CMSC 420: Spring 2022

Programming Assignment 3: Euclidean Minimum Spanning Trees

Handed out: Thu, Apr 28. Due: TBD.

Overview: The principal purpose of this assignment is to combine some of the data structures we have seen to compute a fundamental geometric structure, called the *Euclidean minimum spanning tree* (EMST).

Let us first recall the graph-based minimum spanning tree. Given a connected, undirected graph $G = (V, E)$ with vertex set V and edge set E, assume that each edge $(u, v) \in E$ has an associated weight $w(u, v)$. A *spanning tree* is a subset of edges of E so that the subgraph of G induced by these edges is connected, acyclic, and includes all the vertices of G. The *weight* of the spanning tree is the sum of its edge weights. The *minimum spanning tree* (MST) is the spanning tree of minimum weight. (If there are multiple spanning trees of the same minimum weight, any of them is a valid answer to the MST problem.)

We can define the MST problem in a geometric context. We are given a set $P = \{p_1, \dots, p_n\}$ in $\mathbb{R}^2$. This set implicitly defines a graph, called the *Euclidean graph*, whose vertex set is P, whose edges consist of all (unordered) pairs (p_i, p_j), and where the weight of an edge is the Euclidean distance between these points. Note that $|E| = \binom{n}{2}$, so G has $O(n^2)$ size. The *Euclidean minimum spanning tree* (EMST) is the MST of P's Euclidean graph (see Fig. 1).

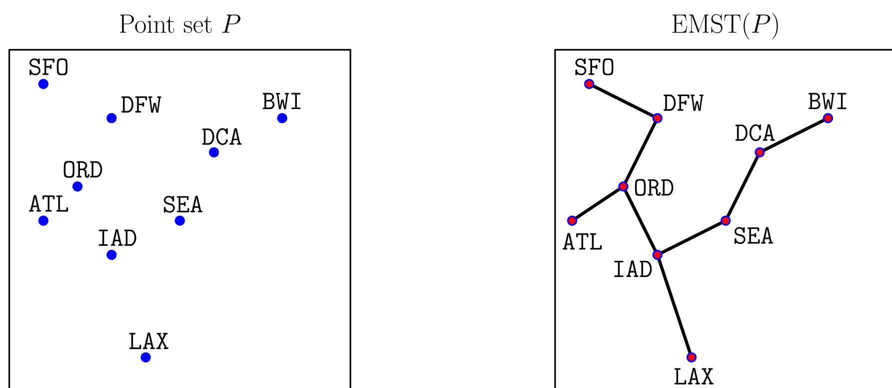

Figure 1: Euclidean minimum spanning tree.

In this assignment, you will implement a class `EMSTree`, whose principal purpose is to compute the EMST of a set of points in the plane efficiently. This class with provide functions for adding points to the set, clearing the set, and computing the EMST of the set. As in our previous assignment, it is templated with the point type, which as in the previous assignment will be labeled point:

```
public class EMSTree<LPoint extends LabeledPoint2D>
```

The EMST will be computed by a geometric variant of Prim's MST algorithm. In order to achieve efficiency, your program will need to implement a *kd-tree* (e.g., your `HBkdTree`

from Programming Assignment 2) augmented with a function for answering *nearest-neighbor queries* and a *priority queue* (e.g., your `QuakeHeap` from Programming Assignment 1).

Points and Distances: As in the previous assignment, the points in our assignment will be any class that implements the `LabeledPoint2D` interface, such as the `Airport` class.

The EMST has the same structure whether defined in terms of Euclidean distances or squared Euclidean distances.[1] By avoiding square roots, it will be both more efficient and more accurate. Letting $p_i = (x_i, y_i)$, the weight of the edge between them will be the *squared Euclidean distance* defined as

$$d(p_i, p_j) \;=\; (x_i - x_j)^2 + (y_i - y_j)^2.$$

To assist you, we have added a member function `double distanceSq(Point2D q)` to the class `Point2D`, which computes the squared distance between the current point and another point `q`. For example, the squared distance between points `p` and `q` can be computed as `p.distanceSq(q)`. This function has the feature that if the argument is `null`, it returns `Double.POSITIVE_INFINITY`.)

Prim's Algorithm: Prim's MST algorithm is given a starting vertex s_0, and builds the spanning tree by repeatedly adding the point that lies outside the tree, but is closest to some point of the tree. A new point is added with each iteration. Let S denote the set of points that are currently in the spanning tree (see the shaded region in Fig. 2). Initially $S = \{s_0\}$ and the algorithm terminates when all the points of P are in S.

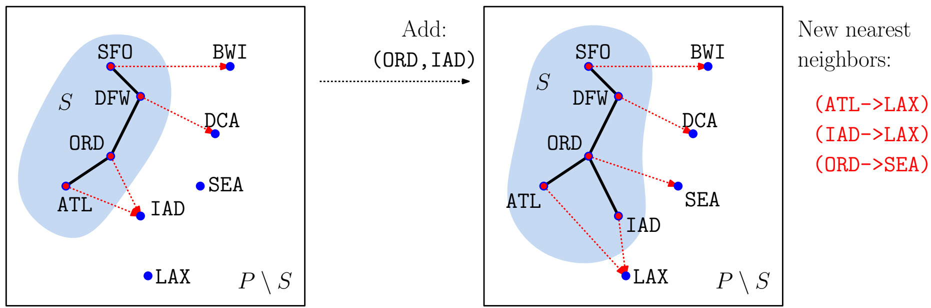

Figure 2: Originally, $S = \{$`SFO`, `DFW`, `ORD`, `ATL`$\}$ with the nearest-neighbor pairs `(SFO,BWI)`, `(DFW,DCA)`,`(ORD,IAD)`, and `(ATL,IAD)`. The closest of these, `(ORD,IAD)` is added to the EMST, `IAD` is added to S, and new nearest neighbors `(ATL,IAD)`, `(ORD,IAD)`, and `(IAD,SEA)` are computed.

Each point $p_i \in S$ computes its nearest point in the complement set $P \setminus S$ (indicated by red broken lines in the figure). Let's call these the *nearest neighbor pairs*. Let (p_i, p_j) be the closest of all the nearest neighbors. In the next iteration, this edge is added to the spanning tree, p_j is added to S, and we need to update the nearest neighbors. This will certainly include p_i, p_j, and any it will also include any other points of S whose nearest neighbor was p_j (see Fig. 2). The process is repeated $n - 1$ times, after which all the points have been added to the spanning tree.

Implementing this algorithm efficiently will involve a number of data structures.

[1] This fact is by no means obvious. In fact it holds for any strictly monotone function of distance.

List: to store the edges of the spanning tree. This can be implemented using a Java `ArrayList` or `LinkedList`. Since Java has no primitive type for storing a pair, we will provide you with a simple generic class `Pair` in the skeleton code. You can create a new pair (`new Pair(a,b)`), access its components (`getFirst()`, `getSecond()`), and test equality (`pair1.equals(pair2)`).

Set: to maintain the points of S. This must support the operations `insert` and `contains` (to test membership). This can be done using a Java `HashSet`.

Spatial index: to store the points of $P \setminus S$ waiting to be inserted into the EMST. This must support the operations of `insert`, `delete`, and `nearestNeighbor`. In a nearest-neighbor query, we are given a query point q, and the answer is the closest point in the tree to q. This can be done using your `HBkdTree` from Programming Assignment 2, and adding a nearest-neighbor function. (See Lecture 14 latex notes for details on how to do this.)

Priority Queue: to store the nearest-neighbor pairs ordered by their distance. Each entry stores the associated pair of points (e.g., from Fig. 2, `(SFO, BWI)` is one of these pairs, and the associated key is the squared distance d(`SFO`,`BWI`).) This can be implemented using your `QuakeHeap` data structure from Programming Assignment 1.

Initially, all the points except the start point s_0 are inserted into a kd-tree, we compute s_0's nearest neighbor, and add this pair to the initially empty priority queue. Then we each iteration, we extract the closest pair (p_i, p_j) from the priority queue, add this edge to the spanning tree edge list, add p_j to the set S, remove p_j from the kd-tree, and finally update the nearest neighbor pairs and insert them in the priority queue based on squared distances.

Dependents Lists: The final question that we need to answer is how to determine which points of S need their nearest neighbors updated at the end of each iteration. Certainly, we need to do this for the new point p_j. In addition, every point $p_k \in S$ that depends on p_j as its nearest neighbor must also be updated.

We say that p_k *depends* on $p_j \in P \setminus S$ if p_j is the nearest neighbor of p_k. The set of all points in S that depend on p_j constitute its *dependents list*, denoted $\text{dep}(p_j)$. Whenever a point $p_j \in P \setminus S$ is added to the spanning tree, we need to update the nearest neighbor of p_j and all the members of $\text{dep}(p_j)$. For example, for the situation shown on the right side of Fig. 2, we have the following. (Note that the points of S do not need dependents lists.)

Point (p)	**Dependency list (dep(p))**
`BWI`	{`SFO`}
`DCA`	{`DFW`}
`SEA`	{}
`IAD`	{`ORD, ATL`}
`LAX`	{}

Each such list can be stored, for example, as a Java `ArrayList`. There is one for each point of $P \setminus S$. Initially, all of these lists are empty. Whenever we add an entry (p_i, p_j) is added to the priority queue, we add p_i to $\text{dep}(p_j)$ as well.

So, when p_j is added to the spanning tree, we iterate through the members of $\text{dep}(p_j)$ and compute its new nearest neighbor. But now the question emerges, how to we access this dependency list efficiently? We can do this by creating one more data structure:

Hash Map: to store the points of $P \setminus S$. Each element of the map is associated with its dependents list. This can be done using a Java `HashMap`.

In the parlance of this assignment, each point is a labeled point, say `LPoint`. An array list of points is of type `ArrayList<LPoint>`. A hash map that maps a point to its associated dependents list is therefore `HashMap<LPoint, ArrayList<LPoint>>`. If we create such an object, called, say, `dependents`, and given a point `pt`, we can access dep(pt) with: `ArrayList<LPoint> dep = dependents.get(pt)`.

Redundant Priority Queue Entries: There is a subtle issue with our algorithm as described. Whenever we compute a new nearest-neighbor pair (p_i, p_j) to add to our priority queue, it is possible that there was already a nearest neighbor pair (p_i, p'_j) in the queue. Ideally, we should remove this from the priority queue, but most priority queues (including our QuakeHeap) do not support deletion.

There is an easy fix, however. The only reason that one pair (p_i, p_j) overrides another (p_i, p'_j) is that p'_j was added to the spanning tree. Whenever we remove a pair (p_i, p'_j) from the priority queue, we check whether p'_j is in the tree. We can do this efficiently by accessing our set data structure for S. If so, we ignore this edge and go on to the next one.

Summary: That's a lot of data structures! But this is typical of many efficient algorithms. We need to access the various structures as efficiently as possible, and the best way to do this is to store them in an appropriate data structure. Fig. 3 demonstrates the iterations of the algorithm. For further information about the algorithm, consult the lecture notes.

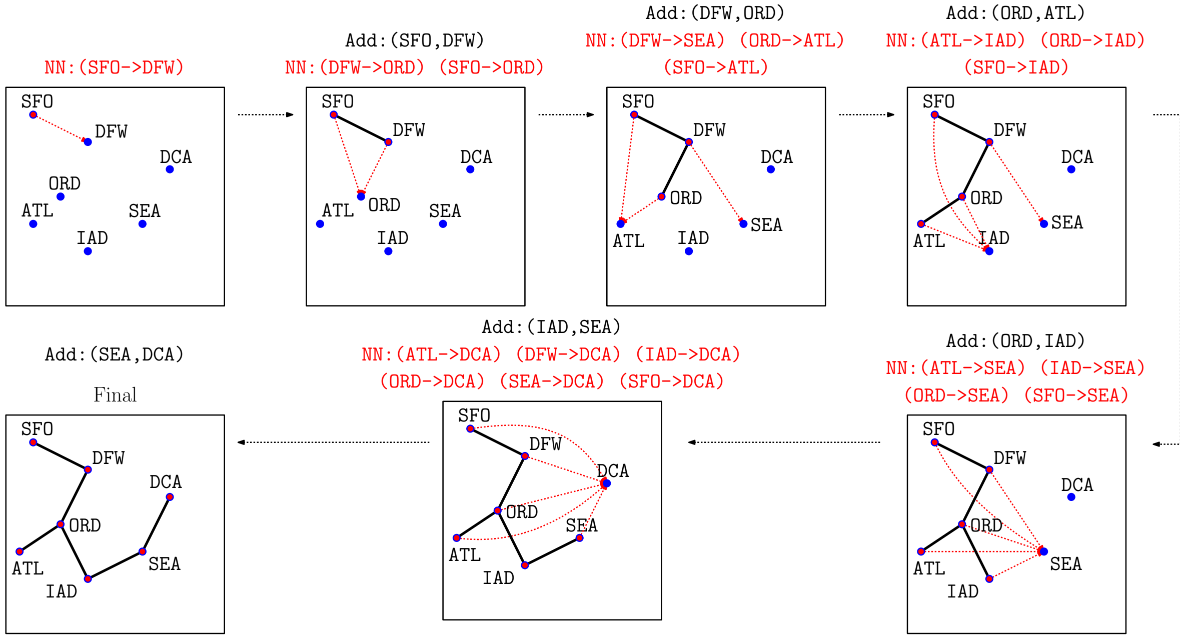

Figure 3: Prim's EMST algorithm walkthrough.

Requirements: Your program will implement the following functions for the `EMSTree`. While you can implement the operations internally however you like (subject to the style and efficiency

requirements given below), the following function signatures should not be altered. As part of the skeleton code, we will provide you with the `LabeledPoint2D` interface, and two useful classes, `Point2D` and `Rectangle2D`. (If you wish to modify these objects, do not alter them. Instead, create your own copy, say `MyPoint2D`, and make modifications there.)

`EMSTree(Rectangle2D bbox):` This initializes by creating the basic objects (set, kd-tree, heap, hashmap) needed by the EMST algorithm. The bounding box can be passed into your constructor for your kd-tree. You may set the other parameters for the kd-tree and quakeheap as you like. (We would suggest using a max-height-difference of 2 for your kd-tree and a number of levels of 10 for your quakeheap.)

`void addPoint(LPoint pt):` This inserts a point into the set P of points that will form the EMST. The EMST is *not* constructed at this point, and no error checking is done.

We would recommend doing two things. First, insert `pt` into your set of points P. Second, create a dependents list for this point. If you are using a hash map of array lists for dependents, this can be done with `dependents.put(pt, new ArrayList<LPoint>())`.

`void clear():` This removes all the points from your points set P, clears the edge list for the EMST. You can also clear the kd-tree, the priority queue, and the hash map used for the dependents lists.

`int size():` Returns the current number of points in P.

`ArrayList<String> buildEMST(LPoint start) throws Exception:` Builds the EMST for the current point set P, where `start` is the starting vertex (called s_0 above). For the purposes of testing, it returns a Java `ArrayList` of strings (described below) that provides a summary of the construction process.

This function checks for the validity of the point set. If any point lies outside the bounding box, it throws an `Exception` with the error message `"Attempt to insert a point outside bounding box"`. If there are two or more points with the same coordinates, it throws an `Exception` with the message `"Attempt to insert a duplicate point"`. (Update: 4/30) If multiple points fail these conditions, the first to be added triggers the exception. If an exception is thrown, the EMST that results is empty.

This function computes the entire EMST. It begins by clearing out the point set S, the list of tree edges, the kd-tree, and the priority queue. It clears the dependents lists for all the points. It inserts all the points into the kd-tree except the start point. And then it starts Prim's algorithm.

`ArrayList<String> listTree():` This lists the edges of the EMST. If the tree has not yet been built (e.g., points were added but `buildEMST` was not called) then it returns an empty list. Otherwise, it returns the edges in the order that they were added by Prim's algorithm. Assuming that you represent your each edges as `Pair<LPoint>`, the edge `e` could be listed using:

```
"(" + e.getFirst().getLabel() + "," + e.getSecond().getLabel() + ")"
```

For example, the tree shown in Fig. 3 would generate an array list with the following 6 entries

```
"(SFO,DFW)" "(DFW,ORD)" "(ORD,ATL)" "(ORD,IAD)" "(IAD,SEA)" "(SEA,DCA)"
```

Summary of `buildEMST`: The array list returned by your `buildEMST` function (see above) summarizes the algorithm's processing. It contains a line for every new point inserted into the EMST. The first line just gives the pair consisting of the start vertex (`start`) and its nearest-neighbor (`nn`),

```
"new-nn: (" + start.getLabel() + "->" + nn.getLabel() + ")"
```

(This is the first entry that is placed in your priority queue.)

For each remaining point added to the tree, it prints the newly added edge. If you store your edge as a `Pair` object, you can just rely on the `toString` method provided by this object. For example, if `edge` denotes the newly created pair, of type `Pair<LPoint>`, the added edge is reported with:

```
"add: " + edge + " new-nn:"
```

Following this, on the same line, you will output all the updated nearest-neighbor pairs. For each new nearest-neighbor pair consisting of a point `pt` from S and its nearest neighbor `nn` from $P \setminus S$, the output consists of the labels of these two points:

```
"(" + pt.getLabel() + "->" + nn.getLabel() + ")"
```

For example, when we added the edge `(DFW,ORD)` in Fig. 3, we generated the following new nearest-neighbor pairs:

```
(DFW->SEA) (ORD->ATL) (SFO->ATL)
```

There is one final issue. For testing purposes, we need your nearest-neighbor list to match ours exactly. For this reason, you should sort the entries of this list. You can do this easily using `Collections.sort`. (There is no need to design a special comparator.)

Below, we given an example of the seven entries in the array list returned by `buildEMST` on the example from Fig. 3. (The "`... more, omitted`" is not really in the string. We just ran out of space!)

```
new-nn: (SFO->DFW)
add: (SFO:(12.0,88.0)--DFW:(30.0,84.0)) new-nn: (DFW->ORD) (SFO->ORD)
add: (DFW:(30.0,84.0)--ORD:(19.0,58.0)) new-nn: (DFW->SEA) (ORD->ATL) (SFO->ATL)
add: (ORD:(19.0,58.0)--ATL:(5.0,51.0)) new-nn: (ATL->IAD) (ORD->IAD) (SFO->IAD)
add: (ORD:(19.0,58.0)--IAD:(32.0,41.0)) new-nn: (ATL->SEA) (IAD->SEA) (ORD->SEA) (SFO->SEA)
add: (IAD:(32.0,41.0)--SEA:(51.0,53.0)) new-nn: (ATL->DCA) (DFW->DCA) ... more, omitted
add: (SEA:(51.0,53.0)--DCA:(65.0,68.0)) new-nn:
```

Skeleton Code: As usual, we will provide skeleton code on the class Projects Page. You should replace the `EMSTree.java` file with your own. We have provided a number of utility objects for you, include the `Point2D`, `Rectangle2D`, and `Airport` from the previous assignment. We have also provided `Pair` for storing edges.

You must use the package "`cmsc420_s22`" for all your source files. (This is required for the autgrader to work.) As usual, we will provide a driver program (`Tester.java` and `CommandHandler.java`) that will input a set of commands. Here is a portion of the class's public interface (and of course, you will add all the private data and helper functions). You should not modify the signature of the public functions, but you are free to set up the internal structure however you like.

```
package cmsc420_s22;

import java.util.ArrayList;

public class EMSTree<LPoint extends LabeledPoint2D> {

    public EMSTree(Rectangle2D bbox) { ... }
    public void addPoint(LPoint pt) { ... }
    public void clear() { ... }
    public int size() { ... }
    public ArrayList<String> buildEMST(LPoint start) throws Exception { ... }
    public ArrayList<String> listEMST() { ... }
    // ... and so on
}
```

Hints: (Update: 4/30) Here are some of the data objects that were referred to in the above descriptions and how they might be implemented. These are best implemented as private members of your `EMSTree` structure. (The following are only suggestions, and you may implement these how you, subject to the efficiency requirements below.)

`ArrayList<LPoint>`	`pointList`	All the points (P)
`HashSet<LPoint>`	`inEMST`	Subset of points in the tree (S)
`ArrayList<Pair<LPoint>>`	`edgeList`	List of edges in the EMST
`HBkdTree<LPoint>`	`kdTree`	The kd-tree for $P \setminus S$
`QuakeHeap<Double, Pair<LPoint>>`	`heap`	priority queue of NN pairs
`HashMap<LPoint, ArrayList<LPoint>>`	`dependents`	dependents lists

We will provide the same objects as in Programming Assignment 2, including `Point2D`, `Rectangle2D`, and a simple class `Pair` for storing pairs. Although we have included skeletons of `QuakeHeap.java` and `HBkdTree.java` in the skeleton code, you can replace them with any objects you wish.

Efficiency requirements: (Update 4/30) While you are encouraged to use your own QuakeHeap and HBkdTree data structures, you are allowed to use any comparably efficient structure (e.g., you may use Java's built-in `PriorityQueue` data structure). You may substitute other structures for the ones we have recommended (e.g., `HashSet`, `HashMap`, `ArrayList`) provided that the required operations can be performed efficiently, say in $O(\log n)$ time.

Testing/Grading: (Update 4/30) We will use the standard Gradescope-based grading process that we have used in previous assignments.

The autograder will provide files `Point2D.java`, `LabeledPoint2D.java`, `Rectangle2D.java`, `Pair.java`, `Tester.java`, and `CommandHandler.java`. You will need to upload all other files as needed by your program. At a minimum, this will consist of `EMSTree.java`. If you use any other files, such as `QuakeHeap.java` and `HBkdTree.java`, these will be uploaded as well. (Update 5/1) An example of what your Gradescope submission window might look like is shown in Fig. 4

Submit Programming Assignment

Upload all files for your submission

SUBMISSION METHOD

Upload GitHub Bitbucket

Add files via Drag & Drop or Browse Files.

NAME	SIZE	PROGRESS
EMSTree.java	11.5 KB	
HBkdTree.java	21.9 KB	
QuakeHeap.java	20.9 KB	

Upload Cancel

Figure 4: Possible Gradescope submission window.