Assignment – 1: Graph Traversal

Submitted by – Farhan Ishmam, 180041120, CSE – A.

Date – 27-SEP-20

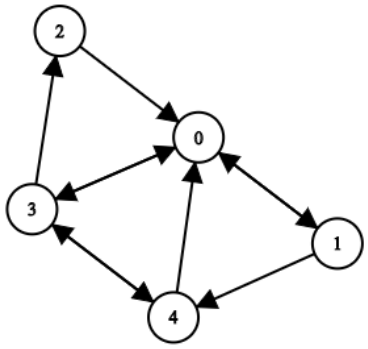
Problem – 1

None

Problem – 2(a)

I looked for websites to draw graphs and the first one was - https://csacademy.com/app/graph_editor/

To draw the graph I had to provide the pair of points, for example (3, 0) for the connection between node 3 and 0. I got my pair of points from the adjacency list. I can also draw the graph directly from the adjacency list but couldn't find a website which would easily let me put nodes and connect them.



Problem – 2(b)

A -> B

B -> C -> D

C -> E -> F

D -> E -> F

E

F -> D -> E

Problem – 2(c)

BFS: A, B, C, D, E, F

DFS: A, B, C, E, F, D

Problem – 2(d)

DAG has no loops or cycles. D – F forms a cycle. So by removing either one of those edges we can form a DAG. Apart from D – F there is no other loop. Similarly, if we perform DFS in lexicographical order we can find one back edge (D – F) and for reverse lexicographical order we can find another back edge (F – D). So these two will be all such possible edges.

By removing D – F: E, D, F, C, B, A

By removing F – D: E, F, C, D, B, A

Problem – 3

The algorithm in short is that we find the node farthest from our root node. Then taking the farthest node as our source node, we traverse the graph again and find the farthest distance. That distance is the diameter of the tree. The implementation steps are given below:

i) Let's add an extra attribute called height to each of our nodes. The height will start at 1 for our source node and will be updated to *(height (parent)) + 1* for the children nodes. Instead of adding height as an attribute, we can use an array to store the heights for each node.

ii) We will have a variable max initialized at 0. If *height (node)>max* then max = height. We will also have a node (or preferably node pointer) variable which will contain the max node. (preferably pointer to the max node)

iii) We will traverse the whole graph using either BFS or DFS from the root of the tree. Both traversal algorithms have O (V + E) time complexity.

iv) After traversing the whole graph we have found the node farthest from the root node and have stored it (or the pointer to it) in the max_node variable. Then taking that as our source node, we traverse the whole graph again using either BFS or DFS.

v) Just like the previous time, we keep another max variable which will store max height and will be updated similarly.

vi) After traversing the whole graph, we will return this max_height variable which will also be the diameter of our graph.

Problem – 4(a)

We construct a graph from the given pairs and then assign party numbers to the pairs. Edges mean opposite parties, so we assign opposite party values. If we find that an assigned node can have two different party numbers then we return false since we need more than 2 parties to invite that person. The implementation steps are given below:

i) We construct the graph from the given pairs. A pair of vertices denote an edge between those two nodes. The pairs can also be transformed into adjacency list or adjacency matrix to be accessed as a graph and it depends on our implementation and problem restrictions.

ii) We add an extra attribute called party to each of our nodes. The party value will be of Boolean data type i.e. either 0 or 1.

iii) Now, we traverse the graph using either of the graph traversing algorithms. Let's use BFS for this case. We start with our source node and assign the party value to 0. Then we assign the opposite party value to the neighboring node (in this case 1). The opposite party values are assigned only if the nodes are not visited.

iv) If the node is visited then we check if the party value we are assigning is same as the party value it has. If the value is same then we continue. Else, we return false.

v) The whole procedure is followed until the whole graph is traversed. If we are successfully able to traverse the whole graph then we return true.

Problem – 4(b)

We construct the graph and while traversing it, we make the visited of the adjacent nodes as TRUE. A count variable is introduced to count the number of unvisited nodes. When we find an unvisited node (i.e visited = FALSE) then we increase the count by 1. After visiting all the nodes, we take different source nodes and return minimum count. The implementation steps are given below:

i) Just like our previous problem we are constructing the graph from the pairs or converting the pairs to adjacency list or adjacency matrix depending on our implementation and problem restrictions.

ii) We keep a count variable initialized at 1 (counting the source node). We start with any node as the source node and use either BFS or DFS for traversing the graph. Let's use BFS for this case.

iii) The visited of the source node is changed to TRUE along with the visited of the neighboring nodes of the source node. Then when we go to the neighboring node and find an unvisited node then we increase the count by 1 and perform the same procedure for the unvisited node, just like we did with the source node (i.e make the visited of the node and its adjacent nodes TRUE)

iv) The steps are to be performed till all nodes are visited.

v) By doing the previous 4 steps, we found one way of traversing the graph and single count value. We repeat these 4 steps by taking each and every node of the graph as a source node. For each source node, we get a different count value and we return the minimum count value we get as our output.