CONNECT-FOUR

INTELIGÊNCIA ARTIFICIAL

MAFALDA AIRES, UP202106550
GONÇALO MONTEIRO, UP202105821

RELATÓRIO

INTRODUÇÃO

Jogos com Oponentes, em inglês *Adversarial Games*, são ambientes competitivos, nos quais dois ou mais agentes tem objetivos conflituosos, isto é, as ações e escolhas de um agente afetam o sucesso do agente adversário.

O objetivo de um agente é vencer o outro agente. Neste contexto, os problemas de busca neste tipo de jogos procura encontrar a melhor jogada possível de um agente num determinado estado do jogo, tentando antecipar as possíveis reações do agente adversário à sua jogada. Os problemas de busca são resolvidos por meio de algoritmos de busca, como o Minimax, o Alpha-Beta Pruning e o Monte Carlo tree search (MCTS).

Exemplo destes tipo de jogos são os jogos de tabuleiro como xadrez, damas, jogo da velha, Connect-Four, entre outros.

ALGORITMO MINIMAX

O algoritmo Minimax baseia-se numa árvore de pesquisa, onde cada nó representa um estado do jogo.

A ideia do algoritmo Minimax é percorrer recursivamente a árvore até uma profundidade limite pré-definida, avaliando no final qual o melhor nó a ser expandido.

Existem dois jogadores na pesquisa: o minimizador, que recebe dos descendentes o menor valor de utilidade; e o maximizador, que por sua vez procura a maior utilidade. O jogo é feito alternadamente, uma jogada por cada jogador.

Concluindo, o Minimax é um algoritmo que irá percorrer recursivamente a árvore de pesquisa até um limite de profundidade pré-definido, retornar o valor da utilidade dos estados terminais e na decisão, caso seja a vez do jogador minimizador, o nó escolhido será aquele que retorna um menor menor valor utilidade e, no caso de ser a vez do jogador maximizador, o nó será o que que representa maior utilidade.

ALGORITMO ALPHA-BETA PURNING

O algoritmo Alpha-Beta Purning é uma decisão implementada pelo algoritmo Minimax, sem a necessidade de examinar todos os nós da árvore até uma certa profundidade pré-definida.

Este algoritmo difere-se do algoritmo Minimax uma vez que, ao longo da sua execução, mantém duas variáveis: alpha (α) e beta (β) que representam os valores das melhores jogadas encontradas até o momento para os jogadores maximizador e minimizador, respectivamente. Os valores das melhores jogadas encontradas até ao momento representam os melhores valores de utilidade encontrados até ao momento (o mais alto, no caso de ser o jogador maximizador e o mais baixo, no caso de ser o jogador minimizador).

Durante a execução do algoritmo, o algoritmo avalia um nó, tendo em conta a sua utilidade. No caso de ser o jogador maximizador: se alpha do nó atual (modificado no caso da utilidade do nó atual ser maior que o alpha) for maior que o beta do nó pai, interrompe-se a expansão desse ramo da árvore. No caso de ser o jogador minimizador: se beta do nó atual (modificado no caso da utilidade do nó atual ser menor que o beta) for menor que o alpha do nó pai, interrompe-se a expansão desse ramo da árvore. Esta interrupção deve-se ao facto de não ir encontrar nenhuma solução ótima naquele ramo.

ALGORITMO MONTE CARLO TREE SEARCH (MCTS)

O Monte Carlo tree search (MCTS) é um algoritmo que se divide em quatro funções essenciais:

- Expansion: A partir de um nó folha já visitado, expande-se os seus filhos (jogadas possíveis).

- Selection: A partir da raiz da árvore (root), um apontador desce a árvore de pesquisa, escolhendo sempre o nó descendente com melhor valor UCT, calculado pela função Upper Confidence Bound. Esta função depende de uma constante pré definida e a relação entre o número de vezes que um nó foi visitado e o número de vezes em que por ele se chegou a uma vitória.

- Simulation: Em seguida, a partir do estado do nó selecionado, realiza-se um jogo aleatório até se chegar a um estado terminal e, tendo em conta o vencedor, realiza-se a função BackPropagation.

- BackPropagation: Faz-se o update do número de visitas e número de vitórias dos nós pais até à raiz (vitórias/visitas):

visita+=1;

vitória+=1 se venceu;

vitória+=0 se perdeu ou empatou.

Repete estes quatro processos até que seja atingido um limite de tempo pré-definido. Depois desse tempo, a melhor jogada é escolhida com base na contagem de visitas dos nós da árvore ou valor de UCT.

JOGO CONNECT-FOUR

DESCRIÇÃO E REGRAS

Connect-Four é um jogo de dois jogadores, alternando a vez entre cada jogador.

O jogador 1 joga sempre 'X' e o jogador 2 joga sempre 'O', sendo o jogador 1 sempre o primeiro a iniciar o jogo.

O tabuleiro do jogo Connect-Four tem medida 6x7.

A jogada de cada jogador consiste na escolha de uma coluna do tabuleiro [1 , 7] e, caso não seja uma coluna inexistente e a coluna não esteja completa, insere a peça do jogador no fundo da coluna escolhida, ocupando o espaço mais baixo disponível.

O objetivo do jogo é criar uma sequência de 4 peças iguais seguidas na horizontal ou na vertical ou na diagonal. Como há dois jogadores e o objetivo do jogo é mútuo, além de tentar criar uma sequência de 4 peças seguidas, o jogador também tem de evitar que o jogador adversário o faça. Assim, para um jogador ganhar, o jogador adversário tem de perder.

FUNÇÕES DE AVALIAÇÃO

1. Função que avalia a vez de jogar (Isto permite saber quem é o próximo jogador a jogar num tabuleiro aleatório).

```
def to_move(self):
    sum_of_moves=0
    for i in self.state:
        for j in i:
            if(j=='X' or j=='O'):
                sum_of_moves+=1
    if sum_of_moves%2==0:
        return 1
    elif sum_of_moves%2==1:
        return -1
```

2. Função responsável por, através de um dado input, jogada, tendo em conta quem é a vez de jogar e verificando se essa coluna não está completa, coloca a peça no devido lugar dessa coluna. No fim da jogada, alterna a vez de jogar.

```
def move(self,col):
    self.last=col
    col-=1
    if self.has_space(col):
        i=0
        while(i<6 and self.state[i][col]==' '):
            i+=1
        i-=1
        if self.turn==1:
            self.state[i][col]='X'
            self.turn=self.to_move()
            return
        elif self.turn==-1:
            self.state[i][col]='O'
            self.turn=self.to_move()
            return
```

3. Função que retorna uma lista com as jogadas possíveis de um tabuleiro aleatório (possible_moves). Não fazemos simplesmente através da função legal_actions, porque isso alteraria o estado atual do jogo. Portanto, optamos por fazer n deepcopy(s) do estado inicial do tabuleiro (n=número de ações possíveis) e, para cada ação possível (avaliado pela função legal_actions), movemos cada deepcopy do estado inicial do tabuleiro, tendo em conta cada ação possível. Assim, através da função possible_moves, retornamos uma lista com os movimentos possíveis do estado atual do tabuleiro, sem alterar o estado atual do tabuleiro.

```
def result(self,m):
    result=deepcopy(self)
    result.move(m)
    return result

def legal_actions(self):
    lista=[]
    if self.is_terminal()==False:
        for i in range(7):
            if self.has_space(i):
                lista.append(i+1)
    return lista

def possible_moves(self):
    lista=[]
    for i in self.legal_actions():
        jogo=self.result(i)
        lista.append(jogo)
    return lista
```

4. Função que verifica se o estado atual do tabuleiro é um estado terminal, isto é se o jogo acabou, recorrendo ao uso da função utilidade.

```
def is_terminal(self):
    if (self.utility()==512) or (self.utility()==-512) or (self.full()==True):
        return True
    return False
```

5. Função que verifica, através do estado atual e terminal do jogo, se o jogador 1 ganhou ('X'), se o jogador -1 ganhou ('O') ou se houve empate (Única maneira de ocorrer empate é se o tabuleiro estiver completo), recorrendo ao uso da função utilidade.

```
def winner(self):
    if self.utility()==512:
        return 1
    elif self.utility()==-512:
        return -1
    return 0
```

6. Função que avalia a utilidade dos estados terminais, baseado no artigo "R. L. Rivest, Game Tree Searching by Min/Max Approximation, AI 34 [1988], pp. 77-96":

"a win by X has a value of +512, a win by O has a value of -512, a draw has a value of 0, otherwise, take all possible straight segments on the grid (defined as a set of four slots in a line: horizontal, vertical, or diagonal), evaluate each of them according to the rules below and return the sum of the values over all segments.

The rules for evaluating segments are as follows:

-50 for three Os, no Xs,
-10 for two Os, no Xs,
- 1 for one O, no Xs,
0 for no tokens, or mixed Xs and Os,
1 for one X, no Os,
10 for two Xs, no Os,
50 for three Xs, no Os."

```
def utility(self):
    if self.full()==True:
        return 0
    segment=[]
    value=0

    for i in range(ROWS):
        for j in range(COLS-3):
            segment=self.state[i][j:j+4]
            k=self.evaluate_segment(segment)
            if k==512 or k==-512:
                return k
            value+=k

    for i in range(COLS):
        for j in range(ROWS-3):
            segment=[self.state[j+inc][i] for inc in range(4)]
            k=self.evaluate_segment(segment)
            if k==512 or k==-512:
                return k
            value+=k

    for i in range(ROWS-3):
        l=i
        for j in range(COLS-3-1-i):
            segment=[self.state[l+inc][j+inc] for inc in range(4)]
            k=self.evaluate_segment(segment)
            if k==512 or k==-512:
                return k
            value+=k
            l+=1
```

```
    for i in range(3,ROWS):
        l=i
        for j in range(COLS-3-(ROWS-i)):
            segment=[self.state[l-inc][j+inc] for inc in range(4)]
            k=self.evaluate_segment(segment)
            if k==512 or k==-512:
                return k
            value+=k
            l-=1

    for i in range(ROWS-3):
        l=i
        for j in range(6,6-3+i,-1):#6,6-3+i,-1
            segment=[self.state[l+inc][j-inc] for inc in range(4)]
            k=self.evaluate_segment(segment)
            if k==512 or k==-512:
                return k
            value+=k
            l+=1

    for i in range(5,2,-1):
        l=i
        for j in range(6,6-(i-2),-1):
            segment=[self.state[l-inc][j-inc] for inc in range(4)]
            k=self.evaluate_segment(segment)
            if k==512 or k==-512:
                return k
            value+=k
            l-=1

    return value
```

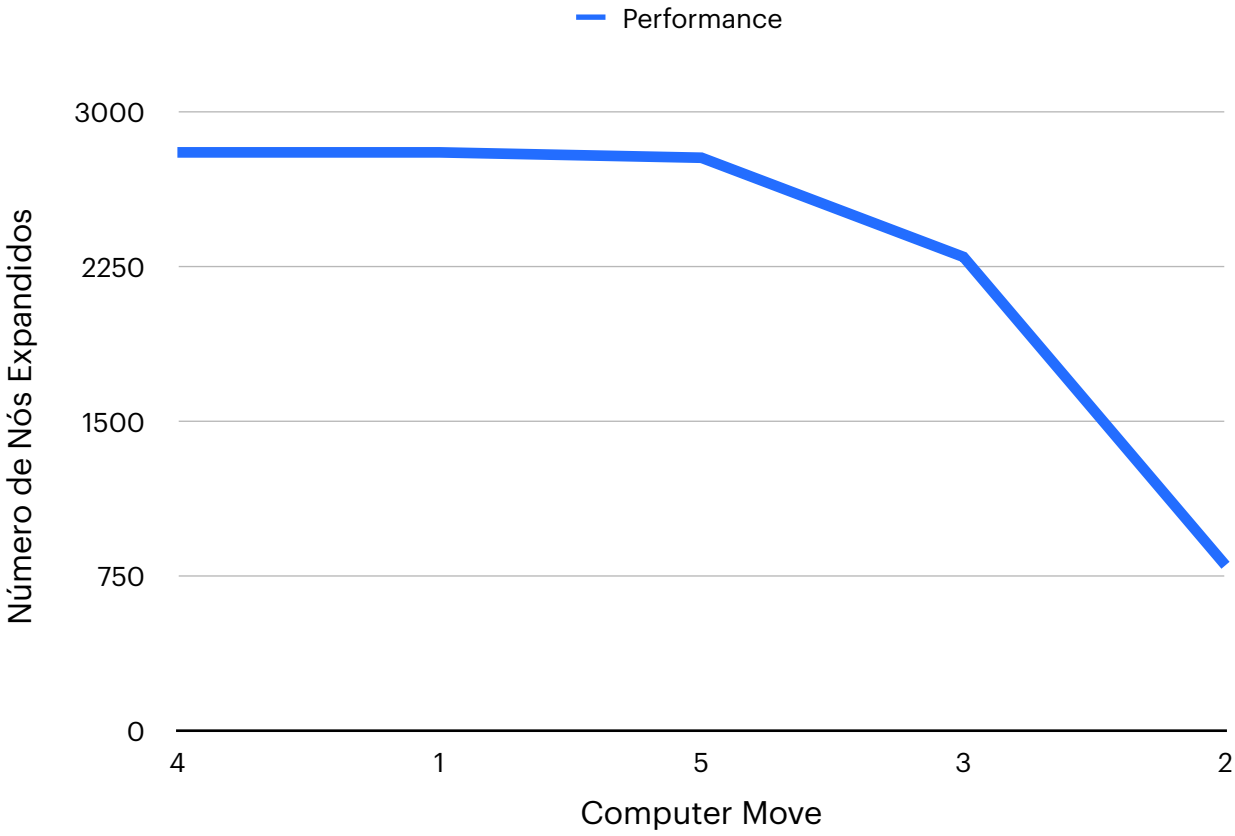
7. Função que serve de auxílio para a função utilidade, sendo responsável pela análise de todas as secções retas de comprimento 4.

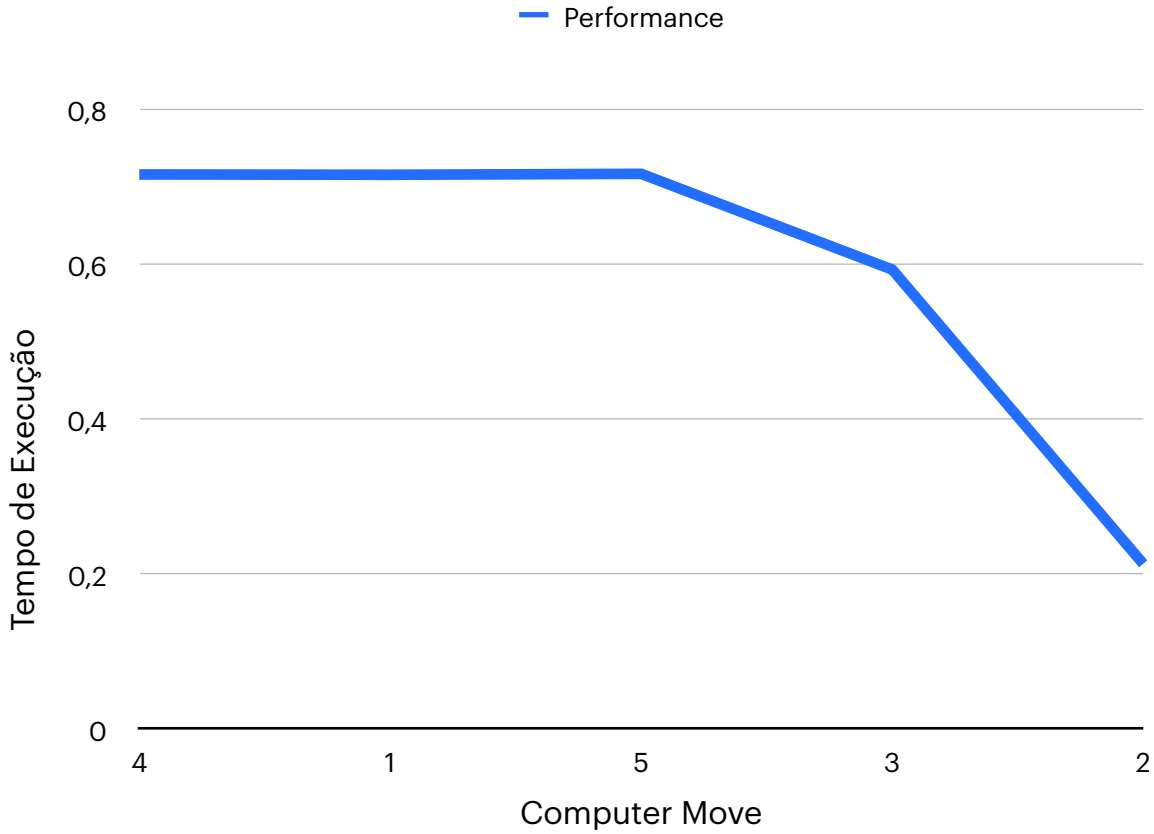
```
def evaluate_segment(self,section):
    countx=0
    county=0
    value=0
    for i in section:
        if i=='X':
            countx+=1
        elif i=='O':
            county+=1

    if county==0 and countx!=0:
        if countx==4:
            return 512
        elif countx==3:
            return 50
        elif countx==2:
            return 10
        elif countx==1:
            return 1
    elif countx==0 and county!=0:
        if county==4:
            return -512
        elif county==3:
            return -50
        elif county==2:
            return -10
        elif county==1:
            return -1
    return 0
```

ALGORITMOS APLICADOS AO CONNECT-FOUR

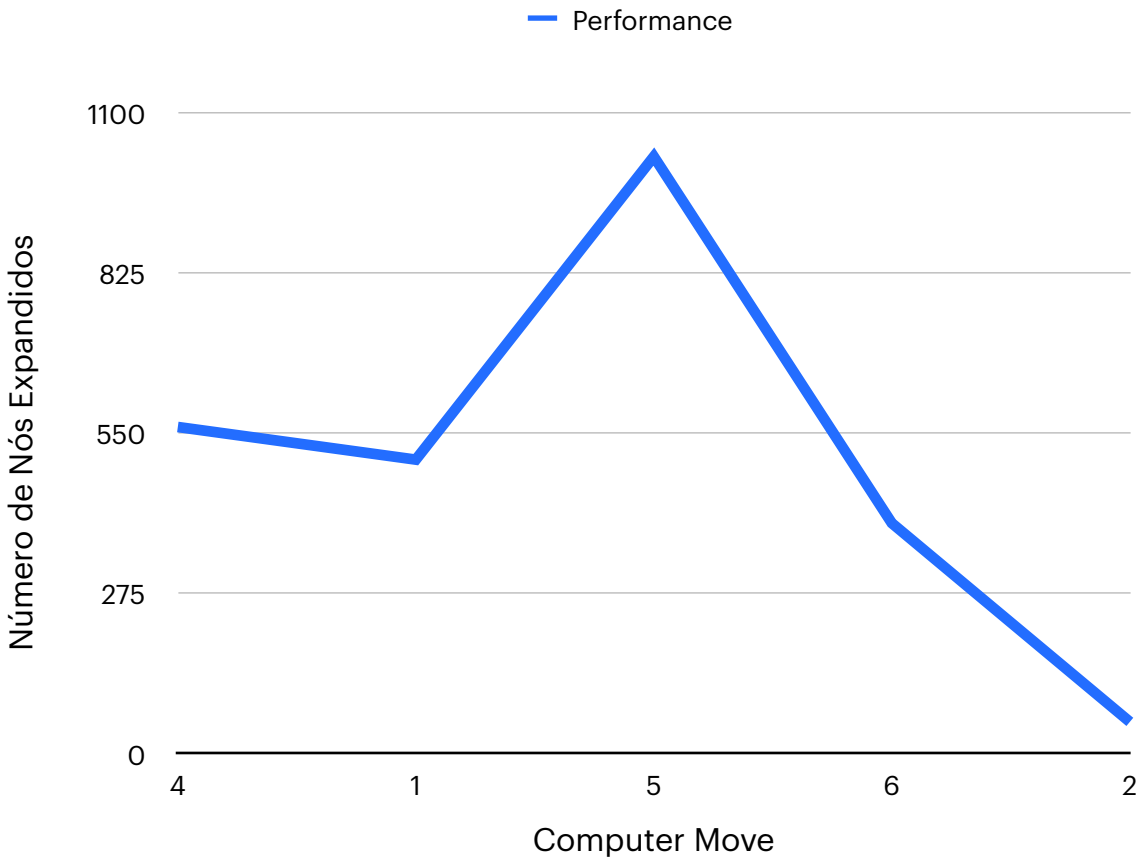
AVALIAÇÃO DA PERFORMANCE DOS ALGORITMOS

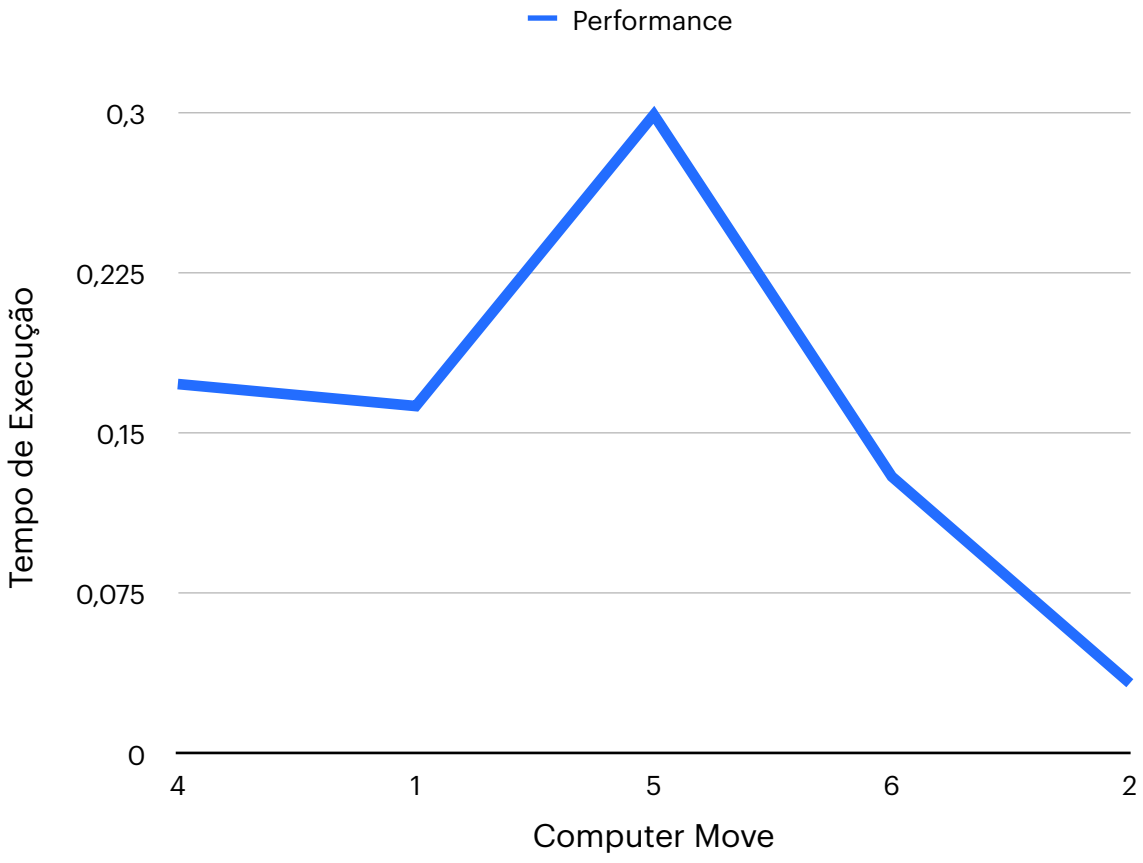
ALGORITMO MINIMAX

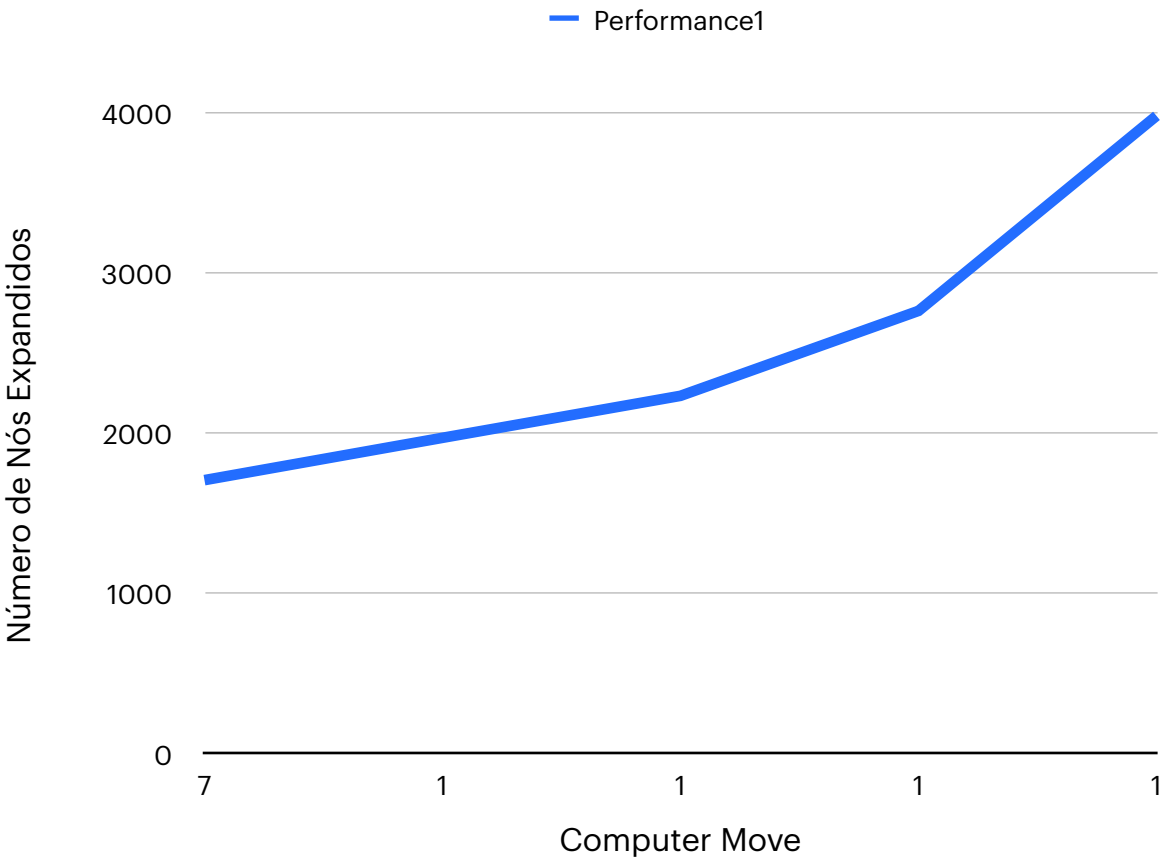


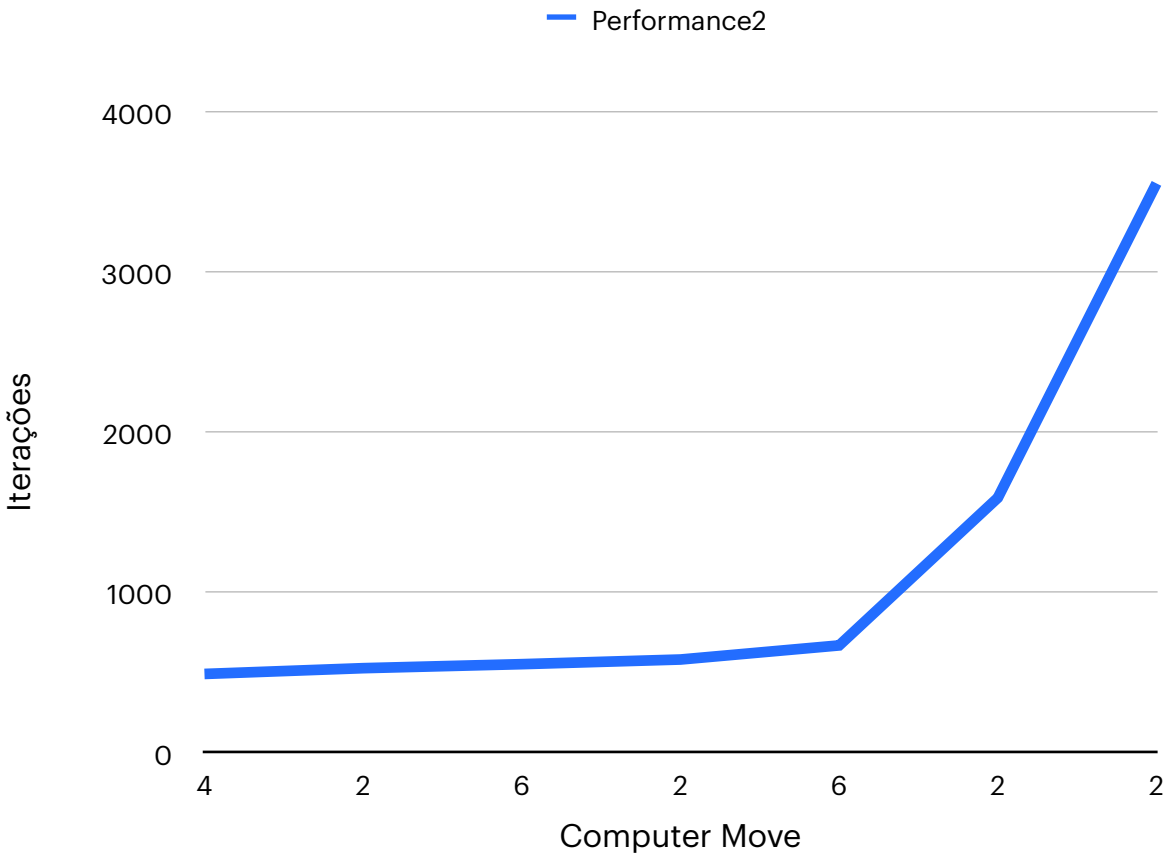


ALGORITMO ALPHA-BETA PURNING






Performance1
4000
3000
2000
1000
0
Número de Nós Expandidos
7
1
1
1
1
Computer Move


Performance2
4000
3000
2000
1000
0
Iterações
4
2
6
2
6
2
2
Computer Move

No gráfico onde avaliamos a Performance1, a minha jogada foi 1, 3, 5, 7, 6.

No gráfico onde avaliamos a Performance2, a minha jogada foi 1, 3, 5, 7, 6, 7, 7.

DESCRIÇÃO DOS ALGORITMOS NO CONTEXTO DO JOGO

ALGORITMO MINIMAX

Quando é a vez da Inteligência Artificial (IA) jogar, a função Minimax opera recebendo o estado do jogo atual e retornando a jogada a fazer dependendo da vez do jogador. Para isto usamos a função minimax( ), que chama a função Min( ) ou a função Max( ).

A função Max( ) recebe o maior valor de utilidade dos seus filhos, sendo que o valor dos seus filhos foi herdado pela função Min( ). Contrariamente, a função Min( ) recebe o menor valor de utilidade dos seus filhos, sendo que o valor dos seus filhos foi herdado pela função Max( ). Ou seja, estas funções serão chamadas recursivamente, alternando entre si.

Por cada chamada da função Min( ) ou Max( ), o atributo depth é decrementado e, quando depth iguala a 0, o algoritmo para e retorna a jogada.

ALGORITMO ALPHA-BETA PURNING

Este algoritmo segue a mesma lógica do algoritmo Minimax, porém acrescentamos as variáveis alpha (α), inicializada com “-infinito” e que, ao longo da execução do algoritmo, guarda a melhor jogada encontrada até o momento para o jogador maximizador e beta (β), inicializada a “+infinito” e que, ao longo da execução do algoritmo, guarda a melhor jogada encontrada até o momento para o jogador minimizador. Estas variáveis serão usadas para cortar a busca em ramos que não são relevantes para encontrar a melhor jogada.

Tanto o o algoritmo Alpha-beta purning como o algoritmo Minimax são pesquisas recursivas, pelo que não são usadas nenhumas estruturas de dados extra para guardar e representar os estados explorados.

ALGORITMO MONTE CARLO TREE SEARCH (MCTS)

Este algoritmo divide-se em duas classes.

Class Node

Os estados do tabuleiro são representados por nós que têm associado um número de visitas 'v' , de vitórias 'w', os seus descendentes 'children' e o nó pai 'parent'.

Os nós comparam-se entre eles através de métodos especiais __eq__ e __lt__ que avaliam o seu UCT dado por : (w/v)+math.sqrt(2)*math.sqrt(math.log(parent.v)/v) .

Funções relevantes:

f_uct( ): retorna o valor UCT de um determinado nó

get_highest_child( ): retorna o filho com maior valor UCT

expand_node( ): retorna todos os "filhos" possíveis de um nó

Class MCTS

Inicializa o algoritmo atribuindo à raiz da árvore (root) a configuração atual.

Funções relevantes:

selection( ): seleciona o nó a ser expandido ou simulado

simulation( ): é responsável pela simulação, isto é, realização de um jogo aleatório

back_propagate( ): onde, após a simulação, se dá a atualização dos nós parent até á root

play_according_to_UCT( ): retorna a próxima jogada a partir do root. Esta função tem em conta o valor de UCT dos filhos do root, selecionando o nó filho com maior valor de UCT.

play_according_to_visits( ): retorna a próxima jogada a partir do root. Esta função tem em conta o número de visitas dos filhos do root, selecionado o nó filho com maior valor ‘v’ (visitas).

A função montecarlo_search( ) combina as duas classes e, durante um determinado tempo predefinido, realiza a pesquisa e retorna a melhor jogada, após este tempo.

COMENTÁRIOS E CONCLUSÕES

- Em ambos os gráficos de avaliação do algoritmo Minimax e do algoritmo Alpha-Beta Purning, a profundidade definida foi 4 e a jogada executada por mim(Humano) foi 1, 1, 1, 1, 1.

- Nos algoritmos Minimax e Alpha-beta Purning aplicados ao jogo Connect-Four apresentamos dois gráficos, um que relaciona o movimento do computador com o número de nós expandidos e outro que relaciona o movimento do computador e o tempo de execução. Note que a variação é semelhante nos dois gráficos, porque o número de nós expandidos é proporcionalmente igual ao tempo de execução.

- O Algoritmo MTCS, para tempo de execução 2 segundos, não performa com um bom desempenho; defende algumas possíveis derrotas, mas é difícil determinar se é fruto da sorte ou de uma boa escolha. Achamos que o algoritmo está bem implementado, mas que, como 2 segundos de pesquisa apenas resultam entre [500-2000] iterações, esse valor é insuficiente para uma decisão robusta.

- Realizamos também uma comparação entre a jogada decidida através do seu valor UCT e o número de vezes que foi visitada na pesquisa. Concluímos, após várias tentativas, que a avaliação feita através do valor UCT é sensivelmente mais acertada.

- Das diferentes estratégias de busca que implementamos, a que se relevou mais eficaz, combinando a submissão da solução ótima com a rapidez desta submissão, foi o Alpha-Beta Purning.

BIBLIOGRAFIA

PowerPoints de Inteligência Artificial - Moodle

Artificial Intelligence, A Modern Approach - Third Edition/Problem-Solving/Chapter 6

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

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

https://en.wikipedia.org/wiki/Alpha–beta_pruning

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

R. L. Rivest, Game Tree Searching by Min/Max Approximation, AI 34 [1988], pp. 77-96