**Lab Exercise #10**
**Due Date: Apr 26th**
**Graphs - Dijkstra's Algorithm**

**Description:**
For this lab exercise, you will implement Dijkstra's algorithm on an undirected Graph. Dijkstra's algorithm is used for finding the shortest path between vertices in a graph, which could represent anything from road networks to computer networks. Named after its inventor, Dutch computer scientist Edsger W. Dijkstra, the algorithm starts from a source vertex and explores all possible paths in the graph until it finds the shortest path to every other vertex. Dijkstra's algorithm guarantees finding the shortest path, but it's designed for graphs with non-negative edge weights.

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

- **class Graph:**
  - Contains two private member variables -
    - **int n:** Representing the number of vertices in the graph. You can assume for this lab exercise, that the graphs will contain vertices in the order 0, 1, ... n - 1 (i.e. vertices with contiguous IDs)
    - **vector<list<Edge>> v:** This will store the adjacency list representation of the undirected graph
      - For example: v[6] contains a linked list of "Edge" (which is a typedef for pair<int, int>) where the first element of the Edge is a neighboring vertex of 6 and the second element of the Edge is the weight associated with the edge between 6 and the corresponding neighboring vertex.
  - The Graph class defines the following public member functions:
    - **void addEdge(int x, int y, int w):** Given a vertex with ID "x" and a vertex with ID "y" and a weight "w", you will have to insert an undirected edge between the vertices "x" and "y" with weight "w". This means that both "x" and "y" should contain this edge within their respective lists.
    - **vector< pair<int, int> > dijkstra(int startNode):** Given a particular source vertex (startNode), run the dijkstra's algorithm. You must return a vector< pair<int, int> > where the index of the vector represents the corresponding vertex ID destination. The pair<int, int> that you must return for each destination vertex is the following: the first element of the pair must be the distance along the shortest path between the start vertex and this destination vertex; the second element of the pair must be the id of the previous vertex along the shortest path between the start vertex and this destination vertex.
      - For example: if the shortest path between source vertex 0 and destination vertex 3 was "0 -> 6 -> 3"

and the total distance along this shortest path was 15:
    - Then vector[3] = {15, 6} where {15, 6} is the pair - 15 is the first element of the pair representing the shortest distance to 3 from 0 and 6 is the second element of the pair representing the previous vertex on the shortest path from 0 to 3.
- You can use the STL Priority Queue to develop Dijkstra's algorithm.

- **string printShortestPath(int startNode, int endNode):**
  - Given a particular source vertex (startNode) and destination vertex (endNode), you must return a string representing the shortest path between the source vertex and the destination vertex. You will find the previous vertices returned by Dijkstra to be useful in implementing this algorithm. For example, if the shortest path between startNode 0 and endNode 3 is "0 -> 6 -> 3", the returned string must be:
    - "0 6 3 "
  - **If there is no path from the startNode to the endNode, you can simply return "".**

- **main.cpp**
  - You can use this file to test your Dijkstra's algorithm.

**[Advisory]**

- **The STL Priority Queue does not support increasing/decreasing existing elements in the Priority Queue. This means that when we want to update the distance associated with a particular destination vertex that already exists in the Priority Queue, we instead have to insert a new entry into the Priority Queue.**
  - **We can extend our Custom PriorityQueueHeap to support updating existing keys by maintaining an index associated with an entry and then bubbling the entry up when the distance is updated of the corresponding entry. However, in this assignment we do not have this capability due to the use of STL priority queues.**
  - **One way to optimize Dijkstra's algorithm using STL priority queues is to ensure that we do not process entries in our priority queue for whose vertices we have already recorded the shortest distance for. You are not expected to incorporate this optimization in this LE.**

**Contract (TASK):**
Your task is to implement the Dijkstra's and printShortestPath functions. 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>-<LE10>.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.**

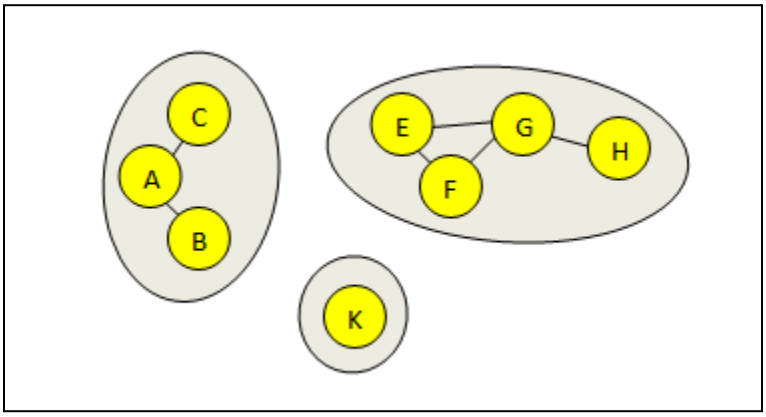
**Grading:**

**Coding (20 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:

- Dijkstra's Algorithm: 10 points
- printShortestPaths (Fully Connected Graph): 8 points
- printShortestPaths (Disconnected): 2 points
    - If there is a destination vertex that is not reachable from source vertex 0, this vertex would be part of a different connected component to that of the vertex 0. In such a case, you must return "" for the printShortestPaths function.
    - For example, in the following illustration (A, C, B) forms a connected component, (K) forms another connected component, and (E, F, G, H) forms a third connected component. A node in one connected component is not reachable from a node in a different connected component. A node can reach the other nodes present in its own connected component:




**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.