CS180 HW3

Xilai Zhang

TOTAL POINTS

100 / 100

QUESTION 1

1 Problem 1 25 / 25

✓ **- 0 pts Correct**

- 5 pts Preprocessing not correct

- 5 pts failed to move two pointers together

- 10 pts error in algorithm; or algorithm runtime not correct

- 20 pts wrong answer but showed efforts

- 25 pts no answer

QUESTION 2

2 Problem 2 25 / 25

✓ **- 0 pts Correct**

- 5 pts did not update queue during the loop

- 10 pts failed to reduce the original problem

- 20 pts wrong answer but showed efforts

- 25 pts wrong answer or no answer

QUESTION 3

3 Problem 3 25 / 25

✓ **- 0 pts Correct**

- 3 pts Complexity exceeds O(E)

- 4 pts Fail to justify answer or need more explanation

- 4 pts Fail to show the algorithm step-by-tep

QUESTION 4

4 Problem 4 25 / 25

✓ **- 0 pts Correct**

- 4 pts fail to justify answer

- 4 pts fail to show algorithm step-by-step

1. Since the spec says we do preprocessing in O(|E|) and requires lowest common ancestor to be find in O(h) time, the only way that fulfills both requirements could be Euler tour of the tree. And thus the algorithm follows:

Create an array of structs A, each struct can store two numbers, the depth and vertex.
Create a stack of vertices B. //so that we can later perform DFS on it.
Create an array of vertices C to check whether each vertex has been visited.
//preprocessing
Initialize depth to be 0.
Push root to B.
While B is not empty:
Look at the last element of B:
    Mark this vertex x in C as visited.
    Append this vertex x to A. Append the current depth to A.
    If this vertex x has outgoing edges:
        If one of the endpoint y of outgoing edges has not been visited in C:
            Increase depth by 1.
            Push the endpoint vertex y into B.
        Else:
            Pop this vertex from top of B.
            Decrease depth by 1.
    Else if the vertex has no outgoing edges:
        Pop this vertex from top of B.
        Decrease depth by 1.
Create a map D of vertex value and the index of its first appearance in A.
Initialize i to be 0.
Iterate through A:
    If the vertex is marked visited in C:
        Store i to be the map of C in D. Mark the vertex unvisited in C.
    i=i+1
//compute lowest common ancestor
Given the pair u,v, we first index into map D and find the index of their first appearance in A as s,t. Assuem s<t, if not, swap s and t.
Initialize struct LCA to have depth INT_MAX (an extremely large number) and vertex null.
Let k=s.
While k<=t:
    If the depth of A[k] < depth of LCA:
        LCA= depth of A[k]
        Vertex of LCA= vertex of A[k]
    K=k+1
//return the vertex of LCA.

Correctness:

If the Euler tour is constructed with DFS, the lowest common ancestor is between the first appearances of the two nodes in the Euler tour of the tree. So if we maintain the value of depth as we construct the Euler tour, we will be able to return the vertex with lowest depth in certain range of the Euler tour, and thus be able to return lowest common ancestor.

Complexity:
In the preprocessing part, stack A,B,C are built by DFS traversal, which is O(V+E) time. Map D is built by iterate through the 2V-1 element of A, and is also O(E) time if we consider O(V) to be O(E). The computation part is O(h) because we just iterate through the edges in between and return the one with lowest depth.

1 Problem 1 **25 / 25**

✓ **- 0 pts Correct**

- 5 pts Preprocessing not correct

- 5 pts failed to move two pointers together

- 10 pts error in algorithm; or algorithm runtime not correct

- 20 pts wrong answer but showed efforts

- 25 pts no answer

2. This problem can be solved by imitating the topological sort in textbook.

//preprocessing
Initialize an empty map A of all cities and the number of each of its incoming flights.
Initialize an empty stack of cities B.
For each city, record the number of incoming flights and store it in A. If a city does not have incoming flights, delete it from A and append the city to B.
While B is not empty:
    Pop the city C on top of B, and find all of its outgoing flights.
    For each outgoing flights that connect to another city D in A:
        Decrease the value of incoming flight of D in A by 1.
        If the value of D in A becomes 0, remove D from A and append D to B.
Return all cities in A.

Correctness:
The correctness of this method is the same as topological sort in the textbook. We keep removing cities that do not have incoming flights, and remove cities affected by these cities. Eventually we will have a group of cities that have at least one incoming flight from within the group.

Complexity:
Creating A requires O(V+E) since we have to iterate the graph. For each city to be removed, it takes constant time, so in the worst case it takes O(V) time. So in total it will take O(V+E) time.

2 Problem 2 25 / 25

✓ **- 0 pts** **Correct**

- 5 pts did not update queue during the loop

- 10 pts failed to reduce the original problem

- 20 pts wrong answer but showed efforts

- 25 pts wrong answer or no answer

3. We can combine techniques in question 1 and 2 to solve this question, and still use DFS to detect cycle.

//Preprocessing
Initialize a stack of vertices A.
For each vertex, if it does not have incoming edges, append it to A.
Initialize a map C of vertex value, and whether it has been visited.
Initialize an empty stack B of vertices.
While A is not empty:
    Pop a vertex x from A.
    Push x to B.
    While B is not empty:
        Look at the vertex y at top of B. Mark y in C as visited.
        If y goes to any vertex z in C that has been visited:
            Break. Return that there is a cycle.
        If y goes to a vertex z in C that has not been visited:
            Push z to B.
        Else:
            Pop y off B. delete y from C.
Return there is no cycle.

Correctness:
Since we traverse the graph in DFS order, if a cycle exists, DFS will take us back to the visited node. Thus if we have visited a node twice, we know there is a cycle.

Complexity:
We first iterate through the vertices to find all vertices with no incoming edges. This will take O(V+E) complexity. Stack C takes O(V) complexity. Finally, the main part DFS search will take O(V+E) complexity. So in total O(E) complexity if O(V)=O(E).

3 Problem 3 25 / 25

✓ **- 0 pts** **Correct**

- 3 pts Complexity exceeds O(E)

- 4 pts Fail to justify answer or need more explanation

- 4 pts Fail to show the algorithm step-by-tep

4. We can use a similar technique to question 1 to solve this problem.

//Preprocessing
Create a super source node C.
Iterate through vertices, for each vertex x that do not have incoming edge, add a directed edge from C to x.
Initialize an empty stack of vertices B.
Initialize a map D of vertices, whether they have been visited, and their topological order.
Initialize currentOrder to be N.
Push C to B.
While B is not empty:
    Look at the vertex y at top of B. mark y as visited in D.
    If y connects to a vertex z that has not been visited in D:
        Push z to B.
    Else:
        Give y in D topological order of currentOrder.
        currentOrder=currentOrder-1
        pop y off B.
return the vertices in D and their topological order

Correctness:
We created a super source node that connects to all previous nodes that do not have incoming edge. Then we can perform DFS on the super source node to assign topological order for all vertices.

Complexity:
In the worst case, if all vertices are disjoint, we need O(V) time to connect them to super source node. Then the DFS we perform on the graph will cost O(V+E) complexity. So in total there will be O(E) complexity if we assume O(V)=O(E).

4 Problem 4 25 / 25

✓ **- 0 pts** **Correct**

- 4 pts fail to justify answer

- 4 pts fail to show algorithm step-by-step