Jason Caplan

Design and Analysis of Algorithms

Professor Leff

HW8

1. We use a standard linked-node structure for the binary tree, where each node has a link to its parent and its 2 children. Additionally, the node object has fields for its id, double value, and a flag we describe later. For translation between id's and nodes, we keep an array of length V called $idToNode$, where the index represents the id and the element held there is the node with that id.
   To find the array of doubles where $value_i$ is the "real value" of the person with id_i, we initialize an array of doubles $result$ where $result_i$ starts out as $idToNode[i].value$. To update these, we consider a "virtual copy" of the tree, where we will "remove" 1 "virtual leaf" node at a time (a "bottom up" traversal), and update its parent's intrinsic value to match its own if it is larger than its parent's current value. We do this "virtual removal" by keeping a set of nodes to consider, holding the leaves of the "virtual tree." This is initialized to the true leaves of the tree[i], and is updated as we iterate, as follows:
   - We remove from the consideration set any "virtual leaf" we consider, and set a flag on the node object itself to indicate that it has been considered.
   - Whenever we consider a node, it is possible that this node was an only child in the "virtual copy" of the tree (i.e. its parent either has no other child, or its other child was already flagged as considered), in which case the parent is a newly created "virtual leaf," so we add the parent to the consideration set.
   - We terminate when the consideration set is emptied.

2.

Correctness

- A node's intrinsic value will only be greater than its initial value (and therefore need to be updated) if there is some path "down" from it to a descendant with greater value. Furthermore, as long as this path's links are traversed in strictly reverse order, i.e. "upwards," such that each step of the traversal updates the parent with the greater value between it and its child, this value will "bubble up" to the ancestor properly. Therefore, we just need to show that our algorithm ensures that:
  - #1) every update along every path goes in strictly "upwards" order with respect to that path, and
  - #2) we traverse every component of every possible path from ancestor to descendant
- Proof of #1 | Every update along every path goes in a stricty "upwards" order with respect to that path

    - Since we only consider nodes once they have been added to the set of “virtual leaves,” we can restate this as “a given node will only become a ‘virtual leaf’ after all of its descendants have been considered and removed from the set of ‘virtual leaves.’”
    - This is trivially true, because we only consider something to be a “virtual leaf” if both of its children have been considered (or do not exist), in which case it will be considered strictly after its children, (and surely after its children’s children, since the children were considered after the grandchildren by this very argument, etc).
- Proof of #2 | We traverse every component of every possible path from ancestor to descendant
    - Generally, since every binary tree contains leaves so long as it is not empty, we will continue to remove nodes 1 at a time until the “virtual tree” is empty, and each of these removals is associated with an update of this removed child’s parent. This means that we will have “traversed” every child-to-parent link. By definition, since any path from ancestor to descendant is simply made up of child-to-parent links, we will have traversed every component of every possible path.

Performance

- By a similar argument as Proof #2 above, since we consider a node a maximum of 1 times (it is only considered it if is a “virtual leaf”, and we remove “virtual leaves” from the set as soon as we consider them 1 time), we perform exactly n considerations. Each consideration consists of $O(1)$ work (namely, retrieving 1 “virtual leaf” from the consideration set, updating its parent’s value if necessary, and adding its parent to the consideration set of “virtual leaves” if it is a new “virtual leaf”). Thus, in total our “bottom-up” traversal is $O(n)$.

---

[i] Finding the true leaves of the tree is an $O(n)$ operation, and therefore does not negatively effect the Big-O runtime of our algorithm. This can simply be done by doing any $O(n)$ traversal of the tree (e.g., in-order), and for each node visited, add it to a set of leaves if both of its children pointers are null. In the Java code, I simplified this by updating the set of leaves during construction, but this is essentially the eager equivalent to the aforementioned lazy determination of leaves.