2

Amortized Analysis and MSTs

2.1 *Minimum Spanning Trees*

Given a connected graph $G = (V, E)$, a *spanning tree* is a subgraph $T = (V, E')$ with $E' \subseteq E$ that has no cycles (so it is a *tree*), and has a single connected component (and hence is *spanning*). If each edge e has a cost/weight $w(e)$, the cost/weight of the tree T is $\sum_{e \in E'} w(e)$. The goal of the MST (minimum-cost spanning tree) problem is to find a spanning tree with least weight.

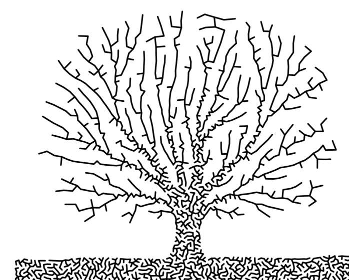

Figure 2.1: A minimum-cost spanning tree (abstract foliage). Opt(imization) art by Bob Bosch

In 1956, Joseph Kruskal proposed the following greedy algoritm that repeatedly selects the least-cost edge that does not form a cycle with previously selected edges. Stated another way: As always,

Algorithm 4: Kruskal's MST Algorithm

4.1 $T \leftarrow \emptyset$
4.2 Sort edges so that $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$.
4.3 **for** $i \leftarrow 1 \ldots m$ **do**
4.4   **if** $T \cup \{e_i\}$ ***does not contain a cycle*** **then**
4.5 $T \leftarrow T \cup \{e_i\}$
4.6 return T

we will ask the two questions: Is this algorithm correct? And how efficient is it?

Theorem 2.1. *Given any connected graph G, Kruskal's MST Algorithm outputs the minimum-cost spanning tree.*

Proof. The first observation is that the subgraph T is indeed a spanning tree: it is acyclic by construction, and it ultimately forms a connected subgraph. Indeed, if T contained a disconnected component C, then the connectivity of G means there is at least one edge between C and $V \setminus C$—and the first such edge would be added to T.

To show it is a minimum-cost spanning tree, define a set S of edges to be *safe* if *there exists* some MST that contains all edges in

S. We will prove that the edges in T maintained by the algorithm are always safe. So when the algorithm stops with a spanning tree T, the only MST containing T is T itself: so T *is* an MST.

To prove the safety of edges in T, we use induction. As a base case, observe that the empty set is safe. The following lemma shows the inductive step.

Lemma 2.2. *Suppose S is safe. If C is some (maximal) connected component formed by the edges of S, and e is the minimum-cost edge crossing from C to $V \setminus C$, then $S \cup \{e\}$ is also safe.*

Proof. Take any MST T^* containing S (but not e). If $e = \{u, v\}$, consider the u-v path in T^*. Since exactly one of the vertices $\{u, v\}$ belongs to C and the other not, there must be a unique edge f on this path with one endpoint in C and the other outside. This means $T' := T^* - f + e$ is another spanning tree. Moreover, since e had the least cost among all edges crossing the cut from C to $V \setminus C$, we have $w(e) \leq w(f)$. This means the new spanning tree T' has no higher cost, and hence is also an MST, showing that $S \cup \{e\}$ is also safe. □

Now each time we add an edge e_i in Kruskal's algorithm, the edge connects two different connected components C, C' (because it does not create any cycles). Since we consider edges in non-decreasing order of costs, it is the cheapest edge crossing from C to $V \setminus C$ (and also from C' to $V \setminus C'$). This means $T \cup \{e_i\}$ is also safe, hence we end with a safe set set, which is the MST. □

2.1.1 The Running Time

The algorithm statement above is a bit vague, because it does not explain how to check whether $T \cup \{e_i\}$ is acyclic. One simple way is to just run *depth-first search* on T to check if the endpoints of e_i are already in the same connected component: this would take $O(n)$ time in general. Since there are m edges, we get an $O(mn)$ runtime. There is also the time to sort the m edges, which is $O(m \log m)$, but that is asymptotically smaller than $O(mn)$ for simple graphs.

Simple graphs are those with no self-loops, and no parallel edges. We can always remove these in a linear-time processing. You can show that any simple graph has at most $\binom{n}{2}$ edges, and any connected graph has at least $n - 1$ edges.

But since we are the ones building T, we can store some extra information that can allow to do this cycle-checking much faster. We maintain an extra data structure, called the ***Set Union/Find*** data structure, that offers the following operations:

- **MakeSet**(u): create a new sington set containing element u.
- **Find**(u): return the "name" of the set containing element u. The name can change over time, and the only property we require from the name is that if we do two consecutive finds for u and v

(without any unions between them) then **Find**$(u) =$ **Find**(v) if and only if u and v belong to the same set.

- **Union**(u, v): merge the sets containing u, v.

Given these operations, we can flesh out the algorithm even more:

Algorithm 5: Kruskal's MST Algorithm (Again)

```
5.1 T ← ∅
5.2 for v ∈ V do MakeSet(v)
5.3 Sort edges so that w(e_1) ≤ w(e_2) ≤ ··· ≤ w(e_m).
5.4 for i ← 1...m do
5.5     let edge e_i = {u, v}
5.6     if Find(u) ≠ Find(v) then
5.7         Union(u, v)
5.8         T ← T ∪ {e_i}
5.9 return T
```

Apart from sorting m numbers (which can be done in time $O(m \log m)$ using MergeSort or HeapSort, say, this algorithm performs n ***MakeSet***, $2m$ **Find**, and $n - 1$ **Union** operations. It is easy to make sure that we can implement each of these operations to take time $O(n)$ per operation. But now we show how to implement them so that they take only $O(\log n)$ ***on average*** per operation! Formally we now show that:

Each **Union** reduces the number of components by 1 and there are n vertices, so the total number of unions is $n - 1$.

Theorem 2.3. *The Set Union/Find data structure has a list-based implementation where any sequence of M makesets, U unions, and F finds (starting from an empty state) takes time $O(M + F + U \log U)$.*

This is enough to ensure that the total runtime of Kruskal's algorithm is $O(m \log m) + O(m + n + n \log n)$; the first term dominates to give a net runtime of $O(m \log m)$.

In fact, we can implement the data strcture quite a bit better:

Theorem 2.4. *The Set Union/Find data structure has a tree-based implementation where any sequence of M makesets, U unions, and F finds (starting from an empty state) takes time $O(M) + O((F + U) \log^* U)$.*

Here the $\log^*$ function is the iterated logarithm, which is loosely the number of times the logarithm function should be applied to get a result smaller than 2. For details of this proof (and further improvements), see the notes on Union/Find from 15-451/651. Note, however, that the asymptotic runtime of Kruskal's algorithm does not improve, since the bottleneck is the $O(m \log m)$ time to sort m numbers.