ERASMUS UNIVERSITEIT ROTTERDAM

Master Thesis
Operational Research and Quantitative Logistics

Including Practical Information in Route Planning Applications

Author:
Nathalie van Zeijl
311210

Academic Supervisor:
Dr. D. Huisman
External Supervisor:
Drs. F. Paanakker
Co-reader:
Prof.Dr. A.P.M. Wagelmans

February 7, 2013

Keywords: Route Planning, Shortest Path, Time Dependency

Abstract

This research is about the effects on different shortest path algorithms of including several kinds of practical information. This practical information either concerns road network properties, like route types, or time dependent information, like traffic or weather information. Five algorithms are tested for this purpose: Dijkstra's algorithm, A*, ALT, CH, and CHALT. It turns out that for including road network properties CH, CHALT or ALT is the best to use, depending on de number of route requests. If time dependent information is also considered, the computation times for all algorithms increase. However, the computation times for CH and CHALT increase more than for the other algorithms, therefore ALT is considered to be the best algorithm for including practical information.

Acknowledgements

This thesis has been written in order to finish the master program Operations Research and Quantitative Logistics at the Erasmus University Rotterdam. It is the result of a nine months during internship at Backbone Systems B.V. in Amersfoort.

First of all, I would like to express the deepest appreciation to my supervisor, Dr. Huisman, who has guided and supported me through the whole process of my thesis. His suggestions helped me to improve my thesis and he also encouraged me to explore my own ideas in my research.

Secondly, I would like to thank my external supervisor F. Paanakker, founder and owner of Backbone Systems B.V. He arranged a challenging internship, recognized a suitable candidate in me and, despite his extremely full working schedule, found time to support and supervise me through the whole process.

Further, I also appreciate the National Data Warehouse for Traffic Information (NDW), for putting an effect in making a new data sample especially for this research.

I also would like to express my thanks to Joris Wagenaar, my partner, who supported and motivated me to get the best out of me, every day of the process. Additionally, he read my thesis several times and gave useful comment, which I also like to thank him for.

My thanks also goes to Tim Lamballais Tessensohn, who also was willing to read my thesis, gave useful comments, and would always listen to my ideas and struggles.

My final thanks goes to my sister, who also supported me during the process, and my parents who supported me throughout my whole study and created the opportunity for me to attend and complete this study.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Currently, if someone wants to travel to an unfamiliar place, it is common to use some kind of route planning. This could be on the internet before leaving, or with a navigation system during the trip. Regardless of whether it is business related, or just for private use, it is no longer possible to imagine a society without route planning applications. The task of such an application is to tell which route should be used for the desired trip. Based on the user's preferences, this route may be different. For example, the shortest route could be different from the fastest route. Given the road network and the preferences of the user, including the origin and destination of the trip, the application determines and presents the route that should be taken.

Due to the popularity of automated route planning, a lot of route planning applications have been developed in the past years. As a result, every producer tries to make his application better and more sophisticated than the competitor. Despite the progress that has been made, there is still much room for improvement. The applications need to become faster, they need to be used on smaller devices and the manageable planning area needs to become larger. The achievements of these goals itself has already inspired a plethora of research, but equally crucial could be to include more practical information in the planning.

Practical information could be any information concerning the circumstances of the route. For instance, if someone wants to travel the next day around five o'clock in the afternoon, there could be traffic congestion. This traffic congestion influences the route to be taken and the necessary travel time. Therefore it would be useful to not only incorporate current traffic information but also predictions about future traffic conditions. Another example is the effect of weather conditions. In case of a storm, parts of the selected route can become blocked, necessitating the application to search for alternative routes that are not blocked by either the storm or additional congestion. So also weather conditions, or weather predictions could be useful to include in route planning.

This research will focus on including five different kinds of practical information in route planning: information about *traffic congestion, weather conditions, occurrences of events, road network properties and time dependent data.* To be more specific, for both traffic congestion and weather conditions this means real time information as well as predicted information.

Occurrences of events can be any traffic disturbing event. It could be an accident on the road, resulting in delay, or a football match causes a large number of spectators travelling to and from the stadium, which also gives traffic issues. Events are strongly related to traffic congestion, but of incidental nature and therefore it may be useful to also consider this kind of information.

Road network properties refer to additional information about the roads in the network that not time dependent. Examples are one-way streets, maximum speed, weight capacities and truck allowance. These kind of issues should be included too, because they give serious trouble if ignored.

The last practical information to be added is time dependent data. This concerns other, not yet covered, time dependent information like, timetables of ferries, time restrictions for trucks about road usage, or different speeds limits for day and night. Again, this information effects the planned route or the travel time, and should therefore be considered as well.

How route planning applications determine a route, depends on the algorithm behind it. The literature (see Chapter 3) provides numerous algorithms that can be used for route planning. Every algorithm has its advantages and disadvantages, especially regarding computation time, the handling of the included information, and time dependence. These algorithms tend to be slow, have storage issues, or if the problem becomes time dependent, parts of the algorithms cannot be used to ensure reliable travel times. Therefore, in this research different algorithms are compared with respect to practical information, execution time.

The research question is formulated as:

Research Question. *What is the best algorithm to plan a route, within reasonable time, considering the five different kind of practical information?*

This thesis is structured as follows. In the next chapter the problem description will be given, which defines the route planning problem and gives an outline of the problems caused by the practical information. Chapter 3 contains a review of relevant literature and in Chapter 4 the algorithms used for this research are explained in detail. Chapter 5 gives the adaptations that are proposed to handle road network properties and Chapter 6 the adaptations to manage the time dependency. Chapter 7 gives an overview of the data, used to evaluate the methods and Chapter 8 gives the results according to that data. We conclude with an advice and improvements for further research.

2. PROBLEM DESCRIPTION

Route planning applications plan a route from one point to another. This is done by algorithms, that essentially are algorithms to solve the shortest path problem. Therefore, we will first define the shortest path problem. Thereafter is discussed what problems are caused by the incorporating practical information and how we propose to overcome these problems.

2.1 Definition Shortest Path Problem

The shortest path problem is a well-known problem from Graph theory. Consider the directed graph $G(V, A)$, where V is the collection of nodes and A is the collection of directed arcs with cost c. Suppose it is desired to find the shortest path from source node s to target node t. This can be formulated as a Linear Programming (LP) problem as (Wolsey [28]):

$$\min \sum_{(i,j)\in A} c_{ij}x_{ij}$$

subject to

$$\sum_{j\in V^+\{i\}} x_{ij} - \sum_{j\in V^-\{i\}} x_{ji} = 1 \qquad \text{for } i = s \tag{2.1}$$

$$\sum_{j\in V^+\{i\}} x_{ij} - \sum_{j\in V^-\{i\}} x_{ji} = 0 \qquad \text{for } i \in V\backslash\{s, t\} \tag{2.2}$$

$$\sum_{j\in V^+\{i\}} x_{ij} - \sum_{j\in V^-\{i\}} x_{ji} = -1 \qquad \text{for } i = t \tag{2.3}$$

$$x_{ij} \geq 0 \qquad \text{for } (i, j) \in A \tag{2.4}$$

Where c_{ij} is the cost of arc (i, j), $V^+\{i\}$ the collection $\{j : (i, j) \in A\}$, $V^-\{i\}$ the collection $\{j : (j, i) \in A\}$ and $x_{ij} = 1$ if arc (i, j) is in the shortest (s,t)-path. The objective is to minimize the total cost of the path from s to t. Constraint (2.1) ensures that the path has no arc going into s, which means the path has to starts at s. In constraint (2.3) is ensured that the path has no outgoing arc from t, which means the path has to end in t. For all other nodes in the path (2.2) ensures that an equal number of arcs comes in and leaves from that node. The problem as formulated previously, can be extended with additional constraints concerning, for instance, properties of the graph.

In our research the graph represents a road network. We use the most common way to represent the road network: roads, or road parts, are represented by arcs, and junctions or connections between road parts, are represented by nodes.

2.2 Practical Information

In this research we adapt existing algorithms to handle five different kind of practical information: traffic congestion, weather conditions, occurrences of events, road network properties and time dependent data. Each gives difficulties to include in the algorithms.

The difficulties caused by traffic congestion, weather conditions and events occurrences are quite similar. Occurrences of each can affect the travel time of a route in a negative way. Additionally, time dependent data can also influence the travel time of a route.

At hours before and after the working day there is a larger probability for traffic congestion occurrences than for time between those periods. The results of traffic congestion are a lower driving speed and thereby an increase in travel time.

Consequences of bad weather conditions are that cars should drive with less speed in order to ensure safety, and traffic congestion can arise. Indirectly, bad weather conditions (or predictions for it) can also cause traffic congestion, due to the fact that more people prefer to travel by car instead of, for instance, by bike. Furthermore, in a very extreme case, weather conditions may even cause roads to be impassable, due to, for example, snowfall. Therefore, bad weather conditions can cause delays.

Events, like an accident, can also cause traffic congestion and thereby longer travel times. Weather conditions can be seen as a specific type of events, but since there is always some kind of weather, we consider it apart from events. Events are of incidental nature, and thereby makes them hard to predict. There is a distinction between two kinds of events. Events for which it is uncertain whether they will occur, like an accident, and events that are very likely to happen, like a football match, but without certainty about the effects on the travel time.

To the issue of time restrictions belongs, for example, the use of a ferry in a route. In this case, the route depends on the timetable of that ferry. If the ferry timetable is unfavorable, the travel cannot be continued and waiting is necessary. This leads to longer travel times.

To tackle the problems of increase in travel time we will express the arc costs in terms of travel time. This travel time is based on a travel time lower bound and additional time caused by the different issues described previously. The travel time lower bound of an arc is defined as the travel time based on the maximum speed and the distance of an arc. The travel time will be described by a linear combination of all different effects. Doing so, some of the original algorithms need to be adapted. Since the arc costs become time dependent, backward search gives problems. Moreover, the correctness of the preprocessing can also become questionable. The preprocessing should be done once, and the results are supposed to be used numbers of times. When arc costs become time dependent, the preprocessing either also become time dependent and has to be repeated more than before, or the results of the preprocess might not give correct results. In Chapter 6 we described the adaptations made for each algorithm separately.

Including additional route information, like turn restrictions, can prevent the planning from resulting in a route that is not allowed. For example, when the route planning tells to go right, but that is restricted by a turn restriction. In order to prevent this, we add restrictions to arcs that need to be checked during the route planning. As a consequence, it is no longer possible to plan routes that are not allowed. In Chapter 5 we explain how this is exactly done.

We will apply the algorithms with realistic data, to achieve a clear impression of the possibilities of the resulting algorithms.

3. LITERATURE

In this chapter we give a literature overview about the latest developments in the field of the shortest path problem. The overview is split into two parts. First, the existing shortest path algorithms will be discussed, followed by literature that relates to this research in terms of practical information that will be added to the problem.

3.1 Shortest Path Algorithms

For solving the shortest path problem, Dijkstra's algorithm (Dijkstra [9]) is the best-known algorithm. This algorithm starts at the source node s and expands to the node v, which is not expanded before, with the lowest cost from s to v. The expansions continue until the search reaches target t. At the end of the algorithm, the shortest path costs from s to all nodes in the search space are saved. In the case that no target would be given to the algorithm, the algorithm searches the whole graph. As a consequence, the algorithm will save the lowest cost of the paths from s to all nodes in the graph. Dijkstra's algorithm finds the optimal solution to the problem, but it is very slow for large network instances, because all nodes n with a cost smaller than the cost from s to t are explored.

Pohl [23] introduces the technique *bidirectional search*. It tries to reduce the calculation time and the search space of the algorithm. As the name suggests, this technique searches the graph in two directions: from the source to the target (forward) and from the target to the source (backward). In every iteration the direction in which to continue the search is decided. It is most common to just alternate between the forward and backward search. The solution is found if for both search directions common node v is expanded. Bidirectional search can be applied to Dijkstra's algorithm and is also used in various other shortest path algorithms.

Related research mainly focusses on improving Dijkstra's algorithm. The reason is that Dijkstra's algorithm is optimal, but can be improved in terms of computation speed. Over time, it turns out that the improvements of the last decades can be divided according to two main ideas. One idea is to focus on goal directed search. This means that the algorithm tries to search in such a way that each next node is closer to the target node. Doing so, the search space is reduced, less nodes need to be investigated, and therefore the algorithm becomes faster. On the other hand, more information is needed, so it uses more storage space. The other idea is to use hierarchical routing, where a graph with multiple levels is constructed during a preprocessing step. In the multi-level graph higher levels are compact graph versions of the lower levels. With such a multi-level graph, the search can be performed more efficiently and therefore faster. Delling et al. [8] is the most recent article that gives an extensive overview of the algorithms corresponding to both ideas.

3.1.1 Goal Directed Search Algorithms

A* (Hart et al. [14]) is the first goal directed search algorithm. The difference with Dijkstra's algorithm is the usage of the cost function. Dijkstra's algorithm only uses a cost function based on the path to a node v. The A* algorithm also uses a cost estimation of the remaining path to the target t and adds this to the cost function. If a good estimation is available, this algorithm prevents searching away from the target and, thereby, forces a goal directed search. As a consequence of the cost estimation, more computer memory is needed compared to Dijkstra's algorithm, because the node positions must also be stored. Besides having to find a good estimator, having to know the spatial layout of the graph is seen as a minor downside of the algorithm.

Goldberg and Harrelson [13] introduce the idea of landmarks. Landmarks are certain nodes on the graph wherefrom, the distance to all other nodes in the graph will be calculated. In the article Goldberg and Harrelson introduce the ALT algorithm which uses a combination of the A^* algorithm, Landmarks and the Triangle inequality. The landmarks together with the triangle inequality are used for the cost estimation to reach the target t, used in the A* algorithm. In the article, it is shown that the ALT algorithm finds good lower-bounds for the cost estimation and leads to a noteworthy reduction in the search space. The inconvenience of the ALT algorithm is that a lot more memory is needed compared to the Dijkstra and A* algorithm, due to the preprocessing calculations.

Lauther [19] suggests the first basic ideas of the usage of partition based Arc-Flags (AF) approach. This algorithm uses a simple rectangular geographic partition that divides the graph into different regions. For each arc it is determined whether it is on the shortest path into another region. This information is saved as a flag, which indicates whether or not it is in such a shortest path. Each arc then contains a flag for every region. It turns out that this results in a very efficient algorithm. In Hilger et al. [15] also AF are investigated but instead of using the simple rectangular geographic partition, they use an arc multi-way separator partitioning. This partitioning improves the algorithm with respect to the one proposed by Lauther [19]. Mohring et al. [20] gives an extended research in several partitioning methods that result in the best algorithm.

3.1.2 Hierarchical Routing Algorithms

Sanders and Schultes [25] introduce the idea of Highway Hierarchies and extend their approach in a second paper (Sanders and Schultes [26]). The idea behind this hierarchical routing is that some arcs are more important than others. An intuitive example is that highway arcs will be much more used in different routes than arcs representing residential streets. The Highway Hierarchies (HH) algorithm is based on that idea. It builds a multi-level graph in the preprocess. In every level shortcuts are used to replace low degree nodes and non-highway arcs from the graph. The algorithm used to solve the shortest path problem in the multi-level graph is based on a bidirectional Dijkstra algorithm. The difference is that certain nodes do not need to be expanded if the search is sufficiently far from the source or target.

Highway-Node Routing (HNR) (Schultes and Sanders [27]) and Contraction Hierarchies (CH) (Geisberger et al. [12]) are based on the ideas of HH but focus merely on the importance of nodes instead of arcs. Here a multi-level graph is build in the preprocess based on which nodes are important. CH is a special case of HNR, where every removed node is a new level. A disadvantage of the hierarchical algorithms is that a lot more shortage space is needed compared to the goal directed algorithms, since new arcs are constructed.

Bast et al. [2] introduce the idea of Transit Node Routing (TNR). The algorithm is based on the concept that only a small set of nodes is needed to cover all sufficiently long shortest paths in any region of a road network. This algorithm turns out to be the fastest algorithm so far for the hierarchical routing algorithms, but needs even more storage space then CH.

3.1.3 Combinations

Holzer et al. [16] is the first to combine several existing speedup techniques for Dijkstra's algorithm. The result of their research is that some combinations work very well and others do not. It turns out that combining goal directed search and hierarchical routing is very promising. Based on these results Bauer et al. [6] continue the research on the combination of goal directed search and hierarchical routing. They formulate three new algorithms combining the ideas, like a core-based ALT and a combination of Hierarchies and Arc-Flags, and give an extensive overview of the results in different situations. The result of this research is that combined algorithms outperform most of the existing algorithms, where the combination of Hierarchies and Arc-Flags is advised for smaller graphs and core-based ALT for more dense graphs.

Abraham et al. [1] also introduce a practical implementation of an algorithm that combines both goal directed search and hierarchical routing. It is a hub-based labeling algorithm (HL) that has promising theoretical bounds according to Abraham et al. [1]. At first HL seemed to be impractical to execute, but it turned out that HL is a very fast algorithm, for road networks, compared to the existing non-combined algorithms.

3.2 Other related work

All previous discussed algorithms assume fixed, known arc costs. Our research extends this by consider practical information such as time dependency. This means that the arc costs are not fixed and may also not be known in advance, which makes the model much more realistic. In the literature time dependency already received some attention.

In Polychronopoulos and Tsitsiklis [24] research is done regarding the uncertainty in the cost of the arcs. This uncertainty can be caused by, for example, traffic congestion, weather conditions and events. Three assumptions regarding the moment when the real cost will be available are analyzed and also a model is proposed.

In the article of Fan et al. [10] a function for the expected travel time is constructed, which is based on the concept of correlated link costs. This means that if one link is congested it is likely a neighbouring one will also be congested. This, however, is an 'online' approach and cannot be used to determine a route, and its corresponding travel time, in advance.

Bauer and Delling [5] argue that the, at that time, fastest speedup techniques are bidirectional and are therefore of no use when the graph is time dependent. They introduce SHARC, a combination of *SH*ortcuts from HH and *ARC*-flags. This is a fast, unidirectional search algorithm that also could be used bidirectionally. The algorithm is faster then most existing algorithms and needs less storage space.

Nannicini et al. [22] are the first to introduce an idea for combining bidirectional search with time dependency. They propose a method for the bidirectional ALT algorithm. To overcome the problem caused by the backward search, the backward part of the route is evaluated twice. First the forward search is performed with the actual cost and the backward search is performed using a lower bound for the arc costs. If the search meet, candidates can be determined. When there are several candidates and there is no possible shorter path for other candidates, the backward part is repeated by a forward search based on the actual cost. In this phase only the nodes that have been found before, are searched again.

In Batz et al. [3] the concept of SHARC and of Nannicini et al. [22] is used to develop a CH algorithm that is able to handle time dependent arc costs if the departure time at the start of the route is known. Due to the shortcuts in CH time dependency causes problems. They propose a profile such, that take a travel time function into account, to determine the travel time of the shortcuts. In Batz et al. [4] their approach is extended with the usage of cost approximations for the shortcuts, together with a better unpack method. As a result, this saves computation time and storage space. They also give ideas for travel time profiles over an interval of potential departure times.

The problems arising when considering road network properties have also been subject to earlier research. Kirby and Potts [18] discuss the problems that turn restrictions can give and propose a mathematical formulation to solve them.

Geisberger and Vetter [11] investigate what the best approach is for incorporating turn costs in routing algorithms. By turn costs is meant travel time loss by having to slow down to make a turn. They compare two different types of graph. A node-based graph, wherein arcs represent roads and nodes represent junctions, and an edge-based graph, where nodes represent roads and arcs the junctions. It appears that using a node-based graph is the most convenient way to take turn cost into account. It performs better in terms of computation time and storage space.

4. ALGORITHMS

In this research we compare five different route planning algorithms, regarding the performance in the case that practical information is considered. We analyze Dijkstra's algorithm, A*, ALT, Contraction Hierarchies (CH) and a combination of ALT and CH: CHALT.

First of all, we use Dijkstra's algorithm because it is the best-known algorithm and always works. It can be seen as a benchmark in the comparison with the other algorithms, because the other algorithms are supposed to improve Dijkstra's algorithm. A* is analyzed because, along with Dijkstra's algorithm, it is another commonly used algorithm. At the same time A* is related with ALT, which makes it even more relevant to add A* to the analysis. The literature review (Chapter 3) mentions that, in the current research on shortest path algorithms two main concepts arise, namely goal directed search (GDS) and hierarchical routing (HR). From both concepts we analyze the algorithm that outperforms other algorithms of the corresponding concept and we also combine these two algorithms. ALT is a good performing GDS algorithm which we will use (see Goldberg and Harrelson [13]) and CH is among the strongest performing hierarchical routing algorithm used for this research (see Geisberger et al. [12]). Since Holzer et al. [16] and Bauer et al. [6] state that combining both concepts can be beneficial, we also apply this in our research by combining CH and ALT, resulting in CHALT.

In this chapter we will explain each algorithm in detail and illustrate them by means of an example. The graph we will use as the example, is given in Figure 4.1. In this graph an arc can be used in both directions with the same cost. For all algorithms, a route from node d to node g will be planned. Besides the explanation, we will also discuss the advantages and disadvantages of the different algorithms.

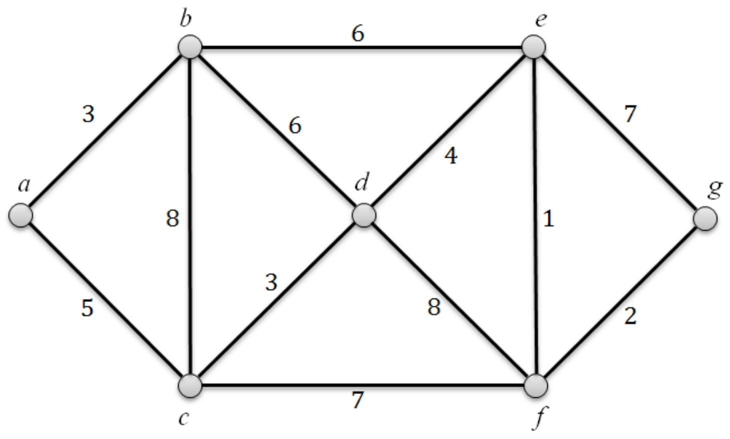


Fig. 4.1: Example Graph

4.1 Dijkstra's Algorithm

Dijkstra's algorithm (or Dijkstra search) is the most reliable algorithm to find the shortest path within a graph. The algorithm is exact and requires only a small amount of storage space. On the other hand, due to the precision of the algorithm, it is slow compared to other algorithms. Dijkstra search starts searching for the shortest path from the source node s of the problem. From there, the algorithm continuously carries on searching for the node v, that has the lowest cost from s to v ($< s, \ldots, v >$), until the destination node t is found. These costs are based on the arc costs of the path from s to v. So the cost function $f_s(v)$ for a path from source node s to a node v, is given by:

$$f_s(v) = g(s, v), \tag{4.1}$$

where $g(s, v)$ is the cost of the path from s to v. To search for the node v with the lowest cost, the cost of the path from s to v needs to be known. This information will therefore be saved with the node. When the algorithm is in a node v, that has the lowest cost, all nodes u reachable from v get assigned the cost of path $< s, \ldots, v, u >$. However, there is a large probability, that within a graph, there is more than one path from s to such a node u. So it needs to be checked which path has the lowest cost. To do this, we can use the following equation:

$$f_s(v) + c_{vu} < f_s(u) \tag{4.2}$$

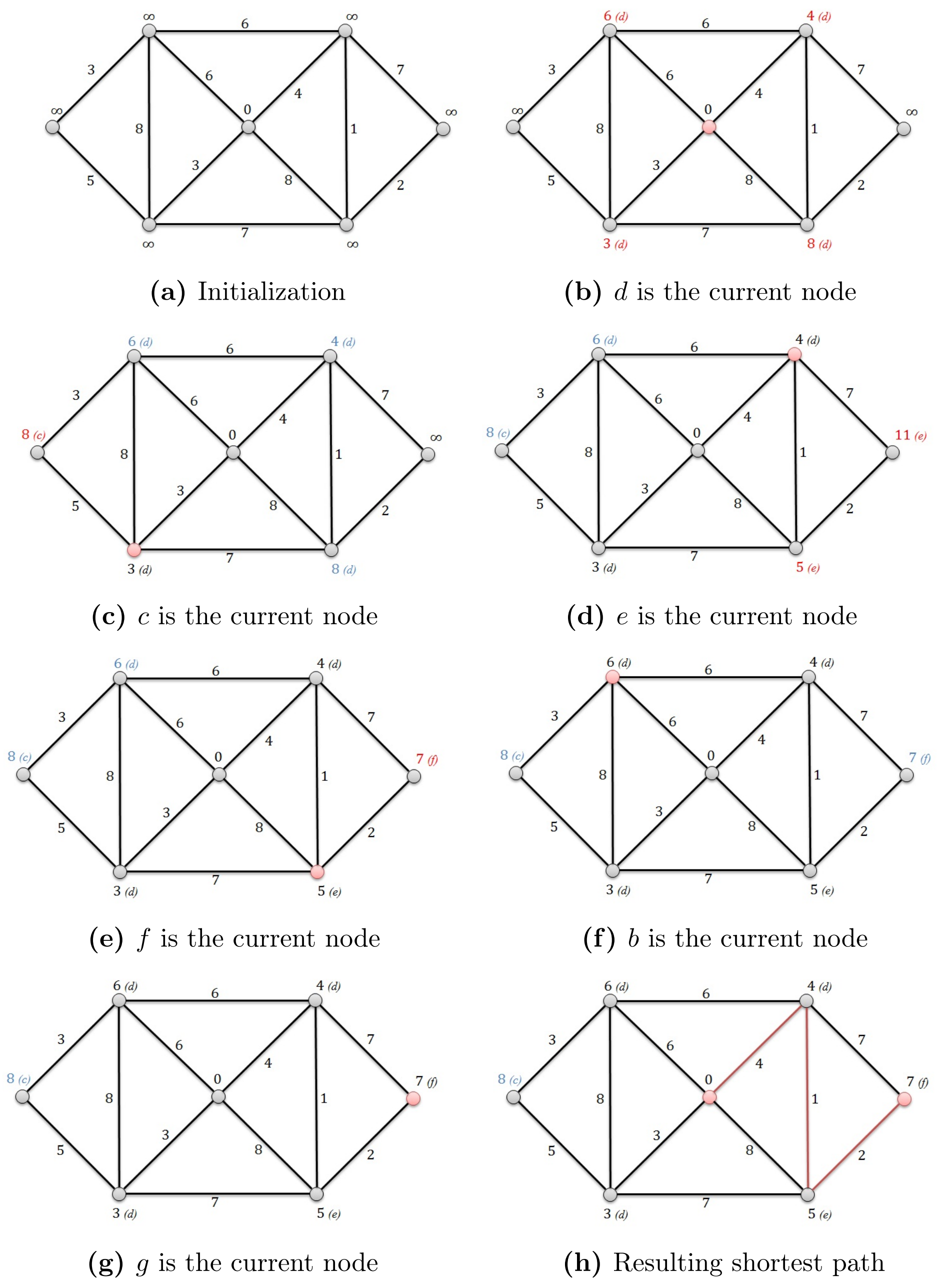
In equation (4.2) the costs of two paths are compared. $f_s(v) + c_{vu}$ is the cost of the newly found path ($< s, \ldots, v, u >$), where $f_s(v)$ is the cost of the path from s to v and c_{vu} the cost of arc (v, u). $f_s(u)$ is the cost of an earlier found path to node u. If equation (4.2) holds, it means that the path to node u via node v is a shorter path. Therefore the cost of this path should be saved with node u. This procedure (together with the assumption that arc costs can not be negative) ensures that at the moment that the algorithm arrives in node u, it is not possible to reach u from s via any other node, leading to a shorter path.

In order to retrieve the shortest path at the end of the algorithm, each node also gets assigned a preceding node, corresponding to the path found by the algorithm. So if equation (4.2) holds, node u gets node v assigned as predecessor.

The algorithm stops at the occurrence of two different situations: in the case that target node t is found, or when all reachable nodes are analyzed. Since each node is saved with the costs and its predecessor, the shortest path from s to t, with the corresponding costs, can be retrieved after finding node t.

To illustrate Dijkstra's algorithm by our example, consider the graph as in Figure 4.1 and note that node d will be the source node and node g the target node. Next, we introduce a mathematical initialization trick, to make each iteration in the algorithm as much the same as possible. We initialize the cost functions of all nodes in the graph. $f_d(d)$ is set to zero (note this actually is true), and all other cost functions are set to infinity. This is done so that in every iteration the check in equation (4.2) can be applied and makes sure that when the algorithm for the first time checks node v, the path will always be saved (since the costs of that path are less than infinity). This is illustrated in Figure 4.2(a), where the costs $f_d(\cdot)$ are labeled to the nodes.

In the first iteration, the node with the lowest cost is node d and so the nodes reachable from d need to be checked. For ease of explanation, we will refer to node d as the current node. In each iteration the current node will be the node that had the lowest cost. Since before the first iteration no paths are found, all nodes reachable from d need to be updated. In Figure 4.2(b) this is illustrated. The red node represents the current node and the red labels are the updated costs as a result of the check of Dijkstra's algorithm. Also note that the preceding node is labeled to the node.


(a) Initialization **(b)** d is the current node

(c) c is the current node **(d)** e is the current node

(e) f is the current node **(f)** b is the current node

(g) g is the current node **(h)** Resulting shortest path

Fig. 4.2: Example Dijkstra's Algorithm

The algorithm continues with the next iteration, where the node with the lowest cost becomes the current node and all reachable nodes are checked and updated if needed. It follows from Figure 4.2(b) that node c becomes the next current node. In Figure 4.2(c) the result of checking and updating, all from c reachable nodes, is displayed. The labels are blue when the costs are not updated, either because the node is not reachable from the current node or if the check finds that no shorter path is found, so equation (4.2) does not hold. The algorithm continues iterating over the nodes, as can be seen in Figures 4.2(d) to 4.2(g). In Figure 4.2(g) node g has become the current node, so the shortest path from d to g is found. The resulting path is given in Figure 4.2(h).

4.2 A*

The A* algorithm is the next algorithm considered and is an extension of Dijkstra's algorithm. The idea of A* is that it should be possible to prevent the algorithm from searching away from the target node t, and thereby saving computation time. To achieve this, A* uses the cost function of Dijkstra search and extends it with an estimator to predict the remaining part of the path ($< v, \ldots, t >$). Taking this estimation into account, ensures that if the algorithm finds a node further away from the target, the cost will be higher. The only requirement is a good estimator. A good estimation never overestimates the actual cost, otherwise is no longer legitimate to stop the search if the target node is found for the first time. The reason is that if the costs are overestimated, it could be that another, not yet found path, turns out to have lower costs. In the case the costs are underestimated, the procedure remains reliable. This estimation should, however, preferable be as high as possible. If a good estimator is used, the algorithm is faster then Dijkstra search. However, due to the estimator, it needs more storage space.

The cost function for a path as used within A*, corresponding to a v, is defined as:

$$f_s(v) = g(s, v) + h(v, t) \tag{4.3}$$

Where $h(v, t)$ is the cost estimation of path $< v, \ldots, t >$. Also the check whether a shorter path is found during the algorithm changes:

$$g(s, v) + c_{vu} + h(u, t) < f_s(u) \tag{4.4}$$

Here the actual cost till node u is used, $g(s, v)$ the costs from s to v and c_{vu} the arc costs of arc (u, v), and the estimation $h(u, t)$ for the remaining path from node u to the target. The rest of Dijkstra's algorithm remains the same within A*.

Many different kind of estimations can be used for this $h(v, t)$. The most simple and common used estimation is given by the Euclidean distances. These Euclidean distances use the coordinates of the nodes to estimate the path from v to t. As a result extra storage space is required to save these coordinates. Assuming each node has an x and y coordinate, the Euclidean distance between two nodes a and b is given by:

$$\text{Eudist} = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}, \tag{4.5}$$

where Eudist stands for the Euclidean distances.

The Euclidean distances clearly underestimates the cost of the actual path, because it uses straight lines. However, the Euclidean distances could give poor estimations in the case of, for instance, in a road network with a lot of one-way streets or when rivers appear in an area with no bridges nearby.

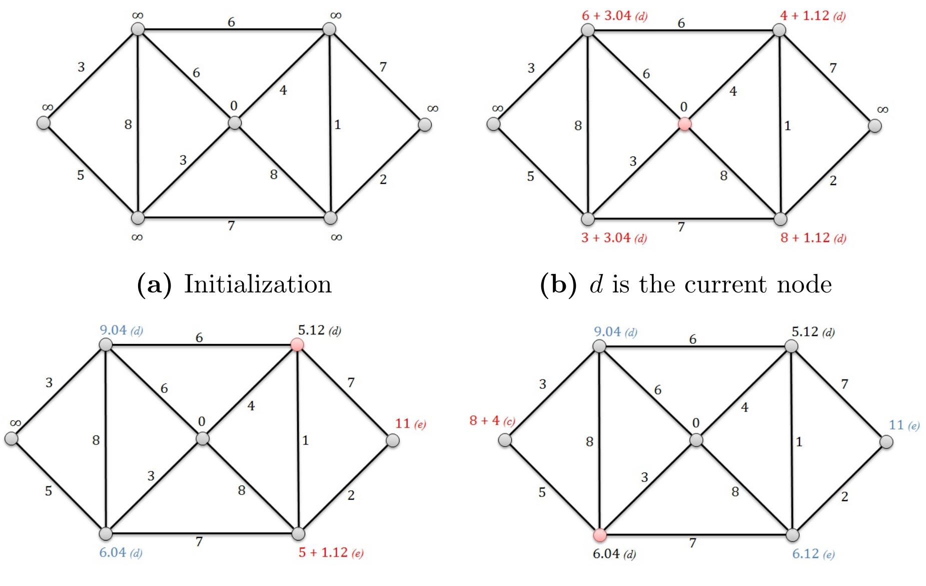
In order to apply A* to our example (Figure 4.1), coordinates need to be defined for each node in the graph. The Euclidean distances also needs to be calculated from every node to the target node (g). Table 4.1 gives the used coordinates for the example, together with the Euclidean distances to node g.

The A* algorithm uses the same initialization step as Dijkstra's algorithm, Figure 4.3(a) illustrates this step once again. The source node d becomes the first current node and in Figure 4.3(b) the cost functions of the nodes reachable from d are updated. Notice

Node	**Coordinates**	**Euclidean distance to g**
a	(0, 0.5)	4
b	(1, 1)	3.04
c	(1, 0)	3.04
d	(2, 0.5)	2
e	(3, 1)	1.12
f	(3, 0)	1.12
g	(4, 0.5)	0

Tab. 4.1: Example: Used coordinates and resulting Euclidean distances

that these costs are the costs to the node added with the estimation: the Euclidean distance for the node to target node g. In Figure 4.3(c) to 4.3(f) the algorithm continues and eventually node g becomes the current node. Figure 4.3(g) gives the shortest path as found by A*. Notice that both Dijkstra search and A* find the same route as shortest path. This is because both algorithms are optimal and therefore always will find the route with the lowest cost. In the case the problem has no unique solution they may find other routes, but with the same lowest cost. Also note that A* needs one iteration less, in this example, to find the route. For larger graphs this difference will be much larger. So, there is a tradeoff between preprocessing and query speed.



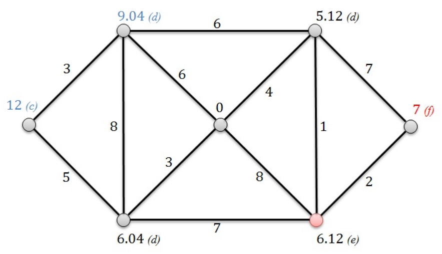
(a) Initialization

(b) d is the current node

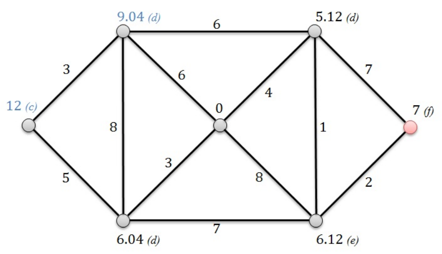
(c) e is the current node

(d) c is the current node

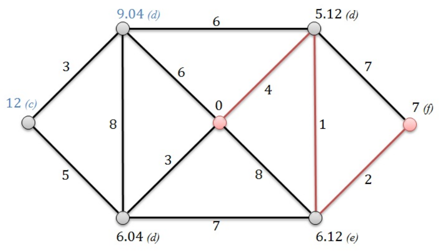
Fig. 4.3: Example A* Algorithm



(e) f is the current node



(f) g is the current node



(g) Resulting shortest path

Fig. 4.3: Example A* Algorithm cont.

4.3 ALT

The ALT algorithm is developed with the purpose to give an improved lower bound estimator for the A* estimator. ALT is a combination of A*, Landmarks, and the triangle inequality. Landmarks are nodes selected from the graph, to enable the usage of the triangle inequality. The triangle inequality then again is used to calculate an estimator for $< v, \dots, t >$, that can be used by A*. During preprocessing of ALT, the shortest paths are calculated from and to each landmark, for all nodes in the graph. These shortest paths are found using Dijkstra search. ALT has as additional advantage that it is also applicable to arbitrary graphs because no geographical information is needed. This is in contrast with other existing estimators for A*, as, for example, the Euclidean distance. On the other hand, the algorithm needs a preprocessing step, causing more needed storage space and a decrease in computation time of the query (the actual search for the shortest path).

When the preprocessing is done, the query can start by using A*. The estimator $h(v,t)$, used within A*, is determined based on the preprocessing computation. By the triangle inequality is known that:

$$g(v,L) - g(t,L) \leq g(v,t) \tag{4.6}$$

and

$$g(L,t) - g(L,v) \leq g(v,t) \tag{4.7}$$

where L represents a landmark node.

Since we need a lower bound as estimator, we can use the cost differences as given in equation (4.6) and (4.7). Furthermore, we prefer the lower bound to be as large as possible, to come closer to reality. To accomplish this, the estimator $h(v,t)$ is set to the maximum of all the differences belonging to node v. In this way the A* algorithm can be applied.

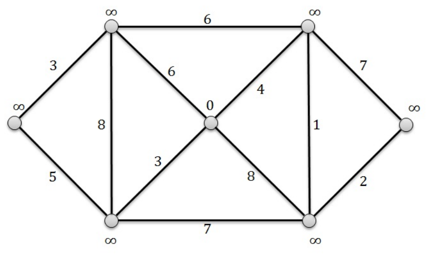
The landmark selection is also a crucial part of this algorithm. In Goldberg and Harrelson [13] several methods are tested for this purpose. It turns out that randomly selecting the landmarks is obviously not the best method but still performs quite well and is the fastest method to use. For this reason we will randomly select the landmarks in this research. Goldberg and Harrelson [13] also conclude that using sixteen different landmarks is a sufficient number to use for all kinds of different graph sizes. So we will also use sixteen landmarks for our research.

The disadvantage of this algorithm is that the preprocessing is a time consuming procedure, in the case of sixteen landmarks, this means thirty-six full graph, Dijkstra searches are needed. Furthermore ALT does need a lot more storage space, compared to Dijkstra search and A*. For each node in the graph, two shortest paths costs for all landmarks need to be saved (from and to each landmark). The benefits are that the computation time of the query is reduced. This is due to the fact that ALT gives a better estimator for A*, which results in a large reduction in search space.

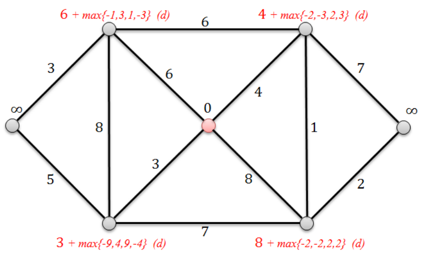
Node	**Landmark** c	**Landmark** e
a	5	9
b	8	6
c	0	7
d	3	4
e	7	0
f	7	1
g	9	3

Tab. 4.2: Shortest path distances from and to landmarks c and e

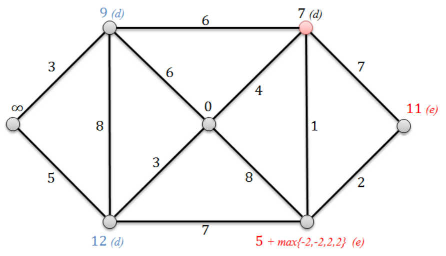
For our example we select two landmarks, namely node c and e. Table 4.2 gives all shortest paths from and to node c and e. Given the preprocessing computations, the query can start. Again the initialization step is the same, see Figure 4.4(a). The estimator is given by calculating the differences as given in equations (4.6) and (4.7) for both landmarks and taking the maximum of the four outcomes, see Figures 4.4(b) and 4.4(c). Note that the cost function of node e and f already represent the cost of the actual shortest path. This is due to the fact that node e is a landmark and so therefore the shortest path costs are known from the preprocessing and node f is within that path. Figures 4.4(b) to 4.4(e) illustrate the process of ALT and in Figure 4.4(f) the shortest path is presented, again the same route as before. Finally notice that this algorithm needs again less iterations to find the shortest path, so again there is the trade-off between preprocessing and query speed.



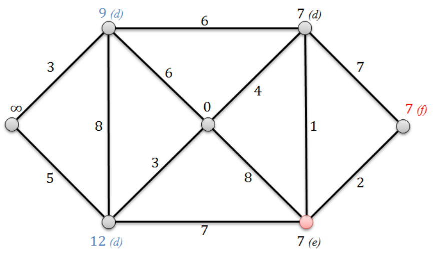
(a) Initialization



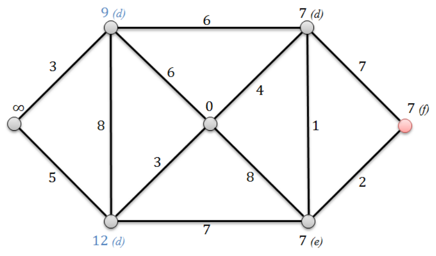
(b) *d* is the current node



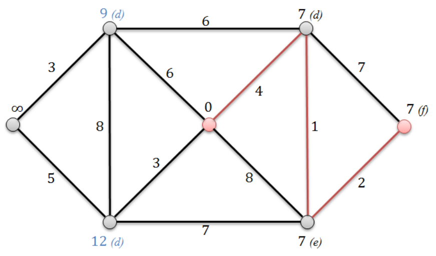
(c) *e* is the current node



(d) *c* is the current node



(e) *f* is the current node



(f) *g* is the current node

Fig. 4.4: Example ALT Algorithm

4.4 Contraction Hierarchies

Contraction Hierarchies (CH) is a relatively new routing algorithm to find the shortest path within a graph Geisberger et al. [12]. The purpose of CH is also to decrease the computation time needed for the routing process. This algorithm, however, uses another approach than changing the cost function, like A* and ALT. CH tries to reconstruct the graph in such a way, that a route can be found using less iterations. The idea is to use shortcuts, that will replace several arcs, to achieve this. The reason to consider CH in this research, is because it shows promising results in literature and we like to investigate whether this also holds in the case of adding the practical information.

The CH algorithm can be split into two separate parts: the preprocessing, i.e. building the multi-level graph, and the query, i.e. finding the best route.

4.4.1 Preprocessing

During the preprocessing a multi-level graph is constructed. A multi-level graph implies a graph, consisting of multiple levels, where a higher level is a compacter version of a lower level graph. The original graph is the lowest level and by contracting nodes, new levels arise. A node contraction is the removal of a node from the graph. As a consequence of such a contraction, the arcs connected to the contracted node need to be replaced by shortcuts in order to retain the paths between the remaining nodes. In each level one node is contracted to construct a new level. Nevertheless, these levels need to be constructed in an efficient way, so the resulting multi-level graph can be used to find routes rapidly. To accomplish this, a node ordering, based on node importance, is used to provide the sequence in which the nodes should be contracted. So, first a node ordering based on importance should be defined. Once the node ordering is known, the preprocessing can continue with contracting the nodes.

Henceforth we will refer to the multi-level graph as $\mathcal{G}$ and a level is referred to as G_v, that is the level from which the v^{th} node will be contracted. As a result $\mathcal{G} = (G_1, G_2, \dots, G_n)$ with G_1 being the original graph.

Node Ordering

A good node ordering is important, because during the query it is preferred to skip parts of the graph that are often, or always used in the same way. The determination of the node ordering is made by a linear combination of several priority criteria. Many different criteria are possible to use in this context. Geisberger et al. [12] investigated the impact of several different criteria and combinations of them. It turned out that a combination of edge difference, deleted neighbors, and search space size preforms best in terms of the total preprocessing time. Therefore we will use this combination throughout this research.

Edge Difference (*ED*) is the most important ordering criteria. The edge difference is defined as the difference between the number of shortcuts needed when v is contracted, and the number of edges incident to v before contracting it (see Figure 4.5). The goal

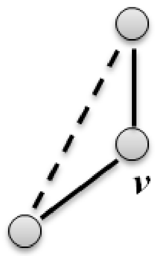


Fig. 4.5: The Edge Difference for v is $1 - 2 = -1$ (Geisberger et al. [12]).

of using ED as an ordering criteria, is to prevent that, while contracting nodes, the graph converges to the complete graph (each node connected to all other nodes in the resulting graph). The ED ensures that nodes for which, when contracted, less shortcuts are constructed than there are disappearing arcs, are assigned a higher priority to be contracted.

Deleted Neighbors (*DN*) is used to ensure more uniformly distributed contractions within the graph. This prevents the ordering from focusing the contractions on the same area, which can give problems in terms of dead-end valleys. Therefore we use a criterion which counts the number of already contracted neighbors.

Search Space Size (*SSS*) is a criterion that tries to limit the size of the query search space. To do this, a simple estimator is used that can be shown to be an upper bound (Geisberger et al. [12]) on the number of arcs in a path from s to v. Initially $SSS(v) = 0$, and when v is contracted each neighbor u of node v is receives a SSS given by:

$$SSS(u) = max\{SSS(u), SSS(v) + 1\} \tag{4.8}$$

In Geisberger et al. [12] extensive sensitivity analysis is used to determine the coefficients for the linear combination:

$$PV = \alpha ED + \beta DN + \gamma SSS \tag{4.9}$$

They concluded that the best values for the coefficients are $\alpha = 190$, $\beta = 120$, and $\gamma = 1$, which we will therefore use throughout this research. PV stands for priority value, and the lower the PV the earlier a node will be contracted.

Considering the ordering criteria, the first issue we notice, is that all these criteria are influenced by the contraction of a node. For this reason, the ordering needs to be updated after every contraction. However, recalculating the criteria for all nodes after every contraction is time consuming. Therefore, we use several approaches that update the criteria in a clever way. For this research we use neighbor update, and lazy update (Geisberger et al. [12]) because together they cover almost all the changes.

Neighbor Update is straightforward, namely updating the ordering criteria of the neighbors of the last contracted node. This is useful because a contraction mainly causes changes in the criteria of its neighbors, in fact DN and SSS affect only the neighbors .

Lazy Update is used, because it is possible that not only the neighbors are affected by ED. To make sure that at least the node with, at that moment, the lowest PV still has the correct PV, the criteria of this node is also updated. This is repeated until the node with the lowest PV remains the node with the lowest PV after recalculation.

Node Contraction

If it is known which node v is the next to be contracted, the contraction can be performed. Shortcuts need to be constructed in order to retain the connections between other nodes, that are connected indirectly via a node v. However, keeping in mind the fact that we want to find the shortest path later on, we only need to add such a shortcut when no other, shorter path exists within the graph. Assume node v is going to be contracted next, and this node is directly connected to nodes u and w, so that path $< u, v, w >$ exists. If v is contracted, a shortcut arc (u, w) should be constructed. However, when by a Dijkstra search it turns out that there exists another path $< u, \ldots, w >$, not containing v, with lower total costs, the shortcut is not needed. So, during the contraction part of the CH algorithm it needs to be checked checked whether, for each direct to v connected neighbor combination, there exists a shorter path within the graph G_v:

$$g(u, w) \leq c_{uv} - c_{vw} \tag{4.10}$$

where $g(u, w)$ is the cost of a path $< u, \ldots, w >$, not containing v. If equation (4.10) does not hold, the shortcut needs to be added.

Note that this search for shorter paths is time consuming, because it contains multiple Dijkstra searches. Therefore, Geisberger et al. [12] introduce the idea of using a fixed number of ‘hops’, where hops are the arcs in the alternative path. So for this maximum number of hops is searched for a shorter path. In the case that within this number of hops the costs of the path become so large that equation (4.10) is violated, it is certain that the shortcut is needed. If within the maximum number of hops a connection can be made, the procedure is straightforward: apply equation (4.10) to check again whether the shortcut is needed. It is also possible that no connection exists. In this case the shortcut is always needed, in order to ensure correctness of the graph. In this research we will use a maximum of 5 hops. The reason is that, according to sensitivity analysis of Geisberger et al. [12], this prevents the number of arcs from exploding.

Summarizing the procedure of preprocessing, first of all, the contraction of every node from G_1 is simulated in order to calculate the initial ordering criterium. Based on this criterium the first node contraction can be performed, resulting in G_2. After this contraction, the ordering criteria of the neighbor nodes are updated, applying Neighbor Update and Lazy Update, by again simulating the contraction of these nodes in the current graph level. Then, the next node can be contracted. This procedure continues until all nodes have been contracted. In each contraction needs to be considered that the graph is directed and therefore no infeasible paths need to be checked. Also, every step does only concern the graph of the corresponding level, so when node of level v is going to be contracted, G_v should be the graph in which searches are performed.

Before continuing with the explanation of the second part of CH, the query, we will first illustrate the preprocessing with our example. As mentioned before, we simulate the contractions of all nodes from the original graph to calculate the initial ordering criteria. Table 4.3(a) gives the results of these calculations. Notice that the DN and SSS in the first iteration are zero and so the initial ordering is only based on the ED. If we look at the ED of, for example, node a, it can be seen that it is -2 because node a has two incident arcs and no shortcut will be made, since $g(c, b) = 8$ is equal to $c_{ca} + c_{ab}$. This means that equation (4.10) holds, see Figure 4.6(a). Actually there are four incident arcs, since each line in the graph represents both directions. However, we will ignore this fact throughout this example, because this will only lead to an factor 2 increase of all values in these calculations. Looking at node b we can seen that one shortcut, (a, e), needs to be constructed. All existing path in the graph not containing node b, are larger than $< a, b, e >$. This evaluation is done for all nodes in the graph.

Node	ED	DN	SS	PV	Level
a	$0-2=-2$	0	0	-380	
b	$1-4=-3$	0	0	-570	
c	$1-4=-3$	0	0	-570	
d	$1-4=-3$	0	0	-570	
e	$3-4=-1$	0	0	-190	
f	$3-4=-1$	0	0	-190	
g	$0-2=-2$	0	0	-380	

(a) First level

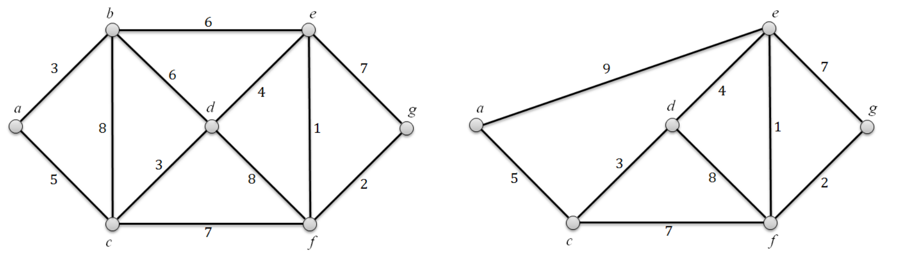
Node	ED	DN	SS	PV	Level
a	$0-2=-2$	1	1	-259	
b					1
c	$1-3=-2$	1	1	-259	
d	$1-3=-2$	1	1	-259	
e	$2-4=-2$	1	1	-259	
f	$3-4=-1$	0	0	-190	
g	$0-2=-2$	0	0	-380	

(b) Second level

Tab. 4.3: Contraction: Determination levels 1 and 2

As a result can be seen that nodes b, c, and d all have the same, and lowest priority value. In that case the algorithm contracts the node that was examined first. In this case that is node b. So node b is contracted and the shortcut (a, e) is added to the graph. It also means that node b is of level 1. The resulting graph is given in Figure 4.6(b). For this graph the neighbors are updated and lazy update is applied. Table 4.3(b) gives the results of these updates. Note that, the priority value of nodes f and g did not change, because f was checked by lazy update and did not change, and e therefore was not checked (notice however that it would not change anyway).

Looking at Table 4.3(b) it follows that node g is to be contracted next and becomes level 2. No shortcut will be constructed, so the graph will be as in Figure 4.7(a) and the



(a) Original Graph

(b) Node b is contracted and shortcut (a, e) is constructed

Fig. 4.6: Contraction: Graph levels 1 and 2

resulting criteria updates are given in Table 4.4(a). Node f is to be contracted next and will be level 3. Again no shortcuts are constructed and the results are given in Figure 4.7(b) and Table 4.4(b). Notice in Table 4.4(b) that the criteria of nodes c and d will not be updated but the priority actually did change to 52.

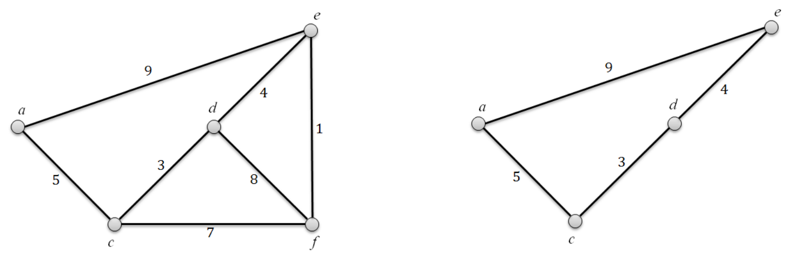
Node	**ED**	**DN**	**SS**	**PV**	**Level**
a	$0-2=-2$	1	1	-259	
b					1
c	$1-3=-2$	1	1	-259	
d	$1-3=-2$	1	1	-259	
e	$2-2=-1$	2	1	51	
f	$0-3=-3$	1	1	-449	
g					2

(a) Third level

Node	**ED**	**DN**	**SS**	**PV**	**Level**
a	$0-2=-2$	1	1	-259	
b					1
c	$1-2=-1$	2	2	-259	
d	$1-2=-1$	2	2	-259	
e	$0-2=-2$	3	2	-18	
f					3
g					2

(b) Fourth level

Tab. 4.4: Contraction: Determination levels 3 and 4



(a) Node g is contracted

(b) Node f is contracted

Fig. 4.7: Contraction: Graph levels 3 and 4

The preprocessing continues with contracting node a and again updating the criteria. Next, node d is contracted and shortcut (c, e) is constructed and then only node c and e remain in the graph. From Table 4.5(b) can be seen that node c will be contracted first, and thereby has level 6 and finally node e is contracted and becomes level 7.

The resulting new graph after the preprocessing is given in Figure 4.9. The level of each node is presented as label to the node, between brackets.

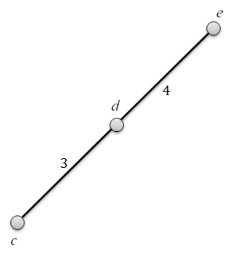
Node	ED	DN	SS	PV	Level
a					4
b					1
c	$0-1=-1$	3	2	172	
d	$1-2=-1$	2	2	52	
e	$0-1=-1$	4	2	292	
f					3
g					2

(a) Fifth level

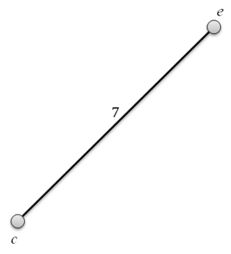
Node	ED	DN	SS	PV	Level
a					4
b					1
c	$0-1=-1$	4	3	293	
d					5
e	$0-1=-1$	5	3	413	
f					3
g					2

(b) Sixth level

Tab. 4.5: Contraction: Determination levels 5 and 6



(a) Node a is contracted



(b) Node d is contracted and shortcut (c, e) is constructed

Fig. 4.8: Contraction: Graph levels 5 and 6

4.4.2 Query

When the preprocessing is completed, the actual query can be performed. The shortest path in the multi-level graph, is found using a bidirectional Dijkstra search. The forward search is executed only upwards and the backward search only downwards. This means that in the forward search an arc (possibly a shortcut) can only be in the shortest path if it connects a node of level v to another node of a higher level. For the backward search this is the opposite.

The stop criteria of this bidirectional search has not been accomplished yet when the same node is expanded in both searches. The reason is that it is possible that a shorter path is in a higher level, including some shortcuts. Therefore the stop criterion prescribes to finish the query when the same node v is expanded in both searches and for both searches the costs of the paths to all not visited nodes is higher than the combined shortest path of both searches, meeting in node v.

Having found the shortest path, the only issue that remains is to retrieve the path in terms of the original graph. So each shortcut within the shortest path needs to be unpacked. For every constructed shortcut is saved which two arcs it replaces at the moment of construction. Therefore every shortcut can be unpacked by replacing it by these two arcs. Note that these arcs can also be a shortcut. If one (or both) of those two arcs is a shortcut, it is unpacked in the same way. This procedure is repeated until only arcs of the original graph are in the shortest path.

In our example, the forward search starts in node d and, due to requirement of upward searching, only nodes c and e are reachable and thus updated, see Figure 4.10(a) (the initialization step is again the same as within Dijsktra's algorithm). The backward search

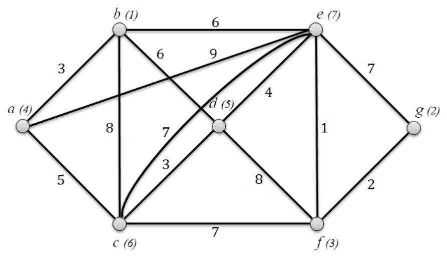


Fig. 4.9: Resulting graph for CH after preprocessing

starts in node g. Both node e and f are reachable from g, because the level of node g is lower than that of node e and f (so the arc from e or f to g goes downward). Thus both nodes get updated, see Figure 4.10(b).

The query continues until in both searches the same node has become the current node. In Figures 4.10(e) and 4.10(f) the current node of both searches is node e. The shortest path cost of both searches together is 7. In none of the searches there is a node that has less than 7 as costs, so the shortest path is found and the forward and backward searches are connected in node e (see Figures 4.10(g), 4.10(h), and 4.10(i)).

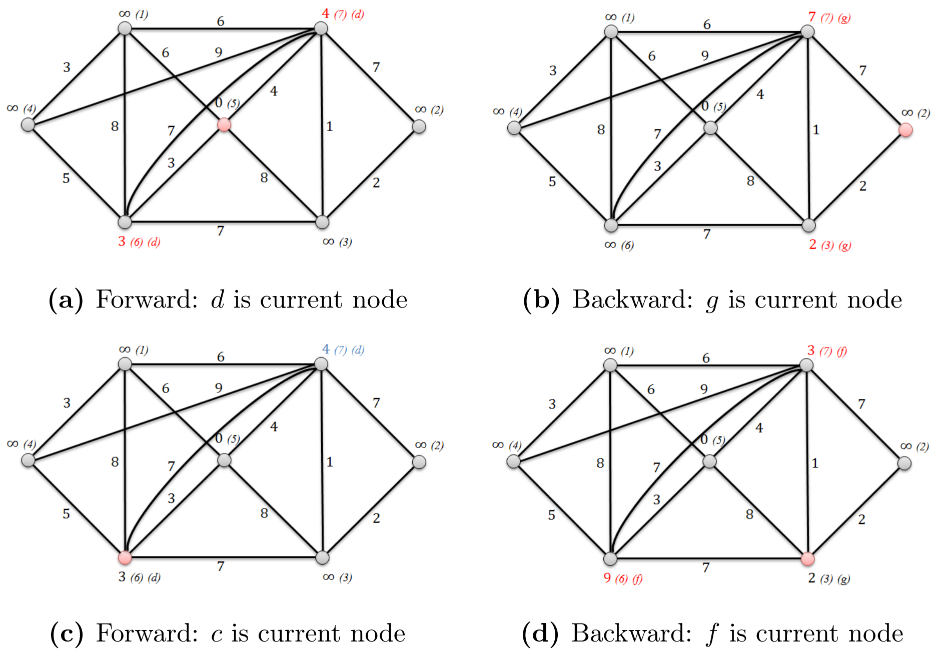
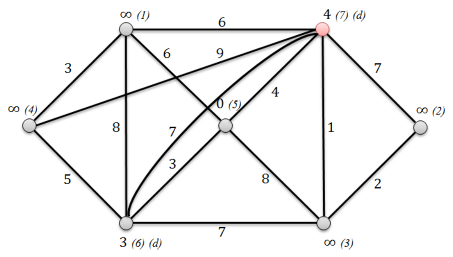
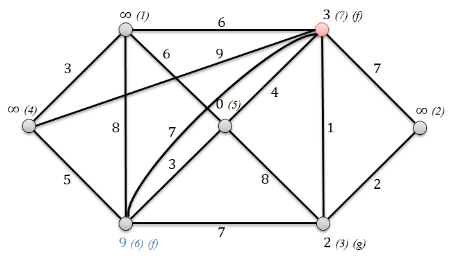


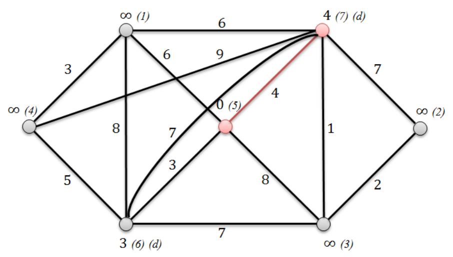
Fig. 4.10: Example CH Algorithm



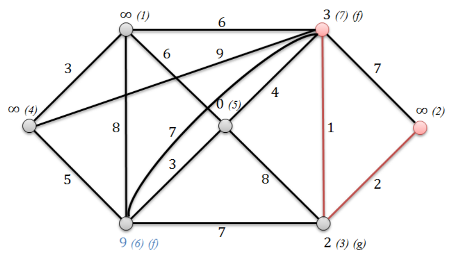
(e) Forward: e is current node



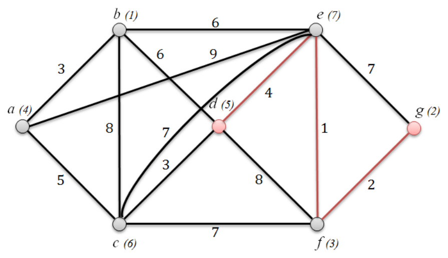
(f) Backward: e is current node



(g) Forward: resulting path



(h) Backward: resulting path



(i) Combined searches shortest path

Fig. 4.10: Example CH Algorithm (cont.)

In this example no shortcuts are used. If we would plan a route from node a to g, the shortcut (a, e) would be used (see Appendix A for the detailed illustration). In that case we know that the shortcut replaced arcs (a, b) and (b, e) and therefore unpacking shortcut (a, e) leads to a shortest path with arcs (a, b) and (b, e).

4.5 *CHALT*

Holzer et al. [16] and Bauer et al. [6] both conclude that combining goal directed search and hierarchical routing can lead to beneficial results in terms of algorithm performance. For this reason, we will also combine the algorithms we used from both concepts for our research. The result is called CHALT, the combination of Contraction Hierarchies (CH) and ALT. The algorithm works as the CH algorithm, but CHALT uses ALT, instead of Dijkstra search, to search for the shortest paths. Dijkstra's algorithm is used at two moments in CH. First during the preprocessing, at a node contraction (say node v), is searched for a shorter path $< u, \ldots, w >$, not containing contracted node v. Furthermore, Dijkstra search is also used during the query of CH. For both processes for CHALT we will use ALT rather than Dijkstra search.

For CHALT we use the same properties as in the individual CH and ALT algorithms. For the ALT-part this means we use 16, randomly selected, landmarks for the preprocess-

Node	**Landmark** c	**Landmark** e
a	5	9
b	8	6
c	0	7
d	3	4
e	7	0
f	7	1
g	9	3

Tab. 4.6: Shortest path distances from and to landmarks c and e

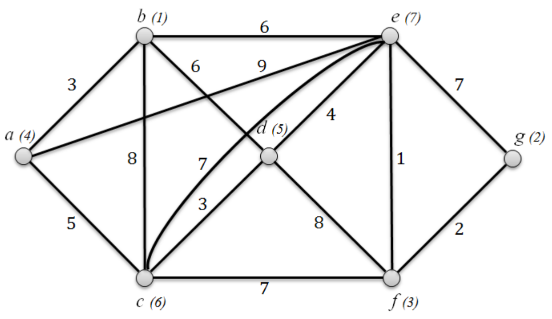


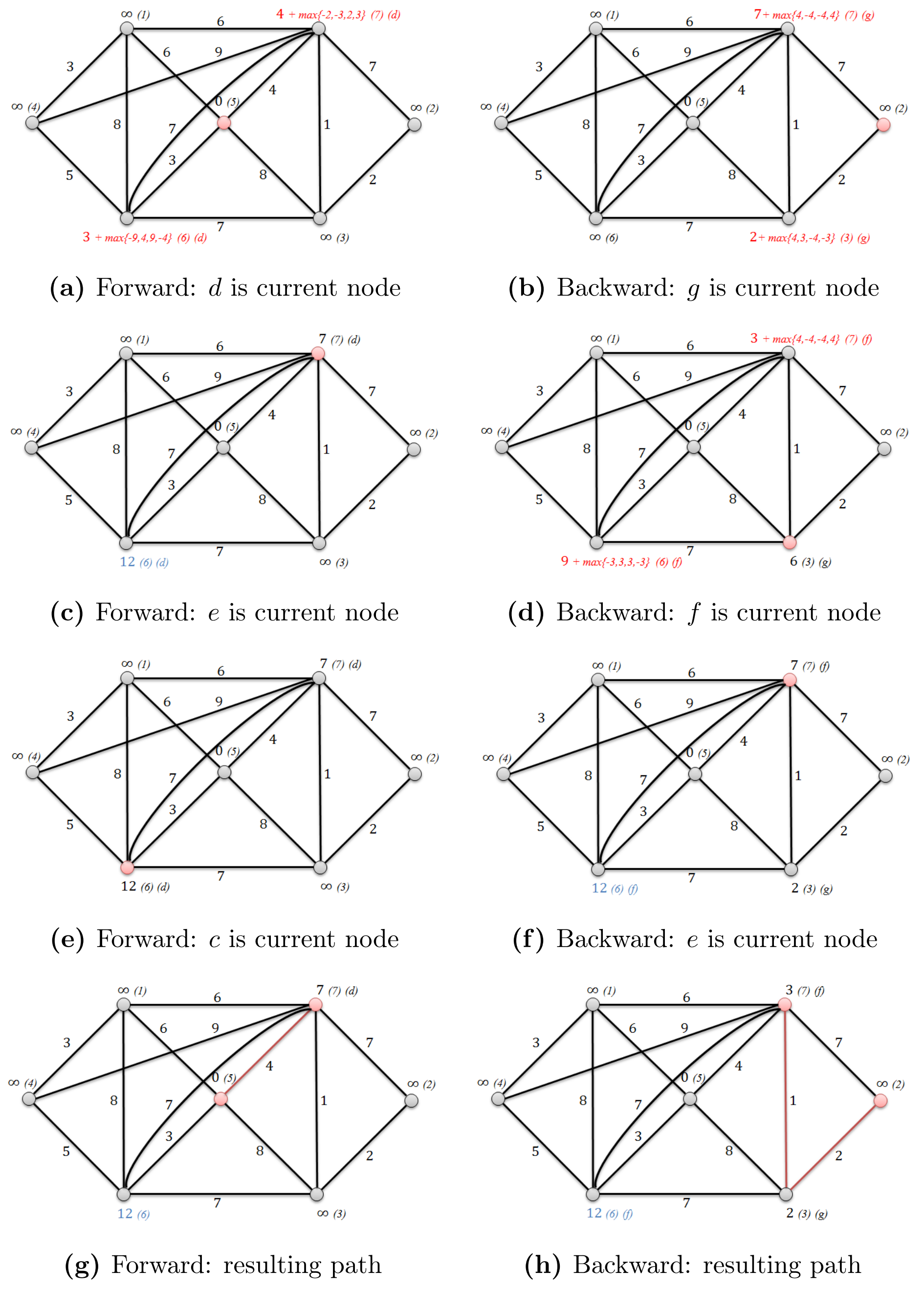
Fig. 4.11: Resulting graph for CHALT after preprocessing

ing. In the CH-part we use again equation (4.9) to decide the priority value of a node, we apply Neighbor Update and Lazy Update to update the ordering criteria (see section 4.4.1), and again we apply the maximum of 5 hops.

We can combine these two algorithms without losing reliability. ALT is an optimal algorithm, so it finds the optimal solution. As a result, ALT will find the same shorter path as Dijkstra search but in less computation time. The critical issues for the reliability of the CHALT algorithm are the stop criteria in both the preprocessing and the query. During the preprocessing of CH we stop searching for a shorter path, either if more than 5 hops are needed for a path, or if the cost of the path to the next current node is greater then the cost of path $< u, v, w >$. The query is stopped if cost of the path to the next current nodes, of both the forward and backward searches, are greater than the cost of the current shortest path candidate (see section 4.4.2). When using ALT instead of Dijkstra search, it needs to be certain that no shorter path can be found after applying the stop criteria. Since we know that the ALT estimation for the costs of the remaining path (from v to t, see section 4.3) underestimates the actual costs, it follows that no shorter path can be found. All stop criteria restrictions can be checked considering the complete cost function of ALT, so including the estimator.

In terms of our example, during the preprocessing of CHALT the landmark data is calculated and is the same as in ALT and the multi-level graph is build, which results in the same graph as in CH. This is not general the case, but only for this specific example. As a reminder, Table 4.6 gives again the landmark calculations (with node c and e as landmarks) and Figure 4.11 gives the resulting graph after the preprocessing.

So for finding the shortest path, ALT is used instead of Dijkstra's algorithm. Figure 4.12 illustrates the CHALT algorithm. Note it is very similar to CH, however the algorithm finds point in good directions faster (first node e instead of c as in CH). In Figure 4.12(c) node *e* is found in the forward search, but the backward search did not find *e* yet (Figure 4.12(d)), so the the forward search continues, see Figure 4.12(e). The *e* is also found in the backward search and 7 (current shortest path candidate) is less than 12 (next current node) so the shortest path is found. Figure 4.12(i) gives the resulting path, combining both search results.



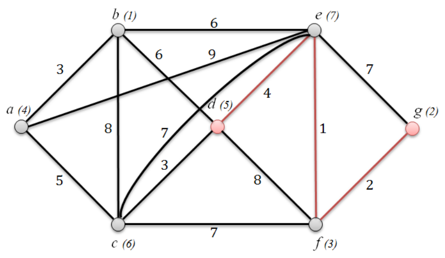
(a) Forward: *d* is current node **(b)** Backward: *g* is current node

(c) Forward: *e* is current node **(d)** Backward: *f* is current node

(e) Forward: *c* is current node **(f)** Backward: *e* is current node

(g) Forward: resulting path **(h)** Backward: resulting path

Fig. 4.12: Example CHALT Algorithm



(i) Combined searches shortest path

Fig. 4.12: Example CHALT Algorithm (cont.)

5. NETWORK PROPERTIES

In our research we want to incorporate practical information in route planning algorithms to achieve better to real situations adapted routes. To do so, algorithms may have to change. This chapter considers the adaptations needed for road network properties. We evaluate the road network properties separately from the other information types. The reason is that all other information types are time dependent and road network properties, as we define them, are not time dependent. If there is a road property that has time dependency it is considered as part of the time dependent data information.

Road network properties include every possible restriction of road segments, such as road width and turn restrictions. For this research we assume:

Assumption 1. *The road network properties are assumed to be readily retrievable from the data.*

To include road properties in route planning it is for most properties possible to save a restriction together with the corresponding arc. For instance, if a road segment is restricted with a certain weight capacity, this information can be saved. At the moment of planning, the exact problem instances are known, like the vehicle type to plan for. Consequently, it is possible to check such restrictions during the planning of the route, so whether the vehicle can be allowed on a road segment according to its weight. Such a restriction in terms of the mathematical formulation, given in Chapter 2, can be formulated as:

$$b\, x_{ij} \leq A_{ij}, \tag{5.1}$$

where A_{ij} is the allowed amount of arc (i, j), for instance the weight capacity. Further, b is the amount regarding the problem instances, so like the vehicle weight, and $x_{ij} = 1$ if arc (i, j) is in the route. Notice that if arc (i, j) is restricted, the arc is only allowed in the route if the $b \leq A_{ij}$. All restrictions concerning numerical capacities of arcs can be expressed by such a restriction.

If there are arc properties that just prescribe whether or not the arc is allowed for a certain problem instance, like truck allowance or road type, the restriction can be formulated as:

$$x_{ij} \leq D_{ij}, \tag{5.2}$$

D_{ij} is a indicator matrix, that indicates whether or not the arc is allowed given the problem instances. In the case of truck allowance, D_{ij} contains a zero if the truck is not allowed and an one otherwise. Suppose a route needs to be planned for a truck, but this truck is not allowed on all arcs. Then arc (i, j) would be part of the route, x_{ij} will be one, which can only be if D_{ij} is one and thereby denotes that the truck is allowed on that arc. Whether or not to use such a restriction depends on the problem instances, if a route is planned for a car, a truck allowance check is not needed.

Inserting these restrictions into the algorithms causes in most cases no problems. Only in combination with CH or CHALT problems arise. Since the restrictions depend on the problem instances, the preprocessing cannot correctly cope with these restrictions for a general case. During the preprocessing of CH (and CHALT) shortcuts are constructed. These shortcuts represent the shortest path between two nodes. However when some arc are no longer allowed by a restriction, it can no longer be guaranteed that the constructed multi-level graph can find the optimal solution. Consider the example graph, used in Chapter 4, after the preprocessing, see Figure 5.1. Now suppose that by a restriction arcs

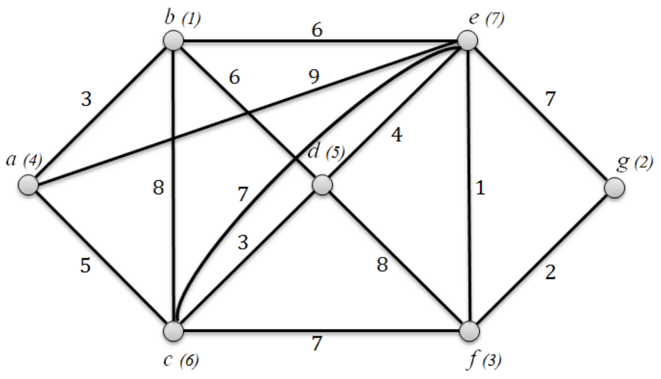


Fig. 5.1: Resulting graph for CH after preprocessing

(b, e) and (c, d) are no longer allowed. That means that shortcuts (a, e) and (c, e) are also no longer allowed. If then a route should be planned from node a to node e, this gives a problem. We can only search upwards, so from a we go to c and at e we cannot go further since it is the highest node. From c we also cannot continue since the connection between c and e is not feasible. So this means no route can be found by CH or CHALT. Note however there still is a path from a to e, more precise there are two paths with the same costs that now become the shortest paths from a to e: $< a, b, d, e >$ and $< a, c, f, e >$ both with a value of 13.

For this reason these restrictions should be considered during the preprocessing phase of CH and CHALT. A drawback is that more than one preprocessing is needed. However, most restriction can often be placed in categories, therefore it suffices to do the preprocessing for the different categories. For example, weight capacity of a road causes problems for vehicles above a certain weight.

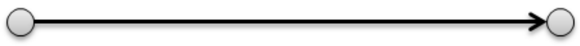
Restrictions that cannot directly be considered as previously described are one-way streets and turn restrictions. We will therefore discuss these properties in more detail.

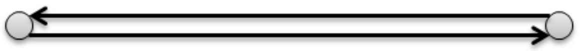
5.1 One-Way streets

In the case of route planning, the road network is represented by a graph. There are several different ways to construct a graph from an actual road network. We explain how the graph is constructed for this research, so that one-way streets can incorporated.

As mentioned before, the graph needs to be a directed graph and we stated that arcs represent roads and nodes represent junctions. Using a directed graph, one-way streets give no problems. However, there is a choice in how to represent two-way streets. It is possible to represent a two-way street by one arc, together with the information that it is allowed to use this arc both ways. The other option is to represent it by two different arcs with opposing directions. In order to achieve the best representation of actual the road network, we use the latter representation.

To achieve an even better representation, we also distinguish between two-way streets, where the lanes are and are not separated by a verge, provided this can be gathered from the data. If roads are separated by a verge, junctions of the two directions are also represented by separate nodes, whereas, if there is no verge, one node is used for both directions. As a consequence, three different kind of road representations exists within the graph. Figure 5.2 gives the graphical representation of those three road representations.


(a) One-way street


(b) Two-way street without verge

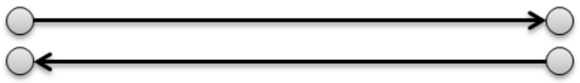

(c) Two-way street with a verge

Fig. 5.2: Different representations of roads.

This type of representation has consequences for the representation of crossroads. Assume a crossroad connecting four different directions. In the case that none of the directions have roads separated by a verge, the representation is as given in Figure 5.3. However, when all roads from all directions do have verge, the crossroad is represented as in Figure 5.4. Notice that it is still possible to go from one direction to all other directions. Figure 5.5 gives a representation of a crossroad, where all directions have a verge, except one.

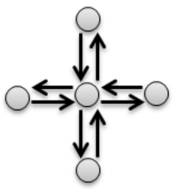


Fig. 5.3: Crossroad: No verges

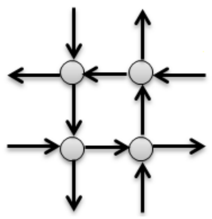


Fig. 5.4: Crossroad: With verges

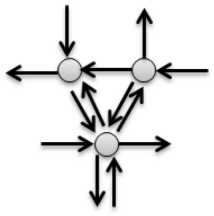


Fig. 5.5: Crossroad: Mix of verges

Notice that this choice of network representation is very detailed, and uses a lot of storage space. However, the goal is to give a realistic view as much as possible. Therefore we choose this representation. A key advantage of this representation is that turn restrictions are easier to incorporate.

5.2 Turn Restrictions

Turn restrictions also need some additional attention. Arising problems due to turn restrictions are best illustrated by an example. Consider a crossroad as in Figure 5.6. Assume a vehicle comes from x and is not allowed to turn left at node v (see the red arcs). This does not follow from this representation of the crossroad, and therefore could be planned by the algorithm. To cope with these restrictions, two different approaches can be used. One option is to store the turn restriction with an arc (similar to the previously described restrictions). However, this is not straightforward, since the restriction depends on more than one arc. The other option is to adjust the graph that represents the road network, so it becomes impossible that the algorithm plans, for example, to turn left (as in the example in Figure 5.6).

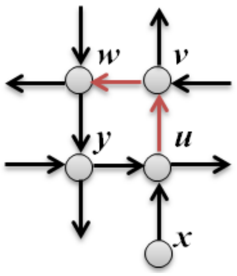


Fig. 5.6: Crossroad representation

We chose to store the restriction with an arc. The reason is that if we adapt the graph (either by removing or adding nodes and arcs), problems arise if the source, or target node is an node that was adapted to make a correct representation of the network. Another reason is that adapting the graph leads to more nodes and/or more arcs, so the size of the graph increases, which is not beneficial. In terms of the mathematical formulation the turn restrictions can be formulated as:

$$x_{li} + x_{ij} \leq I_{lij} + 1, \tag{5.3}$$

where I_{lij} is a three dimensional indicator matrix, that indicates whether arcs (l,i) and (i,j) are allowed to be part of the same route. So, if arcs (l,i) and (i,j) are both in the shortest path, but I_{lij} is zero, the path is not allowed.

Since we use algorithms we can use a more efficient way to check these restrictions, since we can assume not very many arcs are restricted in this way. Every arc subjected to this type of restriction is kept in a list. In addition, for each of these arcs there is a list that specifies the other arcs involved in the restriction. To clarify, again consider Figure 5.6 where it is restricted to turn left, coming from node x. Arc (v,w) is added to the list with restricted arcs, and is assigned the restriction of arc (u,v).

Note that the restriction only exists of two arcs per turn restriction. Meaning that, in Figure 5.6, it is also no longer possible to go from node y to node w. However, if it is not allowed to turn left, it is also not allowed to make a u-turn involving that left-turn, and therefore we could construct the restriction in the way described previously.

Knowing these restrictions, we need to check them during the algorithms. For each node that has been encountered, the predecessor is saved (in order to retrieve the shortest path later on, see section 4.1). If we expand from node v over an arc (v,w), we can check whether $((u,v)(v,w))$ is in the list of turn restrictions. If this is the case, this combination of arcs is not feasible, if not we can include the arc. This checking procedure can be applied within all the algorithms considered.

Still, a problem remains with algorithms that use shortcuts. Within the shortcuts no problem occurs, since only feasible shorter paths are checked for, and when two arcs connected to the contracted node are restricted, they do not even have to be checked. However, if in a route a shortcut becomes connected with another arc, that connection needs to be checked for feasibility. This is done by saving the shortcut in a list if one of the ends (or both) is restricted. Thereby, we also save which original arc is the actual restricted arc, so that during the route planning the original arc can be used to see whether a path is allowed.

6. TIME DEPENDENCY

Traffic information, weather conditions, event occurrences, and time dependent data are the remaining types of practical information to incorporate within the algorithms. Traffic information, weather conditions, and event occurrences differ over time and additionally, time dependent data comprises information about the network regarding specific moments in time or time intervals. So, these types of information are all time dependent. Incorporating time dependency in route planning causes problems within the algorithms. In this chapter we will first explain how the practical information will be considered during the route planning. Subsequently, adaptations to the algorithms are discussed that ensure the algorithms can cope with time dependency.

6.1 Information Handling

Time dependent practical information influences the travel time and feasibility of a route. To integrate these effects on the travel time in the route planning, the travel time will be considered as part of the cost function of an arc. Consequently, the arc costs become time dependent, which will be represented by $g(u, v, \tau)$. Here τ is a point in time, that represents the departure time at node u. To handle the different effects on the travel time, we define the travel time as a linear function of the lower bound together with the effects caused by the practical information incorporated. The lower bound on the travel time is given by the travel time based only on the maximum speeds and the distance of an arc. For this research we will express the arc costs only in terms of travel time and so the arc costs at a certain time τ can be given by:

$$g(u, v, \tau) = \lambda + \delta_\tau \lambda + \eta_\tau \lambda + \theta_\tau \lambda + \omega_\tau \tag{6.1}$$

Here λ is the travel time lower bound. δ_τ is the effect of the traffic information on the travel time, given time τ, as a fraction of the lower bound. η_τ is the effect of the weather information, in terms of amount of precipitation, on the travel time the given time τ, and θ_τ is the effect of the event information on the travel time given time τ. ω_τ is the waiting time given time τ caused by other time dependent information. Further, we assume:

Assumption 2. *The effects δ_τ, η_τ, θ_τ, and ω_τ are assumed to be known from the data or other research.*

In Chapter 7 we will discuss the data used for the computational results of this research and how exactly we use this data to determine the arc costs based on the previous described idea.

6.2 Algorithm Adaptations

If the arc costs are time dependent, the algorithms need to be adapted in order to result in feasible and realistic routes. The first change is that also the departure time is needed as input, together with the source and destinations, to determine the best route. Besides the departure time, the algorithms must also keep track of the time throughout the query, to calculate the correct arc costs. This means more information needs to be saved during the planning. Time dependent arc costs also implies that cost for the same route can differ depending on the departure time. To make sure the problem does not become NP-hard we make the following assumption (Kaufman and Smith [17] and Nannicini and Liberti [21]):

Assumption 3. *We assume the FIFO property, implying that for a pair of time instants* τ, τ' *with* $\tau \leq \tau'$ *holds:* $g(u, v, \tau) + \tau \leq g(u, v, \tau') + \tau'$

The FIFO property basically states that when X_1 leaves from u at τ and X_2 from u at τ', where $\tau \leq \tau'$, it is not possible that X_2 arrives at v before X_1 using arc (u, v).

Other consequences of time dependency for shortest path algorithms are that problems arise concerning preprocessing and bidirectional search. Preprocessing is a calculation procedure that is assumed to be done once before the start of the query, and then can be used to plan any route for a certain time period. If this idea is used in the case of time dependency for a given period, such preprocessing would lead to an immense amount of data that needs to be saved. The reason for this is that for all time points in the given period, the arc costs can be different and so the preprocess may have different results. Suppose for example that there is information for every minute in a week, during a preprocess, factor $60x24x7$ times more information needs to be saved than without time dependent information. Another problem with the preprocessing arises if the available information is only current information. This causes that, to stay up to date, the preprocess needs to be done every time a route needs to be planned, leading to much longer calculation times.

The problem concerning bidirectional search is that it uses backward search from the target node. This causes a problem because only the departure time is known. However, to plan backwardly the arrival time also needs to be known, otherwise it is not possible to calculate the time dependent cost to nodes in the backward search. For this reason algorithms using bidirectional search, like CH in our research, gives problems.

We will discuss for every used algorithm (see Chapter 4) how we adapt it in order to accomplish a correct result.

6.2.1 Dijkstra's Algorithm

Dijkstra's algorithm neither needs preprocessing, nor does it require bidirectional search. As a consequence, not much needs to change in order to apply Dijkstra's algorithm in combination with time dependency. As mentioned previously, the only crucial concern is to keep track of time during the planning. Due to the FIFO properties, the rest of the algorithm remains identical.

6.2.2 A*

For the A* algorithm it is, as for Dijkstra's algorithm, crucial to track of time during the planning, however for A* also preprocessing is needed. Recall from section 4.2 that for this research the Euclidean distances are used for the calculations done in during this preprocessing. This means that no arc costs are considered during the preprocessing. Consequentially, the preprocess is not influenced by the time dependency of the arc costs. However, since now the act costs are expressed in time and no longer in distances, the estimation $h(v, t)$ used for A* does need to change. The calculated Euclidean distances should be divided by the speed to get the travel time needed for that distance. We still have to ensure that the estimations do not overestimate, so the speed used to convert distances to travel time, is the maximum speed allowed within the graph.

6.2.3 ALT

ALT is the first algorithm that requires serious changes. ALT uses preprocessing which is based on the arc costs. In addition, during the preprocessing also backward search is used, see section 4.3. To avoid the issues mentioned above in the case of preprocessing, the idea for this research is to use a fixed arc cost during the preprocessing. Note that, as mentioned several time before, a cost estimation of a path or arc cannot exceed the actual arc cost to ensure correctness. So, using an average or prediction as arc costs is not appropriate. Therefore, we will use the travel time lower bound λ for the preprocessing.

Using arc costs that do not depend on time during the preprocessing has as consequence that the results of the preprocess are based on less accurate information and therefore can be less profitable. However, this is preferred over the consequences of using time dependent arc costs.

Nannicini et al. [22] propose a well working, bidirectional search ALT algorithm for time dependent networks, on which the idea described above are mainly based. We will use and explain the whole procedure in further detail for CH and CHALT.

6.2.4 CH

CH (and therefore also CHALT) is the most complex algorithm to combine with time dependency. Besides the fact that as well preprocessing as bidirectional search are part of the algorithm, CH also encounters problems with the query if the results of the preprocessing are not correct

During the preprocess of CH, shortcuts are constructed. If a node is contracted, a shortcut is constructed if no shorter path between neighbors can be found (recall from section 4.4). So the concept of shortcuts is based on finding shorter paths in the preprocess. However, if the network becomes time dependent, it is possible that for some moments in time a shortcut should be constructed, and for other moments this shortcut will be superfluous. In other words, if in a preprocess based on fixed costs it is decided to construct a shortcut, there is always a possible scenario where the shortcut does not represent the shortest path between two points.

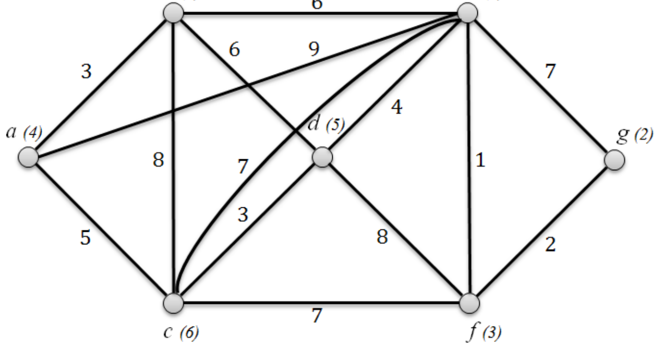


Fig. 6.1: Resulting graph for CH after preprocessing

As an example consider Figure 6.1. Suppose that this graph represents the result of the preprocessing, based on fixed costs (as given in the figure). Suppose, for some moment in time, there occurs a problem on arc (d, e), that leads to an increase of travel time to 99 for that arc. Then the travel time of the shortcut becomes 102, which obviously is not the actual shortest path from c to e anymore.

The major problem of unreliable shortcuts is that the query cannot be performed properly. The CH query consists of an upward forward search and a downward backward search. However, these searches are based on the fact that the shortcuts represent shorter paths, so that a shortest path in higher levels represents the shortest path in the original graph. But if there is no certainty whether the shortcuts represent the actual shortest path between two points, the upward and downward procedures cannot ensure to result in the actual shortest path.

To this end, there are two possible options for adapting CH. Either the preprocessing needs to be adapted to ensure a correct query routine, or the preprocessing remains as it is and the query has to be adapted to ensure correct shortest paths.

Adapting the preprocessing has as disadvantage that it has to become time dependent and it is going to take longer, because of the larger amount of information that needs to be considered. The adaptations proposed in Batz et al. [3] is the best known algorithm for this problem in literature so far. The idea is that if it is possible to state that a shortcut represents the shortest path for all moments within a time interval, the shortcut can be constructed. To prevent very large calculation times they propose to use profile search to make a decision about the shortcut in a more effective manner, and in Batz et al. [4] approximations on the shortcut costs are proposed to decrease the number of profile searches which decreases the preprocessing time even more. However, in these articles it is not considered how to handle situations in which shortcuts may only be needed during certain time intervals, which is exactly what we need for this research. So, these adaptations cause the preprocessing to be time dependent and it not even solves the whole problem.

The other option is to ignore the reliability of the shortcuts and perform the preprocess with a fixed cost, for example the lower bound λ, and then construct a new query routine. This new query can still partly use the upward and downward procedure, but only if all outgoing arcs of a node are either original arcs, or shortcuts with cost equal to the travel time lower bound. If this is not the case, the pruning of lower levels is not allowed. Therefore, the query of CH can be adapted by allowing lower levels in the case the costs of a shortcut are larger than its lower bound. To prevent that no similar problems arise by adding these lower levels, we only add original arcs instead of the considered shortcut. Doing this, the normal rules of the query should still be considered, so a path to a node (over such an original arc) is only saved if it is a shorter path, compared to a possible already consisting path. After allowing these lower levels, the algorithm can continue with the procedure of search upwards. Consider again the example in Figure 6.1. When applying the adaptations, as just proposed, on finding a route from c to e, with (d, e) having cost 99. Then, we find that shortcut (c, e) has larger costs than its lower bound (which was 7) and we will add the paths to nodes b, d, and f. Now, by continuing the upwards search, we can find the path $< c, f, e >$ which is the actual shortest path.

A disadvantage of these adaptations is that the query will result in a larger search space. Especially when route planning evolves to a situation where the actual costs of an arc are never equal to its lower bound, this will be become a bigger problem. The adaptations previously proposed have not been discussed in literature to the best of our knowledge.

The last remaining problem of CH is caused by bidirectional search. The main part of Batz et al. [3] is devoted to tackle this problem for CH, and Delling and Nannicini [7] and Nannicini et al. [22] consider this problem for other algorithms. In Delling and Nannicini [7] bidirectional, time-dependent search is applied on a core-based graph. CH produces a core-based graph, so the procedure would also be suitable for CH. However, since within CH one level only consists of one node, the procedure proposed by the article is somewhat inefficient. In Nannicini et al. [22] they propose a modification to bidirectional search for ALT and on this approach Batz et al. [3] based their ideas for adapting CH.

The idea for the bidirectional search is as follows. Start the forward search from the source with the time dependent arc costs. The backward search is also initiated, but uses only the travel time lower bound λ. If the searches connect for the first time, a lower bound ϕ on the entire (s, t)-path can be determined. This lower bound is given by the time dependent cost of the forward shortest path till the connecting point, together with the travel time lower bound λ of the backward search till the connecting point. In a similar way an upper bound on the route can be calculated. Every time a connection is found, several actions have to be done. First, this connection is saved. Secondly, a check is done whether the lower bound is smaller than the tightest found upper bound and if not, the forward search is pruned from the connecting node. Finally, the upper bound is improved if a tighter bound is found. This procedure is applied until no more paths can be found. Once this happens, there will be several meeting points between the forward and backward search, forming candidate shortest paths. For all these candidates a forward search is performed from the meeting point to the target, using the time dependent arc costs. This forward search only investigates nodes that where also investigated by the backward search. The resulting shortest path is the actual shortest path.

Since we propose a method to allow lower levels to ensure correct routes, we also need to adapt ideas for the bidirectional search. In all forward searches we allow the lower levels as described previously. As a consequence we can no longer limit the second forward search to only search the nodes that where also investigated by the backward search. Therefore this will not be done in our algorithm, again causing an increase in search space.

For CHALT we will combine the adaptations, discussed above, of both CH and ALT.

7. DATA

In this research we investigate the impact on algorithms when incorporating practical information. To do so, we gathered the practical information needed from several data sources. In this chapter, we specify the sources that provided our data and discuss how the data is used in our research.

7.1 Road Network

Route planning applications plan a route from one point to another. To enable such an application to do this, a network is needed. Since this research focusses on route planning for road networks, a road network is desired.

OpenStreetMap[1] (OSM) is a young open source project, that was founded to stimulate the growth, innovation and distribution of freely available geographical information. Since 2006, volunteers all around the world have been updating the information provided and shared by OSM.

For our research we downloaded different maps of OSM. We consider relatively small maps compared to maps used in related literature. The reason is that, keeping in mind the performance of the computer, the route planner should be able to run sufficiently fast with all the incorporated practical information. In literature the map of Western Europe with approximately 18 million nodes and 42.6 million arcs is often used, which is not practicable for this research.

Via JOSM[2], we retrieved a part of Zoetermeer, see Figure 7.1(a). This maps is chosen

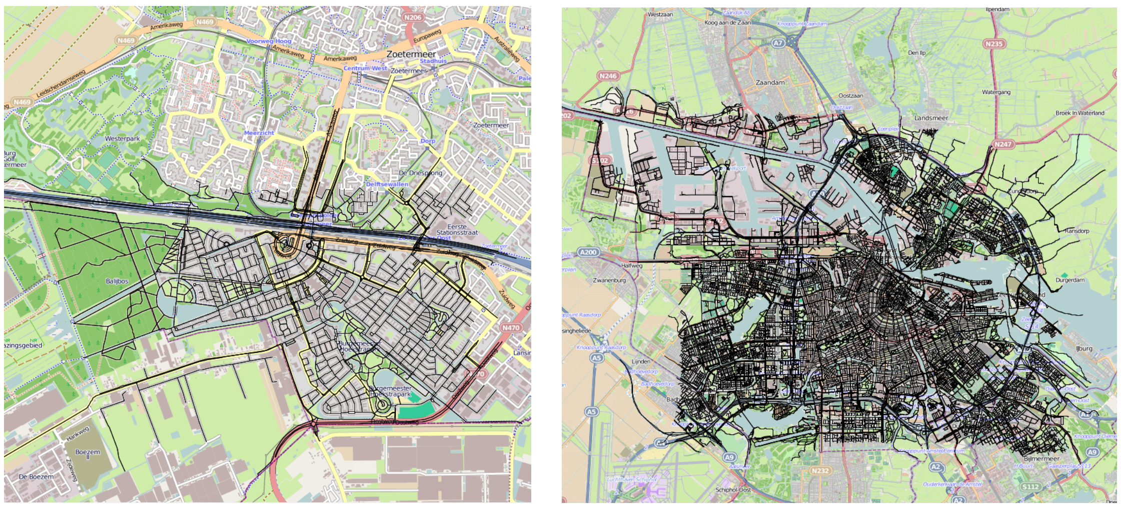
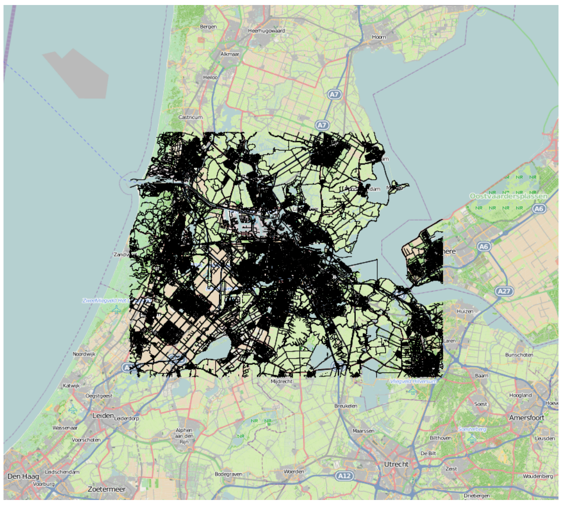

(a) Part of Zoetermeer **(b)** Amsterdam

Fig. 7.1: Maps used for this research

[1] http://www.openstreetmap.org/
[2] http://josm.openstreetmap.de/



(c) Part of Noord-Holland

Fig. 7.1: Maps used for this research (cont.)

because this map is considered as a small map (see Table 7.1 for details), so this map will not cause any capacity issues regarding the computer memory. From JOSM we also retrieved a map of Amsterdam, see Figure 7.1(b). This map is larger than the first one but still not very large. We selected Amsterdam, because it is a large city, with a large amount of one-way street and possible restrictions. The last map we used was downloaded from Metro Extracts[3], and contains a part of a Dutch province called Noord-Holland, see Figure 7.1(c). The map includes again Amsterdam but this one also includes surrounding cities. This map is chosen to construct routes that also require highways.

In section 5.1 we discussed how to translate a map into a graph, so it can be used by the algorithms. We applied this procedure on the maps we used. Table 7.1 gives the resulting details for all used maps. Within these details one-way streets are already considered.

Map	**# Nodes**	**# Arcs**
Part of Zoetermeer	4,047	8,425
Amsterdam	77,106	158,674
Part of Noord-Holland	316,978	663,468

Tab. 7.1: Details of used maps

7.2 Road Network Properties

Within the maps of OpenStreetMap are also network properties saved. We used the information about one-way streets, road type, turn restrictions and maximum speed. One-way street, as mentioned in the previous section, are incorporated in the graph as prescribed in section 5.1. For the road type, we make a distinction between footways, bicycle paths, ferry routes, roads meant for services, and general roads that permit all motor vehicles. The road type meant for services contains roads like bus lanes, or roads

[3] http://metro.teczno.com/

on private areas, like military areas or industrial areas. The road type is added to the arcs and in that way the arcs are restricted as described in Chapter 5. Furthermore, turn restrictions are included as discussed in section 5.2, based on information known from the OpenStreetMap maps. For the map of Zoetermeer there were no turn restrictions. So, in order enable testing turn restrictions within this map, we selected five nodes on which we added a restrictions. The maximum speed is used to compute the travel time lower bound, as defined in Chapter 6. Table 7.2 gives some details about the road network properties, for every map we use.

Map	**# Turn Restrictions**	**# General Roads**	**# Highways**	**# Footways**	**# Bicycle Paths**	**# Ferries**	**# Services**
Part of Zoetermeer	5	7,790	11	241	285	0	98
Amsterdam	32	147,399	429	3,639	5,220	11	1,976
Part of Noord-Holland	93	629,897	1,194	10,472	15,165	48	6,692

Tab. 7.2: Details about the road network properties for the used maps

7.3 Traffic Information

For this research we want to incorporate traffic information. In the Netherlands a lot of traffic information is stored by several authorities and companies. This information is stored in an immense database, which is maintained by the National Data Warehouse for Traffic Information (NDW)[4]. Currently, the NDW has more than 20.000 measurement locations within the Dutch road network. This covers around 5400km of highways, provincial roads and city roads, see Figure 7.2.

However, the database of the NDW is not freely available. Nevertheless, the DNW was willing to provide us one sample day out of their database. To be more specific, they provided all available information for August 28, 2012 from 00:00 to 23:59. They also arranged a set for us, containing all measurement locations in terms of latitude and longitude positions.

The data we received, concerning the 28th of August 2012, contained a generous amount of information. For every minute of the day, several different measurements are stored. These measurements consists of information about traffic intensities, the average traffic speed, and the average travel time. Within this information, the NDW distinguishes between measurements on a location and measurements over an route. The measured traffic intensities and the average traffic speeds are measurements on location, and the average travel times are route measurements.

The traffic intensities give the number of vehicles passing by a measurement location within a given minute. The average traffic speed is the average over the speed measure for all vehicles passing at one point, in a given minute. For the average traffic speed also the standard deviation is known and the number of vehicles on which the average is based. The average travel time is measured as the time that passes as a vehicle goes from one measure point to another and the average is taken over all vehicles passing within a given minute.

For our research we will only use the measured average traffic speed. One reason is that for traffic intensities no exact effect on the travel time is known. However, we do

[4] http://www.ndw.nu/

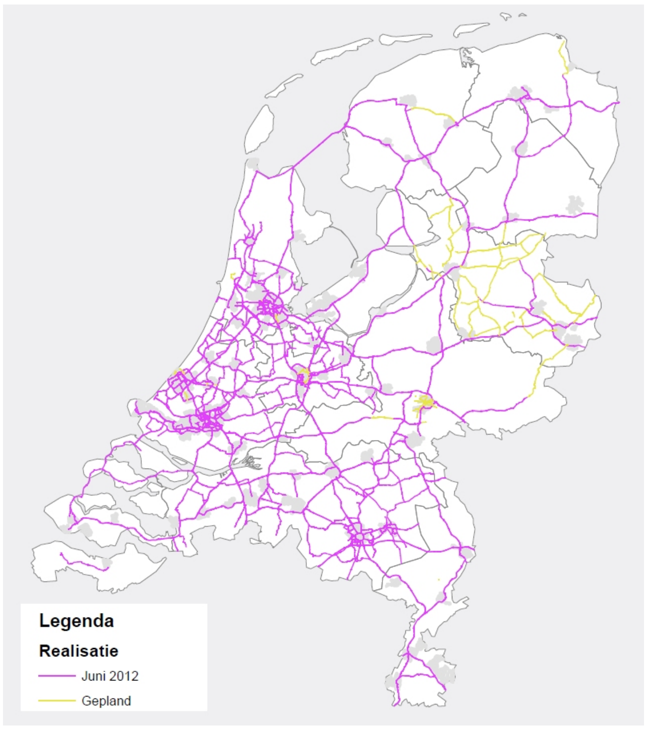


Fig. 7.2: NDW road covering at June 2012

know the distances by the information from maps, so the effect of the traffic speed on the travel time is clear. The reason we do not use the average travel time is because we have to translate the information into our graph. It is unlikely these routes are exactly equal to the routes on our graphs, and therefore it is more desirable to use point measurements.

Unfortunately, it is not possible to calculate a reliable prediction for traffic influences on the travel time of a route on any day with this data. Accordingly, we will use this day as if it is the prediction. August 28, 2012 is a working day at the end of a holiday period. Nevertheless, it is assumed to be representative enough for the purposes of this thesis, as average working day. Another problem of the data as we received it, is that it cannot be ensured that the FIFO property holds. It is possible that by very large changes in the measurements, departing later will lead to arriving earlier at the destination. We will ignore this fact and apply the algorithms as if the FIFO property does hold, since this will only be the case for rare cases.

To incorporate the information of the database into the graph we have constructed, some calculations need to be done. The measurement locations are not linked to our downloaded map, so we determine the nearest arc to the measurement location. Doing this, we keep in mind that the locations are on a sphere and not on a flat surface. Having this link and the measurements corresponding to a measurement location, during the routing we can retrieve the average speed per minute of the arc nearest to the measurement location.

Table 7.3 gives for every map how many measurement locations fall within the area of that map. Since there are far more arcs than measurement locations, especially for the larger maps, we expect that not only the nearest arc is affected by the predictions as they are measured. Therefore, we assume that all arcs that are within five hundred meters of the measurement location are also affected. The effect is weighted based on the distance from the measurement location. So in other words, the further the arc is from

Map	**# Measurement Locations**
Part of Zoetermeer	1282
Amsterdam	3040
Part of Noord-Holland	4589

Tab. 7.3: Number of measurement locations for the used maps

the measurement location, the less the travel time of that arc is effected by the traffic data.

7.4 Weather Information

To include weather information in the graph, weather predictions are collected from the Royal Netherlands Meteorological Institute (KNMI)[5]. The KNMI is the Dutch Institute that provides weather forecasts. These forecasts can take different forms, like predictions for the temperature, the probability for sunshine or the amount of rainfall. These forecasts are broadcast on television, radio, and on the internet, to ensure the public is informed.

For our research we use the radar images of the precipitation predictions. All predicted weather forms could affect the travel time, however, since they are all strongly correlated, precipitation is assumed to have the most effect, and because radar image are easier to translate to our graph, we choose to use this information.

The radar images of an area as the Netherlands can be divided in very small images for local usage. For each of such a radar image, two geographical locations are known: the lower left corner and the upper right corner. In that way the images can be coupled to the maps we use. The radar images themselves contain information about the predicted amount of precipitation in kg/m2/h (this unit is assumed by the KNMI to be equal to mm/h). Different amounts are expressed with different colors. To handle the weather information, we grouped the different amounts in five different precipitation levels, see Table 7.4. We do this, because it is reasonable to assume that different amounts of precipitation within the same level have similar effects on the travel time.

Precipitation Level	**Amount of Precipitation (mm/h)**	**Effect On Travel Time**
0	< 0.316	0 %
1	0.316 - 1.000	1 %
2	1.000 - 3.162	5 %
3	3.162 - 10.00	15 %
4	10.00 - 31.62	50 %
5	31.62 - 100	200 %

Tab. 7.4: Scaling precipitation levels in mm/h and corresponding effects on travel time

Unfortunately, we have no information about what the effects of a certain precipita-

[5] http://www.knmi.nl/

tion amount is on the travel time, so we make an educated guess about these numbers. This does not affect the results of this research, since we want investigate the effects of incorporate this information on the algorithms, instead of investigating the effects of certain numbers on the routes. In Table 7.4, the factions used as effects on travel time per precipitation level are given. These are fractions of the travel time lower bound, which will be added to the arc costs as additional travel time, see section 6.1.

7.5 Events

No real data is available on event occurrences. Therefore, in order to still incorporate events, we simulated 100 pseudo random events. For each simulated event, the following information is randomly selected from a uniform distribution:

- A geographical location within the area of the Netherlands.
- The start time (in minutes), on which the event causes problems for the travel time.
- The time period (in minutes) the event keeps causing problems.
- The distance (in cm) for which the event influences the travel time.
- The additional travel time that is caused by this event.

The geographical location are selected by drawing from a uniform distribution of which the intervals are based on the bounding box of the corresponding map. These locations are linked to the graph in the same way this is done by the traffic measurement locations.

The event causes additional travel time for a certain period of time, starting at a given moment. Both the time period and starting moment are drawings of a uniform distribution on [0, 1439], which represents all minutes in one day. So, the maximum on the duration of an event is set on one day.

The distance that is selected for an event is similar to the five hundred meters used for the traffic information, only for the event is is not set to a fixed number, but has a maximum of one kilometer. So, the distance is uniform on [0, 99.999], which is the distance in cm.

Since there is no information about what the effects of an events are on the travel time, we simply simulate a faction, which is the fraction of the travel time lower bound. This fraction are uniform on [0,1] and are give the additional travel time in terms of the lower bound caused by the event.

7.6 Other Time Dependent Data

For the remaining time dependent data there is also no real data available. Therefore, we also simulated 100 time dependent data. The information are drawing of uniform distributions and for each simulated time dependent data is simulated:

- The arc that is affected by this time dependent data.
- The start time (in minutes) of this time dependent data.
- The time period (in minutes) that the time dependent data takes before the arc can be used again.

The arc is selected based on it ID, and therefore is based on a uniform distribution over all arc IDs. The start time and time period are again uniform distributed [0, 1439], representing all minutes in a day. The difference between a event and other time dependent information is that an time dependent data only affects one arc and the corresponding effect is waiting time for the full duration of it.

8. EXPERIMENTAL RESULTS

We investigate the effects of including several kinds of practical information on different shortest path algorithms. In this chapter the experiments, used to test these algorithms, are explained and we will show the results of these experiments. The experiments are executed on a quat core notebook, with intel® core^{TM} i7-2670QM processor, 2.2 GHz and 4GB work memory.

This chapter is divided in three different parts. In the first section we show how all algorithms work on a real network with one single route. Secondly we will demonstrate what happens to a route when each kind of information is added. Finally, we will present the results based on 1000 randomly selected routes with respect to the performances of the algorithms.

8.1 Different Algorithms

In this research we test five different algorithms with respect to how they perform after adding all different kinds of information. However, before we going into further details about these results, we will first look at how the algorithms originally perform. We already showed how each algorithm works with a very small example in Chapter 4, however now we will pay attention to how the algorithms work on a real network. This is done according to one example route, within Zoetermeer.

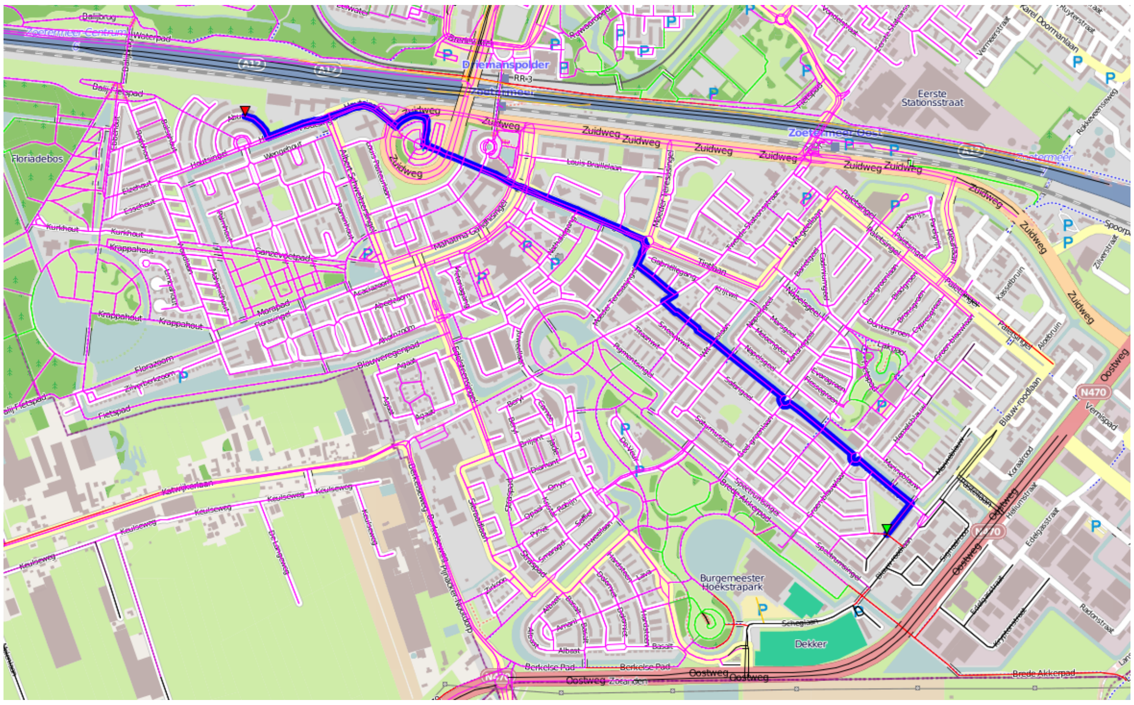


Fig. 8.1: Dijkstra's Algorithm

At first we do not consider road types, maximum speed, turn restrictions, or any time dependent information. These will be added in the next section. One way streets on the other hand are already considered. This is done in order to give a correct representation of the network. In order to calculate the travel time over an arc we set a default speed equal to 50 km/h. For all routes we consider the travel time as arc costs. Therefore, whenever we mention that a route is the shortest or best route, this is in terms of time and not distance. Figure 8.1 gives the the shortest route for the example problem found by using Dijkstra's algorithm.

In Figure 8.1 the source is indicated by the red triangle and the destination by the green triangle. Notice in Figure 8.1 that arcs are represented by several different colors. The meaning of the colors is as follows. First of all, the thick blue line is the resulted shortest route from the source to the destination. The arcs that are magenta colored, are arcs that were investigated during the search of the algorithm, so these are part of the so called search space. We will see that the number of magenta colored arcs will decrease as we go on. Note that in Figure 8.1 almost all arcs are magenta colored. The remaining colors will be seen more often in next figures. Black arcs represent the general routes, red ones the bicycle paths, green represents footways and orange arcs are roads for services. To be clear, within this section we show the different road types in the figures, however during the planning the network properties are not included yet, so each road types is treated as the same and allowed to use. Throughout this chapter we will use the same colors for the same purposes in all figures, as well as that the indication for the source and destination remains unchanged.

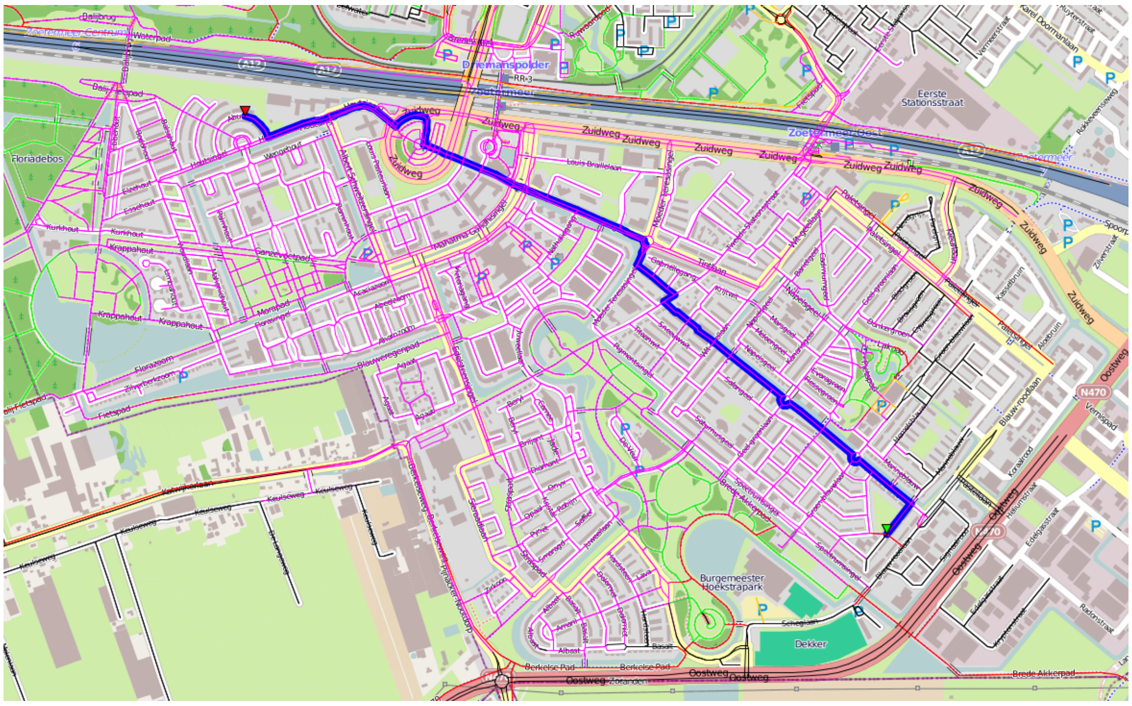


Fig. 8.2: A*

Figure 8.2 gives the route found by A*. A* should find a path of the same cost as Dijsktra's algorithm. Looking at Figure 8.2 we see that A* not only finds a path with the same cost, but in fact finds exactly the same route. Also can be seen that A* needs a slightly smaller search space to find this path. The reason we do not see large improvements on the search space, is because the route starts at one side of the graph and end at the other side.

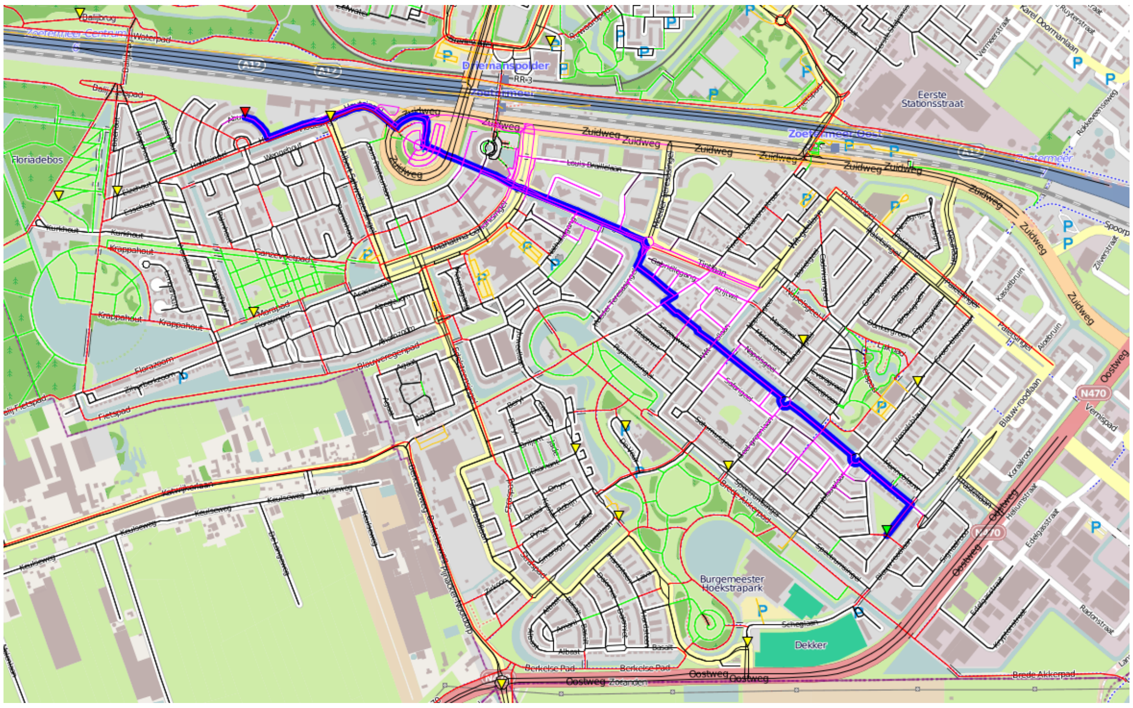


Fig. 8.3: ALT

Figure 8.3 gives the results of the ALT algorithm. Here the landmarks are indicated with a yellow triangle. Again we see that the same route is the best route according to ALT. Also note that the search space for ALT is a lot smaller. So, we see that using

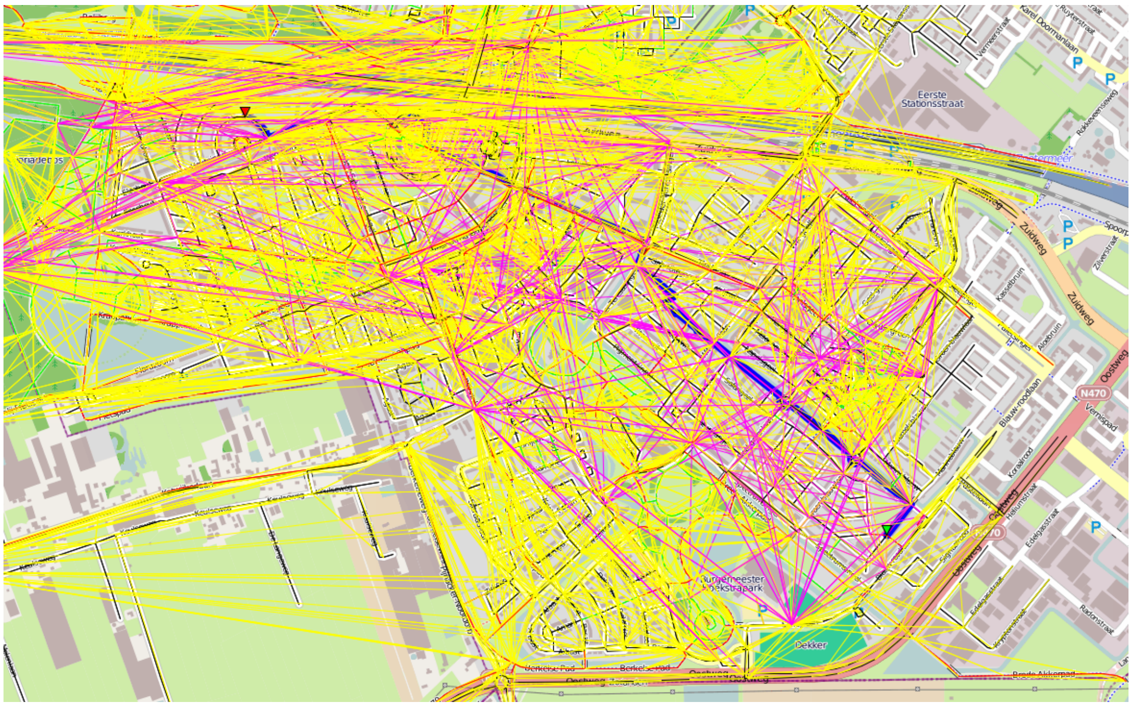


Fig. 8.4: CH

effective preprocessing pays off in terms of search space and thus in terms of computation time. Hence, when you need to investigate less arcs to find the destination, it will also take less time.

In Figure 8.4 a new type of color is introduced and the graph looks less organized. This is due to the shortcut construction that is part of the CH algorithm. The yellow arcs represent these shortcuts, which can go crisscross over the graph, replacing paths formed by a number of arcs. For CH we can also conclude that the search space is has decreased, although the amount of magenta that can be seen in Figure 8.4 does not seem very different from Figure 8.3. However, the number of arcs that are magenta colored did decrease because of the usage of shortcuts.

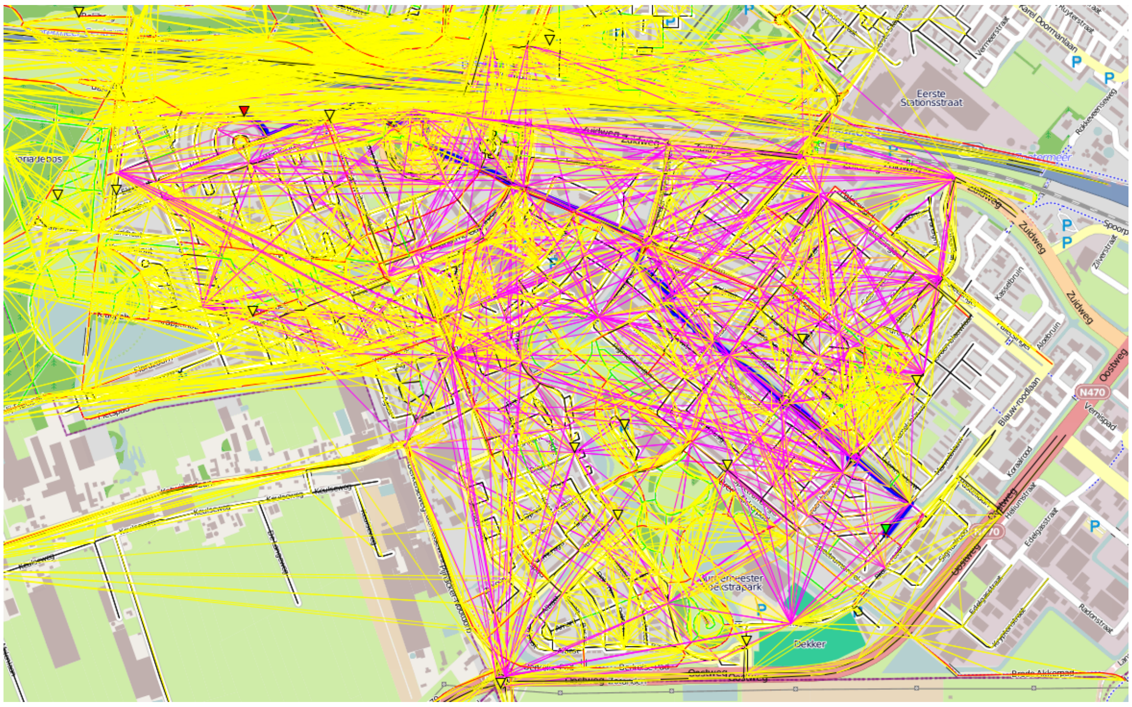


Fig. 8.5: CHALT

The last figure (Figure 8.5) shows the results of CHALT. Again, we see the yellow colored shortcuts, the same shortest route, and an decrease in the search space size.

8.2 Different Practical Information

In the previous section we demonstrated how each algorithm works without practical information. This section describes how practical information influences the shortest route. We use Dijkstra's algorithm for all computations in this section. In the figures we omit the search space, so it can clearly be seen what road types are considered. Figure 8.6 gives the route in the original situation, notice that this is the same route as in the previous section. To reach an even better understanding, we will also pay attention to the differences in total travel time of the route. The travel time of the route in Figure 8.6 is 3:24.52 (min:sec.ms).

First we consider road network properties as additional information. As mentioned in Chapter 7 road network properties include one-way streets, road types, turn restrictions and maximum speed. Note that one-way streets are already included in Figure 8.6. Considering the road types makes it possible to plan for different road users. So, in that

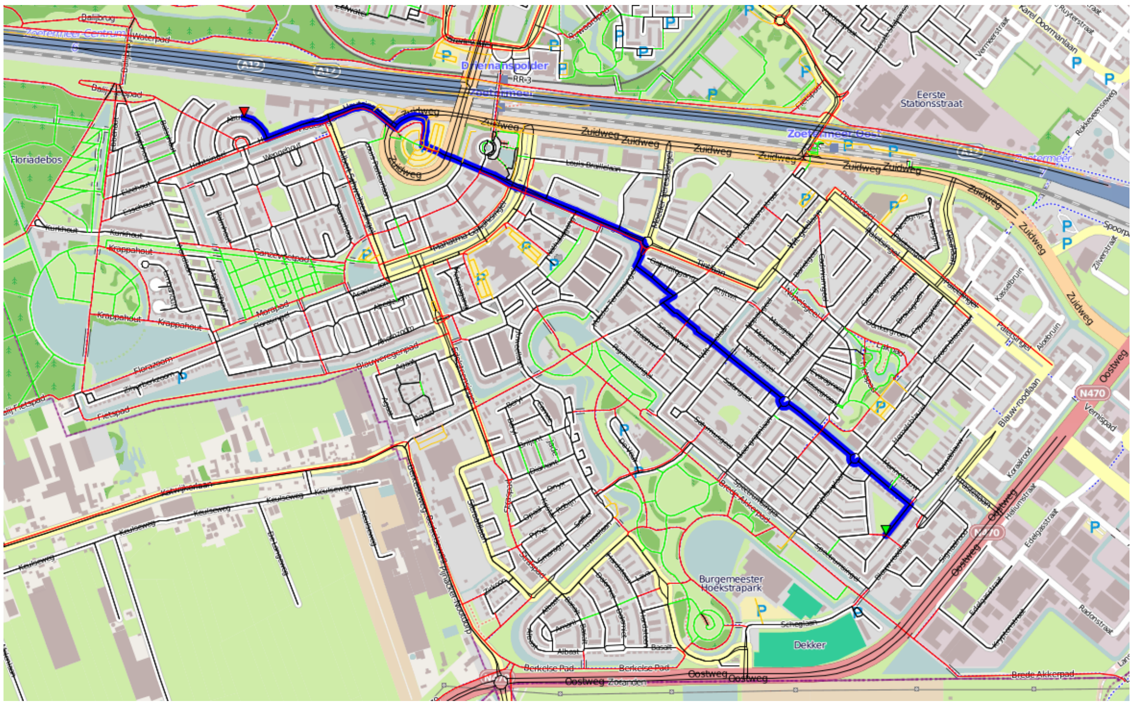


Fig. 8.6: Resulting route in original situation

way a route planning for a cyclist only considers bicycle paths. In our example, the planned route is intended for motor vehicles, such as a car. As can be seen in Figure 8.7, the shortest route no longer uses bicycle paths, footway, or service roads. Also turn

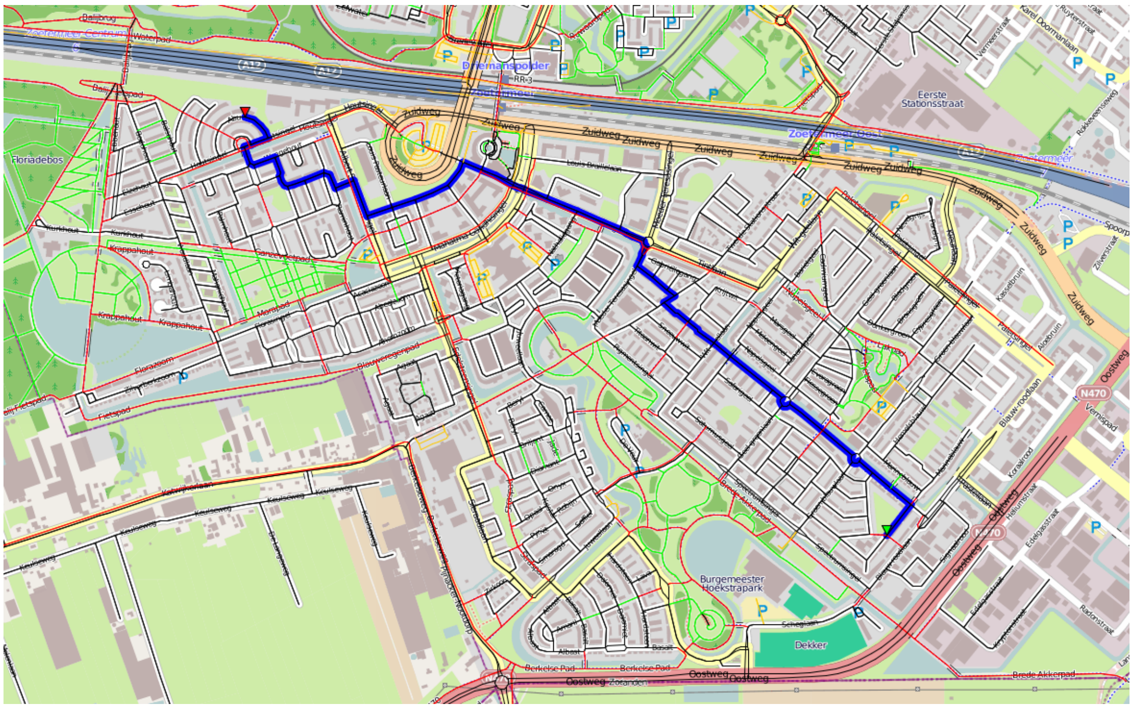


Fig. 8.7: Resulting route when adding road network properties

restrictions are incorporated. One of the added turn restrictions prescribes that it is no longer allowed to turn left at the crossroad that is first encountered by our route, see

Figure 8.7. Including the maximum speed causes no large changes. The reason is that in the original situation we used a speed of 50 km/h and the network of Zoetermeer mostly contains road with a speed limit of 50 km/h. Figure 8.7 gives the resulting shortest route after including road network properties, which has a total travel time of 3:45.18. Note that, not only has the total travel time of the route increased, also the beginning of route changed due to the turn restriction.

The next information considered is the time dependent information. We will discuss each information type related to time dependency separately, so it is clear what causes any changes in the route. We will continue to use the network including the road network properties, since that gives the most realistic definition of the network.

One of the information types causing time dependency is traffic information. We consider the following situation in our example for traffic information. Suppose somewhere along the route there is a measurement location for the traffic information. We marked this location in Figure 8.8 with a red dot. Also suppose this is the only known location. We consider the situation for which this measurement location reports that from 16.30 hrs till 19.00 hrs the measured traffic speed is 10 km/h instead of 50 km/h. This information is incorporated as discussed in Chapter 6 and 7. If in this situation there comes a request to plan a route from the source to the destination as before on 17.15 hrs, this results in the route as given in Figure 8.8. The total travel time of this route is 3:50.08 and we can conclude instead of using the route that apparently is troubled with some traffic congestion, we take another route.

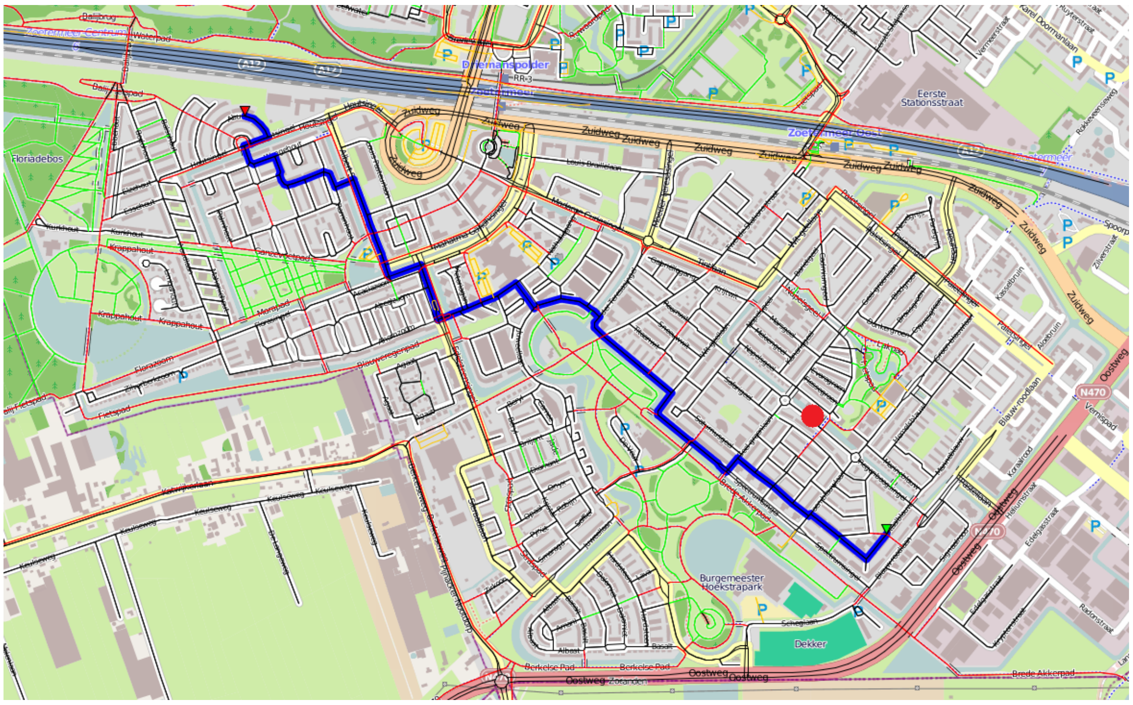


Fig. 8.8: Resulting route when adding traffic information

The next information we consider is weather information in terms of precipitation levels. Suppose it is predicted that on a day, between 12.00 hrs and 14.00 hrs there will fall 5,1 mm/h and this is the same for the entire city of Zoetermeer. From this information we know, considering the distribution discussed in Chapter 7, that this rainfall is of precipitation level three, causing fifteen percent additional travel time. Assume we want to depart at 12.35 hrs on that day. The shortest route for this situation is given in Figure

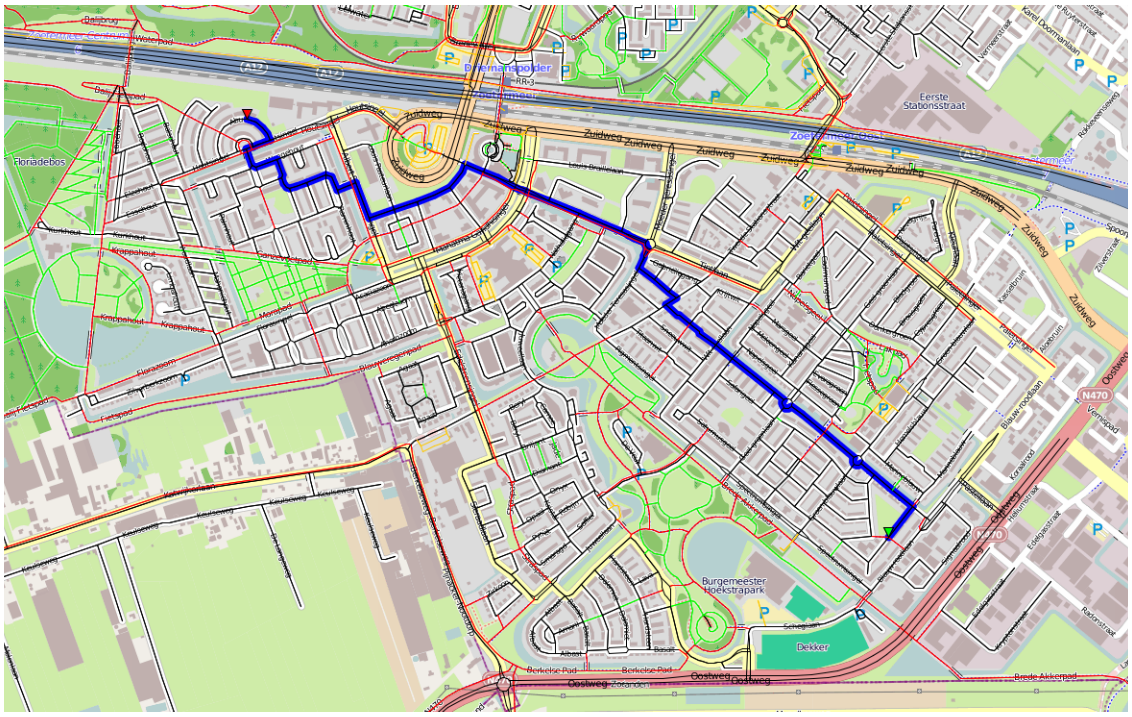


Fig. 8.9: Resulting route when adding weather information

8.9, with a travel time of 3:48.96. Note this is the same route as in Figure 8.7, but takes about 3,8 seconds longer travel time. This makes sense, because the weather influences the travel time of all arcs in the network and therefore it cannot profitable to take another route. The increase in travel time is not exactly 15 percent, due to rounding errors.

For event information we consider the following situation. At 08.00 hrs an event

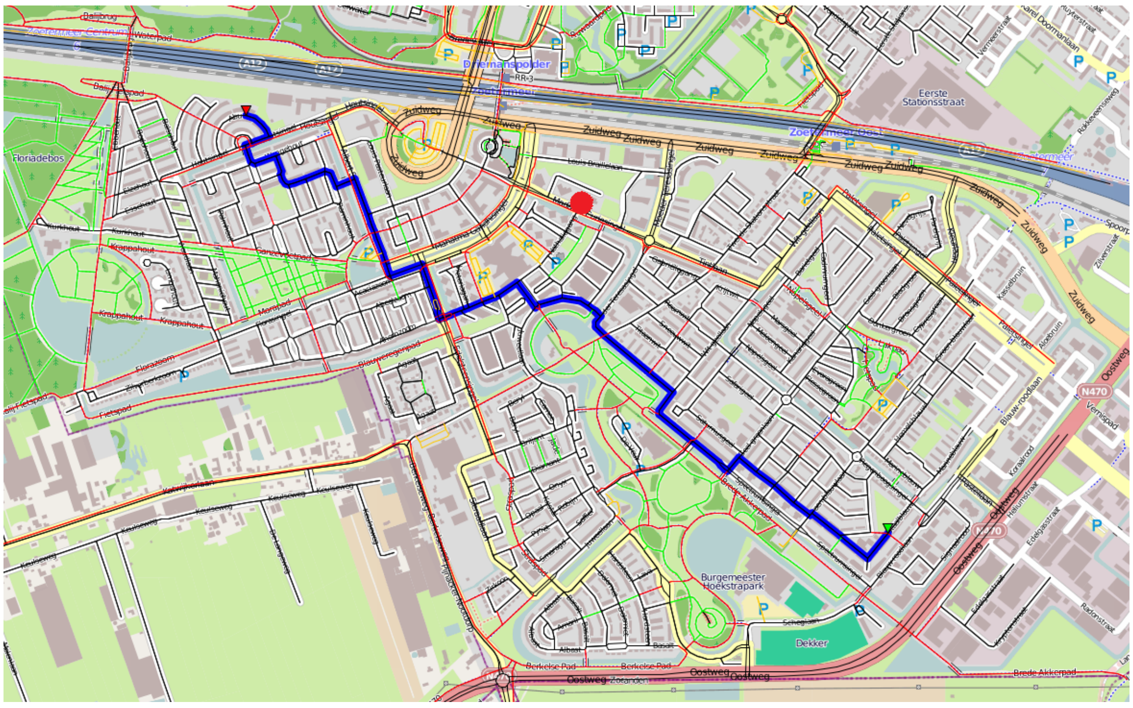


Fig. 8.10: Resulting route when adding event information

occurs, for example an accident, on the location indication with a red dot in Figure 8.10. This event causes delay for an hour. The impact of this event is thirty percent additional travel time for the area with a diameter of 200 meters surrounding this location. Suppose that in this situation the requested departure time is 08:10 hrs, then the resulting shortest route is presented in Figure 8.10. This route has a total travel time of 3:50.08. Observe that this is the same route as in Figure 8.8 and that the travel time is also the same. The reason is that for both situations it is more profitable to take another route, then to encounter the discussed delay.

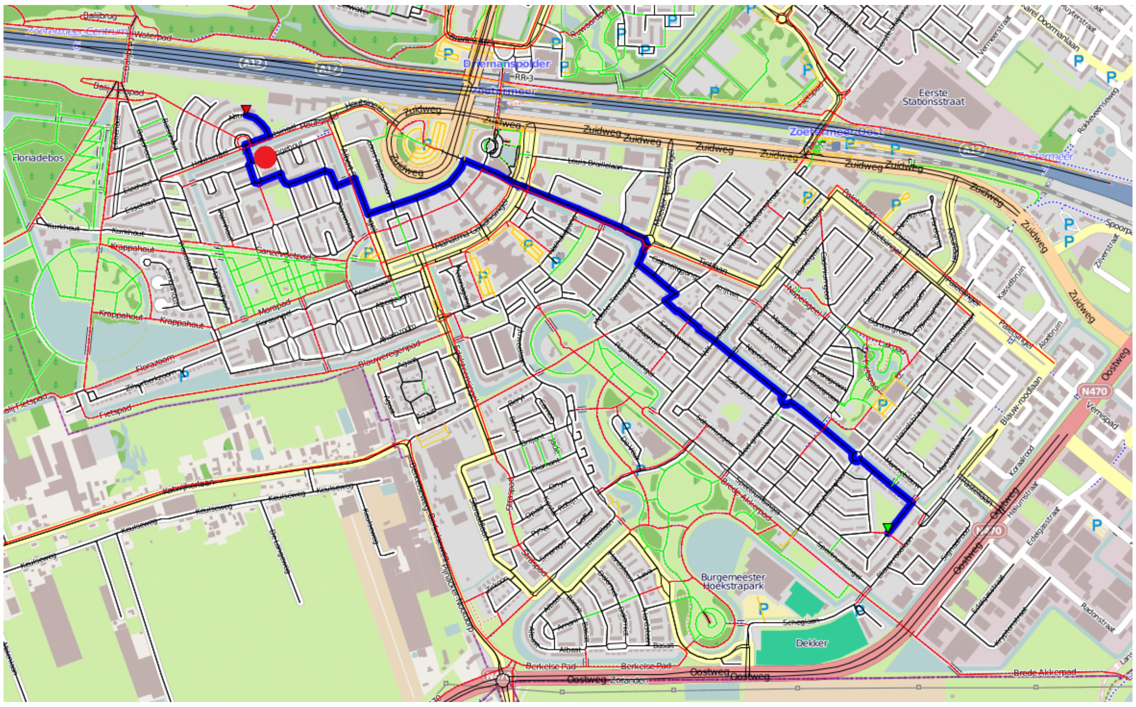


Fig. 8.11: Resulting route when adding other time dependent information

Last type of time dependent information we add to the problem includes all other situation that can also cause delay. An example is a road blockage due to road constructions. Suppose on a day a road is closed from 0.00 hr till 06.00 hrs for construction work. This road again is indicated with a red dot in Figure 8.11. Consider this road blockage and the fact that we want to depart at 02.00 hrs. The shortest route for this situation is given in Figure 8.11, with a total travel time of 3:46.35. As can be seen a small adjustment has been made to the route in order to avoid the obstruction.

In the next section we will the discuss the results of 1000 random selected routes. All time dependent information cause similar effects, namely delay on one, or several arcs. Further, also for all types of time dependent information the algorithms need to adjust in the same way. Therefore, we consider the influence of all time dependent information on the performance of the algorithms at once.

8.3 Overall results

In this section we present the results of five different algorithms, considering the incorporation of practical information. This is done for three different situations. The first situation is without any additional information, to see how the algorithms initially perform. The second situation concerns the road network properties, so without time dependency. In the third situation all practical information, as discussed in Chapters 5 and 6, are included.

In this experiment we test the algorithms on 1000 randomly selected routes within the network. If in a case it would be that a route is infeasible, we ignore this route and skip it for our results. It that way, the results are always based on the 1000 feasible routes. Infeasible routes can occur due to the network construction. The networks used in this research are selections from a larger network, and therefore it can be that some parts of the networks are not connected. We made sure that these 1000 routes are the same for every situation.

Since all algorithms find routes of the same cost, Table 8.1 shows the average travel time for every network in every situation. Notice that the average travel time increases when more information is included. This is due to the fact that firstly, restrictions cause shorter routes to be infeasible, and secondly because all time dependent information causes delays. The reason that the increase for Zoetermeer is larger than for the other networks is because of the concentration of the added information. For example, for events we added 100 random simulated events in all networks. This means that routes within Zoetermeer will more often be influenced by a event than routes within Noord-Holland, because the network is smaller.

	Zoetermeer	**Amsterdam**	**Noord-Holland**
Original situation			
Average travel time (ms)	208,185	612,826	1,271,539
Road network properties			
Average travel time (ms)	300,042	793,659	1,455,369
Time dependent			
Average travel time (ms)	3,946,902	891,400	1,455,408

Tab. 8.1: The average routes distances for all tested situations

Table 8.2 displays the results of the first situations, so without any added information. For Dijkstra's algorithm and A* no preprocessing time is given, since they do not use a preprocessing step. For the other algorithms, the preprocessing computations are done once for all 1000 simulated routes. The average query time is the average time it takes the algorithm to compute the shortest path and average number of inspected nodes are the average number of nodes that are inspected by the algorithm until the destination is found. The maximum query time is also presented to see whether improving algorithm not only improve on average, but also in the hardest routes.

According to the results in Table 8.2, we can conclude several things. First of all the computation times increase when the network becomes larger, which is as expected. Further can we see that, as expected, CH and CHALT perform the best for all networks. However, there is no convincing difference between CH and CHALT. What should be noted is that the preprocessing is only done once, so this should be considered during the analysis of the results. What should be clear is the difference between ALT preprocessing and the preprocessing of either CH or CHALT.

For the second situation we included the road network properties as described in Chapter 5. We run all algorithms with the intention that motor vehicles will use the routes, so only highway and local routes are allowed (see Chapter 5). Table 8.3 presents

	Preprocessing Time (ms)	Average Query Time (ms)	Max Query Time (ms)	Average # Inspected Nodes
Zoetermeer				
Dijkstra	-	1.59	40.24	2,210
A*	-	2.06	23.89	1,715
ALT	105.12	0.29	6.40	229
CH	1,001.41	0.39	5.91	179
CHALT	1,023.09	0.49	5.55	141
Amsterdam				
Dijkstra	-	57.10	118.88	43,061
A*	-	49.27	150.46	28,607
ALT	1,851.79	4.18	35.11	2,669
CH	39,932.95	2.53	8.57	676
CHALT	50,387.08	2.86	8.68	416
Noord-Holland				
Dijkstra	-	285.63	628.26	171,589
A*	-	183.40	1,102.55	89,613
ALT	8,882.91	21.79	135.00	12,705
CH	553,310.64	8.37	27.88	1,314
CHALT	436,428.03	7.73	30.90	829

Tab. 8.2: Results for the original situation

the results for this situation. Comparing these results to the results in Table 8.2 we see the following. Overall we can conclude that Dijkstra's algorithm, A*, CH, and CHALT need less computation time and ALT need slightly more computation time. The reason why those four algorithms are faster compared with the original situation is simply because there are less possible paths, mainly caused by considering road types only for motor vehicles. Since, CH and CHALT need to take the restrictions, like road type, also into account during the preprocessing (also discussed in Chapter 5), the preprocessing times also decrease. The reason that for ALT the average query time increases, is caused by the estimation used within ALT. Recall that the preprocessing of ALT does not become dependent of the situation, therefore the preprocessing computation times remain almost the same. However, this causes that the estimations $h(u,t)$ are further from the actual costs $g(u,t)$. CH, or CHALT are the fastest algorithms for this case, although it should be kept in mind that those need separate preprocessing for every situation, and therefore still leaves room for improvement.

For the time dependent situation we also generated 1000 random start times for the routes. All data as described in Chapter 7 is included in the algorithms as discussed in Chapter 6. In Table 8.4 the results for the last situation, including all data, are given. Overall we can conclude that the computation times have increased. This is due to the fact that for every considered arc the travel time for that moment need to be calculated.

With respect to increasing computation times, this especially applies on CH and CHALT. As discussed in Chapter 6, CH and CHALT cause some trouble in combination with time dependent information. Note that the preprocessing time increased, although it is still based on the lower bound travel time as in the second situation. This is due to

	Preprocessing Time (ms)	Average Query Time (ms)	Max Query Time (ms)	Average # Inspected Nodes
Zoetermeer				
Dijkstra	-	1.86	16.01	2,152
A*	-	2.05	7.65	1,573
ALT	97.24	0.98	5.67	327
CH	512.16	0.40	0.72	105
CHALT	546.88	0.83	1.63	108
Amsterdam				
Dijkstra	-	37.65	56.92	30,438
A*	-	32.72	70.26	20,481
ALT	1,723.89	9.97	72.28	3,224
CH	29,012.20	1.36	6.38	342
CHALT	23,472.93	1.59	25.30	263
Noord-Holland				
Dijkstra	-	172.83	360.06	112,290
A*	-	109.58	374.42	58,178
ALT	8,774.97	30.91	228.72	9,633
CH	427,844.44	2.32	10.88	471
CHALT	452,796.03	2.63	19.50	359

Tab. 8.3: Results including road network properties

the fact we use an upper bound on the travel time, see section 6.2.4. To make the query faster, this upper bound is pre-computed for every arc. This is very time intensive, since all information needs to be evaluated for every arc.

Additionally, also the average query time of CH and CHALT increased more compared to the other algorithms. This is due to two adaptations needed to cope with time dependency. First of all, the backward part of the algorithm needs to be executed twice, and secondly we allow the algorithm several times to ignore the upward searching, again see section 6.2.4.

Combining these facts, we see that CH and CHALT are not the fastest choice for time dependency. Notice however, that CHALT outperforms CH. It might therefore be interesting to do further research on improving CHALT for time dependency.

Finally should be mentioned that the results for CH and CHALT in Table 8.4, in terms of average travel time (see Table 8.1), are not always equal to the solutions of Dijkstra's algorithm. On average thirty routes (depending on the network) differ regarding to the total travel time. This is due to the data structure, currently a part of the data does not completely satisfy the FIFO property. As a consequence, bidirectional algorithms as CH, or CHALT, can find different routes. Therefore, we excluded these routes from the results.

On the other hand we see that ALT performs quite well. The computation times for ALT increase the least. Also compared to Dijkstra and A*, the average query time it is five times smaller for the smallest network and this difference becomes larger for larger network. However, the improvement for the maximum query time is a lot less compared to the query time.

	Preprocessing Time (ms)	Average Query Time (ms)	Max Query Time (ms)	Average # Inspected Nodes
Zoetermeer				
Dijkstra	-	5.93	40.27	2,165
A*	-	4.91	12.26	1,623
ALT	111.38	1.50	12.63	455
CH	6,116.90	43.38	52.02	2,989
CHALT	6,051.40	20.29	42.45	1,163
Amsterdam				
Dijkstra	-	127.99	197.72	30,435
A*	-	95.99	212.48	20,554
ALT	1,831.91	15.12	195.15	3467
CH	138,244.30	1,143.09	1,730.35	33,601
CHALT	140,339.77	527.50	1,410.67	8,443
Noord-Holland				
Dijkstra	-	512.92	998.30	112,277
A*	-	289.79	1,005.77	58,161
ALT	9,199.56	38.69	308.38	9,637
CH	964,913.74	4,504.44	8,452.15	119,970
CHALT	910,074.33	1,403.63	3,883.21	21,129

Tab. 8.4: Results for time dependency

9. CONCLUSION AND RECOMMENDATIONS

In this research we looked at which algorithm would be the best one to use for planning a route, within reasonable time. In this chapter we will discuss our conclusions, issues that could have been done better, and ideas for further research.

9.1 Conclusion and Discussion

First of all we discussed five algorithms which we tested on combining them with practical information. The algorithms we proposed are: Dijkstra's algorithm, A*, ALT, CH, and CHALT. Furthermore, we introduced practical information we wanted to incorporate and we made a distinction between two kinds of information: road network properties, which are not time dependent, and time dependent information. For both information types we proposed changes to the algorithm, such that the information could be incorporated in every algorithm. We presented the data used for this research and finally we showed the results.

The research question we investigated, was:

Research Question. *What is the best algorithm to plan a route, within reasonable time, considering the five different kind of practical information?*

From our research we can conclude that non-time dependent information can be handled without any problems for Dijkstra's algorithm, A*, and ALT, where for CH and CHALT some concessions need to be made. The preprocessing has to become dependent of the situation, in order to ensure correct hierarchies. However, the query computation times are still much better for CH and CHALT. Therefore, if there are enough requests based on one preprocessing, it is still profitable to use either CH, or CHALT. If there is only a small amount of requests per situation, ALT would perhaps be a better choice.

We can also conclude that for the time dependent information CH is the worst choice. CHALT performs better, but takes still more computation time than Dijkstra's algorithm. The hierarchical approach is not very accessible to combine with time dependency. Either the preprocessing needs to become time dependent, or large adjustments need to be made to the query, causing longer calculation times. A lot of research can be done in order to improve this, for which we give some ideas in the next section. For the other algorithms incorporating the information is no problem. As a result, based on the results in this research it can be concluded that ALT is the best algorithm in terms of computation time.

We have suggestions to improve the results given in this research. First of all, all results of this research are achieved by code programmed within Java. The results can be improved if this code would be used in a language as, for instance C++. The results can be even more improved by optimizing the code with respect to memory storage.

Another improvement that can be made concerns ALT and CHALT. In this research we use random selection as landmark selection method, but this could be improved by using a more advanced method of the landmark selection. Using a better selection method can increase the efficiency of the landmarks, for some situations, with fifty percent (Goldberg and Harrelson [13]).

Also some improvements could be made with respect to the data. The data used for this research possibly violates the FIFO assumption as defined in Chapter 6. We ignored this fact throughout this research, but this could be improved by making sure that the increase or decrease in travel time, stays proportional to the time frame in with the change occurs. This can be done by filtering the data to this end.

9.2 Recommendations for Further Research

Some additional research can be done for the weather data. It could be very interesting to also include temperature effects, since glazed frost can cause a lot of traffic problems. This occurs at a low temperature but not necessarily with a high precipitation level, so the effects as used in this research would change depending on the temperature.

Furthermore, in this research not all existing algorithms are evaluated, it could be interesting to see if, for instance Transit Node Routing could be successfully combined with practical information. Further are there many other combinations of goal directed search and hierarchical routing proposed in the literature, and for those it would also be interesting to see how they perform. One example is to again combine ALT and CH, but now the CH is only used for the preprocessing of ALT. This could be profitable in the case of table construction.

Another idea for further research is to make the graph less extensive. If there is a long road without any crossroad, it is not necessary to divide this road into many arcs, which is the case in this research. So, a single arc can be constructed for this road, making the graph less complex and still correct. It should however be considered, in combination with time dependent information, that there should be a maximum on the distance of such an arc. At the staring point of an arc it is known, for a moment in time, what the arc costs will be. So, if such an arc is very long this cost is not reliable, considering that new information is available for, say, every minute. This means there is a tradeoff between the extension of the graph and the reliability of the arc costs.

The last recommendation for further research concerns CH or other hierarchical algorithms. Till now it is not possible to properly combine hierarchical routing algorithms with time dependent information, as discussed in this research. To take these algorithms to a new level, much research needs to be done in how multi-levels graph can handle time-dependency. An idea could be in to zoom in on the problem caused with shortcuts. The approach we used is that when the costs of a shortcut exceeds the lower bound travel time, we add all original arcs to the search space. Therefore it is possible that the problem, causing extra travel time, is at the end of a long shortcut. Therefore it could be beneficial not to add original arcs, but shortcuts of a lower level, reaching till the point were the problems occur.

APPENDIX

A. EXAMPLE CH: ROUTE *A* TO *G*

In this appendix we evaluate the example, used in Chapter 4, in the case of CH, for a route from node a to node g. Figure A.1 gives the graph that results from the preprocessing. Since the preprocessing is not depending on the route, this graph is the same at the one used in section 4.4.2.

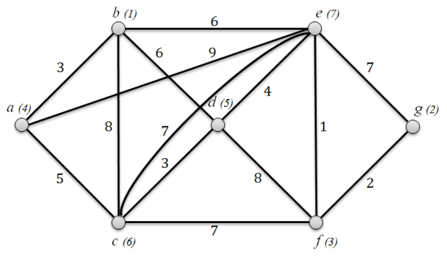
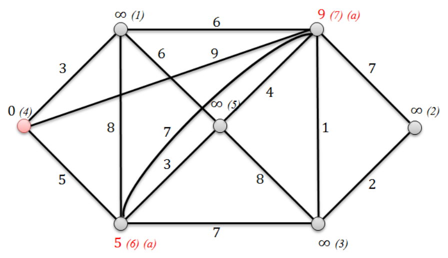
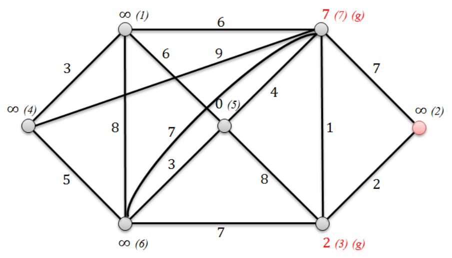


Fig. A.1: Resulting graph after preprocess

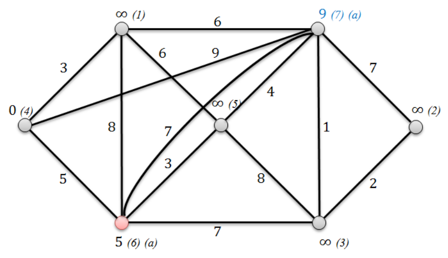
For CH the upward forward and downward backward search are interleaved. In Figure A.2 the routing process of CH is illustrated. Notice, that the backward search is identical to the backward search in section 4.4.2. This is due to the fact that the target node is the same in both situations. The forward search differs, since it start in node a instead of d. In Figure A.2(a) the algorithm find node e by using a shortcut and in Figure A.2(c)



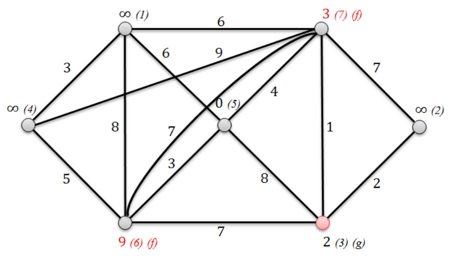
(a) Forward: a is current node



(b) Backward: g is current node



(c) Forward: c is current node

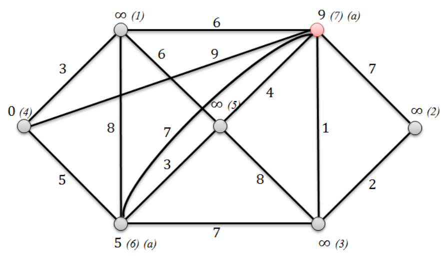


(d) Backward: f is current node

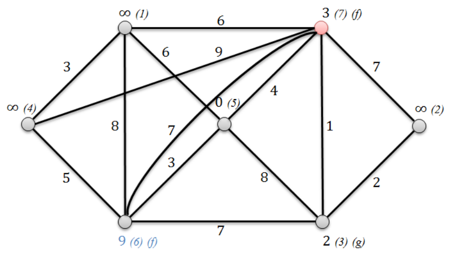
Fig. A.2: Example CH Algorithm

node e is found again (again using a shortcut), however that path has higher costs and therefore nothing changes.

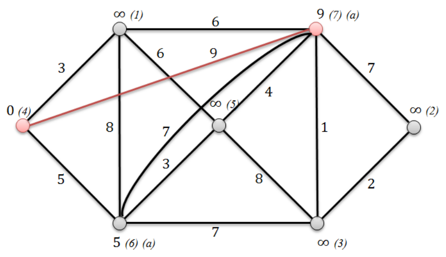
In Figures A.2(g) and A.2(h) the shortest paths of both searches are presented and in Figure A.2(i) both are combined. Note that a shortcuts is part this path. Since we need the shortest path in terms of the original graph, we need to retrieve the actual, underlying path of that shortcut. In this case this is quite simple. We know this shortcut was constructed it replaced arc (a, b) and (b, e) (recall from section 4.4.1), therefore we know the actual path, which is given in Figure A.2(j).



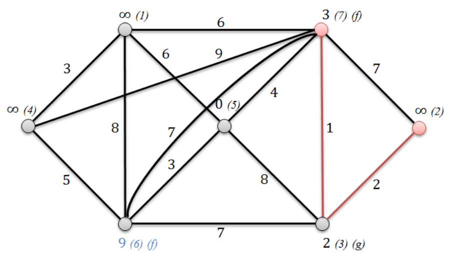
(e) Forward: e is current node



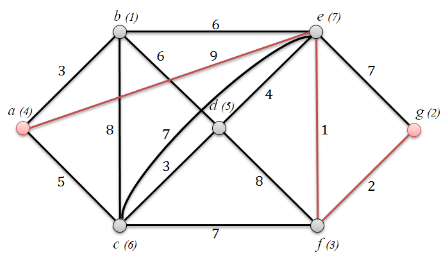
(f) Backward: e is current node



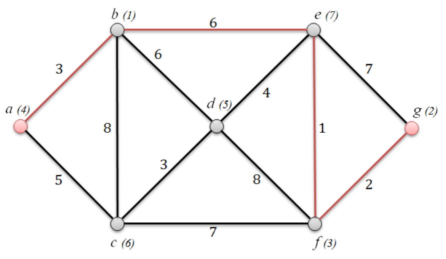
(g) Forward: resulting path



(h) Backward: resulting path



(i) Combined searches shortest path



(j) Combined path in the original graph

Fig. A.2: Example CH Algorithm (cont.)

BIBLIOGRAPHY

[1] I. Abraham, A. V. Goldberg, and R. F. Werneck. A hub-based labeling algorithm for shortest paths on road networks. *Lecture Notes in Computer Science*, 6630:230–241, 2010.

[2] H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast routing in road networks with transit nodes. *Science*, 316(5824):566, 2007.

[3] G. V. Batz, D. Delling, P. Sanders, and C. Vetter. Time-dependent contraction hierarchies. *SIAM*, pages 97–105, 2008.

[4] G. V. Batz, D. Delling, P. Sanders, and C. Vetter. Time-dependent contraction hierarchies and approximation. *Lecture Notes in Computer Science*, 6049:166–177, 2010.

[5] R. Bauer and D. Delling. Sharc: Fast and robust unidirectional routing. *Journal of Experimental Algorithmics*, 14(2):2.4, 2008.

[6] R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner. Combining hierarchical and goal-directed speed-up techniques for dijkstra âĂŹ s algorithm. *Journal of Experimental Algorithmics*, 15(2.3), 2010.

[7] D. Delling and G. Nannicini. Core routing on dynamic time-dependent road networks. *INFORMS Journal on Computing*, 25(2):187–201, 2012.

[8] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering route planning algorithms. *Algorithmics of large and complex networks*, 2:117–139, 2009.

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

[10] Y. Y. Fan, R. E. Kalaba, and J. E. Moore. Shortest paths in stochastic networks with correlated link costs. *Computers & Mathematics with Applications*, 49(9-10): 1549–1564, 2005.

[11] R. Geisberger and C. Vetter. Efficient routing in road networks with turn costs. *Lecture Notes in Computer Science*, 6630:100–111, 2011.

[12] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. *Lecture Notes in Computer Science*, 5038:319–333, 2008.

[13] A.V. Goldberg and C. Harrelson. Computing the shortest path: A * search meets graph theory. In *16th annual ACM-SIAM Symposium on Discrete Algorithms (SODA '05)*, pages 156–165. Society for Industrial and Applied Mathematics, 2005.

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

[15] M. Hilger, E. Kohler, R.H. Mohring, and H. Schilling. Fast point-to-point shortest path computations with arc-flags. *9th DIMACS Implementation Challenge*, (1126), 2006.

[16] M. Holzer, F. Schulz, D. Wagner, and T. Willhalm. Combining speed-up techniques for shortest-path computations. *Journal of Experimental Algorithmics*, 10(2):1–18, 2005.

[17] D.E. Kaufman and R.L. Smith. Fastest paths in timedependent networks for intelligent vehicle-highway systems application. *J Intell Transport Syst*, (1):1–11, 1993.

[18] R. F. Kirby and R. B. Potts. The minimum route problem for networks with turn penalties and prohibitions. *Transportation Research*, 3:397–408, 1969.

[19] U. Lauther. An extremely fast, exact algorithm for finding shortest paths in static net- works with geographical background. *In Geoinformation und Mobilitat - von der Forschung zur praktischen Anwendung*, 22:219–230, 2004.

[20] R. H. Mohring, H. Schilling, B. Schutz, D. Wagner, and T. Willhalm. Partitioning graphs to speedup dijkstra's algorithm. *Journal of Experimental Algorithmics*, 11: 2.8, 2006.

[21] D. Nannicini and L. Liberti. Shortest paths on dynamic graphs. *Int Trans Oper Res*, (15):551–563, 2008.

[22] G. Nannicini, D. Delling, D. Schultes, and L. Liberti. Bidirectional a* search on time-dependent road networks. *NETWORKS*, 59(2):240–251, 2012.

[23] I. Pohl. Bi-directional search. In *Machine Intelligence*, pages 127–140. Elsevier, 1971.

[24] G.H. Polychronopoulos and J. N. Tsitsiklis. Stochastic shortest path problems with recourse. *Networks*, 27:133–143, 1996.

[25] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. *Lecture Notes in Computer Science*, 3669:568–579, 2005.

[26] P. Sanders and D. Schultes. Engineering highway hierarchies. *Lecture Notes in Computer Science*, 4168:804–816, 2006.

[27] D. Schultes and P. Sanders. Dynamic highway-node routing. *Lecture Notes in Computer Science*, 4525:66–79, 2007.

[28] L. A. Wolsey. *Integer Programming.* Wiley Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, 1998.