CSCE 629-601 Homework 4
5th October, 2016

Name: Adil Hamid Malla **UIN:** 425008306

1. Exercise 22.2-8 on page 602

INPUT: We have a $Tree = T(V, E)$, We will assume that the tree is not binary so that we can generalize the answer for other types of trees as well.
OUTPUT: Diameter of the $Tree = T(V, E)$, given by equation below:

$$Diameter = max_{u,v \in V} \delta(u, v)$$

the largest of all shortest-path distances in the tree. **Solution:** The answer to this question is tricky. If we would have considered a binary tree then the solution was trivial. But since we are assuming that the tree is not binary so we will have a problem to solve. Moreover, we can make any node $u \in V$ as our root node so it complicates out procedure.
Idea 1: Since the largest distance between two nodes can be present either in one of the sub trees or can be between two subtrees and including the root vertex. Now we can calculate the maximum height and second maximum height among the subtrees. The sum of these two plus the vertex height on both sides will give us the maximum/largest distance between these two nodes. But the problem with the implementation was i have to keep always the track of the parent subtree which was really an overhead if the number of the nodes increased substantially. So i went with other idea, where i check the longest node from any node first and then apply BFS on that.

The Algorithm can be defined as

Algorithm 1 Diameter of Tree

```
We can take any vetrtex s ∈ V as root
Diameter = BFS(T, v)
procedure BFS(T, s)
    for ∀u ∈ G.n except s do
        u.state = undiscovered
        u.distance = −1                                  ▷ distance from the node
    end for
    s.state = discovered
    s.distance = 0
    Let Q be a queue to store the values and initially Q = φ
    Enqueue(Q, s)
    while Q! = φ do
        u = DEQUEUE(Q)
        for v ∈ G.adj(u) do
            if v.state == undiscovered then
                v.state = discovered
                v.distance = u.distance + 1              ▷ Adding the height from the root node
                if v.distance > maxdistance then
                    maxdistance = v.distance
                    extremenode = v
                end if
            end if
        end for
        u.state = visited
    end while
    Now we have to run the BFS again on the extreme node we got in last run
    Diameter = BFS(T, extremenode)
    The maximum distance obtained from the second BFS is the diameter of the tree
    return Diameter
end procedure
```

Idea 2: One more algorithm can be doing a BFS traversal of the Tree and keep the track of the node farthest from the node, then we can start BFS from that node. Now the farthest point from the second traversal of BFS will gives us the largest of all the shortest-path distance in the tree aka Diameter.
Now Lets talk about the proof of optimality and correctness of the algorithm. Since we know that the tree is an undirected graph and acyclic. Also we know that there is only one path which lies between two nodes. Thus when we first time traverse the tree using the BFS it ensures that each each discover is made first and last time and there is no other path coming back to the discovered node. Furthermore, now when we reach the farthest from the root node, it lies in one of the subtree of the root node. Now lets assume that the extreme node lies in any subtree, then by the definition of the tree, the other subtrees will now be the sub trees/ sub-sub trees of this node. Now when we ran the BFS first time it ensured that we reached one of the most extreme nodes from the root, the root can be anything. Further we ran the BFS from the extreme node we obtained from the first BFS run, this will also traverse the tree and it will find a node which is at maximum distance from the root node of present BFS run that is extreme node.
Since we know that the edges of the tree is equal to number of nodes plus one i.e. E = V + 1 The time complexity of this algorithm will be $O(2(|V| + |E|))$. Since there is a linear relation between E and V we can further say that the order of our algorithm is $O(|V|)$ or $O(|E|)$

1. Exercise 22.3-13 on page 612

INPUT: We have a Directed Graph $G = (V, E)$, We will assume that the graph can be cyclic or acyclic.
OUTPUT: Whether the given graph is singly connected or not.
By singly connected we means that if we have at most one simple path from $u \to v$ for all vertices $u, v \in G$.
Solution:
Now by the definition of the singly connected, we just have to check whether there is no forward or cross edges in a DFS traversal of the graph. If we have the forward or cross edge then there is not only one simple path between the nodes thus not singly linked.

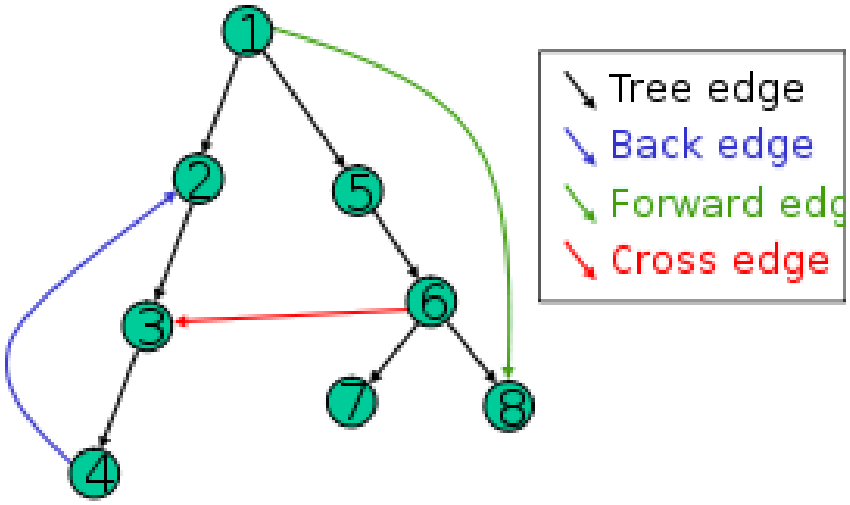

Figure 1: Different Edges of the graph source:Wikipedia

Now to achieve this checking of the forward and cross edges we have to run the DFS algorithm on each node. Moreover, the graph is singly connected even of back edges exist, as back edges implies there is a path $u \to v$ and $v \to u$, which is consistent with the definition of single connected graphs. This can be implemented really easily by running the DFS on each vertex and if we encounter an adjacency from a discovered edge to visited ($finished$) edge then the graph is not singly connected.
In the graph above, we can see that the presence of the forward edge or cross edge results in the graph to have more than one simple paths which is against the criteria of a graph to be simply connected.
The algorithm's pseudo code is given:
The correctness of this algorithm lies in the fact that what we have ensured if any of the condition leading to the graph not being singly connected is covered by our algorithm. There are two conditions which can lead to a graph not being single linked is presence of the cross edge and forward edge. When we are traversing the graph, the cross edges are handled by checking if the node we are trying to discover is already finished, if it is so then we know that we have discovered a cross edge from a node, thus the graph is not singly connected. Similarly the forward edge is also taken care of. If there is one node which has already been visited and it is

Algorithm 2 Singly Connected Graph

```
1: We run DFS on each node s ∈ V
2: DFS(G, s)
3: procedure DFS(T, s)
4:     for ∀u ∈ G.n except s do
5:         u.state = undiscovered
6:         u.distance = −1                                  ▷ distance from the node
7:     end for
8:     s.state = discovered
9:     s.distance = 0
10:    Let S be a Stack to store the values and initially Q = φ
11:    Push(S, s)
12:    while S! = φ do
13:        u = TOP (S)
14:        POP(S)
15:        for v ∈ G.adj (u) do
16:            if v.state == undiscovered then
17:                v.state = discovered
18:                PUSH (S, v)                              ▷ Pushing the new the height from the root node
19:            else
20:                if v.state == visited then
21:                    return Graph Is not Singly Linked
22:                end if
23:            end if
24:        end for
25:        u.state = visited
26:    end while
27:    return Graph is Singly Linked
28: end procedure
```

being discovered from some other node then it means that there are two paths to the same node which revokes the criteria of the graph being singly connected.
The time complexity of the given algorithm is $O(|V|(|V| + |E|))$