# Shortest Path Algorithm with Real World Data

**Brad Peterson - Weber State University**

**Goal:** Use Dijkstra's shortest path algorithm with realistic data on a graph data structure that is both fast and avoids memory waste.

This assignment starts with a functioning shortest path algorithm using a 2D array to store graph data. This 2D array is great for a small number of nodes, but wasteful at thousands of nodes as it stores which nodes don't connect with each other. The assignment starts with support for a CSR graph data structure, then adds supporting functions, then modifies Dijkstra algorithm for the CSR graph, and then determines the exact node-by-node path for a shortest path.

**CSR Review:**

CSR graph storage allows leaving out unconnected and unnecessary node data. For example, the cost for a node from itself to itself is typically zero. But this zero doesn't need to be specified, it's implied. Likewise, a lack of connection between two nodes doesn't need to be specified. For example, observe the waste in the 2D array approach for a graph with 5 nodes and 10 edges

```
      0         1         2         3         4
    +-----------------------------------------
  0 | 0         16        99999999  2         3
  1 | 99999999  0         5         99999999  99999999
  2 | 99999999  3         0         99999999  99999999
  3 | 99999999  12        99999999  0         7
  4 | 99999999  10        4         5         0
```

This graph can be represented in the following CSR data structure:

```
             [0][1][2][3][4][5][6][7][8][9][10]
graphWeights: 16 2  3  5  3  12 7  10 4  5
     columns: 1  3  4  2  1  1  4  1  2  3
         row: 0  3  4  5  7  10
```

These arrays are not intuitive at first glance. To obtain an edge and its weight, such as from node *3* to node *4*, perform the following steps. Take the starting node (*3*), and go to that index in the row array (`row[3] = 5`). Add one to starting node index value (*3+1*), and go to that index in the row array, (`row[3+1] = 7`). The results of *5* and *7* now refer to indexes *[5, 7)* in the `columns` array, meaning indexes *5* and *6* but not *7* in the `graphWeights` and `columns` array refer to edges coming from node *3*. The first edge is `column[5] = 1`, or in other words, node *3* connects to node *1*. That doesn't work, the goal was to find an edge from node *3* to node *4*, and this is an edge for node *3* to node *1*. So next try `column[6] = 4`. That *4* the target destination. This edge is for node *3* to node *4*. Next, obtain the weight, `graphWeights[6] = 7`. In summary, an edge exists from node *3* to node *4* with the weight of *7*.

Consider going from node *4* to node *3*. First, `row[4] = 7` and `row[4+1] = 10`. So, look in *[7, 10)*. The first destination node from node *4* is `column[7] = 1`, which doesn't match. The second destination node is `column[8] = 2`, that doesn't work. The third is `column[9] = 3`. Found it. Get the weight, `graphWeights[9] = 5`. In summary, an edge exists from *3* to *4* and has a weight of *5*.

Consider node *0* to node *2*. First, `row[0] = 0`, and `row[0+1] = 3`. Look in *[0, 3)*. The destination `column[0] = 1`, that's not it. The destination `column[1] = 3`. That's not it. The destination `column[2] = 4`. That's not it. All

options are exhausted. That means an edge between the two nodes doesn't exist.

**Instructions:**

Your goal is implementing all TODOs in the assignment. Specifically:

- Initialize arrays in `createCSRArrays()`
  - Allocate five arrays: `graphWeights`, `columns`, `row`, `pathCost`, and `thisCameFrom`.
  - Load the `graphWeights`, `columns`, and `row` arrays with correct values. All graph information is found in the `edges` collection.
  - Initialize every element of the array `pathCost` to `LARGE_NUMBER`. Initialize every element of the array `thisCameFrom` to `-1`.
- Add code for `getEdgeWeight()`
  - Determine the weight for the edge connecting `sourceIndex` and `destIndex` and then return it. If no edge exists, return `LARGE_NUMBER`.
- Add code to `shortestPath()`
  - Replace the `weights[][]` 2D array with a function call `getEdgeWeight()` that uses as arguments the source node and the destination node. The algorithm has three such instances to replace.
  - Every time `pathCost` is updated with a new value for its destination, also update `thisCameFrom`. For example, if the code updates path at node `5` by executing `path[5] = getEdgeWeight(3,5)`, then `thisCameFrom[5] = 3`. Or in other words, the path to node `5` came from node `3`.
- Add code to `getPath()`
  - The goal here is following the `thisCameFrom` array from destination back to source. Then return a string indicating the path. The format of the string must be: `"index0 -> index1 -> index2"`. For example, a path of `0` to `4` to `2` to `1` would return the string: `"0 -> 4 -> 2 -> 1"`.
  - The logic for this function is a bit tricky. If the path from node `0` to node `1` requires the path `0` to `4` to `2` to `1`, then the logic should start at `thisCameFrom[1]`, see value `2`, go to `thisCameFrom[2]`, see value `4`, go to `thisCameFrom[4]`, see value `0`, and stop as it's the starting index. Note that this logic finds the path from destination to source, so it needs to be reversed prior to building the string.
- Add code to `deleteArrays()`
  - Reclaim the five arrays.

This assignment uses the file `rome99.gr`. The unit tests load all the file data for you into the `edges` collection. CMake should handle placing the file next to the executable correctly regardless of IDE or build process. If not, contact me.