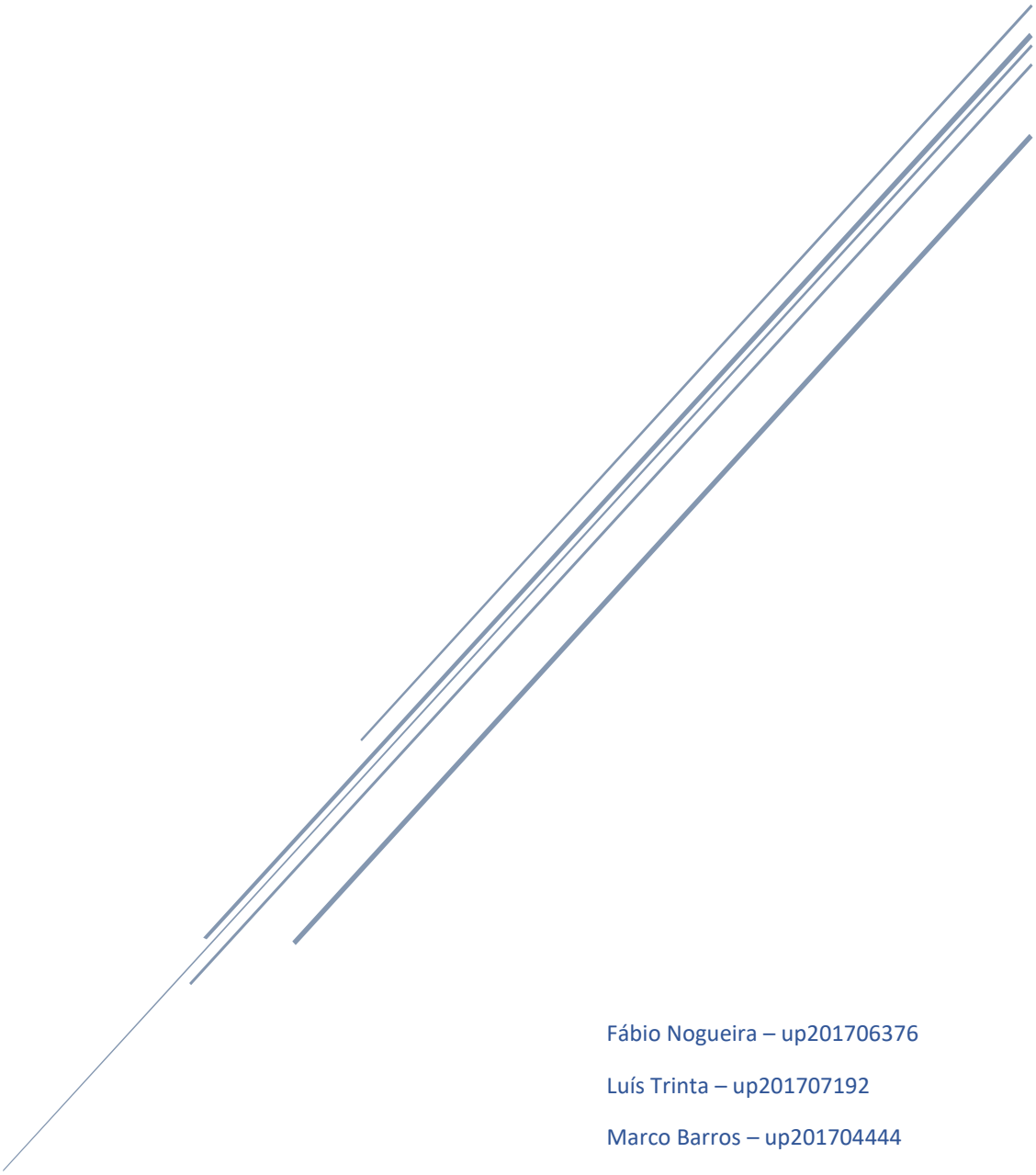
CONNECT 4

Criação e análise de algoritmos para jogos de oponentes



Fábio Nogueira – up201706376

Luís Trinta – up201707192

Marco Barros – up201704444

Inteligência Artificial

Indice:

Introdução

Com este projeto pretendemos comparar diferentes tipos de algoritmos de busca contraditória para o mesmo jogo, *Connect 4*, e avaliar os seus resultados.

Adversarial Search, ou busca contraditória, é um tipo de pesquisa utilizado para calcular a melhor jogada em jogos de dois jogadores onde toda a informação é dada. A pesquisa consiste em procurar todas as jogadas possíveis no jogo onde para cada movimento é representado por um valor dependendo da chance de ganhar ou perder.

O importante neste tipo de procura é maximizar a vitória da máquina e minimizar a possível derrota. Para calcular isto, abordaremos três tipos de algoritmos de busca contraditória: *Minimax*, *Alpha-beta Pruning* (Corte Alfa Beta) e *MCTS* (Busca em Árvore Monte-Carlo).

Minimax

O algoritmo *Minimax* consiste em determinar qual a melhor jogada, prevendo todas as possibilidades dentro dos próximos **m** movimentos, sendo **m** a profundidade máxima dada pelo utilizador ou o número de jogadas até o tabuleiro estar completo.

Para calcular este movimento, o programa deve **minimizar** a possível derrota e **maximizar** as chances de vitória. Para realizar este cálculo utilizaremos uma estrutura de árvore na qual cada nó é uma jogada possível sobre um nó/tabuleiro pai. A partir deste estado, o algoritmo pode gerar todos os possíveis estados futuros que podem resultar após o adversário jogar.

Quando o programa alcança a profundidade máxima (ou atinge um ponto crítico), este retorna uma pontuação do tabuleiro que alcançou, podendo avaliar assim qual o melhor caminho a seguir.

Complexidade temporal: $\boldsymbol{O(b^m)}$

Complexidade espacial: $\boldsymbol{O(m)}$

b = fator de ramificação, **m** = profundidade máxima

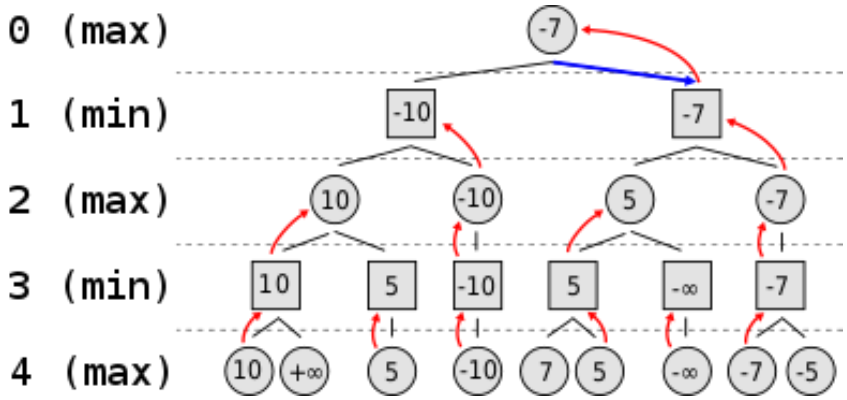


Fig.1-Representação do funcionamento do algoritmo Minimax

Alpha-beta Pruning

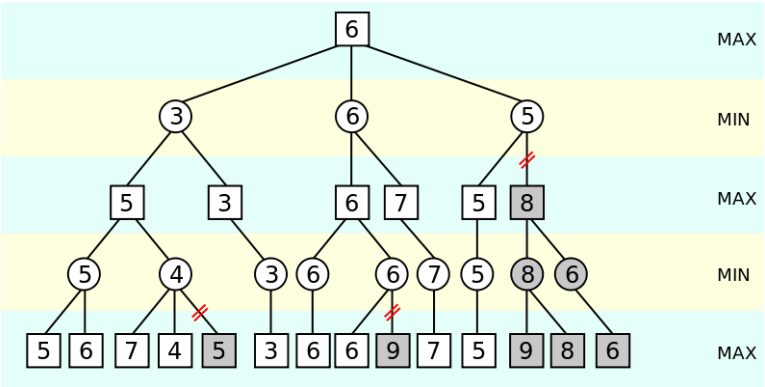
Esta técnica é uma otimização do algoritmo *Minimax* e serve para reduzir significativamente o tempo de processamento. Isto permite ao programa procurar mais rapidamente e até ir mais aprofundadamente na árvore do jogo.

Consiste em cortar caminhos que sejam inúteis, por exemplo, se o programa já sabe que consegue ter uma certa pontuação alta, não terá necessidade de procurar uma pontuação mais pequena.

Complexidade temporal: $\boldsymbol{O}\left(\boldsymbol{b}^{\frac{\boldsymbol{m}}{\boldsymbol{2}}}\right)$

b = fator de ramificação, **m** = profundidade máxima

Fig.2-Representação do funcionamento do algoritmo Minimax com Alpha-Beta Pruning



Monte Carlo *tree search* (MCTS)

MCTS é um método usado para tomar as melhores decisões em problemas de inteligência artificial, geralmente em jogos de movimentos combinatórios. Para que tal seja feito, este encontra o melhor movimento a através de uma **seleção**, **expansão**, **simulação** e **atualização** de nós na árvore para encontrar a solução final.

Seleção: Primeiramente o programa vai selecionar o nó que tem mais possibilidades de ganhar. Nas imagens seguintes utilizamos nós com as devidas chances de o fazer.

Expansão: Para aumentar o número de opções, expande-se o nó selecionado criando-se vários nós filho (Nesta demonstração apenas usaremos um). Estes nós são os futuros movimentos que podem ser jogados no futuro.

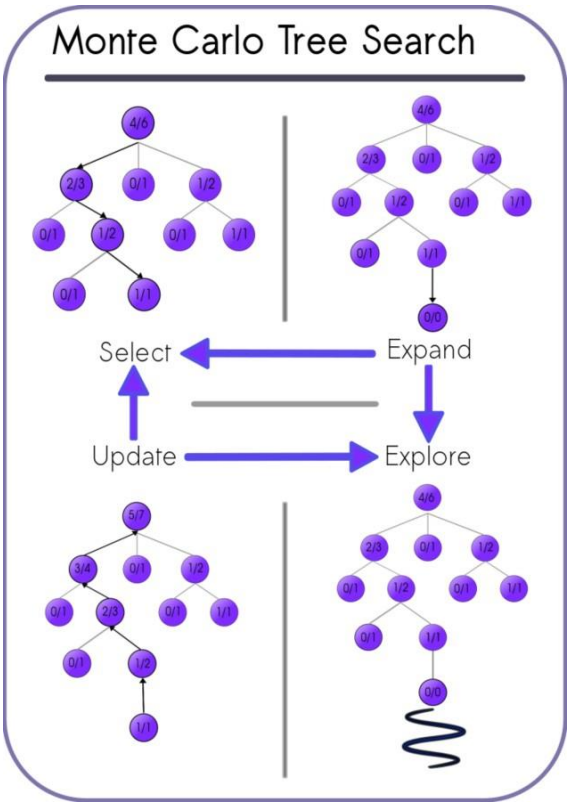
Simulação: Simulamos para descobrir qual o melhor nó expandido. Como obtemos isto? Utilizamos **Reinforcement Learning** (*goal-oriented algorithms* que aprendem a chegar a um objetivo ou a maximizar uma certa dimensão através de movimentos) para fazer decisões aleatórias no jogo a cada nó filho, sendo dado uma recompensa a este (ao calcular quão próximo está esse movimento aleatório do *output* final).

"*When it is not in our power to determine what is true, we ought to act in accordance with what is most probable*"- Descartes

(Neste caso vamos assumir que a simulação devolveu um nó otimista com probabilidade de 1/1)

Atualização: Segundo as pontuações dos novos nós, as pontuações dos seus pais devem ser alteradas subindo a árvore, um a um. Esta atualização muda o estado da árvore e pode mudar o próximo nó selecionado.

Depois de atualizarmos todos os nós, recomeçamos a técnica para encontrar um novo nós que será o resultado vencedor.



Complexidade temporal: $\boldsymbol{O(mkI)}$

Complexidade espacial: $\boldsymbol{O(mk)}$

m = número de filhos aleatórios

k = número de pesquisas paralelas

$\boldsymbol{I}$ = número de iterações

Connect Four

Quatro em linha (ou *connect Four*) é um jogo de tabuleiro entre 2 jogadores sobre um tabuleiro.
Cada jogador pode introduzir apenas uma peça por turno numa das colunas, fazendo a peça cair verticalmente até a primeira casa disponível, de baixo para cima. O primeiro jogador a conseguir colocar as próprias 4 peças seguidas na horizontal, diagonal ou vertical é o vencedor. Se o tabuleiro ficar completo, o jogo termina empatado.
No código usamos um tabuleiro com 7 colunas e 6 linhas e identificadores de peças são uma cruz(x) e um círculo(o).

"Objective: Connect four of your checkers in a row while preventing your opponent from doing the same. But, look out -- your opponent can sneak up on you and win the game!" - Milton Bradley, Connect Four "Pretty Sneaky, Sis" television commercial, 1977

A interpelação para a vitória da máquina pode ser feita de forma algorítmica, apesar do 4 em linha ser um jogo **resolvido**, ou seja, o primeiro jogador tem sempre maior vantagem e poderá sempre ganhar, se forem feitas as jogadas certas.

Contagem de Pontuação

Para o nosso programa calcular a melhor jogada precisamos de criar um algoritmo. Esta porção de código verifica a sequência de peças de cada um dos jogadores e incrementa dois contadores: um próprio e um para o oponente.

Funciona da seguinte forma: Esta função vai verificar quatro tipos de posições – horizontal, vertical, e diagonais – e verificar, de 4 em 4 espaços, que peças é que encontra. Caso encontre só peças de um tipo dentro dessas 4, ele vai incrementar uma variável "total" com 1, 20, 50 ou 512 ponto caso encontre apenas uma, duas, três ou quatro das próprias peças (caso contrário incrementa o oposto destes pontos).

Aplicação de Minimax

Para a implementação do minimax, utilizamos 2 funções principais, uma para o valor mínimo e uma para o valor máximo. Caso o programa verifique que um dos jogadores ganhou, devolve o valor 512, que representa uma pontuação perfeita, caso seja o PC a ganhar e -512 caso o oponente ganhe.

Começando a uma profundidade que o utilizador quiser (utilizamos 8 para obtermos os melhores resultados sem problemas de tempo) calcular todas as combinações de X jogadas possíveis (sendo X a profundidade máxima) e devolver uma pontuação quando este chega à última jogada. Após isto, compara todas as pontuações das n combinações possíveis e devolve a mais favorável.

Aplicação de Alpha-beta Pruning

Esta aplicação é feita a partir do algoritmo já criado minimax. Para cortarmos ramificações temos que devolver pontuações mais cedo. Para fazer isto criamos 2 novas variáveis que acompanharão o algoritmo através das profundidades: alpha e beta. Alpha ficará sempre com o maior valor entre o score de cada nó e o valor dele mesmo no nó pai. Em contrapartida, Beta terá o valor mais baixo entre ele no nó pai e o score atual. Se eventualmente beta for menor que alpha, o programa assumirá que é inútil prosseguir com a avaliação dos filhos desse nó, pois a chance de o oponente não ganhar já é maior do que de ganhar

Aplicação de MCTS

Para a aplicação do MCTS ao jogo 4 em Linha foi necessária a criação de uma class MonteCarlo que por sua vez contem um Node root , que será sempre o estado inicial da árvore , uma constante utilizada no método select e o tempo de execução dado pelo utilizador que corresponde á duração da pesquisa feita pelo algoritmo em si .Dentro da classe MonteCarlo existe também uma classe Node que contem as variáveis necessárias para o funcionamento do algoritmo(Node pai , Array de filhos , Tabuleiro do jogo , Contador de visitas e de pontuação).

O funcionamento do algoritmo em si deve – se ao método getOptimalMove() .Este é responsável pela seleção do nó em si realizar a expansão desse nó, a obtenção do seu valor através da simulação e aplicar a *backpropagation*. Após este processo ser repetido inúmeras vezes dentro do tempo de execução predefinido ,é retornado o *índex* do melhor filho. Esta escolha baseia-se no número de visitas realizadas em cada nó.

Seleção

Na parte da seleção é selecionada sempre a *root* da árvore de Nodes(definida na classe MonteCarlo). Após a escolha da *root* , verificamos se todos os seus filhos foram expandidos ,caso ainda não tenham sido todos expandidos é retornado o nó pai (inicialmente a *root*).No caso de todos os filhos já terem sido expandidos , é selecionado o filho que apresenta um melhor valor de UCB(Upper Confidence Bound) obtido através da fórmula $UCB = Xj +$

$C\sqrt[2]{\frac{2\ln n}{nj}}$. Após a obtenção do melhor filho retorna-se um *select* desse filho e o processo recomeça até ser encontrado um nó pai que ainda necessite expandir filhos.

Expansão

Ao expandirmos um nó , verificamos quais dos filhos desse nó ainda não estão expandidos e se são posições válidas no tabuleiro (coluna tem espaço).
Caso apresentem as condições acima , a posição desses filhos é adicionada a uma *ArrayList* que será utilizada para a escolha aleatória do filho a expandir.É retornado da função *expand* o nó filho em que será realizada a simulação.

Simulação

Na simulação é criada uma nova tabela que será uma cópia da tabela contida no nó em que está a ser aplicado o método *simulate*. De seguida irão ser realizadas n jogadas no tabuleiro até que uma das condições de paragem seja alcançada. De seguida verifica -se o resultado da simulação:
-Em caso de empate ou derrota é retornado 0;
-Em caso de vitória é retornado 1;

Backpropagation

Ao realizarmos a *Backpropagation* de um nó até á *root* da árvore iremos realizar um processo cícliclo até alcançarmos a root:

- Incrementar o número de visitas do nó em 1;
- Somar ao valor de vitoria de cada nó o valor de vitória da simulação do nó escolhido para a simulação;
- Mudar de nó(filho -> pai);

Conclusão

Sendo o corte alfa beta uma modificação do minimax que permite eliminar caminhos inúteis, podemos assumir, sem provas, que o tempo de execução será substancialmente menor. Para confirmarmos essa teoria pusemos AI minimax contra ele próprio, com uma profundidade de 6 e o AI minimax com alfa-beta pruning, também contra ele próprio e com profundidade 6.

Transcrevendo os resultados recebidos, construímos estes dois gráficos:

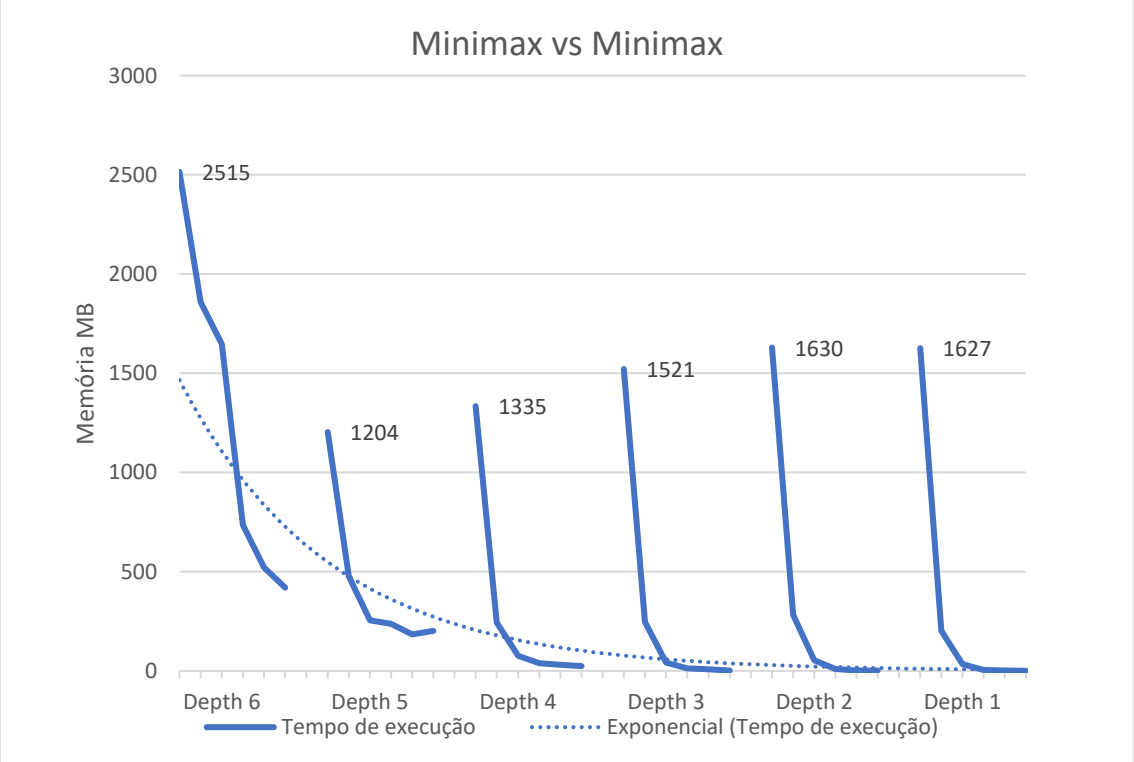


Tabela 1 – Minimax vs Minimax testado em cada profundidade contra todas as profundidades entre 6 e 1.

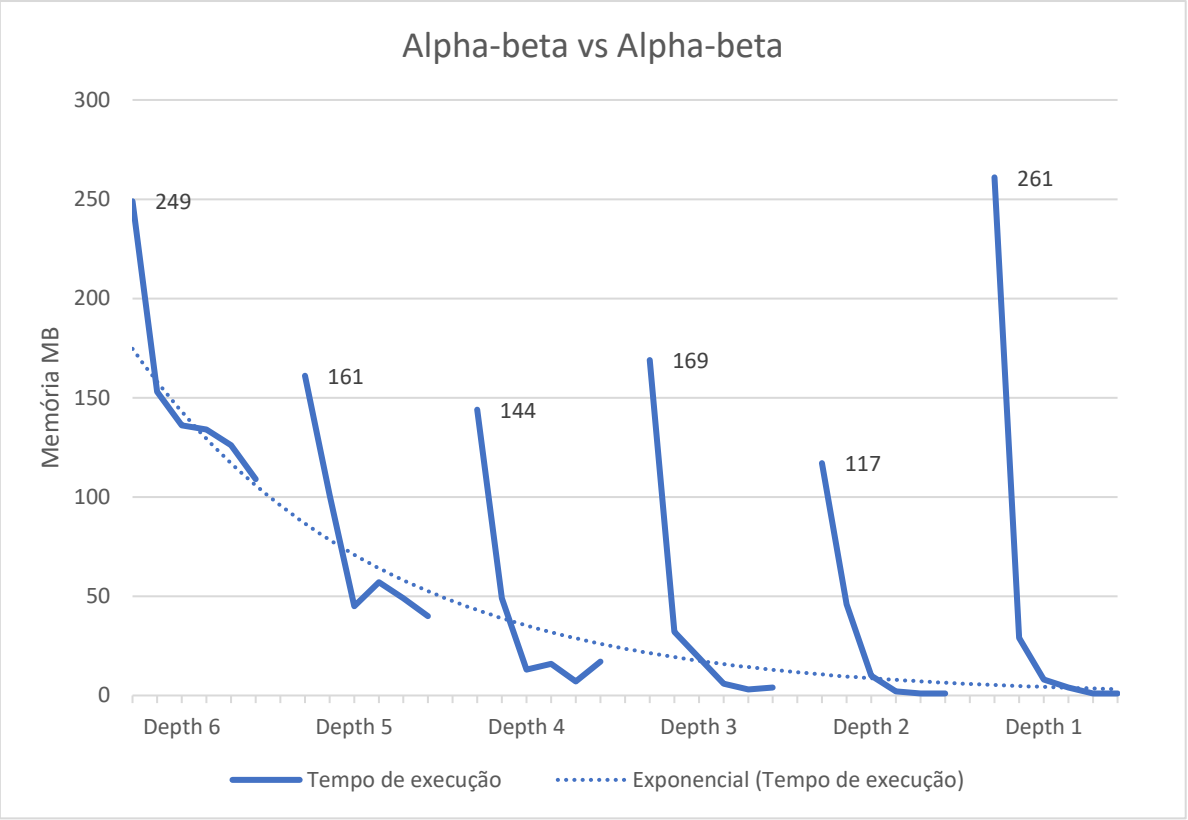


Tabela 2 - Alpha-beta vs Alpha-beta testado em cada profundidade contra todas as profundidades entre 6 e 1 (para termos de comparação com Minimax vs Minimax, Tabela 1.

Como podemos verificar (agora com provas), o tempo de execução chega a ser até 10 vezes menor no alfa-beta em relação ao minimax.

O algoritmo de pesquisa em árvore Monte Carlo explica que em vez de usarmos força bruta e vermos as milhões de formas possíveis do caminho certo (como no minimax), podemos usar Reinforcement Learning calcular mais eficazmente e mais rapidamente. Tendo isto em mente assumiríamos que utilizar o MCTS seria o melhor para este projeto, mas os dados que recolhemos mostram que isto não acontece de forma tão linear:

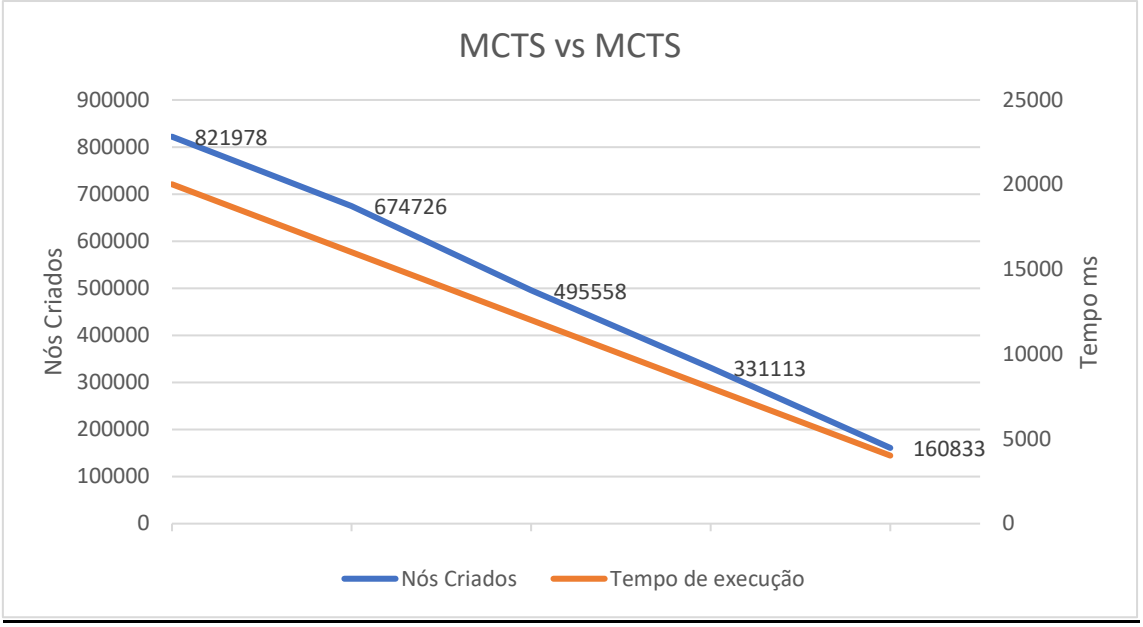


Tabela 3 - MCTS vs MCTS em tempo de execução de 5, 4, 3, 2 e 1 segundos, respetivamente.

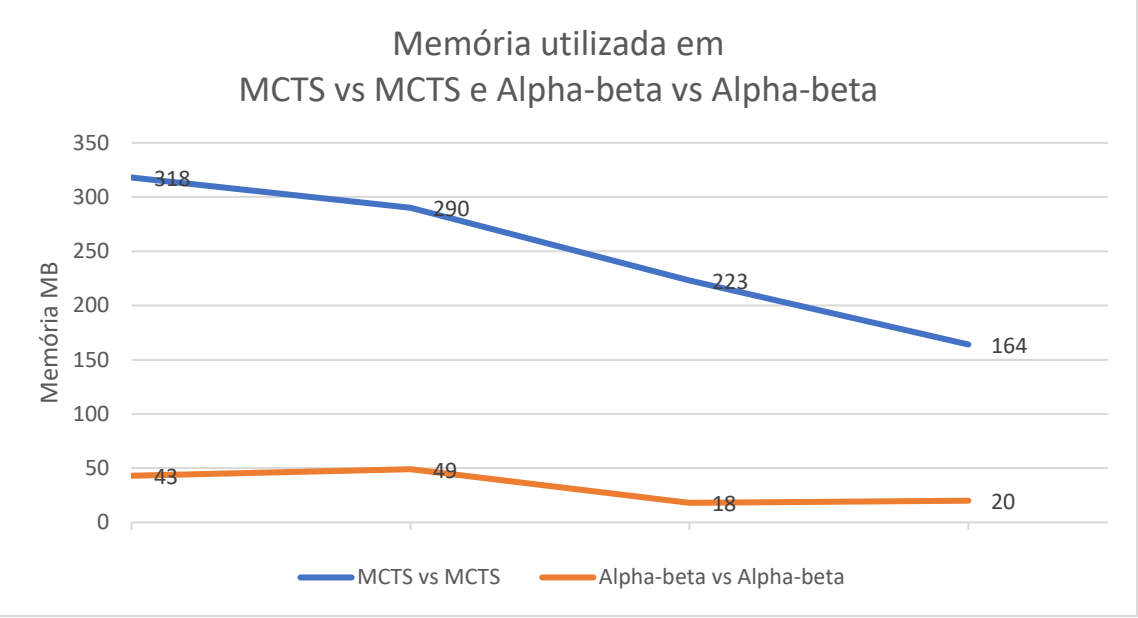


Tabela 4 – MCTS vs MCTS em tempo de execução de 5, 4, 3 e 2 segundos, respetivamente, e Alpha-beta vs Alpha-beta com profundidade 8, 7, 6 e 5, respetivamente.

Isto acontece porque o MCTS é um algoritmo demasiado poderoso para o projeto em questão. Este método deve ser usado para jogos com movimentos infinitos (como por exemplo xadrez), algo que o 4 em linha não é, já que cada movimento diminui o número de espaços livres.

"When faced with a problem, the a priori choice between MCTS and minimax may be difficult. If the game tree is of nontrivial size and no reliable heuristic exists for the game of interest, minimax is unsuitable but MCTS is applicable." - *Cameron Browne, A Survey of Monte Carlo Tree Search Methods, 2012*

Referências

Game theory — Minimax, *by NerdzLab*
https://towardsdatascience.com/game-theory-minimax-f84ee6e4ae6e

Minimax Algorithm in Game Theory, *by Akshay L Aradhya*
https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/

Game-tree Search and Pruning Algorithms, *by Mohammad T. Hajiaghayi, University of Maryland*
http://www.cs.umd.edu/~hajiagha/474GT15/Lecture12122013.pdf

Monte Carlo Tree Seacrh, *by Sagar Sharma*
https://towardsdatascience.com/monte-carlo-tree-search-158a917a8baa

Reinforcement learning, *by Prateek Bajaj*
https://www.geeksforgeeks.org/what-is-reinforcement-learning/

Reinforcement learning, *by Prateek Bajaj*
http://www.cs.umd.edu/~hajiagha/474GT15/Lecture12122013.pdf

What is MCTS?, *by mcts.ai*
http://mcts.ai/about/