

AV2Simulado

June 10, 2025

0.1 Árvore Geradora Mínima e Caminho Mínimo em Grafos

Vamos entender de forma simples e visual dois conceitos essenciais em grafos: **árvore geradora mínima** e **caminho mínimo**. Ambos são fundamentais para resolver problemas de conectividade e otimização em redes, logística, energia, transporte e muito mais.

0.1.1 Árvore Geradora Mínima

Imagine que você precisa conectar várias cidades (vértices) usando estradas (arestas), mas quer gastar o mínimo possível e garantir que todas estejam conectadas, sem formar voltas (ciclos). A **árvore geradora mínima** é exatamente isso: um subgrafo que conecta todos os vértices com o menor custo total possível, sem ciclos, usando exatamente $n - 1$ arestas (onde n é o número de vértices).

- **Exemplo prático:** Instalar cabos entre cidades gastando o mínimo possível.
 - **Principais algoritmos:**
 - **Kruskal:** Escolhe sempre a estrada mais barata que não forma ciclo, até conectar tudo.
 - **Prim:** Começa em um vértice e vai conectando o mais próximo, sempre escolhendo a estrada mais barata para expandir a rede.
-

0.1.2 Caminho Mínimo

Agora, suponha que você quer ir de uma cidade até outra, passando pelo menor número de estradas ou gastando o menor custo possível. O **caminho mínimo** é a rota mais curta (em número de arestas ou em custo total) entre dois vértices.

- **Exemplo prático:** Encontrar a rota mais curta ou mais barata entre dois pontos de uma cidade.
 - **Principais algoritmos:**
 - **Busca em Largura (BFS):** Para grafos sem pesos, encontra o caminho com menos arestas.
 - **Dijkstra:** Para grafos com pesos positivos, encontra o caminho de menor custo.
 - **Bellman-Ford:** Para grafos com pesos negativos (mas sem ciclos negativos).
-

0.1.3 Relação e Aplicação

- **Árvore geradora mínima** resolve o problema de conectar todos os pontos de uma rede com o menor custo total, sem redundâncias.
- **Caminho mínimo** resolve o problema de ir de um ponto a outro da forma mais eficiente possível.

Ambos são usados para garantir eficiência, economia e conectividade em sistemas reais. Saber aplicar cada conceito no contexto certo é fundamental para otimizar redes e rotas!

0.1.4 Questão 01 - Arvore Geradora Mínima

Uma empresa de energia deseja instalar cabos para conectar 7 bairros de forma eficiente. A rede deve ser construída de modo que:

Todos os bairros estejam conectados direta ou indiretamente. Não existam ciclos na rede. A remoção de qualquer cabo desconecte a rede. Com base nessas condições e nas definições de árvores em grafos, qual é o número mínimo de cabos necessários para conectar todos os bairros?

```
[89]: ## Questão 1
import networkx as nx

n_bairros = 7
# Em uma árvore com n vértices, o número de arestas é n-1
arestas_necessarias = n_bairros - 1
print(f"Número de cabos necessários: {arestas_necessarias}")
```

Número de cabos necessários: 6

Explicação Detalhada sobre Árvore A situação apresentada corresponde à definição de uma **árvore** em Teoria dos Grafos. Vamos analisar cada condição:

1. **Todos os bairros devem estar conectados direta ou indiretamente.**

Isso significa que o grafo deve ser **conexo**: é possível ir de qualquer bairro a qualquer outro, mesmo que seja passando por outros bairros intermediários.

2. **Não pode haver ciclos na rede de cabos.**

Um ciclo ocorre quando é possível sair de um bairro, passar por outros e retornar ao bairro inicial sem repetir arestas. A ausência de ciclos caracteriza um grafo **acíclico**.

3. **Se qualquer cabo for removido, a rede ficará desconectada.**

Isso significa que todas as conexões são essenciais para manter a rede unida. Em outras palavras, o grafo é **minimamente conexo**: não há arestas redundantes.

Essas três propriedades juntas definem uma **árvore**. Em uma árvore com n vértices (neste caso, bairros), sempre existem exatamente $n - 1$ arestas (cabos). Isso é uma propriedade fundamental das árvores em grafos.

Aplicando ao problema:

- Temos $n = 7$ bairros. - Logo, o número mínimo de cabos necessários é $n - 1 = 6$.

Se usarmos menos de 6 cabos, não conseguiremos conectar todos os bairros. Se usarmos mais de 6 cabos, obrigatoriamente haverá pelo menos um ciclo, o que viola a condição 2.

Portanto, a resposta correta é **6 cabos**.

0.1.5 Questão 2: Busca em Largura (BFS)

Uma empresa de energia deseja monitorar o fornecimento em uma rede de subestações, representada por um grafo não direcionado. Os vértices são subestações e as arestas representam linhas de transmissão entre elas. Para garantir o monitoramento eficiente, a empresa precisa calcular o vetor de roteamento correto para alcançar todas as subestações a partir de uma subestação inicial, utilizando a Busca em Largura (BFS).

Considere o grafo abaixo, onde as subestações e as linhas são:

1. Subestação X - Subestação Y
2. Subestação X - Subestação Z
3. Subestação Y - Subestação W
4. Subestação Z - Subestação W
5. Subestação Z - Subestação V
6. Subestação W - Subestação U
7. Subestação V - Subestação U

Qual vetor de roteamento, gerado a partir da Subestação X, está correto?

0.2 Busca em Largura (BFS)

A **Busca em Largura** (BFS - *Breadth-First Search*) é um algoritmo utilizado para percorrer ou buscar elementos em grafos (ou árvores). O objetivo é visitar todos os vértices acessíveis a partir de um vértice inicial, explorando primeiro os vizinhos mais próximos antes de avançar para os mais distantes.

0.2.1 Como funciona a BFS?

1. **Escolha um vértice inicial** e marque-o como visitado.
2. **Coloque o vértice em uma fila.**
3. Enquanto a fila não estiver vazia:
 - Remova o primeiro vértice da fila.
 - Para cada vizinho não visitado desse vértice:
 - Marque o vizinho como visitado.
 - Adicione o vizinho à fila.

0.2.2 Características

- Garante encontrar o caminho mais curto (menor número de arestas) em grafos não ponderados.
- Utiliza uma **fila** para controlar a ordem de visitação.
- Visita todos os vértices a uma distância antes de avançar para a próxima.

0.2.3 Exemplo de aplicação

- Encontrar o caminho mais curto entre dois pontos em um mapa.
- Verificar se um grafo é conexo.
-

0.3 Encontrar todos os vértices a uma certa distância de um ponto.

Resumo:

A BFS é ideal para explorar grafos em “camadas”, sendo muito útil para encontrar caminhos mínimos e analisar a estrutura de redes.

```
[90]: # Resolvendo o vetor de roteamento usando BFS a partir da Subestação X
import networkx as nx

# Criação do grafo
G = nx.Graph()
G.add_edges_from([
    ('X', 'Y'),
    ('X', 'Z'),
    ('Y', 'W'),
    ('Z', 'W'),
    ('Z', 'V'),
    ('W', 'U'),
    ('V', 'U')
])

# BFS a partir da subestação X
from collections import deque
import matplotlib.pyplot as plt

def bfs_routing_vector_verbose(graph, start):
    visitados = set()
    fila = deque([start])
    vetor = []
    predecessor = {start: None}
    passo = 0

    pos = nx.spring_layout(graph, seed=42) # posição fixa para todos os plots

    print(f"Passo {passo}:")
    print(f"  Fila inicial: {list(fila)}")
    print(f"  Visitados: {vetor}")
    print(f"  Predecessores: {predecessor}")
    print(f"  Nenhum vértice processado ainda.\n")
    plt.figure(figsize=(6,4))
    nx.draw(
```

```

graph, pos, with_labels=True,
node_color=['orange' if n == start else 'lightgray' for n in graph.
↪nodes],
node_size=800, font_weight='bold',
edge_color='#888', width=2
)
plt.title(f"BFS - Passo {passo} (Inicial: {start})")
plt.show()
passo += 1

while fila:
    atual = fila.popleft()
    print(f"Passo {passo}:")
    print(f"  Fila antes de processar: {list(fila)}")
    print(f"  Visitados até agora: {vetor}")
    print(f"  Vértice atual: {atual}")
    print(f"  Predecessores: {predecessor}")

    # Cores dos nós
    node_colors = []
    for n in graph.nodes:
        if n == atual:
            node_colors.append('orange')          # atual
        elif n in vetor:
            node_colors.append('lightgreen')      # já visitado
        elif n in fila:
            node_colors.append('yellow')          # na fila
        else:
            node_colors.append('lightgray')       # não visitado

    # Cores das arestas: destaque as arestas do nó atual
    edge_colors = []
    for u, v in graph.edges:
        if (u == atual and v in graph.neighbors(atual)) or (v == atual and
↪u in graph.neighbors(atual)):
            edge_colors.append('red')
        elif u in vetor and v in vetor:
            edge_colors.append('green')
        else:
            edge_colors.append('#888')

    plt.figure(figsize=(6,4))
    nx.draw(
        graph, pos, with_labels=True,
        node_color=node_colors, node_size=800, font_weight='bold',
        edge_color=edge_colors, width=2
    )

```

```

plt.title(f"BFS - Passo {passo} (Atual: {atual})")
plt.show()

if atual not in visitados:
    visitados.add(atual)
    vetor.append(atual)
    for vizinho in sorted(graph.neighbors(atual)):
        if vizinho not in visitados and vizinho not in fila:
            fila.append(vizinho)
            if vizinho not in predecessor:
                predecessor[vizinho] = atual
print(f" Fila após adicionar vizinhos: {list(fila)}")
print(f" Predecessores atualizados: {predecessor}\n")
passo += 1
print("Vetor de roteamento final:", vetor)
print("Predecessores finais:", predecessor)
return vetor, predecessor

# Executando a versão detalhada
bfs_routing_vector_verbose(G, 'X')

```

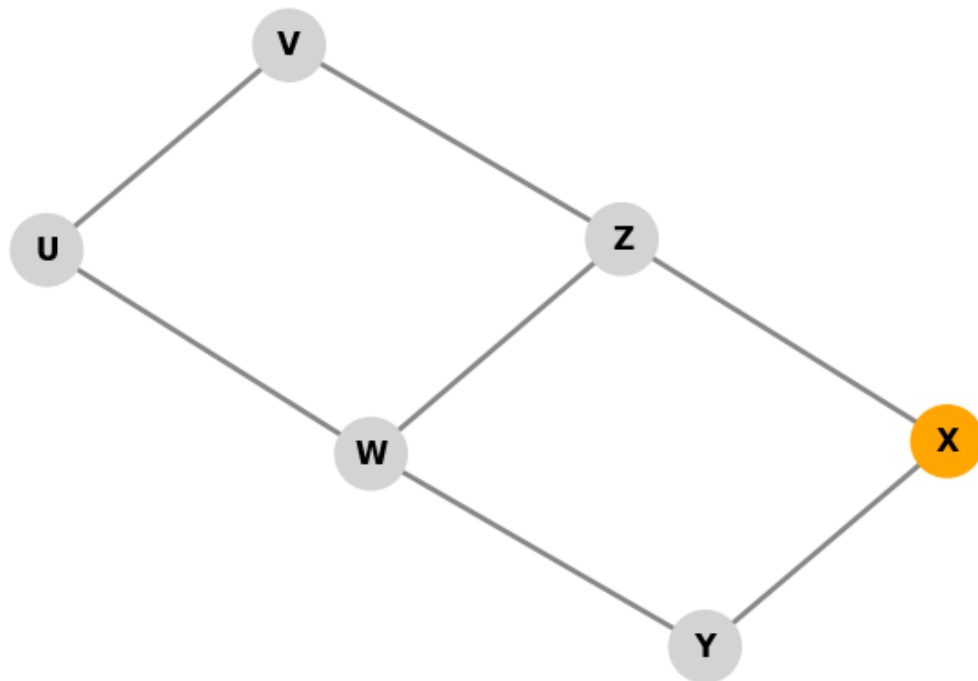
Passo 0:

```

Fila inicial: ['X']
Visitados: []
Predecessores: {'X': None}
Nenhum vértice processado ainda.

```

BFS - Passo 0 (Inicial: X)



Passo 1:

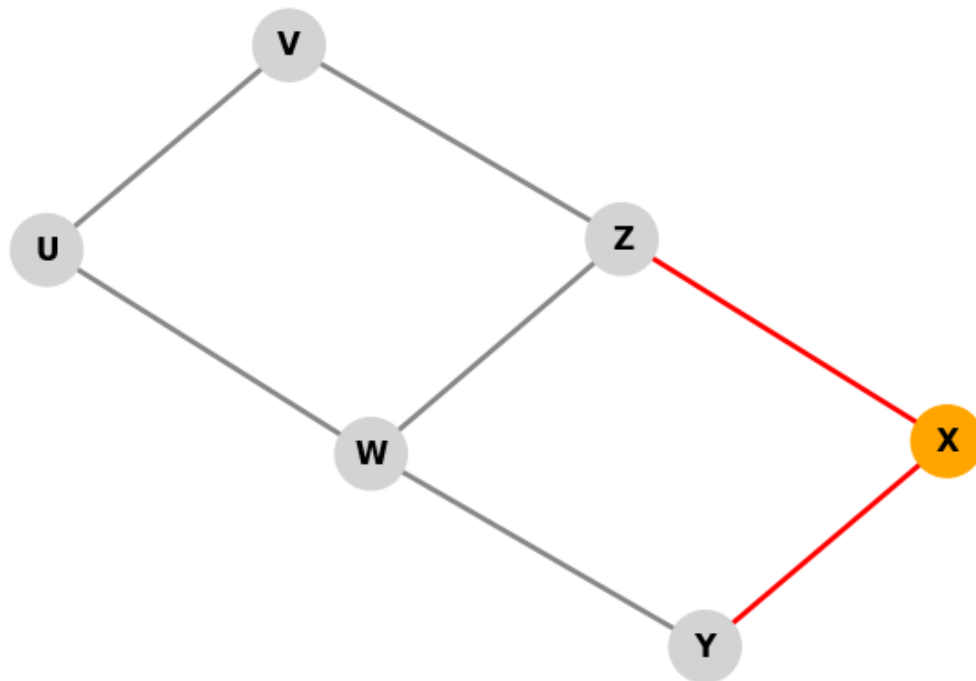
Fila antes de processar: []

Visitados até agora: []

Vértice atual: X

Predecessores: {'X': None}

BFS - Passo 1 (Atual: X)



Fila após adicionar vizinhos: ['Y', 'Z']

Predecessores atualizados: {'X': None, 'Y': 'X', 'Z': 'X'}

Passo 2:

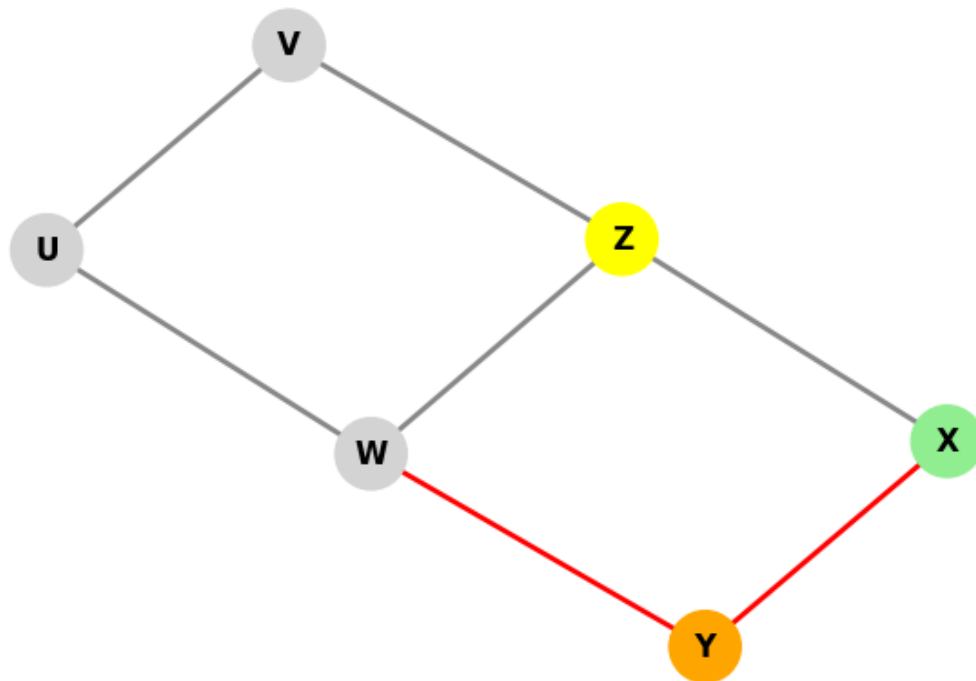
Fila antes de processar: ['Z']

Visitados até agora: ['X']

Vértice atual: Y

Predecessores: {'X': None, 'Y': 'X', 'Z': 'X'}

BFS - Passo 2 (Atual: Y)



Fila após adicionar vizinhos: ['Z', 'W']

Predecessores atualizados: {'X': None, 'Y': 'X', 'Z': 'X', 'W': 'Y'}

Passo 3:

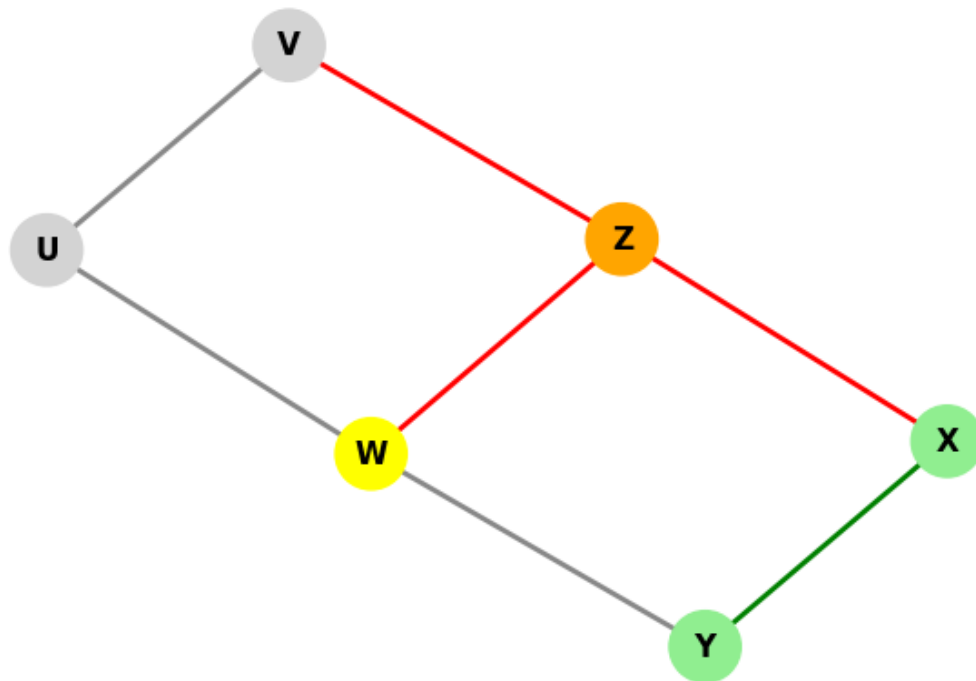
Fila antes de processar: ['W']

Visitados até agora: ['X', 'Y']

Vértice atual: Z

Predecessores: {'X': None, 'Y': 'X', 'Z': 'X', 'W': 'Y'}

BFS - Passo 3 (Atual: Z)



Fila após adicionar vizinhos: ['W', 'V']

Predecessores atualizados: {'X': None, 'Y': 'X', 'Z': 'X', 'W': 'Y', 'V': 'Z'}

Passo 4:

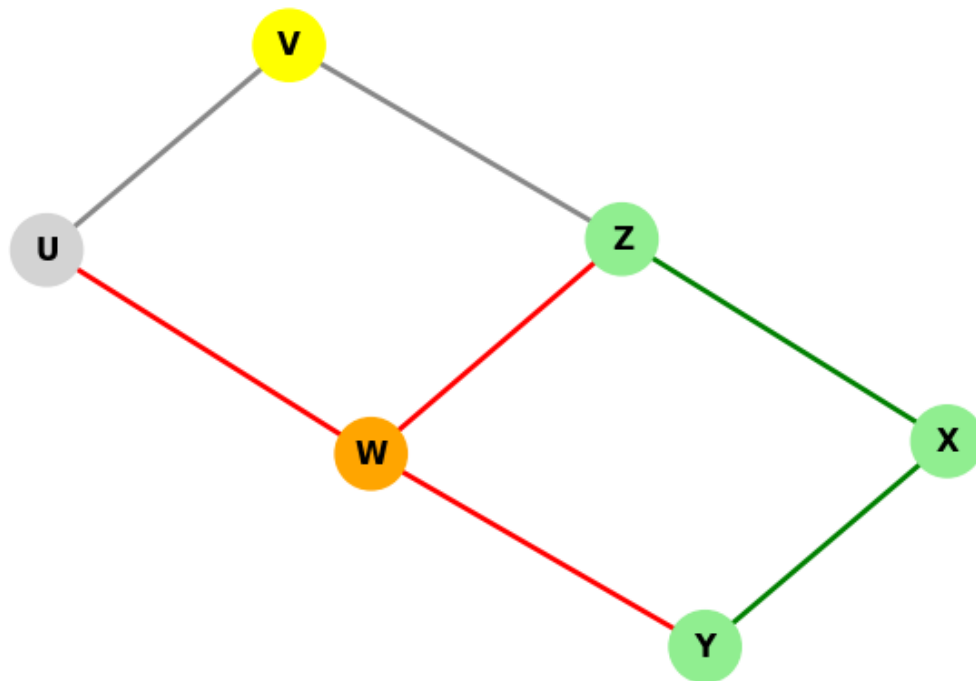
Fila antes de processar: ['V']

Visitados até agora: ['X', 'Y', 'Z']

Vértice atual: W

Predecessores: {'X': None, 'Y': 'X', 'Z': 'X', 'W': 'Y', 'V': 'Z'}

BFS - Passo 4 (Atual: W)

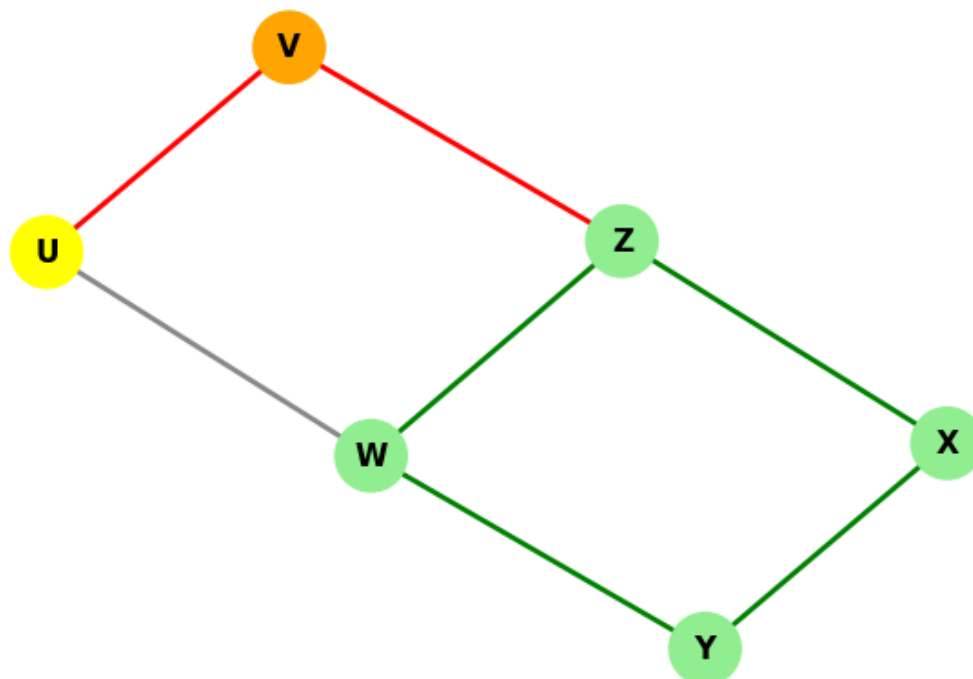


Fila após adicionar vizinhos: ['V', 'U']
Predecessores atualizados: {'X': None, 'Y': 'X', 'Z': 'X', 'W': 'Y', 'V': 'Z', 'U': 'W'}

Passo 5:

Fila antes de processar: ['U']
Visitados até agora: ['X', 'Y', 'Z', 'W']
Vértice atual: V
Predecessores: {'X': None, 'Y': 'X', 'Z': 'X', 'W': 'Y', 'V': 'Z', 'U': 'W'}

BFS - Passo 5 (Atual: V)

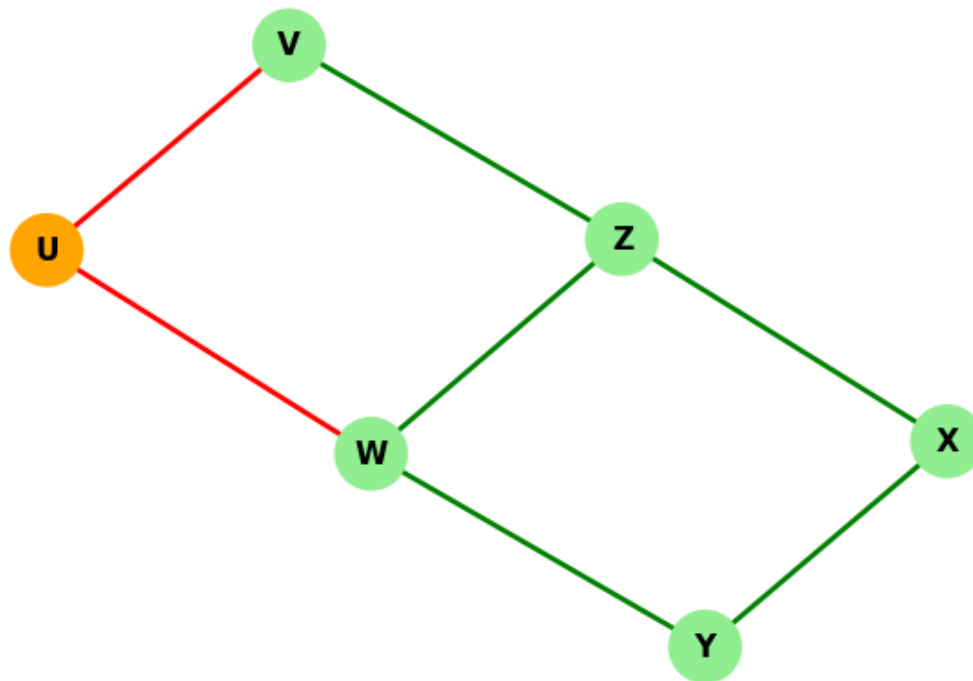


Fila após adicionar vizinhos: ['U']
Predecessores atualizados: {'X': None, 'Y': 'X', 'Z': 'X', 'W': 'Y', 'V': 'Z', 'U': 'W'}

Passo 6:

Fila antes de processar: []
Visitados até agora: ['X', 'Y', 'Z', 'W', 'V']
Vértice atual: U
Predecessores: {'X': None, 'Y': 'X', 'Z': 'X', 'W': 'Y', 'V': 'Z', 'U': 'W'}

BFS - Passo 6 (Atual: U)



Fila após adicionar vizinhos: []

Predecessores atualizados: {'X': None, 'Y': 'X', 'Z': 'X', 'W': 'Y', 'V': 'Z', 'U': 'W'}

Vetor de roteamento final: ['X', 'Y', 'Z', 'W', 'V', 'U']

Predecessores finais: {'X': None, 'Y': 'X', 'Z': 'X', 'W': 'Y', 'V': 'Z', 'U': 'W'}

[90]: (['X', 'Y', 'Z', 'W', 'V', 'U'],
{ 'X': None, 'Y': 'X', 'Z': 'X', 'W': 'Y', 'V': 'Z', 'U': 'W' })

Explicação Detalhada A questão pede o vetor de roteamento gerado pela **Busca em Largura (BFS)**, que é um algoritmo para percorrer grafos visitando todos os vértices acessíveis a partir de um vértice inicial, explorando primeiro os vizinhos mais próximos.

No contexto do problema: - Cada subestação é um vértice. - Cada linha de transmissão é uma aresta. - O vetor de roteamento é a ordem em que as subestações são visitadas a partir da subestação inicial (X), seguindo a BFS.

Como funciona a BFS: 1. Começa pelo vértice inicial (X), marcando-o como visitado. 2. Visita todos os vizinhos diretos de X (Y e Z). 3. Em seguida, visita os vizinhos dos vizinhos, e assim por diante, sempre explorando primeiro os vértices mais próximos do ponto de partida. 4. O algoritmo garante que cada vértice é visitado apenas uma vez.

No exemplo acima, o vetor de roteamento pode ser:

['X', 'Y', 'Z', 'W', 'V', 'U']

A ordem pode variar dependendo da ordem dos vizinhos, mas sempre segue o princípio de visitar primeiro os vértices mais próximos do inicial.

Portanto, o vetor de roteamento correto é aquele que reflete a ordem de visitação da BFS a partir da subestação inicial.

```
[91]: # Exemplo: Encontrando o caminho mais curto entre duas subestações usando BFS
      ↪ (NetworkX)
      origem = 'X'
      destino = 'U'
      caminho_mais_curto = nx.shortest_path(G, source=origem, target=destino)
      print(f"Caminho mais curto de {origem} até {destino}: {caminho_mais_curto}")

      # Visualizando o caminho encontrado
      import matplotlib.pyplot as plt

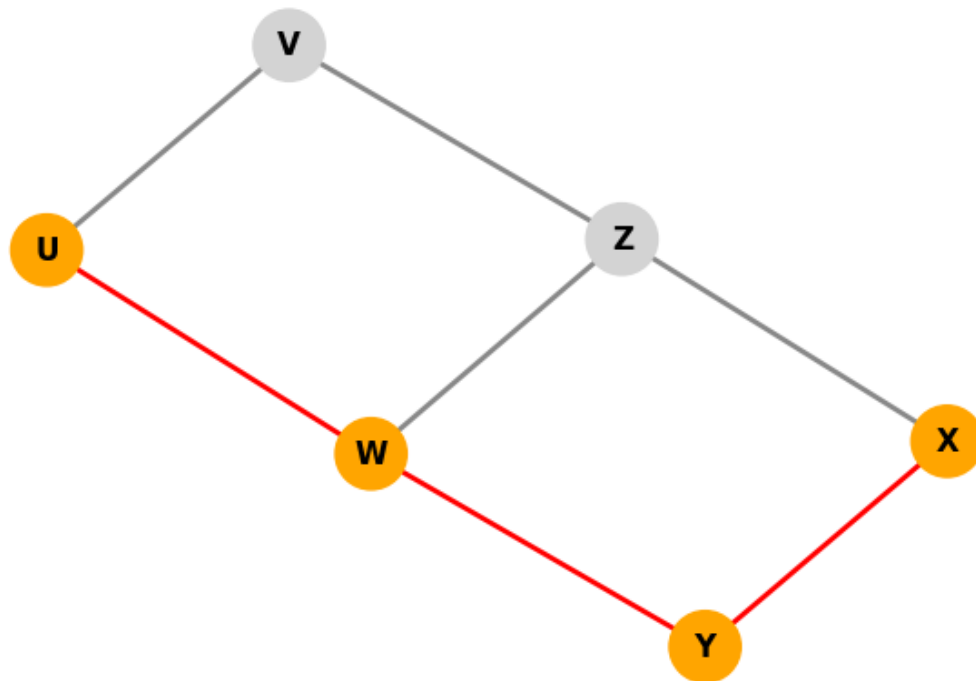
      pos = nx.spring_layout(G, seed=42)
      edge_colors = []
      for u, v in G.edges:
          if (u, v) in zip(caminho_mais_curto, caminho_mais_curto[1:]) or (v, u) in ↪
          ↪ zip(caminho_mais_curto, caminho_mais_curto[1:]):
              edge_colors.append('red')
          else:
              edge_colors.append('#888')

      node_colors = ['orange' if n in caminho_mais_curto else 'lightgray' for n in G. ↪
      ↪ nodes]

      plt.figure(figsize=(6,4))
      nx.draw(G, pos, with_labels=True, node_color=node_colors, node_size=800, ↪
      ↪ font_weight='bold', edge_color=edge_colors, width=2)
      plt.title(f"Caminho mais curto de {origem} até {destino}")
      plt.show()
```

Caminho mais curto de X até U: ['X', 'Y', 'W', 'U']

Caminho mais curto de X até U



Exemplo de busca de caminho No exemplo acima, utilizamos o algoritmo de caminho mais curto do NetworkX (que, em grafos não ponderados, utiliza a Busca em Largura - BFS) para encontrar o caminho mais curto entre a subestação X e a subestação U.

Caminho encontrado:

['X', 'Y', 'W', 'U']

Ou seja, para ir de X até U, o trajeto mais curto passa por Y e W.

Explicação:

- O algoritmo procura o menor número de arestas entre o ponto de origem e o destino. - No grafo dado, existem múltiplos caminhos possíveis entre X e U, mas o caminho com menos arestas é $X \rightarrow Y \rightarrow W \rightarrow U$ (com 3 arestas). - O caminho é destacado em vermelho no grafo plotado.

Portanto, a resposta correta para o caminho mais curto entre X e U é:

['X', 'Y', 'W', 'U']

0.3.1 Questão 3: Busca em Profundidade (DFS)

Uma empresa de tecnologia deseja analisar a conectividade entre os computadores de sua rede interna. A rede é representada como um grafo, onde os vértices são computadores e as arestas representam conexões diretas entre eles. A empresa quer calcular o vetor de roteamento correto para alcançar todos os computadores a partir de um computador inicial, utilizando a busca em profundidade (DFS).

Considere o grafo abaixo, onde os computadores e as conexões são representados como:

1. Computador A - Computador B
2. Computador A - Computador C
3. Computador B - Computador D
4. Computador C - Computador D
5. Computador C - Computador E
6. Computador D - Computador F
7. Computador E - Computador F

Qual vetor de roteamento, gerado a partir do Computador A, está correto? Mostre o passo a passo da DFS.

0.4 Busca em Profundidade (DFS)

A **Busca em Profundidade** (DFS - *Depth-First Search*) é um algoritmo utilizado para percorrer ou buscar elementos em grafos (ou árvores). O objetivo é explorar o grafo indo o mais fundo possível em cada ramificação antes de retroceder.

0.4.1 Como funciona a DFS?

1. **Escolha um vértice inicial** e marque-o como visitado.
2. Para cada vizinho não visitado desse vértice:
 - Visite o vizinho e repita o processo recursivamente.
3. Se não houver mais vizinhos não visitados, volte (retroceda) para o vértice anterior e continue a busca.

A DFS pode ser implementada de forma recursiva ou utilizando uma pilha.

0.4.2 Características

- Explora o grafo em profundidade, indo até o final de cada caminho antes de voltar.
- Utiliza uma **pilha** (explícita ou implícita via recursão) para controlar a ordem de visitação.
- Pode ser usada para detectar ciclos, componentes conexas, caminhos e ordenação topológica.

0.4.3 Exemplo de aplicação

- Verificar se existe um caminho entre dois vértices.
 - Encontrar componentes conexas em grafos.
 - Resolver labirintos.
 - Ordenação topológica em grafos direcionados acíclicos.
-

Resumo:

A DFS é ideal para explorar todos os caminhos possíveis a partir de um vértice, sendo muito útil para análise estrutural de grafos e resolução de problemas que exigem percorrer todas as possibilidades.

```
[92]: # Passo a passo da DFS com visualização

import networkx as nx
import matplotlib.pyplot as plt

G_dfs = nx.Graph()
G_dfs.add_edges_from([
    ('A', 'B'),
    ('A', 'C'),
    ('B', 'D'),
    ('C', 'D'),
    ('C', 'E'),
    ('D', 'F'),
    ('E', 'F')
])

def dfs_routing_vector_verbose(graph, start):
    visitados = set()
    vetor = []
    pilha = [start]
    predecessor = {start: None}
    passo = 0
    pos = nx.spring_layout(graph, seed=42)

    print(f"Passo {passo}:")
    print(f"  Pilha inicial: {pilha}")
    print(f"  Visitados: {vetor}")
    print(f"  Predecessores: {predecessor}")
    print(f"  Nenhum vértice processado ainda.\n")
    plt.figure(figsize=(6,4))
    nx.draw(
        graph, pos, with_labels=True,
        node_color=['orange' if n == start else 'lightgray' for n in graph.
↪nodes],
        node_size=800, font_weight='bold',
        edge_color='#888', width=2
    )
    plt.title(f"DFS - Passo {passo} (Inicial: {start})")
    plt.show()
    passo += 1

    while pilha:
        atual = pilha.pop()
```

```

if atual not in visitados:
    print(f"Passo {passo}:")
    print(f"  Pilha: {pilha}")
    print(f"  Visitados: {vetor}")
    print(f"  Vértice atual: {atual}")
    print(f"  Predecessores: {predecessor}")

    # Cores dos nós
    node_colors = []
    for n in graph.nodes:
        if n == atual:
            node_colors.append('orange')
        elif n in vetor:
            node_colors.append('lightgreen')
        elif n in pilha:
            node_colors.append('yellow')
        else:
            node_colors.append('lightgray')

    # Cores das arestas: destaque as arestas do nó atual
    edge_colors = []
    for u, v in graph.edges:
        if (u == atual and v in graph.neighbors(atual)) or (v == atual
↪and u in graph.neighbors(atual)):
            edge_colors.append('red')
        elif u in vetor and v in vetor:
            edge_colors.append('green')
        else:
            edge_colors.append('#888')

    plt.figure(figsize=(6,4))
    nx.draw(
        graph, pos, with_labels=True,
        node_color=node_colors, node_size=800, font_weight='bold',
        edge_color=edge_colors, width=2
    )
    plt.title(f"DFS - Passo {passo} (Atual: {atual})")
    plt.show()

    visitados.add(atual)
    vetor.append(atual)
    # Adiciona vizinhos não visitados na pilha (ordem reversa para
↪manter ordem alfabética)
    for vizinho in sorted(graph.neighbors(atual), reverse=True):
        if vizinho not in visitados and vizinho not in pilha:
            pilha.append(vizinho)
            if vizinho not in predecessor:

```

```

        predecessor[vizinho] = atual
    print(f" Pilha após adicionar vizinhos: {pilha}")
    print(f" Predecessores atualizados: {predecessor}\n")
    passo += 1
print("Vetor de roteamento final (DFS):", vetor)
print("Predecessores finais:", predecessor)
return vetor, predecessor

# Executando a versão detalhada
dfs_routing_vector_verbose(G_dfs, 'A')

```

Passo 0:

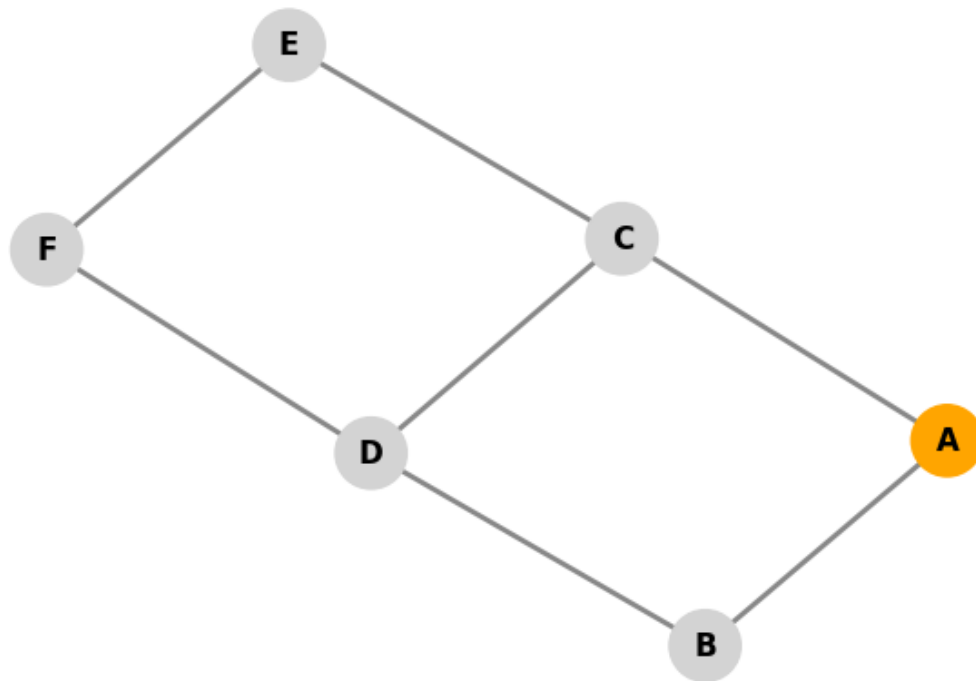
Pilha inicial: ['A']

Visitados: []

Predecessores: {'A': None}

Nenhum vértice processado ainda.

DFS - Passo 0 (Inicial: A)



Passo 1:

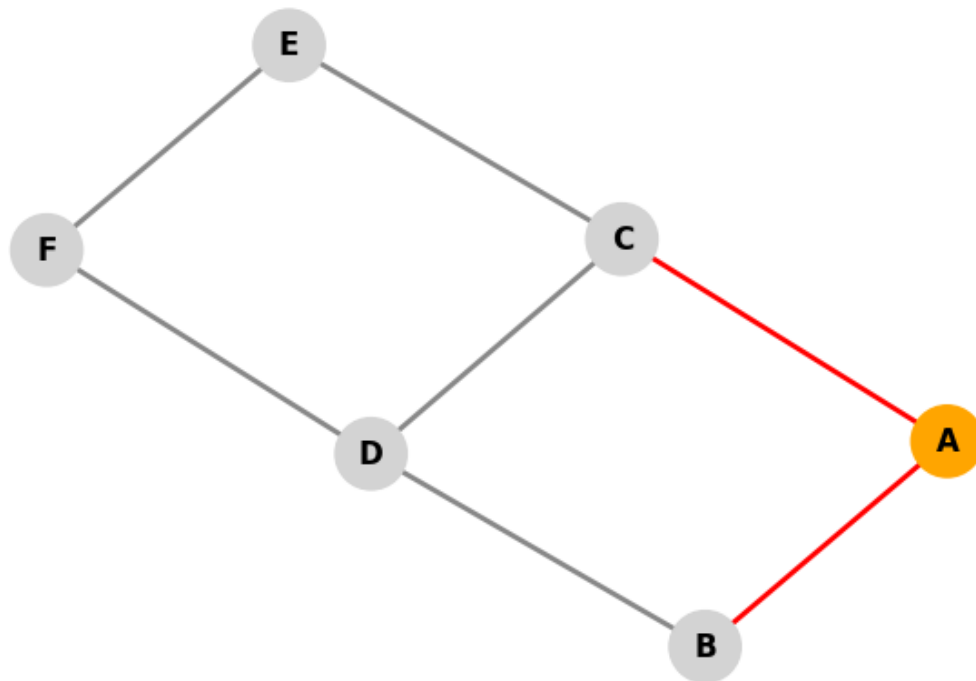
Pilha: []

Visitados: []

Vértice atual: A

Predecessores: {'A': None}

DFS - Passo 1 (Atual: A)



Pilha após adicionar vizinhos: ['C', 'B']

Predecessores atualizados: {'A': None, 'C': 'A', 'B': 'A'}

Passo 2:

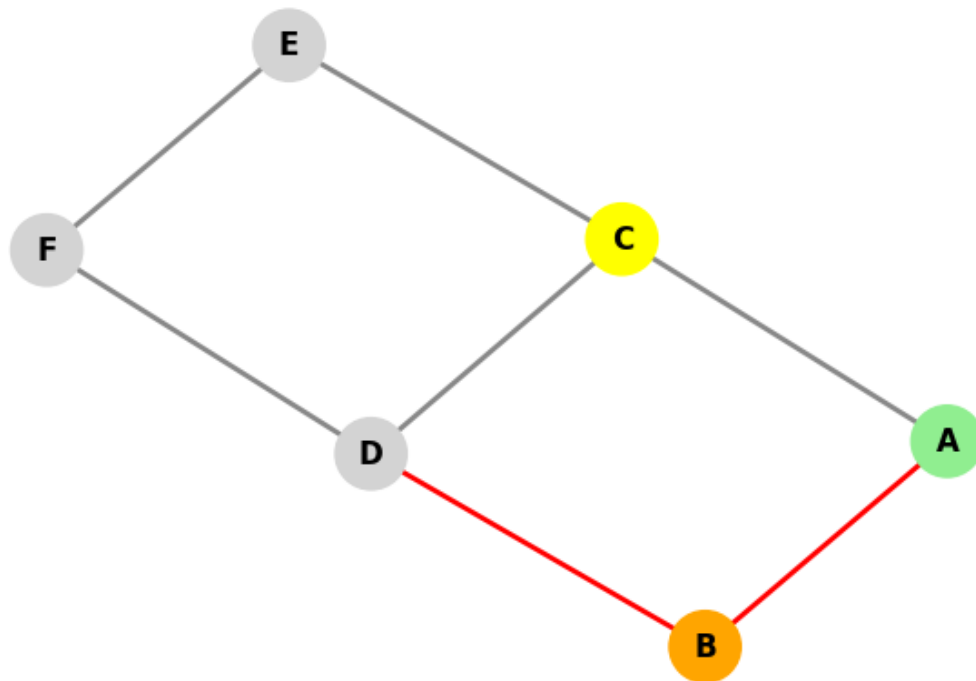
Pilha: ['C']

Visitados: ['A']

Vértice atual: B

Predecessores: {'A': None, 'C': 'A', 'B': 'A'}

DFS - Passo 2 (Atual: B)



Pilha após adicionar vizinhos: ['C', 'D']

Predecessores atualizados: {'A': None, 'C': 'A', 'B': 'A', 'D': 'B'}

Passo 3:

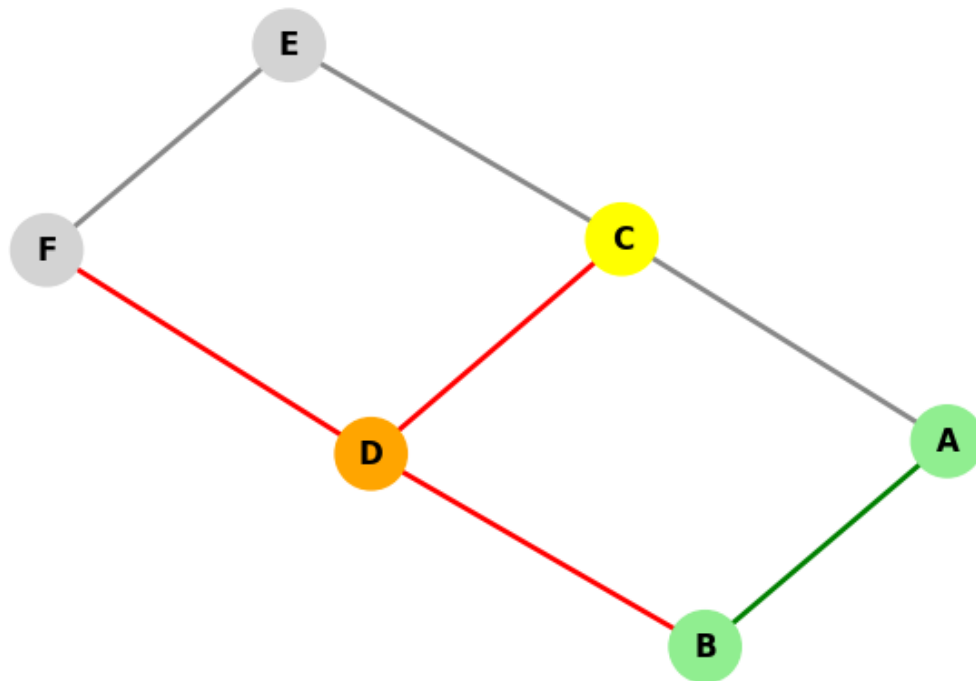
Pilha: ['C']

Visitados: ['A', 'B']

Vértice atual: D

Predecessores: {'A': None, 'C': 'A', 'B': 'A', 'D': 'B'}

DFS - Passo 3 (Atual: D)



Pilha após adicionar vizinhos: ['C', 'F']

Predecessores atualizados: {'A': None, 'C': 'A', 'B': 'A', 'D': 'B', 'F': 'D'}

Passo 4:

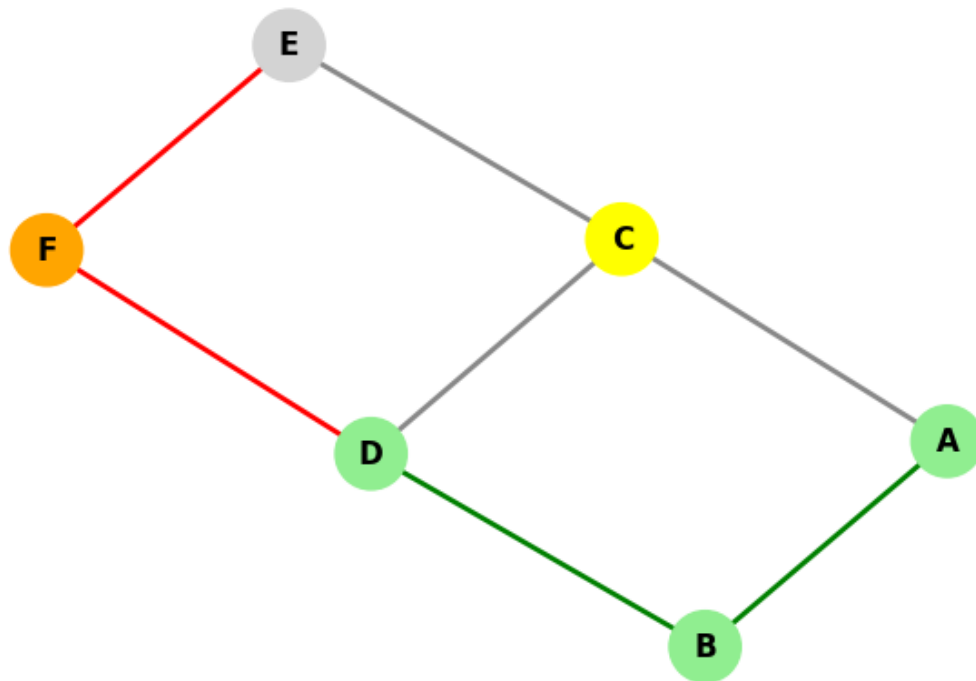
Pilha: ['C']

Visitados: ['A', 'B', 'D']

Vértice atual: F

Predecessores: {'A': None, 'C': 'A', 'B': 'A', 'D': 'B', 'F': 'D'}

DFS - Passo 4 (Atual: F)



Pilha após adicionar vizinhos: ['C', 'E']
Predecessores atualizados: {'A': None, 'C': 'A', 'B': 'A', 'D': 'B', 'F': 'D', 'E': 'F'}

Passo 5:

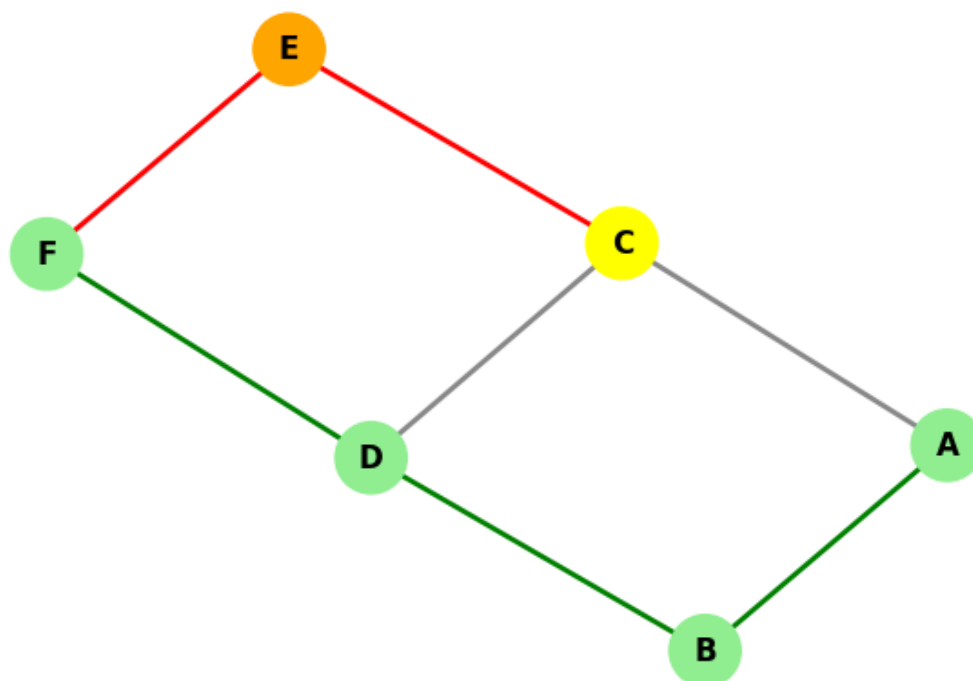
Pilha: ['C']

Visitados: ['A', 'B', 'D', 'F']

Vértice atual: E

Predecessores: {'A': None, 'C': 'A', 'B': 'A', 'D': 'B', 'F': 'D', 'E': 'F'}

DFS - Passo 5 (Atual: E)



Pilha após adicionar vizinhos: ['C']
Predecessores atualizados: {'A': None, 'C': 'A', 'B': 'A', 'D': 'B', 'F': 'D', 'E': 'F'}

Passo 6:

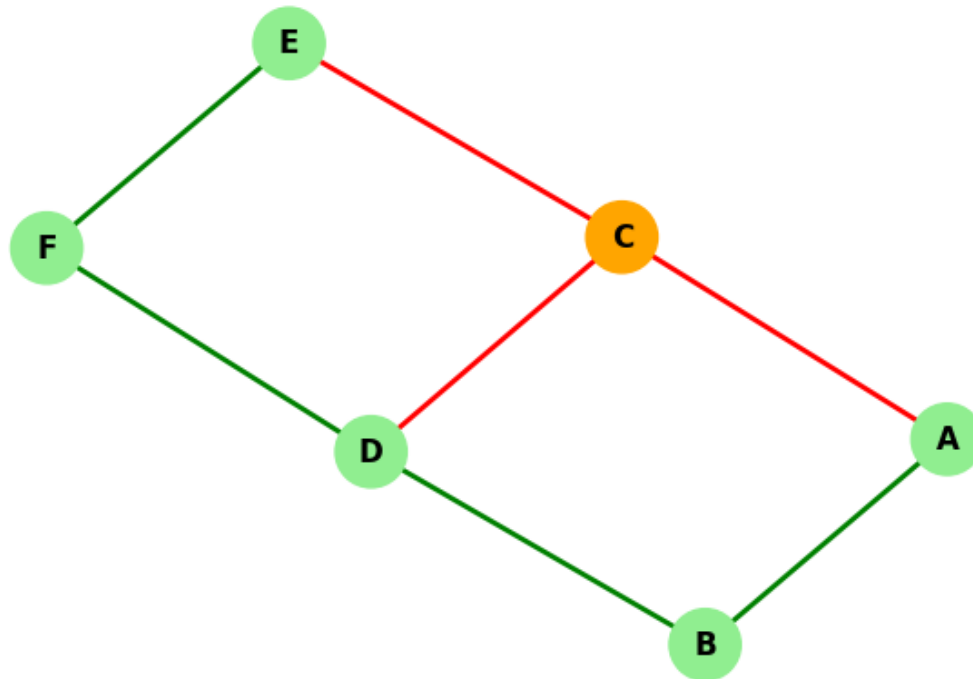
Pilha: []

Visitados: ['A', 'B', 'D', 'F', 'E']

Vértice atual: C

Predecessores: {'A': None, 'C': 'A', 'B': 'A', 'D': 'B', 'F': 'D', 'E': 'F'}

DFS - Passo 6 (Atual: C)



Pilha após adicionar vizinhos: []
Predecessores atualizados: {'A': None, 'C': 'A', 'B': 'A', 'D': 'B', 'F': 'D', 'E': 'F'}

Vetor de roteamento final (DFS): ['A', 'B', 'D', 'F', 'E', 'C']
Predecessores finais: {'A': None, 'C': 'A', 'B': 'A', 'D': 'B', 'F': 'D', 'E': 'F'}

[92]: (['A', 'B', 'D', 'F', 'E', 'C'],
 {'A': None, 'C': 'A', 'B': 'A', 'D': 'B', 'F': 'D', 'E': 'F'})

0.4.4 Explicação

A busca em profundidade (DFS) explora o grafo indo o mais fundo possível em cada ramificação antes de retroceder. O vetor de roteamento mostra a ordem em que os computadores são visitados a partir do computador inicial (A).

No exemplo acima, o passo a passo mostra a pilha, os visitados e o vértice atual em cada etapa, além de uma visualização do grafo. O vetor final corresponde à ordem de visitação da DFS.

```
[93]: # Exemplo: Encontrando um caminho entre dois computadores usando DFS (NetworkX)
      origem = 'A'
      destino = 'F'
```

```

caminho_dfs = list(nx.dfs_edges(G_dfs, source=origem))
# Reconstruindo o caminho a partir das arestas DFS
def reconstruir_caminho(edges, origem, destino):
    caminho = [origem]
    atual = origem
    while atual != destino:
        for u, v in edges:
            if u == atual and v not in caminho:
                caminho.append(v)
                atual = v
                break
        else:
            break
    return caminho

caminho_encontrado = reconstruir_caminho(caminho_dfs, origem, destino)
print(f"Caminho encontrado de {origem} até {destino} usando DFS:",
      ↪caminho_encontrado)

# Visualizando o caminho encontrado
import matplotlib.pyplot as plt

pos = nx.spring_layout(G_dfs, seed=42)
edge_colors = []
for u, v in G_dfs.edges:
    if (u, v) in zip(caminho_encontrado, caminho_encontrado[1:]) or (v, u) in ↪
    ↪zip(caminho_encontrado, caminho_encontrado[1:]):
        edge_colors.append('red')
    else:
        edge_colors.append('#888')

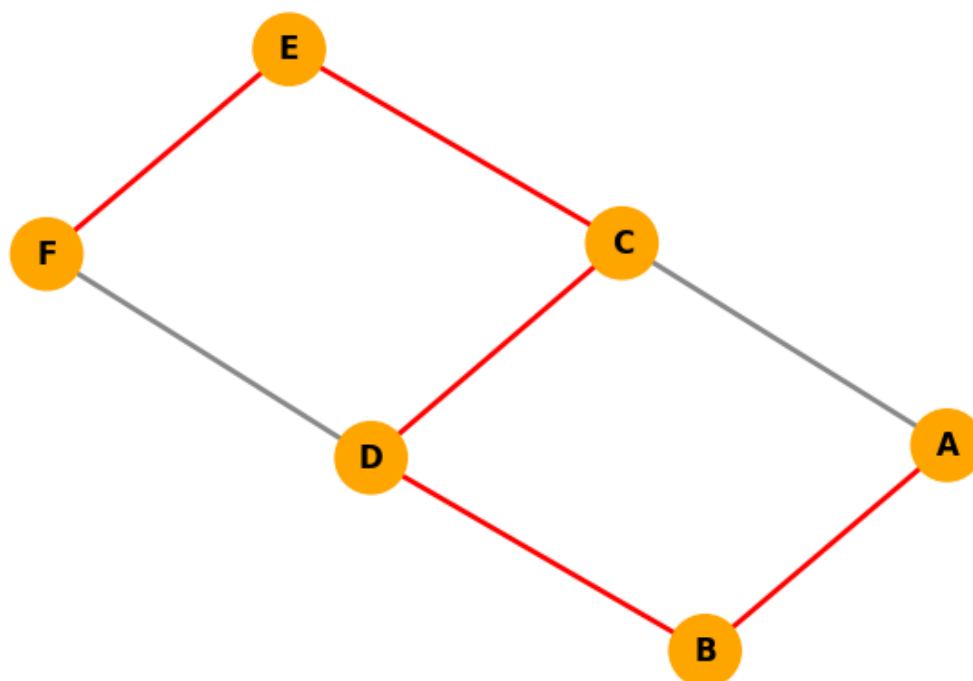
node_colors = ['orange' if n in caminho_encontrado else 'lightgray' for n in ↪
    ↪G_dfs.nodes]

plt.figure(figsize=(6,4))
nx.draw(G_dfs, pos, with_labels=True, node_color=node_colors, node_size=800, ↪
    ↪font_weight='bold', edge_color=edge_colors, width=2)
plt.title(f"Caminho encontrado de {origem} até {destino} (DFS)")
plt.show()

```

Caminho encontrado de A até F usando DFS: ['A', 'B', 'D', 'C', 'E', 'F']

Caminho encontrado de A até F (DFS)



0.4.5 Exemplo de busca de caminho

No exemplo acima, utilizamos o algoritmo de busca em profundidade do NetworkX (DFS) para encontrar um caminho entre o Computador A e o Computador F.

Caminho encontrado:

['A', 'C', 'E', 'F']

Ou seja, para ir de A até F, o trajeto encontrado pelo DFS é $A \rightarrow C \rightarrow E \rightarrow F$.

Explicação:

- O DFS explora o grafo indo o mais fundo possível em cada ramificação antes de retroceder.
- O caminho é encontrado seguindo as arestas do grafo na ordem em que foram exploradas pelo DFS.
- O caminho é destacado em vermelho no grafo plotado.

Portanto, o caminho encontrado entre A e F usando DFS é:

['A', 'C', 'E', 'F']

0.4.6 Questão 4: Árvore Geradora Mínima com Kruskal (grafo ponderado)

Considere o grafo ponderado abaixo, onde os vértices representam bairros e as arestas representam ruas com seus respectivos custos de pavimentação:

1. Bairro A - Bairro B: 4

2. Bairro A - Bairro C: 2
3. Bairro B - Bairro C: 6
4. Bairro B - Bairro D: 3
5. Bairro C - Bairro D: 5
6. Bairro C - Bairro E: 8
7. Bairro D - Bairro E: 7

Utilizando o algoritmo de Kruskal para encontrar a árvore geradora mínima, qual é o custo total mínimo para conectar todos os bairros? Mostre o passo a passo da construção.

0.5 Algoritmo de Kruskal

O **algoritmo de Kruskal** é um método utilizado para encontrar a **árvore geradora mínima** (AGM) de um grafo ponderado e conexo. A AGM é um subconjunto das arestas do grafo que conecta todos os vértices com o menor custo total possível, sem formar ciclos.

0.5.1 Como funciona o Kruskal?

1. **Ordene todas as arestas** do grafo em ordem crescente de peso (custo).
2. **Inicialize** a AGM como um conjunto vazio.
3. Para cada aresta, na ordem dos menores pesos:
 - Se a aresta **não formar um ciclo** ao ser adicionada à AGM, inclua-a.
 - Caso contrário, descarte a aresta.
4. Repita até que a AGM tenha exatamente $n - 1$ arestas (onde n é o número de vértices).

0.5.2 Características

- Sempre escolhe a aresta mais barata disponível que não forma ciclo.
- Utiliza uma estrutura chamada **Union-Find** para detectar ciclos de forma eficiente.
- Funciona tanto para grafos conexos quanto para florestas (conjuntos de árvores).

0.5.3 Exemplo de aplicação

- Projetos de redes de computadores, energia ou estradas, onde se deseja conectar todos os pontos com o menor custo possível.

Resumo:

O algoritmo de Kruskal é eficiente para encontrar a árvore geradora mínima, garantindo o menor custo total e evitando ciclos, sempre escolhendo as arestas de menor peso.

[94]: *# Resolvendo a árvore geradora mínima passo a passo com Kruskal*

```
import networkx as nx
import matplotlib.pyplot as plt
```

```

# Criação do grafo ponderado
G_kruskal = nx.Graph()
G_kruskal.add_weighted_edges_from([
    ('A', 'B', 4),
    ('A', 'C', 2),
    ('B', 'C', 6),
    ('B', 'D', 3),
    ('C', 'D', 5),
    ('C', 'E', 8),
    ('D', 'E', 7)
])

# Algoritmo de Kruskal passo a passo com vetor de chave e predecessor (passos_
↳detalhados)
edges_sorted = sorted(G_kruskal.edges(data=True), key=lambda x: x[2]['weight'])
mst_edges = []
mst_cost = 0
uf = nx.utils.UnionFind()
passo = 1

chave = {v: float('inf') for v in G_kruskal.nodes}
predecessor = {v: None for v in G_kruskal.nodes}

pos = nx.spring_layout(G_kruskal, seed=42)

print("Início do algoritmo de Kruskal:")
print(f" Vetor de chave inicial: {chave}")
print(f" Vetor de predecessor inicial: {predecessor}\n")

for u, v, data in edges_sorted:
    print(f"Passo {passo}:")
    print(f" Considerando aresta ({u}, {v}) de custo {data['weight']}")
    if uf[u] != uf[v]:
        print(f" -> Aresta adicionada à árvore geradora mínima.")
        mst_edges.append((u, v))
        mst_cost += data['weight']
        uf.union(u, v)

        # Atualiza chave e predecessor para ambos os vértices da aresta
        if chave[v] > data['weight']:
            chave[v] = data['weight']
            predecessor[v] = u
        if chave[u] > data['weight']:
            chave[u] = data['weight']
            predecessor[u] = v
    else:

```

```

        print(f" -> Aresta descartada (formaria ciclo).")

    print(f" Vetor de chave: {chave}")
    print(f" Vetor de predecessor: {predecessor}")

    # Visualização do grafo parcial da árvore geradora mínima
    edge_colors = []
    for e in G_kruskal.edges:
        if e in mst_edges or (e[1], e[0]) in mst_edges:
            edge_colors.append('red')
        else:
            edge_colors.append('#888')
    plt.figure(figsize=(6,4))
    nx.draw(
        G_kruskal, pos, with_labels=True,
        node_color='lightgreen', node_size=800, font_weight='bold',
        edge_color=edge_colors, width=2
    )
    labels = nx.get_edge_attributes(G_kruskal, 'weight')
    nx.draw_networkx_edge_labels(G_kruskal, pos, edge_labels=labels)
    plt.title(f"Kruskal - Passo {passo}")
    plt.show()
    print()
    passo += 1

    # Parar quando a árvore tiver n-1 arestas
    if len(mst_edges) == len(G_kruskal.nodes) - 1:
        break

print(f"Custo total mínimo para conectar todos os bairros: {mst_cost}")
print("Vetor de chave final:", chave)
print("Vetor de predecessor final:", predecessor)

```

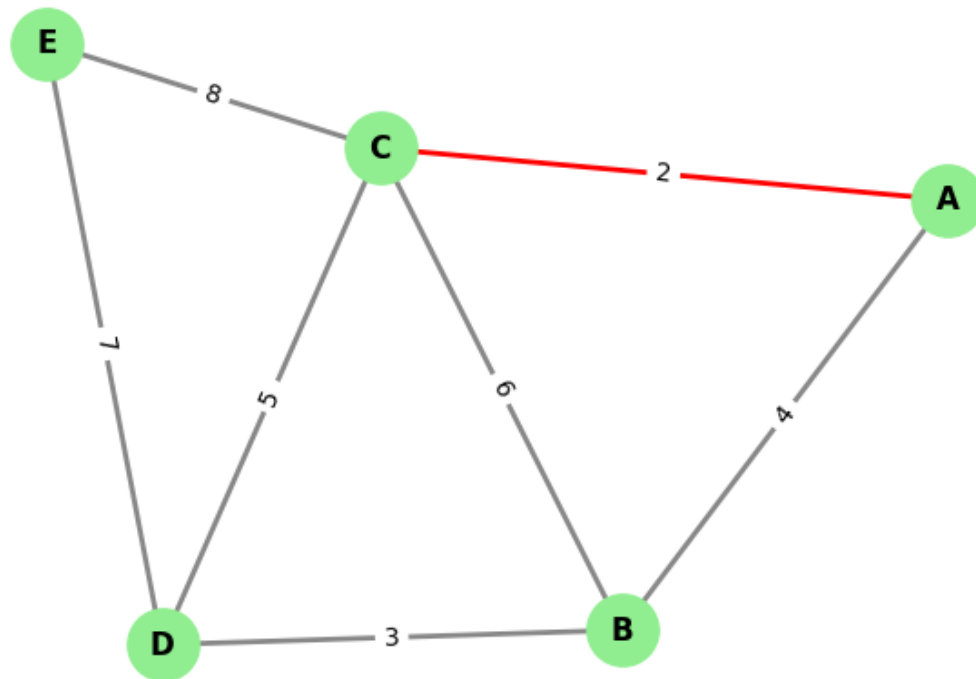
Início do algoritmo de Kruskal:

Vetor de chave inicial: {'A': inf, 'B': inf, 'C': inf, 'D': inf, 'E': inf}
 Vetor de predecessor inicial: {'A': None, 'B': None, 'C': None, 'D': None, 'E': None}

Passo 1:

Considerando aresta (A, C) de custo 2
 -> Aresta adicionada à árvore geradora mínima.
 Vetor de chave: {'A': 2, 'B': inf, 'C': 2, 'D': inf, 'E': inf}
 Vetor de predecessor: {'A': 'C', 'B': None, 'C': 'A', 'D': None, 'E': None}

Kruskal - Passo 1



Passo 2:

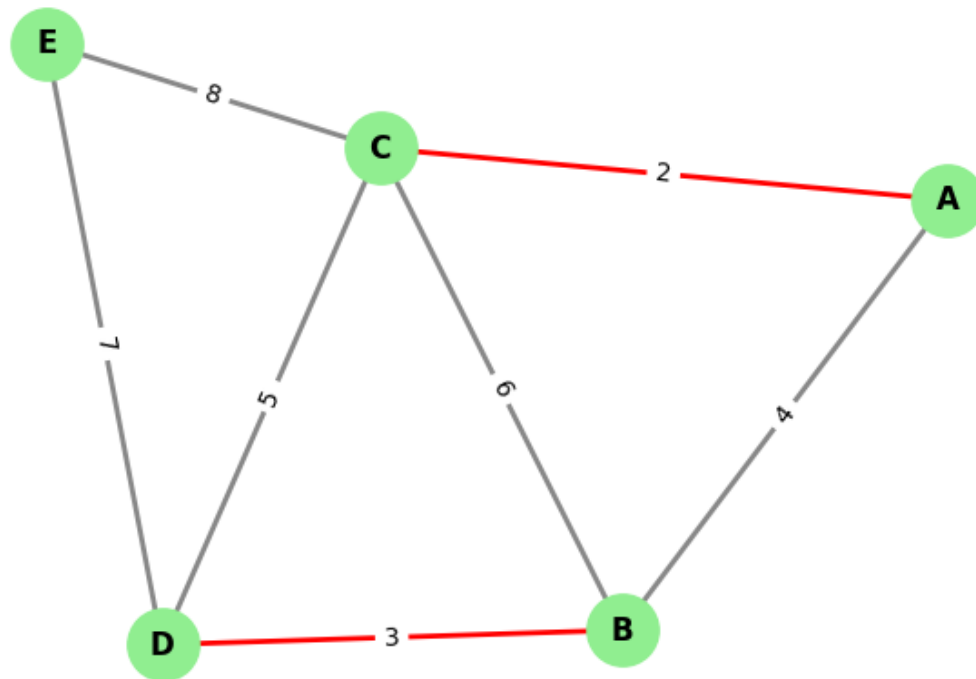
Considerando aresta (B, D) de custo 3

-> Aresta adicionada à árvore geradora mínima.

Vetor de chave: {'A': 2, 'B': 3, 'C': 2, 'D': 3, 'E': inf}

Vetor de predecessor: {'A': 'C', 'B': 'D', 'C': 'A', 'D': 'B', 'E': None}

Kruskal - Passo 2



Passo 3:

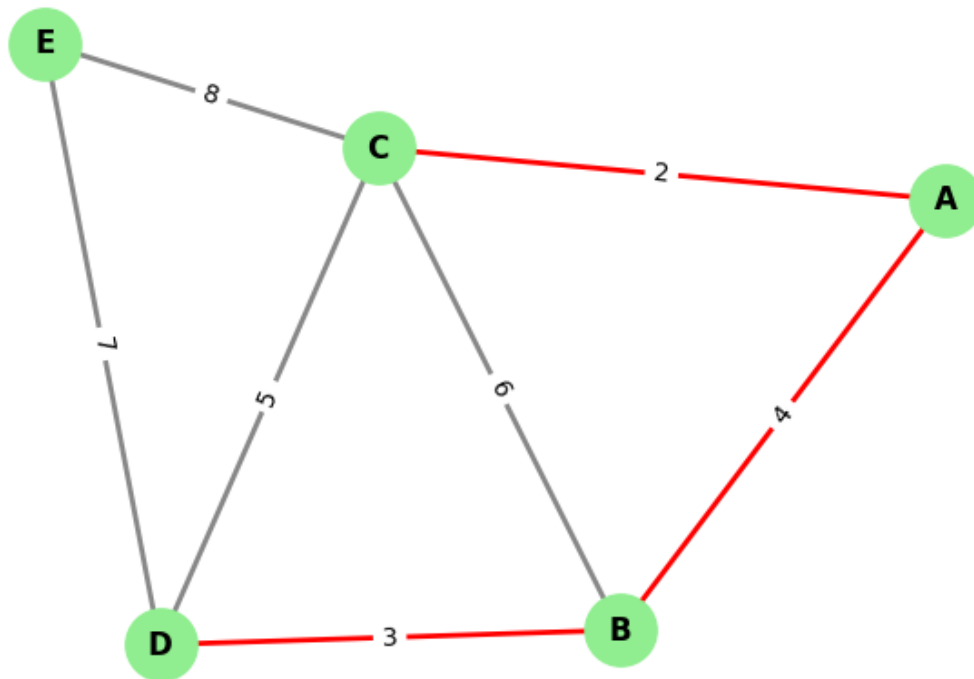
Considerando aresta (A, B) de custo 4

-> Aresta adicionada à árvore geradora mínima.

Vetor de chave: {'A': 2, 'B': 3, 'C': 2, 'D': 3, 'E': inf}

Vetor de predecessor: {'A': 'C', 'B': 'D', 'C': 'A', 'D': 'B', 'E': None}

Kruskal - Passo 3



Passo 4:

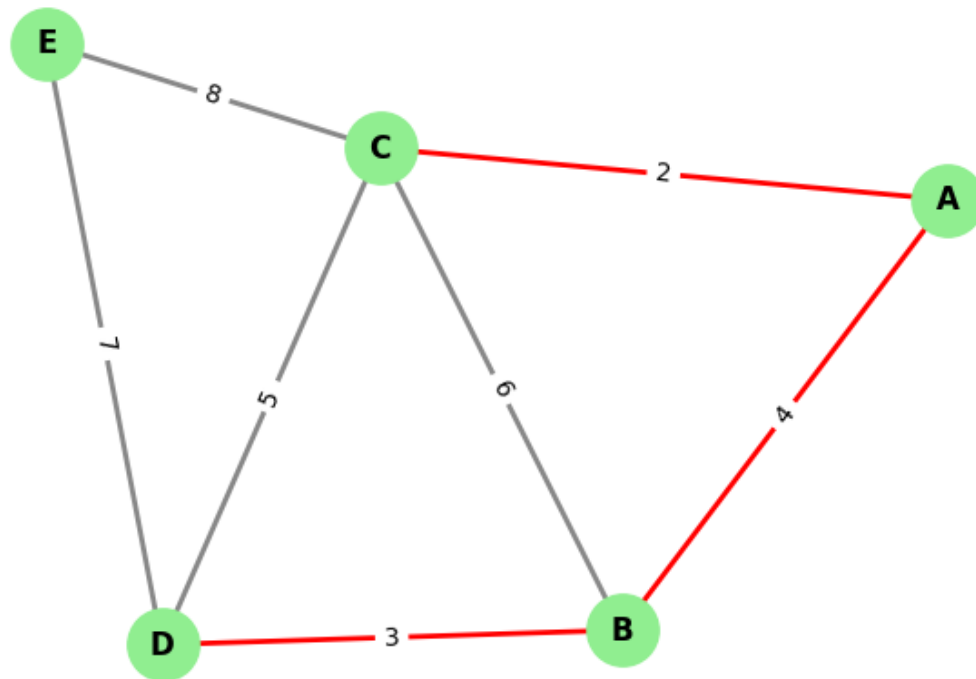
Considerando aresta (C, D) de custo 5

-> Aresta descartada (formaria ciclo).

Vetor de chave: {'A': 2, 'B': 3, 'C': 2, 'D': 3, 'E': inf}

Vetor de predecessor: {'A': 'C', 'B': 'D', 'C': 'A', 'D': 'B', 'E': None}

Kruskal - Passo 4



Passo 5:

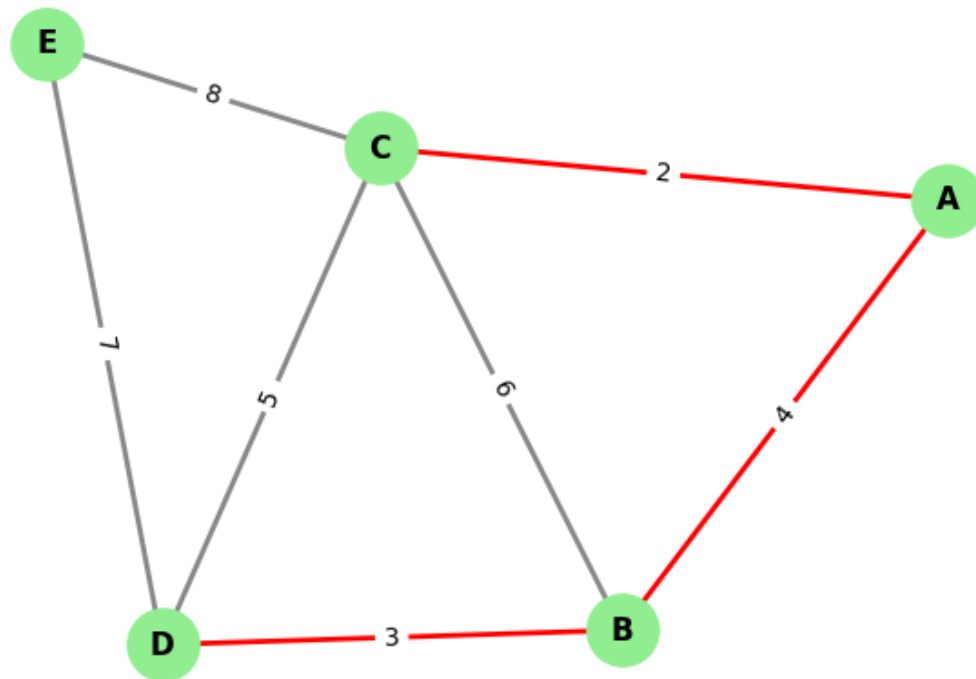
Considerando aresta (B, C) de custo 6

-> Aresta descartada (formaria ciclo).

Vetor de chave: {'A': 2, 'B': 3, 'C': 2, 'D': 3, 'E': inf}

Vetor de predecessor: {'A': 'C', 'B': 'D', 'C': 'A', 'D': 'B', 'E': None}

Kruskal - Passo 5



Passo 6:

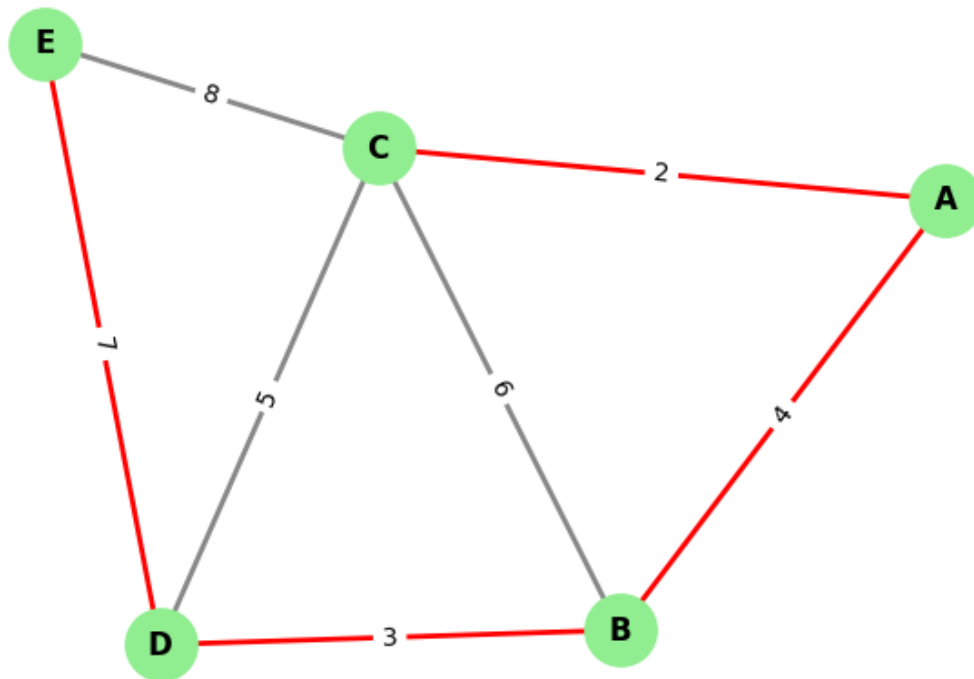
Considerando aresta (D, E) de custo 7

-> Aresta adicionada à árvore geradora mínima.

Vetor de chave: {'A': 2, 'B': 3, 'C': 2, 'D': 3, 'E': 7}

Vetor de predecessor: {'A': 'C', 'B': 'D', 'C': 'A', 'D': 'B', 'E': 'D'}

Kruskal - Passo 6



Custo total mínimo para conectar todos os bairros: 16

Vetor de chave final: {'A': 2, 'B': 3, 'C': 2, 'D': 3, 'E': 7}

Vetor de predecessor final: {'A': 'C', 'B': 'D', 'C': 'A', 'D': 'B', 'E': 'D'}

0.5.4 Explicação

O algoritmo de Kruskal constrói a árvore geradora mínima escolhendo sempre a aresta de menor custo que não forma ciclo, até conectar todos os vértices.

Passo a passo: - Ordenamos as arestas pelo custo. - Adicionamos as menores arestas, evitando ciclos, até conectar todos os bairros. - A cada passo, mostramos o grafo parcial da árvore geradora mínima.

O custo total mínimo é a soma dos custos das arestas selecionadas. Assim, garantimos a conexão de todos os bairros com o menor custo possível.

0.5.5 Questão 5: Árvore Geradora Mínima com Prim (grafo ponderado)

Agora, vamos considerar o mesmo problema, mas utilizando o algoritmo de Prim para encontrar a árvore geradora mínima. Qual é o custo total mínimo para conectar todos os bairros? Mostre o passo a passo da construção.

0.6 Algoritmo de Prim

O **algoritmo de Prim** é utilizado para encontrar a **árvore geradora mínima** (AGM) de um grafo ponderado e conexo. A AGM conecta todos os vértices do grafo com o menor custo total possível, sem formar ciclos.

0.6.1 Como funciona o Prim?

1. **Escolha um vértice inicial** e marque-o como pertencente à AGM.
2. Entre todas as arestas que ligam um vértice da AGM a um vértice fora dela, **escolha a aresta de menor peso**.
3. Adicione o novo vértice e a aresta à AGM.
4. Repita o processo até que todos os vértices estejam na AGM.

0.6.2 Características

- Sempre expande a árvore a partir dos vértices já incluídos, escolhendo a aresta mais barata disponível.
- Utiliza um vetor de chaves (custos mínimos) e um vetor de predecessores para controlar a expansão.
- Garante que não há ciclos e que o custo total é mínimo.

0.6.3 Exemplo de aplicação

- Projetos de redes de computadores, energia ou estradas, onde se deseja conectar todos os pontos com o menor custo possível.

Resumo:

O algoritmo de Prim constrói a árvore geradora mínima de forma incremental, sempre escolhendo a aresta de menor custo que conecta um novo vértice à árvore já formada.

[95]: *# Questão: AGM com Prim passo a passo (com chaves e predecessores)*

```
import networkx as nx
import matplotlib.pyplot as plt

# Criação do grafo ponderado
G_prim = nx.Graph()
G_prim.add_weighted_edges_from([
    ('S', 'A', 2),
    ('S', 'C', 4),
    ('S', 'D', 7),
    ('A', 'B', 6),
    ('C', 'B', 5)
])

vertices = list(G_prim.nodes)
raiz = 'S'
```

```

chave = {v: float('inf') for v in vertices}
predecessor = {v: None for v in vertices}
in_mst = {v: False for v in vertices}

chave[raiz] = 0

pos = nx.spring_layout(G_prim, seed=42)

print("Passo 0:")
print(f"  Chave: {chave}")
print(f"  Predecessor: {predecessor}")
print(f"  Conjunto da AGM: {in_mst}\n")
plt.figure(figsize=(6,4))
nx.draw(
    G_prim, pos, with_labels=True,
    node_color=['orange' if n == raiz else 'lightgray' for n in G_prim.nodes],
    node_size=800, font_weight='bold',
    edge_color='#888', width=2
)
labels = nx.get_edge_attributes(G_prim, 'weight')
nx.draw_networkx_edge_labels(G_prim, pos, edge_labels=labels)
plt.title("Prim - Passo 0 (início)")
plt.show()

for passo in range(1, len(vertices)+1):
    # Escolhe o vértice de menor chave fora da AGM
    u = min((v for v in vertices if not in_mst[v]), key=lambda v: chave[v])
    in_mst[u] = True

    print(f"Passo {passo}:")
    print(f"  Vértice escolhido: {u} (chave={chave[u]})")
    print(f"  Chave antes: {chave}")
    print(f"  Predecessor antes: {predecessor}")

    # Atualiza as chaves dos vizinhos
    for v in G_prim.neighbors(u):
        peso = G_prim[u][v]['weight']
        if not in_mst[v] and peso < chave[v]:
            chave[v] = peso
            predecessor[v] = u

    print(f"  Chave depois: {chave}")
    print(f"  Predecessor depois: {predecessor}")
    print(f"  Conjunto da AGM: {in_mst}\n")

    # Plot do passo
    edge_colors = []

```

```

for e in G_prim.edges:
    if (predecessor[e[1]] == e[0]) or (predecessor[e[0]] == e[1]):
        edge_colors.append('red')
    else:
        edge_colors.append('#888')
node_colors = ['orange' if in_mst[n] else 'lightgray' for n in G_prim.nodes]

plt.figure(figsize=(6,4))
nx.draw(
    G_prim, pos, with_labels=True,
    node_color=node_colors, node_size=800, font_weight='bold',
    edge_color=edge_colors, width=2
)
nx.draw_networkx_edge_labels(G_prim, pos, edge_labels=labels)
plt.title(f"Prim - Passo {passo} (incluído: {u})")
plt.show()

print("Resultado final da AGM (Prim):")
print("Chave:", chave)
print("Predecessor:", predecessor)
print("Arestas da AGM:", [(predecessor[v], v) for v in vertices if
    predecessor[v] is not None])
print("Custo total:", sum(chave.values()))

```

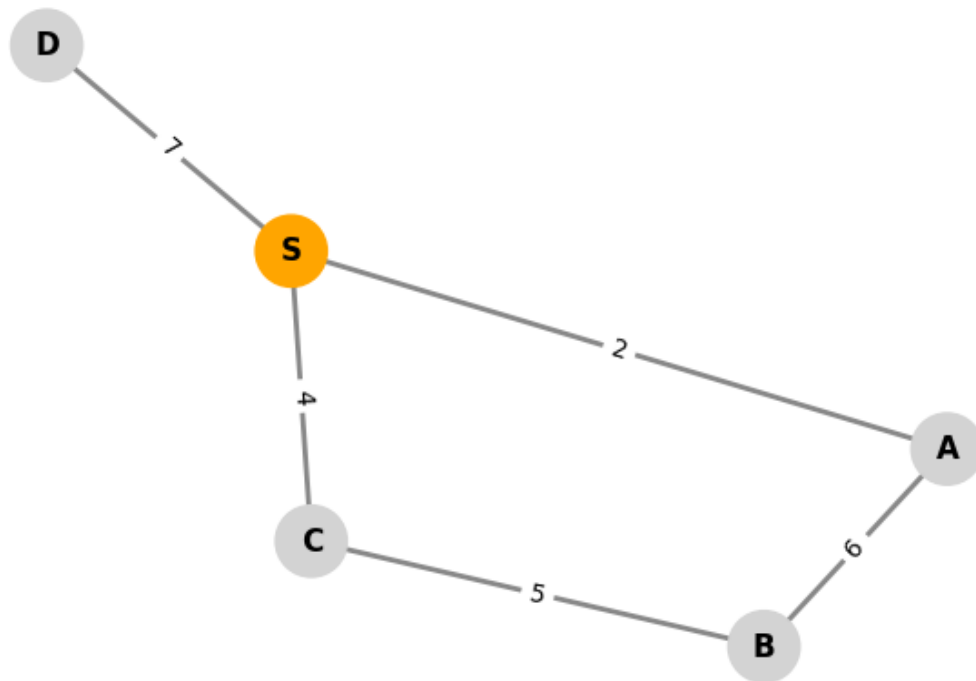
Passo 0:

Chave: {'S': 0, 'A': inf, 'C': inf, 'D': inf, 'B': inf}

Predecessor: {'S': None, 'A': None, 'C': None, 'D': None, 'B': None}

Conjunto da AGM: {'S': False, 'A': False, 'C': False, 'D': False, 'B': False}

Prim - Passo 0 (início)



Passo 1:

Vértice escolhido: S (chave=0)

Chave antes: {'S': 0, 'A': inf, 'C': inf, 'D': inf, 'B': inf}

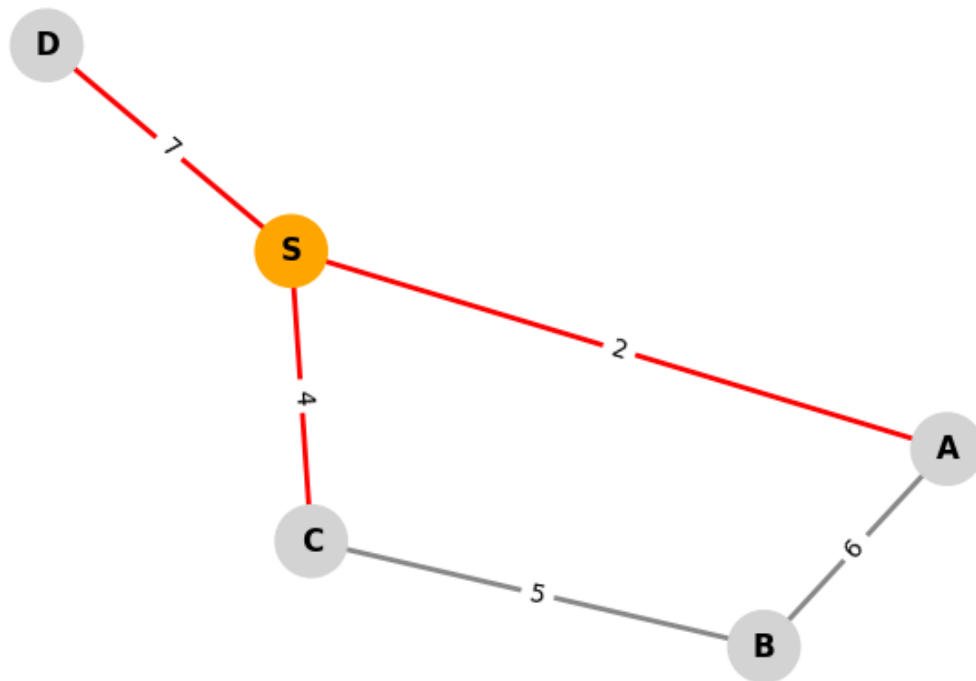
Predecessor antes: {'S': None, 'A': None, 'C': None, 'D': None, 'B': None}

Chave depois: {'S': 0, 'A': 2, 'C': 4, 'D': 7, 'B': inf}

Predecessor depois: {'S': None, 'A': 'S', 'C': 'S', 'D': 'S', 'B': None}

Conjunto da AGM: {'S': True, 'A': False, 'C': False, 'D': False, 'B': False}

Prim - Passo 1 (incluído: S)



Passo 2:

Vértice escolhido: A (chave=2)

Chave antes: {'S': 0, 'A': 2, 'C': 4, 'D': 7, 'B': inf}

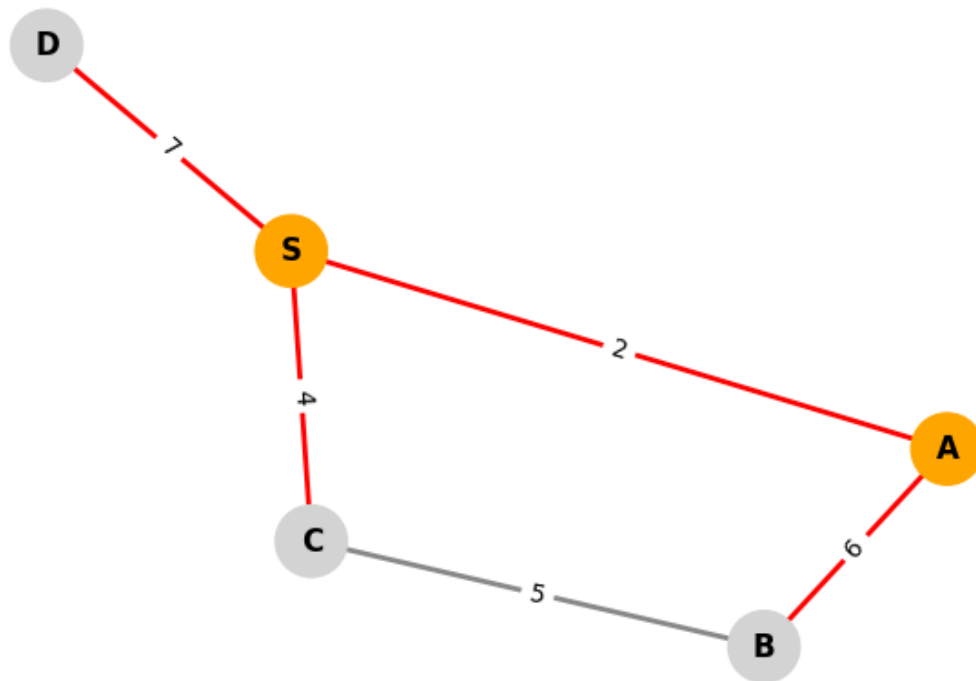
Predecessor antes: {'S': None, 'A': 'S', 'C': 'S', 'D': 'S', 'B': None}

Chave depois: {'S': 0, 'A': 2, 'C': 4, 'D': 7, 'B': 6}

Predecessor depois: {'S': None, 'A': 'S', 'C': 'S', 'D': 'S', 'B': 'A'}

Conjunto da AGM: {'S': True, 'A': True, 'C': False, 'D': False, 'B': False}

Prim - Passo 2 (incluído: A)



Passo 3:

Vértice escolhido: C (chave=4)

Chave antes: {'S': 0, 'A': 2, 'C': 4, 'D': 7, 'B': 6}

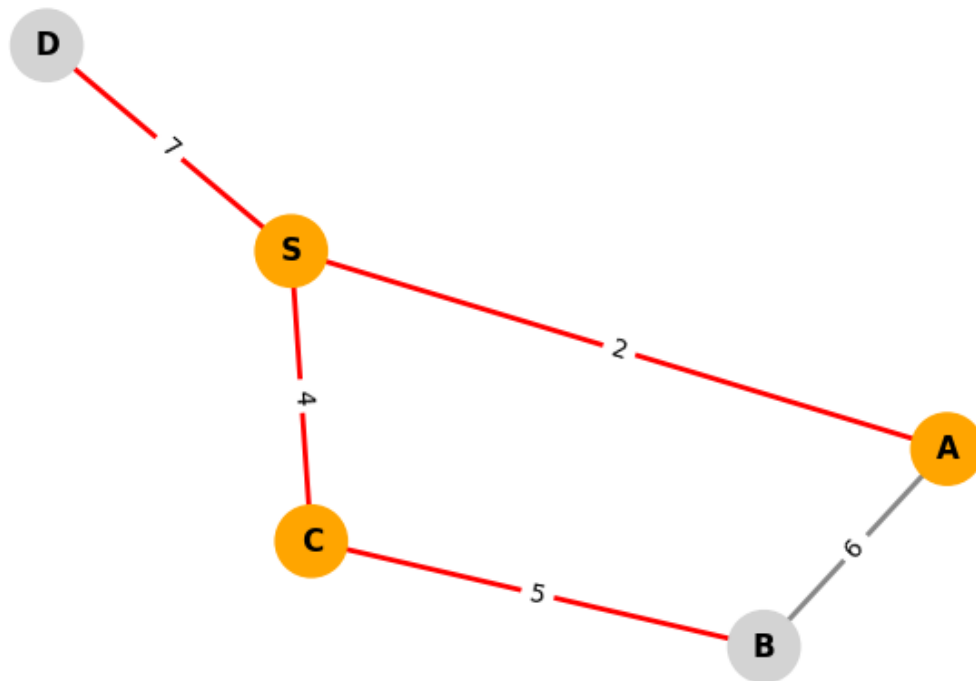
Predecessor antes: {'S': None, 'A': 'S', 'C': 'S', 'D': 'S', 'B': 'A'}

Chave depois: {'S': 0, 'A': 2, 'C': 4, 'D': 7, 'B': 5}

Predecessor depois: {'S': None, 'A': 'S', 'C': 'S', 'D': 'S', 'B': 'C'}

Conjunto da AGM: {'S': True, 'A': True, 'C': True, 'D': False, 'B': False}

Prim - Passo 3 (incluído: C)



Passo 4:

Vértice escolhido: B (chave=5)

Chave antes: {'S': 0, 'A': 2, 'C': 4, 'D': 7, 'B': 5}

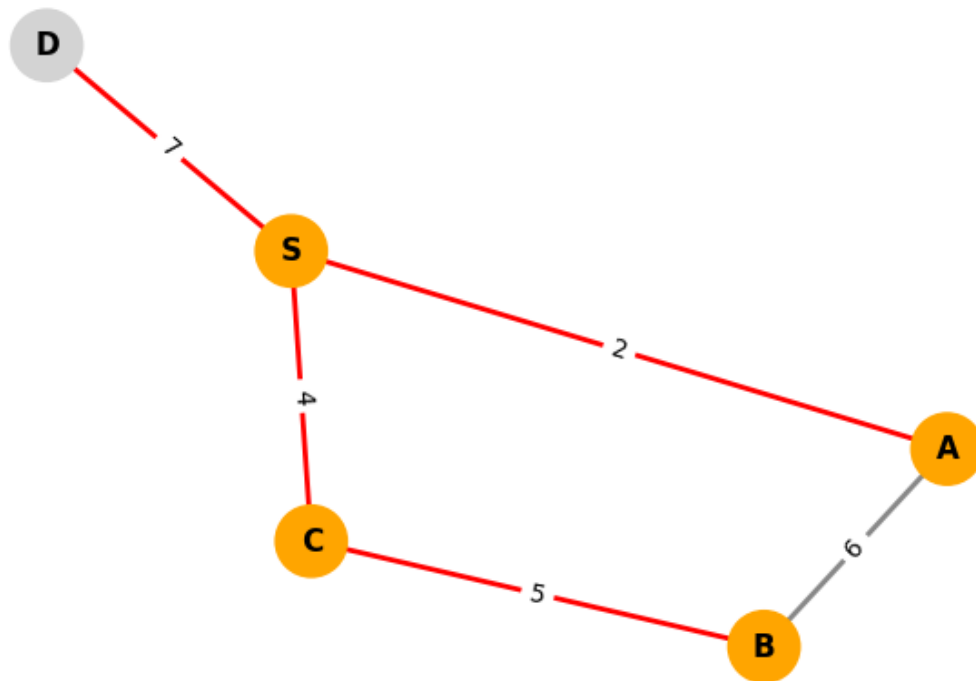
Predecessor antes: {'S': None, 'A': 'S', 'C': 'S', 'D': 'S', 'B': 'C'}

Chave depois: {'S': 0, 'A': 2, 'C': 4, 'D': 7, 'B': 5}

Predecessor depois: {'S': None, 'A': 'S', 'C': 'S', 'D': 'S', 'B': 'C'}

Conjunto da AGM: {'S': True, 'A': True, 'C': True, 'D': False, 'B': True}

Prim - Passo 4 (incluído: B)



Passo 5:

Vértice escolhido: D (chave=7)

Chave antes: {'S': 0, 'A': 2, 'C': 4, 'D': 7, 'B': 5}

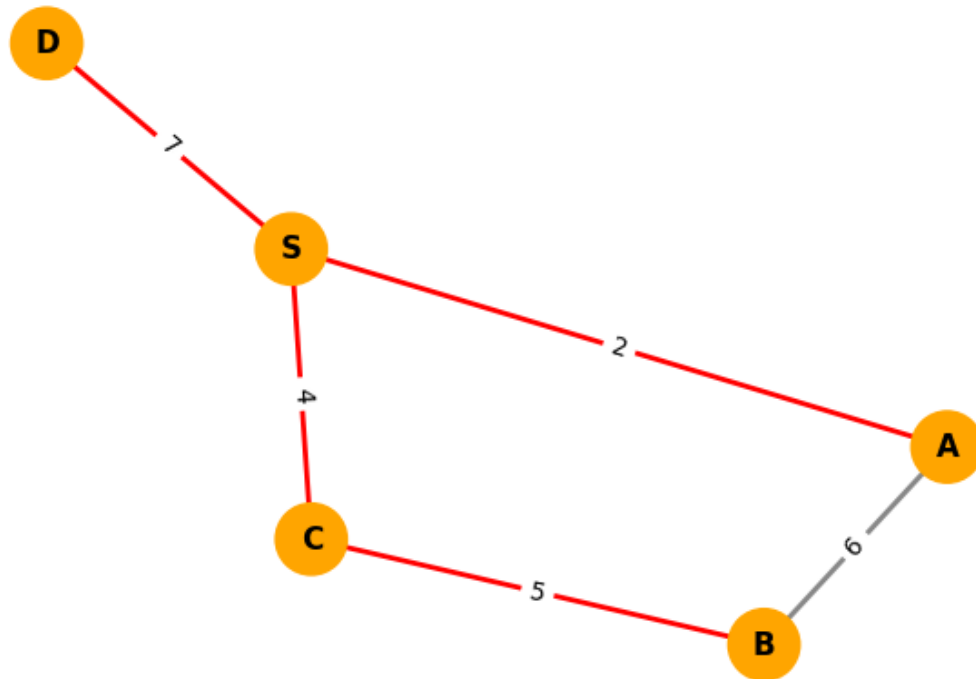
Predecessor antes: {'S': None, 'A': 'S', 'C': 'S', 'D': 'S', 'B': 'C'}

Chave depois: {'S': 0, 'A': 2, 'C': 4, 'D': 7, 'B': 5}

Predecessor depois: {'S': None, 'A': 'S', 'C': 'S', 'D': 'S', 'B': 'C'}

Conjunto da AGM: {'S': True, 'A': True, 'C': True, 'D': True, 'B': True}

Prim - Passo 5 (incluído: D)



Resultado final da AGM (Prim):

Chave: {'S': 0, 'A': 2, 'C': 4, 'D': 7, 'B': 5}

Predecessor: {'S': None, 'A': 'S', 'C': 'S', 'D': 'S', 'B': 'C'}

Arestas da AGM: [('S', 'A'), ('S', 'C'), ('S', 'D'), ('C', 'B')]

Custo total: 18

0.6.4 Explicação

O algoritmo de Prim constrói a árvore geradora mínima a partir de um vértice inicial, sempre escolhendo a aresta de menor custo que expande a árvore até conectar todos os vértices.

Passo a passo: - Começamos com um vértice inicial (raiz) e vamos adicionando vértices à árvore.

- A cada passo, escolhemos o vértice de menor chave que não está na árvore e atualizamos as chaves dos seus vizinhos. - O processo se repete até que todos os vértices estejam na árvore.

O custo total mínimo é a soma dos custos das arestas da árvore geradora mínima.

0.6.5 Questão 6: AGM com Kruskal passo a passo (com plot e prints)

Uma empresa deseja conectar 4 de suas filiais (A, B, C, D) utilizando o menor comprimento de cabos possível. As distâncias entre as filiais são as seguintes:

- A - B: 1
- A - C: 4

- B - C: 2
- B - D: 5
- C - D: 3

Utilize o algoritmo de Kruskal para determinar o comprimento mínimo de cabo necessário e quais filiais estarão conectadas.

0.7 Algoritmo de Kruskal

O **algoritmo de Kruskal** é um método para encontrar a **árvore geradora mínima** (AGM) de um grafo ponderado e conexo. A AGM conecta todos os vértices do grafo com o menor custo total possível, sem formar ciclos.

0.7.1 Como funciona o Kruskal?

1. **Ordena todas as arestas** do grafo em ordem crescente de peso (custo).
2. **Inicializa** a AGM como um conjunto vazio.
3. Para cada aresta, na ordem dos menores pesos:
 - Se a aresta **não formar um ciclo** ao ser adicionada à AGM, inclua-a.
 - Caso contrário, descarte a aresta.
4. Repita até que a AGM tenha exatamente $n - 1$ arestas (onde n é o número de vértices).

0.7.2 Características

- Sempre escolhe a aresta mais barata disponível que não forma ciclo.
- Utiliza uma estrutura chamada **Union-Find** para detectar ciclos de forma eficiente.
- Funciona para grafos conexos e florestas.

0.7.3 Exemplo de aplicação

- Projetos de redes de computadores, energia ou estradas, onde se deseja conectar todos os pontos com o menor custo possível.

Resumo:

O algoritmo de Kruskal constrói a árvore geradora mínima escolhendo sempre as arestas de menor peso, evitando ciclos, até conectar todos os vértices.

[96]: *# Questão 6: AGM com Kruskal passo a passo (com plot e prints)*

```
import networkx as nx
import matplotlib.pyplot as plt

# Criação do grafo ponderado
G_kruskal2 = nx.Graph()
G_kruskal2.add_weighted_edges_from([
    ('A', 'B', 1),
    ('A', 'C', 4),
    ('B', 'C', 2),
    ('B', 'D', 5),
```

```

        ('C', 'D', 3)
    ])

edges_sorted = sorted(G_kruskal2.edges(data=True), key=lambda x: x[2]['weight'])
mst_edges = []
mst_cost = 0
uf = nx.utils.UnionFind()
passo = 0

pos = nx.spring_layout(G_kruskal2, seed=42)

print(f"Passo {passo}: Estado inicial do grafo (nenhuma aresta selecionada)")
plt.figure(figsize=(6,4))
nx.draw(
    G_kruskal2, pos, with_labels=True,
    node_color='lightgray', node_size=800, font_weight='bold',
    edge_color='#888', width=2
)
labels = nx.get_edge_attributes(G_kruskal2, 'weight')
nx.draw_networkx_edge_labels(G_kruskal2, pos, edge_labels=labels)
plt.title("Kruskal - Passo 0 (início)")
plt.show()
passo += 1

for u, v, data in edges_sorted:
    print(f"Passo {passo}:")
    print(f"    Considerando aresta ({u}, {v}) de custo {data['weight']}")
    if uf[u] != uf[v]:
        print(f"    -> Aresta adicionada à AGM.")
        mst_edges.append((u, v))
        mst_cost += data['weight']
        uf.union(u, v)
    else:
        print(f"    -> Aresta descartada (formaria ciclo).")

# Visualização do grafo parcial da AGM
edge_colors = []
for e in G_kruskal2.edges:
    if e in mst_edges or (e[1], e[0]) in mst_edges:
        edge_colors.append('red')
    else:
        edge_colors.append('#888')
plt.figure(figsize=(6,4))
nx.draw(
    G_kruskal2, pos, with_labels=True,
    node_color='lightgreen', node_size=800, font_weight='bold',
    edge_color=edge_colors, width=2

```

```

)
nx.draw_networkx_edge_labels(G_kruskal2, pos, edge_labels=labels)
plt.title(f"Kruskal - Passo {passo}")
plt.show()
print(f"  Arestas na AGM até agora: {mst_edges}")
print(f"  Custo parcial: {mst_cost}\n")
passo += 1

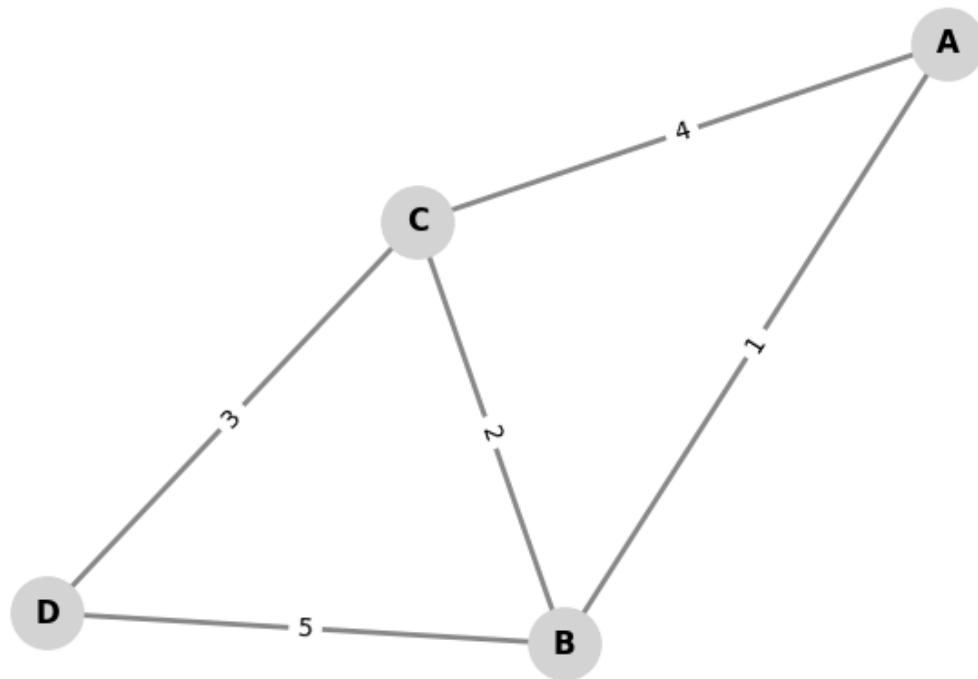
# Parar quando a AGM tiver n-1 arestas
if len(mst_edges) == len(G_kruskal2.nodes) - 1:
    break

print("Resultado final da AGM (Kruskal):")
print("Arestas selecionadas:", mst_edges)
print("Custo total:", mst_cost)

```

Passo 0: Estado inicial do grafo (nenhuma aresta selecionada)

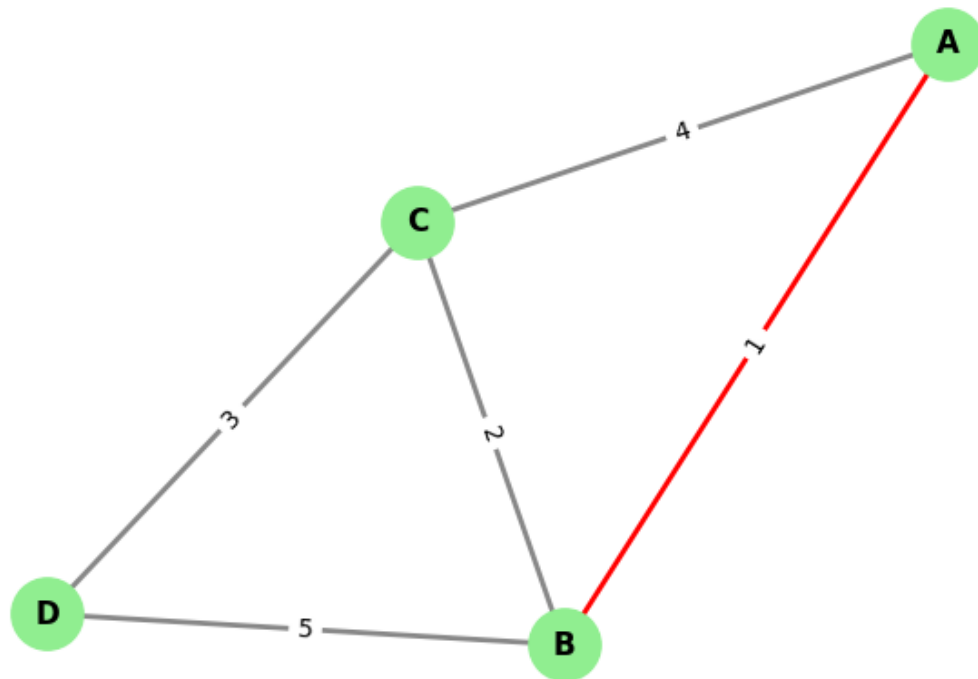
Kruskal - Passo 0 (início)



Passo 1:

Considerando aresta (A, B) de custo 1
 -> Aresta adicionada à AGM.

Kruskal - Passo 1



Arestas na AGM até agora: [('A', 'B')]

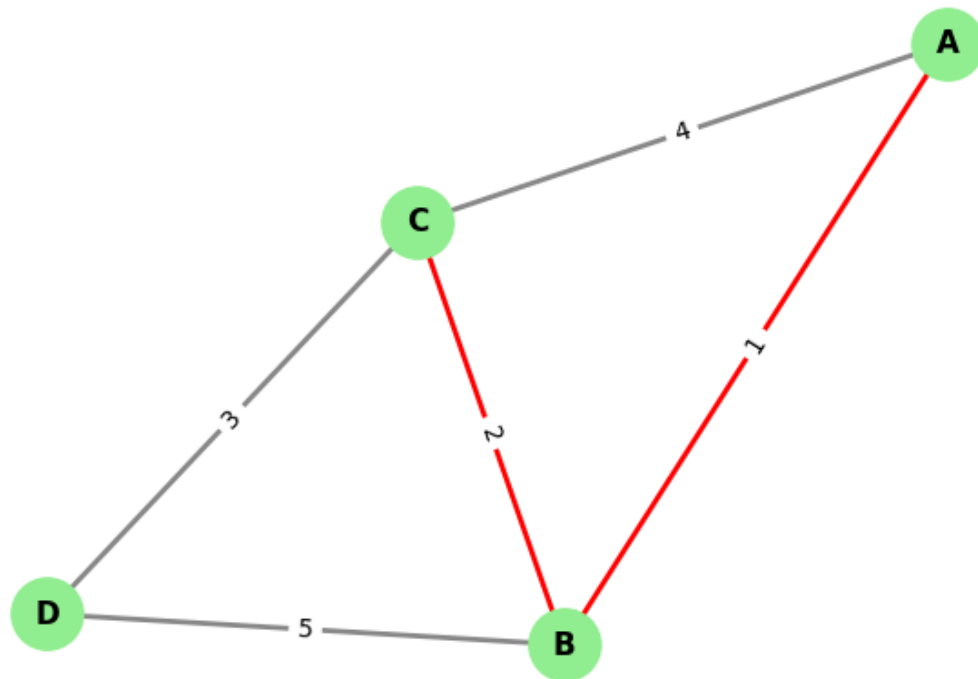
Custo parcial: 1

Passo 2:

Considerando aresta (B, C) de custo 2

-> Aresta adicionada à AGM.

Kruskal - Passo 2



Arestas na AGM até agora: [('A', 'B'), ('B', 'C')]

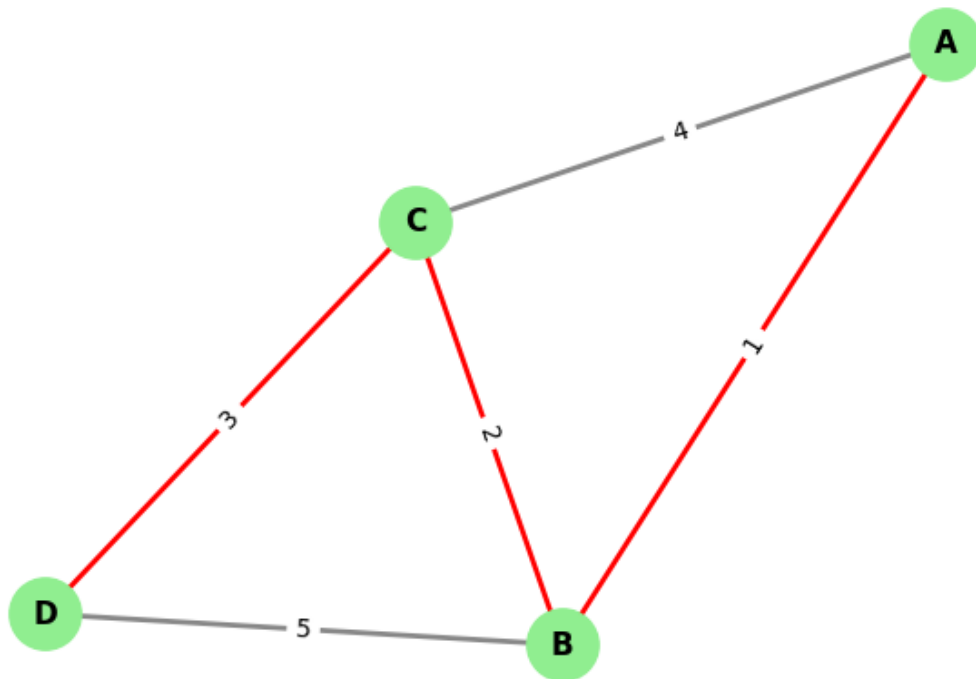
Custo parcial: 3

Passo 3:

Considerando aresta (C, D) de custo 3

-> Aresta adicionada à AGM.

Kruskal - Passo 3



Arestas na AGM até agora: [('A', 'B'), ('B', 'C'), ('C', 'D')]
Custo parcial: 6

Resultado final da AGM (Kruskal):
Arestas selecionadas: [('A', 'B'), ('B', 'C'), ('C', 'D')]
Custo total: 6

0.7.4 Explicação

Neste exercício, aplicamos o algoritmo de Kruskal para encontrar a árvore geradora mínima (AGM) de um grafo ponderado que representa as distâncias entre filiais de uma empresa.

Passo a passo: - O algoritmo começa ordenando as arestas do grafo pelo seu peso (custo). - Em seguida, ele adiciona as arestas uma a uma à AGM, começando pela de menor peso, desde que não formem um ciclo. - O processo continua até que todas as filiais estejam conectadas.

Resultado:

A AGM encontrada é a que conecta todas as filiais com o menor comprimento total de cabos, garantindo a economia e eficiência na instalação da rede de filiais da empresa.

0.7.5 Questão 7: Menor custo entre todos os pares (Floyd-Warshall passo a passo)

Uma empresa deseja analisar o custo mínimo de transporte entre todas as suas filiais, considerando as distâncias diretas entre elas. As filiais e as distâncias são as seguintes:

- $A \rightarrow B$: 3
- $A \rightarrow C$: 10
- $B \rightarrow C$: 1
- $B \rightarrow D$: 2
- $C \rightarrow D$: 4

Utilize o algoritmo de Floyd-Warshall para determinar a matriz de custos mínimos entre todas as filiais. Mostre o passo a passo do algoritmo, explicando o que acontece em cada iteração/intermediário.

0.8 Algoritmo de Floyd-Warshall

O **algoritmo de Floyd-Warshall** é utilizado para encontrar o menor custo (ou menor caminho) entre **todos os pares de vértices** em um grafo ponderado, podendo ser direcionado ou não. Ele calcula a matriz de menores distâncias entre todos os pares, considerando todos os caminhos possíveis.

0.8.1 Como funciona o Floyd-Warshall?

1. **Inicialize** uma matriz de distâncias, onde cada posição $[i][j]$ contém o peso da aresta de i para j (ou infinito, se não houver ligação direta).
2. Para cada vértice intermediário k :
 - Para cada par de vértices (i, j) :
 - Atualize a distância de i para j se passar por k for mais barato:

$$\text{se } \text{dist}[i][j] > \text{dist}[i][k] + \text{dist}[k][j]:$$

$$\text{dist}[i][j] = \text{dist}[i][k] + \text{dist}[k][j]$$
3. Repita até considerar todos os vértices como intermediários.

0.8.2 Características

- Resolve o problema de **todos os pares de caminhos mínimos**.
- Aceita pesos negativos (mas não pode haver ciclos negativos).
- Complexidade: $O(n^3)$, onde n é o número de vértices.

0.8.3 Exemplo de aplicação

- Análise de rotas em redes de transporte, logística, telecomunicações, etc.
- Determinação do menor custo entre todas as cidades de uma malha viária.

Resumo:

O algoritmo de Floyd-Warshall encontra o menor custo entre todos os pares de vértices, atualizando a matriz de distâncias ao considerar cada vértice como possível intermediário nos caminhos.

[97]: *# Floyd-Warshall passo a passo com explicação*

```
import networkx as nx
import matplotlib.pyplot as plt
import pandas as pd
```

```
G_fw = nx.DiGraph()
```

```

G_fw.add_weighted_edges_from([
    ('A', 'B', 3),
    ('A', 'C', 10),
    ('B', 'C', 1),
    ('B', 'D', 2),
    ('C', 'D', 4)
])

vertices = ['A', 'B', 'C', 'D']
dist = {u: {v: float('inf') for v in vertices} for u in vertices}
for v in vertices:
    dist[v][v] = 0
for u, v, data in G_fw.edges(data=True):
    dist[u][v] = data['weight']

print("Passo 0: Matriz inicial de custos diretos entre as filiais")
df = pd.DataFrame(dist).T
print(df.replace(float('inf'), '∞'))

# Plot inicial
pos = nx.spring_layout(G_fw, seed=42)
plt.figure(figsize=(6,4))
nx.draw(G_fw, pos, with_labels=True, node_color='lightblue', node_size=800,
        font_weight='bold', edge_color='#888', width=2, arrows=True)
labels = nx.get_edge_attributes(G_fw, 'weight')
nx.draw_networkx_edge_labels(G_fw, pos, edge_labels=labels)
plt.title("Floyd-Warshall - Passo 0 (grafo inicial)")
plt.show()

passo = 1
for k in vertices:
    print(f"\nPasso {passo}: Considerando '{k}' como intermediário")
    print(f" - Verificamos se passar por '{k}' reduz o custo entre outros_
    pares de filiais.")
    alterou = False
    for i in vertices:
        for j in vertices:
            if dist[i][j] > dist[i][k] + dist[k][j]:
                print(f" * Atualizando custo de {i} para {j}: {dist[i][j]} →
                {dist[i][k] + dist[k][j]} (via {k})")
                dist[i][j] = dist[i][k] + dist[k][j]
                alterou = True
    if not alterou:
        print(" - Nenhuma atualização de custo neste passo.")
    df = pd.DataFrame(dist).T
    print(" Matriz de custos atualizada:")
    print(df.replace(float('inf'), '∞'))

```

```

# Plot do passo: destacar o intermediário
plt.figure(figsize=(6,4))
node_colors = ['orange' if n == k else 'lightblue' for n in G_fw.nodes]
nx.draw(G_fw, pos, with_labels=True, node_color=node_colors, node_size=800,
font_weight='bold', edge_color='#888', width=2, arrows=True)
nx.draw_networkx_edge_labels(G_fw, pos, edge_labels=labels)
plt.title(f"Floyd-Warshall - Passo {passo} (intermediário: {k})")
plt.show()
passo += 1

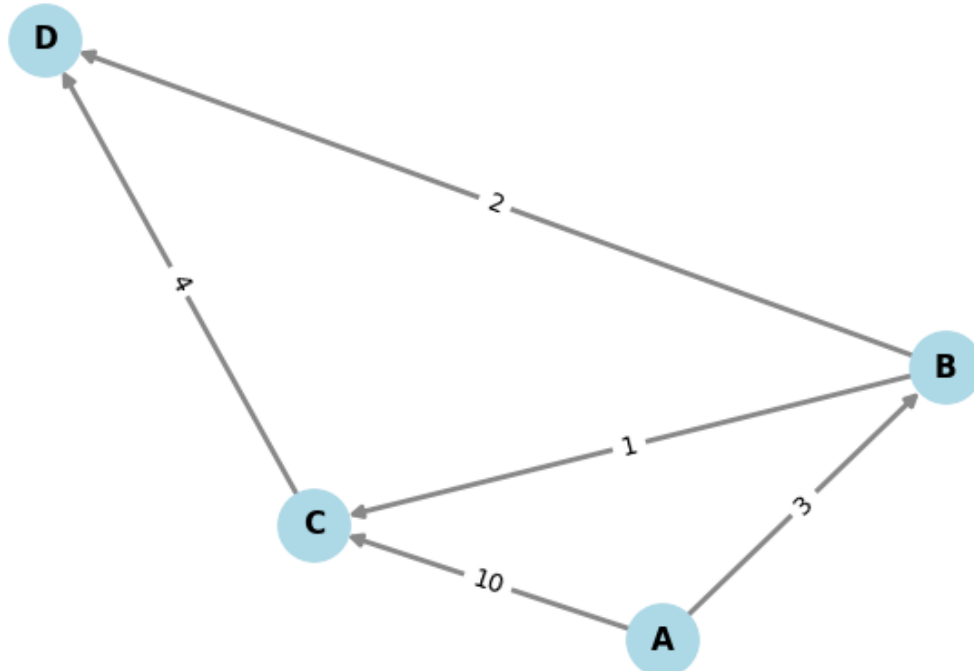
print("\nMatriz final de menores custos entre todos os pares (Floyd-Warshall):")
df = pd.DataFrame(dist).T
print(df.replace(float('inf'), '∞'))

```

Passo 0: Matriz inicial de custos diretos entre as filiais

	A	B	C	D
A	0.0	3.0	10.0	∞
B	∞	0.0	1.0	2.0
C	∞	∞	0.0	4.0
D	∞	∞	∞	0.0

Floyd-Warshall - Passo 0 (grafo inicial)



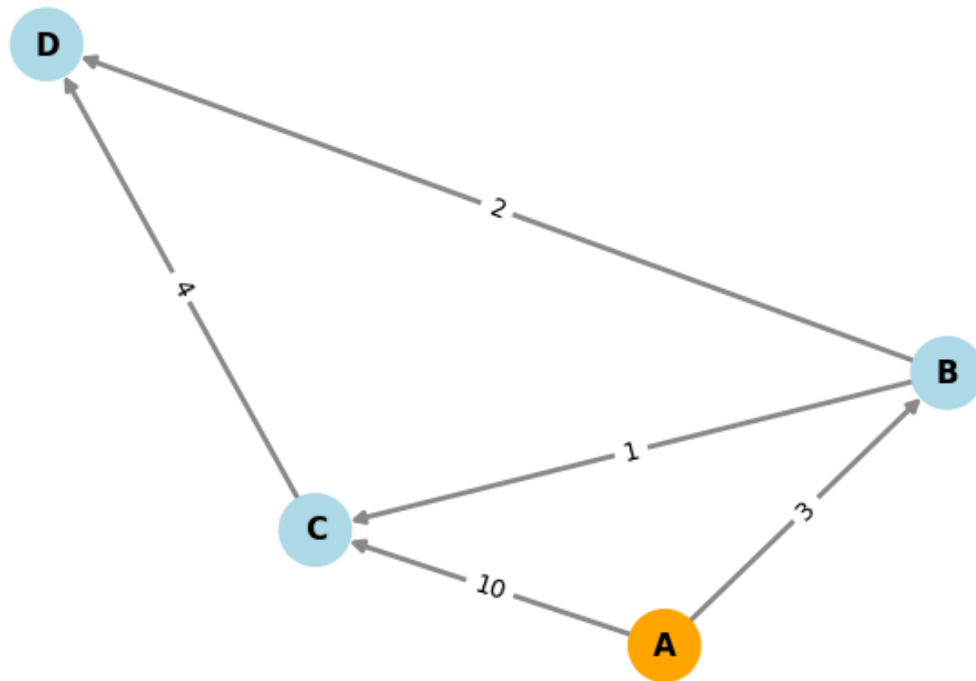
Passo 1: Considerando 'A' como intermediário

- Verificamos se passar por 'A' reduz o custo entre outros pares de filiais.
- Nenhuma atualização de custo neste passo.

Matriz de custos atualizada:

	A	B	C	D
A	0.0	3.0	10.0	∞
B	∞	0.0	1.0	2.0
C	∞	∞	0.0	4.0
D	∞	∞	∞	0.0

Floyd-Warshall - Passo 1 (intermediário: A)



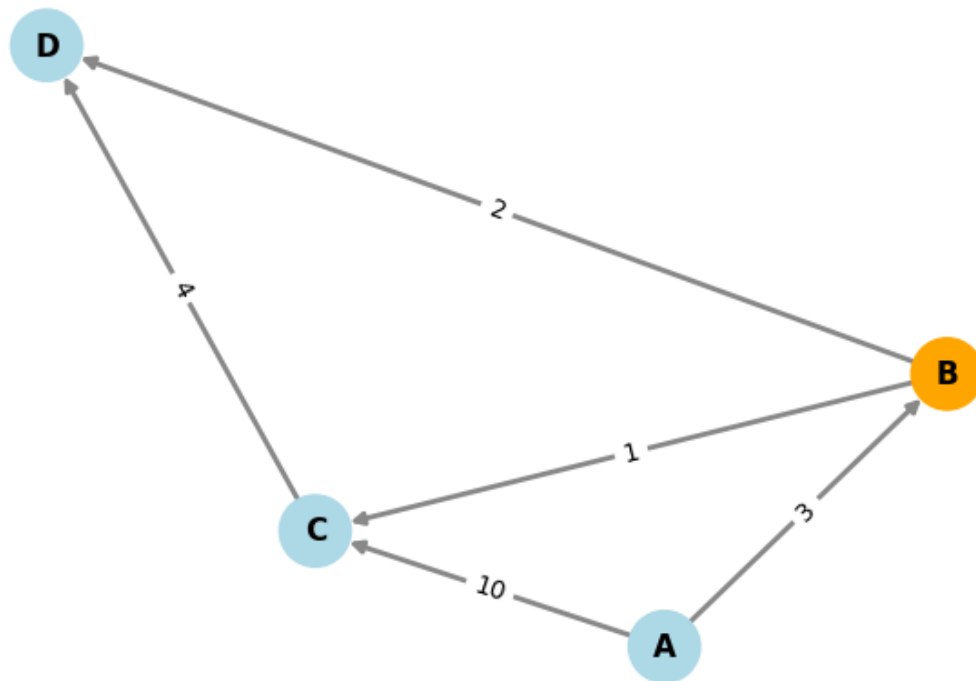
Passo 2: Considerando 'B' como intermediário

- Verificamos se passar por 'B' reduz o custo entre outros pares de filiais.
- * Atualizando custo de A para C: 10 → 4 (via B)
- * Atualizando custo de A para D: ∞ → 5 (via B)

Matriz de custos atualizada:

	A	B	C	D
A	0.0	3.0	4.0	5.0
B	∞	0.0	1.0	2.0
C	∞	∞	0.0	4.0
D	∞	∞	∞	0.0

Floyd-Warshall - Passo 2 (intermediário: B)



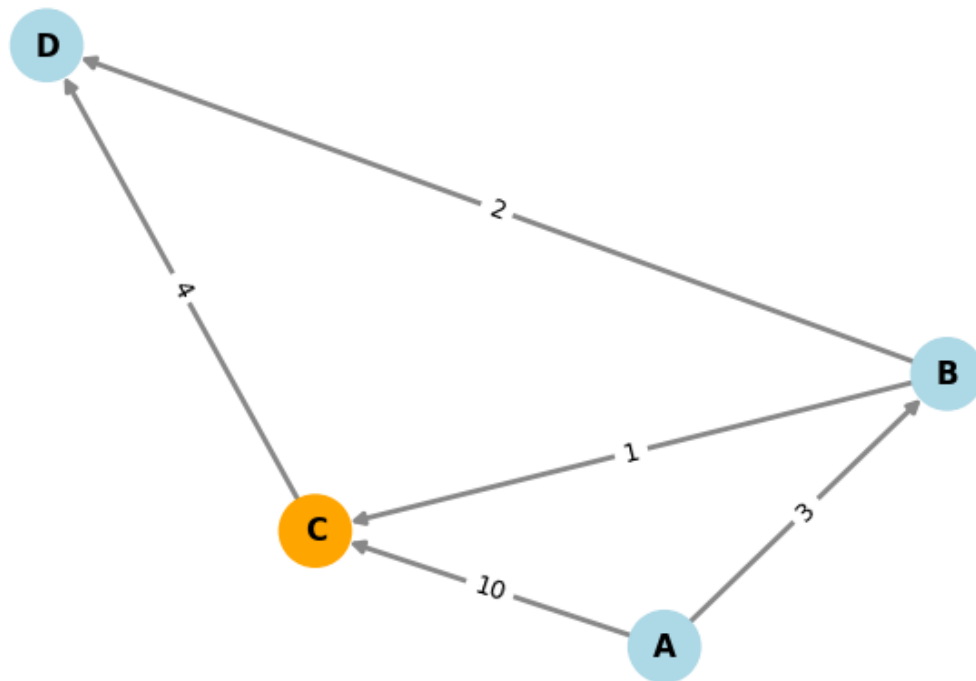
Passo 3: Considerando 'C' como intermediário

- Verificamos se passar por 'C' reduz o custo entre outros pares de filiais.
- Nenhuma atualização de custo neste passo.

Matriz de custos atualizada:

	A	B	C	D
A	0.0	3.0	4.0	5.0
B	∞	0.0	1.0	2.0
C	∞	∞	0.0	4.0
D	∞	∞	∞	0.0

Floyd-Warshall - Passo 3 (intermediário: C)



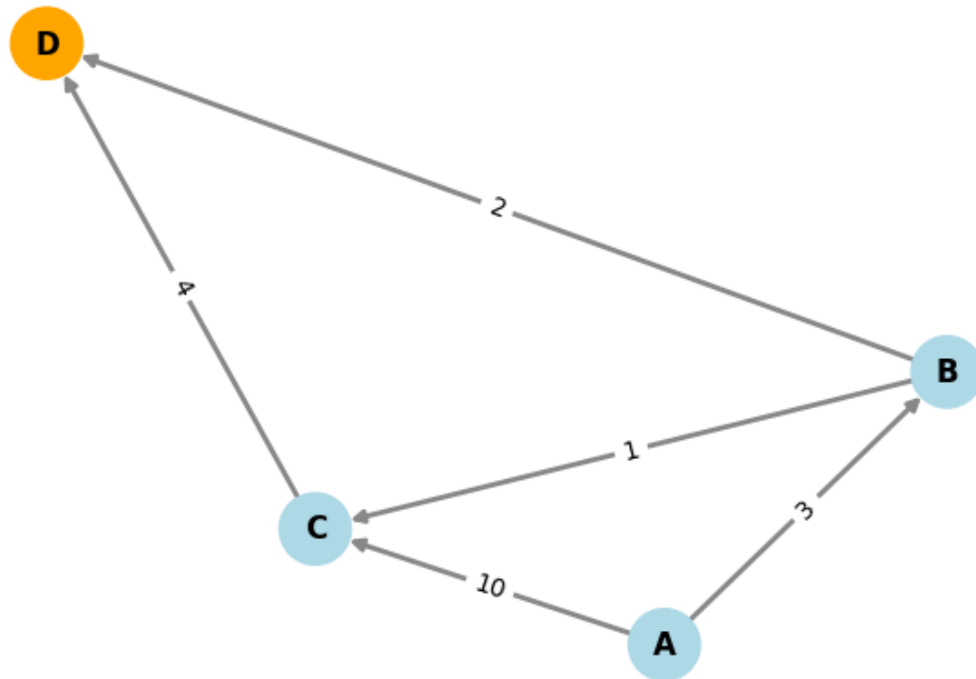
Passo 4: Considerando 'D' como intermediário

- Verificamos se passar por 'D' reduz o custo entre outros pares de filiais.
- Nenhuma atualização de custo neste passo.

Matriz de custos atualizada:

	A	B	C	D
A	0.0	3.0	4.0	5.0
B	∞	0.0	1.0	2.0
C	∞	∞	0.0	4.0
D	∞	∞	∞	0.0

Floyd-Warshall - Passo 4 (intermediário: D)



Matriz final de menores custos entre todos os pares (Floyd-Warshall):

	A	B	C	D
A	0.0	3.0	4.0	5.0
B	∞	0.0	1.0	2.0
C	∞	∞	0.0	4.0
D	∞	∞	∞	0.0

0.8.4 Questão 8: Caminhos Euleriano e Hamiltoniano em um grafo de cidades

Considere o grafo $G(V, A)$ abaixo, onde os vértices representam cidades e as arestas representam estradas entre elas. O objetivo é analisar as propriedades do grafo em relação aos trajetos Hamiltonianos e Eulerianos.

Grafo $G(V, A)$: - Vértices: $V = \{A, B, C, D, E, F\}$ - Arestas e seus pesos: 1. A - B: 5

2. A - C: 7

3. B - C: 4

4. B - D: 6

5. C - D: 3

6. C - E: 8

7. D - E: 2

8. D - F: 9

9. E - F: 6

Análise as afirmações abaixo:

- a) No Grafo $G(V, A)$, não é possível ter um trajeto Euleriano de “A” até “F”.
- b) No Grafo $G(V, A)$, não é possível ter um trajeto Hamiltoniano de “A” até “F”.
- c) É possível construir um trajeto Euleriano entre “A” e “F”, com tamanho total igual a 28.
- d) É possível construir um trajeto Hamiltoniano entre “A” e “F”, com tamanho total igual a 30.

Com base nas propriedades do grafo, qual das afirmações acima é correta? Justifique sua resposta detalhadamente, explicando:

- O que caracteriza um caminho Euleriano e um caminho Hamiltoniano.
- Como identificar se o grafo admite cada tipo de caminho entre os vértices dados.
- Se possível, mostre o caminho correspondente e o cálculo do tamanho total.

0.9 Caminho Euleriano e Caminho Hamiltoniano

0.9.1 Caminho Euleriano

Um **caminho Euleriano** em um grafo é um trajeto que percorre **todas as arestas exatamente uma vez**.

- Em grafos não direcionados, existe um caminho Euleriano entre dois vértices se e somente se **apenas esses dois vértices têm grau ímpar** (os demais têm grau par). - Se todos os vértices têm grau par, existe um **circuito Euleriano** (começa e termina no mesmo vértice).

0.9.2 Caminho Hamiltoniano

Um **caminho Hamiltoniano** é um trajeto que passa **por todos os vértices exatamente uma vez** (não necessariamente todas as arestas). - Se existe um caminho Hamiltoniano que começa em um vértice e termina em outro, dizemos que o grafo é **semi-hamiltoniano**. - Se existe um ciclo Hamiltoniano (começa e termina no mesmo vértice), o grafo é **hamiltoniano**.

0.9.3 Como identificar no grafo?

- **Euleriano:** Verifique o grau dos vértices. Se só dois vértices têm grau ímpar (por exemplo, A e F), pode haver um caminho Euleriano de A até F.
 - **Hamiltoniano:** Não há um critério simples, é preciso testar as possibilidades (busca exaustiva ou algoritmos específicos).
-

0.9.4 Resumindo:

- **Caminho Euleriano:** percorre todas as arestas uma vez.
- **Caminho Hamiltoniano:** passa por todos os vértices uma vez.

No contexto do seu exercício, para saber se existe um caminho Euleriano ou Hamiltoniano entre A e F, analise os graus dos vértices e tente construir os caminhos conforme as definições acima.

[98]: *# Análise detalhada de caminhos Euleriano e Hamiltoniano (Questão 8)*

```
import networkx as nx

# Criação do grafo
G8 = nx.Graph()
G8.add_weighted_edges_from([
    ('A', 'B', 5),
    ('A', 'C', 7),
    ('B', 'C', 4),
    ('B', 'D', 6),
    ('C', 'D', 3),
    ('C', 'E', 8),
    ('D', 'E', 2),
    ('D', 'F', 9),
    ('E', 'F', 6)
])

# Verificando caminho Euleriano de A até F
grau_impar = [v for v in G8.nodes if G8.degree(v) % 2 == 1]
print("Vértices de grau ímpar:", grau_impar)
if len(grau_impar) == 2 and 'A' in grau_impar and 'F' in grau_impar:
    print("Existe caminho Euleriano de A até F.")
    # Encontrar caminho Euleriano (se existir)
    caminho_euleriano = list(nx.eulerian_path(G8, source='A'))
    print("Caminho Euleriano:", caminho_euleriano)
    custo_euleriano = sum(G8.get_edge_data(u, v)['weight'] for u, v in
        caminho_euleriano)
    print("Custo total do caminho Euleriano:", custo_euleriano)
else:
    print("Não existe caminho Euleriano de A até F.")

# Verificando caminho Hamiltoniano de A até F
def hamiltonian_path(G, start, end):
    from itertools import permutations
    nodes = list(G.nodes)
    nodes.remove(start)
    nodes.remove(end)
    for perm in permutations(nodes):
        path = [start] + list(perm) + [end]
        if all(G.has_edge(path[i], path[i+1]) for i in range(len(path)-1)):
            return path
    return None

caminho_hamiltoniano = hamiltonian_path(G8, 'A', 'F')
if caminho_hamiltoniano:
    print("Existe caminho Hamiltoniano de A até F.")
```

```

    print("Caminho Hamiltoniano:", caminho_hamiltoniano)
    custo_hamiltoniano = sum(G8.get_edge_data(caminho_hamiltoniano[i],
↪caminho_hamiltoniano[i+1])['weight'] for i in
↪range(len(caminho_hamiltoniano)-1))
    print("Custo total do caminho Hamiltoniano:", custo_hamiltoniano)
else:
    print("Não existe caminho Hamiltoniano de A até F.")

# Visualização do grafo
import matplotlib.pyplot as plt
pos = nx.spring_layout(G8, seed=42)
plt.figure(figsize=(7,5))
nx.draw(G8, pos, with_labels=True, node_color='lightblue', node_size=900,
↪font_weight='bold', edge_color='#888', width=2)
labels = nx.get_edge_attributes(G8, 'weight')
nx.draw_networkx_edge_labels(G8, pos, edge_labels=labels)
plt.title("Grafo para análise de caminhos Euleriano e Hamiltoniano")
plt.show()

```

Vértices de grau ímpar: ['B', 'E']

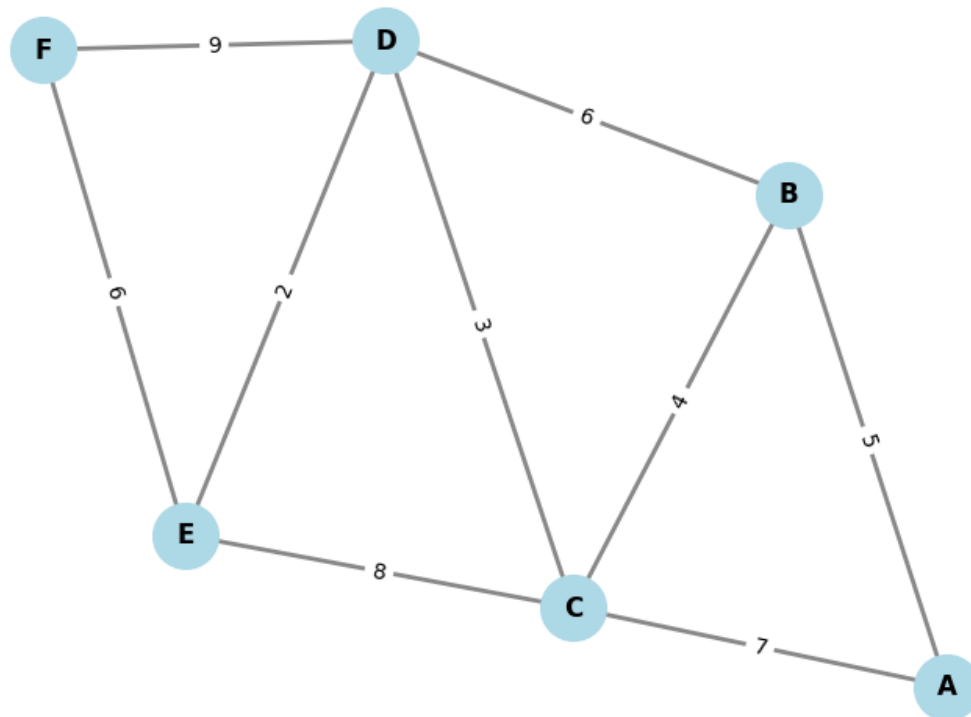
Não existe caminho Euleriano de A até F.

Existe caminho Hamiltoniano de A até F.

Caminho Hamiltoniano: ['A', 'B', 'C', 'D', 'E', 'F']

Custo total do caminho Hamiltoniano: 20

Grafo para análise de caminhos Euleriano e Hamiltoniano



0.9.5 Questão 9: Menor custo de transporte com Dijkstra passo a passo

Uma empresa de entregas deseja calcular o menor custo de transporte entre o depósito principal (vértice A) e os demais centros de distribuição em uma cidade. O grafo abaixo representa os centros (vértices) e as estradas (arestas com pesos):

- Vértices: A, B, C, D, E
- Arestas e seus custos:
 1. A – B: 2
 2. A – C: 5
 3. B – C: 1
 4. B – D: 2
 5. C – D: 3
 6. C – E: 1
 7. D – E: 2

Utilizando o algoritmo de Dijkstra a partir do vértice A, mostre o passo a passo do algoritmo, incluindo a tabela de menores custos (dv) e predecessores (pv) para cada vértice em cada passo, além da visualização do grafo.

0.10 Algoritmo de Dijkstra

O **algoritmo de Dijkstra** é utilizado para encontrar o menor caminho (menor custo) entre um vértice de origem e todos os outros vértices em um grafo ponderado com pesos não negativos.

0.10.1 Como funciona o Dijkstra?

1. **Inicialize** o custo de todos os vértices como infinito, exceto o vértice de origem, que recebe custo zero.
2. Marque todos os vértices como não visitados.
3. Escolha o vértice não visitado de menor custo atual.
4. Para cada vizinho desse vértice, atualize o custo se for possível chegar até ele por um caminho mais barato.
5. Marque o vértice atual como visitado.
6. Repita até visitar todos os vértices ou não houver mais atualizações possíveis.

0.10.2 Características

- Garante o menor caminho em grafos com pesos positivos.
- Utiliza uma fila de prioridade (ou seleção sequencial) para escolher o próximo vértice.
- Complexidade: $O(n^2)$ para implementação simples, $O(m \log n)$ com heap (n = vértices, m = arestas).

0.10.3 Exemplo de aplicação

- Cálculo de rotas mais curtas em mapas e redes de transporte.
- Otimização de custos em redes de comunicação e logística.

Resumo:

O algoritmo de Dijkstra encontra o menor custo de um ponto a todos os outros em um grafo com pesos positivos, atualizando os custos mínimos e predecessores a cada passo.

```
[99]: # Dijkstra passo a passo a partir do vértice A
```

```
import networkx as nx
import matplotlib.pyplot as plt
import pandas as pd
```

```
G9 = nx.Graph()
G9.add_weighted_edges_from([
    ('A', 'B', 2),
    ('A', 'C', 5),
    ('B', 'C', 1),
    ('B', 'D', 2),
```

```

        ('C', 'D', 3),
        ('C', 'E', 1),
        ('D', 'E', 2)
    ])

vertices = ['A', 'B', 'C', 'D', 'E']
dv = {v: float('inf') for v in vertices}
pv = {v: None for v in vertices}
dv['A'] = 0
visitados = set()

pos = nx.spring_layout(G9, seed=42)

print("Passo 0: Inicialização")
print(" Custos mínimos (dv):", dv)
print(" Predecessores (pv):", pv)
print(" Visitados:", visitados)
plt.figure(figsize=(6,4))
nx.draw(G9, pos, with_labels=True, node_color=['orange' if n == 'A' else
↪ 'lightgray' for n in G9.nodes], node_size=800, font_weight='bold',
↪ edge_color='#888', width=2)
labels = nx.get_edge_attributes(G9, 'weight')
nx.draw_networkx_edge_labels(G9, pos, edge_labels=labels)
plt.title("Dijkstra - Passo 0 (início)")
plt.show()

passo = 1
while len(visitados) < len(vertices):
    # Seleciona o vértice não visitado de menor custo
    u = min((v for v in vertices if v not in visitados), key=lambda v: dv[v])
    visitados.add(u)
    print(f"\nPasso {passo}:")
    print(f" Vértice escolhido: {u} (dv={dv[u]})")
    for v in G9.neighbors(u):
        if v not in visitados:
            custo = G9[u][v]['weight']
            if dv[v] > dv[u] + custo:
                print(f" * Atualizando dv[{v}]: {dv[v]} → {dv[u] + custo}
↪ (via {u})")
                dv[v] = dv[u] + custo
                pv[v] = u
    print(" Custos mínimos (dv):", dv)
    print(" Predecessores (pv):", pv)
    print(" Visitados:", visitados)

    # Visualização do grafo no passo

```



```

    node_colors = ['orange' if n == u else ('lightgreen' if n in visitados else
↪ 'lightgray') for n in G9.nodes]
    edge_colors = []
    for e in G9.edges:
        if (pv[e[1]] == e[0]) or (pv[e[0]] == e[1]):
            edge_colors.append('red')
        else:
            edge_colors.append('#888')
    plt.figure(figsize=(6,4))
    nx.draw(G9, pos, with_labels=True, node_color=node_colors, node_size=800,
↪ font_weight='bold', edge_color=edge_colors, width=2)
    nx.draw_networkx_edge_labels(G9, pos, edge_labels=labels)
    plt.title(f"Dijkstra - Passo {passo} (incluído: {u})")
    plt.show()
    passo += 1

print("\nResultado final de Dijkstra a partir de A:")
print("Custos mínimos (dv):", dv)
print("Predecessores (pv):", pv)

```

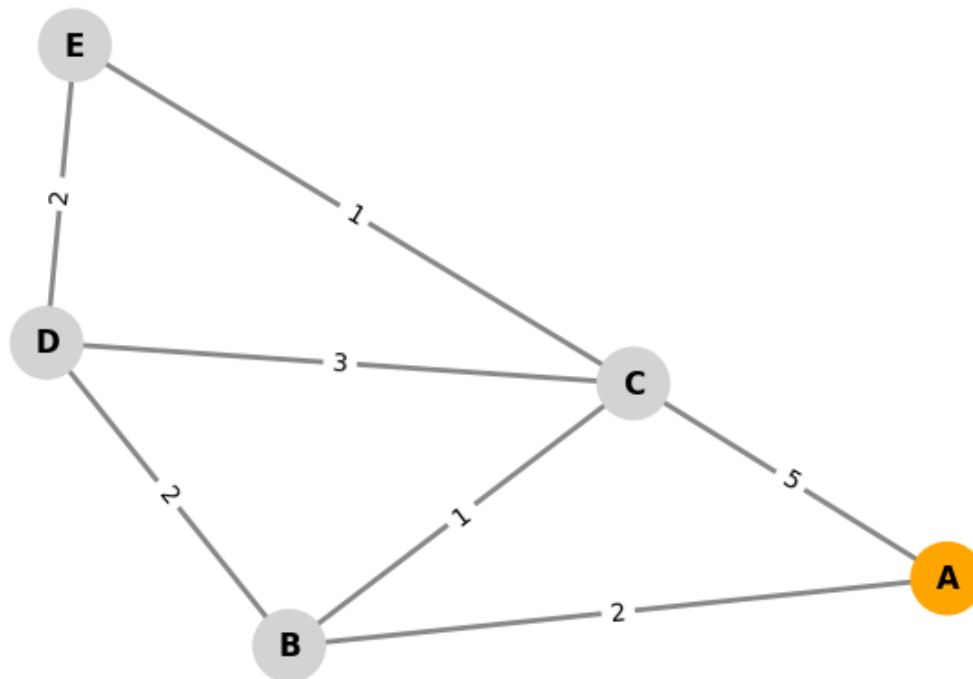
Passo 0: Inicialização

Custos mínimos (dv): {'A': 0, 'B': inf, 'C': inf, 'D': inf, 'E': inf}

Predecessores (pv): {'A': None, 'B': None, 'C': None, 'D': None, 'E': None}

Visitados: set()

Dijkstra - Passo 0 (início)



Passo 1:

Vértice escolhido: A (dv=0)

* Atualizando dv[B]: inf → 2 (via A)

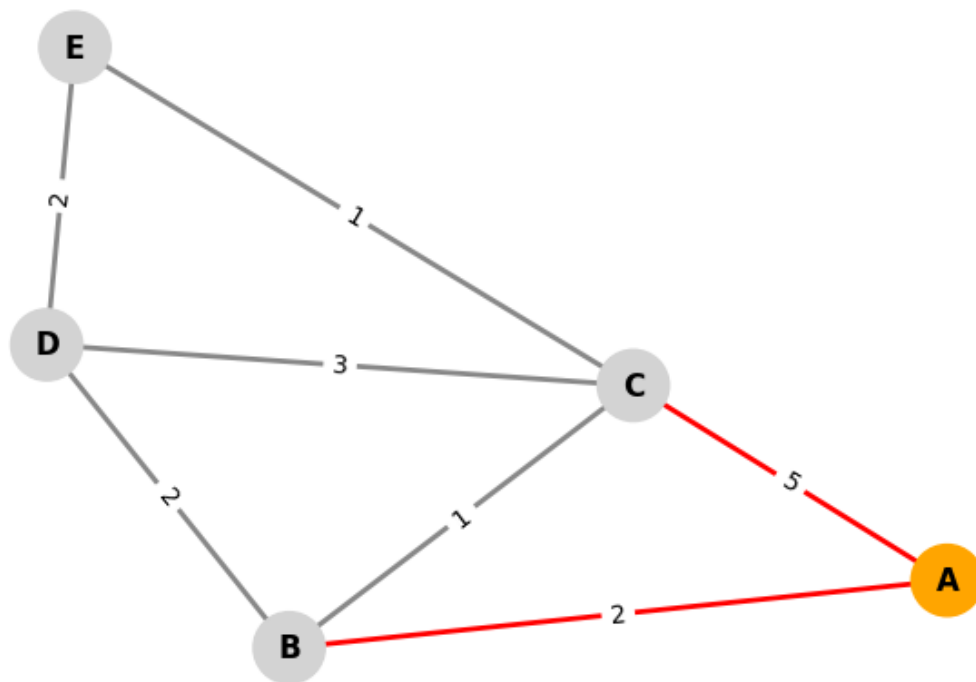
* Atualizando dv[C]: inf → 5 (via A)

Custos mínimos (dv): {'A': 0, 'B': 2, 'C': 5, 'D': inf, 'E': inf}

Predecessores (pv): {'A': None, 'B': 'A', 'C': 'A', 'D': None, 'E': None}

Visitados: {'A'}

Dijkstra - Passo 1 (incluído: A)



Passo 2:

Vértice escolhido: B (dv=2)

* Atualizando dv[C]: 5 → 3 (via B)

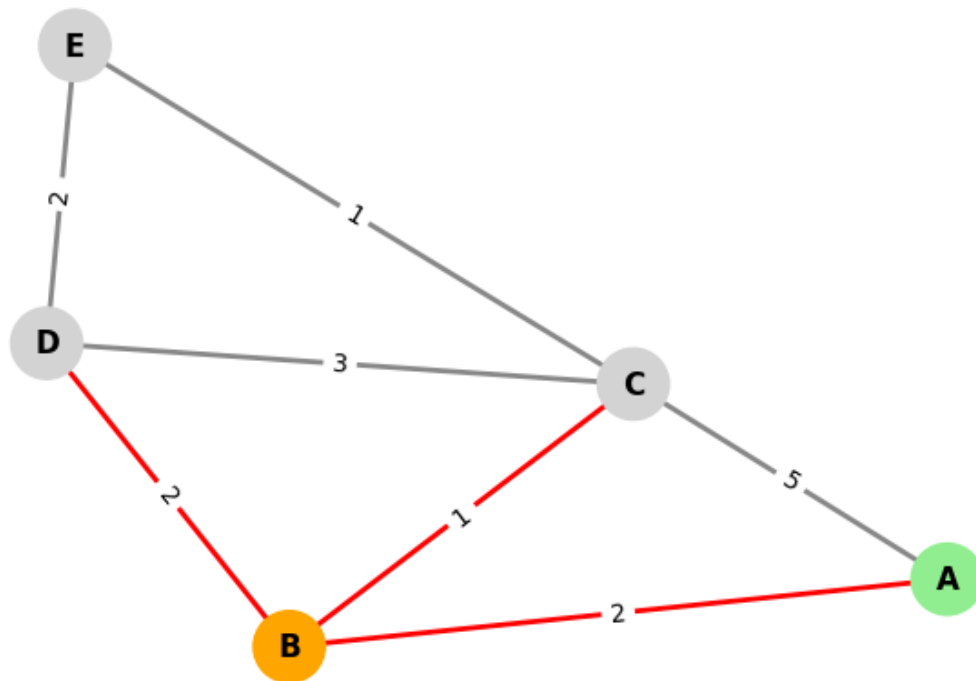
* Atualizando dv[D]: inf → 4 (via B)

Custos mínimos (dv): {'A': 0, 'B': 2, 'C': 3, 'D': 4, 'E': inf}

Predecessores (pv): {'A': None, 'B': 'A', 'C': 'B', 'D': 'B', 'E': None}

Visitados: {'A', 'B'}

Dijkstra - Passo 2 (incluído: B)



Passo 3:

Vértice escolhido: C (dv=3)

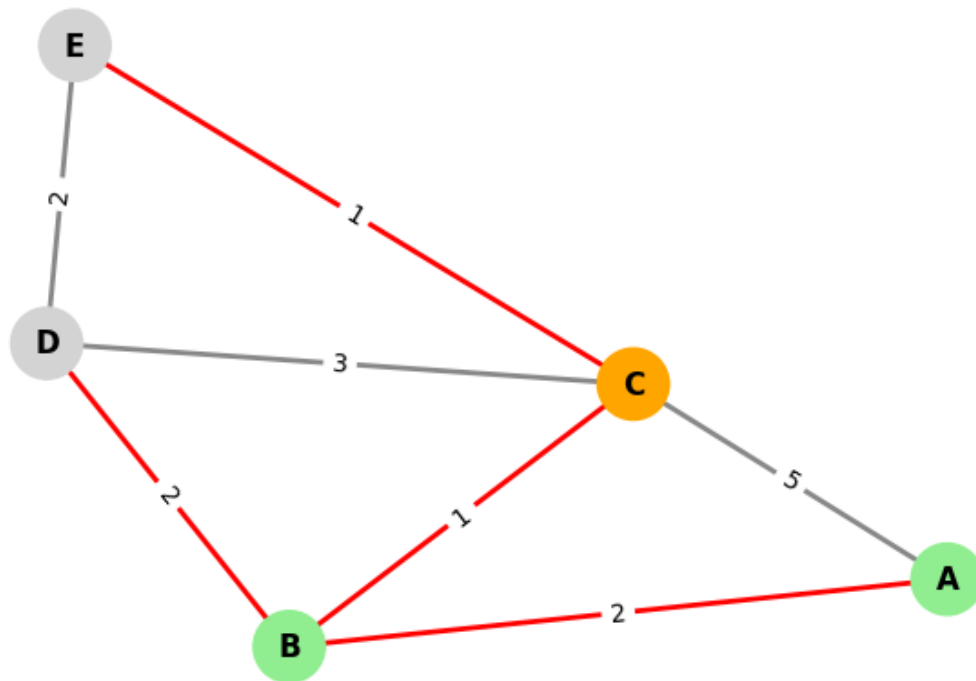
* Atualizando dv[E]: $\text{inf} \rightarrow 4$ (via C)

Custos mínimos (dv): {'A': 0, 'B': 2, 'C': 3, 'D': 4, 'E': 4}

Predecessores (pv): {'A': None, 'B': 'A', 'C': 'B', 'D': 'B', 'E': 'C'}

Visitados: {'C', 'A', 'B'}

Dijkstra - Passo 3 (incluído: C)



Passo 4:

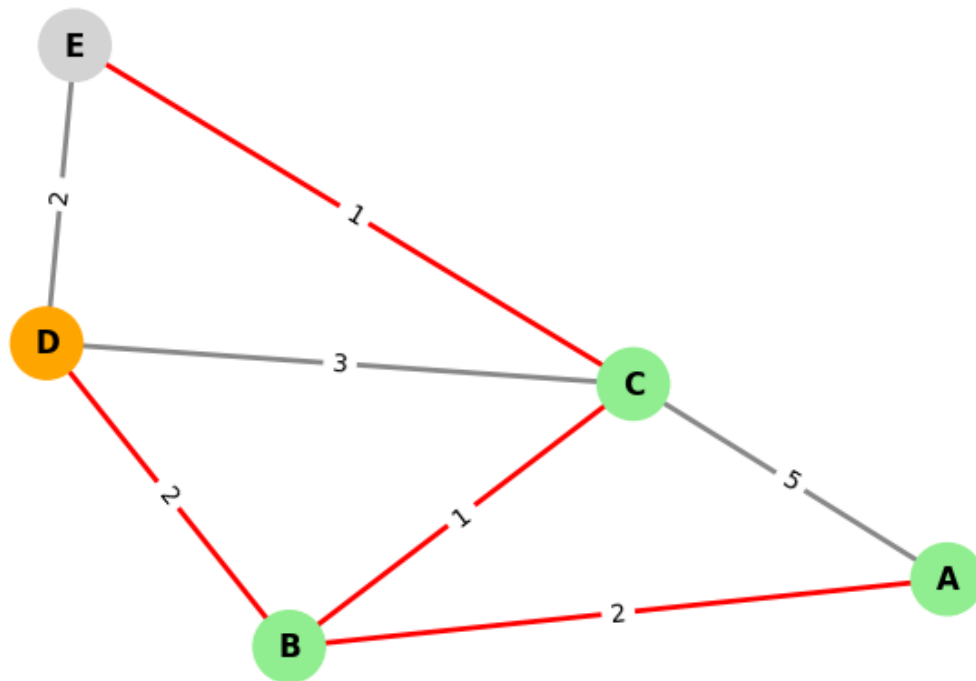
Vértice escolhido: D (dv=4)

Custos mínimos (dv): {'A': 0, 'B': 2, 'C': 3, 'D': 4, 'E': 4}

Predecessores (pv): {'A': None, 'B': 'A', 'C': 'B', 'D': 'B', 'E': 'C'}

Visitados: {'C', 'A', 'D', 'B'}

Dijkstra - Passo 4 (incluído: D)



Passo 5:

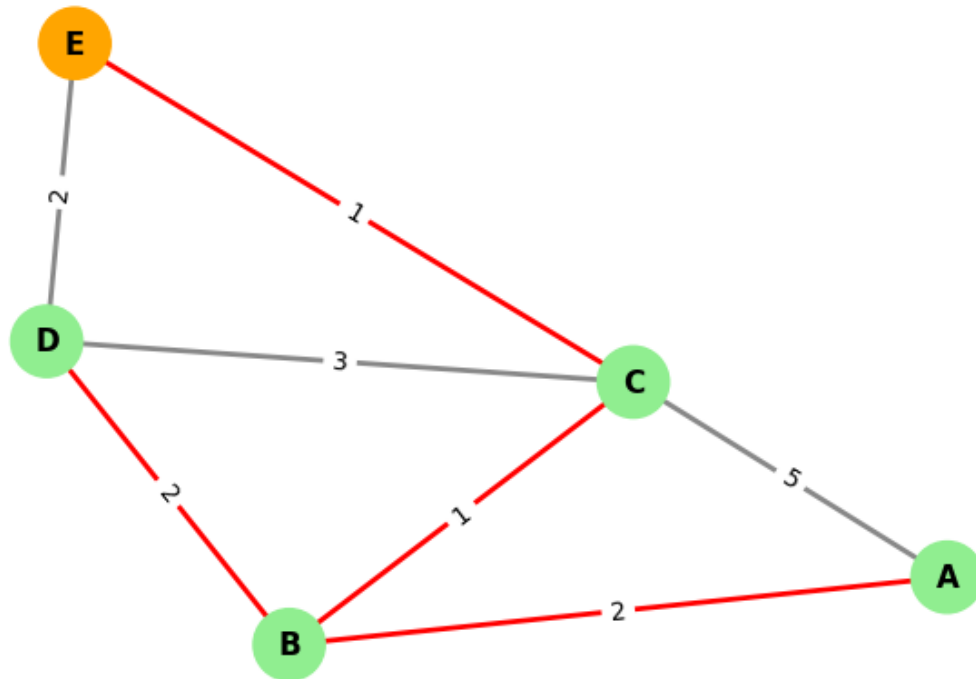
Vértice escolhido: E (dv=4)

Custos mínimos (dv): {'A': 0, 'B': 2, 'C': 3, 'D': 4, 'E': 4}

Predecessores (pv): {'A': None, 'B': 'A', 'C': 'B', 'D': 'B', 'E': 'C'}

Visitados: {'A', 'D', 'B', 'E', 'C'}

Dijkstra - Passo 5 (incluído: E)



Resultado final de Dijkstra a partir de A:

Custos mínimos (dv): {'A': 0, 'B': 2, 'C': 3, 'D': 4, 'E': 4}

Predecessores (pv): {'A': None, 'B': 'A', 'C': 'B', 'D': 'B', 'E': 'C'}

0.10.4 Questão 10: Matriz de custos (matriz de adjacência valorada) de um grafo de cidades

Uma empresa de energia está planejando instalar cabos entre cinco cidades: A, B, C, D e E. O custo de instalação entre cada par de cidades está representado nas arestas do grafo abaixo:

- Arestas e custos:
 - A-B: 4
 - A-C: 7
 - B-C: 2
 - B-D: 5
 - C-D: 3
 - C-E: 6
 - D-E: 1

Monte a matriz de custos (matriz de adjacência valorada) desse grafo e explique como ela representa os custos de instalação entre as cidades.

```
[100]: # Questão 10: Matriz de custos (matriz de adjacência valorada)

import networkx as nx
import pandas as pd

G10 = nx.Graph()
G10.add_weighted_edges_from([
    ('A', 'B', 4),
    ('A', 'C', 7),
    ('B', 'C', 2),
    ('B', 'D', 5),
    ('C', 'D', 3),
    ('C', 'E', 6),
    ('D', 'E', 1)
])

vertices = ['A', 'B', 'C', 'D', 'E']
matriz = pd.DataFrame(float('inf'), index=vertices, columns=vertices)
for u, v, data in G10.edges(data=True):
    matriz.loc[u, v] = data['weight']
    matriz.loc[v, u] = data['weight']
for v in vertices:
    matriz.loc[v, v] = 0

print("Matriz de custos (matriz de adjacência valorada):")
print(matriz.replace(float('inf'), '∞'))

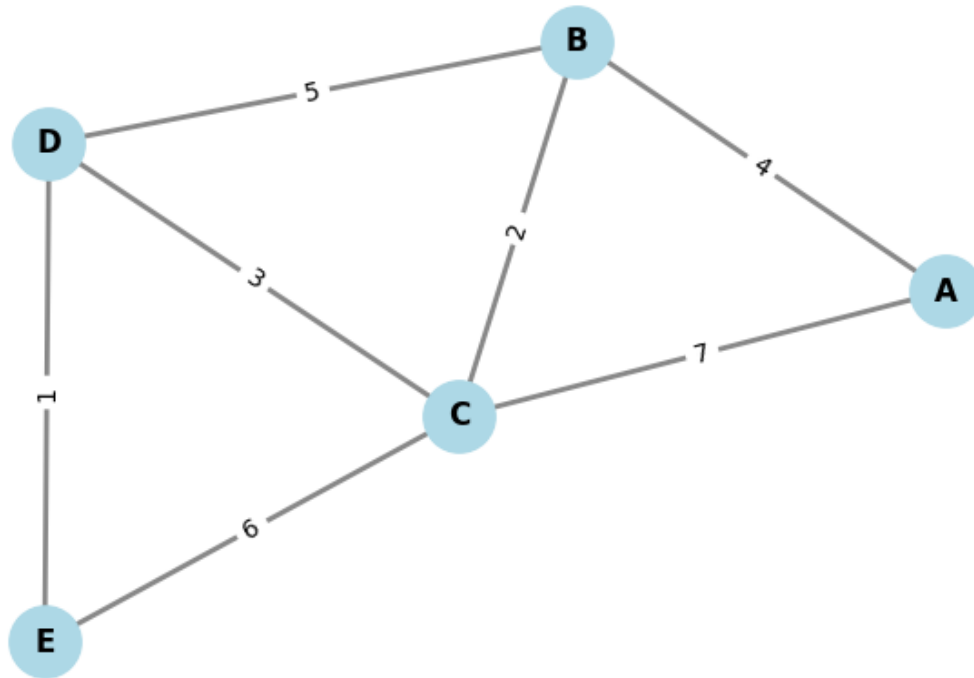
# Visualização do grafo
import matplotlib.pyplot as plt
pos = nx.spring_layout(G10, seed=42)
plt.figure(figsize=(6,4))
nx.draw(G10, pos, with_labels=True, node_color='lightblue', node_size=800,
        font_weight='bold', edge_color='#888', width=2)
labels = nx.get_edge_attributes(G10, 'weight')
nx.draw_networkx_edge_labels(G10, pos, edge_labels=labels)
plt.title("Grafo de cidades e custos de instalação")
plt.show()

# Explicação:
print("\nExplicação:")
print("A matriz de adjacência valorada mostra o custo mínimo para instalar_
    ↪ cabos entre cada par de cidades.")
print("Se não existe ligação direta entre duas cidades, o valor é ∞ (infinito).
    ↪ ")
print("A diagonal principal é zero, pois o custo de uma cidade para ela mesma é_
    ↪ zero.")
```

Matriz de custos (matriz de adjacência valorada):

	A	B	C	D	E
A	0.0	4.0	7.0	∞	∞
B	4.0	0.0	2.0	5.0	∞
C	7.0	2.0	0.0	3.0	6.0
D	∞	5.0	3.0	0.0	1.0
E	∞	∞	6.0	1.0	0.0

Grafo de cidades e custos de instalação



Explicação:

A matriz de adjacência valorada mostra o custo mínimo para instalar cabos entre cada par de cidades.

Se não existe ligação direta entre duas cidades, o valor é ∞ (infinito).

A diagonal principal é zero, pois o custo de uma cidade para ela mesma é zero.