Rust & Murderer

by pranav9413

Problem Submissions Leaderboard Discussions Editorial

Detective Rust is investigating a homicide and he wants to chase down the murderer. The murderer knows he would definitely get caught if he takes the main roads for fleeing, so he uses the village roads (or side lanes) for running away from the crime scene.

Rust knows that the murderer will take village roads and he wants to chase him down. He is observing the city map, but it doesn't show the village roads (or side lanes) on it and shows only the main roads.

The map of the city is a graph consisting N nodes (labeled 1 to N) where a specific given node S represents the current position of Rust and the rest of the nodes denote other places in the city, and an edge between two nodes is a main road between two places in the city. It can be suitably assumed that *an edge that doesn't exist/isn't shown on the map is a village road (side lane)*. That means, there is a village road between two nodes a and b iff(if and only if) there is no city road between them.

Rust wants to calculate the shortest distance from his position (Node S) to all the other places in the city if he travels using the village roads (side lanes).

Note: The graph/map of the city is ensured to be a sparse graph.

Input Format

The first line contains T, denoting the number of test cases. T testcases follow.
First line of each test case has two integers N, denoting the number of cities in the map and M, denoting the number of roads in the map.
The next M lines each consist of two space-separated integers x and y denoting a main road between city x and city y. The last line has an integer S, denoting the current position of Rust.

Constraints

- $1 \leq T \leq 10$
- $2 \leq N \leq 2 \times 10^5$
- $0 \leq M \leq 120000$
- $1 \leq x, y, S \leq N$

Note

- *No city will have a road to itself.*
- *There will not be multiple roads between any pair of cities i.e. there is at most one undirected road between them.*
- *Graph is guaranteed to be sparse.*

Output Format

For each of T test cases, print a single line consisting of N-1 space separated integers, denoting the shortest distances of the remaining N-1 places from Rust's position using the village roads/side lanes in ascending order based on vertex number.

It is guranteed that there will be a path between any pair of cities using the side lanes

Sample Input 0

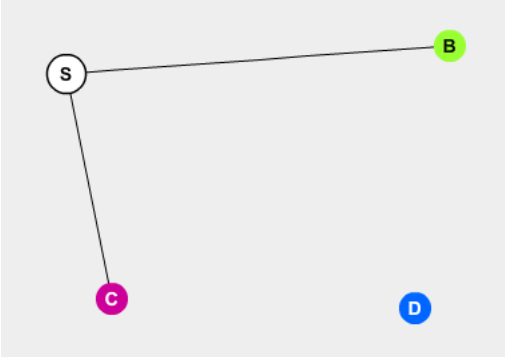
```
1
4 2
1 2
1 3
1
```

Sample Output 0

```
2 2 1
```

Explanation 0

The graph given in the test case is shown as :



S denotes the node 1 in the test case and B,C and D denote 2,3 and 4. Since S is the starting node and the shortest distances using the village roads/side lanes from it are 2, 2, 1 to th nodes B,C and D (2,3 and 4) respectively.

The distance 2 to node B follows the path S->D->B , for node C , the path S->D->C and for node D , the path is S->D , hence justifying the shortest distances.

Submissions: 1603
Max Score: 70
Difficulty: Medium

Rate This Challenge:

More

Current Buffer (saved locally, editable) Java 8

```
1  import java.io.*;
2  import java.util.*;
3
4     class BinaryMinHeap{
5
6          private List<Node> allNodes = new ArrayList<>();
7          private Map<Integer,Integer> nodePosition = new HashMap<>();
8
9          public BinaryMinHeap(){
10
11         }
12
13
14         public boolean containsData(int key){
15             return nodePosition.containsKey(key);
16         }
17
18         public void add(int weight, int key){
19
20             Node node = new Node();
21
22             node.weight = weight;
```

```
            node.key = key;
            allNodes.add(node);

            int nodeSize = allNodes.size();
            int currentPosition = nodeSize - 1;

            int parentIndex = (currentPosition - 1) / 2;
            nodePosition.put(node.key, currentPosition);

            while(parentIndex >= 0){

                Node parentNode = allNodes.get(parentIndex);
                Node currentNode = allNodes.get(currentPosition);

                if(parentNode.weight > currentNode.weight){
                    swap(parentNode,currentNode);
                    updatePositionMap(parentNode.key,currentNode.key,parentIndex,currentPosition);
                    currentPosition = parentIndex;
                    parentIndex = (parentIndex-1)/2;                }
                else{
                    break;
                }

            }
        }

        public int min(){
            return allNodes.get(0).key;
        }

        public boolean empty(){
            return allNodes.size() == 0;
        }

        public void decrease(int data, int newWeight){
                Integer position = nodePosition.get(data);
                allNodes.get(position).weight = newWeight;
                int parent = (position -1 )/2;
                while(parent >= 0){
                    if(allNodes.get(parent).weight > allNodes.get(position).weight){
                        swap(allNodes.get(parent), allNodes.get(position));
                        updatePositionMap(allNodes.get(parent).key,allNodes.get(position).key,parent,position);
                        position = parent;
                        parent = (parent-1)/2;
                    }else{
                        break;
                    }
                }
            }

        public Integer getWeight(int key) {
            Integer position = nodePosition.get(key);
            if( position == null ) {
                return null;
            } else {
                return allNodes.get(position).weight;
            }
        }


        public Node extractMinNode() {
            int size = allNodes.size() -1;
            Node minNode = new Node();
            minNode.key = allNodes.get(0).key;
            minNode.weight = allNodes.get(0).weight;

            int lastNodeWeight = allNodes.get(size).weight;
            allNodes.get(0).weight = lastNodeWeight;
            allNodes.get(0).key = allNodes.get(size).key;
            nodePosition.remove(minNode.key);
            nodePosition.remove(allNodes.get(0));
            nodePosition.put(allNodes.get(0).key, 0);
            allNodes.remove(size);
```

```
96
97              int currentIndex = 0;
98              size--;
99              while(true){
100                 int left = 2*currentIndex + 1;
101                 int right = 2*currentIndex + 2;
102                 if(left > size){
103                     break;
104                 }
105                 if(right > size){
106                     right = left;
107                 }
108                 int smallerIndex = allNodes.get(left).weight <= allNodes.get(right).weight ? left : right;
109                 if(allNodes.get(currentIndex).weight > allNodes.get(smallerIndex).weight){
110                     swap(allNodes.get(currentIndex), allNodes.get(smallerIndex));
111
    updatePositionMap(allNodes.get(currentIndex).key,allNodes.get(smallerIndex).key,currentIndex,smallerIndex);
112                     currentIndex = smallerIndex;
113                 }else{
114                     break;
115                 }
116             }
117             return minNode;
118         }
119
120
121         public int extractMin(){
122             Node node = extractMinNode();
123             return node.key;
124         }
125
126
127         private void swap(Node node1,Node node2){
128
129             int weight = node1.weight;
130             int data = node1.key;
131
132             node1.key = node2.key;
133             node1.weight = node2.weight;
134
135             node2.key = data;
136             node2.weight = weight;
137     }
138
139         private void updatePositionMap(int data1, int data2, int pos1, int pos2){
140             nodePosition.remove(data1);
141             nodePosition.remove(data2);
142             nodePosition.put(data1, pos1);
143             nodePosition.put(data2, pos2);
144         }
145     }
146
147
148     class Node{
149
150         int weight;
151         int key;
152
153         public Node(){
154
155         }
156
157     }
158
159
160
161
162 class Graph{
163
164     private int V;
165     private Set<Integer> adj[];
166
167     public Graph(int V){
```

```
        this.V = V;
        adj = new HashSet[V];

        for(int i = 0 ; i < V ; i++){
            adj[i] = new HashSet<Integer>();
        }

    }

    public void addEdge(int srcVertex, int destVertex){
        adj[srcVertex].add(destVertex);
        adj[destVertex].add(srcVertex);
    }

    public Map<Integer,Integer> runDijkstra(int source, boolean isAllConnected){

        Map<Integer,Integer> output = new HashMap<Integer,Integer>();

        BinaryMinHeap minHeap = new BinaryMinHeap();

        for(int i = 0 ; i < V ; i++){

            if(i != source){
                minHeap.add(20000000,i);
                output.put(i,20000000);
            }
            else{
                minHeap.add(0,i);
                output.put(i,0);
            }
        }


        if(!isAllConnected){

            Set<Integer> set = adj[source];

            for(int i = 0 ; i < V ; i++){

                if(set.contains(i)){
                    output.put(i,2);;
                }
                else{
                    output.put(i,1);
                }

            }

            return output;
        }


        while(!minHeap.empty()){

            Node node = minHeap.extractMinNode();

            Set<Integer> set = adj[node.key];

            for(int i = 0 ; i < V ; i++){

                if(node.key == i || set.contains(i) || !minHeap.containsData(i)){
                    continue;
                }

                if(minHeap.getWeight(i) > node.weight + 1){
                    minHeap.decrease(i,node.weight + 1);
                    output.put(i,node.weight + 1);
                }

            }


        }
```

```
        return output;
    }
}



public class Solution {

    public static void main(String[] args)  throws IOException{

        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        int t = Integer.parseInt(br.readLine());

        for(int i = 0 ; i < t ; i++){

            String[] numbers = br.readLine().split("\\s");

            int n = Integer.parseInt(numbers[0]);
            int m = Integer.parseInt(numbers[1]);

            Graph graph = new Graph(n);

            Set<Integer> set = new HashSet<Integer>();

            for(int j = 0 ; j < m ; j++){
                numbers = br.readLine().split("\\s");
                graph.addEdge(Integer.parseInt(numbers[0]) - 1, Integer.parseInt(numbers[1]) - 1);
                set.add(Integer.parseInt(numbers[0]) - 1);
                set.add(Integer.parseInt(numbers[1]) - 1);
            }


            int s = Integer.parseInt(br.readLine()) - 1;

            Map<Integer,Integer> output = graph.runDijkstra(s,set.size() == n ? true : false);

            for(int a : output.keySet()){

                if(a != s){
                    System.out.print(output.get(a) + " ");
                }
            }

            System.out.println("");
        }
    }
}
```

Line: 1 Col: 1

Upload Code as File Test against custom input Run Code Submit Code

Copyright © 2017 HackerRank. All Rights Reserved

Join us on IRC at #hackerrank on freenode for hugs or bugs.

Contest Calendar | Interview Prep | Blog | Scoring | Environment | FAQ | About Us | Support | Careers | Terms Of Service | Privacy Policy | Request a Feature