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.

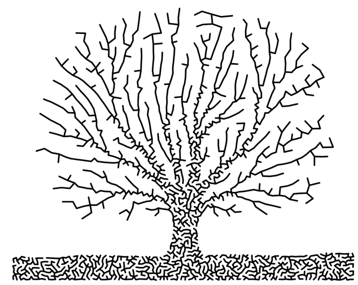

Figure 2.1: A minimum-cost spanning tree (abstract foliage). Opt(imization) art by Bob Bosch

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

$T \leftarrow \emptyset$
Sort edges so that $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$.
for $i \leftarrow 1 \dots m$ **do**
  if $T \cup \{e_i\}$ ***does not contain a cycle*** **then**
    $T \leftarrow T \cup \{e_i\}$
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. $\square$

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. $\square$

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)

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

2.2 Amortized Analysis

The basic idea of amortization is simple: you have a process where some operations are expensive, some others are cheap. You want to show that there is some α such that over any sequence of T operations, you pay at most $T \cdot \alpha$ for some value α. Then you say the amortized (or average) cost per operation is at most α.

2.3 The Binary Counter

You have a binary counter that starts off with all zeroes. Each time the counter increments by 1. So the first few settings are:

The cost of an increment is the number of bits that change. So the first few increments cost

$$1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1, 5, \ldots$$

What is the amortized cost over T operations? There are many ways to solve this:

2.3.1 Brute Force

First imagine that T is a power of 2, say 2^t. Then observe that the first 2^{t-1} costs look the same as the last 2^{t-1}, except for the very last operation costing one more. So if the sum of the first 2^t costs is S_t, we get

$$S_t = 2S_{t-1} + 1.$$

And $S_0 = 1$. Solving this gives $S_t = 2^{t+1} - 1$. And hence the amortized cost (per operation) is

$$\frac{2^{t+1} - 1}{2^t} \leq 2.$$

Then one can argue that if T is not a power of 2, then break the process into prefixes of length that are powers of 2, etc. But all this requires some work. Let's see other approaches.

2.3.2 Summing More Smartly

Observe that the lowest order bit changes each step. The second lowest-order bit changes once every other step. The third one changes once every fourth step. So the number of changes in T steps is at most

$$T + \lceil T/2 \rceil + \lceil T/4 \rceil + \ldots \leq 2T.$$

And hence the amortized cost per operation is at most 2.

2.3.3 The Banker's Method

Let's maintain the invariant that each 1 has a dollar bill sitting next to it, that can pay for changing it back to zero. This is vacuously satisfied at the start: we start with all zeros, so there are no 1s. Now each increment consists of changing some suffix of 1s to zeros, followed by changing a single zero to a 1. All the $1 \to 0$ transitions are paid for by the dollar bills, so we only need to pay for the single $0 \to 1$ transition, and to put down the dollar bill on it. Hence we pay \$2 per operation.

We call this the *banker's method*: we show that for each operation we pay \$$\alpha$, some of which is used to pay for the current operation and some to save for later, more expensive operations. You can think of this as keeping a bank account, and saving the extra amount from the cheaper operations to use for the expensive operations.

2.4 The Potential Function Method

Let us present a more general framework in which most amortized analyses can be placed. At each time, the system is supposed to be in some state, drawn from a collection Ω of states. We maintain a function $\Phi : \Omega \to \mathbb{R}$ called the ***potential function***. With an operation at time t the system moves from some state s_{t-1} to some other state s_t. This changes the potential of the state from $\Phi(s_{t-1})$ to $\Phi(s_t)$. Suppose the operation costs c_t. Then the *amortized cost* at time t is defined as:

$$A_t := c_t + \big(\Phi(s_t) - \Phi(s_{t_1})\big). \tag{2.1}$$

If we sum this over all times, the telescoping sum gives

$$\sum_{t=1}^{T} A_t = \sum_{t=1}^{T} c_t + \Phi(s_T) - \Phi(s_0).$$

Now suppose we can show the following three things:

1. The potential starts at zero (i.e., $\Phi(s_0) = 0$),
2. it is non-negative, so that $\Phi(s_T) \geq 0$, and
3. $A_t \leq \alpha$ for all times t.

Then we get

$$T\,\alpha \geq \sum_{t=1}^{T} A_t = \sum_{t=1}^{T} c_t + (\geq 0 - 0).$$

In other words, the total cost is at most $T\alpha$, and hence the amortized cost is at most α.

This all sounds very abstract: I like to think of $\Phi(s)$ as being the bank account when you get to state s. The expression on the right of (**??**) is the actual cost of the operation, plus the change in the bank account. That is what we have to pay per operation. So sometimes the operations are cheap but the bank account goes up, and we need to pay for this increase. At other times, the operations are expensive, and then the bank account goes down, with this decrease helping to pay this larger cost (and us paying only the difference).

For example: if the current number in the binary counter is s, then define

$$\Phi(s) = \text{number of 1s in the binary representation of } s.$$

Now if we changing from s to $s+1$ requires us to flip f bits, $f-1$ of these flips must be $1 \to 0$, and the last one is $0 \to 1$. So the number of 1s decreases by $f-2$. This means the amortized cost at each timestep is

$$A_{s \to s+1} = f - (f-2) = 2.$$

Observe that the potential here matches the total money in the bank from the previous analysis.

2.4.1 *Why and How*

You may ask: why even talk about potential functions, why not just use the banker's method all the time? The answer is that potentials are a more general idea, and sometimes they will be more useful. We will see some later in the course.

And the next natural question: how do you come up with a potentail function? This is more difficult, there is no silver bullet. The only suggestion is to look closely what the algorithm

2.5 *The List-Based Union-Find Data Structure*

Recall that the operations are:

1. `Makeset`(e): create a singleton set $\{e\}$, costs 1.

2. `Find`(e): return the name of the set containing e, costs 1.

3. `Union`(e, f): merges the sets A, B containing e, f, costs $\min(|A|, |B|)$.

With n elements in total, observe that the *worst-case* cost of an operation could now be $\Omega(n)$. Indeed, if we union two lists of length $n/2$, we pay $n/2$. Of course, building up these long lists from scratch requires many operations, so we may hope the amortized cost is low. This is precisely what we show next.

2.5.1 The Amortized Cost

Theorem 2.5. *If we perform n `Makesets`, m `Finds` and u `Unions` using the list-based union-find data structure, the total cost is at most*

$$m + O(n \log n).$$

Here are two proofs of this theorem:

1. Using the banker's method: each `Makeset`(e) brings $\log_2 n$ dollars with it, which is stored with the element e. Each time e belongs to the smaller set involved in a `Union`, it pays for this operation. But the size of the set containing e at least doubles, so e has to pay at most $\log_2 n$ times.

2. Using potentials: if the sets at the current state s have size $n_1, \ldots, n_k$, let
$$\Phi(s) = \sum_i n_i \log_2(n/n_i).$$
Each makeset costs $log_2 n$, the finds cost 1. And when we replace sets of size $n_i \leq n_j$ by a new set of size $n_i + n_j$, the amortized cost for each union is
$$\begin{aligned} & n_i + (n_i + n_j)\log_2 \frac{n}{n_i + n_j} - n_i \log_2 \frac{n}{n_i} - n_j \log_2 \frac{n}{n_j} \\ & = n_i - n_i \underbrace{\log_2 \frac{n_i + n_j}{n_i}}_{\geq 1} - n_j \underbrace{\log_2 \frac{n_i + n_j}{n_j}}_{\geq 0} \\ & \leq 0. \end{aligned}$$
Hence, the total cost is $O(m + n \log n)$.

One can improve both the analyses a bit and show a total cost of

$$m + n + O(u \log n).$$

Indeed, each makeset only costs a dollar now, but inctead each union operation brings in $\log_2 n$ dollars, with the first union involving element e giving its $\log_2 n$ dollars to e. Since the number of unions is at most n (Be sure you see why?), $m + n + O(u \log n)$ is a better bound than $m + O(n \log n)$.