CMSC 420: Lecture X02
Applications of Data Structures: Euclidean MST
Supplemental

Data Structures and Algorithms: Throughout the semester, we have often explained our interest in data structures on the basis of their application to transactional and query-based systems. For example, "find the student with ID 987654321 in the university database" or "how many patients in my medical database have ages between 50–60 and blood pressure over 120?" or "what is the nearest coffee store to my current location?".

Data structures are often used in other contexts as well. Among the most common is in algorithm design. For example, Dijkstra's shortest-path algorithm requires the use of priority queue to determine the order in which vertices are processed, and Kruskal's minimum spanning tree algorithm uses a union-find data structure to determine whether two nodes are in the same partial spanning tree. The fastest versions of these algorithms are based on the fastest versions of these data structures.

In this lecture, we will see how data structures of various sorts can be combined to solve interesting computational problems. We will consider is the Euclidean minimum spanning tree problem.

Minimum Spanning Trees: A common problem in communications networks and circuit design is that of connecting together a set of nodes (communication sites or circuit components) by a network of minimal total length (where length is the sum of the lengths of connecting wires). We assume that the network is undirected. To minimize the length of the connecting network, it never pays to have any cycles (since we could break any cycle without destroying connectivity and decrease the total length). Since the resulting connection graph is connected, undirected, and acyclic, it is a *free tree*.

The computational problem is called the *minimum spanning tree* problem (MST for short). More formally, given a connected, undirected graph $G = (V, E)$, a *spanning tree* is an acyclic subset of edges $T \subseteq E$ that connects all the vertices together (see Fig. 1(a)). Assuming that each edge (u, v) of G has a numeric weight or cost, $w(u, v)$, (may be zero or negative) we define the cost of a spanning tree T to be the sum of edges in the spanning tree

$$w(T) = \sum_{(u,v)\in T} w(u, v).$$

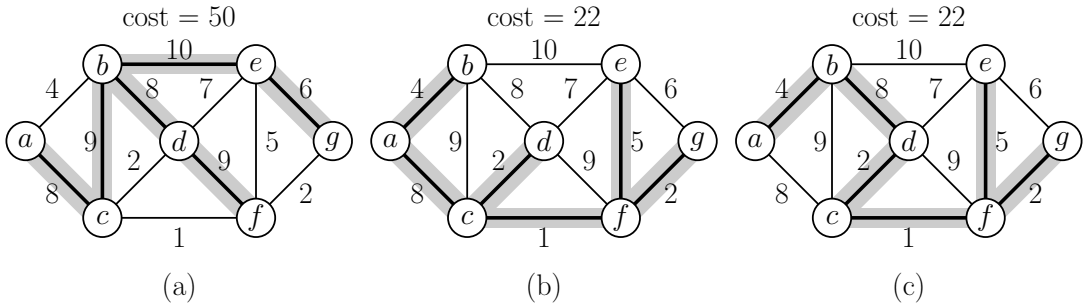

Fig. 1: Spanning trees for a graph. ((b) and (c) are minimum spanning trees).

A *minimum spanning tree* (MST) is a spanning tree of minimum weight. Note that the minimum spanning tree may not be unique (see Fig. 1(b) and (c)), but it is true that if all the edge weights are distinct, then the MST will be distinct. This is a rather subtle fact, which we will not prove.

The following lemma provides a few useful facts about spanning trees (minimum or not).

Lemma: (i) A spanning tree for a graph with n vertices has exactly $n-1$ edges.
(ii) There exists a unique path between any two vertices of a spanning tree.
(iii) Adding any edge to a spanning tree creates a unique cycle. Breaking *any* edge on this cycle results in a spanning tree.

Euclidean Minimum Spanning Tree: Spanning trees are often built in geometric settings. We are given a set $P = \{p_1, \ldots, p_n\}$ of n points in $\mathbb{R}^d$. For concreteness, let's assume that the dimension is 2, and each point $p_i = (x_i, y_i)$. We can define the spanning tree naturally in this context. Consider the graph whose vertex set is P and whose edges consist of all the $\binom{n}{2}$ pairs (p_i, p_j). We define the weight of an edge $w(p_i, p_j)$ to be the Euclidean distance between them, that is

$$w(p_i, p_j) \;=\; \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

This weighted graph is called the *Euclidean graph* of the point set. The minimum spanning tree of this graph is called *Euclidean minimum spanning tree* or, EMST for short (see Fig. 2).

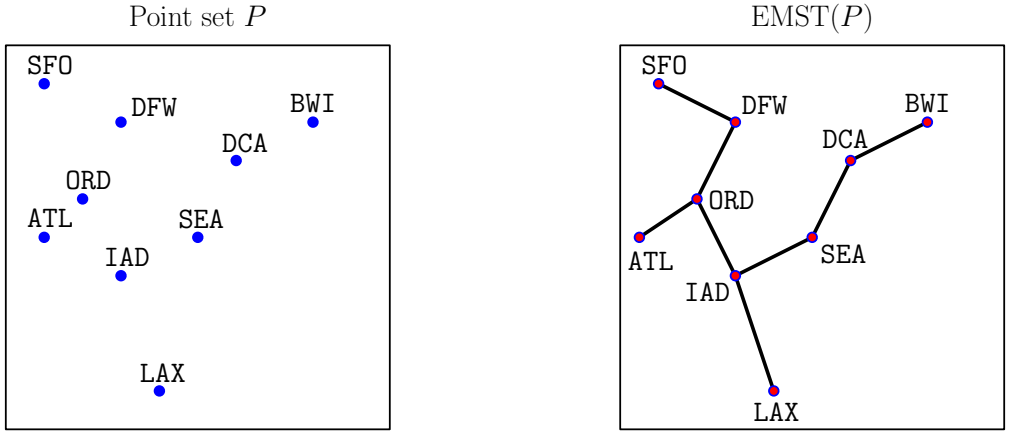

Fig. 2: Euclidean Minimum Spanning Tree.

An interesting fact about MSTs is that they are determined by the order of the edges, not the actual values. So, for our purposes, rather than using the actual Euclidean distance, we can save the effort of computing the square root, and instead use the *squared Euclidean distance*. Henceforth, we assume that the edge weight is:

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

Not only is the more efficient, it has the advantage that integer-valued coordinates lead to integer-valued squared distances, and so we do not need to worry about floating-point round-off errors causing problems.

The EMST is an important geometric structure for numerous reasons:

- It is often used as the first step in clustering algorithms in machine learning
- It is the first step in approximation algorithms for the travelling salesman problem (TSP)
- It is used in constructing wireless ad hoc networks

There are many different algorithms for computing MSTs, including Kruskal's algorithm, Prim's algorithm, and Boruvka's algorithm. All of them can be extended to compute EMSTs, but the problem is the Euclidean graph has $O(n^2)$ edges. If n is large (say, $n = 1,000,000$) the total number of edges in the Euclidean graph is huge. Since the spanning tree has only $n - 1$

Generic approach: The standard algorithms for computing MSTs are all *greedy* (Kruskal's, Prim's, and Boruvka's). The intuition is simple. Add the lightest (lowest weight) edge(s) you can, provided it does not create a cycle. The algorithms differ only in how these lightest edges are identified.

The correctness of these algorithms is based on lemma that shows that whenever the vertices are partitioned into two groups, the lightest edge between the two groups is always safe to add to the MST. In the following, given a subset $S \subset P$, we use the notation $P \setminus S$ (set subtraction) to denote the elements of P that are not in S.

Lemma: Given a point set P and any nonempty proper subset $S \subset P$, the closest pair of points $p_i \in S$ and $p_j \in P \setminus S$ is safe to add to EMST of P.

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. 3). Initially $S = \{s_0\}$ and the algorithm terminates when all the points of P are in S.

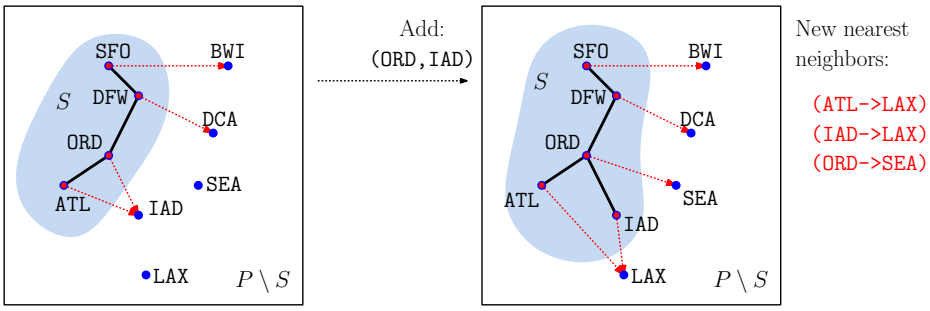

Fig. 3: Originally, $S = \{\texttt{SFO}, \texttt{DFW}, \texttt{ORD}, \texttt{ATL}\}$ with the nearest-neighbor pairs `(SFO,BWI)`, `(DFW,DCA)`,`(ORD,IAD)`, and `(ATL,IAD)`. The closest of these, `(ORD,IAD)` is added to the EMST, `IAD` is added to S, and new nearest neighbors `(ATL,IAD)`, `(ORD,IAD)`, 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. 3). 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.

List: to store the edges of the spanning tree. This can be implemented using a Java `ArrayList` or `LinkedList`. An edge is naturally represented as a pair of points. Let us assume we have access to such a class, called `Pair`).

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

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 kd-tree, for example.

Priority Queue: to store the nearest-neighbor pairs ordered by their distance. Each entry stores the associated pair of points (e.g., from Fig. 3, `(SFO, BWI)` is one of these pairs, and the associated key is the squared distance $d(\texttt{SFO}, \texttt{BWI})$.) This can be implemented using a heap data structure.

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. 3, 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. (For example, this can be done using a Java `HashMap`.)

Redundant Priority Queue Entries: There is an 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 do not support efficient 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.

Example: An example is shown in Fig. 4. With each step we extract the lightest edge from the priority queue (illustrated in red broken lines) and add it to the spanning tree (shown as black edges). The endpoint of this edge is added to the spanning tree and removed from the kd-tree. This new point will have one or more nearest-neighbor pairs (red edges) incident on it. These are the point's dependents. We compute new nearest neighbors for all the dependents and insert them in the priority queue. The algorithm terminates when all the points have been added to the tree, or equivalently when the kd-tree is empty.

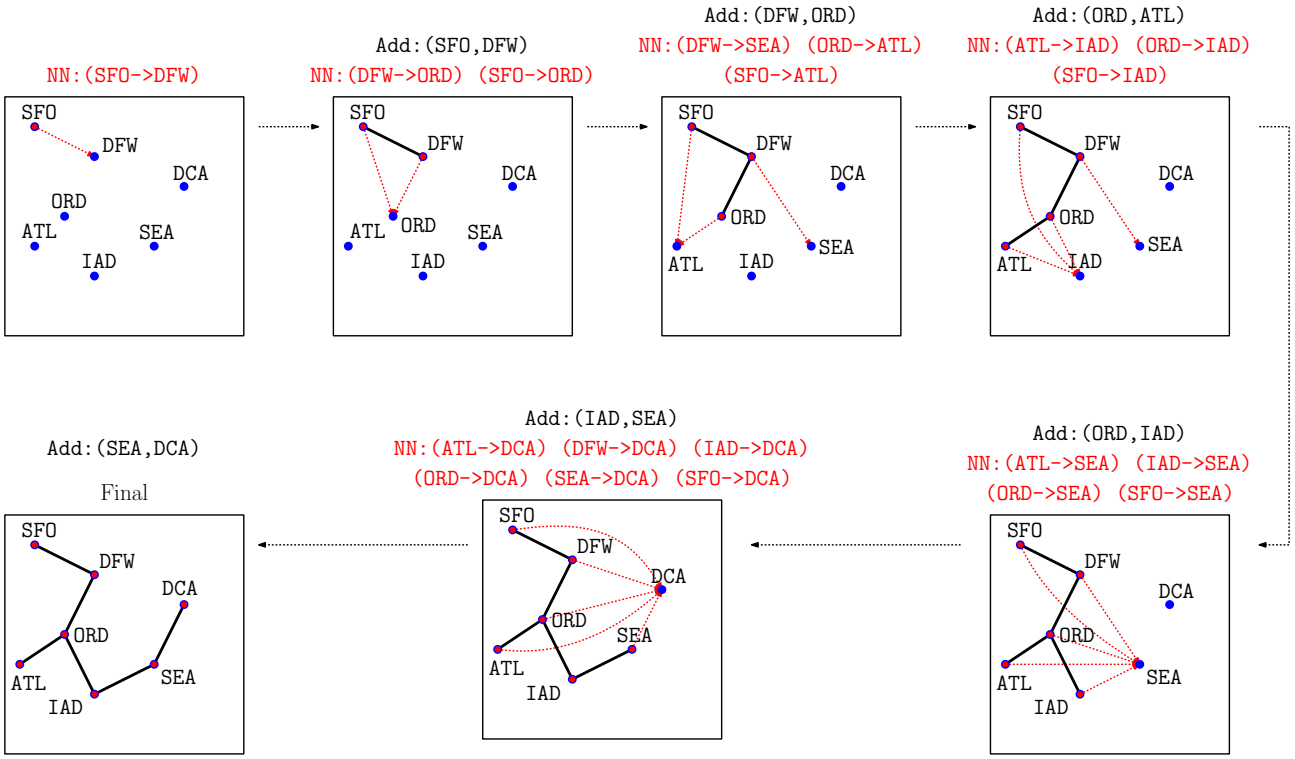

Fig. 4: Full execution of Prim's EMST algorithm on a small example. The text in black indicates the edge that is added to the EMST, and the text in red indicates the new nearest-neighbor pairs.

Implementing the Algorithm: To convert the above description into pseudocode, let's give some better names to our objects. First, it will be useful to define a new class, called `Pair` that stores a pair of points. It supports functions `getFirst()` and `getSecond()`, which extract the first and second components of the pair. We will use this for representing edges of

the EMST, and we will also use this to represent nearest neighbor pairs ($p_i \rightarrow p_j$), where $p_i \in S$ and $p_j \in P \setminus S$.

Let `inEMST` denote our set data structure storing the subset of points S that lie within the current EMST. We can implement it in Java using a `HashSet`. Let `kdTree` denote the kd-tree storing the points of $P \setminus S$. Let `heap` denote the current priority queue consisting of nearest-neighbor pairs. We can use any standard heap data structure. Let `edgeList` denote the list of edge in the EMST. It can be represented as a Java `ArrayList`. Finally, `dependents` is a hash map, where each entry is accessed by a point. The associated value is a Java `ArrayList` of the dependents for this point.

Adds a single edge to the EMST

```
void addEdge(Pair<Point> edge) {
    Point pt2 = edge.getSecond()                    // endpoint to add to EMST
    edgeList.add(edge)                              // add edge to the EMST
    inEMST.add(pt2)                                 // add pt2 to the EMST
    kdTree.delete(pt2)                              // remove pt2 from the kd-tree
    ArrayList<Point> dep2 = dependents.get(pt2)     // get pt2's dependent points
    dep2.add(pt2)                                   // include pt2 as well
    forall (Point pt3 in dep2) {                    // compute new nearest neighbors
        Point nn3 = kdTree.nearestNeighbor(pt3)     // pt3's nearest neighbor
        if (nn3 == null) break                      // out of points? -- we're done
        addNearNeighbor(pt3, nn3)                   // add this near-neighbor pair
    }
}
```

Before presenting Prim's algorithm, we first introduce a couple of utility functions. The first, `addEdge` adds a single edge to the spanning tree. This performs all the necessary actions for adding a new edge. It adds the edge to the list of EMST edges, it includes the new point in the EMST, it removes this point from the kd-tree, and it goes through all the point's dependents and computes new nearest neighbors for each. The resulting pairs are inserted into the priority queue by invoking the other utility function `addNearNeighbor`. Note that whenever a new pair `(pt->nn)` is added to the priority queue, we add `pt` to the dependents list for `nn`. This is shown in the code block.

Adds a new nearest-neighbor pair

```
void addNearNeighbor(Point pt, Point nn) {
    double dist = d(pt, nn)                         // squared distance to nearest neighbor
    Pair<Point> pair = new Pair<Point>(pt, nn)      // new nearest-neighbor pair
    heap.insert(dist, pair)                         // add to priority queue
    dependents.get(nn).add(pt)                      // add to nn's dependents list
}
```

There are a couple oddities you may wonder about with the function `addEdge`. First off, when we invoke the kd-tree to compute the nearest neighbor of `pt3`, why did we check whether the result is `null`? This is because when the very last point is added to the spanning tree and is deleted from the kd-tree, the kd-tree is now empty. We know that the algorithm can terminate at this point, so we simply exit the loop and return from the function. The second oddity is that we add `pt2` to its own dependency list. Why? This is just a coding trick. We need to compute nearest neighbors for all of `pt2`'s dependents and we need to do this for `pt2`

as well. By adding `pt2` to it's own list, we don't need to process it separately. (It's just a lazy coding trick.) The utility `addNearNeighbor` is shown in the following code block.

Given this, we can now present the pseudocode for Prim's algorithm. It is given the starting point, `start`, as its argument. We assume that the points are stored in a global list `pointList`. We assume that the various data structures `edgeList`, `inEMST`, `kdTree`, and `heap` are all globals. The process begins by initializing the various data structures by wiping them out (presented below in the function `initializeEMST`.) It inserts the start point and initializes the priority queue with its nearest neighbor. We then repeatedly extract pairs from the priority queue and add them to the EMST (checking first that the entry is not redundant).

Prim's EMST Algorithm

```
void buildEMST(Point start) {
    initializeEMST(start)                          // initialize
    Point nn = kdTree.nearestNeighbor(start)       // get start's nearest
    if (nn == null) return                         // no more points -- done
    addNearNeighbor(start, nn)                     // add nearest neighbor pair
    while (kdTree is not empty) {
        Pair<Point> edge = heap.extractMin()       // extract next edge
        Point pt2 = edge.getSecond()               // get destination end point
        if (pt2 is not in inEMST) {                // not redundant?
            addEdge(edge)                          // add the edge to the EMST
        }
    }
}
```

Finally, we present the initialization. It clears out all the structures and then inserts all the points, except for `start` in the kd-tree.

Initialization for Prim's algorithm

```
void initializeEMST(Point start) {
    edgeList.clear()                               // clear the edge list
    inEMST.clear()                                 // clear the EMST set
    heap.clear()                                   // clear the heap
    for each (dep in dependents) {                 // clear all the dependents
        dep.clear()
    }
    kdTree.clear()                                 // clear the kd-tree
    for each (pt in pointList)                     // add all but start
        if (pt != start) kdTree.insert(pt)
    inEMST.add(start)                              // add start to EMST
```

Analysis: The correctness of this algorithm follows from the fact that we are always selecting the lightest edge from the current spanning tree to the remaining points.

It is trickier to analyze the running time. Each iteration of the algorithm involves removing a pair from the heap, adding the edge to the tree, and updating the nearest neighbors for the dependents of the newly added point. This last step dominates the running time. There are two issues here. The first is how fast we can compute nearest neighbors in a kd-tree. While there are no good upper bounds (the query time can be $O(n)$ for pathological data sets) the running time is actually closer to $O(\log n)$ for typical data sets.

This then leads to an overall running time of $O(n \cdot c(n) \log n)$, where $c(n)$ is the average number of times each point needs to update its nearest neighbor. This seems to be an interesting question for further research. I would expect that for uniformly distributed point sets $c(n) = O(\sqrt{n})$ in the plane, but I do not have a proof of this. It might be a worthwhile topic for an empirical algorithm analysis.