Estrutura de Dados Avançados
Projeto 1 – Teoria dos Grafos

Gustavo S. Silva[1]

[1]Centro de Ciências Tecnológicas (CCT)
Universidade Estadual do Maranhão (UEMA) – São Luís, MA – Brasil

[1]gustavosilva7@aluno.uema.br, gustavosoares112@gmail.com

1. Introdução

No contexto atual, é notória a importância da utilização de grafos. Aplicações que vão desde a computação de rotas até a escolha do melhor caminho. Por esses e outros motivos, o conhecimento de sua construção se faz conveniente.

Para a execução desse projeto, foi escolhida a linguagem de programação Python por ser mais compacta, e ter tecnologias que simplificam alguns processos. A estrutura de dados selecionada foi a matriz de adjacência – onde os dados serão armazenados, e a API de visualização é a NetworkX.

2. Importações

Diante dos desafios apresentados, a importação de bibliotecas torna- se imprescindível, principalmente quando se fala em complexidade. Métodos que demandariam maior processamento, são facilmente substituídos por uma versão eficiente e compacta de uma biblioteca.

```
1   import numpy as np
2   import networkx as nx
3   import matplotlib.pyplot as plt
```

Figura 1. Bibliotecas utilizadas

A utilização da biblioteca **numpy**, neste caso, foi pensada para a criação da matriz de adjacência através do método *zeros([dimensão_1, dimensão_2])*. As duas última foram aplicadas na construção do grafo: criação e visualização, respectivamente.

3. Codificação dos vértices

Após a leitura, armazenamento e tratamento dos dados do arquivo nos vetores **arestas** e **vértices**, o próximo passo seria jogá-los na estrutura de dados escolhida (matriz de adjacência), mas houve a necessidade de codificá-los, para que cada vértice representasse um índice da matriz.

Dentre as maneiras de fazer isso, a selecionada foi a utilização de um dicionário acrescido do método *enumerate(lista)*. Para isso, foi empregado o conceito de *comprehension* para o dicionário **codigoVertices**, que armazena o vértice em *string* e seu respectivo valor inteiro obtido através da função *enumerate.*

```
100 codigoVertices = {vertice: valor
101                   for valor, vertice
102                   in enumerate(vertices)}
103 for vX, vY in arestas:
104     g.addAresta(codigoVertices[vX], codigoVertices[vY])
```

Figura 2. Codificação dos vértices

Depois de atribuir valores para os vértices, basta jogá-los – os valores – na estrutura de dados a partir das arestas (vetor do tipo [[‘vértice’, ‘vértice’], ...]).

Para melhor demonstrar, imagina-se duas listas, `V = ['v1','v5','v3','v10','v2']` e `A = [['v1','v10'],['v5','v3'],['v1','v2'], ['v10','v5']]`, representando vértices arbitrários.

Pressupondo o código da Figura 2 da linha 100 a 102 para a lista **vértices**, o resultado atibuído a **codigoVertices** será: `{'v1':0,'v5':1,'v3':2,'v10':3,'v2':4}`.

Paras as linhas 103 e 104, as arestas vão sendo passadas de uma a uma para a função addAresta do Grafo, mas através dos valores já estabelecidos dos vértices, ou seja:

Primeria iteração: `A = ['v1','v10']`; valores passados: (0, 3);

Segunda iteração: `A = ['v5','v3']`; valores passados: (1, 2);

(...)

Os próximos passos serão definidos em outras funções.

4. Grafo

Para esse tipo de situação, é recomendável trabalhar com orientação a objetos, principalmente por ter um maior controle em relação às funções, e por ser flexível.

4.1. Construtor

O método construtor da classe *Grafo* é inicializado com o *tipo* (primeira linha do arquivo de texto) e o *tamanhoV* (tamanho da lista de vértices não repetidos). Como a matriz de adjacência quadrada, o *tamanhoV* ditará a dimensão desta. Para isso, utiliza-se a função *zeros* da biblioteca **numpy** para definir uma matriz *tamanhoV* x *tamanhoV* de inteiros (*dtype = int*). Se a quantidade de vértices – não repetidos – for 5, por exemplo, a matriz será 5x5 de zeros.

```
5  class Grafo:
6      def __init__(self, tipo, tamanhoV):
7          self.tipo = tipo
8          self.tamanhoV = tamanhoV
9          self.matriz = np.zeros([tamanhoV, tamanhoV], dtype = int)
```

Figura 3. Método construtor

A matriz de adjacência poderia ter sido gerada através do conceito de *list comprehension*, por exemplo, mas para uma quantidade de vértices relativamente

grande, esse método apresentou um tempo de execução alto. Então, prezando por uma experiência otimizada, empregou-se a função *zeros,* que aumentou significativamente o desempenho, especialmente quando a matriz é mostrada.

4.2. Adicionar arestas

Esse método recebe o código/índice de dois vértices `(vX,vY)` – como visto o Item 3 – e define, na matriz de adjacência, o valor 1 para a coordenada `[vX,vY]`. Se o grafo for não-direcionado, a coordenada `[vY,vX]` também recebe o valor 1. Esse processo preenche a matriz com as adjacências entre os vértices.

```
11   def addAresta(self, vX,  vY):
12     self.matriz[vX][vY] = 1
13     if self.tipo == "ND":
14       self.matriz[vY][vX] = 1
```

Figura 4. Método para adicionar aresta à matriz

Exemplo: tomando uma aresta `A = ['v5','v3']` tem-se as seguintes matrizes:

```
v1  v5  v3  v10  v2        v1  v5  v3  v10  v2
0   0   0   0    0         0   0   0   0    0
0   0   1   0    0         0   0   1   0    0
0   0   0   0    0         0   1   0   0    0
0   0   0   0    0         0   0   0   0    0
0   0   0   0    0         0   0   0   0    0
```

Figura 5. Matriz de um grafo dirigido e não-dirigido, respectivamente

Os índices dos vértices são jogados na matriz como coordenadas: código['v5'] = 1; código['v3'] = 2, e as posições `matriz[1][2]` e `matriz[2][1]` terão o valor 1, representando a aresta. Para o tipo da matriz para o grafo dirigido, apenas a coordenada `matriz[1][2]` seria preenchida.

Após adicionar todas as arestas ao grafo, a estrutura de dados está completa.

4.3. Verificar adjacência

Dados dois vértices, já se possui ambos os índices, então basta verificar na matriz de adjacência se existe uma aresta entre eles, ou seja, quando a coordenada `[vX,vY]` e `[vY,vY]` para não-direcionado e `[vX,vY]` para direcionado, for 1.

```
24  def saoAdj(self, vX, vY):
25    if self.tipo == 'ND':
26      if ((self.matriz[vX][vY] == 1) and (self.matriz[vY][vX] == 1)):
27        print('Adjacentes!')
28      else:
29        print('Não adjacentes!')
30    else:
31      if (self.matriz[vX][vY] == 1):
32        print('Adjacentes!')
33      else:
34        print('Não adjacentes!')
```

Figura 6. Verificar adjacência

Seguindo o mesmo exemplo da Figura 5, é possível afirmar que existe adjacência entre os vértices `v5` e `v3`.

Primeiramente realiza-se a verificação do tipo de grafo trabalhado. Se for não-direcionado, precisa haver uma aresta entre os vértices, ou seja, `matriz[1][2]` e `matriz[2][1]` precisam ser 1. Se for direcionado, apenas `matriz[1][2]` precisa representar uma aresta. Caso verdadeiro, significa que os vértices inseridos são adjacentes.

4.4. Grau de um vértice

Para definir o grau de um vértice, deve-se levar em consideração que, para grafos direcionados, o vértice possui grau de entrada e saída. Para grafos não dirigidos, o grau é o mesmo que o grau de entrada.

```
36  def getGrau(self, vertice):
37      grauEntrada, grauSaida = 0,0
38      for i in self.matriz[:, vertice]:
39          grauEntrada += i
40      for i in self.matriz[vertice, :]:
41          grauSaida += i
```

Figura 7. Definindo o grau de um vértice

O processo consiste em encontrar a linha e coluna de um vértice V inserido. Os valores da linha representam os vértices que V aponta (saída), e a coluna os vértices que apontam para V. Tendo isso em mente, basta verificar se existe aresta entre V e o vértice da linha ou coluna analisada (se coordenada for 1) e somar os valores para cada grau.

Exemplo: Para v5 inserido (ainda levando em consideração a lista V) na função, tem-se as seguintes informações para a matriz do grafo dirigido da Figura 5:

1 – Coluna (self.matriz[:, vertice]): `[0,0,0,0,0]`

2 – Linha (self.matriz[vertice, :]): `[0,0,1,0,0]`

Ou seja, o grau de entrada de v5 será a soma dos elementos da coluna, e o grau de saída será a soma dos elementos de sua linha. Dessa forma, grau_entrada = 0 e grau_saída = 1.

Para um grafo não-direcionado, o grau é concordante com o grau de entrada do vértice. Como, para esse tipo, a coordenada inversa também indica uma aresta (1), a coluna seria [0, 0, 1, 0, 0], resultando em grau 1.

4.5. Função getKey

Como os valores que são adicionados à estrutura de dados são os índices/códigos dos vértices, eventualmente há a necessidade de saber qual vértice representa esse valor. Dessa forma, definiu-se a função auxiliar *getKey( )*:

```
17      def aux_getKey(self, valorV):
18          vertice = [vertice for (vertice, valor)
19                                 in codigoVertices.items()
20                                 if valor == valorV]
21          return vertice
```

Figura 8. Função para pegar a key (vértice) de um valor (índice)

As linhas 18 a 20 mostram que serão recebidos todos os vértices e valores de *codigoVertices.item( )*, que retorna os valores do dicionário (vertice : valor), verificar se o valor do vértice de cada iteração é igual ao *valorV* (valor inserido) e adicionar em *vertice.*

Exemplo: tomando a lista de vértices `V = ['v1', 'v5', 'v3', 'v10', 'v2']` e o dicionário *codigoVertices*. Para *aux_getKey(2)*, por exemplo, o valor retornado do vértice será 'v3'.

4.6. Vizinhos

```
48  def getVizinhos(self, vertice):
49      vizinhos = []
50      for vert in range(self.tamanhoV):
51          if self.matriz[vertice][vert] == 1:
52              vizinhos.append(self.aux_getKey(vert))
53      return vizinhos
```

Figura 9. Código da função getVizinhos

A função de *getVizinhos* é procurar todas as adjacências do vértice indicado no argumento, para isso, a função analisa todos os elementos da linha do vértice – definida pelo seu código. Se o elemento for 1, significa que há adjacência, adicionado o vértice da coluna que esse elemento representa, à lista vizinhos.

Exemplo: Levando em consideração a matriz do grafo não-direcionado da Figura 5. Chamando *getVizinhos* para v5, tem-se os seguintes passos:

1 – Analisar cada elemento da linha 1 (código de v5);

- Linha[1] = `[0,0,1,0,0]`

2 – Verificar as adjacências de v5, ou seja, quais coordenadas equivalem a 1;

- Único elemento 1 da linha equivale ao vértice v3 (`matriz[1][2]=1`)

3 – Adicionar o vértice correspondente a coluna, à lista vizinhos;

- Vizinhos = `['v3']`

4.7. Visitar arestas

```
54  def visitarArestas(self):
55      print('Arestas visitadas: ')
56      for i in range(self.tamanhoV):
57          for j in range(self.tamanhoV):
58              if self.matriz[i][j] == 1:
59                  print(self.aux_getKey(i), self.aux_getKey(j), sep='-')
```

Figura 10. Código para visitar todas as arestas da matriz

A função para visitar as arestas é relativamente simples. Basta percorrer os elementos da matriz, linha por linha, verificar se a coordenada *linha x coluna* corresponde a 1 e, se verdadeiro, imprimir os vértices equivalentes (linha e coluna).

Exemplo: visitar as arestas da matriz do grafo direcionado da Figura 5.

`self.matriz[0][0] == 1? (False), self.matriz[0][1] == 1? (False), ..., self.matriz[1][1] == 1? (False), self.matriz[1][2] == 1? (True)`

`print(key(i),key(j))`. Valores impressos: `'v3', v'5'`

A execução continua até `self.matriz[tamanhoV][tamanhoV]`.

4.8. Visualizar grafo

Para a visualização do grafo, foi utilizada a API **networkx**.

```
61  def visualizarGrafo(self):
62      if self.tipo == "ND":
63          graph = nx.Graph()
64      else:
65          graph = nx.DiGraph()
66
67      graph.add_nodes_from(tuple([vertice for vertice
68                                  in vertices if vertice != '']))
69      graph.add_edges_from(tuple([aresta for aresta
70                                  in arestas if aresta[0] != ''
71                                  and aresta[1] != '']))
```

Figura 11. Tipo do gráfico e adição dos vértices e arestas

Este trecho do código é responsável por definir o tipo do grafo (não-direcionado ou direcionado) e adicionar os valores (vértices e arestas) nele. Com o tipo já adquirido através do arquivo de texto, defini-se *nx.Graph( )* para o tipo não-direcionado e *nx.DiGraph( )* para dígragos, ou grafos direcionados.

Depois de definir a tipagem, as linhas 67 a 71 são responsáveis por anexar os valores no grafo. As linhas 67 e 68 pegam os vértices da lista *vertices* (dados coletados do arquivo). Dois pontos a se considerar são:

1 – A inserção, segundo as normas da API, precisa feita através de uma tupla, por isso o casting para *tuple*.

2 – Eventualmente, um valor **null (‘’)** pode ser carregado dos valores pela adição incompleta dos dados no arquivo de texto, por isso, para evitar que haja uma aresta `[vértice,null]`, é feita a verificação antes de adicionar ao grafo.

O mesmo é válido para as linhas 69 a 71, mas esse intervalo adiciona as arestas à estrutua.

O próximo passo é a visualização, mas antes de mostrá-lo, foram feitas alterações visuais por meio de um dicionário *options* de comandos da API (comando : valor). Após a definição desses valores, a função *nx.draw( )* é chamada, passando como

referência o grafo (com o tipo definido) e as opções gráficas. Essa função é responsável por montar a estrutura. Após este passo, o grafo é mostrado por meio da biblioteca **matplotlib.pyplot**, através do comando *show( ).*

```
73  options = {
74      "with_labels": True,
75      "node_color": "#ffb912",
76      "font_size": 8,
77      "font_weight": "bold",
78      "linewidths": 0.5,
79      "width": 0.4,
80  }
81
82  nx.draw(graph, **options)
83  plt.show()
```

Figura 12. Opções visuais e visualização do grafo

Referências

Networkx. Plot Football. GitHub. Disponível em: https://github.com/networkx/networkx/blob/master/examples/graph/plot_football.py. Acesso em: 30 set. 2020.

NetworkX: Network Analysis in Python. Networkx. Disponível em: https://networkx.github.io/. Acesso em: 30 set. 2020.