

# Teoria dos Grafos – Implementação com Listas de Adjacências

## Atividade Hands-on em Laboratório – 01

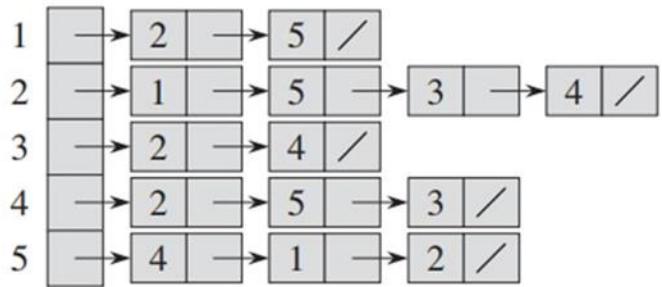
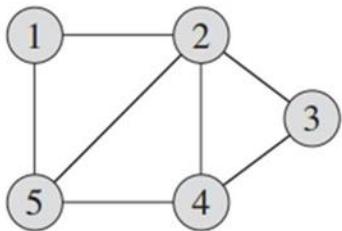
Prof. Calvetti

### 1. Introdução

Um grafo é uma coleção de vértices e arestas. Pode-se modelar a abstração por meio de uma combinação de três tipos de dados: Vértice, Aresta e Grafo. Um vértice (VERTEX) pode ser representado por um objeto que armazena a informação fornecida pelo usuário, por exemplo, informações de um aeroporto. Uma aresta (EDGE) armazena relacionamentos entre vértices, por exemplo: número do voo, distâncias, custos, etc.

A ADT Graph deve incluir diversos métodos para se operar com grafos, podendo lidar com grafos direcionados ou não direcionados. Uma aresta  $(u,v)$  é dita direcionada de  $u$  para  $v$  se o par  $(u,v)$  é ordenado, com  $u$  precedendo  $v$ . Uma aresta  $(u,v)$  é dita não direcionada se o par  $(u,v)$  não for ordenado.

A implementação de grafos por meio de Listas de Adjacências fornece uma forma compacta de se representar grafos esparsos – aqueles para os quais  $|E|$  é muito menor que  $|V|^2$ . Assim, essa implementação é usualmente o método mais escolhido. [Cormen]



A representação de um grafo  $G = (V,E)$  na forma de Listas de Adjacências consiste de um array  $\text{Adj}$  de  $|V|$  listas, uma para cada vértice em  $V$ . Para cada  $u \in V$ , a lista de adjacência  $\text{Adj}[u]$  consiste de todos os vértices  $v$  tais que haja uma aresta  $(u,v)$   $u \in E$ . Ou seja,  $\text{Adj}[u]$  consiste de todos os vértices adjacentes a  $u$  em  $G$ .

Considerando que a lista de adjacências representa as arestas de um grafo, pode-se representar o grafo  $G$  com os atributos:

V: conjunto de vértices de $G$	$\text{Adj}[u]$ : conjunto de arestas de $G$ , para todo $u \in V$ .
--------------------------------	--

Em um grafo  $G$  direcionado, a soma dos nós de todas as listas de adjacências é  $|E|$ . Isso ocorre, uma vez que 1 aresta na forma  $(u,v)$  é representada uma única vez em  $\text{Adj}[u]$ .

Em um grafo  $G$  não-direcionado, a soma dos nós de todas as listas de adjacências é  $2 * |E|$ . Isso ocorre, uma vez que se  $(u,v)$  é uma aresta não-direcionada, então  $u$  aparece em  $\text{Adj}[u]$  e vice-versa.

Para grafos direcionados, a representação em listas de adjacências tem a desejável propriedade que a quantidade de memória necessária é  $\Theta(V + E)$ . [Cormen]

Para grafos não-direcionados, a representação em listas de adjacências tem a desejável propriedade que a quantidade de memória necessária é  $\Theta(V + 2*E)$ . [Cormen]

Pode-se facilmente adaptar-se uma lista de adjacências para representar grafos com pesos. Ou seja, grafos para o qual cada aresta tem um peso associado, tipicamente dado por uma função de pesos:  $w : E \rightarrow R$ .

Por exemplo: seja  $G = (V, E)$  um grafo com pesos com uma função de pesos  $w$ . Pode-se armazenar o valor da função  $w$  para uma aresta  $e \in E$  no nó da lista  $adj[u]$ .

Uma potencial desvantagem da representação por Lista de Adjacências é que ela não provê uma forma rápida de se determinar se uma determinada aresta  $(u, v)$  está presente no grafo. Assim, será necessário pesquisá-la na Lista de Adjacências.

Neste hands-on, iremos fazer a implementação de grafos não-orientados com o emprego de Listas de Adjacências e com a Linguagem Java.

Para a implementação, vamos criar um Projeto Java na IDE chamado Grafo\_AdjList. Em seguida, vamos criar um package denominado br.maua.

## 2. Implementação da Classe Aresta

No package br.maua, iremos implementar a Classe Aresta que irá abstrair as arestas do Grafo a ser implementado. Uma aresta será formada por um vértice de origem e outro de destino.

Vamos também incluir na Classe Aresta o construtor de Arestas.

A classe será definida por:

```
package br.maua;

public class Aresta {

    Vertice origem;
    Vertice destino;

    public Aresta(Vertice origem, Vertice destino) {
        this.origem = origem;
        this.destino = destino;
    }

}
```

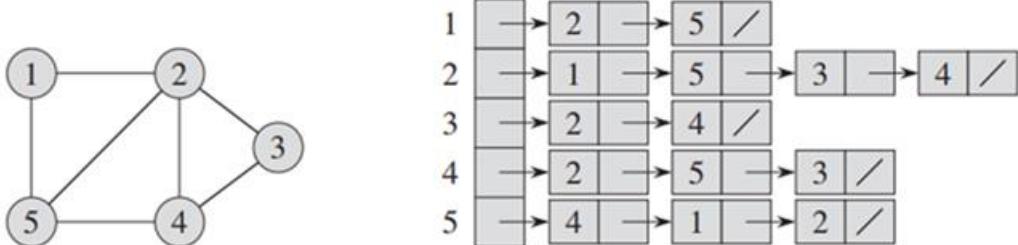
Na implementação da classe Aresta, incluiu-se o construtor de Arestas o qual receberá como parâmetro o vértice de origem e também o vértice de destino.

Estes parâmetros são obrigatórios, uma vez que uma aresta sempre é formada por um par de vértices.

Note que a IDE irá sinalizar erros na definição dos atributos da Aresta, uma vez que ainda não se definiu o objeto Vertice. Após a definição da Classe Vertice, que será feita no próximo item, esses erros serão desmarcados pela IDE.

### 3. Implementação da Classe Vertice

Vamos implementar agora a Classe Vertice. Neste hands-on iremos considerar a implementação com Listas de Adjacências. Portanto, para cada vértice do grafo estaremos associando uma lista de Arestas correspondentes.



Utilizaremos neste hands-on, a implementação de ArrayList do Framework Collections da Plataforma Java. Em cada vértice do grafo, definiremos um atributo que corresponde a informação a ser armazenada no vértice (String dado) e o endereço da Lista de Arestas correspondente.

Assim, utilizaremos a seguinte definição para a Classe Vertice:

```
package br.maua;  
  
import java.util.ArrayList;  
  
public class Vertice {  
  
    String dado;  
    ArrayList<Aresta> listaArestas;  
}
```

Assim, um objeto da classe Vertice será instanciado com os atributos: dado do tipo String e listaArestas do tipo ArrayList<Aresta>.

Considerando que a classe ArrayList não pertence ao package padrão java.lang, será necessário incluir a diretiva import com o nome do package onde a classe ArrayList está implementada, no caso: java.util.

O atributo listaArestas está sendo definido por um ArrayList de Arestas. Caso o vértice não contenha arestas, este atributo estará com valor null.

### 4. Implementação dos construtores da Classe Vertice

Vamos considerar na classe Vertice, dois construtores de Vertice. O primeiro construtor deverá receber como parâmetro o dado String a ser armazenado no vértice e o endereço de uma lista de Arestas

```
public Vertice(String dado, ArrayList<Aresta> listaArestas) {  
    // Complemente o código aqui  
}
```

Complemente o código do construtor, conforme indicado acima. O segundo construtor deverá receber um único parâmetro correspondente ao dado String a ser armazenado no vértice. A lista de arestas correspondentes ao vértice que está sendo instanciado, deverá ser inicializada com null.

```
public Vertice(String dado) {  
    // Complemente o código aqui  
}
```

Complemente o código do construtor, conforme indicado acima.

## 5. Implementação dos métodos da Classe Vertice

Vamos agora implementar mais duas funções na classe Vertice.

A primeira dessas funções será responsável pela inclusão de uma nova aresta incidente ao vértice na Lista de Arestas. Vamos chamar essa função de addAresta(Aresta a);

A implementação será definida por:

```
public void addAresta(Aresta a) {  
    // Complemente o código aqui  
}
```

Complemente o código da função addAresta(Aresta a), conforme indicado acima.

A segunda função será responsável pela remoção de uma aresta incidente ao vértice na Lista de Arestas. Vamos chamar essa função de removeAresta(Aresta a);

A implementação será definida por:

```
public boolean removeAresta(Aresta a) {  
    // Complemente o código aqui  
}
```

Caso a aresta a ser removida -- e passada como parâmetro à função removeAresta(Aresta a) – não existir, a função deverá retornar false. Caso a aresta passada como parâmetro seja removida, a função deve retornar true.

Complemente o código da função removeAresta(Aresta a), conforme indicado acima.

Dica: Considerando que estamos trabalhando nesta implementação com a classe ArrayList do package java.util é importante que o estudante verifique na documentação da API – Java 8, quais os métodos disponíveis para se inserir objetos no ArrayList, bem como para se remover objetos.

Com isso, terminamos a definição da classe Vertice.

## 6. Implementação da Classe Grafo

Vamos agora implementar a classe Grafo. Considerando que estamos trabalhando com listas de adjacências, o grafo terá como atributo uma lista de vértices. Utilizaremos na implementação a classe ArrayList do package java.util.

```
package br.maua;  
  
import java.util.ArrayList;  
  
public class Grafo {  
  
    ArrayList<Vertice> listaVertices;  
}
```

## 7. Implementação dos Construtores da Classe Grafo

Vamos considerar na classe Grafo, dois construtores de Grafo.

O primeiro construtor deverá receber como parâmetro uma lista de vértices

```
public Grafo(ArrayList<Vertice> listaVertices) {  
  
    // Complemente o código aqui  
}
```

Complemente o código do construtor, conforme indicado acima.

O segundo construtor não deverá ter parâmetros. A lista de vértices correspondentes ao grafo que está sendo instanciado, deverá ser inicializada com null.

```
public Grafo() {  
  
    // Complemente o código aqui  
}
```

Complemente o código do construtor, conforme indicado acima.

## 8. Implementação de funções da Classe Grafo

Agora, vamos implementar funções que permitirão operarmos com a estrutura de dados Grafo implementada com listas de adjacência. Complemente os códigos das funções abaixo que farão parte da classe Grafo.

- a) Função addVertice: Adiciona um novo vértice ao grafo

```
public void addVertice(Vertice v) {  
    // Complemente o código aqui  
}
```

- b) Função imprimeArestasDeVertice: Imprime as arestas associadas a um determinado vértice passado como parâmetro. Caso o vértice passado como parâmetro não tenha arestas a ele conectado, a função deverá imprimir a mensagem: “Vértice sem arestas”.

Caso o vértice passado como parâmetro não exista no grafo, a função deverá imprimir a mensagem: “Vértice inexistente no Grafo...”

A função deverá imprimir os dados armazenados em todas as arestas associadas ao vértice passado como parâmetro. A função deverá imprimir para cada aresta associada ao vértice, o dado armazenado no vértice de origem e também o dado armazenado no vértice destino.

```
public void imprimeArestasDeVertice(Vertice v) {  
    // Complemente o código aqui  
}
```

- c) Função imprimeVerticesDoGrafo: Imprime o dado armazenado em cada vértice do grafo.

Caso o grafo não tenha vértice, a função deverá imprimir a mensagem: “Grafo sem vértices...”

```
public void imprimeVerticesDoGrafo() {  
    // Complemente o código aqui  
}
```

- d) Função retornaGrauVertice: Retorna o grau do vértice passado como parâmetro. Caso a lista de arestas do vértice passado como parâmetro seja nula, a função também deverá retornar null.

```
public Integer retornaGrauVertice(Vertice v) {  
    // Complemente o código aqui  
}
```

- e) Função imprimeGrauVertice: Imprime o grau do vértice passado como parâmetro. Caso o vértice não exista no grafo, a função deverá imprimir a mensagem: “Vértice não existente no Grafo...”.

```
public void imprimeGrauVertice(Vertice v) {  
    // Complemente o código aqui  
}
```

- f) Função removeVertice: A função deverá remover do grafo o vértice passado como parâmetro. Caso a função consiga remover o vértice será retornado true. Caso contrário deverá retornar false. Adicionalmente, em caso de retorno true a função deverá enviar a mensagem “Vértice removido com sucesso”. Adicionalmente, caso o vértice passado como parâmetro não exista no Grafo, a função deverá enviar a mensagem: “Vértice inexistente do Grafo...”.

Observação importante: Todas as arestas associadas a outros vértices também deverão ser removidas.

```
public boolean removeVertice(Vertice v) {  
    // Complemente o código aqui  
}
```

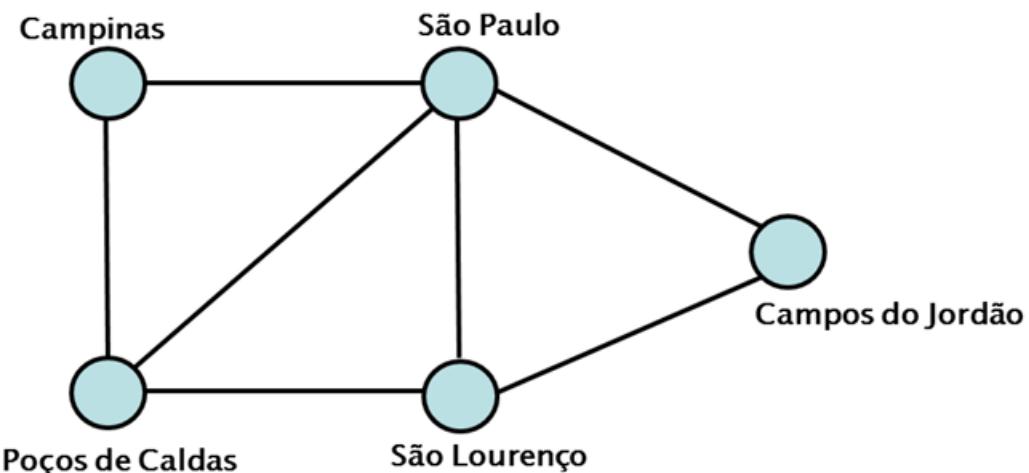
- g) Função imprimeGrafo: A função deverá imprimir a informação armazenada em todos os vértices do grafo e de todas as arestas associadas a cada vértice do grafo.

```
public boolean imprimeGrafo() {  
    // Complemente o código aqui  
}
```

## 9. Implementação da Classe Teste\_Grafo

Vamos agora implementar a classe Teste\_Grafo, com a função main para criarmos um grafo e exercitarmos todas as funções criadas na implementação de Grafos com uma Lista de Adjacência.

Considere o seguinte grafo:



Considere os seguintes Vértices:

- V1: Campinas
- V2: São Paulo
- V3: Campos do Jordão
- V4: São Lourenço
- V5: Poços de Caldas

Complementar a classe Teste\_Grafo conforme esquema abaixo. Criar todos os vértices do Grafo, implementando as arestas na forma de Listas de Adjacências. Exercitar todas as funções definidas nas classes Grafo, Vertice e Aresta.

```
package br.maua;  
  
public class Teste_Grafo {  
  
    public static void main(String[] args) {  
  
        Grafo g = new Grafo();  
  
        Vertice V1 = new Vertice("Campinas");  
  
        // Complemente o código aqui  
  
    }  
}
```