In this programming assignment, you will implement Dijkstra's Algorithm and explore the manner in which shortest paths dynamically shift as the cost function is repeatedly updated. After the user inputs vertices via a graphical interface, a random set of edges will be generated to complete the specification of a graph. At that point, the user may utilize your program to execute Dijkstra’s Algorithm on the graph, using an initial cost value of 1 on each edge. This process may then be repeated, with each edge’s cost reflecting the number of paths passing through that edge in the previous iteration.

For example, for the graph at left below, the initial cost of each edge is assumed to be one, but when Dijkstra’s Algorithm is run for each source node, the illustrated four spanning trees are produced:

![Graph Images]

The updated cost function for the graph will be based upon the number of paths passing through each edge, multiplied by that edge’s current cost. For instance, cost-1 edge (1,2) has 4 paths - (1,2), (1,2,4), (2,1), and (4, 2, 1) - passing through it after the first iteration, so its new cost value becomes 4. The updated graph is illustrated below, along with the spanning trees produced when Dijkstra’s Algorithm is executed on the updated graph:

![Updated Graph Images]

Your program will permit the user to interactively generate vertices on an interactive display. When all vertices have been generated, the user will use the keyboard to randomly generate edges between the vertices. The number of edges generated will be n(n-1)/6 (i.e., one-third the number of possible edges), with additional edges added to ensure connectivity (i.e., use a depth-first search to determine whether there are multiple connected components).

To implement this program, you’ll need a `Graph` class (defined in `graph.h` and implemented in `graph.cpp`) with the following members:

- A two-dimensional adjacency matrix of integer cost values, and an integer indicating the number of vertices. The adjacency matrix should be dynamic, i.e., a dynamic array of dynamic arrays, as demonstrated below.

```c++
typedef int* IntArrayPtr;  // Defined within graph.h
IntArrayPtr *adjacencyMatrix; // Graph data member
int numberOfVertices; // Graph data member
adjacencyMatrix = new IntArrayPtr[numberOfVertices]; // Dynamic creation of array
for (int k = 0; k < numberOfVertices; k++) // once numberOfVertices has
    adjacencyMatrix[k] = new int[numberOfVertices]; // been initialized.
```
• Default, copy, and initializing constructors (the initializing constructor should have a parameterized size). For this assignment, it is suggested that the initializing constructor use a cost value of -1 to indicate the absence of an edge, reserving the zero cost value for edges that exist but are not experiencing any traffic with a current Dijkstra iteration.
• Setting and accessor functions for the number of vertices, and for a particular edge’s cost in the adjacency matrix.
• A public member function to perform Dijkstra’s Algorithm on the **Graph** (returning a dynamic array indicating the resulting list of predecessor nodes). Note that the **climits** library contains an **INT_MAX** constant that will be useful in determining lowest-cost paths.

Your driver program, named **Dijkstra.cpp**, should set up the interface to permit user interaction, recording the set of vertices input by the user in an STL **vector** variable. After the user has generated the desired set of vertices, a keyboard command may be issued to generate the random set of edges (with additional edges added, as necessary, to ensure connectivity).

At this point, the user may perform Dijkstra’s Algorithm on the initial **Graph**, and then repeat the process on the updated **Graph**. Each time the algorithm is executed, the updated graph should be displayed (on top of the original graph’s display), with the cost of each edge visually represented by that edge’s thickness. An example is illustrated below.

An example program, **NOT_Dijkstra.cpp**, that demonstrates the interactive generation of graphical vertices and edges of varying thicknesses is available on the course website.

Zip-compress your entire program folder and place it on your dropbox by the deadline mentioned above.

*You must write your own code on this assignment, with adequate explanatory documentation (i.e., a paragraph at the top of each program file, including your name and a description of the file’s contents, and a sentence preceding each function, describing the function’s purpose).  
Obtaining code assistance from any outside source is considered academic misconduct.  
The only person permitted to see your code prior to the assignment deadline is the instructor.*