Introduction to Analysis of Algorithms — September 23, 2020
Boston University — CS 330
Professor Adam Smith, Dora Erdos — Fall 2020

Homework 3

Collaborators: Patrick Kuzdzal

Problem 1. *Alternating Paths (10 pts.)*

Let $G(V, E, s, c)$ be an undirected graph with s as a source node. Each of its edges $e \in E$ is either blue or red, and c_e denotes the color of edge e. Let all edges adjacent to s be blue.

A path between nodes u and v in graph G, called $P_{u \to v}$, is a *shortest* path if there is no other path between u and v with fewer edges. $P_{u \to v}$ is called *simple* if it never visits the same edge or node twice. A simple path P in G is called *alternating* if the edge colors in the path alternate between red and blue. Find an algorithm that takes as input $G(V, E, s, c)$ and computes whether there is a *shortest alternating path* from s to every other node. If that is the case, then the output should be the alternating path to each node, which should be represented as ordered lists of edges. Otherwise, return "No".

ANSWER

The following algorithm utilizes an altered form of the BFS algorithm shown in class, that, instead of just returning the distances from the source node, will return a tuple, with the first element being the distances, and the second the paths from source to each node.
It would do this by having another array ($paths$) inside itself that would keep track of all paths, by storing each node in the correct spot in the array, as it traverses the graph. In addition to this, the algorithm will handle two identical paths between the same two nodes. It will do this by modifying the if statement that checks if the path is shorter to include and elseIf for the same length, in which case it would create another entry for a path between the two nodes.
On another note, every time you see **Output** in the pseudocode, it is simulating a return, and would not execute anything past that point.
In addition to this, the algorithm utilizes another auxiliary method that loops through the paths to check for equal paths.

As for the algorithm itself, all it does is loop through every vertex in the graph, check if it has a path from the source node (s), by verifying if it's inside the distance array. If any of them aren't it returns "No", otherwise, it keeps going.
If the first check is passed, then it'll loop through every edge that connects s to the vertex in question, and sees if the colors of these edges are alternating, if a single one is found not to be alternating it'll return "No".
Finally, if no "No"'s are returned, and all the loops are complete, the algorithm will return the paths list.
Due to the structure of the ckecks in place, the algorithm will always return a "No" if any single edge doesn't fulfill all requirements.

As for the runtime, the algorithm will take the runtime of BFS + the changes to it + the runtime of the other loops. Since the changes to BFS are constant time O(1), they can be ignored, so we are left with $O(V+E)$ for BFS plus $O(V^2)$ for the remainder loops. Since in the worst case scenario we have $E = V^2$, we are left with $O(V^2) + O(V^2) = O(2V^2)$ for runtime.

Algorithm 1: AlternatingPaths(G(V,E,s,c))

```
Input: $G(V, E, s, c)$
Adj = adjacency list of graph G (Taken from E);
{dist, paths} = AlteredBFS(Adj, s);
for Each u in V do
    if u! = s then
        if dist[u] == undefined then
            Output: "No"
        prevColor = null;
        for Each e in paths[u] do
            if prevColor == null then
                prevColor = $c_e$;
            else
                if prevColor == $c_e$ & & noOtherPathEqual(e) then
                    Output: "No"
                else if !noOtherPathEqual(e) then
                    paths.remove(e);
                    prevColor = $c_e$;
                else
                    prevColor = $c_e$;
Output: paths
```

Problem 2. *Binary Trees (15 pts.)*

Let us represent a binary tree in the computer as follows. Each node of the tree T is a record, r is the root node. In a node v, if $v.left \neq$ `null` then it is the left child of v; similarly for $v.right$.

a. (5 points) Write a recursive program (in pseudocode) to compute the number of leaves.

ANSWER

The following algorithm take a node of a Binary Tree, count the leaf nodes present in the tree. It does this by recursively calling itself to count the leaf nodes of the its own child nodes etc... This works due to couple checks and return statements, as the algorithm will return 0 if the node is null (ie: we're outside a path), 1 if both of the child nodes of the node in question are null (ie: we're inside a child node), or the count of the leaf nodes bellow itself if none of the above happen (ie: we are in the middle of a tree).
The only true edge case is when one of the child nodes is null and the other isn't, but this is fine, as when you call the algorithm on the null one it'll return 0 (which is true since that node doesn't

exist) + it will return however many leaf nodes there are in the other path.

As for the runtime, this algorithm will execute in O(n) time.

Algorithm 2: CountLeaves(node v)

```
   Input: $nodevisanodeinsideaBinaryTree$
1  if $v == null$ then
   └ Output: 0
2  else if $v.left == null\ \&\ \&\ v.right == null$ then
   └ Output: 1
3  else
   └ Output: $CountLeaves(v.left) + CountLeaves(v.right)$
```

b. (10 points) **Kelinberg and Tardos, Chapter 3, Problem 5**

Show by induction that in any binary tree the number of nodes with two children is exactly one less than the number of leaves.

ANSWER

Base Case: n=1 In this case, we have a Binary Tree with a single node, that has 0 children, so $nodesWithTwoChildren = 0$, meanwhile, the tree has 1 leaf so $leafNodes = 1$, as such, this holds for the Base case.

Hypothesis: n=k holds For the sake of argument lets assume this holds for any n so that $n = k$.

Theory: n=k+1 holds Now lets see if this still holds for any n so that $n = k + 1$ nodes.

Now we have a tree with k+1 nodes, in which we have $leafNodes = a$ and $nodesWithTwoChildren = a - 1$. Since we know that a Binary Tree cannot have nodes with more than two children, then we have 2 options here, we add it to an existing node that has 0 children, or an existing node that has 1 child.

In the first case (the parent node doesn't have a child), we are creating a new leaf, however, since we are attatching it to a node that use to be a leaf, we are left with $leafNodes = a+1-1 = a$, and since we aren't creating any nodes with two children, (since the node we added has 0 children and it's parent node now has 1).

As for the second case, we are adding a node to another node that already has a child, so we are creating a new node with two children, so $nodesWithTwoChildren = a - 1 + 1 = a$, on the other hand, we have also added a new leaf, since the parent node wasn't a leaf, and the node we added has no children, so we have that $leafNodes = a + 1$.

Since by the nature of a binary tree we cannot add nodes anywhere else, if our Hypothesis is true, out theory is also true.