Introduction to Analysis of Algorithms September 26, 2020
Boston University CS 330
Professor Adam Smith, Dora Erdos Fall 2020

Homework 4

Due Wednesday, September 30 at 11:59 PM

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]
      ii. DAGs (directed acyclic graphs)
      iii. Trees
      iv. Strongly connected graphs

      ANSWER

      i. Yes. By their own nature, Cycles inherently have a simple unique path that connects any vertex to any other vertex, so this is true for Cycles
      ii. No. DAG's could potentially have multiple paths from a vertex v to a vertex v_2, all it would be needed is for it to have a 3rd vertex v_3, and for there to be a directed edge from v to v_2 and from v to v_3 and then from v_3 to v_2
      iii. Yes. Trees by definition are undirected graphs in which any two vertices are connected by exactly one path, so this is true for Trees
      iv. No. Since a strongly connected graph is, by definition, a graph where, if there is a directed edge between vertex a and vertex b then there is a directed edge between b and a, then it is possible for there to be multiple simple paths between two vertices. All we need if a directed edge between vertex a and b, and a directed edge between a and c, and c and b, and we already have multiple paths.

   (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.]*.

      ANSWER
      The algorithm below will make use of a changed version of DFS that either returns the regular output of all connected notes to the source vertex, or simply the string "no" if, when going through the graph, it finds that a specific node has a directed edge towards an already been discovered. This means that changedDFS will only return a list of connected nodes IF there is no way to get to an already discovered node by using directed edges that stop in another node. This is easily done by altering the if statements inside

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

DFS to, when checking for directly connected nodes, not only check for non-discovered nodes (that it then goes and discovers) but also discovered nodes, in which case it returns "no". This also doesn't change the runtime of the DFS algorithm, which means it will still run in $O(m + n)$.

After the algorithm runs DFS, it'll check if the answer is "no", in which case it returns "no", otherwise, it will compare the answer to the existing list of vertices, and check if they are all in the answer, in which case it returns the list. Otherwise, it'll return "no".

As for the runtime, since changedDFS runs in O(m+n), and the remaining of the algorithm runs in O(n), then the total would be O(m+2n), which is just O(m+n).

Algorithm 1: UniqueSimply(G(V,E), s)

```
  Input: $(G(V,E), s)$
1 $response = AlteredDFS(G, s)$;
2 if $response ==$ "$no$" then
      Output: $response$
3 for $each\ v \in V\ that\ != s$ do
4     if $response.indexOf(v) == undefined$ then
          Output: $no$
  Output: $response$
```

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?

   ANSWER

   This algorithm will make use of an altered BFS that will return the "BFS Tree" of nodes alongside the distance, this wont increase time complexity since the only difference is instead of only adding the distance to the distance array, it'll also add the node as a child of the previous node. So BFS will continue running on O(m+n) time.
   If we run this altered BFS on the graph starting in a node p and then organize all other nodes based on which of the subtrees formed by p's children they belong. With these "sets" of nodes, then we can start looking for edges that aren't in any of the trees and connect two nodes from different subtrees. If so we found a cycle, and from here all we need to check if its the first cycle that appears in the trees, ie: check if its the highest level cycle in the trees. If it is, then it's the smallest possible path from the node p back to itself, without ever going over an edge twice.

```
Algorithm 2: LazyHicker(G(V,E), p)
Input: $(G(V,E),p)$
1  $Adj$ = adjacency list of graph G (Taken from E);
2  $tree, dist = BFS(Adj, s)$;
3  $groupList = tree[p].subTrees()$ /*This method returns the subtrees formed by the
     children of the node p */;
4  $nodeOne = null$;
5  $nodeTwo = null$;
6  for each group in groupList with index i do
7      $Q = FIFOQueue$;
8      $Q.enqueue(group[0])$;
9      $node = null$;
10     while Q not Empty & & $node! = null$ do
11         $n = Q.dequeue()$;
12         for neighbour in Adj[n] do
13             if $neighbour \notin group$ then
14                 $node = e$;
15                 if $nodeOne == null$ then
16                     $nodeOne = node$;
17                     $nodeTwo = neighbour$;
18                 else if $dist[nodeOne] > dist[node]$ then
19                     $nodeOne = node$;
20                     $nodeTwo = neighbour$;
21                 Break;
22             else
23                 $Q.enqueue(neighbour)$;
24 $path = null$;
25 if $finalEdge == null$ then
26     $path$ = "no loops for p";
27 else
28     $path.append(Adj[p, nodeOne])$;
29     $path.append(Adj[nodeOne, nodeTwo])$;
30     $path.append(Adj[nodeTwo, p])$;
Output: $path$
```

Although the algorithm might seem slightly confusing upon reading the code, it's actually quite simple, it starts by running the BFS and gettings both the BFSTree and distance array, then it separates the Tree into subTrees using an aux method .subTrees() that just gets the trees that start with the children of the node in question.
After that, it creates two nodes that will be used to establish the connection between the two subtrees. After that, it begins it's looping via the subgroups, now in the worst case scenario there will be N subgroups, but since if thats true the inside of this loop will be O(1), then

the final runtime will be O(n), however, this is not the worst case scenario for the entire algorithm.
In the worst scenario, this first loop will run a constant time, with the inner loops running at O(m+n), since they follow the same basic logic of the loops inside BFS.
Inside these loops, we simply are looking for the highest order non-tree edge, which we store by storing the two nodes in said edge. We get these nodes by searching each of the tree for the first non-tree edge that is connected to one of the tree's children, since this will either be the highest in the entire BFSTree, or any others in its subtree will be smaller, which is why we break after finding the first in each subtree.
We also only change our global variables for the nodes if the one that we just found is higher order than the ones we have stored, otherwise nothing changes.
In the end all we do is check if there are any cycles, and if not return "no loops for p", and otherwise we append the Adjacency list from p to the first node, and then the edge from nodeOne to nodeTwo, and finally the adjacency list from nodeTwo to p again.
As discussed above, the worst case scenario for this algorithm it'll run O(x*(m+n)) where x is a constant that can be discarded so O(m+n), and since BFS is also of the same order, we end up with O(2(m+n), or just O(m+n).