U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Frozen Forest

Relatório Final

Programação em Lógica

(18 de Novembro de 2018)

Frozen Forest 4

Bruno Miguel da Silva Barbosa de Sousa **up201604145**@fe.up.pt

João Manuel Angélico Gonçalves **up201604245**@fe.up.pt

Índice

1. Introdução

Os objetivos deste trabalho foram implementar, na linguagem Prolog, um jogo de tabuleiro para dois jogadores e desenvolver uma aplicação para o jogar baseada numa interface de texto para o utilizador. Para além disso, o objetivo foi a possibilidade de o jogo possuir três modos de utilização: Humano/Humano, Humano/Computador, Computador/Computador, em que o computador tem dois níveis de jogo diferentes.

O jogo escolhido pelo nosso grupo foi o "Frozen Forest".

2. O Jogo "Frozen Forest"

Frozen Forest é um jogo de tabuleiro desenvolvido por **Néstor Romeral Andrés** com inspiração num jogo de Silvia Romeral Andrés.

Consiste num jogo de estratégia para 2 jogadores, que vão movimentando a sua peça alternadamente, até que não haja nenhum movimento possível, ou até que o tabuleiro não contenha árvores.

Enquanto que um jogador está a esconder-se (Mina), o outro está a tentar procurá-lo (Yuki). O espaço para onde quem procura se movimenta, vai imediatamente transformar-se num espaço vazio, de forma a diminuir os esconderijos possíveis para o outro jogador, sendo que a Mina está escondida se tiver um obstáculo (árvore) entre a linha de visão desta e de Yuki.

Yuki e Mina têm uma linha de visão clara se não houver nenhuma árvore na linha reta que conecta a localização dos 2 jogadores, sendo que a árvore aonde Mina se encontra não conta como um bloqueio da linha de visão.

O jogo vai ser organizado num tabuleiro com 10x10 espaços, com uma peça para cada jogador (a letra "m" a representar Mina e a letra "y" a representar Yuki), várias peças com a letra "w" que representam um espaço vazio (sem árvores ou jogadores), e vai ser completado por peças com letra "t" que vão corresponder á localização das árvores. Cada peça ocupa um espaço no tabuleiro.

Preparação

Inicialmente, os jogadores vão escolher o número de árvores presentes no tabuleiro (10x10, 9x9, …), sendo que estas vão estar centradas no tabuleiro e, o resto do tabuleiro vai ser completado com espaços vazios.

De seguida, o jogador que está a controlar o Yuki (quem está a procurar) começa por colocar a sua peça em qualquer espaço no tabuleiro com uma árvore e depois, o outro jogador coloca a sua peça num espaço válido (fora da linha de visão do Yuki).

Objetivo

O objetivo deste jogo é fazer com que o outro jogador não tenha nenhuma casa possível para onde se movimentar ou, para Yuki, este também pode optar por tentar comer todas as árvores.

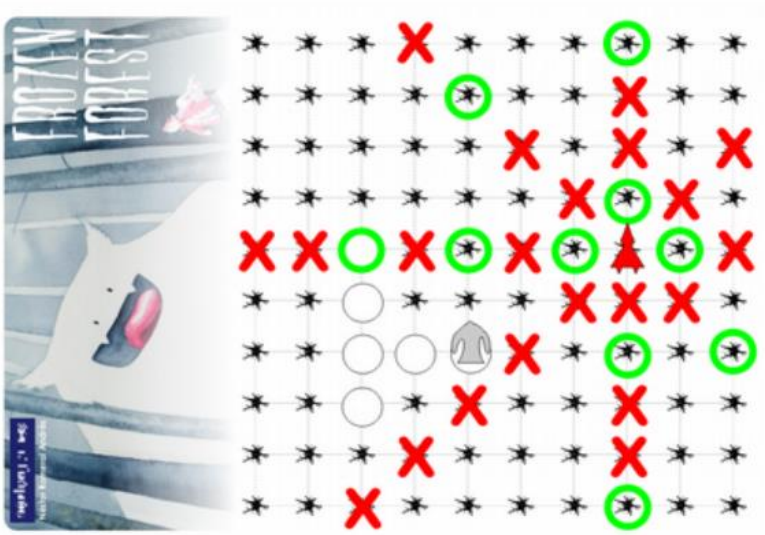
Movimentação de quem procura (Yuki):

Relativamente á movimentação de quem procura, este só se vai poder deslocar uma casa ortogonalmente ou diagonalmente, sendo que este movimento também terá de ser para uma peça que contenha uma árvore e que tenha uma linha de visão clara para Mina. À medida que este vai jogando, o número de casas vazias vai aumentando visto que Yuki vai comendo as árvores.

(Exemplo em que os movimentos válidos de Yuki (peça cinzenta) estão marcados com verde, os locais sem uma linha de visão clara para Mina (peça vermelha) estão representados por uma cruz vermelha, enquanto que os locais vazios estão marcados com uma cruz roxa.)

Movimentação de quem se esconde (Mina):

Quanto á movimentação do jogador que se esconde, este poderá movimentar-se o número de casas que desejar numa linha ortogonal ou diagonal. A única outra restrição no movimento de Mina é que apenas se pode deslocar para um espaço com a linha de visão bloqueada para Yuki.



(Exemplo em que os movimentos válidos de Mina (peça vermelha) estão marcados com verde e os locais com uma linha de visão clara para Yuki (peça cinzenta) estão identificados com uma cruz vermelha).

Fim

Tal como referido anteriormente, um jogador perde o jogo assim que não tenha nenhuma movimentação possível, aquando a sua vez de jogar.

Mas, quando o tabuleiro já estiver sem árvores, se nenhum jogador tiver perdido ate lá, o vencedor é o jogador que estava a procurar (Yuki).

No caso de haver 2 jogos, em que os 2 jogadores alternaram a sua função:

- Se ambos os jogadores ganharam com o Yuki, ganha o que comeu menos árvores.
- Se ambos os jogadores ganharam com a Mina, ganha o que comeu mais árvores.
- Se um jogador ganhar os 2 jogos é claramente o vencedor.

3.Lógica do Jogo

3.1. Representação do estado do jogo

O tabuleiro será representado utilizando listas de listas (10x10) tendo os seguintes atómos como peças:

- t – Árvore
- w – Neve (Sitio onde foi comida árvore)
- y – Yuki
- m – Mina

Para além da representação do tabuleiro, será também guardado:

- O número de vitórias que cada jogador tem.
- O número de árvores que cada jogador comeu jogando como Yuki (para desempate)
- O "personagem" que cada jogador representa (Yuki/Mina)
- O próximo jogador a jogar (p1/p2)
- A personagem com que cada jogador ganhou
- A dificuldade de cada jogador (caso sejam computador).

Representação em Prolog

Estado Inicial

```
wins(0,0).
treesEaten(0,0).
players(y,m).
tab([[t,t,t,t,t,t,t,t,t,t],
    [t,t,t,t,t,t,t,t,t,t],
    [t,t,t,t,t,t,t,t,t,t],
    [t,t,t,t,t,t,t,t,t,t],
    [t,t,t,t,t,t,t,t,t,t],
    [t,t,t,t,t,t,t,t,t,t],
    [t,t,t,t,t,t,t,t,t,t],
    [t,t,t,t,t,t,t,t,t,t],
    [t,t,t,t,t,t,t,t,t,t],
    [t,t,t,t,t,t,t,t,t,t]]).
nextPlayer(p1).
```

No início, todas as "casas" do tabuleiro serão árvores, e será o jogador1 (como Yuki) a jogar.

Estado Intermédio

```
wins(0,0).
treesEaten(8,0).
players(y,m).
tab([[t,t,t,t,t,t,t,t,t,t],
    [t,t,t,t,t,t,t,t,t,t],
    [t,t,t,t,t,t,t,t,t,t],
    [t,t,t,w,w,w,w,t,t,t],
    [t,t,t,t,t,t,w,t,t,t],
    [t,t,t,t,t,t,w,t,t,t],
    [t,t,t,t,t,t,w,y,t,t],
    [t,t,t,t,t,t,t,t,t,t],
    [t,t,t,t,t,t,t,t,m,t],
    [t,t,t,t,t,t,t,t,t,t]]).
nextPlayer(p2).
```

Ainda no primeiro jogo, após 8 jogadas, o Yuki comeu oito árvores (contando com o espaço onde está atualmente). Dado que tem linha de visão sobre a Mina ainda não perdeu. É a vez de a Mina jogar, terá que se esconder do Yuki.

Estado Final

```
wins(0,1).
treesEaten(40,32).
players(m,y).
tab([[t,t,t,t,t,w,t,w,t,t],
    [t,t,w,t,w,t,w,t,w,t],
    [t,w,t,w,t,w,w,w,t,t],
    [t,t,w,t,w,w,y,w,w,t],
    [t,t,t,w,t,w,w,w,t,t],
    [t,t,w,t,t,t,t,t,w,t],
    [t,w,t,m,t,t,t,w,t,t],
    [t,t,w,t,w,t,w,t,w,t],
    [t,t,t,w,t,w,t,w,t,t],
    [t,t,t,t,t,t,t,t,t,t]]).
nextPlayer(p2).
```

No primeiro jogo, a Mina acabou por conseguir escapar do Yuki, tendo o Yuki comido 40 árvores. Neste segundo jogo, é a vez do Yuki jogar. Dado que está preso (nenhuma casa à sua volta é uma árvore), perdeu o jogo. Sendo assim, o jogo ficaria empatado 1-1, sendo necessário recorrer ao número de árvores comida por cada jogador para desempatar. Como ganharam os dois jogadores jogando com a Mina, ganha quem comeu mais árvores. Assim sendo, o jogador 1 foi o vencedor.

3.2. Visualização do tabuleiro

Implementação *display_game(+Board, +Player)*

```
display_game(Board,Player):-
    wins(W1,W2),
    format('~nWins: ~d-~d~n',[W1,W2]),
    treesEaten(T1,T2),
    format('Trees eaten: ~d-~d~n~n',[T1,T2]),
    display_board(0,Board),
    format('~n~nPlayer to move: ~p ',Player),
    display_player(Player),
    write('\n').

display_board(Counter,[Head]):-
    write('                                    \n'),
    format('~d ',Counter),
    display_line(Head),
    write('\n\n   a  b  c  d  e  f  g  h  i  j ').

display_board(Counter,[Head|Tail]):-
    write('                                    \n'),
    format('~d ',Counter),
    display_line(Head),
    write('\n'),
    Next is Counter+1,
    display_board(Next,Tail).
```

```
display_line([Head]):-
    (Head = w,
    write('    '));
    format(' ~p ',Head).

display_line([Head|Tail]):-
    (
        (Head = w,
        write('    '));

        format(' ~p ',Head)
    ),

    display_line(Tail).

display_player(Player):-
    players(P1,P2),
    (
        (Player=p1,
        write_name(P1));

        (Player=p2,
        write_name(P2))
    ).
```

Output produzido:

```
0  t  t  t  t  t  t  t  t  t  t
1  t  t  t  t  t  t  t  t  t  t
2  t  t  t  t  t  m  t  t  t  t
3  t  t  t  t  t  t  t  t  t  t
4  t  t  t  t  t  t  t  t  t  t
5  t  t  t  t  t  t  t  t  t  t
6  t  t  t  y  t  t  t  t  t  t
7  t  t  t  t  t  t  t  t  t  t
8  t  t  t  t  t  t  t  t  t  t
9  t  t  t  t  t  t  t  t  t  t
   a  b  c  d  e  f  g  h  i  j
Player to move: p1 playing as Yuki
```

3.3. Lista de jogadas válidas

Implementação *valid_moves(+Board, +Player, -ListOfMoves)*

```
valid_moves(Board, Player, ListOfMoves):-
    players(P1,P2),
    (
        (Player = p1,
        Name = P1);

        (Player = p2,
        Name = P2)
    ),
    (
        (Name=y,
        valid_moves_yuki(Board, ListOfMoves),
        !);

        (Name=m,
        valid_moves_mina(Board, ListOfMoves),
        !)
    ).
```

Dado que a lógica de jogadas válidas é diferente para os dois personagens, foram criadas duas funções separadas para o cálculo de movimentos válidos de acordo com a personagem do jogador.

```
valid_moves_yuki(Board, ListOfMoves):-
    yuki(X,Y),
    (
        (X =:= -1,
        Y =:= -1,
        allMoves(Moves),
        ListOfMoves = Moves);

        (LastX is X - 1,
        LastY is Y - 1,
        NextX is X + 1,
        NextY is Y + 1,
        valid_move_yuki(Board, LastX, LastY, [], Moves1),
        valid_move_yuki(Board, LastX, Y, Moves1, Moves2),
        valid_move_yuki(Board, LastX, NextY, Moves2, Moves3),
        valid_move_yuki(Board, X, LastY, Moves3, Moves4),
        valid_move_yuki(Board, X, NextY, Moves4, Moves5),
        valid_move_yuki(Board, NextX, LastY, Moves5, Moves6),
        valid_move_yuki(Board, NextX, Y, Moves6, Moves7),
        valid_move_yuki(Board, NextX, NextY, Moves7, ListOfMoves))
    ).
```

Para os movimentos do Yuki, caso esta seja a primeira jogada, ele poderá mover-se para onde quiser. Caso contrário apenas poderá mover-se para "o quadrado" envolta da sua posição atual.

```
valid_move_yuki(Board, X, Y, Moves, NewMoves):-
    (
        (X > -1,
        X < 10,
        Y > -1,
        Y < 10,
        Line is X + 1,
        Col is Y + 1,
        getPeca(Line, Col, Board, Peca),
        Peca = t,
        mina(MX,MY),
        canSee(X,Y,MX,MY,Board),
        append(Moves,[[X,Y]],NewMoves));

        (NewMoves = Moves)
    ).
```

Nesse "quadrado" apenas se poderá mover para posições nas quais consiga ver a Mina.

```
valid_moves_mina(Board, ListOfMoves):-
    mina(X,Y),
    (
        (X =:= -1,
        Y =:= -1,
        allMoves(Moves),
        checkSeen(Board,Moves,[],ListOfMoves));

        (LastX is X - 1,
        LastY is Y - 1,
        NextX is X + 1,
        NextY is Y + 1,
        valid_move_mina(Board, LastX, LastY, [], Moves1, -1, -1),
        valid_move_mina(Board, LastX, Y, Moves1, Moves2, -1, 0),
        valid_move_mina(Board, LastX, NextY, Moves2, Moves3, -1, 1),
        valid_move_mina(Board, X, LastY, Moves3, Moves4, 0, -1),
        valid_move_mina(Board, X, NextY, Moves4, Moves5, 0 , 1),
        valid_move_mina(Board, NextX, LastY, Moves5, Moves6, 1, -1),
        valid_move_mina(Board, NextX, Y, Moves6, Moves7, 1, 0),
        valid_move_mina(Board, NextX, NextY, Moves7, ListOfMoves, 1, 1))
    ).
```

Para os movimentos da Mina, caso esta seja a primeira jogada, ela poderá mover-se para onde quiser (desde que não seja vista pelo Yuki). Caso contrário apenas poderá mover-se na ortogonal e nas diagonais principais.

```
checkSeen(_,[],ValidMoves,ValidMoves).


checkSeen(Board,[Head|Tail],ListOfMoves,ValidMoves):-
    [X,Y] = Head,
    yuki(YX,YY),
    (
        (X =:= YX,
        Y =:= YY,
        checkSeen(Board,Tail,ListOfMoves,ValidMoves));

        (canSee(YX,YY,X,Y,Board),
        checkSeen(Board,Tail,ListOfMoves,ValidMoves));

        (append(ListOfMoves,[Head],MoreMoves),
        checkSeen(Board,Tail,MoreMoves,ValidMoves))
    ).
```

No caso de ser a primeira jogada, será necessário percorrer todos as posições do tabuleiro e verificar quais o Yuki não consegue ver.

```
valid_move_mina(Board, X, Y, Moves, NewMoves, DX, DY):-
    (
        (X > -1,
        X < 10,
        Y > -1,
        Y < 10,
        yuki(YX,YY),
        (
            (X =:= YX,
            Y =:= YY,
            MoreMoves = Moves);

            (canSee(YX,YY,X,Y,Board),
            MoreMoves = Moves);

            (append(Moves,[[X,Y]],MoreMoves))
        ),
        NextX is X + DX,
        NextY is Y + DY,
        valid_move_mina(Board, NextX, NextY, MoreMoves, NewMoves, DX, DY));

        (NewMoves = Moves)
    ).
```

Um movimento apenas é válido se o Yuki não conseguir ver essa posição. Esta função é chamada em todas as direções possíveis e chama recursivamente a mesma direção até atingir o fim do tabuleiro.

3.4. Execução de Jogadas

Implementação *move(+Move, +Board, -NewBoard)*

```
move(Move,Board,NewBoard):-
    [X,Y] = Move,
    players(P1,P2),
    nextPlayer(Player),
    valid_moves(Board, Player, Moves),
    member(Move,Moves),
    (
        (Player=p1,
        Name = P1);

        (Player=p2,
        Name = P2)
    ),
    Line is X + 1,
    Col is Y + 1,
    (
        (Name = y,
        moveYuki(Player,Line,Col,Board,NewBoard));

        (Name = m,
        moveMina(Line,Col,Board,NewBoard))
    ),
    retract(nextPlayer(Player)),
    (
        (Player=p1,
        assert(nextPlayer(p2)));

        (Player=p2,
        assert(nextPlayer(p1)))
    ).
```

O primeiro passo na execução de um movimento é verificar se ele é válido. Se for, pode ser dependente do jogador, um movimento do Yuki ou da Mina. Dado que a lógica dos seus movimentos é diferente, a função move separa-se em duas.

```
moveMina(Line,Col,Board,NewBoard):-
    mina(X,Y),
    beforeMina(Before),
    (
        (X < 0,
        Y < 0,
        getPeca(Line,Col,Board,After),
        setPeca(Line,Col,m,Board,NewBoard));

        (OldX is X + 1,
        OldY is Y + 1,
        setPeca(OldX,OldY,Before,Board,Next),
        getPeca(Line,Col,Board,After),
        setPeca(Line,Col,m,Next,NewBoard))
    ),
    retract(mina(X,Y)),
    retract(beforeMina(Before)),
    NewX is Line - 1,
    NewY is Col - 1,
    assert(mina(NewX,NewY)),
    assert(beforeMina(After)).
```

No movimento da Mina, é necessário verificar o que estava na sua posição antes, colocar lá, guardar o que está na sua próxima posição, movê-la para lá e guardar a sua nova posição.

```
moveYuki(Player,Line,Col,Board,NewBoard):-
    yuki(X,Y),
    treesEaten(T1,T2),
    (
        (X < 0,
        Y < 0,
        setPeca(Line,Col,y,Board,NewBoard));

        (OldX is X + 1,
        OldY is Y + 1,
        setPeca(OldX,OldY,w,Board,Next),
        setPeca(Line,Col,y,Next,NewBoard))
    ),
    retract(yuki(X,Y)),
    retract(treesEaten(T1,T2)),
    NewX is Line - 1,
    NewY is Col - 1,
    assert(yuki(NewX,NewY)),
    (
        (Player = p1,
        NewT is T1 + 1,
        assert(treesEaten(NewT,T2)));

        (Player = p2,
        NewT is T2 + 1,
        assert(treesEaten(T1,NewT)))
    ).
```

No movimento do Yuki, é necessário colocá-lo na posição, atualizar a sua posição e aumentar o número de árvores comidas.

3.5. Final do Jogo

Implementação *game_over(+Board, -Winner)*

```
game_over(Board,Winner):-
    nextPlayer(Player),
    valid_moves(Board,Player,Moves),
    length(Moves,L),
    L =:= 0,
    retract(wonAs(_)),
    (
        (Player = p1,
        wins(W1,W2),
        NewWin is W2 + 1,
        retract(wins(W1,W2)),
        assert(wins(W1,NewWin)),
        players(_,P2),
        assert(wonAs(P2)),
        Winner = p2);

        (Player = p2,
        wins(W1,W2),
        NewWin is W1 + 1,
        retract(wins(W1,W2)),
        assert(wins(NewWin,W2)),
        players(P1,_),
        assert(wonAs(P1)),
        Winner = p1)
    ).
```

O jogo apenas acaba quando um dos jogadores não tiver movimentos possíveis no seu turno. Caso isso acontece, o outro jogador ganha.

```
match_over(Winner):-
    wins(W1,W2),
    Wins is W1 + W2,
    Wins =:= 2,
    (
        (
            W1 > W2,
            Winner = p1
        );

        (
            W1 < W2,
            Winner = p2
        );

        (solve_tie(Winner))
    ).
```

Para além do game_over, dado que em cada jogo são jogados dois "jogos", em que os jogadores jogam como personagens diferentes, implementamos também a função match_over(-Winner), que verifica, no fim de cada "jogo" se já foram jogados os dois "jogos". Caso isso aconteça, se um jogador tiver duas vitórias será o vencedor. Caso tenham ambos uma vitória então é necessário desempatá-los.

```
solve_tie(Winner):-
    treesEaten(T1,T2),
    wonAs(Name),
    (
        (Name = y,
        solve_Yuki_tie(T1,T2,Winner));

        (Name = m,
        solve_Mina_tie(T1,T2,Winner))
    ).
```

Caso os jogadores estejam empatados, então o empate será resolvido pelo número de árvores comidas de acordo com o personagem com que ambos os jogadores ganharam. Se ganharam como o Yuki, ganha o jogador com menos árvores comidas. Caso ganhem como a Mina, ganha o jogador com mais árvores comidas. Caso ambos tenham comido o mesmo número de árvores, o jogo acaba empatado.

3.6. Avaliação do Tabuleiro

Implementação *value(+Board, +Player, -Value)*

```
value(Board,Player,Value):-
    players(P1,P2),
    (
        (Player = p1,
        Name = P1,
        NextPlayer = p2);

        (Player = p2,
        Name = P2,
        NextPlayer = p1)
    ),
    valid_moves(Board,NextPlayer,Moves),
    (
        (Moves = [],
        Value is 9999999);

        (length(Moves,Length),
        reachableTrees(Board,Trees),
        treesAround(Board,Around),
        distance(Distance),
        (
            (Name = y,
            Value is (100 * Trees) + (50 * Around) - (10 * Distance) - Length);

            (Name = m,
            Value is - (100 * Trees) - (50 * Around) + (10 * Distance) - Length)
        ))
    ).
```

Dado que o objetivo do jogo é impedir o outro jogador de jogar, será necessário calcular o número de jogadas válidas do jogador adversário. Para além disso, será também importante verificar se o Yuki não fica preso (calculando o número de árvores as quais ele consegue aceder durante o resto do jogo), não fica bloqueado (quantas árvores tem á sua volta, dado que árvores na diagonal são mais importantes), e a distância à Mina. Para a Mina será importante avaliar o mesmo, mas enquanto que para o Yuki é bom ter muitas árvores possíveis de aceder, é bom ter muitas árvores à sua volta e é bom estar perto da Mina, para a Mina será o contrário.

3.7. Jogada de Computador

Implementação *choose_move(+Board, +Level, -Move)*

```
choose_move(Board,Level,Move):-
    nextPlayer(Player),
    valid_moves(Board,Player,Moves),
    random_shuffle(Moves,[],Random),
    yuki(YX,YY),
    mina(MX,MY),
    beforeMina(Before),
    best(Board,Level,Player,Random,Move,-10000000,_,_,_,_),
    retract(yuki(_,_)),
    retract(mina(_,_)),
    retract(beforeMina(_)),
    assert(yuki(YX,YY)),
    assert(mina(MX,MY)),
    assert(beforeMina(Before)).
```

Ao escolher um movimento, o computador verifica os movimentos válidos, põe-nos numa ordem aleatória (para os jogos não serem sempre iguais) e guarda os predicados dinâmicos que serão alterados ao simular os movimentos (as posições das personagens), escolhe o melhor movimento e volta a colocar as posições guardadas.

```
best(Board,Level,Player,[Head|Tail],Move,Max,MaxMove,MaxValue,MaxBoard,FinalBoard):-
    (
        (Level =:= 1,
        !,
        yuki(YX,YY),
        mina(MX,MY),
        beforeMina(Before),
        simulateValue(Board,Player,Head,Value,NewBoard),
        retract(yuki(_,_)),
        retract(mina(_,_)),
        retract(beforeMina(_)),
        assert(yuki(YX,YY)),
        assert(mina(MX,MY)),
        assert(beforeMina(Before)));
```

Para escolher o melhor movimento, caso o nível seja fácil, ele apenas simula a jogada, verifica o seu valor e vê se é maior do que o valor encontrado até agora.

```
        (yuki(YX,YY),
        mina(MX,MY),
        beforeMina(Before),
        simulateValue(Board,Player,Head,NextValue,NextBoard),
        (
            (NextValue =:= 9999999,
            Value is 9999999);

            ((
                (Player = p1,
                NextPlayer = p2);

                (Player = p2,
                NextPlayer = p1)
            ),
            NewLevel is Level - 2,
            bestNext(NextBoard,NewLevel,NextPlayer,NextNextValue,NextNextBoard),
            (
                (NextNextValue =:= 9999999,
                Value is -9999999);

                (bestNext(NextNextBoard,NewLevel,Player,Value,NewBoard))
            ))
        ),
        retract(yuki(_,_)),
        retract(mina(_,_)),
        retract(beforeMina(_)),
        assert(yuki(YX,YY)),
        assert(mina(MX,MY)),
        assert(beforeMina(Before)))
    ),
```

Caso o nível seja difícil ele irá simular o seu movimento, o movimento do adversário e o seu próximo movimento. Caso o valor deste último movimento seja melhor do que todos os outros até agora, este movimento será o melhor até agora.

```
(
    (Value >= Max,
    !,
    best(Board,Level,Player,Tail,Move,Value,Head,MaxValue,NewBoard,FinalBoard));

    (best(Board,Level,Player,Tail,Move,Max,MaxMove,MaxValue,MaxBoard,FinalBoard))
).
```

Caso o valor do movimento seja melhor que o máximo até agora, este será o melhor movimento.

4. Conclusões

Este projeto teve como objetivo desenvolver um jogo de tabuleiro usando os conhecimentos adquiridos na unidade curricular de Programação em Lógica.

Neste trabalho fomos superando as dificuldades que fomos encontrando, nomeadamente à avaliação do tabuleiro.

Relativamente às melhorias ao trabalho desenvolvido, a avaliação do tabuleiro poderia ser melhorada. Dado que o nosso jogo está divido em dois jogos com os jogadores a jogarem em personagens diferentes trocando no fim do primeiro jogo, e que a estratégia do segundo jogo depende do resultado do primeiro, seriam necessárias seis funções diferentes para o cálculo do valor do tabuleiro:

- Primeiro jogo – Yuki
- Primeiro jogo – Mina
- Segundo jogo – Yuki, tendo ganho o primeiro jogo
- Segundo jogo – Yuki, tendo perdido o primeiro jogo
- Segundo jogo – Mina, tendo ganho o primeiro jogo
- Segundo jogo – Mina, tendo perdido o primeiro jogo

Apesar disso, apenas desenvolvemos duas funções de avaliação de tabuleiro, uma para o Yuki e outra para a Mina, pois pensamos não estar no âmbito deste trabalho ir tão longe na parte de inteligência artificial e porque não tivemos o tempo necessário para as desenvolver e testar.

Em suma, estamos satisfeitos com o resultado deste trabalho e consideramos que este projeto contribuiu para uma melhor compreensão a uma linguagem diferente à que estamos habituados, sendo esta Prolog, e principalmente para a introdução a um novo paradigma de programação, linguagens declarativas.

Bibliografia

Rulebook.

SICStus Prolog User's Manual.

Bratko, I. (2011). Prolog Programming for Artificial Intelligence. 4th Edition. Pearson Education Canada.