COMP 251: Data Structures and Algorithms

Solution for Assignment #4

1 Union-by-Depth

A lot of students missed this question by misinterpreting the question. The claim is that for any tree of size n generated using the union-find data structure, the upper bound of the height of the tree is $O(\log n)$. Hence, you should do an induction on n, not on the height. Also, for the base case, you should at least state that the base case holds trivially for $n = 1$.

For base case, when $n = 1$, the height of the tree is 0 which is $O(\log 1)$. Now, assume the claim holds for all values $n \leq k$. To show that the claim also holds for the case $n = k + 1$, consider the two subtrees merged during the last union operation to construct the tree of size $k + 1$. These two subtrees are of size n_1 and n_2 such that $n_1 + n_2 = k + 1$, and they have height h_1 and h_2, respectively. There are two cases:

1. $h_1 = h_2$: The merged tree has height h_1+1. By the induction hypothesis, $h_1+1 = O(\log n_1) = O(\log n)$.
2. $h_1 \neq h_2$: The merged tree has height $max(h_1, h_2)$. By the induction hypothesis, $max(h_1, h_2) = O(max(\log n_1, \log n_2)) = O(\log n)$.

2 Hex

There are three procedures to design here. Many people used the Union by rank with path compression method asserting that the winning move can be detected by keeping the representative of each disjoint set as the upperleft most (or lowerbottom most) element on the board. This requirement is somewhat unnecessary as shown below.

First, for InitBoard, initialize the $(n + 2) \times (n + 2)$ board via virtual initialization. Then, around the edge of the board, place in each cell either a red piece or a blue piece (ie. blue pieces along the top and bottom edges, red pieces along the left and right edges). This initialization makes it easy to check for a winning move. This uses at most $O(n)$ MakeSet operations.

For RedMove, place a red piece at its desired location(ie. make a set at that location), and check if any neighboring cells are occupied by red pieces. If so, union the newly created set with the set containing the neighbouring cell(So there are at most constant number of Union operation here.).

Then, we check if the representative of the set containing any cell on the left boundary is the same as the representative of the set containing any cell on the right boundary. If they are, this is a winning move and quit the game. Otherwise, continue. This uses 2 FindSet operations.

The runtime bound of this implementation follows directly from the theorem discussed in class.

3 Scheduling again

In this problem, using a priority queue will help you keep the algorithm simple. Furthermore, a lot of students designed algorithms that iterate through every unit of time. (i.e. You do one unit of time worth of work per each iteration.) This is generally bad. When you say an algorithm runs in $O(f(n))$ time, where f is some function of n, n usually refers to the input length(bit length). If you have a parameter such as t_i or r_i in the input, these parameters can be exponential in n. For example, it only requires 20 bits to store the number 1000000. If your algorithm executes the task one unit of time per iteration, and the input contains a task that takes 1000000 seconds to complete, your algorithm will iteration a million times, which is exponential in n.

To design this algorithm, recall the previous scheduling problem discussed in class. In that problem, all the tasks are released at time 0. In that sense, that was a special case of the problem we need to solve here. Hence, if we can break down the tasks into chunks such that we can use the same strategy as the previous problem, we are done.

First, sort all the tasks by its release time r_i. Then, there will be a set of tasks released at the same time, grouped together in the sorted array. We then iteratively take those set of tasks released at the same time, and insert them into a priority queue, using t_i as the key.

After inserting the tasks into the priority queue, we use FindMin operation on the priority queue to find out how much the shortest job takes. Let us call this shortest job T. By comparing the time from now till the next release time against the t_i of T, we either execute the whole process T and ExtractMin on the priority queue, or execute the amount of time left till the next release time and DecreaseKey on T by the time executed for T. If there is still time left till the next release time, repeat.

The running time can be easily analyzed by the number of Insert, ExtractMin, and DecreaseKey operations called, each of which takes $O(\log n)$ time. We insert each task exactly once when it's released, so there are n Insert operations. Similarly, we extract each task exactly once when it's completed, so n ExtractMin operations are called. The number of DecreaseKey operations called is bounded by the number of preemptions. The number of preemptions is then bounded by the number of tasks (since a preempted task is completed right away), so there are n DecreaseKey operations. Therefore, the algorithm takes $O(n \log n)$ time, overall.

4 Minimum Spanning Tree, take three

a)

Many students answered this part incorrectly. The tricky part is, after you take out e from the cycle, you cannot arbitrarily choose another edge on the cycle and add it back to the MST. This is because by adding this new edge, you may be introducing a new cycle into the MST. Some people also argued that all other edges in the cycle have weight strictly less than e to lead themselves to a contradiction, but there may be other edges whose weight is the same as e.

Let C be a cycle in G, and $e = (u, v)$ be an edge on C with the heaviest weight. Let T be an MST of G. If T does not contain e, we're done. So suppose T does contain e. Remove e from T to obtain T'. T' is a forrest, containing two trees T_1 and T_2. Because e was originally on a

cycle C, there must exist an edge f on C, connecting a vertex in T_1 to a vertex in T_2. Add f into T'. Now T' is a spanning tree, reaching all the vertices in G. To argue that T' is also a *minimum* spanning tree, recall that e contained the heaviest weight on C, so $w(f) \leq w(e)$. Since $w(T') = w(T) - w(e) + w(f)$, we have that $w(T') \leq w(T)$, which completes the proof.

b)

We denote G^i as the graph at i^{th} iteration of the algorithm.

1. Loop Invariant: The residual graph G^i contains an MST of the original graph G.
2. Maintennance: The difference between G^i and G^{i-1} is some edge e which is a heaviest edge in some cycle. By the property shown in a), there is an MST for G^{i-1} that does not contain e.
3. Termination: The final residual graph contains no cycle, i.e., it is a spanning tree. Due to the loop invariant, this spanning tree contains an MST. A tree containing an MST is trivially an MST itself.

c)

The sorting edges takes $O(|E| \log |E|)$ time. The for loop runs $|E|$ times, and checking of an edge is part of a cycle takes $O(|E| + |V|)$ time via DFS. Overall, the algorithm runs $O(|E|^2)$ time.