CSE 204

Graph

You have been provided with two different implementations of a *Graph*: an adjacency matrix implementation (*GraphAdjMatrix.cpp*) and an adjacency list implementation (*GraphAdjList.cpp*). Please go through the two files thoroughly to understand each and every line of the program. Here *Graph* data structure has been implemented using a C++ class **Graph**. In addition to this class, two other classes are **already and fully implemented** in the file: ***Queue*** class to be used later by the BFS function of ***Graph*** and ***ArrayList*** class that has been used by the ***Graph*** class to store adjacency list information. Please understand these two helper classes carefully including the constructor and destructor functions.

The graph can be either directed or undirected. This choice will be given as input by the user.

The following functions are already implemented in the **Graph** class:

- *void setnVertices(int n)*: Sets the number of vertices in the graph. This function initializes the graph data structure by allocating memory for the graph storage.
- *void addEdge(int u, int v)*: Adds an edge (*u,v*) to the graph. In case of undirected graph, edge (*u,v*) is the same as edge (*v,u*).
- *void printGraph()*: Prints the graph in adjacency list format.

For this homework, you are required to add the following functions to the above implementations (**both files**):

Task 1: *void removeEdge(int u, int v)*: Removes the edge (*u, v*) from the graph.

Task 2: *bool isEdge(int u, int v):* Returns true if (*u, v*) is an edge, otherwise returns false.

Task 3: *int getDegree(int u, bool out=true)*: Returns the **out degree** of a vertex *u* if *out* is true. *out* is set to true as a default argument. If *out* is set to false while calling the function, the function should return the **in degree** of the vertex *u*.

Task 4: *void printAdjVertices(int u)*: Prints all adjacent vertices of *u*.

Task 5: *bool hasCommonAdjacent(int u, int v)*: Returns true if vertices *u* and *v* have some common adjacent vertices. Otherwise, it should return false.

Task 6: *void bfs(int source)*: Runs BFS algorithm on the graph using *source* as the source vertex. BFS statistics like parent, distance, and color information must be saved in class variables. So, you will need to define these as class private variables. Use the given ***Queue*** class in BFS. Define BFS related variables, i.e., parent, distance, and color, as class variables and initialize them memory appropriately in the ***Graph*** constructor.

Task 7: *int getDist(int u, int v)*: Returns the shortest path distance from vertex *u* to vertex *v*. You will first need to run BFS on the graph using *u* as the source vertex. Then, you will use the *dist* array to find the distance.

Task 8: Graph destructor function: Implement the destructor function for the ***Graph*** class. View the destructor functions of ***Queue*** and ***ArrayList*** to get some hints.

Task 9 (Bonus, **not mandatory**): Modify your adjacency list implementation to enable storing weighted graphs.

In a weighted graph, every edge (*u, v*) is associated with an integer weight *w*. For example, in a road network graph, we may need to store the distance of each road segment in the graph. In that case, we will require storing distance value as a weight of each edge that represent a road segment between two junctions.

So, modify your graph implementation (only the adjacency list form, because it is trivial to store weight in a matrix form) so that weight can be stored. Implement this in a **separate file** named *GraphAdjListWeighted.cpp* . All the functions mentioned above should be implemented in *GraphAdjListWeighted.cpp* . You can maintain an ***ArrayList*** named *weightList* to store weights of neighbor vertices (similar to *adjList*).

Several existing functions may need to be changed in *GraphAdjListWeighted.cpp* such as follows:

- *void addEdge(int u, int v, int w)* : Adds an edge (*u, v*) to the graph with weight value *w*.

Modify other class functions wherever required.

All the functions should consider **both directed and undirected graph**, whenever needed. Also, always check whether the vertex number is valid, i.e. between 0 and *nVertices* (inclusive).

Your final submission will consist of 2 (or 3) files:

- *GraphAdjMatrix.cpp*
- *GraphAdjList.cpp*
- *GraphAdjListWeighted.cpp* (if you do Task 9)

Put these files in a folder. The folder should be named with your 7-digit student ID (like 1705001). Create a zipped version of the folder naming it with your 7-digit student ID (like 1705001.zip) and upload it in moodle.

You must also satisfy the following requirements:

- Write main function codes in all 2 (or 3) cpp files to test all of the above tasks.
- You must extend the given code.