CMSC 351 - Introduction to Algorithms
Spring 2012
Lecture 21

Instructor: MohammadTaghi Hajiaghayi
Scribe: Rajesh Chitnis

1 Introduction

In this lecture we will look at Minimum Spanning Trees and the concept of NP.

2 Prim's algorithm for Minimum-Cost Spanning Tree (MST)

Say there is a set of computers in a office or a set of sites for VPN or a set of cities that we want to connect to each other. We can model all these with an undirected graph whose edges represent connections and the weights are the lengths. We want to find a connected subgraph with min sum of edge lengths. Note that the subgraph should be a tree. More formally, the problem is as follows: Given an undirected connected weighted graph $G = (V, E)$, find a spanning tree that connects every vertex with minimum cost. We now give an algorithm due to Prim for thus problem. Note that this is a greedy algorithm.

Algorithm 1 Prim(G,v)

```
1: $v_{new} = \{w\}$ and $E_{new} = \{\}$ where $w$ is any arbitrary vertex;
2: while $v_{new} \neq v$ do
3:     Find an edge (u,v) where $u \in v_{new}$ and $v \in V \setminus v_{new}$ with min weight
       (break ties arbitrarily).
4:     $v_{new} = v_{new} + \{v\}$;
5:     $E_{new} = E_{new} + \{(u, v)\}$;
```

Time Complexity and Implementation: The implementation is very similar (almost identical) to Djikstra (using heap). At each stage we can keep SE[v] (instead of SP[v] in Djikstra) which keeps the min edge connectivity v_{new}

to this vertex v and we update it each time that we add a new vertex to v_{new}. Identical to Djikstra, the running time is $O\Big((|V|+|E|)\log|V|\Big)$.

Proof of Correctness: This can be shown using induction.

Induction Hypothesis
There is always a best solution (optimum) that has the first k edges that we add to E_{new}

Proof: The base is trivial since there is no edge. Note that v_{new} plays the same role of v_k in Djikstra. Consider the tree OPT which only the first k edges of E_{new}. We prove there is another OPT' with same cost which has the first k+1 edges of E_{new}. Now let (u,v) be the (k+1)-th edge that we add to E_{new} (and thus to the tree) where $u \in v_{new}$ and $v \notin v_{new}$. Now let us add (u,v) to the tree OPT. There should be a cycle and there is an edge (u',v') where $u' \in v_{new}$ and $v' \notin v_{new}$. Since we checked all edges going out of v_{new} we have $w(u',v') \geq w(u,v)$. So we add (u,v) instead of (u',v'), preserve connectivity, maybe lower the total weight and still have k+1 edges in common with new OPT'. ■

The problem is much harder if we have directed graphs. It is in fact NP-complete.

3 NP-completeness

3.1 Easy and Hard Problems

So far we have seen many algorithms. All of them had polynomial running times, i.e., $O(n^k)$ for some constant k. Lots of them were indeed linear $O(n)$ or quadratic $O(n^2)$. These problems are **easy** problems. But are there problems which are **hard**, i.e., they need exponential time like $O(2^n)$ or $O(n!)$. The answer is YES. There are some weird problems for which we know we cannot solve them in polynomial time and we need exponential time, but for lots of normal problems, still we do not know the answer. These problems lie in the class of **NP-complete** problem. Note that NP stands for "Non-deterministic polynomial" and not "Not in polynomial" for now, but the conjecture is that indeed it is true. If you can solve this problem, you will get Millennium Prize of US $ 1,000,000 by Clay Mathematics Institute. Let us be a bit more formal now.

3.2 Decision Problems

Decision problems are problems for which the answer is either YES or NO, e.g,, can we find a shortest path or an MST of cost w. Note that if we can solve such problems then we can solve lots of optimization problems as well by a binary search.

P is the class of all decision problems that can be solved in polynomial time, i.e., $O(n^k)$ for some constant k.

EXP is the class of all decision problems that can be solved in exponential time, i.e., $O(2^{poly(n)})$ where poly(n) is some polynomial in n.

3.3 Polynomial-time Verification

For many problems that may be very hard to solve, we might have the property that it is easy to **verify** whether its answer is correct. For example, consider the coloring problem: Given an undirected graph G, find the min number of colors needed so that we can color each vertex with one of these colors and have a **valid coloring**, i.e., an assignment of colors to vertices such that each vertex is assigned one color and no two adjacent vertices have the same color. The 3-Coloring problem asks if we can have a valid coloring with 3 colors. Note that though finding a 3-coloring of a given graph is not easy, however if you somehow obtain a solution with 3-colors it is easy for someone to **convince** that you did: just check that you did not use more than 3 colors in $O(|V|)$ time and check validity of coloring in $|E|$ time. Thus though we do not know any poly time algorithm to decide if a given graph has a valid 3-coloring, there is a very efficient way to verify if a given graph has your answer as a valid 3-coloring. The solution that you provide is called as a **certificate**. This is some piece of information which allows us to check whether the graph has a valid 3-coloring. If it is possible to verify the accuracy of a certificate in poly time, we say that the problem is **polynomial-time verifiable**.

NP is the set of all decision problems that can be verified by a polynomial time algorithm.

Note that poly-time verifiable and solving in poly time are two very different things, e.g., 3-coloring problem is NP-complete (as we will see later) but is verifiable in poly time. Also note that polynomial verification is not always easy. For example, consider the problem of deciding whether the graph has exactly one valid 3-coloring. It is easy to check that there is one but not clear to show that this is the only one.

Then why do we say NP (non-deterministic poly time) instead of VP (verifiable in poly time)? Due to history, here we are referring to a non-deterministic computer which can make guesses for the certificates and thus we only need to verify the certificate in poly time. You can learn more on this topic in other courses such as Complexity Theory and Formal Language Theory.

Note that it is clear that $P \subseteq NP$, since for any problem that we can solve in poly time, we can surely verify it in poly time. But we still do not know the reverse, i.e., whether $NP \subseteq P$, and this is the big open problem mentioned before in the lecture. Many experts believe that $NP \nsubseteq P$ and thus $P \neq NP$.

References

[1] Udi Manber, *Introduction to Algorithms - A Creative Approach*