Graph & Uninformed Search Algorithm

Finding paths with Breadth First Search

Topics:

- Use object-oriented C++ to build a graph data structure using an adjacency matrix
- Traverse the graph and find paths between nodes using BFS
- Use C++'s built-in file IO to read graph data into your graph using the specified format
- Read and parse command line arguments to modify program behavior

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 builds undirected and directed weighted graphs using an adjacency matrix and searches them with breadth first search (BFS). See next page for project description.

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).
- Be consistent with the number of spaces or tabs that you use to indent.
- Use white space to make your program more readable.
- std namespace Data Structures are forbidden unless otherwise noted

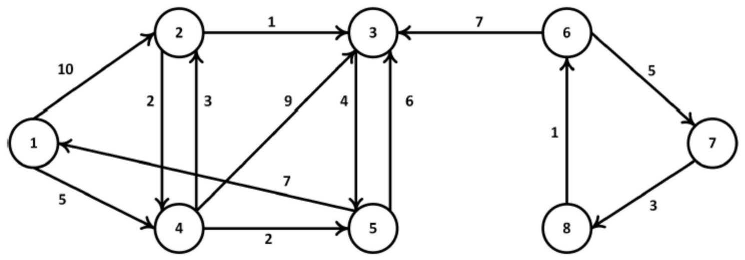
Important Note:

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:

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 building a representation of the graph using a dynamic 2D array.



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

This program will have you develop the class

- MatrixGraph

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

- Arrays
- Stack
- Queue

And build a program to test your graph.

You are expected to know file IO, command line arguments and 2D arrays. If you are struggling with these topics, seek help for them BEFORE starting this assignment.

Specifications:

Classes and Structure:

The following section covers the various classes needed. See submission section for file structure (which classes go in what files).

Queue:

You will be using a queue for your implementation of breadth first search.

You can reuse the queue from your prior projects. Refer to the stupid.os project documentation for an example queue interface.

MatrixGraph

Your graph class should have a simple Interface. Vertices are always given in the order of start then goal in the parameters. (Remember to account for directed vs undirected during all of these operations.

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
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.*

NOTE: toString() and printRaw() are documented in the User Interface section of the document.

Your MatrixGraph class should maintain a 2D array representation of the Adjacency Matrix.

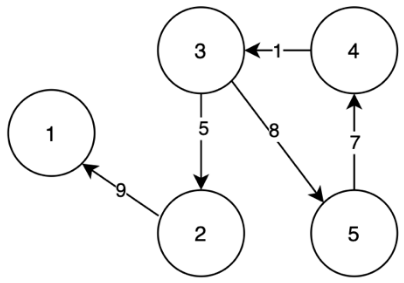
Implementing Your graph

Matrix Data Structure

Below is a simple diagram of what your 2D array will look like for an example weighted graph in both undirected and directed modes.

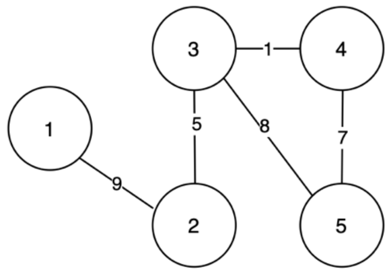
Directed

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



Undirected

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



Empty Spots

Empty spots should be represented with a 0.

In the above examples, the zeroes are omitted for readability.

Reading a Graph from a File/Command Line

You will build your graph from a file specified through a command line argument.

Command line spec:

```
<exe> {-u|-w} <file> [-ud]
```

Argument Explanations:

<exe>: first argument is always the program name

{-u|-w}: the second argument is either -u or -w. -u for unweighted, and -w for weighted.

<file>: path of the graph file. The format of this file is specified in the next section.

[-ud]: The presence of this argument means that the graph is undirected. By default, the file should be considered to be directed. The user should be able to add a -ud to the end of the command line to indicate the graph should be built as undirected.

If a file is processed as undirected, but it has forward and back edges in the file, then the second set overwrites the first. See example 2.

File Structure:

The file will be structured as follows:

Line 1: <number of vertices> <number of edges>

Subsequent lines:

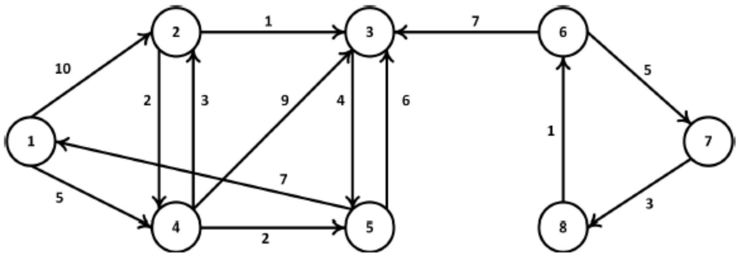
- Weighted:
  - <v1> <v2> <weight>
  - int, int, float
- Unweighted:
  - <v1> <v2>
  - int, int

You can safely assume that all values from the file will be greater than zero.

Example File:

Example weighted input file which represents the following graph

```
8 14
1 2 10
1 4 5
2 3 1
2 4 2
3 5 4
4 2 3
4 3 9
4 5 2
5 1 7
5 3 6
6 3 7
6 7 5
7 8 3
8 6 1
```



Example 2 on next page

Example of Undirected graph with duplicate:
(the following snippet is just part of a graph file)

```
4 5 7
5 4 3
```

The second edge description 5 4 2 would overwrite the 4 5 10 and represent the undirected edge in the adjacency matrix.

State of matrix after reading line 1

	1	2	3	4	5
1					
2					
3					
4					7
5				7	

State of matrix after reading line 2

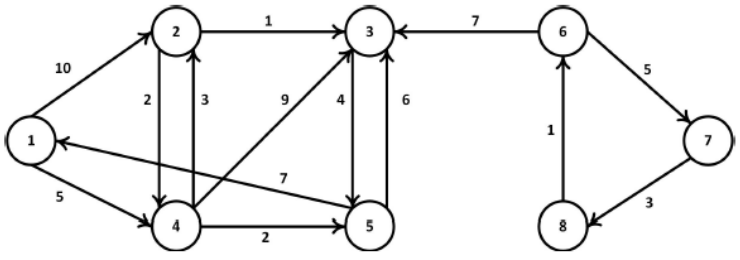
	1	2	3	4	5
1					
2					
3					
4					3
5				3	

Explanation
Since the graph is built as undirected, both 4 and 5 are adjacent to each other through one edge. It is not possible for this edge to have two different weights, so the more recent one (2 in this case) is used.

User Interface

The user interface should be a menu:

```
Welcome to the Graph tester!
1) Print the graph
2) Find a path
3) Start a file
4) Add a path to the file
0) Quit
```



Sample Graph for examples below

Assume that all examples on this page use the above graph and are directed.

Use 0 as the weight for output in the case that your program is reading an unweighted file. (That does not imply anything about how you should store empty spots, it is purely about printing the expected output.)

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 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:

```
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 path from 3 to 8.
```

3) Start a file

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

4) Add a path to the file

If a file has been created with option 3, then prompt the user for two vertices and output the result of Finding a Path between them to that file. 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

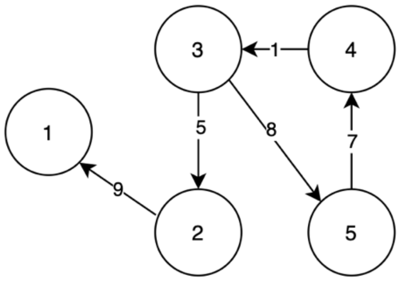
Exits the program

Hidden Cheat Code

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

printRaw()

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



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

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
```

Submission/Grading

File Structure

Turn the following files into Gradescope:

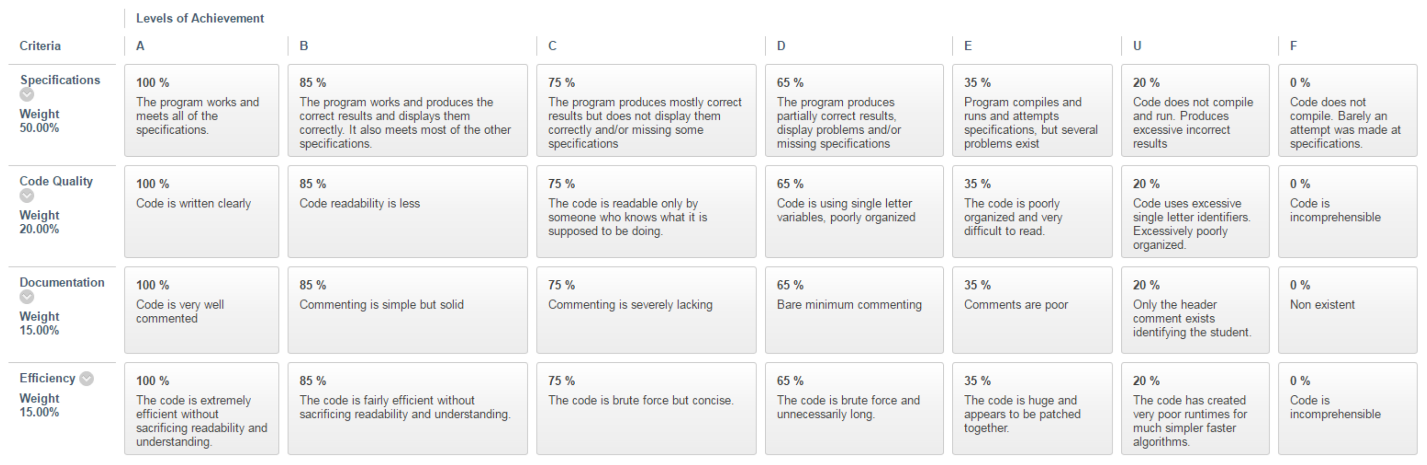
Filename	Description
MatrixGraph_<lastname>.h	Contains the declarations for the MatrixGraph class
MatrixGraph_<lastname>.cpp	Contains the definitions for the MatrixGraph class
Queue_<lastname>.hpp	Contains definitions and declarations for your Queue (from previous project)
<last-name>_TestGraph.cpp	Contains 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.

Rubric:



	Levels of Achievement						
Criteria	A	B	C	D	E	U	F
Specifications Weight 50.00%	100 % The program works and meets all of the specifications.	85 % The program works and produces the correct results and displays them correctly. It also meets most of the other specifications.	75 % The program produces mostly correct results but does not display them correctly and/or missing some specifications	65 % The program produces partially correct results, display problems and/or missing specifications	35 % Program compiles and runs and attempts specifications, but several problems exist	20 % Code does not compile and run. Produces excessive incorrect results	0 % Code does not compile. Barely an attempt was made at specifications.
Code Quality Weight 20.00%	100 % Code is written clearly	85 % Code readability is less	75 % The code is readable only by someone who knows what it is supposed to be doing.	65 % Code is using single letter variables, poorly organized	35 % The code is poorly organized and very difficult to read.	20 % Code uses excessive single letter identifiers. Excessively poorly organized.	0 % Code is incomprehensible
Documentation Weight 15.00%	100 % Code is very well commented	85 % Commenting is simple but solid	75 % Commenting is severely lacking	65 % Bare minimum commenting	35 % Comments are poor	20 % Only the header comment exists identifying the student.	0 % Non existent
Efficiency Weight 15.00%	100 % The code is extremely efficient without sacrificing readability and understanding.	85 % The code is fairly efficient without sacrificing readability and understanding.	75 % The code is brute force but concise.	65 % The code is brute force and unnecessarily long.	35 % The code is huge and appears to be patched together.	20 % The code has created very poor runtimes for much simpler faster algorithms.	0 % Code is incomprehensible

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.