**CCP Report: Efficient Traffic Management System Simulation Using Data Structures**

**Student Information**

* **Name:** Tofique Ahmed
* **Id:**69548
* **Course**: Data Structures and Algorithms
* **Program**: BS(SE)
* **Semester**: Fall 2024
* **Project Type**: Complex Computing Problem (CCP)
* **Mapped CLO**: CLO3
* **Due Date**: 30/06/2025

**1. Project Title**

**Efficient Traffic Management System Simulation Using Data Structures**

**2. Problem Statement**

This project aims to simulate a smart city traffic management system using key data structures such as **queues**, **priority queues**, and **graphs**. The system models vehicle movement through intersections, prioritizes emergency vehicles, and uses pathfinding algorithms to determine optimal routes. The simulation improves traffic flow and reduces congestion by dynamically managing queue lengths and traffic signals.

**3. Objectives**

* Implement realistic traffic queues at intersections.
* Prioritize emergency vehicle movement.
* Simulate pathfinding with Dijkstra’s algorithm.
* Provide a console-based simulation for visualization.
* Analyze and optimize vehicle clearance order and route efficiency.

**4. Tools and Technologies Used**

* **Programming Language**: Java
* **Data Structures**:
  + Queue – for normal vehicles
  + PriorityQueue – for emergency vehicle handling
  + HashMap and ArrayList – for graph modeling
* **Algorithms**: Dijkstra’s Algorithm
* **Environment**: Console-based Java IDE (e.g., IntelliJ, NetBeans)

**5. Design Overview**

**5.1. Class Structure**

**Vehicle**

Represents a vehicle with an ID and emergency status.

**Intersection**

Handles queues at a specific intersection:

* PriorityQueue for emergency vehicles
* Queue for normal vehicles
* Methods to add, get, and print vehicles

**RoadNetwork**

Models the city as a graph.

* addRoad() to connect intersections
* getShortestPath() implements Dijkstra’s algorithm

**5.2. Sample Graph (City Layout)**

less

CopyEdit

A --2--> B --3--> C

| ^

5 |

v 4

D --------

**6. Functionality and Simulation Flow**

**Step 1: Initialization**

* Create intersections A, B, C
* Add roads (edges) between them

**Step 2: Vehicle Input**

* Vehicles added to intersections with priority flags
* Example:
  + A-V1 (normal), A-V2 (emergency), etc.

**Step 3: Queue Processing**

* Emergency vehicles are dequeued first
* Remaining vehicles processed based on normal queue

**Step 4: Pathfinding**

* Uses Dijkstra’s algorithm to find the shortest route from A to C

**Step 5: Output**

* Vehicle clearance order printed
* Shortest path displayed

**7.Program**

package dsproject;

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.HashMap;

import java.util.LinkedList;

import java.util.List;

import java.util.Map;

import java.util.PriorityQueue;

import java.util.Queue;

import java.util.\*;

// ✅ Class name fixed to PascalCase

class Vehicle {

String id;

boolean isEmergency;

public Vehicle(String id, boolean isEmergency) {

this.id = id;

this.isEmergency = isEmergency;

}

@Override

public String toString() {

return id + (isEmergency ? " [EMERGENCY]" : "");

}

}

// ✅ Class name fixed

class Intersection {

String name;

Queue<Vehicle> normalQueue = new LinkedList<>();

PriorityQueue<Vehicle> emergencyQueue;

public Intersection(String name) {

this.name = name;

emergencyQueue = new PriorityQueue<>(Comparator.comparing(v -> !v.isEmergency));

}

public void addVehicle(Vehicle vehicle) {

if (vehicle.isEmergency) {

emergencyQueue.add(vehicle);

} else {

normalQueue.add(vehicle);

}

}

public Vehicle getNextVehicle() {

return !emergencyQueue.isEmpty() ? emergencyQueue.poll() : normalQueue.poll();

}

public boolean isEmpty() {

return emergencyQueue.isEmpty() && normalQueue.isEmpty();

}

public void printQueues() {

System.out.println("Intersection " + name + " Queues:");

System.out.println("Emergency Queue: " + emergencyQueue);

System.out.println("Normal Queue: " + normalQueue);

}

}

// ✅ Class name fixed

class RoadNetwork {

Map<String, List<Road>> adjacencyMap = new HashMap<>();

public void addRoad(String from, String to, int weight) {

adjacencyMap.computeIfAbsent(from, k -> new ArrayList<>()).add(new Road(to, weight));

}

public List<String> getShortestPath(String start, String end) {

Map<String, Integer> distances = new HashMap<>();

Map<String, String> prev = new HashMap<>();

PriorityQueue<Road> pq = new PriorityQueue<>(Comparator.comparingInt(r -> r.weight));

for (String node : adjacencyMap.keySet()) distances.put(node, Integer.MAX\_VALUE);

distances.put(start, 0);

pq.add(new Road(start, 0));

while (!pq.isEmpty()) {

Road current = pq.poll();

if (!adjacencyMap.containsKey(current.destination)) continue;

for (Road neighbor : adjacencyMap.get(current.destination)) {

int newDist = distances.get(current.destination) + neighbor.weight;

if (newDist < distances.getOrDefault(neighbor.destination, Integer.MAX\_VALUE)) {

distances.put(neighbor.destination, newDist);

prev.put(neighbor.destination, current.destination);

pq.add(new Road(neighbor.destination, newDist));

}

}

}

List<String> path = new ArrayList<>();

for (String at = end; at != null; at = prev.get(at)) {

path.add(at);

}

Collections.reverse(path);

return path;

}

}

// ✅ Class name fixed

class Road {

String destination;

int weight;

public Road(String destination, int weight) {

this.destination = destination;

this.weight = weight;

}

@Override

public String toString() {

return destination + " (" + weight + ")";

}

}

// ✅ MAIN CLASSpublic class Simulation {

public class Simulation {

public static void main(String[] args) {

RoadNetwork roadNetwork = new RoadNetwork();

roadNetwork.addRoad("A", "B", 2);

roadNetwork.addRoad("B", "C", 3);

roadNetwork.addRoad("A", "D", 5);

roadNetwork.addRoad("D", "C", 4);

// Create multiple intersections

Intersection intersectionA = new Intersection("A");

Intersection intersectionB = new Intersection("B");

Intersection intersectionC = new Intersection("C");

// Add vehicles to intersection A

intersectionA.addVehicle(new Vehicle("A-V1", false));

intersectionA.addVehicle(new Vehicle("A-V2", true));

intersectionA.addVehicle(new Vehicle("A-V3", false));

// Add vehicles to intersection B

intersectionB.addVehicle(new Vehicle("B-V1", false));

intersectionB.addVehicle(new Vehicle("B-V2", true));

// Add vehicles to intersection C

intersectionC.addVehicle(new Vehicle("C-V1", true));

intersectionC.addVehicle(new Vehicle("C-V2", false));

// Print and process intersection A

System.out.println("===== Intersection A =====");

intersectionA.printQueues();

System.out.println("Vehicle Clearance Order at A:");

while (!intersectionA.isEmpty()) {

System.out.println(intersectionA.getNextVehicle());

}

// Print and process intersection B

System.out.println("\n===== Intersection B =====");

intersectionB.printQueues();

System.out.println("Vehicle Clearance Order at B:");

while (!intersectionB.isEmpty()) {

System.out.println(intersectionB.getNextVehicle());

}

// Print and process intersection C

System.out.println("\n===== Intersection C =====");

intersectionC.printQueues();

System.out.println("Vehicle Clearance Order at C:");

while (!intersectionC.isEmpty()) {

System.out.println(intersectionC.getNextVehicle());

}

// Show shortest path from A to C

System.out.println("\n===== Pathfinding =====");

System.out.println("Shortest Path from A to C:");

System.out.println(roadNetwork.getShortestPath("A", "C"));

}

}

**8.OutPut**

**===== Intersection A =====**

**Intersection A Queues:**

**Emergency Queue: [A-V2 [EMERGENCY]]**

**Normal Queue: [A-V1, A-V3]**

**Vehicle Clearance Order at A:**

**A-V2 [EMERGENCY]**

**A-V1**

**A-V3**

**===== Intersection B =====**

**Intersection B Queues:**

**Emergency Queue: [B-V2 [EMERGENCY]]**

**Normal Queue: [B-V1]**

**Vehicle Clearance Order at B:**

**B-V2 [EMERGENCY]**

**B-V1**

**===== Intersection C =====**

**Intersection C Queues:**

**Emergency Queue: [C-V1 [EMERGENCY]]**

**Normal Queue: [C-V2]**

**Vehicle Clearance Order at C:**

**C-V1 [EMERGENCY]**

**C-V2**

**===== Pathfinding =====**

**Shortest Path from A to C:**

**[A, B, C]**

**9. Expected Outcomes**

* Realistic simulation of traffic queues
* Efficient prioritization of emergency vehicles
* Accurate shortest path calculation
* Performance insights into vehicle clearance patterns
* Strong understanding of practical data structures

**10. Challenges Faced**

* Designing custom priority logic for queues
* Avoiding starvation of normal vehicles
* Integrating Dijkstra’s algorithm dynamically
* Ensuring fairness and response time constraints

**11. Future Enhancements**

* GUI integration with JavaFX for live traffic visualization
* Adaptive signal timing using real-time data
* Simulation of roadblocks and accident handling
* Integration with GPS and real map data

**12. Conclusion**  
This simulation stands as a groundbreaking representation of how Data Structures and Algorithms can revolutionize urban traffic control. By ingeniously integrating queues for vehicle flow, graphs for city-wide road mapping, and smart prioritization for emergency routing, the system offers a dynamic, real-time traffic management solution. It not only mirrors real-world challenges but also demonstrates how computational logic can dramatically reduce congestion, enhance public safety, and pave the way for smarter, more sustainable cities of the future.