Programação em Lógica - 3º Ano

Virus Wars – Grupo 4

Tiago Coelho de Barros Aguiar Barbosa – 20160382

Pedro Maria Passos Ribeiro do Carmo Pereira – 201708807

Introdução

Este projeto, para a unidade curricular de Programação em Lógica, envolve a implementação de um jogo de tabuleiro, cujo tema foi escolhido entre várias possibilidades. Optámos por escolher o jogo Virus Wars.

A implementação do jogo é feita na linguagem Prolog, uma linguagem que de uso geral é especialmente associada com a inteligência artificial e linguística computacional.

Este tipo de jogos é caraterizado pelo próprio tabuleiro, pelas peças que são usadas para jogar, pelo tipo de movimentação possível das peças e pelas formas como é possível ganhar ou perder jogos.

Um dos objetivos é desenvolver três tipos de jogos diferentes Humano vs Humano, Humano vs Computador e Computador vs Computador. O Computador deve ter, pelo menos, dois níveis de inteligência diferentes.

O jogo deve conter uma interface para mostrar o tabuleiro ao utilizador e possibilitar a execução de jogadas.

O Jogo

História

Não há muito que se saiba sobre o jogo, nem a própria origem é conhecida. No entanto, na década de 1980 era regularmente jogado pelos alunos da universidade Saint Petersburg State, situada na Rússia.

Regras

Virus Wars é um jogo de tabuleiro (11 por 11, 13 por 13 ou 15 por 15) que envolve dois jogadores, cujo objetivo é obter o controlo do tabuleiro e deixar o oponente sem jogadas legais.

Cada jogador tem a sua colónia de vírus e em cada jogada tem a opção de fazer crescer a colónia, acrescentando uma peça ao tabuleiro, ou absorver uma peça adversária, tornando a peça num zombie.

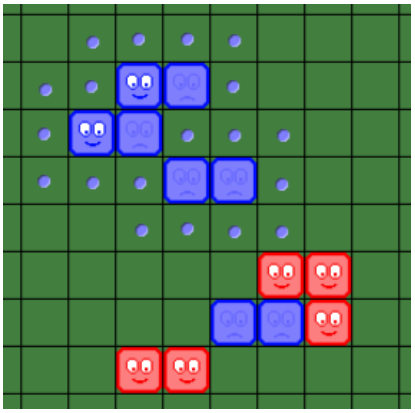
Existem quatro tipo de peças no Virus Wars: os vírus azuis (quem as possui realiza a primeira jogada do encontro) e os vírus vermelhos, os zombies azuis e os zombies vermelhos. Ao gerar um novo vírus acrescenta-se, num espaço vazio do tabuleiro, uma peça vírus. Ao fazer uma absorção, substitui-se o vírus adversário por um zombie. Um zombie nunca pode ser absorvido pelo adversário. Nas imagens abaixo, podemos observar as duas peças que representam os vírus e um zombie azul.



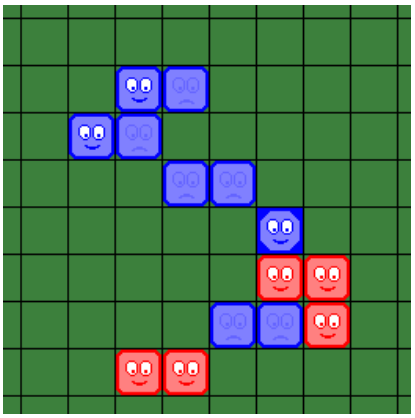
O encontro começa com o tabuleiro vazio e quem controla as peças azuis, deve colocar uma do lado esquerdo do tabuleiro e a primeira peça vermelha deve, também, ser colocada no lado direito do tabuleiro.

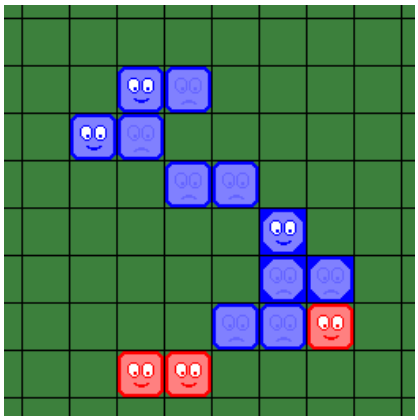
Cada turno de um jogador requer 5 jogadas obrigatórias. Uma jogada é equivalente a acrescentar uma peça ou a uma absorção. As jogadas possíveis podem ser feitas adjacentemente a um vírus presente no tabuleiro, nas direções verticais, horizontais ou diagonais ou adjacentemente a um zombie que esteja ligado a um vírus ou adjacentemente a um zombie que esteja ligado a outro zombie ligado a um vírus e assim consecutivamente.

A figura abaixo representa um exemplo de um tabuleiro e as jogadas possíveis naquela situação (marcadas com pontos azuis). É de notar que não existem jogadas possíveis em torno dos dois zombies na parte de baixo, pois não se encontram ligados a nenhum vírus "vivo".



As duas imagens seguintes representam 3 jogadas consecutivas que sucedem o cenário anterior: a geração de um novo vírus, seguido da absorção dos dois vírus vermelhos adjacentes à nova peça.





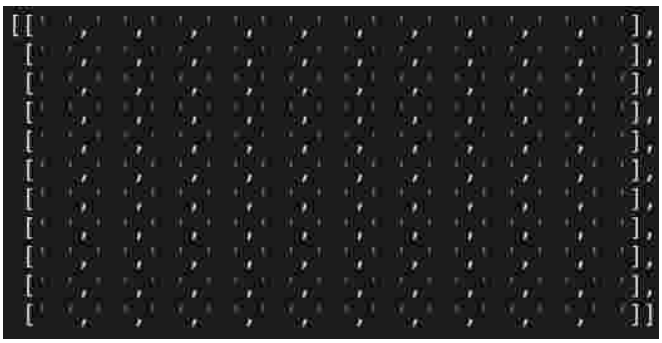
O jogo termina quando um dos jogadores não consegue completar 5 jogadas seguidas legais, o que significa que o seu adversário venceu.

Lógica do Jogo

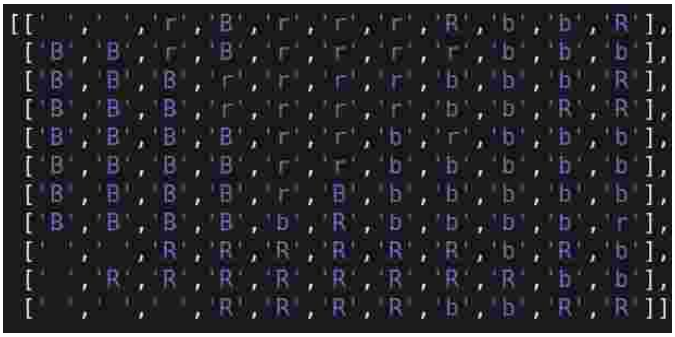
Representação interna do estado do jogo

Para a representação interna do estado do jogo, optámos por utilizar uma matriz (lista de listas) em que cada posição de uma lista contém o carater correspondente à célula que ocupa essa posição. O carater 'R' representa um vírus vermelho e ‘r’ um zombie vermelho, ‘B’ um vírus azul e ‘b’ um zombie azul, e ‘ ‘ para uma célula livre.

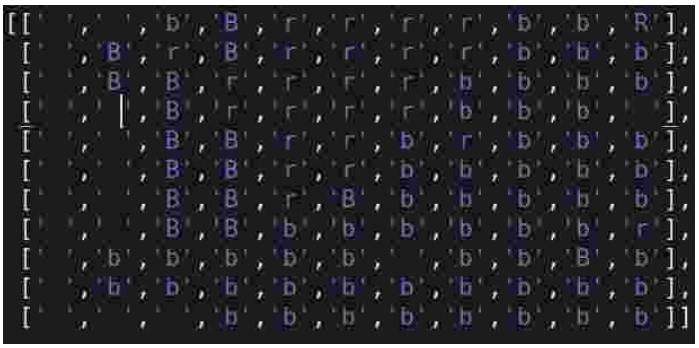
Na imagem abaixo podemos observar a imagem da representação interna de um tabuleiro inicial, sem peças, 11 por 11.



Na próxima imagem, um exemplo de um tabuleiro possível num jogo a decorrer.



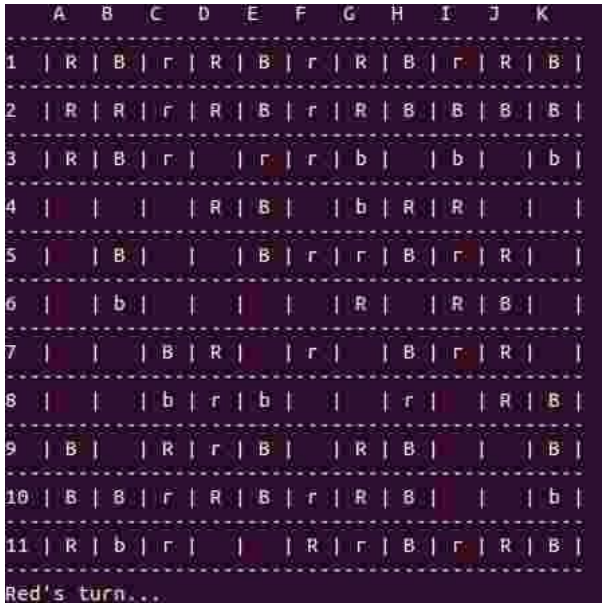
E por fim, um exemplo de um jogo terminado onde o lado azul ganhou, já que o lado vermelho tem apenas um vírus que está rodeado de zombies azuis (relembrando que os zombies não podem ser absorvidos e apenas se pode jogar em torno de um zombie se ele tiver ligado a um vírus, o que deixa o lado vermelho sem jogadas legais possíveis).



Visualização do tabuleiro

A função display_game toma como argumentos a representação interna do estado atual do jogo, na matriz referida anteriormente e o jogador que irá realizar o turno.

Um exemplo de um tabuleiro imprimido quando a função é chamada é o seguinte:



Lista de Jogadas Válidas

```
valid_moves(Board, red, ListOfMoves):- getSize(Board, Size),
                                       getRedsCells(Board,  Size, 0, [], RedsCells),
                                       getCellsPlays(RedsCells, red, Board, [], ListOfMoves), !.
valid_moves(Board, blue, ListOfMoves):- getSize(Board, Size),
                                        getBluesCells(Board,  Size, 0, [], BluesCells),
                                        getCellsPlays(BluesCells, blue, Board, [], ListOfMoves), !.
```

No predicado valid_moves decidimos pesquisar, para cada jogador, as suas células disponíveis em jogo através dos predicados getRedsCells e getBluesCells e, depois, são calculadas as jogadas possíveis para cada célula através de getCellsPlays. Esta última função verifica as jogadas legais para uma certa célula, seguindo as regras descritas anteriormente. Verifica as posições adjacentes e se pode jogar em alguma delas, sendo uma absorção ou uma geração.

Execução de Jogadas

```
move(play(blue, pos(C, L)), Board, Board):- valid_moves(Board, blue, LM), not(member(pos(C,L), LM)), write("\nInvalid Position\n"), !.
move(play(blue, pos(C, L)), Board, NewBoar):- valid_moves(Board, blue, LM), member(pos(C,L), LM), getIndexMatrix(C, L, Board, Elem),
                                              (Elem = ' ' -> alterPos(C, L, Board, 'B', [], NewBoar);
                                              alterPos(C, L, Board, 'b', [], NewBoar)), !.
```

O predicado move começa por calcular as jogadas possíveis (valid_moves) para o jogador e após isso verifica se a jogada escolhida pelo jogador (na coluna C e linha L) pertence à lista de valid_moves. Se esta condição não se verificar, lança uma mensagem de erro. Se se verificar, a célula com as coordenadas indicadas pelo utilizador irá ser atualizada devidamente, seja para um zombie da cor contrária, se já existisse lá um vírus, ou para um vírus, se nada lá existisse.

Final do Jogo

```
game_over(Board, blue):- valid_moves(Board, red, []), !.
game_over(Board, red):- valid_moves(Board, blue, []), !.
```

A função game_over é bastante simples e direta. Verifica apenas, para cada jogador, as suas jogadas disponíveis. Se não existirem jogadas para uma certa pessoa, a outra ganha o jogo.

Avaliação do Tabuleiro

Neste predicado decidimos não criar um bot "demasiado inteligente", pois cada turno são 5 jogadas seguidas pelo mesmo jogador, e com bots que avaliariam demasiadas opções de jogada, o programa chegaria a um ponto onde ficaria lentíssimo ao ponto de se ter que esperar vários minutos até acabar um turno. Por esta razão decidimos na seguinte estrutura para o predicado value:

```
value(Board, TestedBoard, Player, 10):- game_over(TestedBoard, Player), !.
value(Board, TestedBoard, red, 0):- game_over(TestedBoard, blue), !.
value(Board, TestedBoard, blue, 0):- game_over(TestedBoard, red), !.
value(Board, TestedBoard, red, 2):- count_elemM('r', TestedBoard, 0, NN), count_elemM('r', Board, 0, N), N < NN, !.
value(Board, TestedBoard, red, 1):- count_elemM('R', TestedBoard, 0, NN), count_elemM('R', Board, 0, N), N < NN, !.
value(Board, TestedBoard, blue, 2):- count_elemM('b', TestedBoard, 0, NN), count_elemM('b', Board, 0, N), N < NN, !.
value(Board, TestedBoard, blue, 1):- count_elemM('B', TestedBoard, 0, NN), count_elemM('B', Board, 0, N), N < NN, !.
```

Resumindo, quanto maior for o valor do parâmetro Value numa jogada, mais prioridade o bot vai dar a essa jogada sobre outras. Por isso é que uma jogada que faça o bot vencedor vale 10, mais que qualquer outra jogada.

O bot nunca deve dar prioridade a uma jogada que o faça perder, por isso Value é 0.

No resto das jogadas, o bot deve dar prioridade a uma jogada em que faça uma absorção, ou seja, quando o número de zombies do tabuleiro resultante (TestedBoard) é maior do que o tabuleiro a ser avaliado.

Jogada do Computador

```
chose_move(Board, ai1, Player, Move):- valid_moves(Board, Player, LM), getSize(LM, Moves),
                                       SupLim is Moves - 1, random_between(0, SupLim, Index),
                                       getIndexList(Index, LM, Move), !.
chose_move(Board, ai2, Player, Move):- valid_moves(Board, Player, LM), maxValueMove(LM, Board, Player, 0, 0, Move), !.
```

Neste tópico optámos por dois níveis de dificuldade para o computador, um básico que joga aleatoriamente e outro mais avançado que segue as jogadas que irão criar o tabuleiro com mais valor.

No computador básico, a jogada a ser executada é escolhida de forma aleatória dentro da lista de valid_moves para o computador.

Por outro lado, no computador mais avançado, também são calculados os movimentos possíveis, mas é chamada a função maxValueMove. A maxValueMove atualiza o parâmetro Move para o movimento legal com maior valor possível, seguindo as regras do predicado value.

Conclusões

Ao desenvolver e testar este projeto apercebemo-nos da importância de bons algoritmos, uma vez que o predicado valid_moves, em fases finais de um jogo entre dois computadores, demora demasiado tempo a processar, o que leva ao utilizador a ter que esperar vários segundos até que seja completado um turno. Por esta razão, um aspeto a melhorar no nosso projeto seria a eficiência do algoritmo que utilizámos para calcular as jogadas legais.

Bibliografia

IgGameCenter - Virus Wars: http://www.iggamecenter.com/info/en/viruswars.html

Swi-Prolog Official Site: http://www.swi-prolog.org/