Teoria dos Grafos

Ada Cristina França da Silva

[1]Engenharia da Computação – Universidade Estadual do Maranhão

Resumo. *Esse relatório descreve os algoritmos criados para a disciplina de análise e projeto de algoritmos. Cada tópico possui um detalhamento da codificação e exemplos dos resultados. Para o desenvolvimento da solução a biblioteca Pandas, Numpy foram utilizadas para a construção da matriz de adjacência, e a biblioteca networkx foi utilizada para a plotagem gráfica do grafo.*

1. Leitura de Arquivo

O processo de leitura do arquivo é feito pela inserção do nome do arquivo via prompt, no formato 'txt'. Uma função 'leitura_arquivo' foi codificada para receber a url e salvar todas as linhas do arquivos em um array 'lista_linhas', que é retornado pela função.

```
262 # Função leitura_arquivo
263 # Parâmetro: url - caminho do arquivo
264 # Returns: linha_linhas - lista de todas as linhas do arquivo
265 def leitura_arquivo(url):
266 
267     lista_linhas = []
268     try:
269         arquivo = open(url, "r")
270         print("Arquivo carregado.")
271         for linha in arquivo:
272             linha = linha.replace('\n', '')
273             lista_linhas.append(linha)
274     except:
275          print('Problema na leitura do arquivo.')
276 
277     return lista_linhas
```

2. Matriz de Adjacência

A matriz de adjacência é uma matriz formada por 0, 1, onde as colunas e linhas são representadas por cada vértice. O valor '0' é inserido na matriz quando não há conexão entre os vértices, e o valor '1' quando há uma conexão.

A matriz de adjacência pode representar um grafo não direcionado, no qual a direção das arestas não é importante, nesse caso, a matriz é preenchida com valor '1' tanto para coluna e linha, pois a direção entre os vértices não importa. Se o grafo for direcionado, a direção importa, e o '1' é só inserido na direção inserida pelo usuário.

No código implementado, conforme Figura 2. A primeira linha do array serve para identificar se a matriz é direcionada ou não. Se o valor 'lista_linhas[0]' for == 'D', o valor da variável 'self.direcionado' é colocado para 'True'. O grafo foi implementado como uma classe, e nessa classe, o grafo possui as seguintes variáveis:

- direcionado, se o grafo é direcionado ou não;
- matriz_adjacente, matriz pandas para armazenar matriz_adjacente;
- dicionario_vertices, dicionário com todas os vértices da matriz;
- visitado, matriz pandas para verificar se o vértice foi visitado;
- g, objeto networkx para desenhar o grafo.

```
32    def __init__(self, lista_linhas):
33
34        if len(lista_linhas) > 0:
35        # Primeira linha é para identificar se o grafo é direcionado ou não
36            if lista_linhas[0] == "D":
37                self.direcionado = True
```

Figura 2. Código para verificar se é direcionado ou não

3. Grau de um vértice

O cálculo do grau de um vértice é basicamente a quantidade de conexões desse vértice. Com a implementação de uma matriz de adjacência, o grau de um vértice é calculado de forma diferente se o grafo é direcionado ou não. Caso o grafo seja não direcionado, o grau é calculado com a soma dos valores da linha ou da coluna. Se o grafo for direcionado, o grau vai ser dividido em grau de entrada e grau de saída. O grau de entrada é a soma dos valores da linha e o grau de saída a soma dos valores da coluna.

Como a matriz foi implementada usando o pandas dataframe, e o numpy é uma biblioteca matemática, a função np.sum() foi utilizada para soma todas as linhas e colunas do pandas, uma vez que o vértice foi selecionado.

```
84    # grau_vertice
85    # - Recebe vertice
86    # - Descobre o grau do vertice escolhido
87    # - Não retorna nada
88    def grau_vertice(self, vertice):
89        print("")
90        print("GRAU DO VÉRTICE "  + vertice)
91        if vertice in self.dicionario_vertices.values():
92            if not self.direcionado:
93                grau = np.sum(self.matriz_adjacente[vertice])
94                print(self.matriz_adjacente[:][vertice])
95                print('Grau:' + str(grau))
96            else:
97
98                print("Grau de entrada: " + str(np.sum(self.matriz_adjacente.loc[vertice, :])))
99                print("Grau de saída: " + str(np.sum(self.matriz_adjacente.loc[:, vertice])))
100
101       else:
102           print("Vértice não encontrado.")
103
```

Figura 3. Código para cálculo de grau do vértice

Um exemplo da solução é a seguinte matriz não direcionada, no qual foi inserido na aplicação para verificar o grau do vértice 'ada'. Na figura 5 é possível verificar que a soma das conexões ou grau deu o resultado de '4'.

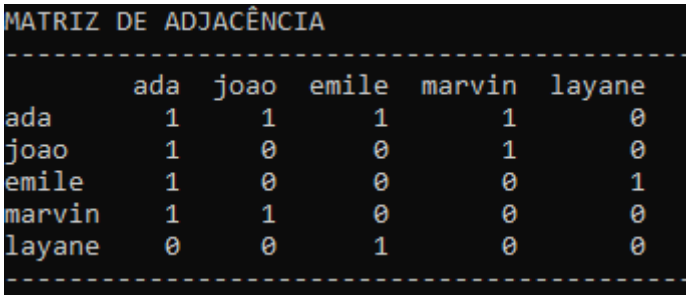

Figura 4. Matriz de exemplo, 5 vértices

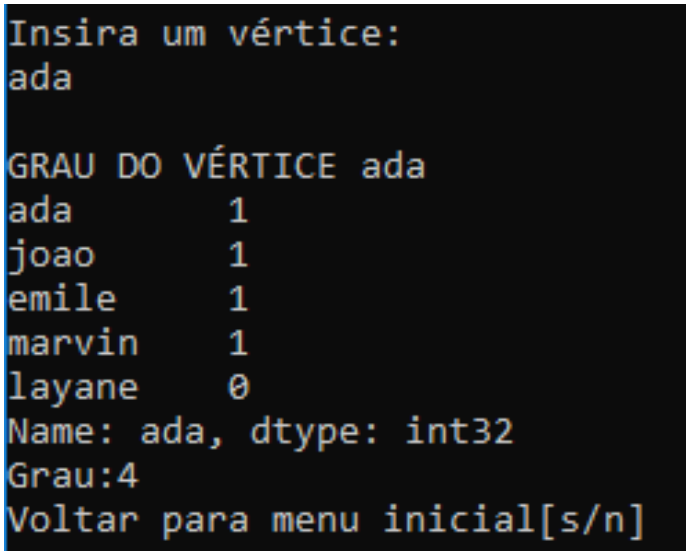

Figura 5. Resultado do cálculo de grau

4. Visitar todas as arestas de um grafo

Para implementar o algoritmo para visitar todas as arestas de um grafo, foi adicionado mais uma matriz booleana, que é utilizada para verificar se a aresta já foi visitada.

Uma função recursiva foi criada, no qual recebe sempre um valor de vértice e dentro de um for visita todas os vértices vizinhos. Se o vértice ainda não foi visitado, a função 'encontra_vizinho' é chamada novamente e o vértice imprimido, e novamente o vértice procura por todos os seus vizinhos.

```
115     # visita_todas_arestas
116     # - Recebe origem de busca
117     # - Função recursiva para percorrer todos os nós
118     # - Não retorna nada
119     def visita_todas_arestas(self, origem):
120
121
122         for adj in self.matriz_adjacente[origem].index:
123             if self.matriz_adjacente[origem][adj] == 1:
124                 if self.visitado[origem][adj] == False:
125                     self.visitado[origem][adj] = True
126                     print("{" + origem + ", " + adj + "}")
127                     self.visita_todos_nos(adj)
128
```

Figura 6. Função recursiva para buscar todas as arestas

Na figura 7, a aplicação lista todos as arestas visitadas, para a matriz de adjacência da figura 4.

```
{ada, ada}
{ada, joao}
{joao, marvin}
{marvin, joao}
{ada, emile}
{emile, layane}
{ada, marvin}
Voltar para menu inicial[s/n]
s
```

Figura 7. Resultado da visita por todas as arestas

5. Receber dois vértices e apresentar se são adjacentes

Dois vértices são adjacentes quando há uma aresta entre eles. No caso da matriz de adjacência, para buscar se dois vértices são adjacente, basta localizar os vértices na matriz e identificar se pelo menos há uma conexão entre eles, no caso, quando o valor na matriz é igual a 1.

Na figura abaixo, como foi utilizado um dataframe, é só passar os nomes dos vértices e identificar se pelo menos há uma conexão na matriz. Condição verificada com o if da linha 77 da figura 8.

```
71    # verifica_adjacencia
72    # - Recebe v1, v2 como o vértice 1 e o vértice 2
73    # - Descobre se dois vértices são adjacentes
74    # - Não retorna nada
75    def verifica_adjacencia(self, v1, v2):
76
77        if (self.matriz_adjacente[v1][v2] == 1) or (self.matriz_adjacente[v2][v1] == 1):
78            print(str(v1) + " e " + str(v2) + " são adjacentes")
79        else:
80            print(str(v1) + " e " + str(v2) + " não são adjacentes")
81
```

Figura 8. Função para verificar adjacência de dois vértices

Na Figura 9, é possível ver o exemplo para a matriz da figura 4, no qual 'ada' e 'joão' são vértices adjacentes.

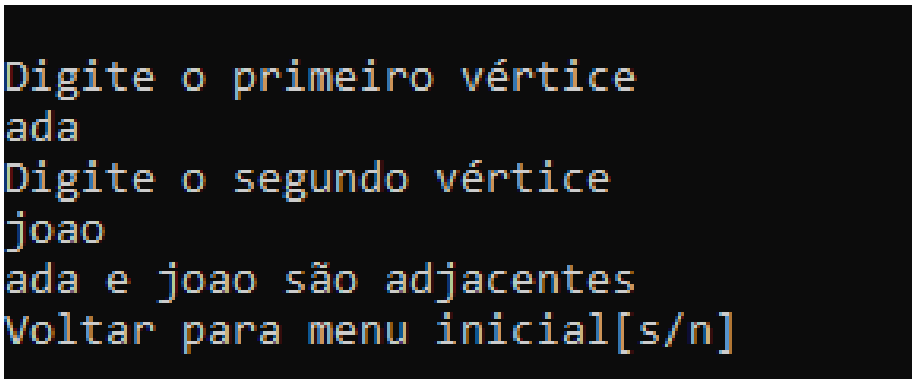

Figura 9. Função para verificar adjacência de dois vértices

6. Buscar todos os vizinhos de um vértice qualquer

Em um grafo, dois vértices v1 e v2 são vizinhos quando há uma aresta a = (v1, v2). No caso, se o grafo for dirigido, a classificação de vizinhos é dividada em sucessor e antecessor. Na busca por todos os vizinhos de um vértice v, os vizinhos sucessores são o vértices que são alcançados pelo vértice v. Os vizinhos antecessores são todos vértices que chegam no vértice v.

Se a matriz for não direcionada, a aplicação vai somente identificar se há uma ligação entre os vértices, com o valor de resultado igual 1, conforme o if da linha 144 da Figura 10.

Se a matriz for direcionada, primeiro há uma busca pelos vizinhos sucessores, no qual os candidatos vizinhos são as colunas em busca, conforme o if da linha 149. Uma segunda etapaé buscar os vizinhos antecessores, agora na busca por linhas, conforme o if da linha 154.

```python
    # grau_vertice
    # - Recebe vertice
    # - Encontra vizinhos desse vertice
    # - Não retorna nada
    def encontra_vizinhos(self, vertice):
        print("")
        print("VIZINHO DO VÉRTICE "  + vertice)
        if vertice in self.dicionario_vertices.values():
            print("-------------------------------------------------------------------------")

            if self.direcionado == False:
                for candidato_vizinho in self.matriz_adjacente[vertice].index:
                    if self.matriz_adjacente[vertice][candidato_vizinho] == 1:
                        print(candidato_vizinho)
            else:
                print("Vizinho Sucessor")
                for candidato_vizinho in self.matriz_adjacente[vertice].index:
                    if self.matriz_adjacente[vertice][candidato_vizinho] == 1:
                        print(candidato_vizinho)

                print("Vizinho Antecessor")
                for candidato_vizinho in self.matriz_adjacente[vertice].index:
                    if self.matriz_adjacente[candidato_vizinho][vertice] == 1:
                        print(candidato_vizinho)
            print("-------------------------------------------------------------------------")
        else:
            print("Vértice não encontrado.")

```

Figura 10. Print do código para encontrar todos os vizinhos

Na Figura 11 é possível verificar o resultado do vértice ’marvin’ para a matriz da Figura 12.

```
/IZINHO DO VÉRTICE marvin
--------------------------------
/izinho Sucessor
joao
/izinho Antecessor
ada
joao
```

Figura 11. Resultado para encontrar os vizinhos do vértice ’marvin’

```
MATRIZ DE ADJACÊNCIA
-------------------------------------------
        ada  joao  emile  marvin  layane
ada       1     0      0       0       0
joao      1     0      0       1       0
emile     1     0      0       0       0
marvin    1     1      0       0       0
layane    0     0      1       0       0
-------------------------------------------
Grafo Dirigido
```

Figura 12. Matriz de adjacência grafo digirido

7. Plotar gráfico

Para plotar o grafo graficamente, a biblioteca networkx foi utilizada. Essa biblioteca é simples, uma vez um objeto grafo criado, a função recebe os vértices que são conectados

e algumas opções de parametrização, como selecionar se o grafo é direcionado ou não. A matriz da Figura 12 foi plotada com a biblioteca conforme a Figura 13.

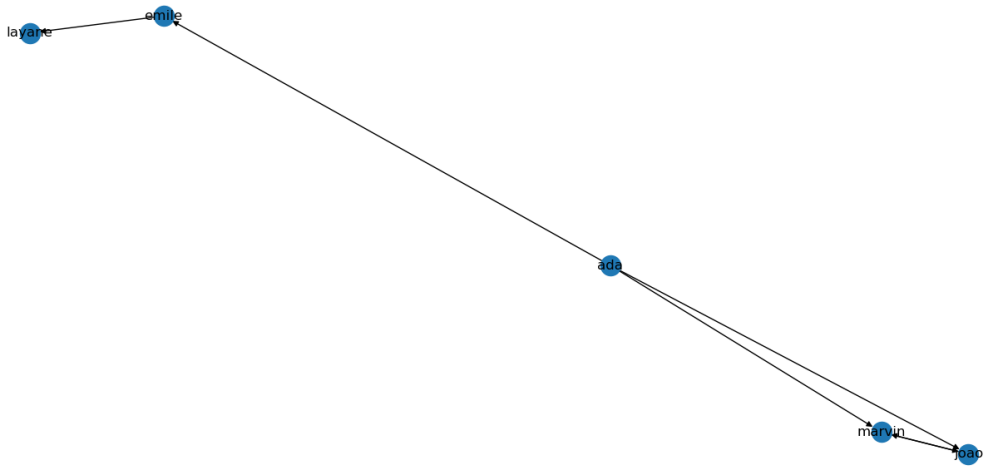

Figura 13. Grafo plotado com a biblioteca networkx

Referências

de Melo, G. S. (2014). Introdução à teoria dos grafos. Tese de Mestrado, Universidade Federal da Paraíba, Mestrado Profissional em Matemática, Paraìba.