

JOGO DA VELHA COM O ALGORITMO MINMAX

INTELIGÊNCIA ARTIFICIAL
UNIVERSIDADE FEDERAL DE UBERLÂNDIA

S U M Á R I O

01

Jogo da velha

- Formulando o problema

02

Visão geral do Algoritmo Min-Max e Poda Alfa-Beta

- Implementação algoritmos

03

Comparação MIN-MAX com e sem poda

04

Demonstração do MIN-MAX no jogo da velha

JOGO DA VELHA

É um jogo de tabuleiro simples e popular para dois jogadores, que geralmente é jogado em um tabuleiro 3x3. Apesar de ser um jogo de estratégia simples, requer planejamento tático para bloquear o oponente e criar oportunidades para ganhar.

REGRAS

- O jogo é jogado em um tabuleiro 3x3, resultando em nove espaços em branco.
- Dois jogadores alternam suas jogadas, um usando "X" e o outro usando "O". O jogador "X" geralmente começa.
- Os jogadores escolhem um espaço vazio no tabuleiro para colocar sua marca durante sua vez.
- O jogo continua até que um jogador forme uma linha contínua de três de suas peças na horizontal, na vertical ou na diagonal, ou até que o tabuleiro esteja completamente preenchido sem que nenhum jogador ganhe (empate).



JOGO DA VELHA

FORMULANDO O PROBLEMA

Um jogo pode ser definido formalmente como uma espécie de problema de busca com os seguintes componentes:

- Estado inicial
- Jogadores
- Ações
- Resultado
- Teste de término
- Utilidade

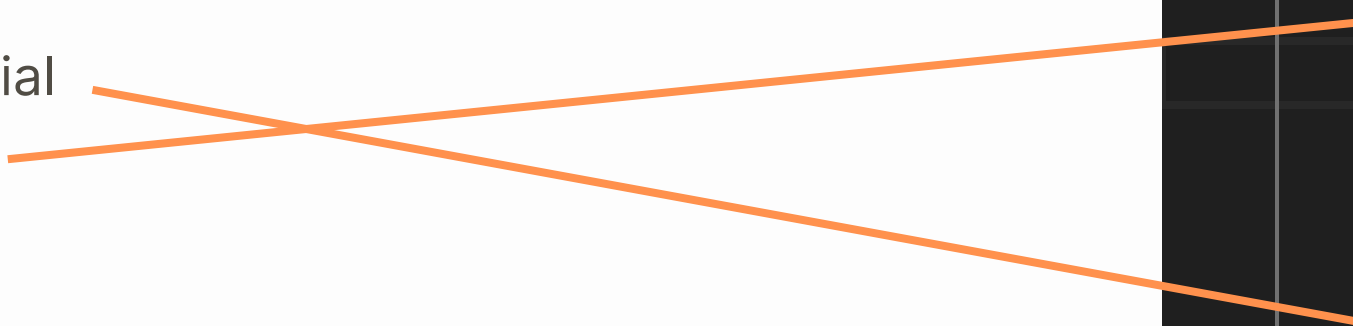
```
class TicTacToe:

    def __init__(self, initial_state):
        self.X = "X"
        self.O = "O"
        self.EMPTY = None
        self.states_visited = 0

        if(initial_state == None):
            self.initial_state = [[self.EMPTY, self.EMPTY, self.EMPTY],
                                   [self.EMPTY, self.EMPTY, self.EMPTY],
                                   [self.EMPTY, self.EMPTY, self.EMPTY]]

        else:
            self.initial_state = initial_state

    def get_states_visited(self):
        return self.states_visited
```

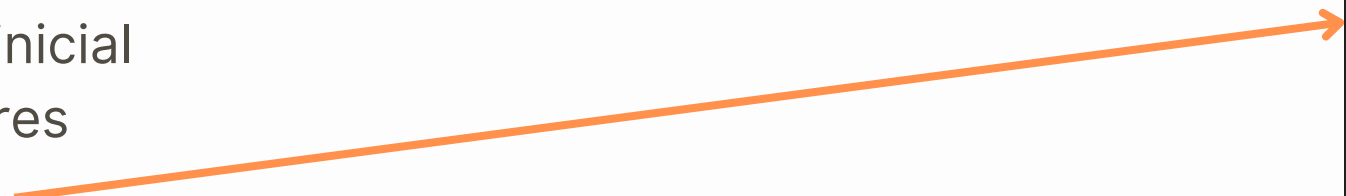


JOGO DA VELHA

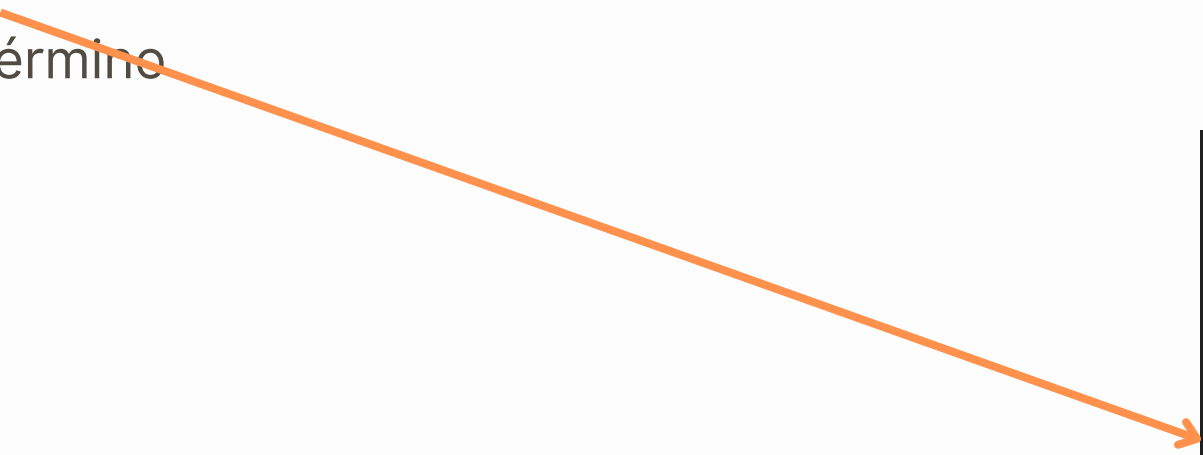
FORMULANDO O PROBLEMA

Um jogo pode ser definido formalmente como uma espécie de problema de busca com os seguintes componentes:

- Estado inicial
- Jogadores
- Ações
- Resultado
- Teste de término
- Utilidade



```
def actions(self, board):  
  
    actions = []  
    for i in range(3):  
        for j in range(3):  
            if board[i][j] == None:  
                actions.append((i, j))  
    return actions
```



```
def result(self, board, action):  
  
    if action not in self.actions(board):  
        raise Exception("Invalid Action")  
  
    new_board = [row[:] for row in board]  
  
    new_board[action[0]][action[1]] = self.player(board)  
  
    return new_board
```

JOGO DA VELHA

FORMULANDO O PROBLEMA

Um jogo pode ser definido formalmente como uma espécie de problema de busca com os seguintes componentes:

- Estado inicial
- Jogadores
- Ações
- Resultado
- Teste de término
- Utilidade

```
def terminal(self, board):
    """
    Retorna True se o jogo terminar, False caso contrário.
    """

    if self.winner(board) != None:
        return True
    else:
        # Check if the board is full
        for i in range(3):
            for j in range(3):
                if board[i][j] == None:
                    return False
        return True
```

```
def winner(self, board):
    """
    Retorna o vencedor do jogo, se houver.
    """

    # Return tictactoe winner if there is one
    for i in range(3):
        # Check rows
        if board[i][0] == board[i][1] == board[i][2]:
            if board[i][0] != None:
                return board[i][0]

        # Check columns
        if board[0][i] == board[1][i] == board[2][i]:
            if board[0][i] != None:
                return board[0][i]

        # Check Diagonals
        if board[0][0] == board[1][1] == board[2][2]:
            if board[0][0] != None:
                return board[0][0]
        if board[0][2] == board[1][1] == board[2][0]:
            if board[0][2] != None:
                return board[0][2]

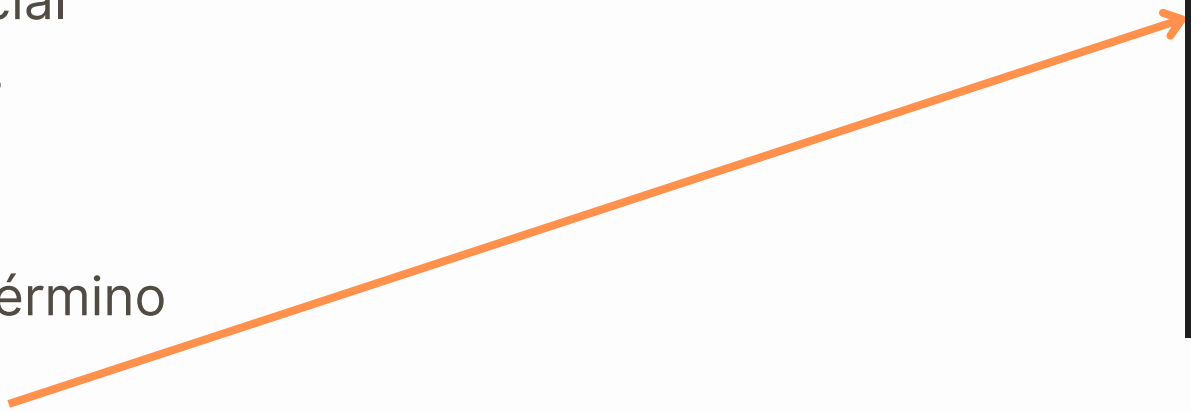
    return None
```

JOGO DA VELHA

FORMULANDO O PROBLEMA

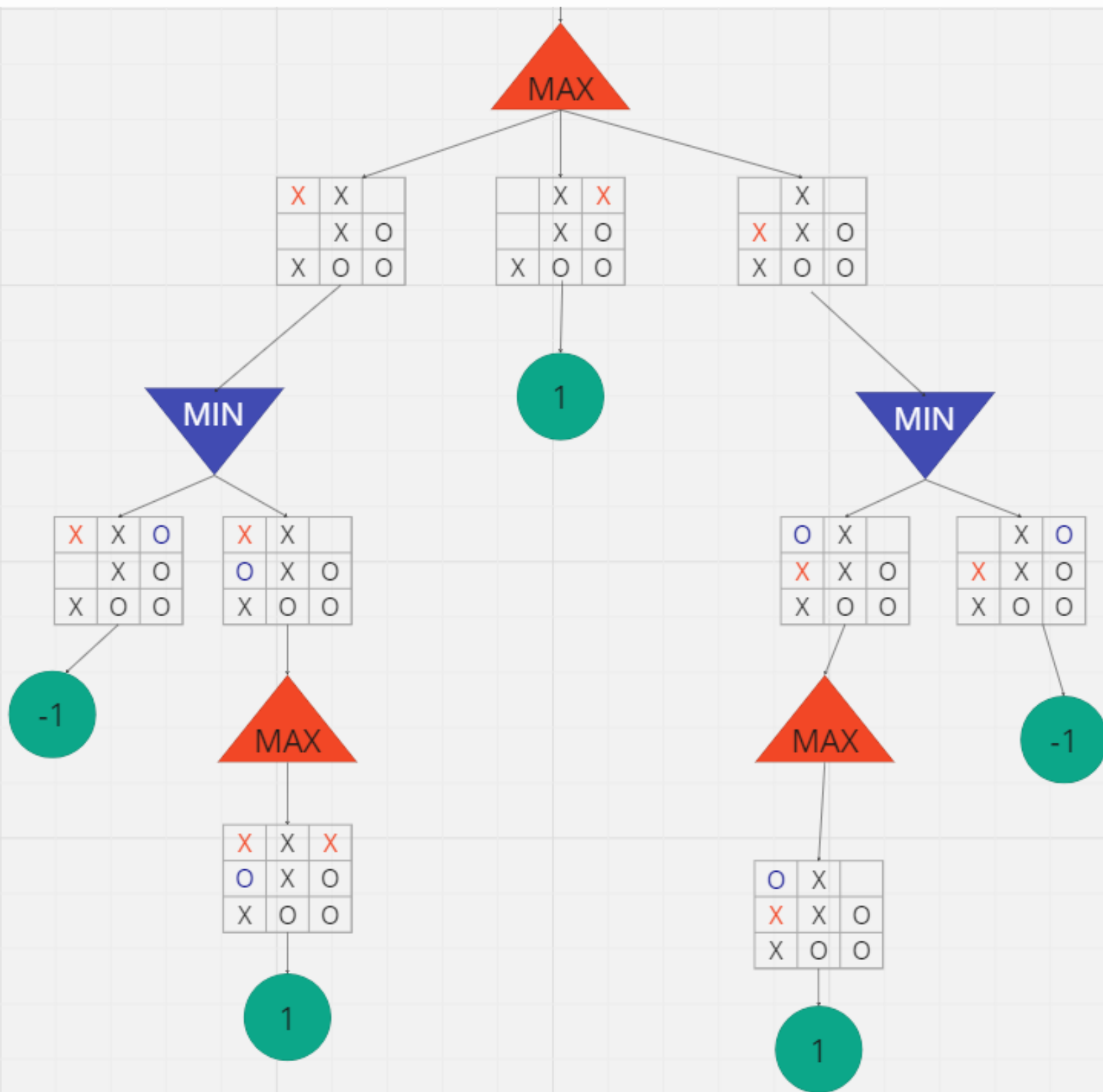
Um jogo pode ser definido formalmente como uma espécie de problema de busca com os seguintes componentes:

- Estado inicial
- Jogadores
- Ações
- Resultado
- Teste de término
- Utilidade



```
def utility(self, board):  
    """  
    Retorna 1 se X ganhou o jogo, -1 se O ganhou, 0 caso contrário.  
    """  
    if self.winner(board) == self.X:  
        return 1  
    elif self.winner(board) == self.O:  
        return -1  
    else:  
        return 0
```

ALGORITMO MIN-MAX



```
function MINIMAX-DECISION(state) returns an action  
return  $\arg \max_{a \in \text{ACTIONS}(s)} \text{MIN-VALUE}(\text{RESULT}(state, a))$ 
```

```
function MAX-VALUE(state) returns a utility value  
if TERMINAL-TEST(state) then return UTILITY(state)  
 $v \leftarrow -\infty$   
for each a in ACTIONS(state) do  
     $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s, a)))$   
return v
```

```
function MIN-VALUE(state) returns a utility value  
if TERMINAL-TEST(state) then return UTILITY(state)  
 $v \leftarrow \infty$   
for each a in ACTIONS(state) do  
     $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s, a)))$   
return v
```

PONTOS CHAVE

1. Calcula a melhor decisão a partir do estado corrente
2. É um algoritmo recursivo que percorre todo o caminho até as folhas da árvore
3. Ao encontrar um estado terminal, retorna o valor da **FUNÇÃO UTILIDADE** (nessa implementação, -1 se jogador O ganhar, 1, se jogador X e 0 empate)
4. Os valores MINMAX são propagados de volta pela árvore, à medida que a recursão retorna.

IMPLEMENTAÇÃO ALGORITMO MIN-MAX

MAX

```
def max_value_nopruning(self, board):  
    """  
    Retorna o valor máximo para o jogador atual no tabuleiro  
    """  
    self.states_visited += 1  
  
    if self.terminal(board):  
        return self.utility(board)  
  
    v = -math.inf  
  
    for action in self.actions(board):  
        v = max(v, self.min_value_nopruning(self.result(board, action)))  
  
    return v
```

MIN

```
def min_value_nopruning(self, board):  
    """  
    Retorna o valor mínimo para o jogador atual no tabuleiro  
    """  
  
    if self.terminal(board):  
        return self.utility(board)  
  
    v = math.inf  
  
    for action in self.actions(board):  
        v = min(v, self.max_value_nopruning(self.result(board, action)))  
  
    return v
```

ALGORITMO MIN-MAX COM PODA ALFA-BETA

PONTOS CHAVE

1. Alfa: é a melhor escolha ou o valor mais alto que encontramos em qualquer instância ao longo do caminho do MAX. O valor inicial para alfa é $-\infty$.
2. Beta: é a melhor escolha ou o valor mais baixo que encontramos em qualquer instância ao longo do caminho do Minimizador. O valor inicial para alfa é $+\infty$.
3. Cada nó deve acompanhar seus valores alfa e beta. Alfa pode ser atualizado apenas quando for a vez de MAX e, da mesma forma, o beta pode ser atualizado apenas quando for a vez de MIN.
4. Os valores dos nós serão passados para os nós superiores em vez dos valores de alfa e beta durante a reversão da árvore.
5. Os valores Alfa e Beta só são passados para nós filhos.

```
function ALPHA-BETA-SEARCH(state) returns an action  
   $v \leftarrow \text{MAX-VALUE}(\text{state}, -\infty, +\infty)$   
  return the action in ACTIONS(state) with value v
```

```
function MAX-VALUE(state,  $\alpha$ ,  $\beta$ ) returns a utility value  
  if TERMINAL-TEST(state) then return UTILITY(state)  
   $v \leftarrow -\infty$   
  for each a in ACTIONS(state) do  
     $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$   
    if  $v \geq \beta$  then return v  
     $\alpha \leftarrow \text{MAX}(\alpha, v)$   
  return v
```

```
function MIN-VALUE(state,  $\alpha$ ,  $\beta$ ) returns a utility value  
  if TERMINAL-TEST(state) then return UTILITY(state)  
   $v \leftarrow +\infty$   
  for each a in ACTIONS(state) do  
     $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$   
    if  $v \leq \alpha$  then return v  
     $\beta \leftarrow \text{MIN}(\beta, v)$   
  return v
```

IMPLEMENTAÇÃO ALGORITMO MIN-MAX COM PODA ALFA-BETA

MAX

```
def max_value(self, board, alpha, beta):
    """
    Retorna o valor máximo para o jogador atual no tabuleiro
    usando poda alfa-beta.
    """
    self.states_visited += 1

    if self.terminal(board):
        return self.utility(board)
    v = -math.inf

    actions = self.actions(board)

    for action in actions:
        v = max(v, self.min_value(self.result(board, action), alpha, beta))

        if v >= beta:
            break

        alpha = max(alpha, v)
    return v
```

MIN

```
def min_value(self, board, alpha, beta):
    """
    Retorna o valor mínimo do jogador atual no tabuleiro
    usando poda alfa-beta.
    """
    #self.states_visited += 1

    if self.terminal(board):
        return self.utility(board)
    v = math.inf

    actions = self.actions(board)

    for action in actions:
        v = min(v, self.max_value(self.result(board, action), alpha, beta))

        if v <= alpha:
            break

        beta = min(beta, v)
    return v
```

IMPLEMENTAÇÃO ALGORITMO MIN-MAX

CONTEXTO JOGO DA VELHA

Quando duas máquinas jogam o jogo da velha usando o algoritmo MIN-MAX, o resultado normalmente **deve resultar em empate se ambas as máquinas jogarem de maneira ótima.**

O algoritmo MIN-MAX é uma estratégia de tomada de decisão que **garante que um jogador (ou máquina) faça o melhor movimento possível em cada jogada,** considerando o pior cenário, assumindo que o adversário também jogue de forma otimizada

```
def minimax(self, board):
    """
    Retorna a ação ideal para o jogador atual no tabuleiro
    usando o algoritmo minimax com poda alfa-beta.
    """

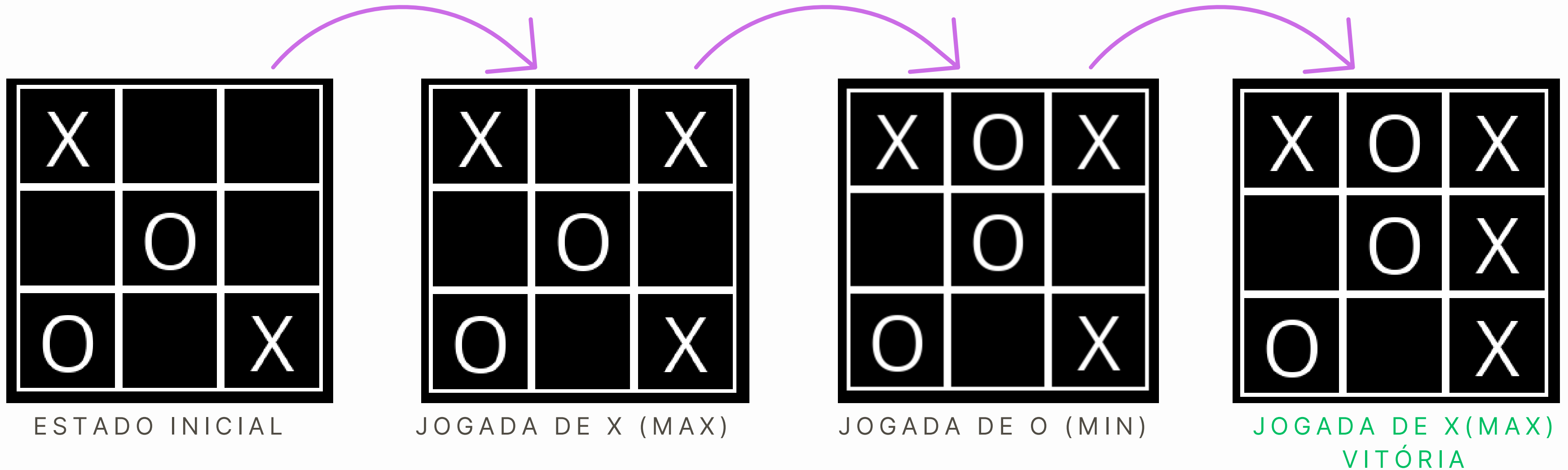
    if self.terminal(board):
        return None

    if self.player(board) == self.X:
        v = -math.inf
        opt_action = None
        actions = self.actions(board)
        for action in actions:
            new_value = self.min_value(self.result(board, action), -math.inf, math.inf)
            if new_value > v:
                v = new_value
                opt_action = action
            if (self.utility(self.result(board, action)) == 1) :
                return action
        return opt_action

    elif self.player(board) == self.O:
        v = math.inf
        opt_action = None
        for action in self.actions(board):
            new_value = self.max_value(self.result(board, action), -math.inf, math.inf)
            if new_value < v:
                v = new_value
                opt_action = action
            if (self.utility(self.result(board, action)) == -1) :
                return action
        return opt_action
```

COMPARANDO OS ALGORITMOS

MOVIMENTOS, DADO ESTADO INICIAL PARA ALGORITMO COM PODA E SEM PODA

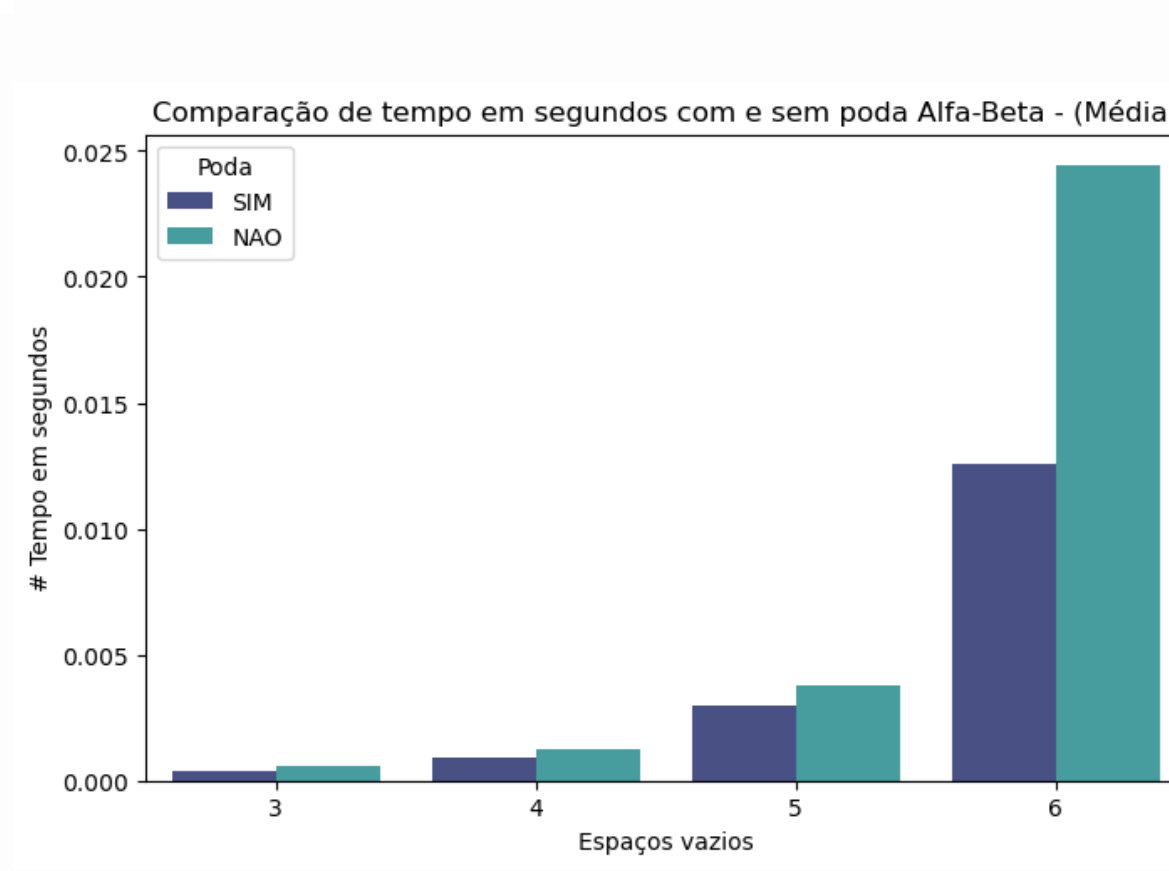
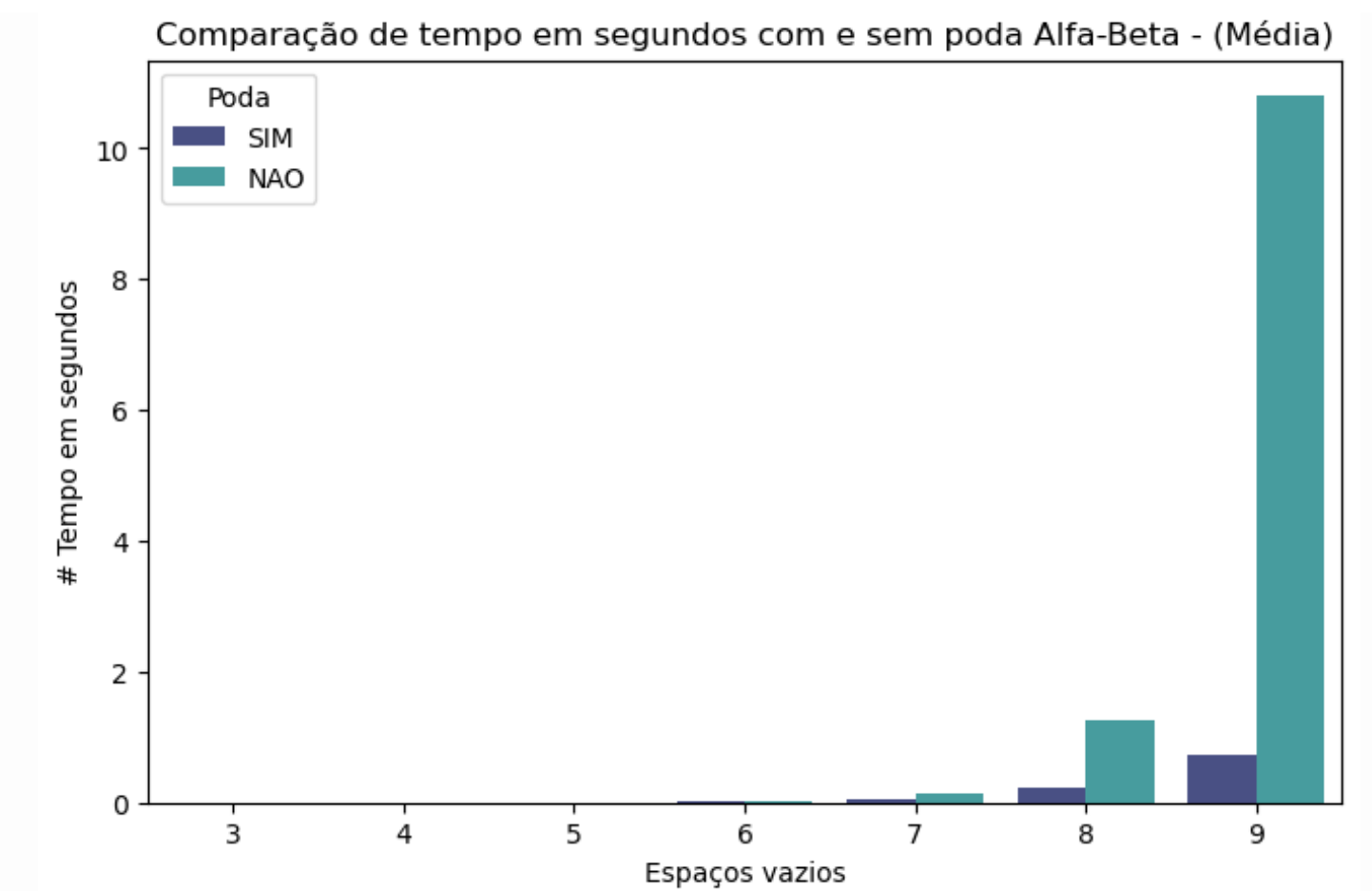
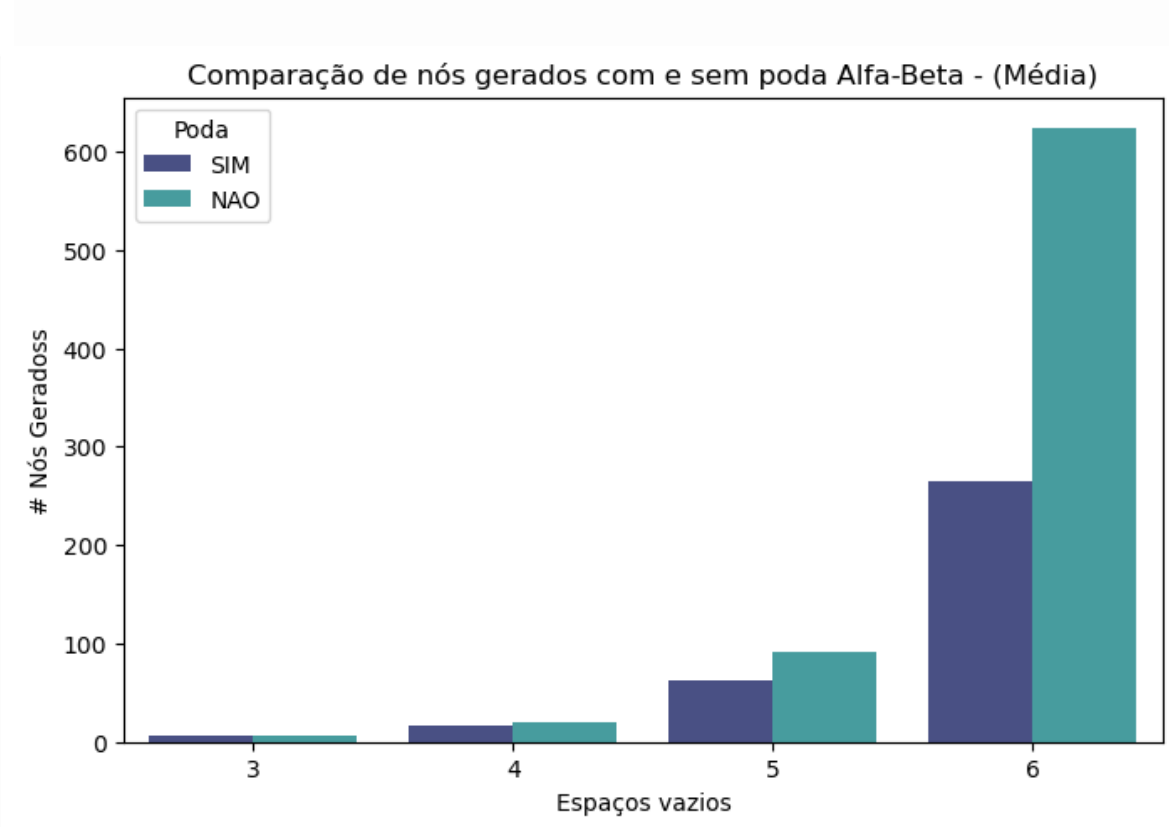
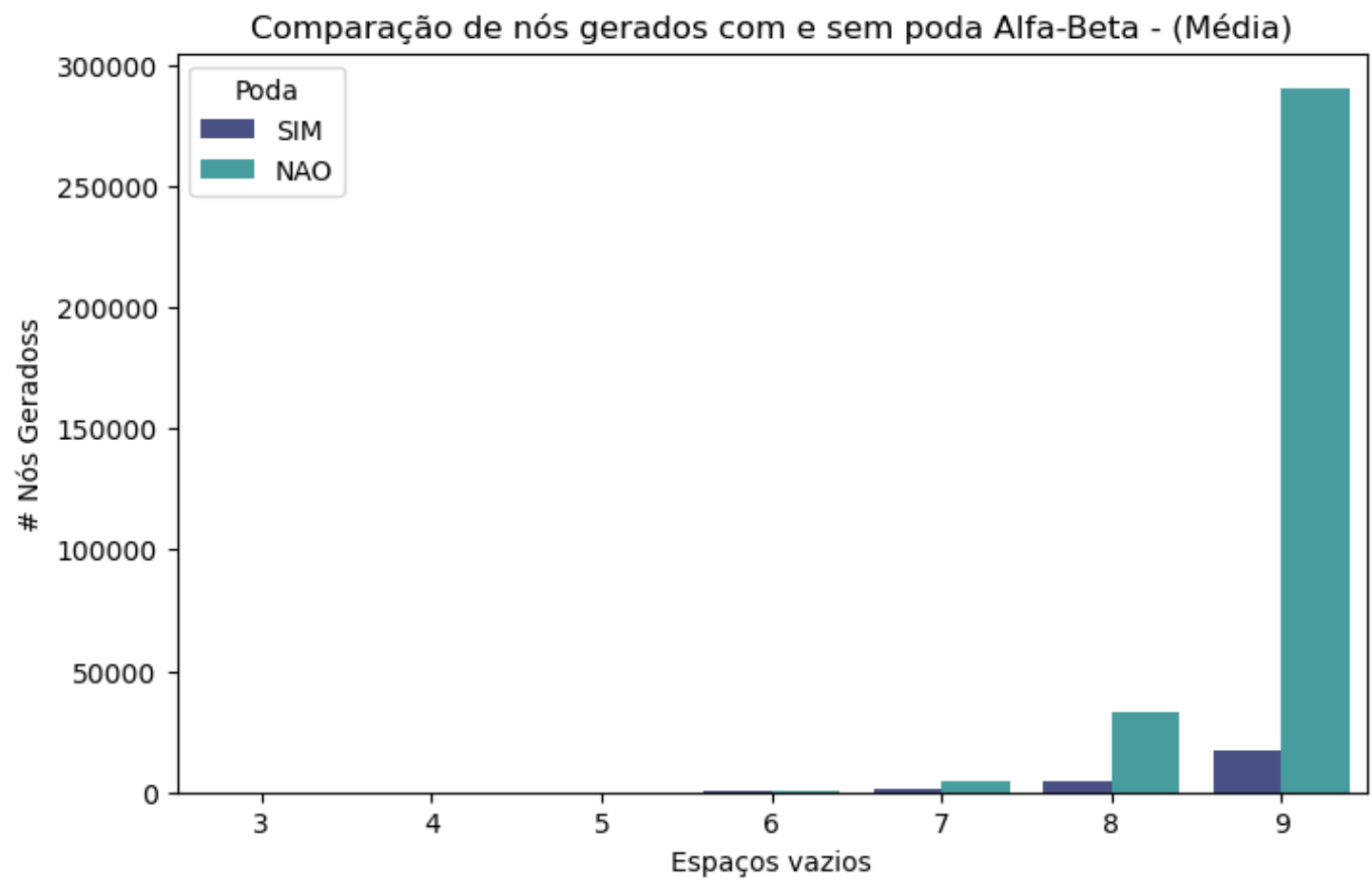


Considerando o tabuleiro vazio, onde a primeira jogada é do jogador X (MAX), com a **poda alfa beta, ao final do jogo, 17446 estados foram explorados**. Já com o algoritmo **sem poda, foram explorados aproximadamente 290 mil estados**, quase 17 vezes maior do que o algoritmo com a poda.

Já com o estado inicial mostrado na figura 1, 58 estados explorados com a poda alfa beta. Sem a poda, 110 estados. Praticamente o dobro!

Pensando em um problema maior, esse número e consequentemente o tempo de execução, em um algoritmo sem a poda será extremamente mais elevado

COMPARANDO OS ALGORITMOS



Foram testados **aproximadamente 5 mil estados distintos, alternando o primeiro a jogar entre MAX e MIN, e o número de espaços vazios no tabuleiro**. Isso é, 6 espaços vazios significa apenas 4 quadros marcados pelos jogadores.

É notável que tanto o tempo de execução quanto a quantidade de nós gerados aumenta exponencialmente quanto mais possibilidades de jogada existir em um dado momento do jogo.

DEMO

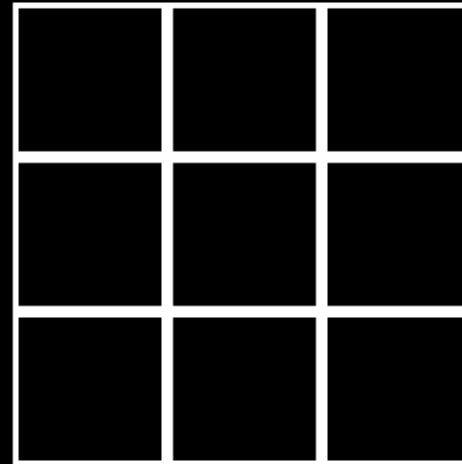
Select the MinMax version

Alpha Beta Pruning

No Pruning

pygame window

Play Tic-Tac-Toe (MinMax with Alpha-Beta Pruning)



Change Algorithm

Choose Initial State

pygame window

Play Tic-Tac-Toe (MinMax with Alpha-Beta Pruning)

Play as X

Play as O

Machine VS Machine

Reset Initial State



REFERÊNCIAS

RUSSELL, Stuart J. Artificial intelligence a modern approach. Pearson Education, Inc., 2010.