Estrutura de Dados e Algoritimos Avançados

Aluno: Ian Costa dos Santos

Turma: 24E2_3

1. **Implemente os seguintes métodos na classe Heap apresentada neste módulo:**
    a. **Método "niveis", que retorna o nº de níveis no heap. Esse método deve ter complexidade O(1);**
    b. **Método "mescla", com um parâmetro para receber um 2º heap. Ele deve mesclar os itens do heap recebido com os que já estavam no heap. Como resultado o 2º heap deve ficar vazio e o heap em que se executa o método deve conter todos os itens que já possuía mais os que estavam no 2º heap. Antes de mesclar os itens, o método deve verificar se as funções "chave" dos 2 heaps são idênticas, gerando uma exceção caso não sejam.**

    Crie 2 filas de prioridade baseadas na classe Heap que você atualizou, cada uma contendo os respectivos registros de pacientes de um hospital, indicados abaixo:

    Os pacientes com maior gravidade (campo "grav") devem ser priorizados. Entre pacientes com mesma gravidade, quem tiver dado entrada antes (de acordo com o valor do campo "dthr") deve ser atendido primeiro.

    Use o método "mescla" para mesclar as duas filas. Em seguida use o método "niveis" para apresentar no console o número de níveis do heap que representa a fila mesclada. Por fim, apresente o heap no console.

R: Além do codigo disponibilizado pelo conteudo do professor na classe Heap, foi desenvolvido dois metodos para realizar a questão.

```python
def niveis(self):
    """Retorna o número de níveis do heap"""
    if self.vazio():
        return 0
    return math.ceil(math.log2(len(self.__dados) + 1))

def mescla(self, outro_heap):
    """Mescla os itens de outro_heap com o heap atual"""
    if self.__chave != outro_heap.__chave:
        raise ValueError("As funções de chave dos heaps são diferentes")

    while not outro_heap.vazio():
        self.insere(outro_heap.remove())
```

Metodo “niveis”: retorna o numero de niveis de um heap, determinado pela altura do mesmo, onde podemos calcular o valor com a formula : Log2(numero de elementos)+1.

Metodo “mescla”: Adiciona os elementos de outro heap do mesmo tipo, utilizando das funções ja existentes (insere e remove), enquanto removemos um elemento de outro heap, adicionamos o mesmo no heap principal.

```python
# Definindo a chave para o heap baseado na gravidade e data/hora de entrada
def chave_paciente(paciente):
    return (paciente["grav"], -datetime.strptime(paciente["dthr"], "%Y-%m-%d %H:%M:%S").timestamp())

# Lista de registros para a 1ª fila
registros1 = [
    {"dthr": "2024-05-30 12:39:01", "nome": "José", "grav": 2},
    {"dthr": "2024-05-30 12:42:03", "nome": "Márcia", "grav": 3},
    {"dthr": "2024-05-30 12:42:06", "nome": "André", "grav": 2},
    {"dthr": "2024-05-30 12:43:10", "nome": "Bruna", "grav": 5}
]

# Lista de registros para a 2ª fila
registros2 = [
    {"dthr": "2024-05-30 12:39:15", "nome": "Ana", "grav": 5},
    {"dthr": "2024-05-30 12:40:21", "nome": "Carlos", "grav": 4},
    {"dthr": "2024-05-30 12:42:28", "nome": "Diego", "grav": 3},
    {"dthr": "2024-05-30 12:42:36", "nome": "Elaine", "grav": 2},
    {"dthr": "2024-05-30 12:43:45", "nome": "Fábio", "grav": 1}
]
```

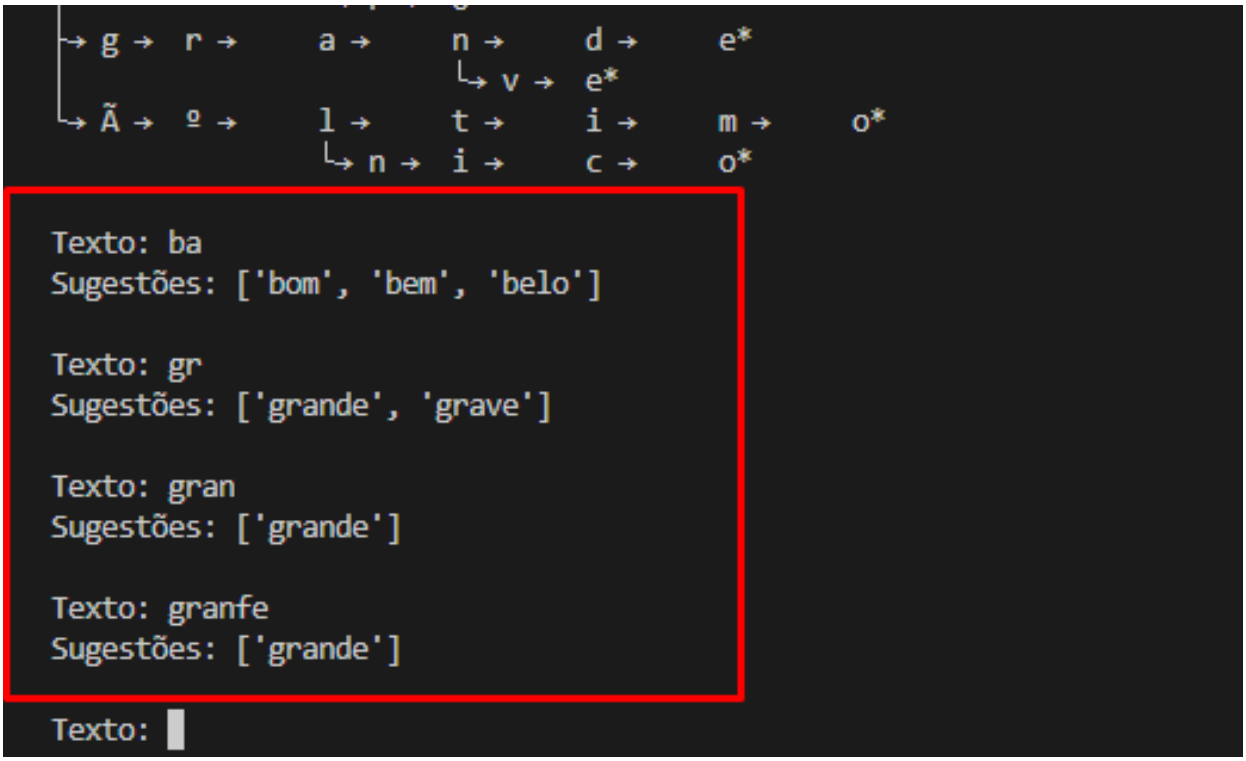
```python
# Criação dos heaps
heap1 = Heap(chave=chave_paciente, max_heap=True)
heap2 = Heap(chave=chave_paciente, max_heap=True)

# Inserindo registros no heap1
for registro in registros1:
    heap1.insere(registro)

# Inserindo registros no heap2
for registro in registros2:
    heap2.insere(registro)

# Mesclando os heaps
heap1.mescla(heap2)

# Verificando o número de níveis do heap mesclado
niveis_heap_mesclado = heap1.niveis()
print(f"Número de níveis do heap mesclado: {niveis_heap_mesclado}")

# Exibindo o heap mesclado
print("Heap mesclado:")
print(heap1)
```

```
$ C:/Users/IanSantos/AppData/Local/Microsoft/WindowsApps/python3.12.exe c:/Users/IanSantos/Docum
Número de níveis do heap mesclado: 4
Heap mesclado:
                ┌→ {'dthr': '2024-05-30 12:42:28', 'nome': 'Diego', 'grav': 3}
        ┌→ {'dthr': '2024-05-30 12:40:21', 'nome': 'Carlos', 'grav': 4}
        │       └→ {'dthr': '2024-05-30 12:42:06', 'nome': 'André', 'grav': 2}
 {'dthr': '2024-05-30 12:39:15', 'nome': 'Ana', 'grav': 5}
        │       ┌→ {'dthr': '2024-05-30 12:42:03', 'nome': 'Márcia', 'grav': 3}
        └→ {'dthr': '2024-05-30 12:43:10', 'nome': 'Bruna', 'grav': 5}
                │       ┌→ {'dthr': '2024-05-30 12:43:45', 'nome': 'Fábio', 'grav': 1}
                └→ {'dthr': '2024-05-30 12:39:01', 'nome': 'José', 'grav': 2}
                        └→ {'dthr': '2024-05-30 12:42:36', 'nome': 'Elaine', 'grav': 2}

[{'dthr': '2024-05-30 12:39:15', 'nome': 'Ana', 'grav': 5}, {'dthr': '2024-05-30 12:43:10', 'nom
nome': 'José', 'grav': 2}, {'dthr': '2024-05-30 12:42:03', 'nome': 'Márcia', 'grav': 3}, {'dthr'
r': '2024-05-30 12:42:36', 'nome': 'Elaine', 'grav': 2}, {'dthr': '2024-05-30 12:43:45', 'nome':
```

2. **Com base neste projeto, implemente uma classe Trie com um método que combine os recursos de autocompletar e autocorreção. O método deve receber como argumento uma string que representa o texto digitado pelo usuário. Assim como no TP2, se a string não estiver na Trie, o método deve executar o recurso de autocorreção, retornando uma lista de palavras sugeridas, contendo palavras armazenadas na Trie que compartilham o prefixo mais longo possível com a string do usuário. Se a string do usuário for encontrada na Trie, o método deve executar o recurso de autocompletar, retornando na lista de sugestões palavras armazenadas na trie com prefixo igual à string do usuário. A lista de sugestões retornada deve conter as palavras mais frequentes dentre as possibilidades aplicáveis, com o máximo de palavras definido por um parâmetro.**

R: No projeto fornecido, foi desenvolvida a função “sugerir_palavras”, que recebe um texto digitado pelo usuário. Se o texto estiver na Trie, ele simplesmente chama o método autocompletar com o texto. Caso contrário, ele reduz progressivamente o texto até encontrar um prefixo na Trie e, em seguida, chama o método autocompletar com esse prefixo. Se nenhum prefixo for encontrado, retorna uma lista vazia.

```
def sugerir_palavras(self, texto, max=3):
    """Combina autocompletar e autocorreção"""
    if self.__busca(texto):
        return self.autocompletar(texto, max)
    else:
        prefixo = texto
        while prefixo:
            if self.__busca(prefixo):
                return self.autocompletar(prefixo, max)
            else:       You, 2 hours ago • feat: q1 e q2
                prefixo = prefixo[:-1]
        return []
```



3. **Altere a classe Grafo apresentada neste módulo para que ela possa representar ambos os tipos de grafos: direcionados e não-direcionados. Adicione um parâmetro "direcionado" ao construtor com valor padrão falso e crie um campo correspondente na classe. Altere todos os métodos que têm comportamento diferente entre os 2 tipos de grafos para que eles se adaptem de acordo com o valor do campo adicionado, inclusive o método "desenha".**
   Construa um grafo de cada tipo com os vértices A, B, C, D, E, F, G, H, I, e J, e as arestas A-G, A-I, C-F, D-A, D-I, H-D, H-E, H-F, H-G, I-H, J-C, e J-H.

   Para cada um:
   a. **Apresente no console os vértices visitados na travessia em profundidade a partir do vértice J;**
   b. **Apresente no console os vértices visitados na travessia em largura a partir do vértice C;**

c. **Apresente no console o caminho entre os vértices J e I encontrado pela busca em profundidade (use a função implementada no TP3);**
d. **Use o método “desenha” para apresentar o grafo.**

R: Utilizando o Classe “Grafo” fornecida pelo conteudo do professor, a principal solução para que os metodos de busca e travessia realizassem sua funcao para ambos tipos de grafos, a principal solução foi:

Modificar método “adiciona_aresta”: Este método é responsável por adicionar arestas ao grafo. Se o grafo for direcionado (direcionado = True), uma aresta é adicionada de v1 para v2, mas não de v2 para v1. Se o grafo for não direcionado (direcionado = False), uma aresta é adicionada em ambas as direções. Isso afeta a estrutura do grafo e, portanto, os resultados dos métodos de travessia e busca.

```
def adiciona_aresta(self, v1, v2):
    v1.vizinhos.append(v2)
    if not self.direcionado:
        v2.vizinhos.append(v1)
```

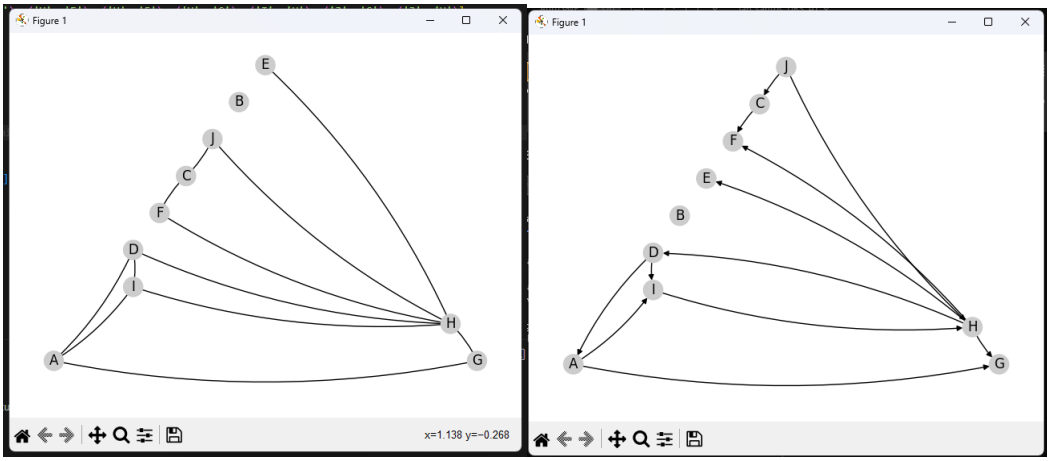
Embora o parâmetro direcionado não seja explicitamente usado nos métodos de travessia e busca, ele influencia a estrutura do grafo que esses métodos percorrem, e assim afeta indiretamente seus resultados. Os metodos de busca e travessia DFS não foram alterados, somente os metodos “desenha” que foi alterado para realizar o desenho do grafico direcionado, e foi tambem adicionado o metodo de “travessia_bfs”

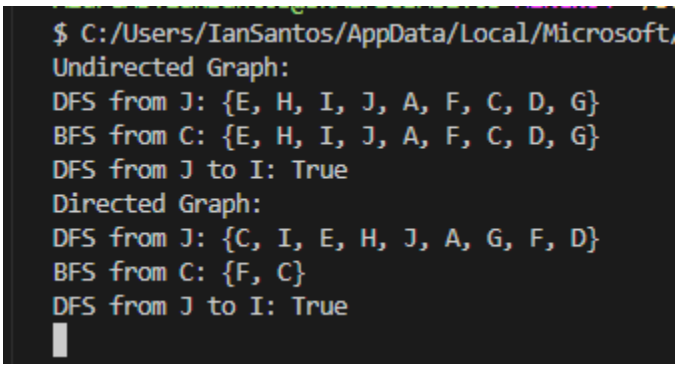
```python
def desenha(self):
    g = nx.DiGraph() if self.direcionado else nx.Graph()
    for vertice in self.vertices:
        g.add_node(vertice)
        for vizinho in vertice.vizinhos:
            g.add_edge(vertice, vizinho)
    pos = nx.planar_layout(g)
    tamanho = [300 * len(str(v)) for v in g.nodes]
    nx.draw(g, pos, with_labels=True, arrows=True,
        connectionstyle='arc3, rad = 0.1',
        node_color="#cccccc", node_size=tamanho)
    plt.show()
```

Testando solução:

```python
vertices = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J']
arestas = [('A', 'G'), ('A', 'I'), ('C', 'F'), ('D', 'A'), ('D', 'I'), ('H', 'D'), ('H', 'E'), ('H', 'F'), ('H', 'G'), ('I', 'H'), ('J', 'C'), ('J', 'H')]

for direcionado in [False, True]:
    grafo = Grafo(direcionado)
    vertice_dict = {v: grafo.cria_vertice(v) for v in vertices}
    for v1, v2 in arestas:
        grafo.adiciona_aresta(vertice_dict[v1], vertice_dict[v2])

    print(f"{'Directed' if direcionado else 'Undirected'} Graph:")
    print("DFS from J:", grafo.travessia_dfs(vertice_dict['J']))
    print("BFS from C:", grafo.travessia_bfs(vertice_dict['C']))
    print("DFS from J to I:", grafo.busca_dfs(vertice_dict['J'], vertice_dict['I']))
    grafo.desenha()
```

Retorno:





4. **Suponha que o governo de um determinado estado quer conectar os reservatórios de água dispersos entre seus municípios. O objetivo é permitir o bombeamento de água em qualquer direção entre aproximadamente 50 reservatórios a fim de evitar falhas no abastecimento em algumas regiões dependendo do nível de precipitação e outras condições. Naturalmente, eles querem minimizar os custos da obra, e é aceitável bombear água via pontos intermediários em vez de conectar cada reservatório diretamente a todos os outros. Que tipo de problema é este? Explique detalhadamente como você representaria as informações e que algoritmo usaria para resolvê-lo.**

R: Este é um exemplo clássico do problema do Mínimo Spanning Tree (MST), que é um subconjunto de arestas de um grafo ponderado conectado. Neste caso, cada reservatório de água pode ser representado como um vértice em um grafo, e cada possível conexão entre reservatórios pode ser representada como uma aresta. O peso de cada aresta seria o custo para conectar dois reservatórios.

Poderiamos solucionar o problema com o algoritimo de Prim ou Kruskal, na imagem abaixo esta um codigo, baseado no codigo “Grafos Ponderados” fornecidos pelo professor no Replit, solucionando com o metodo de Kruskal.

```python
def kruskal_mst(self):
    edges = []
    for vertice in self.vertices:
        for vizinho, peso in vertice.vizinhos.items():
            edges.append((peso, vertice, vizinho))
    edges = sorted(edges, key=lambda x: x[0])  # Ordena arestas por peso

    parent = {}
    rank = {}

    def find(vertice):
        if parent[vertice] != vertice:
            parent[vertice] = find(parent[vertice])
        return parent[vertice]

    def union(v1, v2):
        root1 = find(v1)
        root2 = find(v2)
        if root1 != root2:
            if rank[root1] > rank[root2]:
                parent[root2] = root1
            else:
                parent[root1] = root2
                if rank[root1] == rank[root2]:
                    rank[root2] += 1

    for vertice in self.vertices:
        parent[vertice] = vertice
        rank[vertice] = 0

    mst = []
    for edge in edges:
        peso, v1, v2 = edge
        if find(v1) != find(v2):
            union(v1, v2)
            mst.append(edge)
    return mst
```

```python
def desenha_mst(self, mst):
    g = nx.Graph()
    for edge in mst:
        peso, v1, v2 = edge
        g.add_edge(v1, v2, peso=peso)
    pos = nx.planar_layout(g)
    tamanho = [300 * len(str(v)) for v in g.nodes]
    nx.draw(g, pos, with_labels=True, arrows=True,
            connectionstyle='arc3, rad = 0.1',
            node_color="#cccccc", node_size=tamanho)
    pesos = nx.get_edge_attributes(g, "peso")
    nx.draw_networkx_edge_labels(g, pos, pesos, label_pos=0.8)
    plt.show()
```

Funcionamento do metodo “kruskal_mst”

1) Cria uma lista de todas as arestas do grafo, ordenadas pelo peso.

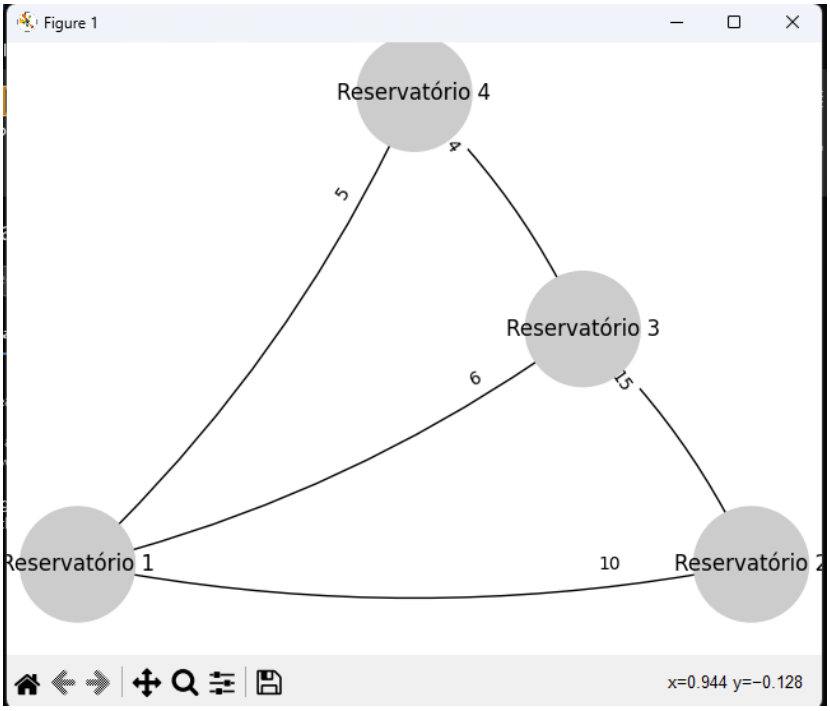
2) Inicializar Estruturas Union-Find:
Cada vértice é seu próprio pai inicialmente (representação de conjuntos disjuntos).
As estruturas parent e rank são usadas para gerenciar as uniões e encontrar as raízes dos conjuntos de maneira eficiente.

3) Encontrar e Unir Conjuntos:
Itera sobre as arestas ordenadas e para cada aresta, verifica se os vértices pertencem a diferentes conjuntos usando a função find.
Se pertencerem a diferentes conjuntos, une os conjuntos usando a função union e adiciona a aresta à MST.

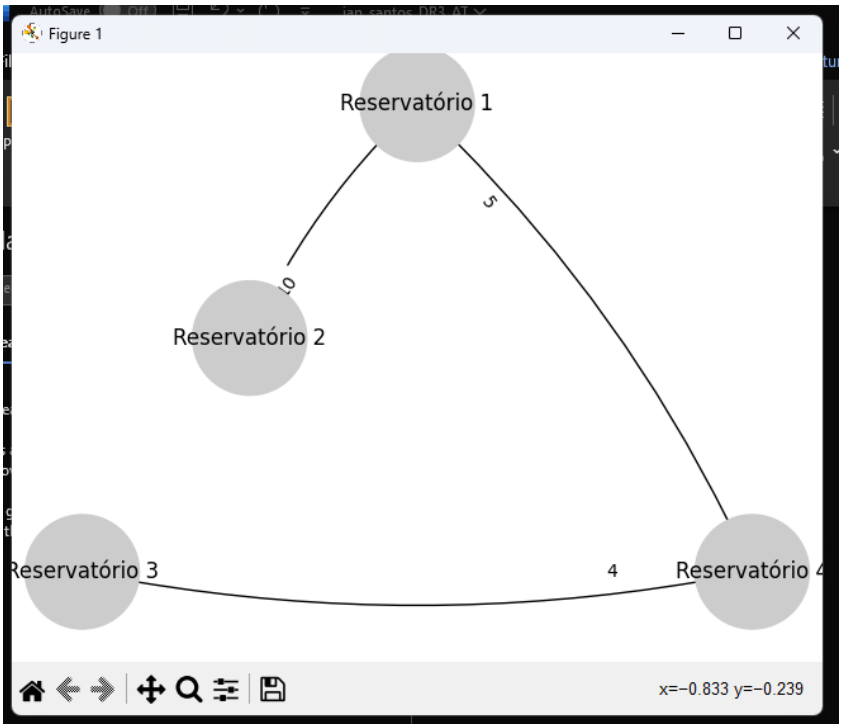
Testando:

```
# Exemplo de uso
grafo = GrafoPonderado()
v1 = grafo.cria_vertice('Reservatório 1')
v2 = grafo.cria_vertice('Reservatório 2')
v3 = grafo.cria_vertice('Reservatório 3')
v4 = grafo.cria_vertice('Reservatório 4')

grafo.adiciona_aresta(v1, v2, 10)
grafo.adiciona_aresta(v1, v3, 6)
grafo.adiciona_aresta(v1, v4, 5)
grafo.adiciona_aresta(v2, v3, 15)
grafo.adiciona_aresta(v3, v4, 4)

grafo.desenha()

mst = grafo.kruskal_mst()
grafo.desenha_mst(mst)
```

Grafo Original:



Grafo MST (Solução):



5. **Muitos problemas requerem encontrar as componentes conexas de um grafo e identificá-las rotulando seus vértices. Para uma rede rodoviária em uma região montanhosa e fria, todas as cidades formam uma única componente conexa quando o tempo está bom. Conforme a neve cai nas montanhas, algumas estradas se tornam intrafegáveis, às vezes cortando a conexão entre cidades. Grafos ponderados podem modelar essa situação representando cidades como vértices e estradas como arestas cujos pesos indicam suas altitudes máximas. Quando a neve interrompe o tráfego acima de uma determinada altitude, as componentes conexas mudam.**

   Implemente uma função chamada "componentes_conexas" que retorne um array com rótulos para cada vértice em um grafo. Vértices com o mesmo rótulo no array são parte da mesma componente conexa. Rótulos diferentes indicam componentes separadas. Além do grafo, a função deve receber uma altitude limite que será usada para identificar quais estradas estão trafegáveis. Seu valor padrão é infinito, situação em que todas as arestas com peso finito são incluídas.

   O algoritmo para encontrar as componentes conexas começa definindo como rótulo para cada vértice o valor do próprio vértice. Então ele faz uma sequência de atualizações para alterar os rótulos. Em cada passada de atualização, o algoritmo verifica todas as arestas com peso abaixo da altitude limite. Se os vértices nas pontas da aresta têm rótulos diferentes, ele substitui o rótulo maior pelo menor (comparando a ordem lexicográfica deles). Isto se repete até que uma passada de atualização percorra todas as arestas sem trocar nenhum rótulo. Os rótulos menores se espalham para todos os vértices em suas componentes conexas até que mais nenhum rótulo se altere.

   Implemente uma 2ª função "vertices_componente" que receba o array de

rótulos e construa uma hash table que mapeie cada rótulo para uma lista de índices de vértices que compartilham o mesmo rótulo. O número de chaves na hash table é o número de componentes conexas. Aplique suas funções ao grafo abaixo usando três altitudes limite diferentes: 50, 21 e 15. Apresente os resultados no console.

R: Utilizando a Classe de GrafoPonderado, foi criado os seguintes metodos:

```python
def componentes_conexas(grafo, altitude_limite=float('inf')):
    rotulos = {v: v for v in grafo.vertices}

    alterou = True
    while alterou:
        alterou = False
        for vertice in grafo.vertices:
            for vizinho, peso in vertice.vizinhos.items():
                if peso <= altitude_limite:
                    if rotulos[vertice] != rotulos[vizinho]:
                        rotulo_maior = max(rotulos[vertice], rotulos[vizinho], key=lambda v: str(v))
                        rotulo_menor = min(rotulos[vertice], rotulos[vizinho], key=lambda v: str(v))
                        for v in rotulos:
                            if rotulos[v] == rotulo_maior:
                                rotulos[v] = rotulo_menor
                        alterou = True
    return [rotulos[v].valor for v in grafo.vertices]

def vertices_componente(rotulos):
    componentes = {}
    for i, rotulo in enumerate(rotulos):
        if rotulo not in componentes:
            componentes[rotulo] = []
        componentes[rotulo].append(i)
    return componentes
    You, 2 minutes ago • Uncommitted changes
```

<u>Método “componentes_conexas”:</u> A função começa atribuindo a cada vértice um rótulo igual ao seu valor. Isso é feito através do dicionário rotulos, que mapeia cada vértice para seu rótulo. Em seguida, a função entra em um loop onde verifica todas as arestas com peso abaixo da altitude limite. Se os vértices nas extremidades da aresta têm rótulos diferentes, o rótulo maior é substituído pelo menor (comparando a ordem lexicográfica deles). Este processo se repete até que uma passagem completa por todas as arestas seja feita sem que nenhum rótulo seja alterado.

Método “vertices_componente”: A função percorre o array de rótulos. Para cada rótulo, se ele ainda não está no dicionário componentes, a função adiciona o rótulo como uma chave e inicializa seu valor como uma lista vazia. Depois, a função adiciona o índice atual à lista de índices associada ao rótulo.

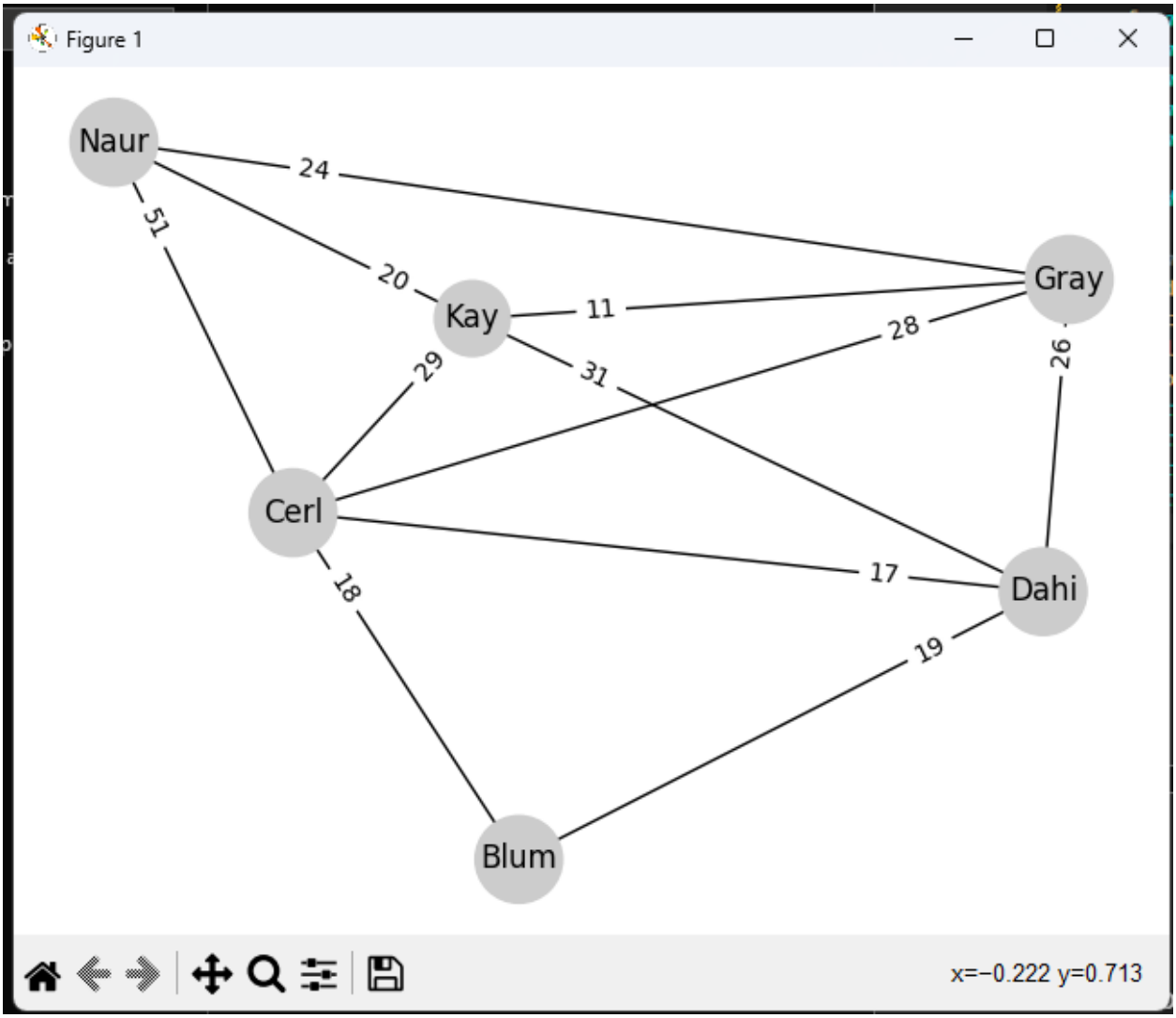
Testando:

```
# # Exemplo de uso
grafo = GrafoPonderado()
v0 = grafo.cria_vertice('Blum')
v1 = grafo.cria_vertice('Cerl')
v2 = grafo.cria_vertice('Gray')
v3 = grafo.cria_vertice('Naur')
v4 = grafo.cria_vertice('Kay')
v5 = grafo.cria_vertice('Dahi')

grafo.adiciona_aresta(v0, v1, 18)
grafo.adiciona_aresta(v0, v5, 19)
grafo.adiciona_aresta(v1, v2, 28)
grafo.adiciona_aresta(v1, v3, 51)
grafo.adiciona_aresta(v1, v4, 29)
grafo.adiciona_aresta(v1, v5, 17)
grafo.adiciona_aresta(v2, v3, 24)
grafo.adiciona_aresta(v2, v4, 11)
grafo.adiciona_aresta(v2, v5, 26)
grafo.adiciona_aresta(v3, v4, 20)
grafo.adiciona_aresta(v4, v5, 31)

grafo.desenha()

# Testando a função com diferentes altitudes
altitudes = [50, 21, 15]
for altitude in altitudes:
    rotulos = componentes_conexas(grafo, altitude)
    componentes = vertices_componente(rotulos)
    print(f"Altitude limite: {altitude}")
    print("Rótulos:", rotulos)
    print("Componentes conexas:", componentes)
    print()
```

Resultado:

```
Altitude limite: 50
Rótulos: ['Blum', 'Blum', 'Blum', 'Blum', 'Blum', 'Blum']
Componentes conexas: {'Blum': [0, 1, 2, 3, 4, 5]}

Altitude limite: 21
Rótulos: ['Blum', 'Blum', 'Gray', 'Gray', 'Gray', 'Blum']
Componentes conexas: {'Blum': [0, 1, 5], 'Gray': [2, 3, 4]}

Altitude limite: 15
Rótulos: ['Blum', 'Cerl', 'Gray', 'Naur', 'Gray', 'Dahi']
Componentes conexas: {'Blum': [0], 'Cerl': [1], 'Gray': [2, 4], 'Naur': [3], 'Dahi': [5]}
```

Grafo:



6. **A figura abaixo representa as linhas férreas existentes entre as seis cidades da região mencionada na questão anterior e seus respectivos tempos de travessia, em minutos.**

   Considerando este cenário:

   a. **Use o algoritmo de Dijkstra para encontrar o caminho mais rápido de Blum a Naur. Apresente o caminho encontrado e o tempo total para percorrê-lo.**
   b. **Implemente uma função para encontrar o caminho mais rápido para percorrer todas as cidades e retornar à cidade de origem. Use essa função para encontrar o caminho mais rápido com início e fim em Blum. Apresente no console o caminho encontrado e o tempo total para percorrê-lo.**
      Dica: você pode usar a função permutations do módulo itertools da biblioteca padrão do Python para gerar todas as permutações de uma coleção de elementos.
   c. **Suponha que, para cortar custos de manutenção, a rede ferroviária da região passará por um processo de enxugamento, mantendo apenas o mínimo de ferrovias necessário para permitir o deslocamento entre todas as cidades. Use o algoritmo de Prim para determinar quais ferrovias devem ser mantidas a fim de minimizar o tempo total necessário para percorrer toda a rede. Apresente-as no console.**
   d. **Suponha que a companhia que administra a rede ferroviária precisa instalar galpões para realizar a manutenção dos trens. Para cada ferrovia, deve haver ao menos um galpão, a fim de que qualquer trem precisando de manutenção seja atendido. Para minimizar os custos, a companhia deseja instalar um número mínimo de galpões que seja suficiente para atender todas as ferrovias. Projete um algoritmo guloso para escolher um**

conjunto de cidades onde serão instalados os galpões, garantindo o atendimento a todas as ferrovias e tentando minimizar o número de galpões instalados.

R: A Partir da classe Grafo Ponderada, foi desenvolvido:

a) A função **"dijkstra"** é utilizada para encontrar o caminho mais curto de Blum a Naur. Ela implementa o algoritmo de Dijkstra para calcular as distâncias mínimas de Blum a todos os outros vértices e retorna o caminho mais curto até Naur, juntamente com o tempo total necessário.

```python
def dijkstra(grafo, inicio, fim):
    distancias = {vertice: float('infinity') for vertice in grafo.vertices}
    distancias[inicio] = 0
    caminho = {vertice: None for vertice in grafo.vertices}
    pq = [(0, inicio)]

    while pq:
        (distancia_atual, vertice_atual) = heapq.heappop(pq)

        if distancia_atual > distancias[vertice_atual]:
            continue

        for vizinho, peso in vertice_atual.vizinhos.items():
            distancia = distancia_atual + peso

            if distancia < distancias[vizinho]:
                distancias[vizinho] = distancia
                caminho[vizinho] = vertice_atual
                heapq.heappush(pq, (distancia, vizinho))

    caminho_reverso = []
    vertice_atual = fim
    while vertice_atual is not None:
        caminho_reverso.append(vertice_atual)
        vertice_atual = caminho[vertice_atual]
    caminho_reverso.reverse()

    return caminho_reverso, distancias[fim]
```

b) A função **“caminho_mais_curto_todas_cidades”** calcula o caminho mais curto para percorrer todas as cidades e retornar a Blum. Ela utiliza permutações e o metodo de dijkstra para encontrar todas as combinações de caminhos possíveis e seleciona o caminho com a menor distância total.

```python
def caminho_mais_curto_todas_cidades(grafo, inicio):
    vertices = grafo.vertices[:]
    vertices.remove(inicio)
    menor_caminho = None
    menor_distancia = float('infinity')

    for perm in permutations(vertices):
        distancia_total = 0
        caminho_atual = [inicio]
        vertice_atual = inicio

        for proximo_vertice in perm:
            _, distancia = dijkstra(grafo, vertice_atual, proximo_vertice)
            distancia_total += distancia
            vertice_atual = proximo_vertice
            caminho_atual.append(proximo_vertice)

        _, distancia = dijkstra(grafo, vertice_atual, inicio)
        distancia_total += distancia
        caminho_atual.append(inicio)

        if distancia_total < menor_distancia:
            menor_distancia = distancia_total
            menor_caminho = caminho_atual

    return menor_caminho, menor_distancia
```

c) A função “**prim”** implementa o algoritmo de Prim para encontrar a árvore geradora mínima (MST) do grafo. Ela começa a partir de um vértice inicial e adiciona repetidamente a aresta de menor peso que conecta um vértice na MST a um vértice fora da MST, até que todos os vértices estejam conectados.

A função **“verifica_caminhos_minimos”** verifica se todos os caminhos mínimos entre pares de vértices estão presentes na MST. Se não estiverem, ela calcula os caminhos

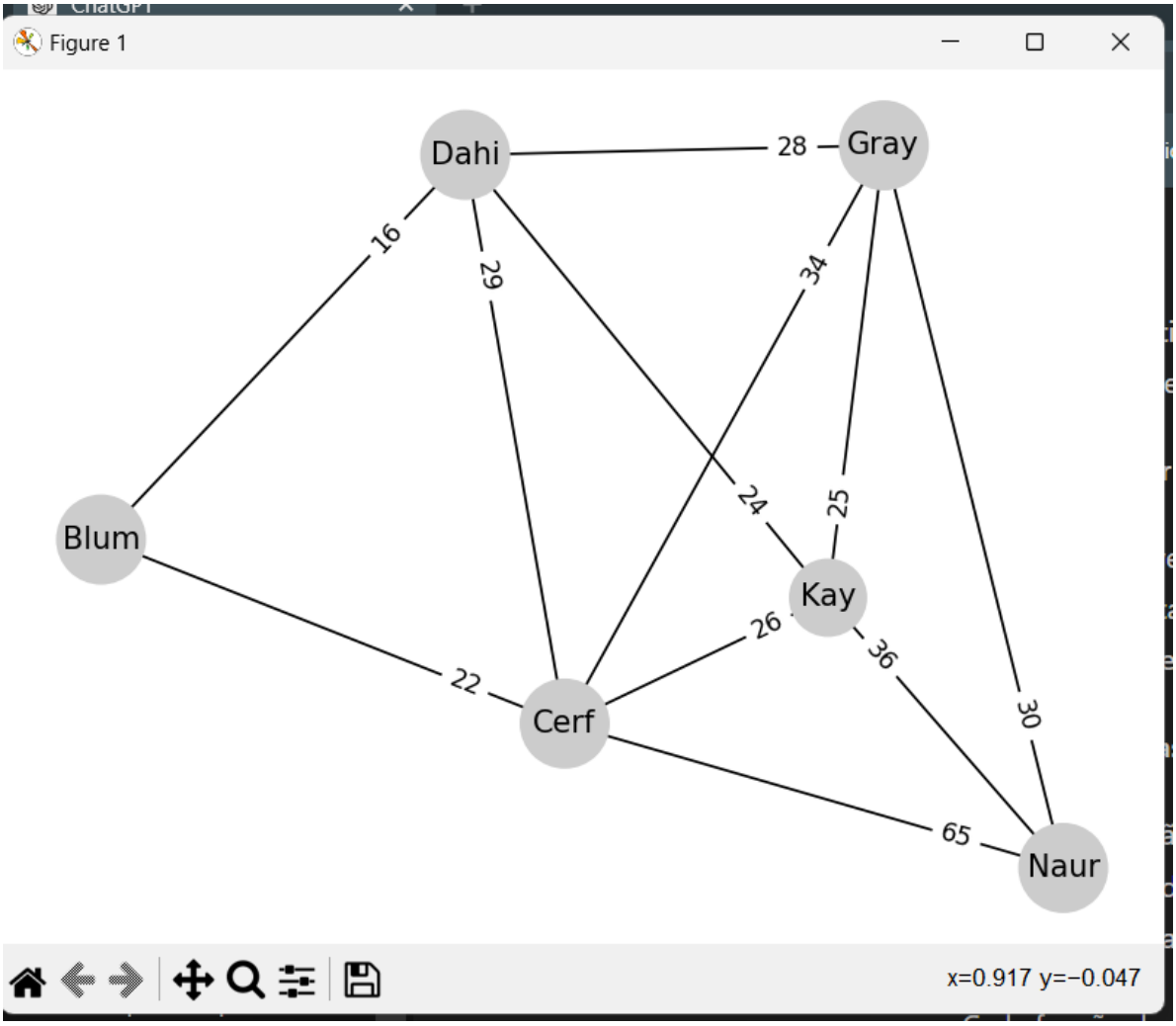
mínimos ausentes usando o algoritmo de Dijkstra e adiciona as arestas correspondentes à MST.
Após verificar os caminhos mínimos necessários, as arestas que compõem esses caminhos são listadas para mostrar as ferrovias necessárias para minimizar o tempo total. Isso é feito removendo duplicatas e ordenando as arestas de acordo com os vértices envolvidos.

```python
def prim(grafo):
  mst = GrafoPonderado()
  visitados = set()
  pq = []

  vertice_inicial = grafo.vertices[0]
  visitados.add(vertice_inicial)
  mst.vertices.append(vertice_inicial)
  for vizinho, peso in vertice_inicial.vizinhos.items():
    heapq.heappush(pq, (peso, vertice_inicial, vizinho))

  while pq:
    peso, v1, v2 = heapq.heappop(pq)
    if v2 not in visitados:
      visitados.add(v2)
      if v2 not in mst.vertices:
        mst.vertices.append(v2)
      mst.adiciona_aresta(v1, v2, peso)
      for vizinho, peso in v2.vizinhos.items():
        if vizinho not in visitados:
          heapq.heappush(pq, (peso, v2, vizinho))

  return mst

def verifica_caminhos_minimos(mst, grafo):
  arestas_necessarias = set()

  for v1 in grafo.vertices:
    for v2 in grafo.vertices:
      if v1 != v2:
        caminho, _ = dijkstra(mst, v1, v2)
        if not caminho:  # Se não há caminho entre v1 e v2 na MST
          _, peso = dijkstra(grafo, v1, v2)
          mst.adiciona_aresta(v1, v2, peso)
          arestas_necessarias.add((v1, v2, peso))
        else:
          for i in range(len(caminho) - 1):
            arestas_necessarias.add((caminho[i], caminho[i+1], mst.vertices[mst.vertices.index(caminho[i])].vizinhos[caminho[i+1]]))

  return arestas_necessarias
```

d) A função instala_galpoes é usada para encontrar as cidades onde os galpões devem ser instalados. Ela seleciona os vértices que têm a maior cobertura de outras cidades não cobertas, garantindo que todas as cidades estejam cobertas pelos galpões.

```python
def instala_galpoes(grafo):      You, 29 minutes ago • q6
    cobertos = set()
    galpoes = set()

    while len(cobertos) < len(grafo.vertices):
        melhor_vertice = None
        maior_cobertura = 0

        for vertice in grafo.verti (variable) vizinho: Any
            if vertice not in galpoe
                cobertura = sum(1 for vizinho in vertice.vizinhos if vizinho not in cobertos) + (1 if vertice not in cobertos else 0)
                if cobertura > maior_cobertura:
                    maior_cobertura = cobertura
                    melhor_vertice = vertice

        galpoes.add(melhor_vertice)
        cobertos.add(melhor_vertice)
        for vizinho in melhor_vertice.vizinhos:
            cobertos.add(vizinho)

    return galpoes
```

Resultado:



Console:

```
$ C:/Users/IanSantos/AppData/Local/Microsoft/WindowsApps/python3.12.exe c:/Users/IanSantos/Documents/faculdade-bloc
/at/q6.py
Caminho mais rápido de Blum a Naur: [Blum, Dahi, Gray, Naur]
Tempo total: 74 minutos
Caminho mais rápido para percorrer todas as cidades e retornar a Blum: [Blum, Cerf, Kay, Naur, Gray, Dahi, Blum]
Tempo total: 158 minutos
Ferrovias a serem mantidas para minimizar o tempo total:
Blum - Cerf: 22 minutos
Blum - Dahi: 16 minutos
Cerf - Dahi: 29 minutos
Cerf - Gray: 34 minutos
Dahi - Gray: 28 minutos
Gray - Naur: 30 minutos
Kay - Cerf: 26 minutos
Kay - Dahi: 24 minutos
Kay - Gray: 25 minutos
Naur - Kay: 36 minutos
Cidades onde serão instalados os galpões: ['Cerf']
```

7. **Indique a quais classes de complexidade cada um dos problemas listados abaixo pertence, justificando suas respostas:**
    a. **Problema da questão 6a.**
    b. **Problema da questão 6b.**
    c. **Problema da questão 6c.**
    d. **Um entregador precisa fazer entregas em 20 casas. Ele precisa encontrar a rota mais curta para fazer essas entregas.**
    e. **Um encanador precisa visitar os clientes A, B e C, respectivamente às 13h, 15h e 17h. Ele analisa o mapa com as vias da cidade a fim de encontrar a melhor rota para visitar os clientes na ordem agendada e voltar para casa.**

R:

a) Problema da questão 6a (Caminho mais rápido de Blum a Naur): Este problema consiste em encontrar o caminho mais curto entre dois vértices em um grafo ponderado. Como estamos usando o algoritmo de Dijkstra para resolvê-lo, que é eficiente e tem complexidade de tempo O(|V| + |E| * log|V|) em um grafo com |V| vértices e |E| arestas, este problema pertence à classe P.

b) Problema da questão 6b (Caminho mais rápido para percorrer todas as cidades e retornar a Blum): Este problema consiste em encontrar o caminho mais curto que passe por todos os vértices de um grafo e retorne ao vértice inicial. Como é utilizado um algoritmo que testa todas as permutações dos vértices, a complexidade é O(n!), onde n é o número de vértices. Portanto, este problema pertence à classe NP-completo.

c) Problema da questão 6c (Ferrovias a serem mantidas para minimizar o tempo total): Este problema envolve encontrar uma árvore geradora mínima em um grafo ponderado, o que é resolvido usando o algoritmo de Prim. A complexidade do algoritmo de Prim é O(|E| * log|V|), onde |V| é o número de vértices e |E| é o número de arestas. Portanto, este problema pertence à classe P.]

d) Um entregador precisa fazer entregas em 20 casas: Este problema é semelhante ao Problema do Caixeiro Viajante, que é NP-completo. Encontrar o caminho mais curto que passe por todas as casas e retorne à casa inicial envolve testar todas as permutações das casas, o que tem complexidade fatorial e pertence à classe NP-completo.

e) Um encanador precisa visitar os clientes A, B e C: Este problema também é semelhante ao Problema do Caixeiro Viajante, com a restrição adicional de que as visitas devem ocorrer em uma ordem específica. Portanto, também pertence à classe NP-completo.

8. **Sauron, também conhecido como o Senhor dos Anéis, nomeou você como responsável pelas forjas de Mordor. Seu papel consiste em supervisionar as forjas e garantir o fornecimento constante de minério de ferro para que elas produzam armas em sua capacidade total.**

   O mapa de Mordor abaixo indica as localizações das forjas, numeradas de 1 a 11. A única mina de ferro está indicada pela letra M. As linhas representam os caminhos existentes entre esses locais. O número

associado a cada caminho é o tempo (em horas) que leva para percorrê-lo.

Dada a importância estratégica de sua missão, sua vida depende de sua produtividade e eficiência! Você precisa percorrer toda Mordor para inspecionar as forjas. Então sua 1ª tarefa é identificar o caminho mais rápido entre todos os locais de interesse. Use o algoritmo de Floyd-Warshall para isso e apresente no console uma matriz com o tempo total para percorrer cada um dos caminhos.

Agora que você conhece o tempo de deslocamento entre todos os pontos, você precisa definir uma rota rápida para fazer a inspeção das forjas de forma eficiente. Implemente uma função que use uma heurística para encontrar uma rota partindo de um ponto, passando por todos os outros e retornando ao ponto inicial. Sua função deve aplicar a heurística a cada um dos pontos iniciais possíveis e selecionar a melhor rota produzida. Se for necessário, a função deve ajustar a rota (para começar e terminar no ponto desejado) antes de retorná-la. Apresente no console a melhor rota encontrada e o tempo total para percorrê-la.

Analise a função que você implementou e use a notação Big O para descrever a complexidade dela, justificando sua resposta.

R:

Metodo “floyd_warshal”: Cria uma matriz de distâncias, onde dist[i][j] representa a distância do vértice i ao vértice j. Inicialmente, a distância é definida como infinita (inf) exceto para a distância de um vértice para si mesmo, que é zero.
Atualização de arestas: Define as distâncias diretas entre vértices conectados pelas arestas do grafo. Usa três laços aninhados para iterar sobre todos os pares de vértices

(i, j) e um vértice intermediário k. Se a distância de i para j passando por k for menor do que a distância direta de i para j, a matriz de distâncias é atualizada.

Complexidade: O(V3): Três laços aninhados sobre todos os vértices para atualizar a matriz de distâncias.

```
def floyd_warshall(self):
    n = len(self.vertices)
    dist = np.full((n, n), np.inf)
    np.fill_diagonal(dist, 0)
    vertice_idx = {vertice: idx for idx, vertice in enumerate(self.vertices)}

    for v1 in self.arestas:
        for v2 in self.arestas[v1]:
            i, j = vertice_idx[v1], vertice_idx[v2]
            dist[i][j] = self.arestas[v1][v2]

    for k in range(n):
        for i in range(n):
            for j in range(n):
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

    return dist, vertice_idx
```

Metodo "encontra_melhor_rota": Usando a Heurística do Vizinho Mais Próximo para vértice inicial possível, Seleciona iterativamente o vizinho mais próximo que ainda não foi visitado. Adiciona esse vizinho à rota e atualiza a lista de vértices visitados. Repete o processo até que todos os vértices sejam visitados. Calcula a distância total para cada rota possível, ajusta a rota para começar e terminar no ponto inicial, e seleciona a rota com a menor distância total.

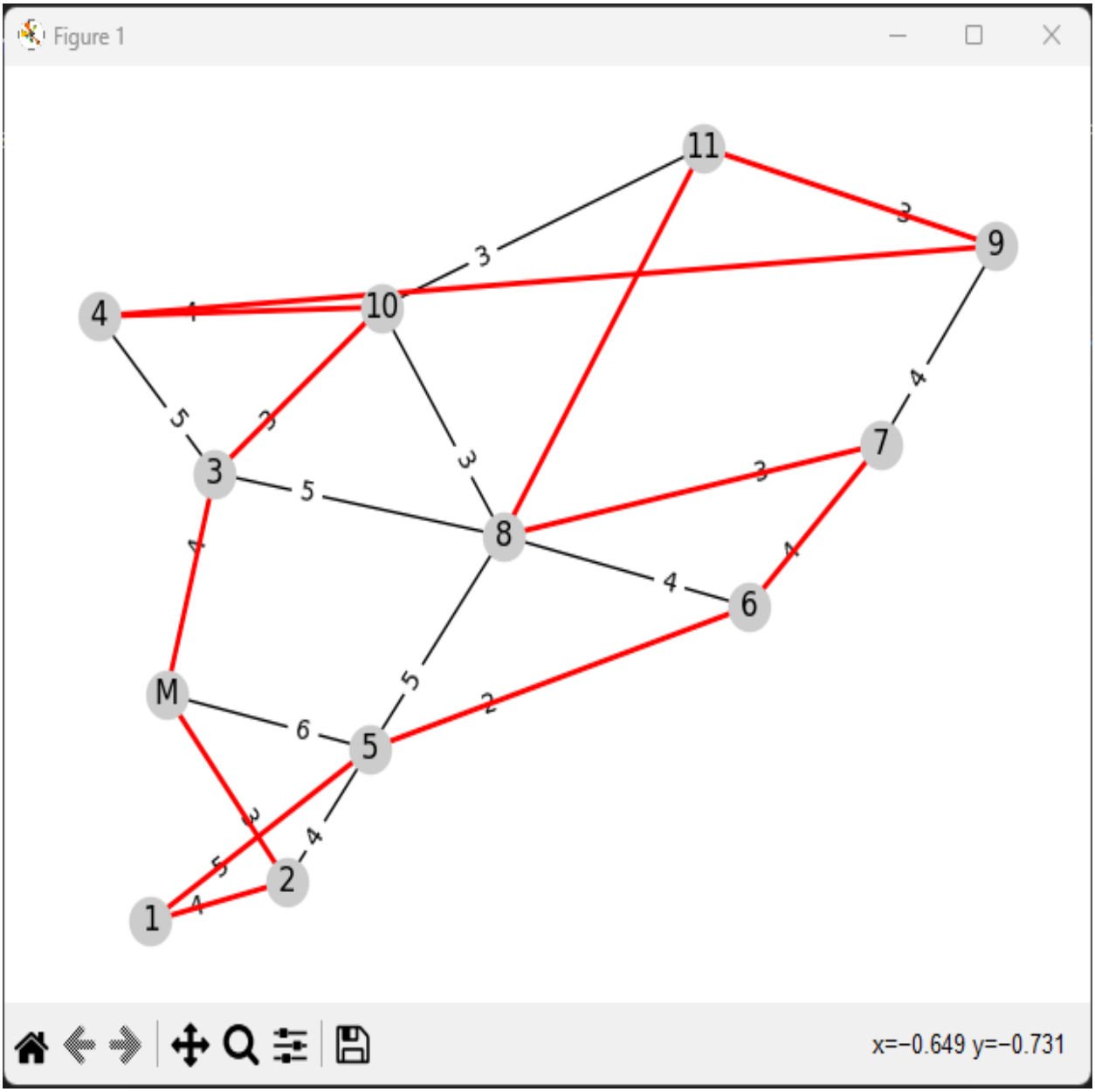
Complexidade: $O(V3)$: Para cada vértice inicial, encontra o vizinho mais próximo entre os vértices restantes, repetido para todos os vértices.

```python
def nearest_neighbor(self, start, dist_matrix, vertice_idx):
    n = len(self.vertices)
    visited = [False] * n
    route = [start]
    total_distance = 0
    current_idx = vertice_idx[start]
    visited[current_idx] = True

    for _ in range(n - 1):
        next_idx = np.argmin([dist_matrix[current_idx][j] if not visited[j] else np.inf for j in range(n)])
        total_distance += dist_matrix[current_idx][next_idx]
        current_idx = next_idx
        route.append(self.vertices[current_idx])
        visited[current_idx] = True

    total_distance += dist_matrix[current_idx][vertice_idx[start]]
    route.append(start)
    return route, total_distance

def melhor_rota(self):
    dist_matrix, vertice_idx = self.floyd_warshall()
    melhor_rota = None
    menor_distancia = np.inf

    for vertice in self.vertices:
        rota, distancia = self.nearest_neighbor(vertice, dist_matrix, vertice_idx)
        if distancia < menor_distancia:
            melhor_rota = rota
            menor_distancia = distancia

    return melhor_rota, menor_distancia
```

Resultado:

```python
dist_matrix, vertice_idx = grafo.floyd_warshall()
print("Matriz de Floyd-Warshall (tempo total para percorrer cada caminho):")
print(dist_matrix)

rota, distancia = grafo.melhor_rota()
print(f"Melhor rota encontrada: {rota}, distância total: {distancia}")
    You, 32 minutes ago • start q8
grafo.desenha(rota)
```

```
/at/q8.py
Matriz de Floyd-Warshall (tempo total para percorrer cada caminho):
[[ 0.  4. 11. 16.  5.  7. 11. 10. 15. 13. 16.  7.]
 [ 4.  0.  7. 12.  4.  6. 10.  9. 14. 10. 13.  3.]
 [11.  7.  0.  5. 10.  9.  8.  5.  9.  3.  6.  4.]
 [16. 12.  5.  0. 12. 11. 10.  7. 10.  4.  7.  9.]
 [ 5.  4. 10. 12.  0.  2.  6.  5. 10.  8. 11.  6.]
 [ 7.  6.  9. 11.  2.  0.  4.  4.  8.  7. 10.  8.]
 [11. 10.  8. 10.  6.  4.  0.  3.  4.  6.  7. 12.]
 [10.  9.  5.  7.  5.  4.  3.  0.  7.  3.  6.  9.]
 [15. 14.  9. 10. 10.  8.  4.  7.  0.  6.  3. 13.]
 [13. 10.  3.  4.  8.  7.  6.  3.  6.  0.  3.  7.]
 [16. 13.  6.  7. 11. 10.  7.  6.  3.  3.  0. 10.]
 [ 7.  3.  4.  9.  6.  8. 12.  9. 13.  7. 10.  0.]]
Melhor rota encontrada: ['4', '10', '3', 'M', '2', '1', '5', '6', '7', '8', '11', '9', '4'], distância total: 51.0
```



OBS: A Matrix pode mudar de acordo com a ordem dos vertices listados.

```
grafo = GrafoPonderado()
v1 = grafo.cria_vertice('1')
v2 = grafo.cria_vertice('2')
v3 = grafo.cria_vertice('3')
v4 = grafo.cria_vertice('4')
v5 = grafo.cria_vertice('5')
v6 = grafo.cria_vertice('6')
v7 = grafo.cria_vertice('7')
v8 = grafo.cria_vertice('8')
v9 = grafo.cria_vertice('9')
v10 = grafo.cria_vertice('10')
v11 = grafo.cria_vertice('11')
vm = grafo.cria_vertice('M')
```

9. **A função abaixo recebe um array que deveria conter todos os números inteiros de 0 a N. Entretanto, falta um nº no array e a função deve retorná-lo. Por exemplo, no array [2, 3, 0, 6, 1, 5] falta o 4 e no array [8, 2, 3, 9, 4, 7, 5, 0, 6] falta o 1.**
   Usando esta abordagem com laços aninhados, a função tem complexidade $O(N^2)$ no pior caso. Você deve otimizá-la para que ela realize a mesma tarefa com complexidade O(N).

R:

```
def numero_faltante(array):
    n = len(array)
    soma_esperada = n * (n + 1) // 2
    soma_atual = sum(array)
    return soma_esperada - soma_atual

print(numero_faltante([2, 3, 0, 6, 1, 5]))
print(numero_faltante([8, 2, 3, 9, 4, 7, 5, 0, 6]))
```

Saida:

```
AzureAD+IanSantos@BRALFS
$ C:/Users/IanSantos/App
/at/q9.py
4
1
```

10. **A função abaixo recebe um array de números e calcula o maior produto de quaisquer dois números no array. À primeira vista, isto parece fácil, pois podemos simplesmente encontrar os dois maiores números e multiplicá-los. Entretanto, o array pode conter números negativos. Por exemplo, no array [5, -10, -6, 9, 4] o maior produto é -10 x -6 = 60. Usando laços aninhados para multiplicar todos os pares de números possíveis, a função tem complexidade O(N^2). Você deve otimizá-la para que ela realize a mesma tarefa com complexidade O(N).**

R:

```
turas_de_Dados_e_Algoritmos_Avançados > at > q10.py > ...
def maior_produto(array):
    if len(array) < 2:
        return None

    maior1 = maior2 = float('-inf')
    menor1 = menor2 = float('inf')

    for num in array:
        if num > maior1:
            maior2 = maior1
            maior1 = num
        elif num > maior2:
            maior2 = num

        if num < menor1:
            menor2 = menor1
            menor1 = num
        elif num < menor2:
            menor2 = num

    return max(maior1 * maior2, menor1 * menor2)

# Exemplos de uso
print(maior_produto([5, -10, -6, 9, 4]))
```

Saida:

```
AzureAD+IanSantos@BRALFSUSABIT03 MINGW64 ~
$ C:/Users/IanSantos/AppData/Local/Microso
/at/q10.py
60
```

11. **A função abaixo recebe 2 arrays e verifica se ambos contêm os mesmos elementos (ignorando a ordem). Essa função tem complexidade O(N*M), onde N e M são os comprimentos dos arrays. Você deve otimizá-la para que ela realize a mesma tarefa com complexidade O(N+M).**

R:

```
def iguais(array1, array2):
    return set(array1) == set(array2)

print(iguais([1, 2, 3], [3, 2, 1]))
print(iguais([1, 2, 2], [2, 1, 1]))
```

Saida:

```
AzureAD+IanSantos@
$ C:/Users/IanSant
/at/q11.py
True
True
```

Libs:

https://docs.python.org/3/library/datetime.html

https://replit.com/@marcelorhmaia/Grafos-ponderados?v=1#grafos.py

https://docs.python.org/3/library/itertools.html#itertools.permutations

https://numpy.org/

https://matplotlib.org/

https://networkx.org/

https://docs.python.org/3/library/heapq.html

https://docs.python.org/3/library/math.html