# Program 3: Dijkstra's algorithm

**Due** Feb 15, 2020 by 11:59pm **Points** 100 **Submitting** a file upload
**File Types** zip

In this program, you will implement Dijkstra's algorithm for shortest paths using an adjacency list representation.

### The shortest straw

Your program will read in a graph from a file, compute the shortest path between every pair of vertices (where they exist), output a shortest path table, and output individual paths when requested.

The file format will be as follows. The first line will contain the number of vertices n. Following are text descriptions of the vertices 1 through n (50 chars max length). After that, each line consists of 3 ints representing an edge. If there is a directed edge from vertex 1 to vertex 2 with a weight of 10, the line will be: 1 2 10. A zero for the first integer signals the end of the data. This is a valid example:

```
4
Olson’s office
Stiber's office
STEM office
The Commons
1 2 10
1 3  5
2 4 10
2 1 15
3 1  5
3 4 20
0 0  0
```

You will have a `findShortestPath` method that computes all shortest paths. A subsequent call to `displayAll` will output a table formatted as follows:

```
Description                 From  To    Dist  Path
Olson’s office
                            1     2     10    1 2
                            1     3     5     1 3
                            1     4     20    1 2 4
Stiber's office
                            2     1     15    2 1
                            2     3     20    2 1 3
                            2     4     10    2 4
STEM office
                            3     1     5     3 1
                            3     2     15    3 1 2
                            3     4     20    3 4
The Commons
                            4     1     --
```

```
4     2     --
4     3     --
```

The output for a single detailed path should have a similar format, but it should also include the location descriptions on additional lines. For a call to `G.display(2, 3);`, the output for this graph should be:

```
2     3     20    2 1 3
Stiber's office
Olson's office
STEM office
```

### Final thoughts

- As mentioned above, you can assume that the input is properly formatted. Here is a data file with the graph above: **HW3.txt (https://canvas.uw.edu/courses/1369252/files/61301562/download?wrap=1) (https://canvas.uw.edu/courses/1369252/files/61301562/download?download_frd=1)** (the Canvas preview of this file may make it look like there are periods in the file - there are not).
- Use an adjacency list to store the edges. This will require dynamic memory. Do not use vectors to store the edge lists.
- The graph will have no more than 100 vertices.
- Your class should have the following *public* methods: constructor, copy constructor, destructor, `operator=`, `buildgraph(ifstream &)`, `insertEdge(int, int, int)`, `removeEdge(int, int)`, `findShortestPath()`, `displayAll()`, `display(int, int)`.
- For `insertEdge`, replace any previous edge that existed between the two vertices.
- Use recursion, not a container to display a path. Remember that you work backwards from the destination to the source to recover the path.
- Your code should compile with this driver: **HW3.cpp (https://canvas.uw.edu/courses/1369252/files/61710522/download?wrap=1) (https://canvas.uw.edu/courses/1369252/files/61710522/download?download_frd=1)**
- I've provided a start to your `Graph` class here: **Graph.h (https://canvas.uw.edu/courses/1369252/files/60815784/download?wrap=1) (https://canvas.uw.edu/courses/1369252/files/60815784/download?download_frd=1)** , **Graph.cpp (https://canvas.uw.edu/courses/1369252/files/60815779/download?wrap=1) (https://canvas.uw.edu/courses/1369252/files/60815779/download?download_frd=1)**
- The header file provides you with a `Table` struct like in the notes and a 2D array of `Table` elements, which is necessary to record all shortest paths (not just a single source).
- At each step of Dijkstra's algorithm, you will need to determine which vertex to visit next. The best way to accomplish this is using a priority queue (heap). However, I recommend starting with a less efficient technique that scans the entire array of vertices for the unvisited vertex with the lowest weight path. This results in an $O(n^2)$ algorithm for a single source (if done correctly). You will run the algorithm using each vertex as the source.

### Relevant module goals

- Be able to implement shortest path algorithms in unweighted and positive-weighted directed graphs using either adjacency matrices or adjacency lists