**ALGORITMOS EM GRAFOS**

**TRABALHO PRÁTICO**

Breno Martins

Jéssica Fehnle

Josué Cardoso

Leonardo

Professor: Glauco Amorim

Turma 2017.1

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*.

1. **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:

1. **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).
2. **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).
3. **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.
4. **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.
5. **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.).
   1. **Implementação**

Esta implementação 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.

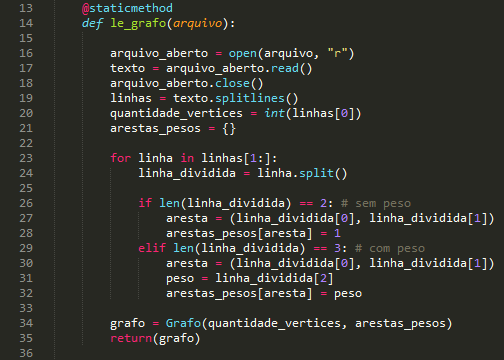
- ***representa\_grafo(****param1****)***: imprime em tela a representação do grafo lido como uma lista de adjacência ou matriz de adjacência, onde *param1* define o tipo de representação entre dois valores possíveis: “lista” ou “matriz”.

- ***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.

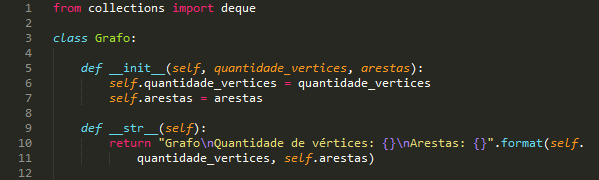
A título de melhor organização desta seção, a dividiremos em seis subseções que abordarão a implementação de cada método individualmente, seguindo a ordem acima definida.

* + 1. **Método *le\_grafo(****param1****)***



**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).



**Figura 4 - Variáveis globais da classe *Grafo***

* + 1. **Método *gera\_arquivo()***



**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 na Tabela 1 a seguir.

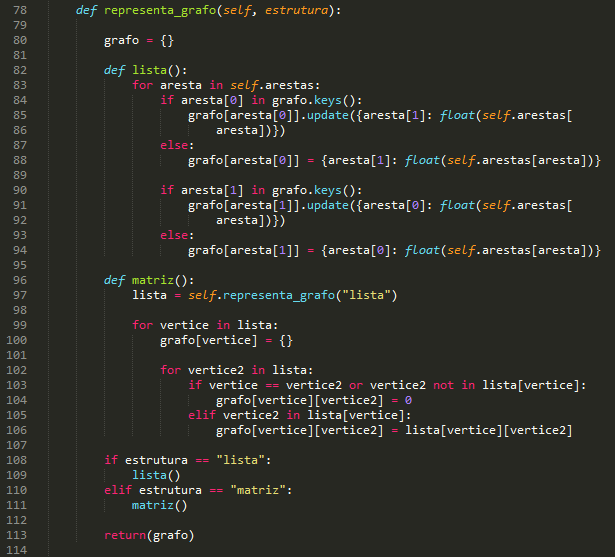
Tabela 1- Conteúdo gerado pelo método *gera\_arquivo()*

|  |
| --- |
| # 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 |

* + 1. **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 as Tabela 2 e 3 apresentam recortes da saída da lista de adjacência e da matriz de adjacência, respectivamente.



**Figura 6 – Método *representa\_grafo()*: implementação**

Tabela 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}, ...} |

Tabela 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}, ...}} |

* + 1. **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).



**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.

A Tabela 4 mostra o recorte do resultado da busca DFS sobre o grafo de entrada:

Tabela 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) |

* + 1. **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, a Tabela 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.

Tabela 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 |



**Figura 8 - Método *descobre\_componentes\_conexas()*: implementação**

1. **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:

1. **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.
2. 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.
   1. **Implementação**

Nononono