Virus Wars

Relatório Final

Mestrado Integrado em Engenharia Informática e Computação
Programação em Lógica

Grupo 02: Virus Wars
Margarida Ramos Pereira da Silva | 201606214
César Manuel Nobre Medeiros | 201605344

Outubro de 2018

Índice

1. Introdução

Este projeto foi desenvolvido no âmbito da Unidade Curricular Programação em Lógica, onde nos foi proposta a realização de um jogo na linguagem Prolog, com o intuito de aplicar os conceitos adquiridos nas aulas teóricas e teórico-práticas.

Na escolha do jogo a desenvolver tivemos vários fatores em consideração, em destaque a sua jogabilidade e atratividade. Por um lado, entendemos que Virus Wars é um jogo simples, e, mesmo assim, um jogo agradável de se jogar e fonte de entretenimento. Por outro lado, por sabermos que iríamos ter de desenvolver uma interface gráfica na UC de LAIG, procurámos um jogo apelativo, e, na nossa opinião, este jogo tem bastante potencial tal efeito, nomeadamente quando uma peça se torna *zombie*.

O objetivo deste trabalho é então perceber como é possível desenvolver um jogo segundo o paradigma da programação lógica, contemplando a sua estruturação, mecanismos simples de inteligência artificial, e interfaces em modo de texto robustas a inputs inesperados.

2. O jogo Virus Wars

2.1. História

A origem exata do jogo é desconhecida, embora tenha sido ativamente ao ser jogado na Saint Petersburg State University em papel, nos anos 80. Haviam várias versões do jogo com regras ligeiramente diferentes. [1]

2.2. Regras

2.2.1. Tabuleiro

Virus wars é um jogo de dois jogadores, num tabuleiro quadrado, que pode ter tamanho variável.[1] No nosso trabalho o utilizador decide o tamanho do tabuleiro no qual pretende jogar. Para demonstrar o funcionamento do jogo foram utilizados tabuleiros 4x4 por forma a ajudar na compreensão.

2.2.2. Peças

Cada jogador joga com peças diferentes, sendo que cada peça representa o seu vírus. Existem ainda mais duas peças - “zombies” - para demonstrar as peças de de jogador que foram “absorvidas” pelo seu adversário. [1]

Peças Vírus

Peças zombie

2.2.3. Jogadas

As posições desocupadas dos tabuleiros podem-se encontrar acessíveis, ou não. Uma posição está acessível se está adjacente (seja qual for a direção) a uma peça vírus ou zombie * previamente colocada pelo próprio jogador.

*No caso de se encontrar adjacente um zombie, este tem de pertencer a uma cadeia de zombies que em algum ponto se interliga a uma peça vírus do jogador em questão.

No início, o tabuleiro está vazio e não há posições acessíveis. Assim, o primeiro jogador pode por a sua peça vírus em qualquer posição do lado esquerdo do tabuleiro. Do mesmo modo, o segundo jogador coloca a sua peça em qualquer posição do lado direito.

Os jogadores vão jogando por turnos. Cada turno é composto por um determinado número N de jogadas. Se um jogador não consegue completar as duas N jogadas, perde o jogo. No nosso jogo, considerámos 3 jogadas por turno.

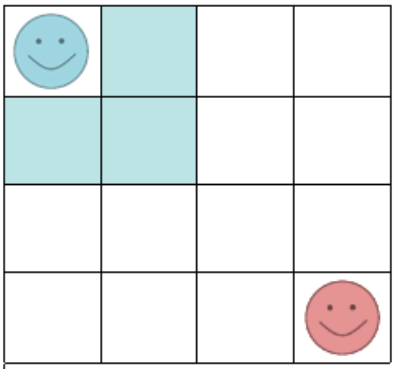
Cada jogada pode corresponder a uma “contaminação” ou “absorção”.

Contaminação: Colocar uma peça de vírus numa posição acessível, “espalhando-o”.

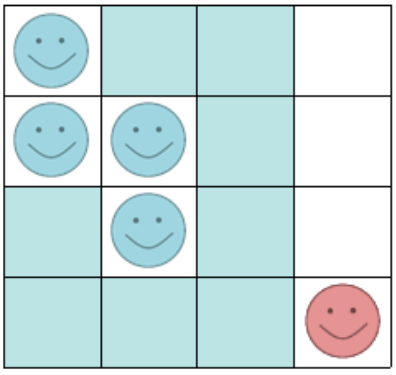
Absorção: Colocar uma peça zombie em substituição da peça de vírus inimiga que esteja numa posição acessível. As peças zombie são imóveis, uma vez colocadas no tabuleiro.

Exemplo de jogo:

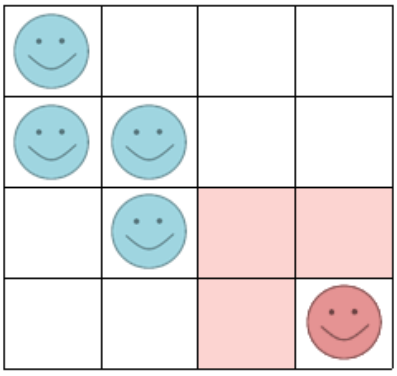
1. O jogo começa. O jogador 1 tem 3 posições acessíveis.



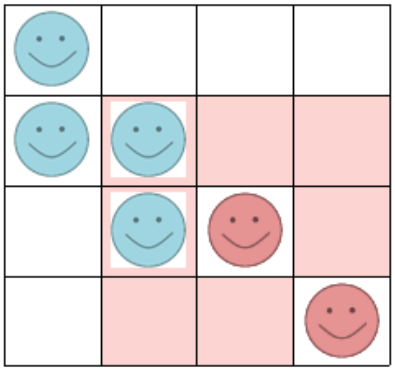
2. O jogador 1 completa a sua jogada e tem novas posições acessíveis



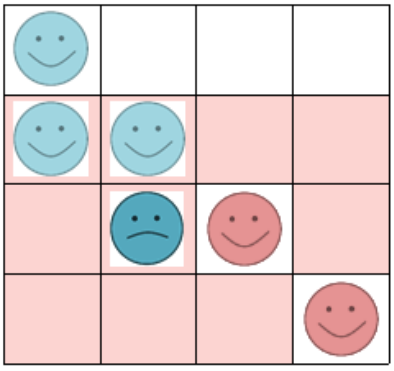
3. É a vez do jogador 2. Tem 3 posições acessíveis.



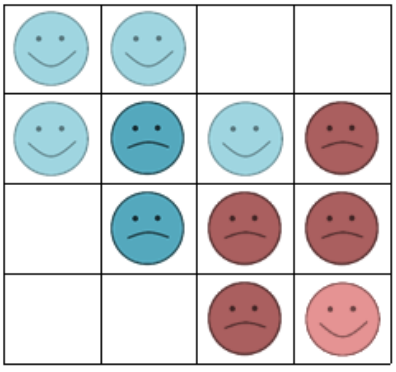
4. O jogador 2 começa a sua jogada. Surge a possibilidade de tornar uma peça vírus do inimigo em *zombie*, tanto pela esquerda como na diagona.



5. Uma peça vírus do jogador 1 é transformada em peça *zombie*.



6. O jogo termina. O jogador 1 vence pois não existem mais posições acessíveis para o jogador 2.



3. Lógica do Jogo

3.1. Representação do Estado do Jogo

O estado do jogo é composto pelos seguintes fatores:

- Jogador responsável pela próxima jogada
- Número da jogada do turno
- Posição das peças no tabuleiro

Quanto ao jogador da próxima jogada, planeamos guardá-lo num simples número inteiro, 0 em caso de ser o utilizador, e 1 no caso de ser o computador. No caso do número da jogada do turno, guardamo-lo como um inteiro que varia de 3 para 1 e volta a ser 3 no final de cada turno.

Por forma a guardar o estado das peças no tabuleiro, pensámos em implementar uma lista de listas, uma vez que se trata de um tabuleiro retangular e, assim, assemelha-se à estrutura de uma matriz. Estabelece-se a seguinte correspondência:

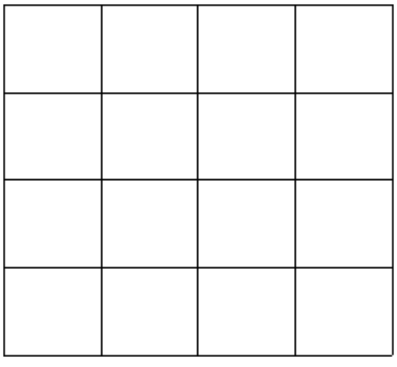
`cell(Row, Col, Value)` - célula na posição (Row, Col) com valor value

Tabela de values:

`‘empty’`	Posição vazia.
`‘bAliv’`	Posição com peça vírus do jogador 1
`’bDead'`	Posição com peça *zombie* do jogador 1
`‘rAliv’`	Posição com peça vírus do jogador 2
`'rDead'`	Posição com peça *zombie* do jogador 2

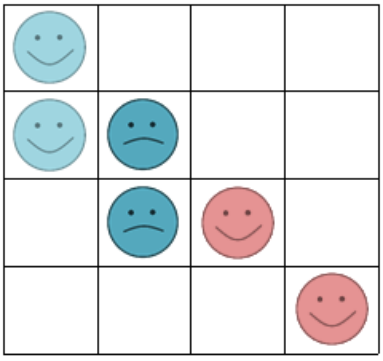
3.1.0.1. Estado Inicial

```
[ [cell(1,1,'empty') , cell(1,2,'empty') , cell(1,3,'empty') , cell(1,4,'empty')],
  [cell(2,1,'empty') , cell(2,2,'empty') , cell(2,3,'empty') , cell(2,4,'empty')],
  [cell(3,1,'empty') , cell(3,2,'empty') , cell(3,3,'empty') , cell(3,4,'empty')],
  [cell(4,1,'empty') , cell(4,2,'empty') , cell(4,3,'empty') , cell(4,4,'empty')] ]
```



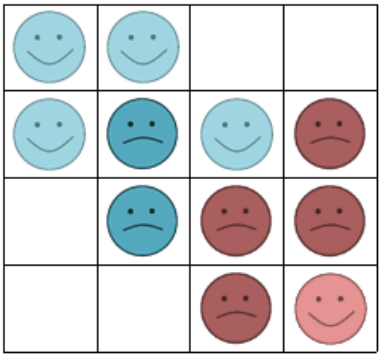
3.1.0.2. Possível Estado Intermédio

```
[ [cell(1,1,'bAliv') , cell(1,2,'empty') , cell(1,3,'empty') , cell(1,4,'empty')],
  [cell(2,1,'bAliv') , cell(2,2,'bDead') , cell(2,3,'empty') , cell(2,4,'empty')],
  [cell(3,1,'empty') , cell(3,2,'bDead') , cell(3,3,'rAliv') , cell(3,4,'empty')],
  [cell(4,1,'empty') , cell(4,2,'empty') , cell(4,3,'empty') , cell(4,4,'rAliv')] ]
```



3.1.0.3. Possível Estado Final

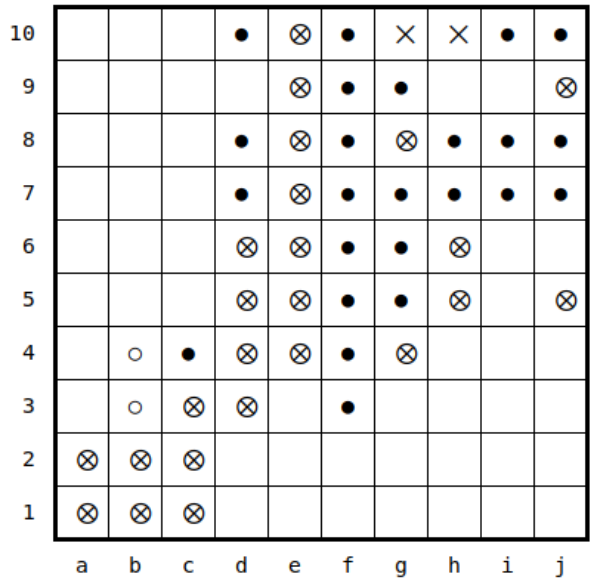
```
[ [cell(1,1,'bAliv') , cell(1,2,'bAliv') , cell(1,3,'empty') , cell(1,4,'empty')],
  [cell(2,1,'bAliv') , cell(2,2,'bDead') , cell(2,3,'bAliv') , cell(2,4,'rDead')],
  [cell(3,1,'empty') , cell(3,2,'bDead') , cell(3,3,'rDead') , cell(3,4,'rDead')],
  [cell(4,1,'empty') , cell(4,2,'empty') , cell(4,3,'rDead') , cell(4,4,'rAliv')] ]
```



3.2. Visualização do Tabuleiro

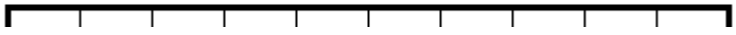
Aqui explicamos as principais funcionalidades desenvolvidas relacionadas com a visualização do tabuleiro.

- **displayGame(+Board)** *display.pl*
  Imprime o tabuleiro por completo.

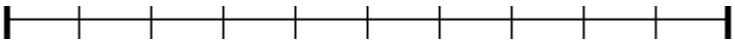


- **printLine(+Line)** *display.pl*
  Imprime uma linha do tabuleiro. Esta função inclui imprimir um número inteiro (coordenadas das linhas) seguido de alguns espaços, imprimir o primeiro caractere delimitador do tabuleiro (linha vertical densa), imprimir os elementos do tabuleiro separados com espaços e um separador (linha vertical), e, de novo, imprimir o caractere delimitador.

- **printFirstLine()** *display.pl*
  Imprime a divisória do início do tabuleiro.



- **printSeparationLine()** *display.pl*
  Imprime uma divisória intermédia do tabuleiro.



- **printFinalLine()** *display.pl*
  Imprime a divisória do fim do tabuleiro.

- **printCoordsLine()** *display.pl*
  Imprime um conjunto de letras espaçadas entre si (coordenadas das colunas).

```
a   b   c   d   e   f   g   h   i   j
```

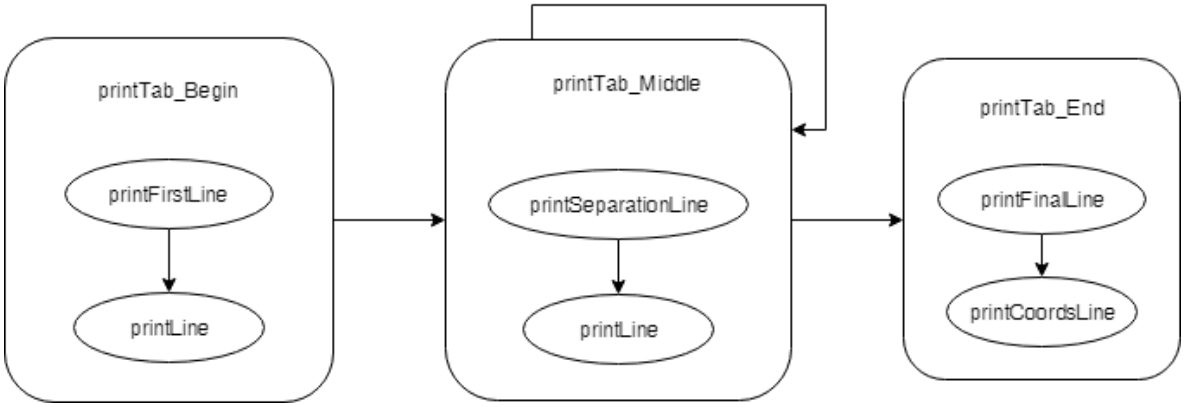


Figura 1: Diagrama de Estados da Impressão do Tabuleiro - função displayGame.

3.3. Lista de Jogadas Válidas

Por forma obter o conjunto de jogadas válidas - necessária para a escolha de uma jogada por parte do computador, por exemplo - implementamos os seguintes predicados.

- **valid_moves(+Board, +Player, -ListOfMoves)** *virus_wars.pl*
  Retorna em *ListOfMoves* uma lista de jogadas válidas para *Player*, num determinado contexto *Board*. Para efetua um findall com o predicado valid_move como argumento.

- **valid_move(+Board, +Player, +Move)** *virus_wars.pl*
  Determina se *Move* é uma jogada válida. Uma jogada é válida se se referir a uma posição do tabuleiro acessível, tal como descrito na secção 2.2.3, podendo esta estar vazia ou conter uma peça vírus do adversário.

- **checkMoveChain(Player, [Row, Col], Board)** *virus_wars.pl*
  Verifica se a posição [Row, Col] está acessível. Para tal analisa se esta se encontra adjacente a uma peça vírus de *Player*, ou, caso contrário, se existe alguma peça *zombie* do adversário que esteja. Este segundo processo é recursivo, acabando por verificar se existe uma cadeira de peças *zombie* do adversário adjacente a *Move* que se conecte em alguma posição a uma peça vírus de Player.

3.4. Execução de Jogadas

Por forma a implementar a execução de jogadas, desenvolvemos os seguintes predicados.

- **move(+Player, +PlayersType, +Board, -NewBoard, -NewPlayer, -Turn, -NewTurn) virus_wars.pl**

Este predicado é responsável pelas seguintes funcionalidades:

1. Determinar o próximo move
2. Efetuar o move e determinar o próximo jogador

1. Determinar o próximo move

O próximo move pode ser encontrado de duas formas diferentes.

No caso de se tratar de um jogador do tipo ‘user’, isto é, manual, são invocados os predicados playInput para obter a jogada que o utilizador pretende efetuar e *valid_move*, descrito na secção 3.3, para verificar se o *move* escolhido é válido.

- **playInput(+Board, -Move)** *input.pl*
  Recolhe input da próxima jogada para Move e verifica se a posição escolhida está dentro dos limites de Board.

No caso de se tratar de um jogador do tipo ‘computer’, isto é, automático, é invocado o predicado *choose_move*, descrito na secção 3.7, gera a próxima jogada segundo o nível de dificuldade escolhido pelo utilizador na interface da secção 3.8.3.

2. Efetuar o move e determinar o próximo jogador

- **makeMove(+Board, +Player, +Move, -NewBoard, -NewPlayer, +Turn, -NewTurn)** *virus_wars.pl*
  Efetua a jogada Move de Player em Board, retornando o tabuleiro novo em NewBoard, bem como o próximo jogador *NewPlayer* e o número da próxima jogada *New Turn*. Invoca o predicado *nextPlayer*.

- **nextPlayer(+Player, -NewPlayer, +Turn, -NewTurn)** *virus_wars.pl*
  Determina o novo jogador a jogar, face ao número de jogadas que ainda lhe faltam para terminar o turno. Se Turn for 0, *NewPlayer* é o oponente de *Player* e *NewTurn* é 3, para se que inicie um novo turno. Se não, *NewPlayer* corresponde a *Player* e *NewTurn* é *Turn* decrementado.

3.5. Final do Jogo

Para avaliar o fim (ou não) do jogo, desenvolvemos este predicado.

- **game_over(+Board, +Player, -Winner)**

Caso existam jogadores que ainda não tenham efetuado a sua primeira jogada, falha, pois é impossível o jogo terminar sem que cada jogador efetue, pelo menos, uma jogada. Caso contrário, verifica se não restam mais jogadas válidas a *Player* (a invocação a *valid_moves*, da secção 3.3, é uma lista de tamanho 0), e deixa em *Winner* o oponente de Player, nesse caso.

3.6. Avaliação do Tabuleiro - to complete

value(+Board, +Player, -Value)

O valor do tabuleiro *Value* para um determinado *Player* é o número de movimentos que este tem disponíveis. Isto porque o objetivo do jogo é deixar o adversário sem jogadas. Quanto maior for o número de jogadas possíveis, melhor é o *Value* do tabuleiro. Desta forma, *Value* corresponde ao tamanho da lista de *valid_moves* para um dado *Board* e *Player*.

3.7. Jogada do Computador - to complete

Os predicados relacionados com as jogadas do computador encontram-se nesta secção.

- **choose_move(+Board, +Level, -Move)** *virus_wars.pl*

Gera uma jogada *Move*, dado o estado do tabuleiro *Board* e o nível de dificuldade *Level* escolhido pelo utilizador.

Quando *Level* é 1, gera uma lista de jogadas válidas L e um número aleatório N e *Move* é a jogada na posição N em L.

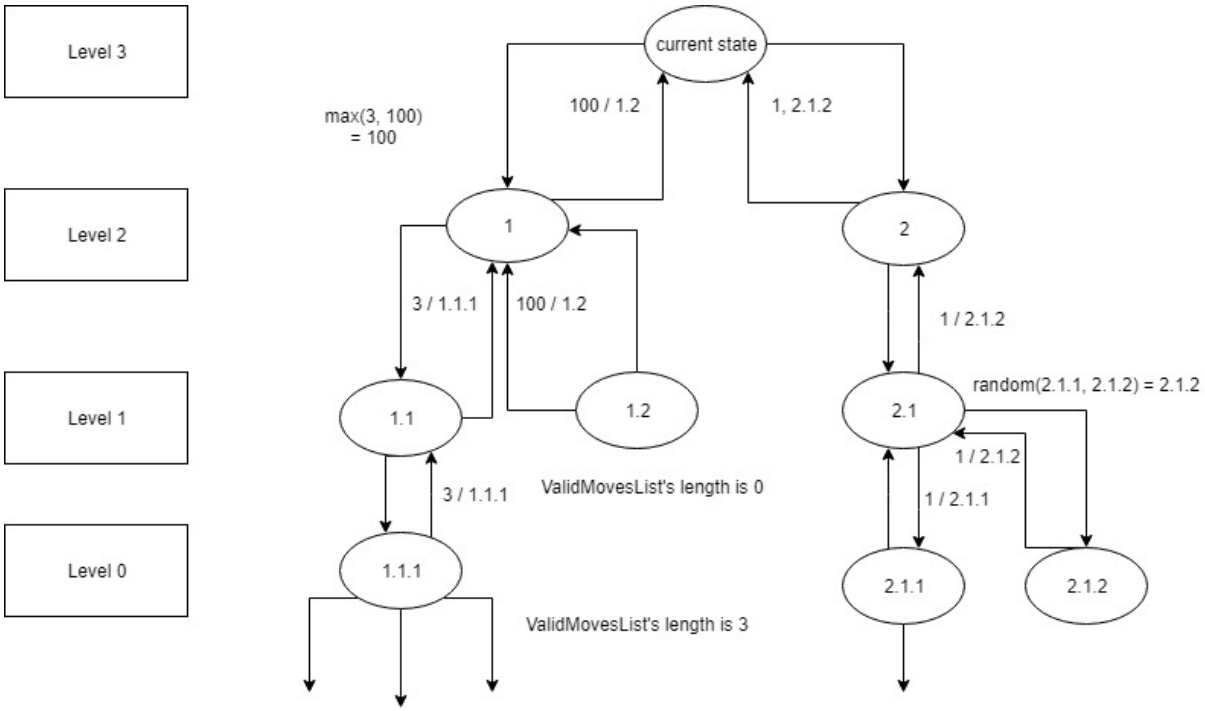
Quando Level é 2 ou 3, invoca o predicado *ai*, que avalia qual a melhor jogada que pode ser escolhida, com 1 ou 3 níveis de profundidade, conforme o utilizador tenha escolhido o nível 2 ou 3.

- **ai(+Board, +Player, +Levels, -Move)** *ai.pl*

Aplica o algoritmo minimax com *Levels* níveis de profundidade por forma a obter o melhor *Move* que pode ser efetuado.

- **minimax(+Board, +Player, +MaxPlayer, -BestNextMove, -Val, +Lvl)** *ai.pl*

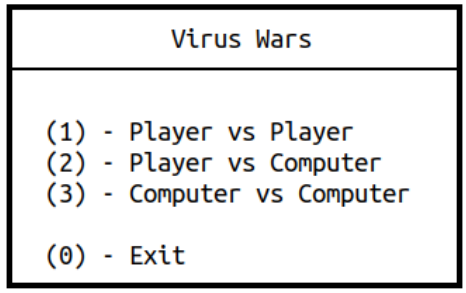
Perante um tabuleiro *Board*, determina a melhor jogada *BestNextMove* para o jogador Max*Player*, com Lvl níveis de profundidade, retornando o valor *Val* dessa jogada. *Player* é um argumento auxiliar. Para tal, é gerada a lista de jogadas válidas dado o estado do jogo, e para cada uma dessas jogadas, o estado o jogo é atualizado com o intuito de chamar minimax de novo para todas essas situações hipotéticas. Este processo é efetuado Lvl vezes. No último nível, é retornado o valor dos tabuleiros, que é dado pela função *value*, descrita na secção 3.6. Se antes, forem encontradas situações de fim de jogo, o valor retornado é 100 no caso de se tratar o utilizador, ou -100 no caso do computador. São valores muito elevados para fazer notar a ocorrência do final do jogo, que é o objetivo. Caso sejam encontradas soluções de *BestNextMove* com *Val* igual, é escolhida uma opção aleatoriamente.



3.8. Interface

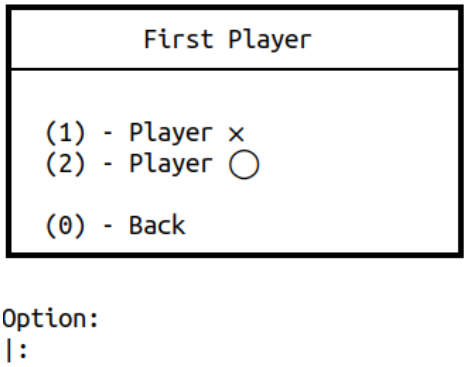
3.8.1. Menu Inicial

Interface o utilizador pode escolher o modo de jogo.



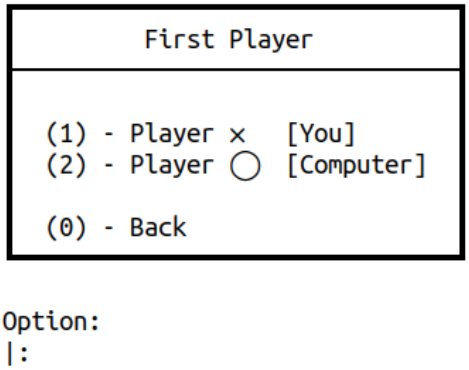
3.8.2. Menu de escolha do jogador que começa

Interface onde o utilizador pode escolher ser o primeiro ou segundo jogador.



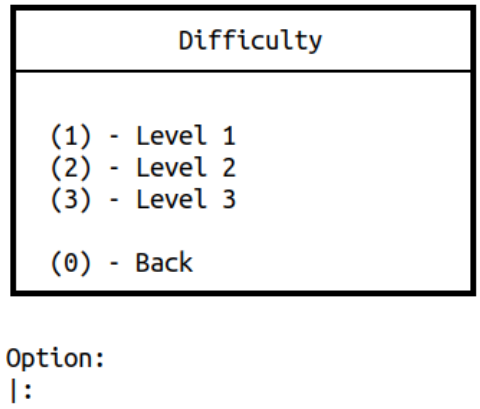
3.8.2.1. O caso Manual vs Computador

Neste caso é alertado qual dos jogadores corresponde ao utilizador e qual corresponde ao computador.



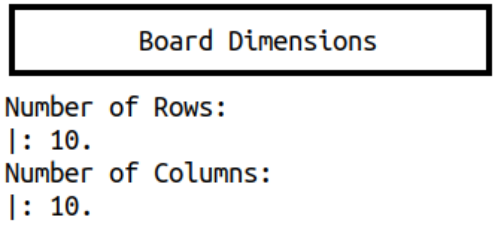
3.8.3. Menu de escolha da dificuldade do jogo (nível de IA do computador)

Interface que apenas aparece caso o utilizador tenha escolhido as opções 2 ou 3 do menu 3.8, e em que pode selecionar a dificuldade, ou seja, o nível de IA da performance do computador.



3.8.4. Menu de escolha das dimensões do tabuleiro

Interface que permite a escolha do número de linhas e colunas do tabuleiro a jogar (mínimo 4 e máximo 19 em ambas as situações).



4. Conclusões

Conclui-se que este projeto foi essencial para uma consolidação da linguagem Prolog, e, genericamente, do pensamento lógico inerente ao paradigma de programação em lógica, bastante menos familiar até então do que o paradigma de programação estruturada. Serviu ainda para a sensibilização da importância no rigor neste paradigma, em especial, pois pequenos erros podem ter um efeito negativo “em cascata” no funcionamento do programa.

Consideramos o trabalho desenvolvido bastante completo e bem estruturado. No entanto, haveria formas de o melhorar. Por exemplo, reestruturar a forma de obtenção de jogadas válidas com recurso a memorização, por forma a tornar esse processo mais eficiente e aumentar o número de níveis de inteligência artificial sem prejuízo de aumento de tempo de execução. No entanto, para as necessidades deste projeto em particular, tal otimização não seria muito percetível uma vez que se tratam de poucos níveis de IA.

Foi de resto um projeto muito interessante, e motivador para continuar a desenvolver em prolog e nas formas de desenvolver IA nessa linguagem.

5. Bibliografia

[1] A. Sandler, Virus Wars [Online]. Available: http://www.iggamecenter.com/info/en/viruswars.html

[2] (February, 2018) War of viruses [Online]. Available: https://de.wikipedia.org/wiki/Kampf_der_Viren