MINIMUM SPANNING TREES

Consider a graph $G = (V, E)$. A spanning subgraph of $G$ is a graph of the form $G' = (V, E')$ where $E' \subseteq E$. It may or may not be a connected graph. If the subgraph is a tree, then it is called a spanning tree.

If $G$ is a weighted graph, there are applications that require the computation of a spanning tree with minimum cost (i.e., total weight of all edges). Such a tree is called a minimum spanning tree.

In fact, one can prove that a minimum connected spanning subgraph of $G$ must be a tree.

(Do you see why? Prove it.)

Algorithms to Compute MST:

1. **Kruskal's algorithm** for graph $G = (V, E)$:
   - Insert edges from $E$ into the tree in increasing order of cost as long as no cycle is introduced.
   - Keep doing so until there is any vertex in $V$ which is not included in the tree.
Kruskal's algorithm:

Notice that Kruskal's algorithm may not yield a tree during intermediate steps (of course, it yields a tree in the last step).

Time complexity of naive implementation — finding min cost edge (not included in the tree as yet) is $O(V^2 + VE)$. Suppose
Kruskal's algorithm.

Consider a graph $G = (V, E)$. We want to create MST $T = (V_T, E_T)$.

Initialize $E_T$ to $\emptyset$, $V_T = \emptyset$.

While ($V_T \neq V$)
{
- Pick the least cost edge $e$ in $E - E_T$.
- $E_T = E_T \cup \{e\}$ if $e$ does not introduce a cycle. In this case $V_T = V_T \cup \{u_1, u_2\}$ where $u_1$ and $u_2$ are the endpoints of $e$.
}

Thus Kruskal's algorithm does not maintain a connected graph at all stages unlike Prim's algorithm. Rather it constructs the tree starting from connected components. Each time a new edge is selected, it could either be a new connected component in its own cycle, or it could be a bridge between two connected components. If $e$ induces a cycle, it means that its endpoints were part of the same connected component (otherwise it cannot induce a cycle).
So we need a data structure to
1) identify efficiently the label of a connected component to which a certain vertex belongs
2) merge two connected components into one.

The first operation is called FIND. The second one is called as UNION. To implement Kruskal's algorithm efficiently, we need a special data structure to handle such operations. It is called the Union-Find data structure.

Consider $G = (V, E)$. We will maintain an array called "Component" where $Component[v]$ contains the connected component to which vertex 'v' belongs. The "FIND(v)" operation, i.e., to find the connected component to which v belongs, now takes $O(1)$ time.

But finding the union of two components, i.e., finding $\text{Union}(S_1, S_2)$ where $S_1$ & $S_2$
are two connected components. To perform this operation we can retain the values of Component $[V]$ if $V \in S_1$ (where $|S_1| \gg |S_2|$) and change the values of Component $[W]$ (where $W \in S_2$) to Component $[V]$ (where $V$ is an arbitrary vertex in $S_1$). The worst case complexity of this operation is still $O(N)$ (though it can be shown that a sequence of $k$ Union operations will take no more than $O(k \log k)$ time).

If we use this data structure exactly as is for implementing Kruskal's algorithm, the total time complexity will be

$$O(E \log |E| + 1 + 1) + O(V \log V)$$

\[\downarrow\]

$$\downarrow$$

to extract min edge

to check the component labels for a sequence of $V$ "union" operations.

\[\downarrow\]

\[\downarrow\]

$= O(E \log E)$

$= O(E \log V)$
Claim: Even though a single union operation takes \( O(1\text{vw}) \) time in the worst case, a sequence of \( k \) union operations takes at most \( O(k \log k) \) time. In other words, the worst case doesn't occur too often.

Proof:
The bulk of the time in any union operation is spent in updating the "Component" array. Initially, every vertex of the graph \( G = (V, E) \) will be in its own connected component. A single union operation combines two single-element connected components into one. Hence after \( k \) union operations, only \( 2k \) elements in \( V \) are affected. In any union operation that affects Component \([v]\), the size of the set containing \( v \) at least doubles (remember: the union uses the label \( v \) from Component value of the larger set). In \( k \) union operations, the maximum size it can reach is \( 2k \). So, Component \([v]\) will get updated at most \( \log_2(2k) \) times. So the total time is \( O(k \log k) \).
as at most 2k elements are involved in any union operation.

Reverse delete algorithm

This is a relatively lesser known but very intuitive MST algorithm. In this method, you detect a cycle in the graph and delete the heaviest edge in it. As you are deleting an edge from the cycle, you are guaranteed not to disconnect the graph.

eg:
Another variant: Delete the heaviest edge that will not disconnect the graph.
Prim's algorithm:

It maintains two sets of vertices $S$ and $V-S$ (both disjoint sets). Initially, an arbitrary vertex is entered into $S$. The set $S$ is grown by adding in a new vertex $V$ such that $V \notin S$ (currently), edge $UV$ is the least cost edge (which does not include a cycle in the tree) amongst all $U \in S$.

---

$S = \{v_1\}$

$S = \{v_1, v_4\}$

$S = \{v_1, v_4, v_2\}$

$S = \{v_1, v_4, v_2, v_3\}$

$S = \{v_1, v_4, v_2, v_3, v_7\}$
$S = \{ V_1, V_4, V_2, V_3, V_7, V_6 \}$

$S = \{ V_1, V_4, V_2, V_3, V_7, V_6, V_5 \}$

Complexity: We maintain a heap of the edges based on edge cost. The total time taken to build the heap is $O(1E1)$. Each time we delete the min-cost edge from the heap, we need to check whether it will create a cycle in the tree. If so, we simply discard it and readjust the heap. Otherwise, we include it in the tree, re-adjust the heap, and proceed exactly as before. The total time taken will be $O(1E1 \log 1E1) = O(1E1 \log 1N1)$. 