

# Teoria dos Grafos – Implementação com Listas de Adjacências

## Atividade Hands-on em Laboratório – 02

Prof. Calvetti

### 1. Introdução

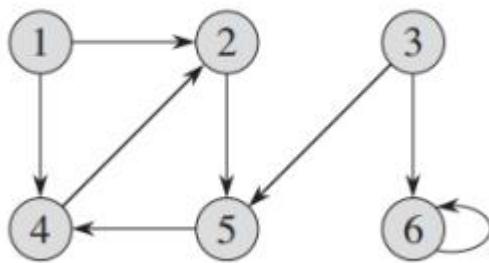
Um grafo é uma coleção de vértices e arestas. Pode-se modelar a abstração por meio de uma combinação de três tipos de dados: Vértice, Aresta e Grafo. Um vértice (VERTEX) pode ser representado por um objeto que armazena a informação fornecida pelo usuário, por exemplo, informações de um aeroporto. Uma 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 Matriz de Adjacências é feita numerando-se os vértices  $1, 2, \dots, |V|$  de alguma maneira arbitrária. A representação de um grafo  $G$ , de acordo com Cormen, consiste de uma matriz  $|V| \times |V| A = (a_{ij})$  tal que:

$$1 \text{ se } (i,j) \in E,$$

$$A_{ij} = 0 \text{ em caso contrário.}$$



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

A matriz de adjacências, de acordo com Cormen, exige a memória  $\Theta(V^2)$ , independentemente do número de arestas do grafo.

Embora a representação de listas de adjacências seja assintoticamente pelo menos tão eficiente quanto a representação de matriz de adjacências, a simplicidade de uma matriz de adjacências pode torná-la preferível quando os grafos são razoavelmente pequenos.

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

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

## 2. Implementação da Classe Grafo

No package br.maua, iremos implementar a Classe Grafo que irá abstrair o grafo por meio de uma matriz de adjacências.

Vamos também incluir na Classe Grafo o construtor de grafos.

O tamanho da matriz é VxV, onde V é o número de vértices no grafo e o valor de uma entrada  $A_{ij}$  é 1 ou 0, dependendo da existência de uma aresta do vértice i ao vértice j.

No caso de grafos não direcionados, a matriz é simétrica em relação à diagonal uma vez que para cada aresta  $(i, j)$ , há também uma aresta  $(j, i)$ .

As operações básicas, como adicionar uma aresta, remover uma aresta e verificar se existe uma aresta do vértice i ao vértice j, são operações de tempo constante extremamente eficientes.

A classe será definida por:

```
package br.maua;  
  
public class Grafo {  
  
    int numVertices;  
  
    boolean [][] adjMatrix;  
}
```

## 3. Implementação do Construtor de Grafo na Classe Grafo

Incluir na classe Grafo, o construtor do grafo que terá como parâmetro o número de vértices do grafo:

```
package br.maua;  
  
public class Grafo {  
  
    int numVertices;  
  
    boolean [][] adjMatrix;  
  
    public Grafo(int numVertices) {  
        // Complemente o código aqui  
    }  
}
```

#### 4. Implementação do método addAresta(i,j) na Classe Grafo

Incluir na classe Grafo, o método addAresta(i,j) para adicionar uma aresta entre dois vértices referenciados por i e j:

```
package br.maua;  
  
public class Grafo {  
  
    int numVertices;  
  
    boolean [][] adjMatrix;  
  
  
    public void addAresta(int i, int j) {  
  
        // Complemente o código aqui  
    }  
}
```

#### 5. Implementação do método removeAresta(i,j) na Classe Grafo

Incluir na classe Grafo, o método removeAresta(i,j) para remover uma aresta entre dois vértices referenciados por i e j:

```
package br.maua;  
  
public class Grafo {  
  
    int numVertices;  
  
    boolean [][] adjMatrix;  
  
    public void removeAresta(int i, int j) {  
  
        // Complemente o código aqui  
    }  
}
```

#### 6. Implementação do método ehAresta(i,j) na Classe Grafo

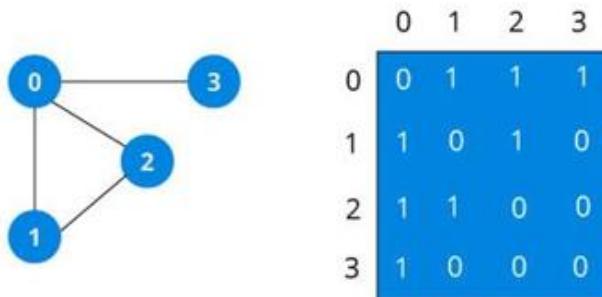
Incluir na classe Grafo, o método ehAresta(i,j) que retorna true se existe aresta entre as referências i e j. Caso contrário, retorna false:

```
package br.maua;  
  
public class Grafo {  
  
    int numVertices;  
  
    boolean [][] adjMatrix;  
  
    public boolean ehAresta(int i, int j) {  
  
        // Complemente o código aqui  
    }  
}
```

## 7. Implementação do método retornaGrafo() na Classe Grafo

Incluir na classe Grafo, o método retornaGrafo() que retorna um String correspondente à matriz de adjacência que representa o grafo.

Por exemplo: Seja o Grafo:



A função retornaGrafo() deverá gerar um String correspondente ao grafo. Ao se executar a função:

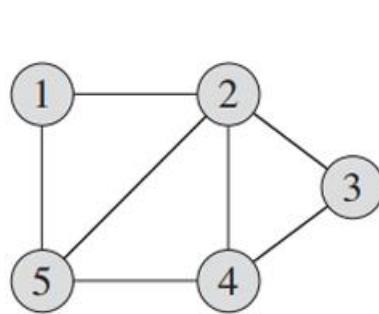
```
System.out.print(g.retornaGrafo());  
  
/* Outputs  
0: 0 1 1 1  
1: 1 0 1 0  
2: 1 1 0 0  
3: 1 0 0 0  
*/
```

```
package br.maua;  
  
public class Grafo {  
  
    int numVertices;  
  
    boolean [][] adjMatrix;  
  
    public String retornaGrafo() {  
        // Complemente o código aqui  
    }  
}
```

## 8. 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 Matriz de Adjacência.

Implementar o grafo e exercitar as funções construídas na classe para o seguinte grafo, obtido de Cormen:



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0