DOMIN-UP

RELATÓRIO FINAL

MIEIC 2015/2016 – 3ª ANO/1º SEMESTRE : PLOG

Grupo: *Dominup_2*
Pedro Pais de Sousa da Costa Carvalho - *up201306506@fe.up.pt*
Pedro Miguel Pereira de Melo - *up201305618@fe.up.pt*

ÍNDICE:

1.Introdução 2
2. O Jogo Domin-Up 3
3. Lógica do Jogo 5
3.1 Representação do estado do jogo 6
3.2 Visualização do Tabuleiro 7
3.3 Lista de Jogadas Válidas 11
3.4 Execução de Jogadas 14
3.5 Final do Jogo 16
3.6 Jogada do Computador 17
4. Interface com o Utilizador 19
5. Conclusões 21
Anexo A – Código do Domin-Up 22

1.Introdução

Este relatório foi realizado no âmbito do primeiro projeto prático da cadeira de Programação em Lógica. O principal objetivo do projeto é a implementação das regras de um jogo utilizando a linguagem PROLOG. Na especificação do projeto foi também pedido que se desenvolvesse uma interface em linha de texto para o mesmo jogo, por forma a poder ser jogado por dois utilizadores humanos, um utilizador humano contra o computador, ou o computador contra si próprio. A especificação requere ainda que o computador jogue com vários níveis de dificuldade.

O presente relatório tem a seguinte estrutura:

- ✓ **Introdução**: onde são descritos os objetivos e a motivação do trabalho;
- ✓ **O Jogo *Domin-Up***: onde é feita uma descrição sucinta do jogo, a sua história e as suas regras;
- ✓ **Representação do Estado do Jogo**: onde são descritas as estruturas de dados utilizadas para representar o estado do jogo;
- ✓ **Visualização do Tabuleiro**: onde são descritas os predicados utilizados para visualizar o conteúdo do tabuleiro;
- ✓ **Lista de Jogadas Válidas**: onde são descritas os predicados utilizados na validação de jogadas e na obtenção das jogadas possíveis para um dado jogador;
- ✓ **Execução de Jogadas**: onde são descritos os predicados utilizados para efetuar as jogadas e atualizar o tabuleiro;
- ✓ **Final do Jogo**: onde são descritos os predicados utilizados quando o jogo termina, onde é revelado o jogador vencedor;
- ✓ **Jogada do Computador**: onde são descritos os predicados utilizados para permitir que o computador faça escolhas e tome estratégias para jogar contra um humano ou contra si mesmo;
- ✓ **Interface com o Utilizador**: onde é descrita a interface implementada para o jogo;
- ✓ **Conclusões**: onde são descritas as conclusões retiradas da execução do projeto.

2. O Jogo Domin-Up

O *Domin-Up* foi criado em 2015 e é uma variante do popular *Dominó*, havendo grandes semelhanças no modo em como ambos os jogos são jogados. Uma partida de *Domin-Up* requere um mínimo de 2 jogadores podendo ser jogada, no máximo, por 4 jogadores em simultâneo.

As peças utilizadas nas partidas de *Domin-Up* assemelham-se também ás peças utilizadas nas partidas de *Dominó*; havendo um total de 36 peças, com formato retangular. Cada uma destas peças é única, estando assinalada em ambas as metades da sua face superior com um inteiro entre 0 e 7 (inclusive), representado em escrita binária com recurso a círculos concêntricos, num padrão único.

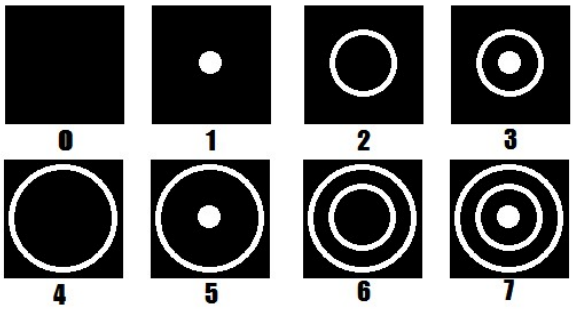


Fig 1. *Os diferentes símbolos visíveis nas peças do Domin-Up.*

O *Domin-Up* não possuí um tabuleiro próprio, podendo deste modo ser jogado sobre qualquer superfície/ambiente.

No início do jogo, é atribuído a cada jogador um igual número de peças, distribuídas aleatoriamente – a *mão* desse jogador. Cabe a cada jogador o dever de guardar em segredo dos seus adversários o conteúdo da sua mão.

O objetivo do jogo consiste em colocar em jogo todas as peças da mão, vencendo o jogador que o conseguir fazer mais rapidamente. Cada jogador deve colocar as suas peças em jogo por forma a impedir que os seus adversários o possam fazer mais rapidamente, ou por forma a complicar a concretização de uma possível estratégia inimiga.

As diferentes jogadas processam-se no sentido contrário ao dos ponteiros do relógio, jogando em primeiro lugar o jogador que tiver a peça na sua mão a peça com ambas as faces assinaladas com o número 7. Esta peça é também a primeira a ser jogada.

Em cada jogada pode-se apenas colocar em jogo uma única peça, de acordo com as regras de colocação de peças. Existem duas maneiras válidas de colocar peças em jogo:

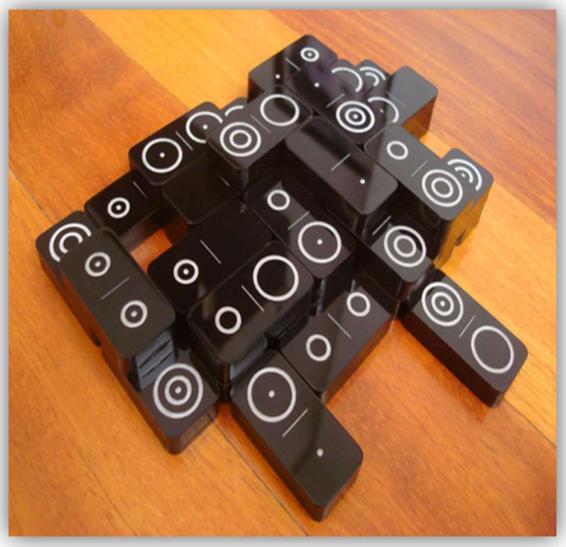


Fig 2. *Um jogo de Domin-Up.*

- ✓ ***Climb*** **–** em que a peça jogada é colocada sobre outras peças já jogadas, de tal forma que os números nas suas faces sejam idênticos aos números nas faces a serem cobertas;
- ✓ ***Expand*** – em que a peça jogada é colocada ao lado de outras peças já jogadas, de tal forma as faces das peças em contacto tenham o mesmo número aí representado. Esta é a clássica jogada do *Dominó*;
- ✓ Um jogador só deve realizar uma jogada do tipo *Expand* quando não lhe for possível realizar uma do tipo *Climb*.

Se um jogador não conseguir "legalmente" colocar nenhuma peça em jogo perde o turno, ficando à espera pela próxima oportunidade de jogar.

3. Lógica do Jogo

O código do *Domin-Up* foi subdividido em vários ficheiros, com o objetivo de melhorar a estrutura do projeto e a permitir uma maior facilidade no seu desenvolvimento. Estes ficheiros são os seguintes:

displays.pl – contem todas os predicados relativos quer ao desenho da interface com o utilizador – *menu_principal*, *main_menu_opcoes*, *menu_dificuldade* e *menu_regras* - quer ao desenho do tabuleiro, das mãos dos jogadores e dos seus conteúdos – *mostra_tabuleiro* e *mostra_mao_jogador*. Contem ainda o predicado *cls,* responsável pela limpeza do terminal;

auxiliar.pl – contem todas os predicados necessários ao manipulamento de listas – *list_element_at*, *list_delete_one* e *matrix_setCell* -, ao redimensionamento automático do tabuleiro – *tabuleiro_dimensiona* - bem como o predicado *readInt* , utilizado para ler dados inseridos pelo utilizador;

cpu.pl – contem todos os predicados necessárias para que o computador possa jogar contra um ser humano – *cpu_uma_ao_calhas*;

code.pl – contem o *main loop* do *Domin-Up.* Para além disso contém a inicialização do tabuleiro e das mãos dos jogadores, os predicados para gerir estas estruturas de dados – *mao_acrescentar_peca, mao_remover_peca* e *tabuleiro_jogar_peca* -, os predicados *jogador_trocar_vez* e *jogador_pode_jogar* que recorrem às regras do jogo para averiguar se uma jogada é válida, permitindo que esta se efetue, assim como a troca de turnos.

Note-se que os predicados acima referidos realizam chamadas a outros vários predicados, que embora não tendo sido descritos nesta secção, serão descriminados no decorrer deste relatório e poderão ser consultados no Anexo A deste documento.

3.1 Representação do estado do jogo

Cada estado do jogo *Domin-Up* é caracterizado pela composição do tabuleiro num dado instante da partida e do jogador a jogar nesse momento.

```
estado(T, J) :- tabuleiro(T), jogador_escolhido(J).
```

Cada jogador pode ser representado pelo seu nome.

```
jogador(jogador1).
jogador(jogador2).
```

Uma peça pode ser representada por uma lista com dois valores, correspondentes aos dois números inscritos nas faces da mesma peça.

O baralho que inicialmente contém todas as peças do jogo, pelo que pode ser visto como uma lista de listas, dado ser uma lista de peças. Como tal foi definido da seguinte forma:

```
baralho([ [0|0], [0|1], [0|2], [0|3], [0|4], [0|5], [0|6], [0|7], [1|1], [1|2],
                        [1|3], [1|4], [1|5], [1|6], [1|7], [2|2], [2|3], [2|4], [2|5], [2|6],
                        [2|7], [3|3], [3|4], [3|5], [3|6], [3|7], [4|4], [4|5], [4|6], [4|7],
                        [5|5], [5|6], [5|7], [6|6], [6|7], [7|7] ]).
```

Da mesma forma uma mão do jogador pode ser visto como uma lista de listas, dado ser um lista de peças – as peças que o jogador tem na sua posse.

Por sua vez, o tabuleiro do jogo pode ser visto como uma lista de listas, sendo cada lista um conjunto de peças. Assim sendo, o tabuleiro foi implementado da seguinte forma:

```
tabuleiro([ [[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|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],[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|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],[0|0],[0|0]],
                        [[0|0],[0|0],[0|0],[0|0],[0|0],[0|0],[0|0],[0|0],[0|0],[0|0]] ]).
```

3.2 Visualização do Tabuleiro

Para representar o tabuleiro, utilizaram-se os predicados mostra_peca, mostra_linha, mostra_tabuleiro, mostra_separador e mostra.

O predicado mostra_peca foi criado para mostrar exclusivamente o conteúdo de uma posição do tabuleiro, isto é, o valor inscrito na face da peça aí colocada e a altura a que essa mesma peça se encontra. O único argumento deste predicado é uma lista constituída pelos seguintes elementos:

- ✓ **V** – o valor (de 0 a 7) inscrito na face da peça;
- ✓ **H** – a altura a que a peça de se encontra.

Este predicado foi implementado da seguinte forma:

```
mostra_peca([_|0]) :- write('     | |').
mostra_peca([V|H]) :- write('  '),
                      write(V),
                      write('  |'),
                      write(H),
                      write('|').
```

O predicado mostra_linha foi criado para mostrar exclusivamente uma linha do tabuleiro. Como uma linha nada mais é do que uma lista de posições, o predicado mostra_linha recorre a chamadas ao predicado mostra_peca. O único argumento deste predicado é uma lista constituída pelos seguintes elementos:

- ✓ **P** – representa uma lista que é passada como argumento ao predicado *mostra_peca*, estando em concordância com a descrição prévia dos argumentos para este predicado;
- ✓ **R** – as restantes posições do tabuleiro na linha.

Este predicado foi implementado da seguinte forma:

```
mostra_linha([]).
mostra_linha([P|[]]) :- P \= [],
                        mostra_peca(P).

mostra_linha([P|R]) :-  P \= [],
                        mostra_peca(P),
                        mostra_linha(R).
```

O predicado *mostra_tabuleiro* foi criado para mostrar todo o ambiente de jogo, recorrendo desta forma a várias chamadas ao predicado *mostra_linha*.

O único argumento deste predicado é uma lista constituída pelos seguintes elementos:

- ✓ **L** – lista que representa uma linha do tabuleiro, que é passada como argumento ao predicado *mostra*, estando em concordância com a descrição prévia dos argumentos para este predicado;
- ✓ **R** – lista que contém as restantes linhas do tabuleiro por mostrar.

Este predicado foi implementado da seguinte forma:

```
mostra_tabuleiro([L|R]) :- length(L,N),
                           mostra_N_col(0, N), nl,
                           mostra(1, [L|R], N),
                           nl.
```

É possível ver que este predicado chama por sua vez o predicado *mostra_N_col* e ao predicado *mostra*. O predicado *mostra_N_col* é responsável por mostrar o número da coluna associado a cada posição no tabuleiro.

Os seus argumentos são:

- ✓ **N** – o número da coluna a ser escrito;
- ✓ **L –** a dimensão do tabuleiro.

Este predicado foi implementado da seguinte forma:

```
mostra_N_col(N,L) :-
        N > L,
        true.

mostra_N_col(N,L) :-
        N = 0,
        N =< L,
        NN is N + 1,
        write('--+'),
        mostra_N_col(NN, L).
```

```
mostra_N_col(N,L) :-
        N < 10,
        N =< L,
        NN is N + 1,
        write('  '),
        write(N),
        write('  +-+'),
        mostra_N_col(NN, L).
```

```
mostra_N_col(N,L) :-
        N >= 10,
        N =< L,
        NN is N + 1,
        write(' '),
        write(N),
        write('  +-+'),
        mostra_N_col(NN, L).
```

O predicado *mostra* é responsável para mostrar o conteúdo de uma linha e os separados utilizados. Os seus argumentos são:

- ✓ **N** – o número da linha a ser escrito;
- ✓ **[LR]** – representa o tabuleiro;
- ✓ **LL** - a largura das linhas.

Este predicado foi implementado da seguinte forma:

```
mostra(_, [], LL) :- mostra_separador(0,LL).
mostra(N, [L|R], LL) :- NN is N+1,
                        N >= 10,
                        mostra_separador(0,LL),
                        write(N), write('|'),
                        mostra_linha(L),
                        nl,
                        mostra(NN, R, LL).
mostra(N, [L|R], LL) :- NN is N+1,
                        N < 10,
                        mostra_separador(0,LL),
                        write(N), write(' |'),
                        mostra_linha(L),
                        nl,
                        mostra(NN, R, LL).
```

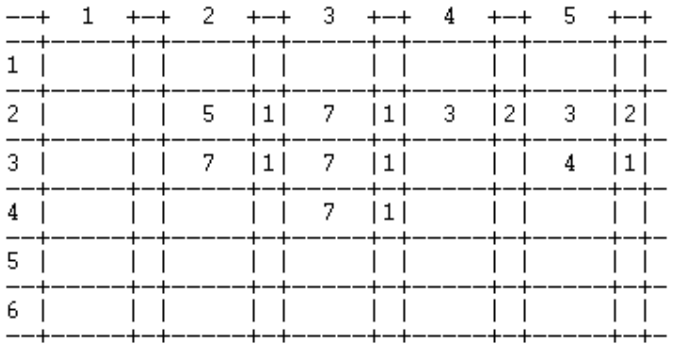
No início de uma partida de *Domin-Up*, em que ainda nenhuma peça tenha sido jogada, ao chamar-se o predicado *mostra_tabuleiro* é mostrado o seguinte:

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

Após a distribuição inicial das peças e a colocação da peça [7|7] no centro do tabuleiro pode ver-se:

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

Após várias jogadas, uma chamada ao predicado *mostra_tabuleiro* pode revelar o seguinte:



As células de maiores dimensões guardam os valores inscritos nas faces das peças, enquanto as células de menores dimensões guardam os valores das alturas das faces das peças.

3.3 Lista de Jogadas Válidas

Para se obter a lista de jogadas válidas, é necessário verificar se uma jogada é válida, pelo que se criaram os predicados *tabuleiro_pode_jogar_peca_expand* e *tabuleiro_pode_jogar_peca_climb***.** Ambos os predicados possuem os mesmos argumentos:

- ✓ **[V1|V2]** – a peça a ser jogada;
- ✓ **[C1|L1]** – a coluna C1 e a linha L1 em que se pretende colocar a parte da peça com o valor V1;
- ✓ **[C2|L2]** – a coluna C2 e a linha L2 em que se pretende colocar a parte da peça com o valor V2.

O predicado *tabuleiro_pode_jogar_peca_climb* verifica se a peça passada como argumento pode ser jogada com recurso a um movimento do tipo *Climb*, tendo sido implementado da seguinte forma:

```
tabuleiro_pode_jogar_peca_climb([V1|V2], [C1|L1], [C2|L2]) :-
        tabuleiro(T),
        %buscar os valores no tabuleiro
                tabuleiro_get(T, [C1|L1], TV1, TA1),
                tabuleiro_get(T, [C2|L2], TV2, TA2),
                !,
        %as coordenadas devem estar a tocar-se
                tabuleiro_distancia_coordenadas([C1|L1], [C2|L2], D),
                D == 1,
        %as alturas devem ser iguais
                TA1 == TA2,
        %a altura não pode ser 0
                TA1 \= 0,
        %os valores no tabuleiro devem ser iguais ao da peça
                TV1 == V1,
                TV2 == V2.
```

Note-se que basta apenas verificar se os valores das peças nas coordenadas [C1|L1] e [C2|L2] correspondem aos valores V1 e V2 respetivamente, assim como se a altura das peças é a mesma.

O predicado *tabuleiro_pode_jogar_peca_expand* verifica se a peça passada como argumento pode ser jogada com recurso a um movimento do tipo *Expand*, tendo sido implementado da seguinte forma:

```
tabuleiro_pode_jogar_peca_expand([V1|V2], [C1|L1], [C2|L2]) :-
        tabuleiro(T),
        %buscar os valores no tabuleiro
                tabuleiro_get(T, [C1|L1], _, TA1),
                tabuleiro_get(T, [C2|L2], _, TA2),
                !,
        %as coordenadas devem estar a tocar-se
                tabuleiro_distancia_coordenadas([C1|L1], [C2|L2], D),
                D == 1,
        %as alturas devem ser iguais
                TA1 == TA2,
        %uma das peças circundantes devem ter o mesmo valor
                A is TA1 + 1,
                (
                        expand_aux(V1, [C1|L1], A, T);
                        expand_aux(V2, [C2|L2], A, T)
                ).
```

Este predicado necessita de verificar se os valores entre as peças estão em contacto – o que é feito recorrendo ao predicado *tabuleiro_distancia_coordenadas* - coincidem e se a altura destas é a mesma. Este predicado invoca o predicado *expand_aux* que verifica se o movimento é possível em todas as direções.

Para obter a lista de jogadas válidas por *Expand* criou-se o predicado *jogador_jogadas_disponiveis_expand*, cujo único argumento corresponde ao jogador que está atualmente a jogar. Este predicado procura todas as possíveis jogadas por Expand para todas as peças na mão do jogador. Este predicado foi implementado da seguinte forma:

```
jogador_jogadas_disponiveis_expand(J) :-
        jogador(J),
    mao(J, M),
        length(M, L),
        (
                jogador_jogadas_disponiveis_expand_aux_linhas(L,M);
                jogador_jogadas_disponiveis_expand_aux_colunas(L,M)
        ).
```

Os predicados auxiliares chamados percorrem respetivamente o tabuleiro em par de linhas e em par de colunas, invocando o predicado *tabuleiro_pode_jogar_expand*.

Para obter a lista de jogadas válidas por *Climb* criou-se o predicado *jogador_jogadas_disponiveis_climb*, cujo único argumento corresponde ao jogador que

está atualmente a jogar. Este predicado procura todas as possíveis jogadas por Climb para todas as peças na mão do jogador. Este predicado foi implementado da seguinte forma:

```
jogador_jogadas_disponiveis_climb(J) :-
        jogador(J),
    mao(J, M),
        length(M, L),
        (
                jogador_jogadas_disponiveis_climb_aux_linhas(L,M);
                jogador_jogadas_disponiveis_climb_aux_colunas(L,M)
        ).
```

Os predicados auxiliares chamados percorrem respetivamente o tabuleiro em par de linhas e em par de colunas, invocando o predicado *tabuleiro_pode_jogar_climb.*

3.4 Execução de Jogadas

No início de cada jogada é mostrado ao jogador o estado atual do tabuleiro e o conteúdo da sua mão. O método de visualização permite que o jogador saiba quantas peças tem ainda na sua mão e qual o índice.

```
Mao do Jogador a:
+ 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10+ 11+ 12+ 13+ 14+ 15+ 16+ 17+
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0 | 1 | 0 | 0 | 0 | 2 | 6 | 3 | 4 | 1 | 1 | 5 | 4 | 1 | 4 | 2 | 5 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 7 | 4 | 3 | 1 | 6 | 7 | 7 | 5 | 6 | 5 | 7 | 7 | 5 | 3 | 7 | 6 | 5 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

De seguida, e se tiver jogadas válidas, o jogador terá de escolher uma peça para jogar. Esta peça é depois retornada pelo predicado *mao_escolher_peca,* implementado da seguinte forma:

```
mao_escolher_peca(J, X, P) :-
        jogador(J),
        mao(J,M),
        list_element_at(P, M, X).
```

De seguida o utilizador deve dizer ainda as coordenadas onde deseja colocar a peça. Se a jogada resultante for válida - o que é verificado pelos predicados previamente descritos, a peça será colocada no tabuleiro pelo predicado *tabuleiro_jogar_peca*. Este predicado foi desenvolvido com o intuito de colocar uma peça no tabuleiro, sendo feito o *assert* deste "novo" tabuleiro e *retract* do tabuleiro "antigo". Este predicado tem os seguintes argumentos:

- ✓ **[V1|V2]** – a peça a ser jogada;
- ✓ **[C1|L1]** – a coluna C1 e a linha L1 em que se pretende colocar a parte da peça com o valor V1;
- ✓ **[C2|L2]** – a coluna C2 e a linha L2 em que se pretende colocar a parte da peça com o valor V2.

O predicado *tabuleiro_jogar_peca* foi implementado da seguinte forma:

```
tabuleiro_jogar_peca([V1|V2], [C1|L1], [C2|L2]) :-
        tabuleiro(TI),
        tabuleiro_get(TI,[C1|L1], _, A1),
        tabuleiro_get(TI,[C2|L2], _, A2),
        A1 == A2,
        A is A1+1,
        tabuleiro_set(TI, [C1|L1], V1, A, TF),
        retract(tabuleiro(TI)),
        assert(tabuleiro(TF)),
        tabuleiro_set(TF, [C2|L2], V2, A, TFF),
        retract(tabuleiro(TF)),
        assert(tabuleiro(TFF)).
```

No seu corpo são invocados os predicados *tabuleiro_get* e *tabuleiro_set* que permitem respetivamente obter e alterar o valor de uma coordenada específica do tabuleiro. Estes foram implementados da seguinte forma:

```
tabuleiro_set(TI, [C|L], V, A, TF) :-
        matrix_setCell(L, C, TI, [V|A], TF).
```

```
tabuleiro_get(T,[C|L], V, A) :-
        C > 0,
        L > 0,
        list_element_at(Y, T, L),
        list_element_at([V|A], Y, C).
```

Após cada jogada é executado o predicado *jogador_trocar_vez* que muda de turno. É ainda removida a peça da mão do jogador, invocando o predicado *mao_remover_peca* implementado da seguinte forma:

```
mao_remover_peca(J, P) :-
        jogador(J),
        mao(J,MV),
        list_delete_one(P, MV, MN),
        retract(mao(J,MV)),
        assert(mao(J,MN)).
```

3.5 Final do Jogo

No *main loop* do *Domin-Up* é chamado o predicado *mao_vazia* a cada jogada sobre a mão do jogador a jogar, que verifica se o jogador ainda tem alguma peça na sua mão. Este predicado tem um argumento J, que é o jogador, e foi implementado da seguinte forma:

```
mao_vazia(J) :-
        jogador(J),
        mao(J, M),
        length(M,0).
```

Caso a mão do jogador esteja vazia, então é porque este é o vencedor da partida. É então chamado o predicado *main_victoria* cujo argumento é o jogador vencedor. Este predicado foi implementado da seguinte forma:

```
main_victoria(J) :-
        cls,
        nl,nl,nl,
        write('                VICTORIA DO JOGADOR '), write(J),nl,nl,
        write('tabuleiro final:'), nl,
        tabuleiro(T),
        mostra_tabuleiro(T).
```

É assim disponibilizado o nome do jogador vencedor e o estado final do tabuleiro. Note-se que na implementação do *Domin-Up* apenas se consideraram dois jogadores; o "jogador a" e o "jogador b".

3.6 Jogada do Computador

Os computadores, tanto os de dificuldade fácil como os de dificuldade difíceis, durante a sua jogada partilham grande parte dos passos com os turnos do jogador humano. A execução da jogada (pousar a peça no tabuleiro), a verificação da jogada, o trocar do turno e expansão do tabuleiro ocorrem todos da mesma forma.

No entanto, enquanto o jogador consegue fazer uma verificação de uma jogada escolhida rapidamente, e normalmente o jogador humano não perde tempo a pedir jogadas inválidas, o computador tem de ser capaz de formular jogadas que façam sentido perante o tabuleiro, e que depois ainda sejam aleatórias ou pensadas consoante a mão do jogador oponente. Portanto no passo da Decisão da Peça a Jogar, ao contrário do que acontece com o jogador, esta ocorre após ser feita uma lista de jogadas válidas.

Foram feitos os predicados *cpu_todas_jogadas* que, através da chamada *findall* consegue obter com outras funções acima faladas, todas as soluções válidas para as jogadas do computador. A obtenção das soluções do tipo *Climb e Expand* foram separadas no predicado principal *cpu_uma_ao_calhas* para que os jogadores CPU deem prioridade a esse tipo de jogadas.

```
cpu_uma_ao_calhas(J,G,T) :-
    cpu_todas_jogadas_climb(J,G,T),
    G \= [].
cpu_uma_ao_calhas(J,G,T) :-
    cpu_todas_jogadas_expand(J,G,T),
    G \= [].

cpu_todas_jogadas_climb(J, G, T) :-
    jogador(J),
    findall([P,CL1,CL2],cpu_uma_jogada_climb(J,P,CL1,CL2,T),G).

cpu_todas_jogadas_expand(J, G, T) :-
    jogador(J),
    findall([P,CL1,CL2],cpu_uma_jogada_expand(J,P,CL1,CL2,T),G).
```

Já para um computador difícil, isto não chega. Como o computador deve ser capaz de prever as jogadas do oponente, deve conhecer então o estado em que o tabuleiro ficaria após efetuar uma jogada antes de a efetuar, e apenas escolher entre as jogadas que mais forem inconvenientes ao oponente. Para tal, criam-se as funções *tabuleiro_se_jogasse_peca*, que formula estes tabuleiros potenciais, e redefinem-se os

grupos de soluções G *de findall* de modo a que cada jogada também agora inclua o tabuleiro resultante TFF dessa jogada.

```
cpu_todas_jogadas_climb_com_resultado(J, G, T) :-
    findall([P,CL1,CL2,TFF],cpu_uma_jogada_climb_com_resultado(J,P,CL1,CL2,T,TFF),G).
cpu_uma_jogada_climb_com_resultado(J,P,CL1,CL2,T,TFF) :-
    jogador(J),
    mao(J, M),
    length(M, L),
    (
        cpu_uma_jogada_climb_aux_linhas1(L,M,P,CL1,CL2,T);
        cpu_uma_jogada_climb_aux_linhas2(L,M,P,CL1,CL2,T);
        cpu_uma_jogada_climb_aux_colunas1(L,M,P,CL1,CL2,T);
        cpu_uma_jogada_climb_aux_colunas2(L,M,P,CL1,CL2,T)
    ),
    tabuleiro_se_jogasse_peca(P,CL1,CL2,TFF).

cpu_todas_jogadas_expand_com_resultado(J, G, T) :-
    jogador(J),
    findall([P,CL1,CL2,TFF],cpu_uma_jogada_expand_com_resultado(J,P,CL1,CL2,T,TFF),G).
cpu_uma_jogada_expand_com_resultado(J,P,CL1,CL2,T,TFF) :-
    jogador(J),
    mao(J, M),
    length(M, L),
    (
        cpu_uma_jogada_expand_aux_linhas1(L,M,P,CL1,CL2,T);
        cpu_uma_jogada_expand_aux_linhas2(L,M,P,CL1,CL2,T);
        cpu_uma_jogada_expand_aux_colunas1(L,M,P,CL1,CL2,T);
        cpu_uma_jogada_expand_aux_colunas2(L,M,P,CL1,CL2,T)
    ),
    tabuleiro_se_jogasse_peca(P,CL1,CL2,TFF).
```

O predicado *qualidade_a_melhor_jogada* obtém este grupo de soluções G e depois percorre-os de forma a estuda-los. O estudo consiste na valorização - para cada tabuleiro obtido, vai atribuir um valor Q, de qualidade. A qualidade é equivalente ao numero de jogadas que o jogador oponente é capaz de fazer no tabuleiro, por forma a quanto maior o número, pior é este valor de qualidade para o computador, e as jogadas ótimas são jogadas com Q de valor nulo, em que o oponente perde o turno.

Uma por uma, o predicado recursivo *qualidade_aux* então procura nas soluções qualidades com valor mais baixos, acabando prematuramente se encontrar uma qualidade igual a 0, devolvendo no final já apenas uma única jogada ao predicado principal, *qualidade_melhor_jogada.*

```
main_jogador_computador_dificil(J) :-
        cls,
        tabuleiro(T),
        mostra_tabuleiro(T),
        mostra_mao_jogador(J),
    %Encontra a lista de todas as jogadas possiveis, com priorida
        write('O computador '), write(J), write(' esta -cuidadosa
    %Escolher a Jogada que vai fazer
        qualidade_a_melhor_jogada(J, [V1|V2],[C1|L1],[C2|L2]),
        nl,write('Peca escolhida: ['), write(V1), write('|'), wri
        write('Posicao Escolhida: '), write(C1), write(':'), writ
        sleep(3),
    %Verificar se e valido
        !,
```

Esta função percorre não só as jogadas do computador, mas também múltiplas de outro, pelo que não é de surpreender que por vezes demore um bocado a resolver.

4. Interface com o Utilizador

Inicializar o jogo requer apenas correr o predicado ***dominup.***

A partir daqui encontramos o menu principal. *Navegação,* tal como a totalidade do *input* no jogo, é feito por input de valores numéricos. Indicação que é sugerida no próprio menu. O menu principal apresenta 3 escolhas: Jogar (entre duas pessoas), Jogar (envolvendo um) CPU e Sair. Apenas a segunda leva a mais menus, que por sua vez permite escolher jogar contra um computador fácil, contra um computador difícil, ou fazer o computador difícil jogar contra si mesmo.

```
++++++++++++++++++++++++++++++++++++++++
++                                    ++
++           D O M I N U P            ++
++                                    ++
++ ---------------------------------- ++
++            1 - Jogar               ++
++            2 - Jogar CPU           ++
++            3 - Sair                ++
++++++++++++++++++++++++++++++++++++++++
Escrever a opção seguida de um ponto
|: 2.
++++++++++++++++++++++++++++++++++++++++
++   Dificuldade?                     ++
++ ---------------------------------- ++
++  1 - Fácil                         ++
++  2 - Dificil                       ++
++  3 - CPU vs CPU                    ++
++++++++++++++++++++++++++++++++++++++++
|:
```

O jogo propriamente dito apresenta mais elementos. No princípio de qualquer jogada, logo no topo nos é mostrado o estado do tabuleiro. A parte de cima do tabuleiro tem numeradas os índices das colunas, e à esquerda os das linhas. Logo abaixo do tabuleiro apresenta-se a mão do jogador de quem é a vez, incluindo os seus próprios índices de seleção.

Estes números dos índices são importantes para as escolhas do jogador. O primeiro pedido feito ao utilizador no seu turno é a da escolha de uma peça a jogar.

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 7 | 7 | 7 | 0 | 2 | 7 | 6 | 1 | 7 | 3 | 3 | 5 | 2 | 1 | 6 | 6 | 5 | 7 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
Qual a peca que quer jogar?
|:
```

Com a escolha da peça feita, a mão do jogador é escondida exceto a peça, e é pedido por ordem, os índices das coordenadas do tabuleiro que a peça deve ocupar. Para ajudar o utilizador, é avisado também qual o valor que está a ser colocado com cada dois conjuntos de índices.

```
Peca escolhida: [5|7]. Valor da cabeca: 5
Quais as coordenadas da cabeca da peca que quer jogar?
Coluna?
|: 5.
Linha?
|: 4.
Peca escolhida: [5|7]. Valor da cauda: 7
Quais as coordenadas da cauda da peca que quer jogar?
Coluna?
|: 5.
Linha?
|: 3.
```

Se a seleção feita for correta, a jogada é aprovada e o muda-se o turno de jogo. Caso contrário, o turno recomeça do princípio.

```
A verificar se a jogada e valida...
A jogada e valida e foi efectuada
```

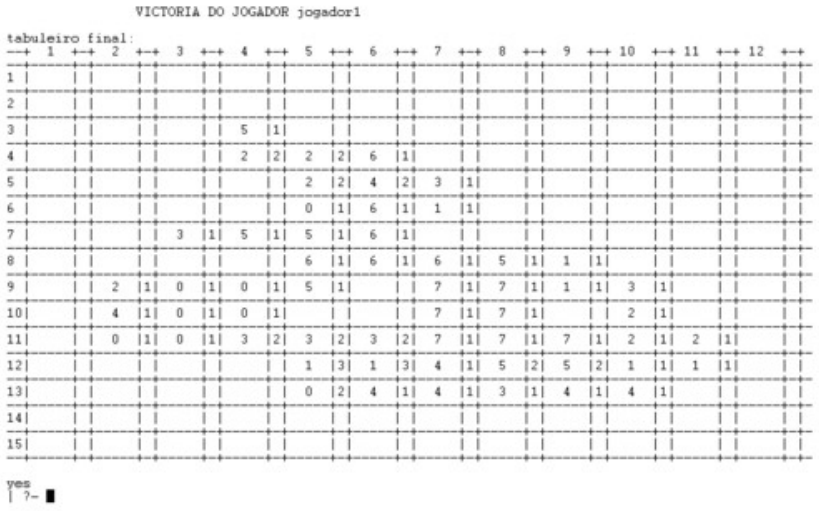
Nos turnos do computador, não é pedido *input* ao jogador. Ao invés disso, é apenas informado ao jogado quais as decisões que o computador está a fazer, com algum tempo dado para este poder assimilar o que está a ser feito.

```
O computador jogador2 esta a escolher uma peca para jogar...

Peca escolhida: [0|4].
Posicao Escolhida: 5:9, Cauda:6:9
```

Se o computador estiver a fazer procuras de dificuldade maior (quando tem menos de 9 peças) este avisa o jogador que ele está a fazer a sua decisão "cuidadosamente".

Quando um jogador jogar a sua última peça, o jogo termina, primeiro anunciando qual o jogador que a atingiu, e depois mostrando de novo o tabuleiro, agora no seu estado final.



5. Conclusões

Nas últimas semanas foi desenvolvido um jogo em PROLOG; funcional, dotado de uma interface de texto, permitindo jogar contra o computador com vários níveis de dificuldade ou até mesmo contra outro humano.

A realização deste projeto contribuiu irrefutavelmente para a consolidação dos assuntos abordados nas aulas teóricas e práticas da cadeira e, como tal, permitiram obter um conhecimento mais profundo de como trabalhar com a linguagem PROLOG e do seu poder enquanto linguagem.

Considera-se ainda importante notar a importância da ajuda dada pelos docentes ao grupo, aquando do confronto com as dúvidas que surgiram durante o desenvolvimento, que de outra forma não teriam sido resolvidas.

Anexo A – Código do Domin-Up

- ✓ ***auxiliar.pl:***

```
%%%%%%%%%%%%%%%%%%%%%%
%% Manipular Listas %%
%%%%%%%%%%%%%%%%%%%%%%
%
%    Funções Importantes:
%
%    :-list_element_at(E,L,N).
%    Devolve o N-enésimo elemento da lista L em E.
%
%    :-list_delete_one(X,L1,L2).
%    Remove o X-enésimo elemento da lista L1, resultando na
%    lista L2.
%
%    :-matrix_setCell(L, C, TI, E, TF).
%    Recebe uma matriz TI (com o propósito de receber tabuleiros), e
%    uma coordenada coluna C + linha L, e
%    devolve a matriz TF igual à inicial exceto que o
%    elemento na coordenada é trocado pelo elemento E.
%
%


list_element_at(X,[X|_],1).
list_element_at(X,[_|L],K) :-
     list_element_at(X,L,K1),
     K is K1 + 1.

list_delete_one(X,L1,L2) :-
     append(A1,[X|A2],L1),
     append(A1,A2,L2).

matrix_setCell(1, Col, [H|T], Piece, [H1|T]) :-
     matrix_setCellCol(Col,H,Piece,H1).
matrix_setCell(N,Col,[H|T],Piece,[H|T1]):-
     Prev is N-1,
     matrix_setCell(Prev, Col, T, Piece, T1).

matrix_setCellCol(1, [_|T], Piece, [Piece|T]).
matrix_setCellCol(N, [H|T], Piece, [H|T1]):-
     Prev is N-1,
     matrix_setCellCol(Prev, T, Piece, T1).


%%%%%%%%%%%%%%%%%%%%%%%%
%% Redimensionar o Tabuleiro        %%
%%%%%%%%%%%%%%%%%%%%%%%%
%
%    Funções Importantes:
%
%    :-tabuleiro_dimensiona.
%    Encontra o tabuleiro actualmente definido pelo jogo e,
```

```
%    com o uso às funções
%
%
%    *tabuleiro_primeira_linha_vazia.
%    *tabuleiro_insere_linha_inicio.
%    *tabuleiro_ultima_linha_vazia.
%    *tabuleiro_insere_linha_fim.
%    *tabuleiro_primeira_coluna_vazia
%    *tabuleiro_insere_coluna_inicio.
%    *tabuleiro_ultima_coluna_vazia.
%    *tabuleiro_insere_coluna_fim.
%
%    identifica se é necessário expandir o tamanho deste tabuleiro
%    para permitir os jogadores %  pousarem       peças na direção
%    ocupada. Um espaço vazio é o elemento [0|0].
%    Cada chamada acrescenta duas linhas ou duas colunas à direção
%    em necessidade.
%    Não faz nada noutros casos.
%    É necessário chamar a função duas vezes para garantir que peças
%    pousadas nos cantos do tabuleiro
%    são lidadas adequadamente.
%


tabuleiro_dimensiona :-
     tabuleiro(T),
     length(T, N),
     repeat,
             (\+tabuleiro_primeira_coluna_vazia(1, T, N) ->
(tabuleiro_insere_coluna_inicio,tabuleiro_insere_coluna_inicio);
             \+tabuleiro_ultima_coluna_vazia(1, T, N) ->
(tabuleiro_insere_coluna_fim,tabuleiro_insere_coluna_fim);
             \+tabuleiro_primeira_linha_vazia ->
(tabuleiro_insere_linha_inicio,tabuleiro_insere_linha_inicio);
             \+tabuleiro_ultima_linha_vazia ->
(tabuleiro_insere_linha_fim,tabuleiro_insere_linha_fim);
             true).

tabuleiro_primeira_linha_vazia :-
     tabuleiro(T),
     list_element_at(L,T,1),
     tabuleiro_linha_vazia(L).

tabuleiro_ultima_linha_vazia :-
     tabuleiro(T),
     length(T, N),
     list_element_at(L,T,N),
     tabuleiro_linha_vazia(L).

tabuleiro_linha_vazia([]).
tabuleiro_linha_vazia([H|T]) :-
     tabuleiro_primeiro_elemento_vazio([H|T]),
     tabuleiro_linha_vazia(T).

tabuleiro_primeira_coluna_vazia(N, _, Max) :-
     N > Max.
tabuleiro_primeira_coluna_vazia(N, T, Max) :-
     list_element_at(L, T, N),
     !,
     tabuleiro_primeiro_elemento_vazio(L),
     N1 is N+1,
```

```
    tabuleiro_primeira_coluna_vazia(N1, T, Max).

tabuleiro_ultima_coluna_vazia(N, _, Max) :-
    N > Max.
tabuleiro_ultima_coluna_vazia(N, T, Max) :-
    list_element_at(L, T, N),
    !,
    tabuleiro_ultimo_elemento_vazio(L),
    N1 is N+1,
    tabuleiro_ultima_coluna_vazia(N1, T, Max).

tabuleiro_primeiro_elemento_vazio([[_|A]|_]) :-
    A == 0.

tabuleiro_ultimo_elemento_vazio(L) :-
    reverse(L, NL),
    tabuleiro_primeiro_elemento_vazio(NL).

tabuleiro_insere_linha_fim :-
    tabuleiro(T),
    list_element_at(PL, T, 1),
    length(PL,C),
    acrescenta_vazio([], C, LF),
    !,
    append(T, [LF], TF),
    retract(tabuleiro(T)),
    assert(tabuleiro(TF)).

tabuleiro_insere_linha_inicio :-
    tabuleiro(T),
    list_element_at(PL, T, 1),
    length(PL,C),
    acrescenta_vazio([], C, LF),
    !,
    append([LF], T, TF),
    retract(tabuleiro(T)),
    assert(tabuleiro(TF)).

tabuleiro_insere_coluna_fim :-
    tabuleiro(T),
    length(T, Max),
    !,
    acrescenta_coluna_fim(T, 1, Max).

acrescenta_coluna_fim(_, N, Max) :-
    N > Max.
acrescenta_coluna_fim(T, N, Max) :-
    list_element_at(L, T, N),
    append(L, [[0|0]], LF),
    select(L, T, LF, TF),
    retract(tabuleiro(T)),
    assert(tabuleiro(TF)),
    N1 is N+1,
    acrescenta_coluna_fim(TF, N1, Max).

tabuleiro_insere_coluna_inicio :-
    tabuleiro(T),
    length(T, Max),
    !,
    acrescenta_coluna_inicio(T, 1, Max).
```

```
acrescenta_coluna_inicio(_, N, Max) :-
     N > Max.
acrescenta_coluna_inicio(T, N, Max) :-
     list_element_at(L, T, N),
     append([[0|0]], L, LF),
     select(L, T, LF, TF),
     retract(tabuleiro(T)),
     assert(tabuleiro(TF)),
     N1 is N+1,
     acrescenta_coluna_inicio(TF, N1, Max).

acrescenta_vazio(L, 1, LF) :-
     append([[0|0]], L, LF).

acrescenta_vazio(L, N, LF) :-
     append([[0|0]], L, LI),
     N1 is N-1,
     acrescenta_vazio(LI, N1, LF).

%%%%%%%%%%%%%%%%%%%%%%
%% Outras funções   %%
%%%%%%%%%%%%%%%%%%%%%%
%
%    Funções Importantes:
%
%    :-readInt(Texto, I, Min, Max).
%    Escreve um texto dado e pede ao jogadorum numero I que tem de
%    se encontrar entre os
%    valores Min e Max, senão pede um novo.
%
%    :-incrementador(P, N, Max).
%    Devolve em N, um a um, todos os valores entre P e Max, e depois
%    falha.
%


readInt(Texto, I, Min, Max) :-
     repeat,
     write(Texto), nl,
     read(I),
     I >= Min, I =< Max.

readInt_NoRepeat(Texto, I, Min, Max) :-
     write(Texto), nl,
     read(I),
     I >= Min, I =< Max.

incrementador(P, N, Max) :-
     P =< Max,
     N is P.
incrementador(P, N, Max) :-
     P =< Max,
     R is P+1,
     incrementador(R, N, Max).
```

- ✓ ***code.pl:***

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

```
:- use_module(library(system)).
:- use_module(displays).
:- use_module(auxiliar).
:- use_module(cpu).

%:-now(time), setrand(time).

%%%%%%%%%%%%%%%%%%%%%%
%%   Init           %%
%%%%%%%%%%%%%%%%%%%%%%
%
%    Algumas funções finais/teste requerem que alguns factos sejam
%    declarados como dinamicos antes de serem usados,
%    pelo que o fazemos logo aqui
%


jogador(jogador1).
jogador(jogador2).

:- dynamic tabuleiro/1.
tabuleiro([
[[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|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],[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|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],[0|0],[0|0]],

     [[0|0],[0|0],[0|0],[0|0],[0|0],[0|0],[0|0],[0|0],[0|0],[0|0]]
]).


:- dynamic mao/2.
mao(jogador1, []).
mao(jogador2, []).

:- dynamic jogador_escolhido/1.
jogador_escolhido(jogador1).

:- dynamic estado/1.
estado(T, J) :- tabuleiro(T), jogador_escolhido(J).

:- dynamic baralho/1.
baralho([ [0|0], [0|1], [0|2], [0|3], [0|4], [0|5], [0|6], [0|7],
[1|1], [1|2],
                    [1|3], [1|4], [1|5], [1|6], [1|7], [2|2],
[2|3], [2|4], [2|5], [2|6],
```

```
                    [2|7], [3|3], [3|4], [3|5], [3|6], [3|7],
[4|4], [4|5], [4|6], [4|7],
                    [5|5], [5|6], [5|7], [6|6], [6|7], [7|7] ]).


%%%%%%%%%%%%%%%%%%%%%%
%%   Main           %%
%%%%%%%%%%%%%%%%%%%%%%
%
%    Funções Importantes:
%
%    :- dominup.
%           A forma correcta que deve ser chamada para o jogador
%    iniciar o jogo.
%
%    :- jogar(Escolha).
%    Predicado que é chamado pelos menus e que indica qual o
%    modo de jogo escolhido à função main_loop,
%    após iniciar o jogo adequadamente. O jogo começa com:
%
%    1- Dar aleatóriamente as peças do baralho aos %
%    jogadores, uma a uma.
%    2- A procura da peça [7|7] nas mãos dos jogadores.
%    3- Esse jogador é forçado a jogar essa peça no centro do %
%    tabuleiro.
%    4- O jogador passa a vez e o outro joga normalmente daqui por d
%    diante.
%
%    O passo 1 é feito por baralho_dar_as_pecas.
%    Os passo 2 e 3 são feito por main_jogada_inicial.
%
%
%    :- main_loop(Escolha).
%    Quatro versões desta funções fazem a lógica do jogo de 7
%    formasdiferentes. Todos tem que iniciar a vez
%    do jogadores ou computadores quando adequado, e todos tem que
%    ser capazes de terminar o jogo quando se
%    atinge a condição de vitória (mão vazia).
%    Modos:
%           1 - Jogador vs Jogador
%           2 - Jogador vs CPU fácil
%           3 - Jogador vs CPU dificil
%           4 - CPU dificil vs CPU dificil
%           O computador difcil comporta-se como um fácil até ter
%    menos que 1 peças.
%
%
%    :- main_victoria(J).
%    Anuncia a vitória do jogador J e acaba o jogo depois de
%    mostrar o tabuleiro final.
%
%    :- main_jogador_humano(J).
%    :- main_jogador_computador_facil(J).
%    :- main_jogador_computador_dificil(J).
%
%    Predicados que fazem displays/reads e verificam as
%    jogadas dos jogadores ou computadores.
%    O computador facil joga aleatoriamente de entre todas
%    as jogadas válidas, o computador dificil
%    escreve a indicação de que está "-cuidadosamente-" a
```

```
%    fazer a melhor decisão entre todas as escolhas.
%    Se o oponente não poder jogar peças no próximo turno,
%    não é trocada a indicação de qual o jogador
%            que joga a seguir.
%
%


dominup :- menu_principal.

jogar(Escolha) :-
     Escolha >= 0, Escolha < 4,
     baralho_reiniciar,
     mao_reiniciar(jogador1),
     mao_reiniciar(jogador2),
     tabuleiro_reiniciar,
     baralho_dar_as_pecas,
     !,
     main_jogada_inicial,
     !,
     repeat,
     main_loop(Escolha).

main_jogada_inicial :-
     mao_quem_tem_peca([7|7], JI),
     mao_remover_peca(JI, [7|7]),
     tabuleiro_jogar_peca([7|7], [5|6], [6|6]),
     jogador_escolhido(J),
     (JI = J -> jogador_trocar_vez(J); true).

main_loop(0) :-
             jogador_escolhido(J),
             main_jogador_humano(J),
             !,
             (mao_vazia(J) ->  main_victoria(J); main_loop(0)).

main_loop(1) :-
     jogador_escolhido(J),
     (J = 'jogador1' -> main_jogador_humano(J);
main_jogador_computador_facil(J)),
     !,
     (mao_vazia(J) ->  main_victoria(J); main_loop(1)).

main_loop(2) :-
     jogador_escolhido(J),
     mao(J, MJ), length(MJ, ML),
     (J = 'jogador1' -> main_jogador_humano(J)
                                    ;
                                    ( ML < 10
                                           ->
main_jogador_computador_dificil(J);

     main_jogador_computador_facil(J)
                                    )
     ),
     !,
     (mao_vazia(J) ->  main_victoria(J); main_loop(2)).

main_loop(3) :-
     jogador_escolhido(J),
     mao(J, MJ), length(MJ, ML),
```

```
     ( ML < 10
            -> main_jogador_computador_dificil(J);
                   main_jogador_computador_facil(J)
     ),
     !,
     (mao_vazia(J) ->  main_victoria(J); main_loop(3)).


main_victoria(J) :-
     cls,
     nl,nl,nl,
     write('                VICTORIA DO JOGADOR '), write(J),nl,nl,
     write('tabuleiro final:'), nl,
     tabuleiro(T),
     mostra_tabuleiro(T).

main_jogador_humano(J) :-
     %Escolher a Peca
            cls,
            tabuleiro(T),
            mostra_tabuleiro(T),
            mostra_mao_jogador(J),
                   mao(J,M), length(M, ML),
            readInt('Qual a peca que quer jogar?', Input, 1, ML),
     %Escolher a Posição da cabeça
            !,
            cls,
            mostra_tabuleiro(T),
            mao_escolher_peca(J, Input, [V1|V2]),
            write('Peca escolhida: ['), write(V1), write('|'),
write(V2), write(']. Valor da cabeca: '), write(V1) , nl,
                   tabuleiro([TH|TR]), length([TH|TR], NL),
length(TH,NC),
            write('Quais as coordenadas da cabeca da peca que quer
jogar?'), nl,
            readInt('Coluna?', C1, 1, NC),
            readInt('Linha?', L1, 1, NL),
     %Escolher a Posição da cauda
            write('Peca escolhida: ['), write(V1), write('|'),
write(V2), write(']. Valor da cauda: '), write(V2) , nl,
                   tabuleiro([TH|TR]), length([TH|TR], NL),
length(TH,NC),
            write('Quais as coordenadas da cauda da peca que quer
jogar?'), nl,
            readInt('Coluna?', C2, 1, NC),
            readInt('Linha?', L2, 1, NL),
     %Verificar se e valido
            !,
            cls,
            write('A verificar se a jogada e valida...'), nl,
sleep(1),
            (
            tabuleiro_pode_jogar_peca_climb([V1|V2], [C1|L1],
[C2|L2]);
            tabuleiro_pode_jogar_peca_expand([V1|V2], [C1|L1],
[C2|L2])
            ),
            write('A jogada e valida e foi efectuada'), nl,
sleep(1),
     %alterar tabuleiro, tirar a peça ao jogador
            tabuleiro_jogar_peca([V1|V2], [C1|L1], [C2|L2]),
```

```
            mao_remover_peca(J, [V1|V2]),
    %Ver se o outro jogador pode jogar, trocar a vez se sim
            !,
            jogador(Joutro), Joutro \= J,
            (jogador_pode_jogar(Joutro) -> jogador_trocar_vez(J);
true),
    %expandir o tabuleiro se necessário
            tabuleiro_dimensiona, tabuleiro_dimensiona.

main_jogador_computador_facil(J) :-
            cls,
            tabuleiro(T),
            mostra_tabuleiro(T),
            mostra_mao_jogador(J),

%    Encontra a lista de todas as jogadas possiveis, com prioridade
%    a Climb
            write('O computador '), write(J), write(' esta a
escolher uma peca para jogar...'), nl,
            cpu_uma_ao_calhas(J,G,T),

%Escolher a Jogada que vai fazer
            length(G, JogadasL),
            random(0, JogadasL, N),
            list_element_at([[V1|V2],[C1|L1],[C2|L2]], G, N),
            nl,write('Peca escolhida: ['), write(V1), write('|'),
write(V2), write('].'), nl,
            write('Posicao Escolhida: '), write(C1), write(':'),
write(L1),write(', Cauda:'), write(C2), write(':'), write(L2), nl,
            sleep(3),

%Verificar se e valido
            !,
            (
             tabuleiro_pode_jogar_peca_climb([V1|V2], [C1|L1],
[C2|L2]);
             tabuleiro_pode_jogar_peca_expand([V1|V2], [C1|L1],
[C2|L2])
             ),
%alterar tabuleiro, tirar a peça ao jogador
            tabuleiro_jogar_peca([V1|V2], [C1|L1], [C2|L2]),
            !,
            (mao_remover_peca(J, [V1|V2]);mao_remover_peca(J,
[V2|V1])),
%Ver se o outro jogador pode jogar, trocar a vez se sim
            jogador(Joutro), Joutro \= J,
            (jogador_pode_jogar(Joutro) -> jogador_trocar_vez(J);
true),
%expandir o tabuleiro se necessário
            tabuleiro_dimensiona, tabuleiro_dimensiona.

main_jogador_computador_dificil(J) :-
            cls,
            tabuleiro(T),
            mostra_tabuleiro(T),
            mostra_mao_jogador(J),
%Encontra a lista de todas as jogadas possiveis, com prioridade a
Climb
            write('O computador '), write(J), write(' esta -
cuidadosamente- a escolher uma peca para jogar...'), nl, sleep(1),
```

```
%Escolher a Jogada que vai fazer
            qualidade_a_melhor_jogada(J, [V1|V2],[C1|L1],[C2|L2]),
            nl,write('Peca escolhida: ['), write(V1), write('|'),
write(V2), write('].'), nl,
            write('Posicao Escolhida: '), write(C1), write(':'),
write(L1),write(', Cauda:'), write(C2), write(':'), write(L2), nl,
            sleep(3),
     %Verificar se e valido
            !,
            (
             tabuleiro_pode_jogar_peca_climb([V1|V2], [C1|L1],
[C2|L2]);
             tabuleiro_pode_jogar_peca_expand([V1|V2], [C1|L1],
[C2|L2])
             ),
     %alterar tabuleiro, tirar a peça ao jogador
            tabuleiro_jogar_peca([V1|V2], [C1|L1], [C2|L2]),
            !,
            (mao_remover_peca(J, [V1|V2]);mao_remover_peca(J,
[V2|V1])),
     %Ver se o outro jogador pode jogar, trocar a vez se sim
            jogador(Joutro), Joutro \= J,
            (jogador_pode_jogar(Joutro) -> jogador_trocar_vez(J);
true),
     %expandir o tabuleiro se necessário
            tabuleiro_dimensiona, tabuleiro_dimensiona.

%%%%%%%%%%%%%%%%%%%%%%
%%   Jogador        %%
%%%%%%%%%%%%%%%%%%%%%%
%
%    :- jogador_trocar_vez(J)
%    Encontra o jogador que esteja marcado para fazer o
%    próximo turno, e troca essa marca para o oponente
%
%    :- jogador_pode_jogar(J).
%    :- jogador_jogadas_disponiveis_climb(J).
%    :- jogador_jogadas_disponiveis_expand(J).
%
%    Percorre as posições do tabuleiro, por 2 linhas e por 2
%    colunas, e vê se é possivel o jogador colocar
%    qualquer uma das peças que tem na mão.
%    Nas funções auxiliares, as variaveis L e M representam a
%    mão do jogador M e a quantidade de peças nela L.
%
%

jogador_trocar_vez(J) :- jogador(X),
                                          X \= J,

     retract(jogador_escolhido(J)),

     assert(jogador_escolhido(X)).

jogador_pode_jogar(J) :-
     (
     jogador_jogadas_disponiveis_climb(J);
     jogador_jogadas_disponiveis_expand(J)
     ).
```

```
jogador_jogadas_disponiveis_climb(J) :-
     jogador(J),
    mao(J, M),
     length(M, L),
     (
            jogador_jogadas_disponiveis_climb_aux_linhas(L,M);
            jogador_jogadas_disponiveis_climb_aux_colunas(L,M)
     ).

jogador_jogadas_disponiveis_climb_aux_linhas(L, M) :-
     tabuleiro([Z|X]),
     length(Z, NC),
     length([Z|X], NL),
     NLL is NL-1,
     !,
     incrementador(1, N, L),
     list_element_at([V1|V2], M, N),
     incrementador(1, L1, NLL),
     L2 is L1+1,
     incrementador(1, C1, NC),
     C2 is C1,
     (tabuleiro_pode_jogar_peca_climb([V1|V2], [C1|L1], [C2|L2]);
     tabuleiro_pode_jogar_peca_climb([V2|V1], [C1|L1], [C2|L2])).

jogador_jogadas_disponiveis_climb_aux_colunas(L, M) :-
     tabuleiro([Z|X]),
     length(Z, NC),
     length([Z|X], NL),
     NCC is NC-1,
     !,
     incrementador(1, N, L),
     list_element_at([V1|V2], M, N),
     incrementador(1, L1, NL),
     L2 is L1,
     incrementador(1, C1, NCC),
     C2 is C1+1,
     (tabuleiro_pode_jogar_peca_climb([V1|V2], [C1|L1], [C2|L2]);
     tabuleiro_pode_jogar_peca_climb([V2|V1], [C1|L1], [C2|L2])).



jogador_jogadas_disponiveis_expand(J) :-
     jogador(J),
    mao(J, M),
     length(M, L),
     (
            jogador_jogadas_disponiveis_expand_aux_linhas(L,M);
            jogador_jogadas_disponiveis_expand_aux_colunas(L,M)
     ).

jogador_jogadas_disponiveis_expand_aux_linhas(L, M) :-
     tabuleiro([Z|X]),
     length(Z, NC),
     length([Z|X], NL),
     NLL is NL-1,
     !,
     incrementador(1, N, L),
     list_element_at([V1|V2], M, N),
     incrementador(1, L1, NLL),
     L2 is L1+1,
     incrementador(1, C1, NC),
```

```
    C2 is C1,
    (tabuleiro_pode_jogar_peca_expand([V1|V2], [C1|L1], [C2|L2]);
    tabuleiro_pode_jogar_peca_expand([V2|V1], [C1|L1], [C2|L2])).

jogador_jogadas_disponiveis_expand_aux_colunas(L, M) :-
    tabuleiro([Z|X]),
    length(Z, NC),
    length([Z|X], NL),
    NCC is NC-1,
    !,
    incrementador(1, N, L),
    list_element_at([V1|V2], M, N),
    incrementador(1, L1, NL),
    L2 is L1,
    incrementador(1, C1, NCC),
    C2 is C1+1,
    (tabuleiro_pode_jogar_peca_expand([V1|V2], [C1|L1], [C2|L2]);
    tabuleiro_pode_jogar_peca_expand([V2|V1], [C1|L1], [C2|L2])).


%%%%%%%%%%%%%%%%%%%%%%
%%   Tabuleiro      %%
%%%%%%%%%%%%%%%%%%%%%%
%
%    :- tabuleiro_set(TI, [C|L], V, A, TF).
%    Recebe um tabuleiro TI, e muda os valores nas
%    coordenadas C,L pelos valores V,A resultando no tabuleiro TF.
%
%    :- tabuleiro_get(T,[C|L], V, A).
%    Encontra no tabuleiro T as coordenadas C,L e devolve os
%    valores [V|A].
%
%    :- tabuleiro_distancia_coordenadas([C1|L1], [C2|L2], D)
%    Dados dois conjuntos de coordenadas, CL1 e CL2, indica %     a
%    distancia a que estão uma da outra.
%    A distancia 1 significa que são adjacentes na
%    horizontal ou vertical.
%
%    :- tabuleiro_jogar_peca(P, CL1, CL2).
%    Joga a peça P no tabuleiro nas coordenadas CL1 (cabeça)
%    e CL2 (cauda). Assume que já foram feitas
%    verificações pelo que só ve se as coordenadas tem a
%    mesma altura.
%
%    :-tabuleiro_pode_jogar_peca_expand(P, CL1, CL2).
%    :-tabuleiro_pode_jogar_peca_climb(P, CL1, CL2).
%
%    As verificações à função anterior. Apenas indicam se
%    pode ser jogado ou não.
%
%    :- tabuleiro_reiniciar.
%           Esvazia o tabuleiro de jogo e retorna-o as suas
%    dimensões às originais.
%
%


tabuleiro_set(TI, [C|L], V, A, TF) :-
    matrix_setCell(L, C, TI, [V|A], TF).


tabuleiro_get(T,[C|L], V, A) :-
```

```
	C > 0,
	L > 0,
	list_element_at(Y, T, L),
	list_element_at([V|A], Y, C).


tabuleiro_distancia_coordenadas([C1|L1], [C2|L2], D) :-
	DX is C1-C2,
	X is abs(DX),
	DY is L1-L2,
	Y is abs(DY),
	D is X+Y.

tabuleiro_jogar_peca([V1|V2], [C1|L1], [C2|L2]) :-
	tabuleiro(TI),
	tabuleiro_get(TI,[C1|L1], _, A1),
	tabuleiro_get(TI,[C2|L2], _, A2),
	A1 == A2,
	A is A1+1,
	tabuleiro_set(TI, [C1|L1], V1, A, TF),
	retract(tabuleiro(TI)),
	assert(tabuleiro(TF)),
	tabuleiro_set(TF, [C2|L2], V2, A, TFF),
	retract(tabuleiro(TF)),
	assert(tabuleiro(TFF)).


tabuleiro_pode_jogar_peca_expand([V1|V2], [C1|L1], [C2|L2]) :-
	tabuleiro(T),
	%buscar os valores no tabuleiro
		tabuleiro_get(T, [C1|L1], _, TA1),
		tabuleiro_get(T, [C2|L2], _, TA2),
		!,
	%as coordenadas devem estar a tocar-se
		tabuleiro_distancia_coordenadas([C1|L1], [C2|L2], D),
		D == 1,
	%as alturas devem ser iguais
		TA1 == TA2,
	%uma das peças circundantes devem ter o mesmo valor
		A is TA1 + 1,
		(
			expand_aux(V1, [C1|L1], A, T);
			expand_aux(V2, [C2|L2], A, T)
		).

			expand_aux(V, [C|L], A, T) :-
				%à esquerda
					C2 is C-1,
					tabuleiro_get(T, [C2|L], V2,
A2),
					A == A2,
					V == V2.
			expand_aux(V, [C|L], A, T) :-
				%à direita
					C2 is C+1,
					tabuleiro_get(T, [C2|L], V2,
A2),
					A == A2,
					V == V2.
			expand_aux(V, [C|L], A, T) :-
				%abaixo
```

```
                                    L2 is L+1,
                                    tabuleiro_get(T, [C|L2], V2,
A2),
                                    A == A2,
                                    V == V2.
                    expand_aux(V, [C|L], A, T) :-
                            %acima
                                    L2 is L-1,
                                    tabuleiro_get(T, [C|L2], V2,
A2),
                                    A == A2,
                                    V == V2.


tabuleiro_pode_jogar_peca_climb([V1|V2], [C1|L1], [C2|L2]) :-
     tabuleiro(T),
     %buscar os valores no tabuleiro
             tabuleiro_get(T, [C1|L1], TV1, TA1),
             tabuleiro_get(T, [C2|L2], TV2, TA2),
             !,
     %as coordenadas devem estar a tocar-se
             tabuleiro_distancia_coordenadas([C1|L1], [C2|L2], D),
             D == 1,
     %as alturas devem ser iguais
             TA1 == TA2,
     %a altura não pode ser 0
             TA1 \= 0,
     %os valores no tabuleiro devem ser iguais ao da peça
             TV1 == V1,
             TV2 == V2.

tabuleiro_reiniciar :-
             tabuleiro(T),
             retract(tabuleiro(T)),
             assert(
                     tabuleiro([
[[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|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],[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|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],[0|0],[0|0]],

     [[0|0],[0|0],[0|0],[0|0],[0|0],[0|0],[0|0],[0|0],[0|0],[0|0]]
])
             ).
```

```
%%%%%%%%%%%%%%%%%%%%%%
%%   mao jogador    %%
%%%%%%%%%%%%%%%%%%%%%%
%
%    :- mao_acrescentar_peca(P, J).
%    Acrescenta a peça [P] à mão do jogador J.
%
%    :- mao_remover_peca(J, P).
%    Remove a peça P da mão do jogador J, se existir.
%
%
%    :- mao_escolher_peca(J, N, P).
%    Devolve em P a N-ésima peça da mão do jogador J.
%
%    :- mao_vazia(J).
%    Sucesso se o jogador não tiver peças na
%    mão.
%
%    :- mao_quem_tem_peca(P, J).
%    Procura a peça P nas mãos dos jogadores todos e devolve
%    em J o jogador que a tiver.
%
%    :- mao_reiniciar(J).
%    Esvazia a mão do jogador J.
%

mao_acrescentar_peca(P, J) :-      mao(J, V),

     append(P,V,N),

     retract(mao(J,V)),

     assert(mao(J, N)).


mao_remover_peca(J, P) :-
     jogador(J),
     mao(J,MV),
     list_delete_one(P, MV, MN),
     retract(mao(J,MV)),
     assert(mao(J,MN)).

mao_escolher_peca(J, X, P) :-
     jogador(J),
     mao(J,M),
     list_element_at(P, M, X).

mao_vazia(J) :-
     jogador(J),
     mao(J, M),
     length(M,0).


mao_quem_tem_peca(P, J) :-
     mao(J,M),
     member(P,M).

mao_reiniciar(J) :-
             jogador(J),
             mao(J, M),
```

```
            retract(mao(J, M)),
            assert(mao(J, [])).

%%%%%%%%%%%%%%%%%%%%%%
%%   Baralho        %%
%%%%%%%%%%%%%%%%%%%%%%
%
%    :- baralho_vazio.
%    Passa se o baralo estiver vazio.
%
%    :- baralho_get_peca_from(P).
%    Remove uma peça aleatória ao baralho e devolve-a em
%    P.
%
%    :-baralho_dar_as_pecas.
%    Corre o predicado baralho_get_peca_from repetidamente
%    de forma a dar todas as peças do baralho
%    aos jogadores, alternadamete. Pára quando o baralho
%    estiver vazio.
%
%    :- baralho_reiniciar.
%    Reinicia o baralho no seu estado original, com uma
%    cópia de cada peça.
%
%

baralho_vazio :-    baralho(B),
                                    mostra_linha(B),
                                    !,
                                    length(B,0).

baralho_get_peca_from(P) :- baralho(B),
                                                    length(B, L),
                                                    LS is L + 1,
                                                    random(0, LS, X),

     list_element_at(P, B, X).

baralho_reiniciar :-
                    baralho(B),
                    retract(baralho(B)),
                    assert(
                    baralho([
                            [0|0], [0|1], [0|2], [0|3], [0|4],
[0|5], [0|6], [0|7], [1|1], [1|2],
                            [1|3], [1|4], [1|5], [1|6], [1|7],
[2|2], [2|3], [2|4], [2|5], [2|6],
                            [2|7], [3|3], [3|4], [3|5], [3|6],
[3|7], [4|4], [4|5], [4|6], [4|7],
                            [5|5], [5|6], [5|7], [6|6], [6|7], [7|7]
])
                    ).


baralho_dar_as_pecas :-
            repeat,
            baralho(B),
            baralho_get_peca_from(P),
            list_delete_one(P,B,C),
            retract(baralho(B)),
            assert(baralho(C)),
```

```
            jogador_escolhido(J),
            mao_acrescentar_peca([P],J),
            jogador_trocar_vez(J),
            baralho(Y),
            length(Y, 0).
```

✓ ***cpu.pl:***

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%   Encontrar Jogadas Aleatorias   %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%    Funções Importantes:
%
%    :-cpu_uma_ao_calhas(J,G,T).
%    Recebe um jogador J e um tabuleiro T, e primeiro
%    devolve em G a lista de, ou:
%    *cpu_todas_jogadas_climb(J,G,T). - Todas
%    as jogadas do typo Climb que pode efectuar
%    e se não existirem;
%    *cpu_todas_jogadas_expand(J,G,T). - Todas
%    as jogadas do typo Expand que pode efectuar
%    a funções encontram jogadas possiveis a percorrer o
%    tabuleiro com as peças posicionadas na vertical
%    ou na horizontal, indicado por aux_linhas# ou % %
%    aux_colunas#. O valor 1 ou 2 do # indica se a peça está
%    a ser testada num sentido ou no outro.
%    Nas posições, as peças verificam se são jogada válida
%    por uso a
%    *cpu_pode_jogar_peca_expand(P, CL1, CL2,
%    T)
%    *cpu_pode_jogar_peca_climb(P, CL1, CL2,
%    T)
%    Onde P é a peça a encaixar, CL1 a coordenada da cabeça %     da
%    peça e CL2 as coordenadas da cauda da peça.
%    Estas três variáveis tem todas dois valores.
%
%
%
%
%

cpu_uma_ao_calhas(J,G,T) :-
     cpu_todas_jogadas_climb(J,G,T),
     G \= [].
cpu_uma_ao_calhas(J,G,T) :-
     cpu_todas_jogadas_expand(J,G,T),
     G \= [].



cpu_todas_jogadas_climb(J, G, T) :-
     jogador(J),
     findall([P,CL1,CL2],cpu_uma_jogada_climb(J,P,CL1,CL2,T),G).

cpu_uma_jogada_climb(J,P,CL1,CL2,T) :-
     jogador(J),
    mao(J, M),
     length(M, L),
     (
            cpu_uma_jogada_climb_aux_linhas1(L,M,P,CL1,CL2,T);
```

```
            cpu_uma_jogada_climb_aux_linhas2(L,M,P,CL1,CL2,T);
            cpu_uma_jogada_climb_aux_colunas1(L,M,P,CL1,CL2,T);
            cpu_uma_jogada_climb_aux_colunas2(L,M,P,CL1,CL2,T)
    ).

cpu_uma_jogada_climb_aux_linhas1(L,M,[V1|V2],[C1|L1],[C2|L2],[Z|X])
:-
    length(Z, NC),
    length([Z|X], NL),
    NLL is NL-1,
    !,
    incrementador(1, N, L),
    list_element_at([V1|V2], M, N),
    incrementador(1, L1, NLL),
    L2 is L1+1,
    incrementador(1, C1, NC),
    C2 is C1,
    cpu_pode_jogar_peca_climb([V1|V2], [C1|L1], [C2|L2],[Z|X]).
cpu_uma_jogada_climb_aux_linhas2(L,M,[V2|V1],[C1|L1],[C2|L2],[Z|X])
:-
    length(Z, NC),
    length([Z|X], NL),
    NLL is NL-1,
    !,
    incrementador(1, N, L),
    list_element_at([V1|V2], M, N),
    incrementador(1, L1, NLL),
    L2 is L1+1,
    incrementador(1, C1, NC),
    C2 is C1,
    cpu_pode_jogar_peca_climb([V2|V1], [C1|L1], [C2|L2],[Z|X]).
cpu_uma_jogada_climb_aux_colunas1(L,M,[V1|V2],[C1|L1],[C2|L2],[Z|X]
) :-
    length(Z, NC),
    length([Z|X], NL),
    NCC is NC-1,
    !,
    incrementador(1, N, L),
    list_element_at([V1|V2], M, N),
    incrementador(1, L1, NL),
    L2 is L1,
    incrementador(1, C1, NCC),
    C2 is C1+1,
    cpu_pode_jogar_peca_climb([V1|V2], [C1|L1], [C2|L2],[Z|X]).
cpu_uma_jogada_climb_aux_colunas2(L,M,[V2|V1],[C1|L1],[C2|L2],[Z|X]
) :-
    length(Z, NC),
    length([Z|X], NL),
    NCC is NC-1,
    !,
    incrementador(1, N, L),
    list_element_at([V1|V2], M, N),
    incrementador(1, L1, NL),
    L2 is L1,
    incrementador(1, C1, NCC),
    C2 is C1+1,
    cpu_pode_jogar_peca_climb([V2|V1], [C1|L1], [C2|L2],[Z|X]).


    %%%%
    %testar:
```

```
    %%%%
            %Interromper dominup.
            %mao_reiniciar(jogador1),mao_acrescentar_peca([[7|7]],
jogador1),mostra_mao_jogador(jogador1).
            %cpu_todas_jogadas_climb(jogador1, G), length(G,L).
            %Length: 2




cpu_todas_jogadas_expand(J, G, T) :-
    jogador(J),
    findall([P,CL1,CL2],cpu_uma_jogada_expand(J,P,CL1,CL2,T),G).


cpu_uma_jogada_expand(J,P, CL1, CL2,T) :-
    jogador(J),
   mao(J, M),
    length(M, L),
    (
            cpu_uma_jogada_expand_aux_linhas1(L,M,P,CL1,CL2,T);
            cpu_uma_jogada_expand_aux_linhas2(L,M,P,CL1,CL2,T);
            cpu_uma_jogada_expand_aux_colunas1(L,M,P,CL1,CL2,T);
            cpu_uma_jogada_expand_aux_colunas2(L,M,P,CL1,CL2,T)
    ).

cpu_uma_jogada_expand_aux_linhas1(L,M,[V1|V2],[C1|L1],[C2|L2],[Z|X]
) :-
    length(Z, NC),
    length([Z|X], NL),
    NLL is NL-1,
    !,
    incrementador(1, N, L),
    list_element_at([V1|V2], M, N),
    incrementador(1, L1, NLL),
    L2 is L1+1,
    incrementador(1, C1, NC),
    C2 is C1,
    cpu_pode_jogar_peca_expand([V1|V2], [C1|L1], [C2|L2],[Z|X]).
cpu_uma_jogada_expand_aux_linhas2(L,M,[V2|V1],[C1|L1],[C2|L2],[Z|X]
) :-
    length(Z, NC),
    length([Z|X], NL),
    NLL is NL-1,
    !,
    incrementador(1, N, L),
    list_element_at([V1|V2], M, N),
    incrementador(1, L1, NLL),
    L2 is L1+1,
    incrementador(1, C1, NC),
    C2 is C1,
    cpu_pode_jogar_peca_expand([V2|V1], [C1|L1], [C2|L2],[Z|X]).
cpu_uma_jogada_expand_aux_colunas1(L,M,[V1|V2],[C1|L1],[C2|L2],[Z|X
]) :-
    length(Z, NC),
    length([Z|X], NL),
    NCC is NC-1,
    !,
    incrementador(1, N, L),
```

```
    list_element_at([V1|V2], M, N),
    incrementador(1, L1, NL),
    L2 is L1,
    incrementador(1, C1, NCC),
    C2 is C1+1,
    cpu_pode_jogar_peca_expand([V1|V2], [C1|L1], [C2|L2],[Z|X]).
cpu_uma_jogada_expand_aux_colunas2(L,M,[V2|V1],[C1|L1],[C2|L2],[Z|X
]) :-
    length(Z, NC),
    length([Z|X], NL),
    NCC is NC-1,
    !,
    incrementador(1, N, L),
    list_element_at([V1|V2], M, N),
    incrementador(1, L1, NL),
    L2 is L1,
    incrementador(1, C1, NCC),
    C2 is C1+1,
    cpu_pode_jogar_peca_expand([V2|V1], [C1|L1], [C2|L2],[Z|X]).


    %%%%
    %testar:
    %%%%
            %Interromper dominup.
            %mao_reiniciar(jogador1),mao_acrescentar_peca([[1|7]],
jogador1),mostra_mao_jogador(jogador1).
            %cpu_todas_jogadas_expand(jogador1, G), length(G,L).
            %Length: 18



cpu_pode_jogar_peca_expand([V1|V2], [C1|L1], [C2|L2],T) :-
    %buscar os valores no tabuleiro
            tabuleiro_get(T, [C1|L1], _, TA1),
            tabuleiro_get(T, [C2|L2], _, TA2),
            !,
    %as coordenadas devem estar a tocar-se
            tabuleiro_distancia_coordenadas([C1|L1], [C2|L2], D),
            D == 1,
    %as alturas devem ser iguais
            TA1 == TA2,
    %uma das peças circundantes devem ter o mesmo valor
            A is TA1 + 1,
            (
                    cpu_expand_aux(V1, [C1|L1], A, T);
                    cpu_expand_aux(V2, [C2|L2], A, T)
            ).

                    cpu_expand_aux(V, [C|L], A, T) :-
                            %à esquerda
                                    C2 is C-1,
                                    tabuleiro_get(T, [C2|L], V2,
A2),
                                    A == A2,
                                    V == V2.
                    cpu_expand_aux(V, [C|L], A, T) :-
                            %à direita
                                    C2 is C+1,
```

```
                                        tabuleiro_get(T, [C2|L], V2,
A2),
                                        A == A2,
                                        V == V2.
                        cpu_expand_aux(V, [C|L], A, T) :-
                                %abaixo
                                        L2 is L+1,
                                        tabuleiro_get(T, [C|L2], V2,
A2),
                                        A == A2,
                                        V == V2.
                        cpu_expand_aux(V, [C|L], A, T) :-
                                %acima
                                        L2 is L-1,
                                        tabuleiro_get(T, [C|L2], V2,
A2),
                                        A == A2,
                                        V == V2.



cpu_pode_jogar_peca_climb([V1|V2], [C1|L1], [C2|L2],T) :-
     %buscar os valores no tabuleiro
             tabuleiro_get(T, [C1|L1], TV1, TA1),
             tabuleiro_get(T, [C2|L2], TV2, TA2),
             !,
     %as coordenadas devem estar a tocar-se
             tabuleiro_distancia_coordenadas([C1|L1], [C2|L2], D),
             D == 1,
     %as alturas devem ser iguais
             TA1 == TA2,
     %a altura não pode ser 0
             TA1 \= 0,
     %os valores no tabuleiro devem ser iguais ao da peça
             TV1 == V1,
             TV2 == V2.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%   Ver a qualidade        %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%    Funções Importantes:
%
%    :-qualidade_a_melhor_jogada(J, P,CL1,CL2).
%             Recebe um jogador J e, assumindo o tabuleiro de jogo T,
%    procura qual é a melhor de todas jogadas,
%             ou seja, qual a peça P e onde - CL1, CL2 - a deve jogar
%    para efectuar a melhor jogada.
%             Utiliza uma redefinição das funções usadas por
%    cpu_uma_ao_calhas:
%    *        cpu_uma_jogada_climb_com_resultado
%    *        cpu_uma_jogada_expand_com_resultado
%             Com comportamento semelhante às versões acima, mas para
%    alem das jogadas, tambem geram o tabuleiro
%             resultante dessas jogadas:
%                     *       tabuleiro_se_jogasse_peca( P,CL1,CL2,
%    TFF) - TFF é igual ao tabuleiro do jogo se a peça P
%                                                                    %
%                                     fosse colocada em CL1 CL2.
%             O que se faz com esses tabuleiros resultantes vai ser
```

```
%    acontecer na função
%                    *       qualidade_aux(N, Q,
%    Pprev,CL1prev,CL2prev,  P,CL1,CL2, G, J) - A partir da lista de
%    jogadas G
%
%    que o jogador J pode efectuar, e por uso de recursividade
%    encontra a jogada
%
%    P,CL1,CL2 que obtem a qualidade de jogada Q mais baixa
%    possivel, ou seja
%
%    aquela cujo tabuleiro oferece o menor numero de jogadas ao
%    jogador oponente.
%
%    Se uma jogada que ofereça 0 jogadas
%    de
%    resposta é encontrada, essa é retornada de imediato.
%

qualidade_a_melhor_jogada(J, P,CL1,CL2) :-
     tabuleiro(T),
     jogador(J),
     cpu_todas_jogadas_climb_com_resultado(J,G,T),
     G \= [],
     %write('CPU Climb'), nl,
     jogador(Joutro), Joutro \= J,
     qualidade_aux(1, 10000, 0,0,0, P,CL1,CL2, G, Joutro).
qualidade_a_melhor_jogada(J, P,CL1,CL2) :-
     tabuleiro(T),
     jogador(J),
     cpu_todas_jogadas_expand_com_resultado(J,G,T),
     G \= [],
     %write('CPU Expand'), nl,
     jogador(Joutro), Joutro \= J,
     qualidade_aux(1, 30000, 0,0,0, P,CL1,CL2, G, Joutro).


qualidade_aux(_, 0,  Pprev,CL1prev,CL2prev, P,CL1,CL2, _, _) :-
     P = Pprev,
     CL1 = CL1prev,
     CL2 = CL2prev.

qualidade_aux(N, _,  Pprev,CL1prev,CL2prev, P,CL1,CL2, G, _) :-
     length(G, L),
     N > L,
     P = Pprev,
     CL1 = CL1prev,
     CL2 = CL2prev.

qualidade_aux(N, Q,  Pprev,CL1prev,CL2prev,  P,CL1,CL2, G, J) :-
     %write(Pprev),write(','),write(CL1prev),write(','),write(CL2pre
v),nl,
     length(G, L),
     N =< L,
     list_element_at(Valores,G,N),
             list_element_at(Pcurr,Valores,1),
             list_element_at(CL1curr,Valores,2),
             list_element_at(CL2curr,Valores,3),
             list_element_at(TFF,Valores,4),
     cpu_todas_jogadas_climb(J,JGC,TFF), length(JGC, L1C),
     cpu_todas_jogadas_expand(J,JGE,TFF), length(JGE, L2E),
```

```
    QRes is L1C+L2E,
    Next is N+1,
    (
    QRes < Q
           ->     qualidade_aux(Next, QRes,
Pcurr,CL1curr,CL2curr, Pnext,CL1next,CL2next,    G, J)
           ;      qualidade_aux(Next, Q,   Pprev,CL1prev,CL2prev,
Pnext,CL1next,CL2next,     G, J)
    ),
    P = Pnext,
    CL1 = CL1next,
    CL2 = CL2next.

cpu_todas_jogadas_climb_com_resultado(J, G, T) :-
    findall([P,CL1,CL2,TFF],cpu_uma_jogada_climb_com_resultado(J,P,
CL1,CL2,T,TFF),G).
cpu_uma_jogada_climb_com_resultado(J,P,CL1,CL2,T,TFF) :-
    jogador(J),
   mao(J, M),
    length(M, L),
    (
           cpu_uma_jogada_climb_aux_linhas1(L,M,P,CL1,CL2,T);
           cpu_uma_jogada_climb_aux_linhas2(L,M,P,CL1,CL2,T);
           cpu_uma_jogada_climb_aux_colunas1(L,M,P,CL1,CL2,T);
           cpu_uma_jogada_climb_aux_colunas2(L,M,P,CL1,CL2,T)
    ),
    tabuleiro_se_jogasse_peca(P,CL1,CL2,TFF).

cpu_todas_jogadas_expand_com_resultado(J, G, T) :-
    jogador(J),
    findall([P,CL1,CL2,TFF],cpu_uma_jogada_expand_com_resultado(J,P
,CL1,CL2,T,TFF),G).
cpu_uma_jogada_expand_com_resultado(J,P,CL1,CL2,T,TFF) :-
    jogador(J),
   mao(J, M),
    length(M, L),
    (
           cpu_uma_jogada_expand_aux_linhas1(L,M,P,CL1,CL2,T);
           cpu_uma_jogada_expand_aux_linhas2(L,M,P,CL1,CL2,T);
           cpu_uma_jogada_expand_aux_colunas1(L,M,P,CL1,CL2,T);
           cpu_uma_jogada_expand_aux_colunas2(L,M,P,CL1,CL2,T)
    ),
    tabuleiro_se_jogasse_peca(P,CL1,CL2,TFF).

tabuleiro_se_jogasse_peca([V1|V2], [C1|L1], [C2|L2], TFF) :-
    tabuleiro(TI),
    tabuleiro_get(TI,[C1|L1], _, A1),
    tabuleiro_get(TI,[C2|L2], _, A2),
    A1 == A2,
    A is A1+1,
    tabuleiro_set(TI, [C1|L1], V1, A, TF),
    tabuleiro_set(TF, [C2|L2], V2, A, TFF).


    %%%%
    %testar:
    %%%%
    %teste :-
%  tabuleiro_jogar_peca([7|7],[7|5],[7|6]), tabuleiro(T),
%  mostra_tabuleiro(T),
%   mao_reiniciar(jogador1),mao_acrescentar_peca([[6|6]],
```

```
%    jogador1),
%    mao_reiniciar(jogador2),mao_acrescentar_peca([[7|7]],
%    jogador2),mao_acrescentar_peca([[6|7]], jogador2),
%    qualidade_a_melhor_jogada(jogador2, P,CL1,CL2),
%    write(P),write(','),write(CL1),write(','),write(CL2),nl.
```

- ✓ ***displays.pl:***

```
%%%%%%%%%%%%%%%%%%%%%%
%%   Limpar         %%
%%%%%%%%%%%%%%%%%%%%%%
%    Funções Importantes:
%
%    :-cls.
%    Limpa o terminal e empurra para o topo novos elementos
%    que forem mostrados.
%
%

cls :- write('\e[2J').


%%%%%%%%%%%%%%%%%%%%%%
%%   Menu           %%
%%%%%%%%%%%%%%%%%%%%%%
%    Funções Importantes:
%
%    :-menu_principal.
%    Mostra o menu principal do jogo e espera pela escolha
%    do jogador.
%    Deve ser chamado directamente pela função dominup/1 %
%    (code.pl).
%
%    :-main_menu_opcoes(N).
%    Cada um dos valores N representa uma das
%    diferentes opções do menu principal.
%
%    :-menu_dificuldade.
%    Mostra o submenu em que se escolhe como se
%    comporta o computador.
%


menu_principal :-
     cls,
     write('+++++++++++++++++++++++++++++++++++++++'), nl,
     write('++                                   ++'), nl,
     write('++           D O M I N U P           ++'), nl,
     write('++                                   ++'), nl,
     write('++ ---------------------------------- ++'), nl,
     write('++            1 - Jogar              ++'), nl,
     write('++            2 - Jogar CPU          ++'), nl,
     write('++            3 - Sair               ++'), nl,
     write('+++++++++++++++++++++++++++++++++++++++'), nl,
     readInt('Escrever a opção seguida de um ponto', I, 1, 3),
     !,
     main_menu_opcoes(I).

main_menu_opcoes(1) :-     jogar(0).
main_menu_opcoes(2) :-     menu_dificuldade.
main_menu_opcoes(3).
```

```
menu_dificuldade :-
     write('++++++++++++++++++++++++++++++++++++++++'), nl,
     write('++   Dificuldade?                     ++'), nl,
     write('++ ---------------------------------- ++'), nl,
     write('++  1 - Fácil                         ++'), nl,
     write('++  2 - Dificil                       ++'), nl,
     write('++  3 - CPU vs CPU                    ++'), nl,
     write('++++++++++++++++++++++++++++++++++++++++'), nl,
     read(I), I < 4, I > 0,
     !,
     jogar(I).

%%%%%%%%%%%%%%%%%%%%%%
%%   Tabuleiro      %%
%%%%%%%%%%%%%%%%%%%%%%
%    Funções Importantes:
%
%    :-mostra_tabuleiro(T).
%    Recebe um tabuleiro e faz uma representação deste.
%
%    :-mostra_N_col(N,L).
%    Monta o cabeçalho do tabuleiro visivel. L é o
%    numero de colunas, N deve começar a 0.
%
%    :-mostra(I, [L|R], N)
%    Recurssivamente monta o tabuleiro com uso das
%    outras funções excepto a do cabeçalho.
%
%    :-mostra_separador(N,L).
%    Monta divisões horizontais no tabuleiro.
%

mostra_peca([_|0]) :- write('     | |').
mostra_peca([V|H]) :- write('  '),
                                        write(V),
                                        write('  |'),
                                        write(H),
                                        write('|').

mostra_linha([]).
mostra_linha([P|[]]) :-
                                        P \= [],
                                        mostra_peca(P).
mostra_linha([P|R]) :-
                                    P \= [],
                                    mostra_peca(P),
                                    mostra_linha(R).

mostra(_, [], LL) :- mostra_separador(0,LL).
mostra(N, [L|R], LL) :-
                                    NN is N+1,
                                    N >= 10,
                                    mostra_separador(0,LL),
                                    write(N), write('|'),
                                    mostra_linha(L),
                                    nl,
                                    mostra(NN, R, LL).
mostra(N, [L|R], LL) :- NN is N+1,
                                    N < 10,
```

```
                                        mostra_separador(0,LL),
                                        write(N), write(' |'),
                                        mostra_linha(L),
                                        nl,
                                        mostra(NN, R, LL).

mostra_separador(N,L):-
     N > L,
     nl,
     true.
mostra_separador(N,L):-
     N = 0,
     N =< L,
     NN is N + 1,
     write('--+-'),
     mostra_separador(NN, L).
mostra_separador(N,L):-
     N =< L,
     NN is N + 1,
     write('----+-+-'),
     mostra_separador(NN, L).

mostra_N_col(N,L) :-
     N > L,
     true.
mostra_N_col(N,L) :-
     N = 0,
     N =< L,
     NN is N + 1,
     write('--+'),
     mostra_N_col(NN, L).
mostra_N_col(N,L) :-
     N < 10,
     N =< L,
     NN is N + 1,
     write('  '),
     write(N),
     write('  +-+'),
     mostra_N_col(NN, L).
mostra_N_col(N,L) :-
     N >= 10,
     N =< L,
     NN is N + 1,
     write(' '),
     write(N),
     write('  +-+'),
     mostra_N_col(NN, L).


mostra_tabuleiro([L|R]) :-
            %tabuleiro([L|R]),
            length(L,N),
            mostra_N_col(0, N), nl,
            mostra(1, [L|R], N),
            nl.

%%%%%%Exemplo - comando |?- exemplo_mostra_tab. em Prolog
exemplo_mostra_tab :- tabuleiro(L), write('Tabuleiro:'), nl,
mostra_tabuleiro(L).
```

```
%%%%%%%%%%%%%%%%%%%%%%
%%   Mãos            %%
%%%%%%%%%%%%%%%%%%%%%%
%    Funções Importantes:
%
%    :-mostra_mao_jogador(J).
%    Recebe um Jogador e faz uma representação das peças que
%    este pode jogar.
%
%    :-mostra_mao_cabecalho.
%    Responsável por mostrar o cabeçaho com o numero
%    de ordem das peças.
%
%    :-mostra_mao_linha_V#.
%    Responsável por mostrar os valores da cabeça ou
%    da cauda de cada peça.
%
%    :-mostra_mao_separador.
%    Responsável por mostrar os separadores
%    horizontais.
%



mostra_mao_linha_V1([]).
mostra_mao_linha_V1([ [V1|_] |R]) :-
     write(' '), write(V1), write(' |'),
     mostra_mao_linha_V1(R).

mostra_mao_linha_V2([]).
mostra_mao_linha_V2([ [_|V2] |R]) :-
     write(' '), write(V2), write(' |'),
     mostra_mao_linha_V2(R).

mostra_mao_cabecalho(L):-
     write('+'),
     !,
     incrementador(1, N, L),
     mostra_mao_cabecalho_aux(N),
     N>=L.
mostra_mao_cabecalho_aux(N) :- N < 10, write(' '),write(N),write('
+').
mostra_mao_cabecalho_aux(N) :- N >= 10, write('
'),write(N),write('+').

mostra_mao_separador(L):-
     write('+'),
     !,
     incrementador(1, N, L),
     write('---+'),
     N>=L.


mostra_mao_jogador(J) :-
     jogador(J),
     mao(J, M),
     write('Mao do Jogador '),
     write(J),
```

```
write(':'),
nl,
length(M, L),
mostra_mao_cabecalho(L),nl,
mostra_mao_separador(L),nl,
write('|'), mostra_mao_linha_V1(M), nl,
mostra_mao_separador(L),nl,
write('|'), mostra_mao_linha_V2(M), nl,
mostra_mao_separador(L),nl.
```