Dominup em Prolog

Relatório Final

Mestrado Integrado em Engenharia Informática e Computação

Programação em Lógica

Dominup4:
Ângela Filipa Pereira Cardoso - up200204375
Nuno Miguel Rainho Valente - up200204376

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

8 de Novembro de 2015

Resumo

Este projeto teve como objetivo a recriação de um jogo, o Dominup. Este jogo de tabuleiro é uma variação do famoso jogo Dominó, ao que tudo indica surgido na China há mais de dois mil anos.

Para implementação do jogo Dominup, utilizamos SICStus Prolog, tendo o cuidado de apresentar a interação com o utilizador na forma mais prática e intuitiva possível.

O Dominup será, posteriormente, alvo de maior detalhe neste relatório no que diz respeito a toda a sua jogabilidade, contudo, globalmente, permitimos que o jogo fosse jogado de três formas distintas:

- Humano contra Humano;
- Humano contra Computador;
- Computador contra Computador.

Nos tipos de jogo que envolvem o computador foi necessário recorrer à implementação de inteligências artificiais, duas neste caso - uma mais simples e uma outra mais complexa.

A solução que obtemos cumpre todos os requisitos do projeto. Conseguimos implementar o programa de forma eficiente, recorrendo sobretudo à base de dados do Prolog e fazendo uso muito restrito de listas. A inteligência artificial mais complexa é a única função cujo resultado não é imediato, no entanto demora apenas alguns segundos.

Conteúdo

1 Introdução

Como forma de aprender a programar em Prolog, foi-nos proposto, no âmbito da unidade curricular de Programação em Lógica, a implementação de um jogo de tabuleiro. No nosso caso, o jogo escolhido foi o Dominup, uma variação do jogo Dominó, que se joga com 36 peças duplas numeradas de 0 a 7.

A implementação permite ao utilizador escolher entre 3 tipos de jogo:

- Humano contra Humano;
- Humano contra Computador;
- Computador contra Computador.

Para cada jogador Computador, o utilizador pode escolher entre os níveis de inteligência fácil e difícil. Além disso, é possível escolher entre começar um novo jogo ou carregar um jogo previamente salvado.

Para implementação do jogo, recorremos sobretudo à base de dados do Prolog, procurando assim uma implementação tão eficiente quanto possível. Isso também permitiu que o processo de salvar e carregar um jogo seja bastante simples.

Fora a inteligência artificial mais complexa, a parte da implementação mais trabalhosa foi a visualização do jogo. Isto porque se trata de um jogo a 3 dimensões, cuja representação em modo de texto gera algumas dificuldades. Tentamos que o resultado final fosse intuitivo, de forma a que não se tornasse um detrimento no jogo. O mesmo princípio se aplicou também na interação com o jogador.

O presente relatório está organizado por capítulos, sendo o primeiro correspondente a esta introdução. No segundo capítulo fazemos uma apresentação detalhada do jogo Dominup com imagens ilustrativas que facilitam o entendimento. No Capítulo 3 pretendemos dar a conhecer a lógica de jogo e a sua forma de implementação em Prolog. No quarto capítulo, indicamos como funciona o módulo de interface com o utilizador em modo de texto. No Capítulo 5 fazemos alguns reparos finais nas conclusões. Terminamos com a bibliografia que utilizamos e em anexo apresentamos todo o código fonte do projeto.

2 O Jogo Dominup

Dominup é uma variação do jogo Dominó para 2 a 4 jogadores, em que, tal como o nome sugere, é possível colocar peças em cima de outras.

No típico Dominó existem 28 peças duplas numeradas de 0 a 6, à semelhança das faces de um dado. Já no Dominup há 36 peças duplas numeradas de 0 a 7, usando códigos binários: o ponto no centro representa 1, o circulo pequeno representa 2 e o circulo grande representa 4, como se pode ver na Figura 1. Este desenho das peças, juntamente com as regras do Dominup e de dois outros jogos, foram criadas por Néstor Romeral Andrés em 2014, sendo o conjunto publicado por nestorgames[1].

Figura 1: Exemplo da peça $3 \cdot 6$.

Existem dois tipos de colocação de peças no Dominup:

- subir - a peça é colocada em cima de duas peças adjacentes que estejam ao mesmo nível, de forma a que os números da peça colocada sejam iguais aos que ficam por baixo (um em cada peça de suporte), tal como mostra a Figura 2.
- expandir - a peça é colocada na superfície de jogo, de forma a que fique adjacente e ortogonal a pelo menos uma peça já colocada, como, por exemplo, as duas peças já colocadas na Figura 2.

Tal como no Dominó, as regras são relativamente simples. Começa-se por distribuir as peças aleatoriamente e de forma equilibrada pelos jogadores, mantendo a face voltada para baixo.

[1] http://www.nestorgames.com

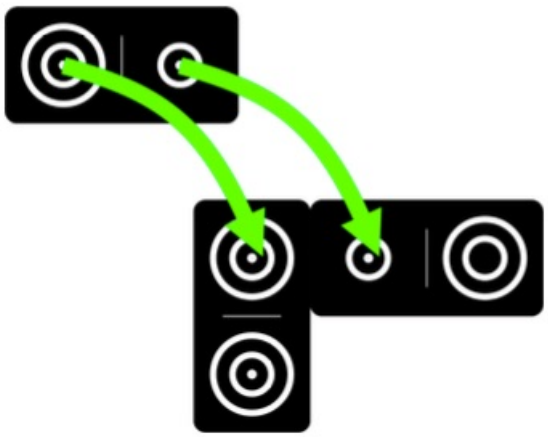

Figura 2: Exemplo de um posicionamento a subir válido.

O jogador com o duplo 7 inicia o jogo, colocando essa peça no centro da superfície de jogo e determinando a ordem dos restantes jogadores, que é dada pelo sentido contrário ao ponteiro dos relógios.

Começando no segundo, cada jogador, na sua vez, realiza ambos os passos seguintes:

1. Enquanto for possível, coloca peças a subir, podendo escolher a ordem em que o faz;
2. Se ainda tiver alguma peça, coloca-a a expandir.

Se, no final da sua vez, o jogador ficar sem peças, é declarado vencedor e o jogo termina. Alternativamente, os restantes jogadores podem continuar, de forma a determinar o segundo, terceiro e quarto lugares.

Na Figura 3 pode ser observado um possível jogo de Dominup a decorrer.

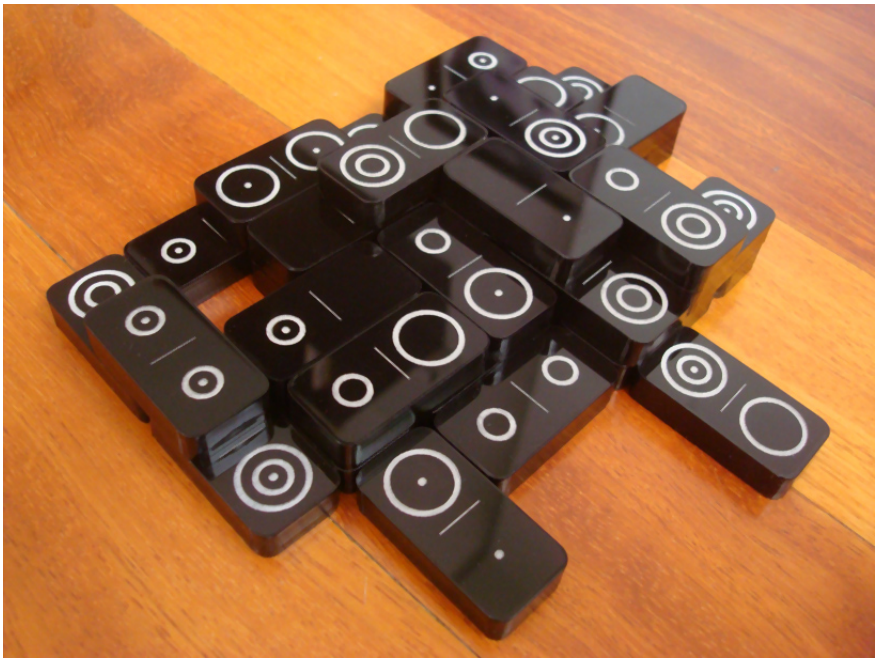

Figura 3: Exemplo de um jogo de Dominup.

3 Lógica do Jogo

Pretendemos nas secções que se seguem descrever o projeto e implementação da lógica do jogo em Prolog, incluindo a forma de representação do estado do tabuleiro e sua visualização, a execução de movimentos, verificação do cumprimento das regras do jogo, determinação do final do jogo e cálculo das jogadas a realizar pelo computador, utilizando diversos níveis de inteligência.

3.1 Representação do Estado do Jogo

O tabuleiro de jogo no Dominup é um quadriculado, cujo tamanho não deve limitar o posicionamento de peças a expandir. Uma vez que expansões sucessivas têm que ser feitas ortogonalmente, uma linha expansiva numa só direção ocupa $2 + 1 + 2 + 1 + \cdots$ quadrículas, assumindo que a primeira peça está colocada horizontalmente. Ao todo temos 36 peças, por isso seriam necessárias $2*18+18 = 54$ quadrículas para acomodar uma tal linha de expansão. Dado que a primeira peça é colocada no centro do tabuleiro, se a expansão fosse feita sempre no mesmo sentido, poderíamos ter que considerar um tabuleiro com 108 quadrículas de lado.

Uma análise mais cuidada das regras do jogo revela que as peças são preferencialmente colocadas a subir. De facto, em cada vez, um jogador coloca tantas peças a subir quanto possível e no máximo uma peça a expandir. Além disso, em geral não será boa estratégia para nenhum dos jogadores expandir sempre no mesmo sentido. Tendo em consideração as limitações de um computador, quer em termos de capacidade de processamento, quer em termos de tamanho do ecrã, decidimos considerar um tabuleiro quadriculado de lado 18. Desta forma, é possível colocar pelo menos 5 peças em cada um dos 4 sentidos a partir do centro do tabuleiro. A experiência revela que este tamanho por vezes se torna limitativo. No entanto, tamanhos superiores entram em desacordo com o nosso princípio de manter uma apresentação visual intuitiva e agradável.

Inicialmente, o tabuleiro era representado por uma lista `board` de linhas do tabuleiro. Por sua vez, cada linha é uma lista de elementos, um para cada quadrícula. Rapidamente nos apercebemos que dado o tamanho do tabuleiro esta solução não seria nada eficiente, além de que se tornava bastante complexa. Adicionalmente, o jogo que pretendíamos implementar apresenta a facilidade de ser essencialmente construtivo, as peças são colocadas na sua posição final, não havendo mudanças. Sendo assim partimos para uma implementação do tabuleiro como um conjunto de factos do tipo `halfPiece(Line, Column, Level, Number, Cardinal)` representando a meia peça de dominó lá colocada, com o seguinte significado:

- `Line` é o linha do tabuleiro;
- `Column` é a coluna do tabuleiro;
- `Level` é o nível do tabuleiro em que a peça está colocada, 1 se for colocada em cima do tabuleiro, 2 se for colocada em cima dessa, etc;
- `Number` é o número da meia peça;
- `Cardinal` é o ponto cardeal que indica a posição da outra metade da peça (n, e, s, w).

Assim, numa quadrícula não temos qualquer predicado do tipo `halfPiece`. Quando é colocado o dominó duplo 7 nas quadrículas `(9, 9)` e `(9, 10)`, teremos `halfPiece(9, 9, 1, 7, e)` e `halfPiece(9, 10, 1, 7, w)`. Numa fase mais avançada pode ser colocado o dominó 2·6 no nível 3 na vertical com `halfPiece(4, 5, 3, 2, n)` e `halfPiece(3, 5, 3, 6, s)`, por exemplo.

Além do tabuleiro, o estado de jogo contém as peças de cada jogador. Dado que apenas consideramos dois jogadores nesta implementação, temos factos do tipo `piece(Number1, Number2, Player, Played)`, com o seguinte significado:

- `Number1` é o menor número da peça;
- `Number2` é o maior número da peça;
- `Player` é 1 ou 2 consoante a peça pertença ao jogador 1 ou 2;
- `Played` é 0, se a peça ainda não foi jogada, e 1, caso contrário.

À medida que vão sendo colocadas no tabuleiro, as peças passam a ter o indicador `Played` a 1, ou seja, para cada peça jogada é revogado o facto que a representa e adicionado um novo facto, desta vez com `Played` a 1.

Finalmente, o estado de jogo tem indicação de qual é o próximo jogador a jogar no facto `turn(Number)`, em que number pode ser 1 ou 2.

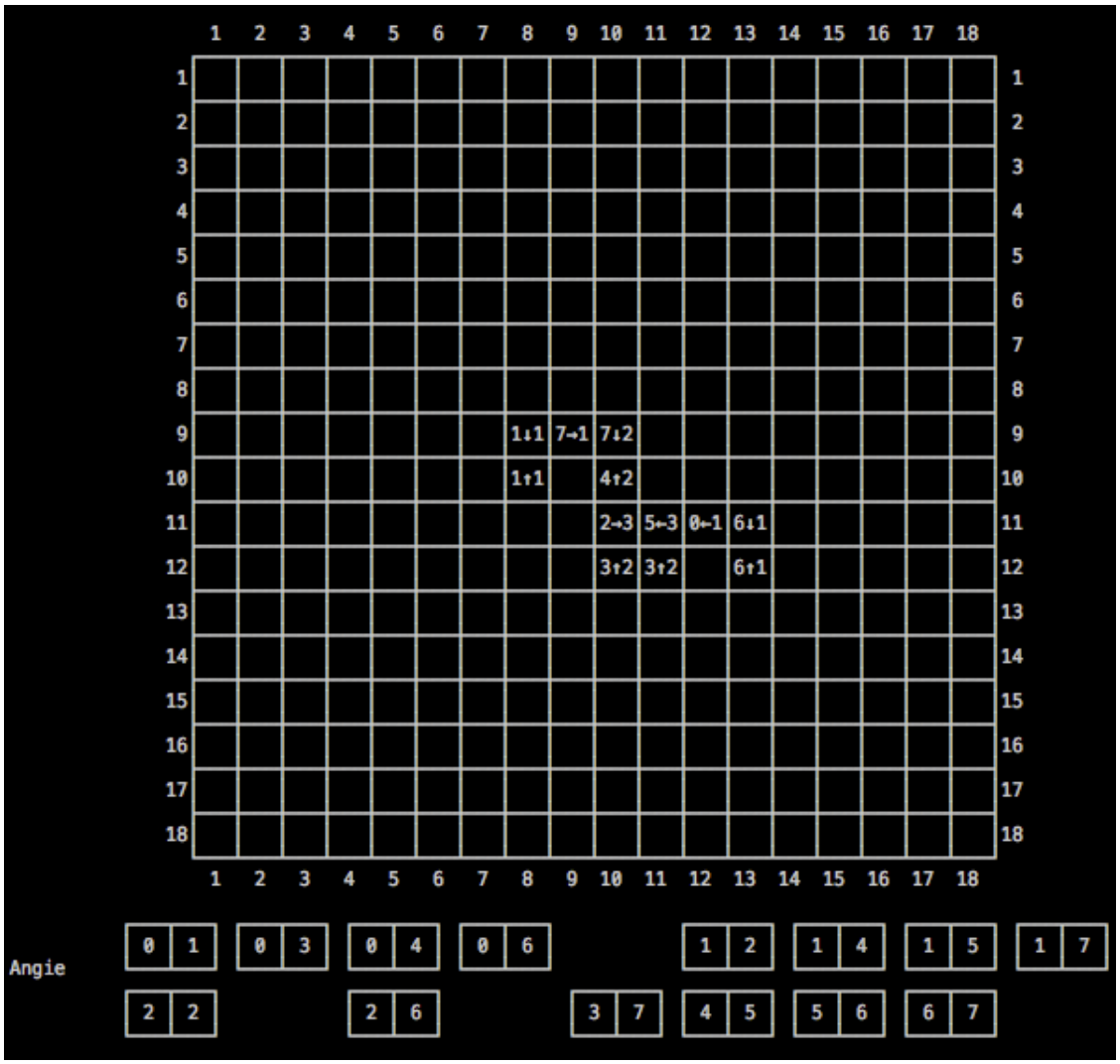

Figura 4: Estado intermédio do jogo.

Após algumas jogadas, podemos chegar ao estado ilustrado pela Figura 4. Em baixo estão as peças do jogador 1, dado que esse é o próximo a jogar. Em cima está o tabuleiro com as peças que contém até então. Cada meia peça colocada no tabuleiro tem do lado esquerdo o seu número e do lado direito o nível em que se encontra. Este estado é obtido com o código apresentado de seguida.

```
/* fixed distribution of pieces used in test phase */
testDistribute :-
        assert(piece(0, 1, 1, 0)) ,
        assert(piece(0, 3, 1, 0)) ,
        assert(piece(0, 4, 1, 0)) ,
        assert(piece(0, 6, 1, 0)) ,
        assert(piece(1, 1, 1, 0)) ,
        assert(piece(1, 2, 1, 0)) ,
        assert(piece(1, 4, 1, 0)) ,
        assert(piece(1, 5, 1, 0)) ,
        assert(piece(1, 7, 1, 0)) ,
```

```
        assert(piece(2, 2, 1, 0)) ,
        assert(piece(2, 3, 1, 0)) ,
        assert(piece(2, 6, 1, 0)) ,
        assert(piece(3, 3, 1, 0)) ,
        assert(piece(3, 7, 1, 0)) ,
        assert(piece(4, 5, 1, 0)) ,
        assert(piece(5, 6, 1, 0)) ,
        assert(piece(6, 7, 1, 0)) ,
        assert(piece(7, 7, 1, 0)) ,
        assert(piece(0, 0, 2, 0)) ,
        assert(piece(0, 2, 2, 0)) ,
        assert(piece(0, 5, 2, 0)) ,
        assert(piece(0, 7, 2, 0)) ,
        assert(piece(1, 3, 2, 0)) ,
        assert(piece(1, 6, 2, 0)) ,
        assert(piece(2, 4, 2, 0)) ,
        assert(piece(2, 5, 2, 0)) ,
        assert(piece(2, 7, 2, 0)) ,
        assert(piece(3, 4, 2, 0)) ,
        assert(piece(3, 5, 2, 0)) ,
        assert(piece(3, 6, 2, 0)) ,
        assert(piece(4, 4, 2, 0)) ,
        assert(piece(4, 6, 2, 0)) ,
        assert(piece(4, 7, 2, 0)) ,
        assert(piece(5, 5, 2, 0)) ,
        assert(piece(5, 7, 2, 0)) ,
        assert(piece(6, 6, 2, 0)) .

/* fixed plays to used in test phase */
testPlay :- playFirstPiece ,
        playPiece(2, 4, 2, 11, 10, n, _) ,
        playPiece(3, 3, 1, 12, 10, e, _) ,
        playPiece(4, 7, 2, 10, 10, n, _) ,
        playPiece(0, 5, 2, 11, 12, w, _) ,
        playPiece(2, 3, 1, 11, 10, s, _) ,
        playPiece(1, 1, 1, 9, 8, s, _) ,
        playPiece(3, 5, 2, 12, 11, n, _) ,
        playPiece(2, 5, 2, 11, 10, e, _) ,
        playPiece(6, 6, 2, 11, 13, s, _) .

/* fixed game start with distribution and plays used in test phase */
test :-
        testDistribute ,
        testPlay ,
        assert(player(1, 'Angie', 1)) ,
        assert(player(2, 'Nuno', 1)).
```

Um possível estado final, tem a ilustração da Figura 5, onde se pode observar além do tabuleiro, anunciado qual o jogador que venceu.

3.2 Visualização do Tabuleiro

Para visualizar o tabuleiro em modo de texto, cada quadrícula é sempre desenhada com traços a toda a volta, independentemente da posição das peças. Cada peça é colocada em duas quadrículas adjacentes e a relação entre as duas metades é identificada pelo conteúdo das células. Isto é, o dominó $2 \cdot 5$ dá origem às células `2→3` e `5←3`, se for colocado no nível 3 na horizontal com o 2 à esquerda do 5. O resultado é aquele que se pode observar nas Figuras 4 e 5.

Os predicados usados para este efeito cuja definição pode ser consultada no Anexo A são os seguintes:

- `printBoard` - para imprimir o tabuleiro de jogo;
- `printRows` - para imprimir as linhas do tabuleiro;
- `printCells` - para imprimir as quadrículas do tabuleiro;

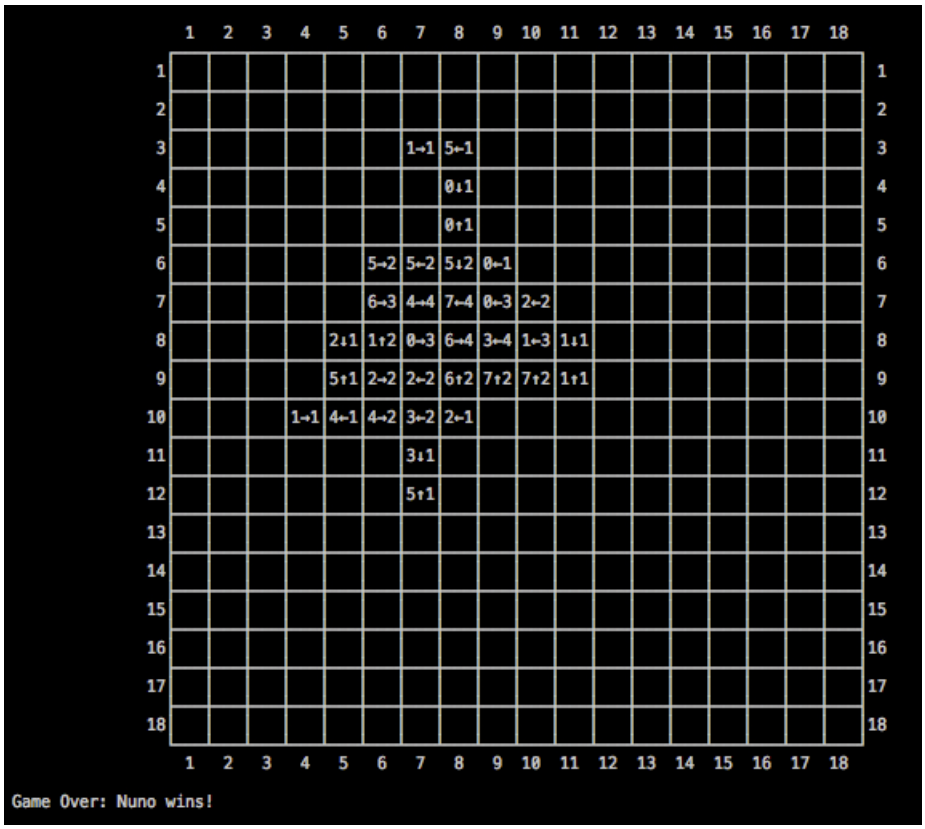

Figura 5: Estado final do jogo.

- `getTopLevel(X, Y, L)` - para obter o nível mais elevado de uma quadrícula;
- `printCardinal` - para imprimir o cardal (n, e, s, w) usando setas;
- `printNumbers` - para imprimir os números no topo e no fim do tabuleiro;
- `printLeftNumbers(X)` - para imprimir o número à esquerda da linha X do tabuleiro;
- `printRightNumbers(X)` - para imprimir o número à direita da linha X do tabuleiro;
- `printGridTop` - para imprimir o topo da grelha do tabuleiro;
- `printGrid(X)` - para imprimir a linha X da grelha do tabuleiro;
- `printSpaces(N)` - para imprimir N espaços;
- `printPlayerName(I)` - para imprimir o nome do jogador I usando exatamente N caracteres;
- `printPlayer(I)` - para imprimir o jogador I;
- `printPieces(I, C, R, N1T, N2T, N1N, N2N, N1B, N2B)` - para imprimir o nome e as peças do jogador I, todos os restantes argumentos começam em 0;
- `printPieceTop(P)` - para imprimir o rebordo do topo da peça, onde P indica se já foi jogada ou não;
- `printPieceNumber(P)` - para imprimir os números da peça, onde P indica se já foi jogada ou não;
- `printPieceBottom(P)` - para imprimir o rebordo do fundo da peça, onde P indica se já foi jogada ou não;
- `printGame(I)` - para imprimir o jogo, de forma que o jogador I possa jogar na sua vez;
- `printGameOver(I)` - para imprimir o estado final do jogo, quando venceu o jogador I.

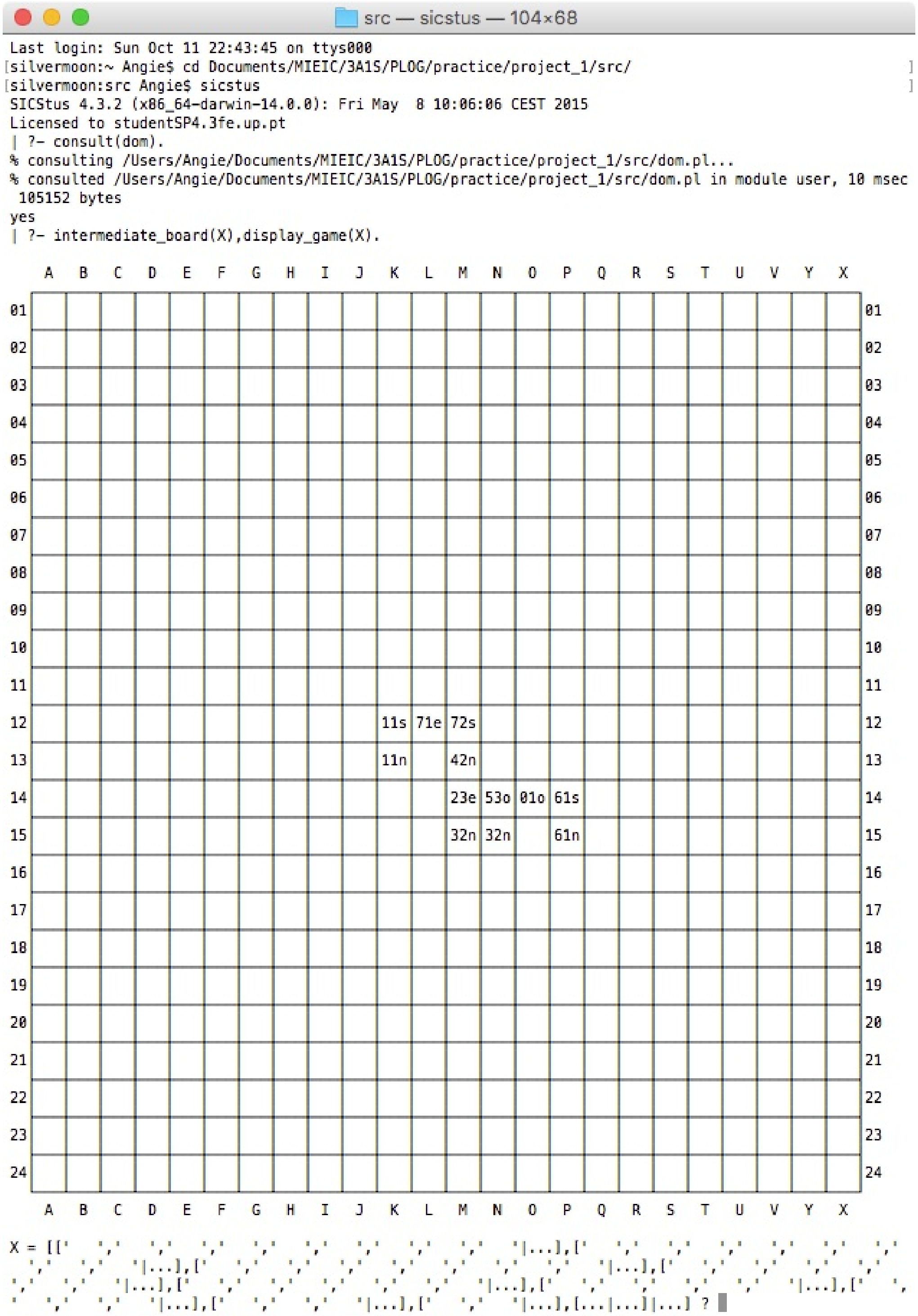

Figura 6: Visualização do tabuleiro em Prolog.

3.3 Lista de Jogadas Válidas

Uma vez que no Dominup há dois tipos distintos de movimentos, subir e expandir, usamos duas listas de jogadas válidas. Estas listas são usadas por ambas as inteligências artificiais para decidir os seus próximos movimentos.

Para obter estas listas são usados os predicados que se apresentam de seguida.

```
/* predicate used to check if a climb movement is valid */
/* this is very similar to checkPlay above, but only for climbs, and
   works with findall */
checkClimb(N1, N2, I, X1, Y1, C1) :-
        checkPlayerPiece(N1, N2, I) ,                          /* check
            if the piece belongs to the player and has not been played
           */
        getTopLevel(X1, Y1, L) ,                               /*
           obtain the current level of the position on the board */
        getOtherHalf(X1, Y1, C1, X2, Y2, _) ,                  /*
           obtain the position of the other half of the piece */
        checkInsideBoard(X1, Y1, X2, Y2) ,                     /*
           verify that the whole piece is inside the board */
        checkLevelStable(L, X2, Y2) ,                          /*
           verify that the level of both halfs is the same */
        checkClimbNumbers(N1, X1, Y1, N2, X2, Y2, L).          /* check
            that the numbers are correct for climbing */

/* predicate used to check if an expand movement is valid */
/* this is very similar to checkPlay above, but only for expand, and
   works with findall */
checkExpand(N1, N2, I, X1, Y1, C1) :-
        checkPlayerPiece(N1, N2, I) ,                          /* check
            if the piece belongs to the player and has not been played
           */
        position(X1, Y1),                                      /*
           verify that X1, Y1 is a valid board position */
        getOtherHalf(X1, Y1, C1, X2, Y2, C2) ,                 /*
           obtain the position of the other half of the piece */
        checkInsideBoard(X1, Y1, X2, Y2) ,                     /*
           verify that the whole piece is inside the board */
        checkLevelStable(0, X1, Y1),                           /*
           verify that the level of both halfs is the 0 */
        checkLevelStable(0, X2, Y2) ,
        checkExpandOrthogonal(X1, Y1, C1, X2, Y2, C2).         /* check
            that there is some other piece on the board orthogonal to
           this one */

/* predicate used to obtain a list of all possible climb movements for a
    player */
getClimbPlays(I, Plays) :-
        findall(play(N1, N2, I, X1, Y1, C1), checkClimb(N1, N2, I, X1,
           Y1, C1), Plays).

/* predicate used to obtain a list of all possible expand movements for
   a player */
getExpandPlays(I, Plays) :-
        findall(play(N1, N2, I, X1, Y1, C1), checkExpand(N1, N2, I, X1,
           Y1, C1), Plays).
```

Estes predicados invocam outros que não estão aqui representados, mas os seus nomes e comentários são elucidativos. De qualquer forma, a sua definição está no Anexo A e na secção seguinte.

3.4 Execução de Jogadas

Sempre que o jogador humano indica um novo movimento este é validado e, em caso de sucesso, a peça é jogada. Para isso usam-se os predicados seguintes:

```
/* predicate used to play a piece */
playPiece(N1, N2, I, X1, Y1, C1, L) :-                          /* L is
    output */
        checkPlay(N1, N2, I, X1, Y1, C1, X2, Y2, C2, L) ->      /* check
            if this play is valid */
        (L1 is L + 1 ,                                          /* the
           new level is 1 above the current level */
         assert(halfPiece(X1, Y1, L1, N1, C1)) ,                /*
            create first halfPiece */
         assert(halfPiece(X2, Y2, L1, N2, C2)) ,                /*
            create second halfPiece */
         retract(piece(N1, N2, I, 0)) ,                         /*
            remove piece from player's hand */
         assert(piece(N1, N2, I, 1))) ;                         /* place
             piece back but marked as played */
        fail.                                                   /* if
           checkPlay fails, fail and do nothing */

/* predicate used to check if a play is valid */
checkPlay(N1, N2, I, X1, Y1, C1, X2, Y2, C2, L) :-              /* X2,
    Y2, C2 and L are output */
        getTopLevel(X1, Y1, L) ,                                /* find
           the current level of the position X1, Y1 */
        getOtherHalf(X1, Y1, C1, X2, Y2, C2) ,                  /*
           compute the position of the other half given X1, Y1 and C1 */
        checkInsideBoard(X1, Y1, X2, Y2) ,                      /*
           verify that the piece will be place inside the board */
        checkPlayerPiece(N1, N2, I) ,                           /*
           verify that the player holds the piece and it has not been
           played yet */
        checkLevelStable(L, X2, Y2) ,                           /*
           verify that the other half has the same level */
        (L == 0 ->                                              /* if
           the current level is 0 */
         checkNoClimbs(I) ,                                     /*
            verify that the player has no more climb moves */
         checkExpandOrthogonal(X1, Y1, C1, X2, Y2, C2) ;        /* and
            check that this expansion is orthogonal to another piece on
            the board */
         checkClimbNumbers(N1, X1, Y1, N2, X2, Y2, L)).         /*
            otherwise verify that the climb is on correct numbers */

/* predicate used to check if the piece is inside the board */
checkInsideBoard(X1, Y1, X2, Y2) :-
        check1InsideBoard(X1) ,                                 /*
           simply check each coordinate */
        check1InsideBoard(Y1) ,
        check1InsideBoard(X2) ,
        check1InsideBoard(Y2).

/* predicate used to check if a given coordinate is inside the board */
check1InsideBoard(Z) :- Z < 1 -> fail ; (Z > 18 -> fail ; true).

/* predicate used to check if the piece belongs to the player and has
    not been played */
checkPlayerPiece(N1, N2, I) :- piece(N1, N2, I, 0).

/* predicate used to check if the other half has the same level */
checkLevelStable(L, X2, Y2) :- getTopLevel(X2, Y2, L).

/* predicate used to check if the numbers for a climb placement are
    correct */
checkClimbNumbers(N1, X1, Y1, N2, X2, Y2, L) :- halfPiece(X1, Y1, L, N1,
     _) , halfPiece(X2, Y2, L, N2, _).
```

```
/* predicate used to check if the expand placement is orthogonal to some
    piece on the board */
checkExpandOrthogonal(X1, Y1, C1, X2, Y2, C2) :-
        checkHalfOrthogonal(X1, Y1, C1) ;                          /*
           succeeds if at least one half has an orthogonal piece */
        checkHalfOrthogonal(X2, Y2, C2).

/* predicate used to check if a piece half has an orthogonal on the
   board */
checkHalfOrthogonal(X, Y, C) :-
        checkNorthOrthogonal(X, Y, C) ;                            /*
           succeeds if at least one cardinal has an orthogonal piece*/
        checkEastOrthogonal(X, Y, C) ;
        checkSouthOrthogonal(X, Y, C) ;
        checkWestOrthogonal(X, Y, C).

/* predicate used to check if a piece half has an orthogonal on the
   North */
checkNorthOrthogonal(X, Y, C) :-
        X1 is X - 1 ,                                              /* North
            line is this line minus 1 */
        (member(C, [e, w]) ->                                      /* if C
           is East or West */
         halfPiece(X1, Y, 1, _, n) ;                               /* North
             half piece must have cardinal North */
         (C == s ->                                                /*
           alternatively, if C is South */
          (halfPiece(X1, Y, 1, _, e) ;                             /* North
              half piece can have cardinal East */
           halfPiece(X1, Y, 1, _, w))                              /* or
              West */
         ; fail)).                                                 /* in
            all other cases, fail */

/* predicate used to check if a piece half has an orthogonal on the East
    */
checkEastOrthogonal(X, Y, C) :-                                    /* very
   similar to North case */
        Y1 is Y + 1 ,
        (member(C, [s, n]) ->
         halfPiece(X, Y1, 1, _, e) ;
         (C == w ->
          (halfPiece(X, Y1, 1, _, n) ;
           halfPiece(X, Y1, 1, _, s)) ;
          fail)).

/* predicate used to check if a piece half has an orthogonal on the
   South */
checkSouthOrthogonal(X, Y, C) :-                                   /* very
   similar to North case */
        X1 is X + 1 ,
        (member(C, [w, e]) ->
         halfPiece(X1, Y, 1, _, s) ;
         (C == n ->
          (halfPiece(X1, Y, 1, _, e) ;
           halfPiece(X1, Y, 1, _, w)) ;
          fail)).

/* predicate used to check if a piece half has an orthogonal on the West
    */
checkWestOrthogonal(X, Y, C) :-                                    /* very
   similar to North case */
        Y1 is Y - 1 ,
```

```
        (member(C, [n, s]) ->
         halfPiece(X, Y1, 1, _, w) ;
         (C == e ->
          (halfPiece(X, Y1, 1, _, n) ;
           halfPiece(X, Y1, 1, _, s)) ;
          fail)).

/* predicate used to check if a player has no more climb moves */
checkNoClimbs(I) :-
        getClimbPlays(I, Plays) ,                                   /*
           compute all climb plays for player */
        length(Plays, Number) ,                                     /*
           obtain the number of climb plays */
        Number == 0.                                                /* if it
            is 0, there are no more climb plays */
```

3.5 Avaliação do Tabuleiro

A inteligência artificial mais complexa necessita de avaliar o estado do jogo a cada passo para decidir como prosseguir. Uma vez que o estado de jogo é um conjunto de factos, para fazer esta avaliação, é jogada uma das peças possíveis, depois o estado é avaliado e a peça é retirada do tabuleiro. Note-se que é necessário fazer isto para todas as jogadas possíveis, de forma a escolher a que apresenta melhor avaliação. Tudo isto é feito com os predicados abaixo.

```
/* predicate used to evaluate the current situation of the game for a
   given player */
evaluateSituation(I, R) :-                                          /* R is
   the result, bigger R means better for player I */
        getNumberPlays(I, Rme) ,                                    /*
           compute the number of pieces player I can play */
        I1 is 3 - I ,                                               /* I1 is
            the other player */
        getNumberPlays(I1 , Ryou) ,                                 /*
           compute the number of pieces the other player can play */
        R is Rme - Ryou.                                            /* the
           result is the difference */

/* predicate used to compute the maximum number of plays for a given
   player*/
getNumberPlays(I, R) :-
        getClimbPlays(I, Plays),                                    /* get
           list of all climbs */
        length(Plays, S) ,                                          /*
           compute maximum number of climbs */
        R is S + 1.                                                 /* the
           result is the number of climbs plus 1, because there is
           always 1 expand */

/* predicate used to play a piece without checking it is valid */
playPieceNoCheck(N1, N2, I, X1, Y1, C1, L) :-                       /* very
   similar to playPiece, but faster */
        getOtherHalf(X1, Y1, C1, X2, Y2, C2) ,
        getTopLevel(X1, Y1, L) , L1 is L + 1 ,
        assert(halfPiece(X1, Y1, L1, N1, C1)) ,
        assert(halfPiece(X2, Y2, L1, N2, C2)) ,
        retract(piece(N1, N2, I, 0)) ,
        assert(piece(N1, N2, I, 1)).

/* predicate used to remove a piece from the board */
removePiece(N1, N2, I, X1, Y1, C1, L) :-                            /* does
   the opposite of playPieceNoCheck */
        getOtherHalf(X1, Y1, C1, X2, Y2, C2) ,
        getTopLevel(X1, Y1, L) ,
```

```
        retract(halfPiece(X1, Y1, L, N1, C1)) ,
        retract(halfPiece(X2, Y2, L, N2, C2)) ,
        assert(piece(N1, N2, I, 0)) ,
        retract(piece(N1, N2, I, 1)).

/* predicate used to evaluate how good a given climb is */
evaluateClimb(N1, N2, I, X1, Y1, C1, R) :-                          /* R is
   the result, bigger R means better climb */
        checkClimb(N1, N2, I, X1, Y1, C1) ,                         /* first
            check if the climb is valid */
        playPieceNoCheck(N1, N2, I, X1, Y1, C1, _) ,                /* then
           play it */
        evaluateSituation(I, R) ,                                   /* then
           evaluate the current situation */
        removePiece(N1, N2, I, X1, Y1, C1, _).                      /* then
           remove the played piece */

/* predicate used to evaluate how good a given expand is */
evaluateExpand(N1, N2, I, X1, Y1, C1, R) :-                         /* very
   similar to evaluateClimb */
        checkExpand(N1, N2, I, X1, Y1, C1) ,
        playPieceNoCheck(N1, N2, I, X1, Y1, C1, _) ,
        evaluateSituation(I, R) ,
        removePiece(N1, N2, I, X1, Y1, C1, _).

/* predicate used to obtain a list of all climb plays evaluated */
evaluateClimbPlays(I, Plays) :-
        findall(play(N1, N2, I, X1, Y1, C1, R), evaluateClimb(N1, N2, I,
            X1, Y1, C1, R), Plays).

/* predicate used to obtain a list of all expand plays evaluated */
evaluateExpandPlays(I, Plays) :-
        findall(play(N1, N2, I, X1, Y1, C1, R), evaluateExpand(N1, N2, I
           , X1, Y1, C1, R), Plays).
```

3.6 Final do Jogo

A cada passo do jogo, é verificado se este terminou, o que acontece quando um dos jogadores ficar sem peças. Esta avaliação é feita pelo seguinte predicado em Prolog:

```
/* predicate used to check if the game is over */
checkGameOver :-                                                    /* the
   name is misleading, fails if game over, succeeds otherwise */
        numberPieces(1, 0, 0, 0, R1) ,                              /* get
           number of pieces of player 1 */
        (R1 == 0 ->                                                 /* if
           player 1 has 0 pieces */
         (printGameOver(1), fail) ;                                 /*
            player 1 has won, print that and fail */
         (numberPieces(2, 0, 0, 0, R2),                             /*
            otherwise, get number of pieces of player 2 */
          (R2 == 0 ->                                               /* if
             player 2 has 0 pieces */
           (printGameOver(2) , fail) ;                              /*
              player 2 has won, print that and fail */
           true))).                                                 /*
              otherwise the game continues */
```

3.7 Jogada do Computador

A jogada do computador é feita consoante o nível de inteligência. Isto é verificado durante a vez do jogador, dado o seu tipo. De facto, em caso de jogador humano (tipo 1), o motor de jogo invoca

os predicados que pedem ao utilizador para indicar a jogada e que a executam; em caso de jogador computador fácil, é invocado o predicado `playRandom`, que faz uma sequência de jogadas aleatórias; em caso de jogador computador difícil, é invocado o predicado `playBest`, que faz uma sequência de jogadas em modo ganancioso, isto é, escolhendo sempre uma das que lhe parecem melhores. A definição destes predicados é a seguinte:

```
/* predicate used play a random climb movement */
randomClimbPlay(I) :-
        getClimbPlays(I, Plays) ,
        random_member(play(N1, N2, I, X1, Y1, C1), Plays) ,
        playPiece(N1, N2, I, X1, Y1, C1, _).

/* predicate used play a random expand movement */
randomExpandPlay(I) :-
        getExpandPlays(I, Plays) ,
        random_member(play(N1, N2, I, X1, Y1, C1), Plays) ,
        playPiece(N1, N2, I, X1, Y1, C1, _).

/* predicate used play a random turn */
playRandom(I) :-
        randomClimbPlay(I) ->                                   /* while
            there are valid climbs */
        playRandom(I) ;                                         /* do
           random climbs */
        randomExpandPlay(I).                                    /*
           afterward do one expand */

/* predicate used to play one of the best climb plays available */
bestClimbPlay(I) :-
        evaluateClimbPlays(I, Plays) ,
        bestPlay(Plays, play(N1, N2, I, X1, Y1, C1, _)) ,
        playPiece(N1, N2, I, X1, Y1, C1, _).

/* predicate used to play one of the best expand plays available */
/* since there are usually many expand plays available, this is the
   slowest function taking a few seconds to finish */
bestExpandPlay(I) :-
        evaluateExpandPlays(I, Plays) ,
        bestPlay(Plays, play(N1, N2, I, X1, Y1, C1, _)) ,
        playPiece(N1, N2, I, X1, Y1, C1, _).

/* predicate used play a greedy turn, always choosing one of the best
   available plays */
playBest(I) :- bestClimbPlay(I) -> playBest(I) ; bestExpandPlay(I).
```

4 Interface com o Utilizador

O utilizador apenas necessita de ir digitando alguns caracteres no teclado, seguidos da tecla Enter, interagindo dessa forma com o jogo.

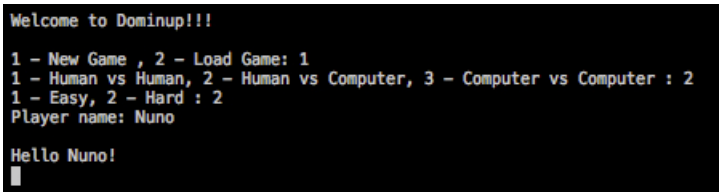

Figura 7: Interface inicial do jogo

No ecrã inicial o jogador tem de escolher se pretende iniciar um novo jogo ou um previamente gravado. Se escolher um jogo já gravado, este é carregado com as definições inicialmente selecionadas pelo utilizador

e o jogo continuará a se desenrolar. Caso contrário, o jogador tem de escolher se pretende jogar contra o computador, contra outro jogador ou observar um jogo entre dois computadores.

Sendo o nível de dificuldade, fácil ou difícil, escolhido logo a seguir, para terminar a configuração do jogo falta apenas escrever o(s) nome(s) do(s) jogador(es). A Figura 7 ilustra o interface inicial com o utilizador.

O jogo inicia mostrando o tabuleiro de jogo com a peça duplo 7 colocada no meio do tabuleiro. O jogador que tem a vez é aquele que não tinha essa peça, dado que o outro já jogou o duplo 7 como obrigam as regras. Para jogar, cada jogador tem acesso às suas peças e é convidado a escolher o movimento para realizar uma jogada. Para tal deve indicar o seguinte:

- colocar o número do lado esquerdo da peça que pretende jogar;
- colocar o número da lado direito da peça que pretende jogar
- indicar onde pretende colocar a metade esquerda da peça, indicando para isso a linha e a coluna
- indicar a orientação da peça, dizendo para isso onde se situa a metade direita da peça em relação à metade esquerda, utilizando as opções n, e, s e w, designando norte, este, sul e oeste, respetivamente.

Em lugar de escolher um movimento, o jogador pode também indicar que pretende salvar o jogo e sair. A Figura 8 ilustra o interface com o utilizador durante o jogo.

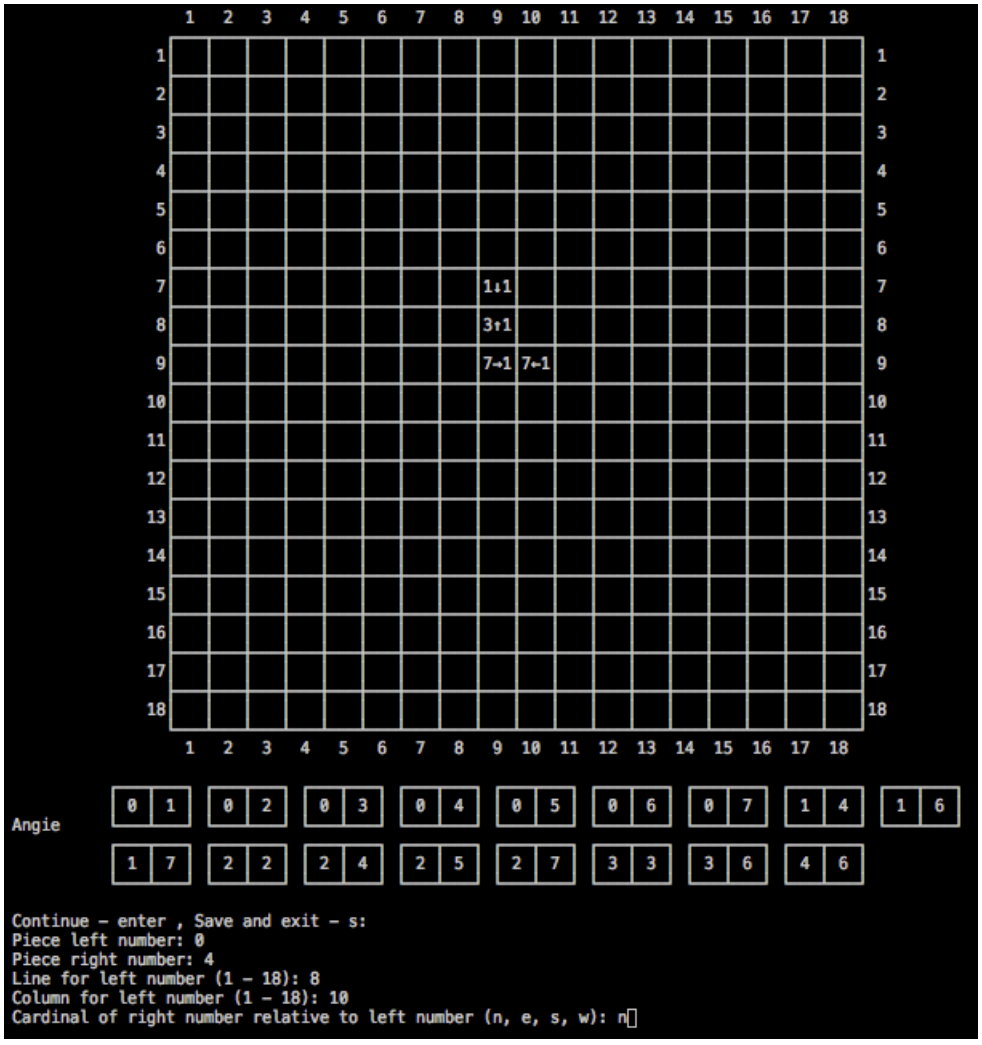

Figura 8: Interface de uma jogada

5 Conclusões

Este jogo foi desde logo uma das nossas primeiras hipóteses de seleção, de entre quatro possíveis, para se desenvolver o primeiro projeto em Prolog que seria objeto de avaliação. Ficamos extremamente felizes com o desenvolvimento de um jogo que à partida trazia alguma nostalgia pelo simples facto de nos lembrar alguns momentos, em que outrora um simples jogo de Dominó fazia a delícia dos mais pequenos.

Somos da opinião que este jogo trás uma espécie de reformulação do típico jogo de dominó e que faz todo o sentido coloca-lo disponível a jogar num computador, apesar de ainda em modo de texto.

Relativamente ao projeto consideramos que tudo o que delineamos e que todos os pontos exigidos na implementação do jogo foram cumpridos. Foi muito proveitoso uma vez que serviu também para praticarmos e aumentarmos a nossa prática em programação lógica.

Gostaríamos de ter desenvolvido mais um nível de inteligência artificial. De momento o nível difícil não avalia como as jogadas do adversário podem afetar os planos do computador, por exemplo. Também queríamos ter colocado a dimensão do tabuleiro de jogo a ser adaptada dinamicamente, isto é, consoante a colocação de peças e os movimentos de expansão, o tabuleiro ir aumentando de dimensão.

Todavia, estamos agradados com o rumo que o projeto tomou e o resultado final foi o que esperávamos.

6 Bibliografia

- http://www.nestorgames.com
- https://sicstus.sics.se/sicstus/docs/4.0.3/html/sicstus/Saving.html
- https://sicstus.sics.se/sicstus/docs/latest4/pdf/sicstus.pdf
- Apontamentos das aulas teóricas

A Código fonte do jogo Dominup em Prolog

Apresenta-se a seguir todo o código fonte utilizado no projeto. Como não conseguimos processar os caracteres unicode para desenhar caixas na consola no LaTeX, o código do ficheiro display.pl é ligeiramente diferente, mas apenas nos predicados que desenham a grelha do tabuleiro e as grelhas das peças.

A.1 Ficheiro main.pl

```
/* -*- Mode:Prolog; coding:iso-8859-1; -*- */

/*************/
/* libraries */
/*************/

:- use_module(library(system)).
:- use_module(library(random)).
:- use_module(library(file_systems)).

:-consult(display).
:-consult(dominup).


/****************/
/* useful stuff */
/****************/

/* not predicate */
not(P) :-
        (P -> fail ; true).

/* predicate used to remove all characters up to a new line from the
   input */
getNewLine :-
        get_code(T) , (T == 10 -> ! ; getNewLine).

/* predicate used to get a single char from the input */
/* does not require full dot at the end,
   removes all other characters up to a new line,
   works also if user only presses enter */
getChar(C) :-
        get_char(C) , char_code(C, Co) , (Co == 10 -> ! ; getNewLine).

/* predicate used to get a single algarism from the input */
/* does not require full dot at the end,
   removes all other characters up to a new line,
   works also if user only presses enter
   but the number will be -38 */
getDigit(D) :-
        get_code(Dt) , D is Dt - 48 , (Dt == 10 -> ! ; getNewLine).

/* predicate used to get a possible double algarism number from the
   input */
/* does not require full dot at the end,
   removes all other characters up to a new line,
   works also if user only presses enter
   but the number will be -38 */
getDoubleDigit(D) :-
        get_code(D1t) ,
        (D1t == 10 -> ! ;
         (get_code(D2t) ,
          (D2t == 10 ->
           (D is D1t - 48) ;
           (getNewLine ,
            D is (D1t - 48) * 10 + D2t - 48)))).
```

```
55
/* predicate used to get a list of chars from the input */
/* does not require full dot at the end,
   removes the new line character,
   the result is placed on OutList,
60    InList should be the empty list */
getCharList(InList, OutList) :-
        get_char(C) ,
        char_code(C, Co) ,
        ( Co == 10 ->
65        OutList = InList ;
          append(InList, [C], InList1),
          getCharList(InList1, OutList)).

/* places the first N elemenst of the list L on the list P */
70 trim(L, N, P) :-
        length(L, M) ,
        (N >= M ->
         P = L ;
         (X is M - N ,
75        length(S, X) ,
          append(P, S, L))).

/* gets an atom with at most 8 characters from the input */
/* does not require full dot at the end,
80    removes the new line character */
get8String(S) :-
        getCharList([], InList) ,
        trim(InList, 8, OutList) ,
        atom_chars(S, OutList).
85
/* gets an atom with any number of characters from the input */
/* does not require full dot at the end,
   removes the new line character */
getString(S) :-
90      getCharList([], OutList) ,
        atom_chars(S, OutList).

/* clears the screen on unix consoles */
cls :- write('\e[H\e[J\e[3J').
95

/************/
/* start up */
/************/
100
/* computes the seed for the random number generator in sicstus */
seedRandom :-
        now(B) ,
        X is B mod 30268 + 1 ,
105     Y is B mod 30306 + 1 ,
        Z is B mod 30322 + 1 ,
        setrand(random(X, Y, Z, B)).
:- seedRandom.

110 /* suceeds with 1/2 probability, fails in other cases */
maybeRandom :-
        random(1, 3, I) ,
        (I == 1 -> true ; fail).

115 /* asks the user if they want to play a new game or load a saved game */
newOrLoad(Option) :-
        prompt(_, '1 - New Game , 2 - Load Game: ') ,
        getDigit(Optiont) ,
```

```
        (Optiont < 1 -> (write('Must be 1 or 2.') , nl , newOrLoad(
            Option)) ;
         (Optiont > 2 -> (write('Must be 1 or 2.') , nl , newOrLoad(
            Option)) ;
          Option is Optiont)).

:- dynamic player/3.
/* player (Number, Name, Type) */
/* Number is 1 or 2, randomly decided at the start of a new game */
/* Name is chosen by player if human, otherwise is Computer or Compute1
   or Compute2 */
/* Type is 1 if Human, 2 if Easy, 3 if Hard */

/* asks the user if about the type of game (human vs human, human vs
   computer, computer vs computer) */
computerOrHuman(Option) :-
        prompt(_, '1 - Human vs Human, 2 - Human vs Computer, 3 -
           Computer vs Computer : ') ,
        getDigit(Optiont) ,
        (Optiont < 1 -> (write('Must be 1, 2 or 3.') , nl ,
           computerOrHuman(Option)) ;
         (Optiont > 3 -> (write('Must be 1, 2 or 2.') , nl ,
            computerOrHuman(Option)) ;
          Option is Optiont)).

/* asks the user if the computer artificial intelligence should be easy
   (random) or hard (greedy) */
easyOrHard(Option) :- prompt(_, '1 - Easy, 2 - Hard : ') ,
        getDigit(Optiont) , (Optiont < 1 -> (write('Must be 1 or 2.') ,
           nl , computerOrHuman(Option)) ;
                             (Optiont > 2 -> (write('Must be 1 or 2.') ,
                                nl , computerOrHuman(Option)) ;
                              Option is Optiont)).

/* asks the user for human player names and randomly chooses player
   numbers, creating the player predicates */
playerNames(CHOption, EHOption1, EHOption2) :-
        CHOption == 1 ->
        (prompt(_, 'Name of one player: ') ,
         get8String(X1) ,
         (X1 = '' -> X = 'Player 1' ; X = X1),
         prompt(_, 'Name of the other player: ') ,
         get8String(Y1) ,
         (Y1 = '' -> Y = 'Player 2' ; Y = Y1) ,
         nl , write('Hello ') ,  write(X) , write(' and ' ) , write(Y) ,
             write('!') , nl ,
         (maybeRandom ->
          (assert(player(1, X, 1)) , assert(player(2, Y, 1))) ;
          (assert(player(1, Y, 1)) , assert(player(2, X, 1))))) ;
        (CHOption == 2 ->
         (prompt(_, 'Player name: ') ,
          get8String(X1) ,
          (X1 = '' -> X = 'Player' ; X = X1) ,
          nl , write('Hello ') , write(X) , write('!') , nl ,
          Y = 'Computer' ,
          (maybeRandom ->
           (assert(player(1, X, 1)) , assert(player(2, Y, EHOption1))) ;
           (assert(player(1, Y, EHOption1)) , assert(player(2, X, 1)))))
               ;
          (assert(player(1, 'Compute1', EHOption1)) ,
           assert(player(2, 'Compute2', EHOption2)))).

/* saves the current state of the game and stops it */
save_game :-
```

```
        prompt(_, 'File name: ') ,
        getString(S) ,
        save_program(S) ,
        break.

/* loads a game state from a file */
load_game :-
        prompt(_, 'File name: ') ,
        getString(S) ,
        atom_concat(S, '.sav', Ssav) ,
        (file_exists(Ssav, exist) ->
         restore(S) ;
         (print(S) , write(' is not a valid save file') , nl , load_game
            )).

/* starts a game */
/* if the game has been loaded proceeds to play the next turn
   otherwise, asks the user for initial setup and then calls playGame*/
startGame :- player(1, _, _) -> playTurn ;
             (cls , nl , write('Welcome to Dominup!!!') , nl , nl ,
                newOrLoad(NLOption) ,
              (NLOption == 1 ->
               (computerOrHuman(CHOption) ,
                (CHOption == 1 -> (playerNames(CHOption, _, _)) ;
                 (CHOption == 2 -> (easyOrHard(EHOptiont) , EHOption is
                    EHOptiont + 1 , playerNames(CHOption, EHOption, _))
                    ;
                  (write('Choose level for Computer 1.') , nl ,
                     easyOrHard(EHOption1) , EHOption1 is EHOption1 +
                     1 ,
                   write('Choose level for Computer 2.') , nl ,
                      easyOrHard(EHOption2) , EHOption2 is EHOption2 +
                       1 ,
                   playerNames(CHOption, EHOption1, EHOption2) , prompt(
                      _, 'Enter to continue.'))))) , sleep(1) , playGame
                      ) ;
              load_game).

/* always begin by calling startGame */
:- initialization(startGame).
```

A.2 Ficheiro display.pl

```
/* -*- Mode:Prolog; coding:utf-8; -*- */

/***************/
/* print board */
/***************/

/* predicate used to print the game board */
printBoard :-
        nl , printNumbers , nl ,                          /* print the
           numbers on top of the board */
        printGridTop , nl ,                               /* print the
           grid top */
        printRows(1) ,                                    /* print the
           board rows */
        printNumbers , nl , nl , !.                       /* print the
           bottom numbers */

/* predicate used to print the rows of the board */
printRows(19) :- !.                                       /* stop after
   row 18 */
printRows(X) :-
        printLeftNumbers(X) ,                             /* print number
```

```
            on the left of the row */
        printCells(X, 1) ,                                  /* print row
            cells */
        printRightNumbers(X) , nl ,                         /* print numbers
             on the right of the row */
        printGrid(X) ,                                      /* print grid
            line */
        X1 is X + 1 , nl ,                                  /* move on to
            next row */
        printRows(X1).                                      /* calling the
            function recursively */

/* predicate used to print the cells of a row of the board */
printCells(_,19) :- !.                                      /* stop after
    cell 18 */
printCells(X, Y) :-
        getTopLevel(X, Y, L) ,                              /* determine the
             current level of the cell */
        (L > 0 ->                                           /* if the
            current level is above 0 */
         (halfPiece(X, Y, L, N, C) ,                        /* determine
            what half piece is at that top level */
          print(N) ,                                        /* print the
             number of the half piece */
          printCardinal(C) ,                                /* print the
             cardinal of the half piece */
          print(L)) ;                                       /* print the
             level of the half piece */
         write('   ')) ,                                    /* otherwise, if
             the level is 0, print 3 spaces */
        write('|') ,                                        /* in any case,
            print the cell grid divider */
        Y1 is Y + 1 ,                                       /* advance to
            the next cell on the same row */
        printCells(X, Y1).                                  /* calling the
            function recursively */

/* predicate used to obtain the top level of a given position */
getTopLevel(X, Y, 0) :- not(halfPiece(X, Y, _, _, _)).      /* if there is
    no half piece on that position the level is 0 */
getTopLevel(X, Y, L) :-                                     /* otherwise,
    the level is L if */
        halfPiece(X, Y, L, _, _) ,                          /* there is an
            half piece on that position on level L */
        L1 is L + 1 ,
        not(halfPiece(X, Y, L1, _, _)).                     /* and there is
            no half piece on that position on level L + 1 */

/* predicate used to print the cardinal of an half piece in an appealing
     way */
printCardinal(n) :- write('↑').
printCardinal(e) :- write('→').
printCardinal(s) :- write('↓').
printCardinal(w) :- write('→').

/* predicate used to print the numbers on top of the board */
printNumbers :-
        write('                  1   2   3   4   5   6   7   8   9  10
            11  12  13  14  15  16  17  18').

/* predicate used to print the numbers on the left of the board */
printLeftNumbers(X) :-
        ((X < 10,                                           /* for numbers
            before 10, there is one extra space */
```

```
            write('               ') ,
            print(X) ,
            write('|')) ;                                   /* also print
               the board grid divider */
           (write('               ') ,
            print(X) ,
            write('|'))).

/* predicate used to print the numbers on the right of the board */
printRightNumbers(X) :-
        ((X < 10 ,
          write(' '),
          print(X)) ;
         (print(X))).

/* predicate used to print the top of the board grid */
printGridTop :- write('                 ---------------------------------
   ----------------------------------------').

/* predicate used to print the board grid */
printGrid(X) :-((X < 18 ,                                   /* the last row
   is different */
                write('                 ---------------------------------
                   ----------------------------------------')) ;
                write('                 ---------------------------------
                   ----------------------------------------')).


/****************/
/* print player */
/****************/

/* predicate used to print N spaces */
printSpaces(N) :-
        N == 0 -> ! ;                                       /* when N = 0,
           stop */
        (write(' ') ,                                       /* otherwise
           write a space */
         N1 is N - 1 ,                                      /* decrease N */
         printSpaces(N1)).                                  /* and repeat */

/* predicate used to print a player's name  taking exactly 9 characters
   */
printPlayerName(I) :-
        player(I, S, _) ,                                   /* find the
           player name S, S has at most 8 characters */
        print(S) ,                                          /* print the
           player name */
        atom_length(S, N) ,                                 /* determine the
            name's length */
        M is 9 - N ,                                        /* M is how many
            characters are missing to 9 */
        printSpaces(M).                                     /* print M
           spaces */

/* predicate used to print a player's name and pieces, with starting
   spaces */
printPlayer(I) :- write('         ') , printPieces(I, 0, 0, 0, 0, 0, 0,
   0, 0).

/* predicate used to print a player's name and pieces */
/* this is a somewhat complex function and we do not present step by
   step comments.
   alternatively, here is the idea:
```

```
    - the pieces are displayed in two rows;
    - each piece has top, numbers and bottom;
    - so each piece row is in fact 3 rows: top, numbers and bottom;
    - to know what to print in each moment, we keep track of the current
      row R;
    - if R is 0 or 3, we are printing tops, for 1 and 4 we print numbers
      and the others are bottoms;
    - row 1 also has the player name;
    - we also need to keep track of the current piece numbers N1 and N2;
    - and we need to do it for all three aspects, top, number and bottom;
    - so we have N1T, N2T, N1N, N2N, N1B, N2B;
    - if a piece has already been played, we print spaces in its place;
    - this way, the pieces stay in the same place from beggining to end;
    - to determine if we should change row, we check if the counter C has
      reached 9;
    - because each row has 9 pieces;
    - we use specific predicates to print tops, numbers and bottoms */
printPieces(I, C, R, N1T, N2T, N1N, N2N, N1B, N2B) :-
        (C == 9 -> (nl , (R == 1 -> printPlayerName(I) ; write('
                   ')) ,
                    C1 is 0 , R1 is R + 1 , printPieces(I, C1, R1, N1T,
                       N2T, N1N, N2N, N1B, N2B)) ;
         ((R == 0 -> (N1T > 7 -> ! ;
                     (N2T > 7 -> (N2T1 is N1T + 1 , N1T1 is N1T + 1 ,
                        printPieces(I, C, R, N1T1, N2T1, N1N, N2N, N1B,
                         N2B)) ;
                      ((piece(N1T, N2T, I, P) -> C1 is C + 1 ,
                         printPieceTop(P) ; C1 is C),
                       N2T1 is N2T + 1 , printPieces(I, C1, R, N1T,
                         N2T1, N1N, N2N, N1B, N2B)))) ;
           ((R == 1 -> (N1N > 7 -> ! ;
                       (N2N > 7 -> (N2N1 is N1N + 1 , N1N1 is N1N + 1 ,
                           printPieces(I, C, R, N1T, N2T, N1N1, N2N1,
                          N1B, N2B)) ;
                        ((piece(N1N, N2N, I, P) -> C1 is C + 1 ,
                           printPieceNumber(N1N, N2N, P) ; C1 is C),
                         N2N1 is N2N + 1 , printPieces(I, C1, R, N1T,
                           N2T, N1N, N2N1, N1B, N2B)))) ;
             ((R == 2 -> (N1B > 7 -> ! ;
                         (N2B > 7 -> (N2B1 is N1B + 1 , N1B1 is N1B + 1
                             , printPieces(I, C, R, N1T, N2T, N1N, N2N,
                            N1B1, N2B1)) ;
                          ((piece(N1B, N2B, I, P) -> C1 is C + 1 ,
                             printPieceBottom(P) ; C1 is C),
                           N2B1 is N2B + 1 , printPieces(I, C1, R, N1T,
                             N2T, N1N, N2N, N1B, N2B1)))) ;
               ((R == 3 -> (N1T > 7 -> ! ;
                           (N2T > 7 -> (N2T1 is N1T + 1 , N1T1 is N1T +
                               1 , printPieces(I, C, R, N1T1, N2T1, N1N
                              , N2N, N1B, N2B)) ;
                            ((piece(N1T, N2T, I, P) -> C1 is C + 1 ,
                               printPieceTop(P) ; C1 is C),
                             N2T1 is N2T + 1 , printPieces(I, C1, R,
                               N1T, N2T1, N1N, N2N, N1B, N2B)))) ;
                 ((R == 4 -> (N1N > 7 -> ! ;
                             (N2N > 7 -> (N2N1 is N1N + 1 , N1N1 is N1N
                                 + 1 , printPieces(I, C, R, N1T, N2T,
                                N1N1, N2N1, N1B, N2B)) ;
                              ((piece(N1N, N2N, I, P) -> C1 is C + 1 ,
                                 printPieceNumber(N1N, N2N, P) ; C1 is
                                 C),
                               N2N1 is N2N + 1 , printPieces(I, C1, R,
                                 N1T, N2T, N1N, N2N1, N1B, N2B)))) ;
                   ((R == 5 -> (N1B > 7 -> ! ;
                               (N2B > 7 -> (N2B1 is N1B + 1 , N1B1 is
```

```
                                        N1B + 1 , printPieces(I, C, R, N1T,
                                        N2T, N1N, N2N, N1B1, N2B1)) ;
145                                    ((piece(N1B, N2B, I, P) -> C1 is C + 1
                                          , printPieceBottom(P) ; C1 is C),
                                       N2B1 is N2B + 1 , printPieces(I, C1, R
                                          , N1T, N2T, N1N, N2N, N1B, N2B1))))
                                           ;
                          !)))))))))))).

    /* predicate used to print a piece's top */
150 printPieceTop(P) :-
            P == 0 ->                                          /* if the piece
               has not been played */
            write(' ---------') ;                              /* print top */
            write('          ').                               /* otherwise
               print spaces */

155 /* predicate used to print a piece's numbers */
    printPieceNumber(N1, N2, P) :-
            P == 0 ->                                          /* if the piece
               has not been played */
            (write(' | ') ,                                    /* print left
               border */
             print(N1),                                        /* print left
                number */
160          write(' | ') ,                                    /* print center
                divider */
             print(N2) ,                                       /* print right
                number */
             write(' |')) ;                                    /* print right
                border */
            write('          ').                               /* otherwise
               print spaces */

165 /* predicate used to print a piece's bottom */
    printPieceBottom(P) :-                                     /* very similar
       to the printPieceTop */
            P == 0 ->
            write(' ---------') ;
            write('          ').
170

    /**************/
    /* print game */
    /**************/
175
    /* predicate used to print the current board and player pieces */
    printGame(I) :-
            cls , printBoard , printPlayer(I).

180 /* predicate used to print the board and the result at the end of the
       game */
    printGameOver(I) :-
            cls , printBoard ,
            write('Game Over: ') , player(I, S, _) , print(S) , write(' wins
               !') ,
            nl , nl , sleep(1).
```

A.3 Ficheiro dominup.pl

```
1 /* -*- Mode:Prolog; coding:utf-8; -*- */

  /*********************/
  /* distribute pieces */
5 /*********************/
```

```
:- dynamic piece/4.
/* piece(Number1, Number2, Player, Played). */
/* each piece has 2 numbers,
   the player it belongs to (1 or 2)
   and a toggle that tells if it has been played (0 if no, 1 if yes) */


/* random distribution of game pieces among players */
distributePieces(N1, N2, NI1, NI2) :-
        (N1 >=  7 -> assert(piece(N1, N2, 1, 0)) ;
           /* player 1 always gets piece 7 | 7 */
         (N2 > 7 -> (N21 is N1 + 1 , N11 is N1 + 1 ,
           /* recursively advance to next piece */
                    distributePieces(N11, N21, NI1, NI2)) ;
                       /* if x | 7 has been done, do x+1 | x+1 */
          (NI1 == 17 -> assert(piece(N1, N2, 2, 0)) ,
             /* if player 1 has 17 pieces all other */
                       NI21 is NI2 + 1 , N21 is N2 + 1 ,
                          /* pieces go to player 2, except 7 | 7 */
                       distributePieces(N1, N21, NI1, NI21) ;
                          /* as we saw above */
           (NI2 == 18 -> assert(piece(N1, N2, 1, 0)) ,
              /* if player 2 has 18 pieces all other */
                        NI11 is NI1 + 1 , N21 is N2 + 1 ,
                           /* pieces go to player 1 */
                        distributePieces(N1, N21, NI11, NI2) ;
            (maybeRandom -> (assert(piece(N1, N2, 1, 0)) ,
               /* otherwise, maybe atribute this piece to player 1 */
                           NI11 is NI1 + 1 , N21 is N2 + 1 ,
                              /* and advance to the next piece */
                           distributePieces(N1, N21, NI11, NI2)) ;
                              /* using the recursive function */
             (assert(piece(N1, N2, 2, 0)) ,
                /* or maybe atribute this piece to player 2 */
              NI21 is NI2 + 1, N21 is N2 + 1 ,
                 /* and advance to the next piece */
              distributePieces(N1, N21, NI1, NI21)))))))).
                 /* using the recursive function */

/* fixed distribution of pieces used in test phase, no longer needed for
    game to run */
testDistribute :-
        assert(piece(0, 1, 1, 0)) ,
        assert(piece(0, 3, 1, 0)) ,
        assert(piece(0, 4, 1, 0)) ,
        assert(piece(0, 6, 1, 0)) ,
        assert(piece(1, 1, 1, 0)) ,
        assert(piece(1, 2, 1, 0)) ,
        assert(piece(1, 4, 1, 0)) ,
        assert(piece(1, 5, 1, 0)) ,
        assert(piece(1, 7, 1, 0)) ,
        assert(piece(2, 2, 1, 0)) ,
        assert(piece(2, 3, 1, 0)) ,
        assert(piece(2, 6, 1, 0)) ,
        assert(piece(3, 3, 1, 0)) ,
        assert(piece(3, 7, 1, 0)) ,
        assert(piece(4, 5, 1, 0)) ,
        assert(piece(5, 6, 1, 0)) ,
        assert(piece(6, 7, 1, 0)) ,
        assert(piece(7, 7, 1, 0)) ,
        assert(piece(0, 0, 2, 0)) ,
        assert(piece(0, 2, 2, 0)) ,
        assert(piece(0, 5, 2, 0)) ,
```

```
        assert(piece(0, 7, 2, 0)) ,
        assert(piece(1, 3, 2, 0)) ,
        assert(piece(1, 6, 2, 0)) ,
        assert(piece(2, 4, 2, 0)) ,
        assert(piece(2, 5, 2, 0)) ,
        assert(piece(2, 7, 2, 0)) ,
        assert(piece(3, 4, 2, 0)) ,
        assert(piece(3, 5, 2, 0)) ,
        assert(piece(3, 6, 2, 0)) ,
        assert(piece(4, 4, 2, 0)) ,
        assert(piece(4, 6, 2, 0)) ,
        assert(piece(4, 7, 2, 0)) ,
        assert(piece(5, 5, 2, 0)) ,
        assert(piece(5, 7, 2, 0)) ,
        assert(piece(6, 6, 2, 0)) .

/**************/
/* play piece */
/**************/

:- dynamic halfPiece/5.
/* halfPiece(Line, Column, Level, Number, Cardinal). */
/* each halPiece has a position in the board (line, column and level),
   a number and the cardinal corresponding to its other half */

/* predicate used to obtain the position of the other half of a piece,
   given the position and cardinal of one half */
getOtherHalf(X1, Y1, n, X2, Y2, s) :- X2 is X1 - 1, Y2 is Y1.
getOtherHalf(X1, Y1, e, X2, Y2, w) :- X2 is X1 , Y2 is Y1 + 1.
getOtherHalf(X1, Y1, s, X2, Y2, n) :- X2 is X1 + 1, Y2 is Y1.
getOtherHalf(X1, Y1, w, X2, Y2, e) :- X2 is X1 , Y2 is Y1 - 1.

:- dynamic playPiece/6.
/* playPiece(Number1, Number2, Player, Line, Column, Cardinal). */
/* to play a piece we need its numbers, the player holding that piece
   and the position for the smaller half of the piece (line, column and
      cardinal) */

/* predicate used to play a piece */
playPiece(N1, N2, I, X1, Y1, C1, L) :-                          /* L is
   output */
        checkPlay(N1, N2, I, X1, Y1, C1, X2, Y2, C2, L) ->      /* check
           if this play is valid */
        (L1 is L + 1 ,                                          /* the
           new level is 1 above the current level */
         assert(halfPiece(X1, Y1, L1, N1, C1)) ,                /*
           create first halfPiece */
         assert(halfPiece(X2, Y2, L1, N2, C2)) ,                /*
           create second halfPiece */
         retract(piece(N1, N2, I, 0)) ,                         /*
           remove piece from player's hand */
         assert(piece(N1, N2, I, 1))) ;                         /* place
            piece back but marked as played */
        fail.                                                   /* if
           checkPlay fails, fail and do nothing */

/* predicate used to check if a play is valid */
checkPlay(N1, N2, I, X1, Y1, C1, X2, Y2, C2, L) :-              /* X2,
   Y2, C2 and L are output */
        getTopLevel(X1, Y1, L) ,                                /* find
           the current level of the position X1, Y1 */
        getOtherHalf(X1, Y1, C1, X2, Y2, C2) ,                  /*
           compute the position of the other half given X1, Y1 and C1 */
```

```
        checkInsideBoard(X1, Y1, X2, Y2) ,                          /*
           verify that the piece will be place inside the board */
        checkPlayerPiece(N1, N2, I) ,                               /*
           verify that the player holds the piece and it has not been
           played yet */
        checkLevelStable(L, X2, Y2) ,                               /*
           verify that the other half has the same level */
        (L == 0 ->                                                  /* if
           the current level is 0 */
         checkNoClimbs(I) ,                                         /*
            verify that the player has no more climb moves */
         checkExpandOrthogonal(X1, Y1, C1, X2, Y2, C2) ;            /* and
            check that this expansion is orthogonal to another piece on
            the board */
         checkClimbNumbers(N1, X1, Y1, N2, X2, Y2, L)).             /*
            otherwise verify that the climb is on correct numbers */

/* predicate used to check if the piece is inside the board */
checkInsideBoard(X1, Y1, X2, Y2) :-
        check1InsideBoard(X1) ,                                     /*
           simply check each coordinate */
        check1InsideBoard(Y1) ,
        check1InsideBoard(X2) ,
        check1InsideBoard(Y2).

/* predicate used to check if a given coordinate is inside the board */
check1InsideBoard(Z) :- Z < 1 -> fail ; (Z > 18 -> fail ; true).

/* predicate used to check if the piece belongs to the player and has
   not been played */
checkPlayerPiece(N1, N2, I) :- piece(N1, N2, I, 0).

/* predicate used to check if the other half has the same level */
checkLevelStable(L, X2, Y2) :- getTopLevel(X2, Y2, L).

/* predicate used to check if the numbers for a climb placement are
   correct */
checkClimbNumbers(N1, X1, Y1, N2, X2, Y2, L) :- halfPiece(X1, Y1, L, N1,
    _) , halfPiece(X2, Y2, L, N2, _).

/* predicate used to check if the expand placement is orthogonal to some
    piece on the board */
checkExpandOrthogonal(X1, Y1, C1, X2, Y2, C2) :-
        checkHalfOrthogonal(X1, Y1, C1) ;                           /*
           succeeds if at least one half has an orthogonal piece */
        checkHalfOrthogonal(X2, Y2, C2).

/* predicate used to check if a piece half has an orthogonal on the
   board */
checkHalfOrthogonal(X, Y, C) :-
        checkNorthOrthogonal(X, Y, C) ;                             /*
           succeeds if at least one cardinal has an orthogonal piece*/
        checkEastOrthogonal(X, Y, C) ;
        checkSouthOrthogonal(X, Y, C) ;
        checkWestOrthogonal(X, Y, C).

/* predicate used to check if a piece half has an orthogonal on the
   North */
checkNorthOrthogonal(X, Y, C) :-
        X1 is X - 1 ,                                               /* North
            line is this line minus 1 */
        (member(C, [e, w]) ->                                       /* if C
           is East or West */
         halfPiece(X1, Y, 1, _, n) ;                                /* North
```

```
            half piece must have cardinal North */
         (C == s ->                                                    /*
            alternatively, if C is South */
          (halfPiece(X1, Y, 1, _, e) ;                                 /* North
              half piece can have cardinal East */
           halfPiece(X1, Y, 1, _, w))                                  /* or
              West */
         ; fail)).                                                     /* in
            all other cases, fail */

/* predicate used to check if a piece half has an orthogonal on the East
    */
checkEastOrthogonal(X, Y, C) :-                                       /* very
   similar to North case */
        Y1 is Y + 1 ,
        (member(C, [s, n]) ->
         halfPiece(X, Y1, 1, _, e) ;
         (C == w ->
          (halfPiece(X, Y1, 1, _, n) ;
           halfPiece(X, Y1, 1, _, s)) ;
          fail)).

/* predicate used to check if a piece half has an orthogonal on the
   South */
checkSouthOrthogonal(X, Y, C) :-                                      /* very
   similar to North case */
        X1 is X + 1 ,
        (member(C, [w, e]) ->
         halfPiece(X1, Y, 1, _, s) ;
         (C == n ->
          (halfPiece(X1, Y, 1, _, e) ;
           halfPiece(X1, Y, 1, _, w)) ;
          fail)).

/* predicate used to check if a piece half has an orthogonal on the West
    */
checkWestOrthogonal(X, Y, C) :-                                       /* very
   similar to North case */
        Y1 is Y - 1 ,
        (member(C, [n, s]) ->
         halfPiece(X, Y1, 1, _, w) ;
         (C == e ->
          (halfPiece(X, Y1, 1, _, n) ;
           halfPiece(X, Y1, 1, _, s)) ;
          fail)).

/* predicate used to check if a player has no more climb moves */
checkNoClimbs(I) :-
        getClimbPlays(I, Plays) ,                                     /*
           compute all climb plays for player */
        length(Plays, Number) ,                                       /*
           obtain the number of climb plays */
        Number == 0.                                                  /* if it
            is 0, there are no more climb plays */

/* fixed plays to used in test phase, no longer needed for game to run
   */
testPlay :- playFirstPiece ,
        playPiece(2, 4, 2, 11, 10, n, _) ,
        playPiece(3, 3, 1, 12, 10, e, _) ,
        playPiece(4, 7, 2, 10, 10, n, _) ,
        playPiece(0, 5, 2, 11, 12, w, _) ,
        playPiece(2, 3, 1, 11, 10, s, _) ,
        playPiece(1, 1, 1, 9, 8, s, _) ,
```

```
        playPiece(3, 5, 2, 12, 11, n, _) ,
        playPiece(2, 5, 2, 11, 10, e, _) ,
        playPiece(6, 6, 2, 11, 13, s, _) .

/* fixed game start with distribution and plays used in test phase, no
   longer needed for game to run */
test :-
        testDistribute ,
        testPlay ,
        assert(player(1, 'Angie', 1)) ,
        assert(player(2, 'Nuno', 1)).

/***************/
/* game engine */
/***************/

/* predicate used to count the number of pieces of a given player */
numberPieces(I, NP, N1, N2, R) :-                                   /* R is
   the result output, NP, N1 and N2 should be 0 */
     N1 > 7 -> R is NP ;                                            /* if N1
         > 7, we have the result */
     (N2 > 7 ->                                                     /* if N2
         > 7, advance to the next piece */
      (N21 is N1 + 1 ,                                              /* which
          is N1 + 1 | N1 + 1*/
       N11 is N1 + 1 ,
       numberPieces(I, NP, N11, N21, R)) ;                          /*
          continue recursively */
      (piece(N1, N2, I, 0) ->                                       /*
         otherwise, check if the piece belongs to the player and has
         not been played */
       (NP1 is NP + 1 ,                                             /* if so
          , the number of pieces must be increased */
        N21 is N2 + 1,                                              /* we
           advance to the next piece */
        numberPieces(I, NP1, N1, N21, R)) ;                         /* and
           continue recursively */
       (N21 is N2 + 1 ,                                             /* if
          not, simply advance to the next piece */
        numberPieces(I, NP, N1, N21, R)))).                         /* and
           continue recursively, without incrementing the number of
           pieces */

/* predicate used to check if the game is over */
checkGameOver :-                                                    /* the
   name is misleading, fails if game over, succeeds otherwise */
     numberPieces(1, 0, 0, 0, R1) ,                                 /* get
        number of pieces of player 1 */
     (R1 == 0 ->                                                    /* if
        player 1 has 0 pieces */
      (printGameOver(1), fail) ;                                    /*
         player 1 has won, print that and fail */
      (numberPieces(2, 0, 0, 0, R2),                                /*
         otherwise, get number of pieces of player 2 */
       (R2 == 0 ->                                                  /* if
          player 2 has 0 pieces */
        (printGameOver(2) , fail) ;                                 /*
           player 2 has won, print that and fail */
        true))).                                                    /*
           otherwise the game continues */

/* predicate used to play the first piece in the game */
playFirstPiece :-                                                   /* the
```

```
    first piece is always 7 | 7 and must be played in the middle of the
    board */
        assert(halfPiece(9, 9, 1, 7, e)),
        assert(halfPiece(9, 10, 1, 7, w)),
        retract(piece(7, 7, 1, 0)) ,
        assert(piece(7, 7, 1, 1)).

/* predicate used to play the game */
playGame :-
        distributePieces(0, 0, 0, 0) ,                          /* first
            distribute the pieces among the players */
        playFirstPiece ,                                        /* then
            play the first piece */
        assert(turn(2)) ,                                       /* the
            next player is player 2 */
        playTurn.                                               /* play
            the next turn */

/* predicate used to play a turn */
playTurn :-                                                     /* when
    the player is human, more than one recursive call is needed to play a
     full turn */
        checkGameOver ->                                        /* check
            if the game is not over and continue if it is not */
        (turn(I) ,                                              /* check
            whose turn it is */
         player(I, _, T) ,                                      /*
            obtain the type of player whose turn it is */
         (T == 1 ->                                             /* if
            player is human */
          (printGame(I) ,                                       /* print
              board and pieces */
           getMove(I, N1, N2, X1, Y1, C1) ,                     /* ask
              the player for the next move */
           (playPiece(N1, N2, I, X1, Y1, C1, L) ->              /* try
              to play the piece */
            (nextPlayer(I, L) ,                                 /*
               determine who is the next player (this player's turn may
               not be over) */
             playTurn) ;                                        /*
                recursively resume playing with whomever is next */
            (write('Invalid movement.') , sleep(1) ,            /* if
               the move is not valid, say so */
             playTurn))) ;                                      /* and
                recursively resume with the same player */
          (T == 2 ->                                            /* if
             the player is the computer on easy mode */
           (I1 is 3 - I ,                                       /*
              obtain the number of the other player */
            printGame(I1) , nl ,                                /* and
               print their board and pieces */
            player(I1, _, T1) ,                                 /*
               compute also their type */
            (T1 > 1 ->                                          /* if
               they are not human, then the game is computer vs computer
                */
             getNewLine ;                                       /* wait
                for the person running the game to press enter */
             (write('Computer thinking...') , sleep(1))) ,      /* if
                the other player is human, tell them the computer is
                thinking */
            playRandom(I) ,                                     /* in
               any case, compute a sequence of random computer movements
                */
```

```
            changeTurn(I) , playTurn) ;                          /*
               change player turn and resume playing recursively */
           (I1 is 3 - I ,                                        /* if
              the computer is on hard mode, obtain the number of the
              other player */
            printGame(I1) , nl ,                                 /* and
               print their board and pieces */
            player(I1, _, T1) ,                                  /*
               compute also their type */
            (T1 > 1 ->                                           /* if
               they are not human, then the game is computer vs computer
                */
             getNewLine ;                                        /* wait
                for the person running the game to press enter */
             (write('Computer thinking...')) ,                   /* if
                the other player is human, tell them the computer is
                thinking */
            playBest(I),                                         /* in
               any case, compute a sequence of greedy computer movements
                */
            changeTurn(I) , playTurn)))) ; !.                    /*
               change player turn and resume playing recursively */

/* predicate used to ask the human player for the next move */
getMove(I, N1, N2, X1, Y1, C1) :-
        nl , getContinue ,                                       /* first
            check if the player wishes to continue playing or wants to
           save and exit */
        getN1(I, N1) ,                                           /* get
           the first number of the piece */
        getN2(I, N1, N2) ,                                       /* get
           the second number of the piece */
        getX1(X1) ,                                              /* get
           the line */
        getY1(Y1) ,                                              /* get
           the column */
        getC1(C1).                                               /* get
           the orientation */

/* predicate used to check if the player wishes to continue playing or
   wants to save and exit */
getContinue :-
        prompt(_, 'Continue - enter , Save and exit - s: ') ,    /* ask
           the question */
        getChar(C) ,                                             /* get
           the answer */
        (C == 's' -> save_game ; ! ).                            /* if
           answer is s, save and quit, otherwise continue playing */

/* predicate used to obtain the first number of the piece */
getN1(I, N1) :-
        prompt(_, 'Piece left number: ') ,                       /* ask
           for a number */
        getDigit(N1t) ,                                          /* get
           the single digit */
        (piece(N1t, _, I, 0) -> N1 is N1t ;                      /* if
           there is such a numbered piece, return the number */
         (write('You have no piece ') ,                          /*
            otherwise inform the player that there is */
          print(N1t) , write(' | ? .') , nl ,                    /* no
             such piece */
          getN1(I, N1))).                                        /* and
             ask for a new number */
```

```
    /* predicate used to obtain the second number of the piece */
310 getN2(I, N1, N2) :-                                          /* very
       similar to the one used to obtain the first number */
          prompt(_, 'Piece right number: ') ,
          getDigit(N2t) ,
          (piece(N1, N2t, I, 0) -> N2 is N2t ;
           (write('You have no piece ') ,
315         print(N1) , write(' | ') , print(N2t) , write(' .') , nl ,
            getN2(I, N1, N2))).

    /* predicate used to obtain the line of the board */
    getX1(X1) :-
320       prompt(_, 'Line for left number (1 - 18): ') ,         /* ask
             for the line */
          getDoubleDigit(X1t) ,                                  /* get
             possible double digit */
          (X1t < 1 ->                                            /* if it
              is below 1 */
           (write('Line number must be at least 1.') , nl ,      /*
              complain to the player */
            getX1(X1)) ;                                         /* and
              ask for a new number */
325        (X1t > 18 ->                                          /* if if
              is above 18 */
            (write('Line number must be at most 18.') , nl ,     /*
              complain to the player */
             getX1(X1)) ;                                        /* and
              ask for a new number */
            X1 is X1t)).                                         /*
              otherwise return the number */

330 /* predicate used to obtain the column of the board */
    getY1(Y1) :-                                                 /* very
       similar to the one used to obtain the line */
          prompt(_, 'Column for left number (1 - 18): ') ,
          getDoubleDigit(Y1t) ,
          (Y1t < 1 ->
335        (write('Column number must be at least 1.') , nl ,
            getY1(Y1)) ;
           (Y1t > 18 ->
            (write('Column number must be at most 18.') , nl ,
             getY1(Y1)) ;
340         Y1 is Y1t)).

    /* predicate used to obtain the orientation of the piece */
    getC1(C1) :-
          prompt(_, 'Cardinal of right number relative to left number (n,
             e, s, w): ') , /* ask for the cardinal */
345       getChar(C1t) ,                                         /* get
             the char */
          (member(C1t, [n, e, s, w]) -> copy_term(C1t, C1) ;     /* if it
              is valid, return it */
           (write('Cardinal must be one of: n, e, s, w.'), nl ,  /*
              otherwise complain */
            getC1(C1))).                                         /* and
              get a new one */

350 :- dynamic turn/1.
    /* predicate turn is used to ensure the next turn is given to the right
       player */

    /* predicate used to move the turn to the next player */
    changeTurn(I) :-
355       I1 is 3 - I ,                                          /* if
```

```
            the current player is I, the next is 3 - I */
        retract(turn(I)) ,                                          /*
            player I is no longer playing */
        assert(turn(I1)).                                           /* now
            it's player's 3 - I turn*/

/* predicate used to check if the turn of a human player has ended */
nextPlayer(I, L) :-
        L == 0 ->                                                   /* the
            turn ends after an expand movement */
        changeTurn(I) ; !.                                          /* that
            is, movement with level L = 0 */


/**************************/
/* artifical intelligence */
/**************************/

:- dynamic position/2.
/* a position on the board with its line and column */

/* predicate used to compute all possible board positions */
getPositions(X, Y) :-
        X > 18 -> ! ;                                               /* if
            the line is above 18, stop */
        (Y > 18 ->                                                  /* if
            the column is above 18 */
         (Y1 is 1 , X1 is X + 1 ,                                   /* move
             on to the first column of the next line */
          getPositions(X1, Y1)) ;                                   /* and
              proceed recursively */
         (assert(position(X, Y)),                                   /*
            otherwise, this is a valid position, add it to the database
            */
          Y1 is Y + 1,                                              /* and
              move on to the next cell */
          getPositions(X, Y1))).                                    /*
              calling the function recursively */

/* add all valid positions to the database so that findall can find
   possible movements */
:- getPositions(1, 1).

/* predicate used to check if a climb movement is valid */
/* this is very similar to checkPlay above, but only for climbs, and
   works with findall */
checkClimb(N1, N2, I, X1, Y1, C1) :-
        checkPlayerPiece(N1, N2, I) ,                               /* check
            if the piece belongs to the player and has not been played
            */
        getTopLevel(X1, Y1, L) ,                                    /*
            obtain the current level of the position on the board */
        getOtherHalf(X1, Y1, C1, X2, Y2, _) ,                       /*
            obtain the position of the other half of the piece */
        checkInsideBoard(X1, Y1, X2, Y2) ,                          /*
            verify that the whole piece is inside the board */
        checkLevelStable(L, X2, Y2) ,                               /*
            verify that the level of both halfs is the same */
        checkClimbNumbers(N1, X1, Y1, N2, X2, Y2, L).               /* check
            that the numbers are correct for climbing */

/* predicate used to check if an expand movement is valid */
/* this is very similar to checkPlay above, but only for expand, and
   works with findall */
```

```
checkExpand(N1, N2, I, X1, Y1, C1) :-
        checkPlayerPiece(N1, N2, I) ,                                   /* check
            if the piece belongs to the player and has not been played
           */
        position(X1, Y1),                                               /*
           verify that X1, Y1 is a valid board position */
        getOtherHalf(X1, Y1, C1, X2, Y2, C2) ,                          /*
           obtain the position of the other half of the piece */
        checkInsideBoard(X1, Y1, X2, Y2) ,                              /*
           verify that the whole piece is inside the board */
        checkLevelStable(0, X1, Y1),                                    /*
           verify that the level of both halfs is the 0 */
        checkLevelStable(0, X2, Y2) ,
        checkExpandOrthogonal(X1, Y1, C1, X2, Y2, C2).                  /* check
            that there is some other piece on the board orthogonal to
           this one */

/* predicate used to obtain a list of all possible climb movements for a
    player */
getClimbPlays(I, Plays) :-
        findall(play(N1, N2, I, X1, Y1, C1), checkClimb(N1, N2, I, X1,
           Y1, C1), Plays).

/* predicate used play a random climb movement */
randomClimbPlay(I) :-
        getClimbPlays(I, Plays) ,
        random_member(play(N1, N2, I, X1, Y1, C1), Plays) ,
        playPiece(N1, N2, I, X1, Y1, C1, _).

/* predicate used to obtain a list of all possible expand movements for
   a player */
getExpandPlays(I, Plays) :-
        findall(play(N1, N2, I, X1, Y1, C1), checkExpand(N1, N2, I, X1,
           Y1, C1), Plays).

/* predicate used play a random expand movement */
randomExpandPlay(I) :-
        getExpandPlays(I, Plays) ,
        random_member(play(N1, N2, I, X1, Y1, C1), Plays) ,
        playPiece(N1, N2, I, X1, Y1, C1, _).

/* predicate used play a random turn */
playRandom(I) :-
        randomClimbPlay(I) ->                                           /* while
            there are valid climbs */
        playRandom(I) ;                                                 /* do
           random climbs */
        randomExpandPlay(I).                                            /*
           afterward do one expand */

/* predicate used to evaluate the current situation of the game for a
   given player */
evaluateSituation(I, R) :-                                              /* R is
   the result, bigger R means better for player I */
        getNumberPlays(I, Rme) ,                                        /*
           compute the number of pieces player I can play */
        I1 is 3 - I ,                                                   /* I1 is
            the other player */
        getNumberPlays(I1 , Ryou) ,                                     /*
           compute the number of pieces the other player can play */
        R is Rme - Ryou.                                                /* the
           result is the difference */

/* predicate used to compute the maximum number of plays for a given
```

```
    player*/
getNumberPlays(I, R) :-
        getClimbPlays(I, Plays),                                    /* get
            list of all climbs */
        length(Plays, S) ,                                          /*
            compute maximum number of climbs */
        R is S + 1.                                                 /* the
            result is the number of climbs plus 1, because there is
            always 1 expand */

/* predicate used to play a piece without checking it is valid */
playPieceNoCheck(N1, N2, I, X1, Y1, C1, L) :-                       /* very
    similar to playPiece, but faster */
        getOtherHalf(X1, Y1, C1, X2, Y2, C2) ,
        getTopLevel(X1, Y1, L) , L1 is L + 1 ,
        assert(halfPiece(X1, Y1, L1, N1, C1)) ,
        assert(halfPiece(X2, Y2, L1, N2, C2)) ,
        retract(piece(N1, N2, I, 0)) ,
        assert(piece(N1, N2, I, 1)).

/* predicate used to remove a piece from the board */
removePiece(N1, N2, I, X1, Y1, C1, L) :-                            /* does
    the opposite of playPieceNoCheck */
        getOtherHalf(X1, Y1, C1, X2, Y2, C2) ,
        getTopLevel(X1, Y1, L) ,
        retract(halfPiece(X1, Y1, L, N1, C1)) ,
        retract(halfPiece(X2, Y2, L, N2, C2)) ,
        assert(piece(N1, N2, I, 0)) ,
        retract(piece(N1, N2, I, 1)).

/* predicate used to evaluate how good a given climb is */
evaluateClimb(N1, N2, I, X1, Y1, C1, R) :-                          /* R is
    the result, bigger R means better climb */
        checkClimb(N1, N2, I, X1, Y1, C1) ,                         /* first
            check if the climb is valid */
        playPieceNoCheck(N1, N2, I, X1, Y1, C1, _) ,                /* then
            play it */
        evaluateSituation(I, R) ,                                   /* then
            evaluate the current situation */
        removePiece(N1, N2, I, X1, Y1, C1, _).                      /* then
            remove the played piece */

/* predicate used to evaluate how good a given expand is */
evaluateExpand(N1, N2, I, X1, Y1, C1, R) :-                         /* very
    similar to evaluateClimb */
        checkExpand(N1, N2, I, X1, Y1, C1) ,
        playPieceNoCheck(N1, N2, I, X1, Y1, C1, _) ,
        evaluateSituation(I, R) ,
        removePiece(N1, N2, I, X1, Y1, C1, _).

/* predicate used to obtain a list of all climb plays evaluated */
evaluateClimbPlays(I, Plays) :-
        findall(play(N1, N2, I, X1, Y1, C1, R), evaluateClimb(N1, N2, I,
            X1, Y1, C1, R), Plays).

/* predicate used to obtain a list of all expand plays evaluated */
evaluateExpandPlays(I, Plays) :-
        findall(play(N1, N2, I, X1, Y1, C1, R), evaluateExpand(N1, N2, I
            , X1, Y1, C1, R), Plays).

/* predicate used to choose the best play from a list */
/* very similar to predicates used to choose maximum from a list */
bestPlay([], _) :- fail.                                            /* if
    the list is empty fail */
```

```
bestPlay([Play|Plays], MPlay) :-                                    /* go
    from 2 arguments to 3 */
        bestPlay(Plays, Play, MPlay).
bestPlay([], play(N1, N2, I, X1, Y1, C1, R), play(N1, N2, I, X1, Y1, C1,
     R)). /* between empty list and one play, choose one play */
bestPlay([play(N11, N21, I1, X11, Y11, C11, R1)|Plays] , play(N12, N22,
    I2, X12, Y12, C12, R2) , play(MN1, MN2, MI, MX1, MY1, MC1, MR)) :-
        (R1 > R2 ->

          /* check with play has bigger value */
         bestPlay(Plays, play(N11, N21, I1, X11, Y11, C11, R1), play(MN1
            , MN2, MI, MX1, MY1, MC1, MR)) ; /* proceed in the list with
             that play */
         bestPlay(Plays, play(N12, N22, I2, X12, Y12, C12, R2), play(MN1
            , MN2, MI, MX1, MY1, MC1, MR))).

/* predicate used to play one of the best climb plays available */
bestClimbPlay(I) :-
        evaluateClimbPlays(I, Plays) ,
        bestPlay(Plays, play(N1, N2, I, X1, Y1, C1, _)) ,
        playPiece(N1, N2, I, X1, Y1, C1, _).

/* predicate used to play one of the best expand plays available */
/* since there are usually many expand plays available, this is the
    slowest function taking a few seconds to finish */
bestExpandPlay(I) :-
        evaluateExpandPlays(I, Plays) ,
        bestPlay(Plays, play(N1, N2, I, X1, Y1, C1, _)) ,
        playPiece(N1, N2, I, X1, Y1, C1, _).

/* predicate used play a greedy turn, always choosing one of the best
    available plays */
playBest(I) :- bestClimbPlay(I) -> playBest(I) ; bestExpandPlay(I).
```