TADS - Teoria dos Grafos
Lab 1 - Implementação de Grafo v 1.2

Prof. Dr. Paulo César Rodacki Gomes - IFC

22 de agosto de 2017

1 Objetivo

O objetivo desta atividade prática em laboratório é implementar classes básicas para manutenção de grafos **não dirigidos**, de acordo com diagrama de classes da figura 1.

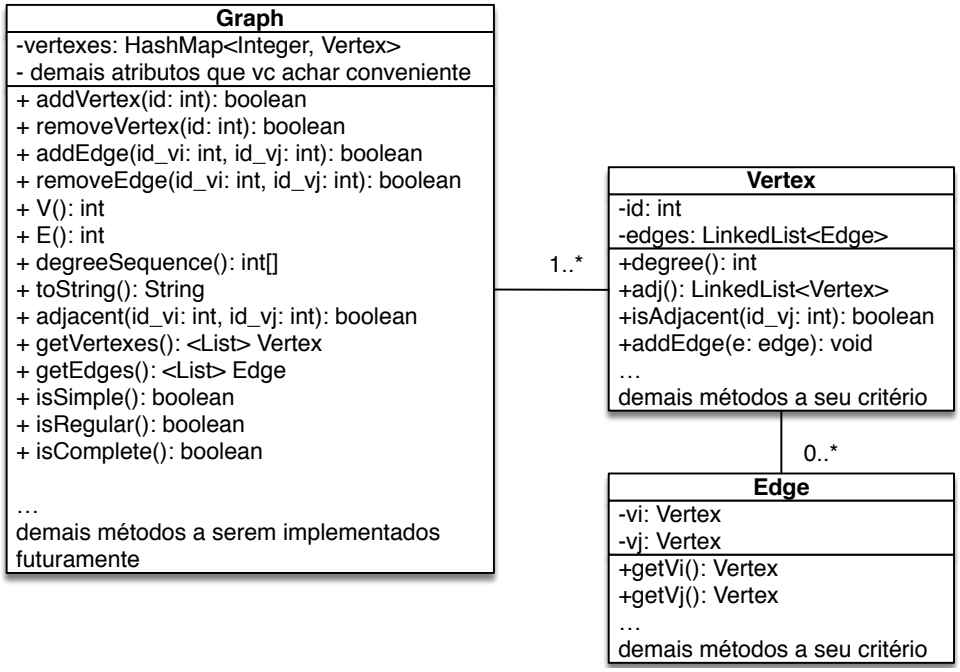

Figura 1: Diagrama de classes para **Grafo não dirigido**

Descrição geral:

A implementação de grafos pode ser feita de várias maneiras diferentes, dependendo dos critérios de projeto mais importantes. Neste exercício estamos propondo a implementação de uma estrutura **dinâmica** para grafos não dirigidos. Obviamente a implementação pode ter diferentes graus de complexidade tanto em termos de métodos oferecidos quanto na questão de design de classes. Aqui, estamos propondo uma estrutura dinâmica, porém simples e enxuta.

A implementação se divide em três classes: *Graph* ,*Vertex* e *Edge*. A classe *Graph* mantém apenas uma estrutura dinâmica contendo todos os vértices, sugerimos o uso de um HashMap java. Cada vértice possui um rótulo (id) numérico que pode ser definido pelo usuário das classes, e uma lista das arestas incidentes no vértice.

Note que não existe lista de arestas na classe grafo, e cada objeto aresta é referenciado duas vezes, uma no vértice v_i e outra no vértice v_j.

Descrição dos principais métodos:

Classe Graph:

1. `public boolean addVertex(int id)`: cria um novo vértice no grafo. O método recebe um id inteiro que será o rótulo do vértice. Deve instanciar um objeto da classe vértice e inclui-lo no grafo (na lista, conjunto ou hash map de vértices). Sugestão: os ids devem ser únicos, portanto sugerimos que você faça a consistência para verificar se o id recebido pelo método já existe no grafo. O método retorna `true` se a operação for realizada com sucesso, e `false` caso contrário.

2. `public poolean removeVertex(int id)`: retira o vértice do grafo. IMPORTANTE: este método deve remover todas as arestas incidentes ao vertice removido. O método retorna `true` se a operação for realizada com sucesso, e `false` caso contrário.

3. `public boolean addEdge(int id_vi, int id_vj)`: cria uma nova Aresta no grafo. O método recebe os 2 ids dos vértices incidentes à aresta. Note que os 2 vértices precisam existir para que a operação seja efetuada com sucesso. O método retorna `true` se a operação for realizada com sucesso, e `false` caso contrário. IMPORTANTE: se a aresta for um laço, ela só deve ser adicionada uma vez à lista de arestas do vértice.

4. `public boolean removeEdge(int id_vi, int id vj)`: retira a aresta do grafo, caso exista. O método retorna `true` se a operação for realizada com sucesso, e `false` caso contrário.

5. `public boolean adjacent(int id_vi, int id_vj)`: retorna `true` se os dois vértices são adjacentes, e `false` caso contrário.

6. `public Collection<Vertex> getVertexes()`: retorna uma coleção contendo todos os Vertices (objetos) do grafo.

7. `public LinkedList <Edge> getEdges()`: retorna uma lista contendo todas as Arestas (objetos) do grafo.

8. `public String toString()`: imprime o grafo de acordo com a organização em listas de adjacência. Portanto, um grafo igual ao da figura 2 seria impresso da seguinte forma:

```
1: 2 3 4 4
2: 1 3
3: 1 2 3
4: 1 1
```

Observação: a ordem dos vértices é arbitrária, portanto **não** é necessário que os *ids* estejam em ordem crescente.

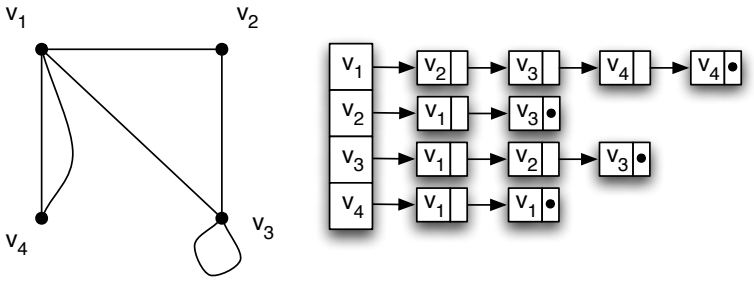

Figura 2: Exemplo de listas de adjacência de grafo não dirigido

9. `public int V()`: retorna a ordem do grafo.

10. `public int E()`: retorna o tamanho do grafo.

11. `public int[] degreeSequence()`: retorna a seqüência de graus do grafo, na forma de um array de inteiros. Obs.: o array deve ser ordenado em ordem crescente.

12. `public boolean isSimple()`: verifica se o grafo é simples ou multigrafo. Retorna `true` se for simples, e `false` caso contrário.

13. `public boolean isRegular()`: retorna `true` se o grafo for regular, e `false` caso contrário.

14. `public boolean isComplete()`: retorna `true` se o grafo for completo, e `false` caso contrário.

Classe Vertex:

1. `public Vertex(int id)`: construtor da classe. Cria um vértice com id inteiro e instancia a lista de arestas incidentes a este vértice (instancia a lista vazia).

2. `public int degree()`: retorna o grau do vértice.

3. `public LinkedList<Vertex> adj()`: retorna a lista de adjacência do vértice, contendo referências para os vértices adjacentes. Obs.: cada aresta incidente ao vértice deve corresponder a um vértice na lista. Por exemplo, se houver arestas paralelas, o vértice na outra extremidade destas arestas deve aparecer mais de uma vez na lista retornada pelo método. Se houver laço, o próprio vértice que executa o método é incluído na lista.

4. `public boolean isAdjacent(int vj)`: retorna verdadeiro se vj é adjacente ao vértice que executa o método. Obs.: lembre-se que este método deve funcionar tambem para laços, quando um vértice é adjacente a si próprio.

5. `public void addEdge(Edge e)`: adiciona a aresta "`e`" à lista de arestas incidentes ao vértice.

Exemplo de função main:

```
    public static void main(String[] args) {
        Graph g = new Graph();

        g.addVertex(1);
        g.addVertex(2);
        g.addVertex(3);
        g.addVertex(4);
        g.addEdge(1, 2);
        g.addEdge(1, 3);
        g.addEdge(1, 4);
        g.addEdge(2, 3);
        g.addEdge(2, 4);
        g.addEdge(3, 4);
        g.addEdge(2, 4);

        System.out.println(g);
        System.out.println("Regular: "   + g.isRegular());
        System.out.println("Simple: "    + g.isSimple());
        System.out.println("Complete: "  + g.isComplete());
        System.out.println("Seq graus: " + Arrays.toString(g.degreeSequence()));
    }
```

Exemplo de saída

```
1: 2 3 4
2: 1 3 4 4
3: 1 2 4
4: 1 2 3 2

Regular: false
Simple: false
Complete: false
Seq graus: [3, 3, 4, 4]
```