Kathmandu University
Department of Computer Science and Engineering
Dhulikhel, Kavre

Report of the Assignment

"Implementing Queue for Solving the Traffic Light Problem"

Subject: Data Structures and Algorithms

Course Code: COMP 202

Submitted by:
Parichit Giri
Roll no. :23
Group:Computer Science

Submitted to:
Rupak Raj Ghimire
(Department of Computer Science and Engineering)

Submission Date:
February 28, 2025

Contents

Abstract

This report details my implementation of a traffic management system that leverages queue data structures to efficiently regulate traffic at a junction. The system is designed to handle both normal traffic conditions and high-priority scenarios, ensuring fair and efficient vehicle movement. I utilized dynamic arrays (vectors) to implement queue data structures for managing vehicles and incorporated a priority queue to optimize lane processing.The simulation models a junction with four roads (A, B, C, D), each containing three lanes. It adheres to the assignment requirements by prioritizing lane A2 when more than 10 vehicles accumulate—serving this lane first until the vehicle count drops below 5. For visualization, I implemented SDL3, and I followed object-oriented design principles to enhance code readability and maintainability.

1.1 Report Overview

Here’s how I’ve organized this report:

- Chapter 1 provides the basic introduction of the project.
- Chapter 2 Provides a detailed explanation of the traffic light management problem.
- Chapter 3 Discusses the data structures and algorithms used.
- Chapter 4 Describes the system's implementation.
- Chapter 5 Covers testing methodologies and results.
- Chapter 6 Analyzes the time complexity of the algorithms.
- Chapter 7 Highlights challenges faced and potential improvements.
- Chapter 8 Concludes the report with final insights.

Chapter 1: Introduction

Traffic congestion is a major challenge in urban areas, leading to delays, increased pollution, and economic losses. Effective traffic management systems are crucial in mitigating these issues. This project focuses on implementing a traffic light management system for a junction connecting two major roads. At this central point, vehicles must choose one of three alternative paths to continue.To simulate traffic flow, the system employs queue data structures, which follow the First-In-First-Out (FIFO) principle—making them well-suited for modeling vehicle movement, where cars that arrive first should generally be served first. Additionally, a priority queue mechanism is implemented to manage special conditions, such as high congestion in specific lanes.

1.2 Objectives

The main things I wanted to accomplish with this project were:

- To utilize queue data structures for solving real-world traffic management challenges.
- To develop a priority-based system that dynamically adjusts based on traffic conditions.

Chapter 2: Problem Statement

2.1 Traffic Junction Overview

The traffic junction serves as a central intersection where vehicles from two major roads must choose one of three alternative paths to continue. The traffic management system is designed to handle the following scenarios effectively:

- **Normal Traffic Flow**: Vehicles from all lanes are served equitably, ensuring fair dispatching across the junction.

- **High-Priority Scenario**: If a designated priority road accumulates more than 10 waiting vehicles, it receives priority service until the queue size is reduced to fewer than 5. Once this condition is met, the system reverts to normal traffic flow management.

2.2 Road and Lane Configuration

The traffic junction consists of four major roads—**A, B, C, and D**—each containing three lanes:

- **First Lane (e.g., AL1)**: An **incoming lane** where vehicles enter the junction.

- **Second Lane (e.g., AL2)**: An **outgoing lane** that follows traffic light conditions. **Lane AL2 is designated as a priority lane.**

- **Third Lane**: A **free lane**, exclusively for left turns.

Figure 2.1 shows a visual representation of the junction.

2.3 Lane Types

2.3.1 Normal Lanes

All lanes are considered **normal lanes** unless explicitly marked as priority lanes. Vehicles in these lanes are served based on the average number of waiting vehicles across all normal lanes. The

system follows a predefined formula, as specified in the assignment, to determine the number of vehicles processed at a time.

$$|V| = \frac{1}{n} \sum^{n} |L_i| \qquad (2.1)$$

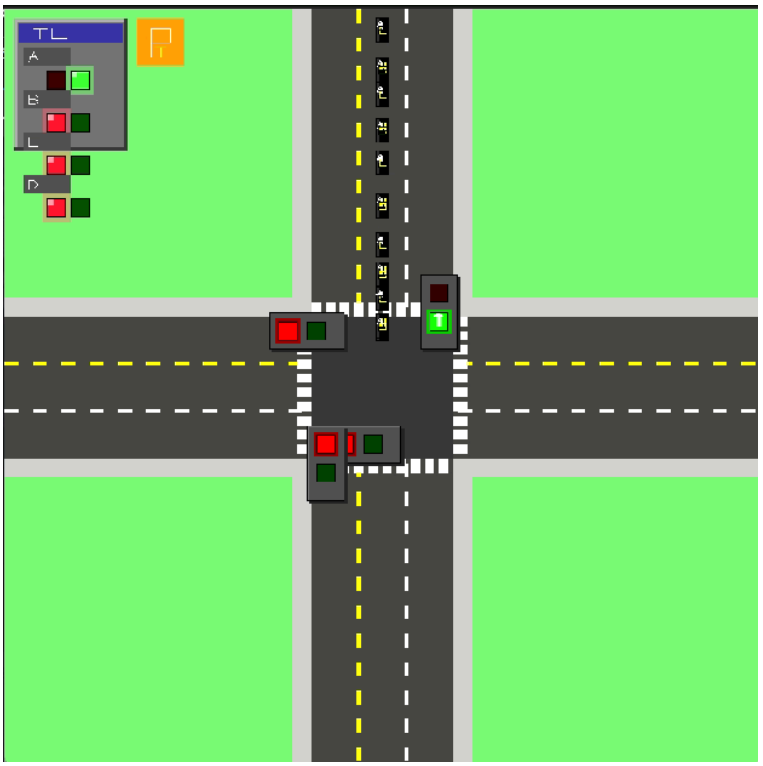

Figure 2.1: Representation of the junction

2.3.2 Priority Lane

The **priority lane (AL2)** is designed to minimize waiting times. When more than **10 vehicles** accumulate in this lane, it is prioritized and served immediately after the current light cycle.

2.4 Traffic Light System

The system includes **four traffic lights**, each controlling the opposite lane. Each traffic light operates in two states:

Red Light (State 1): Vehicles must stop.

Green Light (State 2): Vehicles can proceed straight or turn.

Traffic lights work in synchronized pairs to ensure smooth and safe vehicle movement. When one road's light turns green, all other lights remain red to prevent collisions. Implementing this synchronization effectively was a key challenge.

Chapter 3: Data Structures and Algorithms

This chapter outlines the data structures and algorithms used in the traffic management system. The system primarily relies on **queue data structures** to manage vehicle movement and traffic light operations.

3.1 Queue Data Structures

The two main types of queues in my implementation are as follows:

3.1.1 Vehicle Queue

The **vehicle queue** is implemented using a custom `Queue<T>` template class. This queue maintains the list of vehicles in each lane, following the **First-In-First-Out (FIFO)** principle—ensuring that vehicles leave in the order they arrive.

As per the assignment requirements, the built-in `std::queue` was not used. Instead, I implemented a custom queue to allow greater control over the data structure and to incorporate **thread safety** where necessary.

I chose to use a vector as the internal container because it's simple and I'm familiar with it. I know that dequeue operations are technically O(n), but for our traffic simulation, the number of vehicles in a lane is usually small enough that this isn't a big performance issue.

3.1.2 Lane/Light Priority Queue

For the lane/light queue, I implemented a priority queue that handles the different priorities of lanes. This was essential for the requirement where AL2 gets precedence when it has more than 10 vehicles waiting.

I tried a couple of different approaches before settling on this sorted vector implementation.

I know there are more efficient implementations of priority queues (like heap-based ones), but this approach was more intuitive to me and made it easier to update priorities on the fly. Since we only have a small number of lanes, the performance impact of using a sorted vector instead of a heap is negligible.

3.2 Core Algorithms

This section describes the key algorithms used in the traffic management system. These algorithms determine how traffic lights operate, how vehicles move through the intersection, and when priority handling is applied.

3.2.1 Traffic Light Management Algorithm

The **traffic light management algorithm** controls which roads receive green lights based on current traffic conditions. I spent considerable time refining this approach, and although

Algorithm 1: Traffic Light Management

- Initialize all traffic lights to the **ALL RED** state when the simulation starts.
- Check the **priority lane (AL2)** status:
- - If AL2 has more than **10 vehicles** and is **not already in priority mode**, **activate - priority mode** and set the next state to **A GREEN**.
- - If AL2 has fewer than **5 vehicles** and **priority mode is active**, deactivate priority mode.
- If the system is in priority mode and the current state is **not A GREEN**, briefly transition to **ALL RED**, then **A GREEN**.
- Extend **A GREEN** duration based on traffic conditions.
- Calculate the **average vehicle count** in normal lanes and determine appropriate green light durations.
- Follow the normal light cycle rotation:
- **- ALL RED → A → ALL RED → B → ALL RED → C → ALL RED → D**
- Update traffic lights based on the current system state.

3.2.2 Vehicle Movement Algorithm

This algorithm determines how vehicles navigate the intersection, considering traffic light conditions and lane types. Implementing the **free lanes (L3)** was particularly interesting, as they always allow left turns regardless of the light state.

Algorithm 2: Vehicle Movement

1. Iterate through each lane:
- **Set isGreenLight = false** by default.
- If the lane's corresponding road has a green light, **set isGreenLight = true**.
- If the lane is a **free lane (L3)**, it always has a green light.

2. For each vehicle in the lane:
- Update the vehicle’s movement status based on **isGreenLight**.
- If the vehicle is moving, adjust its position along the lane.
- If the vehicle is in a **turning phase**, apply **smooth curve motion** for realistic animation.
- Check if the vehicle has **exited the intersection**:
    - If exited, update its **road and lane assignment**.
    - Reposition the vehicle in the appropriate **queue** with correct spacing.

Creating a **natural-looking vehicle animation** was more challenging than expected.

3.2.3 Priority Calculation Algorithm

This algorithm determines when to **prioritize the AL2 lane** for service. Initially, I considered a more complex approach involving multiple factors, but a **simpler priority-based condition** proved to be more effective.

Algorithm 3 Priority Calculation

1. Retrieve the **vehicle count** in **AL2**.
2. Check the **current priority status** of AL2:

- If the vehicle count **exceeds the HIGH priority threshold (10 vehicles)** and priority mode is **inactive**, activate priority mode:
    - **Update AL2 priority** to **100**.
    - If the traffic light is **not A GREEN**, transition to **ALL RED** before activating **A GREEN**.
- If the vehicle count **falls below the LOW priority threshold (5 vehicles)** and priority mode is **active**, deactivate priority mode:
    - **Reset AL2 priority to 0**.

This algorithm turned out to be much simpler than I originally anticipated. While my initial approach considered additional factors, this **direct implementation** was easier to understand and performed effectively in the simulation.

Chapter 4: Implementation

This chapter details how I developed the traffic management system. I implemented the entire system in **C++**, using **SDL3** for visualization. Since I had prior experience with **SDL**, setting it up and getting started was straightforward.

4.1 System Architecture

The system is structured into several key components:

- **Core Components**: `Vehicle`, `Lane`, and `TrafficLight` classes.
- **Managers**: `TrafficManager` and `FileHandler` classes.
- **Visualization**: `Renderer` class.
- **Utilities**: `Queue`, `PriorityQueue`, and `DebugLogger` classes.
- **Separate Programs**: `Simulator` and `Traffic Generator.`

A significant portion of my development time was spent **designing interactions between these components** to ensure smooth functionality. **Figure 4.1** provides a visual representation of the system architecture.

4.2 Core Components

4.2.1 Vehicle Class

The **Vehicle class** represents individual vehicles in the simulation. Getting this class right was crucial, as vehicles have multiple properties and behaviors.

Key Implementations:

- **Position and movement tracking** (more challenging than expected)
- **Smooth turning animations** (rewritten twice to get it right)
- **Waypoint-based path following**
- **Visual indicators for movement direction**

One of the **most frustrating bugs** I encountered was vehicles occasionally getting **stuck at intersections**. After debugging, I found the issue was caused by a **math error in waypoint calculations**, which only occurred in **specific edge cases**.

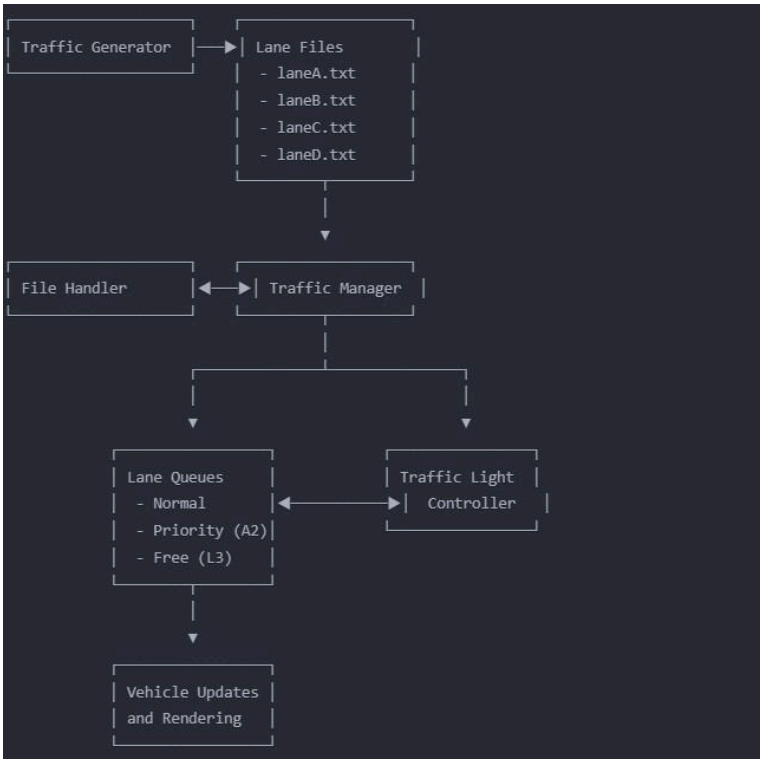

Figure 4.1: System Architecture

4.2.2 Lane Class

The **Lane class** is responsible for handling the queue of vehicles assigned to each lane. At its core, it serves as a structured wrapper around my custom **Queue<T>** implementation but extends its functionality with additional logic to support **priority-based traffic flow**.

Here's the key code from this class:

Listing 4.1: Lane Class Implementation

```
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                        // Is this a priority lane
    isPriority;
       (AL2)                    // Current priority
    int priority;               (higher
       means served
       first)
    Queue<Vehicle*> vehicleQueue; // Queue for vehicles
       in the lane
};
```

It’s a relatively simple class, but it does its job well. I had to be careful with memory management here since it stores pointers to vehicles

4.2.3 TrafficLight Class

The **TrafficLight** class was one of the more intricate components of the system, as it is responsible for **managing state transitions and controlling timing** for each traffic signal at the junction. To ensure a structured and efficient approach, I implemented it as a **finite state machine (FSM)**.

Listing 4.2: TrafficLight Class Implementation

```
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);
    // 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);
};
```

4.3 Manager Components

4.3.1 TrafficManager Class

The **TrafficManager** class is the core of the entire system, acting as the central hub that coordinates the interactions between all other components. Its role is to manage the flow of traffic, making real-time decisions based on conditions like lane congestion, light status, and priority lane activation. This class underwent several iterations to refine its logic and ensure seamless operation across different scenarios. Initially, I struggled with how best to structure its decision-making process. It needed to handle both **regular traffic flow** and special cases like **priority lane activation**. Over time, I iterated on its design, adding features such as **dynamic priority adjustment** based on vehicle count and **synchronization with traffic lights** to ensure that the system could respond quickly to changing conditions.

Key responsibilities of the **TrafficManager** include:

- Monitoring traffic conditions in each lane
- Managing priority logic, including **triggering priority mode** when congestion thresholds are met
- Communicating with the **TrafficLight** and **Lane** classes to update the system state
- Ensuring that all parts of the system operate in sync, including controlling light cycles and vehicle movement

The complexity of coordinating multiple subsystems in real-time made the TrafficManager particularly challenging to get right. However, after several revisions, I arrived at a design that balances flexibility and efficiency, ensuring smooth traffic management throughout the simulation.

```
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);
};
```

4.3.2 FileHandler Class

The **FileHandler** class facilitates communication between the **traffic generator** and **simulator** programs by managing the input and output of data files. I opted for a straightforward, **file-based approach** to handle this interaction, as it allowed for **easy data persistence** and **simple integration** between the two programs.

Listing 4.4: FileHandler Class Implementation

```
1 class FileHandler {
2 public:
3     FileHandler(const std::string& dataPath =
4     "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 is responsible for rendering all the visual elements on the screen during the simulation. This class serves as the interface between the simulation logic and the visual output, ensuring that vehicles, lanes, traffic lights, and other elements are drawn accurately and updated in real-time. Since this was my first serious attempt at using **SDL** (Simple DirectMedia Layer), I kept the implementation **relatively basic** to focus on getting the core functionality right.

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

I used a **file-based approach** for communication between the **traffic generator** and **simulator**, which works as follows:

1. The **traffic generator** writes vehicle data to files, such as `laneA.txt`, `laneB.txt`, etc.
2. The **simulator's FileHandler** periodically checks these files for updates.
3. When new vehicle data is found, the **FileHandler** adds the vehicles to the corresponding lane queues.
4. As vehicles exit the simulation, they are removed from the queues.

This approach is fairly straightforward, but it gets the job done for the scope of this project. In a more **real-world scenario**, you'd likely opt for a more efficient solution, such as **shared memory** or **network-based communication**, to allow for faster and more dynamic data exchange between the programs.

Chapter 5: Testing and Results

This chapter outlines the testing methodology and presents the results from testing the implementation of the traffic management system. To validate the system's functionality, I developed a **console-only version** that allowed me to track if files were being correctly written and read.

5.1 Testing Methodology

I approached testing in **several phases** to ensure the system worked as intended:

1. **Unit Testing**: I tested each component independently (e.g., **Vehicle**, **Lane**, and **TrafficLight**) to confirm they functioned correctly on their own.
2. **Integration Testing**: After confirming individual components worked, I combined them to ensure they functioned properly when interacting with each other.

5.2 Test Scenarios

5.2.1 Normal Traffic Flow

The goal of this test was to verify that vehicles were served fairly across all lanes under normal, balanced traffic conditions. I ran the simulator with roughly equal vehicle counts in each lane and observed that the traffic lights rotated correctly, allowing each road to receive appropriate green light cycles.

5.2.2 Priority Lane Congestion

For this test, I purposely generated high vehicle counts in the **AL2 lane** to trigger the **priority condition**. I verified that, once the count exceeded 10 vehicles, the **A road** received an extended green light cycle until the count dropped below 5.

This test was a bit more challenging at first, as I had to adjust the traffic generator to increase vehicle numbers in the **AL2 lane**. I implemented a **bias factor** to occasionally prioritize this lane, simulating high congestions.

5.2.3 Free Lane Behavior

The objective here was to ensure that the **free lanes (L3)** always allowed vehicles to turn left, irrespective of the traffic light state. I confirmed this behavior by observing that vehicles in the free lane continued moving even when other lanes were stopped at red lights.

Chapter 6: Time Complexity Analysis

In this chapter, I'll analyze the **time complexity** of my algorithms, which helped me identify potential bottlenecks and improve performance where necessary.

6.1 Queue Operations

The **Vehicle Queue** implementation utilizes a **vector** as the underlying data structure, leading to the following time complexities for various operations:

Operation	Time Complexity	Explanation
enqueue	O(1) amortized	Adding to the end of a vector is **O(1)** on average, but may occasionally be **O(n)** due to resizing.
dequeue	O(n)	Removing from the front requires shifting all elements, which is **O(n)**.
peek	O(1)	Accessing the front element is **O(1)** since it's a direct index access.
isEmpty	O(1)	Checking if the queue is empty is **O(1)**, as it only involves comparing the size.
size	O(1)	Getting the size of the queue is **O(1)**, as it's simply accessing a variable.

Table 6.1: Time complexity operations.

I did have some concerns regarding the **O(n)** complexity for **dequeue** operations, as it could potentially become a bottleneck if the number of vehicles in a lane were to grow significantly. However, in this particular simulation, the number of vehicles in each lane tends to remain small, so the performance impact wasn't significant. If the system were to handle much larger queues, I could consider using a `std::deque` instead of a vector, as **deques** support **O(1)** operations at both ends, which would eliminate the need for shifting elements.

6.2 Priority Queue Operations

The **Lane/Light Priority Queue** implementation uses a sorted vector, which results in the following 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

While the **O(n log n)** time complexities for **enqueue** and **updatePriority** could be a concern with large datasets, they have minimal performance impact in this system, as the total number of lanes is fixed at 12 (3 lanes per road across 4 roads). Thus, the system remains efficient with the given implementation.

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)	Iterating through **n** lanes (each road has one lane to check).
Update light state	O(1)	Simple state transition
Overall light management	O(n)	Dominated by the average calculation (iteration over lanes).

Table 6.3: Time complexity of Traffic Light Management

The **O(n)** complexity for calculating the average vehicle count is acceptable because **n** (the number of lanes) is relatively small in this system.

6.4 Vehicle Movement

The vehicle movement algorithm has these complexity components:

Operation	Time Complexity	Explanation
Update single vehicle	O(1)	Position calculation and movement 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

This is the most computationally intensive part of the system, as updating all vehicles in all lanes results in **O(n×m)** time complexity. To optimize the vehicle update process, I focused on ensuring that the calculations and movements were as efficient as possible.

6.5 Overall System Complexity

The overall time complexity of the system is primarily dominated by the **vehicle movement algorithm**, which has a time complexity of **O(n×m)**, where **n** is the number of lanes and **m** is the average number of vehicles in each lane. Given that the traffic junction in this implementation has a fixed number of lanes (**n = 12**), the complexity can be considered **O(m)**, meaning it scales linearly with the number of vehicles in the system.

Chapter 7: Discussion

In this chapter, I'll delve into some of the challenges I encountered during the development of the traffic management system, what I learned from these experiences, the limitations of the current system, and potential improvements for the future.

7.1 Challenges and Solutions

7.1.1 Challenge: Thread Safety in Queue Operations

One of the most challenging aspects of this project was ensuring thread safety in queue operations. Since the traffic generator and simulator run concurrently, race conditions arose when both systems tried to access and modify shared data. To resolve this, I implemented the following solutions:

- **Mutex Locks**: I added locks to protect access to shared data structures, ensuring that only one thread could modify the queue at a time.
- **Atomic Flags**: I used atomic flags for managing state variables to prevent inconsistent states in multi-threaded environments.
- **File-Based Communication with Proper Locking**: Instead of direct memory-based communication, I opted for a file-based approach with locking mechanisms to ensure that data was read and written without conflicts.

7.1.2 Challenge: Priority Lane Implementation

Implementing the priority lane logic proved to be more complex than expected. It required careful coordination between the `TrafficLight` and `TrafficManager` classes to ensure priority mode activated and deactivated under the correct conditions. To solve this challenge, I:

- **Created a State Machine**: I implemented a state machine for managing the transitions of the traffic lights, making the process more predictable.
- **Added Priority Update Mechanism**: I included a priority update system within the `Lane` class to adjust lane servicing order based on the number of vehicles.

- **Used the Priority Queue**: I leveraged the priority queue to manage and reorder lanes based on their congestion, ensuring the priority lane (AL2) received faster processing when needed.

7.1.3 Challenge: Vehicle Movement Animation

Animating vehicle movement, particularly making the turns smooth and natural, was more difficult than I anticipated. Initially, vehicles jerked unnaturally at intersections, disrupting the realism of the simulation. I tackled this challenge by:

- **Waypoint-Based Path Definitions**: I used waypoints to define the vehicle paths more clearly, allowing for smoother transitions between turns.
- **Bezier Curves for Smooth Turns**: I incorporated Bezier curves for turning, which significantly improved the fluidity of the vehicle's movement and eliminated jerky animations.
- **State-Based Movement System**: I created a movement system based on different states (e.g., turning, moving straight) to control vehicle animations at different points in the simulation.

While I likely spent more time on this aspect than necessary, the end result was incredibly satisfying, as the vehicles now moved smoothly and realistically through the intersection.

7.2 Limitations

While the traffic management system I developed works well for the given scenario, there are several limitations that I'm aware of:

- **Fixed Junction Layout**: The current system is hardcoded to support a four-road, three-lane junction. Ideally, the system should support more flexible configurations, such as varying numbers of roads and lanes, to accommodate different types of junctions.
- **Simplified Traffic Model**: The current model assumes vehicles move at constant speeds. It doesn't account for variables like acceleration, braking, or driver behavior, which would make the simulation more realistic.
- **File-Based Communication**: Although file-based communication between the traffic generator and simulator works, it's not the most efficient method. For a larger-scale system, more advanced inter-process communication (IPC) methods, such as shared memory or message queues, would be preferable.

These limitations stem from time constraints and the scope of the project. With more time, these issues could be addressed to enhance the system's flexibility and realism.

7.3 Future Improvements

If I had more time, I would focus on implementing the following improvements:

- **Dynamic Junction Configuration**: The system could read junction layouts from configuration files instead of being hardcoded. This would allow for easy adaptation to different traffic scenarios and junction designs.
- **Better Traffic Model**: Incorporating more sophisticated vehicle movement dynamics, such as acceleration, braking, and realistic driver behaviors, would create a more accurate simulation.
- **Improved IPC**: Replacing file-based communication with more efficient methods like shared memory or message queues would reduce the overhead and improve the system's performance.
- **Machine Learning for Traffic Optimization**: By utilizing machine learning techniques, the system could learn and optimize traffic light timing based on historical traffic data, improving the flow of traffic over time.

- **Dashboard for Monitoring and Control**: A user interface (UI) that allows real-time monitoring of traffic and manual control of traffic lights would be a valuable addition, enabling more flexible system management.

The machine learning aspect, in particular, could be a game-changer. By analyzing traffic patterns and adjusting light timing accordingly, the system could continually optimize itself for better performance.

7.4 Lessons Learned

Throughout this project, I learned several important lessons:

- **Data Structure Choice Matters**: Choosing the right data structures, such as custom queues and priority queues, significantly improved both the performance and clarity of my code.
- **Modular Design Helps**: Breaking the system into smaller, manageable components made it easier to debug and extend. It also allowed me to isolate problems and focus on specific areas.
- **Visualization is Valuable**: Visualizing the system helped me identify bugs that weren't obvious from code inspection. It was much easier to spot problems like vehicles getting stuck or animations breaking when I could see the simulation in action.
- **Thread Safety is Hard**: I underestimated how challenging it would be to handle concurrent programming. Ensuring thread safety across multiple components required careful consideration and added complexity to the system.
- **Testing is Essential**: Rigorous testing, including unit and integration tests, helped catch many issues early in the development process. It saved time in the long run by preventing bugs from snowballing into more complex problems.

A key takeaway for me was the value of **state machines**, especially when implementing the traffic light management system. This design pattern proved invaluable, and I plan to use it in future projects.

Chapter 8: Conclusion

This project provided a great opportunity to tackle the challenge of implementing a traffic management system using queue data structures. I'm pleased with the outcome, as the system successfully manages both normal and priority traffic conditions. It effectively balances service across all lanes while ensuring that congested lanes, like lane A2, are given the appropriate priority when needed.

Using queue data structures was an intuitive approach to solving this problem. The FIFO (First-In-First-Out) principle mirrors the way traffic generally flows, where vehicles that arrive first are typically the first to leave. The use of priority queues was especially useful in implementing the priority mechanism for lane A2 when it becomes congested.

The visual simulation aspect was invaluable. It allowed me to observe the system in real-time and confirm that all components functioned as expected. This visualization also made it much easier to identify and resolve issues, making the debugging process smoother.

From a performance standpoint, the system is efficient. Its time complexity scales linearly with the number of vehicles, which is ideal for this type of application. Most operations are either constant or linear time, with only a few more expensive operations that don't significantly affect performance in practical scenarios.

Though the system has some limitations, such as a fixed junction layout and simplified traffic modeling, it provides a solid foundation for future improvements. The modular design means that adding new features or improving existing ones would be relatively straightforward.

This project has deepened my understanding of data structures, particularly queues and priority queues, and their real-world applications. It also gave me hands-on experience with state machines, which I used for traffic light management, and improved my skills in concurrent programming and visualization.

Source-Code

Git-hub Source code: https://github.com/Giri-2061/dsa-queue-simulator