Graph & Dijkstra's Algorithm

Finding paths

Topics:

- Build a graph with an adjacency list
- Use Dijkstra's algorithm to find best paths
- Code in Object Oriented C++
- Use of File IO

Outcomes:

- Identify, construct, and clearly define a data structure that is useful for modeling a given problem.
- State some fundamental algorithms such as merge sort, topological sort, prim's and Kruskal's algorithm, and algorithmic techniques such as dynamic programming and greedy algorithms
- Combine fundamental data structures and algorithmic techniques in building a complete algorithmic solution to a given problem
- Design an algorithm to solve a given problem

Description

This project is going to build directed & undirected weighted graphs then search them using Dijkstra's algorithm to find best paths between nodes.

Use the following Guidelines:

- Give identifiers semantic meaning and make them easy to read (examples numStudents, grossPay, etc).
- Keep identifiers to a reasonably short length.
- User upper case for constants. Use title case (first letter is upper case) for classes. Use lower case with uppercase word separators for all other identifiers (variables, methods, objects).
- Use tabs or spaces to indent code within blocks (code surrounded by braces). This includes classes, methods, and code associated with ifs, switches and loops. Be consistent with the number of spaces or tabs that you use to indent.
- Use white space to make your program more readable.

Important Note:

All submitted assignments must begin with the descriptive comment block. To avoid losing trivial points, make sure this comment header is included in every assignment you submit, and that it is updated accordingly from assignment to assignment.

Programming Assignment:

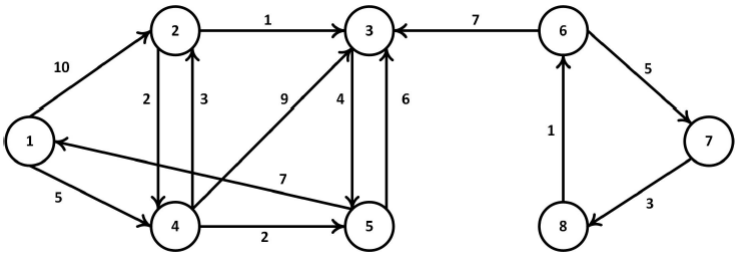
Instructions:

This program will have you build on the Adjacency Matrix you previously completed. You will be adding Dijkstra's Algorithm functionality to it.

And make use of Data Structures you've coded in the past:

- MatrixGraph
- Linked List
- MinHeap/MaxHeap
- Stack
- Arrays

You will be building a Graph Data Structure using the Adjacency Matrix method. It will read the topology of the graph from a file then build the vertices and edges.



The user will then be given an interface to perform searches on the graph from specified starting nodes and goal nodes.

Specifications:

Part 0 – Adjacency Matrix

Complete the prior homework that has you creating a graph using an Adjacency Matrix.

All functionalities should be implemented.

If you have not completed the Matrix Graph, you MUST now.

Part 1 – Adding Dijkstra's Algorithm.

Previous MatrixGraph Class Description:

Method	Description
MatrixGraph(int, bool)	Constructor builds matrix with a number of vertexes and true if directed
addEdge(int, int):void	Adds an edge between two vertices
addEdge(int, int, float):void	Adds an edge with a weight
removeEdge(int, int):void	Removes an edge between two vertices
adjacent(int, int):bool	Returns whether two vertices have an edge between them
getEdgeWeight(int, int):float	Returns the weight of an edge, throws exception if edge doesn't exist
setEdgeWeight(int, int, float):void	Changes an edge weight, throws exception if edge doesn't exist
toString():std::string	Returns a string representation of the graph for easy output
printRaw():void	Prints the 2D Array to standard output (primarily for debugging)
pathExists(int, int):bool	Returns if a path exists between two vertices
getBFSPath(int, int):std::vector<int>	Returns a vector of vertex numbers that shows the path between two vertices in order from start to goal. *This is the only place you can use standard library containers.*

Growing the Matrix Graph to include Dijkstra:

New Methods	Descriptions
getDijkstraPath(int, int):std::vector<int>	This method will take a start vertex and goal vertex and return a path decided by Dijkstra's Shortest Path Algorithm. This Method stops when it finds the target vertex.
getDijkstraAll(int):std::vector<std::vector<int>>	This method will take a starting vertex and run Dijkstra's Algorithm to find the shortest path from that start to ALL other Vertices in the Graph. Each path should be built as a vector and this method returns a Vector of Vectors. Empty Vector means no path!

getDijkstraPath(int, int):std::vector<int>

This method is like the getBFSPath from the previous assignment. The difference is that you'll be performing Dijkstra's Shortest Path Algorithm to find the shortest path in the graph from Start to a Goal. Return that path as a Vector of Integers.

getDijkstraAll(int):std::vector<std::vector<int>>

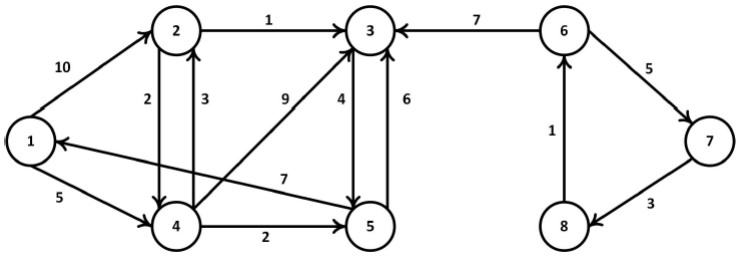
This method has you getting a shortest path from a given Start vertex to EVERY OTHER reachable vertex in the graph.

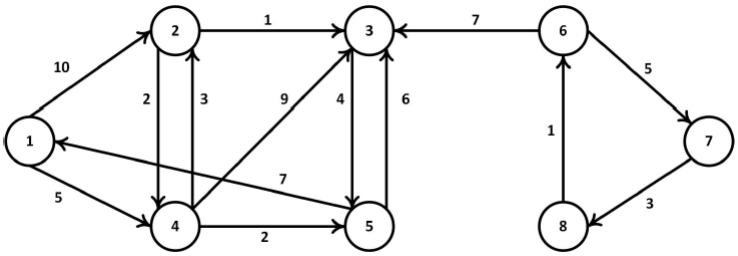
This will return an Vector of Vectors. Each vector contained in the outer Vector will give the shortest path to reach that particular vertex. The outer Vector should be |V| in size. An empty Vector should represent that there is no path.

Since each vertex is numbered, you can create a vector for each index in the array.

On this graph→

It would return an array of Vectors size 8.





Examples:

If you start start at Vertex 1 (index 0):

getDijkstraAll would return an array with these contents:

Vertex	Index		Index values
1	0	Empty Vector (starting point)	
2	1	Vector<int> → [1,4,2]	[0,3,1]
3	2	Vector<int> → [1,4,2,3]	[0,3,1,2]
4	3	Vector<int> → [1,4]	[0,3]
5	4	Vector<int> → [1,4,5]	[0,3,4]
6	5	Empty Vector	
7	6	Empty Vector	
8	7	Empty Vector	

If you start start at Vertex 7 (index 6):

getDijkstraAll would return an array with these contents:

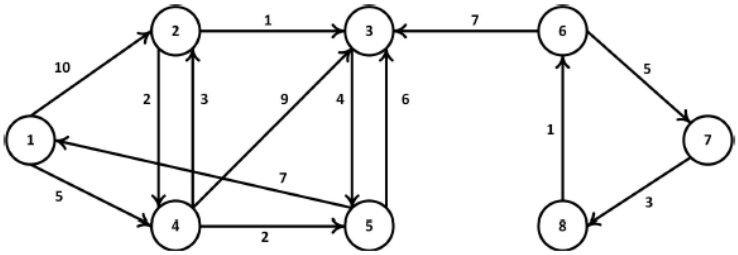
Vertex	Index		Index values
1	0	Vector<int> → [7,8,6,3,5,1]	[6,7,5,2,4,0]
2	1	Vector<int> → [7,8,6,3,5,1,4,2]	[6,7,5,2,4,0,3,1]
3	2	Vector<int> → [7,8,6,3]	[6,7,5,2]
4	3	Vector<int> → [7,8,6,3,5,1,4]	[6,7,5,2,4,0,3]
5	4	Vector<int> → [7,8,6,3,5]	[6,7,5,2,4]
6	5	Vector<int> → [7,8,6]	[6,7,5]
7	6	Empty Vector (starting node)	
8	7	Vector<int> → [7,8]	[6,7]

NOTE: For clarity I'm showing both Vertex values and indexes. The vertex values are in red.

User Interface

The user interface should be a menu:

```
Welcome to the Graph tester!
1) Print the graph
2) Find a BFS path
3) Find a Single Dijkstra Path
4) Find all Dijkstra Paths from a start
5) Start a file
6) Add a BFS path to the file
7) Add single Dijkstra Path to file
8) Add all Dijkstra Paths from a start
0) Quit
```



Sample Graph for examples below

Output

Regardless of which search method is used. The Path output should be the same as the Matrix assignment.

1) Print the graph

Call your toString() and output the string to standard output.

toString()

The string output of the graph should look like this:

```
[<vertex>]:-->[<v1>,<v2>::<weight>]--> …
```

You should use setw(2) or %2 for the vertex numbers.

You should use setw(5) or %5 with a limit of 2 decimal places for the weight.

Single line example:

```
[ 1]:-->[ 1, 2:: 10.10]-->[ 1, 3::123.12]
```

Full example:

```
[ 1]:-->[ 1, 4::  5.00]-->[ 1, 2:: 10.00]
[ 2]:-->[ 2, 4::  2.00]-->[ 2, 3::  1.00]
[ 3]:-->[ 3, 5::  4.00]
[ 4]:-->[ 4, 5::  2.00]-->[ 4, 3::  9.00]-->[ 4, 2::  3.00]
[ 5]:-->[ 5, 3::  6.00]-->[ 5, 1::  7.00]
[ 6]:-->[ 6, 7::  5.00]-->[ 6, 3::  7.00]
[ 7]:-->[ 7, 8::  3.00]
[ 8]:-->[ 8, 6::  1.00]
```

HINT: look into string stream or sprintf.

2) Find a BFS path

Prompt the user for two integers. Then perform a Breadth First Search of the graph. When doing your breadth first, queue the neighbors by walking through the row for that vertex in the adjacency matrix. This essentially queues the neighbors in numerical order rather than clockwise or counterclockwise.

Output the path found between the starting and goal node.

```
[<vertex>:<cost to get there>]==>…
```

Vertex should be setw(2), cost should be setw(5) with 2 decimal places.

Example:

If the user inputs 2 and 5 the output would be:

```
BFS Path from 2 to 5 is:
[ 2:  0.00]==>[ 3:  1.00]==>[ 5:  4.00]
```

If no edge exists output: No path from <v1> to <v2>

For instance, if the user inputs 3 and 8 the output would be:

```
No BFS path from 3 to 8.
```

3) Find a Single Dijkstra Path

Prompt the user for two integers. Then perform a Dijkstra's Best Path Search of the graph. When doing your Dijkstra's Algorithm, process the neighbors by walking through the row for that vertex in the adjacency matrix make your appropriate calculations and add them to the HEAP. This essentially queues the neighbors in numerical order rather than clockwise or counterclockwise.

Output the path found between the starting and goal node.

```
[<vertex>:<cost to get there>]==>…
```

Vertex should be setw(2), cost should be setw(5) with 2 decimal places.

Example:

If the user inputs 2 and 5 the output would be:

```
DIJKSTRA Path from 2 to 5 is:
[ 2:  0.00]==>[ 4:  2.00]==>[ 5:  4.00]
```

If no edge exists output: No path from <v1> to <v2>

For instance, if the user inputs 3 and 8 the output would be:

```
No DIJKSTRA path from 3 to 8.
```

4) Find all Dijkstra Paths from a start

Prompt the user for a starting vertex. The preform Dijkstra's Algorithm to find the shortest path from that start to all other reachable Vertexes.

Follow the same pattern as previously described.

Start at vertex 1 - Data:

Vertex	Index		Index values
1	0	Empty Vector (starting point)	
2	1	Vector<int> → [1,4,2]	[0,3,1]
3	2	Vector<int> → [1,4,2,3]	[0,3,1,2]
4	3	Vector<int> → [1,4]	[0,3]
5	4	Vector<int> → [1,4,5]	[0,3,4]
6	5	Empty Vector	
7	6	Empty Vector	
8	7	Empty Vector	

Output:

```
DIJKSTRA Paths start at Vertex 1
Path to 2: [ 1:  0.00]==>[ 4:  5.00]==>[ 2:  8.00]
Path to 3: [ 1:  0.00]==>[ 4:  5.00]==>[ 2:  8.00]==>[ 3:  9.00]
Path to 4: [ 1:  0.00]==>[ 4:  5.00]
Path to 5: [ 1:  0.00]==>[ 4:  5.00]==>[ 5:  7.00]
Path to 6: No DIJKSTRA path from 1 to 6
Path to 7: No DIJKSTRA path from 1 to 7
Path to 8: No DIJKSTRA path from 1 to 8
```

5) Start a file

Prompt the user for a filename. Output the graph to the file in the same format as the input file.

6,7,8) Add a path to the file

Menu options 6, 7 and 8 follow through on putting paths information out to the file.

If a file has been created with option 5, then run the appropriate routine for finding a path, but send the output to the file instead of the console. This output is formatted as specified in the previous section.

If no file has been created yet, then give the user the error:

```
No file has been created yet.
```

Hint: There's a couple of different ways to do this. You could just keep the file open once it's been created and only close the file when the user exits. You could keep track of the filename and reopen the file for appending.

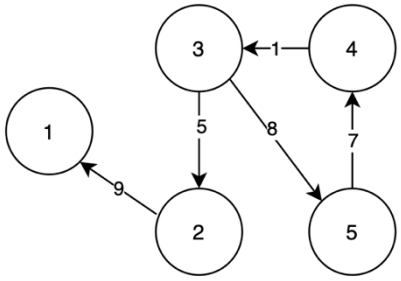
0) Quit

Exit the program. Don't forget to close any files that are open.

Hidden Cheat Code

If the user inputs 9999 to the menu, then printRaw() should be called.

printRaw()



	1	2	3	4	5
1					
2	9				
3		5			8
4			1		
5				7	

This should print the adjacency matrix row by row to standard output. This is a "no fills" command. Each entry should be limited to 2 decimal places and use setw(5) or %5

Output should look like this:

```
Adjacency Matrix:

   0.00   0.00   0.00   0.00   0.00
   9.00   0.00   0.00   0.00   0.00
   0.00   5.00   0.00   0.00   8.00
   0.00   0.00   1.00   0.00   0.00
   0.00   0.00   0.00   7.00   0.00
```

Extra Credit Opportunity:

Adjacency List +3:

Re-implement your graph as an Adjacency List rather than an Adjacency Matrix. This should include classes for Vertex and Edge.

Prim's Algorithm +3:

Implement Prim's Algorithm to find the Minimum Spanning tree of the graph. Add the secrete code 1111 to perform the MST and output the result to the console. The output should be the same as the toString() on the graph, but limited to the MST.

Submission/Grading

File Structure

Turn the following files into Gradescope:

Filename	Description
MatrixGraph_<lastname>.h	Contains the declarations for the MatrixGraph class
MatrixGraph_<lastname>.cpp	Contains the definitions for the MatrixGraph class
Queue_<lastname>.hpp	Contains definitions and declarations for your Queue (from previous project)
minmaxheap_<lastname>.hpp	Contains your previously made Priority Queue Heaps
<last-name>_TestGraph.cpp	Contains your main function with your UI/Interactive functionality
Makefile	Build your code project. Remember there is no file extension on a Makefile
Optional Files	
Stack_<last-name>.hpp	Stack data structure in case you use it to help with the path

(Replace <lastname> with your last name).

Testing your code:

Your code will be compiled on Gradescope using g++12 with C++20. Use this setup either locally or on general.asu.edu for accurate local testing.

Academic Integrity and Honor Code.

You are encouraged to cooperate in study group on learning the course materials. However, you may not cooperate on preparing the individual assignments. Anything that you turn in must be your own work: You must write up your own solution with your own understanding. If you use an idea that is found in a book or from other sources, or that was developed by someone else or jointly with some group, make sure you acknowledge the source and/or the names of the persons in the write-up for each problem. When you help your peers, you should never show your work to them. All assignment questions must be asked in the course discussion board. Asking assignment questions or making your assignment available in the public websites before the assignment due will be considered cheating.

*The instructor and the TA will **CAREFULLY** check any possible proliferation or plagiarism. We will use the document/program comparison tools like MOSS (Measure Of Software Similarity: http://moss.stanford.edu/) to check any assignment that you submitted for grading. The Ira A. Fulton Schools of Engineering expect all students to adhere to ASU's policy on Academic Dishonesty. These policies can be found in the Code of Student Conduct:*

http://www.asu.edu/studentaffairs/studentlife/judicial/academic_integrity.htm

ALL cases of cheating or plagiarism will be handed to the Dean's office. Penalties include a failing grade in the class, a note on your official transcript that shows you were punished for cheating, suspension, expulsion and revocation of already awarded degrees.