Resolução do jogo Block It utilizando o método de pesquisa MiniMax alfa-beta e variantes em linguagem Java (Tema A3 Grupo 5)

1st Eduardo Silva (up201603135)
MIEIC
FEUP
Porto, Portugal
up201603135@fe.up.pt

2nd Mariana Costa (up201604414)
MIEIC
FEUP
Porto, Portugal
up201604414@fe.up.pt

3rd Tiago Castro (up201606186)
MIEIC
FEUP
Porto, Portugal
up201606186@fe.up.pt

***Resumo*—O presente artigo tem como principal objetivo delinear os contornos de desenvolvimento da resolução do jogo Block It utilizando o método de pesquisa MiniMax alfa-beta em linguagem Java. Será realizada uma descrição sucinta do jogo em questão, seguida da formulação do mesmo como problema de pesquisa bem como a exposição de outros trabalhos semelhantes considerados úteis e uma breve conclusão a destacar as perspetivas de desenvolvimento.**

***Index Terms*—Inteligência Artificial, Jogo, Bloqueio, MiniMax, Pesquisa com adversários**

I. Introdução

O trabalho aqui exposto insere-se no âmbito da unidade curricular de Inteligência Artificial do MIEIC. É proposta a resolução do Block It, um jogo competitivo, utilizando métodos de pesquisa MiniMax alfa-beta e variantes em linguagem Java. Este tem a seu dispor um modo de visualização textual, de forma a mostrar com clareza a evolução do tabuleiro e possibilitar a comunicação entre a máquina e o utilizador/jogador. Dispõe também um modo de jogo no qual o computador conduz o jogo autonomamente, com recurso a um método e configuração selecionados previamente pelo utilizador.

II. Descrição do problema

Block It [1] é um jogo competitivo no qual o objetivo principal consiste em percorrer o tabuleiro de jogo, alcançando o lado cuja cor corresponde à cor do jogador em questão (oposto à posição inicial), sendo o vencedor aquele que cumprir este objetivo em primeiro lugar. Em cada turno, cada jogador pode optar por avançar uma casa ou colocar uma barreira. A atual implementação permite a existência de dois jogadores.

III. Formulação do problema

A. Representação do estado

Estados representados por uma matriz 2D de carateres constituída por quatro elementos principais:

- Posição na qual pode ser colocada uma barreira
- Posição para a qual o jogador se pode deslocar
- Barreira
- Jogador

De notar que cada barreira ocupa três posições seguidas na matriz, vertical ou horizontalmente. As posições de fronteira para as quais os jogadores se podem deslocar (ou seja, os limites do tabuleiro) são demarcadas das demais através da indicação da sua cor, de forma a que cada jogador saiba para que lado se deslocar. Cada célula contém dois digitos.

Na figura seguinte encontra-se a representação textual do estado de jogo, cujos elementos se encontram enumerados e descritos de seguida:

- **'RR' / 'BB' / 'GG' / 'YY'**: Jogador vermelho / azul / verde / amarelo
- **'rd '/ 'bl' / 'gr' / 'yl'**: Fronteira vermelho / azul / verde / amarelo
- **'XX'**: Barreira
- **'__'**: Posição para a qual o jogador se pode deslocar
- **Espaço vazio**: Posição na qual pode ser colocada uma barreira

```
    |00|01|02|03|04|05|06|07|08|09|10|11|12|13|14|15|16|
    |bl|bl|bl|bl|bl|bl|bl|bl|bl|bl|bl|bl|bl|bl|bl|bl|bl|
00|gr|__|  |__|  |__|  |__|  |RR|  |__|  |__|  |__|  |__|yl|00
01|gr|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |yl|01
02|gr|__|  |__|  |__|  |__|  |__|  |__|  |__|  |__|  |__|yl|02
03|gr|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |yl|03
04|gr|__|  |__|  |__|  |__|  |__|  |__|  |__|  |__|  |__|yl|04
05|gr|  |  |  |XX|XX|XX|  |  |  |  |  |  |  |  |  |  |  |yl|05
06|gr|__|  |__|  |__|  |__|XX|__|  |__|  |__|  |__|  |__|yl|06
07|gr|  |  |  |  |  |  |XX|  |  |  |  |  |  |  |  |  |  |yl|07
08|gr|__|  |__|  |__|  |__|XX|__|  |__|  |__|  |__|  |GG|yl|08
09|gr|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |yl|09
10|gr|__|  |__|  |__|  |__|  |__|  |__|  |__|  |__|  |__|yl|10
11|gr|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |yl|11
12|gr|__|  |__|  |__|  |__|  |__|  |__|  |__|  |__|  |__|yl|12
13|gr|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |yl|13
14|gr|__|  |__|  |__|  |__|  |__|  |__|  |__|  |__|  |__|yl|14
15|gr|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |yl|15
16|gr|__|  |__|  |__|  |__|  |__|  |__|  |__|  |__|  |__|yl|16
    |rd|rd|rd|rd|rd|rd|rd|rd|rd|rd|rd|rd|rd|rd|rd|rd|rd|
    |00|01|02|03|04|05|06|07|08|09|10|11|12|13|14|15|16|
```

Figura 1. Representação textual de um tabuleiro de jogo

B. Estado inicial

Jogadores dispostos em lados opostos à fronteira da sua cor.

C. Teste terminal

Para que um jogador seja considerado vencedor, é necessário estar numa posição adjacente à fronteira da mesma cor que a sua.

D. Operadores

- *Mover jogador (esquerda, direita, cima, baixo)*
  - **Pré-condições**: A posição de destino deverá estar demarcada pelo carater '_', dentro dos limites do tabuleiro e a uma posição de distância (2 celulas) da posição atual.
  - **Efeitos**: O jogador move-se uma posição na direção indicada. Se se encontrar adjacente á sua barreira ganha.
- *Colocar barreira*
  - **Pré-condições**: A posição de destino deverá permitir a criação de uma barreira que ocupe 3 posições (horizontal ou verticalmente) sendo que a primeira e a última posição da barreira deverão estar adjacentes a uma celula '_', também elas igualmente válidas.
  - **Efeitos**: É criada uma barreira ocupando 3 espaços previamente vazios na posição e direção indicadas.

E. Função utilidade

Para calcular o valor dos leaf nodes são usadas funções heuristicas que retornem um valor que representa o valor do board para o jogador. Foram elaboradas duas heuristicas que calculam:

- comparação das distâncias diretas à parede correspondente a cada um dos jogadores (**competitive heuristic**).
- comparação do comprimento do caminho à parede calculado com o auxílio do algoritmo A* correspondente a cada um dos jogadores (**path heuristic**).

IV. Trabalho relacionado

Como o jogo realizado é bastante específico, a pesquisa centrou-se mais no algoritmo MiniMax em si. Também se focou especialmente em fontes que tivessem Java como linguagem principal de modo a facilitar a possível incorporação de código no nosso trabalho. Como tal, foi encontrado um repositório [2] com uma implementação do algoritmo MiniMax com cortes Alpha e Beta do jogo do galo (note-se que há bastantes implementações deste jogo com MiniMax em vários repositórios no entanto esta pareceu-nos uma das melhores). Além deste, foram também encontrados diversos websites sobre o mesmo tópico que considerámos relevantes [3], [4].

V. Implementação do jogo

Segue-se uma descrição dos principais pormenores da implementação do projeto na linguagem escolhida (Java), dividida nas secções consideradas mais relevantes para uma compreensão robusta do trabalho realizado. De notar que apenas os métodos avaliados como estando a carecer de aprofundamento ou cujo propósito não é de compreensão imediata serão explicados em detalhe.

A. Estrutura de ficheiros

O programa encontra-se repartido por três *packages*, sendo que 1 se encontra nested noutro e cada um com responsabilidades distintas:

- ***game***: Contém toda a lógica de jogo, bem como as classes responsáveis pela resolução do problema com recurso aos diferentes métodos de pesquisa;
- ***node***: Contém as classes dos vários nodes usados;
- ***testing***: Contém as classes utilizadas para o teste autónomo da resolução dos níveis do jogo e a avaliação do desempenho dos algoritmos implementados;

B. Representação do estado do tabuleiro

Um tabuleiro de jogo é representado pela classe *GameBoard*, que contém a informação necessária à avaliação do mesmo:

- ***char[][] board***: Matriz de *chars*, na qual cada celula pode estar vazia com '_' indicando uma posição livre para movimento de jogadores ou com ' ' simbolizando uma posição livre para colocar uma barreira. As celulas podem também representar um jogador ou uma barreira;
- ***int[][] players***: Atributo que contém as posições dos jogadores do board, sendo a posição 0 1 2 e 3 do array relativa as coodernadas x e y de, respetivamente, o jogador vermelho, verde, azul, amarelo;
- ***boolean showMove***: Valor utilizado para determinar se, durante a movimentação de um jogador no tabuleiro, este deve ser impresso na consola ou não;

É possivel defenir no codigo o número de barreiras e o tamanho do board.

C. Operadores

Os operadores de jogo presentes na classe *GameBoard* procuram traduzir todos os movimentos aceites pela lógica de jogo. Tendo em conta que os movimentos válidos consistem unicamente na translação de jogadores no tabuleiro, todos eles seguem uma estrutura semelhante, variando apenas na direção do movimento e consequentemente das posições analisadas para a validação do mesmo.

Assim, os principais métodos utilizados para a concretização de uma jogada possuem a designação de *move*[Sentido] ou *place*[Direção]*Barrier*, aceitando como argumento um *int[]* contendo as coordenadas do jogador a movimentar-se ou dois *int* contendo as coordenadas da celula central da barreira e retornando um objeto do tipo *GameBoard* cujo atributo *board* possui o tabuleiro de jogo atualizado.

Para a validação das jogadas é utilizado o método *validateMove*[Sentido], que determina se, após a jogada, o jogador se encontra dentro dos limites do tabuleiro (ou seja, se as suas coordenadas se traduzem em índices válidos da matriz) e se o sentido para o qual o jogador se pretende dirigir não tem adjacente uma barreira. Aceita como argumento um *int[]* contendo as coordenadas do jogador a movimentar-se e retorna

um booleano que reflete o resultado da análise realizada (*true* se a deslocação for válida, *false* caso contrário).

D. Listagem de jogadas possíveis e análise da viabilidade de um tabuleiro

A listagem de todas as jogadas possíveis num dado tabuleiro é realizada pelo método *expandNode* da classe *GameNode*, que retorna uma *ArrayList* de todos os objetos do tipo *GameNode* descendentes do objeto *GameNode* a ser analisado.

E. Heurísticas

As heuristicas foram implementadas a nível da classe *PlayerNode*, sendo que foi criada a função **calculateHeuristic** para calcular o valor do board em questão mediante a heuristica escolhida previamente pelo utilizador. O utilizador tem a seu dispor duas heuristicas que serão especificadas aseguir:

- **competitiveHeuristic**: Esta heuristica serve-se da distância direta dos jogadores à parede objetivo. O resultado é calculado com a formula seguinte : "**value = (boardSize - mine) - (boardSize - his)**". A variavel **mine** refere-se à distância do jogador em questão e a variavel **his** refere-se à distãncia do jogador adversário. De forma a que sejam priorizados resultados em que o jogador se aproxima mais da sua parede, esses resultados devem possuir um valor superior, de forma que, é subtraido ao tamanho total do board a distância para que, quanto menos for a distância, maior o valor da heuristica. Esta heuristica pode resultar no bot ficar preso quando este fica encurralado por barreiras e o caminho de saida se encontra a um numero de moves superior aos previstos pelo minimax, ou seja, ao aumentar a depth possível, esta heuristica funciona perfeitamente.
- **pathHeuristics**: Esta heuristica serve-se do algoritmo A* para calcular o caminho mais curto que o jogador vai ter que tomar para chegar à sua parede e ganhar o jogo. O A* utiliza a distância direta à parede como heuristica auxiliar . É corrido o algoritmo para o jogador e para o seu oponente para calcular as suas distâncias sendo que, no fim, o valor final da heuristica é calculado da mesma forma que na heuristica anterior, sendo que **mine** é o comprimento do path que o jogador tem que percorrer e **his** é o comprimento do path que o oponente tem que percorrer.

Com o intuito de utilizar a *PriorityQueue* com elementos do tipo *PlayerNode* para o A*, foi concretizado o *overload* ao método *compareTo* para que objetos deste tipo pudessem ser inseridos nesta estrutura de dados.

F. Teste objetivo

A determinação da condição de vitória é realizada pelo método *isWinner* da classe *PlayerNode*; este verifica se o jogador em questão (contém argumento *playerColor* do tipo char representante da cor do jogador a avaliar) está numa posição adjacente à parede objetivo.

G. Obtenção da solução

O método play play chama o método minimax que, tal como o nome indica, excecuta o algoritmo Mini Max em relação aos atributos e tabuleiros atuais. Este irá por sua vez correr um método auxiliar recursivo que irá construir a árvore de nós do algoritmo e atualizar os valores destes. Após isto, é escolhido o nó que possui maior valor, e é retornado o operador deste em formato String. Este operador é consequentemente executado no nó inicial, dando assim continuação ao jogo.

H. Visualização do estado de jogo

Para realizar a interpretação visual do resultado obtido, foi implementado o método *printBoard* na classe *GameBoard*, responsável pela impressão do tabuleiro de jogo na consola.

I. Representação do menu

A classe *BlockIt* possui os métodos necessários à interação entre o utilizador e a máquina, possibilitando a escolha da cor do jogador e o seu modo, sendo os quais:

- **Human**: jogador humano;
- **Normal**: bot que utiliza a competitiveHeuristic para avaliação dos boards;
- **Hard**: bot que utiliza a pathHeuristic para avaliação dos boards.

J. Funções de teste e análise de resultados

Como referido anteriormente, o package *test* encontra-se responsável pelas funções utilizadas no estudo dos resultados obtidos na execução do algoritmo. Para isto, é utilizada a classe *Suite*.

VI. Experiências e resultados

Na imagem seguinte está presente uma tabela contendo os tempos (ms) que um bot com dificuldade X demora a encontrar a melhor solução com recurso a MiniMax contra um bot de dificuldade Y dispondo duma depth da arvore Z. As colunas representam a dificuldade do bot que se movimenta e do oponente na forma X/Y e as linhas representam a depth da arvore Z.

	2/2	2/3	3/2	3/3
3	7	1	750	291
5	45	12	3114	2519
7	308	269	31165	31467

Figura 2. tempos minimax

VII. Conclusões e perspetivas de desenvolvimento

Após conclusão do trabalho, reforçamos o nosso conhecimento relativo ao MiniMax mais especificamente à variante de alpha & beta prunning utilizada no desenvolvimento do programa. O projeto foi realizado com sucesso sendo que, com a implementação de duas heuristicas sendo uma delas especialmente complexa usando o auxilio do algoritmo A* utilizado e estudado no projeto anterior dá uma complexidade extra à arquitetura do projeto e demonstra conhecimentos previamente utilizados.

As dificuldades mais relevantes encontradas passaram por uma preparação e ponderação prévia do trabalho não tão bem conseguida que mais tarde acabou por limitar a nossa capacidade de avançar com o desenvolvimento do projeto. Eventualmente, após análise do codigo, foram encontradas as limitações e eliminadas sendo a elaboração do programa pode então voltar ao normal.

Consideramos o resultado final do trabalho de acordo com os nossos objetivos definidos antes do inicio do desenvolvimento do mesmo, sendo que, não conseguimos pensar em muitas outras formas de melhorar o trabalho para além de permitir mais do que dois jogadores no board ao mesmo tempo, ideia esta que foi conversada com o professor encarregue da nossa turma e afastada devido à complexidade desnecessária adicionada ao trabalho e ao exagerado overhead fruto duma jogada do bot com o auxilio do MiniMax tendo em conta mais que 2 jogadores.

REFERÊNCIAS

[1] https://play.google.com/store/apps/details?id=br.com.nespolo.activity
[2] https://github.com/LazoCoder/Tic-Tac-Toe
[3] https://www.e4developer.com/2018/09/23/implementing-minimax-algorithm-in-java/
[4] https://tutorialedge.net/artificial-intelligence/min-max-algorithm-in-java/