Criação de biblioteca para manipulação de grafos e implementação do algoritmo de Fleury

Ana Julia D. Aguiar[1], **Bianca Rangel Albino**[1],
Gabriel Ferreira M. Mendes[1], **Pedro Henrique A. de Souza**[1]

[1]Pontifícia Universidade Católica de Minas Gerias (PUC Minas)

anajuliadeaguiar@gmail.com, bianca_albinodev@hotmail.com

gabriel.ferreira5584@gmail.com, pedro.alves1n407@gmail.com

Abstract. *This paper aims to present the documentation and operation of a Python library for graph manipulation and an implementation, also in Python, of Fleury's algorithm to indentify Eulerian path using Naive's method from a breadth search.*

Resumo. *Este trabalho tem como objetivo apresentar a documentação e o funcionamento de uma biblioteca em Python para a manipulação de grafos e de uma implementação, também em Python, do algoritmo de Fleury para identificar caminhos eulerianos utilizando o método de Naive a partir de uma busca em largura.*

1. Entrega 1

Foi desenvolvido uma biblioteca para a manipulação de grafos não direcionados, sem arestas paralelas e loops, na linguagem Python. A seguir é possível visualizar a descrição, complexidade e em alguns casos a saída retornada dos métodos da biblioteca **Graph**. A implementação do código está disponível no repositório do GitHub: `https://github.com/Pedro-Alvess/Algorithm_in_Graphs`

1.1. Graph()

Class **Graph(n)** : Cria um grafo com n vértices, com rótulos padrões de acordo com a ordem de inicialização do vértice. Apresenta erro caso o valor n seja menor ou igual a 0.

Parâmetro: n : int: Número inteiro que representa a quantidade de vértices a serem inicializados.

Big O: O(n)

1.2. Graph().create_from_matrix

Método de classe **Graph.create_from_matrix(path_csv)**: Cria um grafo a partir de uma matriz de adjacência contido em um arquivo CSV.

Parâmetro: path_csv : str: Caminho do arquivo CSV ou apenas o nome do arquivo (caso esteja na mesma pasta do código).

Retorna: Objeto de classe iterável do tipo Graph

Big O: Melhor caso : O(n)

Caso médio : O(n^2)

Pior caso : O(n^2)

Exemplo: Graph.create_from_matrix('Graph.csv').show()

Entrada: Graph.csv :

,A,B,C,D,E

A,0,900,3,0,0

B,900,0,0,0,0

C,3,0,0,0,0

D,0,0,0,0,1

E,0,0,0,1,0

Saída:

Vertices: [['A', 1], ['B', 1], ['C', 1], ['D', 1], ['E', 1]]

Edges: [[0, 1, [900, '']], [0, 2, [3, '']], [3, 4, [1, '']]]

1.3. Graph().create_from_list

Método de classe **Graph().create_from_list(path_csv)**: Cria um grafo a partir de uma lista de adjacência contida em uma csv.

Parâmetro: path_csv : str: Caminho do arquivo CSV ou apenas o nome do arquivo (caso esteja na mesma pasta do código).

Retorna: Objeto de classe iterável do tipo Graph

Big O: O(n^2)

Exemplo: Graph.create_from_list('Lista.csv').show()

Entrada: Lista.csv :

A,B,C

B,A,

C,A,

D,E,

E,D,

Saída: Vertices: [['A', 1], ['B', 1], ['C', 1], ['D', 1], ['E', 1]]

Edges: [[0, 1, [1, '']], [0, 2, [1, '']], [3, 4, [1, '']]]

1.4. __check_vertice

Método privado **__check_vertice(v)**: Verifica a existência de um vértice dentro de um tipo de objeto Graph

Parâmetro: v : int: Número inteiro que representa a posição do vértice.

Retorna: bool

Big O: O(1)

1.5. __get_edge_index

Método privado **__get_edge_index(v,w)**: Recupera o index de uma aresta dentro da variável privada Graph._edges.

Parâmetro: v : int: Número inteiro que representa a posição de um vértice.

w : int : Número inteiro que representa a posição de um vértice.

Retorna: int

Big O: Melhor caso : O(1)

Caso médio : O(n)

Pior caso : O(n)

1.6. __get_vertice_index

Método privado **__get_vertice_index(label)**: Por meio de um rótulo, retorna a posição do vértice.

Parâmetro: label : str: String contendo um rótulo/nome de um possível vértice.

Retorna: int

Big O: Melhor caso : O(1)

Caso médio : O(n)

Pior caso : O(n)

1.7. __get_vertice_label

Método privado **__get_vertice_label(v)**: Por meio do index do vértice, retorna o seu rótulo.

Parâmetro: v : int: Número inteiro que representa a posição do vértice.

Retorna: int

Big O: O(1)

1.8. Graph().label_vertice

Método **label_vertice(v,label)**: Adiciona um rótulo ao vértice v. Apresenta erro caso o vértice v não seja encontrado.

Parâmetro: v : int: Número inteiro que representa a posição do vértice.

label : str: String que representa o rótulo a ser inserido no vértice v.

Big O: O(1)

1.9. Graph().label_edge

Método **label_edge(v,w,label)**: Adiciona um rótulo a aresta adjacente a v e w. Apresenta uma mensagem de aviso, caso o não exista uma aresta adjacente a v e w.

Parâmetro: v : int: Número inteiro que representa a posição do vértice.

w : int: Número inteiro que representa a posição do vértice.

label : str: String que representa o rótulo a ser inserido na aresta

Big O: O(1)

1.10. Graph().ponder_edge

Método **ponder_edge(v,w,weight)**: Adiciona um peso a aresta adjacente a v e w. Apresenta uma mensagem de erro, caso o não exista uma aresta adjacente a v e w.

Parâmetro: v : int: Número inteiro que representa a posição do vértice.

w : int :Número inteiro que representa a posição do vértice.

weight : int: Número inteiro a ser inserido na aresta.

Big O: O(1)

1.11. Graph().ponder_vertice

Método **ponder_vertice(v,weight)**: Adiciona um peso ao vértice v.

Parâmetro: v : int: Número inteiro que representa a posição do vértice.

weight : int: Número inteiro a ser inserido no vértice.

Big O: O(1)

1.12. __get_edge_weight

Método privado **__get_edge_weight(v,w)**: Retorna o peso da aresta adjacente a v e w. Apresenta uma mensagem de erro, caso o não exista uma aresta adjacente a v e w.

Parâmetro: v : int: Número inteiro que representa a posição de um vértice.

w : int: Número inteiro que representa a posição de um vértice.

Retorna: int

Big O: O(1)

1.13. Graph().add_edge

Método **add_edge(v,w)**: Adiciona uma aresta adjacente ao vértice v e w.

Apresenta uma mensagem de erro, caso:

- v e/ou w não exista dentro do objeto de classe Graph;
- Já exista uma aresta adjacente a v e w;
- v e w forem iguais;

Parâmetro: v : int: Número inteiro que representa a posição do vértice.

w : int: Número inteiro que representa a posição do vértice.

Big O: O(1)

1.14. Graph().remove_edge

Método **remove_edge(v,w)**: Deleta a aresta adjacente ao vértice v e w.

Apresenta uma mensagem de erro, caso não exista uma aresta adjacente ao vértice v e w.

Parâmetro: v : int: Número inteiro que representa a posição do vértice.

w : int: Número inteiro que representa a posição do vértice.

Big O: O(1)

1.15. Graph().len_vertices

Propriedade **len_vertices**: Retorna a quantidade de vértices dentro do grafo contido no objeto de classe Graph.

Retorna: int

Big O: O(1)

Exemplo:

```
G = Graph(5)
print(G.len_vertices)
```

Saída: 5

1.16. Graph().len_edges

Propriedade **len_edeges**: Retorna a quantidade de arestas dentro do grafo contido no objeto de classe Graph.

Retorna: int

Big O: O(1)

Exemplo:

```
G = Graph(5)
G.add_edge(0,1)
G.add_edge(1,4)
print(G.len_edges)
```

Saída: 2

1.17. Graph().check_empty_graph

Propriedade **check_empty_graph**: Verifica se o grafo está vazio.

Retorna: bool

Big O: O(1)

Exemplo:

```
G = Graph(5)
G.add_edge(0,2)
```

G.add_edge(1,3)

print(G.check_empty_graph)

Saída: False

1.18. Graph().check_full_graph

Propriedade **check_full_graph**: Verifica se o grafo está completo.

Retorna: bool

Big O: O(1)

Exemplo:

G = Graph(2)

G.add_edge(0,1)

print(G.check_full_graph)

Saída: True

1.19. Graph().check_adjacency_between_edges

Método **check_adjacency_between_edges(edge_1 : list, edge_2 : list)** : Verifica a adjacência entre arestas.

Parâmetro: edge_1 : list: Lista contendo dois números inteiros.

edge_2 : list: Lista contendo dois números inteiros.

Retorna: bool

Big O: O(1)

Exemplo:

G = Graph(3)

G.add_edge(0,1)

G.add_edge(2,1)

print(G.check_adjacency_between_edges([0,1],[1,2]))

Saída: True

1.20. Graph().check_adjacency_between_vertices

Método **check_adjacency_between_vertices(v, w)**: Verifica a adjacência entre vértices.

Parâmetro: v : int: Número inteiro que representa a posição do vértice.

w : int: Número inteiro que representa a posição do vértice.

Retorna: bool

Big O: O(1)

Exemplo:

G = Graph(3)

G.add_edge(0,1)

print(G.check_adjacency_between_vertices(0,1))

Saída: True

1.21. Graph().existing_edges

Método **existing_edges(v)**: Retorna todas as arestas adjacentes a v. Apresenta um aviso caso não exista nenhuma aresta adjacente a v.

Parâmetro: v : int: Número inteiro que representa a posição do vértice.

Retorna: list

Big O: O(n)

Exemplo:

G = Graph(3)

G.add_edge(0,1)

G.ponder_edge(0,1,52)

G.label_edge(0,1,'E1')

print(G.existing_edges(0))

Saída: [['0', '1', [52, 'E1']]]

1.22. Graph().adjacency_list

Método **adjacency_list()**: Retorna uma lista de adjacência.

Retorna: list

Big O: O(n^2)

Exemplo:

G = Graph(3)

G.add_edge(0,1)

G.add_edge(0,2)

G.add_edge(2,1)

print(G.adjacency_list())

Saída: [['0', '1', '2'], ['1', '0', '2'], ['2', '0', '1']]

1.23. Graph().show_adjacency_list

Método **show_adjacency_list()**: Imprime a lista de adjacência.

Big O: O(n^2)

Exemplo:

G = Graph(3)

G.add_edge(0,1)

G.add_edge(0,2)

G.add_edge(2,1)

G.show_adjacency_list

Saída:

$0 - - > 1 - - > 2$

$1 - - > 0 - - > 2$

$2 - - > 0 - - > 1$

1.24. Graph().adjacency_list_to_csv

Método **adjacency_list_to_csv(file_name = 'Graph')**: Transforma a lista de adjacência em um arquivo CSV. Caso não seja passado o nome do arquivo no parâmetro, por padrão o arquivo será gerado com o nome de "Graph.csv".

Parâmetro: file_name : str: String referente ao nome do arquivo, não é necessário passar o ".csv" junto ao nome.

Big O: O(1)

1.25. Graph().adjacency_matrix

Método **adjacency_matrix()**: Retorna uma matriz de adjacência.

Retorna: list

Big O: O(n^2)

Exemplo:

G = Graph(3)

G.add_edge(0,1)

G.add_edge(0,2)

G.add_edge(2,1)

print(G.adjacency_matrix())

Saída: [['0', '1', '2'], ['0', 0, 1, 1], ['1', 1, 0, 1], ['2', 1, 1, 0]]

O primeiro índice da lista é referente ao "cabeçalho" da matriz, e todas as lista dentro da lista principal, após a primeira lista, possuem uma string referente a "coluna" 1 da matriz.

1.26. __create_adjacency_matrix

Método privado **__create_adjacency_matrix()**: Cria uma matriz de adjacência

Big O: O(n)

1.27. Graph().show_adjacency_matrix

Método **show_adjacency_matrix()**: Imprime a matriz de adjacência.

Big O: O(1)

Exemplo:

G = Graph(3)

G.add_edge(0,1)

G.add_edge(0,2)

G.add_edge(2,1)

G.show_adjacency_matrix

Saída:

- 0 1 2

0 0 1 1

1 1 0 1

2 1 1 0

1.28. Graph().adjacency_matrix_to_csv

Método **adjacency_list_to_csv(file_name = 'Graph')**: Transforma a matriz de adjacência em um arquivo CSV. Caso não seja passado o nome do arquivo no parâmetro, por padrão o arquivo será gerado com o nome de "Graph.csv".

Parâmetro: file_name : str: String referente ao nome do arquivo, não é necessário passar o ".csv" junto ao nome.

Big O: O(1)

1.29. Graph().show

Método **show()**: Imprime uma lista de vértices e arestas.

Big O: O(1)

Exemplo:

G = Graph(5)

G.add_edge(0,1)

G.add_edge(0,3)

G.add_edge(0,4)

G.show()

Saída: Vertices: [['0', 1], ['1', 1], ['2', 1], ['3', 1], ['4', 1]

Edges: [[0, 1, [1, '']], [0, 3, [1, '']], [0, 4, [1, '']]]

1.30. Gephi

Após a chamada dos métodos adjacency_list_to_csv e adjacency_matrix_to_csv, é gerado as duas saídas respectivamente:

Saída 1:

Graph.csv =

,A,B,C,D,E

A,0,3,3,0,0

B,3,0,0,0,0

C,3,0,0,0,0

D,0,0,0,0,1

E,0,0,0,1,0

Saída 2:

Graph.csv =

A,B,C

B,A,

C,A,

D,E,

E,D,

Com o auxílio do software Gephi foi possível plotar os grafos da figura 1 e 2 a seguir.

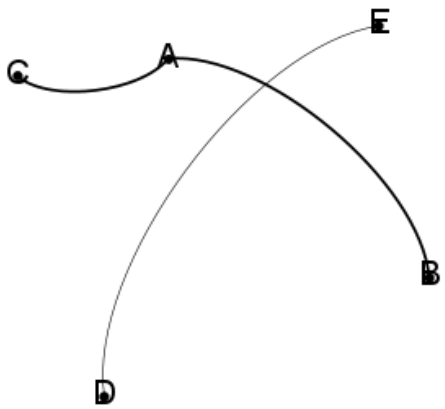

Figure 1. Grafo gerado a partir de uma matriz de adjacência

2. Entrega 2

O algoritmo de Fleury, utilizado para a identificação de ciclos eulerianos em um grafo, foi implementado em Python, através do paradigma funcional. Para essa implementação, foram utilizadas duas bibliotecas: NetworkX, para a construção e manipulação dos grafos; e Matplotlib, para a plotagem dos grafos (Figura 3).

Para a identificação de pontes, necessária durante a execução do algoritmo, foram desenvolvidas duas funções: o método de Naive, que recebe como parâmetros o grafo a ser analisado e a aresta que se deseja verificar se é ponte ou não;

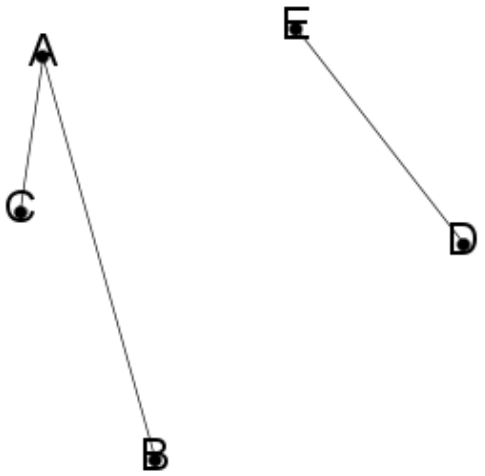

Figure 2. Grafo gerado a partir de uma lista de adjacência

2.1. naive(graph, edge)

Retorna **true** caso a aresta seja ponte e **false** caso contrário.

Big O: $O(n^2)$

É a função para busca em largura, que é utilizada no método de naive para checar a conectividade do grafo após remover a aresta.

2.2. breadth_first_search(graph, starting_node)

Retorna uma **lista** com os vértices visitados após uma busca em largura partindo do vértice inicial informado pelo segundo parâmetro da função.

Big O: $O(n^2)$

A partir dessas funções, foi implementado a função do algoritmo de Fleury, a qual recebe um grafo como parâmetro e retorna uma **lista** com o ciclo euleriano, caso exista, ou uma **lista vazia**, caso contrário:

- Inicialmente, verifica-se se o grafo possui 3 ou mais vértices com grau ímpar. Se for o caso, é retornada uma lista vazia.
- Senão, sucessivamente remove-se cada aresta e a adiciona em uma lista **trail**, escolhendo sempre uma que não seja ponte, a não ser que seja a única ligada ao vértice atual.
  - Se houver mais de uma aresta que seja ponte ligada ao vértice, a função retorna uma lista vazia.
- Caso o algoritmo, em algum momento, esteja em um vértice ao qual não possui mais arestas conectadas, porém o grafo ainda possui arestas, é retornada uma lista vazia.
- Senão, a função retorna a lista **trail**, que contém o ciclo euleriano encontrado.

2.3. fleury(graph)

Retorna uma **lista** contendo o ciclo euleriano, caso exista, ou uma **lista vazia**, caso contrário.

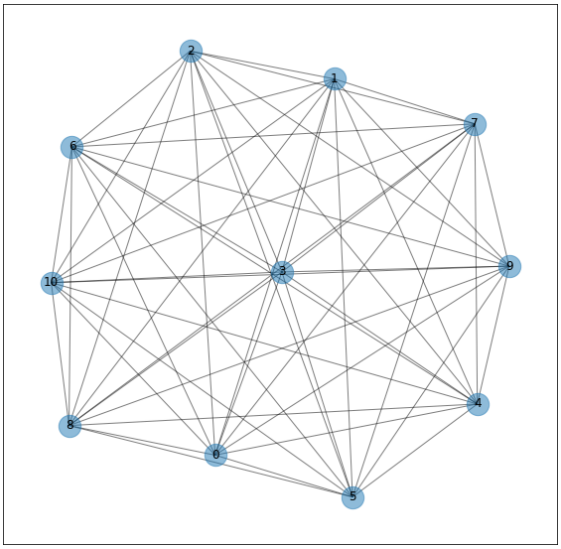

Figure 3. Grafo completo com 11 vértices

Big O: O(n^3)

Para a realização dos testes, o algoritmo foi executado no Google Colab e os seguintes tempos computacionais foram obtidos:

Nº de vértices	**Tempo computacional**
21	0s
31	1s
41	3s
51	10s
101	4min 23s
151	38min

Obs.: Os testes foram feitos com grafos completos.

Não foi possível executar o algoritmo com grafos com números maiores de vértices pois, além do tempo crescer muito, como pode ser observado na diferença entre os testes com 51, 101 e 151 vértices, o Google Colab possui um limite de memória RAM que pode ser utilizada.