DAA Take Home Exam
GROUP-3
Algorithm to traverse between two given nodes in a given complete binary tree

Sai Charan IIT2016039 | Mantek Singh IIT2016007 | Shruti Reddy IIT2016019 | Rakesh Lakra IIT2016052

***Abstract*—Here, we have traverse between two given nodes in a complete binary tree and find a minimum path between them. An $O(n)$ and $\Omega(1)$ algorithm has been designed to solve this problem .**

***Index Terms*—Searching , least common parent, time complexity, graphs**

I. Introduction and Literature Survey

In computer science, a binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child.

A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

Here we are given a binary tree and two nodes we need to traverse between these two node in the binary tree. For this We are using the basic approach of traversing through the binary tree recursively once from left child and then from the right child. We need find their least common parent so that the path is minimum. So we find the two nodes and their least common parent in the binary tree then we traverse from first node to least common parent and from that to next node. To traverse from node to parent easily we maintaining another pointer the parent in every node.

II. Algorithm Design

The algorithm has a simple a binary tree and two nodes are given. 1. First we check that he two given nodes are equal or not.

2. If they are equal then we just need to check whether that node exists in the tree or not if it exits then the path to it is just one element i.e itself.This checking happens in the search function

3. Now if the two nodes are not equal then we recursively traverse in the tree and find the nodes the while returning we store their least common parent because the path trough it is the minimum path.

4. This happens in the LCA function The recursive call will be stop when we found the two node in the given tree.

5. For stopping the recursive call we used a global variable i.e a counter to check whether two elements are found or not

6. Here when two nodes are found we are sotoring the two nodes and their least common parent. 7.While traversing we traverse from first node to parent in print first function and then from parent to second node in the print last function. The pseudo-code for the above explained algorithm is as follows:

Algorithm 1 traversing path

```
1:  Struct node {
2:  val
3:  struct node *left;
4:  struct node *right;        ▷ the struct node datatype is used to store a node along with it's left and right child in the complete binary tree.
5:  };
6:  node node1 ▷ node1 stores the first node where traversal starts
7:  node node2          ▷ node2 stores the second node where traversal ends
8:  lc        ▷ lc stores the node number of the least common ancestor of node1 and node 2
9:  procedure LCA(root, a, b)
10:     if root==NULL then
11:         return 0
12:    ans1=LCA(root.left, a, b)
13:    ans2=LCA(root.right, a, b)
14:
15:     if (root.val = a) or (root.val = b) then
16:
17:         if root.val = a then
18:             node1 ← root
19:         if root.val = b then
20:             node2 ← root
21:         if ans1 + ans2 = 1 and lc = -1 then
22:             lc ← root.val
23:         return ans1+ans2+1
24:     if ans1 + ans2 = 2 and lc = -1 then
25:         lc ← root.val
26:         return ans1+ans2
```

```
27: procedure SEARCH(root, n1)
28:     if root==NULL then
29:         return 0
30:     if root.val = n1 then
31:         return 1
        return SEARCH(root.left, n1)+SEARCH(root.right, n1)
32: procedure PRINTFIRST(node1)
33:
34:     if
35:         node1.val ≠ lc then
36:         print node1.val
37:         PRINTFIRST(node1.parent)
38:     if node1.val = lc then
39:         print lc
40:
41: procedure PRINTLAST(node2)
42:
43:     if node2.val ≠ lc then
44:         PRINTLAST(node2.parent)
45:         print node2.val
```

III. Time Complexity Analysis

The time complexity for the above mentioned algorithm for binary tree of n elements is calculated by considering that it takes unit time for each computation and calculating for different cases.

Worst Case :The worst case is when the nodes are not present in the binary tree in that case we need to traverse the whole tree. As the size of tree is n so we need to traverse through n elements in the worst case. By this we can say that the worst case complexity of this algorithm is proportion to n i.e worst case complexity is $O(n)$.

Concretely the expression obtained for time complexity is $12 * n + 7$ i.e $O(n)$which serves as the upper bound i.e the worst case time complexity.

Best Case :The best case for this algorithm is when both nodes are equal and that node is root itself in that case for a binary tree of any size the complexity will be constant so the time complexity in best case is proportional to $theta(1)$. Concretely the expression obtained for time complexity is constant 5 i.e $\Omega(1)$ which serves as the lower bound i.e the best case time complexity.

The expression for time complexity in the average case is bounded below by the expression for time complexity obtained in the best case and bounded above by the expression for time complexity obtained in the worst case.

g(x) $<$ f(x) $<$ h(x) where g(x) is the tightest lower bound of the function and h(x) is the least upper bound of the function. The complexity in any case is bounded between $O(n)$ and $omega(1)$

IV. Experimental study

We have found the time complexity of the algorithm by calculating it in each step. Now to will verify it with the experimental results.

We have plotted T VS N graph where is T is the count of total number of computations performed for a given N. The different T values for given N values are mentioned in the following table for different cases and graphs for them are plotted accordingly.

Table I
N VS T GRAPH

n	t_{best}	t_{random}	t_{worst}
10	5	113	127
20	5	196	247
30	5	317	367
40	5	388	487
50	5	435	607
60	5	701	727
70	5	460	847
80	5	735	967
90	5	1036	1087
100	5	915	1207

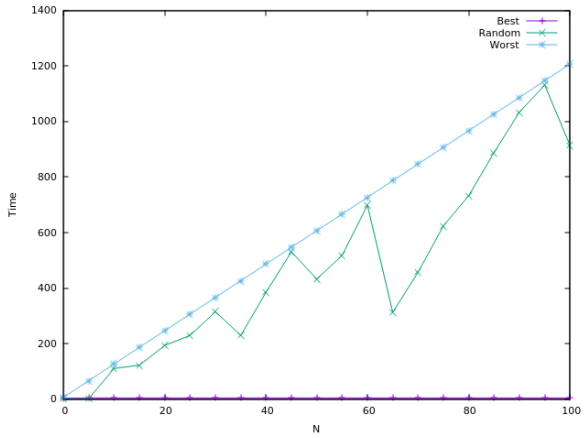

Figure 1. T(y-axis) VS N(x-axis)

From the graph we can clearly see that the best case complexity is $\omega(1)$ i.e best case complexity is proportional to 1 and the worst complexity is directly proportional to n i.e worst case complexity is $O(n)$. Any other case vary between these two curves. By this we can say that the complexity analysis we have done is correct.

V. Discussions

Thus from time complexity analysis and experimental study we can see the overall complexity of the algorithm is always less than or equal to n i.e always $O(n)$.

Here we used a binary tree in which every child has link to its parent so that we can traverse in reverse direction easily. If we try to do it without link to parent then we require another data structure like array or another tree to store the path between given two values and their least common parent.

There are other ways to solve this one of which is we create another tree of same structure as given tree we initialize every element to zero, then increment values along path of two nodes then the last node with two values is least common parent i.e

the corresponding element in the given tree, then we can find path easily.

VI. Conclusion

We have tried to design an efficient algorithm to traverse between two nodes in a given binary tree by traversing through the tree using recursion and finding least common parent.A simple approach of traversing through once through right node and once through left node using recursion is used, the details of implementation and pseudo code along with relevant calculations of time complexities and experimental details is done above to the best of our abilities. Hence, it can be concluded that the given question can be solved in $O(n)$ for worst case and $\Theta(1)$ for best case. Moreover, all the possible cases have been discussed for the formation of as efficient algorithm as possible.