January 2020 CSE208: Data Structures and Algorithms II Sessional

Assignment on All Pairs Shortest Path Problem

In this assignment, you will implement two algorithms to solve the all pairs shortest path problem:

- Floyd-Warshall algorithm
- Johnson's algorithm

You will find detailed descriptions of these two algorithms in your textbook "Introduction to Algorithms is a book on computer programming" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein: Chapter 25.

First, you will implement an **Edge** class which will have the integer vertex numbers and real valued weight as it's private member variables. Also implement necessary public setter and getter functions. Weights are real numbers between -10000 to 10000 inclusive.

You will also implement a **Graph** class which will have an **adjacency list** representation. You can use the built-in **vector** class for this purpose. KEEP THE FOLLOWING FUNCTIONS:

- *void setnVertices(int n)*: Initializes graph and allocates memory as needed.
- *bool addEdge(int u, int v, double w)*: Adds an edge *(u, v)* with weight *w*. Allocates memory for the edge.
- *void printGraph()*: Prints the graph in adjacency list form. See the sample I/O for clarification.
- *void removeEdge(int u, int v)*: Removes edge *(u,v)* from the graph, if it exists.
- *Edge* searchEdge(int u, int v)*: Returns a pointer to the edge *(u,v)*. Returns NULL if there isn't any edge from *u* to *v*.
- *void reweightEdge(int u, int v, double w)*: Reweights edge *(u,v)* to w. If there isn't any edge from *u* to *v*, it adds an edge *(u,v)* with weight w through allocating memory.
- *bool isEdge(int u, int v)*: Returns true if edge *(u,v)* exists, false otherwise.

- *double getWeight(int u, int v)*: returns the weight *w* of edge *(u,v)*, if it exists. You can return some value >10000 if there is no edge from *u* to *v*.

Also properly deallocate memory used in it's destructor function.

Now, in addition to your required member variables, your Graph class must also have the following member variables:

- double **distanceMatrix: It will hold a matrix of size NxN (N is the no. of vertices in the graph). Each element l_{ij} of the matrix would hold the weight of the shortest path from vertex i to vertex j.

- int **parentMatrix: Another NxN sized matrix. It will hold necessary predecessor information. You will need it when you will print the shortest paths.

Allocate memory for them in the *setnVertices(int n)* function.

Add the following functions to your class:

- *void floydWarshall()*: This function will update the distance and parent matrices. Implement **Floyd Warshall's algorithm** here. You can assume when this function is called, there are no negative-weight cycles in the graph.

- *bool bellmanFord()*: This will return true if there is a *negative-weight cycle* in the graph, and false otherwise. Implement Bellman-Ford's algorithm to find it out.

- *void Djikstra(int n)*: This function will run Djikstra's algorithm with source vertex n.

  You can declare necessary member variables to hold distance and parent information obtained as required. You can also declare necessary member functions for the cause. **Do not declare unnecessary member variables or write unneeded member functions.** You can use the built-in priority_queue class for this purpose.

- *void johnsonsAlgo()* : This function will too update the distance and parent matrices like the floydWarshall() function did, except here you will implement **Johnson's Algorithm**. It will print "There is a negative-weight cycle." and return if there is one. **Make sure after returning from this function, the weights of the original graph is unchanged.** (i.e. if you change the weights inside the function, don't forget to set the weights as it was before returning from the function.)

- *double getShortestPathWeight(int u ,int v)*: this function will return the weight of the shortest path from vertex u to vertex v.

- *void printShortestPath(int u, int v)*: this function will print the shortest path from vertex u to vertex v along with the weights of the edges. See the sample outputs for further clarification.

- *void printDistanceMatrix()*: this function will print the matrix D where element d_{ij} is the weight of the shortest path from vertex *i* to vertex *j*. Print 'INF' in places where there were no paths found from vertex *i* to vertex *j*.

- *void printPredecessorMatrix()*: this function will print the matrix P where element p_{ij} is the vertex which is the predecessor of vertex *j* in the shortest path from vertex *i* to vertex *j*. Print 'NIL' in places where there were no paths found from vertex *i* to vertex *j*.

- *void cleanSPInfo()*: cleans up the distance and predecessor matrices. Sets all distances to 'INF' and all predecessor elements to '-1' (which would print 'NIL' if we call *printPredecessorMatrix()*).

Input / Output Format

The input graph is **directed.**

The first line of input will have two space separated integers N and M, denoting the no. of vertices and edges in the graph respectively. The next M lines will have 3 space separated values u, v and w, denoting an edge (u, v) with weight w.

After the M lines are finished, the program will output "Graph Created.\n" in the next line.

The next lines will be commands, the first integer C of the line indicates which command is to be executed. The commands are as follows:

- C = 1: Clear the values of the distance and parent matrices. (call *clearSPvalues()*). Output "APSP matrices cleared" after it is done.
- C = 2: Implement Floyd-Warshall Algorithm. Output "Floyd-Warshall algorithm implemented" after it is done.
- C = 3: Implement Johnson's Algorithm. Output "Johnson's algorithm implemented" after it is done.
- C = 4: This is a query command. C will be followed by two space separated integers *u* and *v* respectively (1<=*u, v*<=N). You are to print the weight of the shortest path and also the path itself along with edge weights from *u* to *v* in the next two lines.

- C = 5: Prints the graph in adjacency list form along with edge weights.
- C = 6: Prints the distance matrix D (call *printDistanceMatrix()*).
- C = 7: Prints predecessor matrix P (call *printPredecessorMatrix()*).

The program will end for any other value of C.

No. of vertices will be between 1 to 1000, inclusive. Vertex no. starts from 1. Weights of edges will be within -10000 to 10000, inclusive.

Sample I/O

Input	Output
5 9 1 2 3 1 3 8 1 5 -4 2 4 1 2 5 7 3 2 4 4 1 2 4 3 -5 5 4 6 1 7 6 2 6 7 4 1 2 4 1 5 5 4 2 3 1 3 6 4 2 3 4 1 5 8	Graph Created. APSP matrices cleared Predecessor Matrix: NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL Distance Matrix: INF INF INF INF INF INF INF INF INF INF INF INF INF INF INF INF INF INF INF INF INF INF INF INF INF Floyd-Warshall algorithm implemented Distance Matrix: 0 1 -3 2 -4 3 0 -4 1 -1 7 4 0 5 3 2 -1 -5 0 -2 8 5 1 6 0 Predecessor Matrix: NIL 3 4 5 1 4 NIL 4 2 1 4 3 NIL 2 1 4 3 4 NIL 1 4 3 4 5 NIL Shortest Path Weight: 1 Path: 1 --> 5(-4) --> 4(6) --> 3(-5) --> 2(4) Shortest Path Weight: -4 Path: 1 --> 5(-4)

	Graph: 1 : 2(3) --> 3(8) --> 5(-4) 2 : 4(1) --> 5(7) 3 : 2(4) 4 : 1(2) --> 3(-5) 5 : 4(6) Shortest Path Weight: -4 Path: 2 --> 4(1) --> 3(-5) APSP matrices cleared Johnson's algorithm implemented Distance Matrix: 0 1 -3 2 -4 3 0 -4 1 -1 7 4 0 5 3 2 -1 -5 0 -2 8 5 1 6 0 Shortest Path Weight: -4 Path: 2 --> 4(1) --> 3(-5) Shortest Path Weight: -4 Path: 1 --> 5(-4)

Special Instructions

Write ***readable, re-usable, well-structured, quality*** code. Do not write unnecessary functions or declare unnecessary variables. Delete all unnecessary variables and functions. Give meaningful names to your variables and maintain proper indentations.

You also cannot use any other built-in functions or libraries other than the vector and priority_queue classes.

Please DO NOT COPY solutions from anywhere (your friends, seniors, internet etc.)

Implement the algorithms with your own style of coding. **Any form of plagiarism (irrespective of source or destination), will result in getting -100% marks in the assignment. It is your duty to protect your code.**

Also, be informed that for repeated offense of plagiarism, the departmental policies suggest stricter measures.

Submission Guideline

1. Create a directory with your 7 digit student id as name.

2. Put the source file only into the directory created in 1.

3. Zip the directory.

4. Upload the zip into moodle.

For example, if your student id is 1705123, create a directory named 1705123. Put your source files(.cpp, .java etc) only into 1705123.zip and upload the 1705123.zip into moodle.

Failure to follow the above mentioned submission guideline will result in some penalty.

Marks Distribution

Implementation of Floyd-Warshall Algorithm	3
Implementation of Johnson's Algorithm	4
Code readability, cleanliness and maintenance of assignment instructions	2
Proper Submission as per given instructions	1
Total	**10**

Deadline

Deadline is set on 13 March, 2020 (Friday) at 11:55 pm.

THIS IS A HARD DEADLINE. THERE WILL BE NO EXTENSIONS FOR ANY REASON WHATSOEVER. START YOUR ASSIGNMENT EARLY.