|  |
| --- |
| import java.util.\*;  import java.util.concurrent.atomic.AtomicInteger;  public class TrafficSimulation {  static class Road {  String id;  Intersection from;  Intersection to;  int travelTime;  public Road(String id, Intersection from, Intersection to, int travelTime) {  this.id = id;  this.from = from;  this.to = to;  this.travelTime = travelTime;  }  @Override  public String toString() {  return from.id + "->" + to.id;  }  }  static class Intersection {  String id;  Map<Road, PriorityQueue<Vehicle>> incomingQueues = new HashMap<>();  TrafficLight trafficLight;  public Intersection(String id) {  this.id = id;  }  public void initializeTrafficLight(List<Road> phases) {  this.trafficLight = new TrafficLight(this, phases);  }  }  static class Vehicle implements Comparable<Vehicle> {  int id;  String type; // "normal" or "emergency"  int startTime;  Queue<Road> remainingPath;  Road currentRoad;  int remainingTime;  int queueArrivalTime;  int totalWaitingTime;  public Vehicle(int id, String type, int startTime, Queue<Road> path) {  this.id = id;  this.type = type;  this.startTime = startTime;  this.remainingPath = new LinkedList<>(path);  }  @Override  public int compareTo(Vehicle other) {  if ("emergency".equals(this.type) && !"emergency".equals(other.type)) {  return -1;  } else if (!"emergency".equals(this.type) && "emergency".equals(other.type)) {  return 1;  }  return Integer.compare(this.queueArrivalTime, other.queueArrivalTime);  }  }  static class TrafficLight {  Intersection intersection;  List<Road> phases;  int currentPhaseIndex = 0;  int currentGreenTime = 0;  int assignedGreenTime;  public TrafficLight(Intersection intersection, List<Road> phases) {  this.intersection = intersection;  this.phases = phases;  this.assignedGreenTime = 10; // Base green time  }  public void update(int currentTime, int baseGreen, int factor, int maxGreen) {  currentGreenTime++;  if (currentGreenTime >= assignedGreenTime) {  switchToNextPhase(currentTime, baseGreen, factor, maxGreen);  }  }  private void switchToNextPhase(int currentTime, int baseGreen, int factor, int maxGreen) {  currentPhaseIndex = (currentPhaseIndex + 1) % phases.size();  currentGreenTime = 0;  Road nextRoad = phases.get(currentPhaseIndex);  int queueSize = intersection.incomingQueues.get(nextRoad).size();  assignedGreenTime = Math.min(maxGreen, baseGreen + factor \* queueSize);  }  }  public static void main(String[] args) {  // Create intersections  Intersection A = new Intersection("A");  Intersection B = new Intersection("B");  Intersection C = new Intersection("C");  Intersection D = new Intersection("D");  // Create roads  Road AB = new Road("A->B", A, B, 2);  Road BA = new Road("B->A", B, A, 2);  Road BC = new Road("B->C", B, C, 3);  Road CB = new Road("C->B", C, B, 3);  Road AD = new Road("A->D", A, D, 5);  Road DA = new Road("D->A", D, A, 5);  Road DC = new Road("D->C", D, C, 4);  Road CD = new Road("C->D", C, D, 4);  // Set up incoming queues  A.incomingQueues.put(BA, new PriorityQueue<>());  A.incomingQueues.put(DA, new PriorityQueue<>());  B.incomingQueues.put(AB, new PriorityQueue<>());  B.incomingQueues.put(CB, new PriorityQueue<>());  C.incomingQueues.put(BC, new PriorityQueue<>());  C.incomingQueues.put(DC, new PriorityQueue<>());  D.incomingQueues.put(AD, new PriorityQueue<>());  D.incomingQueues.put(CD, new PriorityQueue<>());  // Set up traffic lights  A.initializeTrafficLight(Arrays.asList(BA, DA));  B.initializeTrafficLight(Arrays.asList(AB, CB));  C.initializeTrafficLight(Arrays.asList(BC, DC));  D.initializeTrafficLight(Arrays.asList(AD, CD));  // Traffic light parameters  int BASE\_GREEN = 10;  int GREEN\_FACTOR = 1;  int MAX\_GREEN = 30;  int VEHICLE\_RATE = 10; // New vehicles per 100 time steps  // Simulation setup  List<Intersection> intersections = Arrays.asList(A, B, C, D);  List<Road> roads = Arrays.asList(AB, BA, BC, CB, AD, DA, DC, CD);  List<Vehicle> vehiclesOnRoad = new ArrayList<>();  List<Vehicle> completedVehicles = new ArrayList<>();  AtomicInteger vehicleId = new AtomicInteger(1);  Random rand = new Random();  // Predefined paths for demonstration  Map<String, Queue<Road>> pathTemplates = new HashMap<>();  pathTemplates.put("A->C", new LinkedList<>(Arrays.asList(AB, BC)));  pathTemplates.put("A->D", new LinkedList<>(Arrays.asList(AD)));  pathTemplates.put("B->D", new LinkedList<>(Arrays.asList(BC, CD)));  pathTemplates.put("C->A", new LinkedList<>(Arrays.asList(CB, BA)));  pathTemplates.put("D->B", new LinkedList<>(Arrays.asList(DC, CB)));  // Simulation loop  for (int time = 0; time < 1000; time++) {  // Generate new vehicles  if (time % VEHICLE\_RATE == 0) {  for (Intersection inter : intersections) {  if (rand.nextDouble() < 0.3) { // 30% chance per eligible timestep  String type = rand.nextDouble() < 0.05 ? "emergency" : "normal";  List<String> keys = new ArrayList<>(pathTemplates.keySet());  String randomKey = keys.get(rand.nextInt(keys.size()));  Queue<Road> path = new LinkedList<>(pathTemplates.get(randomKey));    Vehicle v = new Vehicle(vehicleId.getAndIncrement(), type, time, path);  Road firstRoad = v.remainingPath.poll();  v.currentRoad = firstRoad;  v.remainingTime = firstRoad.travelTime;  vehiclesOnRoad.add(v);  }  }  }  // Update vehicles on roads  Iterator<Vehicle> roadIterator = vehiclesOnRoad.iterator();  while (roadIterator.hasNext()) {  Vehicle v = roadIterator.next();  v.remainingTime--;  if (v.remainingTime == 0) {  roadIterator.remove();  if (v.remainingPath.isEmpty()) {  completedVehicles.add(v);  } else {  v.queueArrivalTime = time;  v.currentRoad.to.incomingQueues.get(v.currentRoad).add(v);  }  }  }  // Process intersections  for (Intersection inter : intersections) {  // Update traffic light  inter.trafficLight.update(time, BASE\_GREEN, GREEN\_FACTOR, MAX\_GREEN);    // Process current green lane  Road greenRoad = inter.trafficLight.phases.get(inter.trafficLight.currentPhaseIndex);  PriorityQueue<Vehicle> queue = inter.incomingQueues.get(greenRoad);  int processed = 0;  int maxProcess = Math.min(queue.size(), 2); // Process up to 2 vehicles per time step    while (processed < maxProcess && !queue.isEmpty()) {  Vehicle v = queue.poll();  v.totalWaitingTime += (time - v.queueArrivalTime);    Road nextRoad = v.remainingPath.poll();  v.currentRoad = nextRoad;  v.remainingTime = nextRoad.travelTime;  vehiclesOnRoad.add(v);  processed++;  }  }  // Print status periodically  if (time % 100 == 0) {  System.out.println("\nTime: " + time + "s");  System.out.println("Vehicles on road: " + vehiclesOnRoad.size());  System.out.println("Completed vehicles: " + completedVehicles.size());  System.out.println("Intersection Queue Lengths:");  for (Intersection inter : intersections) {  System.out.print(inter.id + ": ");  for (Map.Entry<Road, PriorityQueue<Vehicle>> entry : inter.incomingQueues.entrySet()) {  System.out.print(entry.getKey() + "=" + entry.getValue().size() + " ");  }  System.out.println();  }  }  }  // Generate final report  System.out.println("\n===== Simulation Results =====");  System.out.println("Total vehicles: " + (vehicleId.get() - 1));  System.out.println("Completed vehicles: " + completedVehicles.size());    if (!completedVehicles.isEmpty()) {  int totalTravelTime = 0;  int totalWaitingTime = 0;  for (Vehicle v : completedVehicles) {  totalTravelTime += (v.startTime + v.totalWaitingTime + v.currentRoad.travelTime);  totalWaitingTime += v.totalWaitingTime;  }  System.out.println("Average travel time: " + totalTravelTime / completedVehicles.size() + "s");  System.out.println("Average waiting time: " + totalWaitingTime / completedVehicles.size() + "s");  }    System.out.println("Current vehicles on road: " + vehiclesOnRoad.size());  }  } |