Teoria dos Grafos - Exercícios de Implementação

Integrantes do Grupo:

- Igor Benites Moura - 10403462
- Rodrigo Machado de Assis Oliveira de Lima - 10401873

Implementado em Python

Exercício 1:

Código fonte:

```python
def inDegree(self, v):

        grau = 0

        for i in range(self.n):

            # para cada vértice i verifica se é adjacente a v

            if self.adj[i][v] == 1:

                grau+=1

        return grau
```

Testes:

```python
g = Grafo(4)
#insere as arestas do grafo
#A={(0,1),(0,2),(2,1),(2,3),(1,3)}
g.insereA(0,1)
g.insereA(0,2)
g.insereA(2,1)
g.insereA(2,3)
g.insereA(1,3)
# mostra o grafo preenchido
g.show()
g.showMin()

# grau de entrada do vertice 3:
print(f"grau de entrada do vertice 3: {g.inDegree(3)}")
```

```
grau de entrada do vertice 3: 2
```

Exercício 2:

Código fonte:

```
def outDegree(self, v):
        grau = 0
        for i in range(self.n):
            # para cada vértice verifica se é adjacente a i
            if self.adj[v][i] == 1:
                grau+=1
        return grau
```

Testes:

```
g = Grafo(4)
#insere as arestas do grafo
#A={(0,1),(0,2),(2,1),(2,3),(1,3)}
g.insereA(0,1)
g.insereA(0,2)
g.insereA(2,1)
g.insereA(2,3)
g.insereA(1,3)
# mostra o grafo preenchido
g.show()
g.showMin()

# grau de saida do vertice 0:
print(f"grau de saida do vertice 0: {g.outDegree(0)}")
```

```
grau de saida do vertice 0: 2
```

Exercício 3:

Código fonte:

```
def isVSource(self, v):
        # usa as funções inDegree e outDegree para verificar se é fonte
        return (self.inDegree(v) == 0 and self.outDegree(v) > 0)
```

Testes:

```
g = Grafo(4)
#insere as arestas do grafo
#A={(0,1),(0,2),(2,1),(2,3),(1,3)}
g.insereA(0,1)
g.insereA(0,2)
g.insereA(2,1)
g.insereA(2,3)
g.insereA(1,3)
# mostra o grafo preenchido
g.show()
g.showMin()

# vertice 0 é fonte?
print(f"vertice 0 é fonte? {g.isVSource(0)}")
```

```
vertice 0 é fonte? True
```

Exercício 4:

Código fonte:

```
def isVSink(self, v):

        # usa as funções inDegree e outDegree para verificar se é
sorvedouro

        return (self.inDegree(v) > 0 and self.outDegree(v) == 0)
```

Testes:

```
g = Grafo(4)
#insere as arestas do grafo
#A={(0,1),(0,2),(2,1),(2,3),(1,3)}
g.insereA(0,1)
g.insereA(0,2)
g.insereA(2,1)
g.insereA(2,3)
g.insereA(1,3)
# mostra o grafo preenchido
g.show()
g.showMin()

# vertice 3 é sorvedouro?
print(f"vertice 3 é sorvedouro? {g.isVSink(3)}")
```

```
vertice 3 é sorvedouro? True
```

Exercício 5:

Código fonte:

```
def isSymmetric(self):

        for i in range(self.n):

            for j in range(self.n):

                # se algum elemento da matriz for diferente do elemento simétrico, a matriz não é simétrica

                if self.adj[i][j] != self.adj[j][i]:

                    return False

        return True
```

Testes:

```
g = Grafo(4)
#insere as arestas do grafo
#A={(0,1),(0,2),(2,1),(2,3),(1,3)}
g.insereA(0,1)
g.insereA(0,2)
g.insereA(2,0)
g.insereA(1,0)
# mostra o grafo preenchido
g.show()
g.showMin()

print(f"Grafo é simétrico? {g.isSymmetric()}")
```

```
Grafo é simétrico? True
```

Exercício 6:

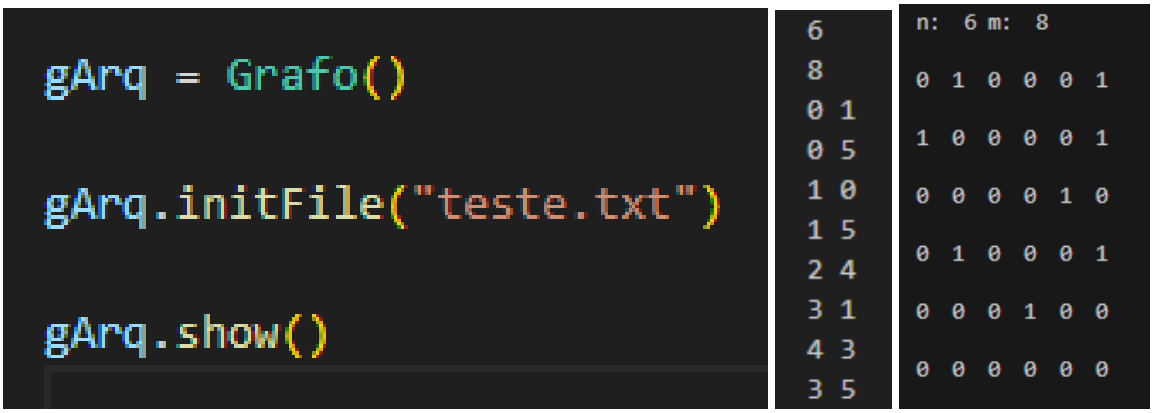
Código fonte:

```
def initFile(self, nomeArq):

        script_dir = os.path.dirname(__file__)

        file_path = os.path.join(script_dir, nomeArq)

        with open(file_path, "r") as arq:

            self.n = int(arq.readline())

            m = int(arq.readline())
```

```
            print(self.m)
            for _ in range(m):
                v, w = map(int, arq.readline().split())
                self.insereA(v, w)
        arq.close()
        print("m:" + str(self.m))
```

Testes:

(Código, arquivo texto e resultado no console da esquerda pra direita):



Exercício 7 e 8:

Código fonte:

```
class GrafoND:
    TAM_MAX_DEFAULT = 100 # qtde de vértices máxima default
    # construtor da classe grafo
    def __init__(self,  n=TAM_MAX_DEFAULT, isWeighted=False,):
        self.n = n # número de vértices
        self.m = 0 # número de arestas
        self.isWeighted = isWeighted
        # matriz de adjacência
        if isWeighted:
            self.adj = [[float('inf') for i in range(n)] for j in
range(n)]
```

```
        else:
            self.adj = [[0 for i in range(n)] for j in range(n)]


    # Insere uma aresta no Grafo tal que
    # v é adjacente a w com peso weight
    def insereA(self, v, w, weight=1):
        #testa de temos a aresta
        if self.isWeighted and self.adj[v][w] == float('inf'):
            self.adj[v][w] = weight
            self.adj[w][v] = weight
        elif not self.isWeighted and self.adj[v][w] == 0:
            self.adj[v][w] = weight
            self.adj[w][v] = weight
        self.m+=1 # atualiza qtd arestas


    # remove uma aresta v->w do Grafo
    def removeA(self, v, w):
        # testa se temos a aresta
        if self.adj[v][w] != 0 and self.adj[v][w] != float('inf'):
            if self.isWeighted:
                self.adj[v][w] = float('inf')
                self.adj[w][v] = float('inf')
            else:
                self.adj[v][w] = 0
                self.adj[w][v] = 0
            self.m-=1 # atualiza qtd arestas
```

Testes grafoND sem peso:

```
# Grafo nao direcionado sem peso

nd = GrafoND(4)
nd.insereA(0,1)
nd.insereA(0,2)
nd.insereA(2,1)
nd.insereA(3,1)

nd.showMin()
```

```
n:  4 m:  4

0  1  1  0

1  0  1  1

1  1  0  0

0  1  0  0
```

Testes grafoND com peso:

```
# Grafo nao direcionado com peso
nd = GrafoND(4, isWeighted=True)

nd.insereA(0,1,20) # insere aresta 0->1 com peso 20
nd.insereA(0,2,30)
nd.insereA(2,1,21)
nd.insereA(1,3,22)
nd.removeA(0,1)

nd.showMin()
```

```
n:  4 m:  3

∞   ∞   30  ∞

∞   ∞   21  22

30  21  ∞   ∞

∞   22  ∞   ∞
```

Exercício 9:

Código fonte direcionado:

```
def removeV(self, v):

        # remove todas as arestas que tem v como origem ou destino

        for i in range(self.n):

            self.removeA(v, i)

            self.removeA(i, v)



        # cria uma nova matriz de adjacência com um vértice a menos

        tempAdj = [[0 for _ in range(self.n - 1)] for _ in range(self.n
- 1)]

        i2 = 0

        j2 = 0



        # copia os valores da matriz antiga para a nova matriz,
excluindo a linha e coluna do vértice removido

        for i in range(self.n):

            j2 = 0

            if i == v:

                continue

            for j in range(self.n):

                if j == v:

                    continue

                tempAdj[i2][j2] = self.adj[i][j]

                j2+=1

            i2+=1

        self.n-=1

        self.adj = tempAdj
```

Código fonte não direcionado:

```
 #Ex9

    def removeV(self, v):

        for i in range(self.n):

            self.removeA(v, i)



        if self.isWeighted:

            tempAdj = [[float('inf') for i in range(self.n - 1)] for j
in range(self.n - 1)]

        else:

            tempAdj = [[0 for i in range(self.n - 1)] for j in
range(self.n - 1)]

        i2 = 0

        j2 = 0

        for i in range(self.n):

            j2 = 0

            if i == v:

                continue

            for j in range(self.n):

                if j == v:

                    continue

                tempAdj[i2][j2] = self.adj[i][j]

                j2+=1

            i2+=1

        self.n-=1

        self.adj = tempAdj
```

Testes:

Direcionado:

```
g = Grafo(4)
#insere as arestas do grafo
g.insereA(0,1)
g.insereA(0,2)
g.insereA(2,3)
g.insereA(1,0)
g.insereA(1,2)

# remove vertice 0
g.removeV(0)
g.showMin()
```

```
n:  3 m:  2

0  1  0

0  0  1

0  0  0
```

Não direcionado:

```
nd = GrafoND(4)
nd.insereA(0,1)
nd.insereA(0,2)
nd.insereA(2,1)
nd.insereA(3,1)

nd.removeV(0)

nd.showMin()
```

```
n:  3 m:  2

0  1  1

1  0  0

1  0  0
```

Exercício 10:

Código fonte:

```
def isComplete(self):

        for i in range(self.n):

            for j in range(self.n):

                if i != j:

                    if self.isWeighted and self.adj[i][j] ==
float('inf'):

                        return False

                    elif self.isWeighted and self.adj[i][j] == 0:

                        return False

        return True
```

Testes:

```
nd = GrafoND(3)
nd.insereA(0,1)
nd.insereA(0,2)
nd.insereA(1,2)

nd.showMin()
print(f"Grafo é completo? {nd.isComplete()}")
```

```
 n:  3 m:  3

 0  1  1

 1  0  1

 1  1  0


fim da impressao do grafo.
Grafo é completo? True
```

Exercício 11:

Código fonte:

```
def isComplete(self):

        for i in range(self.n):

            for j in range(self.n):

                # se algum vertice nao for adjacente a todos os outros,
o grafo nao é completo

                if i != j and self.adj[i][j] == 0:

                    return False

        return True
```

Testes:

```
g = Grafo(3)
#insere as arestas do grafo
g.insereA(0,1)
g.insereA(0,2)
g.insereA(1,0)
g.insereA(1,2)
g.insereA(2,0)
g.insereA(2,1)

g.showMin()
print(f"Grafo é completo? {g.isComplete()}")
```

```
 n:  3 m:  6

 0  1  1

 1  0  1

 1  1  0


fim da impressao do grafo.
Grafo é completo? True
```

Exercício 12:

Código fonte:

```
def complementary(self):

        tempAdj = [[0 for _ in range(self.n)] for _ in range(self.n)]

        for i in range(self.n):

            for j in range(self.n):

                if i != j:  # Pula diagonal

                    # inverte os valores da matriz de adjacência,
tirando a diagonal principal

                    if self.adj[i][j] == 0:

                        tempAdj[i][j] = 1

                    else:

                        tempAdj[i][j] = 0

        complementary = Grafo(self.n)

        complementary.adj = tempAdj

        complementary.m = self.m

        return complementary
```

Testes:

```
g = Grafo(3)
#insere as arestas do grafo
g.insereA(0,1)
g.insereA(0,2)
g.insereA(1,0)
g.showMin()

g2 = g.complementary()
print("Grafo complementar")
g2.showMin()
```

```
 n:  3 m:  3

 0  1  1

 1  0  0

 0  0  0


fim da impressao do grafo.
Grafo complementar

 n:  3 m:  3

 0  0  0

 0  0  1

 1  1  0
```

Exercício 13:

Código fonte:

```
    # Checa se um vértice v está conectado a um vértice w

    def isVConnected(self, v, w):

        visited = [False] * self.n



        def dfs(current):

            if current == w:

                return True

            visited[current] = True

            for neighbor in range(self.n):

                if not self.isWeighted and self.adj[current][neighbor]
!= 0 and not visited[neighbor]:

                    if dfs(neighbor):

                        return True
```

```
                if self.isWeighted and self.adj[current][neighbor] !=
float('inf') and not visited[neighbor]:
                    if dfs(neighbor):
                        return True
            return False

        return dfs(v)

    # Checa se o grafo é conexo
    def isConnected(self):
        for i in range(self.n):
            for j in range(i + 1, self.n):
                if not self.isVConnected(i, j):
                    return False
        return True
```

Testes:

```
nd = GrafoND(3)
nd.insereA(0,1)
nd.insereA(0,2)
nd.insereA(1,2)

nd.showMin()
print(f"Grafo é conexo? {nd.isConnected()}")
```

```
n:  3 m:  3
0  1  1
1  0  1
1  1  0

fim da impressao do grafo.
Grafo é conexo? True
```

```
nd = GrafoND(3)
nd.insereA(0,2)

nd.showMin()
print(f"Grafo é conexo? {nd.isConnected()}")
```

```
n:  3 m:  1
0  0  1
0  0  0
1  0  0

fim da impressao do grafo.
Grafo é conexo? False
```

Exercício 14:

Código fonte:

```
def directTransitiveClosure(self, v):

        # usa busca em largura para encontrar todos os vértices que são
alcançáveis a partir de v

        reach = [0] * self.n

        queue = deque([v])

        reach[v] = 1
```

```
        # enquanto a fila não estiver vazia, verifica os vértices
adjacentes
        while queue:
            current = queue.popleft()
            for j in range(self.n):
                if self.adj[current][j] == 1 and not reach[j]:
                    reach[j] = 1
                    queue.append(j)

        return reach

    def inverseTransitiveClosure(self, v):
        # usa busca em largura para encontrar todos os vértices que
alcançam v
        reach = [0] * self.n
        queue = deque([v])
        reach[v] = 1

        # enquanto a fila não estiver vazia, verifica os vértices
adjacentes
        while queue:
            current = queue.popleft()
            for j in range(self.n):
                if self.adj[j][current] == 1 and not reach[j]:
                    reach[j] = 1
                    queue.append(j)
```

```
        return reach


    def isStronglyConnected(self):
        # usa os metodos de fecho transitivo direto e inverso para verificar se o grafo é fortemente conexo
        for v in range(self.n):
            direct_reach = self.directTransitiveClosure(v)
            inverse_reach = self.inverseTransitiveClosure(v)


            # se nao for uma interseccao completa entre os fechos, o grafo nao é fortemente conexo
            if not all(direct_reach) or not all(inverse_reach):
                return False
        return True


    def isSemiStronglyConnected(self):
        # usa os metodos de fecho transitivo direto e inverso para verificar se o grafo é semi-fortemente conexo
        for v in range(self.n):
            direct_reach = self.directTransitiveClosure(v)
            inverse_reach = self.inverseTransitiveClosure(v)


            for i in range(self.n):
                # se um vértice não for alcançável por v ou não alcançar v, o grafo não é semi-fortemente conexo
                if not direct_reach[i] and not inverse_reach[i]:
```

```
                    return False

        return True


    def isDisconnected(self):

        # Cria o grafo não direcionado equivalente pois direcao em
grafos s-conexos não importa

        NDGraph = Grafo(self.n)

        NDGraph.adj = self.adj

        for i in range(self.n):

            for j in range(self.n):

                if self.adj[i][j] == 1:

                    NDGraph.insereA(j, i)


        # usa os metodos de fecho transitivo direto e inverso para
verificar se o grafo é desconexo

        for u in range(self.n):

            for v in range(self.n):

                if u != v:

                    direct_reach_u = NDGraph.directTransitiveClosure(u)

                    direct_reach_v = NDGraph.directTransitiveClosure(v)

                    # se para algum par de vértices u e v, u não
alcançar v e v não alcançar u, o grafo é desconexo

                    if not direct_reach_u[v] and not direct_reach_v[u]:

                        return True

        return False
```

```python
def connectivity(self):
    # retorna a conexidade do grafo: C3, C2, C1 ou C0
    if self.isStronglyConnected():
        return 3
    if self.isSemiStronglyConnected():
        return 2
    if self.isDisconnected():
        return 0
    return 1
```

Testes:

```python
g.insereA(0,1)
g.insereA(0,2)
g.insereA(1,0)
g.insereA(2,3)
g.insereA(3,0)
g.showMin()

print(f"Conexidade do grafo: {g.connectivity()}")
```

```
Conexidade do grafo: 3
```

```python
g.insereA(0,1)
g.insereA(0,2)
g.insereA(1,0)
g.insereA(2,3)
g.showMin()

print(f"Conexidade do grafo: {g.connectivity()}")
```

```
Conexidade do grafo: 2
```

```python
g.insereA(0,3)
g.insereA(3,1)
g.insereA(3,2)
g.showMin()

print(f"Conexidade do grafo: {g.connectivity()}")
```

```
Conexidade do grafo: 1
```

```
g.insereA(0,3)
g.insereA(3,1)
g.showMin()

print(f"Conexidade do grafo: {g.connectivity()}")
```

```
Conexidade do grafo: 0
```

Exercício 15:

Código fonte:

```
def getStronglyConnectedComponents(self):

        # inicia a lista de sccs e vetor de visitados

        visited = [False] * self.n

        sccs = []



        for v in range(self.n):

            # se o vértice não foi visitado, calcula o fecho transitivo
direto e inverso

            if not visited[v]:

                direct_reach = self.directTransitiveClosure(v)

                inverse_reach = self.inverseTransitiveClosure(v)

                # cria um um scc com a interseccao dos fechos

                scc = [i for i in range(self.n) if direct_reach[i] and
inverse_reach[i]]

                sccs.append(scc)

                # marca todos os vértices do scc como visitados

                for u in scc:

                    visited[u] = True



        return sccs
```

```
    def reduce(self):

        sccs = self.getStronglyConnectedComponents()

        scc_count = len(sccs)

        # cria um dicionário para mapear cada vértices ao seu
respectivo scc

        scc_map = {v: i for i, scc in enumerate(sccs) for v in scc}



        # cria uma matriz de adjacência reduzida com os sccs

        reduced_adj = [[0] * scc_count for _ in range(scc_count)]

        for u in range(self.n):

            for v in range(self.n):

                # se u e v são adjacentes, verifica se os sccs de u e v
são diferentes

                if self.adj[u][v] == 1:

                    u_scc = scc_map[u]

                    v_scc = scc_map[v]

                    # se os sccs são diferentes, adiciona uma aresta
entre os sccs

                    if u_scc != v_scc:

                        reduced_adj[u_scc][v_scc] = 1



        reducedGraph = Grafo(scc_count)

        reducedGraph.adj = reduced_adj

        return reducedGraph
```

Testes:

```
g = Grafo(5)
#insere as arestas do grafo
g.insereA(0,1)
g.insereA(1,2)
g.insereA(2,0)
g.insereA(2,3)
g.insereA(3,4)
g.insereA(4,3)
g.showMin()

gr = g.reduce()
print("Grafo reduzido:")
gr.showMin()
```

```
 n:  5 m:  6

 0  1  0  0  0

 0  0  1  0  0

 1  0  0  1  0

 0  0  0  0  1

 0  0  0  1  0


fim da impressao do grafo.
Grafo reduzido:

 n:  2 m:  0

 0  1

 0  0
```

Exercício 16:

Código fonte:

```
 def __init__(self, n=TAM_MAX_DEFAULT, isWeighted=False):

        self.n = n # número de vértices

        self.m = 0 # número de arestas

        self.isWeighted = isWeighted

        # matriz de adjacência

        if isWeighted:

            self.adj = [[float('inf') for i in range(n)] for j in
range(n)]

        else:

            self.adj = [[0 for i in range(n)] for j in range(n)]



    # Insere uma aresta no Grafo tal que

    # v é adjacente a w

    def insereA(self, v, w, weight=1):

        if self.isWeighted:

            if self.adj[v][w] == float('inf'):

                self.adj[v][w] = weight

                self.m+=1 # atualiza qtd arestas

        else:

            if self.adj[v][w] == 0:

                self.adj[v][w] = 1

                self.m+=1
```

```
    # remove uma aresta v->w do Grafo
    def removeA(self, v, w):
        if self.isWeighted:
            if self.adj[v][w] != float('inf'):
                self.adj[v][w] = float('inf')
                self.m-=1
        else:
            if self.adj[v][w] == 1:
                self.adj[v][w] = 0
                self.m-=1 # atualiza qtd arestas
. . . . . .
      # Apresenta o Grafo contendo
    # número de vértices, arestas
    # e a matriz de adjacência obtida
    def show(self):
        print(f"\n n: {self.n:2d} ", end="")
        print(f"m: {self.m:2d}\n")
        for i in range(self.n):
            for w in range(self.n):
                if self.isWeighted:
                    if self.adj[i][w] != float('inf'):
                        print(f"Adj[{i:2d},{w:2d}] =
{self.adj[i][w]:2d} ", end="")
                    else:
                        print(f"Adj[{i:2d},{w:2d}] = inf ", end="")
```

```
                else:
                    if self.adj[i][w] == 1:
                        print(f"Adj[{i:2d},{w:2d}] = 1 ", end="")
                    else:
                        print(f"Adj[{i:2d},{w:2d}] = 0 ", end="")
            print("\n")
        print("\nfim da impressao do grafo." )



    # Apresenta o Grafo contendo
    # número de vértices, arestas
    # e a matriz de adjacência obtida
    # Apresentando apenas os valores 0 ou 1
    def showMin(self):
        print(f"\n n: {self.n:2d} ", end="")
        print(f"m: {self.m:2d}\n")
        for i in range(self.n):
            for w in range(self.n):
                if self.isWeighted:
                    if self.adj[i][w] != float('inf'):
                        print(f" {self.adj[i][w]:2d} ", end="")
                    else:
                        print(" ∞  ", end="")
                else:
                    if self.adj[i][w] == 1:
```

```
                    print(" 1 ", end="")

                else:

                    print(" 0 ", end="")

            print("\n")

        print("\nfim da impressao do grafo." )



                    print(" 1 ", end="")

                else:

                    print(" 0 ", end="")

            print("\n")

        print("\nfim da impressao do grafo." )
```

Testes:

```
gw = Grafo(5, isWeighted=True)

gw.insereA(0,1,20) # insere aresta 0->1 com peso 20
gw.insereA(0,2,30)
gw.insereA(2,1,21)
gw.insereA(1,3,22)
gw.insereA(3,4,50)

gw.showMin()

gw.removeA(0,1)

gw.show()
```

```
 n:  5 m:  5

 ∞   20  30  ∞   ∞

 ∞   ∞   ∞   22  ∞

 ∞   21  ∞   ∞   ∞

 ∞   ∞   ∞   ∞   50

 ∞   ∞   ∞   ∞   ∞


fim da impressao do grafo.

 n:  5 m:  4

Adj[ 0, 0] = inf Adj[ 0, 1] = inf Adj[ 0, 2] = 30 Adj[ 0, 3] = inf Adj[ 0, 4] = inf

Adj[ 1, 0] = inf Adj[ 1, 1] = inf Adj[ 1, 2] = inf Adj[ 1, 3] = 22 Adj[ 1, 4] = inf

Adj[ 2, 0] = inf Adj[ 2, 1] = 21 Adj[ 2, 2] = inf Adj[ 2, 3] = inf Adj[ 2, 4] = inf

Adj[ 3, 0] = inf Adj[ 3, 1] = inf Adj[ 3, 2] = inf Adj[ 3, 3] = inf Adj[ 3, 4] = 50

Adj[ 4, 0] = inf Adj[ 4, 1] = inf Adj[ 4, 2] = inf Adj[ 4, 3] = inf Adj[ 4, 4] = inf
```

Exercício 17:

Código fonte:

```
def isEqual(self, g):

        if self.n != g.n or self.m != g.m:

            return False

        for i in range(self.n):

            if self.listaAdj[i] != g.listaAdj[i]:

                return False

        return True
```

Testes:

```
g = Grafo(4)
#insere as arestas do grafo
#A={(0,1),(0,2),(2,1),(2,3),(1,3)}
g.insereA(0,1)
g.insereA(0,2)
g.insereA(2,1)
g.insereA(2,3)
g.insereA(1,3)
# mostra o grafo preenchido
g.show()
g.removeA(0,1)
g.show()

g2 = Grafo(4)
g2.insereA(0,2)
g2.insereA(2,1)
g2.insereA(2,3)
g2.insereA(1,3)

g2.show()
print(g.isEqual(g2))
```

```
 n:  4 m:  4

 0:  2
 1:  3
 2:  1 3
 3:

fim da impressao do grafo.

 n:  4 m:  4

 0:  2
 1:  3
 2:  1 3
 3:

fim da impressao do grafo.
Grafos são iguais? True
```

Exercício 18:

Código fonte:

```
def convertToMatrix(self):

        g = Grafo(self.n)

        g.m = self.m

        g.listaAdj = [[0 for i in range(self.n)] for j in
range(self.n)]

        for i in range(self.n):

            for j in self.listaAdj[i]:

                g.listaAdj[i][j] = 1

        return g
```

Testes:

```
g = Grafo(4)
#insere as arestas do grafo
#A={(0,1),(0,2),(2,1),(2,3),(1,3)}
g.insereA(0,1)
g.insereA(0,2)
g.insereA(2,1)
g.insereA(2,3)
g.insereA(1,3)
# mostra o grafo preenchido
g.show()
g.removeA(0,1)
g.show()
print(g.convertToMatrix().listaAdj)
```

```
[[0, 0, 1, 0], [0, 0, 0, 1], [0, 1, 0, 1], [0, 0, 0, 0]]
```

Exercício 19:

Código fonte:

```
def invert(self):

        g = Grafo(self.n)

        g.m = self.m

        for i in range(self.n):

            for j in self.listaAdj[i][::-1]:

                g.listaAdj[i].append(j)

        return g
```

Testes:

```
g = Grafo(4)
#insere as arestas do grafo
#A={(0,1),(0,2),(2,1),(2,3),(1,3)}
g.insereA(0,1)
g.insereA(0,2)
g.insereA(2,1)
g.insereA(2,3)
g.insereA(1,3)
# mostra o grafo preenchido
g.show()
g.removeA(0,1)
g.show()

g.invert().show()
```

```
 n:  4 m:  4

 0:  2
 1:  3
 2:  1 3
 3:

fim da impressao do grafo.
Invertido:

 n:  4 m:  4

 0:  2
 1:  3
 2:  3 1
 3:

fim da impressao do grafo.
```

Exercício 20:

Código fonte:

```
def isVSource(self, v):

        if len(self.listaAdj[v]) == 0:

            return 0



        for i in range(self.n):

            if v in self.listaAdj[i]:

                return 0



        return 1
```

Testes:

```
g = Grafo(4)
#insere as arestas do grafo
#A={(0,1),(0,2),(2,1),(2,3),(1,3)}
g.insereA(0,1)
g.insereA(0,2)
g.insereA(2,1)
g.insereA(2,3)
g.insereA(1,3)
# mostra o grafo preenchido
g.show()
g.removeA(0,1)
g.show()

print(f"Vertice 0 é fonte? {g.isVSource(0)}")
```

```
Vertice 0 é fonte? 1
```

Exercício 21:

Código fonte:

```
def isVSink(self, v):

        if len(self.listaAdj[v]) > 0:

            return 0



        for i in range(self.n):

            if v in self.listaAdj[i]:

                return 1



        return 0
```

Testes:

```
g = Grafo(4)
#insere as arestas do grafo
#A={(0,1),(0,2),(2,1),(2,3),(1,3)}
g.insereA(0,1)
g.insereA(0,2)
g.insereA(2,1)
g.insereA(2,3)
g.insereA(1,3)
# mostra o grafo preenchido
g.show()
g.removeA(0,1)
g.show()

print(f"Vertice 3 é sorvedouro? {g.isVSink(3)}")
```

```
Vertice 3 é sorvedouro? 1
```

Exercício 22:

Código fonte:

```
def isSymmetric(self):

        for i in range(self.n):

            for j in self.listaAdj[i]:

                if i not in self.listaAdj[j]:

                    return 0

        return 1
```

Testes:

```
g = Grafo(4)
#insere as arestas do grafo
#A={(0,1),(0,2),(2,1),(2,3),(1,3)}
g.insereA(0,1)
g.insereA(1,0)
g.insereA(2,1)
g.insereA(1,2)
g.insereA(2,3)
g.insereA(3,2)

# mostra o grafo preenchido
g.show()

print(f"Grafo é simétrico? {g.isSymmetric()}")
```

```
Grafo é simétrico? 1
```

Exercício 23:

Código fonte:

```
def initFile(self, nomeArq):

        script_dir = os.path.dirname(__file__)

        file_path = os.path.join(script_dir, nomeArq)

        with open(file_path, "r") as arq:

            self.n = int(arq.readline())

            m = int(arq.readline())

            for _ in range(m):

                v, w = map(int, arq.readline().split())

                self.insereA(v, w)

        arq.close()
```

Testes:

(Código, arquivo texto e resultado no console da esquerda pra direita):

```
print("Grafo do arquivo inputLista.txt:")
gArq = Grafo()
gArq.initFile("inputLista.txt")
gArq.show()
```

```
6
8
0 1
0 5
1 0
1 5
2 4
3 1
4 3
3 5
```

```
n:  6 m:  8

0:  1 5
1:  0 5
2:  4
3:  1 5
4:  3
5:
```

Exercício 24 e 25 (Funciona para ambos):

Código fonte:

```
def removeV(self, v):

        self.m -= len(self.listaAdj[v])

        self.listaAdj[v] = []



        for i in range(self.n):

            if v in self.listaAdj[i]:

                self.listaAdj[i].remove(v)

                self.m -= 1



        for i in range(self.n):

            self.listaAdj[i] = [x - 1 for x in self.listaAdj[i]]

            if i > v:

                self.listaAdj[i-1] = self.listaAdj[i]



        self.n -= 1
```

Testes:

```
g = Grafo(4)
#insere as arestas do grafo
#A={(0,1),(0,2),(2,1),(2,3),(1,3)}
g.insereA(0,1)
g.insereA(0,2)
g.insereA(1,2)
g.insereA(2,1)
g.insereA(2,3)
g.insereA(3,1)
g.insereA(3,2)

# mostra o grafo preenchido
g.show()

print("Vértice 1 removido:")
g.removeV(1)
```

```
 n:  4 m:  7

 0:  1 2
 1:  2
 2:  1 3
 3:  1 2

fim da impressao do grafo.
Vértice 1 removido:

 n:  3 m:  3

 0:  1
 1:  2
 2:  1
```

Exercício 26:

Código fonte:

```
def isComplete(self):

        for i in range(self.n):

            if len(self.listaAdj[i]) != self.n - 1:

                return False



        return True
```

Testes:

```
g = Grafo(4)
#insere as arestas do grafo
#A={(0,1),(0,2),(2,1),(2,3),(1,3)}
g.insereA(0,1)
g.insereA(0,2)
g.insereA(0,3)
g.insereA(1,0)
g.insereA(1,2)
g.insereA(1,3)
g.insereA(2,0)
g.insereA(2,1)
g.insereA(2,3)
g.insereA(3,0)
g.insereA(3,1)
g.insereA(3,2)

# mostra o grafo preenchido
g.show()

print(f"Grafo é completo? {g.isComplete()}")
```

```
  n:  4 m: 12

  0:  1 2 3
  1:  0 2 3
  2:  0 1 3
  3:  0 1 2

 fim da impressao do grafo.
 Grafo é completo? True
```