

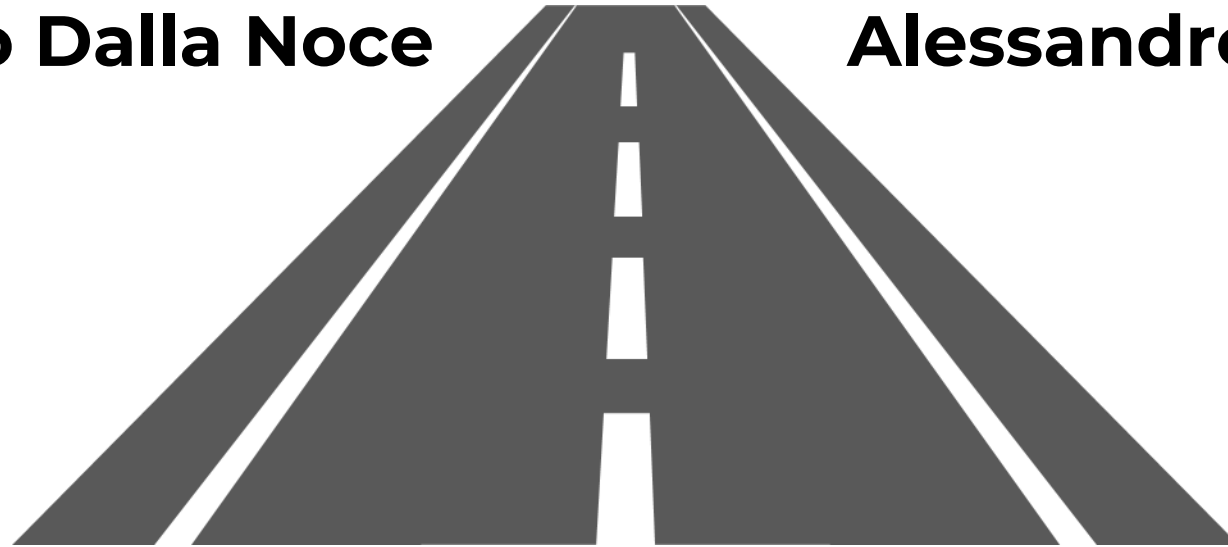


# Highway traffic flow with parallel cellular automata

Computational Models for Complex Systems  
project a.y. 2021/2022

**Niko Dalla Noce**

**Alessandro Ristori**



# Presentation Roadmap

## Traffic Automata

How to integrate cellular automata within a traffic context

01

## Implementation

How the approach has been implemented

02

## Results

How the system performed, with a case study under consideration

03

## Demo

We will show three different scenarios

04



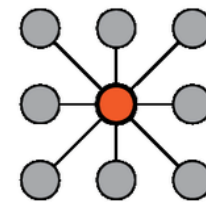
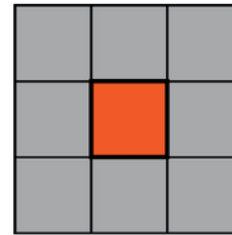
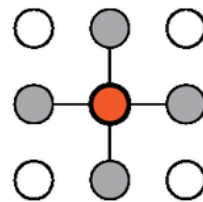
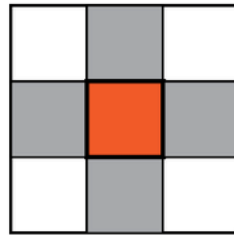
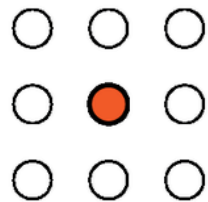
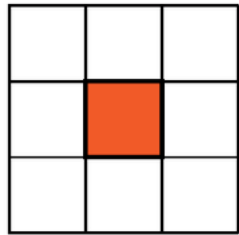
# Traffic Automata



# Cellular Automata, a quick introduction

Recall what the professor said during lesson:

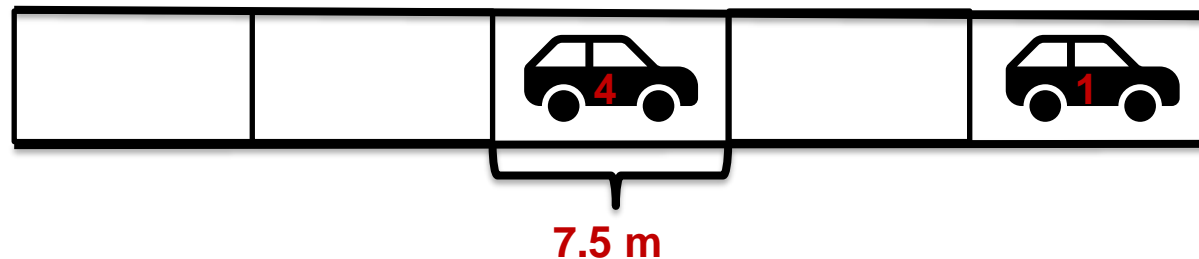
- Cellular Automata (CA) allows describing **1D, 2D or 3D environments**;
- The environment consists of a **matrix of cells**;
- Each cell has its **own state** that can evolve by means of **rules**;
- Each cell knows its **neighborhood**.



# Cellular automata in highway traffic

We can implement cellular automata in a **highway traffic environment** by following simple rules:

- Each **cell corresponds to 7.5 meters** and it's occupied by only one vehicle;
- The interval between each **time step is 1 second**;
- The **vehicle's speed is an integer** that tells how many cells the **vehicle can move forward**;
- The cell neighborhood is **determined by the maximum allowed speed**;
- The **rules depends on the scenario** you want to implement (this will be seen in a few slides).



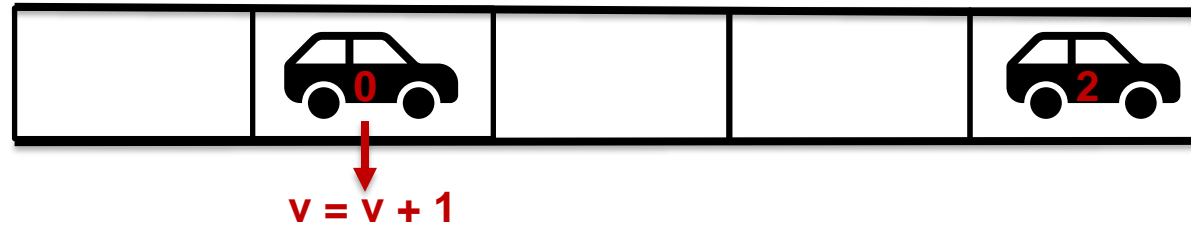
The **maximum allowed speed is 5**, which corresponds to

$$37.5 \text{ m/s} = 135 \text{ km/h}$$

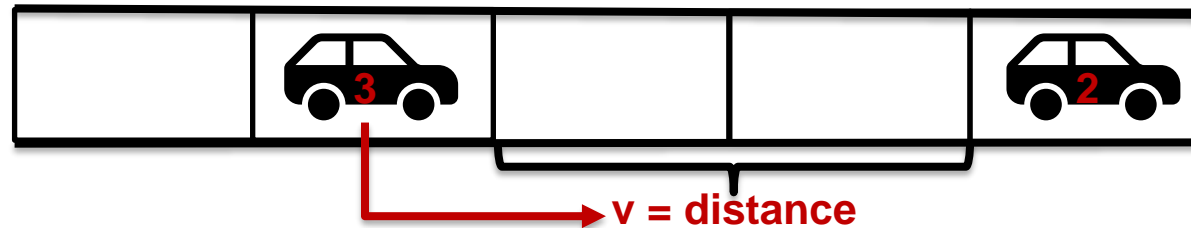
# One lane traffic model

The **Nagel-Schreckenberg model** defines the rules for the speed update.

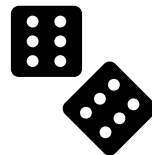
**Acceleration:** increase the vehicle's speed if the distance to cover is bigger than its actual speed, and it has not reached the maximum.



**Deceleration (due to other cars):** if the distance between the vehicle and the one that precedes it is lower than its speed, then set vehicle speed to such distance.



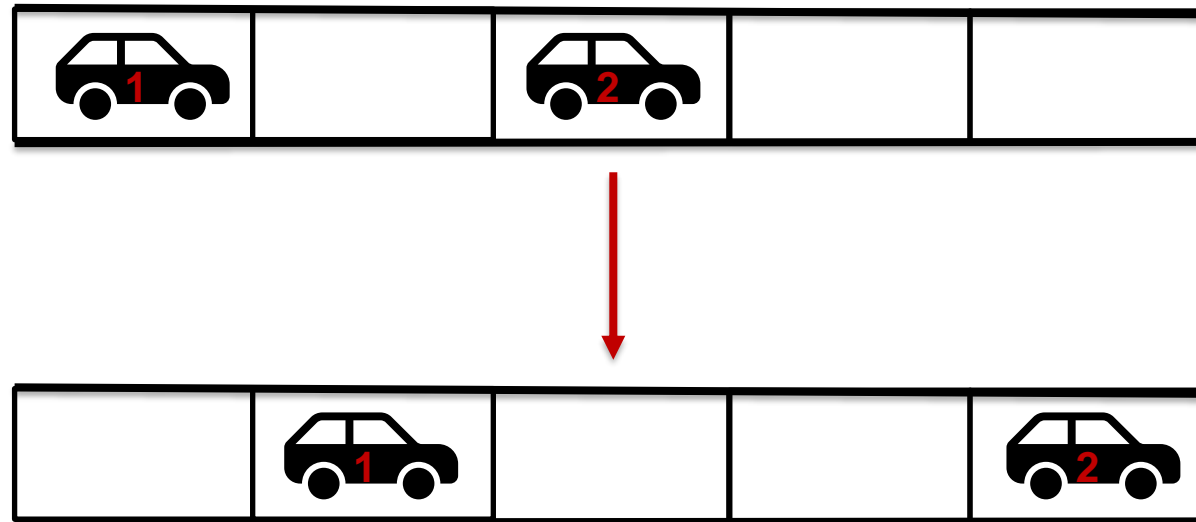
**Random deceleration:** if the vehicle's speed is higher than 0, then reduce its speed with probability  $p$ .



$$\text{random}(0, 1) \leq p \longrightarrow v = v - 1$$

## Move the cars

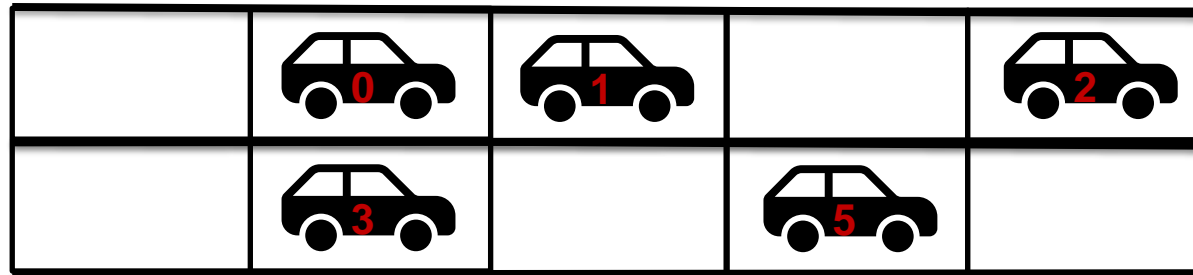
After the speed update, it's time to do the same with the road matrix. Each **vehicle “moves” forward** as many cells as its speed.



The **single lane model it's extremely easy**, what if we have more lanes?

## Two (or more) lane traffic model

The **Nagel-Schreckenberg** model was updated in a following paper considering a **scenario in which vehicles can change lane**. It was proposed only for two-lanes roads, **we decided to extend it to more lanes**.

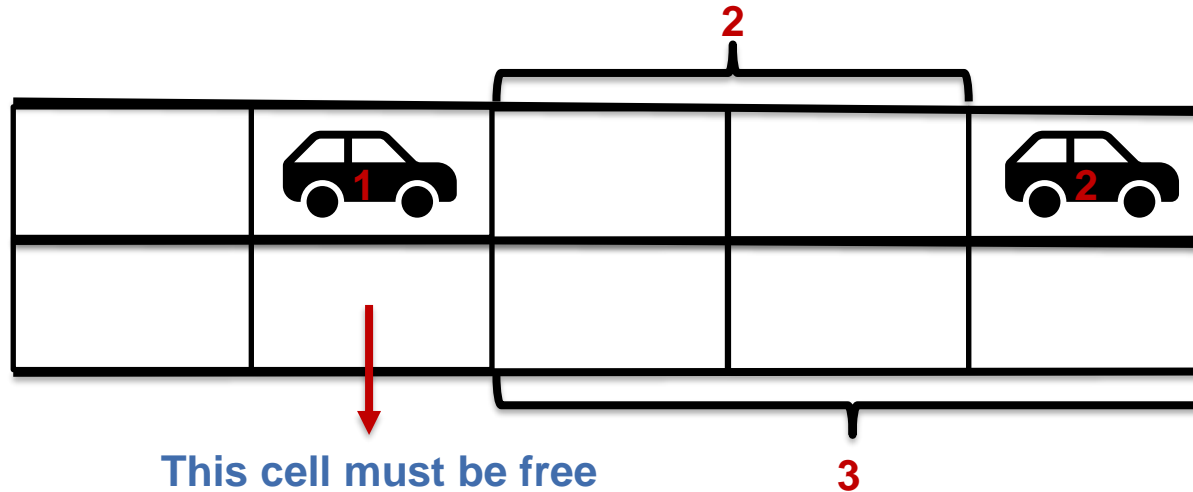


The **speed and road updates still hold** as before, then, when does a vehicle change lane?

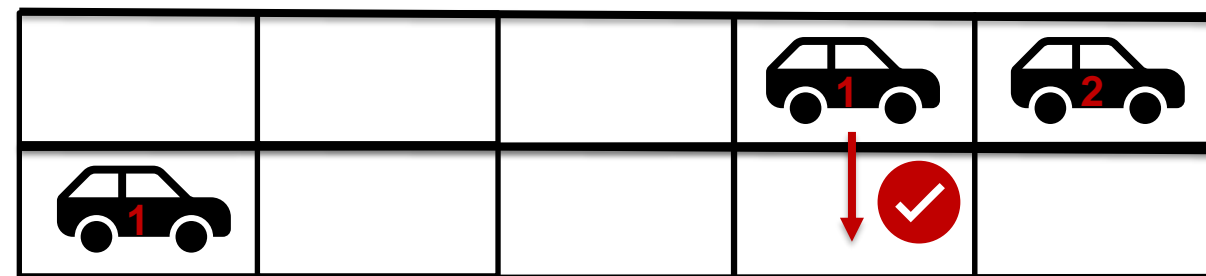
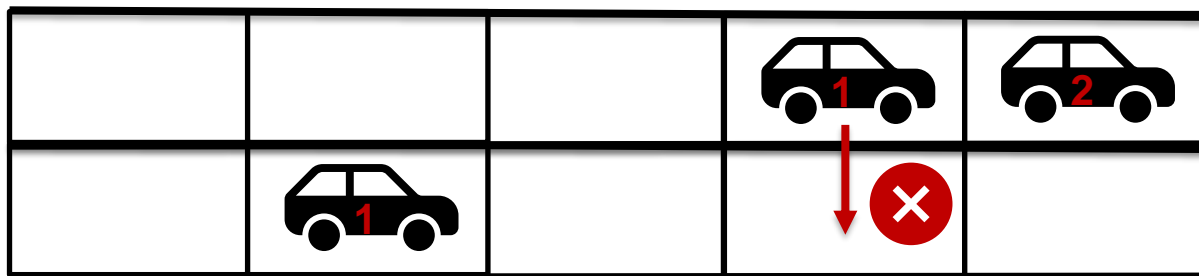


# Rules for changing lane

**Moving forward is convenient:** the drivable distance in the other lane should be higher than the one in the current lane.

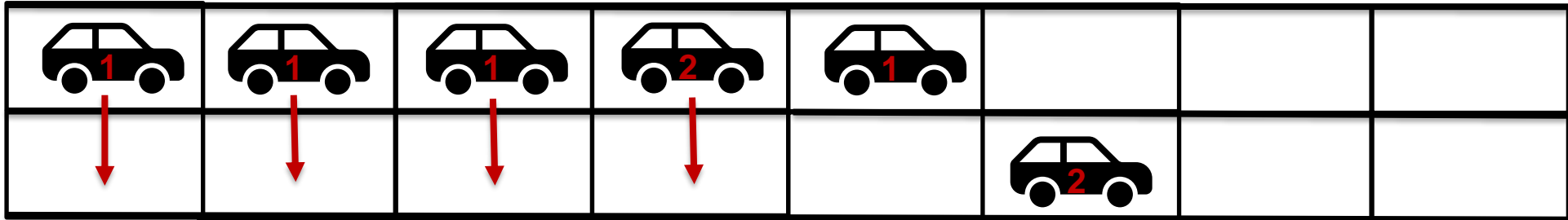


**No vehicle is coming from behind in the other lane:** the distance between the adjacent cell and the following vehicle is at least the maximum allowed speed, as an example let's suppose that the latter is 2.



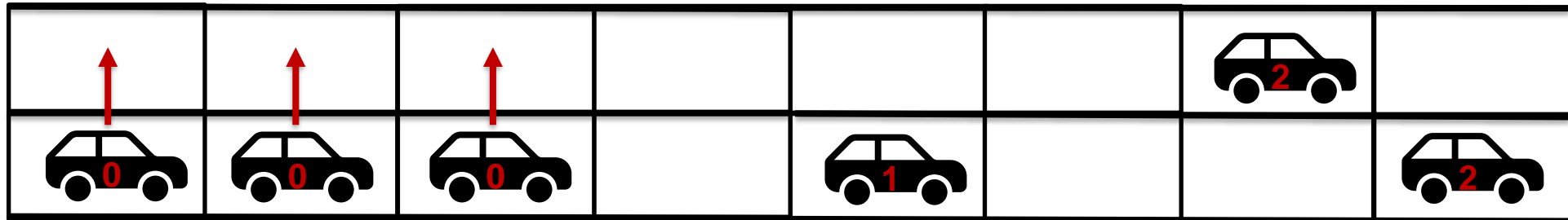
# Issue: the ping pong movement

T

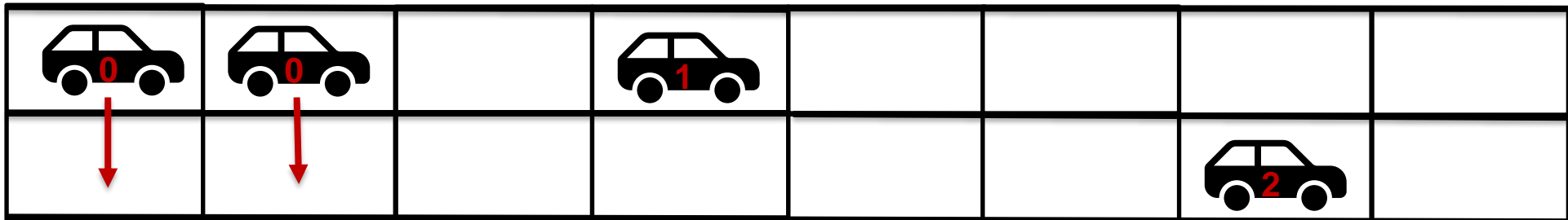


Suppose that  $v_{\max} = 2$

T + 1

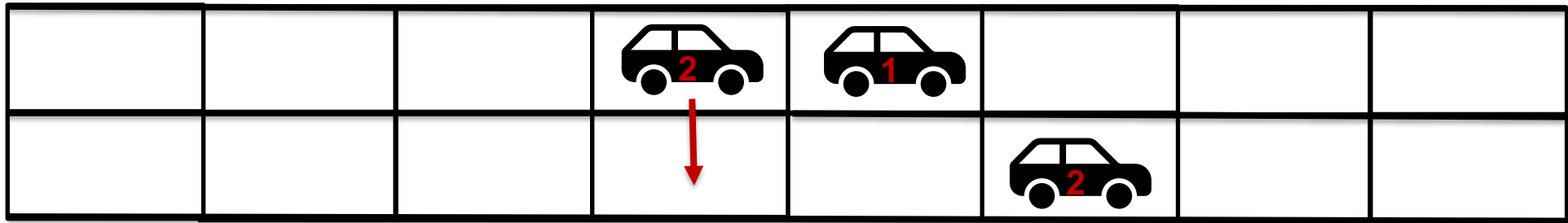


T + 2

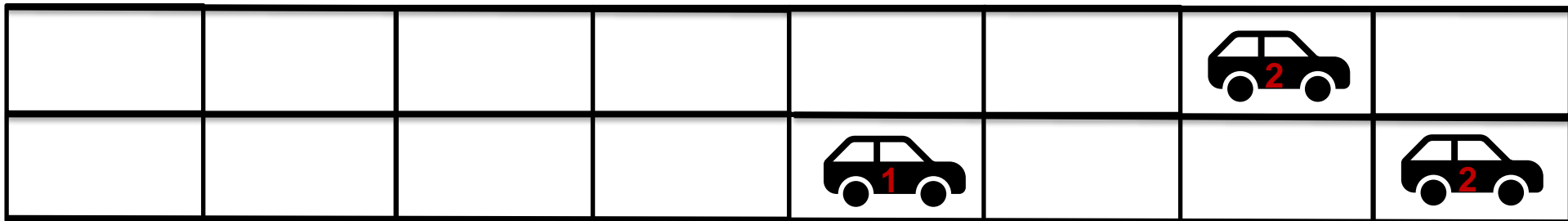


# Solution: randomness

If a vehicle can move in the other lane, then change its position with probability  $pChangeLane$ .



$\text{random}(0, 1) \leq pChangeLane$



# Summary of traffic automata

## Schedule and its rules

### Change lane

- **Moving forward is convenient;**
- **No vehicle is coming from behind in the other lane;**
- **Move with probability  $p_{\text{ChangeLane}}$**

---

### Update speeds

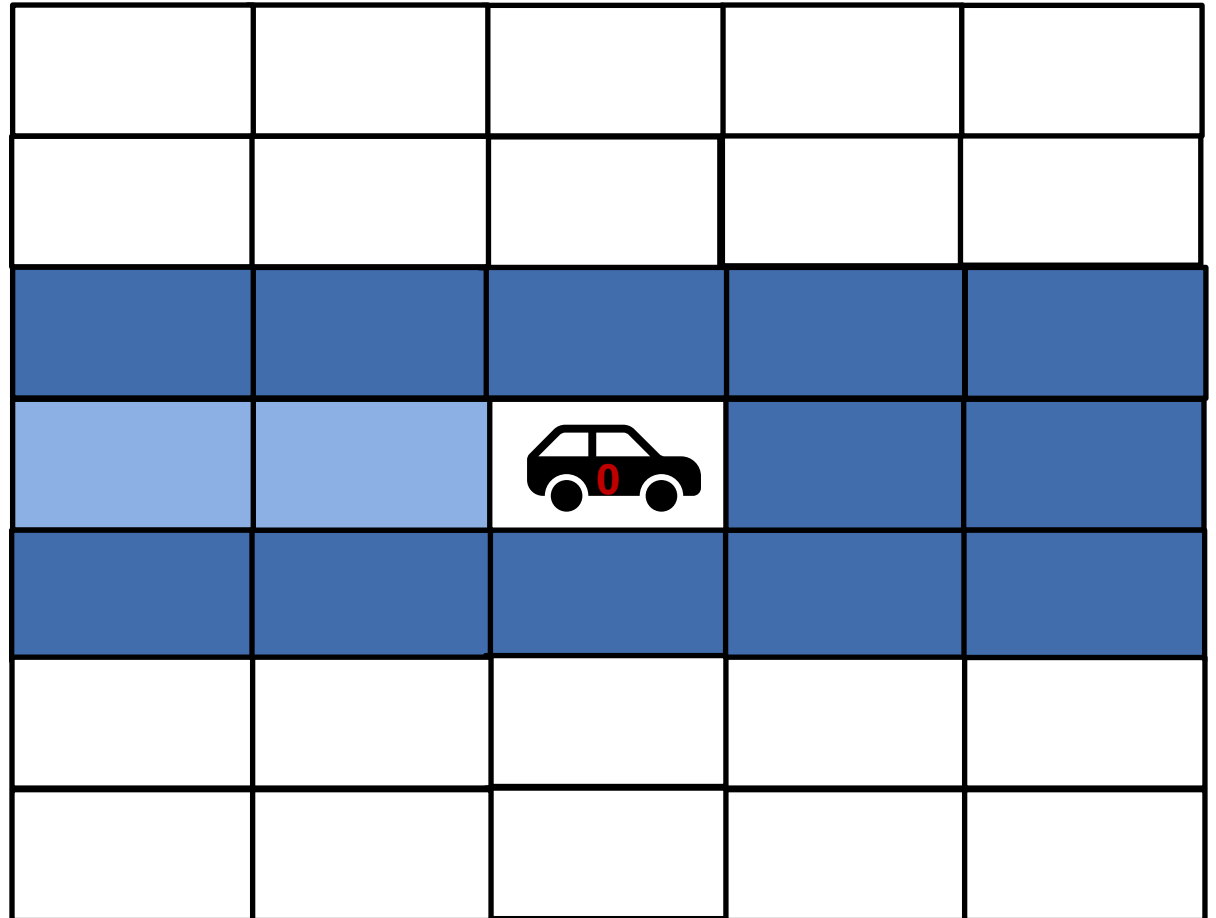
- **Acceleration;**
- **Deceleration (due to other cars);**
- **Random deceleration.**

---

### Update road

- **Move cars.**

## Neighborhood



Neighborhood with  $v_{\text{max}} = 2$



# Implementation



# Rules set

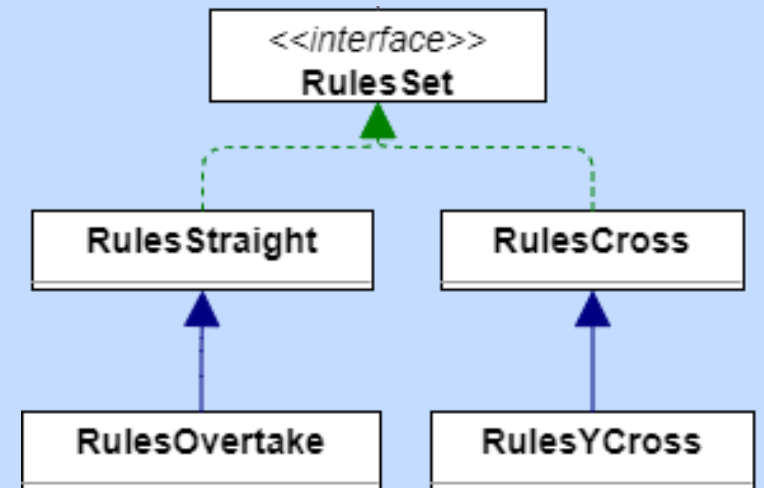
The **rules define how a vehicle moves** within a road:

- We implemented two different kinds of roads: **Straight** and **Ycross**;
- Each kind of road should implement its own rule set;
- Generally, a **rule set in a straight road defines the neighborhood** of a **cellular automata**.
- A **rule set of a cross** defines how the vehicles move from an **incoming road to an outgoing road**.

Road



Rule set



# Rules: code snippets

```
public class RulesOvertake extends RulesStraight {

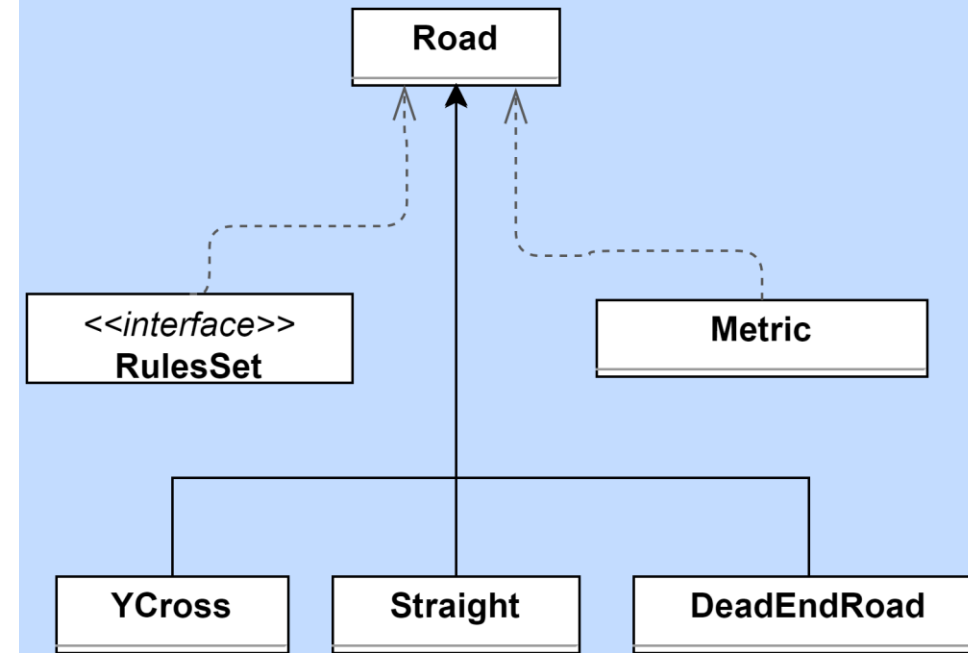
    public RulesOvertake(double pDecreaseSpeed, int direction,
                        double pChangeLane) {
        super(pDecreaseSpeed, direction);
        this.pChangeLane = pChangeLane;
    }

    @Override
    public void apply(Straight road) {
        //Change lanes if possible
        if (road.nLanes() > 1) {
            changeLane(road);
        }
        //Update the cars' speeds and the road state
        super.apply(road);
    }
}
```

```
public class RulesYCross extends RulesCross {
    @Override
    public void apply(YCross road) {
        Road outgoing = road.nextRoad();
        BlockingDeque<Vehicle> queue = road.vehiclesQueue();
        var accepted = true;
        while (accepted && queue.size() > 0) {
            accepted = outgoing.acceptVehicle(queue.element());
            if (accepted) queue.removeFirst();
        }
    }
}
```

# Roads

- Any “concrete” new **road** should extend the abstract class **Road**. We did it for the **Ycross**, **Straight** and **DeadEndRoad** classes.
- The **DeadEndRoad** can be set as an outgoing road to indicate that a road hasn’t any next road.
- Optionally, a **Road** can exploit some **Metrics** to monitor the **behavior** of the vehicles inside of it.





# Roads: code snippets

```
public abstract class Road {  
    public Road(Road outgoing, RulesSet rules) {  
        this.outgoing = outgoing;  
        this.rules = rules;  
        this.roadId = seq;  
        seq++;  
    }  
  
    public void runStep() {  
        rules.apply(this);  
    }  
}
```

```
public class Straight extends Road {  
  
    public Straight(int lanes, int length, double density, int maxSpeed,  
RulesSet<Straight> rules, Road outgoing, List<Metric<Straight, Double>> metrics) {  
        super(outgoing, rules);  
        this.lanes = lanes;  
        this.length = length;  
        this.density = density;  
        this.maxSpeed = maxSpeed;  
        initializeMetrics(metrics);  
        buildRoad();  
    }  
  
    @Override  
    public void computeMetrics(int step) {  
        for (var metric : metrics.keySet()) {  
            var ris = metric.compute(this, step);  
            ris.ifPresent(aDouble -> metrics.get(metric).add(aDouble));  
        }  
    }  
}
```

# Roads: code snippets

```
public class YCross extends Road {  
  
    private BlockingDeque<Vehicle> queue;  
    private final int capacity;  
  
    public YCross(int capacity, Road outgoing, RulesSet<YCross> rules) {  
        super(outgoing, rules);  
        this.capacity = capacity;  
        this.queue = new LinkedBlockingDeque<>(capacity);  
    }  
}
```

@Override

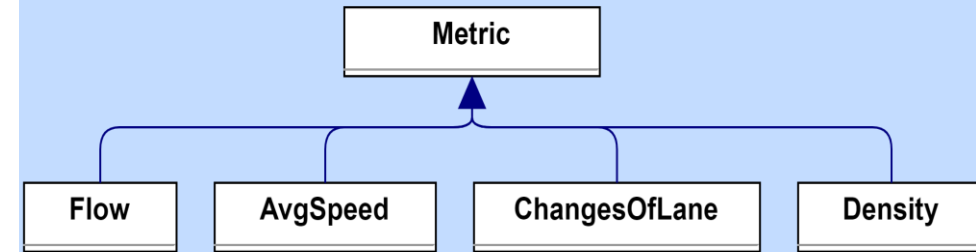
```
public boolean acceptVehicle(Vehicle vehicle) {  
    boolean acceptedVehicle = queue.offer(vehicle);  
    if (acceptedVehicle) {  
        vehicle.setSpeed(1);  
    }  
    return acceptedVehicle;  
}
```

@Override

```
public void runStep() {  
    var accepted = true;  
    while (accepted && queue.size()>0){  
        accepted = outgoing.acceptVehicle(queue.element());  
        if (accepted) queue.removeFirst();  
    }  
}
```

# Metrics

- It is possible to **create new metrics by simply extending the Metric class**.
- For our purposes, we implemented the **traffic flow metric**, the **average speed** of the vehicles, the total number of **lane changes** and the **density** of a Road.



# Metrics: code snippets

```
public abstract class Metric<T extends Road, K extends Number> {
    protected int intervalOfRecording;

    protected Metric(int intervalOfRecording){
        this.intervalOfRecording = intervalOfRecording;
    }

    public Optional<K> compute(T road, int step){
        Optional<K> metric = Optional.empty();
        if (step % intervalOfRecording == 0){
            metric = Optional.of(getRecord(road));
        }
        return metric;
    }

    public abstract K getRecord(T road);
}
```

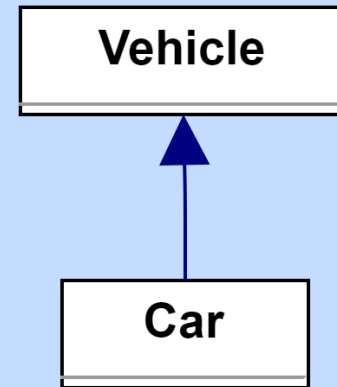
```
public class AvgSpeed extends Metric<Straight, Double> {

    public AvgSpeed(int intervalOfRecording) {
        super(intervalOfRecording);
    }
    @Override
    public Double getRecord(Straight road) {
        return road.averageSpeed();
    }
}
```

```
public class Density extends Metric<Straight, Double> {
    public Density(int intervalOfRecording) {
        super(intervalOfRecording);
    }
    @Override
    public Double getRecord(Straight road) {
        return road.density();
    }
}
```

# Vehicle

- Every **Class** that **extends Vehicle** can be **placed inside** a **road**;
- It can be useful if there is the necessity to **simulate traffic** with vehicles that are larger than a car, for example **bus** or **trucks**.

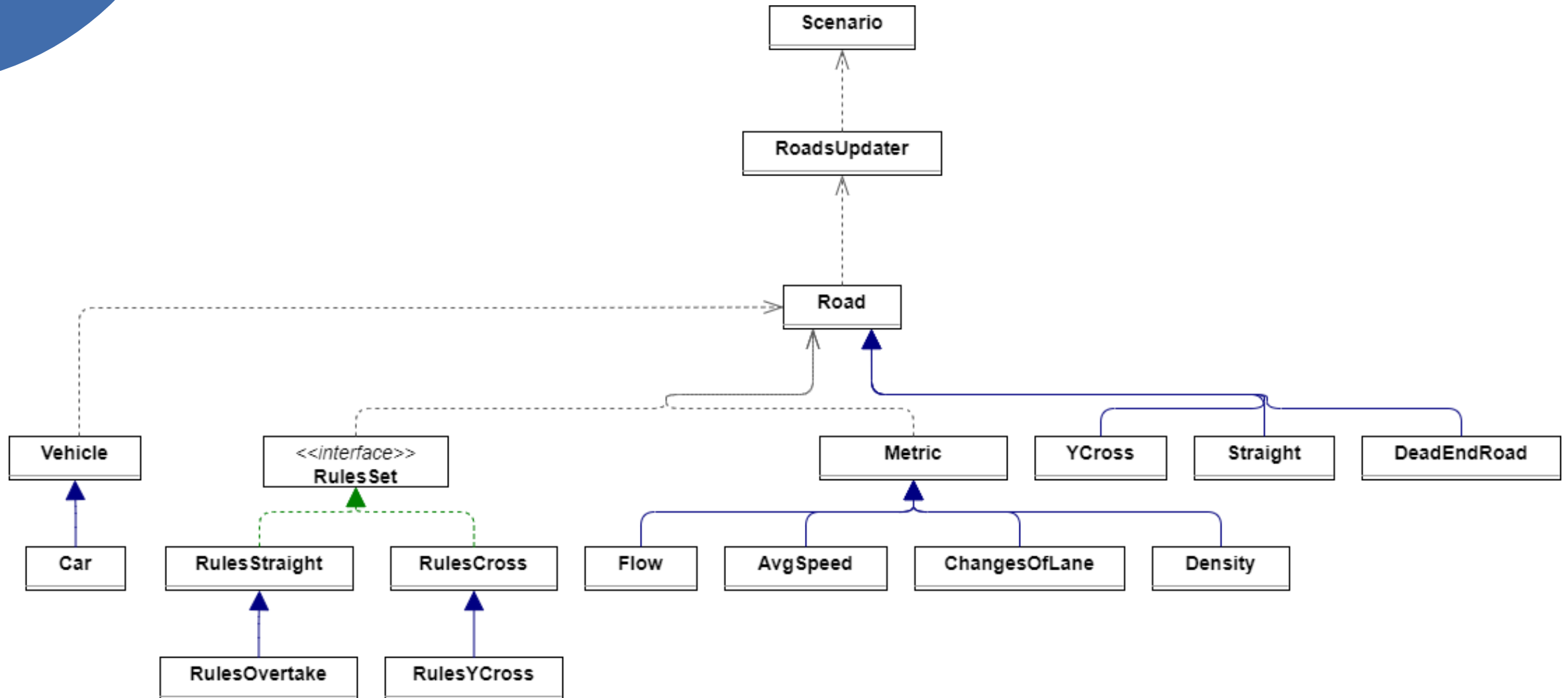


# Vehicle: code snippets

```
public abstract class Vehicle {  
    private final int ID;  
    private static int seq;  
    private int speed;  
    private final int size;  
  
    public Vehicle(int speed, int size) {  
        this.ID = seq++;  
        this.speed = speed;  
        this.size = size;  
    }  
  
    public int getSpeed() {  
        return speed;  
    }  
  
    public void setSpeed(int speed) {  
        this.speed = speed;  
    }  
  
    public int getSize() {  
        return size;  
    }  
}
```

```
public class Car extends Vehicle {  
  
    public Car() {  
        super(1,1);  
    }  
  
    public Car(int speed) {  
        super(speed,1);  
    }  
}
```

# Class diagramm



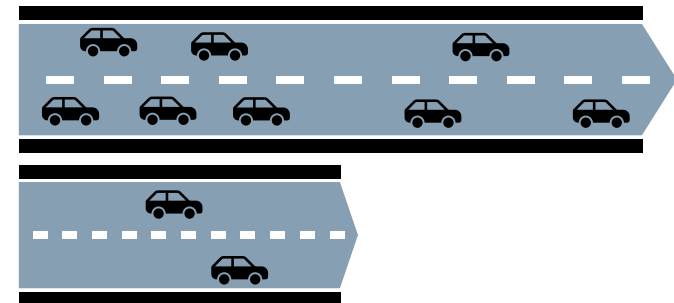
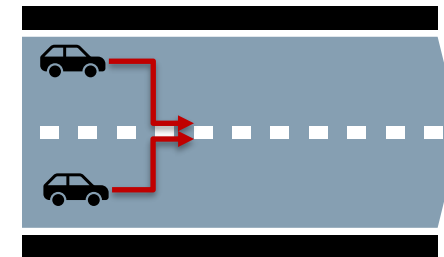
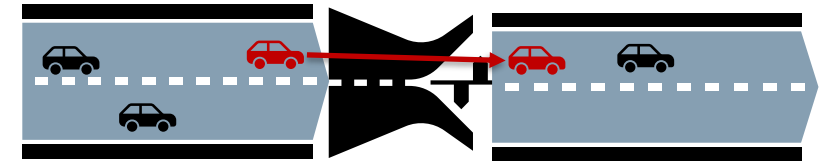
# Multithreading





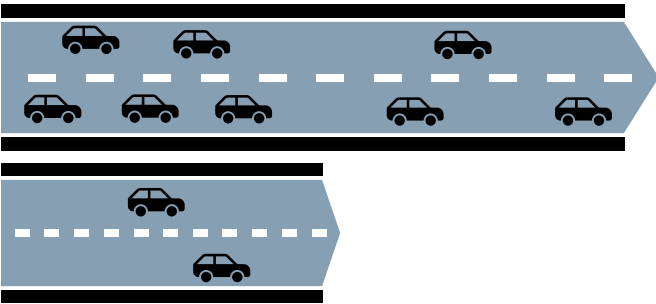
# Managing the concurrency: **problems**

- The **transfer of vehicles from one road to another** is the **most critical** operation;
- It could happen that a **thread managing a road** allows a **vehicle to move** towards the next road while **another thread updates the status** of that road.
- **Threads having less roads/vehicle to update** could run **faster than the others**, causing desynchronized updates at different time steps.



# Managing the concurrency: solutions

- Every time a **thread updates** the **roads** assigned, it **reaches** a **barrier** and **waits** for the **other threads** to **join**.



```
public class RoadsUpdater implements Runnable{
    private final List<Road> roads;
    private static int seq;
    private final int ID;

    public void setFinished(boolean finished) {
        this.finished = finished;
    }

    @Override
    public void run() {
        while (!finished) {
            for (var road : roads) {
                road.runStep();
            }
            try {
                barrier.await();
            } catch (InterruptedException | BrokenBarrierException e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

# Managing the concurrency: solutions

- Any straight should be followed by a cross;
- A thread updates a Cross and then a Straight. It

must follow this order.



- Also, following this “structural” rule, **we avoid some useless synchronizations between threads.**

```
public class YCross extends Road {
    /*some code*/
    @Override
    public boolean acceptVehicle(Vehicle vehicle) {
        boolean acceptedVehicle = queue.offer(vehicle);
        if (acceptedVehicle) {
            vehicle.setSpeed(1);
        }
        return acceptedVehicle;
    }
}
```

```
public class Straight extends Road {
    /*some code*/
    @Override
    public boolean acceptVehicle(Vehicle vehicle) {
        List<Integer> freeLanes = new ArrayList<>();
        for (int i = 0; i < lanes; i++) {
            if (!road[i][0]) {
                freeLanes.add(i);
            }
        }
        if (freeLanes.size() > 0) {
            Collections.shuffle(freeLanes);
            int chosenLane = freeLanes.get(0);
            road[chosenLane][0] = true;
            vehiclePositions.put(vehicle, new Position(chosenLane, 0));
            return true;
        } else {
            return false;
        }
    }
}
```

# Putting all together: Scenario

The **Scenario** class lets the **program start** running.

- It **defines** the **time step**;
- **Assignes** the **roads** to the **threads** in the **order specified** before;
- **Updates** the **metrics**;
- **Defines** a **barrier**.

```
public class Scenario {  
    private List<Thread> threadUpdater;  
    private CyclicBarrier barrier;  
    private List<RoadsUpdater> roadsUpdaters;  
    private int step;  
  
    private boolean verbose;  
    private List<Road> roads;  
  
    private void setup(int numOfWorkers) {  
        this.roadsUpdaters = setupThreadsWorkload(numOfWorkers);  
        this.barrier = new CyclicBarrier(numOfWorkers + 1, this::endOfAStep);  
        this.threadUpdater = new ArrayList<>(numOfWorkers);  
  
        for (var upd : roadsUpdaters) {  
            roads.addAll(upd.getRoads());  
            upd.setBarrier(barrier);  
            Thread thr = new Thread(upd);  
            threadUpdater.add(thr);  
        }  
    }  
}
```

# Results



# Metrics

**Density:** indicates the fullness of the road.

$$\frac{\#vehicles}{lanes * laneLength}$$

---

**Average speed:** average speed of all the vehicles inside the road.

$$\frac{\sum_{vehicle} vehicle(speed)}{\#vehicles}$$

---

**Traffic flow:** the product of the two previous metrics.

$$Density * Average\ speed = \frac{\sum_{vehicle} vehicle(speed)}{lanes * laneLength}$$

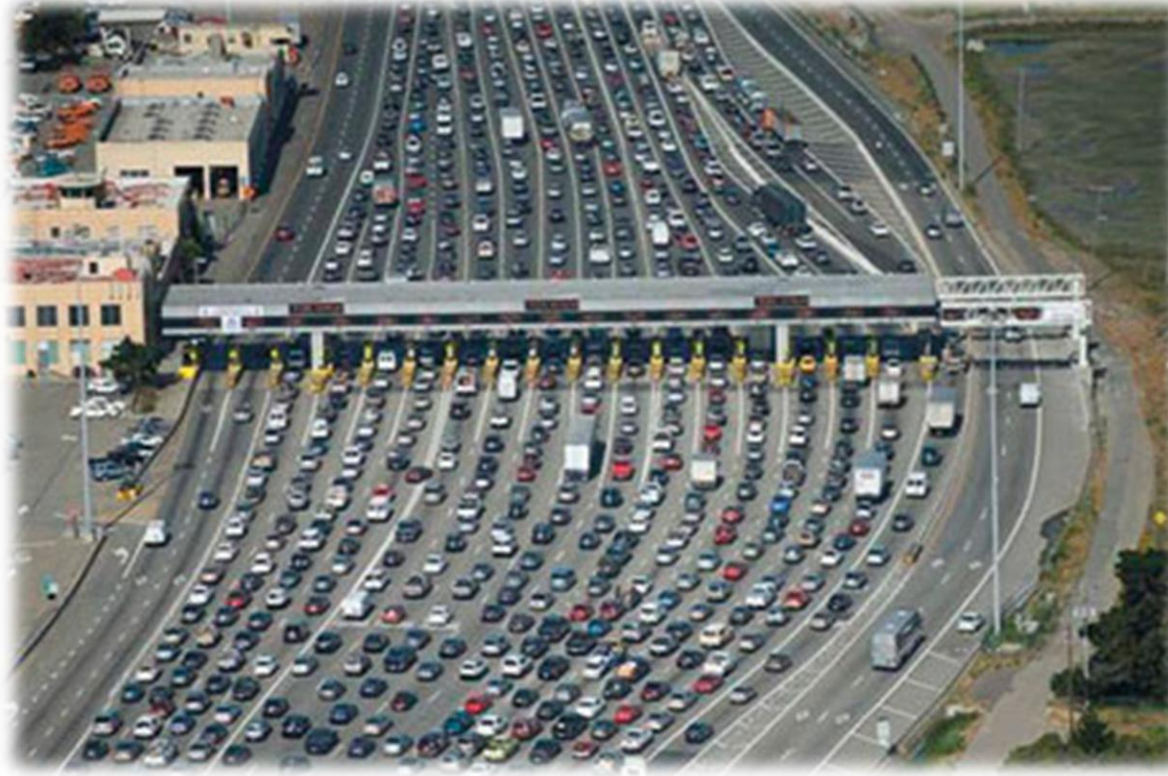
# Lane changes

Average lane changes over **1000 steps**, with  **$pChangeLane = 0.5$** , lanes of **length 50**, **maximum speed = 5** and **one loop road** (the vehicle re-enters at the start, so the density stays the same). The **values are averaged over 5 runs** for each configuration.

Lane changes		Densities			
		0.1	0.3	0.5	0.8
Lanes	2	0.198	0.118	0.018	0
	3	0.3638	0.236	0.047	0.004
	4	0.5896	0.3623	0.0662	0.006
	5	0.7734	0.4871	0.1017	0.0032

As expected, **more lanes and less density increase the number of lane changes**.

# Case study: traffic bottleneck

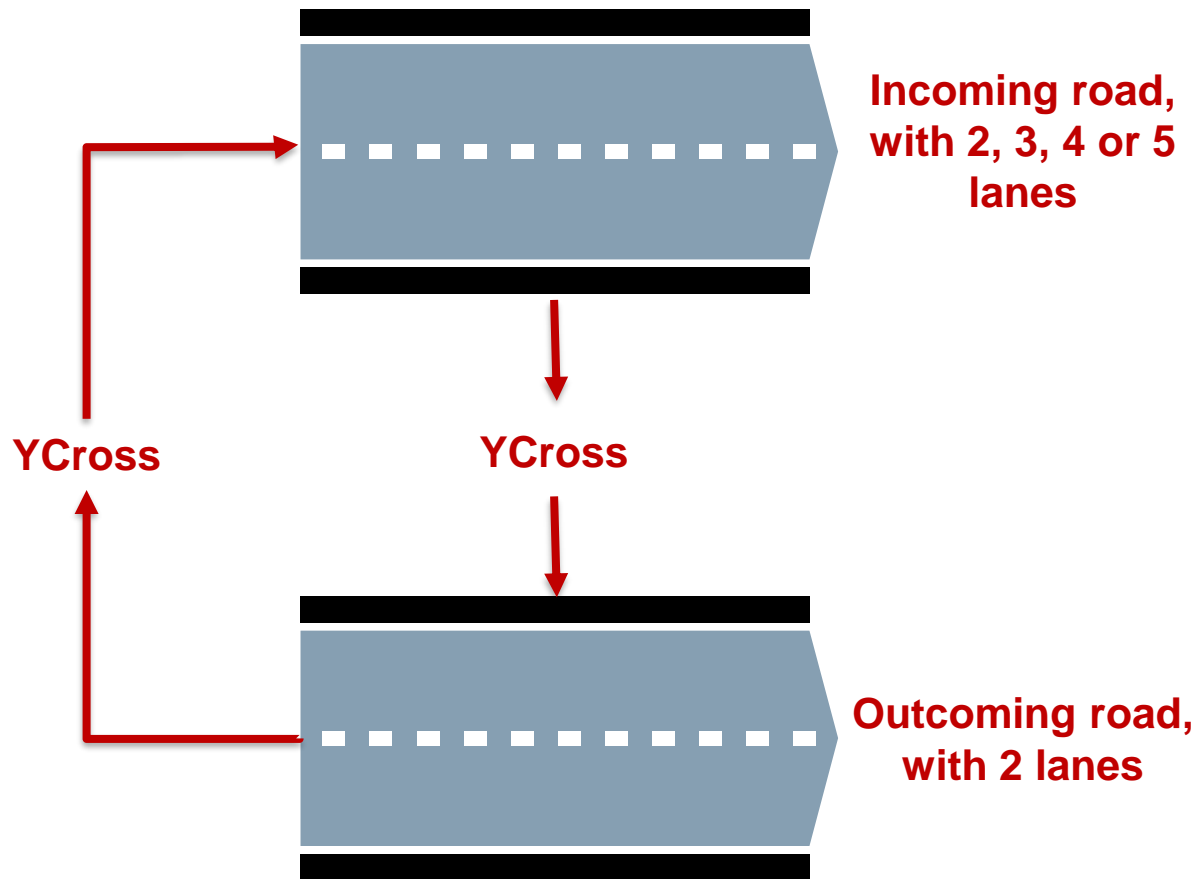


What happens when the **incoming road** has more lanes than the **outcoming one**?



# Traffic bottleneck: setup

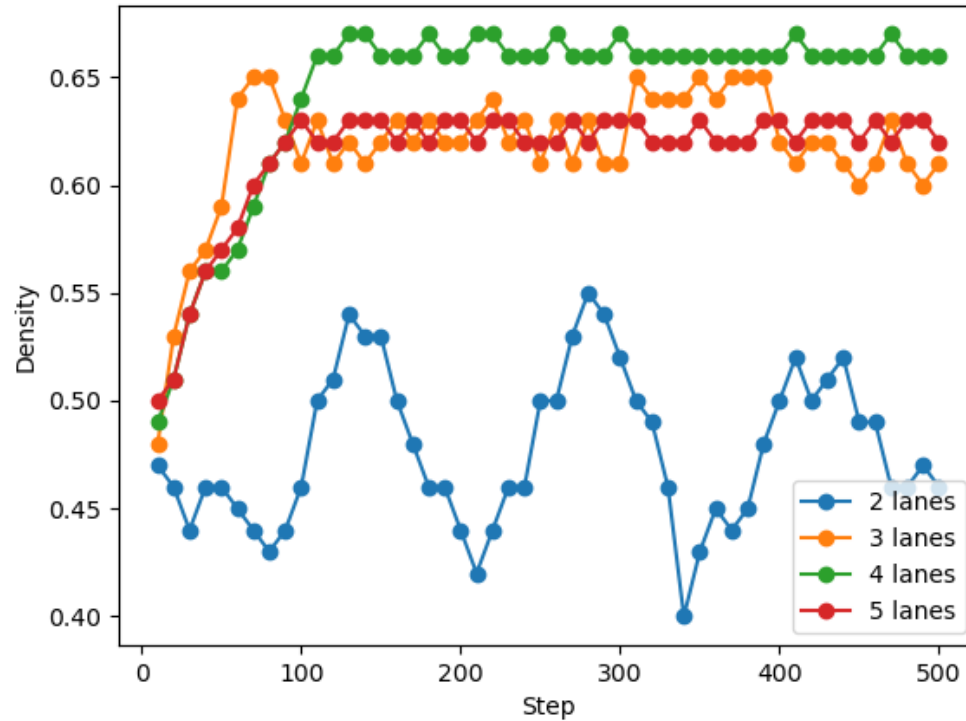
## Scenario structure



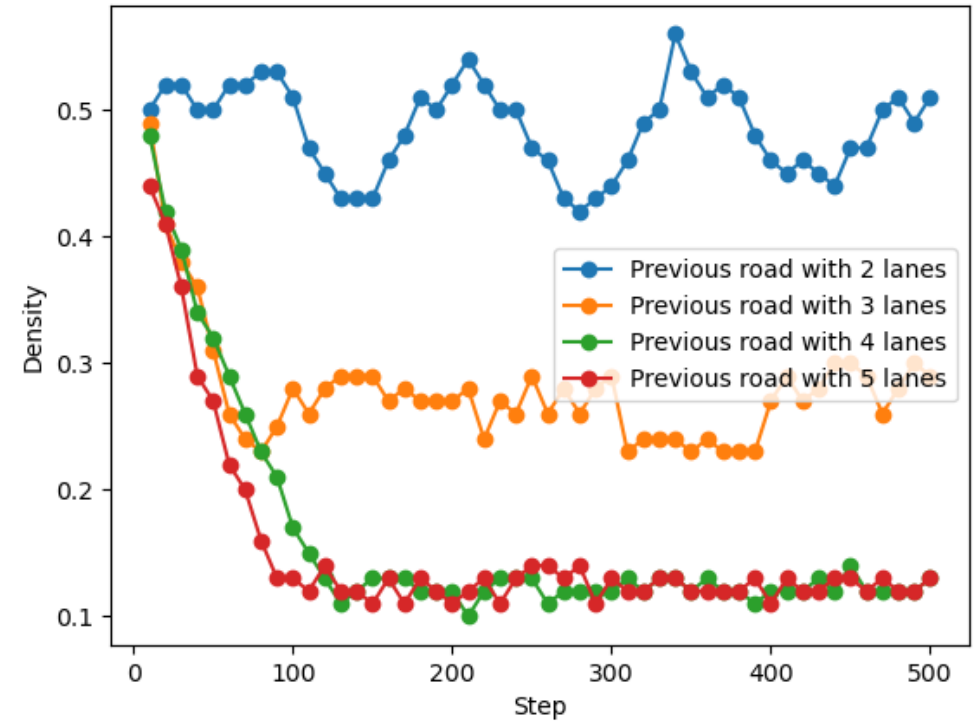
## Parameters

- **Lane length** = 50;
- **Starting density** = 0.5;
- **Maximum speed** = 5;
- **pDecreaseSpeed** = 0.1;
- **pChangeLane** = 0.5.
- **Sample a metric value** every 10 steps.

# Traffic bottleneck: density



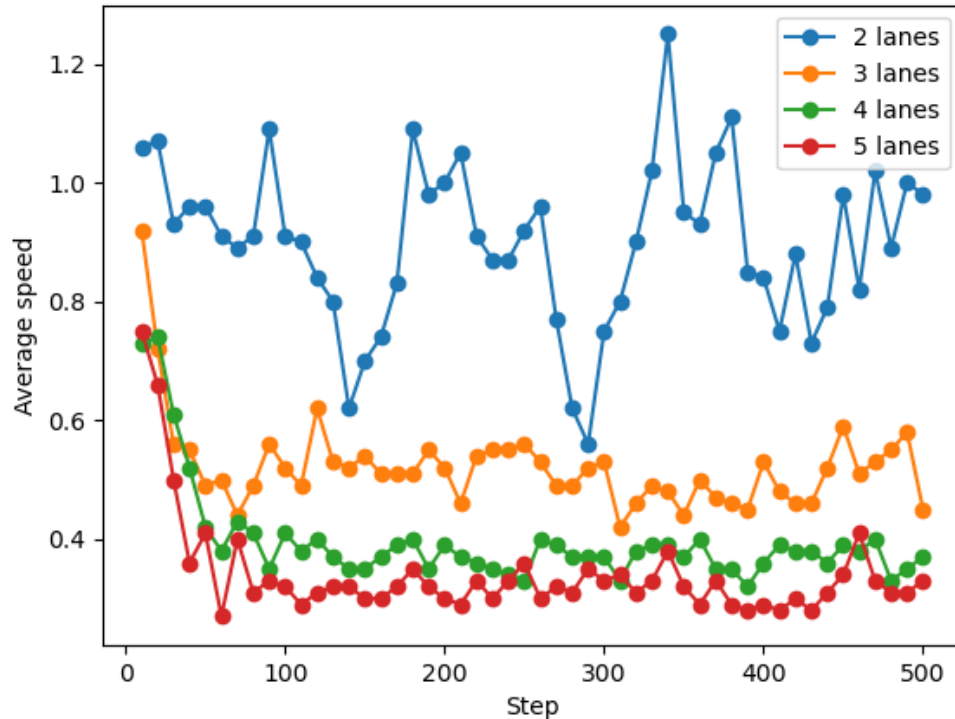
Density for the incoming road



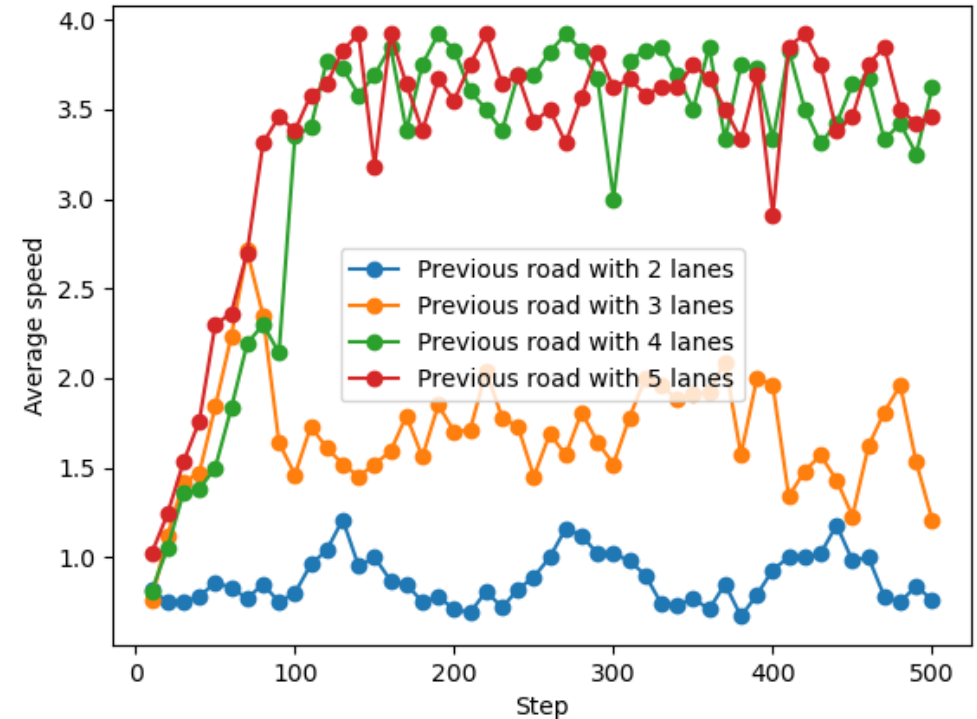
Density for the outgoing road

The **density for the incoming road increases** as the number of lanes grows, meanwhile **it drastically drops for the outgoing road**.

# Traffic bottleneck: average speed



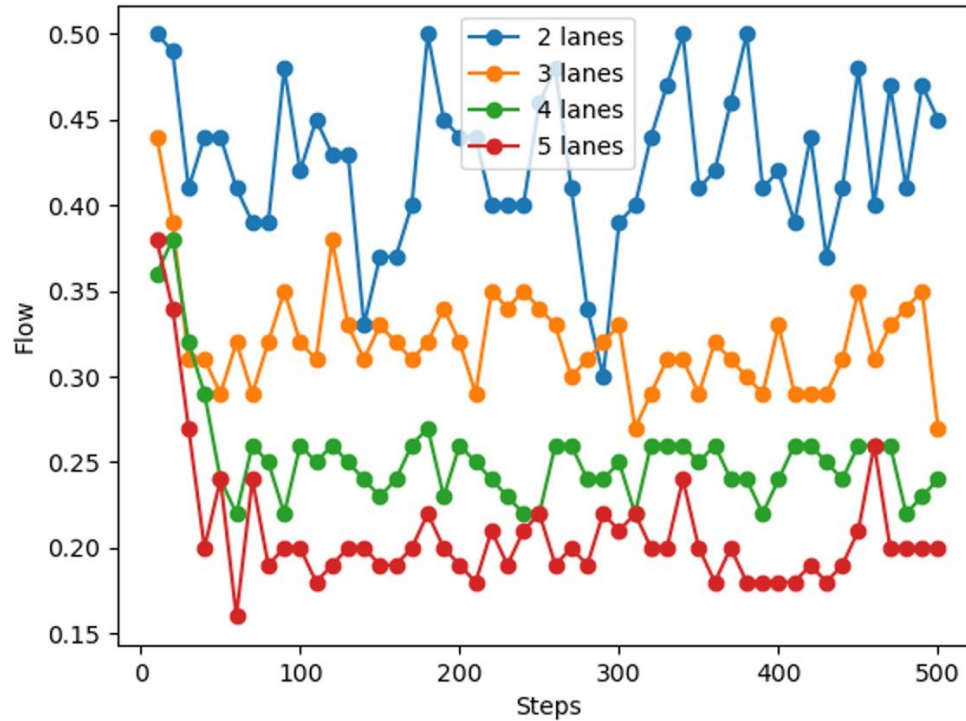
Average speed for the incoming road



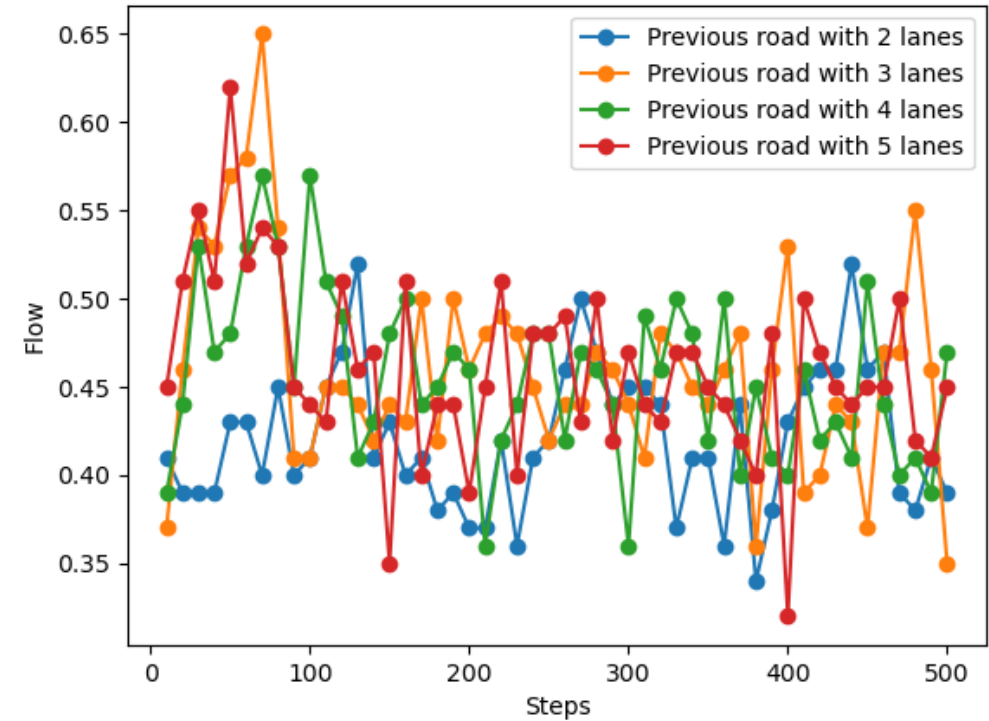
Average speed for the outgoing road

The **average speed for the incoming road decreases** as the number of lanes grows, meanwhile **it increases for the outgoing road**.

# Traffic bottleneck: flow



Traffic flow for the incoming road



Traffic flow for the outgoing road

The **flow for the incoming road decreases** as the number of lanes grows, meanwhile **it stays somewhat the same for the outgoing road**.

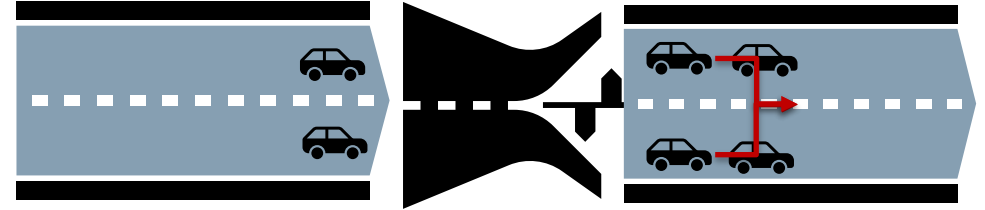


**Demo**

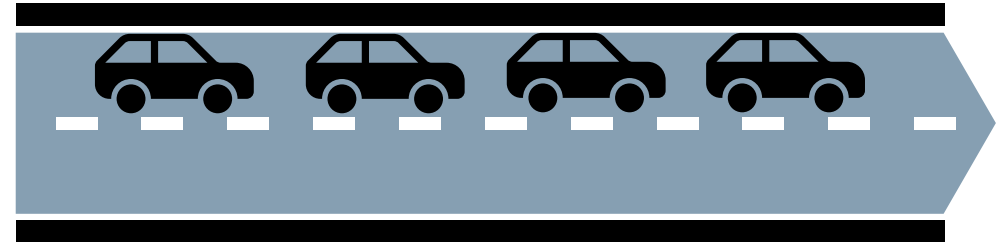


# Three scenarios

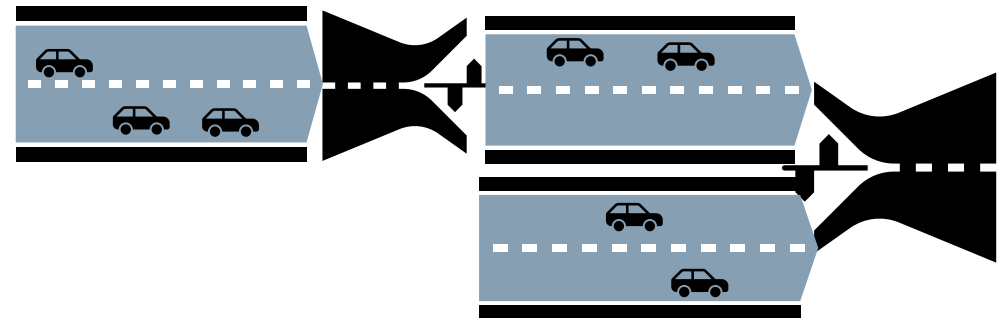
**Overtake and road change:** we will show an example of how to solve the issue of a vehicle changing road while there is an overtake in the outgoing road.



**Ping pong movement:** we will show how the ping pong movement works and when it happens.



**Random scenario:** at last, we will run a random scenario with an incoming road with more lanes than the outgoing one. Moreover, we will also show how two roads converge into one.



# References

Nagel, K., & Schreckenberg, M. (1992). A cellular automaton model for freeway traffic. *Journal de physique I*, 2(12), 2221-2229.

---

Rickert, M., Nagel, K., Schreckenberg, M., & Latour, A. (1996). Two lane traffic simulations using cellular automata. *Physica A: Statistical Mechanics and its Applications*, 231(4), 534-550.



**<https://github.com/nikodallanoco/TrafficAutomata>**