

**UNIVERSIDADE DO ESTADO DO RIO DE JANEIRO**

**FACULDADE DE CIÊNCIAS EXATAS E ENGENHARIAS**

**TRABALHO AV2 ESTRUTURA DE DADOS 2**

**RUAN SOARES DA SILVA FONSECA**

**RIO DE JANEIRO**

**2023**

Q1) Código

import heapq

def dijkstra(mapa, inicio, fim):

distancias = {no: float('inf') for no in mapa} # Dicionário para armazenar as distâncias mínimas de cada nó

distancias[inicio] = 0 # Distância do nó inicial é 0

visitados = set() # Conjunto para armazenar os nós visitados

fila = [(0, inicio)] # Fila de prioridade para selecionar o próximo nó a ser visitado

caminho = {} # Dicionário para armazenar o caminho percorrido

while fila:

distancia\_atual, no\_atual = heapq.heappop(fila) # Seleciona o nó com menor distância atual

if no\_atual == fim:

break # Se o nó atual é o destino, interrompe o loop

if no\_atual in visitados:

continue # Se o nó já foi visitado, passa para o próximo nó

visitados.add(no\_atual) # Marca o nó como visitado

for vizinho in mapa[no\_atual].get("vizinhos", []):

peso = vizinho["peso"] # Peso da aresta entre o nó atual e o vizinho

vizinho\_node = vizinho["node"] # Identificador do nó vizinho

nova\_distancia = distancias[no\_atual] + peso # Calcula a nova distância

if nova\_distancia < distancias[vizinho\_node]:

distancias[vizinho\_node] = nova\_distancia # Atualiza a distância mínima para o vizinho

caminho[vizinho\_node] = no\_atual # Atualiza o nó anterior no caminho

heapq.heappush(fila, (nova\_distancia, vizinho\_node)) # Insere o vizinho na fila de prioridade

if fim not in caminho:

return "Não há rota para o destino especificado." # Se não há caminho para o destino, retorna uma mensagem de erro

# Construir o caminho percorrido

no\_atual = fim

caminho\_percorrido = [no\_atual]

while no\_atual != inicio:

no\_atual = caminho[no\_atual] # Obtém o nó anterior no caminho

caminho\_percorrido.append(no\_atual) # Adiciona o nó atual ao caminho

caminho\_percorrido.reverse() # Inverte o caminho para apresentar na ordem correta

return caminho\_percorrido, distancias[fim] # Retorna o caminho percorrido e a distância total

# Mapa da cidade

nos = {

"1": {

"visitado": False,

"vizinhos": [{"peso": 10, "node": "13", "direcao": "direita"},

{"peso": 8, "node": "2", "direcao": "baixo"}],

},

"2": {

"visitado": False,

"vizinhos": [

{"peso": 7, "node": "14", "direcao": "direita"},

{"peso": 10, "node": "3", "direcao": "baixo"}],

},

"3": {

"visitado": False,

"vizinhos": [

{"peso": 9, "node": "5", "direcao": "direita"},

{"peso": 12, "node": "4", "direcao": "baixo"}],

},

"4": {

"visitado": False,

"vizinhos": [

{"peso": 14, "node": "8", "direcao": "direita"}]

},

"5": {

"visitado": False,

"vizinhos": [

{"peso": 4, "node": "6", "direcao": "direita"},

{"peso": 3, "node": "7", "direcao": "baixo"}

]

},

"6": {

"visitado": False,

},

"7": {

"visitado": False,

"vizinhos": [

{"peso": 5, "node": "6", "direcao": "cima"}

]

},

"8": {

"visitado": False,

"vizinhos": [

{"peso": 9, "node": "7", "direcao": "cima"},

{"peso": 10, "node": "MUSEUM", "direcao": "direita"}

]

},

"MUSEUM": {

"visitado": False,

},

"9": {

"visitado": False,

"vizinhos": [

{"peso": 9, "node": "MUSEUM", "direcao": "esquerda"}

]

},

"10": {

"visitado": False,

"vizinhos": [

{"peso": 9, "node": "9", "direcao": "esquerda"}

]

},

"11": {

"visitado": False,

"vizinhos": [

{"peso": 11, "node": "10", "direcao": "baixo"},

{"peso": 16, "node": "9", "direcao": "baixo"}

]

},

"12": {

"visitado": False,

"vizinhos": [

{"peso": 8, "node": "11", "direcao": "baixo"}

]

},

"13": {

"visitado": False,

"vizinhos": [

{"peso": 13, "node": "12", "direcao": "direita"}

]

},

"14": {

"visitado": False,

"vizinhos": [

{"peso": 6, "node": "13", "direcao": "cima"},

{"peso": 8, "node": "6", "direcao": "baixo"}

]

},

}

inicio = "1"

fim = "MUSEUM"

caminho\_percorrido, distancia\_total = dijkstra(nos, inicio, fim)

if isinstance(caminho\_percorrido, str):

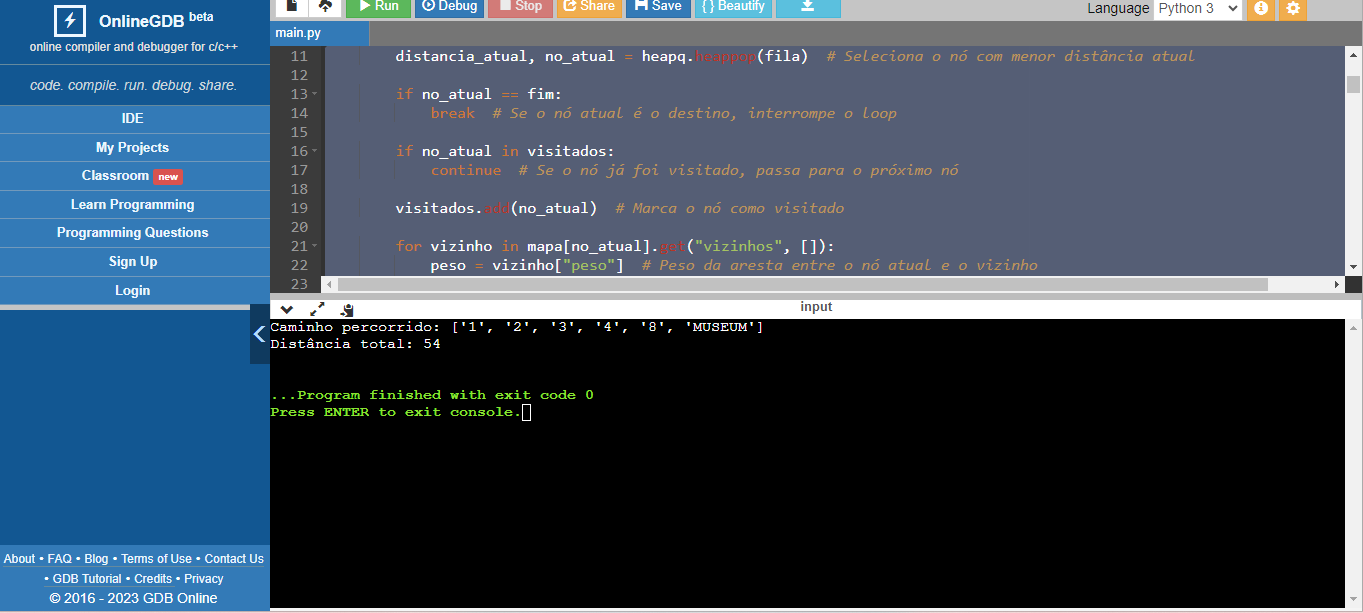
print(caminho\_percorrido)

else:

print("Caminho percorrido:", caminho\_percorrido)

print("Distância total:", distancia\_total)

Seu funcionamento



Q2)Código

def coloracao\_vertices(grafo):

cores = {}

vertices = sorted(grafo.keys())

for vertice in vertices:

cores\_vizinhos = set()

for vizinho in grafo[vertice]:

if vizinho in cores:

cores\_vizinhos.add(cores[vizinho])

cor = "vermelho"

while cor in cores\_vizinhos:

if cor == "vermelho":

cor = "verde"

elif cor == "verde":

cor = "azul"

elif cor == "azul":

cor = "rosa"

else:

cor = "vermelho"

cores[vertice] = cor

return cores

grafo = {

0: [1, 3, 4, 8],

1: [0, 2, 5, 12],

2: [1, 3, 6, 14],

3: [0, 2, 7, 10],

4: [5, 7, 9, 0],

5: [1, 4, 6, 13],

6: [2, 5, 7, 15],

7: [3, 4, 6, 11],

8: [0, 9, 10],

9: [4, 8, 11],

10: [3, 8, 11],

11: [7, 9, 10],

12: [1, 13, 14],

13: [5, 12, 15],

14: [2, 12, 15],

15: [6, 13, 14]

}

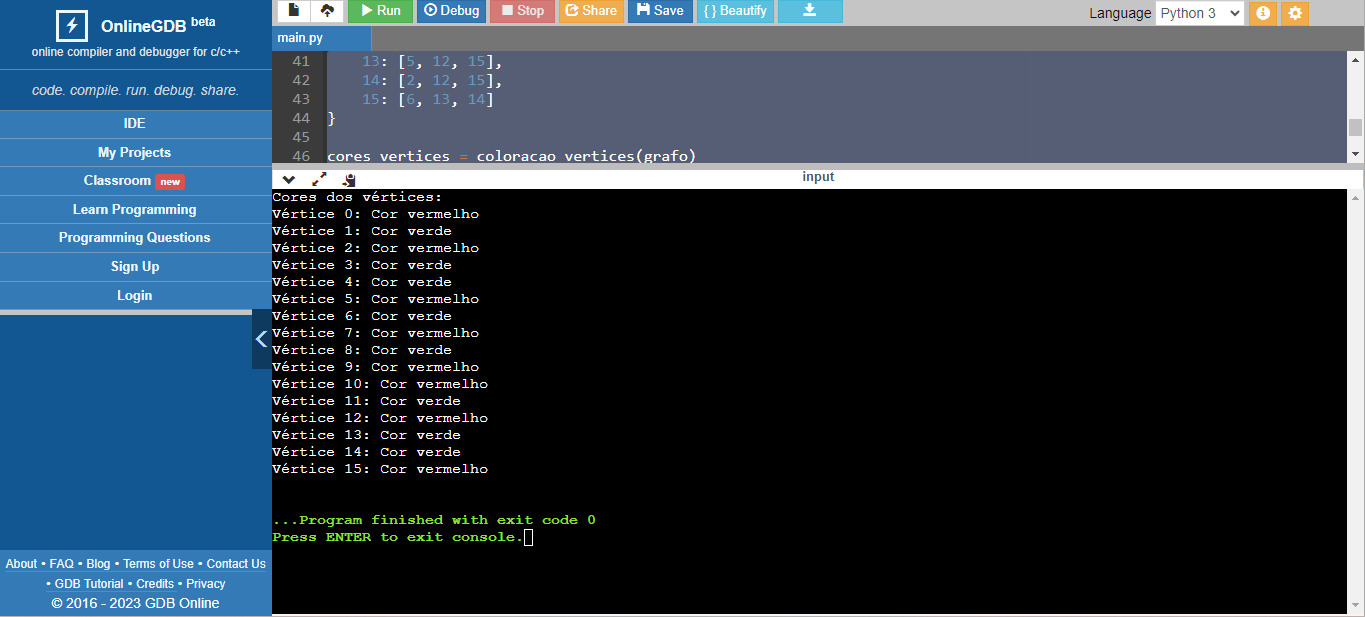
cores\_vertices = coloracao\_vertices(grafo)

print("Cores dos vértices:")

for vertice, cor in cores\_vertices.items():

print(f"Vértice {vertice}: Cor {cor}")

Seu funcionamento



Q3)Código

import csv

# Classe para representar uma aresta no grafo

class Aresta:

def \_\_init\_\_(self, origem, destino, peso):

self.origem = origem

self.destino = destino

self.peso = peso

# Classe para representar a estrutura de conjuntos disjuntos (Union-Find)

class ConjuntosDisjuntos:

def \_\_init\_\_(self, n):

self.pai = [i for i in range(n)]

self.rank = [0] \* n

def encontrar(self, i):

if self.pai[i] != i:

self.pai[i] = self.encontrar(self.pai[i])

return self.pai[i]

def unir(self, x, y):

raiz\_x = self.encontrar(x)

raiz\_y = self.encontrar(y)

if self.rank[raiz\_x] < self.rank[raiz\_y]:

self.pai[raiz\_x] = raiz\_y

elif self.rank[raiz\_x] > self.rank[raiz\_y]:

self.pai[raiz\_y] = raiz\_x

else:

self.pai[raiz\_y] = raiz\_x

self.rank[raiz\_x] += 1

# Função para ler a matriz de adjacências de um arquivo CSV

def ler\_matriz\_adjacencias(nome\_arquivo):

matriz = []

with open(nome\_arquivo, 'r') as arquivo:

leitor\_csv = csv.reader(arquivo)

for linha in leitor\_csv:

matriz.append([int(peso) for peso in linha])

return matriz

# Função para obter as arestas do grafo a partir da matriz de adjacências

def obter\_arestas(matriz):

arestas = []

num\_vertices = len(matriz)

for i in range(num\_vertices):

for j in range(i+1, num\_vertices):

if matriz[i][j] != 0:

arestas.append(Aresta(i, j, matriz[i][j]))

return arestas

def kruskal(matriz\_adjacencias):

arestas = obter\_arestas(matriz\_adjacencias)

arestas = sorted(arestas, key=lambda aresta: aresta.peso)

num\_vertices = len(matriz\_adjacencias)

conjuntos = ConjuntosDisjuntos(num\_vertices)

arvore\_geradora = []

for aresta in arestas:

origem = aresta.origem

destino = aresta.destino

peso = aresta.peso

raiz\_origem = conjuntos.encontrar(origem)

raiz\_destino = conjuntos.encontrar(destino)

if raiz\_origem != raiz\_destino:

arvore\_geradora.append(aresta)

conjuntos.unir(raiz\_origem, raiz\_destino)

return arvore\_geradora

def imprimir\_arvore\_geradora(arvore\_geradora):

custo\_total = 0

for aresta in arvore\_geradora:

origem = aresta.origem

destino = aresta.destino

peso = aresta.peso

custo\_total += peso

print(f"Aresta: {origem} - {destino}, Peso: {peso}")

print(f"Custo total da árvore geradora: {custo\_total}")

nome\_arquivo = 'arquivo.csv'

matriz\_adjacencias = ler\_matriz\_adjacencias(nome\_arquivo)

arvore\_geradora = kruskal(matriz\_adjacencias)

imprimir\_arvore\_geradora(arvore\_geradora)

Seu funcionamento



Q4)Código

labirinto = {

'A': ['01', '04'],

'01': ['06'],

'06': ['07'],

'07': ['02', '08'],

'02': [],

'08': ['15'],

'15': ['14', '21'],

'04': ['05', '09'],

'09': ['10', '29'],

'29': ['30'],

'30': ['51'],

'51': ['52', '58'],

'58': ['57'],

'57': ['36'],

'52': ['47', '59'],

'47': ['31', '48'],

'48': ['54'],

'54': ['53'],

'53': ['60'],

'60': ['61'],

'61': ['55'],

'55': ['56'],

'10': ['16'],

'16': ['17'],

'17': ['12'],

'12': ['11', '13'],

'05': ['19'],

'19': ['18', '20'],

'18': ['24'],

'24': ['25'],

'25': ['34'],

'34': ['33'],

'33': ['38'],

'38': ['37'],

'37': ['41'],

'41': ['42'],

'20': ['27'],

'27': ['26', '28'],

'28': ['46'],

'46': ['45'],

'45': ['35'],

'26': ['40'],

'40': ['39'],

'39': ['43'],

'43': ['44'],

'44': ['49'],

'49': ['50'],

'50': ['B'],

'B': ['63'],

'63': ['62'],

'03': [ ],

'14': [ ],

'21': [ ],

'36': [ ],

'59': [ ],

'22': [ ],

'11': [ ],

'13': [ ],

'42': [ ],

'55': [ ],

'62': [ ],

'35': [ ]

}

visitado = set()

caminho = []

def Profundidade(atual):

visitado.add(atual)

caminho.append(atual)

if atual == 'B':

return True

for proximo in labirinto[atual]:

if proximo not in visitado:

print(f'{atual} -> {proximo}')

if Profundidade(proximo):

return True

if len(labirinto[atual]) == 0:

return False

return False

Profundidade('A')

print("------------")

print("Caminho")

print(caminho)

print("------------")

Seu funcionamento

