Universidade Federal de Viçosa - Campus Florestal

Disciplina: CCF 331 - Teoria e Modelo de Grafos

Professor: Marcus Henrique Soares Mendes

Trabalho Prático 1

Eduardo Vinicius - 3498

Lucas Takeshi - 2665

Victor Hugo - 3510

Paulo Henrique - 4221

Sumário

Introdução

Após, em ambiente virtual de aula, conhecer sobre a estrutura de dados **grafos** e alguns conceitos e algoritmos ao seu redor, o professor Marcus Henrique Soares Mendes propôs a implementação de uma biblioteca para manipulação de **grafos não direcionados ponderados**. Tal biblioteca deve ser facilmente **reusável** para quaisquer programas que necessitem de alguma funcionalidade contida na biblioteca.

Um grafo G é constituído de um conjunto **V** não-vazio cujos elementos são chamados de **vértices** (ou nós), e um conjunto **A** de pares não ordenados de elementos de V, chamados de **arestas**. (RANGEL, et. al, 2018)

A representação computacional utilizada para implementação da biblioteca é a **matriz de valores**, que constrói uma matriz n x n com **n** sendo o número de vértices e atribui a cada posição **ij** se existe a aresta entre o vértice i e o j. A linguagem utilizada para implementação escolhida foi **Python,** portanto será necessária a instalação prévia da linguagem na máquina de execução.

Desenvolvimento

As funcionalidades do trabalho prático foram implementadas, a partir da leitura de uma entrada padrão .txt ou .json, na qual esses dados são utilizados para formar uma **matriz de adjacência** de um grafo ponderado, e a partir da sua estruturação os demais métodos do programa podem ser executados, sendo úteis assim para retornar informações relevantes sobre determinado grafo.

```
class Grafo(object):

    def __init__(self):  # inicializa as estruturas base do grafo

        self.matriz = []
```

Figura 1 - Classe Grafo

Como dito anteriormente, o grafo é armazenado em uma matriz, sendo ela um componente da classe "Grafo", que contém os demais métodos que vão ser descritos a seguir.

Ordem do grafo

O método retorna a **ordem** de um grafo, obtido diretamente através do tamanho da matriz de adjacência.

```python
def ordem(self):  # ordem = n° de vértices = n° de colunas ou n° de linhas da matriz, já que ambos são iguais
    ordem = len(self.matriz)
    return ordem
```

Figura 2 - implementação da funcão *ordem*

Tamanho do grafo

Método retorna o **tamanho** do grafo, percorrendo a matriz de adjacências e verificando o número de arestas que ele possui.

```python
def tamanho(self):
    tamanho = 0
    for i in range(len(self.matriz)):
        for j in range(len(self.matriz)):
            if (self.matriz[i][j] != 0):
                tamanho += 1
    # sabemos que a matriz contém o mesmo peso 2x, uma em cada linha dos vértices ligados, logo,
    # basta dividir tamanho por 2, para obter a quantidade de arestas
    tamanho = (tamanho / 2)
    return int(tamanho)
```

Figura 3 - implementação da funcão *tamanho*

Densidade de um grafo

A **densidade** de um grafo é obtida a partir da divisão do seu tamanho por sua ordem.

```python
def densidade(self):
    return self.tamanho() / self.ordem()  # Densidade = numero de arestas dividido pelo numero de vertices
```

Figura 4 - implementação da funcão *densidade*

Retorna vizinhos de um vértice

Esse método recebe um **vértice**, e percorre entre as colunas da matriz de adjacência procurando arestas conectadas a um outro vértice, caso encontre ele é adicionado à lista de vizinhos.

```
def retornaVizinhos(self, vertice):
    vizinhos = []
    for i in range(len(self.matriz)):
        if self.matriz[vertice - 1][i] != 0:
            vizinhos.append(i + 1)
    return vizinhos
```

Figura 5 - implementação da funcão *retornaVizinhos*

Determina grau de um vértice

O método retorna o grau de um vértice, percorrendo a matriz de adjacência e contabilizando o **número** de vizinhos daquele vértice.

```
def grauVertice(self, vertice):
    grau = 0
    for i in range(len(self.matriz)):
        if self.matriz[vertice - 1][i] != 0:
            grau += 1
    return grau
```

Figura 6 - implementação da funcão *grauVertice*

Verificar se um vértice é articulação

Retorna a um valor booleano, a partir de uma busca para verificar se aquele vértice é ou não uma articulação do grafo.

```
# Verifica se um vertice é articulação
def articulacao(self, vertice):
    if vertice not in self.matriz:
        return False
    for vizinho in self.retornaVizinhos(vertice):
        comVertice = []
        semVertice = []
        self.busca(vizinho, comVertice)
        self.busca(vizinho, semVertice, vertice)
        comVertice.sort()
        semVertice.sort()
        if comVertice != semVertice:  # compara se ouve alguma mudança nos vertices marcados
            return True
    return False
```

Figura 7 - implementação da funcão *articulacao*

Busca em Largura

Realiza a busca em largura, mostrando a ordem de vértices visitados, e as arestas que não fazem parte da busca.

```python
def buscaEmLargura(self, vertice):
    fila = []
    marcados = []
    i = 1

    fila.append(vertice)
    marcados.append(vertice)

    matrizArestas = [[0] * len(self.matriz) for _ in range(len(self.matriz))]
    foraDaBusca = [[0] * len(self.matriz) for _ in range(len(self.matriz))]

    while len(fila) > 0:
        v = fila[0]
        vizinhos = self.retornaVizinhos(v)

        for w in vizinhos:
            if w not in marcados: #se o vertice não foi explorado, o adiciona na fila para ser explorado
                matrizArestas[v - 1][w - 1] = 1  # adiciona as arestas na árvore de busca em largura
                matrizArestas[w - 1][v - 1] = 1
                marcados.append(w)
                fila.append(w)
            elif w in fila:
                foraDaBusca[v-1][w-1] = 1
        fila.remove(v)             #remove o vertice explorado da fila

    print("Ordem de vizitacao dos vertices na busca em largura:")
    for vertice in marcados:
        print(' ->',vertice, end='')
    print()

    print("Arestas que nao fazem parte da busca em largura:")
    for i in range(len(foraDaBusca)):
        for j in range(len(foraDaBusca)):
            if foraDaBusca[i][j] == 1:
                print(f"Aresta {i+1} {j+1}")
```

Figura 8 - implementação da funcão *buscaEmLargura*

Componentes Conexas

O algoritmo descrito na figura 9 apresenta uma maneira diferente das casuais de se obter a **quantidade de componentes conexas** de um determinado grafo. Sabe-se que uma componente conexa de um grafo é um subgrafo conexo maximal de G, ou seja, são os "pedaços" conexos do grafo em questão (MENDES, M. 2022).

A solução em questão é chamada de **Union Find.** Este algoritmo realiza algumas operações determinadas ***Union()*** e ***Find()*** para os vértices do grafo. ***Find()*** é um método responsável por apresentar a componente (**1 ou mais vértices unidos através de suas arestas**) que contém um determinado vértice, passado como parâmetro. ***Union()*** é responsável por unir duas componentes do grafo.

Vale lembrar que o algoritmo considera os vértices do grafo inicialmente separados, logo temos n componentes iniciais, sendo **n = número de vértices**. Outro detalhe importante é que todo vértice é “pai”de si mesmo.

Considerando estes detalhes, o algoritmo percorre todas as arestas do grafo e verifica qual o vértice “**pai**” dos vértices da aresta em questão, após percorrer todas as arestas, temos todos os vértices mapeados para cada uma de suas componentes.

A cada operação de ***Union()*** feita, significa que uma componente foi unida a outra e temos uma componente a menos, ao final, temos o grafo original totalmente obtido e n - (quantidadeVezesOperacoesUnion) componentes conexas.

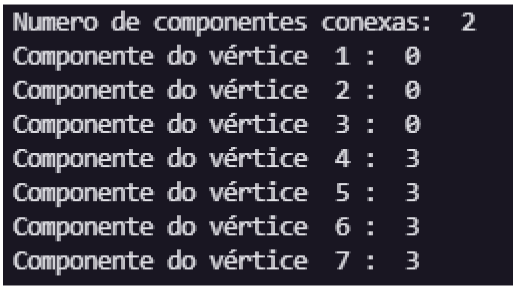
```python
def find(self, n1, pais = []): # encontra a qual componente o i-ésimo nó pertence

    componente = n1

    while(componente != pais[componente]):
        pais[componente] = pais[pais[componente]]
        componente = pais[componente]
    return componente

def union(self, n1, n2, pais = [], tamanhoComponente = []): # faz a união entre as componentes
    p1, p2 = self.find(n1, pais), self.find(n2, pais)

    if p1 == p2: # nao sera feita a uniao, pois ja pertencem a mesma componente
        return 0

    if (tamanhoComponente[p2] > tamanhoComponente[p1]): # sera feita a uniao, pois nao pertencem a mesma componente
        pais[p1] = p2
        tamanhoComponente[p2] += tamanhoComponente[p1]
    else:  # sera feita a uniao também
        pais[p2] = p1
        tamanhoComponente[p1] += tamanhoComponente[p2]

    return 1
```

Figura 9 - ***UnionFind*** responsável por obter as componentes conexas de um grafo e seus vértices

Esta funcionalidade em questão deve receber uma atenção extra, pois foi originalmente apresentada pelo canal NeetCode do *YouTube* e **adaptada** para atender os requisitos deste trabalho prático, além da **estrutura** utilizada.

Além disso, vale ressaltar que cada componente conexa do grafo é rotulada com um determinado vértice, denominado vértice “pai”. Por exemplo, se existem **3** componentes conexas, cada uma delas será rotulada por **3** vértices diferentes, cada vértice pertencente a cada uma delas. Logo, na hora de se obter os vértices e quais suas componentes conexas, deve-se atentar à qual componente conexa um determinado vértice pertence.

Como por exemplo na execução abaixo, para um grafo com 2 componentes conexas:



```
Numero de componentes conexas:  2
Componente do vértice  1 :  0
Componente do vértice  2 :  0
Componente do vértice  3 :  0
Componente do vértice  4 :  3
Componente do vértice  5 :  3
Componente do vértice  6 :  3
Componente do vértice  7 :  3
```

Figura 10 - Execução de um grafo com 2 componentes conexas

Deve-se atentar a componente obtida, os valores foram 0 e 3 (2 valores distintos, cada um referente a uma componente distinta). A componente 1 está sendo representada pelo vértice pai ‘0’ e a componente 2 está sendo representada pelo vértice ‘3’. Sendo assim, se obtivemos um grafo com 1 componente somente, esta será rotulada por 1 vértice pai e **todos os vértices terão componentes referentes ao vértice “pai”**, que pode ser qualquer um, mas devido a estrutura do algoritmo, será sempre o vértice de menor número de uma componente diferente.

Verificar se o grafo possui ciclo

O método retorna um valor booleano para indicar se o grafo possui ou não ciclos, o algoritmo percorre de maneira recursiva os vértices do grafo, marcando aquele que está visitando e olhando em sua matriz de adjacência o próximo vértice vizinho a aquele, caso o algoritmo visite um vértice já marcado então é determinado que aquele grafo possui um ciclo.

```python
# Cria uma copia da matriz de adjacencia e uma lista de vertices, chama a verificacao
def verificaCiclo(self):
    vertices = []
    matriztemp = []
    for i in range(len(self.matriz)):
        vertices.append(Elemento())
        vertices[i].vertice = i + 1
    for i in range(len(self.matriz)):
        matriztemp.append([0] * len(self.matriz))
        for j in range(len(self.matriz)):
            matriztemp[i][j] = (self.matriz[i][j])
    return self.verificacao(vertices, 0, matriztemp)

# Verifica se possui um ciclo, marcando vertices e excluindo arestas ja visitadas.
def verificacao(self, vertices, v, matriztemp):
    if vertices[v].marcado:
        return True
    else:
        for i in range(len(self.matriz)):
            if matriztemp[v][i] != 0:
                vertices[v].marcado = True
                matriztemp[v][i] = 0
                matriztemp[i][v] = 0
                vertices[v].proximo = i + 1
                return self.verificacao(vertices, i, matriztemp)
    return False
```

Figura 11 - ***Verifica ciclo*** responsável verificar se o grafo é cíclico.

Retorna distância e caminho mínimo

O método utiliza o algoritmo de Floyd-Warshall para calcular a distância e o caminho mínimo, esse algoritmo possui a restrição de poder ser utilizado apenas em ciclos não negativos.

Funciona basicamente criando duas matrizes L e R, onde ao final da iteração a matriz L vai armazenar as distâncias de cada par de vértices, e sendo a matriz R a matriz que armazena o caminho entre cada par de vértices.

Caso não exista caminho entre determinado par vértice (i,j), a matriz L vai conter nesse dado (i,j) o valor infinito.

```python
# Algoritmo de Floyd-Warshall (Aplicado a grafos com ciclos positivos)
def menorCaminhoVertice(self, v):
    matrizL = []
    matrizR = []
    infinito = float("inf")
    n = len(self.matriz)
    # Inicialização matriz L e R
    for i in range(n):
        matrizL.append([infinito] * n)
        matrizR.append([0] * n)
    for i in range(n):
        for j in range(n):
            if i == j:
                matrizL[i][j] = 0
            else:
                if self.matriz[i][j] != 0:
                    matrizL[i][j] = self.matriz[i][j]
    for i in range(n):
        for j in range(n):
            if matrizL[i][j] != infinito:
                matrizR[i][j] = i + 1
    # Calculo Menor Caminho
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if matrizL[i][j] > matrizL[i][k] + matrizL[k][j]:
                    matrizL[i][j] = matrizL[i][k] + matrizL[k][j]
                    matrizR[i][j] = matrizR[k][j]
    self.imprimeDistancia(matrizL[v - 1], v)
    for i in range(n):
        self.imprimeCaminhoMinimo(matrizR[v - 1], v, i + 1)
```

Figura 12 - ***menorCaminhoVertice*** responsável por obter menores caminhos entre dois pontos

Determinar a árvore geradora mínima

Para resolver a árvore geradora mínima de um grafo, há dois tipos de algoritmos chamados Kruskal e Prim, escolhemos o método Prim para gerar um mapa em que todas as vértices estão interligadas mesmo que indiretamente. O código cria tentativas e erros para achar uma solução percebemos que grafos conexos não se encaixa pois é preciso estar conectados todos os vértices presentes

```python
def printMST(self, parent, n):
    print ("Edge \tWeight")
    for i in range(1, n):
        parentAux = parent[i]
        print (parentAux+1, "-", i+1, "\t", self.matriz[i][parentAux])

def getParams(self, parent, i):

    aresta1 = parent[i]
    aresta2 = i
    peso = self.matriz[i][aresta1]

    return aresta1+1, aresta2+1, peso

def minKey(self, key, mstSet, n):

    # Initialize min value
    min = sys.maxsize

    min_index = 0

    for v in range(n):
        if key[v] < min and mstSet[v] == False:
            min = key[v]
            min_index = v

    return min_index

def primMST(self, n):
    key = [sys.maxsize] * n
    parent = [None] * n
    key[0] = 0
    mstSet = [False] * n
    parent[0] = -1
    for cout in range(n):
        u = self.minKey(key, mstSet, n)
        mstSet[u] = True
        for v in range(n):
            if self.matriz[u][v] > 0 and mstSet[v] == False and key[v] > self.matriz[u][v]:
                key[v] = self.matriz[u][v]
                parent[v] = u

    self.printMST(parent, n)

    return parent
```

Figura 13 - ***primMST*** função que gera uma árvore com interligada com todos os vértices

:

```
Componete conexo, logo nao eh possivel!
```

```
texto.txt
7
1 2 1.2
2 5 2.3
1 5 0.1
6 3 -8.4
6 4 4.6
4 3 0.3
```

Figura 14 - Possíveis saídas 2 com arquivo texto

```
5
1 2 1.2
2 5 2.3
1 5 0.1
5 3 -8.4
5 4 4.6
4 3 0.3
```

```
->12
5
1   2   1.2
4   3   0.3
5   4   4.6
1   5   0.1
```

Figura 15 - Possíveis saídas 2 com arquivo texto

Verificar se o grafo é euleriano

O algoritmo verifica se um grafo é euleriano verificando se todos os vértices têm grau par, depois verifica se o grafo é conexo. Se o grafo é conexo, o algoritmo faz a chamada da função que retorna uma cadeia euleriana fechada.

```python
def verificarEuliriano(self):
    #verificar se o grafo tem todos os vertices com grau par
    for i in range(len(self.matriz)):
        if self.grauVertice(i) % 2 == 1:
            print("Grafo nao e euliriano vertice", i+1)
            return False

    #verificar se o grafo é conexo
    conexo = []
    self.busca(1, conexo)
    if len(conexo) != len(self.matriz):
        print("Grafo nao e euliriano")
        return False
    #Se passou dos testes o grafo é euliriano
    print("Grafo e euliriano")

    copia = [[] * len(self.matriz) for _ in range(len(self.matriz))]
    for i in range(len(self.matriz)):
        copia[i] = self.matriz[i].copy()
    #determinar uma cadeia euliriana fechada com o algoritmo de Fleury
    self.cadeiaEuliriana()
```

Figura 16 - ***VerificaEuleriano*** função que verifica que se compre as requisições para se considera euleriano

Cadeia euleriana

O algoritmo da cadeia euleriana fechada segue o algoritmo de Fleury, baseado em fila, onde o primeiro vértice a entrar na fila será o primeiro vértice a sair.

O algoritmo é iniciado em um vértice qualquer, o vértice é adicionado na fila, após isso verifica as arestas incidentes nesse mesmo vértice, e escolhe a aresta deixando as pontes para serem "atravessadas" por último. Em seguida o algoritmo marca a aresta atravessada na matriz, adiciona o vértice que é ligado pela aresta na fila, e retira o vértice atual. Assim a repetição estará completa quando não houver arestas para serem atravessadas.

```python
def cadeiaEuliriana(self):

    v0 = 1
    cadeia = [v0] #cadeia inicia num vertice qualquer

    matriz = [[] * len(self.matriz) for _ in range(len(self.matriz))]
    for i in range(len(self.matriz)):
        matriz[i] = self.matriz[i].copy()

    while True:
        vertice = cadeia[len(cadeia) -1] #vertice a ser explorado é o ultimo na cadeia
        arestasIncidentes = []

        for  i in range (len(matriz)):
            if (matriz[vertice-1][i] != 0):
                arestasIncidentes.append(i) #arestas incidentes no vertice

        if (len(arestasIncidentes) == 0): #se não há mais arestas o algoritmo chegou ao fim
            break

        if (len(arestasIncidentes) == 1): #se há somente uma aresta é ela que vamos atravessar
            aresta = arestasIncidentes[0]

        else:
            pos = 1

            for j in arestasIncidentes: #para cada aresta incidente no vertice
                cont = 0
                if pos == len(arestasIncidentes): #se é a ultima aresta no vetor de arestas inicidentes +
                    aresta = j                  # então as outras são ponte, escolheremos essa para atravessar
                else:
                    for aresta in matriz[j]:    #conta a quantidade de arestas incidentes no vertice ligado ao +
                        if aresta != 0:         # vertice atual pela aresta atual.
                            cont +=1
                    if (cont > 1):              #se há mais de uma aresta, a aresta atual não é uma ponte
                        aresta = j
                        break
                    pos += 1
        matriz[vertice-1][aresta] = 0          #Marca a aresta como explorada
        matriz[aresta][vertice-1] = 0          #Marca a aresta como explorada
        cadeia.append(aresta+1)

    print("Cadeia euliriana no grafo: ", cadeia)
```

Figura 17 - ***CadeiaEuleriana*** função responsável para verificar se há uma cadeia Euleriana

Importação em Json

Método faz a leitura de arquivo .json no formato das entradas do site https://paad-grafos.herokuapp.com.

Obs:O arquivo .json deve estar na pasta “./src”.

```python
def lerJson(self, nomeArquivo):
    try:
        with open(f".\\src\\{nomeArquivo}", encoding='utf-8') as meu_json:
            dados = json.load(meu_json)
    except:
        print("arquivo nao encontrado, verifique se esta na pasta src")


    nomeArquivo = nomeArquivo.replace("json","txt")
    try:
        arquivo = open(f".\\src\\{nomeArquivo}", "w+")
    except:
        print("Erro ao abrir o arquivo")

    arquivo.writelines(f"{ dados['data']['nodes']['length'] }\n")

    arestas = []
    for prop in dados["data"]["edges"]["_data"].values():
        arestas.append(prop)

    vertices = []
    for prop in dados["data"]["nodes"]["_data"].values():
        vertices.append(prop)
    for aresta in arestas:
        vertice1 = aresta['from'] if  aresta['from'] == vertices[aresta['from']-1]['label'] else vertices[aresta['from']-1]['label']
        vertice2 =  aresta['to'] if  aresta['to'] == vertices[aresta['to']-1]['label'] else vertices[aresta['to']-1]['label']
        peso = aresta['label']
        arquivo.writelines(f"{vertice1} {vertice2} {peso}\n")
```

Figura 18 - ***lerJson*** responsável por ler arquivos .Json

Exportação em Json

Método exporta arquivo .txt da entrada do programa para o ,json, utilizando como base o arquivo "base.json", o arquivo de saída é formatado conforme o padrão do site https://paad-grafos.herokuapp.com, e tem o nome"data.json".

```python
def escreverJson(self):
    with open(".\\src\\base.json", encoding='utf-8') as meu_json:
        j = json.load(meu_json)


    for v in range (len(self.matriz)):
        vertice = {
                "id": v+1,
                "x": 45,
                "y": 45,
                "label": f"{v+1}"
        }
        j["data"]["nodes"]["_data"][f"{v+1}"] = vertice

    j["data"]["nodes"]["length"] = f"{len(self.matriz)}"

    id = 1

    matriz = [[] * len(self.matriz) for _ in range(len(self.matriz))]
    for i in range(len(self.matriz)):
        matriz[i] = self.matriz[i].copy()

    for i in range (len(matriz)):
        for x in range (len(matriz)):
            if(matriz[i][x] != 0):

                aresta = {
                            "from": f"{i+1}",
                            "to": f"{x+1}",
                            "label": f"{matriz[i][x]}",
                            "id": f"{id}",
                            "color": {}
                        }
                matriz[i][x] = 0
                matriz[x][i] = 0
                j["data"]["edges"]["_data"][f"{id}"] = aresta
                id += 1


    with open('data.json', 'w') as f:
        json.dump(j, f)
```

Figura 19 - ***escreveJson*** responsável por exportar o arquivo como .Json

Conclusão

Ao fim do projeto se tem uma biblioteca funcional com diversos métodos úteis para elaboração de grafos e apresentação de conceitos e propriedades, além da integração através da importação e exportação de grafos do site https://paad-grafos.herokuapp.com.

Esse trabalho prático, foi de grande importância para o grupo, para aprender e reforçar os conceitos de grafos na prática, todas as funcionalidades foram implementadas com base na teoria e nos fundamentos da matéria, de modo que os membros do grupo conseguiram elaborar algoritmos mais complexos e expandir seus conceitos através da construção e exploração de grafos não direcionados e ponderados.

Referências

Samuel Jurkiewicz; Paulo Oswaldo Boaventura Netto. Grafos: Introdução e Prática. Editora Blucher 192 ISBN 9788521211327. Disponível em: Biblioteca virtual Pearson.

Simões-pereira, José Manuel dos Santos. Grafos e redes - Teoria e Algoritmos Básicos. Editora Interciência 356 ISBN 9788571933316 Disponível em: Biblioteca virtual Pearson.

Socorro Rangel, Valeriano A. de Oliveira, Silvio A. Araujo - Elementos de Teorias de Grafos - Notas de Aulas. UNESP. Disponível em: PVANet Moodle

Marcus Mendes - Notas de Aula. UFV-Campus Florestal. Disponível em: PVANet Moodle