CS 355 **Assignment #1** **Due: start of class, Wed. 9/19**

This is an individual assignment. Do your own work. You may not discuss any aspect of the problem (design, approach, code) with anyone but the course instructor. Searching for code online is not doing your own work (submitting it as your work is plagiarism, a kind of cheating).

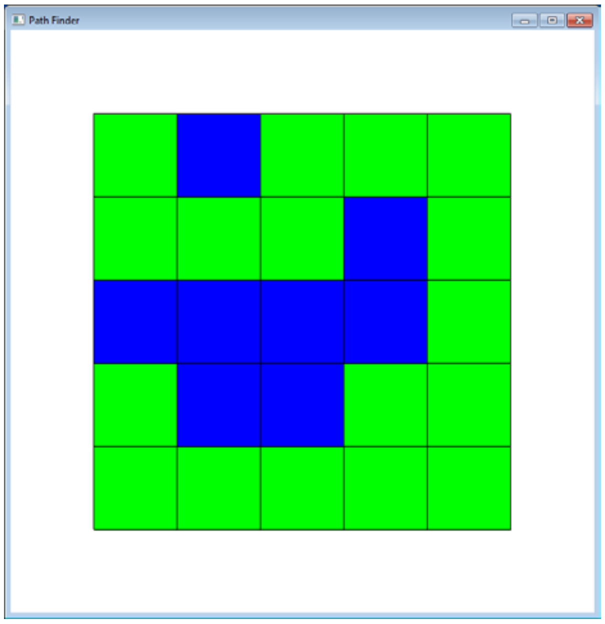
This assignment requires you to create a path finder program for a digital map using an input file, a graph class, and various containers from the STL. **Don't use any new C++ 11 features in this program.** The input file that represents the map will have the following format:

```
numrows numcols // number of rows and columns of data - positive ints
d0 d1 ... dnumcols-1 // first row of 0/1 cell data
d0 d1 ... dnumcols-1 // second row of 0/1 cell data
.
.
.
d0 d1 ... dnumcols-1 // last row of 0/1 cell data (last row based on
numrows specified above)
```

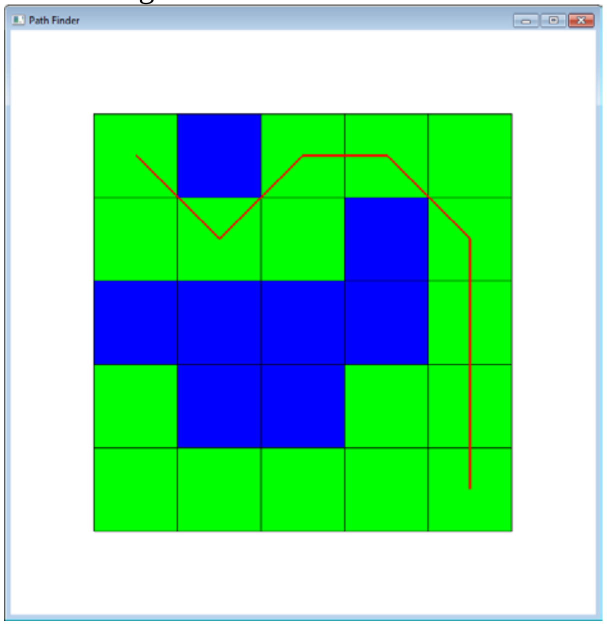
Each data item after the first row is either 0 or 1; 0 indicates traversable open space, 1 indicates an obstacle or water that cannot be traversed. Here's an example of a very simple map file:

```
5 5
0 1 0 0 0
0 0 0 1 0
1 1 1 1 0
0 1 1 0 0
0 0 0 0 0
```

This represents a map that looks like this:



After running your program, the path from top-left cell to bottom-right cell should look like this:



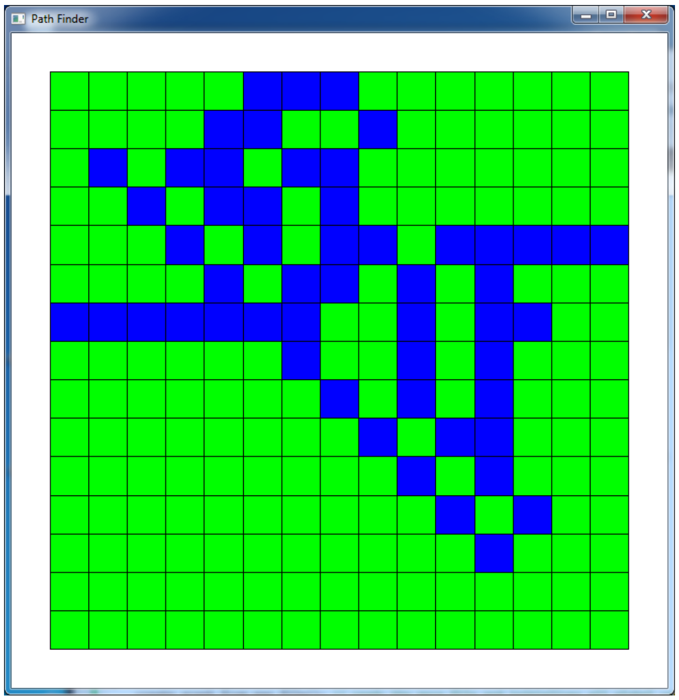
Create a graph object (an instance of the graph class) **called mapGraph** as a global variable declared before the #include directive for graphics.h (it will be used by the graphics code to draw the map).
For this program, the map will always be a square grid of cells, arranged in rows and columns. Each cell's location can be thought of as an ordered pair (row, col) (in that order); row numbers start at 0 and increase downwards and column numbers start at 0 and increase to the right. Note that this is how 2D arrays work in C++. Therefore, **using an input file** whose name is **specified by the user at run time**, **store the map data in a 2D array called mapArray**, which also needs to be a global variable defined before the graphics.h include line. The mapArray will allow you to access all the information you need to find connections between cells and will be used by the provided graphics code to draw the map. You may assume the grid of cells is no larger than 50 x 50 so you don't need to size it based on information in the input file. But you'll need to save the number of rows and columns in the specified map so you only process the relevant information. Initialize the global variables (rows, columns, cellsCount), based on information in the input file.

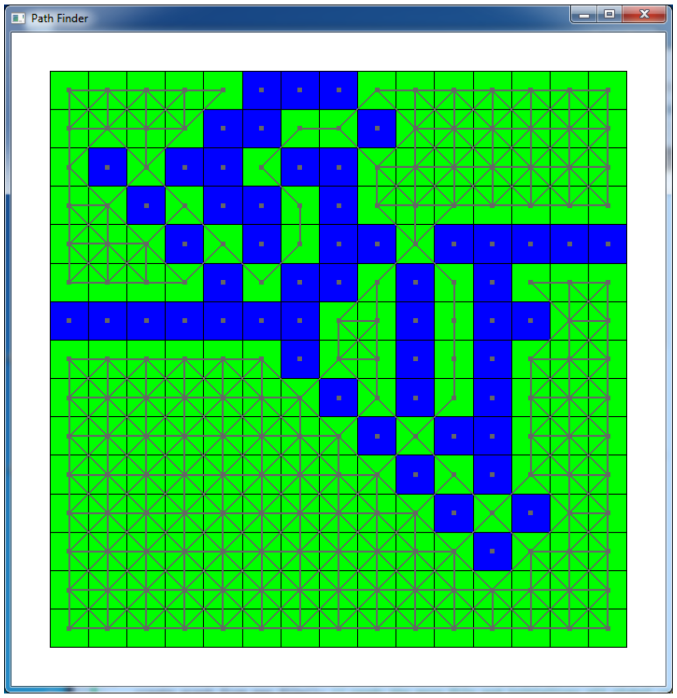
Using the provided starter project's graph.h file, **create an adjacency-list-based graph class**. The graph class doesn't need any memory management functions (constructors, assignment operator, destructor) since it only uses primitive types and STL objects. All pointers are used only to reference existing objects without making a copy, not to dynamically allocate memory. Since the graph class has no explicit constructor defined in it, you get a default constructor that takes no parameters for free. It sets any data members to their default values (an empty vector of vertex structs in this case). This class doesn't adhere to the usual object-oriented convention of having private data (encapsulation). You must keep the data public for the provided graphics code to work properly while keeping the graph class simple. Complete the graph class using separate compilation (a separate .h and .cpp file), and don't implement any of the class's methods in the .h file. None of the methods should interact with the user, read input, or provide output except for the printPath method, which should write the path information to the screen. The only modifications you should make to the graph class's .h file are to add include directives for any I/O functions or auxiliary STL containers used to implement the class's member functions.

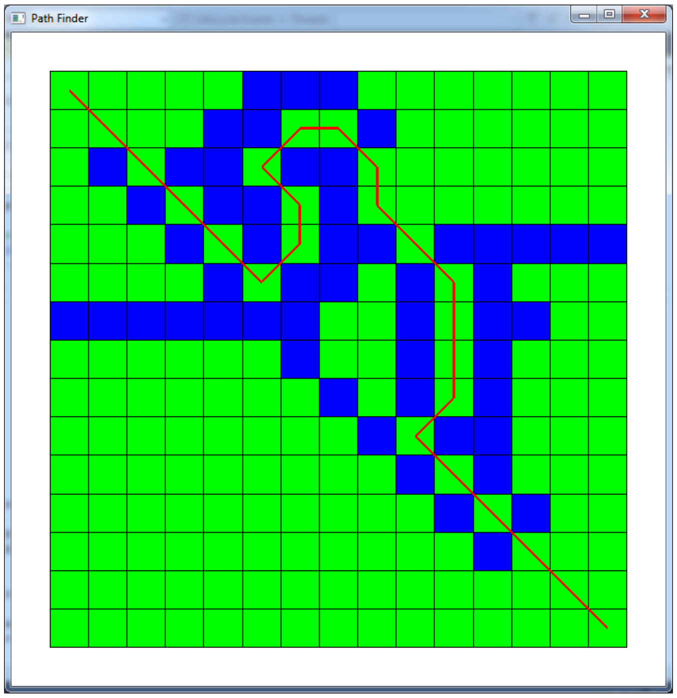
Implement the graph class's member functions addVertex and addEdge. The addVertex method should add a vertex struct to the end of the verts vector. The only data member you need to initialize is its id. The vertex ids should automatically be numbered (don't use an id parameter) consecutively starting at 0, i.e., the id must match the vertex's position in the vertices vector. The addEdge method should add an **undirected** edge between the vertices with the specified ids. This means an edge between two vertices can be specified in either order but still establishes a connection in both directions, i.e., each vertex is a neighbor of the other. So you don't need to and shouldn't call addEdge twice in the code that creates the graph. The graph class's addEdge method makes sure that each specified vertex is a neighbor of the other.

Add vertices to the graph row by row (top to bottom). There should be a vertex for each cell of the map, added left to right within each row. That way, the graphics code can deduce the position of the cell each vertex represents from its id. Use the data in the 2D array to **add edges** between vertices that represent adjacent non-obstacle cells (horizontally, vertically or diagonally). Since pointers are used to refer to neighbors **you must add all vertices before creating edges** (the address of a vertex in memory in a vector could change if the vertex vector grows).

When you run the program, after asking for the map file name and initializing the 2D array and graph, it should display the map when the start_graphics_loop function call in the main function is reached (one example shown below). If you click the mouse's left button once, you'll see the corresponding graph in gray, which is shown on the right for the map on the left.







Your program should find a path from the top-left cell to the bottom-right cell of the map. You'll do this using the graph class's **breadthFirst method**, which should conduct a breadth-first graph search given the vertex id of the start vertex. **Implement** this method using **the second version of the breadth-first search algorithm** presented in class (the one **that uses a queue** in the main loop).

Once you've implemented this method, **write the findAndPrintPath function** in the main.cpp file. This will be called when you click the mouse a second time. For now, just call the breadth-first search method on the map graph using the appropriate start id within the body of this function. Once you add this code to the function, clicking the mouse a second time should show the solution (path through the map) in red as shown on the right. Note that a map could have several paths that take you from the start cell to the end cell, but a breadth-first search will find the shortest path (fewest edges). If there's a tie for the shortest path, which one you get is implementation-dependent, but any shortest path is acceptable.

Implement the printPath method in the graph class, which takes a parameter that specifies the end vertex's id. This method assumes a search has already been conducted from a specified start vertex. It can be called repeatedly with different end vertex ids and should print the vertex ids **in order from the start vertex** used to conduct the last search **to the given end vertex**, using the path pointers stored in the graph’s vertices (don't run a new search). Note that you’ll have to determine the path going back from the end to the start, but print it in the opposite order (from start to end). This can be done with or without an auxiliary data structure, depending on your approach. Either way is okay. **Add a call to this method** with the appropriate vertex id **in the findAndPrintPath function** (after you invoke the graph’s breadthFirst method).

Create a folder named after you in the drop box folder in the network Fall 18 class folder and copy your Visual Studio project or your source files (.cpp, & .h) into the folder. Alternatively, you can make a single zip file out of your project folder or source files and upload it to the drop box over the web using the address http://pathfinder.bloomu.edu.