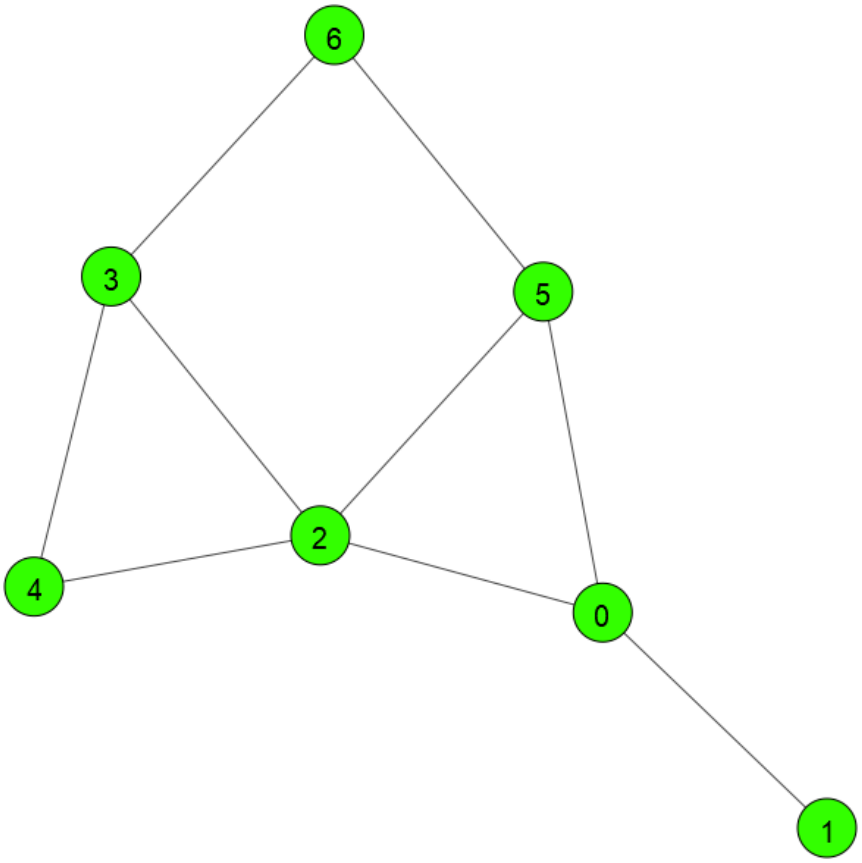
Alex Silva Rodrigues – 2021001925

Relatório da atividade 03

Link para o repositório com o código no Github: https://github.com/azl6/UNIFEI-Grafos-ATV3

Nesta atividade, foi requisitado que implementemos uma gama diversa de funções, que operam sobre os mais diferentes tipos de grafos. Para exemplificar o uso dos métodos implementados, usarei a matriz de adjacência (e seu respectivo grafo gerado) do arquivo **exemplo.txt**, enviado pelo professor.

O referido grafo é estruturado da seguinte maneira:



Neste trabalho, abordarei:

- A implementação das funções requisitadas
- A utilização das funções requisitadas e seus respectivos resultados
- Explicação da lógica das funções requisitadas
- Resultados gerados no terminal
- Resultados gerados no arquivo
- Dificuldades

O primeiro método implementado, **tipoGrafo(matriz)**, visa a identificação do tipo do grafo, a partir de sua matriz de adjacência. Sua implementação foi feita da seguinte maneira:

```
def tipoGrafo(matriz):

    diagonalEhZerada = True
    contemParalelas = False
    ehSimetrica = True

    qtdVertices = np.shape(matriz)[0]

    for vi in range(0, qtdVertices):
        for vj in range(vi + 1, qtdVertices):

            if vi == vj:
                if matriz[vi][vj] != 0:
                    diagonalEhZerada = False

            if matriz[vi][vj] == 2:
                contemParalelas = True

            if matriz[vi][vj] != matriz[vj][vi]:
                ehSimetrica = False

    if diagonalEhZerada and not contemParalelas and ehSimetrica: #SIMPLES
        return 0
    if diagonalEhZerada and not contemParalelas and not ehSimetrica: #DIGRAFO
        return 1
    if diagonalEhZerada and contemParalelas: #MULTIGRAFO
        return 2
    if not diagonalEhZerada and contemParalelas: #PSEUDOGRAFO
      return 3
```

Aqui, defini algumas variáveis do tipo **boolean**, sendo elas a **diagonalEhZerada**, **contemParalelas** e a **ehSimetrica**. Tais variáveis auxiliam na identificação do tipo do grafo. O grafo simples, por exemplo, tem como requisito:

- **Não ter laços** (diagonalEhZerada = True)
- **Não ter arestas** paralelas (contemParalelas = False)
- **Ser simétrica** (ehSimetrica = True)

Com tais variáveis definidas, podemos alternar as combinações para identificarmos os diferentes tipos de grafos requisitados.

Abaixo, segue também a tabela que me auxiliou na definição das condições de diferenciação dos grafos:

Resumo das Características dos Tipos de Grafos

Tipo	Aresta	Arestas múltiplas	Laços
Grafo simples	Não dirigida	Não	Não
Multigrafo	Não dirigida	Sim	Não
Pseudografo	Não dirigida	Sim	Sim
Grafo dirigido	Dirigida	Não	Sim
Multigrafo dirigido	Dirigida	Sim	Sim

O segundo método implementado, **calcDensidade(self, matriz)**, calcula a densidade de grafos dos tipos **simples** ou **digrafos**.

Abaixo, segue a sua implementação:

```
def calcDensidade(self, matriz):

    tipoGrafo = self.tipoGrafo(matriz)
    qtdVertices = np.shape(matriz)[0]

    arestas=0
    for vi in range(0, qtdVertices):
        for vj in range(vi + 1, qtdVertices):
            if matriz[vi][vj] == 1:
                arestas+=1

    if tipoGrafo == 0:
        return (2 * arestas) / (qtdVertices * (qtdVertices - 1))

    if tipoGrafo == 1:
        return arestas / (qtdVertices * (qtdVertices - 1))
```

Para realizamos esse cálculo, precisamos das seguintes variáveis: quantidade de vértices e quantidade de arestas do grafo. A partir do método **np.shape(matriz)[0]**, encontramos a quantidade de vértices. Já para a quantidade de arestas, devemos percorrer o grafo e contar todas as posições cujo valor seja 1 (ou seja, onde entre dois vértices Vi e Vj haja uma aresta). Isso permite que calculemos a densidade de grafos simples e digrafos, onde tal diferenciação é feita pelos últimos dois condicionais, aplicando um cálculo diferente, dependendo do resultado da função **tipoGrafo(matriz)**, implementada anteriormente.

Abaixo, seguem as formulas de cálculo de densidade de um grago, apresentadas pelo professor, que utilizei como base para elaboração dessa função, e que motivam a minha necessidade de calcular a quantidade de vértices e arestas do grafo em questão:

Cálculo da Densidade (D)

Grafo Simples

$$\mathbf{D} = \frac{2E}{V(V-1)}$$

Digrafo

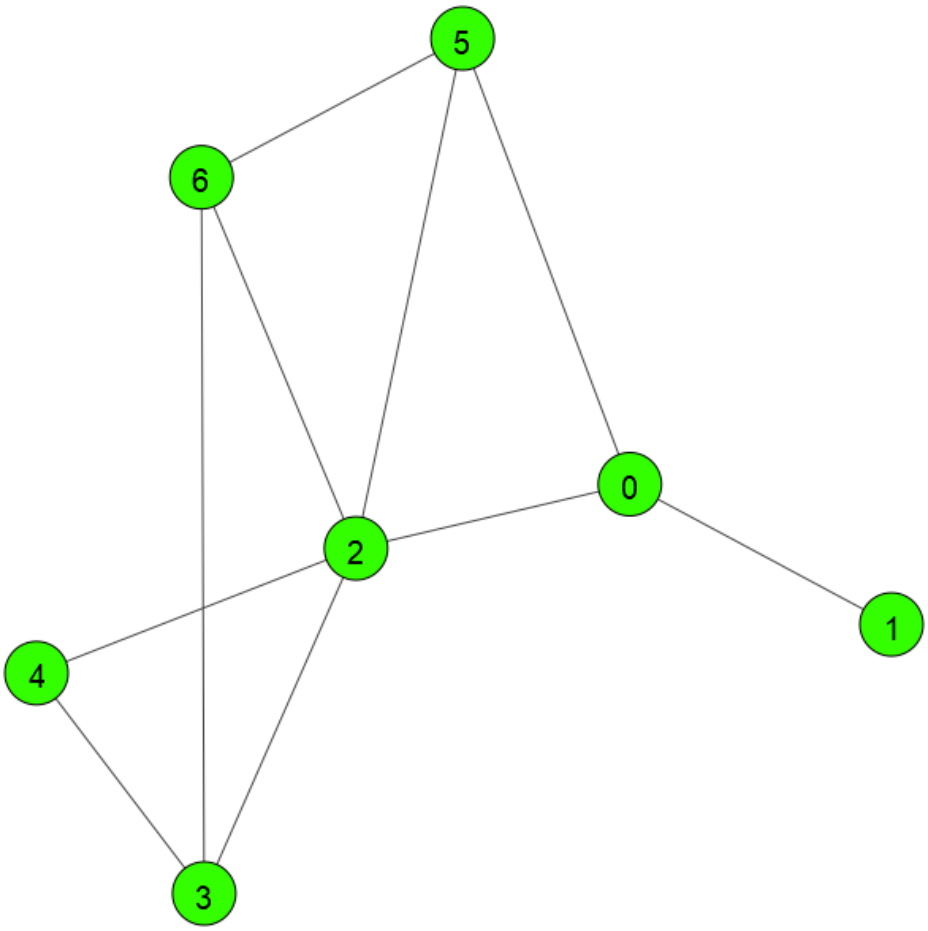
$$\mathbf{D} = \frac{E}{V(V-1)}$$

Já o terceiro método, **insereAresta(matriz, vi, vj)**, visa inserir uma aresta que conecte os dois vértices **vi** e **vj** informados. Sua implementação foi feita da seguinte maneira:

```
def insereAresta(self, matriz, vi, vj):
    tipoGrafo = self.tipoGrafo(matriz)
    if tipoGrafo == 1:
        matriz[vi][vj] = 1
    else:
        matriz[vi][vj] = 1
        matriz[vj][vi] = 1

    return matriz
```

Primeiro, realizei a identificação do tipo do grafo, a fim de manter íntegra a propriedade de **simetria** e **assimetria.** Por exemplo, caso a função **tipoGrafo(matriz)** retorne 1, significa que estamos lidando com um grafo direcionado, que, por natureza, é assimétrico. Por outro lado, qualquer outro retorno significaria um grafo não-direcionado, e por isso, faz-se necessário que mantenhamos a simetria de sua matriz de adjacência. A fim de exemplificar o método, segue abaixo o resultado de sua aplicação, inserindo uma aresta entre os vértices 2 e 6 do grafo **exemplo.txt**



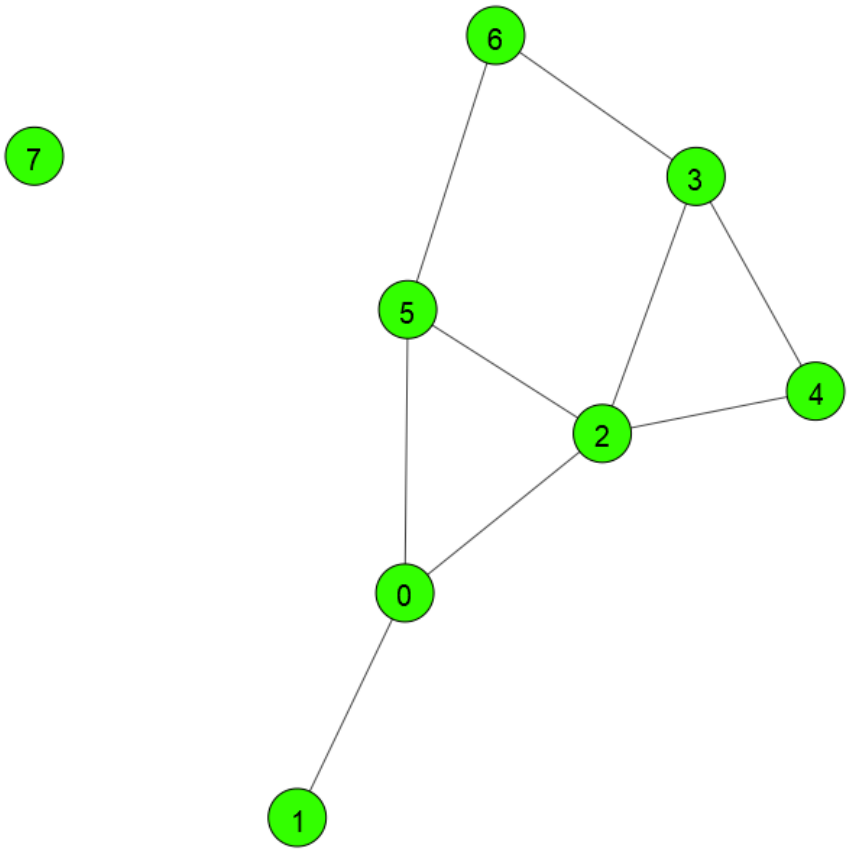
O quarto método implementado visava a inserção de um novo vértice no grafo. Ele foi implementado da seguinte maneira:

```
def insereVertice(matriz, vi):
    shape = matriz.shape
    novaMatriz = numpy.zeros((shape[0] + 1, shape[1] + 1))

    qtdVertices = np.shape(matriz)[0]
    for vi in range(0, qtdVertices):
        for vj in range(0, qtdVertices):
            novaMatriz[vi][vj] = matriz[vi][vj]

    return novaMatriz
```

O método cria uma de **[QUANTIDADE_VÉRTICES_ANTIGOS + 1] x [QUANTIDADE_VÉRTICES_ANTIGOS + 1]** a partir da função **numpy.zeros(...)**. A partir daí, iteramos pela matriz antiga, e transferimos os valores para a matriz nova. O resultado da utilização deste método, no grafo **exemplo.txt** é o seguinte:



Observe que o novo vértice é criado de forma isolada dos demais, já que ainda não existe nenhuma aresta o conectando a um outro vértice.

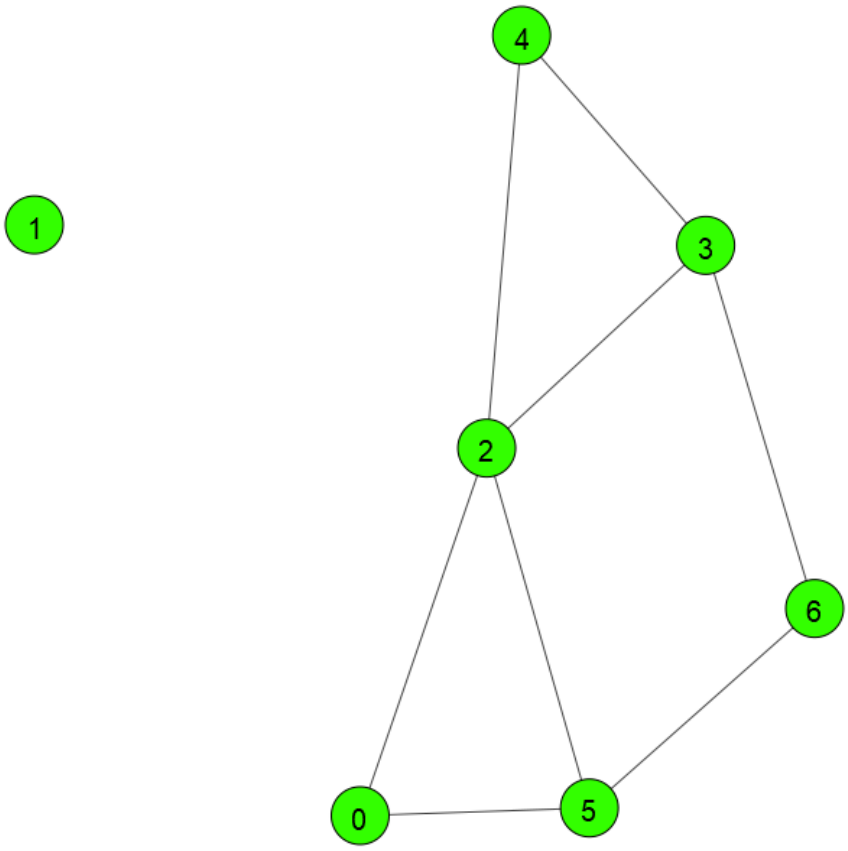
Por fim, o último método implementado, **removeAresta()**, se comporta de maneira similar ao **insereAresta()**. Seu princípio consiste em setar o valor da posição [vi][vj] da matriz para 0, respeitando os princípio de simetria e assimetria do grafo em questão. Sua implementação foi feita da seguinte maneira:

```
def removeAresta(self, matriz, vi, vj):
    tipoGrafo = self.tipoGrafo(matriz)

    if tipoGrafo == 1:
        matriz[vi][vj] = 0
    else:
        matriz[vi][vj] = 0
        matriz[vj][vi] = 0

    return matriz
```

A fim de exemplificar o seu uso, segue abaixo o resultado de sua utilização no grafo **exemplo.txt,** removendo a conexão entre os vértices 0 e 1:



Segue, abaixo, uma prova da realização das operações acima mencionadas. Todas as funções foram aplicadas no grafo **exemplo.txt** base, ou seja, a inserção e remoção de arestas e vértices não é sequencial, mas poderia ser, se assim desejássemos, com uma simples alteração na main().

```
NOME DA INSTÂNCIA: exemplo

[[0 1 1 0 0 1 0]
 [1 0 0 0 0 0 0]
 [1 0 0 1 1 1 0]
 [0 0 1 0 1 0 1]
 [0 0 1 1 0 0 0]
 [1 0 1 0 0 0 1]
 [0 0 0 1 0 1 0]]

IGRAPH U--- 7 9 --
+ attr: label (v)
+ edges:
0 -- 1 2 5     2 -- 0 3 4 5   4 -- 2 3       6 -- 3 5
1 -- 0         3 -- 2 4 6     5 -- 0 2 6

Vertices 0 e 1 são adjacentes? True

Tipo do grafo: 0
Densidade do grafo: 0.42857142857142855
Inserindo aresta ...
Inserindo vértice ...
Removendo aresta ...
```

Resultados gerados no arquivo:

```
exemplo True
exemplo 0
exemplo 0.42857142857142855
exemplo [[0 1 1 0 0 1 0]
 [1 0 0 0 0 0 0]
 [1 0 0 1 1 1 1]
 [0 0 1 0 1 0 1]
 [0 0 1 1 0 0 0]
 [1 0 1 0 0 0 1]
 [0 0 1 1 0 1 0]]
exemplo [[0. 1. 1. 0. 0. 1. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 1. 1. 1. 1. 0.]
 [0. 0. 1. 0. 1. 0. 1. 0.]
 [0. 0. 1. 1. 0. 0. 0. 0.]
 [1. 0. 1. 0. 0. 0. 1. 0.]
 [0. 0. 1. 1. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]]
exemplo [[0 0 1 0 0 1 0]
 [0 0 0 0 0 0 0]
 [1 0 0 1 1 1 1]
 [0 0 1 0 1 0 1]
 [0 0 1 1 0 0 0]
 [1 0 1 0 0 0 1]
 [0 0 1 1 0 1 0]]
```

Dificuldades: Não consegui implementar a remoção de vértices. Foi um pouco mais complicado que as demais funções, já que dependendo do vértice, deveríamos puxar todos os valores da direita para a esquerda, e de baixo para cima. Além disso, busquei uma função das bibliotecas abordadas para fazer isso, mas não consegui encontrá-la.