Relatório: Trabalho 1 de Projeto e Análise de Algoritmos

Alunos: Gustavo Teixeira Magalhães, Davi Arantes.

- Enunciado: Neste projeto de programação, você será desafiado a criar um programa em uma linguagem de programação de sua escolha que seja capaz de resolver automaticamente quebra-cabeças de Sudoku usando o paradigma de Força Bruta. O Sudoku é um jogo de lógica onde o objetivo é preencher uma matriz 9x9 com números de 1 a 9, de modo que cada linha, coluna e submatriz 3x3 contenha todos os números de 1 a 9 sem repetições.

- Método do Paradigma de Força Bruta Utilizado: Backtracking

1. Parte 1: Estruturação do Trabalho:

O programa foi desenvolvido sobre uma estrutura de orientação a objetos, na qual existem duas classes principais:

- A Classe “Board”, que comporta toda a parte estrutural do tabuleiro de sudoku, incluíndo os métodos de verificação de linha, coluna e quadrante, além disso, a classe Board também contém o método que lê o tabuleiro de um arquivo texto e o método que verifica se o tabuleiro está completo.
- A Classe “Solver”, que implementa a lógica de backtracking em si. Solver tem apenas dois métodos: *whichIsSafe*, que retorna uma lista de possibilidades de uma posição no tabuleiro e *solve*, que resolve o tabuleiro.

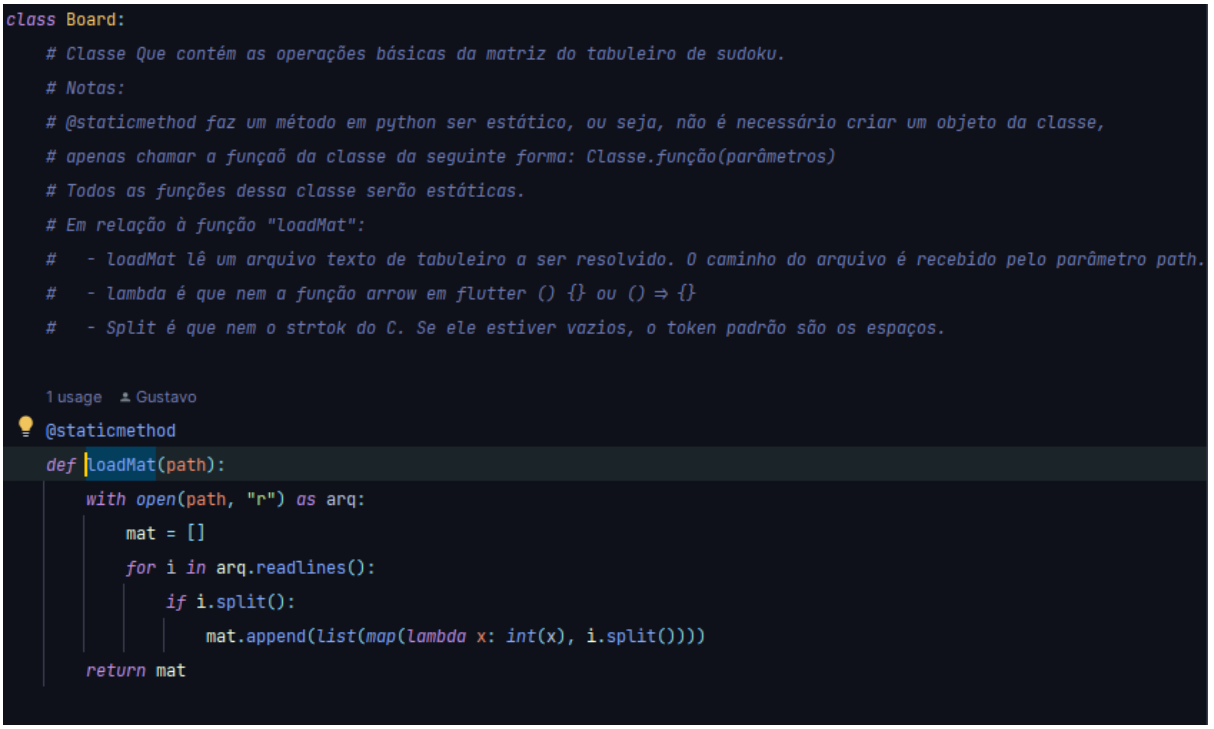
Todos os métodos dessas classes são estáticos, o que é evidenciado pela *keyword @staticmethod* do python.

Além das classes principais, o projeto tem um arquivo main.py que chama a função *solve* e uma pasta “testes” contendo vários tabuleiros de sudoku que podem ser testados.

2. A Classe Board:

A Classe Board, como explicado acima, contém todos os métodos que envolvem a verificação e importação da matriz. São eles:

1. *loadmat(path):*



```
class Board:
    # Classe Que contém as operações básicas da matriz do tabuleiro de sudoku.
    # Notas:
    # @staticmethod faz um método em python ser estático, ou seja, não é necessário criar um objeto da classe,
    # apenas chamar a funçaõ da classe da seguinte forma: Classe.função(parâmetros)
    # Todos as funções dessa classe serão estáticas.
    # Em relação à função "loadMat":
    #   - loadMat lê um arquivo texto de tabuleiro a ser resolvido. O caminho do arquivo é recebido pelo parâmetro path.
    #   - lambda é que nem a função arrow em flutter () {} ou () ⇒ {}
    #   - Split é que nem o strtok do C. Se ele estiver vazios, o token padrão são os espaços.

    1 usage   Gustavo
    @staticmethod
    def loadMat(path):
        with open(path, "r") as arq:
            mat = []
            for i in arq.readlines():
                if i.split():
                    mat.append(list(map(lambda x: int(x), i.split())))
        return mat
```

Parâmetros: *Path* (string)

loadmat recebe como parâmetro uma string, que é path para um arquivo, inicia uma lista e lê as linhas do arquivo com *arq.readlines()* dentro de um for, ou seja, o método lê linha a linha do arquivo, passa pela função *split()*, que retira os espaços em branco da linha e ignora quebras de linha, se houverem, e por fim converte o valor lido em string e concatena na linha da matriz, assim sucessivamente até ler o tabuleiro inteiro. Lembrando que não existe matriz em python, então a matriz retornada pela função é na verdade uma lista de listas.

2. *verificaLinha(verifica, board, linha):*

   Parâmetros: *verifica* (int), *board* (matriz), *linha* (int)

```python
@staticmethod
def verificaLinha(verifica, board, linha):
    if verifica in board[linha]:
        return False
    return True
```

   verificaLinha recebe três parâmetros: um número a ser verificado, o tabuleiro e a linha a ser verificada. O método verifica se o número existe em algum lugar da linha, e se tiver, retorna False, caso contrário, retorna True.

3. *verificaColuna(verifica, board, coluna):*

```python
@staticmethod
def verificaColuna(verifica, board, coluna):
    for linha in range(0, len(board)):
        if verifica == board[linha][coluna]:
            return False
    return True
```

   verificaColuna, de forma análoga à função anterior retorna *False* se tiver uma ocorrência do número na coluna, caso contrário, retorna *True*.

4. *verificaQuadrante(verifica, board, linha, coluna):*

```
@staticmethod
def verificaQuadrante(verifica, board, linha, coluna):
    row_start = (linha // 3) * 3
    col_start = (coluna // 3) * 3
    for i in range(row_start, row_start + 3):
        for j in range(col_start, col_start + 3):
            if verifica == board[i][j]:
                return False
    return True
```

De forma parecida, a função *verificaQuadrante* faz o cálculo (linha // 3) * 3 para separar qual o quadrante do número a ser verificado e retorna *False* se tiver uma ocorrência do número no quadrante e *True,* caso contrário.

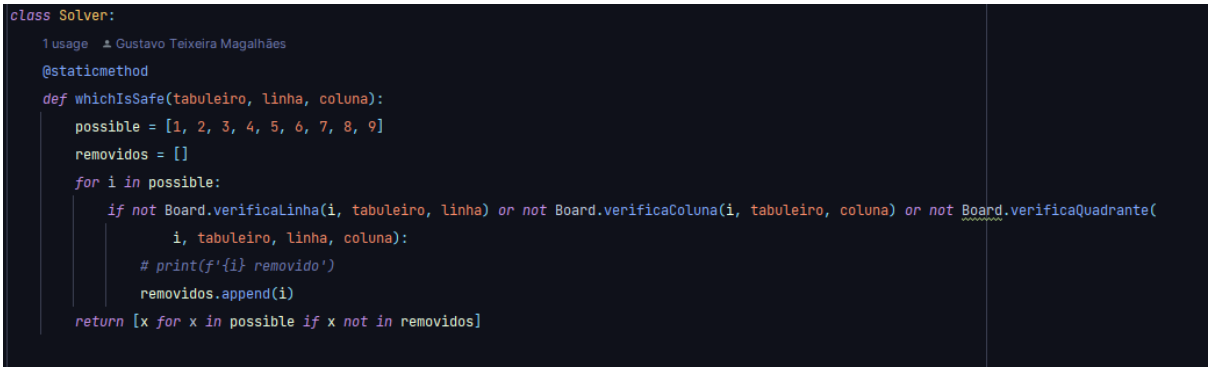
5. *isComplete(board):*

```
@staticmethod
def isComplete(board):
    for linha in range(0, 9):
        if not Board.verificaLinha( verifica: 0, board, linha):
            return False
    return True
```

O método *isComplete* busca na matriz inteira se existe algum 0 (significando que o tabuleiro ainda não está completo), retorna *False*, caso contrário (significando que o tabuleiro foi completo), retorna *True*.

3. A classe Solver:

Solver é a classe mais importante do projeto, ela implementa o backtracking que resolve o Sudoku de fato, além da função que verifica as possibilidades.



```
class Solver:
    1 usage   Gustavo Teixeira Magalhães
    @staticmethod
    def whichIsSafe(tabuleiro, linha, coluna):
        possible = [1, 2, 3, 4, 5, 6, 7, 8, 9]
        removidos = []
        for i in possible:
            if not Board.verificaLinha(i, tabuleiro, linha) or not Board.verificaColuna(i, tabuleiro, coluna) or not Board.verificaQuadrante(
                    i, tabuleiro, linha, coluna):
                # print(f'{i} removido')
                removidos.append(i)
        return [x for x in possible if x not in removidos]
```

1. *whichIsSafe(tabuleiro, linha, coluna):*

whichIsSafe, por mais que pareça simples, é a base na qual está construída toda a lógica de backtracking do trabalho. Ela recebe uma posição do tabuleiro e retorna uma lista dos números possíveis de se colocar naquela posição. Por meio de um *for*, ela utiliza das funções de verificação da classe *Board* para remover os números inválidos para aquela posição, e retorna a lista de possibilidades. Evidentemente, se não houver nenhuma possibilidade, *whichIsSafe* retorna uma lista vazia ([]).

2. *solve(tabuleiro, linha = 0, coluna = 0):*

```python
@staticmethod
def solve(tabuleiro, linha=0, coluna=0):
    if Board.isComplete(tabuleiro):
        return True
    possibilidades = Solver.whichIsSafe(tabuleiro, linha, coluna)
    # Se a posição não estiver vazia
    if tabuleiro[linha][coluna] != 0:
        if coluna == 8:
            if Solver.solve(tabuleiro, linha + 1,  coluna: 0):
                return True
        else:
            if Solver.solve(tabuleiro, linha, coluna + 1):
                return True
    # Se possibilidades não estiver vazia
    elif possibilidades:
        for i in possibilidades:
            tabuleiro[linha][coluna] = i
            if coluna == 8:
                if Solver.solve(tabuleiro, linha + 1,  coluna: 0):
                    return True
            else:
                if Solver.solve(tabuleiro, linha, coluna + 1):
                    return True
            tabuleiro[linha][coluna] = 0
    # Se possibilidades estiver vazia

    return False
```

Solve é a função que implementa o backtracking em si. Ela segue várias verificações e recursividades para fazê-lo. *Solve* utiliza do seguinte algoritmo:

1 . Verifique se o Tabuleiro já está preenchido, se estiver, retorna o tabuleiro (caso base).

2 . Crie a Lista de possibilidades, com a função *whichIsSafe*.

3 . Se a posição no tabuleiro não estiver vazia, chame a função por meio de recurso. (Incrementando por coluna. Se a coluna for igual a 8, incrementa a linha, e zera a coluna, de modo a percorrer a matriz desse jeito).

4 . Caso a lista de possibilidades não esteja vazia, itere a lista, e atribua a posição com o valor da lista, chame a função recursiva novamente, da mesma forma da verificação anterior. Caso a função encontre um ponto sem saída em passos futuros, ela retorna a esta verificação, atribui o valor da posição a 0 e continua a iteração da lista.

5 . Caso nenhum dos valores da lista de possibilidades seja válido, o programa retorna *False* e dá backtracking (retorna na posição anterior).

4. Função *Main:*

```python
from Board import Board
from Solver import Solver
import os

if __name__ == "__main__":
    pastaDeTestes = './Testes'
    caminhos = [os.path.join(pastaDeTestes, nome) for nome in os.listdir(pastaDeTestes)]
    testes = []
    for i in caminhos:
        testes.append(Board.loadMat(i))

    for i in testes:
        print("Tabuleiro Anterior")
        for j in i:
            print(j)
        Solver.solve(i)
        print("\n")
        print("Tabuleiro Resolvido")
        for j in i:
            print(j)
        print("\n\n")
```

A função *main* verifica o caminho da pasta onde estão os *.txt* dos tabuleiros a serem resolvidos e carrega eles em uma lista de tabuleiros (usando a

função *load*, e por meio de um *for*, mostra o tabuleiro antes de ser resolvido, e após ser resolvido.

5. Conclusão e Desafios Enfrentados:

Implementar o paradigma da força bruta foi um exercício de paciência. A maior das dificuldades que encontrei foi abstrair o *backtracking* a nível de código, fazer as funções recursivas e entender quando o programa voltará na posição certa é um desafio considerável. Porém, ao terminar o trabalho, percebo um entendimento melhor acerca do paradigma da força bruta e do uso de recursos no geral. Acima de tudo, foi uma experiência de aprendizado incomparável e que, com toda certeza, ajudará em projetos e problemas futuros.