CS 330 – Fall 2018, Assignment 3 Solutions

Problem 1. *Chapter 4. Exercise 24.*

For each node in the tree, keep track of three values:

```
left_height[v] = longest path length from v to any leaf in its left subtree
right_height[v] = longest path length from v to any leaf in its right subtree
height[v] = max (left_height[v], right_height[v]);
```

Greedy algorithm:

```
for each node v in BFS order do:
    w = left_height[v] - right_height[v];
    if (w > 0)
       add w weight to edge e = (v, rchild[v])
       right_height[v] = left_height[v];
    else if (w < 0)
       add w weight to edge e = (v, lchild[v])
       left_height[v] = right_height[v];
    else
       // w == 0, no imbalance

 // NOTE:  height[v] did not change, since we never changed the max!
```

Running time: We plan to compute the height calculations as a preprocessing step, then do the BFS traversal. There are $2n - 1$ nodes (and $2n - 2$ edges) in a binary tree with n leaves. The heights of all leaves are 0. For an internal node v, its left height is the weight of (v, lchild[v]) + height[child[v]], and similarly for the right height. The preprocessing can be implemented as a recursive or iterative tree traversal in O(n) time. BFS traversal of a tree involves only constant-time operations per node. Also, O(n) time.

Correctness: First observe that there is some root to leaf path where we added no weight. This follows because we add weight either to a node's left or right child, but never both, so we can always keep descending to a leaf. We claim that we synchronize the tree to this path length m. We cannot possibly synchronize to a smaller value than m, as that would require shortening a path in the input.

Next we must show that we synchronize the tree to m and that among the ways to synchronize to m, there is no way to add less weight overall than our algorithm did. To show synchronization, observe that the weight we add always brings the left and right heights of v into balance, and when we do that for every node, it is fully synchronized.

Finally, we will prove that we added the minimum weight using a "Greedy stays ahead" proof by contradiction argument. We have established that an optimal solution must synchronize to m, but suppose OPT did so with a different placement of weights on edges. Take the optimal solution that agrees with our solution's added weights for as long as possible (in a breadth-first ordering). At the position of first disagreement, adding weight to edge $e = (u, v)$, OPT must apply less weight than we did, because if it applied more weight, it would increase the height of u, as well as every vertex above u, causing the root's height to exceed m, a contradiction. If it adds less weight than we did, than that missing weight must be added later to *both* of v's subtrees to synchronize all heights to m. Therefore OPT could achieve a savings, by adding that missing weight once directly to (u, v) instead of multiple times later. This also is a contradiction.

Problem 2. *Chapter 4, Exercise 19, on p. 198.*

We demonstrate that the maximum spanning tree of G, i.e., the spanning tree with maximum total weight, achieves the best bottleneck rate for each pair of nodes (u, v) in G. First, observe that the maximum spanning tree is closely related to the minimum spanning tree, algorithmically and with respect to the cut and cycle properties. Consider the graph $-G$, in which all the weights on all the edges are negated.

Claim 1. . *The minimum spanning tree of $-G$ is the maximum spanning tree of G.*

Proof (by contradiction): Suppose not. Let T be the MST of $-G$, but some other tree T' is the maximum spanning tree of G. The trees have weight $w(T)$ and $w(T')$ in G. Similarly, these same trees have weight $-w(T)$ and $-w(T?)$ in $-G$, where all weights are negated. Since T' is the maximum spanning tree, $w(T') > w(T)$. Negating both sides: $-w(T') < -w(T)$ (negation reverses the inequality). But since T is an MST of $-G$, $-w(T) < -w(T')$, resulting in a contradiction. $\square$

For an efficient algorithm for maximum spanning trees, we could actually go through and negate the graph, or we could more simply just run Kruskals and Prims with edges prioritized in descending order. Also, *using similar negated reasoning*, the maximum spanning tree has the opposite cut and cycle properties from minimum spanning trees: it contains the heaviest edge crossing any (S, T) cut, and omits the lightest edge along any cycle.

Claim 2. *The Maximum Spanning tree gives us the best solution, that is, for each $u, v \in V$, the bottleneck rate of the $u - -v$ path in T is equal to the best achievable bottleneck rate for the pair $u, v \in V$.*

Proof (by contradiction): Suppose, for the purpose of a contradiction, that there exists a pair of nodes (u, v) for which the spanning tree path P has a lower bottleneck rate than an alternate path P'. By definition of bottleneck rate, P has some lowest-weight edge e with weight $w(e)$ lower than all weights of edges in P'. Now consider the cycle formed by P followed by P' in reverse (a cycle going from u to v and back). The edge e is the lowest weight edge in that cycle, so by the cycle property, it should be omitted from the maximum spanning tree. But it is in the maximum spanning tree, a contradiction. No such pair (u, v) exists, so the maximum spanning tree path is always best. $\square$

Problem 3. *Chapter 5. Exercise 1.*

First, it is important to understand exactly what this exercise is asking for: the two databases contain a total of $2n$ numbers, our task is to return the median, that is the nth largest number in the the combined set. Also, we do not have access to, and also do not care, about how the data in the two databases is stored (e.g. is it sorted?, what data structure is used?). It is a black box. The only query that can be asked from each database is "return the kth largest element in this database".

How do we make this a divide and conquer problem? Often, trying to divide by two is a reasonable approach. Here, the key is to relate the problem of finding the overall median to finding the median of two databases of half the size. If we can do that divide step in constant time, we will have a recurrence like binary search, where $T(n)$ is the time to find the median of two databases of size n:

$$T(n) = T(n/2) + O(1)$$

Note that we also need a base case, like $T(1) = O(1)$, but since it is clear that we can find the median of two one-element databases by brute force, that is straightforward.
Key observation: Here's how to divide the problem in half in constant time. Suppose we take the median of the first database DB1 and the median of the second database DB2. Call the smaller of these two medians s and the larger ℓ. Now consider s as a pivot: if we take s, ℓ, and all elements larger than either of those two, that is more than half of the elements, so the median must be in that set. Therefore, we can exclude all elements in s's database that are smaller than s. (We cannot exclude elements smaller than ℓ, as those might be larger than s, and we cannot exclude those!). Similarly, we can exclude all elements in ℓ's database that are larger than ℓ. So, just like binary search, after two database queries, we can eliminate half of the elements. Moreover, by excluding sets that are the same exact size above and below the median, the median of the new restricted databases is the same as the median of the original two databases.

How are we going to implement this? The key idea is to mimic binary search, and maintain an interval of pinning $[\ell, r]$ in which we are still searching. Except now, we need an ℓ and an r for each database.

```
int find_median(l1, r1, l2, r2)   // find median of DB1(l1, r1) and DB2(l2, r2)
{
    if (l1 == r1)              // base case, 1 element per db.  return smaller.
        return min (lookup (DB1, l1), lookup (DB2, l2));

    m1 = lookup(DB1, (l1+r1)/2);
    m2 = lookup(DB2, (l2+r2)/2);

    if (m1 < m2)    // throw out left half of DB1, right half of DB2
        find_median((l1+r1)/2, r1, l2, (l2+r2)/2);

    else  // m1 >= m2.  throw out right half of DB1, left half of DB2
        find_median(l1, (l1+r1)/2, (l2+r2)/2, r2);
}

To compute over both DBs, execute:   find_median (1, n, 1, n);
```

Correctness: The correctness of this algorithm follows from the key observation above. At the start of the algorithm all $2n$ numbers in the database are candidates for the median value. In each recursive call we eliminate an equal number of candidates that are larger and smaller than the median, preserving the overall median.
Running time: The running time in this problem is measured in database calls. Each call to find_median() performs exactly two calls. So the recurrence is $T(n) = T(n/2) + 2$, with $T(1) = 2$, which is an O($\log n$), recurrence, same as binary search. In fact, it's exactly twice as many calls as binary search.

Problem 4. *Minimum Spanning Tree implementation*

First question: Prim's or Kruskal's? Either is fine, and both should be similarly easy to implement. With Prim's you needed a priority queue, a start vertex, and an implementation of the simple greedy updating method. For Kruskal's, all you need a fast sorting algorithm.

Second question: Adjacency list or adjacency matrix? Hmmm, since we are dealing with complete graphs, maybe this isn't that important a distinction. The degree is always $n-1$, and it looks like we need to store a weight for every edge. On that basis, maybe the adjacency matrix is fine.

Based on this (Prim's/Kruskal's + adjacency matrix), I think it's pretty easy to get to all but the very largest input sizes. Most of you are going to run into space trouble allocating an 8192 x 8192 matrix though. The cleverest workaround is to notice on your smaller inputs, that the MSTs produced have very small values as n increases, maybe much much smaller than you expected. For both graphs, it turns out that you are only using teeny-tiny edges, each of length at most $\frac{\log n}{n}$ [check it out!]. Therefore, you never even need to look at these edges. In other words, during the random edge generation, all but very short edges can just be thrown out, like the smallest 1%. So it's actually best to use adjacency lists to store the sparse graph of very short edges. As long as your longest edge used is shorter than your discard threshold, you're safe!

Random edges: The total MST weight grows very slowly. In fact, the total MST weight converges to a constant as n grows large! (The proof is at the level of a graduate randomized algorithms course). The key is that each new vertex has a light edge (weight about $1/n$) that connects it back to the rest of the graph, with high probability.

Euclidean edges: MST does grow, but slowly. The process yields MSTs of total weight that have growth rate about $\log(n)$. Here the intuition is that you have to actually span all the points in Euclidean space, which gives you a bristly tree that reaches every vertex in the unit square. This is closely related to a useful data structure called a Quadtree, used in image compression.