Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

A Report of the assignment

“Implementing Queue for Solving the Traffic Light Problem”

Subject: Data Structures and Algorithms

[Code No: COMP 206]

Submitted by:

Aayush Dahal (16)

Submitted to:

Mr. Rupak Raj Ghimire

Department of Computer Science and Engineering

Submission Date: 2025 /02 /27

Contents

Abstract

Traffic congestion at intersections poses significant challenges, leading to inefficiencies and safety risks. Traditional traffic management systems often struggle to balance fairness and efficiency, particularly during high-priority situations.

This project addresses these challenges by implementing a traffic management system that integrates traffic lights with queue data structures and priority queues. The system efficiently manages traffic flow by utilizing dynamic arrays to create queue data structures for vehicle management and a priority queue for lane processing. A simulation model featuring four roads (A, B, C, D) with three lanes each is developed, incorporating a priority condition for lane A2 to ensure timely service when vehicle buildup exceeds 10. The system is visualized using SDL3 and designed with object-oriented principles for clarity and extensibility. By leveraging these technologies, this system provides a robust solution to traffic congestion, enhancing both efficiency and fairness at intersections.

Chapter 1: Introduction

Urban traffic congestion is a growing concern, leading to delays, increased air pollution, and significant economic losses. As cities expand and vehicle usage rises, the need for effective traffic management systems becomes increasingly critical.

This project focuses on developing a traffic light management system for a junction connecting two major roads. The junction serves as a critical decision point where vehicles choose one of three alternative routes to proceed. To simulate and manage vehicle flow effectively, the system employs queue data structures. These structures operate on the First-In-First-Out (FIFO) principle, making them ideal for modeling traffic scenarios where vehicles are generally served in the order they arrive.

In addition to standard queues, a priority queue mechanism is implemented to address high-congestion scenarios. For instance, when a specific lane experiences heavy traffic buildup, it is prioritized to ensure smoother flow and reduce delays. By leveraging these data structures, this project provides an efficient and scalable solution to manage traffic at busy intersections while minimizing congestion-related challenges.

1.1 Objectives

1. Use queue data structures to solve a real traffic management problem
2. Use priority queue to prioritize road conditions for proper management
3. Develop a simulator to visualize the traffic management process and implementations.

Chapter 2: Problem Description

2.1 Traffic Junction Overview

The traffic junction serves as a pivotal point where two major roads intersect, requiring vehicles to choose from three possible paths to continue their journey. The traffic management system is designed to handle two primary scenarios:

- **Normal Conditions**: Vehicles from each lane are served equally to ensure fairness in dispatching. The system must guarantee that vehicles are processed in a fair and orderly manner.
- **High-Priority Conditions**: If the number of waiting vehicles on a designated priority road exceeds 10, that road is given priority service until the count drops below 5. Once this threshold is met, the system reverts to normal conditions.

2.2 Road and Lane Configuration

The junction features four major roads (A, B, C, D), each with three lanes:

- Incoming Lane: The first lane of each road (e.g., AL1) is designated for incoming traffic.
- Outgoing Lanes: The second lane of each road is for outgoing traffic and is subject to various traffic light conditions.
- Free Lane: The third lane is reserved for left turns only.
- Priority Lane: Lane AL2 is specifically designated as a priority lane.

2.3 Lane Types

2.3.1 Normal Lanes

All lanes are considered normal unless explicitly marked as priority lanes. These lanes are served based on the average number of vehicles waiting across all normal lanes. The formula for determining how many vehicles to serve at once is given by:

$$|V| = \sum_{i=0}^{n} |Li|$$

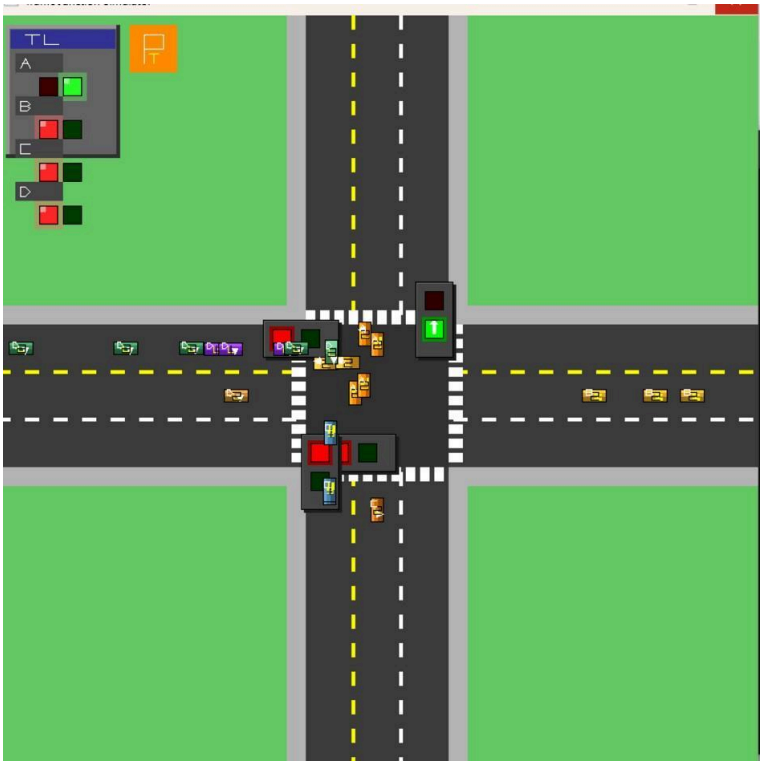

Figure 2.1: Visual representation of the junction

2.3.2 Priority Lane

The priority lane, specifically AL2, is designed to have shorter waiting times. When more than 10 vehicles accumulate in this lane, it receives immediate service following the current light cycle.

2.4Traffic Light System

The system has four traffic lights, each controlling the opposite lane. There are two states in each light:

- Red Light: Stop
- Green Light: Go straight or turn

The lights operate in pairs, ensuring that when any road's light turns green, all others remain red to prevent accidents. Implementing this synchronization was a complex task.

Chapter 3: Data Structures and Algorithms

The Data Structures and algorithms used in the development of the traffic management system are mentioned in this chapter.

3.1 Queue Data Structures

Two main queues are implemented in this system:

3.1.1 Vehicle Queue

This system implements a custom Queue<T> template class to manage the list of vehicles in each lane. The custom queue follows the First-In-First-Out (FIFO) principle, which is suitable for traffic management as it ensures that vehicles that arrive first are generally the first to leave.

The decision to create a custom queue was driven by the assignment requirement to avoid using built-in library queues like std::queue. By implementing a custom queue, the system gains greater control over its behavior and is able to incorporate thread safety features.

```
template<typename T> class
Queue { public:
        Queue() = default;
        ~Queue() = default;

        void enqueue(const T& element) { std::lock_guard<std::mutex>
        lock(mutex); elements.push_back(element);
        }

        T dequeue() { std::lock_guard<std::mutex> lock(mutex); if
                (elements.empty()) { throw std::runtime_error("Empty");
                }
                T element = elements.front();
                elements.erase(elements.begin()); return element;
        }
```

```
// Peek at the front element without removing it
        T peek() const { std::lock_guard<std::mutex> lock(mutex); if
            (elements.empty()) { throw std::runtime_error("Empty queue");
            } return elements.front();
        }

        // Check if the queue is empty bool isEmpty() const {
        std::lock_guard<std::mutex> lock(mutex); return
        elements.empty();
        }

        // Get the size of the queue size_t size() const {
        std::lock_guard<std::mutex> lock(mutex); return
        elements.size();
        }

        // Get all elements for iteration const std::vector<T>&
        getAllElements() const { return elements;
        }

private:
        std::vector<T> elements; mutable std::mutex
        mutex;
};
```

3.1.1: Vehicle Queue Implementation

This system uses vector as an internal container due to it being easy to use and understand. For deque operation the time complexity is technically O(n) but since the simulator has only a few numbers of vehicles, this will not hamper the performance of the system itself.

3.1.2 Lane/Light Priority Queue

Priority Queue has been implemented in the system that handles the priority of lanes for vehicles. This was necessary as the lane AL2 needs priority cases when the waiting list has over 10 vehicles, when the lane gets the highest priority and the lane is served the first.

```
template<typename T> class PriorityQueue { public:
// Element with priority
struct PriorityElement {
T element;
int priority;
// Constructor
PriorityElement(const T& e, int p) : element(e), priority(p) {}
// Comparison operators for sorting
bool operator<(const PriorityElement& other) const {
return priority < other.priority;
}
bool operator>(const PriorityElement& other) const {
return priority > other.priority;
}
};
PriorityQueue() = default;
~PriorityQueue() = default;
// Add element with priority
void enqueue(const T& element, int priority) {
std::lock_guard<std::mutex> lock(mutex);
elements.push_back(PriorityElement(element, priority));
// Sort in descending order (higher priority first)
std::sort(elements.begin(), elements.end(),
std::greater<PriorityElement>());
}
// Get the highest priority element
T dequeue() {
std::lock_guard<std::mutex> lock(mutex);
if (elements.empty()) {
```

```
throw std::runtime_error("Empty Queue");
}
T element = elements.front().element;
elements.erase(elements.begin());
return element;
}
    bool updatePriority(const T& element, int newPriority, std::function<bool(const
    T&, const
                                            T&)> comparator) {
            std::lock_guard<std::mutex> lock(mutex);
            // Find the element auto it = std::find_if(elements.begin(),
            elements.end(),
                                                    [&](const PriorityElement& pe)
                                                { return comparator(pe.element,
                                                  element);
                                            });
            if (it != elements.end()) { // Update the priority it->priority =
                  newPriority; // Re-sort the elements std::sort(elements.begin(),
                  elements.end(), std::greater<PriorityElement>());
                  return true;
            } return false;
      }

private:
      std::vector<PriorityElement> elements; mutable std::mutex mutex;
};
```

Listing 3.2: Priority Queue Implementation

3.2Core Algorithms

3.2.1 Traffic Light Management Algorithm

The system uses an algorithm to change the traffic light according to the road conditions and the traffic. The basic approach of the algorithm is mentioned below:

Algorithm 1 Traffic Light Management

1. Initialize Traffic Light to ALL RED State: Simulation is running.
2. Check Priority Lane (AL2) Status:
    - If AL2 has more than 10 vehicles and is not in priority mode, activate priority mode.
    - Set next state to A GREEN.
3. Adjust Priority Mode:
    - If AL2 has less than 5 vehicles and is in priority mode, deactivate priority mode.
    - If in priority mode and the current state is not A GREEN, transition to ALL RED briefly, then transition to A GREEN.
    - Extend A GREEN duration.
4. Calculate Normal Lane Durations:
    - Calculate the average vehicle count in normal lanes.
    - Determine the appropriate green light duration based on the average vehicle count.
5. Follow Normal Rotation:
    - ALL RED → A → ALL RED → B → ALL RED → C → ALL RED → D.
6. Update Lights Based on Current State.

3.2.2 Vehicle Movement Algorithm

This algorithm handles the vehicle movement through the intersection based on traffic lights and lane types.

Algorithm 2: Vehicle Movement

1. Initialize Lane Status:
    - For each lane in lanes, set isGreenLight to false unless the lane's road has a green light, in which case set isGreenLight to true.
    - If the lane is a free lane (e.g., L3), set isGreenLight to true, as free lanes always have a green light.
2. Update Vehicle Status:

- For each vehicle in the lane, update its status with the correct light status (isGreenLight).

3. Move Vehicles:
    - Move each vehicle along its path based on the current light status.
4. Apply Turning Motion:
    - If a vehicle is in the turning phase, apply smooth curve motion to simulate a realistic turn.
5. Check Vehicle Exit:
    - Check if a vehicle has exited the intersection.
    - If a vehicle has exited, update its road and lane assignment.
6. Position Vehicle in Queue:
    - Position the vehicle in a queue with appropriate spacing to maintain safe distances and simulate real-world traffic flow.

3.2.3 Priority Calculation Algorithm

This algorithm decides the priority condition for the lane AL2 , so the lane is served first under the required condition.

Algorithm 3: Priority Calculation Algorithm

1. Retrieve AL2 Lane Information:
    - Get the current vehicle count in the AL2 lane: vehicleCount = AL2.getVehicleCount().
    - Get the current priority status of AL2: oldPriority = AL2.getPriority().
2. Activate Priority Mode:
    - If vehicleCount > PRIORITY_THRESHOLD_HIGH and oldPriority == 0, activate priority mode.
    - Update AL2's priority using AL2.updatePriority().
    - Set the priority level to a high value (e.g., 100) using AL2.setPriority(100).
    - If the traffic light is not currently in the A GREEN state, set the next state to ALL RED as a transitional state to ensure a safe transition.
3. Deactivate Priority Mode:
    - If vehicleCount < PRIORITY_THRESHOLD_LOW and oldPriority > 0, deactivate priority mode.
    - Update AL2's priority using AL2.updatePriority().
    - Reset the priority to 0 using AL2.setPriority(0).

Chapter 4: Implementation

This chapter describes how the system of traffic management is built. C++ was used for the system development and SDL3 for visualization purposes.

4.1 System Architecture

The system consists of these constituents to complete itself.

- Core Components: Vehicle, Lane, and Traffic Light classes
- Managers: Traffic Manager and File Handler classes
- Visualization: Renderer class
- Utilities: Queue, Priority Queue, and Debug Logger classes
- Separate Programs: Simulator and Traffic Generator

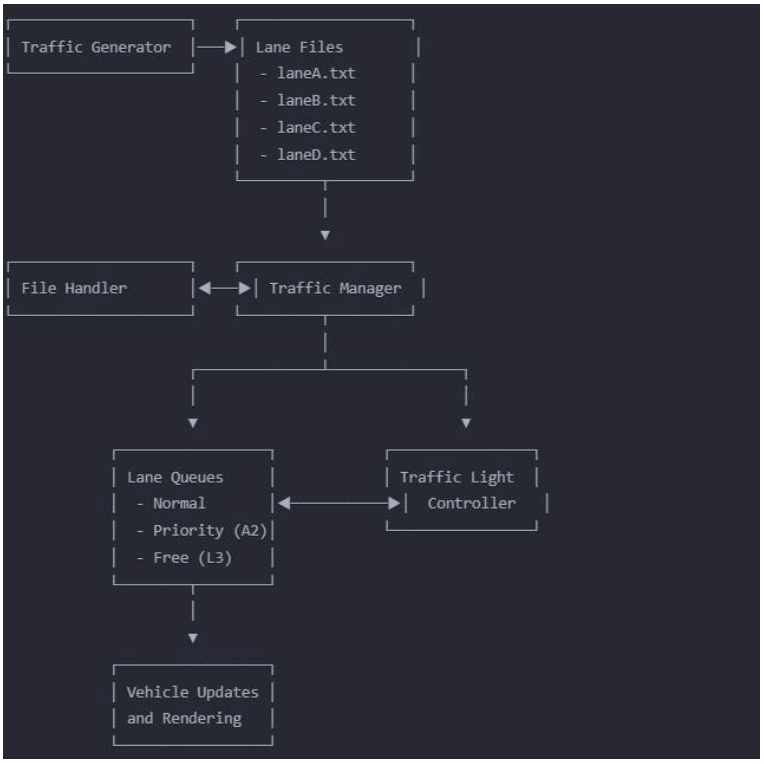

Figure 4.1: System Architecture

4.2 Core Components

4.2.1 Vehicle Class

The Vehicle class represents each vehicle in the simulation. It defines the behaviour of the vehicles and overall appearances as-well.

Key aspects I implemented:

- Tracking vehicle position and movement
- Turning animations
- Implementing waypoint-based path following
- Adding visual indicators showing which direction vehicles are headed

4.2.2 Lane Class

The Lane class manages a queue of vehicles for each lane. It's basically a wrapper around the Queue implementation with some added logic for priorities.

```
class Lane { public:
        Lane(char laneId, int laneNumber);
        ~Lane();

        // Queue operations
        void enqueue(Vehicle* vehicle);
        Vehicle* dequeue(); Vehicle* peek()
        const; bool isEmpty() const; int
        getVehicleCount() const;

    // Priority related operations int
    getPriority() const; void updatePriority();
    bool isPriorityLane() const;

    // Lane identification char getLaneId()
    const; int getLaneNumber() const;
    std::string getName() const;

        // For iteration through vehicles (for rendering) const std::vector<Vehicle*>&
        getVehicles() const;

private:
        char laneId;          // A, B, C, or D int laneNumber; // 1, 2, or 3 bool
        isPriority; // Is this a priority lane
             (AL2)
        int priority;         // Current priority (higher means served first)
        Queue<Vehicle*> vehicleQueue; // Queue for vehicles in the lane
};
```

4.2.3TrafficLight Class

The TrafficLight class was more complex - it handles all the state transitions and timing for the traffic lights. I implemented it as a state machine:

```
class TrafficLight { public:
        enum class State {
                ALL_RED = 0,
                A_GREEN = 1,
                B_GREEN = 2,
                C_GREEN = 3,
                D_GREEN = 4
        };

        TrafficLight();
        ~TrafficLight();

        // Updates the traffic light state based on lane priorities
        void update(const std::vector<Lane*>& lanes);

        // Renders the traffic lights void
        render(SDL_Renderer* renderer);

    // Lane identification char getLaneId()
    const; int getLaneNumber() const;
    std::string getName() const;

        // For iteration through vehicles (for rendering) const std::vector<Vehicle*>&
        getVehicles() const;

private:
        char laneId;          // A, B, C, or D int laneNumber; // 1, 2, or 3 bool isPriority;
           // Is this a priority lane
              (AL2)
        int priority;         // Current priority (higher means served first)
        Queue<Vehicle*> vehicleQueue; // Queue for vehicles in the lane
        };
```

```
        // Returns the current traffic light state
        State getCurrentState() const { return currentState; }

        // Sets the next traffic light state void setNextState(State
        state);

        // Checks if the specific lane gets green light bool isGreen(char lane) const;

private:
        State currentState; State nextState;
        uint32_t lastStateChangeTime; bool
        isPriorityMode;

        // Helper function to calculate average vehicle count float
        calculateAverageVehicleCount(const std::vector<Lane*>& lanes);
};
```

Listing 4.2: TrafficLight Class Implementation

4.3Manager Components

4.3.1 TrafficManager Class

The TrafficManager is the heart of the system, it coordinates everything else. This class went through a few iterations before I was happy with it:

Listing 4.3: TrafficManager Class Implementation

```
class TrafficManager { public:
        TrafficManager();
        ~TrafficManager();

        // Initialize the manager bool initialize();

        // Start/stop the manager void start();
        void stop();

        // Update the traffic state void
        update(uint32_t delta);

        // Get the lanes for rendering
        const std::vector<Lane*>& getLanes() const;

        // Get the traffic light
```

```
TrafficLight* getTrafficLight() const;

        // Check if a lane is being prioritized bool isLanePrioritized(char laneId, int
        laneNumber) const;

private:
        // Lanes for each road std::vector<Lane*> lanes;

        // Priority queue for lane management
        PriorityQueue<Lane*> lanePriorityQueue;

        // Traffic light
        TrafficLight* trafficLight;

        // File handler for reading vehicle data FileHandler* fileHandler;

        // Flag to indicate if the manager is running std::atomic<bool> running;

        // Read vehicles from files void
        readVehicles();

        // Update lane priorities void
        updatePriorities();

        // Process vehicles in lanes void
        processVehicles(uint32_t delta);
};
```

20

4.3.2 FileHandler Class

The FileHandler handles communication between the traffic generator and simulator programs.

```
class FileHandler { public:
        FileHandler(const std::string& dataPath = "data/lanes");
        FileHandler();
```

```
// Read vehicles from lane files std::vector<Vehicle*> readVehiclesFromFiles();

    // Write lane status to file void writeLaneStatus(char laneId, int laneNumber, int
    vehicleCount, bool isPriority);

    // Check if files exist/are readable bool checkFilesExist();

    // Create directories and empty files if they don't exist
    bool initializeFiles();

private:
    std::string dataPath; std::mutex mutex;

    // Read vehicles from a specific lane file std::vector<Vehicle*>
    readVehiclesFromFile(char laneId);

    // Parse a vehicle line from the file
    Vehicle* parseVehicleLine(const std::string& line); };
```

4.4 Visualization Component

The Renderer class handles drawing everything on screen.and is responsible for the simulation of the project.

Listing 4.5: Renderer Class Implementation

```
class Renderer { public:
    Renderer();
    ~Renderer();

    // Initialize renderer with window dimensions bool initialize(int width,
    int height, const std::string& title);

    // Start rendering loop void
    startRenderLoop();
```

```
    // Set traffic manager to render void setTrafficManager(TrafficManager*
    manager);

        // Render a single frame void
        renderFrame();

private:
        // SDL components
        SDL_Window* window;
        SDL_Renderer* renderer;
        SDL_Texture* carTexture;

        // Traffic manager
        TrafficManager* trafficManager;

        // Helper drawing functions void
        drawRoadsAndLanes(); void
        drawTrafficLights(); void
        drawVehicles(); void
        drawDebugOverlay();
};
```

4.5 Communication Between Components

A file-based approach was utilized for communication between the traffic generator and simulator. This method operates as follows:

- The generator writes vehicle data to files such as laneA.txt and laneB.txt.
- The simulator's FileHandler periodically checks these files for new vehicle entries.
- When new entries are found, they are added to the appropriate lane queues.
- Once a vehicle exits the simulation, it is removed from the queue.

This approach is straightforward and effective for the purposes of this project. However, in a real-world scenario, more efficient methods such as shared memory or network communication would be preferable. Initially, a socket-based approach was attempted but encountered issues, leading to the adoption of the simpler file-based method.

Chapter 5: Testing and Results

This chapter outlines the testing methodology employed and presents the results of the traffic management system implementation. A console-based system was developed to verify whether files were being written and read correctly.

5.1 Testing Methodology

The system was tested in multiple phases to ensure its reliability and functionality:

- Unit Testing: Each individual component, including Vehicle, Lane, and TrafficLight, was tested separately to confirm that they functioned correctly on their own.
- Integration Testing: These components were then combined to verify that they worked together seamlessly.
- Specific test scenarios were also created to validate various aspects of the system.

5.2 Test Scenarios

5.2.1 Normal Traffic Flow

This test scenario assessed whether vehicles were served fairly across all lanes under balanced traffic conditions. The simulator was run with approximately equal

vehicle counts in all lanes, and it was observed that the traffic lights rotated through all roads as expected.

5.2.2 Priority Lane Congestion

In this test, a large number of vehicles were intentionally generated in the AL2 lane to trigger the priority condition. It was verified that when the vehicle count exceeded 10, the A road received extended green time until the count dropped below 5. Initially, this test presented some challenges, as it required modifying the traffic generator to create more vehicles for the AL2 lane. A bias factor was added to occasionally direct more vehicles to this lane.

5.2.3 Free Lane Behavior

The behavior of free lanes (L3) was tested to ensure that they always allowed left turns, regardless of the traffic light state. This was confirmed by observing vehicles in L3 moving continuously while other lanes stopped at red lights.

5.2.4 Mixed Traffic Conditions

This was the most realistic test scenario, combining all the above conditions with varying traffic distributions to observe how the system adapted. The system performed well, switching between normal and priority modes as needed.

.

5.1 Testing Methodology

The system was tested in multiple phases to ensure its reliability and functionality:

- Unit Testing: Each individual component, including Vehicle, Lane, and TrafficLight, was tested separately to confirm that they functioned correctly on their own.
- Integration Testing: These components were then combined to verify that they worked together seamlessly.
- Specific test scenarios were also created to validate various aspects of the system.

5.2 Test Scenarios

5.2.1 Normal Traffic Flow

This test scenario assessed whether vehicles were served fairly across all lanes under balanced traffic conditions. The simulator was run with approximately equal vehicle counts in all lanes, and it was observed that the traffic lights rotated through all roads as expected.

5.2.2 Priority Lane Congestion

In this test, a large number of vehicles were intentionally generated in the AL2 lane to trigger the priority condition. It was verified that when the vehicle count exceeded 10, the A road received extended green time until the count dropped below 5. Initially, this test presented some challenges, as it required modifying the traffic generator to create more vehicles for the AL2 lane. A bias factor was added to occasionally direct more vehicles to this lane.

5.2.3 Free Lane Behavior

The behavior of free lanes (L3) was tested to ensure that they always allowed left turns, regardless of the traffic light state. This was confirmed by observing vehicles in L3 moving continuously while other lanes stopped at red lights.

.

5.2.4 Mixed Traffic Conditions

This was the most realistic test scenario, combining all the above conditions with varying traffic distributions to observe how the system adapted. The system performed well, switching between normal and priority modes as needed.

Chapter 6: Time Complexity Analysis

In this chapter, the time complexity of the used algorithms are analysed. Understanding these complexities helped the identification of the potential bottlenecks and optimize where needed.

6.1 Queue Operations

My Vehicle Queue implementation uses a vector as the underlying data structure, resulting in these time complexities:

Operation	Time Complexity	Explanation
enqueue	O(1)	Adding to the end of a vector is O(1) amortized
dequeue	O(n)	Removing from the front requires shifting all elements
peek	O(1)	Accessing the front element is O(1)
isEmpty	O(1)	Checking if the queue is empty is O(1)
size	O(1)	Getting the size of the queue is O(1)

Table 6.1: Time complexity of Vehicle Queue operations

6.2Priority Queue Operations

The Lane/Light Priority Queue implementation uses a sorted vector, which gives these time complexities:

Operation	Time Complexity	Explanation
enqueue	O(n log n)	Insertion + sorting the vector
dequeue	O(1)	Removing the first element
updatePriority	O(n log n)	Finding element + re-sorting

Table 6.2: Time complexity of Priority Queue operations

The O(n log n) operations could be a concern with large datasets, but since we only have 12 lanes total (3 per road × 4 roads), the actual performance impact is minimal.

6.3 Traffic Light Management

The traffic light management algorithm has these complexity components:

Operation	Time Complexity	Explanation
Check priority lane	O(1)	Direct access to AL2 lane
Calculate average vehicle count	O(n)	Iterate through n lanes
Update light state	O(1)	Simple state transition
Overall light management	O(n)	Dominated by average calculation

Table 6.3: Time complexity of Traffic Light Management

The O(n) complexity for calculating average vehicle count is acceptable since n is small (the number of lanes).

6.4 Vehicle Movement

The vehicle movement algorithm has these complexity components:

Operation	Time Complexity	Explanation
Update single vehicle	O(1)	Position calculation is constant time
Update all vehicles in a lane	O(m)	m is the number of vehicles in the lane
Update all lanes	O(n×m)	n lanes with m vehicles each

Table 6.4: Time complexity of Vehicle Movement

6.5 Overall System Complexity

The overall time complexity of the system is dominated by the vehicle movement algorithm, which has a complexity of O(n×m) where n is the number of lanes and m is the average number of vehicles per lane. For a typical traffic junction with a fixed number of lanes (n=12 in this implementation), the complexity effectively becomes O(m), which scales linearly with the number of vehicles in the system. The space complexity of the system is O(n+m), where n is the number of lanes and m is the total number of vehicles in the system.

Chapter 7: Discussion

In this chapter, the challenges faced during the system development are mentioned and also their solutions.

7.1 Challenges and Solutions

7.1.1 Challenge: Thread Safety in Queue Operations

One of the major challenges was ensuring thread safety during queue operations, as both the traffic generator and simulator run concurrently. This initially caused race conditions when accessing shared data. To resolve this issue, the following measures were implemented:

- Mutex Locks: Mutex locks were added to all queue operations to ensure mutual exclusion.
- Atomic Flags: Atomic flags were used for state variables to prevent race conditions.
- File-Based Communication with Proper Locking: A file-based communication approach was adopted, incorporating proper locking mechanisms to avoid simultaneous access conflicts.

7.1.2 Challenge: Priority Lane Implementation

Implementing the priority lane logic proved to be more complex than anticipated. It required careful coordination between the Traffic Light and Traffic Manager classes to ensure that priority mode activated and deactivated correctly. The following solutions were applied:

- State Machine for Traffic Light Transitions: A state machine was created to manage traffic light transitions effectively.
- Priority Update Mechanism: A priority update mechanism was added to the Lane class to handle changes in priority status.
- Priority Queue for Lane Servicing: A priority queue was utilized to manage the servicing order of lanes based on their priority levels.

7.1.3 Challenge: Vehicle Movement Animation

Achieving smooth and natural vehicle movement, particularly during turns, was unexpectedly challenging. The initial implementation caused vehicles to move jerkily at intersections. This issue was addressed by:

- Waypoints for Path Definition: Waypoints were used to define vehicle paths.
- Bezier Curves for Smooth Turns: Bezier curves were implemented to ensure smooth turning motions, which significantly improved animation quality.
- State-Based Movement System: A state-based system was developed to manage vehicle movements systematically.

Although considerable time was spent on this aspect, the resulting smooth vehicle animations were highly satisfying.

7.2 Limitations

The current implementation has several limitations:

- Fixed Junction Layout: The system uses a hardcoded four-road, three-lane layout, limiting flexibility. Supporting dynamic configurations would improve scalability.
- Simplified Traffic Model: Vehicles move at constant speeds without accounting for acceleration, braking, or driver behavior, reducing realism.
- File-Based Communication: While functional, the file-based approach is less efficient compared to other inter-process communication (IPC) methods.

Some of these limitations stem from time constraints, while others were beyond the scope of this project.

7.3 Future Improvements

Given more time and resources, several enhancements could be made:

- Dynamic Junction Configuration: Allowing the system to read junction layouts from configuration files would enable greater flexibility.
- Enhanced Traffic Model: Adding features like acceleration, braking, and realistic driver behavior would improve simulation accuracy.
- Improved IPC Mechanism: Replacing file-based communication with shared memory or message queues would increase efficiency.
- Machine Learning Integration: Implementing machine learning could optimize traffic light timing based on historical traffic patterns.
- Dashboard for Monitoring and Control: Developing a user interface for real-time monitoring and manual control would enhance usability.

The integration of machine learning is particularly promising, as it could enable the system to learn optimal timing patterns from historical data.

7.4 Lessons Learned

This project provided valuable insights into system design and implementation:

- Importance of Data Structures: Choosing appropriate data structures significantly improved both code clarity and performance.
- Benefits of Modular Design: Dividing the system into well-defined components simplified debugging and extension efforts.
- Value of Visualization: Visualizing the simulation helped identify bugs that were not apparent through code inspection alone.
- Challenges of Thread Safety: Concurrent programming proved more complex than initially expected, underscoring the importance of careful design.
- Necessity of Testing: Systematic testing identified many issues early, preventing them from becoming more difficult to resolve later.

Additionally, implementing state machines for traffic light logic was particularly educational and is a design pattern that will likely be used in future projects.

Chapter 8: Conclusion

This project presented a challenging opportunity to design and implement a traffic management system utilizing queue data structures. The outcome is satisfactory, as the system effectively handles both normal and priority traffic conditions, ensuring fair service to vehicles while prioritizing congested lanes when necessary.

The use of queue data structures was a natural choice for this problem. The First-In-First-Out (FIFO) principle inherent in queues aligns well with the typical flow of traffic, where vehicles are served in the order they arrive. The extension to priority queues allowed for the implementation of a special priority condition for the AL2 lane when it becomes congested.

The visual simulation proved particularly beneficial. It provided a clear visual representation of the system's behavior and facilitated verification that all components functioned as expected. Additionally, the visualization significantly aided in identifying and resolving bugs.

From a complexity standpoint, the system scales linearly with the number of vehicles, which is quite efficient. Most operations exhibit constant or linear time complexity, with only a few operations being more computationally intensive.

Although the implementation has some limitations, it provides a solid foundation that could be extended in various ways. The modular design makes it relatively straightforward to add new features or enhance existing ones.

Overall, this project deepened the understanding of data structures and their application to real-world problems. It involved applying queue and priority queue concepts to model traffic flow, utilizing state machines for traffic light logic, and gaining valuable insights into concurrent programming and visualization techniques.