UNIVERSIDADE FEDERAL DE VIÇOSA - *CAMPUS* FLORESTAL

Gabriel Vitor da Fonseca Miranda - mat.3857
Murillo Santhiago Souza Jacob - 4243
Pedro Augusto Maia Silva - mat.3878
Victória Caroline Silva Rodrigues - mat.3584

# Trabalho Prático 1
# Teoria e Modelo de Grafos

Trabalho prático da disciplina Teoria e modelos de grafos - CCF 331, do curso de Ciência da Computação da Universidade Federal de Viçosa - Campus Florestal.

Professor: Marcus Mendes

Florestal
2022

**SUMÁRIO**


1. **Introdução…….3**
2. **Desenvolvimento……...3**

**2.1 Implementação……...4**

**2.1.1 Ordem, Tamanho e Densidade……...4**

**2.1.2 Vizinhos do Vértice……..5**

**2.1.3 Grau do Vértice ……..5**

**2.1.4 Verificar se um vértice é articulação…….5**

**2.1.5 Busca em Largura……...6**

**2.1.6 Cálculo das componentes conexas…….6**

**2.1.7 Verificar se um grafo possui ciclo……..7**

**2.1.8 Determinar distância e caminho mínimo …….8**

**2.1.9 Determinar a árvore geradora mínima…….8**

**2.1.10 Verificar se um grafo é euleriano…….8**

**2.1.11 Determinar uma cadeia euleriana fechada…….8**

**2.1.12 Algoritmo de Fleury…….9**

3. **Conclusão…….9**

**3.1 Resultados…….10**

**3.2 Considerações finais……10**

4. **Referências……11**

# 1. INTRODUÇÃO

Este trabalho consiste na modelagem e implementação de uma biblioteca que terá como finalidade manipular grafos não direcionados ponderados. As arestas do grafo terão valores reais associados para realizar as ponderações. Esses dados serão fornecidos por meio de um arquivo de texto contendo o número de vértices e suas arestas com seus respectivos pesos.

Além disso, deve-se projetar e implementar uma biblioteca que seja facilmente utilizada por outros programas, que seja capaz de representar grafos não-direcionados ponderados e fornecer um conjunto de algoritmos em grafos.

# 2. Desenvolvimento

Para a implementação do código, foi utilizada a linguagem de programação Python, juntamente com a ferramenta Git para gerenciar controle das versões do código. Com intuito de manter a organização do código e facilitar o processo de análise dos algoritmos, criamos vários arquivos separados com determinadas funções específicas no trabalho prático, além de um arquivo principal para realizar os testes.

Para o desenvolvimento deste trabalho prático foi criada uma biblioteca para manipulação de grafos, cujo grafo é representado por uma matriz de recursos. Dito isto, para fazer o encapsulamento desta biblioteca foi criada uma classe com o nome *Grafo*. Dessa forma, é importante salientar que os grafos que serão representados na matriz de recursos são aqueles cujas aresta tem pesos. Voltando para representação do grafo, temos que cada índice da matriz de recursos representa uma aresta do grafo. Nas figuras 1 e 2 temos a representação do grafo não-direcionado G, juntamente com seus pesos em cada aresta e a matriz de pesos, respectivamente.

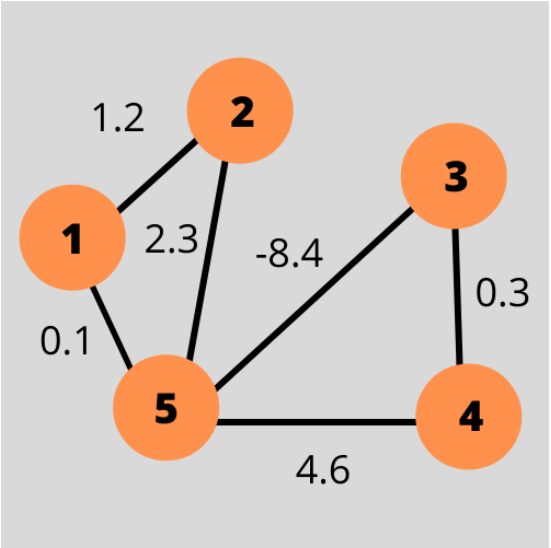


Figura 1. Grafo não-direcionado G

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1.2 | 0 | 0 | 0.1 |
| 2 | 1.2 | 0 | 0 | 0 | -8.6 |
| 3 | 0 | 0 | 0 | 0.3 | -8.4 |
| 4 | 0 | 0 | 0.3 | 0 | 4.6 |
| 5 | 0.1 | 2.3 | -8.4 | 4.6 | 0 |

Figura 2. Matriz de pesos de G

## 2.1 Implementação

A seguir será apresentado e explicado detalhadamente o funcionamento de cada uma das funções implementadas na biblioteca de grafos. Para melhor visualização, estes foram organizados em subtópicos.

### 2.1.1 Ordem, Tamanho e Densidade

Para se fazer o cálculo da ordem, tamanho e densidade do grafo no método *initGrafo*, função esta que inicializa o grafo com arquivo, foi necessário apenas saber o tamanho do grafo que já está sendo definido no arquivo de leitura, já para o tamanho foi necessário apenas saber a quantidade de vértices que se tem no grafo. Por último, para se saber a densidade bastou dividir o tamanho do grafo pela sua ordem. Para solicitar o cálculo destes

três componentes basta seguir o exemplo: *grafo.getOrdem(), grafo.getTamanho()* e *grafo.getDensidade()*.

### 2.1.2 Vizinhos do Vértice

O principal objetivo desta função é identificar os vizinhos de um determinado vértice do grafo, ou seja, como temos uma matriz com pesos, e estes pesos representam uma ligação(aresta) entre dois vértices, e tendo em vista que estes pesos podem ser negativos, sabemos que não há ligação entre dois vértices quando o grafo na posição i-ésima, posição vertice-1 for diferente de 0, ou seja, desta forma:"if grafo[i][vertice-1] != 0", se esta opção for verdadeira todos os vértices vizinhos de um determinado vértice são seus vizinhos**.** Para solicitar esta função basta digitar, *vizinhosVertice(v)*.

### 2.1.3 Grau do Vértice

Na teoria dos grafos, o grau de um vértice de um grafo é o número de arestas incidentes para com o vértice, com os laços contados duas vezes. Ou de forma análoga, o número de vértices adjacentes a ele. Analisando a Figura 1, o grau do vértice 1 seria 2, já que existem duas arestas incidentes a ele. Para se calcular o grau de um vértice nesta biblioteca de grafos basta chamar *getGrau(vertice)*.

### 2.1.4 Verificar se um vértice é articulação

Um vértice v é uma articulação se e somente se a remoção de v produz um grafo com mais componentes conexas que o grafo original.

Ou seja, um vértice é articulação quando a sua remoção se podem gerar mais grafos, se no caso a partir deste vértice aconteça uma separação do grafo surgindo dois ou mais grafos, no exemplo da Figura 3, o vértice 2 seria um vértice articulação já que se ele fosse removido o grafo seria dividido em 2. Para se verificar se um vértice é articulação basta chamar a seguinte função: *isArticulação(V)*.

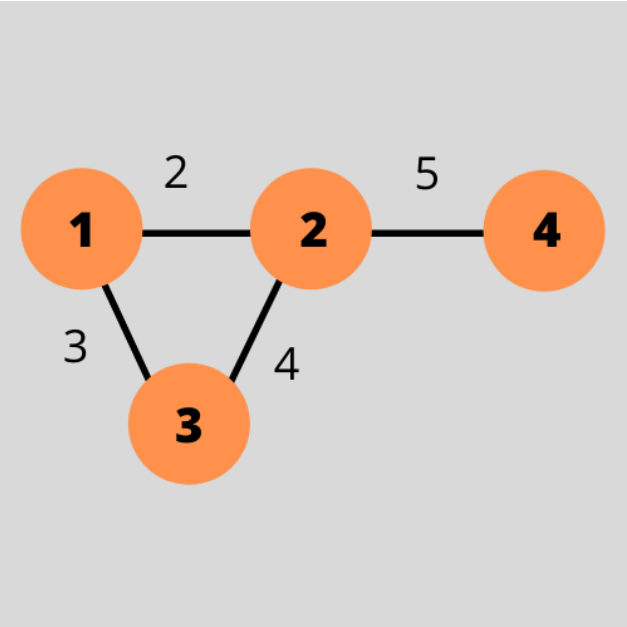


Figura 3

### 2.1.5 Busca em Largura

A função de busca em largura realiza um procedimento simples para percorrer o grafo e permite determinar quais são os vértices visitados, as arestas visitadas e as arestas não exploradas, as quais são chamadas de arestas de retorno.

O algoritmo se favorece da estrutura de lista, fila e árvore, onde os vértices são armazenados em uma lista chamada de vértices visitados, bem como acontece com os vértices explorados. Dessa forma, o vértice inicial que é passado como parâmetro da função é enfileirado, assim os vértices da lista de adjacência pertencentes ao vértice na primeira posição da fila serão explorados caso não esteja na lista de vértices marcados. O processo basicamente vai passando pelos vértices da fila e os desenfileirar, pegando a sua lista de adjacência e explorando os vértices caso não estejam já marcados.

A estrutura da árvore vai explorar todos os vértices que são percorridos a fim de determinar o grafo resultante. Essa estrutura permite analisar no segundo “if” da função se tal aresta já foi explorada, permitindo dizer qual aresta não foi explorada ou se tal aresta é uma aresta de retorno.

### 2.1.6 Cálculo das componentes conexas

Sabemos que uma componente conexa de um grafo são os subgrafos conexos maximais deste grafo, portanto, são subgrafos conexos que não estão estreitamente contidos

em outros subgrafos conexos, na figura 3 temos a representação do grafo G e suas três componentes conexas.

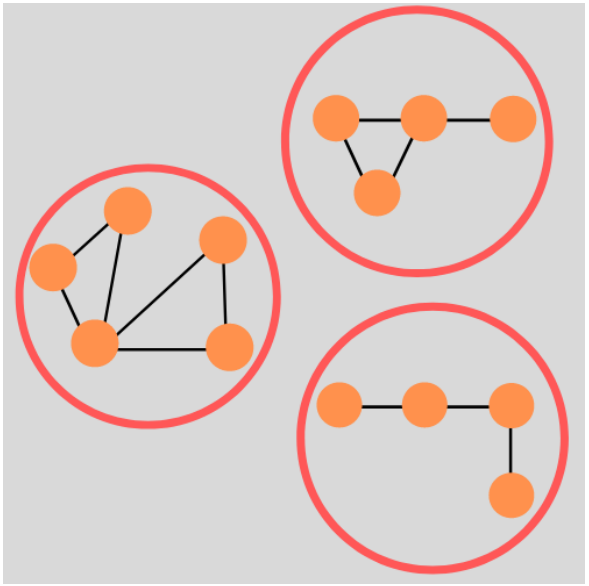

Figura 3. Grafo G e suas componentes conexas

Para realizar o cálculo das componentes conexas de um grafo não-dirigido deste trabalho foram criadas duas funções. A função *calculaComponentesConexas,* que irá percorrer todos os vértices do grafo verificando se foram visitados ou não. Caso o vértice ainda não tenha sido verificado, é chamada a função *DFS,* responsável por utilizar a lista de adjacência para verificar se os vizinhos desse vértice ainda não foram visitados.

Vale ressaltar que os grafos conexos possuem apenas uma componente conexa, e como estamos trabalhando apenas com um grafo conexo não-dirigido por vez, sempre será retornado ao final da execução apenas uma componente conexa e a ordem que os vértices foram visitados.

### 2.1.7 Verificar se um grafo possui ciclo

Um grafo possui um ciclo quando existe no mesmo uma ou mais arestas de retorno. A fim de determinar se um grafo possui ciclo utilizamos um algoritmo parecido com o algoritmo supracitado chamado de “busca em largura”. O procedimento *grafoHaveCicle* explora a característica do algoritmo de busca em largura para poder determinar as arestas de retorno de um grafo. Dessa forma, quando na função é detectado uma aresta em que o vértice já foi visitado, mas a aresta atual não foi explorada, então encontramos uma aresta de retorno.

Assim, ao determinar que existe uma aresta de retorno, a função retorna “True”, ou seja, a função basicamente retorna “True” para grafos que possuem arestas de retorno e “False” para os que não possuem arestas de retorno.

### 2.1.8 Determinar distância e caminho mínimo

O algoritmo usado foi o de Dijkstra, que utiliza dois vetores para calcular o caminho mínimo, o dt que retorna a distância entre os demais vértices, e o caminho, que retorna todos os vértices percorridos para chegar ao destino. Um exemplo, temos 5 vértices, e queremos calcular a distância mínima entre ele, e os demais vértices, e supomos que temos um retorno de dt = [0,4.5,2.4,0.3,2.3] e caminho = [[],[1,3],[1],[1],[3,4]]
Significa que a distância até o vértice 1 é 0, pois já está nele, a distância entre o vértice 2 é 4.5, passando pelo 1 e pelo 3, o menor até o 3 é passando pela própria aresta 1-3, até o 4 é passando pela aresta 1-4, e até o 5, que é passando pelo vértice 3 e 4.

### 2.1.9 Determinar a árvore geradora mínima

Uma árvore geradora mínima é um subconjunto de arestas que incluem todos os vértices, onde o peso total, dado pela soma dos pesos das arestas da árvore é minimizado. Para determinar a árvore geradora mínima foi utilizado o algoritmo de Kruskal. O algoritmo recebe como parâmetros o número de arestas do grafo e uma lista de tuplas (vértice origem, vértice destino) que formam as arestas do grafo.

A partir disso, ele ordena a lista pelo valor do peso de cada aresta. Por meio desta, o algoritmo verifica se o valor da floresta contido na posição vértice de origem e o valor da floresta na posição vértice de destino não estão na mesma árvore, armazena a aresta na árvore geradora e então troca o valor contido na floresta na posição do vértice de origem pelo valor contido na floresta na posição do vértice destino. No final da execução é retornada a árvore geradora mínima.

### 2.1.10 Verificar se um grafo é euleriano

Um grafo conexo G é Euleriano se cada vértice de G possui grau par.  Ou seja, se o grafo é euleriano todos os vértices têm grau par, e além disso, se todos os vértices do grafo tem grau par então o grafo é Euleriano.

Logo, para saber se um grafo é euleriano basta chamar o método: "*grafo.euler(self)*", ele irá retornar True se o grafo for euleriano, e false se não for.

### 2.1.11 Determinar uma cadeia euleriana fechada

Uma cadeia euleriana fechada é um trajeto orientado que inclui todas as arestas de um dado grafo G(V,A) é chamado de trajeto euleriano. Seja G um grafo conexo (fortemente ou

fracamente). Dizemos que G é euleriano se possui um trajeto euleriano fechado. Um digrafo G não-euleriano é dito ser semi-euleriano se possui um trajeto euleriano. Dessa forma, para determinar uma cadeia euleriana fechada usando esta biblioteca de grafos é necessário usar a função *cadeiaEulerianaFechada*.

#### 2.1.12 Algoritmo de Fleury

O algoritmo de Fleury é utilizado para a construção ou identificação de um ciclo euleriano em um grafo euleriano. Um caminho ou um circuito é dito euleriano se ele contém todas as arestas de um grafo. Um grafo que contém um circuito euleriano é um grafo euleriano. Para usar o método basta chamar o método *Fleury*, abaixo será demonstrado os passos do algoritmo de Fleury

O primeiro passo é certificar-se de que o gráfico tenha 0 ou 2 vértices ímpares. Se houver 0 vértices ímpares, podemos iniciar de qualquer vértice. Caso contrário, se houver 2 vértices ímpares, deve-se escolher um deles. Depois siga as bordas uma de cada vez. Caso um dos vértices seja uma ponte e o outro um não ponte, será escolhido o vértice que não é uma ponte. E por fim, finalize o algoritmo quando todos os vértices forem visitados.

## 3. Conclusão

A seguir serão apresentados, respectivamente, os resultados encontrados a partir da execução das funções descritas na aba Implementação e as considerações finais a respeito deste trabalho prático.

### 3.1 Resultados

O código em questão foi testado com o exemplo disponibilizado na especificação do trabalho prático e obteve os resultados esperados. Na tabela 1, tem-se os resultados obtidos a partir da execução das funcionalidades presente no trabalho prático.

| Funcionalidade | Entrada | Resultado |
|---|---|---|
| Ordem do grafo | - | 5 |
| Tamanho do grafo | - | 6 |
| Densidade do grafo | - | 1.2 |
| Vizinhos de um vértice | 5 | 2, 3, 4, 1 |

| | | |
|---|---|---|
| Grau do vértice | 2 | 2 |
| Vértice é articulação | 2 | Não é uma articulação |
| Sequência de vértices visitados na busca em largura | - | 1, 2, 5, 3, 4 |
| Sequência de vértices não visitado na busca em largura | - | 2, 5, 3, 4 |
| Número de componentes conexas | - | 1 |
| Os vértices das componentes conexas | - | 1, 2, 5, 3, 4 |
| Possui círculo | - | O grafo possui círculo. |
| Distância e caminho mínimo | 1 | Distância: 0 \| Caminho:Sem caminho<br>Distância: 1.2 \| Caminho:1<br>Distância: 5.0 \| Caminho:5 4<br>Distância: 4.7 \| Caminho:5<br>Distância: 0.1 \| Caminho:1 |
| Árvore geradora mínima | 7<br>1 2 1<br>1 3 1<br>2 3 1<br>3 4 1<br>3 5 1<br>3 6 1<br>3 7 1<br>4 5 1 | 1->2 \|\|\| Peso = 1<br>1->3 \|\|\| Peso = 1<br>3->4 \|\|\| Peso = 1<br>3->5 \|\|\| Peso = 1<br>3->6 \|\|\| Peso = 1<br>3->7 \|\|\| Peso = 1<br>Peso total = 6 |
| Grafo euleriano | 7<br>1 2 1<br>1 3 1<br>2 3 1<br>3 4 1<br>3 5 1<br>3 6 1<br>3 7 1<br>4 5 1 | 6-3<br>3-1<br>1-2<br>2-3<br>3-4<br>4-5<br>5-3<br>3-7 |

Tabela 1. Apresentação dos resultados obtidos em cada uma das funções da biblioteca.

### 3.2 Considerações finais

Neste trabalho prático, foi explorado um tema de grande importância para o estudo da Teoria e modelos de grafos, que são os próprios grafos, que são amplamente usados em matemática, mas sobretudo em programação. Formalmente, um grafo é uma colecção de

vértices (V) e uma colecção de arcos (E) constituídos por pares de vértices. É uma estrutura usada para representar um modelo em que existem relações entre os objectos de uma certa coleção.

O objetivo principal deste trabalho prático foi criar uma biblioteca de grafos em python, tendo esta biblioteca vários métodos que serão úteis para aplicações de grafos. Inicialmente foi pedido que a representação do grafo fosse feita por uma matriz de adjacência, nesta matriz na i-ésima, j-ésima posição teremos os pesos de cada grafo.

Diante disso, alguns desafios foram encontrados durante o desenvolvimento desta tarefa, como encontrar uma solução para a busca largura, achar o vértice articulação e entre outras funcionalidades. Porém o grupo teve um entendimento rápido para fazer estas tarefas.

Portanto, esta abordagem mais prática da implementação de um grafo e suas funcionalidade melhorou ainda mais o conhecimento adquirido em aula sobre os grafos, pois fez com que conceitos apresentados durante a disciplina fossem utilizados para a implementação desta estrutura.

## 4. Referências


**[1]** Araujo, Miguel. Grafos 1º Parte. Revista Programar, 2007. Disponível em: Link. Acesso em: 26 de Jan. de 2021.

**[2]**Srivastava, Sakshi. Program to count Number of connected components in undirected graph. GeeksforGeeks, 2021. Disponível em: Link . Acesso em: 13 de Jan. de 2021.

**[3]**Feofiloff, Paulo. Componentes conexas. ime.usp, 2019. Disponível em:Link. Acesso em: 02 de Fev. de 2022.

**[4]**Teoria dos Grafos, ufrgs, 2017. Disponível em: Link. Acesso em 02 de Fev. de 2022.

**[5]**Grau(Teoria dos grafos), Wikipedia, 2018. Disponível em:Link. Acesso em 06 de Fev. de 2022.

**[6]**Srivastava, Sakshi. Fleury's Algorithm for printing Eulerian Path or Circuit. GeeksforGeeks, 2021. Disponível em: Link . Acesso em: 13 de Jan. de 2021.