1 Book 6.16: ROTC

Background

There are many sunny days in Ithaca, New York; but this year, as it happens, the spring ROTC picnic at Cornell has fallen on a rainy day. The ranking officer decides to postpone the picnic and must notify everyone by phone. Here is the mechanism she uses to do this.

Each ROTC person on campus except the ranking officer reports to a unique $superior\ officer$. Thus the reporting hierarchy can be described by a tree T, rooted at the ranking officer, in which each other node v has a parent node u equal to his or her superior officer. Conversely, we will call v a *direct subordinate* of u.

To notify everyone of the postponement, the ranking officer first calls each of her direct subordinates, one at a time. As soon as each subordinate gets the phone call, he or she must notify each of his or her direct subordinates, one at a time. The process continues this way until everyone has been notified. Note that each person in this process can only call direct subordinates on the phone.

We can picture this process as being divided into rounds. In one round, each person who has already learned of the postponement can call one of his or her direct subordinates on the phone. The number of rounds it takes for everyone to be notified depends on the sequence in which each person calls their direct subordinates.

Give an efficient algorithm that determines the minimum number of rounds needed for everyone to be notified, and outputs a sequence of phone calls that achieves this minimum number of rounds.

1.1 Mathematical Formulation

Given an input of the tree T with n nodes in total and depth of d. Let the most number of direct subordinates (children) any one officer (parent) has be expressed as c. We will determine least number of calls needed to reach every node of the tree as well as the order in which to acheive this.

1.2 Solution

This algorithm begin by doing a by-level bottom up traversal. As it traverses, it will assign values to each node based off its children. The way it does this is first sorting all of its children from highest value to lowest and such that $v_0 \geq v_1 \geq ... \geq v_{n-1}$ for a node with n children, then saying:

$$value = \begin{cases} 0\ if\ no\ children \\ max_{i \in 0..(n-1)}\ (child_i value + 1 + i)\ else \end{cases}$$

With this we will develope our values and when we reach the top level (n), its value will be the least number of calls needed to be made. Our path to take then is to call the open value node from each seen node.

Algorithm 1 Fill out Tree and get num Calls

```
procedure getNumCalls(Tree T)
    for $level \in 1..d$ do
        for $node \in T.level(level)$ do
            int value = 0
            if node.hasChildren then
                children.inverseSort
                int i = 0
                for $child \in children$ do
                    childV = child.value + 1 + (i++)
                    value = max(value, childV)
            node.setValue(value)
    print(root.value) // least num calls
    traverse(T) // T has all values filled out and inverse sorted
```

1.3 Correctness

Proposition 1.
This algorithm will determine the correct minimum number of calls.

Proof.
We do this by construction. We observe that doing a bottom up traversal we can see that as long as the children have the correct value, then the parent will be able to calculate the correct value. When I say value of a node I am referring to the number of calls required to fully reach the subtree rooted at this node, assuming the node has already been reached. $\implies$ if the node is a leaf, its value = 0. Therefore when we have a node with a single leaf as a child, $\implies$ will take the number of calls that the leaf will make, 0, + 1 because it has to call the leaf. Now if there are multiple children, k of them, and all of them are leaves we will note that we cannot call multiple leaves at the same time, therefore the number of calls that this node will have to make is n and its value will also be n since each of the nodes must make 0 calls.

Now lets change it such that each of the children have thier own value $v_1, ..., v_n$ we note that it will always be most efficient to call the children with more calls to make first, therefore let us inverse-sort them based off of their values s.t. $v_1 \geq ... \geq v_n$ Now we can create a *value* variable and set it equal to cost of calling c_1 which is $v_1 + 1$ beacause we have to call the node itself. We will then check to see if the cost of calling c_2 exceeds that of c_1 as in is cost of calling c_2 second > cost of calling c_1 first? i.e. $value = max(value, v_2 + 1 + 1)$ since it cost 0 to call first, 1 to call second , 2 to call third, etc. We will then continue with this letting $value = max_{i \in 1..(n)}\ (child_i value + 1 + (i - 1))$

After we have done this for every node (building from bottom → up), we just ask for the value of the root which will give us the minimum number of calls we have to make to reach everyone. □

1.4 Analysis

Proposition 2. *The* <u>*space complexity*</u> *of this algorithm is* ***O(size(T))***

Proof.
This is due to the fact that all of our data is stored in the tree itself: □

Proposition 3. *The* <u>*time complexity*</u> *of this algorithm is* $\boldsymbol{O(N \cdot n^* \cdot log(n^*))}$

Proof.
Let us define the average number of children every node has as n^* $\implies$ the cost to sort will be $n^* \cdot log(n^*)$ now since we have to sort for each node, N times $\implies$ $N \cdot n^* \cdot log(n^*)$ Note here though that n^* is likely to be a low number when compared to N. □