Data Structures & Object Oriented Programming
Final Project Report

Lecturers: **Jude Joseph Lamug Martinez MCS and**

Nunung Nurul Qomariyah, S.Kom., M.T.I., Ph.D.

Made by:
Farrell Raffelino Sunarman (2802503476)
Heraisya Putri Thalib (2802536442)
Irene Angelina (2802501060)

Class L2CC
Computer Science Program
School of Computing and Creative Arts

BINUS UNIVERSITY INTERNATIONAL
JAKARTA
2025

TABLE OF CONTENTS

Chapter 1
PROJECT SPECIFICATIONS

1.1. Background

Traffic congestion is a long-standing issue in many major urban areas, especially in rapidly growing cities like Jakarta. Managing traffic flow efficiently is not only important for reducing commute times and fuel consumption but also for ensuring that emergency services can reach their destinations without delay. In many cases, traffic flow is influenced by how intersections are managed, determining which vehicles have priority, how long traffic lights remain green, and how emergency vehicles are prioritized.

This project models a simplified traffic simulation at a four-way intersection using various data structures. The aim is to evaluate how different processing strategies affect the overall performance of the system. Three approaches are compared: **Queue (FIFO)**, **Stack (LIFO)**, and **Priority Queue (based on vehicle priority, such as emergency status)**. The goal is to identify which structure provides the best efficiency in processing a stream of incoming vehicles under different traffic behavior assumptions.

At the same time, the project also serves as an application of object-oriented programming (OOP) principles. It incorporates encapsulation, inheritance, polymorphism, and interfaces in the implementation of vehicle behaviors, traffic management, and simulation flow.

1.2. Problem Description

Jakarta, the capital of Indonesia, consistently ranks among the most congested cities in the world. According to multiple traffic studies, the average travel speed in Jakarta during peak hours can fall as low as **10–20 km/h**, significantly below the ideal urban average of **40–60 km/h**. This level of congestion has serious consequences—not only in terms of wasted time and productivity, but also in increased fuel costs, public health issues, and extended emergency response times.

A key factor in urban congestion lies in the management of **intersections**, where multiple roads converge. Poor management of vehicle priority, inefficient routing, and lack of emergency vehicle prioritization can all contribute to traffic buildup and increased delays.

1.3. Project Scope

This project simplifies the real-world conditions of an intersection into a controlled simulation where incoming vehicles are processed using three different vehicle-handling methods:

- **Queue (FIFO).**
- **Stack (LIFO).**
- **Priority Queue (based on vehicle priority).**

Each method is evaluated using a consistent simulation model and benchmarked across various input sizes (100, 500, 1000, 5000, and 10000).

By using **graph representations** of roads and **A* pathfinding** for route calculation, the simulation not only examines how vehicles are processed but also includes a basic navigation component to model realistic movement from one road segment to another.

1.4. Objectives

- Implement a traffic simulation system using Java.
- Apply data structures to handle vehicle processing.
- Use object-oriented programming principles (inheritance, polymorphism, encapsulation, interfaces).
- Compare the runtime and efficiency of each method across different traffic volumes.

Chapter 2
THEORETICAL FOUNDATION

2.1. Data Structures Overview

2.1.1. Queue (FIFO)
A first-in-first-out structure. Vehicles are processed in the order they arrive.
Time Complexity: O(1) enqueue, O(1) dequeue
Space Complexity: O(n)

2.1.2. Stack (FIFO)
A last-in-first-out structure. The newest vehicle is processed first (simulates vehicles that cut in line).
Time Complexity: O(1) push, O(1) pop
Space Complexity: O(n)

2.1.3. Priority Queue
Vehicles are sorted by priority, ensuring that emergency vehicles and alike are processed before regular traffic.
Time Complexity: O(log n) insert, O(log n) remove
Space Complexity: O(n)

2.2. A* Pathfinding Algorithm
Used to calculate the shortest path between two road segments on a graph. A* combines Dijkstra's algorithm with a heuristic to improve efficiency.

2.3. Time and Space Complexity Concepts
Each structure's performance is evaluated based on:
- **Time complexity:** Efficiency of insert / remove operations
- **Space complexity:** Memory used during processing

2.4. Object-Oriented Programming Concepts

2.4.1. Inheritance

EmergencyVehicle class inherits from **Vehicle class** to customize behavior.

2.4.2. Polymorphism

Vehicle objects are processed through a common interface **(Processable)**

2.4.3. Interface

Defines the required methods **(getId(), getPriority(), etc.)** for all processable vehicle types.

2.4.4. Encapsulation

Class fields are private and accessed via public getter methods.

Chapter 3
SYSTEM DESIGN AND IMPLEMENTATION

3.1. System Overview

This system simulates vehicles arriving at an intersection and being processed according to the selected data structure.

3.2. Graph and Road Network Representation

Roads and intersections are represented as nodes and edges in a weighted graph.

3.3 Simulation Design

3.3.1. Traffic Light & Vehicle Processing Logic

Each mode (queue, stack, priority queue) has a different logic for handling vehicle orders.

3.3.2. Modes

- **Queue:** Vehicles move in arrival order.
- **Stack:** Most recent vehicles move first.
- **Priority Queue:** Emergency vehicles bypass regular vehicles.

3.4. Class Structure and OOP Integration

3.4.1. Vehicle and EmergencyVehicle Classes

EmergencyVehicle class overrides priority value via inheritance.

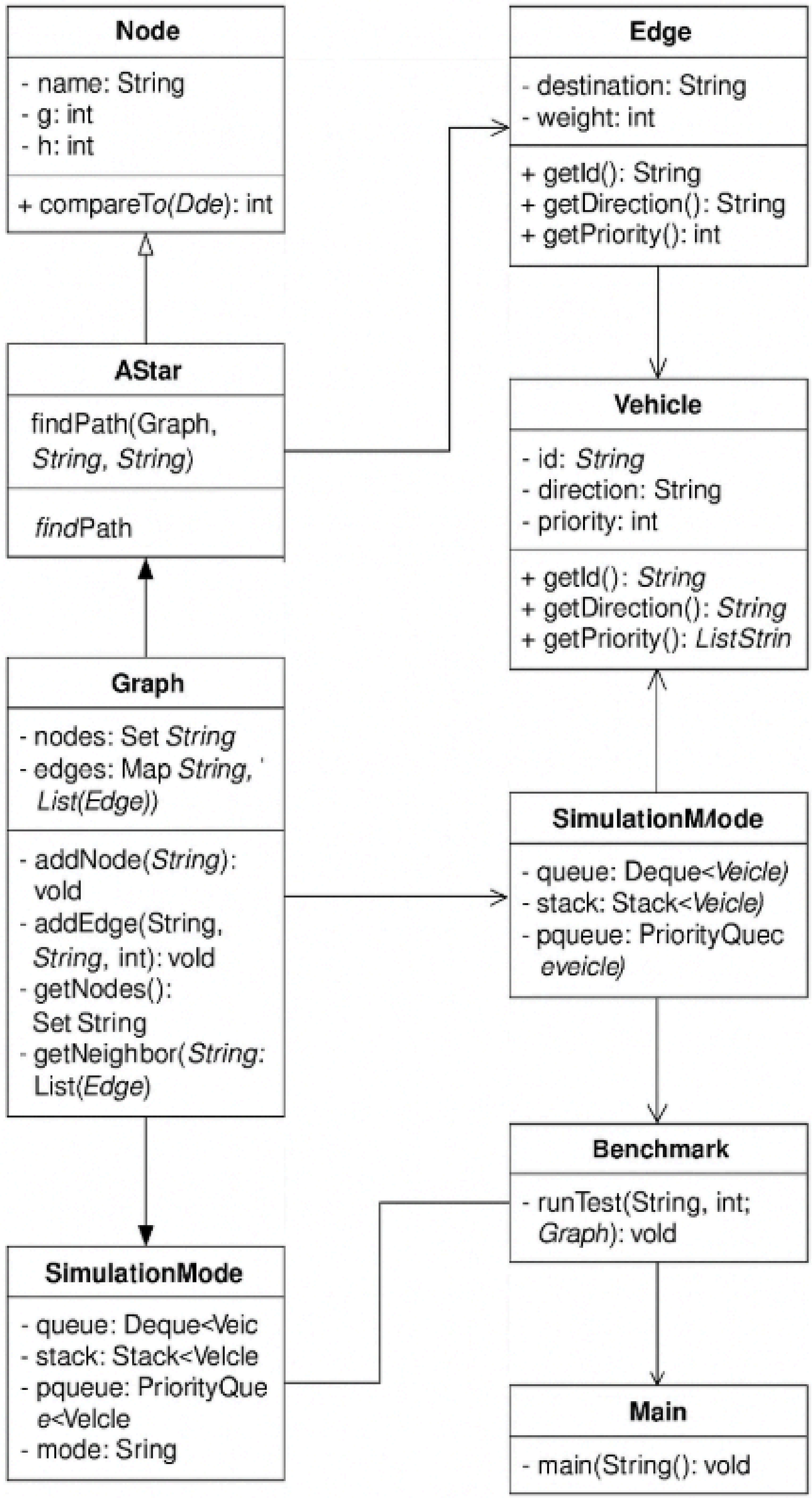
3.4.2. SimulationMode Class

Handles the processing logic based on the selected data structure.

3.4.3. Graph and AStar Classes

Graph models the road layout, while the A* algorithm calculates the routes (for every individual vehicle).

3.4.4. Class Diagram



3.5. User Interface Design

3.5.1. CLI

Text-based mode to record and print runtimes.

3.5.2. GUI

Provides input form, simulation trigger, and results display.

Note: This project was developed and tested using **Java JDK 21**. To ensure compatibility and avoid runtime errors, it is strongly recommended to install **JDK 21 or later** before compiling or executing the program.

3.6 Team Workload

Farrell Raffelino Sunarman

- Wrote most of the final report and created the class diagram.
- Provided input and feedback on simulation design and class layout.
- Contributed to formatting and organization.
- Sourced and cited external references for theoretical foundation and documentation.

Heraisya Putri Thalib

- Compile background information on Indonesian traffic congestion.
- Write the proposal and initial problem description.
- Designed the poster for the project presentation.
- Helped compile and finalize the written report.

Irene Angelina

- Design the overall simulation architecture and data flow.
- Define core classes (e.g., Vehicle, SimulationRunner, AStar) and data structures.
- Implement the main simulation loop that handles vehicle arrival and processing.
- Integrate all data structure implementations within the simulation engine.

All members reviewed and tested the final results. Tasks were shared or rebalanced as needed when facing time constraints.

Chapter 4
TESTING AND EVALUATION

4.1. Testing Methodology

Tests were conducted using inputs of **100, 500, 1000, 5000, and 10000 vehicles.** Each test was repeated five times and averaged. For consistency, the program was closed and reopened between each test to ensure no internal caching or memory reuse affected the results. It was observed that if the program was not restarted, runtimes would decrease regardless of input size or mode due to possible JVM optimizations or caching.

4.2. Test Results

Performance results were recorded for all three modes under identical conditions.

4.3. Detailed Runtime Result Tables

4.3.1. Input Size: 100

Data Structure	Run 1 (s)	Run 2 (s)	Run 3 (s)	Run 4 (s)	Run 5 (s)	Average
Queue	0.02334	0.02421	0.02211	0.02620	0.02223	0.02362
Stack	0.02301	0.02785	0.02506	0.02185	0.02550	0.02465
Priority Queue	0.02485	0.02758	0.02368	0.02303	0.02403	0.02463

4.3.2. Input Size: 500

Data Structure	Run 1 (s)	Run 2 (s)	Run 3 (s)	Run 4 (s)	Run 5 (s)	Average
Queue	0.01131	0.01125	0.01113	0.01155	0.01126	0.01130
Stack	0.01114	0.01098	0.01157	0.01154	0.01193	0.01143
Priority Queue	0.01101	0.01095	0.01112	0.01169	0.01101	0.01116

4.3.3. Input Size: 1000

Data Structure	Run 1 (s)	Run 2 (s)	Run 3 (s)	Run 4 (s)	Run 5 (s)	Average
Queue	0.01314	0.01288	0.01247	0.01267	0.01279	0.01279
Stack	0.01294	0.01353	0.01406	0.01334	0.01330	0.01343
Priority Queue	0.01359	0.01394	0.01375	0.01269	0.01263	0.01332

4.3.4. Input Size: 5000

Data Structure	Run 1 (s)	Run 2 (s)	Run 3 (s)	Run 4 (s)	Run 5 (s)	Average
Queue	0.02683	0.02338	0.02276	0.02588	0.02278	0.02433
Stack	0.02359	0.02542	0.02503	0.02399	0.02330	0.02427
Priority Queue	0.02474	0.02353	0.02363	0.02421	0.02295	0.02381

4.3.5. Input Size: 10000

Data Structure	Run 1 (s)	Run 2 (s)	Run 3 (s)	Run 4 (s)	Run 5 (s)	Average
Queue	0.03371	0.03487	0.03650	0.03564	0.03698	0.03554
Stack	0.03407	0.03299	0.03457	0.03760	0.03354	0.03495
Priority Queue	0.03422	0.03308	0.03431	0.03367	0.03363	0.03378

4.4. Summary of Runtime Averages

Data Structure	100 vehicles	500 vehicles	1000 vehicles	5000 vehicles	10000 vehicles	Average Runtime
Queue	0.02398 s	0.01130 s	0.01279 s	0.02473 s	0.03554 s	0.02167 s
Stack	0.02465 s	0.01143 s	0.01343 s	0.02467 s	0.03455 s	0.02175 s
Priority Queue	0.02463 s	0.01116 s	0.01332 s	0.02381 s	0.03378 s	0.02134 s

4.5. Analysis of Time and Space Efficiency

- **Queue:** O(1) enqueue/dequeue; O(n) space
- **Stack:** O(1) push/pop; O(n) space
- **Priority Queue:** O(log n) insert/remove; O(n) space

While Queue and Stack have faster theoretical operations, Priority Queue consistently performed well even at higher input sizes, offering both efficiency and realistic behavior.

4.6. Overall Ranking and Evaluation

Rank	Structure	Performance
1.	Priority Queue	Best balance of speed & realism
2.	Queue	Simple and efficient
3.	Stack	Efficient but unrealistic

Chapter 5
EVALUATION AND REFLECTION

5.1. Summary of Findings

In this project, we explored how three different data structures – Queue, Stack, and Priority Queue – could be utilized to simulate traffic at intersections. Each data structure was implemented with a focus on efficiency, accuracy, and realistic behavior within a simulated environment.

After extensive testing and simulation at varying input sizes, all three data structures performed well under small to moderate conditions. However, the **Priority Queue** proved to be the most promising candidate for real-world applications due to its ability to handle tasks based on urgency or priority. This aligns well with actual traffic systems, where emergency vehicles, signal prioritization, and road hierarchies affect the order of vehicle movement.

In contrast, the Queue and Stack provided useful baseline simulations but lacked the dynamic responsiveness that Priority Queue has. Stack, operating on a Last-In-First-Out (LIFO) principle, proved to be the least efficient. Meanwhile, Queue, operating on a First-In-First-Out (FIFO) principle, performed well but lacked the flexibility that Priority Queue provides.

Overall, the findings validate the potential of algorithmically driven traffic simulations, even with relatively simple structures.

5.2. Strengths and Limitations

5.2.1 Strengths

- **Clear Structure:** The program was designed using modular, reusable classes, improving maintainability and clarity.
- **Efficient Runtime:** The system performed with low latency under moderate loads, indicating stability.
- **Reusable Logic:** The traffic model can be easily extended or modified due to its object-oriented structure.

5.2.2 Limitations

- **No Visual Vehicle Animations:** Vehicles were represented only through data output or console logs, which limited user understanding and engagement.
- **Simplified Intersection Logic:** The simulation only handled basic four-way intersections, lacking complex route.
- **No Traffic Light or Pedestrian Simulation:** Key elements such as light cycles and pedestrian crossings were omitted for simplicity.
- **Assumes Perfect Conditions:** Factors like road conditions or signal errors were not simulated, which reduces real-world applicability.

5.3. Suggestions for Improvement

To make the traffic simulation more realistic and user-friendly, several enhancements can be considered:

- **Implementation of Traffic Light Timing:** Adding signal timing logic would introduce necessary delays and coordination, reflecting actual road behavior.
- **Visual Vehicle Animations:** A simple animation of vehicles moving could help visualize vehicle movement and improve understanding for both developers and users.
- **Incorporate Randomized Input Events:** Introducing elements such as random delays, accidents, or traffic surges would simulate a more realistic and unpredictable traffic flow.
- **Add Sound or Alert Systems:** Especially in the case of emergency vehicles or congestion build-up, audio/visual cues could help simulate urgency.

5.4. Future Development Opportunities

Beyond improving the current simulation, this project opens the door to several developmental paths:

- **Integration with Real-World Data:** By using APIs or datasets from traffic sensors or GPS services, the simulation could reflect actual traffic patterns in real time.
- **Machine Learning for Traffic Control:** A learning-based system could analyze historical data and adapt signal timing or vehicle prioritization based on predictive modeling.
- **Mobile and Web Optimization:** Making the simulation accessible via mobile or browser platforms would increase its reach and allow for easier demonstrations or educational use.
- **3D or Augmented Reality (AR) Simulation:** To take the visual aspect further, a 3D environment or AR overlay could be used for more immersive presentations, especially useful for urban planning education.

5.5. Conclusion

This project demonstrated the capability of basic data structures in modeling traffic behavior and offered insights into how algorithmic logic could support traffic management. Although simplified, the simulation revealed areas for both technical enhancement and conceptual deepening. With additional work, the framework could evolve into a sophisticated tool used in education, research, or even real-world traffic planning.

Appendices

A. Source Code Access

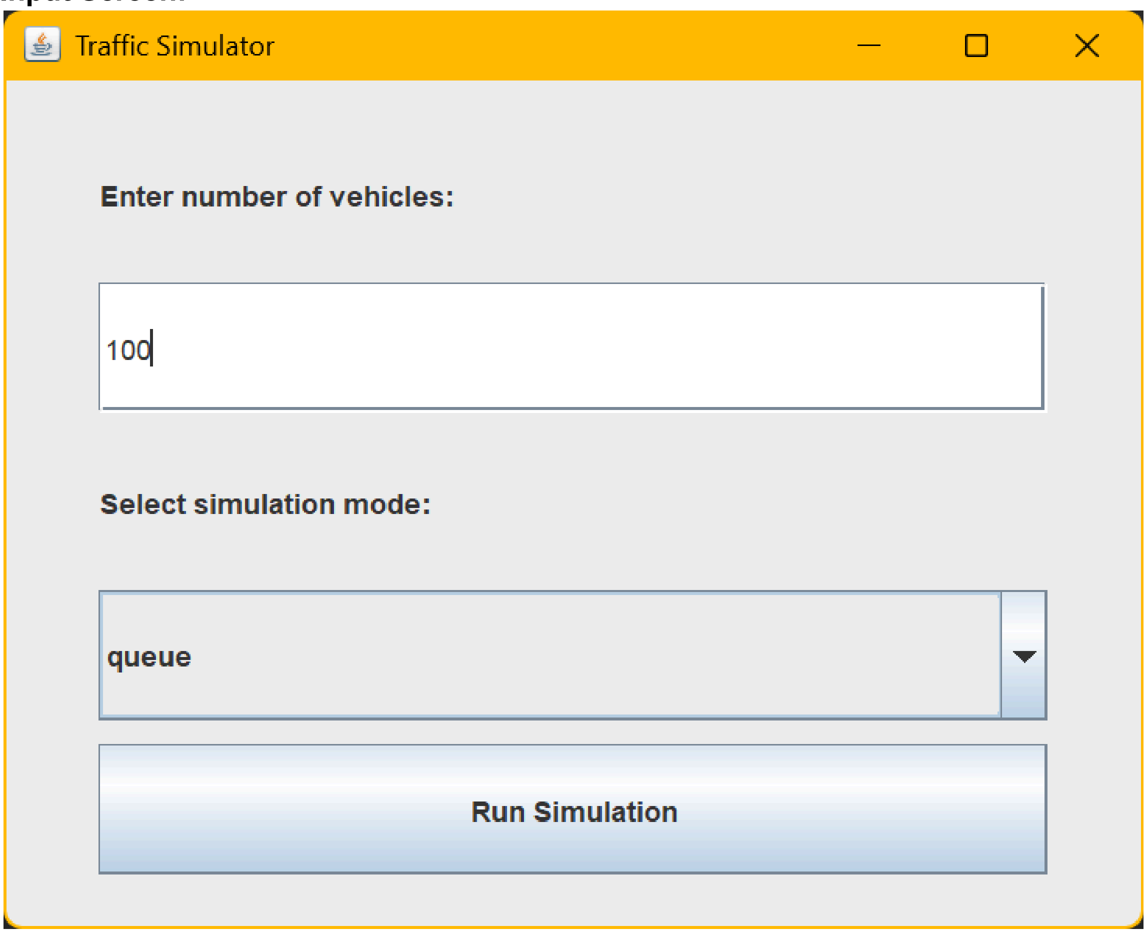
Available at: GitHub Repository.

B. How to Compile and Run

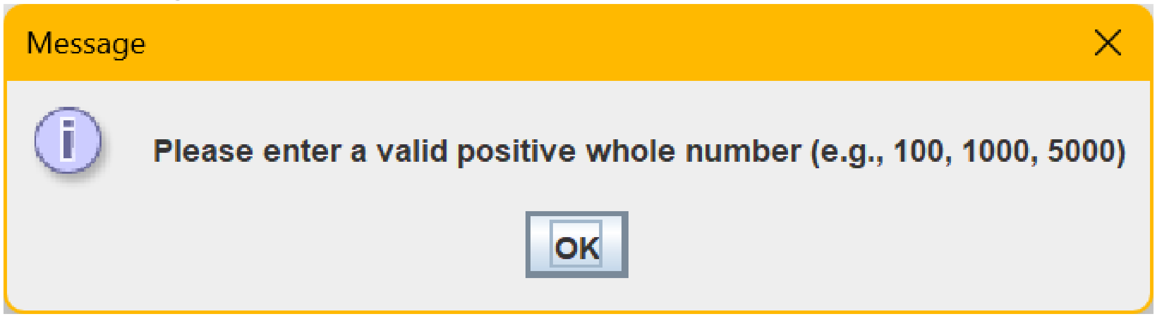
- Open in IntelliJ or VS Code.
- Run Main.java for GUI.
- Run Benchmark.java for CLI testing.

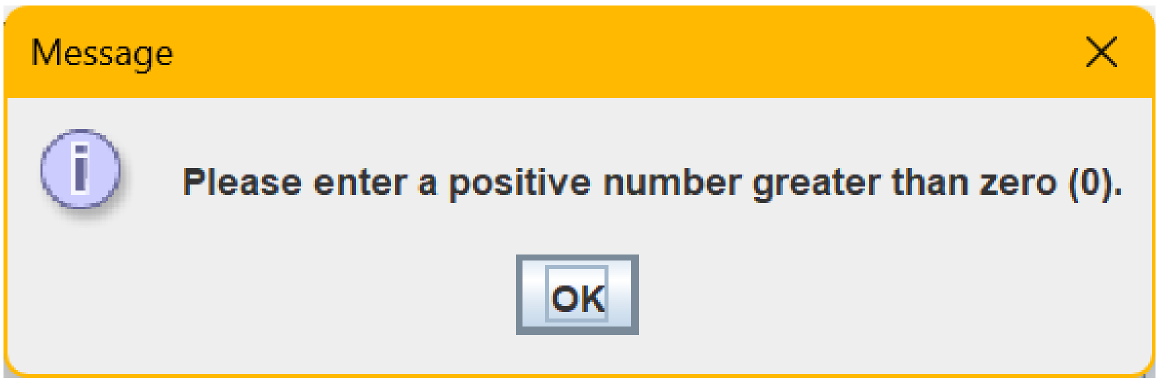
C. Screenshots

Input Screen:

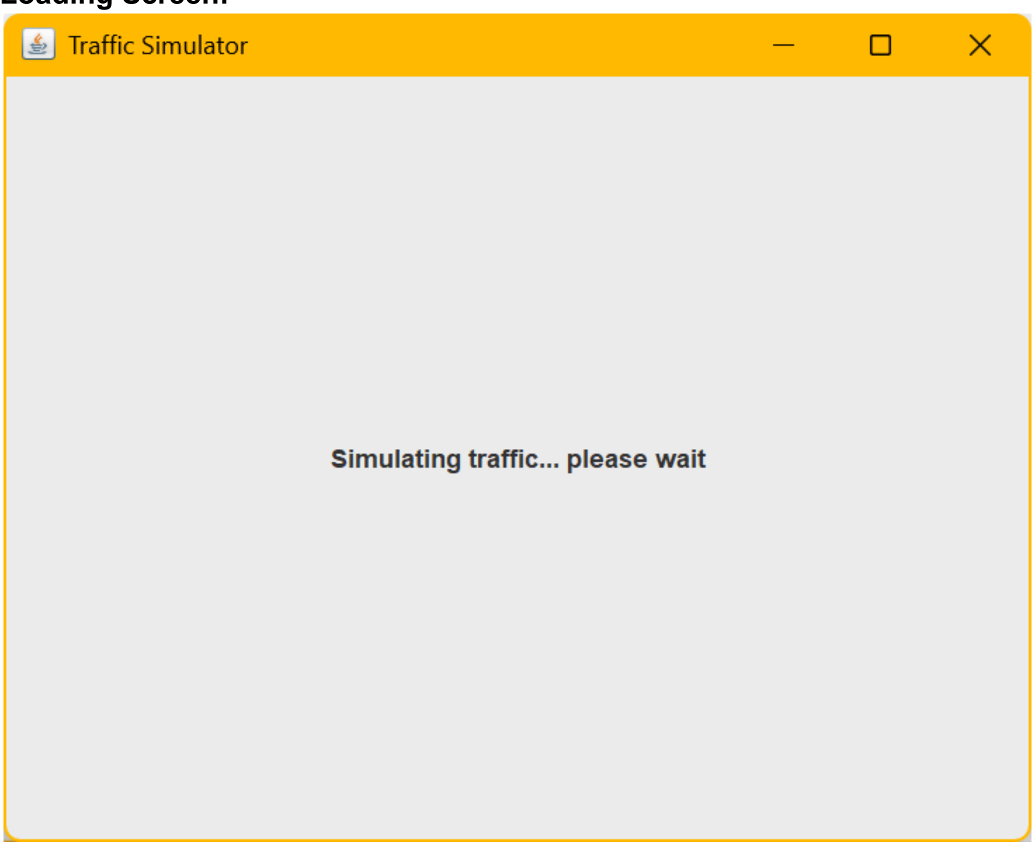


Error Message:

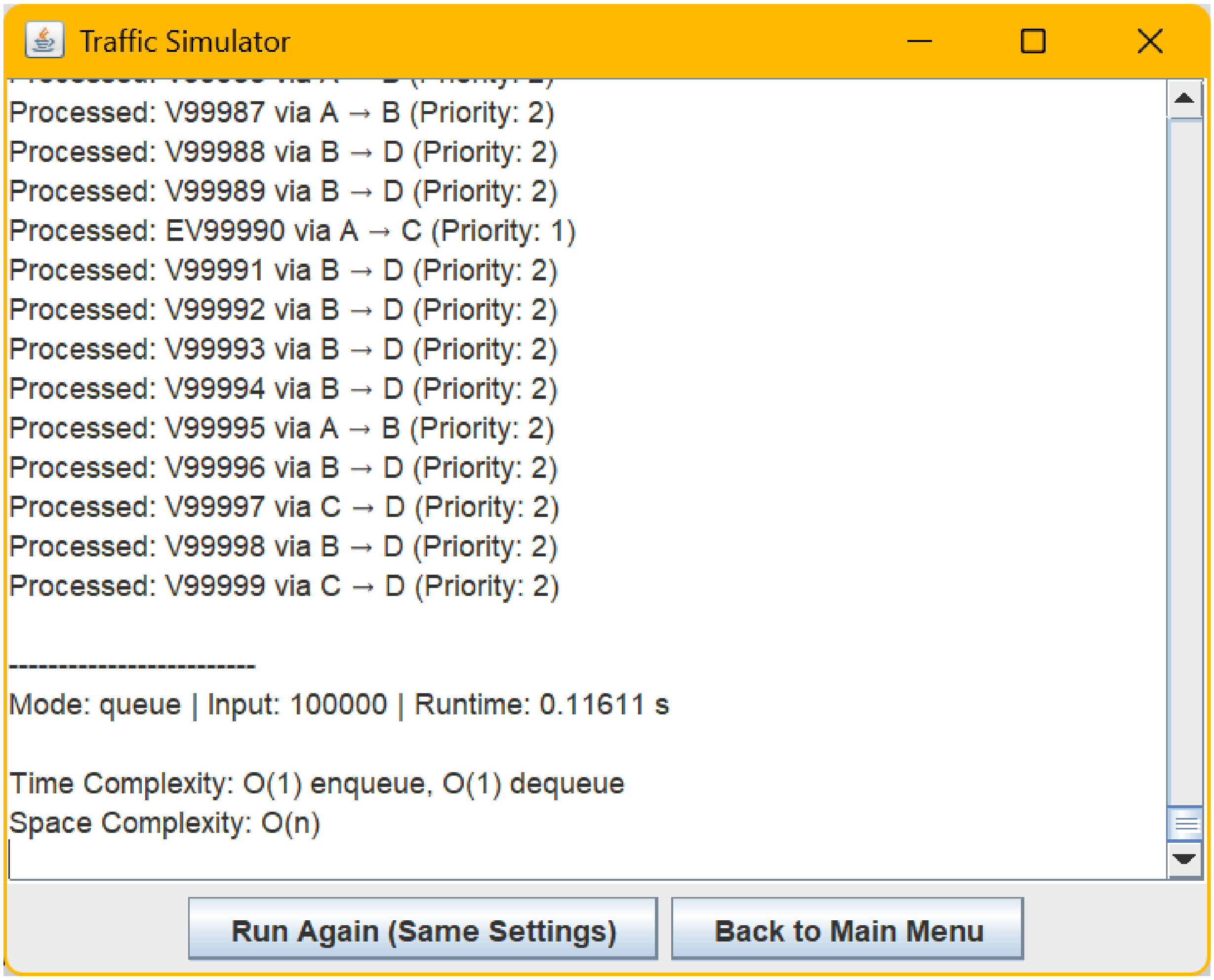




Loading Screen:



Result Screen:



D. References

MTC. “The Dangers of Traffic Congestion.” mtc.ca.gov. Accessed: June 11, 2025. [Online] Available: https://mtc.ca.gov/news/dangers-traffic-congestion

BrokerLink Communications. “What are the problems caused by traffic congestion?” brokerlink.ca. Accessed: June 11, 2025. [Online] Available: https://www.brokerlink.ca/blog/what-are-the-problems-caused-by-traffic-congestion

Mastt. “Everything to Know about Traffic Congestion Risk.” mastt.com. Accessed: June 11, 2025. [Online] Available: https://www.mastt.com/risks/traffic-congestion

J. Mason. “Queues, Stacks, and Priority Queues.” medium.com. Accessed: June 12, 2025. [Online] Available: https://medium.com/age-of-awareness/queues-stacks-and-priority-queues-bd5bbbd9f475

Geeksforgeeks. “How to implement stack using priority queue or heap?” geeksforgeeks.org. Accessed: June 12, 2025. [Online]
Available: https://www.geeksforgeeks.org/implement-stack-using-priority-queue-or-heap/

Winkula. “Java Traffic Simulator (jts).” Github.com. Accessed: May 27, 2025. [Online]
Available: https://github.com/winkula/jts

Yann39. “Highway simulator.” Github.com. Accessed: May 27, 2025. [Online]
Available: https://github.com/Yann39/highway-simulator

E. Project Poster

The final project poster is included in the repository as ‘poster.pdf’.