23XT42 – COMPUTER NETWORKS

TERM PAPER PRESENTATION

FLOYD-WARSHALL ALGORITHM

II Year - MSc Theoretical Computer Science

Applied Mathematics and Computational Sciences

PSG College Of Technology

Date of Submission: 05.04.2025

BY

Roll Number	Name
23PT01	Aakash Velusamy
23PT14	Kabilan S

ABSTRACT

This term paper presentation examines the Floyd-Warshall algorithm, which determines the shortest paths between every pair of points in a weighted graph. We start by laying out the foundations of graphs, where points are joined by lines that carry costs such as time or distance, and then describe how this algorithm carefully finds the best routes through a step-by-step process. The report traces its beginnings back to the 1960s, when Robert Floyd, Stephen Warshall, and Bernard Roy developed its key ideas, and explains its role in graph theory and computer networks. We explore the specific challenge it addresses - finding all possible shortest paths - and compare it with other methods that focus on single starting points. To make it clear, we include an example with a small graph, showing how the distances change as the algorithm works. The study also covers its practical uses in computer networks, such as guiding data between machines, spreading traffic evenly, and understanding network layouts. We discuss its strengths, including its ability to handle paths with negative costs, as well as its drawbacks, like being slow for very large setups, and suggest ways to make it better. Examples from real situations, such as data centres, sensor networks in factories, and internet traffic management, demonstrate its importance. In the end, we review its value for planning network connections and look ahead to how it might improve in the future, providing a thorough understanding of this essential method for linking systems effectively.

ACKNOWLEDGEMENTS

We would like to express my heartfelt gratitude to everyone who supported me during the preparation and completion of this term paper. Their guidance, encouragement, and assistance meant a great deal to us throughout this journey.

First and foremost, we are deeply thankful to PSG College of Technology for providing a platform to explore and present this work. The institution’s commitment to academic excellence and encouragement for student research played a key role in shaping this paper.

We are especially grateful to my faculty mentor, Dr. P. Navaneethan, for his valuable guidance, patience, and profound feedback. His mentorship helped us think more critically and structure our ideas with greater clarity.

We would also like to thank everyone who contributed to the study - your inputs and cooperation made this research possible and meaningful.

Lastly, a special thank you goes to our family and friends. Their constant support, understanding, and encouragement helped us stay focused and motivated throughout this process.

This term paper is the result of many small but meaningful contributions, and we are truly grateful to all who were part of this journey.

TABLE OF CONTENTS

1. INTRODUCTION

1.1. What is the Floyd-Warshall Algorithm?

The Floyd-Warshall algorithm is a computational procedure developed to determine the shortest paths between all pairs of vertices within a weighted graph. Envision a scenario where one must identify the most efficient routes between every pair of cities in a region - not merely from a single starting point, but comprehensively across all possible combinations. This algorithm accomplishes precisely that by systematically evaluating all potential pathways, leveraging intermediate vertices to refine distance estimates until the optimal paths are established. In the realm of computer networks, it translates to calculating the quickest routes for data transmission between all devices, such as routers or servers, ensuring efficient communication across complex systems. Its methodical approach guarantees that no shorter path is overlooked, making it a cornerstone for applications requiring a global perspective on connectivity.

1.2. Historical Origin

The origins of the Floyd-Warshall algorithm are rooted in the early 1960s, a period marked by significant advancements in computational theory. Stephen Warshall laid the initial groundwork in 1962 with his paper, "A Theorem on Boolean Matrices," published in the *Journal of the ACM*. His work focused on computing the transitive closure of a graph - essentially determining whether a path exists between any two vertices, even through multiple hops. Concurrently, Robert Floyd expanded this concept in his 1962 publication, "Algorithm 97: Shortest Path," in *Communications of the ACM*, introducing weights to the edges and transforming the method into a solution for finding the shortest paths. Around the same time, Bernard Roy in France independently proposed similar ideas, contributing to what is sometimes referred to as the Roy-Floyd-Warshall algorithm. These contributions were not isolated; they built upon earlier graph theory developments by pioneers like Richard Bellman, reflecting a collaborative evolution of ideas. This historical synergy underscores the algorithm's robust foundation, as documented in foundational texts like Cormen et al. (2009).

1.3. Where It Fits in Graph Theory and Networking

Within graph theory, the Floyd-Warshall algorithm addresses the all-pairs shortest path (APSP) problem, distinguishing itself from single-source shortest path (SSSP) methods such as Dijkstra's or Bellman-Ford algorithms. It employs dynamic programming - a strategy of solving problems by breaking them into smaller subproblems and reusing their solutions - to construct a comprehensive distance matrix. This matrix encapsulates the shortest path costs between every pair of vertices, offering a holistic view of graph connectivity. In computer networking, this capability is invaluable. Networks, whether local intranets or expansive internet backbones, require efficient routing strategies to minimize latency and optimize resource utilization. The algorithm's ability to precompute all pairwise shortest paths aligns seamlessly with tasks such as generating static routing tables, analysing network topology, or managing traffic flow in dense environments. Its relevance is affirmed in seminal works like *Computer Networks* by Tanenbaum and Wetherall (2011), positioning it as a critical tool at the intersection of theoretical computation and practical network engineering.

2. PREREQUISITES

2.1. Basic Graph Theory Concepts

Graph theory forms the bedrock of the Floyd-Warshall algorithm, providing the mathematical framework to model relationships and pathways. A graph, denoted as $G = (V, E)$, consists of a set of vertices V representing entities (e.g., cities, devices) and a set of edges E indicating connections between them. This abstraction allows us to visualize and analyse systems where connectivity is paramount, laying the foundation for path-finding algorithms.

2.1.1. Directed and Undirected Graphs

Graphs are categorized into directed and undirected types based on edge characteristics. A directed graph, or digraph, features edges with a specific direction, denoted as (u, v), signifying a one-way connection from vertex u to vertex v. For instance, in a network, this might represent a unidirectional data link where traffic flows from one router to another without a return path. Conversely, an undirected graph has bidirectional edges, where (u, v) implies (v, u), akin to a two-way communication channel such as a fibre optic cable supporting data in both directions. Understanding these distinctions is essential, as the Floyd-Warshall algorithm can operate on both types, adapting its computations to the graph structure. In networking, directed graphs often model asymmetric connections, while undirected graphs reflect symmetric infrastructures, influencing how paths are calculated.

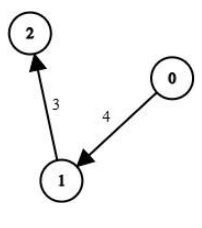
2.1.2. Weighted Graphs

A weighted graph extends the basic graph by assigning a numerical value, or weight $w(u, v)$ to each edge. These weights typically represent a cost - such as distance, time, or resource expenditure - associated with traversing the edge. For example, a weight of 5 between vertices A and B might indicate 5 kilometres or 5 milliseconds of delay. Weights are generally non-negative, though the Floyd-Warshall algorithm can accommodate negative weights provided no negative cycles (loops with a total negative weight) exist. In computer networks, weights are critical, often denoting latency (e.g., 10 ms between routers), bandwidth cost, or hop count. This concept is fundamental because the algorithm uses these values to determine the most efficient paths, making weighted graphs the core structure it operates upon.

2.1.3. Representation: Adjacency Matrix and Adjacency List

Graphs can be represented in two primary forms: an adjacency matrix or an adjacency list, each with distinct advantages. An adjacency matrix is a square $|V| \times |V|$ array where each entry $A[i][j]$ records the weight of the edge from vertex i to vertex j. If no edge exists, the entry is set to infinity (∞), and if $i = j$, it is 0 (no cost to stay at the same vertex). Consider a 3-vertex graph:

X	0	1	2
0	0	4	∞
1	∞	0	3
2	∞	∞	0



An adjacency list, alternatively, maintains a list for each vertex, detailing its neighbours and their weights - e.g., 0: (1), 1: (2). The Floyd-Warshall algorithm opts

for the adjacency matrix because its grid format facilitates direct access and updates to distances, which is particularly efficient for dense graphs and network applications where all pairwise relationships are analysed. While adjacency lists are memory-efficient for sparse graphs, the matrix's structure aligns with the algorithm's iterative process, making it the preferred choice.
Shortest Path Problem

2.2. Shortest Path Problem

The shortest path problem seeks to identify the path between two vertices with the minimum total weight. For a path $P = \{v_0, v_1, \ldots, v_k\}$ from v_0 to v_k, the cost is computed as $w(P) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$. For instance, if the path from A to C via B has weights $w(A, B) = 4$ and $w(B, C) = 3$, the total is 7. The all-pairs shortest path (APSP) problem extends this to every pair of vertices, requiring a solution that encompasses all possible start and end points. In networking, this translates to finding the fastest routes for data between all devices - a router in Chennai to one in Bangalore, and every other combination - ensuring optimal communication across the system.

2.3. Matrix Operations

The Floyd-Warshall algorithm hinges on manipulating matrices, necessitating a basic understanding of matrix operations. A matrix is a grid of numbers, and the algorithm uses a distance matrix D where $D[i][j]$ represents the shortest known distance from vertex i to j. The key operation is the update rule:

$$D[i][j] = min(D[i][j], D[i][k] + D[k][j])$$

This compares the current distance with the potential distance via an intermediate vertex k, using addition and minimum selection. For example, if $D[i][j] = 10$, $D[i][k] = 4$, and $D[k][j] = 3$, the new $D[i][j] = 7$. Proficiency in indexing (rows and columns) and performing these calculations is crucial for following the logic of the algorithm.

2.4. Dynamic Programming Basics

Dynamic programming is a problem-solving technique that tackles complex challenges by solving smaller subproblems and storing their results for reuse. Consider planning a multi-city trip: instead of recalculating the best route from scratch each time, you note the shortest path to each city as you progress, building on previous decisions. The Floyd-Warshall algorithm applies this by incrementally considering each vertex as an intermediate point, refining distances based on prior computations. This approach, grounded in Bellman's principle of optimality (which states that optimal solutions contain optimal sub solutions), ensures efficiency by avoiding redundant work. In networks, this mirrors how routing tables evolve, making it a natural fit.

2.5. Real-World Relevance in Computer Networks

These concepts are not abstract - they directly apply to computer networks, the backbone of modern communication. Graphs model network topology, with vertices as routers or servers and edges as cables or wireless links. Weights represent real costs - like the 5ms delay between two routers or the bandwidth needed for a video call. The shortest path problem ensures data travels efficiently, reducing delays that frustrate users. The Floyd-Warshall algorithm's matrix-based, all-pairs solution is particularly relevant for tasks like precomputing routing tables in a corporate network or optimizing traffic in a data centre. As highlighted in *Data Communications and Networking* by Forouzan (2013), it provides a comprehensive tool for network engineers to maintain seamless connectivity.

3. UNDERSTANDING

3.1. What Does "All-Pairs Shortest Path" Mean?

The all-pairs shortest path (APSP) problem involves calculating the shortest distance between every possible pair of vertices in a graph. For a graph with vertices labelled 0, 1, 2, and 3, this means finding the shortest paths from 0 to 1, 0 to 2, 0 to 3, 1 to 0, 1 to 2, 1 to 3, 2 to 0, 2 to 1, 2 to 3, 3 to 0, 3 to 1, and 3 to 2 - 12 pairs total. Each path's cost is the sum of edge weights along the route, and the goal is a complete table of these minimum distances. This exhaustive approach contrasts with focusing on a single starting point, offering a full map of connectivity that's invaluable when every relationship matters.

3.2. Significance in Routing Optimization

In computer networks, efficient routing is the lifeline of performance. Data packets - whether emails, video streams, or website requests - must travel from source to destination with minimal delay. The APSP problem matters because it provides a global view: every router knows the fastest way to reach every other router, not just its immediate neighbours. This is critical for prebuilding routing tables in a stable network, like a university's LAN, where devices need consistent, quick paths. It also aids optimization - say, in an internet service provider's network, where engineers balance traffic to prevent bottlenecks. By knowing all shortest paths, they can reroute data around a busy link, keeping everything flowing smoothly. This comprehensive insight, as noted in *Computer Networking: A Top-Down Approach* by Kurose and Ross (2020), enhances reliability and user experience in complex systems.

3.3. Comparison with Single Source Shortest Path Algorithms

Single-source shortest path (SSSP) algorithms, such as Dijkstra's, focus on paths from one vertex to all others. For example, Dijkstra's starts at Router 0 and computes distances to Routers 1, 2, and 3, with a time complexity of $O((|V| + |E|) . log|V|$ using a priority queue. It's efficient for one source - perfect if you only need paths from a single server. However, to get all pairs, you'd run it $|V|$ times, costing $O(|V| \cdot (|V| + |E|) . log|V|)$ - a lot for a big network. The Floyd-Warshall algorithm, with $O(|V|^3)$, does it in one pass, making it faster for dense graphs where edges are plentiful ($|E| \approx |V|^2$). In sparse graphs (fewer edges), Dijkstra's repeated runs might edge out, but Floyd-Warshall's single-run completeness is a win for network-wide planning, as explained in *Introduction to Algorithms* by Cormen et al. (2009).

4. ALGORITHM

4.1. Step-by-Step Explanation

The Floyd-Warshall algorithm operates like a meticulous planner refining a travel itinerary. It begins with a matrix of direct distances between vertices, then examines each vertex as a potential stopover, updating distances if a detour proves shorter. This iterative process continues until the matrix reflects the shortest path between every pair, ensuring no opportunity for improvement is missed. It's a patient, thorough approach that builds confidence in the results.

4.2. Pseudocode

The algorithm's structure is captured in this formal pseudocode:

```
Input: Weighted graph G with n vertices, adjacency matrix W
Output: Distance matrix D with shortest path distances

1. Set D[0][i][j] = W[i][j] for all i, j from 1 to n
   - If no edge (i, j)∈ E, set D[0][i][j] = ∞
   - If i = j, set D[0][i][i] = 0

2. For k = 1 to n:
     For i = 1 to n:
       For j = 1 to n:
         If D[k-1][i][k] + D[k-1][k][j] < D[k-1] [i][j]:
           D[k][i][j] = D[k-1][i][k] + D[k-1][k][j]
         Else:
           D[k][i][j] = D[k-1][i][j]

3. Return D[n]
```

4.3. Explanation of the Triple Nested Loop

The algorithm employs three nested loops, each serving a distinct purpose:

- Outer Loop (k): Iterates over each vertex k (1 to n) as a candidate intermediate point, testing whether it offers a shorter path.
- Middle Loop (i): Loops through all source vertices i (rows), representing starting points.
- Inner Loop (j): Loops through all destination vertices j (columns), covering end points.

For every k, it examines every pair (i, j), performing $|V| \times |V| \times |V|$ operations. This exhaustive check ensures all possible detours are evaluated, mirroring how a network engineer might explore every routing option.

4.4. How the Intermediate Node (k) Concept Works

The intermediate vertex k is the heart of the algorithm. For each pair (i, j), it calculates the distance via k as $D[i][k] + D[k][j]$ $D[i][k] + D[k][j]$ $D[i][k] + D[k][j]$ and compares it to the current $D[i][j]$. If the detour is shorter, it updates $D[i][j]$. For instance, if $D[i][j] = 15$, $D[i][k] = 6$, and $D[k][j] = 5$, the new distance is 11, so $D[i][j]$ becomes 11. This leverages the triangle inequality $d(i, j) \leq d(i, k) + d(k, j)$, refining paths incrementally. By cycling through all k, it ensures every possible intermediate is considered, much like checking every connecting flight to find the fastest travel route.

4.5. Time and Space Complexity Analysis

- **Time Complexity**: The triple loop yields $O(|V|^3)$ operations. For 5 vertices, that is 5 × 5 × 5 = 125 steps; for 100, it's 1,000,000. It's consistent regardless of edge count, making it efficient for dense graphs but sluggish for sparse ones.
- **Space Complexity**: The distance matrix requires $|V| \times |V|$ slots - $O(|V|^2)$. For 5 vertices, that's 25 entries; for 100, it's 10,000. In-place updates keep it at one matrix, though adding a path-tracking matrix double this to $2 \cdot |V|^2$. This is feasible for small to medium networks but scales poorly for massive ones (Cormen et al., 2009).

5. EXAMPLE

5.1. Initial Graph Representation

We begin with the initial adjacency matrix D^0, representing the direct distances between nodes in a directed graph (∞ = no direct edge):

$$D^0 = \begin{matrix} \mathbf{0} & \mathbf{4} & \mathbf{10} & \infty \\ \infty & \mathbf{0} & \mathbf{3} & \mathbf{8} \\ \infty & \infty & \mathbf{0} & \mathbf{2} \\ \infty & \infty & \infty & \mathbf{0} \end{matrix}$$

This reflects direct connections: 4 ms from 0 to 1, no direct path (∞) from 0 to 3, etc.

5.2. Step-by-Step Matrix Updation

5.2.1. $k = 0$ – Using vertex 0 as intermediate:

- **1 to 2**:

  Direct = 3, via 0 = $\infty + 10 = \infty$, stays 3.
- **2 to 1**:

  Direct = ∞, via $0 = \infty + 4 = \infty$, stays ∞.
- **1 to 3**:

  Direct = 8, via $0 = \infty + \infty = \infty$, stays 8.

Vertex 0 has limited outgoing edges (to 1 and 2), so no updates occur.

$$D^1 = \begin{matrix} \mathbf{0} & \mathbf{4} & \mathbf{10} & \infty \\ \infty & \mathbf{0} & \mathbf{3} & \mathbf{8} \\ \infty & \infty & \mathbf{0} & \mathbf{2} \\ \infty & \infty & \infty & \mathbf{0} \end{matrix}$$

5.2.2. $k = 1$ - Using vertex 1 as intermediate:

- **0 to 2**:

  Direct = 10, via $1 = 4 + 3 = 7$ → update to 7 (path: 0 → 1 → 2)
- **0 to 3**:

  Direct = ∞, via $1 = 4 + 8 = 12$ → update to 12 (path: 0 → 1 → 3)
- **2 to 3**:

  Direct = 2, via $1 = \infty + 8 = \infty$, stays 2
- **2 to 0**:

  Direct = ∞, via $1 = \infty + \infty = \infty$, stays ∞

Vertex 1 connects 0 to 2 and 3, improving paths.

$$D^2 = \begin{matrix} \mathbf{0} & \mathbf{4} & \mathbf{7} & \mathbf{12} \\ \infty & \mathbf{0} & \mathbf{3} & \mathbf{8} \\ \infty & \infty & \mathbf{0} & \mathbf{2} \\ \infty & \infty & \infty & \mathbf{0} \end{matrix}$$

5.2.3. $k = 2$ - Using vertex 2 as intermediate:

- **0 to 3**:

  Direct = 12, via 2 $= 7 + 2 = 9$ → update to 9 (path: 0 → 1 → 2 → 3)

- **1 to 3**:

  Direct = 8, via 2 $= 3 + 2 = 5$ → update to 5 (path: 1 → 2 → 3)

- **0 to 1**:

  Direct = 4, via 2 $= 7 + \infty = \infty$, stays 4

- **1 to 0**:

  Direct = ∞, via 2 $= 3 + \infty = \infty$, stays ∞

Vertex 2 refines paths to 3 via its edge to 3.

$$D^3 = \begin{matrix} \mathbf{0} & \mathbf{4} & \mathbf{7} & \mathbf{9} \\ \infty & \mathbf{0} & \mathbf{3} & \mathbf{5} \\ \infty & \infty & \mathbf{0} & \mathbf{2} \\ \infty & \infty & \infty & \mathbf{0} \end{matrix}$$

5.2.4. $k = 3$ - Using vertex 3 as intermediate:

- **0 to 1**:

  Direct = 4, via 3 $= 9 + \infty = \infty$, stays 4

- **1 to 0**:

  Direct = ∞, via 3 $= 5 + \infty = \infty$, stays ∞

- **2 to 0**:

  Direct = ∞, via 3 $= 2 + \infty = \infty$, stays ∞

Vertex 3 has no outgoing edges improving existing paths (all incoming).

$$D^4 = \begin{matrix} \mathbf{0} & \mathbf{4} & \mathbf{7} & \mathbf{9} \\ \infty & \mathbf{0} & \mathbf{3} & \mathbf{5} \\ \infty & \infty & \mathbf{0} & \mathbf{2} \\ \infty & \infty & \infty & \mathbf{0} \end{matrix}$$

5.3. Final Result Interpretation

Final matrix D^4:

$$D^4 = \begin{matrix} \mathbf{0} & \mathbf{4} & \mathbf{7} & \mathbf{9} \\ \infty & \mathbf{0} & \mathbf{3} & \mathbf{5} \\ \infty & \infty & \mathbf{0} & \mathbf{2} \\ \infty & \infty & \infty & \mathbf{0} \end{matrix}$$

- 0 to 3: 9 ms (path: 0 → 1 → 2 → 3)
- 1 to 3: 5 ms (path: 1 → 2 → 3)
- 2 to 3: 2 ms (direct)

In a network, this matrix tells each router the fastest path to every other. For example, Router 0 to Router 3 takes 9 ms via 1 and 2. The ∞ entries indicate no return paths due to directionality - typical in directed network graphs.

6. APPLICATIONS IN COMPUTER NETWORKS

6.1. Link-State Routing Protocols

Protocols like OSPF (Open Shortest Path First) depend on a full map of the network. Floyd-Warshall can precompute all-pairs shortest paths, forming a complete routing table. Ideal for:

- Small or medium-sized LANs
- Static networks with minimal topology changes.

(Source: *Computer Networks*, Tanenbaum & Wetherall, 2011)

6.2. Network Topology Analysis

Floyd-Warshall uncovers **how nodes are interconnected**, helping:

- Detect bottlenecks or critical hubs
- Plan upgrades or redundancy
- Visualize entire network structure

(Source: *Computer Networking*, Kurose & Ross, 2020)

6.3. OSPF (Open Shortest Path First)

Though OSPF uses Dijkstra at each node, Floyd-Warshall can:

- Precompute paths in static networks
- Avoid repeated real-time Dijkstra computations
- Suit small OSPF domains (e.g., < 20 routers)

This reduces router workload, aligning with link-state principles.

6.4. Load Balancing

In high-traffic backbones (e.g., ISPs), Floyd-Warshall:

- Finds alternate paths when primary is congested
- Supports multipath routing
- Aids traffic distribution in data centres

(Source: *Data Communications and Networking*, Forouzan, 2013)

7. ADVANTAGES AND LIMITATIONS

7.1. Advantages

- Handles negative weights (not negative cycles)
- Simple implementation with triple loop
- All-pairs shortest paths in one run
- Useful in routing table generation and topology visualization

7.2. Limitations

- High time complexity: $O(|V|^3)$ unsuitable for large graphs
- Space Intensive: $O(|V|^2)$ unsuitable for large graphs
- No support for negative cycles - can result in undefined paths

8. CONCLUSION

The Floyd-Warshall algorithm is a simple yet powerful way to find the shortest paths between every pair of points in a network. Instead of running a path-finding algorithm again and again, it solves everything in one go.

It works well in situations like network routing, data centre traffic planning, and even wireless sensor communication. While it's not the fastest for very large networks, it is great when the network is stable and fully connected.

In short, Floyd-Warshall helps us understand how information moves in a network and how to make that movement as efficient as possible.

9. REFERENCES

- Cormen, T. H., et al. (2009). *Introduction to Algorithms*. MIT Press.
- Tanenbaum, A. S., & Wetherall, D. J. (2011). *Computer Networks*. Pearson.
- Kurose, J. F., & Ross, K. W. (2020). *Computer Networking: A Top-Down Approach*. Pearson.
- Forouzan, B. A. (2013). *Data Communications and Networking*. McGraw-Hill.
- Singh, S. P., & Sharma, S. C. (2015). *Modification of Floyd-Warshall's Algorithm for Shortest Path Routing in Wireless Sensor Networks.*
- Programmiz - Floyd-Warshall Algorithm
- Medium – Floyd-Warshall Algorithm in Networks