Universidade do Porto
Faculdade de Engenharia
FEUP

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

RUA ROBERTO FRIAS, SN, 4200-465 PORTO, PORTUGAL

Ni-Ju

PROGRAMAÇÃO EM LÓGICA

RELATÓRIO FINAL - TP1

Grupo: Ni-Ju_1
Ana Cláudia Fonseca Santos - 200700742
Ana Margarida Oliveira Pinheiro da Silva - 201505505

12 de Novembro de 2017

Resumo

Foi-nos proposto o desenvolvimento do jogo de tabuleiro Ni-Ju para a cadeira de Programação em Lógica, usando a linguagem Prolog.

A linguagem em si foi um desafio, pois é bastante diferente de todas as outras linguagens que havíamos aprendido até à época. Mesmo assim as dificuldades foram superadas e o nosso conhecimento cresceu.

O jogo em si também foi desafiante, logo no início com a visualização do tabuleiro. O facto de termos de desenhar o interior da peça provou-se difícil, mas forçou-nos a fazer funções complexas e com uso a recursividade. Ao longo do decorrer do trabalho, os maiores desafios foram o segundo nível de jogo e a verificação do estado de jogo. Os predicados para estes dois problemas são os mais complexos que fizemos, pois têm que percorrer o tabuleiro inteiro e, para cada peça, verificar as peças adjacentes.

Concluímos que o nosso projeto era difícil, tanto pelo jogo em si, como pela linguagem em que o tivemos que fazer. No entanto, foi uma experiência interessante e desafiante e os nossos conhecimentos em Prolog aumentaram consideravelmente.

Índice

1 Introdução

2 O Jogo Ni-Ju

2.1 História

O “Ni-ju”, que significa “20” em japonês, é um jogo de tabuleiro concebido por Néstor Romeral Andrés e foi lançado em 2016. É um jogo para dois jogadores no qual estes colocam as peças de jogo em padrões específicos. Para além disso, situa-se na categoria de estratégia abstrata, isto é, não tem tema, cada jogada é direta/mecânica e não contêm nenhum elemento de sorte/ocorrência aleatória.

2.2 Peças

Cada jogador tem 20 peças (figura 1) e cada peça tem um padrão diferente, daí o nome do jogo (20). Existem 70 padrões diferentes se incluírmos as rotações. Existem também 40 discos vermelhos.

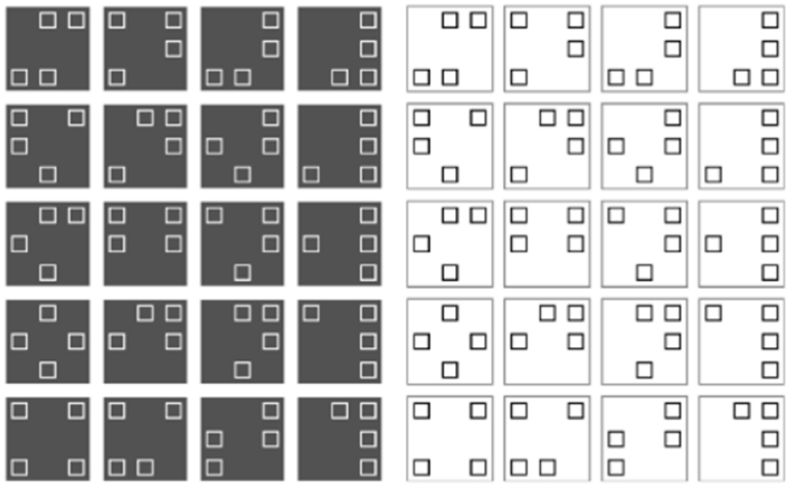


Figura 1: Peças do jogo

2.3 Objetivo

Um jogador ganha quando conseguir cercar uma das suas peças em jogo por pelo menos 4 da sua cor, em que o seu padrão aponta a direção dessas peças, como se verifica na figura 2.

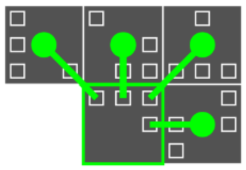
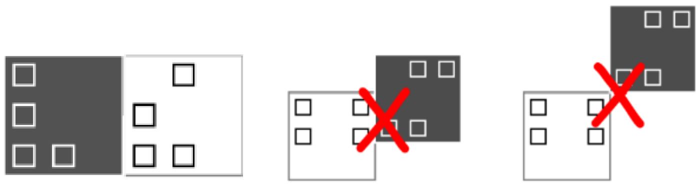


Figura 2: Como ganhar

2.4 Instruções

O "tabuleiro" começa vazio e cada jogador escolhe a sua cor (branco ou preto). A primeira jogada é realizada por aquele que tiver as peças brancas, sendo esta peça colocada numa posição aleatória sobre a mesa. As próximas jogadas tem como intuito alinhar as peças de forma a concretizar o objetivo já referido no ponto anterior, mas também deverão impedir o adversário de ganhar. As jogadas possíveis são sempre referentes às peças já colocadas (excepto a primeira), nas quais têm se ser obrigatoriamente alinhadas de forma horizontal e vertical (figura 5).



(a) Legal **(b)** Ilegal

Figura 3: Exemplo de jogadas

Se acordado entre os jogadores, existe a possibilidade de ser colocado um pequeno disco vermelho sobre as peças em jogo que já não permitem desencadear a jogada da vitória. Adicionado assim clareza ao estado atual de jogo. Na fig. 5 apresentamos um exemplo de quando colocar um disco vermelho, neste caso a peça branca contém um quadrado do padrão "bloqueado" por uma peça preta, não podendo ser ligado nem na horizontal nem na diagonal a outro quadrado de uma peça branca.

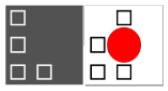
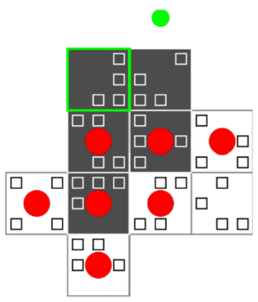


Figura 4: Exemplo de colocação do disco vermelho

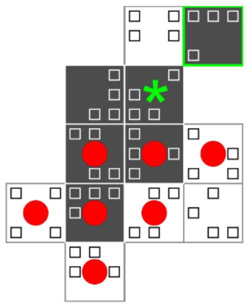
Se as peças dos jogadores se esgotarem sem que nenhum tenha obtido sucesso estes podem continuar, jogando as peças que já se encontravam no tabuleiro, desde que estas tenha pelo menos um dos lados "livres", ou seja, sem peças. Se houver, é necessário remover os discos vermelhos.

2.5 Fim do jogo

Exemplo de uma jogada da figura 5a: o jogador das peças pretas acaba de colocar a peça destacada a verde, conseguindo ligar cada um dos quadrados desta peça as outras pretas já no tabuleiro exceptuando a do canto superior. O jogador das peças brancas tenta bloquear esta jogada colocando uma peça no local do ponto verde. A peça preta em questão é bloqueada porém o jogador das peças pretas na sua vez de jogar consegue chegar à vitória através da peça com o asterisco verde.



(a) Parte 1



(b) Parte 2

Figura 5: Exemplo de jogadas

3 Lógica do jogo

Descrever o projeto e implementação da lógica do jogo em Prolog, incluindo a forma de representação do estado do tabuleiro e sua visualização, execução de movimentos, verificação do cumprimento das regras do jogo,determinação do final do jogo e cálculo das jogadas a realizar pelo computador utilizando diversos níveis de jogo. Sugere-se a estruturação desta secção da seguinte forma:

3.1 Representação do Estado do Jogo

3.1.1 Cada Peça

Informação referente a cada peça:

$$\text{Peça} \begin{cases} \text{Código} \begin{cases} \text{a} \\ \text{b} \\ \text{...} \\ \text{t} \end{cases} \\ \text{Posição} \begin{cases} \text{0 - 0º} \\ \text{1 - 90º} \\ \text{2 - 180º} \\ \text{3 - 270º} \end{cases} \\ \text{Cor} \begin{cases} \text{0 - Preta} \\ \text{1 - Branca} \end{cases} \\ \text{Validade} \begin{cases} \text{0 - Válida} \\ \text{1 - Inválida} \end{cases} \end{cases}$$

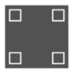


Figura 6: Peça preta com código T

Representação da peça t, na orientação inicial e válida (fig. 6) :

```
piece([1 0 1], [0 0 0], [1 0 1]).
```

Informação da respectiva peça a guardar:

```
pieceInfo([t,0,0,0]).
```

3.1.2 Tabuleiro

Estado inicial do tabuleiro:

```
[[nil]].
```

Informação da respectiva peça a guardar:

```
[[nil, nil,       nil],
 [nil, [t,0,1,0], nil],
 [nil, nil,       nil]].
```

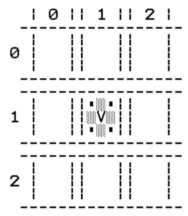


Figura 7: Aspeto da board após o print.

Possível estado intermédio do tabuleiro.

```
[[nil, nil,       nil,       nil],
 [nil, [t,0,1,0], [a,1,0,0], nil],
 [nil, [k,2,1,0], nil,       nil],
 [nil, nil,       nil,       nil]];
```

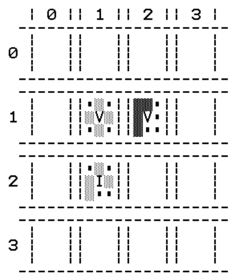


Figura 8: Aspeto da board após o print.

Possível estado final do tabuleiro.

```
[[nil, nil,     nil,        nil,        nil,        nil,        nil],
 [nil, nil,     nil,        nil,        [s, 0, 0, 0],nil,        nil],
 [nil,[j,0,1,0],nil,        nil,        [j, 0, 0, 0],nil,        nil],
 [nil,[i,0,1,0],[p, 0, 1, 0],[b, 0, 1, 1],[t, 0, 0, 1],[p, 0, 0, 0],nil],
 [nil, nil,     nil,        [o, 0, 1, 0],nil,        ,nil,        nil],
 [nil, nil,     nil,        mil,        nil,        nil,        nil]];
```

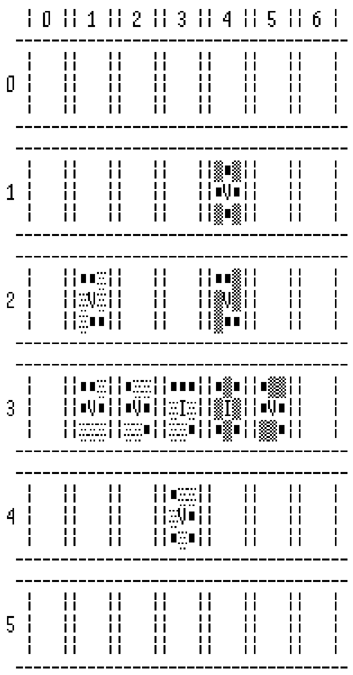


Figura 9: Representação de um fim de jogo onde a peça branca colocada na linha 3 e na coluna 2 ganhou.

3.1.3 Peças disponíveis

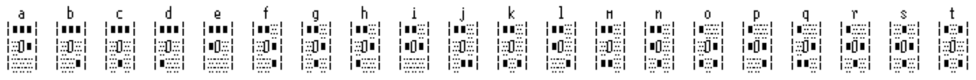


Figura 10: Print das peças dispoviveis do jogador de peças brancas.

3.2 Visualização do tabuleiro

Para a visualização dos tabuleiros representados nas figuras 7, 8 e 14 foi usado o código no anexo 7.7, utilizando para isso os seguintes predicados:

```
rotatePattern(+Row, +Pattern, -Pattern).
getPieceInfo(+Piece, -Color, -Valid).
getPiecePattern(+PieceNum, +Pattern, -NewPattern).
printEachSymbo(+Pattern, +Col, -Color, -Valid).
printPieceSymbols(+PieceNUm, +Pattern, -Color, -Valid).
printPiece(+Piece, +Pattern, +PieceNum).
getPiece(+Piece, -PieceRotation).
printRowPieces(+Piece, +Num, +PieceNum).
printRow(+Num, +PieceNum, +Row, +RowNumber).
printBoard(+Board, -RowCount).
printTopNumbers(+Count, -ColumnsNum).
printSeparator(+ColsNUm).
printBoardMain(+Board).
```

3.3 Visualização das peças disponiveis

Para a visualização das peças disponíveis de cada jogador (exemplo representado na figura 10) foi usado o código no anexo 7.7, utilizando para isso os seguintes predicados:

```
printEachSymbol(+Piece, +Color, +Valid).
printAvailablePiecesRow(+PieceRow, +AvailablePieces).
printAvailablePiecesAux(+PieceRow, +AvailablePieces).
prepareLegendsPieces(+Letters).
printAvailablePieces(+PieceRow, +AvailablePieces).
```

3.4 Lista de Jogadas Válidas

A lista de jogadas válidas é obtida pela analise do tabuleiro a cada jogada, onde uma jogada é válida quando a posição escolhida está livre e alguma posição adjacente a esta está ocupada. Verificado pelo código presente no anexo 7.2, utilizando para isso os seguintes predicados:

```
% Get a set of all valid moves.
getValidMoves(+Board, -ValidMoves).
% Check moves - ensures that the position is free and next to another piece.
validMove(+Board, -NumRow, -NumCol).
```

3.5 Avaliação do Tabuleiro

No segundo nível de jogo, em cada jogada do computador, é necessário calcular a pontuação de cada jogada possível, permitindo assim escolher a melhor. Esse cálculo é feito nos predicados ***getPositionScore/7***, ***checkAroundScore/9*** e ***checkRowScore/10***. Estes predicados retornam a pontuação de cada jogada válida para uma lista, que depois irá ser ordenada. A pontuação de cada jogada possível depende das peças adjacentes a esta, existindo assim 3 situações específicas:

1. Pontuação aumenta 1 valor por cada peça que tenha a mesma cor que a peça que será jogada. No exemplo da Figura 11, a peça da direita, quando calculada a sua pontuação, antes de ser coloca no tabuleiro, teria sofrido um incremento de 1 pois tem uma peça ao seu lado com a mesma cor.

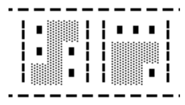


Figura 11: Exemplo de incremento de 1 na pontuação de uma peça

2. Pontuação aumenta 2 valores por cada peça que seja da mesma cor e que fique beneficiada pela colocação de uma peça à sua beira. No exemplo da Figura 12, a peça colocada na linha de cima teria uma pontuação de 4 pois beneficia as peças na linha abaixo com a sua colocação.

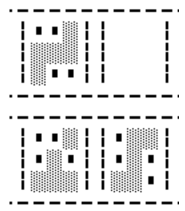


Figura 12: Exemplo de incremento de 2 na pontuação de uma peça

3. Pontuação não é incrementada pois não existem peças da mesma cor à sua volta naquela posição.

3.6 Final de jogo

Ao fim de cada jogada é verificado o estado atual de jogo, assim é possível saber se houve empate ou se algum dos jogadores ganhou e assim o jogo deve terminar. Para isso existe os seguintes predicados ***checkGameEnd/3*** e ***verifyDraw/3***. O primeiro recebe o tabuleiro atual, cria a lista das peças inválidas para fazer uma atualização no tabuleiro e verifica se houve vitória. O segundo é reponsavel por verificar se os jogadores ficaram sem peças, ou seja, empataram e neste caso o processo de jogada é alterado.

```
checkGameEnd(+Board, -NewInvalidPieces, -GameEnd).
vertifyDraw(+PiecesWhite, +PiecesBlack, -NewDraw).
```

3.7 Jogada do Computador

A jogo disponibiliza os modos de jogar ***Humano vs Computador*** e ***Computador vs Computador*** com dois níveis de dificuldade (fig. 13).

No primeiro nível as jogadas são calculadas de forma aleatória, os seguintes predicados são responsáveis por retornar a peça, a posição e a orientação da peça que vai ser jogada. O predicado ***getValidPosition/2*** utiliza o predicado ***getValidMoves/2*** de forma a garantir uma posição válida no jogo.

```
computerInput(+Board, +Pieces, -Letter, -Rotation).
computerInput(+Board, +Pieces, +AdversaryPieces, +ColorPlayer, -Letter,
 ↪  -Rotation, -NumRow, -NumCol, +Level).
computerInputMove(+Board, -SourceRow, -SourceColumn, -Rotation, -DestRow,
 ↪  -DestColumn, +ColorPlayer, +Level).

getPieceLetter(+Pieces, -Letter).
getRotation(-Rotation).
getValidPosition(+Board, -NumRow, -NumCol).
```

No segundo nível as jogadas são calculadas pela seguinte ordem de preferência:

1. Se o jogador estiver a uma jogada de ganhar, faz essa jogada se for possível.

2. Se o adversário estiver a uma jogada de ganhar, este joga nessa posição uma peça aleatória das peças que tem disponíveis, de forma a bloquear o adversário.

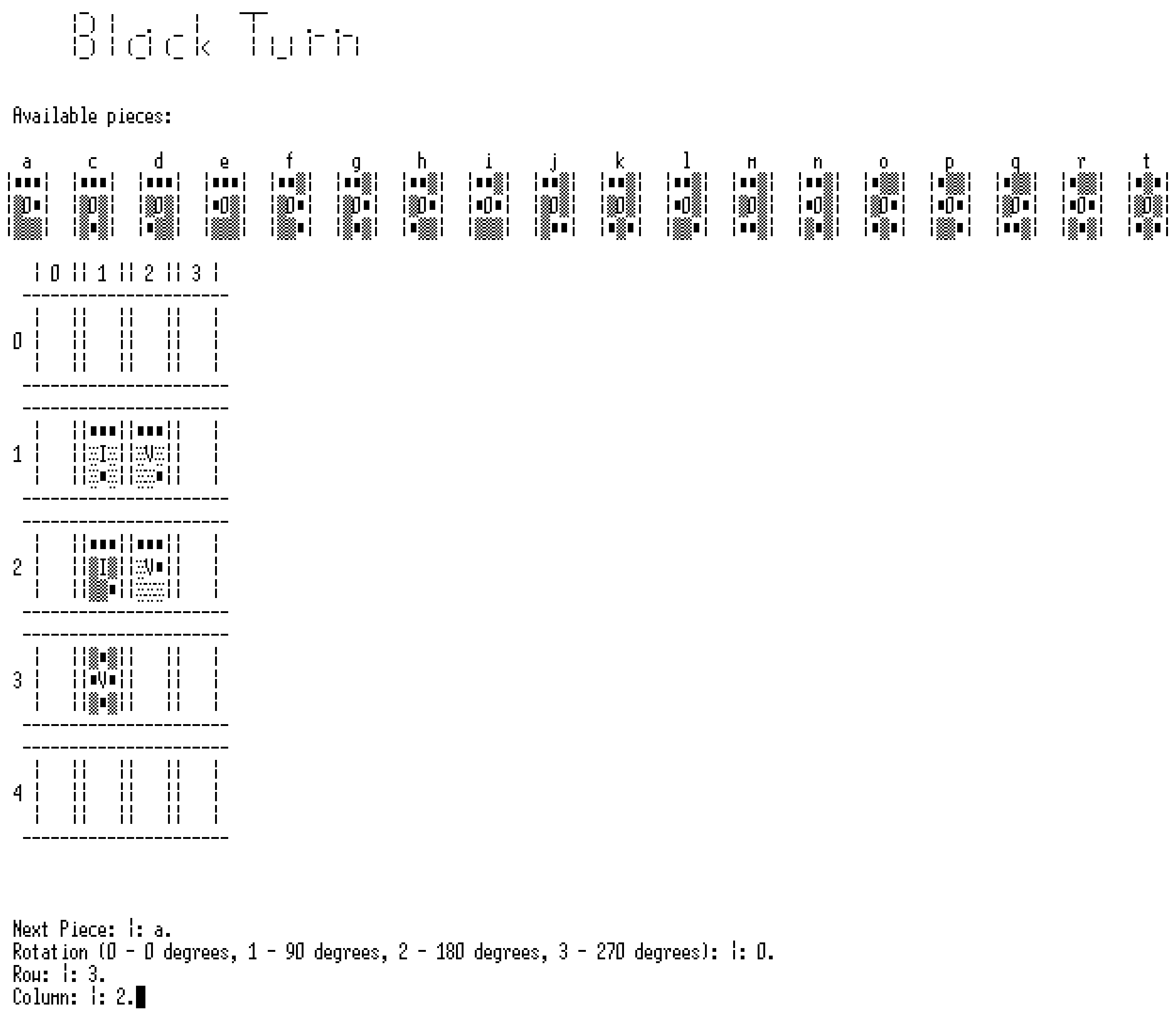
3. Se nenhuma das situações anteriores acontecer, é então calculado a melhor jogada possível de forma ao jogador estar mais próximo da vitória. Para este cálculo é utilizado a avaliação do tabuleiro.

```
computerInput(+Board, +Pieces, -Letter, -Rotation).
computerInput(+Board, +Pieces, +AdversaryPieces, +ColorPlayer, -Letter,
 ↪  -Rotation, -NumRow, -NumCol, +Level).
computerInputMove(+Board, -SourceRow, -SourceColumn, -Rotation, -DestRow,
 ↪  -DestCol, -ColorPlayer, +Level).
```

4 Interface com o Utilizador

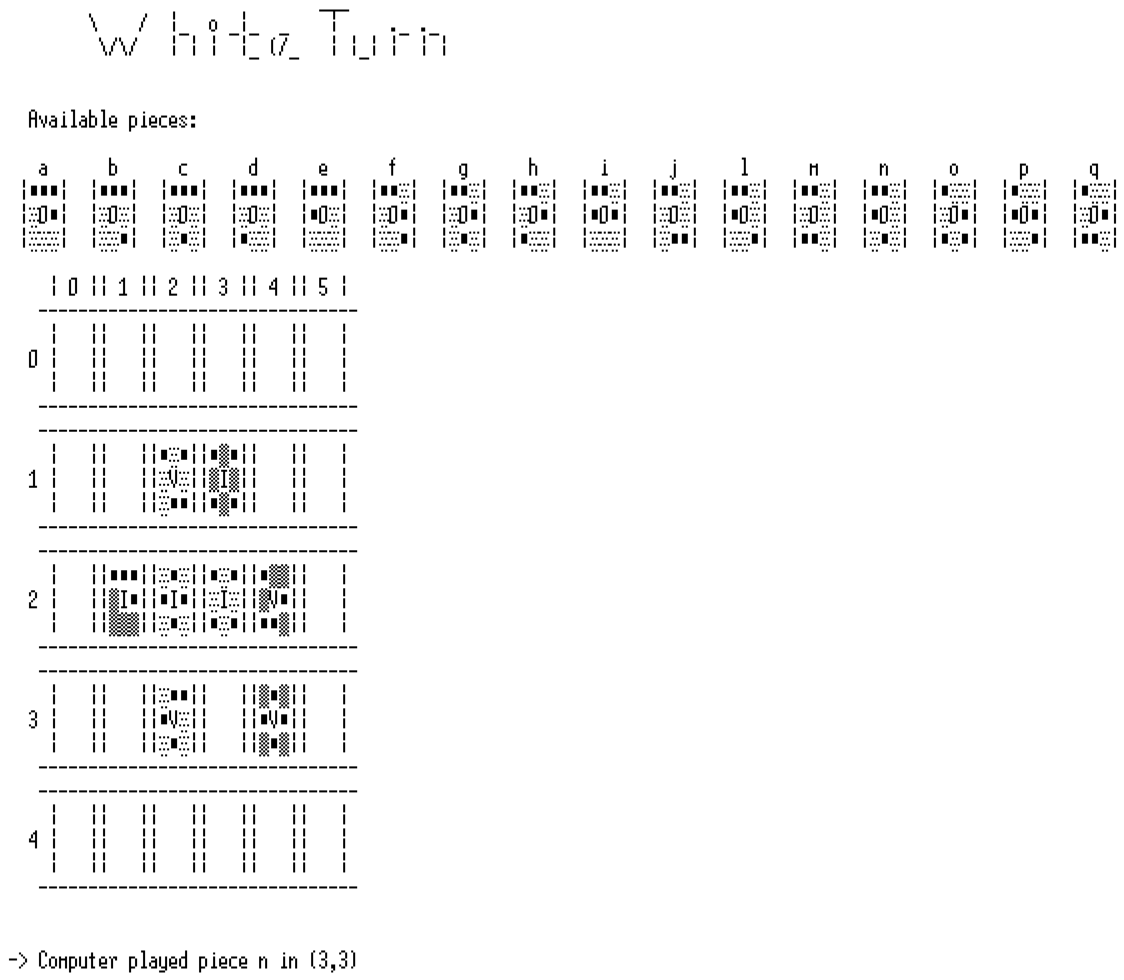
A interface da linha de comandos tem como objetivo simplificar a comunicação entre o utilizador e jogo. O menu permite escolher três modos de jogo, havendo dois níveis de jogo diferentes com o computador como jogador.

Figura 13: Menu mostrado ao utilizador



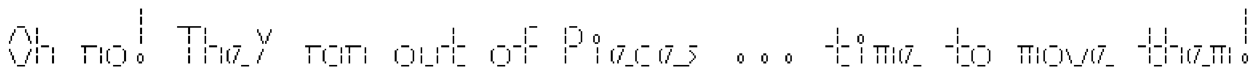
```
Black Turn

Available pieces:

 a    c    d    e    f    g    h    i    j    k    l    m    n    o    p    q    r    t

  | 0 || 1 || 2 || 3 |
 ---------------------
0
 ---------------------
 ---------------------
1          I    V
 ---------------------
 ---------------------
2          I    V
 ---------------------
 ---------------------
3          V
 ---------------------
 ---------------------
4
 ---------------------

Next Piece: |: a.
Rotation (0 - 0 degrees, 1 - 90 degrees, 2 - 180 degrees, 3 - 270 degrees): |: 0.
Row: |: 3.
Column: |: 2.
```

Figura 14: Estado intermédio do jogo no modo Humano vs Humano



```
White Turn

Available pieces:

 a    b    c    d    e    f    g    h    i    j    l    m    n    o    p    q

  | 0 || 1 || 2 || 3 || 4 || 5 |
0
1
2
3
4

-> Computer played piece n in (3,3)
```

Figura 15: Estado intermédio do jogo no modo Computador vs Computador



```
Oh no! They ran out of Pieces ... time to move them!
```

Figura 16: Informação de empate

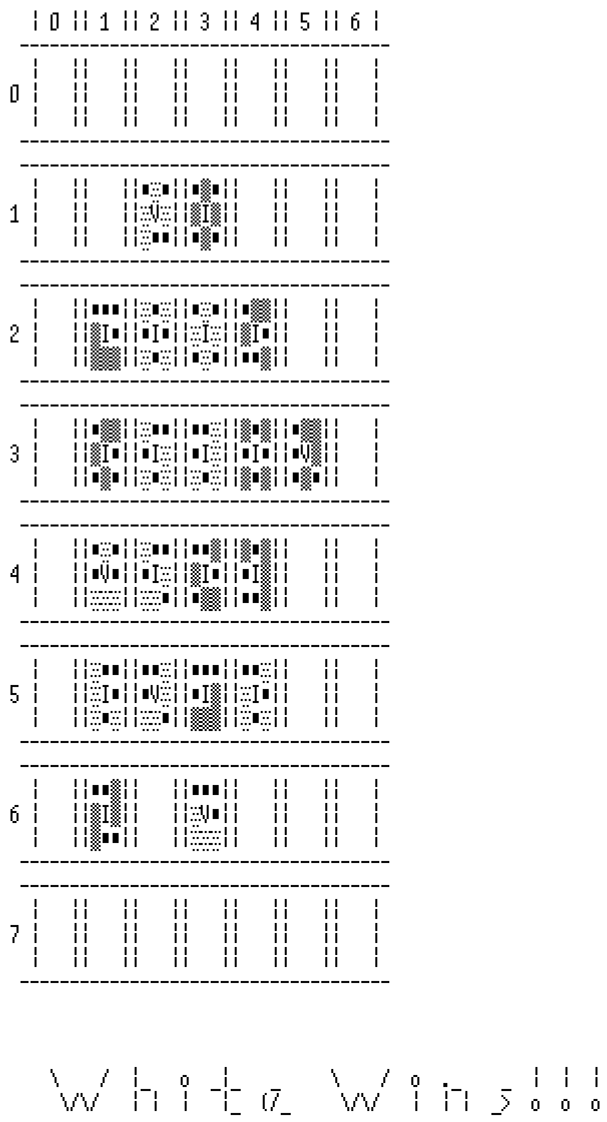


Figura 17: Informação de vitória

5 Conclusão

A elaboração deste projeto consolidou os conhecimentos adquiridos durante as aulas teóricas e práticas e foi um desafio interessante em termos de programação. Prolog é uma linguagem diferente das linguagens que aprendemos durante o curso e, apesar de apresentar bastantes desafios, permite-nos ter um raciocínio mais focado em lógica.

A conclusão deste trabalho deixa-nos com um maior conhecimento da linguagem Prolog e com uma experiência positiva de iniciação a Inteligência Artificial, para além de um jogo interessante e divertido.

6 Referências

[1] W. Clocksin. *Programming in Prolog.* W. Clocksin, Springer, 5ª edição, 2003.

[2] *SICStus prolog.* [ONLINE] Available at: https://sicstus.sics.se . [Accessed 11 November 2017].

[3] *Swi-prolog.*[ONLINE] Available at: http://www.swi-prolog.org/. [Accessed 29 November 2017].

7 Anexos

7.1 main.pl

```
:-include('utils.pl').


/**
* Game cycle - Human vs Human mode
**/
% First move - White player
game2Players(Board, PiecesWhite, PiecesBlack, Draw, GameEnd) :-
        clearScreen,
        ColorPlayer = 1,
        printInfoColor(ColorPlayer), sleep(1),
        printAvailablePieces(0, [ColorPlayer, PiecesWhite]), sleep(1),
        askInput(Board, PiecesWhite, Letter, Rotation),
        addPiece(Board, Letter, ColorPlayer, Rotation, NewBoard),
        removePiecePlayed(PiecesWhite, Letter, NewPiecesWhite),
        NewColorPlayer is ColorPlayer - 1,
        game2Players(NewBoard, NewPiecesWhite, PiecesBlack,
         ↪  NewColorPlayer, Draw, GameEnd).

% Following movements - White player
game2Players(Board, PiecesWhite, PiecesBlack, ColorPlayer, Draw, GameEnd)
 ↪  :-
        clearScreen,
        GameEnd \== 1, Draw == 0, ColorPlayer == 1, !,
        printInfoGame(Board, ColorPlayer, PiecesWhite),
        askInput(Board, PiecesWhite, Letter, Rotation, NumRow, NumCol),
        addPiece(Board, NumRow, NumCol, Letter, ColorPlayer, Rotation,
         ↪  NewBoardAux),
        removePiecePlayed(PiecesWhite, Letter, NewPiecesWhite),
        NewColorPlayer is ColorPlayer - 1,
        checkGameEnd(NewBoardAux, NewInvalidPieces, NewGameEnd),
        updateBoard(NewInvalidPieces, NewBoardAux, NewBoard),
        vertifyDraw(NewPiecesWhite, PiecesBlack, NewDraw),
        game2Players(NewBoard, NewPiecesWhite, PiecesBlack,
         ↪  NewColorPlayer, NewDraw, NewGameEnd).

% Following movements - Black player
game2Players(Board, PiecesWhite, PiecesBlack, ColorPlayer, Draw, GameEnd)
 ↪  :-
        clearScreen,
        GameEnd \== 1, Draw == 0,
        printInfoGame(Board, ColorPlayer, PiecesBlack),
        askInput(Board, PiecesBlack, Letter, Rotation, NumRow, NumCol),
        addPiece(Board, NumRow, NumCol, Letter, ColorPlayer, Rotation,
         ↪  NewBoardAux),
        removePiecePlayed(PiecesBlack, Letter, NewPiecesBlack),
        NewColorPlayer is ColorPlayer + 1,
        checkGameEnd(NewBoardAux, NewInvalidPieces, NewGameEnd),
        updateBoard(NewInvalidPieces, NewBoardAux, NewBoard),
        vertifyDraw(PiecesWhite, NewPiecesBlack, NewDraw),
        game2Players(NewBoard, PiecesWhite, NewPiecesBlack,
         ↪  NewColorPlayer, NewDraw, NewGameEnd).


% After draw moves
game2Players(Board, _PiecesWhite, _PiecesBlack, ColorPlayer, 1, GameEnd)
 ↪  :-
        printInfoDraw,
        game2Players(Board, ColorPlayer, GameEnd).

% Game cycle to move pieces
game2Players(Board, ColorPlayer, GameEnd) :-
        clearScreen, GameEnd \== 1,
        printInfoGame(Board, ColorPlayer),
        askInputMove(Board, SourceRow, SourceColumn, Rotation, DestRow,
         ↪  DestColumn, ColorPlayer),
        movePiece(Board, SourceRow, SourceColumn, Rotation, DestRow,
         ↪  DestColumn, NewBoardAux),
        NewColorPlayer is mod((ColorPlayer + 1), 2),
        checkGameEnd(NewBoardAux, NewInvalidPieces, NewGameEnd),
        updateBoard(NewInvalidPieces, NewBoardAux, NewBoard),
        game2Players(NewBoard, NewColorPlayer, NewGameEnd).

% End Game
game2Players(Board, ColorPlayer, 1)  :-
        printBoardMain(Board),
        WinColorPlayer is mod((ColorPlayer + 1), 2),
        getColorPlayer(WinColorPlayer, Color),
        printInfoWinGame(Color).
game2Players(Board, _PiecesWhite, _PiecesBlack, ColorPlayer, _Draw, 1)  :-
        printBoardMain(Board),
        WinColorPlayer is mod((ColorPlayer + 1), 2),
        getColorPlayer(WinColorPlayer, Color),
        printInfoWinGame(Color).
```

```
/**
* Game cycle - Human vs Computer mode
**/
% First move - Human player (white pieces).
gameHumanVsComputer(Board, Pieces, PiecesBlack, Draw, GameEnd, Level) :-
        clearScreen,
        Draw == 0, ColorPlayer = 1,
        printInfoColor(ColorPlayer), sleep(1),
        printAvailablePieces(0, [ColorPlayer, Pieces]), sleep(1),
        askInput(Board, Pieces, Letter, Rotation),
        addPiece(Board, Letter, ColorPlayer, Rotation, NewBoard),
         ↪  sleep(1),
        removePiecePlayed(Pieces, Letter, NewListAvailablePieces),
        NewColorPlayer is ColorPlayer - 1,
        gameHumanVsComputer(NewBoard, NewListAvailablePieces, PiecesBlack,
         ↪  NewColorPlayer, Draw, GameEnd, Level).

% Following movements - Human player (white pieces).
gameHumanVsComputer(Board, PiecesWhite, PiecesBlack, ColorPlayer, Draw,
 ↪  GameEnd, Level) :-
        clearScreen,
        GameEnd \== 1, Draw == 0, ColorPlayer == 1, !,
        printInfoGame(Board, ColorPlayer, PiecesWhite),
        askInput(Board, PiecesWhite, Letter, Rotation, NumRow, NumCol),
        addPiece(Board, NumRow, NumCol, Letter, ColorPlayer, Rotation,
         ↪  NewBoardAux), sleep(1),
        removePiecePlayed(PiecesWhite, Letter, NewPiecesWhite),
        NewColorPlayer is ColorPlayer - 1,
        checkGameEnd(NewBoardAux, NewInvalidPieces, NewGameEnd),
        updateBoard(NewInvalidPieces, NewBoardAux, NewBoard),
        vertifyDraw(NewPiecesWhite, PiecesBlack, NewDraw),
        gameHumanVsComputer(NewBoard, NewPiecesWhite, PiecesBlack,
         ↪  NewColorPlayer, NewDraw, NewGameEnd, Level).

% Following movements - Computer player (black pieces).
gameHumanVsComputer(Board, PiecesWhite, PiecesBlack, ColorPlayer, Draw,
 ↪  GameEnd, Level) :-
        clearScreen,
        GameEnd \== 1, Draw == 0, ColorPlayer == 0, !,
        printInfoGame(Board, ColorPlayer, PiecesBlack),
        computerInput(Board, PiecesWhite, PiecesBlack, ColorPlayer,
         ↪  Letter, Rotation, NumRow, NumCol, Level),
        addPiece(Board, NumRow, NumCol, Letter, ColorPlayer, Rotation,
         ↪  NewBoardAux), sleep(1),
        removePiecePlayed(PiecesBlack, Letter, NewPiecesBlack),
        NewColorPlayer is ColorPlayer + 1,
        checkGameEnd(NewBoardAux, NewInvalidPieces, NewGameEnd),
        updateBoard(NewInvalidPieces, NewBoardAux, NewBoard),
        vertifyDraw(PiecesWhite, NewPiecesBlack, NewDraw),
        gameHumanVsComputer(NewBoard, PiecesWhite, NewPiecesBlack,
         ↪  NewColorPlayer, NewDraw, NewGameEnd, Level).

% After Draw
gameHumanVsComputer(Board, _PiecesWhite, _PiecesBlack, ColorPlayer, 1,
 ↪  GameEnd, Level)  :-
        printInfoDraw,
        gameHumanVsComputer(Board, ColorPlayer, GameEnd, Level).

% Game cycle to move pieces        - human player
gameHumanVsComputer(Board, ColorPlayer, GameEnd, Level) :-
        clearScreen,
        GameEnd \== 1, ColorPlayer == 1, !,
        printInfoGame(Board, ColorPlayer),
        askInputMove(Board, SourceRow, SourceColumn, Rotation, DestRow,
         ↪  DestColumn, ColorPlayer),
        movePiece(Board, SourceRow, SourceColumn, Rotation, DestRow,
         ↪  DestColumn, NewBoardAux),
        NewColorPlayer is ColorPlayer - 1,
        checkGameEnd(NewBoardAux, NewInvalidPieces, NewGameEnd),
        updateBoard(NewInvalidPieces, NewBoardAux, NewBoard),
        gameHumanVsComputer(NewBoard, NewColorPlayer, NewGameEnd,Level).

% Game cycle to move pieces        - computer player
gameHumanVsComputer(Board, ColorPlayer, GameEnd, Level) :-
        clearScreen,
        GameEnd \== 1, ColorPlayer == 0, !,
        printInfoGame(Board, ColorPlayer),
        computerInputMove(Board, SourceRow, SourceColumn, Rotation,
         ↪  DestRow, DestColumn, ColorPlayer, Level),
        movePiece(Board, SourceRow, SourceColumn, Rotation, DestRow,
         ↪  DestColumn, NewBoardAux),
        NewColorPlayer is ColorPlayer + 1,
        checkGameEnd(NewBoardAux, NewInvalidPieces, NewGameEnd),
        updateBoard(NewInvalidPieces, NewBoardAux, NewBoard),
        gameHumanVsComputer(NewBoard, NewColorPlayer, GameEnd, NewGameEnd,
         ↪  Level).
```

```
% End Game
gameHumanVsComputer(Board, _PiecesWhite, _PiecesBlack, ColorPlayer, _Draw,
 ↪  1, _Level)  :-
        printBoardMain(Board),
        WinColorPlayer is mod((ColorPlayer + 1), 2),
        getTypePlayer(WinColorPlayer, Type),
        printInfoWinGame2(Type).
gameHumanVsComputer(Board, ColorPlayer, 1, _Level)  :-
        printBoardMain(Board),
        WinColorPlayer is mod((ColorPlayer + 1), 2),
        getTypePlayer(WinColorPlayer, Type),
        printInfoWinGame2(Type).


/**
* Game cycle - Computer vs Computer mode
**/
% First move - white player.
gameComputerVsComputer(Board, PiecesWhite, PiecesBlack, Draw, GameEnd,
 ↪  Level) :-
        clearScreen,
        ColorPlayer = 1,
        printInfoColor(ColorPlayer), sleep(1),
        printAvailablePieces(0, [ColorPlayer, PiecesWhite]), sleep(1),
        computerInput(Board, PiecesWhite, Letter, Rotation),
        addPiece(Board, Letter, ColorPlayer, Rotation, NewBoard),
         ↪  sleep(1),
        printBoardMain(NewBoard), nl, sleep(3),
        removePiecePlayed(PiecesWhite, Letter, NewPiecesWhite),
        NewColorPlayer is ColorPlayer - 1,
        gameComputerVsComputer(NewBoard, NewPiecesWhite, PiecesBlack,
         ↪  NewColorPlayer, Draw, GameEnd, Level).

% Following movements - White player
gameComputerVsComputer(Board, PiecesWhite, PiecesBlack, ColorPlayer, Draw,
 ↪  GameEnd, Level) :-
        clearScreen,
        GameEnd \== 1, Draw == 0, ColorPlayer == 1, !,
        printInfoGame(Board, ColorPlayer, PiecesWhite),
        computerInput(Board, PiecesWhite, PiecesBlack, ColorPlayer,
         ↪  Letter, Rotation, NumRow, NumCol, Level),
        addPiece(Board, NumRow, NumCol, Letter, ColorPlayer, Rotation,
         ↪  NewBoardAux), sleep(1),
        removePiecePlayed(PiecesWhite, Letter, NewPiecesWhite),
        NewColorPlayer is ColorPlayer - 1,
        checkGameEnd(NewBoardAux, NewInvalidPieces, NewGameEnd),
        updateBoard(NewInvalidPieces, NewBoardAux, NewBoard),
        vertifyDraw(NewPiecesWhite, PiecesBlack, NewDraw),
        gameComputerVsComputer(NewBoard, NewPiecesWhite, PiecesBlack,
         ↪  NewColorPlayer, NewDraw, NewGameEnd, Level).

% Following movements - Black player
gameComputerVsComputer(Board, PiecesWhite, PiecesBlack, ColorPlayer, Draw,
 ↪  GameEnd, Level) :-
        clearScreen,
        GameEnd \== 1, Draw == 0, ColorPlayer == 0, !,
        printInfoGame(Board, ColorPlayer, PiecesBlack),
        computerInput(Board, PiecesBlack, PiecesWhite, ColorPlayer,
         ↪  Letter, Rotation, NumRow, NumCol, Level),
        addPiece(Board, NumRow, NumCol, Letter, ColorPlayer, Rotation,
         ↪  NewBoardAux), sleep(1),
        removePiecePlayed(PiecesBlack, Letter, NewPiecesBlack),
        NewColorPlayer is ColorPlayer + 1, !,
        checkGameEnd(NewBoardAux, NewInvalidPieces, NewGameEnd),
        updateBoard(NewInvalidPieces, NewBoardAux, NewBoard),
        vertifyDraw(PiecesWhite, NewPiecesBlack, NewDraw),
        gameComputerVsComputer(NewBoard, PiecesWhite, NewPiecesBlack,
         ↪  NewColorPlayer, NewDraw, NewGameEnd, Level).

% After Draw
gameComputerVsComputer(Board, _PiecesWhite, _PiecesBlack, ColorPlayer, 1,
 ↪  GameEnd, Level)  :-
        printInfoDraw,
        gameComputerVsComputer(Board, ColorPlayer, GameEnd, Level).

% Game cycle to move pieces
gameComputerVsComputer(Board, ColorPlayer, GameEnd, Level) :-
        clearScreen,
        GameEnd \== 1,
        printInfoGame(Board, ColorPlayer),
        computerInputMove(Board, SourceRow, SourceColumn, Rotation,
         ↪  DestRow, DestColumn, ColorPlayer, Level),
        movePiece(Board, SourceRow, SourceColumn, Rotation, DestRow,
         ↪  DestColumn, NewBoardAux),
        NewColorPlayer is mod((ColorPlayer + 1), 2),
        checkGameEnd(NewBoardAux, NewInvalidPieces, NewGameEnd),
        updateBoard(NewInvalidPieces, NewBoardAux, NewBoard),
        printBoardMain(NewBoard), nl, sleep(3),
        gameComputerVsComputer(NewBoard, NewColorPlayer,  NewGameEnd,
         ↪  Level).


% End Game
gameComputerVsComputer(Board, _PiecesWhite, _PiecesBlack, ColorPlayer,
 ↪  _Draw, 1, _Level)  :-
        printBoardMain(Board),
        WinColorPlayer is mod((ColorPlayer + 1), 2),
        getColorPlayer(WinColorPlayer, Color),
        printInfoWinGame(Color).
gameComputerVsComputer(Board, ColorPlayer, 1, _Level)  :-
        printBoardMain(Board),
        WinColorPlayer is mod((ColorPlayer + 1), 2),
        getColorPlayer(WinColorPlayer, Color),
        printInfoWinGame(Color).


/***************************
******** MAIN GAME *********
***************************/

% Init board
board1([nil]).

% Init pieces
piecesWhite([a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t]).
piecesBlack([a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t]).

% Options to menu
options([1, 2, 3, 4, 5, 6]).



ni_ju :-
now(X),
setrand(X),
printLoad,
clearScreen,
printMenuScreen,
options(Options),
askMenuInput(Options, Option),
board1(Board),
piecesBlack(PiecesBlack),
piecesWhite(PiecesWhite),
initGame(Option, Board, PiecesWhite, PiecesBlack).


initGame(Option, Board, PiecesWhite, PiecesBlack):-
Option == 1, !,
write('Option 1'), nl,
clearScreen,

game2Players(Board, PiecesWhite, PiecesBlack, 0, 0),
ni_ju.

initGame(Option, Board, PiecesWhite, PiecesBlack):-
Option == 2, !,
write('Option 2'), nl,
clearScreen,
gameHumanVsComputer(Board, PiecesWhite, PiecesBlack, 0, 0, 1),
ni_ju.

initGame(Option, Board, PiecesWhite, PiecesBlack):-
Option == 3, !,
write('Option 3'), nl,
clearScreen,
gameHumanVsComputer(Board, PiecesWhite, PiecesBlack, 0, 0, 2),
ni_ju.

initGame(Option, Board, PiecesWhite, PiecesBlack):-
Option == 4, !,
write('Option 4'), nl,
clearScreen,
gameComputerVsComputer(Board, PiecesWhite, PiecesBlack, 0, 0, 1),
ni_ju.

initGame(Option, Board, PiecesWhite, PiecesBlack):-
Option == 5, !,
write('Option 5'), nl,
clearScreen,
gameComputerVsComputer(Board, PiecesWhite, PiecesBlack, 0, 0, 2),
ni_ju.

initGame(_Option, _Board, _PiecesWhite, _PiecesBlack).
```

7.2 board.pl

```
/**
* Give a list of values.
```

```
**/
addNilSpaces(0, _, []).
addNilSpaces(Length, Value, [Value|NewList]):-
NewLength is Length-1,
NewLength @>= 0,
addNilSpaces(NewLength, Value, NewList).


/**
* Insert new piece at specific position.
**/
insertAt(Piece,Column,Board,NewBoard) :-
same_length([Piece|Board],NewBoard),
append(Before,BoardAux,Board),
length(Before,Column),
append(Before,[Piece|BoardAux],NewBoard).


/**
* Replace one element on matrix.
**/
replace( L , X , Y , Z , R ) :-
append(RowPfx,[Row|RowSfx],L),
length(RowPfx,X) ,
append(ColPfx,[_|ColSfx],Row) ,
length(ColPfx,Y) ,
append(ColPfx,[Z|ColSfx],RowNew) ,
append(RowPfx,[RowNew|RowSfx],R).


/**
* Expands the board after the first move.
**/
addSpaceMatrix(Board, Length, NewBoard):-
addNilSpaces(Length, nil, AuxList1),
append([Board], [AuxList1], AuxNewBoard),
addNilSpaces(Length, nil, AuxList2),
append([AuxList2], AuxNewBoard, NewBoard).


/**
* First move.
**/
addPiece(Board, PieceCode, Color, Rotation, NewBoard):-
append([[PieceCode,Rotation,Color,0]], Board, AuxBoard),
append([nil], AuxBoard, AuxTwoBoard),
length(AuxTwoBoard,Length),
addSpaceMatrix(AuxTwoBoard, Length, NewBoard).


/**
* Following movements.
**/
addPiece(Board, Row, Column, PieceCode, Color, Rotation, NewBoard):-
replace(Board,Row,Column,[PieceCode,Rotation,Color,0], AuxBoard),
verifyExpandBoard(Row, Column, AuxBoard, NewBoard).


/**
* Move piece.
**/
movePiece(Board, SourceRow, SourceColumn, Rotation, DestRow, DestColumn,
 ↪  NewBoard):-
nth0(SourceRow, Board, RowBoard),
nth0(SourceColumn, RowBoard, Piece),
nth0(0, Piece, PieceCode),
nth0(2, Piece, Color),
replace(Board, SourceRow, SourceColumn, nil, NewBoardAux),
replace(NewBoardAux, DestRow, DestColumn, [PieceCode, Rotation, Color, 0],
 ↪  NewBoardAux2),
verifyExpandBoard(DestRow, DestColumn, NewBoardAux2, NewBoard).


/**
* Remove a piece of available pieces after playing.
**/
removePiecePlayed(ListAvailablePieces, PieceCode,
 ↪  NewListAvailablePieces):-
        delete(ListAvailablePieces, PieceCode, NewListAvailablePieces).


/**
* Checks if you need to expand the board
**/
% Case (0,-)
verifyExpandBoard(_Row, Column, Board, NewBoard) :-
Column == 0, !,
length(Board,Height),
addColNilsLeft(Board, Height, NewBoard).
```

```
% Case (-,0)
verifyExpandBoard(Row, _Column, Board, NewBoard) :-
Row == 0, !,
addRowUp(Board, NewBoard).

% Case (length, -)
verifyExpandBoard(Row, _Column, [H | T], NewBoard) :-
length([H | T], AuxHeight),
Height is AuxHeight - 1,
Row == Height, !,
addRowDown([H | T], NewBoard).

% Case (-, width)
verifyExpandBoard(_Row, Column, [H | T], NewBoard) :-
length(H, AuxWidth),
Width is AuxWidth - 1,
Column == Width, !,
length([H | T],Height),
addColNilsRight([H | T], Height, NewBoard).

% Default case
verifyExpandBoard(_Row, _Column, _Board, _Board).


/**
* Add col of nils to left of board.
**/
addColNilsLeft([], 0, []).
addColNilsLeft([H1 | T1], Height, [H2 | T2]) :-
append([nil], H1, H2),
NewHeight is Height - 1,
addColNilsLeft(T1, NewHeight, T2).


/**
* Add col of nils to right of board.
**/
addColNilsRight([], 0, []).
addColNilsRight([H1 | T1], Width, [H2 | T2]) :-
length(H1, Pos),
insertAt(nil, Pos, H1, H2),
NewWidth is Width - 1,
addColNilsRight(T1, NewWidth, T2).
```

```
/**
* Add row of nils to up of board.
**/
addRowUp([H1 | T1], NewBoard) :-
length(H1, Width),
addNilSpaces(Width, nil, AuxList),
append([AuxList], [H1 | T1], NewBoard).


/**
* Add row of nils to down of board.
**/
addRowDown([H1 | T1], NewBoard) :-
length(H1, Width),
addNilSpaces(Width, nil, AuxList),
append([H1 | T1], [AuxList], NewBoard).


/**
* Get all valid moves.
**/
getValidMoves(Board, ValidMoves) :-
        setof([X,Y], validMove(Board, X, Y), ValidMoves).


/**
* Check moves - ensures that the position is free and next to another
 ↪  piece.
**/
% Check if there is a piece in the row before.
validMove(Board, NumRow, NumCol):-
        nth0(NumRow, Board, RowChosen),
        nth0(NumCol, RowChosen, CellChosen),
        CellChosen == nil,

        BeforeNumRow is NumRow - 1,
        nth0(BeforeNumRow, Board, RowBefore),
        nth0(NumCol, RowBefore, CellBefore),
        CellBefore \== nil.

% Check if there is a piece in the row after.
validMove(Board, NumRow, NumCol):-
        nth0(NumRow, Board, RowChosen),
        nth0(NumCol, RowChosen, CellChosen),
        CellChosen == nil,
        AfterNumRow is NumRow + 1,
        nth0(AfterNumRow, Board, RowAfter),
        nth0(NumCol, RowAfter, CellAfter),
        CellAfter \== nil.

% Check if there is a piece in the column before.
validMove(Board, NumRow, NumCol):-
        nth0(NumRow, Board, RowChosen),
        nth0(NumCol, RowChosen, CellChosen),
        CellChosen == nil,

        BeforeNumColumn is NumCol - 1,
        nth0(NumRow, Board, Row),
        nth0(BeforeNumColumn, Row, CellBefore),
        CellBefore \== nil.

% Check if there is a piece in the column after.
validMove(Board, NumRow, NumCol):-
        nth0(NumRow, Board, RowChosen),
        nth0(NumCol, RowChosen, CellChosen),
        CellChosen == nil,

        AfterNumColumn is NumCol + 1,
        nth0(NumRow, Board, Row),
        nth0(AfterNumColumn, Row, CellAfter),
        CellAfter \== nil.



/**
* Get the positions of the pieces that are available to be moved.
**/
getPositionsToRemovePiece(Board, PositionsToRemove) :-
        setof([X,Y], getValidPostionToRemove(Board, X, Y),
         ↪  PositionsToRemove).


/**
* Remove a piece - ensures that the position is occupied and has a free
 ↪  seat next to it.
**/
% Check if there isn't  a piece in the row before.
getValidPostionToRemove(Board, NumRow, NumCol):-
        nth0(NumRow, Board, RowChosen),
        nth0(NumCol, RowChosen, CellChosen),
        CellChosen \== nil, % cell is not free

        BeforeNumRow is NumRow - 1,
        nth0(BeforeNumRow, Board, RowBefore),
        nth0(NumCol, RowBefore, CellBefore),
        CellBefore == nil.

% Check if there isn't  a piece in the row after.
getValidPostionToRemove(Board, NumRow, NumCol):-
        nth0(NumRow, Board, RowChosen),
        nth0(NumCol, RowChosen, CellChosen),
        CellChosen \== nil, % cell is free

        AfterNumRow is NumRow + 1,
        nth0(AfterNumRow, Board, RowAfter),
        nth0(NumCol, RowAfter, CellAfter),
        CellAfter == nil.

% Check if there isn't  a piece in the column before.
getValidPostionToRemove(Board, NumRow, NumCol):-
        nth0(NumRow, Board, RowChosen),
        nth0(NumCol, RowChosen, CellChosen),
        CellChosen \== nil, % cell is free

        BeforeNumColumn is NumCol - 1,
        nth0(NumRow, Board, Row),
        nth0(BeforeNumColumn, Row, CellBefore),
        CellBefore == nil.

% Check if there isn't a piece in the column after.
getValidPostionToRemove(Board, NumRow, NumCol):-
        nth0(NumRow, Board, RowChosen),
        nth0(NumCol, RowChosen, CellChosen),
        CellChosen \== nil, % cell is free

        AfterNumColumn is NumCol + 1,
        nth0(NumRow, Board, Row),
        nth0(AfterNumColumn, Row, CellAfter),
        CellAfter == nil.


/**
* Check if a piece of a certain position belongs to the player.
**/
```

```
checkColorPiece(Board, SourceRow, SourceColumn, ColorPlayer) :-
        nth0(SourceRow, Board, RowBoard),
        nth0(SourceColumn, RowBoard, Piece),
        nth0(2, Piece, ColorPiece),
        ColorPiece == ColorPlayer.
```

7.3 humanInput.pl

```
/**
* Verify if a number is between the interval
**/
betweeMinMax(Min, Max, Num):-
        Num >= Min,
        Num =< Max.


/**
* Asks the user the position.
**/
askBoardPosition([Head | Tail], NumRow, NumCol):-
        write('  Row: '),
        once(read(NumRow)),
        length([Head | Tail], RowMax),
        NRowMax is RowMax - 1,
        betweeMinMax(0, NRowMax, NumRow), !,
        write('  Column: '), once(read(NumCol)), nl,
        length(Head, ColumnMax),
        NColumnMax is ColumnMax - 1,
        betweeMinMax(0, NColumnMax, NumCol).


/**
* Asks the user the rotation to make the piece.
**/
askRotation(Rotation):-
        write('  Rotation (0 - 0 degrees, 1 - 90 degrees, 2 - 180 degrees,
         ↪  3 - 270 degrees): '),
        once(read(Rotation)), !,
        member(Rotation, [0,1,2,3]).


/**
* Ask the user what the next piece.
**/
```

```
askNextPiece(Pieces, Letter):-
        nl, write('  Next Piece: '),
        once(read(Letter)),
        member(Letter, Pieces).


/**
* Asks the user what is the first move.
**/
askInput(_Board, Pieces, Letter, Rotation):-
        repeat,
        once(askNextPiece(Pieces, Letter)),
        once(askRotation(Rotation)).


/**
* Ask the user what is the move with available pieces.
**/
askInput(Board, Pieces, Letter, Rotation, NumRow, NumCol):-
        repeat,
        once(askNextPiece(Pieces, Letter)),
        once(askRotation(Rotation)),
        once(askBoardPosition(Board, NumRow, NumCol)),
        once(validMove(Board, NumRow, NumCol)).


/**
* Ask the user what is the move when there aren't available parts.
**/
askInputMove(Board, SourceRow, SourceColumn, Rotation, DestRow,
 ↪  DestColumn, ColorPlayer):-
        repeat,
        write('  Select piece to move: '), nl,
        once(askBoardPosition(Board, SourceRow, SourceColumn)),
        once(checkIfRemovePieceIsValid(Board, SourceRow, SourceColumn)),
        once(checkColorPiece(Board, SourceRow, SourceColumn,
         ↪  ColorPlayer)),
        once(askRotation(Rotation)),
        write('  Select destination: '), nl,
        once(askBoardPosition(Board, DestRow, DestColumn)),
        once(validMove(Board, DestRow, DestColumn)).


/**
* Ask the user what is the menu option.
```

```
**/
askMenuInput(Options, Option):-
        nl, printSpace(10), write('Choose option: '),
        read(Option),
        member(Option, Options).
```

7.4 computerInput.pl

```
/**
* Level one/two AI - first move.
**/
computerInput(_Board, Pieces, Letter, Rotation) :-
        repeat,
        once(getPieceLetter(Pieces, Letter)),
        once(getRotation(Rotation)), nl,
        printInformation(Letter).


/**
* Level one AI - random add piece.
**/
computerInput(Board, Pieces, _AdversaryPieces, _ColorPlayer, Letter,
 ↪  Rotation, NumRow, NumCol, Level) :-
        Level == 1, !,
        repeat,
        once(getPieceLetter(Pieces, Letter)),
        once(getRotation(Rotation)),
        once(getValidPosition(Board, NumRow, NumCol)), %random positions
        printInformation(NumRow, NumCol, Letter).

/**
* Level two AI - find the piece that guarantees victory.
**/
computerInput(Board, Pieces, _AdversaryPieces, ColorPlayer, Letter,
 ↪  Rotation, NumRow, NumCol, _Level) :-
        getValidMoves(Board, ValidMoves),
        once(bestMoveVitory(Board, Pieces, _BeforeLetter, ValidMoves,
         ↪  ColorPlayer, Letter, Rotation, NumRow, NumCol, Vitory)),
        Vitory == 1, !,
        printInformation(NumRow, NumCol, Letter).

/**
* Level two AI - block opponent
**/
```

```
computerInput(Board, Pieces, AdversaryPieces, ColorPlayer, Letter,
 ↪  Rotation, NumRow, NumCol, _Level) :-
        getValidMoves(Board, ValidMoves),
        AdversaryColorPlayer is mod((ColorPlayer + 1), 2),
        once(bestMoveVitory(Board, AdversaryPieces, _BeforeLetter,
         ↪  ValidMoves, AdversaryColorPlayer, _AuxLetter, Rotation,
         ↪  NumRow, NumCol, Vitory)),
        Vitory == 1, !,
        getPieceLetter(Pieces, Letter),
        printInformation(NumRow, NumCol, Letter).

/**
* Level two AI - find the best possible move.
**/
computerInput(Board, Pieces, _AdversaryPieces, ColorPlayer, Letter,
 ↪  Rotation, NumRow, NumCol, _Level) :-
        getValidMoves(Board, ValidMoves),
        once(getSecondBestMove(Board, Pieces, ValidMoves, ColorPlayer,
         ↪  _PossibleMoves, FinalPossibleMoves)),
        once(playSecondBestMove(FinalPossibleMoves, Pieces, Board, Letter,
         ↪  Rotation, NumRow, NumCol)),
        printInformation(NumRow, NumCol, Letter).


/**
* Level one AI - move parts randomly.
**/
computerInputMove(Board, SourceRow, SourceColumn, Rotation, DestRow,
 ↪  DestColumn, ColorPlayer, Level):-
        Level == 1, !,
        repeat,
        once(getPosition(Board, SourceRow, SourceColumn)),
        once(getValidPostionToRemove(Board, SourceRow, SourceColumn)),
        once(checkColorPiece(Board, SourceRow, SourceColumn,
         ↪  ColorPlayer)),
        once(getRotation(Rotation)),
        once(getValidPosition(Board, DestRow, DestColumn)), %random
         ↪  positions
        once(printInformation(SourceRow, SourceColumn, DestRow,
         ↪  DestColumn)).


/**
* Level two AI - Find on the board the piece that guarantees victory and
 ↪  moves it to the right place.
```

```
**/
computerInputMove(Board, SourceRow, SourceColumn, Rotation, DestRow,
 ↪  DestCol, ColorPlayer, _Level):-
        once(getPositionsToRemovePiece(Board, PositionsToRemove)),
        once(getColorPieces(Board, ColorPlayer, ListPiecesPlayer)),
        once(inter(PositionsToRemove, ListPiecesPlayer,
         ↪  FinalListToRemove)),
        once(getLetters(Board, FinalListToRemove, _LettersPositions,
         ↪  LettersPositionsAux, _LettersAvailable,
         ↪  LettersAvailableAux)),
        once(getValidMoves(Board, ValidMoves)),
        once(bestMoveVitory(Board, LettersAvailableAux, _BeforeLetter,
         ↪  ValidMoves, ColorPlayer, Letter, Rotation, DestRow, DestCol,
         ↪  Vitory)),
        Vitory == 1, !,
        getCoordinates(Letter, LettersPositionsAux, SourceRow,
         ↪  SourceColumn),
        once(printInformation(SourceRow, SourceColumn, DestRow, DestCol)).


/**
* Level two AI - block opponent
**/
computerInputMove(Board, SourceRow, SourceColumn, Rotation, DestRow,
 ↪  DestCol, ColorPlayer, _Level):-
        once(getPositionsToRemovePiece(Board, PositionsToRemove)),
        AdversaryColorPlayer is mod((ColorPlayer + 1), 2),
        once(getColorPieces(Board, AdversaryColorPlayer,
         ↪  ListPiecesAdvPlayer)),
        once(inter(PositionsToRemove, ListPiecesAdvPlayer,
         ↪  FinalListToRemoveAdv)),
        once(getLetters(Board, FinalListToRemoveAdv, _LettersPositions,
         ↪  _LettersPositionsAux2, _LettersAvailable,
         ↪  LettersAvailableAux2)),
        once(getValidMoves(Board, ValidMoves)),
        once(bestMoveVitory(Board, LettersAvailableAux2, _BeforeLetter,
         ↪  ValidMoves, AdversaryColorPlayer, _AuxLetter, Rotation,
         ↪  DestRow, DestCol, Vitory)),
        Vitory == 1, !,
        once(getColorPieces(Board, ColorPlayer, ListPiecesPlayer)),
        once(inter(PositionsToRemove, ListPiecesPlayer,
         ↪  FinalListToRemove)),
        once(getLetters(Board, FinalListToRemove, _LettersPositions,
         ↪  LettersPositionsAux, _LettersAvailable,
         ↪  LettersAvailableAux)),
        getPieceLetter(LettersAvailableAux, Letter),
        getCoordinates(Letter, LettersPositionsAux, SourceRow,
         ↪  SourceColumn),

        once(printInformation(SourceRow, SourceColumn, DestRow, DestCol)).

/**
* Level two AI - find the best possible move.
**/
computerInputMove(Board, SourceRow, SourceColumn, Rotation, DestRow,
 ↪  DestCol, ColorPlayer, _Level):-
        once(getPositionsToRemovePiece(Board, PositionsToRemove)),
        once(getColorPieces(Board, ColorPlayer, ListPiecesPlayer)),
        once(inter(PositionsToRemove, ListPiecesPlayer,
         ↪  FinalListToRemove)),
        once(getLetters(Board, FinalListToRemove, _LettersPositions,
         ↪  LettersPositionsAux, _LettersAvailable,
         ↪  LettersAvailableAux)),
        getValidMoves(Board, ValidMoves),
        once(getSecondBestMove(Board, LettersAvailableAux, ValidMoves,
         ↪  ColorPlayer, _PossibleMoves, FinalPossibleMoves)),
        once(playSecondBestMove(FinalPossibleMoves, LettersAvailableAux,
         ↪  Board, Letter, Rotation, DestRow, DestCol)),
        getCoordinates(Letter, LettersPositionsAux, SourceRow,
         ↪  SourceColumn),
        once(printInformation(SourceRow, SourceColumn, DestRow, DestCol)).


/**
* Get a random piece from a list of available pieces.
**/
getPieceLetter(Pieces, Letter) :-
        length(Pieces, NumPieces),
        random(0, NumPieces, PosPiece),
        nth0(PosPiece, Pieces, Letter).


/**
* Get a random rotation for a piece.
**/
getRotation(Rotation) :-
        random(0, 3, Rotation).


/**
```

```
* Get a random coordinates from valid moves for add next piece.
**/
getValidPosition(Board, NumRow, NumCol) :-
        getValidMoves(Board, ValidMoves),
        length(ValidMoves, NumValidMoves),
        random(0, NumValidMoves, ValidMovePos),
        nth0(ValidMovePos, ValidMoves, ValidMoveChoosen),
        nth0(0, ValidMoveChoosen, NumRow),
        nth0(1, ValidMoveChoosen, NumCol).


/**
* Get a random coordinates for remove a piece.
**/
getPosition([H|T], NumRow, NumCol) :-
        length([H|T], AuxNumRows),
        NumRows is AuxNumRows - 1,
        random(0, NumRows, NumRow),
        length(H, AuxNumCols),
        NumCols is AuxNumCols - 1,
        random(0, NumCols, NumCol).


/**
* Get all the pieces in play of a given player.
**/
getColorPieces(Board, ColorPlayer, ListPiecesPlayer) :-
        setof([X,Y], checkColorPiece(Board, X, Y, ColorPlayer),
         ↪  ListPiecesPlayer).


/**
* Get the piece of a particular position.
**/
getPiece(Board, Row, Col, Piece) :-
        nth0(Row, Board, RowBoard),
        nth0(Col, RowBoard, Piece).


/**
* Get coordinates of Piece from a list of pieces.
**/
getCoordinates(_, [], _, _).
getCoordinates(Letter, [H | _T], SourceRow, SourceColumn) :-
        nth0(0, H, LetterAux),
        LetterAux == Letter, !,
        nth0(1, H, SourceRow),
        nth0(2, H, SourceColumn).
getCoordinates(Letter, [_H | T], SourceRow, SourceColumn) :-
        getCoordinates(Letter, T, SourceRow, SourceColumn).


/**
* Get letters and coordinates of available pieces to move.
**/
getLetters(_Board, [], LettersPositions, LettersPositions,
 ↪  LettersAvailable, LettersAvailable).
getLetters(Board, [H | T], LettersPositions, LettersPositionsAux,
 ↪  LettersAvailable, LettersAvailableAux) :-
        length(LettersAvailable, Aux),
        Aux \== 0, !,
        nth0(0, H, Row),
        nth0(1, H, Col),
        getPiece(Board, Row, Col, Piece),
        nth0(0, Piece, Letter),
        append([Letter], H, LetterPos),
        append(LettersPositions, [LetterPos], NewLettersPositions),
        append([Letter], LettersAvailable , NewLettersAvailable),
        getLetters(Board, T, NewLettersPositions, LettersPositionsAux,
         ↪  NewLettersAvailable, LettersAvailableAux).
getLetters(Board, [H | T], LettersPositions, LettersPositionsAux,
 ↪  _LettersAvailable, LettersAvailableAux) :-
        nth0(0, H, Row),
        nth0(1, H, Col),
        getPiece(Board, Row, Col, Piece),
        nth0(0, Piece, Letter),
        append([Letter], H, LetterPos),
        append(LettersPositions, [LetterPos], NewLettersPositions),
        getLetters(Board, T, NewLettersPositions, LettersPositionsAux,
         ↪  [Letter], LettersAvailableAux).
```

7.5 ai.pl

```
/**
* Against the valid plays, choose the letter that guarantees victory.
**/
bestMoveVitory(_, [], _, _, _, _, _, _, _, _).
bestMoveVitory(Board, [H|T], _BeforeLetter, ValidMoves, ColorPlayer,
 ↪  Letter, Rotation, Row, Col, Vitory) :-
```

```
        Vitory \== 1,
        nth0(0, ValidMoves, BeforeCords),
        bestValidMove(Board, H, ValidMoves, BeforeCords, ColorPlayer,
         ↪  Rotation, Row, Col, Vitory),
        bestMoveVitory(Board, T, H, ValidMoves, ColorPlayer, Letter,
         ↪  Rotation, Row, Col, Vitory).
bestMoveVitory(_Board, [_H|_T], BeforeLetter, _ValidMoves, _ColorPlayer,
 ↪  Letter, _Rotation, _Row, _Col, Vitory) :-
        Vitory == 1, !,
        Letter = BeforeLetter.


/**
* Choose the place that guarantees victory.
**/
bestValidMove(_, _, [], BeforeCords, _, _, Row, Col, Vitory):-
    Vitory == 1,!,
    nth0(0, BeforeCords, X), Row is X,
    nth0(1, BeforeCords, Y), Col is Y.
bestValidMove(_, _, [], _, _, _, _, _, _).
bestValidMove(_Board, _, [_|_], BeforeCords, _, _Rotation, Row, Col,
 ↪  Vitory) :-
    Vitory == 1, !,
    nth0(0, BeforeCords, X), Row is X,
    nth0(1, BeforeCords, Y), Col is Y.
bestValidMove(Board, Letter, [H|T], _BeforeCords, ColorPlayer, Rotation,
 ↪  Row, Col, Vitory) :-
    Vitory \== 1, !,
    nth0(0, H, X), nth0(1, H, Y),
    replace(Board, X, Y, nil, NewBoard),
    bestRotation(NewBoard, Letter, X, Y, ColorPlayer, 0, Rotation, 0,
     ↪  Vitory),
    bestValidMove(NewBoard, Letter, T, H, ColorPlayer, Rotation, Row, Col,
     ↪  Vitory).


/**
* Choose the rotation that guarantees victory.
**/
bestRotation(_, _, _, _, _, 4, _, _, _). % Not vitory
bestRotation(Board, Letter, Row, Col, ColorPlayer, AuxRot, Rotation,
 ↪  Vitory, Aux) :-
        Vitory \== 1, !,
        addPiece(Board, Row, Col, Letter, ColorPlayer, AuxRot, NewBoard),
        checkGameEnd(NewBoard, NewBoard, _InvalidPieces,
         ↪  _FinalInvalidPieces, 0, NewVitory),
        NewAuxRot is AuxRot + 1,
        bestRotation(Board, Letter, Row, Col, ColorPlayer, NewAuxRot,
         ↪  Rotation, NewVitory, Aux).

bestRotation(_Board, _Letter, _Row, _Col, _ColorPlayer, AuxRot, Rotation,
 ↪  Vitory, Aux) :-
        Aux is Vitory,
        Rotation is AuxRot - 1.


/**
* Find the move with the best pontuation.
**/
checkNextCell([Letter, Rotation, Color, Valid], ColorPiece, True,
 ↪  PieceRow, PieceCol):-
        getPiece([Letter, Rotation, Color, Valid], Pattern),
        nth0(PieceRow, Pattern, Row),
        nth0(PieceCol, Row, Cell),
        Color == ColorPiece,
        Valid == 0,
        Cell == 1, !,
        True is 1.
checkNextCell(_Piece, _ColorPiece, True, _PieceRow, _PieceCol):-
        True is 0.


/**
* Check color of a piece.
**/
checkColorPiece(Piece, Color, Count, NewCount, _Valid, PieceRow,
 ↪  PieceCol):-
        checkNextCell(Piece, Color, True, PieceRow, PieceCol),
        True == 1,!,
        NewCount is Count + 2.
checkColorPiece(Piece, Color, Count, NewCount, _Valid, _PieceRow,
 ↪  _PieceCol):-
        nth0(2, Piece, ColorCell),
        ColorCell == Color, !,
        NewCount is Count + 1.
checkColorPiece(_Piece, _Color, Count, NewCount, Valid, _PieceRow,
 ↪  _PieceCol):-
        NewCount is Count,
        Valid is 1.
```

```
/**
* Each case of each row of the pattern of piece.
**/
%row 0, col 0
checkRowScore([Head | Tail], Color, Valid, Board, 0, 0, Row, Col, Count,
 ↪  FinalCount):-
        Head == 1,
        BeforeRowNum is Row - 1,
        BeforeCol is Col - 1,
        nth0(BeforeRowNum, Board, BeforeRow),
        nth0(BeforeCol, BeforeRow, DiagonalCell),
        DiagonalCell \== nil,!,
        checkColorPiece(DiagonalCell, Color, Count, NewCount, Valid, 2,
         ↪  2),
        checkRowScore(Tail, Color, Valid, Board, 0, 1, Row, Col, NewCount,
         ↪  FinalCount).
checkRowScore([_Head | Tail], Color, Valid, Board, 0, 0, Row, Col, Count,
 ↪  FinalCount):-
        checkRowScore(Tail, Color, Valid, Board, 0, 1, Row, Col, Count,
         ↪  FinalCount).

%row 0, col 1
checkRowScore([Head | Tail], Color, Valid, Board, 0, 1, Row, Col, Count,
 ↪  FinalCount):-
        Head == 1,
        BeforeRowNum is Row - 1,
        nth0(BeforeRowNum, Board, BeforeRow),
        nth0(Col, BeforeRow, AboveCell),
        AboveCell \== nil,!,
        checkColorPiece(AboveCell, Color, Count, NewCount, Valid, 2, 1),
        checkRowScore(Tail, Color, Valid, Board, 0, 2, Row, Col, NewCount,
         ↪  FinalCount).
checkRowScore([_Head | Tail], Color, Valid, Board, 0, 1, Row, Col, Count,
 ↪  FinalCount):-
        checkRowScore(Tail, Color, Valid, Board, 0, 2, Row, Col, Count,
         ↪  FinalCount).

%row 0, col 2
checkRowScore([Head | Tail], Color, Valid, Board, 0, 2, Row, Col, Count,
 ↪  FinalCount):-
        Head == 1,
        BeforeRowNum is Row - 1,
        AfterCol is Col + 1,
        nth0(BeforeRowNum, Board, BeforeRow),
        nth0(AfterCol, BeforeRow, DiagonalCell),
        DiagonalCell \== nil,!,
        checkColorPiece(DiagonalCell, Color, Count, NewCount, Valid, 2,
         ↪  0),
        checkRowScore(Tail, _, Valid, _, _, _, _, _, NewCount,
         ↪  FinalCount).
checkRowScore([_Head | Tail], _Color, Valid, _Board, 0, 2, _Row, _Col,
 ↪  Count, FinalCount):-
        checkRowScore(Tail, _, Valid, _, _, _, _, _, Count, FinalCount).

%row 1, col 0
checkRowScore([Head | Tail], Color, Valid, Board, 1, 0, Row, Col, Count,
 ↪  FinalCount):-
        Head == 1,
        BeforeCol is Col - 1,
        nth0(Row, Board, SameRow),
        nth0(BeforeCol, SameRow, BeforeCell),
        BeforeCell \== nil,!,
        checkColorPiece(BeforeCell, Color, Count, NewCount, Valid, 1, 2),
        checkRowScore(Tail, Color, Valid, Board, 1, 1, Row, Col, NewCount,
         ↪  FinalCount).
checkRowScore([_Head | Tail], Color, Valid, Board, 1, 0, Row, Col, Count,
 ↪  FinalCount):-
        checkRowScore(Tail, Color, Valid, Board, 1, 1, Row, Col, Count,
         ↪  FinalCount).

%row 1, col 1
checkRowScore([_Head | Tail], Color, Valid, Board, 1, 1, Row, Col, Count,
 ↪  FinalCount):-
        checkRowScore(Tail, Color, Valid, Board, 1, 2, Row, Col, Count,
         ↪  FinalCount).

%row 1, col 2
checkRowScore([Head | Tail], Color, Valid, Board, 1, 2, Row, Col, Count,
 ↪  FinalCount):-
        Head == 1,
        AfterCol is Col + 1,
        nth0(Row, Board, SameRow),
        nth0(AfterCol, SameRow, AfterCell),
        AfterCell \== nil,!,
        checkColorPiece(AfterCell, Color, Count, NewCount, Valid, 1, 0),
        checkRowScore(Tail, Color, Valid, Board, 1, 2, Row, Col, NewCount,
         ↪  FinalCount). % Alterar para nada
```

```
checkRowScore([_Head | Tail], Color, Valid, Board, 1, 2 , Row, Col, Count,
 ↪  FinalCount):-
        checkRowScore(Tail, Color, Valid, Board, 1, 2, Row, Col, Count,
         ↪  FinalCount).

%row 2, col 0
checkRowScore([Head | Tail], Color, Valid, Board, 2, 0, Row, Col, Count,
 ↪  FinalCount):-
        Head == 1,
        AfterRowNum is Row + 1,
        BeforeCol is Col - 1,
        nth0(AfterRowNum, Board, AfterRow),
        nth0(BeforeCol, AfterRow, DiagonalCell),
        DiagonalCell \== nil,!,
        checkColorPiece(DiagonalCell, Color, Count, NewCount, Valid, 0,
         ↪  2),
        checkRowScore(Tail, Color, Valid, Board, 2, 1, Row, Col, NewCount,
         ↪  FinalCount).
checkRowScore([_Head | Tail], Color, Valid, Board, 2, 0, Row, Col, Count,
 ↪  FinalCount):-
        checkRowScore(Tail, Color, Valid, Board, 2, 1, Row, Col, Count,
         ↪  FinalCount).

%row 2, col 1
checkRowScore([Head | Tail], Color, Valid, Board, 2, 1, Row, Col, Count,
 ↪  FinalCount):-
        Head == 1,
        AfterRowNum is Row + 1,
        nth0(AfterRowNum, Board, AfterRow),
        nth0(Col, AfterRow, UnderCell),
        UnderCell \== nil,!,
        checkColorPiece(UnderCell, Color, Count, NewCount, Valid, 0, 1),
        checkRowScore(Tail, Color, Valid, Board, 2, 2, Row, Col, NewCount,
         ↪  FinalCount).
checkRowScore([_Head | Tail], Color, Valid, Board, 2, 1, Row, Col, Count,
 ↪  FinalCount):-
        checkRowScore(Tail, Color, Valid, Board, 2, 2, Row, Col, Count,
         ↪  FinalCount).

%row 2, col 2
checkRowScore([Head | Tail], Color, Valid, Board, 2, 2, Row, Col, Count,
 ↪  FinalCount):-
        Head == 1,
        AfterRowNum is Row + 1,
        AfterCol is Col + 1,
        nth0(AfterRowNum, Board, AfterRow),
        nth0(AfterCol, AfterRow, DiagonalCell),
        DiagonalCell \== nil,!,
        checkColorPiece(DiagonalCell, Color, Count, NewCount, Valid, 0,
         ↪  0),
        checkRowScore(Tail, _, Valid, _, _, _, _, _, NewCount,
         ↪  FinalCount).
checkRowScore([_Head | Tail], _Color, Valid, _Board, 2, 2, _Row, _Col,
 ↪  Count, FinalCount):-
        checkRowScore(Tail, _, Valid, _, _, _, _, _, Count, FinalCount).

checkRowScore([], _, _Valid, _, _, _, _, _, Count, FinalCount):-
 ↪  FinalCount is Count.


/**
* Calculates the score according to the pieces around.
**/
checkAroundScore([], _Valid, _ColorPlayer, _Board, _PieceRow, _Row, _Col,
 ↪  Count, LastFinalCount):- LastFinalCount is Count.
checkAroundScore([Head | Tail], Valid, ColorPlayer, Board, PieceRow, Row,
 ↪  Col, Count, LastFinalCount):-
        checkRowScore(Head, ColorPlayer, Valid, Board, PieceRow, 0, Row,
         ↪  Col, Count, FinalCount),
        NewPieceRow is PieceRow + 1,
        NewCountPiecesAround is FinalCount,
        checkAroundScore(Tail, Valid, ColorPlayer, Board, NewPieceRow,
         ↪  Row, Col, NewCountPiecesAround, LastFinalCount).


/**
* Get the score of each valid position to play.
**/
getPositionScore(_Board, [], _Position, _Rotation, _ColorPlayer,
 ↪  PositionMoves, AllPositionMoves):-
        copyList(PositionMoves, AllPositionMoves).

getPositionScore(Board, [_Head | Tail], 4, Position, ColorPlayer,
 ↪  PositionMoves, AllPositionMoves):-
        getPositionScore(Board, Tail, 0, Position, ColorPlayer,
         ↪  PositionMoves, AllPositionMoves).

getPositionScore(Board, [Head | Tail], Rotation, Position, ColorPlayer,
 ↪  PositionMoves, AllPositionMoves):-
        getPiece([Head, Rotation, ColorPlayer, 0], Pattern),
        nth0(0, Position, Row),
        nth0(1, Position, Col),
        replace(Board, Row, Col, nil, NewBoard),
        checkAroundScore(Pattern, Valid, ColorPlayer, NewBoard, 0, Row,
         ↪  Col, 0, LastFinalCount),
        LastFinalCount \== 0, Valid \== 1, !,
        append(PositionMoves, [[LastFinalCount, Row, Col, [Head, Rotation,
         ↪  ColorPlayer, 0]]], NewPositionMoves),
        NewRotation is Rotation + 1,
        getPositionScore(Board, [Head | Tail], NewRotation, Position,
         ↪  ColorPlayer, NewPositionMoves, AllPositionMoves).

getPositionScore(Board, [Head | Tail], Rotation, Position, ColorPlayer,
 ↪  PositionMoves, AllPositionMoves):-
        NewRotation is Rotation + 1,
        getPositionScore(Board, [Head | Tail], NewRotation, Position,
         ↪  ColorPlayer, PositionMoves, AllPositionMoves).


/**
* Get de second best move.
**/
getSecondBestMove(_Board, _AvailablePieces, [], _ColorPlayer,
 ↪  PossibleMoves, FinalPossibleMoves):-
        sort(PossibleMoves, FinalPossibleMoves).

getSecondBestMove(Board, AvailablePieces, [Head | Tail], ColorPlayer,
 ↪  PossibleMoves, FinalPossibleMoves):-
        getPositionScore(Board, AvailablePieces, 0, Head, ColorPlayer,
         ↪  _NewPositionMoves, AllPositionMoves),
        append(PossibleMoves, AllPositionMoves, NewPossibleMoves),
        getSecondBestMove(Board, AvailablePieces, Tail, ColorPlayer,
         ↪  NewPossibleMoves, FinalPossibleMoves).


/**
* Play the second best towards the possible moves.
**/
playSecondBestMove(PossibleMoves, Pieces, Board, Letter, Rotation, NumRow,
 ↪  NumCol):-
        length(PossibleMoves, ListSize),
        ListSize == 0,!,
        once(getPieceLetter(Pieces, Letter)),
        once(getRotation(Rotation)),
        once(getValidPosition(Board, NumRow, NumCol)).
```

```
playSecondBestMove(PossibleMoves, _Pieces, _Board, Letter, Rotation,
 ↪  NumRow, NumCol):-
        last(PossibleMoves, BestMove),
        nth0(1, BestMove, NumRow),
        nth0(2, BestMove, NumCol),
        nth0(3, BestMove, [Letter, Rotation, _Color, _Valid]).
```

7.6 gameStatus.pl

```
/**
* Verify Draw.
**/
vertifyDraw(PiecesWhite, PiecesBlack, NewDraw) :-
        length(PiecesBlack, NumPiecesBlack),
        length(PiecesWhite, NumPiecesWhite),
        NumPiecesWhite == 0,
        NumPiecesBlack == 0, !,
        NewDraw = 1.
vertifyDraw(_PiecesWhite, _PiecesBlack, NewDraw) :-
        !, NewDraw = 0.


/**
* Check Color of Piece.
**/
checkColorPiece(Piece, Color, Count, NewCount, _Valid):-
        nth0(2, Piece, ColorCell),
        ColorCell == Color, !,
        NewCount is Count + 1.
checkColorPiece(_Piece, _Color, Count, NewCount, Valid):-
        NewCount is Count,
        Valid is 1.


/**
* Each case of each row of the pattern of piece.
**/
%row 0, col 0
checkRow([Head | Tail], Color, Valid, Board, 0, 0, Row, Col, Count,
 ↪  FinalCount):-
        Head == 1,
        BeforeRowNum is Row - 1,
        BeforeCol is Col - 1,
        nth0(BeforeRowNum, Board, BeforeRow),
        nth0(BeforeCol, BeforeRow, DiagonalCell),
        DiagonalCell \== nil,!,
        checkColorPiece(DiagonalCell, Color, Count, NewCount, Valid),
        checkRow(Tail, Color, Valid, Board, 0, 1, Row, Col, NewCount,
         ↪  FinalCount).
checkRow([_Head | Tail], Color, Valid, Board, 0, 0, Row, Col, Count,
 ↪  FinalCount):-
        checkRow(Tail, Color, Valid, Board, 0, 1, Row, Col, Count,
         ↪  FinalCount).

%row 0, col 1
checkRow([Head | Tail], Color, Valid, Board, 0, 1, Row, Col, Count,
 ↪  FinalCount):-
        Head == 1,
        BeforeRowNum is Row - 1,
        nth0(BeforeRowNum, Board, BeforeRow),
        nth0(Col, BeforeRow, AboveCell),
        AboveCell \== nil,!,
        checkColorPiece(AboveCell, Color, Count, NewCount, Valid),
        checkRow(Tail, Color, Valid, Board, 0, 2, Row, Col, NewCount,
         ↪  FinalCount).
checkRow([_Head | Tail], Color, Valid, Board, 0, 1, Row, Col, Count,
 ↪  FinalCount):-
        checkRow(Tail, Color, Valid, Board, 0, 2, Row, Col, Count,
         ↪  FinalCount).

%row 0, col 2
checkRow([Head | Tail], Color, Valid, Board, 0, 2, Row, Col, Count,
 ↪  FinalCount):-
        Head == 1,
        BeforeRowNum is Row - 1,
        AfterCol is Col + 1,
        nth0(BeforeRowNum, Board, BeforeRow),
        nth0(AfterCol, BeforeRow, DiagonalCell),
        DiagonalCell \== nil,!,
        checkColorPiece(DiagonalCell, Color, Count, NewCount, Valid),
        checkRow(Tail, _, Valid, _, _, _, _, _, NewCount, FinalCount).
checkRow([_Head | Tail], _Color, Valid, _Board, 0, 2, _Row, _Col, Count,
 ↪  FinalCount):-
        checkRow(Tail, _, Valid, _, _, _, _, _, Count, FinalCount).

%row 1, col 0
checkRow([Head | Tail], Color, Valid, Board, 1, 0, Row, Col, Count,
 ↪  FinalCount):-
        Head == 1,
        BeforeCol is Col - 1,
        nth0(Row, Board, SameRow),
        nth0(BeforeCol, SameRow, BeforeCell),
        BeforeCell \== nil,!,
        checkColorPiece(BeforeCell, Color, Count, NewCount, Valid),
        checkRow(Tail, Color, Valid, Board, 1, 1, Row, Col, NewCount,
         ↪  FinalCount).
checkRow([_Head | Tail], Color, Valid, Board, 1, 0, Row, Col, Count,
 ↪  FinalCount):-
        checkRow(Tail, Color, Valid, Board, 1, 1, Row, Col, Count,
         ↪  FinalCount).

%row 1, col 1
checkRow([_Head | Tail], Color, Valid, Board, 1, 1, Row, Col, Count,
 ↪  FinalCount):-
        checkRow(Tail, Color, Valid, Board, 1, 2, Row, Col, Count,
         ↪  FinalCount).

%row 1, col 2
checkRow([Head | Tail], Color, Valid, Board, 1, 2, Row, Col, Count,
 ↪  FinalCount):-
        Head == 1,
        AfterCol is Col + 1,
        nth0(Row, Board, SameRow),
        nth0(AfterCol, SameRow, AfterCell),
        AfterCell \== nil,!,
        checkColorPiece(AfterCell, Color, Count, NewCount, Valid),
        checkRow(Tail, Color, Valid, Board, 1, 2, Row, Col, NewCount,
         ↪  FinalCount). % Alterar para nada
checkRow([_Head | Tail], Color, Valid, Board, 1, 2 , Row, Col, Count,
 ↪  FinalCount):-
        checkRow(Tail, Color, Valid, Board, 1, 2, Row, Col, Count,
         ↪  FinalCount).

%row 2, col 0
checkRow([Head | Tail], Color, Valid, Board, 2, 0, Row, Col, Count,
 ↪  FinalCount):-
        Head == 1,
        AfterRowNum is Row + 1,
        BeforeCol is Col - 1,
        nth0(AfterRowNum, Board, AfterRow),
        nth0(BeforeCol, AfterRow, DiagonalCell),
        DiagonalCell \== nil,!,
        checkColorPiece(DiagonalCell, Color, Count, NewCount, Valid),
        checkRow(Tail, Color, Valid, Board, 2, 1, Row, Col, NewCount,
         ↪  FinalCount).
checkRow([_Head | Tail], Color, Valid, Board, 2, 0, Row, Col, Count,
 ↪  FinalCount):-
        checkRow(Tail, Color, Valid, Board, 2, 1, Row, Col, Count,
         ↪  FinalCount).

%row 2, col 1
checkRow([Head | Tail], Color, Valid, Board, 2, 1, Row, Col, Count,
 ↪  FinalCount):-
        Head == 1,
        AfterRowNum is Row + 1,
        nth0(AfterRowNum, Board, AfterRow),
        nth0(Col, AfterRow, UnderCell),
        UnderCell \== nil,!,
        checkColorPiece(UnderCell, Color, Count, NewCount, Valid),
        checkRow(Tail, Color, Valid, Board, 2, 2, Row, Col, NewCount,
         ↪  FinalCount).
checkRow([_Head | Tail], Color, Valid, Board, 2, 1, Row, Col, Count,
 ↪  FinalCount):-
        checkRow(Tail, Color, Valid, Board, 2, 2, Row, Col, Count,
         ↪  FinalCount).

%row 2, col 2
checkRow([Head | Tail], Color, Valid, Board, 2, 2, Row, Col, Count,
 ↪  FinalCount):-
        Head == 1,
        AfterRowNum is Row + 1,
        AfterCol is Col + 1,
        nth0(AfterRowNum, Board, AfterRow),
        nth0(AfterCol, AfterRow, DiagonalCell),
        DiagonalCell \== nil,!,
        checkColorPiece(DiagonalCell, Color, Count, NewCount, Valid),
        checkRow(Tail, _, Valid, _, _, _, _, _, NewCount, FinalCount).
checkRow([_Head | Tail], _Color, Valid, _Board, 2, 2, _Row, _Col, Count,
 ↪  FinalCount):-
        checkRow(Tail, _, Valid, _, _, _, _, _, Count, FinalCount).

checkRow([], _, _Valid, _, _, _, _, _, Count, FinalCount):- FinalCount is
 ↪  Count.


/**
* Goes throught the piece pattern and checks each row of it.
**/
```

```
checkAroundPiece([], _, _, _, _, _, _, CountPiecesAround, GameEnd):-
↪   CountPiecesAround == 4, !, GameEnd is 1.
checkAroundPiece([], _, _, _, _, _, _, _, _).
checkAroundPiece([Head | Tail], Color, Valid, Board, PieceRow, Row, Col,
↪   CountPiecesAround, GameEnd):-
        checkRow(Head, Color, Valid, Board, PieceRow, 0, Row, Col,
         ↪   CountPiecesAround, FinalCount),
        NewPieceRow is PieceRow + 1,
        NewCountPiecesAround is FinalCount,
        checkAroundPiece(Tail, Color, Valid, Board, NewPieceRow, Row, Col,
         ↪   NewCountPiecesAround, GameEnd).


/**
* Gets the pattern of the piece and checks if it is valid.
**/
checkPieceStatus([Letter, Rotation, Color, Valid], Board, _InvalidPiece,
↪   Row, Col, GameEnd):-
        Valid \== 1,
        getPiece([Letter, Rotation, Color, Valid], Pattern),
        checkAroundPiece(Pattern, Color, _NewValid, Board, 0, Row, Col, 0,
         ↪   NewGameEnd),
        NewGameEnd == 1,!,
        GameEnd is 1.
checkPieceStatus([Letter, Rotation, Color, Valid], Board, InvalidPiece,
↪   Row, Col, _GameEnd):-
        Valid \== 1,
        getPiece([Letter, Rotation, Color, Valid], Pattern),
        checkAroundPiece(Pattern, Color, NewValid, Board, 0, Row, Col, 0,
         ↪   _NewGameEnd),
        NewValid == 1,!,
        append(_, [Row, Col], InvalidPiece).
checkPieceStatus([_, _, _, _], _, _, _, _, _).


/**
* Appends the new Invalid Pieces to the others.
**/
aggregateInvalidPieces(OldInvalidPieces, NewInvalidPieces,
↪   AllInvalidPieces):-
        is_list(OldInvalidPieces),
        is_list(NewInvalidPieces),!,
        append(OldInvalidPieces, NewInvalidPieces, AllInvalidPieces).
aggregateInvalidPieces(OldInvalidPieces, _NewInvalidPieces,
↪   AllInvalidPieces):-
```

```
        is_list(OldInvalidPieces),!,
        copyList(OldInvalidPieces, AllInvalidPieces).
aggregateInvalidPieces(_OldInvalidPieces, NewInvalidPieces,
 ↪  AllInvalidPieces):-
        is_list(NewInvalidPieces),!,
        copyList(NewInvalidPieces, AllInvalidPieces).
aggregateInvalidPieces(_OldInvalidPieces, _NewInvalidPieces,
 ↪  _AllInvalidPieces).


/**
* Goes through the row of the board and finds a Piece.
**/
findPiece([], _, InvalidPieces, NewInvalidPieces, _, _, _):-
        copyList(InvalidPieces, NewInvalidPieces).
findPiece([_ | _], _, _, _, _, _, GameEnd):- GameEnd == 1, !.
findPiece([Head | Tail], Board, InvalidPieces, NewInvalidPieces, Row, Col,
 ↪  GameEnd):-
        Head == nil,!,
        NewCol is Col + 1,
        findPiece(Tail, Board, InvalidPieces, NewInvalidPieces, Row,
         ↪  NewCol, GameEnd).
findPiece([Head | Tail], Board, InvalidPieces, NewInvalidPieces, Row, Col,
 ↪  GameEnd):-
        Head \== nil,!,
        %write(Head),nl,
        checkPieceStatus(Head, Board, NewInvalidPiece, Row, Col, GameEnd),
        NewCol is Col + 1,
        aggregateInvalidPieces(InvalidPieces, NewInvalidPiece,
         ↪  RowInvalidPieces),
        findPiece(Tail, Board, RowInvalidPieces, NewInvalidPieces, Row,
         ↪  NewCol, GameEnd).


/**
* Goes through the rows of the board.
**/
checkGameEnd(Board, NewInvalidPieces, GameEnd):-
        checkGameEnd(Board, Board, _InvalidPieces, NewInvalidPieces, 0,
         ↪  GameEnd).
checkGameEnd([],_,InvalidPieces, FinalInvalidPieces,_,_):-
        copyList(InvalidPieces, FinalInvalidPieces).
        %write('Game continues!'),nl.
checkGameEnd([_ | _],_,_,_,_,GameEnd):-
        GameEnd == 1, !.
        %write('The Game has ended!'),nl.
checkGameEnd([Head | Tail], Board, InvalidPieces, FinalInvalidPieces, Row,
 ↪  GameEnd):-
        findPiece(Head, Board, _RowInvalidPieces, ResultInvalidPieces,
         ↪  Row, 0, GameEnd),
        NewRow is Row + 1,
        aggregateInvalidPieces(InvalidPieces, ResultInvalidPieces,
         ↪  NewInvalidPieces),
        %write('checkGameEnd: '), write(NewInvalidPieces),nl,
        checkGameEnd(Tail, Board, NewInvalidPieces, FinalInvalidPieces,
         ↪  NewRow, GameEnd).


/**
* Goes through the invalid pieces and updates the board.
**/
updateBoardAux([], Board, NewBoard):-
        copyList(Board, NewBoard).
        %write('Board Updated'),nl.
updateBoardAux([Row, Col | Tail], Board, NewBoard):-
        nth0(Row, Board, BoardRow),
        nth0(Col, BoardRow, [Letter, Rotation, Color, _Valid]),
        replace(Board,Row,Col,[Letter,Rotation,Color,1], TempBoard),
        updateBoardAux(Tail, TempBoard, NewBoard).
% Verifies if there are Invalid Pieces
updateBoard(InvalidPieces, Board, NewBoard):-
        is_list(InvalidPieces),!,
        updateBoardAux(InvalidPieces, Board, NewBoard).
updateBoard(_InvalidPieces, Board, NewBoard):-
        copyList(Board, NewBoard).
```

7.7 print.pl

```
/**
* Get Rotation
**/
rotatePattern(0, OldPattern, OldPattern).
rotatePattern(1, OldPattern, NewPattern):- % 90 degrees
        transpose(OldPattern, TempPattern),
        maplist(reverse, TempPattern, NewPattern).
rotatePattern(2, OldPattern, NewPattern):- % 180 degrees
        reverse(OldPattern, TempPattern),
        maplist(reverse, TempPattern, NewPattern).
rotatePattern(3, OldPattern, NewPattern):- % 270 degrees
        transpose(OldPattern, TempPattern),
        reverse(TempPattern, NewPattern).


/**
* Get color
**/
getPieceInfo([],_,_).
getPieceInfo([Head | Tail], Color, Valid):-
        Color = Head,
        validSymbol(Tail, Valid).


/**
* Get pattern
**/
getPiecePattern(PieceNum, Pattern, NewPattern):-
        nth0(PieceNum, Pattern, NewPattern).
getPiecePattern(PieceNum, Letter, Pattern):-
        patternLetter(Letter, TempPattern),
        nth0(PieceNum, TempPattern, Pattern).


/**
* Get the symbols of pattern
**/
printEachSymbol([],_,_,_,_).
printEachSymbol([_Head | Tail], 1, 1, Color, Valid):-
        put_char(Valid),
        printEachSymbol(Tail, 2, 1, Color, Valid).
printEachSymbol([Head | Tail], Col, PieceNum, Color, Valid):-
        getSymbol(Head, Color, Char),
        put_code(Char),
        NewCol is Col + 1,
        printEachSymbol(Tail, NewCol, PieceNum, Color, Valid).
printEachSymbol([],_,_).
printEachSymbol([Head | Tail], Color, Valid):-
        getSymbol(Head, Color, Char),
        put_code(Char),
        printEachSymbol(Tail, Color, Valid).


/**
* Print piece symbols
**/
printPieceSymbols(PieceNum, Pattern, Color, Valid):-
```

```
        getPiecePattern(PieceNum, Pattern, NewPattern),
        printEachSymbol(NewPattern, 0, PieceNum, Color, Valid).
printPieceSymbols(PieceNum, Pattern, Color, Valid):-
         getPiecePattern(PieceNum, Pattern, NewPattern),
         getPiecePattern(PieceNum, Pattern, NewPattern),
        printEachSymbol(NewPattern, Color, Valid).


/**
* Main function for print Pieces
**/
printPiece([], _, _).
printPiece(nil, _, _) :-
        write('|    |').
printPiece([_Letter, _Rotation | Tail], Pattern, PieceNum):-
        getPieceInfo(Tail, Color, Valid),
        write('|'),
        printPieceSymbols(PieceNum, Pattern, Color, Valid),
        write('|').


/**
* Get a piece after apply the rotation
**/
getPiece(nil,nil).
getPiece([Letter, Rotation | _Tail], Piece):-
        patternLetter(Letter, Pattern),
        rotatePattern(Rotation, Pattern, Piece).


/**
* Get one-piece row pattern
**/
printRowPieces([],_, _):- nl.
printRowPieces([Head | Tail], Num, PieceNum):-
        getPiece(Head, Piece),
        printPiece(Head, Piece, PieceNum),
        printRowPieces(Tail, Num, PieceNum).


/**
* Print each row of Board
**/
printRow(0,_,_,_).
printRow(2,PieceNum,Row,RowNumber):-
```

```
        NewPieceNum is PieceNum + 1,
        RowNumber < 10,
        write('  '), write(RowNumber), write(' '),
        printRowPieces(Row, 1, NewPieceNum),
        printRow(1, NewPieceNum,Row, RowNumber).
printRow(2,PieceNum,Row,RowNumber):-
        NewPieceNum is PieceNum + 1,
        write(' '), write(RowNumber), write(' '),
        printRowPieces(Row, 1, NewPieceNum),
        printRow(1, NewPieceNum,Row, RowNumber).
printRow(Num,PieceNum,Row, RowNumber):-
        NewNum is Num - 1,
        NewPieceNum is PieceNum + 1,
        write('    '),
        printRowPieces(Row, NewNum, NewPieceNum),
        printRow(NewNum, NewPieceNum,Row,RowNumber).


/**
* Calls the function to print each row
**/
printBoard([], _).
printBoard([Head | Tail], RowCount) :-
        length(Head, ColumnsNum),
        write('   '), printSeparator(ColumnsNum), nl,
        printRow(3,-1,Head, RowCount),
        NewRowCount is RowCount + 1,
        write('   '),printSeparator(ColumnsNum), nl,
        printBoard(Tail, NewRowCount).


/**
* Board numbers and separators
**/
printTopNumbers(_, 0).
printTopNumbers(Count, ColumnsNum):-
        Count == 0,
        write(' | '), write(Count), write(' |'),
        NewCount is Count + 1,
        NewColumnsNum is ColumnsNum - 1,
        printTopNumbers(NewCount, NewColumnsNum).
printTopNumbers(Count, ColumnsNum):-
        Count =< 9,
        write('| '), write(Count), write(' |'),
        NewCount is Count + 1,
        NewColumnsNum is ColumnsNum - 1,
        printTopNumbers(NewCount, NewColumnsNum).
printTopNumbers(Count, ColumnsNum):-
        write('| '), write(Count), write('|'),
        NewCount is Count + 1,
        NewColumnsNum is ColumnsNum - 1,
        printTopNumbers(NewCount, NewColumnsNum).


/**
* Print the horizontal separator of board
**/
printSeparator(0):- write('--').
printSeparator(ColumnsNum):-
        write('-----'),
        NewColumnsNum is ColumnsNum - 1,
        printSeparator(NewColumnsNum).


/**
* Main funtion of print board
**/
printBoardMain([Head | Tail]):-
        nl,length(Head, ColumnsNum),
        write('   '),printTopNumbers(0,ColumnsNum), nl,
        printBoard([Head | Tail], 0), nl.


/**
* Prints row's of available pieces
**/
printAvailablePiecesRow(_, [_ , []]).
printAvailablePiecesRow(PieceRow, [Head , [Head2 | Tail] ]):-
        write(' |'),
        Valid = '0',
        printPieceSymbols(PieceRow, Head2, Head, Valid),
        write('| '),
        printAvailablePiecesRow(PieceRow, [Head, Tail]).


/**
* Separates each piece per line
**/
printAvailablePiecesAux(3, _).
printAvailablePiecesAux(PieceRow, List):-
```

```
        copyList(List, AuxList),
        printAvailablePiecesRow(PieceRow, List),
        NewPieceRow is PieceRow + 1, nl,
        printAvailablePiecesAux(NewPieceRow, AuxList).


/**
* Prints de legends of the available pieces
**/
prepareLegendsPieces([]).
prepareLegendsPieces([Head|Tail]):-
        getCode(Head, Code),
        put_code(Code),
        write('      '),
        prepareLegendsPieces(Tail).


/**
* Main function to print available pieces
**/
printAvailablePieces(PieceRow,  [Head , [Head2 | Tail]]):-
        write('  Available pieces: '), nl, nl,
        write('   '),
        copyList([Head2 | Tail], AuxTail),
        prepareLegendsPieces(AuxTail), nl,
        printAvailablePiecesAux(PieceRow, [Head , [Head2 | Tail]]).


/**
* Print the black pieces turn
**/
printInfoColor(Player):-
        Player == 0, !,
        printSpace(5),write('    _                ___           '), nl,
        printSpace(5),write('   |_) |  _.  _ |     |     ._ ._  '), nl,
        printSpace(5),write('   |_) | (_| (_ |<    | |_| |  | | '), nl,
         ↪  nl, nl.


/**
* Print the white pieces turn
**/
printInfoColor(_Player):-
                printSpace(5),write('                          ___
                 ↪  '), nl,
                printSpace(5),write('   \\   /  |_  o _|_  _    |     ._
                 ↪   ._  '), nl,
                printSpace(5),write('    \\/\\/   | | |  |_ (/_   | |_| |
                 ↪   | | '), nl, nl, nl.


/**
* Informs that the players have draw and that the playing conditions have
 ↪   changed
**/
printInfoDraw :- nl,
        printSpace(5), write(' _              |    ___
         ↪   _    _
         ↪   | '), nl,
        printSpace(5), write('/ \\|_    __  _  |     | |_  _  \\/    __ _
         ↪   __     _    _|_    _ _|_   |_) o  _  _  _  _
         ↪   _|_ o __  _    _|_ _    __  _     _    _|_|_  _ __  | '), nl,
        printSpace(5), write('\\_/| |    | |(_) o     | | |(/_ /     | (_||
         ↪   |   (_)|_| |_   (_) |    |   | (/_(_ (/__>     o  o  o     |_
         ↪   | |||(/_    |_(_)   |||(_)\\_/(/_    |_| |(/_||| o '), nl,
         ↪   nl, sleep(4).


/**
* Informs that someone has won
**/
printInfoWinGame(Player):- nl,
        Player == 0, !,
        printSpace(5), write('   _
         ↪   '), nl,
        printSpace(5), write(' |_)  |   _.   _  |     \\    /  o  ._    _
         ↪   |  |  | '), nl,
        printSpace(5), write(' |_)  |  (_|  (_  |<     \\/\\/   |  | |  _>
         ↪   o  o  o '), nl, nl, sleep(4).

printInfoWinGame(_Player):- nl,
        printSpace(5), write(' \\    /  |_   o  _|_   _     \\    /  o  ._
         ↪   _  |  |  | '), nl,
        printSpace(5), write('  \\/\\/   | |  |   |_  (/_     \\/\\/   |
         ↪   | |  _>  o  o  o '), nl, nl, sleep(4).

printInfoWinGame2(Type):- nl,
        Type == 'COMPUTER', !,
        printSpace(5), write('   _
         ↪   '), nl,
        printSpace(5), write(' /    _   ._ _   ._        _|_   _   ._
         ↪  \\    /  o  ._    _  |  |  | '), nl,
        printSpace(5), write(' \\_  (_)  | | |  |_)  |_|   |_  (/_  |
         ↪  \\/\\/   |  | |  _>  o  o  o '), nl,
        printSpace(5), write('                 |
         ↪  '), nl, nl, sleep(4).

printInfoWinGame2(_Type):- nl,
        printSpace(5), write(' |_|       ._ _    _.  ._     \\    /  o  ._
         ↪  _  |  |  | '), nl,
        printSpace(5), write(' | |  |_|  | | |  (_|  | |     \\/\\/   |  |
         ↪  |  _>  o  o  o '), nl, nl, sleep(4).


/**
* Prints information about changes to the board status
**/
printInformation(Letter) :-
        write('-> Computer played piece '), write(Letter), nl, nl.

printInformation(NumRow, NumCol, Letter) :-
        write('-> Computer played piece '), write(Letter), write(' in ('),
        write(NumRow), write(','), write(NumCol), write(')'), nl, nl.

printInformation(SourceRow, SourceColumn, DestRow, DestColumn) :-
        write('-> Computer removes piece from position ('),
        write(SourceRow), write(','), write(SourceColumn), write(')'),
         ↪  sleep(2),
        write('... and puts it'), write(' in position ('),
        write(DestRow), write(','), write(DestColumn), write(')'), nl, nl,
         ↪  sleep(2).


/**
* Auxiliary function that prints a certain number of spaces
**/
printSpace(0).
printSpace(Value):-
        write(' '),
        NewValue is Value - 1,
        printSpace(NewValue).


/**
```

```
* Prints the color of the player, the board and available pieces at each
 ↪  game cycle
**/
printInfoGame(Board, ColorPlayer, Pieces) :-
        printInfoColor(ColorPlayer), sleep(1),
        printAvailablePieces(0, [ColorPlayer, Pieces]), sleep(1),
        printBoardMain(Board), nl, sleep(3).

% After draw
printInfoGame(Board, ColorPlayer) :-
        printInfoColor(ColorPlayer), sleep(1),
        printBoardMain(Board), nl, sleep(3).


/**
* Prints the game menu
**/
printMenuScreen :- nl, nl,
        printSpace(20),
         ↪  write('********************************************************************'),nl,
        printSpace(20), write('*
         ↪  --------------------------------------------------------------
         ↪  *'),nl,
        printSpace(20), write('* |
         ↪  | *'),nl,
        printSpace(20), write('* |         .__   __.  __
         ↪  __   __    __           | *'),nl,
        printSpace(20), write('* |         |  \\ |  | |  |
         ↪  |  | |  |  |  |        | *'),nl,
        printSpace(20), write('* |         |   \\|  | |  |  ______
         ↪  |  | |  |  |  |        | *'),nl,
        printSpace(20), write('* |         |  . `  | |  | |______| .--.  |
         ↪  | |  |  |  |        | *'),nl,
        printSpace(20), write('* |         |  |\\   | |  |          |
         ↪  `--\'  | |  \`--\'  |       | *'),nl,
        printSpace(20), write('* |         |__| \\__| |__|
         ↪  \\______/   \\______/         | *'),nl,
        printSpace(20), write('* |
         ↪  | *'),nl,
        printSpace(20), write('* |                      **** WELCOME !
         ↪  ****                     | *'),nl,
        printSpace(20), write('* |
         ↪  | *'),nl,
        printSpace(20), write('* |                      Human vs Human:
         ↪  | *'),nl,
        printSpace(20), write('* |                         1.  Level I
         ↪   | *'),nl,
        printSpace(20), write('* |
         ↪   | *'),nl,
        printSpace(20), write('* |                      Human vs Computer:
         ↪   | *'),nl,
        printSpace(20), write('* |                         2.  Level I
         ↪   | *'),nl,
        printSpace(20), write('* |                         3.  Level II
         ↪   | *'),nl,
        printSpace(20), write('* |
         ↪   | *'),nl,
        printSpace(20), write('* |                   Computer vs Computer:
         ↪   | *'),nl,
        printSpace(20), write('* |                         4.  Level I
         ↪   | *'),nl,
        printSpace(20), write('* |                         5.  Level II
         ↪   | *'),nl,
        printSpace(20), write('* |
         ↪   | *'),nl,
        printSpace(20), write('* |                         6.  Exit
         ↪   | *'),nl,
        printSpace(20), write('* |
         ↪   | *'),nl,
        printSpace(20), write('* |
         ↪   | *'),nl,
        printSpace(20), write('* |                    Ana Santos
         ↪   up200700742                    | *'),nl,
        printSpace(20), write('* |                 Margarida Silva
         ↪   up201505505                  | *'),nl,
        printSpace(20), write('*
         ↪   |______________________________________________________________|
         ↪   *'),nl,
        printSpace(20), write('*
         ↪   *'),nl,
        printSpace(20),
         ↪   write('********************************************************************'),nl.


/**
* Prints the game loading animation
**/
printLoad :-
        clearScreen,
        printSpace(10), write('[
         ↪  ]   0 % '), nl, nl, sleep(0.2), clearScreen,
        printSpace(10), write('[=====
         ↪  ]  10 %'), nl, nl, sleep(0.2), clearScreen,
        printSpace(10), write('[==========
         ↪  ]  20 %'), nl, nl, sleep(0.2), clearScreen,
        printSpace(10), write('[===============
         ↪  ]  30 %'), nl, nl, sleep(0.2), clearScreen,
        printSpace(10), write('[====================
         ↪  ]  40 %'), nl, nl,sleep(0.2), clearScreen,
        printSpace(10), write('[=========================
         ↪  ]  50 %'), nl, nl, sleep(0.2), clearScreen,
        printSpace(10), write('[==============================
         ↪  ]  60 %'), nl, nl, sleep(0.2), clearScreen,
        printSpace(10), write('[===================================
         ↪  ]  70 %'), nl, nl, sleep(0.2), clearScreen,
        printSpace(10), write('[========================================
         ↪  ]  80 %'), nl, nl, sleep(0.2), clearScreen,
        printSpace(10),
         ↪  write('[=============================================     ]
         ↪  90 %'), nl, nl, sleep(0.2), clearScreen,
        printSpace(10),
         ↪  write('[==================================================]
         ↪  100 %'), nl, nl, sleep(0.5).
```

7.8 utils.pl

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

:-include('print.pl').
:-include('humanInput.pl').
:-include('computerInput.pl').
:-include('board.pl').
:-include('gameStatus.pl').
:-include('ai.pl').

:- discontiguous game2Players/6, gameHumanVsComputer/4,
 ↪  gameHumanVsComputer/7, game2Players/6, gameComputerVsComputer/7,
 ↪  gameComputerVsComputer/4.

/**
* Dictionary of pieces
```

```
**/
% Get pattern of piece
patternLetter(a, [[1,1,1],[0,0,1],[0,0,0]]).
patternLetter(b, [[1,1,1],[0,0,0],[0,0,1]]).
patternLetter(c, [[1,1,1],[0,0,0],[0,1,0]]).
patternLetter(d, [[1,1,1],[0,0,0],[1,0,0]]).
patternLetter(e, [[1,1,1],[1,0,0],[0,0,0]]).
patternLetter(f, [[1,1,0],[0,0,1],[0,0,1]]).
patternLetter(g, [[1,1,0],[0,0,1],[0,1,0]]).
patternLetter(h, [[1,1,0],[0,0,1],[1,0,0]]).
patternLetter(i, [[1,1,0],[1,0,1],[0,0,0]]).
patternLetter(j, [[1,1,0],[0,0,0],[0,1,1]]).
patternLetter(k, [[1,1,0],[0,0,0],[1,0,1]]).
patternLetter(l, [[1,1,0],[1,0,0],[0,0,1]]).
patternLetter(m, [[1,1,0],[0,0,0],[1,1,0]]).
patternLetter(n, [[1,1,0],[1,0,0],[0,1,0]]).
patternLetter(o, [[1,0,0],[0,0,1],[1,0,1]]).
patternLetter(p, [[1,0,0],[1,0,1],[0,0,1]]).
patternLetter(q, [[1,0,0],[0,0,1],[1,1,0]]).
patternLetter(r, [[1,0,0],[1,0,1],[0,1,0]]).
patternLetter(s, [[0,1,0],[1,0,1],[0,1,0]]).
patternLetter(t, [[1,0,1],[0,0,0],[1,0,1]]).


% Get symbols to print patterns of pieces
getSymbol(0, 0, 178). % black piece without square
getSymbol(1, 0, 254). % black piece with square
getSymbol(0, 1, 176). % white piece without square
getSymbol(1, 1, 254). % white piece with square

% Get code of piece
getCode(a,97).
getCode(b,98).
getCode(c,99).
getCode(d,100).
getCode(e,101).
getCode(f,102).
getCode(g,103).
getCode(h,104).
getCode(i,105).
getCode(j,106).
getCode(k,107).
getCode(l,108).
getCode(m,109).
getCode(n,110).
getCode(o,111).
getCode(p,112).
getCode(q,113).
getCode(r,114).
getCode(s,115).
getCode(t,116).

% Get valid symbol
validSymbol(0, 'V'). % válida
validSymbol(1, 'I'). % inválida
validSymbol([Head | _], Valid):- validSymbol(Head, Valid).

% Get color Player
getColorPlayer(1,'WHITE').
getColorPlayer(0,'BLACK').
getTypePlayer(1, ' HUMAN  ').
getTypePlayer(0, 'COMPUTER').

% Clone a list
copyList(L,R) :- accCp(L,R).
accCp([],[]).
accCp([H|T1],[H|T2]) :- accCp(T1,T2).

% Interception of two lists
inter([], _, []).
inter([H1|T1], L2, [H1|Res]) :-
    member(H1, L2),
    inter(T1, L2, Res).
inter([_|T1], L2, Res) :-
    inter(T1, L2, Res).

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