

7th Guest: Infection

Relatório Final



Mestrado Integrado em Engenharia Informática e Computação

Programação em Lógica

Grupo 02:

Diogo Machado - up201706832

Eduardo Ribeiro - up201705421

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

16 de Novembro de 2019

Resumo

Este trabalho consistiu no desenvolvimento do jogo *7th Guest: Infection*, na linguagem de programação *Prolog*.

Trata-se de um jogo disputado entre dois jogadores, numa grelha 7x7, e cujo vencedor é o jogador com mais peças no tabuleiro, no final do jogo. Podendo o computador controlar um dos jogadores, ou até ambos, existem assim três modos de jogo: Humano vs Humano, Humano vs Computador e Computador vs Computador.

Os principais objetivos para o projeto eram os de implementar com sucesso todas as regras do jogo, implementar com sucesso os três modos de jogo, implementar com sucesso diferentes níveis de dificuldade para os jogadores controlados pelo computador, e fazer tudo isto com um grau satisfatório de eficiência, em termos de desempenho.

Para a implementação do jogo, as diferentes tarefas foram distribuídas por diferentes módulos, cada um no seu ficheiro *Prolog* (*logic.pl*, *boardManip.pl*, etc), de modo a que os predicados com funcionalidades relacionadas estivessem agrupados.

Seguindo esta organização, implementamos um ciclo de jogo com uma estrutura standard, no qual o estado de jogo é obtido, alterado e armazenado até que se verifique pelo menos uma das condições de fim de jogo.

Tendo permitido verificar a eficiência da linguagem *Prolog* quando aplicada na resolução de problemas de decisão, um dos principais obstáculos à realização do projeto foi o facto de se tratar do primeiro contacto com esta, sendo que, quando se ultrapassou o período inicial de adaptação, o ritmo do desenvolvimento do projeto aumentou bastante.

Após a elaboração do projeto, julgamos ter conseguido cumprir os objetivos a que nos propusemos, sendo que conseguimos implementar todas as funcionalidades que eram pretendidas, sem grandes problemas de performance.

Conteúdo

1	Introdução	4
2	O Jogo - 7th Guest: Infection	5
2.1	História do Jogo	5
2.2	Regras do Jogo	6
3	Lógica do Jogo	7
3.1	Representação do Estado do Jogo	8
3.2	Visualização do Tabuleiro (<i>display_game</i>)	10
3.3	Lista de Jogadas Válidas (<i>valid_moves</i>)	12
3.4	Execução de Jogadas (<i>move</i>)	14
3.5	Final do Jogo (<i>game_over</i>)	17
3.6	Avaliação do Tabuleiro (<i>value</i>)	19
3.7	Jogada do Computador (<i>choose_move</i>)	20
4	Conclusões	22

1 Introdução

Este projeto foi desenvolvido no âmbito da unidade curricular de **Programação em Lógica**, do 3º Ano do Mestrado Integrado em Engenharia Informática e Computação da FEUP.

O projeto consiste na implementação do jogo de tabuleiro *7th Guest: Infection*, no sistema *SICStus Prolog*.

O trabalho teve como objetivo a aplicação da linguagem *Prolog* na implementação do jogo, incluindo vários desafios como a implementação das regras específicas do jogo, ou o desenvolvimento de uma inteligência artificial a partir da qual é possível jogar contra o computador (ou até deixar o computador jogar contra si mesmo).

Este relatório reger-se-á pela seguinte estrutura:

- **O Jogo - 7th Guest: Infection:** Descrição do jogo, incluindo uma breve apresentação da sua história e a descrição das regras
- **Lógica do Jogo:** Descrição da nossa implementação da lógica do jogo, tocando os seguintes aspetos:
 - **Representação do Estado do Jogo:** Explicação de como o estado do jogo é representado na nossa implementação, com exemplos do estado inicial, de estados intermédios e de estados finais do jogo
 - **Visualização do Tabuleiro:** Expliação do método implementado para a visualização do estado interno do jogo, por parte do utilizador
 - **Lista de Jogadas Válidas:** Descrição do método implementado para a geração e validação de jogadas possíveis, para um dado ponto do jogo
 - **Execução de Jogadas:** Explicação do modo como são processadas a validação e a execução das jogadas efetuadas pelos jogadores
 - **Final do Jogo:** Descrição dos predicados que verificam (e, se caso disso, executam) o final do jogo
 - **Avaliação do Tabuleiro:** Explicação do método utilizado para a comparação de diferentes estados do jogo, com base em quão vantajosos são para um dado jogador
 - **Jogada do Computador:** Descrição do predicado que gera a jogada a ser efetuada pelo computador, a um dado ponto no jogo, com base no nível de dificuldade definido

2 O Jogo - 7th Guest: Infection

2.1 História do Jogo

“The 7th Guest: Infection” é um jogo abstrato de estratégia que originalmente apareceu como “Microscope Puzzle” no jogo de computador de 1993, “The 7th Guest”. É baseado na família Ataxx de jogos de tabuleiro, cuja linhagem começou com um jogo de computador de 1988 chamado “Infection”.

A primeira versão deste jogo foi feita em 1988 por Wise Owl software. Escrito por Dave Crum-mack e Craig Galley, “Infection” foi programado nas consolas “Amiga”, “Commodore 64” e “Atari ST”. Os direitos do jogo foram vendidos à Leland Corporation, que lançou o jogo com o nome de “Ataxx”.

Em 1993, Trilobyte lançou “The 7th Guest”, que incluía uma versão deste jogo. Quando Trilobyte re-lançou “The 7th Guest” para o iPhone e iPad em 2010, anunciaram que alguns puzzles, incluindo o “Microscope Puzzle”, não seriam incluídos devido a dificuldades técnicas associadas com a adaptação do jogo original às novas plataformas. No entanto, depois de o jogador completar o jogo, é possível acederem à versão antiga de “Infection”; o puzzle pode então ser jogado através deste método.

Uma versão atualizada do “Microscope Puzzle” foi lançada para o iPad com o nome de “The 7th Guest: Infection”, em Abril de 2011.



Figura 1: Ecrã inicial do jogo

2.2 Regras do Jogo

- O tabuleiro é uma grelha quadrada com **7 linhas e 7 colunas**.
- Cada jogador começa com **dois micróbios**, de cor azul ou vermelha. Os micróbios são colocados nos **quatro cantos do tabuleiro**, sendo que os micróbios azuis ficam nos cantos superior esquerdo e inferior direito, e os vermelhos ficam nos outros 2 cantos. O **primeiro turno** é do jogador controlando os **micróbios vermelhos**.
- Durante um turno, um jogador pode **mover** um dos seus micróbios **uma ou duas casas** em **qualquer direção, incluindo diagonalmente**.
- Se um micrório avançar **uma casa**, um **novo micrório** (da mesma cor) é criado no espaço que o micrório abandonou.
- Se um micrório avançar **duas casas**, o espaço onde o micrório começou o turno **fica vazio**.
- Ao movimentar um micrório, todos os **micróbios do oponente** que estão **adjacentes** ficam “**infetados**”, ficando com a cor do jogador que se movimentou.
- Os jogadores são **obrigados a fazer um movimento** a menos que nenhum movimento legal seja possível.
- O jogo acaba quando se verifica uma das seguintes condições:
 - **todos os espaços do tabuleiro estão preenchidos**
 - **todos os micróbios em campo são de uma cor**

O jogador com **mais micróbios** em campo no fim do jogo é o vencedor!

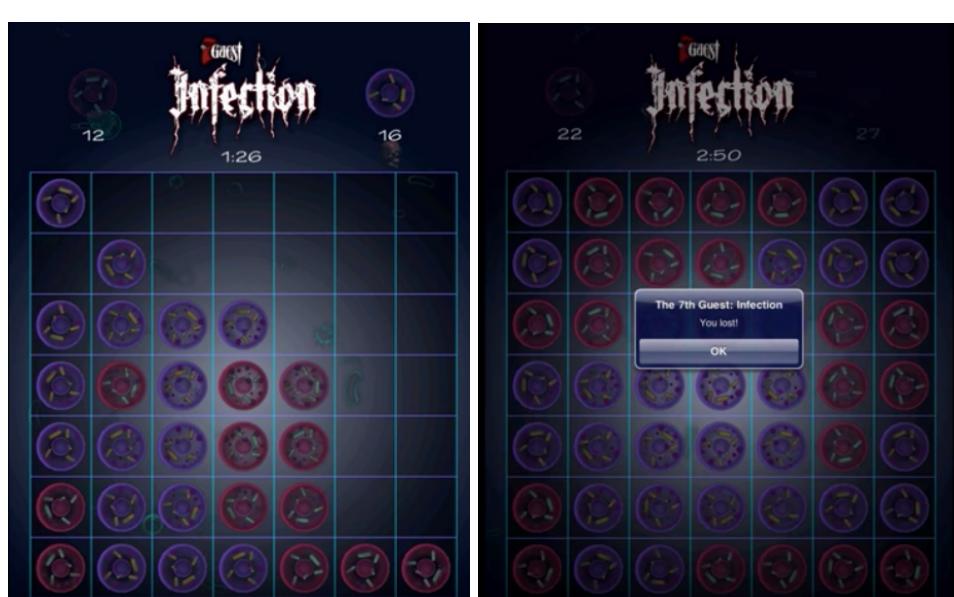


Figura 2: Exemplos do jogo

3 Lógica do Jogo

Irá ser descrita nesta secção a nossa implementação do jogo em Prolog, falando sobre diferentes tópicos desde a visualização do estado de jogo até ao cálculo e geração de jogadas do computador.

O predicado principal do jogo é denominado **play**. Este inicializa o estado do jogo e começa-o (**startGame**), e depois contém o ciclo principal do jogo. Este é formado por 3 tarefas principais: **getState**, que extrai o estado de jogo atual, **changeState**, que gera uma jogada e modifica o estado de jogo, e **saveState**, que tendo em conta essa mudança guarda o estado de jogo modificado, para a próxima iteração do ciclo. Os predicados built-in **repeat** e **once** são utilizados de modo a criar um ciclo, sendo que cada iteração caracteriza um turno/jogada.

O predicado **game_over** é também chamado em cada iteração, sendo verdadeiro apenas se a condição de terminação do jogo foi alcançada; se este for o caso, é executada a saída do ciclo de jogo, e é chamado o predicado **showWinnerAndReset**, que mostra o vencedor do jogo (ou se houve empate), e que restabelece os valores iniciais do estado de jogo, para a próxima vez que o predicado **play** for executado.

```
% <game cycle>

% Main predicate that controls the flow of the game, updating
% it as both players choose their moves until a winner is
% determined
play :-
    startGame,
    repeat,
        once(getState(Board)),
        once(changeState(Board, NewBoard)),
        once(saveState(NewBoard)),
        game_over(NewBoard, Winner),
        showWinnerAndReset(Winner).

% </game cycle>
```

Figura 3: Ciclo principal do jogo

3.1 Representação do Estado do Jogo

O tabuleiro de jogo, de dimensões 7x7, é representado por uma lista (I) de listas (II), sendo que cada lista (II) representa uma fila do tabuleiro. Dentro de cada lista (II), existem 7 átomos, que podem tomar os valores ‘a’, ‘b’, ou ‘ ’, que representam, respectivamente, peças do Jogador A, do Jogador B, e espaços vazios no tabuleiro.

O estado inicial do jogo, como exibido na imagem acima, inclui duas peças do Jogador A, nos cantos superior esquerdo e inferior direito, e duas peças do Jogador B, nos cantos superior direito e inferior esquerdo, e o resto do tabuleiro vazio, sendo este estado internamente representado no seguinte formato:

```
tabuleiroInicial(
    [
        [ 'b', ' ', ' ', ' ', ' ', ' ', 'a'],
        [ ' ', ' ', ' ', ' ', ' ', ' ', ' '],
        [ ' ', ' ', ' ', ' ', ' ', ' ', ' '],
        [ ' ', ' ', ' ', ' ', ' ', ' ', ' '],
        [ ' ', ' ', ' ', ' ', ' ', ' ', ' '],
        [ ' ', ' ', ' ', ' ', ' ', ' ', ' '],
        [ 'a', ' ', ' ', ' ', ' ', ' ', 'b' ]
    ]
).
```

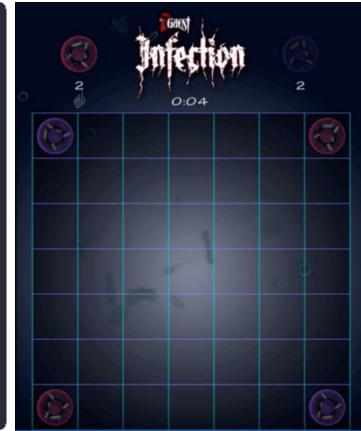


Figura 4: Estado inicial do jogo

Um estado intermédio do jogo incluirá várias peças no tabuleiro, não se verificando ainda nenhuma das condições de final do jogo:

```
tabuleiroIntermedio(
    [
        [ 'a', ' ', ' ', ' ', ' ', ' ', ' '],
        [ ' ', 'a', ' ', ' ', ' ', ' ', ' '],
        [ 'a', 'a', 'a', 'a', ' ', ' ', ' '],
        [ 'a', 'b', 'a', 'b', 'b', ' ', ' '],
        [ 'a', 'a', 'a', 'b', 'b', ' ', ' '],
        [ 'b', 'a', 'a', 'a', 'b', ' ', ' '],
        [ 'b', 'a', 'a', 'a', 'b', 'b', 'a' ]
    ]
).
```



Figura 5: Possível estado intermédio do jogo

O estado final do jogo pode ser atingido a partir de três condições:

- Todas as peças do tabuleiro pertencem ao Jogador A.

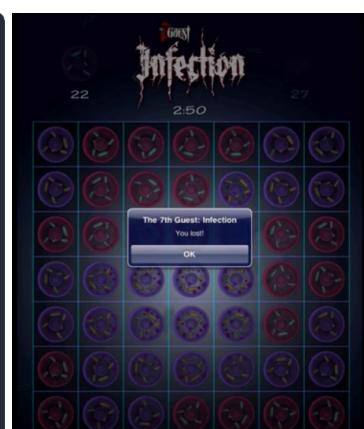
```
tabuleiroFinal(  
    [  
        [ 'a', 'a', 'a', ' ', ' ', ' ', ' ', ' ', 'a'],  
        [ ' ', 'a', ' ', ' ', ' ', ' ', ' ', ' ', 'a'],  
        [ ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'a'],  
        [ 'a', 'a', 'a', 'a', 'a', ' ', ' ', ' ', ' '],  
        [ 'a', 'a', 'a', 'a', 'a', 'a', ' ', ' ', ' '],  
        [ 'a', 'a', 'a', 'a', 'a', 'a', 'a', ' ', ' '],  
        [ 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', ' ']  
    ]  
).
```

- Todas as peças do tabuleiro pertencem ao Jogador B.

```
tabuleiroFinal(  
    [  
        [ 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'],  
        [ ' ', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'],  
        [ ' ', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'],  
        [ ' ', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'],  
        [ ' ', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'],  
        [ ' ', ' ', 'b', 'b', 'b', 'b', 'b', 'b', 'b'],  
        [ 'b', ' ', 'b', 'b', 'b', 'b', 'b', 'b', 'b']  
    ]  
).
```

- O tabuleiro encontra-se completamente preenchido.

```
tabuleiroFinal(  
    [  
        [ 'a', 'b', 'b', 'b', 'b', 'a', 'a', 'a'],  
        [ 'a', 'b', 'b', 'b', 'a', 'a', 'a', 'a'],  
        [ 'b', 'b', 'a', 'b', 'a', 'b', 'b', ' '],  
        [ 'a', 'a', 'a', 'a', 'a', 'b', 'b', ' '],  
        [ 'a', 'a', 'a', 'a', 'a', 'b', 'a', ' '],  
        [ 'b', 'a', 'a', 'a', 'a', 'a', 'a', ' '],  
        [ 'b', 'a', 'a', 'b', 'b', 'b', 'b', ' ']  
    ]  
).
```



3.2 Visualização do Tabuleiro (*display_game*)

O predicado utilizado para a visualização do tabuleiro percorre a **lista de listas** que descreve o estado do jogo, mostrando em seguida o número de pontos (peças no tabuleiro) de cada jogador. É indicado também qual dos dois jogadores vai jogar a seguir.

```
% <game state representation>

% -- Predicate that displays the board, as well as each player's score, and
% -- who's turn it is to play
display_game(Board, Player) :-
    pointsA(PointsA),
    pointsB(PointsB),
    format("-----~n~n", []),
    printBoard(Board),
    nl,
    format("          Player A          Player B          ~n", []),
    format("          ~p          ~p          ~n", [PointsA, PointsB]),
    nl,
    format("          It is player ~ps turn.          ~n",
           [Player]),
    format("-----~n", []).

% -- Predicate that prints the current state of the board
printBoard(Board) :-
    format("----- ~n", []),
    format("     | 1 | 2 | 3 | 4 | 5 | 6 | 7 |~n", []),
    format("-----|-----|-----|-----|-----|-----|-----|~n", []),
    printBoardLine(1, Board).

% -- End condition for the predicate, which iterates over the board's lines
printBoardLine(_, []).

% -- Predicate that iterates over the board's lines, displaying each of them
printBoardLine(NumLine, [X | RestLines]) :-
    format("| ~d |", [NumLine]),
    format(" ~p | ~p | ~p | ~p | ~p | ~p | ~p |~n", X),
    format("-----|-----|-----|-----|-----|-----|-----|~n", []),
    NewNumLine is NumLine + 1,
    printBoardLine(NewNumLine, RestLines).

% </game state representation>
```

Figura 6: Código que apresenta o tabuleiro

```

tabuleiroIntermedio(
    [
        [ a , ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
        [ ' ', a , ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
        [ a , a , a , a , ' ', ' ', ' ', ' ', ' ' ],
        [ a , b , a , b , b , ' ', ' ', ' ', ' ' ],
        [ a , a , a , b , b , ' ', ' ', ' ', ' ' ],
        [ b , a , a , a , b , ' ', ' ', ' ', ' ' ],
        [ b , a , a , a , b , b , a ]
    ]
).

testDisplayTab :-
    tabuleiroIntermedio(Tab),
    updatePointsNewBoard(Tab),
    display_game(Tab, 'A').

```

Figura 7: Predicado de teste da visualização do jogo

```

| ?- testDisplayTab.
-----
+-----+
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+-----+
| 1 | a |   |   |   |   |   |
+-----+
| 2 |   | a |   |   |   |   |
+-----+
| 3 | a | a | a | a |   |   |
+-----+
| 4 | a | b | a | b | b |   |
+-----+
| 5 | a | a | a | b | b |   |
+-----+
| 6 | b | a | a | a | b |   |
+-----+
| 7 | b | a | a | a | b | b |
+-----+
Player A          Player B
18                10
It is player As turn.
-----
yes
| ?-

```

Figura 8: Output do teste da visualização do jogo

3.3 Lista de Jogadas Válidas (*valid_moves*)

O predicado em questão recebe uma variável denominada **Player**, que pode ser ‘A’ ou ‘B’, representando o jogador que deve jogar a seguir; uma variável chamada **Board**, contém o estado atual do jogo (tabuleiro); e retorna **ListOfValidMoves**, uma lista de estruturas que representam um dado movimento que o jogador poderá efetuar (são tuplos do género **OldLine-OldColumn-NewLine-NewColumn**, sendo que as variáveis **Old** se referem ao sítio onde a peça se encontrava, e as variáveis **New** à nova posição que a peça irá ter). Este predicado é utilizado na lógica das jogadas do computador, de modo a gerar todas as jogadas possíveis de modo a escolher a mais vantajosa, e também na lógica das jogadas do humano, de modo a saber se há alguma jogada válida que este poderá fazer (sendo que, se não existir, o seu turno é passado à frente).

O predicado utiliza o predicado built-in **findall**, que chama um predicado feito por nós denominado **findMove**. Este último retorna uma possível jogada que o **Player** poderá fazer, sendo que se o utilizador pedir outra solução (com ponto e vírgula), o predicado irá gerar outra jogada possível (se existir). Desta maneira, utilizando **findall** é possível gerar todas as jogadas válidas possíveis.

Relativamente ao funcionamento do predicado **findMove**, este, tendo em conta o jogador atual a jogar, tenta encontrar uma peça do mesmo que esteja no tabuleiro, gerando depois um possível movimento para essa peça (por exemplo, 1 casa para a esquerda). De seguida, verifica-se a validade dessa jogada, chamando o predicado **checkValidMove**, de modo ao predicado apenas ser verdadeiro se o movimento for válido, tendo em conta o conteúdo do tabuleiro atual. Com a estratégia adotada de gerar uma solução e de seguida verificar a validade da mesma, tira-se partido do mecanismo de retrocesso do Prolog, uma vez que, se o movimento não for válido, o retrocesso faz com que se escolha outro movimento e/ou outra peça do tabuleiro para mover (se for possível). O **once** é utilizado quando se chama **checkValidMove** de modo a evitar retrocesso dentro deste predicado.

```
% <cpu move generation>

% -- Predicate that, using findall() with the findMove() predicate, generates
% -- all possible moves available to the player, returning them in a list
valid_moves(Player, Board, ListOfValidMoves) :-
    findall(OldLine-OldColumn-NewLine-NewColumn,
            findMove(Player, Board, OldLine,
                    OldColumn, NewLine, NewColumn),
            ListOfValidMoves).
```

Figura 9: Código que gera as possíveis jogadas de um jogador

```

% -- Predicate to be used by the Computer, which generates a possible move
% -- according to the board state
% -- Player - the player for which the move is generated
% -- Board - current state of the board
% -- OldLine - line of the microbe to be moved
% -- OldColumn - column of the microbe to be moved
% -- NewLine - line that the microbe should be moved to
% -- NewColumn - column that the microbe should be moved to
findMove(Player, Board, OldLine, OldColumn, NewLine, NewColumn) :-  

    getMicrobeType(Player, MicrobeType),  

    getPositionsForMicrobe(MicrobeType, Board, OldLine, OldColumn),  

    generateValidPosition(OldLine, OldColumn, NewLine, NewColumn),  

    once(checkValidMove(MicrobeType, OldLine, OldColumn,  

        NewLine, NewColumn, Board, _)).  
  

generateValidPosition(Line, Column, LineOut, ColumnOut) :-  

    LineOut is Line - 2, ColumnOut is Column - 2.  
  

generateValidPosition(Line, Column, LineOut, ColumnOut) :-  

    LineOut is Line - 2, ColumnOut is Column.  
  

generateValidPosition(Line, Column, LineOut, ColumnOut) :-  

    LineOut is Line - 2, ColumnOut is Column + 2.  
  

generateValidPosition(Line, Column, LineOut, ColumnOut) :-  

    LineOut is Line - 1, ColumnOut is Column - 1.  
  

generateValidPosition(Line, Column, LineOut, ColumnOut) :-  

    LineOut is Line - 1, ColumnOut is Column.  
  

generateValidPosition(Line, Column, LineOut, ColumnOut) :-  

    LineOut is Line - 1, ColumnOut is Column + 1.  
  

generateValidPosition(Line, Column, LineOut, ColumnOut) :-  

    LineOut is Line, ColumnOut is Column - 2.  
  

generateValidPosition(Line, Column, LineOut, ColumnOut) :-  

    LineOut is Line, ColumnOut is Column - 1.  
  

generateValidPosition(Line, Column, LineOut, ColumnOut) :-  

    LineOut is Line, ColumnOut is Column + 1.  
  

generateValidPosition(Line, Column, LineOut, ColumnOut) :-  

    LineOut is Line, ColumnOut is Column + 2.  
  

generateValidPosition(Line, Column, LineOut, ColumnOut) :-  

    LineOut is Line + 1, ColumnOut is Column - 1.  
  

generateValidPosition(Line, Column, LineOut, ColumnOut) :-  

    LineOut is Line + 1, ColumnOut is Column.  
  

generateValidPosition(Line, Column, LineOut, ColumnOut) :-  

    LineOut is Line + 1, ColumnOut is Column + 1.  
  

generateValidPosition(Line, Column, LineOut, ColumnOut) :-  

    LineOut is Line + 2, ColumnOut is Column - 2.  
  

generateValidPosition(Line, Column, LineOut, ColumnOut) :-  

    LineOut is Line + 2, ColumnOut is Column.  
  

generateValidPosition(Line, Column, LineOut, ColumnOut) :-  

    LineOut is Line + 2, ColumnOut is Column + 2.  
  

% </cpu move generation>

```

Figura 10: Predicado para encontrar uma potencial jogada para um jogador

3.4 Execução de Jogadas (*move*)

O predicado **move** recebe: **Player**, que indica o jogador que está a fazer o movimento; a posição da peça que este pretende mover, através dos argumentos **OldLine** e **OldColumn**; a posição para a qual se pretende mover a peça, com **NewLine** e **NewColumn**; e **Board**, representando o tabuleiro atual. O predicado valida o movimento proposto pelo jogador, e se este for válido então o predicado modifica o tabuleiro, executando o movimento, sendo que o tabuleiro resultante é retornado através do argumento **NewBoard**.

O predicado começa por saber qual é o tipo de peça que corresponde ao jogador, através de **getMicrobeType**. De seguida, chama **checkValidMove** de modo a verificar se o movimento proposto pelo jogador é válido; este predicado verifica se há uma peça do jogador na posição inicial, se a posição final se encontra livre, e se o movimento entre as duas posições é válido.

Se o movimento for válido, é adicionada uma nova peça na posição final (**playMicrobe**), sendo que a peça original é retirada se o movimento não for adjacente, e mantida caso seja (**handleIsAdjacent**). Por fim, chama um predicado que “contamina” todas as peças do oponente que são adjacentes à nova peça, que passam a ser peças do jogador que fez o movimento (**contaminateAdjacent**).

```
% <move execution>

% -- Predicate that validates and executes a proposed play
% -- Player - player who requested the move (important to know the type of
% --           piece to move)
% -- OldLine - Line of the position of the microbe that should be moved
% -- OldColumn - Column of the position of the microbe that should be moved
% -- NewLine - Line of the position to which the microbe should be moved
% -- NewColumn - Column of the position to which the microbe should be moved
% -- Board - board that represents the game state before the move
% -- NewBoard - board where the changes to the game state are represented
move(Player, OldLine, OldColumn, NewLine, NewColumn, Board, NewBoard) :-
    getMicrobeType(Player, MicrobeType),
    checkValidMove(MicrobeType, OldLine, OldColumn,
                   NewLine, NewColumn, Board, IsAdjacent),
    playMicrobe(NewLine, NewColumn, MicrobeType, Board, BoardOut),
    handleIsAdjacent(IsAdjacent, OldLine, OldColumn, BoardOut, BoardOut2),
    contaminateAdjacent(Player, NewLine, NewColumn, BoardOut2, NewBoard).

% -- Predicate that distinguishes moves to adjacent positions, for the
% -- application of the special rules for this kind of moves

% -- -- Case where the move is executed towards a non-adjacent position
handleIsAdjacent('no', OldLine, OldColumn, BoardIn, BoardOut) :-
    playMicrobe(OldLine, OldColumn, ' ', BoardIn, BoardOut).

% -- -- Case where the move is executed towards an adjacent position
handleIsAdjacent('yes', _, _, Board, Board).

% </move execution>
```

Figura 11: Código que valida e executa uma jogada

```
% <move validity check>

% -- Predicate that checks whether a move to be executed by a player is valid
% -- or not. The predicate is true if the move is valid.
% -- Returns an IsAdjacent value, to distinguish moves to adjacent positions,
% -- which carry a fair amount of importance to the rules of the game.
checkValidMove(MicrobeType, OldLine, OldCol, NewLine,
               NewCol, Board, IsAdjacent) :-
    verifyMicrobePositions(MicrobeType, Board, OldLine,
                           OldCol, NewLine, NewCol),
    verifyMicrobeMovement(OldLine, OldCol, NewLine,
                          NewCol, IsAdjacent).

% </move validity check>
```

Figura 12: Validação de uma qualquer jogada

```
% <move positions verification>

% -- Predicate that validates the initial and final positions of a move. For
% -- this to check out, the initial position must contain a piece from the
% -- player, and the final position must be empty.
verifyMicrobePositions(MicrobeType, Board, OldLine,
                       OldColumn, NewLine, NewColumn) :-
    OldLine > 0, OldLine < 8, OldColumn > 0, OldColumn < 8,
    NewLine > 0, NewLine < 8, NewColumn > 0, NewColumn < 8,
    returnMicrobeInPos(OldLine, OldColumn, Board, Microbe1), !,
    Microbe1 = MicrobeType,
    returnMicrobeInPos(NewLine, NewColumn, Board, Microbe2), !,
    Microbe2 = ' '.

% </move positions verification>
```

Figura 13: Verificação da posição para a qual a peça é movida

```
% <microbe movement validation>

% -- Predicate that checks whether the move for the selected microbe is valid.

% -- -- Case where the movement is toward an adjacent position
verifyMicrobeMovement(OldLine, OldColumn, NewLine, NewColumn, IsAdjacent) :-
    (OldLine \= NewLine; OldColumn \= NewColumn),
    LineDif is OldLine - NewLine,
    ColDif is OldColumn - NewColumn,
    abs(LineDif) < 2, abs(ColDif) < 2,
    IsAdjacent = 'yes'.

% -- -- Case where the movement is toward a non-adjacent position
verifyMicrobeMovement(OldLine, OldColumn, NewLine, NewColumn, IsAdjacent) :-
    (OldLine \= NewLine; OldColumn \= NewColumn),
    LineDif is OldLine - NewLine,
    ColDif is OldColumn - NewColumn,
    (abs(LineDif) =:= 0; abs(LineDif) =:= 2),
    (abs(ColDif) =:= 0; abs(ColDif) =:= 2),
    IsAdjacent = 'no'.

% </microbe movement validation>
```

Figura 14: Verificação da direção do movimento

```

% <enemy piece contamination>

% -- Predicate that, given the position of the newly set piece, contaminates
% -- the adjacent enemy pieces
contaminateAdjacent(Player, Line, Column, Board, NewBoard) :-  

    getMicrobeType(Player, MicrobeType),  

    AuxLine1 is Line - 1, AuxCol1 is Column - 1,  

    contaminatePosition(MicrobeType, AuxLine1, AuxCol1, Board, BoardAux1),  

    AuxLine2 is Line - 1, AuxCol2 is Column,  

    contaminatePosition(MicrobeType, AuxLine2, AuxCol2, BoardAux1, BoardAux2),  

    AuxLine3 is Line - 1, AuxCol3 is Column + 1,  

    contaminatePosition(MicrobeType, AuxLine3, AuxCol3, BoardAux2, BoardAux3),  

    AuxLine4 is Line, AuxCol4 is Column - 1,  

    contaminatePosition(MicrobeType, AuxLine4, AuxCol4, BoardAux3, BoardAux4),  

    AuxLine5 is Line, AuxCol5 is Column + 1,  

    contaminatePosition(MicrobeType, AuxLine5, AuxCol5, BoardAux4, BoardAux5),  

    AuxLine6 is Line + 1, AuxCol6 is Column - 1,  

    contaminatePosition(MicrobeType, AuxLine6, AuxCol6, BoardAux5, BoardAux6),  

    AuxLine7 is Line + 1, AuxCol7 is Column,  

    contaminatePosition(MicrobeType, AuxLine7, AuxCol7, BoardAux6, BoardAux7),  

    AuxLine8 is Line + 1, AuxCol8 is Column + 1,  

    contaminatePosition(MicrobeType, AuxLine8, AuxCol8, BoardAux7, NewBoard).

% -- Predicate that contaminates a position of the board, if it belongs to the
% -- opposing player

% -- -- Case where the position will be contaminated
contaminatePosition(MicrobeType, Line, Column, Board, NewBoard) :-  

    Line > 0, Line < 8,  

    Column > 0, Column < 8,  

    returnMicrobeInPos(Line, Column, Board, Microbe),  

    Microbe \= MicrobeType, Microbe \= ' ',  

    playMicrobe(Line, Column, MicrobeType, Board, NewBoard).

% -- -- Case where the position will not be contaminated
contaminatePosition(_, _, _, Board, Board).

% </enemy piece contamination>

```

Figura 15: Contaminação das peças inimigas adjacentes

3.5 Final do Jogo (*game_over*)

Este predicado recebe o estado atual do jogo, através de **Board**, verifica se o jogo acabou e, se tal ocorreu, identifica qual o jogador que venceu o jogo, retornando-o através de **Winner**.

Como descrito anteriormente, o jogo termina se apenas existem peças de um tipo em campo, ou se todas as posições do campo estão preenchidas, sendo que neste caso o vencedor é quem tem o maior número de peças no tabuleiro.

O predicado começa por percorrer o tabuleiro, verificando se existem peças do jogador B (se não houver, o vencedor é A). Verifica depois se existem peças do jogador A (se não houver, o vencedor é B). Por fim, verifica se existem espaços livres no tabuleiro. Caso esta última condição não seja verdadeira, o predicado compara o número de pontos atual de cada jogador, sendo que o vencedor é quem tem a pontuação mais alta (uma vez que o número de pontos de um jogador é igual ao número de peças que este tem em campo).

```
% <game over check>

% -- Predicate that determines whether the game already has a winner,
% -- checking the possible end conditions:
% -- - All pieces on the board belong to the same player
% -- - The board is completely full
game_over(Board, Winner) :- 

    ((getMicrobeType('B', MicrobeB),
      boardEndCheck(MicrobeB, Board));

     (getMicrobeType('A', MicrobeA),
      boardEndCheck(MicrobeA, Board));

     boardEndCheck(' ', Board)),

     pointsA(A),
     pointsB(B),
     declareWinner(A, B, Winner).

% </game over check>
```

Figura 16: Código que verifica e, se caso disso, executa o final do jogo

```

% <endgame procedures>

% -- Predicate that determines the winner, depending on the scores of both
% -- players

% -- -- Case where player A has a higher score
declareWinner(A, B, Winner) :-  

    A > B,  

    Winner = 'A'.

% -- -- Case where player B has a higher score
declareWinner(A, B, Winner) :-  

    B > A,  

    Winner = 'B'.

% -- -- Case where the players have equal scores
declareWinner(A, B, Winner) :-  

    A == B,  

    Winner = 'draw'.

% -- End condition for the predicate, which iterates over the board
boardEndCheck(_, []).

% -- Predicate that iterates over the board, returning false if a piece of the
% -- type P is found
boardEndCheck(P, [Line | Rest]) :-  

    boardLineEndCheck(P, Line),  

    boardEndCheck(P, Rest).

% -- End condition for the predicate, which iterates over a line of the board
boardLineEndCheck(_, []).

% -- Predicate that iterates over a line of the board, returning false if a
% -- piece of the type P is found
boardLineEndCheck(P, [Head | Tail]) :-  

    Head \= P,  

    boardLineEndCheck(P, Tail).

% </endgame procedures>

```

Figura 17: Código auxiliar de final de jogo

3.6 Avaliação do Tabuleiro (*value*)

Este predicado avalia o estado de jogo que lhe é passado como argumento, através de **Board**, retornando em **Value** um valor inteiro que caracteriza o valor desse estado, sendo que quanto maior for esse valor, mais “vantajoso” é esse estado relativamente ao **Player** passado como argumento.

Optamos por desenvolver e integrar 2 níveis de dificuldade do “AI” do nosso jogo: easy (**nível 1**) e medium (**nível 2**). O predicado **value** apenas é chamado quando um computador se encontra no nível 2 de dificuldade (no nível 1 é gerado um value aleatório; irá ser visto mais à frente).

A função **value** retorna a diferença entre o número de peças do utilizador e o número de peças do oponente. É uma estratégia gananciosa que faz com que um computador a jogar no nível 2 escolha sempre a jogada que é mais vantajosa, nessa altura, para ele, uma vez que tendo uma lista de jogadas válidas, a jogada escolhida é a que adiciona mais peças do jogador ao tabuleiro e/ou a que retira mais peças do oponente.

```
% <move evaluation>

% -- Predicate that evaluates the game state, returning a value to
% -- characterize it; only used when the difficulty is set to level 2 (medium):
% -- -- move's value is equal to the piece advantage of the player who executes
% -- -- the move (value can be negative!)
value(Board, Player, Value) :-  
    updatePoints(Board, 0, 0, PointsA, PointsB),  
    valueAux(Player, Value, PointsA, PointsB).

% -- Predicate that calculates the value of the play, based on the number of
% -- pieces on the board from each player

% -- -- Case where player A executes the move
valueAux('A', Value, PointsA, PointsB) :-  
    Value is PointsA-PointsB.

% -- -- Case where player B executes the move
valueAux('B', Value, PointsA, PointsB) :-  
    Value is PointsB-PointsA.

% </move evaluation>
```

Figura 18: Predicado de avaliação do tabuleiro

3.7 Jogada do Computador (*choose_move*)

Este predicado apenas é chamado no turno de um computador, sendo que não é utilizado quando se trata de um humano a jogar. Este gera todos os possíveis movimentos/jogadas que o computador pode fazer, através de **valid_moves**, e através de **chooseBestMove** avalia cada jogada, e retorna aquela que é considerada a melhor.

A maneira como as jogadas são avaliadas varia com o nível de dificuldade do computador que chama o predicado (passado em **Level**).

Para o nível 1, e tal como foi dito anteriormente, a cada possível jogada irá ser atribuída um value aleatório, de 0 a 99 (números arbitrários, podiam ser outros), fazendo com que a jogada escolhida seja completamente aleatória.

Para o nível 2, cada jogada é “simulada”, sendo chamado o predicado **move** para construir o Board resultante da aplicação dessa jogada. Esse Board é então avaliado recorrendo à função **value** descrita anteriormente. Deste modo, a jogada retornada é a que está associada ao maior valor retornado por essa função **value**.

Se o jogador que invoca o predicado não tiver movimentos válidos que possa fazer (**valid_moves** retorna uma lista vazia), então o movimento retornado é **movement(0, 0, 0, 0)**, o que indica a outro setor do código que o jogador não tem jogadas válidas e que o seu turno deverá ser passado à frente.

O predicado recebe como argumentos o **Level**, o **Player** que está a jogar, e o **Board** atual. Retorna o melhor movimento/jogada, em **movement(OldLine, OldColumn, NewLine, NewColumn)**.

```
% <cpu move selection>

% -- Predicate to be used by the Computer, that creates all possible
% -- scenarios and chooses, according to the difficulty level, the best
% -- possible move
choose_move(Level, Player, Board,
            movement(OldLine, OldColumn, NewLine, NewColumn)) :-
    valid_moves(Player, Board,ListOfValidMoves),
    chooseBestMove(Level, ListOfValidMoves, Player, Board,
                  movement(OldLine, OldColumn, NewLine, NewColumn)).
```

Figura 19: Predicado que selecciona a jogada do computador

```

% -- Predicate that receives a list with all possible moves and, according to
% -- the difficulty level, calls the value() predicate for each move,
% -- returning in BestMove the most advantageous move
% -- If there are no available moves, the move returned is (0, 0, 0, 0), and
% -- the turn is skipped
% -- Level - difficulty level according to which the moves will be evaluated
% -- ListOfValidMoves - list containing each possible move
% -- Player - player for which the moves are being computed
% -- Board - board that represents the game state before any move is executed
% -- BestMove - the most advantageous move
chooseBestMove(Level, ListOfValidMoves, Player, Board,
               movement(OldLine, OldColumn, NewLine, NewColumn)) :-  

    chooseBestMoveAux(Level, ListOfValidMoves, Player, -999999, Board,  

                      movement(0, 0, 0, 0),
                      movement(OldLine, OldColumn, NewLine, NewColumn)).  
  

% -- End condition for the predicate, which iterates over the list of
% -- possible moves
chooseBestMoveAux(_, [], _, _, _)
chooseBestMoveAux(_, [movement(OldLine, OldColumn, NewLine, NewColumn)], _, _, _)
chooseBestMoveAux(_, [movement(OldLine, OldColumn, NewLine, NewColumn)], _, _, _).  
  

% -- Predicate that iterates over the list of possible moves, checking if the
% -- current move is better than the previous best option, and returning the
% -- best move available in the end
% -- Level - difficulty level to be passed to the value() predicate
% -- [CurOldLine-CurOldColumn-CurNewLine-CurNewColumn | Rest] -
% -- - Head and Tail of the list of possible moves
% -- Player - player for which the moves' value will be calculated
% -- AuxValue - variable that will store the best value obtained at any point
% -- during the iteration of the list
% -- movement(AuxOldLine, AuxOldColumn, AuxNewLine, AuxNewColumn) -
% -- - variable that will store the move for which the value is highest at
% -- any point during the iteration of the list
% -- movement(OldLine, OldColumn, NewLine, NewColumn) - field that will be
% -- returned with the move for which the value is highest in the list
% -- Level 1
chooseBestMoveAux(1, [CurOldLine-CurOldColumn-CurNewLine-CurNewColumn | Rest],
                  Player, AuxValue, Board,
                  movement(AuxOldLine, AuxOldColumn, AuxNewLine, AuxNewColumn),
                  movement(OldLine, OldColumn, NewLine, NewColumn)) :-  

    random(0, 100, Value),
    Value > AuxValue ->
    chooseBestMoveAux(1, Rest, Player, Value, Board,
                     movement(CurOldLine, CurOldColumn,
                               CurNewLine, CurNewColumn),
                     movement(OldLine, OldColumn,
                               Newline, NewColumn));
    chooseBestMoveAux(1, Rest, Player, AuxValue, Board,
                     movement(AuxOldLine, AuxOldColumn,
                               AuxNewLine, AuxNewColumn),
                     movement(OldLine, OldColumn,
                               NewLine, NewColumn)).  
  

% -- Level 2
chooseBestMoveAux(2, [CurOldLine-CurOldColumn-CurNewLine-CurNewColumn | Rest],
                  Player, AuxValue, Board,
                  movement(AuxOldLine, AuxOldColumn, AuxNewLine, AuxNewColumn),
                  movement(OldLine, OldColumn, NewLine, NewColumn)) :-  

    move(Player, CurOldLine, CurOldColumn, CurNewLine,
          CurNewColumn, Board, AuxBoard),
    value(AuxBoard, Player, Value),
    Value > AuxValue ->
    chooseBestMoveAux(2, Rest, Player, Value, Board,
                     movement(CurOldLine, CurOldColumn,
                               CurNewLine, CurNewColumn),
                     movement(OldLine, OldColumn,
                               Newline, NewColumn));
    chooseBestMoveAux(2, Rest, Player, AuxValue, Board,
                     movement(AuxOldLine, AuxOldColumn,
                               AuxNewLine, AuxNewColumn),
                     movement(OldLine, OldColumn,
                               NewLine, NewColumn)).  
  

% </cpu move selection>

```

Figura 20: Código que seleciona a mais vantajosa das jogadas fornecidas

4 Conclusões

O projeto teve como propósito a aplicação prática dos conhecimentos e competências adquiridas nas aulas teóricas e teórico-práticas da unidade curricular de Programação em Lógica, exigindo a formulação de todo o processo de pensamento do jogo adaptado às necessidades e características da linguagem *Prolog*.

Ao longo do seu desenvolvimento, foram encontradas algumas dificuldades, como a questão de implementar os raciocínios formulados para a resolução de cada problema seguindo as normas da programação em Prolog, sendo uma característica marcante o uso quase constante da recursividade. No entanto, as dificuldades encontradas foram superadas pelo final do projeto.

Como seria de esperar, há aspectos do trabalho que poderiam ser melhorados. Um exemplo seria a implementação de um nível de inteligência artificial que seguisse uma estratégia superior à estratégia gananciosa que foi implementada, tendo possivelmente em conta as possíveis jogadas seguintes à calculada pelo computador. Além disto, seria também positivo otimizar o desempenho dos predicados desenvolvidos.

Concluindo, o trabalho foi concluído com sucesso, tendo contribuído para a aquisição e consolidação das competências pretendidas para a unidade curricular, e proporcionado uma boa aprendizagem da linguagem de programação *Prolog*.

Referências

- [1] The 7th Guest: Infection, accessed on October of 2019,
https://en.wikipedia.org/wiki/The_7th_Guest:_Infection.
- [2] The 7th Guest: Infection — Board Game, accessed on October of 2019.
<https://www.boardgamegeek.com/boardgame/284017/7th-guest-infection>.
- [3] The 7th Guest Infection - iPad - US - Gameplay Trailer,
<https://www.youtube.com/watch?v=4VheoiJdnUM>.
- [4] Eli Hodapp, for TouchArcade, 27 April 2011, accessed on October of 2019,
<https://toucharcade.com/2011/04/27/the-7th-guest-infection-for-ipad/>.