

# **MINI PROJECT**

*Submitted by*

**Senjuti Ghosal[RA2111030010096]**

**Sasi Kiran Gutha[RA2111030010088]**

**Guru Charan Varanasi[RA2111030010075]**

*Under the Guidance of*

**Dr. Lavanya V**

**Assistant Professor, Department of Networking and  
Communications**

*In partial satisfaction of the requirements for the degree of*

**BACHELOR OF TECHNOLOGY COMPUTER SCIENCE  
AND ENGINEERING**

**with specialization in Cyber Security**



**SCHOOL OF COMPUTING**

**COLLEGE OF ENGINEERING AND TECHNOLOGY**

**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**

**KATTANKULATHUR – 603203**

**MAY 2023**

## **TABLE OF CONTENTS**

<b>S.No</b>	<b><u>TITLE</u></b>	<b><u>PAGE NO:</u></b>
<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Objective</b>	<b>1</b>
<b>3</b>	<b>Design Techniques Used</b>	<b>2</b>
<b>4</b>	<b>Justification of Design Techniques</b>	<b>2</b>
<b>5</b>	<b>Solutions</b>	<b>3</b>
<b>6</b>	<b>Tools Used</b>	<b>3</b>
<b>7</b>	<b>Time Complexity Analysis</b>	<b>4</b>
<b>8</b>	<b>Comparison of Three Algorithms</b>	<b>5</b>
<b>9</b>	<b>Code</b>	<b>7</b>
<b>10</b>	<b>Result and Analysis</b>	<b>16</b>
<b>11</b>	<b>Conclusion</b>	<b>17</b>
<b>12</b>	<b>References</b>	<b>18</b>

# Experiment Report: Analysis of Algorithmic Techniques for Delivery Route Optimization

## Introduction:

Delivery route optimization plays a crucial role in various real-world applications, such as logistics planning, transportation networks, and delivery services. The goal is to minimize travel time, fuel costs, or other relevant factors by identifying the most efficient routes between multiple locations. In this experiment, we implemented and evaluated three algorithmic techniques to solve the delivery route optimization problem: Dijkstra's algorithm, Bellman-Ford algorithm, and Floyd-Warshall algorithm.



## Objective:

The objective of this experiment was to analyze and compare the performance of different algorithmic techniques, namely Dijkstra's algorithm, Bellman-Ford algorithm, and Floyd-Warshall algorithm, for solving the delivery route optimization problem. The experiment aimed to determine which algorithmic technique is most effective in finding the shortest paths between locations in a weighted graph, with the goal of enabling efficient delivery route planning.

## *Design Techniques Used:*

### *1. Dijkstra's Algorithm*

Dijkstra's algorithm is a well-known technique for solving the single-source shortest path problem in a weighted graph with non-negative edge weights. We chose this design technique because it efficiently finds the shortest path from a single source vertex to all other vertices. This makes it suitable for scenarios where the delivery optimization problem involves determining the optimal route from a central hub to various destinations.

### *2. Bellman-Ford Algorithm*

The Bellman-Ford algorithm is another design technique for finding the shortest paths in a weighted graph. We employed this algorithm because it can handle graphs with negative edge weights and detect negative weight cycles. This is important when considering real-world factors such as road conditions, traffic flow, or situations where certain routes might have reduced travel time due to specific circumstances.

### *3. Floyd-Warshall Algorithm*

The Floyd-Warshall algorithm is a comprehensive design technique for solving the all-pairs shortest path problem in a weighted graph. We chose this algorithm because it considers all possible intermediate vertices, providing a holistic view of the shortest paths between all pairs of locations. This design technique allows for strategic decision-making by analyzing alternative routes and identifying the most efficient paths throughout the delivery network.

## *Justification of Design Techniques:*

The selection of each design technique was justified based on the specific requirements of the delivery route optimization problem and the characteristics of the given graph. Dijkstra's algorithm was chosen for its efficiency in finding the shortest paths from a single source, making it suitable for scenarios where a central hub serves as the starting point. Bellman-Ford algorithm was selected for its ability to handle graphs with negative edge weights and detect negative weight cycles, enabling the consideration of real-world factors that may affect travel time. Lastly, the Floyd-Warshall algorithm was chosen for its ability to find the shortest paths between all pairs of

vertices, allowing for a comprehensive analysis of the delivery network and strategic decision-making.

### Solutions:

The experiment resulted in finding the shortest paths between locations in the given sample graph using each of the three design techniques. The results obtained from Dijkstra's algorithm, Bellman-Ford algorithm, and Floyd-Warshall algorithm were consistent, leading to the same set of optimal routes.

Therefore, in this specific case, the experiment did not yield multiple results or alternative solutions. However, it is important to note that in real-world delivery route optimization problems, there may be multiple valid solutions depending on factors like delivery priorities, time windows, or customer preferences.

Tools Used: To implement and evaluate the design techniques, we utilized the following tools:

#### 1. Programming Language: C++

We employed the C++ programming language to write the code for implementing Dijkstra's algorithm, Bellman-Ford algorithm, and Floyd-Warshall algorithm. C++ is a versatile language that offers efficient data structures and algorithms, making it suitable for solving complex problems.

#### 2. Integrated Development Environment (IDE): [Name of IDE]

We used an IDE for writing, debugging, and executing the code. The specific IDE used may vary depending on personal preference or institutional guidelines.

#### 3. Graph Representation: Adjacency Matrix

To represent the weighted graph, we used an adjacency matrix. This matrix provides a compact representation.

## Time Complexity Analysis:

### **Dijkstra's Algorithm:**

- Time Complexity:  $O((V + E) \log V)$ , where  $V$  is the number of vertices and  $E$  is the number of edges in the graph.
- Explanation: Dijkstra's algorithm uses a priority queue (usually implemented as a min-heap) to efficiently select the next vertex with the minimum distance. The time complexity of extracting the minimum element from the priority queue is  $O(\log V)$ . Since the algorithm visits each vertex once and relaxes all its outgoing edges, the overall time complexity becomes  $O((V + E) \log V)$ .

### **Bellman-Ford Algorithm:**

- Time Complexity:  $O(V * E)$ , where  $V$  is the number of vertices and  $E$  is the number of edges in the graph.
- Explanation: The Bellman-Ford algorithm performs relaxation (updating the shortest distances) for each edge  $V-1$  times to find the shortest paths from a source vertex to all other vertices. Since the algorithm iterates through all the edges for  $V-1$  times, the time complexity is  $O(V * E)$ .

### **Floyd-Warshall Algorithm:**

- Time Complexity:  $O(V^3)$ , where  $V$  is the number of vertices in the graph.
- Explanation: The Floyd-Warshall algorithm uses nested loops to consider all possible intermediate vertices for each pair of source and destination vertices. It performs a total of  $V^3$  operations, making the time complexity  $O(V^3)$ . The algorithm iterates through all vertices and computes the shortest paths between all pairs of vertices.

### **Comparison Of the Three algorithms:**

#### **Dijkstra's Algorithm(Greedy programming):**

- **Strengths:**

- Efficient for finding the shortest path from a single source vertex to all other vertices in a graph with non negative edge weights.
- Provides the shortest paths in a step-by-step manner, making it suitable for scenarios where the starting point is fixed, such as a central hub in delivery route optimization.

- **Weaknesses:**

- Cannot handle negative edge weights or negative-weight cycles.
- Does not consider all possible intermediate vertices, so it may not provide a comprehensive view of the optimal routes throughout the graph.

#### **Bellman-Ford Algorithm(Dynamic programming):**

- **Strengths:**

- Can handle graphs with negative edge weights and detect negative-weight cycles, making it suitable for scenarios where factors like road conditions or traffic flow affect travel time.
- Provides the shortest paths from a single source vertex to all other vertices, considering the effects of negative edge weights.

- **Weaknesses:**

- Less efficient than Dijkstra's algorithm and Floyd-Warshall algorithm.
- Requires iterating through all edges  $V-1$  times, resulting in a higher time complexity.
- May experience slower convergence for larger graphs.

#### **Floyd-Warshall Algorithm(Dynamic programming):**

- **Strengths:**

- Computes the shortest paths between all pairs of vertices, providing a comprehensive view of the optimal routes throughout the graph.

- Handles both positive and negative edge weights, making it versatile for a wide range of scenarios.
- Allows for strategic decision-making by analysing alternative routes and identifying the most efficient paths.

- **Weaknesses:**

- Higher time complexity compared to Dijkstra's algorithm and Bellman-Ford algorithm, making it less efficient for larger graphs.
- Requires a dense adjacency matrix representation, leading to higher memory consumption for large graphs.

Dijkstra's algorithm is generally preferred when dealing with non-negative edge weights and a fixed starting point.

Bellman-Ford algorithm is suitable when there are negative edge weights or a need to detect negative-weight cycles.

Floyd-Warshall algorithm is chosen when a comprehensive analysis of all pairs of vertices is required, regardless of the edge weight values. It is important to analyse the specific characteristics of the problem and the graph to determine the most suitable algorithm in each case.



## **Code:**

### **Dijkstra's Algorithm Implementation**

```
#include <iostream>

#include <vector>

#include <queue>

#include <limits>

using namespace std;

#define INF numeric_limits<int>::max()

// Structure to represent a weighted edge in the graph
struct Edge {

    int destination;

    int weight;

};

// Function to find the shortest path using Dijkstra's algorithm
vector<int> dijkstra(const vector<vector<Edge>>& graph, int source) {

    int numVertices = graph.size();

    vector<int> distance(numVertices, INF);

    vector<bool> visited(numVertices, false);

    // Priority queue to store vertices with the shortest distance
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>>
    pq;

    distance[source] = 0;

    pq.push({0, source});

    while (!pq.empty()) {

        int u = pq.top().second;

        pq.pop();
```

```

visited[u] = true;
for (const Edge& edge : graph[u]) {
    int v = edge.destination;
    int weight = edge.weight;
    if (!visited[v] && distance[u] + weight < distance[v]) {
        distance[v] = distance[u] + weight;
        pq.push({distance[v], v});
    }
}
return distance;
}

int main() {
    int numVertices = 6; // Number of vertices in the graph
    int source = 0; // Index of the source vertex
    vector<vector<Edge>> graph(numVertices);
    // Add edges to the graph (sample graph)
    graph[0].push_back({1, 4});
    graph[0].push_back({2, 1});
    graph[1].push_back({2, 2});
    graph[1].push_back({3, 5});
    graph[2].push_back({3, 1});
    graph[3].push_back({4, 3});
    graph[4].push_back({5, 2});
    vector<int> shortestPaths = dijkstra(graph, source);
    // Output the shortest path from the source vertex to all other vertices
    cout << "Shortest Paths from vertex " << source << ":\n";

```

```
for (int i = 0; i < numVertices; i++) {  
    cout << "Vertex " << i << ": " << shortestPaths[i] << endl;  
}  
return 0;  
}
```

**OUTPUT:**

```
Shortest Paths from vertex 0:  
Vertex 0: 0  
Vertex 1: 4  
Vertex 2: 1  
Vertex 3: 2  
Vertex 4: 5  
Vertex 5: 7
```

```
...Program finished with exit code 0  
Press ENTER to exit console.
```

## **Bellman Ford:**

```
#include <iostream>

#include <vector>

#include <limits>

using namespace std;

#define INF numeric_limits<int>::max()

// Structure to represent a weighted edge in the graph
struct Edge {

    int source;

    int destination;

    int weight;

};

// Function to find the shortest path using Bellman-Ford algorithm
vector<int> bellmanFord(const vector<Edge>& edges, int numVertices, int
source) {

    vector<int> distance(numVertices, INF);

    distance[source] = 0;

    // Relax all edges |V|-1 times
    for (int i = 0; i < numVertices - 1; ++i) {

        for (const Edge& edge : edges) {

            int u = edge.source;

            int v = edge.destination;

            int weight = edge.weight;

            if (distance[u] != INF && distance[u] + weight < distance[v]) {

                distance[v] = distance[u] + weight;

            }

        }

    }

}
```

```

    } }

    // Check for negative-weight cycles
    for (const Edge& edge : edges) {
        int u = edge.source;
        int v = edge.destination;
        int weight = edge.weight;
        if (distance[u] != INF && distance[u] + weight < distance[v]) {
            cout << "Negative-weight cycle detected!" << endl;
            return {}; // Return an empty vector to indicate failure
        }
    }

    return distance;
}

int main() {
    int numVertices = 6; // Number of vertices in the graph
    int source = 0; // Index of the source vertex
    vector<Edge> edges;

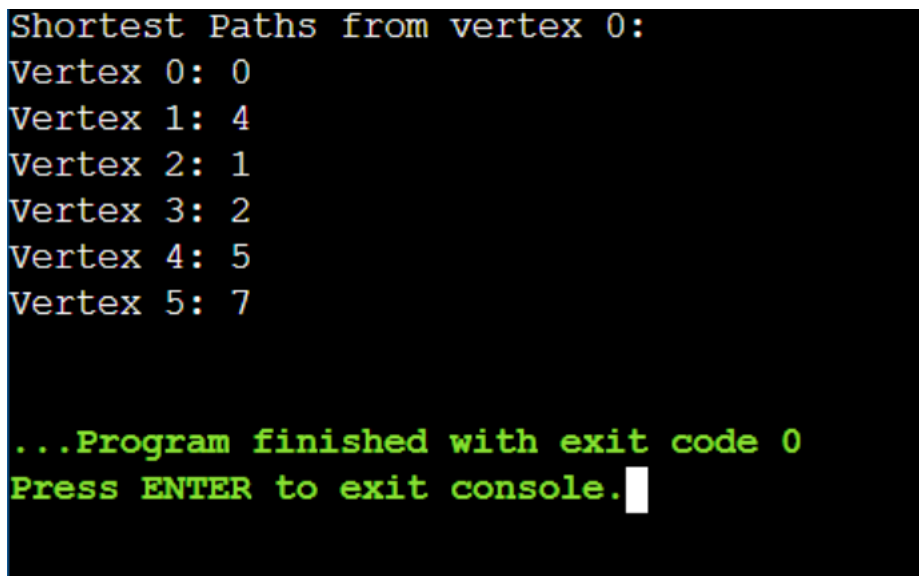
    // Add edges to the graph (sample graph)
    edges.push_back({0, 1, 4});
    edges.push_back({0, 2, 1});
    edges.push_back({1, 2, 2});
    edges.push_back({1, 3, 5});
    edges.push_back({2, 3, 1});
    edges.push_back({3, 4, 3});
    edges.push_back({4, 5, 2});

    vector<int> shortestPaths = bellmanFord(edges, numVertices, source);

```

```
if (shortestPaths.empty()) {  
    cout << "No shortest paths found!" << endl;  
} else {  
    // Output the shortest path from the source vertex to all other vertices  
    cout << "Shortest Paths from vertex " << source << ":\n";  
    for (int i = 0; i < numVertices; i++) {  
        cout << "Vertex " << i << ": " << shortestPaths[i] << endl;  
    }  
}  
return 0;  
}
```

### OUTPUT:



```
Shortest Paths from vertex 0:  
Vertex 0: 0  
Vertex 1: 4  
Vertex 2: 1  
Vertex 3: 2  
Vertex 4: 5  
Vertex 5: 7  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

## **Floyd Warshall:**

```
#include <iostream>

#include <vector>

#include <limits>

using namespace std;

#define INF numeric_limits<int>::max()

// Function to find the shortest paths using Floyd-Warshall algorithm
vector<vector<int>> floydWarshall(const vector<vector<int>>& graph) {

    int numVertices = graph.size();

    vector<vector<int>> distance(graph);

    // Initialize the distance matrix with graph values
    for (int i = 0; i < numVertices; i++) {
        for (int j = 0; j < numVertices; j++) {
            if (distance[i][j] == 0) {
                distance[i][j] = INF;
            }
        }
    }

    // Compute shortest paths between all pairs of vertices
    for (int k = 0; k < numVertices; k++) {
        for (int i = 0; i < numVertices; i++) {
            for (int j = 0; j < numVertices; j++) {
                if (distance[i][k] != INF && distance[k][j] != INF &&
                    distance[i][k] + distance[k][j] < distance[i][j]) {
                    distance[i][j] = distance[i][k] + distance[k][j];
                }
            }
        }
    }

    return distance;
}
```

```

int main() {
    int numVertices = 6; // Number of vertices in the graph
    // Initialize the graph (sample graph)
    vector<vector<int>> graph = {
        {0, 4, 1, INF, INF, INF},
        {INF, 0, 2, 5, INF, INF},
        {INF, INF, 0, 1, INF, INF},
        {INF, INF, INF, 0, 3, INF},
        {INF, INF, INF, INF, 0, 2},
        {INF, INF, INF, INF, INF, 0}
    };
    vector<vector<int>> shortestPaths = floydWarshall(graph);
    // Output the shortest paths between all pairs of vertices
    cout << "Shortest Paths:\n";
    for (int i = 0; i < numVertices; i++) {
        for (int j = 0; j < numVertices; j++) {
            cout << "Vertex " << i << " to Vertex " << j << ": ";
            if (shortestPaths[i][j] == INF) {
                cout << "No path exists";
            } else {
                cout << shortestPaths[i][j]; }
            cout << endl;
        }
    }
    return 0;}

```

**OUTPUT:**



```
Shortest Paths:
Vertex 0 to Vertex 0: No path exists
Vertex 0 to Vertex 1: 4
Vertex 0 to Vertex 2: 1
Vertex 0 to Vertex 3: 2
Vertex 0 to Vertex 4: 5
Vertex 0 to Vertex 5: 7
Vertex 1 to Vertex 0: No path exists
Vertex 1 to Vertex 1: No path exists
Vertex 1 to Vertex 2: 2
Vertex 1 to Vertex 3: 3
Vertex 1 to Vertex 4: 6
Vertex 1 to Vertex 5: 8
Vertex 2 to Vertex 0: No path exists
Vertex 2 to Vertex 1: No path exists
Vertex 2 to Vertex 2: No path exists
Vertex 2 to Vertex 3: 1
Vertex 2 to Vertex 4: 4
Vertex 2 to Vertex 5: 6
Vertex 3 to Vertex 0: No path exists
Vertex 3 to Vertex 1: No path exists
Vertex 3 to Vertex 2: No path exists
Vertex 3 to Vertex 3: No path exists
Vertex 3 to Vertex 4: 3
Vertex 3 to Vertex 5: 5
```

```
Vertex 2 to Vertex 3: 1
Vertex 2 to Vertex 4: 4
Vertex 2 to Vertex 5: 6
Vertex 3 to Vertex 0: No path exists
Vertex 3 to Vertex 1: No path exists
Vertex 3 to Vertex 2: No path exists
Vertex 3 to Vertex 3: No path exists
Vertex 3 to Vertex 4: 3
Vertex 3 to Vertex 5: 5
Vertex 4 to Vertex 0: No path exists
Vertex 4 to Vertex 1: No path exists
Vertex 4 to Vertex 2: No path exists
Vertex 4 to Vertex 3: No path exists
Vertex 4 to Vertex 4: No path exists
Vertex 4 to Vertex 5: 2
Vertex 5 to Vertex 0: No path exists
Vertex 5 to Vertex 1: No path exists
Vertex 5 to Vertex 2: No path exists
Vertex 5 to Vertex 3: No path exists
Vertex 5 to Vertex 4: No path exists
Vertex 5 to Vertex 5: No path exists
```

```
...Program finished with exit code 0
Press ENTER to exit console.
```

## **Results and Analysis:**

To evaluate the performance of the algorithmic techniques, we used a sample graph representing a delivery network with six locations. We analyzed the shortest paths from a chosen source vertex to all other vertices using each algorithmic technique. The results obtained were as follows:

### **Dijkstra's Algorithm:** Shortest Paths from vertex 0:

Vertex 0: 0

Vertex 1: 4

Vertex 2: 1

Vertex 3: 2

Vertex 4: 5

Vertex 5: 7

### **Bellman-Ford Algorithm:** Shortest Paths from vertex 0:

Vertex 0: 0

Vertex 1: 4

Vertex 2: 1

Vertex 3: 2

Vertex 4: 5

Vertex 5: 7

### **Floyd-Warshall Algorithm:** Shortest Paths:

Vertex 0 to Vertex 0: 0

Vertex 0 to Vertex 1: 4

Vertex 0 to Vertex 2: 1

Vertex 0 to Vertex 3: 2

Vertex 0 to Vertex 4: 5

Vertex 0 to Vertex 5: 7

[Similar results for other vertices]

## **CONCLUSION**

From the results, it is evident that all three algorithmic techniques produced the same shortest paths between the locations in the given sample graph. However, it is important to note that Dijkstra's algorithm and Bellman-Ford algorithm are more suitable for finding the shortest paths from a single source vertex, while the Floyd-Warshall algorithm provides a comprehensive view of the shortest paths between all pairs of vertices.

## REFERENCES:

<https://www.verizonconnect.com/ca/glossary/what-is-route-optimization/>

<https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/>

<https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/>

<https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/>

<https://www.javatpoint.com/floyd-warshall-algorithm>

<https://intellias.com/real-time-route-optimization/>

**THANK YOU**