Programming Assignment #6
Incentive Date: Apr 23rd
Due Date: Apr 30th
Graphs - Prim's Algorithm For MST

Description:
For this programming assignment, you will implement Prim's Algorithm. Prim's Algorithm is a method for finding the minimum spanning tree of a weighted undirected graph. This means that it finds a subset of the edges that encompasses all vertices in the graph such that the total weight of the edges comprising the tree is minimal.

starter.zip (Starter Code):
The following is a brief description of the starter code provided to you:

- **class Graph:**
  - Contains a single private member variable:
    - **unordered_map<int, unordered_map<int, int>> adjList** which represents the adjacency list of the graph. Note that we are using an unordered_map to represent the Adjacency List in this PA rather than using a vector of lists in LE10. Vectors are efficient data structures to represent the adjacency lists when we have a fixed size graph with contiguous vertex IDs. However, when we have random IDs (that are not necessarily in contiguous order) and when we must dynamically remove existing vertices and edges from the graph, the vector of lists may not be a good choice. For instance, if we wish to remove an edge, we will need to traverse the list of a particular vertex to find the particular edge we need to remove. Therefore, using unordered_map to access the edge in O(1) amortized time allows us to efficiently remove such edges.
  - The Graph class defines the following public functions:
    - **void addVertex(int u):** adds the vertex with id "u" to the graph
    - **void addEdge(int u, int v, int w):** Adds an undirected edge between vertices "u" and "v" with weight "w"
    - **bool contains_vertex(int u):** Returns true if the vertex with id "u" exists and false otherwise
    - **bool contains_edge(int u, int v):** Returns true if an undirected edge exists between u and v or false otherwise
    - **void removeEdge(int u, int v):** Removes the undirected edge between vertices "u" and "v"
    - **void removeVertex(int id):** Removes the vertex identified by "id" from the graph if such a vertex exists
    - **int numVertices():** returns the number of vertices in the graph
    - **int getEdgeWeight(int u, int v):** If an edge between vertices "u" and "v" exists, the function returns the weight of the undirected edge between u and v. If an edge does not exist between u and v, you can return -1.

        - **vector<pair<int, int>> primMST():** Returns a vector of edges which comprise a minimum spanning tree of the Graph. An Edge is represented by the pair<int, int> where the two elements of the pair represent the two vertices connected by the edge. For example, if Edge 'e' is part of the minimum spanning tree, and 'e' connects Vertex 'a' and Vertex 'b', the pair.first must equal to 'a' and pair.second must equal to 'b' (or vice versa - pair.first can equal to 'b' and pair.second can equal to 'a').
            - The Prim's Algorithm will be tested as follows:
                - Your Graph class will be used to construct a network of Cities where each City is represented by a vertex in the graph. The weight of edges between the different cities represent the cost it would take to lay highways and associated infrastructure between the two cities. If any pair of cities do not have an edge between them, this means that it is not possible to connect the two cities using highways.
                    - Your task will be to return the minimum spanning tree of such a City Network so we can minimize the cost of connecting all the different cities using highways.
- **main.cpp**
    - You can use this file to test the different functions of your Graph class.

Contract (TASK):
Your task is to implement the different functions mentioned in the Graph class. You have been provided with the non-functional definitions of these algorithms. **You must implement this function definition in the given graph.h file,** by replacing the current non-functional definition with your functional implementation.

Deliverables and Submissions:

In a zip folder named using the format **<FirstName>-<LastName>-<UIN>-<PA6>.zip** you must submit the following file(s) to Canvas:

- **graph.h** (containing the functional method definitions of the Graph algorithms)

Do not use angular brackets in the name of the submission folder.

Grading:
Coding (80 points)
The graph functions will be tested using an automatic testing infrastructure. You will be provided with the testing infrastructure which you may use to evaluate your implementations.

The grading rubric is as follows:

- Add Vertex: 5 points
- Add Edge: 5 points
- Remove Edge: 10 points
- Remove Vertex: 10 points
- Prim's Algorithm: 50 points

testInfrastructure.zip (Test Methodology):
In order to familiarize you with the auto-grading environment on our end we are providing you with an automated test infrastructure that can be used to check the correctness of your code. The testing infrastructure is constructed as follows:

- **test_graph.cpp:** This file defines all the test functions that have been mentioned in the grading rubric above. It also runs these tests and outputs your final score.
- **test.py:** This file compiles test_graph.cpp in order to produce the executable. It then runs this executable which will indicate your score (as well as the test cases you have passed or failed).

Note: There is a possibility that one or more of your methods causes a segmentation fault which in turn causes the test script to crash when calling that particular method. In such a case, we will not be able to calculate your score for the assignment and you will be assigned a default grade of 0. After the grades are posted, you will be given 1 week to rectify your error(s) and resubmit the assignment. You will be subject to a 25% penalty on the grade obtained from the resulting resubmission of the assignment.

How to test your program natively (run your own test cases):
You can use main.cpp provided in the starter.zip folder to test your graph functions. You can then test the functionality of your methods by calling these methods and checking whether what they return is what you expect. To compile your own tests in main.cpp, **you can use the makefile provided within starter.zip to compile your program.** In your terminal:

- Run **make** on the terminal (this will create an executable named main)
- Next, run **./main** on your terminal to run your native tests

How to use testInfrastructure:
To check your score using testInfrastructure, you can follow either of the two options mentioned below:

- **Option 1:** Move your graph.h file into the testInfrastructure directory. After this, run **python3 test.py**
- **Option 2:** Move your graph.h file into the testInfrastructure directory. **Ensure that the makefile in this directory is the one given to you originally in the testInfrastructure.zip folder and not in the starter.zip folder.** In your terminal:
    - Run **make** in order to compile the program (which will compile test_graph.cpp and create an executable named ./graph_test).

- Next, run **./graph_test** on your terminal to actually run the tests and output your score.