CS531 Project #3: Graph Library
Due: Monday, Dec 2nd at 11:59PM

This is to be an individual effort. No partners. No Internet Resources
(*See the CS Honor Code Below*)

https://cs.gmu.edu/resources/honor-code/
https://cs.gmu.edu/resources/honor-code/statement-on-academic-integrity/

Before you Start: This project requires a good understanding of basic data structures in C. Review the text and lectures for Memory and Pointers, Structs and **Two-Dimensional Arrays**, and **Graphs**. In this project, you will be working with an **Adjacency Matrix** representation of a graph primarily.

Overview
For this assignment, you are going to use your C knowledge to create a Graph Library. In this case, you will receive a header file (**graph.h**) that may **not** be modified. You will create a file called **graph.c** to implement each of the prototypes in the header file, according to the specifications in this document.

The library code you write will be used by other programs to implement their graph algorithms. For this project, you will need to create your own source file with a main function to do all of the testing. **Do not add a main function inside of graph.c.**

You will only submit **graph.c**, not your main sourcefile or the header file.

Graph Library Overview
Your library will be a single source file that implements the prototypes found in the **graph.h** header file. You will create the graph and maintain it using an **Adjacency Matrix**, which is a two-dimensional static matrix inside of the Graph struct. Each **vertex** will have a single int as its value (label), which will have a non-negative value. Each **edge** will only be between two existing **vertices** and each will have a **positive weight greater than 0**. Each vertex may also have an edge to itself (**self-loop**).

In this project, you will have far more control over how you implement each of the functions. You will have to use the provided prototypes in the header file and you will get a description of what each function has to accomplish, however, you will be able to implement them how you like. Each operation must simply perform the task as described.

Your **graph.c** code must implement all of the prototypes as defined, however, you are welcome to add any additional local functions that you like to help. Only the functions that are described in the header file will be tested and graded with the automated testing software.

Your graph will support operations on a **directed, weighted** graph that may or may not be connected and may have cycles. Self-edges must be supported. All edges must have a positive weight (> 0).

Your graph may have disconnected vertices without any edges.

Adjacency Matrix Review (Theory) // This is not your project format

You will be storing and operating on your graph in the format of an **Adjacency Matrix**. Your graph will be **Directed**, and **Weighted**, which means the entries in your adjacency matrix need to account for these conditions. You may have self-loop edges (eg. 3 has an edge to 3 with a weight of 1), and you may have disconnected portions of your graph.

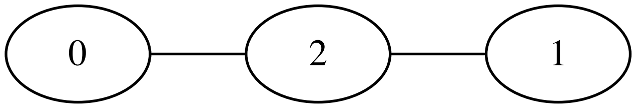
1	0	1
0	1	1
1	1	1

To the left is an Adjacency Matrix for an undirected graph with no self-edges (**as a simple example**). Here, the top row represents Vertex 0, the middle row represents Vertex 1, and the bottom row represents Vertex 2. For the columns, the left column is for Vertex 0, the middle column is Vertex 1, and the right column is Vertex 2.

The bottom right cell represents Vertex 2 itself (row 2 and column 2). Here, we use a 1 to show that this vertex exists. The middle cell (row 1 and column 1) is a 1 to show that there is a Vertex 1. The top left cell (row 0 and column 0) is a 1 to show that there is a Vertex 0.

The shaded bottom row represents the edges outgoing from Vertex 2. Here, Vertex 2 has an edge from 2 -> 0 and from 2 -> 1. There are no self-edges allowed in this example, so the bottom right entry just indicates there is a Vertex 2. The top right element (row 0 and column 2) shows there is also an edge from 0 -> 2. The middle row shows there is an edge from 1-> 2.

This is an image of what this graph would look like:

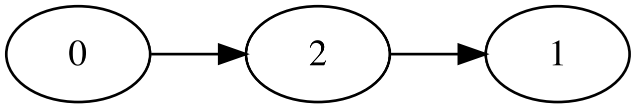


This is not a very exciting graph, but that shows how the Adjacency Matrix can represent a graph.

An Adjacency Matrix can represent all of the needed information about a particular graph, to include which Vertices exist, which edges exist, whether or not there is an edge from a vertex to itself (self-edge), and what weight each of the edges has.

For a **directed graph**, the same information would be in an Adjacency Matrix, but now it may not be symmetric, as shown below.

1	0	1
0	1	0
0	1	1



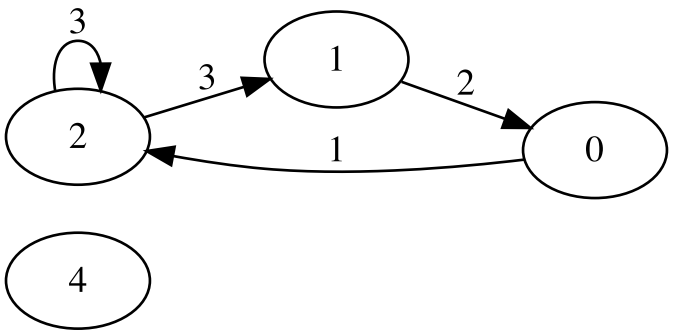
For a **weighted graph**, you would need to include the weight for each edge in the matrix elements.

For a graph with **self-edges**, you will also need to find a way to represent Vertices that have no self-edges and Vertices that do have self-edges, with weights.

You also need to choose a value to use to represent when there is no edge between two vertices.

Project Implementation Details *(Sample Data on the Last Page)*

You will be storing and operating on your graph in the format of an **Adjacency Matrix** using a two-dimensional array. Your graph will be **Directed**, and **Weighted**, which means the entries in your adjacency matrix need to account for these conditions. You also need to support **self-loop** edges. Each edge will also have a weight that must be greater than 0.



In this above example (which is representable by your project), Vertex 2 has an edge itself with a weight of 3. Vertex 4 is disconnected with no edges to it at all.

The row of the matrix will store the **from** vertex for each edge and the column of the matrix will store the **to** vertex for each edge. For example, the int at **adj_matrix[0][3]** would represent the weight of the edge from vertex **0** to vertex **3**.

Your implementation of the Adjacency Matrix will use the struct definition in graph.h:

```
typedef struct graph_struct {
  int max_vertex;
  int adj_matrix[MAX_VERTICES][MAX_VERTICES];
  int visited[MAX_VERTICES];
} Graph;
```

For your library, you will have to implement a **graph_initialize** function, which will create one instance of this Graph struct using malloc. This instance will be passed in to every function in your graph library to let your functions work with the context.

This struct has three members.

- **max_vertex** is an int and you may use this in your implementations to track the highest vertex in the graph, or any other state that you wish. How you use it (if at all) is up to you.

- **adj_matrix** itself is a static two-dimensional array that you **need to use** to implement your adjacency matrix. You may choose what values are used to initialize the array and what values will represent each of the different states: no vertex, no edge, vertex with no edges, or vertex with an edge, and the edge weights.

- **visited** is a one-dimensional array that you may use for any purpose that you may need.

There are also two defined constants in graph.h you may use as needed in your functions:

```
#define FILE_ENTRY_MAX_LEN  30
#define MAX_VERTICES        20
```

FILE_ENTRY_MAX_LEN is a constant defining the size you should use for a character array to read in the data from the file. This will be the max size of any line from either of your file formats. There are three functions to implement that work with files.

MAX_VERTICES is a constant that will define the maximum number of vertices that can be supported by this graph library. The two arrays in the struct are statically defined to support up to MAX_VERTICES indices. Vertex values can range from [0, MAX_VERTICES)

Project Details

In this project, you will create a new file called **graph.c**, in which you will implement each of the functions that are described in **graph.h**. You may create any number of additional functions inside of

graph.h	Contains all of the prototypes for the functions you will need to create.

Functions to Implement

You will be able to implement the functions any way you want as long as they fulfill the requirements. The return values will be very important because these functions will be an API that other programs may call. Each function will be self-contained, which means they will get a pointer to the Graph struct passed in, then they will perform one operation, then return.

As you're writing each function, you can debug and test them independently. Feel free to write any additional functions to use internally to make your implementation easier. Each function description may define what to return on any errors, as well as some of the cases that may be an error. Consider each function to see if anything else may be an error on implementation.

`Graph *graph_initialize();`

- This function should malloc and initialize a new Graph structure, then return a pointer to it.
- Remember to initialize each of the values inside **adj_matrix** as you want them.
  - You will need to loop through all of the indexes to initialize them.
  - After initializing, your matrix should represent a state with no vertices and no edges.
- **Returns**:
  - The pointer to the Graph if successful.
  - NULL on any errors.

`int graph_add_vertex(Graph *graph, int v1);`

- This function should add vertex number **v1** to the graph.
- If it already exists, simply return 0.
- If v1 is not a legal vertex (ie. < 0 or >= MAX_VERTICES), return error.
- **Returns**:
  - Return 0 if successful.
  - On any errors, return -1.

`int graph_contains_vertex(Graph *graph, int v1);`

- This function should check if the graph has vertex number **v1** in the graph.
- **Returns**:
  - Return 1 if it exists.
  - On any errors or if v1 is not in the graph, return 0.

`int graph_remove_vertex(Graph *graph, int v1);`

- This function should remove vertex number **v1** from the graph.
- If there are any edges connected to this vertex, they should also be removed.
- If v1 is not in the graph, you can return success.
- If v1 is not a legal vertex (ie. < 0 or >= MAX_VERTICES), return error.
- **Returns**:
  - Return 0 on successful remove.
  - On any errors, return -1.

`int graph_add_edge(Graph *graph, int v1, int v2, int wt);`

- This function should add an edge from **v1** to **v2** with weight of **wt**
- If this edge already exists, update it to the new weight.
- If either v1 or v2 are not in the graph, consider it an error and return.
- **Returns**:
  - Return 0 on successful add.
  - On any errors, return -1.

`int graph_contains_edge(Graph *graph, int v1, int v2);`

- This function should check if an edge from **v1** to **v2** exists.
- **Returns**:
  - Return 1 if an edge exists.
  - Return 0 in any other case.

`int graph_remove_edge(Graph *graph, int v1, int v2);`

- This function should remove an edge from **v1** to **v2**
- If either v1 or v2 aren't in the graph, consider it an error and return.
- If v1 and v2 exist, but have no edges already, it's a success.
- **Returns**:
  - Return 0 on success.
  - On any errors, return -1.

`int graph_num_vertices(Graph *graph);`

- This function should return the number of vertices in the graph.
- **Returns**:
  - Return the number of vertices on success.
  - On any errors, return -1.

`int graph_num_edges(Graph *graph);`

- This function should return the number of edges in the graph.
- **Returns**:
  - Return the number of edges on success.
  - On any errors, return -1.

`int graph_total_weight(Graph *graph);`

- This function should return the sum of all edge weights in the graph.
- **Returns**:
  - Return the total weight of all edges on success.
  - On any errors, return -1.

`int graph_get_degree(Graph *graph, int v1);`

- This function should return the total degree of edges on Vertex V1
  - This includes both the in-degree and out-degree combined
  - If there is a self-edge, that would count as both in and out, adding 2 to the degree.
- If there is no vertex V1, it is an error.
- **Returns**:
  - Return the degree (number of edges) on v1 on success.
  - On any errors, return -1.

`int graph_get_edge_weight(Graph *graph, int v1, int v2);`

- This function should return the weight of the edge from Vertex V1 to Vertex V2
- If there is no vertex V1 or vertex V2, it is an error.
- **Returns**:
  - Return the weight of the v1 -> v2 edge on success.
  - On any errors, return -1.

`int graph_is_neighbor(Graph *graph, int v1, int v2);`

- This function should check if v1 is connected to v2 (either direction)
- If there is no vertex V1 or vertex V2, it is an error.
- **Returns**:
  - Return 1 if there is an edge from v1->v2 or an edge from v2->v1.
  - On any errors or if there is no such edge, return 0.

`int *graph_get_predecessors(Graph *graph, int v1);`

- This function should return all Predecessors of Vertex V1 as a **dynamically allocated array**.
  - You will need to malloc an array big enough for all predecessor vertices + 1
  - The last entry of this array needs to be -1 to indicate no further predecessors exist.
  - For example, [1, 2, 8, -1] would represent Vertices 1, 2, and 8 are predecessors.
- If there is no vertex V1, it is an error.
- **Returns**:
  - Return a pointer to the new array of predecessors on success.
  - On any errors, return NULL.

`int *graph_get_successors(Graph *graph, int v1);`

- This function should return all Successors of Vertex V1 as a **dynamically allocated array**.
    - You will need to malloc an array big enough for all successor vertices + 1
    - The last entry of this array needs to be -1 to indicate no further successors exist.
    - For example, [1, 2, 8, -1] would represent Vertices 1, 2, and 8 are successors.
- If there is no vertex V1, it is an error.
- **Returns**:
    - Return a pointer to the new array of successors on success.
    - On any errors, return NULL.

`int graph_has_path(Graph *graph, int v1, int v2);`

- This function should check if there is path from v1 to v2.
- If there is no vertex V1 or vertex V2, it is an error.
- Returns:
    - Return 1 if there is a path from v1 to v2.
    - On any errors or if there is no such path, return 0.

`void graph_print(Graph *graph);`

- This function gives you an opportunity to be creative!
- When called, print your graph any way you like.
    - You can output it as an adjacency list, as a graphical representation of your adj matrix, or in any other manner than allows someone to easily interpret your graph.
    - In my class demo, I did a graphical representation of my adj matrix.

`void graph_output_dot(Graph *graph, char *filename);`

- This function outputs your graph in a format for the GraphViz program as a file.

Example Output File:

```
digraph {
0;
0 -> 2 [label = 1];
1 -> 0 [label = 2];
1;
2 -> 1 [label = 3];
2 -> 2 [label = 3];
4;
}
```

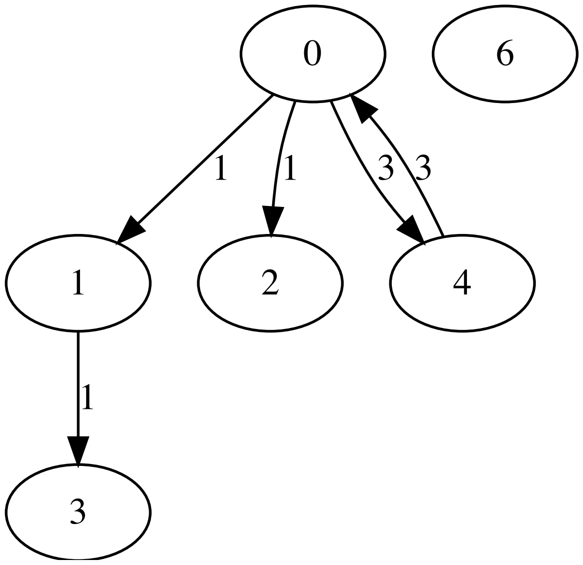
-
    - The format for GraphViz will be described in the next section.
    - When finished, if you opened a file with fopen, you need to use fclose.
- On any errors, return. *(Make sure to close any file that was successfully opened, if applicable)*

`int graph_load_file(Graph *graph, char *filename);`

- This function will load a graph from a file.
- The format for loading/saving files is very easy:
  - If you want to store a vertex with no edges, put the vertex number on its own line.
  - If you have an edge to store, put it in this comma separated format, on its own line:
    - vertex1, vertex2, edge_weight
- Example file to load from:

```
0,2,1
1,3,1
0,1,1
6
4,0,3
0,4,3
```

- Example graph this represents:



- **Returns**:
  - On any errors, return -1. *(Make sure to close any file that was successfully opened, if applicable)*
  - On success, return 0

`int graph_save_file(Graph *graph, char *filename);`

- This function will save your graph to a file.
  - Use the same file format as described in the graph_load_file function above.
- **Returns**:
  - On any errors, return -1. *(Make sure to close any file that was successfully opened, if applicable)*
  - On success, return 0

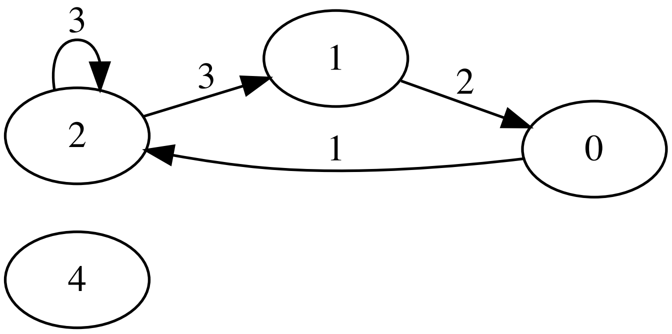
GraphViz (dot file)

The GraphViz (https://www.graphviz.org/) program (specifically **dot**) can be used to process this kind of a file:

Example Dot File (g.txt):

```
digraph {
0;
0 -> 2 [label = 1];
1 -> 0 [label = 2];
1;
2 -> 1 [label = 3];
2 -> 2 [label = 3];
4;
}
```

and make the following output:



This output was made by the GraphViz dot program from the example file above.

- Formatting Guidelines:
    - The first line of the file needs to be "digraph {"
    - The last line of the file needs to be "}"
    - Each line in the middle can be a vertex alone, "4;"
    - Or it can be an edge, "2 -> 1 [label = 3];"
        - The number after the label = is the weight of the edge.
    - These entries can be in any order based on how you want to implement it.
        - Only vertexes with no edges need to be in the file as a sole vertex.
        - However, it doesn't hurt to have the others there too.
        - In my example file, I have vertexes alone for 0, 1, and 4.
        - I have edges that link to vertex 2, so it will show up without problem.

Testing your dot formatted file can be done with the GraphViz package (available for Mac, Windows, or Linux), or you can go to this website and copy and paste your file contents into the window for a demo.

https://graphs.grevian.org/graph

If you copy and paste the example dot file above, you'll get the same graph image as on this page.

Submitting and Grading

Submit your **graph.c** on **blackboard** as the only file to submit.

No other naming formats are accepted, it has to be **graph.c**.

<u>Make sure to put your name and G# as a commented line in the beginning of your graph.c file. Also, in the beginning of your program list the known problems with your implementation in a commented section.</u>

All submissions will be **compiled**, **tested, and graded on Zeus!** Make sure you test your code on Zeus before submitting. If your program does not compile on zeus, we cannot grade it. If your program compiles but does not run or generate output on zeus, we cannot grade it.

Questions about the specification should be directed to the CS 531 Piazza forum, which is the primary location for discussions on this project. However, recall that debugging your program is essentially your responsibility; so please do not post long code segments to Piazza. **Do not post any code on Piazza in a public post.** Always post as a **private** post to **Instructors** for code questions. Any general questions about the assignment can be posted publicly.

You **have one late token** that may be used on any project in the course. No late submissions are allowed without using a token, and you only have one token for all three projects. This is meant for personal emergencies, so plan on submitting on time and saving your token for any emergencies throughout the semester.

Your grade will be determined as follows:

- **80 points** - Correctness. This is graded by automated script by checking the results of each of your functions as you complete them. You will get partial credit for most of the functions, so even if something is not working at the end, you may still get credit for any working code.

- **20 points** - Code & comments: Be sure to document your design clearly in your code comments. This score will be based on (subjective) reading of your source code by the GTA. The grader will also evaluate your C programming style according to the below guidelines.

Test your program by creating a main source file and compiling it with your graph.c file.

```
gcc -o graph main.c graph.c
./graph
```

Your main file should call all of the required functions and can be done like I had it demoed in class with a menu, or just a set of test cases. It is your responsibility to design your own tests to make sure each of the functions are implemented properly!

If your program does not compile, it will get a very low grade (just the code & comments points).

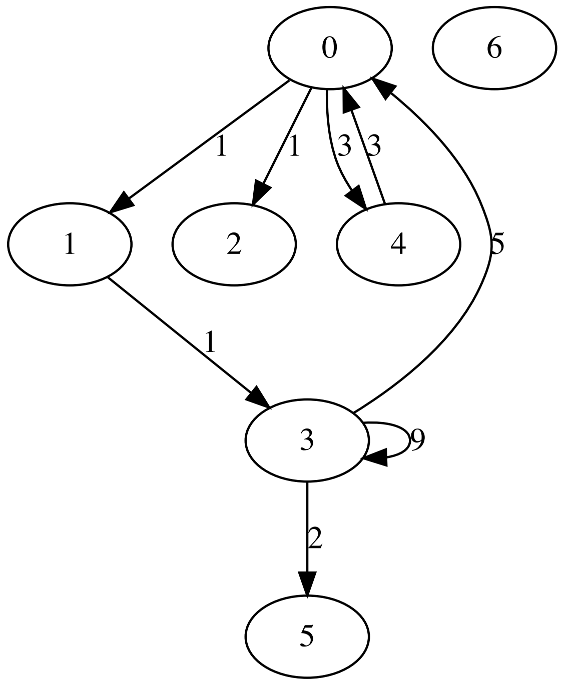
Code Style Guidelines (Will be Part of your Code and Comments Score)

1. No Global Variables may be used. (Global Constants are allowed; eg. const int val = 100;)
2. Always initialize all pointers with a value or to NULL on declaration.
3. Each block of code should increase the indent by 2-4 spaces.
4. Only use one statement per line. (eg. int x = 42; int y = 32; would be wrong)
5. Use Braces { } around all if/if else/else statements, even if they only have one statement.
6. Always check the return value for a call to malloc() to make sure it is not NULL
7. Set pointers to NULL after freeing them.
8. ***(Recommendation only)*** Functions **should** be fairly short (20 lines of code). If you have large functions that perform several operations, you should consider breaking them into smaller functions.

Sample Graph:

Metrics:

Num Vertices: 7
Num Edges: 8
Total Weight of Graph: 25
There is a Path from 4 to 5
There is no Path from 4 to 6
Degree of Vertex 3: 5
Successors of Vertex 3: [0 3 5 -1]
Predecessors of Vertex 3: [1 3 -1]
Is 2 a Neighbor of 0? Yes
Edge Weight of 3 -> 2 is



The graph at the right was loaded from the following file using graph_load_file function:

```
0
0,1,1
0,2,1
0,4,3
1
1,3,1
2
3,0,5
3,3,9
3,5,2
4,0,3
4
5
6
```

This also makes the following dot file (yours may differ, but it should generate the same image):

```
digraph {
0;
0 -> 1 [label = 1];
0 -> 2 [label = 1];
0 -> 4 [label = 3];
1;
1 -> 3 [label = 1];
2;
3 -> 0 [label = 5];
3 -> 3 [label = 9];
3 -> 5 [label = 2];
4 -> 0 [label = 3];
4;
5;
6;
}
```