UNIVERSIDADE FEDERAL DE SÃO CARLOS
CIÊNCIA DA COMPUTAÇÃO

EXPLORANDO A LINGUAGEM PYTHON E A BIBLIOTECA NETWORKX PARA A SOLUÇÃO DE PROBLEMAS EM GRAFOS

TEORIA DOS GRAFOS

Bruna Fernandes Prates **RA: 743513**
Esther Calderan Hoffmann **RA: 743529**

Professor: Alexandre Levada

Sumário

1. Introdução

1.1. Origem da Teoria dos Grafos

A Teoria dos Grafos originou-se a partir de um problema presente na cidade de Konigsberg (Alemanha). Um rio atravessava a cidade dividindo-a em 4 partes e, para interligá-las, foram construídas 7 pontes. Desse modo, o desafio consistia em realizar um passeio percorrendo todas as 7, porém atravessando cada uma delas apenas uma única vez.

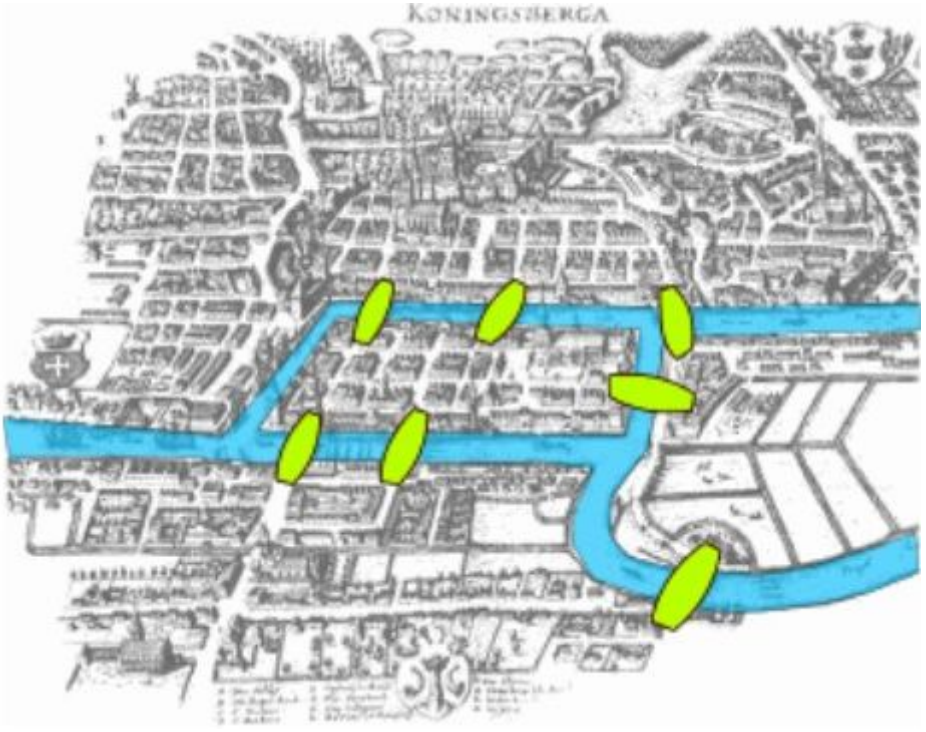

Figura 1.1: Konigsberg Bridges

Diante disto, o matemático e físico suíço Leonhard Euler (1707-1783), em 1736, provou que tal feito é impossível de realizar-se. Utilizou-se de um modelo simplificado das ligações entre as regiões para realizar seus estudo e, por fim, elaborou um teorema responsável por definir as condições necessárias para que o processo de percorrer cada ponte uma única vez seja factível. Através da busca de tal solução, Euler criou um artigo em que fundamentava-se uma nova área de Matemática: a Teoria dos Grafos.

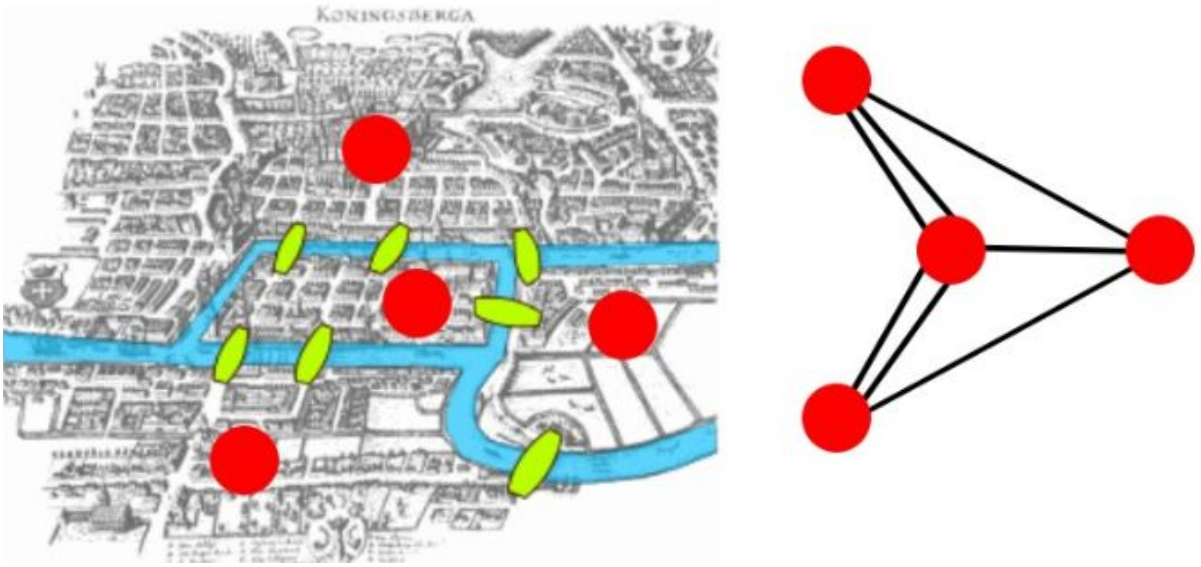

Figura 1.2: Representação simplificada das pontes de Konigsberg.

1.2. O que é um grafo?

Grafo é um modelo matemático que corresponde a representação gráfica de problemas e relações entre objetos. Ademais, é formado por um conjunto de vértices (V) interconectados através de arestas (E). De maneira mais formal, um grafo G equivale a um par (V, E) em que V é um conjunto finito, enquanto E é a relação de pares ordenados sobre V.

1.3. Utilização de grafos

É possível utilizar a modelagem e teoremas de grafos em diversas situações, tornando muito mais prático e rápido a análise de determinados problemas. Em redes de transporte, por exemplo, quando a conectividade entre cidades é fundamental, podemos representá-las através de vértices enquanto as arestas correspondem às conexões existentes, podendo armazenar a quilometragem. O mesmo pode ser aplicado em Mapas, calculando quais são os caminhos mais curtos e/ou mais baratos; Redes Elétricas para determinar sua melhor distribuição possível; Relacionamentos, estabelecendo amizades em comum como é feito na rede social Facebook; dentre diversos outros.

Vale destacar a grande importância deste modelo matemático para a computação, uma vez que sua lógica está fortemente presente em compiladores, rede de computadores, roteamento de pacotes de internet, auxiliar máquinas de busca a localizar informações na web, etc.

1.4. Contextualização

Este artigo foi construído baseando-se nos processos e resultados da elaboração de projetos escolhidos dentre as variadas opções fornecidas pelo docente da disciplina de Teoria dos Grafos, aplicada a turma de Bacharelado em Ciência da Computação da Universidade Federal de São Carlos (UFSCar). Tais projetos apresentam problemas a serem resolvidos utilizando-se os conhecimentos adquiridos na área de Grafos. Encontram-se neste relatório explicações a respeito do modelo matemático em questão, bem como seus algoritmos; códigos utilizados para alcançar o objetivo proposto pelas situações problema; esclarecimentos acerca de todo o passo a passo até a concepção final.

1.5. Linguagem Python

Seguindo as recomendações do professor, os algoritmos foram todos implementados utilizando-se a linguagem de programação Python. Esta é reconhecida pela sua dinamicidade, fácil interpretação, produtividade e legibilidade. Ademais, apresenta uma biblioteca padrão imensa, contendo múltiplas classes, métodos e funções para a execução de tarefas. Uma enorme vantagem em elaborar este projeto em linguagem Python está presente no grau de facilidade para entendimento por indivíduos que não estão habituados à programação; tornando, também, a implementação mais prática e eficaz para o programador.

1.6. Bibliotecas

- NetworkX:

Recomendada pelo professor Alexandre, essa biblioteca torna simples o trabalho com grafos em python. A partir de um grafo criado com ela é possível facilmente ter acesso à seus vértices ou arestas, usando “Grafo.nodes()” ou “Grafo.edges()”, respectivamente. Além disso a NetworkX é composta por diversas funções envolvendo grafos, como a maioria dos algoritmos que foram usados neste documento (Algoritmo de Prim, Busca em Largura, Busca em Profundidade, Dijkstra, etc).

A documentação desta foi intensamente utilizada no decorrer do projeto. Para mais informações e acesso a esta basta direcionar-se à “Documentação NetworkX” nas Referências.

- Matplotlib:

Biblioteca para gerar gráficos em python. Foi usada somente em conjunto da NetworkX para visualização dos grafos. Mais informações: “Matplotlib” em Referências

- NumPy:

Utilizada apenas para ler as matrizes de adjacências dos grafos, quando necessário. Esta biblioteca facilita o uso de matrizes para fins científicos. Mais infomações: “Numpy” em Referências.

1.7. Organização do documento

Para garantir boa interpretação a respeito de todo o procedimento de elaboração do projeto como um todo, este artigo foi dividido em seções específicas. A introdução funciona como uma apresentação geral de tudo que será discorrido, possuindo, também, o objetivo de transmitir algumas informações introdutórias para melhor entendimento do conteúdo que será apontado. Em seguida, a Fundamentação Teórica detalha as características que cada um dos projetos, bem como o que é esperado em sua resolução (algoritmos utilizados, perguntas que devem ser respondidas). O tópico Materiais e Métodos tem como finalidade compartilhar todo tipo de instrumento utilizado para que se tenha chego ao resultado final com êxito. Resultados e discussões apresentam uma explicação mais aprofundada sobre a forma com que o grupo elaborou o método de resolução do problema, além de exibir os resultados obtidos. As considerações finais têm grande importância para tomar conhecimento da visão dos autores a respeito das ferramentas utilizadas, problemas encontrados, vantagens e desvantagens das escolhas feitas. Por fim, as Referências promovem os devidos créditos a todo tipo de informação externa utilizada.

2. Fundamentação Teórica

2.1. Problema 1: Árvore Geradora Mínima

Enunciado: A partir de um dataset específico (grafo ponderado armazenado em arquivo .gml, .graphml, .txt, .net, etc) implementar o algoritmo de Kruskal ou de Prim para extrair uma Minimum Spanning Tree (MST) de G.

Foram fornecidos dois arquivos para a realização da atividade: *"ha30_dist.txt"*, com a matriz de adjacências do grafo; e *"ha30_name.txt",* possuindo o nome das cidades que correspondem a cada vértice do grafo.

1	0	39	22	59	54	33	57	32	89	73	29	46	16	83	120	45	24	32	36	25	38	16	43	21	50	57	46	72	121	73
2	39	0	20	20	81	8	49	64	63	84	10	61	25	49	81	81	58	16	72	60	78	24	69	18	75	88	68	44	83	52
3	22	20	0	39	74	18	60	44	71	73	11	46	6	61	99	61	37	10	51	40	59	5	62	7	57	78	51	51	100	56
4	59	20	39	0	93	27	51	81	48	80	30	69	45	32	61	97	75	31	89	78	97	44	83	38	84	100	77	31	63	42
5	54	81	74	93	0	73	43	56	104	76	76	77	69	111	72	46	56	84	49	53	33	69	12	69	64	7	69	122	73	114
6	33	8	18	27	73	0	45	61	71	88	8	63	22	57	87	77	54	18	68	56	71	20	61	13	75	80	68	52	90	60
7	57	49	60	51	43	45	0	85	88	115	52	103	60	75	64	85	79	63	83	78	70	58	38	52	103	49	102	81	69	92
8	32	64	44	81	56	61	85	0	74	43	55	23	40	81	97	17	8	50	8	7	23	41	53	48	19	53	17	70	92	63
9	89	63	71	48	104	71	88	74	0	38	69	51	75	16	35	75	77	61	77	80	90	76	116	76	58	98	57	19	33	16
10	73	84	73	80	76	88	115	43	38	0	81	28	72	53	55	38	49	70	42	50	53	75	83	80	24	69	27	49	51	39
11	29	10	11	30	76	8	52	55	69	81	0	55	16	57	91	71	48	11	62	50	68	14	64	9	67	81	61	49	93	56
12	46	61	46	69	77	63	103	23	51	28	55	0	44	59	81	32	26	46	29	29	45	47	76	53	15	73	9	49	77	40
13	16	25	6	45	69	22	60	40	75	72	16	44	0	67	105	56	33	16	46	35	53	2	57	9	54	72	48	57	106	60
14	83	49	61	32	111	57	75	81	16	53	57	59	67	0	39	88	82	51	87	85	103	67	113	65	70	109	67	12	39	19
15	120	81	99	61	72	87	64	97	35	55	91	81	105	39	0	84	104	90	93	104	90	104	82	99	79	70	82	50	4	51
16	45	81	61	97	46	77	85	17	75	38	71	32	56	88	84	0	23	67	9	21	15	57	48	64	19	41	23	80	81	70
17	24	58	37	75	56	54	79	8	77	49	48	26	33	82	104	23	0	44	14	3	25	34	51	41	25	54	23	70	100	65
18	32	16	10	32	84	18	63	50	61	70	11	46	16	51	90	67	44	0	58	47	67	16	72	15	59	88	52	42	90	47
19	36	72	51	89	49	68	83	8	77	42	62	29	46	87	93	9	14	58	0	12	16	48	48	55	19	45	21	77	89	69
20	25	60	40	78	53	56	78	7	80	50	50	29	35	85	104	21	3	47	12	0	22	36	48	43	26	51	24	73	100	68
21	38	78	59	97	33	71	70	23	90	53	68	45	53	103	90	15	25	67	16	22	0	54	33	59	33	31	37	93	88	84
22	16	24	5	44	69	20	58	41	76	75	14	47	2	67	104	57	34	16	48	36	54	0	57	7	56	72	50	57	105	61
23	43	69	62	83	12	61	38	53	116	83	64	76	57	113	82	48	51	72	48	48	33	57	0	57	66	18	69	113	84	115
24	21	18	7	38	69	13	52	48	76	80	9	53	9	65	99	64	41	15	55	43	59	7	57	0	63	74	57	57	101	61
25	50	75	57	84	64	75	103	19	58	24	67	15	54	70	79	19	25	59	19	26	33	56	66	63	0	59	7	61	74	52
26	57	88	78	100	7	80	49	53	98	69	81	73	72	109	70	41	54	88	45	51	31	72	18	74	59	0	64	117	71	107
27	46	68	51	77	69	68	102	17	57	27	61	9	48	67	82	23	23	52	21	24	37	50	69	57	7	64	0	57	77	48
28	72	44	51	31	122	52	81	70	19	49	49	49	57	12	50	80	70	42	77	73	93	57	113	57	61	117	57	0	49	11
29	121	83	100	63	73	90	69	92	33	51	93	77	106	39	4	81	100	90	89	100	88	105	84	101	74	71	77	49	0	48
30	73	52	56	42	114	60	92	63	16	39	56	40	60	19	51	70	65	47	69	68	84	61	115	61	52	107	48	11	48	0

Figura 2.1: matriz de adjacências do grafo, presente no arquivo "ha30_dist.txt

Ademais, o grafo do exercício proposto é completo e possui 30 vértices com pesos diferentes para cada aresta, como ilustrado na imagem a seguir.

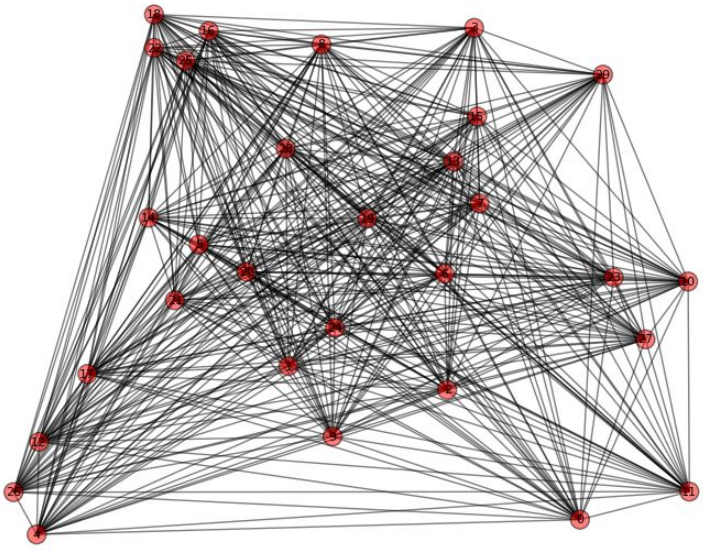

Figura 2.2: Grafo proposto

Algoritmo utilizado: Prim

Para a extração da árvore mínima do grafo, foi escolhido a implementação do algoritmo de Prim. Este encontra, através de iterações, uma árvore geradora mínima, não necessariamente única, para um grafo valorado e não direcionado. Na primeira iteração, tem-se uma subárvore T formada apenas por um vértice e deve-se escolher a aresta de custo mínimo, a qual será adicionada à T. O caráter do algoritmo é nomeado como "guloso" pois, a cada iteração, incorpora uma aresta. Este processo repete-se até que todos os vértices tenham sido visitados e, como resultado, tem-se a árvore geradora mínima.

2.2. Problema 2: Busca em Largura e Profundidade

Enunciado: Implementar os algoritmos BFS e DFS para extrair as árvores BFS-tree e DFS-tree dos grafos a seguir.

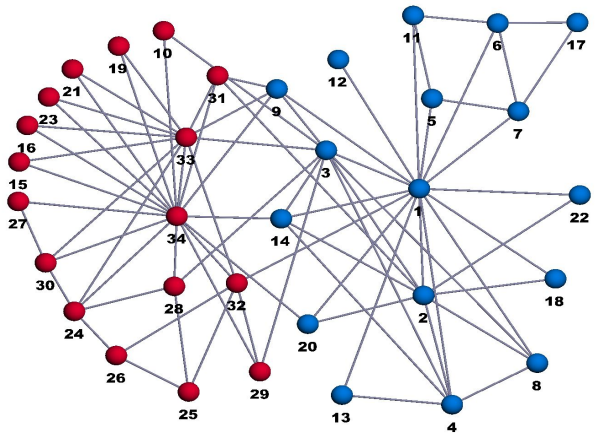

Figura 2.3: Grafo Zachary's karate club

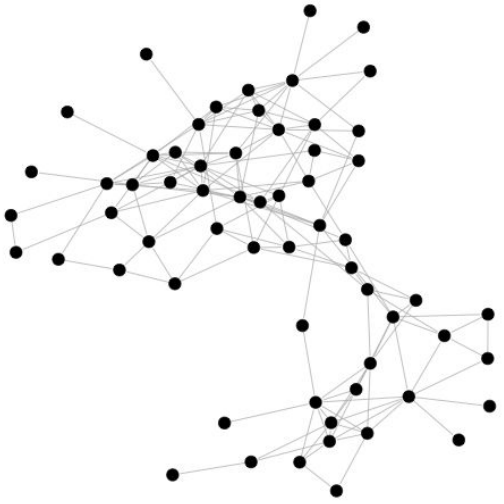

Figura 2.4: Grafo Dolphins social network

Algoritmo utilizado: Depth-First Search (DFS) e Breadth-First Search (BFS)

A Busca em Profundidade é utilizada para percorrer e/ou buscar itens em grafos. Este é percorrido enquanto houverem nós filhos não visitados, se aprofundando cada vez mais para somente depois retornar, o que faz com que assemelhe-se a recursividade. O processo repete-se até que o alvo desejado seja encontrado.

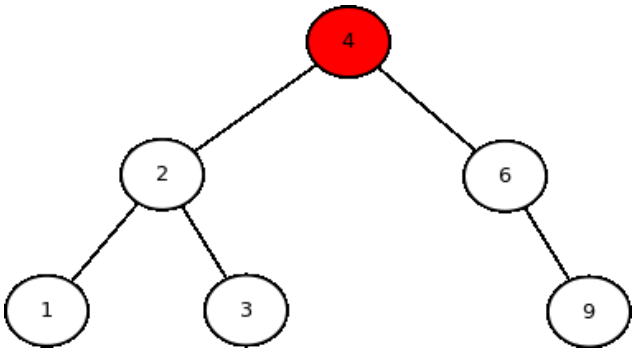

Figura 2.5: Gif representativo algoritmo DFS, disponível em <http://bit.ly/AlgoritmoDFS>.

Já a Busca em Largura (também chamada de busca em amplitude) realiza um caminho diferente, iniciando-se pela raiz e explorando todos os vizinhos desta. Em seguida, para cada um dos vizinhos acessados, explora os vizinhos deste. Tal procedimento também será repetido até que o alvo tenha sido encontra ou todos os vértices tenham sido visitados.

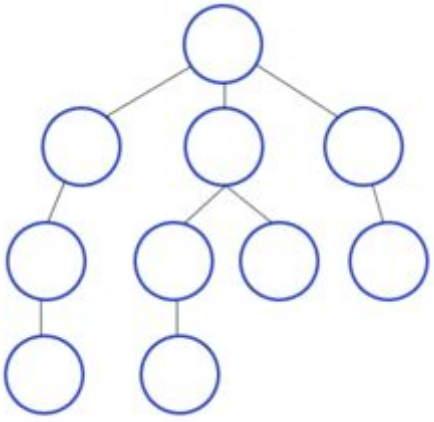

Figura 2.6: Gif representativo algoritmo BFS, disponível em <http://bit.ly/AlgoritmoBFS>

2.3. Problema 3: Árvores de Caminhos Mínimos e Agrupamento de Dados

Enunciado: A partir de um dataset específico (grafo **ponderado** armazenado em arquivo .gml, .graphml, .txt, .net, etc) e implementar o algoritmo de Dijkstra para extrair uma árvore de caminhos mínimos de G.

Metodologia: Após a implementação do algoritmo, uma forma de crescer várias subárvores de caminhos mínimos é inicializar várias sources, ou seja, atribuir custo inicial zero a um número K de vértices. O restante do algoritmo permanece intacto. O que irá acontecer é um processo de disputa entre cada uma das raízes para verificar qual delas irá conquistar cada vértice de G. Ao final da execução um vértice estará "pendurado" apenas a

uma única subárvore, fazendo com que tenhamos vários grupos de nós, similar ao que acontecia com as MST's. Porém aqui há supervisão no processo de formação dos grupos, uma vez que o usuário pode definir de onde as subárvores irão iniciar o crescimento (esses pontos devem ser escolhidos de forma a definir o centro dos agrupamentos).

Questionamentos: Considerando o grafo em questão, mostre os resultados (plote graficamente) obtidos para:

a) 2 agrupamentos (K = 2)

b) 3 agrupamentos (K = 3)

Para este projeto foi fornecido novamente uma matriz de adjacências e uma lista de nomes de cidades para serem usadas de labels. Desta vez, com 59 vértices.

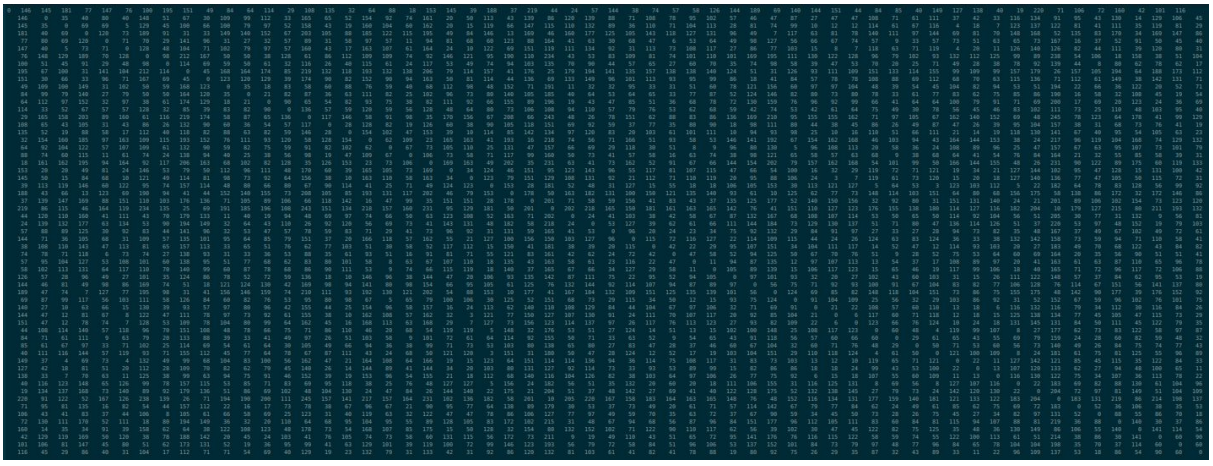

Figura 2.7: Matriz de adjacências do arquivo "wg59_dist.txt"

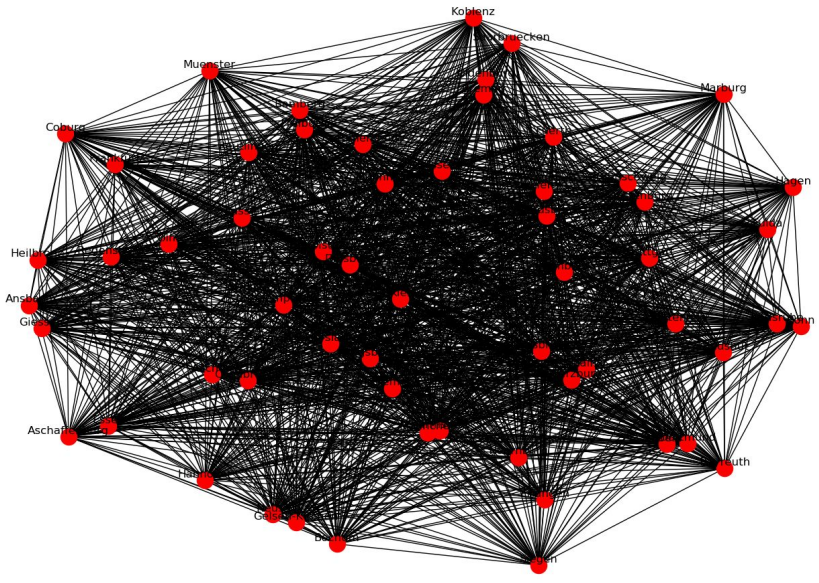

Figura 2.8: Grafo resultante da matriz de adjacências acima

Algoritmo utilizado: Djikstra

Responsável por calcular o caminho de custo mínimo de um determinado vértice do grafo até todos os outros, baseando-se em uma estimativa inicial e, paulatinamente, realiza o ajuste desta. Primeiramente, escolhe-se um vértice inicial como raiz e, com isto, o algoritmo seleciona o vértice mais próximo da origem. Se, ao passar novamente por este após acrescentar um próximo vértice, a distância diminuir seu valor, então a anterior será desconsiderada em prol da atual.

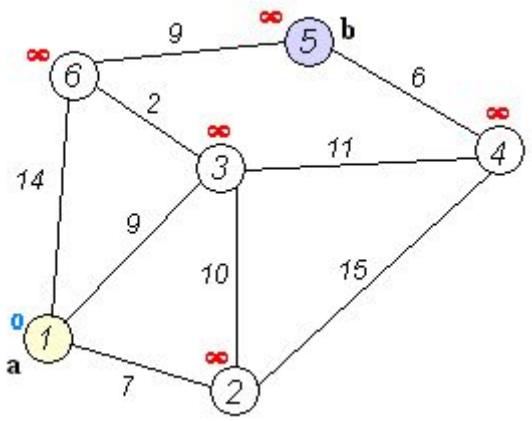

Figura 2.9: Gif representativo algoritmo DFS, disponível em <http://bit.ly/AlgoritmoDijkstra>

2.4. Problema 4: Caixeiro Viajante

Enunciado: Desenvolver um programa que deve ler um grafo Hamiltoniano ponderado a partir de um arquivo qualquer e através de um algoritmo visto em sala (2-otimal ou Twice-Around) obter 10 soluções diferentes para o problema do caixeiro-viajante.

Metodologia: Para obter soluções distintas para o problema há algumas heurísticas comumente adotadas na prática: utilizar diferentes inicializações, ou seja, soluções iniciais. Elas podem ser geradas simplesmente aleatoriamente (selecionando vértices quaisquer) ou utilizando alguma heurística, como por exemplo a escolha do vizinho mais próximo por exemplo. Dessa forma, escolhe-se aleatoriamente apenas o primeiro vértice do ciclo (v0) e depois sempre é escolhido como próximo elemento da sequência o vizinho mais próximo do vértice atual, até que o ciclo Hamiltoniano seja formado (não sobre mais vértices).

Questionamentos: Liste as 3 melhores soluções e as 3 piores obtidas. Qual a diferença de custo entre a melhor e a pior? Discuta como a diferença pode ser significativa.

Algoritmo utilizado: Twice-Around

O problema do Caixeiro Viajante busca determinar a menor rota existente para percorrer várias cidades, de modo que cada uma seja visitada apenas uma vez e que o destino final corresponda à origem. Desse modo, para resolver este problema, escolheu-se o algoritmo Twice-Around, o qual apresenta uma solução aproximada ao final de seu procedimento. Inicialmente, parte de uma árvore geradora mínima e, assim, obtém um ciclo hamiltoniano a partir de um ciclo euleriano, por meio de um percurso que prioriza atalhos e evita repetição de vértices.

2.5. Problema 5: Emparelhamentos estáveis e o algoritmo de Gale-Shapley

Enunciado: Implementar o algoritmo de Gale-Shapley para resolver o problema do emparelhamento estável para as seguintes listas de preferências:

Homens:

abe: abi, eve, cath, ivy, jan, dee, fay, bea, hope, gay
bob: cath, hope, abi, dee, eve, fay, bea, jan, ivy, gay
col: hope, eve, abi, dee, bea, fay, ivy, gay, cath, jan
dan: ivy, fay, dee, gay, hope, eve, jan, bea, cath, abi
ed: jan, dee, bea, cath, fay, eve, abi, ivy, hope, gay
fred: bea, abi, dee, gay, eve, ivy, cath, jan, hope, fay
gav: gay, eve, ivy, bea, cath, abi, dee, hope, jan, fay
hal: abi, eve, hope, fay, ivy, cath, jan, bea, gay, dee
ian: hope, cath, dee, gay, bea, abi, fay, ivy, jan, eve
jon: abi, fay, jan, gay, eve, bea, dee, cath, ivy, hope

Mulheres:

abi: bob, fred, jon, gav, ian, abe, dan, ed, col, hal
bea: bob, abe, col, fred, gav, dan, ian, ed, jon, hal
cath: fred, bob, ed, gav, hal, col, ian, abe, dan, jon
dee: fred, jon, col, abe, ian, hal, gav, dan, bob, ed
eve: jon, hal, fred, dan, abe, gav, col, ed, ian, bob
fay: bob, abe, ed, ian, jon, dan, fred, gav, col, hal
gay: jon, gav, hal, fred, bob, abe, col, ed, dan, ian
hope: gav, jon, bob, abe, ian, dan, hal, ed, col, fred
ivy: ian, col, hal, gav, fred, bob, abe, ed, jon, dan
jan: ed, hal, gav, abe, bob, jon, col, ian, fred, dan

Questionamentos: Qual o resultado se os homens forem dominantes? E se as mulheres forem dominantes? Os resultados são iguais?

Algoritmo utilizado: Gale-Shapley

Tal algoritmo é baseado no problema do emparelhamento estável. Este resume-se a encontrar um emparelhamento entre dois elementos de conjuntos distintos que seja estável, baseando-se em uma fila de prioridades que cada um possui. Vale destacar a diversidade de aplicações do cotidiano que este possui, como processos seletivos de universidades, escolhas de colega de quarto, seleção de estagiários e, como neste projeto, a estabilidade do casamento.

3. Materiais e Métodos

O projeto foi implementado através do sistema operacional Windows 10, utilizando da linguagem de programação Python recomendada pelo docente e que ocasionou em maior qualidade do projeto. Além disso, o desenvolvimento do código foi realizado no ambiente online Repl.it e na plataforma de Python para Data Science, Anaconda. Utilizou-se as bibliotecas propostas: NetworkX, NumPy e Matplotlib

Para garantir resultados devidamente qualificados e com êxito em seu funcionamento, grande parcela dos algoritmos foram testados utilizando-se conjunto de dados fornecidos pela apostila do professor Alexandre Levada. Desse modo, comparações e ajustes foram efetuados até obter-se a implementação adequada.

4. Resultados e Discussões

4.1. Problema 1: Árvore Geradora Mínima

Código do arquivo Prim.py

```
import numpy as np
import networkx as nx
from math import inf
from matplotlib import pyplot as plt

#carregar os labels do grafo
def load_labels():
        labels = {}
        count = 0
        file = open("ha30_name.txt", "r")
        for f in file:
                  if '#' not in f:
                      labels[count] = f.replace("\n", "")
                      count+=1
 return labels

#retorna o vértice com peso mínimo que ainda não foi visitado:
def ExtractMin(Q):
        for i in list(Q):
                if(Q[i] == min(Q.values())):
                    return (i)
                else:
                    print("ERROR")


#algoritmo de prim
def Prim(G, W, initial):

        #dicionário de caminhos mínimos não visitados
        MinEdgeNotVisited = {}

        #dicionário de "pais" dos vértices
        Parent = {}

        #árvore geradora mínima
        MST = []

        #inicializando caminhos mínimos com infinito, e pais como "nenhum"
        for i in list(G.nodes()):
                MinEdgeNotVisited[i] = inf
                Parent[i] = None

        #vértice atual recebe initial (passado por parâmetro)
        current = initial
         #como visitamos o vértice atual, remove ele do dicionário de arestas com
       peso min
        MinEdgeNotVisited.pop(current)
```

```
        #para cada um dos vértices restantes:
        for count in range(len(G.nodes()) - 1):

                #para cada vizinho do vértice atual
                for i in list(G.adj[current]):
                        if (current, i) in W and i in MinEdgeNotVisited:

                                #se o peso entre o atual e o vizinho < do que o
                          menor peso já descoberto desse vizinho
                                if W[(current, i)] < MinEdgeNotVisited[i]:

                                        #então o menor peso é alterado
                                        MinEdgeNotVisited[i] = W[(current, i)]
                                        #definimos o vértice atual como pai dele
                                        Parent[i] = current

                        #mesma coisa, mas para caso (i, current) esteja no grafo
                        elif (i, current) in W and i in MinEdgeNotVisited:
                                if W[(i, current)] < MinEdgeNotVisited[i]:
                                        MinEdgeNotVisited[i] = W[(i, current)]
                                        Parent[i] = current

                    #vértice atual se torna o vértice que possui o menor peso para ser
                visitado
                    current = ExtractMin(MinEdgeNotVisited)

                    #adiciona na MST a aresta (pai do vértice atual, vértice atual)
                    MST.append((Parent[current], current))

                    #remove o vértice atual do dicionário de arestas de menor peso
                    MinEdgeNotVisited.pop(current)

     return MST


A = np.loadtxt('ha30_dist.txt')
G = nx.from_numpy_matrix(A)
W = nx.get_edge_attributes(G,'weight')
labels = load_labels()

#desenha o grafo
pos = nx.kamada_kawai_layout(G)
nx.draw_networkx(G, pos, labels=labels)
plt.show()

MST= Prim(G, W, 0)

#desenha a MST
pos = nx.kamada_kawai_layout(G)
nx.draw_networkx(G, pos, edgelist=MST, labels=labels)
plt.show()
```

A partir do código desenvolvido acima, foi extraída uma árvore geradora mínima para qualquer que seja o grafo G. Ao utilizar o grafo proposto pelo exercício exercício em questão, a função Prim retorna MST = [(0, 12), (12, 21), (21, 2), (21, 23), (23, 10), (10, 5), (5, 1), (2, 17), (1, 3), (0, 16), (16, 19), (19, 7), (7, 18), (18, 15), (15, 20), (7, 26), (26, 24), (26, 11), (24, 9), (20, 25), (25, 4), (4, 22), (3, 27), (27, 29), (27, 13), (29, 8), (8, 28), (28, 14), (22, 6)].

Torna-se possível visualizar, graficamente, a Árvore Geradora Mínima através da biblioteca NetworkX em conjunto com a MatPlotLib; obtendo-se, assim, o seguinte grafo:

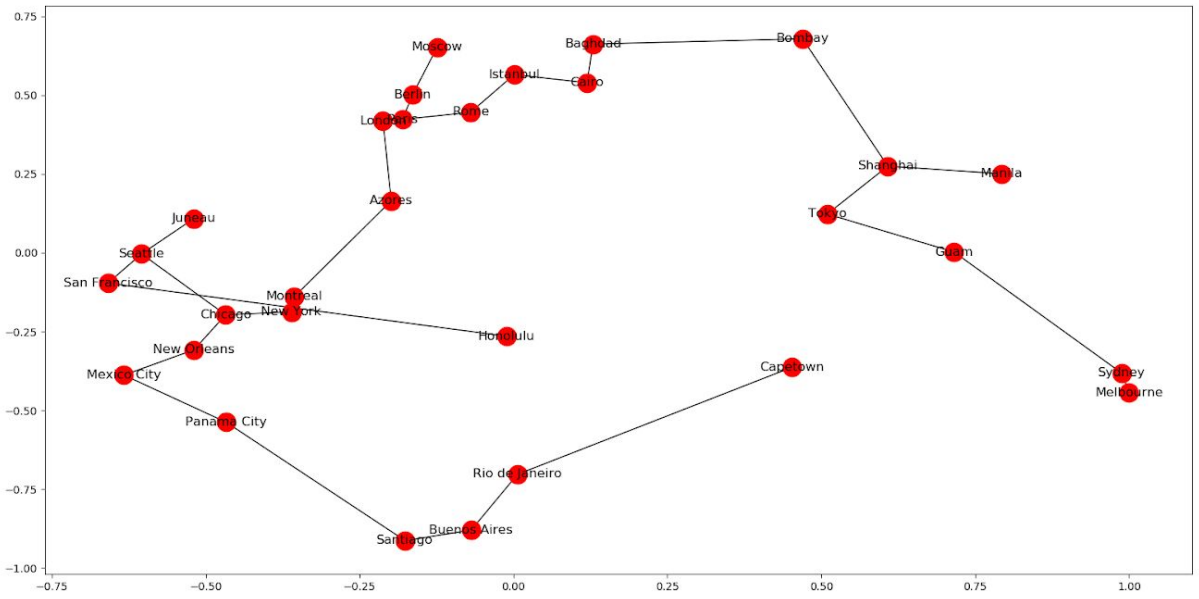

Figura 4.1: Árvore Geradora Mínima de G proposto.

4.2. Problema 2: Busca em Largura e Profundidade

Para o desenvolvimento do algoritmo BFS foi usado o pseudo-código das notas de aula do professor Alexandre. Por outro lado, para o algoritmo de DFS optou-se pelo uso de uma pilha ao invés de recursão (como foi passado nas aulas), visto que a recursão pode causar problemas com grafos muito extensos.

Código do arquivo BFSandDFS.py

```
import networkx as nx
import matplotlib.pyplot as plt
from math import inf

#algoritmo para extrair a BFS-tree
def BFS(G, initial):
        #dicionário com a distância dos vértices
        Distance = {}

        #dicionário com o 'pai' dos vértices
        Parent = {}

        #dicionáio com a cor dos vértices
        Color = {}

        #fila (FIFO - first in, first out)
        Queue = []

        #lista de arestas da árvore extraída
        bfs = []
         #inicializando os valores
        for i in list(G.nodes()):
                Color[i] = "WHITE"
                Distance[i] = inf
                Parent[i] = None

        #vértice atual = valor inicial passado como parâmetro
        current = initial

        #cor do vértice inicial torna-se cinza (marca-se como visitado)
        Color[current] = "GRAY"

        #distancia do vértice inicial é definida como 0
        Distance[current] = 0

        #adiciona o vértice inicial à lista
        Queue.append(current)

        #enquanto a lista não estiver vazia
        while len(Queue) != 0:

                #vértice atual se torna o primeiro elem. da fila (e este é
        removido dela)
                current = Queue.pop(0)

                #se o vértice atual for diferente do inicial
                if current != initial:
                        #salva em bfs a aresta (pai do atual, atual)
                        bfs.append((Parent[current], current))

                #para cada vizinho do vértice atual
                for adj in list(G.adj[current]):
```

```
                        #se o vértice não foi visitado ainda
                        if Color[adj] == "WHITE":
                                Distance[adj] = Distance[current] + 1
                                Parent[adj] = current
                                Color[adj] = "GRAY"
                                Queue.append(adj)

                #marca-se o vértice atual como finalizado
                Color[current] = "BLACK"
        return bfs


#algoritmo para extrair a DFS-tree
def DFS(G, u):
        #lista de vértices visitados
        Explored = []

        #pilha (LIFO - Last In, First Out) inicializada com vértice inicial
        Stack = [u]

        #dicionário dos 'pais' dos vértices
        Parent = {}

        #lista de arestas da árvore extraída
        Dfs = []

        #enquanto a pilha não está vazia
        while len(Stack) != 0:

                 #vértice atual recebe ultimo elem. da pilha (e este é removido
             dela)
                current = Stack.pop()

                #se o atual não foi já visitado
                if current not in Explored:

                        #marca como visitado
                        Explored.append(current)

                        #adiciona a aresta (pai de atual, atual) em Dfs
                        if current != u:
                                Dfs.append((Parent[current], current))

                        #Adj recebe lista de vizinhos do vértice atual
                        Adj = list(G.adj[current])

                        #Lista de vizinhos é ordenada decrescentemente
                        Adj.sort(reverse=True, key=int)

                        #para cada vizinho não explorado, adicionar ele na pilha
                        # e definir o vértice atual como pai dele.
                        for w in Adj:
                                if w not in Explored:
                                        Stack.append(w)
```

```
                                    Parent[w] = current
    return Dfs
```

Antes de verificar o funcionamento deste algoritmo com um grafo de diversos vértices, foram realizados alguns casos testes para melhor desenvolvimento na implementação; os quais basearam-se em exercícios originários da apostila de Teoria de Grafos, escrita pelo Professor Alexandre Levada. Um dos utilizados está exibido a seguir:

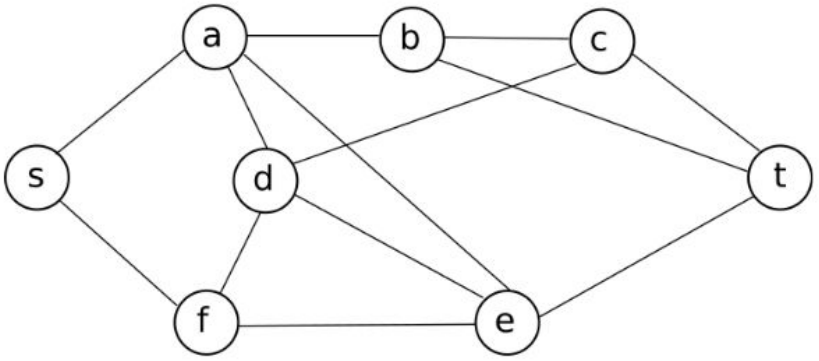

Figura 4.2: Árvore Geradora Mínima de G.

Substituiu-se o nome dos vértices para S = 1, A = 2, B = 3, C = 7, D = 4, E = 6, F = 5, T = 8 e fez-se:

```
117  #testando com exercício simples de sala:
118  H = nx.Graph()
119  H.add_edges_from([(1, 2), (1, 5), (2, 4), (2, 6), (2, 3), (4, 5), (4, 6), (4, 7),
120  (5, 6), (6, 8), (7, 8), (3, 8), (3, 7)])
121  print("\nExercicio Simples: ")
122  Exercicio = DFS(H, 1)
123  print(Exercicio)
```

A partir disto, foi encontrado um dos resultados corretos do exercício em questão:

[(1, 2), (2, 3), (3, 7), (7, 4), (4, 5), (5, 6), (6, 8)]

Tendo certeza do bom funcionamento do algoritmo, aplicou-se a ele os grafos propostos por este projeto. Para garantir a veracidade das respostas, comparou-as com a MST gerada pelas funções de busca em largura e profundidade da NetworkX. O resultado gerado corresponde fielmente ao esperado:

```
Dolphins:
Usando nosso algoritmo:
[('1', '17'), ('1', '19'), ('1', '26'), ('1', '27'), ('1', '28'), ('1', '36'), ('1', '41'), ('1', '54'), ('17', '6'), ('17',
'9'), ('17', '13'), ('17', '22'), ('17', '25'), ('17', '31'), ('17', '57'), ('19', '7'), ('19', '30'), ('28', '8'), ('28',
'20'), ('28', '47'), ('36', '23'), ('36', '37'), ('36', '39'), ('36', '40'), ('36', '59'), ('6', '56'), ('9', '5'), ('9',
'32'), ('57', '48'), ('30', '42'), ('8', '3'), ('8', '45'), ('20', '16'), ('20', '18'), ('20', '38'), ('20', '44'), ('20',
'50'), ('47', '0'), ('47', '10'), ('23', '51'), ('37', '14'), ('37', '21'), ('37', '33'), ('37', '34'), ('37', '43'), ('37',
'61'), ('40', '15'), ('40', '52'), ('32', '60'), ('42', '2'), ('45', '24'), ('45', '29'), ('38', '58'), ('51', '4'), ('51',
'11'), ('51', '55'), ('33', '12'), ('34', '49'), ('43', '46'), ('43', '53'), ('29', '35')]
Usando a função da biblioteca:
[('1', '17'), ('1', '19'), ('1', '26'), ('1', '27'), ('1', '28'), ('1', '36'), ('1', '41'), ('1', '54'), ('17', '6'), ('17',
'9'), ('17', '13'), ('17', '22'), ('17', '25'), ('17', '31'), ('17', '57'), ('19', '7'), ('19', '30'), ('28', '8'), ('28',
'20'), ('28', '47'), ('36', '23'), ('36', '37'), ('36', '39'), ('36', '40'), ('36', '59'), ('6', '56'), ('9', '5'), ('9',
'32'), ('57', '48'), ('30', '42'), ('8', '3'), ('8', '45'), ('20', '16'), ('20', '18'), ('20', '38'), ('20', '44'), ('20',
'50'), ('47', '0'), ('47', '10'), ('23', '51'), ('37', '14'), ('37', '21'), ('37', '33'), ('37', '34'), ('37', '43'), ('37',
'61'), ('40', '15'), ('40', '52'), ('32', '60'), ('42', '2'), ('45', '24'), ('45', '29'), ('38', '58'), ('51', '4'), ('51',
'11'), ('51', '55'), ('33', '12'), ('34', '49'), ('43', '46'), ('43', '53'), ('29', '35')]


DFS:

Karate:
Usando nosso algoritmo:
[('1', '2'), ('2', '3'), ('3', '4'), ('4', '8'), ('4', '13'), ('4', '14'), ('14', '34'), ('34', '9'), ('9', '31'), ('31',
'33'), ('33', '15'), ('33', '16'), ('33', '19'), ('33', '21'), ('33', '23'), ('33', '24'), ('24', '26'), ('26', '25'), ('25',
'28'), ('25', '32'), ('32', '29'), ('24', '30'), ('30', '27'), ('34', '10'), ('34', '20'), ('2', '18'), ('2', '22'), ('1',
'5'), ('5', '7'), ('7', '6'), ('6', '11'), ('6', '17'), ('1', '12')]
Usando a função da biblioteca:
[('1', '2'), ('2', '3'), ('3', '4'), ('4', '8'), ('4', '13'), ('4', '14'), ('14', '34'), ('34', '9'), ('9', '31'), ('31',
'33'), ('33', '15'), ('33', '16'), ('33', '19'), ('33', '21'), ('33', '23'), ('33', '24'), ('24', '26'), ('26', '25'), ('25',
'28'), ('25', '32'), ('32', '29'), ('24', '30'), ('30', '27'), ('34', '10'), ('34', '20'), ('2', '18'), ('2', '22'), ('1',
'5'), ('5', '7'), ('7', '6'), ('6', '11'), ('6', '17'), ('1', '12')]


Dolphins:
Usando nosso algoritmo:
[('1', '17'), ('17', '6'), ('6', '9'), ('9', '5'), ('5', '13'), ('13', '32'), ('32', '60'), ('13', '41'), ('41', '54'), ('54',
'7'), ('7', '19'), ('19', '30'), ('30', '28'), ('28', '8'), ('8', '3'), ('3', '14'), ('14', '0'), ('0', '10'), ('10', '2'),
('2', '42'), ('42', '47'), ('47', '20'), ('20', '16'), ('16', '33'), ('33', '12'), ('33', '21'), ('21', '18'), ('18', '15'),
('15', '24'), ('24', '29'), ('29', '35'), ('29', '43'), ('43', '37'), ('37', '34'), ('34', '44'), ('44', '38'), ('38', '52'),
('52', '40'), ('40', '36'), ('36', '23'), ('23', '45'), ('45', '50'), ('50', '51'), ('51', '4'), ('51', '11'), ('51', '55'),
('45', '59'), ('36', '39'), ('39', '57'), ('57', '48'), ('38', '58'), ('34', '49'), ('49', '46'), ('37', '61'), ('61', '53'),
('7', '27'), ('27', '25'), ('25', '26'), ('5', '56'), ('17', '22'), ('17', '31')]
Usando a função da biblioteca:
[('1', '17'), ('17', '6'), ('6', '9'), ('9', '5'), ('5', '13'), ('13', '32'), ('32', '60'), ('13', '41'), ('41', '54'), ('54',
'7'), ('7', '19'), ('19', '30'), ('30', '28'), ('28', '8'), ('8', '3'), ('3', '14'), ('14', '0'), ('0', '10'), ('10', '2'),
('2', '42'), ('42', '47'), ('47', '20'), ('20', '16'), ('16', '33'), ('33', '12'), ('33', '21'), ('21', '18'), ('18', '15'),
('15', '24'), ('24', '29'), ('29', '35'), ('29', '43'), ('43', '37'), ('37', '34'), ('34', '44'), ('44', '38'), ('38', '52'),
('52', '40'), ('40', '36'), ('36', '23'), ('23', '45'), ('45', '50'), ('50', '51'), ('51', '4'), ('51', '11'), ('51', '55'),
('45', '59'), ('36', '39'), ('39', '57'), ('57', '48'), ('38', '58'), ('34', '49'), ('49', '46'), ('37', '61'), ('61', '53'),
('7', '27'), ('27', '25'), ('25', '26'), ('5', '56'), ('17', '22'), ('17', '31')]


BFS:

Karate:
Usando nosso algoritmo:
[('1', '2'), ('1', '3'), ('1', '4'), ('1', '5'), ('1', '6'), ('1', '7'), ('1', '8'), ('1', '9'), ('1', '11'), ('1', '12'),
('1', '13'), ('1', '14'), ('1', '18'), ('1', '20'), ('1', '22'), ('1', '32'), ('2', '31'), ('3', '10'), ('3', '28'), ('3',
'29'), ('3', '33'), ('6', '17'), ('9', '34'), ('32', '25'), ('32', '26'), ('28', '24'), ('33', '15'), ('33', '16'), ('33',
'19'), ('33', '21'), ('33', '23'), ('33', '30'), ('34', '27')]
Usando a função da biblioteca:
[('1', '2'), ('1', '3'), ('1', '4'), ('1', '5'), ('1', '6'), ('1', '7'), ('1', '8'), ('1', '9'), ('1', '11'), ('1', '12'),
('1', '13'), ('1', '14'), ('1', '18'), ('1', '20'), ('1', '22'), ('1', '32'), ('2', '31'), ('3', '10'), ('3', '28'), ('3',
'29'), ('3', '33'), ('6', '17'), ('9', '34'), ('32', '25'), ('32', '26'), ('28', '24'), ('33', '15'), ('33', '16'), ('33',
'19'), ('33', '21'), ('33', '23'), ('33', '30'), ('34', '27')]
```

Podemos observar que no código da DFS o vetor de vizinhos do vértice atual é ordenado decrescentemente antes de seus elementos serem inseridos na pilha. Essa decisão é totalmente descartável, uma vez que podem existir diversas árvores DFS em um grafo. Contudo, isso foi implementado após concluir que o algoritmo de busca em profundidade da NetworkX “visita” os vizinhos de forma crescente (de acordo com o nome). Logo, empilhou-se o vetor de vizinhos ordenados decrescentemente a fim de obter o resultado idêntico ao da biblioteca, e certificar que o algoritmo funciona corretamente.

Por fim, foram realizados os desenhos gráficos das respectivas árvores.

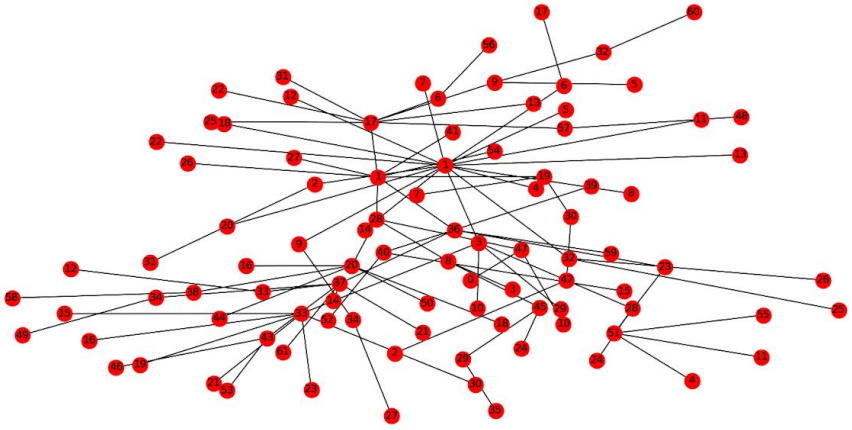

Figura 4.3: BFS-Tree do grafo Karate

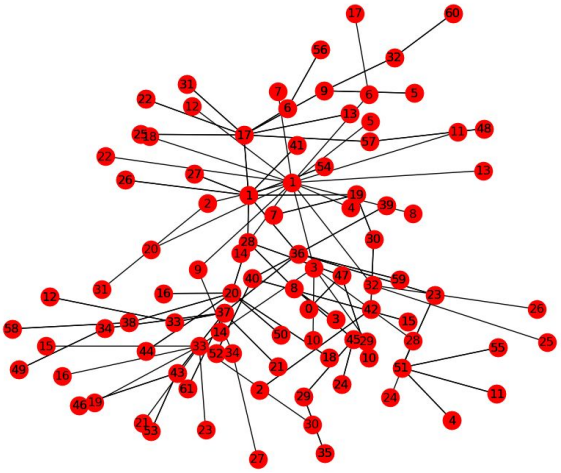

Figura 4.4: BFS-Tree do grafo Dolphin

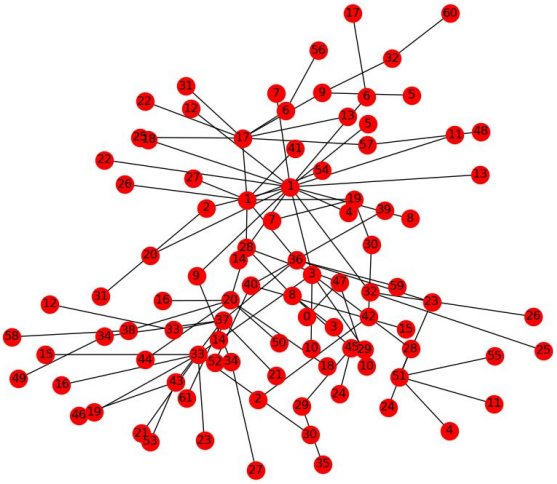

Figura 4.5: DFS-Tree do grafo Karate

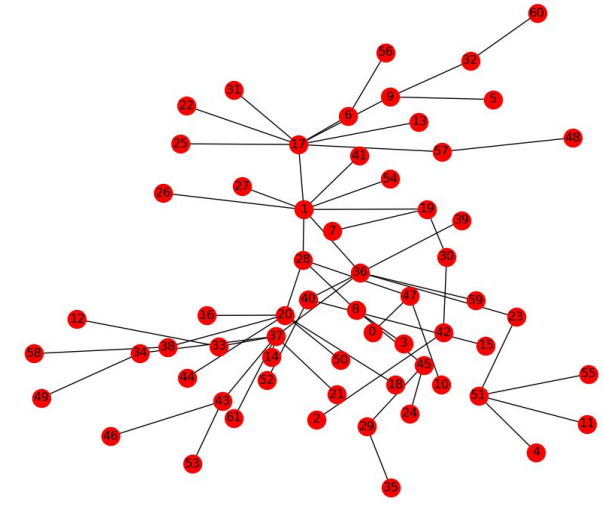

Figura 4.6: DFS-Tree do grafo Dolphins

4.3. Problema 3: Árvores de Caminhos Mínimos e Agrupamento de Dados

Código do arquivo Dijkstra.py

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
from math import inf

A = np.loadtxt('wg59_dist.txt')
G = nx.from_numpy_matrix(A)

#carregar as labels para o grafo
def load_labels(file_):
 labels = {}
 count = 0
 file = open(file_, "r")
 for f in file:
     if '#' not in f:
         labels[count] = f
         count+=1
 return labels

#retorna o vértice com peso mínimo que ainda não foi visitado:
def ExtractMin(Q):
 for i in list(Q):
   if(Q[i] == min(Q.values())):
     return (i)
 else:
   print("ERROR")
   return (None)


def Dijkstra(G):
 #menor custo até o momento para o caminho início->vértice
 MinPathAll = {}

 #menor custo até o momento para o caminho inicio->vértice, removendo visitados
 MinPathNotVisited = {}

 #predecessor dos vértices na árvore de caminhos mínimos
 Parent = {}

 #pesos das arestas do grafo G
 Weight = nx.get_edge_attributes(G,'weight')

 #lista de arestas da árvore de caminhos mínimos
 edges = []

 #inicializando valores para todos os vértices
 for i in list(G.nodes()):
   MinPathAll[i] = inf
   Parent[i] = None
```

```
 #definindo o(s) vértice(s) inicial(is)
 MinPathAll[0] = 0
 MinPathNotVisited = MinPathAll

 #enquanto há vértices não visitados
 while len(MinPathNotVisited) != 0:

   #atual será o vértice com menor caminho
   current = ExtractMin(MinPathNotVisited)

   #se existir pai para o atual, adicionar a aresta
   if Parent[current] != None:
     edges.append((Parent[current], current))

   #para cada vizinho do atual
   for adj in list(G.adj[current]):

     #se ele não foi visitado
     if (current, adj) in Weight and adj in MinPathNotVisited:

       #se o caminho min for maior que o caminho do atual+peso da aresta seguinte
       if MinPathAll[adj] > (MinPathAll[current] + Weight[(current, adj)]):
         #atualiza o valor do caminho minimo do vizinho
         MinPathAll[adj] = MinPathAll[current] + Weight[(current, adj)]
         MinPathNotVisited[adj] =  MinPathAll[adj]
         #define o atual como pai do vizinho
         Parent[adj] = current

     #mesma coisa, para (adj, current) no grafo
     elif (adj, current) in Weight and adj in MinPathNotVisited:
       if MinPathAll[adj] > (MinPathAll[current] + Weight[(adj, current)]):
         MinPathAll[adj] = MinPathAll[current] + Weight[(adj, current)]
         MinPathNotVisited[adj] =  MinPathAll[adj]
         Parent[adj] = current

   #removo o atual dos não visitados
   del MinPathNotVisited[current]

 return edges

#desenha o grafo
def print_min_path_tree(G, edges, labels):
 pos = nx.kamada_kawai_layout(G)
 for (u,v) in edges:
   if (u,v) not in list(G.edges()):
     edges.remove((u, v))
     edges.append((v, u))
 nx.draw(G, pos, edgelist=edges, label=labels )
 plt.show()
```

Novamente, foi conferido o resultado do caminho mínimo extraído por nosso algoritmo com o da função de Dijkstra para apenas um vértice inicial da NetworkX

```
Caminho mínimo nosso algoritmo:
[(0, 20), (0, 50), (0, 28), (0, 14), (0, 17), (0, 25), (0, 31), (0, 23), (0, 45), (0, 49), (0, 56), (0, 27), (0, 42), (0, 10), (0, 29),
(0, 33), (23, 34), (27, 12), (0, 18), (0, 38), (0, 52), (0, 54), (0, 32), (0, 6), (0, 4), (32, 43), (0, 11), (29, 44), (0, 19), (12, 57),
(0, 7), (0, 53), (19, 15), (0, 13), (32, 58), (15, 35), (43, 47), (43, 16), (58, 48), (7, 39), (43, 40), (43, 2), (43, 22), (0, 30), (0,
36), (29, 1), (43, 5), (43, 46), (43, 41), (0, 9), (0, 21), (29, 55), (0, 3), (44, 24), (44, 37), (0, 8), (0, 26), (0, 51)]

Caminho mínimo Dijkstra NetworkX:
{0: [0], 1: [0, 29, 1], 2: [0, 32, 43, 2], 3: [0, 3], 4: [0, 4], 5: [0, 32, 43, 5], 6: [0, 6], 7: [0, 7], 8: [0, 8], 9: [0, 9], 10: [0,
10], 11: [0, 11], 12: [0, 27, 12], 13: [0, 13], 14: [0, 14], 15: [0, 19, 15], 16: [0, 32, 43, 16], 17: [0, 17], 18: [0, 18], 19: [0, 19],
20: [0, 20], 21: [0, 21], 22: [0, 32, 43, 22], 23: [0, 23], 24: [0, 29, 44, 24], 25: [0, 25], 26: [0, 26], 27: [0, 27], 28: [0, 28], 29:
[0, 29], 30: [0, 30], 31: [0, 31], 32: [0, 32], 33: [0, 33], 34: [0, 23, 34], 35: [0, 19, 15, 35], 36: [0, 36], 37: [0, 29, 44, 37], 38:
[0, 38], 39: [0, 7, 39], 40: [0, 32, 43, 40], 41: [0, 32, 43, 41], 42: [0, 42], 43: [0, 32, 43], 44: [0, 29, 44], 45: [0, 45], 46: [0,
32, 43, 46], 47: [0, 32, 43, 47], 48: [0, 32, 58, 48], 49: [0, 49], 50: [0, 50], 51: [0, 51], 52: [0, 52], 53: [0, 53], 54: [0, 54], 55:
[0, 29, 55], 56: [0, 56], 57: [0, 27, 12, 57], 58: [0, 32, 58]}
```

Como a função do Dijkstra da biblioteca retorna um dicionário com o caminho mínimo para cada um dos vértices a partir do ponto inicial, enquanto nosso algoritmo retorna as arestas da árvore de caminhos mínimos, é necessário verificar um por um. Também é possível fazer a comparação através do grafo desenhado:

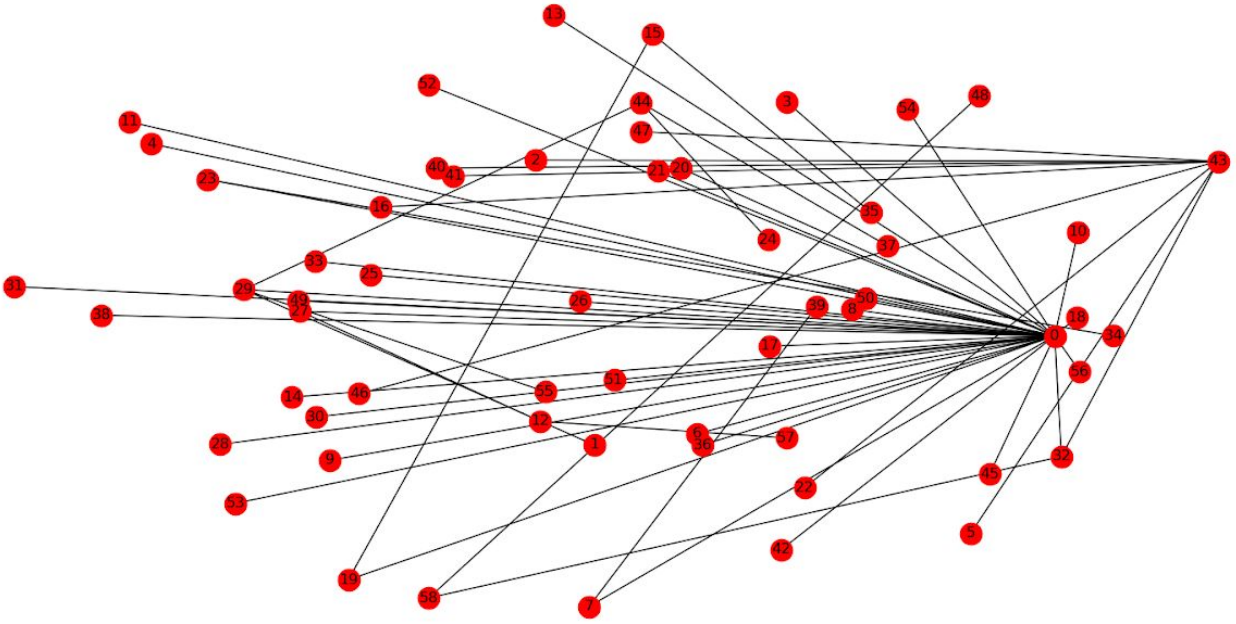

Figura 4.7: Árvore de caminhos mínimos para início no vértice 0

Com este algoritmo é possível gerar as atividades propostas em “Questionamentos”, no enunciado do projeto. Ao definir mais vértices com peso inicial 0 dentro da função de Dijkstra, podemos visualizar os agrupamentos formados ao redor destes vértices iniciais, uma vez que cada vértice do grafo se ligará com o inicial que possui menor caminho até ele. Podemos observar isto nas seguintes figuras, onde o grafo já é desenhado com os labels contendo nomes de cidades.

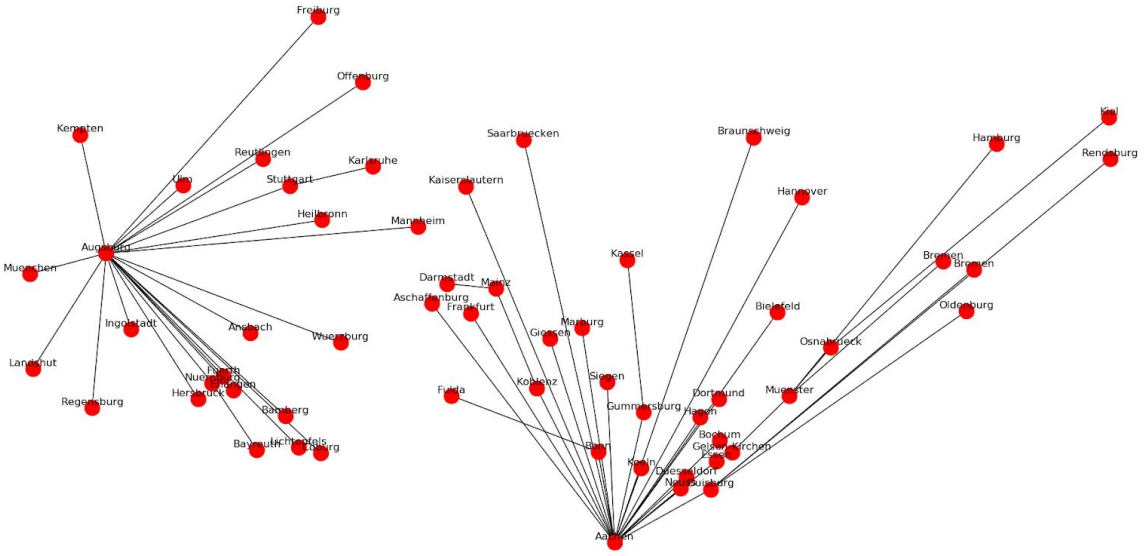

Figura 4.8: Árvores de caminhos mínimos para início nos vértices 0 (Augsburg) e 30 (Aachen)

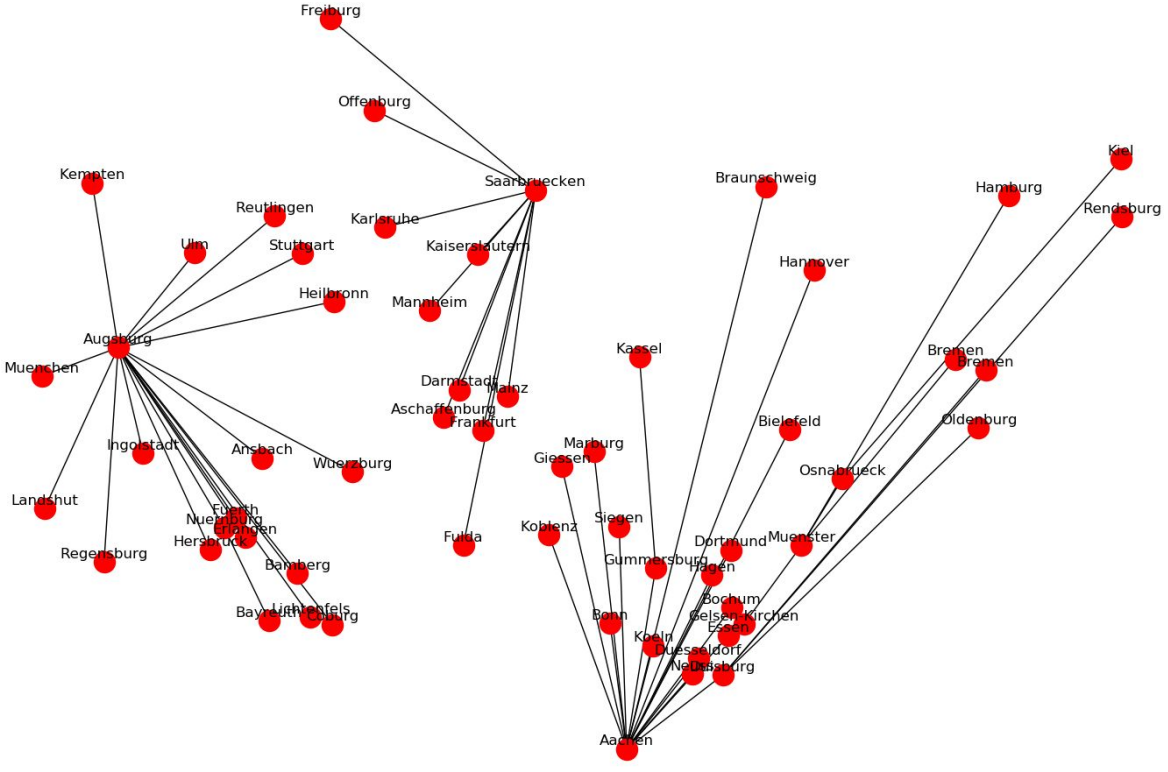

Figura 4.9: Árvores de caminhos mínimos para início nos vértices 0 (Augsburg), 30 (Aachen) e 57 (Shaarbruecken)

4.4. Problema 4: Caixeiro Viajante

Código do arquivo TwiceAround.py

```
import numpy as np
import networkx as nx
from matplotlib import pyplot as plt


A = np.loadtxt('ha30_dist.txt')
G = nx.from_numpy_matrix(A)
w = nx.get_edge_attributes(G,'weight')

def load_labels(file_):
 labels = {}
 count = 0
 file = open(file_, "r")
 for f in file:
     if '#' not in f:
         labels[count] = f
         count+=1
 return labels

def TwiceAround(G, initial):

 #encontrando uma árvore mínima do grafo G com ajuda da NetworkX
 mst = nx.minimum_spanning_tree(G)
 #tranformando a mst em multigrafo (aceita arestas paralelas)
 mst = nx.MultiGraph(mst)

 #declarando o grafo de resultado do algoritmo
 Graph = nx.Graph()

 #criando uma cópia de mst para usar no 'for' a seguir (não pode ser usada a
 #própria mst pois dentro do 'for' adicionamos elementos em mst)
 mstAux = mst.copy()

 #duplicando as arestas de mst
 for u,v in mstAux.edges():
   mst.add_edge(u,v)

 #encontrando um circuito euleriano em mst com ajuda da NetworkX
 #passando como parâmetro um vérticie inicial
 euler_circuit = list(nx.eulerian_circuit(mst, initial))
  #declaração de uma lista de vértices do circuito de euler, em ordem
 list_nodes_euler = []

 #colocando os valores nesta lista
 for u,v in euler_circuit:
   if u not in list_nodes_euler:
     list_nodes_euler.append(u)

   if v not in list_nodes_euler:
```

```
    list_nodes_euler.append(v)

 #adicionando o vértice inicial ao fim do ciclo
 list_nodes_euler.append(initial)
  #populando o grafo de resposta, com o circuito euleriano descoberto
 for node in range(len(G.nodes())):
   parent = list_nodes_euler[node]
   child = list_nodes_euler[node+1]

   Graph.add_edge(parent,child)
   Graph[parent][child]['weight'] = G[parent][child]['weight']
  return Graph
```

Em seguida gerou-se 10 respostas diferentes para o problema do caixeiro viajante, com um vértices iniciais distintos.

```
Result1 = TwiceAround(G, 0)
Result2 = TwiceAround(G, 1)
Result3 = TwiceAround(G, 2)
Result4 = TwiceAround(G, 3)
Result5 = TwiceAround(G, 4)
Result6 = TwiceAround(G, 5)
Result7 = TwiceAround(G, 6)
Result8 = TwiceAround(G, 7)
Result9 = TwiceAround(G, 8)
Result10 = TwiceAround(G, 9)
```

É utilizado o seguinte algoritmo para o cálculo dos custos de cada ciclo Hamiltoniano descoberto:

```
def total_weight(T):
   weight = 0
   weights = nx.get_edge_attributes(T, 'weight')
   for v in T.edges():
       weight += weights[v]
return weight
```

A partir deste conseguimos chegar nos 3 ciclos menos e mais custosos:

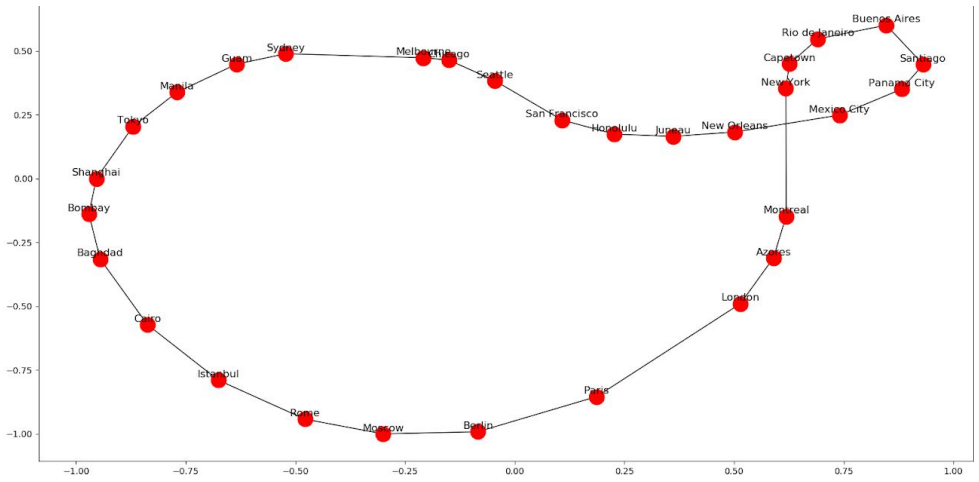

Figura 4.10: Ciclo menos custoso: 523; Vértice 6 como inicial

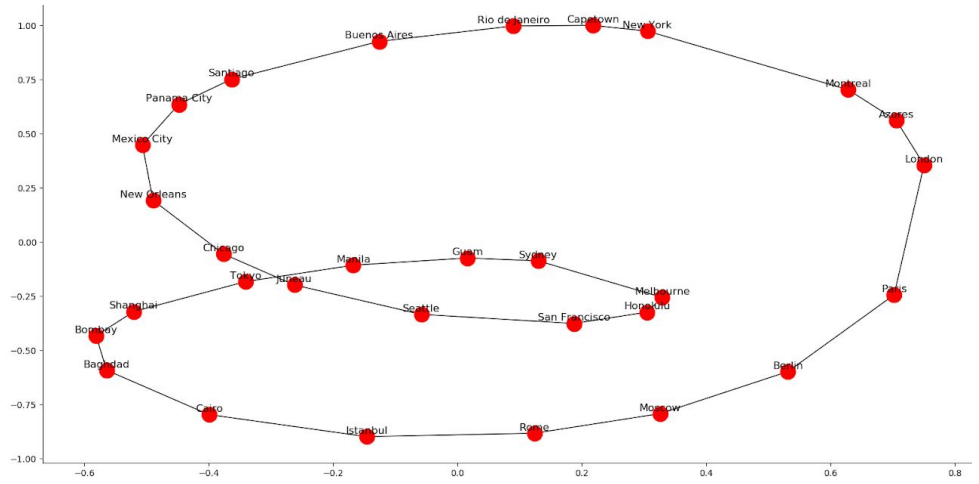

Figura 4.11: 2° Ciclo menos custoso: 550; Vértice 9 como inicial

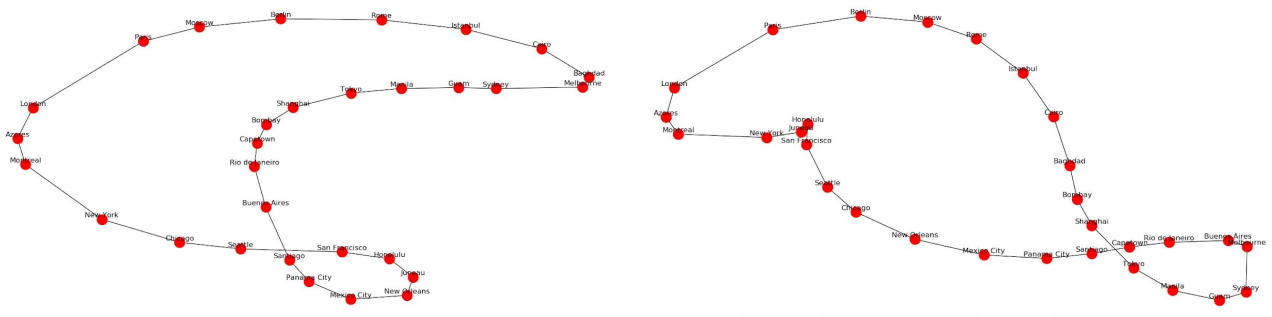

Figura 4.12: 3° Ciclo menos custoso: 573; Vértice 3 e 4 como inicial, respectivamente

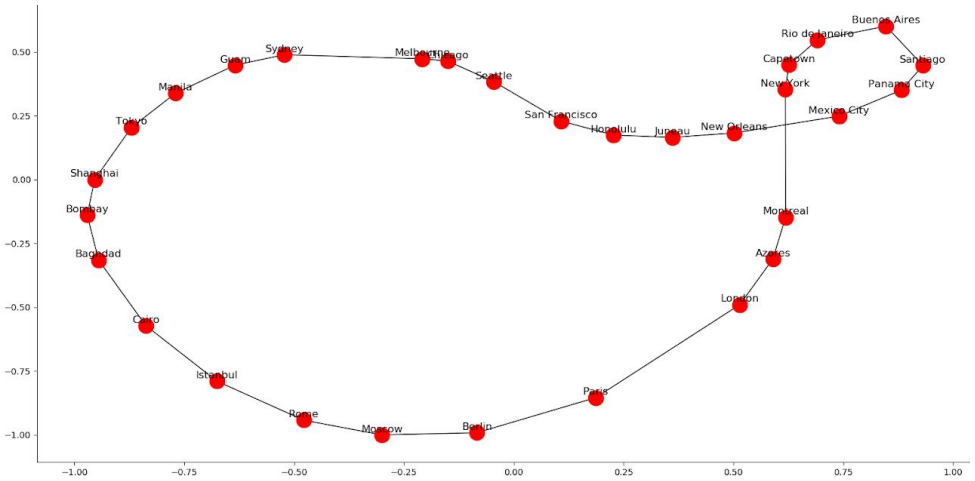

Figura 4.13: Ciclo mais custoso: 626; Vértice 7 como inicial

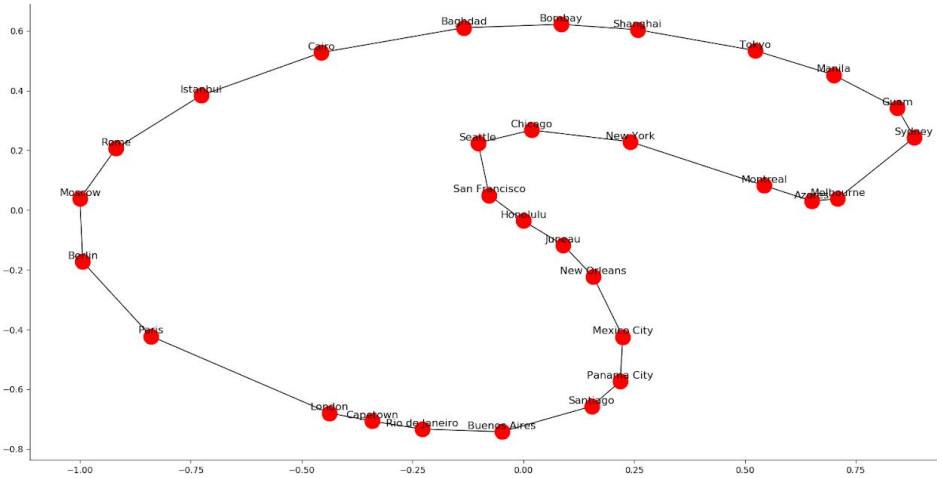

Figura 4.13: 2° Ciclo mais custoso: 622; Vértice 0 como inicial

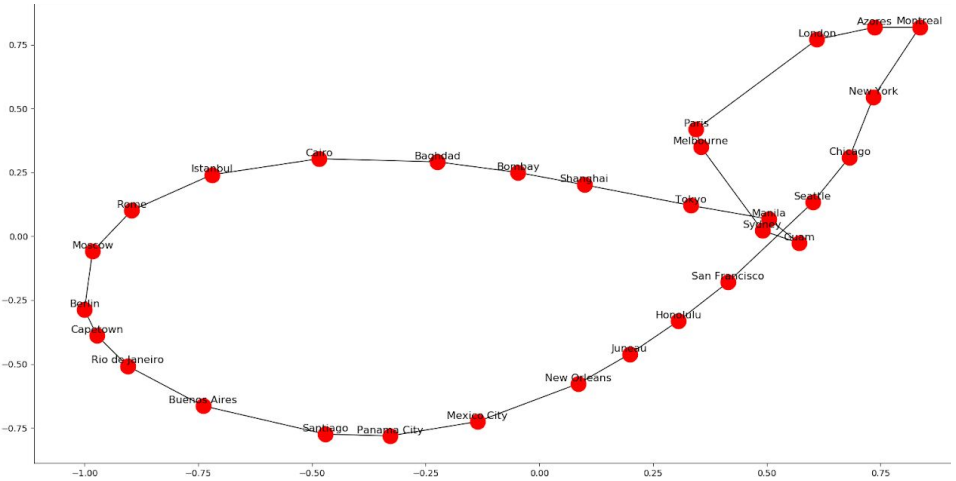

Figura 4.14: 3° Ciclo mais custoso: 617; Vértice 2 como inicial

Como podemos observar, a diferença de custo entre o ciclo mais e o menos pesado é de 103. Este é um número significativo, visto que se equivale a praticamente um quinto (⅕) do custo total do ciclo gerado pelo algoritmo com 6 de vértice inicial. Tal diferença pode ser crucial na resolução de alguns problemas, e provavelmente há ciclos ainda mais "leves" do que o encontrado.

4.5. Problema 5: Emparelhamentos estáveis e o algoritmo de Gale-Shappley

Código do arquivo Gale-Shappley.py

Inicialmente criou-se uma função para alocar em dois vetores, as listas de preferências dos homens e das mulheres.

```
def Set_PList():
    PMen = dict()
    for boy in ("abe", "bob", "col", "dan", "ed", "fred", "gav", "hal", "ian", "jon"):
        PMen[boy]=[boy]
        PMen[boy].remove(boy)

    for girl in ("abi", "eve", "cath", "ivy", "jan", "dee", "fay", "bea", "hope", "gay"):
        PMen["abe"].append(girl)
    for girl in ("cath", "hope", "abi", "dee", "eve", "fay", "bea", "jan", "ivy", "gay"):
        PMen["bob"].append(girl)
    for girl in ( "hope", "eve", "abi", "dee", "bea", "fay", "ivy", "gay", "cath", "jan"):
        PMen["col"].append(girl)
    for girl in ("ivy", "fay", "dee", "gay", "hope", "eve", "jan", "bea", "cath", "abi"):
        PMen["dan"].append(girl)
    for girl in ("jan", "dee", "bea", "cath", "fay", "eve", "abi", "ivy", "hope", "gay"):
        PMen["ed"].append(girl)
    for girl in ("bea", "abi", "dee", "gay", "eve", "ivy", "cath", "jan", "hope", "fay"):
        PMen["fred"].append(girl)
    for girl in ("gay", "eve", "ivy", "bea", "cath", "abi", "dee", "hope", "jan", "fay"):
        PMen["gav"].append(girl)
    for girl in ("abi", "eve", "hope", "fay", "ivy", "cath", "jan", "bea", "gay", "dee"):
        PMen["hal"].append(girl)
    for girl in ("abi", "eve", "hope", "fay", "ivy", "cath", "jan", "bea", "gay", "dee"):
        PMen["hal"].append(girl)
    for girl in ("hope", "cath", "dee", "gay", "bea", "abi", "fay", "ivy", "jan", "eve"):
        PMen["ian"].append(girl)
    for girl in ("abi", "fay", "jan", "gay", "eve", "bea", "dee", "cath", "ivy", "hope"):
        PMen["jon"].append(girl)


    PWom = dict()
    for girl in ("abi", "bea", "cath", "dee", "eve", "fay", "gay", "hope", "ivy", "jan"):
        PWom[girl]=[girl]
        PWom[girl].remove(girl)

    for boy in ("bob", "fred", "jon", "gav", "ian", "abe", "dan", "ed", "col", "hal"):
        PWom["abi"].append(boy)
    for boy in ("bob", "abe", "col", "fred", "gav", "dan", "ian", "ed", "jon", "hal"):
```

```
        PWom["bea"].append(boy)
    for boy in ("fred", "bob", "ed", "gav", "hal", "col", "ian", "abe", "dan", "jon"):
        PWom["cath"].append(boy)
    for boy in ("fred", "jon", "col", "abe", "ian", "hal", "gav", "dan", "bob", "ed"):
        PWom["dee"].append(boy)
    for boy in ("jon", "hal", "fred", "dan", "abe", "gav", "col", "ed", "ian", "bob"):
        PWom["eve"].append(boy)
    for boy in ("bob", "abe", "ed", "ian", "jon", "dan", "fred", "gav", "col", "hal"):
        PWom["fay"].append(boy)
    for boy in ("jon", "gav", "hal", "fred", "bob", "abe", "col", "ed", "dan", "ian"):
        PWom["gay"].append(boy)
    for boy in ("gav", "jon", "bob", "abe", "ian", "dan", "hal", "ed", "col", "fred"):
        PWom["hope"].append(boy)
    for boy in ("ian", "col", "hal", "gav", "fred", "bob", "abe", "ed", "jon", "dan"):
        PWom["ivy"].append(boy)
    for boy in ("ed", "hal", "gav", "abe", "bob", "jon", "col", "ian", "fred", "dan"):
        PWom["jan"].append(boy)
    return PMen, PWom
```

Em seguida foi desenvolvido o algoritmo necessário para a resolução do problema. Este foi feito com base no pseudo-algoritmo descrito nas notas de aula de Teoria dos Grafos do professor.

```
"""
Algoritmo GAle-Shappley

enquanto pertencer m livre (que ainda não tentou todos w pertencente à W)
    seja m esse homem
    seja w a mulher mais bem ranqueada em r(m) (para a qual m ainda não propôs)
    se w está livre:
        (m, w) "engaged" (adiciona em S temporariamente)

    senão:
        #significa que pertence (m2, w) em S
        se P(w, m2, m)
            m continua livre (vai continuar tentando)

        senão:
            #significa que P(w, m, m2)
            (m, w) "engaged
             m2 fica livre (vai tentar outras parceiras)
"""
```

Figura 4.15: Pseudo-código para Gale-Shappley retirado das notas de aula

```
def value(PersonName, PersonList):
    count = 0
    while PersonName != PersonList[count]:
        count = count + 1
    return count


def Gale_Shappley(Pman, Pwom):

    #lista de "casados"
    Engaged = []

    #listas que armazemam os homens e mulheres livres
```

```
    MenNotEngaged = []
    WomenNotEngaged = []

    #incializa as listas acima com todos os nomes
    for Person in Pman:
        MenNotEngaged.append(Person)
    for Person in Pwom:
        WomenNotEngaged.append(Person)
     while len(MenNotEngaged) != 0:

        Man = MenNotEngaged[0]
        Woman = Pman[Man][0]

        if Woman in WomenNotEngaged:
            Engaged.append((Man, Woman))
            WomenNotEngaged.remove(Woman)
            MenNotEngaged.remove(Man)

        else:
            for (u,v) in Engaged:
                if v == Woman:
                    Current_Husband = u

            if value(Man, Pwom[Woman]) > value(Current_Husband, Pwom[Woman]):
                Pman[Man].remove(Woman)

            else:
                Engaged.remove((Current_Husband, Woman))
                MenNotEngaged.append(Current_Husband)
                Pman[Current_Husband].remove(Woman)

                MenNotEngaged.remove(Man)
                Engaged.append((Man, Woman))

    return Engaged

Pman, Pwom = Set_PList()
```

A função criada retorna uma lista de “arestas” ligando o par formado.
Para homens iniciando a escolha, executamos:

CasadosMenChoice = Gale_Shappley(Pman,Pwom)

Agora, para mulheres iniciando, basta inverter a ordem dos parâmetros passados:

CasadosWomenChoice = Gale_Shappley(Pwom,Pman)

Ao chamar a função print abaixo, chegamos na resposta do problema

```
print("Men First: ",CasadosMenChoice, "\nWomen First: ",CasadosWomenChoice)
```

```
Men First:  [('bob', 'cath'), ('ed', 'jan'), ('fred', 'bea'), ('gav', 'gay'), ('hal', 'eve'),
('ian', 'hope'), ('jon', 'abi'), ('col', 'dee'), ('abe', 'ivy'), ('dan', 'fay')]
Women First:  [('jan', 'ed'), ('cath', 'bob'), ('eve', 'hal'), ('dee', 'col'), ('bea', 'fred'),
('abi', 'jon'), ('gay', 'gav'), ('hope', 'ian'), ('ivy', 'abe'), ('fay', 'dan')]
```

Para os dois casos as respostas foram verificadas realizando no papel o algoritmo aprendido em sala.

5. Considerações Finais

Torna-se nítida a ampla quantidade de conhecimentos adquiridos e reafirmados no desenvolvimento deste projeto como um todo. A experiência obtida está presente na maior aproximação aos modelos da Teoria de Grafos, bem como à linguagem Python. Esta mostrou-se inteiramente qualificada para a implementação dos algoritmos.

6. Referências

Origem da Teoria dos Grafos. Disponível em:
<https://universoracionalista.org/origem-da-teoria-dos-grafos-as-7-pontes-de-konigsberg/>

História da Teoria dos Grafos. Disponível em:
<http://www.academia.edu/5225351/Hist%C3%B3ria_da_teoria_dos_grafos>

Teoria dos Grafos - Notas de Aula. Disponível em:
<https://ava.ead.ufscar.br/pluginfile.php/599953/mod_label/intro/Notas_de_Aula_TG.pdf>

Documentação NetworkX. Disponível em:
<https://networkx.github.io/documentation/stable/tutorial.html>

Notas de Aula - Busca em Profundidade, Mário San Felice. Disponível em:
<http://www.ic.unicamp.br/~felice/ensino/2s2018/paa/aula12/buscaprofundidade.txt>

Notas de Aula - Algoritmo de Dijkstra, Mário San Felice. Disponível em:
<http://www.ic.unicamp.br/~felice/ensino/2s2018/paa/aula14/Dijkstra.txt>

Matplotlib. Dispnível em:
<https://matplotlib.org/>

Numpy. Disponível em:
<http://www.numpy.org/>