Connect Four

Francisco Ribeiro, Matheus Bissacot, Sérgio Coelho

Universidade do Porto

Contents

1 Introdução

Os jogos com adversário são um campo importante da inteligência artificial que têm sido estudados desde a década de 1950.

Esses jogos envolvem dois jogadores que tomam decisões alternadas para alcançar um objetivo final. Um dos jogos mais conhecidos é o 4 em linha, um jogo de estratégia em que os jogadores tentam obter quatro peças da mesma cor em uma linha.

Figure 1: Jogo 4 em linha

Existem várias abordagens algorítmicas para resolver jogos adversariais. Neste relatório, analisamos e comparamos três algoritmos diferentes: Minimax, Alpha-Beta Pruning e Monte Carlo Tree Search (MCTS) aplicados ao jogo de 4 em linha.

2 Minimax

O algoritmo Minimax é um método de busca em árvore usado em jogos com adversários para encontrar a melhor jogada possível. Ele funciona simulando todas as possíveis jogadas futuras, alternando entre os jogadores e escolhendo a melhor jogada para o jogador atual.

Infelizmente devido á incapacidade dos computadores de hoje e á complexidade do jogo, o *minimax* para a implementação deste jogo, é limitado pela profundidade da árvore de busca, que pode ser definida por um limite

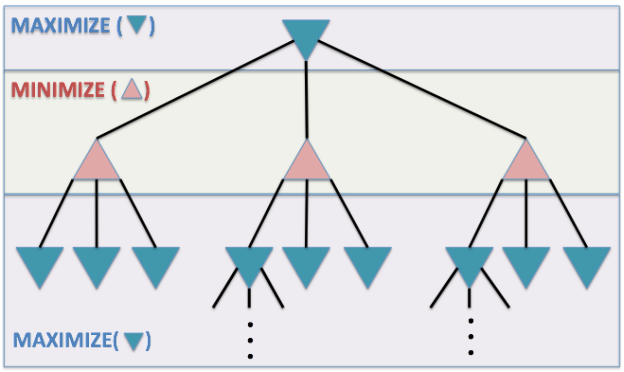


Figure 2: Árvore minimax

de tempo ou profundidade máxima, na nossa implementação utilizamos a última opção.

Como esta busca vai simulando estados futuros, ela tem em conta que o jogador atual joga sempre a melhor jogada, isso quer dizer que está a supor que o próximo jogador vai fazer uma jogado que vai minimazar ao máximo o jogador atual.

Na nossa implementação começamos por definir:

1. O caso base, que é quando chega a uma profundidade 0, aí retorna a pontuação atual;

2. Seguido disto, criamos dois *if clauses* para saber o jogador atual e saber se ele é o maximizante ou o minimizante

3. Depois em cada condição chamamos recursivamente a função e verificamos se a pontuação obtida é maior/menor do que o máximo/mínimo.

4. Quando o loop termina, o algoritmo retorna a pontuação do melhor movimento encontrado e a coluna correspondente.

Com isso em mente chegamos ao seguinte código:

```
def minimax(thisboard, depth, symbol):
    best_move=-1
    if depth==0 or game_over(thisboard, symbol):
        return utility(thisboard), None

    if symbol == 'X':
        maxEval = -math.inf
        #for each child
        for col in (legal_moves(thisboard)):
            row = move_selected(thisboard,symbol, col)
            eval = minimax(thisboard, depth-1, not_symbol(symbol))[0]
            undo_move(thisboard, col, row)
            if eval>maxEval:
                maxEval = eval
                best_move=col
        return maxEval,best_move

    else:
        minEval = math.inf
        #for each child
        for col in (legal_moves(thisboard)):
            row = move_selected(thisboard,symbol, col)
            eval = minimax(thisboard, depth-1, not_symbol(symbol))[0]
            undo_move(thisboard, col, row)
            if eval<minEval:
                minEval = eval
                best_move=col
        return minEval, best_move
```

3 Algoritmo Alpha-Beta Pruning

O Alpha-Beta Pruning é uma extensão do algoritmo Minimax que reduz o número de nós expandidos na árvore de busca.

Ele funciona eliminando ramos da árvore que não afetam o resultado final.

O algoritmo Alpha-Beta Pruning usa uma abordagem de poda para reduzir o número de nós que precisam ser avaliados. Isso o torna **mais eficiente** do que o algoritmo Minimax em termos de tempo de execução.

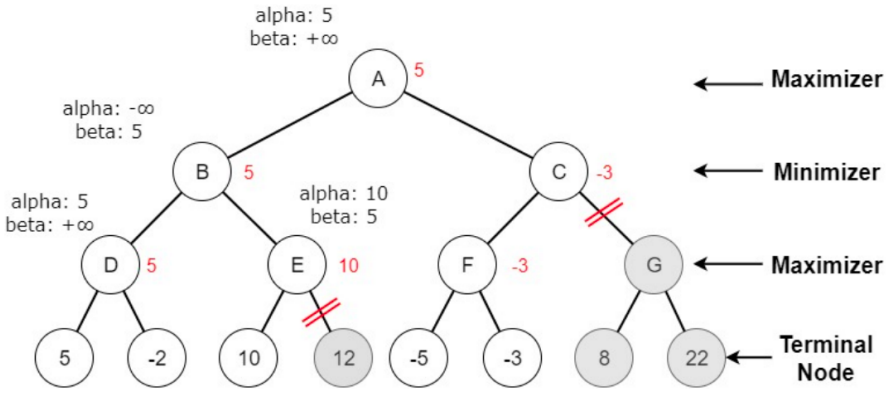


Figure 3: Árvore alphabeta

O algoritmo Alpha-Beta utiliza duas variáveis, alpha e beta, para podar ramos da árvore de jogadas que não precisam ser explorados.

Alpha representa a melhor pontuação encontrada até o momento para o jogador Max, e beta representa a melhor pontuação encontrada até o momento para o jogador Min.

A nossa implementação segue os seguintes passos:

1. O algoritmo começa verificando se atingiu a profundidade máxima de busca ou se o jogo acabou.

2. Em seguida, o algoritmo verifica qual jogador deve jogar (X ou O) e inicia um loop que percorre todas as colunas possíveis do tabuleiro.

3. Para cada coluna possível, o algoritmo realiza um movimento nessa coluna e chama a função recursivamente com um símbolo diferente e uma profundidade reduzida em 1.

4. A função recursiva retorna uma pontuação para o movimento atual e o algoritmo atualiza o valor máximo ou mínimo de acordo com o jogador atual.

5. O algoritmo também atualiza os valores alpha e beta para podar nós desnecessários da árvore.

6. O loop é interrompido quando $\beta \leq \alpha$.

7. Quando o loop termina, o algoritmo retorna a pontuação do melhor movimento encontrado e a coluna correspondente.

```
    def alphabeta(thisboard, depth,symbol , alpha, beta):
    best_move=None
    if depth==0 or game_over(thisboard, symbol):
        return utility(thisboard), None

    if symbol == 'X':
        maxEval = -math.inf
        #for each child
        for col in (legal_moves(thisboard)):
            row = move_selected(thisboard,symbol, col)
            eval = alphabeta(thisboard, depth-1, not_symbol(symbol), alpha,
    beta)[0]
            undo_move(thisboard, col, row)
            if eval>maxEval:
                maxEval = eval
                best_move=col
            alpha= max(alpha, maxEval)
            if beta<=alpha: break
        return maxEval,best_move

    else:
        minEval = math.inf
        #for each child
        for col in (legal_moves(thisboard)):
            row = move_selected(thisboard,symbol, col)
            eval = alphabeta(thisboard, depth-1, not_symbol(symbol), alpha,
    beta)[0]
            undo_move(thisboard, col, row)
            if eval<minEval:
                minEval = eval
                best_move=col
            beta= min(beta, minEval)
            if beta<=alpha: break
        return minEval, best_move
```

4 Monte Carlo Tree Search (MCTS)

O MCTS é uma abordagem mais recente para jogos adversariais que tem sido usada com sucesso em jogos de tabuleiro complexos, como o Go.

Ele funciona simulando várias jogadas aleatórias e avaliando seus resultados usando uma função de avaliação. Essa função é usada para construir uma árvore de busca que é usada para selecionar a próxima jogada. O MCTS

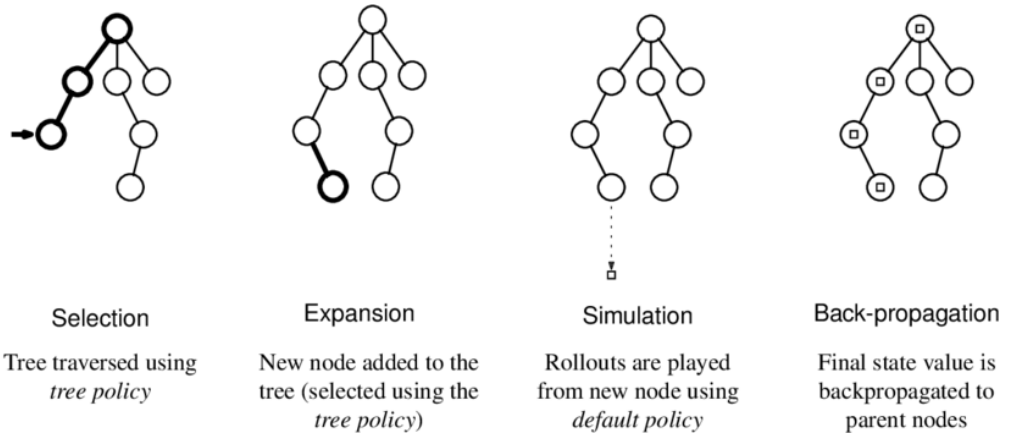


Figure 4: Processo de busca do mcts

pode ser mais eficiente do que o Minimax e o Alpha-Beta Pruning em jogos complexos, mas se o número de simulações for reduzido não será um bom algoritmo para utilizar.

A base do nosso algoritmo foi fundada em harrycodes-MCTS.

Antes de implementar o algoritmo em si, é importante implementarmos a estrutura de nós e de árvores:

Class Node Classe que contém os atirbutos de:

- **move** (o movimento que foi realizado)
- **parent** (o tabuleiro que originou o tabuleiro atual)
- **N** (número de simulações feitas)
- **Q** (número de vitórias)

- **children** (nós filhos)
- **add_children:** Adiciona os filhos;
- **UCB:** Calcula o Upper Confidence Bound. A fórmula exata é:

$$UCB(n_i) = \frac{Q}{N} + c_i\sqrt{\frac{\log(Np)}{N}} \tag{1}$$

Class MCTS Classe que implementa a árvore. Os seus atributos são:

- **root_state** (estado da raíz da árvore)
- **symbol** (símbolo atual da árvore)
- **root** (raíz da árvore)
- **run_time** (tempo para fazer as simulações)
- **node_count** (número de nós)
- **num_rollout** (número de simulações)
- **select_node:** Corresponde à primeira parte do processo do MCTS: a fase de Seleção. O código faz uma ramificação da árvore até encontrar a profundidade atual. A partir disso, é escolhido o nó com o melhor valor UCB. Se esse nó já foi expandido, é escolhido um nó aleatoriamente. Retorna o nó selecionado e o seu respetivo estado
- **expand:** Corresponde à segunda parte do processo do MCTS: a fase de Expansão. Se o nó atual é um nó de vitória, retorno False. Caso contrário, adiciono o nó para a árvore e retorno True. Ou seja, a função retorna se é possível a expansão ou não
- **roll_out:** Corresponde à terceira parte do processo do MCTS: a fase de Simulação. Nesta fase, é feito simulações de jogos até que o resultado final seja de vitória
- **back_propagate:** Corresponde à última parte do processo do MCTS: a fase da Retropropagação. Este código tem como objetivo propagar os vencedores do jogo simulado para o nó selecionado. Não há retorno e apenas calcula o valor de "recompensa" (reward).

- **search:** Função que junta todos os elementos acima e faz a pesquisa até o tempo limite acabar
- **best_move:** Função que seleciona o melhor movimento da árvore

```
#MCTS Algorithm (Trocar pela classe)
def mcts(table, symbol):
    class Node:
        def __init__(self, move, parent):
            self.move = move
            self.parent = parent
            self.N = 0
            self.Q = 0
            self.children = {}

        def add_children(self, children: dict) -> None:
            for child in children:
                self.children[child.move] = child

        #fun   o Upper Confidence Bound (UCB)
        def UCB(self):
            if self.N == 0:
                return 0 if math.sqrt(2) == 0 else float('inf')
            return self.Q / self.N + math.sqrt(2) * math.sqrt(math.log(self.
    parent.N) / self.N)


    class MCTS:
        def __init__(self, state, symbol):
            self.root_state = deepcopy(state)
            self.symbol = symbol
            self.root = Node(None, None)
            self.run_time = 0
            self.node_count = 0
            self.num_rollouts = 0

        def select_node(self) -> tuple:
            node = self.root
            state = deepcopy(self.root_state)

            while len(node.children) != 0:
                children = node.children.values()
                max_value = max(children, key=lambda n: n.UCB()).UCB()
                max_nodes = [n for n in children if n.UCB() == max_value]

                #choices = legal_moves(state)
                node = random.choice(max_nodes)
                move_selected(state, symbol ,node.move)

                if node.N == 0:
                    return node, state

            if self.expand(node, state):
                node = random.choice(list(node.children.values()))
                move_selected(state, self.symbol, node.move)

```

```
51              return node, state
52
53          def expand(self, parent: Node, state) -> bool:
54              if check(state, self.symbol):
55                  return False
56
57              children = [Node(move, parent) for move in legal_moves(state)]
58              parent.add_children(children)
59
60              return True
61
62          def roll_out(self, state):
63              while not check(state, self.symbol):
64                  move_selected(state, self.symbol ,random.choice(legal_moves(
    state)))
65                  self.node_count += 1
66
67              return check(state, symbol)
68
69          def back_propagate(self, node: Node, outcome) -> None:
70
71              # For the current player, not the next player
72              reward = 0 if outcome else 1
73
74              while node is not None:
75                  node.N += 1
76                  node.Q += reward
77                  node = node.parent
78                  if empate():
79                      reward = 0
80                  else:
81                      reward = 1 - reward
82
83          def search(self, time_limit: int):
84              start_time = time.process_time()
85
86              num_rollouts = 0
87              while time.process_time() - start_time < time_limit:
88                  node, state = self.select_node()
89                  outcome = self.roll_out(state)
90                  self.back_propagate(node, outcome)
91                  num_rollouts += 1
92
93              run_time = time.process_time() - start_time
94              self.run_time = run_time
95              self.num_rollouts = num_rollouts
96
97          def best_move(self):
98              if check(self.root_state, self.symbol):
99                  return -1
100
101             max_value = max(self.root.children.values(), key=lambda n: n.N).
    N
102             max_nodes = [n for n in self.root.children.values() if n.N ==
    max_value]
103             best_child = random.choice(max_nodes)
104
```

```
105             return best_child.move
106
107         def move(self, move):
108             if move in self.root.children:
109                 self.root_state.move(move)
110                 self.root = self.root.children[move]
111                 return
112
113             self.root_state.move(move)
114             self.root = Node(None, None)
115
116         def statistics(self) -> tuple:
117             return self.num_rollouts, self.run_time, self.node_count
118
119     mcts = MCTS(table, symbol)
120     mcts.search(3)
121     num_rollouts, run_time, node_count = mcts.statistics()
122     print("Statistics: ", num_rollouts, "rollouts in", run_time, "seconds")
123     atualizar_count(0, node_count)
124     print("Number of nodes: ", node_count)
125     move = mcts.best_move()
126     return move
```

Infelizmente os resultados para o MCTS não foram esperados: em vez de ser o algoritmo mais rápido e que apresenta mais dificuldade para o adversário, o nosso código, apesar de ser rápido, não joga de forma inteligente.

Um possível problema seja a função back_propagate() que não faz os cálculos certos para a recompensa de cada nó, atrapalhando todo o processo no cálculo da melhor jogada.

Podem haver outros possíveis erros e bugs que passaram despercebidos pelo grupo

5 Connect Four/ 4 Em Linha

O jogo *Connect Four* é um jogo de estratégia em que dois jogadores alternam jogadas para tentar obter quatro peças da mesma cor em uma linha.

O jogo é jogado em um tabuleiro vertical com sete colunas e seis linhas.

O jogo termina quando um jogador consegue uma linha de quatro peças ou quando todas as casas do tabuleiro estão preenchidas.

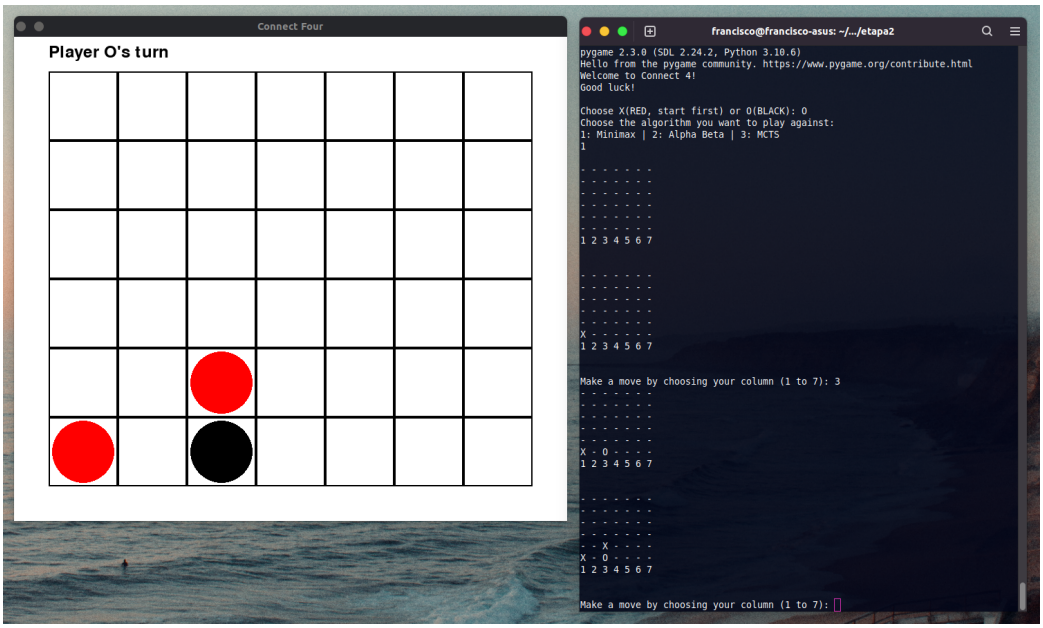


Figure 5: Snapshot de um momento do jogo

Esta imagem é uma implementação do nosso jogo Connect Four usando o pacote Pygame. A interface do jogo é exibida em uma janela gráfica à parte, onde o usuário pode jogar contra o computador através de input no terminal, que usa um dos três algoritmos disponíveis.

5.1 Para jogar, apenas através do terminal:

- O usuário escolhe entre jogar com as peças X(primeiro a jogar, maximizante) ou O.
- O usuário escolhe o algoritmo a utilizar de 1 a 3.

- O usuário faz a sua jogada (1 a 7) contra o CPU até chegar a um estado terminal (vitória ou empate).

6 Implementação

6.1 Requisitos

Para ser possível correr o programa é necessário ter python instalado assim como a biblioteca *pygame*.

- Python preferencialmente Python3.
- Pygame *Version: 2.3.0*

Para instalar Ubuntu/Debian correr o seguinte comando:

```
$ sudo apt-get install python3-pygame
```

6.2 Execução

Para executar o programa, basta correr o seguinte comando:

```
$ python python3 ConnectFour.py
```

Para jogar, seguir os passos no subtópico 5.1; Pode optar por intruduzir um teste de um documento .txt, para isso correr o seguinte comando:

```
$ python python3 ConnectFour.py<teste.txt
```

6.3 Estrutura de dados para o tabuleiro de jogo

Para a representação do tabuleiro usamos uma matriz com 6 linhas e 7 colunas. O tabuleiro inicial fica da seguinte forma:

```
table = [['-','-','-','-','-','-','-'],
         ['-','-','-','-','-','-','-'],
         ['-','-','-','-','-','-','-'],
         ['-','-','-','-','-','-','-'],
         ['-','-','-','-','-','-','-'],
         ['-','-','-','-','-','-','-']]
```

6.4 Movimentos possíveis do jogo

Dada a matriz inicial preenchida na totalidade com '-', escolhemos uma coluna de 1 a 7 e trocamos a peça, na coluna escolhida e na linha mais baixa onde se encontra um '-', por 'X' ou 'O' dependendo de quem é a jogar.

6.5 Funçoes a utilizar:

- **print_board()**: exibe o tabuleiro do jogo na saída padrão;
- **choose_symbol()**: solicita ao usuário que escolha jogar com X ou O e retorna o símbolo escolhido;
- **empate()**: verifica se o tabuleiro está cheio e, neste caso, exibe uma mensagem indicando que o jogo está empatado e retorna True. Caso contrário, retorna False;
- **not_symbol(symbol)**: recebe um símbolo como parâmetro e retorna o outro símbolo;
- **choose_difficulty()**: solicita ao usuário que escolha o nível de dificuldade e retorna um número inteiro correspondente ao algoritmo escolhido;
- **get_move()**: solicita ao usuário que faça uma jogada escolhendo uma coluna e retorna o índice da coluna escolhida.;
- **make_move(symbol, col)**: recebe o símbolo do jogador e o índice da coluna escolhida e realiza a jogada, esta função altera a variável global board para evitar em casos específicos isso criou-se outra:
- **move_selected(thisboard,symbol, col)**;
- **check(table, symbol)**: recebe o tabuleiro e o símbolo do jogador e verifica se o jogador venceu o jogo;
- **utility(table)**: recebe o tabuleiro e retorna a utilidade do tabuleiro para o jogador X;
- **uTable(table)**: recebe o tabuleiro e calcula a utilidade do tabuleiro para o jogador X, basicamente é uma extensão da *utility(table)*.

7 Minimax, alphabeta, mcts aplicados no jogo:

Os algoritmos já foi explicado atrás a sua implementação portanto nesta parte do relatório vamos focar mais na performance dos mesmos.

Corremos alguns testes para comparar o desempenho dos algoritmos.

Para estes resultados usamos uma Depht de 5 para ambos os algoritmos (minimax, alphabeta).

Para mudar a profundidade basta alterar a seguinte linha:

```
# Define depth
DEPTH = 5
```

7.1 Performance

Para a primeira jogada sendo o CPU a começar.

Algoritmos	Tempo	Nós Criados
Minimax	0.79497s	36414
AlphaBeta	0.09926s	2635
MCTS	3s	82098

Não se pode comparar o tempo com o MCTS, pois este não tem limite de profundidade então no sítio do tempo vamos considerar o tempo dado para fazer simulações e nos nós criados, o número de nós explorados em todas as simulações.

7.2 Teste CPU vs Random

Para um jogo random os tempos dos algoritmos foram:

Algoritmos	Minimax	Alphabeta	MCTS
Tempo total	5.5850s	0.5203s	38.76s
Nós totais	311955	16136	640413
Vencedor	CPU	CPU	Player

Jogo em causa:

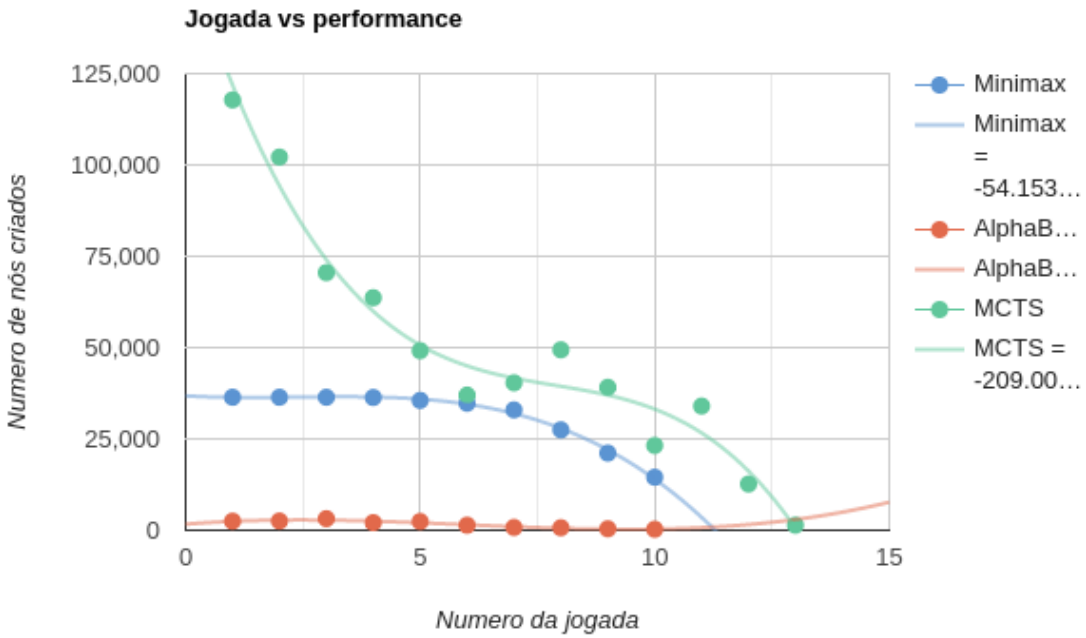
```
4 3 5 3 5 7 3 5 2 6 4 5 2 4 3 6 5 4 2 4
```

Notamos que quando o jogo se desenvolve, os nós e o tempo por jogada também diminuem, isto porque há menos nós para explorar. Isto nota-se significavamente no minimax e alphabeta.

Já no nosso MCTS devido provavelmente a um erro na nossa implementação, ele não é o mais inteligente, então em 3 testes random perdeu-os todos.

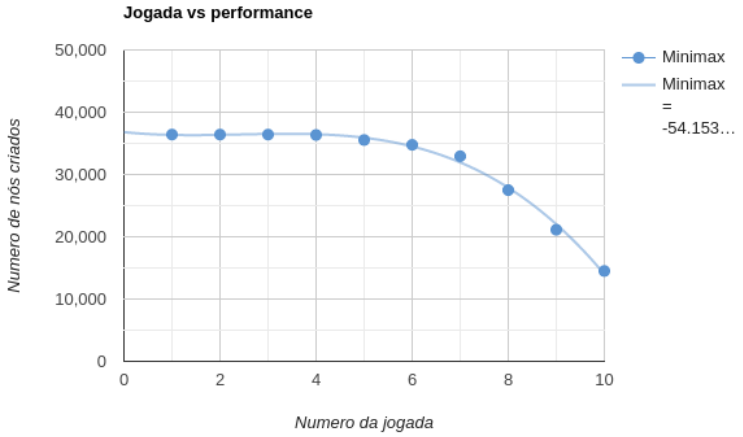
Para o mesmo jogo random, aqui são os resultados para a última jogada individual do CPU:

Algoritmos	Tempo	Nós Criados
Minimax	0.17426s	14511
AlphaBeta	0.00664s	262
MCTS	3s	1385

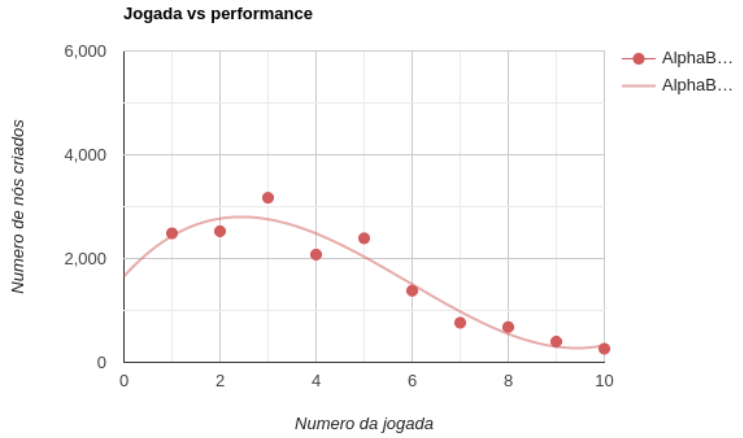


Como esperado, notamos que o alphabeta utiliza muitos menos nós a comparar com o minimax isto deve-se porque ele utiliza máximos/mínimos locais que vão impedir explorar nós que já não serão necessários.

7.2.1 Minimax-resultados

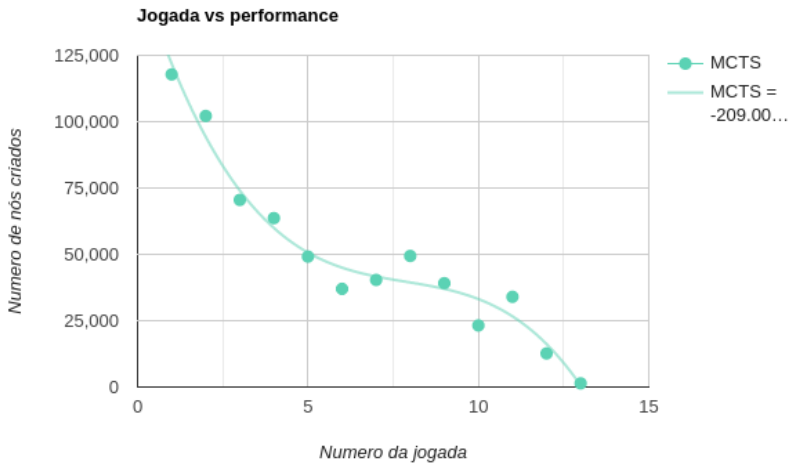


7.2.2 AlphaBeta-resultados



7.2.3 MCTS-resultados

Como o nosso MCTS não é o mais inteligente, os resultados deste algoritmo não serão então os mais viáveis.



7.3 Ai vs Ai (minimax)

Por curiosidade decidimos testar o minimax contra ele próprio, o que obtemos o seguinte resultado:

```
Time taken: 0.0003502368927001953
Nodes visited: 15
O won!
O O X X - O -
X X X O - X -
O O O X O O O
O X X O X X X
X O O X X O X
O O X X X O O
1 2 3 4 5 6 7


Total time CPU played:  12.428297996520996
Total nodes created:  685418
```

8 Conclusão

De acordo com as nossas pesquisas feitas sobre os três algoritmos estudados, conseguimos concluir que o algoritmo Alpha-Beta Pruning é mais eficiente do que o Minimax, visto que é uma versão melhorada do mesmo, onde reduz o numero de nós que são expandidos. Infelizmente não conseguimos comparar nenhum dos algoritmos com o algoritmo Monte Carlo Tree Search, pois não obtemos os resultados esperados.

Visto que não conseguimos comparar nós mesmos, na teoria, enquanto o Minimax como Alpha-Beta são soluções sólidas para abordar jogos onde uma função de avaliação para estimar o resultado do jogo pode ser facilmente definida, o MCTS é uma solução universalmente aplicável, uma vez que nenhuma função de avaliação é necessária devido a sua dependência da aleatoriedade.

9 Webgrafia e Biliografia

9.1 Pesquisas

Locais onde baseamos o nosso código assim como ideologias:

Video-Minimax/Alphabeta

Code for the book "Artificial Intelligence: A Modern Approach"

Monte-carlo

9.2 Imagens

Links de imagens utilizadas:

Minimax

AlphaBeta-pdf

Monte Carlo tree search