CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA
CELSO SUCKOW DA FONSECA – CEFET/RJ

ALGORITMOS EM GRAFOS
Relatório de Trabalho Prático

Breno Martins

Jéssica Fehnle

Josué Cardoso

Leonardo Freire

Professor: Glauco Amorim

Disciplina: Algoritmos em Grafos – Turma 2017.1

CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA
CELSO SUCKOW DA FONSECA – CEFET/RJ

1. Introdução

Este trabalho descreve a projeção, desenvolvimento e validação uma biblioteca para manipular grafos. A biblioteca é capaz de representar grafos, assim como implementar um conjunto de algoritmos em grafos. Seu desenvolvimento foi baseado na linguagem Python, através da implementação da classe *Grafo* com os métodos de manipulação dos grafos. Os métodos são invocados por um arquivo de entrada, que tem a função de passar os dados de carga como parâmetros para as funções de *Grafo*. Os dados de entrada são importados a partir de um arquivo de texto e o retorno das funções invocadas, de igual forma gerará um arquivo de texto.

Para este trabalho, o arquivo componente do projeto responsável pela estruturação da classe Grafo foi nomeado de *grafo.py* e o arquivo de tratamento dos dados de entrada foi nomeado de *teste.py*.

2. Primeira Parte

Nesta primeira etapa do trabalho serão apresentadas as implementações com o uso de grafos não direcionados e arestas sem peso e grafos não-direcionados e arestas com peso, conforme exemplos de Figura 1 e Figura 2, respectivamente.

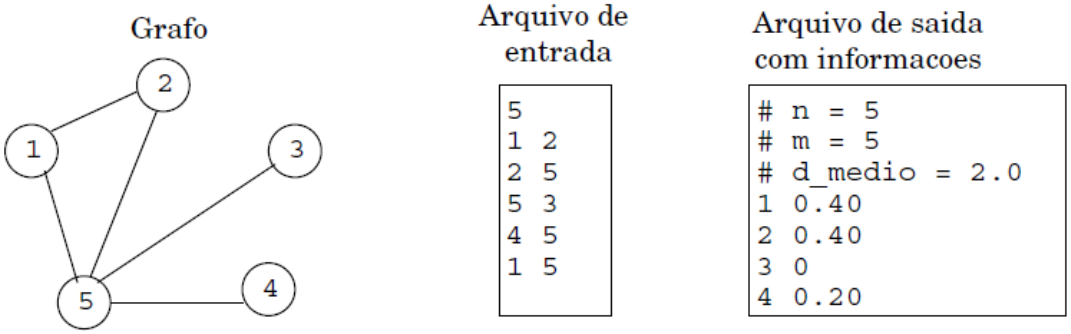

Figura 1 - Grafo não-direcionado e arestas sem peso

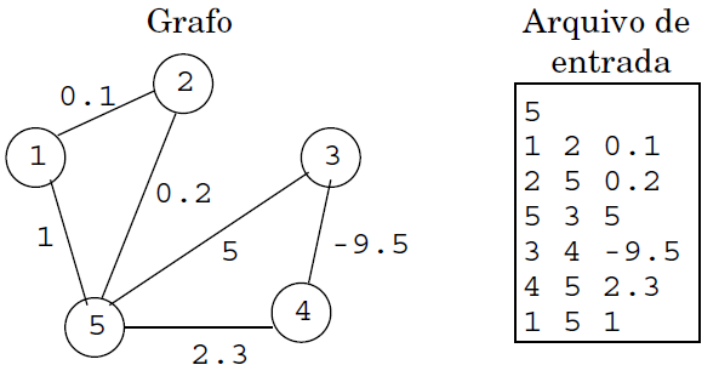

Figura 2 - Grafo não-direcionado e arestas com peso

As parametrizações demandados por este trabalho são a seguir ratificadas:

a) **Arquivo de entrada:** arquivo em formato texto com a primeira linha informando o número de vértices do grafo e cada linha subsequente, as arestas (Figura 1).
b) **Arquivo de saída:** arquivo em formato de texto com as informações de número de vértices, número de arestas e grau médio, além e distribuição empírica do grau dos vértices (Figura 1).
c) **Representação de grafos:** através de matriz de adjacência ou lista de adjacência. O usuário da biblioteca (programa que irá usá-la) poderá escolher a representação a ser utilizada.
d) **Busca em grafos:** largura e profundidade. O vértice inicial será dado pelo usuário da biblioteca. A respectiva árvore de busca deve ser gerada assim como o nível de cada vértice na árvore (nível da raiz é zero). Estas informações devem ser impressas em um arquivo. Para descrever a árvore gerada, basta informar o pai de cada vértice e seu nível no arquivo de saída.
e) **Componentes conexos:** a biblioteca deve ser capaz descobrir os componentes conexos de um grafo. O número de componentes conexas, assim como o tamanho (em vértices) de cada componente e a lista de vértices pertencentes à componente. Os componentes devem estar listados

em ordem decrescente de tamanho (listar primeiro o componente com o maior número de vértices, etc.).

2.1. Implementação

A implementação bordada nesta seção utilizará o arquivo de entrada *grafo_1.txt*, fornecido juntamente com o escopo deste trabalho e sua apresentação envolverá os métodos da classe *Grafo* abaixo descritos:

- ***le_grafo(****param1****)***: responsável pela leitura do arquivo de entrada (*param1*). Sua saída gera parâmetro de entrada para todos os demais métodos desta seção.

- ***gera_arquivo()***: gera o arquivo "saida.txt" contendo informações sobre o grafo lido.

- ***representa_grafo(****param1****)***: imprime em tela a representação do grafo lido como uma lista de adjacência ou matriz de adjacência, onde *param1* define o tipo de representação entre dois valores possíveis: “lista” ou “matriz”.

- ***busca_grafo(****param1****,*** *param2****)***: gera o arquivo "saida.txt" contendo informações da busca em largura (BFS) ou em profundidade (DFS) realizada no grafo lido, conforme definição de parâmetros de entrada, onde *param1* define o tipo de busca, com dois valores possíveis (“bfs” ou “dfs”) e *param2* define o vértice raiz da busca (“A”).

- ***descobre_componentes_conexas()***: gera o arquivo "saida.txt" contendo informações sobre componentes conexas descobertas no grafo lido.

A título de melhor organização desta seção, a dividiremos em seis subseções que abordarão a implementação de cada método individualmente, seguindo a ordem acima definida.

2.1.1. Método *le_grafo(param1)*

```
16      @staticmethod
17      def le_grafo(arquivo):
18
19          arquivo_aberto = open(arquivo, "r")
20          texto = arquivo_aberto.read()
21          arquivo_aberto.close()
22          linhas = texto.splitlines()
23          quantidade_vertices = int(linhas[0])
24          arestas_pesos = {}
25
26          for linha in linhas[1:]:
27              linha_dividida = linha.split()
28
29              if len(linha_dividida) == 2: # sem peso
30                  ponderado = False
31                  aresta = (linha_dividida[0], linha_dividida[1])
32                  arestas_pesos[aresta] = 1
33              elif len(linha_dividida) == 3: # com peso
34                  ponderado = True
35                  aresta = (linha_dividida[0], linha_dividida[1])
36                  peso = linha_dividida[2]
37                  arestas_pesos[aresta] = peso
38
39          grafo = Grafo(quantidade_vertices, arestas_pesos, ponderado)
40          return(grafo)
41
```

Figura 3 - Método *le_grafo()*: implementação

Este método é o responsável pelo tratamento do arquivo de carga *param1*, onde *param1* é o nome do arquivo de carga (que deverá estar contido na pasta do projeto a título desta demonstração). Monta um loop sobre o array de linhas do arquivo, identifica se o grafo possui arestas com peso ou sem peso e preenche as variáveis globais *quantidade_vertices* e *arestas* para consumo pelos demais métodos (Figura 4).

```
1   from collections import deque
2   from queue import PriorityQueue
3   from math import inf
4
5   class Grafo:
6
7       def __init__(self, quantidade_vertices, arestas, ponderado):
8           self.quantidade_vertices = quantidade_vertices
9           self.arestas = arestas
10          self.ponderado = ponderado
11
12      def __str__(self):
13          return "Grafo\nQuantidade de vértices: {}\nArestas: {}".format(self.
14              quantidade_vertices, self.arestas)
15
```

Figura 4 - Variáveis globais da classe *Grafo*

2.1.2. Método *gera_arquivo()*

```
37      def gera_arquivo(self):
38
39          arquivo = open("saida.txt", "w")
40          arquivo.write("# n = {}".format(self.quantidade_vertices))
41          arquivo.write("\n# m = {}".format(len(self.arestas)))
42          arquivo.write("\n# d_medio = {}".format(2 * len(self.arestas) / self.
43              quantidade_vertices))
44          vertices_graus = {}
45
46          for aresta in self.arestas:
47              for vertice in aresta:
48                  if vertice not in vertices_graus:
49                      vertices_graus[vertice] = 1
50                  else:
51                      vertices_graus[vertice] += 1
52
53          grau_maximo = 0
54
55          for vertice in vertices_graus:
56              if vertices_graus[vertice] > grau_maximo:
57                  grau_maximo = vertices_graus[vertice]
58
59          graus_quantidade = {}
60          total_graus = 0
61
62          for grau in range(1, grau_maximo + 1):
63              contador = 0
64
65              for vertice in vertices_graus:
66                  if vertices_graus[vertice] == grau:
67                      contador += 1
68                      total_graus += 1
69
70              graus_quantidade[grau] = contador
71
72          for grau in range(1, grau_maximo + 1):
73              arquivo.write("\n{} {}".format(grau, graus_quantidade[grau] /
74                  total_graus))
75
76          arquivo.close()
77
```

Figura 5 - Método *gera_arquivo()*: implementação

A partir do consumo das variáveis globais alimentadas por *le_grafo()*, comporta o algoritmo para definição do conteúdo de saída definido em 2b, como as informações de grau médio e distribuição empírica do grau dos vértices, a partir do uso de contadores. O conteúdo do arquivo gerado nesta demonstração com o uso do arquivo de entrada *grafo_1.txt* é apresentado no Box 1 a seguir.

Box 1- Conteúdo gerado pelo método *gera_arquivo()*

```
# n = 100
# m = 419
# d_medio = 8.38
1 0.0
2 0.24
3 0.09
4 0.09
5 0.04
6 0.05
7 0.05
8 0.06
9 0.01
10 0.02
11 0.02
12 0.01
13 0.04
14 0.05
15 0.09
16 0.02
17 0.03
18 0.01
19 0.01
20 0.02
21 0.01
22 0.02
23 0.0
24 0.01
25 0.01
```

2.1.3. Método *representa_grafo(param1)*

Trata do atendimento ao requisito 2c, que envolve a representação em tela do grafo fornecido na carga. Recebe o parâmetro *param1*, que informa o tipo de representação escolhida pelo usuário ("lista" ou "matriz") e fornece na saída um array com a estrutura demandada. Para isso, o algoritmo aninha dois métodos internos: *lista()* e *matriz()* para o correto tratamento de cada representação do grafo fornecido.

A seguir, a Figura 6 apresenta a implementação do algoritmo e os Boxes 2 e 3 apresentam recortes da saída da lista de adjacência e da matriz de adjacência, respectivamente.

```
83      def representa_grafo(self, estrutura):
84
85          grafo = {}
86
87          def lista():
88              for aresta in self.arestas:
89                  if aresta[0] in grafo.keys():
90                      grafo[aresta[0]].update({aresta[1]: float(self.arestas[
91                          aresta])})
92                  else:
93                      grafo[aresta[0]] = {aresta[1]: float(self.arestas[aresta])}
94
95                  if aresta[1] in grafo.keys():
96                      grafo[aresta[1]].update({aresta[0]: float(self.arestas[
97                          aresta])})
98                  else:
99                      grafo[aresta[1]] = {aresta[0]: float(self.arestas[aresta])}
100
101         def matriz():
102             lista = self.representa_grafo("lista")
103
104             for vertice in lista:
105                 grafo[vertice] = {}
106
107                 for vertice2 in lista:
108                     if vertice == vertice2 or vertice2 not in lista[vertice]:
109                         grafo[vertice][vertice2] = 0
110                     elif vertice2 in lista[vertice]:
111                         grafo[vertice][vertice2] = lista[vertice][vertice2]
112
113         if estrutura == "lista":
114             lista()
115         elif estrutura == "matriz":
116             matriz()
117
118         return(grafo)
119
```

Figura 6 – Método *representa_grafo()*: implementação

Box 2 - Impressão em tela da lista de adjacências do grafo fornecido em *grafo_1.txt*

```
{'22': {'84': 4.0, '87': 7.0, '34': 7.0, '61': 6.0, '98': 14.0, '6': 3.0, '29':
12.0, '21': 15.0, '23': 11.0, '7': 5.0, '67': 14.0}, '84': {'22': 4.0, '87': 5.0,
'61': 11.0, '34': 6.0, '7': 7.0, '98': 2.0, '83': 8.0, '67': 5.0, '29': 8.0,
'85': 15.0, '6': 12.0}, '30': {'37': 15.0, '93': 1.0, '94': 3.0, '31': 2.0, '72':
5.0, '24': 2.0, '23': 10.0, '29': 14.0}, '37': {'30': 15.0, '93': 7.0, '36': 1.0,
'97': 7.0, '72': 4.0, '98': 8.0, '2': 5.0, '23': 10.0, '94': 1.0, '38': 9.0,
'24': 1.0, '29': 7.0}, '10': {'67': 2.0, '26': 8.0, '40': 3.0, '98': 6.0, '74':
15.0, '9': 2.0, '64': 14.0, '89': 2.0, '24': 4.0, '63': 12.0, '8': 4.0, '11':
9.0, '14': 10.0, '91': 5.0, '85': 5.0, '19': 6.0}, '67': {'10': 2.0, '6': 15.0,
'89': 1.0, '24': 14.0, '19': 5.0, '91': 11.0, '7': 14.0, '87': 15.0, '29': 1.0,
'61': 9.0, '84': 5.0, '26': 1.0, '68': 3.0, '8': 1.0, '40': 6.0, '64': 1.0, '34':
14.0, '22': 14.0, '66': 7.0, '74': 15.0, '14': 14.0, '98': 11.0}, '24': {'64':
6.0, '23': 11.0, '25': 15.0, '94': 15.0, '26': 9.0, '29': 12.0, '72': 6.0, '14':
8.0, '67': 14.0, '40': 4.0, '74': 12.0, '93': 15.0, '10': 4.0, '8': 6.0, '89':
13.0, '19': 7.0, '30': 2.0, '91': 1.0, '37': 1.0}, '64': {'24': 6.0, '14': 3.0,
'10': 14.0, '63': 9.0, '89': 10.0, '19': 9.0, '26': 15.0, '74': 2.0, '67': 1.0,
'91': 6.0, '40': 12.0, '8': 6.0, '65': 4.0}, '55': {'82': 1.0, '87': 2.0, '62':
6.0, '59': 8.0, '53': 7.0, '73': 15.0, '56': 3.0, '72': 4.0, '60': 7.0, '15':
13.0, '45': 15.0, '54': 11.0, '6': 11.0, '48': 11.0, '50': 10.0}, ...}
```

Box 3 - Impressão em tela da lista de adjacências do grafo fornecido em *grafo_1.txt*

```
{'22': {'22': 0, '84': 4.0, '30': 0, '37': 0, '10': 0, '67': 14.0, '24': 0, '64':
0, '55': 0, '82': 0, '73': 0, '6': 3.0, '87': 7.0, '48': 0, '53': 0, '47': 0,
'81': 0, '2': 0, '61': 6.0, '57': 0, '100': 0, '7': 5.0, '79': 0, '95': 0, '29':
12.0, '5': 0, '21': 15.0, '68': 0, '62': 0, '50': 0, '26': 0, '34': 7.0, '89': 0,
'98': 14.0, '85': 0, '40': 0, '74': 0, '92': 0, '93': 0, '14': 0, '8': 0, '39':
0, '88': 0, '3': 0, '4': 0, '23': 11.0, '91': 0, '25': 0, '19': 0, '15': 0, '45':
0, '72': 0, '31': 0, '77': 0, '60': 0, '58': 0, '51': 0, '52': 0, '97': 0, '59':
0, '94': 0, '1': 0, '76': 0, '86': 0, '44': 0, '36': 0, '9': 0, '35': 0, '80': 0,
'41': 0, '20': 0, '99': 0, '75': 0, '96': 0, '13': 0, '71': 0, '17': 0, '16': 0,
'11': 0, '12': 0, '65': 0, '66': 0, '18': 0, '90': 0, '56': 0, '33': 0, '32': 0,
'38': 0, '69': 0, '70': 0, '46': 0, '54': 0, '63': 0, '42': 0, '43': 0, '83': 0,
'27': 0, '28': 0, '78': 0, '49': 0}, '84': {'22': 4.0, '84': 0, '30': 0, '37': 0,
'10': 0, '67': 5.0, '24': 0, '64': 0, '55': 0, '82': 0, '73': 0, '6': 12.0, '87':
5.0, '48': 0, '53': 0, '47': 0, '81': 0, '2': 0, '61': 11.0, '57': 0, '100': 0,
'7': 7.0, '79': 0, '95': 0, '29': 8.0, '5': 0, '21': 0, '68': 0, '62': 0, '50':
0, '26': 0, '34': 6.0, '89': 0, '98': 2.0, '85': 15.0, '40': 0, '74': 0, '92': 0,
'93': 0, '14': 0, '8': 0, '39': 0, '88': 0, '3': 0, '4': 0, '23': 0, '91': 0,
'25': 0, '19': 0, '15': 0, '45': 0, '72': 0, '31': 0, '77': 0, '60': 0, '58': 0,
'51': 0, '52': 0, '97': 0, '59': 0, '94': 0, '1': 0, '76': 0, '86': 0, '44': 0,
'36': 0, '9': 0, '35': 0, '80': 0, '41': 0, '20': 0, '99': 0, '75': 0, '96': 0,
'13': 0, '71': 0, '17': 0, '16': 0, '11': 0, '12': 0, '65': 0, '66': 0, '18': 0,
'90': 0, '56': 0, '33': 0, '32': 0, '38': 0, '69': 0, '70': 0, '46': 0, '54': 0,
'63': 0, '42': 0, '43': 0, '83': 8.0, '27': 0, '28': 0, '78': 0, '49': 0}, ...}}
```

2.1.4. Método *busca_grafo(param1, param2)*

Na implementação do método mostrada na Figura 7 é realizada a busca em largura (BFS) ou em profundidade no gráfico da entrada a partir dos parâmetros *param1* e *param2* fornecidos pelo usuário, gerando um arquivo de saída (linhas 175-185) com as informações pedidas no tópico 2d. Conforme o valor de *param1* (“bfs” ou “dfs”), um dos dois métodos internos do algoritmo, respectivamente *bfs()* e *dfs()*, será invocado (linhas 170-173).

```
115     def busca_grafo(self, busca, raiz = None):
116
117         def bfs():
118             fila = deque()
119             fila.append(raiz)
120             visitado = {}
121             pai = {}
122             nivel = {}
123
124             for vertice in grafo:
125                 visitado[vertice] = False
126                 pai[vertice] = None
127
128             visitado[raiz] = True
129             nivel[raiz] = 0
130
131             while fila:
132                 atual = fila.popleft()
133
134                 for vertice in grafo[atual]:
135                     if not visitado[vertice]:
136                         visitado[vertice] = True
137                         pai[vertice] = atual
138                         nivel[vertice] = nivel[atual] + 1
139                         fila.append(vertice)
140
141             return pai, nivel
142
143         def dfs():
144                 visitado = {}
145                 pai = {}
146                 nivel = {}
147
148                 for vertice in grafo:
149                     visitado[vertice] = False
150                     pai[vertice] = None
151
152                 def dfs_visita(grafo, atual):
153                     visitado[atual] = True
154
155                     for vertice in grafo[atual]:
156                         if not visitado[vertice]:
157                             pai[vertice] = atual
158                             nivel[vertice] = nivel[atual] + 1
159                             dfs_visita(grafo, vertice)
160
161                 for vertice in grafo:
162                     if not visitado[vertice]:
163                         nivel[vertice] = 0
164                         dfs_visita(grafo, vertice)
165
166                 return pai, nivel
167
168         grafo = self.representa_grafo("lista")
169
170         if busca == "bfs":
171             pai, nivel = bfs()
172         elif busca == "dfs":
173             pai, nivel = dfs()
174
175         arquivo = open("saida.txt", "w")
176
177         for vertice in grafo:
178             if pai[vertice] is None:
179                 arquivo.write("{}: (pai: -, nível: {})".format(vertice, nivel[
180                     vertice]))
181             else:
182                 arquivo.write("\n{}: (pai: {}, nível: {})".format(vertice, pai[
183                     vertice], nivel[vertice]))
184
185         arquivo.close()
```

Figura 7 – Método *busca_grafo()*: implementação

O método *bfs()* faz uso do deque (linha 118), tipo de fila *FIFO (First In, First Out)* importado de *collections* pela classe (Figura 4) para remoção dos vértices já visitados (linha 132) e inserção dos vizinhos do atual, conforme os descobre (linha 139). Já o método *dfs()* realiza a incursão pelo gráfico utilizando a estratégia recursiva com uso do método interno *dfs_visita()*, que realiza a visita em profundidade para cada ramo subjacente ao vértice atual.

O Box 4 mostra o recorte do resultado da busca DFS sobre o grafo de entrada:

Box 4 – Recorte da saída da busca DFS sobre o grafo fornecido em *grafo_1.txt*

```
22: (pai: -, nível: 0)
84: (pai: 22, nível: 1)
30: (pai: 37, nível: 26)
37: (pai: 97, nível: 25)
10: (pai: 67, nível: 36)
67: (pai: 6, nível: 35)
24: (pai: 26, nível: 38)
64: (pai: 24, nível: 39)
55: (pai: 87, nível: 3)
82: (pai: 55, nível: 4)
73: (pai: 82, nível: 5)
6: (pai: 34, nível: 34)
87: (pai: 84, nível: 2)
48: (pai: 53, nível: 11)
53: (pai: 50, nível: 10)
47: (pai: 81, nível: 15)
81: (pai: 2, nível: 14)
2: (pai: 61, nível: 13)
61: (pai: 48, nível: 12)
57: (pai: 86, nível: 17)
100: (pai: 57, nível: 18)
7: (pai: 58, nível: 54)
79: (pai: 78, nível: 45)
95: (pai: 79, nível: 46)
29: (pai: 28, nível: 58)
5: (pai: 4, nível: 49)
21: (pai: 62, nível: 22)
68: (pai: 21, nível: 23)
62: (pai: 98, nível: 21)
...
49: (pai: 48, nível: 12)
```

2.1.5. Método *descobre_componentes_conexas()*

Para a descoberta dos componentes conectados do grafo de carga, o algoritmo (Figura 8) implementou a busca em profundidade (DFS), através do método interno *dfs_visita()*. Ao final, gera um arquivo de saída conforme definições do tópico 2e (linhas 232 - 243). Abaixo, Box 5 apresenta recorte da saída gerada pelo método. Os componentes são ordenados decrescentemente com a aplicação da função *sorted* (linha 230) sobre o array de componentes conexas gerado.

Box 5 – Recorte da saída do método *descobre_componentes_conexas()*

```
Quantidade de componentes conexas: 1

Componente 1: 100 vértice(s)
 * Vértice 1
 * Vértice 10
 * Vértice 100
 * Vértice 11
 * Vértice 12
 * Vértice 13
 * Vértice 14
 * Vértice 15
 * Vértice 16
 * Vértice 17
 * Vértice 18
 * Vértice 19
 * Vértice 2
 * Vértice 20
 * Vértice 21
 * Vértice 22
 * Vértice 23
 * Vértice 24
 * Vértice 25
 * Vértice 26
 * Vértice 27
 * Vértice 28
 * Vértice 29
 * Vértice 3
 ...
 * Vértice 99
```

```
187     def descobre_componentes_conexas(self):
188
189         grafo = self.representa_grafo("lista")
190         visitado = {}
191         componente = {}
192         contador = 1
193
194         for vertice in grafo:
195             visitado[vertice] = False
196
197         def dfs_visita(grafo, atual):
198             visitado[atual] = True
199
200             for vertice in grafo[atual]:
201                 if not visitado[vertice]:
202                     dfs_visita(grafo, vertice)
203                     componente[vertice] = contador
204
205         for vertice in grafo:
206             if not visitado[vertice]:
207                 componente[vertice] = contador
208                 dfs_visita(grafo, vertice)
209                 contador += 1
210
211         componente_quantidade = {}
212
213         for numero_componente in range(contador - 1, 0, -1):
214             total_vertices = 0
215
216             for vertice in componente:
217                 if componente[vertice] == numero_componente:
218                     total_vertices += 1
219
220             componente_quantidade[numero_componente] = total_vertices
221
222         decrescente = sorted(componente_quantidade.items(), key = lambda x: x[1
223             ], reverse = True)
224         arquivo = open("saida.txt", "w")
225         arquivo.write("Quantidade de componentes conexas: {}".format(len(
226             componente_quantidade)))
227
228         for quantidade in decrescente:
229             arquivo.write("\n\nComponente {}: {} vértice(s)".format(quantidade[
230                 0], quantidade[1]))
231             for vertice in sorted(componente):
232                 if componente[vertice] == quantidade[0]:
233                     arquivo.write("\n * Vértice {}".format(vertice))
234
235         arquivo.close()
```

Figura 8 - Método *descobre_componentes_conexas()*: implementação

3. Segunda Parte

O foco desta segunda etapa está na implementação de funcionalidades de representação e manipulação de grafos com pesos e algoritmo de descoberta do caminho mínimo, baseado no algoritmo de Dijkstra. A demanda é detalhada nos tópicos a) e b) a seguir:

a) **Grafos com pesos:** representar e manipular grafos não-direcionados que possuam pesos nas arestas. Os pesos, que serão representados por valores reais, devem estar associados às arestas. O arquivo de entrada será modificado, tendo agora uma terceira coluna, que representa o peso da aresta (podendo ser qualquer número de ponto flutuante), como mostrado na Figura 2 da Primeira Parte.

b) **Distância e caminho mínimo:** implementar algoritmo para encontrar a distância entre qualquer par de vértices, assim como o caminho que possui esta distância. Se o grafo não possuir pesos, o algoritmo de busca em largura deve ser utilizado. Se o grafo possuir pesos, o algoritmo de Dijkstra deve ser utilizado. Neste último caso, é necessário verificar se os pesos de todas as arestas são maiores ou iguais a zero, condição necessária para que o algoritmo de Dijkstra funcione corretamente. Além de calcular a distância e caminho mínimo entre um par de vértices, o algoritmo deverá calcular a distância e caminho mínimo entre um dado vértice e todos os outros vértices do grafo.

3.1. Implementação

Julgamos que há redundância entre o requisito 3a desta etapa com a implementação elicitada na Primeira Parte, já que nossos experimentos com o grafo representado no arquivo *grafo_1.txt* é ponderado. Assim, apresentaremos a seguir o algoritmo referente à demanda especificada no tópico 3b desta seção.

```
246     def calcula_distancia(self, vertice_origem, vertice_destino = None):
247
248         def dijkstra():
249             grafo = self.representa_grafo("lista")
250             distancia = {}
251             pai = {}
252             fila = PriorityQueue()
253
254             for vertice in grafo:
255                 distancia[vertice] = inf
256                 pai[vertice] = None
257                 fila.put((distancia[vertice], vertice))
258
259             distancia[vertice_origem] = 0
260
261             while not fila.empty():
262                 atual = fila.get()
263
264                 for vertice in grafo[atual[1]]:
265                     alt = round(distancia[atual[1]] + grafo[atual[1]][vertice],
266                         7)
267
268                     if alt < distancia[vertice]:
269                         distancia[vertice] = alt
270                         pai[vertice] = atual[1]
271
272             return pai, distancia
273
274         if self.ponderado:
275             pai, distancia = dijkstra()
276         else:
277             pai, distancia = self.busca_grafo("bfs", vertice_origem)
278
279         caminho = {}
280
281         def calcula_caminho(destino):
282             if not pai[destino]:
283                 return [destino]
284             else:
285                 caminho = calcula_caminho(pai[destino]) + [destino]
286
287             return caminho
288
289         for vertice in pai:
290             caminho[vertice] = calcula_caminho(vertice)
291
292         if not vertice_destino:
293             return distancia, caminho
294         else:
295             return distancia[vertice_destino], caminho[vertice_destino]
```

Figura 9 - Método *calcula_distancia()*: implementação

Para demonstração da implementação desta etapa foram utilizados diferentes arquivo de entrada para cálculo de distância e caminho mínimo, com representações de grafos não-direcionados ponderados e não ponderados. O método

calcula_distancia() da classe *Grafo* implementa o algoritmo responsável pela recepção das informações do grafo fornecido na carga e informações de distância e caminho mínimo, que por sua vez são impressas em tela pelo seu invocador (arquivo *teste.py,* neste projeto). Para isso, a assinatura do método contempla como parâmetros as variáveis globais preenchidas pelo processamento de *le_grafo()* e os vértices fornecidos pelo usuário (linha 246). O terceiro parâmetro do método recebe valor default nulo já considerando o cálculo entre um determinado vértice e todos os outros do grafo (processamentos geradores das saídas exibidas em Box 7 e Box 10, mais adiante). Adicionalmente, o primeiro parâmetro, que encapsula as variáveis globais (Figura 4, linhas 7-10) informa se o gráfico é ou não ponderado com base na saída de *le_grafo()*. Esse dado é fundamental para a escolha do algoritmo mais adequado para percorrer-se o grafo (linhas 274-277) e calcular-se a distância entre os vértices. Caso o grafo seja ponderado, o algoritmo interno *dijkstra()* é invocado, do contrário a varredura será com o uso do algoritmo BFS (tópico 2.1.4).

Para a definição de caminho mínimo entre os vértices, a implementação de *dijkstra()* considera o uso da lista de adjacência gerada por *representa_grafo("lista")*(tópico 2.1.3) e uma fila de prioridade (linha 252) para atualização dos vértices cujo caminho ainda não foi determinado no decorrer da varredura do grafo pelo algoritmo. A biblioteca *queue* é importada pela classe para uso do objeto de *PriorityQueue* no algoritmo (Figura 4).

A implementação do método comporta ainda o algoritmo *calcula_caminho(destino)* para a definição do caminho mínimo com base nos parâmetros de retorno dos métodos invocados no bloco 274-277.

3.1.1. Grafos não ponderados: distância e caminho mínimo

Utilizaremos aqui como entrada o grafo não ponderado representado no Box 6 abaixo, cujas arestas não possuem peso. Os boxes a seguir apresentam o conteúdo do arquivo de entrada (Box 6) e dos arquivos saída para cálculo entre um vértice e todos os outros (Box 7) e também entre dois vértices (Box 8).

Box 6 – Arquivo de entrada com representação de grafo não ponderado com 7 vértices

```
7
A B
A C
A E
B D
B F
C G
E F
```

Box 7 – Saída com informações de distância e caminho mínimo entre o vértice G e todos os outros do grafo não ponderado do Box 6

```
({'G': 0, 'C': 1, 'A': 2, 'B': 3, 'E': 3, 'D': 4, 'F': 4}, {'A': ['G', 'C',
'A'], 'B': ['G', 'C', 'A', 'B'], 'C': ['G', 'C'], 'E': ['G', 'C', 'A', 'E'],
'D': ['G', 'C', 'A', 'B', 'D'], 'F': ['G', 'C', 'A', 'B', 'F'], 'G': ['G']})
```

Box 8 – Saída com informações de distância e caminho mínimo entre os vértice G e D do grafo não ponderado do Box 6

```
(4, ['G', 'C', 'A', 'B', 'D'])
```

3.1.2. Grafos ponderados: distância e caminho mínimo

A representação do grafo ponderado utilizado no arquivo de entrada para este experimento é mostrada no Box 9 abaixo. Os boxes seguintes exibem os conteúdos de saída para cálculo entre um vértice e todos os outros (Box 10) e também entre dois vértices (Box 11).

Box 9 – Arquivo de entrada com representação de grafo ponderado com 5 vértices

```
5
A B 0.1
B E 0.2
E C 5
C D 3
D E 2.3
A E 1
```

Box 10 – Saída com informações de distância e caminho mínimo entre o vértice A e todos os outros do grafo ponderado do Box 9

```
({'A': 0, 'B': 0.1, 'E': 0.3, 'C': 5.3, 'D': 2.6}, {'A': ['A'], 'B': ['A',
'B'], 'E': ['A', 'B', 'E'], 'C': ['A', 'B', 'E', 'C'], 'D': ['A', 'B', 'E',
'D']})
```

Box 11 – Saída com informações de distância e caminho mínimo entre os vértices A e D do grafo ponderado do Box 9

```
(2.6, ['A', 'B', 'E', 'D'])
```