INSTITUTO UNIVERSITÁRIO DE LISBOA

Ano Letivo 2021/2022

2 de junho de 2022

Facebook - Análise das Comunidades

Trabalho de Grupo Final | EDA

UC Estrutura de Dados e Algoritmos

Licenciatura Ciência de Dados

Grupo 12, CDA1

Docentes

Ana Maria de Almeida

Maria Pinto-Albuquerque

Ricardo António

Rodrigo Sarroeira

André Silvestre Nº104532

Diogo Catarino Nº104745

Francisco Gomes Nº104944

Índice

Introdução

TAD Grafo

Um **Tipo Abstrato de Dados (TAD)** especifica um conjunto de operações (métodos) e os seus processos (o que faz), mas não especifica a implementação das operações, o que o faz ser abstrato.

Um **Grafo** é uma forma de representar relações que existem entre pares de objetos. Ou seja, um grafo é um conjunto de objetos, designados **Vértices**, juntamente com uma coleção de ligações de pares entre eles, designadas **Arestas**.

Há diversas classificações para esta **estrutura de dados não linear**. Se as relações – arestas – não forem dirigidas, isto é, se afetarem ambos os vértices, *u* e *v*, de igual forma, diz-se que o grafo é **não dirigido** *(undirected)* ou **não orientado**. Todavia, se as relações forem direcionadas, diz-se que o grafo é **dirigido** *(directed)*, **orientado** ou **dígrafo**. [1]

Devido à sua estrutura, os grafos têm diversas aplicabilidades na modelação de muitos domínios do quotidiano. A título de exemplo, mapeamentos, transportes, redes de computadores e engenharia elétrica. Sendo os grafos uma das estruturas de dados mais comuns para modelação, este é uma componente chave em muitos dos sistemas que usamos diariamente, tais como, o encaminhamento no *Google Maps,* ou a recomendação de amigos nas redes sociais.

Caso Prático

Neste sentido, foi proposto no âmbito da UC de *Estrutura de Dados e Algoritmos* inserida na licenciatura de Ciência de Dados, um projeto que congrega os Grafos e o estudo de comunidades que se podem obter de um conjunto de dados sobre *profiles* e *circles* (*friends lists*) no ***Facebook***.

As **redes de partilha** e **colaboração** (**Fig.1**) são importantes nos mais variados domínios (pesquisa científica, construção de novos produtos ou serviços, convívio, informação, saúde, governação, entre outros), sendo a rede do *Facebook* apenas um exemplo disso.

A **Análise de Redes Sociais** é um ramo da sociologia que explora estruturas sociais através do uso de ferramentas analíticas, tais como grafos. [2]

Assim, pretende-se com este trabalho implementar algoritmos para a análise de redes sociais aplicando-os num grafo com informações extraídas da rede social do *Facebook,* para o **estudo** das **comunidades** e **sub-comunidades** do mesmo.

Para cumprir esse objetivo, começaremos por **implementar** um **TAD Grafo** com uma função para obter uma **Árvore de Cobertura Mínima (**MST**)**, usando o **algoritmo de Kruskal**.

Posteriormente, **aplicaremos** essa estrutura de Grafo no **estudo de comunidades** que se podem obter de um conjunto de dados sobre ***profiles*** e ***circles*** (*friends lists*) no ***Facebook***. Nesta fase desenvolveremos um **algoritmo de agrupamento** (*clustering*) **hierárquico**; criaremos os **(sub)grafos** para cada ***cluster;*** e usaremos o métodos da biblioteca **`NetworkX`** para o estudo de sub-comunidades em cada comunidade (representada por um grafo).

Figura 1 | Representação de uma Rede de Pessoas. [2]

1. Implementação do TAD Grafo

Vértice

Denominam-se vértices às representações de **nós** num grafo. Estes são um elo fundamental na implementação de um grafo, uma vez que este é criado através da ligação de vértices, formando arestas. [1]

Na nossa implementação criámos uma **`class Vertex`**, cujo construtor da classe tem o atributo **`self._id`**, de forma a que seja possível armazenar informação relativa a esse vértice, independentemente de que natureza seja (números, letras, objetos).

Usámos assim a função ***`__eq__`***, que tem como objetivo comparar dois vértices, verificando os respetivos **`id`**. Este método é de implementação obrigatória, uma vez que definimos a função ***`__hash__`***. Esta tem como objetivo devolver-nos o número inteiro identificador do vértice, baseado no **`self._id`** do mesmo, pelo que usámos a função **`hash()`** do Python.

Por último, definimos igualmente o método ***`__str__`*** que devolve o objeto guardado no vértice em forma de *string*.

```
# ================================================ [CLASS VERTEX] ================================================
class Vertex:  # Classe Vértice - User do Facebook

    # [Construtor] - só tem um atributo/id que guarda o objeto que pretendemos guardar nesse vértice
    def __init__(self, x):
        self._id = x

    # [id] - Devolve o id neste vértice
    @property
    def id(self):
        return self._id

    # [__eq__] - Função booleana que diz se um Vértice de id X é igual ao Vértice em que estamos (o do self)
    #            É obrigatório definir a função __eq__ quando se define a função __hash__
    def __eq__(self, x):
        return x == self._id  # x é um objeto- quero saber com esta função
        # se é igual ao objeto/item que está guadado no self._id

    # [__hash__] - Getter para obter a identificação de um Vértice feito chamando as funções hash() e id() do Python
    #              Devolve um inteiro que identifica este vértice como uma chave num dicionário
    def __hash__(self):
        return hash(self._id)

    # [__str__] - Função que devolve o objeto guardado no Vértice em string
    def __str__(self):
        return '{0}'.format(self._id)
```

Figura 2 | *Class Vertex.*

Aresta

As arestas representam as ligações entre vértices. Estas podem ser classificadas como: direcionadas, caso contenha um vértice origem e um vértice destino definidos; ou não direcionadas, caso contrário. [1]

De forma a implementar a conexão entre vértices, foi criada a **`classe Edge`** (Aresta) que liga, num tuplo, o vértice de origem com o vértice de destino e atribui o peso correspondente. Assim, o construtor da classe inicializa esses mesmos atributos.

Para acessar todos os atributos, definimos **`@properties`** que devolvem os atributos correspondentes. Adicionalmente, criamos a função **`endpoints`**, que devolve um tuplo com os dois vértices associados à aresta, e a função **`opposite`** que, após receber um vértice de uma aresta, vai devolver o outro vértice da mesma aresta.

Tal como na **`class Vertex`**, implementamos a função **`__hash__`** que devolve um identificador único para a aresta, usando a função **`hash()`** do Python, tendo por base o tuplo **`(self._origem, self._destino)`**. Devido à obrigatoriedade de definir a função **`__eq__`** quando se define a **`__hash__`**, implementámo-la, sendo esta uma função booleana que compara duas arestas.

Por fim, o método **`__str__`** que devolve a informação de uma aresta (origem, destino) em forma de *string*.

```
class Edge:  # Classe Aresta - Conecção entre Users
    """Estrutura de Aresta - (origem, destino), com peso - para um grafo"""

    # [Contrutor] - Inicializa os atributos vértice de origem, vértice de destino e peso
    def __init__(self, u, v, p):
        """A aresta será inserida no no Grafo usando insert_edge(u,v,x)"""
        self._origem = u
        self._destino = v
        self._peso = p

    # [origem] - Devolve o Vértice Origem
    @property
    def origem(self):
        return self._origem

    # [destino] - Devolve o Vértice destino
    @property
    def destino(self):
        return self._destino

    # [peso] - Devolve o Peso da Aresta
    @property
    def peso(self):
        return self._peso

    # [endpoints] - Função que devolve num tuple os dois vértices associados à aresta
    @property
    def endpoints(self):
        return self._origem, self._destino
```

```
    # [opposite] - Dado um vértice v (que pode ser origem ou destino desta aresta) devolve o vértice que está na
    #              outra ponta da aresta
    def opposite(self, v):
        # O if - else compacto lê-se -> se v é o vértice origem da aresta, a função devolve o vértice destino
        # Caso contrário devolve o vértice origem
        return self._destino if v is self._origem else self._origem

    # [__eq__] - Função booleana que compara duas Arestas.
    #          É obrigatório definir a função __eq__ quando se define a função __hash__
    def __eq__(self, other):
        return self._origem == other._origem and self._destino == other._destino

    # [__hash__] - Função que devolve um identificador único para a edge/aresta usando a função hash() do
    #              Python sobre o tuplo (vértice origem, vértice destino)
    def __hash__(self):
        return hash((self._origem, self._destino))

    # [__str__] -Função que devolve a informação(origem, destino) em string
    def __str__(self):
        return '({0},{1} | {2})'.format(self._origem, self._destino, self._peso)
```

Figura 3 | *Class Edge.*

Grafo

A estrutura de Grafo pode ser representada de diversas formas, sendo que a maior diferença entre elas é a forma como se organizam as arestas. A implementação realizada neste projeto foi a do **mapa de adjacência**, em que o *depósito secundário* de todos os vértices incidentes de arestas são organizados como um mapa através da estrutura de **dicionário**, em que cada **vértice** é a ***chave*** e a **aresta** é o ***valor***. [1]

A implementação de **TAD Grafo** apresentada de seguida possui métodos de inserção, pesquisa e remoção, entre outros, métodos auxiliares.

```
# ------------------------------------------------ [CLASS GRAFO] ------------------------------------------------
class Graph:
    """Representação de um grafo usando mapas de adjacências (associações)"""

    # [Construtor do Grafo] - Inicializa os atributos directed, number e vertices
    def __init__(self, directed=False):
        """Cria um grafo vazio (contentor _vertices); é orientado se o parâmetro directed tiver o valor True"""
        self._directed = directed  # indica se o grafo é dirigido (valor True) ou não (valor False)
        self._number = 0  # quantidade de nós
        self._vertices = {}  # dicionário com os vários vértices como chaves e em que o valor
        # associado a cada vértice é o dicionário de adjacências desse vértice
        # (associação: outro_vértice -> aresta que os liga)
```

Figura 4 | Construtor da *Class Graph.*

Inicializando a classe, o construtor **`__init__`** cria um dicionário **`self._vertices`** que guarda os vértices de origem como *key* e a aresta como *value*, formando assim o mapa de adjacências. Adicionalmente, tem as variáveis **`self._directed`** e **`self._number`** que armazenam um *booleano* que refere se o grafo é direcionado e o número total de nós do grafo, respetivamente.

```python
# [insert_vertex] - Insere e devolve um novo vértice com o id x
def insert_vertex(self, x):
    v = Vertex(x)  # Criar um vértice chamando o seu construtor e passando o objeto x a guardar no vértice
    if v not in self._vertices.keys():
        self._vertices[v] = {}  # Inicializar o dicionário/mapa de adjacências correspondente ao vértice v a vazio
    return v

# [insert_edge] - Insere e devolve uma nova aresta entre u e v com peso x
def insert_edge(self, u, v, x=None):
    e = Edge(u, v, x)
    self._vertices[u][v] = e  # vai colocar nas incidências de u
    self._vertices[v][u] = e  # e nas incidências de v (para facilitar a procura de todos os arcos incidentes)
    return e
```

Figura 5 | Inserir da *Class Graph*.

O método de **`insert_vertex()`** (Fig.5) insere e devolve um vértice no grafo. Para tal, o elemento **`x`** a inserir é dado como argumento, sendo criado um **`Vertex`** com esse item, seja de que natureza for (número, letras, entre outros), e caso não esteja no dicionário **`if v not in self._vertices.keys():`** o vértice é adicionado como *key* e é inicializado o dicionário/mapa de adjacências como *value* correspondente ao vértice. Por fim, é sempre devolvido o vértice criado/inserido.

Uma vez que o grafo é não direcionado, ambos os vértices afetam/relacionam-se de igual forma na aresta. Este é também um grafo pesado, pois todas as arestas entre dois vértices têm associado um valor numérico, representando o **peso** da ligação, que com a base de dados fornecida terá um significado próprio. [1]

Assim, no método **`insert_edge()`** (Fig.5) insere uma nova aresta entre **`u`** e **`v`** com peso **`x`** e devolve-a. Para tal, cria-se um objeto do tipo **`Edge`** com os vértices de origem e destino e o peso associado, e adiciona-se essa aresta no dicionário **`self._vertices`** ficando assim nas incidências de **`u`** e nas incidências de **`v`** essa aresta adjacente, dado que o grafo a implementar é não dirigido, facilitando assim a procura da aresta.

```python
# [remove_edge] - Remove a aresta entre u e v
def remove_edge(self, u, v):
    if u in self._vertices.keys() and v in self._vertices[u].keys():
        del self._vertices[u][v]
        del self._vertices[v][u]

# [remove_vertex] - Remove o vértice v
def remove_vertex(self, v):
    # remover todas as arestas de [v]
    # remover todas as arestas com v noutros vertices
    # remover o vértice
    lst = [i for i in self.incident_edges(v)]
    for i in lst:
        x, y = i.endpoints()
        self.remove_edge(x, y)
    del self._vertices[v]
    return v
```

Figura 6 | Remover da *Class Graph*

Já quanto ao **`remove`** (Fig.6), faz o processo oposto. Isto é, no caso de remover o vértice, elimina todas as arestas incidentes e por fim elimina-o da lista de *keys* do dicionário.

Enquanto no remover da aresta, apenas é necessário remover o *value* do dicionário que a mesma constava.

No código entregue, existem mais métodos que foram definidos nesta classe, cujos seus objetivos estão descritos em comentários ao longo do código. Estes são, essencialmente, funções auxiliares que terão utilidade ao longo do projeto para as mais diversas implementações. A título de exemplos, salientamos a `@property` `edges` que devolve o conjunto de todas as arestas do grafo, sendo imprescindível para a construção e visualização do grafo através da biblioteca **`NetworkX`**, tal como será explorado mais adiante.

Análise da Complexidade Temporal

Segundo [1], a tabela seguinte (Fig. 7) diz respeito à Complexidade Algorítmica Temporal, representando assim os tempos de execução dos métodos do TAD Grafo, que foram abordadas em aula. É importante denotar que $\boldsymbol{v}$ é o número de vértices, $\boldsymbol{m}$ o número de arestas, $\boldsymbol{d_v}$ o grau de vértice $\boldsymbol{v}$, $\boldsymbol{d_u}$ é o grau do nó $\boldsymbol{u}$ e $\boldsymbol{exp}$ é o tempo esperado.

Operation	Edge List	Adj. List	Adj. Map	Adj. Matrix
vertex_count( )	$O(1)$	$O(1)$	$O(1)$	$O(1)$
edge_count( )	$O(1)$	$O(1)$	$O(1)$	$O(1)$
vertices( )	$O(n)$	$O(n)$	$O(n)$	$O(n)$
edges( )	$O(m)$	$O(m)$	$O(m)$	$O(m)$
get_edge(u,v)	$O(m)$	$O(\min(d_u, d_v))$	$O(1)$ exp.	$O(1)$
degree(v)	$O(m)$	$O(1)$	$O(1)$	$O(n)$
incident_edges(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n)$
insert_vertex(x)	$O(1)$	$O(1)$	$O(1)$	$O(n^2)$
remove_vertex(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n^2)$
insert_edge(u,v,x)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$
remove_edge(e)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$

Figura 7 | Complexidade Algorítmica Temporal das diferentes implementações do TAD Grafo

Já relativamente à Complexidade Algorítmica Espacial, a matriz de adjacências tem uma complexidade de $\boldsymbol{O(n^2)}$, enquanto que as restantes têm complexidade $\boldsymbol{O(n+m)}$.

Assim sendo, e tendo por base a implementação de grafo solicitada (mapa de adjacências), que está salientada na Figura 7, podemos concluir que o TAD Grafo implementado é, teoricamente, de entre todas as implementações aqui representadas, o mais eficiente.

2. Construção do TAD Grafo Teste

Experiência

A fim de testar quer o TAD Grafo implementado, quer posteriormente os Algoritmos de *Kruskal* e *Clustering,* desenhámos um grafo teste (Fig. 8)

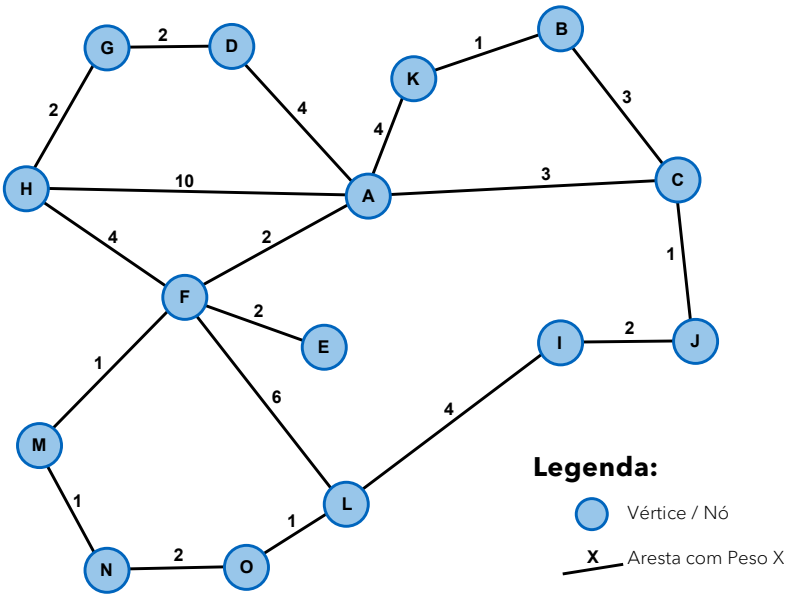

Figura 8 | Esquematização do Grafo criado.

Este grafo é constituído por 15 vértices ligados a 19 arestas, com pesos compreendidos entre 1 e 10. Dado que todos os vértices estão ligados, podemos então considerar este um grafo conexo, permitindo assim aplicar o *Kruskal* e, posteriormente, o Agrupamento Hierárquico.

Para testar o carregamento e visualização deste grafo, criámos o ficheiro `Grafo_Teste.csv` em que cada linha corresponde a uma aresta contendo **`Vértice Origem, Vértice Destino, Peso`**.

De modo a garantir o bom funcionamento deste grafo para o restante projeto, tivemos em conta que cada aresta é única, não havendo arestas paralelas; e que o grafo é conexo, tal como referido anteriormente.

Para carregar e visualizar o grafo, criámos a função **`construir_grafo_teste()`** apresentada de seguida (Fig.9).

```python
def construir_grafo_teste(filename):
    print('[Inicio] - Construir um grafo...')

    # Inicialização da Classe Graph
    g = Graph()

    # Importar os valores do CSV
    f = open(filename, 'r')
    print(f'\nReading... [Origem, Destino, Peso]')

    i = 0
    for row in f:
        Origem, Destino, Peso = row.strip(" ").strip("\n").split(",")
        print(i, " ----- ", Origem, Destino, Peso)

        v1 = g.insert_vertex(Origem)
        v2 = g.insert_vertex(Destino)

        g.insert_edge(v1, v2, Peso)

        i += 1

    return g
```

```python
if __name__ == "__main__":
    g = construir_grafo_teste('Grafo_Teste.csv')

    # Criação do Grafo da biblioteca NetworkX e Carregamento das Arestas
    print(f'\nNetworkX Graph Without Kruskal...')
    nx_graph = nx.Graph()
    for e in g.edges:
        nx_graph.add_edge(str(e.origem.id), str(e.destino.id), weight=int(e.peso))

    # Representação do Grafo
    print(f'\nDrawing NetworkX Graph...')
    nx.draw(nx_graph, node_size=500, with_labels=True, font_size=15, edge_color="#525252", font_weight='bold',
            node_color="#43b3ae", font_color="black", node_shape="o", alpha=0.7, linewidths=4)
    plt.show()
    # ------------------------------------------------------------------------------------------------------------
```

Figura 9 | Implementação em Python da função **`construir_grafo_teste()`** e respetivo teste.

Nesta função, começamos por inicializar um objeto da classe **`Graph`** e importamos os valores do **`csv`** para o mesmo. Nesta importação, usámos as funções do Python **`strip`** e **`split`** para delimitar que parte era os vértices de origem e chegada, e respetivo peso.

Para cada linha percorrida no **`csv`** criámos os vértices **`v1`** e **`v2`** e inserimos a aresta respetiva **`g.insert_edge(v1, v2, Peso)`**. É de notar que a criação de vértices com igual **`self._id`** irá resultar num mesmo vértice, pois na **`class Vertex`**, definimos os métodos de ***`__hash__`*** e ***`__eq__`*** para que se reconheçam os vértices pelo seu **`id`**, evitando a sua duplicação.

Resultante de uma pesquisa na biblioteca **`NetworkX`** em [3], criámos um grafo desta mesma biblioteca e carregámo-lo com a informação contida no grafo anteriormente criado.

Salienta-se que apesar de na documentação estar assegurado que objetos *hashable* podem, em teoria, ser usados para a construção da aresta (objetos da classe **`Vertex`**), este mesmo método parece conter algum *bug*, que impossibilita a criação de nós não duplicados com estes objetivos. Logo, tivemos de optar por dar a informação contida no vértice, **`str(e.origem.id)`** e **`str(e.destino.id)`**, em vez de dar o próprio vértice de origem e destino.

Por fim, através do método **`nx.draw()`**, podemos visualizar o grafo criado (Fig.10). Colocámos os diversos argumentos nessa função, de modo a customizar a visualização.

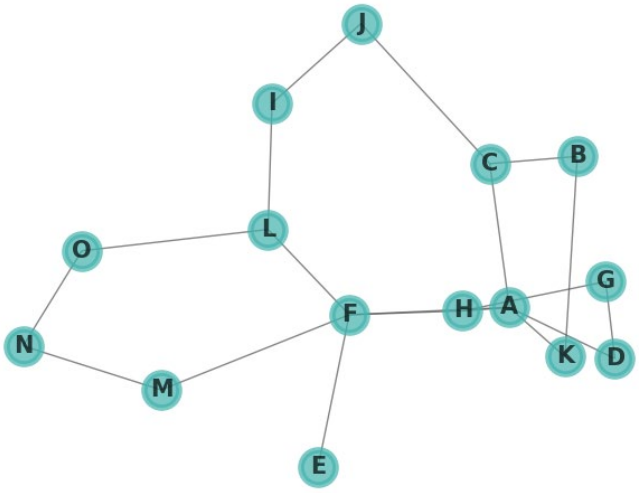

Figura 10 | Resultado da Visualização do grafo construído com o **`NetworkX`**.

3. Algoritmo de Kruskal

O Algoritmo de *Kruskal* surgiu pela primeira vez em 1956 no livro [5], escrito por Joseph Kruskal.

O principal objetivo deste algoritmo é construir uma Árvore de Cobertura Mínima (MST – *Minimum Spanning Tree*), analisando o peso associado às arestas e incluindo uma a uma na árvore. É, por isso, um subgrafo de um grafo, onde a decisão de inclusão de uma aresta privilegia sempre a aresta de custo mínimo que não cria ciclos com as já incluídas na árvore. [6]

Este é considerada uma estratégia *Greedy,* pois escolhe sempre o próximo passo (neste caso, cada aresta que irá adicionar à árvore) de acordo com o que determinámos ser o que melhor favorece o problema em causa. Isto é, a cada passo, é escolhida a solução ótima, ou seja, a aresta que vai ser adicionada é sempre a que tem um menor peso e a que não cria ciclos com as arestas já incluídas na árvore. [7]

O algoritmo de *Kruskal* aplica-se a grafos conexos, ou seja, em grafos não orientados onde é possível atingir qualquer vértice a partir de outro, podendo assim existir um único caminho com o menor peso possível.

Dada a sua complexidade, a implementação mais eficiente de implementá-lo é usar uma estrutura de dados `Union-Find` que fornece um conjunto de operações eficientes, como a união e pesquisa de dois vértices e a verificação da associação dos mesmos. Quando as arestas estão ordenadas, essa estrutura de dados permite verificar facilmente se dois vértices da aresta pertencem à mesma árvore. Se este for o caso, as árvores podem ser facilmente unidas. [4]

Funcionamento do Algoritmo

Antes de construir o algoritmo efetivamente dito, é necessário implementar a estrutura de dados `Union-Find` que implementa as operações auxiliares `MakeSet(v)`, `Find(v)` e `Union(A,B)`. [1]

O método auxiliar `MakeSet(v)` tem como principal objetivo criar um conjunto singular com um vértice `v`, permitindo assim pesquisar e unir vértices com maior facilidade. A complexidade temporal esperada para o mesmo é $\boldsymbol{O(1)}$.

Já a função `Find(v)` funciona como pesquisa, devolvendo o indentificador do conjunto que contém `v`. A complexidade temporal esperada é $\boldsymbol{O(log\ n)}$.

Por fim, o procedimento **`Union(A,B)`** serve essencialmente para unir dois conjuntos de vértices, procurando-os e organizando-os por *ranks*, de modo a decidir qual o vértice que vai unir. A complexidade temporal esperada é $\boldsymbol{O(log\ n)}$.

Já a função relativa ao Algoritmo de **`Kruskal`**, pode ser sistematizada no seguinte pseudocódigo (Fig.11).

```
Kruskal (V, E):
    ordenar E por ordem crescente de peso
    F ← (V, Ø)
    para cada vértice v ∈ V
        MakeSet(v)
    para i ← 1 a |E|
        uv ← i-ésima aresta em E
        se Find(u) ≠ Find(v)
            Union(Find(u), Find(v))
            adicionar uv to F
    devolver F
```

Figura 11 | Pseudocódigo do Algoritmo de **`Kruskal`**.

Dado um grafo constituído por Vértices(**`V`**) e Arestas (**`E`**), começamos por guardar numa estrutura ordenável todas as arestas, de modo a organizá-las por ordem crescente. De seguida, para cada vértice **`v`**, contido no grafo, criamos um conjunto através da função auxiliar **`MakeSet(v)`**.

Após esse ciclo, o algoritmo vai iterar a estrutura usada anteriormente, cujas arestas estão guardadas, à qual procura os vértices de origem e destino através do método auxiliar **`Find()`** e compara os vértices que esta está a ligar. Se estes forem diferentes, significa que pertencem a conjuntos diferentes, e então estas árvores irão unir-se numa só. Caso estes vértices pertencessem ao mesmo conjunto a aresta, não seria adicionada, pois ao uni-los iria criar um ciclo, visto que já pertenciam à mesma árvore.

Como já referido anteriormente, a iteração irá continuar até formar a Árvore de Cobertura Mínima, devolvendo-a. [1][4][6][7]

O Algoritmo de Kruskal possui uma Complexidade Temporal esperada de $\boldsymbol{O(m\ log\ n)}$ onde $\boldsymbol{m}$ representa o número de arestas e $\boldsymbol{n}$ o número de vértices. [1]

Implementação em Python

Transpondo o pseudocódido para a linguagem Python, apresentamos de seguida a implementação realizada (Fig.12 e Fig.13).

```python
# ============================================ [MST] ================================================
# [FUNÇÕES AUXILIARES] - ALGORITMO DO KRUSKAL
conjunto = {}
rank = {}

# [MakeSet (v)] - Criar um conjunto singular com um vértice v
def MakeSet(self, vertice):
    self.conjunto[vertice] = vertice
    self.rank[vertice] = 0

# [Find(v)] - Devolve o vértice v do conjunto
def Find(self, v):
    if self.conjunto[v] == v:
        return v
    else:
        return self.Find(self.conjunto[v])

# [Union(A,B)] - Substitui os conjuntos A e B por um único conjunto contendo a sua união
def Union(self, vertice1, vertice2):
    vertice1 = self.Find(vertice1)
    vertice2 = self.Find(vertice2)
    if self.rank[vertice1] < self.rank[vertice2]:
        self.conjunto[vertice1] = vertice2
    elif self.rank[vertice1] > self.rank[vertice2]:
        self.conjunto[vertice2] = vertice1
    else:
        self.conjunto[vertice2] = vertice1
        self.rank[vertice1] += 1
```

Figura 12 | Métodos da estrutura de **`Union-Find`**.

Começando pelos métodos auxiliares da estrutura de **`Union-Find`** (Fig.12), optámos por implementar como métodos da **`Classe Graph`**, tal como solicitado.

Assim, criámos dois dicionários **`self.conjunto`** e **`self.rank`**, este último guarda os vértices como *keys* e o *ranks* dos mesmos como *values*.

Começando pelo método **`MakeSet`** , é dado um vértice **`v`** que será guardado como *key* e *value* no **`self.conjunto`** e como *key*, com *value* **`0`** no **`self.rank`**. Assim cumpre-se o objetivo de criar um conjunto singular com um dado vértice, sendo este as *keys* do dicionário.

No método, **`Find`** , é dado um vértice **`v`** que será pesquisado recursivamente no **`self.conjunto`** sendo retornado, quando é encontrado.

Por fim, o método **`Union`**, recebe como input os vértices **`vertice1`** e **`vertice2`** que serão procurados e substituídos por um único conjunto contendo a sua união. Neste processo escolhemos qual dos vértices será o "líder" do conjunto. Este *rank* tem por base a quantidade de vértices adjacentes que o mesmo possui. Assim, se o **`self.rank[vertice1] < self.rank[vertice2]`** , isto é, se o *rank* associado ao

vertice1 for menor do que o associado ao **vertice2** , então a união ocorre no conjunto do **vertice1** - **self.conjunto[vertice1] = vertice2** e vice-versa.

Caso contrário, quando os *ranks* são iguais, a união dá-se no conjunto do **vertice2** **self.conjunto[vertice2] = vertice1** e o **self.rank[vertice1]** incrementa 1 unidade.

```python
# [b] - Implementação do Algoritmo de Kruskal
@property
def Kruskal(self):
    # Inicialização de n conjuntos com todos os vértices do Grafo
    for vertice in self.vertices:
        self.MakeSet(vertice)

    # Cria uma lista de arestas
    lista_de_arestas = []
    soma = 0
    for edge in self.edges:
        soma += int(edge.peso)
        lista_de_arestas.append(edge)
    print('Peso Total ->', soma)

    # Ordena as arestas pelo peso
    lista_de_arestas.sort(key=lambda x: int(x.peso))

    # Cria a Árvore de Cobertura Mínima
    MST = Graph()
    soma = 0
    for edge in lista_de_arestas:
        if self.Find(edge.origem) != self.Find(edge.destino):
            self.Union(edge.origem, edge.destino)

            v1 = MST.insert_vertex(edge.origem.id)
            v2 = MST.insert_vertex(edge.destino.id)

            MST.insert_edge(v1, v2, int(edge.peso))

            soma += int(edge.peso)
    print("Peso Kruskal ->", soma)
    return MST
```

Figura 13 | Implementação em Python do Algoritmo de **Kruskal**.

No caso da implementação do Algoritmo de **Kruskal** seguimos o referido no pseudocódigo descrito anteriormente, utilizando como estrutura para guardar as arestas do grafo, uma **list**. No final, este retorna uma **MST** do tipo **Graph**, cujas arestas formam o caminho/ligação com menor peso associado possível.

Experiência do Kruskal no Grafo Teste

Para testar o algoritmo implementado, começámos por resolver primeiro o `Kruskal` no grafo criado para teste, recorrendo ao site [4], e cujo a solução se apresenta de seguida (Fig. 14).

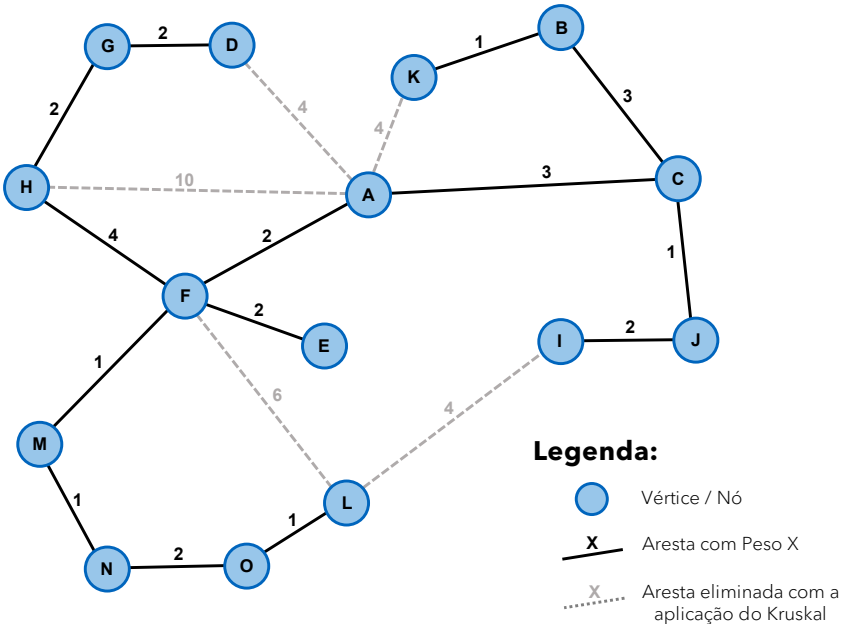

Figura 14 | Possível solução do Kruskal no Grafo Teste.

Logo de seguida, testámos então na implementação realizada em Python, e, para tal, acrescentámos ao teste da função `construir_grafo_teste()` o seguinte excerto de código, de modo a criar um grafo com as arestas aquando da realização do *Kruskal* (Fig. 15).

```
print(f'\nNetworkX Graph With Kruskal...')
nx_graph_Kruskal = nx.Graph()
for e in g.Kruskal.edges:
    nx_graph_Kruskal.add_edge(str(e.origem.id), str(e.destino.id), weight=int(e.peso))

# Representação
print(f'\nDrawing NetworkX Graph...')
nx.draw(nx_graph_Kruskal, node_size=500, with_labels=True, font_size=15, edge_color="#525252", font_weight='bold',
        node_color="#43b3ae", font_color="black", node_shape="o", alpha=0.7, linewidths=4)
plt.show()
# ----------------------------------------------------------------------------------------------------------------
```

Figura 15 | Implementação da função `construir_grafo_teste()` e respetiva utilização do código.

Por fim, ao correr 2 vezes o código obteve-se os grafos da Figura 16.

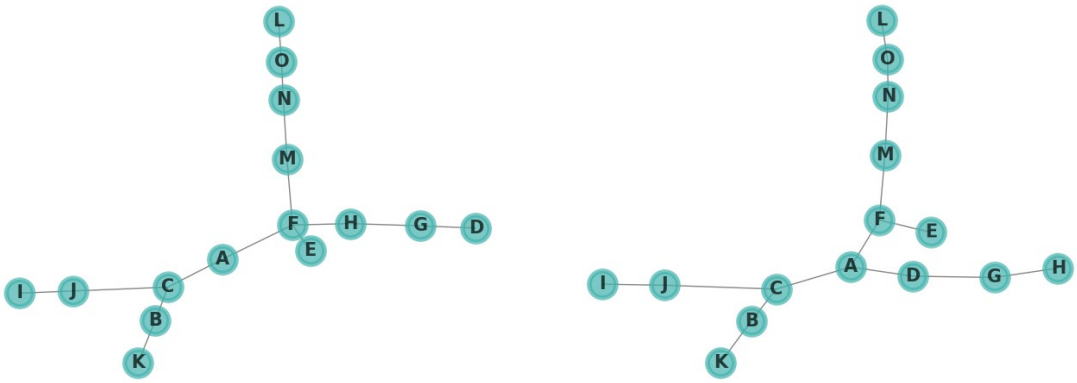

Figura 16 | Aplicação do método `Kruskal` no Grafo Teste e respetiva visualização.

Comparando o resultado do Python (Fig. 14) com as soluções do `Kruskal` (Fig. 16), pode-se verificar que a implementação realizada vai de encontro ao pretendido.

Tal como seria de esperar, pode haver várias soluções para o `Kruskal`, uma vez que poderá haver vários caminhos que contemplem o menor peso possível, tal como se pode verificar nos resultados obtidos.

Significa assim, que as duas Árvores de Cobertura Mínima que formam um único caminho/ligação, com o menor peso associado e com todos os vértices incluídos, são as apresentadas na Figura 16.

Assim sendo, validámos o resultado obtido, confirmando a funcional implementação do **Algoritmo de Kruskal** pelo que dêmos por terminada esta fase do trabalho.

4. Agrupamento Hierárquico *(Clustering)*

Clustering

O *Clustering* é o conjunto de técnicas de análise de dados *(Data Mining),* que visa agrupar dados segundo o seu grau de semelhança. O critério de semelhança difere de algoritmo para algoritmo, e depende do problema em questão. A cada conjunto de dados resultante do processo dá-se o nome *Cluster*. [8]

Apesar de existirem diferentes tipos de *Clustering*, iremos focar a nossa atenção apenas no Hierárquico.

Clustering Hierárquico

O Agrupamento Hierárquico é o método mais popular e amplamente utilizado para analisar dados em Redes Sociais. Neste método, os nós são comparados entre si com base nas suas semelhanças. Os *clusters* maiores, são então construídos através da junção de nós muito semelhantes. [9]

Existem 2 critérios para comparar nós, sendo eles o critério com a abordagem aglomerativa, em que cada nó representa um único *cluster* e começam a unir-se com base nas suas semelhanças; e outro com abordagem divisiva, em que todos os nós pertencem ao mesmo *cluster*, que irá, posteriormente, dividir-se. [9]

Single Linkage k-Clustering

Quer numa abordagem aglomerativa, quer numa abordagem divisiva, é necessário ter me consideração a distância entre dois vértices. Para medir essa distância, há a necessidade de desenvolver um algoritmo bem definido. Para isso, há então cinco abordagens diferentes, geralmente conhecidas como métodos de ligação *(Linkage)*, que podem ser utilizadas. [10][11]

No caso deste projeto, aqui documentado, foi-nos pedido que utilizássemos o algoritmo do *Single Linkage k-Clustering*. Este algoritmo consiste, numa visão aglomerativa, na minimização da distância vértices de *clusters*. [12][13]

$$\boldsymbol{D(t,k) = min(D(r,k), D(s,k))}$$

em que $\boldsymbol{D(t,k)}$ é a distância entre o novo cluster, $\boldsymbol{t}$, e o cluster existente, $\boldsymbol{k}$, em que $\boldsymbol{r}$ e $\boldsymbol{s}$ são dois clusters unidos, formando $\boldsymbol{k}$. [13]

Neste algoritmo, o que é feito é medir a distância mínima entre objetos e, encontradas essas distâncias, agrupa-os de forma a criar comunidades. Deste modo, deixa apenas por agrupar os objetos que têm distâncias máximas entre si.

Nesta última inferência, está a resolução na visão **divisiva** do *Single Linkage k-Clustering*. Isto é, se apenas queremos deixar as arestas com menor distância, podemos dividir as $\boldsymbol{k-1}$ maiores arestas de um grafo, resultando em $\boldsymbol{k}$ *clusters* deste.

É neste sentido que o nosso trabalho avança, uma vez que o `Kruskal` resulta numa **Árvore de Cobertura Mínima**, podendo esta ser agrupada em $\boldsymbol{k}$ *clusters*. [11]

Funcionamento da Função de *Single Linkage k-Clustering*

Para implementar o Algoritmo de *Single Linkage k-Clustering*, tivemos por base o seguinte pseudocódigo (**Fig.17**).

Single Linkage k-clustering (U, k):

//Input: U = {u_1, ..., u_n}, inteiro k > 1

//Output: K-clustering de U

- Inicializar um *grafo completo* com U como conjunto de vértices, G=(U,E), e E conjunto de arestas com peso d(u_i, u_j)
- M = Kruskal(V, E)
- Repetir k-1 vezes:
  remover a aresta com maior peso de M
- Devolver as componentes conexas da floresta M

Figura 17 | Pseudocódigo do Algoritmo de `Single Linkage k-Clustering`.

Esta forma de resolução baseia-se numa visão divisiva deste tipo de *Clustering*, tal como é descrito anteriormente. Isto é, dado que foi implementado o *Kruskal*, este devolve a **Árvore de Cobertura Mínima (**MST**)**, não sendo necessário o cálculo do mínimo de distâncias.

Em vez disso, para criar os $\boldsymbol{k}$ *clusters* de forma aglomerativa, **removemos $\boldsymbol{k-1}$ arestas com maior peso (**valor considerado como distância entre grafos**)**, dividindo, deste modo, o grafo obtido no *Kruskal*. Assim sendo, é necessária a criação de uma estrutura para ordenar as arestas pelo seu peso. Por fim é devolvido o grafo resultante deste processo.

É importante realçar que **esta fase não pode ser realizada caso o grafo não seja conexo**, uma vez que só faz sentido dividir um grafo em $\boldsymbol{k}$ *clusters*, se o mesmo inicialmente tiver $\boldsymbol{k=1}$, isto é, se for um grafo conexo, em que todos os vértices estejam ligados por um único caminho.

Implementação em Python da Função de *Clustering*

Transpondo este pseudocódigo para a linguagem Python, fizemos a seguinte implementação (Fig.18).

```python
def Cluster_Hierarquico(grafo, k):
    assert isinstance(grafo, Graph), "O objeto inserido não é do tipo Grafo!"

    # Cria uma lista de arestas
    lista_de_arestas = []
    for edge in grafo.Kruskal.edges:
        lista_de_arestas.append(edge)

    # Ordena as arestas pelo peso
    lista_de_arestas.sort(key=lambda x: x.peso)

    # Cria a Árvore de Cobertura Mínima
    MST = grafo.Kruskal

    # Remover k-1 arestas com MAIOR Peso
    for i in range(k - 1):
        MST.remove_edge(lista_de_arestas[-1].origem, lista_de_arestas[-1].destino)
        lista_de_arestas.remove(lista_de_arestas[-1])

    return MST
```

Figura 18 | Implementação do **`Single Linkage k-Clustering`** em Python.

Nesta, começámos por importar as arestas do grafo, após realizar o **`Kruskal`**, para uma *list*, uma vez que esta estrutura permite a ordenação, operação realizada de seguida.

Seja **`MST`** a Árvore de Cobertura Mínima do grafo dado como *input*, para criar $\boldsymbol{k}$ *clusters,* removemos $\boldsymbol{k-1}$ arestas com maior peso através de um ciclo **`for`**, usando o método **`MST.remove_edge(lista_de_arestas[-1].origem, lista_de_arestas[-1].destino)`** e por fim removemos essa mesma aresta da lista de arestas criada **`lista_de_arestas.remove(lista_de_arestas[-1])`**.

Experiência do *Clustering* no Grafo Teste

Correndo este algoritmo no grafo teste criado, obtivemos os seguintes resultados.

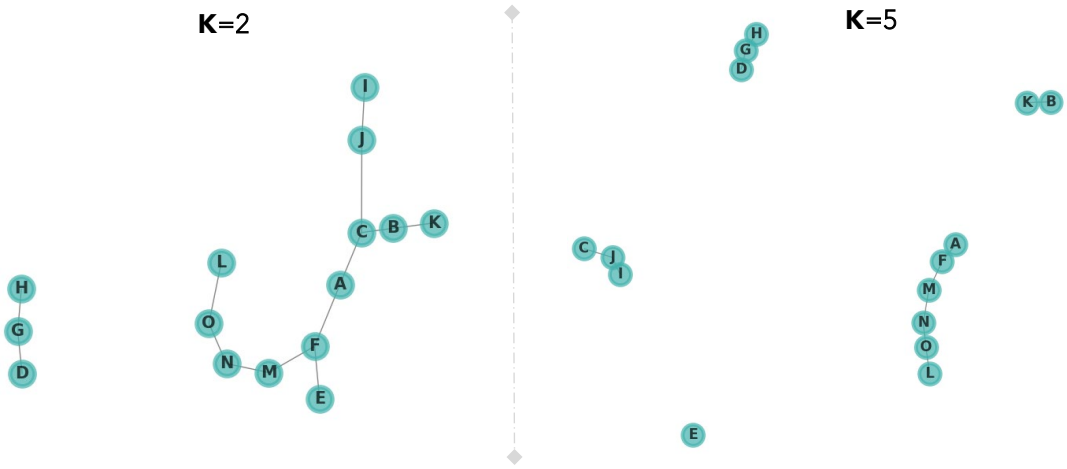

Figura 19 | Aplicação do método de **Clustering** no **Grafo_Teste.csv** e respetiva visualização.

A tabela ao lado (Fig.20) representa as arestas do Grafo Teste com as arestas ordenadas, decrescentemente, pelo seu Peso.

Anteriormente à aplicação do método de **Clustering** no **Grafo_Teste.csv**, as arestas que se encontram a cinzento-escuro (Fig.20), já tinham sido eliminadas pelo algoritmo de *Kruskal*. No entanto, ambas as arestas a cinzento-claro permitem a construção da MST, logo apenas uma vai ser eliminada de forma aleatória.

Podendo encontrar esta aleatoriedade no **Kruskal**, o mesmo se pode suceder no **Clustering**, uma vez que existem várias arestas com o mesmo peso, como são exemplo as arestas **AC** e **CB**. A aleatoriedade provém da ordenação da lista de arestas, em função do seu peso associado. Existindo arestas com os mesmos pesos, ao invocarmos o **Kruskal** e o **Clustering**, o algoritmo escolhe de forma aleatória, uma delas.

X	Y	Peso
A	H	10
F	L	6
A	D	4
A	K	4
H	F	4
L	I	4
A	C	3
C	B	3
A	F	2
D	G	2
H	G	2
F	E	2
N	O	2
I	J	2
C	J	1
K	B	1
F	M	1
M	N	1
L	O	1

Legenda

- Arestas eliminadas pelo *Kruskal*
- Alternativas de *Kruskal* (uma delas será escolhida, a outra eliminada)
- Arestas que podem ser eliminadas no *Clustering*

Figura 20 | Tabela das Arestas do **Grafo_Teste.csv** ordenadas por ordem decrescente de Peso.

5. Construção do TAD Grafo com o *Dataset*

Metodologia CRISP-DM

Para melhor tirar partido do *dataset* fornecido acerca da Rede Social *Facebook,* decidimos que deveríamos aplicar a metodologia CRISP-DM. Esta é formada por seis fases: *Business Understanding*, *Data Understanding*, *Data Preparation*, *Modeling*, *Evaluation* e *Deployment*. [14]

Business Understanding | Data Understanding

Primeiramente, na fase do *Business* e *Data Understanding*, o objetivo foi perceber os dados do *dataset* fornecido e a sua aplicação para o objetivo do trabalho. Assim, constatámos que cada linha do ficheiro `Data_Facebook.csv` correspondiam aos utilizadores – `x` , `y` – e ao número de interações de `x` para `y`.

Este significado denota que os vértices origem e destino na base de dados fornecida tem por base um grafo direcionado. Deste modo, detetámos um erro colossal na base de dados, que impediria cumprir com sucesso o objetivo do trabalho caso usássemos a base de dados como está, uma vez que o pressuposto inerente ao Algoritmo de *Kruskal* e, posterior, *Clutering* é utilizar um grafo cujas arestas não sejam direcionadas, impedindo que haja arestas paralelas.

Data Preparation

Na fase de *Data Preparation* arranjámos os dados para a fase de modelação. Para tal, resolvemos o problema de arestas paralelas, tonando-as não direcionadas. Optámos então por criar uma função extra `limpar_dataset()` que usa a biblioteca `Pandas`.

Primeiramente, tivemos de preparar a base de dados para melhor manuseio no `Pandas`, pelo que, criámos uma coluna com o `Id` de cada Observação/Aresta, para facilitar a indexação, e concatenámos a informação numa só linha, separada por virgulas, mantendo-se o formato do arquivo original (`csv`). Para este processo, usámos o **Excel**, tal como ilustrado na Figura 21.

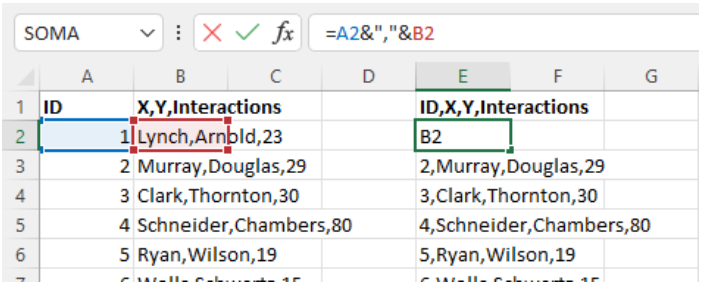

Figura 21 | *Data Preparation* no **Excel** da base de dados `Data_Facebook.csv`.

Deste processo resultou o ficheiro `Data_Facebook_Excel.csv`.

Implementámos então a função `limpar_dataset()` função tal como observável na Figura 22.

```
# ================================================ [DATASET] ================================================
def limpar_dataset():
    data = pd.read_csv(r'Data_Facebook_Excel.csv')
    i = 0
    while i < len(data):
        print(i)
        j = 1
        while j < len(data):
            if data['X'][i] == data['Y'][j] and data['X'][j] == data['Y'][i]:

                print("i=", i, data['X'][i], data['Y'][i], data['Interactions'][i], '|',
                      "j=", j, data['X'][j], data['Y'][j], data['Interactions'][j])

                data['Interactions'][i] += int(data['Interactions'][j])
                print(data.Interactions[i])

                data = data.drop(labels=j, axis=0)
                data = data.reset_index(drop=True)
                j += 1
            else:
                j += 1
        i += 1

    return data.to_csv("Data_Facebook_Alterado.csv")
```

Figura 22 | Implementação em Python da função `limpar_dataset()`.

Nesta usámos as potencialidades da biblioteca `Pandas` ,tal como já referido, criando um `DataFrame`, e ao percorrê-lo identifica as arestas paralelas, eliminando-as, resultando o peso da aresta "original" na adição do peso da aresta paralela. Fica assim corrigido um dos cruciais problemas da nossa base de dados, impossibilitador de realizar os Algoritmos de *Kruskal* e de *Clustering.*

A escolha de limpar o *dataset* antes de inserir no grafo foi em virtude da metodologia seguida, permitindo assim, uma limpeza mais otimizada, sendo a complexidade deste método inferior, em oposição ao que poderia ser implementado aquando do carregamento da informação para o grafo.

Posteriormente, retirámos a informação excedente dessa base de dados, eliminado a coluna relativa ao **`Id`**, uma vez que o próprio **`DataFrame`** do **`Pandas`** cria uma coluna para tal informação, resultando no ficheiro **`Data_Facebook_Alterado.csv`**

Após este processo, visualizamos o grafo obtido, sendo o resultado visível de seguida (Fig.23).

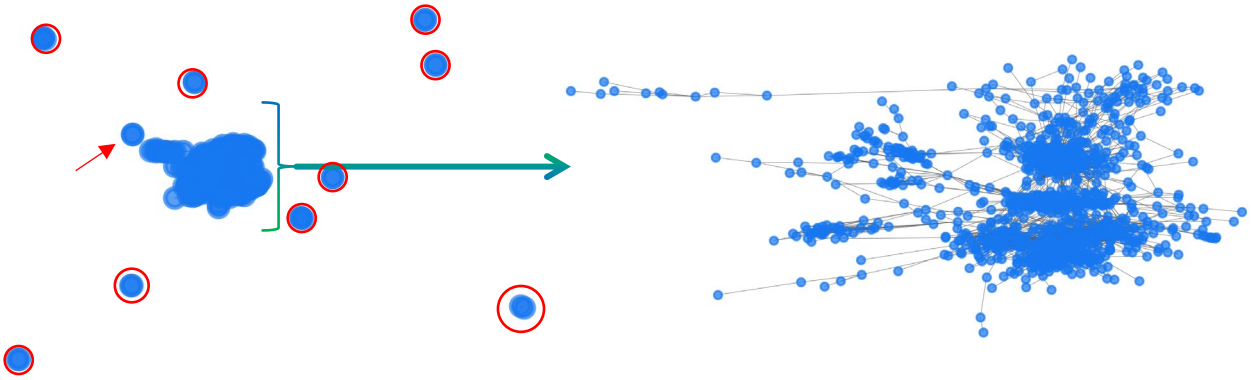

Figura 23 | Visualização do grafo da base de dados **`Data_Facebook_Alterado.csv`**

Tal como é possível observar o garfo obtido não é um único grafo conexo, pelo que optámos por eliminar todas as conexões fora do grafo central. Este processo foi feito individualmente, uma vez que eram poucos os grafos individuais, tal como assinalados na Fig. 23. Deste modo, resolvemos o segundo problema da base de dados, que impossibilitaria dividir ***k clusters***, sendo que havia já grupos separados.

Deste processo resultou então uma base de dados **`Data_Facebook_Alterado_SemNConexos.csv`** constituída por 7223 observações, - arestas com o **`X`**, **`Y`** e **`Interactions`** associado - criando assim o grafo apresentado de seguida.

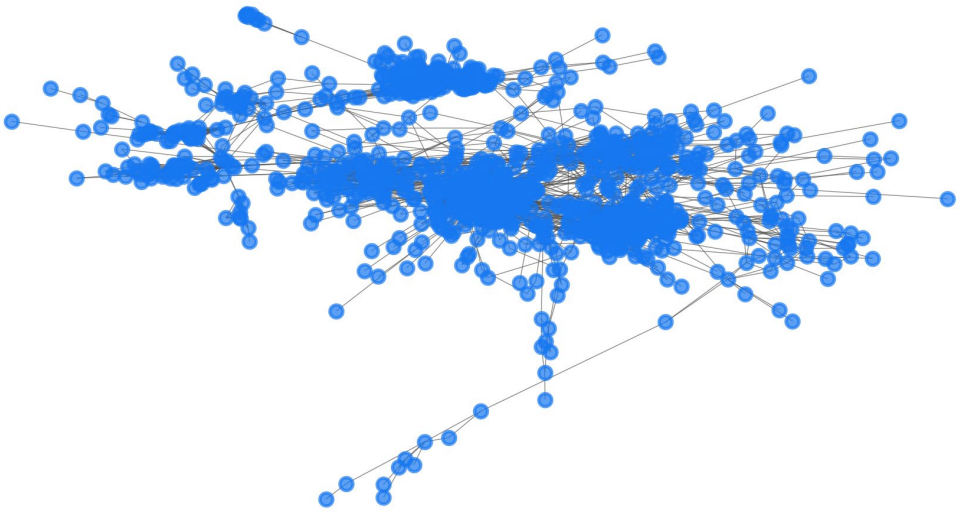

Figura 23 | Visualização do grafo da base de dados `Data_Facebook_Alterado_SemNConexos.csv`

Modeling

Na fase de *Modeling*, de modo a cumprir o objetivo do trabalho, testámos a função de *Clustering*. Esta técnica integra-se nas aprendizagens não supervisionadas e serve, essencialmente, para encontrar padrões ligações intrínsecas em dados. Estes padrões permitem explicar comportamentos e podem ser utilizados para fazer inferências.

Neste caso prático, o agrupamento dos utilizadores em *Clusters* permite estudar os *profiles* e *circles* em *Facebook*, criando sub-comunidades.

Evaluation

Por fim, *Evaluation* é a etapa onde verificamos se respondemos ao objetivo principal e apresentamos os resultados, acrescidos de métricas que os caracterizam.

Construção do Grafo com os Dados do Facebook

```python
def construir_grafo_Facebook(filename):
    print('[Início] - Construir um grafo...')

    # Inicialização da Classe Graph
    g = Graph()

    # Importar os valores do CSV - [VÉRTICES]
    f = open(filename, 'r')
    data = csv.reader(f, delimiter=',')
    headers = next(data)
    print(f'\nReading... {headers}')

    i = 1
    for row in data:
        user_follower, user_followed, interactions = row[1], row[2], row[3]

        print(i, '-----', user_follower, user_followed, interactions)

        v1 = g.insert_vertex(str(user_follower))
        v2 = g.insert_vertex(str(user_followed))

        g.insert_edge(v1, v2, int(interactions))

        i += 1
    f.close()

    return g
```

Figura 22 | Implementação em Python do método **`construir_grafo_Facebook()`**

Foi implementada a função **`construir_grafo_Facebook()`** (Fig.21) que importa os valores da base de dados disponibilizada para a realização do trabalho. Seguidamente, dá print ao nome de cada um dos usuários e a respetiva pessoa que ele segue, mais o número de interações entre ambos.

Nessa função, replicámos a implementação da função **`construir_grafo_teste()`** permitindo assim visualizar o grafo. Foi esta a função usada para visualizar os grafos na fase de *Data Preparation*.

Visualização do Grafo

Depois de correr o código, obtivemos o grafo seguinte, aplicando o algoritmo de *Kruskal*.

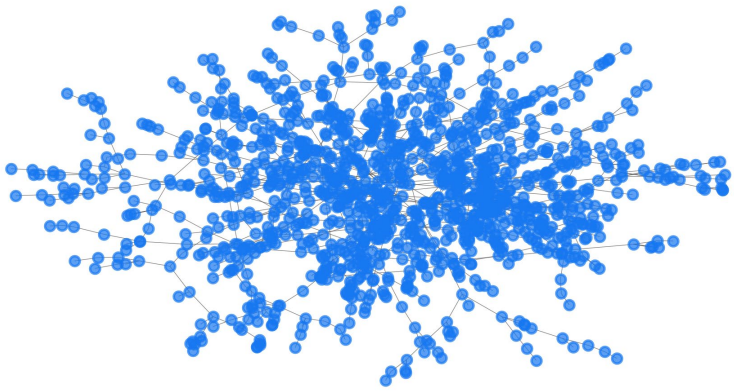

Figura 23 | Resultados obtidos no método **`construir_grafo_Facebook`**

Estudo de Comunidades

Ao aplicarmos o **`Clustering`** com **`k=5`** para dividir em comunidades o grafo obtido apresenta-se de seguida.

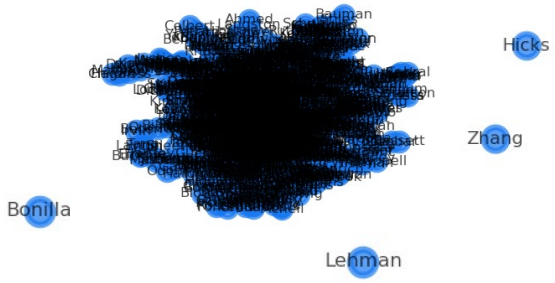

Figura 24 | Grafo visualizado quando aplicamos o **`Clustering`** com **`K=5`**.

Ao analisá-lo, verificámos que o mesmo realiza bem o **`Clustering`**, uma vez que as arestas removidas são verdadeiramente aquelas com maior peso, tal como é pretendido neste algoritmo.

Assim, e variando o **`k`** observámos que as "comunidades" criadas têm apenas um *user* do *Facebook,* não havendo qualquer interesse ou informação relevante dividir as comunidades e posteriormente estudar as sub-comunidades.

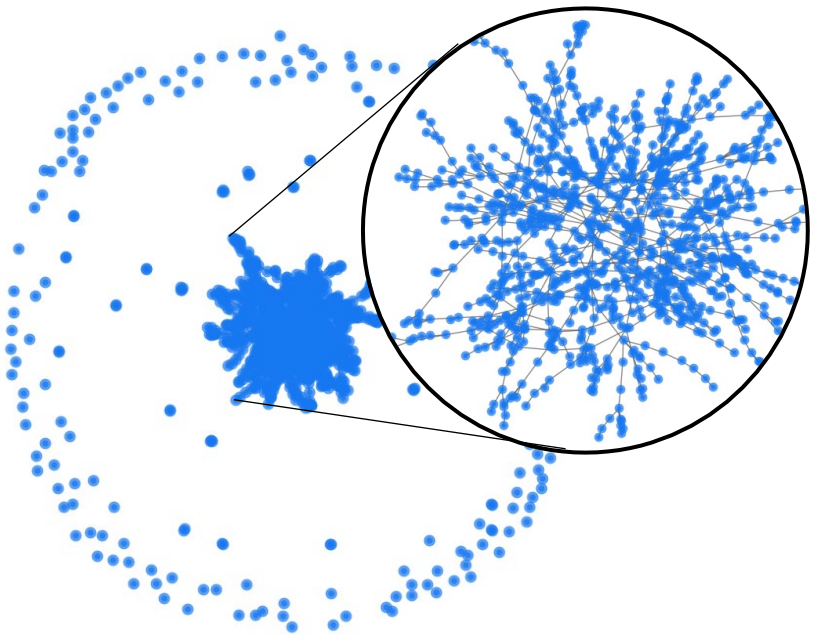

Figura 25 | Grafo visualizado quando aplicamos o **`Clustering`** com **`K=200`**.

Esta mesma ideia é fruto dos vários testes que fizemos na função **`Cluster_Hierarquico`**, as quais iterámos com diferentes **`k`** para tentar obter a melhor divisão de comunidades possível. Após iterarmos com **`k=200`** obtivémos os resultados apresentados na Figura 25, no qual é visível uma comunidade central e os restantes *users* separados de forma orbital.

É de notar que na comunidade central obtida estão representadas ligações entre os *users* com menos interações, uma vez que as ligações desassociadas são aquelas que apresentam um maior peso.

Esta remoção tem por base o que está referido no ficheiro que vinha acompanhado pelos dados **`Facebook_READ_ME`**, no qual é dito que o peso dos *users* é relativo ao número de interações que os mesmos possuem. Ao mudar o significado desta variável, iriamos estar a manipular os dados e a enviesar a realidade que os mesmos representam.

Acrescentar ainda que, tendo por base os resultados obtidos, não nos parece ser racional 2 *users/profiles* com elevado número de interações encontrarem-se em comunidades diferentes.

Métricas para o Estudo das Comunidades

Devido ao mencionado anteriormente, optámos por estudar apenas o grafo como uma única comunidade e, para tal, usámos 3 métricas possíveis, o *Degree Centrality*, o *Closeness Centrality* e o *Betweenness Centrality.*

Note-se que nos gráficos apresentados neste secção, quanto maior a dimensão do vértice/nó, maior é o valor da métrica estudada.

Degree Centrality

A centralidade de grau é a medida mais simples de calcular, sendo a contagem de quantas conexões sociais (arestas) que um vértice possui. [15]

Quanto maior o grau, mais central é o nó. Deste modo, podemos criar uma analogia com a moda (medida de tendência central que indica o número com maior frequência num conjunto). Neste caso, a centralidade de grau indica o vértice que tem mais conexões.

No caso da biblioteca `NetworkX`, os valores resultantes da centralidade de grau `degree_centrality()` são normalizados dividindo-se pelo grau máximo possível em um grafo simples $\boldsymbol{n-1}$, onde $\boldsymbol{n}$ é o número de nós em $\boldsymbol{G}$. [16]

Ao aplicarmos esta métrica ao grafo do *Facebook*, podemos obter uma medida de popularidade, que revela os utilizadores mais importantes e influentes, isto é, que tem mais interações com os restantes utilizadores.

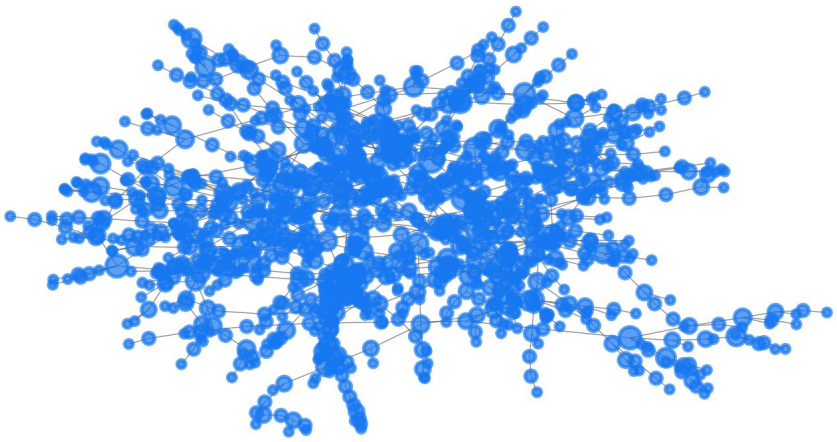

Figura 26 | Visualização do grafo representativo do `Degree Centrality`.

Através de uma breve análise do resultado obtido com o *dataset* em estudo (Fig.26), pode-se verificar que há pouca heterogeneidade nos vértices, relativamente

à centralidade de grau, podendo por isso inferir que os *users* desta base de dados têm uma semelhante influência perante as suas conexões.

Havendo raras exceções, é de salientar que aqueles vértices que apresentam maior centralidade de grau contém o *user* **`Zapata`** e o *user* **`Grace`** (0,00633). No lado oposto, um dos que apresenta menor valor é o **`Simms`** (0,00079).

Closeness Centrality

A centralidade de proximidade é um indicador que representa o quão próximo um nó está de todos os outros nós da rede. É calculado como a média do comprimento do caminho mais curto do nó, para todos os outros nós. [17]

Através do seu uso, podemos concluir, por exemplo, o que quão rápido uma notícia criada por um *profile* demoraria a ser espalhada para os restantes *users*. [17]

Ao usarmos o **`NetworkX`**, o método **`closeness_centrality()`** calcula a distância do caminho mais curto entre $\boldsymbol{u}$ e todos os $\boldsymbol{n-1}$ nós alcançáveis. [18]

$$C(u) = \frac{n-1}{\sum_{v=1}^{n-1} d(v,u)}$$

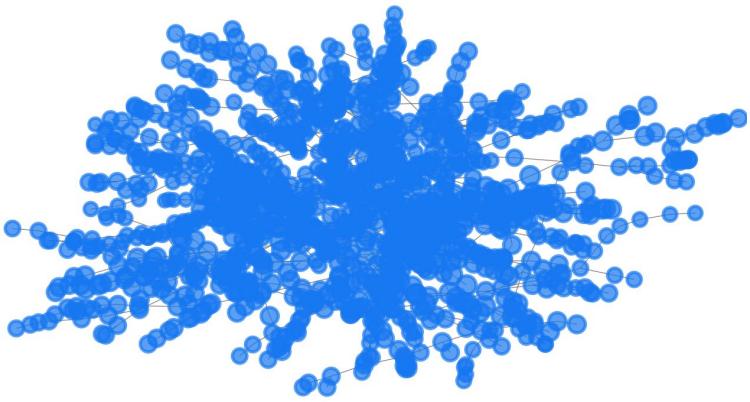

Figura 27 | Visualização do grafo representativo do **`Closeness Centrality`**.

À semelhança do *Degree Centrality*, no caso em estudo, verificou-se que existe uma forte homogeneidade na representação da centralidade de proximidade. Isto verifica-se uma vez que os vértices representativos dos *profiles* possuem todos dimensões idênticas e não é possível fazer uma distinção clara.

Observa-se que valores mais altos de centralidade de proximidade coincidem com maior centralidade de grau. Deste modo, o vértice que apresenta maior *closeness centrality* é o vértice referente ao **Cheng** (0,05458), e, de outra perspetiva, um dos que apresenta menor *closeness centrality* , é o *profile* **Sanchez** (0,0222).

Betweenness Centrality

A centralidade de intermediação é uma medida amplamente utilizada que revela o papel de uma pessoa ao permitir que as informações passem de uma parte da rede para a outra. [19]

Assim, transpondo a ideia para o caso das comunidades do *Facebook*, quanto mais as pessoas dependem de um utilizador para fazer conexões com outras pessoas, maior se torna a centralidade de intermediação desse utilizador. [19]

Relativamente a este indicador, o `NetworkX` calcula a centralidade de intermediação do vértice/nó $\boldsymbol{v}$ como a soma da fração de todos os pares de caminhos mais curtos que passam por $\boldsymbol{v}$. [20]

$$c_B(v) = \sum_{s,t \in V} \frac{\sigma(s,t|v)}{\sigma(s,t)}$$

Onde $\boldsymbol{V}$ é o conjunto de nós, $\boldsymbol{\sigma(s,t)}$ o número de menores caminhos entre $\boldsymbol{(s,t)}$ e $\boldsymbol{\sigma(s,t|v)}$ é o número desses caminhos que passam por algum nó $\boldsymbol{v}$ outro que não seja $\boldsymbol{s}$. Se, $\boldsymbol{s = t, \sigma(s,t) = 1}$, e se, $\boldsymbol{v \in s,t,\ \sigma(s,t|v) = 0}$.

Assim, ao usar o método `betweenness_centrality()` do `NetworkX`, obtivemos o grafo presente na Figura 28.

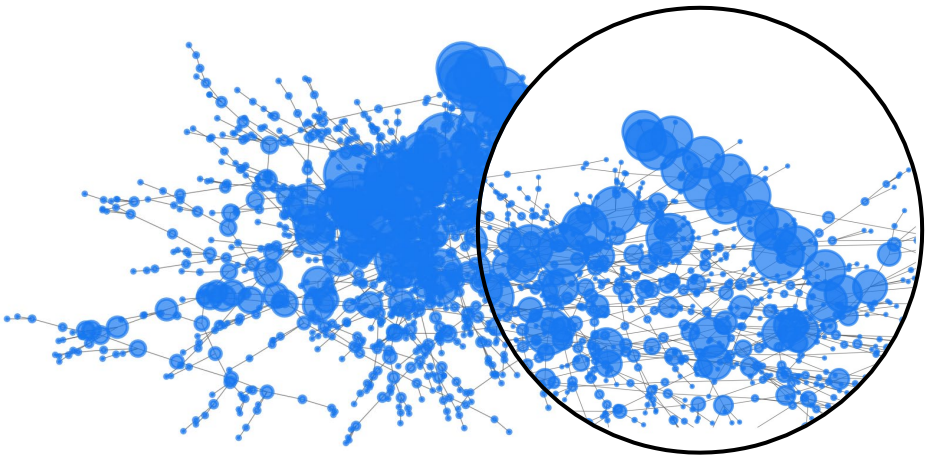

Figura 28 | Visualização do grafo representativo do `Betweenness Centrality`.

Neste grafo, a dimensão de um vértice/nó varia consoante o número de vezes que um nó atua como uma ponte ao longo do caminho mais curto entre dois outros nós. Ou seja, quantos mais utilizadores passarem por um vértice para chegar a outro, maior será a dimensão deste.

Verifica-se assim que o vértice com maior centralidade de intermediação é o *user* **`Cheng`** e um dos que apresenta menor valor é o **`Simms`** (0, 00158).

De um modo geral, a aplicação das métricas de centralidade de grau e de proximidade no grafo com os dados do *Facebook* foram pouco interessantes de estudar, uma vez que havia pouca diversidade de valores. Porém, com uma melhor base de dados, os algoritmos implementados estão capacitados a recebe-la, possibilitando uma melhor extração de informação da mesmo.

Conclusões

Conclui-se com este trabalho, apesar dos resultados pouco relevantes, a importância do estudo das redes sociais, agrupando pessoas em comunidades.

Foi-nos possível estudar e conhecer as estratégias de construção de grafos, o tratamento de bases de dados, algoritmos que resultam na árvore de cobertura mínima, e algoritmos de agrupamento hierárquico.

O grafo é uma estrutura não linear que permite-nos resolucionar diversos problemas do quotidiano, facilitando a criação de redes com ligações de objetivos variados, apartir de um *dataset*, com várias potencialidades de uso.

Para melhor orientação do trabalho, usámos os conhecimentos adquiridos em outras UCs (como *Dados na Ciência, Gestão e Sociedade*; *Análise Exploratória de Dados*; e *Amostragem e Fontes de Informação*), tal como é exemplo, a metodologia adotada neste trabalho, CRISP-DM.

Relativamente às implementações em Python, poderiam, em trabalho futuro serem analisadas empírica e assintoticamente as complexidades espaciais e temporais dos vários algoritmos implementados.

Validámos todos os algoritmos com o grafo teste criado, permitindo assim, verificar a sua correta implementação.

Apesar dos resultados obtidos terem sido de fraca relevância, acreditamos que o trabalho desenvolvido tenha qualidade, para que num projeto futuro, com uma base de dados diferente, haja resultados mais promissores.

Assim, denotamos algumas dificuldades e obstáculos que encontrámos ao longo do trabalho:

Em primeiro lugar, deparámo-nos com a dificultade de implementar o *hash* do vértice da forma correta, em consonância com o grafo pretendido. Este fazia com que os vértices fossem representados mais vezes do que as supostas. Dado que a nossa implementação não detetava a já existência dos vértices, aquando da adição das arestas, percebemos que, para arestas ligadas a um mesmo vértice, este não era identificado pelo seu conteúdo, impossibilitando a sua visualização correta.

De seguida, após a implementação de ambos os algoritmos solicitados, deparámo-nos com alguns erros os quais acreditámos serem fruto das nossas implementações. Porém, o problema residia sobre a base de dados do *Facebook* fornecida, uma vez que não se encontrava de acordo com o objetivo deste trabalho. Verificámos então a existência de arestas paralelas, que obstaculizou a progressão do nosso trabalho.

Após ser reportado este problema aos docentes, apercebemo-nos que as nossas implementações estavam corretas, tendo sido um fator que nos levou a perder tempo crucial de forma desnecessária.

Por último, e depois de resolvidos todos os entraves a este trabalho, é possível concluir através dos resultados obtidos que a base de dados com a qual trabalhamos neste projeto é de fraca qualidade, pois não permite o estudo efetivo de comunidades e sub-comunidades.

Em suma, com a realização deste trabalho foi-nos permitido aplicar uma das estruturas de dados fulcrais, o grafo.

Bibliografia

[1] M. T. Goodrich, M. H. Goldwasser, and R. Tamassia, *Data Structures and Algortihms in Python*. New York: Wiley, Cop, 2013.

[2] S. Nanawati, "Social Network Analytics," *Analytics Vidhya*, Aug. 02, 2019. https://medium.com/analytics-vidhya/social-network-analytics-f082f4e21b16 (acessado a 22 de maio de 2022)

[3] "NetworkX 2.8.2 Documentation" *networkx.org*. https://networkx.org/documentation (acessado a 22 de maio de 2022).

[4] "Kruskal's Algorithm," https://algorithms.discrete.ma.tum.de/graph-algorithms/mst-kruskal/index_en.html (acessado a 24 de maio de 2022).

[5] American Mathematical Society, *Proceedings of the American Mathematical Society*. Menasha & New York, 1956, pp. 48-50.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms,* 3rd ed. The MIT Press, 2009, pp. 624–633.

[7] S. Dasgupta, C. H. Papadimitriou, and Umesh Virkumar Vazirani, *Algorithms*. Boston: Mcgraw-Hill Higher Education, 2006, pp. 133–143. Acessado a: 29 de maio, 2022. [Online]. Available: http://algorithmics.lsi.upc.edu/docs/Dasgupta-Papadimitriou-Vazirani.pdf

[8] T. Segaran, *Programming Collective Intelligence: Building Smart Web 2.0 Applications*. Sebastopol, Calif.: O'reilly, 2007, pp. 33-53, 297–298. Acessado a: 29 de maio, 2022. [Online]. Available: https://axon.cs.byu.edu/~martinez/classes/778/Papers/GP.pdf

[9] "Hierarchical Clustering - an overview | ScienceDirect Topics", *ScienceDirect Topics*. https://www.sciencedirect.com/topics/computer-science/hierarchical-clustering (acessado a 29 de maio de 2022).

[10] D. K. Koech, "Getting Started with Hierarchical Clustering in Python," *Engineering Education (EngEd) Program*. https://www.section.io/engineering-education/hierarchical-clustering-in-python/ (acessado a 29 de maio de 2022).

[11] X. Lv, Y. Ma, X. He, H. Huang, and J. Yang, "CciMST: A Clustering Algorithm Based on Minimum Spanning Tree and Cluster Centers," *Mathematical Problems in Engineering*, vol. 2018, Dec. 2018, doi: 10.1155/2018/8451796

[12] "2.3. Clustering — scikit-learn 0.20.3 documentation," *Scikit-learn.org*, 2010. https://scikit-learn.org/stable/modules/clustering.html (acessado a 29 de maio de 2022).

[13] "12.6 - Agglomerative Clustering | STAT 508," *PennState: Statistics Online Courses*. https://online.stat.psu.edu/stat508/lesson/12/12.6 (acessado a 29 de maio de 2022).

[14] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques*, 3rd edition. Burlington, Ma: Elsevier, 2012. Acessado a: 27 maio de 2022 [Online]. Disponível em http://myweb.sabanciuniv.edu/rdehkharghani/files/2016/02/The-Morgan-Kaufmann-Series-in-Data-Management-Systems-Jiawei-Han-Micheline-Kamber-Jian-Pei-Data-Mining.-Concepts-and-Techniques-3rd-Edition-Morgan-Kaufmann-2011.pdf

[15] "Degree Centrality - an overview | ScienceDirect Topics," *ScienceDirect*. https://www.sciencedirect.com/topics/computer-science/degree-centrality (acessado a 28 de maio de 2022).

[16] "degree_centrality – NetworkX 2.8.2 documentation," *NetworkX*. https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.centrality.degree_centrality.html (acessado a 28 de maio de 2022).

[17] "Closeness Centrality - an overview | ScienceDirect Topics," *ScienceDirect*. https://www.sciencedirect.com/topics/computer-science/closeness-centrality (acessado a 28 de maio de 2022).

[18] "closeness_centrality – NetworkX 2.8.2 documentation," *NetworkX*. https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.centrality.closeness_centrality.html (acessado a 28 de maio de 2022).

[19] "Betweenness Centrality - an overview | ScienceDirect Topics," *ScienceDirect*. https://www.sciencedirect.com/topics/computer-science/betweenness-centrality (acessado a 28 de maio de 2022).

[20] "betweenness_centrality – NetworkX 2.8.2 documentation," *NetworkX*. https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.centrality.betweenness_centrality.html (acessado a 28 de maio de 2022).