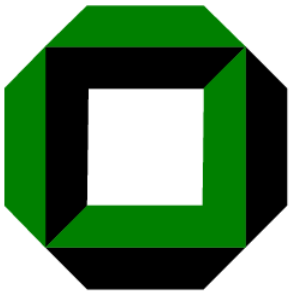
Efficient Computation of Many-to-Many Shortest Paths

Sebastian Knopp

Diplomarbeit



Institut für Theoretische Informatik
Algorithmik I, Lehrstuhl Prof Dr. Dorothea Wagner
Fakultät für Informatik
Universität Karlsruhe (TH)

Danksagung

An erster Stelle möchte ich meinen Eltern danken. Ich hatte während meines gesamten Studiums jederzeit ihre volle Unterstützung. Für die *großartige* Betreuung meiner Diplomarbeit danke ich Dr. Frank Schulz. Vielen Dank auch an Prof. Dr. Peter Sanders und Dominik Schultes, die mir beim Thema Highway Hierarchies mit vielen Ideen und Diskussionen weiter geholfen und mir die Vorberechnung für diese Beschleunigungstechnik zur Verfügung gestellt haben. Ich danke Stefan Hug und den Kollegen bei der PTV AG für die Unterstützung und die Möglichkeit dieses interessante Thema in einer immer sehr angenehmen Arbeitsatmosphäre bearbeiten zu können. Ich danke Prof. Dr. Dorothea Wagner für die Bereitschaft diese Arbeit zu betreuen und die Unterstützung der Zusammenarbeit mit einem Unternehmen. Außerdem danke ich Daniel Delling, dem Betreuer dieser Arbeit am Institut für theoretische Informatik.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig angefertigt habe und nur die angegebenen Hilfsmittel und Quellen verwendet wurden.

Karlsruhe, im Oktober 2006

Abstract

Considering large road networks, we are interested in finding all quickest connections between given source and target locations. The computation of such a distance table is what we call the *many-to-many shortest path problem*. For several problems in the field of logistics this is an important subtask. For example it is required as an input for vehicle routing problems. Also, distance tables are used as an auxiliary tool for several tasks concerning exact and heuristic shortest path computations.

To solve the many-to-many shortest path problem one can simply run Dijkstra's shortest path algorithm for every source node. Considering huge graphs, this approach affords a lot of time: In the road network of Europe instances with 1 000 locations need about three hours, computing distance tables for 10 000 locations takes more than one day. Hence, there is considerable interest to speed up this task.

We consider two general situations in this thesis: The first one does not allow any preprocessing to get along with flexible edge weights. The second setting considers the graph to be static and allows a preprocessing step. We drove extensive experiments to measure the performance of our approaches.

Without preprocessing, two techniques can be used to accelerate point-to-point shortest path queries: goal-directed search and bidirectional search. We show that both approaches can be also used to compute many-to-many shortest paths efficiently. One of our goal-directed many-to-many techniques uses geographical information to search towards the targets. Depending on the input this method is two times faster than Dijkstra's Algorithm. An elaborated variant of goal-directed search uses implicit landmarks to improve the sense of goal direction. This is useful in particular for larger problem instances. Also, bidirectional search can be turned to a method that accelerates the computation of distance tables. For this we also can measure speedups of about two.

Highway hierarchies is a concept that explores the hierarchical structure of a road network in a preprocessing step and uses this information to accelerate all further queries. This efficient technique for point-to-point queries can be used to obtain a very efficient many-to-many algorithm. In the road graph of Europe we see speedups of more than 1 000. As an impressive example, notice that our highway hierarchy based many-to-many technique can solve the problem with 10 000 locations in less than one minute—Dijkstra's Algorithm would take far more than one day.

Contents

Chapter 1

Introduction

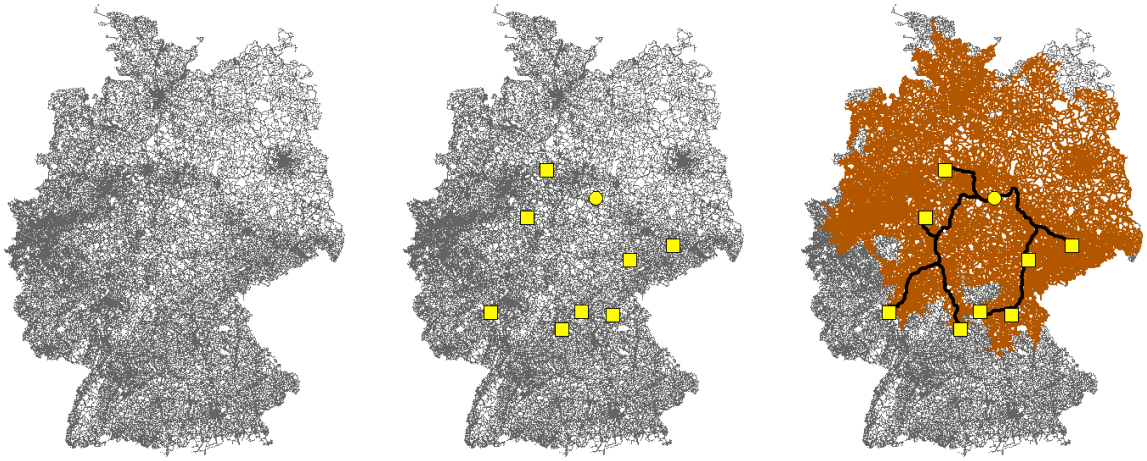
Considering large road networks we are interested in finding quickest connections with respect to costs assigned to each street segment. Costs can be chosen by an objective function we want the routes to be optimised for. In the case of route planning in road networks, this could be travel times for a specific vehicle type depending on the road category of an edge, travel distances or a combination of both. In the field of algorithmic graph theory this task is known as *shortest path problem*. Common variants are the *single source shortest path problem*, the *all pairs shortest path problem* or the *point-to-point shortest path problem*.

What we consider in this work is the *many-to-many shortest path problem*. Given a set of sources S and a set of targets T located somewhere on a road graph, we want to know the distances from all nodes $s \in S$ to all targets $t \in T$. The answer of such a query is a matrix with $|S| \cdot |T|$ entries. Every one of the $|S|$ rows of such a table represents distances from a certain source s, every column represents distances to a certain target t. In practice we deal most often with source and target sets with the property $S = T$, hence the matrices are quadratic.

To solve this, one can simply run Dijkstra's shortest path algorithm for every source node. But if we regard the computation of a large matrix of distances in a huge street graph, for example the detailed road network of Europe with more than 18 millions of nodes, this approach can be very slow. As an example, consider the computation of a matrix with 1 000 sources and 1 000 targets. A fast implementation of Dijkstra's Algorithm takes about 12 seconds to compute one single source query and has to be run for every one of the 1 000 source nodes. Hence, the computation of this single matrix lasts about three hours and 20 minutes. Computing an even bigger matrix would of course take even longer—far more than one day for a $10\,000 \times 10\,000$ matrix. The *goal of this work* is to speed up the computation of distance matrices in road networks.

We are going to consider two different settings. The *first* one does not allow any preprocessing. The costs for traversing road segments can be defined right before a query, satisfying custom requirements. This allows the usage of very flexible edge weights. For example a user can freely configure the velocity profile of his vehicle on different road types or up to date traffic information can be included.

In the *second* setting we assume that the graph can be considered as static to allow a preprocessing step that accelerates all further shortest path queries. In this work we



(a) German road network (b) Many-to-Many request (c) Dijkstra search space

Figure 1.1: *A many-to-many request consists of several source and target nodes located in a road network. The standard method to solve this, is to run Dijkstra's Algorithm for every source node. Dijkstra's Algorithm visits all nodes that are closer to the source node than the farthest target.*

are going to consider both approaches: Chapter 3 deals with the first setup, that forbids preprocessing and in Chapter 4, a preprocessing step is allowed. Chapter 5 shortly discusses some other approaches, among others we will consider how an heuristic algorithm, often used in commercial systems, can profit from the cognition of the former chapters.

1.1 Applications

For road networks two major fields require the computation of distance matrices. Section 1.1.1 explains with the vehicle routing problem a task occurring in the context of logistics, Section 1.1.2 explains the usefulness of large distance matrices as an auxiliary tool that has to be preprocessed and is used to accelerate several other tasks.

1.1.1 Vehicle Routing Problem

Important applications for the computation of many-to-many shortest paths are actual problems as they appear in the field of logistics. A lot of companies, acting in all sorts of branches of trades, have to plan daily tours for a large pool of motor cars, light commercial vehicles or trucks. Examples can be found in a huge number of domains. To name just some of them for example publishing, parcel services, sales representatives, field services or companies transporting many different kinds of goods, like foodstuff, furniture, construction material, etc., should be mentioned here. In literature problems of this kind can be found as *vehicle routing problem*.

For such scenarios the input for an instance of the vehicle routing problem consists of several locations, e.g. factories, shops, depots or customers and a number of domain specific constraints, such as the number and load capacity of vehicles, opening times of depots, timed customer requests and so on. From an algorithmic perspective there are two core problems: The *first step* is the computation of a matrix of distances between the given locations in a road graph. The *second step* is to solve one of the various variants of the vehicle routing problem. Although this are NP-hard problems, there are heuristic algorithms that work very well in practice and even for large problem instances they afford reasonable time in practice. In this work we consider the first problem of computing distance matrices in large road networks between given locations. The vehicle routing problem itself is not considered here, for an overview consider for example [5]. Note, that here the computation of a quadratic matrix with asymmetric[1] distances is required, which seems to be the most common application in road networks.

Regarding algorithm complexity, we have with the vehicle routing problem an NP-hard problem and with the distance matrix calculation one that can be solved by an algorithm with a polynomial running time[2] of $O(|S| \cdot m \cdot \log n)$. It seems to be a somewhat curious result from practice, that for common inputs using an even fast implementation of the plain version of Dijkstra's Algorithm to determine the distance matrix could last hours, while for the second task of vehicle routing, good heuristics can compute the vehicle tours in less than a minute. Hence, the environment of vehicle routing problems can heavily profit from speeding up the initial task of finding routes in road networks.

Considering the distribution of the locations for which we want the distance matrix to be determined, arbitrary constellations are possible. In some cases, e.g. for a local newspaper delivery, points can be concentrated around one city. In other cases the locations could be distributed in a large area, e.g. one country like Germany or even a whole continent like Europe. Also having a handful of clusters is a situation that occurs often in real world data. Practitioners report that common input sizes, requiring at least daily a complete recomputation, are distance matrices with about 200 up to 500 locations.

1.1.2 Large Auxiliary Distance Tables

Very large distance matrices are often used as an auxiliary tool in miscellaneous applications. For example consider a situation, where one is interested in a very fast approximation of distances, though such distances do not have to be exact to the last detail. Hence, a good estimation is sufficient here. For this task in road networks, an approximation using the flight distance is often too coarse, because of several geographical conditions such as rivers or mountains. A precomputed distance matrix, that stores distances between nodes positioned on a grid lying over the graph, can do better here. A reasonable estimation for

[1] In directed graphs in general $d(s,t) \neq d(t,s)$

[2] Standard implementation of Dijkstra's Algorithm using a binary heap. Better theoretical bounds are known.

distances is obtained by taking the sum of the flight distances from source and target to its respective nearest grid point plus the distance of those corresponding grid points, that can be easily looked up from the distance table. Practitioners report that relevant sizes of matrices for this application reach from about $8\,000 \times 8\,000$ up to $12\,000 \times 12\,000$.

There are also up to date results of shortest path research that use large distance tables to accelerate shortest path queries in huge graphs. The preprocessing of the extremely fast technique of transit node routing [1] profits from a fast method for the determination of distance matrices and actually already uses our algorithm presented in Chapter 4. The method of precomputed cluster distances [19] can also profit from this approach.

The preprocessing of this approach requires to compute cluster distances $d(V_i, V_j) := \min_{s \in V_i, t \in V_j} d(s,t)$ between clusters $V_1 \dot\cup \cdots \dot\cup V_k$. With the introduction of additional nodes s_i and t_i, $0 < i \leq k$, that are connected to the border nodes of a cluster by weight zero edges this can be done by solving the many-to-many problem for $S = \{s_1, \ldots, s_k\}$ and $T = \{t_1, \ldots, t_k\}$.

1.2 Related Work

There are results that accelerate many-to-many shortest paths for rather dense graphs with $m \gg n$, e.g., [33]. The only specific result that would be useful for road networks (or any other kind of sparse graphs) we are aware of is [29]. There, goal-directed search is turned to a many-to-many algorithm by combining geometric potential functions of all targets. A bidirectional approach is introduced in terms of goal-directed search, using a Voronoi partitioning of the graph with respect to the target nodes to determine a potential function.

A lot of recent work deals with the acceleration of point-to-point shortest path queries. Before we review several known speedup techniques for such single-pair shortest path queries we give a rough estimation of the performance of a naive adaption. The simplest way is to apply the speedup technique for every pair $(s,t) \in S \times T$. Considering large matrices, even for the fastest point-to-point queries this is outperformed by simply executing the plain version of Dijkstra's algorithm $\min\{|S|, |T|\}$ times[3]. We denote the average time for one Dijkstra run by t_{Dijkstra} and suppose a technique with an average point-to-point speedup of $s = \frac{t_{\text{Dijkstra}}}{t_{\text{Speedup}}}$. Then, with the naive adaption the computation of the matrix needs about $t_{\text{Speedup}} \cdot |S| \cdot |T|$ compared to $t_{\text{Dijkstra}} \cdot \min\{|S|, |T|\}$ using Dijkstra's Algorithm. Hence, comparing both rough estimations we conclude that an acceleration of the matrix computation is only possible for $s > \max\{|S|, |T|\}$.

In practice, it is very common to speed up the single-pair variant of Dijkstra's algorithm using *goal-directed search* [13] or *bidirectional search*. These techniques need no preprocessing and usually yield a speed-up factor of around 2. In Chapter 3 we present approaches to adapt these techniques to the many-to-many shortest path problem. Also,

[3] W.l.o.g. we suppose $|S| < |T|$, else the reverse graph is considered.

we describe an approach that turns the goal-directed search with preprocessed information (e.g., [10, 19]) into a technique without preprocessing by using some of the query nodes as implicit landmarks.

A lot of recent publications deal with the task of accelerating shortest path queries in a situation where the graph is considered to be static and a certain amount of preprocessing is affordable. This situation allows speed-up factors that are orders of magnitude larger.

One of the currently fastest speedup techniques for shortest path queries in road networks is the approach of *highway hierarchies* [25] that takes advantage of the hierarchical structure of the input. Outside some local areas around the source and the target node, only important edges have to be considered during the query. Speedups of up to 8 000 are possible for continental road networks with a preceding preprocessing step that takes only about 20 minutes. This approach is based only on hierarchical properties of the graph and does not use any goal-directing methods[4]. It turns out that this is very well suitable for many-to-many shortest paths. In Chapter 4 we present a very efficient algorithm based on the concept of highway hierarchies to compute many-to-many shortest paths.

Very recently, *transit node routing* [1] has accelerated point-to-point shortest path queries by another two orders of magnitude. However, transit node routing needs considerably more preprocessing time and space. Furthermore *its* preprocessing *uses* our highway hierarchy based algorithm to precompute a huge distance table. Even if the information for transit node routing is available, our algorithm remains up to one order of magnitude faster for large distance tables.

Reach based routing [12] prunes edges if they are sufficiently far away from both source *and target*. By pruning edges based on the supposedly closest target, reach based routing can search for several targets. However, this is a very conservative assumption and the involved bookkeeping is likely to be prohibitive unless $|T|$ is very small.

The bidirectional 'self bounded' variant of reach based routing [7] can perform independent forward and backward searches. We can use techniques described in Chapter 4 to use this for solving many-to-many problems quite efficiently. However, currently, reach based routing without augmentation by goal-directed search produces larger search spaces than highway hierarchies.

Also, the separator based *multi-level method* (e.g., [28, 3]) could be adapted to the many-to-many problem using methods analogous to ours. However, we do not pursue this since considerably more preprocessing time and space are required by this method.

Geometric containers [32] and *edge flags* [20, 16] prune the search at edges that do not lead to the target. Although this test could be generalized to test efficiently whether an edge leads to some target, this is not likely to be successful if the target nodes spread widely over the graph.

[4] A combination that adds goal-direction based on landmarks to highway hierarchies is presented in [4]

1.3 Overview

We present an overview of the structure of this work:

Chapter 2, Preliminaries In this chapter basic definitions of graph theory and graph algorithms are given to have a consistent notation throughout this work. A formal problem description is given and Dijkstra's Algorithm is introduced. Further we describe implementation details and the experimental setup in this chapter because references to experimental results are interspersed at various points in the later chapters.

Chapter 3, Without Preprocessing *Without preprocessing*, two techniques can be used to accelerate point-to-point shortest path queries: goal-directed search and bidirectional search. We show that both approaches can be also used to compute many-to-many shortest paths efficiently. One of our goal-directed many-to-many techniques uses geographical information to search towards the targets. Depending on the input this method is two times faster than Dijkstra's Algorithm. An elaborated variant of goal-directed search uses implicit landmarks to improve the sense of goal direction. This is useful in particular for larger problem instances. Also, bidirectional search can be turned to a method that accelerates the computation of distance tables. For this we also can measure speedups of about two.

Chapter 4, Highway Hierarchies *Highway hierarchies* is a concept that explores the hierarchical structure of a road network in a preprocessing step and uses this information to accelerate all further queries. This efficient technique for point-to-point queries can be used to obtain a very efficient many-to-many algorithm. In the road graph of Europe this technique yields speedups of more than 1 000. The problem with 10 000 locations can be solved in less than one minute—Dijkstra's Algorithm would take far more than one day.

Chapter 5, Conclusion We give a conclusion of this work and give an overview of the performance of our many-to-many algorithms concerning real world instances. An outlook is given that describes possible further developments and the application of our ideas for an heuristic approach that is often used in commercial systems.

Chapter 2

Preliminaries

In this chapter we start with basic definitions of graph theory and graph algorithms. Mostly well known facts which can be found in basic textbooks are presented here. We repeat them to have a consistent notation throughout this work. A formal problem description is given and Dijkstra's Algorithm is introduced.

Further we describe implementation details and the experimental setup already in this chapter, because references to experimental results are interspersed at various points in the later chapters.

2.1 Definitions

A directed graph $G = (V, E)$ is a pair of nodes V and edges $E \subset V \times V$. We will denote the number of nodes $|V|$ by n and the number of edges $|E|$ by m throughout this thesis. We call $\overline{G} = (V, \overline{E})$ with $\overline{E} = \{(v, u) : (u, v) \in E\}$ the *reverse graph* of G.

A path P in the graph G is a sequence of nodes $(v_0, v_1, \ldots, v_n)$ such that $(v_i, v_{i+1}) \in E$ for all $0 \leq i < n$. If $0 < k < l < n$ then $P|_{v_k \to v_l}$ denotes the subpath $(v_k, \ldots, v_l)$ of P. A graph is called *connected* if for every pair (u, v) of nodes there is a path from u to v. Edge weights are given by a function $l : E \to \mathbb{R}_{>0}$. The length $l(P)$ of a path is the sum of the weights of its edges $l(P) = \sum_{i=0}^{n} l(v_i, v_{i+1})$. We call P a *shortest path* from s to t, if there is no P' with $l(P') < l(P)$ and its length is denoted by $d(P)$. The distance $d(s, t)$ of two vertices is the length of a shortest path from s to t. A function $L : V \to \mathbb{R}^2$ is called layout.

2.1.1 Many-to-Many Shortest Path Problem

The task we address in this work is formally described in the following. Given a set of sources $S = \{s_i : 0 \leq i < |S|\} \subset V$ and a set of targets $T = \{t_i : 0 \leq j < |T|\} \subset V$ we want to compute a *distance matrix* $d_{i,j} = D \in \mathbb{R}^{|S| \times |T|}$ such that $d_{i,j} = d(s_i, t_j)$ is the length of a shortest path from s_i to t_i. We will refer to this as the *many-to-many shortest path problem*. If $|S| = |T|$ we call a distance matrix *quadratic*, otherwise *asymmetric*.

```
1   DIJKSTRA( s )
2       ∀ u ∈ V : dist(u) ← ∞
3       ∀ u ∈ V : state(u) ← unreached
4       dist(s) ← 0
5       pqueue.insert(s)
6       while( not pqueue.empty() )
7           u ← pqueue.extractMin()
8           state(u) ← settled
9           for all outgoing edges (u, v) of u
10              RELAX(u, v)
```

```
1   RELAX(u, v)
2       if (dist(u) + l(u, v) < dist(v))
3           dist(v) ← dist(u) + l(u, v)
4           if (state(v) == unreached)
5               pqueue.insert(v, dist(v))
6               state(v) ← reached
7           else
8               pqueue.decreaseKey(v, dist(v))
```

Figure 2.1: *Dijkstra's Algorithm in pseudo code representation.*

2.2 Dijkstra's Algorithm

The classic algorithm for the Single Source Shortest Path problem is the algorithm of Dijkstra (Figure 2.1), which finds shortest paths form a source s to all vertices in the graph. During the algorithm for every node a *tentative distance* is maintained and every node takes one of the three states `settled`, `reached` or `unreached`. Initially, all nodes are `unreached`. Nodes to those any path—not necessarily shortest—has been found are `reached`. A node v is `settled` if a shortest path from s to v has been found, and the distance is exact. We call a tentative distance from s to v *exact*, if it is equal to the length of a shortest path from s to v.

`Reached` nodes are managed in a priority queue, which supports the operations *insert*, *decreaseKey* and *extractMin*. We have a function RELAX that checks for an edge (u, v) if it can improve the path to v. The method *insert* adds an element to the queue. This happens when an edge to an `unreached` node is relaxed. Then, this node is inserted into the queue with its tentative distance as key. Calls to *decreaseKey* are made if RELAX can improve the distance to a `reached` node. This step updates the key to the new tentative distance. Calls to *extractMin* are performed to get the smallest element of the queue. Elements obtained by this operation are known to be exact and can be set to `settled`. We sometimes refer to nodes currently contained in the priority queue as *active nodes*.

2.3 Implementation

In this work, a number of techniques to speed up the computation of many-to-many shortest path queries are examined. An important goal of this work is to compare them using a competitive and efficient implementation. In this chapter, we present important aspects of our implementation, that is written in C++ using the Standard Template Library. A first implementation was made using the algorithm library LEDA [17] and its static graph data type, but we skipped this because of several difficulties. A main problem therewith was an obscure time behaviour that did not correspond to the operation counts we made. Our STL based implementation was at least 25% faster than the one using LEDA static graphs. Also, some preliminary experiments were made with the by far slower general graph type of LEDA. So, all time measurements that we refer to in this work use our graph implementation based on the STL. In a few cases we present results of the LEDA static graph implementation in terms of settled nodes.

2.3.1 Data Structures

The basic data structure we need is a representation of the graph, suitable to perform fast shortest path queries. The structure of the graph is assumed to be static, no new edges and nodes are introduced or deleted after the graph was read once. This assumption applies only when we want to answer queries, a possible preprocessing step is not considered here and is assumed to be part of the input for now. A static graph allows the usage of the efficient forward star representation of graphs. This is our method of choice for implementing the graph data structure. Schultes [23] used this representation in his implementation of Highway Hierarchies, enriched with an additional layer for the hierarchical information. Also the implementation of LEDA static graphs [21] is based on this data structure.

The *forward star* (or *adjacency array*) *representation* stores the complete graph in two arrays, one for the nodes and one for the edges. The position within those arrays induces a numbering of the nodes and edges. Nodes can be stored in arbitrary order, edges must be sorted by the index of its source node. Every node stores the index of its first outgoing edge. Due to the sorting, all of the nodes outgoing edges are stored on consecutive positions. This construction is sufficient to represent the graph.

Using this forward star representation, iterating efficiently over all outgoing edges of a node is easy. To search the reverse graph we need an also efficient way to traverse all incoming edges. For this purpose, we store every directed edge twice, and maintain two flags for each edge. The first flag specifies if an edge may be passed in the forward direction, the second flag shows that it can be passed backwards. If there are two edges (u, v) and (v, u) with the same weight, they can be summed up by using only one edge representation for both, and allowing the forward and the backward direction.

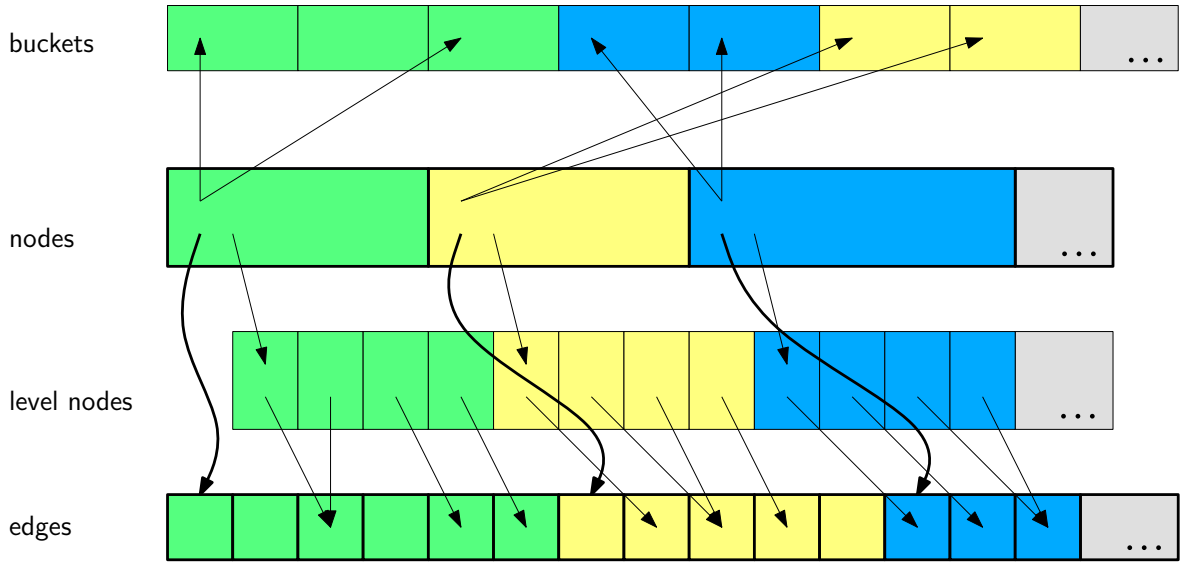


Figure 2.2: *Forward star graph representation, extended by level nodes and buckets. Data structures that are used to store the raw graph are indicated by the thicker lines.*

Level Nodes

For Highway Hierarchies, the algorithm presented in Chapter 4, we additionally store subgraphs $G \subset G_1 \subset \cdots \subset G_k$ of the graph G. For this, we use a *level node* layer that was introduced in [23]. Every node has some additional level nodes that are stored in an adjacency array in the same way as edges are stored in the forward star representation. Those level nodes point to the first outgoing edge of a node in its level. Hence, this construction induces subgraphs and also allows additional attributes to be stored at nodes for each level seperately.

Buckets

Buckets are a concept that is needed later by techniques explained in Section 3.2 and Chapter 4. Because both approaches use bidirectional search, the forward and backward search spaces have to be intersected in some way. Therefore, we introduce a bucket data structure that provides a set of operations to handle this intersection. We store a set of pairs (d, t) at each node, where d is a number representing a distance and t is the index of a target node. We call such a pair *bucket entry*, the number of bucket entries that are stored at a node can reach from 0 to $|T|$. We differentiate between an initial state, where all buckets are empty, and a *write phase* that is followed by a *read phase*.

During the write phase we need an operation `v.insert(d,t)`, that stores a pair (d, t) at a node v. After all insert operations are done, i.e. the write phase is finished, the buckets can be switched to the read phase. Therefore an operation `v.scan()` is needed,

which allows to iterate over all pairs (d, t) that have been attached to v by previous insert operations. Note that during the write phase no `scan` operations, and during the read phase no `insert` operations are performed.

One could use basic algorithmic data structures to implement this. A linked list for every node could store the pairs, also an resizeable array is a possible implementation. For both of them several drawbacks appear. For dynamic arrays, during the write phase a lot of resize operations are necessary. This for itself takes a lot of time and also leads to memory fragmentation. Linked lists are slower during the read phase, since there is an additional indirection for every bucket entry. If one wishes a sorting of the stored pairs, linked lists are also not very suitable.

An implementation that avoids the drawbacks listed in the previous paragraph can be done using an adjacency array representation of the buckets. This technique makes use of the separation into a read and a write phase. During write phase one large array is maintained to store entries made by insert operations, `v.insert(d,t)` simply appends a triple (v, d, t) at the end of the array. After the write phase is completed, the entries of this array are reordered: nodes are grouped by v. Then, at each node that has at least one bucket entry we store the index of its first entry and the number of entries at this node. From here on the information about v is implicitly represented by the grouping and indexing of this array. Hence, we need only to store pairs (d, t) instead of the full triples. This reduces storage size and increases memory locality. Now, bucket entries can be accessed as adjacency arrays.

The grouping that has to be performed when we switch from write to read phase, can be done very efficiently in linear time by a slight modification of counting sort. Preliminary experiments showed that this adjacency array representation of buckets is more than two times faster than an implementation that uses dynamic arrays, even though we have to spend the extra effort for grouping the array of buckets by its corresponding nodes v.

2.3.2 Implementation

A major point in this work is the implementation of the developed many-to-many shortest path algorithms. Main point of view thereby is the efficiency of our implementation to get sound experimental results. Because various speedup techniques are examined, also the flexibility of the framework is an important issue. We want to switch easily between several variants of the algorithms and it would be practical to have one framework to perform time measurements and create algorithm visualizations.

Dijkstra's Algorithm is implemented as a basic member function of a class with *hooks* where aspects can plug in their functionality. The aspects we speak of modify or extend the basic algorithm, e.g. they implement a speedup technique, perform operation counting or visualize the algorithm. The actual source code of our C++ implementation of Dijkstra's Algorithm with hooks is shown in Figure 2.4. If one would realize hooks as virtual functions, aspects could be realized by derived classes that overwrite these virtual functions. They

can perform additional work there and eventually have to call the base class function again. More flexibility could be added to this basic object oriented idea by using parametrized inheritance, as suggested in [34] (Chapter 4, Implementation). In this generic programming approach the base class of an aspect is a template parameter. So, different aspects can be combined freely by plugging template parameters together when an actual object is instantiated.

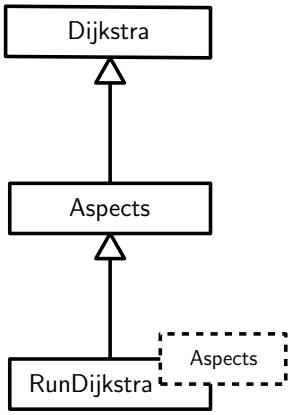
Although this method provides flexibility, virtual function calls, especially in inner loops, are prohibitive in the implementation of efficient algorithms. The reason is that compiler optimisation that perform function inlining can not be applied for virtual function calls in general. The compiler has to generate code for actual function calls, because during compile time it is not clear a priori which function has to be called. Especially for simple functions in inner loops consisting only of a few lines of code this is an immense overhead that can not be afforded.

But it is possible to keep the gain of flexibility of this approach without the drawback of virtual function calls. For this, we exploit the fact that there is only one spot where we need the functionality of virtual functions, namely in the hook enriched method of Dijkstra's Algorithm. Everywhere else, usual function calls are sufficient since there calls go upward in the class hierarchy. So, we outhouse this basic hook based method `run()` of Dijkstra's Algorithm in an extra class `RunDijkstra`. This class inherits from a base class that is given as template parameter. When the class is instantiated, the desired aspect can be inserted here. All data members remain in the base class `Dijkstra`, so they can be accessed from all classes. Figure 2.3 gives an schematic overview of this idea, where the aspect classes are enclosed by the two classes `Dijkstra` and `RunDijksta`.

To give an example for this, suppose we want to visualize the searchspace of Dijkstra's Algorithm that stops when all targets have been found. To create an algorithm object that deals with this, aspects can be plugged together at instantiation:

```
RunDijkstra<VisualDijkstra<AbortDijkstra> > dijkstra(Graph);
```

The basic algorithm implemented in `Dijkstra` always expands the search to the complete graph, `finished()` always returns false. We introduce a new class `AbortDijksta` that counts the number of found targets by overwriting the method `settled()` and overwrite `finished()` so that it returns true after $|T|$ targets have been found. Additionally we have a class `VisualDijkstra` that provides visualization functionality. `AbortDijkstra` inherits directly from `Dijkstra`, the base class of `VisualDijkstra` is given as template argument.




```
template<typename BaseDijkstra>
class RunDijkstra : public BaseDijkstra {
   // ...
};
```

Figure 2.3: *UML class diagram providing the idea of our class hierarchy. Aspects are given to* `RunDijkstra` *as a template argument.*

```
1  void run() {
2      init();
3      while (!finished() && !Queue.empty()) {
4
5          priorityType currentPrio;
6          st_node current = Queue.del_min(currentPrio);
7
8          settle(current,currentPrio);
9          if (!processNode(current)) continue;
10
11         st_edge e;
12         forall_out_edges(e,current) {
13             st_node dest = Graph.target(e);
14
15             if (!processEdge(e,current,dest)) continue;
16
17             priorityType newPrio = priority(dest,e,currentPrio);
18             if (isNodeNew(dest)) {
19                 insert(dest,newPrio,current);
20             } else {
21                 decrease(dest,newPrio,current);
22             }
23         }
24     }
25     post();
26 }
```

Figure 2.4: *Implementation of Dijkstra's Algorithm using C++. Function calls at key points of the algorithm provide hooks to add aspects to the algorithm.*

2.4 Experimental Setup

In order to evaluate the implemented speedup techniques we drove extensive experiments. In this section we present the environment of our experiments, the utilized road networks and the instances of distances table requests we used. The experimental results itself can be found embedded in the chapters, where the respective techniques are presented. Additional results can be found in Appendix A.

2.4.1 Environment

We performed the experiments on two 64-bit machines with 8 GB and 16 GB of main memory, respectively, 1 MB L2 cache using one out of two AMD Opteron processors clocked at 2.6 GHz, running SUSE Linux 10.1. Our programs were compiled with the GNU C++ compiler version 3.4 using optimisation level 3. To plot analyses of the experimental results we used the statistical program package R [22]. For visualisation purposes we used the geographical software package MapInfo [18].

Goal of this work is to optimise the running time of many-to-many shortest path algorithms. Hence, the main quality measure is the overall running time. We will sometimes use the term *speedup*: The factor of runtime reduction compared to the plain version of Dijkstra's Algorithm that is stopped after a shortest path to every target $t \in T$ is found. A second measure we sometimes use is the number of settled nodes as an implementation independent information.

2.4.2 Road Networks

Almost all experiments deal with the detailed road network of Western Europe. It covers the 14 European countries Austria, Belgium, Denmark, France, Germany, Italy, Luxembourg, the Netherlands, Norway, Portugal, Spain, Sweden, Switzerland and the UK. In some cases we consider queries in the graph of the road network of Germany. For some experiments we also used the road network of North America that encompasses Canada and the USA with all Federal States. Data was provided by the company PTV AG for research purposes. The original graphs contain for each edge a length in meter and a speed

	Europe	USA / CAN	Germany
#nodes	18 029 721	18 741 705	4 378 447
#edges	42 199 587	47 244 849	10 668 389

Table 2.1: *Number of nodes and number of directed edges in the road networks of Europe, North America and Germany*

category. There are 13 different speed categories. Edge weights are obtained by assuming average speeds of 10-130 km/h for the European road network and 16-112 km/h for North America. For Germany the 10-130 km/h setup applies too, if not stated otherwise.

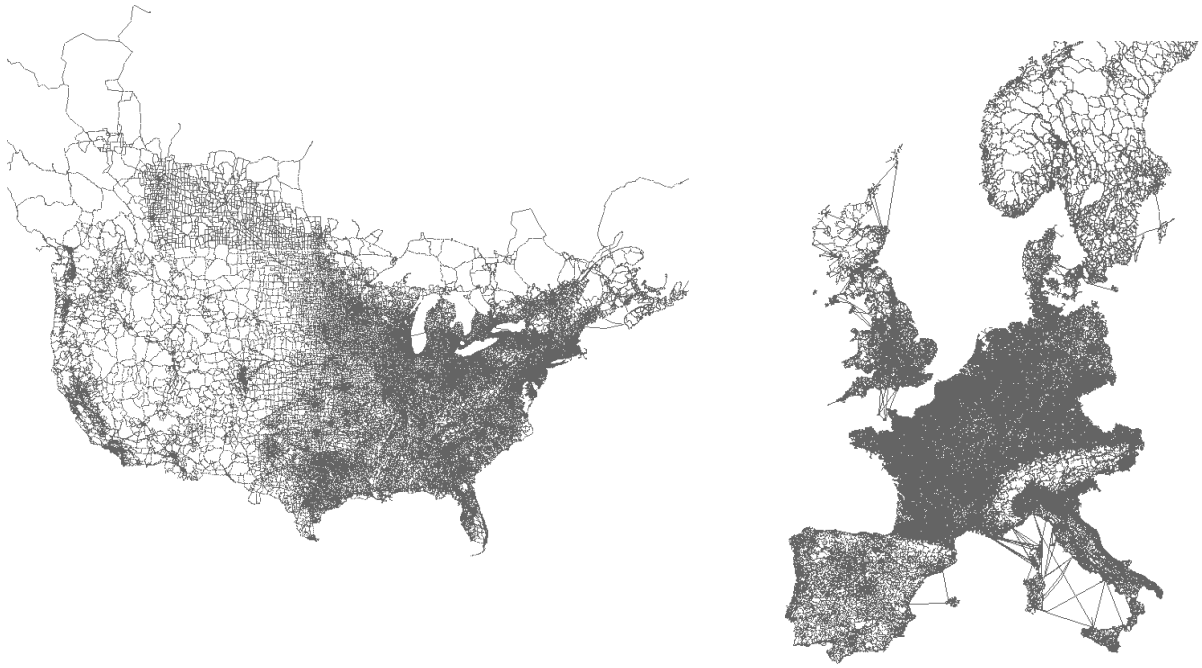


Figure 2.5: *Road networks of Europe and North America. In this picture local roads are not drawn.*

2.4.3 Requests and Instances

For our experiments we use fixed sets of requests. What we call a *request* is a pair of sets of nodes (S, T), $S \subset V$, $T \subset V$. For a many-to-many shortest path request we want to know the distances $d(s, t)$ for all $(s, t) \in S \times T$. This leads to a setup for our experiments. The first step is to generate requests. The next paragraph describes the methods that are used to do this. The second step is to precompute a distance table for every request using the plain version of Dijkstra's Algorithm. Then the complete matrix is stored together with the corresponding request. With such sets of fixed requests we have a robust method to compare different algorithm variants. Stored distance tables enable us to do some sort of program checking. After time measurements are finished we compare all stored and newly computed matrix entries. Assuming a correct Dijkstra implementation, with this we have proven the implementation of our speedup techniques to be correct for every input instance of the experiments.

Request Generation

We need methods to generate requests (S, T) for our experiments. In the following we present three methods that we used to generate requests. Note that we ensured for all generation methods the mathematical property of a set to contain an element at most once.

Real World Instances We use nine instances of many-to-many requests stemming from real world vehicle routing problems. The data was provided by the company PTV AG and was given in form of geographical coordinates of locations in the road network of Western Europe. Attaching each coordinate to its closest node in the graph yields the sets $S = T$. Table 2.2 provides an overview of size and location of the node sets. All experiments comprehend the nodes as part of the graph of Western Europe and perform the experiments within this graph. Exemplarily two of the real world instances are depicted in Figure 2.6.

Random Instances To generate random queries we pick random nodes of the graph to create the sets S and T. This method leads to an even distribution of the locations over the graph. If nothing else is stated, for quadratic requests we choose the node sets such that $S = T$.

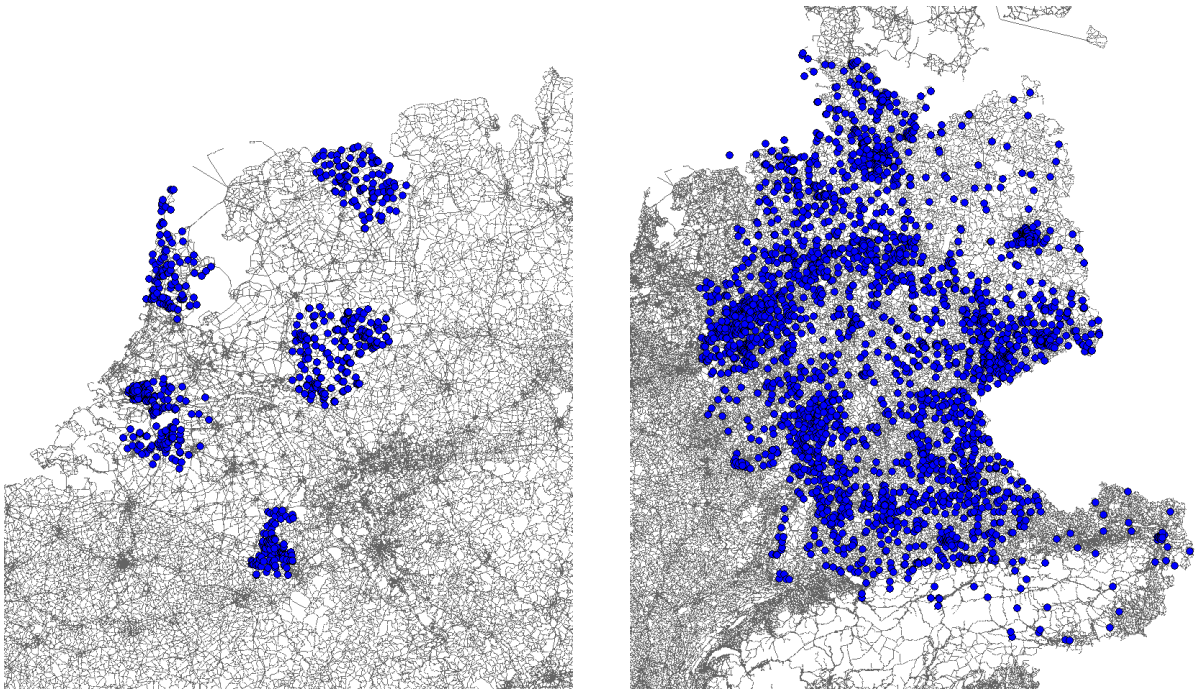
Clustered Random Instances We have two parameters to generate clustered instances: The number of clusters q and the cluster size k in number of nodes. The method works the same like the generation of random instances, but restricted to a subset $C \subset V$ of the nodes.

Let $N_k(s) \subset V$ be a set of nodes with $|N_k(s)| = k$. We call $N_k(s)$ the k-Neighbourhood of s, if for every $v \in V$, $w \in N_k(s)$ the implication $d(s, v) < d(s, w) \Rightarrow v \in N_k(s)$ holds. We denote a fixed k-Neighbourhood of s by $\mathcal{N}_k(s)$.

Then to define C we pick q random nodes c_i, $0 < i \leq q$, and set $C := \bigcup_{i=1}^{q} \mathcal{N}_k(c_i)$.

	Description	#Locations
Real World 1 (RW1)	BeNeLux, slightly clustered	173
Real World 2 (RW2)	BeNeLux, nine clusters	325
Real World 3 (RW3)	Ruhr Area, slightly clustered	354
Real World 4 (RW4)	BeNeLux, five clusters	784
Real World 5 (RW5)	BeNeLux, one cluster	800
Real World 6 (RW6)	Germany, uniformly distributed	990
Real World 7 (RW7)	Germany, uniformly distributed	1463
Real World 8 (RW8)	BeNeLux, uniformly distributed	2403
Real World 9 (RW9)	Germany and Austria, slightly clustered	2892

Table 2.2: *Description of real world instances located in the road network of Europe that we used in our experiments.*



(a) Real World 4 (b) Real World 9

Figure 2.6: *Locations originating from real world instances located in the road network of Europe.*

Chapter 3

Without Preprocessing

For practical applications it is important to be able to configure shortest path requests in road networks. For example, user defined vehicle profiles define variable speeds for several types of street segments or prohibit the usage of certain edges, trucks transporting hazardous materials are not allowed to use every route or the usage of bridges, tunnels or pass roads can be limited due to the weight or size of the vehicle. Also up to date network information, that takes current events like traffic jams or the weather situation into account, can influence edge weights. The usage of time dependent edge weights that try to predict realistic travel times is another example for a situation where flexibility is important.

The examples from the preceding paragraph show that in practice the input graph not always can be considered as static. This is an requirement for the various recent speedup techniques for point-to-point shortest path queries that rely on a preprocessing step. Hence, we study in this chapter many-to-many techniques without preprocessing. Furthermore, due to the fact that we are not aware of any literature about specific results that would speed up many-to-many shortest path queries in our situation, it is a natural approach to study the adaption of classical point-to-point speedup techniques.

For the acceleration of point-to-point queries basic techniques that can do without preprocessing are bidirectional search and goal-directed search. Both are investigated in detail in this chapter and variants of the algorithms to solve the many-to-many shortest problem are developed. We start with the goal-directed approach using the common way of estimation the distance to the target geometrically. This is followed by an approach that turns landmark based goal-directed search into a many-to-many technique without preprocessing. The second part of this chapter deals with bidirectional search. This turns out to adapt to a practical approach in the situation without preprocessing. Crucial ideas of this algorithm are further used in the highway hierarchy based algorithm of Chapter 4 that is based on a bidirectional approach.

3.1 Goal-Directed Search

Goal-directed Search, also known as *A* Search*, is a technique originating from the field of artificial intelligence. It is based on a distance approximation that provides a lower bound for distances between nodes. Heuristics based on distances are used to give the search a sense of direction to guide it towards a target.

Usually, goal-directed search is used with one source and one destination. The aim of this section is to study how these ideas can be applied to the many-to-many shortest path problem. To estimate distances to target nodes we introduce so called *potential functions* in Section 3.1.1 and present general properties. First, we introduce some general properties of potential functions to be able to apply this technique to the computation of many-to-many shortest paths. Then, we turn to concrete potentials, starting with the examination of lower bounds given by geometric distances, followed by an approach that shows how a landmark based approach can be turned into a technique without preprocessing.

3.1.1 Potential Functions

To introduce general properties of potential functions we follow the depiction in [9] and [10]. A *potential function* $\pi : V \to \mathbb{R}$ is a function from nodes to reals. For a given potential we define the reduced cost of an edge as $l_\pi(v, w) = l(v, w) - \pi(v) + \pi(w)$.

Lemma 3.1. *Every shortest path from s to t with respect to edge weights given by l is also a shortest path with respect to the reduced edge weights l_π.*

Proof. For a fixed potential function the length with respect to l_π of an arbitrary path from s to t differs only by the constant value $\pi(t) - \pi(s)$ from the length of a path with unmodified weights. □

Dijkstra's Algorithm requires the graph to contain no edges with negative weights. This leads to the following

Definition 3.2. *A potential function π is called* feasible*, iff $l_\pi(v, w) \geq 0 \; \forall \; (v, w) \in E$.*

With such a feasible potential function we can apply Dijkstra's Algorithm to the graph with the corresponding reduced edge weights. As shown in Lemma 3.1 this yields shortest paths for the graph with unmodified edge weights.

The next Lemma explains why we can think of the potential function as a lower bound of distances:

Lemma 3.3. *Suppose π is feasible and for a vertex $t \in V$ we have $\pi(t) \leq 0$. Then for any $v \in V$: $\pi(v) \leq d(v, t)$.*

Proof. The distance with respect to π is a sum of non-negative edge weights, hence, the distance is also not negative:

$$\begin{aligned} d_\pi(v,t) &= \sum_{(u,w) \in P|_{v \to t}} l_\pi(u,w) \\ &= \sum_{(u,w) \in P|_{v \to t}} l(u,w) - \pi(u) + \pi(w) \\ &= d(v,t) - \pi(v) + \pi(t) \\ &\geq 0 \end{aligned}$$

It follows that $d(v,t) + \pi(t) \geq \pi(v)$ and with the assumption that $\pi(t)$ is not positive we have $\pi(v) \leq d(v,t)$. □

According to [31] we can combine feasible potential functions in various ways:

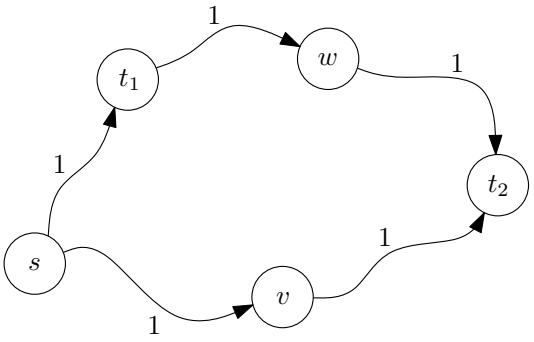
Lemma 3.4. *If π_1 and π_2 are feasible potential functions, then $p = \max\{\pi_1, \pi_2\}$, $p = \min\{\pi_1, \pi_2\}$ and any convex linear combination are feasible potential functions, too.*

3.1.2 Changing Potentials Online

During the following paragraph we suppose all potentials to be feasible. In several situations it is useful to change the potential function during a Dijkstra run. We call this an *online change* of the potential function and show that this is possible while preserving correctness. During Dijkstra' Algorithm is running, we have found a shortest path to every node that is in the state `settled`. Hence, with Lemma 3.1, we can conclude that those nodes can remain `settled` if we switch to another potential function. So, what we have to deal with are the `reached` nodes which are managed in the priority queue.

The counterexample in Figure 3.1.2 shows that continuing the algorithm without changing its keys in the queue leads to incorrect shortest paths. We now explain what goes wrong in this case. First, note that π_1 and π_2 are feasible. We start using π_1 as our potential function. Dijkstra's Algorithm starts with scanning s, so t_1 and v are inserted into the queue with $d_{\pi_1}(t_1) = 1, d_{\pi_1}(v) = 3$. Then, t_1 is scanned and w is inserted into the queue with $d_{\pi_1}(w) = 2$. Now, the first target node t_1 is `settled` and we change to the potential function π_2. The next step is to scan w, thereby t_2 is inserted with $d_{\pi_2}(t_2) = 2$. Beside t_2 now only v is in the queue with $d_{\pi_1}(v) = 3$. So, t_2 is scanned next, although v is involved in the correct shortest path to t_2. Hence, this method leads to a path with length three, although there is a path of length two.

To restore the correctness of the algorithm we follow the idea of [9] and refresh the priority queue after the potential was changed. In [9] this is proven to be correct in the section about *restarting*. Refreshing the priority queue is easy, all we have to do is to reinsert all `reached` nodes with their new tentative distance with respect to the new potential.



	π_1	π_2
s	0	1
v	2	0
w	0	1
t_1	0	2
t_2	1	0

Figure 3.1: *This counterexample shows that refreshing the priority queue is necessary.*

3.1.3 Many-to-Many Algorithm

To run a goal-directed search in the standard case of a point-to-point query we use Dijkstra's Algorithm on a graph with edge weights modified by a potential function. This construction drives the search towards the sole target. But in our situation there are multiple targets located in different directions, so we need to modify the approach. Now, we have for every node in the set of targets $T = \{t_i : 0 \leq j < N\}$ a potential function π_i that drives the search towards that target. In the next paragraphs we point out two techniques of goal-directed search that can be applied to the many-to-many shortest path problem.

Minimum Potential. According to Lemma 3.4 we can combine the potentials by taking the minimum $\pi_{\min} := \min\{\pi_i : 0 \leq j < N\}$ over all target specific potential functions. This minimum potential yields low values for nodes near one target, but high potentials for nodes in areas away from the targets. The query algorithm remains unchanged, we just use an adapted potential function.

Sequential Search. The second approach is to search only towards one target at the same time. We start the search using π_k with $k = 1$. After t_k was found we use π_j as the current potential, where j is the smallest $j > k$ with the state of t_j not being `settled`. We continue until all targets are found. This switching of potential functions during a Dijkstra run still yields correct results if we rearrange the priority queue every time we switch to a new potential as described in Section 3.1.2. One observation is that different orders of the targets result in different search space sizes. In our experiments we use two variants: one that searches towards the nearest, and one that searches towards the farthest target that has not been found yet.

Until here, we made general remarks about potential functions. With this foundation we can turn to concrete lower bounds now. The following sections deal witch actual potentials that are computed using geometrical information or by information provided by a set of implicit landmarks.

3.1.4 Geometric Potential Functions

The traditional approach for goal-directed search in road networks uses a layout $L : V \to \mathbb{R}^2$ of the graph, which for road networks is mostly given by geographic data. Let $||L(v) - L(t)|| : V^2 \to \mathbb{R}$ be a metric called *flight distance* that gives an approximation for the shortest path distance of two nodes. In particular, the triangle inequality holds for flight distances. For classic goal-directed search Euclidean distances are used. We define the *maximum velocity* of the graph as $v_{\max} := \max\{\frac{||L(u)-L(v)||}{l(u,v)} \mid (u,v) \in E\}$.

Lemma 3.5. $\pi(v) := \frac{||L(v)-L(t)||}{v_{\max}}$ *is a feasible potential function.*

Proof.

$$\begin{aligned}
l_\pi(u,v) &= l(u,v) - \pi(u) + \pi(v) \\
&= l(u,v) - \frac{||L(u) - L(t)||}{v_{\max}} + \frac{||L(v) - L(t)||}{v_{\max}} \\
&= \frac{l(u,v) \cdot v_{\max} + ||L(u) - L(t)|| - ||L(v) - L(t)||}{v_{\max}} \\
&\geq \frac{l(u,v) \cdot \frac{||L(u)-L(v)||}{l(u,v)} + ||L(u) - L(t)|| - ||L(v) - L(t)||}{v_{\max}} \\
&= \frac{||L(u) - L(v)|| - ||L(v) - L(t)|| + ||L(u) - L(t)||}{v_{\max}} \\
&\geq 0
\end{aligned}$$

□

Hence, a layout of the graph provides a potential function that can be used for goal-directed search. Using such a potential function arising from geographical information is the classic way to apply goal-directed search to graphs representing street networks. For the considered problem of computing many-to-many shortest paths this geographical potential provides three techniques of goal-directed search: we can use the minimum potential (similarly explained in [29]) and we can use sequential search as described in the preceding paragraph. We will refer to these techniques in experimental evaluations as:

GoalMin Combining potential functions by taking its minimum.

GoalSeqN Searching towards one target at the same time, nearest first.

GoalSeqF Searching towards one target at the same time, farthest first.

Flight Distances and Edge Weights

The performance of goal-directed search is dependent on the quality of the distance approximation. For geometric potentials this is provided as an interaction between edge weights and the determination of the flight distance from the layout of the graph. We studied the behaviour of both matters experimentally and explain several details about edge weights and flight distances in the following paragraphs. Note that this preliminary considerations are related to goal-directed search in general and not specifically tailored to the many-to-many problem. Nonetheless we consider this to be an important topic because the setup of our experiments for goal-directed search becomes more clear.

A setting that works well with goal-directed search is the usage of FLIGHT DISTANCE $||L(u) - L(v)||$ for edges (u, v). Practical settings that we compare in this section obtain the edge weights by computing travel times from a given length in meters and a velocity for every edge. Note that this edge length differs from the calculated flight distance because an edge can represent a curved route. The velocity for an edge is determined by considering the raw input that assigns one out of 13 road categories to every edge. Then, a vehicle specific velocity profile determines the travel time as weight for each edge. Hence, we compare several velocity profiles. The LINEAR profile uses linearly 130-10 km/h for the 13 road categories, MOTOR CAR uses typical average speeds of a car, MOTOR CAR 2 is a similar setting but with a 20% emphasis on travel distance that is realized by a scaling towards 50 km/h. Finally, CONSTANT uses a fixed velocity for all types of roads and so shortest paths in the corresponding graph give actual shortest travel distances.

LINEAR	130	120	110	100	90	80	70	60	50	40	30	20	10
MOTOR CAR	125	115	110	60	50	45	45	40	35	30	25	20	10
MOTOR CAR 2	110	102	98	58	50	46	46	42	38	34	30	26	18
CONSTANT	50	50	50	50	50	50	50	50	50	50	50	50	50

Table 3.1: *Velocity profiles assign a speed in km/h to each road category to determine a travel time as edge weight.*

For the computation of flight distances one has to recall the origin of the layout data. The given coordinates are the result of a map projection, mapping points from the curved surface of the earth on a plane. There are different projections that preserve certain geometric properties of the entities on the earths surface. As an instance of such projections we could mention conformal, equal-area or equidistant map projections. We need this geographical information to compute the flight distances. Then they are used as distance approximation and are related to the edge weights of the graph with the definition of a maximum velocity. Due to Lemma 3.5 this construction of lower bounds for distances gives correct results for arbitrary flight distances. However, to reduce the search space considerably, a good lower bound is necessary.

Here, the crucial point about map projections is the computation of flight distances. If

we treat the coordinates as points of the plane $\mathbb{R}^2$ we can use Euclidean distances. This is a simple and in particular fast method because no expensive trigonometric computations are required. The drawback of Euclidean distances is that they are not exact flight distances that are actually given by an arc of a great circle, an orthodrome, rather than a straight line. This effect becomes more important, the more the street network expands in north, south, east and west direction. For large traffic graphs encompassing a broad range of longitudes and latitudes, this effect is not negligible anymore.

Independent of the concrete map projection that is used, the computation of an exact flight distance between two points is expensive because of the required trigonometric computations. So, in terms of the regarded goal-directed search algorithm there seems to be a tradeoff at this point: Due to the better distance approximation the search spaces with the exact flight distance computation are smaller, but using Euclidean distances affords less time per node because no trigonometric computations have to be performed. A way to get along with this is to use an approximation for the flight distances. Results of the experiments we made show that this works quite well. Our data uses a *plate carrée projection* that is obtained by comprehending the decimal representation of latitude and longitude as coordinates in the plane. This cylindrical projection shows a stretching in east-west direction that is the stronger the closer one gets to a pole. So the approximation we use applies a correction factor for the latitude that depends on the longitude of a point that involves only one cosine computation—in contrast to five trigonometric functions used for the exact flight distance. Then, Euclidean distances are used on the so corrected coordinates.

Experimental results in terms of speedup relative to Dijkstra's Algorithm are given in Figures 3.2 and 3.3, results in terms of absolute numbers can be found in Table 3.2 and Appendix A. The strong difference between runtimes is at a first glance somewhat surprising, because search space sizes do not differ significantly. This can be explained by the number of *decreaseKey* operations. We see a big improvement of the search space when we switch from Euclidean to exact distances, whereas regarding query time, Euclidean distances are still slightly faster. They are both outperformed by the distance approximation that shows a very similar search space gain as exact distances without loosing this performance in terms of query time. Hence, for all further considerations of geometrical goal-directed search we use this distance approximation to determine the lower bounds.

	Dijkstra	Euclidean	Exact	Approx
Linear	959	593	616	468
Motor Car	964	614	600	456
Motor Car 2	858	523	525	389
Constant	595	355	432	294
Flight Distance	583	341	334	227

Table 3.2: *Average query times [ms] for different setups of goal-directed search for point-to-point queries in the road network of Germany.*

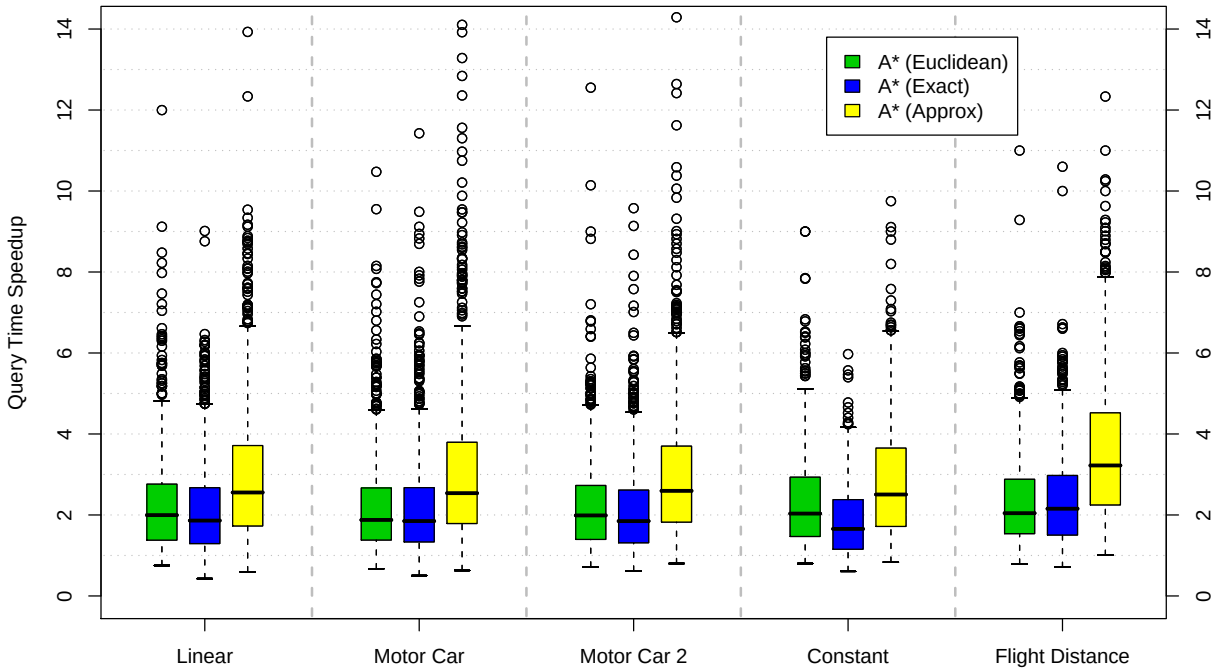


Figure 3.2: *Query time comparison of different setups of goal-directed search for random point-to-point queries in the road network of Germany*

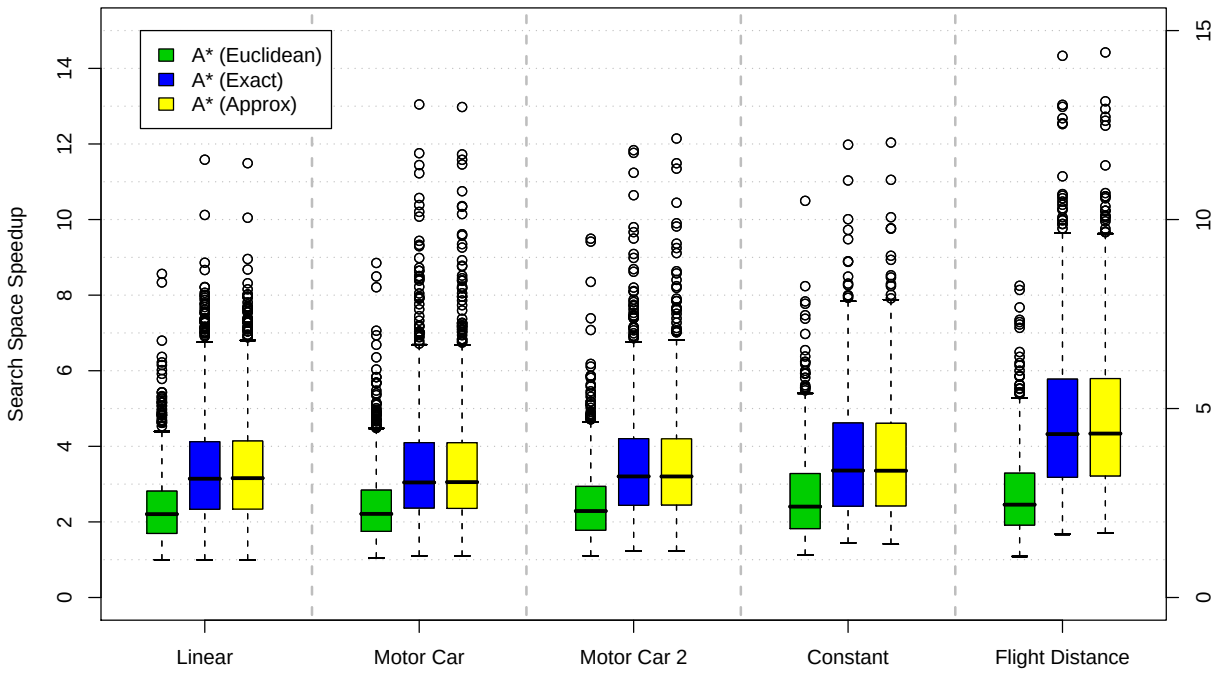


Figure 3.3: *Search space comparison of different setups of goal-directed search for random point-to-point queries in the road network of Germany*

3.1.5 Landmark based Potential Functions

The concept of landmark based goal-directed search that was proposed by Goldberg as ALT algorithm[1] takes advantage of an improved potential function to accelerate point-to-point shortest path queries. Better lower bounds for distances are obtained by a distance estimation that uses a set of prominent nodes, called *landmarks*.

Now, some nodes $L \in V$ are fixed as landmark nodes. Obviously, for shortest path distances d the triangle inequality $d(a,b) \leq d(a,c) + d(c,b)$ holds for all $a, b, c \in V$. With the prerequisite that all the shortest path distances to and from L are known, this property can be used to compute lower bounds on shortest path distances between arbitrary nodes in the graph. We will denote distances to landmarks by $D_{\text{to},L}(v) := d(v, L)$ and distances from landmarks by $D_{\text{from},L}(v) := d(L, v)$.

The triangle inequality yields $d(v,t) \geq D_{\text{to},L}(v) - D_{\text{to},L}(t)$ and $d(v,t) \geq D_{\text{from},L}(t) - D_{\text{from},L}(v)$ as lower bound for the shortest path distance $d(v,t)$ from a node v to a target t. Those lower bounds exactly provide the potential functions that can be used for landmark based goal-directed search: $\pi_{\text{from},L}(v) := D_{\text{from},L}(v) - D_{\text{from},L}(t)$, $\pi_{\text{to},L}(v) := D_{\text{to},L}(t) - D_{\text{to},L}(v)$. The proof that they are feasible is obvious:

Proof. Let $(v,w) \in E$ be an arbitrary edge of G. Then $l_{\pi_{\text{to},L}}(v,w) = l(v,w) + \pi_{\text{to},L}(w) - \pi_{\text{to},L}(v) = l(v,w) + D_{\text{to},L}(t) - D_{\text{to},L}(w) - D_{\text{to},L}(t) + D_{\text{to},L}(v) = l(v,w) - D_{\text{to},L}(w) + D_{\text{to},L}(v) \geq 0$. (Proof for $\pi_{\text{from},L}$ analogous) □

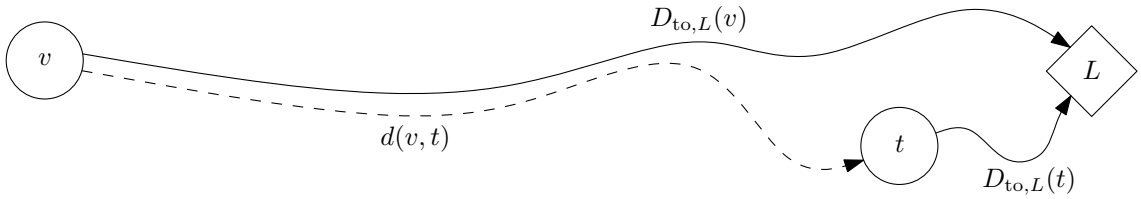
To obtain a tight lower bound these potentials can be combined by taking its maximum $\pi_L := \max\{\pi_{\text{to},L}, \pi_{\text{from},L}\}$. According to Lemma 3.4 this leads again to a feasible potential function. To combine potentials of several landmarks the maximum of their respective potential functions is taken.

Useful lower bounds

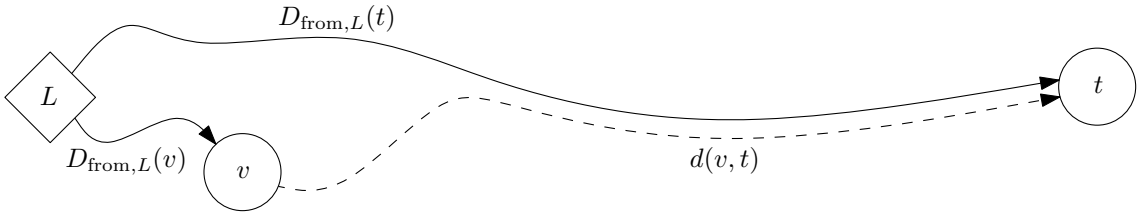
Figure 3.4 depicts schematically the effect of landmarks that show a pleasant behaviour. Viewing from the perspective of v in Figure 3.4(a) the landmark L lies behind the target node t. This means that a prefix of the shortest path from v to t and a prefix of the shortest path from v to L are closely related to each other. In a best case both paths even have a common subpath. In this situation Dijkstra's Algorithm is able to find this correct subpath immediately because the reduced cost weights of edges lying on this subpath are zero. Note, that in this situation the lower bound $\pi_{\text{to}}(v)$ that uses distances from nodes v to the landmark L is useful. Figure 3.4(b) shows the converse case where a landmark is situated ahead of the current node v and the potential $\pi_{\text{from}}(v)$ works well.

Following the considerations of the preceding paragraph we can state two main observations. First, it seems comprehensible that for landmarks it is advantageous to lie behind or ahead shortest paths. This is based on the idea that for a useful potential function

[1] **A***, **L**andmarks, **T**riangle Inequality



(a) *To* Landmark, $d(v,t) \geq D_{\text{to},L}(v) - D_{\text{to},L}(t)$



(b) *From* Landmark, $d(v,t) \geq D_{\text{from},L}(t) - D_{\text{from},L}(v)$

Figure 3.4: *Landmarks are prominent nodes in the graph that yield lower bounds for shortest path distances using the triangle inequality.*

shortest paths from (or to) landmarks should have commonalities with the shortest path we are looking for. Known techniques of landmark selection such as *avoid*, *maxcover* or *advanced avoid* ([9, 4]) make use of this observation and place landmarks near the border of a street network. In contrast, Figure 3.5 shows a case where a landmark does not provide a useful lower bound.

The *second* observation is, that both, the distances *from* and *to* landmarks are important. Regarding Figure 3.4, it is obvious that only one of them is useful at a time while the distance for the opposite direction yields a useless negative lower bound. Certainly notice, that if one is willing to trade off storage size against search space size, one could use the maximum of D_{to} and D_{from} for less memory consumption.

3.1.6 Many-to-Many Landmark Algorithm

The original version of landmark based goal-directed search for point-to-point queries [9] requires a preprocessing step. During this initial work a set of landmarks is determined and all distances between those landmarks and nodes in the graph are precomputed. In contrast, this chapter deals about techniques that do not require such a preprocessing step. We can resolve this contradiction by turning the landmark approach into a method without any preprocessing. To do this we comprehend some of the source nodes as *implicit landmarks*.

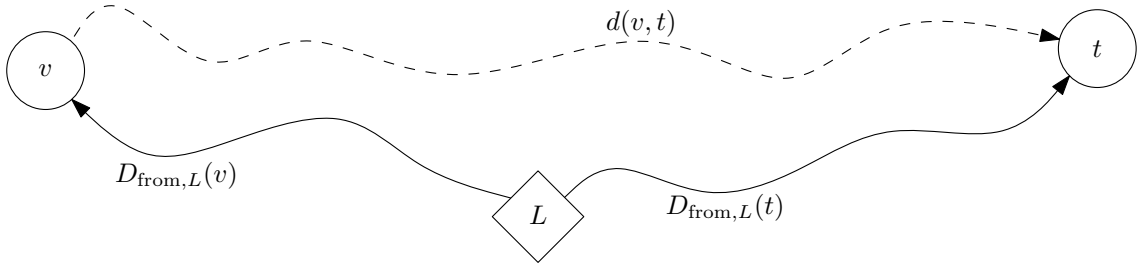


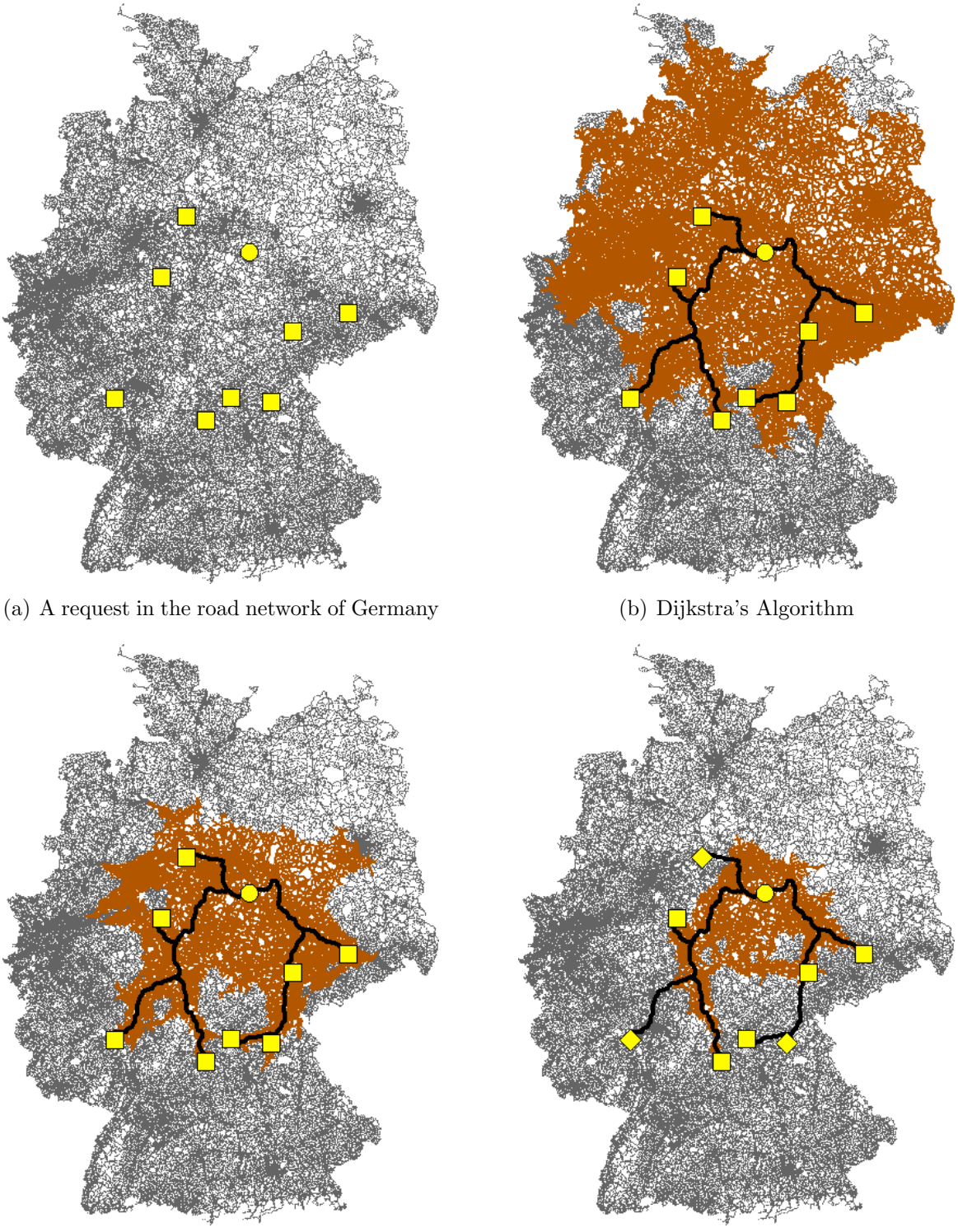
Figure 3.5: *A landmark between a node v and a target t does not yield tight lower bounds.*

The idea of this approach is to kill two birds with one stone: We compute desired shortest path information and get landmark distance information in one go. The algorithm works as follows: For tuning parameter k that fixes the number of landmarks we select source nodes $L_1, L_2, \ldots, L_k \in S$ as implicit landmarks. Then we run Dijkstra's algorithm for every landmark $L_i, 0 < i \leq k$ and store the thereby found distances from the landmarks to all nodes in the graph. For all further queries, starting from nodes $s \in S \backslash \{L_1, \ldots, L_k\}$, we use the maximum of the lower bounds on distances given by the inequality of Figure 3.4(b) for all landmarks $L_i, 0 < i \leq k$ as potential function for our sequential goal-directed algorithm.

To obtain a feasible potential function that is based on landmarks we must know the distances from landmarks to all nodes. Hence, we expand the searches from landmarks to the complete graph—we do not abort the search after all targets have been found. This extra effort is spent to accelerate all further searches from nodes $s \in S \backslash \{L_1, \ldots, L_k\}$. The potential function $\pi_{\text{uni}} := \max\{\pi_{\text{from},L_i} : 0 < i \leq k\}$ that is used here is weaker than the one that also uses distances *to* landmarks. In contrast to the original landmark method here distances to landmarks are not used. So, the question arises if this approach can be improved by using both, distances to and from landmarks. This requires an additional effort of complete backward searches for every landmark. In our experiments we see that this drawback is outweighed by the advantage of the better potential function.

It is important to choose suitable landmarks from the set of source nodes. Landmarks yield useful potential functions for the overall performance if they are positioned at the border of the graph. To respect this insight, we select the leftmost source node as the first landmark L_1. To determine a leftmost node the given geographical coordinates are used. All further landmarks are select by a technique that is called *farthest* ([10]) and uses a greedy method to select landmarks. Suppose that already $i < k$ landmarks are found. Then, the node $s \in S$ is chosen as the next landmark L_{i+1} that has the farthest distance to all landmarks $L_1, \ldots, L_i$.

We will refer to this algorithm in experimental evaluations as **LM K**. This denotes the many-to-many landmark algorithm using K landmarks and a potential that is obtained by distances *to* and *from* landmarks.



(a) A request in the road network of Germany

(b) Dijkstra's Algorithm

(c) Goal-directed search using a geometric potential

(d) Goal-directed search using landmarks

Figure 3.6: *Comparison of Dijkstra's algorithm, goal-directed search using a geometric potential and goal-directed search using landmarks. Landmarks are marked as diamonds, targets are drawn as squares and the circle shows the position of the source node.*

3.1.7 Experimental Results

We made our experiments for goal-directed search in the road network of Germany that has 4 378 447 nodes and 10 668 389 directed edges. We use matrix sizes between $|S| = |T| = 10$ and $|S| = |T| = 320$ for our tests with randomly generated requests. The random instances are generated as described in Section 2.4.3. We use RANDOM generated requests that are uniformly distributed in the complete graph without any clustering. 1 CLUSTER uses the method that generates clustered random instances with 1 cluster of size 1 000 000 nodes. 2 CLUSTERS are generated with a size of 100 000 nodes, 5 CLUSTERS with a size of 20 000 nodes per cluster.

The first experiment given in Figure 3.7 compares the basic algorithm variants of goal-directed search. There the number of settled nodes are considered. We see that goal-directed search using a combination of the target potentials by taking its minimum can reduce the search space for clustered input instances. Sequential goal-directed search works far better. There is almost no difference between the order of the targets we search for during this sequential many-to-many algorithm. Note, that for the random instances that are uniformly distributed in the graph almost no speedup is possible with goal-directed search. The reason is that Dijkstra's algorithm is pruned at the border of the graph and so a speedup can hardly be achieved by goal-directed techniques. For all further experiments we stick to the sequential goal-directed algorithm that searches towards the farthest target first.

Experimental results for sequential goal-directed search using different clustered requests in the road network of Germany are given in Figures 3.8, 3.9 and 3.10. Each bar in the box-and-whisker plots (see [22]) represents 50 fixed requests. For all table sizes and clustered random requests we see a speedup of more than two. The search space speedup is somewhat better than the query time speedup because of the additional effort per node. This extra time is spent for the computation of the geometric potential function and for refreshing of the priority queue. Goal-directed search allows a larger speedup for smaller matrices. For growing matrix sizes wee see a reduction of the speedup.

Considering the landmark technique, we first regard an experiment that examines if it is useful to spend extra effort for additional backward searches from landmarks to obtain better lower bounds. We compare the version that uses only distances from landmarks (*unidirectional landmarks*) with the one that uses both distances (*bidirectional landmarks*) and consumes the same amount of memory. Figures 3.11, 3.12 and 3.13 show that it pays to spend this extra effort. The improvement of the potential functions outweighs the drawback of the additional searches of the complete graph. So, for all further experiments we use bidirectional landmarks.

The next interesting question is how to choose the tuning parameter that determines the number of landmarks. The experiments given in Figures 3.14, 3.15 and 3.16 show that three landmarks are a reasonable choice. Probably a better landmark selection algorithm could improve the results for more than three landmarks. For point-to-point queries the method of active landmarks was shown to be useful. We made several preliminary experiments to

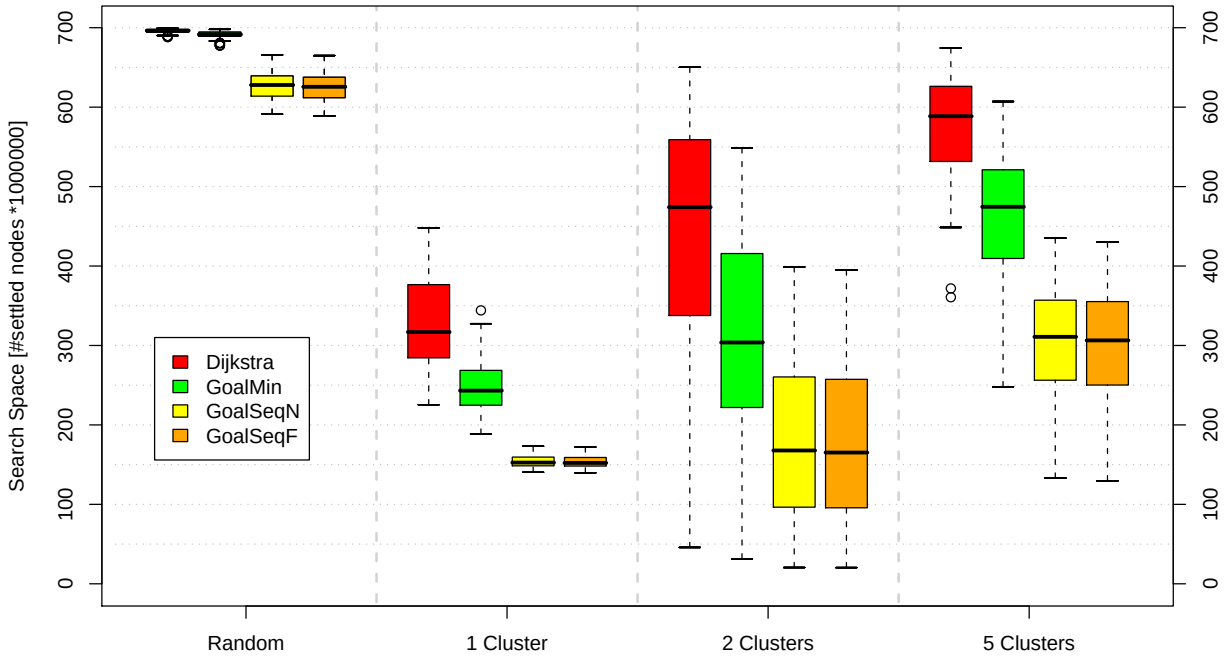


Figure 3.7: *Comparison of several variants of goal-directed search for random matrices with* $|S| = |T| = 160$. *Sequential goal-directed search yields smaller search spaces than goal-directed search using a combination of the target potential functions. The clustering of the input is important for the performance of goal-directed search.*

use only some 'active' landmarks for many-to-many queries without achieving considerable further speedup of search space or query time. However, considering the experimental results in terms of query time in contrast to the search space size (Figure A.3, A.4 and A.5) we see that the more landmarks are used, the more time is spend per node.

Experimental results for landmark based goal-directed search using different clustered requests in the road network of Germany are given in Figure 3.17, 3.18 and 3.19. Similar speedups as for goal-directed search with a geometric potential function can be observed. However, here for small instances the speedup is low because there a major part of the runtime is needed for the computation of landmark distances. The best speedups—about three or four times faster than Dijkstra's algorithm—are achieved for large instances with two clusters.

Further experiments concerning the real world instances in the road network of Europe are presented in Section 3.3.

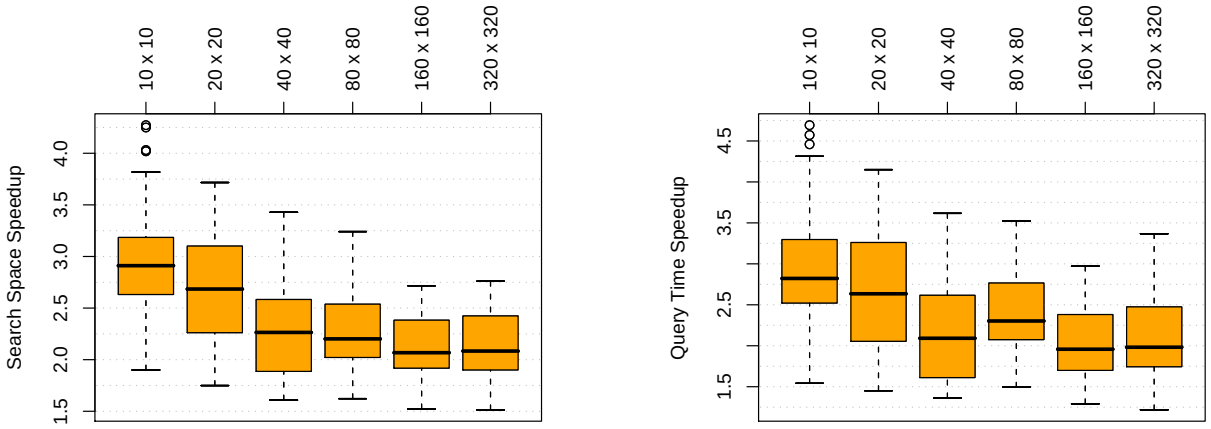


Figure 3.8: *Speedup of sequential goal-directed search in the road network of Germany for instances between* $|S| = |T| = 10$ *and* $|S| = |T| = 320$ *distributed in* 1 CLUSTER.

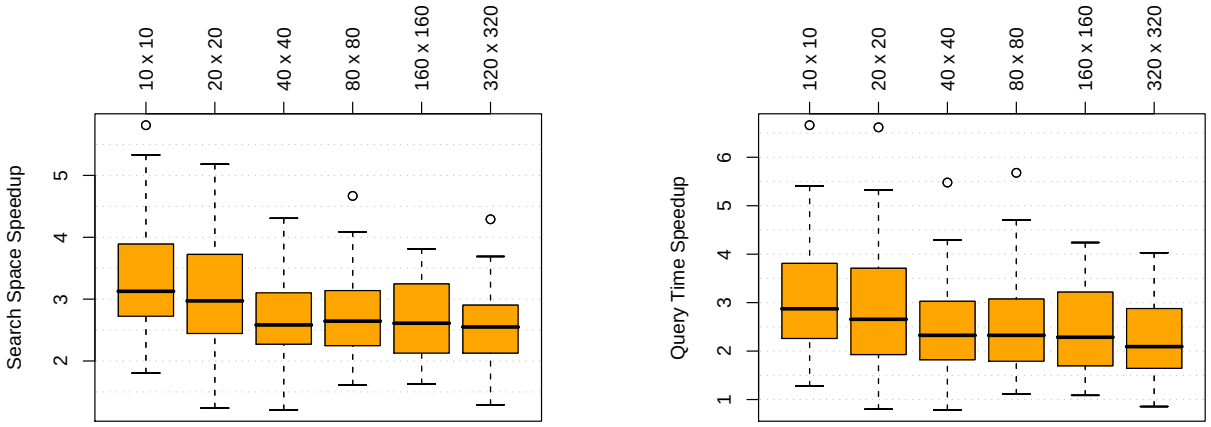


Figure 3.9: *Speedup of sequential goal-directed search in the road network of Germany for instances between* $|S| = |T| = 10$ *and* $|S| = |T| = 320$ *distributed in* 2 CLUSTERS.

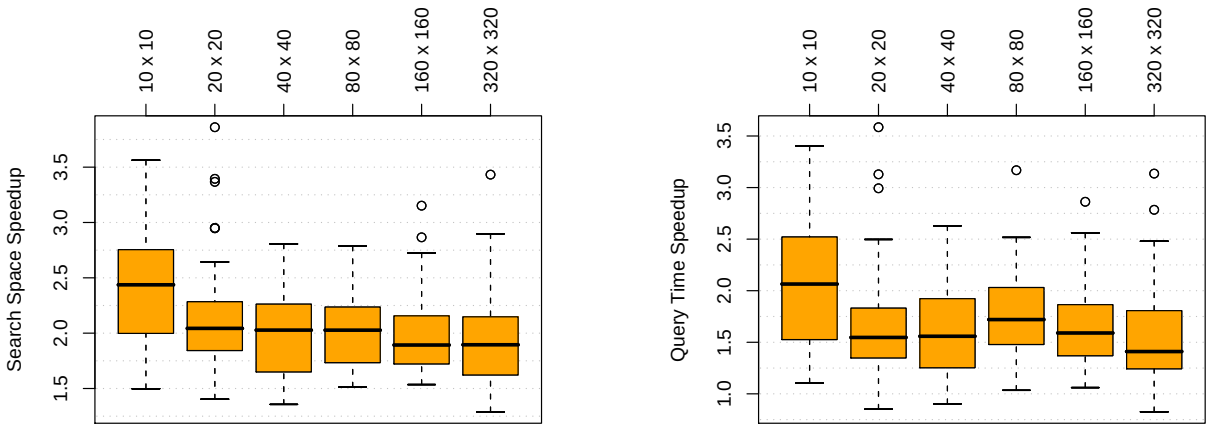


Figure 3.10: *Speedup of sequential goal-directed search in the road network of Germany for instances between* $|S| = |T| = 10$ *and* $|S| = |T| = 320$ *distributed in* 5 CLUSTERS.

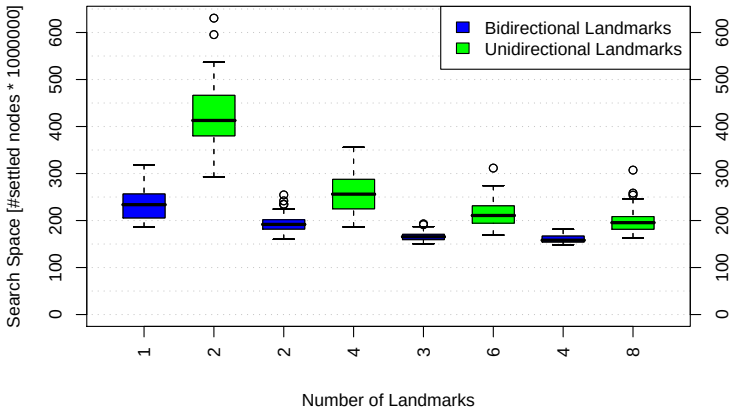


Figure 3.11: *Comparison of the search space of bidirectional and unidirectional landmark based goal-directed search in the road network of Germany for instances with* $|S| = |T| = 160$ *distributed in* 1 Cluster.

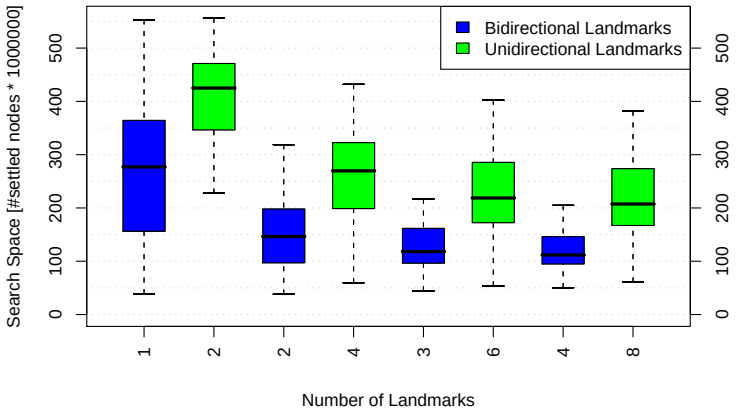


Figure 3.12: *Comparison of the search space of bidirectional and unidirectional landmark based goal-directed search in the road network of Germany for instances with* $|S| = |T| = 160$ *distributed in* 2 Clusters.

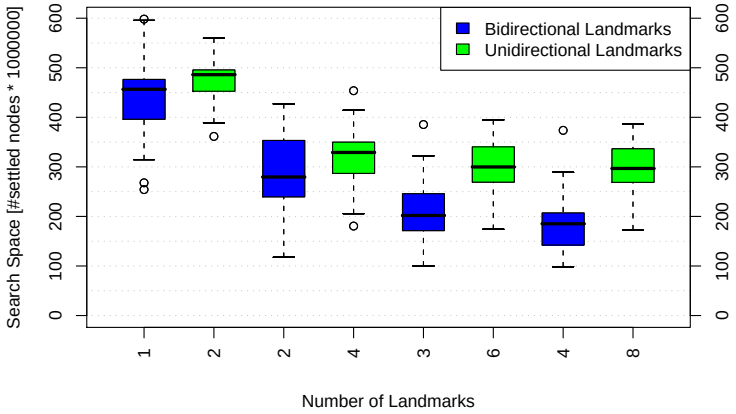


Figure 3.13: *Comparison of the search space of bidirectional and unidirectional landmark based goal-directed search in the road network of Germany for instances with* $|S| = |T| = 160$ *distributed in* 5 Clusters.

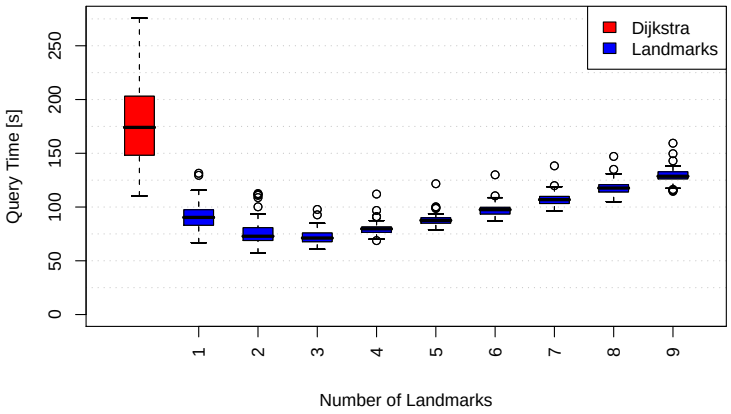


Figure 3.14: *Comparison of query time of landmark based goal-directed search with different choices for the number of landmarks in the road network of Germany for instances with* $|S| = |T| = 160$ *distributed in* 1 CLUSTER.

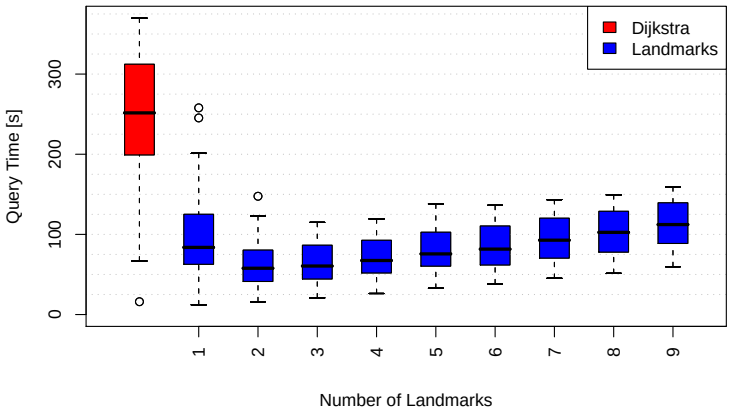


Figure 3.15: *Comparison of query time of landmark based goal-directed search with different choices for the number of landmarks in the road network of Germany for instances with* $|S| = |T| = 160$ *distributed in* 2 CLUSTERS.

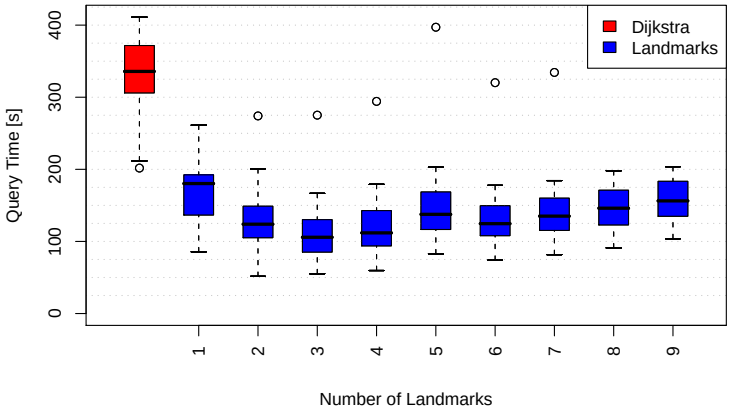


Figure 3.16: *Comparison of query time of landmark based goal-directed search with different choices for the number of landmarks in the road network of Germany for instances with* $|S| = |T| = 160$ *distributed in* 5 CLUSTERS.

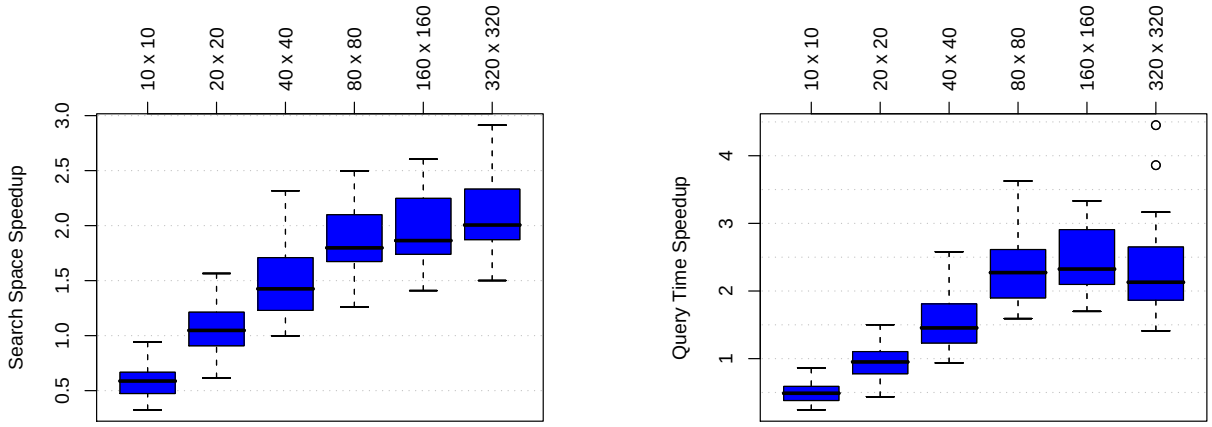


Figure 3.17: *Speedup of landmark based goal-directed search using three landmarks in the road network of Germany for instances between* $|S| = |T| = 10$ *and* $|S| = |T| = 320$ *distributed in* 1 Cluster.

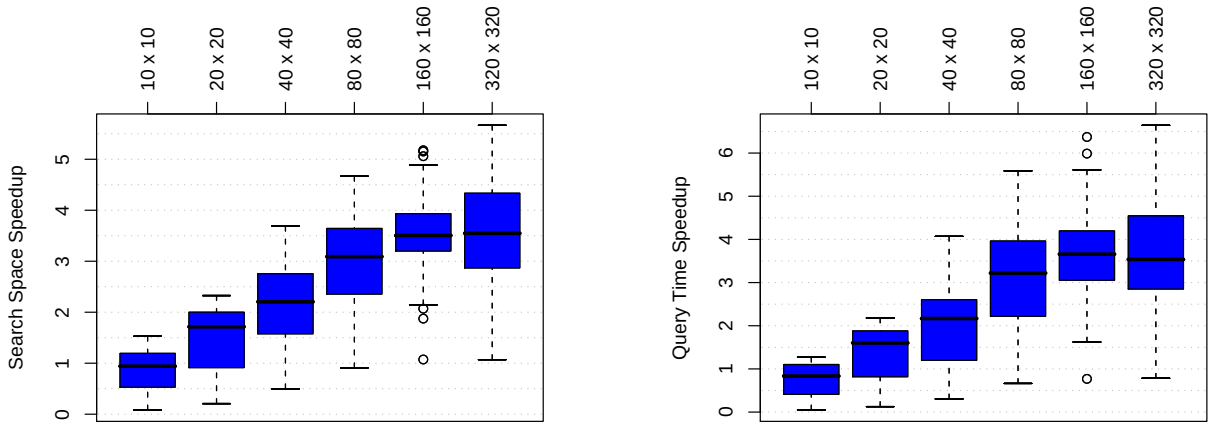


Figure 3.18: *Speedup of landmark based goal-directed search using three landmarks in the road network of Germany for instances between* $|S| = |T| = 10$ *and* $|S| = |T| = 320$ *distributed in* 2 Clusters.

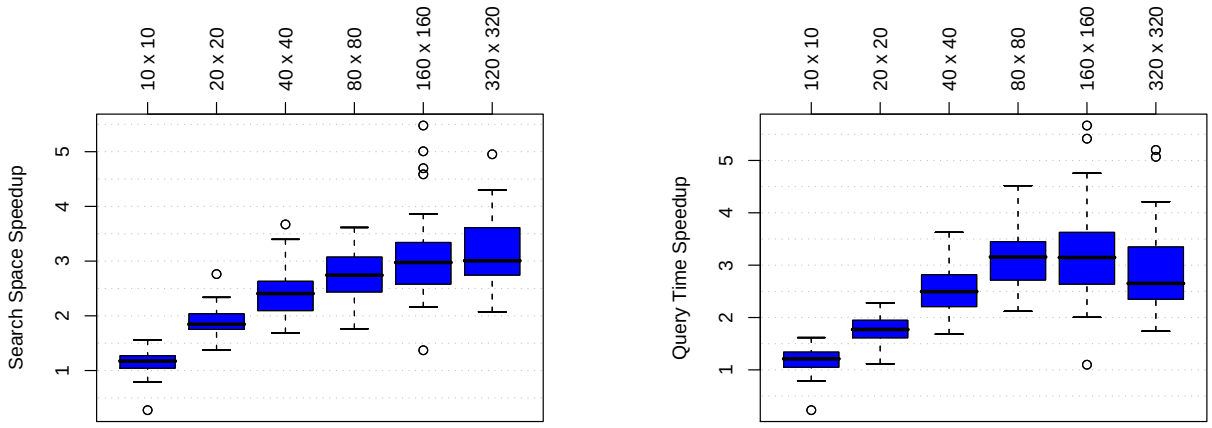


Figure 3.19: *Speedup of landmark based goal-directed search using three landmarks in the road network of Germany for instances between* $|S| = |T| = 10$ *and* $|S| = |T| = 320$ *distributed in* 5 Clusters.

3.2 Bidirectional Search

Bidirectional Search is a basic approach to accelerate Dijkstra's algorithm for point-to-point queries. It performs simultaneously a backward search from the target and a forward search from the source. The procedure can be finished if both search spaces have met. For our case where graphs represent road networks we can expect a search space reduction of about factor two. We can imagine the search space of Dijkstra's algorithm as a ball around the source node with radius r, so about r^2 nodes are visited. If we have two touching balls of half the size, the search space reduces to about $2 \cdot (\frac{r}{2})^2 = \frac{1}{2} \cdot r^2$ nodes.

While designing such a bidirectional search we must pay attention to an abort criterion that ensures correct results and a strategy to alternate between the forward and backward search. During the search we maintain the shortest path that has been found so far. We abort the search if a node becomes `settled` that was settled in the opposite direction before. Using Dijkstra's algorithm the distance of settled nodes are monotonic increasing, so the shortest path can not be improved from here. Hence, this abort criterion yields correct results. We use the following stopping criterion that is at least as strong as the one above and correct for the same reason.

Lemma 3.6 (Abort Criterion for Bidirectional Search)**.** *The bidirectional search can be stopped, if the sum of the minimum labels of the reached nodes for the forward and reverse searches is at least the length of the shortest path seen so far.*

For alternating between forward and backward search several strategies are possible. One can simply use a rotational strategy that processes a node of one direction and then a node of the opposite direction. Regarding cache issues, switching the direction after a larger number of nodes could improve the performance by taking advantage of memory locality. Another possibility for selecting the next node to be settled is to choose the direction with the minimum distance label. This leads to an implementation with only one priority queue for both directions.

Our goal is to adapt this speedup technique for point-to-point shortest paths queries to the many-to-many shortest path problem. There is no straightforward way to do this. The basic idea of bidirectional search to expand a forward search from every source and a backward search from every target until all (s, t) search spaces have met remains, but there are a lot of open questions about the details of this approach. Is it necessary to expand the searches in parallel or is there a practical sequential algorithm to process the numerous searches? How do we manage the intersections of the search spaces and how far do we have to expand the searches?

In the following we present a bidirectional speedup technique for the many-to-many shortest path problem that can be implemented as a sequential algorithm and leads to an appreciable speedup compared to the plain version of Dijkstra's Algorithm. Section 3.2.1 describes the general framework of our algorithm. An important sub task there is the determination of radii for the backward searches a priori. Section 3.2.2 deals with this topic.

3.2.1 Many-to-Many Algorithm

The first step of our bidirectional many-to-many algorithm is to run all the backward searches successively. The problem about this approach is that it is not clear how far the backward searches should be extended, since there is no parallel forward search to apply a stopping criterion. What we do is to estimate a search radius $r(t)$ for every backward search from $t \in T$ a priori. We defer the explanation of this procedure to the following Section 3.2.2 and assume that this step will yield appropriate backward distances for now. These backward radii are used to stop the backward searches: A backward search from t is stopped, if the minimum key μ of the priority queue exceeds the given backward radius $r(t)$.

The second step is the expansion of the forward searches. We also expand them one after another and apply the abort criterion for bidirectional search (Lemma 3.6) to stop them: A forward search from a node $s \in V$ can be stopped if $\mu + r(t) \geq D(s,t)$ for all $t \in T$, again μ denoting the current minimum key of the priority queue. The implementation of this test requires a loop with $|T|$ iterations that is too costly to perform it every time when a node was settled. So, we perform this as a batch job and check the criterion just periodically after a fixed number of settled nodes, in our experiments we used 1 000.

Together with the bidirectional stopping criterion this method guarantees that every node lying on any of the $|S \times T|$ shortest paths has been settled at least by one of both search directions.

Bucket Data Structure

Of course, to determine entries of the distance table just expanding the forward and backward searches is not sufficient. To extract the inherent shortest path information one has to consider search space intersections, i.e. nodes that are examined from both search directions. For this reason we introduce a container that stores distances to target nodes $t \in T$. We maintain for every node $v \in V$ a *bucket* that stores a set of pairs (d, t). Such a pair contains the length of a path[2] from v to t, where $t \in T$ is a node from the set of targets and $d \in \mathbb{R}$ the weight of this path. We will refer to bucket entries attached to a certain node $v \in V$ as $b(v)$.

There are different requirements for accessing buckets during the forward and the backward search. The latter needs an operation `v.insert(d, t)` that appends a pair (d, t) to the bucket at a node v. In contrast, during forward search, new bucket entries are not stored but they have to be accessed. Hence, we need a method `v.scan()` that is able to iterate over all bucket entries and so allows to access the recorded pairs at a node v. In Section 2.3.1 we give the details of an efficient implementation of this data structure. Note that the method of storing distances in this bucket data structure will be also used in Chapter 4 for the very efficient method of many-to-many highway hierarchies as well.

[2] not necessarily shortest

The usage of this bucket data structure in general works as follows: During a backward search from a target $t \in T$ we insert at some nodes $v \in V$ pairs (d, t) into buckets $b(v)$ attached to v. Then, during a forward search from a node $s \in S$ the bucket of every node that becomes settled is scanned. The term *bucket scan* of a node v denotes the following procedure: For all pairs $(d, t) \in b(v)$ we try to improve the current entry of $D(s, t)$ of the tentative distance table. That means we set $D(s, t)$ to $d(s, v) + d$ if $D(s, t) > d(s, v) + d$. Then the node v lies on the current tentative shortest path from s to t.

Search Space Intersection

We insert bucket entries only at *some* nodes during the backward searches. This should be as less nodes as possible to reduce the time spent for all kinds of operations that involve the bucket data structure. To remain correctness we must ensure for all $(s, t) \in S \times T$ that at least one of the nodes on the shortest path from s to t that are found by both search directions gets a bucket entry during backward search. This property for sure is fulfilled in a naive implementation that stores for every settled node v of a backward search originating from $t \in T$ the pair $(d(v, t), t)$. Obviously, this strategy will not lead to an efficient algorithm due to overwhelming storage and time consumption.

A first observation is the following: If $d(s, t) < r(t)$, the shortest path from s to t will be found solely by the backward search and we can immediately store the exact length of this shortest path in the tentative distance matrix. To find shortest paths from all the remaining sources we create bucket entries for all nodes that are contained in the priority queue when a backward search stops. What remains to be shown is that shortest paths are actually found by forward searches from nodes v with $d(s, t) > r(t)$.

Lemma 3.7. *The bidirectional many-to-many algorithm remains correct, if bucket entries are stored only at border nodes of the backward search space.*

Proof. During the execution of Dijkstra's Algorithm every node takes one of the three states `unreached`, `visited` or `settled`. Now consider the shortest path $(s = v_0, v_1, \ldots, v_k = t)$ from a source node s to a target node t that was found during a run of Dijkstra's Algorithm in the reverse graph, starting the search from $t \in T$ and continue until $s \in S$ was settled. We assume that there is a node v_r such that $d(v_r, t) < r(t) \leq d(v_{r-1}, t)$, otherwise s would be found directly by the backward search.

Dijkstra's Algorithm starts with setting the node t to `settled` and relaxing its outgoing edges whereby the penultimate node on the shortest path, v_{k-1}, is set to `visited`. Continuing the relaxation procedure in this way the nodes v_i $(r \leq i \leq k)$ are settled successively in the order $v_k, v_{k-1}, \ldots, v_r$ because every predecessor of a settled node has to be settled himself. We stop the algorithm after the minimum key of the priority queue exceeds the radius $r(t)$. Hence, v_r is `settled` and v_{r-1} is `visited`.

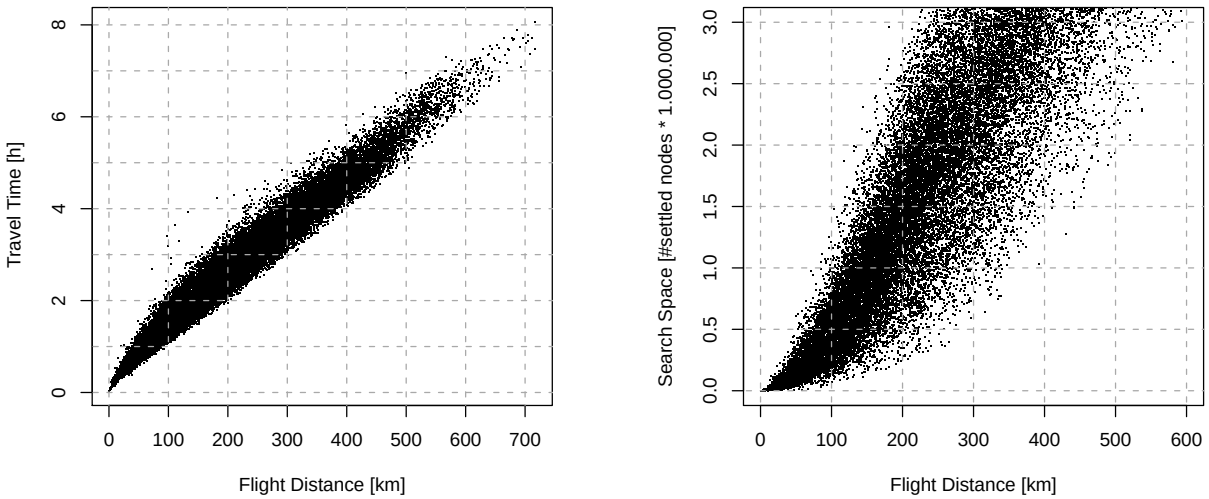
For a node, being `visited` is equivalent to be contained in the priority queue. So, after stopping the backward search from t, $(d(v_r, t) + l(v_{r-1}, v_r), t)$ will be inserted into the bucket $b(v_{r-1})$. The forward search is continued until the bidirectional stopping criterion

holds, so at least all nodes v with $d(s, v) < d(s, t) - r(t)$ will be settled. Hence, v_{r-1} is scanned and therefore the shortest path from s to t running across v_{r-1} will be found. □

3.2.2 Specifying Backward Radii A Priori

As mentioned above, the proposed algorithm requires the radii for the backward searches to be known a priori. So, in this section we define a method that determines those radii. For the considered street networks a geographical layout of the graph is given. We use this layout to estimate graph distances. A nice property of our algorithm is that it can smoothly balance both search directions for different shaped asymmetric problem instances.

Our considerations are based on the assumption that a search spreads circularly around a source node and the search space grows roughly proportional to the area of a circle with a source node $s \in S$ lying in the centre and a radius corresponding to the current minimum key of the priority queue. Figure 3.20(b) shows that this quadratic relation is not very exact, but gives a course picture of the search space that we can expect.



(a) Graph distance in relation to flight distance.

(b) Search space in relation to flight distance.

Figure 3.20: 30 000 *random routes in the road network of Germany*

Estimation of Shortest Path Distances

Bidirectional search is based on partitioning the search into two parts that both do half of the work. In our algorithm the balancing between forward and backward searches is done by the choice of the backward radii. The basic idea is to set the backward radii of a target $t \in T$ to half the length of a shortest path from s to t, where s is the source with the greatest distance to t. Of course, the shortest path length is unknown at this time. Hence, we use an approximation of shortest paths distances.

The correlation between the given geographical layout and the length of a shortest path in the road network is used to estimate yet unknown shortest path distances. We ran several shortest path queries for random source-target pairs to get an average ratio between both variables. Results of this experiment can be found in Figure 3.20(a) that shows a linear relation. Note that we use the approximation method described in Section 3.1.4 to determine geometric distances. This ratio can be expected to remain invariant due to typical changes of the graph as traffic jams, new road segments or similar events and hence it can be ascertained without demanding the graph to be static.

Balancing Forward and Backward Search

For point-to-point queries bidirectional search usually balances the search spaces equivalently between both search directions. We would probably do the same for the computation of quadratic matrices, but for asymmetric problems with $|S| \neq |T|$ perhaps another way of balancing is useful.

The next step is to do an analysis of the search space sizes of the bidirectional many-to-many algorithm to find a reasonable balancing. First, we introduce a *balancing factor* $\alpha \in [0, 1]$ that determines the ratio between the forward and backward radii. Then, we assume an average search radius r for all forward and backward searches. Together with our assumption that the search spaces correspond to the area of a circle, we can estimate the overall search space by the following formula:

$$S(\alpha) = |S| \cdot (\alpha \cdot r)^2 + |T| \cdot ((1 - \alpha) \cdot r)^2$$

We want to minimize this function depending on the balancing factor α. The derivative of $S(\alpha)$ is

$$S'(\alpha) = 2r^2((|S| + |T|)\alpha - |T|)$$

and the second derivative is

$$S''(\alpha) = 2r^2(|S| + |T|)$$

and obviously always positive. Hence, $S(\alpha)$ takes its minimum at the root

$$\alpha = \frac{|T|}{|S| + |T|}$$

of $S'(\alpha)$. For quadratic matrices we get $\alpha = \frac{1}{2}$ as to be expected. For asymmetric matrices this choice of α introduces automatically a smooth balancing and in principle it makes the case distinction superfluous that decides whether to search the reverse graph if $|T| < |S|$.

Basic Radius Selection

Now, we sum up the basic radius selection algorithm we have developed so far. For each target $t \in T$ the distance to the farthest source node $s_f = max\{||L(s, t)|| : s \in S\}$ defines the a priori radius $r(t) = (1-\alpha) \cdot s_f \cdot \gamma$, where γ is the average ratio between the geometric distance and shortest path length and α the balance factor between forward and backward search.

Alternative Radius Selection

There are situations where the basic radius selection algorithm obviously chooses some of the backward radii too large. As an example, consider a query with $S = T = \{a, b, c\}$ as depicted in Figure 3.21. There a node c is located somewhere between nodes a and b. Here $\alpha = \frac{1}{2}$, hence the backward radii r_a and r_b of a and b span over half the distance between a and b. In our model of circular search spaces this is an optimal choice. The potential for improvement comes out if we regard the backward radius r_c of the node c. The basic radius selection we use so far sets the backward radius of c to half the distance between c and b. But the backward search spaces of a and b anyway spread very far and $r'_c = ||L(b, c)|| \cdot \gamma - r_b$ would be sufficient for the backward radius of c. Note that the forward search from c automatically gets a similar search radius due to the bidirectional stopping criterion.

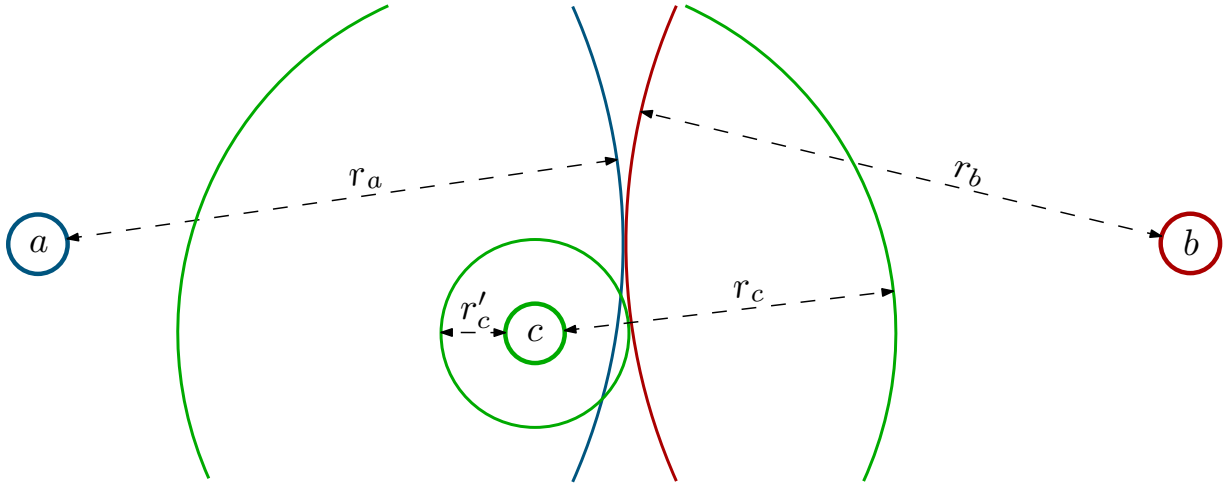


Figure 3.21: *Schematic representation of search spaces in a situation where the basic backward radii determination can be improved.*

This observation leads to an improved algorithm for the determination of the backward radii. The idea of this algorithm is based on an estimation of the forward search radii r_s of source nodes $s \in S$. A forward radius can be estimated as $r_s = \alpha \cdot \gamma \cdot \max\{||L(s, t)|| : t \in T\}$, the distance estimation to its farthest target adjusted by the balancing factor. Then we define the backward radius of a node $t \in T$ as $r_t = \max\{0, \max\{||L(s, t)|| \cdot \gamma - r_s : s \in S\}\}$. This yields backward radii that have at most the size of the radii determined by the basic selection method.

The main effect of this method is the reduction of the radii of central nodes. Conceptually this method would strengthen the search space reduction more than a parallel variant that spreads the searches simultaneously and for that we assume to have no additional communication overhead for the intersection of the search spaces. In our experiments the overhead for this advanced radius determination was imperceptible small in comparison to the overall runtime.

3.2.3 Experimental Results

We made experiments in the road network of Germany (4 378 447 nodes and 10 668 389 directed edges) for our bidirectional many-to-many algorithm. We have two versions of the bidirectional algorithm that are evaluated in our experiments and refer to them as follows:

BiDir Bidirectional search using the basic method to determine backward radii.

BiDir2 Bidirectional search using the alternative method to determine backward radii.

First, we investigate how the backward radius selection performs for asymmetric problem instances. For this, we considered several asymmetric matrices with $|S| \cdot |T| = 32\,400$, source and target locations randomly distributed in 1 Cluster with a size of 1 000 000 nodes. Table 3.3 shows the results of these experiments. We compare the runtime of Dijkstra's algorithm with the runtime of the bidirectional algorithm that uses the alternative backward radius selection. We see that up to $|S| = 100$ the bidirectional algorithm remains faster than Dijkstra's algorithm. Although wee see that the scaling of the balancing factor works, for very asymmetric instances no speedup is achieved. One possible reason is that the average speed of short paths is overestimated by the average ratio between travel time and flight distance. So, the backward searches spread too far. Another point is, that the more asymmetric the instance is, the more overhead of the intersections comes into play and the less is the gain caused by the smaller priority queues.

$\|S\|$	$\|T\|$	Dijkstra	BiDir2
180	180	191.0	89.8
150	216	196.8	118.3
120	270	133.8	103.7
100	324	85.8	85.2
80	405	80.6	98.7
60	540	55.3	68.7
40	810	56.1	66.0
20	1 620	21.4	28.2
10	3 240	7.9	11.1

Table 3.3: *Query time [s] of Dijkstra's algorithm and bidirectional search with alternative radius selection in the road network of Germany for random instances with* $|S| \cdot |T| = 32\,400$ *distributed in* 1 Cluster.

For our experiments with quadratic instances we use sizes between $|S| = |T| = 10$ and $|S| = |T| = 320$ for the distance tables. The random instances are generated as described in Section 2.4.3. 1 Cluster uses the method that generates clustered random instances with 1 cluster of size 1 000 000 nodes. 2 Clusters are generated with a size of 100 000 nodes, 5 Clusters with a size of 20 000 nodes per cluster.

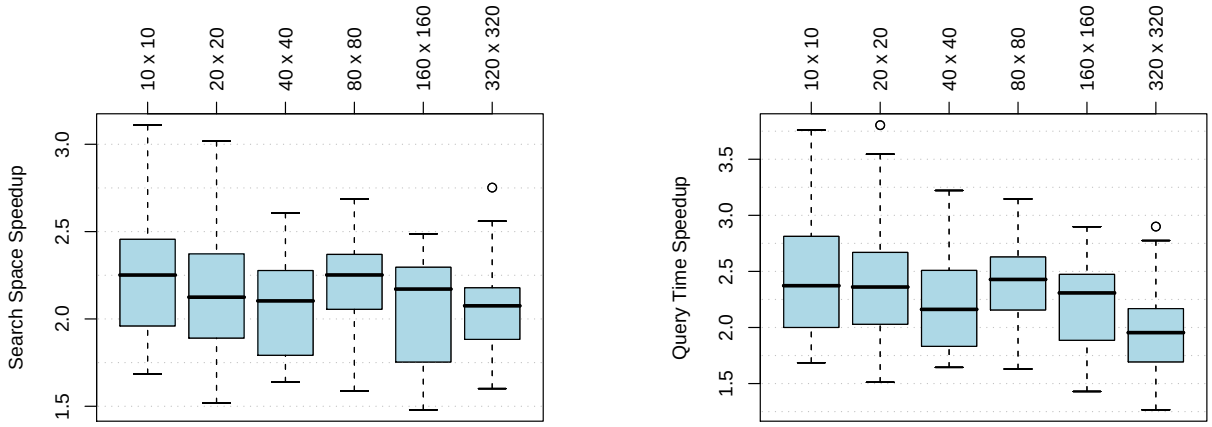


Figure 3.22: *Speedup of bidirectional search with basic radius selection in the road network of Germany for instances between* $|S| = |T| = 10$ *and* $|S| = |T| = 320$ *distributed in* 1 CLUSTER.

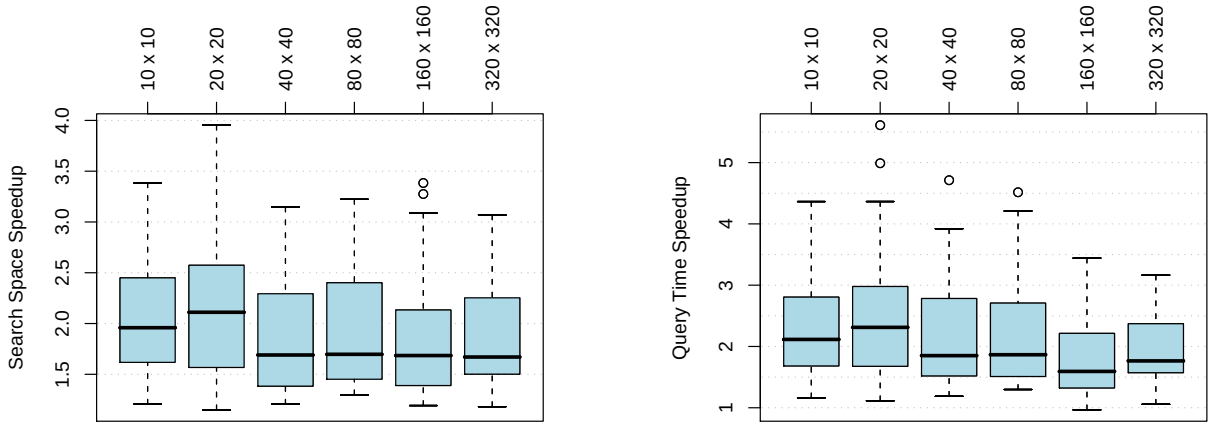


Figure 3.23: *Speedup of bidirectional search with basic radius selection in the road network of Germany for instances between* $|S| = |T| = 10$ *and* $|S| = |T| = 320$ *distributed in* 2 CLUSTERS.

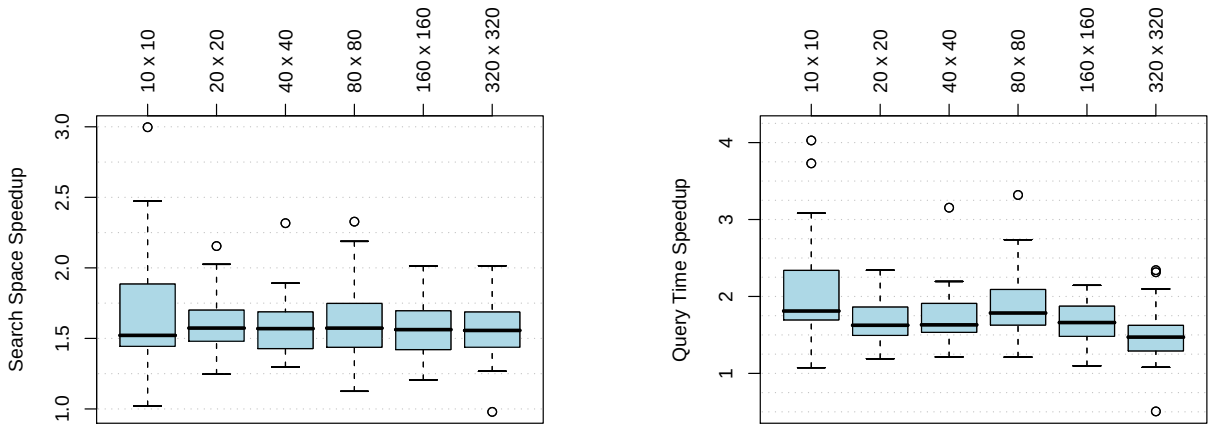


Figure 3.24: *Speedup of bidirectional search with basic radius selection in the road network of Germany for instances between* $|S| = |T| = 10$ *and* $|S| = |T| = 320$ *distributed in* 5 CLUSTERS.

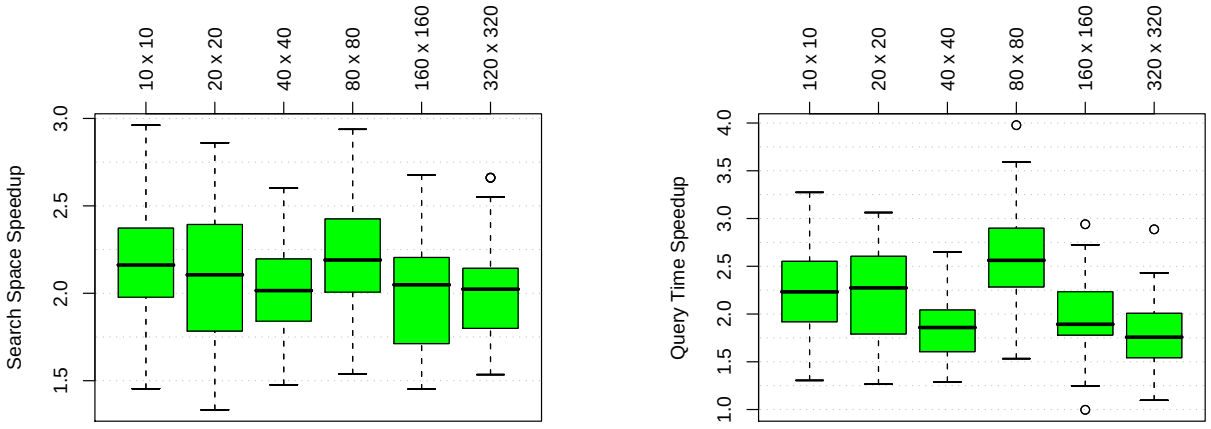


Figure 3.25: *Speedup of bidirectional search with alternative radius selection in the road network of Germany for instances between* $|S| = |T| = 10$ *and* $|S| = |T| = 320$ *distributed in* 1 Cluster.

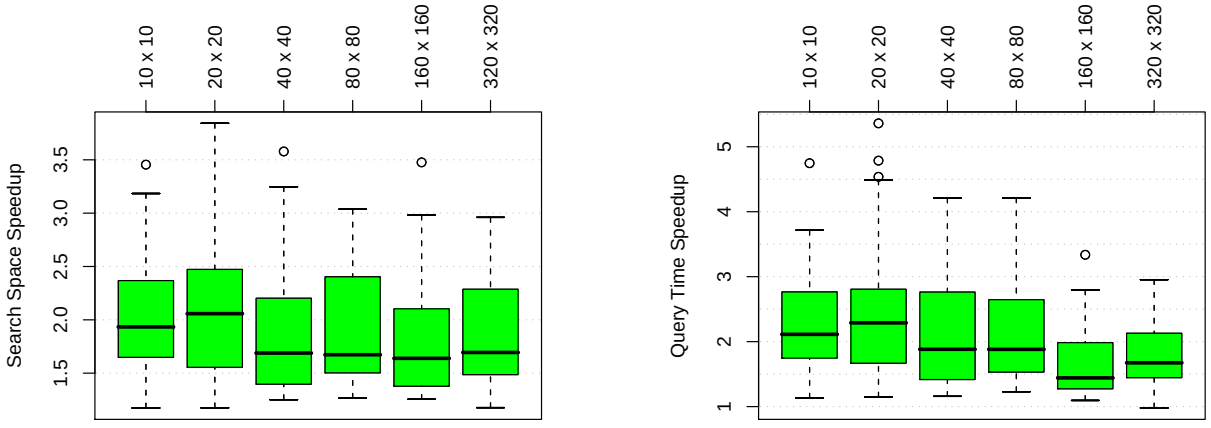


Figure 3.26: *Speedup of bidirectional search with alternative radius selection in the road network of Germany for instances between* $|S| = |T| = 10$ *and* $|S| = |T| = 320$ *distributed in* 2 Clusters.

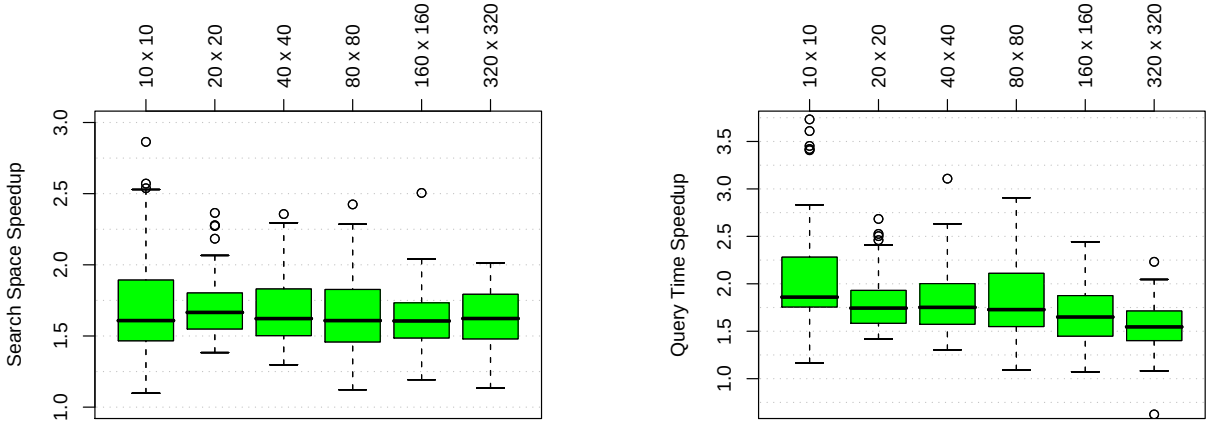


Figure 3.27: *Speedup of bidirectional search with alternative radius selection in the road network of Germany for instances between* $|S| = |T| = 10$ *and* $|S| = |T| = 320$ *distributed in* 5 Clusters.

Experimental results for bidirectional search using requests with different clustering in the road network of Germany are given in Figures 3.22, 3.23 and 3.24 for the basic radius selection and in Figures 3.25, 3.26 and 3.27 for the alternative radius selection. Again, each bar in the box-and-whisker plots represents 50 fixed requests. For all table sizes and clustered random requests we see a speedup of about two. In most cases the basic radius selection works better. Only for instances with five clusters the alternative radius selection yields higher speedups. One can observe that the query time speedup is better than the search space speedup. A reason for this behaviour is the size of the priority queue. If we have search spaces of half the size, then there are less visited nodes maintained in the queue.

3.3 Comparison of Real World Results

Finally, in this chapter about methods without any preprocessing, we apply the presented techniques to the real world instances. As explained in Section 2.4.3, we use nine many-to-many requests having between 173 and 2 892 nodes located in the road network of Europe.

Tables 3.4 and 3.5 show experimental results concerning real world instances and the algorithms without preprocessing. All of the techniques accelerate the real world instances considerably. Most speedups that can be observed are around factor two. Regarding goal-directed search, landmarks work better for larger instances, geometric goal-directed search works very well for all instances. Also, bidirectional search yields speedups of around two for the smaller instances. For the larger instances with our implementation no speedup was measured.

	Dijkstra	GoalSeqF	LM 2	LM 3	LM 4	BiDir	BiDir2
RW1	248	101	117	146	173	121	109
RW2	347	121	132	154	182	155	167
RW3	533	221	225	230	272	270	251
RW4	1363	637	662	651	818	1322	1476
RW5	304	126	148	186	232	279	288
RW6	7408	3906	2341	2468	2488	6670	7682
RW7	8028	5051	3578	3215	3485	9505	13683
RW8	3924	1867	2211	2340	2968	5660	6550
RW9	21882	12817	8022	7395	8526	64922	76818

Table 3.4: *Comparison of many-to-many algorithms without preprocessing. This table gives the query time in seconds for real world instances located in the road network of Europe.*

	GoalSeqF	LM 2	LM 3	LM 4	BiDir	BiDir2
RW1	2.46	2.11	1.69	1.44	2.05	2.28
RW2	2.87	2.63	2.26	1.91	2.24	2.08
RW3	2.41	2.36	2.32	1.96	1.97	2.12
RW4	2.14	2.06	2.09	1.67	1.03	0.92
RW5	2.41	2.06	1.63	1.31	1.09	1.05
RW6	1.90	3.16	3.00	2.98	1.11	0.96
RW7	1.59	2.24	2.50	2.30	0.84	0.59
RW8	2.10	1.77	1.68	1.32	0.69	0.60
RW9	1.71	2.73	2.96	2.57	0.34	0.28

Table 3.5: *Comparison of many-to-many algorithms without preprocessing. This table gives the query time speedup relative to Dijkstra's algorithm for real world instances located in the road network of Europe.*

Chapter 4

Highway Hierarchies

In this chapter we assume the graph to be static. In contrast to Chapter 3 in this situation a preprocessing step to accelerate all further queries is suitable. Regarding the various recent speedup techniques (see Section 1.2 for an overview) we consider highway hierarchies ([25])—one of the currently fastest methods to perform point-to-point queries in road networks—to be a very suitable candidate to be adapted to the many-to-many shortest path problem. The property to use a strong hierarchy without any sense of goal-direction seems to be tailored to accelerate the computation of distance tables.

So, based on highway hierarchies, we present a very efficient algorithm to compute many-to-many shortest paths in a static graph. For example, a huge matrix with an input size of $|S| = |T| = 10\,000$ can be solved in about one minute by our algorithm—Dijkstra's Algorithm would take far more than one day. The one million shortest path distances of a matrix with a size of $1\,000 \times 1\,000$ can be computed in 2.5 seconds—4 680 times faster than Dijkstra's Algorithm.

A straightforward approach to apply highway hierarchies to solve this problem is to use the point-to-point query algorithm, which can answer an s–t query in about one millisecond, for every single entry of the distance table. However, for large matrices this approach is not able to speed up this task considerably. For example a $10\,000 \times 10\,000$ matrix would need about the same time as Dijkstra's algorithm—more than one day. But, as mentioned before, the problem can be solved very efficiently using an other approach that is based on highway hierarchies. We review the basic concepts needed for this technique in Section 4.1 and present our algorithm in Section 4.2. Refinements of the approach are discussed in Section 4.2.1, followed by an analysis in Section 4.3. Finally, experimental results are given in Section 4.4.

4.1 Point-to-Point Algorithm

We start with explaining the concept of highway hierarchies according to [25]. This method was developed for point-to-point queries in large road networks. The basic idea is that outside some local areas around the source and the target node, only a subset of 'important' edges has to be considered in order to be able to find the shortest path. The search can be pruned at roads that are not important for long itineraries. The next section presents

the definition of a highway hierarchy that can be computed in a preprocessing step. For details about the preprocessing, we refer to [25].

4.1.1 Highway Hierarchy

The concept of a *local area* is formalised by the definition of a neighbourhood node set. With given *neighbourhood radii* $r(v) \in \mathbb{R}$ for nodes v in a graph $G = (V, E)$ the *neighbourhood* of a node u is defined as $\mathcal{N}(u) := \{v \in V \ : \ d_u(u, v) \leq r(u)\}$, where $d_u(u, v)$ denotes the distance between u and v in the bidirected graph $G_u := (V, E \cup \{(u, v) : (v, u) \in E\})$. To deal with several paths of the same length the definition of a *canonical shortest path* is needed: For each connected node pair (s, t) we select a canonical shortest path in such a way that each subpath of a canonical shortest path is canonical as well.

We then get a definition of a *highway network* of a graph $G = (V, E)$: An edge $(u, v) \in E$ belongs to the highway network iff there are nodes $s, t \in V$ such that the edge (u, v) appears in the canonical shortest path $\langle s, \ldots, u, v, \ldots, t\rangle$ from s to t in G with the property that $v \notin \mathcal{N}(s)$ and $u \notin \mathcal{N}(t)$.

In such a highway network there are a lot of nodes with a small degree. So, with a contraction procedure the size of a highway network (in terms of number of nodes) can be reduced considerably. For each node v, we check a *bypassability criterion* that decides whether v should be *bypassed*. To bypass a node v means to create shortcut edges (u, w) for edges $(u, v), (v, w)$. The bypassability criterion is fulfilled for a node, if the number of shortcuts that have to be created is smaller than $c \cdot (\deg_{\text{in}}(v) + \deg_{\text{out}}(v))$ for a tuning parameter c. The graph that is induced by the nodes that are not bypassable and enriched by the shortcut edges forms the *core* of a highway network.

A *highway hierarchy* of a graph G consists of several levels $G_0, G_1, G_2, \ldots, G_L$ and is defined inductively. Level 0 corresponds to the original graph: $G'_0 := G_0 := G$. For $i > 0$, $G'_i = (V'_i, E'_i)$ is defined as the core of $G_i = (V_i, E_i)$. G_i itself is the highway network of the core G'_{i-1} of Level $i - 1$.

4.1.2 Query

The highway hierarchies query algorithm [25] performs a modified bidirectional search. We only describe the forward search, since the backward search works analogously. The search is controlled by the current level ℓ of the search and the gap to the border of the current neighbourhood. For every node in the priority queue both values are maintained. Initially, a search starts at a node $v \in V$ in level 0 and with the radius $r(v)$ of v as gap. When an edge (u, v) is relaxed and improves the current tentative path to v, v inherits the level from u and the gap of u decreases by the weight of the edge (u, v). If the gap would become negative, we leave the neighbourhood and the search can ascend to the next level—ℓ increases by one and the gap is set to the radius of v.

A node s_ℓ is called an *entrance point* into level ℓ if it marks the point where the search switches to level ℓ. The next point s'_ℓ on a shortest path from s over s_ℓ that is in the *core* G'_ℓ of level ℓ is called an entrance point into the core of level ℓ. If s_ℓ itself belongs to the core of level ℓ, we have $s_\ell = s'_\ell$.

The speedup of highway hierarchies is achieved by pruning the search space due to two restrictions: Do not relax edges which are not in the current level of the highway hierarchy. Do not relax edges going from nodes in the core of level ℓ to bypassed nodes. Note that the usual stopping criterion for bidirectional search does not work with these pruning rules. The search has to be continued until the search radius of both directions reaches the length of the shortest path found so far. But anyway, in our adaption of the algorithm to the many-to-many shortest path problem such a criterion is not used so far and the search is continued until the priority queue is empty.

4.2 Many-to-Many Algorithm

In this section, we describe how the highway hierarchies query algorithm can be adapted to solve the problem of finding many-to-many shortest paths. Even in very large graphs with millions of nodes only very few nodes are visited by the highway hierarchies query algorithm. These very small search space sizes allow us to store a lot of search spaces completely. This is the basic idea for our many-to-many highway hierarchies algorithm: We remember search spaces and intersect them to compute the desired distance matrices quickly.

We start this section with introducing the terms of forward and backward search spaces for highway hierarchies: The *highway hierarchy forward search space* $\mathrm{F}(s) \subseteq V$ for a node $s \in V$ is the set of nodes that are settled during a query that is performed as described in Section 4.1.2 originating from a source node s in the graph G. Respectively, the *highway hierarchy backward search space* $\mathrm{B}(t) \subseteq V$ for a node $t \in V$ is the set of nodes that are settled during a query originating from a target node t in the reverse graph $\overline{G}$.

If we omit the abort criterion of the highway hierarchies query algorithm for point-to-point queries, we can say that the shortest path distance from a source node s to a target node t is determined by identifying a node $v \in \mathrm{F}(s) \cap \mathrm{B}(t)$ with $d(s,v) + d(v,t) = \min\{d(s,u) + d(u,t) : u \in \mathrm{F}(s) \cap \mathrm{F}(t)\}$. The construction of the highway hierarchy guarantees that a node with this property lies on a shortest path from s to t. In this way one can obtain the distance from s to t for all the pairs $(s,t) \in S \times T$. We will use this formulation of the highway hierarchies algorithm for point-to-point queries to obtain a fast method to obtain entries of the desired distance matrix.

Without loss of generality we will assume $|T| \geq |S|$. Otherwise, it is more efficient to apply the algorithm below to the reverse graph. Now, the idea of our algorithm is the following: We store the backward search spaces $\mathrm{B}(t)$ for all $t \in T$ and intersect them consecutively with the forward search spaces $\mathrm{F}(s)$ for all $s \in S$ to determine the shortest

```
1   forall (s,t) ∈ S×T
2       D(s,t) ← ∞
3   forall t ∈ T
4       BackwardSearch(t)
5           settle(v) : store (d(v,t),t) in b(v) at node v
6   forall s ∈ S
7       ForwardSearch(s)
8           settle(v) : forall (d_t,t) ∈ b(v)
9                           if (d(s,v) + d_t < D(s,t)) then
10                              D(s,t) ← d(s,v) + d_t
```

Figure 4.1: *Highway hierarchies algorithm for many-to-many shortest paths. During a forward or a backward search additional operations are performed when a node becomes settled. In the code listing above those modifications are denoted below the respective call of a Highway Hierarchy search algorithm.*

path distances. To store the backward search spaces we use the bucket data structure that was introduced in Section 3.2.1 for bidirectional search. The implementation of this data structure is described in Section 2.3.1.

Similarly to the idea of the bidirectional algorithm presented in Section 3.2, we start with performing all backward searches from nodes $t \in T$ to store the backward search spaces: For all nodes $v \in \mathrm{B}(t)$ that are settled during a backward search from t the distance $d(v,t)$ is stored as a pair (d,t) in the bucket $b(v)$ of v. Additionally, a tentative distance matrix whose entries are initialised with infinity is maintained. So, after all backward searches have been finished we can start the forward searches to do the intersections. During a forward search no extra data has to be stored. Again, we use the highway hierarchies query algorithm and modify it by introducing additional operations when a node becomes settled. We use the method of *bucket scanning* that was introduced for bidirectional search: At a node v we update all tentative entries of the distance matrix that can be improved by a path including the current node v. To do this we examine for every $d(v,t) \in b(v)$ if $d(s,v) + d(v,t) < D(s,t)$. An overview of the algorithm is given in Figure 4.1 in pseudo code representation.

4.2.1 Optimisation

Asymmetric Search. The time that is spend during the algorithm is not only dependent on the search space sizes. For very large distance matrices the time for bucket scanning dominates the query time. The first idea to reduce the bucket scanning is to introduce an asymmetric approach. Our first step is to introduce a tuning parameter K by using only levels $0, \ldots, K$ of the highway hierarchy. Since level K can have considerable size, it would be wasteful to search it from both directions. So, we reduce the number of nodes with an attached backward distance by stopping the backward searches earlier: Outgoing edges of

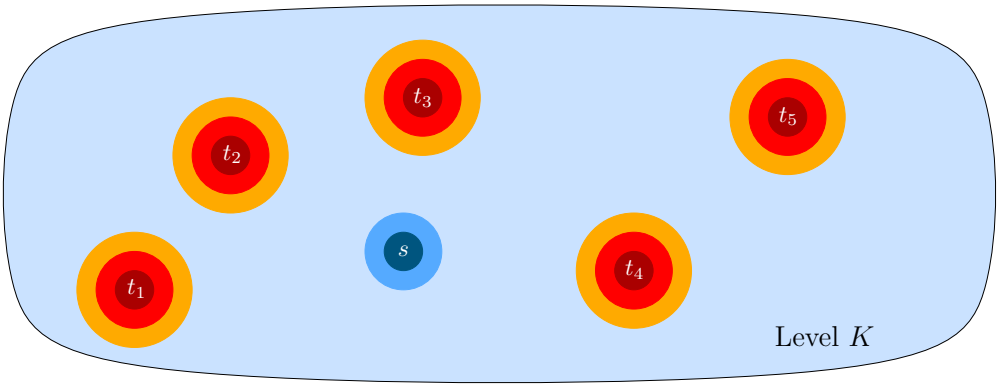


Figure 4.2: *Schematic view of our asymmetric many-to-many highway algorithm. The forward searches explore all of level K whereas the backward searches stop at entrance points to the core of level K.*

an entrance point to the core of level K are not relaxed. Hence, the backward searches never look beyond such core entrance points of level K. The forward search is not aborted and still expands completely in level K.

This approach works similar as an optimised version of the algorithm in [25] where *both* search directions stop searching at level K and the remaining distance is covered by a precomputed distance table between nodes in G'_K. The idea of the asymmetric approach is that the backward searches rely on the forward search to explore enough of level K to meet it. To show that this technique preserves the correctness of the algorithm, we consider a node $v \in \mathrm{F}(s) \cap \mathrm{B}(t)$ lying on a shortest path P from s to t that was found by the basic highway algorithm. If v is found while the backward search is in a level $\ell < K$, obviously it is also found by the asymmetric backward search. If v is found while the backward search is in level K, then there is an entrance point e to the core of the topmost level that lies between v and e on the shortest path P and that is found by the asymmetric backward search. Every forward search finds this node because it explores all nodes $v \in V'_K$, in particular $e \in V'_K$.

This observation leads to the next optimisation: *Fewer Bucket Entries.* From the insights of the previous paragraph we can conclude that some of the bucket entries can be omitted. Our algorithm finds a shortest path P from $s \in S$ to $t \in T$ if there is at least one intermediate node $v \in P$ that is settled during forward search from s *and* backward search from t. Every additional bucket entry at an other node u with this property costs unnecessary extra scanning time. We can save such scans based on the observation that during a highway search the current search level can differ from the actual level of a node. Due to this fact, bucket entries at nodes in the core of level K are made while the search is still in a level $\ell < K$. Because every forward search settles all nodes in G'_K, a bucket entry $(t, d) \in b(v)$ can be omitted if it corresponds to a path of the form $(s, \ldots, v', v)$ where both v' and v are in the core of level K.

Accurate Backward Search. The reduction of bucket scans described in the previous paragraph can be strengthened by performing accurate backward searches. The current version of backward search is not accurate because we break the search at entrance points to the core of the topmost level. To make them exact, backward searches are enlarged: We do not prune the search at core entrance points and continue until all nodes in the priority queue are in the core of level K. This can be implemented by maintaining a state that can be *active* or *passive* for every visited node. Initially all nodes are active. A node inherits the state of its predecessor in the shortest path tree. At entrance points to the core of the topmost level the state is set to *passive*. The search is continued until all nodes are *passive*. This method leads to fewer bucket entries, because the restriction of the previous paragraph applies more often.

4.2.2 Outputting Paths

So far we have only described how to compute distances. We now describe how the algorithm can be modified so that it computes a data structure that allows to output an (s,t)-shortest path P (for $(s,t) \in S \times T$) in time $O(|P|)$. First, note that any path in the highway hierarchy can be efficiently converted to a path in the input: Store the constituent edges (from the same level in the hierarchy) of each shortcut in a separate list. This leads to a linear increase in space consumption and allows efficient recursive conversion of a highway hierarchy edge into a path in the input graph.

We explicitly store the search spaces of forward and backward searches in the highway hierarchy in the form of rooted trees. For each query pair (s,t), the shortest path from s to t consists of a s–v path in the forward search space from s and a v–t path in the backwards search space to t. Hence, all we need to store are pointers to v in the two search spaces. This information is updated during the main computation whenever a better s–t path is encountered.

4.2.3 Computing Shortest Connections Incrementally

In many applications we are not really interested in a complete distance table. For example, many heuristics for the travelling salesman problem start with the closest connections for each node and only compute additional connections on demand [11]. For such applications, the asymmetry in our search algorithm is again helpful. As before, the (small) backward search is done for all $t \in T$ until all entrance points to level K are encountered. The (large) forward searches that require heavy scanning of buckets are only progressing incrementally after their search frontier is completely in the core of level K.

To do this we remember the number of entrance points to level K encountered by each backward search. Each forward search is equipped with a copy of this counter array. When the forward search encounters an entrance point to level K and scans a bucket entry (t,d) it decrements the counter for t. When the counter reaches zero, $D(s,t) = d(s,t)$ and we can output the newly found distance.

4.3 Analysis

Together with the entrance point restriction of the preceding paragraph, the level restriction further reduces the size of the backward search spaces. The forward search spaces and the overall search space grow, but we spend less time per node. So the choice of the level we abort at is a parameter that has to deal with this tradeoff. Dependent on the problem size different abort levels are useful. The bigger the matrix is, the smaller we can choose the maximum level, because for larger matrices we have more backward distances to maintain and to compare with. We also save storage space for the backward distances in main memory which could be a limiting issue for very large matrices.

Since highway hierarchies do not give worst case performance guarantees that hold for arbitrary graphs our analysis will be based on parameterisation and assumptions that still have to be checked experimentally for a given instance. We nevertheless believe such an analysis to be valuable because it explains the behaviour of the algorithm and helps choosing the tuning parameters. For the following analysis, we consider the variant of the algorithm that uses the basic version from Section 4.2 together with the *asymmetric search* optimisation. Let $\mathcal{H}(\ell)$ denote the average number of nodes that are settled in level ℓ during one highway search that is aborted at the entrance points to the core of level K. Let $\mathrm{Dij}(k)$ denote the cost of a Dijkstra-search when exploring k nodes in a road network.

The backward searches have a cost of about $|T| \cdot \mathrm{Dij}(\sum_{\ell=0}^{K} \mathcal{H}(\ell))$. This cost includes the time for storing the distances to targets in the buckets. The forward searches have cost of about $|S| \cdot \mathrm{Dij}(|V'_K| + \sum_{\ell=0}^{K-1} \mathcal{H}(\ell))$ for the search itself where V'_K is the core of level K. To estimate the cost of scanning buckets during forward search, we first state that in level ℓ there are about $\mathcal{H}(\ell) \cdot |T|$ bucket entries made during backward search. We have an average number of bucket entries for each node in level ℓ of $\frac{\mathcal{H}(\ell) \cdot |T|}{|V'_\ell|}$. Hence, based on the assumption that the target nodes are uniformly distributed, the forward searches scan about $|S| \cdot \mathcal{H}(\ell) \cdot \frac{\mathcal{H}(\ell) \cdot |T|}{|V'_\ell|}$ bucket entries in level $\ell < K$ and $|S| \cdot |V'_K| \cdot \frac{\mathcal{H}(K) \cdot |T|}{|V'_K|}$ in level K. So, the overall number of bucket scans is

$$|S| \cdot |T| \cdot \left(\mathcal{H}(K) + \sum_{\ell=0}^{K-1} \frac{\mathcal{H}(\ell)^2}{|V'_\ell|} \right).$$

With this assumptions we get a total cost for the highway hierarchies many-to-many algorithm of

$$|T| \cdot \mathrm{Dij}\left(\sum_{\ell=0}^{K} \mathcal{H}(\ell) \right) + |S| \cdot \mathrm{Dij}\left(|V'_K| + \sum_{\ell=0}^{K-1} \mathcal{H}(\ell) \right) + |S| \cdot |T| \cdot c \cdot \left(\mathcal{H}(K) + \sum_{\ell=0}^{K-1} \frac{\mathcal{H}(\ell)^2}{|V'_\ell|} \right).$$

If both S and T are large, the dominating term is $|S| \cdot |T| \cdot c \cdot \left(\mathcal{H}(K) + \sum_{\ell=0}^{K-1} \frac{\mathcal{H}(\ell)^2}{|V'_\ell|} \right)$. From this we can learn several things. First, the constant behind this term is very small so

that we can expect very good performance for large problems. Second, it is obvious that we can actually reduce the time for bucket scanning by choosing K smaller than the maximum possible level. The experimental section will show that bucket scans only dominate cost for rather large inputs. Based on this estimate and appropriately measured constants of proportionality we would then get a cost model that is accurate enough to choose a (near) optimal value for K.

It is also interesting to look at extreme cases. When $|S| = |T| = 1$, it is best to choose K as the highest level and we essentially get the ordinary highway hierarchy query algorithm (except that we do not stop the backward search early when source and target are close together). When $T = V$, it is best to choose $K = 0$ and we get the ordinary Dijkstra algorithm for (repeated) single source shortest path. In other words, our algorithm smoothly interpolates between the best algorithms for these extreme cases and promises considerable speedups in the middle where none of these other algorithms works very well.

4.4 Experimental Results

For our experiments we use the road network of Europe. For choosing S and T we use random instances of two types. Symmetric instances with $S = T$ and asymmetric ones with $|S| \cdot |T| = 3\,240\,000$. We also tested the nine symmetric real world instances between 173 and 2 892 nodes stemming from vehicle routing problems. On the graph of North America we have also tried random symmetric instances. The results are quite analogous to those for Europe and can be found in Appendix A. The general setup of the experiments is explained in Section 2.4.

Figure 4.3 gives running times for different variants of our algorithm using a large symmetric instance with $|S| = |T| = 20\,000$ and Figure 4.4 an asymmetric instance with $|S| = 100$ and $|T| = 32\,400$. We can see that reducing the number of bucket entries without changing the backward search is always helpful. Investing more into backward search pays only for large symmetric instances and is highly counterproductive for asymmetric instances. From now on we stick to the variant *with* reducing the number of bucket entries but *without* the accurate backward search. This seems to be a good compromise that always improves on the basic variant.

Performance for random symmetric instances with $|S| = |T|$ between 100 and 20 000 with maximum level K between 5 and 9 is given in Figure 4.5. We see that $K = 7$ is always a good value. Only for very large inputs, $K = 6$ is somewhat better. The break even point is near $|S| = |T| = 6\,000$. Since the inputs from our applications are usually symmetric and not so big, we have decided not to implement an automatic algorithm for selecting the best value of K. It is interesting to compare this with the running time of alternative algorithms given in Figure 4.6. Over the entire range of input sizes, our algorithm outclasses both Dijkstra's algorithm and a naive highway hierarchy algorithm that performs $|S| \times |T|$ individual queries.

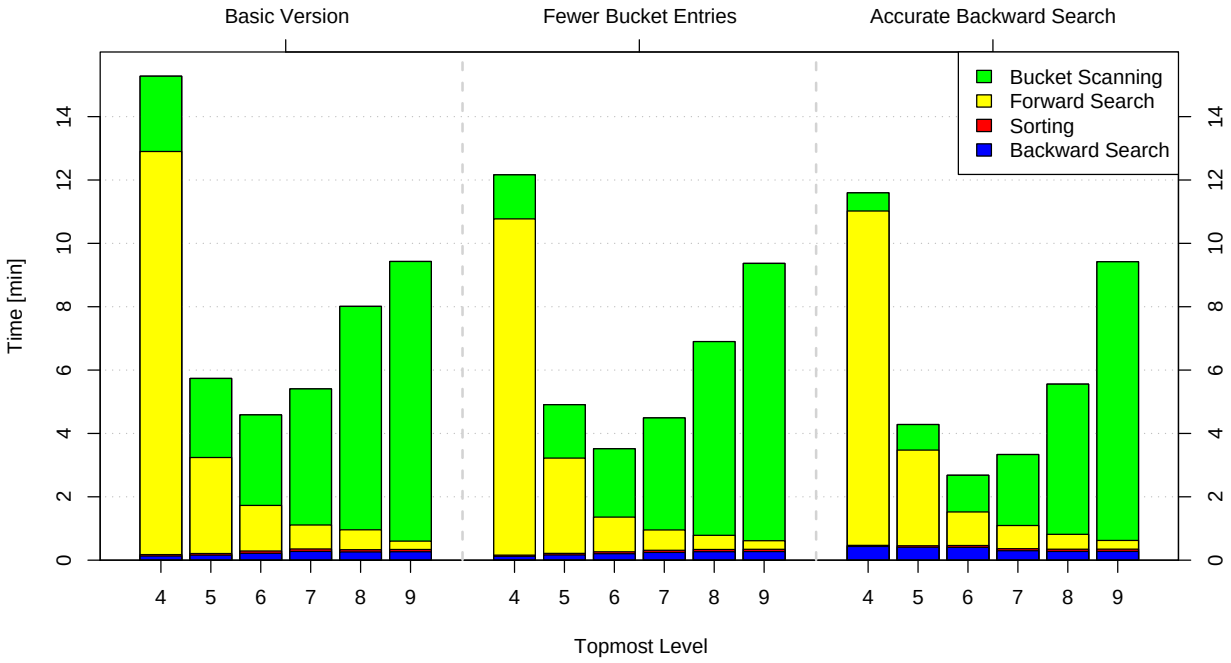


Figure 4.3: *Performance of different algorithm variants for a large symmetric instance with 20 000 source and target nodes.*

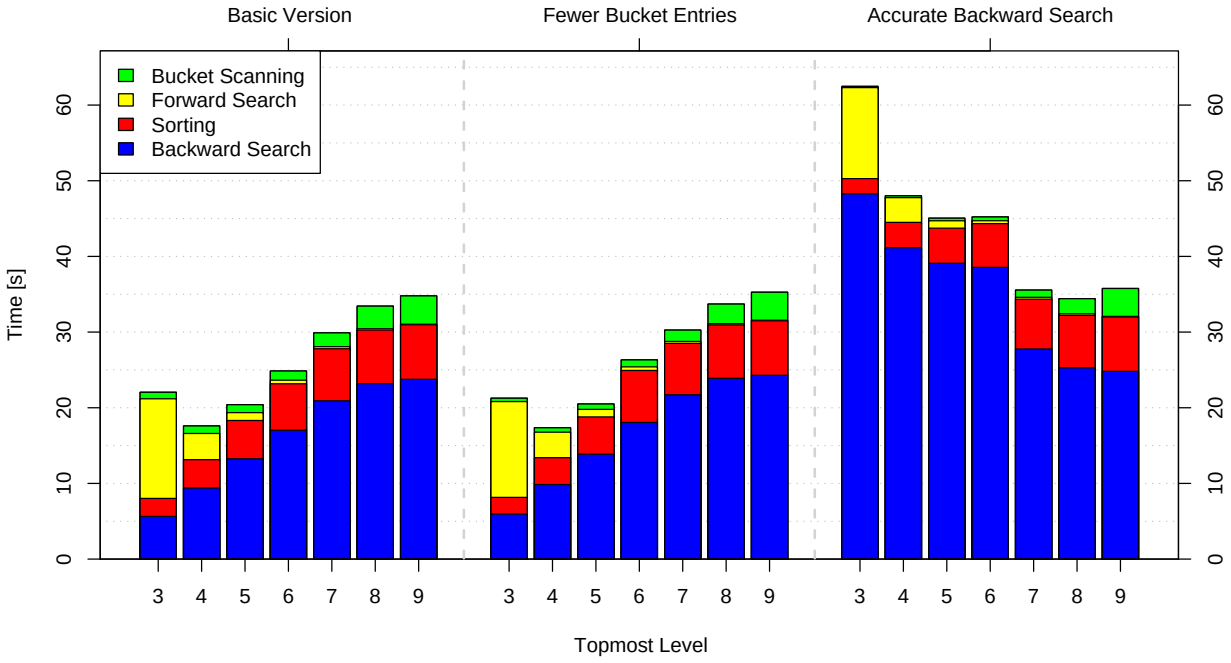


Figure 4.4: *Performance of different algorithm variants for an asymmetric instance with 100 source nodes and 32 400 target nodes.*

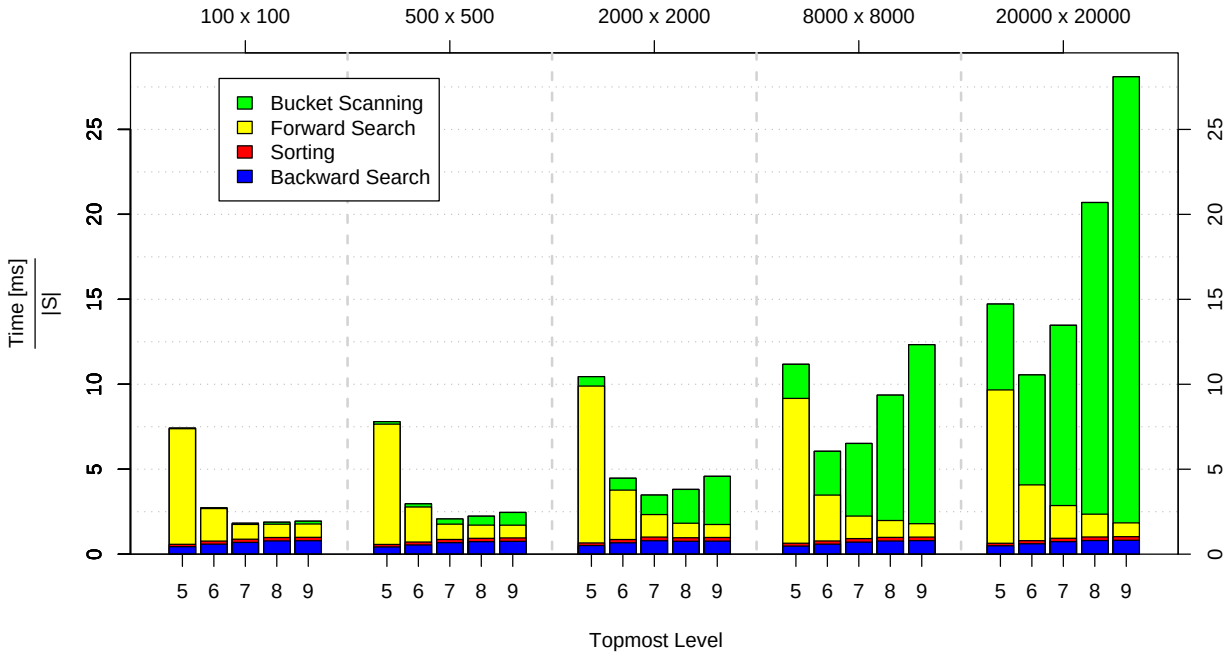


Figure 4.5: *Different choices of maximum level K for quadratic instances.*

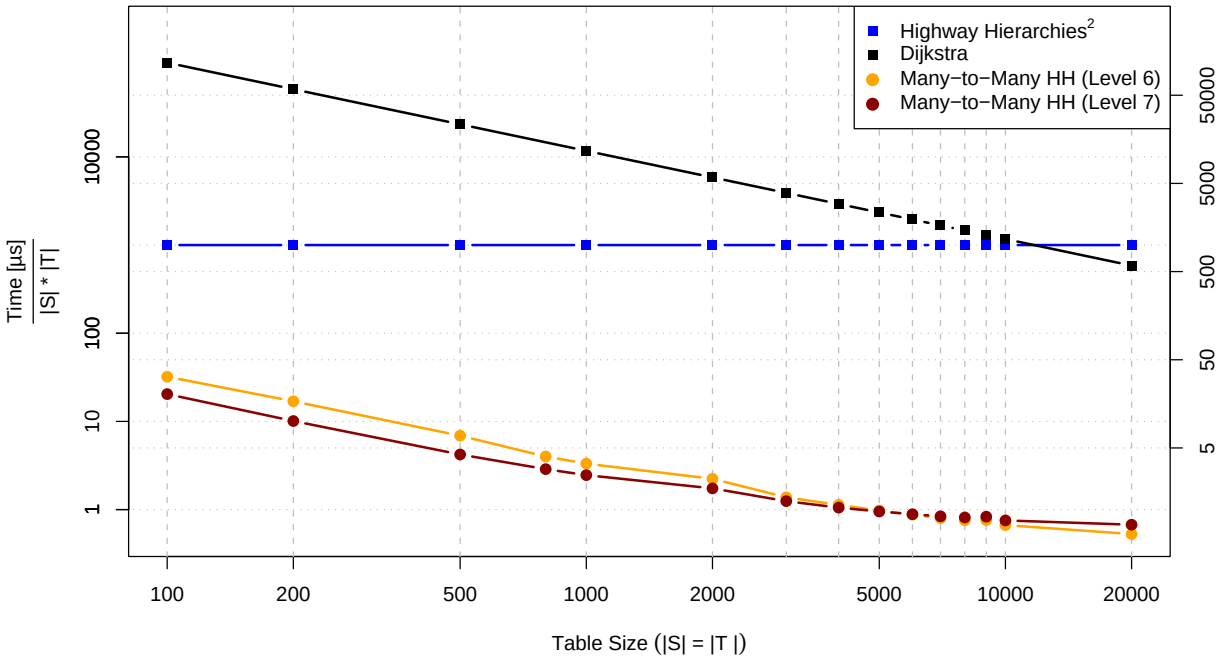


Figure 4.6: *Comparison of our algorithm with alternative algorithms.*

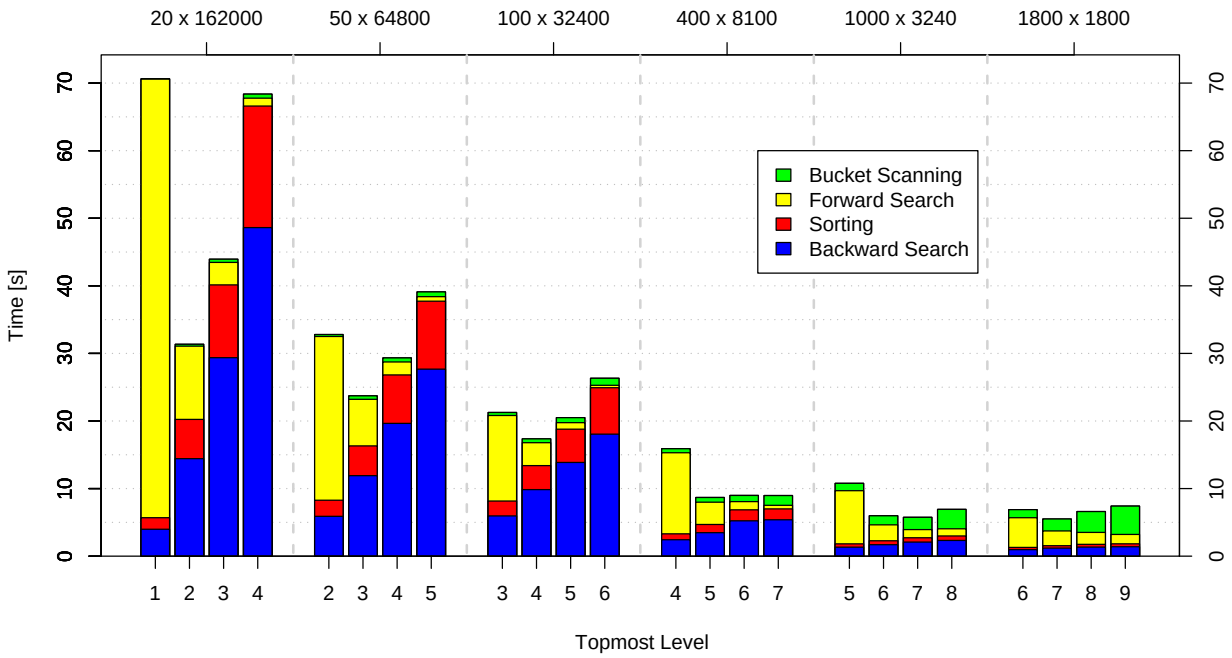


Figure 4.7: *Performance for asymmetric instances at different choices of the maximum level K.*

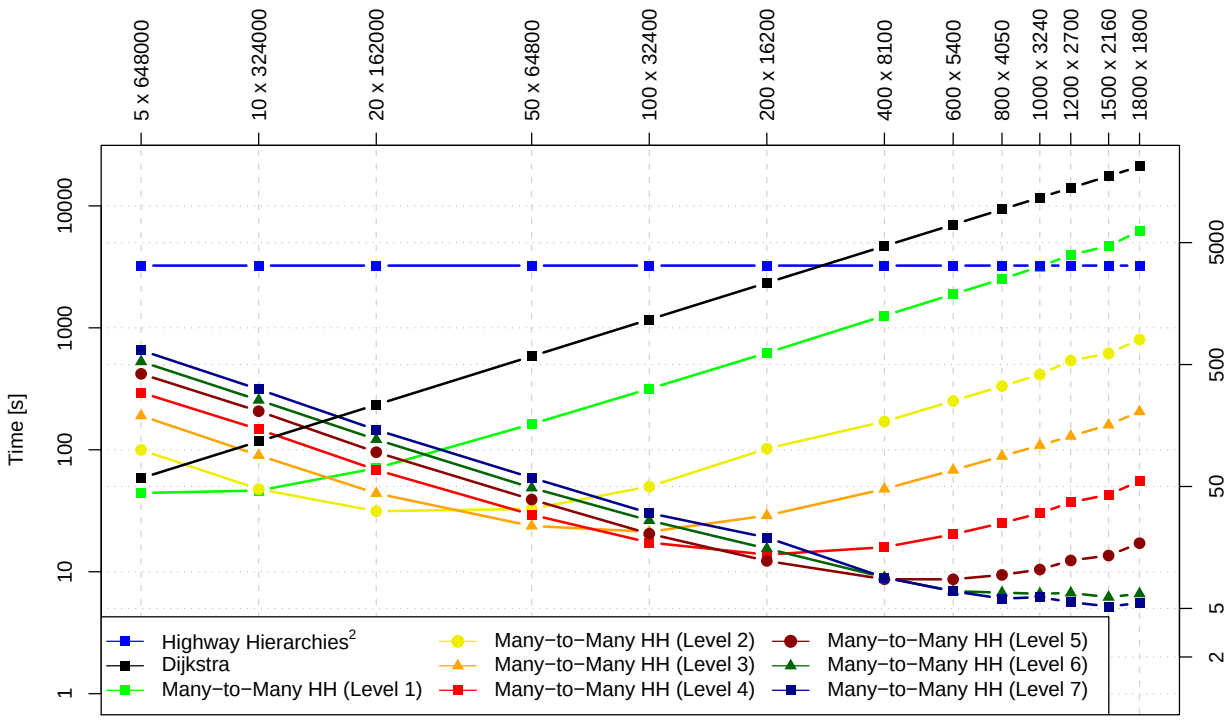


Figure 4.8: *Comparison of algorithms and algorithm variants for asymmetric instances.*

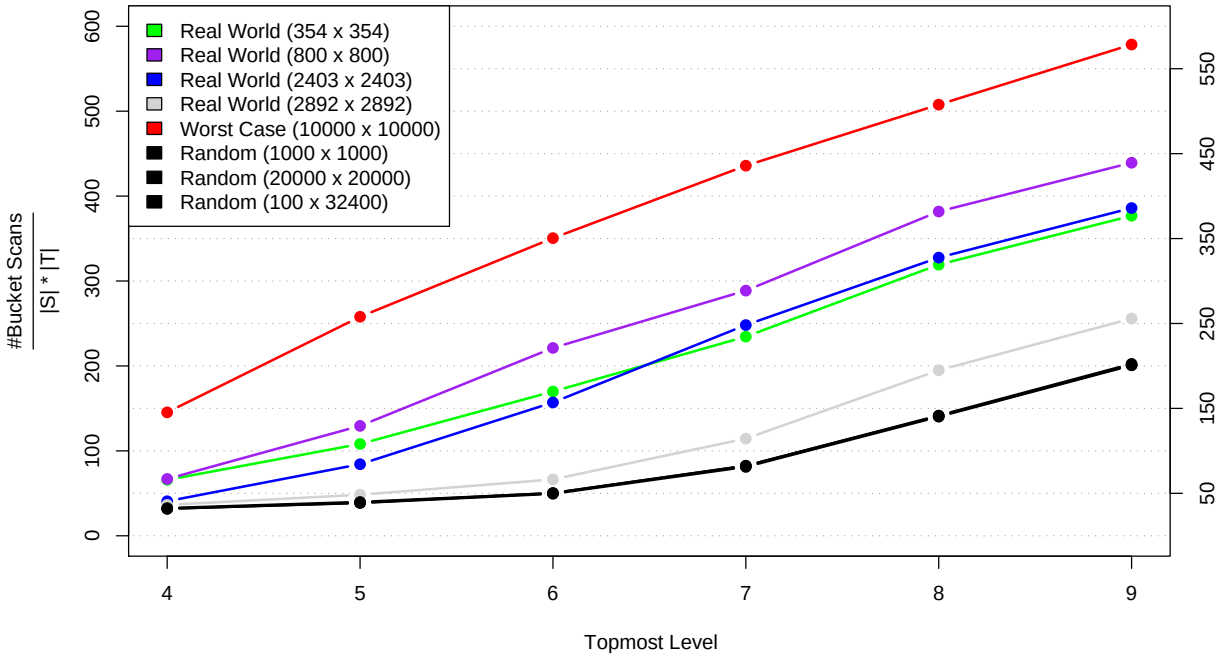


Figure 4.9: *The number of bucket scans per entry of the distance matrix is only dependent on the distributions of the source and target nodes in the graph. Independent of the number of sources and targets it is identic for all random instances in this road network*

Figure 4.7 shows the performance for asymmetric inputs with fixed $|S| \cdot |T| = 3\,240\,000$. As to be expected, forward search and bucket scanning dominates for near symmetric instances while backward search dominates for large $|T|$. With decreasing $|S|$, the optimal level for K goes all the way down from seven to one. But even for $|S| = 5$ our algorithm with level 1 outperforms Dijkstra's algorithm. Figure 4.8 gives the total execution time for an even larger spectrum of size ratios and also includes the competing algorithms.

Figure 4.9 shows that the distribution of the nodes in the graph is crucial for the number of bucket scans. The more the input is clustered, the more intersections occur in the lower levels. A worst case scenario for this is the selection of S and T as the k-nearest nodes of a random node in the graph. Very interesting is the fact, that for a random node selection the number of sources and targets and also the rate of asymmetry is not relevant for the number of bucket scans per table entry. All the tested instances yield invariant results for this piece of data.

Of course, it is interesting whether our measurements with random data have any relation to the performance on real world inputs. Figure 4.10 compares our real world instances with random instances of the same size. An important difference in the inputs is that the real world data is clustered (mostly in some area the size of the Netherlands). Overall, the running times are fairly close together and never more than a factor 1.7 apart. Hence, using random data for measurements is not completely unreasonable. The main difference is that real world instance need considerably more time for bucket scanning.

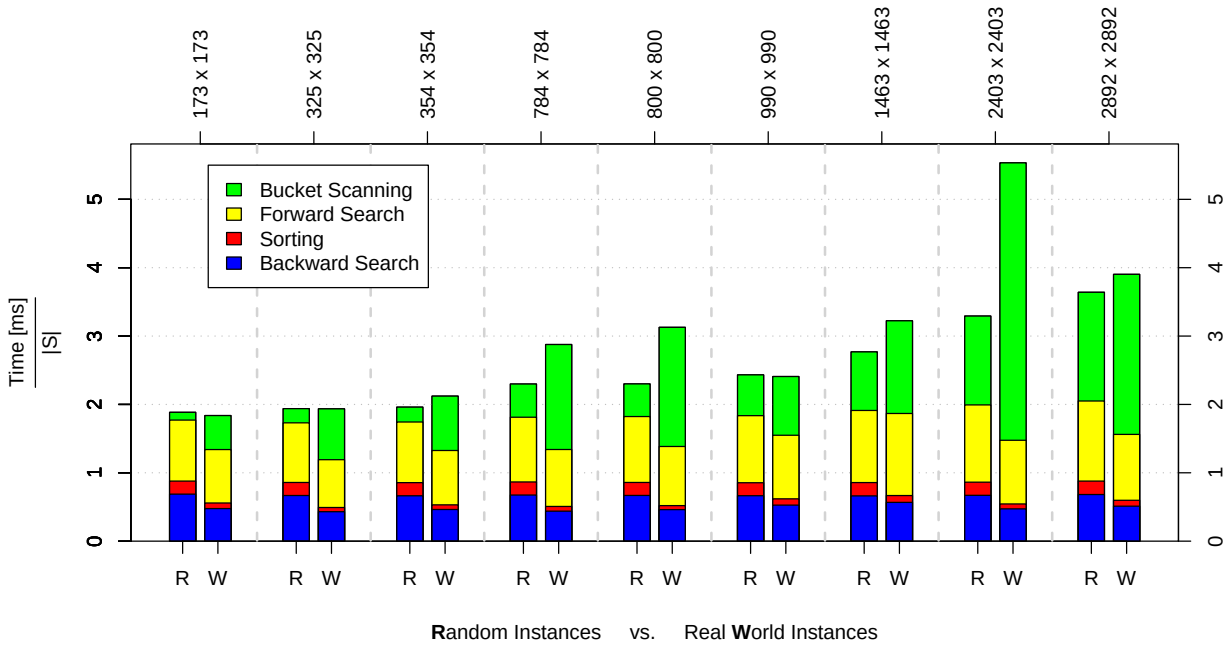


Figure 4.10: *Comparison of random instances (R) and real world instances (W) of the same size.*

This is easily explained by the clustering since search spaces will more often overlap on lower levels of the search. In Figure 4.9 we see the real world instances somewhere between the worst case and uniformly distributed random instances. Real world instances need noticeably less time for backward search and sorting although the search spaces are very similar in size (data not shown here). A possible explanation are cache effects. Subsequent backward searches from clustered nodes are more likely to find the graph data they need in cache.

The time for forward search is about the same for both, real world and random, instance families. The reason is that our current implementation does not break forward search when all entrance points have been covered. Thus, it cannot profit from the clustering of the inputs. it would be interesting to investigate the following modification of our query algorithm that improves the behaviour exactly for such clustered inputs: A forward search from s can stop when its remaining search space is completely in the core of level K and when all nonempty buckets in the core of level K have been scanned. This condition can be checked efficiently and might give significant speedup if sources and targets are concentrated in a small part of the road network.

Chapter 5

Conclusion

We have proposed algorithms for the many-to-many problem in road networks that are able to accelerate shortest path queries considerably. With our *highway hierarchies* many-to-many algorithm we can achieve speedups of more than 1 000. In situations where a *preprocessing* step is *not permissible* bidirectional and goal-directed search can be used and need only half the time of Dijkstra's algorithm.

If the tables are big enough and the locations spread far over a large road graph, our highway hierarchies many-to-many algorithm even beats Dijkstra's algorithm if computing the highway hierarchy is considered to be part of our task. If this is necessary, it might be interesting however to find ways to compute a hierarchy specially tailored to S and T—after all we only need to preserve shortest paths between nodes in S and T. The hope would be that this can be done more efficiently than building a complete highway hierarchy.

In Tables 5.1 and 5.2 we give an overview of the results for real world instances. Additionally, we consider here the highway hierarchies algorithm with a runtime that includes the time for preprocessing (**HHP**). This illustrates that even if we include this extra effort, for large problems we still get high speedups. For our largest real world instance this method is 24 times faster than Dijkstra's Algorithm.

One main observation regarding the Table 5.2 are the fluctuating speedups of the highway hierarchies many-to-many algorithm. The reason is the following: The highway algorithm has very stable runtimes depending almost only on the number of locations. However, in our real world instances the time for Dijkstra's algorithm is dependent on the clustering of the instances. If they are located only in a small area, e.g. only in BeNeLux (RW 8), Dijkstra's algorithm can abort earlier. However, the introduction of an abort criterion for highway hierarchies as sketched at the end of Section 4.4 could weaken this issue.

5.1 Outlook

There are ideas to further improve our acceleration methods for the computation of many-to-many shortest paths. One can image several approaches to refine our techniques without preprocessing: An improved landmark selection method could speed up the method of implicit landmarks. One could try to consider the clustering of the data when selecting the

	Dijkstra	GoalSeqF	LM 3	BiDir	HHP	HH
RW1	248	101	146	121	900	0.35
RW2	347	121	154	155	901	0.64
RW3	533	221	230	270	901	0.78
RW4	1363	637	651	1322	902	2.29
RW5	304	126	186	279	903	2.53
RW6	7408	3906	2468	6670	902	2.37
RW7	8028	5051	3215	9505	905	4.64
RW8	3924	1867	2340	5660	912	12.48
RW9	21882	12817	7395	64922	910	10.44

Table 5.1: *Comparison of many-to-many algorithms. This table gives the query time in seconds for real world instances located in the road network of Europe.*

	GoalSeqF	LM 3	BiDir	HHP	HH
RW1	2.46	1.69	2.05	0.28	709
RW2	2.87	2.26	2.24	0.39	542
RW3	2.41	2.32	1.97	0.59	709
RW4	2.14	2.09	1.03	1.51	595
RW5	2.41	1.63	1.09	0.34	120
RW6	1.90	3.00	1.11	8.21	3125
RW7	1.59	2.50	0.84	8.87	1730
RW8	2.10	1.68	0.69	4.30	314
RW9	1.71	2.96	0.34	24.0	2096

Table 5.2: *Comparison of many-to-many algorithms. This table gives the query time speedup relative to Dijkstra's algorithm for real world instances located in the road network of Europe.*

backward radii during bidirectional search. Another obvious approach is to combine bidirectional and goal-directed search. Preliminary experiments showed that this combination is useful to further reduce the search space.

Regarding the many-to-many highway hierarchies algorithm there is also some room for improvements and generalizations. An improvement for instances with all sources and targets positioned in a local area is an abort criterion that prevents the search to explore the complete graph if not necessary. Incremental computation of the shortest path could be useful in cooperation with travelling salesman algorithms that only need distances between some neighbouring locations. If the performance of a complete matrix computation has to be increased further, [15] sketches a possible parallel implementation of the algorithm.

We see a lot of commonalities between the bidirectional and the highway hierarchies many-to-many algorithm. A compromise between both can be made in a situation where we furthermore do not allow any preprocessing but allow a heuristic for the computation of shortest paths. We accept small errors and are content with almost shortest path distances. This concession is often made in commercial route planning systems. There a heuristic levelling is used that is similar to the highway hierarchies approach—but it is not as fast as the highway algorithm and does not yield exact shortest paths. However, this is a common approach and a handy method that can be applied also in many dynamic situations. A many-to-many algorithm based on such a levelling approach may be considerably faster than our many-to-many methods without preprocessing. The main idea is to use techniques from this thesis that turned out to work well: Perform a bidirectional search and use a bucket data structure to store distances to targets. The backward distances can be stored at nodes where the backward searches ascend to a higher level.

Bibliography

[1] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes. In transit to constant time shortest-path queries in road networks. submitted for publication., 2006.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms. *The MIT Press, Cambridge Massachusetts*, 2001.

[3] Daniel Delling, Martin Holzer, Kirill Müller, Frank Schulz, and Dorothea Wagner. High-Performance Multi-Level Graphs. In *9th DIMACS Implementation Challenge - Shortest Paths*, 2006. To appear.

[4] Daniel Delling, Dominik Schultes, Peter Sanders, and Dorothea Wagner. Highway Hierarchies Star. In *9th DIMACS Implementation Challenge - Shortest Paths*, 2006. To appear.

[5] Bernabé Dorronsoro Díaz. The VRP Web, 2006. `http://neo.lcc.uma.es/radi-aeb/WebVRP/`.

[6] Edsger W. Dijkstra. A note on two problems in connexion with graphs. Numerische Mathematik, 1:269-271, 1959.

[7] A. Goldberg, H. Kaplan, and R. Werneck. Reach for A^*: Efficient point-to-point shortest path algorithms. In *Workshop on Algorithm Engineering & Experiments*, Miami, 2006.

[8] Andrew V. Goldberg. A Simple Shortest Path Algorithm with Linear Average Time. *proceedings of the 9th European Symposium on Algorithms (ESA '01), Springer Lecture Notes in Computer Science LNCS 2161*, pages 230–241, 2001.

[9] Andrew V. Goldberg and Chris Harrelson. Computing point-to-point shortest paths from external memory. *SIAM Workshop on Algorithms Engineering and Experimentation (ALENEX '05), Vancouver, Canada*, 2005.

[10] Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: A^* meets graph theory. In *16th ACM-SIAM Symposium on Discrete Algorithms*, pages 156–165, 2005.

[11] G. Gutin and A. Punnen, editors. *The Traveling Salesman Problem and its Variations*. Kluwer, 2002.

[12] Ron Gutman. Reach-based routing: A new approach to shortest path algortihms optimized for road networks. In Lars Arge, Giuseppe F. Italiano, and Robert Sedgewick, editors, *Proc. Algorithm Engineering and Experiments (ALENEX'04)*, pages 100–111. SIAM, 2004.

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

[14] Martin Holzer, Frank Schulz, and Thomas Willhalm. Combining speed-up techniques for shortest-path computations. In Celso C. Ribeiro and Simone L. Martins, editors, *Experimental and Efficient Algorithms: Third International Workshop, (WEA 2004)*, volume 3059 of *LNCS*, pages 269–284. Springer, 2004.

[15] Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner. Computing many-to-many shortest paths using highway hierarchies. *Submitted to SIAM Workshop on Algorithms Engineering and Experiments (ALENEX '07), New Orleans, Lousiana. To appear.*, 2006.

[16] U. Lauther. An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. In *Münster GI-Days*, 2004.

[17] LEDA, Library of Efficient Data Types and Algorithms, Algorithmic Solutions Software GmbH. `http://www.algorithmic-solutions.com/enleda.htm`.

[18] MapInfo. MapInfo Professional 7.0. `http://www.mapinfo.de/mipro`.

[19] J. Maue, P. Sanders, and D. Matijevic. Goal directed shortest path queries using Precomputed Cluster Distances. In *5th Workshop on Experimental Algorithms (WEA)*, number 4007 in LNCS, pages 316–328. Springer, 2006.

[20] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning graphs to speed up Dijkstra's algorithm. In *4th International Workshop on Efficient and Experimental Algorithms*, 2005.

[21] Stefan Näher and Oliver Zlotkowski. Design and Implementation of Efficient Data Types for Static Graphs. In *10th European Symposium on Algorithms (ESA)*, volume 2461 of *LNCS*, pages 748–759, 2002.

[22] R Development Core Team. R: A Language and Environment for Statistical Computing. `http://www.r-project.org`.

[23] Dominik Schultes. Fast and Exact Shortest Path Queries Using Highway Hierarchies. *Master-Arbeit, Universität des Saarlandes*, 2005.

[24] Dominik Schultes and Peter Sanders. Highway Hierarchies Hasten Exact Shortest Path Queries. *13th European Symposium on Algorithms (ESA)*, 2005.

[25] Dominik Schultes and Peter Sanders. Engineering Highway Hierarchies. *14th European Symposium on Algorithms (ESA)*, 2006.

[26] Dominik Schultes and Peter Sanders. personal communication. 2006.

[27] F. Schulz. *Timetable information and shortest paths.* PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2005.

[28] F. Schulz, D. Wagner, and C. D. Zaroliagis. Using multi-level graphs for timetable information. In *4th Workshop on Algorithm Engineering and Experiments*, volume 2409 of *LNCS*, pages 43–59. Springer, 2002.

[29] Tetsuo Shibuya. Computing the nxm shortest path efficiently. *J. Exp. Algorithmics*, 5:9, 2000.

[30] B. Stroustrup. *The C++ Programming Language.* Addison Wesley, 4th edition, 2000.

[31] T. Ikeda and Min-Yao Hsu and H. Imai and S. Nishimura and H. Shimoura and T. Hashimoto and K. Tenmoku and K. Mitoh. A Fast Algorithm for Finding Better Routes by AI Search Techniques. Vehicle Navigation and Information Systems Conference. IEEE, 1994.

[32] D. Wagner and T. Willhalm. Geometric speed-up techniques for finding shortest paths in large sparse graphs. In *11th European Symposium on Algorithms*, volume 2832 of *LNCS*, pages 776–787. Springer, 2003.

[33] I-Lin Wang. *Shortest Paths and Multicommodity Network Flows.* PhD thesis, Georgia Inst. Tech., 2003.

[34] T. Willhalm. *Engineering Shortest Path and Layout Algorithms for Large Graphs.* PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2005.

List of Figures

List of Tables

Index

Appendix A

Additional Experimental Results

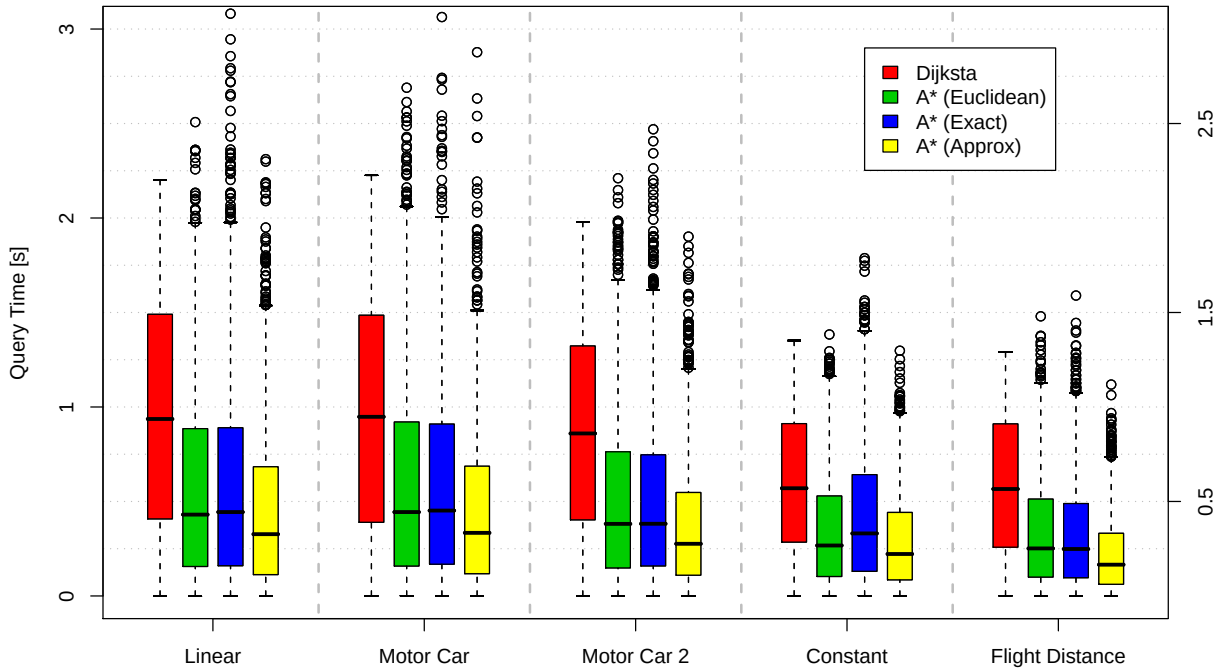


Figure A.1: *Comparison of goaldirected search using different methods to determine geometric distances.*

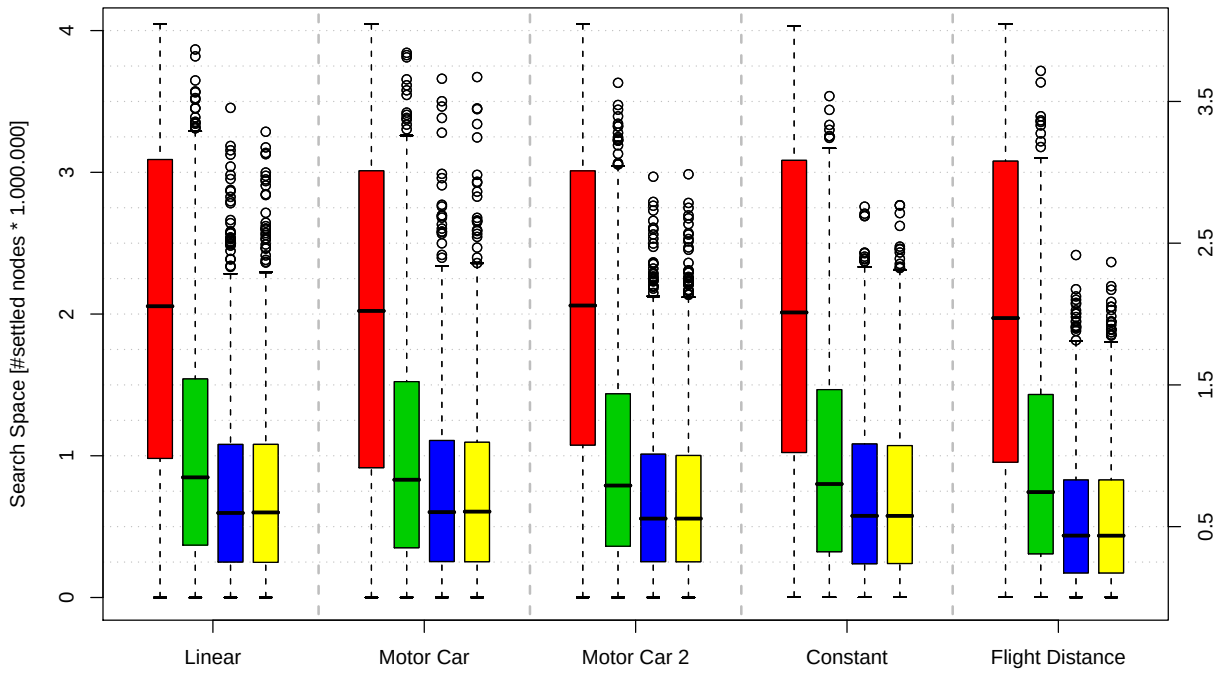


Figure A.2: *Comparison of goaldirected search using different methods to determine geometric distances.*

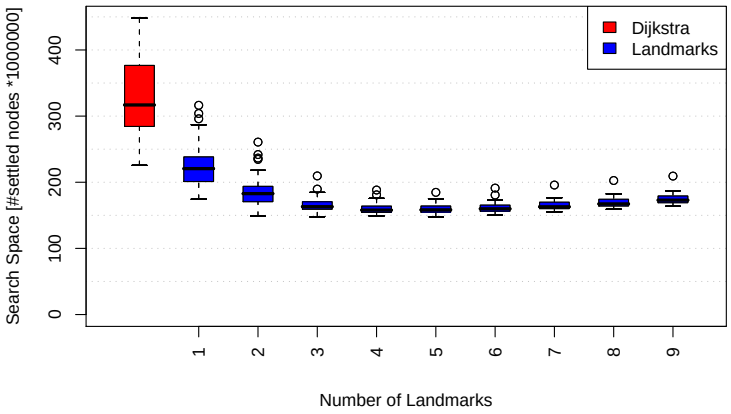


Figure A.3: *Comparison of search space sizes of landmark based goal-directed search with different choices for the number of landmarks in the road network of Germany for instances with* $|S| = |T| = 160$ *distributed in* 1 CLUSTER.

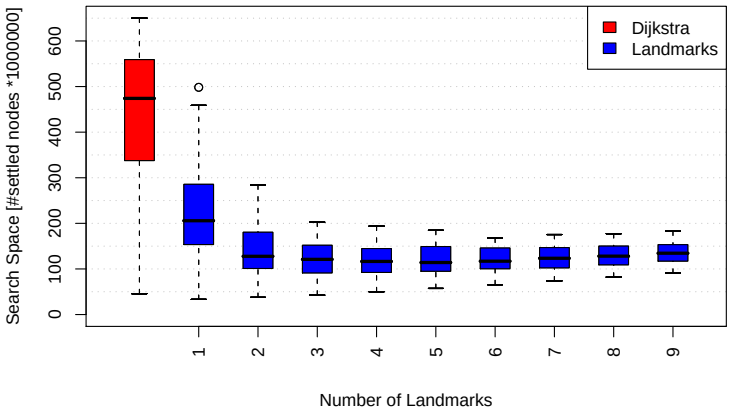


Figure A.4: *Comparison of search space sizes of landmark based goal-directed search with different choices for the number of landmarks in the road network of Germany for instances with* $|S| = |T| = 160$ *distributed in* 2 CLUSTERS.

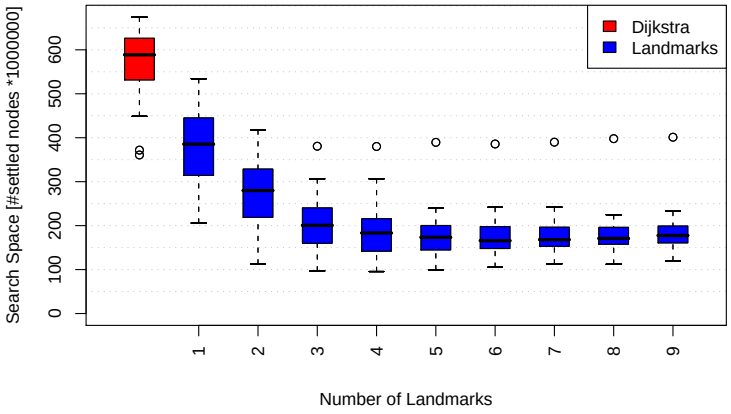


Figure A.5: *Comparison of search space sizes of landmark based goal-directed search with different choices for the number of landmarks in the road network of Germany for instances with* $|S| = |T| = 160$ *distributed in* 5 CLUSTERS.

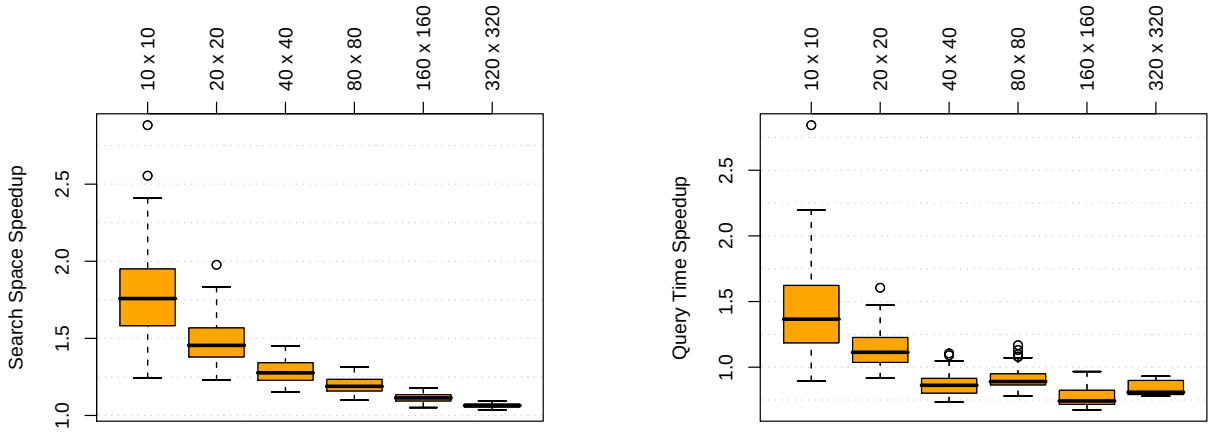


Figure A.6: *Speedup of geometric goal-directed search in the road network of Germany for randomly distributed instances between* $|S| = |T| = 10$ *and* $|S| = |T| = 320$.

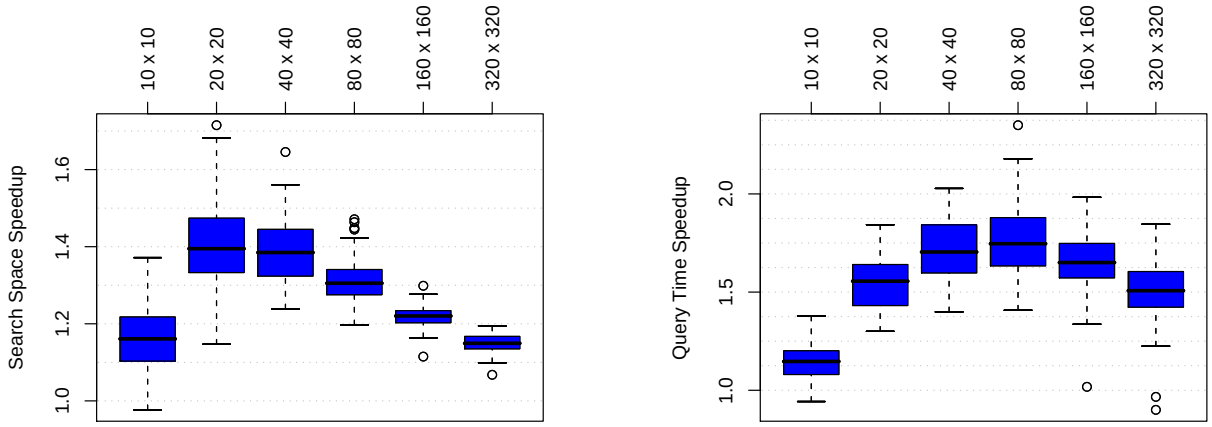


Figure A.7: *Speedup of landmark based goal-directed search using three landmarks in the road network of Germany for randomly distributed instances between* $|S| = |T| = 10$ *and* $|S| = |T| = 320$.

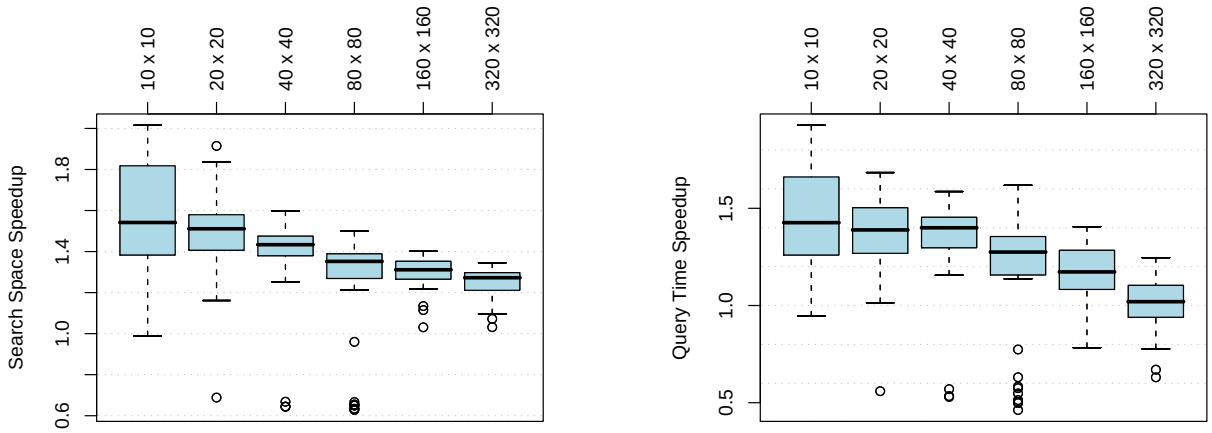


Figure A.8: *Speedup of bidirectional search in the road network of Germany for randomly distributed instances between* $|S| = |T| = 10$ *and* $|S| = |T| = 320$.

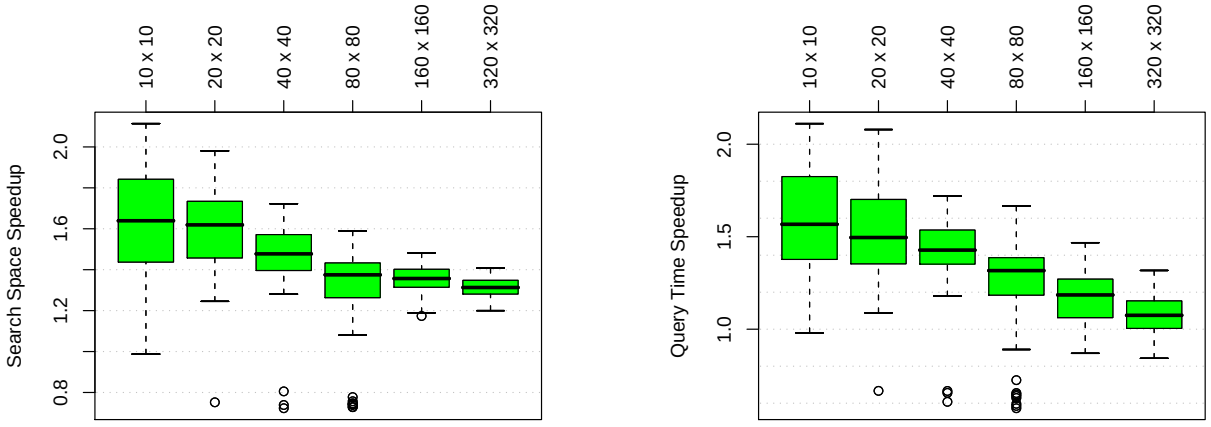


Figure A.9: *Speedup of bidirectional search with the alternative backward radius selection in the road network of Germany for instances between* $|S| = |T| = 10$ *and* $|S| = |T| = 320$ *randomly distributed.*

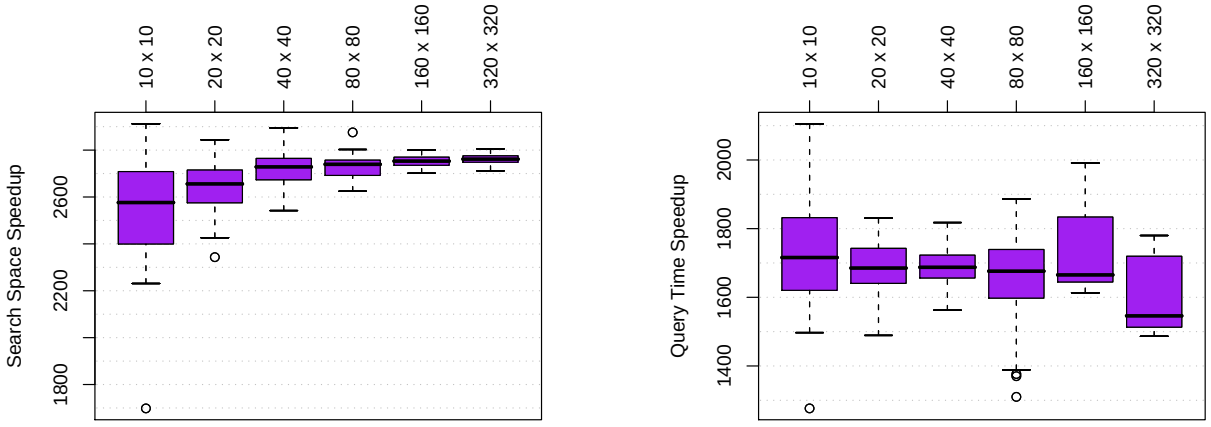


Figure A.10: *Speedup of the highway hierarchies many-to-many algorithm in the road network of Germany for randomly distributed instances between* $|S| = |T| = 10$ *and* $|S| = |T| = 320$.

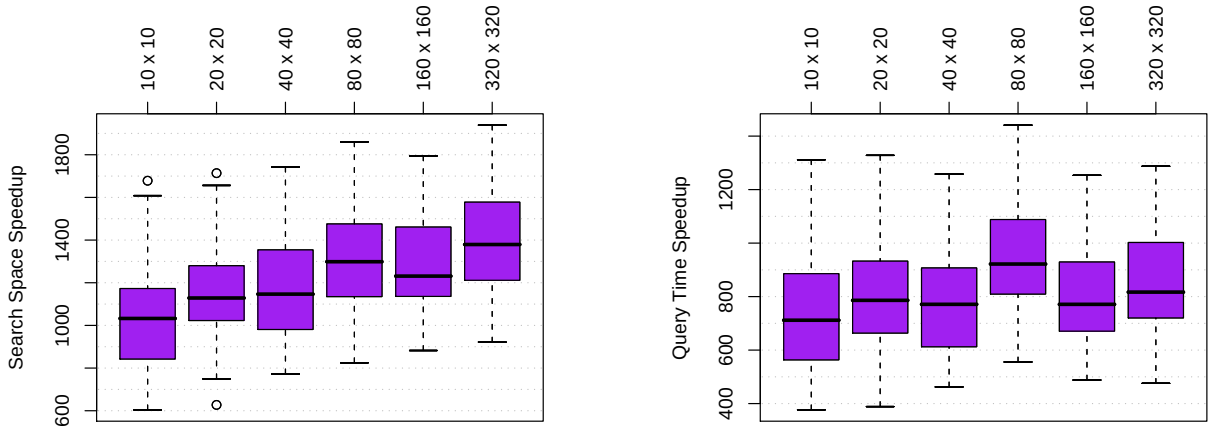


Figure A.11: *Speedup of the highway hierarchies many-to-many algorithm in the road network of Germany for instances between* $|S| = |T| = 10$ *and* $|S| = |T| = 320$ *distributed in* 1 Cluster.

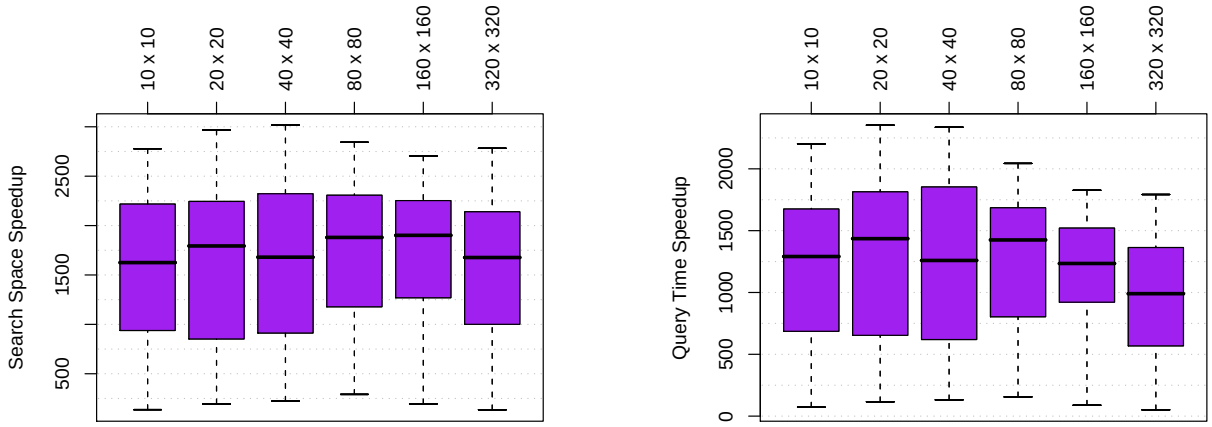


Figure A.12: *Speedup of the highway hierarchies many-to-many algorithm in the road network of Germany for instances between* $|S| = |T| = 10$ *and* $|S| = |T| = 320$ *distributed in* 2 Clusters.

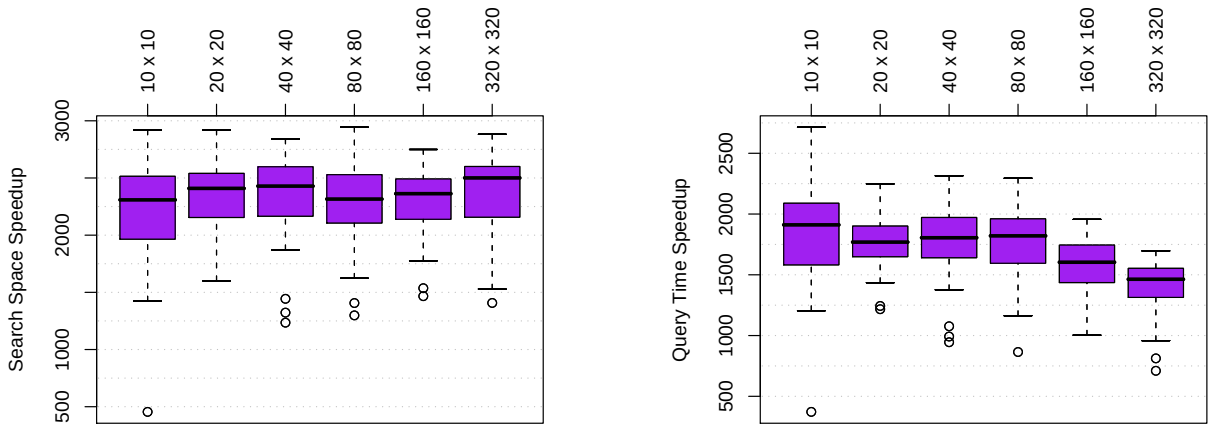


Figure A.13: *Speedup of the highway hierarchies many-to-many algorithm in the road network of Germany for instances between* $|S| = |T| = 10$ *and* $|S| = |T| = 320$ *distributed in* 5 Clusters.

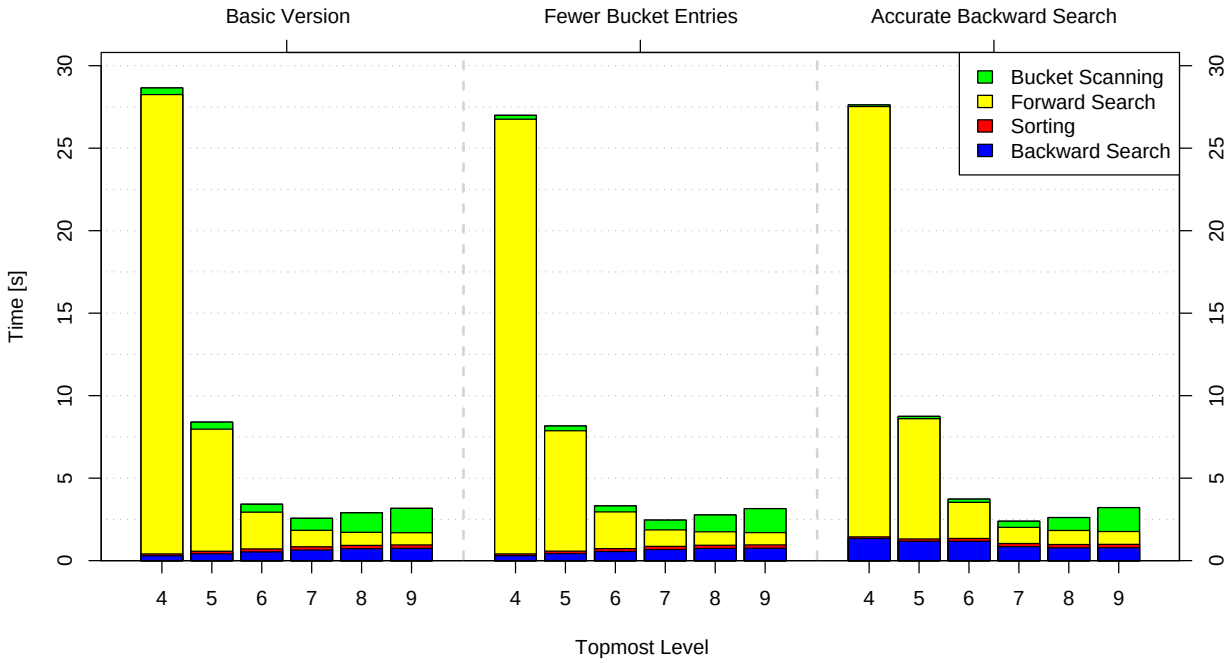


Figure A.14: *Performance of different variants of the many-to-many highway hierarchies algorithm for a large symmetric instance with 1 000 source and target nodes.*

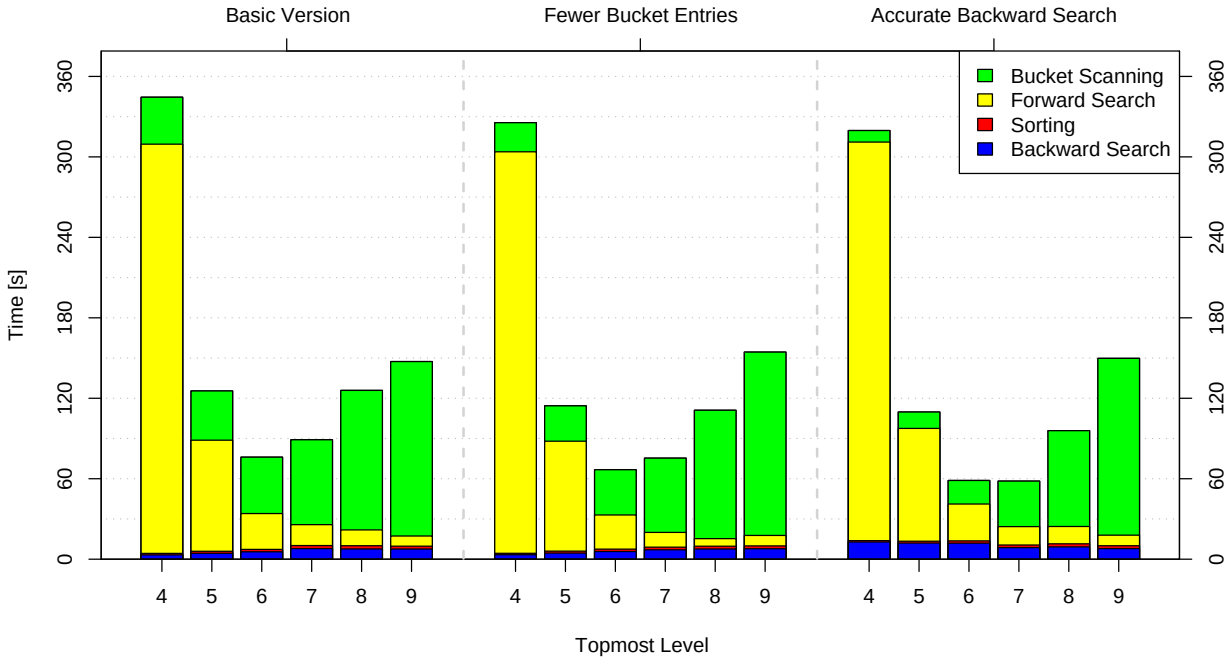


Figure A.15: *Performance of different variants of the many-to-many highway hierarchies algorithm for a large symmetric instance with 10 000 source and target nodes.*

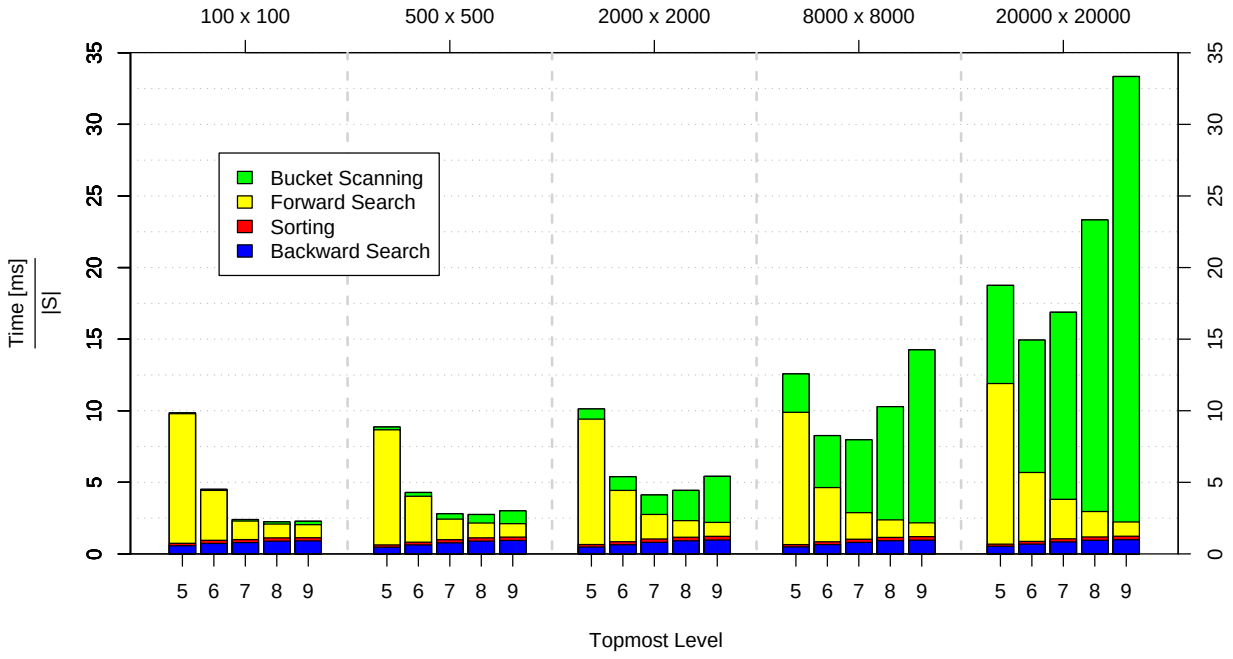


Figure A.16: *Many-to-many highway hierarchies algorithm with different choices of maximum level K for quadratic instances located in the road network of North America.*