Relatório primeira etapa - Trabalho prático de Grafos

Camila Fornaciari Volponi e Rubens Nascimento Correa Junior

Nesta etapa, escolhemos trabalhar com **lista de adjacências**, consideramos ser a forma mais prática para a implementação.

Nosso programa é composto por três classes principais, sendo elas: Grafo, Vértices e Arestas. Também é composto pelas classes Cidade, Leitor e *Main*, aonde na Cidade nos especificamos os elementos principais que cada cidade irá conter, ou seja, o seu código e nome e também foi reescrito as funções *toString()* e *equals()*. E a classe Leitor, auxilia na leitura dos dados solicitados durante o programa.

Cidade.java

grafo > Cidade.java > Cidade

```java
package grafo;

public class Cidade{
    private int codigo;
    private String nome;

    public Cidade(int cod, String nome){
        this.codigo = cod;
        this.nome = nome;
    }

    public int getCodigo() {
        return codigo;
    }

    public String getNome() {
        return nome;
    }

    @Override
    public String toString() {
        // TODO Auto-generated method stub
        return "codigo: " + codigo + "; nome: " + nome;
    }

    @Override
    public boolean equals(Object obj) {
        // TODO Auto-generated method stub
        int cod = ((Cidade) obj).codigo;
        return codigo == cod;
    }
}
```

Figura 01: Classe “Cidade.java”

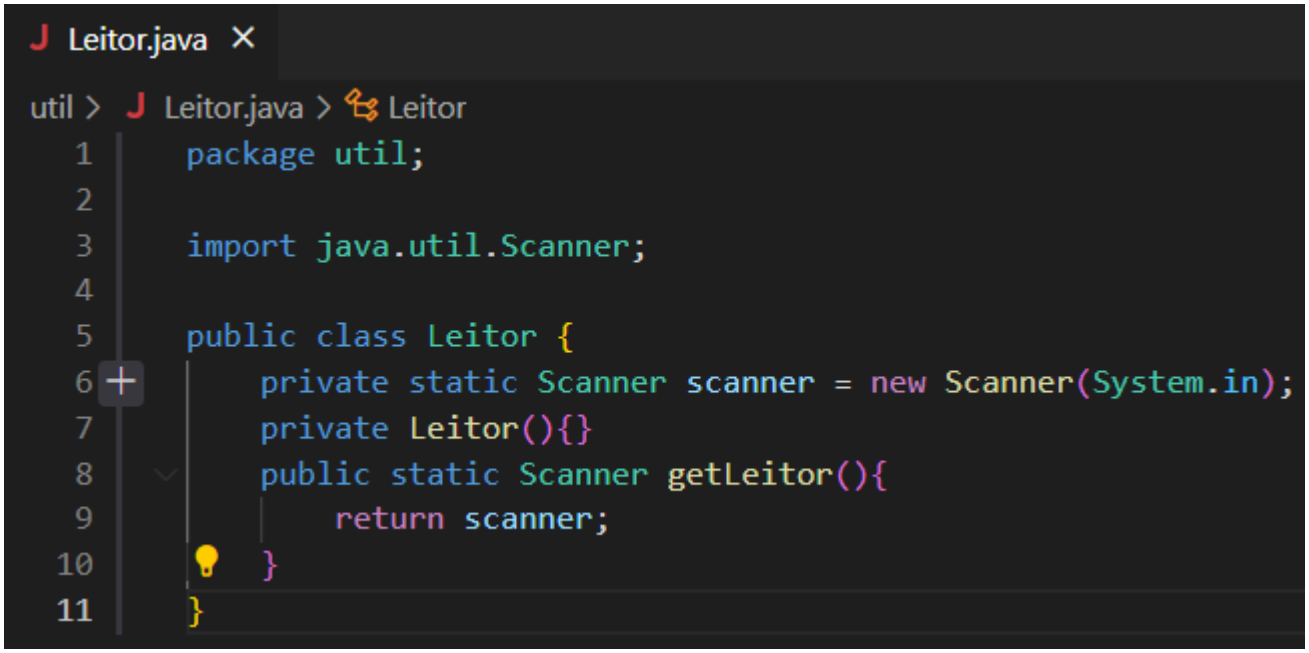

Figura 02: Classe “Leitor.java”

Na classe Vértice, possuímos o objeto “valor” que é a nossa cidade e uma lista de destinos, contendo seus *gets* e *sets* e uma função para adiciona destino:

```
public void adicionarDestino(Aresta<T> aresta){
    this.destinos.add(aresta);
}
```

Figura 03: Função de adicionar destino na lista.

Na classe Aresta, possuímos o peso da aresta e seu destino, onde temos os *gets* e *sets* e reescrevemos a função *toString()*.

```
@Override
public String toString() {
    // TODO Auto-generated method stub
    return destino.getValor() + "; peso: " + this.peso;
}
```

Figura 04: Função reescrita “toString()”

Na classe Grafo, possuímos uma lista de vértices, aonde temos como adicionar vértices, pegar o vértice escolhido e adicionar arestas. Além disso, possuímos também as funções para ser utilizadas no menu, sendo elas: Obter cidades vizinhas e Obter Caminhos.

```java
public void obterCidadesVizinhas(T dado){
    for(Vertice<T> vertice: vertices){
        // Verifica se o vertice atual contém a Cidade igual a que está sendo procurada
        if(vertice.getValor().equals(dado)){
            System.out.println("Cidade escolhida:" + vertice.getValor());
            // Imprime todas as cidades vizinhas a esse cidade
            for(Aresta<T> aresta: vertice.getDestinos()){
                System.out.println(aresta);
            }
        }
    }
}
```

Figura 05: Função para obter cidades vizinhas

```java
public void obterCaminhos(T dado){
    ArrayList<Vertice<T>> marcados = new ArrayList<Vertice<T>>();
    ArrayList<Vertice<T>> fila = new ArrayList<Vertice<T>>();

    Vertice<T> atual = getVertice(dado);
    fila.add(atual);
    //Pego o primeiro vértice como ponto de partida e coloco na fila
    //Poderia escolher qualquer outro...
    //Mas note que dependendo do grafo pode ser que você não caminhe por todos os vétices

    //Enquanto houver vertice na fila...
    while (fila.size()>0){
        //Pego o próximo da fila, marco como visitado e o imprimo
        atual = fila.get(0);
        fila.remove(0);
        marcados.add(atual);
        System.out.println(atual.getValor());
        //Depois pego a lista de adjacencia do nó e se o nó adjacente ainda
        //não tiver sido visitado, o coloco na fila

        ArrayList<Aresta<T>> destinos = atual.getDestinos();
        Vertice<T> proximo;

        for (int i=0; i<destinos.size();i++){
            proximo = destinos.get(i).getDestino();
            if(!marcados.contains(proximo) && !fila.contains(proximo)){
                fila.add(proximo);
            }
        }
    }
}
```

Figura 06: Função para obter caminhos

Na classe Main, possuímos a estrutura de menu e a função que lê o arquivo gerado pelo gerador de arquivos.

```java
public static void lerGrafo(String path, Grafo<Cidade> grafo) throws IOException {
    BufferedReader buffRead = new BufferedReader(new FileReader(path));
    String linha = "";

    int qtdCidades = Integer.parseInt(buffRead.readLine());

    for(int i = 0; i < qtdCidades; i++){
        linha = buffRead.readLine();
        String[] line = linha.split(";");

        int cod = Integer.parseInt(line[0]);
        String cidade = line[1];
        Vertice<Cidade> vertice = new Vertice<Cidade>(new Cidade(cod,cidade));
        grafo.adicionarVertice(vertice);
    }

    for(int i = 1; i < qtdCidades + 1; i++){
        linha = buffRead.readLine();
        if(linha != null){
            String[] line = linha.split(";");

            for(int k = 0; k < qtdCidades; k++){
                float peso = Float.parseFloat((line[k]).replaceAll(",", "."));

                if(peso != 0){
                    Cidade origem = new Cidade(i,nome: "");
                    Cidade destino = new Cidade((k+1),nome: "");
                    grafo.adicionarAresta(peso, origem, destino);
                }
            }
        }
    }
    buffRead.close();
}
```

Figura 07: Função para a leitura do arquivo