Introduction to Algorithms
Assignment 4

Sourabh S. Shenoy
(UIN:225009050)

October 5, 2016

Contents

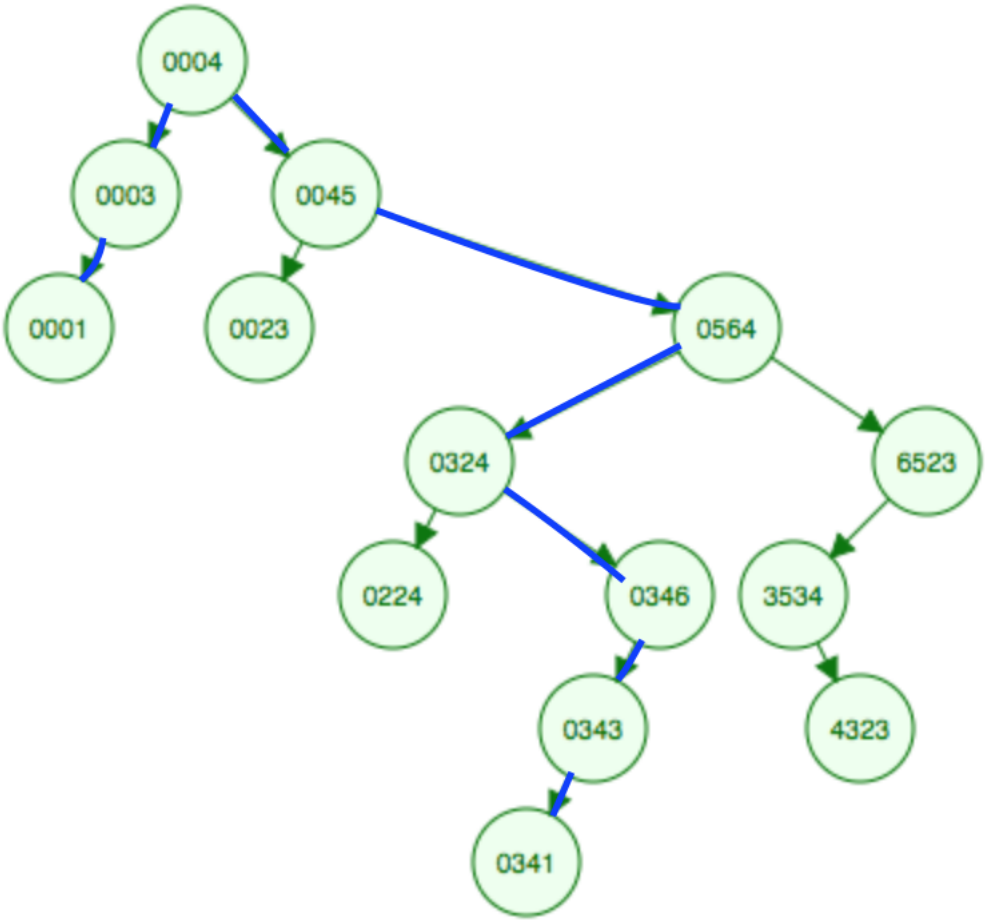
1. Diameter of the tree

1.1 Question

The diameter of a tree $T = (V, E)$ is defined as $\max_{u,v \epsilon V} \delta(u, v)$, that is, the largest of all shortest-path distances in the tree. Give an efficient algorithm to compute the diameter of a tree, and analyze the running time of your algorithm.

1.2 Solution

1.2.1 Idea:

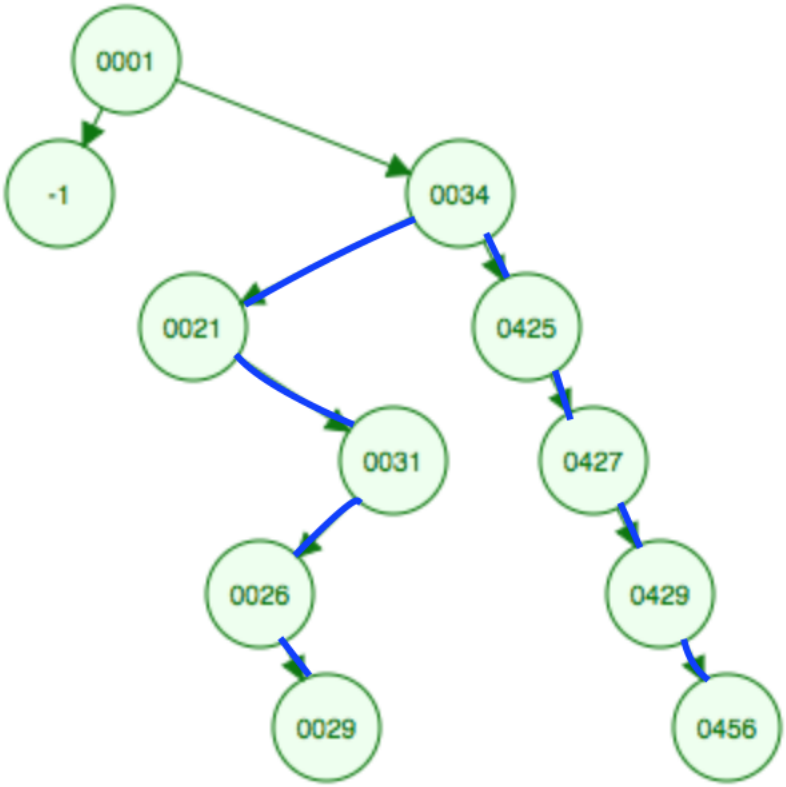


Diameter of a tree is the maximum of all distances between any two nodes of a tree. In the figure above, diameter of the tree is 9 and the diameter is marked in blue and passes through the root.

It is not necessary that the diameter pass through the root node. For instance, if the

left subtree of node 45 was as long as the right sub tree, then the resulting diameter path would not pass through the root node. Thus, we have to take care of both these cases.

The second case, where diameter does not pass through root is shown below



We can use DFS to solve this problem, since the path directly depends on the height of the tree. We can find the height of the left and right subtrees of a node and add 1 - in case the diameter includes the root. Or the diameter may completely exist in the subtree of the root. Since we have to choose the maximum of these two possibilities, we will use 2 arrays to keep track of the costs associated with them.

Let A[] be the array that keeps track of the longest path in the subtree, not including the current node. Let B[] be the array that keeps track of the longest path in the subtree, including the current node (root). We then use DFS and compute the values A[i] and B[i] for each node i in the tree. For each node, the value of A[i] and B[i] will be the maximum of the values of it's children.

1.2.2 Pseudocode:

Step 1: Set the visited (i) to 0 for all nodes i in the tree

Step 2: Set A[i] and B[i] to 0 for all nodes i in the tree

Step 3: Start with root node. Set it's visited to 1

Step 4: For each child j of i do:

if visited(j) != 0 then, recursively call DFS on j

A[i] = max (A[i],A[j])

B[i] = max (B[i],B[j]) // will give cost of path till child + ith node (root for that iteration)

Step 5: Compute max1 and max2, which will be the maximum height of left subtree and right subtree of i. (If the path contains root, then the path will be: leaf in left subtree to root + leaf in right subtree to root. In essence, this is the depth of the left and right subtrees)

Step 6: return A[i] = max (A[i],max1+max2+2) //This will return one of the two cases. If A[i] is max, it means that the path does not contain the root node, as shown in case 2 in the figure. If the second argument is returned as the max, then the path contains the root node.

Step 7: The final returned value for the tree should be the value for A[root]. This is because all paths either pass through the root, or pass through an immediate child in a subtree of the root

1.2.3 Proof of Correctness:

The diameter of a tree is either contained in one of the subtrees of the root, or is a path from leaf of one subtree to root + leaf of another subtree to root. Thus, the relation will be:

Diameter(i) = max {$\text{subtree}_{\text{left}}$(i), $\text{subtree}_{\text{right}}$(i), height of left subtree + height of right subtree + 2 }

As we can see, the diameter directly depends on the height of the tree. Using DFS helps to compute the height of the tree recursively and decide the maximum depth at each node. Once we have the depths till the root node, we can then check to see if the maximum path is located in any of the subtrees of the root or if the path goes from a leaf in one subtree, through the root to a leaf in another subtree. The above mentioned recursive relation does precisely the same thing.

1.2.4 Time Complexity:

The time complexity of this algorithm is the same as the time complexity of DFS, which is O(m+n), where n is the number of nodes and m is the number of edges. In the case of a tree, m=n-1, and hence, n=m+1, so we get O(n) time complexity.

2. Singly Connected

2.1 Question

A directed graph G = (V,E) is singly connected if u –> v implies that G contains at most one simple path from u to v for all vertices u, v ϵ V . Give an efficient algorithm to determine whether or not a directed graph is singly connected.

2.2 Solution

2.2.1 Idea:

A graph G=(V,E) is singly connected if and only if there is atmost one simple path from u to v for all u,v ϵ V. This implies that there should not be any forward edge or cross edge. Back edges may exist, as suppose we have a path from u to v and v to u, it does not violate the single connection property.

Forward edges are not allowed because if we have a forward edge (u,v) in the graph, then it means that there must be another way of reaching v, else uv wouldn't be a forward edge. If that is the case, the graph would not be singly connected. Similarly, if there exists cross edge uv, then we can reach vertex v from the root, as well as through u, which again violates the property of single connectedness. Hence, the graph cannot have either forward edges or cross edges.

2.2.2 Pseudocode:

Step 1: Set the visited (i) to 0 for all nodes i in the tree

Step 2: Start from the root node, and run DFS recursively for each node, setting visited[i] to 1 once we are done with the node

Step 3: For any node i, suppose the graph contains a forward edge or cross edge, return false

Whether an edge is forward edge or cross edge can be determined from the visited time of the edge.
if d[u] < d[v]:

Classify (u, v) as a forward edge

else:

Classify (u, v) as a cross edge

Step 4: Return True if all edges are simple edges

2.2.3 Proof of Correctness:

Let us prove the correctness by method of contradiction. We assume that we have no forward or cross edges in the graph. Suppose the graph is not singly connected, then we have 2 different paths between 2 vertices, say u and v. When we run DFS on the graph, it is going to discover one of those paths.

1) If the discovered path between u and v is the shorter path, then the other path is longer (i.e; has atleast one extra node, say 'a' in between). Then the edge a-v are at the same depth and become cross edge.

2) If the discovered path between u and v is the longer path, then the other path is shorter. This would imply that the shorter path is a forward edge in the graph, without which it is not possible that the destination 'v' is visited.

Both these statements contradict our assumption. Hence, if a graph is singly connected, it does not have forward edges and cross edges.

2.2.4 Time Complexity:

The complexity of the algorithm is O(V.E), since we use DFS traversal for each vertex to check if its edges are forward or cross edges. However, for a tree, E=V-1, so it becomes $O(V^2)$.