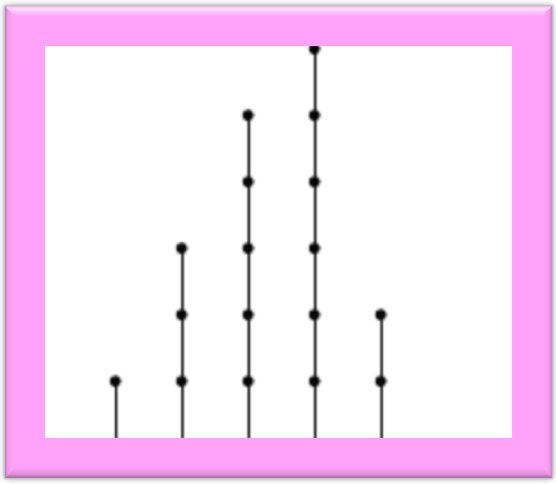
Universidade de Évora
Escola de Ciências e Tecnologias
Inteligência Artificial

3º Trabalho prático

Jogo do Nim e 4 em linha



Docente:
- Professora Irene Rodrigues

Discentes:
- Cláudia Dias nº35308
-João Queimado nº38176

Ano Letivo 2018/2019

Índice

Introdução

Este trabalho foi proposto na Unidade Curricular Inteligência Artificial no qual o objetivo era fazer dois jogos, o primeiro foi o Jogo do Nim e o segundo era à nossa escolha, sendo que optámos pelo jogo 4 em linha.

O Jogo do Nim é um jogo de dois jogadores que usam peças alinhadas em várias colunas. Neste caso o Utilizador irá jogar com o computador. Os jogadores realizam jogadas alternativas. O Jogador a realizar a última jogada ganha.

Existem as regras seguintes:

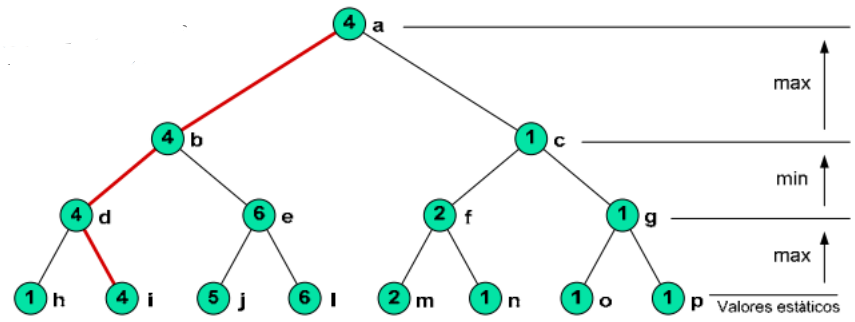
- Cada jogador retira um número qualquer de peças de uma única coluna (pode retirar no mínimo uma peça e no máximo todas as peças da coluna);
- Na mesma jogada o Jogador não pode retirar peças de colunas diferentes.

O segundo jogo 4 em linha é jogador também pelo Utilizador e pelo Computador que é jogado num tabuleiro 7x6 (7 colunas e 6 linhas) e o primeiro jogador a ter 4 peças da mesma cor seguidas na horizontal, na vertical ou na diagonal ganha. Se ninguém conseguir, a partida termina e o jogo resulta no empate.

Minmax

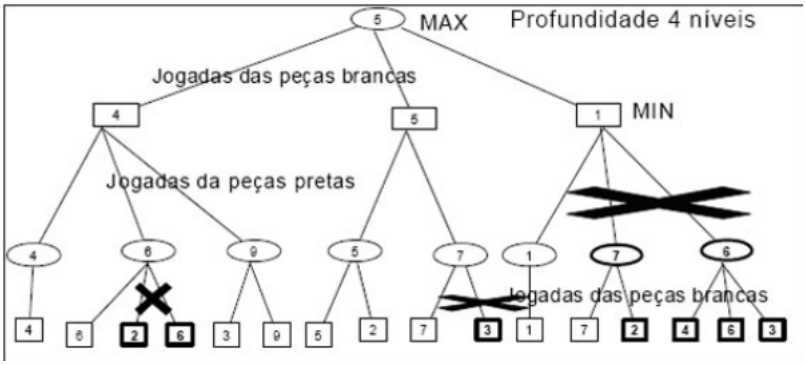
O algoritmo minmax é um algoritmo que se baseia na expansão em profundidade limitada de uma árvore onde cada folha é atribuída um valor de uma heurística.

O valor de cada nó intermédio é representado pelo valor máximo dos seus filhos, caso a profundidade do nó seja ímpar e o valor mínimo caso a profundidade do nó seja par.



Alfa-beta

O algoritmo alfa-beta é uma adaptação do algoritmo minmax onde é recorrido a alfas e betas de forma a reduzir o número de nós expandidos na árvore.



Jogo do Nim

Questão1

- Alínea a)

Escolha uma estrutura de dados para representar os estados do jogo.

Para representar os estados de jogo utilizamos uma estortura chamada "State" que contem uma lista de inteiros "tab" e um inteiro "player". A lista de inteiros representa o tabuleiro de jogo de um determinado estado, sendo que cada posição da lista representa uma estaca de madeira e cada valor representa o número de argolas presentes em cada estaca. O valor "player" representa o jogador que executou a jogada que deu origem ao estado.

```
class State:
    tab = []
    player = 0
    def __init__(self, varas):
        self.tab = varas
        self.player = 0
```

- Alínea b)

Defina o predicado terminal(estado) que sucede quando o estado e terminal.

Um determinado estado é terminal quando todas as argolas do tabuleiro tenham sido retiradas.

No caso do nosso problema é possível avaliar se um estado é terminal através da função “term_state” que dado um determinado estado e um jogador alvo retorna 0 se o estado não for terminal, -1 se o estado é terminal mas o jogador que originou o estado não corresponde ao jogador alvo(Derrota do jogador alvo) e 1 caso o estado seja terminal e o jogador que origina o estado seja o jogador alvo(Vitoria do jogador alvo).

```
def term_state(self, t_player):
    for e in self.tab:
        if e != 0:
            return 0
    if self.player == t_player:
        return 1
    else:
        return -1
```

- Alínea c)

Defina uma função de utilidade que para um estado terminal que deve retornar o valor do estado (ex: -1 perde, 0 empata, 1 ganha).

A função avaliadora de um estado procura atribuir um valor a um nó da arvore sendo este valor inversamente proporcional a distância desse nó ao nó terminal mais próximo.

Na nossa implementação do jogo do Nim a o valor de cada nó é dado pela função:

Valor = -(Sum(todos os nós da lista)) se o jogador for o jogador alvo

Valor = (Sum(todos os nós da lista)) se o jogador for diferente do jogador alvo

```python
from nim import *
from minmax_nim import *
from alfabetacut_nim import *
AI = P1
US = P2

if __name__ == "__main__":
    exp = int(input("nivel de expanção:"))
    alg = input("minmax ou alfabeta?")

    nV = int(input("numero de varas:"))
    fp = input("jogar primeiro:")

    V = []
    for i in range(nV):
        V.append(int(input("vara "+str(i)+":")))

    state = State(V)
    if alg == "minmax":
        tree = MinMaxTree( state )
    else:
        tree = AlfaBetaCut( state )

    if fp != "s":
        tree.minmax(exp, AI, AI)

        for i in tree.root.children:
            if i.value == tree.root.value:
                state = i.get_state()
                break

    f = state.term_state(AI)
    if f == 1:
        print("GANHEI")
    elif f == -1:
        print("Ganhas-te")

    while state.term_state(P1) == 0:
        while True:
            print("Joga")
            state.show()
            vara = int( input("vara a jogar:") )
            val = int(input("nos a tirar:"))
            if state.play(US, vara, val):
                break
            print("Jogada invalida")

        f = state.term_state(AI)
        if f == 1:
            print("GANHEI")
            break
        elif f == -1:
            print("Ganhas-te")
            break

        tree = MinMaxTree( state.cpy() )
        tree.minmax( exp, AI, AI )

        for i in tree.root.children:
            if i.value == tree.root.value:
                state = i.get_state()
                break

        f = state.term_state(AI)
        if f == 1:
            print("GANHEI")
            break
        elif f == -1:
            print("Ganhas-te")
            break
```

- Alínea d)

Use a implementação da pesquisa minimax dada na aula prática para escolher a melhor jogada num estado.

Como optamos por utilizar uma linguagem de programação diferente da linguagem dada nas aulas praticas não nos foi possível utilizar o algoritmo dado nas aulas praticas. No entanto desenvolvemos os nossos próprios algoritmos em python com base no algoritmo dado nas aulas.

```
def minmax_rec(self, node, prof, t_player, player, plim):

    state = node.get_state()
    c = state.term_state(t_player)

    inf = 1000

    if( c == 1 ):
        #state.show()
        #print( "win" )
        v = state.val(player, t_player)
        node.value = v
        return v

    if( c == -1):
        #state.show()
        #print( "lose" )
        v = state.val(player, t_player)
        node.val = v
        return v

    if( plim == prof ):
        #state.show()
        v = state.val(player, t_player)
        node.value = v
        return v

    node.expand( player )

    if( player == 1 ):
        p = 2
    else:
        p = 1

    l = []

    for n in node.children:
        v = self.minmax_rec( n, prof+1, t_player, p, plim)
        l.append( v )

    if( player == t_player ):
        val = max(l)
    else:
        val = min(l)

    node.value = val

    return val


def build_caminho(self, node):
    val = self.root.value
    node = self.root
    l = []
    while True:
        l.append( node )

        if len( node.children ) == 0:
            break

        for e in node.children:
            if e.value == val:
                node = e
                break
    return l
```

- Alínea e) e f)

Implemente a pesquisa Alfa-Beta e compare os resultados (tempo e espaço).

A implementação da pesquisa Alfa-Beta é baseada na pesquisa Min-Max, no entanto é adicionado o corte alfa beta que permite reduzir o número de nós expandidos na arvore assim diminuindo o tempo e o espaço ocupado em comparação com algoritmo Min-Max.

No nosso trabalho foram testados ambos os algoritmos com o mesmo input e verificou-se o número de nós visitados pelo algoritmo Min-Max até a solução ótima é muito superior a ao número de nós visitados pelo algoritmo Alfa-Beta.

Como testes optamos por limitar a expansão da arvore á profundidade 4 e o estado inicial utilizado contem uma argola na primeira estaca, duas na segunda, três na terceira e quatro na quarta. O número de nós visitados pelo algoritmo Min-Max foi de 2978 para as condições apresentadas anteriormente, enquanto que o algoritmo Alfa-Beta obteve a mesma solução em apenas 669 nós visitados.

Jogo 4 em linha

Questão 1 – estrutura de dados

Na representação de um estado utilizamos uma estortura “State” que contem uma lista de movimentos. Cada movimento é representado por um tuplo que contem dois inteiros, jogado e coluna, o valor do jogador representa o jogador que efetua o movimento e o valor da coluna representa a coluna onde foi inserida a peça.

A matriz bidimensional que representa o tabuleiro em um determinado estado e obtida pela função “generate_state” que recebendo um estado reconstrói através da lista de movimentos o tabuleiro.

```
class State:
    moves = []

    # constuc #
    def __init__ (self):
        self.moves = []
```

Questão 2 – terminal

Um certo estado é terminal se qualquer conjunto de 4 células adjacentes preenchidas pelo mesmo jogador e que formem uma linha horizontal, vertical ou diagonal.

No caso do nosso problema um estado final é avaliado pela função "term_state" que recebe um estado e devolve um booleano com o valor True se o estado recebido é final e False caso contrário.

```
def term_state(self):
    matrix = self.generate_state()

    c = self.check_lines(matrix)
    if c != 0:
        return c

    c = self.check_colums(matrix)
    if c != 0:
        return c

    c = self.check_diagonal_1(matrix)
    if c != 0:
        return c

    c = self.check_diagonal_2(matrix)
    if c != 0:
        return c

    return 0
```

Questão 3 – valores

No caso da nossa implementação do quatro em linha, a função de avaliação calcula o valor do nó através da expressão:

Valor = Sum ((casos que que podem originar vitoria) * (o numero de jogadas já preenchidas pelo jogador neste caso))

```
###################### Eval State ##########################

def val_four(self, l, player):
    for e in l:
        if e != player and e != EP:
            return 0
    if l.count(EP) == 0:
        return 0
    return l.count(player)

def val_lis(self, l, player):
    val = 0
    for i in range( len(l)-(WIN-1) ):
        val += self.val_four( l[i:(i+WIN)], player )
    return val

def val_lines(self, matrix, player ):
    val = 0
    for i in range(Y_Size):
        val += self.val_lis( self.get_line(i,matrix), player)
    return val

def val_colums(self, matrix, player ):
    val = 0
    for i in range(X_Size):
        val += self.val_lis( self.get_colum(i,matrix), player)
    return val

def val_diag_1(self, matrix, player ):
    val = 0
    for i in range(X_Size + Y_Size - 1):
        val += self.val_lis( self.get_diag_bot_up(i,matrix), playe
    return val

def val_diag_2(self, matrix, player ):
    val = 0
    for i in range(X_Size + Y_Size - 1):
        val += self.val_lis( self.get_diag_up_bot(i,matrix), playe
    return val

def val(self, player, t_player):
    val = 0
    matrix = self.generate_state()
    val += self.val_lines( matrix, player)
    val += self.val_colums( matrix, player)
    val += self.val_diag_1( matrix, player)
    val += self.val_diag_2( matrix, player)

    if player != t_player:
        return -val

    return val
```

Questão 4 – Agente inteligente com minmax e alfa-beta

A nossa implementação do agente inteligente utiliza os algoritmos apresentados anteriormente para escolher a sua próxima jogada.

Ao iniciar o agente é pedido ao utilizador a profundidade máxima da arvore que o agente vai utilizar, isto é o numero de máximo de jogadas consecutivas que o agente vai prever. Caso o utilizador queira retirar esta limitação basta passar o valor -1 no input fazendo com que o agente preveja todas as jogadas possíveis até atingir um estado terminal.

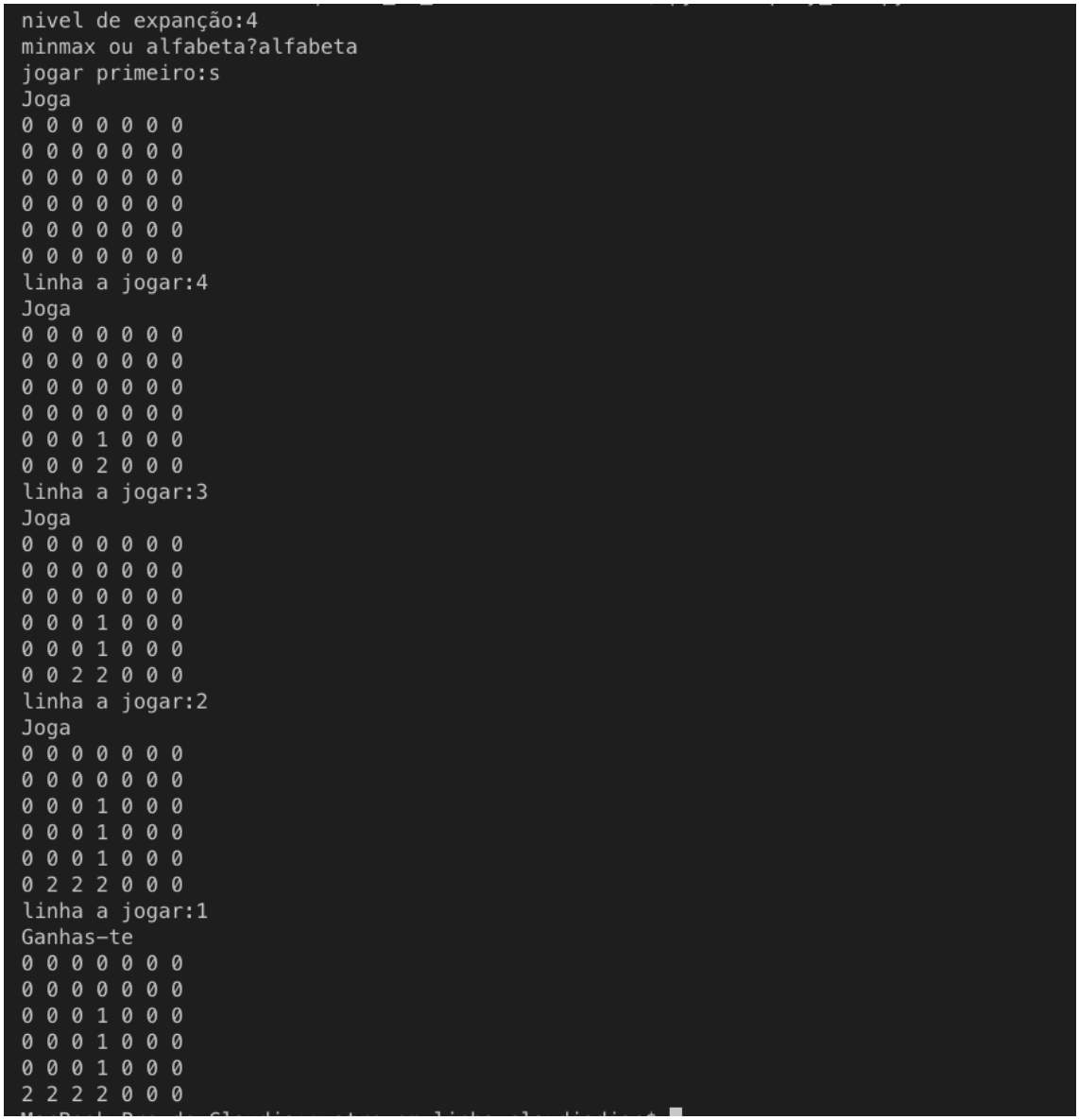
Alem do limite de profundidade, é também pedido ao utilizador pelo algoritmo que irá ser utilizado pelo agente.

```python
from four_in_line import *
from minmax_4il import *
from alfabetacut_4il import *

AI = P1
US = P2

if __name__ == "__main__":
    exp = int(input("nivel de expanção:"))
    alg = input("minmax ou alfabeta?")

    fp = input("jogar primeiro:")

    state = State()
    if alg == "minmax":
        tree = MinMaxTree( state )
    else:
        tree = AlfaBetaCut( state )

    if fp != "s":
        tree.minmax(exp, AI, AI)

        for i in tree.root.children:
            if i.value == tree.root.value:
                state = i.get_state()
                break

    f = state.term_state(AI)
    if f == 1:
        print("GANHEI")
    elif f == -1:
        print("Ganhas-te")

    while state.term_state(P1) == 0:
        matrix = state.generate_state()
        while True:
            print("Joga")
            state.show()
            vara = int( input("linha a jogar:") )
            vara -= 1
            if state.play( matrix, US, vara ):
                state.add_move(US, vara)
                break
            print("Jogada invalida")

        f = state.term_state(AI)
        if f == 1:
            print("GANHEI")
            break
        elif f == -1:
            print("Ganhas-te")
            break

        tree = MinMaxTree( state.cpy() )
        tree.minmax( exp, AI, AI )

        for i in tree.root.children:
            if i.value == tree.root.value:
                state = i.get_state()
                break

        f = state.term_state(AI)
        if f == 1:
            print("GANHEI")
            break
        elif f == -1:
            print("Ganhas-te")
            break
    state.show()
```

Output minmax

```
MacBook-Pro-de-Claudia:quatro_em_linha claudiadias$ python3 play_4il.py
nivel de expanção:4
minmax ou alfabeta?minmax
jogar primeiro:s
Joga
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
linha a jogar:4
Joga
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 1 0 0 0
0 0 0 2 0 0 0
linha a jogar:3
Joga
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 1 0 0 0
0 0 0 1 0 0 0
0 0 2 2 0 0 0
linha a jogar:2
Joga
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 1 0 0 0
0 0 0 1 0 0 0
0 0 0 1 0 0 0
0 2 2 2 0 0 0
linha a jogar:1
Ganhas-te
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 1 0 0 0
0 0 0 1 0 0 0
0 0 0 1 0 0 0
2 2 2 2 0 0 0
MacBook-Pro-de-Claudia:quatro_em_linha claudiadias$
```

Output alfabeta



Conclusão

Em suma, conseguimos implementar os dois trabalhos propostos, superando algumas dificuldades.

Este trabalho foi desenvolvido em linguagem Python, sendo esta uma linguagem mais fácil para nós. Optámos por escolher esta linguagem porque tivemos muitas dificuldades em fazer com linguagem Prolog.