Your Project Report should have the following identifiable sections.:

Title page Team Name: NonPdynomials

CS5592, Fall Semester 2016: Predicting shortest path in the presence of congestion

We understand and have adhered to the rules regarding student conduct. In particular, any and all material, including algorithms and programs, have been produced and written by members of our team. Any outside sources that we have consulted are free, publicly available, and have been appropriately cited. We understand that a violation of the code of conduct will result in a zero (0) for this assignment, and that the situation will be discussed and forwarded to the Academic Dean of the School for any follow up action. It could result in being expelled from the university.

PRASANNA MUPPIDI	16222395	11/26/16
//Your Name://	//Your UMKC-ID://	//This Date//
~~HARSHINI~~ MEDIKONDA	16209380	11/26/16
//Your Name://	//Your UMKC-ID://	//This Date//
SRAVANI MURAKONDA	16221927	11/26/16
//Your Name://	//Your UMKC-ID://	//This Date//
//Your Name://	//Your UMKC-ID://	//This Date//

Please print and sign this page. Subsequently scan it back in and attach it to your report.

Contents

INTRODUCTION:

When planning a road trip from a source to destination, most of us would want to go in the shortest path possible. We have several algorithms to calculate the shortest path from our source to destination. But we are not the only one on the road. There are many other people who would want to travel in the same route we chose to travel or might use the same path though the source and destination are different to get the shortest path. This results in severe congestion on the road. We will have waiting time in addition to the actual shortest time that we normally have if we were the only car on the road. Now, we have implemented an algorithm which considers the congestion on the road and gives us the best possible results for the shortest path.

DESIGN AND ANALYSIS OF ALGORITHMS:

A) DATA STRUCTURES:

Coming to the data structures used, we used adjacency matrix and 3D array. The adjacency matrix is used to store edge matrix, flow matrix, capacity matrix. We stored the load matrix that we calculated based on the flow, in adjacency matrix. We used adjacency matrix for congestion matrix as well. On the other hand, for the path matrix, the adjacency matrix is not just enough if we need to store the nodes that have to be visited for the shortest path at the s-t pair.

It can be stored in adjacency matrix if we store it in a string format. But, it is not a dynamic and optimal approach to do. So, a 3D array approach is one of the best data structures to handle this.

B) EVIDENCE OF APPROACH:

For a given a and b, the algorithm implemented by our team finds the initial shortest paths, loading on each link, shortest predicted paths and actual path lengths correctly. This has been explained well in the further sectionsbelow with an example. We used Floyd-Warshall algorithm to find the shortest path and also for path reconstruction as a part of the prediction and implementation.

APPROACHES:

Approach 1:

We have Edge matrix, Flow matrix and Capacity Matrix as our inputs. We have calculated all-pairs-shortest-paths on the Edge matrix using Floyd Warshall algorithm. We have calculated a Path matrix and a Hop matrix based on the all-pairs-shortest-paths matrix.

Coming to the Flow matrix, we have sent one car at one time between two nodes and have repeated the process until all the cars in the flow are accommodated. In each iteration, the flow on each path is made one. The load is calculated based on the flow between the nodes and all-pairs-shortest path chosen. Since in each iteration, the load is less when compared to the actual complete load, there are high chances that the load on each path is less than the capacity of each path. So, the congestion on each path is less which in turn makes the waiting time much less.

Approach 1 Algorithm:

Congestion-shortest-path (File F)

1. *Read data into E, C, F //Edge, Capacity, Flow matrix*
2. *N = Number of Nodes*
3. *S = Source, D = Destination*
4. *MyDist $^{[0]}$ = E*
5. *for k = 1 to N*
    a. *let MyDist$^{(1)}$ = ($d_{ij}^{(k)}$) be a new matrix*
    b. *for i = 1 to N*
        i. *for j = 1 to N*
            $d_{ij}^{(k)} = min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
            $parent_{ij}^{(k)} = parent_{kj}^{(k-1)}$
6. *for i = 1 to N*
    a. *for j = 1 to N*
        i. *b=j*
        ii. *for k = 1 to N*
            insert parent[i][b] insert i in next location
7. *Initialize $F_{ij}$to 1*
8. *for i = 1 to N*
    a. *for j = 1 to N*
        i. *for k = $Hop_{ij}$to 2*
            Calculate L_{ij}using $path_{ijk}$and F_{ij}
9. *for i = 1 to N*
    for j = 1 to N
        *$G_{ij} = (C_{ij} + 1)/(C_{ij} + 1-L_{ij})$ ^ (Total $Flow_{ij}$/ L_{ij}) * E_{ij}*
10. *Calculate Final_delay[i][j] using $G_{ij}$and $Path_{ijk}$*
11. *return Final_delay[S][D]*

Analysis of the above algorithm:

The running time of the above algorithm is determined by the above lines 1 to 11

Lines 1 to 4 take O(1) time. Floyd warshall algorithm is determined by the triple nested for loops of line 5 because the execution of inner operations take O(1) time. So, total running time of line 5 is $O(n^3)$ time. Line 6 takes $O(n^3)$ time to construct path between every (i,j) pair. Line 7 takes O(1) time. Line 8 takes $O(n^3)$ time to calculate load L_{ij}. To calculate congestion edge matrix G_{ij} we need $O(n^2)$ time i.e., line 9. To calculate all-pairs-shortest-paths using previously determined $path_{ij}$ and G_{ij} takes $O(n^3)$ time i.e., line 10. Line 11 takes O(1) time to return the shortest path with congestion between source and destination i.e., S and D.

KEY IDEA IN OUR IMPLEMENTATION:

The key idea in our implementation is:

When there is a single car in the network, the edge weights are given by some integer values represented by an edge matrix E.

$$\begin{bmatrix} 0 & a & b \\ c & 0 & d \\ e & f & 0 \end{bmatrix}$$

The above edge matrix is when there is only single car on the road.

The congestion dependent travel times is given by

$$\mathbf{G}[i,j] = \left(\frac{\mathbf{C}[i,j]+1}{\mathbf{C}[i,j]+1-\mathbf{L}[i,j]}\right)\mathbf{E}[i,j]$$

So, when we send a load of **L [i, j]**, **G [i, j]** is the new congested path. This is the time that the cars would take to reach the destination when there is load. And this is the waiting time for the remaining cars in the load matrix. So, this becomes the edge matrix E

So, G becomes the new E.

So, G = E' = E_1

In our approach, we are sending the same load every time.

We calculated the load based on how many times that edge is used when the shortest path is opted.

So, using E_1 as the new edge matrix, we calculate the congested edge weights or travelled time for 2nd set of cars(travelled 2nd time) is

E_2 = $G[i,j] = \left(\frac{C[i,j]+1}{C[i,j+1-L[i,j]}\right)E_1$

Then for the 3rd set of cars

E_3 = $G[i,j] = \left(\frac{C[i,j]+1}{C[i,j+1-L[i,j]}\right)E_2$

Then for the 4th set of cars,

E_4 = $G[i,j] = \left(\frac{C[i,j]+1}{C[i,j+1-L[i,j]}\right)E_3$

Substituting the values of E_3

E_4 = $G[i,j] = \left(\frac{C[i,j]+1}{C[i,j+1-L[i,j]}\right)\left(\frac{C[i,j]+1}{C[i,j+1-L[i,j]}\right)E_2$

Substituting the values of E_2 and then E_1

E_4 = $G[i,j] = \left(\frac{C[i,j]+1}{C[i,j+1-L[i,j]}\right)\left(\frac{C[i,j]+1}{C[i,j+1-L[i,j]}\right)\left(\frac{C[i,j]+1}{C[i,j+1-L[i,j]}\right)E_1$

$= \left(\frac{C[i,j]+1}{C[i,j+1-L[i,j]}\right)\left(\frac{C[i,j]+1}{C[i,j+1-L[i,j]}\right)\left(\frac{C[i,j]+1}{C[i,j+1-L[i,j]}\right)\left(\frac{C[i,j]+1}{C[i,j+1-L[i,j]}\right)E$

$= \left(\frac{C[i,j]+1}{C[i,j+1-L[i,j]}\right)^4 * E$

So, the power is the number of iterations we made.

Approach 1 Implementation Output:

Input Edge Matrix:

$$\begin{bmatrix} 0 & 7 & na & 7 & na & 9 \\ na & 0 & 5 & na & 10 & 3 \\ 9 & 10 & 0 & 8 & 4 & 6 \\ 9 & 4 & 2 & 0 & na & na \\ 3 & 5 & 10 & 10 & 0 & na \\ na & 5 & 8 & 10 & na & 0 \end{bmatrix}$$

Input flow matrix:

$$\begin{bmatrix} 0 & 9 & 11 & 12 & 8 & 12 \\ 18 & 0 & 15 & 10 & 17 & 18 \\ 17 & 18 & 0 & 14 & 10 & 10 \\ 17 & 8 & 10 & 0 & 17 & 18 \\ 15 & 9 & 12 & 14 & 0 & 16 \\ 18 & 16 & 15 & 8 & 9 & 0 \end{bmatrix}$$

Input Capacity matrix:

$$\begin{bmatrix} 0 & 13 & na & 33 & na & 20 \\ na & 0 & 67 & na & 5 & 55 \\ 5 & 5 & 0 & 32 & 134 & 17 \\ 23 & 34 & 55 & 0 & na & na \\ 68 & 47 & 20 & 14 & 0 & na \\ na & 16 & 44 & 16 & na & 0 \end{bmatrix}$$

New Flow Matrix:

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

New Load Matrix:

$$\begin{bmatrix} 0 & 1 & na & 3 & na & 1 \\ na & 0 & 4 & na & 0 & 3 \\ 0 & 0 & 0 & 2 & 9 & 1 \\ 1 & 2 & 4 & 0 & na & na \\ 4 & 3 & 1 & 1 & 0 & na \\ na & 1 & 3 & 1 & na & 0 \end{bmatrix}$$

Congestion Matrix:

$$\begin{bmatrix} 0 & 13.63 & na & 18.16 & na & 16.16 \\ na & 0 & 12.32 & na & 10 & 7.78 \\ 9 & 10 & 0 & 16.94 & 10.83 & 10.62 \\ 18.55 & 8.59 & 4.68 & 0 & na & na \\ 8.27 & 12.6 & 17.95 & 26.27 & 0 & na \\ na & 13.18 & 21.01 & 16.24 & na & 0 \end{bmatrix}$$

Actual delay Matrix:
$$\begin{bmatrix} 0 & 13.64 & 22.85 & 18.16 & 33.85 & 16.16 \\ 31.69 & 0 & 12.41 & 29.35 & 23.41 & 7.79 \\ 19.28 & 23.60 & 0 & 16.94 & 11.00 & 10.63 \\ 18.55 & 8.60 & 4.69 & 0 & 15.68 & 16.38 \\ 8.28 & 12.60 & 17.96 & 26.27 & 0 & 20.39 \\ 40.29 & 13.19 & 21.02 & 16.24 & 32.01 & 0 \end{bmatrix}$$

Approach 2:

One of the other approaches we have considered is to send 25% of the flow at one time. We have send the entire flow in 4 parts. We need to do 4 iterations in order to accommodate the entire flow.

Considering the above example,

Input Edge Matrix:
$$\begin{bmatrix} 0 & 7 & na & 7 & na & 9 \\ na & 0 & 5 & na & 10 & 3 \\ 9 & 10 & 0 & 8 & 4 & 6 \\ 9 & 4 & 2 & 0 & na & na \\ 3 & 5 & 10 & 10 & 0 & na \\ na & 5 & 8 & 10 & na & 0 \end{bmatrix}$$

Input flow matrix:
$$\begin{bmatrix} 0 & 9 & 11 & 12 & 8 & 12 \\ 18 & 0 & 15 & 10 & 17 & 18 \\ 17 & 18 & 0 & 14 & 10 & 10 \\ 17 & 8 & 10 & 0 & 17 & 18 \\ 15 & 9 & 12 & 14 & 0 & 16 \\ 18 & 16 & 15 & 8 & 9 & 0 \end{bmatrix}$$

Input Capacity matrix:
$$\begin{bmatrix} 0 & 13 & na & 33 & na & 20 \\ na & 0 & 67 & na & 5 & 55 \\ 5 & 5 & 0 & 32 & 134 & 17 \\ 23 & 34 & 55 & 0 & na & na \\ 68 & 47 & 20 & 14 & 0 & na \\ na & 16 & 44 & 16 & na & 0 \end{bmatrix}$$

We divide the above flow matrix into four parts.

F1(Flow matrix in 1st iteration)=
$$\begin{bmatrix} 0 & 3 & 3 & 3 & 2 & 4 \\ 5 & 0 & 4 & 3 & 5 & 5 \\ 5 & 5 & 0 & 4 & 3 & 3 \\ 5 & 2 & 3 & 0 & 5 & 5 \\ 4 & 3 & 3 & 4 & 0 & 4 \\ 5 & 4 & 4 & 2 & 3 & 0 \end{bmatrix}$$

F2(Flow matrix in 2nd iteration)=
$$\begin{bmatrix} 0 & 2 & 3 & 3 & 2 & 4 \\ 5 & 0 & 4 & 2 & 4 & 5 \\ 4 & 5 & 0 & 4 & 2 & 2 \\ 4 & 2 & 2 & 0 & 4 & 5 \\ 4 & 2 & 3 & 4 & 0 & 4 \\ 5 & 4 & 4 & 2 & 2 & 0 \end{bmatrix}$$

F3(Flow matrix in 3rd iteration)= $\begin{bmatrix} 0 & 2 & 3 & 3 & 2 & 4 \\ 4 & 0 & 4 & 2 & 4 & 4 \\ 4 & 4 & 0 & 3 & 2 & 2 \\ 4 & 2 & 2 & 0 & 4 & 4 \\ 4 & 2 & 3 & 3 & 0 & 4 \\ 4 & 4 & 4 & 2 & 2 & 0 \end{bmatrix}$

F4(Flow matrix in 4th iteration)= $\begin{bmatrix} 0 & 2 & 2 & 3 & 2 & 4 \\ 4 & 0 & 3 & 2 & 4 & 4 \\ 4 & 4 & 0 & 3 & 2 & 2 \\ 4 & 2 & 2 & 0 & 4 & 4 \\ 3 & 2 & 3 & 3 & 0 & 4 \\ 4 & 4 & 3 & 2 & 2 & 0 \end{bmatrix}$

Likewise, we get 4 load matrices corresponding to the above Flow matrices.

L1(Load matrix in 1st iteration)= $\begin{bmatrix} 0 & 3 & na & 8 & na & 4 \\ na & 0 & 17 & na & 0 & 14 \\ 0 & 0 & 0 & 7 & 38 & 3 \\ 5 & 7 & 8 & 0 & na & na \\ 19 & 12 & 3 & 4 & 0 & na \\ na & 4 & 12 & 2 & na & 0 \end{bmatrix}$

L2(Load matrix in 2nd iteration)= $\begin{bmatrix} 0 & 2 & na & 8 & na & 4 \\ na & 0 & 15 & na & 0 & 14 \\ 0 & 0 & 0 & 6 & 33 & 10 \\ 4 & 7 & 10 & 0 & na & na \\ 18 & 11 & 3 & 4 & 0 & na \\ na & 4 & 11 & 8 & na & 0 \end{bmatrix}$

L3(Load matrix in 3rd iteration)= $\begin{bmatrix} 0 & 2 & na & 8 & na & 4 \\ na & 0 & 14 & na & 0 & 12 \\ 0 & 0 & 0 & 5 & 30 & 10 \\ 4 & 7 & 10 & 0 & na & na \\ 16 & 10 & 3 & 3 & 0 & na \\ na & 4 & 10 & 8 & na & 0 \end{bmatrix}$

L4(Load matrix in 4th iteration)= $\begin{bmatrix} 0 & 2 & na & 8 & na & 4 \\ na & 0 & 13 & na & 0 & 12 \\ 0 & 0 & 0 & 5 & 30 & 10 \\ 4 & 7 & 10 & 0 & na & na \\ 15 & 10 & 3 & 3 & 0 & na \\ na & 4 & 9 & 8 & na & 0 \end{bmatrix}$

Congestion Matrix(After the 4 iterations): $\begin{bmatrix} 0 & 14.92 & na & 22.45 & na & 19.23 \\ na & 0 & 14.54 & na & 10 & 8.67 \\ 9 & 10 & 0 & 17.74 & 12.70 & 11.89 \\ 19.92 & 10.73 & 6.96 & 0 & na & na \\ 14.45 & 18.8 & 20.84 & 29.18 & 0 & na \\ na & 17.20 & 24.67 & 19.79 & na & 0 \end{bmatrix}$

Actual delay Matrix(After the 4 iterations): $\begin{bmatrix} 0 & 14.92 & na & 22.45 & na & 19.23 \\ 35.98 & 0 & 14.54 & 30.37 & 10 & 8.67 \\ 27.86 & 30.45 & 0 & 17.74 & 12.70 & 11.89 \\ 19.92 & 10.73 & 6.96 & 0 & 20.15 & 24.78 \\ 14.45 & 18.8 & 20.84 & 29.18 & 0 & 27.75 \\ 46.67 & 17.20 & 24.67 & 19.79 & 36.19 & 0 \end{bmatrix}$

Approach 3:

One of the other approaches is to take a **weighted flow matrix**.

An edge may be used as a part of the shortest path by 1 or more source-destination pairs. So, keeping in mind this point, we come up with the flow matrix and then calculate the load based on the flow. Here there are 3 considerations:

1) **When load is less than the capacity**:
   When the load is less than actual capacity at each edge, we are good to the send the whole load at once, but based on how much the load is less than the capacity of edge at each point, the congestion matrix varies. The values in congestion matrix tells us how much time it takes to reach a destination.

2) **When load is equal to the capacity**:
   When the load is equal to the actual capacity at each edge, we are good to the send the whole load at once, but since load is almost equal to the capacity of edge at each point, the values in the congestion matrix would be large. This would increase the congestion time from source to destination. This may happen at each edge or few edges where the capacity equals the load.

3) **When load is greater than the capacity**:
   When the load is greater than the actual capacity at each edge or any of the edges, we are load at once, instead we tend to send only the load that is equal to the cannot the whole capacity. We send the remaining load in the next iteration. The logic interferes when it is time to decide which load has to be sent among the actual load. It is further explained below:

The key point to consider here is to consider a load that is equal to the capacity, that too as per the edges contribution to the load for that edge. Hence if we have 200% load contributed by 3 edges while passing through their won shortest paths and 100% capacity. Suppose the three edges contributed as edge A of 100%, edge B of 50% and edge C of 50%. Then we would allocate load in such a way that the new load is considered as edge A of 50%, edge B of 25% and edge C of 25%. We would send the remaining load in the next iteration. We would

In this way, we will make sure that the load does not exceed capacity. The edge weights that are obtained by 1st iteration would be the edge weights for the 2nd iteration of cars. In this way, we would allocate as much as possible and reduce the number of iterations in this approach.

Algorithm:
Below is the algorithm assuming that we calculated the hop matrix, path matrix from the edge matrix given. Also, assuming that the flow matrix and capacity matrix are given.

1. Based on flow matrix and path matrix, calculate load matrix.
2. Compare the load and capacity matrices at each edge.
3. If the load matrix is less than or equal to the capacity matrix at each edge, then send the entire flow.
4. Else, at the edges where the load exceeds capacity, we take a weighted sum of loads contributed by each edge to that load such that the weighted sum of load equals capacity.
5. Then we calculate the G matrix, that is congestion matrix.
6. The new load matrix is the cars remaining after iteration.
7. The new congestion matrix becomes the edge matrix for next iteration.
8. Repeat steps 2 to 4 until load becomes zero at each edge.
9. At the end, we get the congestion matrix of the total flow.

We have chosen to implement **Approach 1** for the following reasons:

1. Approach 1 gives us the least waiting time among all the three approaches.
2. Approach 1 doesn't use any loops since we have derived a mathematical formula to calculate the final delay matrix values. Less complex code.
3. Implementation time for Approach 1 is much less when compared to the other approaches.
4. Optimal Congestion delay matrix with least running time.

C) Design and implementation decisions:

While implementing the algorithm that we proposed, we are sending the same amount of load each time. When it comes to the end, we may encounter a situation where we have less load than we tend to send each time. If we calculate the congestion for the same load, then it looks like we are sending the cars that doesn't exist on the road. So, we calculate the load separately for the left overs and then congestion from that. This gives us accurate and better results. This consideration resulted in slight improvement of the performance and reduction of congestion. So, **we could help avoid traffic.**

D) Load Matrix Calculation:

We obtain the shortest path from Floyd-Warshall algorithm. That is stored as a path matrix in a 3D array. Each edge is traversed once to see what routes are being used. If an edge (1,4) is being used while travelling from 2 to 5 and also used when travelling from 3 to 7. Then the load of (1,4) would be the sum of flow along (1,4) , (2,5) and (3,7). This is how the load matrix is calculated.

IMPLEMENTATION AND TESTING OF ALGORITHM:

IMPLEMENTATION:

LANGUAGE: C

We used C language to implement the algorithm because we have the flexibility of dynamic memory allocation, pointers, in C and the most important point to be mentioned is that we are comfortable with using C.

The implementation of our algorithm is described in the algorithm section.

Mathematical implementation:

Say, Total number of nodes as **N**

Edge matrix as **E[N][N]**

Flow matrix as **F[N][N]**

Load matrix as **L[N][N]**

Capacity matrix as **C[N][N]**

New flow matrix as **NF[N][N]**

New load matrix as **NL[N][N]**

Congestion matrix as **G[N][N]**

Final delay congestion matrix as **Final_delay[N][N]**

The number of iterations would be **Num_IT**

In the first iteration, as per our algorithm, all the elements in Flow matrix would be one.

i.e., **NF [i, j] = 1** and the new load matrix is calculated based on the all-pairs-shortest path. So, we get **NL [i, j]**

Using the below formula, we calculate the Congestion Matrix.

G [i, j] = ((C [i, j] + 1)/ (C [i, j] + 1 – L [i, j])) * E [i, j]

Let's assume **(C [i, j] + 1)/ (C [i, j] + 1 – L [i, j]))**as **coef**

G [i, j] = coef * E [i, j]

This calculated Congestion matrix becomes our new Edge matrix say **E1[i, j]**

E1[i, j] is the new edge matrix after our first iteration.

So, **E1[i, j] =G [i, j]**

Therefore, **E1[i, j] = coef * E [i, j]**

Since **F [i, j] = 1** for all the iterations, we will have the same coef for all the iterations.

Similarly, after the second iteration we have **E2[i, j] = coef * E1[i, j]**

Similarly, after the third iteration we have **E3[i, j] = coef * E2[i, j]**

and after **Num_IT** iterations we have **E_Num_IT [i, j] = coef * E_Num_IT-1[i, j]**

When we combine all the above equations, we get

E_Num_IT [i, j] = pow (coef, Num_IT) * E [i, j]

where **Num_IT=floor (L [i, j] /NL [i, j])**

We will still have some leftovers **if (L [i, j] % NL [i, j])! = 0)** since we have used a floor function to calculate the number of iterations.

Leftovers [i, j] = (L [i, j] -(NL [i, j] *Num_IT))

Leftovers_coef [i, j] = (C [i, j] + 1)/ (C [i, j] + 1 –Leftovers [i, j])

Our actual delay matrix after all the iterations would be as shown below.

Final_delay [i, j] = Leftovers_coef [i, j] * E_Num_IT [i, j]

If there are no leftovers **i.e., if (L [i, j] % NL [i, j]) = 0)** actual delay matrix would be

Final_delay [i, j] = E_Num_IT [i, j]

TESTING:

We tested our algorithm with several test cases. We calculated many of them by hand and cross validated it against the implementation.

We gave all the files uploaded in the blackboard as input to the program to calculate the shortest path. The obtained shortest path is cross validated and the results look good. The good thing about our implementation is the time taken is around 100 milliseconds even when we have 75 nodes and it is calculating shortest path between all pairs.

Manual Calculations for Approach 1:

Actual delay Matrix:

$$\begin{bmatrix} 0 & 13.64 & 22.85 & 18.16 & 33.85 & 16.16 \\ 31.69 & 0 & 12.41 & 29.35 & 23.41 & 7.79 \\ 19.28 & 23.60 & 0 & 16.94 & 11.00 & 10.63 \\ 18.55 & 8.60 & 4.69 & 0 & 15.68 & 16.38 \\ 8.28 & 12.60 & 17.96 & 26.27 & 0 & 20.39 \\ 40.29 & 13.19 & 21.02 & 16.24 & 32.01 & 0 \end{bmatrix}$$

PRE-IMPLEMENTATION ANALYSIS:

For finding the shortest path length using Floyd-Warshall, time complexity =O(n^3)

For the path reconstruction from the predecessor matrix, time complexity =O(n^3)

While reconstructing the path, we calculate hop matrix as well. So, there is no extra time complexity for hop matrix.

For calculating the load matrix using path and flow matrices, time complexity =O(n^3)

Calculating G- Matrix from the load matrix and formula, time complexity =O(n^2)

Calculating Actual delay Matrix from the load matrix and formula, time complexity =O(n^3)

As per our approach:

Calculating the congestion edge matrix from the mathematical formula we derived, time complexity =O(n^2)

Calculating Actual delay Matrix, time complexity =O(n^3)

So, the overall time complexity can be given as O(n^3).

Per our analysis of algorithm, it takes **O(n^3)** time which is equal to the program implementation time.

Ouput

COMPARISION and CONCLUSIONS:

The actual path lengths are not equal to predicted, the values are higher than predicted but better and consistent. The values of input and outputs are mentioned in 'OutputComparision.xlsx' file. In that file, the running time of the algorithm is also given for several 'n' values. When we doubled the input size, there is a very subtle difference in the running time, which is not even noticeable. The running time of the algorithm is around 120 milliseconds when we considered 75 nodes. When we consider small test cases, the running time is not even noticeable. The conclusion we can draw from the algorithm is that we get the best path with the best actual time.

Before the implementation of the algorithm, we came with some solutions that would take a higher coefficient of n^3 . But, upon several modifications and mathematical inductions, our strategy evolved into a better form, thus resulting in smaller value of coefficient for n^3.

Output Comparison table:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	No.	Filename	Nodes	Source	Destination	Given Example Strategy						SingleNodeAtATime				
2						Predicted Path	Actual Path	Hop Count	Time(Seconds)	Key-Basic-Operations		Predicted Path	Actual Path	Hop Count	Time	Key-Basic-Operations
3	1	input1.txt	6	5	3	10	23.33	1	0.000548	244		10	17.96	1	0.000622	1072
4	2	input2.txt	6	6	3	8	--nan	1	0.000588	244		8	23.83	1	0.000623	1072
5	3	N10bs.txt	10	6	4	296	5834.67	4	0.001414	714		296	778.72	4	0.001202	3352
6	4	N10cs.txt	10	1	2	104	3254.33	2	0.001551	714		104	271.48	2	0.001338	3189
7	5	N10ds.txt	10	8	2	282	4960	10	0.001296	2429		282	703.64	10	0.001256	5659
8	6	N10tb.txt	10	9	1	280	40680.5	3	0.001194	1478		280	774.44	3	0.002019	4347
9	7	N10ts.txt	10	9	8	-nan	-nan	0	0.001268	1478		-nan	-nan	0	0.001221	4337
10	8	N15as.txt	15	3	15	290	13439.59	5	0.002453	1914		290	785.84	5	0.002433	8112
11	9	N15bs.txt	15	15	1	6	97.64	3	0.002223	2014		6	14.97	3	0.002332	8076
12	10	N20as.txt	20	6	1	15	2406.82	6	0.003919	5154		15	44.15	6	0.004115	16219
13	11	N20bs.txt	20	19	1	276	6347.8	3	0.00379	3941		276	733.9	3	0.003936	15322
14	12	N25ab.txt	25	1	6	102	8121	2	0.005956	6221		102	274.52	2	0.005854	24047
15	13	N25as.txt	25	4	9	189	38176.75	3	0.006952	6221		189	517.55	3	0.004204	24065
16	14	N30as.txt	30	3	26	121	15546.59	8	0.008311	10982		121	340.59	8	0.00853	37931
17	15	N30ab.txt	30	3	26	-nan	-nan	0	0.008738	22351		-nan	-nan	0	0.010863	53195
18	16	N35as.txt	35	33	23	7	310.89	2	0.011631	16411		7	18.86	2	0.011371	52345
19	17	N35ab.txt	35	33	23	7	469.5	2	0.011382	16411		7	18.71	2	0.011574	52345
20	18	N50as.txt	50	2	14	33	39537.53	7	0.024661	95759		33	91.77	7	0.024924	189867
21	19	N50ab.txt	50	2	14	33	99060.63	7	0.032541	95759		33	91.82	7	0.024825	189867
22	20	N75as.txt	75	2	24	178	45936.76	14	0.055919	245815		178	501.8	14	0.058857	469620
23	21	N75ab.txt	75	2	24	178	64418.87	14	0.058137	245815		178	502.89	14	0.059098	469620

EPILOGUE:

UNFORSEEN SITUATIONS:

1) When we initially started the project, we were not very clear about actual and predicted paths, adjustment of flows. But, after working through it for several days, we got new ideas of how to proceed with it and implemented it in a better approach.

2) We considered adjacency matrix for path and stored the shortest path as a string. But, it was not efficient when implementing our approach. So, later on we considered 3D array data structure for this and its implementation saved a lot of time and complexity.

3) What is predicted path length? How to make it better? These were the biggest questions we had. Finally we figured out the terms after several days of discussion and implemented it. It was real fun.

LESSONS LEARNT:

1) We started working on the project from the day it was given to us. We had much time to think through several ways to do it. We would continue the same for further projects.

2) We learnt that working through different data structures gives us a better way to compare and decide among those. Because there is no 1 best data structure that fits everything. So, learnt that it is better to try working new things.

3) It is always better to sit and work with the group, as we were able to share ideas, correct ourselves, modify implementation and several other things.

One more opportunity:

If we have an opportunity to do this project again with additional requirements, we would try our best to optimize the output. Beyond that, we would go a step ahead and design more and more features that would be helpful in real world given enough time to do so. Also, we would think of an approach better than this and try to make it differently, as we always want to make our solution better than the previous one.

Lessons for Future Students:
Start Early, Work Hard. It is going to be fun.

Appendix A:

Implemented Programs:
Our Approach - DAA_Project_SingleNodeAtOneTime.c

Finding the G matrix as per the approach given - DAA_Project_GivenExampleStrategy.c

Compiler :GCC compiler

Operating System: Ubuntu

Running: gcc filename –lm

-lm to link math library to the compilation since we used predefined function pow(),floor().

Appendix B:

Comparison of Outputs is shown in an excel file named **'OutputComparision.xlsx'.**

Input File	Output File
input1.txt	Output_For_input1.txt
input2.txt	Output_For_input2.txt
N10bs.txt	Output_For_N10bs.txt
N10cs.txt	Output_For_N10cs.txt
N10ds.txt	Output_For_N10ds.txt
N10tb.txt	Output_For_N10tb.txt
N10ts.txt	Output_For_N10ts.txt
N15as.txt	Output_For_N15as.txt
N15bs.txt	Output_For_N15bs.txt
N20as.txt	Output_For_N20as.txt
N20bs.txt	Output_For_N20bs.txt
N25ab.txt	Output_For_N25ab.txt
N25as.txt	Output_For_N25as.txt
N30as.txt	Output_For_N30as.txt

N30ab.txt	Output_For_N30ab.txt
N35as.txt	Output_For_N35as.txt
N35ab.txt	Output_For_N35ab.txt
N50as.txt	Output_For_N50as.txt
N50ab.txt	Output_For_N50ab.txt
N75as.txt	Output_For_N75as.txt
N75ab.txt	Output_For_N75ab.txt