U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Virus Wars

PLOG – 2018

Ângelo Miguel Tenreiro Teixeira – 201606516

Henrique Melo Lima – 201606525

1. Introdução

O trabalho realizado consiste no desenvolvimento de um jogo de tabuleiro na linguagem Prolog.

O jogo em questão é intitulado de Virus Wars e foi criado algures nos anos 80, baseando-se na expansão de vários vírus. Além da possibilidade de o utilizador o jogar contra outro utilizador, o projeto conta também com um sistema de inteligência artificial (IA) que permite competir contra o computador ou executar jogos onde o computador controla ambos os jogadores. As entidades de IA criadas dispõem também de vários níveis de dificuldade.

2. O Jogo Virus Wars

Existem dois jogadores (controlados por humanos ou por IA). O jogador 0 é representado por um quadrado e o jogador 1 por um círculo. O objetivo é deixar o oponente sem jogadas possíveis. É jogado num tabuleiro 11x11 que inicialmente começa com um vírus do jogador 0 num canto (célula A0) e um vírus do jogador 1 no canto oposto (célula K10).

Começando pelo azul, os jogadores têm 5 jogadas por turno, sendo que cada jogada pode ser de 2 tipos:

- Colocar um vírus numa célula acessível (descrito abaixo) vazia no tabuleiro
- Absorver – *zombificar* - um vírus oponente em qualquer célula acessível do tabuleiro, p.ex. trocar um vírus adversário por um estado *zombificado* da cor do jogador, que fica permanente até ao fim do jogo, não podendo ser alterado ou removido do tabuleiro

Uma célula é acessível se:

- Está verticalmente, horizontalmente ou diagonalmente adjacente a um vírus do jogador já presente no tabuleiro, mesmo tendo sido colocado num mesmo turno
- Está ligada a um conjunto de *zombies* do jogador verticalmente, horizontalmente ou diagonalmente adjacentes entre si e em que pelo menos um *zombie* do conjunto está adjacente a um vírus do jogador.

O jogo acaba quando um jogador fica sem jogadas possíveis, ganhando o adversário.

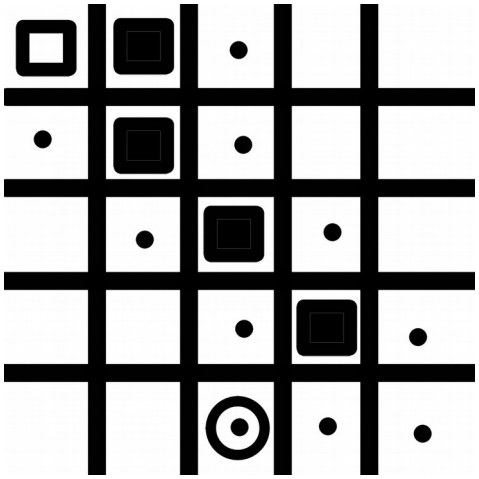

Figura 1 – Exemplo 1 - O jogador 0, representado por quadrados, pode jogar nas células marcadas por um ponto (.), inclusive zombificar o adversário, uma vez que apresenta uma cadeia com pelo menos um vírus ativo adjacente a este.

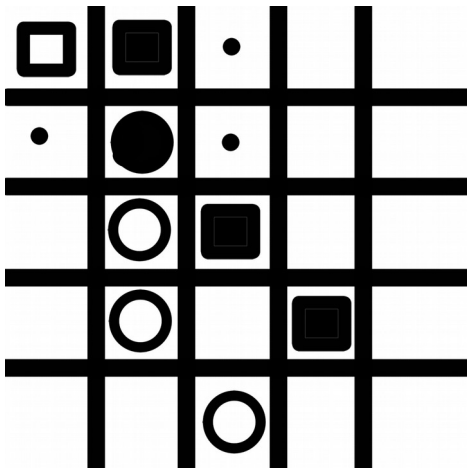

Figura 2 – Exemplo 2 - O jogador 0, representado por quadrados, pode jogar nas células marcadas por um ponto (.), já não pode jogar em todas as células adjacentes a zombies seus, uma vez que nem todos estão adjacentes a um vírus ativo ou a uma cadeia adjacente a este.

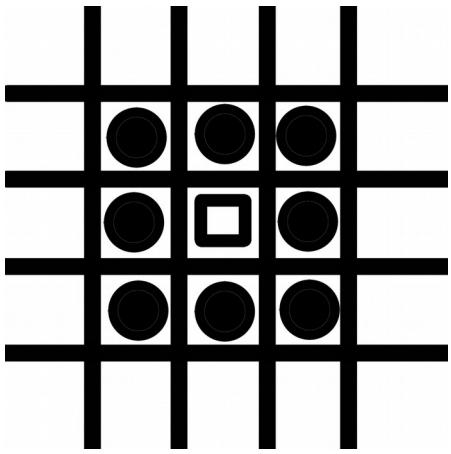

Figura 3 – Exemplo 3 - O jogador 0, representado por quadrados, não pode jogar em nenhuma célula, pois está rodeado de zombies adversários, não tendo mais vírus ativos para expandir.

3. Lógica do Jogo

3.1 Representação do Estado do Jogo

Para representar o estado do jogo, é usado um numero inteiro, 0 ou 1, que representa o jogador atual, para o tabuleiro é utilizada uma lista de listas de átomos (valores inteiros), sendo que o valor de cada átomo segue a seguinte estrutura:

1: Representa uma célula com um vírus do jogador 0 no estado normal/ativo

2: Representa uma célula com um vírus do jogador 1 no estado normal/ativo

3: Representa uma célula com um *zombie do jogador 0*

4: Representa uma célula com um *zombie do jogador 1*

0: Representa uma célula vazia

Nota: Os valores das personagens do jogador 0 são ímpares e os das personagens do jogador 1 são pares. Esta decisão foi tomada de forma a futuramente facilitar a identificação das personagens de cada jogador.

```
start_gameplay(L):-
    L = [
        [1,0,0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0,0,2]
    ].

mid_gameplay(L):-
    L = [
        [1,1,3,0,0,0,0,0,0,0,0],
        [1,3,0,0,0,0,0,0,0,0,0],
        [0,0,3,3,0,0,0,0,0,0,0],
        [0,0,0,0,2,0,0,0,0,0,0],
        [0,0,0,0,2,2,0,0,0,0,0],
        [0,0,0,3,3,2,0,0,0,0,0],
        [0,2,2,0,0,0,2,2,2,2,2],
        [0,0,0,0,0,0,0,0,0,0,2],
        [0,0,0,0,0,0,0,0,0,0,2],
        [0,0,0,0,0,0,0,0,0,0,2],
        [0,0,0,0,0,0,0,0,0,0,2]
    ].

final_gameplay(L):-
    L = [
        [1,1,3,0,0,0,0,0,0,0,0],
        [1,3,0,0,0,0,0,0,0,0,0],
        [0,0,3,3,0,3,0,0,0,0,0],
        [0,0,1,0,3,3,3,0,0,0,0],
        [0,0,0,0,3,3,3,0,0,0,0],
        [0,0,1,3,3,3,1,1,0,0,0],
        [0,3,3,1,0,0,3,3,3,3,3],
        [0,0,0,0,0,0,1,1,0,3,3],
        [0,0,0,0,0,0,0,0,0,3,3],
        [0,0,0,0,0,0,0,0,0,1,3],
        [0,0,0,0,0,0,0,0,0,0,3]
    ].

```

Figura 4: Representação interna do tabuleiro em 3 momentos do jogo

3.2 – Visualização do Tabuleiro

O tabuleiro é desenhado a partir de uma lista de listas, através do predicado *display_game/2*. Este predicado, por sua vez, utiliza predicados auxiliares como:

- **traducao**: garante a tradução de certos valores (de forma a obter, por exemplo, o código unicode do caracter correspondente a um vírus)
- **print_cell**: imprime para a consola o caracter unicode que corresponde à tradução de um elemento e separa-o com o caracter "|".
- **display_line**: imprime para a consola, recursivamente, o caracter unicode que corresponde à tradução de cada elemento de uma lista.
- **display_seperated_line**: à semelhança de display line, imprime para a consola, recursivamente, cada elemento de uma lista. No entanto, utiliza o predicado print_cell para cada elemento, resultado numa separação visual.
- **gen_line:** através de recursão, cria uma lista com um número de elementos especificado na qual todos os elementos têm o valor fornecido.
- **gen_column_labels:** através de recursão, cria uma lista com um número de elementos especificado na qual cada elemento é numerado e incrementado em 1 face ao elemento anterior. O valor do elemento inicial é especificado. Este predicado é utilizado para gerar os códigos unicode dos titulos das colunas do tabuleiro.
- **display_matrix:** utilizando os predicados acima referidos recursivamente, desenha as linhas do tabuleiro, numerando-as.

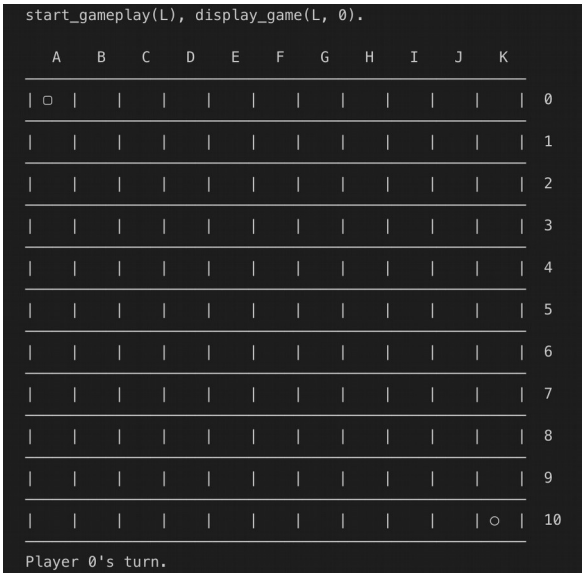

Figura 5: Visualização do tabuleiro no final de um jogo

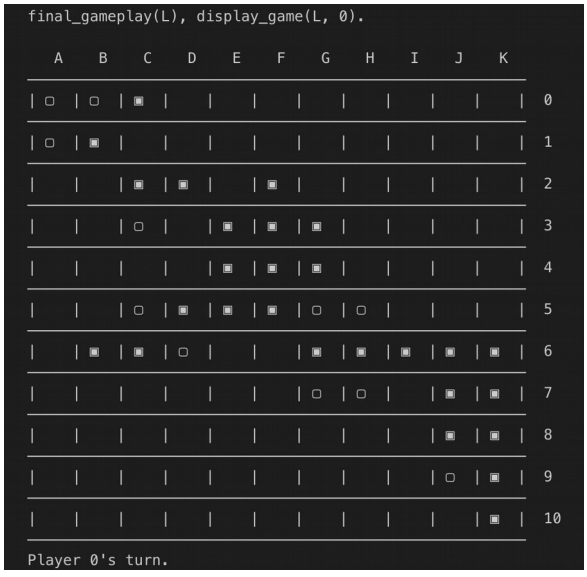

Figura 6: Visualização do tabuleiro inicial

3.3. Lista de Jogadas Válidas

Para fazer a verificação de uma jogada, é necessário validá-la de acordo com as regras do jogo, descritas acima. Isto é feito com recurso ao predicado valid_move/3, sendo que este verifica se está vazia, isto é, não tem nenhum vírus lá colocado, ou se tem um vírus ativo adversário, e verifica se há uma “cadeia ativa” adjacente, ou seja, uma cadeia de vírus do jogador que contenha pelo menos um vírus ativo do mesmo.

Para fazer a verificação de cadeias adjacentes, utilizamos um predicado recursivo, has_active_chain/4 que, para cada celula adjacente, verifica se tem uma adjacente que seja um vírus ativo. De forma a tornar esta verificação mais eficiente, percorremos o tabluleiro como se fosse um grafo, em que cada célula é um nó que está ligado por arestas bi-direcionais a todas as células adjacentes. Usando uma abordagem tipo DFS (*Depth First Search*) procuramos a partir da célula a colocar o vírus um vírus ativo, desde que ligado por elementos do jogador, assegurando que existe uma cadeia com pelo menos um vírus ativo. Para garantir que o algoritmo não corre infinitamente, usamos asserta/1 para marcar células visitadas e saber se já foram analisadas de uma forma mais rápida.

Para obter a lista de jogadas válidas, é usado o predicado valid_moves/3 que utiliza setof(X-Y, valid_move(Board, Player, X-Y), ListOfMoves), sendo ListOfMoves instanciada com uma lista de jogadas do tipo X-Y válidas.

3.4 Execução de Jogadas

Após escolha da próxima jogada, é utilizado o predicado *move\3* de forma a validar e aplicar a jogada. Com recurso ao tabuleiro, ao identificador do jogador e à posição desejada, utiliza-se o predicado *valid_move/3* referido no capítulo anterior. Caso a jogada não seja válida (Apenas acontece quando é um humano a jogar, uma vez que a IA apenas faz jogadas válidas), é mostrada uma mensagem de erro ao utilizador, pedindo novamente que insira uma jogada valida. Assim que *move\3* possuir uma posição que resulte numa jogada válida, é então utilizado o predicado *play/3* de forma a gerar o novo tabuleiro. Para gerar o tabuleiro corretamente, é necessário primeiro perceber qual o valor que deverá ser inserido na célula selecionada. Assim, desenvolvemos o predicado *get_player_new_elem/5* que nos permite instanciar numa variável o valor mencionado. Após isto, é criado um novo tabuleiro com as novas alterações através do predicado *update_matrix_at/5*. Com o *update_matrix_at/5* são percorridas recursivamente as colunas até à linha pretendida, percorrendo também os seus elementos até encontrar o elemento nas coordenadas desejadas. Nesse momento, é reconstruido o tabuleiro refletindo a jogada.

3.5. Final do Jogo

De forma a garantir uma identificação do estado final do jogo e terminação do mesmo, foi criado um predicado *game_over/2* que establece uma ligação entre um tabuleiro e o jogador vencedor segundo o estado desse tabuleiro.

No caso de não existir um vencedor, o predicado falha, permitindo ao ciclo principal do jogo proseguir. No entanto, caso o predicado se verifique, o ciclo termina e é apresentado na consola o vencedor. *game_over/2* apresenta os resultados esperados utilizando a negação do predicado *valid_move/3 (valid_move(Board, Player, _))* descrito em capítulos anteriores, pois o jogo termina quando não existem jogadas possíveis para um dos participantes. Caso exista uma posição válida, *valid_move/3* será executado com sucesso, demonstrando que existe pelo menos uma jogada possível e que o jogo ainda não terminou. Caso contrário, se não existir uma jogada possível, *valid_move/3* falha. Assim, é apenas necessário negar o resultado de *valid_move/3* para saber se o jogo acabou.

Em iterações anteriores foram utilizados métodos com *findall/3* ou *setof/3* a partir de *valid_move/3* , que produzem uma lista de resultados possíveis. No entanto, mostrou-se mais eficiente a utilização do predicado *valid_move/3* apenas, pois assim que é encontrada uma jogada válida conclui-se *game_over* sem haver a necessidade de produzir uma lista com todas as jogadas possíveis.

3.6. Avaliação do Tabuleiro

Para avaliar o tabuleiro, é usada uma fórmula que agrega cadeias de zombies e proximidade ao inimigo, consistindo numa soma ponderada em que a primeira tem um peso superior. Desta forma, a IA tende a criar cadeias de zombies, e, enquanto não pode, vai-se aproximando do adversário para o poder fazer. Contudo, ao longo do desenvolvimento, apercebemo-nos que em casos de jogadas que terminariam o jogo, o jogador que poderia ganhar não estava a realizar a jogada vencedora, preferindo aumentar uma das suas cadeias de zombies. Assim, acrescentamos uma componente à fórmula com maior peso que dá valor caso o oponente fique sem vírus ativos.

Sendo WC (Win condition) o valor dado caso o oponente fique sem vírus ativos, ZVal o valor de cadeias de zombies e ProxVal o valor de proximidade a um inimigo, a formula de avaliação de um tabuleiro é:

$$\mathbf{Value = WC^3 + ZVal^2 + ProxVal}$$

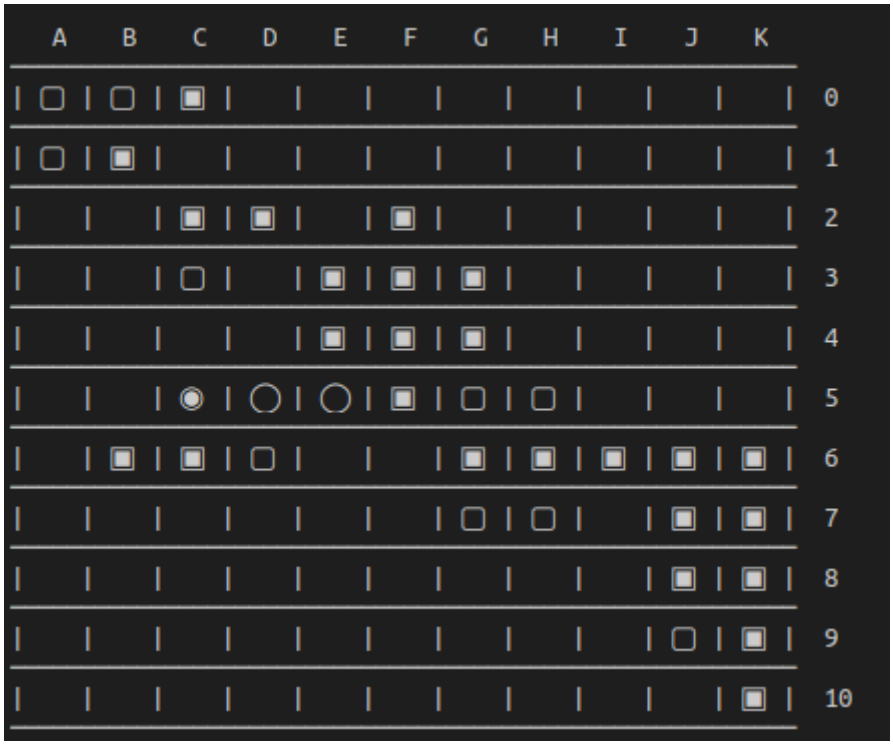

Figura 7: Exemplo de avaliação do tabuleiro num momento do jogo

No caso da figura 7, o valor do tabuleiro para o jogador 0 é 3612, sendo 51 para o jogador 1, devido ao facto do primeiro ter mais cadeias de zombies.

Para avaliar um tabuleiro de acordo com a fórmula descrita, é usado o predicado **value(+Board, +Player, -Value)** que coloca em Value o resultado da fórmula.

3.7 Jogada do Computador

Em termos de jogadas do computador, programámos 3 tipos diferentes de IA: Iniciante, Intermédio e Difícil. Todas elas determinam a jogada a fazer, com base nas jogadas válidas para o tabuleiro atual.

3.7.1 Modo Iniciante

A IA Iniciante é a mais simples de todas, para um tabuleiro, usa *valid_moves/3*, obtendo uma lista de jogadas válidas para aquele estado de jogo e escolhe uma jogada aleatória, sendo apenas automática e não "inteligente" como as duas Inteligências Artificiais, especificadas abaixo.

3.7.2 Modo Intermédio

A IA Intermédia já apresenta alguma "inteligência" na escolha das jogadas a realizar, face à IA descrita anteriormente, uma vez que aproveita o predicado descrito no tópico anterior, ***value/3*** para dar valor às jogadas possíveis, e desta forma, escolher a jogada que o leva a um estado mais vantajoso. Para implementar esta funcionalidade, criámos o predicado *valid_moves_valued/3*, idêntico ao *valid_moves/3*, embora os elementos da lista de jogadas sejam estruturas Val-X-Y, podendo, assim, saber-se o valor de uma jogada. Para escolher a melhor jogada, iteramos sobre a lista de jogadas, encontrando o valor máximo, e de seguida iteramos novamente, unificando uma lista com as jogadas de valor máximo, escolhendo aleatóriamente uma jogada de maior valor, para impedir que a IA seja determinística. Este algorítmo de obtenção de melhores jogadas executa em O(n), sendo mais eficiente do que ordenar uma lista O(n log(n)) e escolher de entre as primeiro(as) jogada(as), que têm o valor máximo.

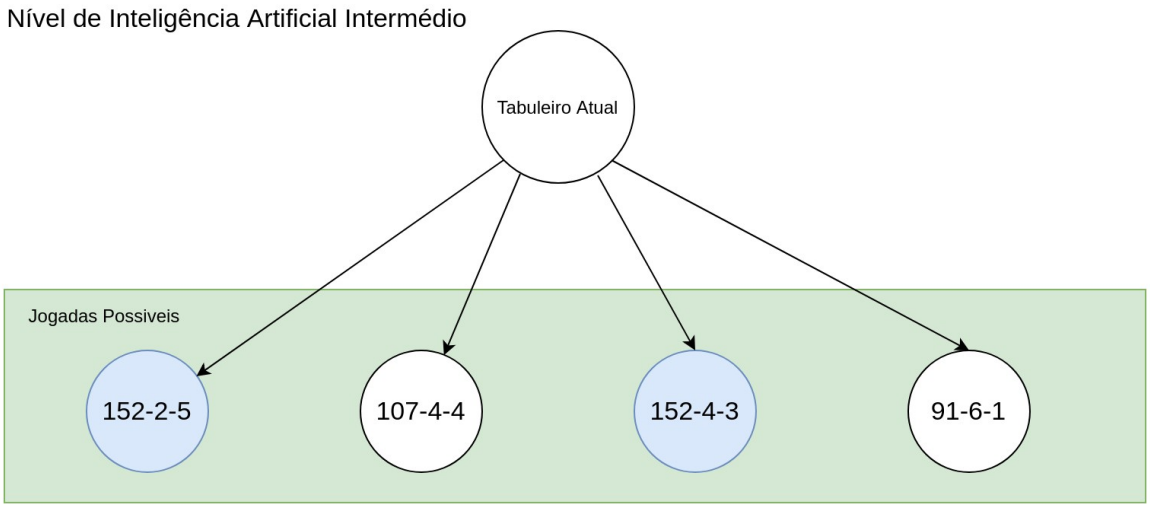

Figura 8: Árvore de jogadas possíveis a partir de um estado do tabuleiro segundo análise no modo de dificuldade intermédia

Como se pode ver na figura 8, para um dado tabuleiro inicial, são calculadas as jogadas válidas do jogador, juntamente com o respetivo valor, representando o valor com que o tabuleiro fica, caso a jogada seja efetuada. Escolhe-se a que oferece melhor valor e, caso haja repetidos, escolhe-se aleatóriamente uma jogada.

3.7.3 Modo Difícil

A IA Difícil realiza um passo adicional em relação à IA intermédia pois, para além de obter as jogadas válidas, vai determinar qual delas origina a pior jogada do adversário, tentando, desta forma, prever o adversário e jogando em conformidade. Para tal, para cada jogada válida, encontra as jogadas válidas do inimigo, e escolhe a jogada que tem como contra-jogada o valor mínimo de entre os valores máximos de contra-jogadas.

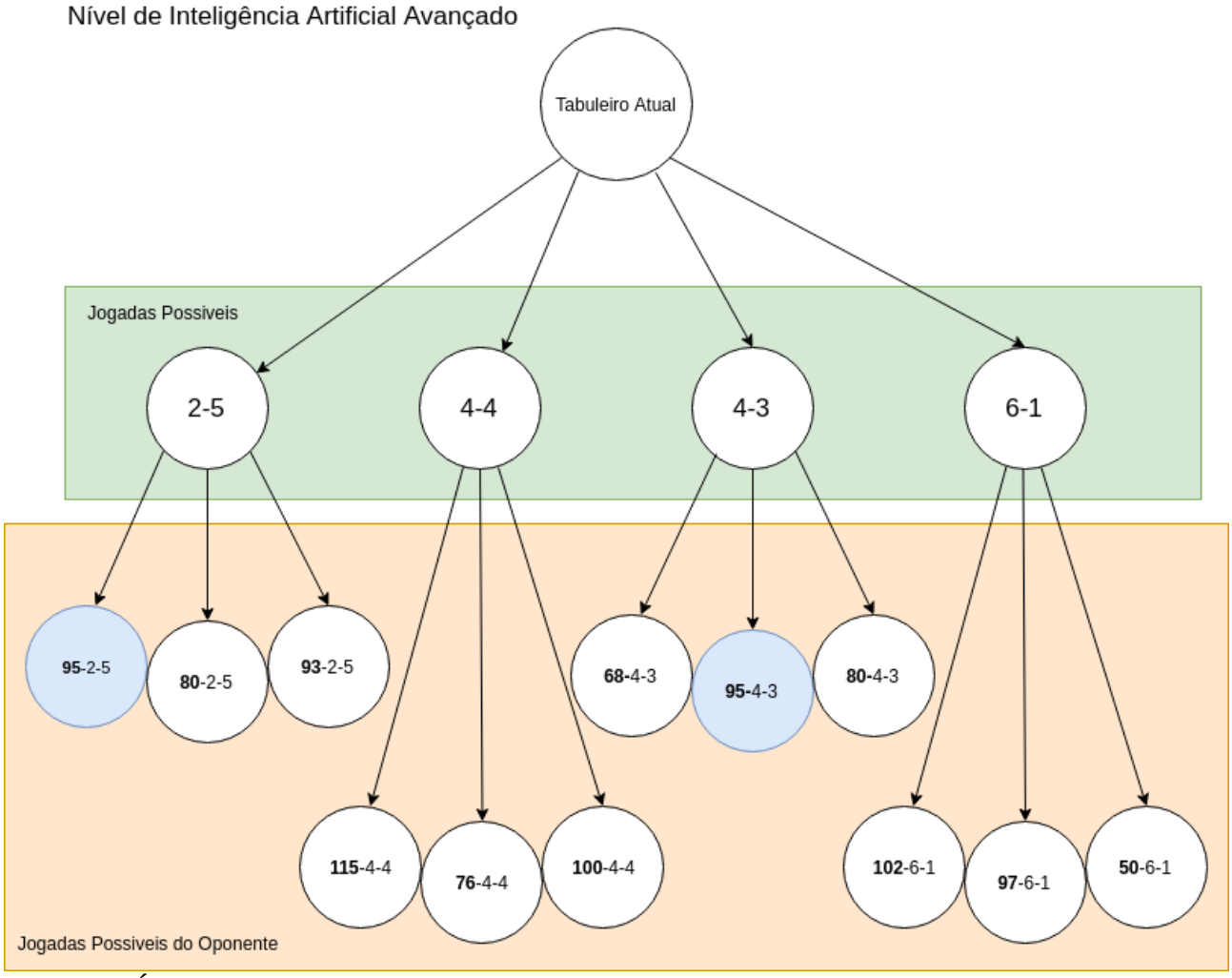

Figura 9: Árvore de jogadas possíveis a partir de um estado do tabuleiro segundo análise no modo de dificuldade difícil

Como se pode observar na figura 9, para um conjunto de jogadas possíveis, são determinadas as respostas válidas do adversário (no formato Valor-X-Y em que X e Y representam a jogada que origina a resposta em causa). De seguida, para cada jogada inicial, verificam-se as jogadas do adversário com maior valor para este. De entre os valores máximos calculados para o adversário, escolhe-se a jogada inicial que origina o mínimo desses valores máximos. No caso de haver duas ou mais jogadas que originem respostas ótimas com o mesmo valor (2-5 e 4-3, neste caso), escolhe-se aleatóriamente uma delas.

4. Conclusões

Com este projeto, concluimos que a linguagem PROLOG oferece bastantes vantagens no que toca a verificação de regras/predicados o que leva a uma diminuição do código necessário para a execução das mesmas tarefas, em comparação com linguagem de programação procedimentais como C++. Contudo sentimos um grande impacto, relativamente à diferença de paradigma de programação face a outros mais usados por nós no curso anteriormente.

Achámos também que o desenvolvimento foi difícil pois, no que toca a testes e *debug, foi* menos intuitivo, o que tornou mais lenta e frustrante a resolução de alguns problemas que fomos encontrando.

Contudo, nem tudo foi difícil, e apreciámos o facto de ser tão fácil programar uma IA em PROLOG, bastando definir pouco mais que as regras do jogo, isto graças ao motor de PROLOG.

No que toca ao trabalho, uma melhoria a fazer seria o desenvolvimento de uma IA com **n** níveis de previsão, de forma a ser mais difícil vencê-la, quanto maior fosse o numero de níveis.

Bibliografia

Virus Wars Wikipedia Page
http://www.iggamecenter.com/info/en/viruswars.html

Sterling, Leon; The Art of Prolog.