Homework 3

Student: Justin DiEmmanuele
Collaborators: Shilpen Patel, George Padavick, Matthew Gilgo

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".

1. **A precise description of the algorithm in English and, if helpful, pseudocode**

   To achieve the desired results, some changes need to be made to the BFS algorithm as described in section 3.3 of *Algorithm Design* (Kleinberg and Tardos). We need to handle the extra constraint that a path must be alternating as described in the problem statement. To do this, we need to track the colors of the previous edge as well as the color of the edge leading to the children we are investigating. Then, in addition to checking if a node has been discovered yet, we will also check if the previous edge color is different from the edge color leading to the child. This effectively only discovers a node if it satisfies the alternating requirement.

   We assume that we take an adjacency list with tuples containing adjacent node and color. In all our data structures, we maintain the information on the color of the edge. When we chose to investigate a node for its adjacent nodes we pull that nodes previous color. As we visit each adjacent node's edge color to the previous one and only add the node to the tree and investigation queue if its edge's color is not equal to the edge we are investigating from.

   The output of the algorithm is a tree stemming from the source root to the leaves in which the level corresponds to the distance from that source. To get the alternating path to each

node, we must traverse the tree from each node to the source (if the path exists).

Algorithm 1: GetPaths($G\,(V, E, s, c)$)

Input: G is an undirected graph as described in the problem statement
Input: s is the source node of graph G

```
1  bfs_tree = BfsAlternatingColor (G) ;
2  n = length(A) ;
3  paths = EmptyArrayLength_n ;
4  for i = 1 to n do
5      path = [] # empty list parent = i ;
6      while parent ≠ s do
7          if parent in bfs_tree then
8              child = parent ;
9              parent = bfs_tree[child].get_parent() ;
10             path.append ((parent, child));
11         else
               Output: EmptyList[]
12     path.reverse() ;
13     paths[i] = path ;
```

Output: *paths*

In this question we are interested in if there is a path that is both the shortest path and alternating between the source and every other node. Since out modified version of BFS will return all the alternating paths and the original version will return the shortest paths, we can simply compare the length of the output for each node. This is the length of each array in *paths* as returned in Algorithm 1. We run Algorithm 1 using the modified alternating BFS and original BFS then run:

Algorithm 2: ComparePaths(*path_alternating*, *path_normal*)

Input: *path_alternating* is the result of *GetPaths* ( ) when run on the modified BFS - an array length n with each index i containing an array of tuples that represent the edges that make up the path to node i
Input: *path_normal* is the same as *path_alternating* but run using the original BFS

```
1  for i = 1 to n do
2      if paths_alternating[i]! = paths_normal[i] then
           Output: "NO"
```

Output: *paths_alternating*

This gives the desired result - the alternating path to each node from the source given that it is also the shortest path or "NO".

2. **A proof of correctness**

- Modified Breadth-First-Search
  The modified BFS algorithm used in algorithm 1 follows the same claim made in KT - claim 3.3. In the original version of the algorithm, any new node that is found is added to the graph as long as it has not been discovered. For the special case, we just add

a further reaching constraint that the node must not only have not been discovered, it must have a different color then the edge leading to its parent.

Since the only changes to BFS are simple statements that are guaranteed to terminate we can be confident the modified version will terminate.

- BFS Tree Traversal

  Given that the returned BFS tree is correct, we must prove algorithm 1 correctly traverses that tree to return the set of edges leading to each destination node. In a BFS tree, each node appears once and each node has only 1 parent. This is true because of the "if discovered" check before a node is added to the tree and the fact that there is no option to give a node two parents per the definition of a tree.

  Because of these facts, we can simply start at each destination and iteratively call for its parent until we reach the source or root node. If a node does not exist in the tree it has no valid alternating path per claim 3.3.

  Since every node of the tree has just one parent per the definition of a tree, and we know BFS returns a tree, we can be sure the program will terminate.

- Comparison

  The comparison simply goes index by index of the array and compares the alternating version and original BFS versions. If they are different the algorithm immediately halts and returns "No". Otherwise, the program runs through the rest of the nodes' paths then outputs the shortest alternating paths for the graph.

  The comparison runs a finite n times. We can be sure the program will terminate.

3. **An analysis of running time and space**

- Running Time
  - Modified BFS Running Time
    The modified BFS will still run in $O(m+n)$ time per claim 3.11 in KT. The only changes to the algorithm are $O(1)$ - look-ups for edge color and logical comparisons to check if colors are alternating.
  - Tree Traversal Running Time
    The tree traversal shown in algorithm 1 will run in $O(n)$ time. Where n is the number of nodes in the tree. At most a path can be length n since a path is the number of edges separating two nodes. The tree traversal will be applied to at most n nodes.
  - Comparison Running Time
    The comparison will need to iterate through the shortest paths for n nodes. This will smaller than iterating through the graph adjacency list which has a run time of $O(n+m)$. The shortest paths contain the same number of nodes as the adjacency list but fewer edges since not all are needed to describe the shortest path.
  - This makes the running time $O(n+m)$ limited by the BFS search algorithm.
- Space Complexity

  The largest structure the algorithm will need to handle is the adjacency-list passed representing the graph. All data other data structures will take less space than this as

they do not need to store the complete information of the graph. This makes the space complexity $O(n+m)$ per the lecture 4 slides.

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.

1. A precise description of the algorithm in english and, if helpful, pseudocode

Algorithm 3: RecursiveLeafCount(r)

Input: r is the root node of a binary tree

1 **if** *r is Null* **then**
  Output: 0

2 **else if** *$r.left()$ and $r.right()$ are Null* **then**
  Output: 1

3 **else**
  Output: $RecursiveLeafCount\,(r.left\,()) + RecursiveLeafCount\,(r.right\,())$

2. A proof of correctness

There are a few cases we can investigate and test against this algorithm to understand if it returns the desired result.

- Root node r has no children.

  In this case, the algorithm will trigger the else if statement on line 2 since both children are null. The output will correctly be 1 leaf.

- Root node r is *null* or does not exist.

  The first if statement will be triggered and zero is correctly returned.

- A child exists.

  The final else statement is executed and the function is called on both children.

For each recursive call one of these cases will apply and they will still hold true. We know that this algorithm will terminate because the algorithm can only move down in layers of the tree. As we move down in layers, the portion of the tree that has not been visited gets smaller. The function will be applied recursively until the entire tree has been visited.

3. An analysis of running time and space

The operations in the $RecursiveLeafCount\,(\,)$ are all $O\,(1)$. They are simply accessing variables and running logical comparisons. The question then becomes, how many times will the algorithm be run on a tree of n nodes? The algorithm only makes calls down the levels of the tree to every child. The algorithm therefore will be $O\,(n)$.

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.

- Base Case

  Take the simplest case - a tree with one root node with two children. These children have no children of their own so they are leaves. Let N represent the number of nodes with two leaves and L represent the number of leaves. We want to prove $L - 1 = N$

  $$L - 1 = N.$$

  $$2 - 1 = 1.$$

- Induction Step

  Take a node with two leaves at the highest level of the tree. The tree has N_1 nodes and L_1 leaves. If we remove both leaves, the tree loses two leaves, but the parent node becomes a leaf itself. Net, the tree loses one leaf. The tree also loses a node since the node in question becomes a leaf. This leaves us with $N_1' = N_1 - 1$ nodes and $L_1' = L_1 - 1$ leaves.

  If we plug this into the equation...

  $$L_1' - 1 = N_1'.$$

  $$(L_1 - 1) - 1 = (N_1 - 1).$$

  $$L_1 - 2 = N_1 - 1.$$

  $$L_1 - 1 = N_1.$$

  We end up with the original statement: $L - 1 = N$