



**School of Computing and Engineering**

**Computer Science Department**

**Fall, 2016**

---

## **Shortest Path First (Floyd Warshall Algorithm)**

### **Project Report**

**Mohamed Gharibi (16199688)**

## Table of Contents

<b>Report Summary.....</b>	<b>3</b>
• Floyd Warshall.....	3
• Dijkstra Algorithm .....	3
• Ford Fulkerson.....	3
• Depth First Search .....	3
• Breadth First Search .....	3
<b>1. Introduction .....</b>	<b>4</b>
<b>2. Algorithms.....</b>	<b>5</b>
2.1 Floyd Warshall Algorithm .....	5
2.1 Dijkstra Algorithm .....	5
2.2 Bellman-Ford Algorithm .....	5
<b>3. Design and Analysis of Algorithms .....</b>	<b>6</b>
3.1 Floyd Warshal Algorithm .....	6
3.1.1 Floyd Warshall Algorithm Implementation.....	7
3.1.2 Behavior with Negative Cycles .....	8
3.1.3 Floyd Warshall Path Reconstruction .....	8
3.1.4 Floyd Warshall Analysis (Time Complexity).....	9
3.1.5 Floyd Warshall Formula .....	9
3.2 Depth First Search .....	10
3.2.1. DFS Recursive Implementation.....	11
3.2.2. DFS non-Recursive Implementation.....	11
3.2.3. DFS Time Complexity .....	11
3.3. Breadth First Search.....	12
3.3.1. Analyzing of BFS .....	12
3.3.2. BFS Recursive implementation .....	12
3.3.3. BFS Example .....	13
3.4. Floyd Warshall VS Dijkstra (The reason why we chose Floyd Warshall) .....	14
3.5. Why Floyd Warshall ? (Decision).....	15
3.6. Depth First Search VS Breadth First Search .....	16
3.7. Why DFS ? .....	17

3.8.	Data Structures .....	17
3.9.	Controlling the flow traffic Algorithm .....	18
<b>4.</b>	<b>Implementaion and Testing .....</b>	<b>20</b>
4.1.	Java As Implementating Programming Language .....	20
4.2.	Finding The Matrices Algorithms .....	21
<b>5.</b>	<b>Validate the pre-implementation analysis with a timing study .....</b>	<b>23</b>
<b>6.</b>	<b>Epilogue .....</b>	<b>25</b>
6.1.	Problems we faced .....	25
6.2.	How did we come up with a solution? .....	25
6.3.	What to change in the Project next time? .....	26
6.4.	What we have learned? .....	27
<b>7.</b>	<b>Conclusions .....</b>	<b>28</b>
<b>8.</b>	<b>References.....</b>	<b>29</b>

## **Report Summary**

This section summarizes the materials I studied, learned, and implemented as a course work of the Design and Analysis of Algorithms course, including the following topics:

- **Floyd Warshall**

Floyd Warshall algorithm is an algorithm for finding shortest paths in a weighted graph whether the graph has a positive or negative edge weights. Once you run the algorithm once it will return all the shortest paths for you from the first node to the last one. Floyd Warshall is an example of Dynamic programming and was published as it is known today by Robert Floyd in 1962.

- **Dijkstra Algorithm**

This algorithm will also find the shortest paths between nodes in a graph which could represent road traffic networks. This algorithm exists in many variants. The original Dijkstra will return the shortest path between two nodes in a graph. Another variant is to fix a single node as the source node and then finds all the shortest paths from the source node to the rest nodes in the graph. It was conceived by Edsg W. Dijkstra in 1965.

- **Ford Fulkerson**

Is an algorithm which is an example of a greedy approach that computes the maximum flow in a flow network, it is also called method instead of an algorithm. It has many different implementations with different running times. It was published by L. R. Ford and D. R. Fulkerson. The name Ford Fulkerson is often also used for Edmonds-Karp algorithm, which is also specialization of Ford Fulkerson algorithm.

- **Depth First Search**

Is an algorithm for traversing or searching tree or graph data structures. It starts from the root and explores as far as possible along each branch from a specific node before backtracking. It was investigated by French mathematician Charles Pierre.

- **Breadth First Search**

It is also an algorithm for traversing or searching a tree or graph structures, but the difference is BFS starts from the root and explores the neighbor's nodes first before moving to the second neighbors level. It was discovered by C. Y. Lee and published in 1961.

# Floyd Warshall Algorithm

## 1. Introduction

Nowadays, with all the technology we have, we are still spending hours and hours driving on the high way, to go to work , school or even having a nice dinner with friends. Nevertheless, we never thought if we are taking the shortest and the fastest path to get to our destination. Most of the people nowadays use GoogleMaps to get to their destination but only a few people thought about the way how is GoogleMaps application actually works. Let us say that we have a graph (Many nodes linked to each other) and this graph represents the cities inside United States. Our, problem occurs here, since we are trying to let all the cars that we have to move from one city to another by the shortest path. Unlike, the real life, you cannot use GoogleMaps here so in this case we have to think and come up with an algorithm to let all the cars move from one node (which we said ealier that each node represents a city) to any other node in our graph. Moreover, in our project we need to think about all the cities (nodes), which means that the starting node (source node) could be any node insinde our graph and the destination node also could be any other node inside the graph as well. That means, what we are trying to do is simply choosing any node as a source node and choosing any other node as a destination node. In addition to that, we have to think about the cars (which we represent them in our project by the flow matrix), which means there might be one car wants to move from city “A” to city “B” and then end up in the city “C”. Another car might want to move from “A” going through “B” and end up in the city “E”. In this case, both of the cars wants to go through the path “A-B”, and here we have another problem, since we have to make our project starts with a single car and then increase the number of the cars. In simple words, there might be a time where more than 100 cars passing in the same path, but unfortunately the path capacity might be only 100 cars. In this case, our algorithm takes place, since we have to think about a proper way to solve this issue without deleting some cars or creating a new paths for the cars, since that is does not make any sense in the real life.

To solve all these problems, we have to think about an algorithm to find the shortest path for the cars which is the first thing we need to do. The good news is that there are couple of algorithms which could solve this problem, but there is a small issue, which we need to understand all these algorithms, analyze them and then choose the best one to implement.

## **2. Algorithms**

### **2.1 Floyd Warshall Algorithm**

Floyd Warshall algorithm is an algorithm for finding shortest paths in a weighted graph whether the graph has a positive or negative edge weights. Once you run the algorithm once it will return all the shortest paths for you from the first node to the last one. Floyd Warshall is an example of Dynamic programming and was published as it is known today by Robert Floyd in 1962.

The genius of Floyd Warshall algorithm is that this algorithm works on finding different formulation for the shortest path sub problem than the path length formulation introduced earlier before Floyd Warshall.

### **2.1 Dijkstra Algorithm**

This algorithm will also find the shortest paths between nodes in a graph which could represent road traffic networks. This algorithm exists in many variants. The original Dijkstra will return the shortest path between two nodes in a graph. Another variant is to fix a single node as the source node and then finds all the shortest paths from the source node to the rest nodes in the graph. It was conceived by Edsg W. Dijkstra in 1965.

Dijkstra is a good algorithm works when you have a single source and you want to reach to a specific destination.

### **2.2 Bellman-Ford Algorithm**

Is another algorithm works like Dijkstra. Unlikely to Floyd Warshall, we use Bellman-Ford only for looking from one node to another in the graph. That means, we cannot use this algorithm to find and search for all the nodes we have in the graph, we use it when we have a single node and we are looking for a single destination.

### 3. Design and Analysis of Algorithms

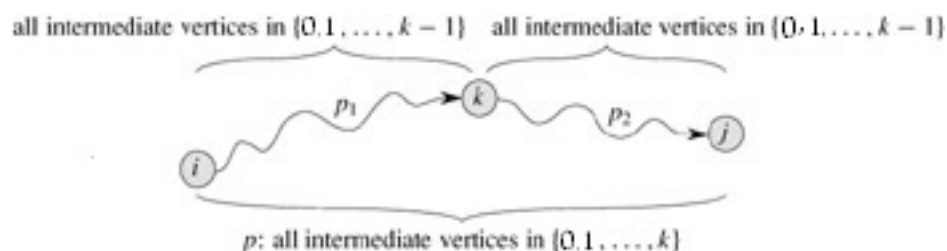
#### 3.1 Floyd Warshal Algorithm

First of all, Floyd Warshall computes all possible paths inside a graph between each pair of vertices (nodes). This algorithm is able to do all this work within  $\Theta(v^3)$  comparisons in a graph. Considering we have an  $v^2$  nodes in a graph.

Secondly, we initialize the solution matrix same as the input graph matrix as a first step in implementation. Then, we update the solution matrix by considering that all the nodes we have as an intermediate node. This approach will lead us to the idea that we are moving node by node from the beginning to find the shortest path for all the nodes including the node which was picked up as an intermediate node in the shortest path. We usually pick  $k$  as the intermediate node in our shortest path. So, as we mentioned we have the intermediate node  $\{0, 1, 2, 3, \dots, k-1\}$  then:

$K$  is not an intermediate node from  $i$  to  $j$ . We keep the value  $\text{dist}[i][j]$  as it is.

$K$  is an intermediate node in the shortest path from  $i$  to  $j$ . We update the value as  $\text{dist}[i][j] = \text{dist}[i][k] + \text{dist}[k][j]$ . The next figure(1) was taken from Cormen book to illustrate this:



figure(1)

### 3.1.1 Floyd Warshall Algorithm Implementation

```

1 let dist be a  $|V| \times |V|$  array of minimum distances initialized
  to  $\infty$  (infinity)
2 for each vertex  $v$ 
3    $\text{dist}[v][v] \leftarrow 0$ 
4 for each edge  $(u,v)$ 
5    $\text{dist}[u][v] \leftarrow w(u,v)$  // the weight of the edge  $(u,v)$ 
6 for  $k$  from 1 to  $|V|$ 
7   for  $i$  from 1 to  $|V|$ 
8     for  $j$  from 1 to  $|V|$ 
9       if  $\text{dist}[i][j] > \text{dist}[i][k] + \text{dist}[k][j]$ 
10         $\text{dist}[i][j] \leftarrow \text{dist}[i][k] + \text{dist}[k][j]$ 
11       end if

```

Figure(2) will illustrate how it works step by step:

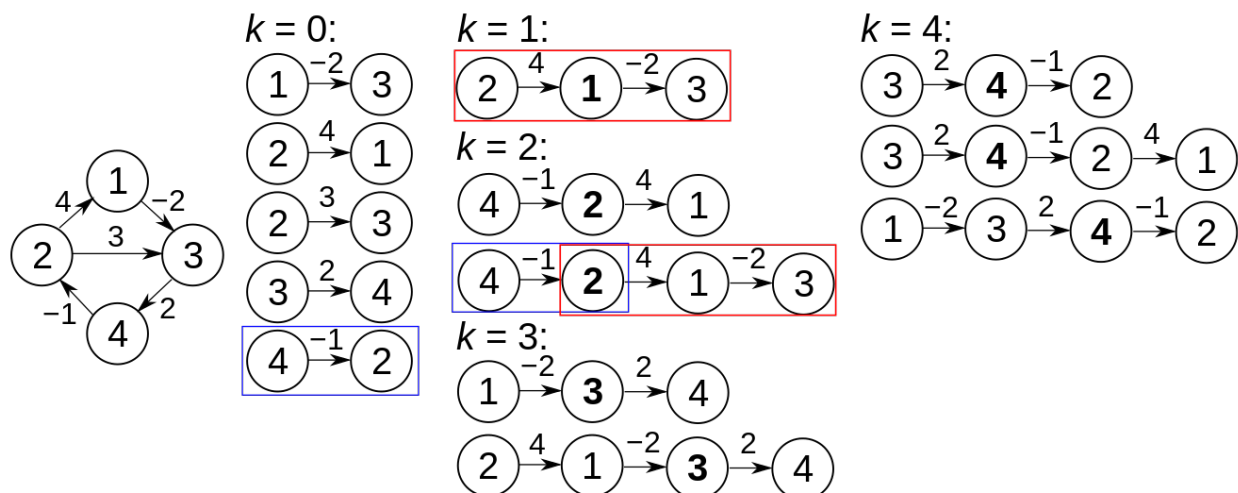


Figure (2)



### 3.1.2 Behavior with Negative Cycles

A negative cycle in a graph is a cycle whose edges are sum to a negative values. There is no shortest path between any two nodes  $I, j$ , which form a part of any negative cycle. That is because the path between  $I$  and  $j$  can be very small (in negative values).

Floyd Warshall in this case assumes there are no negative cycles. Nevertheless, if there are any negative cycle, then Floyd Warshall can detect them by inspecting the diagonal of the path matrix and if you get at least one negative number, that means the graph at least contains one negative cycle.

### 3.1.3 Floyd Warshall Path Reconstruction

```
let dist be a      array of minimum distances initialized to
(infinity)

let next be a      array of vertex indices initialized to null

procedure FloydWarshallWithPathReconstruction ()
  for each edge (u,v)
    dist[u][v]  $\leftarrow$  w(u,v) // the weight of the edge (u,v)
    next[u][v]  $\leftarrow$  v
  for k from 1 to |V| // standard Floyd-Warshall implementation
    for i from 1 to |V|
      for j from 1 to |V|
        if dist[i][k] + dist[k][j] < dist[i][j] then
          dist[i][j]  $\leftarrow$  dist[i][k] + dist[k][j]
          next[i][j]  $\leftarrow$  next[i][k]

procedure Path(u, v)
  if next[u][v] = null then
    return []
  path = [u]
  while u  $\neq$  v
```

```

    u ← next[u][v]
    path.append(u)
return path

```

### 3.1.4 Floyd Warshall Analysis (Time Complexity)

**Floyd-Warshall(W)**

$n = W.rows$

$D^{(0)} = W$

**for**  $k = 1$  **to**  $n$

    let  $D^{(k)} = (d_{ij}^{(k)})$  be a new matrix

**for**  $i = 1$  **to**  $n$

**for**  $j = 1$  **to**  $n$

$d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$

**return**  $D^{(n)}$

**Floyd-Warshall(W)**

$n = W.rows$

$D = W$

$\Pi$  initialization

**for**  $k = 1$  **to**  $n$

**for**  $i = 1$  **to**  $n$

**for**  $j = 1$  **to**  $n$

**if**  $d_{ij} > d_{ik} + d_{kj}$

**then**  $d_{ij} = d_{ik} + d_{kj}$

$\pi_{ij} = \pi_{kj}$

**return**  $D$

As we can see above, the run time complexity in the best case is  $\Theta(v^3)$  in the best case. Similar to the best case is the average case, since the time complexity in the average case is also  $\Theta(v^3)$ . Lastly, the time complexity for the worst case is also  $\Theta(v^3)$ .

### 3.1.5 Floyd Warshall Formula

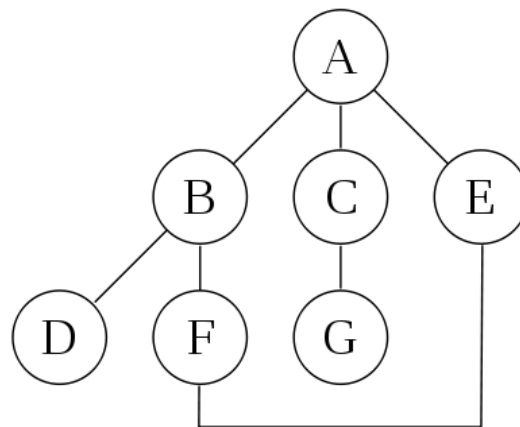
$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1 \end{cases}$$

### 3.2 Depth First Search

Is an algorithm for traversing or searching tree or graph data structures. It starts from the root and explores as far as possible along each branch from a specific node before backtracking. It was investigated by French mathematician Charles Pierre.

The time and the space analysis of this algorithm (Depth First Search) differs according to the way and the area of implementation. In Computer Science, DFS is used in a graph to traverse that graph, and it takes  $\Theta(V + E)$ , linear in the size of the graph. But according to the space complexity, it takes  $\Theta(V)$ , in the worst case to store a stack of vertices.

For some applications which use DFS in a specific domain such as searching in artificial intelligence, the graph which being traversed is either too large or infinite. In these cases, search is implemented to a limited depth, but in our case the search is implemented till the end of the graph, that means all the nodes are being visited while using DFS.



Figure() DFS example

In in the above example, DFS algorithm will start from A, we will assume that the left part of the graph is being visited first (let s say that we are taking the nodes by the alphabetical order) and also let us assume that we are not going to repeat the nodes. Finally, the DFS algorithm will visit all the nodes and return for us this order: A, B, D, F, E, C, G. Another case, if we did not assume that there is no repeating, then the returned order will be A, B, D, F, E, A, D, F, E. I think you

can see that there are some nodes were repeated twice, an so on if we have another branch from A and we also did not assume that there is no repeating. Then, A will be visited three times or more based on the branches in the graph.

### 3.2.1. DFS Recursive Implementation

```
1 procedure DFS( $G, v$ ):  
2     label  $v$  as discovered  
3     for all edges from  $v$  to  $w$  in  $G$ .adjacentEdges( $v$ ) do  
4         if vertex  $w$  is not labeled as discovered then  
5             recursively call DFS( $G, w$ )
```

### 3.2.2. DFS non-Recursive Implementation

```
1 procedure DFS-iterative( $G, v$ ):  
2     let  $S$  be a stack  
3      $S$ .push( $v$ )  
4     while  $S$  is not empty  
5          $v = S$ .pop()  
6         if  $v$  is not labeled as discovered:  
7             label  $v$  as discovered  
8             for all edges from  $v$  to  $w$  in  $G$ .adjacentEdges( $v$ ) do  
9                  $S$ .push( $w$ )
```

As you can see that based on these two variants for DFS, the algorithm will return two different results. The first one will be A, B, D, F, E, C, G. The second one will be A, E, F, B, D, C, G.

### 3.2.3. DFS Time Complexity

Since we are interested in all the nodes in the graph and we want the algorithm to visit all the nodes so after running DFS, we saw that the time complexity for DFS in worst case is  $\Theta(E)$ .

### 3.3.Breadth First Search

It is also an algorithm for traversing or searching a tree or graph structures, but the difference is BFS starts from the root and explores the neighbor's nodes first before moving to the second neighbors level. It was discovered by C. Y. Lee and published in 1961.

Similar to DFS algorithm, BFS has also two variants, in the non-recursive variant implementation is almost same as DFS non-recursive implementation except two things:

- It uses queue instead of a stack.
- It checks whether the node has been visited and discovered before enqueueing the node rather than delaying this check until the node is dequeued from the queue.

At the beginning of the implantation for BFS we consider the all the nodes are INFINITY, which the way to say that this node has not been explored or discovered yet, and therefore it has not yet a path from the starting node. We also could use another symbols to represent that such as -1. Except the number 0, since it has another meaning in this algorithm.

#### 3.3.1. Analyzing of BFS

The time complexity for BFS can be expressed as  $O(V+E)$ , since every vertex and every edge will be explored in the worst time complexity. Moreover, I believe in this project we are interested in the worst time complexity since we want the algorithm to explore all the nodes.  $V$  in the algorithm represents the number of the vertices and  $E$  is the number of the edges in a given graph. Note that in some cases  $O(E)$  may vary between  $O(1)$  and  $O(v^2)$  based on the sparse the input in the graph. In the space, the time complexity is  $O(V)$  since all the vertices have to be visited.

In BFS, the input graph and most of the times is assumed to be finite graph. Such as the DFS algorithm, in some cases such as searching in artificial intelligence then the algorithm may stops if the graph is infinite.

#### 3.3.2. BFS Recursive implementation

```
Breadth-First-Search(Graph, root):
```

```

for each node n in Graph:
    n.distance = INFINITY
    n.parent = NIL

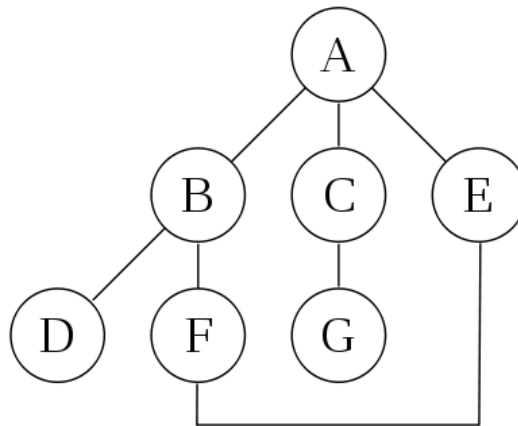
create empty queue Q

root.distance = 0
Q.enqueue(root)

while Q is not empty:
    current = Q.dequeue()
    for each node n that is adjacent to current:
        if n.distance == INFINITY:
            n.distance = current.distance + 1
            n.parent = current
            Q.enqueue(n)

```

### 3.3.3. BFS Example



In the same graph given for DFS example, the BFS result will be A, B, C, E, D, F, G.

Note that the same conditions we assumed here such as there is no repeating for the node.

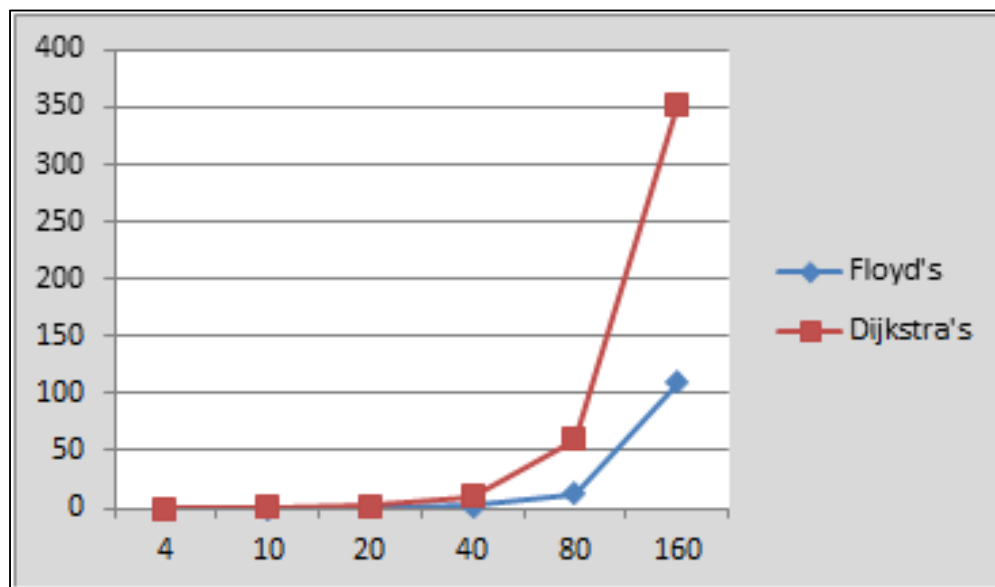
### **3.4.Floyd Warshall VS Dijkstra (The reason why we chose Floyd Warshall)**

There are many differences between these two algorithms, we will present some of these differences below:

- The most important difference and main thing is that Dijkstra algorithm is used only when you have a single source and you are trying to find the shortest path from this single source to another single node (which is the destination node). Unlike Floyd Warshall algorithm, which finds and searches through all the nodes you have in the graph.
- Dijkstra algorithm is an example of Single-Source Shortest Path algorithm (SSSP), which will return for you only the shortest path between two nodes you chose in the graph.
- The way how Dijkstra deals with the negative cycles, depends on the precise variant under discussion. Some variants, such as the variant on Wikipedia computes the shortest path in a fast way but does not return correctly the shortest path when there are negative cycles. Unlike Floyd Warshall, it can compute all the nodes and can also deal with the negative cycles and return correct results.
- In Dijkstra, if there is a negative path which is reachable directly from the source, it will compute it but still that will affect the time complexity in a obvious way.
- The time complexity in Dijkstra algorithm may look smaller than the time complexity in Floyd Warshall, but you have to put in mind that Dijkstra's time complexity is only when you choose a single node as a source and a single node as a destination. Otherwise, you have to increase the time complexity for Dijkstra a lot when you want to run the algorithm for all the nodes as a source and all the nodes as a destination.

### 3.5. Why Floyd Warshall ? (Decision)

After we looked over most of the algorithms that exist today to find the shortest path, we chose Floyd-Warshall to solve the **all-pairs** shortest path problem over Dijkstra's algorithm for several reasons. The main reason is time complexity, Dijkstra's algorithm is designed to find the single-source shortest path problem with time complexity  $O(E + V \lg(V))$ . In our case, we need to find all-pairs shortest path. To find all-pairs shortest path using Dijkstra, we need to run it on every single node which makes the time complexity  $O(VE + V^2 \lg(V))$ . In contrast, the time complexity for Floyd-Warshall algorithm is  $O(V^3)$ .



Figure(1): The comparison between Dijkstra and Floyd-Warshall to find the shortest path for all vertices [1].



In addition, Dijkstra's algorithm fails to give the correct results when there are negative weights whereas Floyd-Warshall works for negative edge weights. Also, it is easier to code and implement. First of all, for implementing Floyd Warshall algorithm, we need to understand the way it works, and to do so we need to convert this formula into a coding lines:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1 \end{cases}$$

As we started to do so, we found that Java programming language is the best programming language we could use to implement, and that is because many reasons:

### 3.6.Depth First Search VS Breadth First Search

- In BFS, you will start exploring and visiting the nodes starting from the root (top of the tree or graph) and then go to the second level to start from the left side till you finish all the second level, then you will go to the third level and so on until you finish all the nodes.
- Unlike BFS, the DFS will start from the root as well and then explore the second level but not all the nodes, rather it will keep going one level after a level in the same branch until it finishes all the nodes in the branch then it will go to another branch.
- Choosing between DFS and BFS depends on the structure of the search tree or the graph you are going to use.
- BFS works better for the trees and graphs whose solution is not far from the top of the tree or graph.
- DFS works in a better way than BFS in the cases where the solution is far from the root or the starting point of the tree or the graph you are using.
- If the solutions are rare then it is better for you to use DFS.
- Unlike DFS, then it is better to use BFS is the solutions are frequent.
- DFS may use more memory when you run it over a very deep solution, but we do not care in this level about the memory a lot, since we are not going to use the deep in our graphs. Moreover, our trained nodes and data are not that large.

### 3.7. Why DFS ?

The main reason for choosing DFS, is that DFS uses stack for the implementation, which we totally need for storing the shortest path nodes and print them in an array. Moreover, from the above comparison, we can say that it is better for us to use DFS in our project for many reasons, since we are looking for a rare solutions and we might search in the deep solutions sometimes. Moreover, the time complexity for DFS is not that large since we are using the stack, which means the space complexity will be relevant as well.

### 3.8. Data Structures

For implementing this project, we used many types of data structures to help us with all the implementations for the above algorithms. We used variables to store some values such as the INFINITY. We also used arrays, which play the main role in our project, since we are storing all the data, numbers, edges, flows, loads, and even the congestions in the graph. We used the stack as well, which we could say it is almost the main data structures we used in this project to store all the nodes as the DFS algorithm works and the to keep the nodes which came out of the stack as the shortest path nodes.

Here are some examples for the data structures we used:

- E: is the matrix which contain the edges weights.
- F: is the matrix which contains the flow values.
- C: is the matrix contains the capacity values.
- L: is the matrix which contains the load in the graph.
- P: is the matrix which contains small arrays as the path nodes.
- Temp: is the small matrix which contains the path nodes from any source to any destination.
- G: is the congestion matrix.
- myStack: is the stack which contains the DFS nodes results.
- Hops: is the matrix which store all the hops count in the graph.
- pathPoints: is the matrix which contains the paths between all the nodes.

We also declared some matrices to use inside the functions such as flowPerStep, toNode, fromNode and so on, but we already listed the main matrices.

### 3.9. Controlling the flow traffic Algorithm

#### A. First Algorithm

We decide to manipulate with Flow (F) which represents the number of cars that travel on the entire path from source to node destination every hour because changing the Flow will explicitly change the traffic load, and correspondingly the congestion will be changed too.

Hence we need two for loops to loop through all the array and we need a condition to check if we hit the 10% of flow that we are trying to get.

```
1. Initialize / Set Flow matrix to 1 For(i from 0 to the number of the edges) {
2.     For(j from 0 to the number of the edges) {
3.         while Flow[i][j] is less than or equal to 10 % of Predicted Path {
4.             if (Flow[i][j] less than Original_Flow[i][j]) {
5.                 Flow[i][j] + 1
6.             }
7.             End If
8.         }
9.         End
10.    while End I loop End j loop
```

We need another loop to make sure that the route, which got affected by the increment, will be adjusted.

```
1. For(i from 0 to the number of the edges) {
2.     For(j from 0 to the number of the edges) {
3.         While Flow[i][j] is Greater than 10 % of Predicted Path
4.             IF Flow[i][j] = 0
5.                 Continue;
6.             Else
7.                 Flow[i][j] - 1 End I loop End j loop
```

Note: There are some cars passing each path per hour. We are going to loop through all the paths and feed each path by a specific number of cars per hour (the first loop). After that, in the second loop we assume that the number of cars in each path are already 0 because all the cars by this time reached the next node, or at least there is a room to let more cars go through.

## B. Second Algorithm

We notice that the congestion increased in the path that has more than one hop, so we decide to add another factor to adjust the flow. For each Path, we see how many hops it has, and then take off a certain Percent (shown in the example below) from the Flow(that matches the edges). For example, if the Path has 3 hops and the Flow is 18, then we will take 70% off from the total Flow (based on the \*Formula) which is going to be 6 cars. That means 6 cars are going to pass the road

```
1. For(I from 0 to the number of edges) {
2.     For(j from 0 to the number of edges) {
3.         Flow[i][j] = Flow[i][j] - (Flow[i][j] * (Hop[i][j] * 10) / 100)
4.     End I loop
5. End j loop
```

\*Formula:

First we multiply the hop number by 10.

Second, we will multiply the result we got by the flow value.

Finally, we divide the total by 100 to get the percentage.

Note, Finding an algorithm was one of the hardest challenges, since we had to think a lot and try many inputs and compare all our outputs. We believe there is no exact solution for this algorithm but there are some solutions which are close enough to be true. Our approaches could be used in a good way to control the traffic and the load of the cars, no matter how many cars are there.

## 4. Implementaion and Testing

### 4.1. Java As Implementating Programming Language

- Java is an Object Oriented Programming language, which means you can deal with the graph and the nodes and everything as an object and so can control it better.
- It is free language and its syntax is very much similar to C and C++.
- Java provides a very nice platform for transfer from one address space to another
- One of the main reasons is that there is AUTOMATIC MEMORY MANAGEMENT implemented by Garbage Collection and NOExplicitPointer.
- It is a language design not a committee driven.
- Provides a lot of reused codes and has a very large third-party library.
- Using Java can easily add some libraries and use another languages with Java such as Python, Scala and so on. Unlike, when you use another languages and then you want to convert.
- Provides good libraries for graphs and has a good using for the collections (Data Structures).
- Easy to implement and does not slow the machine speed.

## 4.2.Finding The Matrices Algorithms

After we listed the reasons why we used java, now we will start to present how we implemented Floyd Warshall algorithm using Java. First, we converted the formula to code to tell the machine to choose the best way and to do so we used three “FOR” loops (which is the best way to do it).

```
1. //Shortest path algorithm
   for (int k = 0; k < N; k++) {
       for (int i = 0; i < N; i++) {
           for (int j = 0; j < N; j++) {
               if (E[i][k] + E[k][j] < E[i][j]) {
                   E[i][j] = E[i][k] + E[k][j];
                   P[i][j] = P[k][j];
               } else {
                   E[i][j] = E[i][j];
               }
           }
       }
   }
```

and before that, to declare the infinity we used a very large number as shown below:

```
1. final static int INF = 99999;
```

We also declared couple of matrices such as the nodes matrix, flow matrix, Load matrix, shortest path matrix (which is a matrix contain many small matrices to store the shortest path nodes) and the hops number matrix and some other matrices.

We also created a function to declare the infinity in the matrices so that we can check the negative cycles as shown below:

```
1. public static int[][] Floyd(int[][] M) {
2.     final int Inf = 10000;
3.     for (int i = 0; i < N; i++) {
4.         for (int j = 0; j < N; j++) {
5.             if (i == j || M[i][j] == Inf)
                P[i][j] = -1;
6.             else P[i][j] = i;
7.         }
8.     }
```

After that, we had to declare a matrix to store all the shortest paths nodes and to do so, we had to use a stack, so that we can push all the explored nodes inside and after that for keeping the shortest path nodes we had to reverse that stack, since the stack works as First In Last Out (FILO) and finally we had to use our matrix to store these results (output nodes) which they are the shortest

path nodes. We had also to create a loop so that every time the stack is not empty we will pop an element and push it in the array.

```

1. public static int[] pathD(int i, int j, int[][] p) {
2.     Stack < Integer > ps = new Stack < Integer > ();
3.     ps.push(j);
4.     int temp = p[i][j];
5.     while (temp != -1) {
6.         {
7.             ps.push(temp);
8.             temp = p[i][temp];
9.         }
10.    }
11.    int[] arr = new int[ps.size()];
12.    int k = 0;
13.    while (!ps.empty()) {
14.        arr[k] = ps.pop();
15.        k++;
16.    }
17.    return arr;
18. }

```

Now, since we have the shortest path we surly can calculate the hop count from the path nodes, since the number of hops is just the number of the nodes minus one. And we did so,

```

1. for (int i = 0; i < N; i++) {
2.     for (int j = 0; j < N; j++) {
3.         pathPoints[i][j] = pathD(i, j, P);
4.         hops[i][j] = pathPoints[i][j].length - 1;
5.     }
6. }

```

We did not do any hard working in the hop count we just needed to print the shortest path nodes again and to calculate the number of node, then we will deduct one and write the output to a new matrix call Hops.

After that, for calculating the Load matrix “L” which is the maximum flow in a certain path, we had to work on it a little bit more since now we have to look inside all the arrays which are inside the path matrix to search for a specific path between two nodes and check how many time this path was appeared and then calculate all the flows that pass through the certain path between two nodes and write the output to a new matrix. And we were able to do it as shown below, using only three “FOR” loop:

```

1. for (int i = 0; i < N; i++) {
2.     for (int j = 0; j < N; j++) {

```

```

3.         int[] currentPath = pathPoints[i][j];
4.         int currentFlow = F[i][j];
5.         for (int k = 0; k < currentPath.length - 1; k++) {
6.             int fromNode = currentPath[k];
7.             int toNode = currentPath[k + 1];
8.             flowPerStep[fromNode][toNode] += currentFlow;
9.         }
10.    }
1. }

```

After we got all of the above, now it is the time to get the Congestion Matrix which we called it G. To get this matrix we have to declare a new matrix “Capacity matrix” which we called C, we got the C matrix value from the project handout. First thing to do now, is simply to implement the formula given in the handout.

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

As we mentioned above, we just declared a new matrix called G to store the Congestion matrix into it, we also read the complexity matrix from the input and got the Load Matrix which we called it “L” from the previous step, and finally we have the edges matrix which we called E. All what we have to do is just implement the formula using java, which we did as below:

```

1. G[i][j] = ((C[i][j] + 1) / (C[i][j] + 1 - flowPerStep[i][j])) * M[i][j];

```

## 5. Validate the pre-implementation analysis with a timing study

From the time complexity for Floyd Warshall and the Depth First Search algorithms and the other loops we used in the code, we can predict the amount of time that the CPU will take to compute all the calculations we have in the code. Moreover, in the theory part we calculated that using the given number of nodes and flows. Fortunately, we proudly can say that we got a very close number to the output we got in the implementation using the CPU usage and speed calculator. To do that, we declared a function in java to compute the time in milliseconds.



I would like to mention that I am using:

- MAC laptop
- Running i7 core CPU
- With 16 GB RAM
- Using IntelliJ Java Integrated Development Environment.
- Java coding
- Calculating all the loops in the code

```
1. long startTime = System.currentTimeMillis();  
2. long stopTime = System.currentTimeMillis();  
3. long runTime = stopTime - startTime;  
4. System.out.println("Run Time" + runTime);
```

The result is for calculating all the code since it starts till it stops takes 20 milliseconds of time. It takes 14 milliseconds to come up with all the matrices and run the code except the Load and the congestion matrices. It also takes 3 milliseconds to run the load matrix, also another 3 milliseconds for running the Congestion matrix.

Our algorithm that we came up with took 1 millisecond to do all the calculations needed, which in my opinion it is a good time since we are measuring the worst case time.

## 6. Epilogue

I would like to mention that the data structures I used were arrays and a stack. Arrays are good containers to use. Similarly the stack, which helped us a lot for storing the nodes that make the shortest path from any node as a source to any node as a destination.

### 6.1. Problems we faced

- In the beginning, we were confused about which algorithm to choose. We chose Floyd Warshall after a lot of studying and comparing between many different algorithms.
- What programming language to use? That was a difficult question as well.
- How to store the path nodes in a specific array or list.
- How get the maximum load values from the array of arrays which contain the path nodes.
- After we got the congestion matrix, we had a problem in the long decimal number after the point.
- How to correctly calculate the usage and the time of the CPU.
- The biggest problem was how to come up with a new algorithm to control all the traffic and load cars in the graph.

### 6.2.How did we come up with a solution?

- For choosing the best shortest path algorithm which we can use in our code, we had to read a lot about the different algorithms, and make some trying and experiments on each algorithm so that we could choose the best one that works perfectly in our project.
- Programming language, we also had to go through many choices and then chose java based on flexibility that java provides. By its libraries and the defined functions.
- For storing the path we had to declare a stack and use it, after that we had to reverse the stack contents and push them in an array.
- Getting the maximum load values was by using another class and declaring another functions to use instead of our used functions. We also had to make three loops to go through all the small elements inside the small arrays which they are inside our matrix.

- For the congestion matrix, we discovered that our results in the beginning were wrong, so we had to do all the calculations again. We also had to create another class for that since the results were affected by another values in our old class.
- The long decimal in our code was being printed as a very long number so we had to write a new function to return only the first number after the decimal.
- Calculating the CPU usage, we also made a function to compute the CPU time and load starting when we run our code until the CPU finishes the implementations.
- Thinking about a new algorithm was not that easy, because we had to think, try, experiment, and write many inputs and checking all our outputs and trying to get the best results we could.

### **6.3.What to change in the Project next time?**

- If I have to do this project again, simply I would use the lists instead of the matrices. I thought about that for many reasons:
- The matrix will contain  $O(N*N)$  size in the memory.
- The matrix takes two for loops to search inside  $O(N^2)$ .
- Wherein the lists are linear.
- I think I will also choose the queue instead of the stack to try the new output.
- I will also have to use the BFS algorithm instead of DFS algorithm if I used a queue.
- Using another programming languages with Java to give a better interface.

#### **6.4.What we have learned?**

- I learned how exactly the shortest path algorithms work.
- I also learned how to implement these algorithms.
- How to use java more and more and implement the graphs.
- I learned about the new ideas were implemented to avoid all the traffic and loads.
- I learned how to calculate the CPU usage and load for any code I am running.
- I totally got the idea of how DFS and BFS are working.
- I had to read and look about other algorithms such as Dijkstra and Bellman Ford.
- I learned how to work within a group and share all the ideas and thoughts.

## 7. Conclusions

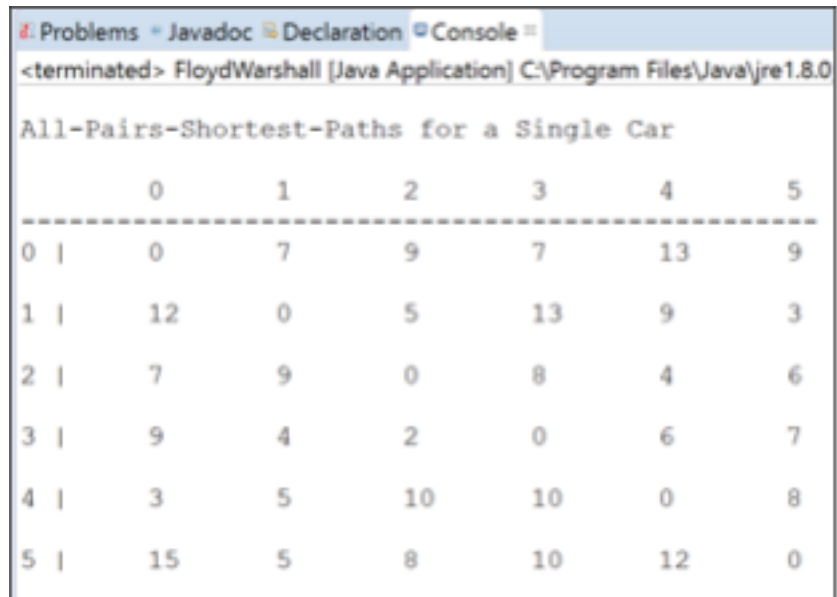
In this project, we had to look and search for all the algorithms that could help us to explore and find the best and shortest path, not only from a single node as a source to a single node as a destination, but from all nodes to choose any one as a source to all the nodes as a destination. We used Floyd Warshall algorithm, we understood it perfectly and then implement it. In our project, we also tried to manipulate with the edges weight, flow direction and weight so that we check the best path not only for a specific car, but to any number of cars that pass through the graph at any time. We also used DFS algorithm to come up with the path nodes and explore the neighbors. Finally, we measured the CPU usage and load to check the time that every algorithm takes while running.

## 8. References

- [1] [https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall\\_algorithm](https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm)
- [2] [https://en.wikipedia.org/wiki/Dijkstra%27s\\_algorithm](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm)
- [3] GeeksForGeeks.com, dynamic programming, Floyd Warshall
- [4] Floyd Warshall Algorithm slides, by Chandler Burfiled.
- [5] “All Pairs Shortest Path with Floyd Warshall Algorithm” PDF, CPSC 490.
- [6] “Shortest Path Algorithms” slides, by Jaehyun Park, Stanford University.
- [7] [https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford\\_algorithm](https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm).
- [8] Stack Exchange, Computer Science section, Dijkstra vs Floyd Warshall (<http://cs.stackexchange.com/questions/50005/dijkstra-vs-floyd-warshall>).
- [9] Chamber, "Shortest path algorithm comparison," in Rebus Technologies, 2011. [Online]. Available: <http://rebus technologies.com/shortest-path-algorithm-comparison/>.
- [10] <https://www.safaribooksonline.com/library/view/learning-spark/9781449359034/>

## 11.Appendix (Output)

### Shortest path matrix

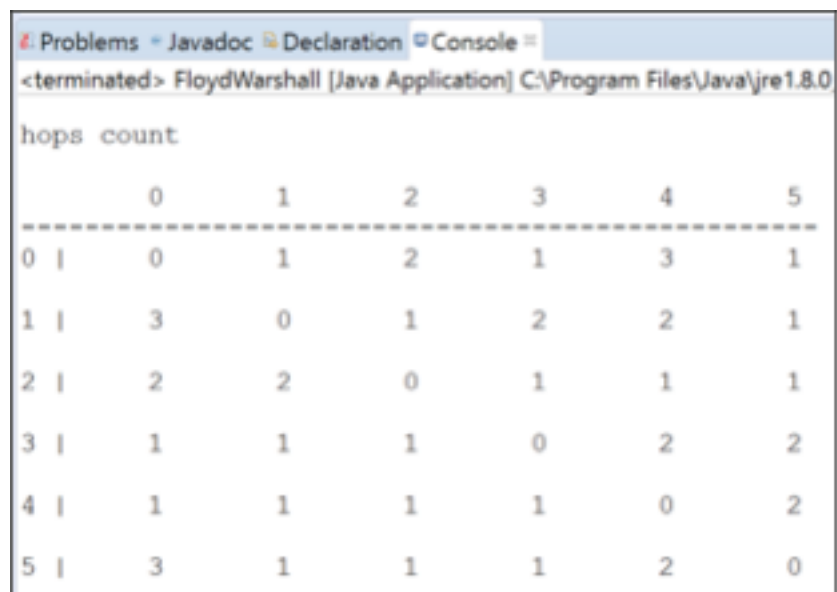


```
<terminated> FloydWarshall [Java Application] C:\Program Files\Java\jre1.8.0

All-Pairs-Shortest-Paths for a Single Car
```

	0	1	2	3	4	5
0	0	7	9	7	13	9
1	12	0	5	13	9	3
2	7	9	0	8	4	6
3	9	4	2	0	6	7
4	3	5	10	10	0	8
5	15	5	8	10	12	0

### Hops count matrix



```
<terminated> FloydWarshall [Java Application] C:\Program Files\Java\jre1.8.0

hops count
```

	0	1	2	3	4	5
0	0	1	2	1	3	1
1	3	0	1	2	2	1
2	2	2	0	1	1	1
3	1	1	1	0	2	2
4	1	1	1	1	0	2
5	3	1	1	1	2	0

The array of arrays “All pairs shortest paths matrix”

```
Run FloydWarshall
(0)(01)(032)(03)(0324)(05)
(1240)(1)(12)(123)(124)(15)
(240)(241)(2)(23)(24)(25)
(30)(31)(32)(3)(324)(315)
(40)(41)(42)(43)(4)(415)
(5240)(51)(52)(53)(524)(5)
```

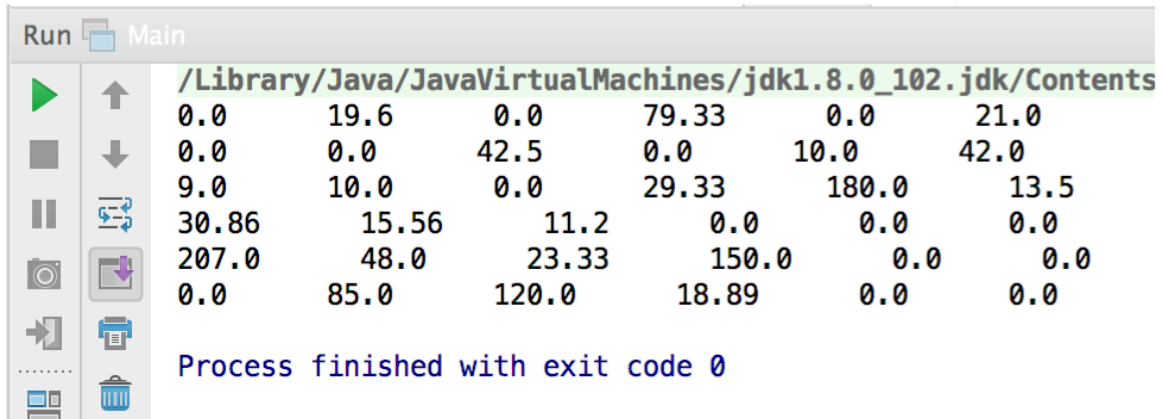
Load Matrix “L”

```
Run PathFlowCalculator
/Library/Java/JavaVirtualMachines/jdk1.8.0
0    9    0    31    0    12
0    0    60   0    0    52
0    0    0    24   132   10
17   26   46    0    0    0
68   43   12   14    0    0
0    16   42    8    0    0

Process finished with exit code 0
```



## Congestion Matrix “G”

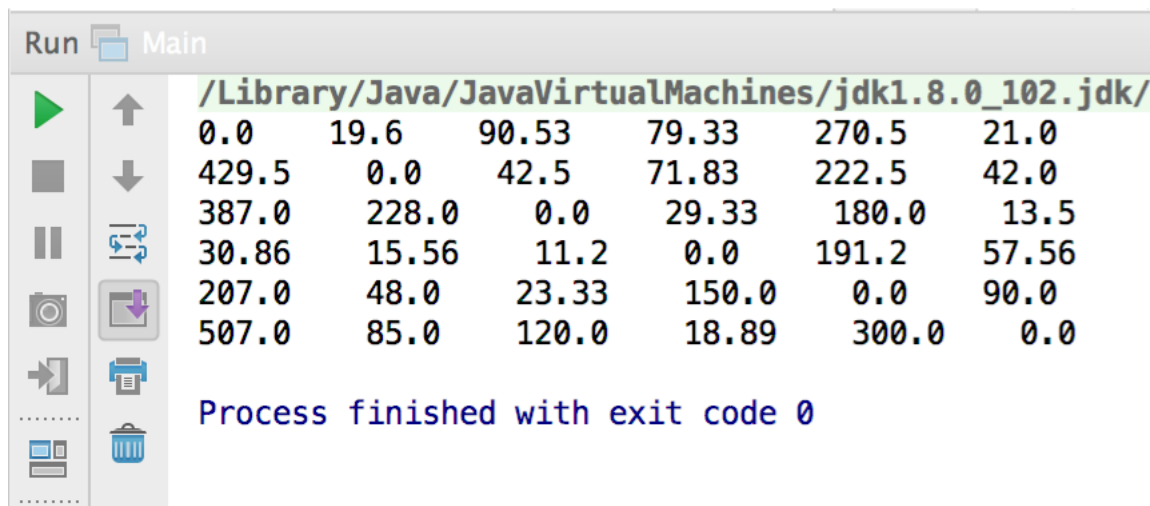


The screenshot shows a Java IDE console window with the title 'Run Main'. The output displays a 6x6 matrix of values representing the Congestion Matrix 'G'. The path is highlighted as `/Library/Java/JavaVirtualMachines/jdk1.8.0_102.jdk/Contents`. The matrix values are as follows:

0.0	19.6	0.0	79.33	0.0	21.0
0.0	0.0	42.5	0.0	10.0	42.0
9.0	10.0	0.0	29.33	180.0	13.5
30.86	15.56	11.2	0.0	0.0	0.0
207.0	48.0	23.33	150.0	0.0	0.0
0.0	85.0	120.0	18.89	0.0	0.0

Process finished with exit code 0

## Actual Path Delay Matrix



The screenshot shows a Java IDE console window with the title 'Run Main'. The output displays a 6x6 matrix of values representing the Actual Path Delay Matrix. The path is highlighted as `/Library/Java/JavaVirtualMachines/jdk1.8.0_102.jdk/`. The matrix values are as follows:

0.0	19.6	90.53	79.33	270.5	21.0
429.5	0.0	42.5	71.83	222.5	42.0
387.0	228.0	0.0	29.33	180.0	13.5
30.86	15.56	11.2	0.0	191.2	57.56
207.0	48.0	23.33	150.0	0.0	90.0
507.0	85.0	120.0	18.89	300.0	0.0

Process finished with exit code 0

Our Approach Matrix

	0	1	2	3	4	5
0	0.0	7.538	17.48	14.875	22.934	9.947
1	15.81	0.0	6.296	15.4	11.751	3.652
2	9.513	11.608	0.0	9.103	5.455	6.353
3	9.818	4.667	2.605	0.0	8.059	8.319
4	4.059	6.154	11.053	11.538	0.0	9.806
5	19.799	5.667	10.286	10.625	15.74	0.0