Universidade Federal de Viçosa - Campus Florestal
Teoria e Modelo de Grafos - CCF 331
Trabalho Prático 01

Trabalho Prático 01
Teoria e Modelo de Grafos

Mariana Souza - 3898
Guilherme Correa - 3059
Mateus Aparecido - 3858
Paula Gibrim - 4234

11 de Fevereiro de 2022

Sumário

1. Introdução

O projeto desenvolvido consiste na implementação de uma biblioteca capaz de manipular grafos não direcionados ponderados. A biblioteca é capaz de representar grafos não direcionados ponderados e de fornecer um conjunto de algoritmos em grafos, assim projetando a biblioteca com o intuito de ter a possibilidade de ser utilizada em outros programas.

A biblioteca desenvolvida é capaz de ler um grafo por um arquivo no formato *.txt e json, analisar o mesmo e fornecer informações para o arquivo de saída, como a ordem do grafo, tamanho, densidade, vizinhos do vértice que foi fornecido, grau do vértice, articulação, sequência de vértices visitados, número de componentes, ciclo, distância e caminho mínimo, árvore geradora mínima e verificar se um grafo é euleriano.

A linguagem de programação escolhida foi Python, uma vez que a mesma disponibiliza ferramentas para criar listas de adjacência de maneira simples, sendo esta a forma mais comum de representação de grafos. Para a colaboração dos integrantes do grupo foi utilizado o GitHub para realizar o versionamento do código e o Notion para uma melhor organização das tarefas a serem desempenhadas.

2. Desenvolvimento

Um dos primeiros passos para a organização da equipe, foi a construção de um painel de trabalho no aplicativo Notion, de forma que cada membro pudesse dividir o que contribuiu com cada parte, bem como disponibilizar links para melhor compreensão de conceitos importantes que se fizeram necessários para o funcionamento do projeto, além de um repositório do GitHub para a colaboração de todos do grupo com a implementação.

A organização do trabalho foi dividida nos arquivos *Grafo.py* e *main.py*, onde no arquivo grafo temos as funções que foram implementadas para a biblioteca e na main temos o menu que foi utilizado para imprimir e listar as opções que a biblioteca contém.

Diante disso, o primeiro passo foi a construção da classe *Grafo*, responsável por definir o que seria o grafo propriamente dito, e seus métodos, para iniciar os vértices, com o conteúdo das linhas que inicialmente são iniciadas em branco, além das funções *adicionaAresta* e *imprimeGrafo*. Assim, transformando o grafo em um objeto, torna-se muito mais claro qualquer tipo de manipulação que será feita a partir desse momento.

```
class Grafo:

    def __init__(self, vertices):
        self.vertices = vertices
        self.grafo2 = defaultdict(list)
        self.graph = []
        #conteudo das linhas em branco, vertices = numero de linhas
        self.grafo = [[0]*self.vertices for i in range(self.vertices)]



    def adiciona_aresta(self, u, v, p):
        # estou pensando em grafos direcionados simples
        self.grafo[u-1][v-1] = p  #trocar = por += ser for grafo múltiplo

        self.grafo[v-1][u-1] = p
    #so pra ficar mais facil de testar
    def mostra_matriz(self, arqOut):
        for i in range(self.vertices):
            arqOut.write(f'\n{self.grafo[i]}')
```

Figura 1: Classe *Grafo* e seus métodos, definidos no arquivo *Grafo.py.*

2.1. Arquivo de entrada

Para fazer a leitura do grafo, e defini-lo como um objeto da classe *Grafo*, utilizamos a função *leArquivo*. É nela que tem-se a abertura do arquivo, e a leitura de cada linha, fazendo a associação correspondente. A primeira linha, é definida como o número de vértices, e as outras representam as arestas de início e fim, bem como seu peso e como é um grafo ponderados as ponderações são valores reais que estão associados às arestas do grafo, como está sendo representado na Figura 2 abaixo. Com esse formato de arquivo para a entrada

foram criados arquivos com formato de texto com diferentes tamanhos de vértices para testar o desempenho da implementação.

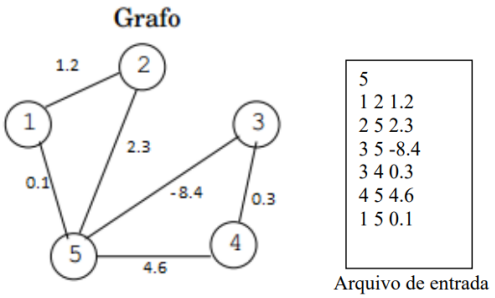

Figura 2: Exemplo do formato do arquivo de entrada.

Grafo-teste1.txt

```
100
22 84 4
30 37 15
10 67 2
24 64 6
55 82 1
73 82 10
6 67 15
55 87 2
48 53 2
47 81 14
2 61 10
57 100 12
7 87 11
79 95 12
6 29 3
5 6 7
21 68 11
55 62 6
50 53 14
10 26 8
6 34 14
67 89 1
62 98 3
26 85 11
40 74 13
92 93 12
14 34 11
```

Figura 3: Exemplo do arquivo de entrada.

```python
@staticmethod
def leArquivo(nomeArquivo):
    with open(nomeArquivo, 'r') as arq:

        #lê a primeira linha
        vertices = arq.readline()
        vertices = int(vertices)

        #cria o grafo G com a quantidade de vértices
        g = Grafo(vertices)
        for line in arq:
            u, v, peso = line.rstrip('\n').split(' ')
            g.adiciona_aresta(int(u),int(v), float(peso))
            g.addEdge(int(u)-1,int(v)-1)
            g.addEdgeKruskal(int(u)-1,int(v)-1, float(peso))

    return g
```

Figura 4: Método *leArquivo*, definido na classe *Grafo*, no arquivo *Grafo.py.*

2.2. Ordem do Grafo

Para a função de determinar a ordem do grafo foi usada a ordem de G a cardinalidade de seu conjunto de vértices, isto é, |N|, assim foi utilizado a função para ordenar o grafo *ordemGrafo( )* fazendo o cálculo e implementado na biblioteca.

```python
def ordemGrafo(self):
        return self.vertices
```

Figura 5: Função *ordemGrafo( )*, no arquivo *Grafo.py*

2.3. Tamanho do Grafo

Denomina-se tamanho de G a cardinalidade de seu conjunto de arestas, isto é, |M|. Tendo isso em vista, utilizamos a função *tamanhoGrafo( )* para fazer esse cálculo. Note que a função faz o somatório da quantidade de arestas que ele tem, onde posteriormente essa quantidade é dividida por dois, assim também foi criada a variável *somaGraus* inicializada com zero para fazer o incremento com a função que apresenta o grau do vértice.

```python
def tamanhoGrafo(self):
    somaGraus = 0
    for i in range(self.vertices):
        somaGraus += self.grauVertice(i)

    return int(self.vertices + somaGraus/2)
```

Figura 6: Função *tamanhoGrafo( )*, no arquivo *Grafo.py*

2.4. Grau do Vértice

O grau d(xi) também chamado de ou valência de um vértice xi, em um grafo não direcionado é igual ao número de arestas incidentes no vértice. Diante disso, utilizamos a função *grauVertice( )* para retornar esse valor, assim lendo o tamanho da função que retorna os vizinhos de um vértice que é fornecido, para encontrar esse vértice fornecido foi pedido que o usuário insira o mesmo no terminal ao executar o programa.

```python
def grauVertice(self, u):
    return len(self.retornaVizinhos(u))
```

Figura 7: Função *grauVertice( )*, no arquivo *Grafos.py*

2.5. Densidade

A relação entre o número de arestas e vértices, muitas vezes denominada densidade de G, pode ser medida pela razão ε(G). Dessa forma, utilizamos a função, *densidadeGrafo()* para fazer esse cálculo.

$$d_M(\mathrm{G}) = \frac{|M|}{|N|} = \frac{1}{2} d_M(\mathrm{G})$$

Figura 8: Cálculo da densidade.

```python
def densidade_grafo(self):
    return self.tamanhoGrafo() / self.ordemGrafo()
```

Figura 9: Função *densidade_grafo.*

2.6. Vizinhos do Vértice

Retorna todos os vizinhos de um vértice através da função *retornaVizinhos()*, assim foi usado uma lista de vizinhos como apresentado na Figura 10 abaixo, então para acontecer esse retorno é informado um vértice através da interação com o usuário, e dessa forma no arquivo de saída é apresentado os vizinhos dessa vértice informado pelo usuário.

```
def retornaVizinhos(self, u):
    vizinhos = []
    c =1
    listaVizinhos = self.grafo[u-1]
    for i in listaVizinhos:
        if i!=0:
            vizinhos.append(c)
        c+=1
    return vizinhos
```

Figura 10: Função retornaVizinhos.

2.7. Vértice articulado

Verifica se um determinado vértice é articulado por meio da função *ehArticulacao()*.

2.8. Busca em Largura

A busca em largura que também é conhecida com sua abreviação BFS determina a sequência de vértices visitados na busca em largura e informa as arestas que não fazem parte da árvore de busca em largura por meio da função *BFS()*.

```
def BFS(self,arqOut):
    visited = [0 for i in range(self.vertices)]

    # Add the start node to the queue
    # Node 0 in this case
    queue = [0]

    # Set the visited value of node 0 to visited
    visited[0] = 1

    # Dequeue node 0
    node = queue.pop(0)
    arqOut.write(f'{node+1} - ')
```

2.9. Componentes conexas

Determina o número de componentes conexas do grafo por meio da função *connectedComponents()* e os vértices de cada componente através da função *Util(),* como estão sendo apresentadas na imagem abaixo apresentando o método para recuperar componentes que estão conectados.

```
def Util(self, temp, v, visited):

    # Mark the current vertex as visited
    visited[v] = True

    # Store the vertex to list
    temp.append(v+1)

    # Repeat for all vertices adjacent
    # to this vertex v
    for i in self.adj[v]:
        if visited[i] == False:
            temp = self.Util(temp, i, visited)
    return temp
 # Method to retrieve connected components
# in an undirected graph
def connectedComponents(self):
    visited = []
    cc = []
    for i in range(self.vertices):
        visited.append(False)
    for v in range(self.vertices):
        if visited[v] == False:
            temp = []
            cc.append(self.Util(temp, v, visited))
    return cc
```

2.10. Grafo cíclico

Verifica se um grafo possui ciclo por meio da função *isCyclicUtil()*. Assim um grafo é considerado conexo de cada par de vértices nele está ligado por um caminho e é cíclico se apresentar um ciclo, assim apresentando uma sequência de vértices v1,v2,...vk tal que vk = v1. Também foi implementada a função *isCyclic* que retorna verdadeiro se o grafo for cíclico e falso se ele não for.

```
def isCyclicUtil(self, v, visited, recStack):

    # Mark current node as visited and
    # adds to recursion stack
    visited[v] = True
    recStack[v] = True

    # Recur for all neighbours
    # if any neighbour is visited and in
    # recStack then graph is cyclic
    for neighbour in self.grafo2[v]:
        if visited[neighbour] == False:
            if self.isCyclicUtil(neighbour, visited, recStack) == True:
                return True
        elif recStack[neighbour] == True:
            return True

    # The node needs to be poped from
    # recursion stack before function ends
    recStack[v] = False
    return False
```

2.11. Distância e Caminho mínimo

Calcula a distância e o caminho mínimo entre um dado vértice e todos os outros vértices do grafo através da função.

2.12. Árvore geradora mínima

Escreve a árvore geradora mínima em um arquivo juntamente com seu peso através da função, a árvore geradora mínima foi criada usando Kruskal que é um algoritmo que busca uma árvore geradora mínima para um grafo conexo com pesos, sendo assim este algoritmo encontra um subconjunto das arestas que forma uma árvore que inclui todos os vértices onde o peso total que é dado pela soma dos pesos das arestas da árvore é minimizado. Dessa forma foi implementada a função *KruskalMST(),* onde é a função principal para implementação deste algoritmo, mas também foram utilizadas outras funções para encontrar o conjunto de um elemento *i* e também para fazer a união dos conjuntos.

```python
def KruskalMST(self,arqOut):

    result = []  # vai armazenar a arvore resultante

    #index usado para ordenação
    i = 0

    #usado para o resultado[]
    e = 0

    '''Classifica todas as arestas em
    ordem não decrescente pelos pesos.
    Se não tivermos permissão para alterar o
    grafo, podemos criar uma cópia'''
    self.graph = sorted(self.graph,
                        key=lambda item: item[2])

    parent = []
    rank = []

    # Crie subconjuntos vertices com elementos únicos
    for node in range(self.vertices):
        parent.append(node)
        rank.append(0)
```

2.13. Euleriano

Determina se um grafo é euleriano utilizando o algoritmo de Fleury que é muito usado para identificar um ciclo euleriano, assim determina uma cadeia euleriana fechada através da função. Desta forma foram usadas funções para identificar se a aresta poderia ser considerada uma aresta no Tour de Euler, além de funções como *addEdge* que adiciona um aresta não direcionada e a *rmvEdge* que remove a aresta do grafo. Na figura abaixo também são apresentadas as funções para imprimir *printEulerUtil* que irá imprimir o tour de Euler começando pelo vértice u e printEulerTour que imprime tour começando do vértice ímpar.

```python
def addEdge(self,u,v):
    self.grafo2[u].append(v)
    self.grafo2[v].append(u)

#Função para remover a aresta u-v do grafo
def rmvEdge(self, u, v):
    for index, key in enumerate(self.grafo2[u]):
        if key == v:
            self.grafo2[u].pop(index)
    for index, key in enumerate(self.grafo2[v]):
        if key == u:
            self.grafo2[v].pop(index)
```

```
def printEulerUtil(self, u,arqOut,c):
    #Recorre para todos os vértices adjacentes a este vértice
    for v in self.grafo2[u]:
        #Se a vertice u-v não for removida e for uma próxima borda válida
        if self.isValidNextEdge(u, v):
            arqOut.write(f'{u}-{v} , ')
            self.rmvEdge(u, v)
            c+=1
            self.printEulerUtil(v,arqOut,c)

    if c==0:
        arqOut.write('Não é euleriano\n')
    else:
        return

def printEulerTour(self,arqOut):
    #encontre um vertice com grau impar
    u = 0
    for i in range(self.vertices):
        if len(self.grafo2[i]) %2 != 0 :
            u = i

            break
    # Imprimir tour começando do vértice ímpar
    self.printEulerUtil(u,arqOut,0)
```

3. Execução

Para executar o arquivo é utilizado o comando *$python main.py,* após a execução é exibida uma tela que contém as opções que podem ser escolhidas, assim é escolhido um arquivo de entrada contendo as informações do grafo e é possível escolher um arquivo de saída para exibir os resultados obtidos.

```
-------------------------IMPRIMINDO GRAFO--------------------------------------

[0, 1.0, 7.0, 0, 0]
[1.0, 0, 0, 3.0, 1.0]
[7.0, 0, 0, 0, 0]
[0, 3.0, 0, 0, 0]
[0, 1.0, 0, 0, 0]
-------------------------------------------------------------------------------
-------------------------ARQUIVO DE SAIDA--------------------------------------
Grafo de Ordem: 5

Grafo de tamanho: 9

Vizinhos do v◆rtice 1: [2, 3]


O 1 ◆ Articula◆ao!!
Densidade do Grafo: 1.8
```

```
Grau do vertice 1: 2
Ciclo euleriano:
N◆o ◆ euleriano


Arvore Geradora Minima:
1 -- 2 == 1.0
2 -- 5 == 1.0
2 -- 4 == 3.0
1 -- 3 == 7.0
Peso total: 12.0

BFS
1 - 2 - 3 - 4 - 5 -
Componentes Conexas : [[1, 3, 2, 4, 5]]

O GRAFO POSSUI CICLO
```

4. Conclusão

Ao desenvolver esse projeto, colocamos em prática vários conceitos apresentados na disciplina, bem como ter uma visualização mais clara disso. Apesar de algumas dificuldades, conseguimos desenvolver uma biblioteca funcional. O arquivo principal para resolução do trabalho é o main.py.

4. Referências

- https://algoritmosempython.com.br/cursos/algoritmos-python/algoritmos-grafos/representacao-grafos/
- https://www.ibilce.unesp.br/Home/Departamentos/MatematicaAplicada/docentes/socorro/grafos---notas-de-aula_set2018.pdf
- https://ava.ufv.br/pluginfile.php/220769/mod_resource/content/8/AULA%2001_R.pdf
- http://www2.dcc.ufmg.br/livros/algoritmos-edicao2/cap7/transp/completo4/cap7.pdf
- https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/kruskal.html