Lecture 19: MST and Shortest Path

First, I showed in the class that if we perturbing the edge weights small enough, then the MST on the perturbed graph must be the same as the one in the original graph. In class, I showed a more detailed analysis.

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

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 $z$ can not 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 $s$, $v_1$, $v_2$, ..., $v_k$ 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).
Lecture 20: Shortest Path and FFT

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 is it 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 \textit{every} edge), then we must get shortest path distance to each node. Read the textbook to ensure you understand why. The algorithm runs in \( O(|V||E|) \) time.

We also 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.

Now we switch to fast Fourier transform (FFT). In general, given two polynomial: \( A(x) = a_0 + a_1x + a_2x^2 + \ldots + a_n x^n \), \( B(x) = b_0 + b_1x + b_2x^2 + \ldots + b_n x^n \). \( A(x)B(x) = c_0 + c_1x + c_2x^2 + \ldots + c_{2n} x^{2n} \). Here, \( c_k = \sum_{i=0}^{k} a_i b_{k-i} \). A naive algorithm will take \( O(n^2) \) time. We now show that the FFT algorithm will run faster.

We need some background on polynomial. A polynomial of degree \( n-1 \) can be represented by its values at \( n \) distinct points. That is, to specify a polynomial, we either give the coefficients \( a_i \) or give values \( A(x_0), A(x_1), \ldots, A(x_n) \). A good property of the value representation is: multiplication of \( A(x) \) and \( B(x) \) is easy (by just multiplying pointwise the values if \( A(x) \) and \( B(x) \) are evaluated at the same points. Now, how can we efficiently convert the coefficient representation to value representation? Naively, we need to evaluate \( n \) points for degree-\( n-1 \) polynomial, and each point needs \( O(n) \) time. So total time is \( O(n^2) \). FFT allows faster computation due to the shared of computation.

Lecture 21: FFT and concepts of NP

The key idea is, if we use +/-\( x_1 \), +/-\( x_2 \), ..., +/-\( x_{n/2} \) for \( n \) points (for a polynomial of \( n-1 \) degree), then there are a lot of sharing since even power of +/-\( x \) are the same. For this purpose, we separate the powers into odd/even as follows: \( A(x) = A_e(x^2) + x A_o(x^2) \). Now, the degrees of \( A_e(x) \) and \( A_o(x) \) are \( n/2 - 1 \) (assume \( n \) is even) at most. Now for +/-\( x_i \), \( A(x_i) = A_e(x_i^2) + x_i A_o(x_i^2) \), and \( A(-x_i) = A_e(x_i^2) - x_i A_o(x_i^2) \). So to evaluate +/-\( x_1 \), +/-\( x_2 \), ..., +/-\( x_{n/2} \), we only need to evaluate \( A_e(x) \) and \( A_o(x) \) (which are at most \( n/2 - 1 \) degree) at only \( n/2 \) points: \( x_1^2, x_2^2, \ldots, x_{n/2}^2 \). This is exactly divide and conquer. Following the Master theorem, the running time is \( O(n \log n) \).

The remaining problem is how we will pick the points so we can carry out the divide and conquer. We will choose the \( n \) points to be complex roots of equation \( x^n = 1 \). Recall that these roots are: \( 1, \omega, \omega^2, \omega^3, \ldots, \omega^{n-1} \), where \( \omega = e^{2\pi i/n} \). This property shows why these \( n \) points make sense: suppose \( n \) is even, then \( w^{n/2} = -w \). (that is, there are +/- pairs in these points). Moreover, when we square these points for \( x^n = 1 \), we get \( n/2 \) points for \( x^{n/2} = 1 \). Why? Simple. \( x^n = 1 \). Then \( (x^n)^{n/2} = 1 \). This implies that suppose \( n \) is a power of 2, then we just pick the points as the \( n \) roots of \( x^n = 1 \), and the divide and conquer works perfectly: each time we have +/- pairs and also when we square we get \( n/2 \) values which again are all the roots for \( x^{n/2} = 1 \).

I omit the algorithm for FFT here (see page 911) of our textbook. Now what about our goal of polynomial multiplication? Given two polynomials \( A(x) \) and \( B(x) \), we convert them to value representations using FFT. Suppose \( A(x) \) and \( B(x) \) have degree \( n \). We can treat both of them as \( 2 \) \( n \) degree by adding \( n \) high order 0 to it. Then we get evaluation of \( A(x) \) and \( B(x) \) at \( 2n \) roots of \( x^{2n} = 1 \). And then, \( C(x) = A(x)B(x) \) has value expression as the pointwise multiplication of \( A(\omega^i)B(\omega^j) \). This is the value expression of the product of \( A \) and \( B \), this can be done in \( O(n \log n) \).

We now switch to Chapter 34: NP-completeness. We define \( P \) 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 $\mathcal{NP}$.

The concept of NP is important: a problem is in NP if this problem can be verifiable in polynomial-time. Note: we are not asking for efficient finding of the solution, but rather just verifying whether a proposed solution is legal or not.

As an example, consider the problem of graph 3-coloring, where we are given a graph $G$ and we want to find coloring each node with three colors s.t. no edge has same color on its two ends. No efficient algorithm is known for this problem. So instead of looking for a valid coloring, you are given a coloring (each node is colored), and this coloring is a proposed solution. Is this a legal solution (i.e. the coloring)? It is easy to see we can verify this coloring property in polynomial time: for each of the $m$ edges, we check to see if its two nodes are colored the same or not; if any edge has same color on its two ends, the solution is not legal. If no such violation is found, the proposed solution is good. For a graph with $m$ edges, this takes $O(m)$ time and so verification can be done in polynomial time. That is, graph 3-coloring is in NP.

Our next problem is the satisfiability problem. We are given a boolean formula in so-called Conjunctive normal form (CNF). A CNF formula contains several clauses (connected by logical and), where in each clause we apply logical or to connect several literals. Literal means a boolean variable $x$ or its negation $\overline{x}$. Example:

$$\Phi = (x \lor y \lor z) \land (x \lor \overline{y}) \land (y \lor z) \land (z \lor \overline{x}) \land (\overline{x} \lor \overline{y} \lor z)$$

Satisfying truth assignment: a setting of boolean variables s.t. each clause is evaluated to be true (i.e. for each clause, at least one literal is true). In class, I explained that this formula does not have satisfying truth assignment. Study this if you forget about the argument. The satisfiability problem asks whether there exists satisfying truth assignment for a given formula. Again, no efficient algorithm is known for satisfiability (SAT) problem. But verifying a proposed truth assignment can be easily done in polynomial time.

Another example is Hamiltonian cycle: given a directed graph $G(V,E)$, find a path that visit each node exactly once and return to the origin. Again, no efficient algorithm is known for this problem. Now what if I proposed a tour of $v_1, v_2, v_3 \ldots v_n$ as the cycle? It is not hard to see we can easily verify whether this proposed solution is legal or not in polynomial time. Thus, HAM-CYCLE is also in NP.

We will see more problems next week.