**REPRESENTATION AND IMPORTANT APPLICATIONS**

**OF GRAPH THEORY**

# Table of Contents

[**Table of Contents**](#_heading=h.jj11qhvcn1lt) **3**

[**Hamilton Path**](#_heading=h.bb611i8unah2) **5**

[Overview](#_heading=h.4eh3eyatv96z) 5

[Design](#_heading=h.5fhu53p2pq1p) 5

[Analysis](#_heading=h.drtm3zummmfk) 6

[Test](#_heading=h.mz153argvpko) 6

[Applications](#_heading=h.jzsgi6t8e466) 6

[**Hamilton circuit**](#_heading=h.sifc9o7stl2a) **7**

[Overview](#_heading=h.5f3r4nkwl0ai) 7

[Design](#_heading=h.ynrd89eeberh) 7

[Analysis](#_heading=h.svaepagttam1) 8

[Test](#_heading=h.cq3qm5jzv24r) 8

[Applications](#_heading=h.fvwcaetuyjom) 8

[**Minimum Humiliation path**](#_heading=h.hg8aqhqk23si) **9**

[Overview](#_heading=h.dh964cju6d8d) 9

[Design](#_heading=h.r03j0g4xex2c) 9

[Analysis](#_heading=h.uqcqk5f46q21) 9

[Applications](#_heading=h.8ixrs7grpafb) 9

[Test](#_heading=h.rry7d2atyrgq) 10

[**Euler Path**](#_heading=h.hrk1bxpn0ts5) **11**

[Overview](#_heading=h.baw6xjbpvwr) 11

[Design](#_heading=h.cmzfn0vge8gk) 11

[Analysis](#_heading=h.r9fvt36dnzsu) 12

[Applications](#_heading=h.hjht2z9p98ti) 12

[Test](#_heading=h.33p9z0ypycd8) 12

[**Euler Circuit**](#_heading=h.3uk3q1cqnl6z) **13**

[Overview](#_heading=h.ezo4dahn8w1g) 13

[Design](#_heading=h.z8jdqx8jzuf3) 13

[Analysis](#_heading=h.gcfox9f118h6) 14

[Applications](#_heading=h.hjht2z9p98ti) 14

[Test](#_heading=h.wytdjkuihov0) 14

[**Map Graph Coloring**](#_heading=h.b4b0kfdgoolj) **15**

[Overview](#_heading=h.ay2shp5xx39a) 15

[Design](#_heading=h.dq8bk54poety) 15

[Analysis](#_heading=h.2muoz91h2e35) 16

[Applications](#_heading=h.nwu7n6oypzpe) 17

[Test](#_heading=h.dusvl2yzpy03) 17

[**Minimum Spanning Tree**](#_heading=h.nuzfwhypvk5) **18**

[Overview](#_heading=h.i9k3maawet12) 18

[Design](#_heading=h.i9k3maawet12) 18

[Analysis](#_heading=h.g8o3xutfltih) 18

[Applications](#_heading=h.qcuajvv0468s) 19

[Test](#_heading=h.ciwjx3oqmvi5) 19

[**How to Use**](#_heading=h.7v0s74yrmy3v) **20**

# 

# Hamilton Path

## Overview

In graph theory, the Hamilton path of a connected directed/undirected graph is the path that visits each vertex exactly once and finally all vertices of the graph will be visited.

## Design

**Input:** Connected undirected/directed graph.

**Output:**

Sequence of vertices such that if you traverse, each vertex will be visited exactly one.

**Pseudocode:**

### 

## 

## Analysis

No efficient method is known for testing if a graph contains a Hamiltonian path, with good analysis of the problem we may determine if the graph contains hamiltonian path or not, A simple observation is that if the graph is complete, it also contains a Hamiltonian path.

**Dirac’s theorem:** If the degree of each node is at least n/2, the graph contains a Hamiltonian path.

**Ore’s theorem:** If the sum of degrees of each non-adjacent pair of nodes is at least n, the graph contains a Hamiltonian path.

The Algorithm is bruteforce, try all possible paths and make check for every path if it is Hamiltonian path or not.

Time Complexity: O(N!), factorial time complexity

Space Complexity O(N) , linear space complexity

## Test

## Applications

When mapping genomes scientists must combine many tiny fragments of genetic code (“reads”, they are called), into one single genomic sequence (a ‘superstring’). This can be done by finding a Hamiltonian path.

# Hamilton circuit

## Overview

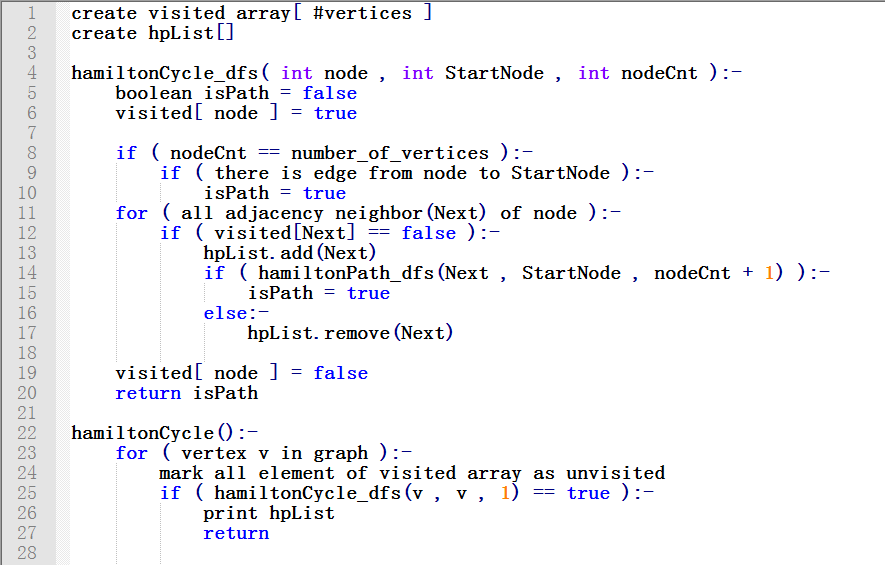
In graph theory, the Hamilton circuit of a connected directed/undirected graph is the path that visits each vertex exactly once and finally all vertices of the graph will be visited and the end vertex is the same as the start vertex.

## Design

**Input:** Connected undirected/directed graph.

**Output:**

Sequence of vertices such that if you traverse, each vertex will be visited exactly one.

**Pseudocode:**

### 

## Analysis

Same as hamiltonian path,No efficient method is known for testing if a graph contains a Hamiltonian cycle.

The Algorithm is bruteforce, try all possible paths and make check for every path if it is Hamiltonian cycle or not.

Time Complexity: O(N!), factorial time complexity

Space Complexity O(N) , linear space complexity

## Test

## Applications

Hamiltonian cycles to plan the best route to pick up students from across the district. Here students may be considered nodes, the paths between them edges, and the bus wishes to travel a route that will pass each students house exactly once.

# 

# Minimum Humiliation path

## Overview

Minimum Humiliation path is a path of a connected weighted edges graph where all the vertices are connected together and each node in the path is visited only once with the minimum possible total edge weight.

## Design

**Input**:   
 1- Number of vertices   
 2- Number of edges  
 3- Source Vertex  
 4- Destination vertex.

**Output**: minimum cost that we can travel each node in the graph only once by it  
  
  **Pseudo code:**  
 for s = 2 to n do

for all subsets S Є {1, 2, 3, … , n} of size s and containing 1

C (S, 1) = ∞

for all j Є S and j ≠ 1

C (S, j) = min {C (S – {j}, i) + d(i, j) for i Є S and i ≠ j}

Return minj C ({1, 2, 3, …, n}, j) + d(j, i)

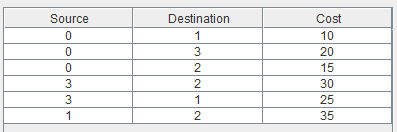
## Analysis

**Time complexity** : O(n ^ 2) .

## Applications

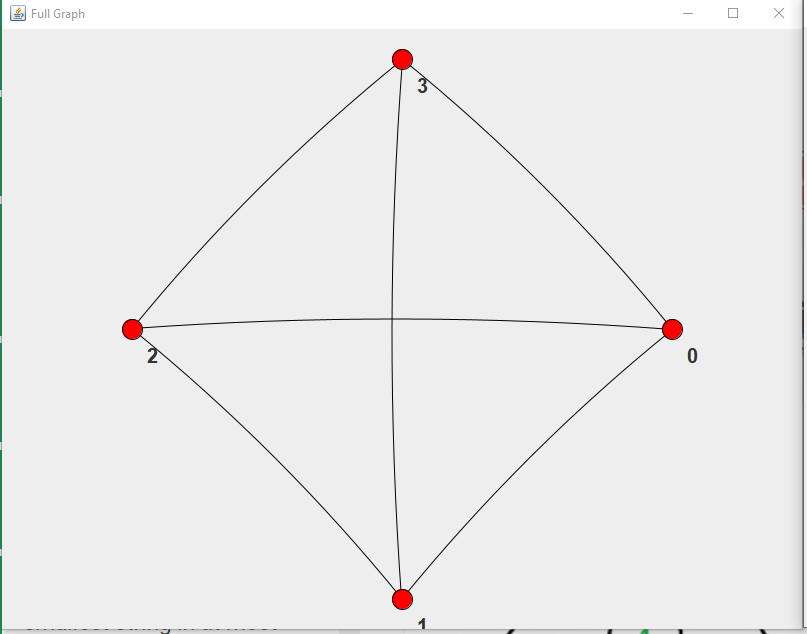
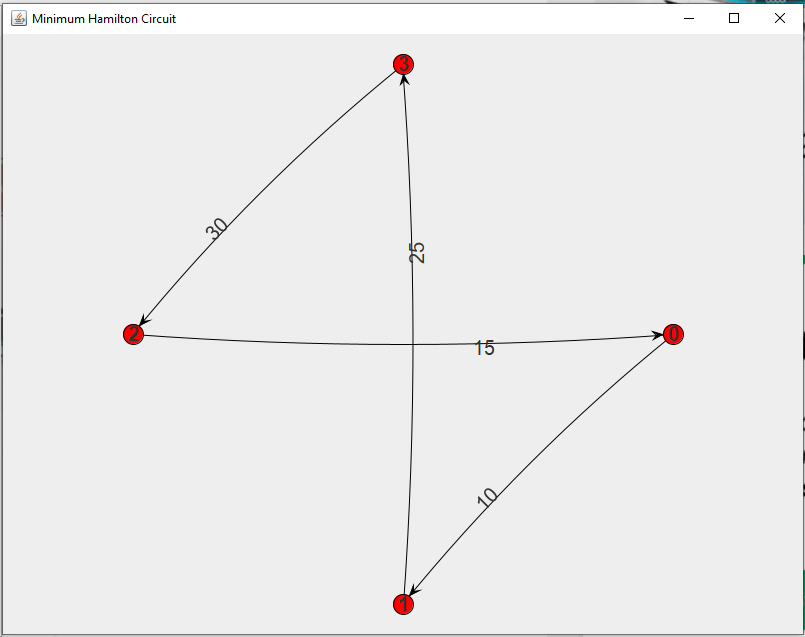
Used in companies that want to know minimum cost that want to distribute their products across different local markets or across cities.

## Test

Edges of the graph   
 

The original full graph is Figure A and the Minimum Hamilton paths generated is Figure B and its cost is 80.

**Figure A Figure B**

# 

# 

# Euler Path

## Overview

An Euler path is a path that uses every edge of a graph exactly once without leaving any edge unvisited.

**Theorem**: A graph will contain an Euler path if it contains at most two vertices of odd degree. If the graph has an odd degree, we must start from it.

## Design

**Input:** Connected undirected graph (edges weights don’t matter).

**Output:** Connected graph directed to show the Euler path.

**Pseudo Code:**

1. n = numberOfVertices
2. m = numberOfEdges
3. if (isConnected() == false)
   1. return null
4. odd = 0
5. start = 1
6. for (i from 1 to n)
   1. if (degree(i)%2 == 1)
      1. increment odd;
      2. start = i
7. if (odd > 2)
   1. return null
8. adj\_List[n+1][n+1]
9. for (i from 1 to n )
   1. neighboursList = getAllNeighboursOf(i).
   2. for(j from 0 to neighboursList.length())
      1. increment adj\_List[i][j]
10. numEdges = 0
11. while (numEdges != m)
    1. neighboursList = getAllNeighboursOf(i).
    2. for(j from 0 to neighboursList.length())
       1. node = neighbourList[j]
       2. if (adj\_List[start][node] != 0)
          1. if the graph has cycle continue
          2. euler.addEdge(start, neighbour)
          3. adj\_List[start][neighbour] = 0
          4. adj\_List[neighbour][start] = 0
          5. start = neighbour

## Analysis

E = edges.

V = vertices.

**Time Complexity:** O(E!)

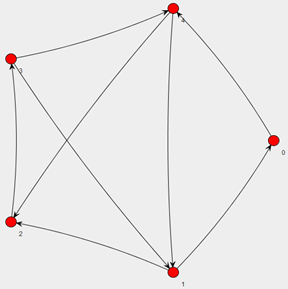
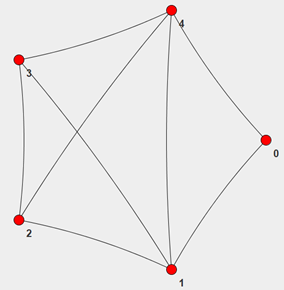
**Space Complexity:** O(V\*V + E)

## Applications

Eulerian trails are used in [bioinformatics](https://en.wikipedia.org/wiki/Bioinformatics) to reconstruct the [DNA sequence](https://en.wikipedia.org/wiki/DNA_sequence) from its fragments.They are also used in [CMOS](https://en.wikipedia.org/wiki/CMOS) circuit design to find an optimal [logic gate](https://en.wikipedia.org/wiki/Logic_gate) ordering.

## Test

Suppose we have this undirected graph Figure A so we can get the Euler path Figure B.

**Figure A Figure B**

# Euler Circuit

## Overview

An Euler circuit is a circuit that uses every edge of a graph exactly once, starts and ends at the same vertex.

**Theorem**: A graph will contain an Euler circuit if all vertices have even degree. If the graph has an Euler circuit then it has an Euler path.

## Design

**Input:** Connected undirected graph (edges weights don’t matter).

**Output:** Connected graph directed to show the Euler circuit.

**Pseudo Code:**

1. n = numberOfVertices
2. m = numberOfEdges
3. if (isConnected() == false)
   1. return null
4. for (i from 1 to n)
   1. if (degree(i)%2 == 1)
      1. return null;
5. if (odd > 2)
   1. return null
6. adj\_List[n+1][n+1]
7. for (i from 1 to n )
   1. neighboursList = getAllNeighboursOf(i).
   2. for(j from 0 to neighboursList.length())
      1. increment adj\_List[i][j]
8. numEdges = 0
9. start = 0
10. while (numEdges != m)
    1. neighboursList = getAllNeighboursOf(i).
    2. for(j from 0 to neighboursList.length())
       1. node = neighbourList[j]
       2. if (adj\_List[start][node] != 0)
          1. if the graph has cycle continue
          2. euler.addEdge(start, neighbour)
          3. adj\_List[start][neighbour] = 0
          4. adj\_List[neighbour][start] = 0
          5. start = neighbour

## Analysis

E = edges.

V = vertices.

**Time Complexity:** O(E!)

**Space Complexity:** O(V\*V + E)

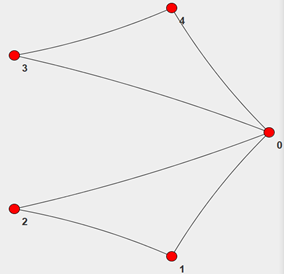
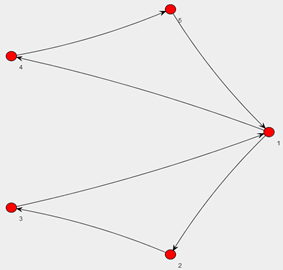
## Applications

Optimizing travel layouts for garbage pickup, mail service delivery and patrolling.

## Test

Suppose we have this undirected graph Figure A so we can get the Euler path Figure B.

We can’t get an Euler circuit from (Figure A) but we can get an Euler circuit from (Figure C) like (Figure D).

**Figure C Figure D**

# Map Graph Coloring

## Overview

Is coloring all nodes of a graph for a certain map such that there's no two adjacent nodes having the same color & according to the theorem the maximum number of colors needed for any map is four.

**Technical Feasibility:**

The Map Graph Coloring program is complete desktop based application.

**Main Technologies & tools associated with the program:-**

1-Java

2-GUI Libraries:

1-Java Swing

2-Jung

Each of the technologies is freely available and skills needed for the program scope are available and since using already built in libraries of java or others built by java; time limitation was synchronized.

**Conclusion**: clearly the program is technically feasible.

**Purpose:** Is solving the map graph coloring problem and visualize both the map graph & graph coloring solution.

## 

## Design

**Input:** Connected undirected graph.

**Output:** Connected undirected colored graph.

**Pseudo Code:**

result = []

fill the result array with -1

the first vertix in result take the color 0

boolean avail[]

fill avail with true

for each vertex

set the first color for the adjacent vertices with false

find the first available color

assign the found color

reset the values of avail back to true

## 

## Analysis

**Requirements:**

1- Program allows visualization of graphs from determining which nodes direct to which other nodes only.

2- Program supports two visualizations of the graph, one to graph before coloring & the other after coloring.

3-Program colors nodes in the colored visualization such that each node has a color different from colors of adjacent nodes to this node.

4-Program doesn't color nodes using more than 4 unique colors.

**Time Complexity:**

O(V^2 + E) in the worst case.

## Applications

Making a Schedule or Time Table: Suppose we want to make an exam schedule for a university. We have listed different subjects and students enrolled in every subject. Many subjects would have common students (of the same batch, some backlog students, etc). How do we schedule the exam so that no two exams with a common student are scheduled at same time? How many minimum time slots are needed to schedule all exams? This problem can be represented as a graph where every vertex is a subject and an edge between two vertices means there is a common student. So this is a graph coloring problem where the minimum number of time slots is equal to the chromatic number of the graph.

## Test

For Example: Figure 1 represents a map & Figure 2 is graph representation for the map (in which the 0, 1, 2, 3 & 4 are the corresponding nodes for the map nodes A, B, C, D & E in the graph) then figure 3 would be the colored graph of graph representation for figure 1 map that could be used then to color the map & it only needed three colors to for the graph (less than 4).

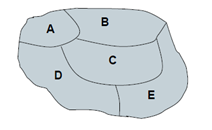
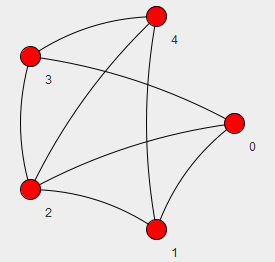
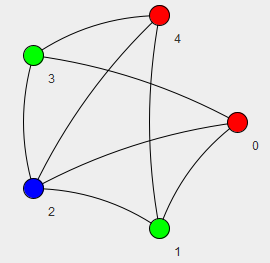
**  **

Figure 1 Figure2 Figure 3

# Minimum Spanning Tree

## Overview

Minimum spanning tree is a subset of a connected weighted edges graph where all the vertices are connected together, without any cycles and with the minimum possible total edge weight.The minimum spanning tree will always have a number of edges which is less than the number of vertices by 1, meaning Edges = Vertices - 1. One method of finding the minimum spanning tree is using Kruskal’s Algorithm which was discovered in 1956 by the American mathematician named Joseph Kruskal. It is a greedy algorithm that we used in this project to find the minimum spanning tree of any connected graph.

## Design

**Input**

Connected unidirectional graph weighted graph. A group of weighted edges connecting between vertices.

**Output**

Connected graph directed to show the minimum spanning tree and its cost.

**Pseudocode**

1. A = []
2. For each vertex in graph:
3. MAKE-SET(v)
4. For each edge (u, v) in graph ordered by increasing order by cost(u, v):
5. if FIND-SET(u) ≠ FIND-SET(v):

1. A = A ∪ {(u, v)}
2. UNION(u, v)
3. return A

## Analysis

E = edges.

V = vertices.

**Time Complexity**

O(E logV) or equivalently O(E logE), most of the time consuming operation is in sorting.

**Space Complexity**

O(E + V) as O(E) is for the array of edges and O(V) is the array of disjoint-set added together giving us linear complexity.

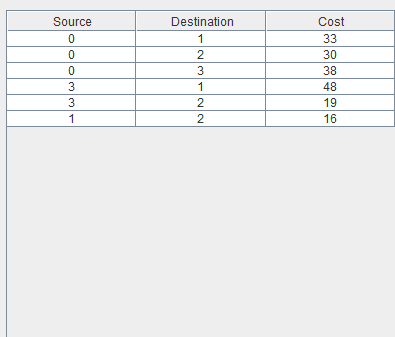
## Applications

Used in constructing trees for [broadcasting](https://en.wikipedia.org/wiki/Broadcasting_(networking)) in computer networks.

Used in c[ircuit design](https://en.wikipedia.org/wiki/Circuit_design) for implementing efficient multiple constant multiplications.

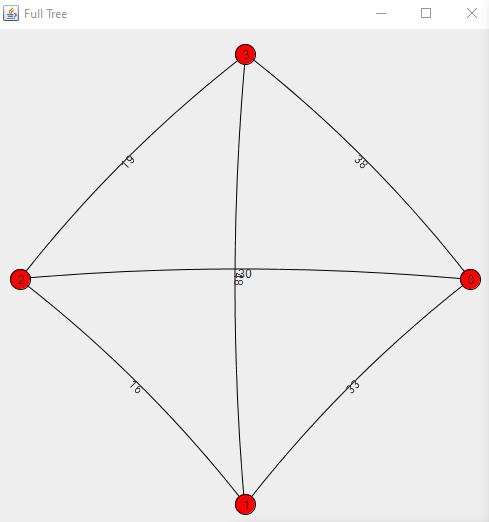
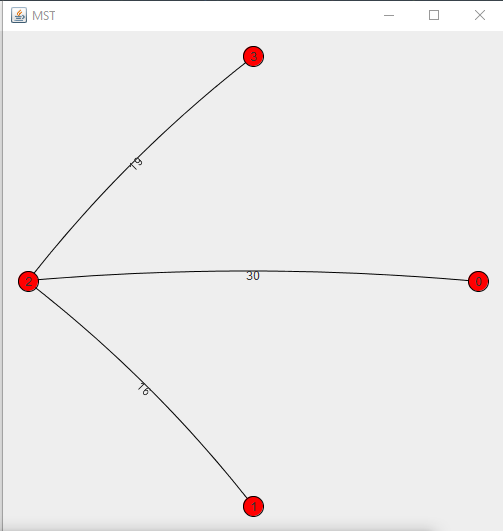
## Test

Edges of the graph:



The original full graph is Figure A and the Minimum Spanning Tree generated is Figure B and has a cost of 65.

**Figure A Figure B**

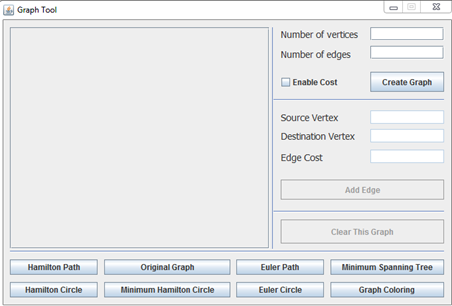
 

# 

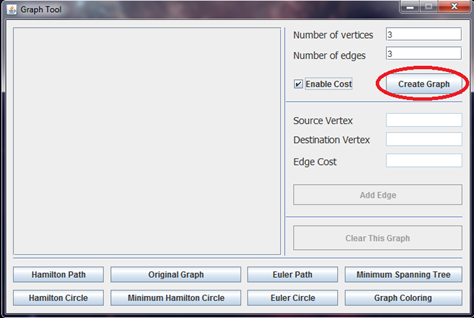
# 

# How to Use

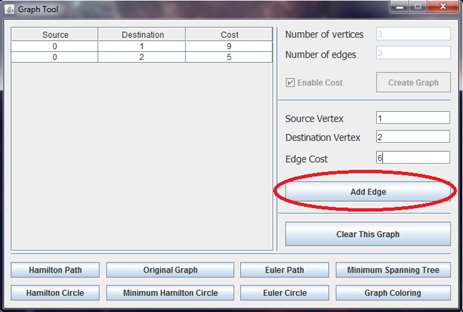
1- As soon as the program starts, the main screen will appear.



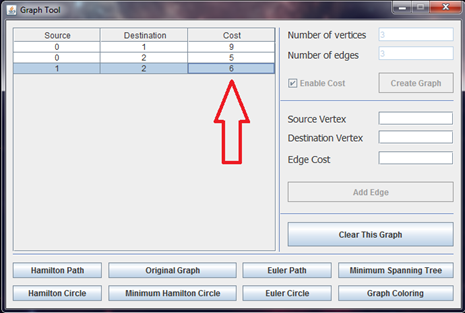
2- Then specify the number of vertices, number of edges and whether or not it is weighted (has cost) or not then click the Create Graph button.



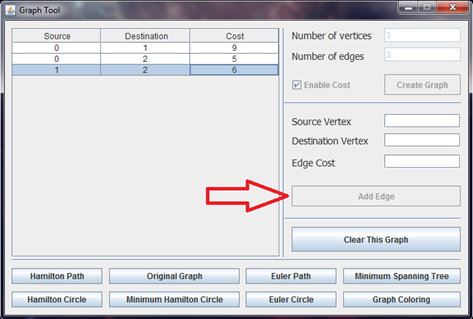
3- Start entering the edges source and destination vertices as well as the cost if existing then click on the Add Edge button.



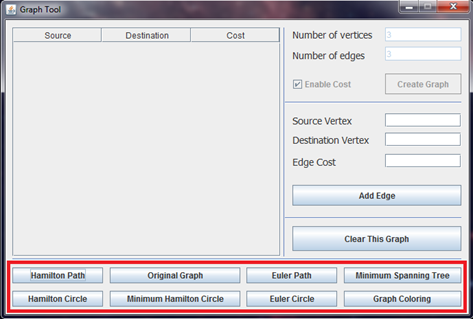
4- The edges entered will appear in the table as shown, if any modifications are needed, double click on the edge inside the table and just edit it.



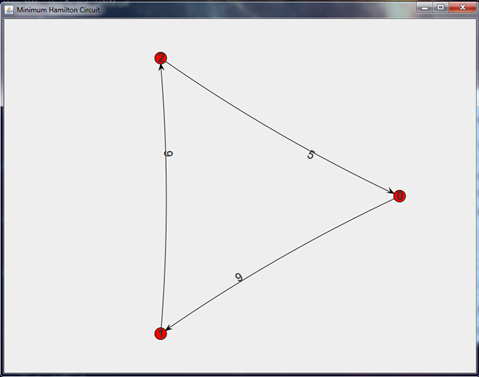
5- When the number of edges specified is reached, the Add Edge button will be disabled.



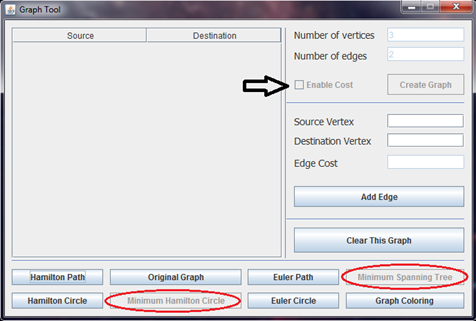
6- Choose what type of graph or algorithm you want to apply to the inserted graph by clicking on any of the buttons on the red square.



7- Once the button is clicked, a window will appear with the output graph.



Note 1: If an algorithm requires cost and cost was not available, it will be unavailable as shown.



Note 2: If at any time you want to clear the graph, just click at the Clear This Graph button and it will take you back to Step 1.

