COEN 244 (Winter 2018) - Assignment 4: Polymorphism + Operator Overloading

Deadline: **Friday March 23rd 23:55pm**

Type: Group Assignment (Groups of 2 students max)
Note: The assignment must be submitted on Moodle.
Submission format: Create only ONE zip file (.gz, .tar, .zip are acceptable. **.rar file is NOT acceptable**) that contains all the header files, cpp files and test files.

Problem statement

A Graph is formally define as *G=(N,E)* consisting of the set *N* of vertices (or nodes) and the set *E* of edges, which are ordered pairs of the starting vertex and the ending vertex. Each vertex has ID (int) and value (int) as its basic attributes. Each edge has a weight (int), starting vertex, and ending vertex.

A Graph can be a directed graph where vertices are connected by edges, and all the edges are directed from one vertex to another.

A Directed Acyclic Graph (DAG) is a finite directed graph with directed cycles. This means from any vertex v, there is no way to follow a sequence of edge that eventually loops back to v again.

An undirected graph is a graph where the edges are bidirectional.

The definition of the abstract class **Graph** is provided below. Note that all its member functions are pure virtual. This is because the implementation of these functions is different from one graph to another. You are allowed to slightly modify this class by adding new member functions.

```
class Graph{
public:
	Graph();
	virtual ~Graph();

	//adds one vertex; returns bool if added successfully.
	virtual bool addVertex(Vertex& v)=0;

	//Bonus question: adds in a set of vertices; returns bool if added
	//successfully
	//virtual bool addVertices(Vertex* vArray) = 0;

	//removes a vertex; the edges that have connection with this vertex need to
	//be removed;
	virtual bool removeVertex(Vertex& v) = 0;

	//adds an edge; returns true if the edge is added successfully.
	virtual bool addEdge(Edge& e) = 0;

	//Bonus question: removes a set of edges; as a result, some nodes may remain
	//as orphan.
	//virtual bool addEdges(Edge* eArray) = 0;

	// remove the edge
	virtual bool remove(Edge& e) = 0;

	// returns bool if a vertex exists in a graph.
	virtual bool searchVertex(const Vertex& v) = 0;

	// returns bool if an Edge exists in a graph.
	virtual bool searchEdge(const Edge& e) = 0;
```

```
        // displays the path that contains the vertex.
        virtual void display(Vertex& v) const = 0;

        // displays the path that contains the edge.
        virtual void display(Edge& e) const = 0;

        // displays the whole graph with your own defined format
        virtual void display() const = 0;

        // converts the whole graph to a string such as 1-2-4-5; 1-3-5; each path
        // is separated by ';'
        virtual string toString () const = 0;

        //remove all the vertices and edges;
        virtual bool clean() = 0;
};
```

Problem 1: (65 marks)
1. Create the classes Vertex and Edge to represent the vertices and edges of a graph. Test the classes (0 Marks, compulsory implementation)

2. Create a concrete derived class of Graph. The class can represent a directed graph, undirected grpah, or DAG. Provide full code of the derived class. Test the classes. **(5*13 = 65 Marks)**

Problem 2: Operator Overloading (5*7 = 35 marks)
Improve the class created in Problem 1 by overloading the operators **=, ==, ++, <<, >, +.** These operators are defined as follows

1. **G1 == G2,** returns true if G1 and G2 have the exact same vertices and edges
2. G1 = G2, assigns Graph G2 to Graph G1;
3. **G++ and ++G**, increases the weights of all edges by one;
4. G3 = G1 + G2, returns a graph that contains all the nodes of G1 and G2, all the edges of G1 and G2;
5. G1 > G2, returns boolean if the sum of weights of G1's edges is greater than the sum of weights of G2's edges;
6. << G outputs the graph G.

Deliverable of Problem 1 and Problem 2 together should be in one project and packed as one zip file. You also need to provide tests by creating graphs and testing the member functions of the graph.

Assignment Marking Scheme:

- Program correctness (75%)
- Program output format, clarity, completeness, and accuracy (10%)
- Program indentation and readability (5%)
- Choice of significant names for identifiers (5%)
- Comments - description of variables and constants (5%)