Exercício prático - Agente de resolução de problemas - Parte 02

Alunos: Antônio Carlos Ramos Filho, Fabiano Amaral Alves, Pablo Junio Souza Santos

1. Linguagem e IDE utilizadas

Nesta segunda parte do exercício continuamos a utilizar a linguagem **Python** juntamente com o Ambiente de Desenvolvimento Integrado disponibilizado pela *JetBrains*, o **Pycharm**.

Este documento foi escrito utilizando um arquivo e sintaxe *Markdown* e sua conversão para arquivo *pdf* foi realizada utilizando uma extensão do *Microsoft Visual Studio Code* chamada **Markdown PDF**.

2. Espaço de estados

Nossa representação de espaço de estados se fundamenta em uma classe python denominada `Espaco_de_estados`, essa classe possui um atributo `espaco_de_estados` ao qual representa um grafo estruturado como um dicionário python, onde cada cidade teria uma lista de cidades vizinhas com a distância em relação a elas.

A seguir detalharemos um pouco do funcionamento da classe e as funcionalidades dos seus métodos.

Atributo - espaco_de_estados: O espaço de estados pode ser interpretado como uma grafo, onde os vértices são os estados e as arestas são as ações. Com isso, o atributo faz jus ao seu nome e representa de fato um espaço de estados, pois em nossa implementação ele receberá um grafo através de seu construtor e a classe poderá dependendo da necessidade retornar a lista de cidades e uma lista de vizinhos.

> Nós pressupomos que o grafo a ser passado pelo arquivo será um dicionário python, então se houver algum erro na extração dos dados no arquivo será criado um objeto do tipo dicionário vazio.

```
class Espaco_de_Estados(object):
    def __init__(self, grafo_estados = None):
        if grafo_estados == None:
            grafo_estados = {}
        self.espaco_de_estados = grafo_estados
```

- Os pequenos trechos de código apresentados aqui serão mostrados de maneira completa formando a classe `Espaco_de_Estado` mais adiante.

Lista de cidades: Caso necessário podemos obter a lista das cidades presentes no grafo através do método `get_cidades`.

> Retornará uma lista dos vértices extraídos do dicionário, note que os vértices ná lista de vizinhos não são incluídos.

```
def get_cidades(self):
    return [self.espaco_de_estados.keys()]
```

Lista de vizinhanças (Arestas): Caso seja necessário utilizar uma lista de vizinhanças presentes no grafo, pode ser obtida através do método `get_visinhanca`.

> Ela utiliza de um método que será explicado mais adiante para retornar a lista dos vizinhos.

```
def get_vizinhancas(self):
    return self.monta_vizinhanca()
```

> Caso precise também do peso (distancia) entre as cidades, poderá-se utilizar do método `get_vizinhaca_com_peso` que retornará uma lista com todos o vizinhos mais a distancia entre eles.

```
def get_vizinhancas_com_peso(self):
    return self.monta_vizinhaca_com_peso()
```

Adicionar uma nova cidade: Caso ocorra algum erro para ler o arquivo e por isso, o atributo `espaco_de_estados` seja definido como um dicionário vazio, podemos adicionar as cidades manualmente ao espaço de estados através do método `add_cidade`.

> também pode ser adicionada a um grafo já preenchido com cidades, nesse caso, não possuiria arestas a principio.

```
def add_cidade(self, vertice):
    if vertice not in self.espaco_de_estados:
        self.espaco_de_estados[vertice] = []
```

Adicionar uma nova vizinhança (Aresta): Assim como podemos adicionar uma nova cidade, também podemos adicionar novos vizinhos, para isso precisamos dos nomes das cidades e da distancia entre elas.

> Pode receber como parâmetro uma tupla, lista ou um conjunto de dados contendo os nomes das cidades e a distancia.

```
def add_vizinhanca(self, aresta):
    aresta = set(aresta)
    (cidade1, cidade2, peso) = tuple(aresta)
    if cidade1 in self.espaco_de_estados:
        self.espaco_de_estados[cidade1].append({cidade2: int(peso)})
    else:
        self.espaco_de_estados[cidade1] = [{cidade2: int(peso)}]
```

"Montar" vizinhanças: Como vimos, podemos obter a lista de todos os vizinhos presentes no grafo, conseguimos obtê-la graças ao método `monta_vizinhanca` que percorre todo o espaço de estados e adiciona as cidades que possuem arestas entre si em uma lista de tuplas.

> O processo de varredura do dicionário é bem interessante, primeiro obtemos a chave, como o valor é uma lista de dicionários, temos que percorrer essa lista pegando as chaves que são as cidades adjacentes.

```
def monta_vizinhanca(self):
    borda = []
    for cidade, vizinho in self.espaco_de_estados.items():
        for k in vizinho:
            for c, p in k.items():
                if {c, cidade} not in borda:
                    borda.append((cidade, c))
    return borda
```

> Para fazer o mesmo processo, mas desta vez adicionando a distancia. Podemos utilizar a `monta_vizinhanca_com_peso` que é uma versão modificada do método anterior, acrescentando a distancia na tupla.

```
def monta_vizinhaca_com_peso(self):
    borda = []
    for cidade, vizinho in self.espaco_de_estados.items():
        for k in vizinho:
            for c, p in k.items():
                if {c, cidade, p} not in borda:
                    borda.append((cidade, c, p))
    return borda
```

Encontrando caminhos: Criamos um método auxiliar para que possamos obter o percurso saindo da cidade inicial até a cidade pretendida.

> Ela retornará uma lista com as cidades visitadas até o destino caso finalize com êxito.

```
def encontrar_caminho(self, inicio, fim, caminho=None):
    if caminho==None:
        caminho = []
    cidades = self.espaco_de_estados
    caminho = caminho+[inicio]
    if inicio == fim:
        return caminho
    if inicio not in cidades:
        return None
    for cidade in cidades[inicio]:
        for c in cidade.keys():
            if c not in caminho:
```

```
                caminho_extendido = self.encontrar_caminho(c, fim, caminho)
                if caminho_extendido:
                    return caminho_extendido
    return None
```

> Para obter o caminho mais a distancia percorrida, podemos utilizar o método `encontrar_caminho_com_custo` que retornará uma tupla com caminho e custo.

```
def encontrar_caminho_com_custo(self, inicio, fim, caminho=None, peso=0):
    if caminho == None:
        caminho = []
    cidades = self.espaco_de_estados
    caminho = caminho + [inicio]
    custo = peso
    if inicio == fim:
        return (caminho, custo)
    if inicio not in cidades:
        return None
    for cidade in cidades[inicio]:
        for c, v in cidade.items():
            if c not in caminho:
                caminho_extendido = self.encontrar_caminho_com_custo(c,
fim, caminho, custo+v)
                if caminho_extendido:
                    return caminho_extendido
    return None
```

Representando o objeto da classe como string: Para recebermos informações de entendimento mais fácil na utilização da classe, sobrescrevemos os dois métodos python para representar objetos da classe como string.

> O método `__repr__(self):` retorna o conteúdo dos atributos de classes como strings, podendo ter texto personalizado.

```
def __repr__(self):
    return f'{self.espaco_de_estados}'
```

> O `__str__(self):` funciona de modo semelhante, mas aqui nós a usamos para apresentar o espaço de estados com a lista das cidades e das arestas.

```
def __str__(self):
    string = 'Cidades: '
    for cidades in self.espaco_de_estados:
        string += str(cidades) + ' '
    string += '\nVizinhanças: '
    for arestas in self.monta_visinhanca():
```

```
            string += str(arestas) + ' '
        return string
```

 # Implementação da classe Espaco_de_Estados

```
# Arquivo: espaco_estado.py

class Espaco_de_Estados(object):

    def __init__(self, grafo_estados = None):
        '''
        Essa classe tratará o espaço de estados como um dicionário, \
        onde receberá um dicionário vindo do método que extrai os \
        dados do arquivo, se nada for passado ao construtor será \
        criado um dicionário vazio.
        Caso contrario a classe receberá o grafo extraído do arquivo.
        '''
        if grafo_estados == None:
            grafo_estados = {}
        self.espaco_de_estados = grafo_estados

    def get_cidades(self):
        '''
        Retorna a lista de cidades
        '''
        return [self.espaco_de_estados.keys()]
        # ou return list(self.espaco_de_estados)

    def get_vizinhancas(self):
        '''
        Retorna uma lista de vizinhanças, ou seja, todas as arestas do
grafo
        '''
        return self.monta_vizinhanca()

    def get_vizinhancas_com_peso(self):
        '''
        Retorna uma lista de vizinhanças com as distancias, ou seja, todas
as arestas do grafo + peso
        '''
        return self.monta_vizinhaca_com_peso()

    def add_cidade(self, vertice):
        '''
        Adiciona uma nova cidade ao espaço de estados
        A principio ela fica sem cidades vizinhas
        '''
        if vertice not in self.espaco_de_estados:
            self.espaco_de_estados[vertice] = []

    def add_vizinhanca(self, aresta):
```

```
        '''
        Adiciona uma nova aresta de cidades no espaço de estados
        :param aresta: pode ser uma lista, tupla ou conjunto contendo as
cidades e a distancia entre elas
        :return: None
        '''
        aresta = set(aresta)

        (cidade1, cidade2, peso) = tuple(aresta)

        if cidade1 in self.espaco_de_estados:
            self.espaco_de_estados[cidade1].append({cidade2: int(peso)})
        else:
            self.espaco_de_estados[cidade1] = [{cidade2: int(peso)}]

    def monta_vizinhanca(self):
        '''
        Esse método percorre o grafo e cria uma lista com as vizinhanças
entre as cidades
        :return: a lista de vizinhos
        '''
        borda = []

        for cidade, vizinho in self.espaco_de_estados.items():
            for k in vizinho:
                for c, p in k.items():
                    if {c, cidade} not in borda:
                        borda.append({cidade, c})
        return borda

    def monta_vizinhaca_com_peso(self):
        '''
        cria a lista de vizinhos, mas acrescentando a distancia entre cada
uma
        :return: lista de vizinhos
        '''
        borda = []

        for cidade, vizinho in self.espaco_de_estados.items():
            for k in vizinho:
                for c, p in k.items():
                    if {c, cidade, p} not in borda:
                        borda.append((cidade, c, p))
        return borda

    def encontrar_caminho(self, inicio, fim, caminho=None):
        '''
        Método auxiliar para obter o percurso de um cidade a outra
        :param inicio: cidade inicial
        :param fim: cidade objetivo
        :param caminho: lista com as cidades que foram visitadas até chegar
no destino
        :return: o caminho que foi percorrido
        '''
```

```
        if caminho==None:
            caminho = []

        cidades = self.espaco_de_estados
        caminho = caminho+[inicio]
        if inicio == fim:
            return caminho
        if inicio not in cidades:
            return None
        for cidade in cidades[inicio]:
            for c in cidade.keys():
                if c not in caminho:
                    caminho_extendido = self.encontrar_caminho(c, fim,
caminho)
                    if caminho_extendido:
                        return caminho_extendido
        return None

    def encontrar_caminho_com_custo(self, inicio, fim, caminho=None,
peso=0):
        '''
        Método auxiliar para obter o percurso de um cidade a outra e a
distancia percorrida
        :param inicio: cidade de partida
        :param fim: cidade objetivo
        :param caminho: percurso percorrido
        :param peso: distancia percorrida
        :return: o caminho + custo ou none caso falha
        '''
        if caminho == None:
            caminho = []
        cidades = self.espaco_de_estados
        caminho = caminho + [inicio]
        custo = peso
        if inicio == fim:
            return (caminho, custo)
        if inicio not in cidades:
            return None
        for cidade in cidades[inicio]:
            for c, v in cidade.items():
                if c not in caminho:
                    caminho_extendido = self.encontrar_caminho_com_custo(c,
fim, caminho, custo+v)
                    if caminho_extendido:
                        return caminho_extendido
        return None

    def __repr__(self):
        '''
        representação do objeto como string
        :return: o objeto em formato string
        '''
        return f'{self.espaco_de_estados}'
```

```
    def __str__(self):
        '''
        Representação personalizada do objeto
        :return: o objeto como string de forma mais legível
        '''
        string = 'Cidades: '
        for cidades in self.espaco_de_estados:
            string += str(cidades) + ' '
        string += '\nVizinhanças: '
        for arestas in self.monta_visinhanca():
            string += str(arestas) + ' '
        return string
```

3. Extração das informações do arquivo e preenchimento do Grafo (Espaço de estados)

Para extrair os dados do arquivo de texto, criamos a `class Arquivo:` com um único método `def extrair_dados(nome_arquivo):` que recebe o arquivo, extrai os dados e devolve a representação de um grafo em uma lista de adjacências.

O método `extrair_dados(nome_arquivo)` devolve a representação do grafo como uma estrutura de dados do python chamada *dicionário*, onde as chaves são vértices e os valores são uma lista de vértices adjacentes com o peso da aresta.

Optamos por usar uma estrutura como *dicionário* para extrair as informações do arquivo e preenchimento do espaço de estados por ser uma maneira mais simples ao nosso ver, e assim nos possibilitar o uso em algoritmos de busca diretamente.

Na estrutura utilizada o 'espaço de estados' ficaria da seguinte forma:

```
{
    'Cidade': [{'vizinho1': distancia1}, {'VizinhoN': distanciaN}]
}
```

Onde `'Cidade'` é tomado como o estado atual, e os `{'VizinhoN': pesoN}` são os estados acessíveis a partir dele sendo `pesoN` o custo para chegar do estado `'Cidade'` a `'VizinhoN'`

Em um teste feito quanto a extração dos dados do arquivo e o retorno do método `extrair_dados`, temos a seguinte saída:

```
defaultdict
(
    <class 'list'>,
        {
            'Salinas': [{'Gmogol': 75}, {'Janauba': 121}],
            'Gmogol': [{'Salinas': 75}, {'Janauba': 96}, {'MOC': 106},
{'Bocaiuva': 123}],
            'Janauba': [{'Salinas': 121}, {'Gmogol': 96}, {'MOC': 120},
{'Januaria': 118}],
```

```
            'MOC': [{'Gmogol': 106}, {'Janauba': 120}, {'Januaria': 148},
{'Pirapora': 129}, {'Bocaiuva': 48}],
            'Bocaiuva': [{'Gmogol': 123}, {'Pirapora': 113}, {'MOC': 48}],
            'Januaria': [{'Janauba': 118}, {'MOC': 148}, {'Pirapora':
213}],
            'Pirapora': [{'Januaria': 213}, {'MOC': 129}, {'Bocaiuva':
113}]
        }
)
```

A seguir veremos a implementação da classe Arquivo.

```
#arquivo: dados.py

import os
from collections import defaultdict

class Arquivo:

    def extrair_dados(nome_arquivo):
        '''
        coloca os dados do arquivo em um dicionário {chave: [valores]}
        '''
        grafo = defaultdict(list)
        '''
        Abre o arquivo com fechamento automático após o fim do bloco with
        '''
        with open(os.path.abspath(nome_arquivo), 'r') as file:
            '''
            lê a primeira linha e descarta
            '''
            file.readline()
            '''
            Também é descartado pois não é necessário com o uso de
dicionário
            '''
            quant_v = file.readline().strip().split('VERTICES: ')
            quant_a = file.readline().strip().split('ARESTAS: ')
            '''
            transforma as linhas restantes em uma lista
            '''
            linhas_rest = file.readlines()
            for linha in linhas_rest:
                '''
                divide a linha em um 'array' com três posições
                '''
                line = linha.strip().split(', ')

                cidade1 = line[0]
                cidade2 = line[1]
                custo= line[2]
```

```
                '''
                cria uma espécie de lista de adjacências no grafo
                '''
                grafo[cidade1].append({cidade2: int(custo)})
                grafo[cidade2].append({cidade1: int(custo)})
        '''
        retorna a estrutura do grafo a ser usada na classe
Espaco_de_Estados
        '''
        return grafo
```