Graphs : Minimum Spanning Trees[1]

In this lecture, we look at a different kind of graph problem which can be solved efficiently. The input is an *undirected* graph $G = (V, E)$ with costs $c(e)$ on the edges $e \in E$; the costs can be positive or negative. The goal is to find a *spanning forest* of minimum cost. More precisely, we want to find a collection of edges $F \subseteq E$ such that $c(F) := \sum_{e \in F} c(e)$ is minimum, F contains no cycles, and the connected components of the graph induced by the edges in F is the same as the connected components of the original graph. For simplicity's sake, we are going to assume G is a connected graph in this lecture, and thus F is a tree (recall, a tree is a connected forest) which contains all vertices of G.

MINIMUM SPANNING TREE
Input: A connected graph $G = (V, E)$, costs $c(e)$ on edges $e \in E$.
Output: A spanning tree $T = (V, F)$ with $c(F)$ as small as possible.

Shortest paths captured the minimum cost path from s to a vertex $v \in G$. Thus, shortest paths captured information "local; to the vertex s. The minimum spanning tree problem is capturing something more "global".

Remark: *At this point, it may be a good guess to think that the shortest path tree that, say* DIJKSTRA*, returns could be the minimum spanning tree (at least when costs are positive). Come up with an example to show this is not the case. Again, the shortest-path tree only cares about paths to s and not the total cost; use this to construct your example.*

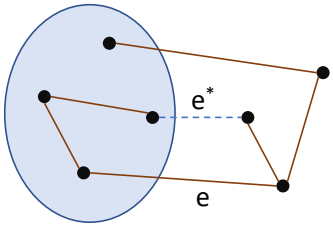
For the sake of simplicity, we are going to assume all the costs $c(e)$ are *distinct*. This is without loss of generality, really; one can either add a teeny-weeny jiggle to every cost (we are not assuming costs are integers), or one can simply set up a tie-breaking rule between edges. That is, we set an arbitrary but fixed numbering on all the edges, and if two edges have the same cost, then the one with the smaller number is said to be cheaper. This allows us to say the following powerful theorem which is used in every spanning tree algorithm.

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.

Before we embark on the proof, please read the statement again and realize its power. We don't know (yet) what the minimum spanning tree is. But the theorem says pick *any* subset you want. For instance, pick a singleton vertex $\{v\}$. Then just looking at the edges that cross this subset, you will get one edge of the minimum spanning tree! In particular, the *minimum* cost edge incident on any vertex must be in the spanning tree. I hope that if you understand the power of the above statement, an algorithm to find the MST will come leaping to your head — pick the minimum cost edges incident on every vertex, "contract" the graph along these edges (how to do this may not be clear yet), and repeat. This is the algorithm we study. Let's prove this powerful theorem.

[1] *Lecture notes by Deeparnab Chakrabarty. Last modified : 2nd Mar, 2025*
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!

Proof. Suppose not. Let $T = (V, F)$ be a minimum spanning tree. Let $e^* = (u, v)$ be the edge in ∂S of minimum cost. Suppose $e^* \notin F$. Now note that $T = (V, F)$ is a *spanning tree*. In particular, every vertex in S has a path to every vertex not in S using only the edges of F. In particular, there is a path from u to v which uses only the edges of F. Since this path originates at $u \in S$ and terminates at $v \notin S$, it must contain *at least* one edge $e := (x, y)$ with $x \in S$ and $y \notin S$. That is, $(x, y) \in \partial S$. By the definition of e^*, $c(e) > c(e^*)$ since the costs are distinct. See the figure below for an illustration; S is the vertices in the blue oval, the red solid edges are the purported edges of T while the blue dashed edge is the minimum cost edge in ∂S.



The key claim is to argue that $F' := F - e + e^*$ is a spanning tree; it clearly has $c(F') < c(F)$ and that would give a contradiction. Also note that $|E(F')| = |E(F)| = n - 1$ where n is the number of vertices in G. Thus, by the "tree theorem" (which you see in CS30), to prove F' is a spanning tree, all we need to prove is that F' is connected.

To show F' is connected, pick any two vertices a, b. We show a walk from a to b in F' which would imply a path. Consider the path q from a to b in F. If this path does not contain (x, y) then q is a valid path in F'. So, we may assume $(x, y) \in q$. So, in q we go from a to x, take (x, y), and then go from y to b. In F', we walk from a to x, then we use the portion of the path p from x to u to get to u in F', then we take the edge (u, v), walk from v to y again using p, and then move from y to b using q. □

Now we are ready to concretely state the algorithm alluded to above. This algorithm is called Borůvka's algorithm. The algorithm proceeds in phases, and in each phase it starts with a bunch of connected components, and finds the minimum cost edges straddling these components. Theorem 1 asserts all these edges must lie in the minimum spanning tree, and so this algorithm picks them and proceeds to the next phase.

```
1: procedure BORŮVKA-MST(G = (V, E), c):
2:     Initialize F ← ∅ and fcomp ← n.
3:     (fcomp, Fcomp[1 : n]) ← FINDCONNCOMP(F).
4:     ▷ Returns (a) number of connected components, (b) for each vertex index of connected component it is in.
5:     while fcomp > 1 do:
6:         F' ← FINDCHEAPESTEDGE(Fcomp) incident to every connected component.
7:         F ← F ∪ F'
8:         (fcomp, Fcomp[1 : n]) ← FINDCONNCOMP(F).
```

To complete the description of the algorithm, we need to describe how FINDCONNCOMP and FINDCHEAPESTEDGE are implemented. The former is something you have seen before; we use DFS which returns fcomp and Fcomp[v] for every vertex indicating the number of the connected component it is in. To implement FINDCHEAPESTEDGE, for each component one maintains the cheapest edge crossing the

component. More precisely, if U is the set of vertices in this connected component, it maintains the cheapest edge in ∂U. This is initialized to empty. One then performs a linear scan of all the edges. While scanning edge (u, v) one checks (a) if u and v are in different components, and (b) if so, checks if $c(u, v)$ is cheaper than the current cheapest edge crossing each component. If so, it modifies the cheapest edge.

```
1:  procedure FindCheapestEdge((fcomp, Fcomp)):
2:      ▷ Returns a collection of fcomp edges F' with possible duplicates.
3:      ▷ These are the cheapest edges straddling the fcomp many components.
4:      Maintain Cheap[1 : fcomp] initialized to ⊥.
5:      ▷ Cheap[i] will contain the cheapest edge in ∂U_i where U_i is the set of vertices in the ith
        connected component.
6:      for all edges (u, v) ∈ E do: ▷ A single scan of all the edges
7:          if Fcomp[u] ≠ Fcomp[v] then:
8:              Let i ← Fcomp[u] and j ← Fcomp[v].
9:              if c(u, v) < c(Cheap[i]) then:
10:                 Set Cheap[i] = (u, v)
11:             if c(u, v) < c(Cheap[j]) then:
12:                 Set Cheap[j] = (u, v)
13:         else: ▷ The edge (u, v) should be deleted as u and v lie in the same component
14:             E ← E \ {(u, v)}
15:     return {Cheap[i] : i ∈ [1 : fcomp]}
```

The running time[2] of FindCheapestEdge is $O(|E|)$. If the graph G is connected, then FindCheapestEdge returns fcomp edges. However, there may be duplicates. Indeed, if $c(u, v)$ is the cheapest edge among all edges which straddle components, then (u, v) will be returned twice, once as Cheap$[i]$ and once as Cheap$[j]$ where $i =$ Fcomp$[u]$ and $j =$ Fcomp$[v]$. However, the algorithm returns *at least* fcomp$/2$ distinct edges simply because any returned edge can be Cheap$[i]$ for at most two i's. This implies

Lemma 1. The total number of **while** loops in Borůvka-MST is $\lceil \log_2 |V| \rceil$.

Proof. From the argument preceding the lemma we see that after a single while-loop, the value of fcomp becomes half or less. More precisely, fix a while loop, and say k was the value of fcomp before the while loop, and ℓ is the value after. Then $\ell \leq k/2$. This is simply because $k = |V| - |F|$ and $\ell = |V| - |F| - |F'| \leq k - k/2$ since $|F'| \geq k/2$. Since fcomp starts off as $|V|$, the number of iterations is at most $\lceil \log_2 |V| \rceil$. □

Theorem 2. The minimum cost spanning tree of an undirected graph can be found in $O(|E| \log |V|)$ time using Borůvka-MST algorithm.

Remark: *The above is a worst-case bound; the number of while loops can be way smaller. In fact, it is a nice exercise to come up with an example where the number of phases is indeed* $\lceil \log_2 |V| \rceil$.

[2] This is pessimistic since we aren't even accounting for the fact that edges are deleted over iterations. Note that in every iteration, we delete *at least* $|V|/2$ edges.