Universidade Federal de São Carlos
Departamento de Computação
Engenharia de Computação

Inteligência Artificial - 22705 - Turma B

Primeiro Trabalho Prático: Buscas

Prof. Dr. Ricardo Cerri

Data de entrega: 16/11/2018

Leonardo Utida Alcantara - RA: 628182
Leonardo Tavares Oliveira - RA: 628174
Tiago Bachiega de Almeida - RA: 628247

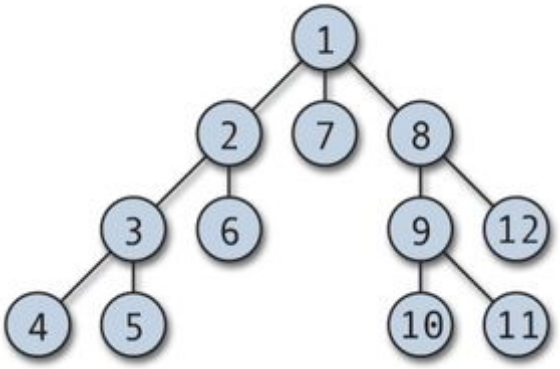
16 de Novembro de 2018
São Carlos

Busca não informada

Para a implementação do código que soluciona o Sudoku, foi armazenado o tabuleiro não resolvido em uma matriz, atribuindo-se o valor 0 às posições livres para serem preenchidas pelo jogador e o valor de cada respectivo campo para as posições ocupadas. O programa percorre, para cada posição livre os valores de 1 a 9, tentando encaixá-los na posição atual do tabuleiro, de acordo com as regras do jogo. Para encontrar a solução que resolve o tabuleiro com busca não informada foi utilizada uma busca em profundidade.

Busca em profundidade consiste em uma busca de forma iterativa entre os nós de uma árvore a partir de um nó inicial. Ou seja, dado um nó, ele deve escolher um nó filho a este e descer por ele tentando encontrar a solução. Caso não a acha neste segundo nó, deve descer por seu filho e assim por diante até que chegue a um nó sem filhos. Neste ponto, o algoritmo volta ao último nó que encontrou e que possuía filhos e desce pelo seu próximo filho, ainda não explorado. Este movimento de percorrer sempre um nó abaixo do atual e não o vizinho que dá a ideia de profundidade a esta busca. A Figura 1 é um exemplo de árvore onde estão assinalados a ordem em que os nós seriam percorridos com este tipo de busca.

Figura 1: Busca em Profundidade



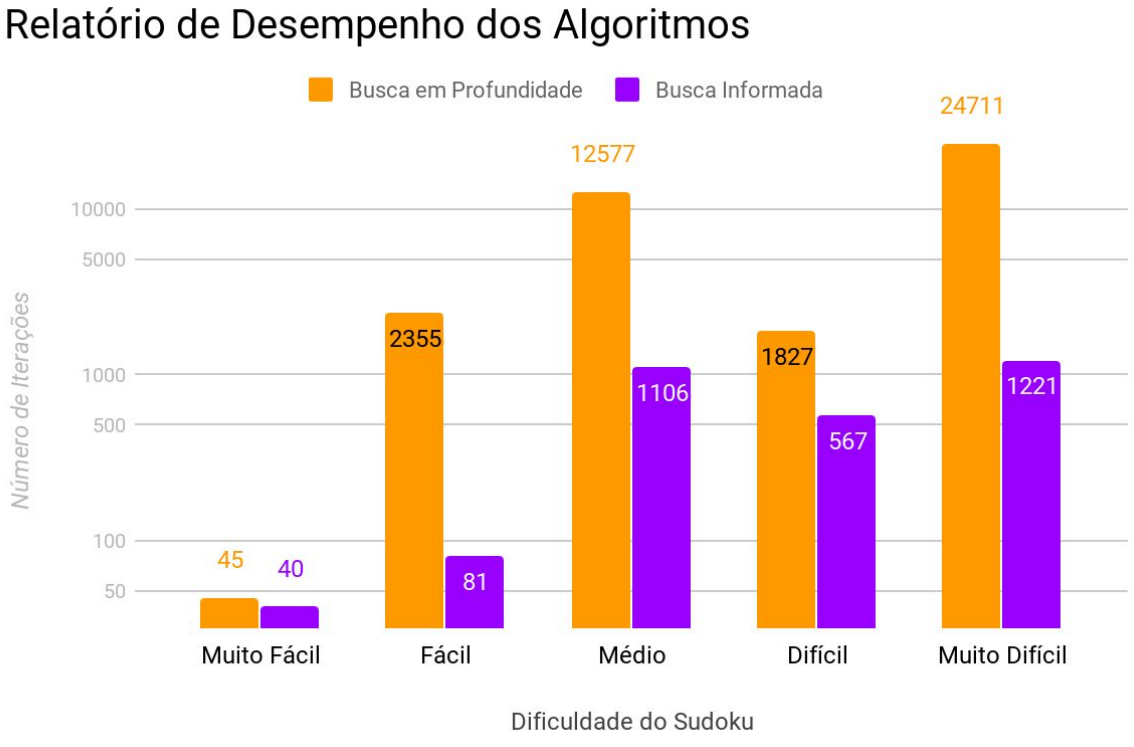
Para o caso do Sudoku, cada campo a ser preenchido com algum número corresponde a um nó, sendo o nó inicial o campo superior esquerdo. Quando o algoritmo chega a um nó ele tenta preenchê-lo com algum número de 1 a 9. Se ele conseguir, desce para o nó filho e tenta novamente. Caso ele não consiga no nó seguinte, é um indicativo que a escolha feita em algum nó acima não foi boa, então ele retorna para o nó anterior e tenta outro número. Caso no primeiro nó o programa indique que não foi possível encontrar uma solução, o tabuleiro em questão não tem solução possível. A descida pelos nós foi implementada como uma recursão: sempre que é escolhido um valor, o programa chama recursivamente a função para se resolver o tabuleiro com os valores atualizados.

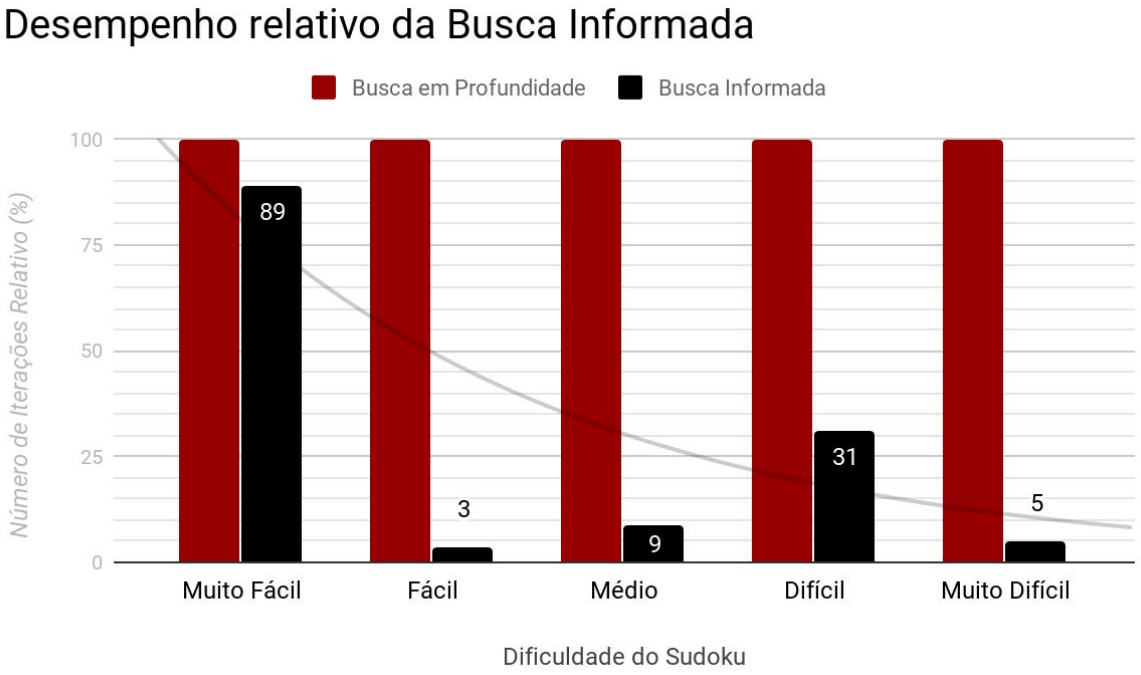
Busca informada

De forma a reduzir o número de iterações necessárias para a resolução do sudoku foi implementada uma versão modificada da busca em profundidade com abordagem *greedy,* de forma que para acelerar a convergência do algoritmo seja sempre feita uma escolha ótima.

Para realização de tal escolha são usadas 3 funções auxiliares, uma que retorna a linha mais cheia, outra que retorna a coluna mais cheia e por fim uma que retorna a seção mais cheia, assim como a quantidade de elementos em tal estrutura. Diferente da busca não informada, neste caso o algoritmo deve se ater a tentar resolver a região mais cheia da iteração atual até que ela esteja resolvida se possível, e só depois procurar outra região de interesse.

Para melhor visualização da otimização causada pela mudança do algoritmo não informado para informado, foram feitos os seguintes gráficos, um absoluto e outro relativo, comparando ambos os tipos de busca. Vale ressaltar que, para facilitar a manipulação das listas, foi utilizado o *numpy,* sendo necessária a sua instalação para a execução do código.





É possível ver, portanto, que o algoritmo de busca informada possui resultado superior ao não informado, obtendo reduções relativas entre 11 e 97%. Além disso, a tendência (como pode ser vista no gráfico de Desempenho relativo) é que haja redução do número relativo de iterações.

Códigos Fonte

A seguir estão os códigos fonte de ambas as buscas implementadas. Além dos detalhes explicados anteriormente, no que diz respeito à execução do código, nota-se a variável *debug.* Esta variável simplesmente insere um delay de 0.5 segundo entre cada iteração se estiver como *True*. Caso esteja como *False*, não há o delay. Ela foi feita apenas para conseguirmos alterar facilmente entre estado de depuração e de execução para a obtenção de resutados.

Busca em Profundidade

```
#|-------------------------------------------------------#
#|Trabalho de buscas - Inteligência artificial          #
#|Grupo:                                                #
#|Leonardo Utida Alcantara 628182                       #
#|Leonardo Tavares Oliveira 628174                      #
#|Tiago Bachiega de Almeida 628247                      #
#|-------------------------------------------------------#
#|Resolucao de um jogo Sudoku utilizando a busca em     #
#|profundidade (não informada)                          #
#|-------------------------------------------------------#
```

```
# A ideia do algoritmo é que a cada iteração ele procura por um espaço vazio (representado por valores de 0)
# Essa busca é sempre feita da esquerda para a direita e de cima para baixo no tabuleiro.
# Após um valor ser encontrado, o algoritmo entra em um loop for que testa todos os 9 possíveis valores para
# aquele espaço, sempre verificando se aquela jogada é possível.
#
# Caso o a jogada (combinação de valor e posição) seja possível, o algoritmo atribui o novo valor para a
# posição e chama a função de resolver o  sudoku recursivamente no novo tabuleiro.
# Quando uma jogada não pode ser feita, o algoritmo tenta o próximo valor possível para aquela posição.
# Caso nenhum valor seja possível, o algoritmo retorna falso para a função acima na recusão indicando que
# o caminho escolhido não leva ao resultado final e este tenta outro valor.
# Caso o valor Falso retorne, para todas as oções, até a primeira chamada da função que resolve o jogo,
# o algoritmo final retorna falso indicando que não foi possível resolver o jogo.


from os import system, name
from time import sleep

#DEBUG
debug = False

# Conta o número de iterações até ser resolvido o jogo
numIter = 0

# Funcao de limpar tela
def limpaTela():
    # Windows
    if name == 'nt':
        _ = system('cls')
    # mac e linux
    else:
        _ = system('clear')

# Função que recebe um tabuleior na forma de listas e printa na tela de um jeito mais fácil de ver
def printaTabuleiro(tabuleiro):
    print("+" + "---+"*9)
    for i, row in enumerate(tabuleiro):
         print(("|" + " {}   {}   {} |"*3).format(*[x if x != 0 else " " for x in row]))
        if i % 3 == 2:
            print("+" + "---+"*9)
```

```
        else:
            print("+" + "   +"*9)

# Função que acha a próxima possição vazia do tabuleiro para realizar a próxima jogada
def achaProxPosVazia(tabuleiro):
    for i in range(0,9):
        for j in range(0,9):
            if tabuleiro[i][j] == 0:
                return i,j
    return -1,-1

# Função que verifica se a jogada é possivel
# Verifica 3 condições:
### Se não há o valor na linha
### Se não há o valor na coluna
### Se não há o valor na seção (quadrado)
def valorLocalValido(tabuleiro, row, col, val):
    # Verifica se não há o valor da jogada na linha
    for j in range(9):
        if tabuleiro[row][j] == val:
            return False
    # Verifica se não há o valor da jogada na coluna
    for i in range(9):
        if tabuleiro[i][col] == val:
            return False

    # Verifica se não há o valor da jogada na seção
    secLinInicial = (row//3) * 3
    secColInicial = (col//3) * 3
    for i in range(secLinInicial, secLinInicial+3):
        for j in range(secColInicial, secColInicial+3):
            if tabuleiro[i][j] == val:
                return False
    # Se nao encontrou nenhum problema retorna verdadeiro
    return True

# Função se resuloção do jogo que é chamada recursivamente
# Recebe o tabuleiro e o tempo de espera para melhor visualização
def resolverSudoku(tabuleiro, tempoEspera):

    #variavel global para o número de iterações
    global numIter

    #Mais uma iteração
    numIter = numIter + 1

    limpaTela()
```

```
    printaTabuleiro(tabuleiro)
    # Espera 0.5 segundos para a próxima jogada para melhor visualização
    sleep(tempoEspera)

    row,col = achaProxPosVazia(tabuleiro)
    if row == -1 or col == -1: #nao ha mais valores 0, o jogo foi concluído
        return True

    #Testa todos os 9 possíveis valores começando em 0
    for val in range(1,10):
        # verifica se a jogada é valida
        if(valorLocalValido(tabuleiro, row, col, val)):
            # Se a jogada é valida, atualiza o tabuleiro e resolve o novo jogo
            tabuleiro[row][col] = val
            if(resolverSudoku(tabuleiro, tempoEspera)):
                return True
             # Se um falso foi retornado desfaz a última jogada e tenta o proximo
valor
            tabuleiro[row][col] = 0
    # Se nenhum valor foi encontrado, retorna falso, indicando que não eh possivel
resolver
    return False


muitoFacil = [
[1, 0, 0, 5, 0, 9, 6, 7, 0],
[4, 0, 2, 8, 7, 0, 0, 1, 9],
[9, 6, 7, 1, 4, 3, 2, 8, 5],
[2, 0, 4, 9, 0, 0, 0, 0, 0],
[5, 0, 6, 0, 0, 0, 0, 0, 8],
[0, 0, 0, 0, 1, 0, 4, 2, 0],
[0, 0, 0, 3, 6, 2, 7, 4, 0],
[6, 4, 0, 0, 0, 1, 0, 5, 0],
[0, 2, 1, 4, 0, 8, 9, 3, 6],
]

facil = [
[1, 2, 9, 0, 6, 0, 0, 0, 3],
[0, 0, 0, 0, 0, 0, 0, 0, 9],
[0, 0, 0, 3, 5, 0, 0, 0, 8],
[0, 0, 0, 8, 0, 3, 9, 0, 0],
[3, 0, 1, 0, 0, 0, 5, 0, 2],
[0, 0, 6, 1, 0, 5, 0, 0, 0],
[4, 0, 0, 0, 7, 6, 0, 0, 0],
[2, 0, 0, 0, 0, 0, 0, 0, 0],
[9, 0, 0, 0, 8, 0, 1, 3, 4],
]
```

```
medio = [
[6, 0, 1, 0, 0, 0, 5, 0, 8],
[0, 0, 0, 0, 8, 0, 0, 0, 0],
[0, 0, 0, 5, 0, 2, 0, 0, 0],
[2, 0, 8, 1, 0, 4, 6, 0, 3],
[1, 0, 0, 3, 0, 7, 0, 0, 4],
[0, 0, 3, 0, 0, 0, 7, 0, 0],
[7, 6, 0, 0, 0, 0, 0, 1, 2],
[0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 2, 0, 6, 0, 8, 0, 4, 0],
]

dificil = [
[0, 0, 7, 5, 0, 1, 9, 0, 0],
[0, 0, 8, 0, 6, 0, 4, 0, 0],
[0, 5, 0, 8, 0, 4, 0, 7, 0],
[0, 0, 2, 0, 7, 0, 3, 0, 0],
[4, 0, 0, 0, 8, 0, 0, 0, 9],
[0, 3, 0, 0, 1, 0, 0, 6, 0],
[1, 9, 0, 0, 0, 0, 0, 5, 4],
[0, 7, 0, 0, 0, 0, 0, 3, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0],
]

muitoDificil = [
[4, 0, 0, 0, 0, 0, 0, 0, 7],
[2, 0, 0, 0, 1, 0, 0, 0, 4],
[0, 1, 0, 0, 0, 0, 0, 5, 0],
[5, 0, 1, 0, 4, 0, 7, 0, 6],
[0, 3, 0, 1, 0, 8, 0, 4, 0],
[0, 0, 0, 6, 0, 3, 0, 0, 0],
[0, 0, 2, 0, 0, 0, 4, 0, 0],
[7, 0, 0, 0, 8, 0, 0, 0, 9],
[0, 5, 0, 0, 2, 0, 0, 7, 0],
]

#VARIAVEIS DE DEBUG
if (debug):
	sleepTime = 0.0
else:
	sleepTime = 0.5

meuSudoku = []

# Opcao do usuario
opcao = 0

limpaTela()
```

```
# Menu inicial
print("**********SUDOKU IA**********")
print("\n\nSelecione o tipo de jogo:")
print("1 - Rodar exemplo Muito Fácil")
print("2 - Rodar exemplo Fácil")
print("3 - Rodar exemplo Médio")
print("4 - Rodar exemplo Difícil")
print("5 - Rodar exemplo Muito Difícil")
print("6 - Inserir um Sudoku")

#Le opção
opcao = int(input())

#Trata cada opção
if opcao == 1:
     print(resolverSudoku(muitoFacil, sleepTime), "\n", "Número de iterações: ",
numIter)
elif opcao == 2:
       print(resolverSudoku(facil, sleepTime), "\n", "Número de iterações: ",
numIter)
elif opcao == 3:
       print(resolverSudoku(medio, sleepTime), "\n", "Número de iterações: ",
numIter)
elif opcao == 4:
      print(resolverSudoku(dificil, sleepTime), "\n", "Número de iterações: ",
numIter)
elif opcao == 5:
    print(resolverSudoku(muitoDificil, sleepTime), "\n", "Número de iterações: ",
numIter)
elif opcao == 6:
    limpaTela()
    print("Digite seu Sudoku, os elementos devem ser números de 0 a 9, sendo 0 uma
casa vazia\n")

    # Lê os inputs do usuários
    for i in range(0, 9):
        elementos = list(map(int, input().split()))

        #Condições do sudoku
        while(len(elementos) != 9):
            print("Cada fileira deve ter 9 elementos")
            elementos = list(map(int, input().split()))
        while(i > 9 or i < 0 for i in elementos):
            print("Os elementos devem ser de 0 a 9")
            elementos = list(map(int, input().split()))
        meuSudoku.append(elementos)
```

```
    print(resolverSudoku(meuSudoku, sleepTime), "\n", "Número de iterações: ",
numIter)
```

Busca Informada

```
#|-------------------------------------------------------#
#|Trabalho de buscas - Inteligência artificial          #
#|Grupo:                                                #
#|Leonardo Utida Alcantara 628182                       #
#|Leonardo Tavares Oliveira 628174                      #
#|Tiago Bachiega de Almeida 628247                      #
#|-------------------------------------------------------#
#|Resolucao de um jogo Sudoku utilizando a busca        #
#|informada                                             #
#|-------------------------------------------------------#


# A ideia do algoritmo é que a cada iteração ele procura por um espaço vazio
(representado por valores de 0)
# Essa busca é sempre feita da esquerda para a direita e de cima para baixo no
tabuleiro.
# Após um valor ser encontrado, o algoritmo entra em um loop for que testa todos
os 9 possíveis valores para
# aquele espaço, sempre verificando se aquela jogada é possível.
#
# Caso o a jogada (combinação de valor e posição) seja possível, o algoritmo
atribui o novo valor para a
# posição e chama a função de resolver o  sudoku recursivamente no novo tabuleiro.
# Quando uma jogada não pode ser feita, o algoritmo tenta o próximo valor possível
para aquela posição.
# Caso nenhum valor seja possível, o algoritmo retorna falso para a função acima
na recusão indicando que
# o caminho escolhido não leva ao resultado final e este tenta outro valor.
# Caso o valor Falso retorne, para todas as oções, até a primeira chamada da
função que resolve o jogo,
# o algoritmo final retorna falso indicando que não foi possível resolver o jogo.
#
# O uso de busca informada para o caso consiste em uma verificação, sempre que o
programa for tentar
# resolver o novo tabuleiro, de qual de suas regiões está mais completa, ou seja,
possui mais valores
# preenchidos. Essas região pode ser uma linha, uma coluna ou uma seção do
tabuleiro. Diferente da
# busca não informada, neste caso o algoritmo deve se ater a tentar resolver a
região mais cheia da
# iteração atual até que ela esteja resolvida se possível, e só depois procurar
outra região de interesse
```

```python
from os import system, name
from time import sleep
import numpy as np

#DEBUG
debug = False

# Conta o número de iterações até ser resolvido o jogo
numIter = 0

# Funcao de limpar tela
def limpaTela():
    # Windows
    if name == 'nt':
        _ = system('cls')
    # mac e linux
    else:
        _ = system('clear')

# Função que recebe um tabuleior na forma de listas e printa na tela de um jeito
mais fácil de ver
def printaTabuleiro(tabuleiro):
    print("+" + "---+"*9)
    for i, row in enumerate(tabuleiro):
         print(("|" + " {}   {}   {} |"*3).format(*[x if x != 0 else " " for x in
row]))
        if i % 3 == 2:
            print("+" + "---+"*9)
        else:
            print("+" + "   +"*9)

# Função que acha a próxima possição vazia do tabuleiro para realizar a próxima
jogada
def achaProxPosVazia(tabuleiro, tipoRegiao):

    # Se a região for uma linha, vai realizar um tratamento
    # no retorno para retornar um valor mais facilmente
    # manipulável pelo resto do programa.
    # Neste caso, vai retornar 0 na posicao que indica a linha pois
    # como ja temos uma lista com a linha, nao é necessário indexá-la
    # e vai retornar i na posição da coluna pois precisamos indexar
    # os elementos da linha.
    # O mesmo raciocínio é aplicado aos demais casos.
    if tipoRegiao == "Linha":
        rows = len(tabuleiro)
        for i in range(rows):
            if tabuleiro[i] == 0:
                return 0,i
```

```
    if tipoRegiao == "Coluna":
        cols = len(tabuleiro)
        for i in range(cols):
            if tabuleiro[i] == 0:
                return i,0
    if tipoRegiao == "Secao":
        rows = len(tabuleiro)
        cols = len(tabuleiro[0])
        for i in range(rows):
            for j in range(cols):
                if tabuleiro[i][j] == 0:
                    return i,j

    return -1,-1

# Função que verifica se a jogada é possivel
# Verifica 3 condições:
### Se não há o valor na linha
### Se não há o valor na coluna
### Se não há o valor na seção (quadrado)
def valorLocalValido(tabuleiro, row, col, val):
    # Verifica se não há o valor da jogada na linha
    for j in range(9):
        if tabuleiro[row][j] == val:
            return False
    # Verifica se não há o valor da jogada na coluna
    for i in range(9):
        if tabuleiro[i][col] == val:
            return False

    # Verifica se não há o valor da jogada na seção
    secLinInicial = (row//3) * 3
    secColInicial = (col//3) * 3
    for i in range(secLinInicial, secLinInicial+3):
        for j in range(secColInicial, secColInicial+3):
            if tabuleiro[i][j] == val:
                return False
    # Se nao encontrou nenhum problema retorna verdadeiro
    return True


#Encontra a coluna mais cheia
def colunaMaisCheia(tabuleiro):
    maxCol = 0
    idxCol = 0
    numElemCol = 0

    rows = len(tabuleiro)
```

```
    cols = len(tabuleiro[0]);

    #procura a coluna com mais elementos
    #diferentes de zeo no tabuleiro
    for i in range(cols):
        numElemCol = 0
        for j in range(rows):
            if tabuleiro[j][i] != 0:
                numElemCol += 1
        if numElemCol > maxCol and numElemCol < 9:
            maxCol = numElemCol
            idxCol = i

    regiao = np.asarray(tabuleiro)

    # retorna o numero de elementos da coluna, a sublista da coluna
    # o indice da coluna no array e a linha inicial da coluna, sempre 0
    return maxCol, regiao[:,idxCol], idxCol, 0

#Encontra a linha mais cheia
def linhaMaisCheia(tabuleiro):
    maxRow = 0
    idxRow = 0
    numElemRow = 0

    rows = len(tabuleiro)
    cols = len(tabuleiro[0]);

    #procura a linha com mais elementos
    #diferentes de zeo no tabuleiro
    for i in range(rows):
        numElemRow = 0
        for j in range(cols):
            if tabuleiro[i][j] != 0:
                numElemRow += 1
        if numElemRow > maxRow and numElemRow < 9:
            maxRow = numElemRow
            idxRow = i

    regiao = np.asarray(tabuleiro)

    # Retorna o número de elementos da linha, a sublista da linha
    # o índice da linha e o índice da primeira coluna da linha, sempre 0
    return maxRow, regiao[idxRow,:], idxRow, 0

#Encontra a area mais cheia
def secaoMaisCheia(tabuleiro):
    maxSec = 0
```

```
    numElemSec = 0
    secCol = 0
    secRow = 0
    idxSecCol = 0
    idxSecRow = 0

    rows = len(tabuleiro)
    cols = len(tabuleiro[0])

    #percorre todas as secoes do tabuleiro
    while(secRow < rows):
        numElemSec = 0
        #soma os elementos da secao
        for j in range(secRow, secRow + 3):
            for k in range(secCol, secCol + 3):
                if tabuleiro[j][k] != 0:
                    numElemSec += 1
        #busca a secao com mais elementos
        if numElemSec > maxSec and numElemSec < 9:
            maxSec = numElemSec
            idxSecCol = secCol
            idxSecRow = secRow

        #coluna onde se inicia a proxima secao
        secCol += 3
        #se percorreu todas as secoes da linha
        #vai para a proxima linha com secoes
        if secCol == 9:
            secCol = 0
            secRow += 3

    regiao = np.asarray(tabuleiro)

    # Retorna o número de elementos da seção, a sublista com a seção
    # e o índice superior esquerdo da seção no tabuleiro
      return maxSec, regiao[idxSecRow:idxSecRow + 3, idxSecCol:idxSecCol + 3],
idxSecRow, idxSecCol

# Entre todas as regiões possíveis do tabuleiro
# retorna informações daquela que está mais cheia
def regiaoMaisCheia(tabuleiro):
    maxRegiao = 0 # numero de posições preencidas da região
    idxColReg = 0 # índice de coluna da região se for o caso, se não é 0
    idxRowReg = 0 # índice de linhas da região se for o caso, se não é 0
    regiao = []   # lista que armazenará a região do tabuleiro
    tipoRegiao = "Nenhuma" # string que informa qual é o tipo da região

    # verifica a coluna mais cheia
```

```
    maxCol, regCol, idxColCol, _ = colunaMaisCheia(tabuleiro)
    # verifica a linha mais cheia
    maxRow, regRow, idxRowRow, _ = linhaMaisCheia(tabuleiro)
    # verifica a seção mais cheia
    maxSec, regSec, idxRowSec, idxColSec = secaoMaisCheia(tabuleiro)

    # entre as 3 possibilidades acima, escolhe a mais cheia de todas
    if maxCol > maxRegiao:
        maxRegiao = maxCol
        tipoRegiao = "Coluna"
        regiao = regCol
        idxRowReg = 0
        idxColReg = idxColCol
    if maxRow > maxRegiao:
        maxRegiao = maxCol
        tipoRegiao = "Linha"
        regiao = regRow
        idxRowReg = idxRowRow
        idxColReg = 0
    if maxSec > maxRegiao:
        maxRegiao = maxSec
        tipoRegiao = "Secao"
        regiao = regSec
        idxRowReg = idxRowSec
        idxColReg = idxColSec

    # Retorna o tipo da região, a lista com os elementos da região
    # o índice de linha e coluna da região em questão
    return tipoRegiao, regiao, idxRowReg, idxColReg

# Função se resuloção do jogo que é chamada recursivamente
# Recebe o tabuleiro e o tempo de espera para melhor visualização
def resolverSudoku(tabuleiro, tempoEspera):

    #variavel global para o número de iterações
    global numIter

    #Mais uma iteração
    numIter = numIter + 1

    limpaTela()
    printaTabuleiro(tabuleiro)

    #procura a regiao mais cheia do tabuleiro
    tipoRegiao, regiao, regRow, regCol = regiaoMaisCheia(tabuleiro)

    #terminou o jogo
    if tipoRegiao == "Nenhuma":
```

```
        return True

    #acha proxima vazia
    row,col = achaProxPosVazia(regiao, tipoRegiao)

    #posicoes equivalentes
    eqvRow = row + regRow
    eqvCol = col + regCol

    #prints
    print("tipo de regiao : " + tipoRegiao)
    print(regiao)
    print("Pos vazia na regiao: " + str(row) + " " + str(col))
       print("Pos inicial da regiao no tabuleiro: " + str(regRow) + " " +
str(regCol))
    print("Pos equivalente no tabuleiro: " + str(eqvRow) + " " + str(eqvCol))

      # Espera 1.0 segundos para a próxima jogada para melhor visualização
    sleep(tempoEspera)

    #Testa todos os 9 possíveis valores começando em 0
    for val in range(1,10):
        # verifica se a jogada é valida
        if(valorLocalValido(tabuleiro, eqvRow, eqvCol, val)):
            # Se a jogada é valida, atualiza o tabuleiro e resolve o novo jogo
            tabuleiro[eqvRow][eqvCol] = val
            if(resolverSudoku(tabuleiro, tempoEspera)):
                return True
             # Se um falso foi retornado desfaz a última jogada e tenta o proximo
valor
            tabuleiro[eqvRow][eqvCol] = 0
    # Se nenhum valor foi encontrado, retorna falso, indicando que não eh possivel
resolver
    return False


muitoFacil = [
[1, 0, 0, 5, 0, 9, 6, 7, 0],
[4, 0, 2, 8, 7, 0, 0, 1, 9],
[9, 6, 7, 1, 4, 3, 2, 8, 5],
[2, 0, 4, 9, 0, 0, 0, 0, 0],
[5, 0, 6, 0, 0, 0, 0, 0, 8],
[0, 0, 0, 0, 1, 0, 4, 2, 0],
[0, 0, 0, 3, 6, 2, 7, 4, 0],
[6, 4, 0, 0, 0, 1, 0, 5, 0],
[0, 2, 1, 4, 0, 8, 9, 3, 6],
]
```

```
facil = [
[1, 2, 9, 0, 6, 0, 0, 0, 3],
[0, 0, 0, 0, 0, 0, 0, 0, 9],
[0, 0, 0, 3, 5, 0, 0, 0, 8],
[0, 0, 0, 8, 0, 3, 9, 0, 0],
[3, 0, 1, 0, 0, 0, 5, 0, 2],
[0, 0, 6, 1, 0, 5, 0, 0, 0],
[4, 0, 0, 0, 7, 6, 0, 0, 0],
[2, 0, 0, 0, 0, 0, 0, 0, 0],
[9, 0, 0, 0, 8, 0, 1, 3, 4],
]

medio = [
[6, 0, 1, 0, 0, 0, 5, 0, 8],
[0, 0, 0, 0, 8, 0, 0, 0, 0],
[0, 0, 0, 5, 0, 2, 0, 0, 0],
[2, 0, 8, 1, 0, 4, 6, 0, 3],
[1, 0, 0, 3, 0, 7, 0, 0, 4],
[0, 0, 3, 0, 0, 0, 7, 0, 0],
[7, 6, 0, 0, 0, 0, 0, 1, 2],
[0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 2, 0, 6, 0, 8, 0, 4, 0],
]

dificil = [
[0, 0, 7, 5, 0, 1, 9, 0, 0],
[0, 0, 8, 0, 6, 0, 4, 0, 0],
[0, 5, 0, 8, 0, 4, 0, 7, 0],
[0, 0, 2, 0, 7, 0, 3, 0, 0],
[4, 0, 0, 0, 8, 0, 0, 0, 9],
[0, 3, 0, 0, 1, 0, 0, 6, 0],
[1, 9, 0, 0, 0, 0, 0, 5, 4],
[0, 7, 0, 0, 0, 0, 0, 3, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0],
]

muitoDificil = [
[4, 0, 0, 0, 0, 0, 0, 0, 7],
[2, 0, 0, 0, 1, 0, 0, 0, 4],
[0, 1, 0, 0, 0, 0, 0, 5, 0],
[5, 0, 1, 0, 4, 0, 7, 0, 6],
[0, 3, 0, 1, 0, 8, 0, 4, 0],
[0, 0, 0, 6, 0, 3, 0, 0, 0],
[0, 0, 2, 0, 0, 0, 4, 0, 0],
[7, 0, 0, 0, 8, 0, 0, 0, 9],
[0, 5, 0, 0, 2, 0, 0, 7, 0],
]
```

```
#VARIAVEIS DE DEBUG
if (debug):
       sleepTime = 0.0
else:
       sleepTime = 1.0

meuSudoku = []

# Opcao do usuario
opcao = 0

limpaTela()

# Menu inicial
print("**********SUDOKU IA**********")
print("\n\nSelecione o tipo de jogo:")
print("1 - Rodar exemplo Muito Fácil")
print("2 - Rodar exemplo Fácil")
print("3 - Rodar exemplo Médio")
print("4 - Rodar exemplo Difícil")
print("5 - Rodar exemplo Muito Difícil")
print("6 - Inserir um Sudoku")

#Le opção
opcao = int(input())

#Trata cada opção
if opcao == 1:
     print(resolverSudoku(muitoFacil, sleepTime), "\n", "Número de iterações: ",
numIter)
elif opcao == 2:
       print(resolverSudoku(facil,  sleepTime),  "\n",  "Número  de  iterações:  ",
numIter)
elif opcao == 3:
       print(resolverSudoku(medio,  sleepTime),  "\n",  "Número  de  iterações:  ",
numIter)
elif opcao == 4:
       print(resolverSudoku(dificil,  sleepTime),  "\n",  "Número  de  iterações:  ",
numIter)
elif opcao == 5:
     print(resolverSudoku(muitoDificil, sleepTime), "\n", "Número de iterações: ",
numIter)
elif opcao == 6:
    limpaTela()
    print("Digite seu Sudoku, os elementos devem ser números de 0 a 9, sendo 0 uma
casa vazia\n")

    # Lê os inputs do usuários
```

```
    for i in range(0, 9):
        elementos = list(map(int, input().split()))

        #Condições do sudoku
        while(len(elementos) != 9):
            print("Cada fileira deve ter 9 elementos")
            elementos = list(map(int, input().split()))
        while(i > 9 or i < 0 for i in elementos):
            print("Os elementos devem ser de 0 a 9")
            elementos = list(map(int, input().split()))
        meuSudoku.append(elementos)

      print(resolverSudoku(meuSudoku, sleepTime), "\n", "Número de iterações: ",
numIter)
```