HW2

Xinhao Luo

Tuesday 12^{th} May, 2020

1 DPV Problem 3.9

Algorithm

Step 1 Loop through each node, and store length of their list into degree[n]

Step 2 Loop through each of node, and sum up each degree[n] in the adjacent list and store it in twodegree[n]

Runtime

Step 1 This step, get the length of the list should be O(1). The for loop should be O(n), assume n nodes, so the time complexity of this step should be O(n)

Step 2 In this step, getting the sum of degree of one node should be O(e), assume e is the number of edge one node have. However, the total number of this loop in total will not exceed 2t, assume t is the total number of edge the graph have, so the time complexity of this step should be O(2t)

conclusion The time complexity of this algorithm is O(n + t), which can be solved in linear time.

2 DPV Problem 3.15 (all parts)

a) Suppose that each intersection as node in graph, and road as edge. If we can prove that the graph generated based on this reflection is strongly connect, then the major is right.

Method

Step 1 Choose a random node s in the graph

Step 2 Run a BFS start from node s, and if any node is not reachable, return false

Step 3 Reverse the directions of all edge in the graph

Step 4 Run BFS again from node s, and if any node is not reachable, return false, otherwise true

Runtime

Step 1 O(1) for choosing a node

Step 2 Time complexity for BFS is O(e + n), assume e is the total number of edges and n is the total number of vertices the graph have, a linear time

Step 3 O(e), assume e is the number of edge of the graph

Step 4 Time complexity for BFS is O(e + n) again

Conclusion The overall time complexity is O(e + 2n), a linear time, so this problem can be solved in linear time

b) Same as part a), formulate intersection as node and road as edge. We now have a given vertex s with an out degree.

Algorithm

Step 1 Start from vertex s, do an BFS, and keep an record of node[] that have visited during searching

Step 2 reverse the direction of all edges in the graph

Step 3 Do BFS again, and remove corresponded node in node[] when searching. If there is node that does not include in the list, return false. If all matched, return true

Runtime

Step 1 Time complexity for BFS is O(e + n), assume e is the total number of edges and n is the total number of vertices the graph have, a linear time

Step 2 Reverse each edge in the graph should be O(e)

Step 3	Time complexity for BFS is $O(e + n)$, and removing specific node from a list can be $O(1)$ if using e.g. hashset
Conclusion	The overall time complexity will be $O(e + n)$, a linear time

3 DPV Problem 3.22

Algorithm

Step 1 Pick a random node s from the graph

Step 2 Start DFS and keep track of the pre and post value of each node

Step 3 Find the node l with the largest post value

Step 4 start doing DFS from node l, mark every node alone the path

Step 5 check if all node is marked, true if all marked, false otherwise.

Runtime

Step 1 O(1)

Step 2 O(n + e) for running DFS and mark node, assume n is the number of node in the graph and e for the number of edges

Step 3 O(n) for finding node with largest post value in the graph

Step 4 O(n + e) again for DFS

Step 5 O(n) for checking all node's mark

Conclusion O(n + e), a linear time solution

4 DPV Problem 3.23

Algorithm

Step 1 Set up an array of path[], initialize with n of zeros, assume n is the number of node in the graph, and set path[t] = 1

Step 2 Start DFS from node s, and each time crossing an edge connected node u to v, set path[v] += path[u]

Step 3 Return path[s]

Runtime

Step 1 O(n) for initialize the array

Step 2 O(n + e) for DFS, assume n is the number of node in the graph and e is the number of edge

Step 3 O(1) for accessing value in array

Conclusion The overal time complexity is O(n + e), a linear time.

5 DPV Problem 3.24

Algorithm

Step 1 Create a array order[] with size n, assume n is the number of node in the graph, and pick a random node s

Step 2 from node s, find the topological order of the graph, and store in order[]

Step 3 for each node in order[], find if current node has connection over the next node in order[]. If not, return false, otherwise true.

Runtime

Step 1 O(n) for creating array and O(1) for picking a random node

Step 2 O(n + e) for building topological order, assume e is the number of edge of the graph

Step 3 O(n) for walking through the order[] and check order

Conclusion O(n + e) is a linear time solution