Name: GIRIJA.B

Reg No: 192311044

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.

Data Collection and Modeling:

1. **External Entities**:
    - **Traffic Sensors**: Collect real-time traffic data.
    - **Traffic Managers**: Monitor and manually adjust signal timings.
    - **City Officials**: View performance metrics and historical data.
2. **Processes**:
    - **Data Collection**: Gather traffic data from sensors.
    - **Data Analysis**: Analyze the collected data to determine traffic patterns.
    - **Signal Optimization**: Adjust signal timings based on the analyzed data.
    - **Visualization and Reporting**: Display real-time traffic conditions and generate reports.
3. **Data Stores**:
    - **Traffic Data Store**: Stores collected traffic data.
    - **Optimization Algorithms Store**: Stores algorithms used for signal optimization.
    - **Reports Data Store**: Stores generated reports and performance metrics.
4. **Data Flows**:
    - **Sensor Data Flow**: Flow of traffic data from sensors to the data collection process.
    - **Analyzed Data Flow**: Flow of analyzed data to the signal optimization process.
    - **Optimized Signals Flow**: Flow of optimized signal timings to the traffic lights.
    - **Visualization Data Flow**: Flow of data to the visualization and reporting process.

- **User Interaction Flow**: Flow of information between the system and traffic managers or city officials.

Algorithm Design:

An algorithm to analyze the collected data and optimize traffic signal timings based on traffic density, vehicle queues, peak hours, and pedestrian crossings.

```
public class TrafficSignalOptimizer {
    public void optimizeSignals(List<Sensor> sensors) {
        for (Sensor sensor : sensors) {
            TrafficData data = sensor.getTrafficData();
            // Implement optimization logic
            if (data.getQueueLength() > 10) {
                // Increase green light duration
            }
            if (data.isPedestrianCrossing()) {
                // Adjust signal timing for pedestrian crossing
            }
        }
    }
}
```

Pseudocode:

```
For each sensor in the list of sensors:
    Retrieve traffic data from sensor
    Calculate traffic density and queue lengths
    If traffic density is high:
        Increase green light duration
    If queue length exceeds threshold:
```

Prioritize green light for that lane

If pedestrian crossing is active:

Adjust signal timing for pedestrian safety

Adjust signal timings dynamically based on real-time data

Implementation:

Code:

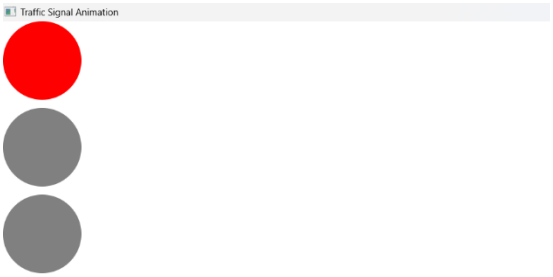
```
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;
import javafx.util.Duration;
import java.util.ArrayList;
import java.util.List;
public class TrafficLight extends Application {
    private List<Sensor> sensors = new ArrayList<>();
    @Override
    public void start(Stage primaryStage) {
        Circle redLight = new Circle(50, Color.RED);
        Circle yellowLight = new Circle(50, Color.GRAY);
        Circle greenLight = new Circle(50, Color.GRAY);
```

```
VBox root = new VBox(10);
root.getChildren().addAll(redLight, yellowLight, greenLight);
Scene scene = new Scene(root, 200, 600);
primaryStage.setTitle("Traffic Signal Animation");
primaryStage.setScene(scene);
primaryStage.show();
sensors.add(new Sensor("Intersection 1"));
sensors.add(new Sensor("Intersection 2"));
Timeline timeline = new Timeline(
        new KeyFrame(Duration.seconds(0), e -> {
            redLight.setFill(Color.RED);
            yellowLight.setFill(Color.GRAY);
            greenLight.setFill(Color.GRAY);
        }),
        new KeyFrame(Duration.seconds(3), e -> {
            redLight.setFill(Color.GRAY);
            yellowLight.setFill(Color.YELLOW);
            greenLight.setFill(Color.GRAY);
        }),
        new KeyFrame(Duration.seconds(6), e -> {
            redLight.setFill(Color.GRAY);
            yellowLight.setFill(Color.GRAY);
            greenLight.setFill(Color.GREEN);
        }),
        new KeyFrame(Duration.seconds(9), e -> {
```

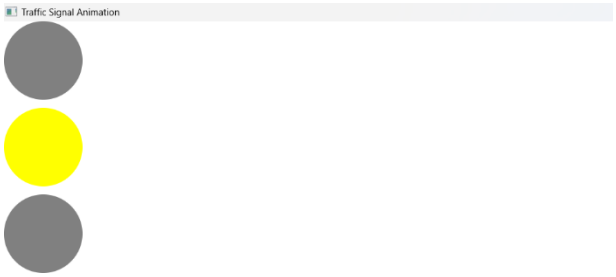
```
                redLight.setFill(Color.RED);
                yellowLight.setFill(Color.GRAY);
                greenLight.setFill(Color.GRAY);
            })
        );
        timeline.setCycleCount(Timeline.INDEFINITE);
        timeline.play();
        Timeline optimizationTimeline = new Timeline(new KeyFrame(Duration.seconds(1), e -
> {
            TrafficSignalOptimizer optimizer = new TrafficSignalOptimizer();
            optimizer.optimizeSignals(sensors);
        }));
        optimizationTimeline.setCycleCount(Timeline.INDEFINITE);
        optimizationTimeline.play();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

Visualization and Reporting:

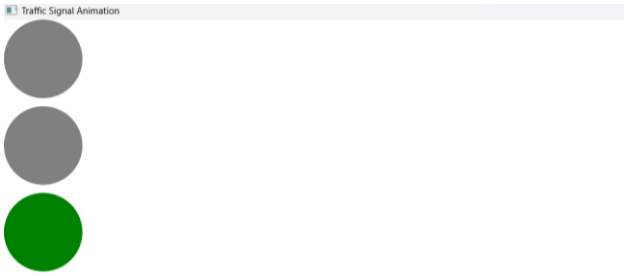
OUTPUT:


Traffic Signal Animation


93°F
Mostly cloudy
Search


Traffic Signal Animation


93°F
Mostly cloudy
Search



Initially, the first circle is red, which means stop. After a few seconds, it turns off, and the second circle lights up in yellow, indicating to wait. Then the second circle goes off, and the third circle turns green, signaling to go. This cycle continues for a few seconds.

User Interface:

1. User Interface for Traffic Managers:

- **Sliders**: Sliders are used for manual adjustment of red, yellow, and green light durations.
- **Apply Button**: A button to apply the manual adjustments to the signal timings.
- **VBox**: A vertical box layout to organize the sliders and button.

2. Dashboard for City Officials:

- **LineChart**: A line chart to display traffic flow metrics over time.
- **TabPane**: A tab pane to switch between the traffic manager's control panel and the city officials' dashboard.

3. Real-Time Monitoring and Adjustment:

- **initTimeline()**: Initializes the timeline for signal animation.
- **adjustTimingsManually()**: Adjusts the signal timings based on the values from the sliders and updates the timeline accordingly.

TESTING:

```
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;
```

```
public class TrafficSignalOptimizerTest {
    @Test
    public void testOptimizeSignals() {
        // Create mock sensors and traffic data
        List<Sensor> sensors = new ArrayList<>();
        sensors.add(new Sensor("Intersection 1"));
        sensors.get(0).setTrafficData(new TrafficData(20, 30.0, 5, false));
        TrafficSignalOptimizer optimizer = new TrafficSignalOptimizer();
        optimizer.optimizeSignals(sensors);
        // Validate the optimization logic
        // Example: Check if the green light duration was increased for high traffic density
    }
}
```