Relatório Técnico - Trabalho 1

Gustavo Nogueira de Sousa[1]

[1]Disciplina de Análise e Projetos de Algoritmos
Programa de Pós-Graduação em Engenharia da Computação e Sistemas
Universidade Estadual do Maranhão (UEMA)
São Luís – MA – Brasil

{sougusta}@gmail.com

1. Introdução

Código fonte e executáveis, há três opções disponíveis com o mesmo código:

- GitHub: https://github.com/gustavons/tarefa_1_analise_projeto_algoritmos
- Google Driver: https://drive.google.com/drive/folders/1LN7O9-XAVYwVqEWPnMlsWLQwaOPt69jm?usp=sharing

Este relatório traz a explicação do funcionamento do programa de carregamento e execução de operações de verificação de adjacências de vértices, cálculo do grau, busca de vizinhos de um dado vértice, visita em todas as arestas do grafo e por fim a visualização gráfica de grafos dirigidos e não dirigidos. A implementação se deu de duas formas, sendo a primeira a representação do grafo e a segundo a visualização gráfica do grafo.

A etapa de representação do grafo inclui as seguintes operações:

- Verificação de adjacências de vértices;
- Calculo do grau de um dado vértice;
- Busca de vizinhos de um dado vértice;
- Visita em todas as arestas do grafo.

A etapa de visualização gráfica do grafo inclui a seguinte operação:

- Leitura de um grafo de um arquivo e representação gráfica de um grafo dirigido e não dirigido.

2. Representação de Grafos

2.1. Tecnologias utilizadas

A implementação foi realizada na linguagem *Python*[1] sem o auxilio de bibliotecas de terceiros.

2.2. Destaques do código - Leitura do arquivo

O processo de processamento do arquivo *"TXT"* com o grafo foi processados por dois diferentes métodos, um para leitura e carregamentos das linhas em um *array* e outro para validar as linhas e separar o que é o tipo do grafo e o que são vértices e arestas.

Na Figura 1 há o método responsável por ler os arquivos, ele recebe como parâmetro o nome do arquivo *(Ex.: "entrada.txt")* ler as linhas do arquivo e as adiciona em um *array*. O retorno deste método é uma lista das linhas do arquivo *(Ex.:* $[linha_1, linha_2, linha_3, ..., linha_n]$*)*.

[1]https://www.python.org/

Na Figura 2 há o método responsável por invocar o método descrito na Figura 1 e por validar as linhas lidas do arquivo. Nele é obtido o tipo do grafo e validado se o tipo está de acordo com as especificações do programa. Este método recebe como paraemteo o nome do arquivo do grafo e retorna as linhas do aquivo *(Ex.:* $[linha_1, linha_2, linha_3, ..., linha_n]$*)* e o tipo do grafo *(Ex.: "ND").*

```
def retorna_linhas_arquivos(nome_arquivo):
    with open(nome_arquivo, "r") as arquivo:
        linhas = [linha.strip() for linha in arquivo.readlines()]
    return linhas
```

Figura 1. Ler arquivo e retorna todas as linhas presentes

```
def ler_grafo(nome_arquivo):
    print(f"Processando o arquivo {nome_arquivo}")
    linhas = retorna_linhas_arquivos(nome_arquivo)
    tipo_grafo = linhas[0]
    if tipo_grafo != DIGIRIDO and tipo_grafo != NAO_DIRIGIDO:
        raise Exception("Tipo do grafo não especificado corretamente")
    return linhas, tipo_grafo
```

Figura 2. Preprocessa as linhas lidas de um arquivos.

2.3. Destaques do código - Geração matriz de adjacência

A matriz de adjacência é gerada por meio de quatro métodos que auxiliam nos processos, o primeiro trata dos ajustes das adjacências, o segundo trata os índices que os vértices terão na matriz, o terceiro é responsável por inserir o valor *"1"* quando dois vértices tiverem adjacências e o ultimo, o quarto, é responsável por estruturar todo o processo de geração da matriz.

Na Figura 3 há o método responsável por ajustar as adjacências dos vértices na matriz quando um novo vértice é inserido. O método recebe dois dicionários[2] do Python (nomes: *"matriz"e "mapeamento_indices_nos"*) e um *array* (nome: *"indices"*) com os índices dos vértices que já estão na matriz de adjacência. Dado uma entrada de uma *matrizEntrada* 2x2 o retorno será uma *matrizRetorno* 3x3, como segue abaixo:

$$matrizEntrada = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \Longrightarrow matrizRetorno = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

[2]https://docs.python.org/3/tutorial/datastructures.html#dictionaries

```
def ajustar_matriz(matriz, mapeamento_indices_nos, indices):

    for ind in indices:
        matriz[mapeamento_indices_nos[ind]].append(0)

    return matriz
```

Figura 3. Método responsável por ajustar adjacências da matriz.

Na Figura 4 há o método responsável por tratar os índices dos vértices na matriz de adjacências. Como parâmetros, o método recebe três dicionários[3] do Python (nomes: *"matriz", "mapeamento_nos_indices"e "mapeamento_indices_nos"*) e uma *string* (nome: *"elemento"*) que representa o vértice a ser inserido na matriz. A execução do método ocorre da forma a seguir:

- Dado que os parâmetros são: matriz = {}, mapeamento_nos_indices = {} e mapeamento_indices_nos = {} e elemento = "a";
- A primeira operação é verificar se o dicionário "matriz"tem o valor da variável "elemento": *"if not matriz.get(elemento)"*
- Caso o dicionário "matriz"já contenha o valor da variável "elemento"o método será finalizado;
- Caso contrario, será executado o código que está dentro do condicional:
  - Em primeiro lugar seriam retornados todos os valores (índices) do dicionário *"mapeamento_nos_indices"*: *"todos_indices = mapeamento_nos_indices.values()"*;
  - Em seguida é feito uma contagem da quantidade de índices: *"indice = len(todos_indices)"*;
  - Os dicionários de mapeamento de índices e vértices são atualizados de acordo com a identificação do vértice na variável "elemento": *"mapeamento_nos_indices[elemento] = indice"* e *"mapeamento_indices_nos[indice] = elemento"*;
  - Caso o valor presenta na variável "elemento"não seja o primeiro índice a ser inserido, ele será terá em todas as adjacência "povoadas"como o valor "0": *"matriz[elemento] = [0] * (indice+1)"*;
  - Caso o valor presenta na variável "elemento"seja o primeiro índice a ser inserido, ele receber um *array* com um elemento: *"matriz[elemento] = [0]"*
  - Como ultima etapa do processo é invocada o método *"ajusta_matriz"*.
- De acordo os parâmetros de exemplo dados nestes exemplo, o método iria retorna uma *"matriz"* 1x1 (*[0]*), o *"mapeamento_nos_indices"* seria o valor {*"a": 1*} e o *"mapeamento_indices_nos"* seria o valor {*1: "a"*}.

[3]https://docs.python.org/3/tutorial/datastructures.html#dictionaries

```python
def prepara_matriz_indices(matriz, mapeamento_nos_indices, mapeamento_indices_nos, elemento):
    if not matriz.get(elemento):
        todos_indices = mapeamento_nos_indices.values()
        indice = len(todos_indices)
        mapeamento_nos_indices[elemento] = indice
        mapeamento_indices_nos[indice] = elemento
        if not indice == 0:
            matriz[elemento] = [0] * (indice + 1)
        else:
            matriz[elemento] = [0]
        matriz = ajustar_matriz(matriz, mapeamento_indices_nos, list(todos_indices)[0:indice])
    return matriz, mapeamento_nos_indices, mapeamento_indices_nos
```

Figura 4. Método responsável por inserir, formatar e ajustar os índices dos vértices na matriz.

Na Figura 5 há o método que inseri uma adjacência na matriz, ou seja, atribui o valor *"1"* na matriz para os vértices que são adjacentes. O método recebe dois dicionários[4] do Python (nomes: *"matriz"e "mapeamento_indices_nos"*) e os dois vértices adjacentes (nomes: *elemento_1 e elemento_2*. Dado uma entrada de uma matriz 2x2 de 0's e vértice *"a"* na variável "elemento_1"e o vértice *"b"* na variável "elemento_2", sendo que o índice do vértice *"a"* é o "0"e o índice do vértice *"b"* é "1"o retorno do método seria a matriz abaixo:

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

```python
def inserir_adjacencia(matriz, mapeamento_nos_indices, elemento_1, elemento_2):
    matriz[elemento_1][mapeamento_nos_indices[elemento_2]] = 1
    return matriz
```

Figura 5. Método responsável por inserir a adjacência dos vértices na matriz.

Na Figura 6 há o método responsável por coordenar o processo de geração da matriz de adjacência. Como parâmetro ele recebe um *array* com todas as relações entre vértices e retorna a matriz de adjacência gerada juntamente com dois dicionários que mapeiam os índices e vértices, sendo que o primeiro dicionário (*"mapeamento_nos_indices"*) contêm as identificação dos vértices nas *"keys"* e os *"values"* são os seus respectivos índices na matriz, o segundo dicionário (*"mapeamento_indices_nos"*) é o inverso do primeiro em que nas *"keys"* estão os índices e nos *"values"* estão a identificação de cada vértice.

[4]https://docs.python.org/3/tutorial/datastructures.html#dictionaries

```python
def gerar_matriz(relacoes_elementos):
    mapeamento_nos_indices = {}
    mapeamento_indices_nos = {}
    matriz = {}
    for relacao in relacoes_elementos:
        elemento_1, elemento_2 = relacao.split(",")
        matriz, mapeamento_nos_indices, mapeamento_indices_nos = prepara_matriz_indices(
            matriz, mapeamento_nos_indices, mapeamento_indices_nos, elemento_1
        )
        matriz, mapeamento_nos_indices, mapeamento_indices_nos = prepara_matriz_indices(
            matriz, mapeamento_nos_indices, mapeamento_indices_nos, elemento_2
        )
        matriz = inserir_adjacencia(matriz, mapeamento_nos_indices, elemento_1, elemento_2)

    return matriz, mapeamento_nos_indices, mapeamento_indices_nos
```

Figura 6. Método responsável pelo fluxo de geração da matriz.

2.4. Destaques do código - Verificação se dois vértices são adjacentes

Na Figura **??** há o método responsável por verificar se dois nós são adjacentes. Esta operação só é possível se o método descrito na Figura 6 já tiver sido executado. Este método recebe como parâmetro a *"matriz"e "mapeamento_nos_indices"* já obtidos pelos outros métodos e a identificação do primeiro vértice *"vert_1"* e do segundo vértice *"vert_2"*. O funcionamento do método é descrito a seguir:

- Dado uma que o parâmetro *matriz* = $\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$
- Dado também que o *mapeamento_nos_indices" = "{"a":0, "b":1}"*. Sendo que o vértice *"a"* no índice *"0"* e o vértice *"b"* no índice *"1"*;
- Deseja-se verificar se o vértice "a"e "b"são adjacentes. Nisto o *"vert_1"* recebe o valor *"a"* e o *"vert_2"* recebe o valor *"b"*;
- A execução do método inicia com um condicional *"if"* que verifica se os vértices passados estão na *"matriz"* de adjacência e fazem parte do grafo. Se algum dos elementos não fizerem parte do grafo uma exceção será lançada e o método encerrado;
- Se os vértices buscados fazerem parte do grafo será buscado no dicionário *"mapeamento_nos_indices"* os indices de cada vertice:
  - *"ind_vert_1 = mapeamento_nos_indices[vert_1.lower()]"*
  - *"ind_vert_2 = mapeamento_nos_indices[vert_2.lower()]"*
- Depois é buscado o valor da adjacencia na *"matriz"*:
  - *"valor_1 = matriz[vert_1.lower()][ind_vert_1]"*
  - *"valor_2 = matriz[vert_2.lower()][ind_vert_2]"*
- Caso o *"valor_1"* ou o *"valor_2"* tenha o valor *"1"* os vértices serão considerados adjacentes e o método retornará "True". Caso contrário os vértices não serão considerados adjacentes e o método retornará "False.

```python
def verifica_se_nos_sao_adjacentes(matriz, mapeamento_nos_indices, vert_1, vert_2):

    if mapeamento_nos_indices.get(vert_1.lower()) is None or mapeamento_nos_indices.get(vert_2.lower()) is None:
        raise Exception("Os vertices infomados na consuta de adjacencia não fazem parte do grafo")

    ind_vert_1 = mapeamento_nos_indices[vert_1.lower()]
    ind_vert_2 = mapeamento_nos_indices[vert_2.lower()]

    valor_1 = matriz[vert_1.lower()][ind_vert_2]
    valor_2 = matriz[vert_2.lower()][ind_vert_1]

    return valor_1 == 1 or valor_2 == 1
```

Figura 7. Método responsável verificar se dois vértices são adjacentes.

2.5. Destaques do código - Calcula grau de um vértice em um grafo não dirigido

Na Figura 10 há o método para calcular o grau de um vértices em um grafo não dirigidos. Esta operação só é possível se o método descrito na Figura 6 já tiver sido executado. Este método recebe como parâmetro a *"matriz"e "mapeamento_nos_indices"* já obtidos pelos outros métodos, e além destes é necessário passar como parâmetros a identificação do vértices buscado (*"vertice_buscado"*) e o tipo do grafo (*"vertice_buscado"*). A execução é descrita a seguir:

- Dado uma que o parâmetro *matriz* = $\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$
- Dado também que o *mapeamento_nos_indices" = "{"a":0, "b":1}"*. Sendo que o vértice *"a"* no índice *"0"* e o vértice *"b"* no índice *"1"*;
- Sendo o *"vertice_buscado"= "a"* e o *"tipo_grafo"= "ND"* em que "ND"significa que o grafo é não dirigido;
- Execução:
  - A identificação do vértice é modificado para minusculo: *"vertice_buscado = vertice_buscado.lower()"*;
  - Em seguida é verificado se o vértice faz parte do grafo através do condicional *"if"*. E caso ele não pertença uma exceção será lançada;
  - De acordo com o parâmetros de exemplo, o vértice buscado faz parte do grafo. E portanto, o método prosseguirá e será montado um *array* com todas as identificações de todos os vértices do grafo: *"lista_de_todos_vertices = list(mapeamento_nos_indices.keys())"*;
  - O elemento buscado será removido do *array "lista_de_todos_vertices"*: *"lista_de_todos_vertices.remove(vertice_buscado)"*;
  - Na Figura 8 é mostrado o trecho de código seguinte. Ele é responsável por verificar para todo vértices (exceto o buscado) se são adjacentes (usando o método da Figura 7) ao vértice buscado. Caso um vértice for adjacente ele será adicionado ao *array "lista_vertices_adjacentes"*: *"lista_vertices_adjacentes.append(vert_da_vez)"*;
  - Em seguida é executado o trecho de código da Figura 9. Nele é verificado se o tipo do grafo está de acordo com o método por meio de um condicional *"if"*. Se o tipo de grafo estiver de acordo será verificado se o vértice buscado tem uma aresta apontando para ele mesmo, se houver, será adicionado 2 elementos a *"lista_vertices_adjacentes"*, pois esse tipo aresta vale

por 2 graus, Feito isso, é retornado de *"lista_vertices_adjacentes"* no qual a quantidade de elementos representa o grau do vértices.

- No caso do parâmetros de exemplo o valor retornado será um *array = ["b"]*. O que diz que o vértice buscado tem grau *1*.

```
if tipo_grafo == NAO_DIRIGIDO:
    valor = matriz[vertice_buscado][mapeamento_nos_indices[vertice_buscado]]
    if valor == 1:
        lista_vertices_adjacentes.append(vertice_buscado)
        lista_vertices_adjacentes.append(vertice_buscado)

    return lista_vertices_adjacentes
```

Figura 8. Parte do método responsável por calcular o grau de um vértice em um grafo não dirigido. Método completo pode ser visto na Figura 10

```
lista_vertices_adjacentes = []
for vert_da_vez in lista_de_todos_vertices:
    retorno = verifica_se_nos_sao_adjacentes(matriz, mapeamento_nos_indices, vertice_buscado, vert_da_vez)
    if retorno:
        lista_vertices_adjacentes.append(vert_da_vez)
```

Figura 9. Parte do método método responsável por calcular o grau de um vértice em um grafo não dirigido. Método completo pode ser visto na Figura 10

```
def calcula_grau_vertice_grafo_nao_dirigido(matriz, mapeamento_nos_indices, vertice_buscado, tipo_grafo):
    vertice_buscado = vertice_buscado.lower()
    if mapeamento_nos_indices.get(vertice_buscado.lower()) is None:
        raise Exception("O vertice buscado infomado na consuta de grau não faz parte do grafo")

    lista_de_todos_vertices = list(mapeamento_nos_indices.keys())
    lista_de_todos_vertices.remove(vertice_buscado)

    lista_vertices_adjacentes = []
    for vert_da_vez in lista_de_todos_vertices:
        retorno = verifica_se_nos_sao_adjacentes(matriz, mapeamento_nos_indices, vertice_buscado, vert_da_vez)
        if retorno:
            lista_vertices_adjacentes.append(vert_da_vez)

    if tipo_grafo == NAO_DIRIGIDO:
        valor = matriz[vertice_buscado][mapeamento_nos_indices[vertice_buscado]]
        if valor == 1:
            lista_vertices_adjacentes.append(vertice_buscado)
            lista_vertices_adjacentes.append(vertice_buscado)

        return lista_vertices_adjacentes
```

Figura 10. Método responsável por calcular o grau de um vértice em um grafo não dirigido.

2.6. Destaques do código - Calcula grau de um vértice em um grafo dirigido

Na Figura 14 há o método para calcular o grau de um vértices em um grafo dirigido. Esta operação só é possível se o método descrito na Figura 6 já tiver sido executado.

Este método recebe como parâmetro a *"matriz", "mapeamento_nos_indices"e "mapeamento_indices_nos"* já obtidos pelos outros métodos, e além destes é necessário passar como parâmetros a identificação do vértices buscado (*"vertice_buscado"*) e o tipo do grafo (*"vertice_buscado"*). A execução é descrita a seguir:

- Dado uma que o parâmetro *matriz* = $\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$
- Dado também que o *mapeamento_nos_indices" = "{"a":0, "b":1}"*. Sendo que o vértice *"a"* no índice *"0"* e o vértice *"b"* no índice *"1"*;
- Sendo o *"vertice_buscado"= "a"* e o *"tipo_grafo"= "ND"* em que "ND"significa que o grafo é não dirigido;
- Execução:
  - A identificação do vértice é modificado para minusculo: *"vertice_buscado = vertice_buscado.lower()"*;
  - Em seguida é verificado se o vértice faz parte do grafo através do condicional *"if"*. E caso ele não pertença uma exceção será lançada;
  - De acordo com o parâmetros de exemplo, o vértice buscado faz parte do grafo. E portanto, o método prosseguirá e executará a parte do método presente na Figura 11, onde serão listadas todos os vértices partindo do *"vertice_buscado"*: *"arestas_saindo_do_vertice_buscado = matriz[vertice_buscado]"*. Depois disto, o restante do trecho de código irá montar um *array* (*"list_indices_emissao"*) com as identificação de todos os vértices a quem as arestas partindo do *"vertice_buscado"*;
  - Depois disto, o método prosseguirá e executará a parte do método presente na Figura 12, onde será montado um *array* com todas as identificações de todos os vértices do grafo: *"lista_de_todos_vertices = list(mapeamento_nos_indices.keys())"*. E o elemento buscado será removido do *array "lista_de_todos_vertices"*: *"lista_de_todos_vertices.remove(vertice_buscado)"*. A partir disto, o trecho de código restante será executado, no qual é responsável por verificar para todo vértices (exceto o buscado) se há arestas em direção ao *"vertice_buscado"*. Caso um vértice tiver aresta em direção ao *"vertice_buscado"* ele será adicionado ao *array "list_indices_recepcao"*: *"list_indices_recepcao.append(vert_da_vez)"*;
  - Em seguida é executado o trecho de código da Figura 13. Nele é verificado se o tipo do grafo está de acordo com o método por meio de um condicional *"if"*. Se o tipo de grafo estiver de acordo será verificado se o vértice buscado tem uma aresta apontando para ele mesmo, se houver, será adicionado 2 elementos a *"lista_vertices_adjacentes"*, pois esse tipo aresta vale por 2 graus;
  - Feito isso, é retornado de *"list_indices_recepcao"* e a *"list_indices_emissao"* nos quais a quantidade de elementos representa o grau do vértices;
- No caso do parâmetros de exemplo o valor retornado será retornado o *array "list_indices_recepcao" = []* e a *"list_indices_emissao" = ["b"]*. O que diz que o vértice buscado tem grau de recepção igual a *0* e grau de emissão igual a *1*.

```
arestas_saindo_do_vertice_buscado = matriz[vertice_buscado]
list_indices_emissao = []
for indice in range(0, len(arestas_saindo_do_vertice_buscado)):
    aresta = arestas_saindo_do_vertice_buscado[indice]
    if aresta == 1:
        list_indices_emissao.append(mapeamento_indices_nos[indice])
```

Figura 11. Parte do método método responsável por calcular o grau de um vértice em um grafo dirigido. Método completo pode ser visto na Figura 14.

```
lista_de_todos_vertices = list(mapeamento_nos_indices.keys())
lista_de_todos_vertices.remove(vertice_buscado)
list_indices_recepcao = []

for vert_da_vez in lista_de_todos_vertices:
    valor = matriz[vert_da_vez][mapeamento_nos_indices[vertice_buscado]]
    if valor == 1:
        list_indices_recepcao.append(vert_da_vez)
```

Figura 12. Parte do método método responsável por calcular o grau de um vértice em um grafo dirigido. Método completo pode ser visto na Figura 14.

```
if tipo_grafo == DIGIRIDO:
    valor = matriz[vertice_buscado][mapeamento_nos_indices[vertice_buscado]]
    if valor == 1:
        list_indices_recepcao.append(vertice_buscado)
        list_indices_emissao.append(vertice_buscado)
```

Figura 13. Parte do método método responsável por calcular o grau de um vértice em um grafo dirigido. Método completo pode ser visto na Figura 14.

```python
def calcula_grau_vertice_grafo_dirigido(matriz, mapeamento_nos_indices, mapeamento_indices_nos, vertice_buscado, tipo_grafo):
    vertice_buscado = vertice_buscado.lower()
    if mapeamento_nos_indices.get(vertice_buscado.lower()) is None:
        raise Exception("O vertice buscado infomado na consuta de grau não faz parte do grafo")

    arestas_saindo_do_vertice_buscado = matriz[vertice_buscado]
    list_indices_emissao = []
    for indice in range(0,len(arestas_saindo_do_vertice_buscado)):
        aresta = arestas_saindo_do_vertice_buscado[indice]
        if aresta == 1:
            list_indices_emissao.append(mapeamento_indices_nos[indice])

    lista_de_todos_vertices = list(mapeamento_nos_indices.keys())
    lista_de_todos_vertices.remove(vertice_buscado)
    list_indices_recepcao = []

    for vert_da_vez in lista_de_todos_vertices:
        valor = matriz[vert_da_vez][mapeamento_nos_indices[vertice_buscado]]
        if valor == 1:
            list_indices_recepcao.append(vert_da_vez)

    if tipo_grafo == DIGIRIDO:
        valor = matriz[vertice_buscado][mapeamento_nos_indices[vertice_buscado]]
        if valor == 1:
            list_indices_recepcao.append(vertice_buscado)
            list_indices_emissao.append(vertice_buscado)

    return list_indices_recepcao, list_indices_emissao
```

Figura 14. Método responsável por calcular o grau de um vértice em um grafo dirigido.

2.7. Destaques do código - Busca todos os vizinhos de um dado vértice

Na Figura 15 há o método responsável por buscar todos os vizinhos de um dado vértice. Esta operação só é possível se o método descrito na Figura 6 já tiver sido executado. Este método recebe como parâmetro a *"matriz"e "mapeamento_nos_indices"* já obtidos pelos outros métodos, e além destes é necessário passar como parâmetros a identificação do vértices buscado (*"vertice_buscado"*). A execução é descrita a seguir:

- Dado uma que o parâmetro *matriz* = $\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$
- Dado também que o *mapeamento_nos_indices" = "{"a":0, "b":1}"*. Sendo que o vértice *"a"* no índice *"0"* e o vértice *"b"* no índice *"1"*;
- Sendo o *"vertice_buscado"= "a"* e o *"tipo_grafo"= "ND"* em que "ND"significa que o grafo é não dirigido;
- Execução:
  - A identificação do vértice é modificado para minusculo: *"vertice_buscado = vertice_buscado.lower()"*;
  - Em seguida é verificado se o vértice faz parte do grafo através do condicional *"if"*. E caso ele não pertença uma exceção será lançada;
  - De acordo com o parâmetros de exemplo, o vértice buscado faz parte do grafo. E portanto, será montado um *array* com todas as identificações de todos os vértices do grafo: *"lista_de_todos_vertices = list(mapeamento_nos_indices.keys())"*. Em seguida o elemento buscado será removido do *array "lista_de_todos_vertices"*: *"lista_de_todos_vertices.remove(vertice_buscado)"*.
  - A partir disto, o trecho de código restante será executado e será responsável por verificar para todo vértices (exceto o buscado) se são adjacentes (usando o método da Figura 7) ao vértice buscado. Caso um vértice

for adjacente ele será adicionado ao *array "lista_vertices_adjacentes"*: *"lista_vertices_adjacentes.append(vert_da_vez)"*;
  - Por fim, será retornado a lista de todos os vértices adjacentes: *"return lista_vertices_adjacentes"*
- No caso do parâmetros de exemplo o valor retornado será retornado o *array "lista_vertices_adjacentes" = ["b"]*. O que diz que o vértice buscado tem apenas o vértice *"b"* como vizinho.

```python
def busca_todos_os_vizinhos(matriz, mapeamento_nos_indices, vertice_buscado):
    vertice_buscado = vertice_buscado.lower()
    if mapeamento_nos_indices.get(vertice_buscado.lower()) is None:
        raise Exception("O vertice buscado infomado na consuta de grau não faz parte do grafo")

    lista_de_todos_vertices = list(mapeamento_nos_indices.keys())
    lista_de_todos_vertices.remove(vertice_buscado)

    lista_vertices_adjacentes = []
    for vert_da_vez in lista_de_todos_vertices:
        retorno = verifica_se_nos_sao_adjacentes(matriz, mapeamento_nos_indices, vertice_buscado, vert_da_vez)
        if retorno:
            lista_vertices_adjacentes.append(vert_da_vez)

    return lista_vertices_adjacentes
```

Figura 15. Método responsável por buscar todos os vizinho de um vértice.

2.8. Destaques do código - Visita todas as arestas do grafo

Na Figura 16 há o método responsável por visitar todas as arestas do grafo. Por meio deste, será impresso na tela todas as arestas e quais vértices elas estão conectando.

```python
def visitar_arestas_grafo(matriz, mapeamento_nos_indices, mapeamento_indices_nos):
    todos_os_vertices = list(mapeamento_nos_indices.keys())

    for vertice in todos_os_vertices:
        lista_de_arestas = matriz[vertice]

        for i in range(0, len(lista_de_arestas)):
            if lista_de_arestas[i] == 1:
                print(f"Passou pela aresta {vertice} - {mapeamento_indices_nos[i]}")
```

Figura 16. Método responsável por visitar todas as arestas do grafo.

3. Visualização de Grafos

3.1. Tecnologias utilizadas

A implementação foi realizada na linguagem *Python*[5] juntamente com a biblioteca da linguagem chamada *Networkx*[6] para a geração da visualização gráfica do grafo.

[5]https://www.python.org/
[6]https://networkx.github.io/

3.2. Destaques do código - Gera visualização do grafo

Na Figura 17 há o método responsável por ligar por meio de aresta dois vértices. Ele recebe um objeto gerado pela biblioteca *"G_symmetric"* e uma lista com as relações dos vértices *"relacoes_elementos"*. Por meio deste, uma aresta será adicionada a geração do imagem do grafo: *"G_symmetric.add_edge(elemento_1, elemento_2)"*.

```
def gerar_grafo_img(G_symmetric, relacoes_elementos):
    for relacao in relacoes_elementos:
        elemento_1, elemento_2 = relacao.split(",")
        G_symmetric.add_edge(elemento_1, elemento_2)
```

Figura 17. Trecho responsável por conectar vértices por meio das arestas grafo.

Na Figura 20 há o trecho do código responsável por organizar o processo de geração da imagem do grafo. No "sub-trecho"na Figura 18 têm os códigos responsáveis por:

- Ler o grafo de arquivo *"TXT"*: *"linhas, tipo_grafo = ler_grafo(nome_arquivo"*;
- E caso o grafo for dirigido o objeto *"G_symmetric"* será gerado: *"G_symmetric = nx.DiGraph()"*
- Caso o grafo for não dirigido o objeto *"G_symmetric"* será gerado: *"G_symmetric = nx.Graph()"*;
- Se o tipo do grafo não for nenhum expecificado uma exceção será lançada.

No "sub-trecho"na Figura 19 há os códigos responsáveis por:

- É responsável por ligar por meio de aresta todos os vértices que tem relação: *"gerar_grafo_img(G_symmetric, relacoes_elementos)"*;
- Definir tamanho da arestas que serão geradas na imagem: *"pos = nx.spring_layout(G_symmetric,k=0.3,iterations=20)"*
- E por gerar a imagem do grafo: *"nx.draw_networkx(G_symmetric, pos)"*;

```
linhas, tipo_grafo = ler_grafo(nome_arquivo)
relacoes_elementos = linhas[1:]
if linhas[0] == DIGIRIDO:
    G_symmetric = nx.DiGraph()
elif linhas[0] == NAO_DIRIGIDO:
    G_symmetric = nx.Graph()
else:
    raise Exception(
        f"Tipo de grafo inserido incorretamente. Por favor, edite o arquivo insira na "
        f"primeira linha {DIGIRIDO} para grafo dirigido e {NAO_DIRIGIDO} para não dirigido.")
```

Figura 18. Parte do código do trecho do código expresso na Figura 20.

```
gerar_grafo_img(G_symmetric, relacoes_elementos)
pos = nx.spring_layout(G_symmetric,k=0.3,iterations=20)
nx.draw_networkx(G_symmetric, pos)
```

Figura 19. Parte do código do trecho do código expresso na Figura 20.

```
linhas, tipo_grafo = ler_grafo(nome_arquivo)
relacoes_elementos = linhas[1:]
if linhas[0] == DIGIRIDO:
    G_symmetric = nx.DiGraph()
elif linhas[0] == NAO_DIRIGIDO:
    G_symmetric = nx.Graph()
else:
    raise Exception(
        f"Tipo de grafo inserido incorretamente. Por favor, edite o arquivo insira na "
        f"primeira linha {DIGIRIDO} para grafo dirigido e {NAO_DIRIGIDO} para não dirigido.")

gerar_grafo_img(G_symmetric, relacoes_elementos)
pos = nx.spring_layout(G_symmetric,k=0.3,iterations=20)
nx.draw_networkx(G_symmetric, pos)
plt.show()
```

Figura 20. Trecho responsável por organizar e gerar imagem do grafo.

Como exemplo da geração da imagem é possível observar a imagem do grafo não dirigido na Figura 22 gerada a partir do arquivo ilustrado na Figura 21.

```
1  ND
2  a,b
3  c,d
4  e,f
5  f,g
6  a,e
7  e,h
8  b,a
9  a,a
10 c,d
11 f,d
```

Figura 21. Linhas do arquivo que será gerado o grafo

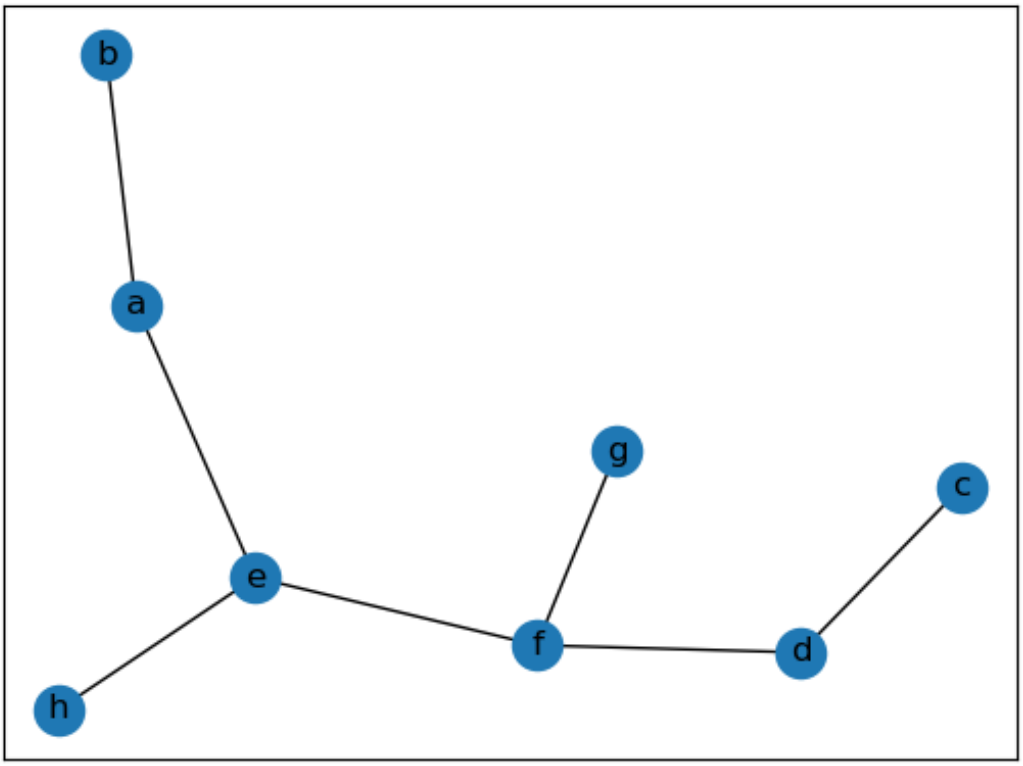

Figura 22. Imagem do grafo gerado a partir do arquivo exemplificado na Figura 21

Referências