

UNIVERSIDADE FEDERAL DE CATALÃO – UFCAT
CIÊNCIAS DA COMPUTAÇÃO

RAFAEL BERNARDES DE JESUS
HENRIQUE FUGA GOMES

TRABALHO FINAL – AGRUPAMENTO COM ESTRATÉGIA GULOSA

CATALÃO
2024

RAFAEL BERNARDES DE JESUS
HENRIQUE FUGA GOMES

Trabalho Final – Agrupamento com estratégia gulosa

Trabalho avaliativo apresentado à Universidade Federal de Catalão a fim de obtenção de nota na componente Análise e Projeto de Algoritmos no curso de Ciências da Computação.
Professor: Dayse Silveira de Almeida.

CATALÃO
2024

RESUMO

O presente trabalho pretende realizar uma apresentação detalhada sobre o uso de algoritmos gulosos baseados nas estratégias de PRIM e KRUSKAL, amplamente divulgados na computação moderna, junto a duas implementações de algoritmos com a finalidade de realizar um agrupamento de pontos cartesianos. Será realizada uma análise assintótica dos algoritmos e a comparação com o agrupamento esperado. A problemática foi fornecida pelo docente da disciplina assim como a base de dados utilizada.

Palavras-chave: Prim. Kruskal. Agrupamento. Clustering. Árvore geradora mínima. AGM. MST.

ABSTRACT

This article aims to provide a detailed presentation about the use of greedy algorithms based on the PRIM and KRUSKAL strategies, widely recognized in modern computing, along with two algorithm implementations designed for clustering cartesian points. An asymptotic analysis of the algorithms will be conducted, and a comparison will be made with the expected clustering outcomes. The problem statement was provided by the course instructor, along with the database used for this study.

Keywords: Prim. Kruskal. Clustering. Minimum Spanning Tree. MST.

SUMÁRIO

1. INTRODUÇÃO	6
1.1 Árvore geradora mínima.....	6
1.2 Algoritmo de PRIM	7
1.3 Algoritmo de KRUSKAL.....	9
1.4 Estrutura HEAP	10
1.5 Union-Find	11
2. PROBLEMÁTICA.....	12
3. IMPLEMENTAÇÃO	13
3.1 Prim	13
3.2 Kruskal	20
3.3 Análise assintótica	25
4. ANÁLISE DE RESULTADOS	26
5. CONCLUSÕES.....	28

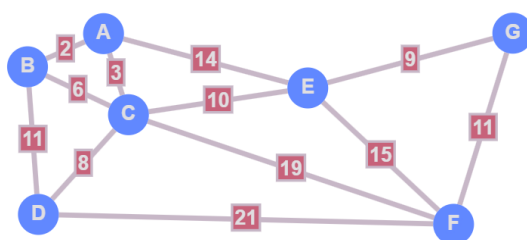
1. INTRODUÇÃO

Para o entendimento da problemática e dos métodos utilizados no trabalho, é necessário reforçar alguns conceitos, evidenciados a seguir.

1.1 Árvore geradora mínima

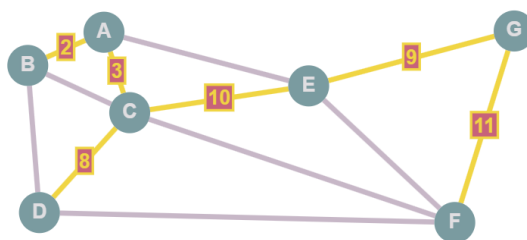
Em inglês, Minimum Spanning Tree (MST), a árvore geradora mínima de um grafo consiste no conjunto de arestas que conecta todos os vértices em que a soma de seus respectivos pesos seja a mínima possível. Logo, para extrair uma MST de um grafo, esse deve ser não-direcionado, conexo e ponderado. As figuras a seguir exemplificam a obtenção de uma MST em um grafo que obedece tais requisitos.

Figura 1 – Grafo de sete vértices



Fonte: graphonline¹

Figura 2 – MST do grafo da figura 1



Fonte: graphonline²

¹ Grafo modelado pelo autor. Disponível em: <<https://graphonline.ru/pt/?graph=RsYQAEvuGqhQMUMd>>. Acesso em: 29 set. 2024.

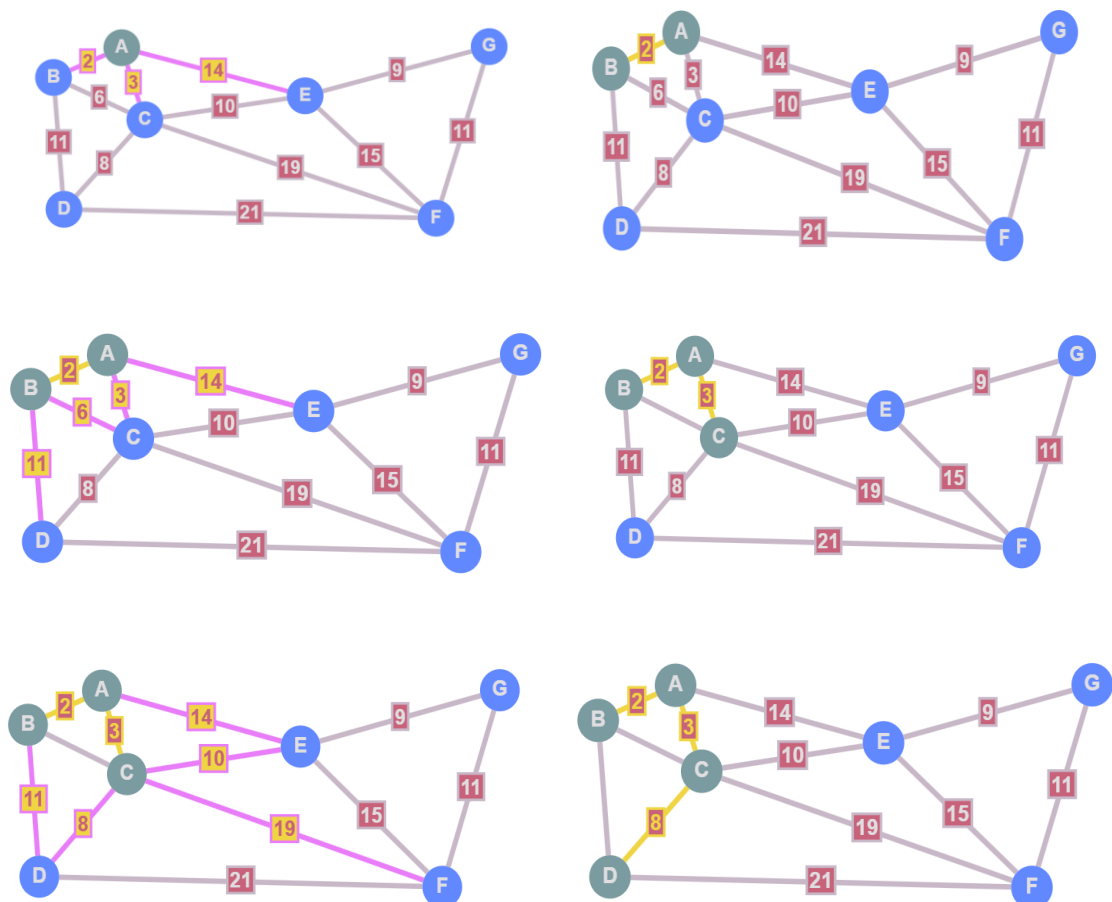
² Grafo modelado pelo autor. Disponível em: <<https://graphonline.ru/pt/?graph=RsYQAEvuGqhQMUMd>>. Acesso em: 29 set. 2024.

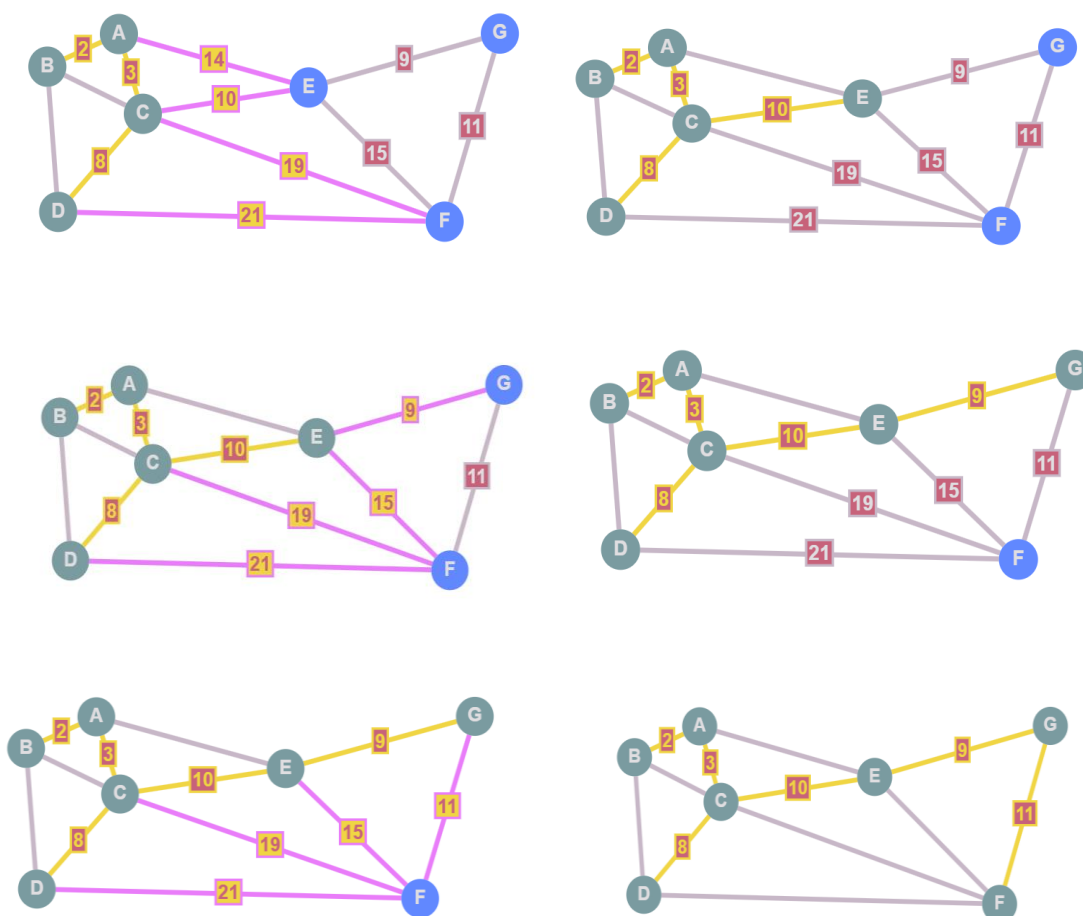
Existem diversas maneiras de se encontrar a árvore geradora mínima de um grafo ponderado, porém os meios mais veiculados na computação moderna são o algoritmo de Prim e o algoritmo de Kruskal, detalhados a seguir.

1.2 Algoritmo de PRIM

Robert Clay Prim (1921–2021) foi um matemático e engenheiro eletrônico norte-americano, desenvolvedor do Algoritmo de Prim. A principal ideia desse algoritmo guloso é construir uma árvore geradora mínima de forma gradual, a cada passo escolhendo a aresta de menor peso que expande a árvore sem formar ciclos até que todos os vértices tenham sido incluídos na árvore, formando assim uma árvore geradora mínima.

Figuras 3-14 – Representação do passo a passo do Algoritmo de Prim no grafo apresentado na Figura 1





Fonte: graphonline³

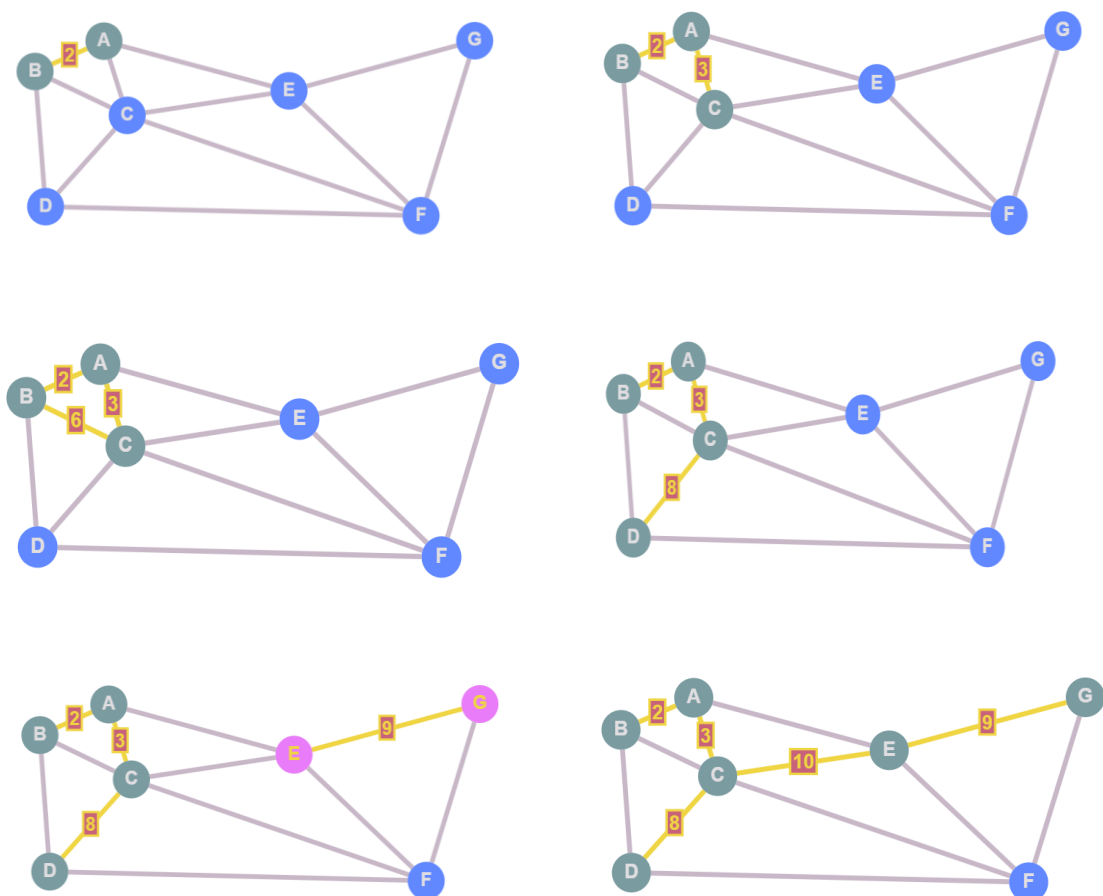
Como mostrado na sequência de figuras, para iniciar o processo do algoritmo de Prim, é selecionado um vértice qualquer. No caso citado, o vértice A foi selecionado (em verde, pois acaba de ser integrado à MST). Logo em seguida, todas as arestas conectadas a esse vértice são postas em uma fila de prioridade, organizada pelas arestas de menor peso (representadas em rosa). Assim, é escolhida a aresta de menor peso da fila que conecta um vértice já na árvore com um vértice fora dela. Essa aresta (agora representada em amarelo) é adicionada à árvore geradora e o novo vértice na árvore também é incluído. Posteriormente, as arestas conectadas ao novo vértice são adicionadas a fila, caso elas não criem ciclos. Esse processo é repetido até que todos os vértices estejam incluídos na árvore.

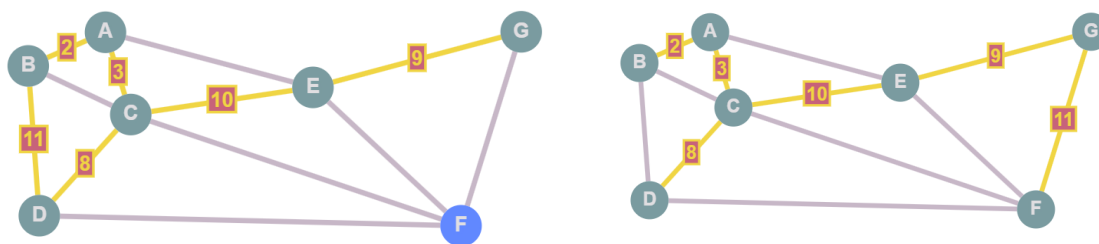
³ Grafos modelados pelo autor. Disponível em: <<https://graphonline.ru/pt/?graph=RsYQAEvuGqhQMUMd>>. Acesso em: 29 set. 2024.

1.3 Algoritmo de KRUSKAL

O mentor do algoritmo de Kruskal foi o cientista da computação, estatístico e norte-americano Joseph Kruskal (1928–2010). O algoritmo de Kruskal encontra a árvore geradora mínima em um grafo selecionando suas arestas em ordem crescente de peso e adicionando-as à árvore geradora, desde que não formem ciclos, também é uma abordagem gulosa.

Figuras 15-22 – Representação do passo a passo do Algoritmo de Kruskal no grafo apresentado na Figura 1





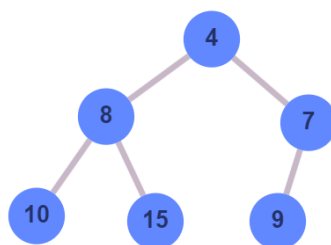
Fonte: graphonline⁴

No algoritmo de Kruskal, o primeiro passo é ordenar todas as arestas do grafo em ordem crescente de peso. Em seguida, as arestas são examinadas uma a uma. A cada iteração, o algoritmo verifica se a adição da aresta vai formar um ciclo (ligar um vértice a outro quando os ambos já estão formando um grupo) ou não. Se não formar um ciclo, a aresta é adicionada à árvore (ou a um grupo, que eventualmente será integrado à árvore), caso contrário, a aresta é descartada. Na sequência mostrada acima, é denotada a montagem da árvore aresta por aresta até todos os vértices serem integrados.

1.4 Estrutura HEAP

A estrutura heap é um tipo de estrutura de dados eficiente na criação de filas de prioridade. Baseada em árvore binária, é dividida em dois tipos principais: Min-heap, na qual a chave de cada nó é menor ou igual às chaves de seus filhos e o menor valor está sempre na raiz da árvore; Max-heap, na qual a chave de cada nó é maior ou igual às chaves de seus filhos e o maior valor está sempre na raiz da árvore.

Figura 23 – Exemplo de Min-heap



Fonte: graphonline⁵

⁴ Grafos modelados pelo autor. Disponível em: <<https://graphonline.ru/pt/?graph=RsYQAEvuGqhQMUMd>>. Acesso em: 29 set. 2024.

⁵ Grafo modelado pelo autor. Disponível em: <<http://graphonline.ru/pt/?graph=MoSEiflmiTlAseB>>. Acesso em: 29 set. 2024.

O uso de heap é bastante efetivo ao ser implantado como uma fila de prioridade, uma vez que em uma heap bem implementada, ao armazenar um conjunto de valores com tal estrutura, sempre se sabe que o valor da raiz é o maior/menor (dependendo do tipo de heap utilizado) de todo conjunto e a remoção ou adição de itens não altera tal estrutura.

Será visto que o uso de min-heap é útil no contexto da implementação do algoritmo de Prim.

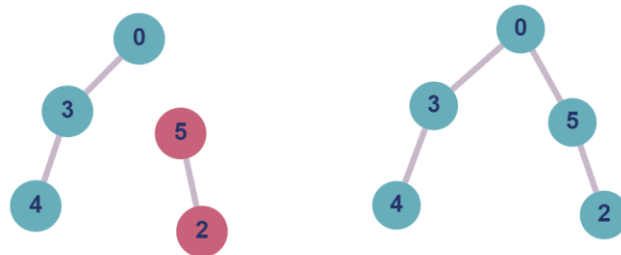
1.5 Union-Find

Também conhecida como Disjoint Set Union (DSU), Union-Find é uma estrutura de dados usada para gerenciar a união e a consulta de conjuntos disjuntos de elementos, isto é, para manter e rastrear subconjuntos que não têm interseções. É constituída de duas operações principais, explicadas a seguir.

A operação **FIND** é utilizada para encontrar o representante ou a raiz do conjunto ao qual um elemento pertence. Isso permite verificar se dois elementos pertencem ao mesmo conjunto.

A operação **UNION** combina dois conjuntos em um único conjunto. Se dois elementos pertencem a conjuntos diferentes, a união de seus conjuntos é feita ao unir seus representantes.

Figura 24 – Exemplo de conjuntos disjuntos após operação union



Fonte: graphonline⁶

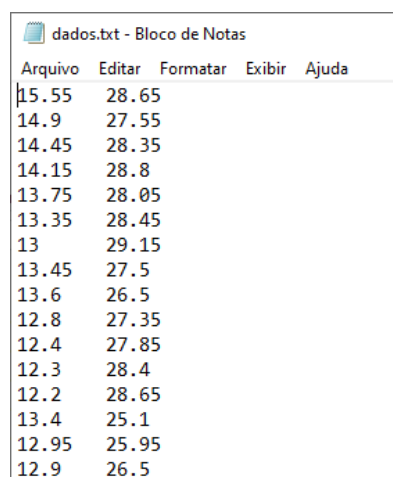
⁶ Grafo modelado pelo autor. Disponível em: < <http://graphonline.ru/pt/?graph=lsThIohcgiOEwKH> >. Acesso em: 29 set. 2024.

2. PROBLEMÁTICA

O objetivo deste projeto é realizar a implementação de dois algoritmos de agrupamento de dados utilizando árvores geradoras mínimas, um deles utilizando os conceitos de Prim, e outro de Kruskal, ambos explicados previamente.

Para avaliar e fomentar a geração dos códigos, foi fornecida uma base de dados com um total de 788 pontos armazenados no arquivo texto “dados.txt”. A primeira coluna representa a coordenada X e a segunda coluna a coordenada Y em um plano cartesiano. Além disso, também foi fornecida a ilustração cartesiana da base de dados e o resultado esperado do agrupamento, realizando a coloração dos grupos identificados pelo algoritmo do autor da base de dados.

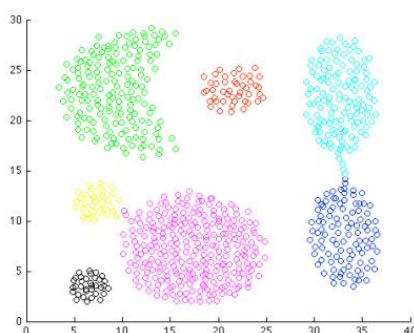
Figura 25 – Recorte de “dados.txt”



15.55	28.65
14.9	27.55
14.45	28.35
14.15	28.8
13.75	28.05
13.35	28.45
13	29.15
13.45	27.5
13.6	26.5
12.8	27.35
12.4	27.85
12.3	28.4
12.2	28.65
13.4	25.1
12.95	25.95
12.9	26.5

Fonte: Captura de tela do autor⁷

Figura 26 – Representação gráfica com os grupos sugeridos



Fonte: Captura de tela do autor⁸

⁷ Captura de tela da base de dados fornecida “dados.txt”.

⁸ Captura de tela do arquivo “Projeto3.pdf”.

Então, foi proposto que os algoritmos a serem implementados realizassem a mesma tarefa de agrupamento, porém utilizando as estratégias citadas anteriormente.

Dentro desse contexto, a resolução do problema proposto deve seguir o seguinte roteiro:

1. Transformar a base de dados com coordenadas (x,y) em pontos, para serem trabalhados como vértices;
2. Realizar todas as ligações possíveis entre os pontos, para serem trabalhadas como arestas;
3. Calcular o tamanho dessas arestas (distância euclidiana entre os pontos), para serem trabalhados como seu peso;
4. Encontrar a Árvore Geradora Mínima usando Prim/Kruskal de toda a base de dados;
5. Tendo as premissas que o que difere um grupo de outro é a distância pequena de seus elementos dispostos em si próprio, e que a retirada de k-1 arestas da AGM (sendo k o número de grupos), serão formados k grupos, deve-se retirar as k-1 arestas de maior peso da AGM formada para que os grupos evidentes sejam formados.
6. Representar graficamente (plotar) o resultado obtido e realizar a comparação com o resultado esperado.

3. IMPLEMENTAÇÃO

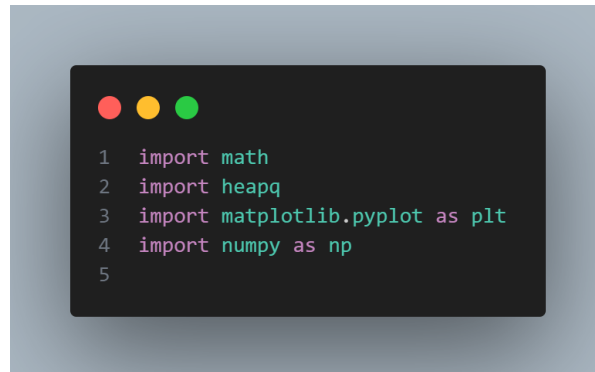
Os algoritmos solicitados pelo docente foram gerados em linguagem Python, com o uso de algumas de suas bibliotecas próprias para o auxílio no uso de estruturas de dados e plotagem de resultados.

3.1 Prim

Para moldar um algoritmo usando as ideias de PRIM, foi requisitada a utilização de uma estrutura heap binária com a finalidade de organizar as arestas a serem analisadas pelo algoritmo em ordem crescente de peso.

Foram utilizadas no algoritmo as bibliotecas: math, para cálculo de distâncias; heapq, para uso simplificado da estrutura heap; numpy e matplotlib para ter acesso às representações gráficas.

Figura 27 – Importação das bibliotecas



Fonte: Código do autor⁹

A função *ler_pontos_do_arquivo*, lê o arquivo de texto passado como parâmetro. Ele lê as coordenadas numéricas e as armazena em uma lista como pares de valores (x, y). Para cada linha do arquivo, a função age removendo espaços desnecessários, ignorando linhas vazias. Ela extrai dois números da linha, os converte para float e, logo após, armazena esses dois números como uma tupla (x, y) na lista pontos.

Figura 28 – Função de leitura da base de dados



Fonte: Código do autor¹⁰

A função *distancia_euclidiana* calcula a distância euclidiana entre dois pontos passados como parâmetro, p1 e p2, no plano cartesiano. Ela faz isso ao aplicar a fórmula da distância euclidiana:

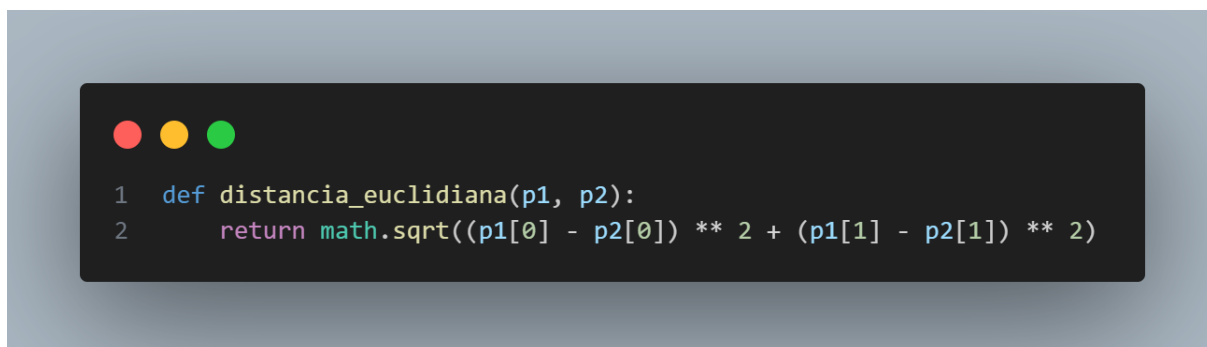
⁹ Captura de tela de trecho do código PRIM.py.

¹⁰ Captura de tela de trecho do código PRIM.py.

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Como p1 e p2 são tuplas que representam coordenadas (x, y), o resultado é a distância linear entre os dois pontos a serem usados.

Figura 29 – Função de cálculo das distâncias entre todos os pontos



Fonte: Código do autor¹¹

Já a função *prim_mst_com_maiores_arestas* encontra a Árvore Geradora Mínima (MST), baseado no algoritmo de Prim e, adicionalmente, retorna as k-1 maiores arestas da MST. Inicialmente, a função cria uma lista de adjacências para armazenar as distâncias entre todos os pares de pontos, calculando as distâncias euclidianas entre eles, usando a função já citada anteriormente.

O algoritmo de Prim é usado para construir a MST. A MST é formada progressivamente, sempre adicionando a aresta de menor peso que conecta um vértice visitado a um não visitado. Para cada aresta adicionada à MST, a função também armazena as maiores arestas. A função usa uma heap (min-heap) para manter as k-1 maiores arestas da MST. Ao adicionar uma aresta à MST, se a heap já contém k-1 arestas, a função substitui a menor aresta (se a nova for maior).

Logo a função retorna todas as arestas da MST com *mst_arestas* e as k-1 maiores arestas encontradas na MST como *maiores_arestas*.

¹¹ Captura de tela de trecho do código PRIM.py.

Figura 30 – Função de montagem da MST com uso de min-heap

```
1 # Função para encontrar a MST e armazenar as maiores arestas
2 def prim_mst_com_maiores_arestas(pontos, k):
3     n = len(pontos)
4     adj = {i: [] for i in range(n)}
5
6     # Construindo a matriz de adjacência com distâncias
7     for i in range(n):
8         for j in range(i + 1, n):
9             dist = distancia_euclidiana(pontos[i], pontos[j])
10            adj[i].append((dist, j))
11            adj[j].append((dist, i))
12
13     # Algoritmo de Prim para encontrar a MST
14     mst_arestas = []
15     visitado = [False] * n
16     pq = [(0, 0, -1)] # (custo, nó atual, nó anterior)
17
18     # Heap para armazenar as (k-1) maiores arestas
19     maiores_arestas = []
20
21     while pq:
22         custo, u, pai = heapq.heappop(pq)
23         if visitado[u]:
24             continue
25         visitado[u] = True
26
27         if pai != -1:
28             mst_arestas.append((pai, u, custo))
29             # Inserir a aresta na heap de maiores arestas (min-heap)
30             if len(maiores_arestas) < k - 1:
31                 heapq.heappush(maiores_arestas, (custo, pai, u))
32             else:
33                 # Manter apenas as (k-1) maiores arestas
34                 heapq.heappushpop(maiores_arestas, (custo, pai, u))
35
36         for proximo_custo, v in adj[u]:
37             if not visitado[v]:
38                 heapq.heappush(pq, (proximo_custo, v, u))
39
40     return mst_arestas, maiores_arestas
```

Fonte: Código do autor¹²

Em essência, a função *prim_agrupamento* realiza o agrupamento (clustering) criando a MST e depois removendo as k-1 maiores arestas para formar k clusters, fazendo a chamada das funções que foram já apresentadas nesse artigo.

¹² Captura de tela de trecho do código PRIM.py.

Cada ponto inicialmente pertence ao seu próprio cluster. Isso é representado pelo array *pai*, onde *pai[i]* é o representante (ou "pai") do cluster ao qual o ponto *i* pertence. A função *encontrar* encontra o "representante" de um cluster, garantindo que todos os pontos de um mesmo cluster compartilhem o mesmo representante.

A função *unir* é responsável por unir dois clusters, ou seja, fazer com que um ponto pertença ao mesmo grupo que outro. Ela une os dois pontos sob o mesmo representante. A função percorre todas as arestas da MST, exceto as *k-1* maiores (armazenadas no conjunto *maiores_arestas_conjunto*), e une os pontos conectados por essas arestas. Assim, os pontos que estão conectados por arestas menores permanecem no mesmo cluster.

Depois de unir os pontos que pertencem ao mesmo cluster, a função percorre todos os pontos e os agrupa de acordo com seus representantes. O resultado final é uma lista de clusters, na qual cada cluster é uma lista de índices de pontos que pertencem ao mesmo grupo.

Assim, a função retorna a lista de clusters, onde cada cluster contém os índices dos pontos que pertencem a ele.

Figura 31 – Função de agrupamento da MST

```
1 # Função para formar k clusters cortando as maiores arestas da MST
2 def prim_agrupamento(pontos, k):
3     # Obter as arestas da MST e as maiores arestas
4     mst_arestas, maiores_arestas = prim_mst_com_maiores_arestas(pontos, k)
5
6     # Converter lista de maiores arestas em um conjunto para fácil remoção
7     maiores_arestas_conjunto = set((u, v) if u < v else (v, u) for _, u, v in maiores_arestas)
8
9     # Inicializar cada ponto como seu próprio cluster
10    pai = list(range(len(pontos)))
11
12    # Função para encontrar o "representante" de um cluster (com compressão de caminho)
13    def encontrar(x):
14        if pai[x] != x:
15            pai[x] = encontrar(pai[x])
16        return pai[x]
17
18    # Função para unir dois clusters
19    def unir(x, y):
20        raizX = encontrar(x)
21        raizY = encontrar(y)
22        if raizX != raizY:
23            pai[raizY] = raizX
24
25    # Unir pontos na MST, exceto pelas k-1 maiores arestas removidas
26    for u, v, _ in mst_arestas:
27        if (u, v) in maiores_arestas_conjunto or (v, u) in maiores_arestas_conjunto:
28            continue # Ignorar as maiores arestas
29        unir(u, v)
30
31    # Agrupando os pontos em seus clusters
32    clusters = {}
33    for i in range(len(pontos)):
34        raiz = encontrar(i)
35        if raiz not in clusters:
36            clusters[raiz] = []
37            clusters[raiz].append(i)
38
39    # Retornar os clusters finais
40    return list(clusters.values())
```

Fonte: Código do autor¹³

Para representar o resultado graficamente, a função *plotar_clusters* define uma lista de cores para a representação. O gráfico é configurado com um tamanho de figura padrão de 8x6 polegadas. A função itera sobre os clusters, extraindo os pontos correspondentes a cada cluster e os plota usando a cor associada. Os pontos de cada cluster são obtidos a partir dos índices dos pontos originais, e as coordenadas X e Y são extraídas usando *np.array*.

¹³ Captura de tela de trecho do código PRIM.py.

Figura 32 – Função de representação gráfica

```
1 def plotar_clusters(pontos, clusters):
2     cores = ['r', 'g', 'b', 'c', 'm', 'y', 'k'] # Cores para os clusters (até 7 clusters)
3
4     plt.figure(figsize=(8, 6))
5
6     for i, cluster in enumerate(clusters):
7         cluster_pontos = np.array([pontos[idx] for idx in cluster])
8         cor = cores[i % len(cores)] # Ciclamos as cores se tivermos mais de 7 clusters
9         plt.scatter(cluster_pontos[:, 0], cluster_pontos[:, 1], c=cor, label=f'Cluster {i + 1}')
10
11     plt.title(f'Agrupamento de Pontos em {len(clusters)} Clusters')
12     plt.xlabel('Coordenada X')
13     plt.ylabel('Coordenada Y')
14     plt.legend()
15     plt.grid(True)
16     plt.show()
```

Fonte: Código do autor¹⁴

No bloco main do código, é colocado o nome do arquivo a ser lido (presente no mesmo diretório do arquivo .py do algoritmo) e o número induzido de clusters desejados, por meio de k. Além disso, são chamadas as funções de leitura, agrupamento e plotagem.

Figura 33 – Bloco main do código

```
1 if __name__ == "__main__":
2     nome_arquivo = "dados.txt" # nome do arquivo com as coordenadas
3     pontos = ler_pontos_do_arquivo(nome_arquivo)
4
5     k = 5 # número de clusters desejados
6
7     clusters = prim_agrupamento(pontos, k)
8
9     print("Clusters formados:")
10    for cluster in clusters:
11        print([pontos[i] for i in cluster])
12
13    # Plotar os clusters formados
14    plotar_clusters(pontos, clusters)
```

Fonte: Código do autor¹⁵

¹⁴ Captura de tela de trecho do código PRIM.py.

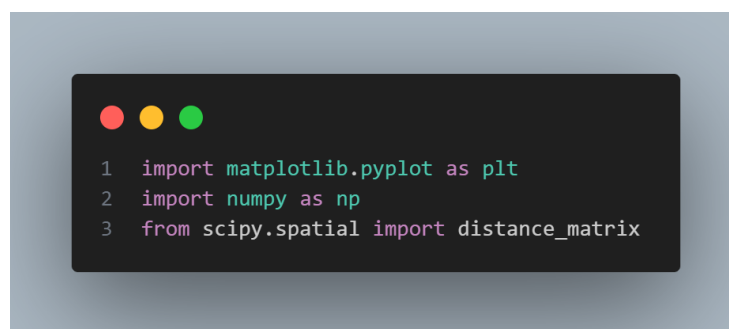
¹⁵ Captura de tela de trecho do código KRUSKAL.py.

3.2 Kruskal

Para moldar um algoritmo usando as ideias de KRUSKAL, foi requisitada a utilização de uma estrutura UNION-FIND, com a finalidade de verificar e realizar uma espécie de agrupamento de vértices e arestas a serem acrescentadas na MST desejada.

Foram utilizadas no algoritmo as bibliotecas: `spicy.spatial`, para cálculo de distâncias; `numpy` e `matplotlib` para ter acesso às representações gráficas.

Figura 34 – Importação das bibliotecas



Fonte: Código do autor¹⁶

A classe *UnionFind* implementa a estrutura de dados Union-Find (ou Disjoint Set Union, DSU), usada para gerenciar e unir os conjuntos disjuntos. Os métodos da classe gerada são:

- **__init__(self, n):** Inicializa a estrutura para *n* elementos. No início, cada elemento é seu próprio pai, e ele lista onde `pai[i]` representa propriamente o "pai" (ou representante) do elemento *i*.
- **encontrar(self, u):** Este método retorna o representante (ou "raiz") do conjunto ao qual o elemento *u* pertence.
- **unir(self, u, v):** Une os conjuntos que contêm os elementos *u* e *v*. Ele primeiro encontra as raízes de *u* e *v* usando o método `encontrar`. Se as raízes forem diferentes, ele une os dois conjuntos, usando o conceito de união por ranque: A raiz do conjunto com maior ranque (altura) se torna a raiz do novo conjunto.
- **componentes_conectadas(self):** Retorna o número de componentes conectadas. A função percorre todos os elementos e conta o número de representantes distintos (raízes). Isso indica quantos conjuntos independentes existem no sistema.

¹⁶ Captura de tela de trecho do código `KRUSKAL.py`.

Figura 35 – Classe UNION-FIND

```
1 class UnionFind:
2     def __init__(self, n):
3         self.pai = list(range(n))
4         self.ranque = [0] * n
5
6     def encontrar(self, u):
7         if self.pai[u] != u:
8             self.pai[u] = self.encontrar(self.pai[u])
9         return self.pai[u]
10
11    def unir(self, u, v):
12        raiz_u = self.encontrar(u)
13        raiz_v = self.encontrar(v)
14
15        if raiz_u != raiz_v:
16            if self.ranque[raiz_u] > self.ranque[raiz_v]:
17                self.pai[raiz_v] = raiz_u
18            elif self.ranque[raiz_u] < self.ranque[raiz_v]:
19                self.pai[raiz_u] = raiz_v
20            else:
21                self.pai[raiz_v] = raiz_u
22                self.ranque[raiz_u] += 1
23
24    def componentes_conectadas(self):
25        return len(set(self.encontrar(i) for i in range(len(self.pai))))
```

Fonte: Código do autor¹⁷

Para cada linha do arquivo, a função *ler_coordenadas* usa *map(float, linha.split())* para dividir a linha em duas partes e convertê-las em números de ponto flutuante (float). Esses valores correspondem às coordenadas X e Y. Em seguida, a função adiciona essas coordenadas como uma tupla (x, y) à lista coordenadas. Após ler todas as linhas, a função converte a lista coordenadas em um array NumPy usando *np.array(coordenadas)*.

¹⁷ Captura de tela de trecho do código KRUSKAL.py.

Figura 36 – Função de leitura das coordenadas



```
1 def ler_coordenadas(caminho_arquivo):
2     coordenadas = []
3     with open(caminho_arquivo, 'r') as f:
4         for linha in f:
5             x, y = map(float, linha.split())
6             coordenadas.append((x, y))
7     return np.array(coordenadas)
```

Fonte: Código do autor¹⁸

A função *agrupar_com_union_find* implementa um algoritmo de agrupamento (clustering) usando a estrutura de dados Union-Find para unir pontos com base em suas distâncias, até que restem exatamente *k* clusters.

A estrutura Union-Find (uf) é inicializada para *n* elementos, permitindo gerenciar as conexões entre os pontos. A matriz de distâncias é calculada usando *distance_matrix(coordenadas, coordenadas)*, onde cada entrada (*i*, *j*) armazena a distância entre os pontos *i* e *j*.

A função cria uma lista arestas contendo todas as arestas possíveis entre os pontos, representadas como tuplas no formato (distância, ponto_i, ponto_j). As arestas são classificadas pela distância, para garantir que as menores distâncias sejam processadas primeiro.

O algoritmo começa com cada ponto em seu próprio cluster, assim a função percorre as arestas (das menores para as maiores) e une os pontos conectados, usando o Union-Find para verificar se os pontos já pertencem ao mesmo cluster. A cada união de dois clusters, o número total de componentes (*num_componentes*) é decrementado. O processo continua até que o número de componentes seja igual a *k*.

Após o processo de união, a função agrupa os pontos de acordo com seus representantes (raízes) na estrutura Union-Find. Para cada ponto, o representante (raiz) é encontrado, e os pontos com o mesmo representante são colocados no mesmo cluster.

¹⁸ Captura de tela de trecho do código KRUSKAL.py.

Figura 37 – Função da montagem da MST e agrupamento

```
1 def agrupar_com_union_find(coordenadas, k):
2     n = len(coordenadas)
3     uf = UnionFind(n)
4
5     matriz_distancias = distance_matrix(coordenadas, coordenadas)
6
7     arestas = []
8     for i in range(n):
9         for j in range(i+1, n):
10             arestas.append((matriz_distancias[i, j], i, j))
11     arestas.sort()
12
13     num_componentes = n
14     for dist, i, j in arestas:
15         if uf.encontrar(i) != uf.encontrar(j):
16             uf.unir(i, j)
17             num_componentes -= 1
18         if num_componentes == k:
19             break
20
21     clusters = {}
22     for idx, coord in enumerate(coordenadas):
23         raiz = uf.encontrar(idx)
24         if raiz not in clusters:
25             clusters[raiz] = []
26         clusters[raiz].append(coord)
27
28     return clusters
```

Fonte: Código do autor¹⁹

Assim como na versão de Prim, a função *plotar_clusters* é responsável por gerar a visualização os clusters gerados após o processo de agrupamento, primeiramente definindo uma lista de cores e criando o gráfico com uma figura de tamanho 8x6 polegadas.

A função itera sobre o dicionário clusters retorna pela função de agrupamento, onde cada chave (*id_cluster*) é o identificador do cluster, e o valor (*pontos*) é a lista de pontos pertencentes a esse cluster.

Cada conjunto de pontos é convertido para um array NumPy para facilitar a extração das coordenadas X e Y. A função utiliza *plt.scatter* para desenhar os pontos de cada cluster com uma cor diferente, extraída ciclicamente da lista de cores.

¹⁹ Captura de tela de trecho do código KRUSKAL.py.

Figura 38 – Função de agrupamento da MST

```
1 def plotar_clusters(clusters):
2     cores = ['b', 'g', 'r', 'c', 'm', 'y', 'k']
3     plt.figure(figsize=(8, 6))
4
5     for i, (id_cluster, pontos) in enumerate(clusters.items()):
6         pontos = np.array(pontos)
7         plt.scatter(pontos[:, 0], pontos[:, 1], c=cores[i % len(cores)], label=f'Cluster {id_cluster+1}')
8
9     plt.legend()
10    plt.xlabel('X')
11    plt.ylabel('Y')
12    plt.title('Agrupamento com Conjunto Disjunto')
13    plt.show()
```

Fonte: Código do autor²⁰

Para representar o resultado graficamente, a função *plotar_clusters* define uma lista de cores para a representação. O gráfico é configurado com um tamanho de figura padrão de 8x6 polegadas. A função itera sobre os clusters, extraindo os pontos correspondentes a cada cluster e os plota usando a cor associada. Os pontos de cada cluster são obtidos a partir dos índices dos pontos originais, e as coordenadas X e Y são extraídas usando *np.array*.

Figura 39 – Função de representação gráfica

```
1 def plotar_clusters(pontos, clusters):
2     cores = ['r', 'g', 'b', 'c', 'm', 'y', 'k'] # Cores para os clusters (até 7 clusters)
3
4     plt.figure(figsize=(8, 6))
5
6     for i, cluster in enumerate(clusters):
7         cluster_pontos = np.array([pontos[idx] for idx in cluster])
8         cor = cores[i % len(cores)] # Ciclamos as cores se tivermos mais de 7 clusters
9         plt.scatter(cluster_pontos[:, 0], cluster_pontos[:, 1], c=cor, label=f'Cluster {i + 1}')
10
11    plt.title(f'Agrupamento de Pontos em {len(clusters)} Clusters')
12    plt.xlabel('Coordenada X')
13    plt.ylabel('Coordenada Y')
14    plt.legend()
15    plt.grid(True)
16    plt.show()
```

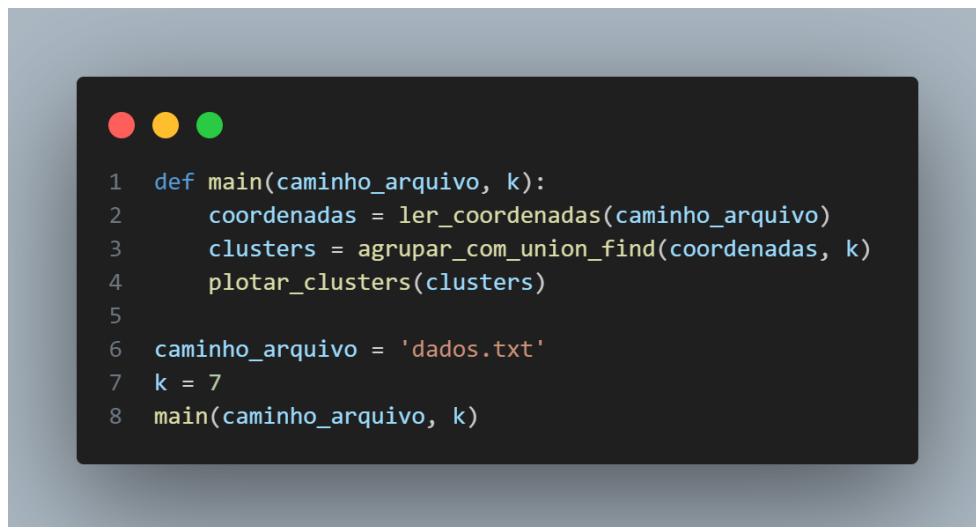
Fonte: Código do autor²¹

²⁰ Captura de tela de trecho do código KRUSKAL.py.

²¹ Captura de tela de trecho do código KRUSKAL.py.

A função `main` aceita dois parâmetros: O caminho para o arquivo que contém as coordenadas dos pontos (`caminho_arquivo`) e o número induzido de clusters a serem formados (k). As funções `ler_coordenadas`, `agrupar_com_union_find` e `plotar_clusters` são chamadas posteriormente, para cumprir o propósito do algoritmo.

Figura 40 – Bloco main do código

A screenshot of a code editor window with a dark background and light-colored text. The editor has three colored window control buttons (red, yellow, green) in the top-left corner. The code is written in Python and consists of eight lines. Lines 1-4 define the `main` function, which calls `ler_coordenadas`, `agrupar_com_union_find`, and `plotar_clusters`. Lines 6-8 set the file path to 'dados.txt', set `k` to 7, and call the `main` function.

```
1 def main(caminho_arquivo, k):
2     coordenadas = ler_coordenadas(caminho_arquivo)
3     clusters = agrupar_com_union_find(coordenadas, k)
4     plotar_clusters(clusters)
5
6 caminho_arquivo = 'dados.txt'
7 k = 7
8 main(caminho_arquivo, k)
```

Fonte: Código do autor²²

3.3 Análise assintótica

O algoritmo de Prim pode ser descritos assintoticamente quanto a complexidade de tempo por $O(n^2 + n \log n)$. Tal ocorrência se deve ao cálculo algébrico das distâncias de todos os pontos entre si, realizados para conseguir otimalidade na construção da MST em cima da base de dados, e por conta da estrutura heap utilizada de complexidade $O(n \log n)$. Visto isso, o algoritmo no geral está limitado superiormente por n^2 , tendo como complexidade $O(n^2)$.

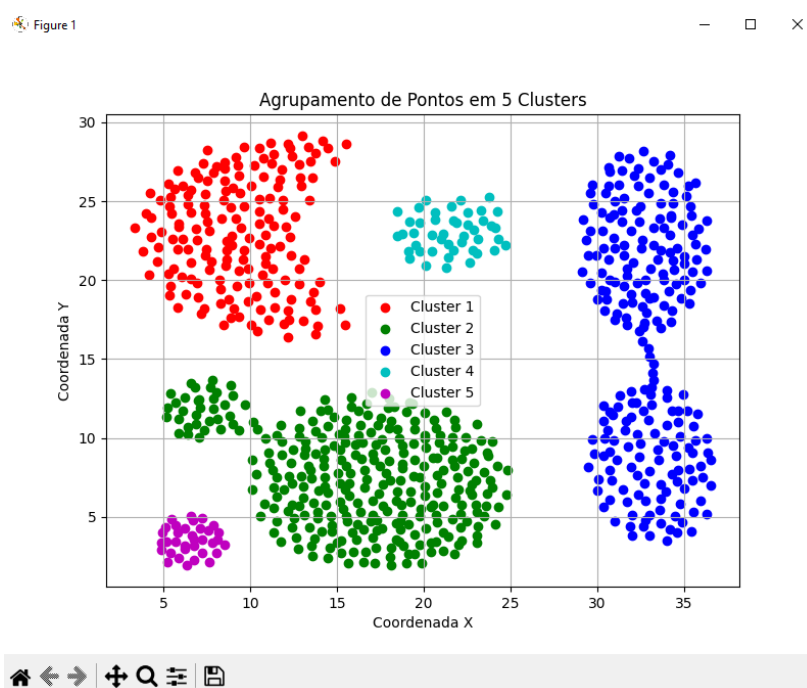
Já o algoritmo de Kruskal, tem complexidade total $O(n^2 \log n)$. Isso ocorre porque para n pontos, o número de arestas é $(n(n-1))/2$, que é $O(n^2)$. A ordenação das arestas, que é realizada com `arestas.sort()`, tem complexidade $O(m \log m)$, onde m é o número de arestas. No pior caso, m é $O(n^2)$. Logo, o custo de ordenação é de $O(n^2 \log n)$.

²² Captura de tela de trecho do código KRUSKAL.py.

4. ANÁLISE DE RESULTADOS

Como visto na figura 26, ao utilizar a base de dados fornecida, eram esperados a formação de sete clusters. Os resultados de ambos algoritmos implementados foram diferentes. Seja o código utilizando a técnica de Prim, seja o de Kruskal, apresentaram dificuldade em separar os grupos que são visualmente distintos, mas são ligados por pontos com uma amplitude pequena entre si. Tais ligações frágeis podem ser denominadas “pontes”.

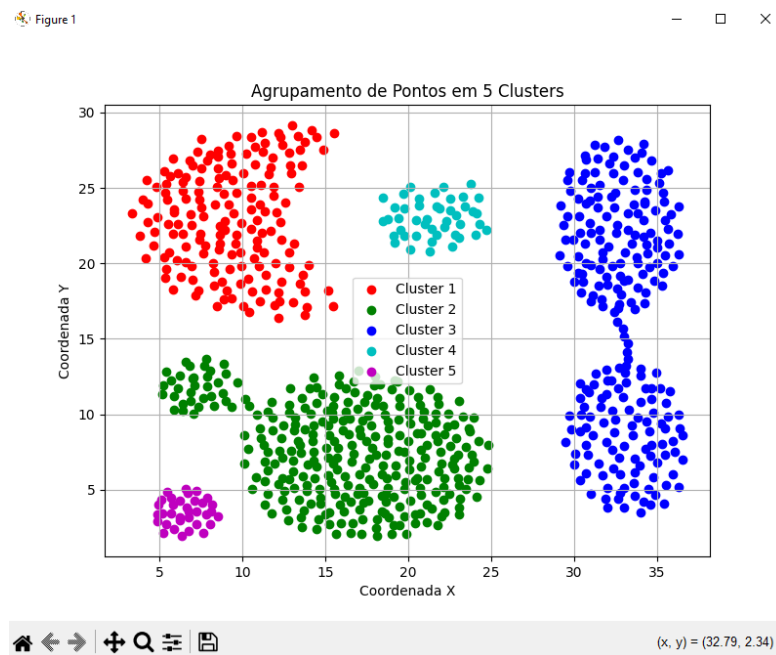
Figura 41 – Plotagem dos resultados de PRIM.py



Fonte: Código do autor²³

²³ Captura de tela da plotagem de resultados de PRIM.py.

Figura 42 – Plotagem dos resultados de KRUSKAL.py



Fonte: Código do autor²⁴

A problemática ocorre porque ambos os algoritmos após realizarem a construção de uma MST com a amostragem de dados total, se baseiam na premissa que os grupos são separados pelas maiores dessa MST, o que pode fazer sentido em amostragens de dados pouco dispersas, mas pode afetar resultados em trechos com estruturas como as pontes, citadas anteriormente.

Quando retiram as $k-1$ maiores arestas desta MST, os algoritmos separam k grupos que eram separados pelas arestas retiradas. Nas plotagens mostradas nas figuras 41 e 42, foi induzido $k=5$, caso a alteração $k=7$ fosse realizada, o resultado com os grupos sugeridos ainda não seria alcançado, uma vez que seriam retiradas mais duas arestas com o valor de peso mais alto dentre as restantes, e não as que separam os grupos visualmente reconhecidos.

Não foi encontrada uma forma eficiente de resolver tal problemática sem alterar drasticamente a estrutura do código, usando os conceitos requisitados na proposição do trabalho e com aspecto eficiente quanto a complexidade de tempo.

²⁴ Captura de tela da plotagem de resultados de KRUSKAL.py.

5. CONCLUSÕES

A construção de Árvores Geradoras Mínimas pode ser efetiva para vários contextos na análise de dados, inclusive para agrupamentos. Nesse contexto, algoritmos que utilizam estratégias de Prim e Kruskal, como os apresentados, conseguem ser de grande utilidade em diversos modelos de análise e clusterização.

Ao realizar abordagens como as utilizadas nesse projeto, é possível que sejam encontrados obstáculos que impeçam a conquista de um resultado satisfatório. Porém, ambos os algoritmos se portam muito bem no agrupamento de dados não tão condensados, realizando sua finalidade dentro de um gasto de processamento raso e com uma complexidade de tempo aceitável.

REFERÊNCIAS

JAIN, A. K.; MURTY, M. N.; FLYNN, P. J. Data clustering: a review. *ACM Computing Surveys*, New York, v. 31, n. 3, p. 264-323, set. 1999. Acesso em: 29 set. 2024.

GIONIS, A., H. MANNILA, and P. TSAPARAS. Clustering aggregation. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 2007. 1(1): p. 1---30. Acesso em: 29 set. 2024.