Smart Traffic Signal Optimization

Name: Shaik Mahummad

Reg no:192325031

Tasks:

1. Data Collection and Modeling:

Data Structure for Real-Time Traffic Data:

- **TrafficSensorData**
  - SensorID (PK)
  - IntersectionID (FK)
  - Timestamp
  - VehicleCount
  - AverageSpeed
  - TrafficDensity
  - QueueLength
  - PedestrianCrossingCount

2. Algorithm Design:

Algorithm for Dynamic Signal Timing Optimization:

- **Inputs:**
  - Real-time traffic data (vehicle counts, speeds, density, queue length, pedestrian counts)
  - Historical traffic patterns
  - Time of day (peak hours vs. off-peak hours)

- **Outputs:**
  - Optimized signal timings (Green, Yellow, Red durations)

Pseudocode:

```plaintext
```

```
Algorithm OptimizeSignalTimings
Input: realTimeData, historicalData, timeOfDay
Output: signalTimings

1. Initialize signaltimings
2. For each intersection:
3.     Retrieve current traffic data from realTimeData
4.     Calculate traffic density and queue length
5.     Determine if pedestrian crossing is active
6.     Retrieve historical traffic patterns from historicalData
7.     If current traffic density > threshold or queue length > threshold:
8.         Increase green light duration
9.     If pedestrian crossing is active:
10.        Allocate sufficient time for pedestrian crossing
11.    Else:
12.        Adjust green, yellow, and red light durations based on historical patterns and timeOfDay
13. Return signalTimings
```

3. Implementation:

Java Code for Algorithm and Application Integration:

```java
import java.util.Map;
import java.util.HashMap;
import java.time.LocalTime;

public class TrafficSignalOptimizer {

    private Map<String, Intersection> intersections;
```

```
    public TrafficSignalOptimizer() {
        intersections = new HashMap<>();
        // Initialize intersections and sensors
    }

    public void optimizeSignalTimings() {
        for (Intersection intersection : intersections.values()) {
            TrafficData currentData = intersection.getCurrentTrafficData();
            HistoricalData historicalData = intersection.getHistoricalTrafficData();
            LocalTime timeOfDay = LocalTime.now();

            SignalTimings newTimings = calculateOptimizedTimings(currentData, historicalData, timeOfDay);
            intersection.updateSignalTimings(newTimings);
        }
    }

    private SignalTimings calculateOptimizedTimings(TrafficData currentData, HistoricalData historicalData, LocalTime timeOfDay) {
        SignalTimings timings = new SignalTimings();

        if (currentData.getTrafficDensity() > THRESHOLD || currentData.getQueueLength() > THRESHOLD) {
            timings.increaseGreenDuration();
        }

        if (currentData.isPedestrianCrossingActive()) {
            timings.allocatePedestrianCrossingTime();
        } else {
            timings.adjustBasedOnHistoricalPatterns(historicalData, timeOfDay);
        }

        return timings;
```

```
    }

    // Other methods to retrieve data, update signals, etc.
}

class Intersection {
    private String id;
    private TrafficData currentTrafficData;
    private HistoricalData historicalData;
    private SignalTimings signalTimings;

    public TrafficData getCurrentTrafficData() {
        // Retrieve current traffic data from sensors
    }

    public HistoricalData getHistoricalTrafficData() {
        // Retrieve historical traffic data
    }

    public void updateSignalTimings(SignalTimings newTimings) {
        // Update signal timings
    }
}

class TrafficData {
    private int vehicleCount;
    private double averageSpeed;
    private double trafficDensity;
    private int queueLength;
    private boolean pedestrianCrossingActive;

    // Getters and setters
```

```
}

class HistoricalData {
    // Historical traffic patterns and data
}

class SignalTimings {
    private int greenDuration;
    private int yellowDuration;
    private int redDuration;

    public void increaseGreenDuration() {
        // Increase green light duration
    }

    public void allocatePedestrianCrossingTime() {
        // Allocate time for pedestrian crossing
    }

    public void adjustBasedOnHistoricalPatterns(HistoricalData historicalData, LocalTime timeOfDay) {
        // Adjust timings based on historical data and time of day
    }
}
```

4. Visualization and Reporting:

Visualizations:

- **Real-Time Traffic Monitoring Dashboard:**
  - Map view with intersections highlighted
  - Current traffic density and queue lengths

  - Signal statuses and timings

- **Reports:**
  - Traffic flow improvements (before vs. after)
  - Average wait times at intersections
  - Congestion reduction metrics

Reporting Code Example:
```java
public class TrafficReportGenerator {

    public void generateTrafficFlowReport(List<Intersection> intersections) {
        // Generate report on traffic flow improvements
    }

    public void generateWaitTimeReport(List<Intersection> intersections) {
        // Generate report on average wait times
    }

    public void generateCongestionReductionReport(List<Intersection> intersections) {
        // Generate report on overall congestion reduction
    }
}
```

5. User Interaction:

User Interface (UI) Design:

- **Traffic Manager Interface:**
  - Real-time monitoring of intersections
  - Manual override to adjust signal timings
  - Alerts for unusual traffic conditions

- **City Official Dashboard:**
  - Performance metrics visualization
  - Historical data and trend analysis
  - Reports on traffic management effectiveness

UI Example:

```java
public class TrafficManagerUI {

    public static void main(String[] args) {
        // Create and display UI for traffic managers
    }

    private void initializeUI() {
        // Initialize and layout UI components
    }

    private void displayRealTimeData() {
        // Display real-time traffic data and signal timings
    }

    private void allowManualOverride() {
        // Allow traffic managers to manually adjust signal timings
    }

    private void showAlerts() {
        // Display alerts for unusual traffic conditions
    }
}

public class CityOfficialDashboard {
```

```
    public static void main(String[] args) {
        // Create and display dashboard for city officials
    }

    private void initializeDashboard() {
        // Initialize and layout dashboard components
    }

    private void displayPerformanceMetrics() {
        // Display performance metrics visualization
    }

    private void showHistoricalData() {
        // Display historical data and trends
    }

    private void generateReports() {
        // Generate reports on traffic management effectiveness
    }
}
```

Testing:

Test Cases:

1. **Functional Tests:**
   - Verify real-time data collection from sensors
   - Validate signal timing adjustments based on traffic conditions
   - Ensure manual override functionality works

2. **Performance Tests:**

   - Test system response under high traffic conditions

   - Measure time taken to adjust signal timings

3. **Integration Tests:**

   - Verify integration with traffic sensors

   - Ensure data flow from sensors to the optimization algorithm and signal controllers

4. **User Interface Tests:**

   - Validate usability of traffic manager interface

   - Ensure city official dashboard displays accurate metrics and reports

Test Example:

```java
public class TrafficSignalOptimizerTest {

    @Test
    public void testSignalTimingOptimization() {
        // Setup test data
        TrafficData testData = new TrafficData();
        HistoricalData historicalData = new HistoricalData();
        LocalTime timeOfDay = LocalTime.of(8, 0); // Peak hour

        // Call optimization method
        TrafficSignalOptimizer optimizer = new TrafficSignalOptimizer();
        SignalTimings timings = optimizer.calculateOptimizedTimings(testData, historicalData, timeOfDay);

        // Assert expected signal timings
        assertEquals(expectedGreenDuration, timings.getGreenDuration());
        assertEquals(expectedYellowDuration, timings.getYellowDuration());
        assertEquals(expectedRedDuration, timings.getRedDuration());
    }
}
```

```

## Deliverable:

1. **Data Flow Diagram:**

   - Illustrate the flow from traffic data collection, analysis, and optimization to signal timing adjustments.

2. **Pseudocode and Implementation:**

   - Provide detailed pseudocode and Java code for the traffic signal optimization algorithms.

3. **Documentation:**

   - Explain design decisions, data structures, assumptions, and potential improvements.

4. **User Interface:**

   - Develop interfaces for traffic managers and city officials.

5. **Testing:**

   - Include comprehensive test cases to validate the system.
```