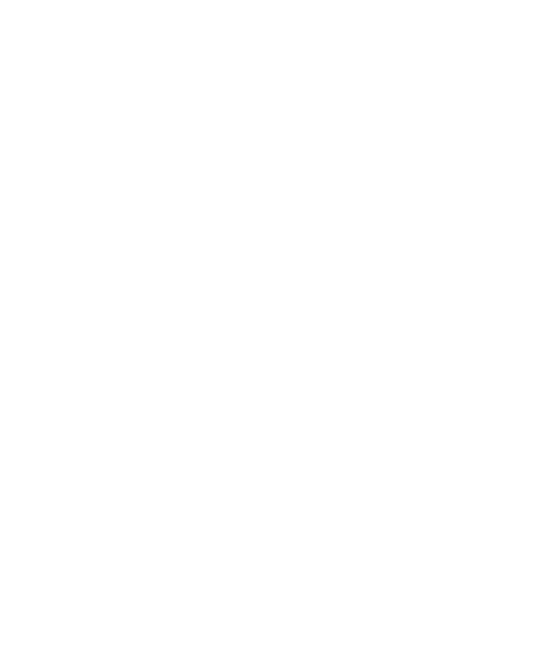
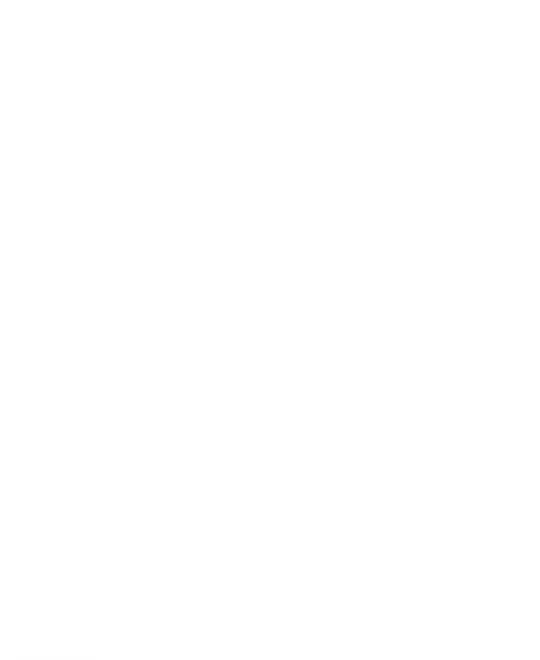
**Problem 1: Collaborators**

**Problem 2: Welcome to the World of Graphs**

1. Representation to Graph



1. Graph to Representation



1. Traversal
2. BFS

* First, we visit A. We add all of the children of A, which is just B here, to a queue. From this point forward, we will visit vertices from the queue one by one, and for all the children we find for that vertex that are not already in the queue, we add them to the queue in lexicographical order.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Queue | B |  |  |  |  |  |
| Visited | A |  |  |  |  |  |
| Parent | ∅ |  |  |  |  |  |

* Next, we visit B. The two children of B, C and D, are added to the queue.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Queue | C | D |  |  |  |  |
| Visited | A | B |  |  |  |  |
| Parent | ∅ | A |  |  |  |  |

* Next, we visit C. The two children of C, E and F, are added to the queue.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Queue | D | E | F |  |  |  |
| Visited | A | B | C |  |  |  |
| Parent | ∅ | A | B |  |  |  |

* Next, we visit D. D has two outgoing edges, to E and F. However, both of these are already in the queue. Nothing will be added to the queue.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Queue | E | F |  |  |  |  |
| Visited | A | B | C | D |  |  |
| Parent | ∅ | A | B | B |  |  |

* Next, we visit E. E has no children, and nothing is added to the queue.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Queue | F |  |  |  |  |  |
| Visited | A | B | C | D | E |  |
| Parent | ∅ | A | B | B | C |  |

* Finally, we visit F. F has one outgoing edge to E and another to its parent, D. Both D and E were previously in the queue and have already been visited. As such, nothing new is added to the queue. The BFS traversal is complete.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Queue |  |  |  |  |  |  |
| Visited | A | B | C | D | E | F |
| Parent | ∅ | A | B | B | C | C |

1. DFS

* For DFS traversal, we add vertices to a stack as we traverse them. First, we visit A and add it to the stack.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Stack | A |  |  |  |  |  |
| Visited | A |  |  |  |  |  |
| Parent | ∅ |  |  |  |  |  |

* Next, we visit B and add it to the stack.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Stack | A | B |  |  |  |  |
| Visited | A | B |  |  |  |  |
| Parent | ∅ | A |  |  |  |  |

* B has two children, C and D. We could visit either child first, but since we are following lexicographical ordering, we will visit C first. We add C to the stack.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Stack | A | B | C |  |  |  |
| Visited | A | B | C |  |  |  |
| Parent | ∅ | A | B |  |  |  |

* C has two children, E and F. Again, following lexicographical ordering, we will visit E first and add E to the stack.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Stack | A | B | C | E |  |  |
| Visited | A | B | C | E |  |  |
| Parent | ∅ | A | B | C |  |  |

* Since E has no children, we shall now pop elements from the stack, thus ‘going backwards’ up the tree, until we find a vertex with unvisited children. In this example, we only need to pop E, since its immediate parent, C, has an unvisited child.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Stack | A | B | C |  |  |  |
| Visited | A | B | C | E |  |  |
| Parent | ∅ | A | B | C |  |  |

* We now visit the unvisited child of C, F. We add F to the stack.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Stack | A | B | C | F |  |  |
| Visited | A | B | C | E | F |  |
| Parent | ∅ | A | B | C | C |  |

* F has two children, D and E. Since D is unvisited, we will now visit D and add it to the stack.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Stack | A | B | C | F | D |  |
| Visited | A | B | C | E | F | D |
| Parent | ∅ | A | B | C | C | F |

* D has two outgoing connections, to E and F. The connection from D to E is a cross-edge and the connection from D to F is a back-edge. Both E and F have already been visited, so we do not need to do anything here.
* We now begin popping elements again, since we have nothing left to do. Since D has been completely traversed, D gets popped.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Stack | A | B | C | F |  |  |
| Visited | A | B | C | E | F | D |
| Parent | ∅ | A | B | C | C | F |

* Similarly, F and C have also been completely traversed and also get popped one after another.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Stack | A | B |  |  |  |  |
| Visited | A | B | C | E | F | D |
| Parent | ∅ | A | B | C | C | F |

* Once we reach B, we find an outgoing connection from B to D. However, since D has already been traversed, this connection is a cross-edge. As such, we do not need to do anything. B has been completely traversed, and can thus be popped.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Stack | A |  |  |  |  |  |
| Visited | A | B | C | E | F | D |
| Parent | ∅ | A | B | C | C | F |

* Finally, A has been completely traversed, and can be popped.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Stack |  |  |  |  |  |  |
| Visited | A | B | C | E | F | D |
| Parent | ∅ | A | B | C | C | F |

* Since all the vertices of the graph have been traversed, our DFS traversal is complete.

1. DAG
2. Removing the edge from F to D.

Topological Sort Order: A B D C F E

1. Removing the edge from D to F.

Topological Sort Order: A B C F D E

**Problem 3: Diameter of a Tree**

Depth First Search Algorithm

The algorithm to find the maximum diameter of a tree makes use of the depth first search (DFS) algorithm, so we need to discuss that first.

In DFS, we pick any node in the graph as the source node, S, to start from. Thus, at the beginning of the program, the current node, curNode, is S. We will also use a stack, which we will simply call stack, in order to keep track of the path we followed to go down a branch so we can come back up, the length of the current branch curLen and the maximum branch length we find, maxLen. We also need to keep a pointer to the node which is furthest from our source, N. Its use will be explained shortly.

For every child of curNode, we will recursively call the function, incrementing curLen every time, thus going down each of the branches. If curLen becomes larger than maxLen, we change the value of maxLen and the pointer N accordingly.

Note that this algorithm varies from a normal DFS algorithm, which also keeps track of which nodes have been visited already using an array, so as to avoid repeatedly visiting the same nodes. However, since we are working with a tree, there is no possibility of visiting the same node twice.

Diameter of a Tree Algorithm

Performing the above DFS algorithm, we will find the distance from the source node we chose, S, to the node furthest from it, N. However, the maximum possible distance between any two nodes in the tree will be the result of perform DFS on N. Consider two branches of S, one that gives a length of to the node N and another which gives a length of to the node M. Thus, the maximum distance, from S to N, is . However, if we start from N, in order to reach M, we would need to go up one branch back to S and then down the other branch. Thus, the distance would be . This is the path that would be followed if we performed DFS on N.

Pseudocode

curLen = -1  
maxLen = 0  
maxNode = NULL  
  
DFS (curNode)  
 curLen++  
 if (curLen > maxLen)  
 maxLen = curLen  
 maxNode = curNode  
 for every child of curNode  
 DFS(child)  
 curLen--  
  
DFS(S)  
DFS(N)

PSEUDOCODE

Algorithm Proof

Starting at the source node , the distance between and one of its children is . We are recursively calling the same algorithm for every node we visit and incrementing the value of for every step downwards it takes. When coming back up, we are decrementing accordingly. We are changing the maximum value, , whenever . As such, for any node, , in the tree that is at a distance from , . If we visit a child of , then and if , . As such, at any node in the tree, we are always able to keep track of the maximum distance from and the node that is at this distance.

For the second part of the algorithm, the same process is simply repeated for a different node. This means the same logic and validity will hold.

Time Complexity

In the DFS algorithm, we are visiting a total of vertices. Since we are following branches, we will also traverse a total of edges, two times per edge, once going down and once coming back up. We are also performing the entire algorithm two times for two nodes, which doubles the count of vertices and edge traversals. This brings the total count up to . Thus, the time complexity of the proposed algorithm is .

References

<https://codeforces.com/blog/entry/5787>

**Problem 4: Properties of a Graph**

1. Pickle Mick

Algorithm

In this algorithm, we will consider every neighbour of Mick to be a vertex. Neighbours who have conflicts will have an undirected edge between them.

We need to divide the neighbours into two groups, and . We can do this using a simple integer variable. By default, we set this value to for everyone.

Now we begin a breadth first search (BFS) traversal from any random node in the graph. We will give the source node the value . We look at every node that is connected to our source node and add it to a queue, giving each of them a value of . We then recursively call the BFS algorithm on each of the nodes in the queue and repeat the process. Since each node in the queue has a value of , every node that is connected to those should have a value of . Thus, we are traversing level-by-level, giving each level an alternating value. We can keep track of which value to give using a simple Boolean variable.

If we find a node that has the same value as the node we are currently on, then we have a problem. This means that both of these nodes were previously designated to be in the same group, due to having problems with a common node. Since we find an edge between then, it means they cannot be in the same group. As such, it is not possible to divide all the nodes into just two groups.

Once the BFS traversal is complete, we need to check for any nodes that still have the initial value. Any such nodes were untraversed and we need to perform BFS on them as well.

Time Complexity

This algorithm is a simple BFS algorithm with nothing added that would change the time complexity. We are visiting every vertex, for a total of vertices, and traversing every edge, at most twice, giving us a total of edges. Notice how we need to traverse non-tree edges as well here, since we need to check the value of the node at the other end of that edge. Overall, the time complexity is , which is linear.

1. Mucced

Algorithm

This problem can be solved using a normal depth first search (DFS) traversal. Each user is considered a node, and users that are friends will have an undirected edge between them.

Say we assign a value of to every user. We can begin our traversal at a particular user, , giving a value of . Every other user encountered while performing the traversal on is mutual with , and will thus be mutual with Zark if he becomes friends with . We give each of these users a value of .

Next, we look for any users that still have the value , meaning they could not be reached via . We need to pick one of these users and run the algorithm on them again, thus making Zark mutual to all of their mutuals as well. We keep repeating this until all users have been reached.

The number of users Zark needs to be friends with is the number of users with the value at the end of the algorithm.

Time Complexity

This algorithm uses simple DFS. We visit vertices, and traverse a total of edges. As such, the time complexity comes up to .