Some results on heuristical algorithms for shortest path problems in large road networks

Inaugural-Dissertation
zur Erlangung des Doktorgrades
der Mathematisch-Naturwissenschaftlichen Fakultät
der Universität zu Köln

vorgelegt von
Stephan Hasselberg
aus Jülich

Köln 2000

Berichterstatter: Prof. Dr. R. Schrader
Prof. Dr. E. Speckenmeyer

Tag der mündlichen Prüfung: 24. Mai 2000

Contents

Abstract

This thesis studies the shortest path problem in large road networks. The classical algorithm for networks with non-negative edge weights is due to Dijkstra and has a worst-case performance of $\mathcal{O}(|E| + |V| \log |V|)$ using a simple priority queue as data structure for temporarily labeled nodes. We present a new, so-called tree heuristic, which is based on the similarity of shortest path trees and which can be used to speed up the shortest path search especially in practical applications like microscopic simulation of traffic or route guidance systems. Instead of searching a path in the original network, the tree heuristic partitions the network into classes of about equal size and constructs a special searchgraph for each class. On a test road network of about one million nodes the tree heuristic outperforms Dijkstra's algorithm by a factor of more than three with respect to runtime and about seven with respect to permanently labeled nodes where the found paths can be expected to have a relative error below 1%, if the starting and end node are not too close to each other. We also analyze the A^*-algorithm with overdo-factor, originally devised for Euclidean networks and derive an interval $[1.27, \ldots, 5]$ from which an optimal overdo-factor should be chosen in practical applications. Finally we give an algorithm which calculates edge tolerances for a shortest path and which can be used to generate reasonable altervative routes to the exact shortest path.

Chapter
One

Introduction

The problem of finding a shortest path between two nodes in a weighted graph is one of the classical problems in network optimization and has been extensively studied for more than forty years. In many applications the determination of a shortest path arises either as a stand-alone problem or as a subproblem in a more complex problem setting. Examples of the first are transportation problems, project management and DNA sequencing, examples of the latter are the approximation of functions and the knapsack problem.

In graph-theoretical notation the shortest path problem can be stated as follows: Let $G = (V, E, c)$ be a weighted graph where V is the node set, E the edge set and c is a given cost function on the set of edges. The problem is to find a path $P = (s = v_0, v_1, \ldots, v_{k-1}, v_k = t)$ between a given starting node s and a target node t with minimal costs with respect to c, where the cost of the path is the sum of the weights of its constituent edges.

Algorithms for solving the shortest path problem iteratively assign tentative distance labels to nodes at each step, which are upper bounds on the shortest path distance from the starting node to these nodes. While so-called label-setting algorithms designate one label as permanent and thus optimal in each step, in label-correcting algorithms all labels are temporary up to the last iteration when all labels become permanent. Label-setting algorithms show a better worst-case performance, but they are applicable only to acyclic networks or problems with nonnegative edge weights. In contrast, label-correcting algorithms can be applied even if negative edge weights are allowed and offer more algorithmic flexibility.

The classical label-setting algorithm is due to Dijkstra [20] dating back to 1959. The algorithm chooses the node v with the minimum temporary label as next node to be permanently labeled. The temporary distance of a neighbour w of v is updated if the path from s to w traversing over v has less weight than the present label of w. By using a simple priority queue as data structure for the temporarily labeled nodes the node selection and distance update operations lead to a worst-case complexity of $\mathcal{O}(|E| + |V| \log |V|)$. Numerous implementations of Dijkstra's

algorithm have been proposed that either improve its running time in practice or its worst-case complexity by using more complicated data structures (see e.g. [3]). Recently, Thorup [97] presented a linear algorithm for undirected graphs which avoids the inherent sorting of nodes in Dijkstra's algorithm by identifying nodes that can be permanently labeled in any order.

Although there are very fast implementations of Dijkstra's algorithm, in practical applications the special structure of the network often allows to devise even faster algorithms which will not necessarily give the optimal path, but generate routes[1] that are satisfactory for the specific purpose. One such application arises from the growing interest in telematics in the past years. Advanced Traffic Management/Information Systems (ATMS/ATIS) analyze the current traffic situation and give routing advice to travelers. This can be done via variable message signs or individually using some kind of onboard navigation system in the car.

Since the market introduction of onboard navigation systems in 1994 the sales figures provided by the suppliers almost doubled every year, reaching an estimate of 600000 sold systems in Germany in the year 2000 [37]. At the start, these navigation systems were nothing more than a digitalized road map on CD-ROM, which allowed to calculate a shortest path based on some static information like geometrical length, average link travel time or preference of specific road types. In a next step up-to-date traffic information could be received via mobile phone indicating a possible recalculation of the route to the user of such a system. Newer systems are capable of reacting to dynamic traffic information received via radio over the traffic message channel (TMC) or mobile phone and GPS-system, generating an alternative route by itself. Adding a wireless communication interface, i.e. on basis of the GPRS-standard [25], to onboard systems in the future will give a great increase in functionality and new fields of applications (cf. [89] for an overview over currently available systems).

Next to the route generation a variety of problems have to be dealt with in order to make route guidance systems an important tool for future traffic control systems. Among these are: collecting state information[2], communication, acceptance and fairness (see [88] for a further discussion of these issues).

Another instrument of increasing significance for a more efficient control and guidance of traffic are simulation models. These models are capable of analyzing the state of traffic and evaluate new strategies to meet the current and future envi-

[1] With the term route generation we mean the calculation of a path between s and t, which can be used as an approximation of the optimal path.

[2] Much (constructional) effort has been invested in the past years to make relevant traffic data available. Solar-driven sensors measuring traffic flow, density and significant speed changes have been installed at around 2000 widely used highway sections [53]. Other sources of information in Germany are the ADAC, community message posts, traffic guidance centers and floating-car-data-systems.

ronmental and social burdens of heavy traffic[3]. Microscopic, i.e. vehicle oriented, simulation of traffic as in [36, 65, 77, 78, 88] allows to compute traffic flows, travel times and emissions in large road networks. In doing so, these models solve the dynamic traffic assignment problem (cf. [87]) iteratively, which theoretically requires the update of the route of each driver in each step. Already in a test study of the rather small city network of Wuppertal with about 9000 nodes and 17000 edges this means the calculation of a shortest path for about 500000 origin-destination pairs in each iteration in the worst case [36], which takes several hours on state of the art computer workstations.

In both of these applications of the shortest path problem the optimal path is not necessarily needed. For the individual online-routing a user will not know the actual shortest path, but rather the given routing recommendation has to match the individual expectation of the user to obtain a high acceptance of these systems. In the traffic simulation the sum of all suggested paths must lead to a realistic picture of the current traffic state, which is not affected by small deviations of individual drivers from the exact path. On the other hand, faster algorithms to generate paths of sufficient quality can significantly reduce the computational effort of traffic microsimulations due to the number of path calculations. For route guidance systems a speedup will give these instruments more flexibility to meet online requirements.

Therefore, heuristical approaches for the shortest path problem are often employed in practice on networks that can be very large, consisting of several hundred thousand nodes or more[4]. These algorithms incorporate the special structure of road networks, in which edges are directed, edge lengths are close to the Euclidean distance between the two endpoints of an edge, a hierarchical structure is given through different types of edges and the graph is almost planar.

In this thesis we cover several aspects of route generation methods in large road networks. First we compare the runtime of various implementations of Dijkstra's algorithm and some label-correcting algorithms on large road networks using the test environments for shortest path algorithms of Cherkassky et al. [12] and Goldberg et al. [43]. On the largest graphs the best implementations show running times of a few seconds on a large multi-processor machine.

Then for an even faster algorithm we propose a new, so-called tree heuristic, which is based on the similarity of shortest path trees. The tree heuristic outperforms Dijkstra's algorithm on the largest networks by a factor of up to eight with respect to runtime and more than twenty with respect to permanently labeled nodes. The paths found by the tree heuristic can be expected to have a relative error below

[3]For example, the average daily traffic intensity on German highways and major roads rose by more than 40% between 1987 and 1998 (cf. [8]), causing a car driver to spend 65 hours per year in traffic jams [98].

[4]The largest network considered in this thesis is the state network of California with more than 1.5 million nodes and almost four million edges.

1%, if the nodes s and t are not too close to each other.

Instead of searching a path in the original network, the tree heuristic partitions the network into classes of about equal size and constructs a special searchgraph for each class that consists of all nodes of the original graph but only approximately as many edges as nodes. More precisely, for each class the searchgraph contains all edges of this class and the union of some shortest path trees in the original network of a small number of so-called base-nodes in each class. That is, the searchgraph is locally dense and globally sparse. Applying a reverse Dijkstra algorithm starting with node t on the searchgraph gives a very fast algorithm due to the tree-like structure of the searchgraph outside each class. Additionally, in contrast to other route generation algorithms the tree-like structure makes the heuristic almost invariant to the path length. The partitioning of the graph makes the tree heuristic also very well applicable for traffic simulations in parallel.

For an application of the tree heuristic the partitioning problem and the searchgraph generation have to be addressed. We compare two new partitioning methods that use characteristics of shortest path trees with a k-way partitioning algorithm due to Karypis and Kumar [62]. The results show that the connectivity of the partition classes very much affects the quality of the solutions of the tree heuristic. Since our partitioning methods almost always generate connected classes, they prove to be very well applicable for this purpose. For the searchgraph generation we test several methods for a suitable selection of base-nodes.

Other algorithms especially designed for the route generation problem in road networks are the A^*-algorithm and the HISPA heuristic. The latter searches for shortest paths to nodes of the highest hierarchy level in a circle with given radius r around s and t. These paths are added as appropriate edges to the highest hierarchy level and a shortest path between s and t is calculated only on this level, which is normally very sparse compared to the whole graph. The drawback of this heuristic is the determination of the optimal radius r.

For Euclidean networks Sedgewick et al. [92] proposed the A^*-algorithm for the shortest path problem that can also be applied to road networks. This algorithm directs the search in Dijkstra's algorithm towards t by using so-called 'future costs' based on geometrical information. This will narrow the area of permanently labeled nodes to a more elliptic shape, but imposes some additional computational effort through the calculation of the future costs. The tree heuristic shows a better runtime performance than both of these algorithms and in almost all cases gives better routing recommendations than the HISPA heuristic.

Recently, the A^*-algorithm was modified by multiplying the future costs with an overdo factor [56]. This gives faster running times but turns the exact algorithm into a heuristic. We analyze the modified A^*-algorithm for different overdo factors on four networks of different size and geometry. As it turns out, high overdo factors can lead to a suggested path which is the sequence of nearest neighbours and

result in relative errors of more than 100% for most of the paths. Under some assumptions which usually hold in road networks we derive a bound for the overdo factor, where a factor greater than this bound will always lead to the above described algorithmic behaviour. Together with a lower bound for the overdo factor in gridgraphs this gives an interval, from which the overdo factor should be chosen in practice.

In practical applications, where routing recommendations are given, information about the robustness of shortest path solutions can be a useful control parameter. Under the premise of one changing edge weight, we give an algorithm, which calculates edge tolerances for a shortest path. These tolerances can then be used to generate alternative routes to the exact shortest path, which allow a better fine-tuning for specific demands on such paths than an application of a k-shortest path algorithm as e.g. Yen's algorithm [101].

This thesis is organized as follows: In the next chapter we describe the shortest path problem in road networks in more detail and shortly review the classical shortest path algorithms along with an analysis of their empirical performance on large road networks. In chapter 3 we present our tree heuristic and analyze the runtime performance and solution quality of paths in great detail. The A^*-algorithm with overdo is studied in chapter 4 on four road networks of different size and geometry and bounds for an optimal value for the overdo factor are derived. Chapter 5 gives a rather short discussion of the robustness of shortest path solutions with an application to the k-shortest path problem. The main results of this thesis are summarized in chapter 6.

Chapter
Two

Classical Algorithms for the Shortest Path Problem

2.1 Introduction

The problem of finding a shortest path with respect to some cost function between two nodes in a network is a core model in network optimization and has been studied theoretically (for extensive references see the survey papers of Ahuja et al. [1, 2] and the bibliography compiled by Deo and Pang [18]) and empirically (e.g. [11, 21, 33, 38, 42, 54, 102]) by various researchers for about forty years. There are numerous applications of the shortest path problem either as standalone problem or as subproblem in more complex problem settings (see [3] for a variety of examples and further literature on this). Although the problem itself is relatively easy to solve, the design and analysis of the most efficient algorithms for solving it show some considerable ingenuity [3].

2.2 Shortest paths in road networks

We represent a road network by a directed graph[1] $G = (V, E)$ with node set V of cardinality n and edge set E with cardinality m. Since we do not consider undirected graphs in this thesis we use the terms edges, arcs and links interchangeably and always mean a directed edge unless otherwise stated.

For each edge $e = (u, v)$ we assume an edge weight $c(e)$ resp. $c_{u,v}$ representing the cost for the edge which will be travel time if not otherwise noted. Let $C = \max\{c_e : e \in E\}$ be the maximal weight in the network. Additionally, for each edge in a road network a type is given, which describes the importance of the road

[1] In most cases it is more convenient to use the line-graph representation of the actual road system as the network in question, since in this representation turning restrictions can easily be included.

in the network. Normally the types are numbered starting from zero, where this is the lowest type. The nodes are given with their coordinates with respect to some coordinate-system like for example the Gauss-Krüger [52] system. The length of a directed path $P(s,t)$ between nodes s and t is the sum of the weights of edges in the path. A shortest path $SP(s,t)$ is a path $P(s,t)$ of minimal length. For further graph-theoretical notation not mentioned here see [3].

The shortest path problem as mentioned in most of the cited literature is to determine for every nonsource node $v \in V$ a shortest path from s to v. In a routing application of the shortest path problem one has given a source node s and additionally a target node t and wants to determine only the shortest path $SP(s,t)$. Obviously, this is a subproblem of the former problem which most shortest path algorithms solve faster since they can be terminated as soon as the target node has been reached. To distinguish between these two problems we call the latter one the one-to-one shortest path problem and the former the one-to-all shortest path problem. In this thesis we are mainly concerned with the one-to-one shortest path problem, but since the two problems are so closely related the focus will not lie entirely on the one-to-one subproblem.

The problem of finding a shortest path between a source node s and a target node t can be viewed as sending one unit of flow as cheaply as possible (where the flow costs are given by the edge weights) from node s to node t in an uncapacitated network. Thus, we have the following linear programming formulation of the one-to-one shortest path problem:

$$\min z(\mathbf{x}) = \sum_{(u,v)\in E} c_{u,v}x_{u,v} \tag{2.1}$$

$$\sum_{\{v:(u,v)\in E\}} x_{u,v} - \sum_{\{v:(v,u)\in E\}} x_{v,u} = \begin{cases} 1 & \text{for } u = s \\ 0 & \text{for all } u \in V \setminus \{s,t\} \\ -1 & \text{for } u = t \end{cases}$$

$$x_{u,v} \geq 0 \text{ for all } (u,v) \in E$$

In most studies of the shortest path problem the following assumptions are imposed: The network is directed and contains a directed path from the source node s to every other node in the network. The edge weights are integers and the network does not contain a negative cycle (i.e. a directed cycle of negative length).

For the special case of road networks only the first assumption is not always fulfilled especially if the network in question is part of a larger road network. In this case there are always nodes at the border that cannot be reached from all nodes in the network. For our analysis of the shortest path problem this violation does not have any further meaning.

The integrality condition imposed on the edge weights is needed for the theoretical analysis of some algorithms. In practical applications irrational numbers have to be converted to rational numbers to represent them on a computer. By multiplying with a suitable large number rational weights can be transformed to integer weights. Therefore, this assumption is not a very restrictive one in practice. In road networks edge weights typically represent the time needed to traverse the edge, i.e. the travel time is the quotient of geometric length and speed of an edge. In order to get integer values for these numbers one can either use a sufficient small time unit like seconds and trunctuate or in a static setting with all possible speeds known in advance one can multiply with the least common multiplier of the speeds.

While the assumption of no negative cycles is trivially fulfilled in road networks with positive travel times as edge weights it is a very crucial one from a theoretical point of view. If negative cycles are allowed then the shortest path problem is an $\mathcal{NP}$ – *complete* problem and thus substantially harder to solve than the shortest path problem without negative cycles[2]. The reason for this is that a negative cycle might be traversed an infinite number of times since each repetition reduces the length of the directed walk[3]. To solve the problem therefore requires to prohibit walks that revisit nodes, which makes the problem theoretically harder.

We assume in this thesis that the weights of the edges are known and that a shortest path calculated with this cost function is indeed the optimal path, even in a dynamic scenario. For an application of the shortest path algorithms in a real-time setting with dynamically changing edge weights we have to make the additional assumption that there is no advantage from waiting at some node in order to save time on the shortest path. This assumption will be fulfilled if the gradient of the graph of weight against time is greater than minus one for each edge. With this assumption we can view the shortest path problem with dynamic edge weights as a static problem if we think of each edge weight as being the actual weight at the time we reach the starting node of the edge. In other words, we claim to have the total information about the network in time and freeze each weight at that time we reach the node on the shortest path from the source. Since there is no gain in waiting at some node the shortest path calculated with these edge weights will be the actual shortest path in time.

2.3 Tree of shortest paths

The one-to-all shortest path problem results in a collection of $n-1$ shortest paths which can be stored very efficiently in the so called shortest path tree. This is a directed out-tree rooted at the source s with the property that the unique path from s

[2] For the concept and terminology of $\mathcal{NP}$ – *completeness* see [35] or [83].

[3] That is, the linear programming formulation (2.1) has an unbounded solution.

to any node is a shortest path from s to that node. The existence of this tree follows from the observation that every subpath $P(s,w)$ of some shortest path $SP(s,t)$ is a shortest path between s and w. If this would not be true, i.e. the subpath $P(s,w)$ for some node w on the shortest path $SP(s,t)$ is longer than the shortest path $SP(s,w)$, then replacing $P(s,w)$ by $SP(s,w)$ yields a path that is shorter than $SP(s,t)$, contradicting its optimality.

If $dist(v)$ denotes the length of the shortest path $SP(s,v)$ then we have $dist(v) = dist(u) + c(e)$ for every edge $e = (u,v)$ on $SP(s,t)$. The reverse is also true: If $dist(v) = dist(u) + c(e)$ for every edge $e = (u,v)$ on a directed path $P(s,t)$ then P must be a shortest path from s to t. To see this, let $s = v_1 - v_2 - \cdots - v_k = t$ be the node sequence of path $P(s,t)$. Then using the fact that $dist(v_1) = 0$ we have

$$\begin{aligned} dist(t) &= (dist(v_k) - dist(v_{k-1})) + (dist(v_{k-1}) - dist(v_{k-2})) + \cdots + (dist(v_2) - dist(v_1)) \\ &= c(v_{k-1}, v_k) + c(v_{k-2}, v_{k-1}) + \cdots + c(v_1, v_2) = \sum_{e \in P} c_e. \end{aligned}$$

Thus, $P(s,t)$ is a directed path of length $dist(t)$ and we have established the following property.

Property 2.3.1 *Let dist represent a vector of shortest path distances. Then a path $P(s,t)$ is a shortest path if and only if $dist(v) = dist(u) + c(e)$ for every edge $(u,v) \in P(s,t)$.*

If we perform a breadth-first search of the network using the edges satisfying the equality $dist(v) = dist(u) + c_{u,v}$ then the breadth-first search tree contains a unique and by property 2.3.1 shortest path from the source s to every node that can be reached from s. This is the desired shortest path tree.

2.4 Label-Setting vs. Label-Correcting Algorithms

Algorithms for solving the shortest path problem are typically classified into two groups: label setting and label correcting algorithms. Both approaches iteratively assign tentative distance labels to nodes at each step which are estimates of (i.e. upper bounds on) the shortest path distance from the source node to these nodes. The core of almost all shortest path algorithms is the labeling method which keeps a distance label $dist(v)$ and a status $S(v) \in \{unreached, labeled, scanned\}$ for each node[4]. Starting with $dist(s) = 0$, $S(s) = labeled$, $dist(v) = \infty$, $S(v) = unreached$, $v \neq s$ the algorithm applies the SCAN operation of figure 2.1 to labeled nodes until none exists, causing the algorithm to stop.

[4]In order to actually output the shortest path we need also the predecessor $\pi(v)$ for each node v on the shortest path for backtracking.

```
procedure SCAN(v)
    begin
      foreach (v, w) ∈ E
      begin
        if dist(v) + c(e) < dist(w)
        begin
          dist(w) := dist(v) + c(e);
          S(w) := labeled;
        end
        S(v) := scanned;
      end

    end
```

Figure 2.1 SCAN procedure for shortest path algorithms.

While label-setting algorithms designate one label as permanent and thus optimal in each step, in label-correcting algorithms all labels are temporary up to the last iteration when all labels become permanent. Besides the different strategies the two types of algorithms are distinguished by the class of problems they solve. Since label-setting algorithms will never return to a node for which the label has been designated as permanent they are applicable only to (1) shortest paths problems defined on acyclic networks with arbitrary edge weights, and (2) to shortest path problems with nonnegative edge weights. In contrast label-correcting algorithms can be applied even if negative edge weights are allowed and offer more algorithmic flexibility. Therefore, label-setting algorithms have a much better worst-case complexity and also show better empirical performance [11]. Another advantage of label-setting algorithms is to stop the algorithms as soon as some target node t has been permanently labeled. Since for label-correcting algorithms all labels become permanent only in the last step, there is no chance for a premature interruption of the search.

The worst-case complexity of the labeling shortest path algorithms is determined by the total amount of work for choosing nodes to scan and updating the temporary distances in the course of the algorithm. We call these two operations node selection and label update and view the overall worst-case complexity of being $\mathcal{O}(nodeselection + labelupdate)$. Thus, for example in an acyclic network with arbitrary edge weights we can use a topological ordering of the nodes for node selection which takes time $\mathcal{O}(m)$. Updating distances for all outgoing edges

of scanned nodes takes also time $\mathcal{O}(m)$ giving an overall worst-case complexity of $\mathcal{O}(m)$. Since any algorithm for solving the problem must examine every edge in the network this bound is also best possible[5]. If cycles are allowed in the network additional work has to be performed for node selection since there is no such topological ordering of the nodes.

2.4.1 Dijkstra's algorithm

The main difference of label-setting to label-correcting algorithms is the operation node selection. Label-setting algorithms designate the label of a node v as being permanent as soon as the operation SCAN is applied to that node. Therefore, in order to work correctly the temporary label of a node for which SCAN is applied must be known to be the minimum possible label. To achieve this, the label-setting algorithms select the node with the minimum temporary label as next one to be scanned. This idea was first proposed by Dijkstra [20] and independently by Dantzig [16] and Whiting and Hillier [99]. A pseudo-code description of Dijkstra's algorithm for the one-to-one shortest path problem is shown in figure 2.2.

By leaving out the first if-statement and continuing the repeat-statement until all nodes have been scanned the code solves the one-to-all shortest path problem. If the edge weights are nonnegative then Dijkstra's algorithm will find the shortest path between two nodes s and t. To prove this, assume that the algorithm chooses a node v for the SCAN operation for which the temporary key $label(v)$ is greater than the length of the shortest path from s to v. Let $SP(s,v) = \{s = v_0, v_1, \ldots, v_k = v\}$ be the shortest path from s to v. Observe that every subpath $P(s,w)$ of the shortest path $SP(s,v)$ is a shortest path between s and w. If this would not be true, i.e. the subpath $P(s,w)$ for some node w on the shortest path $SP(s,v)$ is longer than the shortest path $SP(s,w)$, then replacing $P(s,w)$ by $SP(s,w)$ yields a path that is shorter than $SP(s,v)$, contradicting its optimality.

Let v_l be the node on $SP(s,v)$ which has as permanent label the correct length of $SP(s,v_l)$ and greatest index of all nodes on the path $SP(s,v)$ with this property. There is such a node v_l since node s is on the path and has permanent label zero. Since all edge weights are nonnegative the permanent label of v_l is smaller than $label(v)$ and thus v_l was already chosen for scanning. While scanning v_l the neighbour v_{l+1} of v_l on $SP(s,v)$ was assigned as temporary label the length of the shortest path from s to v_{l+1} since all subpaths $P(s,w)$ of $SP(s,v)$ are shortest paths from s to w. Again by assumption of nonnegative edge weigths $label(v_{l+1}) < label(v)$ and node v_{l+1} was already scanned. But then v_{l+1} is a node on $SP(s,v)$ with correct permanent label which has a greater index than v_l. Therefore $l = k-1$ and while scanning v_l node v was assigned the length of $SP(s,v)$.

[5] Therefore, the label update operation will always have time complexity at least $\mathcal{O}(m)$.

```
procedure DIJKSTRA
    begin
      foreach v ∈ V
      begin
        dist(v) := ∞;
        S(v) := unreached;
      end
      dist(s) := 0;
      S(s) := labeled;
      Repeat
      begin
        v = min_{u∈N} {dist(u) : S(u) = labeled};
        if u = t then break;
        else SCAN(v);
      end

    end
```

Figure 2.2 Dijkstra's algorithm.

Applying Dijkstra's algorithm to the one-to-one shortest path problem in networks with arbitrary edge weights might lead to incorrect search paths, if the algorithm is terminated as soon as the target node t is chosen for scanning and thus permanently labeled. For the one-to-all shortest path problem Dijkstra's algorithm in networks with arbitrary edge weights will find the correct shortest paths but might have an exponential number of scans.

There are numerous implementations of Dijkstra's algorithm which differ in the way the node with minimum temporary label is found. The original implementation of Dijkstra has time complexity $\mathcal{O}(n^2)$ for the overall node selection and therefore runs in $\mathcal{O}(n^2)$ time which is best possible for fully dense networks (i.e. those with $m = \Omega(n^2)$). Applying a simple binary search for finding the node with minimum temporary gives a total running time of $\mathcal{O}(n \log n + m)$. With more sophisticated data structures one can achieve better bounds for the node selection operation. In section 2.5.1 we describe some of these methods in more detail.

2.4.2 Label-correcting algorithms

Label-correcting algorithms use different strategies for node selection and continue until the shortest path optimality conditions of theorem 2.4.1 are satisfied for all edges in the network. The proof of the theorem uses very similar arguments to that of property 2.3.1 and can be found in [3].

Theorem 2.4.1 *For every node $v \in N$, let $d(v)$ denote the length of some directed path from the source node s to node v. Then the numbers $d(v)$ represent shortest path distances if and only if they satisfy the following shortest path optimality conditions:*

$$d(v) \leq d(u) + c_{uv} \quad \textit{for all } (u,v) \in E.$$

As already mentioned, label-correcting algorithms show worse empirical performance and worst-case complexity. Additionally, they always solve the one-to-all shortest path problem since all labels are temporary up to the last step. Nevertheless we include some implementations of label-correcting algorithms in our performance test on road networks in section 2.5.

2.4.3 Variants of Dijkstra's algorithm

A slight variant of Dijkstra's algorithm is the backward (or reverse) Dijkstra algorithm which will be used extensively in our tree heuristic described in chapter 3. Instead of determining the shortest path distances from a starting node s to all nodes in the network, the backward Dijkstra algorithm calculates the shortest paths from all nodes in the network to some (target) node t, which can be seen as the all-to-one shortest path problem. Therefore, the algorithm is also applicable to the one-to-one shortest path problem by halting as soon as node s has been scanned. The algorithm starts with $dist(t) = 0$, $S(t) = labeled$, $dist(v) = \infty$, $S(v) = unreached$, $v \neq t$ and examines in the SCAN operation all incoming edges (w, v) for each node v.

Other variants of Dijkstra's algorithm for the one-to-one shortest path problem are the bidirectional Dijkstra algorithm and the A^*-algorithm which can be applied especially to road networks. We describe them in more detail in the next two sections. Both algorithms make use of the two nodes s and t and do therefore not solve the one-to-all shortest path problem.

2.4.3.1 Bidirectional Dijkstra's algorithm

The bidirectional Dijkstra algorithm [71] solves the one-to-one shortest path problem by using both nodes of the desired path as starting nodes for a Dijkstra-like search proceeding in two phases. In phase 1 a forward Dijkstra starting from node

s and a backward Dijkstra starting from node t are performed simultaneously alternatively designating a node as permanent in each search. This gives two node sets S and T of permanently labeled nodes, i.e. the ones reached from s and the ones reached from t. As soon as both the forward and the backward algorithm have permanently labeled the same node w (i.e. $S \cap T = \{w\}$) the two algorithms are halted and phase 1 ends. The shortest path between s and t is then either the concatenation of the paths $P(s,w)$ with $P(w,t)$ or a path $P(s,u) \cup (u,v) \cup P(v,t)$ for some edge (u,v), $u \in S$ and $v \in T$. Phase 2 of the algorithm therefore finds the desired path by examining all cut edges between S and T. In practice the algorithm will permanently label fewer nodes than a simple forward or backward Dijkstra but does not have a better worst case complexity [3].

2.4.3.2 A*-algorithm

For Euclidean networks[6] it is possible to improve the average case performance of Dijkstra's algorithm for the one-to-one shortest path problem by making use of the inherent geometric information of these graphs. The idea is to direct the search of the shortest path from s to t into the direction of node t by choosing the next vertex x the algorithm scans according to the length of the shortest path from s to x plus the Euclidean distance between x and t. The basis idea is from Sedgewick and Vitter [92] and is originally attributed to Hart, Nilsson and Raphel [46].

More formally let $D(x,y)$ be the Euclidean distance between x and y and define $l(x,y)$ to be the length of the shortest path from x to y in the network where this length is the sum of the edge weights that constitute the path. If we assign each vertex that has not been scanned by the algorithm the value $\min_w\{l(s,w) + D(w,x)\} + D(x,t)$ and always choose as next vertex to be scanned the one which gives the minimum for this value, then the resulting algorithm will find the exact shortest path and will scan fewer nodes than Dijkstra's algorithm on typical graphs. The correctness follows since $D(x,t)$ is a lower bound for $l(x,t)$ and the speedup is accomplished because the shortest path tree grows in direction of t. In contrast to Dijkstra's algorithm the area of scanned nodes will be an ellipsoid.

For the case of road networks where the edge weights correspond to the travelling time on the link the original idea of Sedgewick and Vitter has to be modified. Instead of using $D(x,t)$ as lower bound for $l(x,t)$ we use $D(x,t)/v_{max}$ where v_{max} is the maximal travelling speed observed on an edge. The term $D(x,t)/v_{max}$ is often quoted as future costs and gives a lower bound on $l(x,t)$ in road networks. If the road network is Euclidean the A^*-algorithm will give the exact shortest path. Road networks will in general be not pure Euclidean for a number of reasons, which we

[6] A graph is called Euclidean network if the nodes correspond to points in $\mathbb{R}^d$ and the weight of an edge is equal to the Euclidean distance between its two endnodes. In these networks the triangle inequality is therefore fulfilled.

will discuss in more detail in chapter 4. As consequence the triangle inequality must not be fulfilled in road networks which might cause the A^*-algorithm to visit a vertex more than once or in extreme cases to find not the exact shortest path. This latter phenomenon was never observed in all our tests.

2.5 Performance of shortest path algorithms on road networks

In [11] Cherkassky et al. tested a variety of shortest path algorithms on several families of random graphs with a maximum size of 1048578 nodes and 4194305 edges thereby offering a test environment for shortest path algorithms. This study was extended in [42] by Goldberg et al. to some new implementations[7]. Zhan/Noon [102] tested the algorithms of the first study of Cherkassky et al. for the special class of road networks, their largest being a graph of the state of Georgia with about 93000 nodes and 265000 edges.

To evaluate the heuristic algorithms for the shortest path problem presented in this thesis we used an implementation of Dijkstra's algorithm on the basis of the **LEDA**[8] class library with either using a priority queue or a bounded priority queue (see [73, 74] for details). We compared these two implementations with those of both studies of Cherkassky et al. to check if they are of reasonable performance. Using the two large area networks of the states of Northrhine-Westphalia (NRW) and California with 457124/1050874 nodes/edges resp. 1580305/3934788 nodes/edges also gave the opportunity to test all of these algorithms on realistic road networks of larger scale than in the study of Zhan/Noon.

In the next section we will very shortly review the various shortest path algorithms of both studies and then show the results of our evaluation on the networks of NRW and California in section 2.5.2.

2.5.1 Implementations of shortest path algorithms

We will now give a very short description of the shortest path algorithms of our evaluation. The theoretical worst-case complexity for each of the implementations is shown in table 2.1 along with the performance results. The notation is taken from the studies of Cherkassky et al. For more details about the different implementations see [11], [13] or the original literature cited for each implementation. Note that C is the maximal edge weight in the network.

[7] The codes, generators, and generator inputs of both studies are available at http://www.intertrust.com/star-lab.com/goldberg/soft.html.

[8] **L**ibrary of **E**fficient **D**ata types and **A**lgorithms, developed at the Max-Planck Institut für Informatik, Saarbrücken. **LEDA** is available at http://www.mpi-sb.mpg.de/LEDA/.

BF: The Bellman-Ford-Moore label-correcting algorithm, due to Bellman [6], Ford [29] and Moore [76] where the set of labeled nodes are maintained in a FIFO queue.

BFP: Variant of the Bellman-Ford-Moore algorithm where a node is only scanned if its parent is not in the queue (parent checking).

DIKB: The implementation of Dijkstra's algorithm by Dial [19] using a bucket data structure with $C+1$ buckets where the nodes in each bucket are stored in FIFO order.

DIKBA: Approximate bucket Dijkstra implementation where bucket i contains nodes with labels in the range $[i\alpha,(i+1)\alpha-1]$ with a fixed parameter α and FIFO order. The code implemented uses $\alpha=\lceil C/2^{11}\rceil$.

DIKBM: Dial's algorithm with $B<C+1$ buckets. If at some stage i the B buckets contain labels in the range $[B_i,B_i+B-1]$ an overflow bucket keeps all those nodes with labels greater than B_i+B which are redistributed at the beginning of the next step. The code implemented uses $B=\min(50000,C/3)$.

DIKBD: A k-level bucket Dijkstra implementation for $k=2$. Each level has the same number of buckets but the range of the buckets increases for higher levels. In each level the range of the buckets is the same.

DIKF: The implementation of Dijkstra's algorithm using Fibonacci heaps for storing labeled nodes [31].

DIKH: The implementation of Dijkstra's algorithm using k-ary heaps for $k=3$ (see e.g. [15]).

DIKLB: Implementation of Dijkstra's algorithm with a k-level bucket data structure of Denardo and Fox [13, 17, 43] for $k=2$.

DIKHOT: Dijkstra's algorithm with a k-level hot queue combining a k-level bucket data structure with a heap on top [42, 43] for $k=2$.

DIKQ: The naive implementation of Dijkstra in [20].

DIKR: The implementation of Dijkstra's algorithm using one-level R-heaps [4].

LEDAPQ: Our implementation of Dijkstra's algorithm using the **LEDA** class library and a bucket structure as priority queue [74].

LEDABPQ: Our implementation of Dijkstra's algorithm using the **LEDA**

class library and a bounded bucket structure as priority queue [74].

GOR: Modification of the Bellman-Ford-Moore algorithm with generalized parent checking using a topological sort [41].

GOR1: Modification of the Bellman-Ford-Moore algorithm with generalized parent checking using a topological sort with distance-update [11, 41].

PAPE: Label-correcting algorithm of Pape-Levit [70, 84] with a high-priority set S_1 of scanned nodes and a low-priority set S_2 of labeled, but not scanned nodes. The algorithm maintains S_1 as LIFO stack and S_2 as FIFO queue using the *dequeue* data structure from [82].

TWO-Q: Algorithm proposed by Pallotino [82] where the two sets S_1 and S_2 from the Pape-Levit algorithm are maintained as FIFO queues.

THRESH: A method suggested by Glover et al. [39] using the two subsets NOW and NEXT for labeled nodes. These are maintained as FIFO queues. A threshold parameter controls when nodes are moved from NEXT to NOW.

Recently Thorup presented a linear $\mathcal{O}(m)$ algorithm for solving the shortest path problem in undirected graphs [97]. Although it is a label-setting algorithm it is not a variant of Dijkstra's algorithm, since otherwise the linear running time would imply that the algorithm is capable of sorting n numbers in linear time. Instead of choosing the node with minimum label to be scanned next, the algorithm avoids the sorting bottleneck by building a hierarchical bucketing structure, identifying node pairs that may be visited in any order.

2.5.2 Run-time and space performance

Table 2.1 gives the results of our evaluation of the shortest path implementations described in section 2.5.1 on the two road networks of Northrhine-Westphalia and California. All implementations of the shortest path problem software library by Cherkassky et al. [10, 12] solve the one-to-all shortest path problem thereby generating the shortest path tree for some starting node s. Therefore, the variants of Dijkstra's algorithm for the one-to-one shortest path problem described in section 2.4.3 are not included in this study. Their run time performance will be analyzed in comparison with the tree heuristic in section 3.5.3. We calculated the shortest path trees for the same 100 randomly chosen starting nodes for each implementation and averaged the running times over five independent runs. The experiments were

performed on a Sun Enterprise E450 with 1.15 GByte RAM and four UltraSparc-II CPU's with 400 MHz, running under Solaris 2.5. The code was compiled using the GNU compiler version 2.8.9 with O4 optimization flag.

The different codes are sorted according to their running time on the California network. The last column of the table gives the ratio of runtime in relation to the best code on the California network.

In the extensive study of Cherkassky et al. [11] the three Dijkstra codes DIKB, DIKBA and DIKBD performed very well on almost all problem families. The codes DIKR, DIKH and DIKF were noticeably worse than the former three and DIKBM performed bad on simple grid problems but well on others. The performance of the hotqueue implementations DIKLB and DIKHQ in [13] seems to be as good as that of the other fastest Dijkstra algorithms, although the problem families of the two studies were not the same. The performance of the incremental algorithms PAPE and TWO-Q and the threshold code THRESH was very good on simple grid problems, but poorly on some other problems. The Bellman-Ford-Moore algorithm showed good performance on networks with small shortest path tree depth and on networks with highly metric edge lengths.

The simple grid families from the study of Cherkassky et al. are most interesting for us being of similar structure than road networks. The results of Cherkassky et al. were confirmed in the study of Zhan/Noon [102], where TWO-Q, PAPE, THRESH and DIKBA performed best on the largest road networks.

The results in our tests differ from the one mentioned mainly in the poor performance of TWO-Q and PAPE on the California network, while the performance for NRW was comparable to the one of the other studies. In line with these results is the performance of the Dijkstra implementations DIKBA, DIKLB2, DIKHQ2, DIKB and THRESH which are the fastest in our tests. The DIKBD code was not as good in our tests as in the study of Cherkassky et al. and also DIKF showed a very poor performance. For the Bellman-Ford-Moore algorithms GOR shows a good performance especially if the worst-case complexity is taken into account. BF and BFP show a very poor performance on the California network being in line with the results from Zhan. The codes LEDABPQ and LEDAPQ show a reasonable performance, being fast enough for the analysis of the proposed shortest path heuristics in this thesis. As expected the usage of the LEDA software library increases the memory requirements of the code significantly by a factor of about four. For the other codes the differences in memory between the implementations are small.

2.6 Heuristical shortest path algorithms

There are a number of heuristical shortest path algorithms for the one-to-one shortest path problem especially for routing purposes in road networks. These algo-

Id	Algorithm	Complexity	Memory in Mb		Runtime in sec		Ratio
			NRW	California	NRW	California	
DIKBA	Dijkstra with app. buckets	$O(m + n(\alpha + C/\alpha))$	28.3	97.2	0.72	2.59	1.00
DIKLB2	Dijkstra with 2-level buckets	$O(m + nC)$	44.7	158.2	0.72	2.6	1.00
DIKHQ2	Dijkstra with 2-level-hot-queues	$O(m + nC)$	44.7	158.2	0.73	2.6	1.00
DIKB	Dijkstra with buckets	$O(m + nC)$	28.3	97.2	0.83	2.87	1.11
THRESH	Threshold	$O(nm)$	26.5	91.1	0.67	3.01	1.16
DIKBM	Dijkstra with overflowbucket	$O(m + n(C/B + B))$	28.3	97.3	0.82	3.28	1.27
DIKBD	Dijkstra with double buckets	$O(m + n(\beta + C/\beta))$	28.3	97.3	0.89	3.36	1.30
LEDABPQ	LEDA with bounded priority queue	$O(m + nC)$	113.0	389.0	0.93	3.77	1.45
GOR	Topological sort (T-S)	$O(nm)$	26.5	91.1	1.28	5.6	2.16
LEDAPQ	LEDA with priority queue	$O(m + nC)$	122.0	397.0	1.61	6.12	2.36
DIKR	Dijkstra with R-heap	$O(m + n\log(C))$	28.3	97.3	2.04	7.04	2.72
TWO-Q	Pallotino	$O(n^2m)$	26.5	91.1	0.79	7.24	2.8
DIKH	Dijkstra with k-ary heap	$O(m\log(n))$	24.8	84.9	2.21	7.9	3.05
DIKF	Dijkstra with Fibonacci heap	$O(m + n\log(n))$	33.8	115.9	5.08	17.64	6.81
PAPE	Pape-Levit	$O(n2^n)$	26.5	91.1	0.97	33.51	12.94
GOR1	T-S with distanceupdate	$O(nm)$	26.5	91.1	4.57	40.6	15.68
BFP	B-F with parent checking	$O(nm)$	26.5	91.1	3.99	282.68	109.14
DIKQ	Naive Dijkstra	$O(m + n^2)$	26.5	91.1	73.45	298.14	115.11
BF	Bellman-Ford (B-F) basis	$O(nm)$	26.5	91.1	5.53	651.57	251.57

Table 2.1 Runtime and space performance of various shortest path algorithms for the networks of NRW and California.

rithms try to narrow the search area for Dijkstra's algorithm and either use geometrical properties of the topology underlying road networks or some kind of hierarchy based on the different road types given in these networks. We mention two of them in this section which will be used in the following as benchmarks for our tree heuristic. If applied to the one-to-one shortest path problem the heuristics will calculate a path between two nodes which must not be the shortest path with respect to the weight function. Therefore, we call the path computed by any heuristic the suggested path between starting node s and target node t which might coincide with the actual shortest path between the two nodes.

2.6.1 The HISPA heuristic

The HISPA (**Hi**erarchical **S**hortest **Pa**th) heuristic makes use of the hierarchical structure of road networks [79, 93]. The different types of roads in a road network induce a discrete hierarchy in the graph. The object of this heuristic is to calculate the shortest path on the graph of the highest hierarchy level which is very sparse compared to the whole graph. Since normally the two nodes s and t will not belong to the highest hierarchy level the algorithm first calculates the shortest paths from s resp. t to all nodes v of the highest hierarchy which lie within the circle of a given radius r around s resp. t. If there is no such vertex in the circle with radius r the search is continued until a vertex is found. If t (s) lies within the circle with radius r around s (t) the algorithm stops. For each shortest path from s to such a v resp. from v to t an edge is added to the graph of the highest hierarchy together with the two nodes s and t. The length of these edges is given as the length of the associated shortest path. Finally, a shortest path from s to t is calculated on the modified graph of the highest hierarchy level.

The path found by this algorithm is not necessarily the shortest path for two reasons. The optimal switch-vertices of the highest level for s and t will not always lie within the circle with radius r and the highest hierarchy level must not be closed under shortest path calculations in case there is a shortcut using edges of a lower hierarchy level. Proposed methods to avoid these drawbacks of the algorithm turn out to be very time consuming because of the necessary preprocessing especially in a dynamic setting [95].

2.6.2 The A*-algorithm with overdo

The A^*-algorithm with overdo [92] is a variant of the A^*-algorithm presented in section 2.4.3.2. The basic idea is to direct the search more strongly in direction of the target node t by multiplying the future costs with a factor greater than one. As consequence the future costs are not necessarily a lower bound for the travel

time between some node v and t and the algorithm will not always find the exact shortest path. We analyze this algorithm in great detail in chapter 4.

Chapter
Three

A Heuristic Based on Trees

3.1 Introduction

In this chapter we propose a new heuristic for generating routes in road networks that has a very good runtime performance and gives solution paths of high quality. The results from our empirical test of implementations of Dijkstra's algorithm in section 2.5 show that the one-to-one shortest path problem can be solved in a few seconds even for very large networks on multi-processor machines with high-end processor performance. In applications where a very large number of routes have to be determined, even an application of a very sophisticated Dijkstra implementation might not be of sufficient speed and faster algorithms are desired.

One such application is the traffic microsimulation on the basis of a cellular automaton (cf. [65, 66, 67, 77, 78, 88]). In this day-to-day simulation of traffic flows an equilibrium is generated iteratively. In doing so, the dynamic traffic assignment problem (cf. [87]) is solved in each iteration, which theoretically requires the update of the route of each driver in each step. In a test study of the network of Wuppertal with about 9000 nodes and 17000 edges this leads in each iteration to the calculation of routing recommendations for about 500000 origin-destination pairs in the worst case [36], which takes several hours on state of the art computer workstations[1].

Another application are onboard route guidance systems, which become more and more popular. In those a driver asks for the shortest path from his current to some target location. Although these systems are currently still employed in an almost static setting, dynamic aspects will be more and more integrated. Rickert showed the positive effect of rerouting drivers to avoid grid-locks in a microsimulation of an online-routing system [88], which increases the computational requirements of such a system. For both applications the optimal path must not be

[1] Our traffic simulation bypasses this excessive calculation by rerouting in each iteration only part of the drivers on an individual route set.

found. It suffices that the suggested path is close to the optimal path, in order to make these systems of practical use.

The heuristic we propose is based on the intuition of a greater similarity of shortest path trees for nearby starting nodes in road networks. This intuition is based on the underlying geometry of those networks. Characteristic features of road networks are edge lengths near the Euclidean distance between the two end points of an edge, the hierarchical structure through the classification of the edges in different types and near-planarity. An example for the similarity of shortest paths in these networks are the paths of two different starting nodes s_1 and s_2, which are geometrically not too far apart, to a more distant node t. In a road network we expect the two paths from s_1 and s_2 to differ in the vicinity of the starting nodes but to be the same near t.

Therefore, our so called tree heuristic[2] partitions the network into a few clusters of approximately equal size and constructs a searchgraph on the whole nodeset for each cluster. This searchgraph is locally dense (i.e. contains all edges having one endpoint in the cluster) and globally sparse (i.e. contains only those edges outside the cluster which belong to the union of a few shortest path trees for specific base nodes in the cluster). In order to calculate a shortest path from a node s to a node t the search is performed on the searchgraph of that cluster s belongs to. The number of edges in the searchgraph is approximately the number of nodes in the network and therefore applications of Dijkstra's algorithm on this reduced graph will have faster running times than on the whole graph. The speedup can be significantly increased by applying the backward Dijkstra algorithm (see section 2.4.3) since the searchgraph has a tree-like structure outside of the cluster. The main idea of the tree heuristic is the similarity of shortest path trees for starting nodes that are in some sense close to each other. Therefore, we expect the tree heuristic to perform well for all graphs which satisfy this similarity property.

This chapter is organized as follows: In the next section we give a statistical motivation of the tree heuristic by analyzing the similarity of shortest path trees in road networks. After describing the tree heuristic formally in section 3.3 we give a detailed account of a practical realization in section 3.4. In section 3.5 we analyze the quality of the paths found by the tree heuristic and compare its runtime performance with that of other well-known shortest paths algorithms. Our main results are summarized in section 3.6.

[2] Since the similarity of shortest paths is expected for almost all target nodes t in the network we have similar shortest path trees for nearby starting nodes, giving the heuristic its name.

#SPT	NRW	Cologne_20x25		Cologne_10x10	
		inside	outside	inside	outside
0	37.435%	33.40%	53.51%	27.13%	54.74%
[1-10]	11.37%	12.99%	1.69%	15.08%	0.90%
[11-20]	5.10%	5.75%	0.84%	6.43%	0.36%
[21-30]	5.57%	5.24%	1.02%	7.48%	0.82%
[31-40]	4.96%	5.14%	0.66%	5.98%	0.35%
[41-49]	9.39%	10.53%	1.66%	11.49%	0.85%
50	26.18%	26.96%	40.63%	26.41%	41.16%

Table 3.1 Frequency of edges in shortest path trees.

3.2 A statistical justification of the tree heuristic

The intuition that shortest paths in road networks differ only locally is supported by a statistical comparison of shortest path trees we performed on the network of Northrhine-Westphalia (NRW) having 457124 nodes and 1040687 edges. The roads are classified into five different types, where type four are mostly highway edges. Fifty shortest path trees for randomly chosen starting nodes out of regions of different size were calculated and for each edge of the graph the number of trees was counted in which the edge appeared. The results are shown in table 3.1 for three regions: A square of length 10 kilometers in the center of Cologne, a 25kmx20km rectangle around Cologne and whole NRW. The first resp. last row shows the percentage of edges which appeared in zero resp. in all fifty shortest path trees. The other rows give the results for intervals of size 10 resp. 9 for the intermediate values. For the two areas around Cologne we distinguished between edges having at least one endpoint in the area (inside) and those lying completely outside (outside). The values for each area are given relative to the number of edges inside resp. outside the area.

The results in table 3.1 show that:

- The number of edges inside an area lying in no shortest path tree decreases if the area gets smaller (37% to 27%). On the other hand this number increases for edges outside an area (37% to 54%). This shows that it is unlikely for an edge not to be in some shortest path tree for starting nodes close to the edge.
- The number of edges which are in all shortest path trees increases for edges outside of the area if the area of the starting nodes is small (26% to 41%) showing that outside of a small area trees are likely to have more edges in common.

- If you take the shortest path tree of a randomly chosen starting node out of a small area more than 90% of the outside edges will have the same tree or non-tree status than for any other shortest path tree out of this area.
- Taking the numbers in the table relative to the total number of tree- resp. non-tree edges we can conclude that more than 90% of the edges of a shortest path tree T_s for some node s will be in every shortest path tree for starting nodes out of a small area around s. Of the non-tree edges of T_s more than 92% will be in no shortest path tree for starting nodes out of a small area.

The last observation is also supported by a different evaluation of similarities of shortest path trees in road networks. For the shortest path tree of a given starting node we calculated the percentage of tree and non-tree edges which are also found as tree resp. non-tree edges in a shortest path tree of a different starting node. In figure 3.1 the results are plotted against the distance between the two starting nodes. The results in the plot are averaged over ten randomly chosen starting nodes in each distance bin and the result in each bin is averaged over ten shortest path trees randomly chosen in the whole graph. The length of the distance bin was five kilometers.

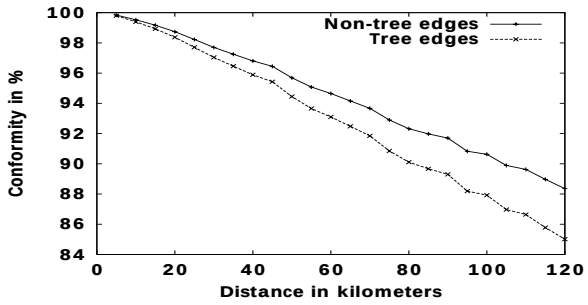


Figure 3.1 Conformity of Shortest Path Trees.

The plot shows that the shortest path trees for two starting nodes which are not more than 20 kilometers apart are expected to have more than 95% of the edges in common. This number decreases to about 90% for nodes being not more than 60 kilometers apart, but is still more than 80% if you take two arbitrary starting nodes. The percentages for the non-tree edges are even a little higher. The variances were below 5% for tree edges and below 3.5% for non-tree edges. The degree of conformity is naturally affected by the travelling speed of the various road types as-

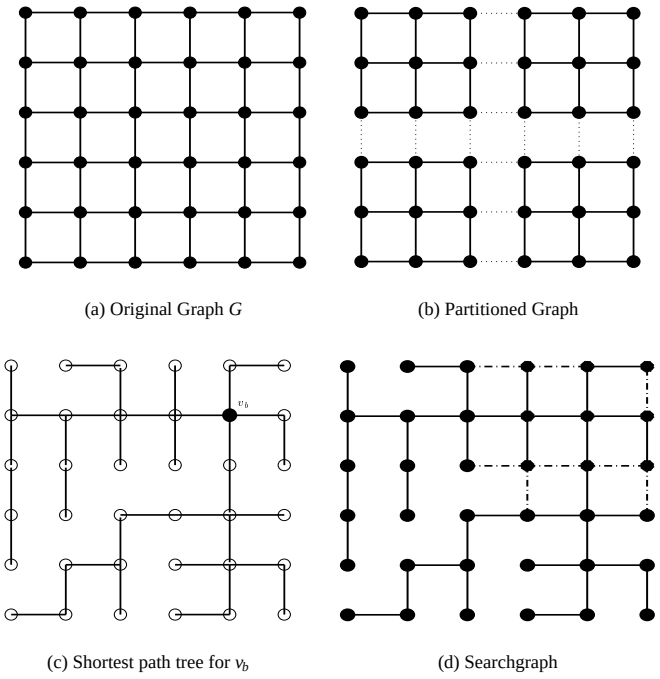
sociated with the links of the road network. The shown data was calculated for speeds of 40, 50, 60, 80 and 100 km/h for the five types of links. If these speeds are changed to 20, 40, 60, 80 and 120 km/h, thus giving different road types a greater speed difference, then the conformity of the shortest path trees is even higher. On the other hand making no difference between the various road types by associating the same speed on all of them results in a greater nonconformity for the shortest path trees. But even then one can expect that the shortest path trees of two arbitrary starting nodes in the network have more than 70% of tree- and non-tree edges in common. In summary the results show that shortest path trees in road networks tend to be very similar and that this conformity is the higher the closer the starting nodes of the trees are.

3.3 A formal description of the tree heuristic

The results from the previous section motivate the following tree heuristic for efficiently finding shortest paths in a road network $G = (V, E)$: The set of nodes V is partitioned into clusters of approximately equal size. For each cluster C a special search graph H_C with $V(H_C) = V(G)$ is constructed which is used for every shortest path calculation with starting node $s \in C$. The main difference between H_C and G is the set of edges of the graphs. In each cluster a few special nodes are chosen and the shortest path trees for these nodes are calculated in the original network. These nodes are called base-nodes for the cluster. The set of edges of H_C consists of the union of the edges of the shortest path trees of all base-nodes of C, all edges having at least one endnode in C and a few edges which are expected to belong to many shortest paths. For road networks those are typically the edges of the highest hierarchy level, the number of which is small compared to the total number of edges.

To describe the tree heuristic more formally let $V(G) = V_1 \cup \ldots \cup V_k$ be a partition of the nodes of G into k clusters of size approximately n/k. For each V_i choose a number of base-nodes $v_1^i, \ldots, v_{b_i}^i$, $b_i \geq 1$, and calculate their shortest path trees $T_1, \ldots, T_{b_i}$ in G. The number b_i of base-nodes must not be the same for all clusters. Let E_i^{in} be the set of edges having at least one node in V_i and E^* be the set of edges which are expected to be of high importance for shortest paths (i.e. the edges of the highest hierarchy level). Let $E_i = E_i^{in} \cup E^* \cup E(T_1) \cup \ldots \cup E(T_{b_i})$ and $H_i = (V(G), E_i)$. For each starting node $s \in V_i$ the shortest path to a node t is calculated in the graph H_i. We exemplify the above procedure in figure 3.2 on a small example network. In the example only one base-node v_b is chosen and the set of special edges E^* is empty.

In summary the tree heuristic consists of four phases:



(a) Original Graph G (b) Partitioned Graph

(c) Shortest path tree for v_b (d) Searchgraph

Figure 3.2 The generation of a searchgraph for a class of a partitioned graph.

Phase I: Partition the graph into k clusters of about equal size.

Phase II: Choose base-nodes for each cluster.

Phase III: Calculate the searchgraph H_i for each cluster V_i in such a way that you will always find a path from every node $s \in V_i$ to every other node t.

Phase IV: To calculate a path from some node s to some node t apply the backward Dijkstra algorithm on the graph H_i.

The first three phases can be seen as preprocessing phases for the actual shortest path search in phase four. For each of the three preprocessing phases a number of different methods can be used to accomplish its goal. We describe a few methods in section 3.4 along with a discussion of their application in a pure dynamical setting with continously changing edge weights.

If d is the average degree in G and $E^* \ll E(G)$ then the number of edges in H_i is approximately $(n-n/k)+d*(n/k)/2 = n+(d/2-1)n/k$. Since d is typically less than five in a road network the search graph H_i has not significantly more edges than there are nodes in G and the number gets smaller if the number of clusters k is increased.

3.4 Processing the tree heuristic

In this section we will describe the different approaches we used for the three preprocessing phases of the tree heuristic, i.e. the graph partitioning, base-node and searchgraph generation. In section 3.5 we show our experimental results of shortest path calculations on the road network of Northrhine-Westphalia.

3.4.1 Graph partitioning

In the first phase of the tree heuristic the vertices of the road network have to be partitioned into a number of k subsets of approximately equal size. This partitioning should reflect the fact that the graph is of special type, i.e. a road network and that the objective is the shortest path calculation between two arbitrary nodes in the network. The requirement of approximate equally sized clusters shall assure that the expected running time for the shortest path computation will be independent of the partition class. Since the computational complexity of the used implementation of Dijkstra's algorithm is $O(|V| \log |V| + |E|)$ a better measurement to fulfill this requirement would be additionally an approximately equal number of edges in each class. However, the structure of road networks with evenly distributed nodes in relation to their degree should make the concentration on the nodes a reasonable approximation.

The partitioning problem described above is closely related to the k-way partitioning problem which is defined as follows [61]: Given a graph $G = (V, E)$ with $|V| = n$, partition V into k subsets $V_1, V_2, \ldots, V_k$ such that $V_i \cap V_j = \emptyset$ for $i \neq j$, $|V_i| = n/k$, $\bigcup_i V_i = V$ and the number of edges of E whose incident vertices belong to different subsets is minimized. Even for $k = 2$ the problem is NP-complete [34] but can be solved in polynomial time by standard network flow techniques if no restriction is made on the sizes of the subsets [57].

While both partitioning problems seek to find subsets of approximately equal size, a good partition for the k-way partitioning problem is one which minimizes the weight of the edge cut. In contrast a good partition for the tree heuristic should generate subsets such that the shortest path trees of the nodes in one cluster are very much alike. Intuitively one would suggest that therefore the induced subgraph of the nodes of every cluster should at least be strongly connected. Combining the objective of the k-way partition with that of the tree heuristic leads to the question to what extent a good partition for the latter is related to a small edge cut.

We tested three different approaches in order to find a good partition of the given road network. The first is a direct partition into k subsets by calculating k shortest path trees of size approximately $|V|/k$ by applying Dijkstra's algorithm. The second approach uses the METIS library [62], a software package by Karypis and Kumar which implements a multilevel k-way partitioning scheme for irregular graphs. This library was chosen because of its fast computation of partitions of high quality. The third approach tries to combine the objective of shortest path tree conformity in the generated subsets with the quality of the partitions generated by METIS and makes use of a so called *treegraph*. The three approaches are described in more detail next.

3.4.1.1 Dijkstra-Partition

A direct approach for finding a partition into k subsets of approximately equal size which gives clusters of high uniformity concerning shortest path trees is to choose k nodes in the network and run Dijkstra's algorithm with these nodes as starting vertices. One of the main advantages of this approach is that the generated clusters will be connected. The proposed algorithm belongs to the greedy type of partitioning algorithms the first of which was presented by Farhat [26]. Our algorithm is closely related to the one developed by Ciarlet et al. in [80] for partitioning meshes into equally sized connected subgraphs.

The latter algorithm uses a Breadth-First-Search approach to iteratively distribute the nodes into k classes of size n_i. n_i is chosen as $(|V| - \sum_{j=1}^{i-1} n_j)/(k - (i-1))$. If there are not enough nodes found for some n_i during the iteration, these nodes are redistributed to neighbouring classes V_j for $j < i$ and n_i is updated accordingly. Thus, the sequence of sizes of the classes is non-increasing and the classes might

become very small towards the end of the iteration. If $k-1$ classes have been determined the last class will consist of the largest connected component of the remaining nodes. Nodes that have not been distributed at that point are assigned to neighbouring classes. Therefore, the classes might vary significantly in size and the algorithm applies a postprocessing to balance and reshape the subsets [81], thereby preserving the connectivity of the subsets. The postprocessing is a variant of the heuristic first proposed by Kernighan and Lin [64] with its modifications by Fiduccia et al. [28] and Hendrickson and Leland [47, 48]. A partitioning algorithm using Breadth-First-Search is also presented in [40], but there the resulting subsets need not be connected.

Our approach for a direct partitioning of the graph into k connected subgraphs differs from the one of Ciarlet et al. in two ways. First we use Dijkstra's algorithm in order to reach nodes that have not been assigned to any class yet. Second we do not need any of the cited postprocessing heuristics to improve the balance of our partition. Instead our algorithm computes k shortest path trees of about equal size and assigns the remaining nodes to neighbouring trees using a minimum spanning tree approach. The algorithm can be divided into three phases which we will now describe in more detail.

Preprocessing phase. In a preprocessing all dead end roads are successively removed from the network and each remaining node is given a weight that corresponds to the number of dead ends which were reduced to this node[3]. We call this the reduced original network. Since stretches of dead end nodes must be assigned to the same class as the starting node w of the stretch in order to get connected subsets, they can be identified with w. By removing dead end nodes we wish to calculate more balanced subsets in the second phase of the algorithm. If the preprocessing is omitted it might happen that balanced shortest path trees calculated get very unbalanced when all stretches of dead end nodes are assigned in the third phase of the algorithm.

Shortest path tree partitioning. In the second phase of the algorithm k nodes are chosen and a shortest path tree for each is computed. The calculation of the shortest path tree stops if no more vertices can be added to the tree or as soon as the tree has size $c_1 * (n/k)$ where the size of the tree equals the sum of the weights of the vertices. If the tree has less than $c_2 * (n/k)$ nodes it is discarded and a new starting node is chosen. If no shortest path tree of sufficient size can be found after some fixed number of iterations it the algorithm terminates without a result[4]. Naturally $c_2 \leq c_1 \leq 1$. The factor c_1 helps to govern the size of the trees such that the algorithm will find a tree of reasonable size even for the last partition class.

[3] With this we mean that a whole stretch of links ending at some dead end node is removed and the starting node of this stretch gets as weight the number of nodes removed plus one.

[4] For the value $it = 100$ this didn't happen in any of our calculations.

Therefore, c_1 should decrease if the number of classes increases. c_2 prevents that the trees all have very different sizes which would make the partition classes very unequally sized.

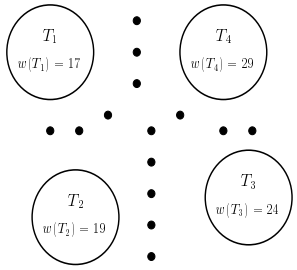
Each starting node is chosen out of those vertices which have not yet been included in a shortest path tree of previously chosen nodes. After the k shortest path trees have been calculated all those vertices have to be distributed which were not visited by any of the k shortest path trees.

Distribution of the remaining nodes using a minimum spanning tree. First we assign all nodes for which all neighbours have been distributed to the class of minimum weight of all neighbours. For the remaining nodes we calculate a minimal spanning tree with Kruskals's algorithm [68] on the following graph H (see figure 3.3): The nodeset of H consists of all nodes of the reduced original network which have not yet been assigned. In addition there is one node for each of the k shortest path trees. There are $k-1$ edges of weight 0 in H connecting the k cluster nodes. For each edge (u, v) in the original graph with u not yet assigned and v in the shortest path tree T^i let u_H and T^i_H be the corresponding vertices in H and c_H be the size of T^i. Create an edge (u_H, T^i_H) in H with weight c_H. For each edge (u, v) in the original graph with u and v both not assigned create an edge (u_H, v_H) in H with weight $c_{max} + 1$ where u_H, v_H are the corresponding vertices in H and c_{max} is the maximum size of the shortest path trees.

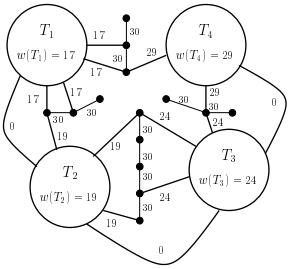
By calculating a minimum spanning tree T_{min} for the graph H each unassigned node u will be connected to a cluster T_i if there is such an edge in H or connected to such a node by a path of edges of weight $c_{max} + 1$[5]. The unassigned nodes that are connected to some T^i_H and those that are connected to T^i_H via a path in T_{min} are assigned to the class i. No unassigned node is connected to two different cluster nodes T_i since all cluster nodes are connected via a path of weight 0 in T_{min}. By giving each edge between an unassigned node and a cluster node the size of the tree as weight an unassigned node is assigned to the class of minimum weight of all neighbouring classes. We calculate the class of each unassigned node by applying Dijkstra's algorithm on the minimum spanning tree T_{min} with node T^0_H as starting node, distributing all nodes to the class of the node T^i_H from which the node is reached.

Correctness of the algorithm. In phase two of the algorithm all nodes of some shortest path tree are assigned to the same class. Thus, all subsets are connected after the shortest path tree computation. Also in the third phase of the algorithm a node is assigned to a cluster only if there is an edge in the original graph to a node which is in the same cluster, since all edges in the graph H that are incident to some unassigned node correspond to some edge in the original graph. Therefore, the generated partition classes will be connected.

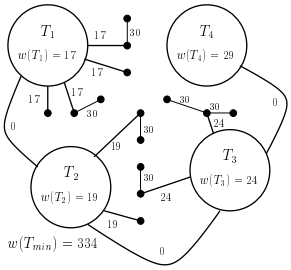
[5] This is true since the original graph is connected.



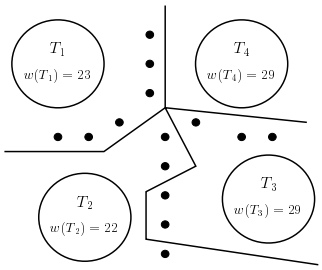
(a) Graph G with 4 classes T_i of shortest path trees. The edges between remaining nodes are not shown.



(b) Construction of the graph H.



(c) Minimal spanning tree for H



(d) Final 4-way Dijkstra partitioning

Figure 3.3 Distribution of remaining nodes using a minimum spanning tree in H.

Complexity of the algorithm. The preprocessing has complexity $\mathcal{O}(|E|)$ since we have to scan each edge of the original graph at most once. In phase two we calculate at most $k \cdot it$ shortest path trees of size at most n/k giving an overall complexity of $\mathcal{O}(|V| \log |V|)$ since it is treated as constant. For the complexity of the third phase let $E(H)$ be the set of edges of graph H for which a minimum spanning tree is computed. Every shortest path tree computed in phase two has at least $c_2 * n/k$ nodes, thus the graph H has at most $(1 - c_2) * |V| + k$ nodes and therefore at most $d * (1 - c_2) * |V| + k$ edges, where d is the average degree in G. The calculation of the minimum spanning tree with Kruskal's algorithm has complexity $\mathcal{O}(|E(H)| \log |E(H)|)$ giving an overall complexity of the partitioning algorithm of $\mathcal{O}(|E| + |V| \log |V| + (d * (1 - c_2) * |V| + k) \log(d * (1 - c_2) * |V| + k))$.

3.4.1.2 METIS-Partition

There are a variety of software packages for the graph partitioning problem including CHACO [49], METIS [62], PARTY [86] and TOP/DOMDEC [27]. In [100] CHACO, METIS and PARTY are extensively studied for the case of road networks. In this study the METIS software-library shows very good results with respect to runtime and quality of the partitioning. Therefore, we used METIS to compute graph partitions for our test network.

This software-library implements a multilevel k-way partitioning scheme described in [61]. The algorithm proceeds in three steps. First the graph is coarsened down to a few hundred nodes and this reduced graph is then partitioned into k parts directly by using the multilevel bisection algorithm from [60]. In the third step the partitioning of the coarser graph is projected back to the original graph using a variant of a k-way Kernighan-Lin algorithm for refinement [50, 64]. For more details see [61] and [59] for a theoretical analysis. The described algorithm computes a k-way partitioning of a graph in O($|E|$) and produces excellent partitions with respect to a minimum edge cut [59]. The algorithm allows to assign weights to the nodes such that the generated node classes are approximately of equal weight.

For our experiments we used the METIS library with different weight functions for the edges. This allowed us to study the effects of different partitioning objectives on the tree heuristic. E.g. the multilevel k-way partitioning algorithm does not necessarily give connected subsets of nodes.

3.4.1.3 Treegraph Partition

The third partitioning method we used is a twofold partitioning algorithm. In the first step a geometrical partitioning procedure is applied using the geometrical information given with road networks. Using this partition a new graph is defined that tries to capture shortest path tree characteristics of the original road network.

This so-called treegraph is of much smaller size than the original graph. In the second step the treegraph is partitioned with the algorithm of the METIS-library [62].

Geometrical partitioning and treegraph generation. A well-known geometrical partitioning heuristic is the inertial bisection, where a graph given with two-dimensional coordinates is partitioned recursively along straight lines until a given number of classes are generated. Our algorithm instead divides the original network G into squares by placing a grid of length l over it. From all nodes of G in a gridsquare S_{ij} we choose a representative v_{ij} of high degree and calculate its shortest path tree T_{ij} in G. All nodes in a gridsquare will later be assigned to the class the representative of the square is assigned to. In order to build the treegraph, we proceed as follows:

For each gridsquare S_{ij} the treegraph T_G has a node w_{ij} and edges (w_{ij}, w_{i+1j}) and (w_{ij}, w_{ij+1}), where the edges in T_G are undirected. For the weight of an edge (w_{ij}, w_{kr}) we count the number of edges which are in both trees T_{ij} and T_{kr} and assign this number as weight to the edge. Each node in T_G gets as weight the number of nodes of G which lie in the corresponding gridsquare. In figure 3.4 we sketch the construction of the treegraph.

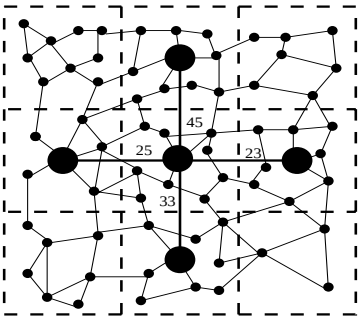


Figure 3.4 Construction of the treegraph: The large nodes are the representatives of each gridsquare. The weight of edges between these nodes counts the number of edges present in both shortest path trees of two nodes.

If we think of each representative of a grid square as being a center node, the treegraph can be viewed as a weighted gridgraph where each edge weight measures, how much the shortest path trees of the corresponding nodes are alike and where each node weight gives the number of nodes the center node represents.

The size of T_G depends on the length l but will in general be much smaller than the original graph G. By applying the k-way multilevel partitioning scheme from the METIS software-library to the treegraph T_G with the described node and edge weights we get a partition of T_G which can directly be expanded to a partition of G into k parts by assigning each node to the part of the corresponding gridsquare of T_G. The use of node weights in the partitioning therefore gives node sets of approximately equal size. The objective of a minimum edge cut in the partition of T_G tries to assign two nodes w_{ij} and w_{kr} to different parts when the shortest path trees T_{ij} and T_{kr} of the two representatives of the corresponding gridsquares do not have many edges in common.

If the gridlength l is small compared to the whole graph the shortest path tree of the representative of a gridsquare should look very much like the shortest path tree of any node in this square. The partition therefore should generate classes for which shortest path trees for two nodes in the same class are expected to be more alike than for two nodes of different classes. On the other hand a smaller l leads to a treegraph of greater size and therefore to more shortest path trees which have to be computed and compared in order to build T_G which causes this approach to have a significant higher running time than the other two algorithms (see also section 3.5.1.1).

3.4.2 Base-node generation

In order to build the searchgraph H_i for each class of the partition we have to determine base-nodes for each class for which a shortest path tree is computed. We tried three different approaches to determine the b base-nodes for each partition class. Note that the number b must not be the same for all classes, but can instead be chosen individually for each class thereby using some possibly known information about the specific class. The third approach is only applicable to a partition generated with the treegraph procedure. In all three procedures only those nodes were chosen as bases which had outdegree at least one such that a shortest path tree could be computed.

Euclideanbase approach. This is a very fast and simple approach that tries to find base-nodes which are geometrically as far apart as possible. The first base-node is chosen near the geometrical center of the class and therefore called *center-node*[6]. The other $b-1$ base-nodes are chosen iteratively with a startbase of nodes near the border of the class trying to maximize the sum of Euclidean distances between each pair of the $b-1$ base-node candidates. In each iteration $b-1$ nodes are randomly chosen and replace the current base if the objective function increases,

[6] In case that the geometrical center of the class is not part of the class we take an arbitrary node of the class.

keeping the centernode fixed. The iteration is halted either after a given number of steps or when there is no change in the basis for some number of steps. We call this procedure the *Euclideanbase* approach.

Sptbase approach. We use the same procedure as in the *Euclideanbase* approach with a different objective function for basis exchange. For each of the $b-1$ basis candidates we compute the shortest path tree in G and take the number of edges appearing in at least one of these trees as the objective function. Maximizing this number should lead to base-nodes with diversified shortest path trees such that the chosen nodes are good representatives of the class. This approach is by far not as fast as the *Euclideanbase* approach since in each iteration $b-1$ shortest path trees have to be generated and the number of different edges in these trees have to be computed. We will refer to this approach as *sptbase* approach.

Instead of choosing the $b-1$ base-nodes at random in each iteration of the latter two procedures we also tried a different approach by picking only neighbours of the current base-nodes as candidates. This leads to only a few iterations in most runs (less than twenty) together with smaller values for the objective functions than in the random case. The approach was therefore not studied in more detail.

Gridbase approach. For the *treegraph-partition* we also used a third method for the base-node generation. Since each partition class consists of a number of gridsquares with discrete coordinates we determined the two gridsquares with minimum and maximum x (y) coordinate for the minimum and maximum y (x) coordinate. Together with a gridsquare in the center of the class this gives a total of nine gridsquares. If two gridsquares are close together (i.e. $|x_1-x_2|+|y_1-y_2| \leq 3$) only one was chosen giving a minimum of five squares. In each square the representative of the square was chosen as base-node. This approach tries to find base-nodes which reflect the geometry of the partition class. We will refer to it as *gridbase* approach in the following.

3.4.3 Searchgraph generation

The proposed tree heuristic builds searchgraphs $H_1, \ldots, H_k$ for each of the k clusters $V_1, \ldots, V_k$ of a given partition for the network. The shortest path from a node s to a node t is computed on the graph H_i if s lies in the cluster V_i. The graph H_i has significantly less edges than the original graph but should be build in a way that the heuristic always finds at least some path between s and t if there is such a path in the original graph G. Since not all edges of the whole graph are present in the searchgraph the path between s and t that is found by the heuristic will not necessarily be the shortest path in G. Therefore, the generation of the searchgraphs should try to make this deviation as small as possible.

Let $G = (V, E)$ be the graph of the whole road network. The searchgraph $H_i =$

(V_i, E_i) of a specific cluster V_i has nodeset $V_i = V$. The set of edges is the union

$$E_i = E_i^{in} \cup E^* \cup E(T_1) \cup \ldots \cup E(T_{b_i})$$

where $E_i^{in} = \{e = (u, v) \in E : u \in V_i \text{ or } v \in V_i\}$ is the set of edges having at least one incident node in V_i, $E(T_j)$ are the edges of a shortest path tree T_j for base-node v_j and E^* are edges of high importance for shortest paths for starting nodes $s \in V_i$. The set E^* is small compared to the total number of edges in H_i. In all our experiments we took the edges of type four as the set E^* making up less than 1% of the edges of the searchgraph for the network of NRW.

In the searchgraph H_i it might happen that a path from a node $s \in V_i$ to a node $t \notin V_i$ is not found because some edge is neither in E_i^{in} nor in the union of the shortest path trees. In this case we say that the partition class is not closed under shortest path calculations. To avoid this phenomenon we compute a shortest path tree T_c for the center base-node b_c on the reversed edge set of G (that is we switch the direction of each edge in E). For all nodes $v \in V_i$ we check that all edges on the reversed path from v to b_c are included in H_i' and if not, add missing edges to E^*. This guarantees that we reach the centernode b_c from all nodes in the class V_i and thus all nodes in G, since for b_c the shortest path tree is included in H_i. This gives the final searchgraph H_i for a class V_i.

To generate H_i we have to compute b shortest path trees on G and another one for b_c on the reversed edge set of G. This gives an overall complexity of $\mathcal{O}(|E| + ||V| \log |V||)$ since making the graph closed under shortest path calculations has complexity $\mathcal{O}(|E|)$ and k and b are treated as small constants.

In section 3.5.1.1 we compare the runtime performance of the first three phases of the tree heuristic for the network of Northrhine-Westphalia.

3.5 Experimental results

We conducted an extensive experimental test of the proposed tree heuristic on the road network of Northrhine-Westphalia with 457124 nodes and 1046087 edges giving an average degree of approximately 4.6. The maximum degree is 12. Each edge has a weight which gives the integer length of the edge in $10m$ and the type of the road the edge represents. Table 3.2 shows the distribution of the edges according to their type. A high percentage of 71.4% of the edges are of type zero, while only 0.4% are of type four making a total of 4263 edges of the highest level. The nodes are given with their $x-y$ coordinates with respect to the Gauss-Krüger system [52]. The graph is connected but not strongly connected because of vertices on the border and unidirectional stretches in the highest hierarchy level.

Type	Description	#Edges	Percentage
0	non-artery artery	746967	71.4%
1	collector artery	85465	8.2%
2	local artery	160788	15.4%
3	secondary artery	48604	4.6%
4	primary artery	4263	0.4%
Total		1046087	100%

Table 3.2 Distribution of the edges according to their type for NRW.

3.5.1 Experimental setup

We tested the tree heuristic with the different partitioning and base-node generation methods described in section 3.4 on the network of NRW. For values $k = 8$, $k = 12$ and $k = 16$ we computed a k-way partitioning choosing different weight functions for the METIS partitioning and different gridlengths l for the treegraph partitioning.

For each partition of the network into k clusters we randomly chose 50 starting nodes in each partition class and 75 target nodes in the whole graph. We then calculated a total of $50 * k * 75$ shortest paths by using the backward Dijkstra algorithm in the searchgraphs of the tree heuristic. The suggested paths of the heuristic were compared with those of Dijkstra's algorithm and a bidirectional variant of Dijkstra's algorithm on the whole graph, with those of the well-known A^*-algorithm and the so called HISPA heuristic (see section 2.4.3 and section 2.6 for details about these algorithms).

For the running time of the algorithms (excluding i/o) the timing system of the operating system with granularity 0.01 seconds (1 tick) was used. To evaluate the quality of the solutions of the different algorithms we reported the following facts for each calculated shortest path: Length of the path, number of edges of the path, number of scanned nodes and the cluster of the starting node. The latter was used for a better understanding of the relationship between the quality of shortest paths and the structure of the partition classes.

To measure the quality of the shortest paths we calculated the average traveling time error, the maximum relative traveling time error and the percentage of paths which have an error greater than a given threshold. For the searchgraphs we concentrate on the different sizes of the partition classes, the number of edges of the searchgraphs and the number of edges which are present in all searchgraphs for a given partition. If this quantity is high the space requirements for the tree heuristic are reduced.

From section 3.5.2.1 to section 3.5.2.3 we present the results of the tree heu-

ristic for an 8-way partitioning of NRW for the three partitioning methods from section 3.4.1 with respect to the characteristics of the searchgraphs and the quality of the shortest paths. For the results we generated five base-nodes for each class for the Dijkstra and METIS partitioning with the *Euclideanbase* approach and for the treegraph partitioning we used the *gridbase* approach giving not always the same number of base-nodes for every class. An overview over the preprocessing times of phases one to three of the tree heuristic for these cases is given in section3.5.1.1.

In section 3.5.2.4 we analyze the effects of choosing $k = 12$ and $k = 16$ in the k-way partitioning. Results for different base-node generation methods and a different number of base-nodes are given in subsequent subsections. The runtime performance of the heuristic compared to Dijkstra's algorithm and its variants is analyzed in section 3.5.3.

Hardware and Software Support. The experiments were performed on a Sun Enterprise E4500/E5500 with 5.0 GByte RAM and 12 UltraSparc-II CPU's with 336 MHz, running under Solaris 2.5. We used the SUN Workshop CC compiler with optimization flag O4. Our software code made use of the software library LEDA [74] using a bounded priority queue for Dijkstra's algorithm and a priority queue for the A^*-algorithm.

3.5.1.1 Comparison of preprocessing times

In table 3.3 we give the running times and space requirements for the different preprocessing phases of the tree heuristic with its various generation methods for an 8-way partitioning with five base-nodes for each class. The Dijkstra and METIS partitionings take less than a minute, where more than 25% of the nodes in the original network lie on the dead-end stretches for the Dijkstra partitioning. The treegraph partitioning proceeds in two steps, where the times given in the table apply to the generation of the treegraph. The actual partitioning of this treegraph takes less than a second.

For the different methods of the base-node generation we observe that the *sptbase* approach takes substantially longer than the other two approaches. The number of iterations until a basis is kept when there is no increase in the objective function is given in the last column. The generation of the searchgraphs for an 8-way partition takes less than five minutes.

3.5.2 Quality of solutions

3.5.2.1 Dijkstra partition

As described in section 3.4.1.1 the Dijkstra partition computes k distinct shortest path trees of size approximately n/k which correspond to the partition classes.

Phase	Method	Runtime in sec	Memory in Mb	Note
Partition	Dijkstra	15.4	213	120277 dead-ends
	METIS	45.1	439	
	Treegraph	1426.3	145	l=10000m
	Treegraph	5341.2	159	l=5000m
Base-nodes	Euclideanbase	2.0	136	10000 iterations
	SPTbase	4064.8	165	50 iterations
	Gridbase	< 1.0	< 1.0	
Searchgraph		211.5	175	

Table 3.3 Preprocessing times of the tree heuristic for NRW.

Nodes that are not reached by any of the trees are assigned to the k classes in such a way that each class stays connected and all classes are approximately of equal size.

We chose the starting nodes for the shortest path trees at random out of all nodes which have at least one outgoing edge and used $c_2 = 0.8$ as factor for the lower bound for the size of the trees. For the 8-way partitioning we chose $c_1 = 0.95$. In table 3.4 some statistical characteristics of the partition are given. The base-nodes were generated with the *Euclideanbase* approach maximizing the sum of the Euclidean distances for the $b-1$ border nodes.

$\sigma_V/|V|$ in table 3.4 is given with $\sigma_V = \sqrt{\frac{1}{k} \cdot \sum_{j=1}^{k} (n_j - \overline{n})^2}$ where n_j is the number of nodes in partition class V_j and $\overline{n}$ the average number of nodes in the classes. Thus, σ_V measures the deviation of the partition with respect to equally sized partition classes. Another measure of this kind is *Nodedev* which shows the difference between the largest and the smallest node class as percentage of the average number of nodes in each class. *Edgedev* and $\sigma_E/|E|$ denote these numbers for the edges. *Avg. Edges* is the average number of edges per class as percentage of the total number of edges in G. While the average number of nodes is determined by k, the value for the edges depends on the structure of the partition. $|E^+|$ is the average number of edges per class which have to be added to guarantee that a shortest path from an arbitrary node in the class will be found. In our terminology $|E^+|$ is the number of edges needed to make the searchgraph closed under shortest path calculations (see section 3.4.3 for details). *Avg. Components* resp. *Max. Components* denote the average resp. maximal number of components of a partition class which is one for the Dijkstra partitioning by construction. *All-edges* gives the percentage of edges of the graph G which are present in the searchgraphs for all partition classes and *Avg. Bases* denotes the average number of bases in each class which is five for all

classes for this partition.

DIJK	# Parts
	8
$\sigma_V/\|V\|$	0.002
Nodedev	5.6%
$\sigma_E/\|E\|$	0.02
Edgedev	11.1%
Avg.Edges	55.9%
$\|E^+\|$	10.5
Avg. Components	1
Max. Components	1
All-edges	38.3%
Avg. Bases	5
c_1	0.95

Table 3.4 Partition characteristics for the 8-way Dijkstra partition.

The results in table 3.4 show that the partition is very well balanced with respect to nodes. With respect to the number of edges the classes vary more in size and this number is of greater importance for the running time of the tree heuristic. The average number of edges in the searchgraphs is about 56% leading to an average degree in the searchgraphs below 2.4. That is, the generation process for the searchgraphs produces a network that has a very path-like structure. By construction all partition classes are connected and the number of edges that have to be added to make the searchgraph closed under shortest path calculations is small. That there are edges that have to be added results from the difference between connectivity and strong connectivity. During the partitioning we only require the partition classes to be connected regardless of the orientation of any edges in the graph. Neglecting the orientations of the edges during the partitioning therefore might lead to missing edges. A picture of this 8-way Dijkstra partition is shown in figure A.1 (a) in the appendix.

To analyze the quality of the shortest paths found by the tree heuristic we use different error measures. One is the difference in travel time between the suggested path found by the heuristic and the actual shortest path in the whole graph G found by Dijkstra's algorithm. Figure 3.5 shows the relative error in percent of the length of the actual shortest path as function of this length for the 8-way Dijkstra partitioning. The shortest paths were grouped into bins of length five minutes and for each bin the mean error was calculated.

The mean error is below 1.5% and decreases below 0.5% for paths longer than

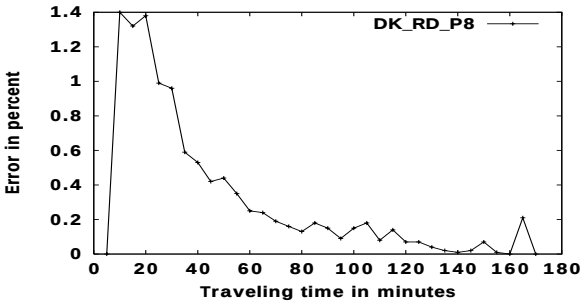


Figure 3.5 Mean error of paths for the Dijkstra partition with random starting nodes.

40 minutes. Since each partition class P contains all edges that have at least one endnode in P the errors made for short paths result from paths where the target node t lies just outside P or the exact shortest path between s and t uses an edge having both endnodes not in P. Because of the shortness of the paths the relative error for these paths will be higher.

As second error measure we use the maximal relative error in traveling time made by paths. Figure 3.6 shows this error, again plotted against the length of the paths. This error is very high for short paths, but decreases to below 15% for paths longer than 60 minutes. Paths with a high relative deviation found by the heuristic are of great interest, especially if they are not rare. To measure the expected frequency of such high deviations we use the fraction of paths with a relative error greater than a given threshold. Figure 3.7 shows the data for the Dijkstra partition for thresholds $0\%, 1\%, 5\%$ and 10%. While the percentage of erroneous paths is around 10% for all path lengths, the fraction of paths with relative error greater than 5% or 10% tends to zero for longer paths.

Summarizing the results one can say that the expected traveling time error of paths found by the tree heuristic on searchgraphs generated with the Dijkstra partition method with random starting nodes lies below a few percent. The relative errors are higher for short paths and can almost be neglected for paths longer than 60 minutes where even the maximum errors are in the range of a realistic traveling time dispersion.

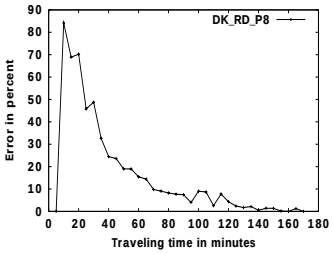


Figure 3.6 Maximum relative error of paths for the Dijkstra partition.

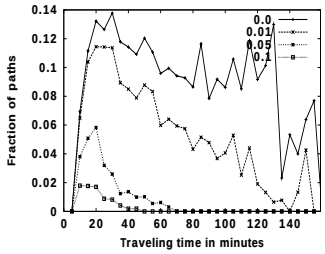


Figure 3.7 Fraction of paths with an error greater than a given threshold for a 8-way Dijkstra partition.

3.5.2.2 METIS partition

The METIS software package [62] computes a k-way partition with the objective of a small edge cut. By using different weight functions we studied the influence of various edge characteristics on the generated partition and ultimately on the tree heuristic. Thereby, we used the following quantities for each edge e: the length $l(e)$ in $10m$, the type $t(e)$ in the range $[0,\ldots,4]$ and the lcm-speed $lcm(t(e))$. The latter is the least common multiplier of the speeds for the different types of edges divided by the speed for edge e. Multiplying $l(e)$ with $lcm(t(e))$ gives the traveling time $tt(e)$ as integer value in $min/lcm(speeds)$.

We generated a k-way partition for $k = 8, 12$ and 16 for four different weight functions $c : E \to \mathbb{N}$ and compare the results of the different weight functions for the 8-way partition in this section. Results for the other values of k are given in section 3.5.2.4. The weight functions can be described as follows:

W1 $c(e) = (1 + l_{max} - l(e)) * lcm(e)$: This gives long edges of high type a smaller weight than short edges of low type. (l_{max} is the length of the longest edge in G.)

W2 $c(e) = l(e) * lcm(4 - t(e))$: This gives short edges of low type a smaller weight than long edges of higher type.

W3 $c(e) = tt(e)$: Minimizes the sum of the travel times for edges in the cut.

W4 $c(e) = 1$: Minimizes the number of edges in the edge cut.

Results for W1 and W2 Table 3.5 show the characteristics of the 8-way partition for the two weight functions W1 and W2. The two functions are contrary to each

other by favouring long edges of high type (W1) resp. small edges of small type (W2) to be in the edge cut. The base-nodes were generated with the *Euclideanbase* approach maximizing the sum of the Euclidean distances for $b-1$ border nodes.

METIS	8-way	
	W1	W2
$\sigma_V/\|V\|$	0.0007	0.005
Nodedev	1.7%	13.6%
$\sigma_E/\|E\|$	0.02	0.01
Edgedev	8.1%	7.4%
Avg.Edges	56.7%	59.6%
$\|E^+\|$	0.1	26.8
Avg. Components	1	2.5
Max. Components	1	5
All-edges	40.0%	42.2%
Avg. Bases	5	5

Table 3.5 Partition characteristics for W1 and W2.

The results show that the partitioning with weight function W1 is very well balanced with respect to nodes while the function W2 gives an equally sized partition with respect to the edges. For weight function W1 the partition classes are all connected and the number of edges that have to be added to make each class closed under shortest path calculations is very small. For function W2 the classes are mostly not connected although the number of connected components is small. The disconnectivity leads to more edges that have to be added to close the graph under shortest path calculations. At least 40% of the edges appear in all searchgraphs. Figures A.1 (c) and A.1 (d) in the appendix show a picture of these partitions.

To evaluate the quality of the paths found by the heuristic we use the same error measures as in the previous section. Figure 3.8 shows the relative error of paths found by the heuristic in percent of the length of the actual shortest path as function of this length for the two 8-way partitionings.

For paths longer than 20 minutes the heuristic on the average finds paths that are within 1% of the actual shortest path lengths for function W1. Only for paths that are very short the relative error increases up to 2.5%. For weight function W2 the results are not as good and have a relative error of about 9% for short paths decreasing slowly below 1%. For the maximal relative error shown in figure 3.9 the partitioning with weight function W1 also gives better results. For paths longer than 60 minutes the relative error is below 15% for W1, while this value is reached for weight function W2 only for paths longer than 100 minutes. Both partitionings

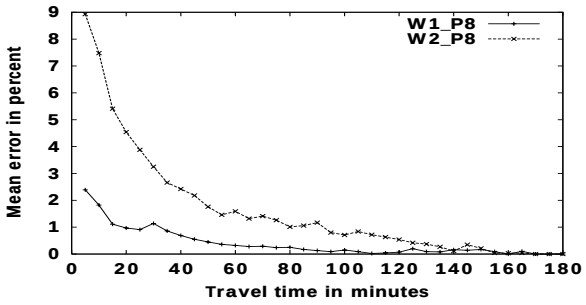


Figure 3.8 Mean error of paths for METIS partitions with weight functions W1 and W2.

have a maximal relative error above 200% for very short paths below 15 minutes. (The data is not shown for better presentation.)

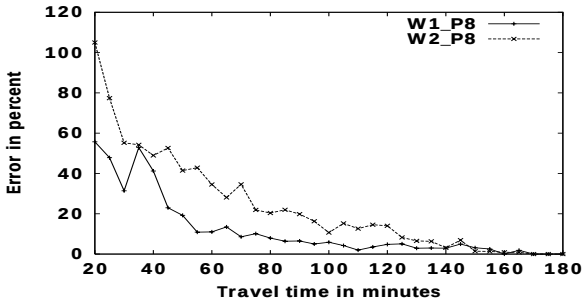


Figure 3.9 Maximal relative error of paths for METIS partitions with weight functions W1 and W2.

To analyze the frequency of paths with high relative error we show the fraction of paths with an error higher than a given threshold for weight function W1 in figure 3.10. The results for W1 are similar to the 8-way Dijkstra partitioning with a fraction of deficient paths below 15% and almost no paths of length at least 60 minutes with error above 5%. Figure 3.11 shows the same plot for weight function

W2. Not only is the fraction of deficient paths much higher for W2, being above 20% for paths up to a length of 120 minutes, but also the high error paths are more frequent. But even for W2 the fraction of paths with relative error greater than 10% is below 1% for paths longer than 60 minutes.

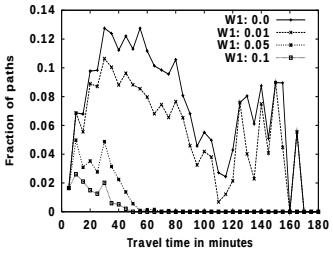


Figure 3.10 Fraction of paths with an error greater than a given threshold for a METIS 8-way partition with weight function W1.

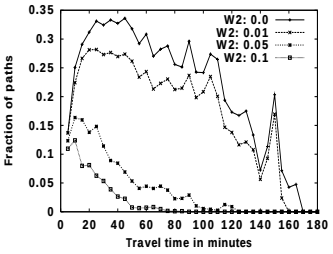


Figure 3.11 Fraction of paths with an error greater than a given threshold for a METIS 8-way partition with weight function W2.

In summary the paths found by the heuristic for a METIS 8-way partition with weight function W1 are of similar quality than those for the 8-way Dijkstra partition, where both partitionings generate connected subclasses. In contrast the 8-way METIS partition with weight function W2 results in some partition classes that are not connected. The qualitiy of paths found by the heuristic for this partition is worse than for the two others.

Results for W3 and W4 Table 3.6 shows the characteristics of the 8-way partition for weight functions W3 and W4, which minimize the sum of travel times of cut-edges resp. the number of cut edges. The base-nodes were again generated with the *Euclideanbase* approach maximizing the sum of the Euclidean distances for $b-1$ border nodes.

Weight function W3 is closely related to W2 since there are many short edges in the graph of low type. The most noteable fact from the data given is the high number of connected components for the METIS partition with weight function W3. No partition class is connected and there is a class with a maximum of 5242 components. This leads to the addition of many edges to make the searchgraphs closed under shortest path calculations and also to searchgraphs with more than 60% of the edges of the original graph on the average. Additionally, the partition is

METIS	8-way	
	W3	W4
$\sigma_V/\|V\|$	0.01	0.001
Nodedev	29.6%	2.5%
$\sigma_E/\|E\|$	0.02	0.003
Edgedev	8.4%	2.0%
Avg.Edges	61.3%	55.7%
$\|E^+\|$	6559.1	2.1
Avg. Components	1618.6	1
Max. Components	5242	1
All-edges	47.3%	39.2%
Avg. Bases	5	5

Table 3.6 Partition characteristics for W3 and W4.

not very well balanced with respect to nodes. In contrast the partition with weight function W4 which minimizes the number of edges in the cut consists of connected subclasses. The partition is very well balanced and the number of edges that have to be added to make the searchgraphs closed under shortest path calculations is small. Figures A.1 (e) and A.1 (f) in the appendix show a picture of these partitions.

Figure 3.12 shows the mean error in percent of paths found by the tree heuristic for the two partitions. For weight function W4 this error is very low, lying below 1% for paths longer than 30 minutes. For shorter paths it is slightly higher but far below the one for weight function W3 which is up to 25% for short paths. For longer paths the error is decreasing but does not tend as fast to zero as for weight function W4.

With respect to the maximal relative error the results for weight function W3 are even worse (see figure 3.13). For paths as long as 140 minutes this error stays above 20%. For weight function W4 the maximal relative error is much smaller, lying below 15% for paths longer than 70 minutes. For very short paths the error climbs up well above 200% for both weight functions.

For the fraction of deficient paths we show the results in figure 3.14 and figure 3.15. For weight function W3 almost 50% of the suggested paths of the tree heuristic are not the shortest path and a significant number of paths have an error greater than 10%. This fraction is well below 1% for weight function W4 for paths longer than 40 minutes and the number of deficient paths is below 15%.

Summary of METIS results The quality of the paths found by the tree heuristic is very much influenced by the weight function used in the partition process. The

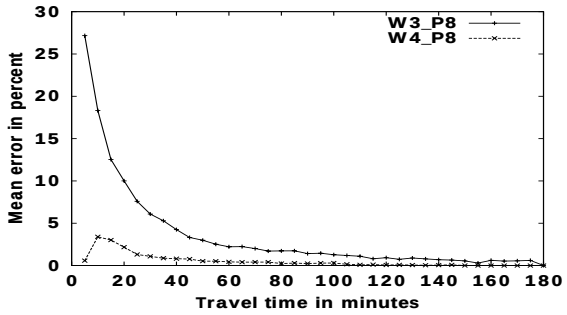


Figure 3.12 Mean error of paths for METIS partitions with weight functions W3 and W4.

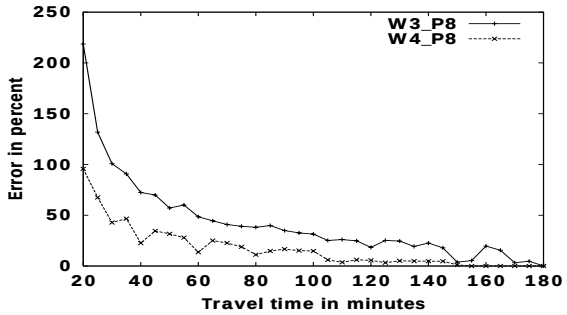


Figure 3.13 Maximal relative error of paths for METIS partitions with weight functions W3 and W4.

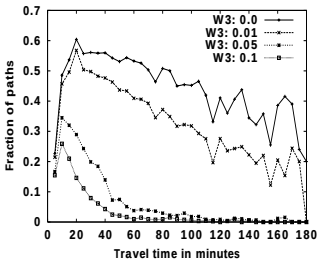


Figure 3.14 Fraction of paths with an error greater than a given threshold for a METIS 8-way partition with weight function W3.

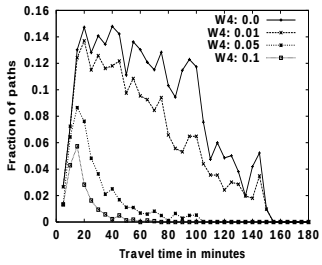


Figure 3.15 Fraction of paths with an error greater than a given threshold for a METIS 8-way partition with weight function W4.

main effect to be observed is that partitions that generate classes with many connected components lead to paths of greater error. Figure 3.16 shows the data for the fractions of paths of error greater than 10% for the 8-way partition using the four different weight functions.

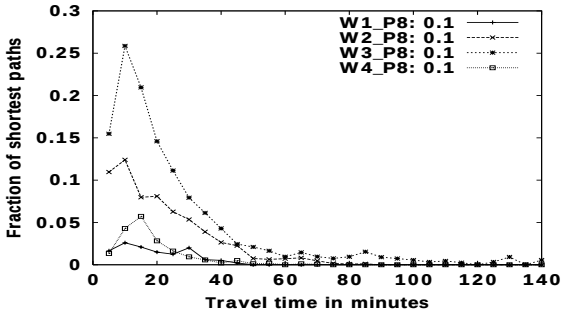


Figure 3.16 Fraction of paths with an error greater than 10% for METIS partitions with different weight functions.

The partitions with weight functions W2 and W3 with many connected components in the classes lead to a rather high percentage of error paths especially for

paths shorter than 40 minutes. For the other two partitions the percentage is small and tends to zero fast. The same classification can be made with respect to the average error of the traveling times. There are some differences for the different weight functions regarding the maximum error in each distance bin, but these are not as significant as for the other measures. This results primarily from the fact that the maximum error depends only on one *s*-*t* path. In most cases the maximum error in all distance bins is caused by the same bad starting node *s* or target node *t* for which the searchgraph misses some important link.

For the different weight functions it can be observed that the ones which favour short edges to be cut edges result in partitions with highly disconnected subclasses. Instead, if long edges of high type tend to be in the edge cut the subclasses are connected and the tree heuristic gives the best results for the METIS partitions. The paths found for the partition minimizing the number of edges in the cut are only slightly worse. Thus, the topology of the road network seems to be of less influence on the quality of the paths found by the tree heuristic than the observation that the partition classes should be connected.

3.5.2.3 Treegraph partition

For a partitioning of the original graph G with the *treegraph* approach (see section 3.4.1.3 for details) we studied the influence of the gridlength l on paths found by the tree heuristic. A smaller l gives a finer grid and thereby the treegraph T_G will reflect the shortest path tree structure in the graph G more accurately. If l was chosen so small that each gridsquare contains exactly one node of G then the treegraph would give a very good picture of shortest path tree differences in G since each edge in T_G gives the number of edges the shortest path trees of the adjacent nodes have in common.

Figure 3.17 shows the effect of the gridlength on the edge weights of the treegraph. The plot shows the cumulative distribution of the inverse edge weights. That is, instead of counting the number of edges two shortest path trees have in common we count the number of edges in which they differ. Thus, each data point (x, y) means that for y% of the edges of the gridgraph the shortest path trees of the two endnodes differ in less than x edges. While for gridlength 2500m more than 90% of the edges have an edge weight less than 10000 the rate is around 40% for gridlength 10000m. For edge weights smaller than 10000 (meaning a very high conformity of the shortest path trees) figure 3.18 shows the graphs in more detail.

The disadvantage of a smaller l is the longer preprocessing time for generating the treegraph and the size of T_G which has to be partitioned (see section 3.5.1.1). We have chosen the three values $l = 2500m$, $l = 5000m$ and $l = 10000m$ for the gridlength and generated an 8-way, 12-way and 16-way partitioning using the METIS software library minimizing the sum of the weights for the edge cut. In

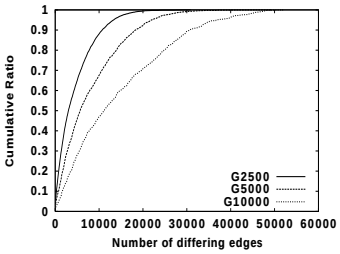


Figure 3.17 Cumulative distribution of edge weights for different gridlengths.

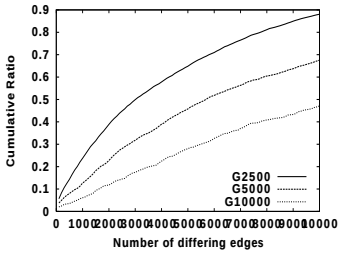


Figure 3.18 Cumulative distribution of edge weights for different gridlengths (detail).

table 3.7 the size of the treegraphs for the three values of l is given together with the average edge weight in the treegraph as percentage of the number of nodes in the original graph.

TG	Gridlength		
	2500	5000	10000
Number of Nodes	5149	1486	401
Number of Edges	9297	2829	739
Avg. Weight	99.0	98.3	96.9

Table 3.7 Size of the treegraphs for different gridlengths.

In this section we show the results for the 8-way partitioning using the *gridbase* approach (see 3.4.2) to generate base-nodes for each partition class. This approach does not give the same number of base-nodes for all classes of a partition. The results using the *Euclideanbase* approach were of similar quality and are therefore not shown here.

Table 3.8 gives the characteristics for the 8-way partition for the three gridlengths. For gridlength $2500m$ there is exactly one class which decomposes into two connected components, while for the other gridlengths all subclasses are connected. Since the treegraph is small compared to the whole graph, it does not have to be coarsened much and therefore we expect most of the partition classes to be connected. All three partitions are well balanced with respect to the nodes, for $l = 5000m$ the classes are also very equally sized with respect to the edges. Compared to a direct partition with the METIS software the number of edges that have

to be added to make the partition closed under shortest path calculations is high although almost all classes are connected. The reason for this is the rectangular shape of the partition classes due to the assigning of nodes into gridsquares. Since missing edges are determined by calculating a shortest path from the center base-node to all nodes of the class the partition using gridsquares will miss some edges on the border of the class which do not fit into the rectangular shape. The number of bases is between five and eight while the number of edges present in all search-graphs is rather low compared to the other partitionings. A picture of the 8-way treegraph partition for $l = 5000m$ is shown in figure A.1 (b) in the appendix.

TG	Gridlength		
	2500	5000	10000
$\sigma_V/\|V\|$	0.003	0.003	0.002
Nodedev	5.8%	6.0%	5.1%
$\sigma_E/\|E\|$	0.02	0.006	0.009
Edgedev	9.1%	3.4%	5.2%
Avg. Edges	56.7%	55.5%	55.7%
$\|E^+\|$	192.8	269.6	212.4
Max. Components	2	1	1
Avg. Components	1.1	1	1
All-edges	40.1%	37.8%	38.5%
Avg. Bases	5.62	6.75	5.88

Table 3.8 Partition characteristics for the treegraph partition.

Figure 3.19 shows the mean error for paths calculated by the tree heuristic for the three partitions. The relative error is small even for short paths and tends to zero for paths longer than 60 minutes. Comparing the three gridlengths shows that for $l = 5000m$ the observed errors are greatest.

The maximal relative error in a distance bin (see figure 3.20) is very small for gridlength $10000m$ being less than 15% for paths longer than 50 minutes. For the other two partitionings the error is slightly higher. For paths shorter than 20 minutes there are errors of more than 200% for gridlength $5000m$.

For the frequency of error paths we show in figure 3.21 to figure 3.23 the fraction of paths with a relative error greater than a given threshold. For all three gridlengths the fraction for threshold 10% is very close to zero. For $l = 2500m$ there are less than 10% of erroneous paths, for gridlength $10000m$ there are only sligthly more.

Compared to the METIS partitioning the quality of the solutions for the tree-graph 8-way partitioning is very good for all three gridlengths. This is especially

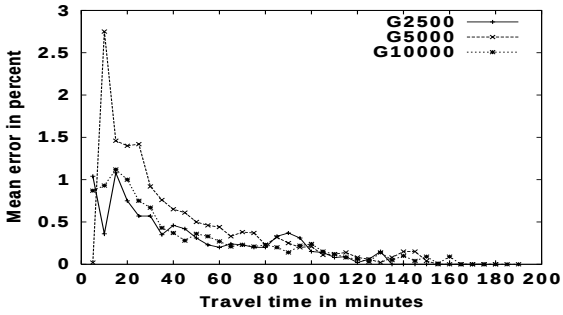


Figure 3.19 Mean error of paths for an 8-way treegraph partition for different gridlengths.

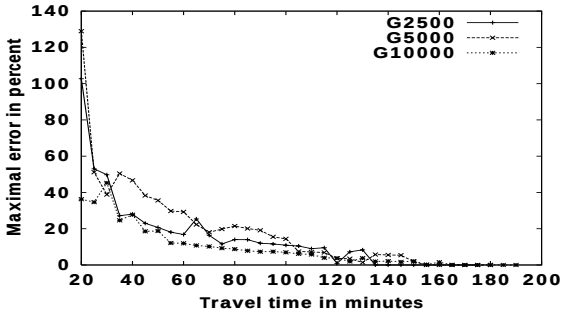


Figure 3.20 Maximal relative error of paths for an 8-way treegraph partition for different gridlengths.

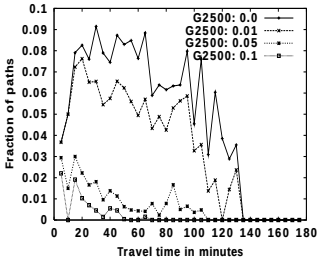


Figure 3.21 Fraction of paths with an error greater than a given threshold for the treegraph 8-way partition with gridlength 2500*m*.

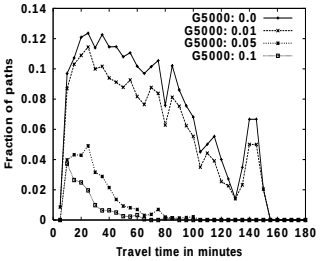


Figure 3.22 Fraction of paths with an error greater than a given threshold for the treegraph 8-way partition with gridlength 5000*m*.

true for the fraction of deficient paths which is very small for the treegraph partitions in comparison to the other methods. The differences between the partitions for the three gridlengths are very small, the quality of paths being slightly worse for $l = 5000m$. Thus, the effect of a smaller gridlength leading to a treegraph containing more information about the structure of the shortest path trees is not very pronounced for the NRW network.

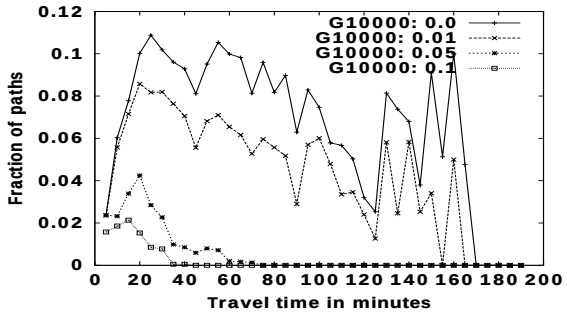


Figure 3.23 Fraction of paths with an error greater than a given threshold for the treegraph 8-way partition with gridlength 10000*m*.

3.5.2.4 Comparison of k-way partitions

In this section we compare the results of the tree heuristic, where different values for k are chosen for the Dijkstra k-way partition, i.e. we study the values $k = 8, 12$ and $k = 16$. We discuss the effect of different values for k for the other partitioning methods at the end of this section. Since the conclusions for the Dijkstra partitioning can also be drawn for the other methods, we abstain from showing them in great detail.

Table 3.9 gives the characteristics of the three Dijkstra partitions, the data for $k = 8$ is the same as in table 3.4. As can be seen the 12-way and 16-way partition are not as equally sized with respect to the number of nodes as the 8-way partition. However, the searchgraphs for these two partitions are more balanced with respect to the number of edges than for the 8-way partition.

DIJK	# Parts		
	8	12	16
$\sigma_V/\lvert V\rvert$	0.002	0.006	0.006
Nodedev	5.6%	25.1%	33.7%
$\sigma_E/\lvert E\rvert$	0.02	0.014	0.013
Edgedev	11.1%	8.6%	8.6%
Avg. Edges	55.9%	53.1%	51.7%
$\lvert E^+\rvert$	10.5	5.7	2.6
Max. Components	1	1	1
All-edges	38.3%	36.6%	35.7%
Avg. Bases	5	5	5
c_1	0.95	0.85	0.9

Table 3.9 Partition characteristics for the Dijkstra partition for different values of k.

Analyzing the quality of the paths found by the tree heuristic for the three partitions in figure 3.24 we observe that the 8-way partition gives slightly better results especially for short paths. For longer paths the differences are small. For the maximal relative error the curves are almost of identical shape and therefore are not shown.

Since the latter error depends only on exactly one s-t pair in each distance bin, it is not so well suited for comparison of different partitions. Instead, we show the plots for the fraction of paths having a relative error greater than a given threshold for $k = 12$ in figure 3.25 and for $k = 16$ in figure 3.26 with thresholds $0\%, 1\%, 5\%$ and 10%. For the 8-way partition the plot was given in figure 3.7. All three partitions show a very identical frequency of deficient paths with the 8-way partition

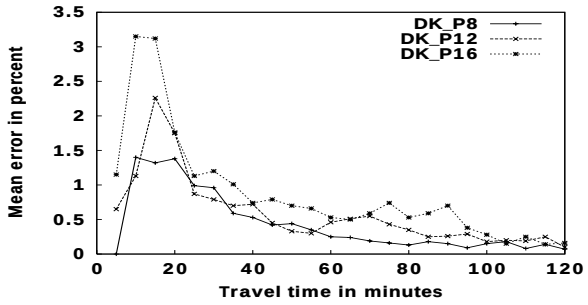


Figure 3.24 Mean error of paths for the Dijkstra partition for different k.

giving the best results for greater error paths.

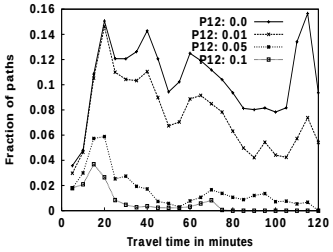


Figure 3.25 Fraction of paths with an error greater than a given threshold for a 12-way Dijkstra partition.

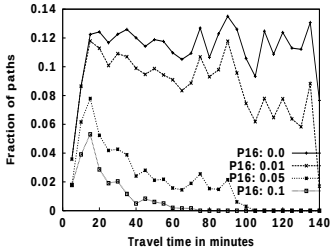


Figure 3.26 Fraction of paths with an error greater than a given threshold for a 16-way Dijkstra partition.

In summary we conclude that the 8-way Dijkstra partition gives slighty better results for all three error measures than the ones for $k = 12$ and $k = 16$. However, the differences are very small and do not allow a definite classification. The same conclusions can be drawn for the other partitioning methods we studied.

For the METIS k-way partitionings for different values of k the quality of the paths found by the tree heuristic was very much influenced by the number of connected components in the partition classes. For the weight functions W1 and W4

the partitioning resulted in connected subclasses in all cases. The errors observed for suggested paths were of the same magnitude as for the 8-way partition analyzed in section 3.5.2.2. For the other two weight functions the subclasses for $k = 12$ and $k = 16$ decomposed into a few connected components for W2 and classes with a few thousand components for W3. In line with our results from section 3.5.2.2 the paths found by the tree heuristic were of much poorer quality for these two weight functions. Since the 8-way partitioning resulted in the minimal number of connected components for all weight functions the quality of the paths was slightly better than for the other values of k.

The treegraph partitioning for gridlength $l = 5000m$ always gave connected subclasses for all three values of k. For $l = 2500m$ there was one subclass with two connected components in all cases. However, the effect on the quality of the suggested paths was not significant. For these two gridlengths the results were very similar for all three k-way partitionings with a slightly better quality for $k = 8$. For $l = 10000m$ the 12-way and 16-way partitioning were not very balanced with respect to nodes. The reason is that the number of nodes of G that are assigned to a gridsquare differs much more for this length. Therefore, it is much harder to find an equally sized partition with respect to the number of nodes of G. The effect on the number of edges per class was less significant. There were some disconnected subclasses for $l = 10000m$, especially for $k = 16$. This resulted in paths of better quality for the 8-way partitioning also for this gridlength.

3.5.2.5 Influence of the geometric shape of classes

Next to the connectivity of partition classes the geometric shape affects the quality of the tree heuristic. In figure 3.27 we show an almost convex class and in figure 3.28 two classes of irregular shape of a 12-way Dijkstra partitioning of NRW.

The different quality of paths found by the tree heuristic for these three classes is shown for the average relative error in figure 3.29 and for the fraction of deficient paths in figure 3.30. While the average error of the almost convex class is very low it is much higher for the classes of irregular shape especially for short paths. The fraction of deficient paths is above 30% even for long paths for the classes of irregular shape while it is below 5% for class 1.

Another aspect where the classes differ, is the number of border nodes, i.e. nodes for which at least one neighbour is in a different class[7]. The classes 10 and 11 have about twice as many border nodes than class 1. While classes of irregular shape will in general have more border nodes, the reverse is not true. There are classes in all our partitionings that are of almost convex shape, but have a high number of border nodes. For these classes the tree heuristic performs equally well

[7] Since the road networks we considered, have a very evenly distributed degree, the number of border edges is proportional to the number of border nodes.

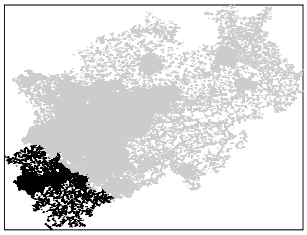


Figure 3.27 Class 1 of a Dijkstra 12-way partition.

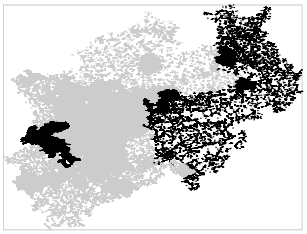


Figure 3.28 Classes 10 and 11 of a Dijkstra 12-way partition.

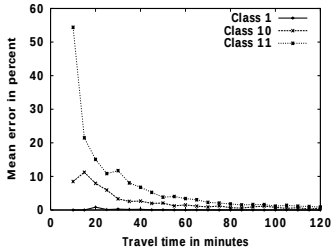


Figure 3.29 Mean error for different classes of a Dijkstra 12-way partition.

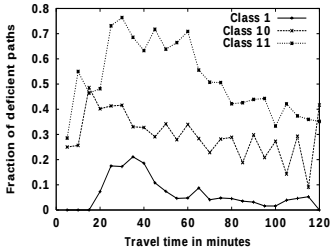


Figure 3.30 Fraction of deficient paths for different classes of a Dijkstra 12-way partition.

as for those of convex shape with few border nodes. Thus, in our analysis of the tree heuristic we did not observe a direct correlation between the number of border nodes and the quality of paths found by the heuristic.

3.5.2.6 Influence of the base-nodes

Generation method For the results shown in the previous sections we either used the *Euclideanbase* or the *gridbase* approach for the generation of the base-nodes (see section 3.4.2 for details). We also studied the *sptbase* approach, where the number of edges in the shortest path trees of the base-nodes is maximized. The generation of base-nodes with this approach is very time consuming compared to the other two methods (see section 3.5.1.1) and did not find paths of better quality. In fact the *Euclideanbase* approach yields in many cases bases where the shortest path trees of the base-nodes differed more than for the *sptbase* approach. In summary our results suggest that especially with respect to running times the *Euclideanbase* and the *gridbase* approach seem to be proper methods for the base-node generation for the tree heuristic.

Number of base-nodes Outside of a partition class P_s which contains the starting node the tree heuristic searches for the shortest path to some node t only on the edges of some shortest path trees for the so called basedodes of P_s. The more base-nodes are chosen the better the paths found by the heuristic are expected to be. The paths would always be exact if all nodes of P_s are chosen as base-nodes but for this an all-pairs-shortest-paths algorithm would have to be performed which is computational by far to expensive for a dynamic application.

Therefore, the number of base-nodes has to be chosen in such a way that the searchgraph can be computed reasonably fast which depends mainly on the time to calculate the shortest path trees for the base-nodes. At the same time the shortest path trees of the base-nodes should capture as much structure of the shortest paths of the partition class in order to keep the fraction of error paths low. To determine a value b for the number of base-nodes which fulfills these assumptions we tested the quality of the solutions of the tree heuristic for values $b = 1, 3, 5$ and $b = 10$ using the Dijkstra 8-way partition with the *Euclideanbase* approach for base-node generation (see section 3.4.2). Table 3.10 shows the characteristics of the searchgraph for the four values of b. The number of base-nodes does not affect the partition thus the number of components and size of the classes with respect to nodes is the same for all values of b.

The sizes of the searchgraphs differ more if b is increased. On the other hand $|E^+|$ decreases. The searchgraphs contain more edges for greater b and the number of edges being present in all searchgraphs increases.

DIJK	# Bases			
	1	3	5	10
Edgedev	1.3%	10.7%	11.1%	12.4%
Avg. Edges	50.8%	55.1%	55.9%	56.5%
$\|E^+\|$	36.6	13.0	10.5	8.4
All-edges	33.2%	37.6%	38.3%	38.9%

Table 3.10 Searchgraph characteristics for different numbers of base-nodes for the 8-way Dijkstra partition.

To analyze the effect of b on the quality of solutions of the tree heuristic we first show the mean error with respect to traveling time for a total of 30000 paths in figure 3.31. If only one base-node in the center of each partition class is chosen the mean error is up to 7% for short paths and still around 1% for longer paths. For the other three values of b the average error is significantly lower. For paths longer than 40 minutes it is below 0.5%, for shorter paths the error is smaller if more base-nodes are chosen.

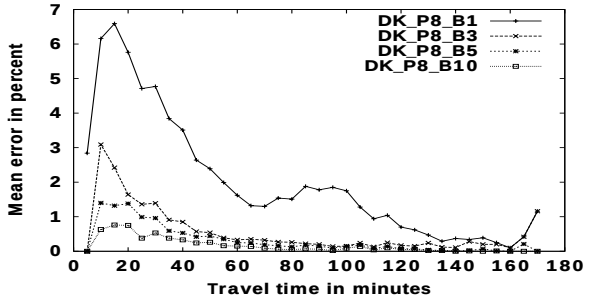


Figure 3.31 Mean error of traveling times for a different number of base-nodes for the Dijkstra 8-way partition.

For the fraction of erroneous paths the effect is even more pronounced. Figure 3.32 shows the results. While for $b = 1$ about 20% of the paths are not the exact path, it is around 5% for short paths, tending to zero for $b = 10$. For the maximal relative error $b = 1$ has deviations of more than 50% even for longer paths up to a length of 100 minutes. For the other three values the error is below 20% for paths

longer than 60 minutes and the differences are very small for the three values. The plot is therefore not shown here.

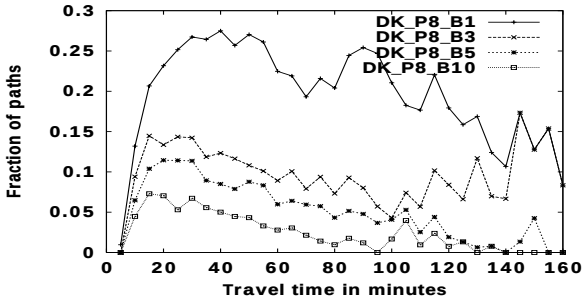


Figure 3.32 Fraction of deficient paths for a different number of base-nodes for the Dijkstra 8-way partition.

In summary the results show that the quality of the solutions found by the tree heuristic can be improved significantly by choosing more than one base-node. A value of $b = 3$ already makes the heuristic well applicable but since even a choice of ten base-nodes does not lead to major computational expenses for the calculation of the searchgraph a number of at least five should be chosen in order to get acceptable solutions.

For a more exact analysis the shape of the partition classes should be taken into account since this affects the quality of the paths significantly as shown in section 3.5.2.5. If the partition classes are almost convex than a smaller number of base-nodes should be sufficient. In figure 3.33 and figure 3.34 we show two partition classes of the Dijkstra 8-way partition of very different shape. Class 1 is almost convex with a few bulges while class 6 has a very irregular shape.

The fractions of deficient paths for the two classes and a different number of base-nodes are shown in figure 3.35 and figure 3.36.

For partition class 1 the fraction of paths with a high error is below 10% for all traveling times even for $b = 1$. If more base-nodes are chosen the fraction decreases but the effect is not very pronounced. For class 6 with its irregular shape the fraction of deficient paths is very high being at most 50% for $b = 1$. If at least three base-nodes are chosen it decreases below 10% and the differences between $b = 3$ and $b = 10$ are small. Also the fractions of paths are of the same magnitude for the two classes when b is at least three. Thus, a greater number of base-nodes

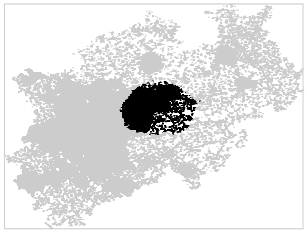


Figure 3.33 Class 1 of the Dijkstra 8-way partition.

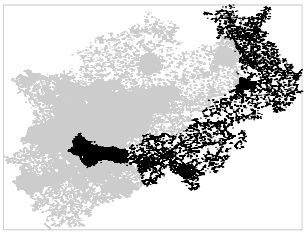


Figure 3.34 Class 6 of the Dijkstra 8-way partition.

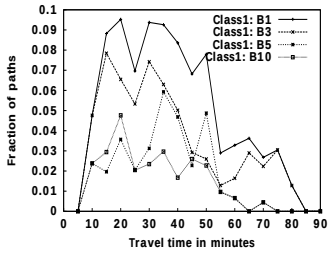


Figure 3.35 Fraction of deficient paths for partition class 1 of the Dijkstra 8-way partition for a different number of base-nodes.

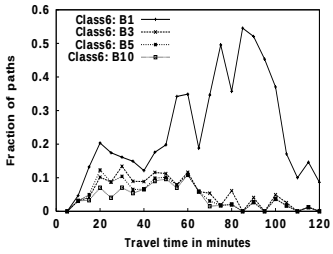


Figure 3.36 Fraction of deficient paths for partition class 6 of the Dijkstra 8-way partition for a different number of base-nodes.

allows the tree heuristic to find the exact shortest path even for partition classes of very irregular shape.

3.5.2.7 Analysis of maximum error paths

In some cases paths found by the tree heuristic show high relative errors for short range paths. For longer paths these errors decrease below 15%. In either case the absolute travel time errors in minutes are about the same size regardless of the path length[8]. In figure 3.37 and figure 3.38 we show two examples of maximal error paths for an 8-way Dijkstra partitioning of NRW. The shortest path in figure 3.37 is a very short path of travel time about four minutes. As can be seen the searchgraph for the heuristic misses some short links close to the target node of the path, resulting in a suggested path of a length of about nine minutes.

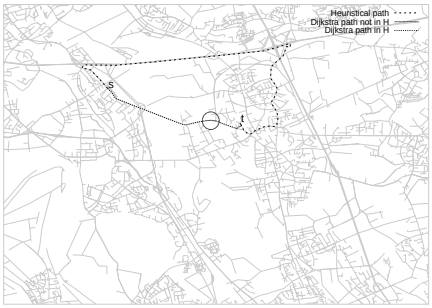


Figure 3.37 An *s*-*t* path with high relative error in NRW: The edges of the shortest path that are not present in the searchgraph are near *t*, shown in solid line in the circle.

For the path in figure 3.38 the searchgraph misses some edges very close to the starting node, causing the heuristic to find a path which starts out in the wrong direction. Although the shortest path has a length of about only 12 minutes, the suggested path by the heuristic takes 25 minutes. The absence of these particular edges in the searchgraph for the heuristic also leads to the maximal errors for longer paths, e.g. there is a path of a length of about 70 minutes which also uses these edges. The path which is suggested by the heuristic for this pair of nodes takes the same deviation as the short path and has a length of 83 minutes.

[8]For the NRW network with the speed model we used the maximal errors were in the range between 10 and 15 minutes.

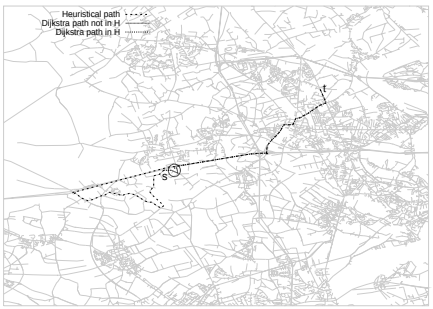


Figure 3.38 An *s*-*t* path with high relative error in NRW: The edges of the shortest path that are not present in the searchgraph are near *s*, shown in solid line in the circle.

In both examples the searchgraph for the heuristic misses only a small number of edges of the actual shortest path. The main difference of the two examples is the location of the missing edge on the shortest path. From the idea of the tree heuristic we expect that most deficient paths miss some edges close to the border of the partition class of the starting node. In order to check if this expectation holds we analyzed a total of 180 maximum error paths of different length and partition classes for an 8-way Dijkstra partitioning with respect to the number of missing edges in the searchgraph and distance of missing edges to the partition class of the starting node in figure 3.39 and figure 3.40.

Figure 3.39 shows that the average number of missing edges of the shortest path in the searchgraph of the tree heuristic is small lying below 15. The number is not much influenced by the length of the path leading to a decreasing fraction of missing edges for longer paths. In figure 3.40 we show the minimal and maximal Euclidean distance between a missing edge and the last node of the partition class of the starting node on the shortest path averaged over all paths in each distance bin. The plot shows that the first edge on the shortest path missing in the searchgraph can be expected to be not further than 20 kilometers away from the partition class. With one exception the maximal distance between missing edges on the shortest path and the partition class is only slightly higher than the minimal distance. The results therefore confirm our expectation that the searchgraphs of the tree heuristic miss mostly edges of shortest paths that lie just outside the partition classes while all edges around the target nodes are present.

For the 180 paths of maximal error there were only 522 edges of the shortest

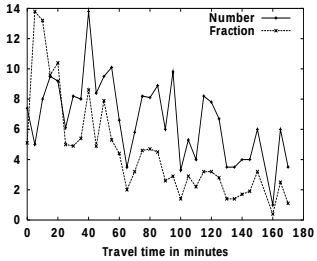


Figure 3.39 Number and fraction (in %) of missing edges of the searchgraph in paths of maximal error.

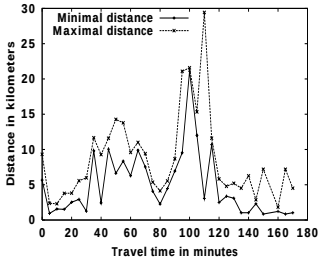


Figure 3.40 Distance between missing edges and the partition class of the starting node for paths of maximal error.

paths that were missing in all searchgraphs. Most of these edges were used not only by a single shortest path and lead to the same absolute deviation for all those suggested paths for the tree heuristic. This is also the reason why the maximal absolute travel time error in minutes is in most cases the same regardless of the path length for the tree heuristic.

In order to see if these missing edges are of significant importance for shortest paths for this specific partitioning we added all 522 missing edges to the set E^* for the searchgraphs of the 8-way Dijkstra partitioning and calculated paths for a different set of starting and target nodes. These paths were compared to the ones found by the tree heuristic if these edges were not added. Additionally we increased the number of bases in each class from five to 15 and calculated the shortest paths for the set of starting and target nodes. This lead to searchgraphs that had about 1% more edges than the searchgraphs for five bases per class.

In figure 3.41 we show the relative error as a function of the travel time for the three approaches. The addition of the 522 edges slightly improves the expected error while increasing the number of bases to 15 shows a significant effect with a relative error below 0.2% for paths longer than 40 minutes. A similar effect can be observed for the fraction of deficient paths (see figure 3.42) for paths up to a length of 80 minutes. For longer paths both approaches of adding edges lead to a decrease of deficient paths. The two plots suggest that by adding only a few edges of importance to the searchgraphs the tree heuristic more likely finds the exact path. On the other hand the quality of a suggested path that is not the exact shortest path is not improved by adding these edges. In order to decrease the expected maximal relative error of the paths the number of bases should be increased. This conclu-

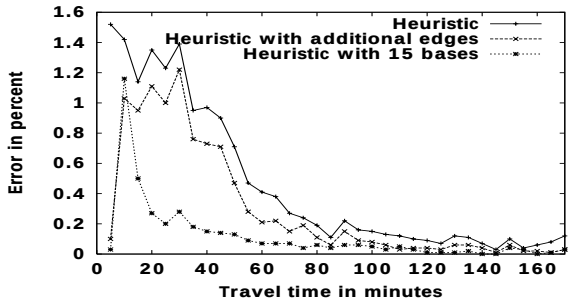


Figure 3.41 Comparison of the relative error if specific edges are added to the searchgraphs or the number of bases increases in an 8-way Dijkstra partitioning.

sion is supported by figure 3.43 where the fraction of paths with an error greater than 10% is shown. The fraction of these paths does almost not improve if the 522 edges are added. In contrast an increase on the number of bases leads to a significant smaller number of high error paths. This is especially true for partition classes of high maximal relative error, while classes of small relative error show some improvement for both approaches of adding edges. This suggests that the searchgraphs of *bad* classes miss a variety of edges while the number of missing edges for *good* classes is much smaller. Therefore, for the latter classes there is some chance of identifying edges of high importance for shortest paths starting from this class and thus improving the quality of the paths found by the tree heuristic by adding these edges, while for *bad* classes only an increase of bases leads to a significant improvement.

3.5.2.8 A theoretical bound for the maximal error

From the partitioning of the road network into k classes we can derive a theoretical bound for the maximal travel time error of paths found by the tree heuristic in comparison to the actual shortest path. For this, let s resp. t be the starting resp. target node of such a path, where s lies in the partition class V_i and $t \in V$. Let $SP(s,t)$ be the shortest path between the two nodes in the network G and $P_{th}(s,t)$ the path found by the tree heuristic in the searchgraph H_i between s and t. Finally let b_c be the center base-node for partition class V_i. The searchgraph H_i contains the shortest path tree T_{b_c} of b_c in G. Figure 3.44 outlines the notation for an idealized example,

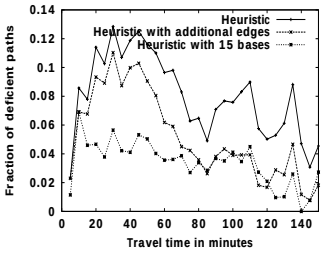


Figure 3.42 Comparison of the fraction of deficient paths if specific edges are added to the searchgraphs or the number of bases increases in an 8-way Dijkstra partitioning.

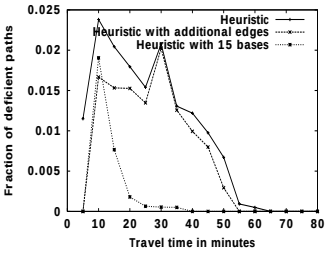


Figure 3.43 Comparison of the fraction of paths with error greater than 10% if specific edges are added to the searchgraphs or the number of bases increases in an 8-way Dijkstra partitioning.

where no paths intersect for a better presentation.

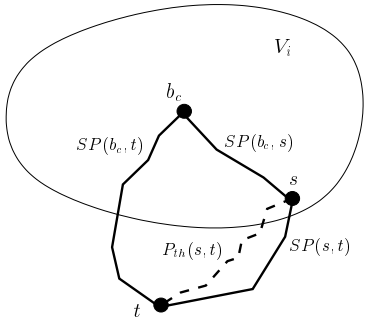


Figure 3.44 Example graph for the bound on the maximal travel time error.

In order to make the searchgraph H_i closed under shortest path calculations, H_i includes those parts of the shortest path tree on the reversed edge set for the center base-node b_c of V_i, that belong to nodes of V_i, i.e. H_i includes all shortest paths from some node $v \in V_i$ to b_c. Since H_i also includes the shortest path $SP(b_c, t)$

from b_c to the target node t, the path $P_{th}(s,t)$ found by the tree heuristic will be no longer than the concatenation of the shortest paths from s to b_c and from b_c to t in H_i. This gives the following inequality:

$$P_{th}(s,t) \leq SP(s,b_c) + SP(b_c,t) \tag{3.1}$$

The shortest path from b_c to t is not longer than the concatenation of the shortest paths from b_c to s and from s to t giving the inequality

$$SP(b_c,t) \leq SP(b_c,s) + SP(s,t) \tag{3.2}$$

Putting these two inequalities together, we get for the maximal travel time error as difference between the two paths:

$$\begin{aligned} P_{th}(s,t) - SP(s,t) &\leq SP(s,b_c) + SP(b_c,t) - SP(s,t) \\ &\leq SP(s,b_c) + SP(b_c,s) + SP(s,t) - SP(s,t) \\ &= SP(s,b_c) + SP(b_c,s) \end{aligned}$$

The derived bound will in general be much greater than the actual errors encountered since the two nodes s and b_c can be very far apart while the suggested path of the heuristic does not need to go all the way back to the center node of the partition class. If the travel times on the links are derived from average speeds given for each road type then we can assume that the travel times of the two shortest paths $SP(s,b_c)$ and $SP(b_c,s)$ are about the same since most of the roads in the network are bidirectional. Thus, the bound for the maximal travel time error reduces to $2 \cdot SP(s,b_c)$.

For an 8-way Dijkstra partitioning we calculated the derived error bound for each partition class and compared it with the actual maximal absolute travel time errors observed in our shortest path calculations. The results are given in table 3.11, showing that the observed errors are clearly below the theoretical bound.

3.5.2.9 Comparison with the HISPA heuristic

For a better evaluation of the quality of the paths found by the tree heuristic we compared the paths with those found by the HISPA heuristic (see section 2.6.1 for a description of the heuristic). A comparison between the tree heuristic and the A^*-algorithm with overdo is given at the end of chapter 4. Using the same set of s-t paths allowed a direct comparison of a specific partition for the tree heuristic with the HISPA heuristic. Observe that the partition has no influence on the paths found by the latter.

The HISPA heuristic first searches a shortest path from either endnode to nodes of the highest hierarchy level in a circle with radius r around the endodes. For each

Class	Theoretical error in min	Observed error in min
0	96	4
1	88	10
2	170	5
3	64	3
4	78	9
5	140	12
6	346	19
7	110	14

Table 3.11 Comparison of theoretical error bound and observed error for an 8-way Dijkstra partitioning.

path found an edge of appropriate length is added to the highest hierarchy level and an *s*-*t* path is computed on the graph of the highest hierarchy level. If no node of the highest hierarchy level is found in the circle the algorithm searches for the first one encountered. In most cases this will suffice to find a path between *s* and *t* unless the only nodes found on the highest hierarchy level for *s* and *t* are not connected by a path. This can be avoided by adding edges between nodes representing a highway stretch for both directions.

We tested the HISPA heuristic for a chosen radius of 10, 20, 30 and 40 minutes. For radius $r = 10$ and $r = 20$ there was a small number of paths that were not found. The running times for $r = 30$ and $r = 40$ minutes were slow outperforming Dijkstra's algorithm only for paths longer than 100 minutes. On the other hand for radius $r = 10$ the average error and the maximal error of the found paths were higher than for the other values of r by a factor of up to five.

Therefore, for the comparison with the tree heuristic we chose $r = 20$ minutes as radius for the HISPA heuristic since for this value the running times were reasonably fast and the errors made by the heuristic were comparable to those of higher radius.

Figure 3.45 shows the average error of the paths found by the HISPA heuristic and those of the tree heuristic using a Dijkstra k-way partition for $k = 8$ and $k = 16$. In figure 3.46 we show this error measure for the HISPA heuristic in comparison with the tree heuristic using a METIS k-way partition with weight function W1 for $k = 8$ and $k = 12$.

For the set of paths of the 8-way Dijkstra partition the HISPA heuristic gives very good results with an average error below 0.5% for all path lengths. The tree heuristic has the same quality only for paths longer than 40 minutes. In contrast

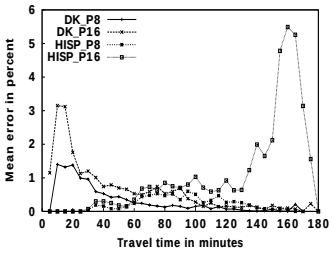


Figure 3.45 Comparison of the mean error of paths between the HISPA heuristic and the tree heuristic using a Dijkstra partition for $k = 8$ and $k = 16$.

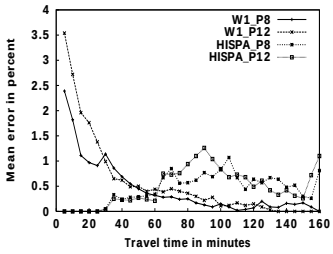


Figure 3.46 Comparison of the mean error of paths between the HISPA heuristic and the tree heuristic using a METIS partition with weight function W1 for $k = 8$ and $k = 12$.

for the set of paths of the 16-way Dijkstra partition the HISPA heuristic has a large average error of up to 5% for long paths while the error for the tree heuristic is below 0.5%.

For the set of paths of the METIS partition with weight function W1 the tree heuristic has an average error between 1% and 3.5% for short paths while the error of the HISPA heuristic is below 0.5%. For paths longer than 60 minutes the error is below 0.5% for the tree heuristic and around 1% for the HISPA heuristic.

For the error measure of the maximal relative traveling time error the data for the set of paths of the Dijkstra partition and METIS partition with weight function W1 is shown in figure 3.47 and figure 3.48.

For the tree heuristic using the Dijkstra partition the maximal relative error is very high for very short paths but below 20% for paths longer than 60 minutes. In contrast the HISPA heuristic finds paths of relative error above 20% even for long paths and of more than 50% for midrange paths. The maximal relative error for the HISPA heuristic is even worse for the set of paths of the METIS partition for paths longer than 30 minutes, where the greatest deviations are observed. The error decreases for longer paths but stays above the maximum of the tree heuristic. On the other hand the tree heuristic makes a great maximal error for very short paths, where the HISPA heuristic finds the exact paths because of the chosen radius.

The fraction of deficient paths in figure 3.49 and figure 3.50 shows that the tree

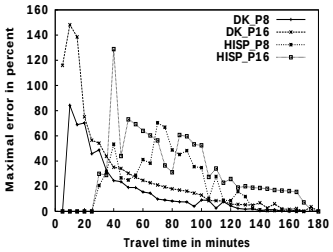


Figure 3.47 Comparison of the maximal relative error of paths between the HISPA heuristic and the tree heuristic using a Dijkstra partition for $k = 8$ and $k = 16$.

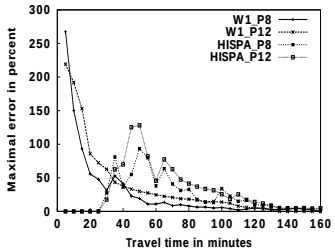


Figure 3.48 Comparison of the maximal relative error of paths between the HISPA heuristic and the tree heuristic using a METIS partition with weight function W1 for $k = 8$ and $k = 12$.

heuristic finds about 85% of the exact shortest paths[9]. In contrast the HISPA heuristic finds all shortest paths up to a length of 20 minutes. For longer paths the fraction of erroneous paths increases, intersecting the graph of the tree heuristic between 60 and 80 minutes. It even reaches a value of one for the set of paths of the METIS partition.

This inverse behaviour of the tree heuristic and the HISPA heuristic can also be observed for the fraction of paths with a relative error greater than 10% (see figure 3.51 and figure 3.52). For short paths the tree heuristic has between 15% and 25% of high error paths and almost none for longer paths. In contrast this fraction is about 5% for the HISPA heuristic.

Evaluating the quality of the solutions found by the two heuristics the tree heuristic performs worse for very short paths where HISPA gives the exact shortest paths. As soon as the paths are longer than the radius used by HISPA the relative error and the fraction of shortest paths with an error greater than 10% are comparable for both heuristics. While the results for these two error measures decrease for the tree heuristic for longer paths they increase for HISPA. The maximal relative error of paths is much higher for HISPA than for the tree heuristic and with values above 50% very high for all distances greater than 20 minutes. Thus, the tree heuristic gives much better results if the two endnodes of the path are not too

[9] We only show the curves for the 8-way partitions, because the ones for the 16-way partitions are almost identical.

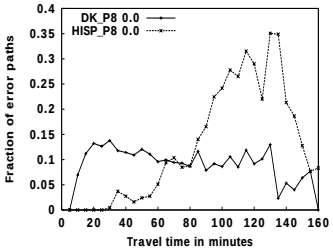


Figure 3.49 Comparison of the fraction of deficient paths between the HISPA heuristic and the tree heuristic using a Dijkstra partition for $k = 8$.

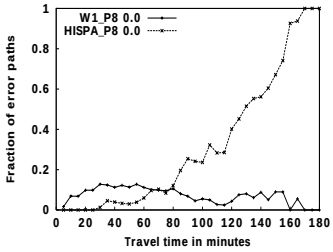


Figure 3.50 Comparison of the fraction of deficient paths between the HISPA heuristic and the tree heuristic using a METIS partition with weight function W1 for $k = 8$.

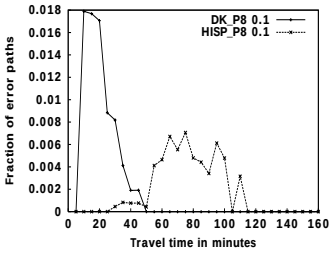


Figure 3.51 Comparison of the fraction of paths with relative error at least 10% between the HISPA heuristic and the tree heuristic using a Dijkstra partition for $k = 8$.

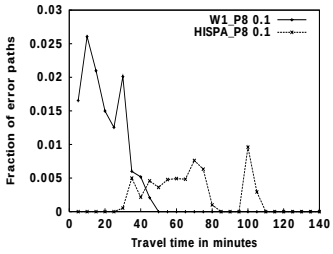


Figure 3.52 Comparison of the fraction of paths with relative error at least 10% between the HISPA heuristic and the tree heuristic using a METIS partition with weight function W1 for $k = 8$.

close to each other in which case the value of the relative error made is of greater importance.

3.5.2.10 Summary for the quality of solutions

For most of the studied partitions of the NRW network the tree heuristic finds paths between two nodes *s* and *t* that are either exact or at least within an error range of a few percent. For paths longer than 40 minutes the expected error for most partitions is below 1% percent. The relative error for shorter paths is higher but still lies below 5% for good partitions. The fraction of paths found by the tree heuristic that have an error greater than 10% is well below 1% for paths longer than 40 minutes and below 5% for shorter paths for the good partitions. However, the maximal relative error observed can be very high, i.e. higher than 100% for very short paths. It is below 15% for paths longer than 40 minutes for good partitions.

The HISPA heuristic with a radius of 20 minutes shows a higher maximal relative error of up to 50% for paths of length at least 20 minutes. For short paths the HISPA heuristic is exact and therefore outperforms the tree heuristic which takes its highest relative errors for these paths. For paths just longer than the radius of the HISPA heuristic the average error and the fraction of paths with high error are comparable for both heuristics, for even longer paths the tree heuristic shows a better performance. The main difference is a decrease of these quantitites for the tree heuristic and an increase for the HISPA heuristic if paths get longer. Another difference is the fraction of error paths which is about 10% to 15% for the tree heuristic regardless of path length and increases from 0% to 100% for the HISPA heuristic.

All three partitioning methods presented find partitions showing a very good solution quality. For almost all methods the 8-way partitions had the least average error and fraction of paths with high error. This quality is very much affected by the number of connected components of the partition classes. For the Dijkstra and most of the treegraph partitions all classes are connected[10] leading to solutions with an expected error below 0.5% for paths longer than 40 minutes and a fraction of paths with a high error tending to zero fast for the 8-way partitions.

The partitions with METIS that lead to connected partition classes show a comparable solution quality. Weight functions leading to disconnected components in the partition classes are of worse solution quality especially for short paths. For longer paths the expected error is still below 2%. Although the partitioning method should be considered in a practical application the tree heuristic seems to be very robust against changes of the partitioning method.

[10] For those treegraph partitions that generate unconnected subclasses, the number of connected components is very small.

3.5.3 Runtime performance

We use different measures to analyze the runtime performance of the heuristic for shortest path computations: The actual runtime of the algorithm, the number of scanned nodes and the ratio of scanned nodes to the length of the path. These quantities were calculated for the tree heuristic, Dijkstra's algorithm, a bidirectional variant of Dijkstra's algorithm, the A^*-algorithm and the HISPA heuristic (see sections 2.4.3 and 2.6 for details about these algorithms). Since the experimental tests were run on a multiuser system the actual runtime of the algorithms is not a truly sophisticated measure. On the other hand for all our tests the running times were fairly consistent usually with relative errors below 4% for identical shortest path calculations. Therefore, we include this measure in our runtime analysis.

In the next subsection we analyze the runtime performance of the tree heuristic for different partitions followed by a comparison with the performance of the other heuristics mentioned.

3.5.3.1 Runtime of the tree heuristic

The runtime of the tree heuristic depends mainly on the number of edges that are scanned by the backward Dijkstra algorithm. Keeping the number k of partition classes fixed the size of the searchgraph with respect to the edges depends mainly on the number of base-nodes for which a shortest path tree is computed. Although the size of the edge sets differs for the described methods of partitioning, the differences are so small that they have almost no measurable effect on the running time of the algorithm.

The choice of the partition method for fixed k did not affect the runtime performance of the tree heuristic in such a way as to allow a complete qualitative rating of the different methods. Only the METIS partitions with weight functions W3 and W4 lead to searchgraphs with slightly more edges than the other partition methods (compare the tables with the partition characteristics in section 3.5.2.2). Both of these weight functions tend to place edges of small weight in the edge cut leading to a great number of edges in the cut. Since edges between two partition classes will be part of both searchgraphs of the adjoining classes a great number of edges will belong to two searchgraphs, thereby increasing the size of the graphs. The running times of the path calculations for these partitions were therefore not as good as for the other partitions. Together with the worse quality of the suggested paths these two weight functions do not seem to be a recommended choice for an application of the tree heuristic. All the other partition methods generated searchgraphs of very similar size with respect to the edges for fixed k.

Runtime performance for different numbers of base-nodes As shown in table 3.10 in section 3.5.2.6 the number of edges in the searchgraph for the 8-way

Dijkstra partition differs for the four values of b for the number of base-nodes by more than 5% due to the different number of shortest path trees which are part of the searchgraph. The effect on the running time of the tree heuristic is shown in figure 3.53 for the number of nodes that are scanned during the shortest path calculation. If only one base-node is used for each partition class the least nodes are scanned. This number is almost the same for $b = 5$ and $b = 10$, being slightly higher than for $b = 1$.

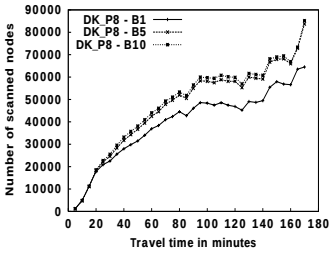


Figure 3.53 Comparison of the number of scanned nodes for a different number of base-nodes using a Dijkstra 8-way partition.

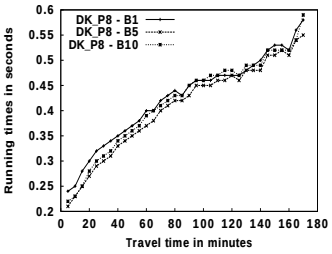


Figure 3.54 Comparison of the running times for a different number of base-nodes using a Dijkstra 8-way partition.

The effect on the actual running time of the algorithm is very small being in the range of a few microseconds on the described system as shown in figure 3.54. Weighting the small running time advantage of one base-node per partition class against the significantly better quality of the found solutions when using for example ten base-nodes per class leads to the conclusion that a choice of five base-nodes gives a good compromise between overall runtime performance and solution quality.

Runtime performance for different values of k If the original graph is partitioned into a higher number of classes, the different classes have fewer nodes and therefore the searchgraph has fewer edges since the portion of edges having at least one endnode in the class is smaller. Therefore, it is expected that the running time of the tree heuristic decreases when the number of classes k is increased.

Figure 3.55 shows the number of scanned nodes during the shortest path calculation for an 8-way, 12-way and 16-way Dijkstra partition. As expected the algorithm scans most of the nodes for the 8-way partition having searchgraphs with

more edges. For paths longer than 60 minutes the 16-way partition scans about half as many nodes as the 8-way partition. The effect on the actual running times for the three partitions is shown in figure 3.56. While the running times for the 12-way and 16-way partition do not differ much, the 8-way partition is about 0.05 seconds slower for longer paths.

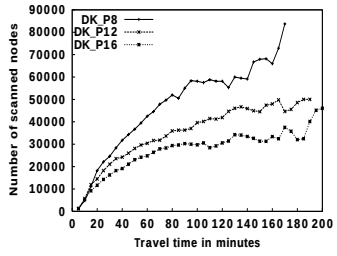


Figure 3.55 Comparison of the number of scanned nodes for different Dijkstra *k*-way partitionings.

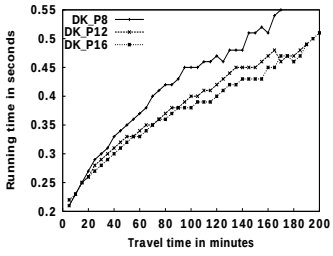


Figure 3.56 Comparison of the running times for different Dijkstra *k*-way partitionings.

Another interesting quantity for measuring the runtime performance of a shortest path search is the number of nodes that are scanned by the algorithm in relation to the number of nodes that lie on the shortest path. While the number of scanned nodes shown in figure 3.55 is an absolute value that in most cases increases with the length of the path, the ratio of scanned nodes to path nodes is a relative measure and therefore allows to examine if an algorithm scales well. Figure3.57 shows this ratio for the three Dijkstra partitions. The 16-way partition takes a maximum ratio of about 150 for paths of length 70 minutes. For longer paths the ratio decreases. The ratio for the 12-way partition is slightly above 150, while the 8-way partition takes values up to 250. The sharp increase for the longest paths for each partitioning is due to an unsufficient number of data points. At least for the 12-way and the 16-way partition the ratio decreases for longer paths showing that the tree heuristic does scale well.

As the plots show the reduced size of the searchgraphs for partitions into more classes leads to faster shortest path calculations. Since the actual runtime differences between the various *k*-way partitions are below 0.1 seconds in our tests, an optimal value for *k* should be better chosen from the viewpoint of solution quality where the 8-way partitions showed in general a slightly better performance.

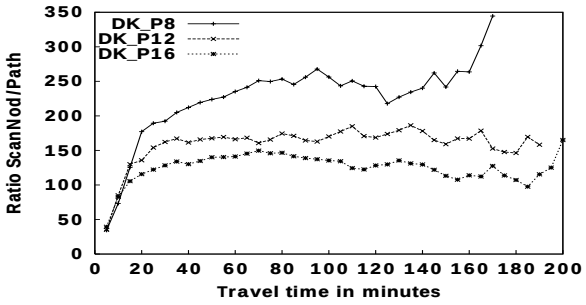


Figure 3.57 Comparison of the ratio of scanned nodes to path nodes for different Dijkstra *k*-way partitionings.

3.5.3.2 Runtime comparison with other shortest path algorithms

In this section we compare the runtime performance of the tree heuristic with that of Dijkstra's algorithm, a bidirectional Dijkstra, the A^*-algorithm and the HISPA heuristic with radius 20 minutes (see section 2.4.3 and section 2.6 for details about these algorithms).

We have chosen the 8-way Dijkstra partition for the tree heuristic and calculated the shortest paths for the other algorithms mentioned on the same set of node pairs. As shown in section 3.5.3.1 partitions for a greater k lead to a faster algorithm, therefore the differences between the tree heuristic and the other algorithms are even more pronounced for $k \geq 8$.

Figure 3.58 shows the plot for the actual run time of the different algorithms. The running times for the tree heuristic increase slowly with the path length, not exceeding 0.6 seconds even for long paths. For paths longer than 30 minutes the tree heuristic outperforms all other algorithms at least by a factor of two. The HISPA heuristic is the worst for paths that are a bit longer than the chosen radius for the search on the whole graph. For these paths the overhead of performing three separate Dijkstra-like searches overhelms the effect of a sparse graph of the highest hierarchy level. For longer paths the running time is second best. The A^*-algorithm has running times closest to the tree heuristic for paths up to a length of 130 minutes. For longer paths the calculation of the future costs starts to slow down the algorithm since the number of scanned nodes gets close to that of Dijkstra's algorithm. For very long paths the A^*-algorithm is the slowest. The bidi-

rectional Dijkstra is the fastest for very short paths and stays below the normal Dijkstra for paths up to a length of about 100 minutes. For longer paths the overhead of performing two searches and finding the shortest path over the cut-edges lead to a slower algorithm than even Dijkstra's algorithm. A similar conclusion about the running time of this variant is stated in [56] for the road network of Portland/Oregon. Dijkstra's algorithm is slower than most of the algorithms for paths of a length up to 80 minutes. For longer paths it is outperformed only by the HISPA heuristic and the tree heuristic.

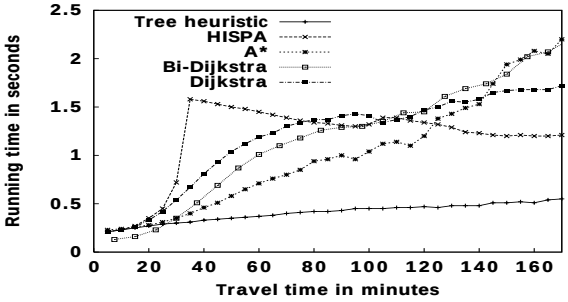


Figure 3.58 Comparison of the running times for different algorithms using a Dijkstra 8-way partition.

Analyzing the shape of the curves for the different algorithms shows that the running time of the tree heuristic increases very gradually. Only the graph of Dijkstra's algorithm is close to convexity while those of the A^*-algorithm and bidirectional Dijkstra show a strong increase for longer paths. The HISPA heuristic has a peak for paths where the two endnodes are just a bit further apart than the given radius. The decrease of the running time for longer paths is due to border effects of the network: The graph of the highest hierarchy level is very sparse compared to the whole graph (about 0.5% of the edges) and therefore has only very little influence on the running time for different paths. Most of the running time is due to the two searches around the endnodes s and t on the whole graph. The denser the network is in the circle with radius r around s and t the longer the search takes. For longer paths there is a higher probability that the randomly chosen endnodes lie in sparse areas of the network leading to a faster running time.

The tree heuristic outperforms Dijkstra's algorithm by a factor of about 3.5, the HISPA heuristic by a factor of two even for longer paths and the A^*-algorithm and the bidirectional Dijkstra by a factor of more than four for longer paths. For the

main portion of midrange paths the tree heuristic outperforms all other algorithms at least by a factor of 2.5.

For an analysis of the runtime performance of the different shortest path algorithms that neglects the used system architecture and all performance overhead resulting from software design and implementation details we study the number of scanned nodes during the shortest path search. Since these numbers differ greatly for the described algorithms we look at the ratio of scanned nodes of Dijkstra's algorithm and the number of scanned nodes for each algorithm. This ratio gives the speedup of each algorithm compared to Dijkstra's algorithm with respect to the search area in the network. Figure 3.59 shows the resulting plot for the 8-way Dijkstra partition.

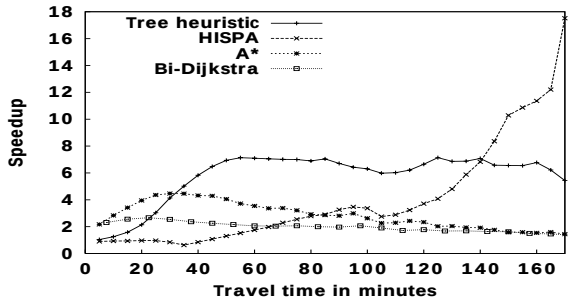


Figure 3.59 Comparison of scanned nodes for different algorithms against Dijkstra's algorithm using a Dijkstra 8-way partition.

The tree heuristic gives a speedup of about seven which means that Dijkstra's algorithm scans seven times more nodes than the tree heuristic. For short paths the ratio is increasing since the effect of the search in the shortest path trees for the tree heuristic evolves. For longer paths the ratio is almost constant since for these paths the additional number of nodes that are scanned is proportional for both algorithms. The ratio for the HISPA heuristic climbs from below one for paths a bit longer than 20 minutes (due to the double application of Dijkstra's algorithm) to a ratio of about four for paths up to a length of 100 minutes. For longer paths the speedup increases rapidly which is caused by the normally sparse areas around the endnodes. If for example the two endnodes lie in rural areas in the East and West of Northrhine-Westphalia the HISPA heuristic searches two sparse circles around the endnodes together with the very thin graph of the highest hierarchy level while Dijkstra's algorithm will expand through the entire very dense Ruhr area.

The A^*-algorithm has the best speedup for short paths where the future costs direct the algorithm towards the endnode. For longer paths the A^*-algorithm scans more than half of the nodes of Dijkstra's algorithm which might result from a too optimistic estimate for the actual travel time which causes the search ellipse to be wider. Together with the additional costs of calculating the future costs the algorithm will therefore in general not lead to a faster algorithm for long paths. The bidirectional Dijkstra has a speedup for the scanned nodes of about two for all paths which shows that the search area indeed is smaller than for Dijkstra's algorithm. But as shown in figure 3.58 the additional overhead of two searches and finding the actual path makes the algorithm in practice even slower than Dijkstra's algorithm. Figure 3.60 shows the different regions of scanned nodes of the algorithms for a path in NRW consisting of 269 nodes.

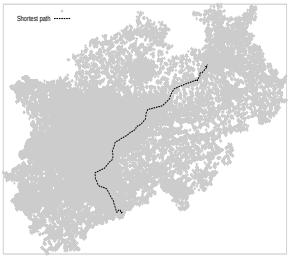
As last performance measure we compare the ratio of scanned nodes to path nodes for the different algorithms in figure 3.61. For the tree heuristic this ratio increases from below 200 to approximately 350. In contrast Dijkstra's algorithm takes values between 1400 and 1900 for paths longer than 50 minutes. That the curve is not monotonously is caused by boundary effects as mentioned earlier. In particular the number of scanned nodes is not only affected by the length of the path (either in minutes or in number of nodes) but also by the density of the area of the two endnodes.

The HISPA heuristic has a peak of 1600 for paths a bit longer than 40 minutes and decreases with path lengths to a ratio below 200. The A^*-algorithm and the bidirectional Dijkstra take similar values for long paths while for midrange paths the A^*-algorithm outperforms the bidirectional Dijkstra by a factor of almost two.

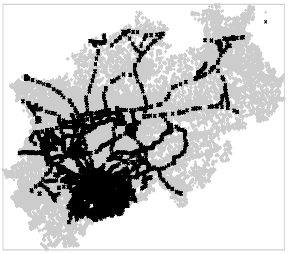
3.5.3.3 Summary of runtime performance

For all three performance measures studied the tree heuristic shows the best results. Compared to Dijkstra's algorithm it gives a speedup for the actual running time by a factor of almost four and for the number of scanned nodes by a factor of about seven on the test network of NRW. For long paths the HISPA heuristic comes closest in the running time being slower by a factor of about two and outperforms the tree heuristic in the number of scanned nodes. Both the A^*-algorithm and the bidirectional Dijkstra are slower by a factor of at least two for all performance measures.

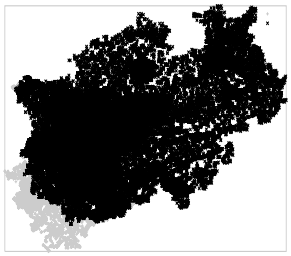
The runtime performance of the tree heuristic evolves homogenously over all path lengths. The actual running time and the number of scanned nodes increase smoothly with the length of the shortest path. In contrast the HISPA heuristic has good runtime performance for long paths but is very slow for short paths. The A^*-algorithm is fast for short paths but takes even longer than Dijkstra's algorithm for long paths. The bidirectional Dijkstra does also not seem to be well suited for long



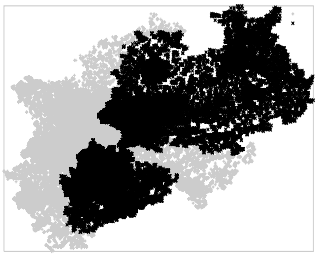
(a) Shortest path with 269 nodes.



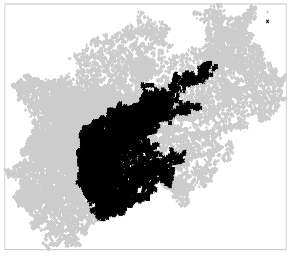
(b) Tree heuristic scanning 70346 nodes.



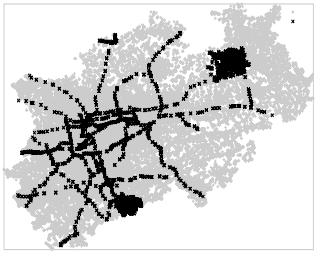
(c) Dijkstra's algorithm scanning 432640 nodes.



(d) Symmetrical Dijkstra scanning 240927 nodes.



(e) A^*-algorithm scanning 192063 nodes.



(f) HISPA heuristic scanning 20745 nodes.

Figure 3.60 Region of scanned nodes for different shortest path algorithms in NRW.

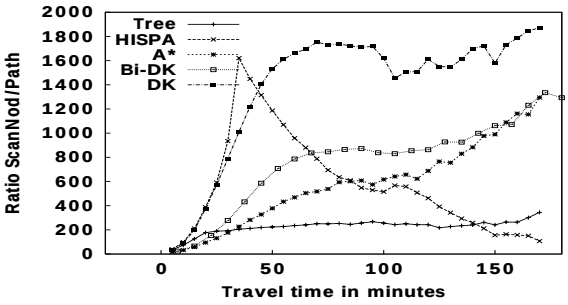


Figure 3.61 Comparison of scanned nodes to path nodes for different algorithms using a Dijkstra 8-way partition.

paths. A k-way partition for higher k will speed up the tree heuristic further with respect to all studied performance measures making the differences to the other algorithms even more pronounced. The effect of the number of base-nodes on the running times of the tree heuristic is small. The used partition method affects the quality of the shortest path solutions in a more significant way than the running times. Therefore, the partition method should better be chosen according to the latter quantity and not according to the runtime performance.

3.5.4 Summary of Experimental Results

The proposed tree heuristic computes shortest paths between two nodes in a road network very efficiently finding about 90% of the exact shortest paths on our test network of NRW. The average error to be expected is small and lies below 1% for most partitions. The quality of the solutions found by the heuristic increases with the length of the paths. Considering the inherent insecurity about expected traveling times in reality due to dynamic changes and data insufficiency errors below 1% make the tree heuristic applicable for practical use.

For the described error measures - average travel time error, maximal relative travel time error and fraction of shortest path with an error greater than a given threshhold - the tree heuristic outperforms the well-known HISPA heuristic for all path lengths but for very short paths.

The tree heuristic gives best results if the partitioning method generates connected partition classes. In our tests the 8-way partitions usually resulted in slightly higher quality of shortest paths than the 12-way and 16-way partitions. However,

for the latter the tree heuristic has a better runtime performance due to the smaller searchgraphs. In our test on the road network of Northrhine-Westphalia between 51 and 57% of the edges of the network were present in the searchgraphs where the k searchgraphs differed usually by not more than 10% of the edges. More than 30% of the edges of the whole graph were found in all searchgraphs for a given partition. That is, more than 50% of the edges of a searchgraph will be part of all searchgraphs for the given partition. Considering this in a practical implementation of the tree heuristic greatly reduces the memory requirements of the algorithm.

Compared to the running times of other shortest path algorithms the tree heuristic with an 8-way partition outperforms Dijkstra's algorithm by a factor of more than three. The bidirectional Dijkstra and the A^*-algorithm are outperformed by a factor of about 2.5 for midrange paths and by a factor of more than four for longer paths. The HISPA heuristic is outperformed by a factor of more than four for midrange paths and by a factor of at least two for long paths.

For the number of scanned nodes the tree heuristic visits about seven times less nodes than Dijkstra's algorithm. For short paths the bidirectional Dijkstra and the A^*-algorithm visit less nodes than the tree heuristic. For longer paths, i.e. 40 minutes for our test network, the tree heuristic is best outperforming the other two by a factor of about three. The speedup of the HISPA heuristic increases over the length of the path, but still visits more nodes than the tree heuristic with the exception of very long paths, i.e. paths of length at least 140 minutes for NRW.

3.6 Summary of results

We evaluated the proposed tree heuristic by conducting extensive experimental tests on the road network of Northrhine-Westphalia. Besides the running time of the heuristic we were mainly interested in the quality of the found solutions since it is not guaranteed that the tree heuristic always finds the exact shortest path between two arbitrary nodes. To this end we tested different approaches for the graph partitioning and base-node generation in relation to structure of the searchgraph and the quality of the suggested path solutions.

For a better classification of the tree heuristic we compared our results with those of some well-known algorithms for finding shortest paths in road networks: Dijkstra's algorithm, a bidirectional variant of Dijkstra's algorithm, the A^*-algorithm which computes exact shortest paths by assigning future costs to the visited nodes and the HISPA heuristic which makes use of the hierarchical structure of the network.

Our main results can be summarized as follows:

- For the test network of NRW the tree heuristic finds about 90% of the exact shortest paths. For good partitionings the fraction of paths with a traveling

time error greater than 10% is below 5% and the quality of the solutions increases with the length of the paths.

- The relative traveling time error usually is below 1% for paths longer than 40 minutes and below 10% for shorter paths. The maximum relative traveling time error can be very high for short paths, but it can be expected to be below 15% for paths longer than 60 minutes. Considering the inherent insecurity about expected traveling times in reality due to dynamic changes and data insufficiency the observed errors below 1% make the tree heuristic well applicable for practical use.
- The tree heuristic gives best results if the partitioning method generates connected partition classes.
- The Dijkstra partitioning always finds connected partition classes and gives shortest path solutions that have a very low average traveling time error.
- The treegraph partitioning also finds paths of high quality. For a smaller gridlength the found paths usually are better. The partition classes are mostly connected or decompose only into a few connected components.
- The METIS partitioning gives very good results if the weight function chosen results in connected partition classes. For some weight functions classes decompose into several hundred components which results in a high fraction of erroneous shortest paths.
- The 8-way partitions usually give better results than the 12-way and 16-way partitions in the test network, but they have a slightly worse runtime performance due to a bigger searchgraph.
- The number of chosen base-nodes affects the quality of the solutions, resulting in more accurate paths if the number increases. If the number of base-nodes is small, i.e. less than ten, the differences in the runtime performance for the searchgraph generation can be neglected.
- For the considered error measures the tree heuristic outperforms the well-known HISPA heuristic for all path lengths but for very short paths. For the tree heuristic the average error decreases with longer paths while for the HISPA heuristic it increases.
- For the running times the tree heuristic with an 8-way partition outperforms Dijkstra's algorithm by a factor of more than three. The bidirectional Dijkstra and the A^*-algorithm are outperformed by a factor of about 2.5 for midrange paths and by a factor of more than four for longer paths. The HISPA heuristic is outperformed by a factor of more than four for midrange paths and by a factor of at least two for long paths.
- The tree heuristic scans about seven times less nodes than Dijkstra's algorithm. For short paths the bidirectional Dijkstra and the A^*-algorithm visit

less nodes than the tree heuristic. For longer paths, i.e. 40 minutes for our test network, the tree heuristic is best outperforming the other two by a factor of about three. The speedup of the HISPA heuristic increases over the length of the path, but still visits more nodes than the tree heuristic with the exception of very long paths, i.e. paths of length at least 140 minutes for NRW.

- The running time of the tree heuristic as a function of the travel time of the paths is almost constant due to the tree-like structure of the searchgraph and an application of the backward Dijkstra. Therefore, the heuristic scales very well.

To verify the above conclusions for the proposed tree heuristic we also tested the heuristic on the road network of the state of California with its 1568089 nodes and 3915521 edges using a 24-way partitioning. Since the network of California is about three times as large as that of NRW the partition classes for $k = 24$ have about the size of those of the 8-way partitionings of NRW. The searchgraphs of the classes were about three times as large as those for NRW.

Figure 3.62 shows the average relative error for a total of 1000 shortest paths for each of the 24 partition classes as function of the travel time, where the errors were averaged in bins of length 20 minutes. For the treegraph partitioning we chose a gridlength of $l = 10000m$. As for the Dijkstra partitioning this resulted in connected partition classes with the exception of one class decomposing into two components. For the METIS partitioning we show the results for weight function W3 (see section 3.5.2.2) which produced partition classes with an average of 2.7 components, where the maximum was 7 components. This contrasts to the very poor partitions with respect to connectivity for this weight function in NRW.

The mean error is below 1% for all three partitionings for paths longer than 200 minutes. The error for shorter paths is not much higher for the Dijkstra and treegraph partitioning. That the two latter partitioning will in general find paths of higher quality is also confirmed in figure 3.63 for the fraction of deficient paths and figure 3.64 for the fraction of paths with an error greater than 10%.

For the runtime performance of the tree heuristic we show in figure3.65 a comparison of Dijkstra's algorithm, the HISPA heuristic with radius 50 minutes and the tree heuristic for a 24-way Dijkstra partitioning. Observe that Dijkstra's algorithm takes about four times longer for long paths than in NRW, while the runtime of the tree heuristic only doubles and is about the same for all path lengths. Here the application of the backward Dijkstra in the tree-like searchgraph outside the partition classes amplifies the speedup even more. This once more confirms that the tree heuristic scales very well. For the number of scanned nodes figure 3.66 shows the ratio of scanned nodes for Dijkstra's algorithm and the two heuristics. Compared to the tree heuristic Dijkstra's algorithm scans up to 25 times more nodes for long paths, while this ratio is up to ten for the HISPA heuristic. In contrast to NRW the

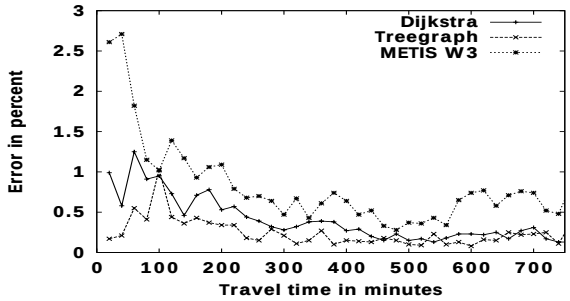


Figure 3.62 Mean error of paths for a 24-way partitioning of California using the Dijkstra, treegraph and Metis approach.

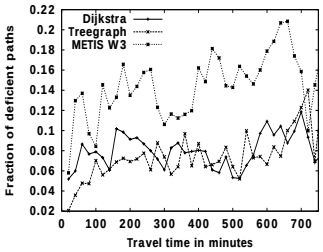


Figure 3.63 Fraction of deficient paths for a 24-way partitioning of California using the Dijkstra, treegraph and Metis approach.

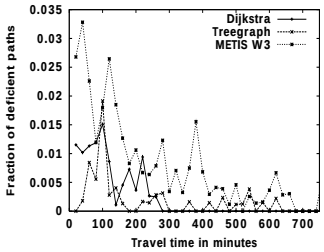


Figure 3.64 Fraction of paths with an error greater than 10% for a 24-way partitioning of California using the Dijkstra, treegraph and Metis approach.

HISPA heuristic therefore scans even more nodes than the tree heuristic for all path lengths. The reasons for this are first the chosen radius of 50 minutes and second the classification of all roads of type greater than zero as highest hierarchy level[11]. This was necessary in order to find at least one path between the starting and target node. But even then the quality of the suggested paths of the HISPA heuristic was very poor with expected errors above 30% even for long paths. Thus, from our tests it seems that the HISPA heuristic is much less applicable to large networks if the hierarchical structure is not so strongly developed.

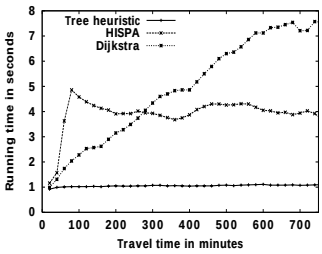


Figure 3.65 Comparison of running times for different algorithms using a 24-way Dijkstra partitioning of California.

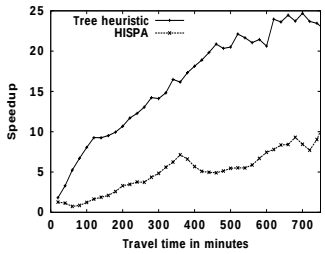


Figure 3.66 Comparison of scanned nodes for different algorithms using a 24-way Dijkstra partitioning of California.

In summary the results for the California network confirm the conclusions for NRW about the quality of paths found by the tree heuristic. While the expected errors of paths are of about the same size for the two networks, the tree heuristic outperforms other shortest paths algorithms in the California network by a significantly higher factor than for NRW.

[11] In NRW only the roads of type four were chosen for the highest hierarchy level.

Chapter
Four

A*-algorithm with Overdo

4.1 Introduction

For networks that are Euclidean it is possible to improve the average case behaviour of Dijkstra's algorithm for the one-to-one shortest path problem by making use of geometrical information inherent in those networks. This approach was first analyzed by Sedgewick and Vitter in [92] and dates back to the work of Hart, Nilsson and Raphel in 1968 [46]. The basic idea is to calculate for each node v scanned during the search of an s-t-path an estimation of the time it takes to reach t from v. The sum of these so called 'future costs' and the travel time of the found path from s to v is used as temporary label in a Dijkstra-like search. Thus, the temporary label for each node v is an estimation of the length of a shortest path from s to t traversing over node v. Note that this approach cannot be generalized to the one-to-all shortest path problem since the target needs to be fixed for the calculation of the future costs.

The classical approach in Euclidean networks for the future costs is to use the time it takes to travel the Euclidean distance between v and t with maximum speed. In this case this so called A^*-algorithm will always find the exact shortest path between two nodes s and t. To see this, observe that in each step of the algorithm, where a node v is scanned, the path $P(s, v)$ by which v is reached is a shortest path from s to v. This follows from the triangle inequalitiy holding in Euclidean networks[1]. Therefore, when the algorithm scans node t a shortest path from s to t has

[1] For a short proof assume that v is reached along a path $P(s, v)$ but there is a shorter path $P'(s, v)$. The temporary label of v is $l(P(s, v)) + fc(v, t)$ with $fc(v, t)$ being the future costs for v. Let w be the last node on $P'(s, v)$ which has been scanned by the algorithm and x the next node after w along $P'(s, v)$. Now $x \neq v$ because otherwise v has a wrong temporary label. Since v is scanned before x we have $l(P'(s, x)) + fc(x, t) \geq l(P(s, v)) + fc(v, t)$. On the other hand, the triangle inequality gives $fc(x, t) \leq fc(x, v) + fc(v, t)$ and $fc(x, v) \leq l(P'(x, v))$. Combining the last two inequalties gives $l(P'(s, x)) + fc(x, t) \leq l(P'(s, x)) + fc(x, v) + fc(v, t) \leq l(P'(s, x)) + l(P'(x, v)) + fc(v, t) = l(P'(s, v)) + fc(v, t) < l(P(s, v)) + fc(v, t)$. Thus, the temporary label of x is smaller than that of v

been found. The A^*-algorithm will on the average have a better run time performance by having a more elliptic search area of expanded nodes.

Road networks using travel times as weights for the edges are in general not pure Euclidean for a number of reasons: Different speed classes give roads of equal geometric length not necessarily the same travel time, in a dynamic setting congested roads can cause great delays on 'short' links and finally networks coming from real life applications sometimes include links that are shorter than the Euclidean distance between the two endpoints due to artificial conventions or data errors (cf. [56]). Therefore, the triangle inequality will not always be fulfilled and road networks are only nearly Euclidean. An application of the A^*-algorithm to nearly Euclidean networks will in most cases[2] still find the exact shortest path but might expand nodes more than once leading to a significant performance slow down.

In a variant of the A^*-algorithm the Euclidean distance between a node v and the endpoint t of the shortest path is weighted with an overdo factor $f_{ov} \geq 1$. This idea was hinted at in [92] and analyzed in [56] for the road network of the Dallas/Fort Worth area in Texas/US using data from a traffic microsimulation of the area. The geometric idea of using such an overdo factor is to narrow the search area of expanded nodes to smaller ellipses if f_{ov} increases. On the other hand, by weighting the Euclidean distance, the future costs are not necessarily a lower bound for the travel time from v to t. Therefore, the A^*-algorithm with overdo factor finds a path between s and t that might not be the exact shortest path. This approximation is expected to get worse the higher the overdo factor is chosen since then the search area of expanded nodes gets smaller and fewer nodes are examined. Figure 4.1 gives a pseudo-code description of the modified A^*-algorithm where $d(v)$ gives the travel time of the path from s to v found by the algorithm, $label(v)$ is the key of each node in the priority queue and $S(v)$ is the status of the node during the algorithm. Additionally, $eu(w,t)$ gives the Euclidean distance between nodes w and t, $c(e)$ are the costs of an edge e and v_{max} is the maximal possible speed on any link in the network.

For the road network of Dallas/Fort Worth with 9863 nodes and 14750 edges Jacob et al. in [56] analyzed the modified A^*-algorithm for overdo factors between 1 and 99. This improved the running times by almost a factor of 40 for overdo parameter 99 while still giving solutions of very good quality with a maximum error no worse than 16%. For overdo parameter 99 almost all paths were erroneous but only around 15% had a relative error of 5% or more. In the analysis of their results they suggested an 'optimal' value for the overdo factor which significantly improves the running time while still giving solution paths of good quality. In order to find

giving the desired contradiction.

[2]There might be the rare case that the triangle inequality is violated for target node t causing the algorithm to halt too early. In our tests of the A^*-algorithm this phenomenon never occurred.

```
procedure A* with overdo
    begin
      foreach v ∈ N
      begin
        d(v) := ∞;
        label(v) := ∞;
        S(v) := unreached;
      end
      d(s) := 0;
      label(s) := 0;
      S(s) := labeled;
      Repeat
      begin
        v = min_{u∈N}{ label(u) : S(u) = labeled};
        if v = t then break;
        foreach (v, w) ∈ E
        begin
          if d(v) + c(e) < d(w)
          begin
            d(w) := d(v) + c(e);
            label(w) := d(w) + f_ov * eu(w, t)/v_max;
            S(w) := labeled;
          end
          S(v) := scanned;
        end

      end

    end
```

Figure 4.1 Modified A^*-algorithm.

such an optimal value we analyze the trade off between run time performance and quality of the generated solution for the described overdo heuristic for a number of different road networks.

4.2 Experimental setup

As long as the future costs that are added to the temporary label of each visited node v are a lower bound on the actual travel time from v to t the algorithm will always find the optimal path and is expected to visit less nodes than Dijkstra's algorithm. For our analysis of the overdo factor, it is useful to study the two components - Euclidean distance and maximum speed - of the future costs individually, since an optimal value should lead to a good aproximation of the actual length of the remaining path and of the actual speed this path can be travelled on. While the actual length of the remaining path depends very much on the geometry of the road network, the actual travel speed will be determined by the travel time on the different links and thus by the load of the network.

Therefore, we studied the A^*-algorithm on various networks of different size and geometry and used various speed models for the links of the networks. In this section we present the main results for four networks: The network of the German city Cologne, the large area network of the German federal state Northrhine-Westphalia and the two large area networks of the US states Kansas and California. Table 4.1 shows the size and average degree for these networks. All networks were normalized in the sense that the length of a link is at least the Euclidean distance between its two endpoints.

Network	Nodes	Edges	Avg.Degree
Cologne	31011	67490	4.35
NRW	457124	1046087	4.58
Kansas	489148	1251913	5.12
California	1580305	3934788	4.98

Table 4.1 Size of the different networks.

Lacking pure dynamical data for the networks we used three different speed models which are supposed to reflect different types of traffic load. A free-flow model where all roads allow the maximum speed of their type, a model of congested roads with low speed on all roads and the uniform model where all types of roads allow the same speed. Although the uniform model is not expected to be a very realistic one[3], it was chosen to capture the pure geometrical properties of the

[3]With the posssible exception of very congested city networks.

shortest path search with the modified A^*-algorithm in road networks. Since all roads allow the same travelling speed, the difference between the exact path and an approximation will be solely the result from varying geometric lengths. Table 4.2 gives the speed in kilometers per hour for the different types of roads for the three models.

Speed Model	Road type				
	0	1	2	3	4
Free-flow	30	30	60	80	100
Congestion	20	20	30	40	50
Uniform	60	60	60	60	60

Table 4.2 Different speed models used on the networks.

The quality of the solutions found by the modified A^*-algorithm were studied using the maximum relative error of travel times and the fraction of paths having an error greater than a given threshold. For the run time performance we studied the actual running times and the number of nodes that are visited by the algorithm.

4.3 Analysis of shortest path quality

4.3.1 Cologne

The road network of Cologne with its 31011 nodes and 67490 edges is the smallest network in our study and has the lowest average degree. Table 4.3 gives some statistical information about the Cologne network. For each road type the ratios according to total number of links and to total length of all links are given in column two and three. Additionally, the maximum link length, average link length and standard deviation of the link distribution are given in meters. The stretch factor gives the average ratio of link length and Euclidean distance between the two endpoints, with its standard deviation shown in brackets. The ratios with respect to length of the total network are higher than those with respect to number of links for types higher than one. The total length of all links in the network is 6498.5 kilometers and the average link length is only 96 meters showing that the network consists mainly of very short links. The longest paths in this network had a travel time of up to 30 minutes for free-flow traffic.

For Cologne we compared 5000 shortest paths found by Dijkstra's algorithm with those found by the modified A^*-algorithm varying the overdo factor between one and ten. In figure 4.2 we show the maximum relative error of paths for the various overdo factors and the three speed models. Up to an overdo factor of four

Road	Ratio		Max.	Avg.	(σ)	Stretch	(σ)
	Number	Length	length	length			
Type 0	71.07	64.07	1550	87	(80)	1.16	(0.22)
Type 1	10.83	10.43	1100	93	(81)	1.13	(0.15)
Type 2	11.55	14.38	1420	120	(119)	1.13	(0.18)
Type 3	6.19	8.04	2380	125	(137)	1.14	(0.23)
Type 4	0.36	3.06	5410	816	(1019)	1.21	(0.7)

Table 4.3 Distribution of road types for Cologne.

the curves for the three different speed models do not differ very much rising to a maximum relative error of about 100%. For higher factors the maximum relative error for the uniform speed model increases more slowly, while for free-flow traffic it climbs above two. Figure 4.3 shows a closeup of the previous plot for overdo factors between one and two. While the curves for free-flow and congested traffic are almost identical the uniform model has slightly higher errors for small overdo factors, which become smaller than those of the other two models if the overdo factor increases.

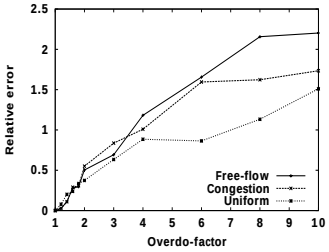


Figure 4.2 Maximum relative error of shortest paths for A^* with overdo for Cologne.

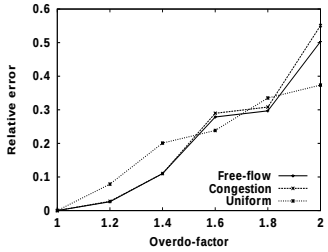


Figure 4.3 Maximum relative error of shortest paths for A^* with overdo $1-2$ for Cologne.

The reason for the smaller maximum relative error for the uniform speed model than for the other two models is just the uniform distribution of speeds for this model. Since all links allow the same speed the only source of potential errors is the geometric length of paths. There is no effect of choosing links of the wrong type. For free-flow and congestion a non exact path might be of the same geomet-

ric length as for the uniform model, but the speed differences up to a factor of 3.5 can lead to significantly larger relative errors. Although this explains why the uniform model gives smaller relative errors it still surprises that even for the uniform model there are paths found by the modified A^*-algorithm that have a relative error of more than one. Since in the uniform model a shortest path is a path of smallest geometrical length, one would expect that the search directed geometrically toward the target node will give paths at least very close to the shortest path.

Therefore, we analyzed the quality of an average path found by the modified A^*-algorithm by looking at the fraction of paths that have a relative error above some given threshold. Figure 4.4 and figure 4.5 show this quality measure for the four thresholds 0, 0.02, 0.1 and 0.5 for the free-flow and uniform traffic speed models.

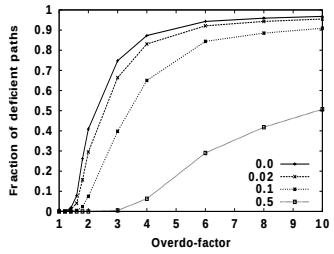


Figure 4.4 Fraction of deficient shortest paths for different thresholds in Cologne with free-flow.

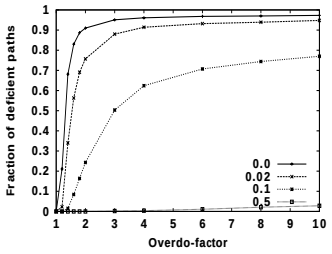


Figure 4.5 Fraction of deficient shortest paths for different thresholds in Cologne with uniform speed.

For the uniform speed model the fraction of paths with a relative error above 0.5 is almost zero even for overdo factor ten. Thus, paths of very high error are very rare and might come from specific geometric properties of the path, i.e. the actual shortest path is directed away from the target node near the starting node to reach some long edges leading straight to the target or there are turning restrictions near the target node causing the found path to use some expensive deviations to reach t. But still, even for the uniform speed model almost all paths are erroneous for overdo factors greater than two. That means that almost no optimal path is just the sequence of nodes where each node minimizes the geometric distance to t of all candidates at that point of the search. For free-flow traffic we observe that there is a smaller fraction of deficient paths compared to the uniform speed model at least for overdo factors below seven. This can be expected since now the different road

Road	Ratio		Max.	Avg.	(σ)	Stretch	(σ)
	Number	Length	length	length			
Type 0	71.41	53.93	4670	114	(115)	1.14	(0.22)
Type 1	8.17	12.93	7810	238	(495)	1.11	(0.16)
Type 2	15.37	22.52	7970	220	(408)	1.11	(0.17)
Type 3	4.65	7.76	7150	251	(428)	1.11	(0.22)
Type 4	0.41	2.86	10630	1056	(1510)	1.13	(0.44)

Table 4.4 Distribution of road types for Northrhine-Westphalia.

types result in shortest paths that are not necessarily the geometric shortest paths. For small overdo factors the speed effect of a link outweighs the effect of the future costs leading to a higher fraction of exact paths. For overdo factors greater than four the future costs dominate the speed effects and the shorter travel times give higher relative errors. For overdo factor ten about half of the suggested paths have a relative error above 0.5.

4.3.2 Northrhine-Westphalia

The network of Northrhine-Westphalia (NRW) with its 457124 nodes and average degree 4.58 is the smallest of the three large area networks. The statistical data in table 4.4 shows that the percentage of roads of type zero and one with respect to length is only about 65% and therefore smaller than for the network of Cologne. This is mainly due to a much higher average link length for type two and three, since the type distribution is very much the same for the two networks. Additionally, the maximum length of links is much longer for all road types than for the road network of Cologne resulting in an average link length of 150 meters for a total of 157241.8 kilometers. The longest paths had a travel time of up to 200 minutes for free-flow traffic. From the data we observe that the two networks have a very similar structure with respect to type four and type zero edges. The differences between the two networks arise mainly from the links of the other types.

Analyzing the quality of shortest paths between 10000 node pairs chosen at random we see in figure 4.6 that the maximum relative error is very much the same for free-flow traffic and congestion up to an overdo factor of about six and reaching a maximum of 2.2 for free-flow traffic and factor ten. For the uniform speed model this maximum relative error increases almost linearly up to a value of 1.1. For overdo factors between one and two (see figure 4.7) the error is about 0.2 for factors greater than 1.4 for all speed models.

The three curves for the maximum relative error for the three speed models look very much the same as those for Cologne especially for small overdo factors

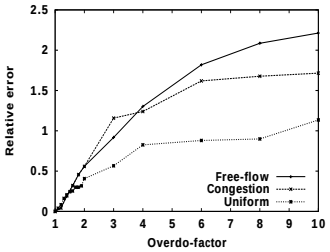


Figure 4.6 Maximum relative error of shortest paths for A^* with overdo for NRW.

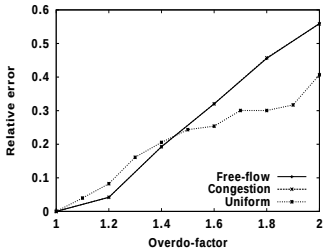


Figure 4.7 Maximum relative error of shortest paths for A^* with overdo $1-2$ for NRW.

(compare figure 4.3 and figure 4.7). Since the network of Cologne is a subgraph of NRW we do not expect the overall maximum relative error to decrease in NRW, although we observe a lower error for overdo factor ten in the uniform model in NRW than in Cologne due to different sets of node pairs for the shortest paths. On the other hand, the plots suggest that the maximum relative error is independent of the size of the network at least for networks of very similar type distribution and geometry.

While the maximum relative errors observed for the modified A^*-algorithm are comparable for NRW and Cologne the fraction of deficient paths is much worse for the large area network of NRW. As can be seen in figure 4.8 for free-flow traffic the curves are much steeper than for Cologne. The fraction of not exact paths is almost 90% for an overdo factor of two. For factor three almost 80% of the paths have a relative error greater than 0.1 and for overdo factor six the fraction of paths having a relative error above 0.5 is higher than 70%, rising to almost 90% for factor ten. The plot for congested traffic looks almost the same with a fraction of about 0.8 for paths with relative error greater than 0.5 for overdo factor ten. For the uniform speed model (see figure 4.9) for overdo factors greater than two the fraction of paths with error greater than 0.0 resp. 0.02 are close to one and above 0.6 for threshold 0.1. Paths of error greater than 50% are very rare in NRW as in Cologne.

Comparing the results for NRW and Cologne shows that the geometric length of the paths found by the algorithm in NRW tend to deviate more from the exact paths than in Cologne. A reason for this might be the longer average link length in NRW resulting in longer deviations. For free-flow traffic the fraction of paths with high error is significantly higher in NRW than in Cologne. This suggests that the type distribution of paths found by the algorithm in NRW differs much more

from the distribution of exact paths than in Cologne.

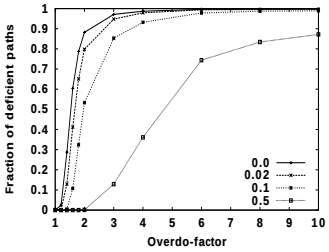


Figure 4.8 Fraction of deficient shortest paths for different thresholds in NRW with free-flow.

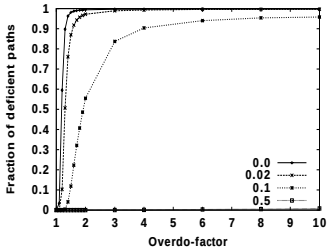


Figure 4.9 Fraction of deficient shortest paths for different thresholds in NRW with uniform speed.

4.3.3 Kansas

The road network of Kansas has 489148 nodes and an average degree of 5.12. It is therefore only slightly larger than the one of NRW and was chosen because of its very grid-like structure, more typically for networks in the United States. The statistical data in table 4.5 shows that in contrast to NRW the Kansas network consists mainly of type zero links that are on the average four times as long as the respective links in NRW. For the other types of roads the NRW network has more and longer links. There is not much difference between the link distributions with respect to number and length. The average link length is 441 meters, about three times as long as in NRW. The total length of roads in Kansas is 552589 kilometers giving paths with a travel time of up to 640 minutes for free-flow traffic. Note also that the stretch factor of the links is very close to one, indicating very straight links between the two respective endpoints. The data supports our expectation that the Kansas network is of simpler geometrical structure than the almost equally sized NRW network.

Figure 4.10 compares the maximum relative errors for 10000 shortest paths for overdo factors between one and ten. For the uniform speed model this error climbs up to 0.5 already for overdo factor less than two but rises not much for greater factors. In fact it is almost constant for factors greater than four suggesting that the increase of the factor always leads to the same path of maximum relative error. Although an error of 50% resulting solely from a geometrical deviation is very high

Road	Ratio		Max.	Avg.	(σ)	Stretch	(σ)
	Number	Length	length	length			
Type 0	93.66	93.62	13980	441	(487)	1.07	(0.37)
Type 2	2.77	2.98	6950	476	(495)	1.05	(0.14)
Type 3	2.89	2.79	5130	426	(459)	1.05	(0.19)
Type 4	0.69	0.61	5890	393	(510)	1.05	(0.11)

Table 4.5 Distribution of road types for Kansas.

the results for the uniform speed model are better than for the NRW network. This is in line with the observed simpler geometric structure of the Kansas network. On the other hand the maximum relative error is higher for the Kansas network if different speeds are used on the roads. For congested traffic the relative error is almost two for overdo factor four and reaches a maximum close to three for free-flow traffic. These extreme deviations are even more surprising remembering that almost 95% of the roads in the Kansas network are of type zero and thus a higher speed is used only on very few roads. Therefore, the fact that these errors occur means that the algorithm finds paths consisting more or less of type zero edges while the exact paths instead use mainly edges of type four. The closeup to overdo factors between one and two in figure 4.11 shows that there is almost no difference between the three speed models in this range but the observed errors are slightly higher than for the NRW network.

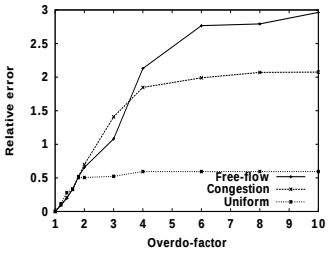


Figure 4.10 Maximum relative error of shortest paths for A^* with overdo for Kansas.

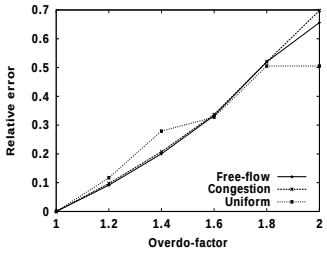


Figure 4.11 Maximum relative error of shortest paths for A^* with overdo $1-2$ for Kansas.

The differences between Kansas and Northrhine-Westphalia for the fraction of deficient paths are less significant resulting in very similar plots. For the uniform

speed model in the Kansas network (see figure 4.13) the fraction for threshold zero is very close to one but rises not above 0.7 for threshold 0.1. In NRW the fraction for threshold 0.1 was above 0.9. This means that the algorithm finds paths of slightly smaller relative error in Kansas for uniform speed. On the other hand for free-flow traffic and threshold 0.5 the fraction is clear above 0.9 (see figure 4.12). For the other three thresholds the curves are not as steep as for the NRW network especially for overdo factors below three. The same applies for congested traffic.

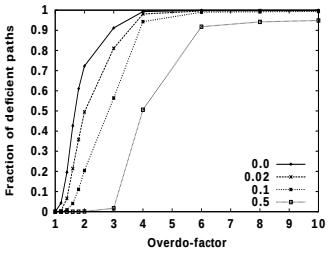


Figure 4.12 Fraction of deficient shortest paths for different thresholds in Kansas with free-flow.

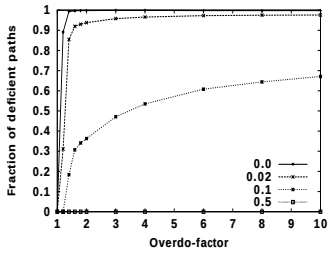


Figure 4.13 Fraction of deficient shortest paths for different thresholds in Kansas with uniform speed.

4.3.4 California

The network of California is by far the largest of the networks in this study. It has 1580305 nodes and almost four million edges giving an average degree of almost five. The distribution of the different road types is very much the same as for the Kansas network with an even higher percentage of type zero links. Although there are longer links than in any other of the networks the average link length is 288 meters, thus lying in between that of the Kansas and the NRW network. The total length of all links combines to 1134068 kilometers resulting in paths with a travel time as long as 1260 minutes for free-flow traffic.

We see in figure 4.14 that the maximum relative errors for overdo factors between one and ten in California are the highest of all four networks. Even for the uniform speed model the error is as high as 1.5 for long paths and thus three times as high as in Kansas. Similar to Kansas the maximum relative error for uniform speed increases more slowly than for NRW. For free-flow traffic we get relative errors of up to 3.5. For overdo factors between one and two (cf. figure 4.15) the difference between the three speed models is small with uniform speed being slightly

Road	Ratio		Max.	Avg.	(σ)	Stretch	(σ)
	Number	Length	length	length			
Type 0	96.45	95.62	23720	286	(471)	1.12	(0.56)
Type 2	2.23	2.87	15180	371	(553)	1.07	(0.18)
Type 3	2.89	2.79	7520	447	(607)	1.05	(0.09)
Type 4	0.69	0.61	12990	309	(497)	1.06	(0.18)

Table 4.6 Distribution of road types for California.

worse for small factors. Thus, here the size of the network affects the maximum relative error, although Kansas and California have a very similar type distribution.

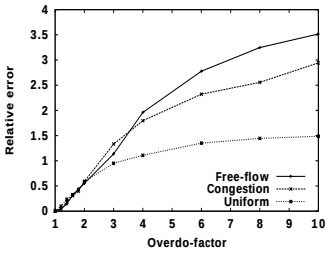


Figure 4.14 Maximum relative error of shortest paths for A^* with overdo for California.

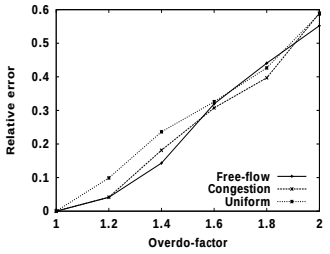


Figure 4.15 Maximum relative error of shortest paths for A^* with overdo $1-2$ for California.

The fraction of deficient paths for free-flow traffic rises not as steeply for California as for NRW for overdo factors below three and is very similar than for Kansas (cf. figure 4.16). Almost all paths in California have errors greater than 50% for overdo factor ten with free-flow traffic. For uniform speed the fractions of deficient paths in California are also highest. While in the other large area networks the fraction of paths with relative errors greater than 0.5 were below 1% even for high overdo factors, it climbs up to 10% in California. Even the fraction of paths with errors greater than 10% already lies above 90% for overdo factor three in California.

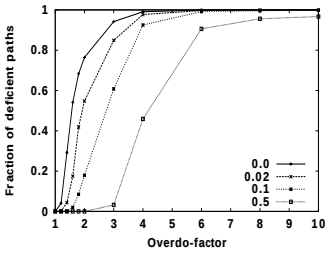


Figure 4.16 Fraction of deficient shortest paths for different thresholds in California with free-flow.

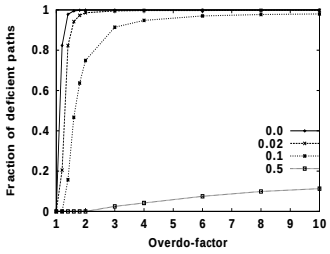


Figure 4.17 Fraction of deficient shortest paths for different thresholds in California with uniform speed.

4.3.5 A closer look at maximum error paths

The results presented in the previous sections for four road networks of different size and geometry showed that the modified A^*-algorithm with overdo will already for an overdo factor greater than two generate shortest paths that can have extreme high relative errors above 50%. For higher overdo factors the fraction of paths that deviate from the exact path by such great errors increases above 0.5 at least for free-flow and congested traffic. Although paths of such poor quality are not satisfying in any case, from a practical approach it is also important to know if such high deviations occur for long journeys where high relative errors might cause a delay of several hours or only for short trips of a couple of minutes.

Therefore, we investigated how the relative errors relate to the length of the actual shortest path. In figure 4.18 and figure 4.19 we present the results for the road network of Cologne and the large area network of California for overdo factors two, four and ten using free-flow traffic since for this speed model the relative errors were highest. The travel time was divided into bins of one minute length for Cologne and ten minute length for California.

The plots show that the paths found by the modified A^*-algorithm are of poor quality regardless of the length of the exact path. Therefore, they confirm the previous observation that for overdo factors greater than two the A^*-algorithm with overdo will mostly generate paths of very high relative error. This is surprising at first hand since the idea of the modified A^*-algorithm is to direct the search towards the target node and thus, we expect the found paths to be close to the optimal path especially in the uniform speed model.

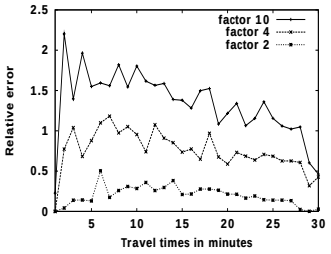


Figure 4.18 Maximum relative errors subject to travel time for Cologne with free-flow traffic.

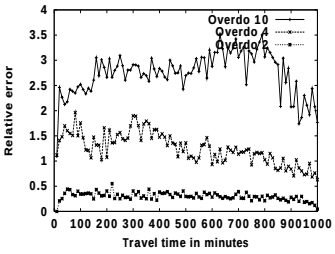


Figure 4.19 Maximum relative errors subject to travel time for California with free-flow traffic.

One potential source of error lies in the strongly hierarchical structure of road networks where turning restrictions, bridges and highway access might lead to very poor results when searching directly toward the target node. Figure 4.20 gives an example for the Cologne network with the uniform speed model for a very short path of length about three kilometers and overdo factor ten.

But even if the modified A^*-algorithm finds a path that is actually directed towards the target node the errors can be very high due to the different road types. Figure 4.21 shows a shortest path in NRW of length almost 250 kilometers. The modified A^*-algorithm finds a path with relative error 1.5 that runs almost parallel to the shortest path. But while the shortest path uses almost exclusively highway edges, the path found by the heuristic uses only edges of low type. The reason for this is that by using the overdo factor any improvement in the Euclidean distance to the target is weighted higher than the travelling time on the used edge. This is proven by the sequence of successive labels which is almost monotonely decreasing for the heuristic and increasing for the optimal path. Thus, if at some point a node of the shortest path has a higher temporary label than the current scanned node in the heuristic than this node will never be scanned by the heuristic since the following labels are all smaller. As the example shows this effect already arises for overdo factor three and leads to the dramatic relative errors for higher overdo factors. Eventually, for such high factors the algorithm always scans as next node a neighbour of the just previously scanned node. The label of this neighbour is either smaller than the label of the previous scanned node or at least smaller than the labels of previous nodes that have not been scanned yet. Thus, if the algorithm once takes a 'wrong' turn, it will never come back to this deviaton point. Figure 4.22 shows an example for Cologne and figure 4.23 one for the network of NRW

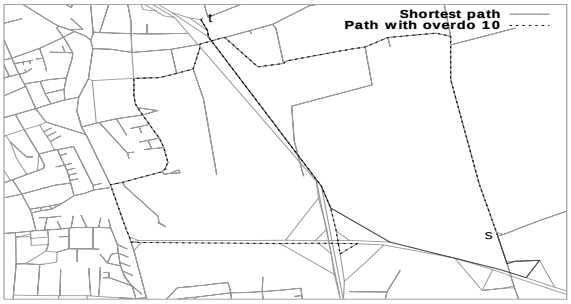


Figure 4.20 A path of relative error 2.5 in Cologne with uniform speed for overdo factor 10. The path found by the modified A^*-algorithm (shown with dashed lines) is directed from the starting node s towards the target node t which lies on a highway and almost reaches it. Since there is no opportunity to turn from the bridge on the highway the algorithm must find the closest highway access point from there. The first node on the shortest path (shown with solid lines) after the starting node has such a high label that it is never scanned by the A^*-algorithm.

for overdo factor ten.

As shown by the examples for high overdo factors the modified A^*-algorithm chooses the sequence of scanned nodes not according to any aspects of shortest distances from the starting node but by accounting only for neighbours of the previous scanned node. Once a bad decision is made there is no way to recover from it. Instead, the path found to the target node is only based on reachability. In this case the sequence of successive labels is strictly decreasing or at least close to it. In section (4.5) we will derive an upper bound for the overdo factor with the property that for higher factors the algorithm will show the described behaviour.

4.3.5.1 Summary for the quality of solutions

The results of our experimental study of the modified A^*-algorithm on the different networks with various speed models can be summarized as follows:

- For overdo factors between one and two the maximum relative error is about the same for all four networks. Also, there is almost no difference for this error between the three speed models we studied.

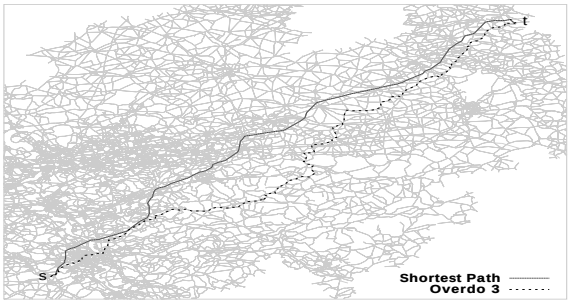


Figure 4.21 A path of relative error 1.5 in NRW with free-flow traffic for overdo factor 3. The path found by the modified A^*-algorithm (shown with dashed lines) runs almost parallel to the shortest path (shown with solid lines). At some point the shortest path deviates slightly away from the target node t and the heuristic prefers the geometrically direct way. The labels on the dashed path are almost monotonously decreasing, thus the alternative node on the shortest path is never scanned. The shortest path has 169 edges, 87% of which are highway edges. The path found by the heuristic has 859 edges, none of which is a highway edge.

- For overdo factors greater than three the sequence of successive labels of the nodes on the suggested path are more or less decreasing. That means, that the algorithm always scans as next node that neighbour of the previous node that is geometrically closest to the target node. Nodes that were reached but not scanned earlier in the search are never returned to. Thus, turning restrictions, bridges and highway access have a very strong influence on the search and the algorithm follows a wrong turn according to reachability to the target node, while shortest path distances from the starting node are of almost no relevance for the search. For such overdo factors the maximum relative errors can get very high even for the uniform speed model where the shortest path is the one with the least geometrical distance.
- For the uniform speed model the Kansas network has the lowest maximal relative errors while those of the other three networks are about the same. This suggests that the simpler grid-like structure of the Kansas network with the highest average degree of all networks is slightly better suited for an application of the A^*-algorithm with overdo.

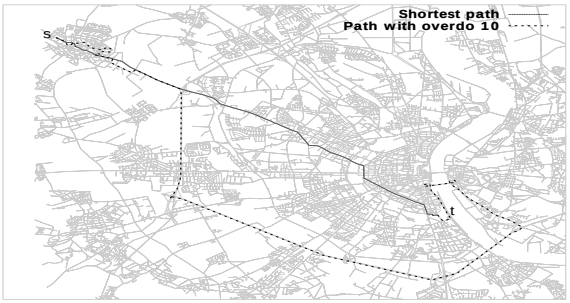


Figure 4.22 An *s*-*t*-path of relative error 2.1 in Cologne with uniform speed for overdo factor 10. The path found by the modified A^*-algorithm (shown with dashed lines) always scans a neighbour of the previously scanned node. At some node it turns on the highway leaving the shortest path (shown with solid lines). From then on, the heuristical path stays on the highway since there is no left-turn until it becomes cheaper to turn right on the highway than to continue straight ahead.

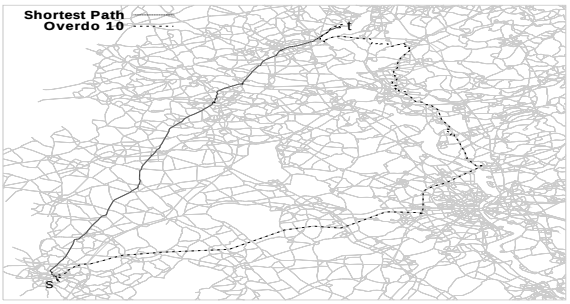


Figure 4.23 An *s*-*t*-path of relative error 1.66 in NRW with uniform speed for overdo factor 10. The path found by the modified A^*-algorithm (shown with dashed lines) chooses near the starting node a node that is geometrically closer to *t* than the next node on the shortest path (shown with solid lines). From then on, the sequence of labels is strictly decreasing. The node on the shortest path just after the deviation node is never scanned.

- For free-flow traffic the maximum relative errors are significantly higher for the US networks. One of the reasons is the high fraction of low type edges in the found paths while the shortest paths use links of high type. The difference between the German and US networks can be explained with the much higher average length of high type edges in Cologne and NRW compared to the other types and the US networks. This will include more edges of high type in paths found by the algorithm even for high overdo factors, reducing the relative error of these paths. This effect leads to a higher fraction of paths with a high relative error for the US networks.
- On the other hand for free-flow traffic and overdo factors up to three the ratios of deficient paths with an error greater than a given threshold are highest for the NRW network. This is due to the fact that for this network the fraction of type four edges in shortest paths is the highest. If the algorithm does not find the exact path then the suggested path most likely contains edges of lower type instead, leading to higher relative errors. The fraction of type four edges in shortest paths is lowest in Cologne, accordingly the errors of deficient paths do not increase as fast with large overdo factors.
- The travel time on the paths in the uniform speed model is not affected by different speeds on the links. This effect causes the relative errors for uniform speed to be smaller leading to a higher fraction of paths with errors greater than 10% for free-flow and congested traffic.

Comparing our results on the four networks with those of Jacob et al. [56] for the Dallas/Fort Worth network we observe that the modified A^*-algorithm finds paths of very poor quality already for overdo factors greater than five in our experimental study. This contrasts to a maximum relative error of no more than 16% in the study of Jacob et al. even for overdo factor 99. There are several points that might explain this discrepancy [55]: The used Dallas/Fort Worth network was not as detailed as the ones in our study. Only in a very small area all roads were present, while for the other parts only major roads were included. Additionally, the input data of the roads was less exact in Dallas/Fort Worth. If fewer links are present in the network then a search strategy of choosing always a neighbour as next node to be scanned will give better paths as long as the search is directed towards the target node. Another major difference between the two studies is the source of the edge weights. While our study is based on different static speed models with a few classes of possible speeds, Jacob et al. used load data coming from a traffic microsimulation of the area. This data is much less structured in the sense that stretches of edges allowing exactly the same speed are more uncommon. Therefore, a wrong turn on a subpath of minor roads will probably have a greater effect on the quality of the path in the static model.

4.4 Analysis of runtime performance

In order to examine the runtime performance of the modified A^*-algorithm we use the actual running time of the algorithm and the number of nodes scanned during the shortest path calculation. Both measures have their drawbacks: The running time depends in the first place on the used hardware and the stated results therefore do not allow a general quantification of the running time performance but give only an indication of possible speedups. In contrast is the number of scanned nodes hardware-independent and thus captures the algorithmic behaviour of the shortest path search more accurately. But this measure does not include the algorithmic work that has to be performed in order to calculate the Euclidean distances for the A^*-algorithm. For Dijkstra's algorithm this calculation is not necessary and therefore, the measured number of scanned nodes favours the A^*-algorithm.

4.4.1 Dijkstra versus modified A*-algorithm

In figure 4.24 we show the speedup of the modified A^*-algorithm for overdo factors between one and ten for the different networks with free-flow traffic. For each overdo factor we averaged the speedup of all shortest path calculations for each network. All four networks show a significant speedup for overdo factors between one and two climbing from 1.5 to almost six for the Kansas network. For greater overdo factors the speedup is almost constant or even declining as for the Cologne network. The corresponding running times are shown in figure 4.25 for the three large area networks. The runtime of Dijkstra's algorithm is plotted as overdo factor zero.

Looking at the gain of the modified A^*-algorithm regarding scanned nodes in figure 4.26 we observe that Dijkstra's algorithm scans more than a hundred times more nodes than the heuristic for overdo factors greater than three for the three large area networks. For smaller overdo factors between one and two (cf. figure 4.27) the speedup is still significant especially for the three large area networks.

The idea behind the modified A^*-algorithm is to direct the search toward the target node thereby scanning not as many nodes as Dijkstra's algorithm. The higher the overdo factor gets the fewer nodes are expected to be permanently labeled by the algorithm. On the other hand for the modified A^*-algorithm additional algorithmic work has to be performed by calculating the Euclidean distance between the target node and each visited node. This will slow down the modified A^*-algorithm and is expected to have a greater effect, if the path gets longer. This expectation is confirmed by figure 4.28 for the Cologne network and figure 4.29 for the California network both with free-flow traffic. The figures show the speedup of the modified A^*-algorithm against Dijkstra's algorithm with respect to running time plotted against the length of the shortest path in minutes.

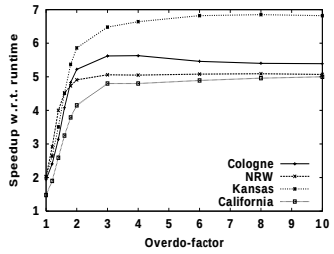


Figure 4.24 Runtime speedup of the modified A^*-algorithm compared to Dijkstra's algorithm for different networks.

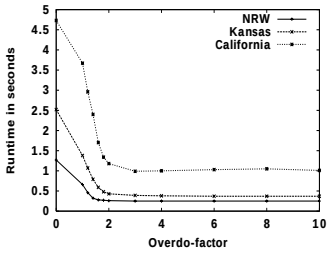


Figure 4.25 Runtime of the modified A^*-algorithm for the three large area networks.

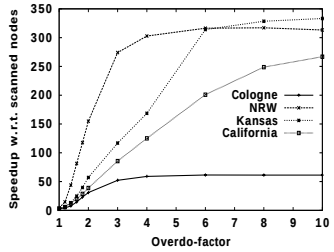


Figure 4.26 Speedup w.r.t. to scanned nodes of the modified A^*-algorithm compared to Dijkstra's algorithm for different networks.

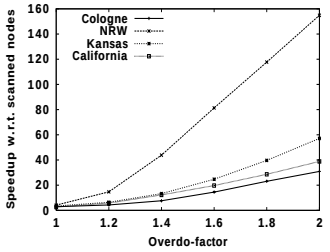


Figure 4.27 Speedup w.r.t. to scanned nodes of the modified A^*-algorithm for the three large area networks and overdo factors between one and two.

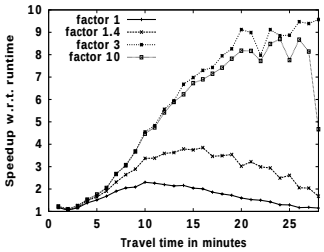


Figure 4.28 Speedup w.r.t. to runtime of the modified A^*-algorithm compared to Dijkstra's algorithm evolving over travel time for the Cologne network with free-flow traffic.

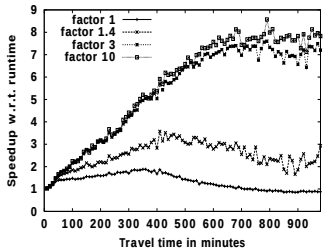


Figure 4.29 Speedup w.r.t. to runtime of the modified A^*-algorithm compared to Dijkstra's algorithm evolving over travel time for the California network with free-flow traffic.

For the classical A^*-algorithm (overdo factor one) the speedup increases only for paths up to a length of 10 minutes (Cologne) resp. 400 minutes (California). For longer paths the speedup declines and is even less than one for the longest paths, i.e. Dijkstra's algorithm is faster than the A^*-algorithm. For overdo factor 1.4 we observe a similar behaviour but at least the heuristic is not slower than Dijkstra's algorithm. For overdo factors three and ten the speedup of the heuristic stays on a high level.

In figure 4.24 we observed an effect of declining speedup for larger overdo factors in the Cologne network which can be seen again in figure 4.28 for longer paths. The only explanation for this effect is that the modified A^*-algorithm visits and thus scans more nodes for overdo factor ten than for overdo factor three. This would be absolute contrary to the desired effect of the heuristic. Next to the scanning of the nodes there is also the additional computation of Euclidean distances if more nodes are visited.

We observed this effect of an increasing number of scanned nodes for high overdo factors in all networks. While for most networks the increase was only very gradual, in one network the algorithm scanned about 20% of the nodes scanned by Dijkstra's algorithm for overdo factor four but as many nodes as Dijkstra's algorithm for overdo factor ten. It is possible that a node that is not scanned for some overdo factor f might be scanned for overdo factors greater than f (see [56] for a small numerical example) but a strong increase of scanned nodes for high overdo factors cannot be explained with this observation. Instead, the reason is that nodes

are scanned more than once by the algorithm. That means that the triangle inequality is not fulfilled for these networks and for high overdo factors the algorithm first neglects some node v being the predecessor of a node w on the path. Node v is scanned later resulting in a rescanning of all nodes between the scanning of w and v. In fact we observed shortest path calculations in the Cologne network where more than 2000 nodes were scanned more than once. For other paths we had a few nodes that were scanned more than 80 times. In larger networks more than 15000 nodes were multi-scanned for some paths and there were paths where some nodes were scanned more than 700 times.

4.4.2 Comparison of Euclidean against Manhattan distance

By weigthing the Euclidean distance between some node v and target node t of the desired path with some overdo factor the actual geometric length of the path $\mathcal{P}(v,t)$ is approximated. One can think therefore about using a different distance measure for the approximation of the path length which might be computationally more efficient than the Euclidean distance. One such measure is the Manhattan distance *mhdist*, i.e. $mhdist(v,t) = |x_v - x_t| + |y_v - y_t|$ which avoids the operation of taking the square root.

We analyzed the runtime performance of the two versions of the modified A^*-algorithm on the network of NRW averaging over 10000 shortest paths. Note that the two versions of the algorithm will in general choose different nodes for scanning and the found paths will not be the same since the future costs, being one part of the label at each node, are different due to the varying distance measures. Most importantly will the number of nodes that are scanned not be the same and thus the difference of the actual running times of the two algorithms will be mostly affected by the numerical difference of scanned nodes. To actually measure the effect of using the Manhattan distance instead of the Euclidean distance we therefore used as label for a node v only the distance from s to v and additionally calculated the future costs for each node visited without using this information[4].

Figure 4.30 shows that the speedup of the running times using the Manhattan against the Euclidean distance for the NRW network increases for longer paths but is less than 2% and therefore almost insignificant. Since the modified A^*-algorithm scans in general far less nodes than Dijkstra's algorithm the speedup in a direct application of the two algorithms must be expected to be even smaller. For a high fraction of paths using the Euclidean distance results even in faster running times as figure 4.31 shows. The fraction of paths using the Manhattan distance with faster

[4] Thus, we just ran two normal Dijkstra algorithms and additionally calculated the future costs using the Euclidean resp. Manhattan distance for each node encountered.

running times is below 75% for almost all travelling times.

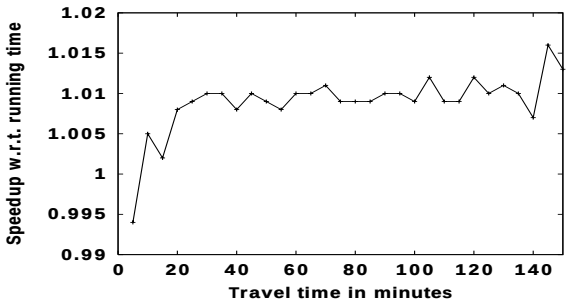


Figure 4.30 Speedup w.r.t. running times using Manhattan against Euclidean distance for the A^*-algorithm on the network of NRW. The travel times were binned in five minute intervals.

While the Euclidean distance gives a lower bound on the length of the actual path between two nodes the Manhattan distance might be longer. Therefore the A^*-algorithm with an overdo factor of one using the Euclidean distance will find the exact shortest path, while using the Manhattan distance with an overdo factor of one might lead only to approximate paths[5]. By appropriately choosing an overdo factor of less than one the algorithm with Manhattan distances can be forced to find the exact path. The marginal runtime savings although suggest that there is no real payoff for the additional work.

Our analysis of the run time performance of the modified A^*-algorithm shows that the heuristic outperforms Dijkstra's algorithm on the average by a factor of at least two and more than four for overdo factors greater than two. For overdo factors of up to two Dijkstra's algorithm scans significantly more nodes, but for long paths the additional computation of the Euclidean distance reduces the run time speedup of the heuristic and even leads to a slower algorithm than Dijkstra in some cases. For high overdo factors the speedup of the heuristic with respect to number of scanned nodes goes up to a factor of several hundred, resulting in a much faster algorithm even for long paths. On the other hand the violation of the triangle inequality for these overdo factors leads to the rescanning of nodes. This slows the heuristic down and in some cases countereffects the idea of the modified

[5] In our test on the network of NRW we observed more than 10% of erroneous paths with errors as high as 17%. About 2% of the paths had errors of 5% or more.

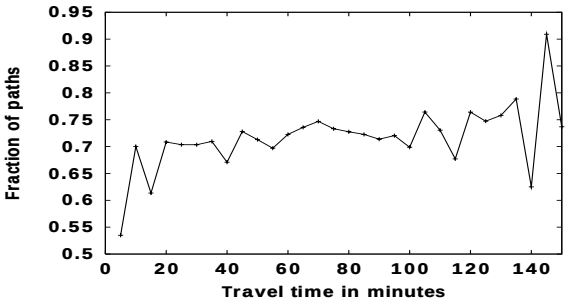


Figure 4.31 Fraction of paths with better running times using the Manhattan distance on the network of NRW. The travel times were binned in five minute intervals.

A^*-algorithm when some nodes are scanned several hundred times.

In summary small overdo factors generate on the average solutions of good quality with a reasonable speedup against Dijkstra's algorithm. For longer paths the computation of the Euclidean distance reduces the gain in runtime of the heuristic significantly. High overdo factors lead to a very fast algorithm, but show a very poor solution quality. Additionally, in some cases a multiple rescanning of nodes can be observed.

4.5 Theoretical bounds for the overdo factor

In section (4.3.5) we saw that for high overdo factors the modified A^*-algorithm scans as next node always a neighbour of the previously scanned node. The distance from the starting node has almost no effect and there is almost no chance to recover from a wrong turning decision. Thus, the found path is mainly characterized by reaching the target somehow, but not according to some kind of distance minimization. This will always happen if the savings in the future costs of being closer to the target node for a node w are greater than the additional expenses to reach w. In this case the sequence of labels of successive nodes on the path is often decreasing.

From this observation we will derive in section (4.5.1) an upper bound for the overdo factor with the property that for higher factors the sequence of successive labels is strictly decreasing. If the sequence of successive labels has this prop-

erty, the algorithm always scans a neighbour of the current node as next node. To achieve the upper bound we have to make some assumptions on the network which we expect to be fulfilled in road networks in most cases.

In section 4.5.2 we derive an expected lower bound for the optimal overdo factor in a gridgraph with uniform speed on all links. Although this bound must not hold in an arbitrary road network with uniform speed together with the upper bound from section 4.5.1 it gives an interval for the choice of the overdo factor in a practical application with dynamic link weights.

For the analysis of the algorithmic behaviour we distinguish between scanning and visiting a node. If a node u is scanned then the temporary label of u becomes the permanent label[6] and for all of its neighbours w the algorithm calculates the sum of travelling time to w via u and future costs of w. This sum is compared to the temporary label of w and possibly updated if the sum is smaller. To visit a node u shall mean that a temporary label has been calculated but the node was not scanned yet.

4.5.1 Upper bound for the overdo factor

In this section we will derive an upper bound $f_{ov}^{\max}$ for the overdo factor such that for overdo factors greater than $f_{ov}^{\max}$ the modified A^*-algorithm will scan as next node always one of the neighbours of the previous scanned node (if there is at least one neighbour). Thus, if the algorithm at some point scans a node u, all not yet scanned nodes that were visited from nodes on the path from s to u will never be scanned as long as their temporary label is not decreased at some later stage of the algorithm. This means that if the algorithm takes a 'wrong' turn at some node u it will never come back to the other nodes visited from u and thus will not be able to find the exact shortest path which might lead over one of those neighbours as shown by the examples in section 4.3.5.

For the described algorithmic behaviour of always scanning a neighbour we have to assume that there are no dead-end nodes in the network for otherwise the node scanned after the backtracking of a dead-end path must not be a neighbour of the last node scanned. Although realistic road networks always have dead-end nodes our assumption is easily justified in the context of analysis of the algorithmic behaviour since leaving out dead-end paths does not alter all possible paths between s and t as long as both nodes do not lie on such dead-end paths.

We will need the following notation for node u and edge e in an application of the modified A^*-algorithm calculating a path for fixed starting node s and target t.

f_{ov}: Overdo factor

[6] We assume here that the triangle inequality is fulfilled and there is no cycling in the algorithm.

l(e):	Length of edge e in meters
v(e):	The travelling speed for edge e in meters/seconds
v_{max}:	Maximal speed of an edge in the network in meters/seconds
d(u):	Travel time on the shortest path from s to u in seconds
eu(u,t):	Euclidean distance between nodes u and t in meters
fc(u):	Future costs of node u calculated as $eu(u,t)/v_{max}$
label(u):	Label of u according to which the nodes are chosen by the algorithm in seconds. The label is the sum of d(u) and fc(u).

The main result of this section is stated in the following lemma:

Lemma 4.5.1 *Let u be a node scanned in a dead-end-free network at some stage of the modified A*-algorithm.*

(i) If there is an edge $e = (u, w)$ for which the angle between line **(u, t)** *and e is at most* $45°$*, $eu(u,t) > eu(w,t)$ and the overdo factor f_{ov} satisfies*

$$\frac{l(e)}{eu(u,t)\cdot\left(1-\sqrt{1-\frac{\sqrt{2}\cdot l(e)}{eu(u,t)}+\left(\frac{l(e)}{eu(u,t)}\right)^2}\right)}\cdot\frac{v_{max}}{v(e)} \;<\; f_{ov} \qquad (4.1)$$

then $label(w) < label(u)$.

(ii) If node w is scanned just after node u and $label(w) < label(u)$, then there is an edge (u, w), i.e. w is a neighbour of u.

Before we prove the lemma we will discuss the algorithmic implications. The first part of the lemma gives a sufficient condition in terms of the overdo factor and the existence of a specific edge for the sequence of successive labels to be strictly decreasing. The second part of the lemma then shows that the modified A^*-algorithm always scans a neighbour of the previous scanned node if the sequence of successive labels is strictly decreasing. Thus, under the conditions of the first part of the lemma the algorithm will always scan a neighbour of the previous scanned node, leading to paths that are mainly characterized by reachability as discussed earlier.

A necessary condition for this behaviour certainly is that there is some edge between consecutively scanned nodes. The lemma shows that for sufficiency we need a bound on the overdo factor and an additional condition for such an edge. Note also that the decreasing sequence of labels is a stronger claim on the algorithmic behaviour since always adding a neighbour w of the previous scanned node u to the path can also occur if $label(w) \geq label(u)$. The label of w only has to be

smaller than all other temporary labels at that point of the algorithm. Figure 4.22 shows this effect at the node where the found path leaves the highway. By deriving the stronger property from the condition on the overdo factor we will eventually get an upper bound for the overdo factor which might prove too high in practice.

Proof of lemma 4.5.1. For the proof of the first part of the lemma let us first assume that there is an edge $e = (u, w)$ for which the angle between line (**u, t**) and e is exactly 45° (see figure 4.32).

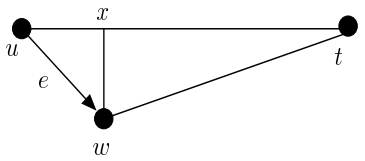


Figure 4.32 Edge e having a 45° angle with line (**u, t**).

For the Euclidean distance between w and t we have:

$$\begin{aligned} eu(w,t) &= \sqrt{(eu(u,t) - eu(u,x))^2 + eu(x,w)^2} \\ &= \sqrt{(eu(u,t) - l(e)/\sqrt{2})^2 + (l(e)/\sqrt{2})^2} \\ &= \sqrt{eu(u,t)^2 - 2 \cdot eu(u,t) \cdot l(e)/\sqrt{2} + 2 \cdot (l(e)/\sqrt{2})^2} \\ &= eu(u,t) \cdot \sqrt{1 - \frac{\sqrt{2} \cdot l(e)}{eu(u,t)} + \left(\frac{l(e)}{eu(u,t)}\right)^2}. \end{aligned} \tag{4.2}$$

Plugging equation (4.2) into inequality (4.1) gives

$$\frac{l(e)}{eu(u,t) \cdot \left(1 - \sqrt{1 - \frac{\sqrt{2} \cdot l(e)}{eu(u,t)} + \left(\frac{l(e)}{eu(u,t)}\right)^2}\right)} \cdot \frac{v_{max}}{v(e)} < f_{ov}$$

$$\Leftrightarrow \quad \frac{l(e)}{eu(u,t) - eu(u,t) \cdot \sqrt{1 - \frac{\sqrt{2} \cdot l(e)}{eu(u,t)} + \left(\frac{l(e)}{eu(u,t)}\right)^2}} \cdot \frac{v_{max}}{v(e)} < f_{ov}$$

$$\Leftrightarrow \quad \frac{l(e)}{eu(u,t) - eu(w,t)} \cdot \frac{v_{max}}{v(e)} < f_{ov}. \tag{4.3}$$

If the angle between line (**u, t**) and edge $e = (u, w)$ is smaller than 45°, then $eu(w, t)$ will be smaller than the right hand side of equation (4.2) for fixed length of e, thereby increasing the denominator of the first fraction of the left hand side of inequality (4.3), since we assumed that $eu(u, t) > eu(w, t)$. Thus, inequality (4.3) holds if the assumptions of the first part of the lemma are satisfied. Together with $eu(u, t) > eu(w, t)$ the assumption for the overdo factor in lemma 4.5.1 leads to the following sequence of inequalities:

$$\frac{l(e)}{eu(u,t)\cdot\left(1-\sqrt{1-\frac{\sqrt{2}\cdot l(e)}{eu(u,t)}+\left(\frac{l(e)}{eu(u,t)}\right)^2}\right)}\cdot\frac{v_{max}}{v(e)} < f_{ov}$$

$$\Rightarrow \quad \frac{l(e)}{eu(u,t)-eu(w,t)}\cdot\frac{v_{max}}{v(e)} < f_{ov}$$

$$\Leftrightarrow \quad \frac{l(e)}{v(e)} < f_{ov}\cdot\frac{eu(u,t)-eu(w,t)}{v_{max}}$$

$$\Leftrightarrow \quad d(u)+\frac{l(e)}{v(e)}+f_{ov}\cdot\frac{eu(w,t)}{v_{max}} < d(u)+f_{ov}\cdot\frac{eu(u,t)}{v_{max}}$$

$$\Leftrightarrow \quad d(w)+f_{ov}\cdot\frac{eu(w,t)}{v_{max}} < d(u)+f_{ov}\cdot\frac{eu(u,t)}{v_{max}}$$

$$\Leftrightarrow \quad label(w) < label(u) \qquad (4.4)$$

For the second part of the lemma recall that the algorithm always chooses the node with minimal temporary label for scanning. Thus, $label(u) \leq label(w)$ when u was chosen for scanning. But then the label of w must have changed when u was scanned in order to give $label(w) < label(u)$. Since in each step only adjacent nodes reached over edges with the scanned node as starting node are visited, there must be an edge (u, w). □

Discussion of the assumptions in lemma 4.5.1. Lemma 4.5.1 shows that the algorithm shows a non-desirable behaviour for overdo factors greater than the left hand side of inequality (4.1) if the algorithm encounters in each step an edge $e = (u, w)$ for which the angle between line (**u, t**) and e is in the interval $[0, \ldots, \pi/2]$ and additionally w lies geometrically closer to t than u.

For the first assumption we observe that many nodes in the network are part of a four-road crossing in reality. In the line-graph representation of the network where turning restrictions are included these lead to nodes of degree at least six (indegree=outdegree=3) which make up more than 45% of the dead-end node free network[7]. For a common four-road crossing the three outgoing edges will normally

[7]The exact numbers for the four networks are: Cologne 48.7%, NRW 53.6%, Kansas 59.4%

lie within a radius of about 90°.

Also we know that the incooperation of future costs prefers edges that are directed into the direction of the target node especially at the start of the search. The geometry of paths without crossings will normally not be of a zig-zag type but roughly keep a once chosen direction. Thus, if the search starts out into the direction of the target node, we can even expect to find an edge of appropriate angle at nodes of indegree and outdegree at least two of which there are more than 90% in our dead-end free networks[8].

If the algorithm encounters such an edge of appropriate angle then we can expect the geometric distance $eu(w,t)$ to be smaller than $eu(u,t)$ with high probability since the average length of the edges is small compared to Euclidean distances to the target node along the search path. Only in the near vicinity of the target node there might be the effect that node u is closer to t than node w. But at that point most of the suggested path has already be determined by the algorithm for an average length path.

From these observations we conclude that we can expect to find an appropriate edge in road networks with high probablility and therefore, the derived theoretical bound for the overdo factor will also be of relevance in practice, especially since the algorithmic behaviour of always adding the geometrically closest neighbour to the path might occur even if the sequence of labels is not strictly decreasing.

Practical value for the overdo factor. In order to derive an explicit value for $f_{ov}^{\max}$ in a practical application from the theoretical bound in lemma 4.5.1 we set $x = l(e)/eu(u,t)$ for edge $e = (u,w)$. We assume that $x \leq 1$, which will be true for the most part of the shortest path calculation, only for nodes u near the target node t this condition might be violated. Plugging this into the left hand side of inequality (4.1) yields

$$\frac{x}{(1-\sqrt{1-\sqrt{2}\cdot x+x^2})}\cdot\frac{v_{max}}{v(e)} < f_{ov}$$

$$\Rightarrow \quad label(w) < label(v)$$

Looking at the graph of the function $f(x) = x/(1-\sqrt{1-\sqrt{2}\cdot x+x^2})$ for $x \in [0,1]$ in figure 4.33 we see that the function values lie in between 1.4 and 4.5. We can expect for most nodes scanned during the run of the algorithm that the length of the relevant edge will only be a very small fraction of the Euclidean distance between node u and target t. By looking at the average link length of less than 500 meters for the used networks a ratio of 0.1 seems to be a sufficient upper bound

and California 70.4%.

[8]The exact numbers for the four networks are: Cologne 90.6%, NRW 94.1%, Kansas 99.9% and California 99.6%.

for x for almost all nodes in a shortest path calculation. Therefore, we expect the value of function f to be around 1.5 for almost all nodes during the shortest path search. The ratio of v_{max} and speed $v(e)$ for the used speed models (see table 4.2) is at most 3.3. Thus, we get a bound of approximately five for $f_{ov}^{\max}$, meaning that for overdo factors greater than five we will have a decreasing sequence of labels and the algorithm will add as next node to the found path always a neighbour of the previous scanned node. A value of five for $f_{ov}^{\max}$ is very much in line with the results of our experimental analysis in section 4.3.

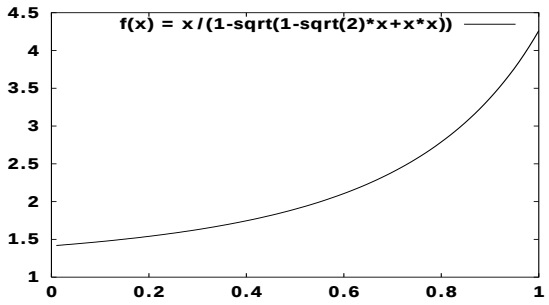


Figure 4.33 Function describing the ratio of $l(e)$ and $eu(u,t) - eu(w,t)$.

In a pure dynamical setting with heavy congestion the ratio of v_{max} and $v(e)$ must be expected to be greater than 3.3 thus leading to a greater bound for a practical overdo factor.

4.5.2 Lower bound for an optimal overdo factor

In this section we derive a lower bound for an optimal overdo factor for the special class of gridgraphs with equal speed on all links. The future costs for a node u used in the A^*-algorithm are calculated as the Euclidean distance between u and target t priced with the maximum speed on any link in the network. The actual shortest path between u and t will normally be longer in geometric length and slower due to the fact that it is not possible to realize the maximum speed on all edges of the path. Thus, an optimal overdo factor can be thought of consisting of two factors (greater or equal to one), one approximating the real geometric length with the Euclidean distance and the other approximating the actual speed with maximum speed. We call the first factor the deviation factor and the second one the speed factor.

By assigning the same speed to all links on the network we concentrate our analysis on the deviation factor and neglect the speed factor. For the uniform speed model the speed factor will be one, while for other speed models or dynamic link weights this factor is at least one. Thus, the derived bound is indeed a lower bound for an optimal overdo factor in gridgraphs even with dynamic speeds. The restriction to gridgraphs is a good approximation for small city-networks. For large area networks as NRW or California one will expect that the Euclidean distance between two nodes is a better approximation than for gridgraphs, making the lower bound for the deviation factor derived in this section an approximation for arbitrary networks with uniform speed on all links.

Let $G = (V, E)$ be a gridgraph with target node $t = (x_t, y_t)$ and $v = (x_v, y_v)$ be an arbitrary node, each given with their coordinates. Let $x = x_t - x_v$ and $y = y_t - y_v$. Since we assume that there is the same speed on all links the length of the shortest path $SP(v, t)$ from v to t is just $|x| + |y|$. For the Euclidean distance between the two nodes we have $r = \sqrt{x^2 + y^2}$. The ratio

$$F = \frac{|x| + |y|}{r}$$

gives the deviation factor for this pair of nodes. To derive the desired lower bound we take r as constant and average the ratio F over the circumference of the circle with radius r in a fine grid.

$$\begin{aligned}
\langle F \rangle &= \frac{1}{2\pi r} \cdot \int_0^{2\pi} \frac{|x| + |y|}{r} \cdot r d\phi \\
&= \frac{1}{2\pi r} \cdot \int_0^{2\pi} r \cdot (|\sin\phi| + |\cos\phi|) d\phi \\
&= \frac{1}{2\pi} \cdot 2 \cdot \int_0^{\pi} [\sin\phi + |\cos\phi|] d\phi \\
&= \frac{1}{\pi} \cdot 2 \cdot \underbrace{\int_0^{\pi} \sin\phi d\phi}_{=2} \\
&= \frac{4}{\pi} \\
&\sim 1.273
\end{aligned}$$

Thus, we can expect that the deviation factor will be about $4/\pi$ in gridgraphs with uniform speed on all links, thereby giving a lower bound $f_{ov}^{\min}$ for the overdo

factor in gridgraphs with arbitrary edge weights. The approximation of the deviation factor will be more accurate if the radius r is large, since then we can expect to find a sufficient number of nodes on the circumference of the circle.

As already mentioned we do expect the deviation factor to be smaller than $f_{ov}^{\min}$ in arbitrary road networks[9]. Experimental tests with a value of 1.273 as overdo factor for 10000 shortest path calculations support this expectation as the results for the maximum relative error in table 4.7 and for the fraction of deficient paths in table 4.8 show for the four different networks.

	Speed model		
Network	Free-flow	Congestion	Uniform
Cologne	0.04	0.05	0.12
NRW	0.06	0.07	0.12
Kansas	0.12	0.14	0.21
California	0.06	0.08	0.15

Table 4.7 Maximum relative errors for overdo factor 1.273.

The results show that for uniform speed the errors are greatest and a high fraction of paths are erroneous. If the edge weights are not uniform, then the speed factor is greater than one and the derived lower bound is more likely to hold. Note also that for the more gridlike network of Cologne we have the best results especially for the number of deficient paths.

We conclude from the bounds derived in this section that an optimal overdo factor should be chosen from the interval $I_{ov} = [1.27, \ldots, 5]$ in a practical application. In section (4.6) we compare the observed ratio of path lengths and future costs from our experimental analysis with the described bounds.

	Speed model		
Network	Free-flow	Congestion	Uniform
Cologne	0.002	0.003	0.46
NRW	0.08	0.09	0.86
Kansas	0.08	0.09	0.96
California	0.12	0.17	0.93

Table 4.8 Fraction of deficient paths for overdo factor 1.273.

[9] For the four networks of our study the empirical deviation factor of the shortest paths was in the interval $[1.2, \ldots, 1.35]$ with high probability.

4.6 A statistical approach to an optimal overdo factor

In section 4.3 we analyzed the shortest paths found by the modified A^*-algorithm with different overdo factors on different road networks of various size. To this end we calculated the exact shortest path using Dijkstra's algorithm. This gave the opportunity to estimate an optimal value for the overdo factor statistically. By backtracking along the exact shortest path from t to s we compared for each encountered node the future costs for v with the travelling time from v to t and also the actual geometric length of the path $P(v,t)$ with the Euclidean distance between v and t. While the ratio of future costs to travel time gives an estimation for the overdo factor, the ratio of geometric length to Euclidean distance will approximate the deviaton factor for which we derived a lower bound for gridgraphs in section 4.5.2.

Figure 4.34 shows how the curves of the described ratios evolve dependent on the Euclidean distance between node v and target node t for the Cologne network. For free-flow traffic the ratio of future costs and travel time (denoted with 'overdo') lies between 1.6 and 2 for distance greater than five kilometers. The ratio of geometric length of the path $P(v,t)$ and Euclidean distance (denoted with 'deviation') is around 1.3 in this interval for free-flow traffic. It is slightly smaller for uniform speed on all links[10]. The deviation factor is very much in line with the theoretically derived lower bound of 1.27 from section 4.5.2. Since the overdo factor is the product of deviation and speed factor we tried to estimate the speed factor from the statistical data we had from the shortest paths calculations. We calculated the ratio of the different road types with respect to geometric length in all the shortest paths . From this we estimated an average speed on a typical path by summation over the road type speeds weighted with this ratio. The speed factor was then chosen as the ratio of this average speed and the maximum speed in the network. For Cologne this resulted in a speed factor of 1.28. Multiplying the curve for the geometric deviation with this speed factor gives a good approximation for the overdo factor as the fourth curve in figure 4.34 shows.

For nodes very close to the target node we observe for free-flow traffic an experimental overdo factor that is significantly higher than for nodes being further away. The reason for this is the high fraction of roads of type zero in our networks. For most target nodes the shortest path will consist of type zero links at the end. For the future costs for the nodes on this stretch the Euclidean distance to the target node, which will be very close to the actual geometric length of the path, is weighted with the maximum speed although the path allows only the min-

[10] Since all links allow the same speed, the ratio for the overdo factor is the same as the one for the deviation factor for the uniform speed model.

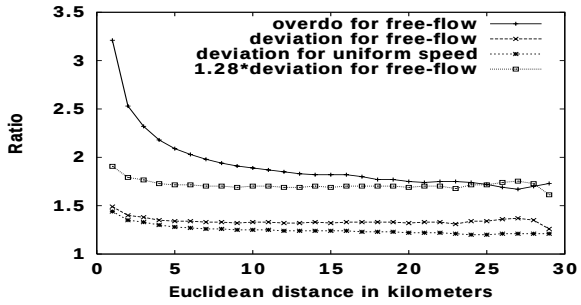


Figure 4.34 Experimental overdo and deviation factors for the Cologne network with free-flow traffic and uniform speed. Multiplying the deviation factor with an empirical speed factor of 1.28 gives a good approximation for the empirical overdo factor.

imum speed of the type zero links. For free-flow traffic the ratio of maximum to minimum speed is 3.3 which is very close to the experimental overdo observed for short Euclidean distances.

In figure 4.35 and figure 4.36 we show the results for the empirical overdo and deviation factor for the network of NRW and California. As in Cologne the deviation factor is around 1.3 while the empirical speed factor is much smaller due to the higher fraction of type four roads in the shortest paths. This leads to a higher average speed which reduces the empirical speed factor. Again the multiplication of the empirical speed factor with the deviation factor gives a good approximation of the empirical overdo factor.

In summary we conclude from the empirical study of the overdo factor in road networks:

- The deviation factor is close to the lower bound $f_{ov}^{min} = 1.27$ derived in section 4.5.2. It is independent of the speeds on the links and of the distance to the target node.
- The overdo factor is below two for nodes not in the vicinity of the target node. For nodes very close to the target node the overdo factor is near the ratio of maximum to minimum speed in the network.
- The speed factor is approximated very well by taking the ratio of maximum speed and average speed where for the average speed the different link types are weighted according to the ratio of this type in the shortest paths.

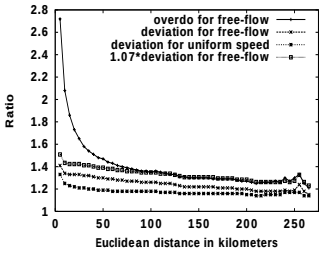


Figure 4.35 Experimental overdo and deviation factors for the NRW network with free-flow traffic and uniform speed. Multiplying the deviation factor with an empirical speed factor of 1.07 gives a good approximation for the empirical overdo factor.

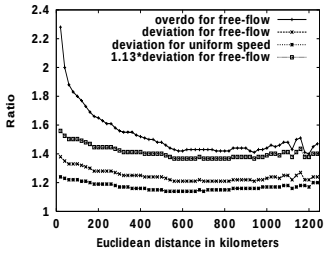


Figure 4.36 Experimental overdo and deviation factors for the California network with free-flow traffic and uniform speed. Multiplying the deviation factor with an empirical speed factor of 1.13 gives a good approximation for the empirical overdo factor.

4.7 Comparison of A* with overdo and the tree heuristic

In chapter 3 we showed that the proposed tree heuristic is a fast shortest path heuristic generating solutions of very good quality. To achieve this the network has to be preprocessed and the memory requirements are significantly higher than for Dijkstra's algorithm or the HISPA heuristic. In contrast, the modified A^*-algorithm with overdo is a heuristic that can be directly applied to the network without any computational work for some preprocessing. In this section we compare the tree heuristic and the modified A^*-algorithm regarding solution quality and run time performance for an 8-way Dijkstra partitioning of the NRW network. For the comparison of suggested paths we chose for the overdo factor values of 1.1, 1.5 and the lower bound 1.273 derived in section 4.5.2.

In figure 4.37 we show the mean relative error for a set of 10000 shortest paths with randomly chosen starting and target nodes. For overdo factor 1.1 all found paths are exact, while for factor 1.273 there is a slightly higher mean error for longer paths than for the tree heuristic. Overdo factor 1.5 results in paths with an expected error above 1% for paths longer than 40 minutes. The error rises up to

about 2.5% for long paths and is significantly higher than for the tree heuristic.

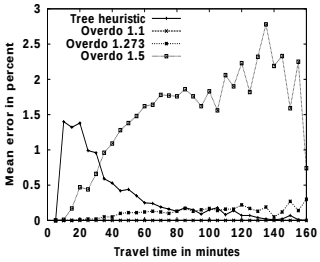


Figure 4.37 Comparison of the mean relative error for the tree heuristic and the A^*-algorithm with overdo in the NRW network.

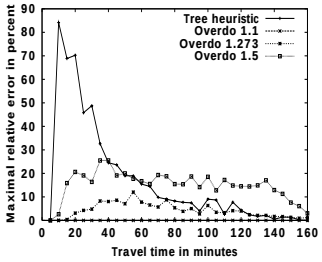


Figure 4.38 Comparison of the maximal relative error for the tree heuristic and the A^*-algorithm with overdo in the NRW network.

For the maximal relative error (see figure 4.38) an overdo factor of 1.273 results in maximal errors of about 10%. This error increases to about 20% for overdo factor 1.5 for almost all path lengths. In contrast, the tree heuristic shows a high relative error of about 90% for very short paths and the error decreases for longer paths below 10% for paths longer than 70 minutes.

For the fraction of deficient paths figure 4.39 shows that for the modified A^*-algorithm the number of erroneous paths increases the longer the shortest path gets. For overdo factor 1.5 more than 50% of the paths are not exact for paths longer than 60 minutes. In contrast, the tree heuristic leads to about 10% deficient paths regardless of path length. For paths longer than 80 minutes this fraction is thereby smaller than for overdo factor 1.273. The fraction of paths with an error greater than 10% is almost zero for overdo factors 1.1 and 1.273 (cf. figure 4.40). Overdo factor 1.5 shows an increasing fraction while it is decreasing for the tree heuristic.

In summary the modified A^*-algorithm generates paths of better quality than the tree heuristic if the overdo factor is close to one. Already for overdo factors greater than about 1.4 the A^*-algorithm with overdo gives a high number of erroneous paths with significantly greater relative errors than the tree heuristic especially for longer paths. This difference in solution quality gets more and more pronounced the greater the overdo factor is chosen[11].

[11] For overdo factors greater than two the fraction of paths with error greater than 10% already rises above 50%.

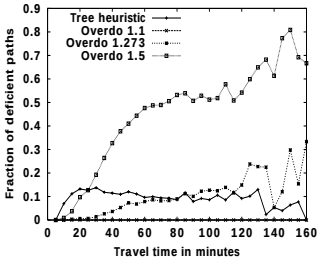


Figure 4.39 Comparison of the fraction of deficient paths for the tree heuristic and the A^*-algorithm with overdo in the NRW network.

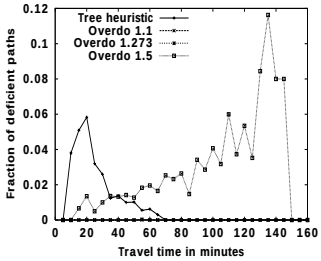


Figure 4.40 Comparison of the fraction of paths with error greater than 10% for the tree heuristic and the A^*-algorithm with overdo in the NRW network.

For the comparison of runtime performance figure 4.41 shows the actual running times of the different algorithms in seconds. We include the classical A^*-algorithm (overdo factor one) in our analysis. For overdo factor one and 1.1 the algorithm is only slightly faster than Dijkstra's algorithm and even slower for long paths due to the additional computation of the Euclidean distances. For overdo factor 1.273 the running times of the A^*-algorithm and the tree heuristic are very similar, while for overdo factor 1.5 the algorithm is the fastest. Looking at the speedup for the number of scanned nodes of the heuristics against Dijkstra's algorithm we see in figure 4.42 that the tree heuristic scans about eight times less nodes than Dijkstra's algorithm. For overdo factor 1.273 the speedup is about ten, but the additional computation of Euclidean distances makes the two heuristics about equally fast. For higher overdo factors the speedup increases significantly while for overdo factor one and 1.1 it is about four.

From the runtime and solution quality analysis we conclude that the tree heuristic and the modified A^*-algorithm with overdo factor close to the derived lower bound of 1.273 from section 4.5.2 have a very similar performance. For both the expected errors are small and the fraction of deficient paths and runtime of the shortest path computation are almost the same. For smaller overdo factors the solution quality increases but the effect on the runtime compared to Dijkstra's algorithm is small or even negative. For overdo factors greater than 1.4 the modified A^*-algorithm outperforms the tree heuristic for the runtime but the suggested paths must be expected to show significantly greater errors.

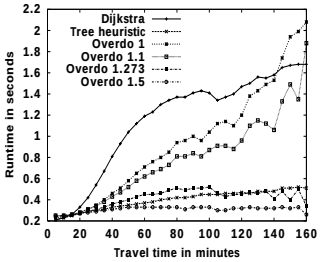


Figure 4.41 Comparison of the running times for the tree heuristic and the A^*-algorithm with overdo in the NRW network.

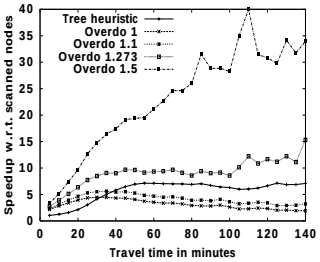


Figure 4.42 Comparison of the speedup w.r.t to scanned nodes for the tree heuristic and the A^*-algorithm with overdo in the NRW network. For this the number of scanned nodes of Dijkstra's algorithm is divided by this number for the heuristics.

Chapter
Five

Sensitivity Analysis for Shortest Paths and its Application to the k-Shortest Path Problem

The robustness of a generated solution is a natural question that arises for optimization problems in networks with a time-dependent cost function (see [44] for an extensive online bibliography of sensitivity analysis and [32] for recent advances). This is especially true for the various applications of the routing problem in road networks with dynamic edge weights. For the individual online routing a recommended path should ideally be actualized whenever a dynamic change of the edge weights occurs. If the routes for all users of such an online-routing system are recalculated, the best implementations of Dijkstra's algorithm or an application of even faster algorithms like the tree heuristic or the modified A^*-algorithm might not succeed in sufficiently short response times for a practical application. Sensitivity information for already calculated paths can help to reduce the computational effort for the online routing in two ways. On the one hand unnecessary recalculations can be avoided if the dynamic changes of the edge weights do not alter the shortest path and this conclusion can be drawn from the sensitivity data. On the other hand a complete recalculation of a route might be avoided if the robustness information allows a quick update of the path in case it is affected by the change of the edge weights. Information about the robustness of shortest paths is also useful for the routing problem in microsimulations of traffic, where theoretically the routes of all drivers have to be calculated in each interation.

In this chapter we present an algorithm to determine edge tolerances for a shortest path in a network. For edges of the shortest path the tolerance shows how much the weight of the edge can increase such that the edge is still included in the shortest path. In contrast, for edges that are not on the shortest path the tolerance gives the necessary decrease of the edge weight in order to make the edge part of the shortest path. Next to the applications of such sensitivity data described above we use these

edge tolerances to determine alternatives for the shortest path. This is closely related to the k-shortest path problem, but in our setting the alternative paths derived from the edge tolerances are not necessarily the k shortest paths but can incorporate other useful requirements such as a greater diversity or bypassing of specific roads. In an online-routing application our method allows to present each user a set of meaningful alternatives which can even be individualized. For an application in the traffic simulation our method gives a different approach to generate an initial set of routes for each driver.

5.1 Edge tolerances for the one-to-one shortest path problem

If a shortest path $SP(s,t)$ between a starting node s and a target node t has been calculated in a weighted graph $G = (V,E,c)$ this path might be affected by dynamic changes of the edge weights. Either the weights of some edges on the path increase leading to a shorter path between s and t which avoids these edges. Or the weights of some edges that are not part of $SP(s,t)$ decrease making it cheaper to use these edges. A decrease of the weight of a shortest path edge or an increase of the weight of a non-shortest path edge will never alter the shortest path. Although for an application in road networks all edge weights will be positive, we assume for the moment that the cost function can take arbitrary values from $\mathbb{R}$, but that the graph has no negative cycles.

More formally for a path $SP(s,t)$ between two nodes s and t in a graph $G = (V,E,c)$ we are interested in two quantities $\delta^+(e) \geq 0$ and $\delta^-(e) \leq 0$ for each edge e of G, such that for a change δ of the edge weight with $\delta^- \leq \delta \leq \delta^+$ the path $SP(s,t)$ is still the optimal path between s and t. δ^- is called the lower tolerance and δ^+ the upper tolerance of the edge with respect to a shortest path $SP(s,t)$. For the problem of finding a shortest path between nodes s and t in road networks the edge weights $c(e)$ are given as the current travel time on each link when the starting node of the edge is reached and the shortest path $SP(s,t)$ is calculated on the basis of these edge weights. The two quantities δ^- and δ^+ for each edge e of G then define an interval

$$[c(e) + \delta^-, c(e) + \delta^+]$$

such that $SP(s,t)$ is a shortest path between s and t as long as the weight of the edge lies in this interval. Note that this interval of edge tolerance is defined for each edge individually, i.e. the weights of all other edges are assumed to be fixed. The concept of edge tolerances for a shortest path presented here is closely related to that of edge tolerances for shortest path trees in [94].

We already observed that $\delta^-(e) = -\infty$ for an edge e lying on the shortest path $SP(s,t)$ and $\delta^+(e) = +\infty$ if $e \notin SP(s,t)$. In order to determine the remaining tolerances we need some more notation. Let T_s be a shortest path tree in G for starting node s and T_t a shortest path tree for node t on the reversed edge set $\overleftarrow{E}$ of G, i.e. $\overleftarrow{E} = \{(v,u) : (u,v) \in E\}$. In T_t the time reaching t from v is given for each node v. If an edge $e_0 = (u,v) \in SP(s,t)$ is removed from the tree T_s then the set of nodes decomposes into two disjoint subtrees, one of which contains node s and the other contains node t. The same applies for the shortest path tree T_t and thus the following node sets are well-defined:

$$V(T_s(e_0)) = \{w \in V : s \text{ and } w \text{ lie in the same subtree of } T_s \setminus \{e_0\}\} \quad (5.1)$$

$$\overline{V(T_s(e_0))} = \{w \in V : t \text{ and } w \text{ lie in the same subtree of } T_s \setminus \{e_0\}\}$$

$$V(T_t(e_0)) = \{w \in V : t \text{ and } w \text{ lie in the same subtree of } T_t \setminus \{e_0\}\} \quad (5.2)$$

$$\overline{V(T_t(e_0))} = \{w \in V : s \text{ and } w \text{ lie in the same subtree of } T_t \setminus \{e_0\}\}$$

For a node $v \in V$ let $d_s(v)$ be the travel time from s to v in the shortest path tree T_s and $d_t(v)$ the time it takes to reach t from v in T_t. Once the trees T_s and T_t are calculated, $d_s(v)$ and $d_t(v)$ are known for each node v. Also let $d(s,t) = d_s(t) = d_t(s)$ be the length of the shortest path between s and t. Note that the shortest path $SP(s,t)$ is included in both shortest path trees[1]. Then the reduced costs for each edge $e = (u,v)$ are defined as

$$\Delta(e) = d_s(u) + c(e) + d_t(v) - d(s,t). \quad (5.3)$$

For edges of the shortest path these reduced costs are zero and the optimality criterion for the shortest path requires that $\Delta(e) \geq 0$ for all non-path edges $e \notin SP(s,t)$. For these edges $\Delta(e)$ gives the additional costs of using the path $P(s,u,v,t)$[2] instead of $SP(s,t)$. Thus, $\delta^-(e) = -\Delta(e)$ for all $e \notin SP(s,t)$. The remaining upper tolerance δ^+ for edges on the shortest path can be calculated by using the node sets defined above, which we present together with the other tolerances in the following lemma.

Lemma 5.1.1 *Let $SP(s,t)$ be a shortest path between nodes s and t in a weighted graph $G = (V,E,c)$ and define $\Delta(e)$ for each edge $e \in E$ as in (5.3).*

[1] We assume in the following that the shortest path from s to t is the same in both T_s and T_t. This can be achieved by starting Dijkstra's algorithm for tree T_t with a node v of the shortest path $SP(s,t)$ pre-labeled with $d(s,t) - d_s(v)$. These are the correct labels in T_t for those nodes and they will therefore not be updated during the algorithm, giving the desired s-t-path in T_t.

[2] $P(s,u,v,t)$ is the path in G that consists of the shortest path from s to u in T_s, the edge $e = (u,v)$ and the shortest path from v to t in T_t.

If $e \notin SP(s,t)$ *then*

$$\begin{aligned} \delta^-(e) &= -\Delta(e) \\ \delta^+(e) &= \infty \end{aligned}$$

If $e_0 \in SP(s,t)$ *then*

$$\begin{aligned} \delta^-(e_0) &= -\infty \\ \delta^+(e_0) &= \min\{\Delta(e) : e=(u,v),\, u \in V(T_s(e_0)),\, v \in V(T_t(e_0)), e \neq e_0\} \end{aligned} \quad (5.4)$$

Proof. From the above discussion it remains to proof the upper tolerance $\delta^+(e_0)$ for an edge $e_0 \in SP(s,t)$. Increasing the cost $c(e_0)$ of a path edge e_0 by an amount $\delta > 0$ affects the distances of nodes in the shortest path trees T_s and T_t as follows:

$$\hat{d}_s(v) = \begin{cases} d_s(v) & : \text{ if } v \in \overline{V(T_s(e_0))} \\ d_s(v)+\delta & : \text{ if } v \in V(T_s(e_0)) \end{cases}$$

$$\hat{d}_t(v) = \begin{cases} d_t(v) & : \text{ if } v \in \overline{V(T_t(e_0))} \\ d_t(v)+\delta & : \text{ if } v \in V(T_t(e_0)) \end{cases}$$

Here $\hat{d}_s(v)$ and $\hat{d}_t(v)$ signify the perturbed distances of nodes in the shortest path trees T_s and T_t due to the perturbation of edge e_0. These changes of the distances affect the quantities $\Delta(e)$ for an edge $e=(u,v) \neq e_0$, which enter the optimality critertion for the shortest path:

$$\hat{\Delta}(e) = \begin{cases} \Delta(e) & : \text{ if } u \in V(T_s(e_0)),\, v \in V(T_t(e_0)) \\ \Delta(e)+\delta & : \text{ if } u \in \overline{V(T_s(e_0))} \\ \Delta(e)+\delta & : \text{ if } v \in \overline{V(T_t(e_0))} \end{cases}$$

Thus, for edges $e=(u,v)$ with $u \in V(T_s(e_0))$ and $v \in V(T_t(e_0))$ $\Delta(e)$ does not change. Intuitively for these edges the alternative path does not use the path edge e_0. The minimum Δ of all those edges gives the maximal range for which the optimality of $SP(s,t)$ is preserved. □

Lemma 5.1.1 shows that the edge tolerances can be determined if the two shortest path trees T_s and T_t are given. The calculation of the upper tolerance of a path edge involves the minimization over an appropriate cutset. However, determining cutsets can be computationally burdensome. To avoid this, we will proceed along a different approach. For each edge $e \notin SP(s,t)$ we update the upper tolerance for all those edges $e_0 \in SP(s,t)$ for which e is in the appropriate cutset of e_0.

The shortest paths in a shortest path tree T with root-node r define a partial ordering, the root ordering, of the nodes of T. We write $v_1 \preceq v_2$ for nodes $v_1, v_2 \in T$

if $v_1 \in SP(r, v_2)$. Any two nodes $v_1, v_2 \in T$ have a *greatest lower bound* (*GLB*) $v_0 = GLB(v_1, v_2)$, with respect to the root ordering, namely the greatest node v_0 in the root order such that $v_0 \in SP(r, v_1)$ and $v_0 \in SP(r, v_2)$. For a shortest path $SP(s, t)$, node $v \notin SP(s, t)$ and shortest path trees T_s on the edge set E and T_t on the edge set $\overleftarrow{E}$ we define

$$m_s(v) = GLB(v, t) \text{ w.r.t. root ordering in } T_s \quad (5.5)$$

$$m_t(v) = GLB(v, s) \text{ w.r.t. root ordering in } T_t \quad (5.6)$$

Intuitively $m_s(v)$ ($m_t(v)$) is the first node encountered on the shortest path $SP(s, t)$ when backtracking from v to s (t) in T_s (T_t). According to lemma 5.1.1 to determine the upper tolerance $\delta^+(e_0)$ for an edge $e_0 \in SP(s, t)$ we have to minimize over the cutset of all those edges $e = (u, v)$ for which $u \in V(T_s(e_0))$ and $v \in V(T_t(e_0))$ where $V(T_s(e_0))$ and $V(T_t(e_0))$ are given as in (5.1) and (5.2). From equalities (5.5) and (5.6) we can express the upper tolerance of a path edge $e_0 = (u_0, v_0)$ as

$$\delta^+(e_0) = \min\{\Delta(e) : m_s(u) \preceq u_0 \text{ and } m_t(v) \preceq v_0,\ e = (u, v), e \neq e_0\} \quad (5.7)$$

From equality (5.7) we can determine the upper tolerances of edges on the shortest path by updating for a non-path edge $e = (u, v)$ the minimum along the subpath $SP(m_s(u), m_t(v))$ of $SP(s, t)$. Note that if $v_1 \preceq v_2$ w.r.t. root ordering in T_s, then $v_2 \preceq v_1$ w.r.t. root ordering in T_t. Therefore, the subpath $SP(m_s(u), m_t(v))$ is well-defined. In figure 5.1 we give a short description of our algorithm for determining edge tolerances for the one-to-one shortest path problem, using the notation introduced in this section.

For the complexity of the algorithm the calculation of the shortest path trees takes time at most $\mathcal{O}(|V| \cdot \log|V| + |E|)$. For the first loop over all non-path edges we note that the backtracking to reach $m_s(u)$ and $m_t(v)$ must not be performed for each edge individually. If $n_1 \preceq n_2$ w.r.t. root ordering in T_s (T_t) for two nodes $n_1, n_2 \notin SP(s, t)$ then $m_s(n_1) = m_s(n_2)$ ($m_t(n_1) = m_t(n_2)$). Thus, we can halt the backtracking for some node n_1 as soon as we reach a node n_2 for which $m_s(n_2)$ ($m_t(n_2)$) has been determined, setting $m_s(n) = m_s(n_2)$ ($m_t(n) = m_t(n_2)$) for all nodes $n \in SP(n_2, n_1)$. The amortized costs of calculating $m_s(u)$ and $m_t(v)$ for each non-path edge $e = (u, v)$ are therefore $\mathcal{O}(|E|)$. This is also the reason, why we presented the backtracking and the calculation of the upper tolerances in two separate loops. The second loop over all non-path edges has a worst-case complexity of $\mathcal{O}(|E| \cdot |V|)$ since the shortest path can have up to $n - 1$ edges, giving an overall complexity of $\mathcal{O}(|E| \cdot |V|)$ for the algorithm.

5.1.1 Edge tolerances for shortest path trees

Robustness of shortest path trees for dynamic changes of edge weigths has been studied by various authors, see for example [7, 16, 45, 58, 94, 96]. In [94] an al-

```
procedure Edge Tolerances
  begin
    Calculate the shortest path tree T_s for node s on edge set E;
    Calculate the shortest path tree T_t for node t on edge set Ē;
    foreach e_0 ∈ E, e_0 ∈ SP(s, t)
    begin
      δ⁻(e_0) := −∞;
      δ⁺(e_0) := ∞;
    end
    foreach e = (u, v) ∈ E, e ∉ SP(s, t)
    begin
      Δ(e) := d_s(u) + c(e) + d_t(v) − d(s, t);
      δ⁻(e) := −Δ(e);
      δ⁺(e) := ∞;
      Find m_s(u) by backtracking along T_s;
      Find m_t(v) by backtracking along T_t;
    end
    foreach e = (u, v) ∈ E, e ∉ SP(s, t)
    begin
      foreach e_0 ∈ E, e_0 ∈ SP(m_s(u), m_t(v))
      begin
        δ⁺(e_0) := min{δ⁺(e_0), Δ(e)};
      end

    end

  end
```

Figure 5.1 Algorithm to determine edge tolerances for the one-to-one shortest path problem.

gorithm due to Dantzig [16] is presented which gives upper and lower edge tolerances for a shortest path tree. Very similar to our approach for a shortest path these quantities measure how much the weight of an edge can vary without changing the shortest path tree for some node s, thereby keeping the edge weights of all other edges fixed.

To state the main result about these tolerances we need the following notation: Let T_s be a shortest path tree for starting node s with shortest path distances $d(v)$ for each node $v \in V$. Define the reduced costs for an edge $e = (u,v) \in E$ as $D(e) = d(u) + c(e) - d(v)$. Removing a tree edge $e_0 = (u_0, v_0)$ from T_s partitions the node set into two components $N(e_0)$ with $u_0 \in N(e_0)$ and $\overline{N}(e_0)$ with $v_0 \in \overline{N}(e_0)$. For a tree edge $e_0 \in T_s$ two cut-sets can be defined, i.e.

$$\begin{aligned} C^+(e_0) &= \{e = (u,v) \in E \,:\, u \in N(e_0), v \in \overline{N}(e_0)\}, \\ C^-(e_0) &= \{e = (u,v) \in E \,:\, u \in \overline{N}(e_0), v \in N(e_0)\}. \end{aligned}$$

Lemma 5.1.2 *Let T_s be a shortest path tree for node s in a weighted graph $G = (V,E,c)$ with distances $d(v)$ for nodes $v \in V$ and reduced costs $D(e)$ for edges $e \in E$.*

If $e \notin T_s$ then

$$\begin{aligned} \delta^-(e) &= -D(e) \\ \delta^+(e) &= \infty \end{aligned}$$

If $e_0 \in T_s$ then

$$\delta^-(e_0) = \max\{-D(e) \,:\, e \in C^-(e_0)\}, \tag{5.8}$$

$$\delta^+(e_0) = \min\{D(e) \,:\, e \in C^+(e_0), e \neq e_0\} \tag{5.9}$$

Intuitively equalities (5.8) and (5.9) show that the costs of a tree edge e_0 can be increased for a shortest path tree as long as there is no cheaper path over a non-tree edge in the equally directed cutset. The costs of e_0 can be decreased until a path to some node using e_0 and an edge of the co-directed cutset becomes cheaper than the present path to this node. Each non-tree edge $e = (u,v)$ closes an (undirected) cycle $\{P(m,u), e, P(m,v)\}$ in the tree where $m = GLB(u,v)$ w.r.t. the root ordering and $P(x,y)$ is the distinct tree path between x and y. From this we get for $e = (u,v) \notin T_s$ and $e_0 \in T_s$

$$e \in C^+(e_0) \;\Leftrightarrow\; e_0 \in P(m,v), \tag{5.10}$$

$$e \in C^-(e_0) \;\Leftrightarrow\; e_0 \in P(m,u), \tag{5.11}$$

These last two equations allow to determine the edge tolerances for tree edges without a direct minimization and maximization over the cutsets. Instead, for each non-tree edge the minima and maxima are updated along the two paths to the greatest lower bound m. Each edge in Dijkstra's algorithm is scanned exactly once and if it is not added to the shortest path tree, the edge is a non-tree edge, for which the two paths to m already have been determined. Thus, the edge tolerances can be calculated within Dijkstra's algorithm, which was first observed by Dantzig in [16] (a pseudo-code description of the algorithm is given in [94]). The worst-case complexity of the algorithm is $\mathcal{O}(|E| \cdot |V|)$, since the cycle which is closed by each non-tree edge can be of length $|V|$.

There are two main differences between edge tolerances for a shortest path and a shortest path tree. For the latter a decrease of an tree edge might change the shortest path tree, while a decrease of a path edge will not alter the shortest path. The reduced costs $D(e)$ for a non-tree edge are known at that point of the algorithm, when the edge is scanned. For the reduced costs $\Delta(e)$ of a non-path edge both shortest path trees T_s and T_t must have been calculated in order to determine $\Delta(e)$. Therefore, these costs cannot be determined within the tree calculation.

5.1.2 Edge tolerances in road networks

In lemma 5.1.1 we showed that a dynamic change of an edge weight will alter the shortest path between two nodes only if a non-path edge decreases by more than $\Delta(e)$ or a path edge increases by more than $\delta^+(e)$. If the graph in question is a road network from a practical real-world application additional properties of these quantities have to be considered. Obviously the weight of an edge in a road network will always be positive. But then the weight of a non-path edge e can decrease by at most $c(e)$, which means that edges with $\Delta(e) \geq c(e)$ are no realisitic alternative for the shortest path between s and t. For the lower tolerance of non-path edges we therefore state:

If $e \notin SP(s,t)$ then

$$\delta^-(e) = \begin{cases} -\Delta(e) & : \text{ if } \Delta(e) < c(e) \\ -\infty & : \text{ if } \Delta \geq c(e) \end{cases} \tag{5.12}$$

However, even if $c(e) \leq \Delta(e)$, these edges have to be considered for the minimum in δ^+ of appropriate path edges, since there is no obvious bound for a maximal increase for path edges.

In an online-routing application we do not want to generate routes that contain a cycle, since these are not acceptable from a user point of view. Therefore, we have to exclude non-tree edges from the calculation of the upper tolerances of the tree edges, for which the alternative path from s to t contains a cycle. Every non-path edge closes a cycle in the shortest path trees T_s and T_t, but these cycles do not

necessarily lead to an alternative s-t-path containing a cycle (see figure 5.2 (d)). There are three ways for a non-path edge $e = (u, v)$ to belong to an alternative s-t-path $\tilde{P}(s, t)$ with cycle $C_{\tilde{P}}$.

(a) e closes a cycle in T_s, that does not intersect with $SP(s, t)$ (see figure 5.2 (a)). Then $v \in P(m_s(u), u)$ in T_s.

(b) e closes a cycle in T_t, that does not intersect with $SP(s, t)$ (see figure 5.2 (b)). Then $u \in P(m_t(v), v)$ in T_t.

(c) The shortest path $SP(s, t)$ and $C_{\tilde{P}}$ intersect (see figure 5.2 (c)). In this case $d_s(m_t(v)) < d_s(m_s(u))$, that is the node $m_t(v)$ comes before $m_s(u)$ on the shortest path.

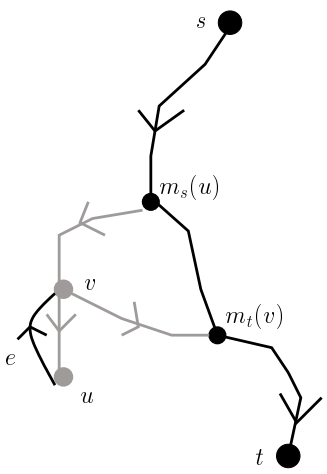
The condition for case (c) can be checked directly. For case (a) we need to backtrack in T_s from node u until we either reach v or $m_s(u)$. If $v \in P(m_s(u), u)$ in T_s, then $m_s(u) = m_s(v)$ and $d_s(v) < d_s(u)$. By checking these two equalitites we can avoid the time consuming backtracking for most non-path edges. There are similar equalities for case (b) which allow to speed up the cycle tracing. Any non-path edge, which gives an s-t-path with a cycle as in (a), (b) or (c), will not be considered for the upper tolerance of path edges and its lower tolerance is set to $-\infty$. Thereby, the edge tolerances calculated with our algorithm give practicable routing alternatives.

For the runtime performance of the algorithm we already showed that it has a worst-case complexity of $\mathcal{O}(|V| \cdot |E|)$. We tested the algorithm for 10000 shortest paths between randomly chosen nodes in the road network of Northrhine-Westphalia with 457124 nodes and 1040687 edges. The test was performed on a Sun Enterprise E450 with 1.15 GByte RAM and four UltraSparc-II CPU's with 400 MHz, running under Solaris 2.7. The code was compiled using the GNU compiler version 2.8.9 with O4 optimization flag.

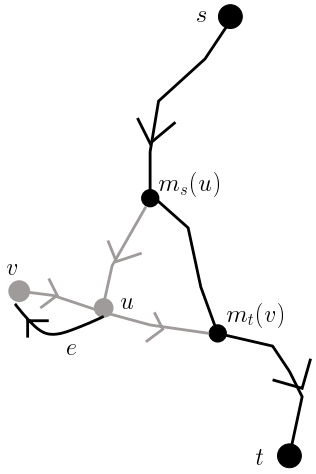
In figure 5.3 we show statistics about the described algorithm for edge tolerances in road networks. The runtime of the algorithm is below eight seconds and therefore about half as fast as a one-to-all Dijkstra algorithm. The number of path edges, for which upper tolerances are determined is between 100 and about 300 for the longest paths. The number of non-path edges, for which the lower tolerance is not equal to $-\infty$ is below 10000. However, we already noted, that those edges still have to be considered for the upper tolerances of the path edges unless the alternative paths with these edges include a cycle.

5.2 An application to k-shortest paths

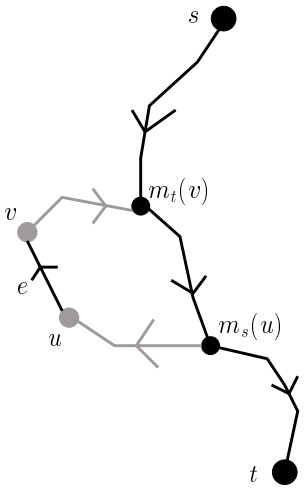
By not including non-path edges in the minimization of 5.7, which fulfill any one of the conditions (a), (b) or (c), we get tolerances for the path edges of $SP(s, t)$ for



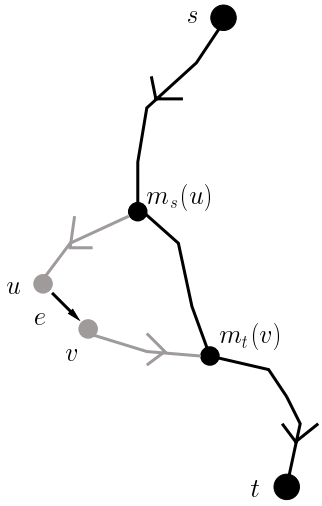
(a) Alternative path over non-path edge e with cycle in T_s.



(b) Alternative path over non-path edge e with cycle in T_t.



(c) Alternative path over non-path edge e with cycle intersecting $SP(s,t)$.



(d) Alternative path over non-path edge e without cycle.

Figure 5.2 Different alternative paths for a non-path edge e. The shortest path and edge e are shown in black, the alternative path in grey.

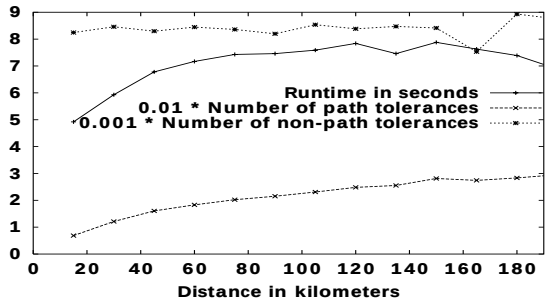


Figure 5.3 Performance of the edge tolerance algorithm. The y-axis represents different attributes.

acyclic paths which can be used to give drivers a set of meaningful alternatives for the shortest path between two nodes. For every path edge the upper tolerance gives the additional costs of an alternative *s*-*t*-path, that uses that non-path edge, which determines the minimum of (5.7) for the path edge. By giving a set of alternative paths our algorithm is closely related to the *k*-shortest path problem.

This network programming problem has been studied as long as the classical shortest path problem [22, 51, 101], but without a similar intensity, despite its obvious practical interest (see [23] for a very complete online bibliography). The problem is typically divided into two classes: the unconstrained problem, where the objective is to find the k shortest paths between two nodes and the constrained problem, where additional constrains, such as cycle-free or edge-disjoint must be satisfied.

For the unconstrained problem in a directed graph $G = (V, E)$ with $|V| = n$ and $|E| = m$, Fox [30] presented a method based on Dijkstra's algorithm which with more recent improvements in priority queue data structures [31] takes time $\mathcal{O}(m + kn \log n)$. Other algorithms use a path deletion method [5, 72] or vector/-matrix methods [69, 75]. Recently, Eppstein proposed an approach with a binary heap at each node where edges are kept, that are not part of the shortest path, giving an $\mathcal{O}(m + n \log n + k)$ algorithm [24].

For the problem with the additional constraint that the k shortest paths are required to be loopless one of the first algorithms is due to Clarke et al. [14]. Its main weakness of storing a greatly varying number of candidate paths, was overcome by Yen's method [101] for an $\mathcal{O}(kn^3)$ algorithm with a generally smaller number of candidate paths. This algorithm shows a very good performance in practice [85].

For undirected graphs the method of Katoh et al. [63] has the lowest worst case complexity $\mathcal{O}(kn^2)$ of all known algorithms. Algorithms for the constrained problem with arc capacities are dealt with in [9] and [90].

By appropriately choosing the k minimal upper tolerances of path edges one gets a set of k alternative paths between s and t. With 'appropriately' we mean that the chosen tolerances should lead to different paths. For two path edges f_1, f_2 with $\delta^+(e_1) = \delta^+(e_2)$ the corresponding non-path edges $e_1 = (u_1, v_1), e_2 = (u_2, v_2)$ may belong to the same alternative path, if $m_s(u_1) = m_s(u_2)$ and $m_t(v_1) = m_t(v_2)$. If one of these equalities does not hold, then the two paths will be different,[3] since the nodes m_s and m_t are the deviation nodes of the alternatives from the shortest path. Such a set of k different paths must not be the set of the k shortest paths, as the example in figure 5.4 shows.

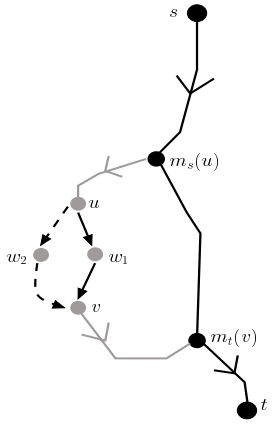


Figure 5.4 Example that the upper tolerances for path edges do not give the set of k shortest paths. If the upper tolerance of all path edges between $m_s(u)$ and $m_t(v)$ is determined by the reduced cost of non-path edge (u, w_1), then the alternative using the edge (u, w_2) can not be deduced from the tolerances.

However, from the calculation of the tolerances the deviation nodes $m_s(u)$ and $m_t(v)$ are known for alternative paths using some non-path edge $e = (u, v)$. These

[3] The reverse must not be true: Even if $m_s(u_1) = m_s(u_2)$ and $m_t(v_1) = m_t(v_2)$, the two paths may differ, although to check this, the paths have to be traversed.

nodes allow to control the alternatives which are given as routing recommendations in a more sophisiticated way than algorithms, which calculate the k shortest (loopless) paths between two nodes s and t. The k shortest paths found by these algorithms tend to be very similar especially in realworld applications [91]. For these, small deviations which bypass for example a longer link by two smaller links or which take the next link after a highway exit and returning on the opposite direction[4] give alternatives, that have slightly higher costs than the shortest path, but are not satisfactory from a routing perspective. Travelers and operators of such systems will view those paths not as true alternatives.

One approach to avoid this phenomenon is link elimination, which excludes specific links of the shortest path. The disadvantage is that in many instances the traveler/operator will not be able to identify such 'bad' links, but rather prefer a 'different' path, where different is not clearly defined. Scott et al. proposed a different approach, where an alternative path has not many links in common with the best path [91]. There approach leads to a constrained linear programming problem, which is solved using Lagrangian relaxation. For this the number of links, in which the two paths differ must be known in advance.

By using the information about the deviation nodes m_s and m_t of each non-path edge, it is possible to generate alternative paths in a more flexible way. For example if a whole stretch of the shortest path between two nodes u_0 and v_0 shall be avoided, then only those non-path edges $e = (u, v)$ must be considered for the calculation of the path tolerances, for which $m_s(u) \leq u_0$ and $m_t(v) \geq v_0$. Or by excluding non-path edges $e = (u, v)$, for which $|m_t(v) - m_s(u)| \leq c$ for some bound c, alternatives that are just small deviations of the shortest path are avoided. In figure 5.5 we give an example of such alternative paths for the road network of Cologne, that seem to be more acceptable for realworld routing applications. Figure 5.6 shows three alternatives for which path edges with smallest upper tolerance are avoided.

[4] This is a loopless path due to individual links for each highway direction.

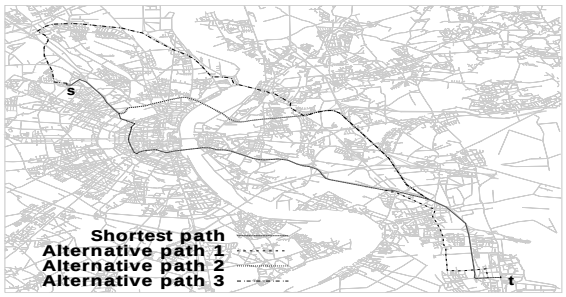


Figure 5.5 Alternative routes for a shortest path in Cologne, where the non-path edges fulfill $|m_t(v) - m_s(u)| \leq 0.25 \cdot d(s,t)$. The shortest path has a length of 17.9 minutes, the three alternatives have a length of 18.1, 20.2 and 23.9 minutes.

Figure 5.6 Alternative routes for a shortest path in Cologne of length 17.9 minutes, where path edges with smallest upper tolerance are avoided. The alternatives bypass only a very small stretch of the shortest path (shown in circles). The length of all three alternatives is 18 minutes.

Chapter
Six

Summary and Outlook

In this work we have studied different algorithms for shortest path problems in large road networks. Fast implementations of the classical Dijkstra algorithm calculate a shortest path between a starting and a target node in a few seconds on a Sun Enterprise E450 for the network of California consisting of almost four million edges. In applications like online-routing and dynamic traffic assignment for traffic microsimulations even faster route generation algorithms are needed to enhance the practical use of these instruments.

This can be achieved by employing heuristical methods which incorporate the special structure of road networks. A path generated by such an algorithm must not necessarily be the optimal path, but if it is of sufficient quality a heuristical solution will be acceptable for many practical applications.

In this thesis we presented a new heuristic based on the similarity of shortest path trees to generate routes in road networks. On the networks considered in this treatise this so-called tree heuristic outperforms Dijkstra's algorithm by a factor of at least four with respect to running time, finding paths of very good quality.

A different approach for a faster generation of shortest paths uses the geometrical information of the edges in road networks to calculate future costs. Multiplying these future costs with an overdo factor turns the algorithm into a heuristic. We analyzed this modified A^*-algorithm on different road networks and derived theoretical bounds for an optimal overdo factor.

In chapter 2 we introduced the shortest path problem for the special class of road networks and shortly reviewed various implementations of Dijkstra's algorithm and some label-correcting algorithms. An empirical test showed that the fastest of these algorithms find a shortest path in the network of Northrhine-Westphalia with about one million edges in less than a second and within a few seconds in the network of California with almost four million edges on a fast multiprocessor machine. In chapter 3 we presented the tree heuristic which partitions the network into a number of equally sized classes and generates a searchgraph for each partition class. The searchgraphs have a tree-like structure and are much

sparser than the original network, leading to very fast shortest path calculations by applying a backward Dijkstra algorithm. We studied different methods for partitioning the network into k clusters. Our analysis showed that the connectivity of the partition classes is of great importance for the quality of found paths. The partitioning based on k shortest path trees proved to be well applicable by generating paths with an expected error below 1% for the networks of NRW and California.

A comparison of the runtime performance of the tree heuristic with Dijkstra's algorithm, a bidirectional variant of Dijkstra, the HISPA heuristic and the A^*-algorithm proved that the tree heuristic has the best running times of all algorithms. With respect to the number of scanned nodes only the HISPA heuristic performs better, but generates paths of significantly worse quality. Also, the runtime of the tree heuristic is not much affected by the length of the shortest path. Since we did not test the tree heuristic with dynamically changing weights on the edges, it would be interesting to study the influence of the partitioning of the network on the quality of paths in such a dynamic setting. Also, the correlation between the quality of the heuristic and the edge cut of the partition can be analyzed in more detail. For a practical application of the tree heuristic methods to reduce the memory requirements of the algorithm should be considered.

In chapter 4 we studied the A^*-algorithm with overdo on four road networks using three speed models. We could not identify significant differences for the quality of paths found by the algorithm on these networks. Instead, we observed for all networks an algorithmic behaviour that favours reachability over shortest path distances if the overdo factor increases. More precisely, for high overdo factors the algorithm chooses always a neighbour of the previously scanned node as next node on the path. This leads to a high fraction of paths with errors greater than 100%. We derived a theoretical bound for the overdo factor, such that higher factors will always lead to the described algorithmic behaviour. For a practical application we expect a value of about five for this bound under some assumptions on the structure of the network and the observed speeds. For small overdo factors between one and 1.5 the modified A^*-algorithm finds the exact path or at least paths of very good quality. While the exact A^*-algorithm can be even slower than Dijkstra's algorithm for long paths due to the additional calculation of the future costs, a value of 1.5 gives a considerable speedup. Since we used static speed models for our analysis, a verification of our results in a dynamical setting would be of great interest.

Finally, in chapter 5 we presented an algorithm to calculate edge tolerances for a shortest path with respect to one changing edge weight. An application of this algorithm on the network of NRW showed that only for a small fraction of the edges a changing edge weight might influence the shortest path. The information about the edge tolerances additionally allows to generate alternatives for the shortest path that are of a more meaningful diversity for a practical application than those calculated by standard k-shortest path algorithms.

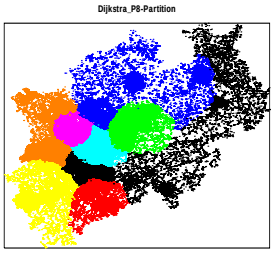
Appendix
A

Partitions of NRW

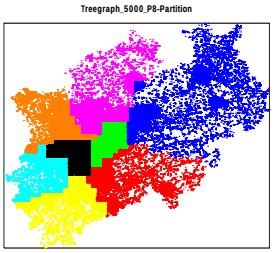
This appendix shows some coloured plots of different partitionings of the road network of Northrhine-Westphalia (NRW) into eight classes. The NRW network has 457124 nodes and 1040687 edges. The plots only show the nodes of the network but no edges. Therefore, the connectivity of the subclasses cannot be deduced directly from the plots.

For the partitioning methods we used the Dijkstra partitioning (cf. 3.4.1.1), the METIS software-library [62] (cf. 3.4.1.2) and the treegraph partitioning (cf. 3.4.1.3). The Dijkstra partitioning (see figure A.1 (a)) always generates connected subclasses and as the plot shows, the classes are of rather regular geometric shape. For the treegraph partitioning (cf figure A.1 (b)) there is one subclass, which is not connected decomposing into two components. Since one of these components is very small, it cannot not identified from the plot. Due to the geometric partitioning using a grid of length $l = 5000m$ the border of the classes is made of parallels to the x- und y-axis.

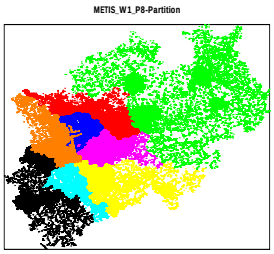
The four partitionings using the METIS software-library result in partitions of various connectivity. Weight functions W1 and W4 (cf. 3.4.1.2) lead to connected subclasses (see figures A.1 (c) and (f)), while the other two weight functions result in classes that in some cases have several hundred connected components (see figures A.1 (d) and (e)). Accordingly, the geometric shape of the subclasses of the connected partitionings are more regular than the others, which are partly very scattered.



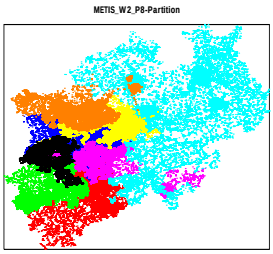
(a) 8-way Dijkstra partition of NRW.



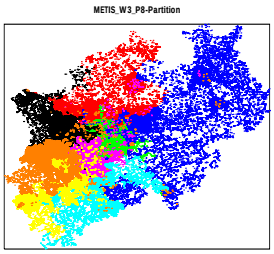
(b) 8-way Treegraph partition of NRW for gridlength $l = 5000m$.



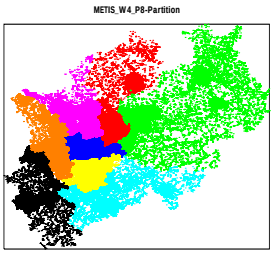
(c) 8-way METIS partition of NRW with weight function W1.



(d) 8-way METIS partition of NRW with weight function W2.



(e) 8-way METIS partition of NRW with weight function W3.



(f) 8-way METIS partition of NRW with weight function W4.

Figure A.1 Different partitions of NRW.

Bibliography

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. Network Flows. In *Handbooks in Operations Research and Management Science, Vol.1: Optimization*, pages 211–369, Amsterdam, 1989. North-Holland.

[2] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. Some Recent Advances in Network Flows. *SIAM Review* **33**:175–219, 1991.

[3] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows*. Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[4] R. K. Ahuja, K. Melhorn, J. Orlin, and R. E. Tarjan. Faster Algorithms for the Shortest Path Problem. Technical Report CS-TR-154-88, Departement of Computer Science, Princeton University, Princeton (USA), 1988.

[5] J. A. Azevedo, M. S. Costa, J. S. Madeire, and E. Martins. An Algorithm for the Ranking of Shortest Paths. *Eur. J. Operational Research* **69**:97–106, 1993.

[6] R. E. Bellman. On a Routing Problem. *Quart. Appl. Math.* **16**:87–90, 1958.

[7] H. Booth and J. Westbrook. A Linear Algorithm for Analysis of Minimum Spanning and Shortest-Path Trees of Planar Graphs. *Algorithmica* **11**:341–352, 1994.

[8] Bundesverkehrsministerium, editor. *Verkehr in Zahlen*. Deutsches Institut für Wirtschaftsforschung, 1999.

[9] Y. L. Chen. Finding the k Quickest Simple Paths in a Network. *Information Processing Letters* **50**:89–92, 1994.

[10] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. SPLIB. Available athttp://www.intertrust.com/star-lab.com/goldberg/soft.html

[11] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming* **73**(2):129–174, 1996.

[12] B. V. Cherkassky, A. V. Goldberg, and C. Silverstein. HQ. Available at http://www.intertrust.com/star-lab.com/goldberg/soft.html

[13] B. V. Cherkassky, A. V. Goldberg, and C. Silverstein. Buckets, Heaps, Lists, and Monotone Priority Queues. *SIAM Journal on Computing* **28**:1326–1346, 1999.

[14] S. Clarke, A. Krikorian, and J. Rausen. Computing the N Best Loopless Paths in a Network. *J. Soc. Indust. Appl. Mathematics* **11**:1096–1102, 1963.

[15] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, 1990.

[16] G. B. Dantzig. On the Shortest Route Through a Network. *Management Science* **6**:187–190, 1960.

[17] E. V. Denardo and B. L. Fox. Shortest-Route Methods: 1. Reaching, Pruning and Buckets. *Oper. Res.* **27**:161–186, 1979.

[18] N. Deo and C. Pang. Shortest Path Algorithms: Taxonomy and Annotation. *Networks* **14**:275–323, 1984.

[19] R. B. Dial. Algorithm 360: Shortest Path Forest with Topological Ordering. *Comm. ACM* **12**:632–633, 1969.

[20] E. W. Dijkstra. A Note on Two Problems in Connection with Graphs. *Numer. Math.* **1**:269–271, 1959.

[21] J. J. Divoky and M. S. Hung. Performance of Shortest Path Algorithms in Network Flow Problems. *Management Science* **36**:661–673, 1990.

[22] S. E. Dreyfus. An appraisal of some shortest path algorithms. *Operations Research* **17**:395–412, 1969.

[23] D. Eppstein. Bibliography on algorithms for k shortest paths. Available at http://liinwww.ira.uka.de/bibliography/Theory/k-path.html.

[24] D. Eppstein. Finding the k Shortest Paths. *SIAM on Journal on Computing* , to appear.

[25] European Telecommunications Standards Institute (ETSI). Digital Cellular Telecommunications System (Phase2+); General Packet Radio Service (GPRS); Service description, Stage 1; European Standard (Telecommunications seriers) GSM 02.60 version 6.1.0, Release 1997, 1997.

[26] C. Farhat. A Simple and Efficient Automatic FEM domain decomposer. *Computers and Structures* **28**(5):579–602, 1988.

[27] C. Farhat, S. Lanteri, and H. D. Simon. TOP/DOMDEC - a Software Tool for Mesh Paritioning and Parallel Processing. *J. of Computing Systems in Engineering* **6**(1):13–26, 1995.

[28] C. M. Fiduccia and R. M. Mattheyses. A Linear-time Heuristic for Improving Network Partitions. In *Proceedings of the 19th IEEE Design Automation conference*, pages 175–181. IEEE, 1982.

[29] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton Univ. Press, Princeton, New Jersey, 1962.

[30] B. L. Fox. k-th Shortest Paths and Applications to the Probablistic Networks. *ORSA/TIMS Joint National Mtg.* **23**:B263, 1975.

[31] M. L. Fredman and R. E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *J. Assoc. Comput. Mach.* **34**:596–615, 1987.

[32] T. Gal and H. Greenberg, editors. *Advances in Sensitivity Analysis and Parametric Programming*. Kluwer, Dordrecht, Netherlands, 1997.

[33] G. Gallo and S. Pallotini. Shortest Paths Algorithms. *Annals of Oper. Res.* **13**:3–79, 1988.

[34] M. R. Garey and D. S. Johnson. Some Simplified NP-Complete Graph Problems. *Theoretical Computer Science* **1**:237–267, 1976.

[35] M. R. Garey and D. S. Johnson. *Computers and Intractability*. Freeman, New York, 1979.

[36] C. Gawron. *Simulation-Based Traffic Assignment*. PhD thesis, University of Cologne, 1998.

[37] W. Gillmann. Mit elektronischen Piloten auf Kurs. *Handelsblatt* **21**:18, 01.02.1999. In German.

[38] F. Glover, R. Glover, and D. Klingman. A Computational Study of an Improved Shortest Path Algorithm. *Networks* **14**:25–37, 1984.

[39] F. Glover, D. Klingman, and N. Phillips. A New Polynomially Bounded Shortest Paths Algorithm. *Oper. Res.* **33**:65–73, 1985.

[40] T. Goehring and Y. Saad. Heuristic Algorithms for Automatic Graph Partitioning. Technical Report UMSI-94-29, Departement of Computer Science, University of Minnesota, Minneapolis (USA), 1994.

[41] A. V. Goldberg and T. Radzik. A Heuristic Improvement of the Bellman-Ford Algorithm. *Applied Math. Let.* **6**:3–6, 1993.

[42] A. V. Goldberg and C. Silverstein. Computational Evaluation of Hot Queues. Technical Report 97-104, NEC Research Institute, Princeton, New Jersey, 1997.

[43] A. V. Goldberg and C. Silverstein. Implementations of Dijkstra's Algorithm Based on Multi-Level Buckets. In *Network Optimization, Springer Lecture Notes in Economics and Mathematical Systems 450*, pages 292–327. Springer, 1997.

[44] H. Greenberg. An Annotated Bibliography for Post-solution Analysis in Mixed Integer Programming and Conbinatorial Optimization. Available at http://www.cudenver.edu/~hgreenbe/aboutme/papers/survey/mip.bib

[45] D. Gusfield. A Note on Arc Tolerances in Sparse Shortest-Path and Network Flow Problems. *Networks* **13**:191–196, 1983.

[46] P. Hart, N. Nilsson, and B. Raphel. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Trans. on System Science and Cybernectics* **4**(2):100–107, 1968.

[47] B. Hendrickson and R. Leland. Multidimensional Spectral Load Balancing. Technical Report 93-0074, Sandia National Laboratories, 1993.

[48] B. Hendrickson and R. Leland. A Multilevel Algorithm for Paritioning Graphs. Technical Report 93-1301, Sandia National Laboratories, 1993.

[49] B. Hendrickson and R. Leland. *The Chaco User's Guide Version 2.0*. Albuquerque (USA), 1995.

[50] B. Hendrickson and R. Leland. An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computing. *SIAM Journal on Scientific and Statistical Computing* **16**(2):452–469, 1995.

[51] R. Hoffman and R. R. Pavley. A Method for the Solution of the n th best path problem. *Journal of the ACM* **6**:506–514, 1959.

[52] J. Hoschek. *Mathematische Grundlagen der Kartographie*. B.I.-Wissenschaftsverlag, 1969. In German.

[53] S. Hübner. Elektronische Lotsen weisen Autofahrern den Weg. *Handelsblatt* **213**:31, 04.11.1998. In German.

[54] M. S. Hung and J. J. Divoky. A Computational Study of Efficient Shortest Path Algorithms. *Computers and Operations Research* **15**:567–576, 1988.

[55] R. Jacob. personal communication.

[56] R. Jacob, M. Marathe, and K. Nagel. A Computational Study of Routing Algorithms for Realistic Transportation Networks. Unclassified Report LA-UR 98-2949, LANL, 1998. Submitted to ISCOPE 98. Available at http://www-transims.tsasa.lanl.gov/documents-research98.html

[57] R. Karp. Reducibility among combinatorial problems. In R. Millera and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–104. Plenum Press, New York, 1972.

[58] R. Karp and J. Orlin. Parametric Shortest Path Algorithms with an Application to Cyclic Staffing. *Discrete Applied Mathematics* **3**:37–45, 1981.

[59] G. Karypis and V. Kumar. Analysis of Multilevel Graph Partitioning. Technical Report No. 95-037, Departement of Computer Science, University of Minnesota, Minneapolis (USA), 1995.

[60] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. Technical Report No. 95-035, Departement of Computer Science, University of Minnesota, Minneapolis (USA), 1995.

[61] G. Karypis and V. Kumar. Multilevel k-way Partitioning Scheme for Irregular Graphs. Technical Report No. 95-064, Departement of Computer Science, University of Minnesota, Minneapolis (USA), 1995.

[62] G. Karypis and V. Kumar. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices, Version 4.0*. Minneapolis (USA), 1998.

[63] N. Katoh, T. Ibaraki, and H. Mine. An Efficient Algorithm for K Shortest Simple Paths. *Networks* **12**:411–427, 1982.

[64] B. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell Systems Technical Journal* **29**(2):291–307, 1970.

[65] S. Krauß. *Microscopic Modeling of Traffic Flow: Investigation of Collision Free Vehicle Dynamics*. PhD thesis, University of Cologne, 1998.

[66] S. Krauß, P. Wagner, and C. Gawron. Continuos Limit of the Nagel-Schreckenberg model. *Phys. Rev. E* **54**:3707, 1996.

[67] S. Krauß, P. Wagner, and C. Gawron. Metastable States in a Microscopic Model of Traffic Flow. *Phys. Rev. E* **5597**:3707, 1996.

[68] J. B. Kruskal. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proceedings of the AMS* **7**:48–50, 1956.

[69] E. L. Lawler. Comment on Computing the k Shortest Paths in a Graph. *Communications of the ACM* **20**:603–604, 1977.

[70] B. J. Levit and B. N. Livshits. Neleneinye Setevye Transportnye Zadachi. *Transport* , 1972. In Russian.

[71] M. Luby and P. Ragde. A Bidirectional Shortest-Path Algorithm with Godd Average-Case Behavior. *Algorithmica* **4**:551–567, 1989.

[72] E. Q. V. Martins. An Algorithm for Ranking Paths that may contain Cycles. *Eur. J. Operational Research* **18**(1):123–130, 1984.

[73] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[74] K. Mehlhorn, S. Näher, M. Seel, and C. Uhrig. *The LEDA User Manual, Version 3.7*. http://www.mpi-sb.mpg.de/LEDA.

[75] E. Minieka. On Computing Sets of Shortest Paths in a Graph. *Communications of the ACM* **17**:351–353, 1974.

[76] E. F. Moore. The Shortest Path Through a Maze. In *Proc. of the Int. Symp. on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

[77] K. Nagel. *High-speed microsimulations of traffic flow*. PhD thesis, University of Cologne, 1995.

[78] K. Nagel and M. Schreckenberg. A Cellular Automaton Model for Traffic Flow. *J. Physique I* **2**:2221, 1992.

[79] J. Opdenplatz. Hierarchische Kürzeste-Wege-Verfahren. Diploma thesis, University of Cologne, 1991. In German.

[80] J. P. Ciarlet and F. Lamour. On the Validity of a front-oriented Approach to Partitioning Large Sparse Graphs with a Connectivity Constraint. Technical Report No. 94-37, Computer Science Departement, University of California, Los Angeles (USA), 1994.

[81] J. P. Ciarlet and F. Lamour. Recursive Partitioning Methods and Greedy Partitioning Methods: A Comparison on Finite Element Graphs. Technical Report No. 94-9, Computer Science Departement, University of California, Los Angeles (USA), 1994.

[82] S. Pallotino. Shortest-Path Methods: Complexity, Interrelations and New Propositions. *Networks* **14**:257–267, 1984.

[83] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, Massacusetts, 1994.

[84] U. Pape. Implementation and Efficiency of Moore Algorithms for the Shortest Root Problem. *Math. Prog.* **7**:212–222, 1974.

[85] A. Perko. Implementation of Algorithms for K Shortest Loopless Paths. *Networks* **16**:149–160, 1986.

[86] R. Preis and B. Diekmann. *The PARTY Partitioning-Library, User Guide - Version 1.1*. Paderborn (BRD), 1996.

[87] B. Ran and D. E. Boyce. *Modeling Dynamic Transportation Networks*. Springer, 2nd edition, 1996.

[88] M. Rickert. *Traffic Simulation on Distributed Memory Computers*. PhD thesis, University of Cologne, 1998. Available at http://www.zpr.uni-koeln.de/~mr/dissertation.

[89] P. Röbke-Doerr. Nie mehr im Stau? Verkehrsleittechnik und inviduelle Navigation. *c't* **5**:140–142, 2000. In German.

[90] J. B. Rosen, S.-Z. Sun, and B.-L. Xue. Algorithms for the Quickest Path Problem and the Enumeration of Quickest Paths. *Computers and Operations Research* **18**:579–584, 1991.

[91] K. Scott, G. Pabon-Jimenez, and D. Bernstein. Finding Alternatives to the Best Path. *Transpn. Res. Board* , 1997. Preprint 970682.

[92] R. Sedgewick and J. Vitter. Shortest Paths in Euclidean Graphs. *Algorithmica* **1**(1):31–48, 1986.

[93] J. Shapiro, J. Waxman, and D. Nir. Level Graphs and Approximate Shortest Path Algorithms. *Networks* **22**:619–717, 1992.

[94] D. Shier and C. Witzgall. Arc Tolerances in Shortest-Path and Network Flow Problems. *Networks* **10**:277–291, 1980.

[95] V. A. Stausberg. Ein Dekompositionsverfahren zur Berechnung kürzester Wege auf fast-planaren Graphen. Diploma thesis, University of Cologne, 1995. In German.

[96] R. Tarjan. Sensitivity Analysis of Minimum Spanning Trees and Shortest-Path Trees. *Bell Systems Technical Journal* **14**:30–33, 1982.

[97] M. Thorup. Undirected Single Source Shortest Paths in Linear Time. Presented at the FOCS'1997, 1997. Available at http://www.diku.dk/~mthorup.

[98] D. Wendeln. Navigatoren weisen den Weg. *Handelsblatt* **164**:B10, 27.8.1997. In German.

[99] P. D. Whiting and J. A. Hillier. A Method for Finding the Shortest Route through a Road Network. *Operations Research Quarterly* **11**:37–40, 1960.

[100] R. Wimmershoff. *Untersuchungen zur parallelen, objektorientierten Verkehrssimulation*. PhD thesis, University of Cologne, to be published.

[101] J. Yen. Finding the k shortest loopless paths in a network. *Management Science* **17**:712–716, 1971.

[102] F. B. Zhan and C. Noon. Shortest Path Algorithms: An Evaluation using Real Road Networks. *Transportation Science* **32**:65–73, 1998.

Deutsche Zusammenfassung

Das Problem, einen kürzesten Weg zwischen zwei Knoten in einem gewichteten Graphen zu finden, ist eines der klassischen Probleme in der Netzwerkoptimierung, das seit über vierzig Jahren Gegenstand ausgiebiger Forschungstätigkeit ist. In vielen praktischen Anwendungen taucht die Bestimmung von kürzesten Wegen entweder als eigenständige Fragestellung (z.B. bei Transportproblemen, Projektmanagement und DNA-Sequenzierung) oder als Teilproblem in einem komplexeren Problemzusammenhang auf (z.B. bei der Approximation von Funktionen und dem Knapsack-Problem).

Das Ziel der vorliegenden Arbeit ist die Untersuchung verschiedener Aspekte von algorithmischen Verfahren zur kürzesten-Wege-Bestimmung in sehr großen Straßengraphen. Motiviert wurde der Untersuchungsgegenstand durch die zunehmende praktische Bedeutung der Generierung von geeigneten Routenempfehlungen für eine Vielzahl von Fahrern in Straßennetzen.

Ausgelöst wurde diese Entwicklung durch die Notwendigkeit einer effektiveren Verkehrslenkung aufgrund des immer weiter steigenden Verkehrsaufkommens und den Einsatz neuer Technologien im Bereich der Fahrzeugelektronik in den letzten Jahren (siehe [89] für eine aktuelle Übersicht). Hier ist insbesondere die zunehmende Verbreitung von individuellen Navigationssystemen und das damit verbundene Gebiet der Telematik zu nennen. Seit der Markteinführung solcher Navigationssysteme vor einigen Jahren verzeichnen die Hersteller jährlich sich verdoppelnde Umsatzzahlen mit einem erwarteten Absatz von über 600000 Geräten in Deutschland für das Jahr 2000 [37]. Dabei geht die Entwicklung langsam aber stetig hin zu einer Routenführung, bei der immer mehr dynamische Verkehrsdaten berücksichtigt werden.

Ein weiteres Instrument zur effizienteren Planung und Steuerung von Verkehr ist die einzelfahrzeugbasierte Mikrosimulation wie z.B. in [36, 88]. Bei der iterativen Bestimmung des dabei angestrebten Verkehrsgleichgewichts müssen idealerweise in jedem Iterationsschritt die Routen aller Fahrer neu berechnet werden. Bereits für ein relativ kleines Untersuchungsgebiet wie die Stadt Wuppertal mit ungefähr 17000 Kanten im Graphen bedeutet dies die Berechnung von 500000 Rou-

ten pro Iterationsschritt [36], was selbst auf Computerworkstations der aktuellen Technik mehrere Stunden in Anspruch nimmt.

Beiden beschriebenen Anwendungen gemeinsam ist, daß eine benötigte Route nicht unbedingt der tatsächlich kürzeste Weg zwischen den beiden Endpunkten sein muß. Es reicht, wenn die Fahrtzeit einer vorgeschlagenen Routenempfehlung hinreichend nahe an derjenigen des optimalen Weges ist. Für die Zufriedenheit des Benutzers eines individuellen Navigationssystems ist es dabei wichtig, daß seine persönlichen Erwartungen erfüllt werden. Dagegen steht im Rahmen einer Verkehrssimulation eine ausreichende Beschreibung des realen Verkehrszustandes im Vordergrund, der von vereinzelten nicht zu großen Abweichungen von optimalen Wegen nicht beeinträchtigt wird.

In der Sprache der Graphentheorie läßt sich das kürzeste-Wege-Problem wie folgt darstellen: Finde für einen Startknoten s und einen Endknoten t eines gewichteten Graphen $G = (V, E, c)$ mit Knotenmenge V, Kantenmenge E und Kostenfunktion c, definiert auf den Kanten, einen Pfad zwischen s und t mit minimalen Kosten. Dabei ergeben sich die Kosten eines Pfades als Summe der Gewichte der Pfadkanten.

In Graphen mit nicht-negativer Kostenfunktion wie z.B. Straßengraphen ist der klassische Algorithmus zur Bestimmung von kürzesten Wegen das bereits 1959 von Dijkstra [20] vorgeschlagene Verfahren. Dabei wird in jedem Schritt der Knoten v mit minimaler temporärer Markierung gewählt und permanent markiert. Für alle Nachbarn w von v wird geprüft, ob die temporäre Markierung größer ist als der Weg von s über v nach w. Wenn ja, wird die temporäre Markierung von w aktualisiert. Bei Verwendung einer priority queue als Datenstruktur für die Verwaltung der temporär markierten Knoten führt der Aufwand der Knotenauswahl und der Distanzaktualisierung zu einer worst-case-Laufzeit von $\mathcal{O}(|E| + |V| \log |V|)$.

Im Laufe der Jahre sind eine Vielzahl von Implementierungen von Dijkstra's Algorithmus vorgeschlagen worden, die entweder zu praktischen Laufzeitverbesserungen oder zu einer verbesserten worst-case-Komplexität führen (eine Übersicht gibt z.B. [3]). Kürzlich wurde von Thorup [97] ein linearer Algorithmus für ungerichtete Graphen vorgeschlagen, der die inhärente Sortierung der Knoten nach Distanzen in Dijkstra's Algorithmus durch geeignete Identifizierung von Knoten, die in beliebiger Reihenfolge bearbeitet werden können, umgeht.

Die schnellsten Implementierungen von Dijkstra's Algorithmus benötigen für die Bestimmung eines kürzesten Weges im Straßengraphen von Nordrhein-Westfalen mit etwas über einer Million Kanten weniger als eine Sekunde auf einer Sun Enterprise E450 mit vier mit 400 MHz getakteten UltraSPARC-II Prozessoren (s. Kapitel 2).

In praktischen Anwendungen kommen häufig heuristische Verfahren zur Routengenerierung in Straßengraphen zum Einsatz. Solche Methoden nutzen zumeist die spezielle Struktur von Straßengraphen mit gerichteten Kanten, Kantenlängen

nahe der euklidischen Distanz der beiden Endknoten, Fastplanarität und einer hierarchischen Struktur aufgrund einer Typisierung der Kanten nach Wichtigkeit.

In Kapitel 3 wurde eine vom Autor entwickelte Heuristik vorgestellt, die die Ähnlichkeit von kürzesten-Wege-Bäumen in Straßengraphen für nahe beeinander liegende Startknoten ausnutzt. Diese so genannte Baumheuristik ist in Bezug auf die Rechenzeit auf den größten betrachteten Netzen etwa um einen Faktor acht schneller als Dijkstra's Algorithmus. In Bezug auf die Anzahl permanent markierter Knoten erzielt die Heuristik einen Gewinn um etwa den Faktor 20. Dabei werden Pfade gefunden, die im Mittel um weniger als 1% vom optimalen Weg abweichen, sofern die beiden Endknoten nicht zu nahe beeinander liegen.

Die Baumheuristik gliedert sich in mehrere Phasen. Zuerst wird der Graph in k Klassen möglichst gleicher Größe partitioniert und dann ein Suchgraph für jede Partitionsklasse erzeugt. Die Suchgraphen haben eine baumähnliche Struktur und enthalten wesentlich weniger Kanten als der Gesamtgraph, aber alle Knoten des Graphen. Sie werden erzeugt, indem für jede Partitionsklasse eine Reihe von Basisknoten bestimmt wird. Die Kantenmenge eines Suchgraphen besteht dann aus allen Kanten der Klasse und der Vereinigung aller Kanten von kürzesten-Wege-Bäumen der Basisknoten. Die Suchgraphen sind somit lokal dicht, aber global dünn besetzt. Die Anwendung eines Rückwärtsdijkstra bei der eigentlichen Routensuche führt dann zu einem sehr schnellen Algorithmus, der darüber hinaus wenig von der Länge der Wege beeinflußt wird. Durch die Zerlegung des ursprünglichen Graphen eignet sich die Baumheuristik auch für eine Verkehrssimulation auf Parallelrechnern.

Bei der Zerlegung des Graphen wurden zwei neue Methoden, die Charakteristiken von kürzesten-Wege-Bäumen berücksichtigen mit einem k-way partitioning Algorithmus von Karypis und Kumar [62] verglichen. Dabei zeigte sich, daß der Zusammenhang der Partitionsklassen einen großen Einfluß auf die Güte der gefundenen Wege hat, wobei die in der Arbeit entwickelten Zerlegungsmethoden fast immer zusammenhängende Partitionsklassen lieferten.

Andere Verfahren zur Routengenerierung in Straßengraphen sind der A^*-Algorithmus und die HISPA Heuristik. Letzere sucht kürzeste Wege von den beiden Endknoten zu Knoten der obersten Hierarchieebene in einem Kreis mit einem gegebenen Radius r. Diese Pfade werden als Kanten entsprechender Länge zur obersten Hierarchieebene hinzugefügt und ein kürzester Weg auf der so modifizierten Ebene zwischen s und t bestimmt. Dabei ist die Anzahl der Kanten der obersten Hierarchieebene sehr klein im Vergleich zur Gesamtzahl der Kanten, jedoch ist eine optimale Bestimmung des Radius r schwierig.

Für euklidische Netzwerke haben Sedgewick et al. [92] den A^*-Algorithmus vorgeschlagen, der sich auch für Straßengraphen verwenden läßt. Dieses exakte Verfahren steuert die Suche bei Dijkstra's Algorithmus in Richtung des Zielknotens durch Verwendung von so genannten 'future costs', die aus geometrischen In-

formationen gewonnen werden. Dadurch wird der Suchbereich schmäler, auf der anderen Seite erzeugt die Berechnung der future costs zusätzlichen rechnerischen Aufwand. Im Vergleich zur Baumheuristik schneiden beide Verfahren in Bezug auf die Rechenzeit schlechter ab, und in fast allen Fällen war die Qualität der gefundenen Routen bei der Baumheuristik wesentlich besser als bei der HISPA Heuristik.

Kürzlich wurde von Jacob et al. [56] eine Modifizierung des A^*-Algorithmus auf Straßengraphen untersucht, bei der die future costs mit einem overdo-Faktor multipliziert werden. Dies beschleunigt die Rechenzeit des Algorithmus, führt jedoch zu einem heuristischen Verfahren. In Kapitel 4 wurde dieser modifizierte Algorithmus auf vier verschiedenen Straßennetzen ausgiebig untersucht. Für kleine overdo-Faktoren zwischen eins und 1.5 findet der Algorithmus den optimalen Pfad oder Routen mit nur geringen Abweichungen. Während der exakte A^*-Algorithmus bei sehr langen Pfaden wegen der zusätzlichen Berechnung der future costs zum Teil längere Laufzeiten als Dijkstra's Algorithmus aufweist, wird für einen Faktor von 1.5 eine deutliche Laufzeitverbesserung erzielt.

Diese steigert sich mit größeren overdo-Faktoren, jedoch führen solche Faktoren in der Regel zu Pfaden von sehr schlechter Güte. Der Grund dafür ist, daß der Algorithmus für große Faktoren die geometrische Entfernung der Nachbarknoten des zuletzt permanent markierten Knotens zum Zielknoten als einziges Auswahlkriterium für den nächsten Knoten auf dem Pfad aufweist. Dadurch ist der Pfad die Folge von 'nächsten' Nachbarn, was dazu führt, daß eine falsche Abzweigung im späteren Verlauf des Algorithmus nicht mehr korrigiert werden kann. Unter bestimmten Voraussetzungen an den Graphen, die bei Straßennetzen in der Regel erfüllt sind, wurde eine theoretische Schranke für den overdo-Faktor hergeleitet, so daß der Algorithmus für größere Werte das beschriebene Verhalten zeigt. Für eine praktische Anwendung wurde daraus ein zu erwartender Wert von höchstens fünf für diese Schranke bestimmt.

Informationen über die Robustheit gegebener Routenempfehlungen bei sich ändernden Kantengewichten können in praktischen Anwendungen ein hilfreicher Entscheidungsparameter für eine eventuelle Neuberechnung einer Route sein. Unter der Voraussetzung, daß sich nur ein Kantengewicht ändert, wurde in Kapitel 5 ein Algorithmus vorgestellt, der Toleranzen für Kanten in Bezug auf einen kürzesten Weg bestimmt. Dabei zeigte sich für den Straßengraphen von Nordrhein--Westfalen, daß der kürzeste Weg nur von Änderungen eines sehr geringen Anteils der Kanten beeinflußt wird. Die mit Hilfe der Kantentoleranzen gewonnenen Informationen können darüber hinaus benutzt werden, um Alternativrouten zu ermitteln. Dabei können die Alternativen gezielter an spezielle Anforderungen an solche Wege in der praktischen Anwendung angepaßt werden als bei der Verwendung einschlägiger k-kürzester-Wege Algorithmen wie z.B. [101].

Deutsche Kurzzusammenfassung

In dieser Arbeit wird das Problem, einen kürzesten Weg in einem großen Straßengraphen zu finden, untersucht. Der klassische Lösungsalgorithmus für Graphen mit nicht-negativer Kostenfunktion ist der Algorithmus von Dijkstra mit einem Laufzeitverhalten von $\mathcal{O}(|E| + |V| \log |V|)$ unter Benutzung einer einfachen Priority Queue als Datenstruktur für temporär markierte Knoten. Wir stellen eine neue, sogenannte Baumheuristik vor, die auf der Ähnlichkeit von kürzesten-Wege-Bäumen basiert und besonders in praktischen Anwendungen wie beispielsweise in der mikroskopischen Verkehrssimulation oder bei Online-Routingsystemen eingesetzt werden kann. Anstatt einen Weg im Gesamtgraphen zu suchen, partitioniert die Baumheuristik den Graphen in Klassen von in etwa gleicher Größe und konstruiert einen speziellen Suchgraphen für jede Klasse. Auf einem Testgraphen mit ca. einer Million Knoten ist die Baumheurisitk um einen Faktor größer drei schneller als Dijkstra's Algorithmus bzgl. der Laufzeit. Die von der Baumheuristik gefundenen Wege haben dabei einen erwarteten Fehler von unter 1%, sofern Start- und Endknoten der Suche nicht zu nahe beieinander liegen. Ferner analysieren wir den A^*-Algorithmus mit Overdo-Faktor, der ursprünglich für Euklidische Netzwerke entworfen wurde, und leiten ein Intervall $[1.27, \ldots, 5]$ her, aus dem ein optimaler Overdo-Faktor in praktischen Anwendungen gewählt werden sollte. Abschließend stellen wir einen Algorithmus zur Bestimmung von Toleranzen für die Kantengewichte eines kürzesten Weges vor, der verwendet werden kann, um sinnvolle Alternativrouten zum kürzesten Weg zu finden.

Danksagung

Bei der Erstellung dieser Arbeit habe ich von verschiedenen Seiten Unterstützung erhalten. Mein besonderer Dank gilt

Prof. Dr. Rainer Schrader für die Betreuung dieser Arbeit und die Möglichkeit, mich mit einem interessanten und aktuellen Thema zu beschäftigen,

Rolf Böning für die vielen Impulse und die unvergleichliche Unterstützung bei der Planung und Durchführung dieser Arbeit,

Christian Gawron für die vielen Antworten auf Fragen zur computertechnischen Problembewältigung, zum Teil bevor ich die Fragen stellen konnte,

Marcus Metzler für die Bereitschaft, sich mit meinen Fragen zu kürzesten Wegen auseinanderzusetzen und die Schaffung eines mobilen Arbeitsplatzes,

Nils Eissfeldt, Peter Wagner, Georg Hertkorn, Stephan Rosswog und allen anderen jetzigen und ehemaligen Kollegen der „Verkehrsgruppen“ des ZPR/ZAIK und des DLR für die gute Zusammenarbeit,

Dagmar Groth und Olaf Wendisch für ihren Beitrag zu kürzesten Wegen auf Straßengraphen,

Karin Weinbrecht und allen Anderen für das aufmerksame Korrekturlesen,

allen Mitarbeitern des ZPR/ZAIK für die tolle Arbeitsatmosphäre, die ich sehr genossen habe,

meinen Eltern, die mich immer (nicht nur finanziell) unterstützt und dadurch diese Arbeit maßgeblich mit ermöglicht haben,

Frank Lathe für die großartige Gastfreundschaft und vorzügliche nahrungstechnische Versorgung,

Barthel Steckemetz für die Jobvermittlung,

Romane und Piet Schiffeler für die moralische Unterstützung,

Nicole Schiffeler für viel Verständnis, Glaube, Liebe, Hoffnung und vor allem für Lioba,

Lioba für schlafreiche Nächte und das Glück, nach einer (arbeitsbedingt) kurzen Nacht morgens als erstes ein strahlendes Babylächeln zu erblicken.

Further I wish to thank all authors of free software which I used while working on this thesis, among them Colin Kelley, Donald E. Knuth, Leslie Lamport, Brian V. Smith, Richard Stallman, Supoj Sutanthavibul, Larry Wall, and Thomas Williams.

Erklärung

Ich versichere, daß ich die von mir vorgelegte Dissertation selbständig angefertigt, die benutzten Quellen und Hilfsmittel vollständig angegeben und die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken im Wortlaut oder dem Sinn nach entnommen sind, in jedem Einzelfall als Entlehnung kenntlich gemacht habe; daß diese Dissertation noch keiner anderen Fakultät oder Universität zur Prüfung vorgelegen hat; daß sie – abgesehen von unten angegebenen Teilpublikationen – noch nicht veröffentlicht worden ist sowie, daß ich eine solche Veröffentlichung vor Abschluß des Promotionsverfahrens nicht vornehmen werde. Die Bestimmungen dieser Promotionsordnung sind mir bekannt. Die von mir vorgelegte Dissertation ist von Prof. Dr. R. Schrader betreut worden.

Teilpublikationen

S. Rosswog, C. Gawron, S. Hasselberg, R. Böning, P. Wagner. Computational Aspects in Traffic Simulation Problems. *To appear in Future Generation Computer Systems*, 2000.

Lebenslauf

Persönliche Daten

Name	Stephan Hasselberg
Adresse	Flandrische Straße 27, 52076 Aachen
geboren am	24.04.1968 in Jülich
Eltern	Christa Hasselberg, geb. Strebe-Gustmann, Günter Hasselberg
Staatsangehörigkeit	deutsch
Familienstand	verheiratet, ein Kind

Schulbildung

1974 – 1978	Grundschule in Jülich
1978 – 1987	Gymnasium in Jülich
1984 – 1985	Austauschschüler in Hutchinson, Kansas/USA

Zivildienst

1987 – 1989	im St. Hildegardis-Altenheim in Jülich

Studium

1989 – 1991	Grundstudium der Mathematik an der Rheinischen Friedrich–Wilhelms–Universität Bonn
1991	Vordiplom
1991 – 1995	Hauptstudium an der Rheinischen Friedrich–Wilhelms–Universität Bonn
1993 – 1994	Studentische Hilfskraft am Institut für Diskrete Mathematik der Rheinischen Friedrich–Wilhelms–Universität Bonn
1995	Diplom in Mathematik
1995 – 1996	Promotionsstudium der Informatik an der Gerhard Mercator Universität Duisburg Gesamthochschule
1996 – 2000	Promotionsstudium der Informatik an der Universität zu Köln
seit 1994	Wirschaftswissenschaftliches Zusatzstudium für Naturwissenschaftler an der Fernuniversität Hagen
1996	Vordiplom im wirtschaftswisschenschaftlichen Zusatzstudium

Berufstätigkeit

1995 – 1996	Wissenschaftliche Hilfskraft an der Gerhard Mercator Universität Duisburg Gesamthochschule
1996 – 2000	Wissenschaftlicher Mitarbeiter am Zentrum für Angewandte Informatik (ZAIK) der Universität zu Köln