

## Jogo dos 4 em Linha Relatório Final



**Unidade Curricular:** Inteligência Artificial

**Regentes:** Inês Dutra

Francesco Renna

**Elementos:** Beatriz Marques de Sá - up202105831  
Marisa Peniche Salvador Azevedo - up202108624  
Marta Luísa Monteiro Pereira - up202105713

23 de abril de 2023

## Índice

1. Introdução	3
1.1. O que são algoritmos para jogos?	3
1.2. Quais são os métodos utilizados para resolver algoritmos para jogos?	3
2. Descrição do jogo estudado (4 em linha)	4
3. Algoritmos	5
3.1. Minimax	5
3.2. Alpha-Beta Pruning	6
3.3. Monte Carlo Tree Search	8
4. Heurística	10
5. Descrição da Implementação	11
5.1. Escolha da linguagem de programação utilizada	11
5.2. Estruturas de dados utilizadas	11
5.1. Estruturação do código	12
6. Resultados	14
7. Comentários Finais e Conclusões	15
8. Referências Bibliográficas	16

## Índice de Figuras

Figura 1 - Árvore Minimax	6
Figura 2 - Árvore Alphabet	7
Figura 3 - Árvore Monte Carlo	9
Figura 4 - Gráfico: número de nós gerados	14
Figura 5 - Gráfico: tempo de execução	14

## 1. Introdução

### 1.1. O que são algoritmos para jogos?

Algoritmos para jogos são conjuntos de instruções lógicas usados para definir as regras e a lógica de jogos. Eles são usados para determinar como os personagens se movem, como os objetos interagem e como o jogo responde às ações do jogador.

A inteligência artificial desempenha um papel importante no desenvolvimento de algoritmos para jogos. Ela é usada para criar personagens não jogáveis (NPCs) que agem de forma realista e para criar sistemas de jogo que se adaptam ao comportamento do jogador.

Um exemplo de algoritmo para jogos é o utilizado no jogo dos 4 em linha. O algoritmo é responsável por determinar a melhor jogada para um jogador em qualquer situação. Ele usa uma combinação de heurísticas e técnicas de busca para avaliar o estado atual do jogo e prever o resultado das possíveis jogadas. O objetivo é maximizar as chances de vencer o jogo, considerando as jogadas possíveis do adversário.

### 1.2. Quais são os métodos utilizados para resolver algoritmos para jogos?

Existem diversos métodos utilizados para resolver algoritmos em jogos eletrônicos. Alguns exemplos desses métodos incluem algoritmos de busca, aprendizado por reforço, algoritmos genéticos e redes neurais.

Os algoritmos de busca são responsáveis por explorar todas as possibilidades de jogada em um jogo e encontrar a melhor solução, como o algoritmo Minimax, utilizado em jogos de tabuleiro. A aprendizagem por reforço é um método que ensina um agente de inteligência artificial a jogar através de tentativa e erro, recompensando o agente por ações corretas e penalizando por ações erradas até que ele aprenda a jogar de forma eficiente.

Os algoritmos genéticos simulam a evolução biológica para resolver problemas complexos, como a criação de personagens não jogáveis (NPCs) realistas. Eles geram diversas soluções aleatórias e as combinam para gerar novas soluções cada vez melhores. Já as redes neurais são redes de neurônios artificiais que aprendem com dados de treinamento para resolver problemas complexos, como a criação de NPCs realistas ou a identificação de padrões no comportamento do jogador.

É importante lembrar que muitos jogos requerem a combinação de múltiplos métodos para resolver seus algoritmos de forma eficiente.

## **2. Descrição do jogo estudado (4 em linha)**

O jogo 4 em linha é um jogo de estratégia em que dois jogadores jogam num tabuleiro vertical com sete colunas e seis linhas horizontais. Cada jogador tem uma cor que representa todas as suas peças e, por sua vez, coloca uma peça em uma das colunas, permitindo que ela caia até a posição mais baixa possível. O objetivo é conectar quatro peças da mesma cor numa linha horizontal, vertical ou diagonal. Se o jogador conseguir fazer uma linha de quatro peças, ele vence o jogo. Se todas as colunas estiverem preenchidas e nenhum jogador conseguir fazer uma linha de quatro peças, o jogo termina e é empate.

Apesar de ser fácil de aprender, o jogo oferece muitas possibilidades de jogadas e estratégias, permitindo que os jogadores bloqueiem o avanço do oponente ou criem armadilhas para induzir o adversário a cometer um erro. Como o jogo se baseia em estratégia e previsão, é comum que seja utilizado como objeto de estudo em inteligência artificial e em pesquisas sobre jogos.

### 3. Algoritmos

#### 3.1. Minimax

O algoritmo MiniMax é um algoritmo de tomada de decisão usado em jogos de dois jogadores com informações perfeitas, como xadrez, damas ou jogos de tabuleiro similares. Este algoritmo funciona avaliando todas as possíveis jogadas que cada jogador pode fazer e escolhendo a melhor jogada para um jogador, supondo que o adversário sempre fará a jogada que minimiza a pontuação do primeiro jogador. O objetivo do MiniMax é minimizar as perdas máximas possíveis.

O algoritmo MiniMax funciona através da construção de uma árvore de decisão, em que cada nó representa uma jogada possível num determinado estado do jogo. Os nós da árvore alternam entre representar os movimentos possíveis do jogador atual e do adversário. A árvore é percorrida em profundidade, usando uma busca recursiva que retorna a pontuação esperada de um determinado nó.

Para relacionar o algoritmo MiniMax com o jogo dos 4 em linha, o primeiro passo é representar o estado atual do jogo como um nó da árvore. Em seguida, são gerados todos os possíveis movimentos que o jogador atual pode fazer, criando novos nós na árvore. Para cada um desses novos nós, é então gerada a árvore de todos os movimentos possíveis que o adversário pode fazer.

A pontuação final de cada nó é então determinada usando uma função de avaliação que atribui uma pontuação a um estado do jogo com base em quão favorável é para o jogador atual. A pontuação é propagada de volta pela árvore de decisão, de modo a que cada nó pai recebe a pontuação mínima ou máxima dos seus filhos, dependendo se o nó representa o jogador atual ou o adversário.

Por fim, o algoritmo MiniMax seleciona o movimento que leva ao nó com a maior pontuação possível para o jogador atual. Este movimento é então realizado, e o processo é repetido para o próximo estado do jogo.

Em resumo, o algoritmo MiniMax funciona encontrando o melhor movimento para o jogador atual, considerando todas as possíveis jogadas do adversário, e escolhendo

a jogada que minimiza a perda máxima possível. No jogo dos 4 em linha, isso significa avaliar todos os possíveis movimentos de colocação de peças e escolher o que minimiza a probabilidade do adversário vencer.

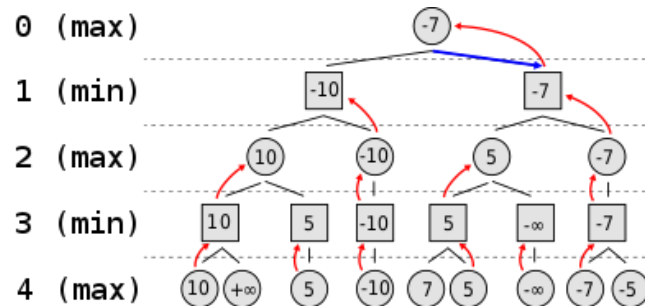


Figura 1 - Árvore MiniMax

Complexidade Temporal:  $O(b^d)$

Complexidade Espacial:  $O(b \times d)$

onde:

$b \rightarrow$  possíveis jogadas em cada ponto

$d \rightarrow$  profundidade máxima da árvore

Esta estratégia é completa e ótima.

### 3.2. Alpha-Beta Pruning

O algoritmo Alpha Beta Pruning tem como objetivo reduzir o número de nós que precisam de ser avaliados durante a busca numa árvore de jogo, sem comprometer a qualidade da jogada selecionada. Este algoritmo é uma otimização do Minimax, que avalia as possíveis jogadas de ambos os jogadores num jogo. No entanto, o Minimax pode ser computacionalmente custoso e demorar muito tempo para ser executado em jogos com muitas possibilidades, como o jogo dos 4 em linha.

O Alpha Beta Pruning resolve esse problema ao “podar” ramos da árvore de busca que já foram avaliados e que não resultarão numa jogada melhor. Isso é feito comparando o valor da jogada atual com o valor da jogada anterior e descartando as jogadas que não irão melhorar a posição atual do jogador. Em outras palavras, se numa ramificação da árvore de busca o algoritmo perceber que não há nenhuma

jogada melhor para o jogador humano do que a jogada atual, ele não vai continuar a análise dessa ramificação. O mesmo acontece para as jogadas do computador.

Dessa forma, o algoritmo Alpha Beta Pruning reduz significativamente o número de ramos da árvore que precisam ser avaliados e, conseqüentemente, o tempo de execução necessário para encontrar a jogada ideal. Isso torna o jogo mais eficiente em termos de tempo de processamento.

O valor de "alpha" e "beta" são parâmetros usados no algoritmo Alpha Beta Pruning para fazer as comparações e determinar quais os ramos da árvore de busca que serão podados. "Alpha" representa o valor máximo atualmente encontrado para o jogador humano, enquanto "beta" representa o valor mínimo atualmente encontrado para o jogador computador. Durante a busca, os valores de "alpha" e "beta" são atualizados à medida que novos nós são avaliados. Se o valor de um nó for maior que o valor de "alpha", significa que é uma jogada melhor para o jogador humano e "alpha" é atualizado com esse novo valor. Se o valor de um nó for menor que o valor de "beta", significa que é uma jogada pior para o jogador computador e "beta" é atualizado com esse novo valor.

Esses valores de "alpha" e "beta" são usados para fazer a poda dos ramos da árvore de busca. Se o valor de "alpha" for maior ou igual ao valor de "beta" em um determinado nó, significa que não há mais necessidade de continuar a avaliar os ramos dessa parte da árvore, pois não afetarão a decisão final. Essa poda permite reduzir ainda mais o número de nós que precisam ser avaliados, acelerando o processo de busca e tornando o algoritmo ainda mais eficiente.

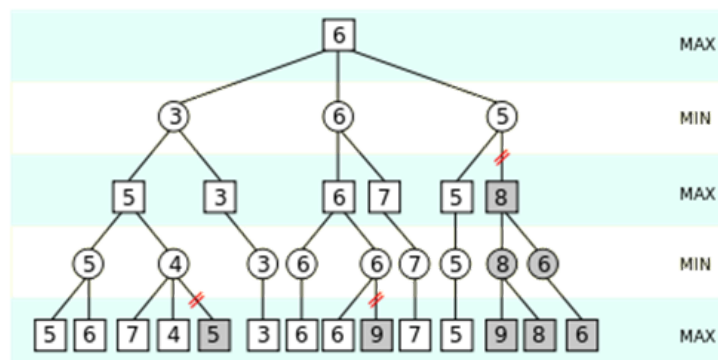


Figura 2 - Árvore Alphabeta



Complexidade Temporal:  $O(b^{\frac{d}{2}})$

Complexidade Espacial:  $O(b \times d)$

onde:

$b \rightarrow$  possíveis jogadas em cada ponto

$d \rightarrow$  profundidade máxima da árvore

### 3.3. Monte Carlo Tree Search

O Monte Carlo Tree Search (MCTS) é um algoritmo de busca usado em Inteligência Artificial (IA) que se baseia em simulações de jogadas aleatórias para estimar o valor de diferentes ações em um determinado estado do jogo.

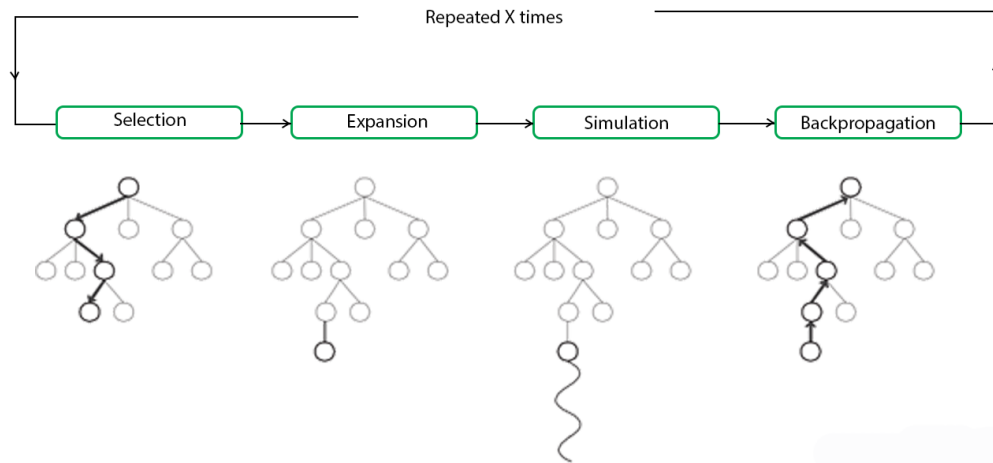
O MCTS é geralmente usado em jogos de estratégia, onde o espaço de busca é muito grande para ser explorado completamente. O objetivo é encontrar a melhor ação para realizar num estado de jogo específico, considerando as possíveis ações subsequentes e as recompensas associadas a essas ações.

O algoritmo MCTS começa com uma árvore de busca vazia, onde cada nó representa um estado do jogo. A partir do estado atual do jogo, o MCTS expande a árvore de busca gerando novos nós para representar as possíveis ações que podem ser tomadas. Em seguida, realiza simulações de jogos aleatórias (rollout) a partir desses nós expandidos, jogando até ao fim do jogo ou até a um limite de profundidade pré-definido.

Após as simulações de jogos aleatórios, o MCTS obtém informações sobre as recompensas de cada simulação e atualiza os valores dos nós na árvore de busca, de forma a atribuir um valor estimado a cada ação. Essa atualização é realizada usando uma estratégia de backpropagation, onde as recompensas são propagadas dos nós folha até a raiz da árvore.

O MCTS também usa uma política de seleção de nós chamada UCT (Upper Confidence Bound for Trees), que leva em consideração a estimativa de valor de um nó e o número de vezes que o nó foi visitado para determinar a prioridade com que os nós devem ser explorados.

O MCTS continua a iterar entre a expansão da árvore de busca, a realização de simulações de jogadas aleatórias e a atualização dos valores dos nós até que o critério de paragem seja atingido, neste caso o limite de tempo.



*Figura 3 - Árvore Monte Carlo*

Upper Confidence Bounds for Trees:

$$UCBI = \overline{X}_j + C \sqrt{\frac{2 \ln(n)}{n_j}}$$

onde:

$\overline{X}_j$ : recompensa estimada da escolha  $j$

$n$ : número de vezes que o pai já foi visitado

$n_j$ : número de vezes que a escolha  $j$  foi feita

*Exploitation* (primeira parcela da soma):

- enfatiza a recompensa
- torna a busca mais guiada

*Exploration* (segunda parcela da soma):


- reforça a exploração de nós menos frequentemente visitados
- reduz o efeito de “rollouts” com pouca sorte
- constante  $C$  equilibra *Exploitation* e *Exploration*

## 4. Heurística

A heurística utilizada para o jogo dos 4 em linha foi a utilidade de cada tabuleiro.

Tendo como base a seguinte tabela:

Pontuação	Nº de X's	Nº de O's
-50	0	3
-10	0	2
-1	0	1
0	0	0
1	1	0
10	2	0

 além disso, ter X's e O's misturados resulta na pontuação 0, pois impossibilita o X ou O de ganhar através daquele subconjunto.

Fizemos a implementação da utilidade da seguinte forma:

Para calcular a utilidade de cada tabuleiro, procedemos à análise de todos os subconjuntos de 4 elementos presentes no tabuleiro. Para cada subconjunto, verificamos a pontuação correspondente atribuída a esse conjunto específico. Após analisar todos os subconjuntos, somamos todas as pontuações obtidas para obter a utilidade total do tabuleiro.

Esta análise é realizada de forma minuciosa, examinando cada grupo de 4 elementos adjacentes ou sobrepostos no tabuleiro. Para cada subconjunto, é atribuída uma pontuação com base nos critérios previamente estabelecidos.

A utilidade do tabuleiro, pode ser usada num algoritmo minimax ou num algoritmo alphabeta para encontrar a jogada mais vantajosa para o jogador computador.

## **5. Descrição da Implementação**

### **5.1. Escolha da linguagem de programação utilizada**

Para este trabalho, optamos por utilizar a linguagem Python, pois esta é uma linguagem de programação de código aberto, orientada a objetos e de alto nível. Assim, podemos usá-la sem nenhum custo adicional e temos acesso ao seu código fonte para personalizá-la de acordo com as nossas necessidades.

Para além disso, Python é uma linguagem bastante popular com uma ampla disponibilidade de bibliotecas que nos pode ajudar na análise e no processamento dos dados recolhidos.

Outra vantagem é existir uma grande comunidade de desenvolvedores nesta linguagem de programação, o que significa que podemos encontrar facilmente suporte e documentação para resolver problemas e dúvidas que possam surgir durante o desenvolvimento do projeto.

Em suma, consideramos que a escolha de Python para a implementação dos algoritmos é uma escolha sólida.

### **5.2. Estruturas de dados utilizadas**

A estrutura de dados utilizada para o jogo 4 em Linha foi listas.

Cada coluna do tabuleiro foi representada por uma lista, e todas essas listas de colunas foram agrupadas em outra lista para formar uma matriz. Além disso, também utilizamos uma lista para armazenar todos os possíveis próximos movimentos a partir de um tabuleiro específico.

### 5.1. Estruturação do código

Para a estruturação do código decidimos dividir o mesmo em vários ficheiros:

- **board.py** define a classe Board, que representa um tabuleiro do jogo. Alguns dos métodos disponíveis são:
  - **\_\_init\_\_**: Método construtor da classe Board que inicializa o tabuleiro e outras variáveis de jogo, como a peça selecionada, o número de movimentos, a cor e o rótulo do jogador vencedor, e a visita do tabuleiro.
  - **set\_turn**: Alterna o turno entre os dois jogadores.
  - **count\_pieces**: Conta o número de peças numa coluna específica do tabuleiro.
  - **checkWin**: Verifica se o jogador venceu o jogo, analisando as combinações vencedoras no tabuleiro.
  - **utility**: Calcula a pontuação de utilidade do tabuleiro para o jogador atual, com base nas combinações de peças no tabuleiro.
  - **draw**: Desenha o tabuleiro na tela do jogo usando a biblioteca Pygame.
  - **printBoard**: Imprime o tabuleiro na consola
  - **successors**: Gera e retorna uma lista de sucessores do estado atual do tabuleiro de jogo. Cada sucessor é representado como um par de coordenadas (coluna, novo tabuleiro).
  - **check\_subset\_pontuation**: verifica a pontuação de cada um dos subconjunto de peças numa linha, coluna ou diagonal do tabuleiro.
- **strategies.py** implementação dos algoritmos:
  - Minimax
  - AlphaBeta
  - Monte Carlo
- **interface.py** interface gráfica do jogo utilizando a biblioteca Pygame. Algumas das funções que estão neste ficheiro são:
  - **mouse\_motion**: lê o movimento do mouse durante o jogo e desenha um círculo na posição em que o mesmo está, indicando assim onde uma peça seria colocada.

- **humam\_output**: trata do movimento do jogador humano e atualiza o tabuleiro e a interface gráfica de acordo.
- **main**: função principal do programa, que inicia o jogo e permite que os jogadores escolham o modo de jogo e a estratégia do computador.
- **constant.py** define as constantes utilizadas na interface gráfica do jogo:
  - **ROWS**: Define o número de linhas do tabuleiro.
  - **COLS**: Define o número de colunas do tabuleiro.
  - **SQUARE\_SIZE**: Define o tamanho de cada quadrado no tabuleiro.
  - **WIDTH**: Define a largura total do tabuleiro, multiplicando o número de colunas pelo tamanho de cada quadrado.
  - **HEIGHT**: Define a altura total do tabuleiro, multiplicando o número de linhas mais uma unidade pelo tamanho de cada quadrado.
  - **RADIUS**: Define o raio de cada peça do jogo, como metade do tamanho de cada quadrado, subtraído por 5 unidades.
  - **COMPUTER\_PIECE**: Define o símbolo utilizado para representar as peças do computador no jogo.
  - **PLAYER\_PIECE**: Define o símbolo utilizado para representar as peças do jogador no jogo.
  - **MAX\_DEPTH**: Define a profundidade máxima para a busca do melhor movimento pelo computador no jogo.
  - **TIME**: Define um limite de tempo para a tomada de decisão do computador no jogo.
  - **C**: Define uma constante utilizada em um algoritmo de busca utilizado no jogo.
  - **SNOW**: Cor utilizada para o fundo do jogo.
  - **STEELBLUE**: Cor utilizada para representar as peças do computador.
  - **DEEPSKYBLUE**: Cor utilizada para representar as peças do jogador.
  - **LIGHTSTEEL**: Cor cinza utilizada para o tabuleiro do jogo.

Para mais informações sobre a estruturação do código, deixamos aqui o link para o GitHub onde se encontra o repositório:

- <https://github.com/beatrizmsa/Connect4>

## 6. Resultados

profundidade	Minimax		Alphabeta	
	<u>nós gerados</u>	<u>tempo (s)</u>	<u>nós gerados</u>	<u>tempo (s)</u>
1	7	0.0010426	7	0.0012240
2	56	0.0120351	56	0.0141603
3	742	0.1698398	742	0.1366972
4	10298	2.1294042	9486	1.5295937
5	139679	28.508542	99493	14.237972

Nós Gerados

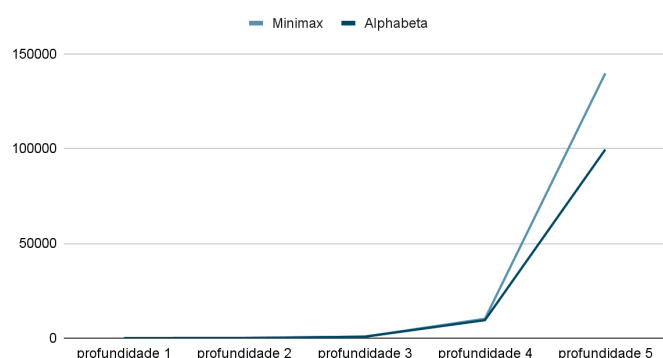


Figura 4 - Gráfico: número de nós gerados

Tempo de Execução

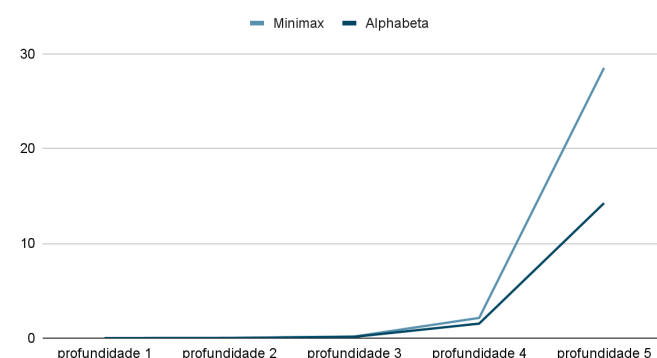


Figura 5 - Gráfico: tempo de execução

### Monte Carlo (MCTS):

- Nós gerados: N/A (o MCTS é um algoritmo de busca por tempo que não gera uma quantidade fixa de nós como o Minimax ou o Alphabeta)
- Tempo (s): N/A (o MCTS é um algoritmo baseado em amostragem e simulação, o tempo de execução pode variar dependendo do número de iterações definidas)

## 7. Comentários Finais e Conclusões

Com base nos resultados obtidos acima para os algoritmos de busca Minimax e Alphabeta no jogo dos 4 em linha, pode-se notar que à medida que a profundidade aumenta, o número de nós gerados e o tempo de execução também aumentam para ambos os algoritmos. Isso é esperado, uma vez que quanto maior a profundidade, maior é o espaço de busca a ser explorado.

Comparando os resultados entre os dois algoritmos, é possível observar que o Alphabeta gerou menos nós e teve tempos de execução menores em comparação com o Minimax em profundidades mais altas (profundidade 4 e 5). Isso é devido à poda alfa-beta, uma técnica utilizada no algoritmo que permite eliminar certos ramos da árvore de busca, reduzindo o número de nós a serem avaliados e, conseqüentemente, diminuindo o tempo de execução.

Em termos de tempo de execução, o Minimax foi mais rápido em profundidades mais baixas (profundidade 1, 2 e 3), enquanto o Alphabeta foi mais rápido em profundidades mais altas (profundidade 4 e 5). Isso indica que o Alphabeta é mais eficiente em termos de tempo de execução em profundidades mais profundas em comparação com o Minimax.

Uma conclusão para o algoritmo MCTS é que pode ser uma abordagem eficaz para jogos complexos, como o jogo das 4 em linha, especialmente quando não é possível ou prático explorar todas as possíveis jogadas em uma árvore de busca

No entanto, o desempenho do MCTS pode variar dependendo do número de iterações definidas para as simulações e do equilíbrio entre exploração e exploração do espaço de busca. É importante ajustar adequadamente os parâmetros do algoritmo MCTS para obter um bom desempenho em um jogo específico.



## 8. Referências Bibliográficas

### WebGrafia:

[https://drive.google.com/file/d/1aL24PZ3uHJ9ASO\\_Z6PoG1KiwxvtAgiFD/view?usp=share\\_link](https://drive.google.com/file/d/1aL24PZ3uHJ9ASO_Z6PoG1KiwxvtAgiFD/view?usp=share_link) (slides da aula teórica)

<https://www.organicadigital.com/blog/algorithmo-minimax-introducao-a-inteligencia-artificial/>

<https://www.hackerearth.com/blog/developers/minimax-algorithm-alpha-beta-pruning/>

<https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/>

[https://en.wikipedia.org/wiki/Alpha-beta\\_pruning](https://en.wikipedia.org/wiki/Alpha-beta_pruning)

<https://en.wikipedia.org/wiki/Minimax>

[https://en.wikipedia.org/wiki/Monte\\_Carlo\\_tree\\_search](https://en.wikipedia.org/wiki/Monte_Carlo_tree_search)

### BiblioGrafia:

Russell, S. J., & Norvig, P. (2010). Artificial intelligence: a modern approach.

Elaine Rich and Kevin Knight (2017). Artificial Intelligence.