Bacharelado em Sistemas de Informação – Noturno

Instituto de Ciências Matemática e de Computação

Algoritmos e Estruturas de Dados II

Relatório do Exercício 5

Prof.: Leonardo Tortoro Pereira

Rodrigo Teixeira Ribeiro da Silva – 11816164
Rafael Jun Morita – 10845040

10/07/2022

Índice

1. Introdução

O exercício 5 da disciplina de Algoritmos e Estruturas de Dados II consiste na realização da travessia do grafo através do algoritmo de Floyd Warshall, e então logo após a realização do cálculo do vértice mais central, o vértice mais periférico, e o vértice mais longe do vértice mais periférico.

Cada um dos vértices do digrafo representará uma cidade em um plano cartesiano, e as arestas as estradas que conectam essas cidades..

O número de cidades, a posição das cidades, o número de estradas, a origem e o destino das estradas serão recebidos através da entrada padrão.

Ao final da execução da solução desenvolvida, a saída padrão deve imprimir as posições do vértice mais central, do vértice mais periférico e do vértice mais distante do vértice mais periférico, respectivamente..

2. Desenvolvimento

Para o desenvolvimento dessa aplicação foi utilizado a biblioteca de grafos desenvolvida ao longo da disciplina.

A classe vértice, que é padrão da biblioteca, é estendida pela classe cidade:

```
public class City extends Vertex {

    private static int idCounter = 0;
    private int id;
    Position pos;
```

que cada vez que é instanciada, recebe um id novo e único, bem como deve ser inicializada com um posição no plano cartesiano (classe Position representa um ponto (X,Y) no plano cartesiano).

Para a execução da rotina principal do programa, a classe "Brabalog" foi criada para abrigar a função main, onde inicialmente é instanciada um Scanner, que será utilizado para receber a entrada padrão.

```
// Inicialização do Scanner que será utilizado para ler a
// entrada padrão.
//
//
Scanner stdin = new Scanner(System.in);
```

Após isso, a propriedade estática "graphController" da classe Brabalog é instanciada.

```
// Instanciação do controlador do gráfo, bem como a estratégia de travessia.
//
//
graphController = new GraphController();
```

Então é recebida o número de cidades que serão recebidas através da entrada da padrão, valor esse será utilizada para inicializar o loop de recebimento dos vértices do grafo.

```
// Solicita a entrada do numero de cidades.
//
//
  System.out.println("Number of cities: ");
int numberOfCities = Integer.parseInt(stdin.nextLine());

// Inicializa as cidades
//
//
  System.out.println("Enter all of the " + numberOfCities + " cities: ");
for (int i = 0; i < numberOfCities; i++) {

    String cityName = "Cidade " + i;

    String[] positions = stdin.nextLine().split(",");
    int x1 = Integer.parseInt(positions[0]);
    int y1 = Integer.parseInt(positions[1]);

    graphController.vertices.add(new City(cityName, x1, y1));
}
```

Após o recebimento de todos os vértices, o grafo do "graphController" é instanciado como uma matriz de dígrafo, recebendo todos os vértices como parâmetro.

```
graphController.g = new DigraphMatrix(graphController.vertices);
```

O mesmo processo utilizado para o recebimento dos vértices é então utilizado para o recebimento das arestas:

```
// Inicializa as avenidas
//
//
  System.out.println("Enter all of the " + numberOfHighways + " highways: ");
for (int i = 0; i < numberOfHighways; i++) {

    String[] stringAux1 = stdin.nextLine().split(","); // str1[0](x1): 0, str1[1]: 0:4, str1[2]: 3(y2)
    String[] stringAux2 = stringAux1[1].split(":"); // str2[0](y1): 0, str2[1](x2): 0:4,

    int x1 = Integer.parseInt(stringAux1[0]);
    int y1 = Integer.parseInt(stringAux2[0]);
    int x2 = Integer.parseInt(stringAux2[1]);
    int y2 = Integer.parseInt(stringAux1[2]);

    Position source = new Position(x1, y1);
    Position destination = new Position(x2, y2);

    int minusX = (x1 - x2) * (x1 - x2);
    int minusY = (y1 - y2) * (y1 - y2);

    float weight = (float) Math.sqrt(minusX + minusY); // Distância euclidiana

    if (cityIndexByPosition(source) != -1) {
        if (cityIndexByPosition(destination) != -1) {
            graphController.g.addEdge(
                    graphController.g.getVertices().get(cityIndexByPosition(source)),
                    graphController.g.getVertices().get(cityIndexByPosition(destination)),
                    weight
            );
        }
    }
}
```

Uma observação importante é que nesse ponto a função "cityIndexByPosition" já é utilizada para retornar o índice da cidade através da posição recebida por parâmetro.

O próximo passo é inicializar a estratégia de travessia com o algoritmo desejado, no caso o utilizado foi o Floyd Warshall.

```
graphController.traversalStrategy = new FloydWarshallTraversal(graphController.g);
```

Após a seleção do método de travessia, é chamada a função de travessia do grafo.

```
// Traverse graph
//
//
graphController.traversalStrategy.traverseGraph(graphController.g.getVertices().get(0));
```

Agora que a travessia já foi realizada através do algoritmo de Floyd Warshall, a matriz de distâncias entre os vértices já foi criada, e será utilizada para o cálculo das respostas.

São instanciadas então as três cidades que serão impressas na saída padrão ao final da execução:

```
City mostCentralCity = new City();
City mostPeripheralCity = new City();
City mostDistanceFromMostPeripherical = new City();
```

Cálculo do vértice mais central

```
public Vertex getCenterMostVertex(float[][] distanceMatrix)
{
    var maxDistanceInCollumn = new float[distanceMatrix.length];
    Arrays.fill(maxDistanceInCollumn, Float.NEGATIVE_INFINITY);
    for (var i = 0; i < distanceMatrix.length; i++)
    {
        for (var j = 0; j < distanceMatrix[0].length; j++)
        {
            if (maxDistanceInCollumn[i] < distanceMatrix[i][j])
            {
                maxDistanceInCollumn[i] = distanceMatrix[i][j];
            }
        }
    }
    int vertexIndex = getMinDistanceIndexInCollumn(maxDistanceInCollumn);
    return vertices.get(vertexIndex);
}
```

Para o cálculo do vértice mais central, a função recebe a matriz de distância, e a percorre para cada valor, registrando a maior distância em cada uma das colunas.

Ao final, é calculado o índice do vértice que possui a menor distância nesse novo array criado, e então a função "getCenterMostVertex" retorna o vértice mais central.

Cálculo do vértice mais periférico

```
public Vertex getPerifericalMostVertex(float[][] distanceMatrix)
{
    var maxDistanceInCollumn = new float[distanceMatrix.length];
    Arrays.fill(maxDistanceInCollumn, Float.NEGATIVE_INFINITY);
    for (var i = 0; i < distanceMatrix.length; i++)
    {
        for (var j = 0; j < distanceMatrix[0].length; j++)
        {
            if (maxDistanceInCollumn[i] < distanceMatrix[i][j])
            {
                maxDistanceInCollumn[i] = distanceMatrix[i][j];
            }
        }
    }
    int vertexIndex = getMaxDistanceIndexInCollumn(maxDistanceInCollumn);
    return vertices.get(vertexIndex);
}
```

O cálculo do vértice mais periférico segue a mesma lógica do vértice mais central, porém, ao invés do menor valor do array maxDistanceIn Column criado, dessa vez queremos o maior valor.

Cálculo do vértice mais distante do vértice mais periférico

```
public Vertex getMostDistanceFromPerifericalMostVertex (float[][] distanceMatrix, int rowIndex)
{
    var maxDistanceInCollumn = new float[distanceMatrix.length];
    Arrays.fill(maxDistanceInCollumn, Float.NEGATIVE_INFINITY);
    for (var j = 0; j < distanceMatrix.length; j++)
    {
        if (maxDistanceInCollumn[j] < distanceMatrix[rowIndex][j])
        {
            maxDistanceInCollumn[j] = distanceMatrix[rowIndex][j];
        }
    }

    int vertexIndex = getMaxDistanceIndexInCollumn(maxDistanceInCollumn);
    return vertices.get(vertexIndex);
}
```

Por fim, para o cálculo do vértice mais distante do vértice mais periférico, é recebida como parâmetro, além da matriz de distância, o índice da linha na matriz que corresponde ao vértice mais periférico, essa linha é percorrido até encontrar o maior valor, que é utilizado para retornar o vértice em questão.

Ao final da rotina são impressas todas as cidades calculadas:

```
System.out.println("\n");
System.out.println(mostCentralCity.getPosition().toString());
System.out.println(mostPeriphericalCity.getPosition().toString());
System.out.println(mostDistanceFromMostPeripherical.getPosition().toString());
```

3. Resultados

Ao final da execução da rotina criada, foi possível calcular e imprimir o vértice mais central, o mais periférico e o mais distante do mais periférico de qualquer grafo que fosse passado, mostrando assim a importância da utilização do método Floyd Warshall de realização da travessia, que cria a matriz de distâncias utilizada para os principais cálculos do exercício.

Para a entrada do caso de teste 1, a saída esperada era:
6,2
0,0
9,10
e a saída do algoritmo gerada foi:
(6, 2)
(0, 0)
(9, 10)

Para a entrada do caso de teste 2, a saída esperada era:
8,8
15,15
13,0
e a saída do algoritmo gerada foi:
(8, 8)
(15, 15)
(13, 0)

Obs.: A diferença na saída é causada apenas por um método toString() diferente para a classe Position.

4. Referências

- Repositório da biblioteca de grafos da matéria
- Repositório do código da entrega