Analysis of Algorithms I: Minimum Spanning Tree

Xi Chen

Columbia University

We discuss the Minimum Spanning Tree problem. The input of the problem is an undirected and connected weighted graph

$$G = (V, E) \quad \text{with weight function} \quad w : E \to \mathbb{R}$$

For convenience, assume all the weights are nonnegative (though this assumption is not necessary). The goal is to find a spanning tree T of G (i.e., a tree T with $n - 1$ edges from E such that all $n = |V|$ vertices are connected) with minimum total weight:

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

So MST is an optimization problem. We start with a "generic" method that grows a spanning tree from scratch by adding one edge at a time. We then present two algorithms that implement the generic method: Kruskal's algorithm and Prim's algorithm.

Let $A \subseteq E$ denote a set of edges. Then we say A is *safe* if there exists a minimum spanning tree T of G such that A is a subset of T. This gives us the following generic MST method:

1. set $A = \emptyset$ (A is clearly safe at the beginning)
2. while $|A| < n - 1$ and thus, does not form a spanning tree
3. find $(u, v) \in E - A$ s.t. $A \cup \{(u, v)\}$ remains safe
4. set $A = A \cup \{(u, v)\}$

This method is clearly correct by the definition of "safe" :), but is meaningless if we don't know how to find such a (u, v) efficiently.

So here comes the Cut Theorem that tells us how to find such an edge (u, v) efficiently. We need the following notation: A cut $(S, V - S)$ of an undirected graph $G = (V, E)$ is a partition of V, where S denotes a nonempty subset of V. We say an edge $(u, v) \in E$ crosses $(S, V - S)$ if one of its endpoints is in R and the other is in $V - R$. We say an edge $(u, v) \in E$ is a light edge crossing a cut $(S, V - S)$ if its weight is the minimum among all edges that cross $(S, V - S)$. (Note that given a cut $(S, V - S)$, there might be multiple light edges.)

Theorem

Let $A \subseteq E$ be a safe set of edges. Let $(S, V - S)$ be any cut such that no edge of A crosses it. Then for any light edge (u, v) crossing $(S, V - S)$, we have $A \cup \{(u, v)\}$ must be safe as well.

We prove the Cut theorem using the exchange argument. Before the proof, we need the following simple lemma:

Lemma

Let T be a spanning tree of G and $(u, v) \in E - T$. Then adding (u, v) to T forms a unique cycle:

$$(u, v) + \textit{the unique path between } u \textit{ and } v \textit{ in } T$$

Removing any edge $(u', v') \in T$ on the cycle gives us a new spanning tree T' of G. The total weight of T' is

$$\textit{total weight of } T + w(u, v) - w(u', v')$$

Now we prove the Cut theorem. As A is safe, by definition there is an MST T of G that includes A. If (u, v) is an edge in T, then we are done. Otherwise, the plan is to exchange (u, v) with an edge in T so that the new tree T' remains an MST of G.

To see this, using the previous lemma, we know there is a unique cycle in $T \cup \{(u, v)\}$ consists of (u, v) and the unique path between u and v in T. Also notice that (u, v) crosses $(S, V - S)$ so that u, v are on opposite sides of the cut $(S, V - S)$. There must be at least one edge on the path between u and v in T, which also crosses $(S, V - S)$ (why?). Let (x, y) be any such edge. Then we must have $(x, y) \notin A$ because A does not cross the cut.

To finish the proof, we exchange (x, y) with (u, v) to get a new spanning tree T' from T. The total weight of T' is

$$\text{total weight of } T + w(u, v) - w(x, y) \leq \text{total weight of } T$$

because (u, v) is a light edge of $(S, V - S)$ and thus,

$$w(u, v) \leq w(x, y)$$

It follows that T' must be a minimum spanning tree as well. The theorem follows because $A \cup \{(u, v)\} \subset T'$ (why?).

Now we present two algorithms based on the "generic" method. Both algorithms grow an MST from $A = \emptyset$ by adding edges to A one by one, while maintaining the property that A is safe. As we will see, both algorithms have a flavor of greedy algorithms.

Kruskal's algorithm. Let A be a safe set of edges. Because it is safe, there can be no cycles and thus, edges in A must form a forest ("multiple" trees) whose vertices are all those of the given graph. For example, $A = \emptyset$ at the beginning, which can be viewed as a forest of n trees, each consists of a vertex $v \in V$ only. For each round, Kruskal's algorithm finds an edge (u, v), of least weight, that connects two trees of the forest:

1. set $A = \emptyset$
2. while $|A| < n - 1$ do
3. find an edge (u, v), of minimum weight, that connects two trees of the forest formed by edges in A
4. set $A = A \cup \{(u, v)\}$

The correctness of Kruskal follows from the cut theorem (how?). An efficient implementation of Kruskal's algorithm is the following. Start by setting $A = \emptyset$ and sorting the $m = |E|$ edges $e_1, \ldots, e_m$ in nondecreasing order of weight:

$$w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$$

We process the edges in order of weight: $e_1, e_2, \ldots, e_m$. For the ith round, letting $e_i = (u, v)$, we check if u and v are in different trees in the current forest formed by A:

1. same tree, cannot add (u, v) to A (why?)
2. different trees, add (u, v) to A

It is easy to show that if $e_i = (u, v)$ and u, v lie in different trees (and thus, we add (u, v) to A), it must be the case that (u, v) is an edge, of least weight, that connects two trees in the current forest. This is because for every e_j with $j < i$ and $w(e_j) \leq w(e_i)$: either $e_j \in A$ already; or the two endpoints of e_j belong to the same tree (why?). As a result, we know that after each round, A remains safe and it is a safe set of edges upon termination. But is it possible that, after going through all edges in the nondecreasing order, A is still not a spanning tree? i.e., $|A| < n - 1$? Prove that this cannot happen and A must be a spanning tree upon termination and thus, a minimum spanning tree.

The implementation of Kruskal's algorithm uses a disjoint-set data structure. It maintains a collection of disjoint sets of vertices: each set contains the vertices in one tree of the current forest. So at the beginning, the data structure consists of n sets: $\{v\}$ for each $v \in V$. A disjoint-set data structure supports the following two operations: Find-Set(u) returns a representative element from the set that contains u (so we can determine whether two vertices u and v belong to the same tree by comparing Find-Set(u) and Find-Set(v)). Union(u, v) unites the two sets that contain u and v. After adding (u, v) to A (when u, v lie in different trees of the current forest), we can call Union(u, v) to combine the two sets of vertices: the set of vertices of u's tree and that of v's tree.

Kruskal (G, w):

1. set $A = \emptyset$
2. for each vertex $v \in G$ do
3. Make-Set (v) (at the beginning there are n singleton sets)
4. sort the edges into nondecreasing order by their weights
5. for each edge $(u, v) \in G$ in the nondecreasing order
6. if Find-Set $(u) \neq$ Find-Set (v)
7. $A = A \cup \{(u, v)\}$
8. Union (u, v)

The running time of Kruskal depends on the implementation of the disjoint-set data structure: n Make-Set, m Find-Set and n Union operations. If we use the linked-list representation discussed in Section 21.2, together with the weighted-union heuristic, each Make-Set costs $O(1)$, each Find-Set costs $O(1)$ and all the Union operations take $O(n \lg n)$ in total. So the total running time is $\Theta(m \lg m)$ for sorting plus $O(m + n \lg n)$, which is $\Theta(m \lg m) = \Theta(m \lg n)$ (why?). The total running time is clearly dominated by the sorting step. If the edges are already sorted, then using the forest representation discussed in Section 21.3 and 21.4 would give us an almost linear-time algorithm.

Prim's algorithm: Another example of Greedy Algorithms. Also you may find its structure similar to BFS, DFS and Dijkstra's algorithm (for single-source shortest path to be discussed next). Let $r \in V$ be an arbitrary vertex in the graph and set $A = \emptyset$ and $S = \{r\}$ at the beginning, where A is a set of edges and S is a set of vertices. Prim's algorithm maintains the following invariant: Prior to each iteration, A is a safe set and its edges form a tree that has vertices S and is rooted at r. For example, $A = \emptyset$ is clearly safe and it can be viewed as an empty tree rooted at r.

For each round, Prim's algorithm uses the cut theorem to grow A, a safe set of edges that form a tree rooted at r, as follows. It finds a light edge (u, v) crossing $(S, V - S)$, with $u \in S$ and $v \in V - S$, and adds (u, v) to A. By the cut theorem, it is easy to show that

$$A \cup \{(u, v)\} \text{ remains safe (why)}$$

Moreover, we have

$$\text{edges in } A \cup \{(u, v)\} \text{ still form a tree rooted at } r$$

but the new tree has vertices $S \cup \{v\}$. So the invariant holds.

To summarize, here is Prim's algorithm:

1. let r be a vertex of G; set $A = \emptyset$ and $S = \emptyset$
2. while $|A| < n - 1$ do
3. find a light edge (u, v) crossing $(S, V - S)$, $v \in V - S$
4. set $A = A \cup \{(u, v)\}$ and $S = S \cup \{v\}$

Correctness of Prim's algorithm can be proved using the Cut theorem and induction. A naive implementation is then for each round, enumerate all edges that cross $(S, V - S)$ to find a light edge, which takes time $\Theta(nm)$. A more efficient implementation is the following:

1. Each vertex $v \in V - S$ has two additional attributes $v.\text{key}$ and $v.\pi$. (We do not care the two attributes of those vertices already in S.) For each $v \in V - S$, let $u = v.\pi$, then

$$w(v.\pi, v) = v.\text{key} = \min_{u \in S} w(u, v)$$

2. We maintain a priority queue Q that contains all vertices in $V - S$, sorted based on the attribute $v.\text{key}$ of the vertices.

Suppose this is what we have at the moment, and A is a safe set that forms a tree rooted at r and connects S. Then it is easy to find a light edge crossing $(S, V - S)$: Simply call Extract-Min (Q) to get a vertex in Q with the minimum key, say $v \in Q$ (note that Extract-Min also removes v from Q); then (u, v) must be a light edge, where $u = v.\pi$. Now can we continue, after adding (u, v) to A and v to S? No! since the new S now contains v, we need to update the keys of those vertices in Q: recall that we need

$$w(v.\pi, v) = v.\text{key} = \min_{u \in S} w(u, v), \quad \text{for each } v \in V - S$$

We now present details of Prim's algorithm:

MST-Prim (G, w, r), where r is a vertex in G

1. set $A = \emptyset$ and $S = \{r\}$
2. for each $v \in V - \{r\}$ do
3. set $u.\text{key} = w(r, v)$ and $u.\pi = r$
4. Priority-Queue-Init $(Q, V - \{r\})$
5. while Q is nonempty do
6. $u =$ Extract-Min (Q)
7. for each $v \in \text{adj}[u]$ do
8. if $v \in Q$ and $v.\text{key} > w(u, v)$ then
9. set $v.\pi = u$ and Decrease-Key $(v, w(u, v))$

To see its correctness, check that by the end of each while-loop, the two attributes of every vertex v in Q still satisfy:

$$w(v.\pi, v) = v.\text{key} = \min_{u \in S} w(u, v)$$

The time complexity of the algorithm depends on the data structure we use for Q: its total running time is $O(n + m)$ plus the time needed for the initialization of Q plus the time needed for n Extract-Min and m Decrease-Key. If we use Heap (or Red-Black tree), total running time is $O(n + m + (n + m) \lg n) = O(m \lg n)$. If we use Fibonacci Heap (which we did not cover, Chapter 19), then each Extract-Min takes $O(\lg n)$ but each Decrease-Key takes $O(1)$ amortized time, so the total running time is $O(m + n \lg n)$.