JOGOS COM OPONENTES

FCUP

Trabalho 2

Inteligência Artificial

21-04-2022

Ana Catarina da Silva Costa Gomes (up201804545)
Cláudia da Costa Maia (up201905492)
Maria Santos Sobral (up201906946)

ÍNDICE

Ana Catarina da Silva Costa Gomes (up201804545)
Cláudia da Costa Maia (up201905492)
Maria Santos Sobral (up201906946)

INTRODUÇÃO

Os jogos com dois jogadores não podem ser resolvidos recorrendo a algoritmos de buscas, uma vez que estes não têm em consideração a presença de um oponente.

Para a resolução deste tipo de jogos é necessário considerar não só as ações do outro jogador, como também as suas consequências.

Estes jogos são tradicionalmente mais difíceis de resolver, pois envolvem muitas jogadas, pelo que a árvore de pesquisa terá muitos nós.

Existem duas ferramentas essenciais para a escolha de uma jogada ótima: a poda da árvore (que consiste em ignorar ramos que não interferem na escolha final) e as heurísticas (que são funções de avaliação que nos permitem calcular de forma aproximada a utilidade de uma jogada, sem termos de fazer uma pesquisa completa).

ALGORITMO MINIMAX

Neste algoritmo, MAX (o computador) considera que MIN (o seu oponente) joga de forma ótima, pelo que valores mínimos de utilidade favorecem o oponente e valores máximos de utilidade favorecem o computador. Assim sendo, temos a seguinte função recursiva:

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

onde para um determinado estado *s* do tabuleiro, *Player(s)* indica quem é o próximo a jogar, *Terminal-Test(s)* retorna verdade no caso de o jogo já ter terminado, *Actions(s)* é o conjunto das jogadas possíveis, *Utility(s)* é a função utilidade usada e *Result(s,a)* é o resultado da jogada *a*.

O Minimax é uma estratégia completa, que tem complexidade espacial de O(bm), onde b é o fator de ramificação e m é o nível de profundidade máxima da árvore. Porém, o número de nós da sua árvore é exponencial relativamente à profundidade da árvore, tendo complexidade temporal de O(b^m).

Ana Catarina da Silva Costa Gomes (up201804545)
Cláudia da Costa Maia (up201905492)
Maria Santos Sobral (up201906946)

ALGORITMO ALPHA-BETA

Esta estratégia consiste em podar a árvore do algoritmo Minimax. Uma vez que existem ramos que nunca serão escolhidos (partindo do princípio que o adversário joga de forma ótima), podemos eliminar essas partes, visto que não terão influência alguma na decisão final. Para isso, precisamos apenas que o algoritmo guarde as melhores jogadas encontradas até ao momento para MIN e também para MAX:

- alfa = valor da melhor escolha para MAX (que corresponde ao maior valor encontrado)
- beta = valor da melhor escolha para MIN (que corresponde ao menor valor encontrado)

Os valores de alfa e de beta vão sendo atualizados e os restantes ramos são eliminados a partir do momento em que sabemos que o valor do nó corrente é pior que o valor atual de alfa/beta para o MAX/MIN, respetivamente.

Através desta técnica, o Alpha-Beta, apesar de utilizar o mesmo raciocínio que o Minimax, consegue reduzir significativamente a sua complexidade temporal para O(b^m/2).

MONTE CARLO TREE SEARCH (MCTS)

Este algoritmo é relativamente recente e consiste numa aplicação do método de Monte Carlo, que transforma problemas determinísticos em problemas estatísticos, a árvores de pesquisa com alternância entre jogadores. Havendo uma série de versões deste algoritmo, dependendo particularmente do tipo de jogo que se pretende jogar, focaremos a explicação na estratégia específica que implementamos, tendo usado como principal referência a quarta edição do livro recomendado para esta UC, assim como o link (1) sugerido no Moodle (ver referência na bibliografia).

Assim, o MCTS pretende determinar a jogada mais promissora expandindo apenas ramos que lhe parecem úteis. Em vez de expandir todos os nós até atingir um terminal ou usar uma heurística a partir de determinada profundidade, o valor de um dado estado do jogo é estimado de acordo com a média das simulações de jogos completos que começam naquele estado. Para tornar este valor confiável, fazemos um número considerável de iterações do seguinte conjunto de etapas.

Ana Catarina da Silva Costa Gomes (up201804545)
Cláudia da Costa Maia (up201905492)
Maria Santos Sobral (up201906946)

Existem quatro etapas fundamentais que fazem parte deste processo e que vão sendo repetidas sucessivamente:

- seleção: começando na raiz da árvore, escolhemos um movimento (de acordo com a política de seleção), o que nos leva a um sucessor do nó corrente; repetindo o processo, vamos descendo na árvore até atingir uma folha.
- expansão: a árvore vai sendo expandida cada vez que escolhemos um nó novo.
- simulação: escolher movimentos para ambos os jogadores de forma aleatória; estes movimentos não são guardados na árvore de pesquisa, servem apenas para testar qual a escolha mais inteligente.
- propagação de volta: através dos resultados das simulações, atualizar todos os nós da árvore desde o atual até à raiz, contabilizando a vitória, caso se registe.

Para que este algoritmo seja eficiente, precisamos de ter uma política de seleção que consiga balancear a exploração dos nós menos visitados com a exploração de nós que tiveram bons resultados em simulações anteriores (processo a que se dá o nome de exploration/exploitation tradeoff).

A política de seleção que iremos utilizar neste trabalho chama-se *Upper Confidence Bounds*. Para um determinado nó n, a fórmula é a seguinte:

$$\text{UCB (n)} = \frac{U(n)}{N(n)} + C * \sqrt{\frac{log_e \text{N(Parent(n))}}{N(n)}}$$

onde U(n) é total de vitórias registadas nas simulações que partem de n, N(n) é o número de simulações através de n e Parent(n) é o pai do nó n.

Assim sendo, $\frac{U(n)}{N(n)}$ é o termo que representa a *exploitation* e $\sqrt{\frac{log_e \text{N(Parent(n))}}{N(n)}}$ é o termo que considera a *exploration*. A constante C equilibra os dois termos.

CONNECT FOUR

O *Connect Four* é representado por um tabuleiro de 6 linhas e 7 colunas, onde dois participantes vão jogando vez à vez, sendo que uma jogada consiste em deixar cair uma das suas peças numa coluna que ainda não esteja totalmente preenchida (as peças dos dois jogadores têm cores diferentes para os distinguir). Esta peça irá ficar na posição mais baixa dessa coluna que ainda não esteja ocupada. Um concorrente ganha se conseguir formar uma sequência com 4 das suas peças seguidas na horizontal, na vertical ou na diagonal. De forma a simplificar a representação do tabuleiro, iremos distinguir as peças dos dois jogadores não

através de cores diferentes, mas através dos caracteres 'X' e 'O'. Sendo que o 'X' é o computador e 'O' representa o seu adversário.

Para o propósito deste trabalho, iremos utilizar a função de avaliação seguinte, uma vez que esta se revelou ser bastante eficaz:

- uma vitória de 'X' vale +512
- uma vitória de 'O' vale -512
- um empate vale 0

Tendo em conta o branching factor da árvore de decisão desencadeada para este jogo, torna-se necessário usar uma heurística para estimar a utilidade do nó quando atinge um dado limite de profundidade, estabelecido à priori. Essa função consiste em analisar todos os possíveis segmentos de tamanho 4, seja em linha, coluna ou diagonal, e atribuir os seguintes valores consoante o número de X e O.

3 X -> 50	1 O -> -1
2 X -> 10	2 O -> -10
1 X -> 1	3 O -> -50

0 X, 0 O ou mistura de X e O -> 0

DESCRIÇÃO DA IMPLEMENTAÇÃO

Começando pela escolha da linguagem, Java foi a escolhida, uma vez que é a que os elementos do grupo mais usam, tendo adquirido uma certa destreza particularmente com esta linguagem. A estrutura de dados usada para a representação do tabuleiro é uma matriz de Strings 6x7. Para a representação dos nós, a escolha foi feita tendo em conta as necessidades de cada algoritmo, pelo que no Minimax e AlphaBeta não foi necessário customizar nenhuma estrutura específica, uma vez que os dados relevantes que devem acompanhar o tabuleiro eram passados como parâmetros entre as sucessivas chamadas recursivas às respetivas funções. Já no MCTS, a informação que era necessário guardar era diferente, além de ser em maior quantidade, a estrutura do algoritmo não permitia a passagem de informação como parâmetros das várias funções, pelo que se tornou necessário criar uma estrutura de nós da árvore customizada para o propósito, caracterizada pela representação habitual do tabuleiro, jogada que lhe deu origem, apontador para o nó pai, array para guardar os filhos, número de visitas do nó e número vitórias. Além disso, era necessário ter uma função que calculasse o UCB.

Ana Catarina da Silva Costa Gomes (up201804545)
Cláudia da Costa Maia (up201905492)
Maria Santos Sobral (up201906946)

Relativamente à implementação do Minimax, esta parte do estado inicial do tabuleiro e realiza uma jogada com "X", partindo em seguida para a determinação do valor deste tabuleiro, através da escolha do valor mínimo dos estados sucessores, fazendo-se uma troca de jogador e de objetivo, minimizar ou maximizar, a cada descida no nível da árvore. O caso base desta recursão, teoricamente, é quando se atinge um estado terminal, atribuindo o respetivo valor. No entanto, tendo em conta a dimensão da árvore de decisão para este jogo e o facto de os recursos serem limitados, impõe-se um limite de profundidade a partir do qual se estima o valor do tabuleiro usando a heurística acima descrita.

Este limite é determinado com base no número da jogada, na segunda e terceira jogadas do computador o nível de profundidade máximo é 3 e 6, respetivamente, a partir daqui passa a ter um limite de profundidade fixo de 8. Esta opção é fundamentada pelo facto de no início do jogo ainda há imensas possibilidades de ganhar, sendo também impossível perder nas 3 primeiras jogadas, já que ainda não foram colocadas 4 peças de cada jogador. Além disso, tem-se um tabuleiro bastante vazio, pelo que existem mais nós para expandir. A partir da terceira jogada torna-se ainda mais importante ter uma decisão mais fundamentada, pelo que se impõe o limite de 8, mais do que este limite torna o jogo com o algoritmo Minimax demasiado lento e, para efeitos de comparação, o AlphaBeta tem que ter o mesmo limite de profundidade. Convém ainda referir que se for o robot a começar a jogar a coluna escolhida é sempre a do meio, já que não se justifica realizar uma pesquisa num tabuleiro vazio de todas as vezes que se inicia um jogo e é sabido que a coluna do meio tem mais opções, teoricamente, de ganhar, pelo que é um bom começo.

Sobre a implementação do AlphaBeta, esta é muito parecida com o Minimax, a diferença é que requer mais 2 parâmetros das funções recursivas que guardam o valor da melhor jogada registada até ao momento, alfa para o jogador "X" e beta para "O". Assim, sempre que passamos por um nó em que o valor estimado é inferior a alpha, estando no nível Min, ou superior a beta, no nível Max, abandonamos a exploração desse ramo, retornando como valor do ramo esse mesmo em questão. O facto de a verdadeira utilidade daquele ramo é ou não o valor que retornamos é indiferente, já que um jogador que jogue de forma ótima não escolheria aquele ramo, pelo que o valor não fará qualquer diferença no resultado final. Uma prova que de facto esse corte não interfere no resultado final é a escolha das jogadas de "X", tendo as mesmas jogadas de "O", ser igual às que o Minimax seleciona.

Por fim, o MCTS é um algoritmo mais trabalhoso e complexo, mas que pode ser divido em essencialmente 4 etapas, fora a escolha final da melhor jogada. Começando sempre na raiz, selecionamos um nó folha, e por folha entenda-se o último nó daquele ramo que faz parte da árvore de decisão construída até ao momento, de acordo com a política de seleção referida anteriormente, mas dando prioridade aos nós que nunca foram expandidos. Isto é, sempre que um nó do caminho, que a política de seleção nos permite destacar, tem um filho que ainda não foi explorado, então determina-se esse como próximo nó a expandir. Quando todos os filhos daquele nó tiverem sido explorados, então aplicamos a política de seleção, escolhendo o que tem um UCB

maior. Se eventualmente ainda não estamos numa folha da árvore, então fazemos uma chamada recursiva da função select para o nó selecionado até ao momento, retornando precisamente uma folha.

Posto isto, expandimos um filho aleatório desse nó selecionado e prosseguimos para a simulação, um processo que gera sucessores aleatórios de um nó, simulando um jogo aleatório, até chegar a um nó terminal. De seguida retorna 1 se o jogo terminou em vitória para o "X" e 0 caso contrário. Este resultado é retornado para trás, ou seja, atualiza-se o número de visitas e vitórias do caminho da raiz até ao nó expandido, já que esteve envolvido nesta simulação, adicionando-se este nó expandido á árvore guardada em memória. Este processo repete-se 4000 vezes, de modo a ter estatísticas confiáveis para poder tomar uma decisão.

Importa referir que a estrutura de dados escolhida para guardar a árvore foi um array do objeto MCnode, impondo uma capacidade segura o suficiente para não causar exceções/erros durante a execução. Para isso, optamos por um limite igual ao dobro das iterações realizadas.

Além disso, o cálculo do UCB é de extrema importância para a eficácia do método, principalmente a escolha da constante c, uma vez que esta vai permitir balancear a importância que se atribui ao fator de exploration, responsável por explorar nós muito pouco visitados, e ao fator exploitation, responsável por explorar nós com uma win rate atrativa.

Uma vez realizadas todas as iterações do processo descrito acima, retornámos o sucessor da raiz com mais visitas, o que faz sentido já que o ramo que foi mais vezes expandido foi o que foi considerado como promissor mais vezes, mesmo sabendo que é preciso dar uma certa importância aos nós que foram visitados muito poucas vezes durante o algoritmo, após um certo número de iterações já se tem confiança o suficiente para descartar certos ramos da árvore por não serem boas opções.

DADOS DE EXEMPLOS DE JOGOS

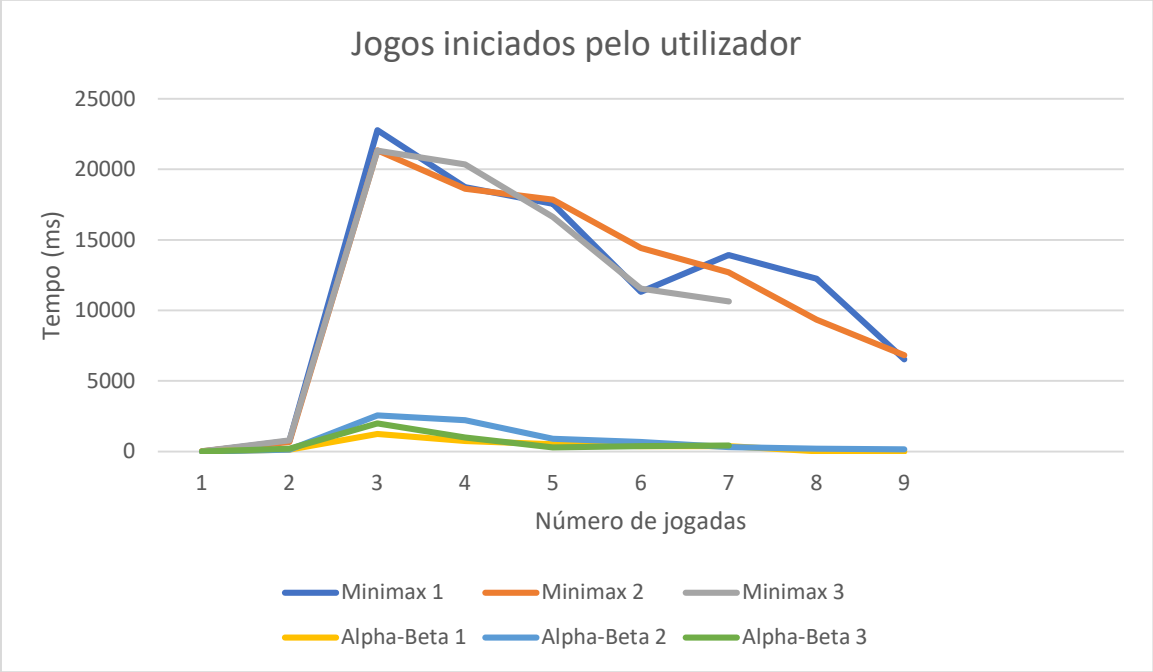

Figura 1 Tempo de execução de exemplos usando os algoritmos Minmax e AlphaBeta

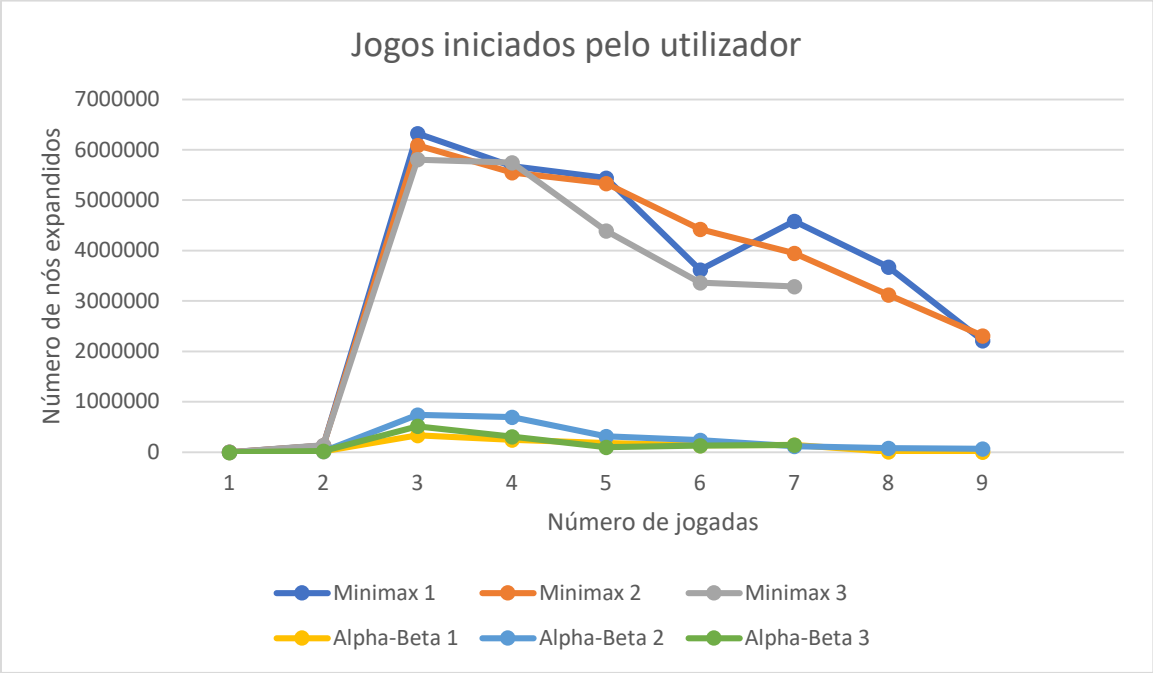

Figura 2 Número de nós expandidos de exemplos usando os algoritmos Minmax e AlphaBeta

Ana Catarina da Silva Costa Gomes (up201804545)
Cláudia da Costa Maia (up201905492)
Maria Santos Sobral (up201906946)

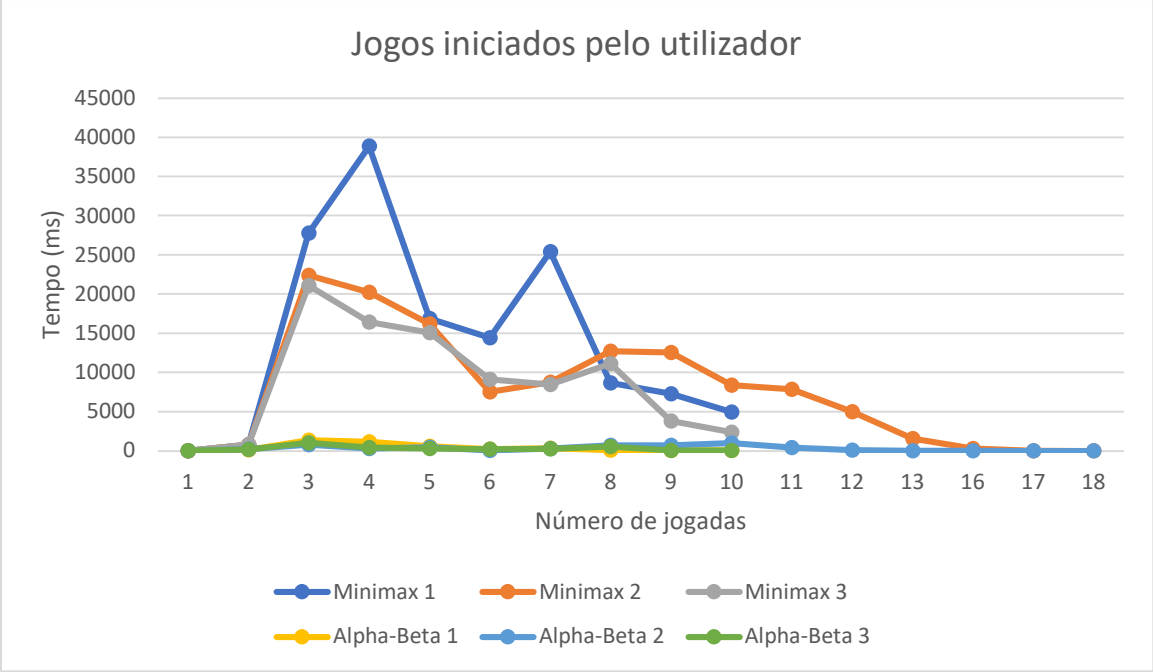

Figura 3 Tempo de execução de exemplos usando os algoritmos Minmax e AlphaBeta

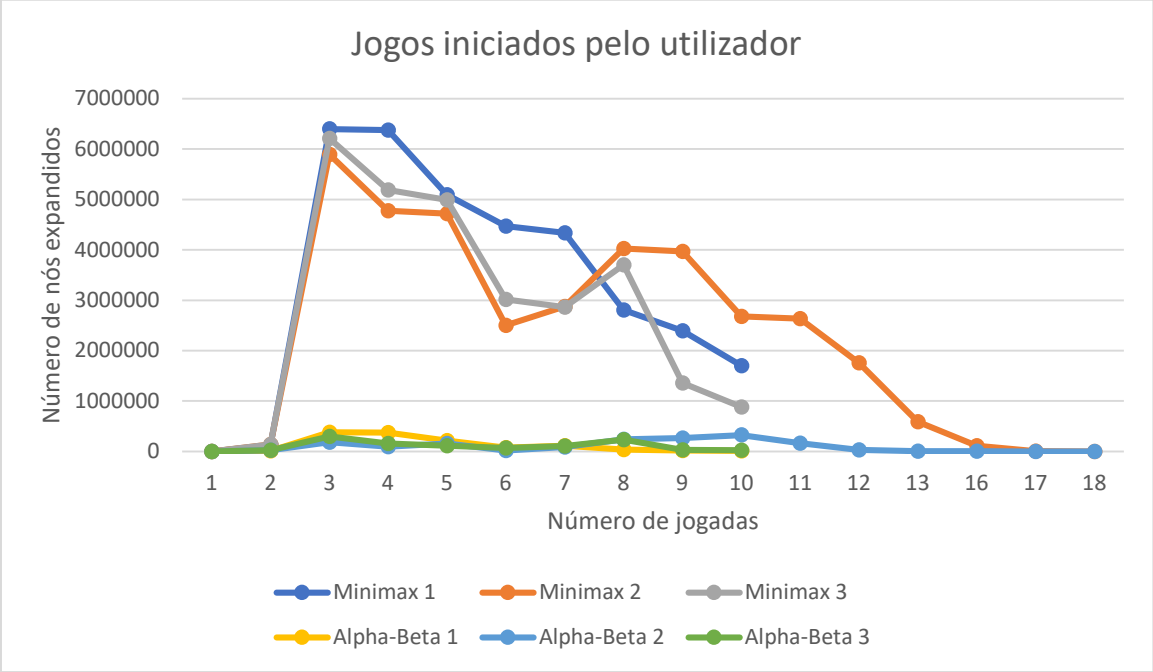

Figura 4 Número de nós expandidos de exemplos usando os algoritmos Minmax e AlphaBeta

Ana Catarina da Silva Costa Gomes (up201804545)
Cláudia da Costa Maia (up201905492)
Maria Santos Sobral (up201906946)

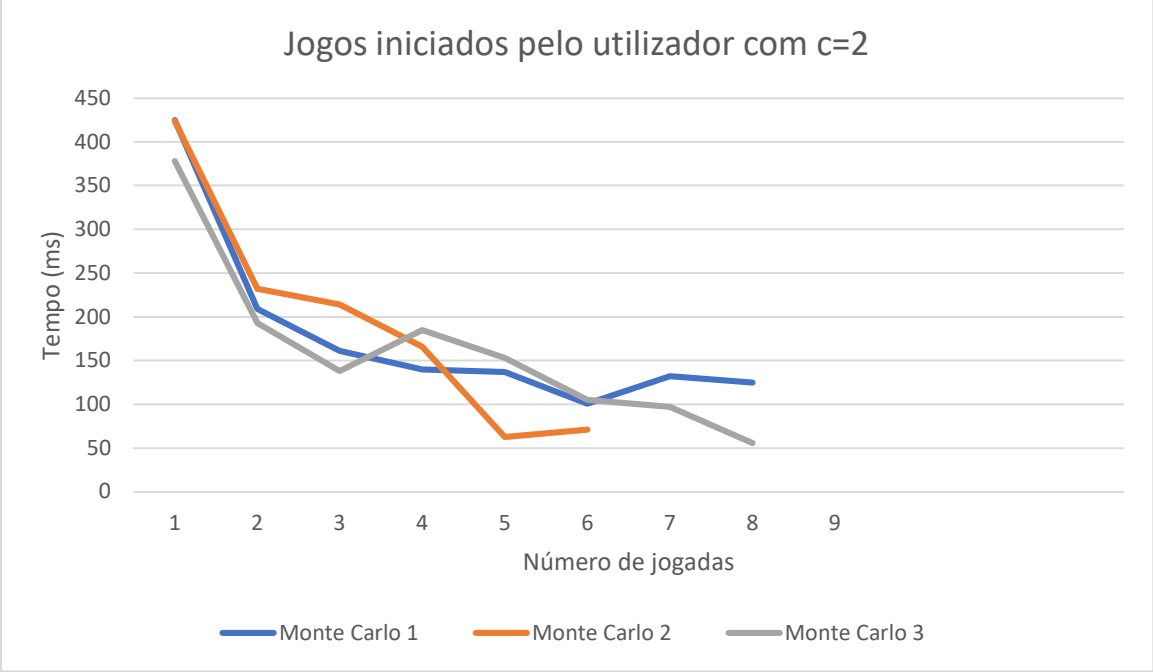

Figura 5 Tempo de execução de exemplos usando o algoritmo MCTS

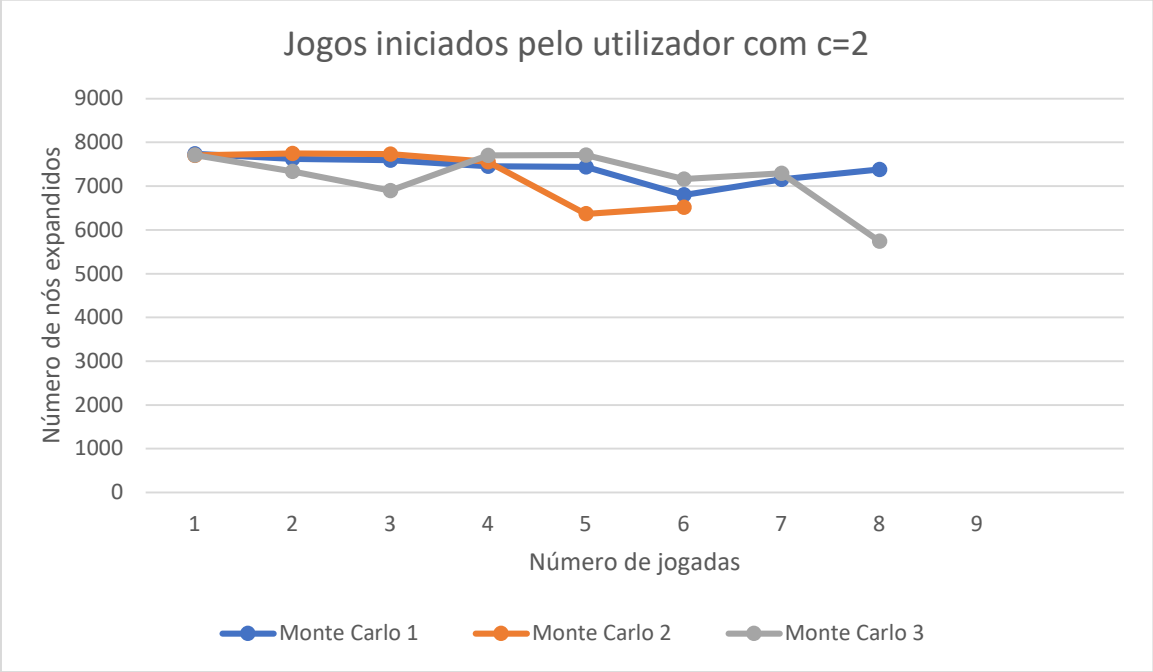

Figura 6 Número de nós expandidos de exemplos usando o algoritmo MCTS

Ana Catarina da Silva Costa Gomes (up201804545)
Cláudia da Costa Maia (up201905492)
Maria Santos Sobral (up201906946)

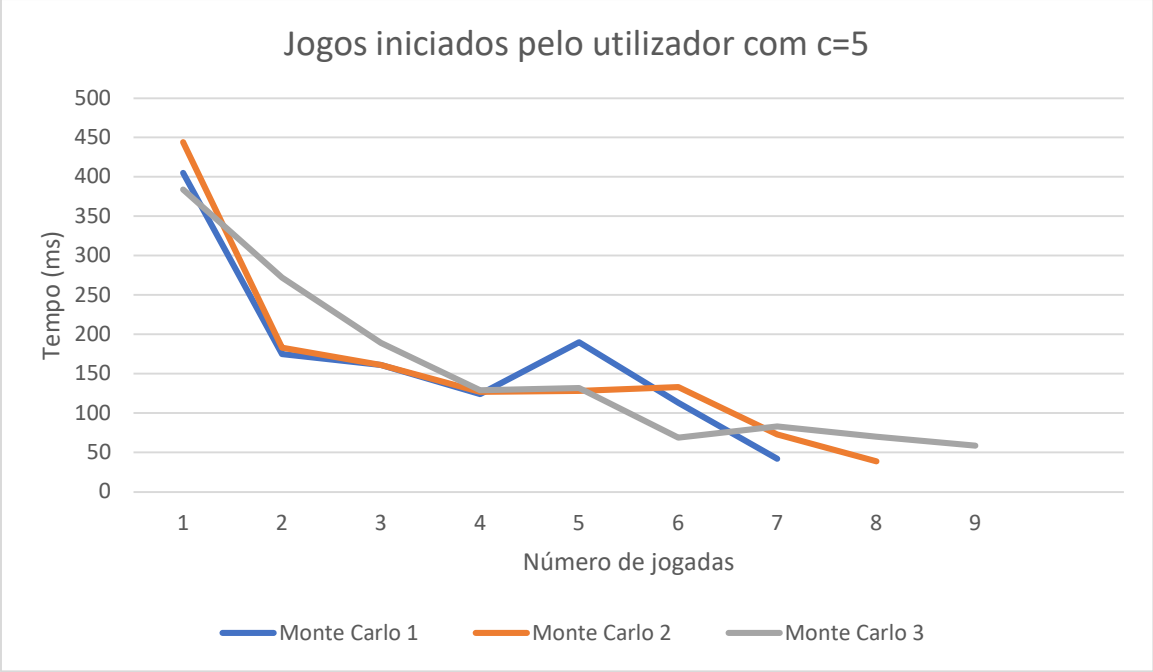

Figura 7 Tempo de execução de exemplos usando o algoritmo MCTS

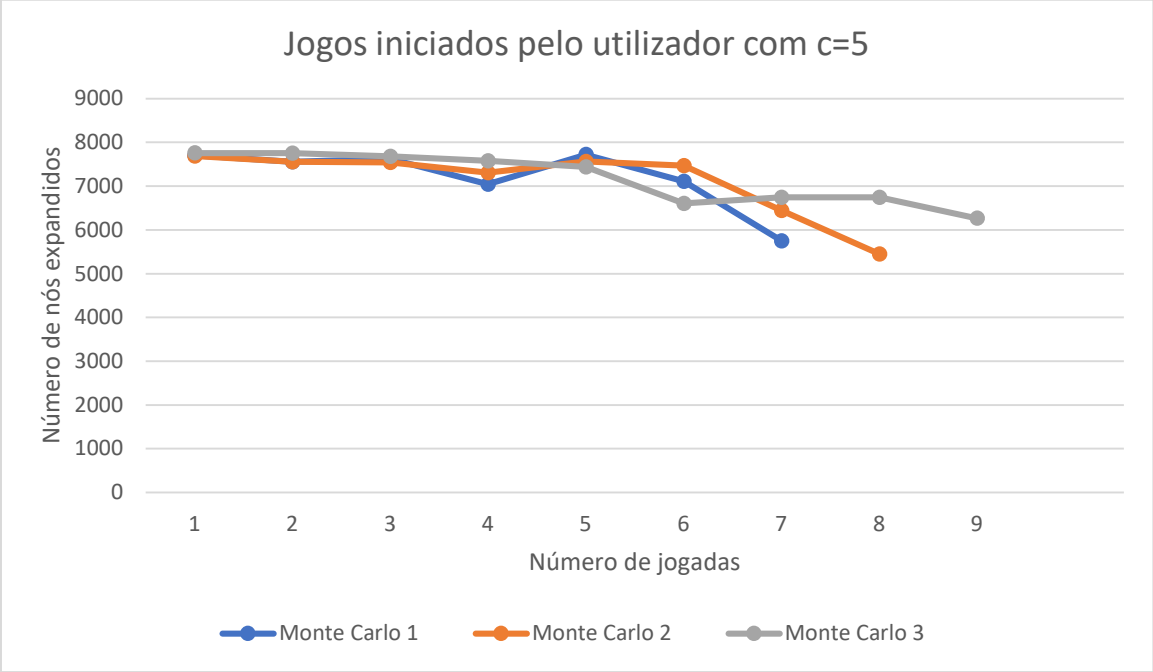

Figura 8 Número de nós expandidos de exemplos usando o algoritmo MCTS

Ana Catarina da Silva Costa Gomes (up201804545)
Cláudia da Costa Maia (up201905492)
Maria Santos Sobral (up201906946)

CONCLUSÃO

Com base na análise dos gráficos das figuras 1 a 8, verificamos que tanto em termos de tempo de execução, como em termos de memória gasta, o melhor algoritmo é o MCTS (com c=2 ou c=5). Este algoritmo para além disto, é também o mais eficiente uma vez que é o que prevê melhor as jogadas do oponente fazendo as decisões mais acertadas, sendo que como referido anteriormente, o MinMax e o AlphaBeta fazem sempre as mesmas decisões, que por sua vez dependem da heurística utilizada, que apesar de termos a garantia que funciona, não é propriamente eficiente. Na teoria, ou seja, dispondo de recursos ilimitados ou suficientes para percorrer toda a árvore de decisão, o AlphaBeta seria o algoritmo mais assertivo, permitindo poupar tempo, enquanto escolheria a melhor jogada possível em todo o momento.

Contudo, notamos uma diferença muito significativa entre os algoritmos MinMax e AlphaBeta, uma vez que este último, como utiliza a técnica do corte, elimina logo uma grande quantidade de nós desnecessários, não perdendo assim tempo nem espaço com eles. Ao contrário do MinMax, como podemos verificar nas figuras 1 a 4 onde as curvas de ambos os algoritmos são muito distintas, sendo que as do MinMax estão sempre por cima.

No seguimento, verificamos ainda que o algoritmo MCTS é significativamente mais assertivo com c=5, visto que prevê melhor o pensamento do oponente, no sentido em que ele não joga só para ganhar, mas também impede que o oponente ganhe, isto porque à semelhança do simulated annealing, este c permite que a busca não pare num máximo local e tente chegar ao máximo global. Com c=2, a exploration acaba por se sobrepor à exploitation, logo o computador dá mais valor a ganhar mais rápido, não explorando outros caminhos, já que quanto menor for o c mais o algoritmo se assemelha ao hill climbing. Assim sendo, é notório que 5 é de facto uma melhor escolha para a constante c. Para além disso, com base nos dados verificamos que à medida que se vai jogando, o tempo de execução do algoritmo é decrescente, e o número de nos expandidos é em média constante, mas mesmo no final tende também a decrescer.

Concluindo, o melhor algoritmo é de facto o MCTS, já que para além de ter complexidades temporal e espacial menores, é o que toma melhor decisões, sendo que a constante c que encontramos melhor é 5.

Ana Catarina da Silva Costa Gomes (up201804545)
Cláudia da Costa Maia (up201905492)
Maria Santos Sobral (up201906946)

REFERÊNCIAS BIBLIOGRÁFICAS

• Slides da cadeira Inteligência Artificial (CC2006) do ano letivo 2021/2022

• Artificial Intelligence: a Modern Approach, by Stuart Russell and Peter Norvig, 3rd edition, Prentice Hall

• https://www.analyticsvidhya.com/blog/2019/01/monte-carlo-tree-search-introduction-algorithm-deepmind-alphago/ (1)

• https://en.wikipedia.org/wiki/Monte_Carlo_tree_search

• https://mcts.netlify.app/

Ana Catarina da Silva Costa Gomes (up201804545)
Cláudia da Costa Maia (up201905492)
Maria Santos Sobral (up201906946)