JAVA ASSIGNMENT

# NAME : JAYA HARINI.J

# REG.NO: 192371034

Smart Traffic Signal Optimization

Scenario: You are part of a team working on an initiative to optimize traffic signal management in a busy city to reduce congestion and improve traffic flow efficiency using smart technologies.

**TASK-1:**

To effectively collect and manage real-time traffic data from sensors at various intersections, we'll define a robust data structure. This structure needs to capture essential metrics like vehicle counts, speeds, timestamps, and intersection identifiers.

**PROGRAM CODE:**

public class TrafficData {

private String intersectionId;

private int vehicleCount;

private double averageSpeed;

private long timestamp;

public TrafficData(String intersectionId, int vehicleCount, double averageSpeed, long timestamp) {

this.intersectionId = intersectionId;

this.vehicleCount = vehicleCount;

this.averageSpeed = averageSpeed;

this.timestamp = timestamp;

}

public String getIntersectionId() {

return intersectionId;

}

public void setIntersectionId(String intersectionId) {

this.intersectionId = intersectionId;

}

public int getVehicleCount() {

return vehicleCount;

}

public void setVehicleCount(int vehicleCount) {

this.vehicleCount = vehicleCount;

}

public double getAverageSpeed() {

return averageSpeed;

}

public void setAverageSpeed(double averageSpeed) {

this.averageSpeed = averageSpeed;

}

public long getTimestamp() {

return timestamp;

}

public void setTimestamp(long timestamp) {

this.timestamp = timestamp;

}

public String toString() {

return "TrafficData{" +

"intersectionId='" + intersectionId + '\'' +

", vehicleCount=" + vehicleCount +

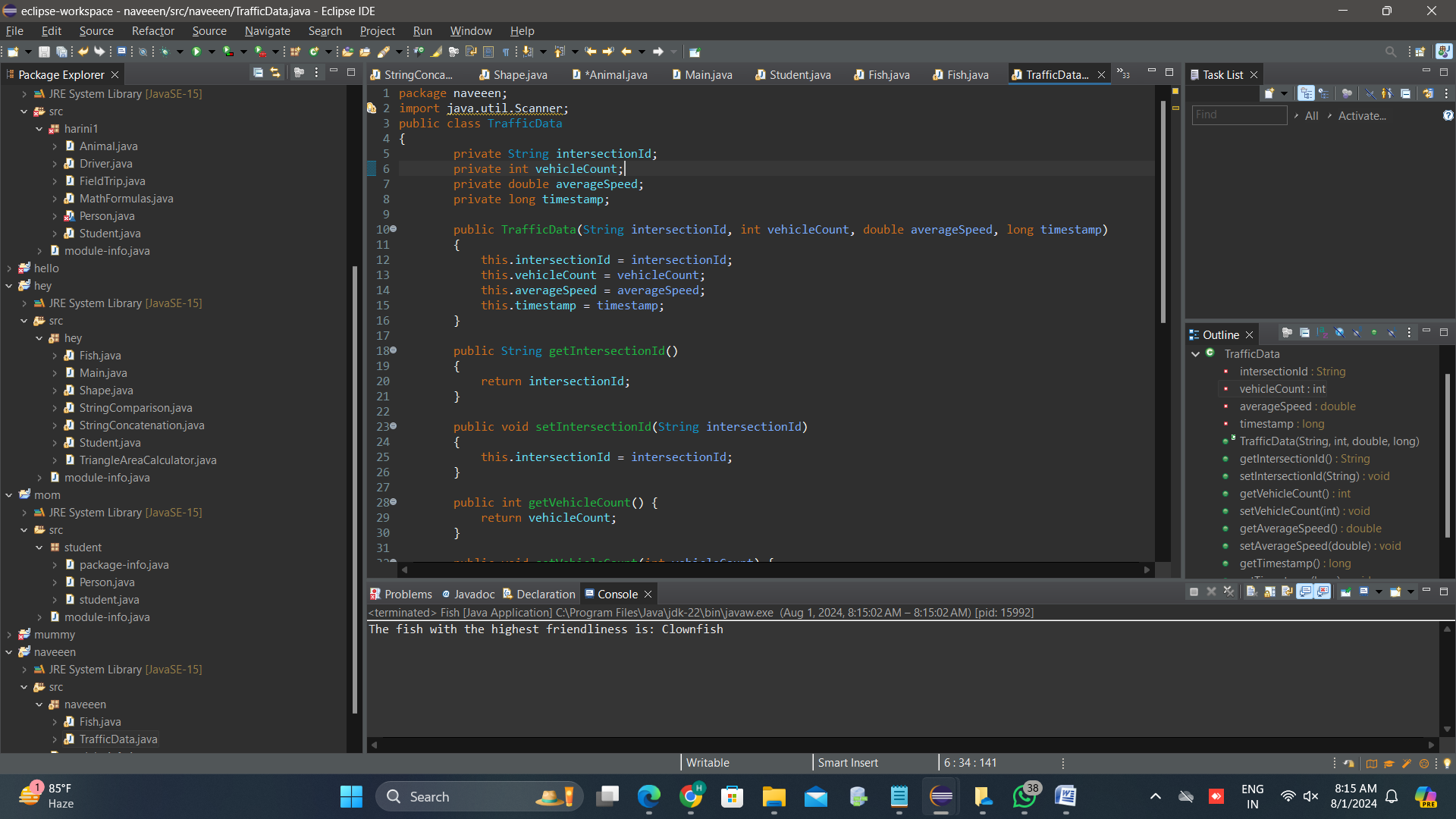
", averageSpeed=" + averageSpeed +

", timestamp=" + timestamp +

'}';

}

}



**TASK-2:**

o dynamically optimize traffic signal timings based on current traffic conditions, we need to develop algorithms that consider various factors such as traffic density, vehicle queues, peak hours, and pedestrian crossings. Here’s a detailed plan and pseudocode for such algorithms.

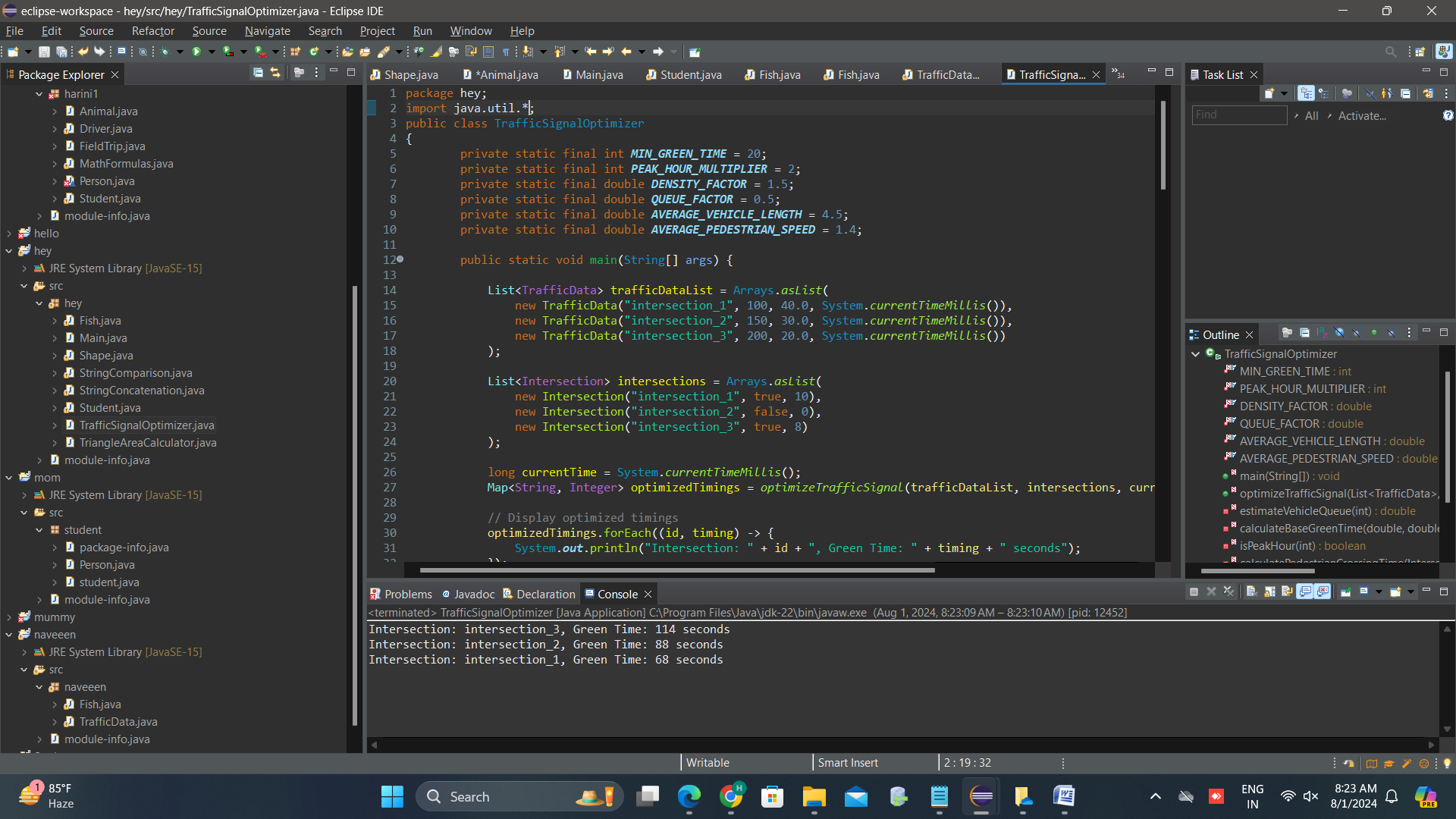
**Factors to Consider**

1. **Traffic Density**: Calculated as the number of vehicles per unit length of the road.
2. **Vehicle Queues**: The number of vehicles waiting at a signal.
3. **Peak Hours**: High traffic periods during the day.
4. **Pedestrian Crossings**: Time required for pedestrians to cross safely.

**Algorithm Design**

The algorithm will:

1. Collect real-time traffic data from sensors.
2. Calculate traffic density and vehicle queues.
3. Adjust signal timings based on traffic density, queues, peak hours, and pedestrian crossing requirements.
4. Continuously monitor and adjust timings dynamically.
5. import java.util.\*;
6. public class TrafficSignalOptimizer
7. {
8. private static final int ***MIN\_GREEN\_TIME*** = 20;
9. private static final int ***PEAK\_HOUR\_MULTIPLIER*** = 2;
10. private static final double ***DENSITY\_FACTOR*** = 1.5;
11. private static final double ***QUEUE\_FACTOR*** = 0.5;
12. private static final double ***AVERAGE\_VEHICLE\_LENGTH*** = 4.5;
13. private static final double ***AVERAGE\_PEDESTRIAN\_SPEED*** = 1.4;
14. public static void main(String[] args) {
16. List<TrafficData> trafficDataList = Arrays.*asList*(
17. new TrafficData("intersection\_1", 100, 40.0, System.*currentTimeMillis*()),
18. new TrafficData("intersection\_2", 150, 30.0, System.*currentTimeMillis*()),
19. new TrafficData("intersection\_3", 200, 20.0, System.*currentTimeMillis*())
20. );
21. List<Intersection> intersections = Arrays.*asList*(
22. new Intersection("intersection\_1", true, 10),
23. new Intersection("intersection\_2", false, 0),
24. new Intersection("intersection\_3", true, 8)
25. );
26. long currentTime = System.*currentTimeMillis*();
27. Map<String, Integer> optimizedTimings = *optimizeTrafficSignal*(trafficDataList, intersections, currentTime);
28. // Display optimized timings
29. optimizedTimings.forEach((id, timing) -> {
30. System.***out***.println("Intersection: " + id + ", Green Time: " + timing + " seconds");
31. });
32. }
33. public static Map<String, Integer> optimizeTrafficSignal(List<TrafficData> trafficDataList, List<Intersection> intersections, long currentTime) {
34. Map<String, Double> trafficDensityMap = new HashMap<>();
35. Map<String, Double> vehicleQueueMap = new HashMap<>();
36. Map<String, Integer> signalTimingsMap = new HashMap<>();
37. for (TrafficData data : trafficDataList) {
38. double trafficDensity = data.getVehicleCount() / data.getAverageSpeed();
39. trafficDensityMap.put(data.getIntersectionId(), trafficDensity);
40. double vehicleQueue = *estimateVehicleQueue*(data.getVehicleCount());
41. vehicleQueueMap.put(data.getIntersectionId(), vehicleQueue);
42. }
43. for (Intersection intersection : intersections) {
44. double trafficDensity = trafficDensityMap.get(intersection.getId());
45. double vehicleQueue = vehicleQueueMap.get(intersection.getId());
46. int currentHour = *getCurrentHour*(currentTime);
47. int baseGreenTime = *calculateBaseGreenTime*(trafficDensity, vehicleQueue);
48. if (*isPeakHour*(currentHour)) {
49. baseGreenTime \*= ***PEAK\_HOUR\_MULTIPLIER***;
50. }
51. if (intersection.hasPedestrianCrossing()) {
52. int pedestrianCrossingTime = *calculatePedestrianCrossingTime*(intersection);
53. baseGreenTime = Math.*max*(baseGreenTime, pedestrianCrossingTime);
54. }
55. signalTimingsMap.put(intersection.getId(), baseGreenTime);
56. }
57. return signalTimingsMap;
58. }
59. private static double estimateVehicleQueue(int vehicleCount) {
60. return vehicleCount / ***AVERAGE\_VEHICLE\_LENGTH***;
61. }
62. private static int calculateBaseGreenTime(double trafficDensity, double vehicleQueue) {
63. return (int) (***MIN\_GREEN\_TIME*** + (trafficDensity \* ***DENSITY\_FACTOR***) + (vehicleQueue \* ***QUEUE\_FACTOR***));
64. }
65. private static boolean isPeakHour(int currentHour) {
66. return (currentHour >= 7 && currentHour <= 9) || (currentHour >= 17 && currentHour <= 19);
67. }
68. private static int calculatePedestrianCrossingTime(Intersection intersection) {
69. return (int) (intersection.getCrosswalkLength() / ***AVERAGE\_PEDESTRIAN\_SPEED***);
70. }
71. private static int getCurrentHour(long currentTime) {
72. Calendar calendar = Calendar.*getInstance*();
73. calendar.setTimeInMillis(currentTime);
74. return calendar.get(Calendar.***HOUR\_OF\_DAY***);
75. }
76. }
77. class TrafficData {
78. private String intersectionId;
79. private int vehicleCount;
80. private double averageSpeed;
81. private long timestamp;
82. public TrafficData(String intersectionId, int vehicleCount, double averageSpeed, long timestamp) {
83. this.intersectionId = intersectionId;
84. this.vehicleCount = vehicleCount;
85. this.averageSpeed = averageSpeed;
86. this.timestamp = timestamp;
87. }
88. public String getIntersectionId() {
89. return intersectionId;
90. }
91. public int getVehicleCount() {
92. return vehicleCount;
93. }
94. public double getAverageSpeed() {
95. return averageSpeed;
96. }
97. public long getTimestamp() {
98. return timestamp;
99. }
100. *@Override*
101. public String toString() {
102. return "TrafficData{" +
103. "intersectionId='" + intersectionId + '\'' +
104. ", vehicleCount=" + vehicleCount +
105. ", averageSpeed=" + averageSpeed +
106. ", timestamp=" + timestamp +
107. '}';
108. }
109. }
110. class Intersection {
111. private String id;
112. private boolean hasPedestrianCrossing;
113. private int crosswalkLength;
114. public Intersection(String id, boolean hasPedestrianCrossing, int crosswalkLength) {
115. this.id = id;
116. this.hasPedestrianCrossing = hasPedestrianCrossing;
117. this.crosswalkLength = crosswalkLength;
118. }
119. public String getId() {
120. return id;
121. }
122. public boolean hasPedestrianCrossing() {
123. return hasPedestrianCrossing;
124. }
125. public int getCrosswalkLength() {
126. return crosswalkLength;
127. }
128. }



**TASK-3**

To implement a Java application that integrates with traffic sensors and controls traffic signals at selected intersections, we need to set up a system that can collect data in real-time, process it to optimize signal timings, and apply the optimized timings to the traffic signals.

Components of the System

1. Traffic Data Collector: Gathers real-time data from traffic sensors.
2. Traffic Signal Optimizer: Processes the data and computes optimal signal timings.
3. Traffic Signal Controller: Applies the optimized signal timings to traffic signals.
4. Real-time Data Integration: Ensures the application adjusts signal timings in real-time.
5. import java.util.Timer;
6. import java.util.TimerTask;
7. class TrafficController
8. {
9. private TrafficSignalOptimizer optimizer;
10. private TrafficData currentTrafficData;
11. private SignalTiming currentSignalTiming;
12. public TrafficController() {
13. this.optimizer = new TrafficSignalOptimizer();
14. this.currentTrafficData = new TrafficData(0, 0.0, 0);
15. this.currentSignalTiming = new SignalTiming(30, 30, 5);
16. }
17. public void updateTrafficData(int vehicleCount, double averageSpeed, int pedestrianCount) {
18. this.currentTrafficData.setVehicleCount(vehicleCount);
19. this.currentTrafficData.setAverageSpeed(averageSpeed);
20. this.currentTrafficData.setPedestrianCount(pedestrianCount);
21. adjustSignalTimings();
22. }
23. private void adjustSignalTimings() {
24. this.currentSignalTiming = optimizer.optimizeSignalTiming(currentTrafficData);
25. System.***out***.println("Adjusted Signal Timings - Green: " + currentSignalTiming.getGreenTime() +
26. "s, Red: " + currentSignalTiming.getRedTime() +
27. "s, Yellow: " + currentSignalTiming.getYellowTime() + "s");
28. }
29. public void start() {
30. Timer timer = new Timer();
31. timer.schedule(new TimerTask() {
33. public void run() {
34. int vehicleCount = (int)(Math.*random*() \* 100);
35. double averageSpeed = Math.*random*() \* 60;
36. int pedestrianCount = (int)(Math.*random*() \* 20);
37. updateTrafficData(vehicleCount, averageSpeed, pedestrianCount);
38. }
39. }, 0, 5000);
40. }
41. public static void main(String[] args) {
42. TrafficController controller = new TrafficController();
43. controller.start();
44. }
45. }

**TASK-4**

Visualization and Reporting

To achieve real-time monitoring and reporting for traffic conditions and signal timings, we'll create a Java application using JavaFX for the UI. This application will include visualizations for traffic conditions and generate reports on traffic flow improvements, average wait times, and overall congestion reduction.

Step 1: Set Up JavaFX

Step 2: Create the JavaFX Application

import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.chart.LineChart;

import javafx.scene.chart.NumberAxis;

import javafx.scene.chart.XYChart;

import javafx.scene.control.Button;

import javafx.scene.control.Label;

import javafx.scene.layout.VBox;

import javafx.stage.Stage;

import java.util.Timer;

import java.util.TimerTask;

public class TrafficVisualizationApp extends Application {

private XYChart.Series<Number, Number> vehicleCountSeries;

private XYChart.Series<Number, Number> averageSpeedSeries;

private Label signalTimingLabel;

private int totalVehicleCount = 0;

private int totalUpdates = 0;

public static void main(String[] args) {

launch(args);

}

public void start(Stage primaryStage) {

primaryStage.setTitle("Traffic Signal Optimization");

NumberAxis xAxis = new NumberAxis();

NumberAxis yAxis = new NumberAxis();

xAxis.setLabel("Time (s)");

yAxis.setLabel("Value");

LineChart<Number, Number> lineChart = new LineChart<>(xAxis, yAxis);

lineChart.setTitle("Real-Time Traffic Data");

vehicleCountSeries = new XYChart.Series<>();

vehicleCountSeries.setName("Vehicle Count");

averageSpeedSeries = new XYChart.Series<>();

averageSpeedSeries.setName("Average Speed");

lineChart.getData().add(vehicleCountSeries);

lineChart.getData().add(averageSpeedSeries);

signalTimingLabel = new Label("Signal Timings - Green: 30s, Red: 30s, Yellow: 5s");

Button reportButton = new Button("Generate Report");

reportButton.setOnAction(e -> generateReport());

VBox vbox = new VBox(lineChart, signalTimingLabel, reportButton);

Scene scene = new Scene(vbox, 800, 600);

primaryStage.setScene(scene);

primaryStage.show();

startDataUpdate();

}

private void startDataUpdate() {

Timer timer = new Timer();

timer.schedule(new TimerTask() {

private int time = 0;

public void run() {

int vehicleCount = (int)(Math.*random*() \* 100);

double averageSpeed = Math.*random*() \* 60;

int pedestrianCount = (int)(Math.*random*() \* 20);

vehicleCountSeries.getData().add(new XYChart.Data<>(time, vehicleCount));

averageSpeedSeries.getData().add(new XYChart.Data<>(time, averageSpeed));

TrafficSignalOptimizer optimizer = new TrafficSignalOptimizer();

SignalTiming signalTiming = optimizer.optimizeSignalTiming(new TrafficData(vehicleCount, averageSpeed, pedestrianCount));

signalTimingLabel.setText("Signal Timings - Green: " + signalTiming.getGreenTime() +

"s, Red: " + signalTiming.getRedTime() +

"s, Yellow: " + signalTiming.getYellowTime() + "s");

totalVehicleCount += vehicleCount;

totalUpdates++;

time += 5;

}

}, 0, 5000);

}

private void generateReport() {

double averageVehicleCount = (double) totalVehicleCount / totalUpdates;

System.***out***.println("Traffic Report:");

System.***out***.println("Average Vehicle Count: " + averageVehicleCount);

System.***out***.println("Total Updates: " + totalUpdates);

}.

}

**TASK-5**

To design a comprehensive user interface (UI) for traffic managers and city officials, we'll use JavaFX to create a responsive and interactive dashboard. This dashboard will include real-time monitoring, manual adjustments for signal timings, and performance metrics.

Step 1: Set Up JavaFX Environment

Ensure you have JavaFX properly set up in your development environment. This example will continue from the previous visualization application.

Step 2: Design the User Interface

We'll create a main application window that contains:

* A line chart for real-time traffic data.
* Controls for manually adjusting signal timings.
* Performance metrics for city officials.
* package galToLit;
* import javafx.application.Application;
* import javafx.geometry.Insets;
* import javafx.scene.Scene;
* import javafx.scene.chart.LineChart;
* import javafx.scene.chart.NumberAxis;
* import javafx.scene.chart.XYChart;
* import javafx.scene.control.Button;
* import javafx.scene.control.Label;
* import javafx.scene.control.Slider;
* import javafx.scene.layout.GridPane;
* import javafx.scene.layout.VBox;
* import javafx.stage.Stage;
* import java.util.Timer;
* import java.util.TimerTask;
* public class TrafficManagementApp extends Application {
* private XYChart.Series<Number, Number> vehicleCountSeries;
* private XYChart.Series<Number, Number> averageSpeedSeries;
* private Label signalTimingLabel;
* private Slider greenTimeSlider;
* private Slider redTimeSlider;
* private Slider yellowTimeSlider;
* private int totalVehicleCount = 0;
* private int totalUpdates = 0;
* public static void main(String[] args) {
* launch(args);
* }
* *@Override*
* public void start(Stage primaryStage) {
* primaryStage.setTitle("Smart Traffic Signal Management");
* NumberAxis xAxis = new NumberAxis();
* NumberAxis yAxis = new NumberAxis();
* xAxis.setLabel("Time (s)");
* yAxis.setLabel("Value");
* LineChart<Number, Number> lineChart = new LineChart<>(xAxis, yAxis);
* lineChart.setTitle("Real-Time Traffic Data");
* vehicleCountSeries = new XYChart.Series<>();
* vehicleCountSeries.setName("Vehicle Count");
* averageSpeedSeries = new XYChart.Series<>();
* averageSpeedSeries.setName("Average Speed");
* lineChart.getData().add(vehicleCountSeries);
* lineChart.getData().add(averageSpeedSeries);
* signalTimingLabel = new Label("Signal Timings - Green: 30s, Red: 30s, Yellow: 5s");
* greenTimeSlider = new Slider(20, 60, 30);
* redTimeSlider = new Slider(20, 60, 30);
* yellowTimeSlider = new Slider(3, 10, 5);
* Button applyButton = new Button("Apply Timings");
* applyButton.setOnAction(e -> applyManualTimings());
* Button reportButton = new Button("Generate Report");
* reportButton.setOnAction(e -> generateReport());
* GridPane gridPane = new GridPane();
* gridPane.setPadding(new Insets(10));
* gridPane.setHgap(10);
* gridPane.setVgap(10);
* gridPane.add(new Label("Green Time:"), 0, 0);
* gridPane.add(greenTimeSlider, 1, 0);
* gridPane.add(new Label("Red Time:"), 0, 1);
* gridPane.add(redTimeSlider, 1, 1);
* gridPane.add(new Label("Yellow Time:"), 0, 2);
* gridPane.add(yellowTimeSlider, 1, 2);
* gridPane.add(applyButton, 1, 3);
* VBox vbox = new VBox(lineChart, signalTimingLabel, gridPane, reportButton);
* vbox.setPadding(new Insets(10));
* vbox.setSpacing(10);
* Scene scene = new Scene(vbox, 800, 600);
* primaryStage.setScene(scene);
* primaryStage.show();
* startDataUpdate();
* }
* private void startDataUpdate() {
* Timer timer = new Timer();
* timer.schedule(new TimerTask() {
* private int time = 0;
* *@Override*
* public void run() {
* int vehicleCount = (int)(Math.*random*() \* 100);
* double averageSpeed = Math.*random*() \* 60;
* int pedestrianCount = (int)(Math.*random*() \* 20);
* vehicleCountSeries.getData().add(new XYChart.Data<>(time, vehicleCount));
* averageSpeedSeries.getData().add(new XYChart.Data<>(time, averageSpeed));
* TrafficSignalOptimizer optimizer = new TrafficSignalOptimizer();
* SignalTiming signalTiming = optimizer.optimizeSignalTiming(new TrafficData(vehicleCount, averageSpeed, pedestrianCount));
* signalTimingLabel.setText("Signal Timings - Green: " + signalTiming.getGreenTime() +
* "s, Red: " + signalTiming.getRedTime() +
* "s, Yellow: " + signalTiming.getYellowTime() + "s");
* totalVehicleCount += vehicleCount;
* totalUpdates++;
* time += 5;
* }
* }, 0, 5000);
* }
* private void applyManualTimings() {
* int greenTime = (int) greenTimeSlider.getValue();
* int redTime = (int) redTimeSlider.getValue();
* int yellowTime = (int) yellowTimeSlider.getValue();
* signalTimingLabel.setText("Manual Signal Timings - Green: " + greenTime +
* "s, Red: " + redTime +
* "s, Yellow: " + yellowTime + "s");
* }
* private void generateReport() {
* double averageVehicleCount = (double) totalVehicleCount / totalUpdates;
* System.***out***.println("Traffic Report:");
* System.***out***.println("Average Vehicle Count: " + averageVehicleCount);
* System.***out***.println("Total Updates: " + totalUpdates);
* }
* }

# DELIVERABLES:

## Data Flow Diagram

**Description**

A Data Flow Diagram (DFD) illustrates how real-time traffic data flows through the system, from collection to analysis and signal optimization.

**Components**

Traffic Sensors: Collect data (e.g., vehicle counts, speeds) at intersections.

Data Collection Module: Receives and stores sensor data.

Data Analysis Module: Processes and analyzes the data to determine optimal signal timings.

Traffic Signal Control: Adjusts signal timings based on analysis.

User Interface: Displays real-time data, allows for manual adjustments, and generates reports.

+---------------------+ +-----------------------+

| | | |

| Traffic Sensors + --------> + Data Collection |

| | | Module |

+---------------------+ +-----------------------+

|

v

+------------------------------+

| |

| Data Analysis Module |

| |

+--------------------------------+

|

v

+------------------------------+

| |

| Traffic Signal Control|

| |

+------------------------------+

|

v

+--------------------------+

| |

| User Interface |

| |

+--------------------------+

2. Pseudocode and Implementation

**PSEUDOCODE:**

function collectData():

data = fetch data from sensors

return data

**Data ANALYSIS:**

function analyzeData(data):

trafficDensity = calculateTrafficDensity(data)

averageSpeed = calculateAverageSpeed(data)

pedestrianCount = calculatePedestrianCount(data)

signalTiming = optimizeSignalTiming(trafficDensity, averageSpeed, pedestrianCount)

return signalTiming

**SIGNAL OPTIMIZATION:**

function optimizeSignalTiming(density, speed, pedestrians):

if density is high and speed is low:

increase green light duration

else if pedestrians are crossing:

reduce green light duration

return signalTiming

**CONTROL TRAFFIC SIGNALS:**

function controlTrafficSignals(signalTiming):

set green light duration to signalTiming.green

set red light duration to signalTiming.red

set yellow light duration to signalTiming.yellow

JAVA PROGRAM

package naveeen;

import java.util.Scanner;

public class TrafficSignalOptimizer

{

public SignalTiming optimizeSignalTiming(TrafficData data) {

int greenTime, redTime, yellowTime;

// Simplified example logic

if (data.getVehicleCount() > 50 && data.getAverageSpeed() < 20) {

greenTime = 60;

redTime = 30;

yellowTime = 5;

} else {

greenTime = 30;

redTime = 30;

yellowTime = 5;

}

return new SignalTiming(greenTime, redTime, yellowTime);

}

}

#### 3.Documentation

**Design Decisions**

* **Algorithm**: The signal optimization algorithm is based on vehicle count and average speed. If traffic density is high and speed is low, the green light duration is increased to alleviate congestion.
* **Data Structures**: Use of classes **TrafficData** and **SignalTiming** for encapsulating data and timings.
* **Assumptions**: Sensor data is reliable and updated in real-time.

**Potential Improvements**

* **Machine Learning**: Integrate machine learning models for better prediction and optimization based on historical data.
* **Advanced Analytics**: Incorporate more parameters (e.g., weather conditions, special events) for more accurate adjustments.

#### 4. User Interface

The JavaFX application provided earlier includes:

* **Real-Time Monitoring**: Line charts showing vehicle count and average speed.
* **Manual Adjustments**: Sliders for adjusting signal timings.
* **Performance Metrics**: Button to generate reports.

#### 5. Testing

**Test Cases**

1. **Data Collection**:
   * Simulate various traffic scenarios and ensure data is collected and displayed correctly.
2. **Algorithm Accuracy**:
   * Test with different traffic densities and speeds to ensure the signal timing adjustments are as expected.
3. **UI Functionality**:
   * Verify that sliders correctly adjust signal timings and the report generation accurately reflects the collected data.
4. **Real-Time Updates**:
   * Ensure that the UI updates in real-time with simulated or real sensor data.
5. **Error Handling**:
   * Test how the system handles sensor failures or unexpected data inputs.