Lecture 20: MST and Shortest path

The second algorithm for MST is the 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 \( v \notin S \) s.t. \( \text{cost}(u, v) \) 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: \( O(m \log(n) + m \log(n)) = O(m \log(n)) \). 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.

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. First, the way of its initialization is a common step in shortest path problem. Second, the way of update (extension) is also very useful (called relaxation). We also analyzed the running time for this algorithm.

Lecture 21: Shortest path and NP-completeness.

We now finish the shortest path problem by studying a different algorithm for the case when edge weights can be negative. In this case, Dijkstra’s algorithm no longer applies (why?). The following two observations are useful. (a) If we apply relaxation on a shortest path from \( s \) to \( u \) in the order of the edges of this path, then these relaxation gives optimal \( d(u) \) value. To get some intuition, consider the example where you have 4 nodes along the path, and each edge weight is 1. Experiment with it to see what happens if you apply relaxation along the path or out-of-order. (b) Relaxation is harmless: if we mix some other relaxation with the key relaxation as in (a), we still will get optimal \( d(u) \). Think about it: why it is correct. These two cases allow us to solve two problems. (i) For a directed acyclic graph. In this case, we first apply topological sorting and then we apply relaxation with the order of edges according to their appearance in the order. The correctness is due to the fact that every path will have edges in one direction. Read the book chapter to make sure you understand this. (ii) For a general graph, we do not know which is the optimal order to apply relaxation. The nice observation by Bellman-Ford algorithm is, if we apply \( n - 1 \) rounds of relaxation
(while in each round we relax every edge), then we must get shortest path distance to each node. Read the textbook to ensure you understand why.

We now switch to Chapter 34: NP-completeness. We define $\mathcal{P}$ as the set of problems that have polynomial-time algorithms. We define $\mathcal{NP}$ as the set of problems that we can verify proposed solutions in polynomial-time. The key concept discussed in class is the concept of verifying a proposed solution. In graph 3-coloring problem, for example, a proposed solution is the coloring of nodes of the graph. Remember in this problem we need to assign one of the three colors to the nodes s.t. no edge has the same color at its two end nodes. We can easily check whether the coloring satisfies the coloring property. So this problem is in $\mathcal{NP}$. On the other hand, no one knows polynomial-time algorithm to find a legal coloring for a graph. So it is not known whether this problem is in $\mathcal{P}$ or not. Another example is the Hamiltonian cycle problem. We are given a graph, and we want to find a path that visits each exactly once and return to the starting node. Again, no one knows efficient algorithms for this problem, while it is easy to see this problem is also in $\mathcal{NP}$. We will see more problems next week.