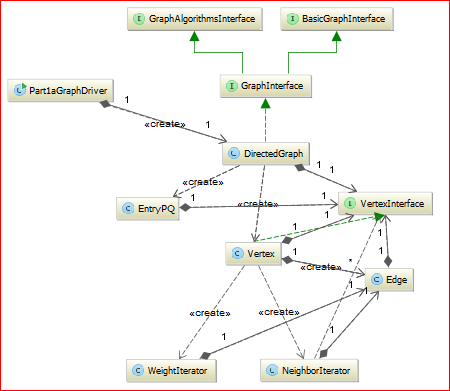
# Comp151 Lab14

Lab14 consists of **Part1a**, **Part1b**, **Part1c**, and **Part2**.

Each part should be worked on **as a separate project in IDEA** as you will be changing common files. Please zip each part separately for grading.

# PART1a

The attached package contains implementations of the graph algorithms that we studied in the class.



Two of the methods in DirectedGraph class need to be implemented:

1. getDepthFirstTraversal (see algorithm 28.13)
2. getCheapestPath (see algorithm 28.24)

Start by copying the respected algorithms as comments and then implement the code. Test it with Part1aGraphDriver.java provided. Once finished your output should match the provided sample run below:

Testing the directed, weighted graph in Figure 28-18a.

Graph has 9 vertices and 13 edges.

Edges exist from the first vertex in each line to the other vertices in the line.

(Edge weights are given; weights are zero for unweighted graphs):

A -> B 2.0 D 5.0 E 4.0

B -> E 1.0

C -> B 3.0

D -> G 2.0

E -> F 3.0 H 6.0

F -> C 4.0 H 3.0

G -> H 1.0

H -> I 1.0

I -> F 1.0

Number of vertices = 9 (should be 9)

Number of edges = 13 (should be 13)

Edges are OK.

Breadth-First Traversal beginning at vertex A:

A B D E G F H C I <-- Calculated

A B D E G F H C I <-- Expected

Depth-First Traversal beginning at vertex A:

A B E F C H I D G <-- Calculated

A B E F C H I D G <-- Expected

Finding the cheapest path in the graph in Figure 28-18a:

The cheapest path from A to B is

A B

and has a cost of 2.0.

The cheapest path from A to C is

A B E F C

and has a cost of 10.0.

The cheapest path from A to D is

A D

and has a cost of 5.0.

The cheapest path from A to E is

A B E

and has a cost of 3.0.

The cheapest path from A to F is

A B E F

and has a cost of 6.0.

The cheapest path from A to G is

A D G

and has a cost of 7.0.

The cheapest path from A to H is

A D G H

and has a cost of 8.0.

The cheapest path from A to I is

A D G H I

and has a cost of 9.0.

Done

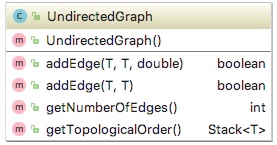
# PART1b

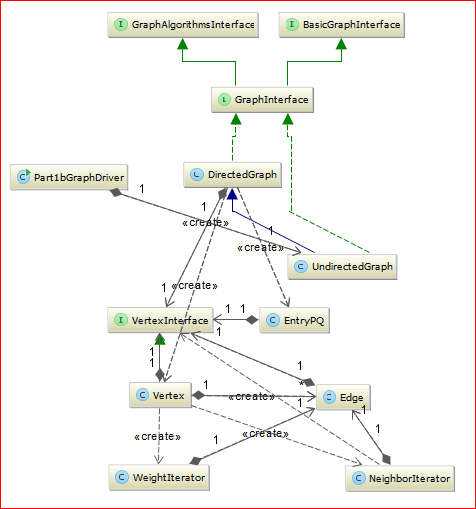
Create UndirectedGraph class that **inherits** from the DirectedGraph class and implements GraphInterface:

**public class UndirectedGraph<T> extends DirectedGraph<T> implements GraphInterface<T>, java.io.Serializable**

The UndirectedGraph class overwrites only several methods as shown in the UML diagram below; these methods must take advantage of inheritance and utilize appropriate methods from the **super** class:

* addEdge – must support edges going from begin to end and from end to begin
* getNumberOfEdges – the number of edges is the number of added edges divided by 2
* getTopologicalOrder – must throw UnsupportedOperationException





Test it with the Part1bGraphDriver.java class provided. Your output should match the provided sample run below:

Testing the undirected, weighted graph.

Graph has 9 vertices and 13 edges.

Edges exist from the first vertex in each line to the other vertices in the line.

(Edge weights are given; weights are zero for unweighted graphs):

A -> B 2.0 D 5.0 E 4.0

B -> A 2.0 E 1.0 C 3.0

C -> B 3.0 F 4.0

D -> A 5.0 G 2.0

E -> A 4.0 B 1.0 F 3.0 H 6.0

F -> E 3.0 C 4.0 H 3.0 I 1.0

G -> D 2.0 H 1.0

H -> E 6.0 F 3.0 G 1.0 I 1.0

I -> H 1.0 F 1.0

Number of vertices = 9 (should be 9)

Number of edges = 13 (should be 13)

Edges are OK.

Breadth-First Traversal beginning at vertex A:

A B D E C G F H I <-- Calculated

A B D E C G F H I <-- Expected

Depth-First Traversal beginning at vertex A:

A B E F C H G D I <-- Calculated

A B E F C H G D I <-- Expected

Finding the cheapest path in the graph in Figure 28-18a:

The cheapest path from A to B is

A B

and has a cost of 2.0.

The cheapest path from A to C is

A B C

and has a cost of 5.0.

The cheapest path from A to D is

A D

and has a cost of 5.0.

The cheapest path from A to E is

A B E

and has a cost of 3.0.

The cheapest path from A to F is

A B E F

and has a cost of 6.0.

The cheapest path from A to G is

A D G

and has a cost of 7.0.

The cheapest path from A to H is

A D G H

and has a cost of 8.0.

The cheapest path from A to I is

A B E F I

and has a cost of 7.0.

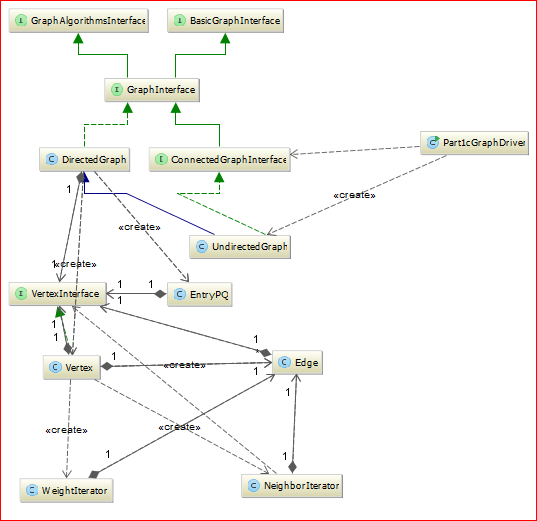
CORRECT UnsupportedOperationException thrown: "Topological sort illegal in an undirected graph."

Done

# PART1c

Implement a method with following signature **public boolean** isConnected(T origin)

in UndirectedGraph class that detects whether a graph is connected. The method is defined in ConnectedGraphInterface and your UndirectedGraph class will need to be changed to implement this interface. Test the method with the provided Part1cGraphDriver.java



Once finished your output should match the provided sample run below:

The graph in Figure 28-1

without the edge joining Hyannis and Barnstable, and

without the edge joining Orleans and Chatham is not connected:

isConnected() returns false

The original graph in Figure 28-1 with all of its edges is connected:

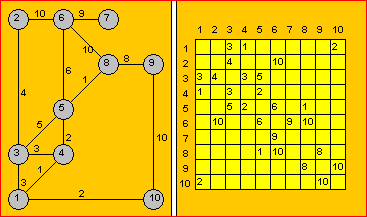
isConnected() returns true

# PART2

Implement *Prim’s Algorithm* for finding a **Minimum Spanning Tree** (MST) embedded in a **weighted connected** graph. The MST problem is to find a tree that is made up of all the nodes in the graph and a subset of the edges such that the sum of the edge weights is a minimum.

The algorithm is stated formally as follows: <http://en.wikipedia.org/wiki/Prim%27s_algorithm>

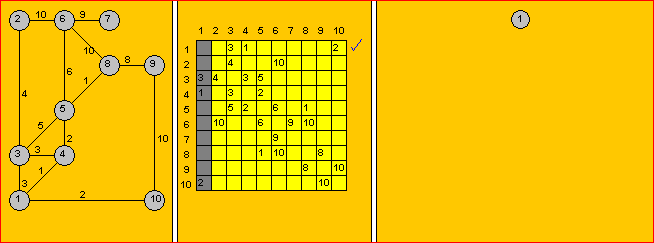
Write your own class that uses the **adjacency matrix** to represent the graph. This array is specified by the costs. Ask the user for the number of nodes and the probability of edge existing between two vertices. Generate weights and edges randomly. Since we will not use directed graphs, this array will be symmetrical:

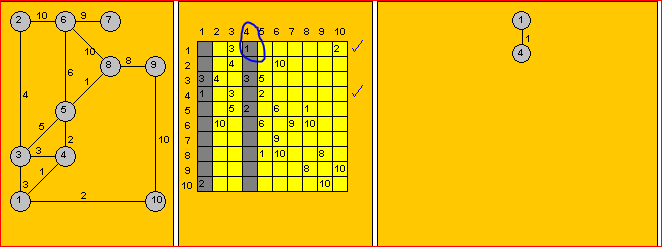


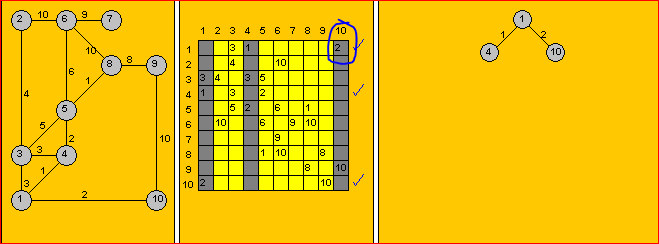
The *Prim’s Algorithm* steps are as follow:

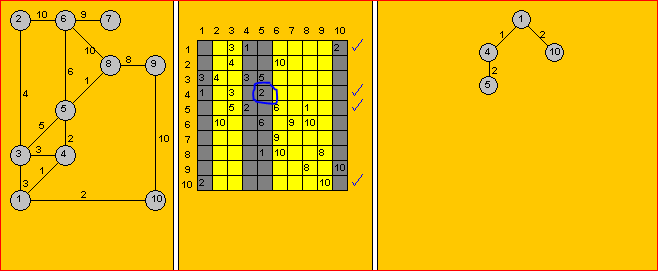
1. *We put a mark beside the first row and we gray-out the first column*
2. *Between the elements that are not grayed-out and belong to a row with a mark we choose the least A(j,k). If all elements are grayed-out, the algorithm terminates.*
3. *We put mark beside the kth row and we gray-out the kth column. We return to step b.*

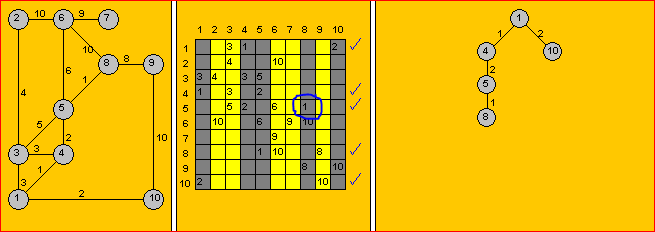
NOTE that in every step *Vk* node becomes *Vj*‘s child:

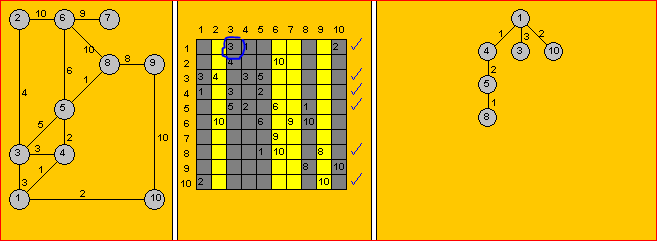


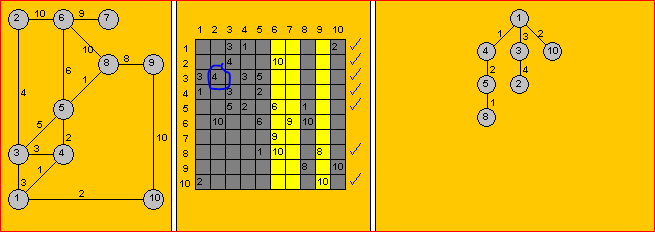


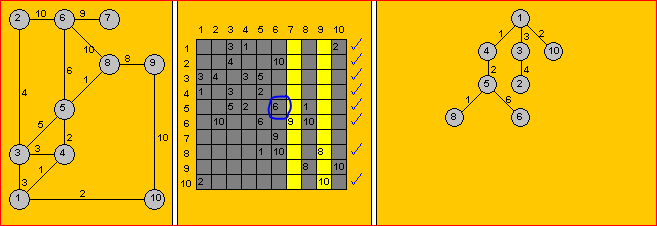


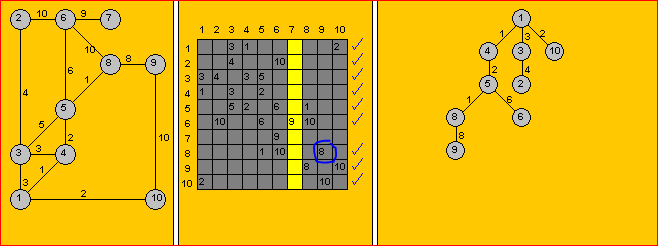


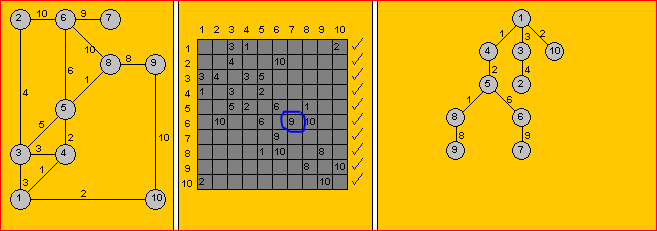












The output of the finished program should display the adjacency matrix for the graph that is connected, and the computed adjacency matrix for the minimum spanning tree for that graph. Notice that row 0 and column zero are not displayed since they are used for checkmarks. For example, see a sample run below:

How many nodes in your graph?

10

Probability of edge? (type 70 for 70%)

50

\*\*\*\*\* GENERATED GRAPH \*\*\*\*\*

[ 1][ 2][ 3][ 4][ 5][ 6][ 7][ 8][ 9][10]

[ 1] 3 1 2

[ 2] 4 10

[ 3] 3 4 3 5

[ 4] 1 3 2

[ 5] 5 2 6 1

[ 6] 10 6 9 10

[ 7] 9

[ 8] 1 10 8

[ 9] 8 10

[10] 2 10

The graph has cycles.

\*\*\*\*\* MINIMUM SPANNING TREE FOR THE ABOVE GRAPH \*\*\*\*\*

[ 1][ 2][ 3][ 4][ 5][ 6][ 7][ 8][ 9][10]

[ 1] 3 1 2

[ 2] 4

[ 3] 3 4

[ 4] 1 2

[ 5] 2 6 1

[ 6] 6 9

[ 7] 9

[ 8] 1 8

[ 9] 8

[10] 2

The minimum spanning tree is acyclic.

Process finished with exit code 0

If the generated graph is not connected the minimum spanning tree cannot be computed. For example, see a sample run below (to get this matrix I used Random with seed of 101):

How many nodes in your graph?

8

Probability of edge? (type 70 for 70%)

10

\*\*\*\*\* GENERATED GRAPH \*\*\*\*\*

[ 1][ 2][ 3][ 4][ 5][ 6][ 7][ 8]

[ 1]

[ 2] 3 4

[ 3]

[ 4] 3

[ 5] 3

[ 6] 4

[ 7] 3 2

[ 8] 4 4 2

The graph is not connected, the minimum spanning tree will not be calculated

Process finished with exit code 0

The following UML design was utilized in implementation of the above solution:

# /Users/ania/Desktop/Untitled.jpeg

**This is your last Comp151 project, have fun!!!**