Lecture 18: Minimum Spanning Tree

Now minimum spanning tree (ch. 23). For a connected (undirected) graph G, find a tree that connects all the nodes such that the total edge costs are the smallest. The well-known Kruskal’s algorithm is a greedy algorithm: first sort the edges by edge weights; then for each edge, add edge if not cycles is created by this edge. We discuss in class to claim this algorithm will always lead to a spanning tree (see your notes/textbook). Question: is this optimal? Yes, and we ow show each added edge is safe to add: safe means we are not making mistakes at that step (the edges we picked so far belong to some MST) at each step.

We first assume all edge costs are distinct. A fundamental property of graph cut: let S, V − S be partitions the nodes, and edge e = (v, w) is the cheapest edge across S and V − S (say v ∈ S). Claim: every MST contains e. Proof: consider spanning tree T does not contain e. Will prove T is not MST. by the exchange argument (used several times before fro greedy algorithm): will find an edge e' of T that is most costly than e and we will replace e' with e to get a new cheaper T’. Now, since T does not have e = (v, w), There is a path in T from v to w. Since v and w on different partition, we follow this path from v to w and we will leave S at node v' and enter V − S at w' (for the first time). And e' = (v', w') belongs to T. Now, we remove e’ and include e instead. Claim: what we get is a spanning tree. First it is tree: still connected: each pair of node can still reachable (now going through e instead of e’). It is also not hard to show what we have is acyclic. The changed tree T is less costly.

Now we use this property to show Kruskal’s algorithm is correct (meaning optimal). Consider one edge e = (v, w) added by the algorithm. We let S be the nodes reachable from v by the selected edges by the algorithm so far. In class, we show this S works: first v in V but not w (why?). Also, e must be the first edge between S and V − S (why?). Thus, e is the cheapest between S and V − S. So e belongs to every MST.

The second algorithm for MST is Prim’s algorithm. The idea is to grow a MST from a node s, and each time, add the cheapest edge to connect a new node from the partial tree (and thus each step is a tree). The Prim’s algorithm maintains a set S be the nodes connected by the current spanning tree so far. Initially, S = {s}. While the size of S is less than n, we find a node v ∉ S. s.t. cost(u, v) is minimized for some u in S. We add v to S. The correctness follows the cut property. The selection of new node is done using a priority queue which represents for each un-connected node, the cost of linking to the partial tree. Read the textbook to make sure you know how the connection cost is maintained. Time analysis: suppose use binary heap to implement a priority queue. We need O(n) to initialize. There are O(n) iterations, each for a new node. Selecting this new node takes \( \log(n) \) time using the heap. Then for each node, need to update their neighbors, which takes total \( O(m) \) since the total length of the adjacency lists is \( 2m \). One update operation will take \( O(\log n) \) each. So total time: \( O(n \log n + m \log n) = O(m \log n) \). Note this can be improved using more advanced data structure. As comparison, the Kruskal’s algorithm will run slightly slower. See the textbook for its time analysis.

Lecture 19: Shortest Path

We now move to Chapter 24: the shortest path problem. This is a classic graph algorithm problem: given a weighted directed graph G(V, E) and a node s (the source), we want to find the shortest path from s to each node u. The length of path P is the sum of weights of edges of path. The shortest path between nodes u and v is the smallest weight of all possible paths between u and v). For now assume edge weights are positive.

Our first algorithm, called Dijkstra’s algorithm, has some similarity as the Prim’s algorithm (but of course for different problems). We start from s. We grow a region which contained nodes whose distance to s are already computed. The key step is, when growing the region, we always pick the nearest node (called v, which is not in the region yet) to s. Why do we choose this order? It is because that particular node v has useful properties. First, let us consider a shortest path from s to v, where the last edge is (u, v). It is not hard to see u must be in the region. Why? Think about it. This allows us to extend the region to include v using a single edge (called extension). But what about there are many choices to extend? The simple solution is: pick the node with smallest of all possible extension, and add it to the region. Why does this give shortest path to v? To think about this, consider what if there exists an even shorter path from s to v, and that will mean. Think about this yourself. Then read the textbook for the detailed algorithm. The algorithm has two aspects that worth attention. We also analyzed the running time for this algorithm.
There are more properties of shortest path. First, let $P$ be the shortest path from $s$ to $t$, then any *segment* of $P$ must also be the shortest path for the two end nodes of that segment (why?). A key property is called path relaxation property, which is essentially the last step of the Dijkstra’s algorithm is a recurring in shortest path. In particular, Relax$(v,z,w)$ would update distance info of neighbor $z$ of $v$. Intuition: suppose $(v,z)$ is an edge. Then, distance from $s$ to $z$ cannot be bigger than dist from $s$ to $v$ and then go from $v$ to $z$. There are three properties of relaxation: (1) if $d(s,v)$ is the length of the shortest path from $s$ to $v$, and the path from $s$ to $v$ and then to $w$ is the shortest path from $s$ to $w$, then $d(w)$ is set optimally after relaxation. (2) Relaxation is harmless: even if $d(v)$ is not yet optimal, it will not cause problem. That is, it may not set to optimal value (higher) and then later when $d(v)$ becomes optimal you can still get to optimal. (3) let a shortest path $P$ contain nodes $s, v_1, ... v_k$ in that order. Then if we relax each edge along $P$ in this order, then we get correct $d(s, v_k)$ in the end, no matter what other relaxation performed in addition (intermixed with other relaxation). Think about it and convince yourself.

Based on the above observations, we know how to find shortest path on a directed acyclic graph (DAG). Basically, we first topologically sort the nodes in the DAG; and then perform relaxation of each outgoing edge of nodes in the topologically order. This works because all shortest paths will be relaxed in the right order and also relaxation is harmless. This gives a simple $O(n + m)$ algorithm (where $n$ is the number of nodes and $m$ is the number of edges).

We note that for the case when edge weights can be negative, Dijkstra’s algorithm no longer applies (why?). We show in class a simple graph where Dijkstra’s algorithm does not work.