Minimum Spanning Tree Problem

Notation

We will consider an undirected graph \((V, E)\) and for each \(e \in E\) there is positive \(c(e)\).

A spanning network is \(T \subseteq E\) such that \((V, T)\) is connected.

Our goal is to find a spanning network with minimum \(c(T)\).

Observation. Suppose that \(C\) is a simple cycle, \(e \in C \subseteq T\) and \(T\) is a spanning network. Then \(T - \{e\}\) is a spanning network as well.

Because removing edges from cycles decreases \(c(T)\), a spanning network with the minimum cost has no cycles — hence it is a tree. Thus Minimum Spanning Tree Problem.

Generalization. We generalize the problem in such a way that a decision to select an edge produces a smaller instance (unless it is a really stupid decision). We specify an edge set \(T_0\) and we minimize \(c(T)\) such that \((V, T)\) is connected and \(T_0 \subseteq T \subseteq E\). We will denote such an instance \((V, E, c, T_0)\).

Selection of an edge reduces the problem by inserting it to \(T_0\). The basic case is when \((V, T_0)\) is connected.

Definition. For \(A \subseteq V\) we define \(cut(A)\) as the set of edges with one endpoint in \(A\) and another in \(V - A\).

Observation. If \(cut(A)\) is not empty, any spanning network must have an edge from \(cut(A)\).
Good choice for Minimum Spanning Tree. If $A$ is a connected component of $(V, T_0)$ and $e$ is an edge of $\text{cut}(A)$ with the minimum $c(e)$, then the minimum costs for $(V, E, c, T_0)$ and $(V, E, c, T_0 \cup \{e\})$ are the same.

To see that, suppose $T^*$ is the best tree for $(V, E, c, T_0)$. If $e \in T^*$, we are done. Otherwise, $e = \{u, v\}$ and $T$ contains a path $P$ from $u$ to $v$. $P \cup \{e\}$ is a simple cycle, moreover, $P$ contains an edge $e' \in \text{cut}(A)$. Thus $T' = T^* \cup \{e\} - \{e'\}$ is also a solution. Because $c(e) \leq c(e')$, $c(T') \leq c(T^*)$.

There are several algorithms for computing Minimum Spanning Tree that are making good choices based on that principle.

In Prim’s algorithm we have a starting node $s$ and we select a minimum cost edge from $\text{cut}(A)$ where $A$ is a connected component of $(V, T)$ that contains $s$. Thus we grow a single tree.

In Kruskal’s algorithm we select a minimum cost edge that is not contained in any of the connected components of $(V, T)$.

The only remaining consideration is how to implement these principles with the minimum running time (the memory will always be proportional to the size of the graph representation).
Prim’s Algorithm.

We have a set $B$ of nodes that are not in the component of $s$, so the component of $s$ is $V - B$.

For each $u \in B$ we have the least cost edge $\{u, v\}$ that connects it to $V - B$; we have $Cost[u] = c(u, v)$ and $Neighbor[u] = v$. If there is no such edge, we set $Cost[u] = \text{infty}$.

Initialization. $Cost[u] = \text{infty}$ for every $u \in B = V$ except for $Cost[s] = 0$.

Iteration.

```plaintext
find $v \in B$ with the minimum $Cost[v]
remove v from B, add \{v, Neighbor[v]\} to the tree
for (every edge \{v, w\}) {
    if (w not in B)
        continue;
    if (Cost[w] > c(v, w))
        Cost[w] = c(v, w),
        Neighbor[w] = v;
}
```

To find $v \in B$ with the minimum $Cost[v]$ we use a priority queue. If $|V| = n$, $|E| = m$, we perform $n - 1$ DeleteMin operations and up to $m$ operations of decrease a priority. Let $M$ be the time needed to perform a DeleteMin, and $D$ the average time needed to decrease a priority.

The running time is $nD + mD$.

The simplest implementation: DeleteMin by scanning arrays, $M = n$, and decrease a priority by updating arrays, $D = 1$. This gives running time of $O(n^2)$.

Using a binary heap for priority queue we have $D = M = \log n$. This gives running time of $O(m \log n)$.

Using Fibonacci heaps or 2-3 heaps for priority queue we have $D = 1, M = \log n$. This gives running time of $O(m + n \log n)$. 
Kruskal’s Algorithm.

To select the least cost edge among the edges we have not considered yet we simply sort edges. This takes time $O(n \log n)$, but we may have a situation when faster sorting is available, e.g. small range of edge costs that allows us to use Radix Sort.

When we consider an edge, both endpoints may be in the same connected component of $(V, T)$. Then we discard it.

If we did not discard that edge, we insert it to $T$.

One case see that after sorting, the critical operations are

a) checking if two edge endpoints are in the same components (Find operation)

b) if those two components are different, inserting the edge to $T$ replaces them with their union (Union operation)

This is data structure for disjoint sets. We also initialize the structure, with $n$ singleton sets. Note that it does not matter how we identify the components, it only matters if we properly recognize if two identifications are different or equal.
Disjoint Sets with linking by rank and path compression

This is a simple and VERY efficient implementation. We use two arrays, \( \text{Rank}[n] \) and \( \text{Parent}[n] \)

initialization

\[
\text{for } (i = 0; i < n; i++) \\
\text{Rank}[i] = 0, \text{Parent}[i] = i;
\]

Find(i)

\[
\{ \text{j = i; } \\
// \text{searching loop} \\
\text{while } (\text{Parent}[i] \neq i) \\
\text{i = Parent}[i]; \\
// \text{path compression loop} \\
\text{while } (\text{(k = Parent}[j]) \neq i) \\
\text{Parent}[j] = i, \\
\text{j = k; } \\
\text{return } i;
\}
\]

Link(i,j)

\[
\{ \text{if } (\text{Rank}[i] < \text{Rank}[j]) \\
\text{Parent}[i] = j; \\
\text{else if } (\text{Rank}[i] > \text{Rank}[j]) \\
\text{Parent}[j] = i; \\
\text{else} \\
\text{Rank}[i]++, \\
\text{Parent}[j] = i;
\}
\]

Union(i,j) \equiv \text{Link(Find(i),Find(j))}

The running time of \( \text{Link} \) is constant, of \( \text{Find} \), linear in the worst case. We can be precise: set the cost of a \( \text{Link} \) to be 1, and the cost of \( \text{Find} \) to be the number of nodes/elements that are visited. And we can assume that after the initialization we perform \( N \) of these operations.

The bad news is that we can have such sequences of \( N \) operations that the cost is at least \( N\alpha(n) \) where \( \alpha \) is a function that goes to infinity.

The good news is that if \( n < 2^{90} \) then this cost is at most \( 3N \).