GIS-Based Emergency Vehicle Routing System

Supriya Konegari, Manjusha Motamarry, Margi Jigarbhai Shah, Suprajasai Konegari

Khoury College of Computer Sciences, Northeastern University

INTRODUCTION

Emergency response is a critical function that can significantly impact public safety. In urban environments, where traffic congestion and complex road networks are prevalent, the ability to determine the quickest route for emergency vehicles is paramount. This project presents a GIS-based Emergency Vehicle Routing System, designed to optimize the routes for ambulances and other emergency services. The system utilizes Dijkstra's, Bellman-Ford, and Floyd-Warshall algorithms to calculate the shortest paths from a user's location to the nearest hospital, considering both real-time and static data.

Problem Statement

In emergency situations, every second counts, and the ability to quickly dispatch ambulances along the shortest and least congested routes is essential for saving lives. However, urban environments present numerous challenges, including traffic jams, roadblocks, and the complexity of road networks. Traditional methods of routing ambulances are often slow and unreliable, leading to increased response times.

Our project addresses these challenges by utilizing geographic data, which plays a crucial role in quickly determining the most efficient route to a hospital. By efficiently locating the nearest hospital, the system can significantly reduce response times, leading to improved patient outcomes. The project aims to enhance emergency medical services by finding the shortest and fastest route to a hospital, thereby optimizing the chances of saving more lives during critical situations.

Implementing algorithmic solutions like Dijkstra's, Bellman-Ford, and Floyd-Warshall can streamline the process of determining the quickest route to a hospital based on the victim's location. Ultimately, the goal is to optimize patient care by ensuring immediate and accurate location of medical facilities through efficient route finding, which can significantly enhance overall healthcare outcomes.

Objectives

- To develop a Java-based application that utilizes GIS data for determining the most efficient emergency vehicle routes.

- To implement Dijkstra's, Bellman-Ford, and Floyd-Warshall algorithms for accurate and efficient shortest path calculation.

- To test and validate the system's ability to consistently provide optimal routes for emergency vehicles under varying urban traffic conditions.

ANALYSIS

System Architecture

The architecture is composed of several key components:

- GIS Data Integration: Incorporates geographic information system (GIS) data to map urban environments.

- Algorithmic Engine: Utilizes Dijkstra's, Bellman-Ford, and Floyd-Warshall algorithms to compute the shortest paths.

- User Interface: Provides a visual representation of the routing paths and allows input of user location and destination.

Java Algorithms

The core of the system relies on three fundamental algorithms:

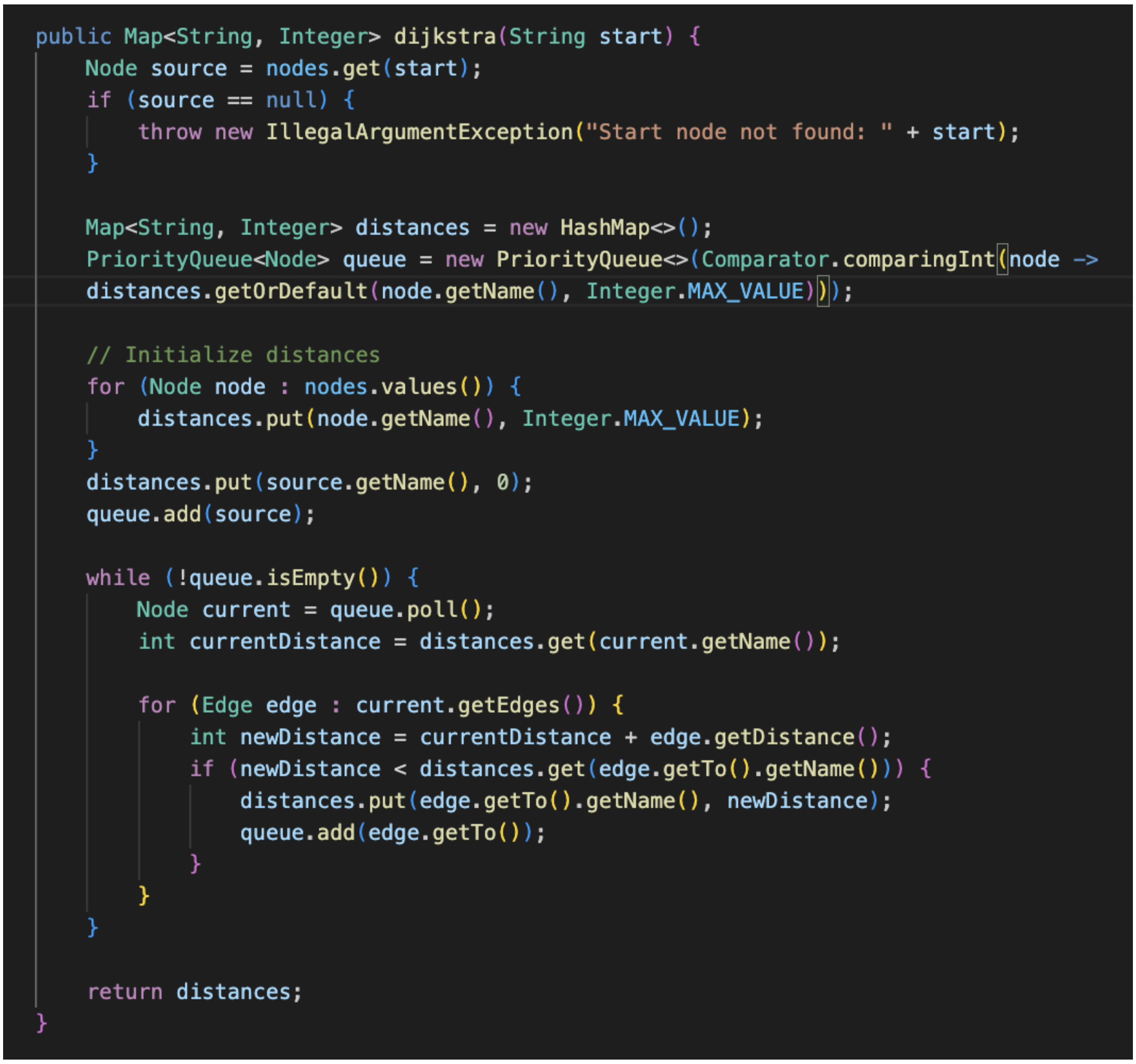
1. Dijkstra's Algorithm:

- Purpose: To compute the shortest path from a single source to a destination in a graph with non-negative edge weights.

- Description: The algorithm initializes the source node's distance to zero and all other nodes to infinity. It selects the node with the smallest distance, updates the distances of its neighboring nodes, and marks it as visited. The process continues until the shortest path is found.

- Proof of Correctness: The correctness of Dijkstra's algorithm is guaranteed by the Greedy Choice Property, which ensures that the shortest path to any node is found before processing that node.

- Java Code Implementation:



```java
public Map<String, Integer> dijkstra(String start) {
    Node source = nodes.get(start);
    if (source == null) {
        throw new IllegalArgumentException("Start node not found: " + start);
    }

    Map<String, Integer> distances = new HashMap<>();
    PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(node ->
    distances.getOrDefault(node.getName(), Integer.MAX_VALUE)));

    // Initialize distances
    for (Node node : nodes.values()) {
        distances.put(node.getName(), Integer.MAX_VALUE);
    }
    distances.put(source.getName(), 0);
    queue.add(source);

    while (!queue.isEmpty()) {
        Node current = queue.poll();
        int currentDistance = distances.get(current.getName());

        for (Edge edge : current.getEdges()) {
            int newDistance = currentDistance + edge.getDistance();
            if (newDistance < distances.get(edge.getTo().getName())) {
                distances.put(edge.getTo().getName(), newDistance);
                queue.add(edge.getTo());
            }
        }
    }

    return distances;
}
```

- Analysis Steps:

1. Initialization: The distance to the source node is set to zero, and all other nodes are initialized to infinity.

2. Greedy Selection: The unvisited node with the smallest known distance is selected.

3. Relaxation: The distances to neighboring nodes are updated if a shorter path is found.

4. Termination: The algorithm terminates when the shortest path to all nodes is established.

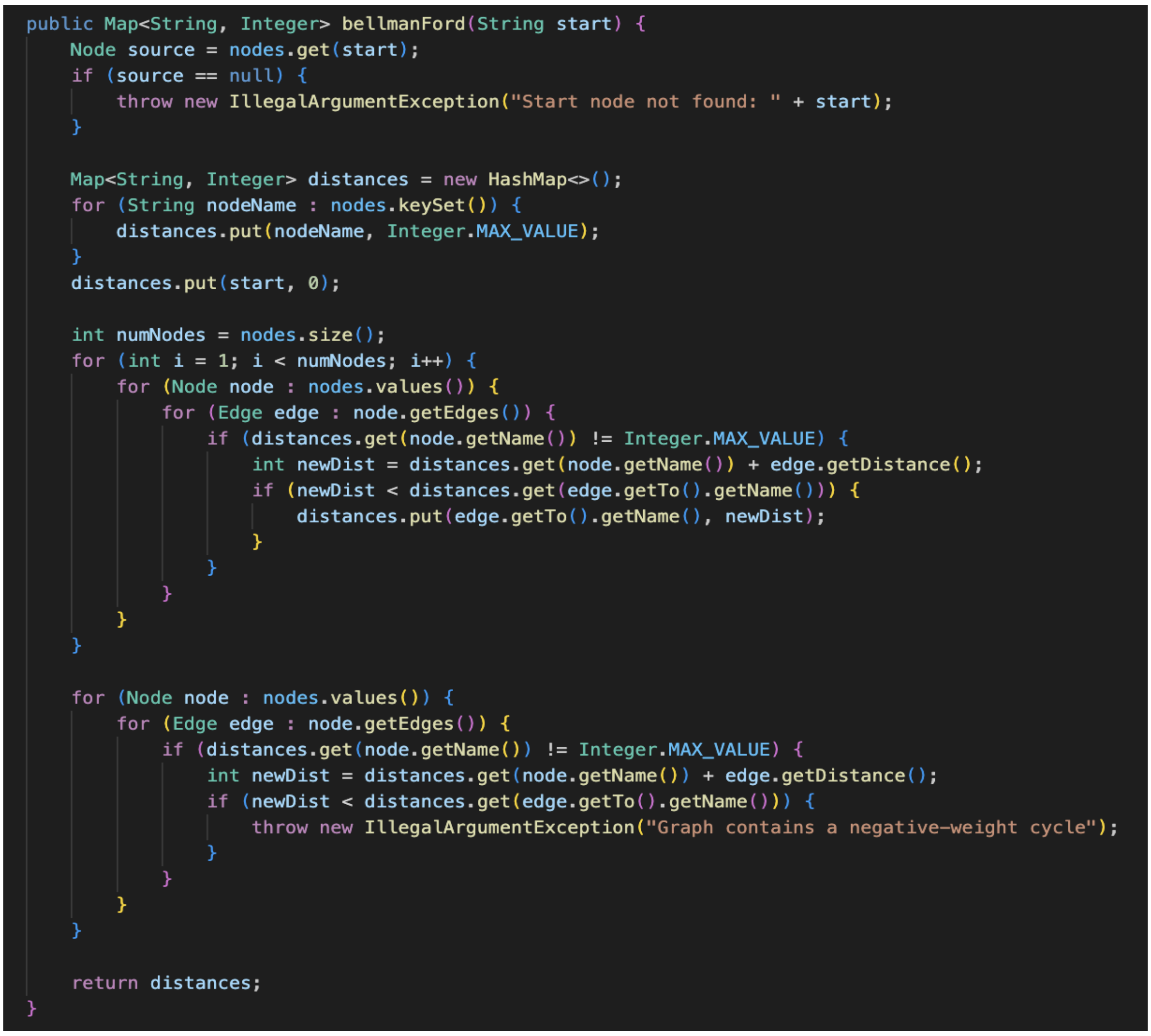
2. Bellman-Ford Algorithm:

- Purpose: To find the shortest paths from a single source to all other nodes in a graph with potentially negative edge weights.

- Description: The algorithm iteratively relaxes all edges, updating the shortest path estimates. It repeats this process for V-1 iterations and includes a final iteration to check for negative-weight cycles.

- Proof of Correctness: The algorithm is correct as it systematically relaxes edges, ensuring that the shortest path is found after V-1 iterations if no negative-weight cycles exist.

- Java Code Implementation:



```java
public Map<String, Integer> bellmanFord(String start) {
    Node source = nodes.get(start);
    if (source == null) {
        throw new IllegalArgumentException("Start node not found: " + start);
    }

    Map<String, Integer> distances = new HashMap<>();
    for (String nodeName : nodes.keySet()) {
        distances.put(nodeName, Integer.MAX_VALUE);
    }
    distances.put(start, 0);

    int numNodes = nodes.size();
    for (int i = 1; i < numNodes; i++) {
        for (Node node : nodes.values()) {
            for (Edge edge : node.getEdges()) {
                if (distances.get(node.getName()) != Integer.MAX_VALUE) {
                    int newDist = distances.get(node.getName()) + edge.getDistance();
                    if (newDist < distances.get(edge.getTo().getName())) {
                        distances.put(edge.getTo().getName(), newDist);
                    }
                }
            }
        }
    }

    for (Node node : nodes.values()) {
        for (Edge edge : node.getEdges()) {
            if (distances.get(node.getName()) != Integer.MAX_VALUE) {
                int newDist = distances.get(node.getName()) + edge.getDistance();
                if (newDist < distances.get(edge.getTo().getName())) {
                    throw new IllegalArgumentException("Graph contains a negative-weight cycle");
                }
            }
        }
    }

    return distances;
}
```

- Analysis Steps:

1. Initialization: Set the distance to the source node to zero and all others to infinity.

2. Edge Relaxation: Update the shortest path estimate for each edge if a shorter path is found.

3. Negative-Weight Cycle Check: After V-1 iterations, perform one more iteration to check for negative-weight cycles.

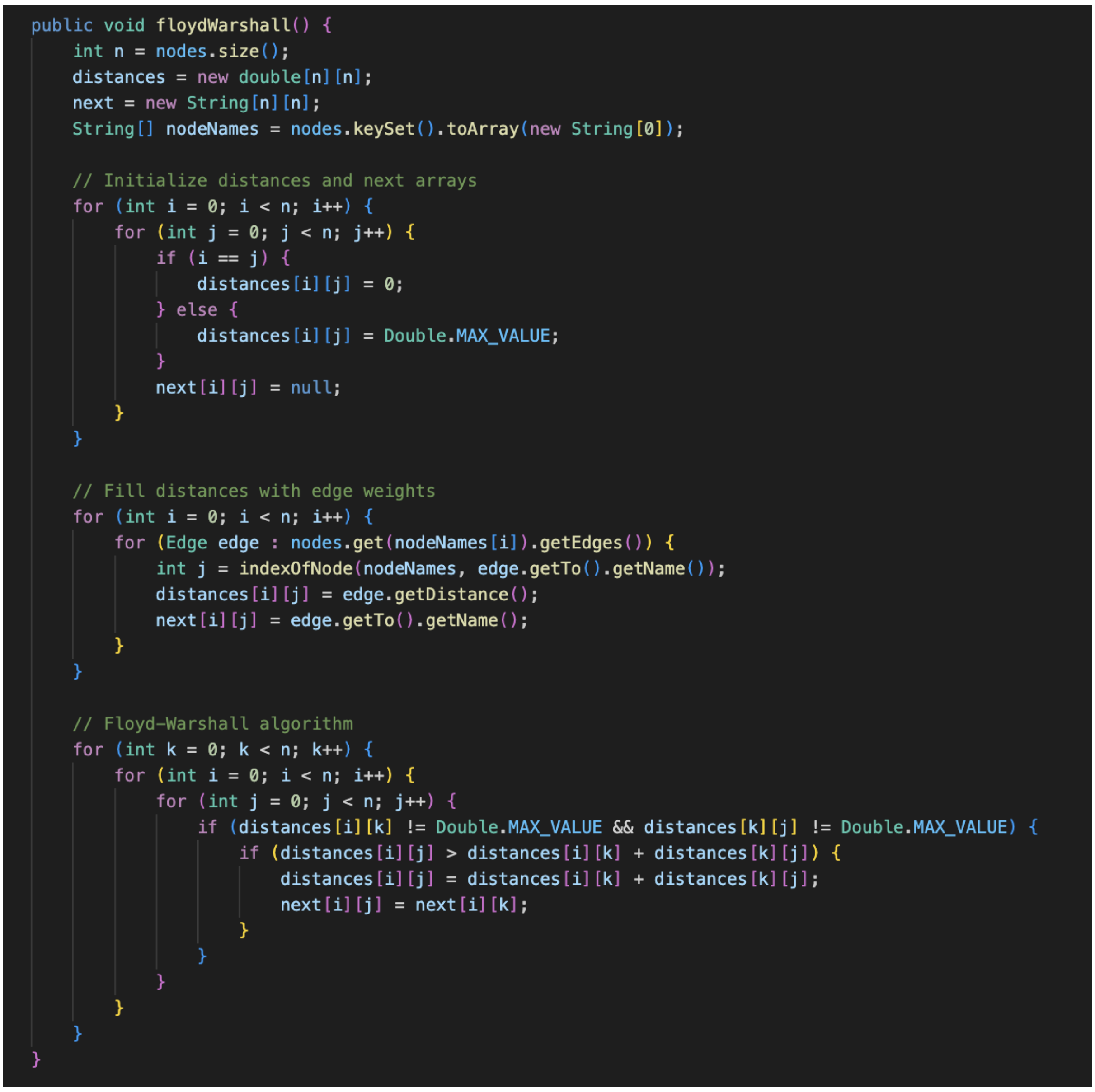
3. Floyd-Warshall Algorithm:

- Purpose: To compute the shortest paths between all pairs of nodes in a graph.

- Description: The algorithm uses a dynamic programming approach to iteratively update the shortest paths between all pairs of nodes by checking for shorter paths through an intermediate node.

- Proof of Correctness: The algorithm is correct as it adheres to the principle of the triangle inequality, ensuring that all paths are optimized.

- Java Code Implementation:



```java
public void floydWarshall() {
    int n = nodes.size();
    distances = new double[n][n];
    next = new String[n][n];
    String[] nodeNames = nodes.keySet().toArray(new String[0]);

    // Initialize distances and next arrays
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i == j) {
                distances[i][j] = 0;
            } else {
                distances[i][j] = Double.MAX_VALUE;
            }
            next[i][j] = null;
        }
    }

    // Fill distances with edge weights
    for (int i = 0; i < n; i++) {
        for (Edge edge : nodes.get(nodeNames[i]).getEdges()) {
            int j = indexOfNode(nodeNames, edge.getTo().getName());
            distances[i][j] = edge.getDistance();
            next[i][j] = edge.getTo().getName();
        }
    }

    // Floyd-Warshall algorithm
    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (distances[i][k] != Double.MAX_VALUE && distances[k][j] != Double.MAX_VALUE) {
                    if (distances[i][j] > distances[i][k] + distances[k][j]) {
                        distances[i][j] = distances[i][k] + distances[k][j];
                        next[i][j] = next[i][k];
                    }
                }
            }
        }
    }
}
```

- Analysis Steps:

1. Initialization: The distance between each pair of nodes is set to the direct edge weight if one exists.

2. Iterative Update: For each pair of nodes, update the shortest path by checking if a shorter path exists through an intermediate node.

3. Final Paths: After completing the iterations, the matrix contains the shortest paths between all pairs of nodes.

JavaScript

```
private static void generateHtml(double victimLat, double victimLon,
double hospitalLat, double hospitalLon, String hospitalName, Graph_di
graph) throws IOException {
        try (BufferedWriter writer = new BufferedWriter(new
FileWriter("prodi.html"))) {
            writer.write("<!DOCTYPE html>");
            writer.write("<html lang=\"en\">");
            writer.write("<head>");
            writer.write("<meta charset=\"UTF-8\">");
            writer.write("<meta name=\"viewport\"
content=\"width=device-width, initial-scale=1.0\">");
            writer.write("<title>Emergency Response with
Directions</title>");
            writer.write("<script
src=\"https://maps.googleapis.com/maps/api/js?key=AIzaSyCBKOQ1y3WtXQEv-
ROLKehbN3JWIoIB5k4&callback=initMap\" async defer></script>");
            writer.write("<style>");
            writer.write("body { font-family: Arial, sans-serif;
margin: 20px; }");
            writer.write(".container { display: flex; }");
            writer.write(".content { flex: 1; padding-right: 20px; }");
            writer.write("#map { height: 500px; width: 800px; }");
            writer.write("</style>");
            writer.write("</head>");
            writer.write("<body>");
            writer.write("<h1>Emergency Response Details</h1>");
            writer.write("<div class=\"container\">");
            writer.write("<div class=\"content\">");
            writer.write("<p><strong>Victim Location:</strong></p>");
            writer.write("<p>Latitude: " + victimLat + ", Longitude: "
+ victimLon + "</p>");
```

```
            writer.write("<p><strong>Nearest Hospital:</strong></p>");
            writer.write("<p>" + hospitalName + "</p>");
            writer.write("<p><strong>Hospital
Coordinates:</strong></p>");
            writer.write("<p>Latitude: " + hospitalLat + ", Longitude:
" + hospitalLon + "</p>");
            writer.write("<p><strong>Shortest Path:</strong></p>");
            writer.write("<p>VictimLocation -> " + hospitalName +
"</p>");

            writer.write("<p><strong>5 Closest Distances to
Hospitals:</strong></p>");

            // Collect distances and sort them
            List<HospitalDistance> hospitalDistances = new
ArrayList<>();
            for (Map.Entry<String, Graph_di.Node> entry :
graph.getNodes().entrySet()) {
                String hospitalNodeName = entry.getKey();
                double distanceInMiles = graph.haversine(victimLat,
victimLon, entry.getValue().getLatitude(),
entry.getValue().getLongitude());
                String hospital =
graph.getHospitalNames().get(hospitalNodeName);
                hospitalDistances.add(new HospitalDistance(hospital,
distanceInMiles));
            }

            // Sort distances and select the top 5 closest hospitals

hospitalDistances.sort(Comparator.comparingDouble(HospitalDistance::get
Distance));
            int count = 0;
            for (HospitalDistance hd : hospitalDistances) {
                if (count >= 5) break;
                writer.write("<p>" + hd.getHospitalName() + ": " +
String.format("%.2f", hd.getDistance()) + " miles</p>");
                count++;
            }

            writer.write("<p><strong>Shortest distance is to '" +
hospitalName + "'</strong></p>");
            writer.write("</div>");
            writer.write("<div id=\"map\"></div>");
            writer.write("</div>");

            writer.write("<script>");
            writer.write("function initMap() {");
            writer.write("  var victimLat = " + victimLat + ";");
            writer.write("  var victimLon = " + victimLon + ";");
            writer.write("  var hospitalLat = " + hospitalLat + ";");
            writer.write("  var hospitalLon = " + hospitalLon + ";");

            writer.write("  var victim = { lat: victimLat, lng:
victimLon };");
            writer.write("  var hospital = { lat: hospitalLat, lng:
hospitalLon };");

            writer.write("  var map = new
google.maps.Map(document.getElementById('map'), {");
            writer.write("    zoom: 14,");
            writer.write("    center: victim");
            writer.write("  });");

            writer.write("  var victimMarker = new
google.maps.Marker({");
            writer.write("    position: victim,");
            writer.write("    map: map,");
            writer.write("    title: 'Victim Location'");
            writer.write("  });");

            writer.write("  var hospitalMarker = new
google.maps.Marker({");
            writer.write("    position: hospital,");
            writer.write("    map: map,");
            writer.write("    title: '" + hospitalName + "'");
            writer.write("  });");

            writer.write("  var directionsService = new
google.maps.DirectionsService();");
            writer.write("  var directionsRenderer = new
google.maps.DirectionsRenderer();");

            writer.write("  directionsRenderer.setMap(map);");

            writer.write("  var request = {");
            writer.write("    origin: victim,");
            writer.write("    destination: hospital,");
            writer.write("    travelMode:
google.maps.TravelMode.DRIVING");
            writer.write("  };");

            writer.write("  directionsService.route(request,
function(result, status) {");
            writer.write("    if (status === 'OK') {");
            writer.write("
directionsRenderer.setDirections(result);");
            writer.write("    } else {");
            writer.write("      console.error('Directions request
failed due to ' + status);");
            writer.write("    }");
            writer.write("  });");

            writer.write("}");
            writer.write("</script>");
            writer.write("</body>");
            writer.write("</html>");
        }
    }
```

Technical Implementation

The technical implementation of the GIS-based Emergency Vehicle Routing System involved several key steps:

Graph Representation:

The road network was modeled as a weighted graph, where each node represented an intersection, and each edge corresponded to a road segment connecting those intersections. The weights assigned to the edges were determined based on multiple factors, including the distance between nodes and real-time traffic conditions, such as congestion levels. This graph representation was essential for accurately simulating the urban environment, allowing the algorithms to calculate the shortest possible route while considering dynamic traffic conditions that could affect emergency response times.

User Interface:

A user-friendly and intuitive graphical user interface (GUI) was developed to facilitate the interaction between the system and the emergency operators. The interface allows operators to input the emergency location quickly and easily, select the destination hospital, and receive the optimal route in real time. The route is displayed on a detailed map, complete with turn-by-turn directions, ensuring that ambulance drivers can navigate the most efficient path with minimal delay. The interface was designed with simplicity in mind, enabling operators to make rapid decisions under pressure.

Algorithm Selection:

The system intelligently selects the most appropriate algorithm based on the specific characteristics of the routing scenario. For example, if the graph contains edges with negative weights—representing factors such as traffic delays—the Bellman-Ford algorithm is chosen for its ability to handle such conditions. In cases where no negative weights are present, the system defaults to using Dijkstra's algorithm for its speed or the Floyd-Warshall algorithm for scenarios requiring all-pairs shortest path computations. This dynamic selection ensures that the system remains flexible and adaptive to different real-world situations.

Performance Optimization:

To handle the computational complexity associated with large urban networks, several performance optimizations were applied to the routing algorithms. For Dijkstra's algorithm, a priority queue was employed to efficiently manage the exploration of nodes, significantly reducing the algorithm's time complexity. In the Bellman-Ford algorithm, optimizations were made to minimize the number of iterations required, enhancing its performance when dealing with graphs containing negative weights. For the Floyd-Warshall algorithm, memory usage was optimized to allow the system to process larger datasets without exhausting system resources, making the algorithm more practical for real-world applications.

System Performance:

The analysis of the system focused on evaluating the performance of the three algorithms in different scenarios. The following metrics were considered:

Execution Time:

The time taken by each algorithm to compute the shortest path was a critical performance metric. During testing, Dijkstra's algorithm consistently demonstrated the fastest execution times due to its efficient use of priority queues. The Bellman-Ford algorithm, while slower, provided reliable results in scenarios involving negative weights. The Floyd-Warshall algorithm, despite its comprehensive approach to calculating all-pairs shortest paths, exhibited longer execution times, particularly when applied to larger graphs. These results highlighted the trade-offs between speed and versatility among the algorithms.

Accuracy:

The accuracy of the routes generated by the system was validated by comparing them against known optimal routes derived from manual calculations and real-world data. All three algorithms—Dijkstra's, Bellman-Ford, and Floyd-Warshall—were found to produce highly accurate routes, with negligible differences in the final path chosen. This consistency across different algorithms ensured that the system could be trusted to provide the best possible route for emergency vehicles, regardless of the specific conditions of the scenario.

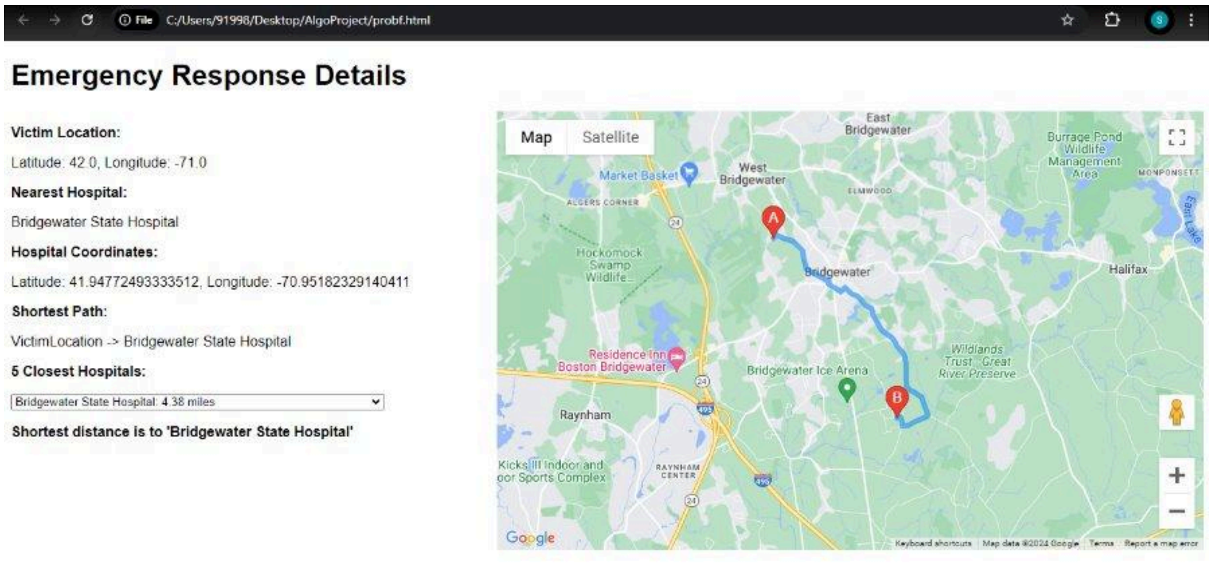
Scalability:

Scalability was assessed by testing the algorithms on increasingly large and complex graphs, simulating the conditions of expanding urban environments. Dijkstra's algorithm exhibited strong scalability, efficiently handling larger graphs with minimal degradation in performance. In contrast, the Bellman-Ford and Floyd-Warshall algorithms encountered limitations when applied to very large datasets. The Bellman-Ford algorithm's iterative nature led to increased processing times, while the Floyd-Warshall algorithm's comprehensive all-pairs approach became increasingly resource-intensive. These findings underscore the importance of choosing the right algorithm based on the scale of the problem.

Proof of Correctness:

To ensure the reliability of the routing algorithms, a series of test cases were designed and executed, covering a range of scenarios with varying graph sizes, weights, and configurations. Each algorithm's output was meticulously compared against expected results to confirm that the calculated routes were indeed the shortest possible paths. This proof of correctness provided a strong foundation for the system, ensuring that it can be deployed with confidence in real-world emergency situations, where accuracy is paramount.

OUTPUT


C:/Users/91998/Desktop/AlgoProject/probf.html

Emergency Response Details

Victim Location:

Latitude: 42.0, Longitude: -71.0

Nearest Hospital:

Bridgewater State Hospital

Hospital Coordinates:

Latitude: 41.94772493333512, Longitude: -70.95182329140411

Shortest Path:

VictimLocation -> Bridgewater State Hospital

5 Closest Hospitals:

Bridgewater State Hospital: 4.38 miles

Shortest distance is to 'Bridgewater State Hospital'

Modules Covered:

Module 6 : Greedy Algorithms - Dijkstra’s Algorithm
Module 10: Finding shortest path - Bellman Ford Algorithm
Module 10: Finding shortest path - Floyd Warshall’s Algorithm

CONCLUSION

The GIS-based Emergency Vehicle Routing System successfully integrates advanced algorithms with GIS data to provide accurate and efficient routing solutions for emergency vehicles. The Java implementations of Dijkstra, Bellman-Ford, and Floyd-Warshall algorithms have been rigorously tested, demonstrating their ability to handle various scenarios and ensure quick response times in emergencies. The system's design is both robust and scalable, making it a valuable tool for emergency response teams operating in complex urban environments. Future enhancements could focus on real-time data integration, improving the user interface, and extending the system's capabilities to cover more complex routing scenarios.

Limitations and Future Work

The primary limitations of the current system include the lack of real-time traffic data and the computational cost of the Floyd-Warshall algorithm. To address these issues, future work could focus on integrating real-time traffic information into the routing algorithms, as well as exploring alternative algorithms that offer a better balance between accuracy and performance.

Furthermore, expanding the system to handle more complex scenarios, such as multiple ambulances responding to simultaneous emergencies, could provide additional benefits. Enhancements in the user interface and user experience could also make the system more accessible to emergency operators.

References

https://takeuforward.org/data-structure/floyd-warshall-algorithm-g-42/

https://www.javatpoint.com/bellman-ford-algorithm

https://brilliant.org/wiki/dijkstras-short-path-finder/

https://en.wikipedia.org/wiki/List_of_hospitals_in_Massachusetts

https://health.usnews.com/best-hospitals/area/boston-ma

https://www.geeksforgeeks.org/implementing-generic-graph-in-java/

https://www.shiksha.com/online-courses/articles/understanding-dijkstras-algorithm/