Data Estructures and Algorithms

Practice 3: Greedy algorithms on graphs

2021-2022

1. Introduction

Practice objectives:

- Review and apply the concepts of greedy algorithms (in graphs)
- Implement a greedy algorithm by applying its resolution scheme
- Solve a real problem about connecting a set of objects together in the most "economical" way possible, or minimum spanning tree problem
- Represent and obtain optimal solutions to this type of problems

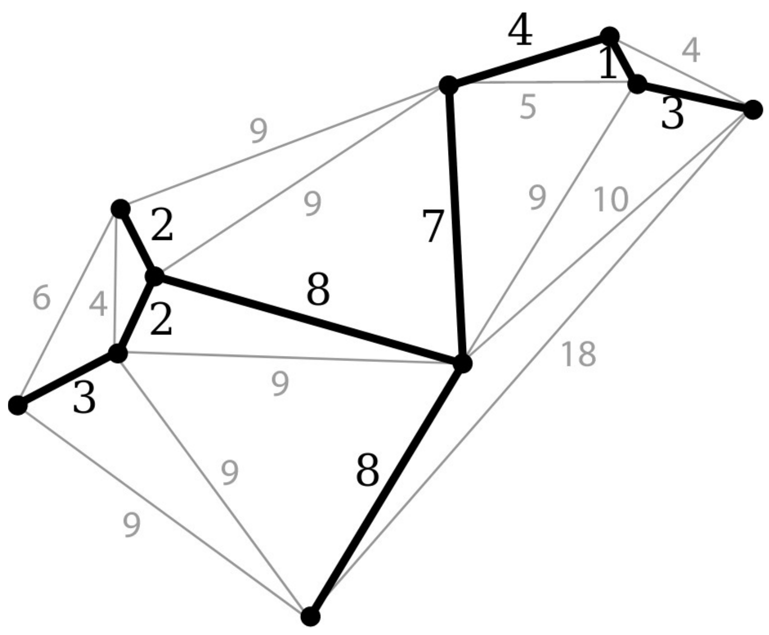
2. Minimum spanning tree

Given a connected weighted undirected graph, a minimum spanning tree or minimum weight spanning tree is a subgraph in a form of a tree that contains all the vertices of the initial graph with the minimun total weitgh of its edges.

For example, if an electric company needs to install power lines between a group of cities at the lowest possible cost, it should install the wiring between the cities that are part of the minimum spanning tree. This would require considering the cities as the vertices of a graph whose edge weights would correspond to the cost of connecting them to each other.

The minimum spanning tree allows to reach all the vertices of the network with a minimum cost.

In the following graph a minimun span tree has a cost of 38.



2.1. Kruskal Algorithm

Kruskal's algorithm allows finding a minimum covering tree in an undirected, connected and weighted graph. That is, it searches for a subset of edges that, forming a tree, include all the vertices and where the value of the sum of all the edges of the tree is the minimum. If the network is not connected, then it searches for a minimum expanded forest.

Kruskal's algorithm is an example of a greedy algorithm that works as follows:

- A forest (a set of trees) is created, where each vertex of the graph is a separate tree.
- A set containing all the edges of the network is created.
- As long as the set is not empty:
  - The edge with the minimum weight edge is eliminated from the set.
  - If the chosen edge connects two different trees it is added to the forest, combining the two trees into one.
  - Otherwise, the edge is discarded.

At the end, the forest has only one component, which forms a minimum spanning tree of the graph. The number of edges of the tree will be the number of nodes minus one.

Kruskal's algorithm pseudocode:

```
función Kruskal(G)
    T ← Ø

    foreach v en V[G] do
        New set C(v) <- {v}

    New MinHeap Q with G edges ordered by weight.
    // n es el número total de vértices

    while !Q.empty() do
        (u,v) ← Q.removeMin()

        // To prevent cycles in T: add (u,v) if u y v are in
         //different sets
        // C(u) returns the set where u belongs.

        si C(u) != C(v) hacer
            Add edge (u,v) to T
            Merge C(v) y C(u)

    return T
```

3. Kruskal's algorithm Implementation

To implement this algorithm it will be necessary to create the following classes:

- `Vertex`
- `Edge`
- `Graph`
- `MinHeap`
- `Main`
- `Canvas` (already implemented)

3.1. Vertex Class

Stores the X and Y coordinates of a vertex.

```
public class Vertex
{
    public final int x, y;
    ...
}
```

3.2. Edge Class

Consists of two vertices U (origin) and V (destination), and the weight W.

```
public class Edge implements Comparable<Edge>
{
    public Vertex getU() { ... } // Returns the origin vertex
    public Vertex getV() { ... } // Returns the destination vertex
    public int    getW() { ... } // Returns the edge weight
    ...
}
```

The Edge class should be comparable taking into account only the weight to allow to add the edges to the MinHeap and get them from lowest to highest weight.

3.3. Graph Class

Represent the graph in such a way that the set of vertices and the set of edges can be obtained.

```
public class Graph
{
    // Adds an egde to the graph between the
    // vertex (ux,uy) y (vx,vy) with weight 'w'.
    public void add(int ux, int uy, int vx, int vy, int w) { ... }

    public Set<Vertex> vertices() { ... }
    public Set<Edge>   edges()    { ... }
}
```

In the implementation of the Graph class it must be taken into account to avoid storing repeated vertices and edges:

- The same vertex will be added as many times as edges contain it. To avoid repeating vertices you can use a hash table with the Vertex type for both keys and values so that before adding a new vertex you can check if it exists in the hash table, and if it does, then the vertex in the table must be used.
- It is necessary to avoid adding repeated edges with the same origin and destination as well as with the same exchanged ends (since it is an undirected graph). If a repeated edge is detected, an IllegalArgumentException must be thrown informing about the repeated edge.

It is important that the vertices and edges functions return immutable sets so that the network cannot be accidentally corrupted from outside its implementation. The following utility can be used for this purpose:

```
Collections.unmodifiableSet(Set<? extends T> s);
```

3.4. MinHeap Class

This class will be necessary to store the edges of the graph and to obtain at each step the edge with the lowest weight.

```
public class MinHeap<E extends Comparable<E>>
{
    private E[] array;
    private int size;

    @SuppressWarnings("unchecked")
    public MinHeap(int capacity)
    {
        array = (E[])new Comparable[capacity];
    }
    ...
}
```

3.5. Main Class

The Main class will be executable and is where Kruskal's algorithm will be implemented.

The main function will accept as parameter a file containing the definition of a graph in which each line will define an edge with the following format:

```
ux uy vx vy weight
```

Where:

- **`ux`** : *X coordinate* of the origin of the edge
- **`uy`** : Y *coordinate* of the origin of the edge
- **`vx`** : *X coordinate* of the destination of the edge
- **`vy`** : Y *coordinate* of the destination of the edge
- **`weight`** : edge weight

The reading of the file can be done as explained in point 6 of the practice 1. The following must be done for each line:

- Create a StringTokenizer object and read token by token the values of the edge.
- Each token can be converted to integer with the function:

```
    Integer.parseInt(st.nextToken())
```

- If the file is malformed, an IOException must be thrown.

After reading the file, a graph g will have been created, for which the minimum overlapping tree t will have to be calculated, and finally displayed by means of the Canvas class:

```
Canvas.paint(g, t);
```

HINT: in Kruskal's algorithm the object C can be represented as follows:

```
HashMap<Vertex,HashSet<Vertex>> c = new HashMap<>();
```

3.6. Canvas Class

This class is provided implemented to be able to represent graphs graphically. It has only two public functions:

```
public class Canvas extends JPanel
{
    // Dibuja el grafo 'g' en una ventana.
    public static void paint(Graph g) { ... }

    // Dibuja el grafo 'g' con su árbol de recubrimiento mínimo 't'.
    public static void paint(Graph g, Graph t) { ... }
}
```

4. Deliverable

In order to obtain the maximum grade, the following work must be done:

- Implement the classes of point 3, except Canvas.
- Correct implementation of Kruskal's algorithm.
- Document all classes using Javadoc.

For the development of this practice it is allowed to use any Java API class.

The source code (.java files) must be delivered packaged in a single compressed file (.zip, .tgz,...) through the PoliformaT task.

All classes (.java files) must be in the p3 package.

```
package p3;  // ¡p lowercase!
...
```