More MST Algorithms: Prim and Kruskal's Algorithm[1]

In this note we describe two other MST algorithms. Their correctness follows from the same property we used for designing Borůvka's algorithm. Namely,

Theorem 1 (Cut Crossing Property)**.** Assume all costs are distinct. Let $S \subseteq V$ be any subset of vertices. Recall, $\partial S := \{(u, v) : u \in S, v \notin S\}$. Then, the edge in ∂S with the minimum cost **must** be in every minimum spanning tree.

The idea is similar to Boruvka's in that the algorithm proceeds in phases, and in each phase the algorithm recognizes some new edges of the MST. It does so by recognizing edges e which are the minimum cost edges in ∂S for some S. Prim's algorithm discovers exactly one edge per phase (thus, there are $|V| - 1$ phases in all). The algorithm maintains a subset $S \subseteq V$ and labels $\mathsf{wt}[v]$ for every vertex $v \in V$. The invariant we will maintain is that $\mathsf{wt}[v]$ will denote the weight of the cheapest edge from S to v. $\mathsf{parent}[v]$ will point to $u \in S$ such that (u, v) is this minimum cost edge. That is, $\mathsf{wt}[v]$ will be $\min_{u \in S:(u,v) \in E} c(u, v)$. If there is no such edge, then we define $\mathsf{wt}[v] = \infty$. Initially, $S = \emptyset$ and $\mathsf{wt}[v] = \infty$ for all $v \in V$.

In each phase, the algorithm adds the *cheapest* edge in ∂S to F. This is found by looking at all $\mathsf{wt}[v]$ for $v \notin S$ and taking the one with the minimum value. The edge $(\mathsf{parent}[v], v)$ is added to F and v is added to S. As soon as v enters the subset, it modifies $\mathsf{wt}[w]$ for all $w \notin S$ with $(v, w) \in E$ as $\mathsf{wt}[w] = \min(\mathsf{wt}[w], c(v, w))$. This is much like Dijkstra's algorithm; indeed, the running time of Prim is exactly the same as that of Dijkstra. That is, if implemented with Fibonacci heaps, it takes $O(|E| + |V| \log |V|)$ time.

```
1:  procedure PRIM(G = (V, E), c : E → ℝ):
2:      Initialize S ← ∅, wt[v] = ∞.
3:      Pick an arbitrary "root" vertex s ∈ V and set wt[v] = 0.
4:      while S ≠ V do:
5:          Let v be the vertex ∉ S with smallest wt[v].
6:          S ← S + v.
7:          for all neighbors u of v not in R do:  ▷ The distance labels set for only vertices outside R
8:              if (wt[u] > c(v, u)) then:
9:                  Set wt[u] ← c(v, u).
10:                 Set parent[u] ← v.
```

Theorem 2. PRIM's algorithm finds the minimum spanning tree of a graph in $O(|E| + |V| \log |V|)$ time, if implemented using Fibonacci heaps.

The next algorithm is probably the most famous algorithm for finding the MST and the most succinctly state-able one: go over all the edges in an increasing order of cost and pick it in F if it doesn't form a cycle with the existing edges. This is called KRUSKAL's algorithm.

[1] *Lecture notes by Deeparnab Chakrabarty. Last modified : 19th Mar, 2022*
These have not gone through scrutiny and may contain errors. If you find any, or have any other comments, please email me at deeparnab@dartmouth.edu. Highly appreciated!

```
1: procedure KRUSKAL(G = (V, E), c : E → ℝ):
2:     F ← ∅.
3:     Sort edges in increasing order of cost.
4:     for edge e in this order do:
5:         if F ∪ e has no cycles then: F ← F + e.
6:     return F.
```

However, the succinct description might be the only thing going for this algorithm. Note that a naive implementation takes $O(|E| \log |E| + |E||V|)$ time with the bulk of time used to check if $F \cup e$ contains a cycle (which one can do by say doing a DFS on F taking $O(|V|)$ time). This is of course wasteful and we will soon see how to get the time down to sorting time, which is $O(|E| \log |E|) = O(|E| \log |V|)$ time.

Before we go to the implementation, why is the above algorithm correct? Is it always picking an edge which is the minimum cost edge in ∂S for some S? The answer is yes. Consider the tth edge the algorithm picks. Just before that time, suppose the content of F is F_{t-1} and the tth edge is $e_t = (u, v)$. Note that u and v must be in different connected components of $G[F_{t-1}]$ for otherwise we would have a cycle. Let U be the connected component of F_{t-1} which contains u. We claim that (u, v) is the cheapest edge in ∂U. This is because if there was a cheaper edge $e' = (u', v')$, then that edge must have been considered before. This edge doesn't form a cycle with F_{t-1}, and therefore would not have formed a cycle earlier either (we only had a subset of edges earlier). And yet it was not picked. Contradiction.

We now describe a faster implementation of Kruskal's algorithm using a data-structure called the UNION-FIND data structure. This data structure works over a universe of elements and in our case this universe is V, the vertices of the graph. It maintains a collection of *disjoint sets* of V which partitions the universe. There are two operations. One is FIND(v) which returns the index of the set which contains the element v. And the other is UNION(u, v) which merges the sets FIND(u) and FIND(v). We will show how to maintain this data structure such the *total* time spent on UNION is at most $O(n \log n)$, where n is the size of the universe. Believing this for the time-being, we should be ableto see how one can get a $O(|E| \log |V|)$ implementation of Kruskal's algorithm. Indeed, the sets we maintain are the connected components of F, and whenever we add an edge (x, y), we perform a UNION. In particular,

```
1: procedure KRUSKAL-WITH-UNION-FIND(G = (V, E), c : E → ℝ):
2:     F ← ∅.
3:     Sort edges in increasing order of cost.
4:     for edge e = (x, y) in this order do:
5:         if FIND(x) ≠ FIND(y) then: ▷ x and y are in different components
6:             F ← F ∪ e.
7:             UNION(x, y).
8:     return F.
```

Now to the UNION-FIND data structure. Here is one implementation. We maintain a *set* object which contains (a) an index/name, (b) a list of elements it contains, and (c) its size. So, if S is a set, we have S.name as its index, S.elem to be a list of elements, and S.size to be the size of this array. We initially create $|V|$ sets initializing the ith sets S_i.name $\leftarrow S_i$ as S_i.elem $\leftarrow [v_i]$, and S_i.size $= 1$. We also maintain pointers such that every vertex v in S.elem points to this object S.

The FIND operation is trivial; on input $v \in V$, we just use the pointer from v to find the set which contains it. When we call UNION(x, y) when x and y point to two different sets, we first let $S_i =$FIND(x)

and S_j =FIND(y). Then we compare S_i.size and S_j.size. If S_i.size $\leq S_j$.size, then we do the following (a) we append S_i.elem to S_j.elem, (b) we set S_j.size $\leftarrow S_j$.size + S_i.size, and (c) for every $v \in S_i$.elem, we move their pointers from S_i to S_j. At this point, we can delete S_i. The time taken by UNION is $O(S_i.\text{size})$.

Claim 1. Consider a seqeunce of UNION operations. The total time spent is $O(n \log n)$.

Proof. The sort of reasoning we are going to do is called *amortized analysis*. Although a single UNION step can take $\Theta(n)$ time, and there can be n UNION calls, nevertheless the total time is not $\Theta(n^2)$ but will be much smaller. This may sound magical, but it is just saying that the $\Theta(n)$ time is worst-case and can't happen much of the time.

The proof is really a simple charging argument. Whenever two sets A and B are being union-ed, we take approximate $\min(|A|, |B|)$ time. Let us "charge" this time to the elements of the smaller set. More precisely, if $|A| \leq |B|$, we put 1 unit of charge on every element of A. We do this at every UNION call. So, at the end of the sequence of UNION calls, every element will have some non-zero charge, and the total time taken is equal to $O(C)$, where C is the *total* charge in the system.

It suffices to prove $C \leq n \log_2 n$. Indeed, every time an element obtains a charge, its set (what we get when we call FIND) more than doubles in size. Indeed, UNION(A, B) will move elements of A to $A \cup B$, which is of size $\geq 2|A|$ since $|B| \geq |A|$. This doubling cannot occur more than $\log_2 n$ times, and so the total charge is at most $n \log_2 n$. □