BFS

Send Feedback

Given an undirected and disconnected graph G(V, E), print its BFS traversal.

Note:

1. Here you need to consider that you need to print BFS path starting from vertex 0 only.
2. V is the number of vertices present in graph G and vertices are numbered from 0 to V-1.
3. E is the number of edges present in graph G.
4. Take graph input in the adjacency matrix.
5. Handle for Disconnected Graphs as well

Input Format :

The first line of input contains two integers, that denote the value of V and E.
Each of the following E lines contains space separated two integers, that denote that there exists an edge between vertex a and b.

Output Format :

Print the BFS Traversal, as described in the task.

Constraints :

0 <= V <= 1000
0 <= E <= (V * (V - 1)) / 2
0 <= a <= V - 1
0 <= b <= V - 1
Time Limit: 1 second

Sample Input 1:

```
4 4
0 1
0 3
1 2
2 3
```

```
import java.util.LinkedList;

import java.util.Queue;

import java.io.BufferedReader;

import java.io.InputStreamReader;

import java.io.IOException;

import java.util.Scanner;



public class Solution {

    //A print helper function to solve the print function issues

    public static void printHelper(int edges[][], int sv,boolean visited[]){

        Queue<Integer> q = new LinkedList<>();

        q.add(sv);

        visited[sv]=true;

        while(q.size()!=0){

            int firstelem = q.poll();

            System.out.print(firstelem+" ");

            for(int i=0; i<edges.length; i++){
```

```
            if(edges[firstelem][i]==1 && !visited[i]){
                q.add(i);
                visited[i]=true;
            }
        }
    }
}
// we have to deal with both connected and non connected
public static void print(int edges[][]){
    boolean visited[] = new boolean[edges.length];

    for(int i=0; i< edges.length; i++){
        if(!visited[i]){
            printHelper(edges, i, visited);
        }
    }
}


public static void main(String[] args) throws NumberFormatException, IOException {
    Scanner s = new Scanner(System.in);
    int V = s.nextInt();
    int E = s.nextInt();
    int edges[][] = new int[V][V];

    for(int i =0; i< E; i++){
        int fv = s.nextInt();
        int sv = s.nextInt();
        edges[fv][sv] = 1;
        edges[sv][fv] =1;
    }

    print(edges);

  }



}
```

Has Path?

Send Feedback

Given an undirected graph G(V, E) and two vertices v1 and v2 (as integers), check if there exists any path between them or not. Print true if the path exists and false otherwise.

Note:

1. V is the number of vertices present in graph G and vertices are numbered from 0 to V-1.
2. E is the number of edges present in graph G.

Input Format :

The first line of input contains two integers, that denote the value of V and E.
Each of the following E lines contains two integers, that denote that there exists an edge between vertex 'a' and 'b'.
The following line contain two integers, that denote the value of v1 and v2.

Output Format :

The first and only line of output contains true, if there is a path between v1 and v2 and false otherwise.

Constraints :

0 <= V <= 1000
0 <= E <= 1000
0 <= a <= V - 1
0 <= b <= V - 1
0 <= v1 <= V - 1
0 <= v2 <= V - 1
Time Limit: 1 second

Sample Input 1 :

```
4 4
0 1
0 3
1 2
2 3
1 3
```

Sample Output 1 :

```
true
```

```
import java.util.*;

import java.util.LinkedList;

import java.util.Queue;

import java.io.BufferedReader;

import java.io.InputStreamReader;

import java.io.IOException;

public class Solution {



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

    /* Write Your Code Here
```

```
       * Complete the Rest of the Program
       * You have to take input and print the output yourself
       */
 Scanner sc = new Scanner(System.in);
 int V = sc.nextInt();
 int E = sc.nextInt();
 int edges[][] = new int[V][V];
 for(int i=0;i<E;i++){
  int sv = sc.nextInt();
  int ev = sc.nextInt();
  edges[sv][ev] = 1;
  edges[ev][sv] = 1;
 }
 int V1 = sc.nextInt();
 int V2 = sc.nextInt();
 if(V2>=V){
  System.out.println("false");
  return;
 }
 boolean visited[] =new boolean[V];
 boolean ans = hasPath(edges,V1,V2,visited);
 System.out.println(ans);

   }
public static boolean hasPath(int [][] edges,int V1, int V2,boolean visited[]){
 if(V1>edges.length || V2>edges.length){
  return false;
 }
 if(edges[V1][V2]==1){
  return true;
```

```
        }
        Queue<Integer>q = new LinkedList<>();
        q.add(V1);
        visited[V1]= true;
        while(!q.isEmpty()){
            int n = q.remove();
            for(int i=0;i<edges.length;i++){
                if(edges[n][i]==1 && !visited[i]){
                    q.add(i);
                    visited[i] = true;
                }
            }
        }
        if(visited[V2]==true)
            return true;
            else
                return false;



    }


}
```

Get Path - DFS

Send Feedback

Given an undirected graph G(V, E) and two vertices v1 and v2(as integers), find and print the path from v1 to v2 (if exists). Print nothing if there is no path between v1 and v2.

Find the path using DFS and print the first path that you encountered.

Note:

1. V is the number of vertices present in graph G and vertices are numbered from 0 to V-1.
2. E is the number of edges present in graph G.
3. Print the path in reverse order. That is, print v2 first, then intermediate vertices and v1 at last.
4. Save the input graph in Adjacency Matrix.

Input Format :

The first line of input contains two integers, that denote the value of V and E.
Each of the following E lines contains two integers, that denote that there exists an edge between vertex 'a' and 'b'.
The following line contain two integers, that denote the value of v1 and v2.

Output Format :

Print the path from v1 to v2 in reverse order.

Constraints :

2 <= V <= 1000
1 <= E <= (V * (V - 1)) / 2
0 <= a <= V - 1
0 <= b <= V - 1
0 <= v1 <= 2^31 - 1
0 <= v2 <= 2^31 - 1
Time Limit: 1 second

Sample Input 1 :

```
4 4
0 1
0 3
1 2
2 3
1 3
```

Sample Output 1 :

```
3 0 1
```

```
import java.util.*;

import java.util.ArrayList;

import java.io.BufferedReader;

import java.io.InputStreamReader;

import java.io.IOException;

public class Solution {



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



    /* Write Your Code Here

                * Complete the Rest of the Program

                * You have to take input and print the output yourself

                */

     Scanner sc = new Scanner(System.in);

    int V = sc.nextInt();

    int E = sc.nextInt();

    int edges[][] = new int[V][V];

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

       int sv = sc.nextInt();

       int ev = sc.nextInt();

       edges[sv][ev] = 1;
```

```
        edges[ev][sv] = 1;
      }
      int V1 = sc.nextInt();
      int V2 = sc.nextInt();

      boolean visited[] =new boolean[V];
      ArrayList<Integer> ans = getPathDFS(edges,visited,V1,V2);
      if(ans!=null){
        for(int elem:ans){
      System.out.print(elem+" ");
      }
          }
  }
    public static ArrayList<Integer> getPathDFS(int [][] edges,boolean[] visited,int V1, int V2){
      if(V1==V2){
        ArrayList<Integer> ans  = new ArrayList<>();
        visited[V1] = true;
        ans.add(V1);
        return ans;
      }
      visited[V1] = true;
      for(int i=0;i<edges.length;i++){
        if(edges[V1][i]==1 && !visited[i]){
          ArrayList<Integer> arr = getPathDFS(edges,visited,i,V2);
          if(arr!=null){
            arr.add(V1);
            return arr;
          }
        }
      }
```

```
        return null;

            }

    }
```

Get Path - BFS

Send Feedback

Given an undirected graph G(V, E) and two vertices v1 and v2 (as integers), find and print the path from v1 to v2 (if exists). Print nothing if there is no path between v1 and v2.

Find the path using BFS and print the shortest path available.

Note:

1. V is the number of vertices present in graph G and vertices are numbered from 0 to V-1.
2. E is the number of edges present in graph G.
3. Print the path in reverse order. That is, print v2 first, then intermediate vertices and v1 at last.
4. Save the input graph in Adjacency Matrix.

Input Format :

The first line of input contains two integers, that denote the value of V and E.
Each of the following E lines contains two integers, that denote that there exists an edge between vertex a and b.
The following line contain two integers, that denote the value of v1 and v2.

Output Format :

Print the path from v1 to v2 in reverse order.

Constraints :

2 <= V <= 1000
1 <= E <= (V * (V - 1)) / 2
0 <= a <= V - 1
0 <= b <= V - 1
0 <= v1 <= 2^31 - 1
0 <= v2 <= 2^31 - 1
Time Limit: 1 second

Sample Input 1 :

```
4 4
0 1
0 3
1 2
2 3
1 3
```

Sample Output 1 :

```
3 0 1
```

```
import java.util.*;

import java.util.ArrayList;

import java.io.BufferedReader;

import java.io.InputStreamReader;

import java.io.IOException;

public class Solution {



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

    /* Write Your Code Here

                 * Complete the Rest of the Program
```

```
                        * You have to take input and print the output yourself
                        */
    Scanner sc = new Scanner(System.in);
    int V = sc.nextInt();
    int E = sc.nextInt();
    int edges[][] = new int[V][V];
    for(int i=0;i<E;i++){
        int sv = sc.nextInt();
        int ev = sc.nextInt();
        edges[sv][ev] = 1;
        edges[ev][sv] = 1;
    }
    int V1 = sc.nextInt();
    int V2 = sc.nextInt();
    boolean visited[] = new boolean[V];
    ArrayList<Integer> ans = getPathBFS(edges,visited,V1,V2);
    if(ans!=null){
        for(int elem:ans){
            System.out.print(elem+" ");
        }
    }

       }
public static ArrayList<Integer> getPathBFS(int [][] edges, boolean [] visited,int V1,int V2){
    if(V1==V2){
        ArrayList<Integer> ans = new ArrayList<Integer>();
        ans.add(V1);
        visited[V1] = true;
        return ans;
    }
    Queue<Integer> q = new LinkedList<Integer>();
    HashMap<Integer,Integer> h = new HashMap<>();
    ArrayList<Integer> ans = new ArrayList<>();
    q.add(V1);
    visited[V1] = true;
    while(!q.isEmpty()){
      int first = q.remove();
      for(int i=0;i<edges.length;i++){
        if(edges[first][i]==1 && ! visited[i]){
          visited[i] = true;
          q.add(i);
          h.put(i,first);
          if(i==V2){
            ans.add(i);
            while(!ans.contains(V1)){
              int b = h.get(i);
              ans.add(b);
              i = b;
            }
            return ans;
          }
        }
      }
    }
    return null;
  }
}
```

isConnected?

Send Feedback

Given an undirected graph G(V,E), check if the graph G is connected graph or not.

Note:

1. V is the number of vertices present in graph G and vertices are numbered from 0 to V-1.
2. E is the number of edges present in graph G.

Input Format :

The first line of input contains two integers, that denote the value of V and E.
Each of the following E lines contains two integers, that denote that there exists an edge between vertex a and b.

Output Format :

The first and only line of output contains "true" if the given graph is connected or "false", otherwise.

Constraints :

0 <= V <= 1000
0 <= E <= (V * (V - 1)) / 2
0 <= a <= V - 1
0 <= b <= V - 1
Time Limit: 1 second

Sample Input 1:

```
4 4
0 1
0 3
1 2
2 3
```

Sample Output 1:

```
true
```

```
import java.util.*;

import java.io.BufferedReader;

import java.io.InputStreamReader;

import java.io.IOException;



public class Solution {



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


    /* Write Your Code Here

            * Complete the Rest of the Program

            * You have to take input and print the output yourself

            */

  Scanner scanner = new Scanner(System.in);

            int v = scanner.nextInt();

            int e = scanner.nextInt();

    if(v==0){ //this checks for case when the user enters 0 vertex and 0 edges

      System.out.print("true");
```

```
            return;
        }
                            int[][] edges = new int[v][v];
                            for(int i=0;i<e;i++) {
                                           int sv = scanner.nextInt();
                                           int ev = scanner.nextInt();
                                           edges[sv][ev] = 1;
                                           edges[ev][sv] = 1;
                            }
                            System.out.println(isConnected(edges,0));


}
    public static boolean isConnected(int [][] edges, int sv){
        boolean [] visited  = new boolean[edges.length];
        Queue<Integer> q = new LinkedList<>();
        q.add(sv);
        visited[sv] = true;
        while(!q.isEmpty()){
            int front = q.poll();
            for(int i=0;i<edges.length;i++){
                if(!visited[i] && edges[front][i]==1){
                    q.add(i);
                    visited[i] = true;
                }
            }
        }
        for(boolean b : visited){
            if(!b){
                return false;
            }
        }
        return true;
    }
}
```

Islands

Send Feedback

An island is a small piece of land surrounded by water . A group of islands is said to be connected if we can reach from any given island to any other island in the same group . Given V islands (numbered from 1 to V) and E connections or edges between islands. Can you count the number of connected groups of islands.

Input Format :

The first line of input contains two integers, that denote the value of V and E.
Each of the following E lines contains two integers, that denote that there exists an edge between vertex a and b.

Output Format :

Print the count the number of connected groups of islands

Constraints :

0 <= V <= 1000
0 <= E <= (V * (V-1)) / 2
0 <= a <= V - 1
0 <= b <= V - 1
Time Limit: 1 second

Sample Input 1:

```
5 8
0 1
0 4
1 2
2 0
2 4
3 0
3 2
4 3
```

Sample Output 1:

```
1
```

```
public class Solution {
    public static void helpDFS(int [][] edges,boolean [] visited, int start,int n){
        visited[start] = true;
        for(int i=0;i<n;i++){
            if(edges[start][i]==1 && !visited[i]){
                helpDFS(edges,visited,i,n);
            }
        }


    }
```

```
       public static int numConnected(int[][] edges, int n) {


               /* Your class should be named Solution
                * Don't write main().
                * Don't read input, it is passed as function argument.
                * Return output and don't print it.
                * Taking input and printing output is handled automatically.
    */
    boolean [] visited = new boolean[n];
    int count = 0;
    for(int i=0;i<n;i++){
      if(!visited[i]){
        count++;
        helpDFS(edges,visited,i,n);
      }
    }
    return count;
       }



}
```

Coding Ninjas

Send Feedback

Given a NxM matrix containing Uppercase English Alphabets only. Your task is to tell if there is a path in the given matrix which makes the sentence “CODINGNINJA” .

There is a path from any cell to all its neighbouring cells. For a particular cell, neighbouring cells are those cells that share an edge or a corner with the cell.

Input Format :

The first line of input contains two space separated integers N and M, where N is number of rows and M is the number of columns in the matrix.
Each of the following N lines contain M characters. Please note that characters are not space separated.

Output Format :

Print 1 if there is a path which makes the sentence “CODINGNINJA” else print 0.

Constraints :

1 <= N <= 1000
1 <= M <= 1000

```
Time Limit: 1 second
```

Sample Input 1:

```
2 11
CXDXNXNXNXA
XOXIXGXIXJX
```

Sample Output 1:

```
1
```

```
public class Solution {


        int solve(String[] Graph , int N, int M)
        {
                /* Your class should be named Solution
                 * Don't write main().
                 * Don't read input, it is passed as function argument.
                 * Return output and don't print it.
                 * Taking input and printing output is handled automatically.
    */
    String searchString = "CODINGNINJA";
    boolean [][] visited = new boolean[Graph.length][];
    for(int i=0;i<Graph.length;i++){
      visited[i] = new boolean[Graph[i].length()];
    }
    for(int i=0;i<Graph.length;i++){
    for(int j=0;j<Graph[i].length();j++){
      if(Graph[i].charAt(j)=='C'){
          boolean ans = dfs(Graph,visited,searchString.substring(1),i,j);
         if(ans){
           return 1;
         }


      }
    }
```

```
                }
    return 0;


}
public static boolean dfs(String [] graph,boolean [][] visited, String searchString, int i,int j){
    if(searchString.length()==0){
        visited[i][j] = true;
        return true;
    }
    visited[i][j]= true;
    int [] X = {-1,1,0,0,1,-1,1,-1};
    int [] Y = {0,0,-1,1,1,-1,-1,1};

    for(int k = 0;k<X.length;k++){
        int x = i+X[k];
        int y = j+Y[k];
        if(x>=0 && y>=0 && x<graph.length && y<graph[x].length()
            && graph[x].charAt(y) == searchString.charAt(0) && !visited[x][y] ){
            boolean smallAns = dfs(graph,visited,searchString.substring(1),x,y);
            if(smallAns){
                return smallAns;
            }
        }
        }
        visited[i][j] = false;
        return false;
    }
}
```

Connecting Dots

Send Feedback

Gary has a board of size NxM. Each cell in the board is a coloured dot. There exist only 26 colours denoted by uppercase Latin characters (i.e. A,B,...,Z). Now Gary is getting bored and wants to play a game. The key of this game is to find a cycle that contain dots of same colour. Formally, we call a sequence of dots d1, d2, ..., dk a cycle if and only if it meets the following condition:

1. These k dots are different: if i ≠ j then di is different from dj.
2. k is at least 4.
3. All dots belong to the same colour.
4. For all 1 ≤ i ≤ k - 1: di and di + 1 are adjacent. Also, dk and d1 should also be adjacent. Cells x and y are called adjacent if they share an edge.

Since Gary is colour blind, he wants your help. Your task is to determine if there exists a cycle on the board.

Input Format :

The first line of input contains two space separated integers N and M, where N is number of rows and M is the number of columns of the board.
Each of the following N lines contain M characters. Please note that characters are not space separated. Each character is an uppercase Latin letter.

Output Format :

Print true if there is a cycle in the board, else print false.

Constraints :

2 <= N <= 1000
2 <= M <= 1000
Time Limit: 1 second

Sample Input 1:

```
3 4
AAAA
ABCA
AAAA
```

Sample Output 1:

```
true
```

```
import java.util.*;



public class Solution {



   int solve(String[] board , int n, int m)

           {

                       /* Your class should be named Solution

                        * Don't write main().

                        * Don't read input, it is passed as function argument.

                        * Return output and don't print it.

                        * Taking input and printing output is handled automatically.

       */

     boolean[][] visited = new boolean[n][m];

                       for(int i=0;i<n;i++) {
```

```
            for(int j=0;j<m;j++) {
                if(!visited[i][j]) {
                    if(hasCycle(board,-1,-1,i,j,visited)==1) {
                        return 1;
                    }
                }
            }
        }
        return 0;
    }
   public static int hasCycle(String[] board, int fromX, int fromY, int i, int j, boolean[][] visited) {

        if(visited[i][j]) {
            return 1;
        }

        int[] X_dir = {1,0,0,-1};
        int[] Y_dir = {0,1,-1,0};
        visited[i][j] = true;
        for(int l=0;l<X_dir.length;l++) {
            int x = X_dir[l] + i;
            int y = Y_dir[l] + j;
            if( x==fromX && y == fromY ) {
                    continue;
            }

            if( x>=0 && y>=0 && x<board.length && y<board[x].length() &&
board[x].charAt(y) == board[i].charAt(j) ) {
```

```
                int ans = hasCycle(board, i, j, x, y, visited);

                if(ans == 1) {

                    return 1;

                }



            }



        }

        return 0;





    }





}
```

Largest Piece

Send Feedback

It's Gary's birthday today and he has ordered his favourite square cake consisting of '0's and '1's . But Gary wants the biggest piece of '1's and no '0's . A piece of cake is defined as a part which consist of only '1's, and all '1's share an edge with each other on the cake. Given the size of cake N and the cake, can you find the count of '1's in the biggest piece of '1's for Gary ?

Input Format :

The first line of input contains an integer, that denotes the value of N.
Each of the following N lines contain N space separated integers.

Output Format :

Print the count of '1's in the biggest piece of '1's, according to the description in the task.

Constraints :

1 <= N <= 1000
Time Limit: 1 sec

Sample Input 1:

```
2
1 1
0 1
```

Sample Output 1:

```
3
```

```
public class Solution {



 static int[][] dir = { { 1, 0 }, { -1, 0 }, { 0, 1 }, { 0, -1 } };
```

```
    public static int dfs(String[] edge, int n) {
        /* Your class should be named Solution
         * Don't write main().
         * Don't read input, it is passed as function argument.
         * Return output and don't print it.
         * Taking input and printing output is handled automatically.
  */
  boolean[][] visited = new boolean[n][n];
        int max = 0;
        for(int i=0;i<n;i++){
            for(int j=0;j<n;j++){
                if(!visited[i][j] && edge[i].charAt(j)=='1' ){
                    int ans = max1s( edge,visited,i,j,n);
                    if(max <ans){
                        max = ans;
                    }
                }
            }
        }
        return max;
    }

public static int max1s(String [] edges, boolean [][] visited,int x,int y,int n){
  int [] X_dir = {0,1,-1,0};
  int [] Y_dir = {1,0,0,-1};

  int max= 0;
  visited[x][y] = true;
  for(int i=0;i<X_dir.length;i++){
    int a = X_dir[i]+x;
```

```
            int b = Y_dir[i]+y;



            if(a>=0 && b>=0 && a<n &&

                                                                        b<n && edges[a].charAt(b) =='1'

                                                                        && !visited[a][b] ){

                int ans = max1s(edges,visited,a,b,n);

                max = max+ans;

            }

        }

        return max+1;

    }




}
```

3 Cycle

Send Feedback

Given a graph with N vertices (numbered from 0 to N-1) and M undirected edges, then count the distinct 3-cycles in the graph. A 3-cycle PQR is a cycle in which (P,Q), (Q,R) and (R,P) are connected by an edge.

Input Format :

The first line of input contains two space separated integers, that denotes the value of N and M.
Each of the following M lines contain two integers, that denote the vertices which have an undirected edge between them. Let us denote the two vertices with the symbol u and v.

Output Format :

Print the count the number of 3-cycles in the given graph

Constraints :

0 <= N <= 100
0 <= M <= (N*(N-1))/2
0 <= u <= N - 1
0 <= v <= N - 1
Time Limit: 1 sec

Sample Input 1:

```
3 3
0 1
1 2
2 0
```

Sample Output 1:

```
1
```

```
import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;
```

```
public class Solution {


        static BufferedReader br = new BufferedReader(new InputStreamReader(System.in));


        public static int solve(boolean[][] graph, int n) {
                /*
                 * Your class should be named Solution
                 * You may write your code here
                 */
    int count = 0;
    for(int i=0;i<graph.length;i++){
      for(int j=0;j<graph.length;j++){
        if(graph[i][j]==true){
          for(int k=0;k<graph.length;k++){
            if(k!=i && graph[k][j]==true && graph[i][k]==true){
              count++;
            }
          }
        }
      }
    }
    return count/6;
        }


        public static boolean[][] takeInput() throws IOException {
                String[] strNums;
                strNums = br.readLine().split("\\s");
                int n = Integer.parseInt(strNums[0]);
                int m = Integer.parseInt(strNums[1]);
                boolean[][] graphs = new boolean[n][n];
                int firstvertex, secondvertex;

                for (int i = 0; i < m; i++) {
                    String[] strNums1;
                    strNums1 = br.readLine().split("\\s");
                    firstvertex = Integer.parseInt(strNums1[0]);
                    secondvertex = Integer.parseInt(strNums1[1]);
                    graphs[firstvertex][secondvertex] = true;
                    graphs[secondvertex][firstvertex] = true;
                }
                return graphs;
        }

        public static void main(String[] args) throws NumberFormatException, IOException {
                boolean[][] graphs = takeInput();

                int ans = solve(graphs, graphs.length);
                System.out.println(ans);


        }
}
```