# Homework 4

**Due Wednesday, September 30 at 11:59 PM**

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

## Homework Guidelines

**Collaboration policy** Collaboration on homework problems, with the exception of programming assignments and reading quizzes, is permitted, but not encouraged. If you choose to collaborate on some problems, you are allowed to discuss each problem with at most 5 other students currently enrolled in the class. Before working with others on a problem, you should think about it yourself for at least 45 minutes. Finding answers to problems on the Web or from other outside sources (these include anyone not enrolled in the class) is strictly forbidden.

*You must write up each problem solution by yourself without assistance, even if you collaborate with others to solve the problem.* You must also identify your collaborators. If you did not work with anyone, you should write "Collaborators: none." It is a violation of this policy to submit a problem solution that you cannot orally explain to an instructor or TA.

**Solution guidelines** For problems that require you to provide an algorithm, you must give the following:

1. a precise description of the algorithm in English and, if helpful, pseudocode,
2. a proof of correctness,
3. an analysis of running time and space.

You may use algorithms from class as subroutines. You may also use any facts that we proved in class.

You should be as clear and concise as possible in your write-up of solutions.

A simple, direct analysis is worth more points than a convoluted one, both because it is simpler and less prone to error and because it is easier to read and understand. Points might be subtracted for illegible handwriting and for solutions that are too long. Incorrect solutions will get from 0 to 30% of the grade, depending on how far they are from a working solution. Correct solutions with possibly minor flaws will get 70 to 100%, depending on the flaws and clarity of the write up.

1. (**Unique simple path (15 pts)**) Given a **directed** graph $G = (V, E)$, vertex $s$ has *unique simple paths to all vertices* if for every $v \in V$ that is reachable from $s$, there is at most one simple path from $s$ to $v$ (Recall that a path is simple if all vertices on the path are distinct).

   The figure below gives example graphs and points out pairs of vertices that do and do not have unique paths. Go over the examples to make sure you understand the definition.

   (a) For each of the following types of graphs, is it the case that in every graph of that type, every vertex $s$ has a unique simple paths to all reachable vertices? For each type of graph, give either a short proof (one or two sentences), if yes, or a counterexample, if no.

      i. Cycles[1]
         - **Yes**
         - A cycle is a sequence of **unique** nodes that cycles back to where it began as defined in section 3.1 of *Algorithm Design*. If all the nodes in the sequence are unique than the edges between them are unique. Since all the edges in the graph are involved in the cycle, there must be a simple unique path to all vertices. If there was not, there would be a sub cycle in the graph.

      ii. DAGs (directed acyclic graphs)
         - **No**
         - The following counterexample is a DAG as it has no cycles but it also has **two** simple paths from 1 to 3.

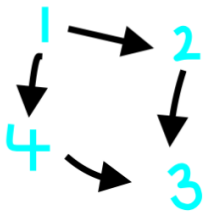


Figure 1: 1.a.ii Counterexample

      iii. Trees
         - **Yes**
         - Any node pair with two paths between them must have a node in the path that has two edges leading into it and another node in the path must have two edges leading out of it. This is impossible because every node in a tree has one parent.

      iv. Strongly connected graphs
         - **No**
         - The following figure shows a counterexample. The graph is a strongly connected directed graph as there are paths to and from each set of vertices. There are two paths from 1 to 4 - (1, 2, 4) and (1, 4).

[1] A directed graph on $n$ vertices with directed edges of the form $(i, i+1)$ for $i = 1, ..., n-1$ and the edge $(n, 1)$.

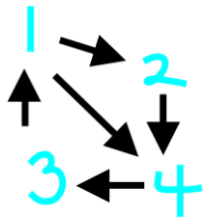


Figure 2: 1.a.iv Counterexample

(b) Give an efficient algorithm that takes a directed graph $G$ and a vertex $s$ as input and checks whether $s$ has unique simple paths to all other vertices reachable from $s$. Your algorithm should output "no" if at least one vertex $v$ has more than one path from $s$ to $v$; otherwise it should output "yes", together with a tree of paths from $s$ to each reachable vertex in $G$. For full credit, it should run in $O(m+n)$ time *[Hint: Use DFS. Think about different kinds of non-tree edges.]*.

- **A precise description of the algorithm in English and, if helpful, pseudocode**
  - To achieve the desired result, a slight modification will be made to the DFS algorithm presented in the CS330 lecture. The code presented in lecture is in

black and the changes are in teal.

**Algorithm 1:** UniqueSimplePaths(G, s)

**Input:** $G$ A graph in adjacency list form
**Input:** $s$ source node to look for unique simple paths

```
output = "Yes";
for each u in G.V do
    u.color = WHITE
time = 0 ;
DFS − Visit (G, s) ;
```

**Output:** *output*

**Algorithm 2:** DFS-Visit(G, u)

**Input:** $G$ A graph in adjacency list form
**Input:** $u$ edge to visit

```
time = time + 1 ;
u.d = time ;
u.color = gray ;
for each v in G.Adj[u] do
    if v.color = WHITE then
        v.p = u ;
        DFS − Visit(G, u)
    else if v.color = BLACK then
        output = "No"
u.color = Black ;
time = time + 1 ;
u.f = time ;
```

From the lecture, the node label of white means a node has not yet been explored, Grey means the node has been discovered, and black means the node has been finished. The main thing we need to check if there is more than one simple, unique path to a node is to know if we come across a finished (Black) node when exploring another node. We only care about paths from the source node, so we only explore from there. This allows us to not have to worry about cross edges.

- **A proof of correctness**
  - Our claim is that if the DFS tree is exploring a node and comes across a finished (black) node, the edge it is exploring creates a second simple path.
  - If we find a finished node $a$ while exploring node $b$, we know that node $a$ is not an ancestor of $b$.
    - We know this because if $a$ is an ancestor of $b$ it would still be in process of exploration. DFS goes deep before going wide. Any node currently being explored has all it's ancestors as unfinished.
    - We care if $a$ is an ancestor of $b$ because if a non-tree edge is to an ancestor of itself it can not violate the unique simple path condition. The path it would

create is not simple as node $a$ would be repeated.

- We will not see cross edges since we are only running BFS from the source node. This means we do not have to worry about cross edges. Cross edges would have nothing to do with simple paths from the source.
- Any remaining non-tree edges are either forward edges or edges to a non-ancestor family member like an uncle. Edges of this type open up a second simple path to the finished node. Since the node is not an ancestor there must be a split where the two paths taken allow for both paths to be simple.

- **An analysis of running time and space**
  - As the only changes to the algorithm are constant time comparisons, the running time and space will be the same as for normal DFS.
  - Running time is $O(n+m)$ where $n$ is the number of nodes in the graph and $m$ is the number of edges.
  - Space complexity is $O(n+m)$ as we must maintain the adjacency list graph representation.

2. (**Lazy Hiker (10 pts)**) Suppose you are planning a hike in a forest with many paths. You want to start and end at the parking lot without visiting the same area of the forest twice. In addition, you get lost easily, so you want to choose the hike with the least possible number of trail intersections.

   You have a map, which you can interpret as a graph $G$: each vertex is a trail intersection (or start/end point) and each edge is a section of trail between two intersections. Note that $G$ is undirected.

   Write an efficient algorithm that takes as input a graph $G = (V, E)$ and a "parking lot vertex" $p$ and outputs the a loop hike (starting and ending at $p$) that visits the smallest possible number of vertices while never covering the same trail segment twice. If there are no such loop hikes, your algorithm should output "no loops from $p$". For full credit, your algorithm should run in $O(n + m)$ time. See example graphs and correct outputs below.

   *Hint 1:* Run BFS starting from $p$ and divide the BFS tree into subtrees rooted at the children of $p$. For each vertex $v$, figure out and store the subtree that it is part of.

   *Hint 2:* The following questions may help you get thinking in the right direction. Do not hand these in.

   - Prove that if you run BFS on an undirected graph, for every non-tree edge $(u, v)$ either $u$ and $v$ are in the same BFS layer, or $u$ and $v$ are off by one layer (e.g. $u$ is in layer $k$ and $v$ is in layer $k \pm 1$)
   - Non-tree edges create cycles. What do the numbers of the layers of the endpoints tell you about the length of the cycle and edge creates?

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

**Algorithm 3:** LazyHiker(G, s)

```
Input: G A graph in adjacency list form
Input: s source node to look for the shortest cycle
discovered[] = empty list ;
discovered[s] = True ;
nodes_by_layer[] = empty list length n;
nodes_by_layer.append([s]) A list containing s ;
layer_counter = 0 ;
subtree_tracker = empty list to store which subtree each node is in ;
while len(nodes_by_layer[layer_counter]) > 0 do
    for node in nodes_by_layer[layer_counter] do
        for adj_node in G.adj[node] do
            if layer_counter = 0 then
                subtree_tracker[adj_node] = adj_node
            if not discovered[adj_node] then
                discovered[adj_node] = True ;
                adj_node.parent = node ;
                nodes_by_layer[layer_counter + 1].append(adj_noe);
                if layer_counter > 0 then
                    subtree_tracker[adj_node] = subtree_tracker[node]
            else
                if subtree_tracker[adj_node] ≠ subtree_tracker[node] then
                    Output: node, adj_node
    layer_counter + +
Output: "no paths from p"
```

If BFS finds an edge that is already discovered and the discovered edge is in a different child of $p$'s sub tree, we know that that path including the vertices returned ($adj\_node$ and $node$) and their ancestors is the shortest cycle with no repeated edges including $p$.

To calculate the nodes in the loop, we take the returned nodes from $LazyHiker(\ )$ and find all their ancestors and load them into a list. The resulting list length is the length of the path. One could do this simple by iteratively calling $node.parent$ on each of the returned nodes. This would run maximum $2n$ times and therefore has a run time of $O(n)$.

- **A proof of correctness**
  - If we find a non-tree edge, we know that that edge creates a cycle.
    - This is discussed in lecture
  - For a cycle including $p$ to not have any repeating edges, the end of the cycle in which we return to $p$ must be a different edge than when we first leave $p$.
    - The algorithm tracks which tree of $p$'s children each node belongs to by keeping an integer array in which each index corresponds to a node in $G$.

    - Using this array we can check as we come across a node that has already been discovered (a non-tree edge) if that node belongs to the current explored node's sub tree. If it is not, we know that this is a cycle that includes $p$ and travels through that edge.
    - This is true because of the definition of a tree. We know BFS creates a tree. If we have a cycle in which an edge crosses from one of $p$'s children's trees to another, the cycle must go through $p$ since both sides of that edge share $p$ as an ancestor.
  - The cycle we chose must be the shortest cycle including $p$.
    - We know that if we chose the first edge that has the properties described above, we will chose the shortest possible cycle.
    - The BFS level corresponds to the distance of a node from the root $p$. Since BFS explores the lower levels first, we will always find the closest edge that creates a cycle and therefore the shortest cycle.
- **An analysis of running time and space**
  - The algorithm runs in $O(n+m)$ time. The time complexity does not change from what was proved in the book and in lecture because the only operations that were added run in $O(1)$ or $O(n)$ time. Added operations are: initializing sub tree tracking array, comparisons, filling of the array with values, accessing the array, and crawling the BFS tree for ancestors of the outputted nodes.
  - The algorithm's space complexity does not change from its original $O(n+m)$. The only added data structure is the sub tree tracker which has space $O(n)$.