****

**Green University of Bangladesh**

**Department of Computer Science and Engineering (CSE)**

**Faculty of Sciences and Engineering**

**Semester: (Spring, Year: 2023), B.Sc. in CSE (Day)**

**LAB REPORT NO: 02**

**Course Title: Algorithms Lab**

**Course Code: CSE-206 Section: DC**

**Lab Experiment Name: Shortest path**

**Student Details**

|  |  |  |
| --- | --- | --- |
| **Name** | | **ID** |
|  | **Md. Sohanur Rahman** | **213902106** |

**Lab Date : 12.03.2023**

**Submission Date : 17.03.2023**

**Course Teacher’s Name : Md. Sultanul Islam Ovi**

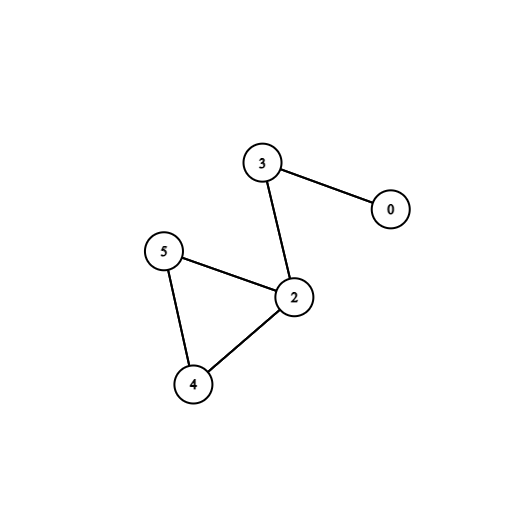
**[For Teachers use only: Don’t Write Anything inside this box]**

|  |
| --- |
| **Lab Report Status**  **Marks: ………………………………… Signature: ...................**  **Comments: .............................................. Date: ..............................** |

**\*\*\*\* Point to be noted: Explanation is given inside the code. \*\*\*\*\*\*\***

**Problem Statement:** Write a program to detect the cycle in a graph using BFS.

**Graph:**

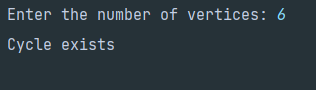


**Figure: 01**

**Code:**

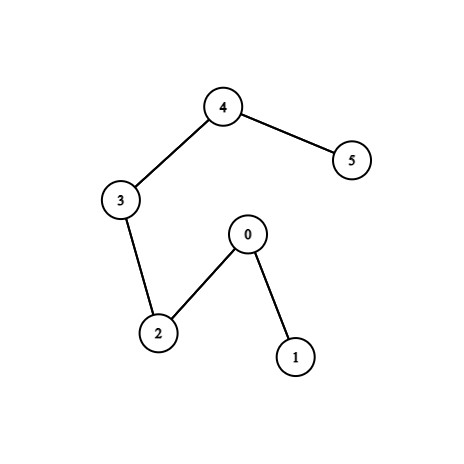
*package* Algorithms\_Lab.LabReportTwo;  
  
*import* java.util.\*;  
  
*//Write a program to detect the cycle in a graph using BFS.  
public class* Task1 {  
 *static* ArrayList<Edge>[] *graph*;  
 *static int v*;  
 *static class* Edge{  
 *int* src,dest;  
 *public* Edge(*int* src,*int* dest){  
 *this*.src=src;  
 *this*.dest=dest;  
 }  
 }  
 *//This method used to create the graph.  
 public static void* createGraph(ArrayList<Edge>[] graph){  
 *for*(*int* i=0;i<graph.length;i++){  
 graph[i]=*new* ArrayList<>();  
 }  
 graph[0].add(*new* Edge(0,3));  
 graph[2].add(*new* Edge(2,3));  
 graph[2].add(*new* Edge(2,4));  
 graph[2].add(*new* Edge(2,5));  
 graph[3].add(*new* Edge(3,0));  
 graph[3].add(*new* Edge(3,2));  
 graph[4].add(*new* Edge(4,2));  
 graph[4].add(*new* Edge(4,5));  
 graph[5].add(*new* Edge(5,2));  
 graph[5].add(*new* Edge(5,4));  
 }  
 *//if this method return true that means there have  
 // cycle in the graph else no cycle.  
 public static boolean* bfs(*int* src,*boolean*[] visited){  
 *Queue*<Integer> queue=*new* LinkedList<>();  
 *int*[] parent=*new int*[*v*];  
 queue.add(src);  
 Arrays.*fill*(parent,-1);  
 *while*(!queue.isEmpty()){  
 *int* current=queue.remove();  
 *if*(!visited[current]){  
 visited[current]=*true*;  
 *for*(*int* i=0;i<*graph*[current].size();i++){  
 Edge e=*graph*[current].get(i);  
 *if*(!visited[e.dest]){  
 parent[e.dest]=current;  
 queue.add(e.dest);  
 *// visited[e.dest]=true;* }*// check this condition if this condition is  
 //true then there is definitely has cycle  
 else if*(parent[current]!=e.dest){  
 *return true*;  
 }  
 }  
 }  
 }  
 *//if return false that means no cycle.  
 return false*;  
 }  
 *//this method create for detect the cycle.  
 public static boolean* isCycle(){  
 *boolean*[] visited=*new boolean*[*v*];  
 *//Initially the visited array is fill by false.* Arrays.*fill*(visited,*false*);  
 *//check the condition v-1 times.  
 for*(*int* i=0;i<*v*;i++){  
 *if*(!visited[i]){  
 *if*(*bfs*(i,visited)){  
 *return true*;  
 }  
 }  
 }  
 *return false*;  
 }  
 *//this is mean function  
 public static void* main(String[] args) {  
 Scanner scan=*new* Scanner(System.***in***);  
 *//The number of vertices is 6.* System.***out***.print("Enter the number of vertices: ");  
 *v*=scan.nextInt();  
 *graph*=*new* ArrayList[*v*];  
 *createGraph*(*graph*);  
 *//if isCycle return true there has cycle  
 if*(*isCycle*()){  
 System.***out***.println("Cycle exists");  
 }  
 *//there has no cycle.  
 else*{  
 System.***out***.println("Cycle not exists");  
 }  
 }  
}

**Output:**



**Problem Statement:** Write a program to find the level of each node using BFS.

**Graph:**

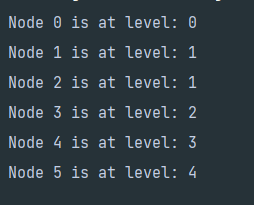
****

**Figure: 02**

**Code:**

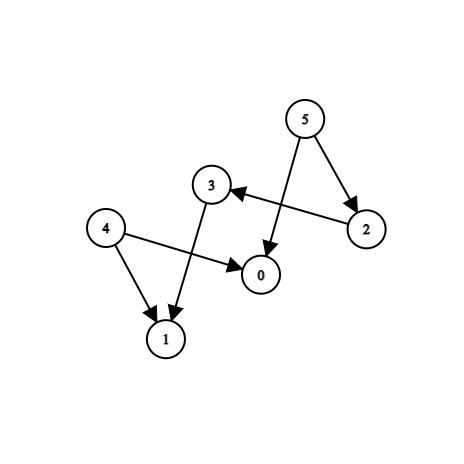
*package* Algorithms\_Lab.LabReportTwo;  
  
  
*import* java.util.ArrayList;  
*import* java.util.LinkedList;  
*import* java.util.*Queue*;  
  
*//Write a program to find the level of each node using BFS.  
public class* Task2 {  
 *//Here create Edge.  
 static class* Edge{  
 *int* src,dest;  
 *public* Edge(*int* src,*int* dest){  
 *this*.src=src;  
 *this*.dest=dest;  
 }  
 }  
 *//This method create the graph.  
 public static void* createGraph(ArrayList<Edge>[] graph){  
 *//  
 for*(*int* i=0;i<graph.length;i++){  
 graph[i]=*new* ArrayList<>();  
 }  
 graph[0].add(*new* Edge(0,1));  
 graph[0].add(*new* Edge(0,2));  
 graph[1].add(*new* Edge(1,0));  
 graph[2].add(*new* Edge(2,0));  
 graph[2].add(*new* Edge(2,3));  
 graph[3].add(*new* Edge(3,2));  
 graph[3].add(*new* Edge(3,4));  
 graph[4].add(*new* Edge(4,3));  
 graph[4].add(*new* Edge(4,5));  
 graph[5].add(*new* Edge(5,4));  
 }  
 *//Here calculate level of every node using BFS.  
 public static void* nodeLevel(ArrayList<Edge>[] graph,*int* src){  
 *Queue*<Integer> queue=*new* LinkedList<>();  
 *boolean*[] visited=*new boolean*[graph.length];  
 *int*[] level=*new int*[graph.length];  
 queue.add(src);  
 level[src]=0;  
 *while*(!queue.isEmpty()){  
 *int* current=queue.remove();  
 System.***out***.println("Node "+current+" is at level: "+level[current]);  
 *if*(!visited[current]){  
 visited[current]=*true*;  
 *//push the not visited node in the queue.  
 for*(*int* i=0;i< graph[current].size();i++){  
 Edge e=graph[current].get(i);  
 *if*(!visited[e.dest]){  
 level[e.dest]=level[current]+1;  
 queue.add(e.dest);  
 }  
 }  
  
 }  
  
 }  
 }  
 *public static void* main(String[] args) {  
 *int* v=6;  
 *//Allocate memory for the graph.* ArrayList<Edge>[] graph=*new* ArrayList[v];  
 *createGraph*(graph);  
 *nodeLevel*(graph,0);  
  
 }  
}

**Output:**

****

**Problem Statement:** Write a program to perform topological sort using BFS.

**Graph:**

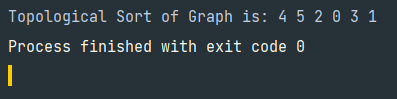
****

**Figure: 03**

**Code:**

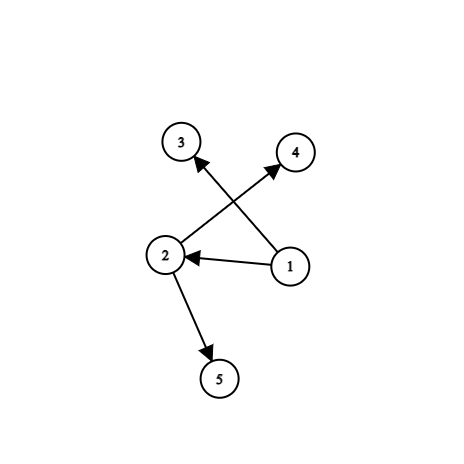
*package* Algorithms\_Lab.LabReportTwo;  
  
*import* java.util.ArrayList;  
*import* java.util.LinkedList;  
*import* java.util.*Queue*;  
  
*public class* Task3 {  
 *//Edge class for create given graph.  
 static class* Edge{  
 *int* src,dest;  
 *public* Edge(*int* src,*int* dest){  
 *this*.src=src;  
 *this*.dest=dest;  
 }  
 }  
 *//Here graph is created.  
 public static void* createGraph(ArrayList<Edge>[] graph){  
 *for*(*int* i=0;i< graph.length;i++){  
 graph[i]=*new* ArrayList<>();  
 }  
 graph[2].add(*new* Edge(2,3));  
 graph[3].add(*new* Edge(3,1));  
 graph[4].add(*new* Edge(4,0));  
 graph[4].add(*new* Edge(4,1));  
 graph[5].add(*new* Edge(5,2));  
 graph[5].add(*new* Edge(5,0));  
 }  
 *//This method create for sorting  
 public static void* topologicalSort(ArrayList<Edge>[] graph,*int* v){  
 *Queue*<Integer> queue=*new* LinkedList<>();  
 *boolean*[] visited=*new boolean*[v];  
 *int*[] inDegree=*new int*[v];  
 *//calculate inDegree of every node.  
 for*(*int* i=0;i<v;i++){  
 *for*(*int* j=0;j<graph[i].size();j++){  
 Edge e=graph[i].get(j);  
 inDegree[e.dest]++;  
 }  
 }  
 *//Add the node in the queue.  
 for*(*int* i=0;i<inDegree.length;i++){  
 *if*(inDegree[i]==0){  
 queue.add(i);  
 }  
 }  
 *//pop sorted node from the queue and print.  
 while*(!queue.isEmpty()){  
 *int* current=queue.remove();  
 visited[current]=*true*;  
 System.***out***.print(current+" ");  
 *//find all the unvisited node .  
 for*(*int* i=0;i< graph[current].size();i++){  
 Edge e=graph[current].get(i);  
 *if*(!visited[e.dest]){  
 inDegree[e.dest]--;  
 *//if inDegree=0 then add in the queue.  
 if*(inDegree[e.dest]==0){  
 queue.add(e.dest);  
 }  
 }  
 }  
 }  
 }  
 *public static void* main(String[] args) {  
 *int* v=6;  
 ArrayList<Edge>[] graph=*new* ArrayList[v];  
 *//call createGraph for the following graph.  
 createGraph*(graph);  
 System.***out***.print("Topological Sort of Graph is: ");  
 *//At last call the method and print the sorted node.  
 topologicalSort*(graph,v);  
 }  
}

**Output:**

****

**Problem Statement:** Write a program to find the minimum depth of a binary tree.

**Graph:**

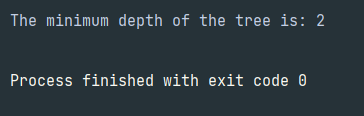
****

**Figure: 04**

**Code:**

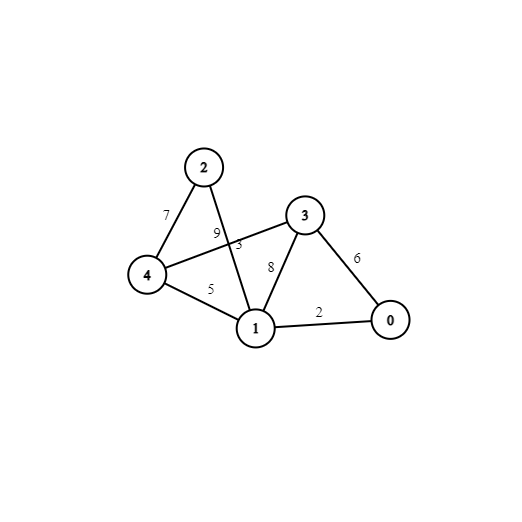
*package* Algorithms\_Lab.LabReportTwo;  
  
*import* java.util.LinkedList;  
*import* java.util.*Queue*;  
  
*public class* Task4 {  
 *static class* TreeNode{  
 *int* value;  
 TreeNode left;  
 TreeNode right;  
 TreeNode(*int* value){  
 *this*.value=value;  
 }  
 }  
 *//this method calculate the minimum depth  
 //and finally return the depth.  
 public static int* minDepth(TreeNode root){  
 *Queue*<TreeNode> queue=*new* LinkedList<>();  
 *//check the root of tree is null or not.  
 if*(root==*null*){  
 *return* 0;  
 }  
 queue.add(root);  
 *int* level=1;  
 *//find the minimum depth until the condition is false.  
 while*(!queue.isEmpty()){  
 *int* size=queue.size();  
 *for*(*int* i=0;i<size;i++) {  
 TreeNode node=queue.remove();  
 *//if these condition is true that means  
 //we reach the leaf node.  
 if* (node.left == *null* && node.right == *null*) {  
 *return* level;  
 }  
 *//until find the leaf node the below  
 //condition is executed.  
 if*(node.left!=*null*){  
 queue.add(node.left);  
 }  
 *if*(node.right!=*null*){  
 queue.add(node.right);  
 }  
 }  
 level++;  
 }  
 *//finally return the level of minimum depth.  
 return* level;  
 }  
 *public static void* main(String[] args) {  
 *//Here create the tree.* TreeNode root=*new* TreeNode(1);  
 root.left=*new* TreeNode(2);  
 root.right=*new* TreeNode(3);  
 root.left.left=*new* TreeNode(4);  
 root.left.right=*new* TreeNode(5);  
 *//root.right.right=new TreeNode(6);  
 //At last print the minimum depth or minimum distance  
 //from root to leaf node is printed.* System.***out***.println("The minimum depth of the tree is: "+*minDepth*(root));  
 }  
}

**Output:**

****

**Problem Statement:** Write a program to find the number of distinct minimum spanning trees for a given weighted graph using Kruskal algorithm.

**Graph:**

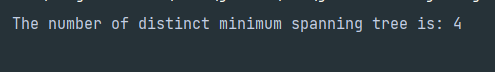
****

**Figure: 05**

**Code:**

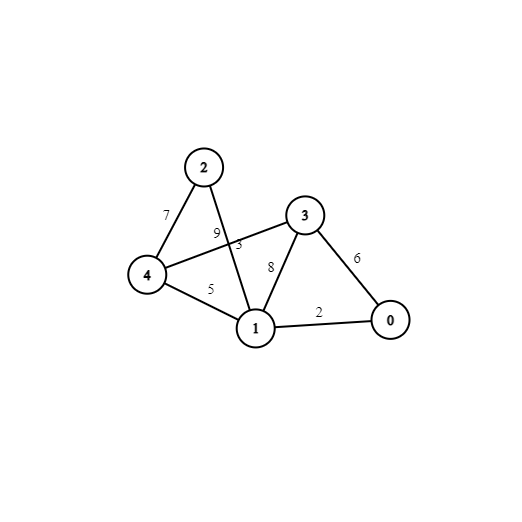
*package* Algorithms\_Lab.LabReportTwo;  
  
*import* java.util.ArrayList;  
*import* java.util.Arrays;  
*import* java.util.Collections;  
*import* java.util.*List*;  
  
*public class* Task5 {  
 *int* vertices;  
 *int*[][] adjacencyMatrix;  
 *List*<Edge> edges;  
 *public* Task5(*int* v){  
 *this*.vertices=v;  
 adjacencyMatrix=*new int*[v][v];  
 edges=*new* ArrayList<Edge>();  
 }  
 *//addEdge method for create a undirected graph.  
 //and add the adjacency nodes.  
 public void* addEdge(*int* i,*int* j,*int* w){  
 adjacencyMatrix[i][j]=w;  
 adjacencyMatrix[j][i]=w;  
 edges.add(*new* Edge(i,j,w));  
 }  
 *//count the number of distinct minimum spanning tree.  
 public int* countDistinctMst(){  
 *int* count=0;  
 *int*[] parent=*new int*[vertices];  
 Arrays.*fill*(parent,-1);  
 Collections.*sort*(edges);  
 *//Enhance for loop for find the unvisited node.  
 for*(Edge e: edges){  
 *int* u=e.src;  
 *int* v=e.dest;  
 *if*(find(parent,u)!=find(parent,v)){  
 count++;  
 union(parent,u,v);  
 }  
 }  
 *return* count;  
 }  
 *private int* find(*int*[] parent, *int* i) {  
 *if* (parent[i] == -1) {  
 *return* i;  
 }  
  
 *return* find(parent, parent[i]);  
 }  
 *private void* union(*int*[] parent, *int* x, *int* y) {  
 *int* xSet = find(parent, x);  
 *int* ySet = find(parent, y);  
  
 parent[xSet] = ySet;  
 }  
 *//Edge class for create the edges of graph.  
 class* Edge *implements Comparable*<Edge>{  
 *int* src,dest,weight;  
 *public* Edge(*int* src,*int* dest,*int* weight){  
 *this*.src=src;  
 *this*.dest=dest;  
 *this*.weight=weight;  
 }  
 *//This method return the nodes by sorting  
 //Depends on their weight.  
 public int* compareTo(Edge p2){  
 *return this*.weight-p2.weight;  
 }  
 }  
 *public static void* main(String[] args) {  
 *//Here number of vertices is five in graph.* Task5 obj=*new* Task5(5);  
 obj.addEdge(0, 1, 2);  
 obj.addEdge(0, 3, 6);  
 obj.addEdge(1, 2, 3);  
 obj.addEdge(1, 3, 8);  
 obj.addEdge(1, 4, 5);  
 obj.addEdge(2, 4, 7);  
 obj.addEdge(3, 4, 9);  
 *int* count=obj.countDistinctMst();  
 System.***out***.println("The number of distinct minimum spanning tree is: "+count);  
 }  
}

**Output:**

****

**Problem Statement:** Write a program to find the number of distinct minimum spanning trees for a given weighted graph using the Prim algorithm.

**Graph:**

****

**Figure: 06**

**Code:**

*package* Algorithms\_Lab.LabReportTwo;  
  
*import* java.util.ArrayList;  
*import* java.util.Arrays;  
*import* java.util.*List*;  
  
*public class* Task6 {  
 *//Declar some variable and array.  
 private int* vertices;  
 *private int*[][] adjacencyMatrix;  
 *private int*[] key;  
 *private boolean*[] visited;  
 *private int*[] parent;  
 *private List*<Integer> mstVertices;  
 *//Allocate memory for variable and arrays.  
 public* Task6(*int* v){  
 *this*.vertices=v;  
 adjacencyMatrix=*new int*[v][v];  
 key=*new int*[v];  
 visited= *new boolean*[v];  
 parent=*new int*[v];  
 mstVertices=*new* ArrayList<>();  
 Arrays.*fill*(key, Integer.***MAX\_VALUE***);  
 Arrays.*fill*(visited,*false*);  
 Arrays.*fill*(parent,-1);  
 }  
 *//addEdge method for create the graph.  
 public void* addEdge(*int* i,*int* j,*int* weight){  
 adjacencyMatrix[i][j]=weight;  
 adjacencyMatrix[j][i]=weight;  
 }  
 *//this method count the number of distinct minimum spanning trees.  
 public int* findDistinctMSTs() {  
 *int* count = 0;  
  
 *for* (*int* i = 0; i < vertices; i++) {  
 *if* (!visited[i]) {  
 count++;  
 mstVertices.clear();  
 mstVertices.add(i);  
  
 key[i] = 0;  
 visited[i] = *true*;  
  
 *while* (!mstVertices.isEmpty()) {  
 *int* u = mstVertices.get(0);  
 mstVertices.remove(0);  
  
 *for* (*int* v = 0; v < vertices; v++) {  
 *if* (adjacencyMatrix[u][v] > 0 && !visited[v]) {  
 *if* (adjacencyMatrix[u][v] < key[v]) {  
 parent[v] = u;  
 key[v] = adjacencyMatrix[u][v];  
 }  
  
 *if* (adjacencyMatrix[u][v] == key[v]) {  
 *if* (!mstVertices.contains(v)) {  
 mstVertices.add(v);  
 }  
 }  
 }  
 }  
  
 visited[u] = *true*;  
 }  
 }  
 }  
  
 *return* count;  
 }  
 *public static void* main(String[] args) {  
 Task6 obj=*new* Task6(5);  
 obj.addEdge(0, 1, 2);  
 obj.addEdge(0, 3, 6);  
 obj.addEdge(1, 2, 3);  
 obj.addEdge(1, 3, 8);  
 obj.addEdge(1, 4, 5);  
 obj.addEdge(2, 4, 7);  
 obj.addEdge(3, 4, 9);  
  
 *int* count = obj.findDistinctMSTs();  
 System.***out***.println("Number of distinct minimum spanning trees is: " + count);  
 }  
}

**Output:**

****