CS 180 - Homework 2

Darren Tsang, Discussion 1I

Produced on Sunday, May. 03 2020 @ 11:27:28 PM

Question 1

We are trying to prove that in any binary tree the number of nodes with two children is exactly one less than the number of leaves. We will proceed with induction.

Base Case: Assume that there is a binary tree with exactly one node. Thus, there are zero nodes with two children, which satisfies the condition we are trying to prove.

Inductive Hypothesis: Assume that a binary tree T has n nodes, and the number of nodes with two children is one less than the number of leaves.

Inductive Step: Assume the inductive hypothesis holds. We will obtain a binary tree T' by adding a node (denoted u) to one of the nodes (denoted v) from T that either has zero or one children. This results in two cases:

Case 1: Node v had zero children before u was inserted. Now, v is no longer a leaf, and u is a leave. The number of nodes with two children and the number of leaves remained the same. Thus, our coniditon is satisfied

Case 2: Node v had one child before u was inserted, which means v was not a leaf. Now, v has two children, which means the number of nodes with two children increases by one. Additionally, the number of leaves increases by one because u is now a leave. Thus, our condition is satisfied.

Question 2

```
L[0] = {s}

initialize visited[v] = false, $\forall v \neq s$
visited[s] = true

initialize shortest[v] = $\infty$, $\forall v \neq s$
shortest[s] = 0

initialize count[v] = -1, $\forall v \neq s$
count[v] = 1

i = 0

while (L[i] $\neq \emptyset$){
    for each u $\in$ L[i] {
        for all v such that (u,v) $\in$ E {
            if (i + 1 = shortest[v]) {
                count[v] = count[v] + 1
            } endif

            if (visited[v] = false){
                add v to L[i + 1]
                visted[v] = true
                shortest[v] = i + 1
                count[v] = 1
            } endif
        } endfor
    } endfor
    i = i + 1
} endwhile
return count, shortest
```

The algorithm above is essentially the BFS algorithm with two new arrays, shortest and count. Shortest will keep track of the shortest path from the source node s to another node v. The shortest path will be when the node v is first visted. At this point, we will set $count[v] = 1$. Then, if the node v is visited again, the algorithm will check if the current path has the same length of the shortest; if so, count will increase by one.

We know that the BFS algorithm from class is $O(m)$. Our algorithm is the same, but with an additinal for loop that does not affect the time complexity, which means it is also $O(m)$

Question 3

Part A

```
max = -1

for all v $\in$ V {
  run algorithm from Q2 with v as s
  denote $shortest_v$ as the array $shortest$ that was retured from the previous step
  if (max < max($shortest_v$)) {
     max = max($shortest_v$)
  }
}
return max
```

The algorithm above runs in $O(nm)$ time because we are running the algorithm from Q2 n times, and the algorithm from Q2 runs in $O(m)$ time.

Part B

```
findDiameter(root){
  if root is NULL {
     return 0
  }

  leftDiameter = findDiameter(left node of current root)
  rightDiameter = findDiameter(right nide of current root)

  leftHeight = findHeight(left subtree of current root)
  rightHeight = findHeight(right subtree of current root)

  answer = max(leftDiameter, rightDiameter, leftHeight + rightHeight + 1)
  return answer
}
```

In the algorithm above, we can see that findDiameter() is called for every node, which makes the run time $O(n)$. The algorithm is doing what was suggested on the homework; the diameter and longest path is being calculated for every subtree.

Question 4

Part A

```
obtain the graph $G' = (v', e')$ from topological sort on $G$
if($\exists$ edge from ${v_i}'$ and ${v_{i+1}}'$ for all i = 1,...,k-1){
    return true
} else {
    return false
}
```

The bottleneck for the algorithm above is performing the topological sort, which is $O(m)$. This means that the overall algorithm is $O(m)$ as well.

The algoirthm is correct because the checking of edges shows that there is a path P from one node u' to another node w' that will pass all other nodes. Thus, if such P exists, there exists $P' \subseteq P$ between any two nodes, which satisfies the property of semi-connectedness.

Part B

Assume, for the sake of contradction, that G' is not a DAG. This means that there exists a a cycle in G'. If there exists a cycle, that means that there was a cycle between the SCCs produced by Kosaraju's algorithm, which is a contradiction of the property of Kosaraju's algorithm that the SCCs are maximal.

Part C

```
obtain the graph $G' = (V', E')$ from the algorithm proposed in Q4b
perform topological sort on $G'$

if ($\exists$ edge between ${V_i}'$ and ${V_{i+1}}'$ for all i = 1,...,k-1){
    return true
} else {
    return false
}
```

In the algorithm above, true is returned if there is a path from ${V_i}'$ to ${V_j}'$ with $i < j$. This means that the nodes in the SCC represented by ${V_i}'$ has a path to every node in the SCC represented by ${V_j}'$. Thus, the algorithm is correct and will only ouput true if the original graph G is indeed semi-connected.