrentPieces),
53     display_move(CurrentPlayer, Move),
54     end_move(CurrentPlayer, NewBoard, OpponentPieces,
       ↪ NewCurrentPieces).
55
56 player_vs_cpu(Board, CurrentPieces, OpponentPieces,
   ↪ CurrentPlayer):-
57     display_board(Board),
58     write_pieces(CurrentPlayer, CurrentPieces),
59     insert_piece(Board, CurrentPlayer, NewBoard, CurrentPieces,
       ↪ NewCurrentPieces),
60     end_move(CurrentPlayer, NewBoard, OpponentPieces,
       ↪ NewCurrentPieces).
61
62 %------------------------ CPU vs CPU ------------------------%
63 cpu_vs_cpu(Board, CurrentPieces, OpponentPieces, CurrentPlayer):-
64     display_board(Board),
65     cpu_move(Board, CurrentPlayer, Move, NewBoard, CurrentPieces,
       ↪ NewCurrentPieces),
66     display_move(CurrentPlayer, Move),
67     end_move(CurrentPlayer, NewBoard, OpponentPieces,
       ↪ NewCurrentPieces).
```

A.2 make_move.pl

```
1  :-use_module(library(lists)).
2
3  %-------------------- Auxiliar predicates -------------------%
4
5  %checks if a move is possible
6  possible_move(Line):-
7      member(0, Line).
8
9  %inserts piece at the beginning of a line
10 insert_head_line(Player,Line,NLine):-
11     append([Player],Line,NLine).
12
```

```
13  %inserts piece at the end of a line
14  insert_end_line(Player,Line,NLine):-
15      append(Line,[Player],NLine).
16
17  %removes the first zero in a line
18  remove_first_zero(Line, NLine) :-
19          Line = [0 | Line2],
20          NLine = Line2.
21
22  remove_first_zero(Line, NLine) :-
23          Line = [X | Line2],
24          X \= 0,
25          NLine = [X | NLine2],
26          remove_first_zero(Line2, NLine2).
27
28  remove_first_zero([],[]).
29
30  %removes the last zero in a line
31  remove_last_zero(Line, NLine) :-
32          reverse(Line, _Line2),
33          remove_first_zero(_Line2, _NLine2),
34          reverse(_NLine2, NLine).
35
36
37  %------------------- Obtaining player move ------------------%
38  %asks the player for their move (board edge and line/column)
39  get_move(Edge, Row):-
40      repeat,
41          once(get_edge(Edge)), once(get_row(Row)).
42
43  %gets the edge
44  get_edge(Edge):-
45      write('Choose a board edge to insert the piece (up, down,
        ↪  left, right): '), nl,
46      read(Edge), member(Edge,['up', 'down', 'left', 'right']).
47
48  get_edge(Edge):-
49      write('Invalid edge; Try again.'), nl,
50      get_edge(Edge).
51
52  %gets the row
53  get_row(Row):-
54      get_integer('Choose a row: ', 1, 7,Row).
55
56
57  %---------------- Insert piece on board edge ----------------%
58  %inserts a piece in the board (predicate called by the main game
    ↪  predicate)
59  insert_piece(Board, CurrentPlayer, NewBoard, CurrentPieces,
    ↪  NewCurrentPieces):-
60          get_move(Edge, Row),
```

```
61          insert_piece(Board, Edge, Row, CurrentPlayer, NewBoard,
            ↪ CurrentPieces, NewCurrentPieces).
62
63  insert_piece(Board, left, N, CurrentPlayer, NewBoard,
    ↪ CurrentPieces, NewCurrentPieces):-
64      get_line(N, Board, Line), possible_move(Line),
65      remove_first_zero(Line, _line),
        ↪ insert_head_line(CurrentPlayer, _line, NLine),
66      replace_nth(Board, N, NLine, NewBoard),
67      NewCurrentPieces is CurrentPieces - 1.
68
69  insert_piece(Board, right, N, CurrentPlayer, NewBoard,
    ↪ CurrentPieces, NewCurrentPieces):-
70      get_line(N, Board, Line), possible_move(Line),
71      remove_last_zero(Line, _line), insert_end_line(CurrentPlayer,
        ↪ _line, NLine),
72      replace_nth(Board, N, NLine, NewBoard),
73      NewCurrentPieces is CurrentPieces - 1.
74
75  insert_piece(Board, up, N, CurrentPlayer, NewBoard,
    ↪ CurrentPieces, NewCurrentPieces):-
76      transpose(Board, _tempInit),
77      insert_piece(_tempInit, left, N, CurrentPlayer, _tempEnd,
        ↪ CurrentPieces, NewCurrentPieces),
78      transpose(_tempEnd, NewBoard).
79
80  insert_piece(Board, down, N, CurrentPlayer, NewBoard,
    ↪ CurrentPieces, NewCurrentPieces):-
81      transpose(Board, _tempInit),
82      insert_piece(_tempInit, right, N, CurrentPlayer, _tempEnd,
        ↪ CurrentPieces, NewCurrentPieces),
83      transpose(_tempEnd, NewBoard).
```

A.3 end_move.pl

```
1   %---------------- Verificacao de Fim de Jogo ---------------%
2
3   %initial predicate to check if player won
4   check_for_win(Pieces, Player, Board):-
5           Pieces =< 17, check_for_win(Player, Board),
            ↪ display_board(Board).
6
7   %checking lines columns and diagonals
8   check_for_win(Player, Board):- check_lines(Player,Board).
9   check_for_win(Player, Board):- check_colums(Player, Board).
10  check_for_win(Player, Board):- check_diagonals(Player, Board).
11  check_for_win(Player, Board):- reverse(Board, NBoard),
    ↪ check_diagonals(Player, NBoard).
12
13  %base model checks if a set of five consecutive player pieces
    ↪ exists in a line (easier checked in a list)
```

```
14 check_lines(Player, [X|_Rest]):-
   ↪ sublist([Player,Player,Player,Player,Player], X).
15 check_lines(Player, [_|Rest]):- check_lines(Player, Rest).
16
17 check_colums(Player, Board):- transpose(Board, TBoard),
   ↪ check_lines(Player, TBoard).
18
19 %obtaining a list of the coordinates where you can fit five
   ↪ diagonal consecutive pieces
20 check_diagonals(Player,Board) :- get_diagonal(1,1,Board,[],Line),
   ↪ check_lines(Player,[Line]).
21 check_diagonals(Player,Board) :- get_diagonal(1,2,Board,[],Line),
   ↪ check_lines(Player,[Line]).
22 check_diagonals(Player,Board) :- get_diagonal(1,3,Board,[],Line),
   ↪ check_lines(Player,[Line]).
23 check_diagonals(Player,Board) :- get_diagonal(2,1,Board,[],Line),
   ↪ check_lines(Player,[Line]).
24 check_diagonals(Player,Board) :- get_diagonal(3,1,Board,[],Line),
   ↪ check_lines(Player,[Line]).
25
26 %obtaining list with diagonal pieces
27 get_diagonal(8, _, _Board, Line, Line):- ! .
28 get_diagonal(_, 8, _Board, Line, Line):- ! .
29 get_diagonal(L, C, Board, Line, FLine):-
30     L < 8, C < 8, L1 is L+1, C1 is C+1,
31     get_line(L, Board, _tmp), nth1(C, _tmp, _value),
32     append(Line, [_value], NLine),
33     get_diagonal(L1,C1,Board,NLine, FLine).
34
35
36 %---------------------- Fim de Jogada ----------------------%
37
38 %ends move by first checking if the player one, then if the
   ↪ player lost and then
39 %changing the player and starting a new move
40
41 end_move(CurrentPlayer, NewBoard, _OpponentPieces,
   ↪ NewCurrentPieces):-
42     check_for_win(NewCurrentPieces, CurrentPlayer, NewBoard),
43     end_game(CurrentPlayer).
44
45 end_move(CurrentPlayer, NewBoard, _OpponentPieces, 0):-
46         switch_player(CurrentPlayer, NewPlayer),
47         display_board(NewBoard),
48         write('Player '), write(CurrentPlayer),
49         write(', you are out of pieces!'), nl,
50         end_game(NewPlayer).
51
52 end_move(CurrentPlayer, NewBoard, OpponentPieces,
   ↪ NewCurrentPieces):-
53     game_mode(1),
54     switch_player(CurrentPlayer, NewPlayer),
```

```
55      player_vs_player(NewBoard, OpponentPieces, NewCurrentPieces,
        ↪ NewPlayer).
56
57  end_move(CurrentPlayer, NewBoard, OpponentPieces,
    ↪ NewCurrentPieces):-
58      game_mode(2),
59      switch_player(CurrentPlayer, NewPlayer),
60      player_vs_cpu(NewBoard, OpponentPieces, NewCurrentPieces,
        ↪ NewPlayer).
61
62  end_move(CurrentPlayer, NewBoard, OpponentPieces,
    ↪ NewCurrentPieces):-
63      game_mode(3),
64      switch_player(CurrentPlayer, NewPlayer),
65      cpu_vs_cpu(NewBoard, OpponentPieces, NewCurrentPieces,
        ↪ NewPlayer).
66
67  %ends game and asks if the player would like to play again
68  end_game(Player):-
69      write('Game over, winner is player '), write(Player),
        ↪ write('!'), nl,
70      abolish(game_mode/1), abolish(cpu_player/1),
        ↪ abolish(cpu_level/1),
71      get_boolean('Would you like to return to the main menu?
        ↪ (yes/no)', Choice),
72      return_to_main_menu(Choice).
```

A.4 user_interface.pl

```
1   %---------------------- Print menu ----------------------%
2   %mostra o titulo do jogo
3   display_title:-
4       write('      _     _  __ _'), nl,
5       write('     | |   (_)/ _| |'), nl,
6       write('  ___| |__  _| |_| |_ __ _  __ _  ___'), nl,
7       write(' / __|  _ \\| |  _| __/ _` |/ _` |/ _ \\'), nl,
8       write(' \\__ \\ | | | | | | || (_| | (_| | (_) |'), nl,
9       write(' |___/_| |_|_|_|  \\__\\__,_|\\__, |\\___/ '), nl,
10      write('                            __/ |'), nl,
11      write('                           |___/'), nl.
12
13  %mostra as opcoes do menu
14  display_options:-
15      write('+-------------------------------------+'), nl,
16      write('  1. Player vs Player'), nl,
17      write('  2. Player vs COM'), nl,
18      write('  3. COM vs COM'), nl,
19      write('  0. Exit'), nl,
20      write('+-------------------------------------+').
21
22  %mostra o menu principal
23  display_main_menu:-
```

```
24      display_title, nl, display_options, nl.
25
26  %---------------------- Print board ----------------------%
27  %chamadas recursivas
28  display_matrix(8, _).
29  display_matrix(N, [L|T]) :-
30      N1 is N+1, N =< 7,
31      display_line(L, N),
32      display_matrix(N1, T).
33
34  %imprime a linha
35  display_line(L, N):-
36      write(N), put_char(' '),
37      put_char('|'), display_line(L).
38
39  %transforma caracteres internos a matriz em elementos
    ↪  representados
40  translate_char(0,'.').
41  translate_char(1,'X').
42  translate_char(2,'O').
43
44  %ultimo elemento da linha
45  display_line([X]) :- translate_char(X,N), write(N), write('|'),
    ↪  put_code(10).
46  display_line([X|R]) :- translate_char(X,N), write(N), write(' '),
    ↪  display_line(R).
47
48  %escreve o tabuleiro
49  display_top :- write('   1 2 3 4 5 6 7'), put_code(10).
50  display_edge :- write('  +-------------+'), put_code(10).
51  display_board(X):- display_top, display_edge, display_matrix(1,
    ↪  X), display_edge.
52
53  display_move(Player, Move):-
54      write('CPU player '), write(Player), write(' played '),
        ↪  write(Move), write('.'), nl.
55
56  %writes the number of pieces
57  write_pieces(Player, 1):-
58      write('Player '), write(Player),
59      write(', it\'s your turn. You only have one piece left, make
        ↪  it count!'), nl.
60
61  write_pieces(Player, Pieces):-
62      write('Player '), write(Player),
63      write(', it\'s your turn. You have '),
64      write(Pieces), write(' pieces left.'), nl.
65
66  %---------------------- Obter inputs ----------------------%
67  get_integer(Prompt, Min, Max, Option):-
68      write(Prompt), nl,
69      read(Option), integer(Option),
```

```
70      Option >= Min, Option =< Max.
71
72  get_integer(Prompt, Min, Max, Option):-
73      write('Invalid input; Try again.'), nl, get_integer(Prompt,
        ↪  Min, Max, Option).
74
75  get_boolean(Prompt, Option):-
76      write(Prompt), nl,
77      read(Option), member(Option, ['yes', 'no']).
78
79  %checks if the player wants to leave the game
80  return_to_main_menu(no).
81  return_to_main_menu(yes):-
82      shiftago.
83
84  get_cpu_difficulty:-
85      get_integer('Please choose a difficulty level for the CPU
        ↪  (1,2,3):', 1, 3, Level),
86      asserta(cpu_level(Level)).
87
88  /*
89     1 2 3 4 5 6 7
90    +-------------+
91  1 |. . O . X . .|
92  2 |. . . O X X O|
93  3 |. . . . O . .|
94  4 |. . . . X O X|
95  5 |. . . . . . O|
96  6 |X X . . . . .|
97  7 |. . . . O . .|
98    +-------------+
99  */
```

A.5 cpu.pl

```
1   /*
2   Move is represented by [edge , row]
3   */
4
5   :-use_module(library(lists)).
6   :- use_module(library(random)).
7
8   %---------------------- VALID_MOVE ----------------------%
9   %Gives the NewBoard for each Move. In case the move is not
    ↪  possible NewBoard is instantiated to 0
10  valid_move(Board, Player,  ['left', N], NewBoard) :-
11
12          N>0, N<8,
13          get_line(N, Board, Line), possible_move(Line),
14      remove_first_zero(Line, _line), insert_head_line(Player,
        ↪  _line, NLine),
15      replace_nth(Board, N, NLine, NewBoard),!.
```

```
16
17  valid_move(Board, Player,  ['right', N], NewBoard) :-
18
19          N>0, N<8,
20          get_line(N, Board, Line), possible_move(Line),
21      remove_last_zero(Line, _line), insert_end_line(Player, _line,
        ↪  NLine),
22      replace_nth(Board, N, NLine, NewBoard),!.
23
24  valid_move(Board, Player,  ['up', N], NewBoard) :-
25
26          N>0, N<8,
27          transpose(Board, Trans_Board),
28          valid_move(Trans_Board, Player, ['left',N],
            ↪  Trans_NewBoard),
29          transpose(Trans_NewBoard,NewBoard),!.
30
31  valid_move(Board, Player, ['down', N], NewBoard) :-
32
33          N>0, N<8,
34          transpose(Board, Trans_Board),
35          valid_move(Trans_Board, Player, ['right',N],
            ↪  Trans_NewBoard),
36          transpose(Trans_NewBoard,NewBoard),!.
37
38  valid_move(_Board, _Player, _Move, 0).
39
40
41  %--------------------- VALID_MOVES_MAKE_LIST
    ↪  ----------------------%
42  %Gives the list of valid moves for each Board and Player. A 0
    ↪  represents a move that is not valid
43  valid_moves_make_list_aux(Board, Player, ['left', 8],
    ↪  ListOfMoves_0):-
44          !,
45          valid_moves_make_list_aux(Board, Player, ['right',1],
            ↪  ListOfMoves_0).
46
47  valid_moves_make_list_aux(Board, Player, ['right', 8],
    ↪  ListOfMoves_0):-
48          !,
49          valid_moves_make_list_aux(Board, Player, ['up',1],
            ↪  ListOfMoves_0).
50
51  valid_moves_make_list_aux(Board, Player, ['up', 8],
    ↪  ListOfMoves_0):-
52          !,
53          valid_moves_make_list_aux(Board, Player, ['down',1],
            ↪  ListOfMoves_0).
54
55  valid_moves_make_list_aux(_Board, _Player, ['down', 8],
    ↪  ListOfMoves_0):-
```

```
56          !,
57          ListOfMoves_0=[].
58
59  valid_moves_make_list_aux(Board, Player,Move, ListOfMoves_0):-
60          Move=[Edge, X],
61          ListOfMoves_0=[NewBoard | L2],
62          valid_move(Board, Player, Move, NewBoard),
63          X1 is (X + 1),
64          valid_moves_make_list_aux(Board, Player, [Edge, X1], L2).
65
66
67  valid_moves_make_list(Board, Player, ListOfMoves_0):-
68          valid_moves_make_list_aux(Board, Player,['left',1],
            ↪  ListOfMoves_0).
69
70
71  %---------------------- VALID_MOVES ----------------------%
72  %Same as Valid_Moves, with the only difference that it eliminates
    ↪  the 0's from the list, so it only returns the moves that are
    ↪  actually valid
73  cmp_lists(ListOfMoves, ListOfMoves_0):-
74          (ListOfMoves=[], ListOfMoves_0=[]);
75          (ListOfMoves_0=[0|L_02], cmp_lists(ListOfMoves, L_02)).
76
77  cmp_lists(ListOfMoves, ListOfMoves_0):-
78          ListOfMoves_0=[X_0 | L_02],
79          X_0 \= 0,
80          ListOfMoves=[X | L2],
81          X=X_0,
82          cmp_lists(L2,L_02).
83
84
85
86  valid_moves(Board, Player, ListOfMoves):-
87          valid_moves_make_list(Board, Player, ListOfMoves_0),
88          cmp_lists(ListOfMoves, ListOfMoves_0).
89
90
91  %---------------------- MAX_PIECES_ADJ ----------------------%
92  %Returns the maximum number of adjacent pieces in the Board for a
    ↪  certain Player
93  max_pieces_adj_aux_main(Board, Player, Position, Value,
    ↪  Position_tmp, Value_tmp, Ind, Line):-
94       Ind1 is Ind + 1, rate_adj(Player, Line, X),(
95      (X > Value_tmp,
96      max_pieces_adj_aux(Board, Player, Position, Value, Ind, X,
        ↪  Ind1));
97      (\+ ( X > Value_tmp),
98      max_pieces_adj_aux(Board, Player, Position, Value,
        ↪  Position_tmp, Value_tmp, Ind1))).
99
```

```
100 max_pieces_adj_aux(_Board, _Player, Position, Value, Position,
    ↪ Value, 25):- ! .
101
102 max_pieces_adj_aux(Board, Player, Position, Value, Position_tmp,
    ↪ Value_tmp, Ind):-
103     Ind<8, !, nth1(Ind, Board, Line),
104     max_pieces_adj_aux_main(Board, Player, Position, Value,
        ↪ Position_tmp, Value_tmp, Ind, Line).
105
106 max_pieces_adj_aux(Board, Player, Position, Value, Position_tmp,
    ↪ Value_tmp, Ind):-
107     Ind < 15, !, N is (Ind - 7),length(Board, _Tam),
        ↪ transpose(Board, Trans_Board), nth1(N, Trans_Board,
        ↪ Line),
108     max_pieces_adj_aux_main(Board, Player, Position, Value,
        ↪ Position_tmp, Value_tmp, Ind, Line).
109
110 max_pieces_adj_aux(Board, Player, Position, Value, Position_tmp,
    ↪ Value_tmp, Ind):-
111         Ind<18, !,C is Ind-14, get_diagonal(1,C,Board,[],Line),
112         max_pieces_adj_aux_main(Board, Player, Position, Value,
            ↪ Position_tmp, Value_tmp, Ind, Line).
113
114 max_pieces_adj_aux(Board, Player, Position, Value, Position_tmp,
    ↪ Value_tmp, Ind):-
115         Ind<20, !,L is Ind-16, get_diagonal(L,1,Board,[],Line),
116         max_pieces_adj_aux_main(Board, Player, Position, Value,
            ↪ Position_tmp, Value_tmp, Ind, Line).
117
118 max_pieces_adj_aux(Board, Player, Position, Value, Position_tmp,
    ↪ Value_tmp, Ind):-
119         Ind<23, !,C is Ind - 19,reverse(Board, RBoard),
            ↪ get_diagonal(1,C,RBoard,[],Line),
120         max_pieces_adj_aux_main(Board, Player, Position, Value,
            ↪ Position_tmp, Value_tmp, Ind, Line).
121
122 max_pieces_adj_aux(Board, Player, Position, Value, Position_tmp,
    ↪ Value_tmp, Ind):-
123         Ind<25, !,L is Ind-21,reverse(Board, RBoard),
            ↪ get_diagonal(L,1,RBoard,[],Line),
124         max_pieces_adj_aux_main(Board, Player, Position, Value,
            ↪ Position_tmp, Value_tmp, Ind, Line).
125
126 max_pieces_adj(Board, Player, Position, Value):-
127         max_pieces_adj_aux(Board, Player, Position, Value, -1,
            ↪ -1000,1).
128
129 max_pieces_adj(0, _Player, -1, -1).
130
131
132 %-------------------- RATE_ADJ_FORMULA ----------------------%
```

```
133 %This function computes a formula that determines the value of a
    ↪ certain Line for a certain Player. This formula is calculated
    ↪ as the sum of all
134 %adjacent pieces in that line raised to the power of five
135 %So for example, if a Line is [1,1,1,0,2,2,1,1] and the player is
    ↪ 1 this formula gives the result 3^5+2^5=275. In case the
    ↪ player is 2, it gives
136 %the result 2^5=32.
137 rate_adj_formula_count(_Elem, [], N, Count, Sum,
    ↪ _ElemJustAppeared):-
138     N is Sum + Count^5.
139
140 rate_adj_formula_count(Elem, List, N, _Count, Sum,
    ↪ ElemJustAppeared):-
141     List=[X | L2],
142     ElemJustAppeared = 0,
143     X \= Elem,
144     rate_adj_formula_count(Elem, L2, N, 0, Sum, 0).
145
146 rate_adj_formula_count(Elem, List, N, _Count, Sum,
    ↪ ElemJustAppeared):-
147     List=[X | L2],
148     ElemJustAppeared = 0,
149     X = Elem,
150     rate_adj_formula_count(Elem, L2, N, 1, Sum, 1).
151
152 rate_adj_formula_count(Elem, List, N, Count, Sum,
    ↪ ElemJustAppeared):-
153     List=[X | L2],
154     ElemJustAppeared = 1,
155     X \= Elem,
156     Sum1 is Sum+Count^5,
157     rate_adj_formula_count(Elem, L2, N, 0, Sum1, 0).
158
159 rate_adj_formula_count(Elem, List, N, Count, Sum,
    ↪ ElemJustAppeared):-
160     List=[X | L2],
161     ElemJustAppeared = 1,
162     X = Elem,
163     C1 is Count + 1,
164     rate_adj_formula_count(Elem, L2, N, C1, Sum, 1).
165
166
167 rate_adj_formula(Elem, List, N):-
168         rate_adj_formula_count(Elem, List, N, 0, 0, 0).
169
170 %--------------------- VALUE_FORMULA ----------------------%
171 %This formula returns an intermediate level of a value of a board
    ↪ for a certain player.
172 %It is computed by calculating the sum of rate_adj_formula for
    ↪ each diagonal, line and column
173 %The final value of a Board is computed using the function value
```

```
174 value_formula_aux_main(Board, Player, Value, Count, Ind, Line):-
175      Ind1 is Ind + 1, rate_adj_formula(Player, Line, X),
176      NCount is Count+X,
177     value_formula_aux(Board, Player, Value, NCount, Ind1).
178
179 value_formula_aux(_Board, _Player, Value,  Value, 25):- ! .
180
181 value_formula_aux(Board, Player, Value, Value_tmp, Ind):-
182     Ind<8, !, nth1(Ind, Board, Line),
183     value_formula_aux_main(Board, Player, Value, Value_tmp, Ind,
        ↪  Line).
184
185 value_formula_aux(Board, Player, Value, Value_tmp, Ind):-
186     Ind < 15, !, N is (Ind - 7), transpose(Board, Trans_Board),
        ↪  nth1(N, Trans_Board, Line),
187     value_formula_aux_main(Board, Player, Value,  Value_tmp, Ind,
        ↪  Line).
188
189 value_formula_aux(Board, Player,  Value, Value_tmp, Ind):-
190         Ind<18, !,C is Ind-14, get_diagonal(1,C,Board,[],Line),
191         value_formula_aux_main(Board, Player,  Value,  Value_tmp,
            ↪  Ind, Line).
192
193 value_formula_aux(Board, Player,   Value, Value_tmp, Ind):-
194         Ind<20, !,L is Ind-16, get_diagonal(L,1,Board,[],Line),
195         value_formula_aux_main(Board, Player, Value,  Value_tmp,
            ↪  Ind, Line).
196
197 value_formula_aux(Board, Player,  Value, Value_tmp, Ind):-
198         Ind<23, !,C is Ind - 19,reverse(Board, RBoard),
            ↪  get_diagonal(1,C,RBoard,[],Line),
199         value_formula_aux_main(Board, Player,  Value,  Value_tmp,
            ↪  Ind, Line).
200
201 value_formula_aux(Board, Player, Value, Value_tmp, Ind):-
202         Ind<25, !,L is Ind-21,reverse(Board, RBoard),
            ↪  get_diagonal(L,1,RBoard,[],Line),
203         value_formula_aux_main(Board, Player,  Value,  Value_tmp,
            ↪  Ind, Line).
204
205 value_formula(Board, Player,  Value):-
206         value_formula_aux(Board, Player, Value, 0, 1).
207
208 %--------------------- VALUE ----------------------%
209 %This function returns the value of a Board for a certain player.
210 %It is computed by calculating the value_formula of the Board and
    ↪  subtracting the maximum value_formula that a the next player
    ↪  can obtain in the following move of his own
211 value(0, _Player, -10^7):- ! .
212
213 value(Board, Player, Value):-
```

```
214         max_pieces_adj(Board, Player, _PositionPlayer,
            ↪  ValuePlayer),
215         ValuePlayer >= 5, !,  Value is 10 ^ 6.
216
217 value(Board, Player, Value):-
218         NPlayer is mod(Player,2) + 1, valid_moves(Board, NPlayer,
            ↪  ListOfMoves), length(ListOfMoves, Size),
            ↪  create_list(NPlayer, Size, ListPlayer),
219         map_redefined(max_pieces_adj, ListOfMoves, ListPlayer,
            ↪  _Positions, Values),  max_list( Values,
            ↪  _PositionNPlayer, ValueNPlayer),
220         ValueNPlayer>=5, !, Value is (0 - 10^6).
221
222 value(Board, Player, Value):-
223         value_formula(Board, Player,  ValuePlayer),
224         NPlayer is mod(Player,2) + 1, valid_moves(Board, NPlayer,
            ↪  ListOfMoves), length(ListOfMoves, Size),
            ↪  create_list(NPlayer, Size, ListPlayer),
225         map_redefined(value_formula, ListOfMoves, ListPlayer,
            ↪  Values), max_list( Values, _PositionNPlayer,
            ↪  ValueNPlayer),
226         Value is ValuePlayer-ValueNPlayer.
227
228 %---------------------- CPU_MOVE -----------------------%
229 %Gives the Move that the cpu is going to make for a certain Board
    ↪  and Player
230 cpu_move( _Board, _Player, 'undefined', 'undefined', 0,  0):- ! .
231
232 cpu_move(Board, Player, Move, NewBoard, CurrentPieces,
    ↪  NewCurrentPieces):-
233         cpu_level(3),
234         valid_moves_make_list(Board, Player, ListOfMoves),
            ↪  length(ListOfMoves, Size), create_list(Player, Size,
            ↪  ListPlayer),
235         map_redefined( value, ListOfMoves, ListPlayer, Values),
            ↪  max_list(Values, PositionMove, ValueMove),
236         ((ValueMove is (0 -10^7), Move='undefined',
            ↪  NewBoard='undefined', NewCurrentPieces=0) ;
237         (ValueMove > (0 -10^7),convert_order_Move(PositionMove,
            ↪  Move),  nth1(PositionMove, ListOfMoves, NewBoard),
            ↪  NewCurrentPieces is CurrentPieces - 1)).
238
239 cpu_move(Board, Player, Move, NewBoard, CurrentPieces,
    ↪  NewCurrentPieces):-
240         cpu_level(2),
241         valid_moves_make_list(Board, Player, ListOfMoves),
            ↪  length(ListOfMoves, Size), create_list(Player, Size,
            ↪  ListPlayer),
242         map_redefined( max_pieces_adj, ListOfMoves, ListPlayer,
            ↪  _Positions, Values), max_list(Values, PositionMove,
            ↪  ValueMove),
```

```
243         ((ValueMove is (0 -10^7), Move ='undefined',
            ↪ NewBoard='undefined', NewCurrentPieces=0) ;
244         (ValueMove > (0 -10^7), convert_order_Move(PositionMove,
            ↪ Move), nth1(PositionMove, ListOfMoves, NewBoard),
            ↪ NewCurrentPieces is CurrentPieces - 1)).
245
246 cpu_move(Board, Player, Move, NewBoard, CurrentPieces,
    ↪ NewCurrentPieces):-
247         cpu_level(1),
248         valid_moves_make_list(Board, Player, ListOfMoves),
249         cpu_generate_move(ListOfMoves, NewBoard, PositionMove),
250         convert_order_Move(PositionMove, Move),  NewCurrentPieces
            ↪ is CurrentPieces - 1.
251
252 cpu_generate_move(ListOfMoves, NewBoard, PositionMove):-
253         (length(ListOfMoves, Size), random(1, Size,
            ↪ PositionMove), nth1(PositionMove, ListOfMoves,
            ↪ NewBoard), NewBoard \= 0);
254         cpu_generate_move(ListOfMoves, NewBoard, PositionMove).
255
256 %---------------------- CONVERT_ORDER_MOVE ----------------------%
257 %This function converts a number that represents the order of a
    ↪ Move in an array to the move itself
258 %The order of the Moves in the array is left->right->up->down, in
    ↪ ascendant order for the numbers
259 convert_order_Move(N, Move):-
260                 N<8, !,
261                 Move=['left', N].
262
263 convert_order_Move(N, Move):-
264                 N<15, !,  N1 is N-7,
265                 Move=['right', N1].
266
267 convert_order_Move(N, Move):-
268                 N<22, !,  N1 is N-14,
269                 Move=['up', N1].
270
271 convert_order_Move(N, Move):-
272                 N<29, !,  N1 is N-21,
273                 Move=['down', N1].
```

A.6 utils.pl

```
1 :-use_module(library(lists)).
2
3 %---------------------- SUBLIST ----------------------%
4 %Verifies if a list L1 is a member of a sublist L
5 sublist_start([],_L):- ! .
6 sublist_start(L1, L):- L1=[X | L12], L=[X | L2],
    ↪ sublist_start(L12 , L2).
7
8 sublist(L1, L) :- sublist_start(L1, L).
```

```
9
10  sublist(L1,L) :- L=[_X | L2], sublist(L1 , L2).
11
12  %switches the two players
13  switch_player(1,2).
14  switch_player(2,1).
15
16  %---------------------- RATE ----------------------%
17  %Counts the number of times that Elem occurs in List
18  rate_count(_Elem, [], N, N):- ! .
19
20  rate_count(Elem, List, N, Count):-
21      List=[X | L2],
22      ((X=Elem,C1 is Count + 1, rate_count(Elem, L2, N, C1));
23      (X \= Elem, rate_count(Elem, L2, N, Count))).
24
25
26
27  rate(Elem, List, N):-
28      rate_count(Elem, List, N, 0).
29
30
31  %---------------------- RATE_ADJ ----------------------%
32  %Gives the maximum sublist whose only element is Elem that occurs
    ↪  in List
33  rate_adj_count(_Elem, [], N, Count, Max_Count,
    ↪  _ElemJustAppeared):-
34      (Count > Max_Count, N=Count) ;
35      ( \+ (Count > Max_Count), N=Max_Count).
36
37  rate_adj_count(Elem, List, N, _Count, Max_Count,
    ↪  ElemJustAppeared):-
38      List=[X | L2],
39      ElemJustAppeared = 0,
40      X \= Elem,
41      rate_adj_count(Elem, L2, N, 0, Max_Count, 0).
42
43  rate_adj_count(Elem, List, N, _Count, Max_Count,
    ↪  ElemJustAppeared):-
44      List=[X | L2],
45      ElemJustAppeared = 0,
46      X = Elem,
47      rate_adj_count(Elem, L2, N, 1, Max_Count, 1).
48
49  rate_adj_count(Elem, List, N, Count, Max_Count,
    ↪  ElemJustAppeared):-
50      List=[X | L2],
51      ElemJustAppeared = 1,
52      X \= Elem,
53      ((Count>Max_Count,
54      rate_adj_count(Elem, L2, N, 0, Count, 0));
55      (\+ (Count>Max_Count),
```

```
56      rate_adj_count(Elem, L2, N, 0, Max_Count, 0))).
57
58  rate_adj_count(Elem, List, N, Count, Max_Count,
    ↪  ElemJustAppeared):-
59      List=[X | L2],
60      ElemJustAppeared = 1,
61      X = Elem,
62      C1 is Count + 1,
63      rate_adj_count(Elem, L2, N, C1, Max_Count, 1).
64
65
66  rate_adj(Elem, List, N):-
67          rate_adj_count(Elem, List, N, 0, 0, 0).
68
69
70  %---------------------- MAX_LIST ----------------------%
71  %Gives the Value and Position of the maximum number that occurs
    ↪  in a list
72  max_list_aux([], Position, Value, Position, Value, _Ind):- ! .
73
74  max_list_aux(List, Position, Value, Position_tmp, Value_tmp,
    ↪  Ind):-
75      List=[X | L2],
76      Ind1 is Ind + 1,(
77      (X > Value_tmp,
78      max_list_aux(L2, Position, Value, Ind, X, Ind1));
79      (\+ ( X > Value_tmp),
80      max_list_aux(L2, Position, Value, Position_tmp, Value_tmp,
        ↪  Ind1))).
81
82
83  max_list(List, Position, Value):-
84      Min is 0-10^8,
85      max_list_aux(List,Position, Value, -1, Min, 1).
86
87  %---------------------- GET_LINE ----------------------%
88  %Gives the N th line of Board
89  get_line(N, Board, Line):-
90      nth1(N, Board, Line).
91
92  %---------------------- REPLACE_NTH ----------------------%
93  %Replaces the nth element of a list
94  replace_nth([_|Rest], 1, X, [X|Rest]).
95  replace_nth([X|Rest], N, Elem, [X|NRest]):-
96      N > 1, N1 is N-1, replace_nth(Rest, N1, Elem, NRest).
97
98
99  %---------------------- CREATE_LIST ----------------------%
100 %Creates a list of size N, whose only element is Elem
101 create_list(_Elem, 0, []):- !.
102
103 create_list(Elem, N, List):-
```

```
104     List=[X | L2],
105     X=Elem,
106     N1 is N-1,
107     create_list(Elem, N1, L2).
108
109 %---------------------- MAP_REDEFINED ----------------------%
110 %Redefines the function map_redefined for 4 and 5 elements
111 map_redefined(_Pred, [], [], []):- ! .
112
113 map_redefined(Pred, L1, L2, L3):-
114     L1=[X1 | L12],
115     L2=[X2 | L22],
116     L3=[X3 | L32],
117
118     call(Pred, X1, X2, X3),
119     map_redefined(Pred, L12, L22, L32).
120
121 map_redefined(_Pred, [], [], [], []):- ! .
122
123 map_redefined(Pred, L1, L2, L3, L4):-
124
125     L1=[X1 | L12],
126     L2=[X2 | L22],
127     L3=[X3 | L32],
128     L4=[X4 | L42],
129     call(Pred, X1, X2, X3, X4),
130     map_redefined(Pred, L12, L22, L32, L42).
```