# Minimum Spanning Tree Algorithms

**Deadline**: February 8, 2019

In this DM, we will implement Minimum Spanning Tree algorithms over a particular type of graph, in which nodes represent cities on a map, and edges are drawn between nearby cities. Adjacencies are not determined by road connections, but instead by their distance; more precisely, two cities are considered adjacent if their geodesic distance is smaller than a certain maximal value. The weight of each edge is given by the geodesic distance between the pair of cities it connects. You will be asked to implement two classical algorithms, Kruskal's algorithm and Prim's algorithm, in order to find the minimum spanning tree in *each* of the connected component of the graph of cities, minimizing the cost of the tree with respect to the given edge weights. Figure 1 depicts an instance of the problem and its solution.

## 1 Getting Started

You are given a code template in the zip file `dm_3.zip`, which you need to complete. Download and unzip it within a Java project (within the `src` folder of your project, if it has one). This will create three packages called `trees`, `geomap` and `graph`, as well as a `Test` class.

The package `trees` contains the skeletons of two classes which you will have to fill in: class `UnionFind`, and class `SpanningTree`.

The package `geomap` contains two classes `GeoMap` and `Visualization`, which are used to load and visualize a map in a graphical window. These classes are used for testing and their details are not important, **but they should not be modified.**

The package `graph` contains four classes, which you should look at to understand the graph data structure, **but do not modify these files**:

- The class `Place` represents a city on the map by its name and coordinates (latitude, longitude).
- The class `Edge` represents a link between two neighbouring cities.
- The class `EdgeComparator` compares two `Edge` objects according to their length.
- The class `EuclideanGraph` represents a graph with all the cities on a map as vertices and edges between places within a specified upper bound on geodesic distance.

We recommend that you study these classes before writing your program.

We will test our programs on the map of France, which is given to you in the zip file. (Remember to keep the `fr.txt` file in the *working directory* of your project if you are using an IDE, or to have it in the *current directory* if you are running Java from the command line.) If you would like to work on files for other countries, you can find them at this URL.

To test that your installation works, compile and execute `Test.java`, which will execute the function `test0`. You should see on the console a message:

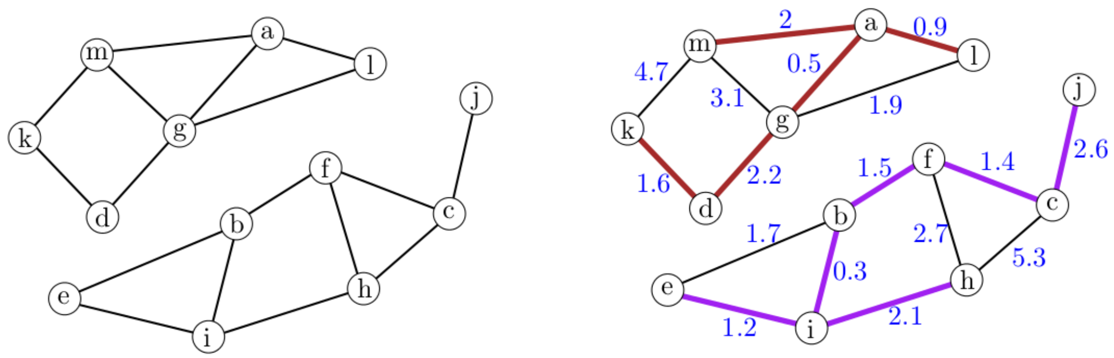

Figure 1: Example of a graph and its minimum spanning tree: Left: a graph with two connected components. Right: the edges of the graph are assigned weights; the edges of the two minimum spanning trees (one for each connected component) are shown with a bold line.

```
There are 167616 edges in this graph.
```

You should also see a new window with a picture that looks something like Figure 2. By zooming in on this picture, you will see red edges between close cities.

## 2 Java Standard Library: Containers

We will use several interfaces and classes from the Java standard library, mostly as containers for storing data structures. You should have an understanding of the intended use and available methods for the following interfaces and classes:

- the Map interface, associating values to objects. We use its LinkedHashMap implementation, based on hash tables;
- the Collection interface, representing finite multisets. It is implemented in particular by LinkedLists;
- the List interface, representing ordered collections. It is implemented in particular by LinkedLists;
- the Set interface, representing finite sets. We do not care about its implementations;
- the Queue interface, representing first-in-first-out collections. It is implemented using LinkedLists and PriorityQueues;
- the Comparator interface, which gives a total ordering between objects of a certain type.

## 3 Tasks

### 3.1 Implementing Union-Find

In order to implement Kruskal's algorithm, we will need a union-find structure for `Edge` objects, which you are to implement in the class `UnionFind.java`. For this, you are already provided a skeleton. The class defines a field `HashMap<Place,Place> parent`. You can call `parent.get(v)` to get the parent of a city (which can be `null`), and you can call `parent.put(v,u)` to set `u` as the parent of `v`. Intuitively, cities with

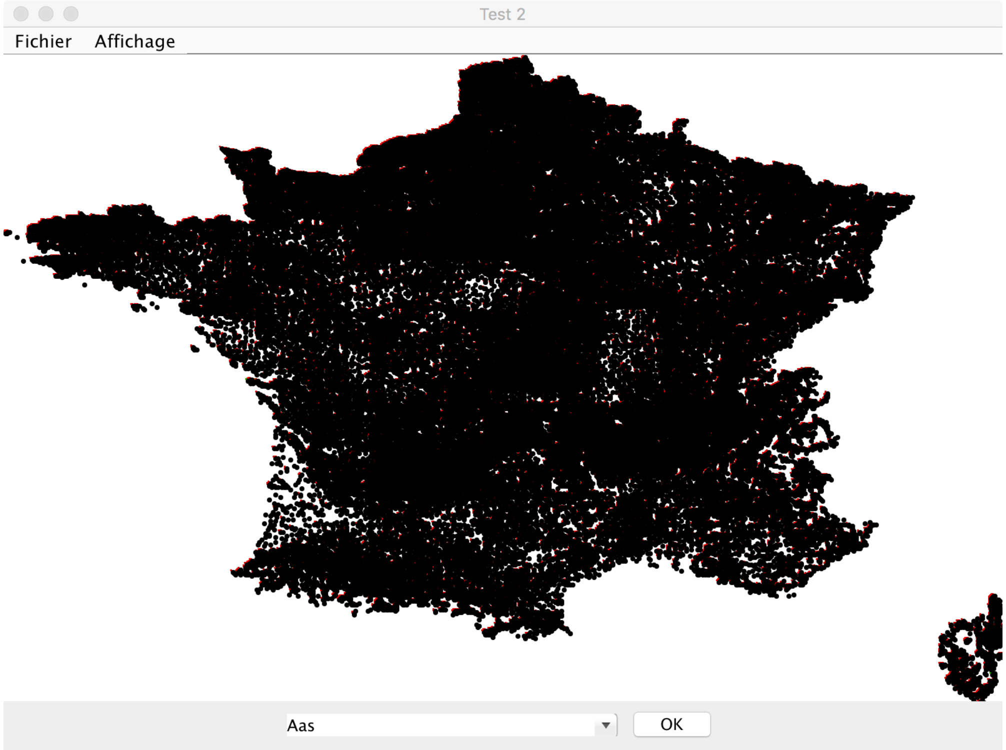

Figure 2: Result of running test0 in Test.java

the same ancestors are in the same set. We treat the oldest (highest) ancestor of a city, which can be found by iteratively following the `parent` map, as the representative of the full set.

Complete the constructor `UnionFind` so that it initializes the `parent` hash map. Then implement the methods `union` and `find` in the class `UnionFind.java`. Recall that `find(v)` should return the representative of the set in which `v` is a member; so it should compute the oldest (highest) ancestor of `v`. Calling `union(v0,v1)` does nothing if `v0` and `v1` are in the same set; otherwise it modifies the `parent` map so that the two sets are merged. Note that when testing for equality between two `Place` objects, you must use the `equals` method on `Place`, **not** the equality operator (==).

**Remark:** when computing the union, you will need to pick one of the representatives of the two sets as the representative of the union. Usually, we would choose the representative of the "larger" set. However, in our examples, the union operations will be applied to edges of a geographic map that are considered in the order given by the geodesic distance; it is reasonable to assume that the components along the way will look quite "random", so that the underlying trees of the union-find structures are expected to be quite well balanced. Therefore you do not have to use size comparison when choosing the representative of the `union`, you can pick either one. However, you should use iterative path compression in the `find` method; that is, every time `find(v)` is called and `v` is several parents away from the set representative, you should modify `parent` to shorten this distance to 1.

Test your code by calling `test1` in `Test.java`. The result in the console will be:

```
true
true
false
```

The implementation of the union-find operation should be such that the asymptotic running time does not exceed $O(\log n)$ amortized time per operation (union or find) on the data structure.

## 3.2 Implementing Kruskal's algorithm: returning a spanning forest

We now move on to the problem of computing minimum spanning trees of a graph $G$. Since $G$ can be made up of several connected components $G_1, \ldots, G_k$ (as with the proximity graph when the maximal value for the distances becomes small), we consider more generally the problem of computing the minimum "connecting spanning forest" of $G$ (a connecting spanning forest of $G$ is a forest of trees $T_1, \ldots, T_k$ such that each $T_i$ is a spanning tree of $G_i$, for each $i \in [1..k]$), as shown in Figure 1.

In this task, you are to implement a first version of Kruskal's algorithm. Here is a quick reminder of how Kruskal's algorithm works:

1. the edges are sorted in increasing order of weight (you can use an instruction as `Collections.sort(col,ac)`, where `col` is of type `Collection<Edge>`, and `ac` is of type `EdgeComparator`),

2. the edges $e_1, \ldots, e_n$ are considered one by one in increasing order, and at each step $i$ one decides if $e_i$ is to be selected: $e_i$ is selected if and only if it satisfies the property that its two extremities are not connected using the previously selected edges (testing this property is handled by the union-find structure).

The output (the collection of selected edges) forms the minimum connecting spanning forest of the graph. Note that this forest will not contain some trivial components which are formed from a single vertex and which are therefore without an edge.

Fill in the method:
`public static Collection<Edge> kruskal(UnionFind u, EuclideanGraph g)`
in the class `SpanningTree.java`, which is to return the collection of the edges that forms the minimum connecting spanning forest. The `UnionFind` argument `u` is initially empty and, at return time, it should contain a description of the non-trivial connected components of g. This side-effect will be useful in the next exercise.

To initially collect all the edges of the `EuclideanGraph g` (which are then to be sorted at the beginning of the method `kruskal`), you can use the method `List<Edge> getAllEdges()` in the class `EuclideanGraph`.

**Remark:** beware that `Collection<E>` (with `E` the type of elements in the collection) is not a class but an interface. Therefore, it has to be instantiated using an implementing class, for instance with an instruction such as `Collection<E> col=new LinkedList<E>()`.

Test your code by running `test2` in `Test.java`. The program should print out on the console something like the following (the computation time may differ depending on device/implementation):

```
Computation time (Kruskal first version) : 613 ms
Total length  : 110191528 m
49390 connected cities
```

and you should see a window as depicted in Figure 3.

## 3.3 Implementing Kruskal's algorithm: separating the components

Using the first `kruskal` method, fill in the method:
`public static Collection<Collection<Edge>> kruskal(EuclideanGraph g)`
which is to return the list of collections of edges (one collection for each non-trivial tree-component) of the trees of the minimum connected spanning forest of g.

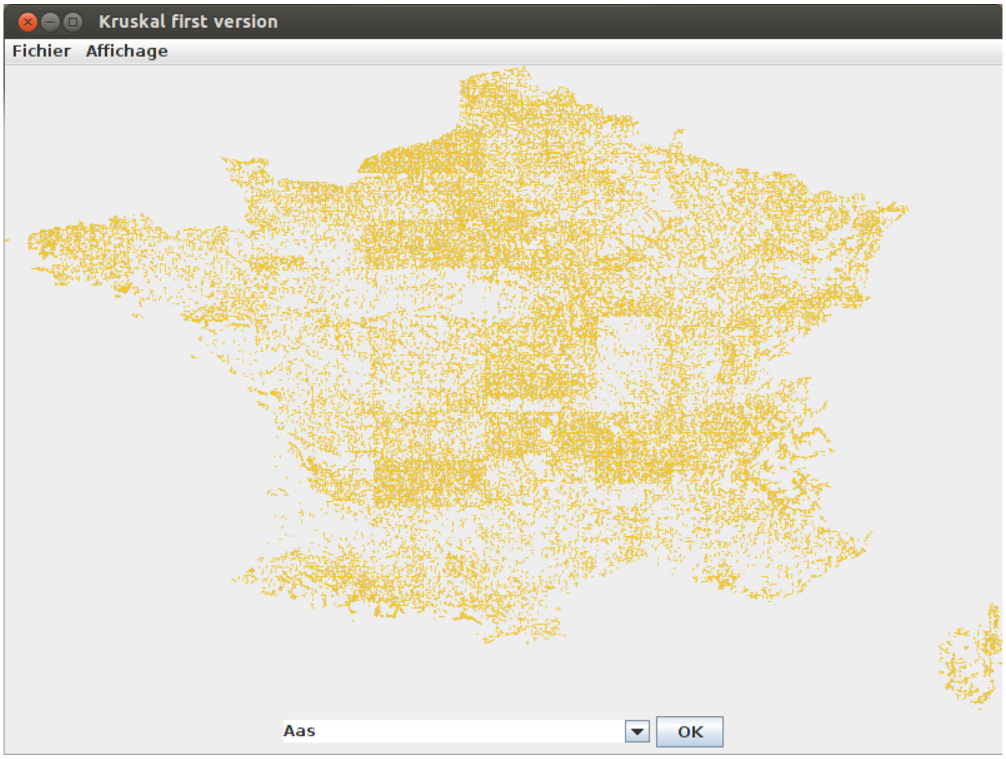

Figure 3: Result of testing Kruskal (test2): the edges in the minimum connecting spanning forest F are colored orange, and edges not in F are light gray

**Remark:** it is recommended to use a `HashMap<Place,Collection<Edge>> edgelist` that maps each root of the UnionFind (a root for each connected component) to the list of edges of the minimum spanning tree of the corresponding connected component; then you can return `edgelist.values()` at the end of the method.

Test your code by running `test3`. You should see the window in Figure 4 and the following text in the console:

```
Computation time (Kruskal second version) : 707 ms
Total length  : 110191528 m
49390 connected cities
3330 components in the forest
```

### 3.4 Implementing Prim's algorithm

First, we recall how Prim's algorithm works. It uses a priority queue `q` (here, implemented using the Java `PriorityQueue` implementation over edges, where higher priority is given to an edge with a smaller distance.) The algorithm starts at a `Place v`, puts all edges connected to `v` into `q`, and removes `v` from the set of non-visited elements. Then as long as `q` is not empty:

1. pick the next `Edge a` from `q` (you can use `q.poll()`),
2. if the target `u` of the edge `a` has not been visited, then add to `q` all the edges connected to `u`, and remove `u` from the set of non-visited elements.

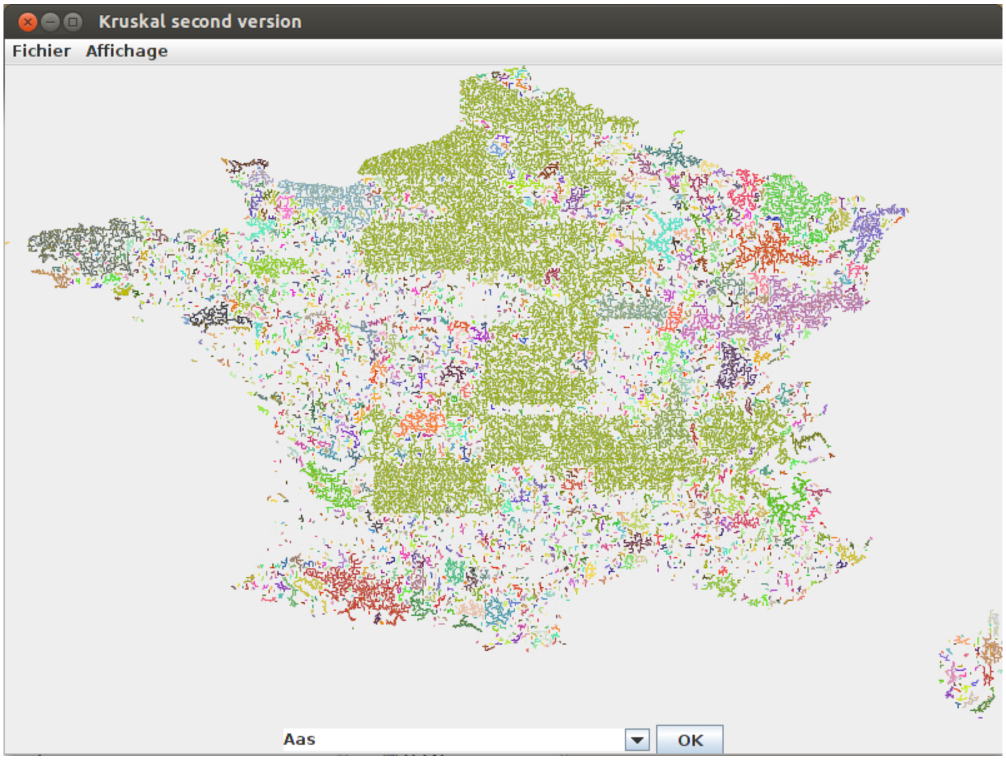

Figure 4: Result of testing Kruskal (test3) with separated tree components: a color is randomly assigned to each connected tree

Note that Prim's algorithm only visits nodes (`Place` objects) that are in the connected component *C* containing `v`; its output is the minimum spanning tree of *C*.

Fill in the method:

```
public static Collection<Edge> primTree(HashSet<Place> nonVisited,
                                       EuclideanGraph g, Place start)
```

which is to return (in the form of a collection of edges) the minimum spanning tree of the connected component *C* containing `start` (in the graph `g`). While computing this tree, the `HashSet` object `nonVisited` is to be modified so that, at the end of the execution, all the `Place` in *C* have been taken out of `nonVisited` (this side effect will be useful for the next exercise).

Test your code with `test4`. You will see the result:

```
Computation time (Prim first version) : 109 ms
Total length:  : 44973288 m
20002 connected cities
```

and a window that looks like Figure 5.

### 3.5 Running Prim's algorithm for all components

Using the `primTree` method, fill in the method:
`public static Collection<Collection<Edge>> primForest(EuclideanGraph g)`

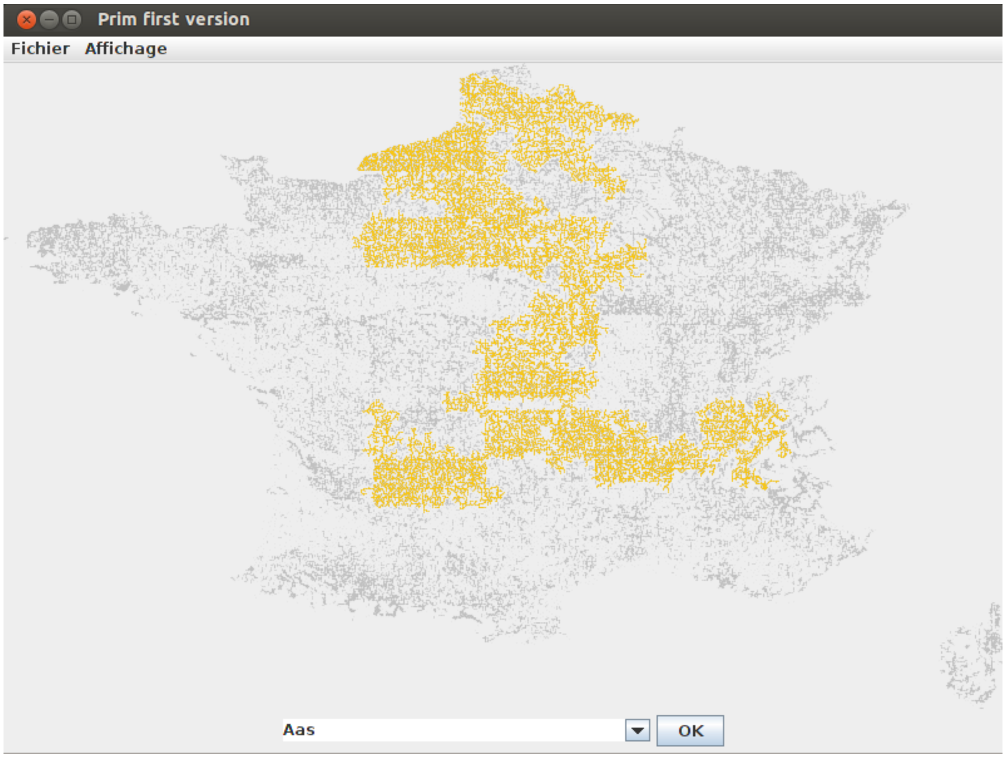

Figure 5: Result of testing Prim for a single component (test4): the edges in the minimum spanning tree of the component containing ”Palaiseau” are colored orange, and edges not in this component are light gray

which is to return the list of collections of edges (one collection for each tree-component) of the minimum connecting spanning forest of g. Again, note that this forest should not contain the trivial components which are formed from a single vertex and empty sub-collections are not allowed in the ouput of `primForest`.

**Remark:** You may find it useful to write an auxiliary method that can peek (without removing) one element from a collection or a set.

Test your code with `test5`. You should see the window in Figure 6 and the following result in the console:

```
Computation time (Prim second version) : 506 ms
Total length  : 110191528 m
49390 connected cities
3330 components in the forest
```

## 4 Scoring

You should submit a zip file containing only two files, named `UnionFind.java` and `SpanningTree.java`, and these file should only have one class each, called `UnionFind` and `SpanningTree`, respectively (with the right capitalization). Otherwise, the automated grading system will reject your homework.

Notice that in the file `Test.java`, you are given five simple tests, one for each of the five tasks, but you should systematically test your code with more examples. The test suite used to compute your programme's final grade is not accessible in advance, but the only graph used in testing will be `fr.txt`.

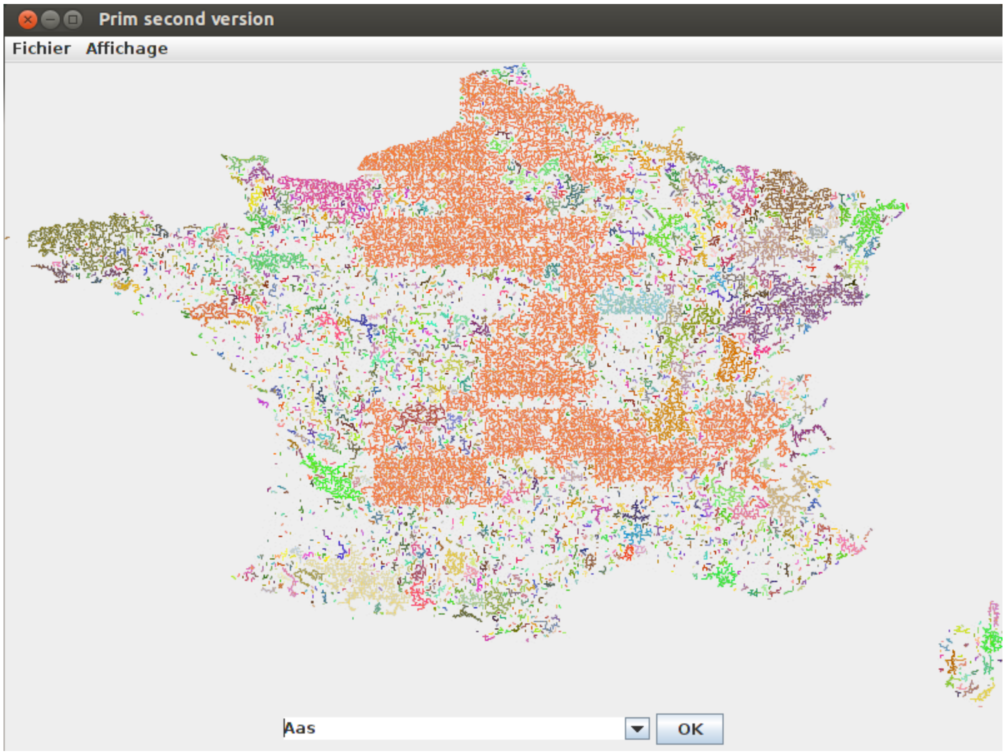

Figure 6: Result of testing Prim for all components (test5): edges in the minimum connecting spanning forest are colored (a color is randomly assigned to each tree-component), and edges not in the forest are light gray

Each of the five tasks is equally valuable (though clearly not all can be performed independently of the others). For solving the three tasks related to union-find and Kruskal's algorithm, you can receive a total of 12 points. For solving the two tasks related to Prim's algorithm, you can receive a total of 8 points. Overall, the maximum score is 20 points.