EAST WEST UNIVERSITY

Project

Campus Navigating System Using Floyd - Warshell Algorithm

Submitted To

Dr. Md. Tauhid Bin Iqbal

Associate Professor

Department of Computer Science & Engineering

Submitted by

Md. Ariful Islam Opi	2023-1-60-141
MD. Abir Ahmed	2023-1-60-247
Ohiduzzaman Rayhan	2023-1-60-108
Tasnia Taher Esha	2023-1-60-173

Section: 07

Date of Submission: 24/05/2025

Group Contribution

- Md. Ariful Islam Opi - 25%
  Role: Done the code and made final corrections.

- MD. Abir Ahmed - 25%
  Role: Writing the first draft of the report.

- Ohiduzzaman Rayhan - 25%
  Role: Editing and formatting the report.

- Tasnia Taher Esha - 25%
  Role : Creating presentation slides.

Each group member contributed equally (25%) to all stages of the project, including research, writing, editing, and presentation preparation. All decisions and tasks were done collaboratively.

Introduction:

The Floyd-Warshall algorithm is a fundamental solution in graph theory for finding the shortest paths between all pairs of vertices in a weighted graph. This project explores the theoretical foundations, implementation, and practical applications of the algorithm. By employing dynamic programming, the Floyd-Warshall algorithm systematically updates a distance matrix to reflect the shortest paths through intermediate vertices. It supports graphs with negative edge weights, provided there are no negative cycles, making it suitable for a wide range of real-world problems such as network routing and urban navigation systems. The algorithm's cubic time complexity ($O(n^3)$) is efficient for dense graphs with a moderate number of nodes. This report presents a detailed analysis of the algorithm's structure, performance, and limitations, supported by implementation examples and test cases.

This Campus Navigator project uses the Floyd-Warshall algorithm to find the shortest paths between all pairs of locations on a campus. The Floyd-Warshall algorithm is used to precompute the shortest paths between every pair of campus locations (vertices in a graph). Once this precomputation is done, the navigator can quickly answer user queries about the shortest distance and the path between any two given locations.

1. **Named Locations**: Users interact with human-readable names (e.g., “Library”, “Canteen”) instead of numeric indices.
2. **Interactive Console Interface**: Prompts users to enter locations and paths,Allows real-time queries,Keeps running until the user types exit.
3. **Shortest Path Queries**: Users can request the shortest path between any two locations. Shows: Path (e.g., "A → B → C") and Total distance
4. Path Reconstruction: The program doesn’t just give the distance — it shows which path to follow, using the next matrix.
5. **Handles Redundant Input Gracefully**: If a path is entered more than once with a shorter distance, it updates to keep the shortest.
6. **Error Checking**: Prevents invalid location names.
7. **Supports Disconnected Graphs**: Uses INF to represent unreachable paths and handles “No path exists” scenarios.
8. **Directed Graph Support**: Paths are entered as one-way unless explicitly added in both directions.

Why Use a Floyd - Warshell?

1. **All-Pairs Shortest Path**: Unlike Dijkstra's or Bellman-Ford (which find paths from a single source), Floyd-Warshall computes the shortest paths between all pairs of nodes in one run.

Use case: In a campus navigator, users might ask for the shortest path between any two locations, not just from a fixed source.

2. **Preprocessing for Instant Queries**: After running Floyd-Warshall once ($O(n^3)$ time),

Instantly answer shortest-path queries in O(1) time,Reconstruct paths using a next matrix ,

Benefit: Fast real-time performance once preprocessing is complete.

3. **Handles Negative Edge Weights**: Floyd-Warshall can work with negative edge weights, as long as there are no negative cycles.

For example: If some shortcuts "save" time (e.g., elevator vs. stairs), you can represent them with negative edges — Dijkstra can't handle this, but Floyd-Warshall can.

4. **Simple and Elegant Implementation**: Its triple nested loop is simple to write and understand. It's easy to implement in just a few lines of code.

5. **Space-Efficient for Path Reconstruction** : use of a next[][] matrix allows easy reconstruction of actual paths between locations — another built-in benefit of Floyd-Warshall.

Real Life Application:

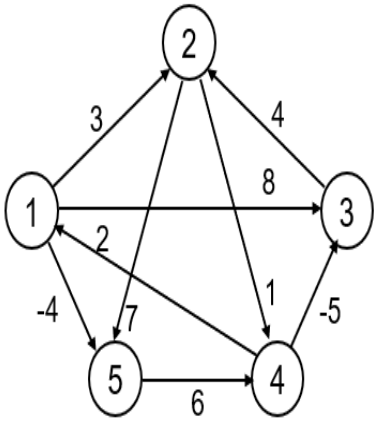
- Navigation Systems: Used in GPS and campus navigation tools to determine the shortest route between multiple locations.
- Network Routing: Optimizes data packet transmission in computer networks by finding efficient paths between nodes.
- Social Network Analysis: Identifies shortest connections between users in a social graph.
- Flight and Railway Scheduling: Helps airlines and railway systems determine optimal travel routes between multiple destinations.
- Supply Chain & Logistics: Used to optimize warehouse distribution and minimize transportation costs.

Game Development: Helps AI characters find paths through game environments efficiently.

Operation	Description	Time Complexity	Space Complexity
Insert Location	Takes input and stores location names and indices using a map.	O(1) per location	O(V) total
Insert Path	Adds directed edge between two locations with distance. Updates matrix if shorter.	$O(V^3)$	$O(V^2)$
Floyd-Warshall	Computes all-pairs shortest paths and builds the next matrix.	O(V) per query	O(1)
Shortest Path Query	Finds and prints the shortest path between any two locations using next[][].	O(m)	O(1)
Path Reconstruction	Follows next[][] from source to destination and prints the path.	O(V)	O(1)
Invalid Input Check	Ensures only known locations and valid paths are accepted.	O(1) per check	O(1)
Program Loop (Navigation)	Allows multiple shortest path queries until exit is typed.	O(V) per query	O(1)

The final Time Complexity: $O(n^3)$

Example:



	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

$D^{(1)}$

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	5	-5	0	-2
5	∞	∞	∞	6	0

$D^{(2)}$

	1	2	3	4	5
1	0	3	8	4	-4
2	∞	0	∞	1	7
3	∞	4	0	5	11
4	2	5	-5	0	-2
5	∞	∞	∞	6	0

$D^{(3)}$

0	3	8	4	-4
∞	0	∞	1	7
∞	4	0	5	11
2	-1	-5	0	-2
∞	∞	∞	6	0

$D^{(4)}$

0	3	-1	4	-4
3	0	-4	1	-1
7	4	0	5	3
2	-1	-5	0	-2
8	5	1	6	0

	1	2	3	4	5
1	0	1	-3	2	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

$P^{(5)}$

	1	2	3	4	5
1	-	3	4	5	1
2	4	-	4	2	1
3	4	3	-	2	1
4	4	3	4	-	1
5	4	3	4	5	-

Source: 5, Destination: 1

Shortest path: 8

Path: 5 …1 : 5…4…1: 5->4…1: 5->4->1

Source: 1, Destination: 3

Shortest path: -3

Path: 1 …3: 1…4…3: 1…5…4…3: 1->5->4->3

Functions:

1.Header Files:

- **#include <iostream>** This is used for input and output operations in C++. It allows you to use cin for input and cout for output.
- **#include <vector>** This provides the vector container, which is a dynamic array that can resize automatically. It's useful when you need a flexible array-like structure.
- **#include <map>** This provides the map container, which stores key-value pairs in a sorted manner. It is useful for fast lookups and organizing data in an associative way.
- **#include <climits>** This defines macros for limits of integer types, such as INT_MAX and INT MIN. It is useful for handling extreme values in calculations.
- **using namespace std;** simply allows you to use standard library components (like vector, cout, etc.) without prefixing them with std::

2. void floydWarshall(vector<vector<int>>& dist, vector<vector<int>>& next, int V)

This function Implements the Floyd-Warshall algorithm**,** Computes the shortest distance between every pair of nodes, Builds the next[][] matrix for path reconstruction.

```
void floydWarshall(vector<vector<int>>& dist, vector<vector<int>>& next, int V) {
   for (int k = 0; k < V; k++)
      for (int i = 0; i < V; i++)
         for (int j = 0; j < V; j++)
            if (dist[i][k] != INF && dist[k][j] != INF &&
               dist[i][j] > dist[i][k] + dist[k][j]) {
               dist[i][j] = dist[i][k] + dist[k][j];
               next[i][j] = next[i][k];
            }
}
```

3. void printPath(map<int, string>& locations, vector<vector<int>>& next, int u, int v)

This function Prints the actual **shortest path** from location u to location v using the next matrix, Uses a while loop to reconstruct the full path from u to v, Displays the path in human-readable format using the locations map.

```
void printPath(map<int, string>& locations, vector<vector<int>>& next, int u, int v) {
    if (next[u][v] == -1) {
        cout << "No path exists!";
        return;
    }
    cout << "Path: " << locations[u];
    while (u != v) {
        u = next[u][v];
        cout << " -> " << locations[u];
    }
}
```

3. int main()

This function Orchestrates the entire program, Handles user input, constructs the graph, calls Floyd-Warshall, and processes queries, Takes input for locations and paths, Builds adjacency matrices dist[][] and next[][],Accepts and processes user queries in a loop until "exit" is typed, Calls floydWarshall(...) and printPath(...).

```
if (idxMap.find(from) == idxMap.end()) {….
        }
        if (idxMap.find(to) == idxMap.end()) {…
        }
```

This code checks if from and to locations exist in idxMap. If a location is missing, it prints an error message, adjusts i to retry, and continues to the next iteration.

```
if (d < dist[u][v]) {
            dist[u][v] = d;
            next[u][v] = v;
        }
```

Keep shortest distance if multiple edges exist

Conclusion

The Campus Navigator project successfully demonstrates the practical application of the Floyd-Warshall algorithm in solving real-world pathfinding problems. By precomputing the shortest paths between all pairs of campus locations, the system offers instant and efficient navigation assistance, enabling users to quickly find the optimal route between any two points.

The use of adjacency matrices and path reconstruction via a next matrix ensures both accurate distance calculation and clear path display, enhancing the overall user experience. The project also incorporates input validation and dynamic path updates, making it robust and adaptable for larger or changing campus maps.

In conclusion, this project not only showcases the strength of the Floyd-Warshall algorithm in graph-based problems but also underlines the importance of efficient algorithms in developing user-friendly tools for navigation, planning, and decision-making.

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS E:\VS Code> cd "e:\VS Code\algoritham\" ; if ($
Welcome to Campus Navigator!

Enter number of locations: 4
Enter name for location 0: Library
Enter name for location 1: Cafeteria
Enter name for location 2: Ground
Enter name for location 3: Hall

Enter number of paths: 5
Enter paths (from to distance):
Path 1: Library Cafeteria 4
Path 2: Library Ground 2
Path 3: Ground Cafeteria 1
Path 4: Cafeteria Hall 3
Path 5: Ground Hall 5
```

```
Navigation Ready! Type 'exit' to quit

Enter starting point: Library
Enter destination: Hall

Shortest path from Library to Hall
Distance: 6 units
Path: Library -> Ground -> Cafeteria -> Hall
```