U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Jogos com Oponentes

Relatório 2

Rui Santos, Tiago Teixeira e Carla Henriques

CC2006: Inteligência Artificial

Profª. Inês Dutra

Prof.º Francesco Renna

Prof.º Nuno Guimarães

Abril de 2023

Índice

Introdução

Jogos com adversários são jogos envolvendo dois jogadores com objetivos opostos. O que um jogador quer alcançar, o outro tenta evitar. Jogos como xadrez, damas e connect-four são exemplos de jogos competitivos populares. A solução desses jogos pode ser vista como um problema de busca onde o espaço de estados representa todas as configurações possíveis do jogo. Dada a quantidade de possibilidades que existem em um jogo, encontrar a melhor jogada é uma tarefa difícil. Para resolver esses jogos, algoritmos de localização de oponentes foram desenvolvidos para encontrar a melhor jogada possível para cada jogador. Alguns exemplos desses algoritmos são: Minimax, Alpha-Beta e Monte Carlo tree search (MCTS).

Minimax

O Algoritmo Minimax é um dos principais algoritmos usados em jogos de dois jogadores, como Xadrez, Damas e Connect-four. O objetivo deste algoritmo é determinar a melhor jogada que um jogador deve fazer em um determinado estado do jogo, sabendo que o outro jogador dará a melhor resposta possível para ela.

O minimax funciona através de uma árvore de jogadas possíveis, onde cada nó representa um estado do jogo e os nós filhos representam os movimentos possíveis desse estado e a procura começa na raiz da árvore que representa o estado atual do jogo.

A ideia do algoritmo minimax é avaliar cada nó da árvore do jogo calculando a pontuação ou utilidade desse nó para o jogador que está jogando no momento. Essa pontuação é calculada usando uma função de utilidade que atribui uma pontuação a cada estado do jogo. A pontuação de movimento mais alta é selecionada para o jogador atual, enquanto a pontuação de movimento mais baixa é selecionada para o oponente. A ideia é minimizar (minimizar) a perda máxima para o jogador e maximizar o ganho mínimo de (maximizar) para o oponente.

A complexidade do algoritmo minimax aumenta exponencialmente com a profundidade da árvore de movimentos possíveis.

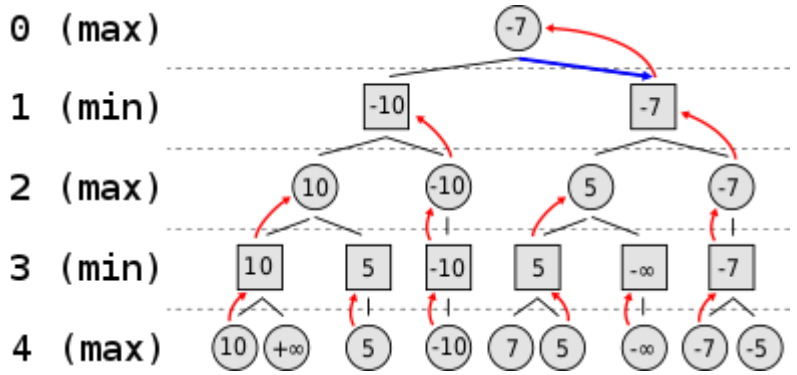

Figura 1 – Árvore gerada pelo algoritmo Minimax. Obtida em Mendonza (2012).

Alpha-Beta

De forma a reduzir significativamente o tempo de processamento necessário no algoritmo Minimax para avaliar jogadas em um jogo de dois jogadores foi criado o algoritmo Alpha-Beta. O seu objetivo é determinar qual a melhor jogada a fazer num certo estado do jogo, nunca esquecendo que o jogador adversário irá também fazer a melhor jogada possível.

Este algoritmo tem um funcionamento semelhante ao Minimax, ambos funcionam através de uma árvore de jogadas possíveis onde cada um dos nós representa uma possível jogada a partir de um estado e a sua ideia é fazer a avaliação de cada nó da arvore de jogo e calcular a pontuação do nó com base numa função de utilidade que escolhe sempre a utilidade mais alta.

No entanto, o Alpha-Beta usa duas variáveis, α e β, que representam a melhor escolha do jogador maximizador (α) e do jogador minimizador (β). Em cada um dos nós os valores de α e β são atualizados e no fim da procura esses valores são utilizados para decidir qual é a melhor jogada a ser feita.

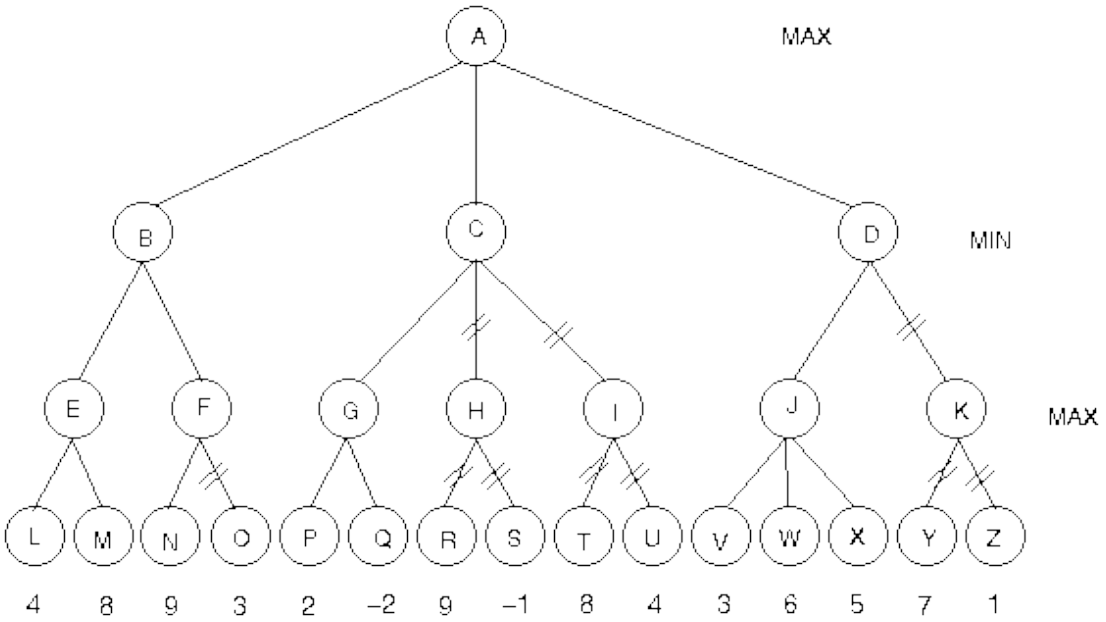

Figura 2 – Exemplo de árvore na qual o caminho com branches cortados não precisam ser explorados, pois caminhos melhores já foram encontrados. Obtida em Piech (2013).

Monte Carlo tree search

Este algoritmo está repartido em quatro fases diferentes: seleção, expansão, simulação e retropropagação. Ele simula várias jogadas escolhidas aleatoriamente e avalia a sua qualidade e no fim escolhe a melhor jogada baseando-se nos resultados obtidos nesse conjunto de simulações.

Após criar a sua arvore de procura e em cada um dos nós, escolhidos de forma

aleatória da arvore, o Monte Carlo tree search simula possíveis jogadas e faz a sua avaliação. Depois essa mesma avaliação é trazida novamente para o nó pai e atualiza o seu valor, sendo que no fim, e depois de ter repetido este processo enumeras vezes, ele escolhe o nó com o a melhor avaliação.

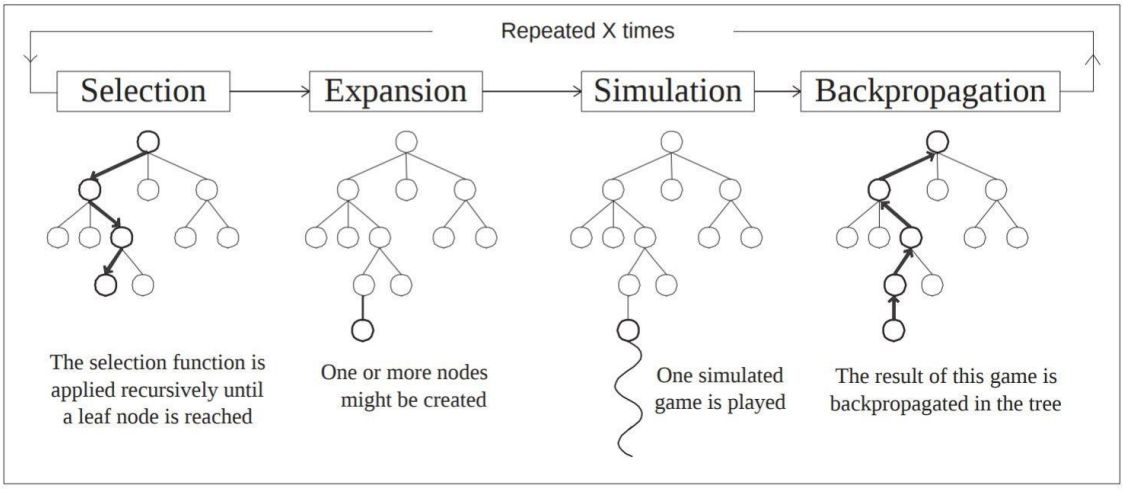

Figura 3 – Esquema para a Montecarlo Tree Search. Obtida em Chaslot, Winands e Henrik (2008)

Connect Four

O jogo de estratégia Connect Four consiste num tabuleiro vertical com 7 colunas de largura e com, no máximo 6 peças em cada coluna, em que dois jogadores, cada um com uma cor ou símbolo respetivo, vão alternando as suas jogadas. O primeiro que consiga 4 das suas peças alinhadas numa coluna, linha ou diagonal vence o jogo, sendo que é possível que um jogo termine empatado sem vencedor.

No nosso jogo jogaremos contra o computador, este terá ajuda dos algoritmos acima mencionados como ajuda para vencer. Ao computador será atribuído o símbolo "X" e ao outro jogador o símbolo "O". Para ajudar na implementação serão usadas as seguintes regras para avaliar os vários segmentos do tabuleiro:

- -50 para três "O" consecutivos sem nenhum "X",
- -10 para dois "O" consecutivos sem nenhum "X",
- -1 para um "O" sem nenhum "X",
- 0 para nenhuma ficha, ou com mistura de "X" e "O",
- 1 para um "X" sem nenhum "O",
- 10 para dois "X" consecutivos sem nenhum "O",
- 50 para três "X" consecutivos sem nenhum "O".

Descrição da Implementação

Linguagem Utilizada

A escolha da linguagem Java para implementar as buscas no problema de jogos com oponentes deve-se a várias razões. Em primeiro lugar, Java é uma linguagem de programação amplamente utilizada e com uma grande comunidade de desenvolvedores, o que torna mais fácil encontrar recursos e soluções para problemas específicos. Além disso, Java é uma linguagem orientada a objetos, o que significa que a implementação do problema pode ser modelada usando objetos e classes, tornando a implementação mais fácil de entender e manter.

Outra vantagem do uso de Java para resolver o problema do jogo Connect Four é a sua portabilidade. Java é uma linguagem multiplataforma, o que significa que o código pode ser executado em diferentes sistemas operacionais sem a necessidade de recompilação. Isso é útil para testar e executar as implementações das buscas em diferentes ambientes. Finalmente, Java também oferece uma grande variedade de bibliotecas e frameworks para trabalhar com estruturas de dados e algoritmos, o que pode tornar a implementação das buscas mais fácil e eficiente.

Estrutura do Código

Detalhamento das principais estruturas utilizadas no código que implementamos para o Connect Four:

Classe Puzzle

Representa o tabuleiro do jogo. Os seus atributos e alguns dos seus métodos auxiliares (apenas aqueles que achamos importante mencionar) são:

- Char [][] state: Representa o estado do jogo.
- Int[] valid: Representa a linha mais "funda" em que é possível jogar (para ocupada).
- String turn: Representa o próximo jogador a jogar (se for igual a "X's turn" é a vez do computador e se for igual a "O's turn" é a vez do jogador).
- Boolean draw: Indica se houve ou não empate no Puzzle atual.
- Int Utility: Guarda o valor da calculado com a função utilidade para o puzzle atual (só usamos este atributo para o Minimax e para o Alpha-beta)
- Void upload_board(String s): Carrega o tabuleiro representado por uma string s, para char[][] state.
- Void To_Move(): Tendo em conta o estado do tabuleiro (char[][] state) , verifica qual e o próximo jogador a jogar e atualiza turn.

- Void Actions(): Atualiza int[] valid consoante o estado do tabuleiro (char[][] state).
- Void Choose_col (int col): Permite escolher uma coluna em que vamos inserir a peça (caso essa coluna seja valida)
- Puzzle Result (Puzzle p, int col): Retorna o Puzzle resultante apos a escolha da jogada na coluna col (sem alterar o Puzzle original).
- Int Calculate_Utility(): Calcula o valor de utilidade associado ao Puzzle atual.
- Boolena Is_Terminal(): Com o auxílio de outros métodos, que não mencionamos, presentes na classe Puzzle, indica se o Puzzle atual é terminal.

Classe Node

Permite representar um nó da arvore de jogo. Os seus atributos e métodos auxiliares são:

- Puzzle current: Representa o puzzle associado ao nó atual.
- Int Utility: Representa o valor de utilidade do puzzle associado ao n
- Node Successor_is_Final(): Método que verifica se algum no sucessor do no atual e final.

Classe Main

Classe onde o utilizador pode indicar: o estado do Puzzle , o algoritmo que quer utilizar (Minimax ,Alpha-Beta ou MCTS) , ainda os respetivos "limites" associados a cada um dos algoritmos e por fim jogar contra o computador (seguindo as indicações que vao aparecendo no terminal)

Classe Diagonais

Utilizamos esta classe para guardar as diagonais de um Puzzle (só contem as diagonais de tamanho pelo menos 4), para facilitar o cálculo da função utilidade associada a um no e também para facilitar a verificação de Puzzle "final".

Classe MiniMax

Esta classe aplica o algoritmo Minimax a um nó, com um limite de profundidade dado pelo utilizador. Depois da execução do algoritmo o próximo nó a ser escolhido e guardado no atributo Node bestNode.

Classe AlphaBeta

Esta classe aplica o algoritmo AlphaBeta a um nó, com um limite de profundidade dado pelo utilizador. Depois da execução do algoritmo o próximo nó a ser escolhido e guardado no atributo Node bestNode.

Estrutura de dados usadas

As estruturas de dados usadas foram:

- ArrayList<Node>: Utilizamos para guardar os sucessores de um dado nó. Escolhemos esta estrutura por já ter uma implementação robusta no java e por ser análoga a um array. Em termos eficiência de manipulação dos dados achamos que para a objetivo, que é guardar e aceder a nós, é uma boa escolha.
- Matriz 6*7: Escolhemos esta estrutura de dados por ser a mais "simples" de entender e utilizar, na nossa opinião.

MCTS applied to Connect Four

Descrição da implementação:

- Classe Puzzle: Utiliza a mesma classe Puzzle que o Minimax e o Alpha-Beta utilizam.
- Classe Node_MCTS: Permite representar os nós que vamos utilizar na arvore de procura. Os seus atributos e métodos auxiliares são:
- Double value: Corresponde ao valor associado ao nó
- Int n: Número de vezes que o nó atual foi visitado
- Int N: Número de visitas do nó Pai do nó atual.
- Double UCB1: Valor Upper Confidence Bounds for Trees associado a este nó.
- Puzzle current: Puzzle associado ao nó atual.
- ArrayList<Node_MCTS> filhos: Lista de filhos do nó atual (inicialmente não contém nenhum nó, mas caso o nó atual seja expandido passa a conter os filhos do nó atual).

- Boolean isLeaf(): Verifica se o nó atual e um nó folha.
- Double Calculate_UCB1(): Retorna o valor do campo UCB1 do nó atual

Classe MCTS

Nesta classe é onde aplica o algoritmo de MCTS (seleção, expansão, simulação e retropropagação, O número de vezes que aplicamos o MCTS é definido por o utilizador.

Estrutura de dados usadas:

- ArrayList<Node_MCTS>: Utilizamos para guardar os sucessores de um dado nó. Escolhemos esta estrutura por já ter uma implementação robusta no java e por ser análoga a um array. Em termos eficiência de manipulação dos dados achamos que para a objetivo, que é guardar e aceder a nós, é uma boa escolha.
- Matriz 6*7: Escolhemos esta estrutura de dados, para representar o puzzle, por ser a mais "simples" de entender e utilizar, na nossa opinião.
- Node_MCTS: Utilizamos para representar um nó na árvore de jogo.

Execução do Programa

De forma a executar o programa temos de seguir um número de passos. Primeiro, e com o terminal aberto no diretório onde se encontra a pasta com este programa e a suas respetivas classes, compilamos e executamos o programa com o seguinte comando:

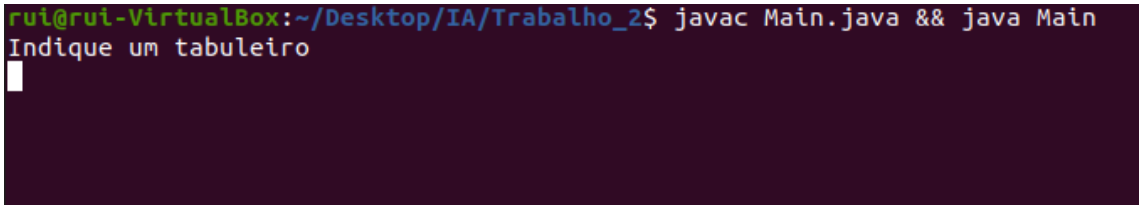

Figura 4 – Passo 1

Neste próximo passo iremos indicar o tabuleiro no qual será jogado este jogo. Na pasta do programa existe um ficheiro denominado de (..) onde podemos copiar o tabuleiro e depois colar no terminar:

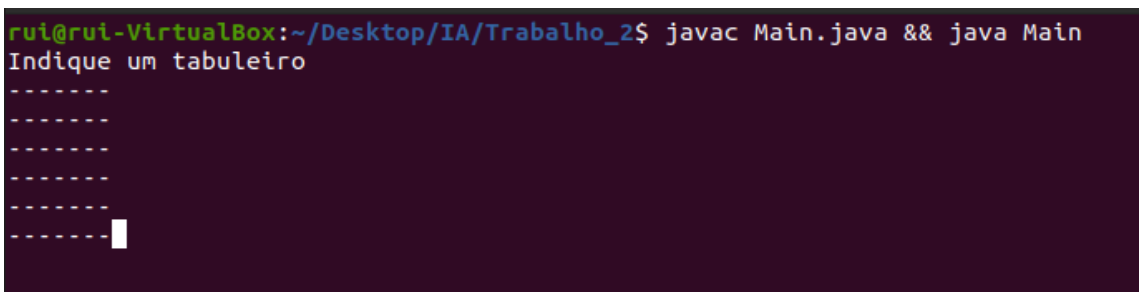

Figura 5 – Passo 2

De seguido será apresentada a seguinte mensagem em que temos de selecionar qual o algoritmo desejado com o qual desejamos jogar:

```
rui@rui-VirtualBox:~/Desktop/IA/Trabalho_2$ javac Main.java && java Main
Indique um tabuleiro
-------
-------
-------
-------
-------
-------
Indique contra qual algoritmo quer jogar
minimax   alphabeta     montecarlo
```

Figura 6 – Passo 3

Escolhemos uma das três opções selcionadas escrevendo no terminar o nome da qual pretendemos utilizar:

```
rui@rui-VirtualBox:~/Desktop/IA/Trabalho_2$ javac Main.java && java Main
Indique um tabuleiro
-------
-------
-------
-------
-------
-------
Indique contra qual algoritmo quer jogar
minimax   alphabeta     montecarlo
minimax
Indique a profundide maxima (< 8 senao demora muito)
7
-------
-------
-------
-------
-------
-------
0123456

It is now O's turn
Make a move by choosing your coordinates to play (0 to 6).
```

Figura 7 – Passo 4

Neste caso foi escolhido o algoritmo MiniMax e, tal como acontece quando selecionamos AlphaBeta, temos de indicar a profundidade a ser utilizada. No entanto, para o algoritmo montecarlo é pedido para escolher o número de vezes que este algoritmo será aplicado:

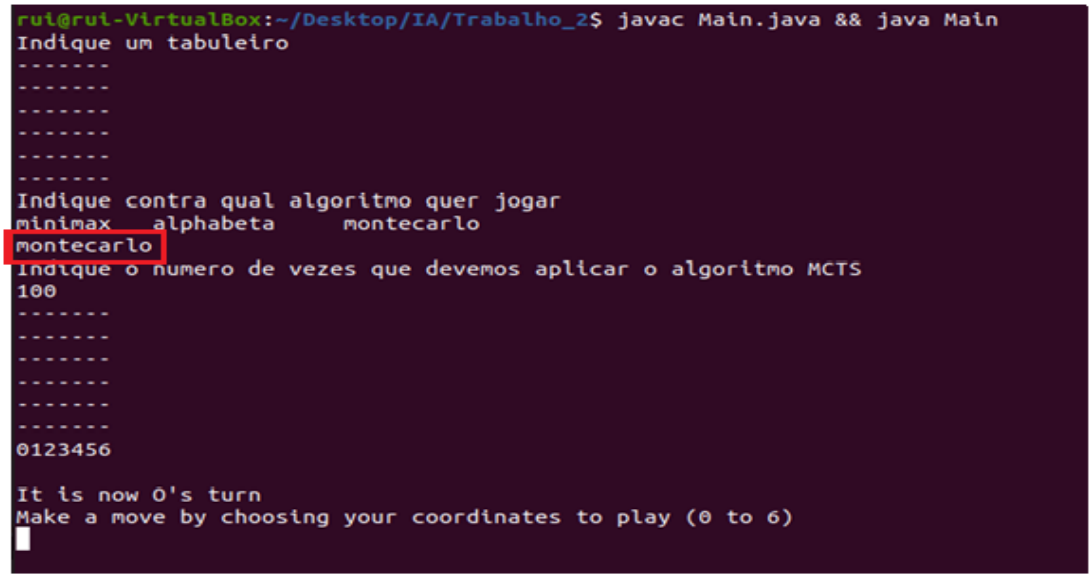

Figura 8 – Passo 5

Depois para jogar basta escolher um número entre 0 e 6, que reprensetam o número das colunas nas quais se vai jogar. Este processo é repetido até se alcançar um resultado final, seja vitória, empate ou derrota:

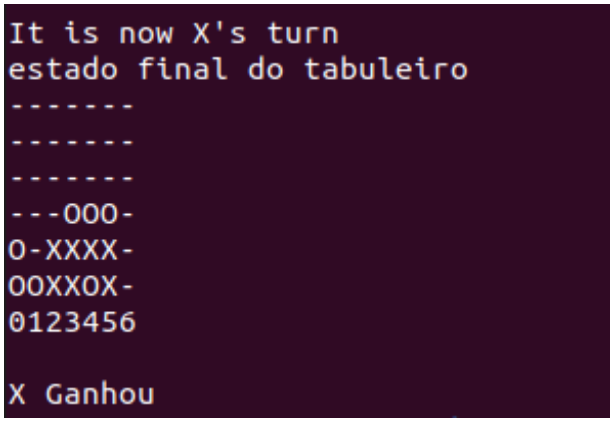

Figura 9 – Passo 6

Conclusão

Com base na análise dos resultados obtidos durante a elaboração deste projeto podemos chegar a diferentes conclusões sobre os diferentes algoritmos utilizados. Cada uma das abordagens aprensenta as suas limitações e vantagens.

Embora cada um dos algorismos tenha as suas diferentes utilidades, podemos concluir que geralmente a melhor opção para jogos mais complexos, como é o Connect Four, é a Monte Carlo Search Tree, , como constatado por Sheoran, Dhand, Dabaszs et al., (2022).. A MCTS apenas procura as partes mais promissoras do espaço de procura economizando assim tempo e memória como podemos ver nos próximos gráficos por comparação aos outros algoritmos utilizados:

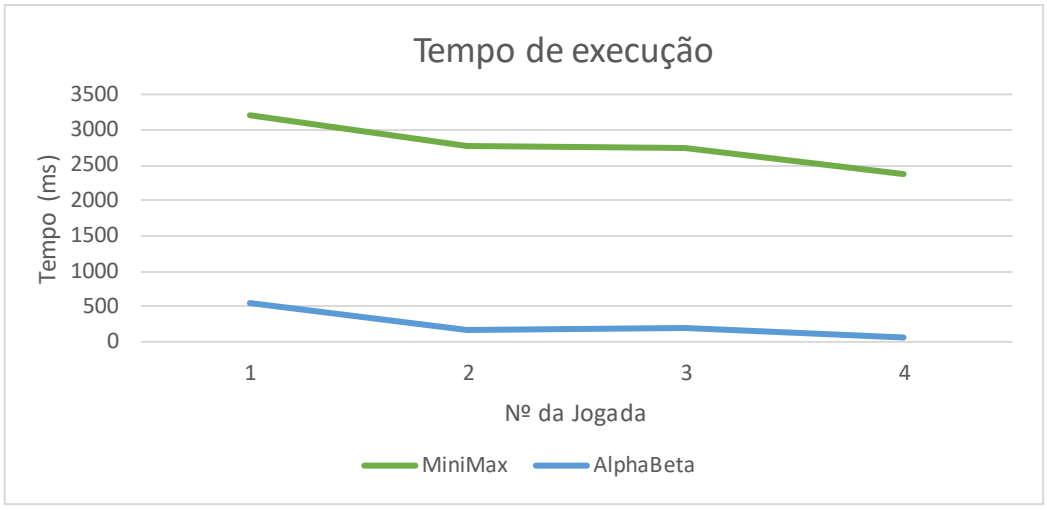

Gráfico 1- Tempo de execução por jogada

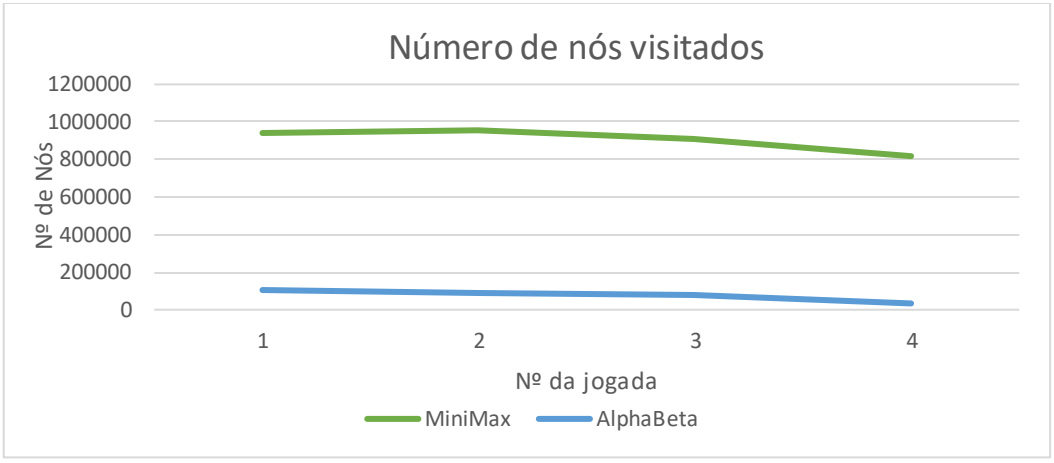

Gráfico 2 - Nós visitados por jogada

De forma a analisar melhor os resultados decidimos seprar o MCTS dos outros dois algarismos, porque a disparidade entre os números é muito alta e desta forma facilitar a análise. Assim sendo obtivemos os seguintes gráficos:

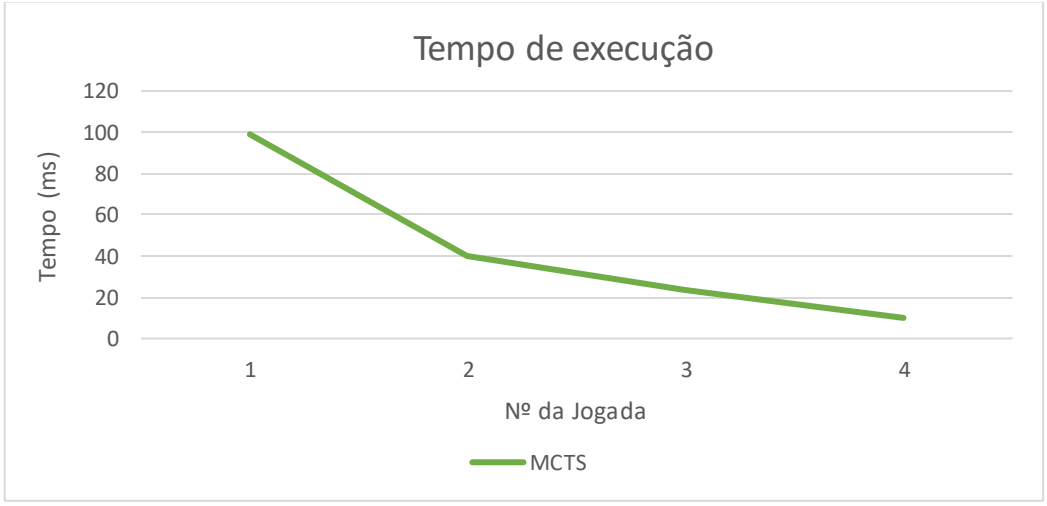

Gráfico 3 – Tempo de execução por jogada em MCTS

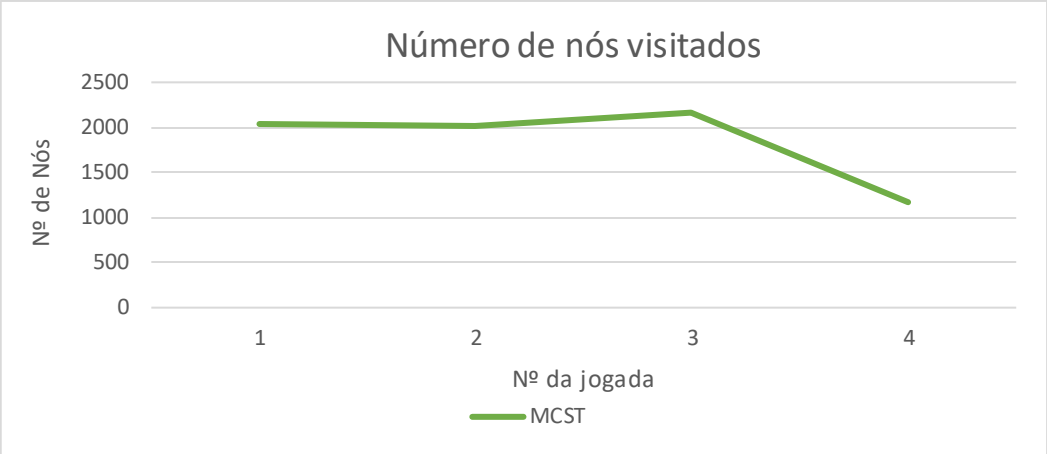

Gráfico 4 – Nós visitados por jogada em MCST

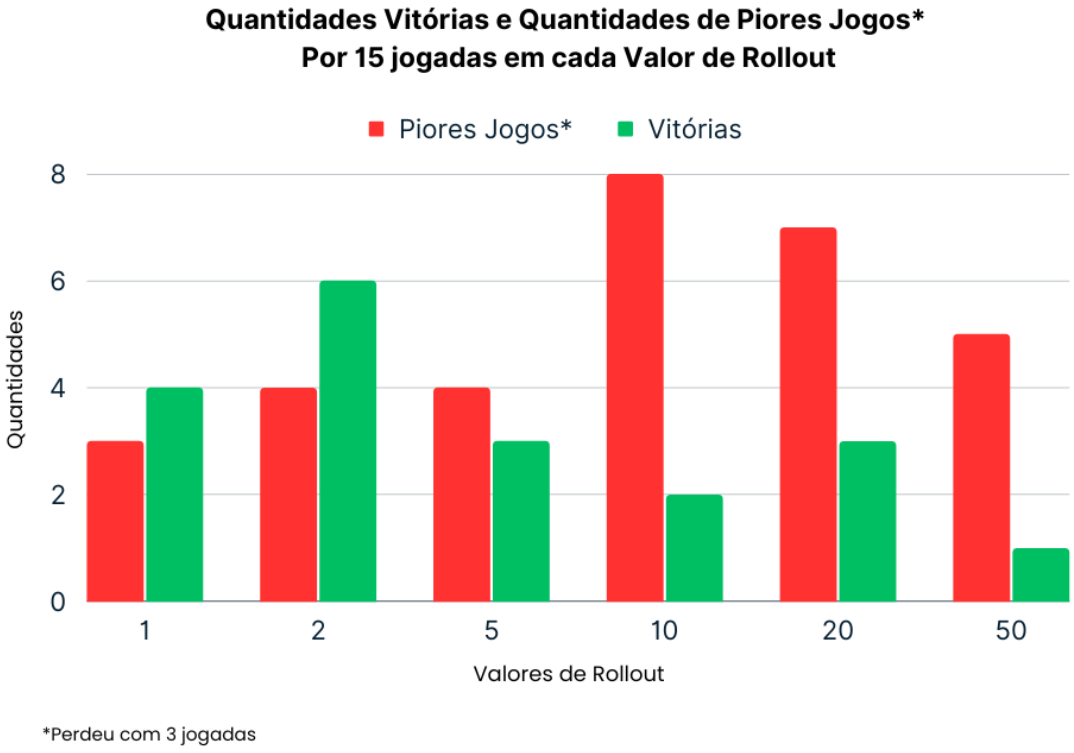

Gráfico 5- Análise dos resultados da MCTS

Com relação a como determinamos o valor de rollout, a quantidade de jogadas é muito baixa e os jogos aleatórios, portanto estes valores não são uma representação suficiente. Contudo, mesmo assim fizemos estes testes para tentar perceber alguma diferença no comportamento das jogadas do computador por valores de rollout diferente, com a finalidade e tentativa de determinar de modo sistemático (Gaina, Devlin, Lucas e Perez-Liebana, 2021) um melhor valor para o rollout. Por fim determinamos que usaríamos 2.

Bibliografia

Gaina, R., Devlin, S., Lucas, S., e Perez-Liebana, D. (2021). Rolling horizon evolutionary algorithms for general video game playing. *IEEE Transactions on Games*, 14(2), 232-242. Disponível em https://ieeexplore.ieee.org/abstract/document/9357946

Mendoza, P. (2022). *Alpha-Beta Pruning Algorithm: The Intelligence Behind Strategy Games.* Disponível em https://www.researchgate.net/publication/360872512

Nilsson, N. (1998). *Artificial Inteligence: a new synthesis.* Morgan Kaufmann Publishers.

Piech, C. (2013). *Deep Blue.* Disponível em https://stanford.edu/~cpiech/cs221/apps/deepBlue.html

Poole, D. e Mackworth, A. (2017). *Artificial Intelligence: Foundations of Computational Agents.* Cambridge University Press.

Ribeiro, P. (2022). *Estruturas de Dados.* Disponível em https://www.dcc.fc.up.pt/~pribeiro/aulas/edados2223/

Russell, S., Norvig, P. (2022). *Artificial Intelligence: A Modern Approach*. (4a edição). Pearson Education Limited. Global Edition.

Salloum, Z. (2019). *Monte Carlo Tree Search in Reinforcement Learning.* Disponível em: https://towardsdatascience.com/monte-carlo-tree-search-in-reinforcement-learning-b97d3e743d0f

Sheoran, K., Dhand, G., Dabaszs, M., Dahiya, N., & Pushparaj, P. (2022). Solving Connect 4 using Optimized Minimax and Monte Carlo Tree Search. *Advances and Applications in Mathematical Sciences.* Vol 21(6), April 2022. Disponível em: https://www.mililink.com/upload/article/279817393aams_vol_216_april_2022_a25_p3303-3313_kavita_sheoran,_et_al..pdf