Event Driven Model with an Objective to Control Traffic Lights in the Netherlands

Marko Boon[1], Mark van den Bosch[2], Paul Breeuwsma, Alessandro Di Bucchianico[1], Mona Emampour[1], Bart van Ginkel[3],
Tjebbe Hepkema[4], Philipp Holzinger[5], Rik Timmerman[1], Stephan Trenn[6]

Abstract

The study group participants of SWI 2020 with regard to the challenge proposed by the company Sweco were tasked to initiate a discrete-event dynamic model into Smart Traffic. Smart Traffic is cloud driven software developed by Sweco, implementing real-time predictive traffic signal control. Currently, the microscopic traffic simulator SUMO is being used within Smart Traffic to predict the traffic pattern for the short-term future, with the purpose of optimising traffic signal settings. However, in practice, microscopic traffic simulators appear to be too slow and hence infeasible considering its application. We employ discrete-event simulations as a tool to predict the future traffic state

[1]Eindhoven University of Technology, The Netherlands
[2]University of Leiden, The Netherlands
[3]Delft University of Technology, The Netherlands
[4]Utrecht University, The Netherlands
[5]Vienna University of Technology, Austria
[6]University of Groningen, The Netherlands

both efficiently and effectively, even though those type of simulations are usually employed in a different context.

We were in particular advised to focus on devising an event-driven model for a single, general intersection. This enabled us to create a thorough mathematical basic model. We are able to study various performance characteristics of the traffic light, such as the total delay or to the total squared delay.

Accompanied with the mathematical basic model, we deliver in correspondence a fully functioning program written in Python. Our article includes a detailed yet relatively simple example based on this program. This example additionally demonstrates the difference in an optimal outcome when using the total delay or the total squared delay.

Ultimately, we note that our model is easily extendable and several feasible extensions are proposed in this article.

Keywords: Smart Traffic, SUMO, Discrete-Event Simulation, Optimisation, Delay, Network, Control Applications

4.1 Introduction

Nowadays, everyone experiences traffic congestion – commonly characterised by longer trip times, slower speeds, and increased queuing of vehicles – which is severely problematic. Simply put, when traffic demand is big enough that the interaction between vehicles start to matter, the speed of the traffic flow is slowed and this will result in (some) congestion. Currently, the traffic demand is very high, as according to TomTom, there are more than a hundred cities around the globe where a trip during an arbitrary moment of the day has an increased duration of more than 30% compared to free flow TomTom International BV (2019, accessed December 20th 2019).

Sweco is an European engineering consultancy company who are mainly active in the fields of architecture, infrastructure and also in environmental and traffic engineering. One of their products which is currently in full development, is *Smart Traffic*: a real-time, data-driven software package that can be used to control traffic lights more efficiently, by making short-term predictions of the traffic pattern. Sweco

has already won prizes with this concept Sweco.nl (2019, accessed February 19 2020). Using information from detector loops and various other sources, Smart Traffic wants to achieve detailed predictions for upcoming traffic conditions, in order to control traffic lights to optimise future traffic flow. Improving the traffic flow yields a decrease in total delay and diminishes irritation levels, but simultaneously we will also be able to decrease the amount of vehicle emissions. It is commonly known that vehicles standing still and accelerating vehicles cause the most emissions, and Sweco expects that with Smart Traffic a decrease in emission of approximately 15% to 20% is possible. This is already achieved at some places in the Netherlands, e.g. in Helmond Sweco.nl (2020, accessed February 19 2020).

Smart Traffic is a tool to find good traffic light settings. Smart Traffic takes input from loop detectors and various other sources and predicts the future traffic state using various traffic light settings. Based on this, the choice for the best traffic light setting is made. This procedure is repeatedly executed, to make sure that a good performance is achieved. Currently, Smart Traffic is applied to each intersection separately.

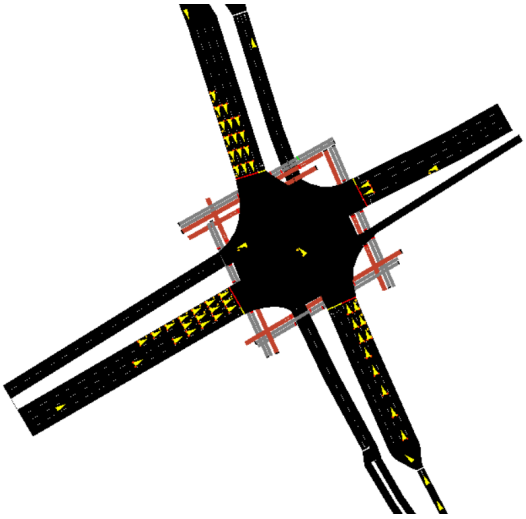

Figure 4.1: A fixed moment of a SUMO simulation. The intersection visualised, made by Sweco, represents the crossing of the 'Europaweg' and the 'Laan van Spitsbergen' in Apeldoorn.

One of the key tools that Sweco uses in their Smart Traffic application, is the microscopic traffic simulator SUMO Lopez et al. (2018). SUMO is able to accurately simulate traffic flows, taking vehicle-to-vehicle interactions into account. SUMO is one of the fastest microscopic traffic simulation tools, yet it is not fast enough for the purpose of Sweco, namely predicting the future traffic state with the purpose of operating nearly in real-time. Ideally, a whole network of intersections is considered, because the traffic demand and traffic light settings at one intersection might influence the demand and settings at others. Yet, because SUMO is not able to predict the future traffic state for a whole network within a reasonable time, these dependencies in the network are not taken into account.

In this project, we take a different approach and step away from the common traffic simulators. Instead, we apply discrete-event simulation. Discrete-event simulation has been widely applied in several technical applications, but so far it has received little attention in the traffic light control applications. An extensive overview of discrete-event simulation can be found in the monograph Cassandras and Lafortune (2009). Jang and Park (2018) applies discrete-event simulation to mission reliability of traffic lights rather than traffic flow control. Sumaryo, Halim, and Ramli (2013) develop an M/M/1 queuing model of a single junction with 4 entrances, each of them consisting of 2 lanes with signal groups. The simulation model is implemented using the SIMULINK environment and the SimEvents toolbox of MATLAB. No control scenarios are being investigated. Schanzenbacher et al. (2017) propose a mathematical model for an event-based train dynamics with one junction. By using the Max-plus algebra model, the average of the train frequency is a function of many parameters such as train travel time, the total number of trains on the line and, the number of trains on each branch. The results will be used for traffic control. Soh et al. (2013) mentions a simple model of discrete event simulation (DES). Applying the decomposition method from the queue theory caused a simple model. The method enables us to decompose a big scale network of roads into junctions and analyse them independently to come up with overall results for the whole system. Jagnere and Bansal (2013) also reports an event-driven simulation that updates the states of the system as well as the time when an event takes place. Cha and Mun

(2014) design a discrete event model to update the dynamic change of passengers for the Maglev, which is a new transportation system in Korea, by predicting the demand of passengers and making a plan on the operation of trains for each train station. Ben-Naoum et al. (1995) describes and compares three mathematical methods, Petri net, Max algebra, and Lagrangian relaxation, to model a discrete event. Zhang et al. (2019) points out a discrete event and hybrid traffic simulation by using MATLAB, Simulink and SimEvents for assessing an intelligent transportation system (ITS). Miller, Peng, and Bowman (2017) proposes a microscopic discrete event-based simulation to optimize the traffic system characteristics such as inter-arrival time and traffic light timing as well as forecasts the traffic system by applying ARIMA, regression and, neural network to build a more reliable traffic prediction system.

We will use discrete-event simulation, but apply it in a slightly different way than is usually done. Often, a discrete-event simulation is employed to study certain steady-state performance characteristics of the model at hand. However, we will use it as a tool to predict the future traffic state and to control the traffic lights. Indeed, this is one of the main advantages of using simulation compared to a mathematical analysis. A time-dependent (or *transient*) analysis of queuing systems is a notoriously hard problem, which is the reason why the vast majority of papers in the queuing literature focus on steady-state performance characteristics. Our discrete-event simulation is an excellent tool to study the transient behaviour of the system and make short-term predictions of the traffic behaviour.

The paper is organised as follows. In Section 4.2 we describe the problem. In Section 4.3 we subsequently introduce the model and assumptions. Moreover, we also give a detailed description of the implemented discrete-event simulation. In Section 4.4 we show in what manner the model can be used and the output that it produces. In the next section, we discuss several model extensions. Finally, in Section 4.6 we present conclusions and recommendations.

4.2 Context and Problem Description

Before we initiate and explain our basic model, we would like to briefly describe how Sweco works currently. First of all, the process of Smart Traffic with respect to the traffic light control can be divided into four main blocks: monitoring, forecast, controlling and communication.

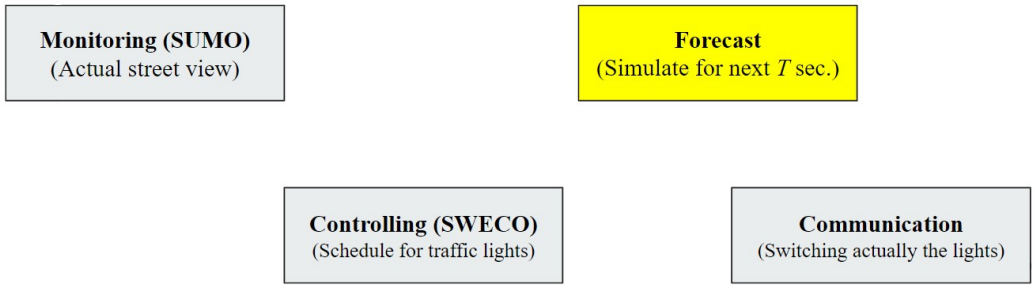

Figure 4.2: Current process of the traffic simulation and prediction framework at Smart Traffic.

The block *Controlling* is where everything starts and everything comes together. Sweco gathers all information they obtain from cameras on the street, loop detector information – which will be of major importance in our basic mathematical model – and data of local public transportation movements (e.g. buses). The block *Monitoring* is for simulating the actual *current* situation on the streets with the help of the software SUMO. *Forecast* is the block, where the prediction for the next T seconds is done. The last block *Communication* is then the execution of the decisions of Sweco, where the communication with the traffic lights is done.

Ultimately, Sweco wants to decide the schedule for the traffic lights at e.g., one junction. The above described processes are used to obtain this schedule. Sweco gathers the information and feeds the *Monitoring* block with all relevant data to simulate the current situation at the junction. At the same time they send numerous possible and useful schedules of traffic lights to the *Forecast* block. The *Forecast* block fetches the actual street view from the *Monitoring* block and starts to predict all needed data for all schedules, like dilation times, position of cars, or how many cars left the system, et cetera. All of this is currently

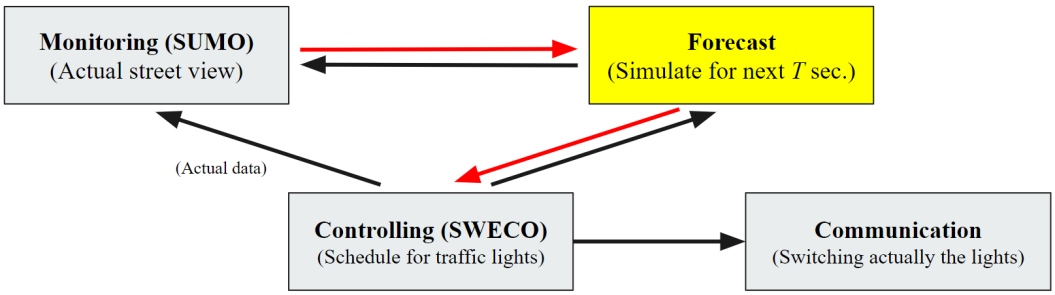

Figure 4.3: Current architecture of the traffic simulation framework at Smart Traffic.

done with the software SUMO. Then the *Forecast* sends all predictions back to the *Controlling* block and there Sweco decides what the best schedule is, and sends the best schedule to the *Communication* block, where the schedule will be delivered and executed on the traffic lights. This process happens every 3 seconds.

One of the problems here is the complicated connection between the three blocks, *Controlling*, *Forecast* and *Monitoring* and the involvement of SUMO in these three blocks. Since SUMO simulations are very detailed, taking e.g., vehicle-to-vehicle interactions into account, it produces more data than needed. This leads to high computation times and the application of SUMO in Smart Traffic is thus rather limited. One possible solution to this problem is merging the two blocks *Monitoring* and *Forecast*, and create a new program that does both at the same time, see the corresponding figure below.

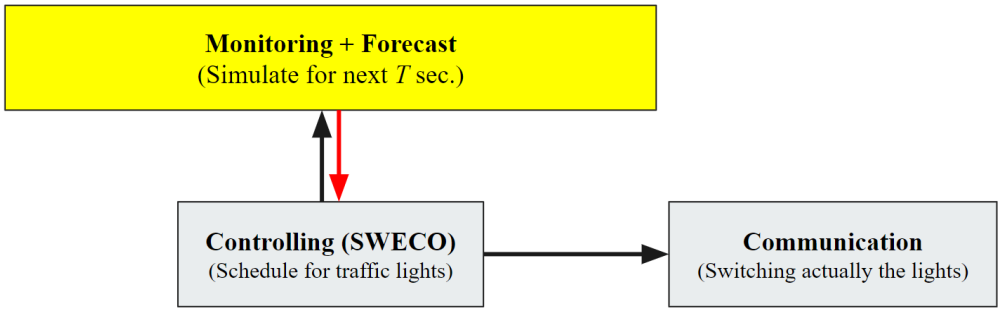

Figure 4.4: Goal architecture of the traffic simulation framework at Smart Traffic.

Moreover, SUMO is time based. This means that every time step every object in the simulation will be updated, which adds even more to the processing time, next to the many details that are already taken into account by SUMO.

The approach that we take here is not time based: we provide a simulation that is event based. Instead of checking what happens in every time step, we only check what happens if an event occurs, which might drastically reduce the number of updates in the simulation. A discrete-event simulation (DES) is a dynamic, asynchronous system, where the state transitions are initiated by events that occur at discrete instants of time. Typical examples of applications of DES are flexible manufacturing systems, telecommunication networks and multiprocessor operating systems. Intuitively formulated: the algorithm list all events that will happen, like changing of a traffic light, arrival times of particular cars, departure times of particular cars, etc., and sort them by their time. When an event is happening all objects that are influenced by that event will be updated and eventually new events will be added to the system and put in the list of events that will happen. With this new and simple approach, Sweco hopes for faster and more useful outputs.

4.3 Mathematical Basic Model Initiation

Before we are able to describe the basic model, see Subsection 4.3.3, we first need to initialise the setup: we give several definitions and point out major assumptions. Some assumptions are required for feasibility of the model whereas other assumptions could be relaxed upon. We refer to Subsection 4.5 for this.

In the Netherlands we usually have the following setup at an intersection. As is being illustrated in Figure 4.5, there are three types of sensors present on roads leading towards a signalised intersection, which are called *loops* in the jargon of Smart Traffic.

More specifically, at each lane we have the *enter loop* (red), the *long loop* (blue), and *head loop* (green). Whereas these are the official names for the loops, within this article we will be calling them the *arrival, long* and *departure loop* respectively. This suggestive terminology highlights

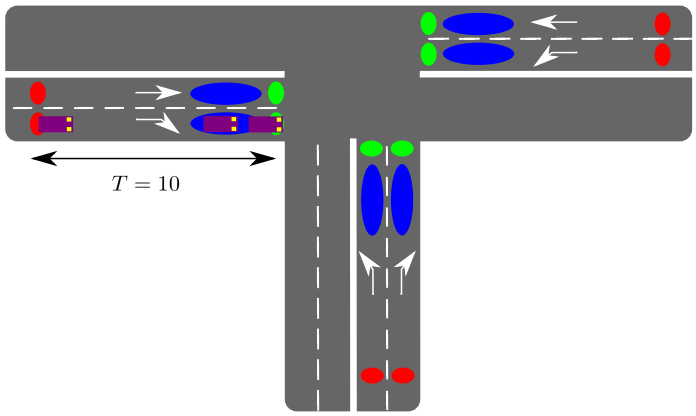

Figure 4.5: Visualisation of a T-junction with the arrival, long and departure loops represented by red, blue and green ellipsoids respectively. In this illustration, the (time-)distance between the red and green loop is $T = 10$.

that the arrival loop produces a signal whenever a car passes that loop and enters a (possible) queue on that lane. A departure loop, logically, relates to vehicles departing from the queue. To be a bit more precise, these two loops give us a value 1 when there is a car on top of the loop and gives us a value 0 when this is not the case, thus the change from 1 to 0 gives us the indication that a car has passed the loop.

In addition, the long loop outwards a signal, i.e. gives us an 1, whenever there is a car on top of it. Therefore, this loop tells us whether there are cars present in front of a traffic light or not. We would like to point out that the basic model does not use this loop, but we definitely recommend to use this loop which is further elaborated upon in Subsection 4.5.1.

Finally, according to Sweco, the arrival and departure loop on a lane are often at most $T = 10$ seconds apart one another, meaning that when you are allowed to drive with a speed at most 80 km/h – roughly 22 m/s – the distance between the two loops is at most 220 metres. This also explains the forecasting range of $T = 10$ seconds.

4.3.1 Definitions

Now that we have dealt with the setup of an intersection, we continue with some definitions which we use throughout this article.

Definition 4.3.1. A *system* is a formal description of the intersection, being the collection of lanes. Each *lane* consists of a traffic light, its direction arrow, and their three loops. A *signal group* is a collection of lanes with a coinciding direction arrow. This is visualised in Figure 4.6.

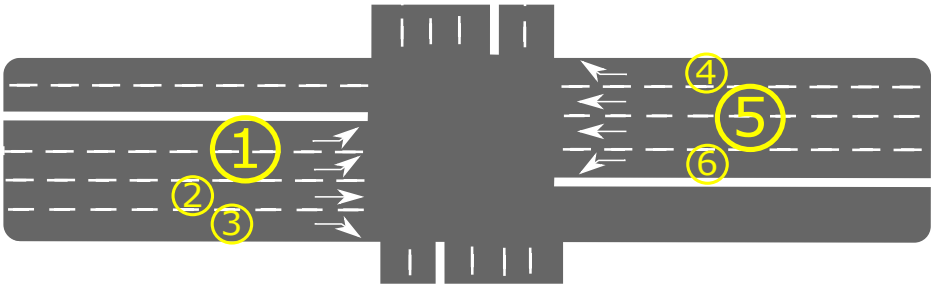

Figure 4.6: Sample of a system with six registered signal groups, where signal group 1 and 5 contain two lanes.

Definition 4.3.2. A *configuration* of a system at time t, denoted by X_t, is the state of all the traffic lights at time t of the intersection considered. A *scenario*, also called *stage*, with respect to T seconds is a possible sequence of configurations, i.e. a sequence $X = \{X_{t_k}\}_{k=0}^{N}$ with $N \in \mathbb{N}$ the amount of changes in the configuration from the start t_0 (which usually is equal to 0) within a time span of T seconds (and therefore $t_0 < t_1 < ... < t_N < T$).

In order to perform the forecasting of the next 10 seconds, we will be doing discrete-event simulations for a list of given scenarios, as requested by Smart Traffic. Additional input data that we take into account is the layout of the intersection; the traffic light configuration at t_0; and the arrival times of those cars who entered and not yet left the system up to time t_0, i.e. those cars that are beyond the arrival loop but have not yet passed the departure loop.

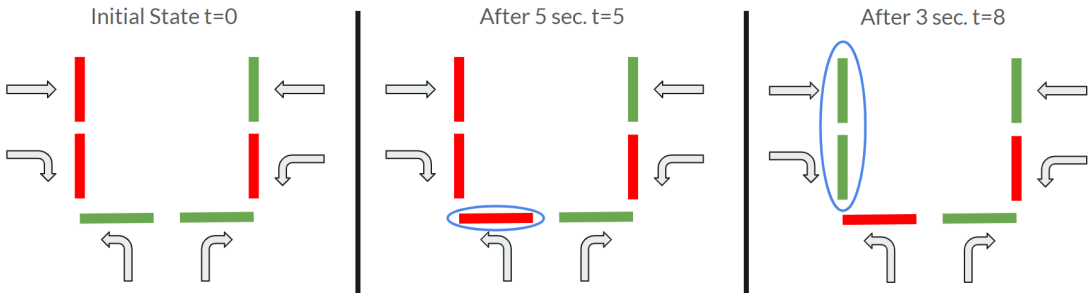

Figure 4.7: Schematic illustration, with $t_0 = 0$, of a traffic scenario with $T = 10$ seconds on a T-junction. Here we have $N = 2$ changes and the different configurations $X_0 = (0, 0, 1, 1, 0, 1)$; $X_5 = (0, 0, 0, 1, 0, 1)$; and $X_8 = (1, 1, 0, 1, 0, 1)$. Importantly observe we have for example $X_t = X_0$ for $0 < t < 5$.

As output data, we are interested in knowing the total (squared) delay per signal group and the *state* at both times t_0 and $t_0 + T$, i.e., the queue lengths per signal group at times t_0 and $t_0 + T$.

Definition 4.3.3. The *queue length* in a signal group is defined as the number of cars that are not driving in the corresponding signal group. The beginning of the queue is at the departure loop, whenever the queue length is non-zero. The end of the queue is not easily defined. We assume that anyone who has not yet crossed the intersection and who would have reached the departure loop if he/she would be traveling at the maximum speed, is in the queue.

Note that our definition of the queue length is an underestimation of the actual number of vehicles in the queue, as a vehicle is not able to drive up to the departure loop if there already is a queue.

An advantage of our use of discrete-event simulations is that we are able to keep track of each car individually, which is convenient for tracking the queue length and the delay, which is our next definition:

Definition 4.3.4. Consider some signal group S. Denote τ_S for the time a car needs to go through the system in an ideal situation, i.e. with green light and no other vehicles present. The *delay* of some car i at time t, denoted by $c_i(t)$, is defined as $\max\{0, \tau_i(t) - \tau_S\}$ where τ_i is the total time car i is in the system up to time t. Suppose at time

t_0 there are N cars in the signal group S, then the *total delay in* S at time t with $t_0 \leq t \leq t_0 + T$ is defined as the sum $\sum_{i=1}^{N} c_i(t)$. Let us denote

$$\text{delay}_S = \sum_{i=1}^{N} c_i(t).$$

Note that $\tau_i(t_1) = \tau_i(t_1 + \epsilon)$ for all $\epsilon > 0$ when the car has left the system at t_1. From a fuel consumption perspective the above definition is reasonable, but it does not take into account the "frustration" factor of car drivers who have to wait a very long time. In particular, when optimizing the traffic flow with respect to the delay above, it might happen, that one very busy signal group gets green all the time and a single vehicle has to wait very long. This problem can circumvented by choosing quadratic delays for each vehicle as a performance measure, i.e. we define the *squared total delay* by

$$\text{delay}_S^2 = \sum_{i=1}^{N} c_i(t)^2.$$

4.3.2 Assumptions

We note the following simplifying assumptions are implemented for the single-junction model, i.e. the basic model (see Subsection 4.3.3). If one would like to extend the basic model to one that covers multiple intersections, one should pursue mostly the same assumptions below. The main assumptions are:

1. *There is one type of vehicles present in the system.* We neglect other vehicles or road users, e.g. pedestrians, in the basic model (see also Subsection 4.5.5);

2. *Cars neither accelerate nor decelerate.* In other words, denote v_{cars} being the velocity of any car, it satisfies $v_{\text{cars}} = \max(0, v_{\max})$ with $v_{\max} \in \mathbb{R}_+$. This does not have any severe implications for e.g. the delay of a vehicle, as long as the departure time is chosen well, see Subsection 4.5.6 for an extension;

3. *Vehicles stick to their signal group.* The cars are assumed to stay in the same signal group, thus after being recognised at the arrival loop the car does not change to a lane with a different direction;

4. *Loop sensors work perfectly.* It is commonly known, according to Sweco, that loop sensors sometimes are wrong: sometimes a car is not recognised by a loop, the loop is damaged, or it gets accidentally triggered. This might cause an overestimation or underestimation of the number of cars present in the queue of a signal group. Despite the above, we neglect this fact as we do not have any detailed information on this (see also Subsection 4.5.1);

5. *During a simulation for T seconds, no cars enter the system.* In line with the perspective of Sweco, we only make predictions on aspects that we are certain of. This implies that we assume that the arrival detectors are not triggered for the prediction horizon T. Many existing (discrete-event) models let cars enter the system during a simulation according to a Poisson process Soh et al. (2013) and Sumaryo, Halim, and Ramli (2013), but we do not make such assumptions. Note this could (and perhaps also should) be implemented when the prediction horizon T is ex- tended or if e.g. a network of intersections is considered (see Subsection 4.5.3).

4.3.3 Basic Model

Based on the assumptions that we made in Subsection 4.3.2, we are able to formulate a model based on discrete-event simulation. We will first give an intuitive description of the basic idea of discrete-event simulation; then we explain how we use such a simulation to do predictions; present and explain all events that are in our implementation; and then finally present some pseudocode that describes the simulation.

The key idea in discrete-event simulation is that, even though events may happen at any moment in time, we do not continuously update the system. Only at the specific moment that an event occurs, we update the system. At such moments, we might also schedule new events, depending on the type of the event that has occurred. As an easy example, if the event is "arrival loop went off", we know that after

some travel time, the vehicle will either join the queue present in front of it (event "join the queue"), or it will trigger the departure loop when crossing the intersection (event "cross the intersection"). The events are thus the most important part of the model: as soon as all events are defined it is generally easy to implement the simulation.

To use discrete-event simulation, we initialise the system with the input data, like number of vehicles present between the arrival and departure loop and the moments at which those vehicles passed the departure detector and the configuration of the traffic lights. After this initialisation, we are able run the simulation for a prediction horizon T. Every time we need to make a prediction, we re-initialise the system in the same way, so using the same input data and a corresponding configuration.

We are able to define the events now. To model a general intersection, we have chosen to work with seven events. We give first the name of the event and subsequently provide a small description of the event.

- ARRIVAL_AT_SG: the event that a vehicle passes the arrival loop at a considerable (known) distance from the intersection of a certain signal group.
- ARRIVAL_AT_QUEUE: the event that a vehicle joins the queue at the stop line of the intersection.
- DEPARTURE_FROM_SG: the event that a vehicle departs from the signal group and crosses the intersection.
- DEPARTURE_FROM_QUEUE: the event that a vehicle departs from the queue (which happens right before DEPARTURE_FROM_SG)
- TORED: this event switches the state of the traffic light from amber to red.
- TOAMBER: this event switches the state of the traffic light from green to amber.
- TOGREEN: this event switches the state of the traffic light from red to green.

As the core part of the model is defined now (the events), we are able to provide a pseudocode version of our algorithm. We start with initialising the simulation, i.e. creating the junction, the signal groups and the future event set (FES), with the arrivals at the arrival loop that did not cross the departure loop yet and the switch events of the traffic lights. After the initialisation, we keep checking the FES until the prediction horizon has ended and the relevant items are updated. The description of the algorithm can be found in Algorithm 1.

One part of the input for the model is the configuration that has to be evaluated. These correspond to the event types TOGREEN, TOAMBER and TORED. Several performance measures can be used to evaluate the configuration. The two most important ones that we implemented are the number of vehicles in the queue and the (squared) delay of those vehicles. Depending on the event, we update those numbers or values. Based on all these values, Smart Traffic might base its decision on the allocation of the green times for the traffic lights. We will exploit this in a small case study in Section 4.4.

4.3.4 Rejected Ideas

The implemented algorithm builds up a queue from the available arrival times at the beginning of the simulation (by initiating a suitable cascade of events). Another idea we discussed was to initialize the simulation of a signal group with a certain queue length which is obtained from the simulation run in the past. While this approach probably leads to more accurate results, it complicates the algorithm and it is not clear how the actual measurements from the departure loop can be used to improve the simulation. Therefore, we decided against an implementation of a model with a given initial internal state, but rather just used the available data from measurements to build the initial internal state in an ad hoc way.

Furthermore, we also discussed to have multiple queues in one signal group and to keep track of the vehicles in the queue (in order to more accurately predict the time at which the vehicle leaves the signal group). While these extensions can easily be implemented in the event-based framework, we decided not to do so to keep the first model simple and also because it is not clear whether the actual simulation results would

Algorithm 1 Basic model – discrete-event simulation

```
T = 60 # length of prediction horizon;
t = 0 # starting time;
create junction with signal groups etc.;
create all events and add them to the events list (FES);
while t < T do
    select next event from event list (FES);
    save the time, t, of this event;
    if event type == ARRIVAL_AT_SG: # not needed now, probably
    needed when looking at a network of intersections then
        create vehicle with arrival time "time at red detector";
        add vehicle to signal group;
    else if event type == ARRIVAL_AT_QUEUE then
        if queue length of sg == 0 and its current traffic light color == green
        or amber then
            # the vehicle can just pass, hence they depart after driving from
            the loop at the stop line to the traffic light;
            schedule departure at time t + travelTime;
        else
            add vehicle to the queue;
            if there are more vehicles in the signal group that are not in the
            queue then
                schedule the event ARRIVAL_AT_QUEUE of the next vehicle;
            end if
        end if
    else if event type == DEPARTURE_FROM_QUEUE then
        remove first vehicle from queue;
        add the delay of the vehicle to the total intersection-wide delay;
        if queue length of sg > 0 and its current traffic light color == green
        then
            # there are still vehicles waiting and the light is green
            schedule departure at time t + reactionTime;
        end if
        # we also remove the vehicle from the signalgroup;
        schedule DEPARTURE_FROM_SG at time t;
    else if event type == DEPARTURE_FROM_SG then
        we remove the vehicle from the signal group;
    else if event type == TOGREEN then
        turn light color of signal group to green;
        if queue length of signal group > 0 then
            # there are vehicles waiting and the light just became green;
            schedule departure at time t + reactionTime;
        end if
    else if if event type == TOAMBER then
        turn light color of signal group to amber;
    else if event type == TORED then
        turn light color of signal group to red;
```

be more accurate (e.g., compared to simulations from SUMO).

4.4 Example Model Output

As ultimate goal we want to apply this model for relatively big intersections, like Sweco currently does for intersections in Apeldoorn, such as the one illustrated in Figure 4.1. In principle, our model can applied to such an intersection. However, to show some insights and as a proof-of-concept, we assume that there are two streams of vehicles, which share a common signalised intersection as in Figure 4.8.

4.4.1 Description of the Example

We assume that the current time is 0 and that all traffic lights are red at that time. All vehicles that we take into account, crossed the arrival detector at a negative time (we cannot look ahead, so any vehicles arriving after time 0 are not taken into account when doing the discrete-event simulation). At signal group 1, we have arrivals of vehicles at the following times:

$$\{-70, -69, -68, -29, -7, -5, -3\}.$$

At signal group 2, we have the following times of arrival:

$$\{-40, -35, -25, -15\}.$$

Note that some vehicles are already waiting for quite a while, whereas others are not even in the queue at time 0 (because the arrival time at the signal group is smaller than the time needed to travel through the whole signal group, which is five seconds in our example). If we perform our simulation, we simulate until a predetermined time, T, which we put to 60 in this example.
We evaluate three different configurations in this example, which are as follows:

1. Signal group 1 receives green until there are no vehicles anymore in the signal group. After that, we switch to signal group 2 after an amber period for signal group 1.

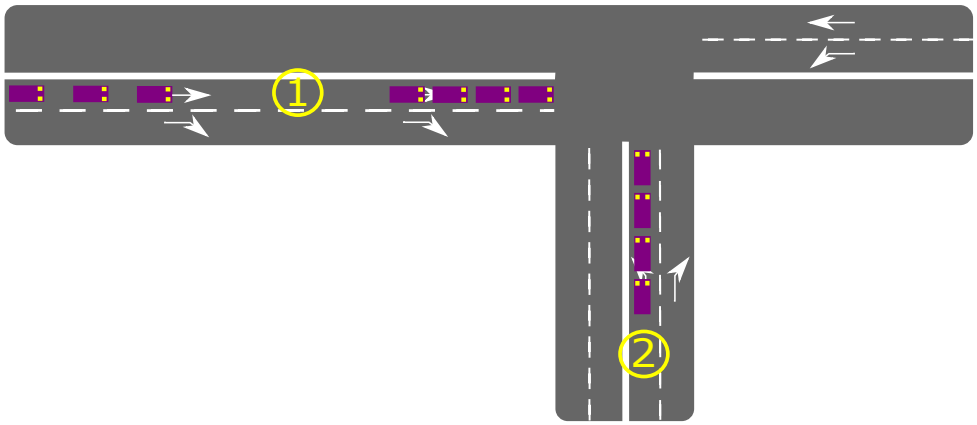

Figure 4.8: A schematic visualisation of the situation described above, where both signal groups are represented by its number. Observe that for this proof-of-concept example various intersection are probable, assuming that cars are not on the middle section of the junction when the other signal group gets green.

2. Signal group 2 receives green until there are no vehicles anymore in the signal group. After that, we switch to signal group 1 after an amber period for signal group 2.

3. Signal group 1 receives green until the first four vehicles from the signal group have crossed the intersection. Then we switch to signal group 2 after an amber period, because those vehicles have been waiting for a longer period than the last three vehicles in signal group 1. After signal group 2 has become empty, we switch back to signal group 1, again after an amber period.

4.4.2 Output of the Example

We present plots of the evolution of the queue length for both signal groups for each of the three traffic light configurations as described in the previous subsection (see Figures 4.9 until 4.11). The dots represent events, while the colour indicates the colour of the traffic light of the signal group. The height of the black line represents the number of vehicles waiting at the stop line to cross the intersection. The height of the gray coloured area is the number of vehicles present in the signal

group. Note that for signal group 1, this indicates that the last vehicle is not in the queue yet at time 0 (it arrives around time 2).

Figures 4.9 until 4.11 give a good indication of what happens at the intersection in the various configurations. It is easy to see if the configurations give reasonable behaviour, yet it is difficult to select the optimal settings based on those figures. For this purpose, we also keep track of both the delay and the squared delay.

The results for the delay and the squared delay are presented in Table 4.1. It is interesting to see that configuration 1 results in the smallest delay. This seems logical, as we start at the signal group with the most vehicles and stay there until all vehicles have left (and they therefore do not have to wait for e.g. an amber light). Only when the whole queue is cleared, we switch to the other signal group. This, probably, relates to a known result in queuing theory, stating that exhaustive strategies (keep serving until the queue is empty) are optimal for these kind of problems Liu, Nain, and Towsley (1992). This means that, once we start a green time at a signal group, we have to clear the queue there, which is a reasonable explanation why configuration 1 and configuration 2 are better than configuration 3.

This changes when we look at the squared delay. Then suddenly configuration 3 is optimal. This can probably be explained by the fact that a long delay becomes much larger once it is squared, whereas a small delay only increases only a bit when squaring. This means that the last three vehicles in signal group 1 have a smaller influence on the total squared delay than the vehicles in signal group 2. It is better to switch before clearing signal group 1 as a whole and to switch back after clearing the vehicles in signal group 2.

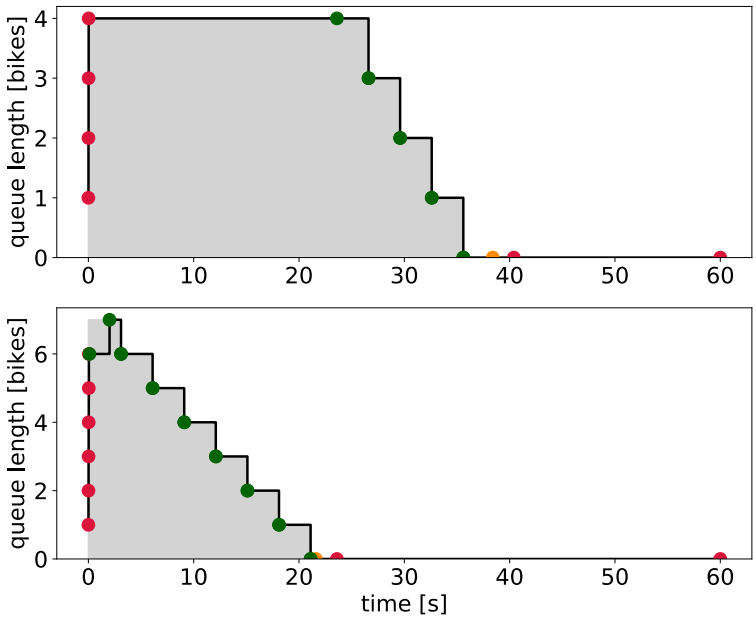

Figure 4.9: Evolution of the queues in configuration 1 (signal group 2 on top, signal group 1 at the bottom).

Table 4.1: Delay (in sec) and squared delay (in sec^2) for configurations 1, 2 and 3.

	delay sg 1	delay sg 2	total	delay^2 sg 1	delay^2 sg 2	total
Config. 1	300.77	219.44	**520.21**	17044.08	12197.23	29241.31
Config. 2	402.27	125.44	527.71	27238.16	4092.55	31330.71
Config. 3	350.27	183.84	534.11	19653.72	8608.04	**28261.76**

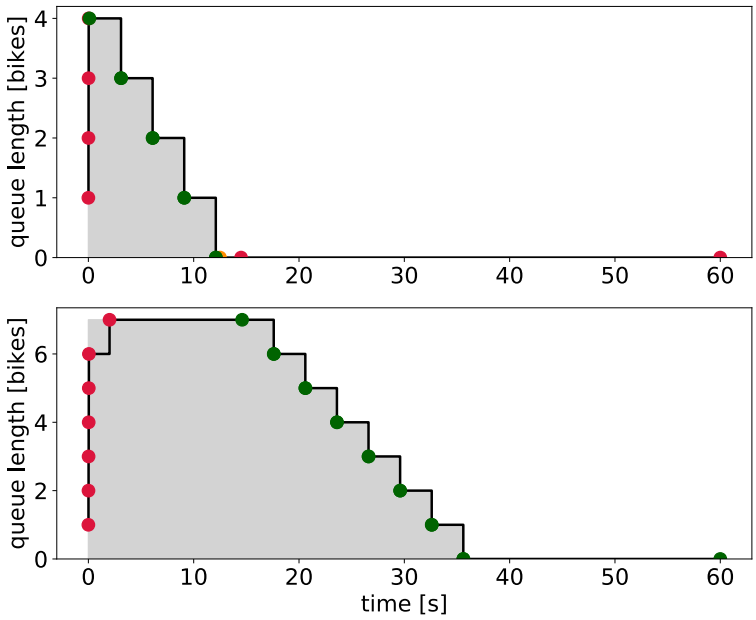

Figure 4.10: Evolution of the queues in configuration 2 (signal group 2 on top, signal group 1 at the bottom).

4.5 Extensions to the Model

The current implementation is a first basic model. This model can easily be improved upon or extended in a lot of ways. The main goal of this section is to record the ideas that were obtained during the week for such extensions (and sometimes for event-driven implementations). Sweco could use this list as a starting point for a further development of the basic model that we implemented.

In general we need to keep the goal in mind: we want to make an as good as possible prediction of the relevant performance characteristics, yet we want the computation time to be small. So, if an extension makes very accurate predictions about traffic behavior, but does not significantly improve the predicted delays, it mostly adds to the complexity and (therefore) computation time and should thus be discarded.

Below we describe some possible extensions with rough ideas of how to implement them.

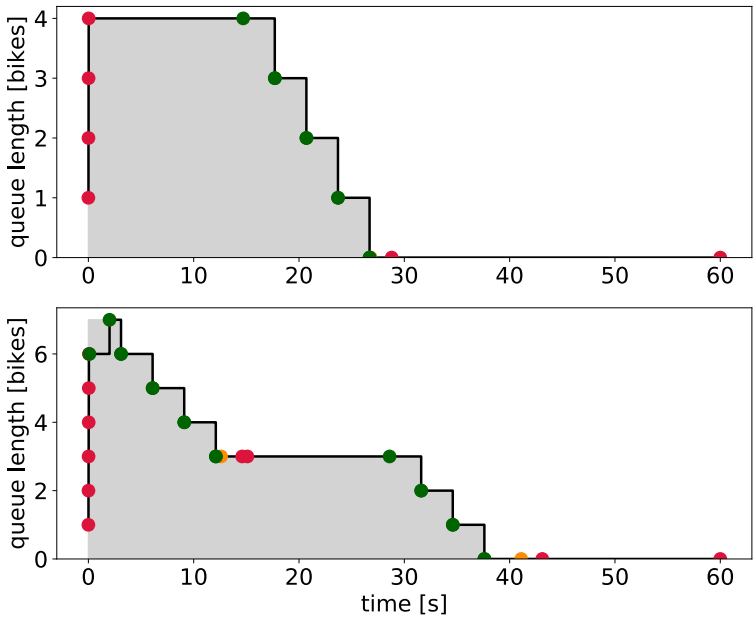

Figure 4.11: Evolution of the queues in configuration 3 (signal group 2 on top, signal group 1 at the bottom).

4.5.1 Usage of Long Loops

In the basic model the long loop is not used. The reason is that the only input information is the arrival and the departure loops and based on that, the current location of the cars at the starting time of the simulation is estimated. However, the long loops might be used by the controller to validate that the cars that should have left according to the model actually left. Moreover, one might want to use the information obtained from the blue loop to make a more accurate initialisation of the simulation.

4.5.2 Loops On and Off

In our model we regard the loop event as a moment in time when a car passes. In reality a loop changes value when a car is above it and changes back to the ground value when there is no car anymore. This contains more information than only the passing of a car. Indeed, one could consider to use the rest of the information for one or multiple

goals. Examples are:

- Estimate the speed of the car that is passing, this could enhance the prediction of when a car will arrive at the queue;

- Estimate the length of the vehicle (i.e. if all passing vehicles have a short signal and one has a long signal, it is likely that the change in sign length is not due to the speed of the vehicle (since the others around it went reasonably fast) but due to the length of the vehicle). This could help recognising for example trucks or buses and make a more accurate simulation of their future behaviour;

- In the extreme case that a loop stays on for a long time, this could indicate that the queue is so long that it has reached the arrival loop (at least the tail of the queue, the front might already be moving). This information may then be used to initialise the simulation in a better way.

4.5.3 Multiple Junctions: a Network

Once we are done with one junction, we can model a whole network of junctions. All junctions should do their independent simulations and computations. However, some information based on the network can be added. If we know in which direction a car leaves a certain junction, we can estimate when it will arrive at the next junction. If this arrival is to be expected within the next 10 seconds, it is relevant for the simulation of the next junction and should appear there (note that in a network setting, the 10 seconds might be extended to a longer time). Moreover, based on this information, the controller could possibly be improved. There are still some challenges left here.

First, cars could appear or disappear between junctions that are controlled by Sweco, since there might be intersections without traffic lights in between. Second, if a car comes from a nearby junction, it will trigger the arrival loop. We should be careful not to count such cars twice. Third, in principle we do not know in which direction cars leave a junction. Like in the extension in Subsection 4.5.4, we could e.g.

think of either assigning directions randomly based on known statistics (based on historical data).

Note that in this case, it might be worthwhile to add an extra output parameter: the amount of stops of a car during its journey. For reasons of convenience for the driver and for environmental reasons, a car should not have to stop too often. A possible mitigation strategy is that a car adjusts its speed before it would join a queue. In a context with several junctions the number of such steps and the mitigation strategy may become quite significant. Depending on the information available in the simulation, the amount of stops can be counted in different ways. If we only know which cars are in queues, we can count how often an individual car enters a queue. If we have a more advanced traffic simulation, we might actually count all stops of vehicles.

4.5.4 Conflicting Lanes

In the basic model, we model the signal groups separately. There is no interaction between them. In practice this is often true. Indeed, the controller could make sure that the considered configurations are valid and do not have conflicting green lights. An exception is the following: in some cases the cars that want to go to the left are given a green light together with the cars that want to go straight at the opposite side of the junction. In this case the cars that want to go left, have to wait for the cars from the opposite side. Such cases are not covered in the basic model.

The interaction that was just described can happen in several ways and can be dealt with in several way. We list some possibilities.

Situation 1: there are separate lanes for going left and going straight. In this situation we can simply tell the left lane to be "red" temporarily whenever the queue at the opposing straight lane is not empty.

Situation 2: there is a joint lane for going left and going straight.

- 2a: There is space in the middle of the junction for cars to wait (so cars behind it can pass). In this case a car going left first passes to the middle of the junction (if it is empty). The middle

of the junction can be modelled as a queue that can contain at most 1 (or maybe 2, depending on the junction) cars and that can be emptied whenever the opposing straight lane either is empty or has a red light. Any car going straight can still pass this car (or these cars) in the middle. However, when a second (or third) car tries to go left when the middle of the junction is full, the departure is cancelled (and hence all cars behind it automatically wait too). The departure of a car from the middle of the junction should in such a case schedule a new departure event from the junction and from the lane.

- 2b: There is no such space (i.e. if the car in the front waits, all the other cars wait). In this situation when a car tries to go to the left at the same moment that the opposing straight queue is not empty, the departure is cancelled. When the opposing queue becomes empty, it should schedule a departure event for the car that tries to go left. In this way all the cars behind the car trying to go to the left are waiting (including those that want to go straight).

This raises several conceptual/algorithmic problems:

- We only know where cars come from, not where they are going (so whether they go left or straight). There are multiple possible solutions:
  - Assign directions to cars randomly (according to some known statistics). The drawback is that the resulting calculated delay is subject to randomness. This could be dealt with by running the simulation many times and averaging, but of course this adds to the computation time.
  - Assigning an average delay to each car. However, it is not directly clear what kind of delay would be reasonable.
  - Working with non-integer amounts of cars (so a part of a car goes left, the rest goes straight). However, since in the current code each car is a separate object, this does not fit well in the current architecture of the software.

In conclusion, there a several ways to extend the basic model to account for conflicting traffic flows.

4.5.5 Special Road Users

In the current model all the road users are treated the same. In reality, this is not what we want. To make the model more realistic, one could add the following additional road users:

- Buses/trams
- Cyclists
- Pedestrians
- Emergency services
- Motorbikes
- Heavy vehicles (such as trucks)

With each of these road users, two questions have to be answered. First: how do they react to the traffic lights (and possibly to other traffic)? And second: what information do we have about their arrival time at the traffic lights? Additionally, for some road users their delay is more relevant then for others (public transport and emergency services should get priority). We briefly discuss all the listed additional road users. All of them can be implemented as different "car-like" objects in the model.

Buses/trams. For buses there are two different possibilities. First of all, they could be on the same lane as the rest of the traffic. In this case, they can basically be treated as a car in how they react to the traffic lights and traffic (although in case there are "heavy vehicles" is implemented, buses should behave like one of those).

It could also be the case that buses have a separate signal group. In this case they can be directed directly to this lane. In that sense they are similar to trams, since they also have their own "lane" and signal group. In this case, it is easy to prioritize them.

Then we need to answer the question how to schedule arrival events for these objects. For buses (and probably trams) we have more information then for (non-smart) cars. They can e.g. tell us where they are. This means that we can schedule their arrival at the junction in advance, e.g. by using this distance and an average speed. However, a

bus will trigger the red loop as well. To solve this issue, one of the red loop events should be assigned to be the bus.

As we have the above information, we might also prioritize such vehicles. The controller should account for this.

Cyclists. Since bicycles often have separate lanes, they can be added easily as extra signal groups to the junction. The only complexity here is that cars turning right might get a green light at the same time when cyclists going straight get green. This is a conflict of paths, which is dealt with in the extension in Subsection 4.5.4.

The detection of bikes can be similar to the detection of cars (so a loop at a distance and loops at the light itself). In this case the arrival of bikes can be predicted using the loop at a distance. If the arrival loops for bicycles are absent, no arrivals events are taken into account and only the presence or absence of cyclists waiting at the traffic light at the start of the simulation.

In case the cyclists are using an app like Schwung on their phones, this information can be used to schedule future arrivals (using the distance of the cyclist to the light at the time of the start of the simulation and an average cycling speed).

Pedestrians. This is similar to bikes, except that there is no way to detect them with loops. This means that the only information that is known at the start of the simulation is whether there are pedestrians waiting or not (we might obtain this information from the button at which pedestrians usually can press when they are at a signalized intersection).

Emergency services. These vehicles (ambulances/fire trucks/police cars) can be part of the simulations, but this might not be necessary. Since these vehicles have absolute priority, the controller might want to take this into account by giving a green light to this vehicle as soon as possible. In this way the traffic keeps flowing in the direction that the emergency vehicle wants to go and there is no crossing traffic.

Motorbikes/heavy vehicles. Although this category is quite broad, the main point is that the behaviour is different from cars in a way that is determined by certain parameters. Here one can think of the average speed, acceleration pattern or the reaction time at the traffic light (the time between the vehicle knowing that it may go and its actual departure, see also Subsection 4.5.6).

In either of these two cases the main problem would be the detection of a special vehicle. It is not very straightforward to known from the loop information whether a passing vehicle might be a truck or a motorbike, so making a distinction is difficult, and hence making a distinction in the simulation is difficult as well.

4.5.6 Non-constant Inter-Departure Times

In the basic model, cars pass a green traffic light with a constant time between them. To be precise: whenever a car passes the traffic light, a next departure event is scheduled after a fixed amount of time. In practice this is not true. Due to the acceleration of vehicles, the time between the first and the second car is rather large (since speeds are low), and the time between later cars should be shorter (since they already travel at a higher speed). Therefore we propose the following extension:

- At a switch event where the light switches to green, the current time is recorded (as an attribute of the signal group object), we could call it t_{green}.
- When a departure event occurs, a new departure event is scheduled. This is done at time $t+f(t-t_{\text{green}})$, where f is a pre-specified (often decreasing) function.

We make the following remarks. First of all, this function f can be determined in a number of ways. We could look at the data obtained from practice. Alternatively, we could use SUMO or even make our own computations. Bottom line is that this function f is very easy to include and to adjust. Second, it should be noted that it could be that a light is already green at the start of the simulation. Therefore as extra input data, we need to know from the past when all lights turned green if they are green at the moment that the simulation starts.

4.5.7 Start/End Queues

Sweco is also interested in the length of the various queues in the configurations that are tested. For this they have two different reasons,

which could lead to two different definitions of queue lengths (one in number of vehicles, the other in meters).

The first reason for the interest of Sweco is that based on queue lengths the controller can decide how long a traffic light should remain green. In this sense, the corresponding queue length is the amount of cars that want to pass the green light. This queue length is rather easy to obtain from the simulation, since we already keep track of this.

The second reason for Sweco to know what the queue length is, has to do with smart traffic applications. Sweco would like to know where traffic is standing still to be able to inform smart cars about this delay and help them plan their routes in a clever way. For this reason they would like to know where the queues are and where slow driving traffic is present. The slow traffic is not only situated just before the traffic lights, since when standing traffic starts to move, it takes a while for traffic at the back of the queue to start moving. We should measure this queue length in meters or in the location where the queue begins and ends. In order to know the latter, it is important to model how cars behave when cars in front of them stop or start moving. One could use SUMO to obtain such information.

4.6 Conclusions and Recommendations

We have suggested and implemented a basic model, based on discrete-event simulation, as solution to the computationally intractable problem of controlling traffic lights that Sweco currently is facing when applying the general traffic simulator SUMO.

Accompanied we have a fully operational program written in Python that can be used as guidance for implementing it in Smart Traffic. For instance, coding our mathematical basic model gave us insight in the computational complexity of it and several interesting insights into optimal traffic light settings. It is noteworthy that there might be a considerable difference when looking at the total delay or the total squared delay, as elaborated upon in Section 4.4.

Besides the latter revelation, we have not been able to thoroughly test our model because of time-constraints. Yet, some suggestions are made on how to tackle several problems/extensions in Section 4.5. This

shows that our discrete-event simulation is very flexible and can be made even more realistic when desiring to mimic actual traffic behaviour with a limited influence on the computational complexity.

Overall, we recommend Sweco to implement the concepts described in this article, which are based on discrete-event simulation. This implies that Sweco should discard their current approach. Our basic model is easily extendable and, most importantly, it is way faster in comparison with the current model, both based on the comments of Sweco and our own findings.

References

Ben-Naoum, L et al. (1995). "Methodologies for discrete event dynamic systems: A survey". In: *Journal A* 36.4, pp. 3–14.

Cassandras, C. G. and S. Lafortune (2009). *Introduction to Discrete Event Systems*. 2nd. Springer.

Cha, Moo Hyun and Duhwan Mun (2014). "Discrete event simulation of Maglev transport considering traffic waves". In: *Journal of Computational Design and Engineering* 1.4, pp. 233–242.

Jagnere, P. and A. Bansal (2013). "Road Traffic Simulation - A Discrete Event Driven Model". In: *International Conference on Reliability, Infocom Technologies and Optimization (ICRITO - 2013)*.

Jang, J.S. and S.C. Park (2018). "Discrete Event Simulation-Based Reliability Evaluation of a Traffic Signal Controller". In: *Modelling and Simulation in Engineering*.

Liu, Z., P. Nain, and D. Towsley (1992). "On optimal polling policies". In: *Queueing Systems* 11.1-2, pp. 59–83.

Lopez, P. A. et al. (2018). "Microscopic traffic simulation using SUMO". In: *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, pp. 2575–2582.

Miller, John A, Hao Peng, and Casey N Bowman (2017). "Advanced tutorial on microscopic discrete-event traffic simulation". In: *2017 Winter Simulation Conference (WSC)*. IEEE, pp. 705–719.

Schanzenbacher, F. et al. (Dec. 2017). "A discrete event traffic model explaining the traffic phases of the train dynamics in a metro line

system with a junction". In: *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*.

Soh, A. C. et al. (2013). "A discrete-event traffic simulation model for multilane-multiple intersection". In: *2013 9th Asian Control Conference (ASCC)*. IEEE, pp. 1–7.

Sumaryo, S., A. Halim, and K. Ramli (2013). "Improved discrete event simulation model of traffic light control on a single intersection". In: *2013 International Conference on QiR*. IEEE, pp. 116–120.

Sweco.nl (2020, accessed February 19 2020). *Smart Mobility voor een leefbare stad en minder CO2-uitstoot | Sweco.nl*. URL: `https://www.sweco.nl/innovaties/Smart-Mobility-voor-een-leefbare-stad%20-en-minder-CO2-uitstoot/`.

— (2019, accessed February 19 2020). *Smart Traffic can Sweco wint Cobouw Innovatieprijs | Sweco.nl*. URL: `https://www.sweco.nl/nieuws/nieuwsartikelen/smart-traffic-van-sweco-%20wint-cobouw-innovatieprijs/`.

TomTom International BV (2019, accessed December 20th 2019). *Traffic congestion ranking | TomTom Traffic Index*. URL: `https://www.tomtom.com/en%5C_gb/traffic-index/ranking/`.

Zhang, Yue et al. (2019). "A discrete-event and hybrid traffic simulation model based on SimEvents for intelligent transportation system analysis in Mcity". In: *Discrete Event Dynamic Systems* 29.3, pp. 265–295.

4.A Appendix: Code

The code of the model can be found at:
https://bitbucket.org/Tjebbe/traffic/.

4.A.1 Main simulation files

run_simulation.py

The main file of the simulations, which loads the input data, sets up the needed data-structures, starts the simulations and produces the output files.

```
1  from JunktionSimulation import JunktionSimulation
2  from SWI_Traffic_Objects import SignalGroup
3  from PlotResults import PlotResults
4  import json, datetime, os, errno, sys
5
6  # Make folder for the output
7  datestamp = datetime.datetime.now().strftime("%d_%b_%Y_(%Hh%M)")
8  outputfolder = 'output/output_{}/'.format(datestamp)
9  try:
10     os.makedirs(outputfolder)
11 except OSError as e:
12     if e.errno != errno.EEXIST:
13         raise
14
15 # Read junction layout from json file.
16 with open('input/junction.json','r') as f:
17    junctionData = json.load(f)
18
19 sgIDs = junctionData["sgIDs"]
20
21 # Read scenario from input json files.
22 # Alternatively one could make de input files .py files and just
       import the dictionaries directly.
23 with open('input/scenario1.json', 'r') as f:
24     scenario1 = json.load(f)
25 with open('input/scenario2.json', 'r') as f:
26     scenario2 = json.load(f)
27 with open('input/scenario3.json', 'r') as f:
28     scenario3 = json.load(f)
29 with open('input/scenario4.json', 'r') as f:
30     scenario4 = json.load(f)
31 with open('input/scenario5.json', 'r') as f:
32     scenario5 = json.load(f)
33
34 scenarios = [scenario1, scenario2, scenario3, scenario4,
       scenario5]
35
36 # Read arrival times and time it takes to enter and leave sg
       when light is green
37 with open('input/arrivalTimes.json', 'r') as f:
38     arrivalTimes = json.load(f)
39
40 print('running...', end='')
41 sys.stdout.flush()
42 i = 0
43 for scenario in scenarios:
44     # make signal groups and add them to the junction list
45     # note that after one scenario the sg still has its history
           of the previous.
46     # here we overwrite it (that is why this is in the for loop;
           maybe improve later).
47     junction = []
48     for sgID in sgIDs:
49         sg = SignalGroup(sgID, arrivalTimes[sgID]["travelTime"],
               arrivalTimes[sgID]["arrivalTimes"])
```

```
50          # add signalgroups to junction
51          junction.append(sg)
52
53      # make class for simulation
54      sim = JunktionSimulation()
55
56      # run simulation
57      RunTime = 60        # run time in seconds
58      res = sim.simulate(RunTime, junction, scenario, verbose=
            False)
59
60      #write output to file
61      with open(outputfolder + 'output.txt','a+') as f:
62          f.write('Senario {}'.format(i+1))
63          f.write(' ({})\n\n'.format(scenario["description"]))
64          sumTotalDelay = 0
65          for sg in junction:
66              f.write('Total delay at signal group {}: {:5.2f}
                    sec\n'.format(sg.identifier, res[sg.identifier
                    ].totalDelay))
67              sumTotalDelay += res[sg.identifier].totalDelay
68          f.write('-----------------------------------------+\
                n')
69          f.write('                                    {:5.2f} sec\n'
                .format(sumTotalDelay))
70          f.write('\n')
71          sumQuadraticDelay = 0
72          for sg in junction:
73              f.write('Quadratic delay at signal group {}: {:5.2f
                    } sec^2\n'.format(sg.identifier, res[sg.
                    identifier].quadraticDelay))
74              sumQuadraticDelay += res[sg.identifier].
                    quadraticDelay
75          f.write('
                -----------------------------------------------
                +\n')
76          f.write('                                           {:5.2
                f} sec^2\n'.format(sumQuadraticDelay))
77          f.write('\n')
78      #plots
79      plot = PlotResults(res)
80      plot.plotQueueLengthVsTime(outputfolder + 'scenario_{}.pdf'.
            format(str(i+1))) # save plots
81      # plot.plotQueueLengthVsTime()  # show plots
82
83      i += 1
84  print('finished!')
```

JunktionSimulation.py

The event-driven simulation which goes through the event list, updates the internal state accordingly and adds new events to the event list.

```
1  from scipy import stats
2  from collections import deque
3  # from dist.Distribution import Distribution
4  from ggcQL.Event import Event
5  from ggcQL.FES import FES
6  from ggcQL.SimResults import SimResults
7  from SWI_Traffic_Objects import SignalGroup, Bicycle
8
9  class JunktionSimulation :
10
11     def importTrafficLightEvents(self, fes, sg, times_turn_green
           , times_turn_amber, times_turn_red):
12         """
13         add traffic light events of signal group sg to event
               list fes
14         """
15         numGreens = len(times_turn_green)
16         numAmbers = len(times_turn_amber)
17         numReds = len(times_turn_red)
18
19         # initialize traffic light changes
20         for i in range(numGreens):
21             newEvent = Event(Event.TOGREEN, times_turn_green[i],
                   sg)
22             fes.add(newEvent)
23         for i in range(numAmbers):
24             newEvent = Event(Event.TOAMBER, times_turn_amber[i],
                   sg)
25             fes.add(newEvent)
26         for i in range(numReds):
27             newEvent = Event(Event.TORED, times_turn_red[i], sg)
28             fes.add(newEvent)
29
30
31     def simulate(self, T, junction, scenario, verbose=False):
32         """
33         main simulation
34         """
35         t = 0                  # current time
36         smalldelay = 1e-2      # to schedule events 'now', but
               still after the current one
37
38         fes = FES()
39
40         res = {}  # results keys are identifier of sg
41
42         for sg in junction:
43
44             # add traffic light events of signal group sg to
                   event list fes
45             times_turn_green = scenario[sg.identifier]['
                   times_turn_green']
46             times_turn_amber = scenario[sg.identifier]['
                   times_turn_amber']
```

```
47              times_turn_red    = scenario[sg.identifier]['
                    times_turn_red']
48              self.importTrafficLightEvents(fes,sg,
                    times_turn_green, times_turn_amber,
                    times_turn_red)
49
50              sg.initializeSimulatedBikes()       # copy real bikes
                     to sim bikes
51
52              res[sg.identifier] = SimResults()  # class for
                    results
53
54              fes.add(sg.initialEvent(t))         # add initial
                    event to fes
55
56              # add closig event at time T
57              closingEventAttimeT = Event(Event.STOP_SIMULATION,T,
                    sg)
58              fes.add(closingEventAttimeT)
59
60          while any([sg.finished == False for sg in junction]):
61              e = fes.next()                          # jump to next
                    event
62              t = e.time                              # time of event
63              sg = e.sg
64
65              if verbose:                               # signal
                    group of event
66                  print('-------------------------------------------------')
67                  print('new event: ', end='')
68                  print(e)
69                  print('before event: number of sim bikes = {}'.
                        format(sg.numberSimBicycles()))
70                  print('before event: queue length = {}'.format(
                        sg.queueLength()))
71                  print('before event: traffic color = {}'.format(
                        sg.currentTrafficLight()))
72
73              if e.type == Event.ARRIVAL_AT_SG :
74                  # event is an arrival at signal group:
75                  assert False, 'event is arrival to signal group
                        (not allowed for now!)'
76
77              if e.type == Event.ARRIVAL_AT_QUEUE :
78                  # event is an arrival at queue
79
80                  bicycle = e.bicycle                     # bicycle
                         from event
81                  if sg.queueLength() == 0 and sg.
                        currentTrafficLight() in [sg.GREEN, sg.AMBER
                        ]:
82                      # if queue lenght is zero, dont add to queue
                             but drive through
83                      # schedule departure from signal group
```

```
84                      dep = Event(Event.DEPARTURE_FROM_SG, t +
                            smalldelay, sg, bicycle)
85                      fes.add(dep)
86                  else:
87                      # else add to queue
88                      sg.addBicycleToQueue(bicycle)           # add
                            bicycle to queue
89
90                      nextBikeInSignalGroup = sg.nextBicycle(
                            bicycle)
91                      if nextBikeInSignalGroup != None:
92                          # if there are more bicycles in sg
                                schedule the next arrival to queue
93                          timeToNewEvent = max(0,sg.travelTime - (
                                t-nextBikeInSignalGroup.arrivalTime)
                                )
94
95                          arrival = Event(Event.ARRIVAL_AT_QUEUE,
                                t + timeToNewEvent + smalldelay , sg
                                , nextBikeInSignalGroup)
96                          fes.add(arrival)
97
98              elif e.type == Event.DEPARTURE_FROM_SG :
99                  # event is departure from signal group
100
101                 # remove bicycle from signal group
102                 removedBicycle = sg.removeFirstBicycle(t)
103                 # check if bicycle of event is in front of queue
104                 assert (e.bicycle == None or e.bicycle ==
                        removedBicycle), 'removed bicycle is not the
                        first one'
105
106                 # add delay of removed bicycle to total delay
107                 res[sg.identifier].registerTotalDelay(
                        removedBicycle)
108
109
110             elif e.type == Event.DEPARTURE_FROM_QUEUE :
111                 # event is departure
112
113                 if (sg.currentTrafficLight() == sg.GREEN):
114                     # remove bicycle from queue
115                     removedBicycle = sg.
                            removeFirstBicycleInQueue(t)
116                     # check if bicycle of event is in front of
                            queue
117                     assert (e.bicycle == None or e.bicycle ==
                            removedBicycle), 'removed bicycle is not
                            the first one'
118
119                     # schedule departure from signal group
120                     dep = Event(Event.DEPARTURE_FROM_SG, t +
                            smalldelay, sg, removedBicycle)
121                     fes.add(dep)
122                 if (
```

```
123                     sg.queueLength() > 0
124                     and (sg.currentTrafficLight() == sg.GREEN)
125                 ):
126                     # as long as the queue is nonempty and the
                            light is green
127                     # make event for departure at time t +
                            reaction time of the bicycles
128                     nextBicycleInQueue = sg.
                            getFirstBicycleInQueue()
129                     dep = Event(Event.DEPARTURE_FROM_QUEUE, t +
                            Bicycle.reactionTime, sg,
                            nextBicycleInQueue)
130                     fes.add(dep)
131
132
133             elif e.type == Event.TOGREEN:
134                 # event is traffic light changed to green
135
136                 sg.changeTrafficLight(sg.GREEN,t)       # change
                        trafficlight to green
137
138                 if sg.queueLength() > 0:
139                     # if light turns green and bicycles are
                            waiting schedule departure
140                     dep = Event(Event.DEPARTURE_FROM_QUEUE, t +
                            Bicycle.reactionTime,sg)
141                     fes.add(dep)
142
143             elif e.type == Event.TORED:                 # event is
                    traffic light changed to red
144                 sg.changeTrafficLight(sg.RED,t)         # change
                        trafficlight to red
145
146             elif e.type == Event.TOAMBER:               # event is
                    traffic light changed to amber
147                 sg.changeTrafficLight(sg.AMBER,t)       # change
                        trafficlight to amber
148
149             elif e.type == Event.STOP_SIMULATION:       #
                    simulation end reached
150                 # remove all bikes from signal-group (which
                        calculates the delays), add a delay if they
                        are in a queue and add their delay
151                 # ATTENTION: not feasible if the simulation
                        should be continued afterwards with a warm
                        start
152                 cumulativeDelay = 0 #addded up reaction times
                        for bikes in the queue
153                 while sg.numberSimBicycles() > 0:
154                     removedBicycle = sg.removeFirstBicycle(t)
155                     firstBikeInQueue = sg.getFirstBicycleInQueue
                            ()
156                     if removedBicycle == firstBikeInQueue:
157                         cumulativeDelay += Bicycle.reactionTime
158                         removedBicycle.delay += cumulativeDelay
```

```
159                         sg.removeFirstBicycleInQueue
160
161                     res[sg.identifier].registerTotalDelay(
                            removedBicycle)
162
163                 # set signal flag to finish
164                 sg.finished = True
165
166
167             if verbose:
168                 print('after event: number of sim bikes = {}'.
                        format(sg.numberSimBicycles()))
169                 print('after event: queue length = {}'.format(sg
                        .queueLength()))
170                 print('after event: traffic color = {}'.format(
                        sg.currentTrafficLight()))
171
172             res[sg.identifier].registerSignalGroupLength(t, sg)
                    # register the number of bikes in signal group
                    at time t
173             res[sg.identifier].registerQueueLength(t, sg)
                    # register the queue length after event at time
                    t
174             res[sg.identifier].registerTrafficLightColor(t,sg)
                    # register traffic light color after event at
                    time t
175
176         return res
```

PlotResults.py

Producing nice plots from the simulations results.

```
1  from numpy import NaN, isnan, array, concatenate
2  import matplotlib.pyplot as plt
3
4  class PlotResults:
5
6      def __init__(self, res):
7          self.res = res
8          self.numberOfResults = len(self.res)
9
10     def plotQueueLengthVsTime(self, figname = None):
11         '''
12         plot steps of queue length versus time
13         '''
14         fig = plt.figure(figsize=(10,8))
15         plt.rcParams.update({
16             'font.size'   : 18,
17             # 'text.usetex': 1,
18             # 'font.family':'serif',
19             # 'font.serif' :'Computer Modern Typewriter'
20             })
```

```
21
22          i = 0
23          for result_id,result in self.res.items():
24
25              lists_sgl = sorted(result.numberSimBicyclesPlot.
                    items()) # sorted by key, return a list of
                    tuples
26              t_sgl, sgls = zip(*lists_sgl) # unpack a list of
                    pairs into two tuples
27
28
29              lists_ql = sorted(result.queueLengthPlot.items()) #
                    sorted by key, return a list of tuples
30              t_ql, qls = zip(*lists_ql) # unpack a list of pairs
                    into two tuples
31
32              lists_color = sorted(result.trafficlightPlot.items()
                    ) # sorted by key, return a list of tuples
33              t_color, colors = zip(*lists_color) # unpack a list
                    of pairs into two tuples
34
35              # assert set(t_color) <= set(t_ql), 't_color is not
                    part of t_ql'
36              assert t_color == t_ql, 't_color is not the same as
                    t_ql'
37              t = array(t_ql)
38
39              red   = array([1 if c == 0 else NaN for c in colors
                    ])
40              amber = array([1 if c == 1 else NaN for c in colors
                    ])
41              green = array([1 if c == 2 else NaN for c in colors
                    ])
42              red_indx   = ~isnan(red)
43              amber_indx = ~isnan(amber)
44              green_indx = ~isnan(green)
45
46              qls = array(qls)
47              sgls = array(sgls)
48              colors = array(colors)
49
50              ymax = max(sgls)
51              # assert ymax > 0, 'ymax is zero'
52
53              # ax = plt.subplot(self.numberOfResults*100 + 10 + i
                    +1)
54              ax = plt.subplot(self.numberOfResults*100 + 10 + (
                    self.numberOfResults - i))
55              # ax.set_title(result_id)
56              ax.plot(t,qls, drawstyle='steps-post',color='k',
                    linewidth = 2)
57
58              # avoid to show flushing to calculate totalDelay
59              sgls = concatenate((sgls[:-1],[sgls[-2]]))
```

```
60              ax.fill_between(t,[0]*len(sgls),sgls,color='
                    lightgray')
61
62              ax.plot(t[red_indx]   ,qls[red_indx]   ,'o', color='
                    crimson'     , markersize=10)
63              ax.plot(t[amber_indx],qls[amber_indx],'o', color='
                    darkorange' , markersize=10)
64              ax.plot(t[green_indx],qls[green_indx],'o', color='
                    darkgreen'   , markersize=10)
65
66              ax.set_ylim([0,1.05*ymax])
67              # ax.xticks(t)
68              # ax.ylabel(u"queue length \U0001F6B2")
69
70              # if i == self.numberOfResults - 1:
71              if i == 0:
72                  ax.set_xlabel('time [s]')
73              ax.set_ylabel("queue length [bikes]")
74              i+=1
75
76          if figname == None:
77              plt.tight_layout()
78              plt.show()
79          else:
80              # fig.savefig('output/senario_{}.pdf'.format(figname
                    ),format='pdf',bbox_inches='tight')
81              fig.savefig(figname,format='pdf',bbox_inches='tight'
                    )
```

4.A.2 Data structures

SWI_Traffic_Objects.py

Contains all junction related data structures.

```
1  from collections import deque
2  from ggcQL.Event import Event
3
4  class Bicycle:
5      """
6      class for bicycle
7      or C02 neutral car if you want
8      """
9
10     reactionTime = 3  # time the bicycle needs to start driving
           after seeing the light change to green
11
12     def __init__(self,identifier,arrivalTimeGiven):
13         assert type(identifier)==str
14
15         self.identifier = identifier          # name of bicycle
```

```
16          self.arrivalTime = arrivalTimeGiven  # arrival time of
                the bicycle
17          self.delay = 0                       # delay of this
                bicycle
18
19      def __str__(self):
20          return self.identifier
21
22  class SignalGroup:
23
24      RED   = 0
25      AMBER = 1
26      GREEN = 2
27
28      def __init__(self,identifier,travelTimeGiven,arrivalTimes):
29          assert type(identifier)==str
30
31          # public
32          self.identifier = identifier          # string
33          self.travelTime = travelTimeGiven     # time needed to
                drive from first loop to exit
34          self.realBicycles = deque()           # list of Bicycles
                 that are physcially in the signal group based on
                sensor data
35          self.finished = False                 # flag to check if
                 sg simulation is finished
36
37          #private
38          self._simulatedBicycles = deque()      # (empty) list of
                 Bicycles that are simulated
39          self._trafficLight = self.RED          # red traffic
                light
40          self._queue = deque()                  # list of
                Bicycles in a queue
41
42          self.importRealBicycles(arrivalTimes)  # convert arrival
                 times to real bikes
43
44
45      def __str__(self):
46          return self.identifier
47
48      def importRealBicycles(self, arrivalTimes):
49          '''
50          arrivalTimes are arrival times of bikes in past
51
52          fills the list of realBicycles
53          '''
54
55          numArrivals = len(arrivalTimes)   # number of bicycles
                arrive the signal group
56          # initialize arrivals
57          for i in range(numArrivals):
58              self.realBicycles.append(Bicycle('bike{}'.format(i
                    +1),arrivalTimes[i]))
```

```
59
60
61      def initializeSimulatedBikes(self):
62          '''
63          copy real bikes to sim bikes
64          '''
65          # copy realBicycles into simulatedCycles
66          for bicycle in self.realBicycles:
67              newSimBicycle = Bicycle('{}sim{}'.format(self.
                    identifier,bicycle.identifier),bicycle.
                    arrivalTime)
68              self._simulatedBicycles.append(newSimBicycle)
69          # for bike in self._simulatedBicycles:
70          #     print(bike)
71
72      def initialEvent(self,currentTime):
73          """
74          create the initial event
75          """
76
77          firstSimBicycle = self._simulatedBicycles[0]
78          if self._trafficLight in [self.GREEN]:
79              # To do: make sure it does not pass red (and add
                    self.AMBER)
80
81              # schedule departure event of first Bicycle
82              timeToNewEvent = max(Bicycle.reactionTime,self.
                    travelTime - (currentTime-firstSimBicycle.
                    arrivalTime))
83              newEvent = Event(Event.DEPARTURE_FROM_SG,currentTime
                    +timeToNewEvent,self,firstSimBicycle)
84          else:
85              # schedule arrivale at queue list for first Bicycle
86              timeToNewEvent = max(0,self.travelTime - (
                    currentTime-firstSimBicycle.arrivalTime))
87              newEvent = Event(Event.ARRIVAL_AT_QUEUE,currentTime+
                    timeToNewEvent,self,firstSimBicycle)
88
89          return newEvent
90
91      def addBicycle(self,bicycle):
92          """
93          add bicycle to signal group
94          """
95          if len(self._simulatedBicycles)>0:
96              assert bicycle.arrivalTime > self._simulatedBicycles
                    [-1].arrivalTime, 'Trying␣to␣add␣bicycle␣to␣
                    signal␣group␣which␣arrived␣before␣the␣last␣
                    Bicycle␣already␣in␣the␣signal␣group'
97          self._simulatedBicycles.append(bicycle)
98
99      def addBicycleToQueue(self,bicycle):
100         """
101         add bicycle to queue
102         """
```

```
103             assert bicycle in self._simulatedBicycles, 'Trying to
                    add bicycle which is not in signal group to queue'
104             self._queue.append(bicycle)
105
106         def queueLength(self):
107             """
108             return the queue length
109             """
110             return len(self._queue)
111
112         def numberSimBicycles(self):
113             """
114             return number of bikes in the signal group
115             """
116             return len(self._simulatedBicycles)
117
118         def nextBicycle(self,bike):
119             '''
120             return the bike after the given bike
121             '''
122             assert bike in self._simulatedBicycles, 'bicycle is not
                    in signal group'
123             if bike == self._simulatedBicycles[-1]:
124                 # if the bike was the last bike there is no bike
                        after this one
125                 return None
126             else:
127                 # get index of bike and return the next one.
128                 index_bike = self._simulatedBicycles.index(bike)
129                 return self._simulatedBicycles[index_bike+1]
130
131         def removeFirstBicycle(self,time):
132             """
133             remove and return the first bicycle from the signal
                    group
134             """
135             firstBicycle = self._simulatedBicycles.popleft()
136             firstBicycle.delay = (time - firstBicycle.arrivalTime) -
                     self.travelTime  # time needed - ideal time
137             return firstBicycle
138
139         def removeFirstBicycleInQueue(self,t):
140             """
141             remove the first bicycle from the queue
142             """
143             return self._queue.popleft()
144
145         def getFirstBicycleInQueue(self):
146             """
147             return the first bike in the queue
148             """
149             if len(self._queue) == 0:
150                 return None
151             else:
152                 return self._queue[0]
```

```
153
154     def changeTrafficLight(self,newTrafficLight,time):
155         """
156         change the traffic light color at time time
157         """
158         self._trafficLight = newTrafficLight
159
160         # later on we may do more stuff depending on the new
                value
161
162     def currentTrafficLight(self):
163         """
164         return current traffic light color
165         """
166         return self._trafficLight
167
168 # to add later
169 # class Junction:
170 #     def __init__(self,identifier):
171 #         assert type(identifier)==str
172 #         self.identifier = identifier
173 #         self.signalGroups = []
174
175 #     def addSignalGroup(self,signalGroup):
176 #         self.signalGroups.append(signalGroup)
```

Event.py in folder ggcQL

The event class.

```
1  # Based on code of lecture notes: Stochastic Simulation using
       Python by Mark Boon a.o.
2
3  class Event:
4
5      TORED    = 0    # constant for change of traffic-light to RED
6      TOAMBER = 1    # constant for change of traffic-light to
           AMBER
7      TOGREEN = 2    # constant for change of traffic-light to
           GREEN
8
9      ARRIVAL_AT_SG          = 3      # constant for arrival type
10     ARRIVAL_AT_QUEUE       = 4      # constant for arrival at queue
           type
11     DEPARTURE_FROM_SG      = 5      # constant for departure type
12     DEPARTURE_FROM_QUEUE = 6      # constant for departure from
           queue type
13
14     STOP_SIMULATION = 7            # stop simulation
15
16     def __init__(self, typ, time, sg, bicycle = None):  # type
           is a reserved word
17         self.type = typ                  # type of event
```

```
18          self.time = time                    # time of event
19          self.sg   = sg                      # signal group where
                this events takes place
20          self.bicycle  = bicycle                     # bicycle of
                event (optional)
21
22      def __lt__(self, other):                # compare to other
            events
23          return self.time < other.time
24
25      def __str__(self):
26          s = (
27              'ToRed',
28              'ToAmber',
29              'ToGreen',
30              'Arrival for sg',
31              'Arrival at queue',
32              'Departure from sg',
33              'Departure from queue',
34              'Derminate simulation'
35          )
36          string = '{:20s} at t = {:3.2f} of {} '.format(s[self.
                type],self.time, self.sg)
37          if self.bicycle:
38              string += 'of {}'.format(self.bicycle)
39          return string
```

FES.py in folder ggcQL

The event list.

```
1   # Based on code of lecture notes: Stochastic Simulation using
        Python by Mark Boon a.o.
2
3   import heapq
4
5   class FES :
6
7       def __init__(self):
8           self.events = []
9
10      def add(self, event):
11          heapq.heappush(self.events, event)
12
13      def next(self):
14          return heapq.heappop(self.events)
15
16      def isEmpty(self):
17          return len(self.events) == 0
18
19      def __str__(self):
20          # Note that if you print self.events, it would not
                appear to be sorted
```

```
21          # (although they are sorted internally).
22          # For this reason we use the function 'sorted'
23          s = ''
24          sortedEvents = sorted(self.events)
25          for e in sortedEvents :
26              s += str(e) + '\n'
27          return s
```

SimResults.py in folder ggcQL

Class for storing the simulation results.

```
1   # Based on code of lecture notes: Stochastic Simulation using
        Python by Mark Boon a.o.
2
3   from numpy.ma.core import zeros
4
5   class SimResults:
6
7       MAX_QL = 1000
8
9       def __init__(self):
10
11          self.trafficlightPlot = {}              # traffic light
                color vs time
12          self.queueLengthPlot  = {}              # queue lengths vs
                event time
13          self.numberSimBicyclesPlot  = {}        # number of bikes
                in the sg vs event time
14
15          self.queueLengthHistogram = zeros(self.MAX_QL + 1)  #
                histogram output
16          self.oldTime = 0        # save time of previous event for
                 histogram.
17
18          self.totalDelay = 0     # total delay
19          self.quadraticDelay = 0   # total quadratic delay cost
20          self.sumQL = 0
21
22      def registerQueueLength(self, t, sg):
23          '''
24          t is time
25          sg is signal group
26
27          saves the queuelength of the signal group at time t
28          saves histogram that shows distribution of queuelength
                vs time
29          '''
30          ql = sg.queueLength()                   # queue length of
                signal group
31
32          self.queueLengthPlot[t] = ql            # save queue length
                at time t
```

```
33
34          self.queueLengthHistogram[min(ql, self.MAX_QL)] += (t -
                self.oldTime)
35          self.sumQL += (t - self.oldTime) * ql
36          self.oldTime = t          # save previous time
37
38      def registerSignalGroupLength(self, t, sg):
39          '''
40          t is time
41          sg is signal group
42
43          saves the number of bicycles in the signal group at time
                t
44          '''
45          ql = sg.numberSimBicycles()                 # queue length
                of signal group
46
47          self.numberSimBicyclesPlot[t] = ql          # save queue
                length at time t
48          # self.queueLengthHistogram[min(ql, self.MAX_QL)] += (t
                - self.oldTime)
49          # self.oldTime = t          # save previous time
50
51      def registerTrafficLightColor(self, t, sg):
52          '''
53          t is time
54          sg is signal group
55
56          saves the current traffic light color of the signal
                group at time t
57          '''
58          color = sg.currentTrafficLight()          # queue length
                of signal group
59          self.trafficlightPlot[t] = color          # save queue
                length at time t
60
61      def registerTotalDelay(self, bicycle):
62          '''
63          bicycle is a bicycle
64
65          addes delay of individual bicycle to total delay (and to
                quadratic cost as well)
66          '''
67          self.totalDelay += bicycle.delay
68          self.quadraticDelay += bicycle.delay * bicycle.delay  #
                quadratic delay
69
70
71      def getMeanQueueLength(self):
72          return self.sumQL / self.oldTime
73
74      def getQueueLengthProbabilities(self) :
75          return [x/self.oldTime for x in self.
                queueLengthHistogram]
76
```

```
77      def plotQueueLengthHistogram(self, maxq=25):
78          ql = self.getQueueLengthProbabilities()
79          maxx = maxq + 1
80          plt.figure()
81          plt.bar(range(0, maxx), ql[0:maxx])
82          plt.ylabel('P(Q = k)')
83          plt.xlabel('k')
84          plt.show()
85
86      # def plotQueueLengthVsTime(self):
87      #     '''
88      #     plot steps of queue length versus time
89      #     '''
90
91      #     lists_sgl = sorted(self.numberSimBicyclesPlot.items())
            # sorted by key, return a list of tuples
92      #     t_sgl, sgls = zip(*lists_sgl) # unpack a list of pairs
            into two tuples
93
94
95      #     lists_ql = sorted(self.queueLengthPlot.items()) #
            sorted by key, return a list of tuples
96      #     t_ql, qls = zip(*lists_ql) # unpack a list of pairs
            into two tuples
97
98      #     lists_color = sorted(self.trafficlightPlot.items()) #
            sorted by key, return a list of tuples
99      #     t_color, colors = zip(*lists_color) # unpack a list of
            pairs into two tuples
100
101     #     # assert set(t_color) <= set(t_ql), 't_color is not
            part of t_ql'
102     #     assert t_color == t_ql, 't_color is not the same as
            t_ql'
103     #     t = array(t_ql)
104
105     #     red   = array([1 if c == 0 else NaN for c in colors])
106     #     amber = array([1 if c == 1 else NaN for c in colors])
107     #     green = array([1 if c == 2 else NaN for c in colors])
108     #     red_indx   = ~isnan(red)
109     #     amber_indx = ~isnan(amber)
110     #     green_indx = ~isnan(green)
111
112     #     qls = array(qls)
113     #     sgls = array(sgls)
114     #     colors = array(colors)
115
116     #     ymax = max(sgls)
117     #     # assert ymax > 0, 'ymax is zero'
118     #     plt.figure()
119     #     plt.plot(t,qls, drawstyle='steps-post',color='k',
            linewidth = 2)
120     #     # plt.fill_between(t,[0]*len(sgls),sgls,color='#373F51
            ')
121
```

```
122         #       # avoid to show flushing to calculate totalDelay
123         #       sgls = concatenate((sgls[:-1],[sgls[-2]]))
124         #       plt.fill_between(t,[0]*len(sgls),sgls,color='lightgray
                ')
125
126         #       plt.plot(t[red_indx]   ,qls[red_indx]   ,'o', color='
                crimson'     , markersize=10)
127         #       plt.plot(t[amber_indx],qls[amber_indx],'o', color='
                darkorange' , markersize=10)
128         #       plt.plot(t[green_indx],qls[green_indx],'o', color='
                darkgreen'   , markersize=10)
129
130         #       plt.ylim([0,1.05*ymax])
131         #       # plt.xticks(t)
132         #       plt.xlabel('time [s]')
133         #       # plt.ylabel(u"queue length \U0001F6B2")
134         #       plt.ylabel("queue length [bikes]")
135         #       plt.show()
136
137         # def __str__(self):
138         #       ql = self.getQueueLengthProbabilities()
139         #       s = 'Mean queue length: ' + str(self.
                getMeanQueueLength()) + '\n'
140         #       s += 'Queue-length probabilities: \n'
141         #       for i in range(21):
142         #           s += 'P(Q = ' + str(i) + ') = ' + str(ql[i]) + '\n
                '
143         #       return s
144
145 # t = [1, 2, 3.5, 4, 4.5, 5]
146 # q = [4, 5, 5  , 4, 4  , 4]
147 # c = [0, 0, 2  , 2, 1  , 0]
148
149 # pif = SimResults()
150 # for i in range(len(t)):
151 #       pif.queueLengthPlot[t[i]] = q[i]
152 #       pif.trafficlightPlot[t[i]] = c[i]
153
154 # pif.plotQueueLengthVsTime()
```

4.A.3 Input files

arrivalTimes.json

```
1  {
2      "sg1" : {
3          "travelTime" : 5,
4          "arrivalTimes" : [-70, -69, -68, -29, -7, -5, -3]
5      },
6      "sg2" : {
7          "travelTime" : 5,
```

```
8           "arrivalTimes" : [-40, -35, -25, -15]
9       }
10  }
```

junction.json

```
1   {
2       "description" : "Junction with two simple signal groups",
3       "sgIDs" : ["sg1","sg2"]
4   }
```

scenario3.json

This is the first scenario as considered in the example of Section 4.4.

```
1   {
2       "description" : "sg1 green until sg1 is empty",
3
4       "sg1" :{
5           "times_turn_green" : [0.1],
6           "times_turn_amber" : [21.6],
7           "times_turn_red"   : [23.6]
8       },
9
10      "sg2" :{
11          "times_turn_green" : [23.6],
12          "times_turn_amber" : [38.4],
13          "times_turn_red"   : [40.4]
14      }
15
16  }
```

scenario4.json

This is the second scenario as considered in the example of Section 4.4.

```
1   {
2       "description" : "sg2 green until sg2 is empty",
3
4       "sg1" :{
5           "times_turn_green" : [14.6],
6           "times_turn_amber" : [60],
7           "times_turn_red"   : [62]
8       },
9
10      "sg2" :{
11          "times_turn_green" : [0.1],
12          "times_turn_amber" : [12.5],
13          "times_turn_red"   : [14.5]
14      }
15
16  }
```

scenario5.json

This is the third scenario as considered in the example of Section 4.4.

```
1  {
2      "description" : "first␣let␣sg1␣go␣for␣a␣bit,␣then␣sg2,␣then␣
           sg1␣again",
3
4      "sg1" :{
5          "times_turn_green" : [0.1, 28.6],
6          "times_turn_amber" : [12.6, 41.1],
7          "times_turn_red"   : [14.6, 43.1]
8      },
9
10     "sg2" :{
11         "times_turn_green" : [14.7],
12         "times_turn_amber" : [26.8],
13         "times_turn_red"   : [28.8]
14     }
15
16 }
```