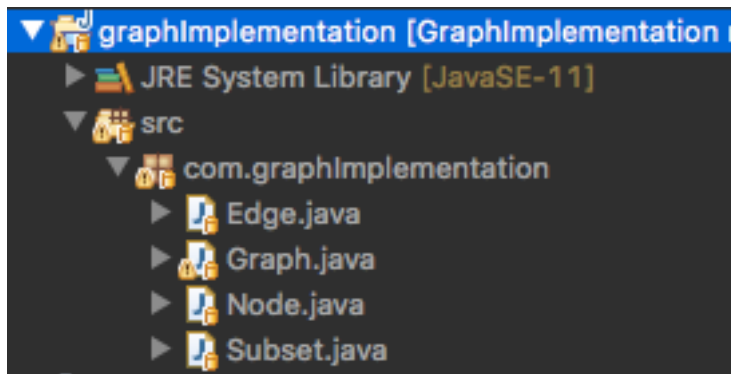


INTRODUCTION

My project consists of a package and four classes. My project is a Java console project.



Also, all output is here.

```
Console x Git Staging Git Repositories
<terminated> Graph [Java Application] /Library/Java/JavaVirtualMachines
Degree Of Vertex : 5

-----
Breadth First Traversal : 2 0 3 1

-----
Depth First Traversal : 2 0 1 3

-----
Kruskal's Minimum Spanning Tree :
2 --- 3 == 4
0 --- 3 == 5
0 --- 1 == 10

-----
Shortest Path :
Path (0 -> 1): Minimum Cost = 4 and Route is [0, 4, 1]
Path (0 -> 2): Minimum Cost = 6 and Route is [0, 4, 1, 2]
Path (0 -> 3): Minimum Cost = 5 and Route is [0, 4, 3]
Path (0 -> 4): Minimum Cost = 3 and Route is [0, 4]

-----
Strongly connected components :
4
0 2 1
3

-----
0 1 2 3 Given directed graph is NOT eulerian
```

1. Degree of a Vertex

For find the degree of a vertex I use two classes, first one is Edge Class and also Graph class that my main class.

```
public class Graph {  
    private int V, E;  
    private int[][] dir;  
    private int in[];  
    private LinkedList<Integer> adj[];  
    Edge edge[];  
    public List<List<Edge>> adjList = null;  
}
```

```
public class Edge implements Comparable<Edge> {  
    public int src, dest, weight;  
    public Edge() {  
    }  
    public Edge(int src, int dest, int weight) {  
        super();  
        this.src = src;  
        this.dest = dest;  
        this.weight = weight;  
    }  
    public int compareTo(Edge compareEdge)  
    {  
        return this.weight-compareEdge.weight;  
    }  
}
```

Graph class constructor method is;

```
public Graph(int v, int e) {  
    this.V = v;  
    this.E = e;  
    dir = new int[v][];  
    for (int i = 0; i < v; i++) {  
        dir[i] = new int[v];  
    }  
    edge = new Edge[E];  
    for (int i = 0; i < e; ++i) {  
        edge[i] = new Edge();  
    }  
}
```

For the given vertex then check if a path from this vertices to other exists then increment the degree. Then, Return degree;

```
public static int findVertexDegree(Graph G, int v) {  
    int degree = 0;  
    for (int i = 0; i < G.V; i++) {  
        if (G.dir[v][i] == 1)  
            degree++;  
    }  
    return degree;  
}
```

In “public static void main(String args[])” method;

Create the graphs adjacency matrix from src to des (Edge Class) ;

```
Graph G = new Graph(6, 7);
```

```
G.dir[0][1] = 1;  
G.dir[0][2] = 1;  
G.dir[0][3] = 1;  
G.dir[0][4] = 1;  
G.dir[0][5] = 1;
```

```
G.dir[1][0] = 1;  
G.dir[1][2] = 1;  
G.dir[1][3] = 1;  
G.dir[1][4] = 1;  
G.dir[1][5] = 1;
```

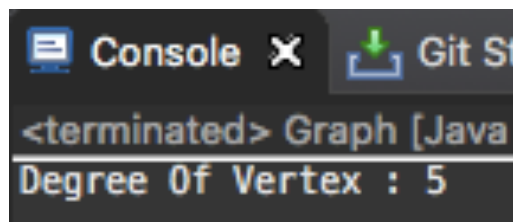
```
G.dir[2][0] = 1;  
G.dir[2][1] = 1;  
G.dir[2][3] = 1;  
G.dir[2][4] = 1;  
G.dir[2][5] = 1;
```

```
G.dir[3][0] = 1;  
G.dir[3][1] = 1;  
G.dir[3][2] = 1;  
G.dir[3][4] = 1;  
G.dir[3][5] = 1;
```

```
G.dir[4][0] = 1;  
G.dir[4][1] = 1;  
G.dir[4][2] = 1;  
G.dir[4][3] = 1;  
G.dir[4][5] = 1;
```

```
G.dir[5][0] = 1;  
G.dir[5][1] = 1;  
G.dir[5][2] = 1;  
G.dir[5][3] = 1;  
G.dir[5][4] = 1;
```

```
System.out.println("Degree Of Vertex : " + findVertexDegree(G, 0));  
System.out.println();  
System.out.println("-----");
```



```
<terminated> Graph [Java  
Degree Of Vertex : 5
```

2. Breadth First Search

The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. This class represents a directed graph using adjacency list representation.

```
public class Graph {  
    private int V, E;  
    private int[][] dir;  
    private int in[];  
    private LinkedList<Integer> adj[];  
    Edge edge[];  
    public List<List<Edge>> adjList = null;  
}
```

Graph class constructor method is;

```
public Graph(int v) {  
    V = v;  
    adj = new LinkedList[V];  
    in = new int[V];  
    for (int i = 0; i < v; ++i) {  
        adj[i] = new LinkedList();  
        in[i] = 0;  
    }  
}
```

Function to add an edge into the graph

```
public void addEdge(int v, int w) {  
    adj[v].add(w);  
    in[w]++;  
}
```

Prints BFS traversal from a given source s,

```
public void BFS(int s) {  
    boolean visited[] = new boolean[V];  
    BFSUtil(s, visited);  
}
```

Get all adjacent vertices of the dequeued vertex s, If an adjacent has not been visited, then mark it visited and enqueue it;

```

void BFSUtil(int s, boolean visited[]) {
    visited[s] = true;
    LinkedList<Integer> queue = new LinkedList<Integer>();
    queue.add(s);
    while (queue.size() != 0) {
        s = queue.poll();
        System.out.print(s + " ");
        Iterator<Integer> i = adj[s].listIterator();
        while (i.hasNext()) {
            int n = i.next();
            if (!visited[n]) {
                visited[n] = true;
                queue.add(n);
            }
        }
    }
}

```

In “public static void main(String args[])” method;

```

Graph g = new Graph(5);

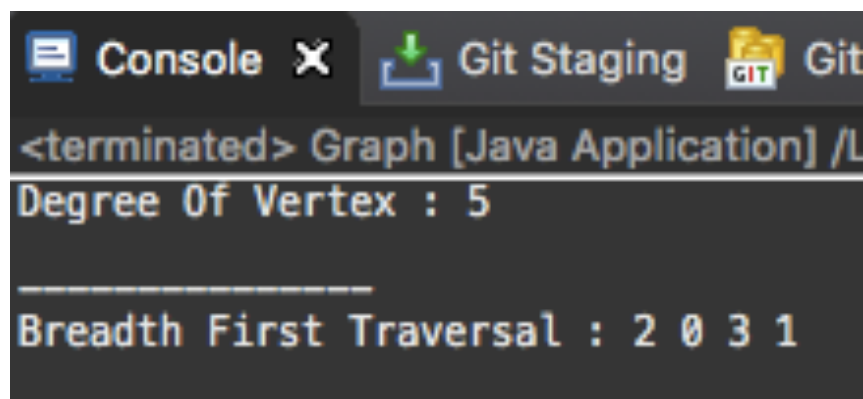
g.addEdge(0, 1);
g.addEdge(0, 2);
g.addEdge(1, 2);
g.addEdge(2, 0);
g.addEdge(2, 3);
g.addEdge(3, 3);

```

```

System.out.print("Breadth First Traversal : ");
g.BFS(2);
System.out.println();
System.out.println();
System.out.println("-----");

```



```

<terminated> Graph [Java Application] /L
Degree Of Vertex : 5
-----
Breadth First Traversal : 2 0 3 1

```

3. Depth First Search

Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally print the nodes in the path.

```
public class Graph {  
    private int V, E;  
    private int[][] dir;  
    private int in[];  
    private LinkedList<Integer> adj[];  
    Edge edge[];  
    public List<List<Edge>> adjList = null;  
}
```

Graph class constructor method is;

```
public Graph(int v) {  
    V = v;  
    adj = new LinkedList[V];  
    in = new int[V];  
    for (int i = 0; i < v; ++i) {  
        adj[i] = new LinkedList();  
        in[i] = 0;  
    }  
}
```

```
public void addEdge(int v, int w) {  
    adj[v].add(w);  
    in[w]++;  
}
```

Mark the current node as visited and print it, Recur for all the vertices adjacent to this vertex;

```
void DFSUtil(int v, boolean visited[]) {  
    visited[v] = true;  
    System.out.print(v + " ");  
    Iterator<Integer> i = adj[v].listIterator();  
    while (i.hasNext()) {  
        int n = i.next();  
        if (!visited[n])  
            DFSUtil(n, visited);  
    }  
}
```

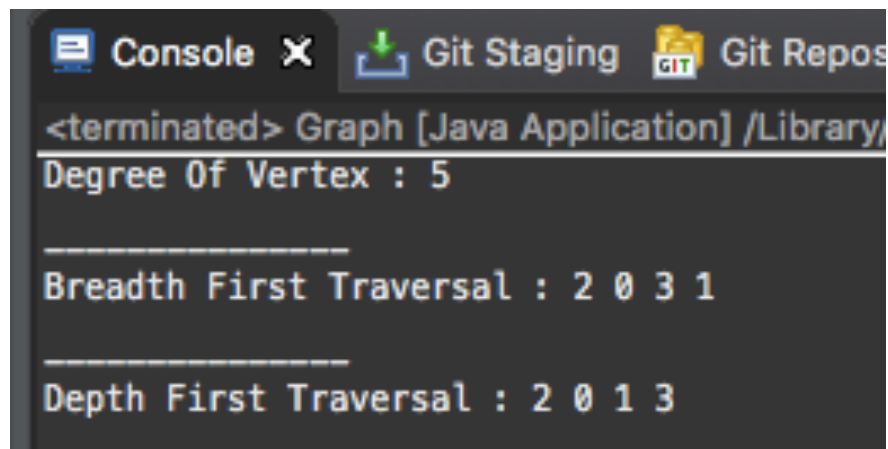
The function to do DFS traversal. It uses recursive DFSUtil();

```
void DFS(int v) {  
    boolean visited[] = new boolean[V];  
    DFSUtil(v, visited);  
}
```

In “public static void main(String args[])” method;

```
Graph g = new Graph(5);  
g.addEdge(0, 1);  
g.addEdge(0, 2);  
g.addEdge(1, 2);  
g.addEdge(2, 0);  
g.addEdge(2, 3);  
g.addEdge(3, 3);
```

```
System.out.print("Depth First Traversal : ");  
g.DFS(2);  
System.out.println();  
System.out.println();  
System.out.println("-----");
```



```
<terminated> Graph [Java Application] /Library/  
Degree Of Vertex : 5  
  
-----  
Breadth First Traversal : 2 0 3 1  
  
-----  
Depth First Traversal : 2 0 1 3
```

4. Minimum Spanning Tree

Given a connected and undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

For find the Minimum Spanning Tree I use three classes, first one is Edge Class, second one is Subset class and also Graph class that my main class.

```
public class Graph {  
    private int V, E;  
    private int[][] dir;  
    private int in[];  
    private LinkedList<Integer> adj[];  
    Edge edge[];  
    public List<List<Edge>> adjList = null;
```

```
public class Subset {  
    public int parent, rank;  
}
```

```
public class Edge implements Comparable<Edge> {  
    public int src, dest, weight;  
    public Edge() {  
    }  
    public Edge(int src, int dest, int weight) {  
        super();  
        this.src = src;  
        this.dest = dest;  
        this.weight = weight;  
    }  
}
```

A utility function to find set of an element i ,(uses path compression technique);

```
public int find(Subset subsets[], int i) {  
    if (subsets[i].parent != i)  
        subsets[i].parent = find(subsets, subsets[i].parent);  
    return subsets[i].parent;  
}
```


A function that does union of two sets of x and y (uses union by rank);

```
public void Union(Subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}
```

The main function to construct MST using Kruskal's algorithm

- 1: Sort all the edges in non-decreasing order of their weight. If we are not allowed to change the given graph, we can create a copy of array of edges,
- 2: Pick the smallest edge. And increment the index for next iteration, also print the contents of result[] to display the built MST.

```
public void KruskalMST() {
    Edge result[] = new Edge[V];
    int e = 0;
    int i = 0;
    for (i = 0; i < V; ++i)
        result[i] = new Edge();

    Arrays.sort(edge);

    Subset subsets[] = new Subset[V];
    for (i = 0; i < V; ++i)
        subsets[i] = new Subset();

    for (int v = 0; v < V; ++v) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    i = 0;
    while (e < V - 1) {
        Edge next_edge = new Edge();
        next_edge = edge[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);
        if (x != y) {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
    }

    System.out.println("Kruskal's Minimum Spanning Tree : ");
    for (i = 0; i < e; ++i)
        System.out.println(result[i].src + " — " + result[i].dest + " == " + result[i].weight);
}
```

In “public static void main(String args[])” method;

```
Graph graph = new Graph(4, 5);

graph.edge[0].src = 0;
graph.edge[0].dest = 1;
graph.edge[0].weight = 10;

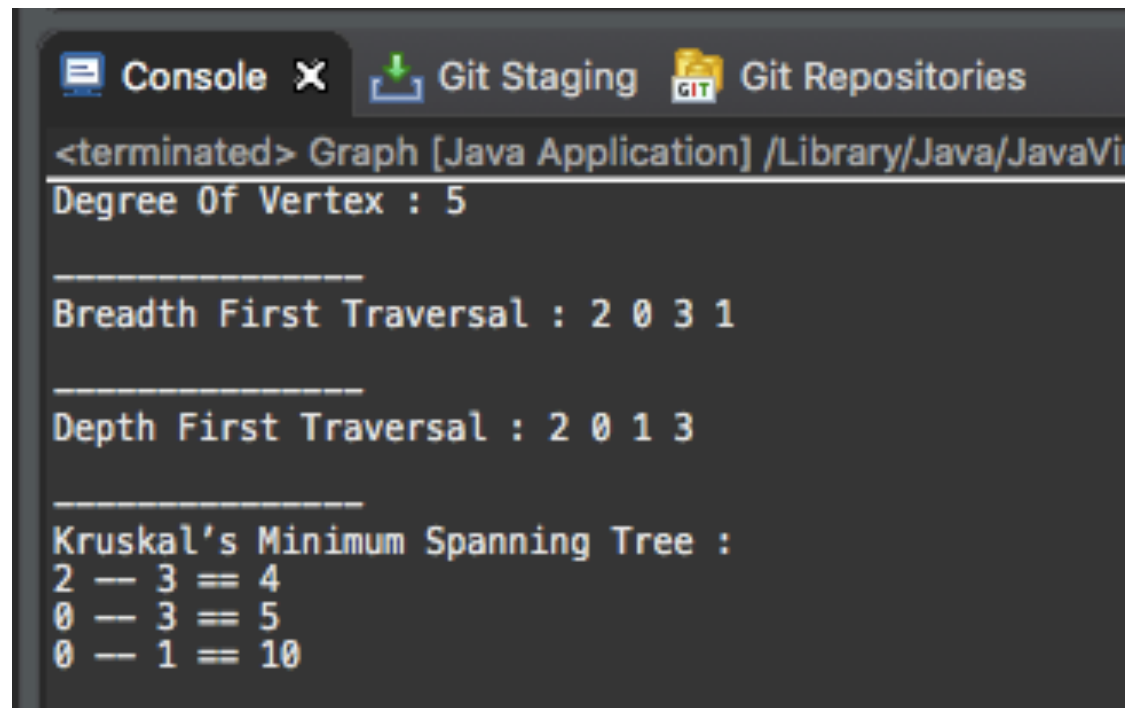
graph.edge[1].src = 0;
graph.edge[1].dest = 2;
graph.edge[1].weight = 6;

graph.edge[2].src = 0;
graph.edge[2].dest = 3;
graph.edge[2].weight = 5;

graph.edge[3].src = 1;
graph.edge[3].dest = 3;
graph.edge[3].weight = 15;

graph.edge[4].src = 2;
graph.edge[4].dest = 3;
graph.edge[4].weight = 4;
```

```
graph.KruskalMST();
System.out.println();
System.out.println("-----");
```



```
<terminated> Graph [Java Application] /Library/Java/JavaVi
Degree Of Vertex : 5

-----
Breadth First Traversal : 2 0 3 1

-----
Depth First Traversal : 2 0 1 3

-----
Kruskal's Minimum Spanning Tree :
2 --- 3 == 4
0 --- 3 == 5
0 --- 1 == 10
```

5. Single Source Shortest Path

For find the Single Source Shortest Path, I use three classes, first one is Edge Class, second one is Node class and also Graph class that my main class.

```
public class Graph {  
    private int V, E;  
    private int[][] dir;  
    private int in[];  
    private LinkedList<Integer> adj[];  
    Edge edge[];  
    public List<List<Edge>> adjList = null;  
}
```

```
public class Edge implements Comparable<Edge> {  
    public int src, dest, weight;  
    public Edge() {  
    }  
    public Edge(int src, int dest, int weight) {  
        super();  
        this.src = src;  
        this.dest = dest;  
        this.weight = weight;  
    }  
}
```

```
public class Node {  
    public int vertex, weight;  
    public Node(int vertex, int weight) {  
        this.vertex = vertex;  
        this.weight = weight;  
    }  
}
```

For this situation, I create another Constructor method;

```
public Graph(List<Edge> edges, int N) {  
    adjList = new ArrayList<>(N);  
    for (int i = 0; i < N; i++) {  
        adjList.add(i, new ArrayList<>());  
    }  
    for (Edge edge : edges) {  
        adjList.get(edge.src).add(edge);  
    }  
}
```

Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree. Like Prim's MST, we generate a SPT (shortest path tree) with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

```
public static void shortestPath(Graph graph, int source, int N) {
    PriorityQueue<Node> minHeap;
    minHeap = new PriorityQueue<>((Comparator.comparingInt(node -> node.weight)));
    minHeap.add(new Node(source, 0));

    List<Integer> dist = new ArrayList<>(Collections.nCopies(N, Integer.MAX_VALUE));
    dist.set(source, 0);

    boolean[] done = new boolean[N];
    done[source] = true;

    int prev[] = new int[N];
    prev[source] = -1;

    List<Integer> route = new ArrayList<>();

    while (!minHeap.isEmpty()) {
        Node node = minHeap.poll();
        int u = node.vertex;

        for (Edge edge : graph.adjList.get(u)) {
            int v = edge.dest;
            int weight = edge.weight;

            if (!done[v] && (dist.get(u) + weight) < dist.get(v)) {
                dist.set(v, dist.get(u) + weight);
                prev[v] = u;
                minHeap.add(new Node(v, dist.get(v)));
            }
        }
        done[u] = true;
    }

    for (int i = 1; i < N; ++i) {
        if (i != source && dist.get(i) != Integer.MAX_VALUE) {
            getRoute(prev, i, route);
            System.out.printf("Path (%d -> %d): Minimum Cost = %d and Route is %s\n", source, i, dist.get(i), route);
            route.clear();
        }
    }
}
```

```
private static void getRoute(int prev[], int i, List<Integer> route) {
    if (i >= 0) {
        getRoute(prev, prev[i], route);
        route.add(i);
    }
}
```

In “public static void main(String args[])” method;

```
List<Edge> edges = Arrays.asList(new Edge(0, 1, 10), new Edge(0, 4, 3), new Edge(1, 2, 2), new Edge(1, 4, 4),
    new Edge(2, 3, 9), new Edge(3, 2, 7), new Edge(4, 1, 1), new Edge(4, 2, 8), new Edge(4, 3, 2));

Graph gra = new Graph(edges, 5);
```

```

System.out.println("Shortest Path : ");
shortestPath(gra, 0, 5);
System.out.println();
System.out.println("-----");

```

```

Console X Git Staging Git Repositories
<terminated> Graph [Java Application] /Library/Java/JavaVirtualMachines
0 -- 3 == 5
0 -- 1 == 10

-----
Shortest Path :
Path (0 -> 1): Minimum Cost = 4 and Route is [0, 4, 1]
Path (0 -> 2): Minimum Cost = 6 and Route is [0, 4, 1, 2]
Path (0 -> 3): Minimum Cost = 5 and Route is [0, 4, 3]
Path (0 -> 4): Minimum Cost = 3 and Route is [0, 4]

```

6. Strongly Connected Components

DFS of a graph produces a single tree if all vertices are reachable from the DFS starting point. Otherwise DFS produces a forest. So DFS of a graph with only one SCC always produces a tree. The important point to note is DFS may produce a tree or a forest when there are more than one SCCs depending upon the chosen starting point.

```

public class Graph {
    private int V, E;
    private int[][] dir;
    private int in[];
    private LinkedList<Integer> adj[];
    Edge edge[];
    public List<List<Edge>> adjList = null;
}

```

```

public Graph(int v) {
    V = v;
    adj = new LinkedList[V];
    in = new int[V];
    for (int i = 0; i < v; ++i) {
        adj[i] = new LinkedList();
        in[i] = 0;
    }
}

```

Create an empty stack 'S' and do DFS traversal of a graph. In DFS traversal, after calling recursive DFS for adjacent vertices of a vertex, push the vertex to stack. In the above graph, if we start DFS from vertex 0, we get vertices in stack as 1, 2, 4, 3, 0.

```

public void printSCCs() {
    Stack stack = new Stack();

    boolean visited[] = new boolean[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            fillOrder(i, visited, stack);

    Graph gr = getTranspose();

    for (int i = 0; i < V; i++)
        visited[i] = false;

    while (stack.empty() == false) {
        int v = (int) stack.pop();

        if (visited[v] == false) {
            gr.DFSUtil(v, visited);
            System.out.println();
        }
    }
}

```

Reverse directions of all arcs to obtain the transpose graph.

```

public Graph getTranspose() {
    Graph g = new Graph(V);
    for (int v = 0; v < V; v++) {
        Iterator<Integer> i = adj[v].listIterator();
        while (i.hasNext())
            g.adj[i.next()].add(v);
        (g.in[v])++;
    }
    return g;
}

```

One by one pop a vertex from S while S is not empty. Let the popped vertex be 'v'. Take v as source and do DFS (call DFSUtil(v)). The DFS starting from v prints strongly connected component of v. In the above example, we process vertices in order 0, 3, 4, 2, 1 (One by one popped from stack).

```

public void fillOrder(int v, boolean visited[], Stack stack) {
    visited[v] = true;

    Iterator<Integer> i = adj[v].iterator();
    while (i.hasNext()) {
        int n = i.next();
        if (!visited[n])
            fillOrder(n, visited, stack);
    }

    stack.push(new Integer(v));
}

```

```

void DFSUtil(int v, boolean visited[]) {
    visited[v] = true;
    System.out.print(v + " ");

    Iterator<Integer> i = adj[v].listIterator();
    while (i.hasNext()) {
        int n = i.next();
        if (!visited[n])
            DFSUtil(n, visited);
    }
}

```

In “public static void main(String args[])” method;

```

Graph g = new Graph(5);

g.addEdge(0, 1);
g.addEdge(0, 2);
g.addEdge(1, 2);
g.addEdge(2, 0);
g.addEdge(2, 3);
g.addEdge(3, 3);

```

```

System.out.println("Strongly connected components : ");
g.printSCCs();
System.out.println();
System.out.println("-----");

```

```

<terminated> Graph [Java Application] /Library/Java/JavaVirtualMachines
0 -- 3 == 5
0 -- 1 == 10

-----
Shortest Path :
Path (0 -> 1): Minimum Cost = 4 and Route is [0, 4, 1]
Path (0 -> 2): Minimum Cost = 6 and Route is [0, 4, 1, 2]
Path (0 -> 3): Minimum Cost = 5 and Route is [0, 4, 3]
Path (0 -> 4): Minimum Cost = 3 and Route is [0, 4]

-----
Strongly connected components :
4
0 2 1
3

```

7. Eulerian Circuit

A directed graph has an eulerian cycle if following conditions are true

- 1) All vertices with nonzero degree belong to a single strongly connected component.
- 2) In degree is equal to the out degree for every vertex.

```
public class Graph {  
    private int V, E;  
    private int[][] dir;  
    private int in[];  
    private LinkedList<Integer> adj[];  
    Edge edge[];  
    public List<List<Edge>> adjList = null;
```

```
    public Graph(int v) {  
        V = v;  
        adj = new LinkedList[v];  
        in = new int[V];  
        for (int i = 0; i < v; ++i) {  
            adj[i] = new LinkedList();  
            in[i] = 0;  
        }  
    }  
}
```

To compare in degree and out-degree, we need to store in degree and out-degree of every vertex. Out degree can be obtained by the size of an adjacency list. In degree can be stored by creating an array of size equal to the number of vertices.

The main function that returns true if graph is strongly connected;

```
Boolean isSC() {  
    boolean visited[] = new boolean[V];  
    for (int i = 0; i < V; i++)  
        visited[i] = false;  
    DFSUtil(0, visited);  
    for (int i = 0; i < V; i++)  
        if (visited[i] == false)  
            return false;  
    Graph gr = getTranspose();  
    for (int i = 0; i < V; i++)  
        visited[i] = false;  
    gr.DFSUtil(0, visited);  
    for (int i = 0; i < V; i++)  
        if (visited[i] == false)  
            return false;  
    return true;  
}
```


This function returns true if the directed graph has a eulerian cycle, otherwise returns false ;

```
Boolean isEulerianCycle() {  
    if (isSC() == false)  
        return false;  
  
    for (int i = 0; i < V; i++)  
        if (adj[i].size() != in[i])  
            return false;  
  
    return true;  
}
```

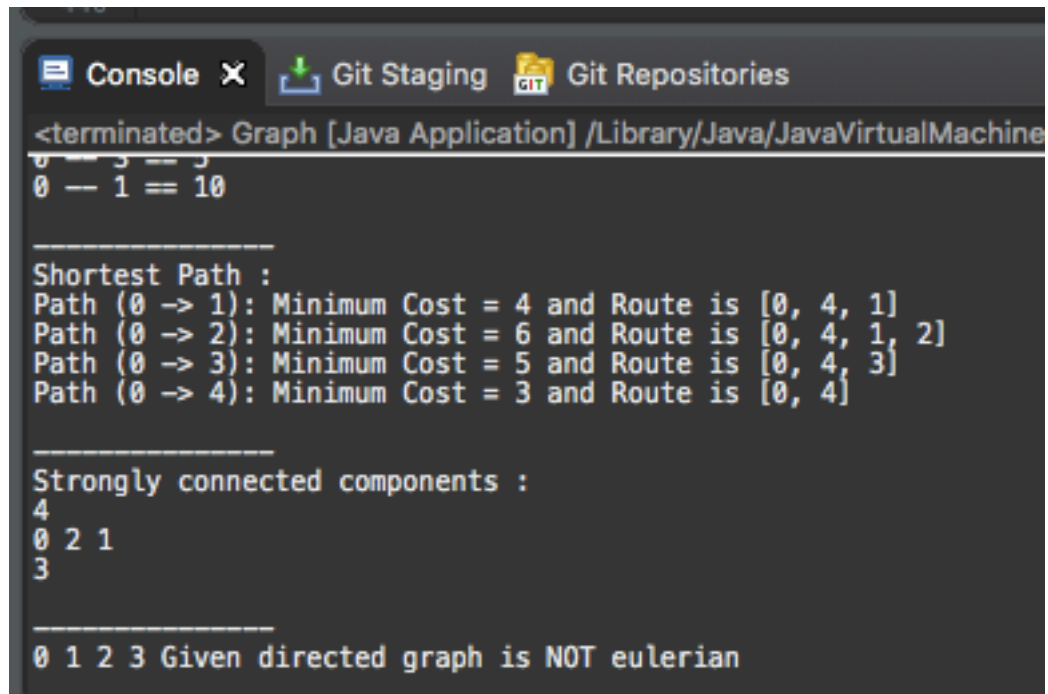
```
void DFSUtil(int v, boolean visited[]) {  
  
    visited[v] = true;  
    System.out.print(v + " ");  
  
    Iterator<Integer> i = adj[v].listIterator();  
    while (i.hasNext()) {  
        int n = i.next();  
        if (!visited[n])  
            DFSUtil(n, visited);  
    }  
}
```

```
public Graph getTranspose() {  
    Graph g = new Graph(V);  
    for (int v = 0; v < V; v++) {  
        Iterator<Integer> i = adj[v].listIterator();  
        while (i.hasNext())  
            g.adj[i.next()].add(v);  
        (g.in[v])++;  
    }  
    return g;  
}
```

In “public static void main(String args[])” method;

```
Graph g = new Graph(5);  
  
g.addEdge(0, 1);  
g.addEdge(0, 2);  
g.addEdge(1, 2);  
g.addEdge(2, 0);  
g.addEdge(2, 3);  
g.addEdge(3, 3);
```

```
if (g.isEulerianCycle())
    System.out.println("Given directed graph is eulerian ");
else
    System.out.println("Given directed graph is NOT eulerian ");
```



```
<terminated> Graph [Java Application] /Library/Java/JavaVirtualMachine
0 == 3 == 3
0 -- 1 == 10

-----
Shortest Path :
Path (0 -> 1): Minimum Cost = 4 and Route is [0, 4, 1]
Path (0 -> 2): Minimum Cost = 6 and Route is [0, 4, 1, 2]
Path (0 -> 3): Minimum Cost = 5 and Route is [0, 4, 3]
Path (0 -> 4): Minimum Cost = 3 and Route is [0, 4]

-----
Strongly connected components :
4
0 2 1
3

-----
0 1 2 3 Given directed graph is NOT eulerian
```