Virus Wars

Relatório Final

Mestrado Integrado em Engenharia Informática e Computação

Programação em Lógica

Grupo 03:
João Carlos Fonseca Pina de Lemos - 201000660
Luís Guilherme da Costa Castro Neves - 201306485

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

20 de Novembro de 2018

1 Resumo

Na cadeira de Programação em Lógica do Mestrado Integrado em Engenharia Informática e de Computação foi-nos pedido a realização de um projeto no qual, utilizando a linguagem de programação Prolog criássemos uma versão digital do jogo de tabuleiro Virus Wars.

Neste Relatório apresentaremos as regras e contexto histórico do jogo assim como o trabalho desenvolvido até à data de entrega deste relatório

Conteúdo

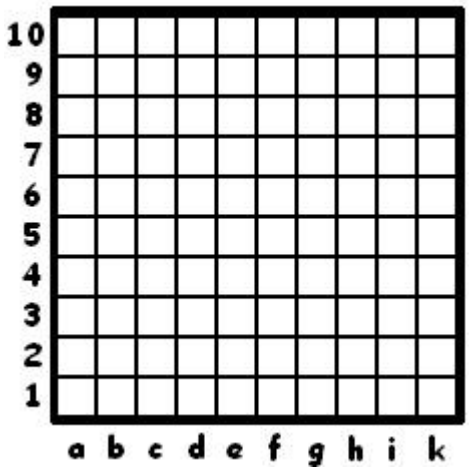


Figura 1: Tabuleiro 10x10

2 O Jogo Virus Wars

2.1 Contexto Histórico

Apesar de ter origem desconhecida o jogo Virus Wars era ativamente jogado com papel e caneta na Universidade de Petersburg durante a década de 80. É um jogo de dois jogadores que simula a evolução de duas colónias de virus que vão crescendo e absorvendo-se uma à outra. Existe uma variedade de jogos com regras ligeiramente diferentes dependendo do jogo escolhido.

2.2 Regras

Para jogar Virus Wars podem ser usados tabuleiros quadrados de diferentes dimensões, sendo que o clássico era jogado num tabuleiro de dimensões 10x10.

Existem 4 tipos de peças disponíveis ao jogadores: as peças de virus azul e vermelha e as peças zombies azuis e vermelhas que simbolizam as peças zombificadas pelo jogador.
Cada jogador deve realizar cinco jogadas por turno. Cada jogada implica colocar um virus novo em jogo (propagar) ou 'zombificar' (assimilar) um dos virus do adversário. Chamamos propagação ao ato de colocra um novo virus, da cor do jogador, em qualquer célula vazia que esteja disponível no tabuleiro e Assimilação quando zombificamos um virus do adversário em qualquer uma das células do tabuleiro que estejam disponíveis, isto é, substituindo um virus do adversário com a especial peça "zombie", da cor do jogador. Os "Zombies"permanecem estáveis e estáticos até ao fim do jogo não podendo ser reanimados, recuperados ou removidos do tabuleiro de jogo. Uma célula diz-se disponível quando obedece às seguintes condições:

- A "célula"está vertical, horizontal or diagonalmente adjacente a um virus do jogador, já colocado no tabuleiro (ainda que esse virus tenha sido colocado na ultima jogada, durante o mesmo turno);
- A "célula"está ligada a um virus do jogador, já existente no tabuleiro, através de uma cadeia de "zombies"interligados, da mesma cor que o jo-

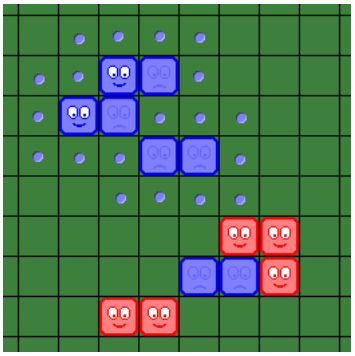
gador, quer horizontal, vertical ou diagonalmente adjacentes (ainda que esses "zombies"sejam gerados na ultima jogada, durante o mesmo turno).

O objetivo do jogo é deixar o adversário sem jogadas válidas.
O jogo inicia-se com o tabuleiro vazio e não existem células disponiveis, ambos os jogadores possuem um número ilimitado de virus da cor escolhida pelo jogador: Azul ou Vermelha. O jogador das peças Azuis pode colocar o seu primeiro virus em qualquer uma das células, na zona esquerda do tabuleiro e o jogador das peças Vermelhas pode colocar o seu primeiro virus, em qualquer uma das células no canto direito do tabuleiro. Cada jogador deve realizar as 5 jogadas durante o seu turno, em caso de impossibilidade de completar as 5 jogadas o jogador em questão perderá o jogo.

2.3 Exemplo

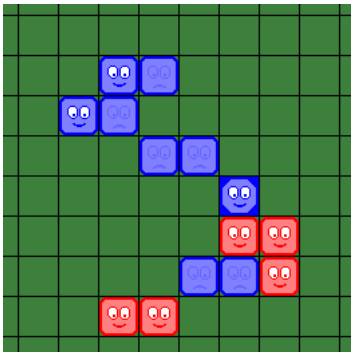
Iremos agora analizar um exemplo de um turno do ponto de vista do jogador que controla as peças azuis passo a passo.

2.3.1 Jogada 1



Jogadas Possíveis 1

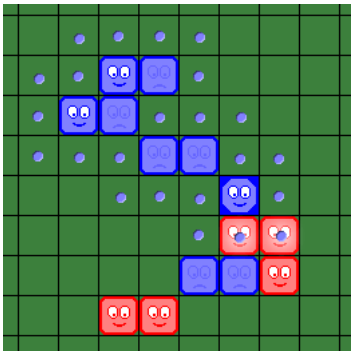
Na situação apresentada, todas as células disponíveis ao jogador Azul estão assinaladas com pontos azuis. Tenha em conta que as células adjecentes às peças zombies azuis mais abaixo não estão assinaladas uma vez que estas peças zombies não estão ligadas às células azuis 'vivas.'



Propagação

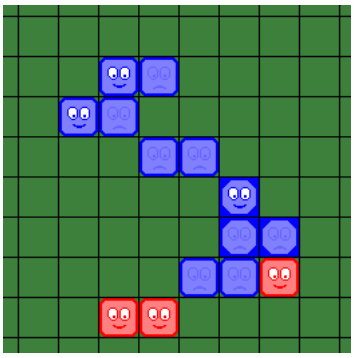
O jogador Azul faz a sua primeira jogada e coloca o seu virus numa das células disponiveis.

2.3.2 Jogada 2 e 3



Jogadas Possíveis 2

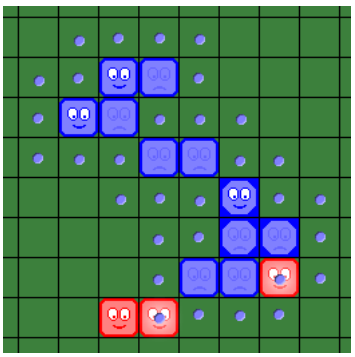
O novo vírus azul dá ao jogador mais 5 células disponíveis sendo que, agora duas peças vermelhas podem ser assimiladas.



Assimilação

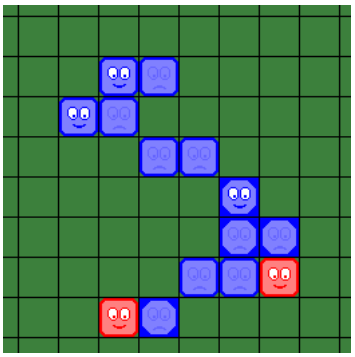
O jogador Azul faz as suas segunda e terceira jogadas assimilando os dois virus vermelhos.

2.3.3 Jogada 4



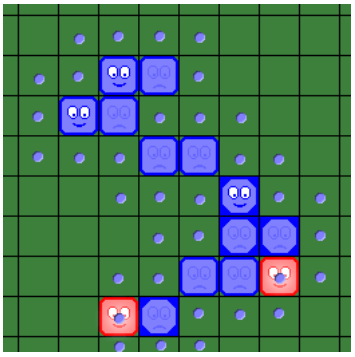
Jogadas Possíveis 3

As ultimas jogadas deram ao jogador azul acesso a mais dois virus vermelhos, sendo que vai assimilar um deles.



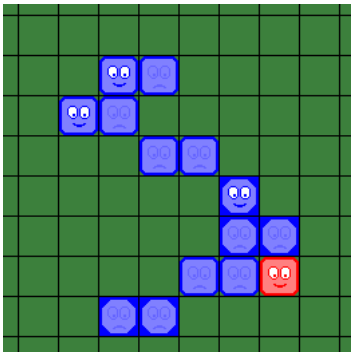
Assimilação 2

2.3.4 Jogada 5



Jogadas Possíveis 4

Agora, na ultima jogada do turno o jogador azul pode, novamente assimilar duas peças vermelhas. Estando terminada esta jogada o jogador vermelho fica com uma peça e é a sua vez de jogar.



Assimilação 3

3 Lógica do Jogo

3.1 Representação do estado de jogo

Para a representação do estado de jogo (ainda não desenvolvida em código concordante com o apresentado no anexo 1) iremos pegar num tabuleiro 5x5 fazendo uma lista que contém 5 listas que contém 5 listas, no caso de tabuleiros de diferentes dimensões, uma vez que são todos quadrados as dimensões seriam NxN. O jogo utiliza 4 tipos de peças:

- Peças Azuis - Representadas por um B (Blue);
- Peças Vermelhas - Representadas por um R (Red);
- Peças Zombificadas Azuis e Vermelhas - Representadas por um zb e zr respetivamente.

Células sem nenhuma peça são represtadas por uma lista vazia nessa célula.

O tabuleiro inicial 5x5 é representado pelo seguinte código:

```
createEmptyBoard(Board) :-
        Board = [
                [ [], [], [], [], [] ], % x = 0, y = 0 to y = 4
            [ [], [], [], [], [] ],
            [ [], [], [], [], [] ],
            [ [], [], [], [], [] ],
            [ [], [], [], [], [] ]].% x = 4
```

O tabuleiro em estado intermédio de jogo pode ser representado por:

```
new_test(Board) :-
    Board = [
        [[],[red],[],[],[]],
        [[],[zb],[],[],[]],
        [[],[],[],[blue],[]],
        [[],[zr],[],[],[]],
        [[],[],[],[],[red]]
    ].
```

Como no problema apresentado o tabuleiro inicial podia ter diversos tamanhos sendo sempre quadrado desenvolvemos um predicado para gerar uma matriz com as dimensões pretendidas. Consideramos esta execussão melhor do que desenvolver várias versões do mesmo predicado para cada tabuleiro no nosso range.

```
create_empty_board(Length, Width, Board) :-
    length(Row, Width),
    maplist(=([]), Row),
    length(Board, Length),
    maplist(=(Row), Board).
```

3.2 Vizualisação do tabuleiro

Usando o código desenvolvido e apresentado no Anexo 1 chegamos à vizualização do tabuleiro de jogo vazio:

```
| ?- start(_).

Choose board Size
Board Size must be between 5 and 15
Input Value: 8.

You have choosen a 8x8 board
     A    B    C    D    E    F    G    H
   ---------------------------------------
1  |    |    |    |    |    |    |    |    |
   ---------------------------------------
2  |    |    |    |    |    |    |    |    |
   ---------------------------------------
3  |    |    |    |    |    |    |    |    |
   ---------------------------------------
4  |    |    |    |    |    |    |    |    |
   ---------------------------------------
5  |    |    |    |    |    |    |    |    |
   ---------------------------------------
6  |    |    |    |    |    |    |    |    |
   ---------------------------------------
7  |    |    |    |    |    |    |    |    |
   ---------------------------------------
8  |    |    |    |    |    |    |    |    |
   ---------------------------------------
```

Tabuleiro Inicial

A nossa interpretação pergunta ao utilizador as dimensões do tabuleiro que deseja utilizar (tendo as dimensões sido limitadas por nós entre 5 e 10) e desenha-o colocando também as referências das coordenadas que futuramente ajudarão o utilizador a colocar as peças na posição pretendida. Atualmente a nossa implementação funciona para qualquer tipo de tabuleiro nos limites estipulados.

Para desenharmos o tabuleiro implementamos a função draw_board(Board,Size). Esta função para além de desenhar o tabuleiro e as peças que estejam lá estejam cria também as referências para facilitar ao utilizador colocar peças em jogo.

```
draw_board(Board,Size):-
    Y is 0,
    Alphabet=['NULL','A','B','C','D','E','F','G','H','I','J',
    'K','L','M','N','O'],
    draw_margin(_),
    draw_column_name(Size,1,Alphabet),
    draw_margin(_),
    draw_horizontal_padding(Size),
    draw_board_interior(Board, Y, Size).

draw_board_interior(Board, Y, Size):-
    Y < Size,
    Line is Y + 1,
    write(Line),
    write('  '),
    draw_vertical_padding(Board, 0, Y, Size),
    draw_margin(_),
```

```
    draw_horizontal_padding(Size),
    NewY is Y+1,
    draw_board_interior(Board, NewY, Size).

draw_board_interior(_, Size, Size).

draw_horizontal_padding(Size):-
    Size>0,
    write('-----'),
    X is Size-1,
    draw_horizontal_padding(X).

draw_horizontal_padding(0):-
    write('-\n').


draw_vertical_padding(Board, X, Y, Size):-
    X < Size,
    getCell(Board, X, Y, Cell),
    write('|'),
    draw_cell(Cell),
    %write('     '),
    NewX is X+1,
    draw_vertical_padding(Board, NewX, Y, Size).

draw_vertical_padding(_, Size, _, Size):-
    Size = Size,
    write('|\n').
```

Sendo que o indice do eixo horizontal é escrito usando a função:

```
draw_column_name(Size ,X, Alphabet):-
    X =< Size,
    nth0(X, Alphabet, Output),
    write('  '),
    write(Output),
    write('  '),
    Y is X+1,
    draw_column_name(Size ,Y, Alphabet).

draw_column_name(Size, Size, _):-
    write('\n').
```

Para colocar as peças em jogo, usamos o predicado place_piece(Board, X, Y, Piece, NewBoard, Size) existente em board.pl, este predicado para colocar as peças em jogo usa a ferramenta append, isto tendo em conta as regras mencionadas em cima faz com que no código possam estar ao mesmo tempo numa unica célula duas peças. Na visualização para contornarmos esse problema foi desenvolvido o seguinte código que já tinha em conta o erro anterior.

```
draw_cell(Cell):-
    Cell == [blue],
    write('blue').
```

```
draw_cell(Cell):-
    Cell == [zb],
    write('zomb').

draw_cell(Cell):-
    Cell == [zr],
    write('zomr').

draw_cell(Cell):-
    Cell == [red,zb],
    write('zomb').

draw_cell(Cell):-
    Cell == [blue,zr],
    write('zomr').

draw_cell(Cell):-
    Cell == [red,blue],
    write('zomb').

draw_cell(Cell):-
    Cell == [blue,red],
    write('zomr').

draw_cell(Cell):-
    Cell == [],
    write('    ').

draw_cell(Cell):-
    Cell == [red],
    write('red ').

draw_cell(_):-
    write('    ').
```

3.3 Jogadas

Para a aceitação de jogadas desenvolvemos os predicados:

```
makePlay(Piece, InBoard, Size, PlayerName, OutBoard, Num):-
    write_player_name(PlayerName, Num),
    ask_X_input(_),
    get_X_input(X),
    ask_Y_input(Size),
    get_Y_input(Y),
    place_piece(InBoard, X, Y, Piece, OutBoard, Size), write(OutBoard),nl.

place_piece(Board, X, Y, Piece, NewBoard, Size) :-
        checkValidMove(Piece, Board, X, Y, Size),
        nth0(Y, Board, Column),
        nth0(X, Column, Cell),
    select_piece(Cell, Piece, NewPiece),
        append(Cell, NewPiece, NewCell),
        replaceAtListIndex(Column, X, NewCell, NewColumnValue),
        replaceAtListIndex(Board, Y, NewColumnValue, NewBoard).

checkValidMove(Piece, Board, X, Y, Size):-
    check_in_bounds(X, Y, Size),
    check_valid(Piece, Board, X, Y).
```

A validação da peça é feita pelo predicado select_piece(Cell, Piece, NewPiece).

```
select_piece(Cell, Piece, NewPiece):-
    Cell == [],
    NewPiece = Piece.

select_piece(Cell, Piece, NewPiece):-
    Cell == [red],
    Piece == [blue],
    NewPiece = [zb].

select_piece(Cell, Piece, NewPiece):-
    Cell == [blue],
    Piece == [red],
    NewPiece = [zr].

select_piece(_, _, _):-
   write('Invalid Piece'),nl,fail.
```

A verificação das regras de posição na colocação de uma peça (apenas pode ser colocada existindo uma peça da mesma cor adjacente é feita através do seguinte código:

```
check_valid(Piece, Board, X, Y):-
    Piece == [blue],
    P = [red,zb],
    P2 = [zb],
    Right is X+1,
    Up is Y-1,
    Left is X-1,
```

```
    Down is Y+1,
    (check_adjacent_coodenates(Piece, Board,
    X, Y, Right, Up, Left, Down);
    check_adjacent_coodenates(P, Board,
    X, Y, Right, Up, Left, Down);
    check_adjacent_coodenates(P2, Board,
    X, Y, Right, Up, Left, Down)).

check_adjacent_coodenates(Piece, Board,
X, Y, Right, Up, Left, Down):-
    (check_if_ok(Piece, Board, Left, Up);
    check_if_ok(Piece, Board, X, Up);
    check_if_ok(Piece, Board, Right, Up);
    check_if_ok(Piece, Board, Left, Y);
    check_if_ok(Piece, Board, Right, Y);
    check_if_ok(Piece, Board, Left, Down);
    check_if_ok(Piece, Board, X, Down);
    check_if_ok(Piece, Board, Right, Down)).

check_if_ok(Piece, Board, X, Y):-
    nth0(Y, Board, Column),
        nth0(X, Column, Cell),
    Cell == Piece.
```

3.4 Ciclo de jogo

Na primeira fase é pedido aos jogadores para colocarem uma peça de cala lado do campo, o jogador com as peças azuis junto à margem esquerda e o jogador com as peças vermelhas junto à margem direita.

```
loop_one(Board, Size, Type):-
    Type = 1,
    place_first_blue_player(Board, Size, 'Player1', NewBoard),
    draw_board(NewBoard, Size),
    place_first_red_player(NewBoard, Size, 'Player2', NewerBoard),
    draw_board(NewerBoard, Size),
    game_loop(NewerBoard, Size, Type).

place_first_blue_player(Board, Size, PlayerName, NewBoard):-
    Piece = [blue],
    X is 0,
    display_firstPlayer_message(PlayerName),
    get_Y_input(Y),
    nth0(Y, Board, Column),
        nth0(X, Column, Cell),
        append(Cell, Piece, NewCell),
        replaceAtListIndex(Column, X, NewCell, NewColumnValue),
        replaceAtListIndex(Board, Y, NewColumnValue, NewBoard).

place_first_red_player(Board, Size, PlayerName, NewBoard):-
    Piece = [red],
    X is Size-1,
    display_secoundPlayer_message(PlayerName),
    get_Y_input(Y),
    nth0(Y, Board, Column),
        nth0(X, Column, Cell),
        append(Cell, Piece, NewCell),
        replaceAtListIndex(Column, X, NewCell, NewColumnValue),
        replaceAtListIndex(Board, Y, NewColumnValue, NewBoard).
```

Depois desta primeira faze o jogo inicia o ciclo normal de 5 jogadas por turno por jogador.

```
game_loop(Board, Size, Type):-
    Type = 1,
    play_cycle_player([blue], Board, Size, 'Player1', NewBoard),
    play_cycle_player([red], NewBoard, Size, 'Player2', NewerBoard),
    game_loop(NewerBoard, Size, Type).

play_cycle_player(Piece, InBoard, Size, PlayerName, OutBoard) :-
    makePlay(Piece, InBoard, Size, PlayerName, OutB1, 1),
    draw_board(OutB1, Size),
    makePlay(Piece, OutB1, Size, PlayerName, OutB2, 2),
    draw_board(OutB2, Size),
    makePlay(Piece, OutB2, Size, PlayerName, OutB3, 3),
    draw_board(OutB3, Size),
    makePlay(Piece, OutB3, Size, PlayerName, OutB4, 4),
    draw_board(OutB4, Size),
```

```
    makePlay(Piece, OutB4, Size, PlayerName, OutBoard, 5),
    draw_board(OutBoard, Size).

makePlay(Piece, InBoard, Size, PlayerName, OutBoard, Num):-
    write_player_name(PlayerName, Num),
    ask_X_input(_),
    get_X_input(X),
    ask_Y_input(Size),
    get_Y_input(Y),
    place_piece(InBoard, X, Y, Piece, OutBoard, Size), write(OutBoard),nl.
```

3.5 Em falta

Aquando a execussão e subimissão deste projeto e relatório houveram 3 pontos que não conseguimos terminar, a primeira delas foi uma regra de jogo descrita no ponto 2, outra foi a condição de fim de jogo para a qual não conseguimos desenvolver o perdicado necessário para tal, e a terceira foi a integração de npcs, ponto no qual nem chegamos a tocar não tendo conseguido desenvolver os dois pontos anteriores. No entanto tentaremos ter estes pontos desenvolvidos aquando a apresentação prática do trabalho em aula.

Referências

[1] igGameCenter, *"Virus Wars" http://www.iggamecenter.com/info/pt/viruswars.html*

[2] Musikhin A.K., *"War of viruses" // Logic or fortune? Games for everyone.*

A Código Fonte

O ficheiro em questão está disponível em:
https://drive.google.com/drive/folders/1FFdCJGfV6Af6-ttwTQWsXS-gXBTUasUg?usp=sharing

A.1 main.pl

```
:-[board].
:-[interface].
:-[cycle].
:-[helper_lib].


start(_):-
    displayTitleCard(_),
    repeat,
    displayInstructions(_),
    chooseOption(_),
    %chooseGameType(Type),
    choose_board_size(Size),
    create_empty_board(Size, Size, Board),
    draw_board(Board,Size),
    Type is 1,
    loop_one(Board, Size, Type).
    %game_loop(Board,Size,Type).

chooseGameType(Choice):-
    display_game_types(_),
    choose_game_type(Choice).

test(_):-
    new_test(Board),
    write('Board 1: '), nl,
    draw_board(Board,5), nl,
    place_piece(Board, 2, 2, [red], NewBoard, 5),
    write('NewBoard: '), nl,
    draw_board(NewBoard,5),
    place_piece(NewBoard, 3, 5, [blue], NewerBoard, 5),
    draw_board(NewerBoard, 5),
    write(NewerBoard).
```

A.2 board.pl

```
:- use_module(library(lists)).
:- [helper_lib].

%create_empty_board(Board, Size):-
 %    generate_matrix(Size, Size, Board).

create_empty_board(Length, Width, Board) :-
    length(Row, Width),
    maplist(=([]), Row),
```

```
    length(Board, Length),
    maplist(=(Row), Board).

%returns the cell at X,Y position in board
getCell(Board, X, Y, Cell) :-
        nth0(Y, Board, Column),
        nth0(X, Column, Cell).

place_piece(Board, X, Y, Piece, NewBoard, Size) :-
        checkValidMove(Piece, Board, X, Y, Size),
        nth0(Y, Board, Column),
        nth0(X, Column, Cell),
    select_piece(Cell, Piece, NewPiece),
        append(Cell, NewPiece, NewCell),
        replaceAtListIndex(Column, X, NewCell, NewColumnValue),
        replaceAtListIndex(Board, Y, NewColumnValue, NewBoard).

select_piece(Cell, Piece, NewPiece):-
    Cell == [],
    NewPiece = Piece.

select_piece(Cell, Piece, NewPiece):-
    Cell == [red],
    Piece == [blue],
    NewPiece = [zb].

select_piece(Cell, Piece, NewPiece):-
    Cell == [blue],
    Piece == [red],
    NewPiece = [zr].

select_piece(_, _, _):-
   write('Invalid Piece'),nl,fail.

check_in_bounds(X, Y, Size) :-
    X >= 0, X < Size,
    Y >= 0, Y < Size.


checkValidMove(Piece, Board, X, Y, Size):-
    check_in_bounds(X, Y, Size),
    check_valid(Piece, Board, X, Y).
    %check_valid_piece(X, Y, Board).

check_valid(Piece, Board, X, Y):-
    Piece == [red],
    P = [blue,zr],
    P2 = [zr],
    Right is X+1,
    Up is Y-1,
    Left is X-1,
    Down is Y+1,
```

```
    (check_adjacent_coodenates(Piece, Board, X, Y, Right, Up, Left, Down);
    check_adjacent_coodenates(P, Board, X, Y, Right, Up, Left, Down);
    check_adjacent_coodenates(P2, Board, X, Y, Right, Up, Left, Down)).


check_valid(Piece, Board, X, Y):-
    Piece == [blue],
    P = [red,zb],
    P2 = [zb],
    Right is X+1,
    Up is Y-1,
    Left is X-1,
    Down is Y+1,
    (check_adjacent_coodenates(Piece, Board, X, Y, Right, Up, Left, Down);
    check_adjacent_coodenates(P, Board, X, Y, Right, Up, Left, Down);
    check_adjacent_coodenates(P2, Board, X, Y, Right, Up, Left, Down)).

check_adjacent_coodenates(Piece, Board, X, Y, Right, Up, Left, Down):-
    (check_if_ok(Piece, Board, Left, Up);
    check_if_ok(Piece, Board, X, Up);
    check_if_ok(Piece, Board, Right, Up);
    check_if_ok(Piece, Board, Left, Y);
    check_if_ok(Piece, Board, Right, Y);
    check_if_ok(Piece, Board, Left, Down);
    check_if_ok(Piece, Board, X, Down);
    check_if_ok(Piece, Board, Right, Down)).

check_if_ok(Piece, Board, X, Y):-
    nth0(Y, Board, Column),
        nth0(X, Column, Cell),
    Cell == Piece.

place_first_blue_player(Board, Size, PlayerName, NewBoard):-
    Piece = [blue],
    X is 0,
    display_firstPlayer_message(PlayerName),
    get_Y_input(Y),
    nth0(Y, Board, Column),
        nth0(X, Column, Cell),
        append(Cell, Piece, NewCell),
        replaceAtListIndex(Column, X, NewCell, NewColumnValue),
        replaceAtListIndex(Board, Y, NewColumnValue, NewBoard).

place_first_red_player(Board, Size, PlayerName, NewBoard):-
    Piece = [red],
    X is Size-1,
    display_secoundPlayer_message(PlayerName),
    get_Y_input(Y),
    nth0(Y, Board, Column),
        nth0(X, Column, Cell),
        append(Cell, Piece, NewCell),
        replaceAtListIndex(Column, X, NewCell, NewColumnValue),
```

```
        replaceAtListIndex(Board, Y, NewColumnValue, NewBoard).

new_test(Board) :-
    Board = [
        [[],[red],[],[],[]],
        [[],[zb],[],[],[]],
        [[],[],[],[blue],[]],
        [[],[zr],[],[],[]],
        [[],[],[],[],[red]]
    ].
```

A.3 interface.pl

```
:- use_module(library(lists)).
:-[board].

displayTitleCard(_) :-
write('\n\t************************************'),
write('\n\t*                                  *'),
write('\n\t*            Virus_Wars_3          *'),
write('\n\t*                                  *'),
write('\n\t************************************').

displayInstructions(_) :-
        write('\n1 - Instructions'),nl,
        write('2 - Start playing'),nl,
        write('3 - Exit'),nl.

display_game_types(_) :-
    write('\n1 - Player vs Player'),nl,
        write('2 - Player vs NPC'),nl,
        write('3 - NPC vs NPC'),nl,
    write('4 - Exit'),nl.

printBoardExplanation(_) :-
        write('\nWrite Rules\n').

chooseOption(_) :-
        read(X),nl,
        (
                X == 1 -> printBoardExplanation(_),fail;
                X == 2 -> !;
                X == 3 -> halt(0);
                invalidInputMessageDisplay(_),fail
        ).

choose_game_type(X) :-
    read(X),nl,
    (
        X == 1 -> print_type(1),fail;
        X == 2 -> print_type(2),fail;
        X == 3 -> print_type(3),fail;
        X == 4 -> halt(0);
        invalidInputMessageDisplay(_),fail
    ).


continueScreen(_) :-
        write('Continue?\n1 - Yes\n2 - No'),nl,
        read(X),nl,
        (
                X == 1 -> start(_),fail;
                X == 2 -> halt(0);
```

```
                invalidInputMessageDisplay(_),fail
        ).

invalidInputMessageDisplay(_) :-
        write('Invalid Input'),nl.

print_type(X):-
    X = 1,
    write('You have choosen Player vs Player'),nl.

print_type(X):-
    X = 2,
    write('You have choosen Player vs NPC'),nl.

print_type(X):-
    X = 3,
    write('You have choosen NPC vs NPC'),nl.

choose_board_size(Size):-
    write('\nChoose board Size\nBoard Size must be between 5 and 10\n
    Input Value: '),
    read(Size),
    write('\n'),
    check_valid(Size),
    display_board_message(Size).

draw_board(Board,Size):-
    Y is 0,
    Alphabet=['NULL','A','B','C','D','E','F','G','H','I','J','K','L',
    'M','N','O'],
    draw_margin(_),
    draw_column_name(Size,1,Alphabet),
    draw_margin(_),
    draw_horizontal_padding(Size),
    draw_board_interior(Board, Y, Size).

draw_column_name(Size,X,Alphabet):-
    X =< Size,
    nth0(X,Alphabet,Output),
    write('  '),
    write(Output),
    write('  '),
    Y is X+1,
    draw_column_name(Size,Y,Alphabet).

draw_column_name(Size,Size,_):-
    write('\n').

draw_board_interior(Board, Y, Size):-
    Y < Size,
    Line is Y + 1,
    write(Line),
```

```
    write('   '),
    draw_vertical_padding(Board, 0, Y, Size),
    draw_margin(_),
    draw_horizontal_padding(Size),
    NewY is Y+1,
    draw_board_interior(Board, NewY, Size).

draw_board_interior(_, Size, Size).

draw_horizontal_padding(Size):-
    Size>0,
    write('-----'),
    X is Size-1,
    draw_horizontal_padding(X).

draw_horizontal_padding(0):-
    write('-\n').


draw_vertical_padding(Board, X, Y, Size):-
    X < Size,
    getCell(Board, X, Y, Cell),
    write('|'),
    draw_cell(Cell),
    %write('     '),
    NewX is X+1,
    draw_vertical_padding(Board, NewX, Y, Size).

draw_vertical_padding(_, Size, _, Size):-
    Size = Size,
    write('|\n').

draw_cell(Cell):-
    Cell == [blue],
    write('blue').

draw_cell(Cell):-
    Cell == [zb],
    write('zomb').

draw_cell(Cell):-
    Cell == [zr],
    write('zomr').

draw_cell(Cell):-
    Cell == [red,zb],
    write('zomb').

draw_cell(Cell):-
    Cell == [blue,zr],
    write('zomr').
```

```
draw_cell(Cell):-
    Cell == [red,blue],
    write('zomb').

draw_cell(Cell):-
    Cell == [blue,red],
    write('zomr').

draw_cell(Cell):-
    Cell == [],
    write('    ').

draw_cell(Cell):-
    Cell == [red],
    write('red ').

draw_cell(_):-
    write('    ').

draw_margin(_):-
    write('   ').

check_valid(Size):-
    Size>4,
    Size<11.

check_valid(Size):-
    Size<5,
    write('\nInvalid Input\n'),
    start(_).

check_valid(Size):-
    Size>10,
    write('\nInvalid Input\n'),
    start(_).

display_board_message(Size):-
    write('You have choosen a '),
    write(Size),
    write('x'),
    write(Size),
    write(' board\n').

write_player_name(PlayerName, X):-
    write(PlayerName),
    write(' play '),
    write(X),
    write(' of 5.'), nl.

ask_X_input(_):-
    write('X coordenate: [A...J]: ').
```

```
ask_Y_input(Size):-
    write('Y coordenate: [1...'),
    write(Size),
    write(']: ').

get_X_input(X):-
        read(Num),nl,write(Num),nl,
        ((
                Num == 'A' -> X is 0;
        Num == 'B' -> X is 1;
        Num == 'C' -> X is 2;
        Num == 'D' -> X is 3;
        Num == 'E' -> X is 4;
        Num == 'F' -> X is 5;
        Num == 'G' -> X is 6;
        Num == 'H' -> X is 7;
        Num == 'I' -> X is 8;
        Num == 'J' -> X is 9;
        Num == 'a' -> X is 0;
        Num == 'b' -> X is 1;
        Num == 'c' -> X is 2;
        Num == 'd' -> X is 3;
        Num == 'e' -> X is 4;
        Num == 'f' -> X is 5;
        Num == 'g' -> X is 6;
        Num == 'h' -> X is 7;
        Num == 'i' -> X is 8;
        Num == 'j' -> X is 9;
        invalidInputMessageDisplay(_),fail)
        ).

get_Y_input(Y):-
    read(Num),nl,
    Y is Num - 1.

display_firstPlayer_message(PlayerName):-
    write(PlayerName),
    write(' Choose Line to place the 1st Blue Piece: ').

display_secoundPlayer_message(PlayerName):-
    write(PlayerName),
    write(' Choose Line to place the 1st Red Piece: ').
```

A.4 cycle.pl

```
:-[helper_lib].
:-[interface].

loop_one(Board, Size, Type):-
    Type = 1,
    place_first_blue_player(Board, Size, 'Player1', NewBoard),
    draw_board(NewBoard,Size),
    place_first_red_player(NewBoard, Size, 'Player2', NewerBoard),
    draw_board(NewerBoard,Size),
    game_loop(NewerBoard, Size, Type).

game_loop(Board, Size, Type):-
    Type = 1,
    play_cycle_player([blue], Board, Size, 'Player1', NewBoard),
    play_cycle_player([red], NewBoard, Size, 'Player2', NewerBoard),
    game_loop(NewerBoard, Size, Type).

game_loop(Board, Size, Type):-
    Type = 2,
    play_cycle_player(red, Board, Size, 'Player', NewBoard),
    play_cycle_npc(blue, NewBoard, Size, 'NPC', NewerBoard),
    game_loop(NewerBoard, Size, Type).

game_loop(Board, Size, Type):-
    Type = 3,
    play_cycle_npc(red, Board, Size, 'NPC1', NewBoard),
    play_cycle_npc(blue, NewBoard, Size, 'NPC2', NewerBoard),
    game_loop(NewerBoard, Size, Type).

play_cycle_player(Piece, InBoard, Size, PlayerName, OutBoard) :-
    makePlay(Piece, InBoard, Size, PlayerName, OutB1, 1),
    draw_board(OutB1, Size),
    makePlay(Piece, OutB1, Size, PlayerName, OutB2, 2),
    draw_board(OutB2, Size),
    makePlay(Piece, OutB2, Size, PlayerName, OutB3, 3),
    draw_board(OutB3, Size),
    makePlay(Piece, OutB3, Size, PlayerName, OutB4, 4),
    draw_board(OutB4, Size),
    makePlay(Piece, OutB4, Size, PlayerName, OutBoard, 5),
    draw_board(OutBoard, Size).

makePlay(Piece, InBoard, Size, PlayerName, OutBoard, Num):-
    write_player_name(PlayerName, Num),
    ask_X_input(_),
    get_X_input(X),
    ask_Y_input(Size),
    get_Y_input(Y),
    place_piece(InBoard, X, Y, Piece, OutBoard, Size), write(OutBoard),nl.
```

A.5 helper*lib.pl*

```
%given a list, returns a new list, where at index the Element is placed
replaceAtListIndex(List, Index, Element, ResultingList) :-
        Index > 0,
        List = [ListHead | ListTail],
        ResultingList = [ListHead | ResListTail],
        IndexNext is Index - 1,
        replaceAtListIndex(ListTail, IndexNext, Element, ResListTail).

replaceAtListIndex(List, Index, Element, ResultingList) :-
        Index = 0,
        List = [ _ListHead | ListTail],
        ResultingList = [ Element | ListTail].

getAtomFromNumber(Number, Atom) :-
        number_chars(Number, Y), atom_chars(Atom, Y).

length_list(N, List) :- length(List, N).

generate_matrix(Cols, Rows, Matrix) :-
    length_list(Rows, Matrix),
    maplist(length_list(Cols), Matrix).
```