Data Structures & Algorithms LAB – Fall 2015
(BS-SE-F14 Morning & Afternoon)

Lab # 13

Instructions:

- **Attempt the following tasks exactly in the given order.**
- Make sure that there are no ***dangling pointers*** or ***memory leaks*** in your programs.
- Indent your code properly.
- Use meaningful variable and function names. Follow the naming conventions.
- Use meaningful prompt lines/labels for all input/output.

Task # 1 *(Max Time: 20 Minutes)*

In this lab you are going to implement a class **Graph** for storing and processing **directed graphs**. The declaration of your class should be as shown below:

```
class Graph {
  private:
      int** adjMatrix;          // Adjacency matrix of the graph
      int maxVertices;          // Max number of vertices which can be present in the graph
      int n;                    // Current number of vertices present in the graph
      bool* visited;            // Array to keep track of visited/unvisited vertices

  public:
      Graph (int maxV, int currV);
            // Constructor to allocate and initialize an "empty graph" of the specified size.
            // Here, maxV indicates the max # of vertices and currV indicates the current # of
            // vertices

      ~Graph ();                // Destructor

      bool addVertex (int& v);
            // Adds a new vertex to the graph. After adding the new vertex, this function should
            // store the number of the newly inserted vertex into the reference parameter v and
            // should return true. If the new vertex could not be added, this function should
            // return false.

      bool addEdge (int u, int v);          // Adds the edge <u,v> to the graph
      bool removeEdge (int u, int v);       // Removes the edge <u,v> from the graph
};
```

Note that if an object of the above **Graph** class contains **n** vertices, then these vertices will be assumed to be numbered from **1** to **n**.

Task # 2 *(Max Time: 25 Minutes)*

Add the following public member functions to the **`Graph`** class:

`bool isEmpty ()`
This function will return true if the graph is empty, it will return false otherwise. A graph is said to be empty if there are no edges in the graph.

`bool isComplete ()`
This function will return true if the graph is *complete*, it will return false otherwise.

`void clear ()`
This function will remove all edges from the graph. Vertices will NOT be removed.

`void display ()`
This function will display the adjacency matrix of the graph on the screen in a neat and readable way. Only data of first **`n`** vertices should be displayed (where **`n`** is the current number of vertices present in the graph).

`int inDegree (int v)`
This function will determine and return the in-degree of the vertex **`v`**. It will return **`-1`** if **`v`** is invalid.

`int outDegree (int v)`
This function will determine and return the out-degree of the vertex **`v`**. It will return **`-1`** if **`v`** is invalid.

Task # 3 *(Max Time: 20 Minutes)*

Now, implement the following public member functions of the **`Graph`** class:

`void findUniversalVertex ()`
This function should find and display the number of the Universal vertex in the graph. Note that a **Universal vertex** is a vertex that has **an outgoing edge to every other vertex in the graph**. If there are more than one universal vertices in the graph, then this function should display the numbers of all such vertices. If the graph does not contain any universal vertex, this function should display an appropriate message on screen.

`void findIsolatedVertex ()`
This function should find and display the number of the Isolated vertex in the graph. Note that an **Isolated vertex** is a vertex that has **no incoming edge and no outgoing edge**. If there are more than one isolated vertices in the graph, then this function should display the numbers of all such vertices. If the graph does not contain any isolated vertex, this function should display an appropriate message on screen.

Task # 4 *(Max Time: 25 Minutes)*

Now, implement the following member functions of the **`Graph`** class:

`void DFS ()` ***// public member function***

This function will initialize the **`visited`** array to **false**. Then, it will ask the user to enter the number of the vertex from which DFS traversal should be started. After that, it will call the following private member function on that vertex.

`void DFS (int v)` ***// private member function***

This function will perform a DFS traversal of the graph starting from the vertex **`v`**. This function MUST be implemented **recursively**. During the traversal, this function should **display the vertices in the order in which they are traversed**. During the traversal, whenever this function has more than a single choice of vertices at a certain point, this function should traverse them in **increasing order** of vertex numbers.

If after the completion of DFS traversal which started from vertex **`v`**, the graph still contains some unvisited vertices, then this function should select the minimum-numbered-remaining-vertex and start the DFS traversal from that vertex. This process should be repeated until all vertices have been visited.

Task # 5 *(Max Time: 10 Minutes)*

Implement the copy constructor of the Graph class. The prototype should be:

`Graph (const Graph&)`

Task # 6

Redo all of the above-mentioned functionalities of the **`Graph`** class, but use an **Adjacency List** to store the graph, instead of an Adjacency Matrix.