FINDING THE SHORTEST TRAVEL TIME BETWEEN TWO NORTHEAST U.S. TRAIN STATIONS

BY:
MATTHEW KOSER
JONATHAN HENRY

CS 253-03
Central Connecticut State University

Description:

The Floyd-Warshall Algorithm is an incredibly simple but widely used algorithm used to find the shortest path between all vertices in a weighted and directed graph, which may contain either positive or negative edge weights. We have implemented the edges on our graph where each edge (u,v) corresponds to the amount of time traveled in between train stations on a direct trip instead of a distance traveled. Therefore our algorithm calculates the shortest time traveled between all given vertices in the graph.

Implementation:

The pseudocode for the Floyd-Warshall algorithm is as follows:

Input: A directed graph $G = (V,E)$, where $V = \{1,2, \ldots, n\}$ and a cost matrix C[i,j]

Output: Cost matrix A[1...n, 1...n] where A[i,j] is the cost of the cheapest path from i to j.

```
let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
for each edge (u,v)
  dist[u][v] ← w(u,v)  // the weight of the edge (u,v)
for each vertex v
  dist[v][v] ← 0
for k from 1 to |V|
  for i from 1 to |V|
```

```
    for j from 1 to |V|
      if dist[i][j] > dist[i][k] + dist[k][j]
        dist[i][j] ← dist[i][k] + dist[k][j]
      end if
```

This algorithm utilizes transitive closure as a method to find the shortest time possible between each pair of vertices. To understand this, let's take a closer look at the pseudo code.The crux of this algorithm starts at the 3 loops. These loops utilize 3 counter variables that go up to the max number of vertices . This algorithm when ran sets i, j, and k equal to 0. We have 11 vertices so the i j and k will loop 11 times.

K = 0 1 2 3 4 5 6 7 8 9 10

I = 0 1 2 3 4 5 6 7 8 9 10

J = 0 1 2 3 4 5 6 7 8 9 10

For every j iteration(up to 10), K & i =0. If there exists a distance between dist[i][k] + dist[k][j] which is less than dist[i][j], then you have essentially found another path to dist[i][j]who edges' weights combined are less than the existing path between [i][j]. If this condition is met, we update dist[i][j] to the new shorter path. In the case of our program, we have a path that takes less "time" (Not run time, the edge weights correspond to the time spent on the train rather than distance traveled). Although we have found a 'shorter' path, we do not know if we have found the 'shortest' path. There may be up to 121 edges in our graph because we have 11 vertices.(V^ 2). After J finishes 11 iterations: k = 0 (stays the same), i = 1, j = 0. J will now again iterate 11 times again.

Moreover the if statement will again evaluate a shorter path, and will update the matrix if the if conditions are met. When i iterates 11 times(and for every iteration j cycles 11 times), the for loop will end and k will equal 1 (k=1). After this happens, This entire process will repeat with: k = 1, i = 0, j = 0. Now after the last iteration we have now found and returned an adjacency matrix, where graph[a][b] is the shortest path between these two nodes.

The property that connects the nodes that have not been defined with an edge is called Transitive Closure, a way of finding an edge between two vertices in a graph when there is another path to the vertices. As an example, in our data, Boston is not connected to New York City, however Boston is connected to Hartford, which is connected to New York City. Since the two cities are not connected, the algorithm will initialize this value to 999999(essentially INF), higher than any other value in the data. This property of Transitive Closure will connect Boston to New York City with an edge weight of (Boston -> Hartford) + (Hartford -> NYC).

We chose to implement the Floyd-Warshall algorithm on the Amtrak train system in the Northeast section of the United States. In order to do this we went to the Amtrak Online site and looked at the map of the NorthEast part of the country. This gave us a good idea of where to start with our data collection. As an added bonus, most of the stations typically have two sets of tracks running in parallel, making it easy to see the data for the trip both coming to and leaving from. We used the site ‘wanderu.com’ that uses

Amtrak train data, making sure to only include “Direct Trips”. We then used the time necessary to reach the destination as our edge weight. This data we then recorded onto a spreadsheet before simplifying it into code.

Index	Connections(Minutes)		
Hartford: 0	Randolph(VT):285	NYC: 150	Providence: 105
NYC: 1	Hartford(CT): 150	Trenton(NJ): 60	Penn: 87
Boston: 2	Durham(NH): 85	Portland(ME): 150	Randolph(VT): 205
Trenton: 3	Hartford(CT): 225	Philadelphia(PA): 30	Baltimore: 110
D.C: 4	Hartford(CT): 400	Philadelphia(PA): 120	Baltimore: 30
Providence: 5	Hartford(CT): 105	Trenton(NJ): 280	Philidelphia(PA):290
Portland: 6	Durham(NH): 65	Boston, (MA): 85	
Durham: 7	Portland(ME): 65	Hartford(CT): 265	
Randolph: 8	Boston(MA):225	Hartford(CT): 285	Baltimore(ML): 626
Philadelphia: 9	Trenton(NJ): 30	NYC(NY): 87	Boston(MA): 360
Baltimore: 10	Trenton(NJ): 110	NYC(NY): 180	Randolph(VT):626

In order to store our data, we used a set of dictionaries. We used the name of the station as a string, and inserted its index in the array as the data. This was done in order to smooth the process of converting the user input to an array lookup. When a user inputs a station name, for example “Baltimore”, the program will convert the string into uppercase (“BALTIMORE”), and search the dictionary for the data that corresponds to the key, which is its array index. It will then use that index and the index of the destination station in the result graph to find the value of the shortest time.

The following matrix represents the edge weights and connections of the graph prior to running the Floyd-Warshall algorithm on it:

```
Adjacency matrix - INF represents no edge:
0.0  150.0  120.0  225.0  400.0  105.0  INF   INF   285.0  285.0  365.0
150.0  0.0  INF   60.0  215.0  186.0  INF   INF   443.0  87.0  180.0
120.0  INF   0.0  340.0  INF   40.0  150.0  85.0  225.0  360.0  684.0
225.0  60.0  340.0  0.0  150.0  280.0  INF   INF   INF   30.0  110.0
400.0  215.0  INF   150.0  0.0  450.0  INF   INF   669.0  120.0  30.0
105.0  186.0  40.0  280.0  INF   0.0  INF   INF   INF   290.0  350.0
INF   INF   85.0  INF   INF   INF   0.0  65.0  INF   INF   INF
265.0  INF   INF   INF   INF   INF   65.0  0.0  INF   INF   INF
330.0  444.0  225.0  517.0  669.0  INF   INF   INF   0.0  548.0  626.0
285.0  87.0  360.0  30.0  120.0  290.0  INF   INF   INF   0.0  90.0
365.0  180.0  684.0  110.0  INF   408.0  INF   INF   626.0  90.0  0.0
```

For clarity - the following is a line by line list of connected vertices for out data set.

Station HARTFORD (index 0) is connected to: NEW YORK CITY, BOSTON, TRENTON, WASHINGTON DC, PROVIDENCE, RANDOLPH, PHILADELPHIA, BALTIMORE,

Station NEW YORK CITY (index 1) is connected to: HARTFORD, TRENTON, WASHINGTON DC, PROVIDENCE, RANDOLPH, PHILADELPHIA, BALTIMORE,

Station BOSTON (index 2) is connected to: HARTFORD, TRENTON, PROVIDENCE, PORTLAND, DURHAM, RANDOLPH, PHILADELPHIA, BALTIMORE,

Station TRENTON (index 3) is connected to: HARTFORD, NEW YORK CITY, BOSTON, WASHINGTON DC, PROVIDENCE, PHILADELPHIA, BALTIMORE,

Station WASHINGTON DC (index 4) is connected to: HARTFORD, NEW YORK CITY, TRENTON, PROVIDENCE, RANDOLPH, PHILADELPHIA, BALTIMORE,

Station PROVIDENCE (index 5) is connected to: HARTFORD, NEW YORK CITY, BOSTON, TRENTON, PHILADELPHIA, BALTIMORE,

Station PORTLAND (index 6) is connected to: BOSTON, DURHAM,

Station DURHAM (index 7) is connected to: HARTFORD, PORTLAND,

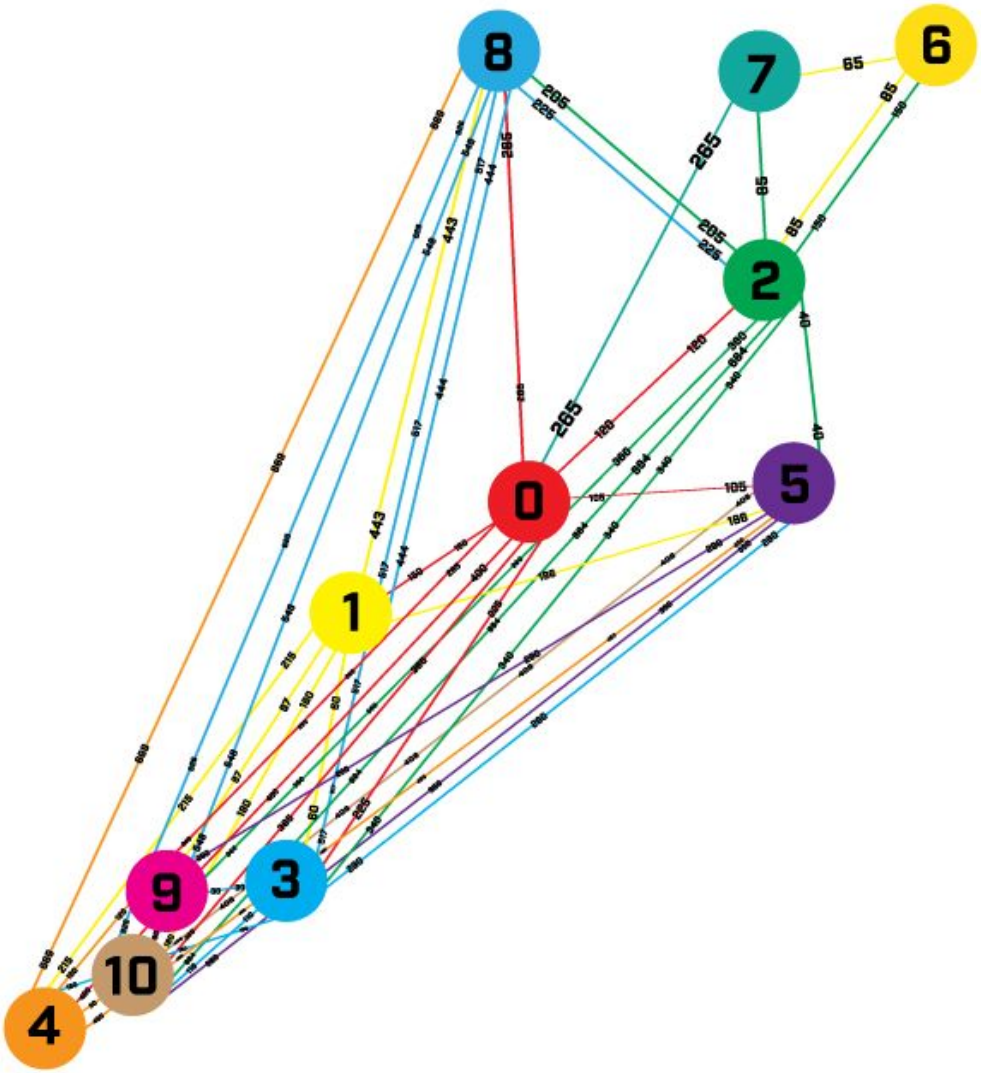
Station RANDOLPH (index 8) is connected to: HARTFORD, NEW YORK CITY, BOSTON, TRENTON, WASHINGTON DC, PHILADELPHIA, BALTIMORE,

Station PHILADELPHIA (index 9) is connected to: HARTFORD, NEW YORK CITY, BOSTON, TRENTON, WASHINGTON DC, PROVIDENCE, BALTIMORE,

Station BALTIMORE (index 10) is connected to: HARTFORD, NEW YORK CITY, BOSTON, TRENTON, PROVIDENCE, RANDOLPH, PHILADELPHIA,

To expand upon the INF edges, the main vertices that contain these were cities in Maine, New Hampshire and Vermont (vertexes 6, 7, and 8 respectively). These states do not have as large as a population as the rest of the states and thus the train system is not as robust.

The following is a visualization of our data pre Floyd-Warshall application



Sample Outputs:

```
What would you like to do?

1. Calculate the travel time between two stations
2. Show list of all avaliable stations.
3. Exit the program
2

Here is every station able to be used for evaluation:

HARTFORD
WASHINGTON DC
NEW YORK CITY
TRENTON
PORTLAND
RANDOLPH
PHILADELPHIA
PROVIDENCE
BOSTON
BALTIMORE
DURHAM


What would you like to do?

1. Calculate the travel time between two stations
2. Show list of all avaliable stations.
3. Exit the program
1

Please enter where you would like to start:
Durham
Please enter your destination:
Washington dc
The time it takes to travel from Durham to Washington dc is: 9 hours and 43 minutes
What would you like to do?
```

```
Please enter where you would like to start:
philadelphia
Please enter your destination:
boston
The time it takes to travel from philadelphia to boston is: 5 hours and 13 minutes
```

```
Please enter where you would like to start:
hartford
Please enter your destination:
trenton
The time it takes to travel from hartford to trenton is: 3 hours and 30 minutes
```

Conclusion:

In comparison from the Floyd-Warshall algorithm to Dijkstra's Algorithm for finding the shortest path between nodes, we chose to implement the Floyd-Warshall algorithm due to both its simplicity and our curiosity in the subject. In a dense graph with with most pairs of vertices are linked, it will be optimal to use Floyd-Warshall over Dijkstra. But in a sparse graph (lots of vertices not connected by edges) with positive weights it will be more time efficient to do Dijkstra's algorithm multiple times. We had previously intended to collect a larger data set, however we found our number of nodes to be sufficient for demonstration. The Floyd-Warshall algorithm has a time complexity of O(N^3), while Dijkstra's algorithms runtime efficiency is O(Elog(v)), which repeated becomes O(|E||V|+|V|^2log(|V|)). For our specific data set this gives us (11)^3 vs (|68||11|+|11|^2log(|11|)), which equals 1331 vs 874.008. Due to the ratio of edges to vertices based on our data it would actually be more efficient to repeat Dijkstra's for 11 iterations, however when using Floyd-Warshall's, we only have to run the algorithm once to obtain an adjacency matrix which contains all shortest paths.

References:

“2.1.3.” ***Handbook of Graph Theory: Edited by Jonathan L. Gross, Columbia University New York, USA ; Jay Yellen, Rollins College Winter Park, Florida, USA ; Ping Zhang, Western Michigan University, Kalamazoo, USA*****,***

by Jonathan L. Gross et al., CRC Press, 2014.

Hochbaum, Dorit. “Graph Algorithms and Network Flows.” IEOR 266.

Nuutila, Esko. “Efficient Transitive Closure Computation in Large Digraphs.” ***Helsinki University of Technology*****, 1995.**