INSTITUTO FEDERAL DE MINAS GERAIS
CAMPUS OURO BRANCO
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

MARCUS VINÍCIUS RIBEIRO ANDRADE

CARLOS HENRIQUE CENACHI

ALGORITMO DE KRUSKAL

Ouro Branco, MG

2023

Marcus Vinícius Ribeiro Andrade
Carlos Henrique Cenachi

Algoritmo de kruskal

Professor: Ângelo Magno

Ouro Branco, MG

2023

RESUMO

Este projeto, conta com a implementação do algoritmo de Kruskal para obtenção da Árvore Geradora Mínima de um grafo. A visualização gráfica do grafo original e da árvore geradora mínima foi integrada a uma aplicação web usando o micro-framework python Flask.

Sumariamente, os passos principais incluem a leitura de uma matriz a partir de um arquivo de texto, a geração visual do grafo original, a implementação do algoritmo de Kruskal para obter a árvore mínima e a visualização desta árvore. Tudo isso foi integrado em um aplicativo Flask, com rotas específicas para exibir um formulário de upload de arquivo, processar o arquivo enviado pelo usuário e apresentar as visualizações geradas.

A página web resultante utiliza Bootstrap para criar um layout responsivo, garantindo uma apresentação visual agradável, mesmo em dispositivos móveis. Desse modo, serve, não apenas como validação no estudo, mas também como forma visual da aplicação do algoritmo.

Palavras-chaves: kruskal, algoritmo, grafo.

ABSTRACT

This project relies on the implementation of the Kruskal algorithm to obtain the Minimum Spanning Tree of a graph. The graphical visualization of the original graph and the minimum spanning tree was integrated into a web application using the Flask python micro-framework.

Briefly, the main steps include reading a matrix from a text file, visually generating the original graph, implementing the Kruskal algorithm to obtain the minimum tree, and visualizing this tree. All of this was integrated into a Flask application, with specific routes to display a file upload form, process the user-uploaded file, and present the generated previews.

The resulting web page uses Bootstrap to create a responsive layout, ensuring a pleasant visual presentation, even on mobile devices. In this way, it serves not only as validation in the study, but also as a visual way of applying the algorithm.

Keywords: kruskal, algorithm, graph.

SUMARIO

1 INTRODUÇÃO

O algoritmo de Kruskal, concebido por Joseph Kruskal em 1956, é um algoritmo do tipo greedy e é bastante eficiente para encontrar Árvores Geradoras Mínimas (AGMs) em grafos ponderados. Sua abordagem, portanto, consiste em ordenar as arestas do grafo por peso e adicionar, iterativamente, aquelas de menor custo, assegurando que não formem ciclos.

Diante disso, A importância do algoritmo se destaca em sua aplicabilidade prática, sendo empregado em diversas áreas. Por exemplo, em redes de comunicação, otimiza custos para estabelecer conexões eficientes; em logística, minimiza custos de transporte.

Além disso, o algoritmo pode ser aplicado em design de circuitos, redes de computadores e análises genéticas para construir árvores filogenéticas. Sua simplicidade e eficiência fazem dele uma escolha valiosa na resolução de problemas práticos, contribuindo para a otimização de recursos em diversas áreas da ciência da computação e engenharia.

Assim, dado a importância é aplicabilidade dessa ferramenta, o presente trabalho irá discorrer sobre o uso do algoritmo em uma matriz enviada, como um arquivo de texto, para que possa ser exibido a árvore geradora mínima resultante da aplicação do método de kruskal, sendo, de fato, feito a partir da popular linguagem de programação Python.

2. OBJETIVO

Este trabalho explora o uso do algoritmo de Kruskal em uma matriz fornecida por meio de um arquivo de texto. O objetivo é apresentar a visualização da árvore geradora mínima resultante da aplicação do método de Kruskal, implementado na linguagem de programação Python.

Desse modo, é esperado que se possa elucidar melhor a formação do grafo a partir de uma matriz e, portanto, evidenciar melhor, através da representação gráfica, a árvore resultante.

3. O ALGORITMO

No algoritmo de Kruskal, a árvore geradora inicial contém apenas os nós de o gráfico e não contém arestas. Então o algoritmo passa pelo arestas ordenadas por seus pesos e sempre adiciona uma aresta à árvore se isso não acontecer crie um ciclo.O algoritmo mantém os componentes da árvore. Inicialmente, cada nó de o gráfico pertence a um componente separado. Sempre quando uma aresta é adicionada ao árvore, dois componentes são unidos.

Finalmente, todos os nós pertencem ao mesmo componente,e uma árvore geradora mínima foi encontrada (Laaksonen .A).

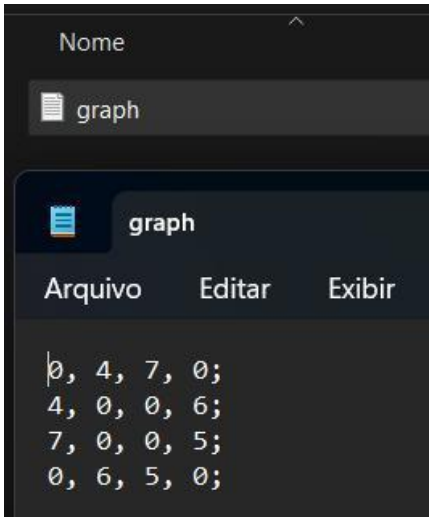
Com base nisso, o código, todo escrito em Python, necessário para desenvolver o algoritmo será apresentado abaixo:

3.1.1 Ler a matriz (input):

```
def read_graph_from_file(filename):
    with open(filename, "r") as file:
        lines = file.readlines()

    graph = [list(map(int, line.strip().replace(";", ""
        ).split(","))) for line in lines]
    return graph
```

O primeiro passo é ler a matriz, que está em um arquivo de texto mostrado abaixo, a função, portanto, lê o conteúdo da matriz como uma lista de strings. Feito isso, a variável graph recebe uma lista de listas com os conteúdos da lista anterior. Ou seja, será feito sublistas com os números (convertidos de texto para inteiro), que serão combinados, pela função map, ficando no formato [[1, 0, 2], [3, 4, 5]], por exemplo.



3.1.2 Kruskal:

```
def kruskal_mst(graph):
    plt.close()

    G = nx.Graph()
    sum = 0
    edges = []

    for i in range(len(graph)):
        for j in range(i + 1, len(graph[i])):
            if graph[i][j] != 0:
                w = graph[i][j]
                edges.append((i, j, w))
```

Inicialmente, o algoritmo recebe o grafo carregado como matriz, com base no supracitado. Feito isso, é criado uma instância de Graph, do módulo networkx, a variável sum, que será responsável por exibir o peso final, uma lista para receber as arestas.

Então, é percorrido um loop para alimentar o novo grafo, ignorando os locais sem conexão, ou seja, valores com zero.

```
edges.sort(key=lambda x: x[2])

parent = [i for i in range(len(graph))]
rank = [0] * len(graph)
```

Com isso feito, é ordenado por pesos as arestas do grafo, visto que, é necessário escolher os menores caminhos.

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

A função interna find, procura pelo o que será a raíz da árvore, assim, procura pelos pais de cada nó na árvore.

```
def union(x, y):
    root_x = find(x)
    root_y = find(y)

    if root_x != root_y:
        if rank[root_x] < rank[root_y]:
            parent[root_x] = root_y
        elif rank[root_x] > rank[root_y]:
            parent[root_y] = root_x
        else:
            parent[root_x] = root_y
            rank[root_y] += 1
        return True
    # forma ciclo:
    return False
```

A função interna union é conhecida como union-find, basicamente, é um técnica que auxilia na otimização do algoritmo, pois, opera sobre dois conjuntos disjuntos, ou seja, que os elementos só pertencem a um único conjunto (Laaksonen .A).

Assim, verifica se o elemento é diferente, ou seja, já não está no conjunto. Se estiver, indica que formará ciclo e, portanto, o retorno é falso. Se não, verifica se o grau de um é maior que o outro, se for o caso, é realizado a troca. Se não, une os conjuntos formando uma única árvore.

Isso, entretanto, é feito pelas atualizações do parent e do rank, variáveis declaradas acima que irão definir se forma ciclo ou se já usou determinado vértice.

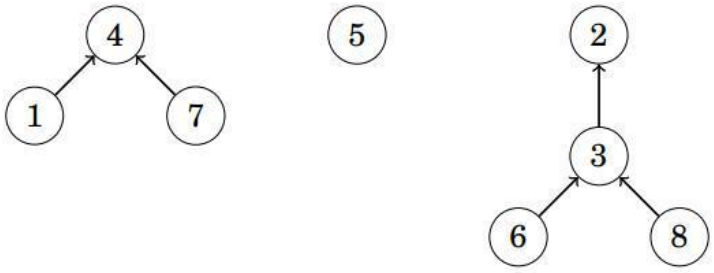
```
for edge in edges:
    u, v, weight = edge
    if union(u, v):
        G.add_edge(u, v, weight=weight)
        sum += weight

with open("static/weight.txt", "w") as f:
    f.write(str(sum))

pos = nx.spring_layout(G)
labels = {(u, v): weight for u, v, weight in G.edges(data
    ="weight")}
```

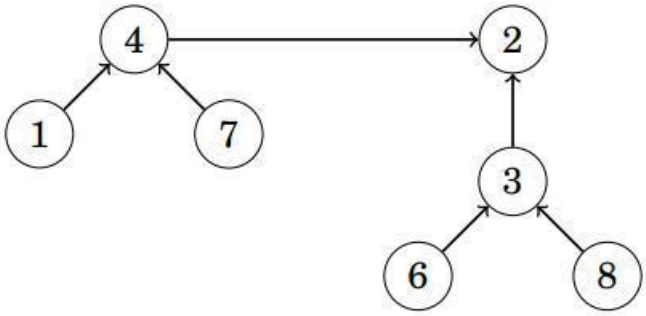
Acima, têm-se então a implementação do union, onde será percorrido as arestas do grafo, contendo a posição e o peso, se for possível realizar a união, será adicionado ao novo grafo e será acumulado a soma (para exibir depois), dos pesos das arestas.

O arquivo weight.txt criado contém o peso total e será lido pelo flask depois. Assim, as variáveis pos é responsável por fazer a ligação do grafo(através da biblioteca networkx-responsável por manipular e estruturar redes), pelo algoritmo de Fruchterman-Reingold (via biblioteca mesmo), e a variável label apenas cria um dicionário (map) que tem como chave as posições e valor o peso, para ser aplicado no desenho do grafo posteriormente (Laaksonen .A).



[2]The structure presented here was introduced in 1971 by J. D. Hopcroft and J. D. Ullman [38]. Later, in 1975, R. E. Tarjan studied a more sophisticated variant of the structure [64] that is discussed in many algorithm textbooks nowadays.

Acima, há um esquema do union find para melhor entendimento.


4
2
1
7
3
6
8

3.1 A PARTE GRÁFICA

Para representar, graficamente, foi utilizado a biblioteca de representação gráfica pyplot, do módulo matplotlib e, como supracitado, a biblioteca networkx. A primeira é responsável por fornece funções semelhantes às utilizadas no MATLAB para a criação de gráficos. Assim, permite criar gráficos e formas de visualização bem rapidamente.

Agora, a networkx fornece ferramentas para modelar, analisar e visualizar a estrutura de redes, incluindo grafos direcionados e não direcionados.

Assim, como ênfase, é necessário instalá-las, para isso, com o python3 e o pip (gerenciador de pacotes python) já instalados no computador, basta fazer os comandos:

- ***python -m pip install -U pip***
- ***python -m pip install -U matplotlib***
- ***pip install networkx***

E por fim, para *executar* o trabalho, basta fazer:

- ***python main.py***

Enfim, quanto a implementação da parte gráfica, primeiramente, foi necessário criar duas variáveis para servir de ponteiro, pois, segundo o professor, seria importante que a árvore geradora possuísse o mesmo formato que o grafo original. Assim, essas variáveis conservam o estado do grafo original.

```
ORIGINAL = None
GLOBAL_POS = None
```

```
def visualize_graph(graph):
    global ORIGINAL, GLOBAL_POS

    G = nx.Graph()
    ORIGINAL = G
    for i in range(len(graph)):
        for j in range(i + 1, len(graph[i])):
            if graph[i][j] != 0:
                G.add_edge(i, j, weight=graph[i][j])

    pos = nx.spring_layout(G)
    GLOBAL_POS = pos
    labels = {(i, j): graph[i][j] for i, j, _ in G.edges(data=True)}

    nx.draw(
        G,
        pos,
        with_labels=True,
        labels={i: str(i) for i in G.nodes()},
        node_size=724,
        node_color="skyblue",
        font_size=16,
    )
    nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
    plt.title("Graph")
    plt.savefig("static/image.jpg")
```

Acima, tem-se a função de visualização, que é responsável por criar e um gráfico de um grafo não direcionado. Logo, as bibliotecas networkx e matplotlib.pyplot são utilizadas para realizar essa tarefa. No início, ela cria um objeto de grafo não direcionado usando networkx e itera sobre os nós do grafo original. Para cada par de nós conectados por uma aresta no grafo original, uma aresta correspondente é adicionada ao grafo criado.

A função utiliza um layout para posicionar os nós de forma que as forças entre eles sejam minimizadas. Isso é feito através da função nx.spring_layout. Em seguida, a posição global desse layout é armazenada para uso posterior. Além disso, rótulos são criados para as arestas com base nos pesos.

O grafo é então desenhado usando nx.draw, adicionando rótulos aos nós e às arestas, ajustando o tamanho e a cor dos nós, e finalmente, adicionando rótulos de peso às arestas com nx.draw_networkx_edge_labels. Um título é atribuído ao gráfico usando plt.title.

A visualização do gráfico é salva como uma imagem no diretório "static" por meio de plt.savefig. Isso pois, permite que a imagem seja posteriormente utilizada e exibida por um aplicativo Flask em uma interface web.

Além disso, a função que desenha a árvore geradora mínima e que, de fato, usa as variáveis referenciadas acima é:

```
labels = {(u, v): weight for u, v, weight in G.edges(data='weight')}

nx.draw(G, GLOBAL_POS, with_labels=True, labels={i: str(i) for i in G.nodes()}, node_size=724, node_color="lightgreen",
        font_size=16)
nx.draw_networkx_edge_labels(G, GLOBAL_POS, edge_labels=labels)
plt.title("Minimum Spanning Tree")
# plt.show()
plt.savefig("static/im.jpg")
```

Ou seja, basicamente, faz o mesmo que a anterior, mas, reutiliza as posições do grafo original, em GLOBAL_POS, para desenhar a árvore geradora mínima e salvá-la como uma imagem a ser renderizada pelo flask.

3.2 RENDERIZAÇÃO COM FLASK

O Flask é um micro-framework web em Python que facilita a criação de aplicativos web. Ele é minimalista e flexível, permitindo o desenvolvimento rápido de aplicações web. Flask é utilizado para criar endpoints HTTP, gerenciar rotas e facilitar a interação entre o back-end e o front-end (Ronacher, A).

Diante disso, o framework foi utilizado para permitir uma certa facilidade para o usuário visualizar o gráfico inserido. Mesmo que, tenha que parar a aplicação para carregar um novo arquivo, devido o pyplot não poder trabalhar fora da thread main.

O script inicia importando as bibliotecas necessárias, como Flask para criar o aplicativo web, e funções personalizadas relacionadas ao processamento de gráficos. Além disso, a biblioteca 'os' é utilizada para lidar com operações relacionadas ao sistema operacional.

Em seguida, há uma mudança para o diretório do script, garantindo que o caminho do arquivo seja interpretado corretamente.

```
from flask import Flask, render_template, send_from_directory, request, redirect
from .graph import read_graph_from_file, visualize_graph, kruskal_mst
import os

os.chdir(os.path.dirname(os.path.abspath(__file__)))
app = Flask(__name__)
```

O objeto Flask é criado com o nome "name", indicando o nome do módulo atual. Esse objeto será usado para configurar e executar a aplicação web.

O código define três rotas principais usando o decorador @app.route:

'/': Esta rota mapeia para a função index(). Quando um usuário acessa a página inicial, a função index() renderiza o modelo 'form.html', para que o usuário insira a matriz desejada:

```
@app.route('/')
def index():
    return render_template('form.html')
```

```
<nav class="navbar navbar-primary bg-primary text-light p-3">
    <h1 class="font-monospace">Algoritmo de Kruskal</h1>
</nav>
<div class="container mt-5 d-flex justify-content-center">
    <form action="/upload" method="post"
          style="width: 30rem; height: 30rem; display:flex; justify-content:center; background:#cbd5e1;"
          class="card d-flex flex-column p-3 m-3" enctype="multipart/form-data">
        <input type="file" class="mb-3" name="file">
        <input class="btn btn-primary" type="submit" value="Upload">
    </form>
</div>
```

Imediatamente acima, está o código HTML que será renderizado. Para personalização, foi utilizado o framework Bootstrap4.

Agora, a rota '/upload', mapeia para a função index_post(). Quando um usuário envia a matriz através de um formulário na página inicial, esta rota é acionada. O arquivo é salvo na pasta 'input' e, em seguida, a página é redirecionada para a rota '/kruskal'.

```
@app.route('/upload', methods=["POST"])
def index_post():
    print(request.files)
    if "file" not in request.files:
        return
    file = request.files['file']
    file.save("input/" + file.filename)
    return redirect("/kruskal")
```

Por fim, a rota '/kruskal' mapeia para a função kruskal(), onde a matriz de grafo é lida, e é onde são chamados os métodos responsáveis por realizar o algoritmo de Kruskal para gerar a árvore geradora mínima. O peso dessa árvore é salvo em um arquivo 'weight.txt'. A página renderiza e exibe o resultado, incluindo o peso da árvore geradora mínima.

```
@app.route('/kruskal')
def kruskal():
    filename = "input/graph.txt"
    graph = read_graph_from_file(filename)
    visualize_graph(graph)
    kruskal_mst(graph)
    with open("static/weight.txt", 'r') as f:
        w = f.read()
    return render_template('index.html', weight=w)
```

4. CONCLUSÃO

Diante do exposto e considerando a implementação do algoritmo de Kruskal em conjunto com o framework Flask, permite a construção de uma aplicação web interativa para análise de árvores geradoras mínimas em grafos. O Flask proporciona um ambiente leve para desenvolvimento web em Python, permitindo a visualização das imagens geradas.

Ademais, a aplicação desenvolvida demonstra a capacidade de receber, processar e visualizar grafos enviados pelos usuários. A utilização do algoritmo de Kruskal possibilita a identificação da árvore geradora mínima, evidenciando sua aplicabilidade em problemas relacionados a redes, transporte e otimização.

Por fim, o projeto permite visualizar quaisquer grafos passados a partir de matrizes, de forma responsiva o que, portanto, pode auxiliar no estudo dessa importante temática.

REFERÊNCIAS

Laaksonen, A. (2018). **Competitive Programmer's Handbook** (Draft July 3, 2018).

Ronacher, A. (2010). Flask Documentation. Recuperado de https://flask.palletsprojects.com/en/2.1.x/