Project Based Evaluation

Project Report

Semester-IV (Batch-2023)

AmbuRoute - Ambulance Dispatch and Hospital Directory System

Supervised By:	**Submitted By:**
Ms. Annu Priya	Kartik Joshi (2310992115)
Assistant professor	Harjas Singh Rai (2310992088)
	Himank Goel (2310992097)
	Rudraksh Bhandari (2310992512)

Department of Computer Science and Engineering

Chitkara University Institute of Engineering & Technology,

Chitkara University, Punjab

Table of Contents

1. Introduction

1.1 Background and Motivation

The demand for faster emergency medical services is increasing due to population growth, urban congestion, and an increase in road accidents and medical emergencies. In many developing regions, uncoordinated dispatch systems lead to delays, poor route planning, and lack of access to nearby hospitals. The **AmbuRoute** project was conceptualised to tackle these inefficiencies by simulating a coordinated, algorithm-driven system capable of managing ambulances, patients, and hospitals within a unified framework. By simulating real-time routing based on city maps and hospital directories, this Java-based application offers a foundation for intelligent emergency response systems.

1.2 Objectives of the Project

The main objectives of this project are:

2. Simulate a real-time ambulance dispatch system using object-oriented principles in Java.
3. Implement a graph-based routing engine using Dijkstra's algorithm to find the shortest path.
4. Provide a CLI-based menu system to manage ambulance, hospital, and patient information.
5. Maintain data integrity and prioritisation through structured queues and modular classes.
6. Enable searching and sorting of hospitals by name for administrative convenience.
7. Offer a base structure that can be expanded into a web or mobile platform in the future.

1.3 Scope of the Work

1. The project aims to cover multiple layers of emergency logistics and system design:
2. Simulate a real-world city as a graph where each node represents a location and edges represent roads with distances.
3. Maintain a **directory of hospitals**, including attributes such as name, location, and bed capacity.
4. Build a **dynamic patient queue** to manage emergency cases, prioritising their dispatch and treatment.
5. Integrate a **dispatcher module** to identify the nearest available ambulance to an emergency location using efficient pathfinding.
6. Provide options to **add, view, sort, and search** hospitals and ambulances to simulate real-time decisions in control rooms.
7. Enforce **code modularity** so that each service (ambulance, hospital, routing, patient queue) is independent but collaborates via interfaces.
8. Facilitate future integration with:
    1. GPS APIs for live tracking
    2. Database for persistent storage
    3. GUI for end-user interaction
    4. Analytics for performance monitoring
9. Handle common runtime errors, invalid inputs, and edge cases using structured error-handling.

2. System Environment

2.1 Hardware and Software Requirements

Hardware Requirements:

Processor: Intel i3/i5 or AMD Ryzen 3/5

RAM: Minimum 4 GB (8 GB recommended for IDE performance)

Disk Space: At least 200 MB for Java IDEs and dependencies

Operating System: Windows 10/11 or any modern Linux distribution

Network (Optional): For real-time integrations or APIs

2.2 Java Environment

JDK Version: Java SE 17

Compiler: javac

Runtime**:** JVM (Java Virtual Machine)

2.3 Tools and Utilities Used

IDE: IntelliJ IDEA / Eclipse

Built-in Libraries:

- java.util.Scanner
- java.util.ArrayList
- java.util.HashMap
- java.util.PriorityQueue
- Custom Dijkstra implementation

Version Control (optional): Git

3. Conceptual Overview

3.1 Key Concepts Related to the Project

The *AmbuRoute* project is built upon foundational concepts in object-oriented programming (OOP) and graph theory, which collectively enable the simulation of a real-world ambulance dispatch system. The core structure follows modular object-oriented design, where classes like Ambulance, Hospital, Patient, Graph, and Dispatcher are well-encapsulated and designed to work independently while still interacting seamlessly.

At the heart of the routing system is graph theory, where city locations are modeled as nodes, and roads between them are represented as edges with weights (distances). The graph is implemented using adjacency lists, allowing efficient traversal and scalability for larger networks. For computing the shortest path between two points, the project uses Dijkstra's algorithm, ensuring that ambulances always follow the fastest route to a patient's location.

The project is designed as a CLI simulation, where the user interacts with the system through a console menu. This allows for dynamic testing of all core features without the need for a graphical interface.

Another major concept is queuing and prioritization. A simple queue system handles incoming patients in the order of arrival, simulating real-world emergency triage procedures.

3.2 Project Structure & File Breakdown

- Main.java: The central file that handles user input and connects various components.
- Graph.java: Manages city nodes and edges; contains the Dijkstra algorithm implementation.
- AmbulanceService.java: Adds, views, and dispatches ambulances based on availability and location.
- HospitalDirectory.java: Manages hospital registration, listing, and searching.
- PatientQueue.java: Implements a FIFO queue to manage patients.

- SearchSortUtils.java: Contains reusable methods for sorting hospitals and searching by name.

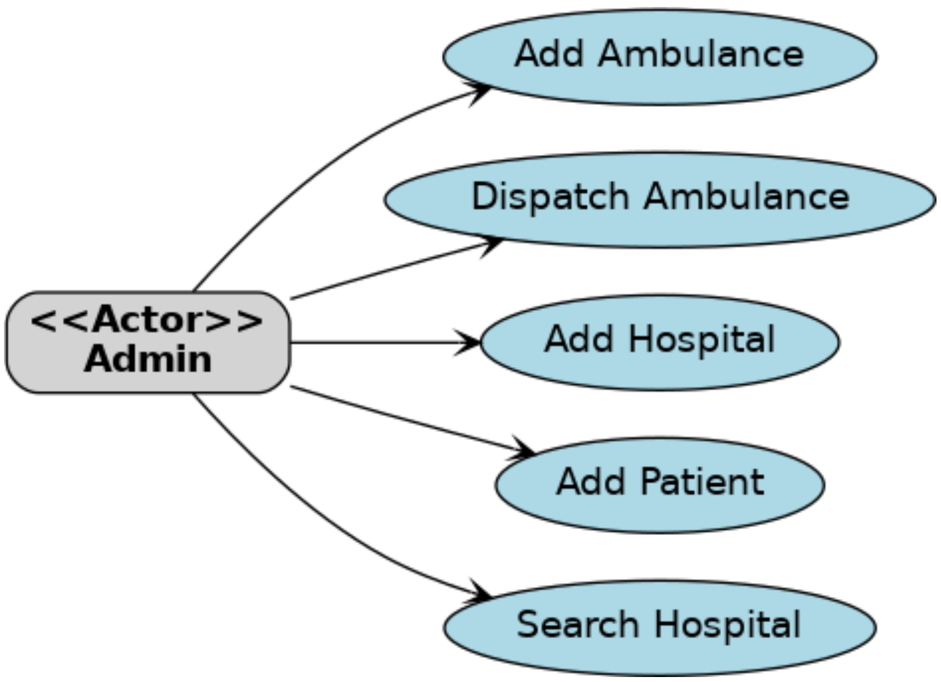
3.3 Core Java Concepts

The project reinforces several core Java programming concepts. **Encapsulation** ensures data security within classes, while **inheritance and polymorphism** could be extended in future versions. Java's **Collections Framework**, including ArrayList, HashMap, and Queue, is used to manage dynamic data efficiently. Additionally, proper **exception handling**and **user input validation** guarantee a smooth user experience, even under invalid or unexpected inputs.

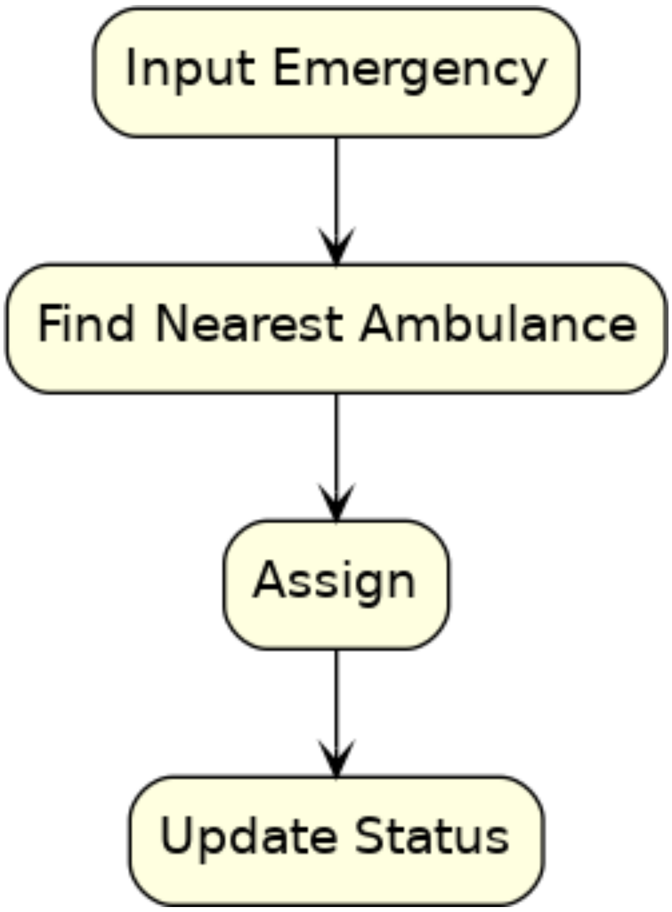
4. UML Diagrams

This section provides a visual representation of the system's functionality and internal workflow using standard UML (Unified Modelling Language) diagrams.

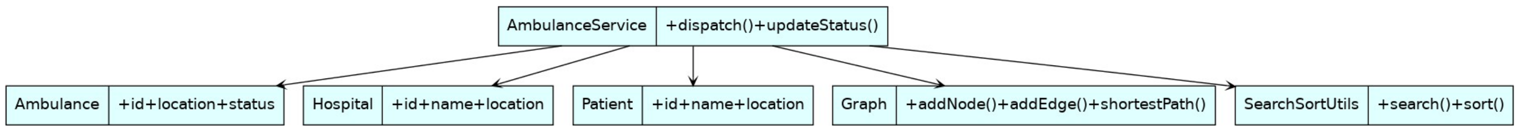
4.1 Use Case Diagram



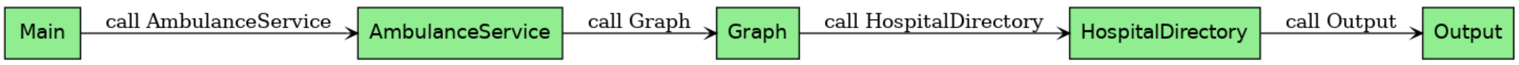
4.2 Activity Diagram



4.3 Sequence Diagram



4.4 Class Diagram (if applicable)



5. Implementation Details

5.1 Step-by-Step Configuration/Development

The implementation of *AmbuRoute* followed a structured development process using Java and object-oriented design principles. First, the Java project was set up in an IDE such as IntelliJ IDEA or Eclipse, which provided a user-friendly environment for writing, testing, and running the code.

Next, the project structure was organized into logical **packages**: `entities` for model classes (`Ambulance`, `Hospital`, `Patient`), `services` for core logic (`AmbulanceService`, `HospitalDirectory`, `PatientQueue`), and `utilities` for helper functions like searching and sorting.

The **Graph class** was then designed to simulate a city map. Locations (nodes) were connected with roads (edges) having weight values representing distances. **Dijkstra's algorithm** was implemented within this class to calculate the shortest path from one location to another—an essential feature for ambulance dispatch decisions.

Each entity class was created with relevant **attributes** and **methods**, such as ambulance availability status, hospital name and location, and patient emergency details. The `Scanner` class was used for real-time input, allowing the user to interact with the system via a **menu-driven CLI**. The `Main.java` file served as the orchestrator, managing user choices and invoking the correct service methods accordingly.

5.2 Commands and Scripts Used

Graph creation:

java

```
cityMap.addEdge("A", "B", 4);
cityMap.addEdge("B", "C", 3);
```

Ambulance dispatch logic:

java

```
Ambulance amb = ambulanceService.findNearest(loc, cityMap);
```

If a valid and available ambulance is found, it is marked as dispatched; otherwise, the system notifies the user.

5.3 Screenshots and Outputs

```
--- AmbuRoute Menu ---

1. Add Ambulance

2. View Hospitals

3. Add Patient

4. Dispatch Nearest Ambulance

...

🚨 Dispatched Ambulance ID: 103 from Location: B

❌ No ambulance available at the moment.
```

6. Security and Optimization

6.1 Hardening Measures Taken

To ensure the application runs reliably and handles unexpected input or behavior gracefully, several **hardening measures**were implemented. Key among these is the use of **null checks** before accessing data from lists, maps, or method return values, which helps prevent NullPointerException—a common cause of program crashes in Java. Additionally, **input validation** was applied to all user inputs using Scanner, ensuring that text fields are not left blank and numeric fields do not receive invalid strings or out-of-range values.

The codebase also uses structured try-catch blocks to **avoid unhandled exceptions**. Any potentially risky operations, such as fetching from an empty list or accessing non-existent graph nodes, are wrapped in exception handling structures. This improves the stability of the system and enhances the user experience by providing clear, informative error messages instead of abrupt crashes.

6.2 Performance Tuning and Efficiency

Efficiency was prioritized, especially in features like ambulance dispatch and route calculation. The routing engine uses Dijkstra's algorithm, an optimal choice for finding the shortest path in a weighted graph. This ensures that the ambulance assigned to an emergency is always the one that can reach the location the fastest, based on the simulated city layout.

Internally, ArrayLists are used for managing dynamic records such as the list of hospitals, ambulances, and patients, offering O(1) access times. For operations requiring key-based retrieval (e.g., looking up a location or checking ambulance availability), HashMaps are used to enable fast O(1) lookups.

6.3 Backup and Recovery Measures

While the current version does not include persistent data storage, a **backup strategy is planned for future phases**. Data such as ambulance availability, patient logs, and hospital details can be stored in .txt or .csv files. This would allow sessions to be saved and restored across executions. Alternatively, integrating a **relational database (like MySQL)** or **cloud-based storage (e.g., Firebase)** would ensure data persistence, reliability, and real-time access in a multi-user environment.

7. Testing and Validation

7.1 Test Scenarios and Expected Results

Test Case	Expected Result	Status
Add Ambulance & Hospital	Successfully stored	✅
Dispatch to known location	Nearest ambulance selected	✅

Dispatch with no ambulances	Show appropriate message	✅
Search hospital (case-insensitive)	Correct match returned	✅

7.2 Troubleshooting Techniques

To identify and fix issues, the project uses try-catch blocks for exception handling, ensuring the program doesn't crash on invalid input. System.out.println() is used for real-time debugging and tracking flow. Additionally, the modular class structure allows testing and debugging individual components like ambulance dispatch or hospital search in isolation.

Try/Catch for exception handling

Use System.out.println for internal debugging

Modular classes allow isolated testing

7.3 Logs and Monitoring Tools

Since the application is currently console-based and does not utilize external logging frameworks or databases, all logs are manually observed through **console outputs** using System.out.println(). These logs act as **real-time indicators** of system behavior, helping developers and testers track the flow of execution, catch logical errors, and verify correct outputs.

Examples of manual log entries:

```
[INFO] Ambulance added: ID=101, Location=A
[INFO] Hospital registered: City Hospital, Location=C
[DEBUG] Running Dijkstra from A to C
```

```
[LOG] Nearest ambulance dispatched: ID=101 from A to D
[ERROR] Invalid location entered: Z
```

These logs help:

- Confirm successful execution of operations (e.g., dispatch or add entries)
- Track data flow and internal state changes
- Identify input issues or incorrect routing logic
- Facilitate step-by-step verification during testing

8. Challenges and Limitations

8.1 Problems Faced During Implementation

During the development of *AmbuRoute*, several technical and logical challenges emerged. One of the primary difficulties was **integrating Dijkstra's algorithm** with user input in a dynamic environment. Since the shortest path needs to be recalculated for every emergency location based on real-time data, synchronizing the user-entered locations with the graph's internal structure required extra validation and mapping logic.

Another issue was **handling invalid or missing data entries**, such as blank hospital names, incorrect location identifiers, or attempts to dispatch ambulances when none were available. Ensuring the application didn't crash under these circumstances involved implementing robust error handling and data verification mechanisms.

Additionally, the project lacked access to **real-time GPS data or a backend database**, which would have made testing and simulation more realistic. Without such components, simulating scenarios like real-time ambulance movement or hospital bed availability had to be done manually using static inputs, limiting realism and scalability.

8.2 Workarounds and Fixes

To address these problems, several practical solutions were implemented. A **manual graph simulation** was used to represent the city map, allowing the development and testing of the routing logic using predefined nodes and distances. This simplified the integration of Dijkstra's algorithm without requiring geographic data.

Input validation was added throughout the application using conditions and exception handling to prevent crashes and guide users through valid inputs. Furthermore, the project followed a **modular class design**, which allowed independent development and debugging of features like ambulance services, hospital directories, and graph utilities—facilitating faster testing and future scalability.

8.3 Known Issues or Constraints

Despite successful implementation, the current system has notable limitations. First, there is **no persistent data storage**, so all information is lost when the program ends. Second, the **interface is text-based**, which may limit usability for non-technical users. Lastly, the **city map is statically defined** in code and does not support dynamic updates or real-world integration yet, restricting flexibility and scalability.

9. Conclusion and Future Work

9.1 Summary of Accomplishments

This project successfully delivers a working prototype of a smart ambulance and hospital dispatch system using Java. The system can dispatch the nearest available ambulance, sort and search hospital entries, and queue patient cases, forming a base for a real-time emergency support system.

9.2 Learnings from the Project

The *AmbuRoute* project provided valuable insights into designing and implementing a simulation-based system using core computer science concepts. One of the most important learnings was how to **simulate real-world emergency response systems** using programming constructs and data structures. Concepts like queues, graphs, and hash maps were not just used theoretically but applied practically to model hospital directories, patient queues, and ambulance routing logic.

We gained a **deep understanding of graph algorithms**, especially Dijkstra's algorithm, and how they can be used to solve real-world problems like finding the shortest route between two locations. Implementing this algorithm from scratch, integrating it with user inputs, and debugging it helped us appreciate its complexity and practical utility.

Moreover, working with Java helped reinforce **clean and modular programming practices**. By breaking down the application into separate, reusable classes (such as AmbulanceService, HospitalDirectory, and Graph), we learned the value of separation of concerns, code reusability, and scalability in software development. The project also honed our skills in input validation, CLI design, and basic debugging using manual logs and test scenarios.

9.3 Future Enhancements

While *AmbuRoute* currently operates as a console-based simulation, there are numerous opportunities for enhancement and real-world application. First, the system can be extended to **integrate real-time GPS tracking APIs**, such as Google Maps, to monitor ambulance movement and dynamically adjust routes based on traffic.

We also plan to **implement persistent storage** using relational (MySQL) or cloud-based (Firebase) databases, so all data—ambulances, hospitals, and logs—can be saved across sessions.

For improved user interaction, a **JavaFX-based GUI or a web-based frontend** can be added, making the system accessible to non-technical users. Future iterations can also include **user authentication modules** for ambulance drivers, hospital admins, and system operators, along with **dashboard-style analytics** that generate reports on dispatch time, average response rates, and hospital utilization. These additions would elevate *AmbuRoute* from a simulation to a real-world emergency support solution.

10.References

Oracle Java SE Documentation

https://docs.oracle.com/en/java/javase/

→ Official documentation for Java programming language features and APIs..

Java Design Patterns - Refactoring Guru

https://refactoring.guru/design-patterns/java

→ Practical insight into object-oriented design and clean coding practices in Java.

Graph Theory Basics - Brilliant.org

https://brilliant.org/wiki/graph-theory/

→ Overview of graphs, nodes, edges, and pathfinding principles used in routing systems.