1 Introduction and Objective

In this tutorial, we will continue our discussion on Graph problems, specifically the Minimum Spanning Tree (MST) problem.

We will heavily use http://visualgo.net/mst.html during our discussion.

Get ready to be transformed by the discussions in this tut06, especially on Question 2-3-4 :). We recommend that you read the questions and think for a while first before attending this tutorial as such questions will be the one that will appear in Written Quiz 2 and Final Exam. You need to be able to grasp the thinking process, not just to record down the solutions form your tutor...
2 Tutorial 06 Questions

Basic Stuffs about MST Algorithms

Q1. In Lecture 07, you are presented with two MST algorithms: Prim’s and Kruskal’s. First, the tutor will (re-)demonstrate the executions of Prim’s and Kruskal’s on a small connected weighted undirected graph using http://visualgo.net/mst.html. The tutor may invite some students to do this live demonstration together. Then, the tutor will ask you to list down as many similarities and as many differences that you can find out of these two MST algorithms!

Similarities:
1. Both are graph algorithms that can solve the MST problem,
2. Both runs in $O(E \log V)$
3. Both are greedy algorithms.
(Other similarities not listed here are possible).

Differences:
1. The inventors are different, hence the names of the algorithms are different,
2. The way each algorithm works are different (see below):

In Prim’s algorithm, the MST is built by growing it one (lightest) edge at a time from a single tree. Thus at any point in time, cycle detection is easy. From vertex $u$, when we want to consider taking a edge $(u, v)$, we can just check whether the other endpoint vertex $v$ is already included in the tree. If it is, then including this edge will form a cycle and thus we can ignore this edge. If not, we can safely include that edge. We order the edges dynamically using a Priority Queue.

In Kruskal’s, we first sort all $E$ edges upfront using any standard $O(E \log E) = O(E \log V)$ sorting algorithm (we do not really need a Priority Queue here). Then, we add the (lightest) edges one by one. This can cause multiple components/tree to spring up as the algorithm progresses (thus making a forest with diminishing number of trees until we finally get one single tree). When we add a new edge, we are not sure whether that edge links 2 different trees together or they cause a cycle in an existing component. Thus cycle detection is not as easy compared to Prim’s algorithm above. We need another DS: the Union-Find Disjoint Sets (UFDS) to do this cycle check efficiently. In the end, all the different components will be linked up to form one single component.

Not-So-Standard MST Question

Q2. Given a graph $G$ with $V$ vertices and $E$ edges and the MST of $G$, produce the best algorithm to update the MST if a new edge $(A, B)$ is to be inserted into $G$.

First, we need to realize that we do not need to insert $(A,B)$ into the original graph $G$ and re-run Prim’s or Kruskal’s. Although obviously correct, this runs in $O((E + 1) \log V) = O(E \log V)$.

We can do a little bit better by only inserting the edge $(A,B)$ into the MST itself and then re-run
Prim’s or Kruskal’s on this MST+1 edge. As an MST is a tree, it only has \( E = V - 1 \) edges, so this strategy runs in \( O((V - 1 + 1) \log V) = O(V \log V) \). Is this the best that we can do?

Do more observation. Before insertion of this edge \((A,B)\) into the MST, there is only a path from vertex \( A \) to vertex \( B \) in the MST. After the insertion of edge \((A,B)\), we have a cycle which consists of the edges of the path from \( A \) and \( B \) and the new edge \((A,B)\). In order to update the MST, we only need to find the largest edge in this cycle and remove it. This will ensure that the resultant graph will remains the MST for the new Graph \( G \cup (A,B) \).

In order to do so, simply run DFS from \( A \) in the original MST. We keep track of the largest edge \( X \) seen so far as the DFS recursively explores all path, until it hits \( B \). Now the largest edge \( X \) must be the largest edge of the path from \( A \) to \( B \). Stop the DFS and compare \( X \) with the new edge \((A,B)\). If weight of \((A,B)\) is larger than \( X \), then the MST does not change. Otherwise, we remove \( X \) and insert \((A,B)\) instead into the MST to have a better MST.

This algorithm runs in \( O(V) \) since DFS is \( O(V) \) on a tree and removing \( X \) and inserting \((A,B)\) can also be done in \( O(V) \) (e.g. we use a (sorted) EdgeList, we can find \( X \) with an \( O(V) \) loop and we can insert \((A,B)\) in the correct position also in \( O(V) \)).

**MST Application 1: Clearly Has MST Flavor**

Q3. An ambitious cable company has obtained a contract to wire up the government offices in the city with high speed fiber optics to create a high speed intra-net linking up all the different governmental departments. In the beginning they were confident that the minimum cost of connecting all the offices will be within budget. However, they later found they made a miscalculation, and the minimum cost is in fact too costly. In desperation, they decided to group the government offices into \( K \) groups and link up the offices in each group, but not offices between groups to save on the cost. This effectively creates \( K \) intra-nets instead of one big intra-net.

Given \( V \) government offices, the cost of linking \( E \) pairs of government offices, and a budget \( B \), help the company design a program which will tell them what is the smallest value of \( K \) (so as to minimize the number of intra-nets). The program also needs to output the government offices in each of the \( K \) groups and how they should be linked such that the total cost of all selected links is within budget \( B \).

The program should model the problem as a graph. It should also run in \( O(E \log V) \) time. Where \( E \) and \( V \) are the number of vertices (government offices) and edges (possible links between those government offices), respectively.

The algorithm is simple: Just run Kruskal’s which sorts \( E \) edges from cheapest to most expensive (Prim’s algorithm is not suitable to solve this problem), keep track of sum of the weights of the edges picked so far. Once an edge that is going to be added will cause the sum to exceed the given budget \( B \), we stop the algorithm. Then report \( K \) = the number of disjoint sets currently in the UFDS. We can do an \( O(V) \) pass to trace the \( p[i] \) array of the UFDS to list down members of each groups. As we don’t do anything other than potentially stopping Kruskal’s earlier, this is at worse \( O(E \log V) \), same as the standard Kruskal’s algorithm.
MST Application 2: The MST Problem is Not Obvious


Simply run Kruskal’s but sort edges in non-increasing (downwards) order. You will get Max ST. Edges not taken by Kruskal’s for the MaxST will be the edges where we have to place the cameras. Sum their weights! The time complexity is simply $O(E \log V)$, the same as Kruskal’s time complexity. The solution is very short but not easy to arrive at such solution. Your tutor will go through the ‘thinking process’ on how to arrive at such solution during the actual tutorial.

We will later revisit this problem during Lab Demo 06, so you may want to try solving [https://uva.onlinejudge.org/external/12/1234.pdf](https://uva.onlinejudge.org/external/12/1234.pdf) using this newfound knowledge.

Problem Set 4

Q5. Finally, the tutor will quickly discuss another trivial idea for PS4 Subtask A.

In Subtask A, you are given a tree... In a tree, there is only one possible path to go from one vertex to another. So simply run either $O(V)$ DFS/BFS from $s$ to $t$ and report the edge with the highest effort rating along that path. The time complexity is $O(V)$ as $E = V - 1$ in a tree.