VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
Faculty of Applied science

Linear Algebra (MT1007)

Assignment

Floyd Warshall Algorithm

Advisor: Nguyen Tien Dung
Students: Phan Quang Minh - 2252486.
Nguyen Binh Nguyen - 2252545.
Nguyen Le Duy Long - 2252445.
Doan Vuong Bao Ngoc - 2252532.

HO CHI MINH CITY, DECEMBER 6

Contents

1 Member list & Workload

No.	Fullname	Student ID	Problems	Percentage of work
1	Phan Quang Minh	2252486	Theory	25%
2	Nguyen Binh Nguyen	2252545	Producing and testing code	25%
3	Nguyen Le Duy Long	2252445	Example	25%
4	Doan Vuong Bao Ngoc	2252532	Solution	25%

2 Introduction

Imagine a company that operates a fleet of delivery vehicles tasked with transporting goods from a central warehouse to various customer locations across a city. Each road segment connecting different locations has a specific travel time or distance associated with it, reflecting the traffic conditions, road quality, or geographical factors. To maximize the profit we must find the most efficient route for delivery vehicles to minimize the time or distance traveled while reaching customer locations from the central warehouse. At this time the problem of the shortest path begin. To solve that we have a plenty of choice like Dijkstra's, Bellman-Ford, or even the Floyd-Warshall algorithm. But in this report we will study about the Floyd-Warshall algorithm. By addressing the shortest path problem in this delivery network scenario, the company can enhance its overall operational efficiency, reduce fuel costs, and improve customer satisfaction by delivering products in the shortest possible time. This application exemplifies how the shortest path problem is not merely an abstract concept but a practical and impactful challenge in various real-world contexts.

The Floyd-Warshall algorithm, a dynamic programming approach to solving the all-pairs shortest path problem in graph theory, was introduced independently by two renowned computer scientists, Robert Floyd and Stephen Warshall, in the early 1960s. The algorithm's development is a significant milestone in the history of computer science and graph theory.The Floyd-Warshall algorithm has become a classic in graph theory and algorithms, known for its simplicity and generality. Its significance lies in its ability to find the shortest paths between all pairs of vertices in a graph, regardless of whether the edges have positive or negative weights.

Over the years, the Floyd-Warshall algorithm has found widespread application in various fields, including network routing, transportation planning, and optimization problems. While its time complexity is cubic in the number of vertices, making it less efficient for large graphs, its versatility and ease of implementation make it a valuable tool for solving certain types of shortest path problems. The contributions of Floyd and Warshall to the development of this algorithm have left an enduring impact on the field of computer science and algorithm design.

3 Theory

3.1 Algorithm Description

First thing to do here is the pseudo code for this Algorithm and after that I will explain it in detail

```
Create a |V| x |V| matrix, M, that will describe the distances between vertices
For each cell (i, j) in M:
    if i == j:
        M[i][j] = 0
    if (i, j) is an edge in E:
        M[i][j] = weight(i, j)
    else:
        M[i][j] = infinity
for k from 1 to |V|:
    for i from 1 to |V|:
        for j from 1 to |V|:
            if M[i][j] > M[i][k] + M[k][j]:
                M[i][j] = M[i][k] + M[k][j]
```

To do the Floyd-warshall algorithm we have several step because we must solve it recursively.

Consider a graph, G = V, E where V is the set of all vertices present in the graph and E is the set of all the edges in the graph. The graph, G, is represented in the form of an adjacency matrix, A, that contains all the weights of every edge connecting two vertices.

To initiate the Floyd-Warshall algorithm, the first step involves the construction of an adjacency matrix, denoted as A, encapsulating the value associated with each edge within the given graph. For vertices connected by a path, the matrix is populated with the respective edge. In instances where no direct path exists between two vertices, the matrix entry is designated as ∞ to signify an infinite distance, reflecting the absence of a connecting edge. This meticulously crafted adjacency matrix serves as the foundational data structure upon which the algorithm systematically computes the shortest paths between all pairs of vertices in the subsequent phases.

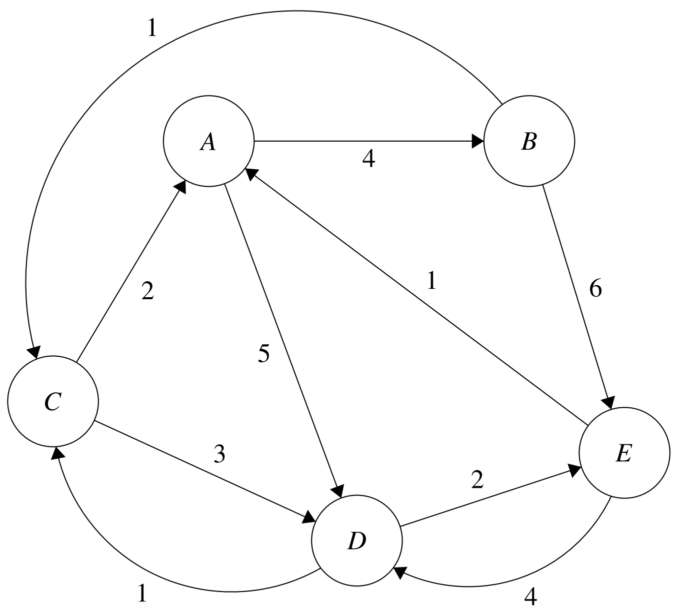
Step 2, we derive another adjacency matrix A_1 from A keeping the first row and first column of the original adjacency matrix intact in A_1. And for the remaining values, say $A_1[i, j]$, if $A[i, j] > A[i,k]+A[k, j]$ then replace $A_1[i, j]$ with $A[i,k]+A[k, j]$. Otherwise, do not change the values. Here, in this step, k=1 (first vertex acting as pivot).

For this step, Repeat Step 2 for all the vertices in the graph by changing the k value for every pivot vertex until the final matrix is achieved.

Finally, The final adjacency matrix obtained is the final solution with all the shortest paths.

3.2 Example

Suppose we have a graph shown below:



- **Step 1:** Initialize the Distance[][] matrix using the input graph such that Distance[i][j]= weight of edge from i to j, also Distance[i][j] = Infinity if there is no edge from i to j.

	A	**B**	**C**	**D**	**E**
A	0	4	∞	5	∞
B	∞	0	1	∞	6
C	2	∞	0	3	∞
D	∞	∞	1	0	2
E	1	∞	∞	4	0

- **Step 2:** Treat node A as an intermediate node and calculate the Distance[][] for every i,j node pair using the formula: = Distance[i][j] = minimum (Distance[i][j], (Distance from i to A) + (Distance from A to j)) = Distance[i][j] = minimum (Distance[i][j], Distance[i][A] + Distance[A][j])

	A	**B**	**C**	**D**	**E**
A	0	4	∞	5	∞
B	∞	0	1	∞	6
C	2	6	0	3	∞
D	∞	∞	1	0	2
E	1	5	∞	4	0

- **Step 3:** Treat node B as an intermediate node and calculate the Distance[][] for every i,j node pair using the formula:

= Distance[i][j] = minimum (Distance[i][j], (Distance from i to B) + (Distance from B to j))

= Distance[i][j] = minimum (Distance[i][j], Distance[i][B] + Distance[B][j])

	A	**B**	**C**	**D**	**E**
A	0	4	5	5	10
B	∞	0	1	∞	6
C	2	6	0	3	12
D	∞	∞	1	0	2
E	1	5	6	4	0

- **Step 4:** Treat node C as an intermediate node and calculate the Distance[][] for every i,j node pair using the formula:

  = Distance[i][j] = minimum (Distance[i][j], (Distance from i to C) + (Distance from C to j))

  = Distance[i][j] = minimum (Distance[i][j], Distance[i][C] + Distance[C][j])

	A	**B**	**C**	**D**	**E**
A	0	4	5	5	10
B	3	0	1	4	6
C	2	6	0	3	12
D	3	7	1	0	2
E	1	5	6	4	0

- **Step 5:** Treat node D as an intermediate node and calculate the Distance[][] for every i,j node pair using the formula:

  = Distance[i][j] = minimum (Distance[i][j], (Distance from i to D) + (Distance from D to j))

  = Distance[i][j] = minimum (Distance[i][j], Distance[i][D] + Distance[D][j])

	A	**B**	**C**	**D**	**E**
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

- **Step 6:** Treat node E as an intermediate node and calculate the Distance[][] for every i,j node pair using the formula:

  = Distance[i][j] = minimum (Distance[i][j], (Distance from i to E) + (Distance from E to j))

  = Distance[i][j] = minimum (Distance[i][j], Distance[i][E] + Distance[E][j])

	A	**B**	**C**	**D**	**E**
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

- **Step 7:** Since all the nodes have been treated as an intermediate node, we can now return the updated Distance[][] matrix as our answer matrix.

	A	**B**	**C**	**D**	**E**
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

3.3 Real-world Applications

The versatility of the Floyd-Warshall algorithm makes it applicable to a wide range of real-world problems where determining the shortest paths between all pairs of locations is crucial for optimization and efficiency. Its ability to handle graphs with both positive and negative edge weights further contributes to its usefulness in diverse applications.

The Floyd-Warshall algorithm is a powerful tool for finding the shortest paths between all pairs of nodes in a network. It has many applications in different fields, such as computer science, engineering, and mathematics. For example, in computer networks, the algorithm can be used to optimize the routing of data packets between routers or nodes, ensuring efficient data transmission and minimizing network congestion. In traffic engineering, the algorithm can be used to model and analyze the flow of vehicles in road networks, helping to optimize traffic signal timings, plan road expansions, and improve overall traffic management. The algorithm can also be used in GPS navigation systems, where it can calculate the shortest route between two locations. Another interesting application is in the board game Settlers of Catan, where players use a simplified version of Dijkstra's algorithm, which is a special case of the Floyd-Warshall algorithm, to find the shortest path between their settlements.

4 Code

4.1 Solution

```
% Example Graph
D = [
    0, 4, inf, 5, inf;
    inf, 0, 1, inf, 6;
    2, inf, 0, 3, inf;
    inf, inf, 1, 0, 2;
    1, inf, inf, 4, 0;
];
[d_result]= floydWarshall(D);
disp("The answer matrix: ")
disp(d_result);


function [D] = floydWarshall(D)
for k = 1:length(D)
  D = min(D,D(:,k) + D(k,:));
  disp("Intermediate node: ");
  disp(k);
  disp(D);
end
end
```

4.2 Explanation

4.2.1 Main function

```
% Example Graph
D = [
    0, 4, inf, 5, inf;
    inf, 0, 1, inf, 6;
    2, inf, 0, 3, inf;
    inf, inf, 1, 0, 2;
    1, inf, inf, 4, 0;
];
```

This is the adjacency matrix of the example graph. The matrix can be replaced by a different matrix representing the corresponding graph.

```
[d_result]= floydWarshall(D); % Function call
```

```
disp("The answer matrix: ")
disp(d_result);
```

These 2 lines display the result matrix, which represent the shortest path between all the pairs of vertices

4.2.2 Floyd-Warshall function

```
function [D] = floydWarshall(D)
for k = 1:length(D)
  D = min(D,D(:,k) + D(k,:));
```

- The first line declares a function named ”floydWarshall” that takes a matrix D as an input and returns 1 matrix D. The function aims to find the shortest paths between all pairs of vertices in a graph represented by the distance matrix D.

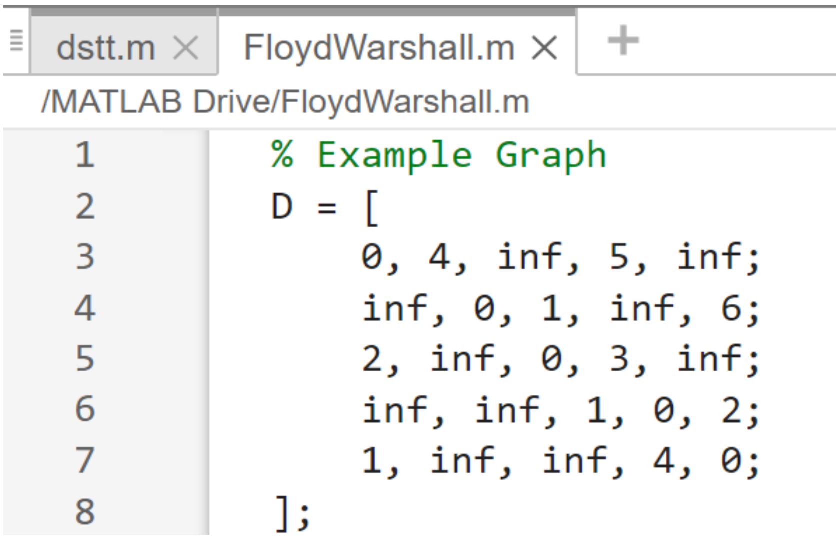
- The second line starts a loop that iterates over all vertices in the graph, represented by the variable k.
- The third line updates the distance matrix D by considering the possibility of shorter paths through the vertex k. It calculates the element-wise minimum between the current distances (D) and the sum of distances from vertex k to all other vertices both horizontally and vertically.

```
disp("Intermediate node: ");
  disp(k);
  disp(D);
```

These 3 lines display the matrix after taking node kth as the intermediate node.

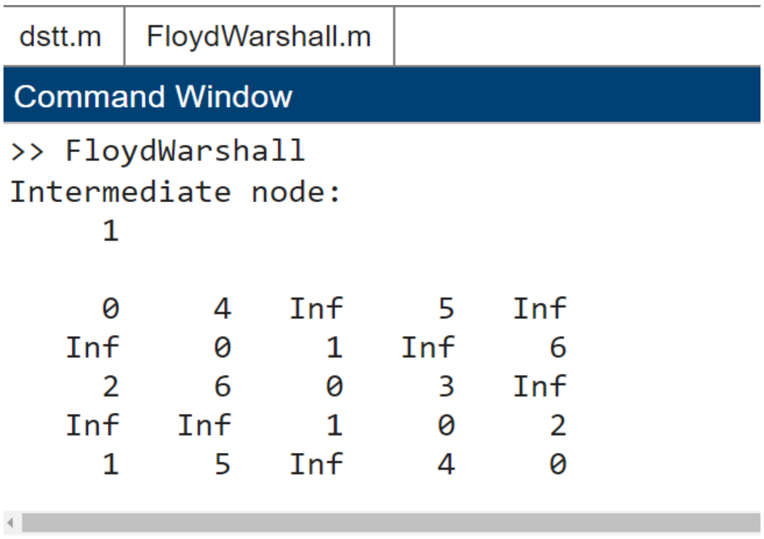
4.3 Test and result

In this section, we will take the graph in Section 3.2 as an example. From the graph we can construct the input graph as below:



```
% Example Graph
D = [
    0, 4, inf, 5, inf;
    inf, 0, 1, inf, 6;
    2, inf, 0, 3, inf;
    inf, inf, 1, 0, 2;
    1, inf, inf, 4, 0;
];
```

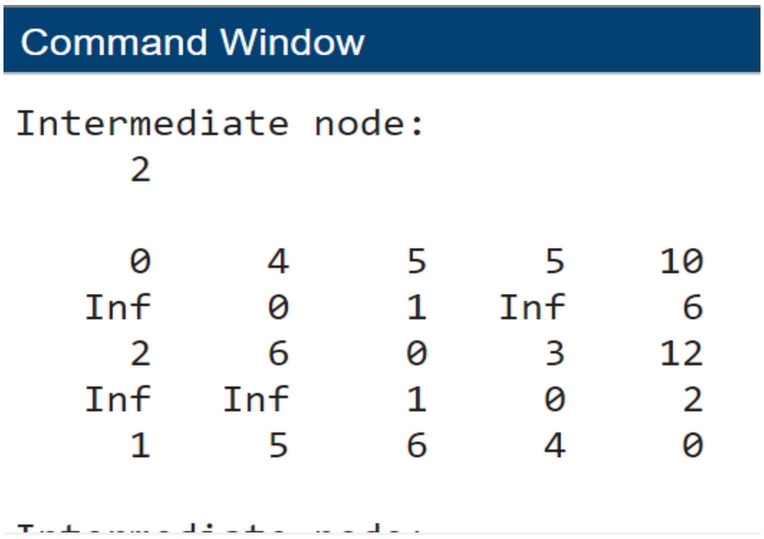
- The matrix after finishing loop 1:



```
>> FloydWarshall
Intermediate node:
     1

     0     4   Inf     5   Inf
   Inf     0     1   Inf     6
     2     6     0     3   Inf
   Inf   Inf     1     0     2
     1     5   Inf     4     0
```

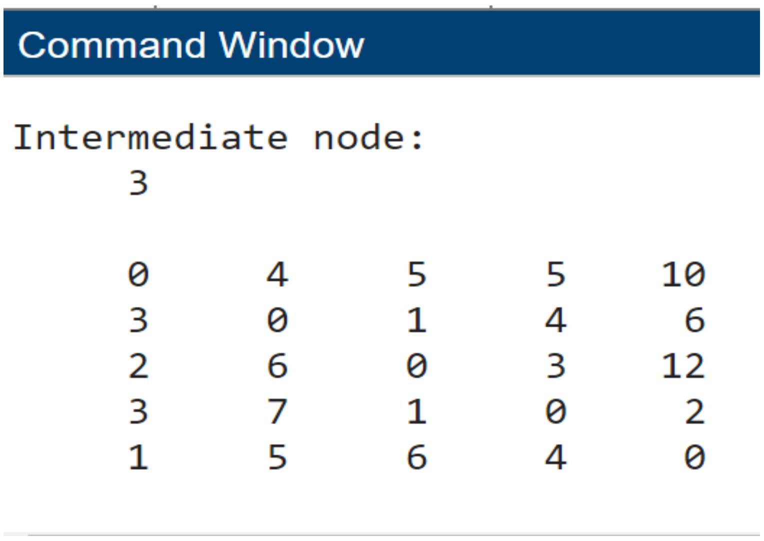
- The matrix after finishing loop 2:



```
Intermediate node:
     2

     0     4     5     5    10
   Inf     0     1   Inf     6
     2     6     0     3    12
   Inf   Inf     1     0     2
     1     5     6     4     0
```

- The matrix after finishing loop 3:

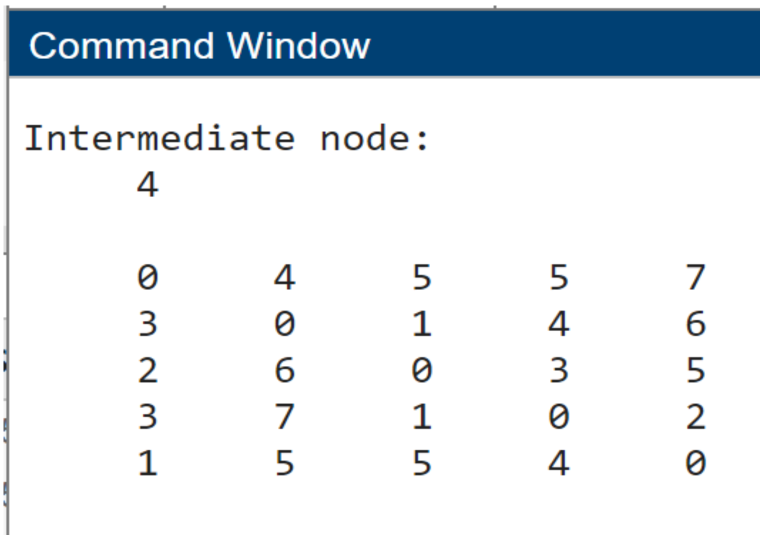


```
Command Window

Intermediate node:
     3

     0     4     5     5    10
     3     0     1     4     6
     2     6     0     3    12
     3     7     1     0     2
     1     5     6     4     0
```

- The matrix after finishing loop 4:

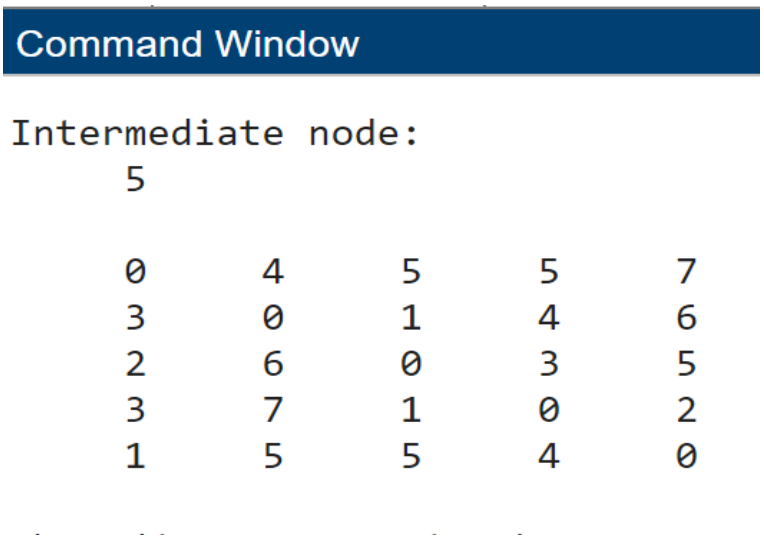


```
Command Window

Intermediate node:
     4

     0     4     5     5     7
     3     0     1     4     6
     2     6     0     3     5
     3     7     1     0     2
     1     5     5     4     0
```

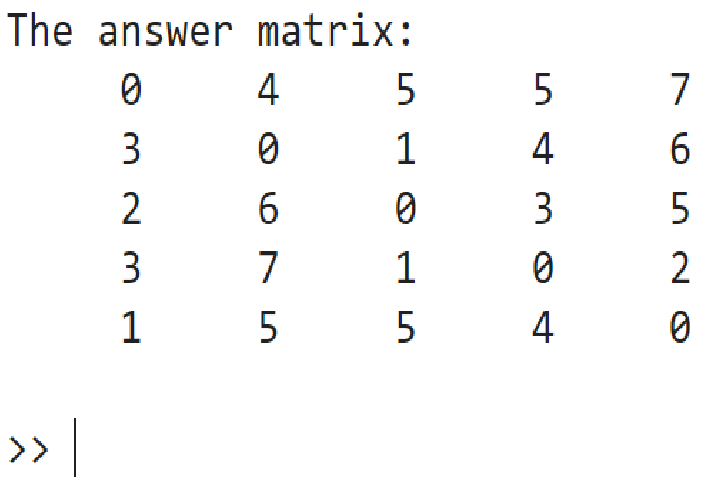
- The matrix after finishing loop 5:



```
Intermediate node:
     5

     0     4     5     5     7
     3     0     1     4     6
     2     6     0     3     5
     3     7     1     0     2
     1     5     5     4     0
```

- The final matrix is the answer matrix:



```
The answer matrix:
     0     4     5     5     7
     3     0     1     4     6
     2     6     0     3     5
     3     7     1     0     2
     1     5     5     4     0

>> 
```

5 Discussion

5.1 Applicability and Limitations

Understanding the applicability and limitations of the Floyd-Warshall algorithm is crucial for selecting the most suitable algorithm based on the characteristics of the problem at hand and the nature of the graph being analyzed.

The Floyd-Warshall algorithm is specifically designed for finding the shortest paths between all pairs of vertices in a weighted graph. It pretty goods in such a scenarios where a comprehensive understanding of the shortest paths is required. One significant strength of the algorithm is the ability to handle graphs with both positive and negative edge weights. This makes it suitable for various application, including scenarios where negative weights may represent gains. The algorithm can efficiently determines whether there is a path between all pairs of vertices, make it has a lot more application than just solve shortest path problem.

Despite having the ease in implementation and very effective for both sparse and dense graphs, The most noticeable limitation is the algorithm's cubic time complexity (O(V^3)), (V is the number of vertices). This can be a drawback for large graphs, making it less efficient compared to other algorithms for specific use cases. While the algorithm can find all-pairs shortest paths, it is not optimal for solving the single-source shortest path problem. Other algorithms like Dijkstra's or Bellman-Ford are more efficient in such cases. In addition, one major draw back of the Floyd-Warshall algorithm is designed for static graphs, and its application to dynamic graphs with frequently changing edge weights can be inefficient. The need to reconstruct the entire matrix for each change in the graph structure is a computational overhead.

References

[1] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L. (1990). Introduction to Algorithms (1st ed.). MIT Press and McGraw-Hill. ISBN 0-262-03141-8. See in particular Section 26.2, "The Floyd–Warshall algorithm", pp. 558–565 and Section 26.4, "A general framework for solving path problems in directed graphs", pp. 570–576.

[2] Design and Analysis - Floyd Warshall Algorithm. Access from https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_floyd_warshall_algorithm.htm

[3] Weisstein, Eric W. "Floyd-Warshall Algorithm". Access from https://mathworld.wolfram.com/Floyd-WarshallAlgorithm.html.