Improving vehicle routing using traffic predictions

Master's thesis

Pieter Loof

Improving vehicle routing using traffic predictions

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Pieter Loof
born in Rhenen, the Netherlands

Algorithmics Group
Software and Computer Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
alg.ewi.tudelft.nl

Almende B.V.
Westerstraat 50
Rotterdam, the Netherlands
www.almende.com

Improving vehicle routing using traffic predictions

Author: Pieter Loof
Student id: 1362925
Email: pcloof@gmail.com

Abstract

Vehicle routing through road networks is an important topic of research: time and money can be saved by reducing traffic jams, which would also reduce the burden on the environment. The problem of minimizing the travel time for a vehicle trip is easy to grasp but hard to solve, it is a highly complex shortest path problem. Our goal is to show that routing advice can be improved by using historical traffic data to predict the traffic conditions in the nearby future. We introduce several algorithms that combine historical data with live data in a smart way, our algorithm set contains fixed path algorithms, adaptive path algorithms and policy algorithms. In order to test their performance we create historical scenarios and test scenarios using micro-simulations because they can produce network-wide traffic data. We evaluate the realism of the simulations and highlight the problems that are encountered when trying to mimic reality. The results of the routing algorithms are compared to obtain a good insight in the advantages and drawbacks of the different algorithm types. We show that the best algorithms clearly outperform an algorithm based on the concept of modern in-car routing devices, even when only a limited amount of live data is available, which clearly shows that routing advice can be significantly improved using traffic predictions.

Thesis Committee:

Chair:	Prof. dr. C. Witteveen, Faculty EEMCS, TU Delft
University supervisor:	Dr. M.T.J. Spaan, Faculty EEMCS, TU Delft
Committee Member:	Dr. V.L. Knoop, Faculty CEG, TU Delft
Committee Member:	Dr. D.W.F. van Krevelen, Almende B.V.

Preface

This thesis is the result of my final master project, performed in cooperation with the Algorithmics group at the TU Delft. Special thanks goes to my daily supervisor Matthijs Spaan for his guidance and advice during the project. Next to that I am grateful that Almende gave me the opportunity to work together with them during this project; Almende is a small research company situated in Rotterdam that is specialized in information and communication technologies. Joris Scharpff also advised me regularly on behalf of Almende. I am also grateful for receiving a part of my data from Bram Bakker who works for the company Cygnify.

I enjoyed the stimulating working environment, mostly the regular lunches we had together with the group. I also had the pleasure of having a working desk in a room together with fellow master students, nice memories remain of the fun we had together and our numerous coffee breaks.

Pieter Loof
Delft, the Netherlands
August 7, 2014

Contents

List of Figures

Chapter 1

Introduction

The Dutch road network is an intensively used road network where traffic jams occur on a daily basis. Many people spend a lot more time on the road than they would like to, commuters often find themselves in rush hour traffic jams. A lot of work is already performed on the topic of routing advice. Although the problem is easy to understand, solving it is much harder.

Take for example the network shown in Figure 1.1 which contains the ring road around Rotterdam and a few other main roads. The start and destination locations of a trip for which providing good routing advice is challenging are highlighted. The shortest path leads straight through the center of the city. It might seem like a good route but one should take into account that the allowed speed is much lower than on the ring road and that there are several traffic lights. The shortest route might therefore not be optimal. One could also

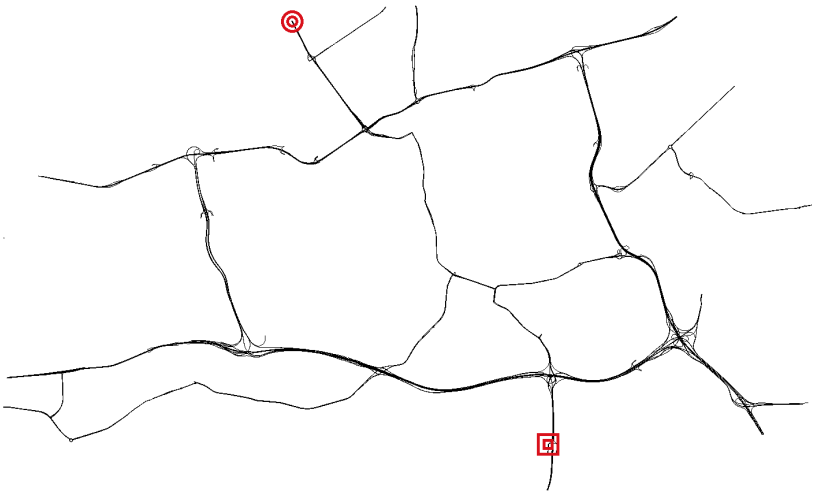


Figure 1.1: The Rotterdam network with its highway ring road and a few primary roads. The departure and arrival location for an example trip are highlighted with circles respectively squares.

advise to use the ring road, one can drive clockwise or counterclockwise around the city, both routes are about equal in length. Which one is faster can change from hour to hour depending on the traffic conditions. It might be the case that we learn of a fast growing traffic jam on the route we planned to take just before reaching the ring road, then we can adapt to the new situation by choosing a different route. Or maybe a traffic jam always starts around seven o'clock in the morning on our favourite route and suppose we reach the ring road at a few minutes before seven. Today's navigation devices are likely to advise to take that route anyway, despite the fact that one will likely end up in a traffic jam halfway. By using historical data we can gain insight in the regular traffic patterns for such networks and by combining this information with recent or live data we can predict how the traffic conditions will evolve, which makes better routing advice possible.

Imagine that your routing device correctly foresees future traffic jams and proactively leads you around it whenever a faster detour is available. Our aim is to make this possible and therefore we focus on coming up with routing algorithms that combine historical data with recent or live data in order to improve the quality of routing advice for individual vehicle trips. When giving routing advice we assume the current traffic conditions to be given, so we assume that the provided routing advice has no impact on the traffic conditions. We can safely state that changing the route of one vehicle has very little effect on the total traffic conditions. However, by making this choice we have to make clear that the results of the routing algorithms do not necessarily apply when a significant part of all vehicles follows their advice.

In the remainder of this chapter we will define the main problem to solve and further discuss the aspects that should be taken into account. This leads to a main research question and after that we will discuss what has already been done in this field of research and what our contributions are.

1.1 Problem description

Simply said the problem is to get any person as fast as possible from A to B by car, within the limitations of physics and the law. However, reality is very complex since traffic results from constantly changing human activities and the fact that vehicles are still driven by humans with different driving behaviour. Furthermore, traffic can be influenced a lot by accidents, road constructions, the weather, etc. To be able to tackle this problem we need to simplify the problem and leave out the less important factors. Road networks can be represented as graphs in which junctions become nodes and the roads are the edges. The edge weights can be defined as the time needed to traverse the edges. Finding the fastest path therefore comes down to finding the shortest path in a graph.

To make our model as realistic as possible, a graph representing the traffic conditions for a certain network and time span has quite complicated properties. First of all, edge weights are not constant but time-dependent since the travel time depends on constantly changing traffic conditions. There is however a constraint on how edge travel times develops over time. When dealing with road networks it makes sense to assume the FIFO property (first in, first out) since it is true that one cannot arrive earlier by departing later, when keeping the

route and vehicle parameters constant. Assuming the FIFO property basically means that the expected travel time of an edge does not change by more than one second per second.

Secondly, travel times of edges are uncertain. At the time we want to give routing advice for a trip to be performed, the future traffic conditions are unknown and therefore the travel time for different edges is not known in advance. When trying to find the shortest path, we focus on finding the path with the lowest expected travel time. Although incorporating variance in the quality measure of paths is relevant, it is not within the scope of this work to explicitly model travel time uncertainties.

A third property that we consider is the fact that travel times of edges are correlated. When a traffic jam occurs it has an effect on upstream roads. There is also a correlation that is not directly causal: during rush hours the vehicle density on a lot of roads increases around the same time, while the whole network is almost empty during the night. The common cause is a changing need for humans to travel. Another example is that the average speed might drop in a large area during heavy rainfall. Therefore, distributions for the travel times of edges are not to be assumed independent.

Smart routing algorithms can come up with a route that seems pretty fast on the moment of departure but because of changing traffic conditions during the trip, which route is optimal might change during the trip. Therefore we also consider the possibility of adaptiveness of routing advice while executing vehicle trips. Adaptiveness can be reactive but also proactive. Based on a combination of historical and live data, algorithms can try to predict the future traffic conditions and base their routing advice on this prediction.

1.2 Research questions

Our main goal is to improve vehicle navigation by predicting the future traffic conditions and using these predictions to compute better routing advice. Therefore, our main research question is:

- Can traffic forecasting based on historical data improve quality of routing advice, and if yes, by how much?

As already explained above, we expect that today's traffic conditions are similar to patterns revealed by historical data, so we expect that good predictions can be made. Some subquestions arise when we try to answer our main question. For computing routing advice, different types or routing algorithms have been used already by others, as we will see in Section 1.3. This leads to the question which algorithm types can produce good routing advice under what conditions, which results in the first subquestion stated below. In order to make good predictions the routing algorithms have to combine recent traffic information with historical data. How this should be done is the topic of the second subquestion. The quality of the routing advice and the possibilities for combining recent data with historical data depend on which data is available and in what amounts. We will introduce different algorithms that use different types and amounts of data such that we can draw conclusions about the usefulness of certain data. By varying the amount of historical data available

to the algorithms we obtain insight in how much historical data is required for improving routing advice. Our third subquestion covers these topics.

In Chapter 4 we explain why simulations are potentially useful for obtaining realistic traffic scenarios and what the other advantages and drawbacks are. The last subquestion deals with the question whether traffic simulations are sufficiently realistic: can traffic patterns be mimicked correctly by simulations and is the error bounded by a relatively low constant value? A second question regarding the usefulness of simulations is whether simulations of large networks can be done fast enough. Simulations could also be used for simulating future traffic conditions, but in order to become a useful prediction they must be performed in multiple real-time.

To wrap up, the four subquestions are:

1. What types of algorithms provide good routing advice in real world situations and what are their strengths and weaknesses?
2. How can we combine historical data with recent or live data in a meaningful way?
3. How does the performance of routing algorithms depend on the type and amount of available data?
4. Are traffic simulations useful in the process of obtaining traffic scenario data and computing routing advice?

1.3 Related Work

A lot of work has been performed already on the topic of finding shortest paths on graphs. Since we are dealing with road networks in which the weights of the edges are related to travel times, the edge weights are strictly positive. For fixed graphs with positive edge weights we have the well known algorithm from Dijkstra [14] and the A* variant from Hart et al. [21]. These algorithms can find the shortest path in a fixed graph efficiently. However, we are interested in graphs with both time-dependent and uncertain edge weights.

1.3.1 Time dependent and stochastic edge weights

The first paper dealing with graphs with time-dependent networks is to our knowledge by Cooke and Halsey [12]. The first paper we found that combines time dependency with stochasticity for edge weights is by Hall [20]. The authors show that Bellman's equation as stated by Bellman [3] cannot be used since the assumption that the best solution for a subpath is also the best solution when extending the path is wrong, which is shown using a simple example network. But in this example a vehicle is not allowed to wait and the FIFO property is not assumed to be true. However, it is safe to assume the FIFO property despite the fact that in reality vehicles overtake each other regularly. In practice small exceptions to the FIFO rule are possible: on a multi-lane road a vehicle that comes to a halt in front of a traffic light might be overtaken by a vehicle with the same driving style that arrives later and meets a green light before it comes to a halt. However, when driving a vehicle one does

not know on what exact times traffic lights will be red or green so the expected delay caused by a traffic light is equal for each vehicle. Therefore, one can not forward the expected arrival time by voluntarily waiting at intermediate locations, so we can safely assume the FIFO property. This is also defended by Sung et al. [31] for road networks in general and by Qian et al. [27] for networks with traffic lights. This is also noticed by Chen et al. [9], they conclude that by using the FIFO property finding the path with the lowest expected travel time can be done faster. Dean [13] show that finding the shortest path in a general time-dependent graph is already a NP-hard problem but under the FIFO assumption some nice properties hold. They explain that in such a case waiting is never beneficial. In combination with the assumption of having only positive edge weights, shortest paths are always acyclic and any subpath of a shortest path is also a shortest path itself, which makes clear that under the FIFO assumption the Bellman equation holds. We conclude that it is safe to assume the FIFO property and that this gives way to finding faster path finding algorithms.

1.3.2 Spatial correlation of edge weights

In a lot of work in the field of shortest path finding on road networks the assumption is made that the travel times of roads are normally distributed, according to Chang et al. [6]. The assumption is also realistic according to Rakha et al. [28]. However, as explained in Section 1.1, edge weights are correlated so assuming that edge distributions are independent is undesirable. Because we want to incorporate spatial correlations a lot of proposed algorithms are of little value for solving our problem. Some other choices for edge travel time distributions also do not fit the problem we are investigating, for example the fuzzy edge weights approach taken by Hernandes et al. [22]. Spatial dependencies are taken into account in the work of Chen et al. [8], who take the approach of having each edge weight depend on the weights of nearby edges. Ji et al. [23] show that taking spatial dependencies into account matters: assuming a positive correlation between links on a single path increases the variance over the total path.

1.3.3 Adaptive path finding

Traffic conditions can change during the trip, which asks for algorithms that can adapt the advised route on-line. Wellman et al. [33] point to the usefulness of adaptive path planning. Policies are mentioned, which are conditional paths: they prescribe for each location and possible arrival time how to continue the route. Fu [17] compares algorithms that are non-adaptive to algorithms that are reactive and algorithms that are proactive. They show that adaptive algorithms perform well and that proactive algorithms in which uncertainties are incorporated perform even better. They assume that the travel time of an edge is known once it is reached. The idea of providing only traffic information of close-by edges to a routing algorithm while a trip is performed is also used by a few algorithms that we will investigate.

The idea of using routing policies is also explained by Gao et al. [18]. Routing policies are also known in literature as hyperpaths or strategies. The authors introduce a different idea for the probability distribution of edge weights: instead of defining a separate distri-

bution for each edge a single distribution over whole road networks is used. They explain that the distribution is based on a set of support points where each support point is a vector defining one weight for each edge. A policy algorithm then computes the shortest path for each support point. By defining one distribution in this way, spatial correlations are taken into account. However, in the simple example they give, they define four support points resulting from all possible combinations of possible travel times over a few different edges. By defining a support point for each combination of possible travel times, information about spatial correlations is thrown away. Furthermore this can result in a huge number of support points for real world examples. We will continue in this approach by defining traffic scenarios as time-dependent support points. A traffic scenario consists of one travel time for each node-time pair in the network. We will look at a few algorithms that make use of these traffic scenarios in different ways. We will also introduce algorithms that compute routing advice for all possible combinations of edge travel times, but with some limitations.

1.3.4 Reliable path finding

As shown by Wellman et al. [33], Dijkstra's algorithm does not guarantee optimality in time-dependent stochastic networks. Suppose that the first part of a route has an uncertain travel time, then the time at which the second part of the route starts is not known in prior. In a network that is both stochastic and time-dependent this means that the distribution itself for the second part of the route depends on the travel time of the first part. The authors explain this with a small example that shows that using the expected travel time per part of the path is not sufficient. A model that uses the expected travel time per edge is usually referred to as the expected value model, as explained by Chen and Ji [7]. The authors explore a different branch of research that focuses on α-reliable paths, which means trying to find the path for which the probability of arriving before a certain deadline is maximal.

In this work we focus on finding the path with the lowest expected travel time, we do not explicitly model stochasticity of edge weights. Note that the output of a simulation results in one time-dependent travel time per road segment. One could argue that when providing a set of historical scenarios, we are actually providing a distribution for the travel time of each edge. The routing algorithms that we will introduce make use of this data in different ways, some of them take multiple possible travel times per edge into account. Most policy algorithms that we test compute decisions for different arrival times at intermediate locations because they consider different possible travel times for roads. However, the decisions are based on averaged road travel times, the travel time variance of edges or paths is not incorporated when giving routing advice. When introducing the routing algorithms, we will mention how travel time variance could be incorporated.

1.3.5 Predicting future traffic conditions

An important aspect in producing good routing advice is that routing algorithms take future traffic conditions into account although the future is of course uncertain. Vlahogianni et al. [32] give an overview of work performed in the area of traffic prediction. They point to the fact that although traffic data is available per minute or less, most researchers aggregate

the data over intervals of five minutes because traffic flow and speed fluctuate too much on small time scales. Another conclusion they draw is that travel time predictions are more easily applied to highways than to urban areas since it is harder to obtain reliable speed data for urban areas. In this work we will also focus on highway networks and main roads. We will aggregate traffic data per five minutes before offering it to our routing algorithms. The reasons for these choices are further discussed in Chapter 4.

Which data should be used and how to predict future traffic conditions from it? Using only live data might not be sufficient since it does not provide any information about how the traffic could evolve while historical information can help to predict the future. This problem also has been investigated extensively in literature. Chien and Kuchipudi [10] compare the difference of the prediction error when using the latest live data to using historical data and conclude that using the latest live data gives better results than using historical data. Of course this depends quite a lot on the way in which historical data is used and on the time span of the test. According to Chrobok et al. [11] live data is mainly useful for short time predictions, for example only fifteen minutes ahead. However, we want to provide routing advice for trips that can take a few hours, so using live data only might not be sufficient. The authors further explain that network conditions depend on the time of the day and on the day of the week, they distinguish four different day types. They also introduce the idea of comparing recent live data with data from the different day types in order to automatically determine to which day type a given scenario is most similar. Their results show that this classification works very well. In our work we will also use this approach. In Section 1.3.3 we already introduced the idea of using historical traffic scenarios as input to routing algorithms. Instead of predefining a set of different day types, some of our algorithms will compare received live data with each provided historical scenario individually.

Real data can come from different sources. In the Netherlands the highway network is covered quite well with inductive loop detectors that measure vehicle flow and speed on fixed locations in the road network. Van Lint et al. [26] use loop detector data, they also discuss the problem of data incompleteness and how to deal with this. We will also investigate this problem in Chapter 4. Since travel time estimations are usually calculated by first computing average vehicle speeds within a road network, a good alternative real data source is floating car data that comes from in-car devices. A great advantage is that this data can provide speed data for the whole network, provided that a significant part of all vehicles contributes to the gathering of data. However, the availability of floating car data on a larger scale is limited, while loop detector data exists in abundance. Because historical loop detector data was made available for this project, we will focus on using loop detector data. During the course of this project, live loop detector data even became freely available in the Netherlands.

1.4 Contributions

In this work we test several routing algorithms, based on three basic algorithm types: fixed path algorithms, adaptive path algorithms and policy algorithms. By testing several variants based on these types we can answer the first research subquestion. We come up with a few

ways of combining historical and live data and by comparing the results of the algorithms we deal with the second subquestion. We test a fixed path algorithm that does not incorporate any traffic, which can be compared to basic route planners that are available on the internet. A bit smarter version also makes use of available recent traffic data, one might think of the routing advice based on current traffic which is provided when planning a trip using Google Maps [19] nowadays. Another improvement used in practice is the use of live data to update routing advice, this is done by modern in-car routing devices. Such devices sometimes alter the advised route when information about a traffic jam ahead becomes available. These routing systems fall in the category of adaptive path algorithms in the way we use this term. These systems are usually reactive but not proactive: they do not foresee traffic jams but only react once they occur. We will also include such an adaptive path algorithm so that we can compare our other algorithms with it. The idea of using policies traces back to the work of Gao et al. [18] as mentioned above already, we will also test multiple policy algorithms.

Our major contribution is that we create and test algorithms that make use of historical data to predict future traffic conditions. The results from our experiments can be used to make a lot of useful comparisons. Our algorithm versions use different types of current or live data in combination with historical data and they are tested with different amounts of data, which leads to an answers for the third research subquestion. Then we can also answer the first subquestion because the results of fixed path algorithms are compared to the results of adaptive path algorithms and policy algorithms. The set of algorithms also contains a few algorithms that do not use historical data in order to find out how important using historical data is. We show that combining historical traffic data with recent or live data results in improved routing advice.

We use traffic simulations for producing scenarios, which is a direction of investigation that is taken by few people. We will explain why simulations are useful. We also investigate the quality of the chosen traffic simulation software, which provides insight in the problems that threaten the realism of simulations. We will highlight problems and provide a solution or workarounds for some of them, which results in answering our fourth research subquestion.

1.5 Outline

We now provide a short outline for the rest of the thesis. In Chapter 2 we define the routing problem in a formal way and we define the main types of algorithms that we use. In the next chapter all the routing algorithms are introduced. Chapter 4 aims at explaining why traffic simulations are useful and a comparison between good traffic simulation packages is given. After that the parameters of the simulations are discussed and the chosen values are motivated. We will elaborate on the problems that we encountered regarding the realism of simulations. Chapter 5 then explains the experiments that are performed to test the performance of the routing algorithms. In Chapter 6 the results of the experiments are presented and discussed. Finally we come to our concluding chapter in which the main results are summarized. More detailed information about the testing framework or the implementation of routing algorithms can be found in the appendices.

Chapter 2

Formal definitions

In this chapter we will provide formal definitions for our representation of road networks and for routing on them, which includes an explanation of how the travel times of routes are computed. After that we will introduce the main types of routing algorithms on which all our algorithms are based. By introducing these algorithm types we already gain insight in some of their advantages and drawbacks, which is a first step in answering the first research subquestion.

2.1 Formal problem definition

The problem we are dealing with is finding the fastest route between a chosen start and end location in a road network, we formalize this problem now. The road network is represented as a graph $G = (N, E)$ where N is a set of nodes and E is a set of directed edges between pairs of nodes. T is a set of equidistant discrete time points with a distance of one or a few minutes. We also have a set of traffic scenarios X, each scenario contains the average speed of all edges for every time point in T. More formally, $x : (e_{fg}, t_i) \rightarrow v_{fg}^{x,t_i}$ with $x \in X$, $e_{fg} \in E$ an edge from node f to g and $t_i \in T$. So within each traffic scenario the average speed for each edge changes every few minutes. The weight of an edge is defined as the travel time of the edge, which in its turn depends on the travel speed of the edge. The weights of edges, so their travel times, are not stored separately because they are computed on demand from the average speed data for any continuous time value. Nonetheless, we will introduce a parameter for edge weights: $w_{fg}^{x,t}$ where time t is a real number.

An optimal routing problem is defined as finding a path $p = \{e_{si}, e_{ij}, \ldots, e_{vw}, e_{wd}\}$ from start node s to destination node d, given departure time t_s, such that the travel time is as low as possible. Computing the travel time of a path for a given scenario $x \in X$ is done quite accurately using the time-dependent weights. The travel time w_{si}^{x,t_s} for the first edge of the path is computed based on the departure time, then the time at which the second edge is reached is known, so the travel time of the second edge can be computed. In this way, the travel time per edge of the path is computed which finally results in the total travel time for the whole path. How the travel time of an edge is computed is explained in more detail in the next section.

2.1.1 Computing edge travel times

In Section 2.1 we introduced a variable for the travel time of an edge: w_{fg}^{x,t_f}. We know that the input data from historical traffic scenarios contains the average speed for each edge for a set of discrete and equidistant time intervals. This speed data has the form $v_{fg}^{x,t_1}, v_{fg}^{x,t_2}, \ldots, v_{fg}^{x,t_h}$ for a set of discrete times $\{t_1, t_2, \ldots, t_h\}$. A first thought might be that the travel time of an edge e_{fg} for departure time t_f and traffic scenario x can be computed by dividing the distance of the edge by the average speed. In order to make the speed a continuous function of the departure time, we could choose to use linear interpolation between data points. Suppose that the departure time lies between two time points for which the average speed is known: $t_a < t_f \leq t_b$, then we could compute the travel time as follows:

$$w_{fg}^{x,t_f} = \frac{\| e_{fg} \|}{(t_b - t_f) v_{fg}^{x,t_1} + (t_f - t_a) v_{fg}^{x,t_2}} \tag{2.1}$$

The problem with this approach is that it can lead to completely erroneous travel times that do not respect the FIFO property. In the introduction we explained that it is reasonable to assume the FIFO property so we do not want our definition of edge travel times to spoil that. We can give a simple example to show the incorrectness of this method. Suppose we have a piece of road with a length of 10 km. Suppose that the average speed at time t_1 is 10 km/h and ten minutes later, at time t_2, the average speed is 50 km/h. With the method as we just described, the travel time when departing at t_1 would be one hour but by leaving ten minutes later the travel time would only be twelve minutes. This method extrapolates the speed at the moment of departure to the whole time span of traversing a road, neglecting the fact that during the trip new time points pass by for which the data prescribes other average speeds. This example might look exaggerated, but in our framework we tested traffic scenarios based on real data for the FIFO property, and the property can indeed be violated using this definition of edge travel times. Therefore we compute edge travel times in a more sophisticated manner such that the problem we just described does not occur. We define the speed function as follows:

$$v_{fg}^{x}(t) = \begin{cases} v_{fg}^{x,t_1}, & t < t_1 \\ v_{fg}^{x,t_h}, & t \geq t_h \\ \frac{(t_{i+1} - t) \cdot v_{fg}^{x,t_i} + (t - t_i) \cdot v_{fg}^{x,t_{i+1}}}{t_{i+1} - t_i}, & \text{else} \end{cases} \tag{2.2}$$

The function linearly interpolates between the data points for which the speed is known. The speed of the first time point is extended to negative infinity and the speed of the last time point to positive infinity. An example graph is shown in Figure 2.1. Any area under such a graph represents a travel distance, since it is a integral of speed over time. The time of entering the edge defines a left boundary of an area under the graph. The algorithm for determining w_{fg}^{x,t_f} finds a right boundary t_g such that the area under the graph equals the length of e_{fg}; more formally:

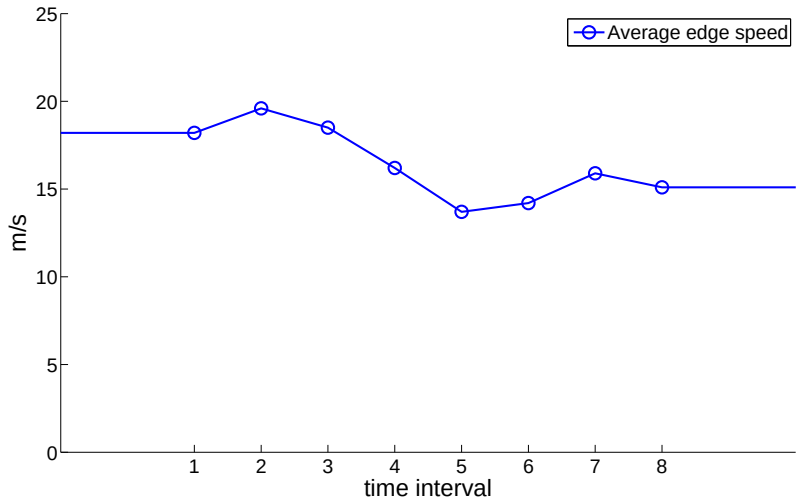


Figure 2.1: An example graph for a speed function for an edge like in Equation 2.2 for a traffic scenario with eight data points that are highlighted by the circles.

$$w_{fg}^{x,t_f} = (t_g - t_f \mid \int_{t=t_f}^{t_g} v_{fg}^{x}(t)\,\mathrm{d}t = \| e_{fg} \|) \tag{2.3}$$

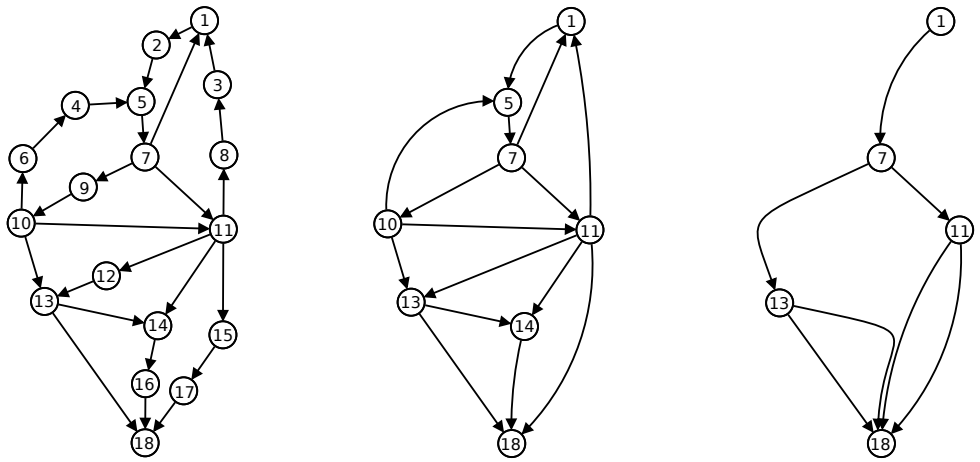
Finding t_g in Equation 2.3 is done in a recursive way. The function computes whether the end of the edge is reached before the next speed data point which is rather easy since the speed function is linear between two consecutive data points. If the end of the edge is not reached before the next data point, then a recursive call is made with the next data point as start time, for the remaining length of the edge. Multiple of such recursive calls are made when traversing the edge takes several time intervals. At the last recursive call a right boundary must be found for the area under a linear piece of the speed function, say between data points v_{fg}^{x,t_i} and $v_{fg}^{x,t_{i+1}}$. If we define a new time scale $\tilde{t}$ that is zero at the start of this interval, then we can write the speed function for this interval as follows:

$$y = a\tilde{t} + b, \quad \text{with } a = \frac{v_{fg}^{x,t_{i+1}} - v_{fg}^{x,t_i}}{t_{t_{i+1}} - t_{t_i}},\ b = v_{fg}^{x,t_i} \tag{2.4}$$

Here, a is the speed difference divided by the time span. Finding an area below such a linear function means solving the following equation

$$Y = \int_0^{\tilde{t}_g} a\tilde{t} + b\,\mathrm{d}\tilde{t} = \left[\frac{1}{2}a\tilde{t}^2 + b\tilde{t} + c\right]_0^{\tilde{t}_g} = \frac{1}{2}a\tilde{t}_g^2 + b\tilde{t}_g \tag{2.5}$$

Equation 2.5 can be solved using the quadratic formula. The procedure above results in an algorithm that computes the travel time of an edge accurately for any departure time



(a) OSM network (b) SUMO network (c) routing network

Figure 2.2: An example OSM road network and how this network might look in SUMO after simplification is applied. To the right is a routing network derived from the SUMO network as a result of finding four good paths from node 1 to 18.

for any traffic scenario. This procedure is used whenever the travel time of a path through a time-dependent network is computed.

2.1.2 Routing networks

Road networks can be very large and they might contain a lot of edges that are not relevant for certain routing queries. In order to simplify and speed up the computations of routing algorithms, a routing network $R = (N', P)$ is computed as a subset from a graph G for a given routing problem with a specific start and end location, it holds that $N' \subseteq N$. P is a set of subpaths, each subpath consists of one or more consecutive edges of the SUMO network. An algorithm is made that finds a set of short paths between the start and end location. The resulting set of paths can be transformed to a set of subpaths such that routing decisions only have to be made at the end of each subpath. To accomplish this each path is split up at a certain node when different paths exit from it using different outgoing edges. Together with the start and finish node, these nodes form the set N', called the decision nodes. A simple example is shown in Figure 2.2.

In this example four paths are explored on the SUMO network, these four paths can be expressed as the list of nodes they visit: (1, 5, 7, 10, 13, 18), (1, 5, 7, 10, 13, 14, 18), (1, 5, 7, 11, 18) and (1, 5, 7, 11, 14, 18). Since all paths go from node 1 via node 5 to node 7, the decision at node 5 is trivial and it is therefore discarded. The subpath from node 1 to 7 is contained in each of the four paths but the subpath has to be stored only once. From node 7 two paths go via node 10 and two paths exit towards node 11 so node 7 is a decision node. At node 14, two paths come in from different directions but since they exit it using the same remaining route, the decision is trivial so node 14 is not a decision node. The edge

from node 14 to 18 is used by two different subpaths, so subpaths are allowed to overlap partially.

In large road networks, finding multiple good paths is not a trivial task. Setting the constraint that each next path to find is not allowed to be exactly equal to already found paths is often not sufficient. Suppose for example that the algorithm first finds the shortest path between a given start and end node. When it searches for a second good path, it might copy the first path almost exactly except that it advises to take a highway off-ramp and then the next on-ramp. A lot of very similar paths could be found by the algorithm. In order to stimulate the algorithm to find diverse paths, weight of edges are increased each time an edge is part of a found path. Edge weights are multiplied with a factor $h(c,k)$, where c is the number of times an edge is used in previously found paths and k is the number of paths to find, the formula that we use is shown in Equation 2.6. The weight factor of edges increases linearly with their usage count such that each next path search is stimulated to find new route options. The exception to this linear increment is when a path is used only one, two or three times before, then its weight factor is set to 1.5. Due to this choice, the first few paths found always have a nice level of diversity, regardless of the number of paths to find.

$$h(c,k) = \begin{cases} 1, & c = 0 \\ 1.5, & 0 < c < 4 \\ 1.5 + 0.8 \cdot \frac{c-3}{k-3}, & c \geq 4 \end{cases} \tag{2.6}$$

We will make use of routing networks for all the algorithms that we test. Computing a routing network can be rather time-consuming. Suppose we have a network with n nodes and m edges. In order to build the routing network, k shortest path searches are performed on the SUMO network using the A* algorithm, which has a total runtime order $O(k(n \log n + m))$. A typical value for k is 100, which is a sufficient number of paths to create a routing network with enough routing options for a trip of a few hundred kilometers. Although the runtime order is low, the number of nodes and edges can be quite large. A few simple routing algorithms can perform much faster without using a routing network, but we test all algorithms on routing networks because it makes the comparison easier. Another reason is that we provide historical data to a lot of algorithms, this data is defined per subpath of the routing network.

2.2 Routing algorithm types

We will now look into the different types of routing algorithms. We discuss three main types: fixed path algorithms, adaptive path algorithms and policy algorithms. The various algorithms introduced in the research field of shortest path finding are usually based on one of these three types. For finding possible paths, Dijkstra's algorithm [14] can be used. The A* variant introduced by Hart et al. [21] adds a heuristic for guiding the search towards the destination. We will use A* for all routing algorithms whenever possible because it is either faster than Dijkstra's algorithm or the graph is so small that the runtimes are very low such that the work overhead for A* is not a problem. The A* algorithm uses a heuristic distance from each node towards the end to guide the search, we use the straight distance divided by

the highest speed limit found in the network. Some of our algorithms need to find bounds on the arrival time for all nodes in a routing network, for which Dijkstra's algorithm is used because all nodes of the network need to be visited. Both variants are implemented using priority queues as introduced by Fredman and Tarjan [16] and they both have a runtime order of $O(n \log n + m)$ for a graph with n nodes and m edges.

We explain the advantages of the different algorithm types and provide runtime orders in the next few sections. When describing the runtime orders of the algorithms, n and m are used for the number of nodes and edges in a routing network from now on. The runtime for computing a routing network is kept apart from the runtime analysis of the algorithms. The algorithms try to find the traffic condition dependent shortest path through the network. With 'shortest path' we do not mean the path with the lowest travel distance but the fastest path. It is a shortest path problem since we define the problem as finding a path with minimal cost through a graph in which the edge weights are based on travel times.

2.2.1 Fixed path algorithms

Fixed path algorithms compute the shortest path before departure, the routing advice does not change during the trip so these algorithms do not make use of any information that becomes available during the trip. Depending on what information such algorithms base their routing advice on, they can perform quite decently when the traffic conditions evolve slowly or when they evolve in an expected way, but lacking adaptiveness their usefulness for real world situations is limited. Algorithms of this type usually are simple and fast. The runtime is usually dominated by $O(n \log n + m)$ for applying the A* algorithm. An additional component can be the use of traffic data at the moment of departure, which has an order $O(m)$ because traffic information for all edges from the SUMO network are provided. If i historical scenarios are also provided, then an addition component $O(mi \log i)$ is needed for reading the data and sorting the possible travel times of each edge.

2.2.2 Adaptive path algorithms

Adaptive path algorithms are similar to fixed path algorithms except for the fact that they make use of information that becomes available during the trip. When new information becomes available these algorithms update their advice by computing the shortest path again. The usefulness of adaptiveness is already shown by others like Wellman et al. [33] and Fu [17]. Each path computed by an adaptive path algorithm is in itself fixed, the adaptiveness lies in the fact that the algorithm can come up with a different path during the trip when new traffic information becomes available. Some of these algorithms make use of historical information as well in order to predict how the current traffic state evolves. Algorithms of this type are expected to perform better than fixed path algorithms since they receive more information so they have more data to combine with historical data. However, for this algorithm type an on-board routing device is needed while fixed paths can be computed on a desktop computer before departure.

This type of algorithm is also rather fast. Coming up with a shortest path has a complexity of $O(n \log n + m)$ just as for the fixed path algorithms. During the trip the shortest

route from the live vehicle location to the destination is updated each time new information is received, typically once every few minutes should be sufficient. Each live data update received by algorithms of this type has to be imported, which takes $O(m)$. Because the shortest path is updated with each live data update, the update runtime is at least $O(n \log n + m)$ too. When historical scenarios are used there is an additional factor $O(mi \log i)$ before departure for importing and sorting the data. Note that most of the computation time is spent during the trip. Providing a route for the first time before departure takes about as long as for fixed path algorithms.

2.2.3 Policy algorithms

A policy is a mapping from a set of possible states to routing decisions as already introduced by Gao et al. [18]. The general idea of a policy algorithm is to compute a large set of possible states before departure, one routing decision is computed for each state. A state contains a location and time and for some algorithms also traffic information which is explained in more detail in the next two sections. While a trip is being performed only lookups have to be performed to give routing advice. During the trip new information becomes available such as the arrival time at an intermediate location on the route or information about the current traffic conditions. This information forms a state for which the policy algorithm already has computed routing advice. Of course simplifications have to be made since the number of possible states is virtually unlimited. While the mapping itself is fixed the advised route might change during the trip, for example because the arrival time at an intermediate decision node differs from the expected arrival time. Policy algorithms are said to be adaptive but they can only adapt to the possible states for which routing advice was computed before departure. A policy algorithm is fed with historical traffic scenarios such that it can deal with common traffic conditions but it cannot deal very well with irregular conditions, which might occur for example when an accident happens. However, some policy algorithms have access to live data so adapting to changing traffic conditions is possible to some extent.

The size of the policy to compute has a large impact on the runtime and memory usage. A significant reduction in space is already achieved by computing policies for routing networks instead of for original SUMO networks. We expect that computing the policy makes all policy algorithms significantly more time-consuming than fixed path algorithms and adaptive path algorithms. The size of the policy depends on several choices and parameters and is therefore quite different for different policy algorithms. We provide more detailed information when specifying the policy algorithms in Section 3.5. Policy algorithms can be divided further into two main types: one type uses only the location and arrival time in the state representation and the second type also includes traffic information. We discuss these two types separately now.

Policies based on location and time

The first policy algorithm type computes a policy of the form $(n,t) \rightarrow e$, where n is a node, t is a time and e is the outgoing edge from node n at time t belonging to the expected fastest route to the destination. This type only takes the time and location as input each time the

routing device is updated. The idea behind this is that the arrival time at an intermediate location already provides information about the traffic conditions because a policy algorithm deals with one specific trip for a known departure time. The traffic conditions on the network are to quite some extent spatially dependent. During rush hour the amount of traffic increases on a lot of roads at the same time. Besides this correlation, disruptions also propagate through nearby parts of the road network so the spatial dependency is at least partly causal in nature. Suppose a car uses only half an hour to travel the first 60 km of its trip because it was able to maintain a high speed, which means that this part of the road was not congested. On a different day or time this part of the trip might take an hour which means that the road must have been quite congested. Because of spatial dependency the traffic conditions for the remaining part of the trip can be guessed. Although this guess might be quite off it is likely better than guessing randomly or just using an average historical scenario to predict future traffic conditions. By comparing the intermediate arrival time with the expected intermediate arrival time for different historical scenarios, the algorithm can guess which historical traffic conditions are similar to the current conditions.

Policies based on location, time and traffic information

The second policy algorithm type incorporates possible traffic conditions in the policy it produces. The mapping is of the form $(n,t,f) \rightarrow e$, where n, t and e are again a node, time and edge and f contains traffic information. This traffic information can consists of different types and amounts of data for various algorithms. One possibility is to use only a few discrete travel time options per road segment which restricts the number of possible states. Another option is to use only partial information, for example only information about road segments that are close to the vehicle for which the routing advice is computed. So f contains travel time information about a part of the routing network such that policy algorithms of this type have more data to base their advice on.

2.2.4 Summary of advantages and drawbacks

We introduced three types of routing algorithms: fixed path algorithms, adaptive path algorithms and policy algorithms and now we can partially answer our first research subquestion about the strengths and weaknesses of algorithm types. Fixed path algorithms do not use live data which is a drawback for application in the real world. Policy algorithms might be slower than most other algorithms in their initialization but faster in updating their routing advice because decisions for all possible states are computed before departure, their adaptiveness is therefore limited. Adaptive path algorithms are expected to be faster than most policy algorithms and to perform better since they can use live data to its full potential.

Regarding our second research subquestion, we discovered that fixed path algorithms can combine historical data with recent data but not with live data, while the other algorithm types can. Furthermore, when discussing policy algorithms we found that one way of combining historical data with live data is to use the arrival time at intermediate locations of a trip to find similar historical scenarios. We will discuss more ways of combining historical data with recent or live data in Chapter 3.

Chapter 3

Routing algorithms

In this chapter we define all the routing algorithms that are tested, the main types of algorithms are already introduced in Section 2.2. When presenting the algorithms we explain some advantages and drawbacks and new ways of combining historical data with recent or live data are introduced, which helps to further discover the answer to the first two research subquestions.

All our algorithms operate on routing networks that are computed from road networks as explained in Section 2.1.2. Computing a routing network takes $O(k(n \log n + m))$ time for a network with n nodes and m edges. However, when a driver makes the same trip more than once the routing network can be stored after initialization such that it can be used each time the trip is made without computing it again. Therefore we leave out this runtime component in our runtime analyses. When importing historical scenarios as input the different possible travel times per subpath of the routing network are sorted, which together costs $O(mi \log i)$. It is possible to provide a large set of historical scenarios once and then perform a trip multiple times using the same historical information. However, in order to initialize the algorithms within our test framework the data should be imported anyway which already takes $O(im)$. Therefore, we will incorporate a component $O(mi \log i)$ for all algorithms that use historical data but we will not mention this in the run time analysis. The algorithm overview at the end of this chapter takes this component into account.

Suppose for the rest of this chapter that the routing network, which is derived from the original road network, has n nodes and m edges. Suppose further that the travel time of a trip is Δt and the number of historical traffic scenarios equals i. In this chapter we first define algorithm categories and after that we define some commonly used properties. This is followed by an explanation of all our algorithms, accompanied by a brief runtime analysis and the size of the advice space as far as these are not mentioned in Section 2.2. An overview of the algorithms with their properties and runtimes is shown in Section 3.7. More information about implementation details can be found in Appendix A.

3.1 Algorithm categorization

In this chapter twenty routing algorithms are introduced. With such a high number of algorithms it becomes hard to maintain overview and to produce compact comparisons when showing the results of experiments. Therefore we divide the algorithms in three groups and we will mainly compare algorithms with other algorithms of the same group. The distinction is made on the type of traffic data they receive. This makes it easier to say something about how smart algorithms are since we compare them with algorithms that have access to similar information. The groups are defined as follows:

- No live data: algorithms that may receive historical data and arrival times at decision nodes during the trip, but no recent or live traffic data.
- Partial data: algorithms that have access to the same data as the no live data group plus partial live traffic data. Each time a decision node is reached, travel time data is provided only for the subpaths in the routing network that start at that decision node.
- Full data: algorithms that receive the same data as the partial data group except that live travel time data is provided for the complete routing network each time a decision node is reached.

Not all algorithms within a group make use of all the data that is available. There are for example fixed path algorithms in the partial and full data group that make use of recent data before departure but ignore live data updates.

	no live data	partial data	full data
fixed path algorithms	FP	FP-HC', FP-HXC'	FP-C, FP-HC, FP-HXC
adaptive path algorithms		AP-HL', AP-HXL'	AP-L, AP-HL, AP-HXL
policy algorithms	P-HT, P-HIT, P-HI, P-HXI	P-HL'T, P-HL'IT, P-HXL'IT	P-HLIT, P-HXLIT

Table 3.1: Overview showing for all routing algorithms in which category they fall.

In Section 2.2 we defined three main types of algorithms, all our algorithms have one of these main types as their foundation. Table 3.1 shows all the algorithms that are introduced in this chapter such that it becomes clear to which category each algorithm belongs. It shows one empty cell: there are no adaptive path algorithms in the no live data group since such algorithms are not possible without live data according to our definition of adaptive path algorithms. This is true because adaptive path algorithms that do not update their routing advice during the trip are actually fixed path algorithms. The algorithm names are chosen in such a way that one can see from the names to which categories they belong. All fixed path algorithms have 'FP' as prefix, all adaptive path algorithms have prefix 'AP' and the prefix for policy algorithms is 'P'. All algorithms in the partial data and full data group have either

the letter C or L in their name. For the partial data group this letter is always followed by an apostrophe. In the next section we will explain the meaning of all the letters used in the algorithm names.

3.2 Notation of common properties

Because certain properties are used by different algorithms in different combinations, we will first explain these properties and give them an abbreviation letter such that we can use short notations for our algorithms. A main distinction between algorithms is the kind of data they use. All algorithms except the fixed path algorithms receive the intermediate arrival time when the vehicle reaches a decision point at which routing advice is requested. Any additional data they receive can be distinguished in the following types:

- C: current data, which is the most recent available travel time information before departure. Algorithms that make use of this data type receive the most recent information just before the vehicle trip starts.
- L: live data, travel time information that is updated from time to time during a trip. Algorithms that have this property receive travel time information both before departure and during the trip so this data is a superset of the data available for algorithms with property C.
- H: historical data, travel time information about several days from the past. Historical information is used by some algorithms to predict how the traffic will evolve during a trip. This can be combined with current or live data.

Another major distinction that can be made is whether algorithms receive traffic data for the complete routing network or only for road segments close to the current location of the vehicle that is being advised. This distinction will be made by adding an apostrophe to property C or L when the provided data is partial. Partial information means that when routing advice is computed for a certain decision node in the routing network, travel time information is provided for all subpaths starting from this decision node. Subpaths typically have a length between a few hundred meters and several kilometers. The idea behind this kind of partial data is that the travel time of a certain road segment is known quite accurately at the moment one enters it, an assumption also used by Fu [17]. The advantage is that the amount of data is restricted well since the number of paths exiting from a decision node is bounded by the maximum degree of network nodes. The number of roads leaving from any junction is in practice hardly ever higher than four.

Algorithms that receive a lot of data can use this data in several ways, which can be expressed as combinations of certain properties that we describe below. Properties T and I are mainly relevant for policy algorithms, therefore these two letters are only used in the names of policy algorithms. Property X is used by some algorithms of each algorithm type.

- T: round times to discrete time intervals during the computational process, we use time intervals of five minutes. For example, when in a set of historical scenarios a

certain intermediate location is reached always between 9:01 and 9:13 for a certain trip departure time then a routing decision for this node is computed only for the times 9:00, 9:05, 9:10 and 9:15. When a large historical data set is available, rounding times can greatly reduce the amount of calculations performed since further computations make use of results computed for smaller parts of the routing network. Decision times are rounded to data points lying five minutes from each other, the routing advice for each time interval is determined by majority vote. By aggregating to a limited set of time points the advice space is smoothed a bit so the advice is expected to vary less between similar arrival times at any intermediate location.

- I: treat travel times of different historical scenarios independently. Suppose ten scenarios are provided, then there are ten possible travel times per subpath in the routing network. Applying this property means that for each state travel times from only one scenario are used. This greatly reduces the number of possible combinations, since the number of combinations then equals the number of instances provided. This property thus assumes a strong correlation between travel times of links. Without applying this assumption, all possible combinations of travel times are used for every subpath of the network so for any intermediate location the path with the lowest average expected travel time to the end would be advised. Therefore, not using this property means that the average best path is advised, which is still time-dependent though.

- X: several algorithms that receive historical data compare it to live traffic data in order to find the most similar historical scenario. When a lot of historical scenarios are provided, applying this property means that instead of finding the single most similar historical scenario the few most similar scenarios are found and aggregated. Up to 13 historical scenarios it only considers the single most similar scenario. From 14 to 34 scenarios the three most similar scenarios are used, from 35 to 69 scenarios the five most similar ones are used and for 70 or more scenarios the seven most similar ones. This heuristic is chosen somewhat arbitrary, the borders between different numbers of historical scenarios to use are put at multiples of seven because the amount of provided historical data can then be expressed in weeks. Only odd numbers of decisions are used in order to increase the probability of a single winner when majority voting is applied. Applying this property makes the advice space smoother and it can be regarded as an attempt to prevent overfitting. Since policy algorithms compute a policy before departure the aggregation of the few most similar scenarios is done by a majority vote over the best routing decision for each of the scenarios. For other algorithms the aggregation is done by averaging the travel times of the most similar scenarios and then the best decision is computed for the conditions of this averaged scenario.

3.3 Fixed path algorithms

We have several algorithm versions that compute a fixed path, we discuss them below one by one. The first one has full knowledge since it receives the exact testing conditions as

input such that it always gives the real shortest path. The second and third one are good algorithms to compare the more complicated algorithms with. The last four are variations of a smarter algorithm. Since all algorithms of this type return a single path as result, the size of the result is bounded by $O(n)$ since one subpath to choose has to be stored per decision node on the advised path. Applying the A* algorithm accounts for a large part of the runtime which has order $O(n \log n + m)$. The abbreviated names of these algorithms start with 'FP' which stands for fixed path. The names are followed by the abbreviation letters of the used properties as explained in Section 3.2.

3.3.1 The optimal path

A special algorithm is the algorithm that has full knowledge about the traffic state, it receives the exact conditions on which it is tested and therefore the routing advice of this algorithm is guaranteed to be optimal within our testing framework. Exactly knowing the future is of course nothing less but cheating, but the result of this algorithm is used to define a scale on which to measure the performance of other routing algorithms as well defined values. Therefore, the runtime or memory usage of this algorithm is not interesting, this algorithm is not part of the twenty algorithms that we test. We will refer to its result as the optimal path or the optimal travel time.

3.3.2 FP

A very basic algorithm for computing the shortest path is to ignore all information except for the road network topology. It uses an A* algorithm to compute the shortest path between the start and destination. The travel time of each road segment is computed by dividing the distance in meters by the maximum allowed speed in meters per second. The advantage of this algorithm is that it is very simple and fast and it does not need any information apart from the network topology and speed constraints. Applying this algorithm can be compared to a person that makes a trip by computing the fastest route on his computer and then performs the trip without a navigation device and without taking any available traffic information into account. This algorithm serves mainly as a basic algorithm to compare the performance of other algorithms to. When tested on an empty road network the result is the absolute minimal travel time under the assumption that the speed limits are not exceeded, which provides insight in the effect of the traffic conditions on trip durations.

3.3.3 FP-C

Naturally the next step of improvement is to compute routing advice based on the traffic state at the moment of departure, thus using property C. Such algorithms do exist already and are to some extent available to the public. The advantage of this algorithm is that the computed expected travel time is more realistic and the route advice takes current traffic disturbances into account. However, it does not take into account how the traffic conditions evolve. Apart from applying the A* algorithm an additional $O(m)$ is required for reading traffic speed data for the whole routing network. The results of this algorithm can be quite decent, especially for short trips because the traffic conditions do not change that much

within a short time span. This is a good algorithm to compare more advanced algorithms to.

3.3.4 FP-HC

The next step is an algorithm that tries to think ahead about how the traffic conditions evolve. Algorithm FP-HC receives a set of historical traffic scenarios as input as well as the traffic conditions on the moment of departure, thus combining properties H and C. It is not trivial how to combine recent data with the historical scenarios to predict how the traffic will evolve. We know that if the data for the moment of departure is accurate then this information is the only relevant information for the start of the trip. The further the routing algorithm looks into the future when planning the trip, the more important the historical information becomes since the data for the moment of departure becomes less relevant. Therefore, this algorithm uses a heuristic when computing the route advice: it interpolates between the traffic conditions at the moment of departure and the most similar historical scenario. The heuristic prescribes that the contribution of the most similar historical scenario grows linearly from zero to one hundred percent over a certain time span, we chose for an interpolation time span of fifteen minutes. From some tests with different parameter values we found that on rapidly changing traffic conditions a time span of only a few minutes is best, while on other test cases a time span of an hour ore more is better. An optimal value is hard to define since it depends quite heavily on the chosen set of test scenarios. The most similar historical scenario is defined as the scenario with the lowest sum of squared errors over the difference in travel time per subpath of the routing network. More formally, suppose the routing network consists of a set of subpaths P, the average vehicle speed of a certain path p for departure time t is v_p^t according to the live data and $v_p^{x,t}$ according to historical scenario x. Finding the most similar historical scenario is done in the following way:

$$\arg\min_x \left(\sum_{p \in P} (v_p^t - v_p^{x,t})^2 \right) \tag{3.1}$$

The runtime for computing which historical scenario is most similar to the current data is bounded by $O(im)$ since i is the number of scenarios to consider and m is the total number of edges where the subpaths are made of.

We can also explain the heuristic for interpolating between recent and historical data in a more formal way. Suppose the A* algorithm is applied from a certain node s at time t_s and it is visiting node v at time t_v and it then updates the time at which neighbour node w can be reached. The latest live data known at time t_s prescribes a travel time for the path from v to w, say Δt_1. The algorithm also finds the latest time interval $t_i < t_v$ for which the most similar historical traffic scenario prescribes a travel time, say Δt_2. Assuming the time parameters are in seconds it then computes a factor $f = \max(1, (t_v - t_s)/z)$, where z is the interpolation time span in seconds. The travel time from node v to w is then computed as $(1-f)\Delta t_1 + f\Delta t_2$.

3.3.5 FP-HC', FP-HXC and FP-HXC'

We implemented three algorithms that are variants of FP-HC. Algorithm FP-HC' receives only travel time data for the subpaths in the routing network that start at the location of departure. When computing the shortest path, this algorithm uses the live data for these close-by subpaths and for other network parts it uses the travel times from the most similar historical scenario. Therefore, no interpolation is needed between live data and historical data. Comparing the recent data with historical scenarios only costs $O(i)$ because the amount of current data is bounded by a constant, since the number of outgoing paths from the start node is in practice bounded by a constant value. A disadvantage of this version is that it might perform worse in finding a similar scenario because there is very little traffic data to compare the historical scenarios to.

Algorithm FP-HXC receives recent traffic data for the whole network, just like FP-HC, but adds property X. This means that when a significant amount of historical scenarios is provided, it will base its advice on the few most similar scenarios instead of only on the single most similar one. Finally, FP-HXC' combines the two previous algorithms, it receives only partial data and applies property X.

3.4 Adaptive path algorithms

We implemented several adaptive path algorithms, one using only live data and a few variants that try to combine live data with historical data in order to predict the future traffic conditions. All algorithms of this type return a single path as result so the size of the result is bounded by $O(n)$. The runtime is already explained in Section 2.2.2. Algorithms of this type are abbreviated with 'AP' for adaptive path, followed by the abbreviation letters of the used properties.

3.4.1 AP-L

The simplest adaptive shortest path algorithm is very similar to the fixed path algorithm that relies on the traffic conditions at the time of departure. It also computes the shortest path based on the latest traffic information only. However, during the trip updated information of the traffic conditions become available. The algorithm then updates the shortest path by computing the shortest path again based on the new information. This algorithm can quickly adapt to changing traffic conditions when the update interval is small. However, it does not think ahead: when a common traffic jam starts in morning's rush hour then the algorithm does not foresee that the jam will keep growing. The time interval between live data updates that we use in our experiments is five minutes.

3.4.2 AP-HL

A more sophisticated version of an adaptive algorithm thinks ahead about how the traffic conditions evolve. A set of historical traffic conditions is given as input, combining the latest live data with the most similar historical scenario is done in a way very similar to the

method described for algorithm FP-HC. The further the routing algorithm looks ahead when planning the remaining part of the trip, the heavier the decisions will depend on historical data. The major difference with the fixed path variant is that this algorithm receives live data during the trip. With each live data update the algorithm recomputes which historical scenario is most similar to the live data and then it computes the shortest path again by applying the A* algorithm.

3.4.3 AP-HL', AP-HXL and AP-HXL'

We created three algorithms that are very similar to AP-HL. Algorithm AP-HL' receives only partial travel time data with each live data update. It is the live data variant of algorithm FP-HC' and similarly it does not need to interpolate between historical and live data because the live data is only given for subpaths that are just being reached. Therefore, finding the most similar scenario for each query has order $O(i)$ instead of $O(im)$. Algorithm AP-HXL receives network-wide live data and uses the few most similar historical scenarios to predict the future traffic conditions. AP-HXL' combines the previous two algorithms, it uses both partial data and property X.

3.5 Policy algorithms

All policy algorithms compute a mapping from possible states to routing decisions before departure. The number of possible states largely depends on the amount of information represented and the way in which this information is aggregated. The idea is that several historical scenarios are provided to a policy algorithm. This results in a list of possible travel times for each road segment. The question is which states to build from this information for which routing advice should be computed. The various policy algorithms use their input data differently, some common options are defined in Section 3.2.

A subroutine is made that is used by a lot of the policy algorithms. It uses Dijkstra's algorithm to find the arrival time at each decision node of a routing network given a departure time and a set of travel times for the subpaths. The subroutine can be performed using the travel times of a certain traffic scenario or for the lowest or highest travel time per edge from the set of all historical scenarios. When using the lowest time per edge it effectively computes a lower bound on the arrival time at each decision location. When using the highest travel time per edge it computes an upper bound under the assumption that detours are not allowed. The runtime of this subroutine is $O(n \log n + m)$. In the remainder of this chapter we will refer to this algorithm as the 'dijkstraBound' algorithm.

We will now explain the different policy algorithms, they all start with abbreviation 'P' for policy. The runtime analysis excludes the time for computing and importing the routing network and sorting the possible travel times per subpath but including the runtime for the dijkstraBound routine whenever it is used. The size of the policy is also provided for each policy algorithm separately.

3.5.1 P-HT

A policy algorithm that only applies property T apart from using historical data like all policy algorithms. First it computes a lower and upper bound for the time at which each decision node can be reached by calling dijkstraBound twice. After this is done the network is traversed from back to front, computing routing advice for each decision node for each time interval of five minutes between the computed time bounds which results in a policy of size $O(\Delta tn)$. Each decision is the outgoing subpath belonging to the total remaining path for which the weighted expected travel time is lowest. By traversing it backwards, computed decisions can be used again when computing the optimal route for nodes earlier in the network in a dynamic programming way. Because loops may occur in the routing network this traversal might have to be performed a few times, but the number of repeats is limited to a constant to limit the runtime. Assumption I is not included in this algorithm so the advised route is the route with the lowest travel time when averaging over all historical input scenarios. The advice is time dependent so the advised route still depends on the departure time and the arrival time at intermediate locations.

The runtime order for computing lower and upper bounds for possible arrival times at decision nodes is $O(n\log n+m)$. Computing routing advice per decision node per time interval means that for each node-time pair the expected travel time via each outgoing path is computed as the average travel time of that path plus the earlier computed expected time to the end from the node this path ends in, together this costs $O(\Delta t(n+im))$. The total run time of the algorithm is bounded by $O(n\log n+\Delta t(n+im))$.

3.5.2 P-HIT

Algorithm P-HIT also uses property T so departure times are rounded to intervals of five minutes. It also assumes property I so travel times from different historical scenarios are not combined. For each historical scenario there is only one possible travel time per subpath and only the shortest path to each network node is considered, there is therefore only one possible arrival time per node thus subroutine dijkstraBound is performed only once for each historical scenario. For each scenario one decision is computed per node by traversing the routing network from back to front one or a few times in the same way as described for P-HT. When this step is finished, multiple decisions might be computed from different scenarios for the same time stamp for certain nodes. Therefore the results are aggregated to a single routing decision per node per time interval by using majority voting. The size of the computed policy is therefore bounded by $O(\Delta tn)$.

Running dijkstraBound for each historical scenario separately has a runtime order of $O(i(n\log n+m))$. Computing a routing decision per scenario per decision node costs $O(i(n+m))$. For aggregating the results at the end each node-time pair has to be visited and the number of decisions to aggregate is maximally i. Aggregating multiple decisions means that all decisions are visited once to count the number of occurrences of each advised subpath and then the subpath with the most votes wins. Both these steps are $O(i)$ so the total runtime for the aggregation is $O(i\Delta tn)$. The total runtime is therefore bounded by $O(i(n\log n+\Delta tn+m))$.

3.5.3 P-HI

An algorithm similar to P-HIT is P-HI, the latter one does not use time roundings during the whole process. Again subroutine dijkstraBound is performed once for each historical scenario, followed by computing one decision per node. Because no time roundings are involved, it is very likely that the arrival time at any node is different for each historical scenario. Since no aggregation takes place, the number of decisions stored per node exactly equals the number of provided scenarios, the size of the resulting policy is thus $O(in)$. When more and more scenarios are provided the policy becomes larger than for algorithm P-HIT. Each time a routing advice has to be provided to the driver the arrival time at the current decision location is given to this algorithm. It then returns the already computed decision for this node that has the closest-by time stamp. The policy algorithms that we already introduced all perform a fast lookup whenever a routing decision has to be communicated. This version however performs a small computation for each lookup. The advantage is that arrival times do not have to be rounded during the computation and no aggregation by majority voting has to be applied which makes the algorithm likely a bit faster. Rounding times is not really necessary for reducing the number of computations because applying property I already reduces the policy size significantly.

Similar to P-HIT, applying dijkstraBound for each instance has a runtime order of $O(i(n\log n+m))$ and computing all decisions takes $O(i(n+m))$. The total runtime for computing the policy is $O(i(n\log n+m))$. For each query to this algorithm the provided time is compared to the time stamp of i decisions for the provided current location so each query costs $O(i)$.

3.5.4 P-HXI

This policy algorithm is equal to P-HI except for the computation performed for each query. P-HI always returns the decision with the time stamp closest to the query time. However, it is likely that some decisions computed from quite different traffic scenarios have close-by time stamps for certain nodes so the routing advice could vary a lot within a small arrival time span. In order make the advice space smoother and more robust this algorithm version applies property X.

The amount of work performed for each routing query is higher than for P-HI but it still has the same runtime order. This is true because the number of best decisions to consider is limited to a constant. Finding a set of decisions with the closest-by time stamps is done by visiting each possible decision once, checking whether its time stamp belongs to the set of closest by ones has a run time order $O(1)$. When this set is computed the number of occurrences of each advised path is counted which is limited by $O(i)$. Then the subpath with the largest count is found and returned which also takes $O(i)$. Therefore, the runtime for both creating the policy and for returning advice on queries is the same as for P-HI.

3.5.5 P-HL'T

This algorithm supposes that with a routing advice query for a certain location and time a discrete travel time level is given for all subpaths that start from the query location. The

algorithm runs routine dijkstraBound twice to find the time bounds for the arrival time of all decision nodes. Then it traverses the network backwards, computing a routing advice for each decision node for each relevant time interval for each possible combination of traffic levels of the subpaths starting in that node. The number of outgoing paths from any decision node is limited to the number of roads exiting the physical junction it represents. Because this number is in practice limited to four or five we can safely assume that the total number of combinations is $O(1)$ although the number of combinations equals the number of discrete traffic levels to the power of the number of outgoing paths. To limit the number of possible states for which a decision has to be computed the number of discrete traffic levels is set to three which gives a coarse distribution of possible travel times per subpath in the routing network. Given a set of historical scenarios the travel time for any subpath can be computed for each scenario, resulting in a set of possible travel times. Three discrete travel times are computed for each subpath by averaging the longest one-third of travel times, the middle one-third of travel times and the lowest one-third of travel times. The size of the policy then becomes $O(\Delta tn)$. When computing a routing advice the travel times from the most similar historical scenario are used for all subpaths for which no traffic information is given. Since this algorithm makes use of coarsely discretized partial live data the performance of the algorithm might still be a bit poor because of the simplifications and the possibly large runtime.

Storing which subpaths are outgoing from each node is already done on creation of the routing network. The runtime for computing arrival time bounds for the nodes is $O(n \log n + m)$. Computing the policy takes $O(\Delta t(n+m))$. Since the number of combinations for discrete traffic levels of close-by paths is bounded by a constant, the query time is also $O(1)$ since a constant amount of live data has to be compared with a constant amount of states.

3.5.6 P-HLIT

Algorithm version P-HLIT combines several properties. It runs dijkstraBound twice for each historical scenario to find a lower and upper bound for the arrival time at each node. Then a routing decision is computed per node per time interval per scenario. During a trip the algorithm receives full information about the current traffic conditions each five minutes. Each time a decision node is reached it uses the most recently received data to determine which historical scenario is most similar. Similarity is again measured by computing the sum of squared differences between the average subpath speeds from the live data and the speed data from each historical scenario. The live data is not used to compute the travel times of subpaths directly but only to look up the routing advice computed for a similar historical scenario. This means that the policy size and runtime are hardly influenced by the amount of information in live data updates, the size of the policy is $O(i\Delta tn)$. When the most similar scenario is found the decision already computed for this scenario is returned for the location and time of the query.

Running dijkstraBound twice has a run time bound of $O(n \log n + m)$. Computing a decision per interval-node-time tuple takes $O(i\Delta t(n+m)$. The combined runtime complexity is $O(n \log n + i\Delta t(n+m)$. With each live data update, the most similar historical scenario has to be found again. Computing the similarity error for each scenario by visiting each

edge for which the live data provides a travel time costs $O(im)$ time, which is also a bound for the time needed to retrieve the minimal error. Each time a query is made the algorithm checks whether the live data has been updated since the previous query. If this is the case it first computes the most similar scenario again. So we can say that the runtime for a query is bounded by $O(im)$.

3.5.7 P-HL'IT, P-HXLIT and P-HXL'IT

Similarly as for FP-HC and AP-HL we introduce three variants of P-HLIT. Algorithm P-HL'IT is the partial data variant of P-HLIT, the amount of live data it receives with each update is of constant order size, which keeps the query time low. The algorithm compares the live data with all historical scenarios to find the most similar one but because the live data is $O(1)$ in size the comparison and thus the query time is bounded by $O(i)$ instead of $O(im)$. P-HXLIT receives data for the complete network and applies property X. By using majority voting over the few most similar historical scenarios the advice space becomes smoother, we expect its routing advice to be a bit better than the advice given by P-HLIT. Algorithm P-HXL'IT combines receiving partial live data with applying property X, so it also has a query time order of $O(i)$.

3.6 Guidelines for including travel time variance

In Section 1.3.4 we explained that we do not model travel time variance directly, our algorithms optimize the expected arrival time instead. We do not include any algorithms that take variance into account because this changes the definition of the best path and then we cannot compare the advice quality of all the algorithms so easily. We discuss now how variance could be combined with our algorithms. A first line of thought is to look at algorithms that use property X. These algorithms look at a set of similar historical scenarios that prescribe a time-dependent average speed per road segment. Some of these algorithms average over the different possible speeds per road segment, this is the case for FP-HXC, AP-HXL, P-HXI and the partial data variants of the first two. One could think of defining a normal distribution based on the computed possible travel times and then for example pick the 90th percentile travel time instead of the mean. Then there is a 90% probability that the travel time for a subpath of the network is smaller than the time considered by the algorithm. In principle, redefining edge weights to take variance into account could be combined with all our algorithms. However, the algorithms that use property X only consider a handful of historical scenarios, which is likely not enough for extracting realistic travel time distributions for all the roads.

However, a problem arises if we want to include variance in this way since the travel time distribution of any road is time-dependent. Because the arrival time of an intermediate location in the routing network is not known in prior we do not know what the travel time distribution will be for the roads after the current subpath because this depends on the time of arrival at the next decision location. If we incorporate variance by applying a somewhat pessimistic heuristic for the travel time of a subpath in the network then the error between the real and estimated arrival time at the end of the subpath is on average worse than when

the expected travel time would be used. This could propagate to a larger error in the guessed travel time distributions for planning the remainder of the route. When using a heuristic like this it is far from trivial to find out how large the probability of arriving before a certain deadline is for any complete path from start to destination. This problem is due to the edge weights of graphs being both uncertain and time-dependent.

Another way of taking variance into account is by looking at the mean and variance of travel times of complete paths. Suppose that a set of potentially optimal paths is identified first. Given a set of (similar) historical scenarios, an algorithm could compute the travel time for each complete path separately for each scenario. Based on these results the mean and variance of each path can be computed. Which path is optimal then depends on how the mean and variance are weighted in the optimization function or whether a certain arrival time deadline is to be met. The described procedure is quite different from all our algorithms but it could be added to our framework easily since it already support storing multiple historical scenarios and computing the travel time of paths per scenario separately.

3.7 Algorithms overview

We introduced a lot of different algorithms with different properties, therefore we now give an overview of the algorithms and their advice space and runtime complexities. Table 3.2 shows the algorithm properties, the runtime is split in the time needed for calculations before departure and the calculation time for each query. To make the comparison easier the runtime orders are simplified by replacing each m by n. This simplification is valid when m is $O(n)$ which is the case because in road networks the average number of edges connected to a junction is definitely not higher than four so the difference between n and m is bounded by a constant factor. The time for building the routing network is excluded since it is a necessary initialization step for all our algorithms.

algorithm	advice space	initialize time	query time
FP	n	ω	-
FP-C	n	ω	-
FP-HC	n	$\omega+\phi$	-
AP-L	n	ω	ω
AP-HL', AP-HXL'	n	$\omega+\phi$	$\omega+i$
AP-HL	n	$\omega+\phi$	$\omega+in$
P-HT	Δtn	$\omega+\phi+i\Delta tn$	1
P-HIT	Δtn	$i\omega+\phi+i\Delta tn$	1
P-HI, P-HXI	in	$i\omega+\phi$	i
P-HL'T	Δtn	$\omega+\phi+\Delta tn$	1
P-HL'IT, P-HXL'IT	$i\Delta tn$	$\omega+\phi+i\Delta tn$	i
P-HLIT, P-HXLIT	$i\Delta tn$	$\omega+\phi+i\Delta tn$	in

Table 3.2: Overview of the routing advice space and run time orders of the routing algorithms, where $\omega = n\log n$ and $\phi = ni\log i$.

The advice space order for all fixed path algorithms is $O(n)$ because these algorithms only compute a single path as routing advice. Adaptive paths compute the shortest path again from time to time but they only have to store the latest computed path. Policy algorithms have a larger advice space since they compute a policy containing a lot of possible states.

All algorithms make use of a Dijkstra or A* algorithm, either for finding paths through the routing network or for computing bounds on the possible arrival time at decision nodes which is reflected in the overview by component $\omega = n \log n$. Furthermore, most algorithms have a run time component $\phi = ni \log i$ for reading the travel times of historical scenarios and sorting them per subpath. The query runtime is not relevant for the fixed path algorithms since they do not update their routing advice during the trip. Under the assumption that a lookup from a hashmap data structure can be done in constant time, some policy algorithms have a query runtime order of $O(1)$ since they only have to perform a lookup. We see that the more sophisticated algorithms have a higher query time complexity. Adaptive path algorithms have a component ω since they compute the shortest path with each live data update.

In this chapter we discussed a lot of algorithms and saw some of their advantages and drawbacks, which further clears the way for answering our first research subquestion. The results of the experiments will further answer which algorithms perform well under what conditions. We partly answered the second research subquestion by introducing new ways of combining historical data with recent or live data. Historical scenarios similar to the current situation can be found using the arrival time at intermediate locations or by comparing historical data with recent or live traffic speed data available for the whole network or around the current location only. Live and historical speed data can also be combined to predict future conditions by linear interpolation over time such that the contribution of live data decreases when planning further into the future.

Chapter 4

Traffic simulation

In this chapter we explain why simulations are useful and how we use it for our purposes, it aims at answering our fourth research subquestion. This is followed by a comparison between a few different traffic simulators which results in the choice for traffic simulator SUMO. We then go into detail about how real data is obtained and used by SUMO. The parameters for the simulation and their chosen values are explained. After this we will elaborate on the various problems encountered regarding the realism of simulations and whether we can apply corrections to solve them.

4.1 The need for simulations

A practical reason for using simulations is that they can provide complete data. In Section 4.3 we explain that we have access to loop detector data, which only provides information about specific locations in the road network. The limited character of real data is also acknowledged by Chrobok et al. [11] who also propose to use simulations because they can produce network-wide information. Using simulations to produce network-wide traffic data is the major reason why simulations are useful to us, which is a first step to answering our fourth research subquestion.

A second reason that is closely related to the first one is that simulations can be used to extrapolate limited data in a realistic and meaningful way since reality can be mimicked instead of using some heuristics to fill in the gaps. As long as the traffic is not very dense, heuristics for guessing the average speed on locations for which no data is available might perform quite well. However, situations with more dense traffic are more interesting because then good routing advice can really make a difference. Capturing the dynamics of traffic is harder to do for more dense traffic when not using microscopic simulations. Suppose for example that a certain detector measures very low vehicle speeds and the first detector upstream reads normal vehicle speeds. If these detectors are a few kilometers away from each other then a heuristic guess for the length of the traffic jam can be quite off. Duncan and Littlejohn [15] also take a stand for using microscopic traffic simulation for traffic prediction because microscopic simulations can cope with the more complex ways in which traffic behaves.

Thirdly, one can use simulations directly to predict future traffic states. In order to do this, the inflow of new vehicles in the future must be guessed. Using heuristics to predict the future based on historical traffic patterns is an alternative to simulating the future directly. Suppose a traffic accident occurs that blocks one lane for twenty minutes. Because accidents are usually quite unique in location, time, severity and duration, the impact on the traffic conditions is hard to predict with heuristics. Using a simulator, such an accident can be simulated to find out the impact on the future traffic conditions.

There are some drawbacks to using traffic simulations. Using simulations to predict the future from the latest live data requires that the simulation runs in multiple real-time, which is hard to achieve for very large road networks. Furthermore it is not easy to run a simulation that starts with the latest live data as starting condition since the current occupancy of a road network cannot be obtained directly from the loop detector data we have access to. Mainly for these reasons the implemented routing algorithms will not use simulations directly to produce routing advice, simulations will only be used to produce scenarios with network-wide data from the limited real data. Another disadvantage of using traffic simulations is that the vehicle parameters and interaction are by default the same for each simulation. In reality the desired speed and headway of vehicles might depend on the weather conditions. During heavy rainfall vehicles tend to drive slower and keep a larger headway which is not incorporated in our simulations.

Apart from the advantage of being able to simulate future traffic conditions, simulations are not necessary when network-wide real data is available. Floating car data, which is data sent by in-car devices, has the potential of delivering network-wide data. When a significant part of the vehicles in a road network continuously send their location and speed to a server then the average speed can be computed quite accurately for the whole network. For road segments where no or few vehicles are driving the maximum allowed speed could be used. However, since we do not have access to floating car data we use simulations to create network-wide data that can be used to test routing algorithms on.

4.2 The choice for SUMO

SUMO, which stands for 'Simulation of urban mobility', is a microscopic simulator which means that it simulates individual vehicles. It is an open source program written mainly in C++. It contains a lot of functionality regarding importing and altering networks, creating vehicle flows and simulating traffic scenarios. Although a lot of functionality is already implemented the development continues, via SUMO's website a lot of documentation can be found [30]. Of course SUMO is not the only open source traffic simulator, other options are also available. A few traffic simulation software programs are listed below accompanied by a short summary. These programs are the open source simulators that seem to be promising on first sight. The only exception is VISSIM, which is a well known commercial software package but it is included just for comparison.

- VISSIM: Verkehr in Städten Simulationsmodell, simulates as realistic as possible with all types of road users and with good visualization that is even in 3D. Because of

this high realism and nice visualization only very small networks can be simulated. Another disadvantage is that it only runs on the Windows operation system.

- SUMO: Simulation of urban mobility, simulates vehicles as individual agents with their own route and properties. Some simplifications are applied for faster simulation but vehicle interaction still has a decent level of realism. Large networks can be simulated but when simulating complete countries it becomes too slow.
- MATSim: Multi-agent transport simulation, models traffic based on a population of citizens. Each citizen has a daily plan which includes taking part in traffic. For simulating traffic a queue model is used for roads for faster simulation which reduces the vehicle realism somewhat.
- Transims: Transportation analysis and simulation system, also models traffic based on a human population. Uses a cellular road model which makes fast simulations possible but reduces vehicle realism quite a lot. Visualization is available only after a simulation is finished for which a very basic visualization tool is available. There is a better external visualization tool but this is commercial software.

The short summary of the simulations already mentions some advantages and drawbacks of the different programs. Table 4.1 shows the advantages and disadvantages of these programs in more detail by comparing them on specific properties.

	VISSIM	SUMO	MATSim	Transims
accessibility	commercial	open source	open source	open source
operating systems	win	win/lin/mac	win/lin/mac	win/lin
documentation	medium	good	good	medium
community & support	good	good	medium	good
main language	C++	C++	Java	C++
space domain	continuous	continuous	continuous	discrete
simulation speed	low	medium	high	high
traffic realism	high	medium	medium	low
graphical editor	yes	yes	yes	yes
import from OSM	?	yes	yes	?
visualization level	high	low	low	low
public transport	yes	yes	yes	no
pedestrians	yes	no	no	no
video recording	yes	no	yes	yes

Table 4.1: A comparison between a few micro simulators. Operating systems are abbreviations of Microsoft Windows, Linux and Mac OS. OSM is an abbreviation of OpenStreetMap. Some values are guessed or unknown to us.

Each of these programs has its advantages and drawbacks. For this thesis project it is a requirement to use open source simulation software so VISSIM is not an option at all.

Besides that its simulations would be too slow for our purposes. Transims also dropped out quickly, mainly because of the fact that the discretization of the spatial dimension reduces the realism a lot. The two main candidates therefore are SUMO and MATSim. After a more thorough investigation and installing and trying them out, SUMO became the final choice because of the amount of documentation and support as well as the many different functionalities it has from which it becomes clear that it can be used for several purposes and working methods. MATSim seems to promise a faster simulation speed but we found it harder to work with and it focusses more on simulation of urban areas and thus it seemed to be less lenient for other working methods. It advertises about being very modular so that one can replace modules easily by own implementations but SUMO's functionalities are also divided into clearly separated modules.

4.3 Obtaining real data

We will discuss now how we obtain road networks and traffic information from real data and how this can be used by SUMO to produce useful output.

4.3.1 Road networks

There are several ways to obtain a road network in SUMO. Simple networks can be defined by hand but this option is not sufficient since we also want to work with larger and real world scenarios. We use the functionality of SUMO to import networks from OpenStreetMap (OSM) data which is open source data for road networks all over the world. One major advantage of using OSM is that this data is freely available to anyone so the data itself is kept up to date by the public. A downside is that the data is not guaranteed to be complete and sometimes it is incorrect. Which roads there are, where they are situated and how they are connected to each other is usually represented correctly in OSM data. The most important additional information for routing is usually present like the maximum allowed driving speed and the number of lanes. The number of lanes however is sometimes wrong, either because someone simply filled in the wrong value or because there is only one OSM element for a road segment in which the number of lanes changes within the segment. The latter cause does not occur a lot because the number of lanes usually changes near a junction and junctions are always locations at which road segments are split up. Despite the small flaws OSM data can be used to route on quite well.

4.3.2 Traffic data

In order to create traffic simulations we need to compute traffic flows from real data input. One of the options is to use data from loop detectors which are sensors in the road that measure the number of vehicles and their speed per time interval. For the Dutch road network this data is available from NDW, which stands for 'Nationale Databank Verkeersgegevens' which roughly translates to national database of road traffic data. Although receiving data for whole time periods is not free of charge, live data can be received for free since September 2013. The data is aggregated over time intervals of one minute and consists of the

average vehicle speed and the number of vehicles passing which is also called vehicle flow, both are usually measured per lane. One also has the option to receive data that is aggregated over larger time intervals. Throughout this project we use time intervals of five minutes for all flow and speed data of the roads in the networks. By using intervals of five minutes instead of one minute the amount of data is reduced significantly without losing much accuracy. Furthermore, the flow and speed at road segments develops a bit smoother over time in this case.

The data from loop detectors gives exact information but only for specific points in the road network. The coverage on the Dutch highway network is quite good, there are detectors close to most on-ramps and off-ramps so one can often guess the number of vehicles entering and leaving the highway. There are hardly any loop detectors on other road types so if we want to simulate traffic on smaller roads, for example within a city, then we hardly have any data to rely on. There are other sources for real data apart from loop detectors. Floating car data is a promising data source since it gives speed data which is the kind of data needed for producing routing advice. Therefore, floating car data is potentially a better data source, provided that a lot of data is available and that a significant part of all vehicles in the networks of interest contribute. However, these conditions are not generally satisfied. Because loop detector data from NDW was made available for this project and because the functionality for using this data is present in SUMO, we choose to focus on loop detector data.

4.3.3 Creating traffic from detector data

When a road network has been imported and the NDW loop detector data is available the question is how to generate traffic on the network based on this data. SUMO has a module for creating traffic from loop detector input called DFRouter. The module works best when there are loop detectors at all major entry and exit locations in the road network. First it assigns types to the given detectors, the possible types are ‘source’, ‘sink’ or ‘between’. The idea is basically to search upstream from each detector location for other detectors. If no upstream detectors are found, then a detector becomes a source detector. Similarly, detectors for which no downstream detectors can be found become sink detectors. The remaining detectors are then the in between detectors. The module then creates routes starting at source detectors. All route possibilities are explored, routes end when a sink detector is encountered or when a dead end of a road network is reached.

When working with a network in which detectors are present at all roads following each junction, SUMO can compute a correct route probability distribution. Suppose for example that we have a highway with an off-ramp that measures 100 vehicles in an hour and a detector downstream the highway that measures 400 vehicles in the same hour then the route probability distribution will be such that 20% of the vehicles in the simulation will take the off-ramp. A basic example is shown in Figure 4.1, where there is one source detector and three sink detectors. The generated vehicles are divided over the three possible routes according to the vehicle counts at the sink detectors. The number of vehicles reaching each sink detector corresponds very well with the input measurements of these detectors which shows that DFRouter is able to assign the right number of vehicles to each route. Small

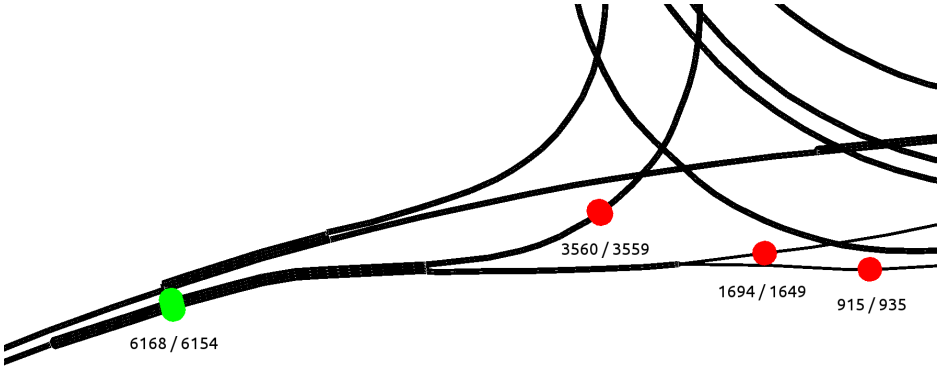


Figure 4.1: A small example network with four loop detectors. The green dot on the left represents a source detector, the three red dots represent sink detectors. Below each detector two numbers are shown, the left one is the number of vehicles passing in a time span of three hours according to NDW data, the other value is the number of vehicles passing in a SUMO simulation based on this data.

errors can still occur, for example because not all vehicles reach a sink detector before the simulation has ended.

With full coverage of loop detectors DFRouter can create vehicle flows that are quite realistic in numbers. However, there are hardly any detectors in the Netherlands on roads other than highways. If we import a road network that includes a lot of minor roads that do not lie on a fast route between a source and sink detector then no traffic will be generated for populating these roads. Consider for example the network in Figure 4.2 where a part of the detectors lying on the highways are used. Most urban roads are filtered out because most of them will not see any traffic when creating a simulation based on data from the given detectors. Because loop detectors are almost exclusively situated on highways we will focus on highway networks. Validation of the vehicle counts on a highway network will be discussed in Section 4.6.

4.3.4 Generating output from traffic simulations

SUMO simulations are performed based on a file that contains all vehicles with their routes which results from a vehicle creation process using SUMO's module DFRouter. This module takes the network as input as well as a file with definitions of loop detectors and a file with the real data readings for these detectors. From this it produces a file with vehicles, each vehicle receives a route and departure time. Running a simulation means that all these vehicle trips are executed from which the traffic in the road network emerges. Because vehicles have to adjust their speed to the other traffic no information is stored about the travel time prior to the simulation because the travel time is not known before executing the simulation.

There are several options for computing output during a SUMO simulation. The option we are interested in is that SUMO can create an output file which contains the average speed

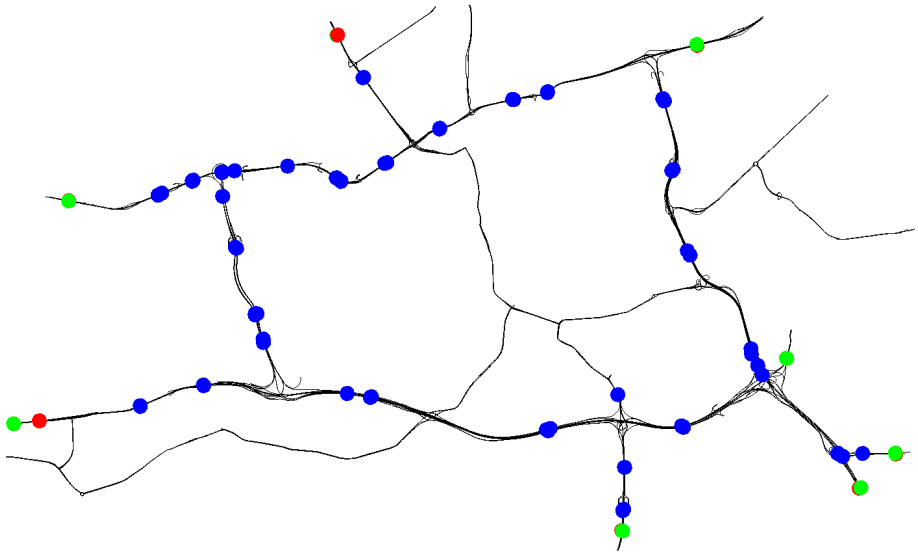


Figure 4.2: Visualization of a network in SUMO. This is the area of Rotterdam, the smallest minor roads are filtered out. The coloured dots are loop detector locations, green ones are sources, red ones sinks and the blue dots are in between detectors. At some locations a source and sink detector are so close to each other that the dots representing them almost fully overlap.

per road segment per time interval. For road segments at which no vehicles pass during a certain time interval the maximum allowed speed can be written by default. The advantage of such an output file is that it can be used quite well to perform routing on the network. While loop detector data only provides information for a limited set of locations in the network the output file of the simulation covers the whole network.

4.4 Simulation parameters

In the previous sections we explained why we use traffic simulations and that we choose to work with SUMO. SUMO has of course made a model for traffic simulation in which simplifications are made. Furthermore, there are several aspects that might undermine the level of realism of simulations. This chapter deals with the parameter choices that are made in order to make the simulations as realistic as possible. After this some encountered problems are explained together with their solution or workaround if available. Since we focus on parts of the Dutch road network we base our parameter choices on statistics available for the Netherlands as much as possible.

When we have a road network imported we also want to have vehicles driving around on it. In order to produce realistic traffic, it is important that the vehicles have realistic parameters defining their behaviour and that they interact with each other in a natural way. The vehicle interaction and the behavioural parameters are discussed now.

4.4.1 Vehicle interaction

The interaction between the vehicles in a SUMO simulation result mainly from implemented rules for the car-following model and the lane changing model. The exact workings of these algorithms are not easy to grasp from the large amounts of code SUMO is made from. The car-following model dictates how a car adapts its speed depending on the distance to the car in front of it. SUMO uses the Krauß model by default as introduced by Krauß [24]. Its parameters are the desired speed, the reaction time and the acceleration and deceleration capacities. It is a safe distance based model which means that a vehicle should always be able to prevent collision with a leading vehicle. Development is going on in order to improve the car-following model, we will stick to this default model. The lane change model has some heuristic rules that define when a vehicle should change lanes. For each vehicle the best lane choice is stored for the current and next road segment on its route. Which lane is best is vehicle dependent since it depends on which way it continues its route at a nearby road split. It also depends on the traffic density in front of the vehicle for the different lanes.

There are some shortcomings in the interactions between vehicles. SUMO is designed to be collision free which means that any vehicle should always be able to avoid a collision with the cars in front of it by braking as fast as possible. This sounds very nice but it is actually not realistic since it is quite common for people to drive too close behind a leading car, especially on highways. From Belgian statistics [29] we learn that about 40% of the cars and 20% of the trucks do not keep a safe two seconds distance when driving on highways. We will overcome the unrealistic simplification made by SUMO by increasing the deceleration ability of vehicles and slightly decreasing the minimal time gap between vehicles such that they keep more realistic distances.

4.4.2 Vehicle parameters

An important aspect for the realism of the simulation is how the parameters of individual vehicles are defined. Below is a short overview of important vehicle parameters, their names are copied from the SUMO documentation and that is why mostly self-explanatory names are used instead of symbols. For the last three parameters a constant value is used for all vehicles.

- length: the length of a vehicle in meters.
- maxSpeed: the maximum speed that a vehicle can physically drive in m/s. Can also be used for setting a global maximum speed, for example for trucks.
- speedFactor: the desired speed of a vehicle is drawn from a normal distribution. This parameter sets the center of this distribution as a factor of the maximum allowed speed.
- speedDev: the desired speed of a vehicle is drawn from a normal distribution. This parameter sets the deviation of this distribution.

- accel: the acceleration of a vehicle in m/s^2. This value is constant for each vehicle and is independent of its speed. Therefore it is best to interpret this parameter as the average acceleration.
- decel: the deceleration of a vehicle in m/s^2 that is applied under all circumstances so it can be considered as the average deceleration.
- impatience: the willingness of a vehicle to impede vehicles with higher priority, the parameter value must be between 0 and 1. When a vehicle does not have priority and wants to enter a road, value 0 means that it waits for a gap large enough so that other vehicles do not have to brake for it while a value of 1 means that it joins the road as soon as it does not lead to a collision although an approaching vehicle might have to hit the brakes. The impatience of a vehicle grows linearly to value 1 when its waiting time grows to five minutes.
- minGap: the minimal distance in meters that a vehicle keeps between itself and the vehicle in front of it. This is mainly interesting for obtaining a correct vehicle density in traffic jams.
- tau: the minimal time gap between a vehicle and the vehicle in front of it. This is also related to the reaction time of vehicles. Reactions of vehicles are currently limited to the discrete time steps of the simulation.
- sigma: a measure for driver imperfection. It is hard to find in the documentation what this parameter actually does. We know that a value of 0 means that a vehicle applies exactly the acceleration and speed that fit the situation it is in. For a value larger than zero a vehicle might accelerate slower than its parameter or drive a little bit below its desired speed.

Because vehicles tend to have different desired speeds in reality, vehicles of one type can be created with their desired speed drawn from a normal distribution for which the parameters ‘speedFactor’ and ‘speedDev’ are used. The length of vehicles together with ‘minGap’ is important because it defines how many space vehicles need. The width of vehicles is irrelevant for the simulation since this is ignored by the simulation for simplification. Now that we explained important vehicle parameters we provide the values of important parameters for average cars and trucks using Table 4.2.

The length of vehicles can be found rather straight forward by looking at regular cars and trucks used in the Netherlands. The value for the absolute maximum speed that any vehicle may have in simulations is set somewhat arbitrarily to 50 m/s for cars, which equals 180 km/h. For traffic speed parameters we found useful information from Belgian statistics. Belgium is quite similar to the Netherlands regarding speed limits and driving behaviour. Therefore, we will use some Belgian statistics because the Dutch counterparts were not found in all cases. All Belgian statistics we refer to are from a statistics report of the year 2011 [29]. The average speed on Belgian highways is 117.9 km/h for a limit of 120 km/h which means that the relative speed factor is about 0.96. For roads with limits below 90 km/h the average driving speed is around the limit or even a little higher, but in our test

	car	truck
length	4.2	18.0
maxSpeed	50	24
speedFactor	0.96	0.94
speedDev	0.11	0.034
accel	1.5	0.5
decel	1.8	1.3
impatience	0	0.3
minGap	1.5	1.5
tau	0.9	0.9
sigma	0.5	0.5

Table 4.2: Realistic vehicle parameters on which the vehicle types that are used in SUMO simulations are based.

scenarios we focus on highways mostly. For the Belgian highways a standard deviation of about 13 km/h is realistic which is computed from the known 85th and 95th percentiles of the measured speed data, assuming a normal distribution. This comes down to a standard deviation of just below 0.11 relative to the maximum allowed speed of 120 km/h. The average speed of trucks on Belgian highways is 89.2 km/h for a limit of 90 km/h. In the Netherlands, the maximum allowed speed for trucks is 80 km/h. Therefore we set a global maximum speed for trucks to 24 m/s which is about 86 km/h. Together with a speed factor of 0.94 we ensure that trucks drive around 80 km/h on highways and on average a little slower than cars on other roads. Because of speed limiters and the fact that the allowed speed is lower the variance in the speed of trucks is lower than for regular cars. From the Belgian statistics a standard deviation around 3.1 km/h is derived which is about 0.034 relative to the Belgian limit of 90 km/h.

The acceleration is taken constant by SUMO, so it does not depend on the actual speed or the vehicle's maximum achievable speed. This simplification is a bit rough but diving into SUMO to change this is not within the scope of this project. A normal average car acceleration lies around 1.5 m/s^2, the literature agrees quite well on this, see for example the work of Baerwald [2] and Akçelik and Besley [1]. For acceleration of trucks realistic values are a bit harder to find. The maximum acceleration depends on the load of the truck and usually lies in the range 0.5 to 1 m/s^2 according to Woodrow and Poplin [34]. The acceleration depends quite heavily on the amount of load a truck is carrying since a fully loaded truck can be several times heavier than an empty one.

For deceleration a lot of different values can be found in literature ranging from 0.28 to 4.9 m/s^2 according to the overview given by Maurya and Bokare [25]. The authors come up with mean values around 0.5 m/s^2 for trucks and 1.2 m/s^2 for cars. The Traffic Engineering Handbook from Baerwald [2] gives a deceleration of 3 m/s^2. A value of 1.8 m/s^2 is found by Akçelik and Besley [1]. Apparently different braking behaviour can be applied by drivers in different situations. The literature mostly agrees that drivers coming from a higher speed brake more firmly, Akçelik and Besley [1] and Bennett [4] suggest that drivers do not adjust

their braking distance enough to their speed, thereby accepting a higher deceleration when approaching with a higher speed. From experience large differences in braking behaviour are to be expected: when driving towards traffic lights drivers usually brake firmly compared to the small and smooth adjustments that vehicles make a lot when adjusting to vehicles in front of them. In SUMO vehicles brake with a deceleration that is independent of their speed and they start braking as late as possible such that no collisions occur. Therefore we make a compromise between different braking behaviours. This work focuses on traffic on highways, so vehicles mainly make small speed adjustments, they rarely go from high to very low speeds. Therefore, a good average deceleration guess is 1.8 m/s^2 for cars. For trucks we found very little information, we give them a bit lower deceleration than for cars.

Another value that might need explanation is the impatience. For regular cars the default value of 0 is sufficient, since they usually respect the priority of other cars and wait until there is a gap large enough to insert without significantly disturbing the speed of other vehicles. Buses and especially trucks are large and slow and they might have to wait relatively long when they want to enter a road without disturbing other vehicles. In practice, they therefore enter the road also when an upcoming vehicle has to brake a little because of it, they expect other drivers to be lenient to give trucks possibilities to enter the road. Therefore a non-zero impatience is chosen for trucks, the value 0.3 is somewhat arbitrary. Finally, we choose SUMO's default value for the vehicle imperfection sigma since we do not have sufficient information about this parameter to tweak it and as long as vehicles are controlled by humans some driver imperfection is realistic. Running extensive tests to optimize all parameters is not the main goal of this project.

4.4.3 Vehicle distributions

We want the traffic in simulations to be as realistic as possible. One aspect of real traffic is the apparent randomness in vehicle density throughout a road. Varying densities are a result of vehicles having different desired speeds, accelerations and decelerations. Trucks can have a large impact on the traffic since their speed and acceleration are significantly lower, the speed difference is especially large on highways. The parameters of SUMO already enable us to define a normal distribution from which the desired speed of each vehicle is drawn. However, all other parameters are the same for all vehicles within one vehicle type. In order to obtain more realistic traffic, vehicles should apply different acceleration and deceleration behaviour as well. Therefore we define several vehicle types with slightly different parameter values. The parameter values as explained in Section 4.4.2 serve as average values from which good guesses for the parameters of the different types are made. The exact choices of different vehicle types and parameter value differences between them are mostly guessed since no data was found on this topic for Dutch traffic. Next to that there are a lot of vehicle brands with hundreds of different vehicles types that all have different parameters and modelling this exactly would make the model unnecessary complex.

The values for parameters minGap, tau and sigma are equal to the values given in Table 4.2 for all vehicle types. Most parameter values in Table 4.3 are already explained in Section 4.4.2 but a few parameter distribution choices are explained in more detail now.

	cars						trucks		
type	tiny	short	reg.	sw	sport	dv	bus	heavy	light
length	3.6	3.9	4.3	4.7	4.4	5.2	13.0	18.0	18.0
maxSpeed	50	50	50	50	50	50	30	23	25
speedFactor	0.96	0.96	0.96	0.96	1.10	0.94	0.94	0.94	0.96
speedDev	0.11	0.11	0.11	0.11	0.22	0.11	0.07	0.03	0.05
accel	1.2	1.4	1.6	1.9	2.8	1.4	0.9	0.4	0.8
decel	2.8	3.0	3.1	3.3	4.2	2.8	2.3	1.9	2.3
impatience	0	0	0	0	0.1	0	0.2	0.3	0.3
probability	0.18	0.22	0.26	0.16	0.04	0.14	0.10	0.60	0.30

Table 4.3: The vehicle parameters for all vehicle types that are used in our SUMO simulations. The vehicle type abbreviations are from left to right: regular, station wagon and delivery van.

Touring cars and buses have a global speed limit of 100 km/h in the Netherlands and therefore their maximum speed is set to 30 m/s which equals 108 km/h. Because of their speed factor their average is very close to the allowed speed. The sports type represents drivers with good cars who like to drive firmly. Their average speed and speed deviation is higher as well as the acceleration and deceleration, the exact values are somewhat arbitrary. Sports cars are added because there are usually a few vehicles on the road that drive way too fast and because their behaviour can be significantly different from other traffic they might have a relatively large impact on the traffic. The impatience factors are as proposed in Table 4.2 except for a non-zero value for sports cars. Their drivers might be a bit impatient and egoistic sometimes and therefore they receive a non-zero impatience.

One might notice that the deceleration values for the vehicle types are significantly higher than the values proposed in Table 4.2. When applying the suggested low deceleration SUMO is sometimes not able to insert as many vehicles as measured in reality at a loop detector. A simulation log file showed that thousands of vehicles can be waiting for insertion when simulating a scenario with a time span of a few hours. In SUMO vehicles keep such a distance that they always can avoid collisions, we already briefly explained this problem in Section 4.4.1. When the deceleration value is low the minimal distance that vehicles keep on a highway can become so large that the number of vehicles SUMO can safely insert within a certain time interval is lower than the number of vehicles passing in reality. We showed earlier that in reality a lot of vehicles do not keep a safe distance. Collisions might be rare still because drivers are usually able to look ahead further than only the vehicle directly in front of them and it does not happen a lot that a vehicle has to hit the brakes maximally for several seconds. In order to achieve more realistic headways we choose a higher deceleration for the vehicles in our simulations. This ensures that the number of vehicles measured by detectors can be inserted correctly in a simulation because the distances that vehicles keep are more realistic. The deceleration of regular cars is chosen around 3 m/s^2 which is above average but still within the braking capacity of regular cars.

The probabilities at the bottom of Table 4.3 define a probability ratio of occurrence

for the different types of cars and trucks. The probabilities sum up to 1 for all cars and for all trucks separately. The probability for each vehicle to be of super type car or truck can be chosen separately for each simulation. According to Dutch statistics for 2012 [5] all vehicles together travelled 127 billion km while the contribution of buses and trucks is together 7.6 billion km. Based on this ratio we choose a 6% probability for any vehicle in a simulation to be a truck or bus. Buses contribute for just over 0.6 billion km which is more than 8% of 7.6 billion km, we round this value upwards to 10%. The other 90% is taken by heavy and light trucks, the distribution between these two types is a guess. For the normal car types delivery vans contribute for 17 billion km, which is about 14% of the total 119.4 billion vehicle kilometers travelled by all vehicle types excluding trucks and buses. The probability for the other car types is somewhat arbitrary since the division in car types itself is made up in order to get more complex and diverse traffic.

4.4.4 Time step size

SUMO simulations advance in discrete time steps. The length of each time step is constant throughout a simulation but any value can be chosen for it. The default time step size in SUMO is one second. The value is a compromise between realism and simulation speed. Shorter time steps means that vehicles behave more realistically but it makes the simulation slower. Time steps of one second are a good choice since seemingly realistic traffic behaviour can then be observed and stop-and-go waves can also occur.

In order to ensure that SUMO can insert a high number of vehicles on a road the deceleration values of cars are chosen on the high side already. Furthermore, the time gap parameter tau is set to 0.9 to further decrease the minimal distances between vehicles. Since a value for tau lower than the simulation step size is not supported and since SUMO only inserts vehicles at time steps and not in between, we choose to use time steps of half a second.

4.5 Traffic problems

Earlier we described how to generate traffic on a network and we explained the choices we made for the simulation parameters. Several problems can occur with the produced traffic which can have a great influence on the level of realism. Problems that deal with the number of vehicles passing through the network will be the topic of Section 4.6. In this section we will discuss a few other problems that mainly have to do with vehicle interaction. First we will look at some problems caused by errors in the network topology and after that we will discover some problems that are caused by unrealistic vehicle behaviour.

- A major network-related problem is that the number of lanes is not always correct. We can not blame SUMO for this since the errors already occur in the OSM data from which the network is constructed. The number of lanes of highways can be either too high or too low. The number of lanes being too low is likely the more unconvenient variant since a highway section can then become a bottleneck from which traffic jams start while in reality there might be enough lanes for the traffic to flow through

normally. Other problems like incorrect maximum speeds can also occur but are less common and likely have a smaller impact.

- Highway ramps are usually tagged as highways in the OSM data but they can also be stored with a lower road type. When filtering a network such that only highways remain we throw away some of the ramps. Another problem that occurs when working with highway networks is that an off-ramp and on-ramp are sometimes connected such that the entry and exit points are lost. In reality certain junctions are both the end of an off-ramp and the start of an on-ramp. When filtering smaller roads away the ramps are usually preserved but the road it was connected to is removed and the effect is that the off-ramp and on-ramp are then connected by the junction. Topologically seen it then becomes a tiny detour to the highway from which no cars can leave or enter the highway. It is hard to automatically distinguish such merged ramps from the small road sections used for visiting motels or petrol stations along highways.

- A problem with a very large impact is that vehicle flows can block at certain junctions when the vehicle rules at the junction are not properly defined by SUMO. Although a junction might not be blocked, vehicle flows from different directions can be standing still while waiting for nothing. This problem is observed at some signalized intersections in the center of the Rotterdam network from Figure 4.2. It is not easy to grasp why this behaviour occurs at some junctions and creating an automatic procedure to fix this is not relevant for our work. Therefore we visually checked the few signalized junctions in our test networks and manually fixed the connections and vehicle priorities in such a way that traffic flows do not block at these junctions.

- Another junction-related problem is that the traffic light programs at signalized intersections are sometimes quite unrealistic. SUMO can automatically merge intersections that are very close to each other such that a large junction is also represented as a single junction in SUMO. However, it can happen that intersecting vehicle flows receive green light at the same time or more connections might face a red light than necessary. We manually defined a traffic program for the few signalized junctions that occur in our networks.

- There is another problem related to vehicle interaction: it happens regularly in our simulations that vehicles on a highway on-ramp come to a halt when the traffic density on the highway is considerable. In reality vehicles on the highway often change lanes or increase their headway in order to give room for vehicles from an on-ramp. Vehicles on on-ramps usually adjust their speed to the vehicles on the highway. They manoeuvre themselves to gaps and accept small gaps for insertion, accepting unsafe distances for a few moments. After the insertion the distances between the vehicles smoothly increases to normal levels. SUMO however fails at this point because of its collision-free model. When the gaps between vehicles on a highway are not large enough for safe insertion then the vehicles on the on-ramp will decelerate and finally come to a halt at the end of the on-ramp. Merging with the highway traffic becomes even harder when the speed difference increases. As an effect short jams occur at

on-ramps regularly so smooth merging of vehicle flows while maintaining speed is often not achieved in simulations.

- Another type of problem is the lane usage which is defined by the lane changing model within SUMO. The current lane changing model is able to let vehicles drive on the right-most lane most of the time when possible which is realistic behaviour on the Dutch road network. However, it still happens too often that trucks are driving at the left lane of a two-lane highway, especially when the traffic is a bit denser. Compared to the other problems we just mentioned this problem has a less severe effect on the realism of the simulation. It can cause the average speed to be on the low side because slow vehicles are sometimes slowing down other traffic more than in the real world.

- Another major problem is that loop detectors are not available for each road between any pair of junctions. Because of this SUMO does not know how many vehicles should continue in each direction at certain junctions. Errors in the number of vehicles taking a certain road from a junction can have the effect that a large part of the network is used by way too many or way too few vehicles. Errors can stack up and greatly reduce the realism of simulations. So even if 90% of the roads are covered, the missing parts can cause significant errors in the number of vehicles per route. When there is no detector on an off-ramp SUMO will send half of all the vehicles from the highway to this off-ramp which is quite unrealistic in general. Because of the gravity of this problem we will deal with it extensively in the next section.

4.6 Corrections on generated traffic

In Section 4.5 we discussed some problems that arise when simulating traffic in SUMO. The focus of this part of the chapter is to improve the realism of the vehicle flows in the network. We first recall shortly why our focus is on highway networks and then we will address problem related to wrongly assigned loop detector types and missing detector data. After that will introduce a correction algorithm that solves the problem partially.

4.6.1 Focus on highway networks

As already mentioned in Section 4.3.3 traffic is only generated on routes starting at loop detectors. Because there are hardly any detectors on roads other than highways, little or no traffic will be generated that uses minor roads. Because of a lack of information when using NDW detector data it is not possible to create traffic on minor roads that is guaranteed to reach a certain level of realism. A workaround for this problem is to focus on highway networks only and leave out the smaller roads. Because solving this problem is not the main purpose of this project our test scenarios include mainly parts of highway networks. However, even if we only take highways into account the number of vehicles can be off quite a lot. We will discuss how we deal with this problem in Section 4.6.4.

4.6.2 Correction on loop detector types

SUMO automatically computes the type of each loop detector in the network. It does so based on the network topology. A detector becomes a vehicle source when there are no detectors in front of it. However, this does not always result in the correct type being assigned. Suppose that we have a highway starting at the west boundary of a network, then the first detector on the direction eastwards should become a source detector. Suppose that vehicles driving from east to west have the option to take an off-ramp and enter the highway in the other direction and then pass our detector. In this case, a detector located at the other driving direction might prevent our detector from becoming a source detector. A similar problem can occur for detectors that should become sinks.

It is not easy to automatically fix wrongly assigned detector types. Since only a few detectors will receive the wrong type in our test scenarios we check and correct the detector types manually for our test scenarios.

4.6.3 Missing loop detector data

Another problem related to loop detectors is that detectors are sometimes out or order or they provide wrong data because of malfunctions. Sometimes a loop detector does not read data for a period of time, the number of vehicles it returns for such a time span then equals zero. A failing detector can have a severe impact on the realism of traffic simulations. When the first detector after a split point in the network reads zero vehicles then all vehicles passing that split point will receive a route leading via the other road that exits from the junction. A coarse solution to this problem is applied automatically by our framework in which the simulations are prepared and run: detectors that read zero vehicles throughout the whole time span of a simulation scenario are removed from the simulation before the traffic is generated. Usually a detector further down the highway is available such that removing a detector does not have a large impact on the scenario. There is however a small chance that a source detector is removed. In such a case SUMO can automatically define a close by detector as source but this will not always happen, the reasons are explained before in Section 4.6.2. So in a rare case a part of the traffic might be missing in a scenario.

A hard problem to tackle is when a detector sends wrong data for a short time span that is smaller that the total simulation time. It is difficult to automatically detect whether a loop detector provides correct information since it might happen that no vehicles are passing a detector for a few minutes which can be caused by traffic fluctuations or because a road is temporarily blocked because of an accident. Most detectors are functioning normally for most of the time but this problem still occurs quite a lot. When we requested loop detector data the NDW tool already showed that a lot of detectors are not providing data a few percent of the time. We were indeed confronted a few times with unrealistic irregularities in vehicle flow graphs showing NDW data of a single detector. It is hard to apply correction to detector data when one is not completely certain about the correctness of any data part and since making simulations as realistic as possible is not the main goal of our work we do not put further effort in creating correction algorithms.

4.6.4 Vehicle flow correction

Loop detectors are usually located close to on-ramps or off-ramps but since they usually do not measure on the ramps themselves we do not know directly how many vehicles enter or exit the highway at these ramps. If there is an off-ramp without detector SUMO assumes by default that 50% of all vehicles will take the off-ramp. Because of this the sum of measured vehicles at the loop detectors was way too low in an early stage of our experiments because a lot of vehicles left the highway prematurely. In order to prevent SUMO from assigning a lot of vehicles to off-ramps without detectors we started with the not so elegant solution of disallowing all traffic at these ramps. Our framework algorithm looks at split points in the highway network and detects whether one of the two continuing roads is a dead end, it then adds that off-ramp to the list of forbidden roads that will be given to SUMO. In the Rotterdam test scenario that we will introduce in Section 5.2.2 we manually added more complex highway off-ramps at the boundary of the network to the list of forbidden roads. After this correction most vehicles stay on the highway and end their route at a highway sink detector. Since on-ramps and off-ramps usually come in pairs the number of vehicles on the on-ramp and off-ramp usually cancel each other out for a major part. Therefore the error decreases for most cases when forcing all vehicles to stay on the highway. After this basic solution is applied we can introduce a solution to automatically fix the number of vehicles entering and leaving the highway at ramps.

We introduce an algorithm for adding fake detectors at on-ramps and off-ramps and attaching realistic vehicle measurements to them. With this new information SUMO can compute the percentage of cars that leave or enter the highway at a certain ramp. When performing a simulation SUMO can output validation data for the detectors it received as input. The number of vehicles passing each detector in the simulation can then be compared to the vehicle counts that acted as input data. A small example is shown in Table 4.4. The error is stored as the input values minus the simulation values.

time interval	1	2	3	4	5	6	7	8
NDW measurements	100	110	123	106	111	132	122	109
SUMO measurements	90	100	115	98	105	132	125	110
error	10	10	8	8	6	0	-3	-1

Table 4.4: An example showing how the vehicle flow error at a detector is computed.

For each detector we compute the error between the input measurements and the simulation measurements for the number of vehicles per time interval. Note that in general there can be zero, one or multiple on-ramps or off-ramps between two detectors so we cannot assume that at least one on-ramp and off-ramp are present such that one could be used to correct all positive errors and the other for fixing all negative errors. In order to make sure that we do not add or remove too many extra vehicles we use a carry when computing the number of vehicles that should exit or enter the highway at a ramp. A simple example is shown in Table 4.5. The number of vehicles to add at time interval i is computed by subtracting the upstream error from the downstream error (because the difference in errors is

caused by errors made in between those two detectors) and then the carry from interval $i-1$ is added. If the result is negative then the number of cars to add becomes 0 and the negative result is stored in the carry for interval i. In the example there are 9 cars too few at the upstream detector and 10 cars too few at the downstream detector during the first time interval which means that the shortage of cars is increased by one. One vehicle can be added via an on-ramp to fix this, if such a ramp is present. In the second and third time interval the error is three vehicles lower at the downstream detector which carries over to time interval four and five. In these time intervals the error increases with 4 vehicles so the carry goes to zero and one vehicle can be added again. In a similar way the list for the removal of vehicles via a possible off-ramp can be computed by subtracting the downstream error from the upstream error and again negative values result in a carry that propagates to later time intervals.

time interval	1	2	3	4	5	6	7	8
error downstream	10	10	8	8	6	0	-3	-1
error upstream	9	11	10	6	4	-2	-1	-1
adding carry	0	-1	-3	-1	0	0	-2	-2
add vehicles	1	0	0	0	1	2	0	0
removal carry	-1	0	0	-2	-4	-6	-4	-4
remove vehicles	0	0	2	0	0	0	0	0

Table 4.5: An example showing how many vehicles could be added via an on-ramp or removed via an off-ramp given the errors of two surrounding detectors.

An algorithm is implemented that adds fake detectors at on-ramps and off-ramps based on the method we just described. The algorithm starts an investigation by looking for each highway on-ramp and off-ramp whether a fake detector should be added. It only considers off-ramps with a dead end and on-ramps with a dead start so no junctions may occur anywhere on the ramp and furthermore no detectors may be already present at these ramps. This choice is made to ensure that only ramps are included on which there is no traffic yet. The algorithm searches for a detector both upstream and downstream of each location where a ramp connects to a highway. If no detector is present between the ramp location and an upstream or downstream junction then no fake detector is added because no realistic correction can be applied when there are multiple ramps or junctions without detectors in between. Based on the error information of the upstream and downstream detector vehicle addition and removal lists are created as explained above and are given as measurements to the fake detectors.

To show that this algorithm works we will now look at a small example network of a highway with one on-ramp and one off-ramp as shown in Figure 4.3. Vehicles start at the left side of the network at a source detector. After the off-ramp they pass an in between detector and after the on-ramp the vehicles leave the network at a sink detector. This stretch of highway is about 1.5 km long. NDW measurements from the 21th of October 2013, from 7 AM to 11 AM are used as input for the detectors. The number of vehicles is aggregated per five minutes. The number of vehicles created by SUMO at the source detector exactly

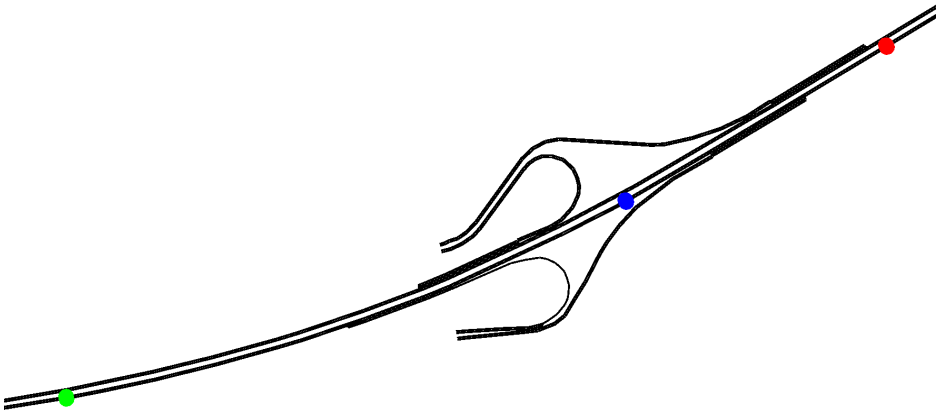


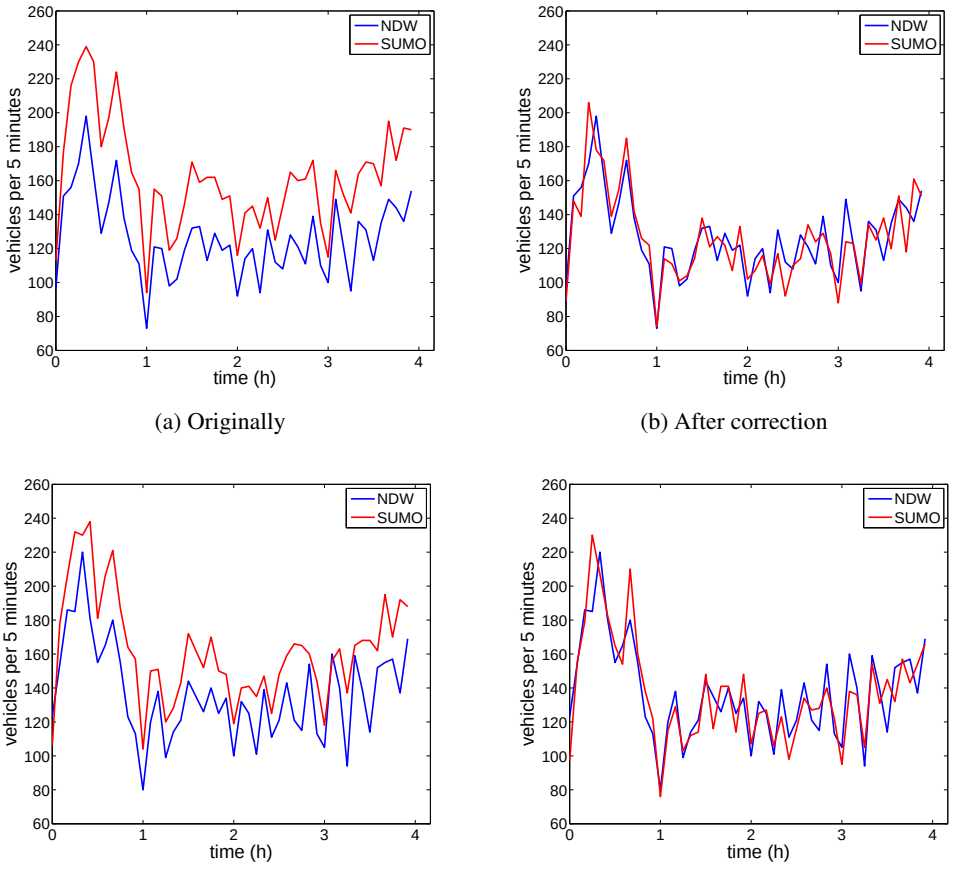
Figure 4.3: A small highway network with three loop detectors, it is a part of the Dutch highway A67 near Geldrop. The green dot at the left represents a source detector, the blue dot halfway is an in between detector and the red dot at the right represents a sink detector.

mimics the input data, 7,830 cars pass there. In SUMO almost the same number of cars pass the two downstream detectors when we disallow traffic on the ramps because all cars drive the same route. Only the cars added close before the end of the time span might not have enough time to pass all three detectors. According to the real data the number of cars passing the middle detector after the off-ramp is only 6,076 and the number of cars at the sink detector is 6,572. We can deduct that about 1,750 vehicles left the highway at the off-ramp and about 500 vehicles entered the highway at the on-ramp during that morning. Figure 4.4 visualizes the errors at the halfway detector and the sink detector. The graphs also show the number of vehicles passing in the simulation after our correction algorithm is applied. The graphs clearly show that the correction helped a lot, the total number of vehicles passing these two detectors is only 16 respectively 6 after correction is applied.

This algorithm only fixes the local errors, once a large error occurs in the number of vehicles driving on a certain highway the algorithm will not fix this. Suppose that at a certain highway section one thousand vehicles too few enter during the time span of one hour. The described algorithm tries to fix the vehicle flows at the ramps at this section such that the number of cars leaving the section is also one thousand too low. Therefore, errors caused for other reasons can still have a global impact.

4.7 Realism of the simulations

We discussed several traffic problems and some corrections and manual fixes. As already explained in Section 4.5 we cannot fix all problems and together these problems make the traffic simulations quite unrealistic. In Chapter 5 we introduce some test scenarios, including the Rotterdam test scenario that uses both a real road network and real traffic data. For this test we created validation graphs for several scenarios similar to the ones of Figure 4.4. The number of vehicles passing certain detectors can be several times too high or too



(a) Originally

(b) After correction

(c) Originally

(d) After correction

Figure 4.4: The number of vehicles according to the NDW input data and the number of vehicles passing in the simulation for the network shown in Figure 4.3. The top two graphs show the error for the in between detector and the bottom two are for the sink detector.

low over a time span of a few hours. It even happens that in reality thousands of vehicle pass a certain detector while zero vehicles pass the same location in the simulation. For the Rotterdam scenario we looked at seven different week days of seven different weeks and computed the average number of vehicles read at all detectors together. On average about 739,000 vehicles were measured by the NDW detectors while only about 159,000 vehicle-detector passings are measured in the simulations. The average speed in the simulations is with 90.2 km/h quite close to the average of 92.8 km/h from the NDW data. These validation results indicate that the scenarios are not realistic enough to copy conclusions of our experiments directly to conclusions about their usefulness in the real world. However, we will be able to show important proofs of concept for our routing algorithms.

This answers our fourth research subquestion completely: simulations are useful for producing network-wide traffic data from limited real data, but they are clearly less realistic than we hoped for so they are of very limited use.

Chapter 5

Experimental setup

The goal of this chapter is to present a test framework for evaluating the performance of the routing algorithms introduced in Chapter 3. We want an answer to our first research subquestion: which algorithm is best overall and which algorithms perform well under what circumstances. This is strongly related to the third subquestion, that wonders how the performance of the algorithms depends on the type and amount of available data. Our experiments are designed in such a way that these questions can likely be answered.

5.1 Goals

The main aspect of algorithm performance we are interested in is the quality of the routing advice that algorithms compute. Next to this there are other aspects to take into consideration. The different criteria used to evaluate the performance of the routing algorithms are explained in more detail now.

5.1.1 Quality of the route advice

Quality of routing advice can be interpreted in several ways, it is not trivial what the best criterion is. In the first place the fastest route might not always be the best one, depending on the preferences of the driver. For now we will focus on minimizing the travel time. After a test vehicle has made a trip based on the advice of a routing algorithm, we want to say something about the optimality of the driven path.

The optimality of a driven path is of course relative, therefore we should compare it to the travel times for other paths. One way of defining quality would be to order all possible paths and rank them from fastest to slowest. The score of a driven path is then simply the rank of the path so the optimal routing algorithm then is the one that results in the lowest average rank of the paths resulting from its advice. However, this solution could be too coarse. Consider for example a situation in which there are three possible paths, resulting in a travel time of respectively 30, 31 and 60 minutes. The difference between choosing the best or the second best path is almost negligible but taking the slowest path is a much worse decision. The travel times could also have been 30, 59 and 60 minutes, then the second best

path can also be considered a bad choice. Therefore, we have to incorporate the travel time of the different paths in our quality testing.

A much better solution is to compare the travel time that results from following the advice of a routing algorithm to the travel time of the optimal path. Computing the optimal path is rather easy, we do this by computing a fixed shortest path on the exact test conditions of the scenario, as already mentioned in Section 3.3.1. Since the optimal path serves as a lower bound, we can express the quality of other paths by dividing their travel times by the optimal time. The travel time of a path then can be said to have a certain relative delay percentage. The quality of a routing algorithm can be measured as the average delay percentage over a test set. A similar way of measuring is to sum the absolute delay of all the test scenarios, which is also a nice way of comparing the performance of the routing algorithms. When discussing the results of the experiments we will use the average delay percentage as the main criterion for expressing the quality of the routing advice.

5.1.2 Predicted arrival time

Having an algorithm that provides good routing advice is one thing. It is also useful for the user to be informed about the expected arrival time for a trip. When routing algorithms compute a shortest path or when an expected shortest path can be derived from a policy, then algorithms also have an indication of the travel time for the shortest path. A good prediction of the travel time makes a routing device more reliable so the user will be more inclined to follow its advice. Next to this a comparison between the real and the guessed travel time also provides an indication of how well an algorithm interprets a traffic situation. Apart from the routing advice it gives, the extent to which the interpretation is accurate is a measure for the extent to which available information is used and interpreted in the right way.

Each algorithm computes the expected arrival time by computing how long the trip would take when following the routing advice they computed themselves. The arrival time error will be averaged over the scenarios within each test set. For each scenario the arrival time error is the positive difference in seconds between the real arrival time for the route the algorithm prescribes and the predicted arrival time. Adaptive path algorithms and policy algorithms update the expected arrival time each time a decision node of the routing network is reached, for these algorithms the travel time error for one test trip is computed as the average over all expected arrival time errors.

5.1.3 Setup time

Another aspect of algorithm performance that we consider is the computation time. One of the ways in which algorithms can be time-consuming is in their initialization. This is mainly an issue for algorithms that make use of a lot of data. Policy algorithms compute a whole mapping from possible traffic states to routing decisions, the number of traffic states they consider can be quite large. In order to get reliable results, the computed initialization time is taken as an average of several runs. The initialization time depends on the size of the network and the amount of information provided.

5.1.4 Longest query time

A different aspect of runtime performance is to measure the time needed for returning a routing decision each time a query is made. Each time a test vehicle reaches a decision node in the routing network, it sends a query to the algorithm which then returns the next subpath in the network to take. This holds for all algorithms except fixed path algorithms since they do not update their routing advice. Suppose that all algorithms have proposed a route and that the advised route changes because of new information, then the question is how long it takes an algorithm to process this new information and come up with an updated routing advice, regardless of whether the advised route changes or not. Because we are interested in the longest time that we have to wait to make sure the algorithm has updated its advice, we use the longest update time of a scenario as the performance measure. In our test sets we store the longest query time for each test trip and average over this. We do not use the overall worst query time since the results could then be influenced quite a lot by fluctuating computer performance.

5.1.5 Solution space size

The amount of memory that is used by routing algorithms is also an important aspect of performance. Nowadays the amount of available memory is usually not a bottleneck for most computations. However, routing advice systems are used a lot for in-car advice and being able to create a routing advice that needs little computation and/or little memory space might reduce the costs for producing such devices as well as reduce the amount of energy they consume which could be an issue since a routing device operates on the car's energy. The aspect of memory usage is also important to look at because some of the policy algorithms compute large policies. We will not look at the memory usage as indicated by the test computer but we will investigate the size of the solution space that the algorithms produce. The solution space size will be defined as the number of decisions that are created by each algorithm. We only apply this measure to policy algorithms. For fixed path and adaptive path algorithms the solution space size equals the number of decision nodes on the path they return, which usually equals only a few tens or hundreds of decisions.

5.2 Test scenarios

We test our algorithms on several types of test cases that reflect situations that can occur in reality. The test scenarios should not be trivial so the number of possible routes is larger than one in all our test sets. We provide a list of parameters that are varied between different scenarios or for which a fixed value is chosen:

- Traffic density: we test scenarios with different traffic densities.
- Network size: we look at small and large networks in order to be able to say useful things about runtimes and complexities of the algorithms.

- The number of available historical traffic scenarios: by varying the amount of available information we can see how the runtime and the quality of the routing advice of algorithms depends on the amount of information available.
- The amount of available information: live data can be network-wide or partial. We introduced several algorithms that receive full data, partial data or no live traffic data so the difference in their performance tells us how important the types and amounts of provided information are.
- Live data update interval: we choose a constant update interval of five minutes in all our test sets.
- Delay in information updates: we choose a live data delay of two and a half minutes for all our tests. Because travel time data is updated every five minutes, the latest live data is always between 2.5 and 7.5 minutes old, so the latest live data available is on average five minutes old. This seems rather realistic and at least not too optimistic.
- Accuracy of the updated information: we provide completely accurate information to our algorithms, so no noise is added to it.

The delay of live data and the accuracy are not varied because of the time scope of this project, we do not want to solve for all possible parameters. For our set of test scenarios to be useful, the following properties need to be satisfied in some of the scenarios:

- The physically shortest path is not the fastest path. Algorithms should be able to prove whether they can advise longer routes that are faster.
- The shortest path is not the same as the path that seems to be best at the time of departure. The idea is that algorithms must show whether they can adapt to changing traffic situations during the trip.
- The set of historical input scenarios should not be biased towards the optimal solution. Using the average of all input scenarios should not always lead to the optimal result.

Furthermore, the number of different test cases should not be too small. For some test cases it might happen that a certain algorithm gives a good advice despite the fact that it misinterpreted the situation, it might just have been lucky. Similarly, a good advice could now and then result in an unexpectedly poor result. With a high number of test scenarios the luck factor becomes lower so the results will be more useful. Testing on a lot of different networks leads to more reliable results but it is not strictly necessary because a lot of different traffic conditions can be tested by choosing different time intervals from available real data.

Based on this information and all our considerations, we come up with a list of test scenarios. First we introduce a simple made up scenario that tests for very specific situations, then a real world scenario that makes use of real traffic data and thirdly a real world network with fake data that is used to measure runtimes and complexities of algorithms.

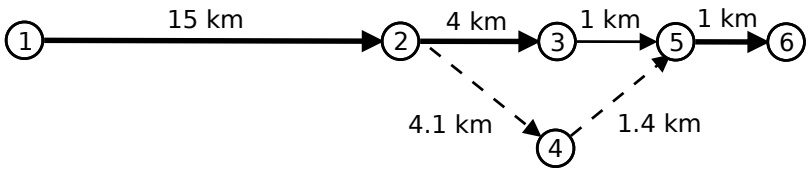


Figure 5.1: The detour network shown schematically. Thick arrows are two-lane roads, other arrows are single-lane roads. The speed limit is 120 km/h for solid arrows and 90 km/h for dashed arrows.

5.2.1 Detour network

The first test scenario is a small hand-made network with one short route and one detour, as shown in Figure 5.1. A vehicle has to find its way from node 1 to 6, it can take the straight highway via node 3 or the detour road with a lower speed limit via node 4. All the other traffic in the scenario drives from node 1 to 6 via node 3. From node 3 a traffic jam can build up in upstream direction because the number of lanes goes from two to one there. This only happens when the vehicle density between nodes 1 and 3 is high enough.

The scenario has a time span of four hours. There are eight different made-up historical scenarios, for historical scenario 1 the number of vehicles departing from node 1 equals 1,000, for scenario 2 the number of vehicles is 2,000, which continues similarly up to 8,000 vehicles for scenario 8. The vehicle insertion is uniformly divided over the time span of four hours. All eight scenarios are simulated using SUMO, the average travel time is produced per five minutes per road segment. Up to 4,000 vehicles no traffic jam occurs but for the four busiest scenarios a traffic jam occurs that grows to a length between 3 and 14 km at the end of the simulation time span. Whether the detour is faster or not depends on whether there is a traffic jam around the time our test vehicle is making the trip from node 1 to 6. Two test scenarios are made, one in which 2,500 vehicles are inserted and one in which 7,500 vehicles are inserted, these numbers are chosen deliberately different from all eight historical scenarios. In this way the conditions in some historical scenarios are similar to the conditions in the test scenarios but they are never exactly equal.

We will run a set of tests on this simple network in order to find out some basic properties of the routing algorithms. The tests are ordered below from basic to more advanced. Some of the algorithms round their decisions to intervals of five minutes, therefore the time of departure of the test vehicle might influence the results since the number of intervals between the rounded departure time and the rounded time at which the start of the detour is reached can differ. Therefore we will run five versions of most tests below. When departure time range d to $d+4$ in minutes is mentioned in any test explanation below, then the test will be run with departure times d, $d+1$, $d+2$, $d+3$ and $d+4$.

- Time test: the test vehicle departs ten minutes before the start of the vehicle flow. The average speed of the roads the test vehicle uses then equals the maximum allowed speed at the time the vehicle passes so the straight highway route is then the fastest one. We test this on the scenario in which 7,500 vehicles are inserted so a traffic

jam will start to grow but our test vehicle is not hindered by this. All eight historical scenarios will be provided as input to the algorithms. This tests whether the algorithms correctly see that the traffic jam occurs on a different time than the time the test vehicle uses these roads.

- History test: a scenario in which 2,500 vehicles are inserted, so no traffic jam occurs. As input scenarios scenarios 3 to 8 will be given, so the majority of historical scenarios will contain a traffic jam. The test vehicle departures are between 60 and 64 minutes after the start of the scenario, which is late enough to ensure that the detour would be faster if one of the busy input scenarios would be the test scenario. For the test scenario, the highway route takes about 11.5 minutes while the detour takes 12.5 to 13 minutes. With this scenario we test whether the algorithms average the travel times from the input scenarios or whether they see the correspondence with a similar historical scenario.

- Reactive test: in the test scenario 7,500 vehicles are inserted so a traffic jam occurs. The test vehicle departure times are between 60 and 64 minutes after the scenario starts. By the time it reaches node 2, the traffic jam on the highway is already significant so the algorithms should react to this and choose the detour. The highway route takes more than half an hour in this scenario while the detour takes only around 15 minutes. Historical scenarios 1 to 6 are given as input to the algorithms so a majority contains no jams. This test could be regarded as the inversion of the history test, the algorithms should react to the traffic jam by advising the detour route.

- Proactive test: again the scenario with 7,500 vehicles, the test vehicle is inserted between 6 and 10 minutes after the start of the scenario. For these departure times the test vehicle reaches node 2 when the traffic jam on the highway is very small such that taking the highway seems to be faster than taking the detour when looking at the latest live data. However, while the vehicle would drive on via the highway, the traffic jam would become larger so eventually the highway route takes between 15 and 17 minutes while the detour takes about 14 minutes. The algorithms are provided with historical scenarios 1 to 6. This scenario tests whether the algorithms use prediction so whether they foresee that the traffic jam will become larger based on the historical data.

- Bias test: a test with the same parameters as the reactive test except that the algorithms receive only historical scenarios 1 to 4 as input, which are all scenarios in which no traffic jam occurs. We can then see whether the algorithms can cope with a situation that is significantly different from all known historical scenarios. Algorithms that do not use current or live data might likely fail this test. For the other algorithms it is interesting to see whether they can cope with useless historical information by relying on current or live data or the arrival time at the decision node.

- Adaptive bias test: this test is equal to the bias test except that the test vehicle departure times are between 8 and 12 minutes after the start of the scenario. In this case, based on the data at the moment of departure one would still advise to take the

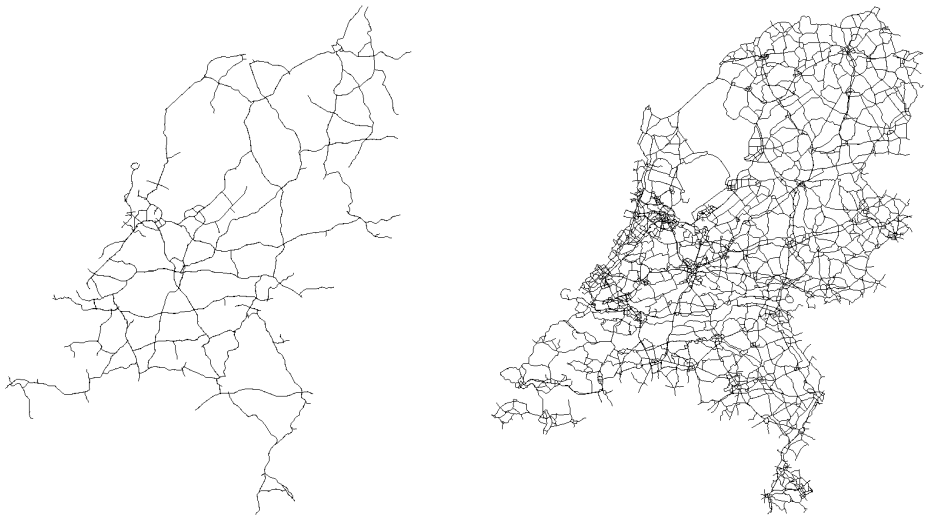
highway but by the time node 2 is reached the detour is better according to the live data for departure times of 9 up to 12 minutes. If we would incorporate a departure time of 13 minutes or more after the start, then algorithm FP-C would pass, meaning that the data at the moment of departure points to taking the detour, which was not the intention of this test. Algorithms that listen to the live data can react based on the updated information that becomes available just before reaching node 2, so this scenario tests whether algorithms react on on-line data that does not correspond to any historical input. A proactive version of this test is not made since without similar historical input scenario there is no way in which any algorithm could predict that there will be a traffic jam at all.

The results of these tests tells something about the robustness of the algorithms, about how well they will be able to cope with different traffic conditions. The bias tests might be less relevant in reality since traffic conditions are hardly ever completely different from a set of historical scenarios, especially when the historical data set is large. However, traffic accidents can cause irregular traffic conditions so therefore the bias tests are still useful.

5.2.2 Rotterdam ring network

The second test set we introduce focuses on the quality of the routing advice for a small real world network. The network consists of the highway ring road of Rotterdam plus a few primary roads, it is shown already in Figure 4.2 including the locations for which real data is used to generate traffic. The vehicle trip used for the test vehicle leads from highway A13 to the north of the Rotterdam ring towards highway A29 south of Rotterdam, as already shown in Figure 1.1. For this trip there are multiple routes that are comparable in travel time. From NDW we obtained data for the loop detectors for all mornings from January 12 to April 26, 2014. This time period is chosen because it contains a lot of normal traffic days, it contains one small school vacation and no national holidays. A simulation is run using SUMO for each morning in this range from 7 AM to 11 AM, which results in 105 scenario days, which equals fifteen complete weeks. As explained in Chapter 4 the traffic conditions in the scenarios are not highly realistic because of the many problems encountered, however, the overall average vehicle speed is rather similar. It is still interesting to see how the routing algorithms perform on these scenarios since the results can still be used to proof certain algorithm concepts and to draw conclusions about the usefulness of data. The test details are described below.

- History size test: We like to draw conclusions about how the routing quality depends on the number of historical input scenarios and how this affects the runtime. Therefore we run test scenarios where the algorithms receive historical data for different number of days: one week, two weeks, three weeks and similarly up to fourteen weeks. Historical data is always provided for other weeks than the week of the test day such that the test conditions are not part of the historical input. For each number of weeks provided, the results of all algorithms are averaged over 420 tests: all 105 mornings are tested for departure times 7:05, 7:15, 7:30 and 8:00 AM. These departure times are chosen such that for the first two the traffic conditions change rapidly



(a) trunks and highways (b) secondary up to highways

Figure 5.2: The Dutch road network with two different levels of filtering out smaller roads.

because the simulation starts shortly before it. Giving proper routing advice can be tricky then since traffic jams might start to grow significantly around the time a certain path is traversed. The last two departure times are a bit later such that the network already contains quite some traffic when the test vehicle departs, the traffic conditions might be a bit easier to predict and using live data then has a good chance of resulting in proper advice. We will mainly look at the quality of the advice produced by the algorithms but we also look at the arrival time prediction error. Next to this we will investigate the runtimes of the algorithms and the policy size for the different policy algorithms for different amounts of historical data.

5.2.3 Dutch road network

In order to say useful things about the scalability of the algorithms we include a large test scenario. For this we use the Dutch road network, the smallest roads are filtered out. We will look at two versions: one with only highways and trunks and one where also primary and secondary roads are included, as shown in Figure 5.2. The road graph for the first version contains 12,361 junctions and 13,834 roads, according to the framework in which the network is loaded. The second version has 39,981 junctions and 101,203 roads. Simulations for networks of such a size become are significantly slower than real time when a realistic number of vehicles is simulated on them. Creating a test scenario and multiple other scenarios that can serve as historical input is therefore very time and memory consuming. In order to accomplish this we would need a lot of detectors and a huge amount of readings for these detectors. Therefore we decided not to simulate any scenarios.

We will run tests on both network versions. For all tests below the test trip starts west of Rotterdam at highway A15 (OSM node 44157385) and ends at highway A7 to the east

of Groningen (OSM node 896787311), the travel distance is about 300 km. The test details are as follows:

- Dutch highways test: we use the Dutch highway network for a test scenario without any traffic, which means that for each edge the average speed is simply the maximum allowed speed. This scenario is also provided to all algorithms so they all know the conditions on which they are tested. This is not a problem since we will focus on the runtime of the algorithms for this test. The results are averaged over one hundred runs.
- Dutch main roads test: for this test the network version is used that includes both highways and primary and secondary roads. The number of possible routes and thus the number of decision locations is expected to be much larger than for the highways test. The algorithms have access to 25 fake historical scenarios. Per road segment per time interval in each scenario the average speed is the maximum allowed speed multiplied with a randomly drawn factor from the uniform distribution $[0.5, 1]$. The scenarios are clearly not realistic since the way in which the edge speeds are computed lacks any spatial and temporal dependency. However, the purpose of this test is to push the algorithms to the limits in order to see how well they scale. We intended to provide one hundred fake historical scenarios but such an amount of data can not be imported due to memory limitations. Because of the high runtime of this test, the test results are averaged over 25 runs instead of over 100 runs as in the Dutch highways test.

5.3 Hypotheses

We will not go into detail about the results that we expect for specific test scenarios but we will give some general expectations about the algorithms. The fixed path algorithms are quite basic and might perform worse than the other types of algorithms but the version that combines current and historical data might do pretty well. These algorithms are expected to be faster than most other algorithms. The adaptive path algorithms make use of live data so they are expected to perform decently, especially when combining historical data with live data. They are quite fast and compute only a single path at a time. Policy algorithms are very different because they compute a possibly large mapping of decisions for different locations and times. They are expected to be slower in their initialization but faster in query times than adaptive algorithms. We expect that the best policy algorithms produce routing advice of similar quality as the adaptive path algorithms, however some policy algorithms might produce policies so large that the memory usage could become an important issue.

5.4 The test framework

A framework is made in Java in which the data structures are present to work with graphs representing road networks. Multiple traffic scenarios that define time-dependent weights for such graphs can be imported. When importing a set of historical scenarios the travel

times that the different scenarios prescribe for any edge are sorted, such that the shortest and longest times can be obtained in constant time by the routing algorithms whenever they need it. The framework can be used to test the routing algorithms, it measures the runtime for initialization of algorithms and it keeps track of the longest query time of each algorithm. It contains a method for computing a set of shortest paths in a network under the constraints as explained in Section 2.1.2. Another method then receives this set of paths and returns the set of subpaths that together with their start and end nodes form a routing network.

The framework also handles necessary calls to external programs such as Osmosis for filtering an OSM network and SUMO for running scenarios. In order to prepare scenarios the framework contains methods for reading data files and converting them to files that can be used as input for SUMO simulations. The framework keeps track of files it receives and creates, such that on later runs earlier provided file locations do not have to be given over and over again. For more detailed information about the framework we refer to Appendix B.

Chapter 6

Results

In this chapter we present the results of the tests introduced in Section 5.2 and we will discuss these results per test network separately in order to evaluate all the routing algorithms that were defined in Chapter 3. After that we will take a wider perspective and draw overall conclusions of the performance of our algorithms. This leads to answering research subquestions one and three and because of all the answers we gathered by then we will finally tackle our main question.

Because runtimes are also investigated, some details about the test environment might be relevant. The tests are performed on an Intel® Core™ i7-4600U CPU laptop with four cores and 8 GB memory with Ubuntu 12.04 as operating system. During all tests the laptop is connected to the power grid, internet connection is disabled and unnecessary programs that use a significant amount of memory or CPU time are closed. The code is run using Java 7.

6.1 Simple detour scenarios

First we will look at the detour test scenarios that were introduced in Section 5.2.1. These scenarios are designed to test specific properties of the algorithms. For each test the algorithms can either return the best route or the worst route. Because we want to draw conclusions about the main properties of the algorithms we will only show which algorithms passed which tests and discuss these results. Therefore, other results like runtimes or errors in the predicted arrival time are not considered for this test set.

Table 6.1 shows an overview of all the test results for the detour network. Because some algorithm versions are very similar in behaviour, not all algorithms are listed: versions using property X or partial data are not shown when a version without X or with full data exists, since their results are the same for this test set. We will now interpret and explain these results per test scenario. At first we see that all algorithms pass the time test, from which we deduct that all algorithms take time dependency into account. This is true because for at least three quarters of the scenario time span it holds that the traffic jam on the highway is so significant that the detour is much faster. An algorithm that would average over the whole time span would therefore advise to take the detour and thus fail the test.

algorithm	time	history	reactive	proactive	bias	ad. bias
FP	X	X	-	-	-	-
FP-C	X	X	X	-	X	-
FP-HC	X	X	X	-	X	-
AP-L	X	X	X	0.4	X	0.8
AP-HL	X	X	X	0.4	X	0.8
P-HT	X	-	X	-	-	-
P-HIT	X	0.4	-	0.2	-	-
P-HI	X	X	X	0.8	0.6	-
P-HL'T	X	X	X	X	-	-
P-HLIT	X	X	X	X	0.6	-

Table 6.1: Overview of detour network test scenario results. A cross means that an algorithm passed the test for all instances, a dash means it failed in all cases and a decimal number indicates the success rate out of the five different runs of a test.

In the history test two policy algorithms fail: P-HT and P-HIT. The fact that there is dense traffic with a traffic jam ahead does influence the arrival time at the decision node but apparently this makes only a minor difference in the time at which the decision node is reached. So likely the arrival time at the decision node is still rounded to the same five minute data point as for the historical input scenarios with low traffic density. Therefore the simpler policy algorithms cannot make the distinction and advise the wrong route. P-HI does not fail because it does not round the arrival times to intervals of five minutes, it compares the real arrival time at the decision node with the historical scenarios. Therefore, the advice space might fluctuate a lot when varying the time, but it passes this test. The other policy algorithms succeed because they receive live traffic data that helps them to make the right decision. The basic fixed path algorithm is simply lucky, it always advises the highway route, which is in this case faster.

The basic fixed path algorithm and P-HIT fail the reactive test but all other algorithms correctly advise the detour, based on the later arrival time at the decision location, the data at the moment of departure or the live data updates. All data types point to the traffic jam on the straight route and therefore most algorithms pass the test. P-HIT suffers from the fact that it computed only one decision for the node at which the detour starts so it fails to distinguish between the different possible scenarios.

The proactive test is similar to the reactive test, except for the fact that the data at the moment of departure suggests to take the highway while the detour is in fact faster. The fixed paths therefore fail, which was to be expected. The policy algorithms that make use of live data pass the test for all five departure times. The adaptive path algorithms fail for the earliest departure times but pass for the last two. This is likely due to the fact that adaptive path algorithms depend more heavily on the live data while policy algorithms compute their decisions mainly based on historical data, guided by the received live data. P-HT and P-HIT fail, again only one decision was computed for the point at which the detour starts. However, for one departure time the arrival times at the decision point were rounded to

different time intervals because once the generated policy size was one larger and for this instance P-HIT passes. P-HT fails because it lacks property I so it averages over historical scenarios instead of only looking for a similar one.

Several algorithms fail at both bias tests. FP fails because it always advises the highway route, which is not optimal here. Policy algorithms receive the arrival time at the decision node and some of them also traffic information, but they mostly fail because they rely heavily on the misleading historical data. Exceptions are P-HI that partly passes by relying on the non-rounded arrival time at the decision location and P-HLIT that receives full live data and also manages to pass for three of the five tested departure times. The two fixed path algorithms that use current data and the adaptive path algorithms pass the first bias test because the data at the moment of departure is sufficient for foreseeing the problems on the straight route. In the adaptive bias test the data at the moment of departure is not sufficient, only live data available shortly before reaching the decision node provides information about the seriousness of the traffic jam although it is not so obvious as for the normal bias test. Therefore, all fixed path algorithms and policy algorithms fail the adaptive bias test. The adaptive path algorithms pass the test, except for the earliest of the five departure times since the traffic jam is then still small enough to take the straight route according to the live data available when reaching the decision node.

In general we see that algorithm FP is not very smart, it always advises the shortest route, which happens to be the optimal one for the first two tests. P-HT and P-HIT seem to suffer from their time roundings, which make them fail more than P-HI and more than the other policy algorithms that receive live data. P-HI actually performs pretty well given the fact that it does not use current or live data at all. The policy algorithms with live data and the adaptive path algorithms pass a lot of tests, a bit more than the fixed path algorithms that did not pass the proactive test. Overall we can say that fixed path algorithms have the disadvantage of not being able to use new traffic information. Policy algorithms P-HI, P-HLIT and the variants using partial data and/or property X seem to be more promising than P-HT and P-HIT. The adaptive path algorithms can pass all tests and are therefore very promising.

6.2 Rotterdam ring scenarios

The Rotterdam ring is a small real world network introduced in Section 5.2.2. It has 776 junctions and 934 roads. The routing network build from it for a trip from the north side to the south side of the ring has 16 junctions and 29 subpaths.

6.2.1 History size test

We will now discuss the results of the history size test, in which we investigate how the advice quality and other properties depend on the amount of history provided. This is a large experiment with a lot of results that will be discussed in the following sections. First we will look at the quality of the routing advice and how that depends on the size of the historical input data. After that we will show and discuss how well the algorithms are able to predict the arrival time. The next topic of evaluation is the runtime of the algorithms and

then the policy size of the policy algorithms is investigated, both these results are used to draw conclusions about the scalability of the algorithms.

Quality of the routing advice

We will first look at the quality of the routing advice using the measure explained in Section 5.1.1: the average delay percentage. It is computed relative to the optimal travel time, which is obtained by running a fixed path algorithm with the exact test conditions as input. The average delay percentage of all the algorithms is shown in Table 6.2 for fourteen weeks of historical input, sorted by the groups that we defined in Section 3.1. Figures 6.1 and 6.2 show the results graphically for the partial data group and the full data group for one up to fourteen weeks of historical input, so the results in the table correspond to the right end of the graphs in the figures. A lot can be said about the results, we will explain the results per group and finally draw some conclusions about the overall results.

The results of the no live data group are not so interesting to show graphically, the delay percentages of all algorithms in this group are about constant, regardless of the amount of historical scenarios provided. The basic fixed path algorithm has a very high average delay factor because it always advises to take the road through the center of Rotterdam, regardless of the traffic conditions. Algorithm P-HT performs very well if we consider that it does not apply property I, so it computes the average travel time over all historical scenarios and extracts a time-dependent decision per location. Its advice depends on the arrival time at intermediate locations. The algorithms P-HIT, P-HI and P-HXI all perform a bit better. Therefore, it seems that using property I has a positive effect, which leads to the conclusion that spatial dependencies between road travel times are to be taken into account since using property I means that the travel times of the whole network are only related to each other within the same historical scenario and time interval.

Now we look at the results for algorithms of the partial data group, the group in which algorithms only receive current or live data for subpaths starting from the current decision location. From Table 6.2 we can see that the delay percentages of all algorithms in this category are rather low. We see that the fixed path algorithms perform the worst and the adaptive path algorithms perform best. It makes sense that the adaptive path algorithms versions beat the fixed versions since the two adaptive algorithms basically do the same as their fixed path algorithm counterparts, except that they receive live data updates that they use to reconsider their advice. A remarkable result is that the algorithm versions with property X perform equal or worse than their counterparts without this property. Using more than one similar historical scenario is therefore not better when the provided current or live data is very limited. Figure 6.1 further shows how the delay factor depends on the amount of historical information provided. When only a few weeks of information are provided, all algorithms except P-HL'T perform significantly worse but for four or more weeks of information the performance stabilizes at a low delay percentage. The fact that the performance of P-HL'T is more stable might be due to the fact that it computes a decision for each possible combination of discrete travel times of subpaths exiting from the query node. Therefore, even with very little historical data it computes a decision for a significant set of states.

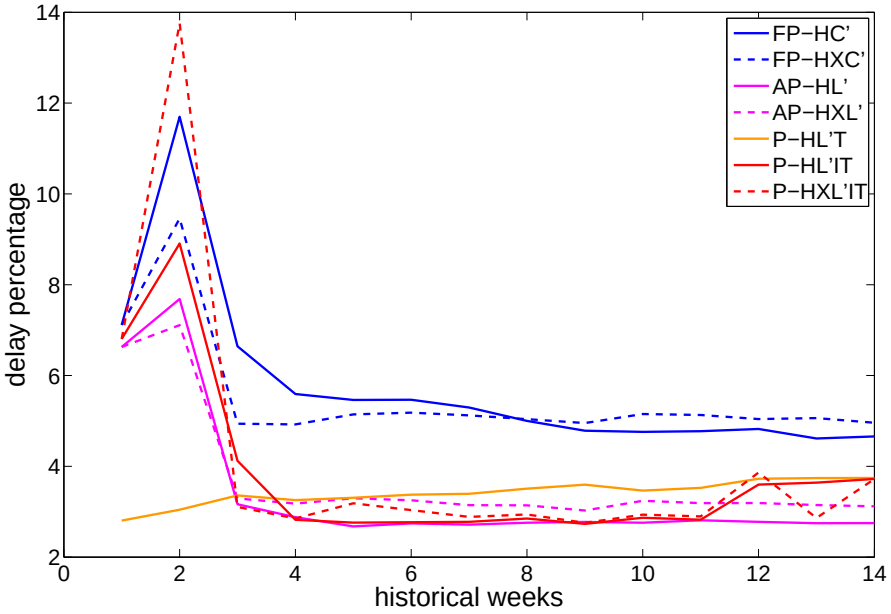


Figure 6.1: The average delay percentage for algorithms of the partial data group for varying amounts of historical input.

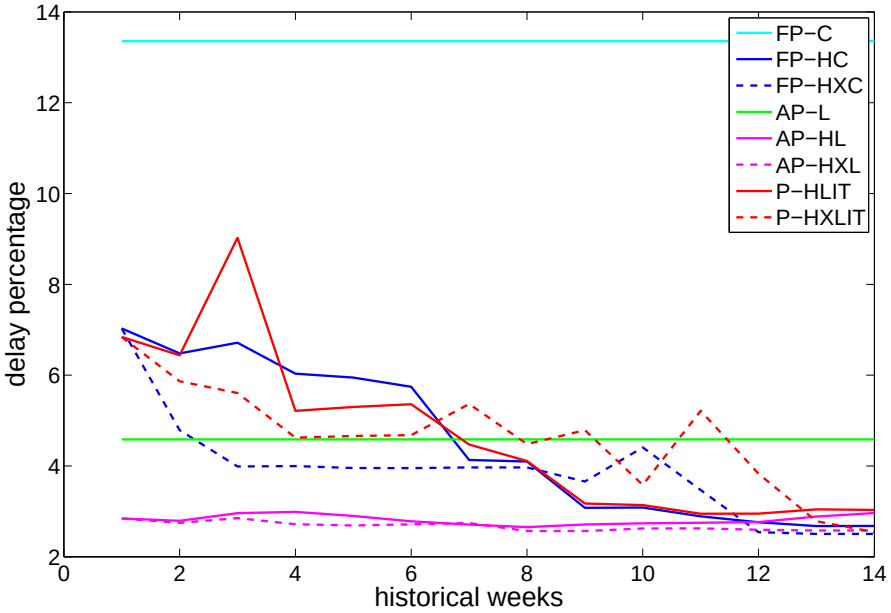


Figure 6.2: The average delay percentage for algorithms of the full data group for varying amounts of historical input.

no live data	
algorithm	delay
FP	49
P-HT	3.6
P-HIT	2.4
P-HI	2.6
P-HXI	2.5

partial data	
algorithm	delay
FP-HC'	4.7
FP-HXC'	5.0
AP-HL'	2.8
AP-HXL'	3.1
P-HL'T	3.7
P-HL'IT	3.7
P-HXL'IT	3.7

full data	
algorithm	delay
FP-C	13
FP-HC	2.7
FP-HXC	2.5
AP-L	4.6
AP-HL	3.0
AP-HXL	2.6
P-HLIT	3.0
P-HXLIT	2.5

Table 6.2: Average delay percentage of the algorithms for fourteen weeks of historical input data. The results are sorted per algorithm group.

The full data group contains a lot of well performing algorithms, only FP-C has a significantly higher delay percentage. It uses only the most recent data known at the moment of departure. However, our test set is chosen in such a way that it also contains scenarios in which the traffic conditions change significantly during the trip. A second observation from Table 6.2 is that AP-L performs a bit worse than most other algorithms in the group, despite the fact that it updates its route advice with each five minute live data update. The other algorithms make use of historical data, which is used to predict how the traffic will evolve. Predicting the future clearly helps improving the quality of the routing advice in our test set. The algorithms that use prediction are quite similar in performance but we notice here that the algorithms that use property X perform better than their counterparts without property X. This is remarkable, since for the partial data group we derived the opposite conclusion. Why this is the case is not clear to us, we would expect that using property X would always have a positive effect on the performance.

At last, we compare the results of the different algorithm groups. An important observation is that the best algorithms of each group are very similar in performance. Another important result is that using historical information can really improve the quality of the routing advice. Compared to the most rudimentary algorithm FP, large improvements are achieved by using recent data (FP-C) and then also live data (AP-L). The best algorithms that use historical data further reduce the average delay with almost a factor two compared to AP-L, even when the travel time data is spatially limited or when only arrival times at intermediate locations are used. We ran this test on one network for one specific trip, so we can not state that all these algorithms easily beat AP-L under all circumstances. However, from the fact that so many algorithms perform significantly better with the same or less information we do conclude that historical data can be used successfully to make predictions that help improve the routing advice.

Arrival time prediction errors

Now we will look at the performance of the algorithms by comparing their arrival time prediction error, a criterion introduced in Section 5.1.2. We show absolute errors instead of percentages because an arrival time prediction error expressed in seconds is quite intuitive. To put this in perspective: the optimal route for the test trip usually takes about twenty to thirty minutes. Algorithms that find one similar historical scenario to base their advice on will use the travel times of this scenario to compute the remaining travel time. Similarly, algorithms that use the few most similar scenarios use the average travel times over these scenarios. So to compute the expected arrival time, each algorithm uses the available information in the same way in which it computed its routing advice.

no live data	
algorithm	time error
FP	1 572
P-HT	80
P-HIT	69
P-HI	121
P-HXI	189

partial data	
algorithm	time error
FP-HC'	44
FP-HXC'	44
AP-HL'	43
AP-HXL'	45
P-HL'T	34
P-HL'IT	61
P-HXL'IT	59

full data	
algorithm	time error
FP-C	207
FP-HC	26
FP-HXC	22
AP-L	66
AP-HL	43
AP-HXL	38
P-HLIT	39
P-HXLIT	33

Table 6.3: Average error in seconds between the real arrival time and the expected arrival time that the algorithms compute, for fourteen weeks of historical input data. The results are sorted per algorithm group.

The arrival time error is shown per algorithm group in Table 6.3. When we compare the error differences to the differences in the delay percentages of Table 6.2 we see a lot of correspondence. Algorithms with a very high delay percentage also have a large error in their arrival time prediction, this holds mainly for FP and FP-C. Just as for the delay percentages, AP-L performs a bit worse than most other algorithms in the full data group. Furthermore, algorithms P-HI and P-HXI perform significantly worse, it is not clear why this is the case although it makes sense that the error is higher for policy algorithms that do not receive recent or live traffic data. There is one important new thing that we learn from the arrival time errors: a lot of algorithms in the full data group perform better than most algorithms from the partial data group, and the lowest errors in the no live data group are significantly higher than most errors in the other groups. From this we conclude that algorithms that receive travel time information are better able to estimate the arrival time than algorithms that don't.

We also notice that the best fixed path algorithms actually show very low travel time errors, despite the fact that they only compute the expected arrival time before departure. This means that they can predict pretty well how the traffic conditions will evolve based on similar historical data, for our test set they are not even inferior to the best policy algorithms

and adaptive path algorithms. However, one could think of scenarios in which all fixed path algorithms would perform worse, for example when a traffic accident occurs during the trip. Fixed path algorithms cannot correct for this while most other algorithms can.

Runtime

The next discussion will focus on runtime results, which is split up in the setup time and the longest query time of each routing algorithm. We computed the setup time and longest query time for all twenty algorithms for one up to fourteen weeks of historical input. We will not show how the setup time changes depending on the amount of historical information because there are so many similarities that we can explain it in a few sentences. Algorithms FP, FP-C and AP-L that do not use historical data have a setup time between 50 and 100 microseconds. Their setup time does not depend on the amount of historical information available, which makes sense since they do not use this information at all. All the other algorithms have a setup time that grows about linearly with the number of historical weeks of data provided. For all these algorithms the setup time for one week of historical data lies between 60 and 80 ms. When the amount of historical input data grows to fourteen weeks, the setup times of all these algorithms grow about linearly to a range of 600 to 670 ms. Compared to the algorithms that do not use historical data, the setup time of these algorithms starts about a factor 1,000 higher and becomes about a factor 10,000 higher for fourteen weeks of historical data. When we compare these runtime results with the setup time orders presented earlier in Table 3.2, we see that all algorithms with a relatively high setup time have a component ϕ in their setup time order. This component of $O(ni\log i)$ represents the time needed to import and prepare the historical input data. Due to the part $i\log i$ the setup time should grow about linearly when the amount of historical information increases, which is the case here. The setup time for all algorithms is still below one second, which is in practice fast enough for a routing device. However, the test network is rather small and a lot of minor roads are filtered out.

The query time of all algorithms is very low, the longest query time lies in the range of three microseconds and two hundred microseconds. The adaptive path algorithms and P-HLIT and P-HXLIT have query times above one hundred microsecond. For the other algorithms the query time is in tens of microseconds, except for P-HT and P-HIT that have a longest query time of a few microseconds. We will not further discuss the query times for this test since they are so low that it simply means instantly from human perspective. We will discuss the scalability of algorithms further when looking at the results of the Dutch road test scenarios.

Decision space size

Now we look how the size of the policies that policy algorithms compute depends on the amount of historical information provided, the results are shown in Figure 6.3. The shown policy size values are the number of stored decisions, where each decision belongs to a certain node and time stamp or time interval.

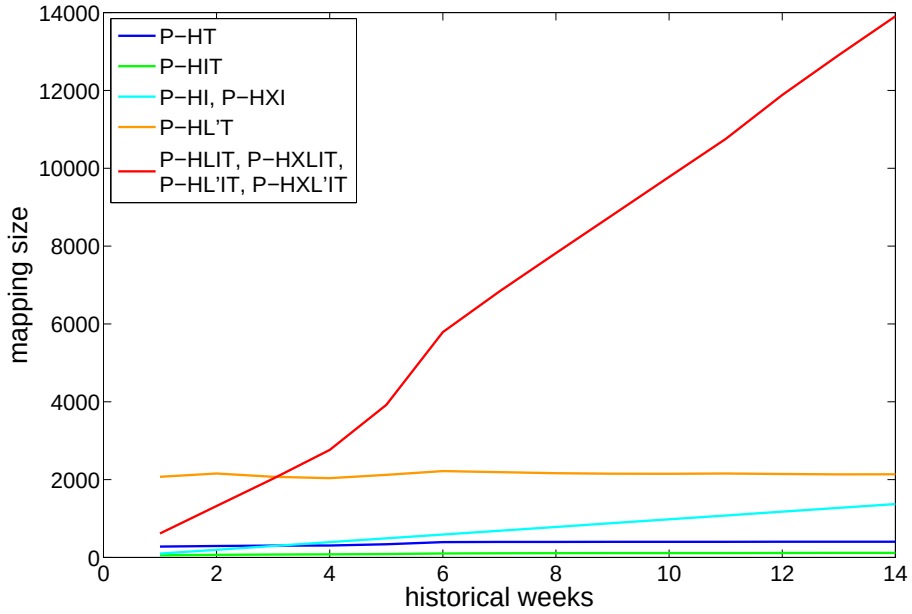


Figure 6.3: Policy size of the different policy algorithms in the Rotterdam history test for varying amounts of historical input data.

In table 3.2 we already presented the decision space orders for the routing algorithms. There, we saw that the order of P-HI and P-HLIT and their counterparts with partial data and/or property X includes a component i, so the amount of historical data provided. The figure shows that the policy size indeed grows linearly with the amount of historical data provided for these algorithms. The size of the other policies is independent of the amount of historical information, which also corresponds to the results in the figure since the corresponding graphs show that the policy size for these algorithms is almost constant regardless of the amount of historical input data. The fact that the number of decisions increases a little for algorithms P-HT and P-HIT is due to the fact that the possible arrival time span for each decision node of the routing network can become a bit wider when more scenarios are provided. Because both algorithms apply property T, the arrival time span for each node is split up in equidistant discrete times so the policy size does not grow linearly with the historical data size.

Algorithm P-HLIT and the variants with partial data or property X relatively have a higher risk of running into memory problems. A policy size just over ten thousand is not very high yet, but one should take into account that it can be much larger when applying the algorithms on much larger networks. P-HI can run into trouble for the same reason, but its policy size is one order of magnitude lower so this algorithm has better chances to being useful when dealing with larger networks. We will look at the policy sizes again when presenting the results of the Dutch main roads test, which has a much larger network.

6.3 Dutch road scenarios

The Dutch road network is a much larger network than the networks in the other experiments. The scenarios defined for this test set are meant to draw conclusions about the scalability of the routing algorithms. We will first discuss the size of the created routing networks for the two tests defined in Section 5.2.3 and the time needed to create them. After that we will present the runtime results of both test sets at once.

6.3.1 Routing network size and experiment setup time

First we look at the network version with only highways and trunks, as visualized in Figure 5.2a. Computing the routing network is done by finding 100 good paths through the network as explained in Section 2.1.2. The choice for 100 paths is a trade-off between keeping the runtime low and selecting enough paths to ensure that multiple good routes become available. The routing network computed for the test trip has 70 decision nodes and 137 subpaths. Compared to the original road network size as introduced in Section 5.2.3, this is a reduction of respectively a factor 175 and 100. This is a significant reduction that helps to reduce the size of policies and the runtime of most algorithms significantly. Computing the routing network takes about four seconds. Furthermore, loading the original network into the framework takes about two seconds and importing one traffic scenario takes about four seconds. These steps contribute a lot to the total runtime because all the algorithms run in one second or less. However, without making use of a routing network a lot of algorithms would likely take several seconds.

The second test applies to the network version with both highways and primary and secondary roads, as shown in Figure 5.2b. Compared to the highways version this network contains a lot more nodes and edges. The computed routing network has 653 nodes and 1,320 paths connecting these nodes, these numbers are about a factor ten higher than for the highways test. Compared to the original network, the number of nodes and paths is respectively about 60 and 80 times lower. However, computing the routing network for a trip leading almost from one end of the network to the other end takes about 23 seconds. There are more time consuming steps in this test procedure: reading the network files and importing the network into the framework takes about 15 seconds. Importing one traffic scenario describing the time-dependent travel time per edge of the original SUMO road network takes about 30 seconds. One traffic instance is already close to 1 GB data in XML format. In this test the algorithms have access to 25 fake historical scenarios, instead of just one as for the highways test.

Runtime

When we presented the results of the Rotterdam history test, we saw that the setup time of all algorithms that make use of historical data is dominated by the time to import and prepare this data. It is theoretically possible to import historical information only once and then use an algorithm multiple times for routing advice. Therefore we will present the setup time results excluding the time to import historical information. The time for importing

one historical scenario is around 25 milliseconds for the highways test and loading the 25 historical scenarios in the main roads test takes about 15 seconds.

	highways test		main roads test	
algorithm	setup	l. query	setup	l. query
FP	0.66	-	5.3	-
FP-C	0.42	-	3.2	-
FP-HC'	0.63	-	4.6	-
FP-HXC'	0.66	-	5.2	-
FP-HC	0.67	-	8.8	-
FP-HXC	0.63	-	7.8	-
AP-L	0.38	0.30	2.4	3.9
AP-HL'	0.58	0.40	4.8	1.1
AP-HXL'	0.57	0.34	4.7	0.96
AP-HL	0.68	0.50	7.3	9.8
AP-HXL	0.67	0.50	6.8	9.6
P-HT	1.4	0.012	467	0.026
P-HIT	1.5	0.008	170	0.072
P-HI	0.84	0.015	265	0.034
P-HXI	0.88	0.014	146	0.12
P-HL'T	2.7	0.031	549	0.067
P-HL'IT	1.0	0.026	492	0.051
P-HXL'IT	0.93	0.024	1032	0.044
P-HLIT	0.88	0.13	703	6.2
P-HXLIT	0.91	0.12	1069	5.9

Table 6.4: Overview of runtime results for the Dutch highways test and main roads test, all runtimes are in milliseconds. The setup time and the longest query time are shown.

Table 6.4 shows all the runtime results. We will first look at the setup times. For the highways test the setup times of all algorithms are quite low: around one millisecond. If we compare them to the results for the main roads tests, then we see that the setup time of some algorithms hardly increases while the time for other algorithms increases a lot. For the fixed path algorithms and adaptive path algorithms we see that the setup time is about a factor ten higher in the main roads test compared to the highways test. Remember that the routing network for the main roads test is also about a factor ten larger than for the highways test. In Table 3.2 we presented the setup time orders, it shows that the setup time of these algorithms mainly scale with factor $\omega = n \log n$ which explains the growth of the setup times quite well. The setup times of the policy algorithms are between a factor 100 and 1,000 larger for the main roads test than for the highways test. The setup time order for these algorithms is also worse compared to the other algorithms. All policy algorithm setup times for the main roads test are about one second or much lower, which is fast enough for practical usage in routing devices. The main roads test is already of a size large enough for most real world vehicle

routing problems. However, we can conclude that the fixed path algorithms and adaptive path algorithms are better scalable than the policy algorithms, they can easily be applied to significantly larger problems without posing practical problems regarding the runtime.

We will now discuss the longest query times of the algorithms. Fixed path algorithms do not update their routing advice during the trip, which explains why they do not have a longest query time. We notice that the longest query time of the adaptive path algorithms and P-HLIT and P-HXLIT are about an order of magnitude larger than for the other algorithms for the highways test. The query time for these algorithms seems to be linear in the size of the routing network, since the longest query time is about a factor ten or twenty higher for the main roads test for most of these algorithms. In Section 3.7 we already showed that exactly these algorithms have a component n in their query time order while the other algorithms don't. The other policy algorithms have a much lower query time and their query time results are hardly affected by the size of the routing network. This also corresponds to the orders given in Section 3.7, since these algorithms have a query time order of (1) or (i). The query times of all algorithms are below ten milliseconds, which is very low for practical purposes. Therefore, the query times are not really a bottleneck for any of the algorithms.

Decision space size

We now report on the policy size of the policy algorithm for the Dutch test scenarios. For the highways test all policies consist of 69 decisions, which is one decision per node in the routing network, the destination node excluded. For the main roads test the results are more interesting to look at, since the policies then become quite large and there are differences between the algorithms because of the different ways in which they use the 25 historical traffic scenarios.

algorithm	mapping size
P-HT	5 118
P-HIT	1 349
P-HI, P-HXI	18 201
P-HL'T	30 459
P-HLIT, P-HXLIT, P-HL'IT, P-HXL'IT	142 389

Table 6.5: Overview of policy sizes of policy algorithms for the Dutch main roads test.

Table 6.5 shows the size of the policy produced by each policy algorithm. We see that P-HLIT and the variants with partial data and/or property X produce a mapping that is quite large. Such policy sizes can still be handled by modern computers but the policies should not become much larger. Quite some space is needed for storing policies of such a size, which might be a practical disadvantage for implementation in in-car routing devices since the devices would need a sufficient amount of memory and computing power, which increases the production costs. The policy size computed by P-HI, P-HXI and P-HL'T is a bit lower, but still quite significant. P-HIT managed to produce a mapping that is small

compared to the others which is a good result if we consider the fact that it performed well in the Rotterdam test scenarios.

6.4 Algorithm evaluation

In this section we will combine the results of the different test scenarios to draw overall conclusions about the performance of the routing algorithms. In Section 6.2.1 we showed the main performance results for the Rotterdam network. For this section we will look at the few best performing algorithms per group, based on the delay percentages from the Rotterdam history test, and use the results from all test sets to draw more general conclusions. Regarding the runtime results we will mention query times only briefly since the query times for the largest test are still so low that they will hardly influence our conclusions about the overall performance of routing algorithms. From the no live data group we will pick P-HIT, P-HI and P-HXI, from the partial data group FP-HC', AP-HL', P-HL'T and P-HL'IT and from the full data group FP-HXC, AP-HXL and P-HXLIT.

No live data group

We have chosen three algorithms with a delay percentage between 2.4 and 2.6 percent for the Rotterdam test set: P-HIT, P-HI and P-HXI. From the arrival time prediction errors we saw that P-HIT was better at predicting the arrival time than the other two algorithms. The setup times are quite similar. When we look at the decision space size as shown in Figure 6.3 then we see that P-HIT produces a significantly smaller policy than P-HI, it is independent of the amount of historical information. Table 6.5 shows similar results: the produced policy of P-HIT is more than ten times smaller than for P-HI and P-HXI. So far it seems that P-HIT is the best performing algorithm from this group. The only disadvantage of P-HIT compared to P-HI and P-HXI is shown in the Detour test results in Table 6.1. P-HIT passes less tests than P-HI, it computes too little decisions for such small tests, which is not a problem for the other experiments. P-HI and P-HXI have a small advantage because they do not round their decisions to discrete time intervals. So P-HIT performs well but if the computed policy size does not become a bottleneck, then P-HI and P-HXI reach almost the same performance while being overall a bit more robust. There is no clear winner in this group.

Partial data group

From the partial data group we have chosen four algorithms: a fixed path algorithm, an adaptive path algorithm and two policy algorithms. Regarding the delay percentages in the Rotterdam test, AP-HL' performs better than P-HL'T and P-HL'IT and these two on their turn outperform FP-HC'. The errors in the arrival time predictions are rather similar, with a slight advantage for P-HL'T. The setup times are also very similar but if we ignore the component for importing historical information then there are significant differences. Table 6.4 shows that FP-HC' and AP-HL' then have a setup time of a few milliseconds while the setup time of the two policy algorithms lies around half a second. The fixed and adaptive

path algorithms have a large advantage over the policy algorithms since they do not compute a large mapping of decisions. The mapping size of P-HL'T is significantly lower than for P-HL'IT and it does not depend on the amount of historical data. Because the query time of the adaptive path algorithm is still low enough not to cause any practical problems, algorithm AP-HL' is for us the overall best algorithm in this group, it also passes more tests from the Detour test set than the other algorithms.

Full data group

We picked the best algorithm of each main type from the full data group: FP-HXC, AP-HXL and P-HXLIT. Their delay percentages in the Rotterdam tests are so similar that the differences are not to be taken into account when deciding which algorithm is most useful. The arrival time prediction errors are also quite similar, with a slight advantage for FP-HXC. Since they all use historical information, their setup times are similar too. However, Table 6.4 shows large differences when not taking the import of historical data into account. While the time for the fixed and adaptive path algorithm are just below ten milliseconds for the Dutch main roads test, P-HXLIT needs a whole second for its initialization. Regarding query times FP-HXC has a clear advantage over the other two algorithm. When looking at the computed decision space, P-HXLIT has a large disadvantage over the other two since it computes a policy with more than one hundred thousand decisions according to the Dutch main roads test. If one hundred historical scenarios would have been provided instead of 25 then the policy would likely become larger than half a million decisions. Good computers might still be able to handle this but for creating small and cheap in-car routing devices this is a significant drawback. For this group we must conclude that FP-HXC is the overall best algorithm. It performs as good as the other two while using less information, memory and time. One does not even need an on-board routing device for applying FP-HXC, which is a large advantage. AP-HXL does not outperform it in our experiments, despite the fact that it receives live data updates, although it performs similarly well in advice quality within reasonable time.

6.4.1 Overall performance

When looking at the overall results, an interesting observation is that the best performing algorithm is a policy algorithm in the no live data group, an adaptive path algorithm in the partial data group and a fixed path in the full data group. In the first group little is known about the traffic conditions, therefore computing decisions for a set of possible states is a good idea. When only partial current or live data is available, the available information is not sufficient to find the best fixed path already. Live updates are quite valuable for the partial data group since the vehicle drives along different decisions points, the live data updates provide information about different road segments for each query because the given information depends on the location of the vehicle. The combination of historical data and some live data is good enough to simply update the single best computed path, while making good policies with partial live data that is not known before departure is a bit harder. For the full data group, the combination of historical information and full information about

the network state close before departure is so powerful that a smart fixed path algorithm already performs very well. With such an abundance of information, the best adaptive path algorithms and policy algorithms perform similarly well.

All these observations finalize our answer to the first research subquestion about the strengths and weaknesses of the various routing algorithms. We also answered the third subquestion about how important the availability of certain data is. Combining historical data with current or live data gives a significant improvement over using live data only. Because the best algorithms from the no live data group and the partial data group outperform algorithm AP-L, we can even conclude that historical data is more important than live data. Regarding our second subquestion we found out that combining historical data with live data using linear interpolation produces good results. Routing on chosen historical scenarios similar to the current conditions based on the arrival time at intermediate nodes also performed surprisingly well. The fact the algorithms using these methods perform better than the algorithm that uses only live data means that historical data is used in a meaningful way.

The largest threat to the scalability and practical usability of the best performing algorithms is the setup time, especially when the historical data first has to be imported. For the Dutch main roads test the setup time for the best algorithms then lies around 15 seconds. Table 6.2 makes clear that using historical data leads to better routing advice. So if we want to be able to deliver better routing advice in the future, then we should continue in the path of using historical data and try to come up with ways to reduce the runtime, for example by using different hierarchical levels within the network such that small roads far from the departure or destination location are filtered away which effectively reduces the network size.

6.5 Answering the main research question

Our main research question is whether traffic forecasting based on historical data can improve routing advice and by how much. We can now conclude that the main answer is yes since a lot of algorithms performed significantly better than algorithm AP-L, which is the best algorithm that does not use historical data. We saw that routing based on the conditions at the moment of departure performed 13% worse than optimal and adaptive path algorithm AP-L improves this further to 4.6% above optimal by using live data. The best algorithms that combine historical data with recent or live data perform around 2.5% above optimal, which is a smaller improvement but still a very significant one. The fact that the adaptive path algorithm AP-HL' performs much better than AP-L, 2.8% above optimal, stresses the importance of using historical data since it does so despite receiving only partial live data. Algorithm FP-HXC also outperforms AP-L despite receiving only recent data just before departure and no live data. Furthermore, the best algorithms in the no live data group also reach a performance around 2.5% above optimal, which means that even without recent or live traffic data AP-L can be outperformed by using historical data in combination with the arrival time at intermediate locations once they are reached. This means that using historical data is in fact more important than receiving live traffic data updates. Besides, accurate live

data is harder to obtain than historical data.

6.6 Looking beyond our experiments

In Section 6.4 we drew some overall conclusions about the performance of our routing algorithms. However, we performed a limited amount of tests in which certain variables were quite restricted. We concluded that for the Rotterdam test results having full live data is of very limited additional value when full data from close before departure is available as well as historical data. This could be caused by the fact that the major decision location in the Rotterdam test trip is rather close behind the starting location. Once this decision point is passed, little correction can be applied to the advised route when live data is received. Therefore, we expect that the best adaptive path algorithms and policy algorithms might outperform the best fixed path algorithms in practice now and then, depending on the network structure between the start and end location of a trip. This is especially true in case of unforeseen traffic conditions. Suppose that after departure a traffic accident happens on the currently advised route. An adaptive path algorithm or policy algorithm could in theory learn on time about this and guide the vehicle around the congested area, while fixed path algorithms would certainly fail to do so. Therefore fixed path algorithms are less robust in practice. So if full live data is available, then AP-HXL might be preferred over FP-HXC. At least FP-HXC is a worthy opponent, by comparing future adaptive algorithms to this algorithm one can test by how much live data improves the routing advice.

In our experiments, all routing algorithms made use of routing networks while this is not strictly necessary. Most algorithms would not be significantly faster when operating on original road networks, since the amount of provided data would be a lot larger and most policy algorithms would compute a policy that is so much larger that it might cause memory problems. However, some simpler algorithms can perform much faster when not using a routing network. FP can run about a factor hundred faster for the larger test networks since it only computes a shortest path once, while one hundred short paths were explored for building a routing network from the Dutch test networks. FP-C is as fast as FP, except that it has to read recent traffic information, which contains more data when it is provided for each road segment separately instead of for each subpath in a routing network. However, these algorithms were among the worst performing algorithms in our experiments. Filtering away irrelevant parts of the road network, given a certain start and destination location, becomes even more attractable when it only has to be performed once per start-destination pair since commuters perform the same trip a lot of times.

Chapter 7

Conclusions

In the previous chapters we defined a formal framework for the problem of vehicle routing. After defining a set of algorithms, we investigated the realism of traffic simulation software SUMO and discussed the problems that we encountered. In Chapter 6 we reached several conclusions about the different routing algorithms. We will summarize the most important results now and give some recommendations for future work.

7.1 Summary of main results

We obtained an answer to our main research question, which asks whether routing advice can be improved by predicting the future traffic state using historical data. The answer is yes, we showed that historical data can be combined with different types of recent or live data in such a way that a significant improvement is achieved compared to the best algorithm that does not use historical data, which is adaptive path algorithm AP-L that behaves similar to most modern in-car navigation devices. It receives accurate traffic speed data for the whole network each few minutes and performs a shortest path search each time new data is received. Its routing advices in our largest test set resulted in a travel time that was on average 4.6% higher than optimal. Several other algorithms that make use of historical data obtained a performance around 2.5% from optimality which is a significant improvement. Several of these algorithms receive only partial or no recent traffic data. We concluded that using historical data is even more important than receiving live data.

A set of twenty routing algorithms has been tested. The algorithms can be divided in three main types: fixed path algorithms, adaptive path algorithms and policy algorithms. Furthermore, a different classification in three types can be made based on the type of data available to them: only the arrival time at decision locations during the trip, spatially limited travel time data or full travel time data. We concluded that fixed path algorithms are rather simple and fast but they can perform pretty well when combining recent data with historical scenarios, as demonstrated by the results of algorithm FP-HXC. A major advantage is that fixed path algorithms do not need an in-car navigation device. This comes with a major drawback as well: they cannot adapt to unforeseen conditions, which can for example occur due to accidents. Policy algorithms are conditional plans that have the ability to adapt to

changing traffic conditions. Their advantage is that decisions can be made very fast during the trip, since they compute routing advice for all possible states before departure. A disadvantage is that the initialization time and the memory requirements are higher than for other algorithms. The policy algorithm versions that do not use live data also do not need an in-car device but they also cannot adapt well to unforeseen conditions. Adaptive path algorithms and policy algorithms that use live data can adapt to unforeseen conditions, adaptive path algorithms have a smaller setup time and are less memory-consuming. Adaptive path algorithms require more time than other algorithms for updating their advice during the trip, but the update times of all algorithms are far below one second, which is instantly from human perspective. The setup time of all algorithms is around one second or less for our largest test, which is low enough for practical usage. In our test framework importing and preparing the historical data is a bottleneck for the setup time of most algorithms. However, in practice historical data could be loaded once into a server and then a lot of trips can be performed without having to load the historical data each time.

We now briefly explain how our best performing algorithms work. The fixed path algorithm FP-HXC receives the latest traffic data for the whole network just before departure. It then compares the data to received data for traffic days from the past and extracts a set of the few most similar historical scenarios. It then averages the travel times of the road segments for these similar historical scenarios and then computes the shortest path through the resulting time-dependent network. Algorithm AP-HXL also performed well, it is the live update variant of FP-HXC. This means that it repeats the process of finding similar historical scenarios and computing the shortest path each time new traffic data is received. It can therefore adapt its routing advice during the trip that it is guiding. In case only traffic data for close-by road segments is available with live data updates, algorithm AP-HL' performs best in our experiments. It finds the historical scenario that is most similar to the latest available live data. For close-by roads it then uses the live data and for the other roads it uses data from the chosen historical scenario to compute routing advice. Finally, when no traffic data is available apart from learning during the trip at what time each decision location is reached, then policy algorithms P-HIT and P-HXI can do a good job. Both algorithms regard each historical scenario as one possible state. P-HIT computes the best decision at each decision location for each historical scenario separately, the arrival time at each node is rounded to a five minute time interval. Since the rounded arrival time at certain locations can be the same for multiple scenarios, the final decision for each location-time pair is determined by majority vote. P-HXI also computes the best option per decision location for each scenario, but now the arrival times are not rounded. On the moment a certain decision node is reached, the algorithm finds a fixed number of scenarios, depending on the amount of historical data provided, for which the arrival time is most similar to the query time. A majority vote over the decisions computed for these scenarios is then used to determine which road should be chosen next.

We explained that using traffic simulations offers some major advantages. Simulations can be used to produce network-wide traffic data from limited real data. Another major advantage is that simulations can be used to make a prediction of future traffic conditions in which temporal and spatial dependencies are automatically satisfied, provided that the simulations are sufficiently realistic. We motivated our choice for simulator SUMO and

provided realistic vehicle parameter choices based on real data. We investigated the realism of the simulations and discovered several problems that together make the simulations not very realistic, although performing experiments on them still leads to useful results and proofs of concepts.

7.2 Future work and recommendations

There is a lot of room for improvement regarding the realism of traffic simulation software, which is at least true for SUMO. By reaching a certain level of realism, results of tested routing algorithms are more certain to reflect their performance in the real world. Vehicles act very individually and therefore their interaction can be improved. In reality vehicle flows on highways can usually merge well without losing much speed, while the vehicles in the simulation often come to a halt at the end of a highway on-ramp, waiting for a safe gap to enter the highway. Further improvements could be made in the lane changing model and in the DFRouter module that is used to generate traffic based on loop detector data. The hardest part is likely to generate realistic traffic from spatially limited loop detector data, although switching to floating car data is likely the better option once this data type reaches good network coverage.

One of the major advantages of using simulation software is that the future traffic state can be predicted by running a simulation in multiple real-time, starting from recent data. Spatial and temporal correlations between traffic on nearby roads are then included automatically. Because SUMO runs several times slower than real time when simulating large regions or whole countries, we did not step further in this direction but instead we predicted future traffic conditions by combining historical data with recent or live data. However, if realistic and fast simulations are available now or in the future, then traffic predictions might become more accurate such that the quality of routing advice can be improved further.

From the different algorithm versions we tested it became clear that routing algorithms should treat each historical input scenario as one independent state, so no states should be considered in which travel times from one scenario are combined with data of other scenarios. Spatial dependency within traffic conditions is then automatically taken into account, it is a way to capture regular traffic patterns. When algorithms have access to both live data and historical data, the live data can be used to find similar historical traffic conditions, which on their turn can be used to predict how the current traffic conditions will develop in the near future. Interpolation between live and historical data can be used to ensure that the role of live data becomes smaller when planning further into the future. In most cases it is a good idea to compare live data to multiple historical scenarios instead of one to obtain more robust advice. The idea of rounding arrival times during the computation to reduce the computation time and policy size of policy algorithms is useful for reducing the policy size. However, rounding can be avoided when the state space and runtime is already well constrained.

In the search for good routing algorithms, adaptive path algorithms are more promising than policy algorithms since the latter category is quite time and space consuming. However, policy algorithms are useful when no live traffic data is available. Advanced fixed path

algorithms can perform very well, but lacking the ability to adapt to unforeseen traffic conditions is a major drawback. The adaptiveness of the routing algorithms that we introduced could be tested further using experiments in which unusual traffic conditions occur, such as traffic accidents. We expect the adaptive path algorithms to have an edge over fixed path algorithms for such situations. Although we tested a large set of algorithms with different properties quite thoroughly, further tests can be performed to obtain even more reliable results.

Another aspect to investigate further is whether routing algorithms can perform well with less historical data or whether this data can be stored and imported faster, since the time and memory required for using historical data can become a major part of the runtime of routing algorithms. This problem might be solved by having a constantly online server that imports historical data once in a few weeks or months and keeps track of live data. Navigation devices could then make queries to this server, communication is required anyway for adaptiveness of routing algorithms. Another data-related topic for further research is the effect of live traffic data being not completely accurate.

Another important line of future work is testing how the algorithms perform when a significant part of all vehicles uses a certain routing algorithm to determine their route. The performance will likely be much worse since our algorithms are designed under the assumption that the traffic conditions do not depend on the given routing advice. Some concepts and ideas that we applied might be useful when creating algorithms for optimizing the travel time of all vehicles together. Historical data can still be used to predict today's traffic conditions such that distributed advice can be given to avoid roads being used above capacity.

Bibliography

[1] R. Akçelik and M. Besley. Acceleration and deceleration models. In *23rd Conference of Australian Institutes of Transport Research (CAITR 2001), Monash University, Melbourne, Australia*. IEEE Computer Society Press, 2001.

[2] J. E. Baerwald. *Traffic engineering handbook*. ITE, 6th edition, 2009. (ISBN 193345234X).

[3] R. E. Bellman. On a routing problem. Technical report, 1958.

[4] C. R. Bennett. Modelling driver acceleration and deceleration behaviour in New Zealand. *ND Lea International, Vancouver*, 1994.

[5] Centraal Bureau voor de Statistiek. Verkeersprestaties motorvoertuigen; kilometers, voertuigsoort, grondgebied, 2013. URL `http://statline.cbs.nl/StatWeb/publication/?DM=SLNL&PA=80302ned&D1=0&D2=a&D3=l&HDR=T&STB=G1,G2&VW=T`.

[6] T. Chang, L. K. Nozick, and M. A. Turnquist. Multiobjective path finding in stochastic dynamic networks, with application to routing hazardous materials shipments. *Transportation Science*, 39(3):383–399, 2005.

[7] A. Chen and Z. Ji. Path finding under uncertainty. *Journal of Advanced Transportation*, 39(1):19–37, 2005.

[8] B. Y. Chen, W. H. K. Lam, A. Sumalee, and Z. Li. Reliable shortest path finding in stochastic networks with spatial correlated link travel times. *International Journal of Geographical Information Science*, 26(2):365–386, 2012.

[9] B. Y. Chen, W. H. K. Lam, Q. Li, A. Sumalee, and K. Yan. Shortest path finding problem in stochastic time-dependent road networks with stochastic first-in-first-out property. *IEEE Transactions on Intelligent Transportation Systems*, 14(4):1907–1917, 2013.

[10] S. I. Chien and C. M. Kuchipudi. Dynamic travel time prediction with real-time and historic data. *Journal of Transportation Engineering*, 129(6):609–616, 2003.

[11] R. Chrobok, J. Wahle, and M. Schreckenberg. Traffic forecast using simulations of large scale networks. In *Intelligent Transportation Systems Conference*, pages 434–439. IEEE, 2001.

[12] K. L. Cooke and E. Halsey. The shortest route through a network with time-dependent internodal transit times. *Journal of Mathematical Analysis and Applications*, 14(3): 493–498, 1966.

[13] B. C. Dean. Shortest paths in fifo time-dependent networks: Theory and algorithms. *Technical report, MIT*, 2004.

[14] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[15] G. Duncan and J. K. Littlejohn. High performance microscopic simulation for traffic forecasting. In *Computing and Control Division Colloquium on Strategic Control of Inter-Urban Road Networks, London, England*, 1997.

[16] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[17] L. Fu. An adaptive routing algorithm for in-vehicle route guidance systems with real-time information. *Transportation Research Part B*, 35(8):749–765, 2001.

[18] S. Gao, E. Frejinger, and M. Ben-Akiva. Adaptive route choices in risky traffic networks: A prospect theory approach. *Transportation Research part C*, 18(5):727–740, 2010.

[19] Google Maps. URL `http://maps.google.com`.

[20] R. W. Hall. The fastest path through a network with random time-dependent travel times. *Transportation Science*, 20(3):182–188, 1986.

[21] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4 (2):100–107, 1968.

[22] F. Hernandes, M. T. Lamata, J. L. Verdegay, and A. Yamakami. The shortest path problem on networks with fuzzy parameters. *Fuzzy Sets and Systems*, 158(14):1561–1570, 2007.

[23] Z. Ji, Y. S. Kim, and A. Chen. Multi-objective α-reliable path finding in stochastic networks with correlated link costs: A simulation-based multi-objective genetic algorithm approach (SMOGA). *Expert Systems with Applications*, 38(3):1515–1528, 2011.

[24] S. Krauß. *Microscopic modeling of traffic flow: Investigation of collision free vehicle dynamics*. PhD thesis, Universität zu Köln, 1997.

[25] A. K. Maurya and P. S. Bokare. Study of deceleration behaviour of different vehicle types. *International Journal for Traffic & Transport Engineering*, 2(3):253–270, 2012.

[26] J. W. C. van Lint, S. Hoogendoorn, and H. J. van Zuylen. Accurate freeway travel time prediction with state-space neural networks under missing data. *Transportation Research part C*, 13(5):347–369, 2005.

[27] C. Qian, C. Chan, and K. Yung. Reaching a destination earlier by starting later: Revisited. *Transportation Research part E*, 47(5):641–647, 2011.

[28] H. Rakha, I. El-Shawarby, M. Arafeh, and F. Dion. Estimating path travel-time reliability. In *Intelligent Transportation Systems Conference, 2006. ITSC'06. IEEE*, pages 236–241. IEEE, 2006.

[29] F. Riguelle. Nationale gedragsmeting "snelheid op autosnelwegen" - 2011. Belgisch Instituut voor de Verkeersveiligheid - Kenniscentrum Verkeersveiligheid, Brussels, Belgium, 2012.

[30] SUMO - user documentation. URL `http://sumo-sim.org/userdoc/`.

[31] K. Sung, M. G. H. Bell, M. Seong, and S. Park. Shortest paths in a network with time-dependent flow speeds. *European Journal of Operational Research*, 121(1):32–39, 2000.

[32] E. I. Vlahogianni, J. C. Golias, and M. G. Karlaftis. Short-term traffic forecasting: Overview of objectives and methods. *Transport Reviews*, 24(5):533–557, 2004.

[33] M. P. Wellman, M. Ford, and K. Larson. Path planning under time-dependent uncertainty. In *Proceedings of the eleventh Conference on Uncertainty in Artificial Intelligence*, pages 532–539. Morgan Kaufmann, 1995.

[34] M. Woodrow and P. E. Poplin. Acceleration of heavy trucks. In *4th Annual Southeastern Reconstruction Conference, 2002, Charleston, South Carolina*.

Appendix A

Routing algorithms pseudo code

The workings of the different routing algorithms that we tested are explained in Chapter 3. To give insight in the algorithms we now present pseudo-code for a few important algorithm parts. First we will provide code for the subroutine dijkstraBound that we introduced in Section 3.5. After that we will present code for a few routing algorithms.

Algorithm 1: Dijkstra algorithm that computes the arrival time for each node.

```
Function dijkstraBound(routing network r, time t, scenario set I, index i)
    result ← { }
    for each node n in r do
        n.time ← ∞
    s ← start node of r
    s.time ← t
    pqueue ← (s)
    while pqueue not empty do
        currentNode ← remove node from pqueue with lowest time
        currentTime ← currentNode.time
        result.put( (currentNode : currentTime) )
        for each path p outgoing from currentNode do
            nextNode ← end node of p
            if i < 1 then
                pathTime ← shortest travel time from I for p at currentTime
            else if i > size of I then
                pathTime ← longest travel time from I for p at currentTime
            else
                pathTime ← travel time from i ∈ I for p at currentTime
            newTime ← currentTime + pathTime
            if newTime < nextNode.time then
                pqueue.remove(nextNode)
                nextNode.time ← newTime
                pqueue.add(nextNode)
    return result
```

The dijkstraBound algorithm uses a priority queue to keep an ordered list of nodes to visit, the nodes are ordered on their 'time' parameter. The given scenario index determines what the algorithm has to compute. Remember that there are usually multiple historical scenarios available to the routing algorithms so there are multiple possible travel times per edge. For a negative input value the lowest travel time is taken for each edge, so this algorithm then computes an overall lower bound for the arrival time at each node. For an index larger than the number of scenarios the longest travel time per edge is used. In this case the algorithm computes an upper bound for the arrival time at each node, under the assumption that detours are not allowed so the vehicle is supposed to follow the advice of the routing algorithm. For any other scenario index the travel times from that scenario are used, in this case the arrival time at each node is the expected arrival time for the traffic conditions of that scenario. This function is used by a lot of policy algorithms, it will be referred to in the pseudo-code of the policy algorithms below.

Now we will look at how algorithm P-HIT computes its policy, the pseudo-code is shown in Algorithm 2. For each provided historical traffic scenario the following procedure is followed. Visiting all nodes is done from the destination back to the start, this has to be done only once if there are no loops in the routing network. For each node the expected travel time to the end is computed for each outgoing subpath. This is done by calculating the travel time of the subpath and adding the earlier computed travel time to the end from the end node of this subpath. The lowest expected time is stored and the subpath that corresponds to it. This is performed in a dynamic programming way, this only works if there are no loops in the routing network. If there is a loop, then at some point a subpath is visited such that the travel time from the end of the subpath to the destination has not been computed yet. In such a case, the option to travel via this path is not taken into account, but at the end the enclosing while loop will be repeated. It is repeated maximally three times, after a few runs likely all outgoing paths for each node are included in the computed routing decisions.

Method 'getTravelTime' computes a time-dependent travel time for a subpath with the method described in Section 2.1.1. The whole procedure is performed for each historical traffic scenario. This results in a policy with one routing decision per stored path-time-scenario tuple. At the end method 'aggregateResults' is applied, which stores one decision for each unique path-time pair using majority voting, since each pair can occur in combination with different scenarios.

Now we will take a look at algorithm P-HXLIT, the pseudo code for computing the policy is shown in Algorithm 3. The algorithm computes a lower and upper bound for the arrival time at each node, based on all historical data. It then goes through the network from end to start once or a few times in case of loops in the routing network, in order to compute the best decision per node per scenario per relevant time interval. After the policy is created, each time a decision location is reached P-HXLIT will find a set of similar historical scenarios. It then looks up the decisions stored for the given location and time for the similar historical scenarios and it determines the best choice by majority vote.

The last algorithm we discuss is AP-HXL. This adaptive path algorithm receives the average vehicle speed for the whole network each five minutes. Each time a decision point is reached the function below will be executed. In case no new information is received after

Algorithm 2: Policy algorithm P-HIT's method for computing the policy.

```
Function P-HIT-computePolicy(routing network r, time t, scenario set I)
    nodeVisitOrder ← r.getVisitOrder()
    result ← { }
    for i from 1 to size of I do
        nodeArrivalTimes ← dijkstraBound(r, t, I, i)
        timeToEnd = { (end node of r : 0) }
        finished ← false
        while not finished and not passed 3 times already do
            finished ← true
            for each node n in nodeVisitOrder do
                t ← nodeArrivalTimes.get(n)
                t̄ ← round t to closest five minute point
                bestExpectedTime ← ∞
                bestNextPath ← null
                for each sub path p outgoing from n do
                    remTime ← timeToEnd.get(end node of p)
                    if remTime = null then
                        finished ← false
                        continue
                    pathTime ← getTravelTime(p, t, I, i)
                    expectedTime ← pathTime + remTime
                    if expectedTime < bestExpectedTime then
                        bestExpectedTime ← expectedTime
                        bestNextPath ← p
                timeToEnd.put( (n : bestExpectedTime) )
                result.put( ([n, t̄, i] : bestNextPath) )
    return aggregateResults(result)
```

the previous decision point was reached, the computed shortest path from the previous call contains the necessary information for making the next decision, although the pseudo-code does not show that the whole path to the destination is stored. When new information is received, an A* algorithm is applied for finding the shortest path. This is done using a priority queue that sorts the reachable nodes on their heuristic time which is computed as the time for reaching it plus a heuristic time to the destination node, computed as the straight distance to the destination divided by the highest allowed speed in the network. Each time a node is visited the shortest time for reaching any of its neighbours is updated. The time for traversing any subpath is computed as a trade-off between the travel time prescribed by the live data and by similar historical scenarios, depending on the time span between the time stamp of the live data and the time of entering the subpath. The contribution of live data decreases when the time between reaching a subpath and the time stamp of the latest live data increases.

At the start of the shown algorithm a call to obtain similar scenarios is made. How

Algorithm 3: Policy algorithm P-HXLIT's method for computing the policy.

```
Function P-HXLIT-computePolicy(routing network r, time t, scenario set I)
    nodeVisitOrder ← r.getVisitOrder()
    result ← { }
    lowerBounds ← dijkstraBound(r, t, I, -1)
    upperBounds ← dijkstraBound(r, t, I, size of I + 1)
    end ← end node of r
    for each instance i in I do
        for each interval dt between lowerBounds.get(end) and upperBounds.get(end) do
            timeToEnd.put( ([end,i,dt] : 0) )
    finished ← false
    while not finished and not passed 3 times already do
        finished ← true
        for each node n in nodeVisitOrder do
            lowerBound ← round to time interval lowerBounds.get(n)
            upperBound ← round to time interval upperBounds.get(n)
            for each time interval dt between lowerBound and upperBound do
                for each instance i in I do
                    bestExpectedTime ← ∞
                    bestNextPath ← null
                    for each sub path p outgoing from n do
                        pathTime ← getTravelTime(p, dt, I, i)
                        pEnd ← end node of p
                        timeAtpEnd ← round to time interval (dt + pathTime)
                        remTime ← timeToEnd.get([pEnd,i,timeAtpEnd])
                        if remTime = null then
                            finished ← false
                            continue
                        expectedTime ← pathTime + remTime
                        if expectedTime < bestExpectedTime then
                            bestExpectedTime ← expectedTime
                            bestNextPath ← p
                    timeToEnd.put( ([n,i,dt] : bestExpectedTime) )
                    result.put( ([n,i,dt] : bestNextPath) )
    return result
```

similarity of historical scenarios is measured is explained in Section 3.3.4. In the last for-loop a method 'getTravelTimes' is called, which computes the travel time of a given subpath of the network using the method of Section 2.1.1 for multiple scenarios separately. Upon completion it returns a list with the travel time per similar historical scenario, these times are then averages by the algorithm.

When the end node is visited the shortest path is reconstructed. Each time a faster path to any node was found during the process, the investigated subpath ending in that node was

Algorithm 4: Adaptive path algorithm AP-HXL, this code shows what the algorithm does each time a decision point is reached.

```
Function AP-HXL-updateDecision(routing network r, node n, time t, scenario set I, traffic
info f, interpolation time span span)
    if no new data received since previous call then
        return decision from previously computed shortest path
    S ← getSimilarScenarios(t)
    cameFrom ← { }
    nodeTimes ← { }
    for each node v in r do
        nodeTimes.put( (v : ∞) )
        v.time ← ∞
    nodeTimes.put( (n : t) )
    n.time ← t + n.getHeuristicTimeToEnd()
    pqueue ← (n)
    while pqueue not empty do
        currentNode ← remove node from pqueue with lowest time
        if currentNode = end node of r then
            backtrackNode ← currentNode
            backtrackPath ← null
            while backtrackNode not equal to n do
                backtrackPath ← cameFrom.get(backtrackNode)
                backtrackNode ← start node of backtrackPath
            return backtrackPath
        currentTime ← nodeTimes.get(currentNode)
        for each path p outgoing from currentNode do
            pEnd ← end node of p
            pathTimes ← getTravelTimes(p, currentTime, I, S)
            pathTime ← average(pathTimes)
            pathTimeLive ← f.getTime(p)
            factor ← min(1, (currentTime - f.getInfoTime()) / span)
            newTime ← currentTime + factor*pathTime + (1 - factor)*pathTimeLive
            if newTime < nodeTimes.get(pEnd) then
                pqueue.remove(pEnd)
                nodeTimes.put( (pEnd : newTime) )
                pEnd.time ← newTime + pEnd.getHeuristicTimeToEnd()
                cameFrom.put( (pEnd : p) )
                pqueue.add(pEnd)
```

stored. So from the end node we can look up which subpath leading to it belongs to the shortest path. The same request can be made for the start node of this subpath. In this way we can backtrack until we reach the decision node for which the function was called. The function now knows the shortest path to the end, it returns the best subpath from the current location.

Appendix B

Experiments framework

In this chapter we provide insight in the framework that was made for implementing and testing the routing algorithms, its functionalities will be discussed and its structure is made clear using diagrams. We introduce a few possible extensions to the possibilities of the framework. We also provide some guidelines for implementation and indications of the effort needed to add them.

B.1 Framework overview

A framework is made in Java from which several kinds of tasks can be performed. The framework can make calls to SUMO or Osmosis including all necessary parameters whenever needed. It stores information about known file locations and necessary paths so that the user does not have to provide the same information on each run. The main functions that can be performed are listed below. The first four functions are visualized in Figure B.1, the next two in Figure B.2 and the last four in Figure B.3.

- Filter OSM data of a road network using Osmosis.
- Create a SUMO network from OSM data using Netconvert.
- Load a SUMO network into the framework as a graph structure.
- Map loop detectors to the network.
- Convert NDW data for a chosen time range to traffic files using DFRouter.
- Create a distribution of vehicle types using probabilities.
- Run a simulation for a traffic scenario with SUMO.
- Validate the simulation and output error information.
- Run a limited correction algorithm based on a produced validation.
- Run a set of routing algorithms on a set of test scenarios.

- Run simulations in which intervention on individual vehicle level is possible.

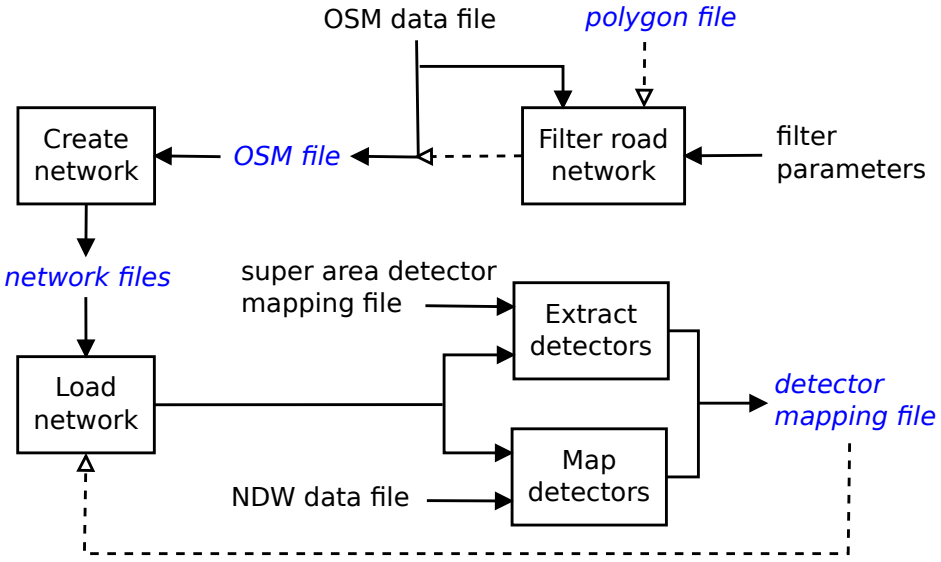


Figure B.1: Flow diagram showing how to get from an OSM data file to a network with detectors.

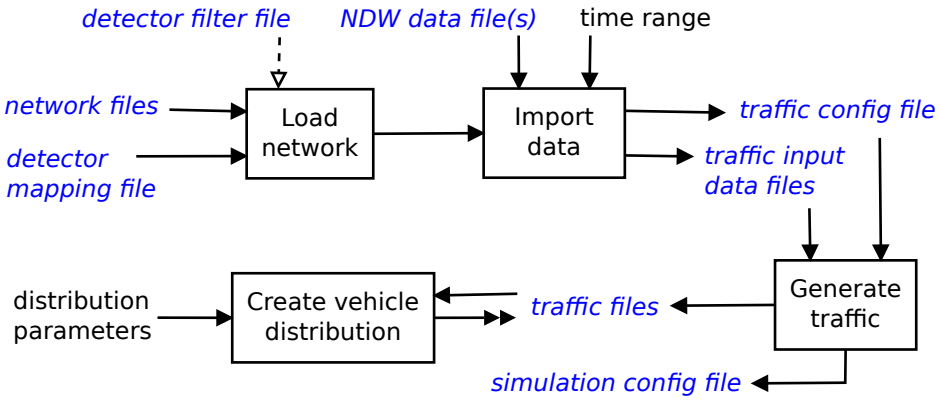


Figure B.2: Flow diagram showing how to prepare a simulation from NDW data.

The framework has a main class from which all functionalities are managed. Each functionality requires certain input files and they might depend on the results of other functions. Figures B.1, B.2 and B.3 show the dependencies between the modules and which input they require. In these diagrams, solid incoming arrows represent required input and dashed arrows are optional input. When two arrows merge then only one of them needs to be fulfilled in order to satisfy the module where the joined arrows point to. An incoming arrow with a double tip indicates that it updates the file it points to, so it is not a required input to create

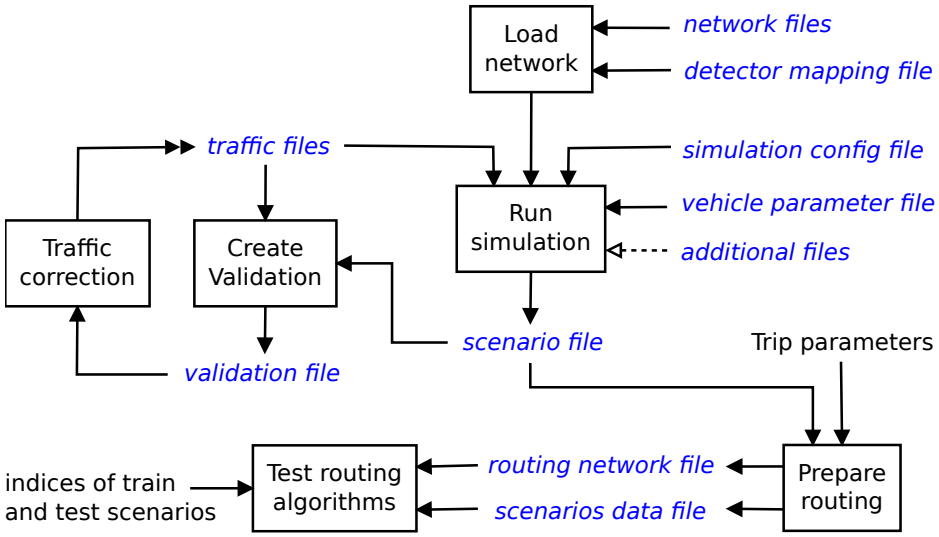


Figure B.3: Flow diagram showing how to run simulations, validate and correct them and how to test routing algorithms using generated scenarios.

that file. Blue names in italics are files that are remembered by the framework once they are provided or generated. So if a blue file name is known to the framework and it does not need to be updated then you can assume it to be already present, also when the framework is restarted before using the file again.

For the 'load network' module, providing a detector mapping file is optional but in order to generate traffic it is necessary. Therefore, providing the mapping file is shown as optional in Figure B.1 and as necessary input in Figure B.2.

The functions and dependencies required for loading a network and generating traffic on it are already described above. The framework uses a specific structure for storing files. In order to run any functionality a network directory must be set first. In this directory network topology files are stored as well as detector mapping information and parameter files for vehicles and roads. Within the network directory there can be multiple scenarios, each scenario has its own subdirectory. A scenario directory can contain NDW data files, generated traffic files and output of simulations. The framework stores configuration files that are used for running SUMO modules, these files give insight in the used parameters. The framework also creates own configuration files (*.cfg) for each network and each scenario, which it uses to remember files, dependencies and a few parameters.

B.2 Preparing and running a scenario

The first thing to do when one wants to run traffic scenarios is generating a network fitting one's needs. One can choose to download an OSM file and import it right away. However, the OSM file contains a lot of irrelevant information. In order to filter on a specific

region or filter out certain road types, some preprocessing is needed, Figure B.1 shows the preprocessing structure.

B.2.1 Filtering a road network

In order to filter on a region of any shape, one must first create a polygon file that contains a list of longitude-latitude pairs that define a polygon on the earth's surface. Here is a very basic example that shows the contents of a file that defines a simple polygon:

```
example
1
   4.305747986047   51.95988355698
   4.666923523143   51.87433183300
   4.278282165736   51.84549556020
END
END
```

The first line could be the file name, any string is allowed. The second up to the sixth line defines a triangular polygon. Multiple of such blocks can be put behind each other to define multiple polygons, increase the integer at the start of each block for subsequent blocks. The last line contains the sign that the file ends. Save the file with extension '.poly'. Tools for retrieving the correct polygon coordinates can be found on the internet. A lot of polygon files for countries or provinces can also be found online, as well as OSM maps of a lot of countries or continents, an overview of websites can be found via `http://wiki.openstreetmap.org/wiki/Planet_dumps`.

One can choose to download an OSM file of an area of the region of one's interest in normal OSM format or as a compressed file, for example with extension '.pbf' if you are planning on filtering it. The compressed OSM file and the polygon file can be given to the framework, as well as a parameter for filtering out certain road types. The framework then handles the call to Osmosis such that the filtering is applied. OSM data uses a lot of data types, the framework only distinguished the following types, ordered from large to small roads: highway, trunk, primary, secondary, tertiary, residential and other. Given a road type, the framework can filter away either all smaller roads or all larger roads. Alternatively, one can provide a ready-to-use OSM file to the framework directly. More information on how Osmosis is used for filtering can be found in Appendix C.

B.2.2 Loading detectors

In order to generate traffic based on loop detector data a set of detectors needs to be added to the network. There is a detector mapping file for the whole Netherlands that was used for our test scenarios. Given this mapping file, the framework can extract all the detectors that lie within the chosen network automatically. Alternatively, there is a module for mapping detectors to the network based on an NDW data file for which one needs to provide a NDW data file, which could be the same file that will be used to create the traffic from. The module reads the detectors from the NDW file, for each detector it reads the location and

it finds road segments in the network that are close-by. It proposes the road segment to which the detector should belong. The user can visually inspect it and confirm the proposed mapping or choose a different road segment to which the detector should belong from a list of close-by road segments.

When the detectors are extracted or mapped, the framework stores information about the detectors in a mapping file. This file can be fed back to the network loader to load the detectors.

B.2.3 Generating traffic

Once a network with detectors is loaded, traffic must be generated in order to run a simulation. Figure B.2 shows an overview of the required steps. NDW data files must be given to the framework, once these are set several simulations for different time frames can be run using these files. A time range must be provided that will be used for the simulation. The framework then creates a file with detector definitions, a file with the measurements of the detectors and a configuration file for a DFRouter command. DFRouter is then run using these files, which results in a file with routes, a file with vehicles and a file for displaying detectors. After this, the framework can be ordered to prepare the simulation, it then creates a configuration file for running the simulation as well as a file that tells SUMO what output it should produce during the simulation.

SUMO's module for generating traffic from loop detectors works with measurement input for one or two default vehicle types only: 'PKW' for normal cars and 'LKW' for trucks, other types are not possible. This is likely because most loop detectors do not distinguish between different vehicle types. For this reason, the framework creates a measurement file for only one vehicle type. After the file with all the vehicles is created by DFRouter, the framework can alter the type of each vehicle with a certain probability, such that a distribution of several vehicle types emerges. Each vehicle receives one of the types as defined in a vehicle parameter file that the user can alter manually. In the vehicle parameter file a distribution can be defined for each vehicle type. If we have for example a type 'truck' then we could define two subtypes in it, one for heavy trucks and one for trucks without load. A probability of occurrence can be defined such that a certain percentage of trucks is heavy. Each vehicle subtype can have it's own parameters, like maximum speed, acceleration etc. Further details about vehicle parameters are described in Section 4.4.2.

B.2.4 Run simulations

Once the necessary traffic files and a configuration file are created by the framework and the user provides a vehicle parameter file, a simulation can be run using SUMO. Figure B.3 shows the process schematically. Additional files can be set before running a simulation, usually for fixing the network topology. One could think of correcting the number of lanes for certain road segments or changing traffic light patterns.

While the simulation runs, SUMO creates an output file as requested by the framework. This output file contains per time interval of five minutes the average speed of all vehicles for each road segment in the network. Output files of simulations all have the same file

name except for one index number. The framework keeps track of the latest index number, so it remembers the number of output files that are present in the scenario folder. Each simulation output is called a traffic scenario because it represents a time-dependent set of average velocities per road segment. Multiple of these traffic scenarios can be loaded by the framework. One can then use some of them as test scenarios for routing experiments and others as training data for the routing algorithms.

B.3 Extension: interactive vehicle control

Sometimes one likes to intervene in the simulation of a scenario. For example if one wants to know the effect of a traffic accident on the traffic conditions. First, one could set up a scenario without this accident and run it in order to produce the traffic conditions to compare with. In order to run exactly the same simulation but then with the accident, SUMO's module TraCI can be used, which stands for traffic control interface. There is a version that can be used in a Java environment, called TraCI4J. The framework contains a module for running interactive simulations using TraCI4J, although this module is not used in our final experiments.

Using TraCI, a simulation is performed step by step, the default step size SUMO uses is one second. On any step in the simulation the user can read a lot of information, for example the position and speed of each vehicle. Several changes can be made on the individual vehicle level when running an interactive simulation. The most important possibilities are:

- Change the speed of a vehicle, which acts as an individual speed limitation. When the given speed is lower than the current speed of the vehicle, it will brake to reach the new speed. By giving value 0 the vehicle will stop and by providing a negative value any previously given limitation will be removed from this vehicle.
- Let a vehicle change lanes.
- Change the route of a vehicle. A vehicle can be rerouted or its route can be extended bit by bit.
- Add a new vehicle to the simulation at any location with a chosen speed. Insertion might be delayed until there is enough room for the new vehicle to be inserted safely.
- Remove a vehicle from the simulation, it simply disappears from the simulation instantly.

Step by step simulation and the possibility to control individual vehicles come at a cost: the simulation speed is lower than for normal SUMO simulations. The runtime heavily depends on the number of queries made. It is possible to bundle multiple queries to one large query, which is highly advisable for each time step in which more than a handful of queries is made. Example code for interaction with a SUMO simulation using TraCI4J can be found at the end of this appendix.

B.4 Extension: real-time input data

For all our experiments we used traffic data from the past to route on, meaning that the test conditions are taken from historical data. A nice extension would be to be able to run SUMO on live data. Altering a simulation each few minutes using new data is not easily achieved. The files used to run a simulation must be known to SUMO before starting a simulation. In principle, it might be possible to add new vehicles to the file containing all vehicle information while the simulation is running. In order to keep memory usage lower, the vehicle file is ordered chronologically on the departure time of vehicles. While simulating, SUMO loads vehicles into memory a certain time ahead. New vehicles could be appended at the end of the file, provided that reading and writing the file can be performed simultaneously.

One way of simulating reality is by simulating at real time speed, with a few minutes delay for receiving and processing real data. While SUMO performs the simulation for the latest available time interval, the data for the next interval can be prepared. When using loop detector data as input, a new measurement file can be created when new data becomes available. DFRouter must then be run via a different instance of SUMO to compute the vehicles for the new time interval, using the existing file with detector definitions and the new measurements file. A simple script could be written to copy the vehicles from the new vehicle file and paste them at the bottom of the vehicle file on which the simulation is running. The real data is then used to determine how many vehicles enter the simulation at any point in time. Meanwhile, the traffic conditions in the simulation can differ from reality, which is not directly corrected by new data. Furthermore, the simulation cannot make a future prediction because the detector measurements of the future are not available to the simulation. So the usefulness of this way of simulating is quite limited.

If we want to make predictions of future traffic states using simulations, we have to use heuristics to guess the future vehicle measurements based on historical data or the latest available live data. Suppose SUMO is running a simulation of reality using the latest real data. When new data comes in at a later point in time, then it will likely differ a bit from the traffic state in the simulation at that same time. Correcting the traffic state of the simulation means that the prediction of the future traffic state also changes. So in fact, correcting the traffic state comes down to starting a new simulation. Therefore, a second option is to start a simulation based on historical data and the latest live data and then run the simulation without further intervention. When new data becomes available, a new simulation can be started based on this newer information. In this way a few simulations might be running in parallel. The prediction for the traffic state for any future point in time is then updated with each next simulation that reaches that point in time.

One of the current problems with SUMO simulations is that the network is by default completely empty at the start of a simulation, which makes the start of each simulation unrealistic. However, it is possible to save the state of a simulation such that one can later load this state and continue simulating from there. So suppose that we start a simulation that runs for example from 4 AM to 5 AM, then we can store the simulation state at 5 AM. Suppose we want to predict the traffic conditions at 6 AM, then we can start a simulation using the saved state of 5 AM as the beginning condition and measurements between 5 and

6 AM are then used to determine the number of vehicles entering the network during the simulation. In this way the simulations start with an already populated network.

B.5 Extension: live topology modification

Another interesting extension is to be able to make small modifications to the road network. Suppose information about road maintenance is made known in prior and suppose that at a certain maintenance job one of the lanes of a highway is temporarily closed, or a complete road is blocked for all motorized traffic. It would be nice if this can be modelled in SUMO with only a simple adjustment, so without rebuilding the network.

Blocking entire roads is fairly easy. When using SUMO's module DFRouter to compute routes and vehicles, it is possible to add names of disallowed edges when providing the run options. No routes will be generated that use these edges. There is however a small problem: routes may be made that end at the disallowed road, resulting in vehicles that drive towards the disallowed road and disappear on reaching it. It is as if SUMO internally removes the edge, such that the road part upstream becomes a dead end. Whether this happens or not depends on the way the traffic is generated. When traffic is generated based on loop detectors, SUMO might send vehicles towards the disallowed road because there is no detector between the last junction and the disallowed road. One solution is to add a fake detector in front of the blocked road and store in the measurement file that it measures zero vehicles. Another solution is to remove all roads upstream of the disallowed road up to the first upstream junction.

Blocking only one lane is a bit more difficult. One option is to run an interactive simulation using TraCI, as explained in Section B.3. Using this it is possible to wait for a vehicle to enter the section of the lane that should be closed and let that vehicle halt for the rest of the simulation, thereby effectively blocking that lane. A bit smarter solution is to insert one or more extra vehicles at the blocked lane at the start of the simulation, giving them a maximum speed of zero. In this way multiple lanes can be blocked too. However, such an interactive simulation is slower and takes a bit more effort. A different solution is to manually alter the network file. One can look up the definition of the edge in the network file and simply remove one of the lanes. Changing the number of lanes of an edge can be done right before running a simulation, it does not have any effect on the generation of routes and vehicles. A disadvantage is that it involves manual editing of XML files, which is an error-prone process. Another disadvantage is that the number of lanes can only be changed for complete edges and not for a part of the edge. Splitting an edge in three parts in order to change the number of lanes for a specific section is a bit more work and requires more manual editing. The chance of making mistakes then becomes reasonable, we therefore discourage this solution.

Example code for using TraCI4J with SUMO

```
import it.polito.appeal.traci.*;

   public static void main(String[] args) {
      String pathSumo = "./sumo-0.20.0/bin/sumo";
      System.setProperty("it.polito.appeal.traci.sumo_exe", pathSumo);
      String fileConfig = "Networks/Rotterdam/Rotterdam.sumo.cfg";
      int seed = 12345;

      SumoTraciConnection conn = new SumoTraciConnection(fileConfig, seed);
      try {
         conn.runServer();

         // run the simulation up to the moment where we want to intervene
         for(int i = 0; i < 600; i++)
            conn.nextSimStep();

         // intervention example 1: add a (special) vehicle
         {
            String vType = "DEFAULT_VEHTYPE";   // vehicle type
            String vName = "myVehicle";         // identifier
            String routeId = "1_to_6";          // route for the new vehicle
            int lane = 0;                       // lane number (0 = rightmost)
            double position = 1d;               // offset on edge in m
            double speed = 20d;                 // depart speed in m/s

            AddVehicleQuery vhq = conn.queryAddVehicle();
            VehicleType type = conn.getVehicleTypeRepository().getByID(vType);
            if(type == null)
               throw new RuntimeException("VehicleType '" + vType + "' is not found");

            Route route = conn.getRouteRepository().getByID(routeId);
            if(route == null)
               throw new RuntimeException("Route '" + routeId + "' is not found");

            vhq.setVehicleData(vName, type, route, lane, position, speed);

            // run the query, so TraCI will try to insert the vehicle now
            vhq.run();

            // run the simulation until the vehicle is inserted successfully
            while(conn.getVehicleRepository().getByID(vName) == null)
               conn.nextSimStep();
         }

         // intervention example 2: simulate an accident
         {
            String roadId = "10";               // identifier of a SUMO edge
            int lane = 0;                       // lane number (0 = rightmost)
            double areaStartPos = 1000;         // accident area start, offset in m
            double areaEndPos = 1500;           // accident area end, offset in m
            int numberOfStepsBlock = 900;       // accident duration
```

```
            String accidentVehId = null;
            Edge blockEdge = conn.getEdgeRepository().getByID(roadId);
            Lane blockLane = conn.getLaneRepository().getByID(roadId + "_" + lane);
            while(accidentVehId == null) {
               // first check whether there is a vehicle on the lane
               if(blockLane.getLastStepVehicleNumber() > 0) {
                  Map<String, Vehicle> allVehicles =
                     conn.getVehicleRepository().getAll();
                  Iterator<Entry<String, Vehicle>> it =
                     allVehicles.entrySet().iterator();
                  while(it.hasNext()) {
                     Entry<String, Vehicle> entry = it.next();
                     Vehicle veh = entry.getValue();
                     Edge vehEdge = veh.getCurrentEdge();
                     int vehLane = veh.getLaneIndex();
                     if(vehEdge.equals(blockEdge) && vehLane == lane) {
                        // check whether the vehicle is on the right section
                        double lanePos = veh.getLanePosition();
                        if(lanePos >= areaStartPos && lanePos < areaEndPos)
                        {
                           // set the vehicle's speed limit to zero
                           veh.changeSpeed(0d);
                           accidentVehId = veh.getID();
                           break;
                        }
                     }
                  }
               }
               conn.nextSimStep();
            }

            // run the simulation for the number of steps we want to block the lane
            for(int i = 0; i < numberOfStepsBlock; i++)
               conn.nextSimStep();

            // get the vehicle via its ID and remove the speed limitation
            Vehicle blockVehicle = conn.getVehicleRepository().getByID(accidentVehId);
            if(blockVehicle != null)
               blockVehicle.changeSpeed(-1d);
         }

         conn.close();
      }
      catch (Exception e) { e.printStackTrace(); }
   }
}
```

Appendix C

Manuals

This appendix provides instructions for installing and using the programs that were used during this project, such as SUMO and Osmosis. We explain briefly how SUMO was installed and after that we show how networks can be imported and manipulated using Osmosis and SUMO. In our framework the calls to Osmosis and SUMO are handled automatically, the workings of the framework are described already in Appendix B. In this appendix we show typical calls to Osmosis and SUMO that our framework uses, to give insight in how Osmosis and SUMO can be used.

C.1 Installing SUMO

SUMO is open source traffic simulation software that is still under development. Newer versions come out from time to time, information about the latest stable release can be found via SUMO's website. To set up the project under Linux, go to `http://sumo-sim.org/wiki/Installing/Linux_Build` and follow the instructions. This includes installing some libraries to a manually chosen location. As the website instructs, find and download the source packages proj, gdal and fox. These libraries are not necessary for running SUMO, but they are required in order to make SUMO's visualization tool work. Extract each zipped download to a folder on your computer. For each of the packages, navigate in a terminal to its folder and run the following command:

```
> ./configure --prefix=[path-to-lib]/ && make install
```

This will put the required files in the folder 'include' and some other folders and files within a separate library directory. Because gdal depends on proj, this command should be performed for proj before performing it for gdal. It might be the case that errors occur because the versions of proj and gdal are not compatible with each other. Since gdal uses proj, it seems best to build a recent version of gdal and not the most recent version of proj, thus a version of gdal that came out later than the version of proj. The library versions we used are fox-1.6.49, proj-4.6.0 and gdal-1.10.0. Another package that you need is xerces-c. This is often part of the system already or can be installed via the package manager when

using Ubuntu. If this is the case, then the correct path to it should be included when building SUMO.

Download a source package of SUMO via `http://sumo-sim.org/userdoc/Downloads.html` and extract it to a location on your computer. Navigate to its folder in a terminal. If xerces-c is installed such that the 'include' folder is below the 'usr' folder, then the command for building SUMO is:

```
> ./configure --with-fox=[path-to-lib]/
              --with-proj-gdal=[path-to-lib]/
              --with-xerces=/usr/
```

You can also choose to build xerces-c in the same way as the other three packages. In that case you can use the following command to build SUMO:

```
> ./configure --with-fox=[path-to-lib]/
              --with-proj-gdal=[path-to-lib]/
              --with-xerces=[path-to-lib]/
```

When configuring SUMO the output ends with two lines showing which packages are enabled and which are not. When the installation succeeded, gui, proj and gdal should be mentioned as enabled. After the configuration is completed successfully, SUMO can be installed by running:

```
> make
```

C.2 Preprocessing a network using Osmosis

Osmosis is a useful tool for importing OSM networks and applying filters to it. Installing it is rather straightforward and will not be discussed here, more information can be found via `http://wiki.openstreetmap.org/wiki/Osmosis`. We now give an example call that shows how to run Osmosis from a terminal including some useful filters, paths are put in square brackets and example file names are used.

```
> [path-to-Osmosis]/bin/osmosis
     --rb file=[path-to-pbf]/network.osm.pbf
     --bp file=[path-to-poly]/network.poly
       clipIncompleteEntities=true
     --tf accept-ways highway=motorway,motorway_link,trunk,trunk_link,
       primary,primary_link,traffic_signals
     --tf reject-relations
     --un
     --wx [path-to-osm]/network.osm
```

The first line shows the call to the executable of Osmosis. On the second line a compressed OSM network is provided as input and on the third line a polygon file is given. Network files and polygon files can be found on several websites, an overview is given on

http://wiki.openstreetmap.org/wiki/Planet_dumps. The fourth line is an additional option for the polygon file, stating that roads lying partly outside the filtering area should be removed. The next line defines which road types should be kept, one can also choose to use 'reject-ways' instead. There are many types of entries, some other entries of important road types are: secondary, tertiary, residential, living-street and unclassified. There are more types for other means of transportation and for signs and buildings etc. The option 'un' removes nodes that become isolated due to the filtering process and rejecting relations means that a lot of information is removed that is not important for routing purposes. The last line tells Osmosis where to store the processed OSM file.

C.3 Parsing an OSM network using SUMO's Netconvert

In order to obtain real world road networks, first a map from OpenStreetMap should be downloaded. Documentation about how to prepare an OSM file for SUMO is available at http://sumo-sim.org/userdoc/Networks/Import/OpenStreetMap.html. A lot of useful preprocessing can already be done using Osmosis, which has more filter options than SUMO's module Netconvert. Netconvert is mainly used to read an OSM network and convert it to a network that can be used by other modules of SUMO. A minimal command for parsing a network only provides the OSM file and the path to the network file to write. Several parameters can be added to the Netconvert call, see http://sourceforge.net/apps/mediawiki/sumo/index.php?title=NETCONVERT for a list of possible options. We will now give an example of a typical Netconvert call that our Java framework uses when calling Netconvert:

```
> [path-to-SUMO]/bin/netconvert
     --junctions.join --junctions.join-dist 25
     --ramps.guess --ramps.ramp-length 180
     --roundabouts.guess
     --tls.guess --tls.join --tls.join-dist 40 --no-turnarounds.tls
     --geometry.remove
     --proj.plain-geo --plain-output-prefix [path-to-plain]
     --osm [path-to-osm]/network.osm
     -o [path-to-net]/network.net.xml
     -l [path-to-log]/netconvert.log
```

On the first line the Netconvert module is called. The second line joins junctions that are less than 25 meters away from each other. The third line adds a 180 meter long extra lane at highway on-ramps and off-ramps such that the ramps do not abruptly stop at the point where they are connected to a highway. The fourth line enables automatic roundabout guessing. The next line guesses which junctions are controlled by traffic lights. Traffic lights that are very close to each other receive a joined traffic light program using the second option. The last command on the line disables the possibility of making a 180 degree turn at signalized junctions, which to some extent prevents vehicles from receiving weird routes. Removing the geometry means that only nodes representing real junctions are kept; nodes that are

only meant to define the shape of a road are removed. Enabling this option does not remove the shape of roads since these are stored in a different parameter in the network file for each lane. The next line tells Netconvert to output separate files for the nodes, edges and connections of the network. The option at the start of the line states that geo-coordinates should be written in the plain output file that contains the node definitions. These extra output files are not used by SUMO but our framework uses it to import all nodes and edges more easily. On the last three lines, first the location of the OSM file is provided, then the location of the network file to write to and finally a log file that can be used to check whether any problems occurred during the process.

Once the network is generated, it can be viewed using SUMO's visualization tool. The command to do this is:

```
> [path-to-SUMO]/bin/sumo-gui -n [path-to-net]/network.net.xml
```

C.4 Creating traffic using DFRouter

When the network is parsed using Netconvert, traffic needs to be generated in order to run a simulation. We show an example call to SUMO's module DFRouter that is used for generating traffic, the command is similar to the calls made by our framework when preparing simulations. The extra extension between the last and second last dots of file names is usually not necessary.

```
> [path-to-SUMO]/bin/dfrouter
     --time-factor 1 --time-step 300
     -b 0 -e 3600
     --strict-sources
     --max-search-depth 100
     --disallowed-edges 12345,67890
     -n [path-to-net]/network.net.xml
     -d [path-to-det]/detectors.det.xml
     -f [path-to-msr]/measurements.msr.csv
     -o [path-to-rou]/routes.rou.xml
     --emitters-output [path-to-veh]/vehicles.veh.xml
     --detector-output [path-to-det]/detectors.det.xml
     --detectors-poi-output [path-to-poi]/detectors.poi.xml
     --validation-output [path-to-val]/validation.add.xml
     --validation-output.add-sources
     --error-log [path-to-log]/dfrouter.log
```

The second line defines the time conversion factor between the measurement data and the simulation, the time step defines the interval length in seconds between consecutive data in the detector measurements. The third line defines the begin and end time of the simulation in seconds, which defines which part of the measurements to use. The option 'strict sources' ensures that detectors can only become sources when all roads leading to them do not have a detector. Without this option detectors become sources when at least one road leading to them does not have a detector on it. The maximum search depth prescribed the maximum

number of consecutive road segments without detectors to follow before quitting the search branch when determining vehicle routes. The sixth line is the option that can be used to manually disallow vehicles on certain edges, edge names can be looked up using SUMO's visualization tool.

The next four lines provide locations of files. The first three provide the network, the file with detector definitions and the file with detector measurement values. The option 'o' defines to which file the computed routes should be outputted. The emitters output option specifies to which file all the generated vehicles should be written. The next line defines where the detector definitions are written including the computed detector types. One can put the same file there as the detector file that serves as input. A file with detector locations can also be written in a points-of-interest file. By opening the visualization tool with the network and this file, the detector locations and types can be inspected visually. The command to do this is given below. A validation of the number of vehicles passing at detectors in the simulation can be done when running simulations. In order to do so, the third last line tells DFRouter to produce an additional file that can be provided to the simulation run command later. The second last line can be used to also include the source detectors in the validation process.

```
> [path-to-SUMO]/bin/sumo-gui
     -n [path-to-net]/network.net.xml
     -a [path-to-poi]/detectors.poi.xml
```

C.5 Running a SUMO simulation

After a network is created and traffic is generated, a simulation can be run. An example command with some useful options is given below.

```
> [path-to-SUMO]/bin/sumo
     --lanechange.allow-swap
     -e 3600 --step-length 0.5
     -n [path-to-net]/network.net.xml
     -r [path-to-rou]/routes.rou.xml,[path-to-veh]/vehicles.veh.xml
     -a [path-to-out]/output.add.xml,[path-to-val]/validation.add.xml
```

The first line gives the command, one can also choose to use 'sumo-gui' to run the simulation in the visualization tool. The second line adds the option that two vehicles may swap when they want to switch to each others lane and got stuck while trying. In reality people usually manage to get to the correct lane before reaching the end of a junction, else they drive on via a different road. This option helps to avoid traffic getting stuck for no good reason, although swapping vehicles is of course not realistic. The third line defines the time span of the simulation in seconds and the time per simulation step, also in seconds. In the next two lines the network file is provided as well as the route file and the vehicle file. With option 'a' all other files that are relevant can be added, the order in which they are provided matters. If importing a certain file requires another file to be loaded, then this other file must be provided first. In the same way, the route file must come before the vehicle file in

the fifth line. The additional files are completely loaded before the simulation starts. The vehicle file can also be given as an additional file, but by giving it via the '-r' option it is loaded gradually during the simulation which reduces memory usage.

To let SUMO produce an output file that contains the average vehicle speed per road segment per time interval, a file describing which output to produce must be given as input to the simulation, in the example above this is the additional file 'output.add.xml'. The contents of this file should be something like the following:

```
<add>
    <edgeData id="dump_1" freq="300" excludeEmpty="defaults"
    withInternal="0" vTypes="car" file="output_1.out.xml"/>
</add>
```

The identifier is not really important, unique names must be used when multiple of such 'edgeData' entries are put in this file. The frequency parameter defines the time intervals in seconds over which the output should be aggregated. The next option ensures that when no vehicles are read for a certain road segment and time interval, then the maximum allowed speed is written. Internal edges can be skipped since they only define how vehicles should drive on junctions. With option 'vTypes' one can choose which vehicle types should be included in the output. By putting all normal car types here, the average speed of these vehicle types is computed and other vehicle types are not taken into account, which is a good choice when we want to compute routing advice for a regular car based on this simulation. If one wants to use multiple vehicle types, the types should be defined in a vehicle file, which is not included in earlier example calls but the file can usually be added as an additional input file. The 'file' parameter can be used to define to which file the results should be written.