Path between two nodes in Full Binary Tree

Biki Chaudhary
Email:ism2014001@iiita.ac.in
Tara Prasad Tripathy
Email:ihm2014003@iiita.ac.in

Abstract - The objective of this paper is to find the shortest path between two given nodes in a full binary tree. This paper describes an efficient approach that is by finding lowest common ancestor between these two nodes to show the path between them.

Index Terms - Full binary tree, Lowest common ancestor, Nodes

I. Introduction

A full binary tree is a binary tree in which every node has exactly 0 or 2 children. Some examples of full binary tree are shown below in Fig.1. The path between two nodes is traced efficiently by finding a lowest common ancestor(LCA). Let T be a rooted tree. Then, the lowest common ancestor between two nodes n_1 and n_2 is defined as the lowest node in T that has both n_1 and n_2 as descendants. The LCA of n_1 and n_2 in T is the shared ancestor of n_1 and n_2 that is located farthest from the root.

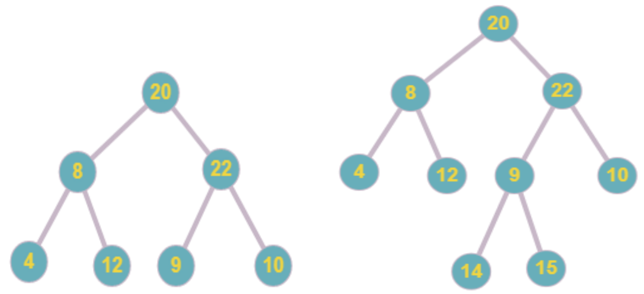

Fig.1. Full Binary Trees

From Fig.1, some examples of LCA between two different nodes of the tree are:-
LCA(4, 12) = 8
LCA(14, 10) = 22
LCA(4, 14) = 20
LCA(22, 14) = 22, etc.
Therefore, after finding lowest common ancestor of two nodes, it becomes easier and efficient to find the path between them. Different approaches are explained in this paper to trace a path between two nodes using lowest common ancestor.

II. Motivation

The problem finds its applications in a lot of varied fields. In class hierarchy of object oriented programming languages, the lowest common ancestor is used to find the most specialized superclass S that has been inherited by two different classes C_1 and C_2. So, any modifications or methods that need to be inherited by the classes C_1 and C_2 can be done in the superclass S. In computational biology, we can study the mutations in our DNA due to a certain virus by locating the origin of the mutation. Tracing the ancestry from two different mutant virus would help to locate where it started. This would help us to take the first step out of the many in the analysis of the stimuli that triggered the mutation. The problem also has applications in fault analysis in servers. To retrace the steps leading to the cause of a system failure could be correlated to the finding the ancestor with the graph representing set of events that happened.

III. Algorithms

A. Algorithm 1: Brute-force approach

This approach is simple that follows by storing root to n1 and root to n2 paths. Data structure for each node used in this approach is as follows:

```
struct Node{
   int key;
   struct Node *left, *right;   //self-referential structure for
//left and right child};
```

Input: Full binary tree, root, node1, node 2
Output: Path between node1 and node2

1.) Find path from root to node1 and store the path in array or vector.
2.) Find path from root to node2 and store the path in another array or vector.

3.) Trace both the paths until the values in the array are same.
4.) The common element just before the mismatch is our lowest common ancestor. Return the index of lowest common ancestor.
5.) Print the path from node 1 to LCA and then LCA to node2 which is our required path between two nodes node1 and node2.

Time Complexity: O(n) where n is the number of nodes. The tree is traversed twice and the path arrays are compared.

B. Algorithm 2: Using Parent Pointer

In this approach we can have one parent pointer with each node that stores its ancestor. Data structure for each node used in this approach is as given below:

```
struct Node{
        int key;
        struct Node *left, *right, *parent;
}
```

Input: Full binary tree, Address of node1 and node2

Output: Path between node1 and node2

1.)Create an empty hash table in the form given as:-

   map < Node*, bool > ancestors;

2.)Insert node1 and all of its ancestors in hash table.
3.)Create stack to store the nodes that does not exist in hash table while searching for node2 and its ancestors.
4.)check node2 or any of its ancestors exists in hash table.
5.)If yes then the first existing ancestor is our lowest common ancestor. Return LCA.
6.)Else, store it in stack and continue from step4.
7.)Finally, print the path from node1 to LCA and then LCA and all the elements from stack.
8.)This is the required path between two nodes node1 and node2.

Time Complexity: O(h) where h is the height of the tree.

C. Algorithm 3: Using depth of the tree

Input: Full binary tree, node1, node2

Output: Path between node1 and node2

1) height_l = height(node1)
2) height_r = height(node2)
3) l = node1, r = node2
4) WHILE l is not equal to r
   i. IF height_l >= height_r:
      1. l = PARENT(l)
      2. height_l = height_l - 1
   ii. ELSE:
      1. r = PARENT(r)
      2. height_r = height_r - 1
5) LCA = r
6) RETURN path from node1 to LCA to node2

Time Complexity: O(h) where h is the height of the full binary tree. h = O(log(n)), where n is the total number of nodes in the full binary tree.

IV. Implementation and Results

The performance analysis was done for increasing heights of the tree till 20 for the algorithm 2 and algorithm 3. For each tree, 1000000 trials were conducted where each time the input was provided randomly. The running time complexity of both is O(log(n) and only differ by a constant term which can be seen in the figure below. The one with the higher constant term is the one using extra memory.

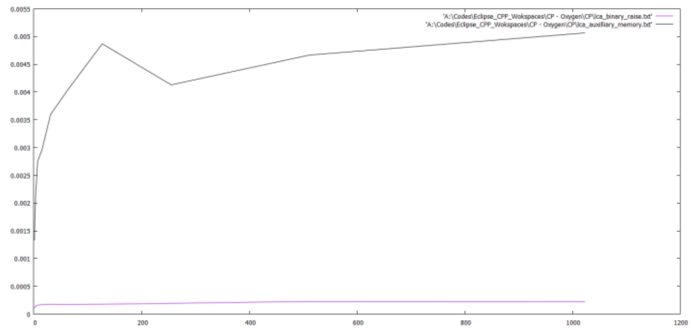

Fig.2. Time complexity of two implementations

V. Conclusion

Two approaches other than the brute force one were analyzed and implemented. Algorithm 1 takes O(n) where n is the number of nodes. This was improved to O(h) time where h is the height of the tree in algorithm 2 and algorithm 3. Algorithm 3 was further improved in terms of space complexity. In algorithm 2 hash table is the extra space required to store all the ancestors from first node. But in case of algorithm 3 no such hash table required. Therefore, we observed and analyzed three different approaches to find the path between two nodes in a full binary tree.