**Representation and important applications**

**of graph theory**

**Third Year- Computer Science**

**Spring Semester**

**Submitted to:**

Dr. Mostafa Reda El-Tantawi

**Team members:**

|  |  |
| --- | --- |
| **Name** | **ID** |
| Ahmed Sayed Mansour | 20170022 |
| Ashraf Samir Ali | 20170053 |
| Abdelraman Bahig | 20170143 |
| Atef Magdy | 20170136 |
| Hatem Sayed | 20170084 |
| Ahmed Sayed Ebrahim | 20170021 |
| Mohamed Mostafa | 20170257 |

* **Minimum Spanning Tree problem:-**

Abstraction:

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

Analysis:

we compared between The two algorithms which are used in minimum spanning tree and find that Both **Prim’s and Kruskal’s algorithm** finds the Minimum Spanning Tree and follow the Greedy approach of problem-solving, but there are few **major differences between them**.

|  |  |
| --- | --- |
| PRIM’S ALGORITHM | KRUSKAL’S ALGORITHM |
| It starts to build the Minimum Spanning Tree from any vertex in the graph. | It starts to build the Minimum Spanning Tree from the vertex carrying minimum weight in the graph. |
| It traverses one node more than one time to get the minimum distance. | It traverses one node only once. |
| Prim’s algorithm has a time complexity of O(V^2), V being the number of vertices. | Kruskal’s algorithm’s time complexity is O(logV), V being the number of vertices. |

So, we chose Kruskal Algorithm Based on his Time Complexity

Using Kruskal Algorithm:

Kruskal Implementation Pseudocode

KRUSKAL(G):

A = ∅

For each vertex v ∈ G.V:

MAKE-SET(v)

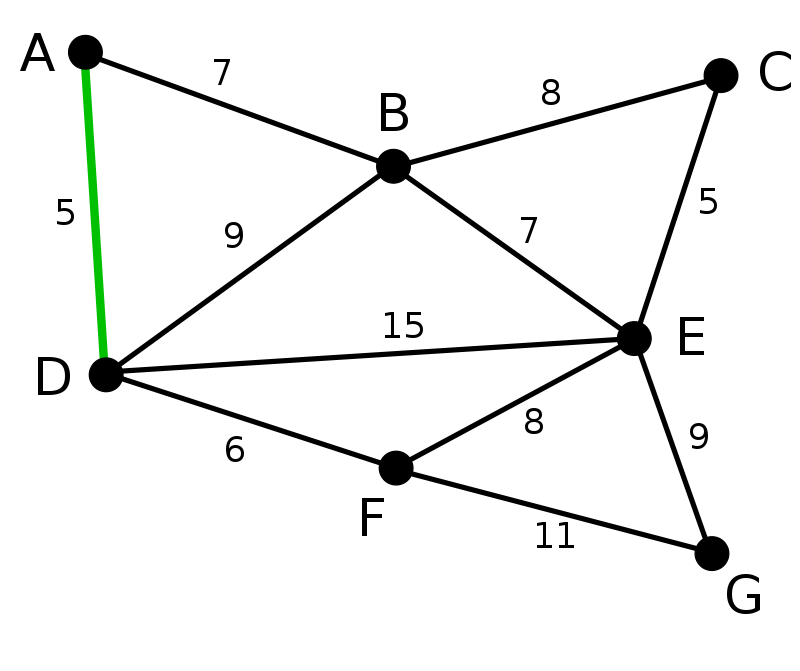
For each edge (u, v) ∈ G.E ordered by increasing order by weight (u, v):

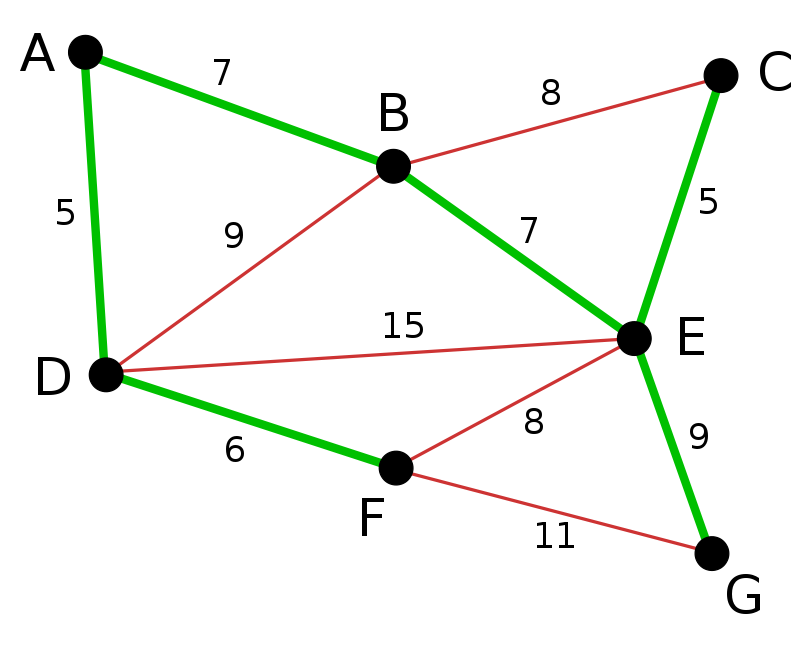
if FIND-SET(u) ≠ FIND-SET(v):

A = A ∪ {(u, v)}

UNION (u, v)

return A

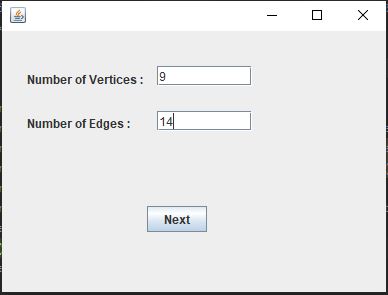




The Program (Run)

Input:

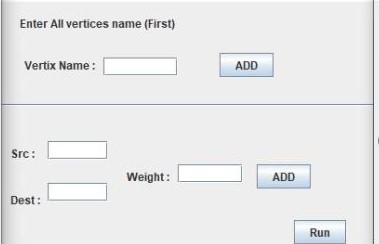
At the first should enter the number of Vertices and number of Edges you want (e.g. 4, 5)

****

Then Name your Vertices (e.g. 0, 1, 2, 3)



After that you can enter the source edge, destination edge and Weight (e.g. from 0 to 1, weight is 10). Do this as many times as the number of the Edges.



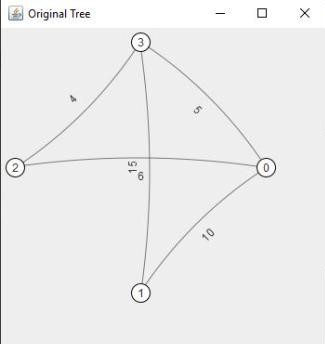
Note:

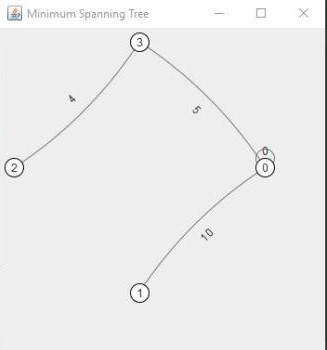
**Must fill the Vertices names with the number of Vertices you entered. You can’t enter the Edges unless you enter all Vertices name.**

Output:

Two Screens:

|  |  |
| --- | --- |
| Frist Screen  Original Graph | Second Screen  Minimum Spanning Tree |





Conclusion:

- Minimum Spanning tree used to reduce cost, time, weight etc. By finding the shortest path between any two nodes.

- Kruskal’s algorithm is an “Edge based algorithm “.

- Kruskal’s algorithm is a good method to find The Minimum Spanning Tree according to its Time Complexity O(logV).

References:

<https://www.ics.uci.edu/~eppstein/161/960206.html>

<https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/>

<https://www.javatpoint.com/kruskals-minimum-spanning-tree-algorithm>

* **Graph Representation :-**

**Abstraction:**

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.

Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, locale etc.

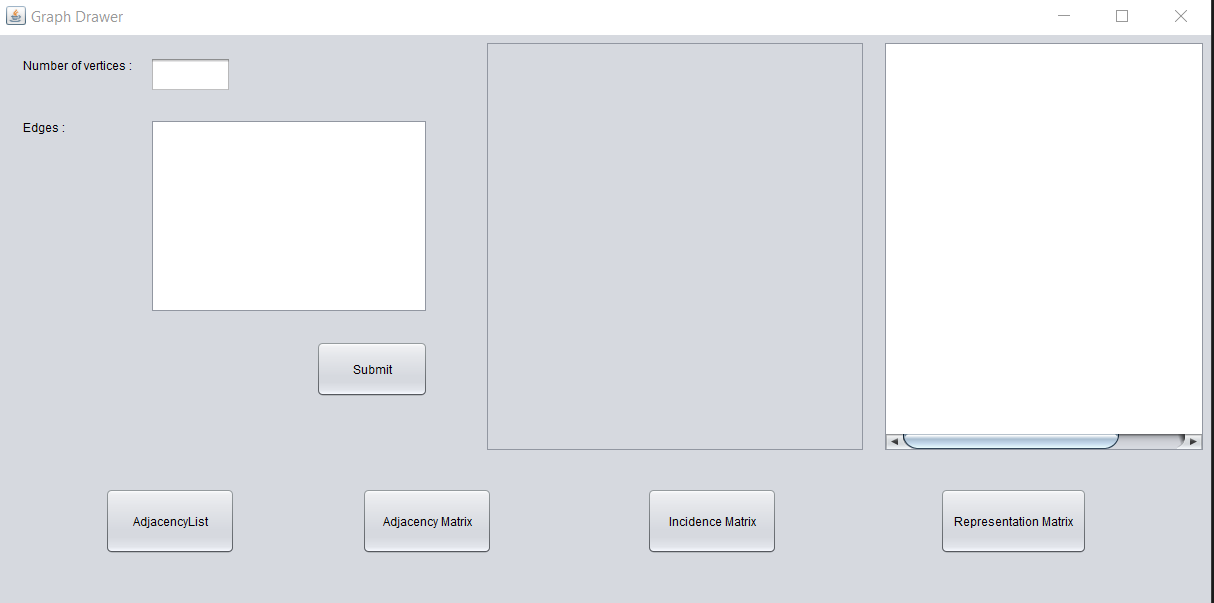
***How to use the program:***

4

3

2

1



*1. Enter number of vertices*

*2. Enter edges edge by edge*

*Edge input: Vertex ”space” Vertex  
Ex : 1 2*

*2 3*

*3. Click submit*

*4. Output appears.*

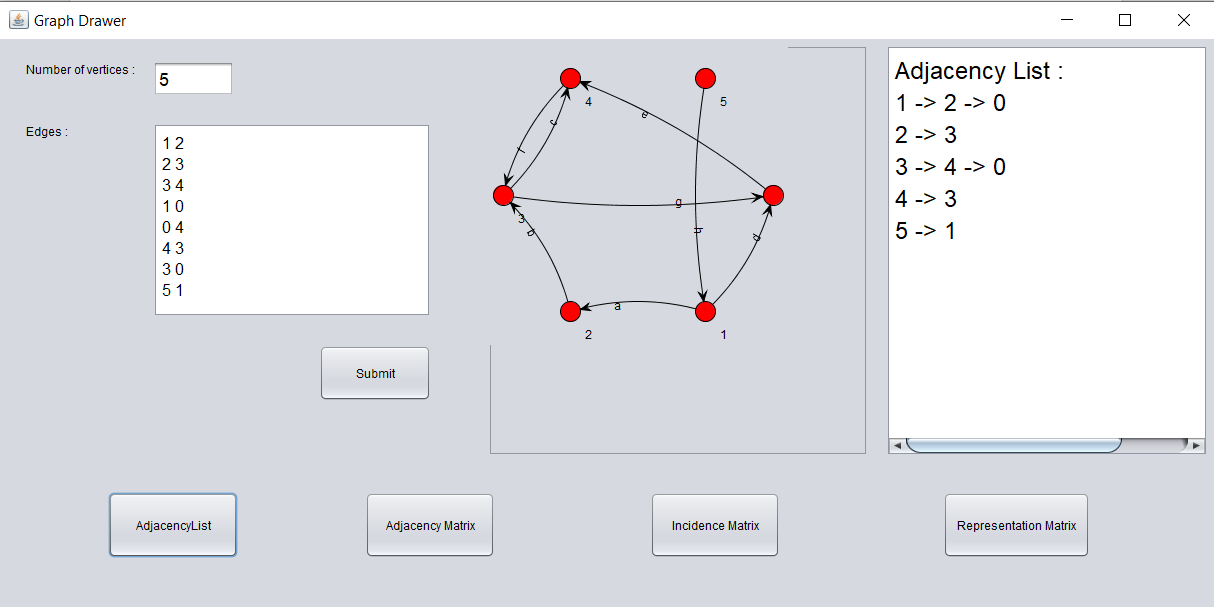
***Program in run:***

1

2

4

3



*1 - Adjacency List*

*2 - Adjacency Matrix*

*3 - Incidence Matrix*

*4 - Representation Matrix*

***Pseudo code for my program:***

*Adjacency List:*

Procedure Adjacency-List(maxN, E): // maxN denotes the maximum number of nodes

edge[maxN] = Vector() // E denotes the number of edges

for i from 1 to E

input -> x, y // Here x, y denotes there is an edge between x, y

edge[x].push(y)

edge[y].push(x)

end for

Return edge

*Adjacency Matrix:*

Procedure AdjacencyMatrix(N): //N represents the number of nodes

Matrix[N][N]

for i from 1 to N

for j from 1 to N

Take input -> Matrix[i][j]

endfor

endfor

*Incidence matrix:*

Begin

ed\_cnt := ed\_cnt + 1

inc\_matrix[u, ed\_cnt] := 1

inc\_matrix[v, ed\_cnt] := 1

End

References:

<https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/>

<https://www.javatpoint.com/graph-and-graph-theory-introduction>

* **Coloring Graph problem:-**

***Introduction:***

**The Color-Graph** program target is to represent the (n) vertices entered by user, then connect them with edges labeled alphabetically, after connection the program will color the vertices with random RBG colors every submit. The main target of the program is to color the adjacent vertices (connected by edge) with **different colors**!

(ex: vertex 5 and vertex 4 are blue and purple, not both blue or both purple).

A screenshot of a map

Description automatically generated- The image below shows how the adjacent connected vertices are colored differently.

Image revealing the Edges picked by user and the vertices coloring process

***Analysis:***

In order to achieve the different colors for adjacent nodes, adjacency list is used for the graph, which facilitates determining which nodes are adjacent and which are not connected.

***Implementation and Design:***

**Implementation**: The program is implemented by Java.

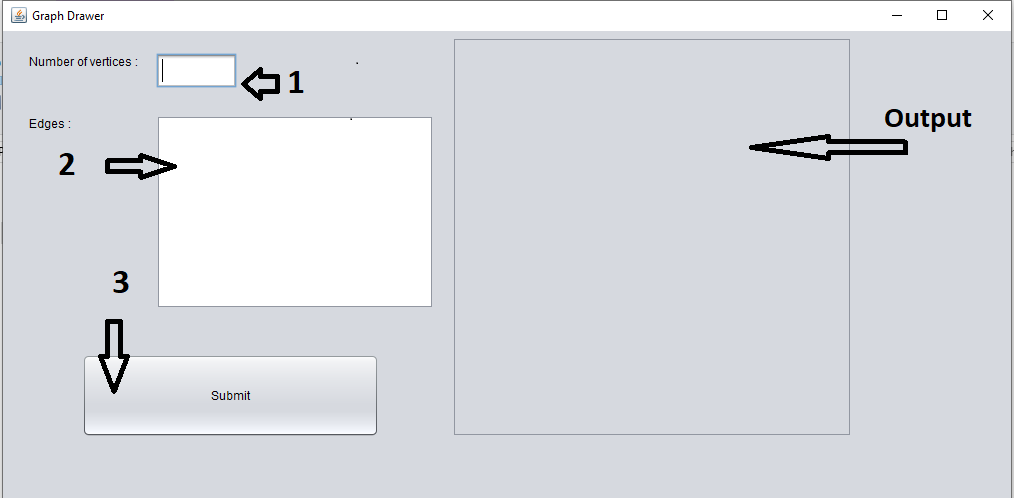
Adjacency list was used in class graph, which will help in coloring the nodes as if two nodes are adjacent, they get different colors.

**Design**: The program is designed by java swing library which is a set of program component to give the Graphical User Interface. For example (clicks, buttons and input text).

**Graph Implementation**: For the Graph implementation, JUNG library is used to implement the graph.

JUNG — the Java Universal Network/Graph Framework--is a software library that provides modeling, analysis, and visualization of data that can be represented as a graph or network, which in this case its benefits are drawing the graph network.

***How to use the program:***

******

**Input:**

*1. Enter number of vertices (Example : 5)*

*2. Enter edges, edge by edge*

*Edge input: Vertex ”space” Vertex*

Ex : 1 2

2 3

Note:

3 4

4 5

5 1

**If you used a non-existing node while entering the edges,**

**(Example: number of vertices = 5 and edge (1 , 6) )**

**The program will show no result).**

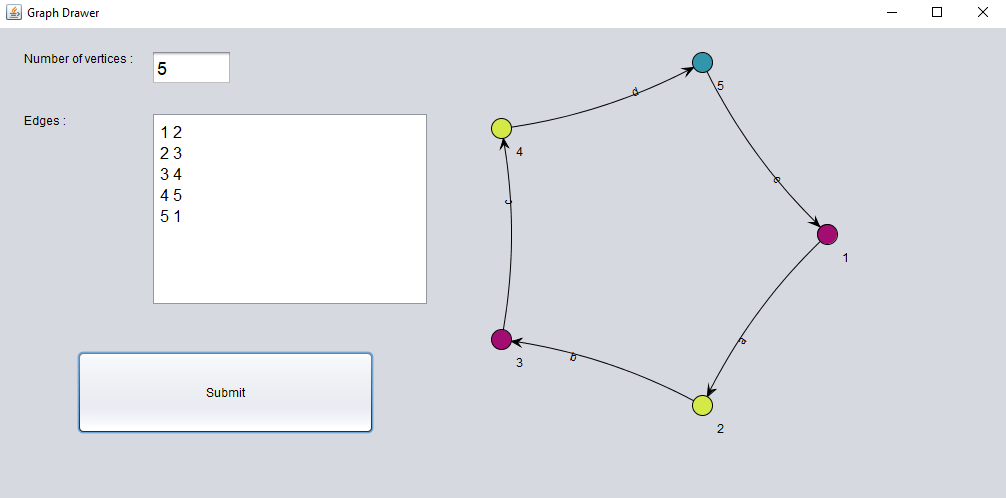
*3. Click submit*

**Output:**

- Number of vertices represented in the window (Example: 1,2,3,4,5)

- Edges connecting all the vertices labeled alphabetically (Example: a,b,c,d,e).

- Vertices are colored randomly, no two vertices connected adjacently have the same color. (Example: node 2 and node 3 are connected by an edge, thus their colors are different: yellow and violet)

*The input and output flows are shown in the picture below:*

This image shows the program draw after submitting the inputs

Pseudo code for the program:

1) Input number of Vertices  
2) Input Edges

3) Color first vertex with first color.

4) Do following for remaining V-1 vertices.

5) Consider the currently picked vertex and color it with the lowest numbered color that has not been used on any previously colored vertices adjacent to it.

6) If all previously used colors appear on vertices adjacent to v,

6.1) assign a new color to it.

7) Output the graph.

Conclusion:

The adjacency list used in the graph class to indicate which node is adjacent to other nodes proved to be efficient and easy for coloring the nodes.

References:

<http://jung.sourceforge.net/doc/index.html>

<https://www.javatpoint.com/java-swing>

<https://www.vainolo.com/2011/02/14/learning-jung-java-universal-networkgraph-framework/>

* **Euler Path problem:-**

***Introduction:***

**The Euler path** In graph theory, an  **Eulerian path** is a trail in a finite graph that visits every edge exactly once (allowing for revisiting vertices). Similarly, an **Eulerian circuit** or **Eulerian cycle** is an Eulerian trail that starts and ends on the same vertex. They were first discussed by [Leonhard Euler](https://en.wikipedia.org/wiki/Leonhard_Euler) while solving the famous [Seven Bridges of Königsberg](https://en.wikipedia.org/wiki/Seven_Bridges_of_K%C3%B6nigsberg) problem in 1736. The problem can be stated mathematically like this:

***Analysis:***

In order to achieve the Euler path for adjacent nodes, adjacency list is used for the graph, which facilitates determining which nodes are adjacent and which are not connected.

***Implementation and Design:***

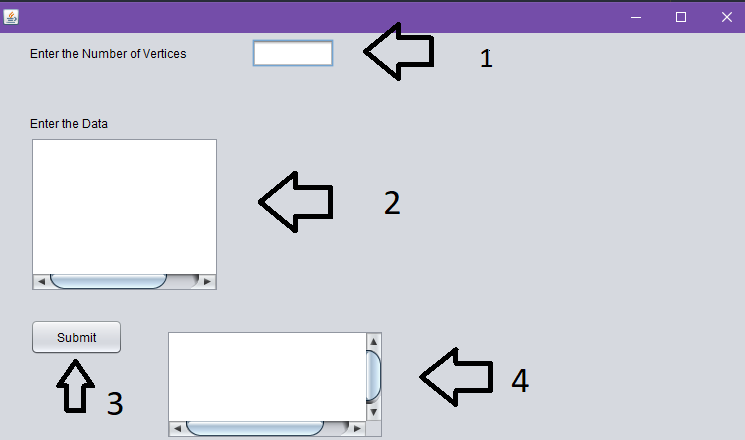
**Implementation**: The program is implemented by Java.

Adjacency list was used in class graph, which will help in detecting the euler path.

**Design** : The program is designed by java swing library which is a set of program component to give the Graphical User Interface. For example (clicks, buttons and input text).

**Graph Implementation**: For the Graph implementation, JUNG library is used to implement the graph.

JUNG — the Java Universal Network/Graph Framework--is a software library that provides modeling, analysis, and visualization of data that can be represented as a graph or network, which in this case its benefits are drawing the graph network.

*How to use the program:  
*

*1. Enter number of vertices*

*2. Enter edges edge by edge (Representation Matrix)*

*Ex : vertices number = 2*

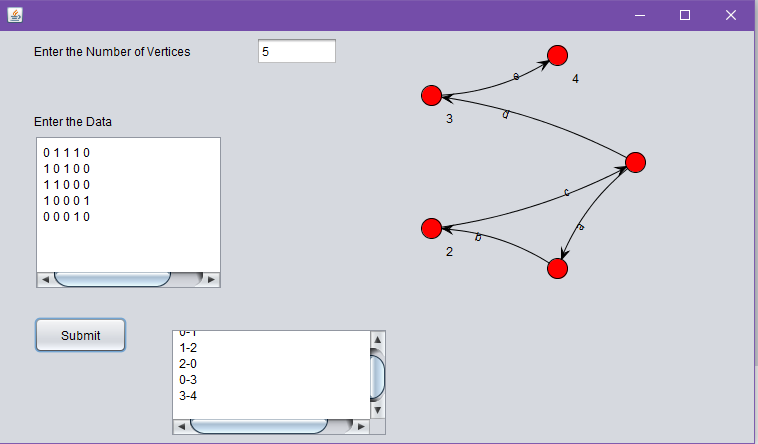
*1 0*

*0 1*

*3. Click submit*

*4. Euler Path.*

*Program in run:*

**

***Pseudo code for my program:***

*->Detect Euler path,Circuit*

Begin

   if isConnected() is false, then

      return false

   define list of degree for each node

   oddDegree := 0

   for all vertex i in the graph, do

      for all vertex j which are connected with i, do

         increase degree

      done

      if degree of vertex i is odd, then

         increase dooDegree

   done

   if oddDegree > 2, then

      return 0

   if oddDegree = 0, then

      return 2

   else

      return 1

End

Conclusion:

The adjacency list used in the graph class to indicate which node is adjacent to other nodes proved to be efficient and easy Euler path detection.

References:

<http://jung.sourceforge.net/doc/index.html>

<https://www.javatpoint.com/java-swing>

<https://www.vainolo.com/2011/02/14/learning-jung-java-universal-networkgraph-framework/>

* **Sales Man problem:-**

**Why – What**

The main reason to make this program is to solve the hamilton circuit problem in java in a very effiecient solution, the algorithms is one of the famous greedy algorithms that solve the sales man problem to give the minimal path to visit all towns from a town.

**Hamiltonian Path** in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in the graph) from the last vertex to the first vertex of the Hamiltonian Path. Determine whether a given graph contains Hamiltonian Cycle or not. If it contains, then prints the path. Following are the input and output of the required function.

**How**

This is a Backtracking Algorithm that try all possible solution and give the optimal one and output the path to the user.

Create an empty path array and add vertex 0 to it. Add other vertices, starting from the vertex 1. Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added. If we find such a vertex, we add the vertex as part of the solution. If we do not find a vertex then we return false.

This is the main screen of my program where the user has the ability to freely enter the adjacency matrix of the roads linking the cities with each other, taking in consideration entering the number of Vetricies aka towns.

**Where and When**

the output is a small screen like this and the user can see the number on each **red node** the weight of each path is also given on each directed arrow (Verticies) .

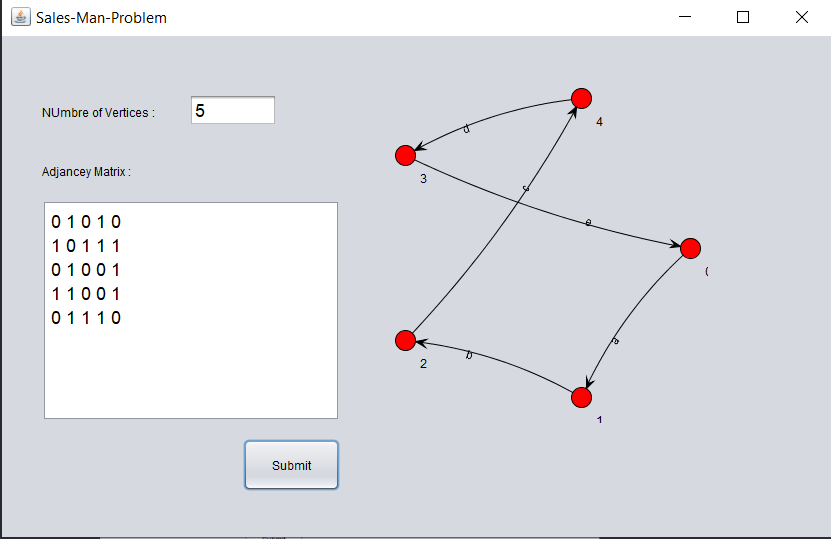
**Conclusion**

The output is really clear and precise using Java GUI Swing that is friendly to any user Who want to try this algorithm without Any back knowledge in programming.

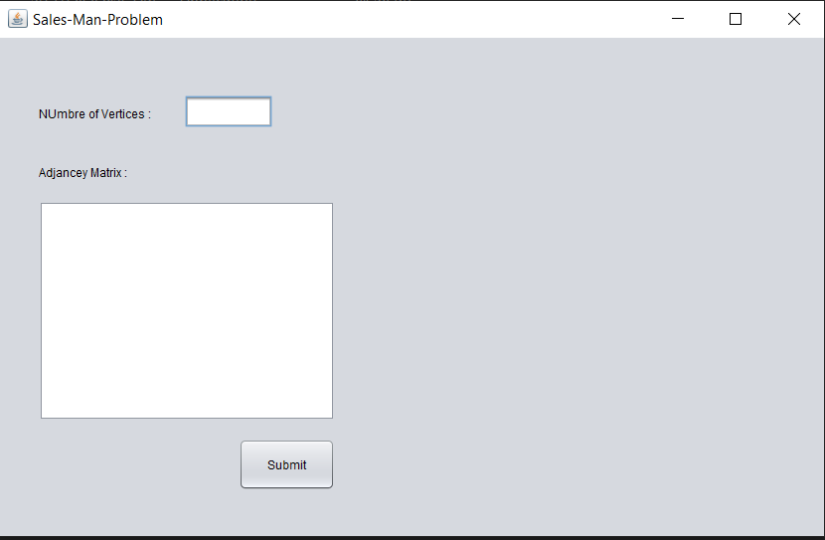
When the user enters the towns and the edges details, the user should then press submit to produce the input regarding to his problem the output should follows

**Testing**

We tested the application ourselves in various black box test cases and we made sure that the output is right each test case we entered into the program



**Screenshot of the Output**



**Screenshot of the Input**

This program was mainly written in java but here is the pseudo code you can implement: -

1. Consider city 1 as the starting and ending point. Since route is cyclic, we can consider any point as starting point.
2. Generate all (n-1)! permutations of cities.
3. Calculate cost of every permutation and keep track of minimum cost permutation.
4. Return the permutation with minimum cost.

**Maintenance**

We found that the program needs some maintenance to make sure that is running good on each device beside the problem that **JAR** is not working we encountered the enhancements we made and amends we did was really huge in the source code because in the first time we ran this program to machine we found that the code is not printing the right solutions to some test case, so we fixed this using refactoring and debugging to run the program right at the final moment.

**References**

https://en.wikipedia.org/wiki/Travelling\_salesman\_problem

<https://www.baeldung.com/java-simulated-annealing-for-traveling-salesman>

<https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/>

* **Hamilton problem:-**

Abstraction:

Hamilton Path & Cycle: Given In the [mathematical](https://en.wikipedia.org/wiki/Mathematics) field of [graph theory](https://en.wikipedia.org/wiki/Graph_theory), a **Hamiltonian path** (or **traceable path**) is a [path](https://en.wikipedia.org/wiki/Path_(graph_theory)) in an undirected or directed graph that visits each [vertex](https://en.wikipedia.org/wiki/Vertex_(graph_theory)) exactly once. A **Hamiltonian cycle** (or **Hamiltonian circuit**) is a Hamiltonian path that is a [cycle](https://en.wikipedia.org/wiki/Cycle_(graph_theory)). Determining whether such paths and cycles exist in graphs is the [Hamiltonian path problem](https://en.wikipedia.org/wiki/Hamiltonian_path_problem), which is [NP-complete](https://en.wikipedia.org/wiki/NP-complete_problem).

Analysis:

**Backtracking Algorithm**  
Create an empty path array and add vertex 0 to it. Add other vertices, starting from the vertex 1. Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added. If we find such a vertex, we add the vertex as part of the solution. If we do not find a vertex then we return false.

Node emptyPath;

emptyPath[0] = vertex0;

while(thereIs(vertex)){

if((adjacent to the previously added vertex)&(!added before)){

emptyPath.add(vertex); //emptyPath is the solution

}

Else{

Return false;

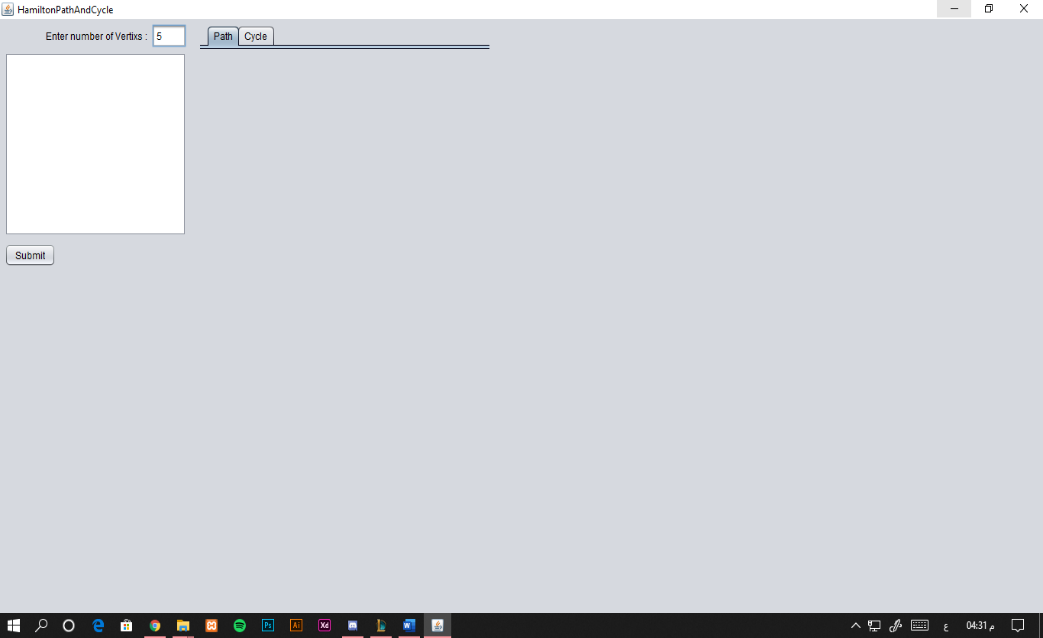
}

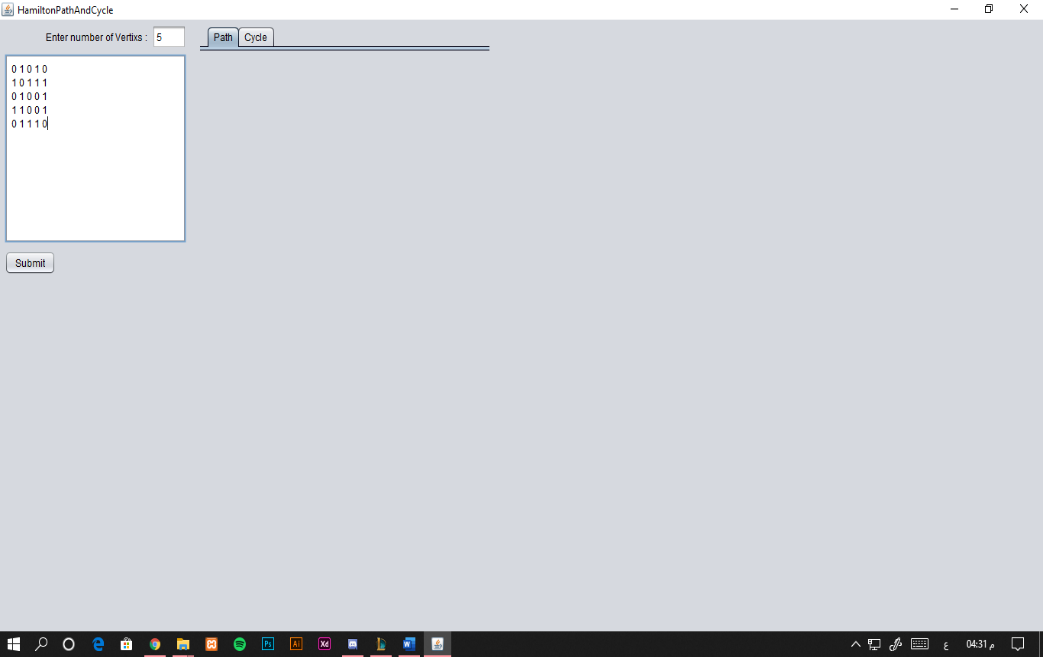
}

The Program (Run)

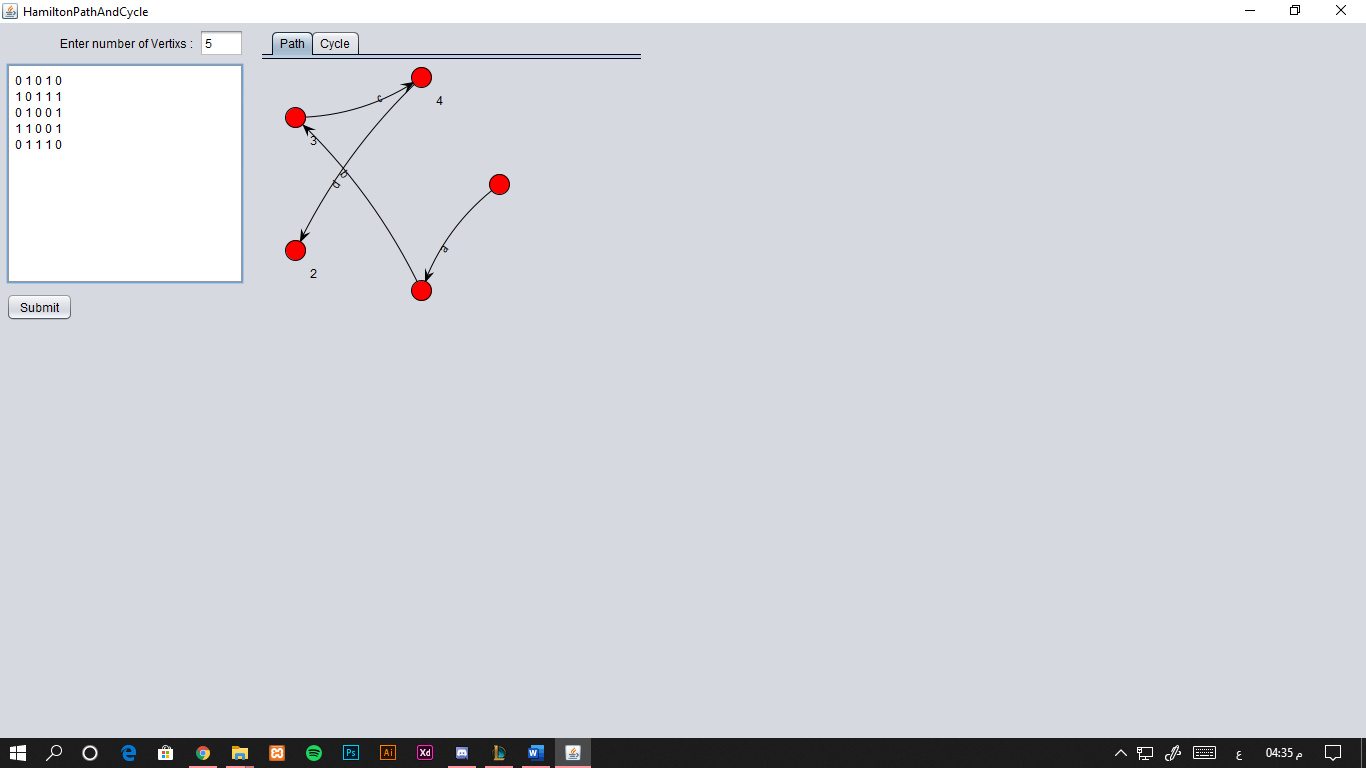
Input:

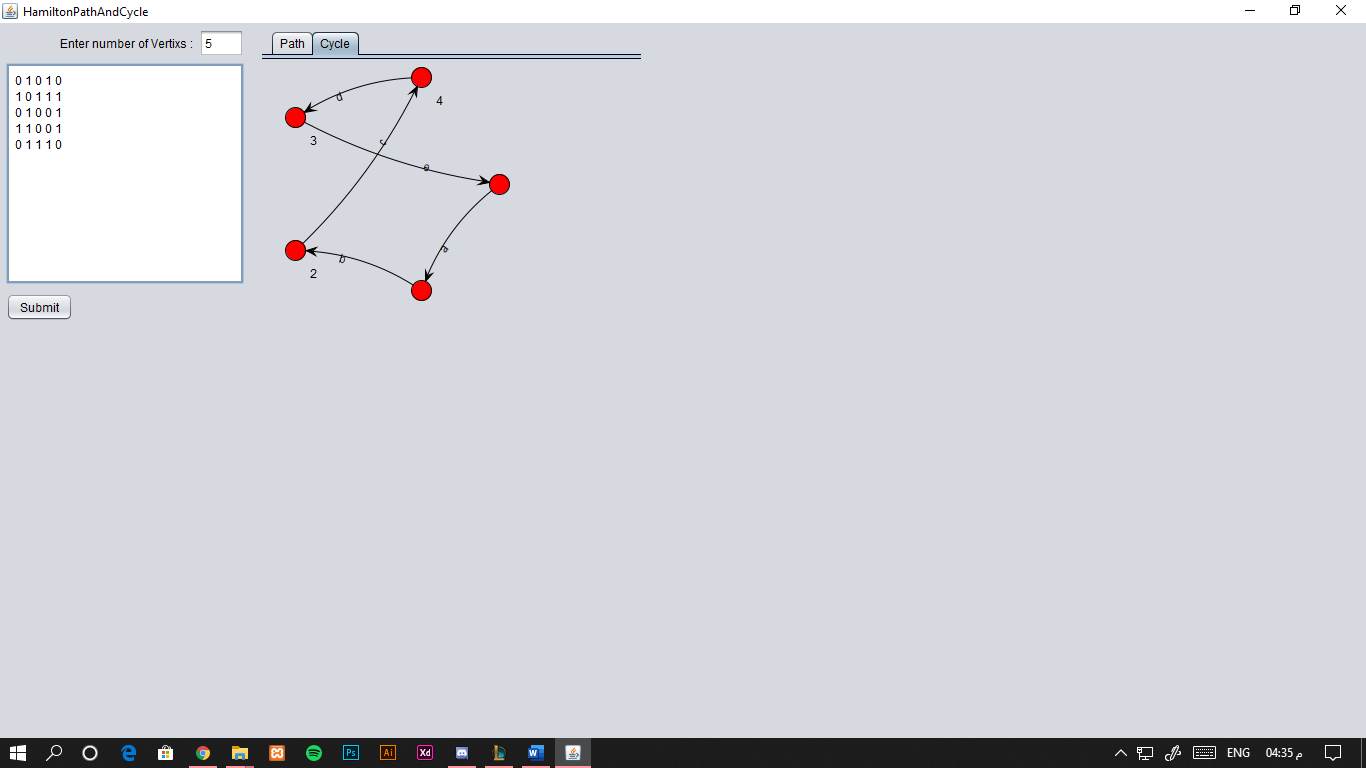
*Input:*  
A 2D array graph[V][V] where V is the number of vertices in graph and graph[V][V] is adjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edge from i to j, otherwise graph[i][j] is 0.





Output: *An array path[V] that should contain the Hamiltonian Path. path[i] should represent the ith vertex in the Hamiltonian Path. The code should also return false if there is no Hamiltonian Cycle in the graph.*





Conclusion:

- Hamiltonian path is a [path](https://en.wikipedia.org/wiki/Path_(graph_theory)) in an undirected or directed graph that visits each [vertex](https://en.wikipedia.org/wiki/Vertex_(graph_theory)) exactly once.

Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian path that is a [cycle](https://en.wikipedia.org/wiki/Cycle_(graph_theory)). Determining whether such paths and cycles exist in graphs is the [Hamiltonian path problem](https://en.wikipedia.org/wiki/Hamiltonian_path_problem)

 which is [NP-complete](https://en.wikipedia.org/wiki/NP-complete_problem).

References:

<https://en.wikipedia.org/wiki/Hamiltonian_path>

<https://www.geeksforgeeks.org/hamiltonian-cycle-backtracking-6/>

<https://www.geeksforgeeks.org/hamiltonian-cycle-backtracking-6/>