ISEP
Algorithmics and Advanced programming
May 10th 2017

Tutorial course 5: Minimal Spanning trees (MST)

Objectives

- Getting familiar with MST algorithms.

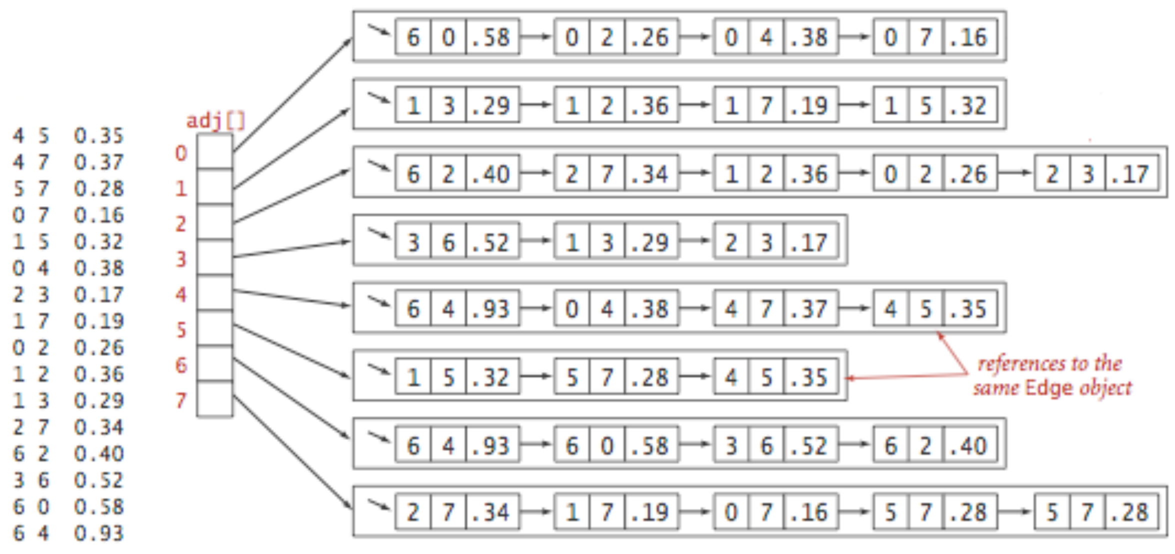
Manipulating undirected weighted graphs

We will consider the adjacency list representation undirected weighted digraphs. As in the previous tutorial course, consider de class called `Edge` but this time for undirected graphs:

```
public class Edge {
    private final int v;
    private final int w;
    private final double weight;

…

public int either() {
        return v;
    }

public int other() {
        return w;
    }

public double weight() {
        return weight;
    }
}
```

where `v` and `w` represent the two end-points of the edge and `weight` represents the edge-weight. The functions **`either`**`()`, **`other`**`()` and **`weight`**`()` constitute the getter functions for the two end-points and the edge-weight respectively.

Create a class called *`EdgeWeightedGraph`* to represent weighted-graphs. This class will use an adjacency list representation of the graph, where the entry corresponding to vertex *`v`* will contain the list of all edges connected to *`v`* and the associated weights. In other words, that will be a list of objects of type `Edge`. As in the following example (taken from http://algs4.cs.princeton.edu/44sp/):



Edge-weighted graph representation

Test the functions of this class on the graph *graph-WG.txt.* The input file lines:

```
4 5 0.35
4 7 0.37
5 7 0.28
…
```

must be interpreted as follows:

- The first line indicates that there is an edge between vertices 4 and 5 and its weight is 0.35

… and so on.

Prim's algorithm

The Prim's algorithm for detecting minimum spanning trees (MST) has been studied in lecture 5. Roughly its steps are the following:

1. Start with a randomly chosen vertex in *T* (the output MST),
2. Add to T an edge of minimal weight with exactly one endpoint in T and the other one outside T.
3. The algorithm stops when N-1 edges have been added to the tree.

Create a class called `PrimMST`. This class will use a priority queue to store all edges of the input graph, ordered by their weight. This leads to an *O(|E| log |E|)* worst-case running time.

The class `PrimMST` will contain at least:

- A function `void prim(EdgeWeightedGraph G, int s)` that will run the algorithm starting from a given vertex `s`.
- A function called `edges()` that will return the edges in a minimum spanning tree.
- A function called `weight()` that will return the weight of the MST.

Test this class in the graph *WG-MST.txt*. What is the length of the MST? What edges are included in the MST? Is there a way to find out if the graph is connected? How?

There is an improved version the Prim's algorithm implementation. It consists in storing vertices instead of edges. The priority queue should order the vertices by the smallest edge-weight that connects them to any vertex in the partially constructed minimum spanning tree (MST) (or infinity if no such edge exists).

Every time a vertex *v* is chosen and added to the MST, the only possible change with respect to each non-tree vertex *w* is that adding *v* brings *w* closer than before to the tree. In short, it is not necessary to keep on the priority queue all of the edges from *w* to vertices tree—we just need to keep track of the minimum-weight edge and check whether the addition of v to the tree necessitates that we update that minimum (because of an edge *v-w* that has lower weight). In other words, we maintain on the priority queue just one edge for each non-tree vertex: the shortest edge that connects it to the tree. This version runs in time $O(|E| \log |V|)$.

Create a new class called `OptPrimMST` containing the same functions as the `PrimMST` class. This new version will use a priority queue for the vertices as described. Test this class with the *WG-MST.txt* graph.

To go further, you can compare the running time of both versions in bigger graphs, taken, for example from the internet, like a facebook, linkedin graphs, etc. For example, some graphs taken from the internet site http://snap.stanford.edu/data/index.html

Kruskal's algorithm

Kruskal's algorithm processes the edges in order of their weight values (smallest to largest), taking for the MST each edge that does not form a cycle with edges previously added, stopping after adding V-1 edges.

To implement Kruskal's algorithm, you will use a priority queue to consider the edges in order by weight, a union-find data structure to identify those that cause cycles.

Create a class called `KruskalMST.` This class will use a priority queue to store all edges of the input graph, ordered by their weight.

The class `KruskalMST` will contain at least:

- A function `void KruskalMST(EdgeWeightedGraph G)` that will run the algorithm.
- A function called `edges()` that will return the edges in a minimum spanning tree.
- A function called `weight()` that will return the weight of the MST.

Test this class in the graph *WG-MST.txt*. What is the length of the MST? What edges are included in the MST? What happens if the graph is disconnected?