ALESSANDRO SOARES DA SILVA

MATRICULA: 20231023705

6° LISTA DE ALGORITMO

1. Defina um grafo acíclico direcionado relacionado a área do seu mestrado/doutorado e use a sua implementação para ordenar topologicamente esse grafo.

Resposta:

Para meu projeto de mestrado tenho a ideia de medir o desempenho de um servidor com base em seu consumo energético e identificar possíveis sobrecargas ou falhas com base em picos de consumo de corrente e tensão, claro que ainda é uma ideia embrionaria, mas acredito que será possível desenvolve-lo seguindo os passos a seguir:

1. Escolha dos instrumetos de medição

- Adiquirir um medidor de consumo de energia elétrico, que pode registrar a potência consumida ao longo do tempo.
- Utilizar o dispositivo para medir a corrente elétrica, para capturar picos de corrente.

2. Preparar o ambiente

- Garantir que o servidor esteja configurado da manneira que desejamos testar. Isso pode incluir a carga de trabalho que se deseja avaliar.

3. Conectar os Instrumentos

- Conectar o medidor de consumo de energia elétrica à fonte de alimentação do servidor.
- Usar o medidor amperímetro para medir a corrente elétrica que flui para o servidor.

4. Coleta de dados

- Registrar os dados de consumo de energia e corrente elétrica durante o período que se deseja analisar. É importante garantir que as medições sejam sincronizadas.

5. Análise de dados

- Utilizar software apropriado para analisar os dados coletados. Onde podemos calcular a média de consumo de energia, identificar picos de corrente e tensão, correlacionar esses dados com o desempenho do servidor.
- Observar os momento se que ocorrem picos de consumo de corrente e avaliar se estes são relacionados a períodos de sobrecargas ou falhas no servidor.

6. Interpretação dos resultados com reconhecimento de padrões

- Usar ténicas de detecção de anomalias, como Isolation Forest, One-Class SVM ou métodos baseados em redes neurais, para identificar automaticamente picos de consumo de corrente incomuns que podem estar associados a falhas ou problemas de desempenho no servidor.

- Implementar algoritmos de classificação para categorizar o comportarmento do servidor com base nos dados coletados. Por exemplo, classificar os padrões de consumo de energia em “normal”, “sobrecarga” e “falha”.
- Utilizar técnicas de aprendizado não supervisionado, como clustering, para agrupar dados semelhantes de consumo de energia e corrente, o que pode ajudar a identificar grupos de servidores com desempenho semelhante.

7. Tomada de ações corretivas com machine learning

- Implementar um sistema de recomendação com base em aprendizado de máquina que possa sugerir ações corretivas com base na análise, Por exemplo, o sistema pode recomendar ajustes na carga de trabalho, na configuração de hardware ou na políticas de resfiramento para otimizar o desempenho.
- Utilizar modelos de previsão de falhas para antecipar problemas de desempenho ou falhas iminentes com base nas tendências históricas de consumo de energia e corrente. Isso permite a manutenção proativa.
- Implementar algoritmos de otimização automatizada que possam ajustar automaticamente os parâmetros do servidor em tempo real para otimizar o consumo de energia e o desempenho com base nas condicçoes atuais.
- Configuração de um sistema de feedback contínuo em que os dados coletados sejam usados para treinar modelos de machine learning em tempo real, permitindo que o sistema se adapte às mudanças nas condições e melhore suas recomendações ao longo do tempo.

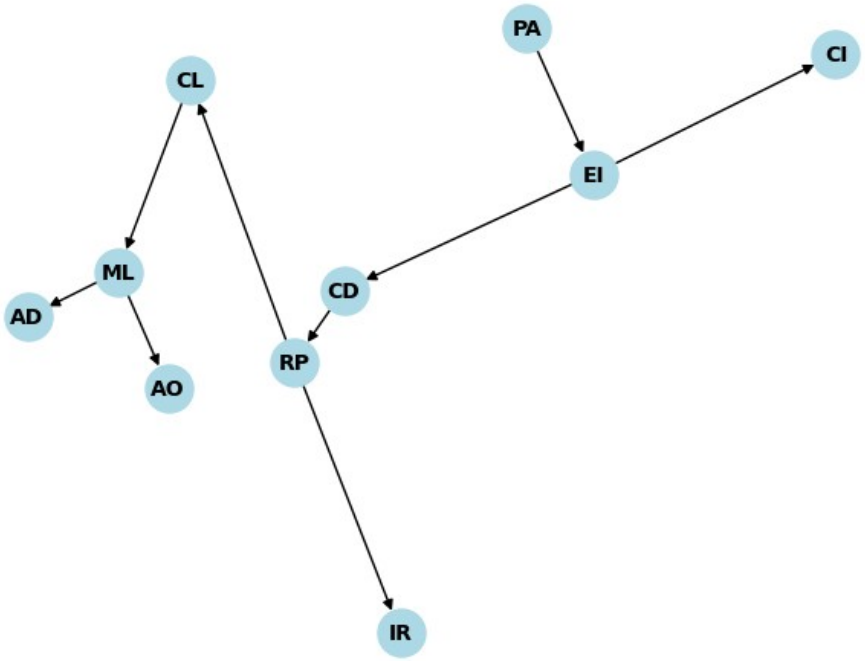
Algoritmo implementado

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

# A função implementa a ordenação topológica usando o algoritmo
# de Busca em Profundidade
def ordenacao_topologica_dfs(grafo):
    # A função dfs é uma função interna que realiza a busaca
    # em profundidade a partir de um nó "item". Ela é usada
    # para explorar os nós do grafo recursivamente, seguindo
    # as dependências.
    def dfs(item):
        # visitado é um conjunto que rastria os nós visitados
        # durante a busaca em profundidade. Isso garante que
        # não haja ciclos no grafo
        visitado.add(item)
        for prerequisito in grafo.get(item, []):
            if prerequisito not in visitado:
                dfs(prerequisito)
```

```
        # resultado é uma lista que armazena os nós na ordem
        # correta de ordenação topológica.
        resultado.append(item)
    visitado = set()
    resultado = []
    # o loop abaixo itera por todos os nós no grafo, denteo do
    # do loop, verifica se o nó "item" não está no conjunto
    # "visitado", ou seja, não foi visitado
    for item in grafo:
        if item not in visitado:
            # Se o nó item não foi visitado, a função dfs é chamada
            # para explorar o grafo a partir desse nó
            dfs(item)
    return resultado[::-1]  # Inverte a lista resultado para obter a ordenação t

grafo_projeto = {
    'PA': [],               # PA = Preparar Ambiente
    'EI': ['PA'],           # EI = Escolha dos Intrumentos
    'CI': ['EI'],           # CI = Conectar INstrumentos
    'CD': ['EI'],           # CD = Coletar dados
    'RP': ['CD'],           # RP = Reconhecimento de Padrões
    'IR': ['RP'],           # IR = Interpretação dos resultados
    'CL': ['RP'],           # CL = Clustering
    'ML': ['CL'],           # ML = Machine Learning
    'AO': ['ML'],           # AO = Algoritmos de otimização
    'AD': ['ML'],           # AD = Análise de Dados
}
ordem_projeto = ordenacao_topologica_dfs(grafo_projeto)
print("Ordem do projeto (DFS):", ordem_projeto)

grafo_projeto = nx.DiGraph()

grafo_projeto.add_node('PA')
grafo_projeto.add_node('EI')
grafo_projeto.add_node('CI')
grafo_projeto.add_node('CD')
grafo_projeto.add_node('RP')
grafo_projeto.add_node('IR')
grafo_projeto.add_node('CL')
grafo_projeto.add_node('ML')
grafo_projeto.add_node('AO')
grafo_projeto.add_node('AD')

grafo_projeto.add_edge('PA', 'EI')
grafo_projeto.add_edge('EI', 'CI')
grafo_projeto.add_edge('EI', 'CD')
grafo_projeto.add_edge('CD', 'RP')
grafo_projeto.add_edge('RP', 'IR')
grafo_projeto.add_edge('RP', 'CL')
grafo_projeto.add_edge('CL', 'ML')
grafo_projeto.add_edge('ML', 'AO')
grafo_projeto.add_edge('ML', 'AD')
```

```
pos = nx.spring_layout(grafo_projeto)
nx.draw(grafo_projeto, pos, with_labels=True, node_size=500,
node_color='lightblue', font_size=10, font_color='black', font_weight='bold')
plt.show()
```

Grafico do grafo em questão



2. Implemente e aponte vantagens e desvantagens dos algoritmos de Kruskal e Prim para gerar uma árvore de abrangência mínima. Use exemplos práticos da sua área para demonstrar suas conclusões.

Árvore geradora de custo mínimo: é a árvore geradorea de menor custo dentre todas as possíveis em um grafo.

Algoritmo de Kruskal:

1. **Tipo de Grafo:** Funciona em qualquer tipo de grafo não direcionado e ponderado, incluindo grafos conectados ou desconectados.
2. **Seleção de Arestas:** Inicialmente, nenhuma aresta está na MST. O algoritmo seleciona arestas em ordem crescente de peso, adicionando a próxima aresta mais leve que não forma um ciclo na MST parcial, o algoritmo termina quando iterar n-1 vezes, onde n é o número de vértices. Gera a árvore conectando os vértices e não as arestas, além disso é possível formar árvores separadas e posteriormente uni-las.
3. **Estrutura de Dados:** Geralmente é implementado com uma estrutura de dados de lista de arestas (uma lista de todas as arestas) e uma estrutura de dados de conjunto disjunto para verificar a conectividade entre vértices, mas pode trabalhar também com matriz de adjacência, no entanto perde performânce

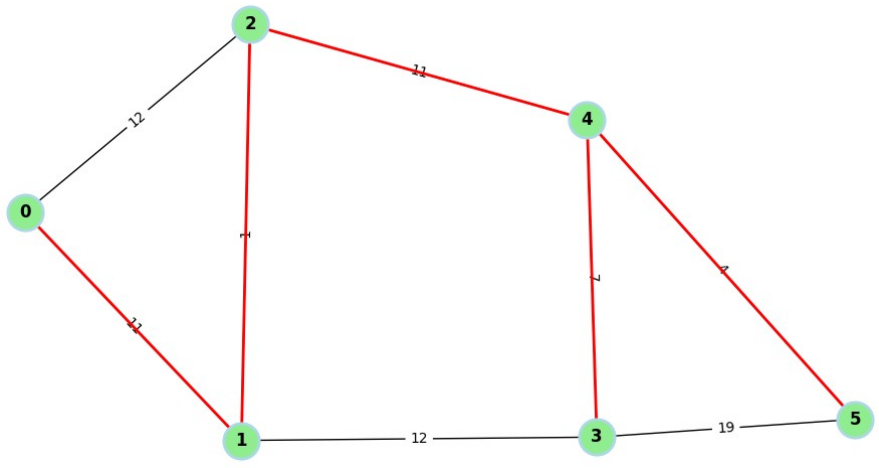
4. **Complexidade:** A complexidade de tempo do algoritmo de Kruskal é tipicamente O(E * log(E)) ou O(E * log(V)), onde E é o número de arestas e V é o número de vértices.

Implementação do Algoritmo

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

# define quem é o pai na árvore montada
def encontrar(pai, i):
    if pai[i] == i:
        return i
    return encontrar(pai, pai[i])
# altera os valores da lista "pai" e "rank" para definir quem é pai no grafo
def unir(pai, rank, u, v):     # pai[1,1,1,4,4,4] - rank[0,1,0,0,1,0]
    raiz_u = encontrar(pai, u)
    raiz_v = encontrar(pai, v)
    if rank[raiz_u] < rank[raiz_v]:
        pai[raiz_u] = raiz_v
    elif rank[raiz_u] > rank[raiz_v]:
        pai[raiz_v] = raiz_u
    else:
        pai[raiz_v] = raiz_u
        rank[raiz_u] += 1
def kruskal(vertices, arestas):
    resultado = []
    i = 0
    e = 0
    # ordena a lista "aresta" de forma crescente com base no peso
    arestas = sorted(arestas, key=lambda item: item[2])
    # lista todos os vertices existentes
    pai = list(range(vertices))
    # cria um vetor de valores 0 de tamanho da quantidade de vértices
    rank = [0] * vertices

    while e < vertices - 1:
        u, v, w = arestas[i]
        i += 1
        x = encontrar(pai, u)
        y = encontrar(pai, v)
        # se os vertices são diferentes unir os dois formando uma sub-árvore
        if x != y:
            e += 1
            resultado.append([u, v, w])
            unir(pai, rank, x, y)
    return resultado

```

```
vertices = 6
arestas = [(0, 1, 11), (0, 2, 12), (1, 2, 1), (1, 3, 12), (2, 4, 11), (3, 4, 7),
           (3, 5, 19), (4, 5, 4)]

inicio = time.time()
mst = kruskal(vertices, arestas)
final = time.time()

print("Arestas no MST (Árvore de Abrangência Mínima):")
for u, v, w in mst:
    print(f"{u} - {v}: {w}")
print(final-inicio)

# Crie um gráfico direcionado para representar o grafo original
G = nx.Graph()

for u, v, w in arestas:
    G.add_edge(u, v, weight=w)

# Crie um gráfico direcionado para representar a MST
MST = nx.Graph()

for u, v, w in mst:
    MST.add_edge(u, v, weight=w)
# Posições dos vértices no gráfico
pos = nx.spring_layout(G)

# Desenhe o grafo original
plt.figure(figsize=(10, 5))
nx.draw(G, pos, with_labels=True, font_weight='bold', node_size=700,
        node_color='lightblue')
labels = nx.get_edge_attributes(G, 'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)

# Desenhe a árvore de abrangência mínima
nx.draw(MST, pos, edge_color='red', width=2, node_size=500,
        node_color='lightgreen')

plt.title("Grafo Original e Árvore de Abrangência Mínima (Kruskal)")
plt.show()
```

Grafo gerado pelo exemplo:



Algoritmo de Prim:

1. **Tipo de Grafo:** Funciona apenas em grafos não direcionados e ponderados que são conectados. Se o grafo não for conexo, você deve executar o algoritmo em cada componente conexo separadamente.
2. **Seleção de Arestas:** Começa com um vértice inicial e, em cada passo, adiciona a aresta de menor peso que conecta o conjunto de vértices já incluídos àqueles que ainda não fazem parte da MST. Não cria árvores separadas, a medida que vai iterando, as arestas são agregadas a árvore principal.
3. **Estrutura de Dados:** Pode ser implementado com uma matriz de adjacência (mais eficiente) ou uma lista de adjacência para o grafo. Não é necessário um conjunto disjunto.
4. **Complexidade:** A complexidade de tempo do algoritmo de Prim é tipicamente $O(V^2)$ para uma matriz de adjacência ou O(E + V * log(V)) para uma lista de adjacência, onde E é o número de arestas e V é o número de vértices.

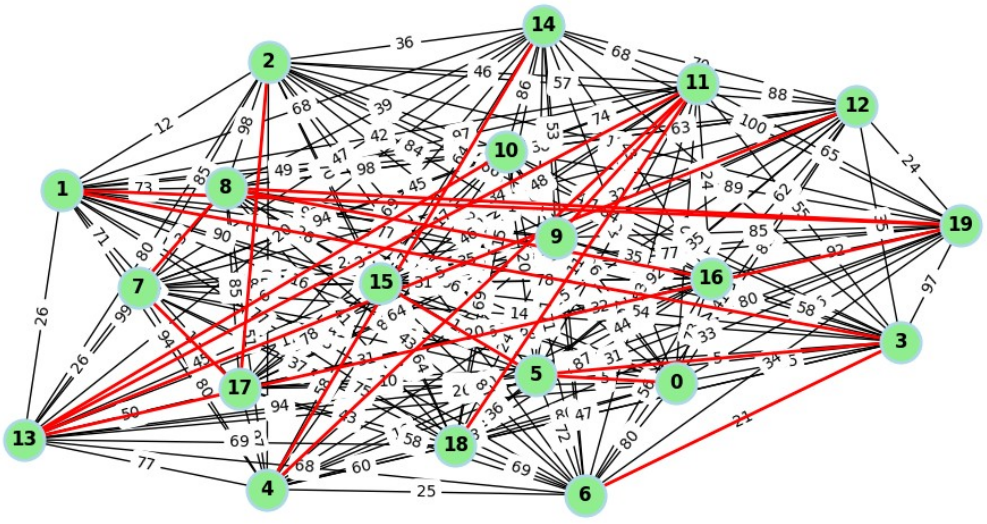
Implementação do Algoritmo

```python
import sys
import matplotlib.pyplot as plt
import networkx as nx
import time
# Essa função me retorna o vértice com a menor chave que ainda não está na
# árvore
def min_chave(chave, mst_set):
    min_valor = sys.maxsize
    min_indice = -1
    # nessa iteração a chave é ordenada do menor para o maior e retorna o menor
    # dos valores do peso
    for v in range(len(chave)):
        # verifica se a chave do vértice "v" é menor do que a menor chave
        # encontrada até o momento a outra condição verifica se o vértice "v"
        #não está na árvore, é o controle dos valores que ja foram encontrados
        if chave[v] < min_valor and not mst_set[v]:
            min_valor = chave[v]
            # atualiza o índice do vértice, isso permite que o algoritmo
            # acompanhe qual vértice a menor chave
            min_indice = v

    return min_indice
```

```python
def prim(grafo):
    V = len(grafo)
    # sys.maxsize é um valor que representa o maior valor inteiro que o sistema
    # pode representar Isso por que não se sabe de início do algoritmo o valor
    # do custo mínimo, conforme o algoritmo progride, os valores em chave serão
    # atualizados com os custos mínimos à medida que os vértices
    # são adicionados à árvore.
    chave = [sys.maxsize] * V
    chave[0] = 0 # aqui definimos a chave do vértice de origem como 0
    mst_set = [False] * V
    pai = [-1] * V
    # a complexidade de para matriz de adjacência e v² por que o algoritmo
    # percorre a a matriz inteira
    for a in range(V):
        # encontraremos o vértice com menor a chave
        u = min_chave(chave, mst_set)
        # marcamos o vértice com incluído na árvore
        mst_set[u] = True
        # aqui ao percorrer a coluna o vetor chave é preenchido com os valores
        # dos peços continos no garfo fornecido
        for v in range(V):
            if grafo[u][v] > 0 and not mst_set[v] and chave[v] > grafo[u][v]:
                chave[v] = grafo[u][v] # atualiza a chave de v
                pai[v] = u # define o "pai" de v na árvore como u

    return pai
grafo = [[0, 11, 12, 0, 0, 0],
         [11, 0, 1, 12, 0, 0],
         [12, 1, 0, 0, 11, 0],
         [0, 12, 0, 0, 7, 19],
         [0, 0, 11, 7, 0, 4],
         [0, 0, 0, 19, 4, 0]]

start_time = time.time()
pai = prim(grafo)
end_time = time.time()

print("Arestas no MST (Árvore de Abrangência Mínima):")
for i in range(1, len(pai)):
    print(f"Aresta: {pai[i]} - {i}, Peso: {grafo[i][pai[i]]}")
print(end_time-start_time)

G = nx.Graph()

arestas = []
num_vertices = len(grafo)
for u in range(num_vertices):
    for v in range(u + 1, num_vertices):
        peso = grafo[u][v]
        if peso > 0:
            arestas.append((u, v, peso))
```

```python
for u, v, w in arestas:
    G.add_edge(u, v, weight=w)

# Crie um gráfico direcionado para representar a MST
MST = nx.Graph()

for i in range(1,len(pai)):
    MST.add_edge(pai[i], i, weight=grafo[i][pai[i]])

# Posições dos vértices no gráfico
pos = nx.spring_layout(G)

# Desenhe o grafo original
plt.figure(figsize=(10, 5))
nx.draw(G, pos, with_labels=True, font_weight='bold', node_size=700,
        node_color='lightblue')
labels = nx.get_edge_attributes(G, 'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)

# Desenhe a árvore de abrangência mínima
nx.draw(MST, pos, edge_color='red', width=2, node_size=500,
        node_color='lightgreen')

plt.title("Grafo Original e Árvore de Abrangência Mínima (Kruskal)")
plt.show()
```

Grafo gerado pelo exemplo:

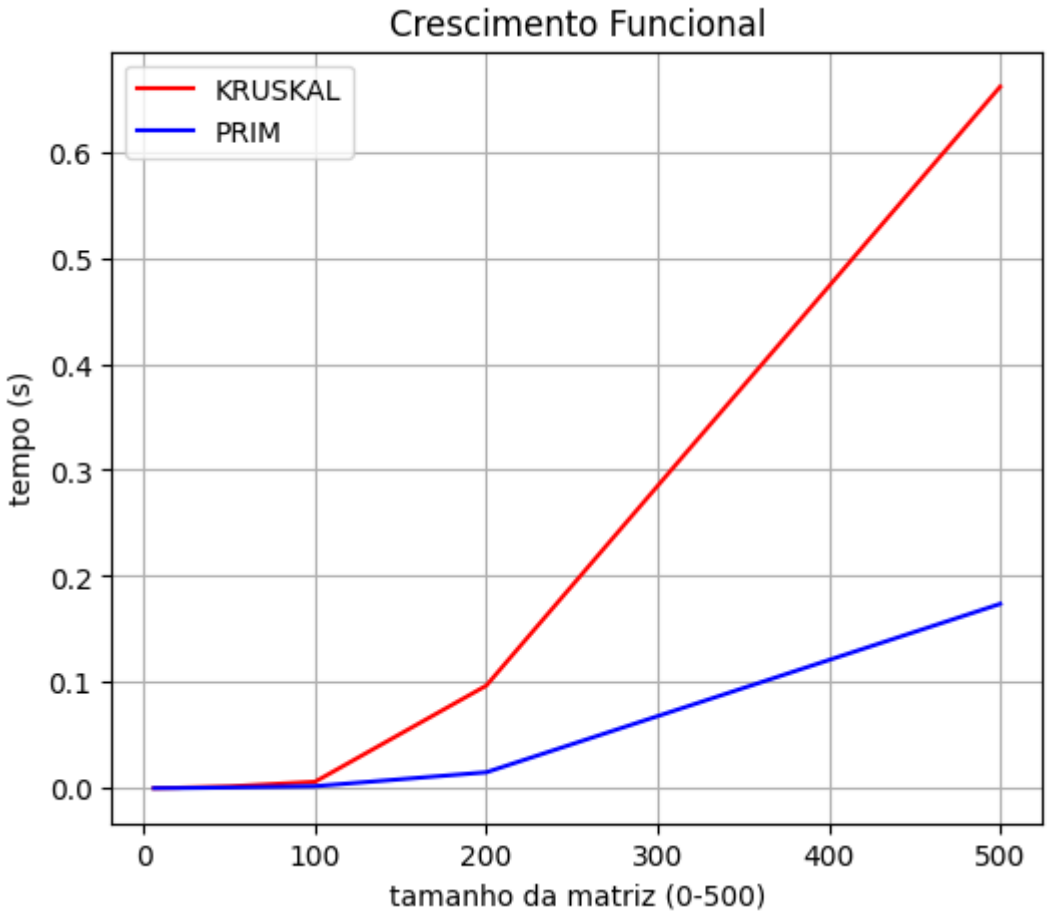


Comparação Geral:

- O algoritmo de Kruskal é mais versátil, pois pode ser aplicado a uma variedade de tipos de grafos, incluindo grafos desconectados, enquanto o algoritmo de Prim requer um grafo conectado.

- O algoritmo de Prim é geralmente mais eficiente em termos de tempo quando você tem um grafo denso, porque a complexidade do Kruskal depende do número de arestas.
- O algoritmo de Kruskal pode ser mais eficiente em termos de espaço quando você tem um grafo esparsamente conectado, pois ele não exige armazenar uma matriz completa de adjacência.
- A escolha entre Kruskal e Prim depende das características do seu grafo e das restrições do problema que você está resolvendo. Cada algoritmo tem seus próprios pontos fortes e fracos.

Comparação Numerica:



A partir do gráfico podemos verificar que de fato, a medida que o tamanho do vetor de entrada cresce de tamanho, ou seja, a rede se torna mais densa, torna-se evidante que o algoritmo de Kruskal apresenta um comportamento menos eficiente em comparado ao de Prim com relação ao tempo de processamento.

Comparação Final

Prim:

- Finalidade: Encontra a árvore de abrangência mínima de um grafo ponderado conectado (uma subestrutura que conecta todos os nós do grafo com o menor custo possível).
- Aplicação típica: Usado em problemas de rede, design de circuitos, otimização de rotas, etc.

- Restrição: Pode ser usado em grafos ponderados com arestas positivas ou negativas.
- Estrutura de saída: A árvore de abrangência mínima (conjunto de arestas que conectam todos os nós com o menor custo).

Kruskal:

- Finalidade: Encontra a árvore de abrangência mínima de um grafo ponderado, mas é aplicável a qualquer grafo (não necessariamente conectado).
- Aplicação típica: Usado em problemas de design de redes, logística, otimização de cabos, entre outros.
- Restrição: Pode ser usado em grafos ponderados com arestas positivas ou negativas.
- Estrutura de saída: A árvore de abrangência mínima (conjunto de arestas que conectam todos os nós com o menor custo).

3. Implemente o algoritmo do Dijkstra e utilize-o para resolver um problema prático da sua área de interesse.

Teoria:

O algoritmo de Dijkstra é um algoritmo usado para encontrar o caminho mais curto em um grafo ponderado com arestas não negativas. É amplamente utilizado em problemas de roteamento, como encontrar o caminho mais curto entre duas cidades em um mapa rodoviário, encontrar a menor latência em uma rede de computadores ou até mesmo em otimização de rotas em logística.

Abaixo está uma passo a passo do funcionamento do algoritmo de Dijkstra:

1. **Inicialização**: O algoritmo começa a partir de um nó de origem. Ele define a distância da origem para si mesma como zero e todas as outras distâncias como infinito (ou um valor muito grande), similiar ao que é feito com o algoritmo de Prim.
2. **Seleção do Vértice**: Escolhendo o nó com a distância mínima (o nó mais próximo da origem que ainda não foi visitado).
3. **Relaxamento**: Para todos os vizinhos do nó escolhido, o algoritmo atualiza a distância se o caminho passando por esse nó for mais curto do que o atualmente conhecido. Isso envolve comparar a distância atual do nó vizinho com a soma da distância do nó escolhido e o peso da aresta entre eles. Se for mais curto, a distância é atualizada.
4. **Marcação**: É marcado o nó escolhido como visitado para garantir que ele não seja escolhido novamente.
5. **Repetição**: Os passos 2 a 4 são repetidos até que todos os nós tenham sido visitados ou até que o nó de destino seja alcançado.
6. **Resultado**: Quando todos os nós tiverem sido visitados ou o nó de destino for alcançado, teremos a menor distância a partir da origem para todos os outros nós no grafo.

Vantagens do Algoritmo de Dijkstra:

- Encontra o caminho mais curto em um grafo ponderado.
- Pode ser usado para encontrar a menor latência em redes de computadores.
- Útil em problemas de roteamento e otimização em logística.

Desvantagens do Algoritmo de Dijkstra:

- Requer que todas as arestas tenham pesos não negativos.
- Não funciona corretamente com arestas de peso negativo, pois pode entrar em loops infinitos.
- Requer uma quantidade significativa de recursos computacionais em grafos grandes.

Algoritmo Implementado

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

# Grafo representando a rede de roteadores
# Cada valor no grafo representa a latência (atraso) entre os roteadores.
# Um valor de 0 indica que os roteadores estão diretamente conectados.
# Um valor infinito (float('inf')) indica que os roteadores não estão
# diretamente conectados.

grafo_rede = [
    [0, 5, 2, float('inf'), float('inf')], # Roteador 0
    [5, 0, 0, 2, float('inf')],            # Roteador 1
    [2, 0, 0, 3, 6],                       # Roteador 2
    [float('inf'), 2, 3, 0, 2],            # Roteador 3
    [float('inf'), float('inf'), 6, 2, 0]  # Roteador 4
]
origem = 0  # Vamos partir do Roteador 0
destino = 3  # Queremos chegar ao Roteador 3
def dijkstra(grafo, origem):
    # Verifica o numero de vértices
    num_roteadores = len(grafo)
    # inicializa um vetor com valores infinitos e tamanho igual quantidade de
    # roteadores
    distancias = [float('inf')] * num_roteadores
    # marca a origem com valor 0
    distancias[origem] = 0
    # innicializa um vetor para armazenar o caminhos já visitados
    visitados = [False] * num_roteadores
    # percorre cada linha do grafo
    for x in range(num_roteadores):
        u = encontrar_vertice_minimo(distancias, visitados)
        visitados[u] = True
        # verifica se a condição para relaxar a aresta do vértice u para o
        # vértice v é atendida. Essa condição é parte essencial do algoritmo de
        # Dijkstra e é usada para determinar se podemos encontrar um caminho
        # mais curto
        # passando pelo vértice u para chegar ao vértice v.
        for v in range(num_roteadores):
            if (not visitados[v] and grafo[u][v] > 0 and distancias[u] + grafo[u][v] < distancias[v]):
                # not visitados[v]: Isso verifica se o vértice v ainda não foi
                # visitado
                # grafo[u][v] > 0: Isso verifica se há uma aresta (ligação)
                # direta entre os vértices u e v (aresta válida)
                # distancias[u] + grafo[u][v] < distancias[v]: verifica se a
                # distância do vértice de origem (no caso,o vértice u) até o
                #vértice v através do vértice u é menor do que a distância
                # atualmente conhecida para v. Se isso for verdade, significa
                # que encontramos um caminho mais curto para v passando por u,
                # e atualizamos
                # a distância mínima para v.
                distancias[v] = distancias[u] + grafo[u][v]
    return distancias
```

```python
def encontrar_vertice_minimo(distancias, visitados):
    minimo = float('inf')
    minimo_indice = -1
    # varre o vetor distancia para encontar o valor mínimo
    for v in range(len(distancias)):
        if distancias[v] < minimo and not visitados[v]:
            # se valor percorrido for menor armazena o seu valor e índice
            minimo = distancias[v]
            minimo_indice = v
    return minimo_indice

inicio = time.time()
distancias = dijkstra(grafo_rede, origem)
fim = time.time()
print(f"Rota mais curta do Roteador {origem} para o Roteador {destino}:
      {distancias[destino]} unidades de latência")
print(fim - inicio)

G = nx.Graph()
for i in range(len(grafo_rede)):
    G.add_node(i)

for i in range(len(grafo_rede)):
    for j in range(len(grafo_rede[i])):
        if grafo_rede[i][j] != float('inf'):
            G.add_edge(i, j, weight=grafo_rede[i][j])

pos = nx.spring_layout(G)
nx.draw(G, pos, with_labels=True, node_size=800, node_color='lightblue',
        font_size=10, font_color='black', font_weight='bold')
labels = nx.get_edge_attributes(G, 'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
plt.title("Rede de Roteadores com Rotas Mais Curtas")
plt.show()
```

Grafo do exemplo

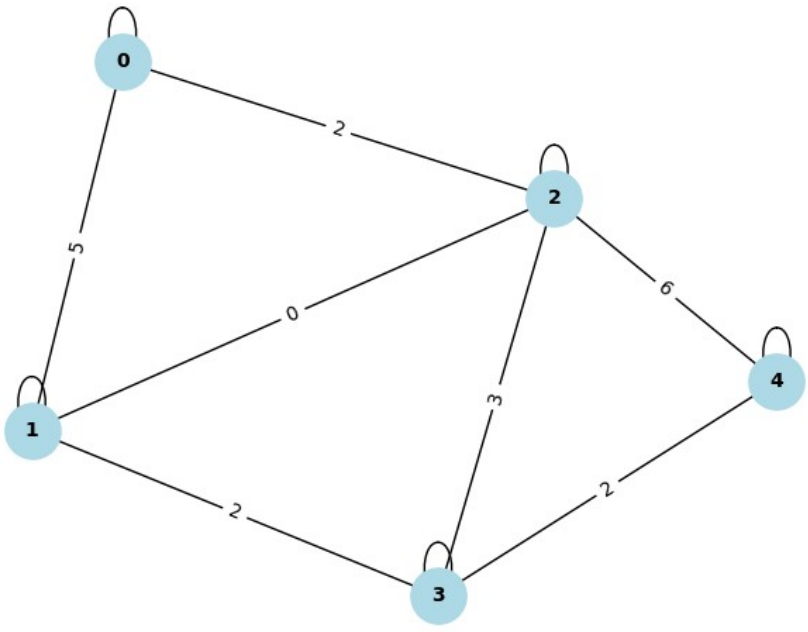
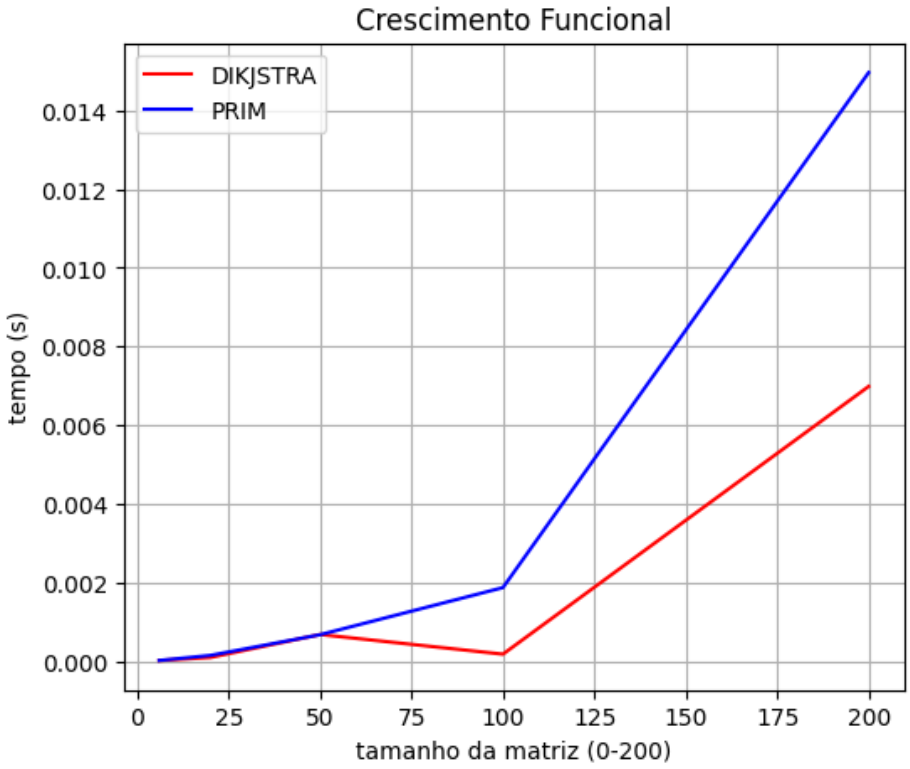


Gráfico de Aplicação

Foi feito uma levantamento de performance com relação ao tempo em função do tamanho da entrada da matriz de adjacência, onde podemos verificar que Dikjstra e bem mais rápido, no entanto esse comparativo nem deveria ser levado em consideração visto que os algoritmos em questão tem aplicação e e objetivos diferentes.



Considerações finais

Em resumo, o algoritmo de Dijkstra é usado para encontrar o caminho mais curto em um grafo a partir de um único nó de origem, enquanto os algoritmos de Prim e Kruskal são usados para encontrar a árvore de abrangência mínima de um grafo, conectando todos os nós do grafo com o menor custo possível. Dijkstra é mais focado em distâncias mínimas, enquanto Prim e Kruskal estão mais preocupados em criar uma estrutura de árvore com um custo mínimo.