Bryan Arnold

CSE 3500

Homework 5

3/6/17

Minimum spanning tree

a) To begin, let graph G have vertices v1 through v4. There exists an edge between each pair of vertices, and with the weight on the edge from vi to vj equal to i + j. Since every tree has a bottleneck edge of weight being greater than or equal to 5, the tree consisting of a path through vertices v3, v2, v1, and v4 is a minimum bottleneck tree (weight being 10). A minimum spanning tree would have a total weight that is greater than the tree with edges from v1 to another other vertex. This means that the following tree of graph G is a bottleneck tree as stated earlier, yet is not a minimum spanning tree as it violates the total weight property of a minimum spanning tree. Therefore, this counterexample proves that not every minimum bottleneck tree of G is a minimum spanning tree of G.

b) First, suppose that T is a minimum spanning tree of G and T1 is a spanning tree with lesser weighted bottleneck edge. This means that T contains an edge e that is heavier than every edge in T1. If we were to add to T1, it forms a cycle C on which it is the heaviest edge. By utilizing the cut property, then e does not belong to any minimum spanning tree, contradicting the fact that it is in T and T is a minimum spanning tree. Therefore, this proves that every minimum spanning tree is a minimum bottleneck tree of G.

MST

1. It is given that edge e = (u, v) does not belong to an MST iff there is a path between u and v with all other edges being cheaper than e. Let us assume the opposite of this. Let us suppose that e belongs to a minimum spanning tree denoted as T. By removing e, T will separate into two subtrees with the two ends of e in different subtrees. The remainder of cycle C reconnects the subtrees. Therefore, there is another edge f of C with ends in different subtrees. This means that edge f reconnects the subtrees into a tree T1 with weight less than that of T, because the weight of f is less than the weight of e. This is a contradiction to the supposition, thus making the given statement that e = (u, v) does not belong to any MST iff there is a path between u and v with edges all cheaper than e true.

2. For my algorithm to test if the edge e connecting vertices u and v will be a part of some MST or not, the following must be done.

   First, run a depth first search from one of the points of the edge e, so point u or point v, considering only those edges that have weight less than that of e. There are two instances to detect whether this edge e is in an MST:

The first instance is if at the end of this depth first search, u and v get connected, then edge e cannot be a part of some minimum spanning tree. This is because in this case there definitely exists a cycle in the graph with the edge e having the maximum weight and it cannot be a part of the MST. This is explained completely in the first part of this problem. If this instance happens, display that edge e is part of some MST. Next, the second instance.

For instance two, if at the end of the depth first search u and v stay and do not become connected, then edge e must be the part of some MST. Edge e is not always the maximum weight edge of all the cycles that it is a part of, also explained in the earlier part of this problem. If this instance occurs, simply display that e is part of some MST.

Now that the algorithm is complete, it is easy to decipher the run time of this algorithm by simply looking at DFS. Since DFS has to check through all the edges and vertices, and the function of checking whether or not u and v are connected for edge e only takes O(1) time, the runtime can be described as O(|V| + |E|).

Divide and conquer algorithm

To find the depth of a node in each tree, a divide and conquer method must be implemented. Here is how my algorithm would play out:

First, take in a node of a tree as the parameters. This allows for a starting place for the algorithm.

Next, set up a base case statement to eventually end the recurrence in my algorithm. This base case would simply be if the root has no value or is null, return a value of 0, meaning there is no depth.

Initialize an integer variable, h, to store the counted depth so far, which would be 0. Next, loop through each node of the given root node’s children. As you go through all the nodes, find the maximum value between the counted depth so far and another run of the algorithm for the specific node n we are on and store it in h. It would look something like this:

```
For(TreeNode n : children of the root)
        h = maximum value between h and getDepth(n)
```

Note: getDepth is the name of the algorithm

Finally, the recurrence will eventually end, and simply return h + 1 for the depth of the tree.

Since each node only visits the nodes below itself in this recurrence relation, the runtime is O(n), since all the nodes are only visited once in the for loop.

Recurrence

1. $T(n) = 12T(\frac{n}{4}) + n^{1.5}$

$a = 12, b = 4, f(n) = n^{1.5} \Rightarrow c = 1.5$

$\log_b a = \log_4 12 \approx 1.79$ $\quad \log_b a > c$

Since $\log_b a > c$, use case 1 of the master theorem:

$T(n) = \Theta(n^{\log_b a}) = \boxed{\Theta(n^{1.79})}$ $\quad \varepsilon = ?$

2. $T(n) = T(\sqrt{n}) + \log n$

$a = 1, b = 1, f(n) = \log n$

$f(n) = \Theta(n^c \log^k n)$ $\quad c = 1, k = 0$

$\log_b a = \log_1 1 = 1$, so $c = \log_b a$

Since $\log_b a = c$, use case 2 of the master theorem

$T(n) = \Theta(n^1 \log^1 n) = \boxed{\Theta(n \log n)}$

3. $T(n) = T(\frac{n}{2}) + \Theta(1)$

$= T(\frac{n}{4}) + \Theta(1) + \Theta(1) = T(\frac{n}{4}) + 2\Theta(1)$

$= T(\frac{n}{8}) + 3\Theta(1)$

$= T(\frac{n}{16}) + 4\Theta(1)$

$= T(\frac{n}{32}) + 5\Theta(1)$

$\vdots$

$T(n) = T(1) + \log_2(n) \cdot \Theta(1)$

$= \boxed{O(\log_2(n))}$