Modelling and Simulation

Optimizing Traffic Lights in Urban Street Grids

Merijn Schröder
Ethan Waterink

Group 10

November 9, 2020

1 Introduction

Controlling traffic lights in city streets can be done in a number of ways. The simplest is just switching the lights according to a fixed schedule, using a clock. Even here, there could be various parameters to control, e.g. which lights stay green for longer (based e.g. on traffic statistics). Other methods use either magnetic loops in the road or radar to assess current traffic load, and use some form of intelligent system to select an optimal strategy. These are still local strategies. It should in theory be possible to let traffic lights communicate, so they can optimize their behaviour globally. The aim of this project is to create a simulation which explores if a global method is better than local ones.

This problem is of interest for a multitude of reasons. First of all, optimizing global traffic light behaviour can save the drivers a great deal of time. This is because the waiting time at traffic lights is minimized, allowing them to get from one place to the other more quickly. It also prevents long waiting queues or possibly traffic jams, benefiting the flow of traffic. On top of that, it is more environmentally friendly, as drivers don't have to brake as often, resulting in less acceleration and tyre wear.

We outline our more precise research question:

"Is a global (communication) traffic light network better than local methods?"

We answer this research question in the following sections. In section 2 we look at some existing studies on this subject. In Section 3 we state and describe the models and methods we are using for our simulations and analyses. In Section 4 we show and describe our results, and discuss them in Section 5. Finally, we conclude the report in Section 6.

2 Related Studies

There are several studies done on how the configuration of traffic lights affect the traffic flow. For instance, A. Latif and P. Megantoro focused on reducing traffic congestion by finding the best duration for a traffic light to be green [1]. For this they looked at the vehicle arrival rate at an intersection. Based on that value the decision is made on how long the traffic light should be green.

A second study develops an emission-saving signal timing model [2]. Again, for this model the focus lies on determining the best duration for a traffic light to be green. The goal is to reduce acceleration of vehicles as much as possible.

Another study dives into the behaviour of individual cars, for instance the distance required for a car traveling at speed limit to come to a full stop [3]. Not only red and green lights are taken into account, but also the yellow lights play a role in the scheme developed. The most important finding is that the optimal time for a certain light color to be on primarily depends on the speed limit, and the spacing and pattern of intersections.

These studies either focus only on the duration of the traffic light being green, or focus on all aspects which could be relevant at the same time. In this report, we investigate only the best order of traffic lights turning green, which we isolate from the other aspects. An important notion is that we are investigating the expected performance of the models, and not trying to optimize for a real-life street grid.

3 Methods

In order to answer the research question, we develop and test multiple models. First, we introduce the concepts for our simulation in Section 3.1, and discuss some assumptions that are made in Section 3.2. Then, we discuss the models in Section 3.3, and how they fit in the simulation in Section 3.4. To get an idea about the performances, we describe our analysis and derive the score for a simulation in Section 3.5. Finally, we discuss some implementation details in Section 3.6.

3.1 Concepts

We first describe some concepts used for creating the model.

Traffic With traffic we are only concerned with vehicles V, so no pedestrians or (motor)cyclists, and for simplicity, only passenger cars will be considered. The vehicles will drive over roads to intersections, which they will cross to continue their drive.

Road A road R connects two intersections and is one-directional. In order for two-directional flow between two intersections two roads have to be present, one for each direction.

Intersection An intersection I is where roads come together, i.e. where they intersect, see Figure 1 for a visualization. There are three possible kinds of intersections:

- 4-way (cross intersection): there are roads at all four directions,
- 3-way (T-intersection): there are roads at three directions,
- 2-way (corner or straight): there are only two roads.

Note that we do not consider 1-way intersections (U-turns).

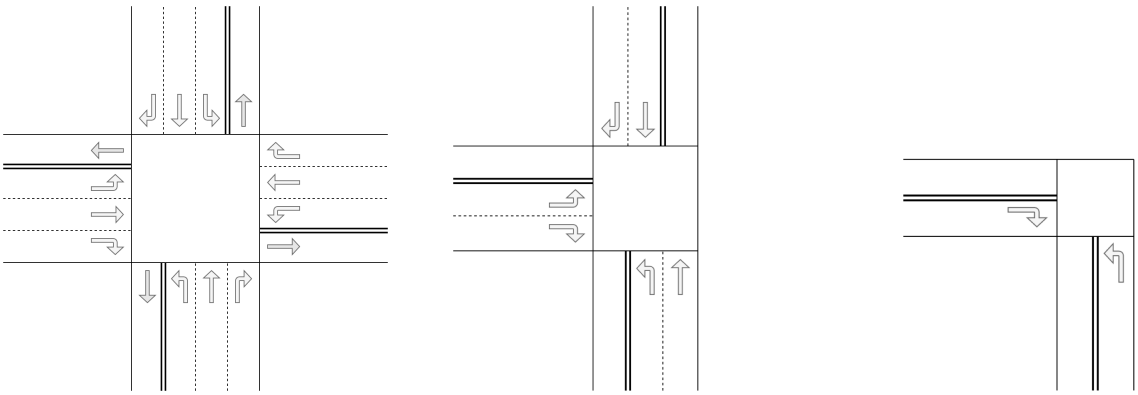


Figure 1: A 4-way intersection (left), 3-way intersection (middle) and 2-way intersection (right).

Lane A lane L is where cars position themselves to turn, which is at the end of the road. Note that these are dedicated turning lanes, i.e. we do not consider shared turning lanes (e.g. one lane for turning straight and right).

Traffic light Traffic lights are signalling devices positioned at intersections to control flows of traffic. For some lanes, it might not be necessary to have a traffic light, as we discuss later. A traffic light T can be in two states, either green (vehicles can drive) or red (vehicles have to wait). This state is determined by some model which changes the configuration of all traffic lights at the intersection.

Street grid With the street grid G we mean all the intersections and roads, where the intersections are placed in a grid pattern. A part of the street grid is shown in Figure 2.

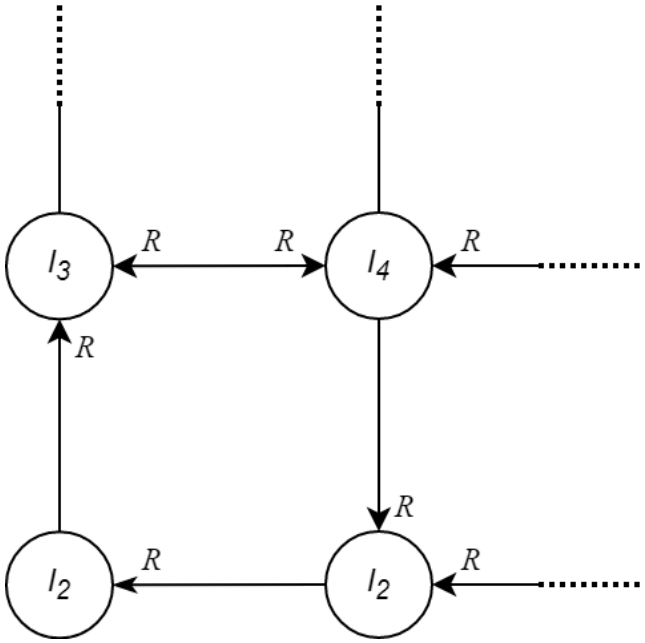


Figure 2: A part of the street grid showing the three kinds of intersections I. Note that the roads R are one-directional.

3.2 Assumptions

There are multiple assumptions we make in our search of finding the best way to control traffic lights. The most significant assumption is that we use discrete time steps instead of continuous time, which greatly simplifies the models and simulation. The vehicles drive with a constant speed over the roads, which is determined by the maximum speed of the roads. This maximum speed can be represented by the road length r, where a smaller r indicates higher maximum speed and vice versa. For this number to make sense in an urban city street grid, it should not be too high, so we limit it to 3. This represents the following:

$$r \equiv \begin{cases} 50\,km/h & \text{if } r = 1, \\ 30\,km/h & \text{if } r = 2, \\ 15\,km/h & \text{if } r = 3. \end{cases} \tag{1}$$

These numbers correspond to different kind of roads: $r = 1$ corresponds to 'normal roads' in a built-up area; $r = 2$ to 30-zones; and $r = 3$ to living streets.

The roads are one-directional, but there can be two roads between two intersections, i.e. one road in one direction and/or one in the other. We do not allow overtaking, assuming it is not safe to do so in an urban city street grid. We assume that every intersection has at least one incoming road and one outgoing road, and we do not allow U-turns.

A road splits into l lanes when close to an intersection, where l is the number of possible directions a vehicle can turn (maximum of three), see Figure 3. The roads in Figure 1 all split into the maximum number of lanes. However, it is also possible some lanes are not present, indicating is not allowed to turn there or there is no outgoing road. We assume that every incoming road has at least one lane, and that for every outgoing road there is at least on lane that goes to that road. For simplicity, the lanes do not have a capacity, so there is no limit to the number of vehicles waiting in one lane.

As roads split into lanes only at the end of the roads, we assume that vehicles enter a lane only when they arrive at the intersections. As a consequence, we can not know beforehand which lane they will enter, only that they are approaching the intersection.

For the vehicles, we assume they all obey the traffic rules and behave in the same way (i.e. no personal driving habits). On top of that, we assume that it takes no time to accelerate and decelerate. In combination with the constant speed of the vehicles, we have to control how many vehicles can pass a green light in one step, which we call the *flow through* (f) of a traffic light. Imagine that the flow through is 4, then the fifth vehicle waiting in front of the traffic light cannot drive at this step. We allow some variation in the flow through per traffic light. For example, if all f vehicles react quickly, the $(f + 1)^{th}$ vehicle has a chance to go through as well.

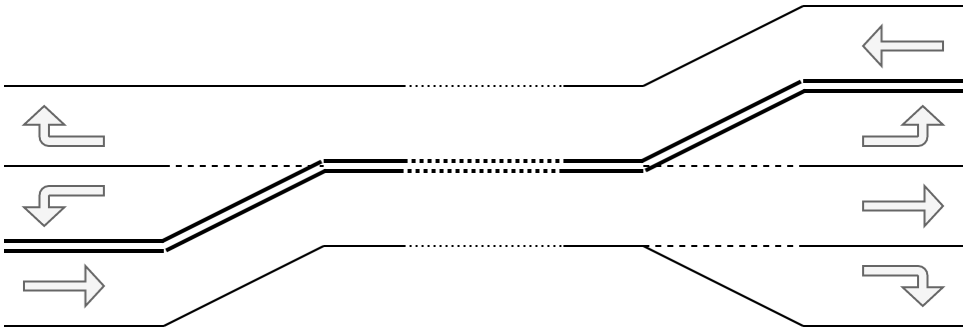


Figure 3: Two one-directional roads between two intersections, which splits into two lanes near the left intersection, and into three lanes near the right intersection.

On the other hand, if a vehicle responds slowly, one or two vehicles behind might not make it. Traffic lights are updated every t time steps, which is fixed at $t = 1$.

3.3 Model

Traffic lights can be either green or red, but not all can be green at the same time, only in certain combinations with each other. This is because (i) vehicles going in one direction can block other vehicles from going in another direction, and (ii) the outgoing road can be entered by only one stream of vehicles at a time. We can represent this green/red configuration in a table, see Table 1. It shows the possible traffic light combinations (TLCOMB)

	T_L^D	T_S^D	T_R^D	T_L^{D+L}	T_S^{D+L}	T_R^{D+L}	T_L^{D+S}	T_S^{D+S}	T_R^{D+S}	T_L^{D+R}	T_S^{D+R}	T_R^{D+R}
T_L^D	*	✓	✓	✗	✗	✓	✓	✗	✗	✗	✗	✓
T_S^D	✓	*	✓	✗	✗	✓	✗	✓	✓	✗	✗	✗
T_R^D	✓	✓	*	✓	✗	✓	✗	✓	✓	✓	✓	✓

Table 1: Traffic light combinations (TLCOMB) on a 4-way intersection, showing if traffic light T_X^D can be green simultaneously with T_Y^{D+Z}.

on a 4-way intersection with three lanes on all directions. T_X^D indicates the traffic light at the lane for turning $X \in [L, S, R] \equiv$ [Left, Straight, Right] on the road at direction $D \in [N, E, S, W] \equiv$ [North, East, South, West], see also Figure 4. A ✓ indicates that the traffic light at T_X^D can be green at same time as T_Y^{D+Z} without any conflict, whereas a ✗ would result in conflict. $D + Z$ is the direction of the outgoing road when a vehicle comes from road D and turns $Z \in [L, S, R]$. For example, if a vehicles enters a lane on the $D = N$ side for turning $Z = R$, it will end up at the $D + Z = N + R = W$ side (again, see Figure 4). A $*$ indicates that the two traffic lights are in fact the same. Note that we can compact the table in this way because of rotational symmetry, i.e. it does not matter from which direction the vehicle comes. If we decide to change that more than vehicle stream can enter an outgoing road at the same time, we only have to change the allowed traffic light combinations in TLCOMB.

Traffic light models transform the current traffic light configuration to a new traffic light configuration, and different models transform the configuration in different ways and based on different information. A traffic light configuration (TLCONF) can be represented as a 1-dimensional list, with maximum size $4 \times 3 = 12$:

$$\text{TLCONF} = [T_L^N, T_S^N, T_R^N; T_L^E, T_S^E, T_R^E; T_L^S, T_S^S, T_R^S; T_L^W, T_S^W, T_R^W] \tag{2}$$

Each of the traffic lights in the TLCONF can be either green or red. Traffic lights (and thus lanes) that are not present in an intersection or do not have a traffic light are excluded from the list. For example, if there is no lane for turning right on the North side of the intersection, the T_R^N element would be removed. Also, if the intersection is 3-way, with no incoming South-side road, all the T^S traffic lights would be empty. These two examples give the TLCONF $= [T_L^N, T_S^N, -; T_L^E, T_S^E, T_R^E; -, -, -; T_L^W, T_S^W, T_R^W]$.

The traffic light models follow the same general form of Algorithm 1, that is, all models have a SETUP and UPDATE method. In the SETUP, model-specific attributes can be set for each intersection in the grid, while the UPDATE method updates the TLCONF of the intersection. It also provides two utility methods, ALLTRAFFICLIGHTSRED and FINDNONCONFLICTING. The former simply turns all traffic lights at an intersection to red. The latter returns a list of lanes (from the input lanes) that do not conflict with the reference lane. Whether a lane conflicts with the reference lane is determined by the TLCOMB (Table 1).

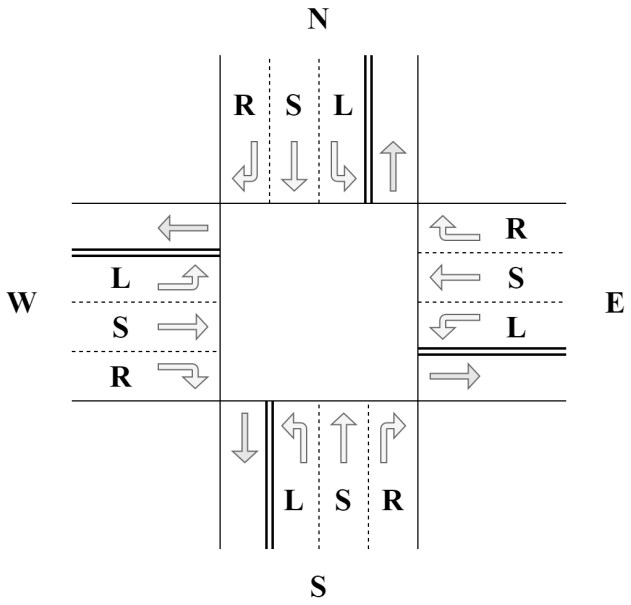


Figure 4: A 4-way intersection showing T_X^D for directions $D \in [N, E, S, W]$ and turnings $X \in [L, S, R]$.

Algorithm 1: Traffic light model (base).

```
1 Function Setup(G):
      Input: The grid G containing all intersections, roads and lanes.
2 Function Update(I):
      Input: An intersection I to be updated.
3 Function AllTrafficLightsRed(I):
      Input: An intersection I.
4     foreach inc_R ∈ I.incoming_roads do
5         foreach L ∈ inc_R.lanes do
6             turn L to red
7 Function FindNonConflicting(reference_L, lanes):
      Input: A lane reference_lane and a list of lanes lanes.
      Output: A list of lanes non_conflicting (out of lanes) that do not conflict with reference_lane.
8     non_conflicting := []
9     foreach other_L ∈ lanes do
10        if traffic light combination is possible with reference_L and other_L then
11            append other_L to non_conflicting
12    return non_conflicting
```

There are four main models for controlling the traffic lights, which we describe below ordered in increasing complexity.

3.3.1 Clock

The simplest model for controlling the traffic lights is just switching the lights according to a fixed schedule, using a clock. As only a maximum of four traffic lights can be green at the same time, we have to find a fair and suitable way to cycle through the lanes. A simple approach would be to start with setting all traffic lights on side D to green. Then after t steps turn them red and the next traffic lights on side $D + L$ turn green. This way, we turn the traffic lights green and red in a clock-wise manner. If an incoming direction is missing or it has no lanes with traffic lights, we can skip it and go to the next direction. We note that if all T_L^D, T_S^D, T_R^D are green, there is one other traffic light that can be green as well. From Table 1 (and our own experience in real life) we see that T_R^{D+L} can green at the same time, as this column contains three ✓.

We can now define how to go from the current TLConf to the next. For example, if we start on the East-side, we go the South-side:

$$[✗,✗,✗;✓,✓,✓;✗,✗,✓;✗,✗,✗] \implies [✗,✗,✗;✗,✗,✗;✓,✓,✓;✗,✗,✓] \tag{3}$$

We see that it basically shifts the TLConf to the right. The Clock traffic light model is shown in Algorithm 2.

Algorithm 2: Clock traffic light model.

```
1  Function Setup(G):
2      first_time_calling := True
3      lanes_per_direction := {}
4      foreach I ∈ G.intersections_with_traffic_lights() do
5          lanes_per_direction[I] := []
6          foreach inc_D, inc_R ∈ I.incoming_roads do
7              if inc_R has traffic lights then
8                  append LanesToChange(I, inc_D) to lanes_per_direction[I]

9  Function LanesToChange(I, D):
       Input: Intersection I, direction D.
       Output: The lanes lanes corresponding to D.
10     lanes := []
11     foreach L ∈ I.incoming_roads[D].lanes do
12         if L has traffic light then
13             add L to lanes
14     next_D := D.next()                          /* D.next() ≡ D + L */
15     if next_D ∈ roads := I.incoming_roads then
16         if RIGHT ∈ next_lanes := roads[next_D].lanes then
17             if next_L := next_lanes[RIGHT] has traffic light then
18                 append next_L to lanes
19     return lanes

20 Function Update(I):
21     lanes := lanes_per_direction[I]
22     if first_time_calling == True then
23         shuffle lanes
24         foreach L ∈ lanes[0] do
25             turn lane to green
26         first_time_calling := False
27         return
28     foreach L ∈ lanes[0] do
29         turn L to red
30     shift lanes
31     foreach L ∈ lanes[0] do
32         turn L to green
```

In the Setup method, we see that the clock model uses two class-specific attributes, namely $first\_time\_calling$ and $lanes\_per\_direction$. The $first\_time\_calling$ is a Boolean variable which is used to check if the Update method is called for the first time or not. $lanes\_per\_direction$ is a dictionary indexed by intersection and stores a list of lists. We initialize $first\_time\_calling$ to True (line 2) as the Update method is still to be called for first the time. line 3 initializes the empty dictionary. Then for each intersection that has traffic lights, we initialize an empty list for it in the dictionary (line 5). For every incoming road that has traffic lights, we add a list of the corresponding lanes for that direction to the list of the intersection.

Which lanes correspond to a direction is determined by LanesToChange(I, D). As stated before, these are the lanes that are on the D side of the intersection (lines 11-13) plus possibly the lane for turning right on the next direction (lines 14-18). We only add the lanes that have a traffic light.

In the UPDATE method we refer to an intersection's *lanes_per_direction* as *lanes* for brevity (line 21). If this is the first time we call the UPDATE method, the first thing we do is shuffle the lanes. By doing so, we randomize the order in which the directions are updated (so no longer in a clock-wise manner) to avoid any bias. Then we turn all the lanes for the first direction (*lanes*[0]) to green (line 25) and set $first\_time\_calling$ to False (line 26). Now, every new call to UPDATE will skip lines 23-26, and proceed as follows: all current lanes are turned to red (line 29), we shift the lanes as described in (3) (line 30), and turn the new lanes to green (line 32). This way, we cycle through the list of directions.

3.3.2 First come first serve

An improved model uses a first come first approach. The lane which is entered first will be the first to turn green. The First Come First Serve (FCFS) traffic light model is shown in Algorithm 3.

Algorithm 3: First come first serve traffic light model.

```
1  Function Setup(G):
2      queues := {}
3      foreach I ∈ G.intersections_with_traffic_lights() do
4          queues[I] := []

5  Function Update(I):
6      queue := queues[I]
7      lanes_with_vehicles := I.get_all_lanes_with_vehicles()
8      shuffle lanes_with_vehicles
9      foreach L in lanes_with_vehicles do
10         if L ∉ queue then
11             append L to queue
12     AllTrafficLightsRed(I)
13     options := copy(queue)
14     while LEN(options) > 0 do
15         L := options.POP(0)
16         turn L to green
17         options := FindNonConflicting(L, options)
18         remove L from queue
```

In the SETUP we initialize the *queues* attribute, which is a dictionary indexed by intersection and stores a list (queue) of lanes. These lists are initialized to empty for every intersection that has traffic lights.

In the UPDATE method we refer to an intersection's *queues* as *queue* for brevity (line 6). First, we get a list of all the lanes at this intersection that have vehicles waiting (line 7). Because the order of this list is always the same (*get_all_lanes_with_vehicles*() is a deterministic function which returns the same list every time), we cannot distinguish which lanes were entered first. Hence, we shuffle the list, as if they appear in random order (line 8). All lanes that are not already in the queue are added to it (line 11). All the traffic lights are turned to red (line 12), as we are going to determine a new `TLConf`. First, we make a copy of the *queue*, called the *options* list (line 13). This list contains the candidate lanes to be turned green. Then, while there are still options left, we do the following: we pop the lane L which was entered first, i.e. the first lane in the queue (line 15) and turn it to green (line 16). Then we update the *options* list by keeping the lanes that do not conflict with L. In other words, we remove the lanes that do conflict, which is done with the FINDNONCONFLICTING(L, *lanes*) method (line 17). An important property of this method is that it keeps the relative order of the lanes. Finally, we remove L from the *queue* (line 18). If there are still options left, we return to line 15. Note that this is not necessarily the lane that entered second after the first lane: it is the lane that entered next which does not conflict with the first.

3.3.3 Local Optimum

More refined models aim to achieve a local optimum. The main approach is to use either magnetic loops in the road or radar to assess current traffic load, and use some form of intelligent system to select an optimal strategy. An idea is to use some priority, which could be, e.g., the number of vehicles waiting in front of a traffic light. The more vehicles there are, the higher the priority. The Local Optimum model is shown in Algorithm 4.

Algorithm 4: Local Optimum traffic light model with priority.

```
1  Function Setup(G):
2      pass
3  Function Update(I):
4      AllTrafficLightsRed(I)
5      options := I.get_all_lanes_with_traffic_lights()
6      while LEN(options) > 0 do
7          highest_priority_L := HighestPriority(options)
8          turn highest_priority_L to green
9          remove highest_priority_L from options
10         options := FindNonConflicting(highest_priority_L, options)
11 Function HighestPriority(lanes):
       Input: A list of lanes lanes.
       Output: The lane highest_priority_lane with the highest priority.
12     highest_priority_L := None
13     highest_queue_length := −1
14     foreach L ∈ lanes do
15         if LEN(L.queue) > highest_queue_length then
16             highest_priority_L := L
17             highest_queue_length := LEN(L.queue)
18     return highest_priority_lane
```

This models does not require any setup, so the SETUP method is empty. In the UPDATE method, we start with turning all traffic lights red (line 4) and initializing the *options* list of traffic lights at intersection I (line 5), from which the algorithm can choose traffic lights. Then, while there are still options left, we do the following: first, get the lane with the highest priority in the *options* list using the HIGHESTPRIORITY(*lanes*) method (line 7). This method simply loops through all the lanes in *lanes* and returns the lane with the longest queue (highest priority). This lane is turned to green (line 8) and removed from the options (line 9). Then we update the *options* list by keeping the lanes that do not conflict with L using the FINDNONCONFLICTING(L, *lanes*) method (line 10). Note that at most 4 lights can green at the same time anyway, so this process is repeated at most 3 times before completing.

3.3.4 Global Optimum

Finally, we aim to optimize the traffic lights' global behaviour by letting traffic lights communicate with each other. One approach for this global optimization between traffic lights is to turn the traffic light of a neighbouring intersection to green after r time steps (where r is the road length, see Section 3.2). Another approach is to let the traffic lights of neighbouring intersections communicate. When a stream of vehicles cross an intersection to enter a road, the intersection can tell the other intersection on the other end of the road how many vehicles are coming.

An idea is to use utilize the PRIORITY method from the Local Optimum model in combination with data of incoming traffic, so that incoming vehicles have a chance to continue driving, without having to stop. As roads have a length of r steps, we would have to store the number of incoming vehicles for every step from one intersection to another. E.g., if $r = 3$, we would know the current traffic situation and the incoming traffic at step $n + 1$, $n + 2$ and $n + 3$. In general, this is true for $n + i$ for $i = 1..r$. These can be used to determine the best `TLConf` to enhance traffic flow. For simplicity, we will only take into account the incoming traffic at $n + 1$.

As discussed in Section 3.2, we do not know which lane vehicles will enter at steps $n + i$, but only at step n. But at that time, the vehicles are (possibly) already waiting in front of the traffic lights. The only thing we know is the number of vehicles that are on their way. What we can do is equally distribute the number of vehicles at step $n + 1$ over the lanes, as if they would all enter a random lane with equal probability. This way, it would appear to the intersection that more vehicles are waiting in front of the traffic lights, while actually they are on their way and arrive at the next time step, allowing them to continue driving. Then, we can apply the Local Optimum model to this extended traffic situation, as shown in the Global Optimum model in Algorithm 5. Note that this implementation assumes we have the incoming traffic at I available.

Algorithm 5: Global Optimum traffic light model with extended priority.

```
1  Function Setup(G):
2      pass
3  Function Update(I):
4      AllTrafficLightsRed(I)
5      options := I.get_all_lanes_with_traffic_lights()
6      while LEN(options) > 0 do
7          highest_priority_L := HighestPriority(options)
8          turn highest_priority_L to green
9          remove highest_priority_L from options
10         options := FindNonConflicting(L, options)
11 Function HighestPriority(lanes):
       Input: A list of lanes lanes.
12     highest_priority_L := None
13     highest_queue_length := −1
14     lane_extra := Distribute(I)
15     foreach L ∈ lanes do
16         if LEN(L.queue) + lane_extra[L] > highest_queue_length then
17             highest_priority_L := L
18             highest_queue_length := LEN(L.queue)
19 Function Distribute(I):
       Input: Intersection I.
20     lanes_extra := {}
21     foreach inc_R ∈ I.incoming_roads do
22         num_lanes := LEN(inc_R.lanes)
23         add_per_lane = LEN(inc_R.incoming_vehicles)/num_lanes
24         foreach L ∈ inc_R.lanes do
25             lanes_extra[L] := add_per_lane
26     return lanes_extra
```

This model, too, does not require any setup, so the SETUP method is empty. Note that the UPDATE method is exactly the same as the one in the Local Optimum model, see Algorithm 4. The difference lies in the HIGHESTPRIORITY(*lanes*) method, albeit a small difference. In this method, we compute an extra weight for each lane (line 14) and add it to the length of the queue when try to determine the lane with the highest priority. As stated before, the weight is calculated per incoming road in the DISTRIBUTE(I) method. We start by initializing the *lanes_extra* variable, which is a dictionary indexed by lane and stores the additional weight (line 20). Then for each road we do the following: get number the number of lanes on this road (line 22) and compute how many additional vehicles should be added per lane (line 23), which we store for each lane in the dictionary (line 25).

3.4 Simulation

Before the simulation can be started, a street grid G has to be randomly generated, which consists of a $w \times h$ grid of intersections. For every intersection there might be an outgoing road to another neighbouring intersection, where not all neighbouring intersections will be connected with a road, creating 4-, 3- and 2-way intersections. After the initial placement of the roads, we have to make sure they meet two requirements:

1. every intersection has at least one incoming and outgoing road, and
2. every incoming road has at least one outgoing road, and vice versa (which is not on the same direction).

Requirement 1 makes sure that every intersection can be entered and left. If, for example, there are three incoming roads, but no outgoing road, then incoming vehicles have no where to go. Another reason is to minimize the chance that part of the grid is disconnected from the rest. Requirement 2 ensures that vehicles can cross the intersection to go to another road. It also makes sure that every outgoing road can be entered.

Once all the roads are set up, we arrange the lanes on the roads. Similarly to the road set up, for every road there might be a lane going to another road. Again, after the initial placement of the lanes, we have to make sure they meet two requirements:

1. every incoming road has at least one lane, and
2. every outgoing road can be reached by at least one lane.

Both requirements are related to the road requirements. Requirement 1 makes sure that the road is not a dead end, and Requirement 2 ensures that a road can be entered from at least one lane.

After all the lanes have been set up, we add the traffic lights. Note that not all lanes require traffic lights. Only the lanes that conflict with at least one other lane (in the TLCOMB) require traffic lights.

Finally, we set up the vehicles, which are distributed over the intersections (specifically the lanes), which is their starting intersection. This represents people leaving their work. Vehicles do not have a destination, but instead have to travel a specified number of roads. While doing so, they randomly enter lanes and travel around the grid. This number of roads is not the same for all vehicles. Instead, we define a minimum and maximum number of roads a vehicle can travel. The minimum should not be too low (relative to the grid size), so we set it to $\frac{w+h}{2}$. For the maximum, it wouldn't make much sense if a vehicle travels more than twice the perimeter of the grid, so we set it to $2(w + h)$.

During the simulation no new vehicles enter the grid. This way the traffic lights experience a rush hour, the busiest time of the day. Over time, vehicles arrive at their destination and leave the grid. If we would introduce new vehicles during the simulation, then it would constantly be a rush hour. While we could decrease the introduction rate over time, the same is achieved by simply not introducing new vehicles.

With all these random initializations, it is important to seed the random generator so that the simulations can be reproduced. Remember that we are investigating the expected performance of the models, and not trying to optimize for a real-life street grid.

A general overview of the simulation is listed in Algorithm 6. The TRAFFICLIGHTMODEL can be any of the aforementioned traffic light models, e.g. the Clock model.

Algorithm 6: General traffic light street grid simulation.

```
1  Function Simulate(G, TLM):
       Input: Grid G, Traffic ligh model TLM.
       Output: Data about the vehicles, e.g. mean number of waiting steps.
2      vehicles_driving := LEN(G.vehicles)
3      step := 0
4      while vehicles_driving > 0 do
5          foreach I ∈ G.all_intersections_with_traffic_lights() do
6              if step % I.traffic_light_length == 0 then
7                  TLM.UPDATE(I)
8          foreach incoming V do
9              arrive at V.destination and enter a lane
10         foreach V in a lane L do
11             if L does not have a traffic light or is green then
12                 if L's flow trough limit not reached then
13                     V.cross_intersections()
14             else
15                 V.wait()
16         decrease vehicles_driving with the number of finished vehicles
17         step := step + 1
18     return Data about the vehicles
```

The simulation starts with initializing *vehicles_driving* to the number of vehicles in the grid (line 2). It keeps count of how many vehicles are still driving, i.e. not finished. We also keep track of the current step number, which is initialized to 0 (line 3). Then, while there are still vehicles driving we do the following: first update all the intersections' traffic lights with the TLM model (line 7). Note that this is done once every

$I.traffic\_light\_length$ step. Next, the incoming vehicles arrive at the intersection and enter a lane (line 9). After that, all the vehicles that are in a lane L can cross the intersection (line 13) if L does not have a traffic light or its traffic light is green. That means that the vehicles that just arrived can continue driving without having to stop (if the flow through limit has not been reached yet). If L's traffic light is red the vehicles have to wait another step (line 15). Finally, we decrease the number of driving vehicles by the number of vehicles that have finished this step (line 16), and increment the step counter (line 17).

3.5 Analysis

With our models and simulation set up, we have to come up with some score to determine the performance of the model in a simulation. Let's consider vehicle v as it drives around from intersection to intersection. It does so in a total of v_n steps, of which v_w spent waiting in front of traffic lights and v_d driving, such that $v_w + v_d = v_n$. This means that the vehicle has travelled a distance of v_d, which is equivalent to $v_R = \frac{v_d}{r}$ roads travelled. We also consider the number of traffic lights encountered on its path, v_{TL}. To clarify, this number represents the number of lanes with traffic lights driven, not the number of intersection with traffic lights or the number of unique traffic lights on its path.

We want to minimize the waiting time in front of traffic lights, so a good indication would be the average waiting time in front of traffic lights. Vehicle v experienced an *optimal* drive if $v_w = 0$, i.e. it did not have to wait for any traffic light and could drive the whole time. As this most unlikely to happen, we consider a drive to be *good* when v had to wait for less than (or equal to) half of the encountered traffic lights *on average*. On the other hand if $v_w = v_{TL}$ it had to wait for every traffic light on its path *on average*. We consider this to be the best worst-case scenario, as vehicles can wait in front of traffic lights for longer than one time step. In that case $v_w > v_{TL}$ and, consequently, its average waiting time in front of traffic lights is > 1. Of course, v could spend more time waiting at one traffic light than at another, but what is important is the *average* over the drive. If, e.g., v spends a lot of steps waiting at one traffic light, but does not have to wait for the others on its path, its average can be considered good.

The average waiting time in front of traffic lights $\hat{v}_{wTL} = \frac{v_w}{v_{TL}}$ can be classified as follows:

$$score = \begin{cases} \text{optimal} & \text{if } \hat{v}_{wTL} = 0 \\ \text{good} & \text{if } 0 < \hat{v}_{wTL} \leq 0.5 \\ \text{average} & \text{if } 0.5 < \hat{v}_{wTL} \leq 1 \\ \text{bad} & \text{if } \hat{v}_{wTL} > 1 \end{cases} \tag{4}$$

Note that if v did not encounter any traffic lights, $v_{TL} = 0$, which causes trouble in (4). In this case, we discard it as it provides no information about the traffic light model and hence is of no value. It is also worth mentioning that with a high traffic load, the simulation score naturally increases and may be classified as bad, but could nevertheless perform relatively better than others.

Now we want to use this in the calculation of a simulation score. A simple approach is to sum up all average waiting times in front of traffic lights and take the average over all vehicles:

$$simulation\_score = \frac{1}{\#V} \sum_{v}^{V} \hat{v}_{wTL}. \tag{5}$$

The simulation can be classified in the same way as in (4). In the most optimal case, not a single vehicle had to wait and $simulation\_score = 0$ (again, highly unlikely). In the good case all vehicles had to wait at less than half of the encountered traffic lights. In the average case, all vehicles had to wait for some, but not all traffic lights. And if all vehicles had to wait for all traffic lights on their paths, we have a bad simulation score. Note, again, that these are *on average*. Moreover, we do not use $\frac{v_w}{v_n}$, as vehicles can drive without encountering traffic lights, blurring the performance score. In summary, the lower the score, the better.

The performance score could be more complex by taking into account, e.g., road length r, the flow through of the traffic lights f, and total time steps v_n. However, we decided to keep it simple.

We want to find out how the models respond to different amounts of traffic loads. Given a number of vehicles, we do multiple runs on different grids and compute the average score. This will give a general impression of how the models perform. Then, we do this over a relatively large range of vehicle loads to observe how they perform on different traffic loads.

3.6 Implementation details

The simulation is implemented using Python and the code consists roughly of three main parts: setting up the street grid, running the simulation and visualizing the results. Each part has its own configuration values, which can be set in the configuration file (*config.py*). We provide a brief discussion of both parts with their configuration values.

Setup In this part a random street grid is generated where vehicles are randomly distributed over the intersections' lanes. The setup follows the description in Section 3.2 and 3.4. A `GRID_WIDTH` by `GRID_HEIGHT` grid is created, and the initial roads and lanes are added with probabilities, which are determined by the constants `ROAD_PROBABILITY` and `LANE_PROBABILITY`, respectively. For the vehicles, the minimum and maximum roads to travel are determined by respectively `VEHICLE_MIN_ROADS = (GRID_WIDTH+GRID_HEIGHT) // 2` and `VEHICLE_MIN_ROADS = (GRID_WIDTH+GRID_HEIGHT) * 2`.

One thing that is not mentioned is how the road length r is decided. We define two constants: `ROAD_LENGTH_BASE` and `ROAD_LENGTH_DIFF`. In Section 3.2, we already mentioned that $1 \leq r \leq 3$, and we want most of the roads to have a speed limit of $50\,km/h$, so `ROAD_LENGTH_BASE` $= 1$. Let's say that we want 85% of the roads to have $r = 1$, 10% $r = 2$ and 5% $r = 3$. Then `ROAD_LENGTH_DIFF` is a list `85*[0] + 10*[1] + 5[2]` (which is a non-uniform distribution) from which we randomly choose a value for the initialization of the roads. The final road length is thus `ROAD_LENGTH_BASE + random.choice(ROAD_LENGTH_DIFF)`.

The traffic light models are implemented as classes and they implement the algorithms as described in Section 3.3. TrafficLightModel is a base class with abstract methods Setup and Update, and the models are subclasses which implement the methods.

Simulation In this part we run the simulation as described in Section 3.4. At each step, we call the Update method of a model (which is defined for all models). As described in Section 3.5, we want to test the models over a relatively large range of vehicles. Given the starting traffic load `TRAFFIC_LOAD_START`, ending traffic load `TRAFFIC_LOAD_END` and the number of samples `TRAFFIC_LOAD_NUM`, we create a list of evenly spaced traffic loads over this interval. Then, for each traffic load, we run the simulation `SIMULATIONS_PER_TRAFFIC_LOAD` times. The seed for the random number generator can be set with `RANDOM_SEED`.

Similar to the road length setup, for the flow through f we define constants `FLOW_THROUGH_BASE` and `FLOW_THROUGH_DIFF`. Every time we update a lane, we compute how many vehicles can go through. As described in Section 3.2 most of the time the flow through is f. But it can happen that vehicles react quickly ($f + 1$) or slowly ($f - 1$ or even $f - 2$). Let's say that 80% of the time the flow through is f, 5% it is $f + 1$, 10% $f - 1$ and 5% $f - 2$. Then `FLOW_THROUGH_DIFF = 5*[+1] + 80*[0] + 10*[-1] + 5*[-2]` (which is, too, a non-uniform distribution) from we randomly choose a value. The final flow through is thus `FLOW_THROUGH_BASE + random.choice(FLOW_THROUGH_DIFF)`.

Visualization After each simulation we record data of the vehicles. We use matplotlib and R to generate the visualizations. With matplotlib, we make plots of the average simulation score versus the number of vehicles for each model. The regions of the simulation score classes (4) are coloured for visual aid. With R we create density plots.

4 Results

In this section, we show the results of the models in the simulations. As mentioned before, for each traffic load we compute the average simulation score over a number of runs. For the results in this report, we have used the following configuration values:

```
# Simulation
RANDOM_SEED = 42
TRAFFIC_LOAD_NUM = 20
TRAFFIC_LOAD_START = 10
TRAFFIC_LOAD_END = 3000
SIMULATIONS_PER_TRAFFIC_LOAD = 100

# Grid
GRID_WIDTH = 10
GRID_HEIGHT = 10
```

```
# Road
ROAD_PROBABILITY = 0.9
ROAD_LENGTH_BASE = 1
ROAD_LENGTH_DIFF = 85*[0] + 10*[+1] + 5*[+2]

# Lane
LANE_PROBABILITY = 0.9
TRAFFIC_LIGHT_LENGTH = 1
FLOW_THROUGH_BASE = 8
FLOW_THROUGH_DIFF = 5*[+1] + 80*[0] + 10*[-1] + 5*[-2]

# Vehicle
VEHICLE_MIN_ROADS = (GRID_WIDTH+GRID_HEIGHT) // 2
VEHICLE_MAX_ROADS = (GRID_WIDTH+GRID_HEIGHT) * 2
```

As we want to test the models on low traffic load (`TRAFFIC_LOAD_START = 10`) up to high traffic load (`TRAFFIC_LOAD_END = 3000`), we should carefully consider what the `GRID_WIDTH` and `GRID_HEIGHT` values should be. If `GRID_WIDTH` and `GRID_HEIGHT` are low, then it gets already (too) busy with relatively low traffic loads, and so the models are not adequately tested on them. On the other hand, if `GRID_WIDTH` and `GRID_HEIGHT` are high, then it stays relatively quite, and so the models are not adequately tested on rush-hour situations. Hence, we consider `GRID_WIDTH = 10, GRID_HEIGHT = 10` to be a good compromise.

We start with a very small traffic load, but end with a traffic load of 3000 vehicles. Let's say the grid is maximally connected, that is, all intersections have incoming and outgoing roads for each neighbour, and all roads have the maximum number of lanes, then there will be approximately $\texttt{GRID\_WIDTH} \times \texttt{GRID\_HEIGHT} \times 4 \times 3 = 1200$ lanes. With 3000 vehicles, that is an average of 2.5 vehicles per lane (!), quite the busy city.

The number of simulations per traffic load should not be low as we are after a general performance. Setting it too high would be too time-consuming. Hence, we set `SIMULATIONS_PER_TRAFFIC_LOAD = 100`, which is a relatively high number and the simulations finish in a reasonable amount of time,

To allow some, but not too drastic variations in the connectivity of the grid we set `ROAD_PROBABILITY = 0.9` and `LANE_PROBABILITY = 0.9`.

We applied all the models (Clock, FCFS, Local Optimum and Global Optimum) on the different traffic loads. The results of the average simulation score over multiple runs versus number of vehicles are shown in Figure 5. We see that the Clock fully lies in the bad region and the FCFS model almost completely. The Local Optimum model lies for traffic load less than 2000 in the average region, and above in the bad. The Global Optimum lies for low traffic loads in the good region, and for a higher traffic load it follows the Local Optimum model.

For completeness, we show the average simulation score versus the number of vehicles for two different `FLOW_THROUGH_BASE` values, namely `5` and `11`, in Figure 6. They look similar to Figure 5, but the main difference is the intersection with the average/bad border at $simulation\_score = 1$ for the Local and Global Optimum models.

The distribution of the simulation score can be seen in Figure 7. While they are at different positions and of different sizes, all the shapes of the models are in fact quite similar. The shape resembles a normal distribution, albeit slightly stretched to the right.

To formalize whether there is a difference in the mean simulation score of the models, we perform an ANOVA test in R. More importantly, we also perform a paired t-test between the Global and Local Optimum models. Before we can compute the ANOVA, we check if our data meets the assumptions ANOVA. We pay special attention to two of them: (i) the dependent variable should be approximately normally distributed for each category of the independent variables, and (ii) we should have independence of observations [4]. We already saw that the dependent variable is approximately normally distributed for each model in Figure 7. Assumption (ii) means that there is no relationship between the observations in each model or between models themselves. This is also satisfied as each model is run on its own, that is independent of the others (it does not take into consideration what the other models do). With all assumptions met, we can perform the ANOVA test, which produces the following results:

```
               Df Sum Sq Mean Sq F value Pr(>F)
results\$model 3 1390 463.4 2352 <2e-16 ***
Residuals 7996 1575 0.2
---
Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

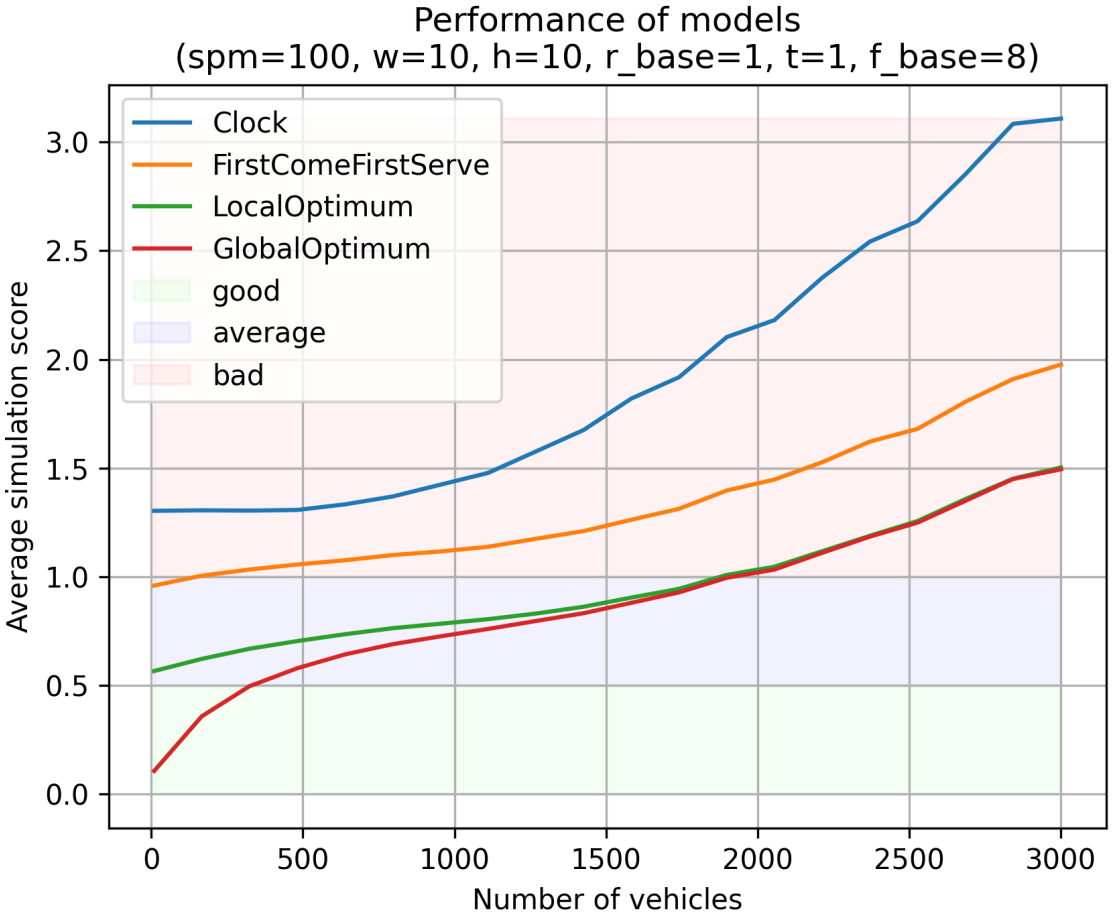
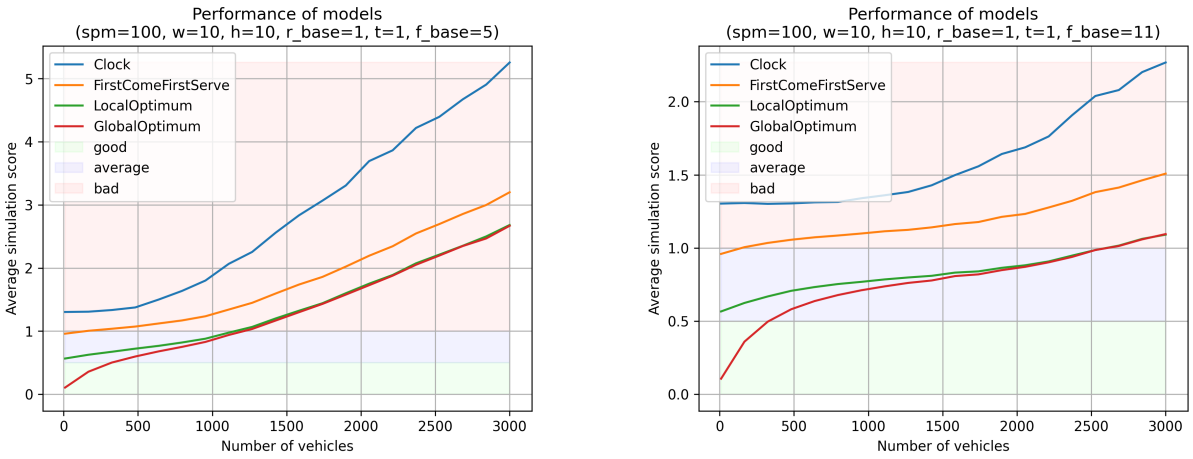


Figure 5: Average simulation score over multiple runs versus the number of vehicles.



(a) `FLOW_THROUGH_BASE = 5`.

(b) `FLOW_THROUGH_BASE = 11`.

Figure 6: Average simulation score over multiple runs versus the number of vehicles with different `FLOW_THROUGH_BASE` values.

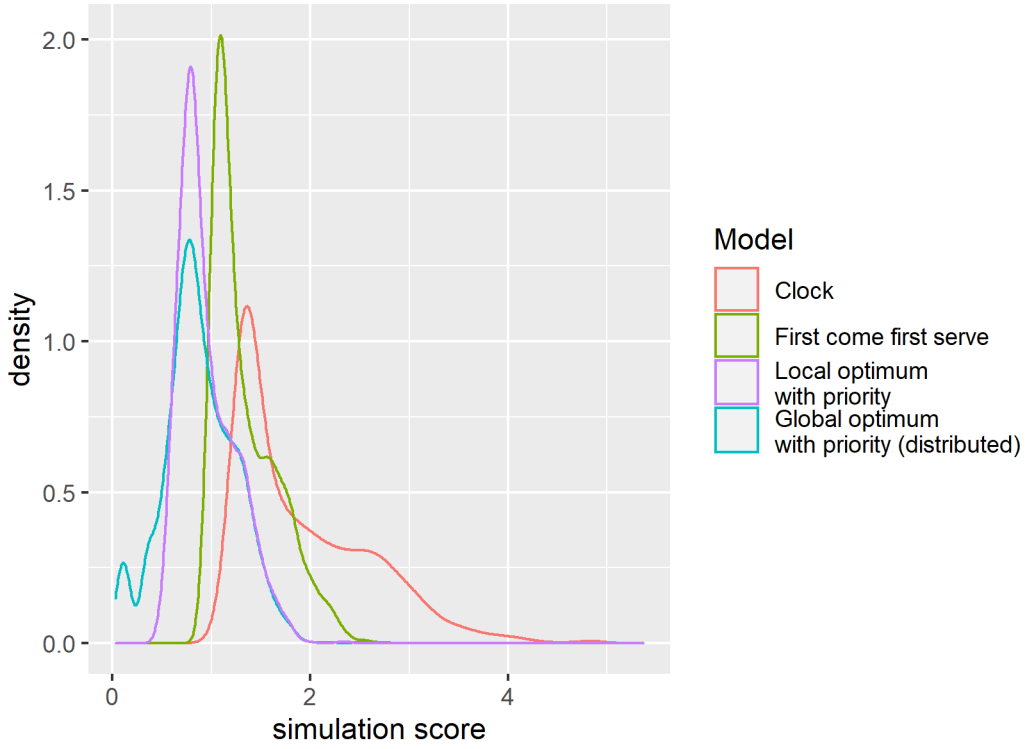


Figure 7: Comparison of mean simulation scores between the models in a density plot (with `F_BASE = 8`)

We observe that the p-value is smaller than $2 \cdot 10^{-16}$. Then, we compute the paired t-test between the Global Optimum and Local Optimum:

```
data: simulation_score_local_optimum and simulation_score_global_optimum
t = 28.839, df = 1999, p-value < 2.2e-16
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 0.06792691 0.07783953
sample estimates:
mean of the differences
             0.07288322
```

We note that the p-value is smaller than $2.2 \cdot 10^{-16}$ and that the mean of the differences is about 0.073.

5 Discussion

Looking at Figure 5, we can clearly see that the Clock model performs the worst out of all models. The Clock model lies fully in the bad region. This is because the Clock model does not consider the current traffic situation at an intersection. It just cycles through `TLConf`s without caring about its performance.

The FCFS model is the second worst performing model. It starts in the average region, but mainly lies in the bad region. It is better than the Clock model, because it does make use of the current traffic situation. One of the reasons why it does not perform any better is because of the first come first serve nature. If a lane was entered first with only one vehicle, it can go first, even though there might be a lane with far more vehicles.

The Local Optimum model is the second best performing model and it lies mainly in the average region. The problem of the FCFS is alleviated in this model. However, one of the reasons it performs worse for larger traffic loads, is that lanes with few vehicles waiting have to wait for the lanes with more. So it can take some time before they can actually cross the intersection.

The Global Optimum model is the best performing model. With low traffic load, there are not as many vehicles waiting in the lanes as with higher ones. So incoming vehicles have a higher chance of being able to continue driving without stopping for a red light. Note that with a higher traffic load, the Local and Global Optimum models have the same performance, albeit that Global Optimum performs every so slightly better.

This is because the traffic load is too high for the Global Optimum model to use the incoming vehicles as an advantage.

We can see that the simulation score goes up with a higher traffic load, but that is to be expected. Note again that even though all models are in the bad region with a traffic load of more than 2000, we can still say that the Global and Local models perform better than the FCFS and Clock.

From Figures 5 and 6 it is clear the flow through has a significant impact on the simulation score. With a lower flow through the overall simulation score is higher, and vice versa (note the y-axis). Increasing the flow through delays the moment at which the Local and Global models intersect the average/bad border, that is, it happens on a higher traffic load as it is able handle it better. We do, however, note that the shapes in the figures are equivalent and so the relative performance of the models is the same with a different flow through.

Looking at Figure 7 we see the same results but then in a density plot. The Global Optimum model has a small peak at a low simulation score. This corresponds to the part that lies in the good region. This plot also shows that the simulation scores go up with higher traffic load, as the right tail is somewhat stretched. As noted before, the simulation score approximates a normal distribution.

The ANOVA test shows we can reject the null hypothesis with confidence and so, as already noted from the plots, there is a significant difference in simulation score between the different models. The paired t-test on the Local and Global models confirms that the Global model performs better than the Local model. However, the mean difference in simulation score is about 0.073 which is a relatively small difference.

6 Conclusion

In this report we investigated if a global traffic light model performs better than local ones. We did this by employing the different models over multiple runs for different traffic loads. We saw that for low traffic loads the Global Optimum model outperforms the Local Optimum and the other models. For a higher traffic load it performs just as well as the Local model. All things considered, we conclude that a global (communication) network is better than local ones.

References

[1] Abdul Latif and Prisma Megantoro. "Traffic Light Regulatory System Based on Fuzzy Algorithm Using Microcontroller". In: *Journal of Physics: Conference Series* 1464 (Feb. 2020), p. 012034. DOI: 10.1088/1742-6596/1464/1/012034. URL: https://doi.org/10.1088%2F1742-6596%2F1464%2F1%2F012034.

[2] Rao Qian et al. "A Traffic Emission-saving Signal Timing Model for Urban Isolated Intersections". In: *Procedia - Social and Behavioral Sciences* 96 (2013). Intelligent and Integrated Sustainable Multimodal Transportation Systems Proceedings from the 13th COTA International Conference of Transportation Professionals (CICTP2013), pp. 2404–2413. ISSN: 1877-0428. DOI: https://doi.org/10.1016/j.sbspro.2013.08.269. URL: http://www.sciencedirect.com/science/article/pii/S1877042813023951.

[3] Ryan Bennink. "Red Light, Green Light: A Model of Traffic Signal Systems". In: *Pi Mu Epsilon Journal* 10.5 (1996), pp. 353–363. ISSN: 0031952X. URL: http://www.jstor.org/stable/24337934.

[4] Lund Research Ltd. *One-way ANOVA in SPSS Statistics*. https://statistics.laerd.com/spss-tutorials/one-way-anova-using-spss-statistics.php. Accessed: 2020-11-08.