**Vrije Universiteit Brussel**
**Faculteit Wetenschappen**
**Departement Informatica en Toegepaste Informatica**


# A realistic simulation for self-organizing traffic lights.

Proefschrift ingediend met het oog op het behalen van de graad van Licentiaat in de Informatica

Door: Seung Bae Cools
Promotor: Prof. Bart D'Hooghe
September 2006

© Vrije Universiteit Brussel, all rights reserved.

With thanks to Carlos Gershenson and Bart D’Hooghe for everything they did.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Cars have become a very popular form of transportation. Owners of cars are not only using their vehicle for long trips, but also for short trips like visiting family and friends or shopping. Kölbl and Helbing say that the total average traveling time is inversely proportional to the energy it takes for the traveler himself [KH03]. The traffic density has been growing during the last decades. At some places this becomes a problem. The traffic flow is slowing down or even blocked. This causes not only a loss of time, but it is frustrating, harmful for the environment and economics. All improvements to the traffic flow can help and are needed to keep up with the growing traffic density. A lot of research is done on optimization or adaptation of traffic light controllers. Testing the efficiency or optimization of decision variables can be done with simulations.

Carlos Gershenson proposes a solution "in which traffic lights self-organize to improve traffic flow" [Ger05].

## 1.1 Simulations

Improvements to the road infrastructure are expensive and new projects changing the infrastructure need to be evaluated on their effects on traffic. Simulations are very useful for such evaluations and have been used by many people [DS04, Her04, WVVK04, Ohi97, Nag04]. It is not only possible to evaluate the modifications under normal conditions, but specific scenarios can also easily be

tested.

The traffic light controllers of Carlos Gershenson have decision variables which influence the performance of these controllers. Simulations are very useful for optimizing decision variables of a system, when it is evaluated on its performance [OK02].

Simulations are done for many purposes and a common approach consists of formulating the problem, analyzing, modeling and experimenting. In this way simulation results can be obtained successfully [Bal98].

New traffic light controllers can be checked in different scenarios on a realistic traffic simulator. The simulation model has to be as realistic as possible to obtain convincing results. Testing traffic light controllers in the real world would not only be expensive in time and money, but it could also disturb the traffic flow or even cause physical damage to vehicles and drivers.

Some have used probabilistic cellular automata for simulating traffic [FSS04b, FSS04a].

Drivers, traffic light controllers and others make their own decisions which results in an emergent behaviour. Agent technology can be used for constructing detailed behaviours for individual entities. It is possible to run simulations on multiple computers for a given road map. When the map is divided into smaller pieces, one for each computer, the computational performance will increase [KEW]. Also others have used agent technology for simulating traffic [Sot, DS04, Nag04, WS02]

A description of an agent-based traffic simulator by Levi and WenthWorth [KEW]:

> The main entities in the traffic network are road segments, intersections, vehicles, traffic lights, signs and sensors, which are modeled as agents. The traffic network can be partitioned into regions, along geographical boundaries. Agents corresponding to entities in the same region are hosted together in the same agent community, so that they have a high-bandwidth, low latency communication among them. Vehicle agents are mobile: as they enter a new region, they will physically migrate to the agent community hosting that region.

Regions are highly autonomous. When simulation of a new region is deployed, it will automatically connect and synchronize with the adjoining regions already running, accepting incoming vehicles, and sending outgoing vehicles. . . . This approach allows flexible simulation of large traffic networks. Regions can be distributed to the computers on the network, and depending on the level of interest, each region can be simulated in detail in micro level, abstractly in macro level, or it may not even be simulated, by relying on probability distributions to model the traffic coming out of that region. Thus this approach helps integrate partitions of large traffic network simulations that use very different simulation techniques.

Intersection agents are responsible for controlling the flow of vehicle agents from one segment to the next. They operate the traffic lights. Intersection agents at the boundary of a region can act as sources or sinks when the adjoining region is not being simulated. In such a situation, the intersection will generate the incoming traffic using a probability distribution.

A vehicle agent contains the physical attributes of the vehicle such as length, acceleration, type, as well as the driver's characteristics, such as aggressiveness and route. It also includes the car-following and lane-changing behavior. In implementing those behaviors, the vehicle agent needs to continuously interact with the vehicles around it. In order to provide fast communication among interacting vehicle agents, vehicle agents are implemented as aglets, and hosted on the road segment agent that they travel on. An aglet has the same conceptual attributes and the behavior as an agent, except that fellow aglets hosted on the same agent can have very fast, shared-memory communication. The road segment agent monitors the position and the lane of the vehicles on it, and interfaces with the two intersection agents at either end. It also hosts the sensor and advisory-sign aglets.

## 1.2 Goals

For simulating the self-organizing traffic light controllers, a realistic traffic simulator is needed in which those traffic light controllers can be tested. My first goal is to get such a traffic simulator. Other programmers have already worked on such projects and the Green Light District / iAtracos project is a good starting point for implementing the simulator. Those projects already have a lot of features implemented and I added extra features to get a more realistic traffic simulator. Cars can accelerate and decelerate and they keep a stop distance to the previous one. They respect the maximum speed of the road they are moving on, etc.

My second goal is to demonstrate that the sotl-platoon controller of Carlos Gershenson performs much better than a green wave controller. This is shown by simulating those controllers in different scenarios, and comparing the average total waiting time of the road users for different controllers.

The third goal is to show the *Ministry of the Brussels-Capital region* that the sotl-platoon traffic light controller is better than the current controller in the Wetstraat of Brussels. The Wetstraat is currently using a green wave algorithm towards the centre of Brussels. The Wetstraat has to be studied first concerning the design of the road, traffic density, origin and destination of cars that travel on the Wetstraat, the green times and periods of the current traffic light controller, etc. These data are then translated into a map which will be used by the simulator. Finally the simulations of the new traffic light controllers can be tested on the map and the simulation data will be compared for different traffic light controllers.

## 1.3 Outline

In chapter two the traffic light controllers of Carlos Gershenson are introduced and some other research on optimization and adaptation of traffic lights. For the simulation I made improvements to an existing traffic simulator project, which is discussed in chapter three, and in chapter four a full report is given on the improvements on this simulator. Chapter five presents and discusses the simulation results for three scenarios with five traffic light controllers. Chapter six contains the final conclusions of the improvements to the simulator and the simulation results.

# Chapter 2

# Optimization and adaptation of traffic lights

Contributions to traffic management such as synchronizing traffic lights on a primary road have been an improvement. Those synchronizations are called green-waves: a car that gets green at the first traffic light will get green at the next traffic light when the car is approaching it (when driving at given speed). This green-wave is pre-calculated for a given travel speed. When the traffic density is high, the given speed cannot be reached and the aim of the green-wave cannot be fulfilled.

Some research is done on optimizing traffic by letting the traffic light controller learn the best actions. Wiering et al proposed to use reinforcement learning algorithms. Those algorithms learn the expected waiting times of cars for red and green lights at each intersection, and sets the traffic lights to green for maximal gain configuration. [WVVK04]

Other researchers are focussing on adaptation of traffic light controllers. Traffic demands varies during the day and the traffic lights should sense the changes in traffic flow and adapt to it. [Ger05, DH05]

# Self organizing traffic lights by Carlos Gershenson

Carlos Gershenson presents three self-organizing control methods: sotl-request, sotl-phase and sotl-platoon. Marching control and Optim control are non adaptive methods and are used to compare those self-organizing control methods. The definitions of the controllers are given below, which are quoted from the paper of Carlos Gershenson [Ger05]. Those definitions are for one direction lanes. The realistic simulation has two-direction roads for scenario 1 and 2.

## Marching Control

> All traffic lights "march in step": all green lights are either southbound or eastbound, synchronized in time. Intersections have a phase $\varphi_i$, which counts time steps. $\varphi_i$ is reset to zero when the phase reaches a period value p. When $\varphi_i = 0$, red lights turn green, and yellow lights turn red. Green lights turn yellow one time step earlier, that is, when $\varphi = p - 1$. A full cycle of an intersection consists of 2p time steps. "Marching"intersections are such that $\varphi_i = \varphi_j, \forall i, j$.

## Optim control

> This method is implemented trying to set phases $\varphi_i$ of traffic lights in such a way that, as soon as a red light turns green, a car stopped by this would find the following traffic lights green. In other words, we obtain a fixed solution so that green waves flow to the southeast. This green wave method is very popular for avenues that have preference over crossing streets.

## Sotl-request control

> All three self-organizing control methods use a similar principle: traffic lights keep a counter $\kappa_i$ which is set to zero when the light turns red and then incremented at each time step by the number of cars approaching only the red light independently of the status or speed of

the cars. When $\kappa_i$ reaches a threshold $\theta$, the green light at the same intersection turns yellow, and the following time step it turns red with $\kappa_i = 0$, while the red light which counted turns green.

The sotl-request method has no phase or internal clock. Traffic lights change only when the given conditions are met. If there are no cars approaching a red light, the complementary one can stay green. However, depending on the value of theta, high traffic densities can trigger the lights to switch too fast, obstructing traffic flow.

## Sotl-phase control

The sotl-phase method differs from sotl-request by adding the following constraint: A traffic light will not be changed if the number of time steps is less than a minimum phase, that is, $\varphi_i < \varphi_{min}$. Once $\varphi_i \geq \varphi_{min}$, the lights will change when $\kappa_i \geq \theta$. This prevents the fast switching of lights.

## Sotl-platoon control

The sotl-platoon method adds two further restrictions to sotl-phase to regulate the size of platoons. Before changing a red light to green, it checks if a platoon is not crossing through, in order not to break it. More precisely, a red light is not changed to green if on the crossing street there is at least one car approaching within $\omega$ patches from the intersection. This keeps crossing platoons together.

The second restriction is: Restriction one is not taken into account if there are more than $\mu$ cars approaching the intersection.

# Chapter 3

# GLD and iAtracos

For the simulations of the new traffic light controllers a realistic traffic simulator is needed. Programmers of the University of Utrecht have created a good traffic simulator with a lot of usable features. This project has been extended by a group of programmers from Argentina. My project started from the iAtracos project [ea] which is extended with extra physics, driving behaviour etc. In this chapter a short overview is given of the delivered features of the GLD and iAtracos projects.

## 3.1 GLD

Green Light District (GLD) was developed by Intelligent Systems Group at the University of Utrecht [gld, WVVK04]. GLD is an open source program which supports editing complex city maps, traffic simulation on these maps and changing parameters for the simulation. Evaluation tools support displaying analysed data of the simulation in a various number of ways:

- In-view statistics: this will colour junctions to indicate the average waiting time.
- Statistics window: contains precise data on the current simulation map.
- Tracking window to track total waiting queue length, average trip waiting time or average junction waiting time.

> New advances in physical detection of passing road users and improved traffic light control algorithms are to be combined to optimize traffic flow in the future. Deciding the best use of these new technologies is best done by detailed simulation, to find out whether a costly new system would be profitable when applied to a certain infrastructure. This is the purpose of the Green Light District [gld].

More information can be found at the GLD website. [gld]

## 3.2 iAtracos

i-Atracos stands for “Intelligent Argentinean TRAffic COntrol System”, which is an expanded and improved version of GLD, implemented by Gaston Escobar et al.

- Dynamic incident generation: Users can declare roads unusable with a mouse click
- Flow traffic schedule: it is possible to run a simulation with changing spawn frequencies by specifying intervals in an XML file.
- Simulation data sender interface: simulation data can be exported in XML format
- Traffic interval sender interface: interface to send the SignConfig intervals.

More information can be found at the sourceforge website of this project [ea].

# Chapter 4

# Coding process

It is not always easy to build further on an existing project. First the design and code of the project needs to be studied. After understanding the code, improvements or extra features can be added.

When a feature has been added, thorough testing is needed.

The most relevant pieces of the code are discussed in section *Reverse engineering*. Then all modifications are given in section *List of modifications*.

## 4.1 Open source

The Green Light District project has started as an open source project, which was extended by the iAtracos project. Open source projects are very useful because existing projects can be used and changed to your own interest. My interest in this project was to add real world properties like physics etc. to the project like acceleration, deceleration, maximum allowed speed on a road, keeping stopdistance, difference between primary and secondary roads, etc.

The source code of the extended program is available at sourceforge.
`http://morevts.cvs.sourceforge.net/morevts/`.

## 4.2 Reverse engineering

When new features have to be implemented into an existing project, programmers need to understand the design of the project and how the relevant pieces of code work. This can be a very intensive work and tools which search for declarations, type hierarchies and call hierarchies are very useful in this process. Eclipse supports these features and those were very useful to learn the project's implementation.

Also the communication with Gaston Escobar has helped in understanding the code. I have mailed and chatted with him, asking questions about the code. The Green Light District project was provided with some UML diagrams which gave better insight in the global design of the project.

The most relevant parts of the code which were needed for implementing the extra features are discussed below.

### 4.2.1 Speed of roadusers and its representation on a drivelane

The speed is represented as a number of blocks moved per *cycle*. In every cycle all roadusers move a number of blocks on the *drivelane* as specified in the method *speed*.

The *roadusers* on a specific drivelane are stored in a linked list. When a *roaduser* moves on the drivelane, the position represented in the specific *roaduser*-class is changed. The length of a drivelane is stored in the class *Road*, which contains a number of drivelanes. When a roaduser enters a drivelane, its position is the length of the road, and when it moves forward, the position of the roaduser decreases. The end of the road is represented by $position = 0$.

### 4.2.2 Nodes and subclasses

The class *Node* and all its subclasses are located in *gld.infra* Node is the abstract class of all nodes. *Junction* and *SpecialNode* are subclasses of *Node*. *Junction*

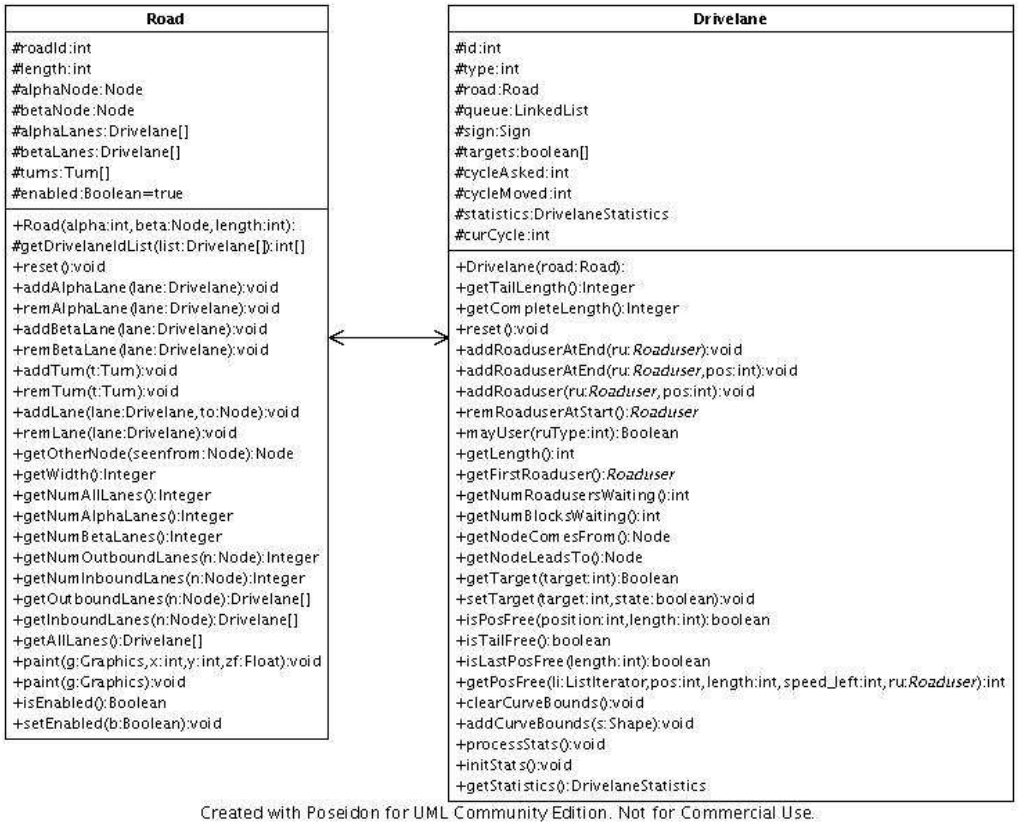


Figure 4.1: UML diagram: Road and Drivelane

represents all intersections, those with traffic lights and those without traffic lights (*NonTLJunction* which is a subclass of *Junction*). The subclasses of *SpecialNode* are *EdgeNode* and *NetTunnel*. At an *EdgeNode* road users will start and end their trip.

### 4.2.3 Roadusers and how they move

*Roaduser* is the most super class representing all road users. It has 2 subclasses: *Automobile* and *Bicycle*. *Roaduser* and *Automobile* are abstract classes. *Automobile* has 2 subclasses: *Car* and *Bus*. The *roadusers* have an attribute *pos*, which represents their current position on the lane. This position is changed by its accessors at other places, like in *moveLane*.

**Moving the roadusers** is done in *(gld.sim) SimModel* in the methods *moveAllRoadusers*, *moveLane* and *moveRoaduserOnLane*. *moveAllRoadusers* calls the method *moveLane* for all inbound lanes when the lane has not moved yet at the current cycle.
*moveLane* is moving the roadusers on a given lane. For every road user on this lane, beginning with the first road user on the lane. A lot of different states are possible for the road user. When a road user is moved, the traffic light controller is informed. This information can be used by the algorithm for providing a table containing Q-values for each trafficlight in its 'green' setting.[1] A small flowchart (not detailed) follows:

- calculate the ranges for every drivelane that this road user could get into.
- handle road users that possibly can cross a node
  - handle road users that get to special nodes
  - handle road users that are (nearly) at a sign,
    can cross, there is place on the node.
    –> move road user from present lane to destination lane
  - handle road users that are nearly at a sign
    can cross, there is no place on the node
    –>check if next lane should move and do it
  - handle road users that are nearly at a sign
    can cross but only moved some places
- handle road users that are not near a sign

*moveRoaduserOnLane* actually moves the road user and is called by *moveLane*. The road user is the first on the road or not. If it is the first, it can move the maximum distance, if not, it checks the distance to the previous road user and moves.

[1] from info (gld.algo.tlc) TLController.java

### 4.2.4 Traffic light controllers

*Junction.signconfigs* contains all possible combinations of signs which may be turned green at the same time. It is set by *addSCData* of (*gld.edit*) *Validation*

The method *calcSC* of *(gld.algo.edit) SignConfigCalculator* calculates all possible combinations of green lights.

*SignController.switchTrafficLights* checks which *signconfig* has the highest sum of Q-values. The *signconfig* with the highest sum is chosen.

***TLController*** is the abstract class for traffic light algorithms. It is informed on every movement made by road users. In this way not every road user has to be iterated. By using this information the TLController provides a table containing Q-values for each trafficlight in its green setting.[2]
*tld* is an array in 2 dimensions. *tld[i]* represents a node i of the infrastructure. *tld[i][j]* represents a *TLDecision* (for trafficlight) j of node i.
The concrete traffic light controllers are created by *TLCFactory* in the method *getInstanceForLoad* which is called by *genTLC*. *genTLC* is called by method *setTLC* of *SimController*.

[2] info TLController.java

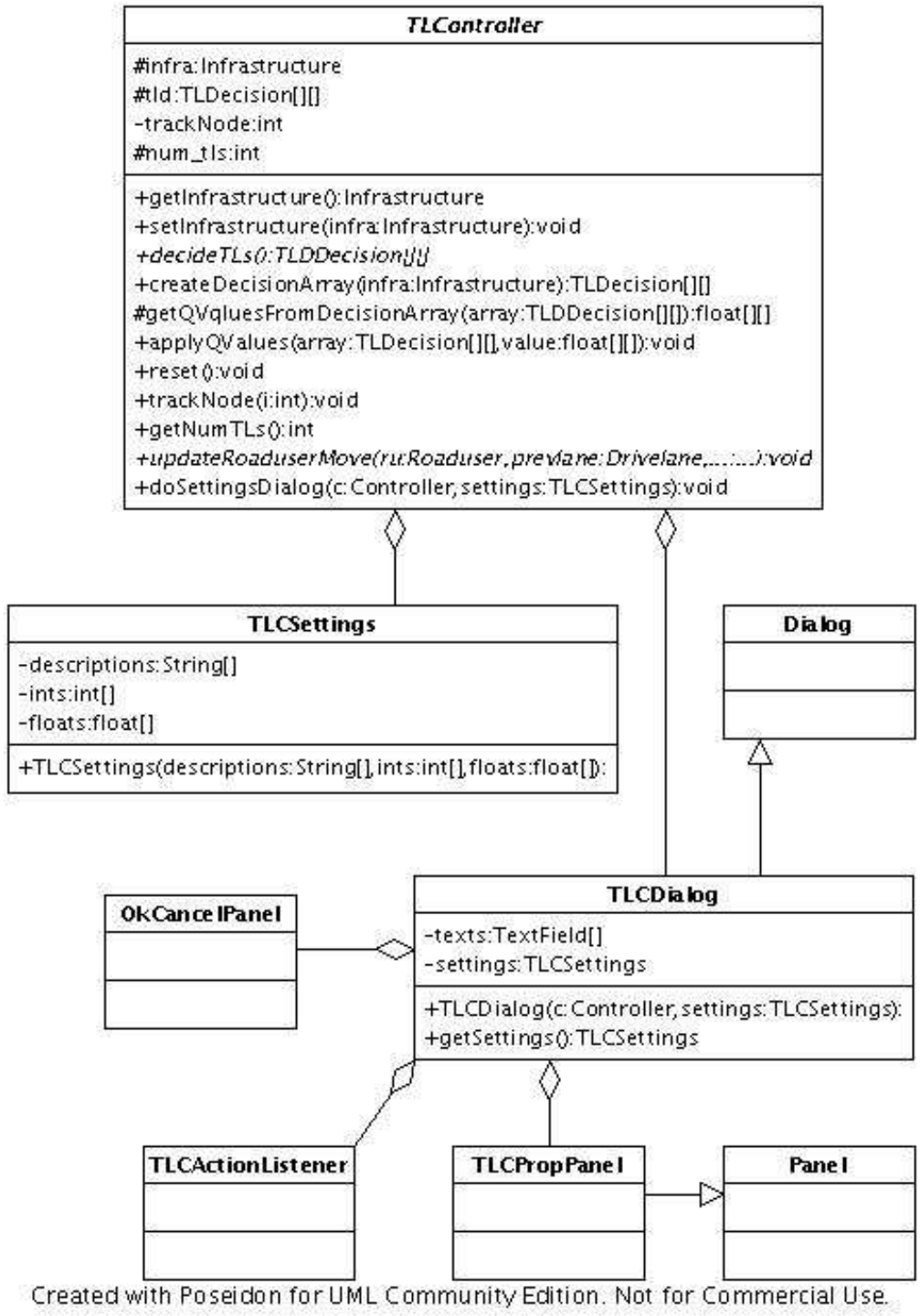


Figure 4.3: UML diagram: TLController

***TLDecision*** holds a tuple of a TrafficLight and a float value to represent the reward for the TrafficLight to be kept.[3]

**TLCFactory** is used to get instances of traffic light controllers. An instance is created in the method setTLC of SimController.

**SignController** decides how each sign should work. Only one instance is created in the SimModel.
The method *switchSigns* is called by *doStep* of *SimModel*.
switchTrafficLights

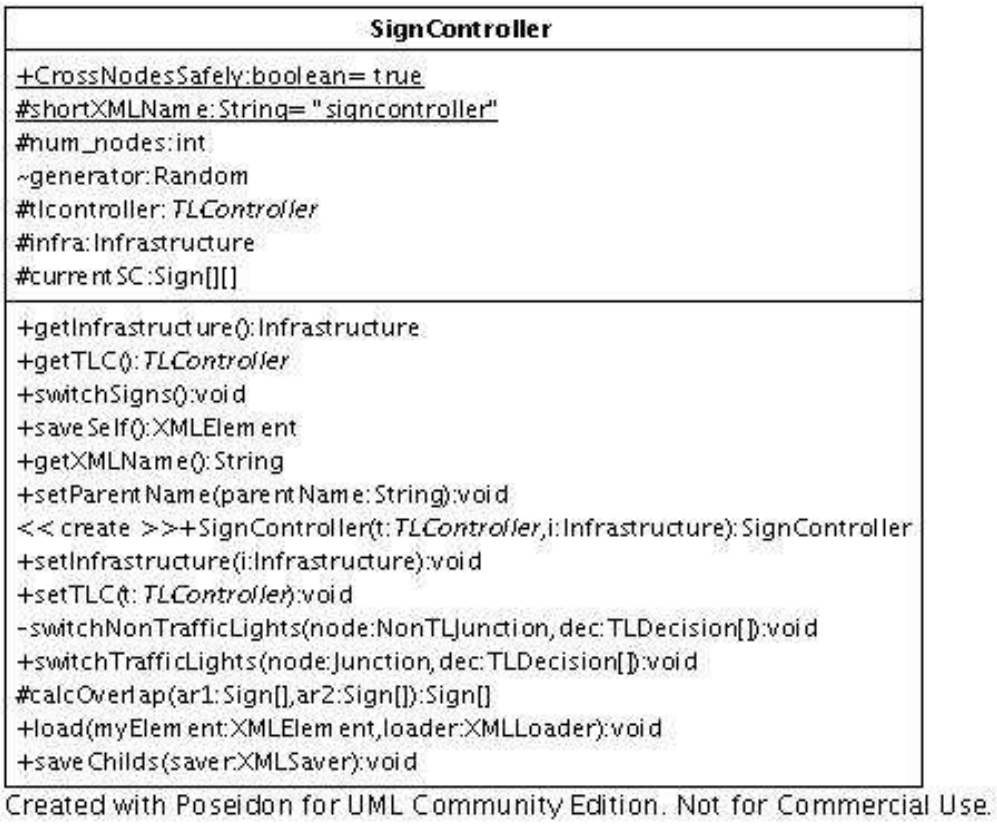


Figure 4.4: UML diagram: SignController

[3]from info TLDecision.java

### 4.2.5 Probability on enter/exit nodes

*SimModel.spawnNewRoadusers* places new road users on the roads when necessary. When roads are full, new road users are queued.

*EdgeNode* supports destination frequencies. Destination frequencies are set in *Validation* by *addFrequencies*. For every edgenode the destination frequencies to the other edgenodes are stored.

To change spawn and destination frequencies in the simulator: click on the configuration button and select a roaduser type. Now select an edgenode from which the frequencies have to be changed. When an edgenode is selected, it has a magenta square around it. To deselect, right click. To change the spawn frequency, drag the mouse from the edgenode to the left or the right. To change the destination frequency, drag the mouse from the destiniation edgenode to the left or the right. This functionality is implemented by (gld.tools)*EdgeNodeTool*

### 4.2.6 Statistics

**Total average waiting time (ATWT)** is implemented by TotalWaitTrackingView, which tracks the average trip waiting time. The function calcDelay of Node, Junction and SpecialNode was not correct anymore because of the variable speed of roadusers. I updated this function to give the correct results. calcDelay calculates the delay of a roaduser during his trip on a certain lane. $delay = (stoptime - starttime) - (lengthOfLane/maxSpeedOfLane)$.

## 4.3 List of modifications

In this section all modifications to the project are discussed.

### 4.3.1 Change size of blocks to model speed in m/s

The easiest way to represent speed is $m/s$. The length of the road has to be bigger when it is drawn in the editor. This gives a better view on the map when simula-

tions are done. The length of the $road$ is set by *CurveUtils.setupRoadSizes* which is called by *EditModel.addRoad*. To enlarge the length of the road, the length of a block needs to be smaller. This is specified by *Infrastructure.blockLength*.

The change of the block-length has an influence on how the roadusers are drawn in the simulator. The size of the roadusers are too big. Because the blocks shrinked with a factor 10, the roadusers also should be a factor 10 smaller, but because of visualization issues the width and length must be at least 1. The roadusers are made a factor 3 smaller. With this configuration it is possible that roadusers seem to get over eachother, but this is because of this problem. In the internal representation can be seen that the roadusers move correct.

The first testing gave strange results. When running the simulator it has been observed that cars are staying in the middle of the street for some time. The internal representation of the positions of the road users were correct. The bug had to be found where the road users are drawn. This is in the class $Road$. In the method *paintRoaduser* the blocklength was hard-coded, which needed to be changed in *Infrastructure.blockLength*. After testing it for a second time, everything worked correctly.

### 4.3.2 Acceleration and deceleration of road users and respecting stop-distance

Every road user has to respect the laws of physics. They accelerete and decelerate when they are driving. The roads need a maximum allowed speed. When road users enter a road with a higher maximum allowed speed than their current speed, the cars need to accelerate. When a car in front of a road user is driving slower and if the stop-distance has to be respected, the road user needs to slow down to the same speed as the car in front of him and keep some distance.

**Changing the code** Every roaduser needs a rate at wich it can accelerate and decelerate, the rate is called *acceleration*. Here the assumption is made that the acceleration and deceleration have the same rate. The support of an emergency

stop function is needed, which can be used for avoiding accidents (red light, car in front is not moving).

All road users have a maximum speed, a current speed and an acceleration. The setter of the current speed is added, also the methods for acceleration and deceleration of road users.
The roads need a maximum speed. This is implemented in class *Road*. The accessors are *getSpeedMaxAllowed* and *setSpeedMaxAllowed*.
In the editor it is needed to specify the maximum allowed speed of the roads which are drawn. To add this functionality, *(gld.config) EditRoadPanel* has to be changed. The data which is collected in the editor is saved in an infra-file, which has a XML-layout.
The *EditRoadPanel* was to small, so the panel needed to be enlarged in *ConfigDialog*. The maximum allowed speed is written in Road and has to be stored. This is done in the functions *load* and *saveSelf*. When a *edit-road-panel* is requested, the value of the maximum allowed speed needs to be initialized. This is done in the method *EditRoadPanel.setRoad*.
In the Simulator we should be able to check what the maximum speed is of the road. This is implemented in *(gld.config) SimRoadPanel*.

For testing, a map with 3 roads with different maximum allowed speeds is created and used in the simulator. All road users were changing their speed to the allowed speed on the road where they were driving on.

For letting the road users respect the stop-distance and the maximum allowed speed, the maximum allowed speed on the road, the distance to the previous road-user and his speed need to be checked. This has to be done every time a road user moves. The stop-distance needs to be calculated depending on the speed of the roaduser. $stopdistance = \frac{velocity^2}{2 \star stopforce} + reaction$. [stoa, stob] In a simulator the $reaction$ is zero.

Moving the road users is done in *(gld.sim) SimModel* in the methods *moveLane* and *moveRoaduserOnLane*.

The stop-distance is implemented in *moveRoaduserOnLane*. For all road users the method *getStopDistance* is added, which returns the stop-distance of the specific road user, calculated from the current speed and its *stopforce*. To see the current stop-distance of a specific road user in the simulation, this data is added in (gld.config) RoaduserPanel.

Adapting the speed happens also after a certain reaction-time, for example 1 second. The first road user can do what he wants and probably he will speed up to the maximum allowed speed on the road, except when he aproaches a traffic light. The following road users have to adjust their speed to the road user close in front of him.
Adjusting the speed is implemented in the method *checkRoaduserSpeed* which takes a lane as argument. This method checks for every road user on the lane if his speed has to be changed and it begins with the first road user on the lane. If the road user is the first one on the lane, it can speed up to the maximum allowed speed on the lane, except when he approaches a red traffic light, then he has to slow down in function of his stop-distance. The following roadusers need to check the speed of the previous road user. If there is enough space (2 times the stop-distance) to the previous road user, it can speed up to the maximum allowed speed on the road. If the distance to the previous is dangerous small, it has to decelerate at maximum rate. When he is at a good driving distance to the previous, it takes the speed of the previous, and follows at the same distance. This method *checkRoaduserSpeed* is called by *moveLane*. Testing this was successful.

### 4.3.3 Support for primary lanes

When a traffic control method is needed which supports the green wave, it has to be able to specify where the green wave has to occur. It has to be possible to specify in the editor wich roads are $primary$ and which are $secondary$ when adding a new road. When the roads are marked like this, it is easier to select the roads with the major traffic flow in the nodes.

The *Road* gets a new property *primary*, and the accessors *isPrimary* and *setPrimary*. The methods *load* and *saveSelf* are changed to load/save this property. To be able to give this property to a road in the editor, this was added in *EditRoadPanel*. To see this property in the simulator, this support is also added in *SimRoadPanel*.

This property of roads is tested by creating a map in the editor with primary roads and secondary roads. Those values were checked in the simulator and corresponded to the created map in the editor.

### 4.3.4 Traffic light with clearance time

Between 2 configurations of trafficlights of a certain junction, all trafficlights of that junction have to be red. During this period, all roadusers have the time to leave the junction.

This is implemented by adding *allRed* in Node. This Boolean has to be used by traffic light controllers. When a configuration of a traffic light controller has finished, it has to set this variable to $true$ and the next cycle to $false$.

In *Junction*, the method *getSignConfigs* is modified. If *allRed* is $true$ it has to return an empty sign vector, else it can return the possible sign configurations.

*switchTrafficLights* of *SignController* calls *getSignConfigs* which will return no possible configurations if *allRed* is true. In this case all trafficlights are set to red.

### 4.3.5 Traffic light controllers of Carlos Gershenson

In the paper *Self-organizing Traffic Lights*, Carlos Gershenson presents the following control methods:

- Marching Control
- Optim Control

- Sotl-request control
- Sotl-phase control
- Sotl-platoon control

**Marching control** is a simple method. All intersections will have green lights in the same direction. Intersections have a phase which counts time steps. When the phase reaches a period value *p*, the phase is reset to zero. When $phi$ has the value 0, red lights turn green after the clearence time has passed, and green lights turn red. "Marching" intersections have equal phases.

In our simulation we have roads with traffic in both directions. We need a more complex solution for the traffic lights. The traffic lights have te be green in the following sequence.

- the lights on the primary lanes turning left
- the lights on the primary lanes going straight on and turning right
- the lights on the lanes crossing primary road turning left
- the lights on the lanes crossing primary road going straight on and turning right

This control method is implemented in *MorevtsMarchingTLC*. This class is a sublcass of *TLController*. The method *decideTLS* implements the actual algorithm. The variable $phase$ is added in *TrafficLight* with its accessors $getPhase$, $setPhase$ and $incrPhase$. This wil be used as a counter when the traffic lights have to switch.
For this method we need a global counter. $curCycle$ of SimModel would be good, but it is not accessible by a $TLController$. $curCycle$ is added in $TLController$ which has the same value as the $curCycle$ of $SimModel$. The $curCycle$ of the $TLController$ is updated by $SimModel$.

$decideTLs$ decides which traffic lights should turn green, and which should turn

red. There is a difference between a primary lane, and a secondary lane. Primary lanes get green at the first half of the period. Secondary lanes get the second half. This is again divided into turning left, and going straight on / right.

When a new traffic light controller is implemented, it has to be added in *TLCFactory*.

This method is tested in the simulator with a map containg a primary road with 2 intersections. Secondary roads are intersecting with this primary road. When running the simulator, you can see that the traffic lights of the nodes on the primary road have the same configuration at the same cycle.

**Optim control** assures (only in 1 direction) a green wave of lights on the primary road, once the roaduser has green.

Trafficlights at different nodes have a different $phase$. To keep track of the phase, the difference is stored in the $Junction$. This difference is the time needed to travel the distance of the lane in front of the light, $phaseDiff = \frac{lengthOfLane}{maxAllowedSpeed}$. The base of the phase is stored in $TLController$. The actual phase is calculated as the substraction: $phase = phaseBase - phaseDiff$. *setPhaseDiffs* of $MorevtsOptimTLC$ sets the *phaseDiff* of the nodes on the primary road. The calculation of the $phaseDiffs$ has to start at the beginning of the primary road for the green wave. The first traffic light has a $phaseDiff$ of 0, the second has a $phaseDiff_1 = t1$, the third has a $phaseDiff_2 = t1 + t2$, and so on.

In the editor the support is added for defining nodes which have to deliver a green wave and specifying the start and the finish of a green wave. The direction of the green wave is set by $start$ and $finish$. The roads in between need to be primary. In *Junction* is added: *isGreenWaveNode*, *isGreenWaveStart*, *setGreenWaveStart*, *isGreenWaveFinish* and *setGreenWaveFinish*. Those data members need to be stored/loaded by *load* and *saveSelf*. *EditJunctionPanel* is modified to support specifying the data. *SimJunctionPanel* is modified to see these green wave / start / finish values. This is succesfully tested.

The traffic light controller is implemented in *MorevtsOptimTLC*. *setInfrastructure* searches the path from start till finish of the green wave, calculates the phase differences and sets it in the junctions. *decideTLs* is the same as Marching control, except the phase is calculated for every node.

This traffic light controller is tested with different maps, and everything works properly.

**Sotl-request control** keeps a counter $kappa$ which is set to zero when the light turns red and then incremented at each time step by the number of cars that are approaching. When $kappa$ reaches a threshold $theta$, the light turns green. This controller is implemented in the class $MorevtsSotlRequest$

The code in $switchTrafficLights$, which chooses a random sign configuration if all gains are zero, is removed. When the light is switched, the current cycle is stored in $Sign$: $cycleSwitched$.

Every time $kappa$ needs to be incremented, $countRoadusers(Drivelane)$ is called, which counts the number of vehicles within a number of blocks (meters) for a given lane. This number of blocks is specified by $VISIBLE$.

$updateKappa()$ updates $kappa$ for all inbound lanes of all nodes. When the traffic light has been switched in the previous cycle, $kappa$ is set to zero and the $KeepTLDFlag$ is set to $true$. This will keep the current $TLDecision$ until $KeepTLDFlag$ is set to $false$.

$decideTLs()$ will decide what the traffic light should do. It first calls $updateKappa()$ which will reset $kappa$ when needed. Then for all $TLDecisions$ the counter $kappa$ is updated if the traffic light is red, by adding the count of cars. If $kappa$ exceeds the threshold $theta$, the q-value of the TLDecision is set to $kappa$ and the $KeepTLDFlag$ is set to $false$ which allows switching of the traffic lights. When $kappa$ has not reached $theta$, the Q-value of the $TLDecision$ will be set to

0. The source code of decideTLs() is available in the appendix.

$KeeptLDFlag$ is used because traffic lights have to keep their configuration until a request for green light occurs. The use of a flag is needed because $switchTrafficLights$ of $SignController$ will search for a new traffic light configuration when the flag is not used, or when the flag is set to false.

**Sotl-phase control** is the same as $sotl-request$, except that decisions of traffic lights have to hold for a minimum phase.

In $updateTLDs()$ all updates are done of $keepTLDFlag$ and $phaseMinimal$ of $Node$, and $kappa$ of $TLDecision$. When a traffic light has been switched to green in the previous cycle for a specific node, the $keepTLDFlag$ of that node is set to $true$ and the minimal green time is set to *PHASE_MIN*. $kappa$ of $TLDecision$ is set to zero for all traffic lights which are switched to green.

$decideTLs()$ will first call $updateTLDs()$ for updating the parameters. Then for all nodes it will update the time the green configuration has to be kept. For all inbound lanes it will add the count of cars to $kappa$ and it will check if this $kappa$ has reached its threshold. When this is the case and the minimal green time has been reached, the $Q-value$ of $TLDecision$ will be set to $kappa$ and $keepTLDFlag$ is set to $false$ for allowing the traffic lights to switch. If $kappa$ did not reach the threshold or the minimal green time has not been reached, the $Q-value$ is set to zero.

The testing of this traffic light controller was succesfull.

**Sotl-platoon control** adds two restrictions to sotl-phase for regulating the size of platoons. Before changing red lights to green, it checks if a platoon is not crossing through, in order not to break it.

A red light is not changed to green if there is at least one car on the crossing street approaching within $omega$ patches from the intersection. For our simula-

tion it is needed to find an alternative but equivalent restriction. This could be translated as a green light is not turned red if there is at least one car approaching within $omega$ patches from the intersection.
The second restriction can be kept: restriction one is not taken into account if there are more than $mu$ cars approaching the intersection.

These restrictions are implemented in $updateTLDs$. When the minimal phase is almost at its end, and there is at least one car but no more than $mu$ cars approaching within omega patches, the minimal phase is incremented.

Testing succeeds. (had to fix bug in simModel where speed is changed. when the light is green and the first roaduser could cross the node, it has to slow down if the destination lane is full.)

### 4.3.6 Marching and Optim for the Wetstraat

Because the traffic flow in the Wetstraat is in one direction towards the centre, the marching and optim controller can be made more efficient. The green time on the Wetstraat has to be longer than the green time for the crossing roads. The crossing roads of the Wetstraat are in one direction and because of this the green time can be divided into green for the Wetstraat and green for the crossing roads.

The green times are the same as the green times in the Wetstraat. This is done to be able to compare the current traffic light controller with the proposed self-organizing traffic light controller.

### 4.3.7 Other traffic light controllers

The following traffic light controllers were created because of wrong interpretation, but are also self-organizing traffic lights.

**Counting-cars-1 control** counts the cars in front of the traffic lights. The traffic light configuration with the highest count of cars is chosen.

**Counting-cars-2 control** same as Counting-cars-1 but when a configuration has been chosen, it stays for a given moment. In this way, the traffic lights are not switching too fast.

### 4.3.8 Automatic simulations

For testing diffent maps, traffic light controllers and parameters, it is easier to write a program. This is done in $GLDSimTester$. It is possible to sum the different maps, the different traffic light controllers and the different parameters which have to be tested by simulation. It will start the simulation for every combination of a map, controller and parameters. After 3600 cycles, it will export the simulation data to files with appropriate names. The average total waiting time and the waiting queues of the nodes will be exported.

Every simulation with specific parameters is done 5 times.

The average trip waiting time files contain all intermediate ATWT values at different cycles, until cycle 3600.

The waiting queues are given for every edge-node.

### 4.3.9 Processing simulation data

When testing a few maps, traffic densities, controllers and parameters which have to be done multiple times, results are shown in a lot of export files. For every combination there will be 5 files for ATWT and 5 for waiting queues. The values of those files need to be collected and the average values have to be calculated and written to a new file. This is done by $DataProcessor$.

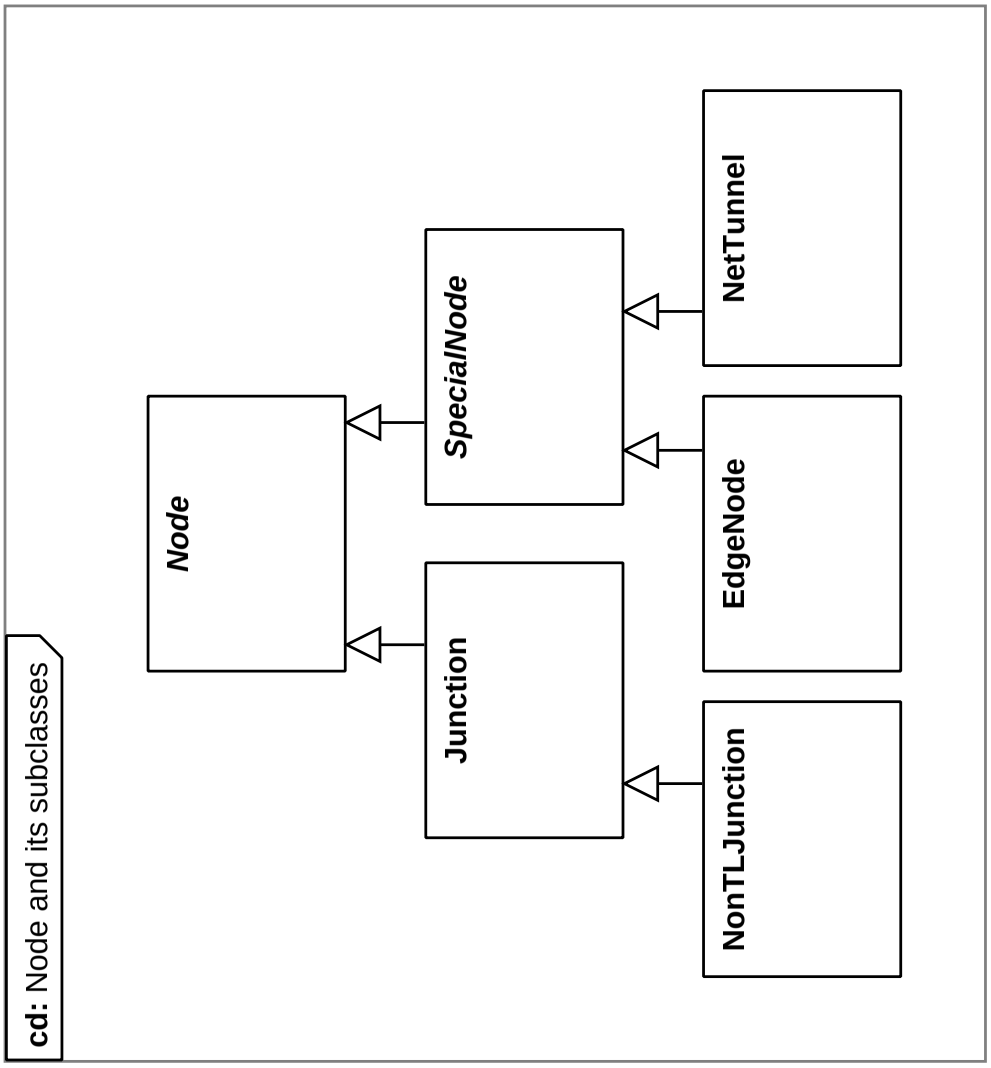


Figure 4.2: UML diagram: Node and its subclasses

# Chapter 5

# Simulation results

The proposed self-organizing traffic light controllers are tested by traffic simulations. Two other traffic light controllers, *optim* (green wave) and *marching*, will be used to compare those self-organizing controllers with non self-organizing controllers.

All simulations are done five times and the average values of the simulation results are given in the following sections. The resulting data consists of *ATWT* and *TWQL*. Those values are taken after 3600 cycles (representing 1 hour) of simulation. The *ATWT* is the average total waiting time of the road users in the simulation. The edge-nodes are the borders of the scenario. Those edge-nodes spawn road users into the scenario. When the traffic density is high and the road connected to the edge-node is full, spawned road users cannot enter and those road users are queued in the edge-node. The *TWQL* (total waiting queue length) is the sum of all queued road users in the edge-nodes.

Every map has its spawn and destination frequencies. Those frequencies define how many road users an edge-node will produce, and where those road users will go. For every edge-node there is one spawn frequency and $n - 1$ destination frequencies (one for every possible destination). A spawn frequency is a value between zero and one. If it is zero, no vehicles will be spawned. If it is one, the edge-node will spawn a vehicle at every $cycle$.

The first scenario is a simple road map with 2 junctions and traffic in all directions. The second scenario contains 5 junctions with traffic in all directions.

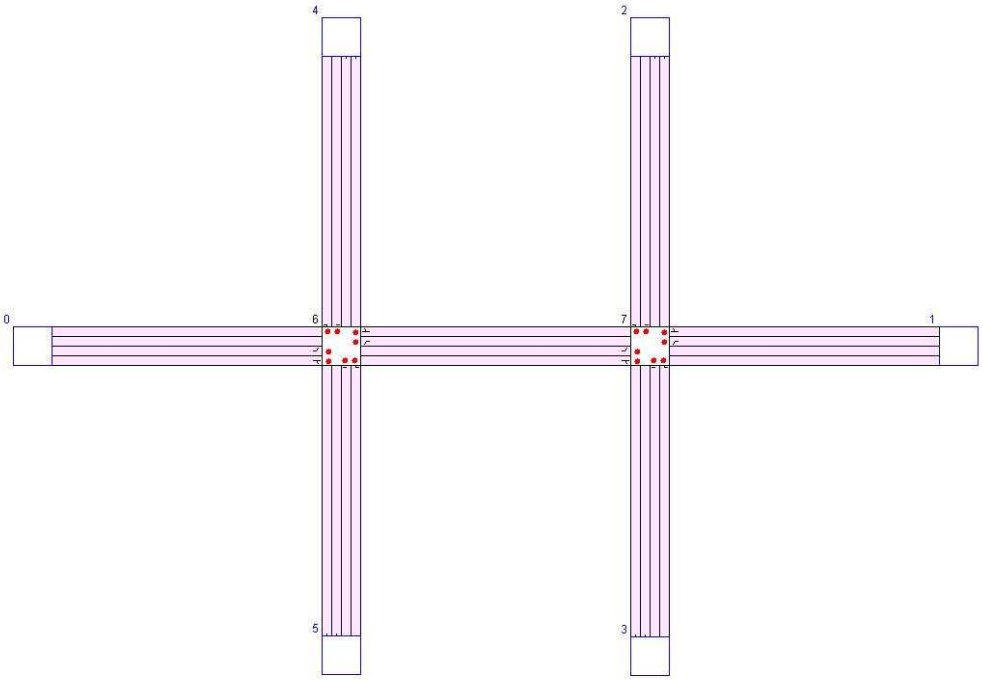


Figure 5.1: Map1 with 2 junctions

Those two scenarios are chosen to experiment with the traffic light controllers. The last scenario represents the Wetstraat in Brussels and is created as realistic as possible. The optim controller for the simulation of the Wetstraat is modified such that they have the same green times as in reality. The simulation results of the self-organizing traffic lights are compared to the current traffic light controller of the Wetstraat.

## 5.1 Scenario 1: 2 junctions, all directions

This scenario exists of 2 junctions, 6 edge-nodes and 7 roads (two directions). Two different traffic densities will be used for simulation: $map1\_hd$ for high density and $map1\_ld$ for low density. Scenario 1 at high and low traffic density has the same set of destination frequencies which is shown in table 5.1 on the following page.

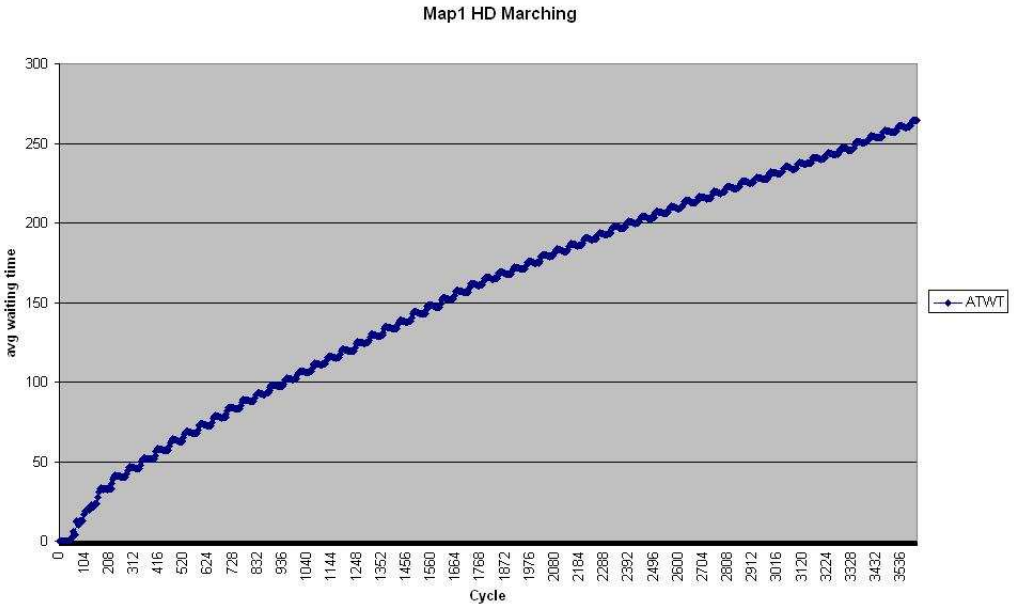


Figure 5.2: Map1 - high density - marching control - ATWT

| to | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | x | **0.6** | 0.1 | 0.1 | 0.1 | 0.1 |
| 1 | **0.6** | x | 0.1 | 0.1 | 0.1 | 0.1 |
| 2 | 0.2 | 0.2 | x | **0.4** | 0.1 | 0.1 |
| 3 | 0.2 | 0.2 | **0.4** | x | 0.1 | 0.1 |
| 4 | 0.2 | 0.2 | 0.1 | 0.1 | x | **0.4** |
| 5 | 0.2 | 0.2 | 0.1 | 0.1 | **0.4** | x |

Table 5.1: Destination frequencies of map1

For the low traffic density the spawn frequencies of edge-node 0 and 1 are set to 0.2 and for the other edge-nodes it is set to 0.1. For high traffic density the spawn frequencies of all edge-nodes are set to 0.25.

### 5.1.1 Marching controller

**High traffic density** The marching control yields after 3600 cycles (1 hour) an average trip waiting time (ATWT) of 264 cycles. The Waiting queues are big in edge-node 0 and 1 with values above 600 vehicles. The total count of waiting road

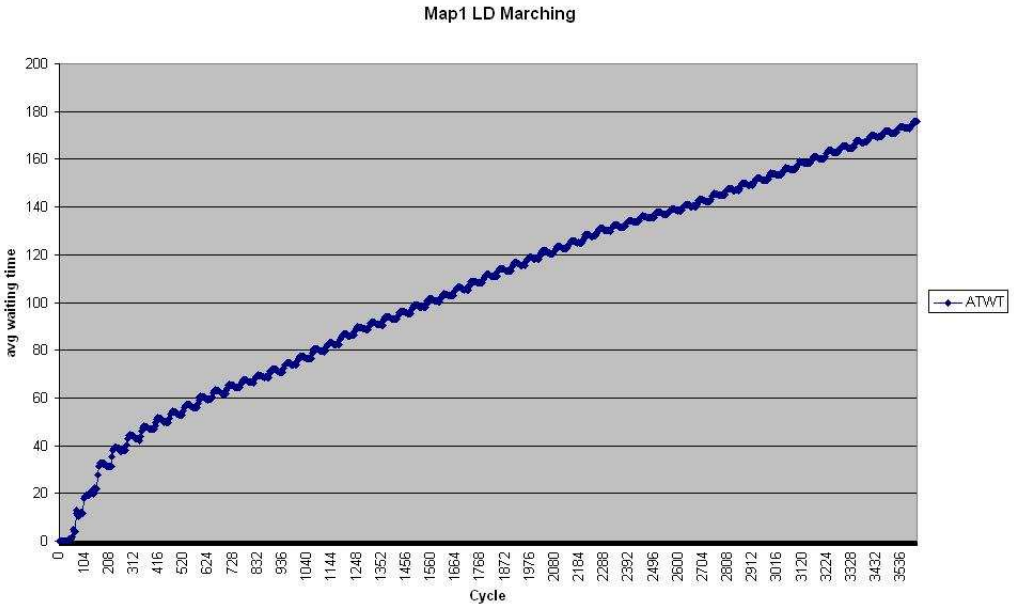


Figure 5.3: Map1 - low density - marching - ATWT

users in those queues (TWQL) is 1434.

**Low traffic density** The marching control has an ATWT of 175 and a total waiting queue length (TWQL) of 428.

### 5.1.2 Optim controller

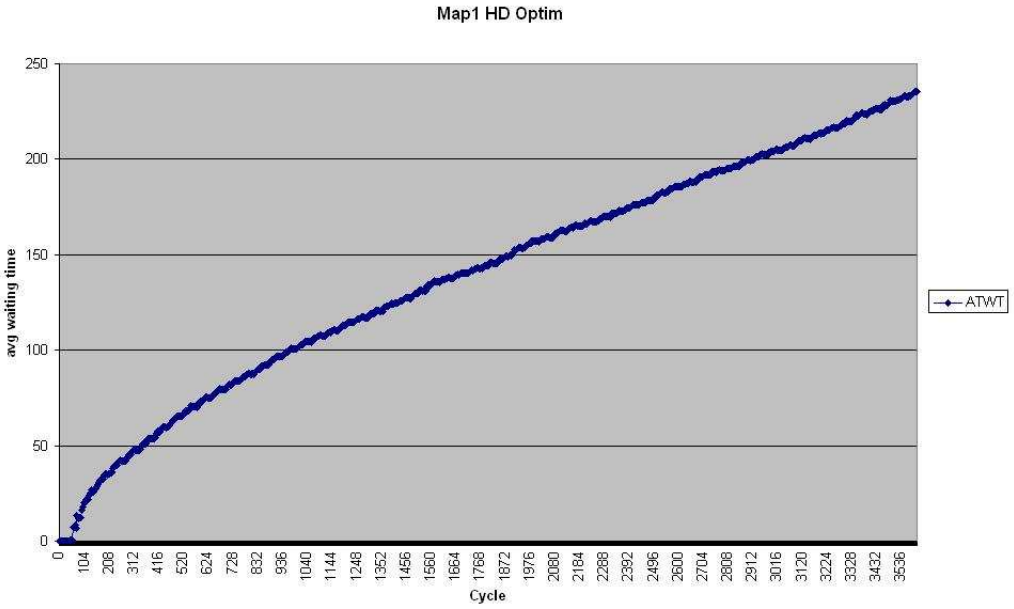


Figure 5.4: Map1- high density - optim control - ATWT

**High traffic density** The optim control has an ATWT of 235 cycles, but the total count of roadusers in the waiting queues is 1790, which is more than with the marching control. The waiting queue for edge-node 0 is only 295 and for edge-node 1 it is 628. Those values show us that the controller is optimizing the traffic in one direction, which is the direction of the green wave. Edge-node 5 has a very big waiting queue of 754. This is because the traffic is optimized from edge-node 0 to edge-node 1 and the destination frequencies of edge-node 5 to 4 is 0.4 and from edge-node 5 turning right is 0.4. What happens is that the lane from junction 6 to 7 is filled easier and the traffic from edge-node 5 turning right is blocking the lane for the traffic going to edge-node 4.

**Low traffic density** The Optim control has an ATWT of 216 and a total waiting queue length of 481.


Figure 5.5: Map1 - low density - optim - ATWT

### 5.1.3 Sotl-request controller

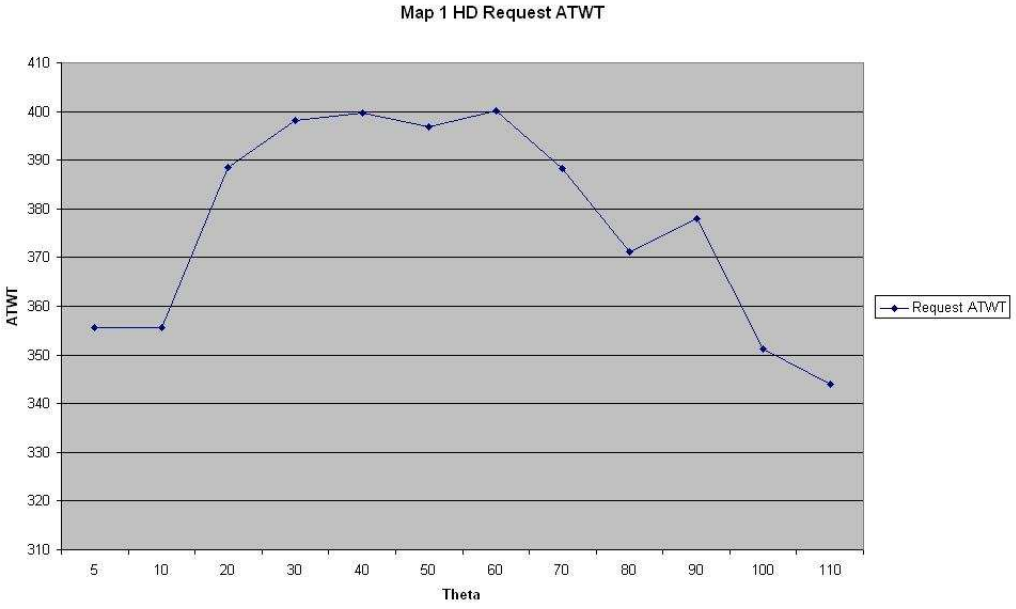


Figure 5.6: Map1- high density - Request control - ATWT

**High traffic density** The sotl-request controller has the lowest ATWT value (343.94) for $\theta = 110$. For this $\theta$ it has also the lowest TWQL (1457). Figure 5.6 on the previous page gives the lowest ATWT values for very low $\theta$ or big values of $\theta$. For $\theta = 5$, the traffic lights are switching very fast because the threshold is reached immediately. It is more useful to take a bigger $\theta$ because the green times will be longer and the best results are found. The length of the waiting queues is the lowest (1457) for $\theta = 110$ and the highest (1864) for $\theta = 50$.

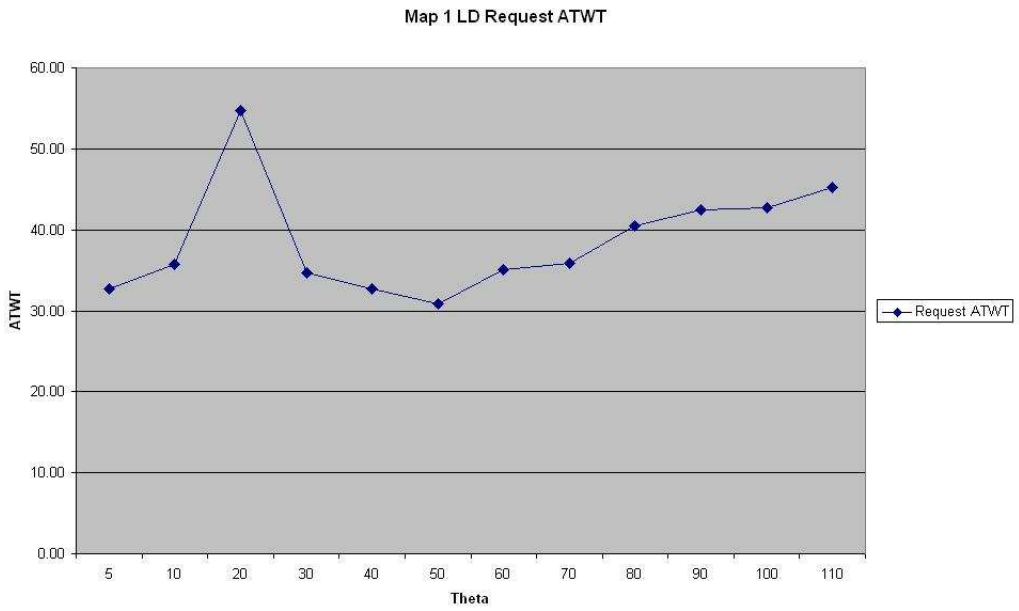


Figure 5.7: Map1- low density - Request control - ATWT

**Low traffic density** The best ATWT for the request controller at low traffic density is given for $\theta = 50$. This gives a value (ATWT) of 30.92 and no waiting road users in the queues. The highest value is given for $\theta = 20$ with an ATWT of 54.72. (see figure 5.7)

### 5.1.4 Sotl-phase controller

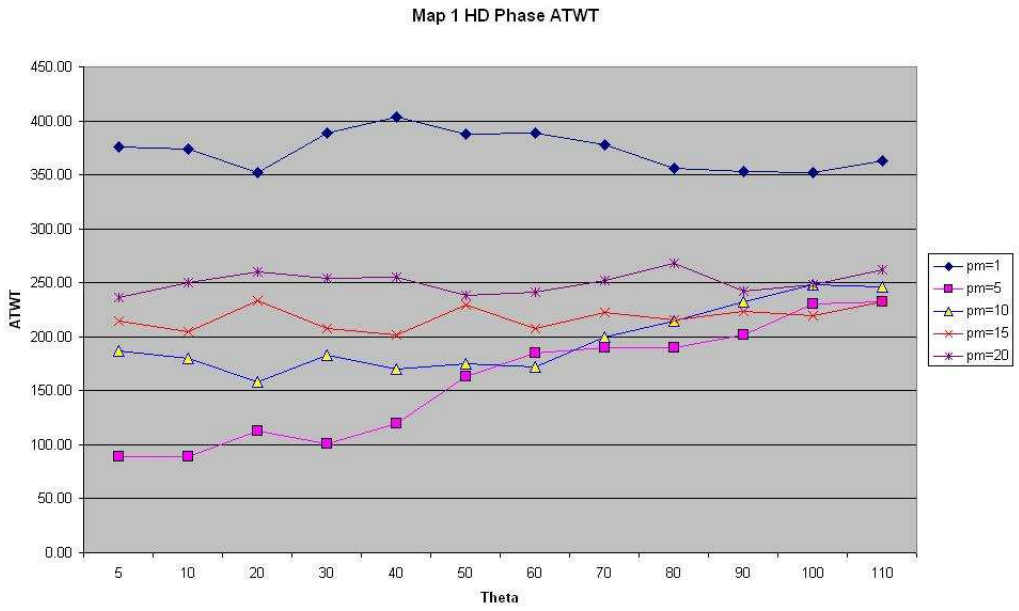


Figure 5.8: Map1- high density - Phase control - ATWT

**High traffic density** The sotl-phase controller has the lowest ATWT if the green time ($\varphi_{min}$) is set to 5 seconds. This is visible in figure 5.8. $\varphi_{min} = 5$ and $\theta = 10$ results in an ATWT of 88.88 and a TWQL of 144. The value of $\varphi_{min}$ has a big influence on the ATWT for low values of $\theta$. For larger values of $\theta$, the ATWT values are almost the same, except for $\varphi_{min} = 1$. This is also true for the TWQL values.

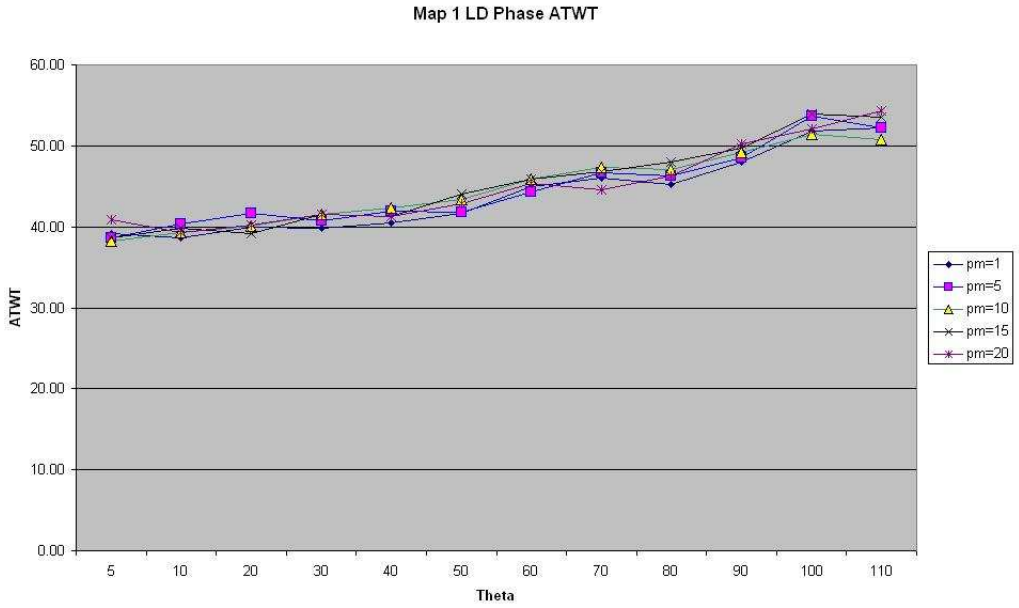


Figure 5.9: Map1- low density - Phase control - ATWT

**Low traffic density** Figure 5.9 shows that all values of $\theta$ give almost the same result in ATWT. At low traffic density, the $\theta$ has almost no influence on the ATWT value. The queues are emtpy for all values of $\theta$. The best ATWT is found for $\theta = 5$ and $\varphi_{min} = 10$, which gives a value of 38.21. The results for $\theta = 10$ and $\varphi_{min} = 1$, and $\theta = 5$ and $\varphi_{min} = 5$ are also good.

### 5.1.5 Sotl-platoon controller

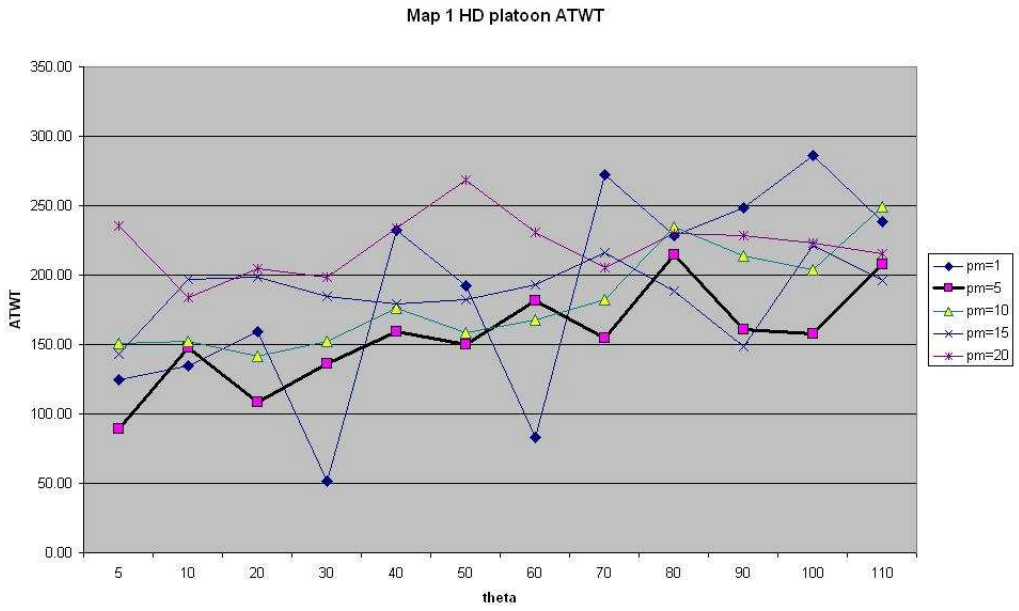


Figure 5.10: Map1- high density - Platoon control - ATWT

**High traffic density** The sotl-platoon controller has the best results if the green time is set to 5 seconds. When $\theta$ is set to 5 and $\varphi_{min}$ is also 5, the simulator gives an ATWT of 89.22 and an TWQL of 155. Figure 5.10 shows a slowly increasing ATWT value for growing $\theta$. The graph for $\varphi_{min} = 1$ is jumping up and down and shows that the controller with that $\varphi_{min}$ value is unstable.

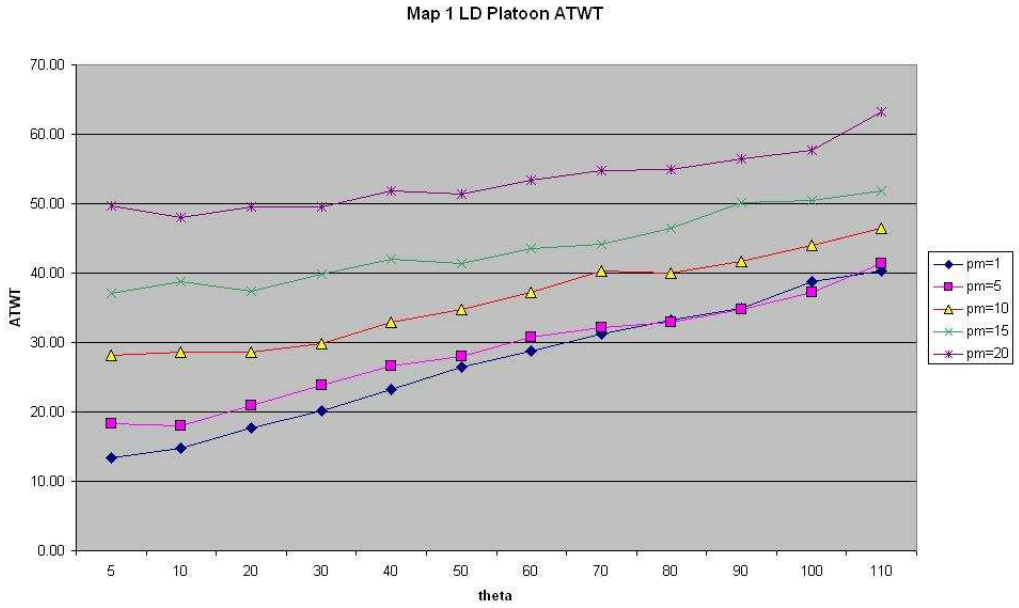


Figure 5.11: Map1- low density - Platoon control - ATWT

**Low traffic density** The platoon controller has the best ATWT values for $\varphi_{min}$ set to 1 or 5. The difference is very small but $\varphi_{min} = 5$ should be preferred because $\varphi_{min} = 1$ can sometimes cause very fast switching of the traffic lights. This gives for $\theta = 5$ an ATWT value of 18.33 and empty queues. The highest value is given for $\varphi_{min} = 20$ and $\theta = 110$. Figure 5.11 shows that all values are increasing for increasing $\theta$ and $\varphi_{min}$. Thus lower $\theta$'s give the best results

### 5.1.6 Summary

| controller | density | ATWT | TWQL |
|---|---|---|---|
| marching | HD | 264 | 1434 |
| optim | HD | 235 | 1790 |
| request | HD | 344 | 1457 |
| phase | HD | 89 | 144 |
| platoon | HD | 89 | 155 |
| marching | LD | 175 | 428 |
| optim | LD | 216 | 481 |
| request | LD | 31 | 0 |
| phase | LD | 38 | 0 |
| platoon | LD | 18 | 0 |

Table 5.2: Scenario 1: Best results

The self-organizing traffic light controllers are better than the optim and marching controller, except for sotl-request at high traffic density. The sotl-request controller for high traffic density is worse than the optim and marching controller. This is because at high traffic density requests are granted almost immediately. This results in very fast switching of the traffic lights. The value of $\theta$ can regulate the green times of the traffic lights. This is why the highest tested value for $\theta$ gives the best ATWT value (Average Trip Waiting Time).

The sotl-phase controller is better than the optim and marching controller for both traffic densities. Compared to the optim controller, the sotl-phase controller at high traffic density reduces the ATWT with 62% and the TWQL (Total waiting queue length) with 91%. At low traffic density, this is a reduction of 82% for the value of ATWT and an elimination of the TWQL value.

At high traffic density it can be concluded that the sotl-platoon controller is better than optim . Compared to the optim controller, the sotl-platoon controller has reduced the ATWT with 62% and the TWQL with 90%.
The best controller for low traffic density is the sotl-platoon controller. It reduces the ATWT with 90% compared to the optim controler and the TWQL is eliminated.

Concluding, at both densities the platoon controller is a lot better than the optim controller.

## 5.2 Scenario 2: 5 junctions, all directions

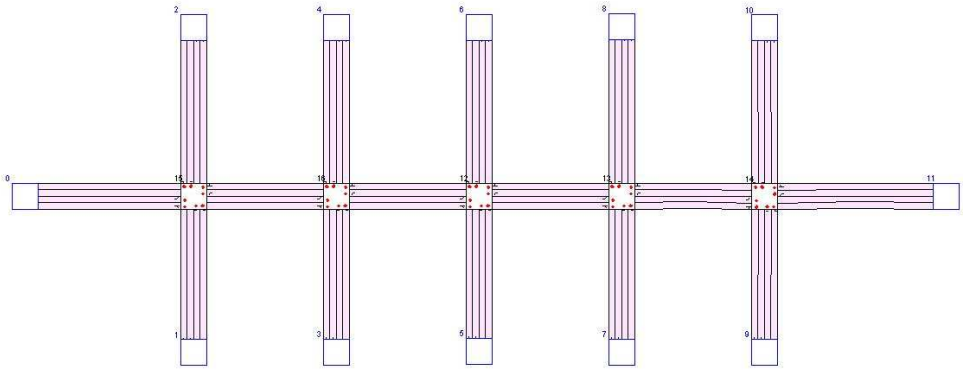

Figure 5.12: Map2 with 5 junctions

The second map has 5 junctions (see figure 5.12). The green wave is set for the sequence of junctions 8, 9, 7, 11, 10.

**Spawn and destination frequencies** The destination frequencies for map2 are in table 5.3.

| to | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | x | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | **0.5** |
| 1 | 0.05 | x | **0.5** | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 |
| 2 | 0.05 | **0.5** | x | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 |
| 3 | 0.05 | 0.05 | 0.05 | x | **0.5** | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 |
| 4 | 0.05 | 0.05 | 0.05 | **0.5** | x | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 |
| 5 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | x | **0.5** | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 |
| 6 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | **0.5** | x | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 |
| 7 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | x | **0.5** | 0.05 | 0.05 | 0.05 |
| 8 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | **0.5** | x | 0.05 | 0.05 | 0.05 |
| 9 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | x | **0.5** | 0.05 |
| 10 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | **0.5** | x | 0.05 |
| 11 | 0.5 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | 0.05 | x |

Table 5.3: Destination frequencies for map2

The spawn frequencies for high traffic density are set with the following values

- edge-node 0 is set to 0.25
- edge-nodes 1-10 are set to 0.1
- edge-node 11 is set to 0.15

The spawn frequencies for low traffic density are set with the following values

- edge-node 0 is set to 0.15
- edge-nodes 1-10 are set to 0.05
- edge-node 11 is set to 0.1

### 5.2.1 Marching controller

**High traffic density** The marching controller has an ATWT of 304 and the TWQL is 2346. Edge-node 0 has the most waiting roadusers in its waiting queue (842), followed by edge-node 11 with 492 waiting roadusers in its queue.

**Low traffic density** The marching controller yields an ATWT of 282 and a TWQL of 1128. Edge-node 0 and 11 have the biggest queues, 484 and 301 respectively.

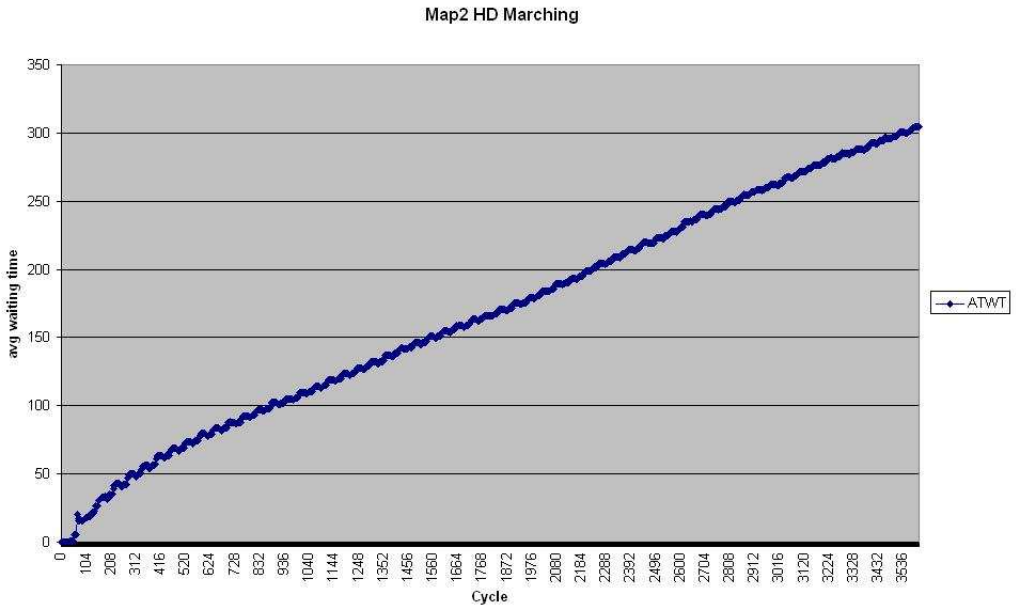


Figure 5.13: Map2 - high density - Marching control - ATWT

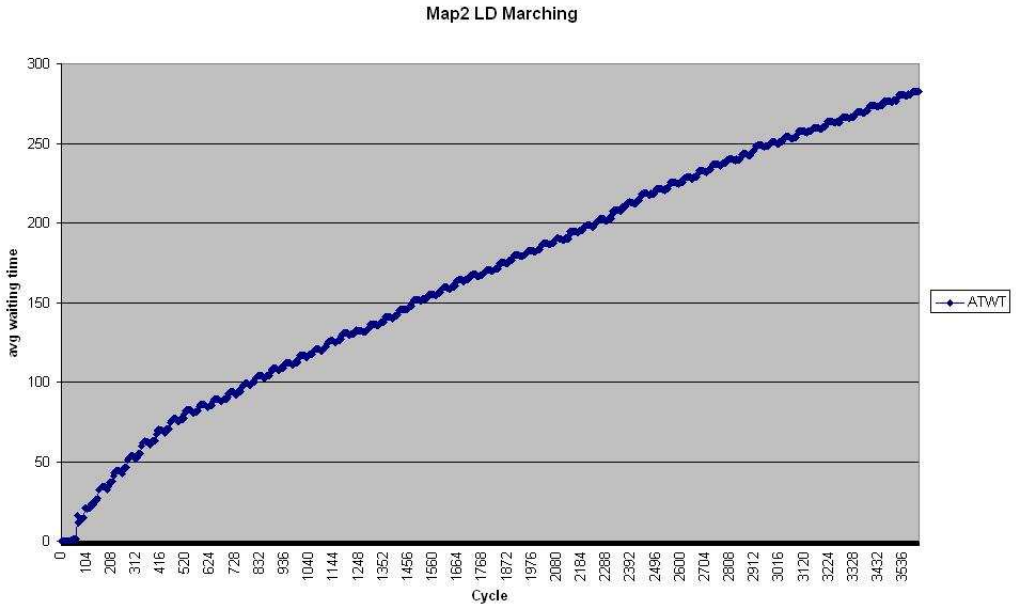


Figure 5.14: Map1 - low density - Marching controller - ATWT

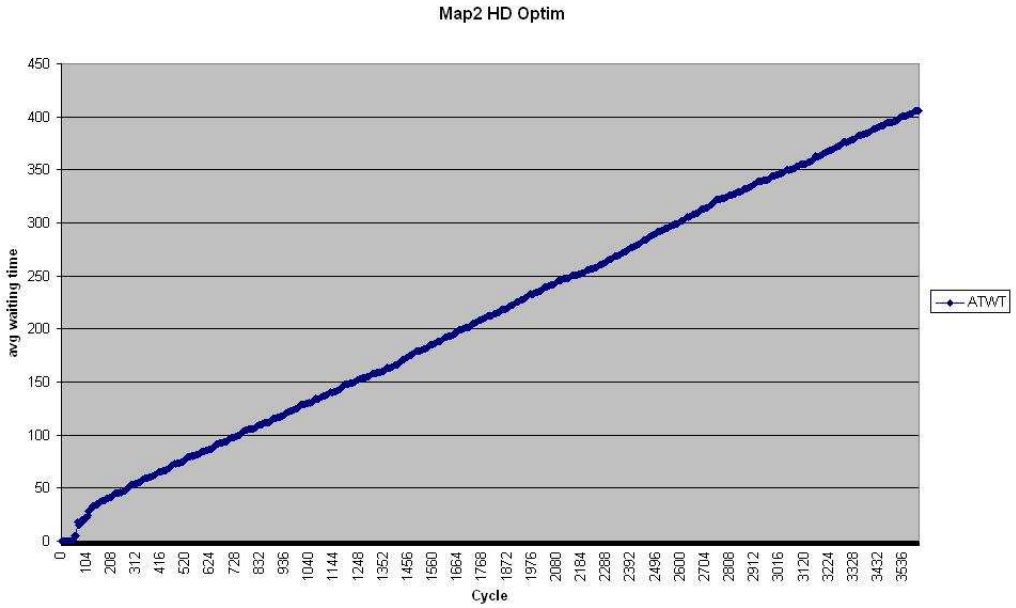


Figure 5.15: Map2 - high density - optim ATWT

### 5.2.2 Optim controller

**High traffic density** The optim controller has an ATWT of 406 and the TWQL is 2467. (See figure 5.15 for the waiting times)

**Low traffic density** The optim controller has an ATWT of 332 and a TWQL of 1040. With this controller, edge-node 11 has the biggest waiting queue (372). This also shows that the traffic from edge-node 0 to 11 has an advantage to the traffic from edge-node 11 to 0. The spawn-frequency of edge-node 0 is 0.15 and for edge-node 11 it is 0.1.


Figure 5.16: Map2 - low density - optim - ATWT

### 5.2.3 Sotl-request controller

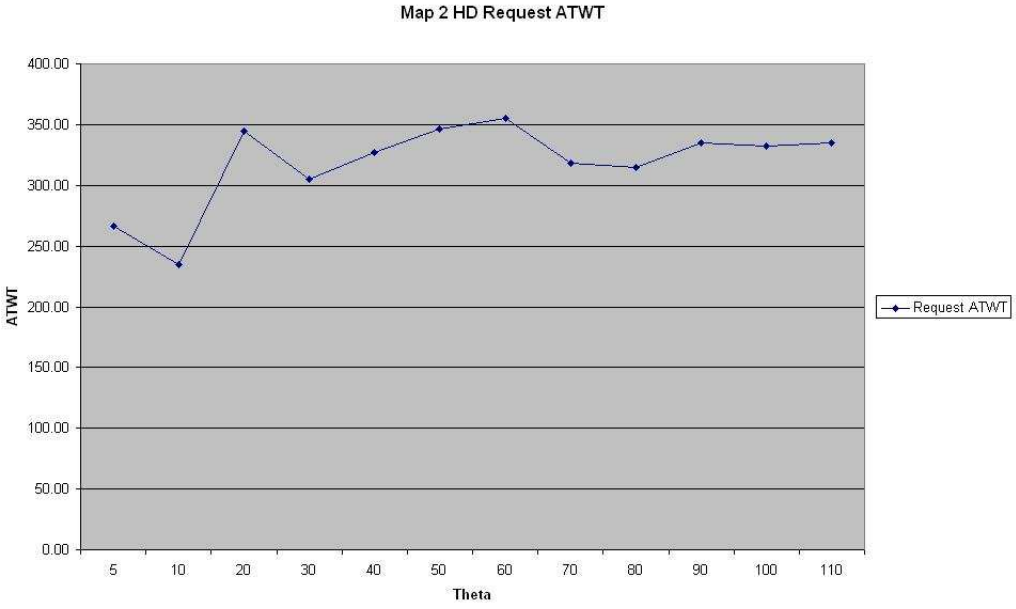


Figure 5.17: Map2- high density - Request control - ATWT

**High traffic density** The sotl-request controller has the best result for $\theta = 10$ with an ATWT value of 234.35 and a TWQL of 1294. The worst result is for $\theta = 60$ with an ATWT of 355.12 and a TWQL of 2385. For $\theta = 10$, edge-node 0 has the biggest waiting queue (648) followed by edge-node 11 (waiting queue of 236). For all other values of $\theta$ the waiting queues are the biggest for edge-nodes 0 and 11.

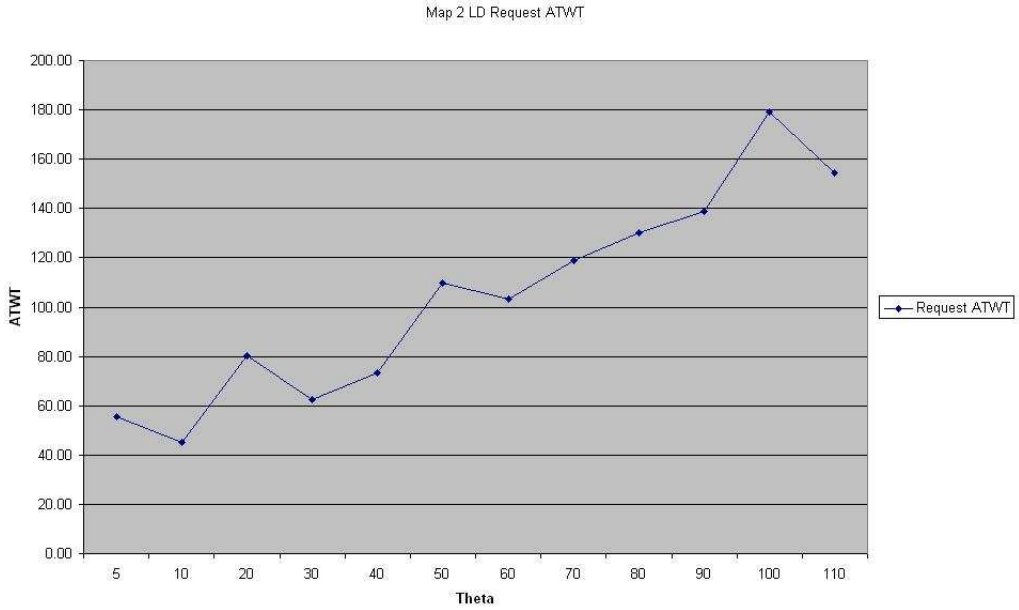


Figure 5.18: Map2- low density - Request control - ATWT

**Low traffic density** The best result is given for $\theta = 10$ with an ATWT of 45.19 and TWQL of 6.

### 5.2.4 Sotl-phase controller

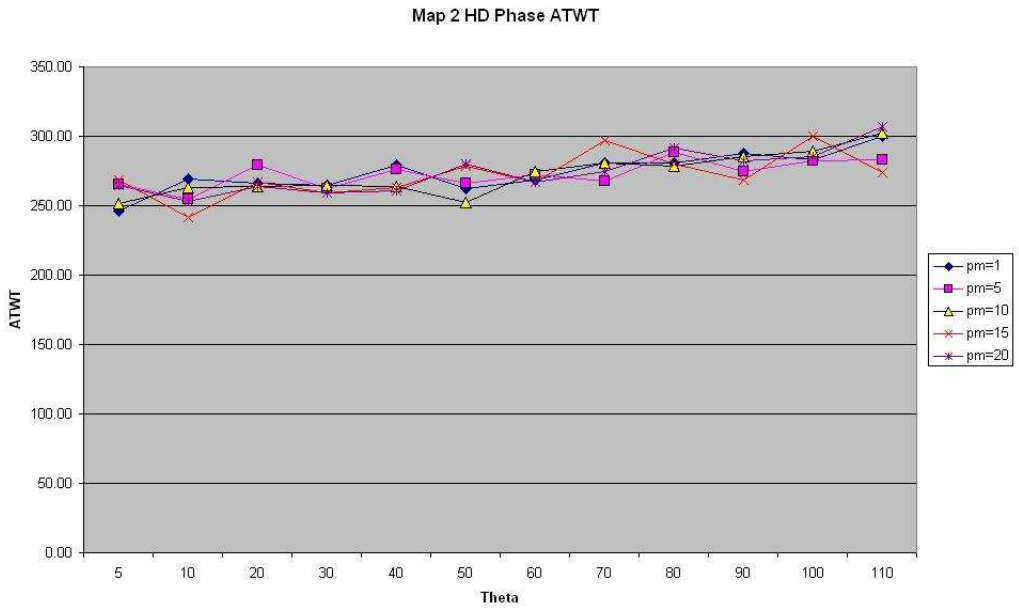


Figure 5.19: Map2- high density - Phase control - ATWT

**High traffic density** Figure 5.19 indicates that sotl-phase is robust to changes in $\theta$ and $\varphi_{min}$. The best ATWT is found for $\varphi_{min} = 15$ and $\theta = 10$ and gives a value of 241.64, and the corresponding TWQL is 1164. For $\varphi_{min} = 5$ and $\theta = 10$, the ATWT has a value of 254.59 and a TWQL of 1166.

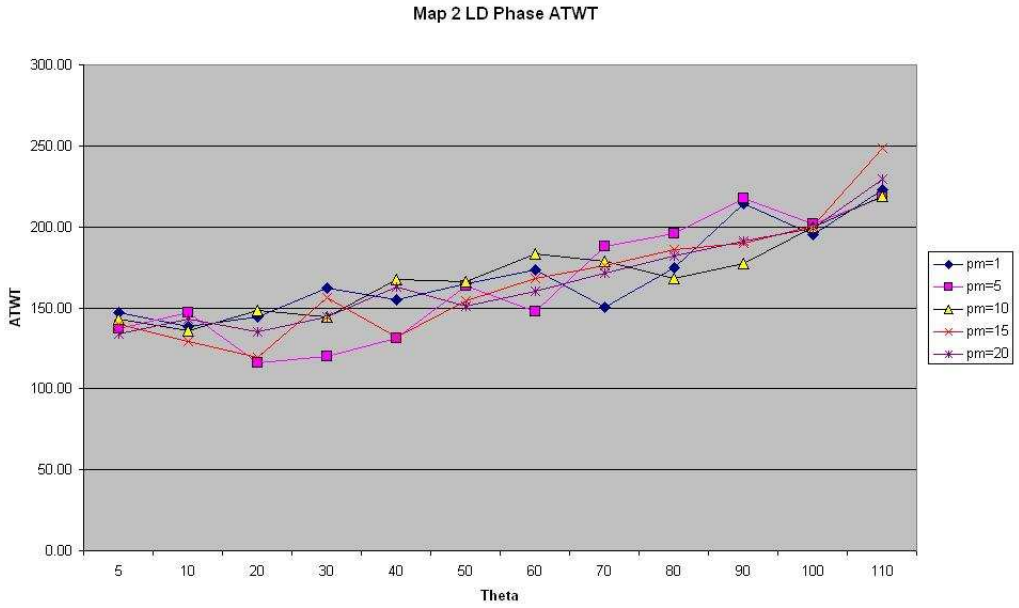


Figure 5.20: Map2- low density - Phase control - ATWT

**Low traffic density** Figure 5.20 shows that $\varphi_{min}$ has almost no effect on the ATWT and that a lower value for $\theta$ yields better ATWT values. The best result is given for $\varphi_{min} = 5$ and $\theta = 20$ with an ATWT of 116.11 and TWQL of 101.

### 5.2.5 Sotl-platoon controller

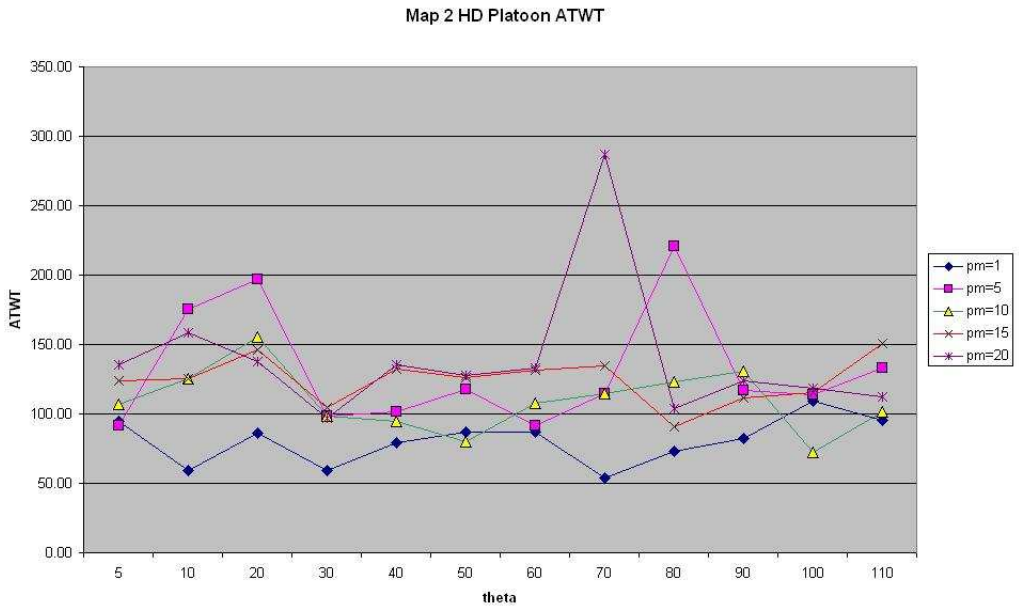


Figure 5.21: Map2- high density - Platoon control - ATWT

**High traffic density** Figure 5.21 shows that the ATWT value is not as predictable for platoon controller as for sotl-phase. For $\varphi_{min} = 1$ and $\theta = 70$, the graph shows the best results with the lowest ATWT value of 54.17 and a TWQL of 5484. For $\varphi_{min} = 10$ and $\theta = 100$, the lowest ATWT value is 72.29 with TWQL of 5241. These are very high values for TWQL. This implies that the calculated values for ATWT are not satisfactory to represent the real ATWT of $all$ road users. The ATWT values are calculated for the road users that $arrive$ at their destination. Because most of the road users do not reach their destination, those road users will not be taken into account.

In the beginning of the simulation, everything works as expected, but the roads get filled up. When all main stream lanes are full, the system gets blocked. Traffic light configurations stay the same, because the conditions of a *platoon is crossing* are satisfied. This can happen when a few longer vehicles are waiting before a green light and cannot move because the next road is full.

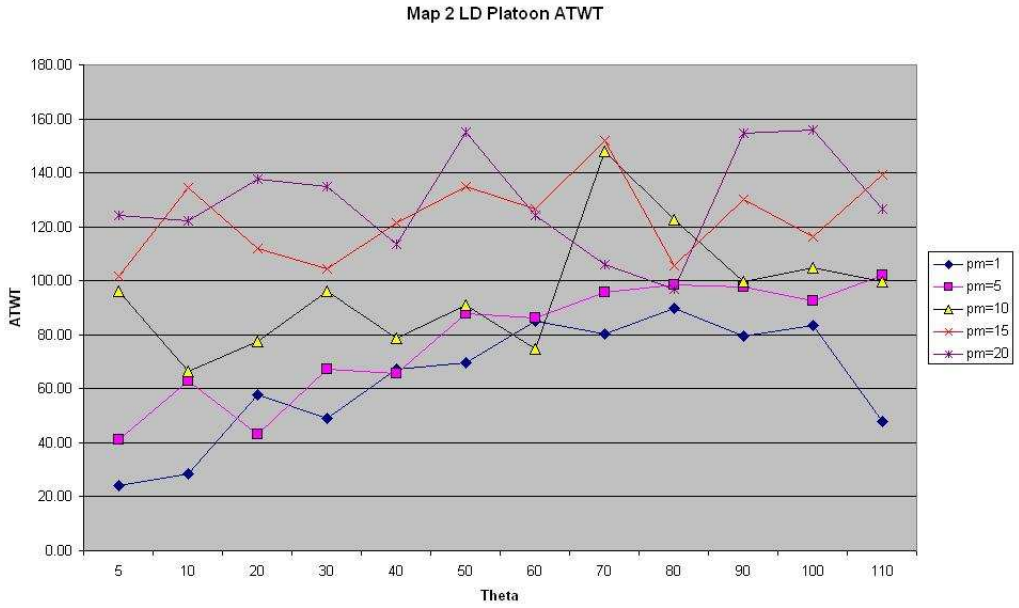


Figure 5.22: Map2- low density - Platoon control - ATWT

**Low traffic density** For $\varphi_{min} = 1 or 5$ and $\theta = 5$, the ATWT has the lowest value. The ATWT has a value of 24.18 for $\varphi_{min} = 1$, and 41.07 for $\varphi_{min} = 5$. The corresponding TWQL are respectively 0 and 421.

### 5.2.6 Summary

| controller | density | ATWT | TWQL |
|---|---|---|---|
| marching | HD | 304 | 2346 |
| optim | HD | 406 | 2467 |
| request | HD | 234 | 1294 |
| phase | HD | 242 | 1164 |
| platoon | HD | x | x |
| marching | LD | 282 | 1128 |
| optim | LD | 332 | 1040 |
| request | LD | 45 | 6 |
| phase | LD | 116 | 101 |
| platoon | LD | 24 | 0 |

Table 5.4: Scenario 2: Best results

The request, phase and platoon controller are better than the optim controller.

The request controller at high traffic density reduces the ATWT with 51% and the TWQL with 47% compared with the optim controller. For low traffic density it is a reduction of 86% for ATWT and 99% for TWQL.

For the phase controller the reduction is 50% for ATWT and 52% for TWQL, at high traffic density. At low traffic density, the controller reduces the ATWT with 65% and the TWQL with 90%.

For the platoon controller a problem is discovered at high traffic density. When all roads are congested, and the conditions for *platoon is crossing* are satisfied, almost all traffic is blocked. This problem can be solved with an extra rule for the controller (one of the following):

- the case *platoon is crossing* is stopped when a maximum time has reached.
- when a possible destination lane is full, the case *platoon is crossing* is stopped when a maximum time has reached.

The first rule can cut platoons when no problem occurs, the second rule need sensors at all inputs and outputs of a junction but will not cut a platoon when no problem occurs. It is also possible to give a bigger value to busses or other longer vehicles when the condition *platoon is crossing* is checked.

The platoon controller for low traffic density reduces the ATWT with 92% and eliminates the waiting queues.

## 5.3 Scenario 3: Wetstraat

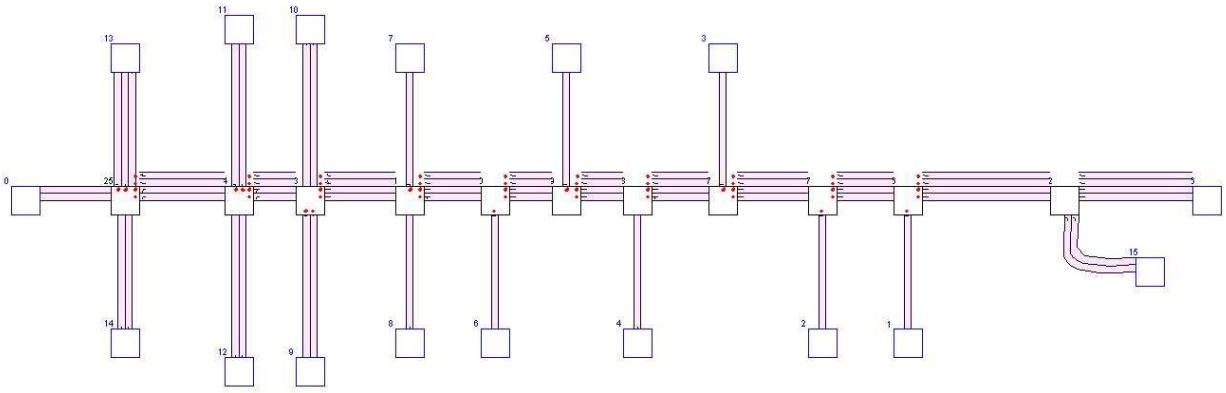

Figure 5.23: Map3 - Wetstraat

The simulations for the Wetstraat will be done for different traffic densities during the day. Those traffic densities are represented in table 5.5. The values in this table are the vehicle counts at the beginning of the Wetstraat. Theses counts are done by *Ministry of the Brussels-Capital region*. The spawn frequencies will be calculated based on those counts.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 476 | 255 | 145 | 120 | 175 | 598 | 2933 | 5270 | 4141 | 4028 | 3543 | 3353 |

| 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 3118 | 3829 | 3828 | 3334 | 3318 | 3519 | 3581 | 3734 | 2387 | 1690 | 1419 | 1083 |

Table 5.5: Vehicle count of Wetstraat

Carlos Gershenson and I made a field trip to the Wetstraat to get an idea of the destination frequencies of the Wetstraat. We counted the cars from the Wetstraat turning into crossing streets, and cars entering the Wetstraat from the crossing streets. At the intersection of the Wetstraat and the Kunstlaan we used a camera to help counting the cars.

For the intersection of Wetstraat with Trierstraat

- From Wetstraat → straight on: 200
- From Trierstraat → turning left (Wetstraat): 16

For the intersection of Wetstraat with Aarlenstraat

- From Wetstraat → straight on: 200
- From Aarlenstraat → turn left (Wetstraat): 9

For the intersection of Wetstraat with Spastraat

- From Wetstraat → straight on: 200
- From Spastraat → turn right (Wetstraat): 10

For the intersection of Wetstraat with Wetenschapsstraat

- From Wetstraat → straight on: 200
- From Wetstraat → turn left (Wetenschapsstraat): 10

For the intersection of Wetstraat with Tweekerkenstraat

- Tweekerkenstraat was closed. Work in progress.

For the intersection of Wetstraat with Nijverheidsstraat

- From Wetstraat → straight on: 200
- From Nijverheidsstraat → turn left (Wetstraat): 5

For the intersection of Wetstraat with Handelsstraat

- From Wetstraat → straight on: 200
- From Wetstraat → turn left (Handelsstraat): 8
- From Handelsstraat → straight on (Handelsstraat): 6
- From Handelsstraat → turn right (Wetstraat): 4

For the intersection of Wetstraat with Kunstlaan:

- from Wetstraat → turn left: 99
- from Wetstraat → turn right: 106
- from Wetstraat → straight on: 56
- from Kunstlaan South → straight on: 14
- from Kunstlaan South → turn left: 6 (3 U-turns)
- from Kunstlaan North → straight on: 22
- from Kunstlaan North → turn right: 5

For the intersection of Wetstraat with Hertogstraat

- From Wetstraat → straight on: 61
- From Wetstraat → turn left: 20
- From Wetstraatt → turn right: 2
- From Hertogstraat → straight on: 10
- From Hertogstraat → turn right: 1

The calculation of the destination frequencies for each edge-node is done by normalizing the count of cars. At the first junction 200 cars enter from the Wetstraat and 16 cars enter from the Trierstraat. This is a total of 216 cars which will enter the second junction from the Wetstraat. The counts of the second junction are multiplied with $216/200$, which will normalize the values. This calculation is done for all junctions, from start till end.

The calculation of the destination frequency of a specific edge-node $x$ to a possible destination edge-node $y$ is done as following:

1 search the edge-nodes $y$ which can be reached by the specific edge-node $x$

2 sum the cars that enter the possible edge-nodes $y$

3 the destination frequency from edge-node $x$ to $y$ is given by $\frac{sumOfCarsEnteringEdgenodeY}{sumOfCarsEnteringAllReachableEdgenodesFromX}$

| to | 0 | 4 | 8 | 10 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|
| 0 | x | x | x | x | x | x | x |
| 1 | 0.133 | 0.040 | 0.052 | 0.351 | 0.354 | 0.004 | 0.065 |
| 2 | 0.133 | 0.040 | 0.052 | 0.351 | 0.354 | 0.004 | 0.065 |
| 3 | 0.133 | 0.040 | 0.052 | 0.351 | 0.354 | 0.004 | 0.065 |
| 4 | x | x | x | x | x | x | x |
| 5 | 0.139 | x | 0.055 | 0.366 | 0.369 | 0.004 | 0.067 |
| 6 | 0.139 | x | 0.055 | 0.366 | 0.369 | 0.004 | 0.067 |
| 7 | 0.139 | x | 0.055 | 0.366 | 0.369 | 0.004 | 0.067 |
| 8 | x | x | x | x | x | x | x |
| 9 | 0.147 | x | x | 0.387 | 0.390 | 0.005 | 0.071 |
| 10 | x | x | x | x | x | x | x |
| 11 | 0.240 | x | x | x | 0.637 | 0.008 | 0.116 |
| 12 | x | x | x | x | x | x | x |
| 13 | 0.660 | x | x | x | x | x | 0.319 |
| 14 | x | x | x | x | x | x | x |
| 15 | 0.133 | 0.040 | 0.052 | 0.351 | 0.354 | 0.004 | 0.065 |
| 16 | 0.133 | 0.040 | 0.052 | 0.351 | 0.354 | 0.004 | 0.065 |

Table 5.6: Destination frequencies Wetstraat

The calculated destination frequencies for the Wetstraat are listed in table 5.6. Combinations of edge-nodes that are not possible are marked with $x$.

The spawn frequencies need to be calculated relative to the values in table 5.5. This calculation is done as following:

1 take the normalized counts of vehicles entering the Wetstraat from a side-road

2 select a time of day with counts of vehicles at the beginning of the Wetstraat

3 multiply with [1] with $\frac{countBeginningWetstraat}{normalizedCountSideRoad}$

4 dividing [3] by 3600 gives the spawn frequency of the specific edge-node at the specific time of day

The calculated spawn frequencies for all edge-nodes can be found in tables 5.7, 5.8 and 5.9. Each row represents the spawn frequencies for a certain edge-node for the different times of the day.

| time | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 1 | 0.011 | 0.006 | 0.003 | 0.003 | 0.004 | 0.013 | 0.065 | 0.117 |
| 2 | 0.006 | 0.003 | 0.002 | 0.002 | 0.002 | 0.008 | 0.040 | 0.071 |
| 3 | 0.007 | 0.004 | 0.002 | 0.002 | 0.003 | 0.009 | 0.046 | 0.083 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0.004 | 0.002 | 0.001 | 0.001 | 0.001 | 0.005 | 0.023 | 0.041 |
| 7 | 0.008 | 0.004 | 0.002 | 0.002 | 0.003 | 0.010 | 0.047 | 0.084 |
| 9 | 0.011 | 0.006 | 0.003 | 0.003 | 0.004 | 0.014 | 0.071 | 0.127 |
| 11 | 0.015 | 0.008 | 0.005 | 0.004 | 0.006 | 0.019 | 0.095 | 0.171 |
| 13 | 0.005 | 0.002 | 0.001 | 0.001 | 0.002 | 0.006 | 0.029 | 0.051 |
| 15+16 | 0.132 | 0.071 | 0.040 | 0.033 | 0.049 | 0.166 | 0.815 | 1.464 |

Table 5.7: Spawn frequencies Wetstraat - Part 1

| time | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|
| 1 | 0.092 | 0.090 | 0.079 | 0.075 | 0.069 | 0.085 | 0.085 | 0.074 |
| 2 | 0.056 | 0.054 | 0.048 | 0.045 | 0.042 | 0.052 | 0.052 | 0.045 |
| 3 | 0.065 | 0.063 | 0.056 | 0.053 | 0.049 | 0.060 | 0.060 | 0.052 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0.032 | 0.031 | 0.028 | 0.026 | 0.024 | 0.030 | 0.030 | 0.026 |
| 7 | 0.066 | 0.065 | 0.057 | 0.054 | 0.050 | 0.061 | 0.061 | 0.053 |
| 9 | 0.100 | 0.097 | 0.085 | 0.081 | 0.075 | 0.092 | 0.092 | 0.080 |
| 11 | 0.135 | 0.131 | 0.115 | 0.109 | 0.101 | 0.125 | 0.125 | 0.108 |
| 13 | 0.040 | 0.039 | 0.034 | 0.033 | 0.030 | 0.037 | 0.037 | 0.032 |
| 15+16 | 1.150 | 1.119 | 0.984 | 0.931 | 0.866 | 1.064 | 1.063 | 0.926 |

Table 5.8: Spawn frequencies Wetstraat - Part 2

| time | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
|---|---|---|---|---|---|---|---|---|
| 1 | 0.074 | 0.078 | 0.080 | 0.083 | 0.053 | 0.038 | 0.032 | 0.024 |
| 2 | 0.045 | 0.048 | 0.048 | 0.050 | 0.032 | 0.023 | 0.019 | 0.015 |
| 3 | 0.052 | 0.055 | 0.056 | 0.059 | 0.037 | 0.026 | 0.022 | 0.017 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0.026 | 0.028 | 0.028 | 0.029 | 0.019 | 0.013 | 0.011 | 0.008 |
| 7 | 0.053 | 0.056 | 0.057 | 0.060 | 0.038 | 0.027 | 0.023 | 0.017 |
| 9 | 0.080 | 0.085 | 0.086 | 0.090 | 0.057 | 0.041 | 0.034 | 0.026 |
| 11 | 0.108 | 0.114 | 0.116 | 0.121 | 0.078 | 0.055 | 0.046 | 0.035 |
| 13 | 0.032 | 0.034 | 0.035 | 0.036 | 0.023 | 0.016 | 0.014 | 0.011 |
| 15+16 | 0.922 | 0.978 | 0.995 | 1.037 | 0.663 | 0.469 | 0.394 | 0.301 |

Table 5.9: Spawn frequencies Wetstraat - Part 3

Tweekerkenstraat (edge-node 5) was closed for traffic when the vehicle count was done.
The traffic light controllers marching and optim are modified to a cycle of 90 seconds with 65 seconds green time on the wetstraat and 25 seconds green time on the crossing streets. In the previous simulations for scenarios 1 and 2, the cycle was divided into 4 periods, one for each direction. For the Wetstraat the cycle can be divided in 2 periods, one for the Wetstraat and one for the crossing streets. This is because the crossing streets are single direction streets. (Those controllers are implemented in $MorevtsMarchingWetstraatTLC$ and $MorevtsOptimWetstraatTLC$).

### 5.3.1 Best parameters

The parameters $\theta$ and $\varphi_{min}$ can be optimized for sotl-phase and sotl-platoon. The $\varphi_{min}$ is the minimal period of time a traffic light has to keep the green light. The results show that smaller values for $\varphi_{min}$ lead to better results (ATWT and waiting queue). Suppose that one car is waiting on a crossing lane of the Wetstraat. After a while he gets a green light for $\varphi_{min}$ seconds. On the Wetstraat a lot of cars are waiting and its threshold $\theta$ is already reached, but they have to wait until the $\varphi_{min}$ period is finished. This is why lower $\varphi_{min}$ values results in lower ATWT values.

The values of $\theta$ are also important to optimize the traffic flow. A higher $\theta$ will result in a longer waiting time on the crossing lanes of the Wetstraat, because

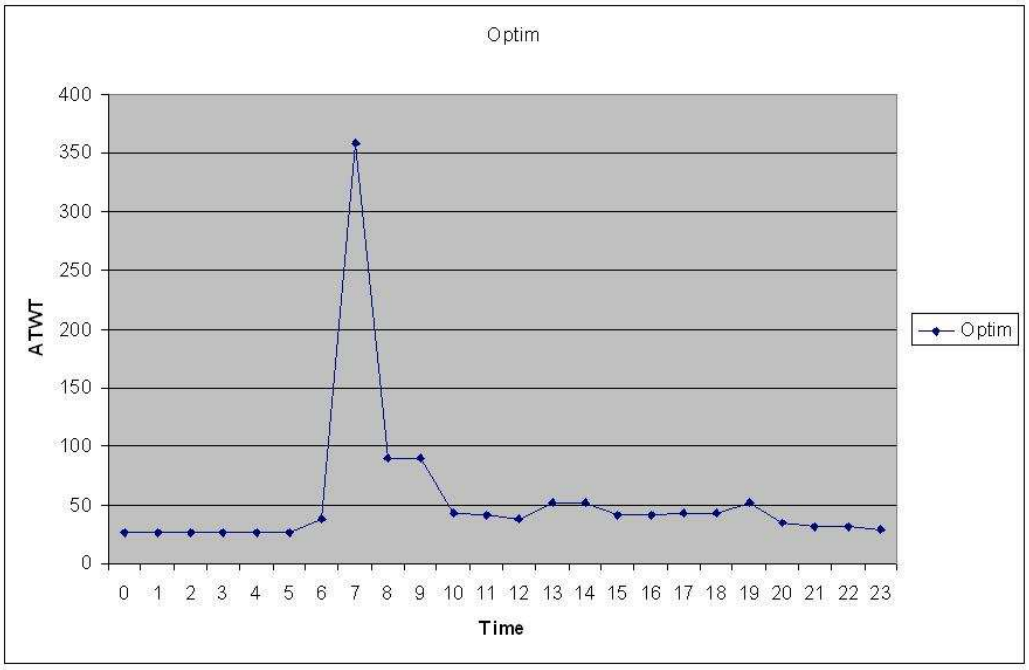


Figure 5.24: Wetstraat - Optim controller. At 7 am there is a high peak

there is not much traffic and the threshold is reached after a longer period of time. For a high traffic density a higher value of $\theta$ gives better results, but for low traffic density a lower $\theta$ is best.

The sotl-platoon controller can be changed to check the traffic density and adjust its own parameters depending on the current traffic density.

### The results

The results are shown in different graphs, one for each controller. The graphs show a relation between the time of day and the average value of ATWT at cycle 3600.

## 5.3.2 Optim controller

The optim controller has the same green time as in the Wetstraat. This is 65 seconds green and 25 seconds red. Because the green time is much longer than the red time, it can be seen in the simulations that the red light is moving on from

the beginning to the end of the Wetstraat. The traffic density is too high for this controller. The road users cannot keep the predefined speed, and this is why the aim of the green wave cannot be fulfilled.
For the highest traffic density at 7am a big queue of road users is formed at the beginning of the Wetstraat. Sometimes there is only one road user waiting at a red traffic light on a side-road. When it gets green, the light stays green for 25 seconds.
During the average traffic density at 6am the controller is much better. The flow of cars moving on the Wetstraat is divided by the red lights. The last road users getting green can keep up the speed and will get green at the next traffic light. The optim controller works successfully for this traffic density.
At 7am there are 5270 cars counted at the Wetstraat and at 6am there are only 2933 cars counted. This is about 55% of the traffic density at 7am.
For a low traffic density at midnight the cars only have to wait on the Wetstraat if the first traffic light is red, otherwise, the cars on the Wetstraat do not have to stop.

### 5.3.3 Request controller

The sotl-request controller works best for a high $\theta$. With a high $\theta$, the road users at the side-roads have to wait longer, and the priority goes to the road users on the Wetstraat (because the traffic density on the Wetstraat is much higher). The green time on the side-roads is only a fraction (about 10%) of the green time on the Wetstraat.

The results show that this controller can manage the traffic density at 7am. Almost all traffic lights at the Wetstraat are green and sometimes one or two traffic lights turn red on the Wetstraat. This gives a very good result.

At lower traffic densities, the red lights on the Wetstraat stay red for a longer period of time. This is because the traffic light turns green when it is requested by waiting road users in front of those traffic lights. In this case, the road users on the Wetstraat have to stop more often and for a longer period of time as the $\theta$ becomes bigger. For lower traffic densities, a lower $\theta$ gets better results, because the roadusers have to wait for a shorter period of time to get a green light. With a very low $\theta$ ($\theta = 5$) the road users will get green almost immediately, but this

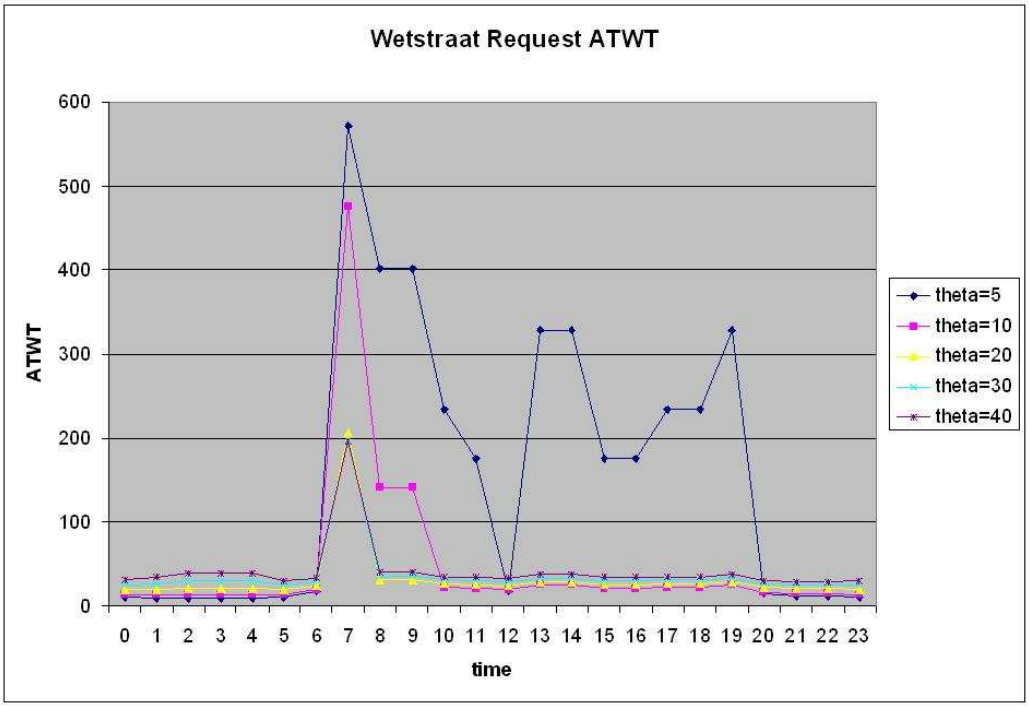


Figure 5.25: Wetstraat Request controller ATWT. Different values of $\theta$ have different results. $\theta = 5$ has the worst results.

is only useful for a low traffic density. When a low $\theta$ is used for a high traffic density, the traffic lights would switch too fast (almost every second). This will give poor results. More green time is needed at the Wetstraat and this can be done by a higher $\theta$. A higher $\theta$ will make it more difficult for the side-roads to get a green light.

### 5.3.4 Phase Controller

For the higher traffic densities it seems that small values for $\theta$ give unwanted results. A very small $\theta$ will grant an green-request very fast. In combination with a small value for $\varphi_{min}$, the traffic lights will switch very fast. This problem is solved for values of $\theta \geq 20$. The phase controller with $\theta = 20$ and $\varphi_{min} = 1$ gives the wanted behaviour: short green times for the side-roads and longer green times for the Wetstraat.

A higher $\varphi_{min}$ will result in a very high ATWT value. The reason is that the side-roads get much longer green time than needed. This will stop the traffic flow on the Wetstraat for too long, which will result in waiting queues for the Wetstraat.

Not only $\varphi_{min}$ specifies the green time, but also $\theta$ has an influence. Higher values for $\theta$ will cause a longer waiting time before granting a request.

At low traffic density there is no problem for a low value of $\theta$. From 20pm till 5am the best results are given for low values of $\theta$ and $\varphi_{min}$.

At the highest traffic density (7am), a very high $\theta$ is needed. There are passing 5270 vehicles on the Wetstraat. The flow on the Wetstraat is a priority and the green times for the Wetstraat should be as long as possible, without creating long queues on the side-roads.

### 5.3.5 Platoon controller

The platoon controller works best for a low minimum green time ($\varphi_{min}$) because the traffic light stays green when a platoon is crossing the traffic light. The density of a platoon can be defined with the parameters $\mu$ and $\omega$.

A low minimal green time ($\varphi_{min}$) with higher values of $\theta$ has a special consequence, because the controller does not break platoons. When a queue of road

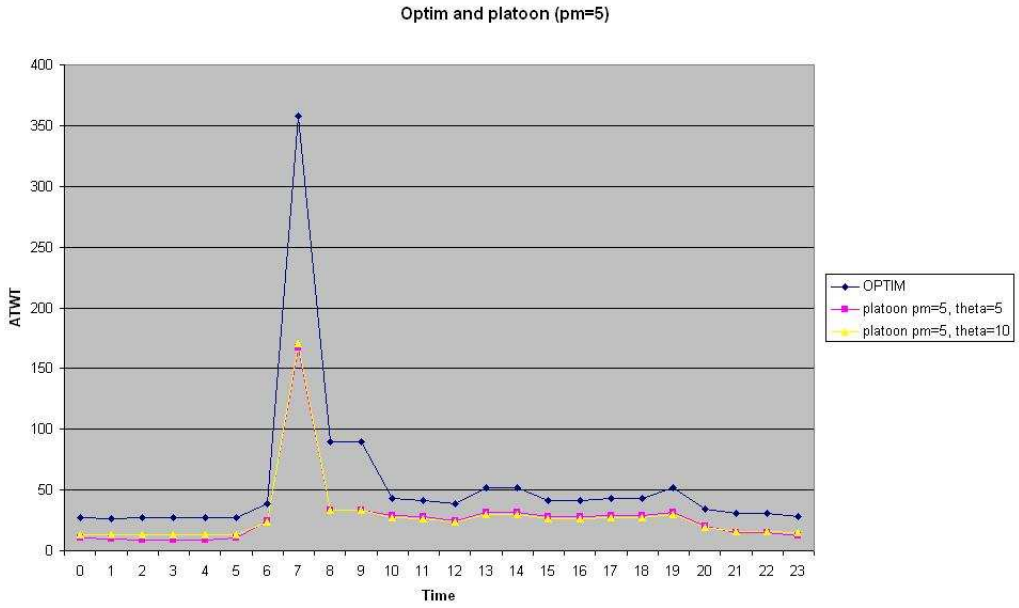


Figure 5.26: Wetstraat, optim controller and platoon controller with $\varphi_{min} = 5$ and $\theta = 5; 10$

users is waiting in front of a red light, all the road users can cross the junction when it gets green and after the last road user has crossed the junction, the traffic light at the other direction can request green, which it gets immediately when cars are approaching the traffic light. In this way the traffic light controller keeps platoons together. However, when $\theta$ is set to 5 and $\varphi_{min}$ to 1, then the second condition of the controller will cause problems. Requests for green light are allowed very fast when those parameters are used. When the cars get green, they have not enough time to get away to reduce the density of cars in front of the traffic lights. When the controller counts more than $\mu$ cars, other requests are granted. In this way the light switches very fast and the traffic cannot flow through fast enough.

For high density (Wetstraat at 7am) the Wetstraat has all its traffic lights turned green for most of the time. On the side-roads few cars are approaching the red light and a queue of road users is growing slow. When the $\theta$ has been reached and there is no platoon crossing, the side-road gets green and all the waiting cars are entering the Wetstraat. After the last car has crossed the junction, the Wetstraat gets green again. Because of the high traffic density, platoons are not visible.

For lower traffic density, platoons are much more visible. On the side-roads, only a few cars are waiting in front of a red traffic light. Those cars get green faster, because there is less traffic on the Wetstraat. When the last car has past the junction, the traffic light of the side-road stays green, until the cars on the Wetstraat are approaching the traffic light and the conditions are met. This is how platoons are formed. When a single car gets to a red light, it has to wait until the $\theta$ has been reached. More cars are joining and the condition is reached faster. When the traffic light turns to green, a group of cars can move together.

For the lowest traffic densities, a lower minimal green time ($\varphi_{min}$) and a lower $\theta$ is needed, otherwise cars have to wait in front of a traffic light, when there is no traffic on the side-roads.

### 5.3.6 Best parameters for Platoon at highest traffic density.

For higher values of $\theta$ better results are given for lower $\varphi_{min}$ values, but when $\varphi_{min} = 1$ for a lower values of $\theta$, the traffic lights do not show the expected behaviour. Some scenarios are given below. The ATWT values for the Wetstraat at 7am for the different parameters can be found in table 5.10.

- If the destination road behind the junction is full, the traffic lights switch at maximum speed, depending on the $\varphi_{min}$ value. This is because when the destination road is full, the cars that have to cross the junction will slow down and the density of cars will rise. The second condition of the platoon controller will not be satisfied and the request of the other road will be accepted, but this road will have the same problem. This will cause the switching at a speed depending on $\varphi_{min}$. The problem is solved when the road after the junction is creating some open space.

- If $\varphi_{min} = 1$ and the threshold value $\theta$ can be reached within a few seconds, it can happen that the traffic lights switch very fast and this will interleave the traffic of the Wetstraat and its side-road. This will increase the queue of cars in the side-roads, but the Wetstraat will decrease its waiting queue. Because of this, the platoon controller with $\theta = 5$ and $\varphi_{min} = 1$ has a very low average trip waiting time. The low ATWT is very nice, but the behaviour is

| 7 am | 1 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|
| 5 | **100.78** | **176.36** | 227.51 | 315.94 | 431.76 |
| 10 | **331.23** | **170.70** | 209.57 | 328.41 | 447.78 |
| 20 | **171.06** | **159.35** | 220.92 | 317.50 | 455.86 |
| 30 | **174.27** | 186.60 | 201.32 | 313.75 | 434.55 |
| 40 | 163.65 | **158.75** | 218.66 | 298.10 | 395.19 |
| 50 | 185.92 | **174.17** | 212.77 | 270.32 | 382.63 |
| 60 | 194.06 | **176.39** | 201.84 | 266.33 | 380.58 |
| 70 | **168.73** | 189.32 | 206.75 | 257.62 | 379.14 |
| 80 | **196.74** | 210.29 | 188.80 | 258.15 | 375.52 |
| 90 | 210.41 | **199.20** | 208.46 | 256.69 | 401.59 |
| 100 | 219.50 | **205.07** | 208.69 | 248.20 | 338.72 |
| 110 | 223.47 | **221.07** | 223.38 | 276.40 | 349.14 |

Table 5.10: Wetstraat 7am, platoon controller: ATWT values for different values of $\theta$ and $\varphi_{min}$. Rows represent different values for $\theta$, columns represent different values for $\varphi_{min}$.

not always predictable. With $\varphi_{min} = 5$ the traffic light controller works as expected but will have a higher ATWT which is still good.

- If $\theta = 10$ and $\varphi_{min} = 1$ the first junction of the Wetstraat will interleave the traffic of the Wetstraat and its side-road. This causes a very high average trip waiting time and a large waiting queue at the beginning of the Wetstraat. Behind the first junction the traffic density is much lower and the next junctions have no problems. If $\varphi_{min} = 5$ for the same $\theta$ this problem is solved.

If those ATWT values are compared with the results for the optim control method (ATWT=358.11), it shows that all values for $\varphi_{min} = 5 or 10$ are smaller than for optim. The lowest value is only 44.32% of the ATWT value of optim. The highest value is 63.53% of the optim ATWT value. The platoon controller with $\theta = 5$ and $\varphi_{min} = 1$ has the lowest ATWT (28.14% of optim ATWT), but it has some unexpected side-effects, which are not acceptable.

If the best ATWT has to be selected without side-effects it would be around 170, which is a reduction in ATWT of 52%.

### 5.3.7 Platoon as good as request

For the lower traffic densities (map 0, 1, 2, 5, 6, 20, 21, 23) the best results are given for $\theta = 5$. For the higher traffic densities, higher values of $\theta$ are better. This is because a higher $\theta$ will give priority to the traffic on the Wetstraat. The traffic coming from the side-roads has to wait longer to get green compared with the traffic coming from the Wetstraat. This implies that the green time on the Wetstraat is longer than the green time on the side-roads, and the desired behaviour of traffic lights is reached.

For the lower traffic densities (map 0, 1, 2, 5, 6, 20, 21, 23), the platoon controller with minimal green time set to 1 second has the same results as the request controller. For higher traffic densities, higher values for $\theta$ are better and the results are again similar.

### 5.3.8 Optim (green wave) versus platoon

The green wave algorithm is a good algorithm which optimizes the traffic flow on the Wetstraat. This gives good results, but the platoon controller has better results for all traffic densities. For the traffic densities at 0, 1, 2, 5, 6, 20, 21, 23 hour, platoon reduces the average total waiting time with 59-67%.

The simulation results show that the optim controller has a queue of 1186 road users at the beginning of the Wetstraat at 7 o'clock (highest traffic density). For the best result of the platoon controller ($\theta = 70$, $\varphi_{min} = 1$) there is a queue of 267 road users. This is a reduction of the queue with 77%.

### 5.3.9 Traffic density versus $\theta$, growing as a monotonic function?

For the platoon controller with $\varphi_{min}$ set to 1, the best results are obtained with $\theta = 5$. With the higher traffic density, the best ATWT is for $\theta = 5$, but this gives unexpected side-effects and is not acceptable. This strange behaviour is also shown in figure 5.28 on the next page. This figure does not show a gradual transition in $\theta$ for the best results. It is possible that this gradual transition occurs for a density between 4141 and 5270. This can be exploited in further research.

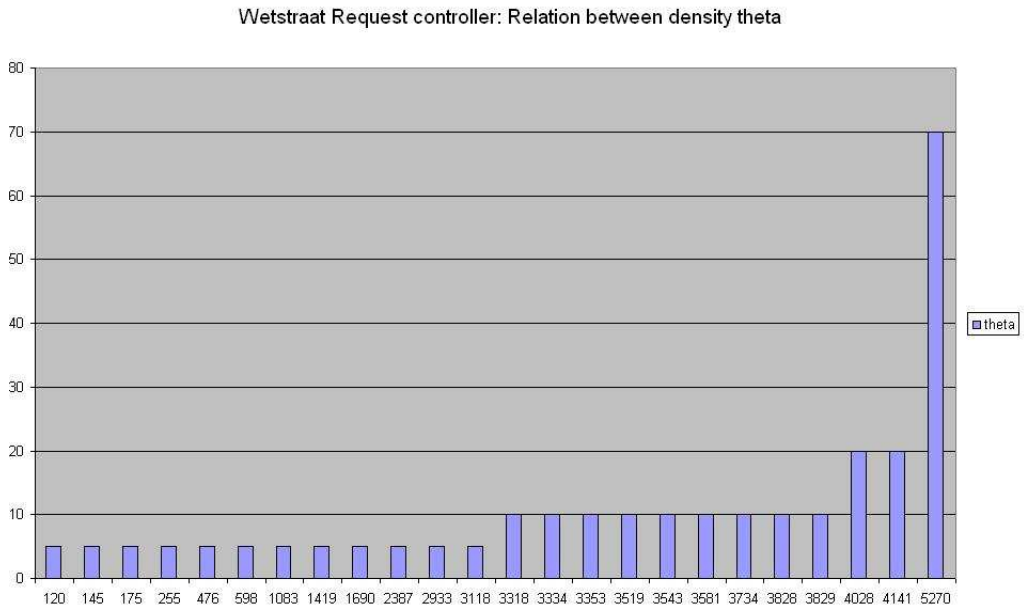


Figure 5.27: This graph shows the best values of $\theta$ in function of the density in the Wetstraat with the request controller.

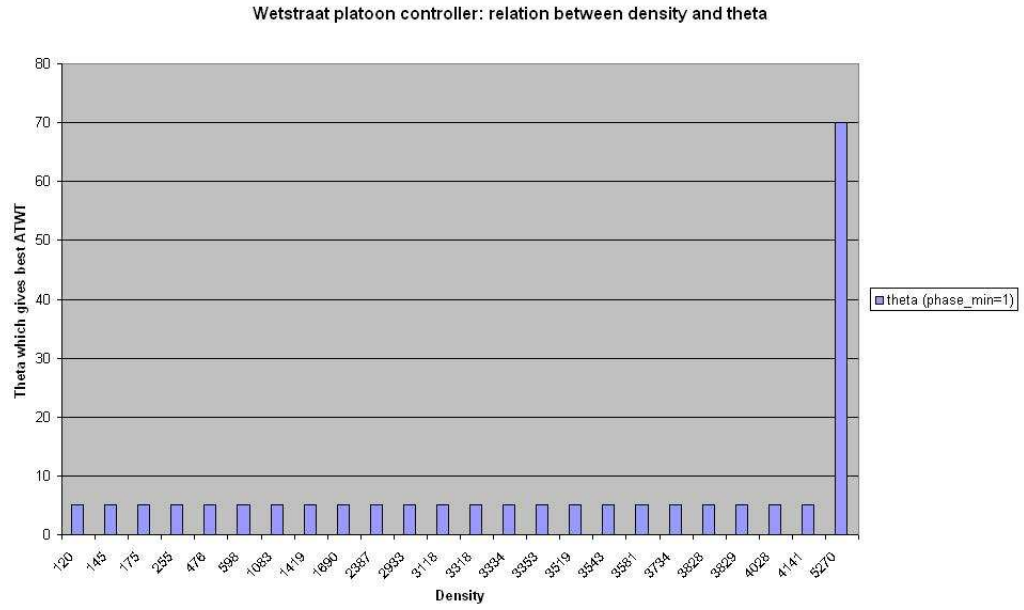


Figure 5.28: This graph shows the best values of $\theta$ in function of the density in the Wetstraat with $\varphi_{min}$ set to 1. Platoon controller is used.

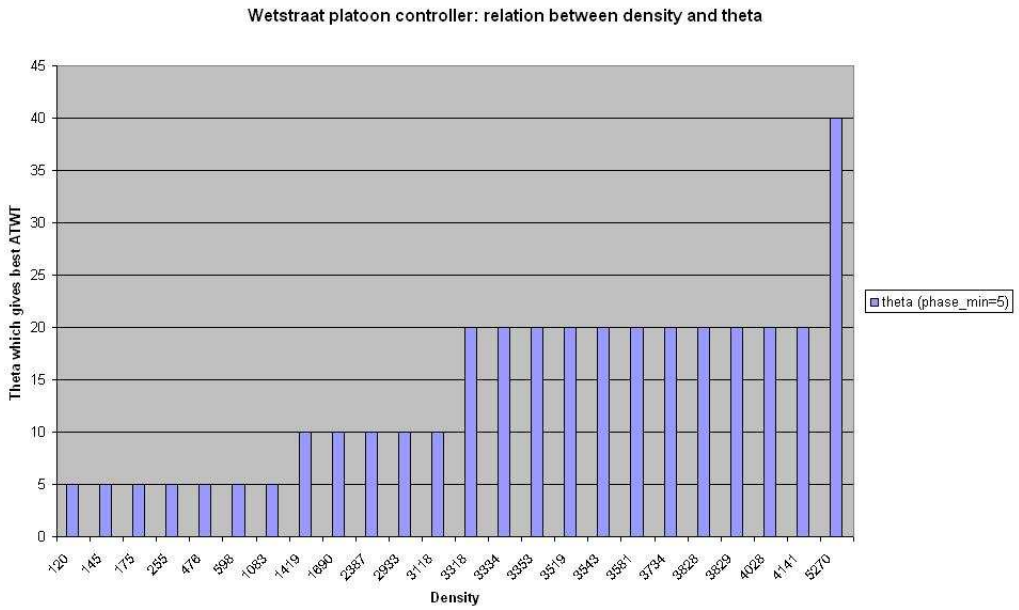


Figure 5.29: This graph shows the best values of $\theta$ in function of the density in the Wetstraat with $\varphi_{min}$ set to 5. Platoon controller is used.

The graph for $\varphi_{min} = 5$ shows us that a higher traffic density needs a higher $\theta$. With a higher $phases\_min$ value, the graph will grow more.

The sensors of a sotl-method could be used to decide in real time which $\theta$ to use and it seems that $\varphi_{min} = 5$ does not need to be changed.

# Chapter 6

# Conclusion

In this chapter all results are given. The first goal was to build a realistic traffic simulator and the second goal was to demonstrate that the sotl-platoon controller is much better than a green wave controller. The last goal was to convince the *Ministry of the Brussels-Capital region* that the sotl-platoon controller is better than the current controller in the Wetstraat of Brussels. The first goal is captured in the next section. The second section gives the results of the simulations, which cover the second and third goal.

## 6.1 Traffic simulator

The traffic simulator used for simulating the traffic flow with five traffic light controllers for different scenarios, was created starting from an existing traffic simulator (GLD / iAtracos).

- The model has been changed to represent the speed in $m/s$, which makes it easier to model realistic scenarios.
- Road users will now accelerate and decelerate when needed. In the previous simulator the road users moved at a constant speed, or were not moving. Another improvement is that the road users are respecting a stop-distance to avoid accidents.

- For the green wave controller it was needed to specify which roads have priority to others. The big traffic flows can be set with this property.
- When traffic lights change, there has to be a period of time where all lights stay red. Road users on the junction can leave the junction during this period.
- Five traffic light controllers are implemented: 3 self-organizing controllers and 2 non-self-organizing controllers.

## 6.2 Self-organizing traffic light controllers

The proposed self-organizing traffic light controllers of Carlos Gershenson have been tested on 2 imaginary scenarios and on one realistic scenario: Wetstraat of Brussels. Those controllers are sotl-request, sotl-phase and sotl-platoon. Two non self-organizing traffic light controllers, marching and optim controller, have been implemented and tested on the same scenarios.

The results of the simulations are given in ATWT and TWQL. ATWT stands for Average Trip Waiting Time and TWQL stands for Total Waiting Queue Length.

### 6.2.1 Scenario 1

The first scenario is an infrastructure of 2 junctions, 6 edge-nodes and 7 roads (two directions). Two different traffic densities are used for the simulation: high density (HD) and low density (LD). The traffic flow in this scenario is symmetric for opposing edge-nodes.

| controller | density | ATWT | TWQL |
|---|---|---|---|
| marching | HD | 264 | 1434 |
| optim | HD | 235 | 1790 |
| request | HD | 344 | 1457 |
| phase | HD | 89 | 144 |
| platoon | HD | 89 | 155 |
| marching | LD | 175 | 428 |
| optim | LD | 216 | 481 |
| request | LD | 31 | 0 |
| phase | LD | 38 | 0 |
| platoon | LD | 18 | 0 |

Table 6.1: Scenario 1: Best results

From table 6.1 can be concluded that in almost all cases the self-organizing traffic lights are better than the marching and optim controller. The sotl-request controller for high traffic density is worse. This is so because at high traffic density requests are granted almost immediately. This results in the very fast switching of traffic lights. The value of $\theta$ can regulate the green times of the traffic lights. This is why the highest value for $\theta$ gives the best ATWT value (Average Trip Waiting Time).

The sotl-phase controller is better than the optim and marching controller for both traffic densities. Compared to the optim controller, the sotl-phase controller at high traffic density reduces the ATWT with 62% and the TWQL (Total waiting queue length) with 91%. At low traffic density, this is a reduction of 82% for the value of ATWT and an elimination of the TWQL value.

The sotl-platoon controller has very good results. Compared to the optim controller, the sotl-platoon controller has reduced the ATWT with 62% and the TWQL with 90% at high traffic density. At low traffic density, the sotl-platoon controller reduces the ATWT with 90% compared to the optim controler and the TWQL is eliminated.

### 6.2.2 Scenario 2

Scenario 2 is an infrastructure with five junctions, 12 edge-nodes, and 16 roads in two directions. There is a bigger traffic flow from edge-node 0 to edge-node 11

for both traffic densities.

| controller | density | ATWT | TWQL |
|---|---|---|---|
| marching | HD | 304 | 2346 |
| optim | HD | 406 | 2467 |
| request | HD | 234 | 1294 |
| phase | HD | 242 | 1164 |
| platoon | HD | x | x |
| marching | LD | 282 | 1128 |
| optim | LD | 332 | 1040 |
| request | LD | 45 | 6 |
| phase | LD | 116 | 101 |
| platoon | LD | 24 | 0 |

Table 6.2: Scenario 2: Best results

The request, phase and platoon controller are better than the optim controller.

The request controller at high traffic density reduces the ATWT with 51% and the TWQL with 47% compared with the optim controller. For low traffic density there is a reduction of 86% for ATWT and 99% for TWQL.

For the phase controller the reduction is 50% for ATWT and 52% for TWQL, at high traffic density. At low traffic density, the controller reduces the ATWT with 65% and the TWQL with 90%.

For the platoon controller a problem is discovered at high traffic density. When all roads are congested, and the conditions for *platoon is crossing* are satisfied, almost all traffic is blocked. This problem can be solved with an extra rule for the controller (one of the following):

- the case *platoon is crossing* is stopped when a maximum time has reached.
- when a possible destination lane is full, the case *platoon is crossing* is stopped when a maximum time has reached.

The first rule can cut platoons when no problem occurs, the second rule need sensors at all inputs and outputs of a junction but will not cut a platoon when no problem occurs. It is also possible to give a bigger value to busses or other longer vehicles when the condition *platoon is crossing* is checked.

The platoon controller for low traffic density reduces the ATWT with 92% and eliminates the waiting queues.

### 6.2.3 Scenario 3: Wetstraat

The simulations of the Wetstraat are based on the real traffic flow in the Wetstraat. The spawn frequencies at the beginning of the Wetstraat are counted by the *Ministry of the Brussels-Capital region*. The destination of cars at every junction are approached by counting the destination of cars coming from every direction at every junction. The spawn and destination frequencies of all edge-nodes are calculated based on those counts.

The self-organizing traffic light controllers are simulated for every traffic density. The marching and optim controller are modified to approach the green times of the traffic lights in the Wetstraat.

The performance of the self-organizing traffic lights depends on the decision variables of the algorithms. Different values are tested for those decision variables and the values which perform best have been selected. Those values depend on the topology of the infrastructure and on the traffic density at each junction. The self-organizing traffic light controllers can be modified to detect the traffic density and select the best values for those decision variables. The simulation results show that the traffic density is growing monotonicly with the value of $\theta$ to obtain the best ATWT results. Further simulations for those modified traffic light controllers are needed.

The simulation results are given for fixed decision variables and are compared with the optim controller for all traffic densities.

The simulations for the Wetstraat give similar results as obtained in the previous scenarios. The green wave algorithm is a good algorithm which optimizes the traffic flow on the Wetstraat. This gives good results, but the platoon controller has better results for all traffic densities. For the traffic densities at 0, 1, 2, 5, 6, 20, 21, 23 hour, platoon reduces the average total waiting time with 59-67%.

The simulation results show that the optim controller has a queue of 1186 road

users at the beginning of the Wetstraat at 7 o'clock (highest traffic density). For the best result of the platoon controller ($\theta = 70$, $\varphi_{min} = 1$) there is a queue of 267 road users. This is a reduction of the queue with 77%.

## 6.3 Future work

I added a clearance time for road users to leave a junction when a light is switching. The simulator should be extended with a yellow light, but this will affect the working of the simulator.

The simulation can be extended with unexpected events, like cars ignoring traffic lights and traffic rules.

**Traffic simulator** The traffic simulator has been extended with extra physics, driving behaviour and traffic rules. These aditions were needed to approach real traffic behaviour. More features can be implemented to the simulation model, to get a more realistic approach.

- Vehicles should be able to switch lanes on a road where it is allowed. This needs markings on the road to notify the road users. When there is no vehicle in front, the road user should take the lane at the right side of the road. When a road user wants to pass a road user in front of him, he should move to the lane left of him, pass the road user and move back to the previous lane. At a junction a road user needs to select a lane which collects vehicles with the same direction. In the current simulation this is done at the beginning of the road. The destination of the road user is looked up and the road user will choose a lane from where it can reach its destination.
- Traffic lights have orange lights to warn the road users that the light is going red. Road users which cannot stop at time, can drive through the orange light but all road users should stop for a red light.

**Simulations** The traffic light controllers are simulated in three scenarios. Those simulations gave good results, but more extensive simulations could be done and special cases could be studied.

- What is the influence of adding or removing traffic lights on the traffic density and waiting times of road users?
- A lot of things can happen which disturb the traffic flow. There can be work in progress, a taxi can wait for a client, an accident can happen, etc. What is the influence of blocking lanes on the Wetstraat?
- The destination frequencies in the Wetstraat are calculated on data from one counting at each junction. This gives an idea of the destination frequencies at one point in time. Studying the destination frequencies from every starting point is very difficult to do and will take a lot of time.
- All simulations for the Wetstraat are done with only cars and no other type of vehicles. This is because there was no data about how many cars and how many longer vehicles are driving on the Wetstraat. When simulations are needed with different road user types, all countings need to be done again with distinction between the different types of vehicles.

# Appendix A

# Source Code

In this chapter are some parts of code given which are discussed in chapter 4 (Coding process). The complete source code is available at the sourceforge site for the project $moreVTS$ [mor].

## A.1 control on switching lights

For the proposed traffic light controllers it was needed to have control of when traffic lights should switch. The use of a few flags will make it able to do this. $keepSwitchControl$ is used to specify which traffic light controllers need to control the switching itself. $keepTLDFlag$ is used to fixate a specific traffic light configuration.

```
if ( tlcontroller .getKeepSwitchControl()) {
    if (!( node.getKeepTLDDFlag()))
        switchTrafficLights (( Junction )node, decisions [ i ]);
    else
        switchTrafficLights (( Jucntion )node, decisions [ i ]);
}
```

## A.2 countRoadUsers for the self organizing traffic ligths

For updating $kappa$ in the proposed self-organizing traffic light controllers it is needed to count the vehicles of a specific lane.

```
public int countRoadusers(Drivelane lane, int range) {
    int cntr = 0;
    Roaduser ru;
    boolean stop = false;
    LinkedList queue = lane.getQueue();
     ListIterator li = queue. listIterator ();
    while (li.hasNext() && !stop) {
        try
        {
            ru = (Roaduser) li.next();
        }
        catch (Exception e)
        {
            // When this exception is thrown you removed the first element
            // of the queue, therefore re-create the iterator .
            System.out. println ("CME");
            li = queue. listIterator ();
            continue;
        }
        if (ru. getPosition () <= range)
            cntr++;
        else
            stop = true;
    }
    return cntr;
}
```

## A.3 MorevtsSotlPhase

$updateTLDs$ and $decideTLs$ of the Sotl-phase traffic light controller is given below.

```
public void updateTLDs() {
    for (int i = 0; i < tld.length; i++) {
        boolean switched = false;
        for (int j = 0; j < tld[i].length; j++)
        if (tld[i][j].getTL().getCycleSwitched()==getCurCycle()-1)
            switched = true;

        if (switched) {
            if (! tld[i][0].getTL().getNode().getKeepTLDFlag())
            {
                tld[i][0].getTL().getNode().setKeepTLDFlag(true);
                tld[i][0].getTL().getNode().setPhaseMinimal(PHASE_MIN);
                for (int j= 0; j < tld[i].length; j++)
                    if (tld[i][j].getTL().getState())
                        tld[i][j].setKappa(0);
            }
        }
    }
}

public TLDecision[][] decideTLs() {
    TLDecision currentDec;
    Drivelane curLane;

    // adjust data after last cycle
    updateTLDs();

    for (int i = 0; i < tld.length; i++) { // for all nodes
        Node currentNode = null;
        for (int j = 0; j < tld[i].length; j++)
            if (currentNode == null) currentNode= tld[i][j].getTL().getNode();
```

```
        if (currentNode != null && currentNode.getKeepTLDFlag()) {
            currentNode.decrPhaseMinimal();
        }

        for (int j = 0; j < tld[i].length; j++) { // for all inbound lanes in node
            currentDec = tld[i][j];
            curLane = tld[i][j].getTL().getLane();
            int cntr = countRoadusers(curLane);

            if (!tld[i][j].getTL(). getState ())
                currentDec.addKappa(cntr);

            if (currentNode.getPhaseMinimal() <= 0 && currentDec.getKappa() >= TETA) {
                currentDec.setQValue(currentDec.getKappa());
                currentNode.setKeepTLDFlag(false);
            } else {
                currentDec.setQValue(0);
            }
        }
    }
    return tld;
}
```

## A.4 MorevtsSotlPlatoon

$updateTLDs$ and $decideTLs$ of the Sotl-platoon traffic light controller is given below.

```
public void updateTLDs() {
    for (int i = 0; i < tld.length; i++) { // for all nodes
        boolean switched = false;
        boolean platoonCrossing = false;
        int platoonRoaduserCount = 0;
        int nrOfPlatoonLanes = 0;
```

```
        int platoonRuCntAvg = 0;

        for (int j = 0; j < tld[i].length; j++)
            if (tld[i][j].getTL().getCycleSwitched()==getCurCycle()-1)
                switched = true;

        // set flag and reset kappa
        if(switched) {
            if (!tld[i][0].getTL().getNode().getKeepTLDFlag()) {
                tld[i][0].getTL().getNode().setKeepTLDFlag(true);
                tld[i][0].getTL().getNode().setPhaseMinimal(PHASE_MIN);
                for (int j = 0; j < tld[i].length; j++)
                    if (tld[i][j].getTL(). getState ())
                        tld[i][j].setKappa(0);
            }
        }

        // check if platoon is crossing.
        for (int j = 0; j < tld[i].length; j++)
            // if tl is green
            if (tld[i][j].getTL(). getState ()) {
                platoonRoaduserCount += countRoadusers(tld[i][j].getTL().getLane(), OMEGA);
                nrOfPlatoonLanes += 1;
            }
        if (nrOfPlatoonLanes>0) {
            platoonRuCntAvg = platoonRoaduserCount/nrOfPlatoonLanes;
            if (platoonRuCntAvg >= 1 && platoonRuCntAvg <= MU){
                    platoonCrossing = true;
            }
            if(tld[i].length>=1)
                tld[i][0].getTL().getNode(). setPlatoonCrossing(platoonCrossing);
        }
    }
}
```

```
public TLDecision[][] decideTLs() {
    TLDecision currentDec;
    Drivelane curLane;

    // System.out. println ("#### TLDATA-"+this.getCurCycle()+" ####");

    // adjust data after last cycle
    updateTLDs();

    for (int i = 0; i < tld.length; i++) { // for all nodes
        Node currentNode = null;
        for (int j = 0; j < tld[i].length; j++)
            if (currentNode == null) currentNode= tld[i][j].getTL().getNode();

        if (currentNode != null && currentNode.getKeepTLDFlag()) {
            currentNode.decrPhaseMinimal();
        }

        for (int j = 0; j < tld[i].length; j++) { // for all inbound lanes in node

            currentDec = tld[i][j];
            curLane = tld[i][j].getTL().getLane();
            int cntr = countRoadusers(curLane,VISIBLE);

            if (! tld[i][j].getTL(). getState ())
                currentDec.addKappa(cntr);

            if (currentNode.getPhaseMinimal() <= 0
                    && !currentNode.isPlatoonCrossing ()
                    && currentDec.getKappa() >= TETA) {
                currentDec.setQValue(currentDec.getKappa());
                currentNode.setKeepTLDFlag(false);
                // curDec.setQValue(1);
            }
            else
```

```
            {
                currentDec.setQValue(0);
            }
        }
    }
    return tld;
}
```

# Appendix B

# Simulation data

## B.1 Wetstraat Marching and Optim

| time | Optim ATWT | TWQL |
|---|---|---|
| 0 | 26.97 | 0 |
| 1 | 26.53 | 0 |
| 2 | 27.05 | 0 |
| 5 | 26.93 | 0 |
| 6 | 38.31 | 0 |
| 7 | 358.11 | 1186 |
| 8 | 90.02 | 15 |
| 10 | 42.67 | 0 |
| 11 | 41.59 | 0 |
| 13 | 52.29 | 4 |
| 20 | 34.59 | 0 |
| 21 | 30.92 | 0 |
| 23 | 28.56 | 0 |

Table B.1: Wetstraat Optim controller: ATWT and TWQL.

## B.2 Wetstraat Request

| time | theta5 | theta10 | theta20 | theta30 | theta40 |
|---|---|---|---|---|---|
| 0 | 9.83 | 13.08 | 19.96 | 25.27 | 31.13 |
| 1 | 9.35 | 12.79 | 20.14 | 27.23 | 34.15 |
| 2 | 8.86 | 12.98 | 20.85 | 30.55 | 38.58 |
| 5 | 9.78 | 12.98 | 19.23 | 25.10 | 30.16 |
| 6 | 18.02 | 19.52 | 23.69 | 28.32 | 32.26 |
| 7 | 570.87 | 475.15 | 206.72 | 198.32 | 196.89 |
| 8 | 401.72 | 141.67 | 32.24 | 35.95 | 39.86 |
| 10 | 234.41 | 22.73 | 26.54 | 31.11 | 34.56 |
| 11 | 175.13 | 21.64 | 25.60 | 30.16 | 34.00 |
| 13 | 328.99 | 26.18 | 29.10 | 32.93 | 37.09 |
| 20 | 14.26 | 16.83 | 21.87 | 26.38 | 30.46 |
| 21 | 11.96 | 14.92 | 20.34 | 25.08 | 28.80 |
| 23 | 10.96 | 13.93 | 19.75 | 25.22 | 29.33 |

Table B.2: Wetstraat Request ATWT

| 00:00 | 1 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|
| 5 | 9.84 | 10.45 | 13.11 | 16.84 | 20.38 |
| 10 | 12.99 | 13.39 | 14.59 | 17.96 | 22.12 |
| 20 | 19.64 | 19.97 | 20.35 | 22.28 | 24.75 |
| 30 | 25.73 | 26.03 | 25.89 | 26.55 | 28.48 |
| 40 | 31.84 | 31.80 | 31.59 | 32.22 | 32.97 |

Table B.3: Wetstraat phase [00:00]

| 01:00 | 1 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|
| 5 | 9.09 | 9.71 | 11.27 | 13.67 | 16.24 |
| 10 | 12.77 | 13.02 | 13.58 | 15.45 | 18.48 |
| 20 | 20.02 | 20.33 | 20.52 | 21.21 | 21.78 |
| 30 | 27.40 | 27.91 | 27.85 | 27.53 | 29.71 |
| 40 | 33.80 | 34.94 | 33.89 | 35.87 | 35.63 |

Table B.4: Wetstraat phase [01:00]

## B.3 Wetstraat Phase

The results for 7am are not listed. The results were wrong and the simulation has to be done again.

| 02:00 | 1 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|
| 5 | 8.86 | 9.18 | 10.49 | 12.16 | 14.21 |
| 10 | 12.94 | 12.92 | 13.69 | 14.88 | 14.41 |
| 20 | 21.39 | 21.35 | 21.48 | 21.93 | 23.16 |
| 30 | 28.42 | 29.43 | 30.59 | 29.45 | 31.12 |
| 40 | 36.46 | 38.16 | 36.88 | 37.95 | 38.87 |

Table B.5: Wetstraat phase [02:00 | 03:00 | 04:00]

| 05:00 | 1 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|
| 5 | 9.87 | 10.62 | 13.67 | 17.93 | 23.98 |
| 10 | 13.11 | 13.10 | 15.08 | 18.47 | 23.67 |
| 20 | 18.95 | 19.72 | 19.79 | 21.96 | 26.23 |
| 30 | 25.34 | 24.85 | 25.60 | 26.58 | 28.98 |
| 40 | 30.30 | 30.69 | 30.39 | 31.34 | 31.97 |

Table B.6: Wetstraat phase [05:00]

| 06:00 | 1 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|
| 20 | 57.39 | 59.85 | 60.11 | 58.84 | 58.62 |
| 30 | 51.68 | 51.16 | 52.93 | 51.81 | 52.88 |
| 40 | 49.27 | 48.79 | 48.95 | 49.40 | 49.34 |
| 50 | 47.75 | 48.109 | 47.62 | 47.09 | 47.57 |
| 60 | 48.19 | 47.46 | 47.41 | 47.59 | 47.10 |
| 70 | 48.36 | 49.21 | 48.47 | 47.67 | 48.30 |

Table B.7: Wetstraat phase [06:00 | 12:00]

| 08:00 | 1 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|
| 50 | 42.87 | 43.76 | 67.28 | 208.60 | 339.35 |
| 60 | 46.54 | 46.59 | 58.33 | 181.50 | 312.31 |
| 70 | 50.76 | 50.98 | 56.03 | 129.45 | 283.24 |
| 80 | 53.83 | 54.10 | 56.94 | 130.54 | 285.08 |
| 90 | 57.81 | 56.81 | 60.02 | 92.67 | 263.52 |
| 100 | 60.73 | 61.09 | 63.14 | 97.73 | 232.14 |

Table B.8: Wetstraat phase [08:00 | 09:00]

| 10:00 | 1 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|
| 40 | 35.08 | 35.03 | 43.69 | 71.36 | 171.41 |
| 50 | 39.00 | 38.53 | 43.12 | 60.71 | 135.72 |
| 60 | 41.65 | 41.91 | 44.04 | 59.10 | 92.89 |
| 70 | 45.24 | 45.35 | 47.13 | 57.33 | 86.08 |
| 80 | 48.66 | 47.68 | 49.72 | 57.94 | 79.07 |
| 90 | 51.59 | 51.48 | 52.04 | 58.72 | 76.79 |

Table B.9: Wetstraat phase [10:00 | 17:00 | 18:00]

| 11:00 | 1 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|
| 40 | 33.96 | 34.08 | 42.14 | 62.01 | 127.22 |
| 50 | 38.01 | 37.90 | 41.65 | 57.24 | 86.22 |
| 60 | 40.97 | 41.69 | 43.35 | 55.10 | 77.34 |
| 70 | 45.08 | 45.26 | 45.53 | 54.13 | 74.30 |
| 80 | 47.94 | 47.74 | 48.72 | 55.04 | 72.67 |
| 90 | 51.31 | 50.56 | 51.51 | 55.97 | 72.15 |

Table B.10: Wetstraat phase [11:00 | 15:00 | 16:00]

| 13:00 | 1 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|
| 40 | 37.21 | 37.10 | 52.07 | 143.30 | 283.73 |
| 50 | 40.63 | 41.30 | 48.44 | 102.83 | 255.23 |
| 60 | 44.19 | 44.61 | 48.74 | 79.66 | 215.49 |
| 70 | 47.94 | 47.79 | 50.89 | 70.39 | 196.30 |
| 80 | 50.51 | 50.81 | 52.87 | 69.34 | 150.72 |
| 90 | 53.11 | 54.63 | 55.01 | 66.93 | 135.58 |

Table B.11: Wetstraat phase [13:00 | 14:00 | 19:00]

| 20:00 | 1 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|
| 5 | 14.45 | 21.54 | 33.36 | 47.60 | 58.89 |
| 10 | 16.99 | 20.46 | 33.07 | 45.42 | 59.54 |
| 20 | 21.74 | 22.42 | 30.90 | 43.03 | 55.85 |
| 30 | 26.33 | 26.25 | 30.80 | 40.45 | 53.41 |
| 40 | 29.87 | 30.36 | 32.38 | 40.28 | 51.65 |
| 50 | 34.14 | 34.00 | 34.42 | 40.68 | 50.14 |

Table B.12: Wetstraat phase [20:00]

| 21:00 | 1 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|
| 5 | 11.96 | 15.49 | 23.34 | 32.92 | 41.79 |
| 10 | 14.89 | 16.36 | 23.00 | 32.24 | 42.70 |
| 20 | 20.32 | 20.39 | 23.97 | 31.34 | 41.57 |
| 30 | 24.75 | 24.88 | 26.46 | 32.24 | 40.66 |
| 40 | 29.04 | 29.28 | 29.93 | 33.47 | 39.58 |
| 50 | 32.97 | 32.64 | 33.50 | 35.88 | 40.69 |

Table B.13: Wetstraat phase [21:00 | 22:00]

| 23:00 | 1 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|
| 5 | 10.96 | 12.87 | 18.48 | 25.60 | 33.62 |
| 10 | 14.01 | 14.79 | 18.76 | 26.29 | 33.29 |
| 20 | 19.86 | 19.93 | 21.71 | 27.09 | 33.32 |
| 30 | 24.69 | 25.17 | 25.72 | 28.35 | 34.63 |
| 40 | 28.94 | 29.42 | 29.47 | 31.00 | 35.44 |

Table B.14: Wetstraat phase [23:00]

## B.4 Wetstraat Platoon

| 00:00 | 1 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|
| 5 | 9.74 | 10.31 | 12.91 | 17.00 | 20.50 |
| 10 | 13.05 | 13.17 | 14.83 | 17.70 | 21.03 |
| 20 | 19.58 | 19.38 | 20.17 | 21.47 | 24.10 |
| 30 | 25.56 | 25.70 | 26.13 | 26.88 | 28.83 |
| 40 | 31.64 | 30.79 | 31.96 | 32.13 | 33.28 |

Table B.15: Wetstraat Platoon ATWT [00:00]

| 01:00 | 1 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|
| 5 | 9.23 | 9.53 | 11.37 | 13.61 | 16.58 |
| 10 | 12.81 | 12.96 | 13.57 | 15.41 | 17.53 |
| 20 | 20.72 | 20.34 | 20.56 | 20.92 | 22.84 |
| 30 | 27.74 | 27.42 | 27.38 | 29.02 | 28.79 |
| 40 | 34.56 | 35.33 | 34.33 | 35.28 | 35.21 |

Table B.16: Wetstraat Platoon ATWT [01:00]

| 02:00 | 1 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|
| 5 | 8.71 | 9.22 | 10.42 | 11.92 | 14.45 |
| 10 | 12.83 | 13.17 | 13.12 | 14.47 | 17.08 |
| 20 | 22.19 | 20.92 | 21.39 | 22.47 | 22.77 |
| 30 | 29.05 | 29.94 | 30.63 | 29.69 | 31.43 |
| 40 | 36.35 | 37.51 | 37.92 | 37.53 | 37.34 |

Table B.17: Wetstraat Platoon ATWT [02:00 | 03:00 | 04:00]

| 05:00 | 1 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|
| 5 | 9.82 | 10.70 | 13.88 | 17.97 | 23.63 |
| 10 | 13.07 | 13.10 | 15.01 | 18.41 | 23.62 |
| 20 | 19.16 | 19.74 | 19.99 | 21.86 | 25.31 |
| 30 | 24.91 | 25.17 | 24.96 | 26.85 | 28.15 |
| 40 | 29.22 | 29.87 | 30.57 | 30.18 | 32.25 |

Table B.18: Wetstraat Platoon ATWT [05:00]

| 06:00 | 1 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|
| 5 | 16.31 | 24.79 | 37.62 | x | x |
| 10 | 18.88 | 23.77 | 37.10 | x | x |
| 20 | 23.61 | 24.04 | 34.72 | 46.52 | 61.77 |
| 30 | 27.35 | 27.65 | 33.74 | 44.77 | 58.00 |
| 40 | 31.45 | 31.56 | 34.16 | 43.72 | 55.43 |
| 50 | 34.79 | 34.82 | 36.19 | 44.20 | 54.26 |
| 60 | 38.11 | 38.26 | 38.94 | 44.58 | 53.95 |
| 70 | 41.15 | 41.47 | 41.90 | 45.07 | 54.44 |

Table B.19: Wetstraat Platoon ATWT [06:00 | 12:00]

| 07:00 | 1 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|
| 5 | 100.78 | 176.36 | 227.51 | 315.94 | 431.76 |
| 10 | 331.23 | 170.70 | 209.57 | 328.41 | 447.78 |
| 20 | 171.06 | 159.35 | 220.92 | 317.50 | 455.86 |
| 30 | 174.27 | 186.60 | 201.32 | 313.75 | 434.55 |
| 40 | 163.65 | 158.75 | 218.66 | 298.10 | 395.19 |
| 50 | 185.92 | 174.17 | 212.77 | 270.32 | 382.63 |
| 60 | 194.06 | 176.39 | 201.84 | 266.33 | 380.58 |
| 70 | 168.73 | 189.32 | 206.75 | 257.62 | 379.14 |
| 80 | 196.74 | 210.29 | 188.80 | 258.15 | 375.52 |
| 90 | 210.41 | 199.20 | 208.46 | 256.69 | 401.59 |
| 100 | 219.50 | 205.07 | 208.69 | 248.20 | 338.72 |
| 110 | 223.47 | 221.07 | 223.38 | 276.40 | 349.14 |

Table B.20: Wetstraat Platoon ATWT [07:00]

| 08:00 | 1 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|
| 5 | 23.56 | 33.44 | 49.92 | 70.74 | 154.92 |
| 10 | 24.18 | 32.99 | 49.80 | 76.42 | 181.42 |
| 20 | 27.91 | 31.38 | 46.87 | 69.31 | 177.64 |
| 30 | 32.39 | 33.02 | 45.04 | 67.23 | 162.99 |
| 40 | 36.36 | 36.61 | 44.02 | 61.47 | 131.04 |
| 50 | 40.56 | 40.17 | 44.09 | 57.49 | 107.50 |
| 60 | 43.75 | 43.39 | 45.42 | 56.46 | 93.64 |
| 70 | 46.59 | 47.07 | 47.91 | 58.35 | 85.28 |
| 80 | 50.06 | 50.84 | 51.74 | 58.10 | 93.49 |
| 90 | 53.35 | 53.56 | 53.20 | 57.90 | 78.66 |
| 100 | 55.70 | 56.00 | 56.27 | 59.70 | 81.24 |

Table B.21: Wetstraat Platoon ATWT [08:00 | 09:00]

| 10:00 | 1 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|
| 5 | 19.11 | 28.84 | 44.09 | 58.44 | 76.05 |
| 10 | 21.02 | 27.58 | 42.67 | 58.25 | 79.25 |
| 20 | 25.22 | 27.38 | 39.98 | 55.72 | 73.21 |
| 30 | 29.40 | 29.80 | 38.21 | 51.72 | 68.09 |
| 40 | 33.57 | 33.40 | 38.24 | 50.44 | 66.82 |
| 50 | 37.23 | 37.42 | 39.55 | 49.59 | 62.75 |
| 60 | 40.57 | 40.41 | 41.60 | 49.61 | 62.75 |
| 70 | 43.38 | 43.48 | 43.70 | 48.96 | 61.74 |
| 80 | 46.95 | 46.06 | 47.26 | 51.00 | 60.23 |
| 90 | 49.60 | 49.33 | 49.60 | 52.18 | 61.43 |

Table B.22: Wetstraat Platoon ATWT [10:00 | 17:00 | 18:00]

| 11:00 | 1 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|
| 5 | 18.47 | 27.80 | 41.85 | x | x |
| 10 | 20.77 | 26.44 | 41.27 | x | x |
| 20 | 25.01 | 26.24 | 38.33 | x | x |
| 30 | 29.16 | 29.20 | 37.22 | x | x |
| 40 | 32.83 | 33.03 | 37.47 | 47.81 | 63.16 |
| 50 | 36.35 | 36.75 | 38.08 | 47.76 | 61.48 |
| 60 | 39.89 | 39.74 | 40.73 | 47.95 | 59.55 |
| 70 | 43.22 | 42.81 | 42.88 | 49.34 | 59.86 |
| 80 | 46.08 | 46.44 | 45.94 | 49.20 | 59.29 |
| 90 | 48.56 | 48.66 | 49.16 | 51.09 | 58.31 |

Table B.23: Wetstraat Platoon ATWT [11:00 | 15:00 | 16:00]

| 13:00 | 1 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|
| 5 | 20.72 | 31.29 | 46.61 | x | x |
| 10 | 22.64 | 30.26 | 46.06 | x | x |
| 20 | 26.95 | 29.08 | 43.65 | x | x |
| 30 | 31.07 | 31.27 | 41.77 | x | x |
| 40 | 34.87 | 35.03 | 40.23 | 54.32 | 75.61 |
| 50 | 38.91 | 38.71 | 41.77 | 54.25 | 70.90 |
| 60 | 42.16 | 42.00 | 43.91 | 53.30 | 70.38 |
| 70 | 45.26 | 45.27 | 46.23 | 53.44 | 67.20 |
| 80 | 47.87 | 47.82 | 48.67 | 54.43 | 69.19 |
| 90 | 51.27 | 51.11 | 51.84 | 54.84 | 66.09 |

Table B.24: Wetstraat Platoon ATWT [13:00 | 14:00 | 19:00 ]

| 20:00 | 1 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|
| 5 | 14.09 | 20.15 | 31.03 | 42.80 | 54.06 |
| 10 | 16.66 | 19.75 | 30.41 | 41.89 | 52.35 |
| 20 | 21.67 | 22.20 | 29.32 | 39.17 | 51.58 |
| 30 | 25.75 | 26.14 | 29.58 | 38.06 | 49.75 |
| 40 | 29.74 | 29.69 | 31.13 | 37.91 | 48.70 |
| 50 | 32.98 | 33.03 | 34.31 | 38.89 | 47.89 |

Table B.25: Wetstraat Platoon ATWT [20:00]

| 21:00 | 1 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|
| 5 | 11.84 | 14.98 | 22.96 | 31.29 | 42.04 |
| 10 | 14.75 | 16.24 | 23.24 | 31.67 | 40.69 |
| 20 | 20.38 | 20.32 | 23.91 | 30.98 | 39.79 |
| 30 | 24.80 | 24.92 | 26.48 | 31.11 | 39.22 |
| 40 | 28.55 | 28.73 | 29.52 | 32.31 | 39.85 |
| 50 | 32.50 | 32.41 | 32.91 | 34.72 | 39.71 |

Table B.26: Wetstraat Platoon ATWT [21:00 | 22:00]

| 21:00 | 1 | 5 | 10 | 15 | 20 |
|---|---|---|---|---|---|
| 5 | 10.87 | 12.57 | 19.03 | 25.43 | 32.66 |
| 10 | 14.02 | 14.75 | 18.72 | 25.61 | 32.85 |
| 20 | 19.27 | 19.67 | 21.26 | 26.30 | 33.18 |
| 30 | 24.29 | 24.75 | 25.43 | 28.11 | 34.28 |
| 40 | 29.37 | 28.65 | 29.50 | 30.26 | 34.18 |

Table B.27: Wetstraat Platoon ATWT [23:00]

# Bibliography


[Bal98] Osman Balci. Verification, validation, and testing. In J. Banks, editor, *Handbook of simulation*, pages 335–393. John Wiley and Sons, New York, NY, 1998.

[DH05] Jean-Patrick Lebacque Dirk Helbing, Stefan Lammer. Self-organized control of irregular or pertubed network traffic. In C. Deissenberg and R. F. Hartl, editors, *Optimal Control and Dynamic Games*, pages 239–274. Springer, Dortrecht, 2005.

[DS04] Kurt Dresner and Peter Stone. Multiagent traffic management: A reservation-based intersection control mechanism. In *The Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 530–537, 2004.

[ea] Gaston Escobar et al. `http://sourceforge.net/projects/morevts`.

[FSS04a] M. Ebrahim Fouladvand, Zeinab Sadjadi, and M. Reza Shaebani. Characteristics of vehicular traffic flow at a roundabout. *Physical Review E*, 70:046132, 2004.

[FSS04b] M. Ebrahim Fouladvand, Zeinab Sadjadi, and M Reza Shaebani. Optimized traffic flow at a single intersection: Traffic responsive signalization. *J. Phys. A: Math. Gen.*, 37:561–576, 2004.

[Ger05] Carlos Gershenson. Self-organizing traffic lights. *Complex Systems*, 16(1):29–53, 2005.

[gld] `http://www.students.cs.uu.nl/swp/2001/isg/index.html`.

[Her04] Luis Miramontes Hercog. Co-evolutionary agent self-organization for city traffic congestion modeling. In *Genetic and Evolutionary Computation - GECCO 2004: Genetic and Evolutionary Computation Conference, Seattle, WA, USA, June 26-30, 2004. Proceedings, Part II*, pages 993–1004. Springer-Verlag, 2004.

[KEW] Renato Levy Kutluhan Erol and James Wentworth. Application of agent technology to traffic simulation. `http://www.tfhrc.gov/advanc/agent.htm`.

[KH03] Robert Kölbl and Dirk Helbing. Energy laws in human behaviour. *New Journal of Physics*, (5):48.1 – 48.12, 2003.

[mor] `http://morevts.cvs.sourceforge.net/morevts/`.

[Nag06] Kai Nagel. *Multi-Agent Transportation Simulation*. Book in progress, 2006.

[Ohi97] Toru Ohira. Autonomous traffic signal control model with neural network analogy. In *Proceedings of InterSymp'97: 9th International Conference on Systems Research, Informatics and Cybernetics*, Baden-Baden, Germany, August 1997. SCSL-TR-97-004.

[OK02] Sigurdur Olafsson and Jumi Kim. Simulation optimization. In J. L. Snowdon E. Yücesan, C.-H. Chen and J. M. Charnes, editors, *Proceedings of the 34th conference on Winter simulation: exploring new frontiers, San Diego, California*, pages 79–84. Winter Simulation Conference, 2002.

[Sot] http://homepages.vub.ac.be/ cgershen/sos/SOTL/SOTL.html.

[stoa] `http://www.comite-verkeersveiligheid.be/index.php?option=com\_content\&\\task=view\&id=57\&Itemid=85`.

[stob] `http://nl.wikipedia.org/wiki/Remweg`.

[WS02] U. Wilensky and W. Stroup. NetLogo HubNet Gridlock model, 2002. http://ccl.northwestern.edu/netlogo/models/HubNetGridlock.

[WVVK04] M. Wiering, J. Vreeken, J. Van Veenen, and A. Koopman. Simulation and optimization of traffic in a city. In *IEEE Intelligent Vehicles Symposium (IV'04)*, pages 453–458. IEEE, 2004.