Fast and Exact Shortest Path Queries Using Highway Hierarchies

Dominik Schultes

July 2005

Master-Arbeit

Fachrichtung 6.2 – Informatik, Universität des Saarlandes
angefertigt nach einem Thema von Prof. Dr. Kurt Mehlhorn, Max-Planck-Institut für Informatik
unter Betreuung von Prof. Dr. Peter Sanders, Universität Karlsruhe (TH)

In Erinnerung an
meine Oma

Acknowledgements

I would like to thank my supervisor Peter Sanders for the numerous interesting discussions, his encouragement and support. Domagoj Matijevic and Jens Maue proofread a preliminary and the final version of my thesis, respectively. Their suggestions were of great value. Frank Schulz helped with the compilation of the section on related work. Martin Holzer, Domagoj Matijevic, Frank Schulz, and Thomas Willhalm also assisted with data and tools for processing graphs. Last but not least, I would like to thank Kurt Mehlhorn for his willingness to examine my thesis.

This thesis is based on [25], a joint work with Peter Sanders.
For future developments refer to http://www.dominik-schultes.de/hwy/.

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Saarbrücken, im Juli 2005

Abstract

The computation of shortest paths in a graph is a well-known problem in graph theory. One of the most obvious practical applications is route planning in a road network, i.e., finding an optimal route from a start location to a target location. We assume that a given road network does not change very often and that there are many source-target queries on the same network. Therefore, it pays to invest some time for a preprocessing step that accelerates all further queries.

We present a new speedup technique for route planning that exploits the hierarchy inherent in real-world road networks. In a preprocessing step, we investigate the given road network in order to extract and prepare a hierarchical representation. Our route planning algorithm then takes advantage of this data. It is an adaptation of the bidirectional version of DIJKSTRA's algorithm, massively restricting its search space.

In several experiments, we concentrate on the computation of fastest routes in Western Europe and the USA. Both networks consist of about 20 million nodes each. Our algorithm preprocesses these networks in a few hours using linear space. Queries then take around eight milliseconds to produce optimal routes. This is more than 2 000 times faster than using DIJKSTRA's algorithm. There are numerous possibilities to further improve and extend our approach.

Zusammenfassung

Die Berechnung kürzester Pfade in einem Graphen ist ein bekanntes Problem aus der Graphentheorie. Eine der naheliegendsten praktischen Anwendungen ist die Routenplanung in einem Straßennetz, also die Bestimmung einer optimalen Route von einem Start- zu einem Zielort. Wir gehen davon aus, dass ein gegebenes Straßennetz sich nicht sehr oft ändert und dass viele Start-Ziel-Suchen im gleichen Straßennetz durchgeführt werden. Dadurch lohnt es sich, zunächst etwas Zeit in einen Vorverarbeitungsschritt zu investieren, der dann alle nachfolgenden Suchanfragen beschleunigt.

Für das Problem der Routenplanung stellen wir eine neue Beschleunigungstechnik vor, die die hierarchischen Eigenschaften von realen Straßengraphen ausnutzt. In einem Vorverarbeitungsschritt untersuchen wir das gegebene Straßennetz, um eine hierarchische Darstellung zu gewinnen und aufzubereiten. Der Routenplanungsalgorithmus profitiert dann von den gewonnenen Daten. Es handelt sich dabei um eine Anpassung der bidirektionalen Variante des Algorithmus von DIJKSTRA, die den Suchraum deutlich einschränkt.

In mehreren Experimenten beschäftigen wir uns mit der Berechnung von schnellsten Routen in Westeuropa und den USA. Beide Netze bestehen aus jeweils ca. 20 Millionen Knoten. Die Vorverarbeitung dieser Straßennetze dauert wenige Stunden, wobei nur ein linearer zusätzlicher Platzbedarf anfällt. Suchanfragen dauern dann ungefähr acht Millisekunden, um optimale Routen zu bestimmen. Dies ist mehr als 2 000 mal schneller als die Verwendung von DIJKSTRAs Algorithmus. Es gibt zahlreiche Möglichkeiten, diesen Ansatz weiter zu verbessern und auszubauen.

Contents

1 Introduction **1**

2 Preliminaries **8**
2.1 Shortest Paths and DIJKSTRA's Algorithm 8
2.2 Highway Hierarchy 10

3 Construction **14**
3.1 Fast Construction of the Highway Network 14
3.2 Speeding up Construction 20
3.3 Contraction of the Highway Network 21

4 Query **24**
4.1 Multilevel Query Algorithm 24
4.2 Collapse of the Vertical Dimension 29
4.3 Abort-on-Success 30

5 Implementation **33**
5.1 Data Structures 33
5.2 Construction 37
5.3 Query 39

6 Experiments **40**
6.1 Environment and Instances 40
6.2 Parameters 42
6.3 Multilevel Queries 45

7 Discussion **50**

A Canonical Shortest Paths **55**
A.1 Modifications of DIJKSTRA's Algorithm 55
A.2 FIFO Priority Queues 56

B Examples **57**
B.1 Construction 57
B.2 Highway Hierarchy 61
B.3 Query 65

Chapter 1

Introduction

Motivation

Finding an optimal route from A to B is an everyday problem. Since using a map to aid the route planning is rather inconvenient and does not necessary lead to an optimal route, during the last years, many applications and tools were developed that try to determine good routes in order to achieve a reduction of travel distance, time and costs. Two representative examples are route planning services provided in the internet and car navigation systems. There is a great interest in *efficient* route planning methods: in the former case, due to the huge amount of requests that are sent to the server, and in the latter case, due to the limited computing power of car navigation systems. Furthermore, for obvious reasons, there is a great interest in methods that do not only find approximations, but *exact* solutions.

A road network can easily be represented as a *graph*, i.e., as a collection of nodes V (junctions) and edges E (roads) where each edge connects two nodes. Each edge is assigned a weight, e.g. the length of the road or an estimation of the time needed to travel along the road. In graph theory, the computation of *shortest*[1] *paths* between two nodes is a classical problem. From a worst case perspective, the problem has largely been solved by DIJKSTRA in 1959 [9] who gave an algorithm that finds all shortest paths from a starting node s using at most $m + n$ priority queue operations for a graph $G = (V, E)$ with n nodes and m edges. However, these bounds are not satisfying in practice when we deal with very large road networks. There are several aspects that suggest that we can do better:

1. In a sense, DIJKSTRA's algorithm is an overkill since it computes the shortest paths from a given node s to *all* nodes $v \in V$ and not only to *one* given node t. This can be improved by stopping DIJKSTRA's algorithm as soon as the shortest path to t is found, but still the shortest paths from s to all nodes v that are closer to s than t are determined (Fig. 1.1).

2. We assume that a given road network does not change very often and that there are many source-target queries on the same network. Therefore, it can pay to invest some time for a *preprocessing* step that accelerates all further queries.

3. We do not deal with general graphs, but with road networks, which have certain properties. For instance, it is quite unusual for a node in a road network to have degree

[1] Note that, depending on the chosen edge weight, 'shortest' can refer not only to 'spatial distance', but also, for instance, to 'travel time'.

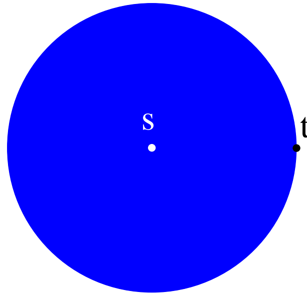


Figure 1.1: Schematic representation of the search space of DIJKSTRA's algorithm.

five or more, i.e., a road network is a very *sparse* graph. Furthermore, road networks are almost *planar* (because there are only a few bridges and tunnels in comparison to the total number of road segments). Usually, a *layout* is given that is based on the geographic coordinates of each node. Moreover, road networks exhibit *hierarchical properties*: for example, there are 'more important' streets (e.g. motorways) and 'less important' ones (e.g. urban streets).

Specification of the Goals

On a given *large* road network, we allow a *fast preprocessing* step in order to make *fast source-target queries* possible. The queries return *exact* solutions. *Low space consumption* is a constraint. Furthermore, the method should be *scale-invariant*, i.e., it should be optimised not only for long paths. In other words, the running time of the computation of a shortest path (e.g. from Karlsruhe to Saarbrücken) in a large graph (e.g. Western Europe) should be not much higher than the running time of the same computation in a smaller graph (e.g. Germany).

Related Work

There is so much literature on shortest paths and preprocessing that we can only highlight selected results that help to put our work into perspective. For recent, more detailed overviews we refer to [14, 44, 11]. In the following, *speedup* refers to a comparison of average query times to those of the unidirectional variant of DIJKSTRA's algorithm that stops when the target is found. These speedup factors provide an indication of the performance of each approach. However, it is important to note that the speedups are likely to depend on the size and structure of the graph that is used for the experiments. Therefore, since each author uses different graphs, these numbers have to be interpreted with caution.

Without Preprocessing. The main focus of **theoretical** work on shortest paths has been how to reduce or avoid the overhead of priority queue operations. The original version of DIJKSTRA's algorithm [9] runs in $O(n^2)$. This bound has been improved several times, e.g., to $O(m \log n)$ using binary heaps [45], $O(m + n \log n)$ using Fibonacci heaps [12], $O(m \log \log n)$ [30, 33], and $O(m + n \log \log n)$ using a sophisticated integer priority queue [35, 37] that supports *deleteMin* operations in $O(\log \log n)$ and all other operations in constant time. For integer edge weights in a range from 0 to C, DIAL proposed an $O(m + nC)$

algorithm using buckets [8]. This bound has been improved to $O(m \log\log C)$ [41], $O(m + n\sqrt{\log C})$ [1], and $O(m+n\log\log C)$ [35, 37]. Linear time algorithms for the single source shortest path problem have been presented for *planar* [18] and *undirected* graphs [31, 32]. MEYER [21] gives an algorithm that works in linear time with high probability on an arbitrary directed graph with random edge weights uniformly distributed in the interval $[0, 1]$. However, so far, no linear time algorithm (with respect to the worst case) for directed graphs is known.

Experimental studies [6] indicate that in **practice** even very simple priority queues like binary heaps only induce a factor 2–3 overhead compared to highly tuned ones. In particular, it does not pay to accelerate *decreaseKey* operations since they occur comparatively rarely in the case of sparse road networks.

Bidirectional search is a classical technique that has the potential to give a speedup of up to a factor of two. It simultaneously searches forward from s and backwards from t until the search frontiers meet (Fig. 1.2).

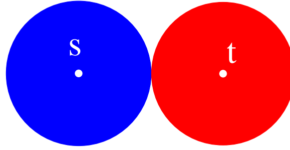


Figure 1.2: Schematic representation of the search space of the bidirectional version of DIJKSTRA's algorithm.

A^* search [16], a heuristic search technique from the field of Artificial Intelligence, is a *goal-directed* approach, i.e., it adds a sense of direction to the search process. For each vertex v, a lower bound $d'(v, t)$ on the distance to t is required. In each step of the search process, the node v is selected that minimises $d(s, v) + d'(v, t)$. This approach can be combined with bidirectional search [23]. The performance of the A^* search depends on a good choice of the lower bounds. If the geographic coordinates of the nodes are given, the Euclidean distance from v to t can be used as lower bound. This leads to a simple, fast, and space efficient method, which, however, gives only a small speedup, in particular when the edge weights are not Euclidean distances, but, for instance, travel times.

With Preprocessing. An extreme case would be to precompute all shortest paths. This allows constant time queries, but is prohibitive for large graphs due to space and time constraints. In general, there is a trade-off between the time needed for *precomputation*, the *space* needed for storing the precomputed information, and the resulting *query time*.

Perhaps the most interesting **theoretical** results on route planning are algorithms for *planar* graphs that might be adaptable to route networks since those are almost planar. Using $O(n \log^3 n)$ preprocessing time, query time $O(\sqrt{n} \log^2 n)$ can be achieved [10] for directed planar graphs without negative cycles. In a planar graph with integer edge weights in a range from 0 to C, queries accurate within a factor $(1 + \varepsilon)$ can be answered in time $O(\log\log(nC) + 1/\varepsilon)$ using $O(n(\log n)(\log(nC))/\varepsilon)$ space and $O(n(\log n)^3(\log(nC))/\varepsilon^2)$ preprocessing time [34, 36].

For *undirected* graphs that are not necessarily planar, THORUP and ZWICK presented a distance oracle [38, 39] that answers queries in constant time using $O(m\sqrt{n})$ expected time for preprocessing and $O(n\sqrt{n})$ space; the *approximate* distance returned is accurate within

a factor of three. Furthermore, they show that any approximate distance oracle for *directed* graphs must use at least $\Omega(n^2)$ bits of storage on at least one n-vertex graph. In **practice**, the graphs used for the USA or Western Europe already have around 20 million nodes so that significantly superlinear preprocessing time or even slightly superlinear space is prohibitive. Hence, the above approaches seem not directly applicable to the problem at hand.

In [13, 14], an algorithm is presented that is based on A^* search, *landmarks*, and the triangle inequality. After selecting a small number of landmarks, for all nodes, the distances to and from each landmark are precomputed. For two nodes v and t, the triangle inequality yields for each landmark ℓ a lower bound $d'(v,t) := d(\ell,t) - d(\ell,v) \leq d(v,t)$. The maximum of these lower bounds is used during an A^* search. For global queries, about 16 global shortest path computations during preprocessing suffice to achieve a speedup factor of around 16 in a road network consisting of about 6.7 million nodes. However, the landmark method needs a lot of space – one distance value for each node-landmark pair. It is also likely that for real applications each node will need to store distances to different sets of landmarks for global and local queries. Hence, landmarks have very fast preprocessing and reasonable speedups but consume too much space for very large networks.

Reach based routing [15] excludes nodes from consideration if they do not contribute to any path long enough to be of use for the current query. Speedups up to ten (17 when combined with A^*) are reported for graphs with about 400 000 nodes using more than two hours preprocessing time. Our method is an order of magnitude faster in terms of both query and preprocessing time.

High speedups are reported for *geometric containers* [27, 42, 44]. For each edge e, the set $S(e)$ is determined that contains all nodes that can be reached on a shortest path starting with e. Then, a simple geometric container $C(e)$ (e.g. a rectangular bounding box) is computed that contains at least all elements of $S(e)$. During the execution of DIJKSTRA's algorithm, an edge e can be ignored if the target node lies outside $C(e)$. The preprocessing step of this approach requires a very expensive all-pairs shortest paths computation.

A related method, which achieves speedups of up to a factor of 1 400 in a road network with about one million nodes [19], is based on *edge flags* [20, 19, 22]. The graph is partitioned into k regions. For each edge e and each region r, one flag is computed that indicates whether e lies on a shortest path to a node in region r. In order to determine the edge flags, for each edge that leaves a region, one shortest paths computation is performed. After these preprocessing steps have been completed, DIJKSTRA's algorithm can take advantage of the edge flags: edges have to be relaxed only if the flag of the region that the target node belongs to is set. Note that the preprocessing costs of this approach are better than those of the geometric containers. Still, the edge flag method is probably too slow when it has to deal with very large road networks consisting of several millions of nodes since the preprocessing of less than half a million nodes already takes more than two hours [19]. An extension to multiple levels, which reduces the space consumption, is suggested in [22].

The previous approach closest to ours is the *separator based multilevel method* [27, 28, 26]. The idea is to partition the graph into small subgraphs by removing a (hopefully small) set of separator nodes. These separator nodes together with edges representing precomputed paths between them constitute the next level of the graph. Queries then only need to search in the partitions of s and t and in the higher level graph. This process can be iterated. Speedups around ten are reported for railway transportation problems [28] and for road networks [44] that contain mostly nodes with degree two. Disadvantages compared to our

method are that performance depends on very small (and thus hard to find) separators and that the higher level graphs get quite dense so that going to many levels quickly reaches a point of diminishing return. In contrast, our method, which is based on a different notion of multilevel graphs, has a very simple definition of what constitutes the higher level graphs and our higher level graphs remain sparse.

Many of the above techniques can be combined. In [27], a *combination* of a special kind of geometric container, the separator based multilevel method, and A^* search yields a speedup of 62 for a railway transportation problem. In [17], combinations of A^* search, bidirectional search, the multilevel method, and geometric containers are studied: Depending on the graph type, different combinations turn out to be best. For real-world graphs, a combination of bidirectional search and geometric containers leads to the best running times.

In contrast to our method, some approaches (e.g. geometric containers) require for each node its geographic coordinates, which might not always be available. However, there are studies that indicate that it is possible to *generate a layout* of a graph so that speedup techniques can be applied successfully. In some cases (where an original layout is available), generated layouts even result in a slightly higher speedup than the original layout does. [4, 5] deals with the special case of a timetable information system; a more general approach is presented in [43].

Our Approach

Let us consider the following naive route planning method:

1. Look for the next reasonable motorway.

2. Drive on motorways to a location close to the target.

3. Leave the motorway and search the target starting from the motorway exit.

Of course, it is true that this fast method does not always yield the optimal solution, but, in many cases, we obtain a reasonable approximation (provided that source and target are not too close together and that we travel in a country whose motorway network is well developed). This naive route planning method is based on a simple rule of thumb: when we are on our way to a remote target and pass by a city on a motorway, it usually does not pay to leave the motorway and look for a faster way through the city; in other words, usually, we can safely ignore all 'less important' city streets and stick to the 'more important' motorway since we *know* that the motorway provides the fastest way. The approach that is used by some commercial route planning systems is based on the above idea:

1. Search from the source and target node ('*bidirectional*') within a certain radius (e.g. 20 km), consider *all roads*.

2. Continue the search within a larger radius (e.g. 100 km), consider only *national roads and motorways*.

3. Continue the search, consider only *motorways*.

Note that the actual implementations of this approach are more sophisticated than our simplified presentation suggests. Again, we get a method which is fast, but still returns inaccurate

results – albeit better ones than those of the naive route planning method. We cannot guarantee exact results because we cannot exclude that sometimes it actually might be better to leave a 'more important' road (e.g. a motorway) and use some 'less important' street (e.g. a local road) that provides some kind of shortcut. In other words, a street that we considered to be 'less important' might turn out to be 'more important' than its category suggests. This observation is the starting point of our approach.

Similar to the commercial approach, we first perform some kind of *local search* from s and from t and then switch to searching in a *highway network* that is much thinner than the complete graph (Fig. 1.3). Our main contribution is the fact that we define the notion of *local*

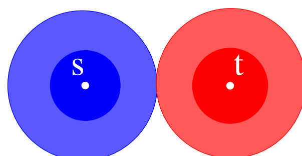


Figure 1.3: Schematic representation of the local search (dark colours) and the search in the highway network (light colours).

search and *highway network* appropriately so that *exact* shortest paths can be computed. This is very simple. We define local search to be a search that visits the H closest nodes from s (or t) where H is a tuning parameter. This definition already fixes the highway network. An edge $(u, v) \in E$ should be a highway edge if there are nodes s and t such that (u, v) is on the shortest path from s to t, v is not within the H closest nodes from s, and u is not within the H closest nodes from t.

At first glance it might appear that a (prohibitively expensive) all-pairs shortest path computation is needed to find the highway network. However, we will show that each highway edge is also within some local shortest path tree B rooted at some $s \in V$ such that all leaves of B are 'sufficiently far away' from s.

So far, the highway network still contains all the nodes of the original network. However, we can prune it significantly: Isolated nodes are not needed. Trees attached to a biconnected component can only be traversed at the beginning and end of a path. Similarly, paths consisting of nodes with degree two can be replaced by a single edge. The result is a *contracted highway network* that only contains nodes of degree at least three.

We expect that search in the two-level network defined above can already be used to achieve speed comparable to some currently used commercial systems without sacrificing exactness. However, we can continue, define local search on the highway network, find a 'superhighway network', contract it, and so on. We arrive at a multilevel highway network – *highway hierarchy*. Now, the query algorithm works in the following way: first, perform a local search in the original graph (level 0); second, switch to the highway network (level 1) and perform a local search in the highway network; then, switch to the next level of the highway hierarchy, and so on. Figure 1.4 gives a schematic representation of the search space, Fig. 1.5 a real-world example. Another, more detailed example is given in Section B.3.

Outline

Chapter 2 gives a more formal definition of the *basic concepts* used in this paper. In Chapter 3, we deal with the efficient *construction* of the highway hierarchies. First, we give an algorithm that computes the exact highway network of a given graph. Then, we introduce

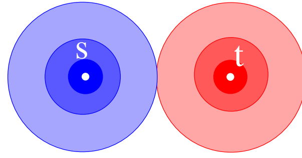


Figure 1.4: Schematic representation of the search in level 0 (dark colours), level 1 (light colours), and level 2 (very light colours) of a highway hierarchy.

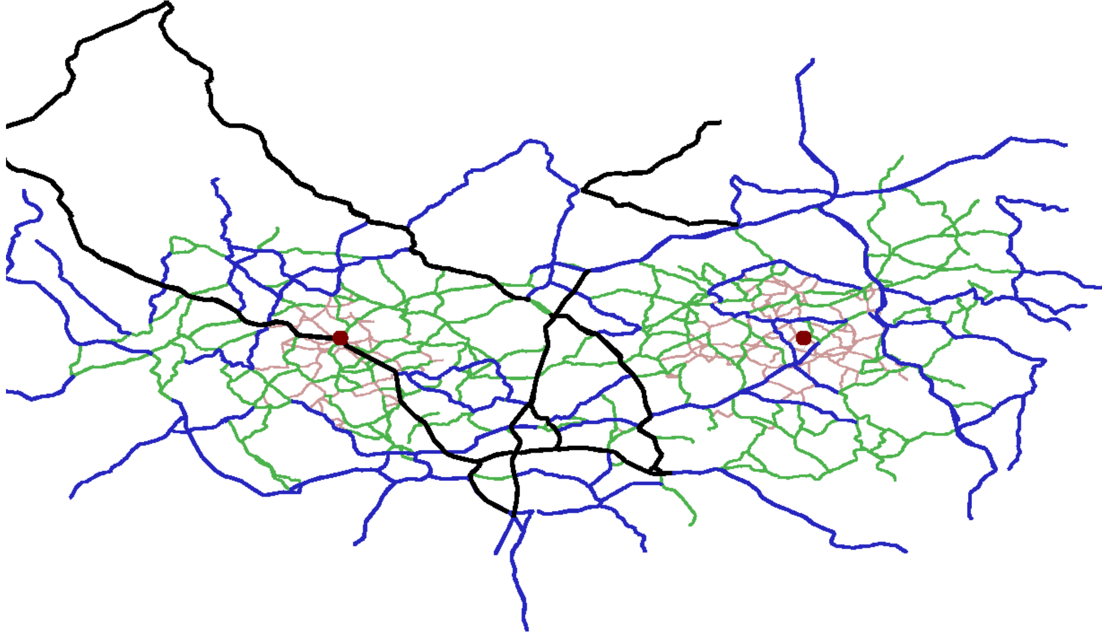


Figure 1.5: Search space for a query from Limburg (a German city) to a location 100 km east of the source node. Source and target are marked by a circle. The thicker the line, the higher the search level. Note that edges representing long subpaths are not drawn as direct shortcuts, but by showing the actual geographic route taken.

a tuning parameter that allows to speed up the construction at the price of no longer computing the actual highway network, but a superset of it. Finally, we show how the highway network can be contracted. Note that the construction algorithm only works if the shortest path search performed during precomputation computes not just arbitrary shortest paths, but *canonical shortest paths*, i.e., the algorithm has to break ties in such a way that subpaths of shortest paths that are determined by the search are also determined. In Appendix A we show that any priority queue data structure can be modified to guarantee canonical shortest paths. Chapter 4 develops a *query* algorithm that uses highway hierarchies. After several correctness preserving transformations, we get a bidirectional, DIJKSTRA-like search in a single graph that contains all levels. The only modifications affect the selection of edges to be relaxed and how to finish the search when the search frontiers from s and t meet.

Chapter 5 highlights some interesting aspects of an *implementation* of our approach. In Chapter 6, we summarise *experiments* using detailed road networks for Western Europe and the USA. Using a uniform neighbourhood size H of 125 and 225, respectively, the graphs shrink geometrically from level to level. This leads to a preprocessing time of around four hours and average query times below 8 ms. Possible *future improvements* are discussed in Chapter 7. Several real-world *examples* are included in Appendix B.

Chapter 2

Preliminaries

2.1 Shortest Paths and DIJKSTRA's Algorithm

Graphs and Paths. We expect an *undirected* graph $G = (V, E)$ with n nodes and m edges e with *nonnegative* weights $w(e)$ as input.[1] We assume w.l.o.g. that there are no self-loops, parallel edges, and zero weight edges in the input – they could be dealt with easily in a preprocessing step. The *length* $w(P)$ of a path P is the sum of the weights of the edges that belong to P. $P^* = \langle s, \ldots, t \rangle$ is a *shortest path* if there is no path P' from s to t such that $w(P') < w(P^*)$. The *distance* $d(s, t)$ between s and t is the length of a shortest path from s to t. If $P = \langle s, \ldots, s', u_1, u_2, \ldots, u_k, t', \ldots, t \rangle$ is a path from s to t, then $P|_{s' \to t'} = \langle s', u_1, u_2, \ldots, u_k, t' \rangle$ denotes the *subpath* of P from s' to t'. An example for these concepts is given in Fig. 2.1.

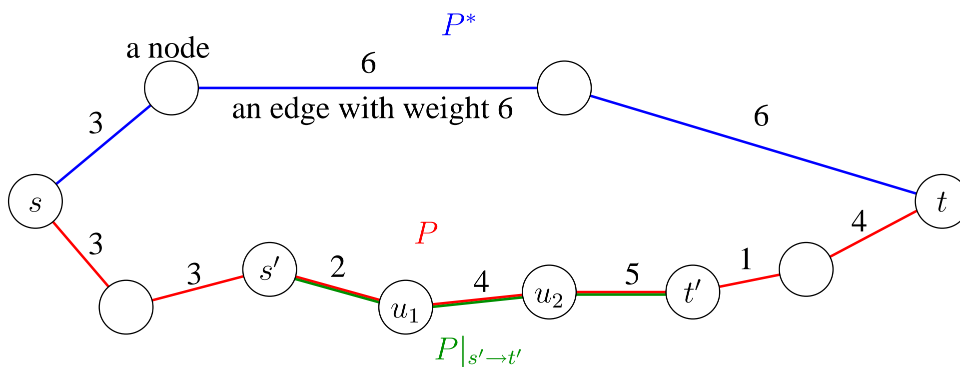


Figure 2.1: An undirected graph with $n = 10$ nodes and $m = 10$ edges. Two paths P and P^* from node s to node t are marked. The length $w(P)$ of P is 22; $w(P^*) = 15$. P^* is a shortest path. The distance from s to t is $d(s, t) = w(P^*) = 15$. A subpath $P|_{s' \to t'}$ of P from s' to t' is highlighted.

DIJKSTRA's Algorithm. DIJKSTRA's algorithm [9] can be used to solve the *single source shortest path (SSSP) problem*, i.e., to compute the shortest paths from a single source node s to all other nodes in a given graph. It is covered by virtually any textbook on algorithms, e.g.

[1] Unless otherwise stated, we always deal with *undirected* edges. The restriction to undirected graphs simplifies the presentation of our approach and the implementation. However, our method can be generalised to *directed* graphs. In further footnotes we will outline what has to be done.

[7, 29]. For the sake of self-containment, we give an outline introducing the terminology used throughout this thesis.

Starting with the source node s as root, DIJKSTRA's algorithm grows a *shortest path tree* that contains shortest paths from s to all other nodes. During this process, each node of the graph is either *unreached*, *reached*, or *settled*.

- A node that already belongs to the tree is *settled*. If a node u is settled, a shortest path P^* from s to u has been found and the distance $d(s, u) = w(P^*)$ is known.
- A node that is adjacent to a settled node is *reached*. Note that a settled node is also reached. If a node u is reached, a path P from s to u, which might not be the shortest one, has been found and a *tentative distance* $w(P)$ is known.
- Nodes that are not reached are *unreached*.

The nodes that are reached but not settled are managed in a *priority queue*, which supports the operations

- *insert* – insert an element into the priority queue,
- *deleteMin* – retrieve the element with the smallest key and remove it from the priority queue,
- *decreaseKey* – set the key of an element that already belongs to the priority queue to a new value that is less than the old value.

The *key* of a node in the priority queue is its tentative distance.

Initially, s is inserted into the priority queue with the tentative distance 0. Thus, s is reached, all other nodes are unreached. While the priority queue is not empty, the node u with the smallest tentative distance is removed (*deleteMin*) and added to the shortest path tree, i.e., u becomes settled. Furthermore, u's outgoing edges are *relaxed*:

- if an edge (u, v) leads to an unreached node v, v is added to the priority queue (*insert*); now, v is reached;
- if an edge (u, v) leads to a reached but not settled node v, v's key in the priority queue is updated (*decreaseKey*) provided that the length of the path from s via u to v is less than v's old key;
- if an edge (u, v) leads to a settled node v, it is ignored.

Canonical Shortest Paths. A *selection of shortest paths* $\mathcal{SP}$ contains for each connected pair $(s, t) \in V \times V$ exactly one shortest path from s to t. Such a selection is called *canonical* if $P = \langle s, \ldots, s', \ldots, t', \ldots, t\rangle \in \mathcal{SP}$ implies that $P|_{s' \to t'} \in \mathcal{SP}$. The elements of a canonical selection are called *canonical shortest paths*. If DIJKSTRA's algorithm is started from each node $s \in V$, for each connected pair (s, t) exactly one shortest path is determined. In Appendix A some modifications of DIJKSTRA's algorithm are described which ensure that the obtained selection of shortest paths is canonical. Figure 3.4 in Section 3.1 explains the importance of this concept.

2.2 Highway Hierarchy

Locality. Let us fix any rule that decides which element DIJKSTRA's algorithm removes from the priority queue in the case that there is more than one queued element with the smallest key. Then, during a DIJKSTRA search from a given node s, all nodes are settled in a fixed order. The *Dijkstra rank* $r_s(v)$ of a node v is the rank of v w.r.t. this order. s has DIJKSTRA rank $r_s(s) = 0$, the closest neighbour v_1 of s has DIJKSTRA rank $r_s(v_1) = 1$, and so on. For a given node s, the distance of the H-closest node from s is denoted by $d_H(s)$, i.e., $d_H(s) = d(s, v)$, where $r_s(v) = H$. The *H-neighbourhood* $\mathcal{N}_H(s)$ (or just *neighbourhood* $\mathcal{N}(s)$) of s is $\mathcal{N}(s) := \{v \in V \mid d(s, v) \leq d_H(s)\}$. [2] Figure 2.2 gives an example.

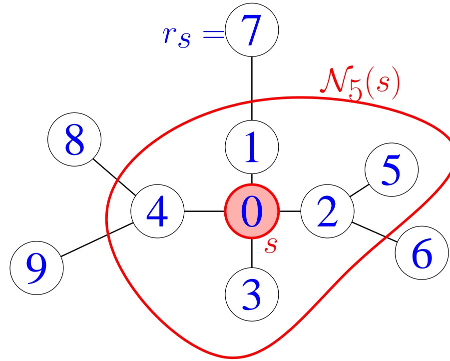


Figure 2.2: A graph with a given source node s. The DIJKSTRA rank of all nodes and the 5-neighbourhood of s are depicted. The weight of an edge is the length of the line segment that represents the edge in this figure.

Highway Hierarchy. For a given parameter H, the *highway network* $G_1 = (V_1, E_1)$ of a graph G is defined by the set E_1 of edges: an edge $(u, v) \in E$ belongs to E_1 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 with the property that $v \notin \mathcal{N}_H(s)$ and $u \notin \mathcal{N}_H(t)$. The set V_1 is the maximal subset of V such that G_1 contains no isolated nodes. Figure 2.3 illustrates this definition, Fig. 2.4 and 2.5 show examples of highway networks.

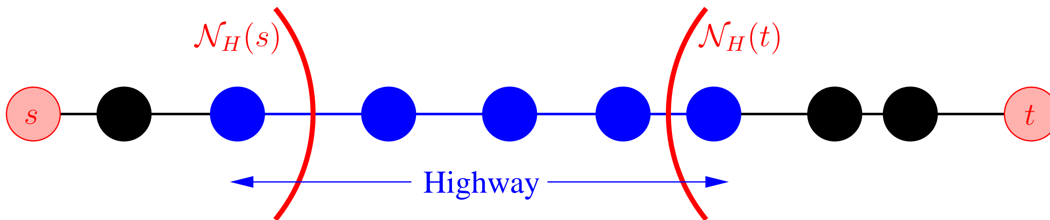


Figure 2.3: A canonical shortest path from a node s to a node t. Edges that leave the neighbourhood of s or t and edges that are completely outside the neighbourhoods of s and t are highway edges.

[2] For directed graphs we also need an analogous value $\bar{d}_H(\cdot)$ that refers to the reverse graph $\bar{G} := (V, \{(v, u) \mid (u, v) \in E\})$. $\bar{\mathcal{N}}(\cdot)$ is defined correspondingly. From now on, whenever the target node t or the backward search from t is concerned, we have to keep in mind that $\bar{G}$, $\bar{d}_H(\cdot)$, and $\bar{\mathcal{N}}(\cdot)$ apply.

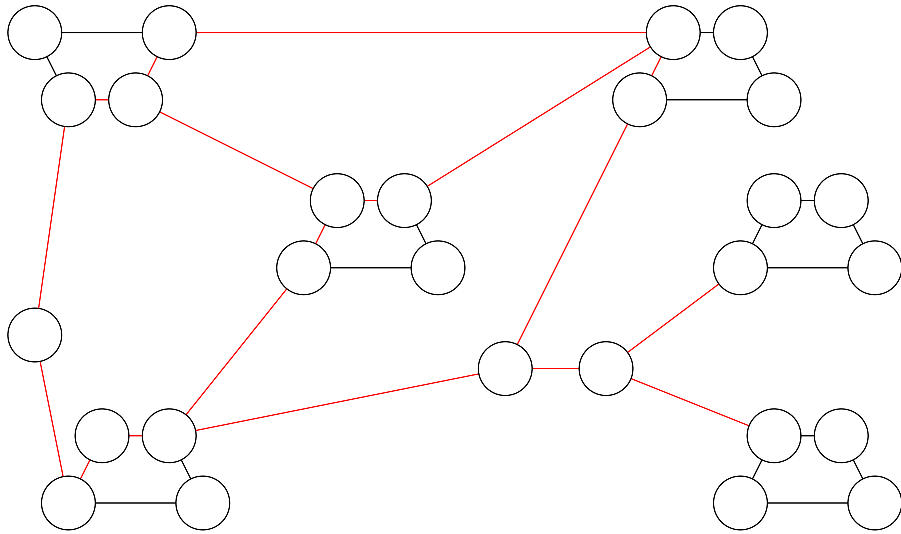


Figure 2.4: A simple example of a highway network. The highway edges are highlighted. The weight of an edge is the length of the line segment that represents the edge in this figure. The neighbourhood size H is 3.

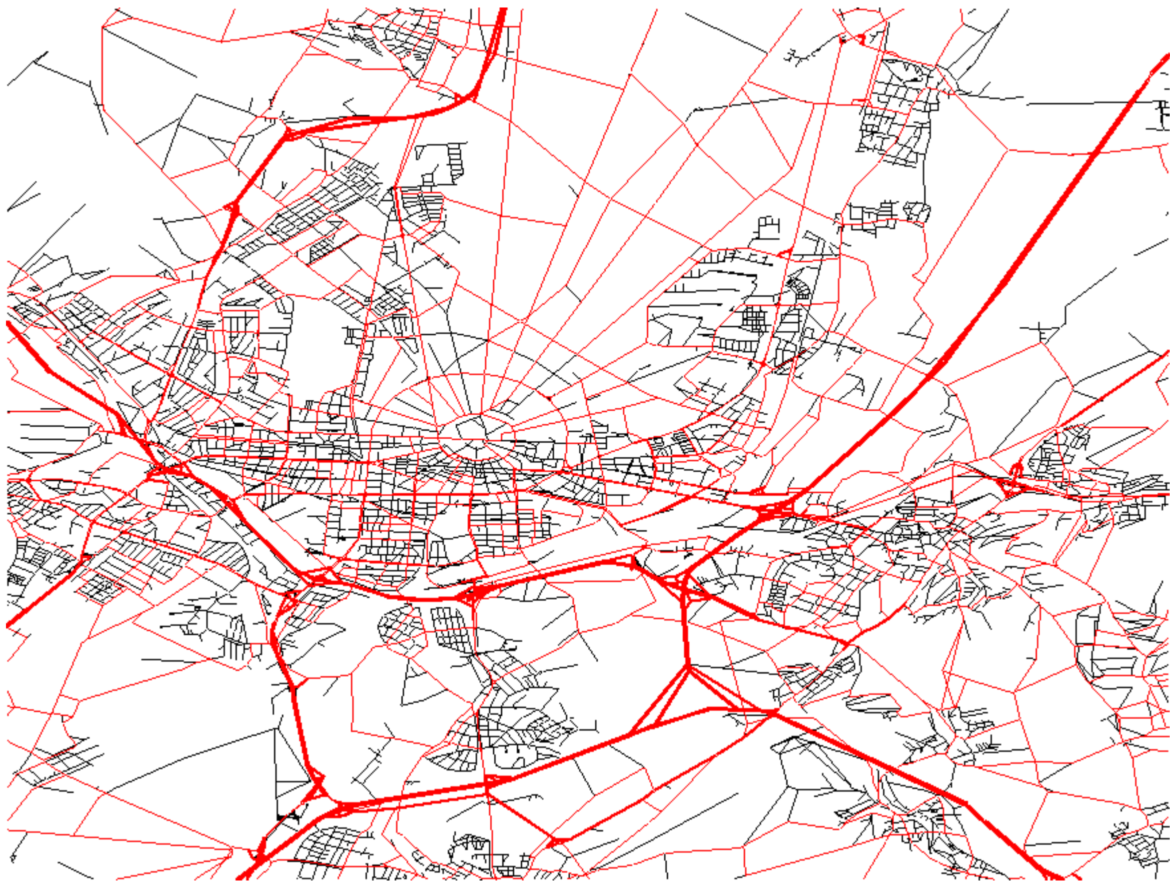


Figure 2.5: The highway network of Europe, clipped by a bounding box around Karlsruhe. The highway edges are highlighted.

The *2-core* of a graph is the maximal vertex induced subgraph with minimum degree two. A graph consists of its 2-core and *attached trees*, i.e., trees whose roots belong to the 2-core, but all other nodes do not belong to it (Fig. 2.6). A *line* in a graph is a path $\langle u_0, u_1, \dots, u_k \rangle$

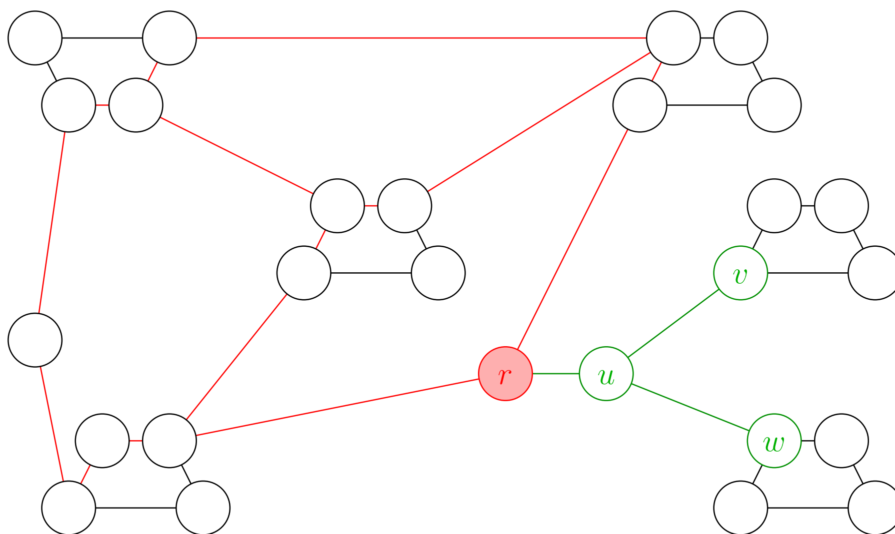


Figure 2.6: The 2-core of the highway network from Fig. 2.4 and an attached tree whose root r belongs to the 2-core. Note that u does not belong to the 2-core although it has degree 3 in the highway network.

consisting of more than two nodes where the inner nodes $u_1, \dots, u_{k-1}$ have degree two (Fig. 2.7). From the highway network G_1 of a graph G, the *contracted highway network* G'_1 of the graph G is obtained by taking the 2-core of G_1 and then removing the inner nodes of all lines $\langle u_0, u_1, \dots, u_k \rangle$ and replacing each line by an edge (u_0, u_k) (Fig. 2.8). Thus, the highway network G_1 consists of the contracted highway network (also called *core*) G'_1 and some *components*, where 'component' is used as a generic term for 'attached tree' and 'line'. In this thesis, 'components' is always used in this specific sense, but never to denote 'connected components' in general. Sometimes it is convenient to use the term 'endpoint(s) of a component' to denote either the endpoints of a line or the root of a tree.

The *highway hierarchy* is obtained by applying the process that leads from G to G'_1 iteratively. The original graph $G_0 := G'_0 := G$ constitutes level 0 of the highway hierarchy, G_1 corresponds to level 1, the highway network G_2 of the graph G'_1 is called level 2, and so on. As an example, the European highway hierarchy is visualised in Section B.2.

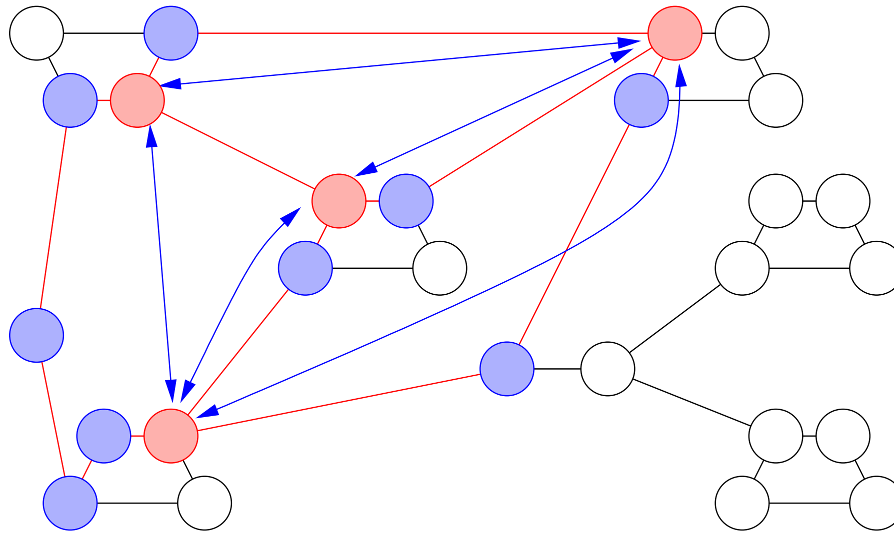


Figure 2.7: The 2-core of the highway network from Fig. 2.4 containing five lines. Endpoints and inner nodes of lines are marked. Both endpoints of a line are connected by a shortcut.

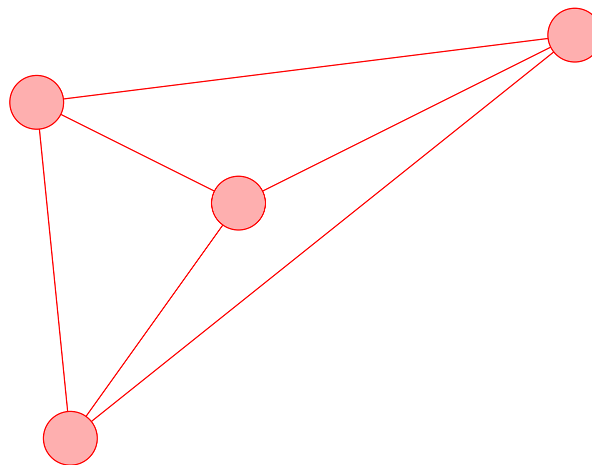


Figure 2.8: The *contracted highway network* obtained from the highway network from Fig. 2.4.

Chapter 3

Construction

3.1 Fast Construction of the Highway Network

We start with an empty set of highway edges E_1. For each node s_0, two phases are performed: the forward construction of a partial shortest path tree B and the backward evaluation of B. The construction is done by an SSSP search from s_0; during the evaluation phase, paths from the leaves of B to the root s_0 are traversed and for each edge on these paths, it is decided whether to add it to E_1 or not. The crucial part is the specification of an abort criterion for the SSSP search in order to restrict it to a 'local search'.

Phase 1: Construction of a Partial Shortest Path Tree. A DIJKSTRA search from s_0 is executed. During the search, a reached node is either in the state *active* or *passive*. The source node s_0 is active; each node that is reached for the first time (*insert*) and each reached node that is updated (*decreaseKey*) adopts the activation state from its (tentative) parent in the shortest path tree B. When a node p is settled and the *abort criterion* (see below) is fulfilled, p's state is set to passive. When no active unsettled node is left, the search is *aborted* and the growth of B stops.

Abort Criterion. When a node p is settled using the path P' as depicted in Fig. 3.1, then p's state is set to passive if $|\mathcal{N}(s_1) \cap \mathcal{N}(p)| \leq 1$.

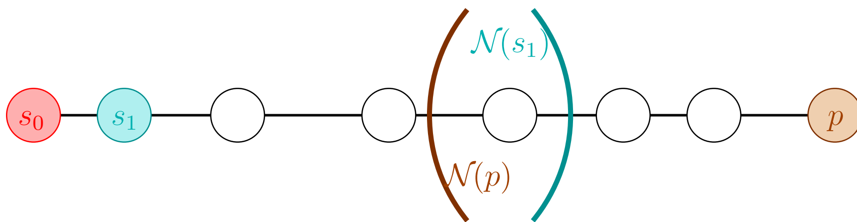


Figure 3.1: Abort criterion.

An example for Phase 1 of the construction is given in Fig. 3.2. Note that the simpler abort criterion $\mathcal{N}(s_0) \cap \mathcal{N}(p) = \emptyset$ does not work – Fig. 3.3 gives a counter-example. The intuitive reason for s_1 (which is the first successor of s_0 on the path P') to appear in the abort criterion is the following: When we deactivate a node p during the search from s_0,

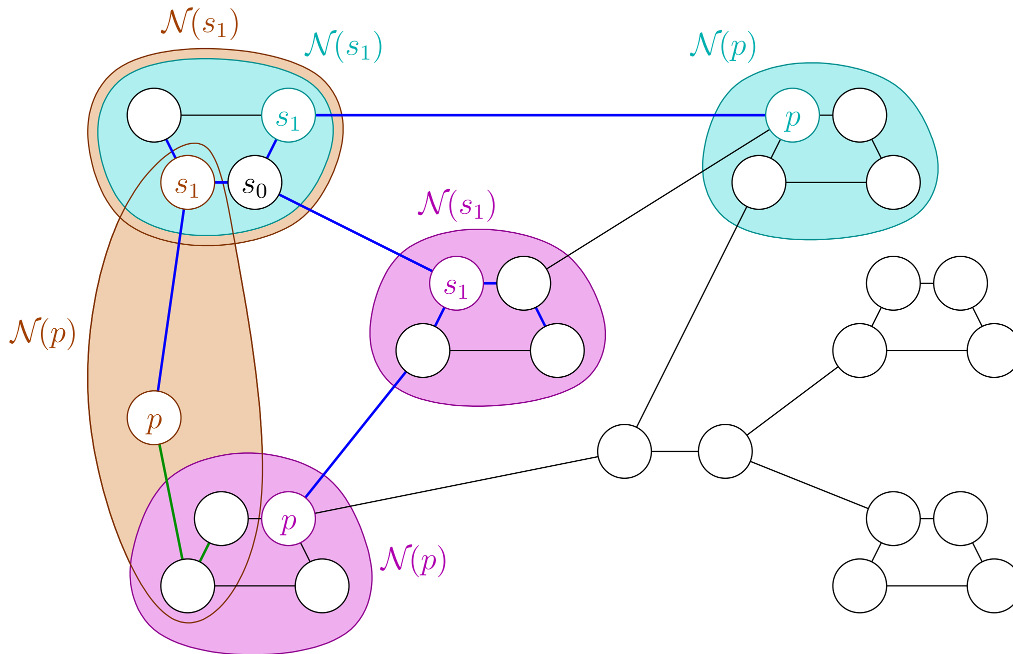


Figure 3.2: An example of Phase 1 of the construction. The weight of an edge is the length of the line segment that represents the edge in this figure. The neighbourhood size H is 3. An SSSP search is performed from s_0. The abort criterion applies three times: the involved nodes s_1 and p and the corresponding neighbourhoods are marked in cyan, magenta, and brown, respectively. In the brown case, the intersection of the concerned neighbourhoods contains exactly one element; in the other two cases, the intersections are empty. All edges that belong to s_0's partial shortest path tree are coloured: edges that leave active nodes are blue, edges that leave passive nodes are green.

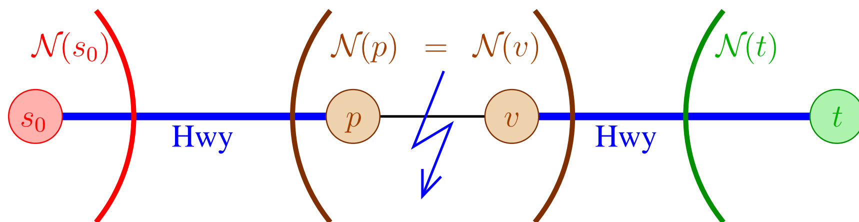
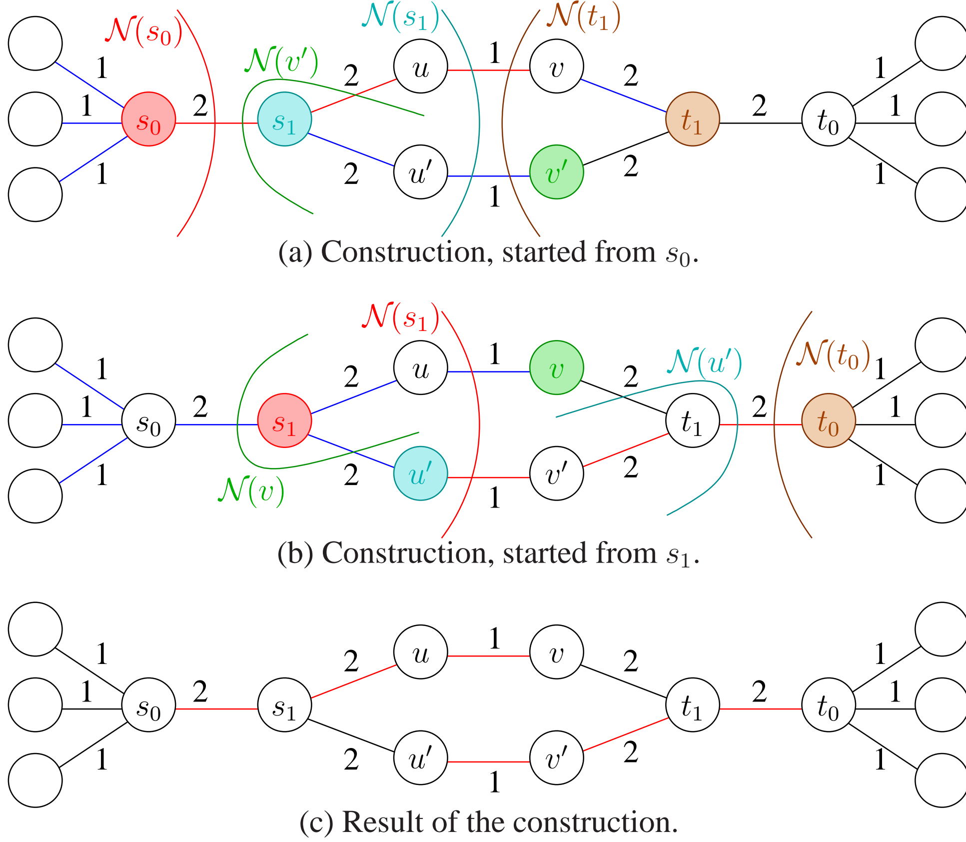


Figure 3.3: Counter-example for the wrong abort criterion $\mathcal{N}(s_0) \cap \mathcal{N}(p) = \emptyset$. If the wrong abort criterion was applied, the search from s_0 and from t would be aborted at p and v, respectively. Hence, the edge (p, v) would not be added to the highway network. However, for the shortest path search from s_0 to t, this edge would have to belong to the highway network.

we decide to ignore everything that lies behind p. We are free to do this because the abort criterion ensures that s_1 can take 'responsibility' for the things that lie behind p, i.e., further important edges will be added during the search from s_1. (Of course, s_1 will refer a part of its 'responsibility' to its successor.) In this context, Fig. 3.4 illustrates why the concept of *canonical shortest paths* has been introduced in Section 2.1.



(a) Construction, started from s_0.

(b) Construction, started from s_1.

(c) Result of the construction.

Figure 3.4: An example of the construction without *canonical shortest paths*. The neighbourhood size H is 3. In (a) and (b), the edges are coloured that belong to the partial shortest path tree rooted at s_0 and s_1, respectively; edges that are added to the highway network are red. We assume that the search from s_0 (a) and from t_1 'prefers' the upper branch (u and v), while the search from s_1 (b) and from t_0 'prefers' the lower branch (u' and v'). The result is a 'broken' highway network (c). In contrast, the concept of canonical shortest paths guarantees that s_0 and s_1 'prefer' the same branch so that 's_1 can finish what s_0 started'.

Phase 2: Selection of the Highway Edges. During Phase 2, exactly all edges (u, v) are added to E_1 that lie on paths $\langle s_0, \ldots, u, v, \ldots, t_0 \rangle$ in the partial shortest path tree B with the property that $v \notin \mathcal{N}(s_0)$ and $u \notin \mathcal{N}(t_0)$, where t_0 is a leaf of B. The example from Fig. 3.2 is continued in Fig. 3.5.

Lemma 1 *Consider a shortest path $\langle u, \ldots, t, \ldots, t' \rangle$, where $t' \in \mathcal{N}(u)$. Then, $t \in \mathcal{N}(u)$.*

Proof: Follows directly from the definition of the neighbourhood since $d(u, t) < d(u, t')$. $\square$

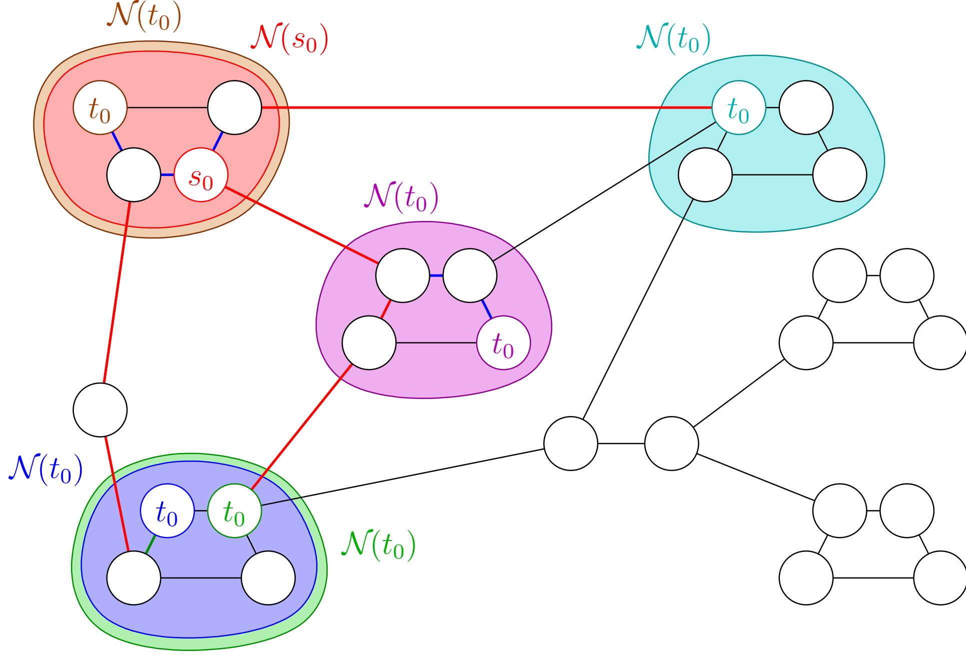


Figure 3.5: An example of Phase 2 of the construction. s_0's partial shortest path tree has five leaves t_0, which are marked in different colours. The edges that are added to E_1 are highlighted.

Lemma 2 *Consider a shortest path $\langle u, \dots, t, \dots, t' \rangle$, where $u \in \mathcal{N}(t')$. Then, $u \in \mathcal{N}(t)$.*

Proof: Since $\forall v \in \mathcal{N}(t) : d(v, t') \leq d(v, t) + d(t, t') \leq d_H(t) + d(t, t')$, we have, in particular,

$$\max_{v \in \mathcal{N}(t)} d(v, t') \leq d_H(t) + d(t, t'). \tag{3.1}$$

Furthermore, $|\mathcal{N}(t)| \geq H + 1$ implies that $\max_{v \in \mathcal{N}(t)} r_{t'}(v) \geq H$ (because the DIJKSTRA ranks are unique and the smallest DIJKSTRA rank is 0). Thus,

$$\max_{v \in \mathcal{N}(t)} d(v, t') \geq d_H(t'). \tag{3.2}$$

(3.1) and (3.2) lead to

$$d_H(t') \leq d_H(t) + d(t, t'). \tag{3.3}$$

The condition that $u \in \mathcal{N}(t')$ implies

$$d(u, t') \leq d_H(t'). \tag{3.4}$$

(3.3) and (3.4) lead to

$$d(u, t') \leq d_H(t) + d(t, t') \iff d(u, t') - d(t, t') \leq d_H(t) \iff d(u, t) \leq d_H(t). \tag{3.5}$$

Thus, $u \in \mathcal{N}(t)$. □

Lemma 3 *Consider a shortest path P where t is not a predecessor[1] of s and v is not a predecessor of u. Furthermore, $u \in \mathcal{N}(t)$ and $v \in \mathcal{N}(s)$ — a "cross-over situation". Then, $u \in \mathcal{N}(s)$ and $v \in \mathcal{N}(t)$.*

[1] Note that 'a predecessor' does *not* necessarily mean 'the first/direct predecessor'.

Proof: We prove the statement $u \in \mathcal{N}(s)$; the proof of $v \in \mathcal{N}(t)$ is symmetric. We distinguish between three cases.

1. $u = s$. Trivial.
2. u is a predecessor of s, i.e., $P = \langle \dots, u, \dots, s, \dots, t, \dots \rangle$. $u \in \mathcal{N}(s)$ follows from Lemma 2 since $u \in \mathcal{N}(t)$.
3. u is a successor of s, i.e., $P = \langle \dots, s, \dots, u, \dots, v, \dots \rangle$. $u \in \mathcal{N}(s)$ follows from Lemma 1 since $v \in \mathcal{N}(s)$. □

Theorem 1 *An edge $(u, v) \in E$ is added to E_1 by the construction algorithm iff it belongs to some canonical shortest path $P = \langle s, \dots, u, v, \dots, t \rangle$ and $v \notin \mathcal{N}(s)$ and $u \notin \mathcal{N}(t)$.*

Proof: $\Leftarrow$) Consider the node s_0 on $P|_{s \to v} = \langle s, \dots, s_0, s_1, \dots, u, v \rangle$ such that $v \notin \mathcal{N}(s_0)$ and $d(s_0, v)$ is minimal. Note that $v \in \mathcal{N}(s_1)$ [*]. Such a node s_0 exists because the condition $v \notin \mathcal{N}(s_0)$ is always fulfilled for $s_0 = s$. We show that the edge (u, v) is added to E_1 when Phase 1 and 2 are executed from s_0. It is important to note that $P|_{s_0 \to t}$ is a canonical shortest path and, thus, if a node v' on $P|_{s_0 \to t}$ is settled during Phase 1, then its parent in B is its predecessor on $P|_{s_0 \to t}$. In other words, the shortest path from s_0 to v' that is traversed during Phase 1 is not an arbitrary shortest path, but the canonical shortest path $P|_{s_0 \to v'}$. After Phase 1 has been completed, we distinguish between two cases.

Case $t \in B$. We know that $u \notin \mathcal{N}(t)$. Let t_0 be any leaf of B that is either a descendant of t or t itself. By Lemma 2, we obtain $u \notin \mathcal{N}(t_0)$.

Case $t \notin B$. The search is not continued from some node $t_0 \neq t$ on $P|_{s_0 \to t}$. In general, the search of Phase 1 is not continued from a node t'_0 if and only if there is no canonical shortest path from s_0 via t'_0 to another node, or the abort condition is fulfilled, i.e., there is no active unsettled node left. In this case, the first condition cannot apply since for each node $t'_0 \neq t$ on $P|_{s_0 \to t}$, there is a canonical shortest path from s_0 via t'_0 to another node, namely to t. Hence, the second condition must be fulfilled. We can conclude that t_0 is passive because, otherwise, its successor on $P|_{s_0 \to t}$ would adopt its active state and the search would not be aborted at that time. Since s_0 is active and t_0 is passive, either t_0 or one of its ancestors must have been switched from active to passive. Let p denote the first passive node on $P|_{s_0 \to t} = \langle s_0, s_1, \dots, p, \dots, t_0, \dots, t \rangle$. Due to the definition of the abort condition, we have $|\mathcal{N}(s_1) \cap \mathcal{N}(p)| \leq 1$ [**]. In order to obtain a contradiction, we assume $u \in \mathcal{N}(p)$. Furthermore, we have $v \in \mathcal{N}(s_1)$ [see *]. Lemma 3 yields $u \in \mathcal{N}(s_1)$ and $v \in \mathcal{N}(p)$. Hence, $\{u, v\} \subseteq \mathcal{N}(s_1) \cap \mathcal{N}(p)$. Therefore, $|\mathcal{N}(s_1) \cap \mathcal{N}(p)| \geq 2$, which is a contradiction to [**]. We can conclude that $u \notin \mathcal{N}(p)$. Furthermore, we know that v (and, consequently, u) is not a successor of p: If v was a successor of p, Lemma 1 would yield $v \notin \mathcal{N}(p)$ since $u \notin \mathcal{N}(p)$; this would be a contradiction to Lemma 2 since $v \in \mathcal{N}(s_1)$ [see *]. From the facts that $u \notin \mathcal{N}(p)$, u is not a successor of p and p is not a successor of t_0, we can conclude that $u \notin \mathcal{N}(t_0)$ due to Lemma 2.

So, in both cases, we have $u \notin \mathcal{N}(t_0)$ and t_0 is a leaf of B. Furthermore, u is not a predecessor of s_0 (due to the choice of s_0) and v is not a successor of t_0. From these facts and the specification of Phase 2, we can infer that the edge (u, v) is added to E_1.

$\Rightarrow$) Each path in B from s_0 to a leaf t_0 is a canonical shortest path due to the modifications of DIJKSTRA's algorithm as described in Appendix A. Hence, the claim follows directly from the specification of Phase 2. □

Details on Phase 2. For a node $u \in B$, define L_u as the set of leaves t_0 of B that are the endpoints of paths of the form $\langle s_0, \ldots, u, \ldots, t_0 \rangle$. The *slack* $\Delta(u)$ of a node $u \in B$ is defined in the following way: $\Delta(u) := \min_{t_0 \in L_u} (d_H(t_0) - d(u, t_0))$. For a leaf t_0, we have $L_{t_0} = \{t_0\}$ and $\Delta(t_0) = d_H(t_0)$. The slack of an inner node u can be computed using only the slacks of its children C_u: $\Delta(u) = \min_{c \in C_u} \Delta_c(u)$, where $\Delta_c(u) := \Delta(c) - d(u, c)$. This leads to an equivalent, recursive definition.

The slacks $\Delta(t_0)$ of all leaves t_0 of B are set to $d_H(t_0)$. The tentative slacks $\widehat{\Delta}(u)$ of all other nodes u of B are set to $+\infty$. A FIFO queue Q is filled with all leaves of B (in an arbitrary order). All elements of Q are processed one after the other until Q is empty. We maintain the invariant that the tentative slack $\widehat{\Delta}(u)$ of an element u that is removed from Q is equal to the actual slack $\Delta(u)$. When a node u is removed from Q, we compute $\Delta_u(p) = \Delta(u) - d(p, u)$, where p is the parent of u in B. If $\Delta_u(p) < 0$, the edge (p, u) is added to E_1. If $\widehat{\Delta}(p) = +\infty$ and the node p does not belong to $\mathcal{N}(s_0)$, then p is added to Q. If $\Delta_u(p) < \widehat{\Delta}(p)$, the tentative slack $\widehat{\Delta}(p)$ is set to $\Delta_u(p)$. Figure 3.6 gives an example.

Figure 3.6: An example of the *slack-based method* that realises Phase 2 of the construction. The process is shown only for a part of the tree. As before, the weight of an edge is the length of the line that represents the edge in this figure. For the sake of transparency, the (rounded) weights are given explicitly for the relevant edges. Furthermore, the slacks of the involved nodes are given. Edges that are added to E_1 are red, edges that are not added blue.

Theorem 2 *An edge (u, v) is added to E_1 by the* slack-based method *introduced above iff it lies on a path $\langle s_0, \ldots, u, v, \ldots, t_0 \rangle$ in the partial shortest path tree B with the property that $v \notin \mathcal{N}(s_0)$ and $u \notin \mathcal{N}(t_0)$, where t_0 is a leaf of B.*

Proof: $\Leftarrow$) From the definition of the slack of a node, it follows that

$$\Delta_v(u) = \Delta(v) - d(u, v) \leq d_H(t_0) - d(v, t_0) - d(u, v) = d_H(t_0) - d(u, t_0) < 0$$

because $u \notin \mathcal{N}(t_0)$. Since $v \notin \mathcal{N}(s_0)$, the node v is inserted in Q at some point. When it is removed from Q, $\Delta_v(u)$ is computed and, since it is negative, the edge (u, v) is added to E_1.

$\Rightarrow$) Only edges that belong to a path in B from s_0 to a leaf t_0 are considered. The condition $v \notin \mathcal{N}(s_0)$ is never violated because the traversal from the leaves to the root, and consequently, the addition of edges to E_1, is not continued when a node p belongs to $\mathcal{N}(s_0)$. If an edge (u, v) is added, the condition $\Delta_v(u) < 0$ is fulfilled. Hence, $\Delta(u) = \min_{t_0 \in L_u} (d_H(t_0) - d(u, t_0)) \leq \Delta_v(u) < 0$. Therefore, there is a leaf t_0 such that $d(u, t_0) > d_H(t_0)$, i.e., $u \notin \mathcal{N}(t_0)$. □

Theorem 3 *Let V_B denote the set of nodes of s_0's partial shortest path tree B. Let $G_B = (V_B, E_B)$ denote the subgraph of G that is vertex induced by V_B. The complexity of Phase 1 and 2 started from s_0 is $T_{\mathit{Dijkstra}}(|G_B|)$.*

Proof: The number of nodes of G_B is denoted by n', the number of edges by m'. The complexity of Phase 1 corresponds to the complexity of a SSSP search in G_B started from s_0, i.e., $O(n' + m')$ outside the priority queue plus n' *insert* and n' *deleteMin* operations plus at most m' *decreaseKey* operations. The initialisation of the (tentative) slacks for Phase 2 can be done during Phase 1 without any additional effort. During Phase 2, at most n' nodes are processed. For each node, only a constant number of operations is performed, particularly, only one edge (to the parent node in B) is considered. □

3.2 Speeding up Construction

In the optimal case, the sizes of the partial shortest path trees B are bounded by a small multiple of the neighbourhood size H. However, in less balanced cases, the trees can get quite big. Figure 3.7 gives an example of such a scenario. It shows the road network of Italy including ferry connections. When the search starts from a node s_0 close to a harbour so that a very long ferry edge is relaxed, the search cannot be aborted until the arrival point p of the ferry has been settled and deactivated. This means that all roads that lead to nodes whose distance from s_0 is less than $d(s_0, p)$ have to be traversed. In our example, it might be the case that instead of a local area almost the entire country is involved when, for instance, the edge Genoa–Palermo is relaxed.

In order to deal with this problem, we introduce the concept of *mavericks*: An active node v is declared to be a *maverick* if $d(s_0, v) > f \cdot d_H(s_0)$, where f is a parameter. When all active nodes are mavericks, the search from passive nodes is no longer continued.

In our example, we now have the following situation: When the search is started from Genoa, then Palermo and the arrival points of the other ferries are mavericks because they are very far away from Genoa. Hence, a local area around Genoa is searched until all nodes that have been reached by road have been deactivated. Then, the search on the roads is not continued, but the arrival points of the ferries are settled immediately. Thus, the search has been restricted to a local area plus a few remote nodes where the ferries arrive.

However, this improvement has a disadvantage. Aborting the search at the passive nodes abolishes the guarantee that only shortest paths are found. For instance, it might be faster to drive from Genoa to Palermo by car (crossing the Strait of Messina from the southern tip of the Italian mainland to Sicily by ferry) instead of using the direct ferry link. If we abort the search on the mainland, the direct ferry connection might be wrongly added to the highway network, which should contain only edges that belong to some shortest path.

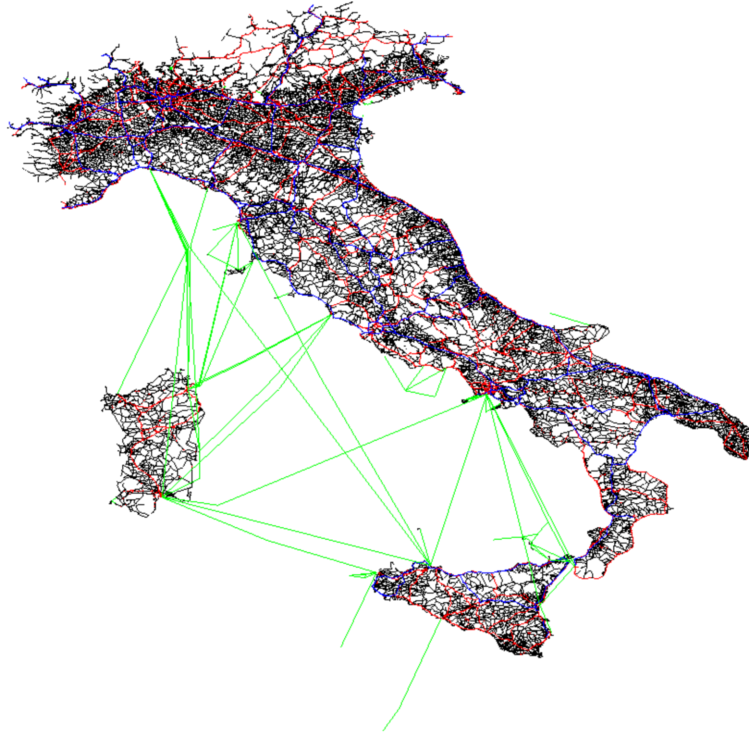


Figure 3.7: The road network of Italy including ferry connections.

Theorem 4 *The accelerated construction method yields a superset of the highway network.*

Proof: The $\Leftarrow$)-part of the proof of Theorem 1 still holds. □

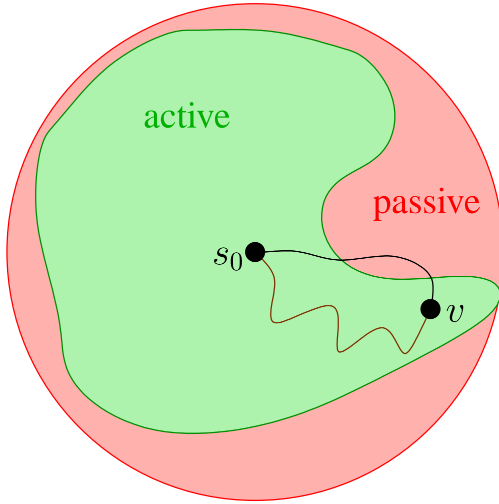
Hence, queries will be slower, but still compute exact shortest paths. Figure 3.8 compares the precise construction method from Section 3.1 with the accelerated method from this section.

The parameter f enables us to adjust the trade-off between construction and query time. $f = \infty$ yields the precise method from Section 3.1, which is comparatively slow, but permits the best query times because the exact highway network is computed. $f = 0$ leads to a very fast construction method, which adds a lot of needless edges to the highway network, which slow down the queries. In Section 6.2 we will look for a good compromise between these two extreme choices of f.

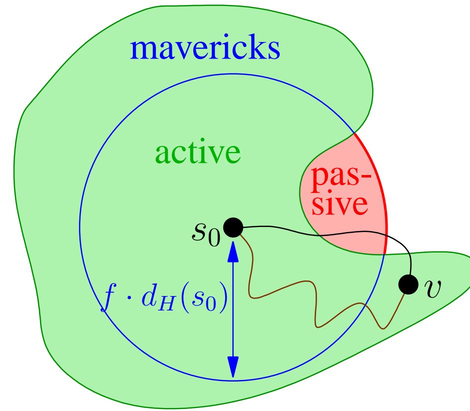
3.3 Contraction of the Highway Network

Theorem 5 *The highway network can be contracted in time $O(m + n)$.*

Proof: In order to determine the 2-core of G_1, we can use a simplified version of the more general algorithm presented in [2]. We remove nodes of degree one (and the incident edge) until all remaining nodes have degree at least two. (Note that the removal of such a node reduces the degree of another node so that a new degree-one node can emerge.) During this process, we manage a list that contains the roots of the attached trees. After the 2-core has



(a) The precise construction method (Section 3.1). As long as there is at least one active node, the search from passive nodes is continued. The shortest path from s_0 to v is found.



(b) The accelerated construction method (Section 3.2). Active nodes whose distance from s_0 is above a certain threshold are declared to be mavericks. When this threshold is reached, the search from passive nodes is no longer continued. The shortest path from s_0 to v is *not* found.

Figure 3.8: Comparison between different construction methods.

been determined, each root r initiates a traversal of the corresponding tree: each node u of the tree (except the root) creates a direct *shortcut* to the root, i.e., a directed edge (u, r) whose weight is equal to the length of the already existing path from u to r. For an example, refer to Fig. 3.9.

For each node that has degree two in the 2-core, but has not been assigned to a line yet, the line is traversed in an arbitrary direction until an endpoint, a node of degree greater than two, is encountered. Then, the line is traversed in the reverse direction: each node creates a shortcut to the already known endpoint. Finally, after the second endpoint has been reached, the line is traversed another time: each node creates a shortcut to the second endpoint. Note that there is also an undirected shortcut between both endpoints. Some special cases, namely cycles with and without an exit (a node of degree greater than two), have to be dealt with appropriately.[2] Figure 3.10 gives an example. □

Highway Hierarchy. The result of the contraction is the contracted highway network G'_1, which can be used as input for the next iteration of the construction procedure in order to obtain the next level of the highway hierarchy. A real world example of the construction process can be found in Section B.1.

[2]For directed graphs, we basically pretend that the graph was undirected and determine the components as usual. However, a shortcut from u to v is added only if there is a corresponding path from u to v in the directed graph. In addition, we add shortcuts from the endpoints to the nodes inside the component iff there is a corresponding path in the directed graph.

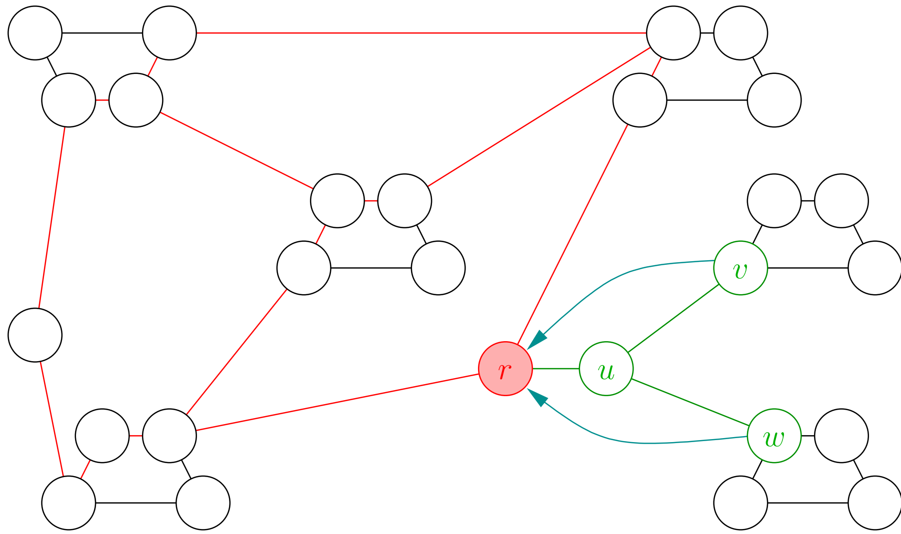


Figure 3.9: The 2-core of the highway network from Fig. 2.4 and an attached tree with shortcuts.

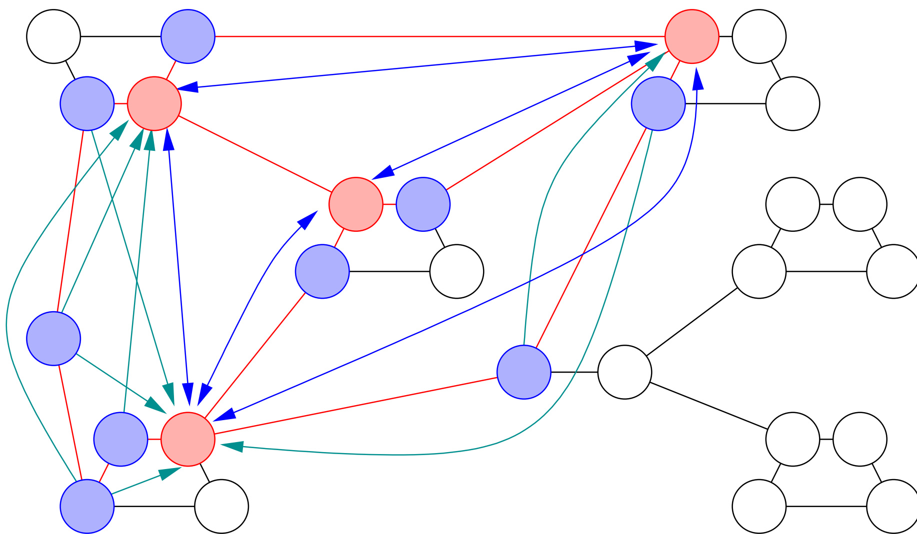


Figure 3.10: The 2-core of the highway network from Fig. 2.4 containing five lines. Both endpoints of a line are connected by an undirected shortcut. There is a directed shortcut from each inner node of a line to both endpoints.

Chapter 4

Query

In Section 4.1, we define the *highway hierarchy* as a multilevel graph and present an algorithm that finds for given nodes s and t a path in the multilevel graph that corresponds to a shortest s-t-path in the original graph. The algorithm is a modification of the bidirectional version of DIJKSTRA's algorithm, but *without aborting* when both search scopes meet. In Section 4.2, we explain how the multilevel graph can be represented in an one-level graph enhanced by some additional data. Furthermore, we indicate how the multilevel query algorithm can be adapted to the new situation. Note that the multilevel representation of Section 4.1 simplifies the proof of correctness, while the one-level representation of Section 4.2 is more suitable as a foundation of an implementation. In Section 4.3, we show that the naive abort-on-success criterion ("abort when both search scopes meet") would invalidate the correctness of our query algorithm. Therefore, we present a more sophisticated criterion that preserves the correctness.

4.1 Multilevel Query Algorithm

The *highway hierarchy* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ consists of the graphs $G_0, G_1, G_2, \ldots, G_L$, which are arranged in $L + 1$ levels. For each node $v \in V$ and each $i \in \{j \mid v \in V_j\}$, there is one copy of v, namely v_i, that belongs to level i of $\mathcal{G}$. Accordingly, there are several copies of an edge (u, v) when u and v belong to more than one common level. These edges, which connect two nodes in the same level, are called *horizontal* edges. Additionally, $\mathcal{G}$ contains a directed edge $(v_\ell, v_{\ell+1})$ for each pair $v_\ell \in V_\ell, v_{\ell+1} \in V_{\ell+1}$, where v_ℓ and $v_{\ell+1}$ are copies of the same node $v \in V$. These additional edges are called *vertical* and have weight 0. For each node v, not only one value $d_H(v)$ is known, but for each level $\ell < L$, there is a distance $d_H^\ell(v)$ from v to the H-closest node in the core G'_ℓ of level ℓ; if a node v does not belong to G'_ℓ, $d_H^\ell(v)$ is defined to be $+\infty$; furthermore, $d_H^L(v) := +\infty$. Correspondingly, we use the notation $\mathcal{N}^\ell(v)$ to refer to the set $\{v' \in V'_\ell \mid d(v, v') \leq d_H^\ell(v)\}$, which is the *neighbourhood* of v in the graph G'_ℓ. Note that the neighbourhood of a node that belongs to a component is unbounded, i.e., it contains all nodes of the core of the corresponding level. The same applies to $\mathcal{N}^L(v)$, for any v. Figure 4.1 gives an example of a highway hierarchy.

The *multilevel query algorithm* that works on $\mathcal{G}$ is a modification of the bidirectional version of DIJKSTRA's algorithm. The source and target nodes of an s-t query are the corresponding copies of s and t in level 0. For the time being, we omit the abort-on-success criterion, i.e., we do not abort when both search scopes meet, but continue until both searches

terminate; then, we consider all nodes that have been settled from both sides as meeting points and take the shortest path that has been found by this means. We do *not* have to apply the modifications presented in Appendix A, which ensure that only canonical shortest paths are found; this is required only during the construction process. We introduce two restrictions:

1. *No horizontal edge in level ℓ is relaxed that would leave the neighbourhood $\mathcal{N}^{\ell}(v^*)$ of the corresponding entrance point v^*.* Each node that belongs to the core and has been settled via a horizontal edge that leaves a component and each node that has been settled via a vertical edge is an *entrance point*. In addition, the source and the target nodes of the query are entrance points. The *corresponding entrance point* of a settled node v is the last entrance point on the path to v.

2. *Components are never entered using a horizontal edge.* An edge (u, v) *enters* a component if either u belongs to the core and v to a component or u belongs to a line and v to an attached tree. However, an edge from an attached tree to a line *leaves* the attached tree and does not rank among the edges that enter a component. Note that the endpoint(s) of a component do not belong to the component but to the core (or to the line in case of the root of a tree that is attached to a line).

Figure 4.2 is a schematic diagram of a multilevel query, Fig. 4.3 gives a two-level example.

Theorem 6 *For any given $s, t \in V$, the multilevel query algorithm finds the shortest path from s to t in G.*

Proof Idea: It is known that the bidirectional version of DIJKSTRA's algorithm works correctly. We have to show that the imposed restrictions do not affect the correctness. When Restriction 1 applies, it is always possible to switch to the next level using a vertical edge. Due to the definition of the highway network, it is guaranteed that the corresponding part of the shortest path which we are looking for can be found in the next level. A path from s that *enters* a component is not traversed due to Restriction 2. However, from the point of view of t, this path *leaves* the component so that the edge that has been skipped during the search from s can be relaxed in the reverse direction during the search from t. Hence, the path can be found in spite of Restriction 2. These arguments can be used in an inductive proof over the number of levels.

Proof: In a graph $G = (V, E)$, for two given subsets $S, T \subseteq V$ and an initial weight w for each node in S and each node in T, a *multi-source-multi-target* (MSMT) search determines the shortest path of all paths from a node $s \in S$ to a node $t \in T$, where the initial weights of s and t are added to the corresponding path lengths. An MSMT search is equivalent to a normal $\widehat{s}$-$\widehat{t}$-search in $\widehat{G} := (V \cup \{\widehat{s}, \widehat{t}\}, E \cup \{(\widehat{s}, s, w(s)) \mid s \in S\} \cup \{(\widehat{t}, t, w(t)) \mid t \in T\})$. ($\widehat{G}$ is generated by adding two pseudo-nodes $\widehat{s}$ and $\widehat{t}$ to G; each $s \in S$ can be reached via an edge from $\widehat{s}$, which is weighted by the initial weight of s, and, correspondingly, each $t \in T$ can be reached from $\widehat{t}$.)

Let $\mathcal{G}_\ell$ denote the highway hierarchy that consists only of the levels $\ell, \ell+1, \ldots, L$. Note that $\mathcal{G}_0 = \mathcal{G}$. Furthermore, let $\mathcal{G}'_\ell$ denote the highway hierarchy that consists of the core of level ℓ and the complete levels $\ell + 1, \ell + 2, \ldots, L$, i.e., $\mathcal{G}'_\ell$ is equal to $\mathcal{G}_\ell$ without the components in level ℓ. We prove the following more general statement by induction:

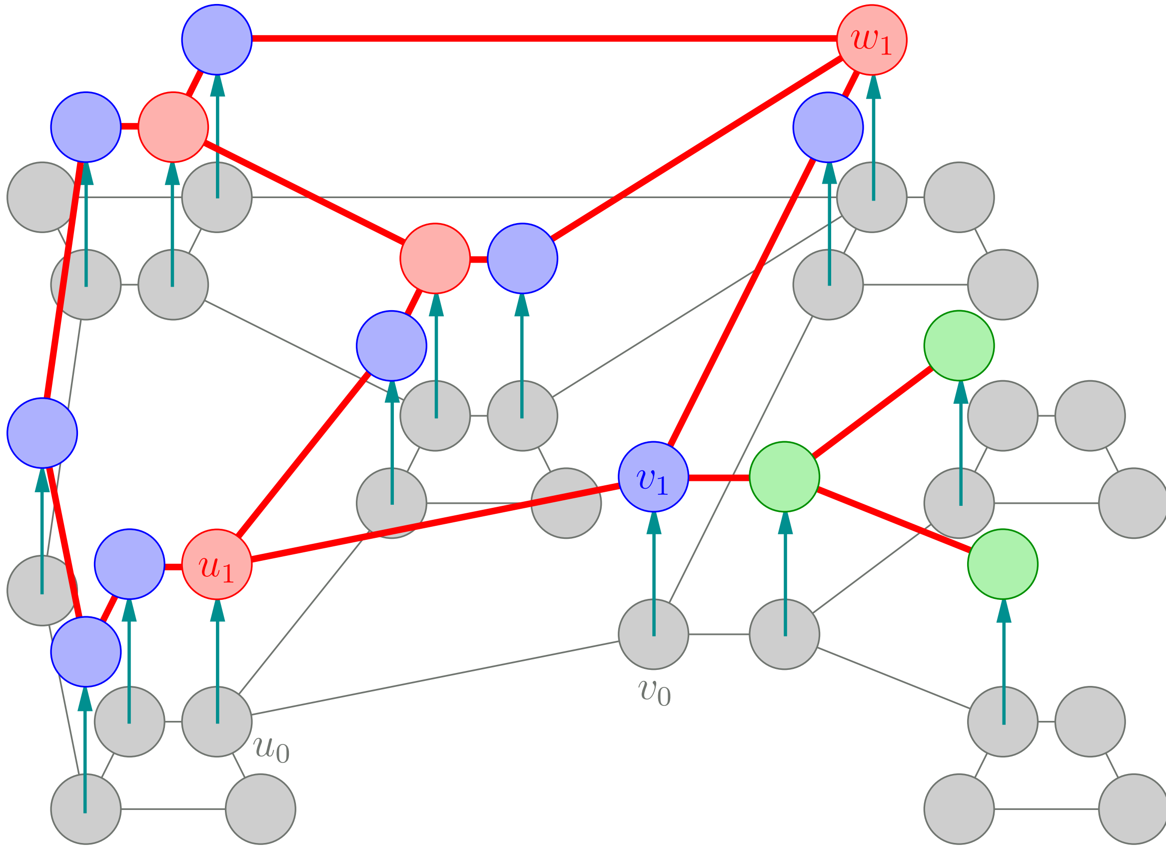


Figure 4.1: A highway hierarchy $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ of the graph given in Fig. 2.4 consisting of two levels: $G_0 = (V_0, E_0)$ ('level 0') and $G_1 = (V_1, E_1)$ ('level 1'). As in the previous examples, the neighbourhood size H is 3. Nodes and horizontal edges in level 0 are plotted in grey. There are directed vertical edges from level 0 to level 1. Horizontal edges in level 1 are red. The nodes in level 1 are coloured by type: tree nodes, line nodes, and nodes that belong to the core $G'_1 = (V'_1, E'_1)$. $u_0 \in V_0$ and $u_1 \in V_1$ are copies of the node $u \in V$. $(u_0, v_0) \in E_0$ and $(u_1, v_1) \in E_1$ are copies of the edge $(u, v) \in E$. The third closest node to u_1 in G'_1 is w_1. Hence, $d^1_H(u)$ would be equal to $d(u, w)$ if there was another level in the hierarchy. However, since there are only two levels (i.e. $L = 1$), $d^1_H(u)$ is defined to be $+\infty$.

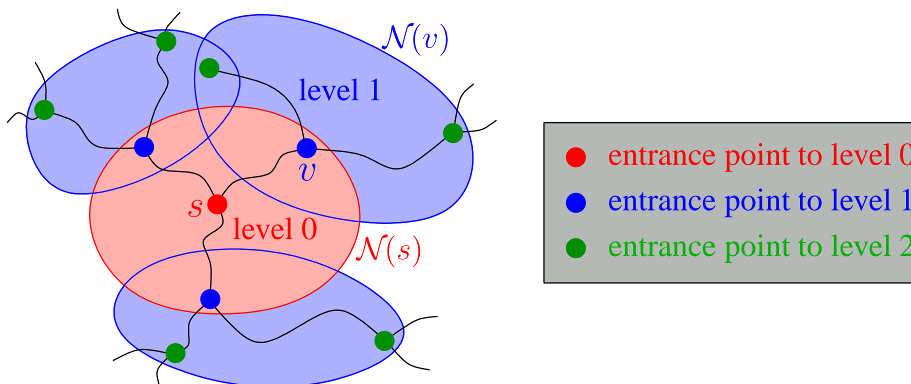
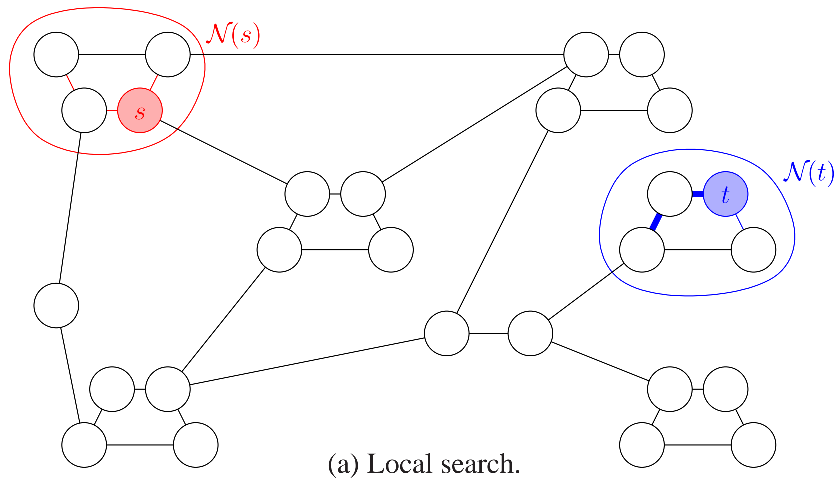
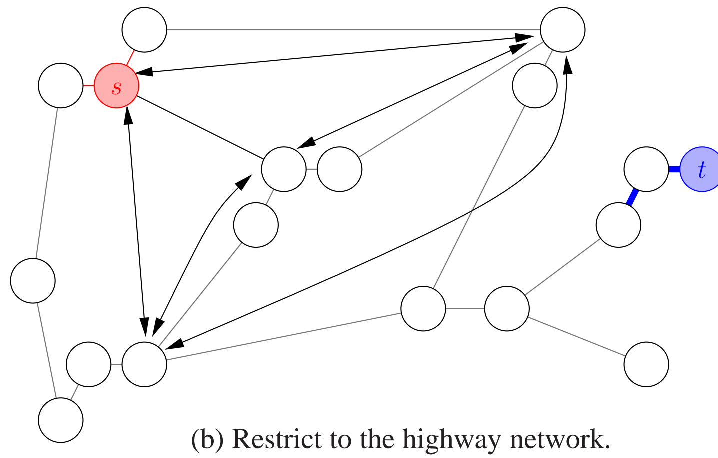


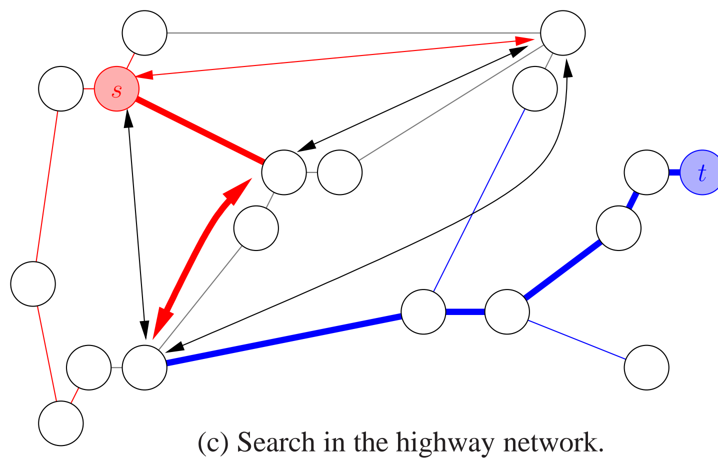
Figure 4.2: A schematic diagram of a multilevel query. Only the search started from the source node s is depicted.



(a) Local search.



(b) Restrict to the highway network.



(c) Search in the highway network.

Figure 4.3: An example of a two-level query. The *search space* from the source node s and from the target node t is represented; thick edges are part of the shortest path. After the local search is completed, i.e., the borders of $\mathcal{N}(s)$ and $\mathcal{N}(t)$ have been reached (a), we switch to the next level (b). The further search takes place in the highway network (c).

For any $\ell \in \{0, 1, \ldots, L\}$, any subsets $S, T \subseteq V_\ell$, and arbitrary initial weights of the elements of S and T, the multilevel query algorithm working on $\widehat{\mathcal{G}}_\ell$ finds the shortest path from $\widehat{s}$ to $\widehat{t}$.

Base Case. *First, we show that the multilevel query algorithm works correctly in $\mathcal{G}'_L$.* The additional edges in $\widehat{\mathcal{G}}'_L$ that leave $\widehat{s}$ or $\widehat{t}$ are interpreted as vertical edges. Hence, all elements in S and T are entrance points. Since all entrance points belong to level L, their neighbourhood is unbounded so that Restriction 1 never applies. Restriction 2 does not apply either because $\widehat{\mathcal{G}}'_L$ does not contain any component. Therefore, in this case, the multilevel query algorithm corresponds to the bidirectional version of DIJKSTRA's algorithm, which is known to be correct.

Induction Step 1. *We assume that the algorithm works correctly on $\mathcal{G}'_\ell$. We show that it also works correctly on $\mathcal{G}_\ell$ for any given sets $S, T \subseteq V_\ell$ and any initial weights.* We distinguish between two cases.

Case 1: there is a shortest path $P = \langle \widehat{s}, s, \ldots, t, \widehat{t} \rangle$ which does not include a node that belongs to $G'_\ell = (V'_\ell, E'_\ell)$. In this case, Restriction 1 never applies since the neighbourhoods of the entrance points s and t are unbounded and no other entrance point is encountered on P. If s and t belong to the same component, Restriction 2 does not apply either so that P is found. Otherwise, s and t belong to different components. They cannot belong to two different lines because two lines are connected only by the core. If either node belongs to a tree and the other one to a line, the tree has to be attached to the line. (Otherwise, the assumption of Case 1 that P does not intersect the core cannot be true.) Then, Restriction 2 prevents that the tree is entered, but it allows to leave the tree so that P can be found. If both nodes belong to two different trees, both trees must be attached to the same line. Both trees can be left so that both search scopes can meet within the line.

Case 2: all shortest paths from $\widehat{s}$ to $\widehat{t}$ pass through the core. Trees can be used only at the ends of a shortest path because they have only one endpoint. We could have a shortest path of the form "core – line – core" if both endpoints of the line belong to the path, but in this case, we can use the equivalent shortcut between both endpoints, which belongs to the core, so that it is sufficient to deal with shortest paths of the form "tree – line – core – line – tree", where the components are optional. In other words, after the components have been left from both sides, the 'middle' part of the shortest path belongs, without interruption, to the core. Let S' be the set $S \cap V'_\ell$ united with the set of all nodes that belong to the core and can be reached from $\widehat{s}$ via an edge that leaves a component. The initial weight of a node $s' \in S'$ is equal to the distance $d(\widehat{s}, s')$ in $\widehat{G}_\ell$. T' and the corresponding initial weights are defined accordingly. Paths from $\widehat{s}$ and $\widehat{t}$ to the elements of S' and T', respectively, consist only of edges that either stay inside the same component or leave a component. Hence, both restrictions do not apply so that for each $s' \in S'$ and each $t' \in T'$, the shortest paths from $\widehat{s}$ to s' and from $\widehat{t}$ to t' are found. Due to our assumption, the algorithm works correctly on $\mathcal{G}'_\ell$ for the sets S' and T'. Hence, the shortest path from $\widehat{s'}$ to $\widehat{t'}$ in $\widehat{\mathcal{G}}'_\ell$ is found. Obviously, this shortest path is also a shortest path from $\widehat{s}$ to $\widehat{t}$ in $\widehat{\mathcal{G}}_\ell$.

Induction Step 2. *We assume that the algorithm works correctly on $\mathcal{G}_\ell$. We show that it also works correctly on $\mathcal{G}'_{\ell-1}$ for any given sets $S, T \subseteq V'_{\ell-1}$ and any initial weights.*

Restriction 2 does not apply in level $\ell - 1$ since this level consists only of the core. Hence, if both search scopes meet in level $\ell - 1$ before Restriction 1 intervenes, the shortest path is found. Otherwise, all shortest paths pass through $\mathcal{G}_\ell$. Let us consider any shortest path $\langle \widehat{s}, s^*, \ldots, t^*, \widehat{t} \rangle$ from $\widehat{s}$ to $\widehat{t}$. Take the canonical shortest path P^* from s^* to t^* in $G'_{\ell-1}$. Then, $P = \langle \widehat{s}, P^*, \widehat{t} \rangle$ is also a shortest path from $\widehat{s}$ to $\widehat{t}$. Note that s^* and t^* are entrance points to level $\ell - 1$. Let $s^\dagger$ and $t^\dagger$ be the last nodes on $P^* = \langle s^*, \ldots, s^\dagger, s^\ddagger, \ldots, t^\ddagger, t^\dagger, \ldots, t^* \rangle$ that belong to the neighbourhood of s^* and t^*, respectively, i.e., $d(s^*, s^\ddagger) > d_H^{\ell-1}(s^*)$ and $d(t^*, t^\ddagger) > d_H^{\ell-1}(t^*)$. Note that $s^\dagger$ and $t^\dagger$ are entrance points to level ℓ. According to the definition of the highway network, the subpath $P^*|_{s^\dagger \to t^\dagger}$ belongs to G_ℓ, thus, it belongs to $\mathcal{G}_\ell$. Let $S' := V_\ell \cap \bigcup_{s \in S} \mathcal{N}^{\ell-1}(s)$ and for each $s' \in S'$, $w(s') = d(\widehat{s}, s')$. T' and the corresponding initial weights are defined accordingly. Note that $s^\dagger \in S'$ and $t^\dagger \in T'$. For any nodes $s' \in S'$ and $t' \in T'$, the shortest paths $\langle \widehat{s}, s, \ldots, s' \rangle$ and $\langle \widehat{t}, t, \ldots, t' \rangle$ from $\widehat{s}$ to s' and from $\widehat{t}$ to t', respectively, are found because Restriction 1 does not apply on shortest paths from s to s' and from t to t'. Due to the induction hypothesis, the algorithm works correctly on $\mathcal{G}_\ell$ for the sets S' and T'. Hence, a shortest path $Q' = \langle \widehat{s'}, s', \ldots, t', \widehat{t'} \rangle$ from $\widehat{s'}$ to $\widehat{t'}$ in $\widehat{\mathcal{G}}_\ell$ is found. We know that the path $Q = \langle \widehat{s'}, P^*|_{s^\dagger \to t^\dagger}, \widehat{t'} \rangle$ belongs to $\widehat{\mathcal{G}}_\ell$. The length $w(Q)$ of Q is equal to $d(\widehat{s'}, s^\dagger) + w(P|_{s^\dagger \to t^\dagger}) + d(t^\dagger, \widehat{t'}) = w(P|_{\widehat{s} \to s^\dagger}) + w(P|_{s^\dagger \to t^\dagger}) + w(P|_{t^\dagger \to \widehat{t}}) = w(P)$. Since Q' is a shortest path in $\widehat{\mathcal{G}}_\ell$, we have $w(Q') \leq w(Q) = w(P)$. The algorithm returns the path P', which corresponds to Q' when the edges $(\widehat{s'}, s')$ and $(\widehat{t'}, t')$ are replaced with shortest paths in $\widehat{G}'_{\ell-1}$ from $\widehat{s}$ to s' and from $\widehat{t}$ to t'. To sum up, P is known to be a shortest path from $\widehat{s}$ to $\widehat{t}$ in $\widehat{\mathcal{G}}'_{\ell-1}$ and the algorithm returns a path P', whose length $w(P') = w(Q')$ is less than or equal to the length of P. Hence, the algorithm returns a shortest path from $\widehat{s}$ to $\widehat{t}$ in $\widehat{\mathcal{G}}'_{\ell-1}$. □

4.2 Collapse of the Vertical Dimension

So far, we allow that several copies of the same node are reached. However, we can show that it is sufficient if at most one copy of a node is reached via a horizontal edge. We enhance our algorithm by adding the following rule:

> Let us assume that exactly one copy v_i of a node v in level i has been reached via a horizontal edge and another horizontal edge is about to be relaxed to another copy v_j of v. Then, only the copy with the smaller tentative distance should be inserted (or remain) in the priority queue or – if the tentative distances of v_i and v_j are equal – the copy in the lower level. (Note that if v_i has already been settled, then the tentative distance of v_j is greater than the tentative distance of v_i since we assume that no horizontal edge has weight 0. In this case, the edge leading to v_j is disregarded and v_i stays in its settled state.)

Lemma 4 *The above rule does not invalidate the correctness of the algorithm.*

Proof: Case 1. The tentative distances differ. Independently of the level, an edge that leads on a path Q from s to a copy of v cannot belong to a shortest path P from s to t if there is a shorter path Q' to another copy of v because, then, the replacement of the subpath $Q = P|_{s \to v}$ by Q' would yield a shorter path P'.

Case 2. The tentative distances are equal. It cannot be wrong to prefer the lower level copy in this case because the lower level is a superset of the higher level. Furthermore, a higher level can be reached from a lower level at any time, while the converse is not true. □

As a consequence of the above rule, it is not necessary to go upwards using a vertical edge if all horizontal edges could be relaxed without breaching Restriction 1.

Due to these observations, we can let the vertical dimension collapse. We can interpret the highway hierarchy $\mathcal{G}$ as one plain graph, i.e., there are no copies of the nodes distributed over several levels. Basically, this graph corresponds to the original graph G enhanced by shortcuts and some additional data: each edge (u, v) is assigned a maximum level $\ell(u, v)$, i.e., it belongs to the levels $0, 1, \ldots, \ell(u, v)$; each node v is assigned to at most one component $c(v)$; a component $c(v)$ belongs to a certain level $\ell(c(v))$, which is equal to the level its inner edges belong to. With this interpretation of $\mathcal{G}$ in mind, we can get another view of the multilevel query algorithm. Let us consider the search from the source node s; the search from the target t works analogously. Each reached node u is assigned a certain search level $\ell_s(u)$. On a shortest path $P_s = \langle s = s'_0, \ldots, s_1, \ldots, s'_1, \ldots, s_2, \ldots, s'_2, \ldots \rangle$, the search levels increase monotonically: the first nodes up to and including s_1 belong to search level 0, s_1 is the entrance point to level 1, all successors up to and including s_2 belong to level 1, s_2 is the entrance to level 2, and so on. If s_ℓ belongs to a component in level ℓ, then there is another entrance point s'_ℓ, namely, the first node on the path P_s that belongs to the core of level ℓ; otherwise, we have $s_\ell = s'_\ell$. The entrance point $s_{\ell+1}$ is the last node on the path P that belongs to $\mathcal{N}^\ell(s'_\ell)$. An edge (u, v) can be relaxed only if $\ell(u, v) \geq \ell_s(u)$. A shortest path P from s to t has the form $\langle s = s'_0, \ldots, s_1, \ldots, s'_1, \ldots, s_\ell, \ldots, s'_\ell, \ldots, t'_\ell, \ldots, t_\ell, \ldots, t'_1, \ldots, t_1, \ldots, t'_0 = t \rangle$, where s'_ℓ and t'_ℓ are omitted if both search scopes meet inside a component in level ℓ.

4.3 Abort-on-Success

In the bidirectional version of DIJKSTRA's algorithm, we can abort as soon as both search scopes meet, i.e., there is one node v that is settled in both search scopes. Then, the shortest path P from s to t does not necessarily consist of the shortest paths from s to v and from v to t, but it is well known that it is always ensured that the right meeting point v' has already been reached from both sides. The following lemma is a generalisation of this fact.

Lemma 5 *If $d_s + d_t$ is an upper bound for the length of the shortest path, all nodes whose distance from s is less than d_s have been settled in the search scope of s, and all nodes whose distance from t is less than d_t have been settled in the search scope of t, then there is a shortest path $P = \langle s = s_0, s_1, \ldots, s_i, v', t_j, \ldots, t_1, t_0 = t \rangle$ such that all nodes s_x and t_y have been settled in the search scope of s or t, respectively, and v' has been reached in both search scopes.*

Proof: Let $P = \langle s, \ldots, s', v', t', \ldots, t \rangle$ be a shortest path from s to t. The distance from s to the first node v' that is unsettled in the search scope of s is greater than or equal to d_s. The fact that the predecessor s' of v' has been settled implies that v' has been reached from s. From $w(P) \leq d_s + d_t$, we can conclude that $d(v', t) = w(P) - d(s, v') \leq d_t$. Hence, $d(t', t) < d_t$ because we exclude edges of weight 0. Therefore, t' has been settled in the search scope of t so that v' has been reached from t as well. □

Thus, if one node v is settled in both search scopes, the precondition of Lemma 5 is fulfilled for $d_s = d(s, v)$ and $d_t = d(t, v)$. Therefore, the shortest path can be determined by choosing the node v' as meeting point that has been reached in both search scopes and minimises $d(s, v') + d(t, v')$.

Unfortunately, we cannot adopt the abort-on-success criterion as it stands because, in general, the multilevel query algorithm does not fulfil the precondition of Lemma 5 as several edges are not relaxed due to Restriction 1 and 2 so that we cannot guarantee that all nodes up to a certain distance have been settled. In other words, if we aborted the search, it might happen that we miss the shortest path in the case that the shortest path contains one of the skipped edges. In contrast, we have already shown that the algorithm without abort is correct: if an edge that belongs to the shortest path is not relaxed (e.g. a component is not entered), then it is – at some point in time – relaxed from the other side (e.g. the component is left). Thus, our multilevel query algorithm has the chance to improve the tentative result after both search scopes have met. Nevertheless, instead of waiting until the search is completely finished, we can use a less conservative approach, which relies on the following very general and self-evident lemma.

Lemma 6 *After both search scopes have met, we can abort as soon as we can* exclude *that we would be able to improve the (tentative) result if we continued.*

The next lemma provides a version of Lemma 6 that is more specific to our situation.

Lemma 7 *After both search scopes have met, we can abort as soon as it is certain that for all edges $e = (u, v)$ that have been skipped at node u, the edge e will not be relaxed from v during the oncoming search.*

Lemma 7 trivially applies as soon as all skipped edges have been relaxed from the other side: obviously, this implies that they will not be relaxed in the future (since each edge is relaxed at most once). However, we can do better. A search level ℓ is said to be *finished* when there are no reached but unsettled nodes in level ℓ or below. Note that if the search level ℓ is finished, edges e in levels $\ell(e) \leq \ell$ cannot be relaxed any longer.

Lemma 8 *Let $\mathcal{E}_s$ denote the set of all horizontal edges that have been skipped during the search from s. $\mathcal{E}_t$ is defined accordingly. After both search scopes have met, we can abort as soon as the search from t has finished search level $\hat{\ell}_s := \max_{e \in \mathcal{E}_s} \ell(e)$ and the search from s has finished level $\hat{\ell}_t := \max_{e \in \mathcal{E}_t} \ell(e)$.*

Proof: When the search from t has finished search level $\hat{\ell}_s$, it is certain that no edge e that belongs to a level $\ell(e) \leq \hat{\ell}_s$ will be relaxed during the oncoming search, in particular, no edge that has been skipped during the search from s will be traversed by the backward search. The same argument applies to the reverse search direction. □

Improvements of the Abort-on-Success Criterion.

1. If level $\ell - 1$ is finished and a component in level ℓ has not been entered yet, it will not be entered in the future either because the only way to enter a component is via a vertical edge from the level below. If a component is entered, i.e., a node v that belongs to the component is settled, all edges that leave v are relaxed including the shortcut(s) to the

endpoint(s) of the component. These shortcuts correspond to the shortest paths from v to the endpoints and, thus, they contain the reverse edges of the horizontal level ℓ edges that enter the component. From these facts, we can conclude that when level $\ell - 1$ is finished, the reverse edge of a horizontal edge that enters a component in level ℓ either has already been relaxed or will not be relaxed at all. Therefore, when we skip an edge e that enters a component, it is sufficient to pretend that e belongs to level $\ell(e) - 1$ instead of $\ell(e)$, i.e., we can redefine $\hat{\ell}_s$ to be equal to $\max_{e \in \mathcal{E}_s} \ell'(e)$, where $\ell'(e) := \ell(e) - 1$, if e enters a component, and $\ell'(e) := \ell(e)$, otherwise.

   However, we have to deal with the special case that the root of an attached tree does not belong to the core, but to a line. When such a root is reached, its outgoing edges, including the shortcuts to the endpoints of the line, are relaxed immediately so that it is ensured that the preceding statements apply to this special case as well.

2. For each $x \in \{s, t\}$ and each level ℓ, we manage a value $\delta_{x,\ell} = \min_{(u,v) \in \mathcal{E}_{x,\ell}} d(x, v)$, where $\mathcal{E}_{x,\ell} := \{e \in \mathcal{E}_x \mid \ell'(e) = \ell\}$. $\delta_{x,\ell}$ is the minimum distance from x to the endpoint v of a level-ℓ'-edge (u, v) that has been skipped. Let $\bar{x} = t$ if $x = s$ and vice versa, and let $\bar{u}$ be the minimum element in the priority queue of $\bar{x}$. When a tentative shortest path P from s to t has been found and $\delta_{x,\ell} + d(\bar{x}, \bar{u}) \geq w(P)$, then we can ignore the fact that level-ℓ'-edges have been skipped in the search scope of x, i.e., we can pretend that $\mathcal{E}_{x,\ell}$ is empty, because it is certain that we will not find another s-t-path that contains an edge $e \in \mathcal{E}_{x,\ell}$ and is shorter than P.

3. Let us consider the search started from s. Between the meeting of both search scopes and the fulfilment of the advanced abort criterion, we do not have to relax edges from nodes v_ℓ that belong to a level $\ell > \hat{\ell}_t$, unless v_ℓ belongs to a component in level ℓ and $\ell = \hat{\ell}_t + 1$. Since the highway hierarchy does not contain any downward edges, the continuation of the search in the higher levels does not provide any chances of finding a shorter path that uses a reverse edge of a skipped edge in a lower level. The same argument applies to the search started from t.

Corollary 7 *For any given $s, t \in V$, the improved multilevel query algorithm finds the shortest path from s to t in G.*

Proof (Sketch): Follows from Theorem 6, Lemma 4, Lemma 8, and some remarks in Section 4.2 and 4.3. □

Chapter 5

Implementation

An exhaustive description of the implementation would go beyond the scope of this thesis so that we restrict ourselves to some important aspects. The program was written in C++ from scratch, not using any libraries, except for the C++ Standard Template Library. We make extensive use of *generic programming* techniques using C++'s template class mechanism. This applies to the graph data structure (Section 5.1.1), the priority queue (Section 5.1.2), and the implementation of DIJKSTRA's algorithm (Section 5.3). Our current implementation leaves room for reducing both running time and memory usage. The main program and the auxiliary programs[1] consist of 4 555 and 2 415 lines of code, respectively.

5.1 Data Structures

5.1.1 Graph Representation

The graph representation is based on the remarks in Section 4.2. We distinguish between a dynamic and a static version of our graph data structure. The dynamic version is used during the construction, while the graph is modified, particularly, due to the addition of shortcuts. Then, we switch to the static version, which is more compact and allows efficient traversals of the graph; it is used by the queries.

Static Graph. After the construction has been completed, the graph is static, i.e., there is no need of incorporating any changes while queries are processed. Therefore, we represent the graph in an *adjacency array*, which is very space-efficient and allows fast traversal of the graph. The undirected graph is represented as a bidirected graph, i.e., each undirected edge is represented as two directed edges. There are two arrays, one for the nodes (`Node`)[2] and one for the edges (`Edge`). The edges are grouped by the source node and store only the ID of the target node and the weight. Each node u 'knows' the index of its first outgoing edge in the edge array. Furthermore, it stores the level 0 neighbourhood radius $d^0_H(u)$. In order to deal with the additional requirements of the highway hierarchy, we extend this data structure in the following way. For each node u, all outgoing edges (u, v) are grouped by

[1] The auxiliary programs provide functionality to convert different graph file formats, draw graphs, and evaluate log files.

[2] Class names that will appear in Fig. 5.2 are given in parentheses.

the maximum level $\ell(u, v)$ they belong to. Between the node and the edge array, we insert another layer: for each node u and each level $\ell > 0$ that u belongs to, there is a *level node* u_ℓ (StaticLevelNode) that stores the value $d_H^\ell(v)$ and the index of the first outgoing edge (u, v) with the maximum level ℓ. All level nodes are stored in a single array. Each node u knows the index of the level node u_1. Figure 5.1 illustrates the graph representation. Furthermore, each node belongs to at most one component (Component), which belongs to a certain level.

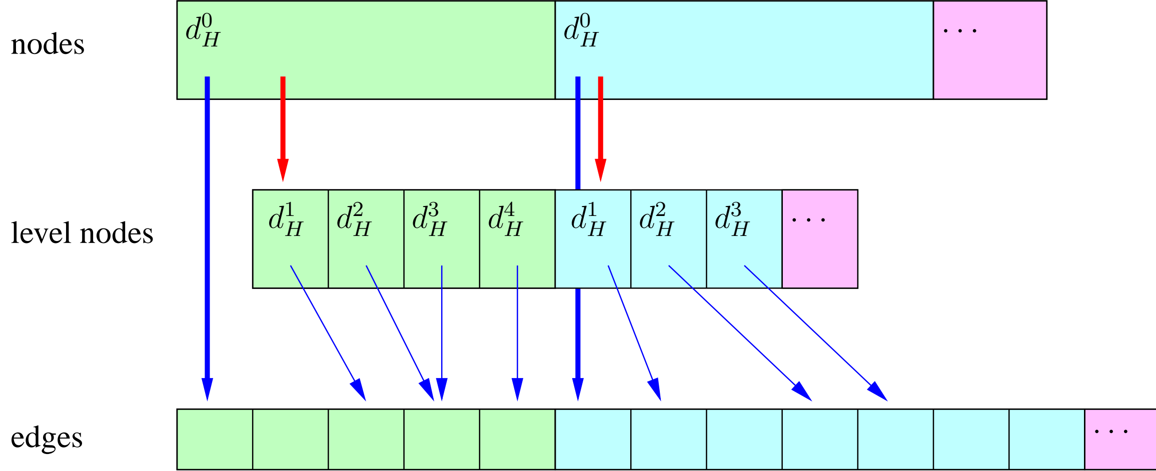


Figure 5.1: Adjacency array, extended by a level node layer.

Dynamic Graph. During the construction, the graph is modified in two respects: first, already existing edges are added to the next level of the highway hierarchy, i.e., the maximum level of the edges is increased; second, new edges, namely shortcuts, are added. We use a variant of the static graph data structure to handle this dynamic situation. When an edge (u, v) is promoted to the next level, u's edges are regrouped by a simple swap operation and the concerned level node updates the index of its first outgoing edge. The level nodes (DynamicLevelNodes) are not stored in an array, but in a linked list in order to allow the insertion of new level nodes. In an additional edge stack, we store new edges (CompleteLeveledEdge). Shortcuts from tree nodes to the root and from inner nodes to the endpoints of a line are added to this stack. Shortcuts between both endpoints of a line are *not* added to the stack. Instead, in the main edge array, we replace both edges that lead from the endpoints to the first respective inner node by the new shortcuts. The replaced edges are pushed on the stack. That way, we make sure that the stack contains only edges that are irrelevant to the further construction process.

When the construction has been completed, the dynamic graph is converted into the static graph. All level nodes are sorted into an array: a level node u_k precedes v_ℓ iff $u < v$ or $u = v$ and $k < \ell$. The additional edges are sorted by source node and level. They are merged with the original edge array in order to obtain one edge array that contains all edges. Figure 5.2 gives a UML class diagram[3] [3] of the concerned classes.

[3]In our UML class diagrams, all *dependencies* are stereotyped as bind, i.e., they are used to represent the instantiation of a template class with actual parameters. However, for the sake of clarity, we omit the keyword <<bind>> in each of these cases. Furthermore, we enhance the concept of *generalisation* by providing template parameters for the superclass: we extend the instantiation of a template class without explicitly representing the instantiation.

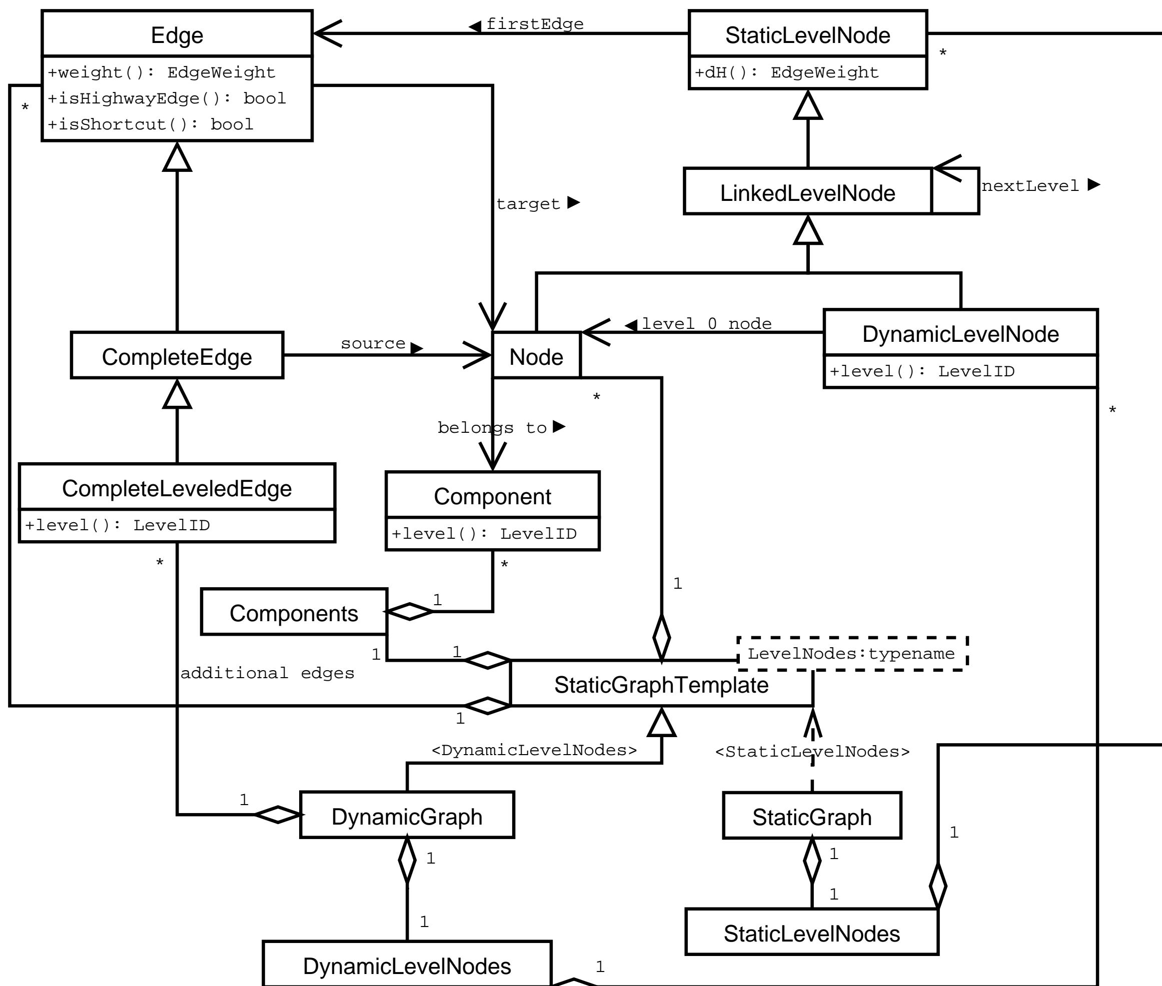


Figure 5.2: A UML class diagram of the graph representation.

5.1.2 Priority Queue

The priority queue is implemented as a binary heap (`BinaryHeap`) (e.g. [7]), which is realised as a template class. Its elements (`BinaryHeapElement`) are composed of a *key* and associated *data*. The key specifies the priority of an element, i.e., a *deleteMin* operation returns the element with the smallest key. The data object contains application-specific attributes, e.g., the index of the parent in the shortest path tree. A node (`Node`) has two pointers to elements in the priority queue, one for the forward search and one for the backward search. Thus, we separate the data that is related to the search process from the representation of the static graph: Initially, each node has only two unused pointers to binary heap elements. Not until a node is reached during a search, it is added to the priority queue and enhanced by additional data that are stored inside the binary heap element. This approach is reasonable with respect to the space consumption because only a small fraction of the nodes is reached during a multilevel query.

We need several variants of the priority queue, realised by appropriate instantiations and extensions of the binary heap template class: a normal variant (`NormalPQueue`)

that is used by DIJKSTRA's algorithm, a variant used by the multilevel query algorithm (`HwyPQueue`), and a variant used during the construction (`ConstrPQueue`). For the first two variants, we use the tentative distance (of type `EdgeWeight`) as key. For the construction, we need a priority queue that has the *FIFO property* (`FIFOBinaryHeap`) (see Section A.1), i.e., if there is more than one minimum element, then the older element is removed first. For this purpose, we can extend the binary heap using the tuple (tentative distance, timestamp) as key (see Section A.2). Each variant of the binary heap has its respective elements (`NormalPQElement`, `HwyPQElement`, `ConstrPQElement`). The data object (`PQueueNode`) that belongs to a normal priority queue element contains the indices of the parent node in the shortest path tree and of the edge from the parent. The data objects that belong to the other two priority queue elements are extensions of `PQueueNode`. In case of the multilevel query, the data object (`PQueueNodeHwySearch`) contains, in addition, the search level and the distance to the border of the neighbourhood of the current entrance point. In case of the construction, the data object (`PQueueNodeConstruction`) contains the slack and values that are required to test the abort criterion. Figure 5.3 gives an overview in the form of a UML diagram.

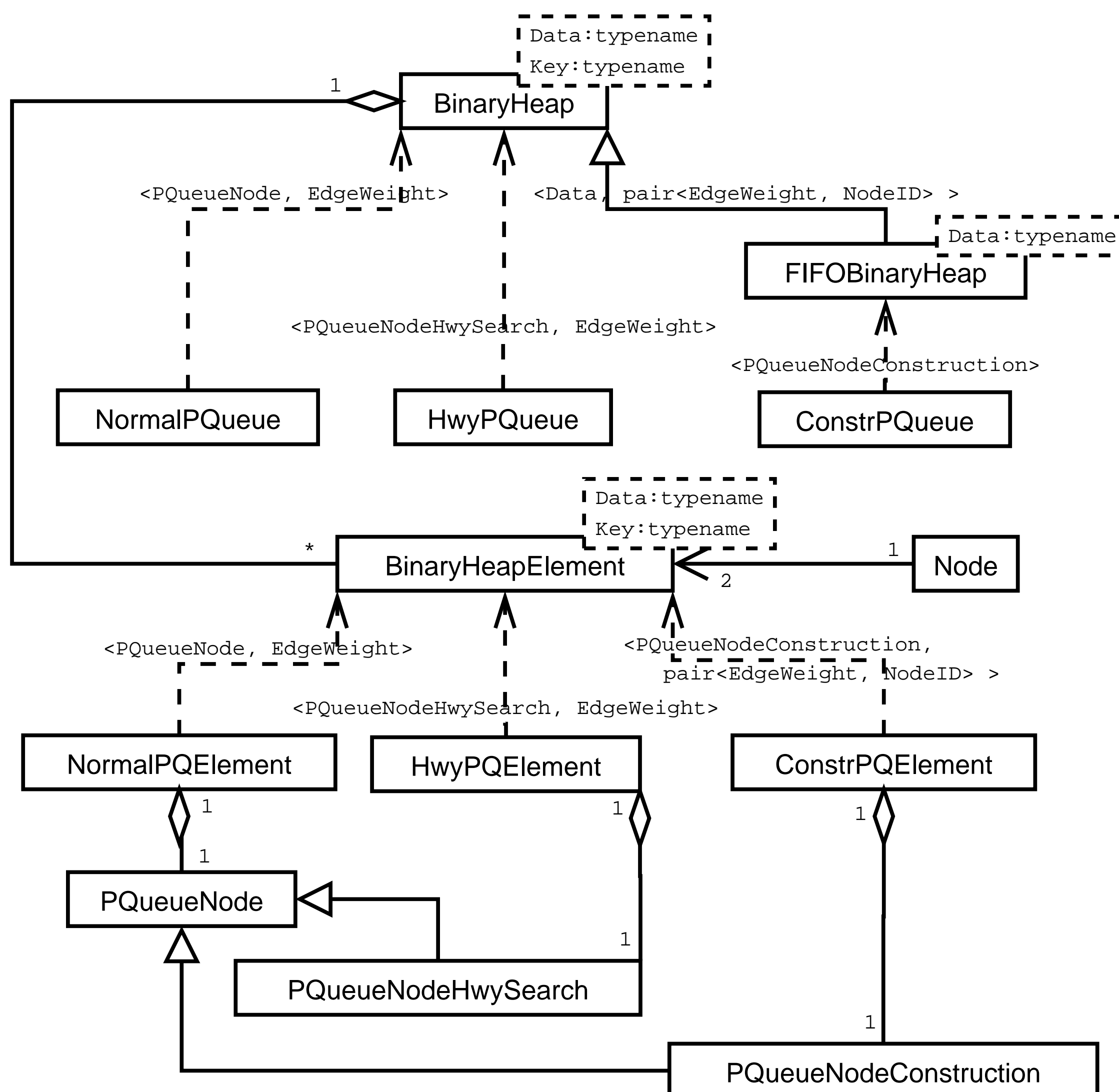


Figure 5.3: A UML class diagram of the priority queue and related classes.

5.2 Construction

Exact Arithmetic. In order to guarantee the uniqueness of the canonical shortest paths, it is important to exclude arithmetic inaccuracies. Therefore, the edge weights are integers between 0 and 2^{53} and are stored in double-precision floating-point numbers, which allow exact arithmetic in this range. If necessary, given edge weights are mapped to this range in such a way that it is ensured that the length of a shortest path never exceeds 2^{53}.

Initial Step. For each node $s_0 \in V$, we compute and store the value $d_H(s_0)$. This can be easily done by a DIJKSTRA search from each node s_0 that is aborted as soon as H nodes have been settled.

Phase 1. The abort criterion presented in Section 3.1 can be refined in the following way:

> When a node p is settled using the path $P' = \langle s_0, s_1, \ldots, p \rangle$, then p's state is set to passive if $p \notin \mathcal{N}(s_1)$ and $|P' \cap \mathcal{N}(s_1) \cap \mathcal{N}(p)| \leq 1$.

Lemma 9 *The refined abort criterion does not invalidate Theorem 1.*

Proof: In the proof of Theorem 1, in order to obtain a contradiction, we had to show that there were at least two nodes in $\mathcal{N}(s_1) \cap \mathcal{N}(p)$, namely u and v. Due to the refined abort criterion, we now have to prove that, under the same assumptions, there are at least two nodes in $\mathcal{N}(s_1) \cap \mathcal{N}(p)$ that belong to $P|_{s_0 \to p} = \langle s_0, s_1, \ldots, p \rangle$ as well. We know that u is not a predecessor of s_0 (due to the choice of s_0). Furthermore, v cannot be a successor of p. (If v was a successor of p, Lemma 1 would yield $p \in \mathcal{N}(s_1)$ since $v \in \mathcal{N}(s_1)$. This would be a contradiction to the first part of the refined abort criterion ($p \notin \mathcal{N}(s_1)$).) Hence, $u, v \in P|_{s_0 \to p}$. □

This criterion, in turn, can be reformulated to obtain:

> When a node p is settled using the path $\langle s_0, s_1, \ldots, \overline{u}, \overline{v}, w, \ldots, p \rangle$, where $d(s_1, \overline{v}) \leq d_H(s_1) < d(s_1, w)$, then p's state is set to passive if p is a successor of $\overline{v}$ and $d(\overline{u}, p) > d_H(p)$.

Lemma 10 ($p \notin \mathcal{N}(s_1)$ *and* $|P|_{s_0 \to p} \cap \mathcal{N}(s_1) \cap \mathcal{N}(p)| \leq 1$) $\Leftrightarrow$ (p *is a successor of* $\overline{v}$ *and* $d(\overline{u}, p) > d_H(p)$)*, i.e., both formulations of the refined abort criterion are equivalent.*

Proof: It is easy to see that "$p \notin \mathcal{N}(s_1)$" and "$p$ is a successor of $\overline{v}$" are equivalent. We still have to prove that $|P|_{s_0 \to p} \cap \mathcal{N}(s_1) \cap \mathcal{N}(p)| \leq 1 \Leftrightarrow d(\overline{u}, p) > d_H(p))$, provided that $p \notin \mathcal{N}(s_1)$ [*] is true.

$\Leftarrow$) $d(\overline{u}, p) > d_H(p)$ implies $\overline{u} \notin \mathcal{N}(p)$. Furthermore, we have $w \notin \mathcal{N}(s_1)$. Hence, we obtain (by Lemma 1) $\forall x \in P|_{s_0 \to \overline{u}} : x \notin \mathcal{N}(p)$ (since p is a successor of $\overline{u}$ (due to [*])), and $\forall x \in P|_{w \to p} : x \notin \mathcal{N}(s_1)$. Thus, only $\overline{v}$ can belong to $P|_{s_0 \to p} \cap \mathcal{N}(s_1) \cap \mathcal{N}(p)$.

$\Rightarrow$) We prove the contraposition. $d(\overline{u}, p) \leq d_H(p)$ implies $\overline{u} \in \mathcal{N}(p)$. Furthermore, we have $\overline{v} \in \mathcal{N}(s_1)$. Lemma 3 yields $\overline{u}, \overline{v} \in \mathcal{N}(s_1) \cap \mathcal{N}(p)$. Due to its definition, $\overline{u}$ cannot be a predecessor of s_0. Furthermore, $\overline{v}$ cannot be a successor of p. (Otherwise, since $\overline{v} \in \mathcal{N}(s_1)$, Lemma 1 would yield $p \in \mathcal{N}(s_1)$, which would be a contradiction to [*].) Hence, $\overline{u}, \overline{v} \in P|_{s_0 \to p} \cap \mathcal{N}(s_1) \cap \mathcal{N}(p)$. □

This version of the criterion can be tested efficiently: In order to find the first node w outside $\mathcal{N}(s_1)$, the distance to the border of the neighbourhood of s_1 is set to $d_H(s_1)$ at the node s_1; each descendant y of s_1 adopts the distance to the border from its parent x in B and decreases it by the weight of the edge (x, y). w is found as soon as this value gets negative. In order to be able to compute $d(\overline{u}, p)$, each descendant of $\overline{v}$ adopts the value $d(s_0, \overline{u})$ from its parent. Since the distance from s_0 to the current node is always known, we can use the formula $d(\overline{u}, p) = d(s_0, p) - d(s_0, \overline{u})$ to obtain the required value.

Corollary 8 *The refined abort criterion preserves the correctness of the construction process and can be tested in constant time for each node that is settled.*

Note that the refined abort criterion can increase the growth of B in some cases because $|\mathcal{N}(s_1) \cap \mathcal{N}(p)| \leq 1$ only implies $|P|_{s_0 \to p} \cap \mathcal{N}(s_1) \cap \mathcal{N}(p)| \leq 1$, but not $p \notin \mathcal{N}(s_1)$, i.e., sometimes the refined abort criterion is not fulfilled when the original criterion is fulfilled. However, the following lemma states that this overhead is limited.

Lemma 11 *If the original abort criterion is fulfilled for some node p while the refined criterion is not, and q is a direct successor of p on a shortest path, then the refined abort criterion is fulfilled for q.*

Proof: We assume that $|\mathcal{N}(s_1) \cap \mathcal{N}(p)| \leq 1$, but $p \in \mathcal{N}(s_1)$, i.e., the original criterion is fulfilled, but the refined one is not. Let q be an arbitrary direct successor of p on a shortest path. In order to obtain a contradiction, let us assume that $q \in \mathcal{N}(s_1)$. Then, Lemma 2 yields $q \in \mathcal{N}(p)$. Hence, $p, q \in \mathcal{N}(s_1) \cap \mathcal{N}(p)$, which is a contradiction to $|\mathcal{N}(s_1) \cap \mathcal{N}(p)| \leq 1$. Therefore, we can conclude that $q \notin \mathcal{N}(s_1)$. Furthermore, $|\mathcal{N}(s_1) \cap \mathcal{N}(p)| \leq 1$ implies $|P|_{s_0 \to p} \cap \mathcal{N}(s_1) \cap \mathcal{N}(p)| \leq 1$. In order to obtain a contradiction, we assume that $|P|_{s_0 \to q} \cap \mathcal{N}(s_1) \cap \mathcal{N}(q)| > 1$. Since we already know that $q \notin \mathcal{N}(s_1)$, there has to be a node x on $P|_{s_0 \to p}$ that belongs to $\mathcal{N}(q)$, but not to $\mathcal{N}(p)$. This is a contradiction to Lemma 2. Thus, we have $|P|_{s_0 \to q} \cap \mathcal{N}(s_1) \cap \mathcal{N}(q)| \leq 1$ so that the refined abort criterion is fulfilled for q. □

During the search of Phase 1, a list of all leaves of B is managed: when a node is settled, it is added to the list and its parent is removed from it. This list is the starting point for Phase 2.

Phase 2. The implementation of Phase 2 is straightforward and based on the detailed description in Section 3.1.

Final Step. After all nodes s_0 have been processed, G_1 is made bidirected by adding edges (v, u) to E_1 if (u, v) already belongs to E_1.

Contraction. For each node u of degree one, we determine its only (unused) edge, which leads to its parent p in the tree. p is added to the list of roots (which is initially empty) and u is removed from the list (if applicable). The edge (p, u) is marked as used, and the degrees of p and u are decremented. If the remaining degree of p is one, these steps are applied recursively to p. After all nodes of degree one have been processed, each root r initiates a traversal of the corresponding tree and broadcasts its own ID and a unique ID for the tree: each node u of the tree (except the root) stores the tree ID and creates a direct *shortcut* to the root, i.e., a directed edge (u, r) whose weight is equal to the length of the already existing path from u to r. The implementation of the line contraction is straightforward.

5.3 Query

We provide a template class that implements several versions of DIJKSTRA's algorithm. By instantiating it with appropriate template parameters, it can be used for the normal version of DIJKSTRA's algorithm, the bidirectional version of DIJKSTRA's algorithm, the computation of $d_H(\cdot)$, the construction, and the multilevel query. The implementation of the multilevel query algorithm is based on the remarks in Section 4.2. The compliance with both restrictions (cp. Section 4.1) is ensured in the following way:

1. *No horizontal edge in level ℓ is relaxed that would leave the neighbourhood $\mathcal{N}^\ell(v^*)$ of the corresponding entrance point v^*.*

   Each node that has been reached during the search knows its search level and the distance to the border of the neighbourhood of the current entrance point (cp. Section 5.1.2). Initially, the search levels of s and t are set to 0 and the distance-to-border values to $d_H^0(s)$ and $d_H^0(t)$, respectively. When a node v is reached, it adopts the search level from its parent p and the distance value minus the weight of (p, v). If this value gets negative, the neighbourhood of the corresponding entrance point would be left. Therefore, the search level of v is incremented, i.e., we try to switch to the next level ℓ. If the maximum level of (p, v) is less than the new search level, this attempt fails: the edge cannot be relaxed. Otherwise, p is the entrance point to the new search level: the distance-to-border value of v is set to $d_H^\ell(p) - w(p, v)$.

   If a node p belongs to the core and has been settled via a horizontal edge that leaves a component, it is an entrance point. The distance-to-border values of all its children v are set to $d_H^\ell(p) - w(p, v)$.

2. *Components are never entered using a horizontal edge.*

   Let c be a component that belongs to level ℓ, and (u, v) an edge that enters the component c. Thus, (u, v) belongs to level ℓ as well. During the construction process, when the component c is contracted, the level of the directed edge (u, v) (that enters c) is decremented by one, while the level of the reverse edge (v, u) (that leaves c) remains unchanged – this distinction is possible because we use a bidirected graph representation. By this means, there is no need of explicit checks during the query, but Restriction 2 is respected automatically: a component in level ℓ cannot be entered using a horizontal edge since all crucial edges have been downgraded so that they do not appear in level ℓ.

The implementation comprises the basic abort-on-success criterion and all three improvements as described in Section 4.3.

Each node in the shortest path tree stores a pointer to its parent. When we follow these pointers starting from the optimal meeting point of both search scopes until we reach s and t, we can reconstruct the shortest path. In order to do so, we have to expand shortcut edges so that we obtain the complete path in the original graph. We provide a basic recursive routine to solve this problem: When an endpoint u of a shortcut (u, v) is discovered, all outgoing edges (u, x) of u are scanned in order to find the right one that leads inside a corresponding line to v. We can easily check whether x is the right node due to the fact that all nodes inside a line have shortcuts to both endpoints of the line. Hence, if there is a shortcut (x, v) such that $w(u, x) + w(x, v) = w(u, v)$, then x is the right node. If the edge (u, x) is a shortcut as well, we apply the expansion routine recursively. Then, we go on to look for the next node y on a line that eventually leads to v.

Chapter 6

Experiments

We conducted extensive experiments in order to evaluate the performance of our approach. In total, the experiments took more than 1 190 hours of computing time. 3 930 131 909 670 *deleteMin* operations were logged.

6.1 Environment and Instances

Environment. The experiments were done on a 64-bit machine with 8 GB main memory and 1 MB L2 cache, using one out of four AMD Opteron processors clocked at 2.2 GHz, running SuSE Linux (kernel 2.6.5). The program was compiled by the GNU C++ compiler 3.3.3 using optimisation level 3.

Instances. Basically, we deal with two test instances, namely, the road networks of the United States of America and of Western Europe (Fig. 6.1 and 6.2). The former represents the road network of the District of Columbia and the 48 contiguous states (all but Alaska and Hawaii). It was obtained from the TIGER/Line Files [40] by extracting the relevant data of all counties and merging them. The latter comprises 14 European countries, namely, Austria, Belgium, Denmark, France, Germany, Italy, Luxembourg, the Netherlands, Norway, Portugal, Spain, Sweden, Switzerland, and the UK. The data has been made available for scientific use by the company PTV AG. In some cases, we restrict our experiments to the German road network. In all cases, as we deal with undirected graphs, we ignored the restrictions caused by one-way streets.

The original graphs contain for each edge a length and a road category. In the USA, there is the distinction between

- primary highways with limited access (e.g. interstate highways),
- primary roads without limited access (e.g. US highways),
- secondary and connecting roads (e.g. state highways), and
- local, neighbourhood, and rural roads.

In the European road network, there are 13 different categories. Each of these categories belongs to one out of four supercategories, namely

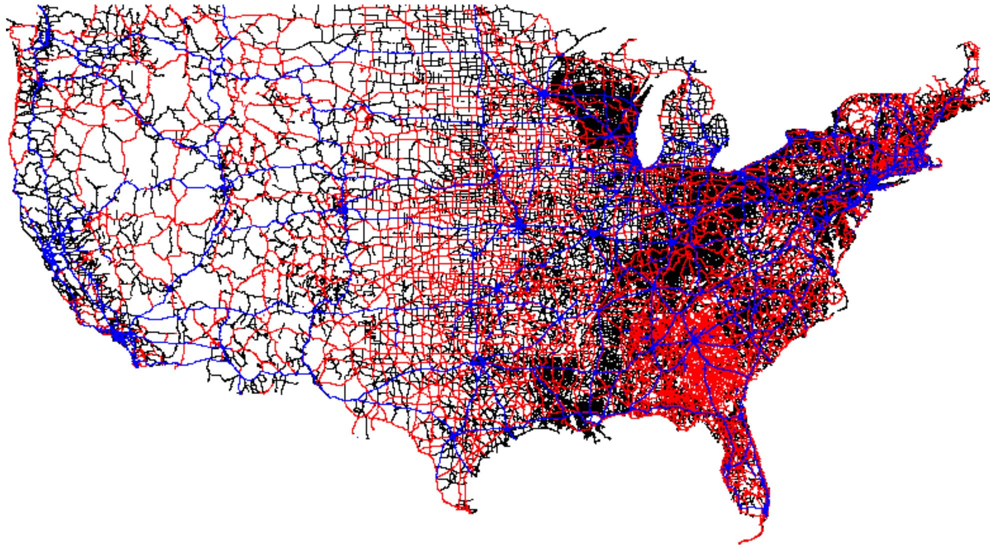


Figure 6.1: Road network of the USA. The colours indicate the road category: primary highway with limited access, primary road without limited access, secondary and connecting road. The slowest category (local, neighbourhood, and rural road) has been omitted in this figure.

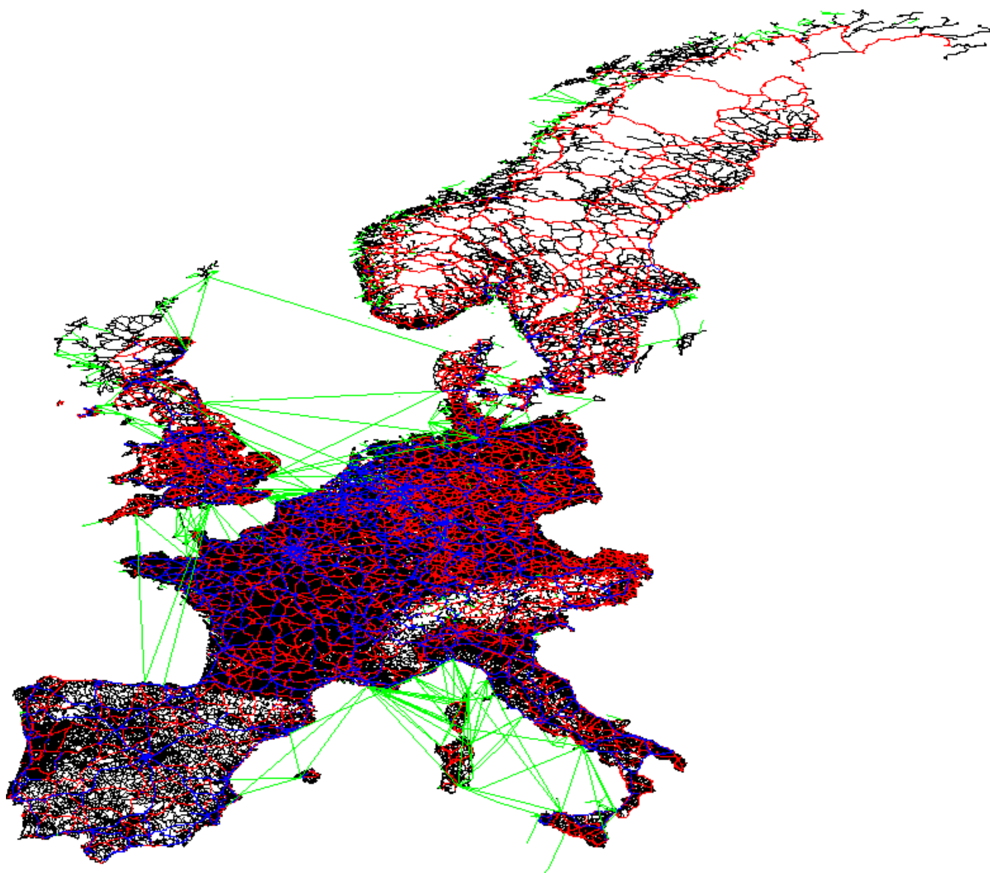


Figure 6.2: Road network of Western Europe. The colours indicate the road category: ferry, motorway, national road, regional road. The slowest category (urban street) has been omitted in this figure.

- motorway,
- national road,
- regional or local road, or
- urban street.

We assign average speeds to the road categories, compute for each edge e the average travel time, and use it as the weight of e. In addition, our European graph contains edges that represent ferry connections. For these edges, the average travel times are already given in the input so that we can adopt them as edge weights. Table 6.1 summarises important properties of the used road networks and the key results of the experiments.

Table 6.1: Overview of the used road networks and key results. The parameter H is used iteratively until the construction leads to an empty highway network. We provide average values for 10 000 queries, where the source and target nodes are chosen randomly. 'Speedup' refers to a comparison with DIJKSTRA's algorithm[1]. 'Efficiency' [14] denotes the number of nodes that belong to the computed shortest paths divided by the number of nodes that are settled by the multilevel query algorithm. For Germany, we give the memory usage on a 32-bit machine in parentheses.

		USA	Europe	Germany
input	#nodes	24 278 285	18 029 721	4 345 567
	#edges	29 106 596	22 217 686	5 446 916
	#degree 2 nodes	7 316 573	2 375 778	604 540
	#road categories	4	13	13
parameters	average speeds [km/h]	40–100	10–130	10–130
	H	225	125	100
construction	CPU time [h]	4.3	2.7	0.5
	#levels	7	11	11
query	CPU time [ms]	7.04	7.38	5.30
	#settled nodes	3 912	4 065	3 286
	speedup (CPU time)	2 654	2 645	680
	speedup (#settled nodes)	3 033	2 187	658
	efficiency	113%	34%	13%
	main memory usage [MB]	2 443	1 850	466 (346)

6.2 Parameters

Fast vs. Precise Construction. During various experiments, we came to the conclusion that it is a good idea *not* to take a fixed maverick factor f for all levels of the construction process, but to start with a low value (i.e. fast construction) and increase it level by level (i.e. more precise construction). Table 6.2 contains the construction time and the average query time for several sequences of maverick factors. In addition to the criterion $d(s_0, v) > f \cdot d_H(s_0)$ presented in Section 3.2, we considered to use $w(u, v) > f \cdot d_H(s_0)$ as maverick criterion, where u is the (tentative) parent of v in the shortest path tree, i.e., the decisive factor is the length of the incoming edge but not the distance from the source node. Good results were obtained for $d(s_0, v) > f \cdot d_H(s_0)$ as maverick criterion using $0, 2, 4, 6, \ldots$ as sequence of maverick factors. These parameters were used for all experiments that follow.

[1]The averages for DIJKSTRA's algorithm are based on only 1 000 queries.

Table 6.2: Fast vs. precise construction: maverick criterion $x > f \cdot d_H(s_0)$. The first group of experiments starts with the fastest construction method ($f = 0$) and switches to a fixed $f > 0$. In the second and third group, f is increased by adding 2 and multiplying by 2, respectively. If the results for one test instance did not show promise, the other test instance was skipped (resulting in blank entries in the table). A very good choice of f and x is **marked**.

		Europe		USA	
f	x	constr [h]	query [ms]	constr [h]	query [ms]
0 0 4	$d(s,v)$	2.2	9.57		
0 0 4	$w(u,v)$	2.7	8.25		
0 0 8	$w(u,v)$	2.9	8.05		
0 0 16	$w(u,v)$	3.2	7.88		
0 0 32	$w(u,v)$	3.9	7.73		
0 0 64	$w(u,v)$	5.1	7.47		
0 0 4 6 8 10	$w(u,v)$	2.7	7.76		
0 2 4 6 8 10	**d(s, v)**	**2.3**	**7.91**	**4.2**	**7.48**
0 2 4 6 8 10	$w(u,v)$			5.9	7.14
0 0 4 8 16	$d(s,v)$	2.5	7.79		
0 0 4 8 16	$w(u,v)$	2.7	7.34		
0 1 2 4 8	$d(s,v)$	2.2	9.21	3.8	8.06
0 1 2 4 8	$w(u,v)$	3.2	7.14	5.6	7.17
1 2 4 8 16	$w(u,v)$	> 12			

Best Neighbourhood Sizes. For two levels ℓ and $\ell + 1$ of a highway hierarchy, the *shrinking factor* is defined as the ratio between $|E'_\ell|$ and $|E'_{\ell+1}|$. In our experiments, we observed that the highway hierarchies of the USA and Europe were almost *self-similar* in the sense that the shrinking factor remained nearly unchanged from level to level when we used the same neighbourhood size H for all levels. We kept this approach and applied the same H iteratively until the construction led to an empty highway network. Table 6.3 shows our results for various values of H.

Table 6.3: Choice of good neighbourhood sizes. For different neighbourhood sizes H, we compare the construction time, the number of levels in the highway hierarchy, and, for queries between random source and target nodes, the average number of settled nodes and the average query time. The neighbourhood size that has been chosen for further experiments and the minima in the query columns are marked.

USA				
	Construction		Query	
H	t[h]	#level	#nodes	t[ms]
100	2.3	18	5748	13.02
150	3.0	11	4142	8.22
175	3.4	9	3952	7.51
200	3.8	8	**3895**	7.19
225	4.3	7	3912	**7.04**
250	4.7	7	3955	**7.04**
300	5.6	6	4109	7.05
400	7.5	6	4517	7.25

Europe				
	Construction		Query	
H	t[h]	#level	#nodes	t[ms]
50	1.8	30	7476	19.37
75	1.9	17	4581	9.81
100	2.3	13	4103	8.20
125	2.7	11	**4065**	7.38
150	3.1	10	4119	**7.15**
175	3.6	9	4256	7.19
200	4.0	9	4413	7.19
300	6.1	7	4962	7.86

Figure 6.3 demonstrates the shrinking process for Europe. Provided that the neighbourhood size is sufficiently large, we observe an almost constant shrinking factor for most levels (which appears as a straight line due to the logarithmic scale of the y-axis). The greater the neighbourhood size, the greater the shrinking factor. The first iteration (level 0→1) and the last few iterations show a different behaviour: in the first iteration, the construction works very well due to the characteristics of the real world road network (there are many trees and lines that can be contracted); in the last iterations, the highway network collapses, i.e., it shrinks very fast, because nodes that are close to the border of the network usually do not belong to the next level of the highway hierarchy, and when the network gets small, almost all nodes are close to the border. Figure 6.4 shows a similar shrinking process for the road network of the USA.

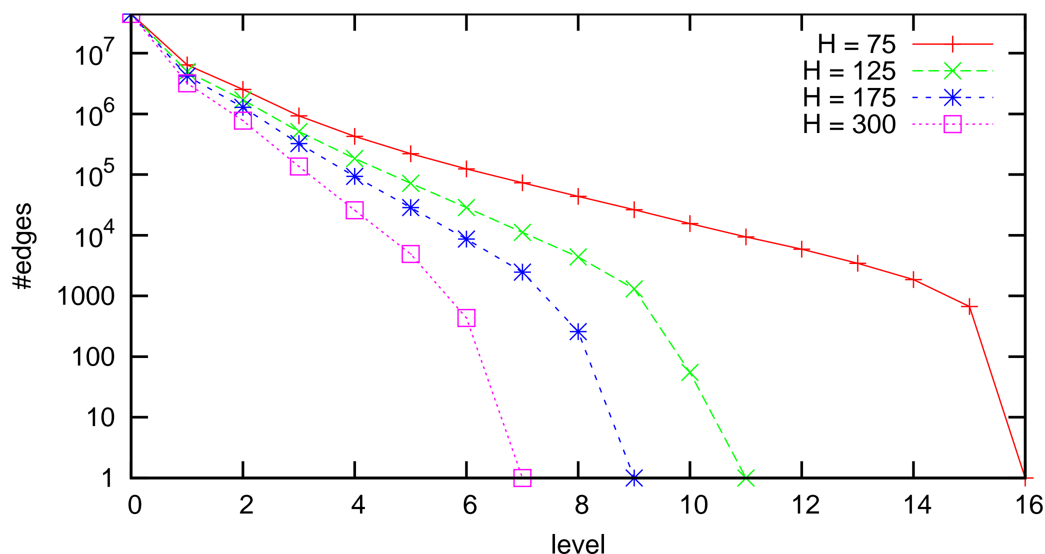


Figure 6.3: Shrinking of the highway networks of Europe. For different neighbourhood sizes H and for each level ℓ, we plot $|E'_\ell|$, i.e., the number of edges that belong to the core of level ℓ.

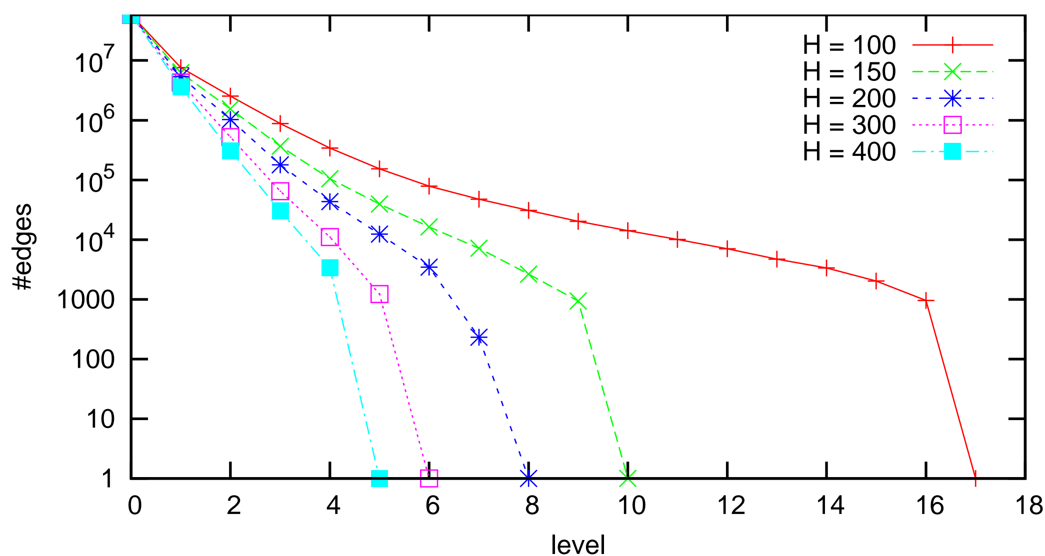


Figure 6.4: Shrinking of the highway networks of the USA, analogous to Fig. 6.3.

6.3 Multilevel Queries

Average Values. Table 6.1 contains average values for queries, where the source and target nodes are chosen randomly. For the two large graphs we get a speedup of more than 2 000 compared to DIJKSTRA's algorithm with respect to both query time[2] and the number of settled nodes.

For our largest road network (USA), the number of nodes that are settled during the search is *less* than the number of nodes that belong to the shortest paths that are found. Thus, we get an efficiency that is greater than 100%. The reason is that edges at high levels will often represent long paths containing many nodes.[3]

Local Queries. For use in applications it is unrealistic to assume a uniform distribution of queries in large graphs such as Europe or the USA. On the other hand, it would be hardly more realistic to arbitrarily cut the graph into smaller pieces. Therefore, we decided to measure local queries within the big graphs: For each power of two $r = 2^k$, we choose random sample points s and then use DIJKSTRA's algorithm to find the node t with DIJKSTRA rank $r_s(t) = r$. We then use our algorithm to make an s-t query. By plotting the resulting statistics for each value $r = 2^k$, we can see how the performance scales with a natural measure of difficulty of the query. Figure 6.5 shows the query times. The speedup with respect to DIJKSTRA's algorithm is shown in Fig. 6.6. Note that the median query times are scaling quite smoothly and the growth is much slower than the exponential increase we would expect in a plot with logarithmic x axis, linear y axis, and any growth rate of the form r^ρ for DIJKSTRA rank r and some constant power ρ. The curve is also not the straight line one would expect from a query time logarithmic in r.

Against the trend, the query times in Europe drop at DIJKSTRA rank 2^{24}. This rank is very close to the total number of nodes, which means that the target node is always close to the border of the road network. In general, the multilevel query algorithm does not exhibit any goal-directed behaviour, i.e., the search space extends in all directions. However, when the search is started from the border, it gets a 'trivial sense of direction' because it cannot spread in all directions. Therefore, the query times improve.

The average running time of queries in the German road network is 5.30 ms (cp. Table 6.1). Since the German road network consists of roughly 2^{22} nodes, we can expect an average DIJKSTRA rank of about 2^{21} for queries between nodes that are picked at random. The average running time of queries in the European road network from a random node s to a node t with DIJKSTRA rank 2^{21} is 5.29 ms. These results suggest that there is virtually no difference between executing the same query within the German or the European road network. This means that we can use a large road network (e.g. Europe) for all kinds of queries; it is not necessary to restrict the search to a part of it (e.g. Germany) in order to get fast queries within this part.

[2]It is likely that Dijkstra would profit more from a faster priority queue than our algorithm. Therefore, the time-speedup could decrease by a small constant factor.

[3]The reported query times do not include the time for expanding these paths. We have made measurements with our naive recursive expansion routine (cp. Section 5.3) which never take more than 50% of the query time. Also note that this process could be radically sped up by precomputing unpacked representations of edges.

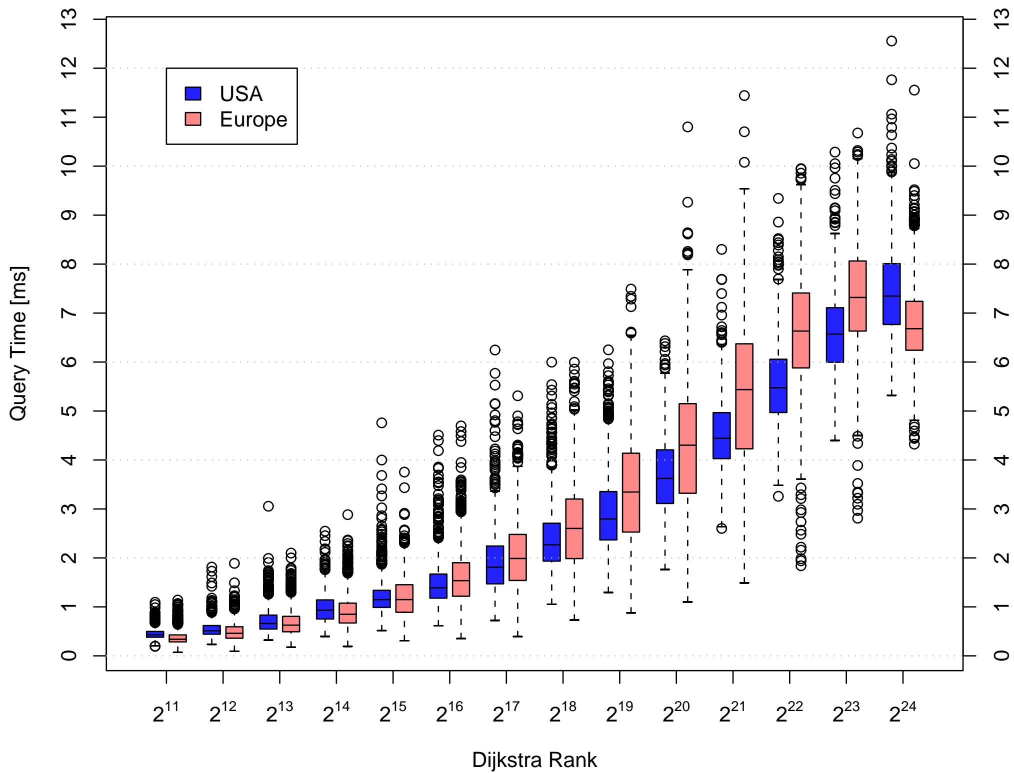


Figure 6.5: Multilevel Queries. For each road network and each DIJKSTRA rank on the x-axis, 1 000 queries from random source nodes were performed. The results are represented as box-and-whisker plot [24]: each box spreads from the lower to the upper quartile and contains the median, the whiskers extend to the minimum and maximum value omitting outliers, which are plotted individually.

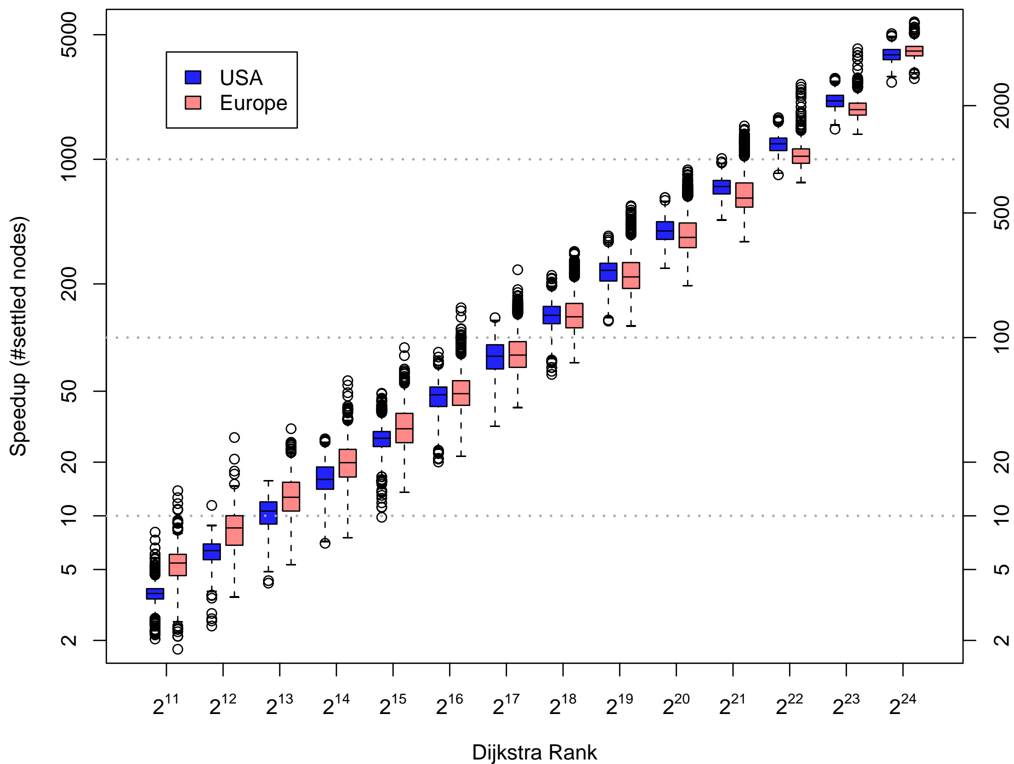


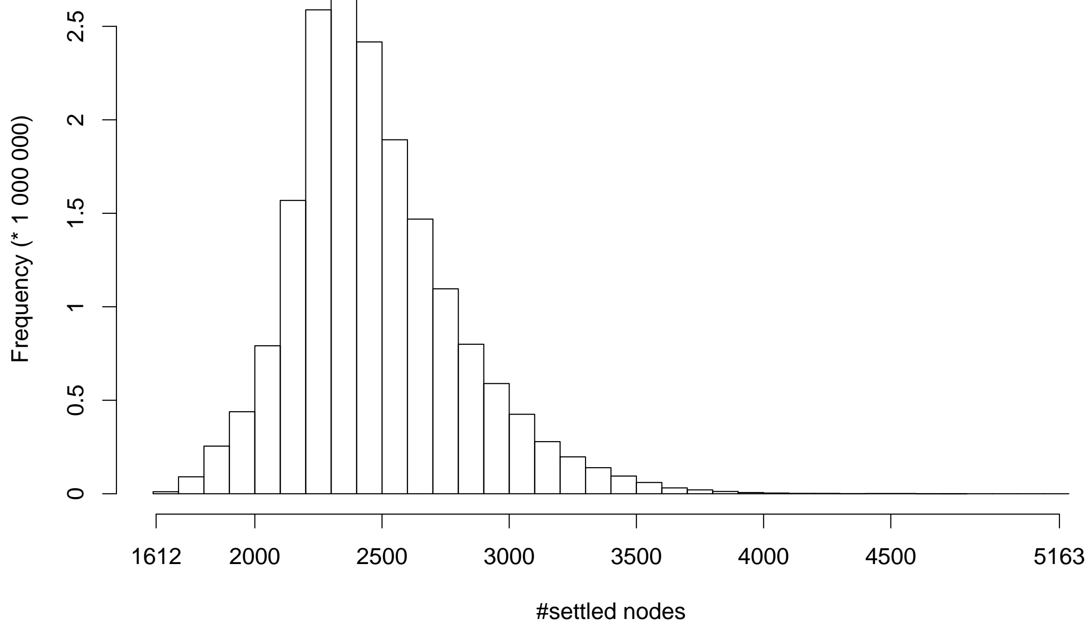
Figure 6.6: Multilevel Queries. Speedup in terms of number of settled nodes.

Worst Case. In order to determine the worst case for a query between two locations in Europe or the USA, we would have to perform all n^2 possible queries, which would be *very* time-consuming. However, we can provide an *upper bound* for the worst case executing only n queries: We add an isolated dummy node t to the graph and run one s-t query for each node s. Of course, the search from t terminates immediately, while the search from s explores the complete search space since the abort-on-success criterion never applies. Obviously, the worst case cannot be worse than twice the maximum of these n queries. By this means, we obtained an upper bound for the number of settled nodes for queries in Europe (the USA) of 10 326 (8 678), i.e., no more than 0.057% (0.036%) of all nodes are ever settled.

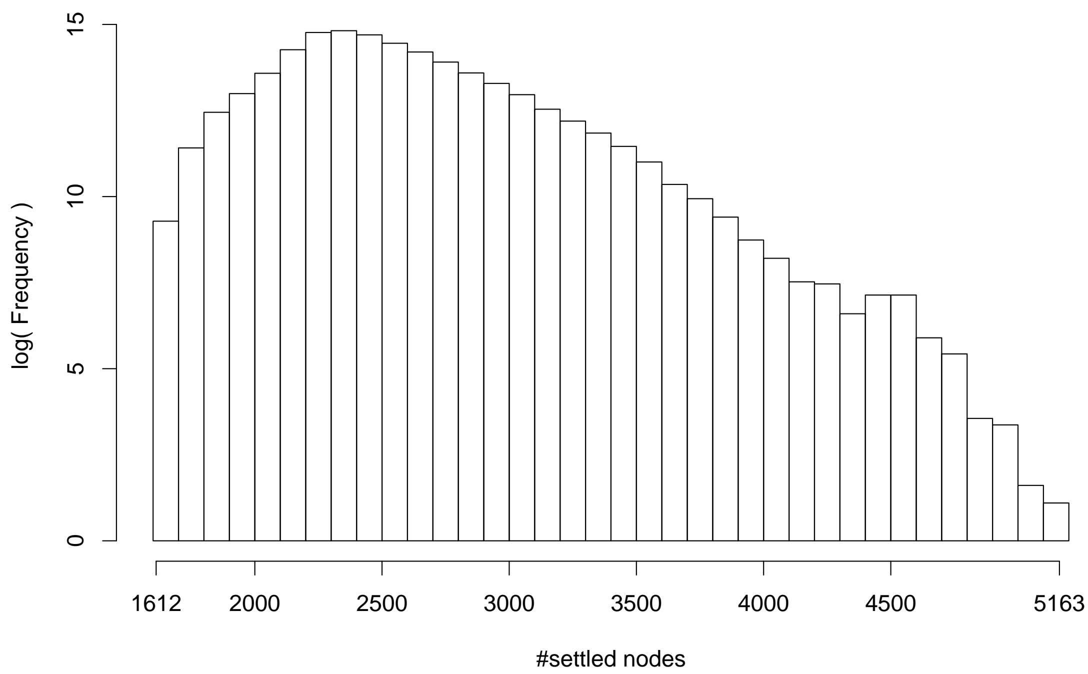
Histograms of these experiments are given in Fig. 6.7 and 6.8. We find that the costs of a search differ from source node to source node. Although most source nodes cause similar costs, there are a few outliers. We further investigated the costs with respect to the geographic location of the source node. Figure 6.9 indicates that the search from a congested urban area is rather easy, while the search from a rural area that is surrounded by several urban areas is rather difficult. The road network of a congested urban area is very compact. While the neighbourhood size in terms of the number of nodes is always the same, the geometric neighbourhood size is comparatively small in a city. Hence, the search switches to higher levels before spreading too far away. Thus, in the first levels, the search space stays compact and the scope of the different entrance points overlap. In contrast, the search started from a rural area spreads very far away before switching to higher levels. If it enters several surrounding urban areas when it is still in a low level, it gets quite expensive because it has to traverse all of them starting in a very detailed level.

Furthermore, Fig. 6.9 confirms our earlier statement that the search from nodes close to the border is comparatively easy: for instance, the search from the eastern border of Germany is easy due to the fact that the road networks of Poland and the Czech Republic do not belong to our test instance.

Distance Instead of Travel Time. Using travel times as edge weights intensifies the hierarchical properties of real world road networks, which is the foundation of our approach. If we use spatial distance as edge weights, the road networks still exhibit a (less distinct) hierarchy. We performed several experiments with the German road network using distances as edge weights. In this case, the effect of self-similarity, which we observed during previous experiments, did not occur, i.e., the shrinking factor decreased when we used the same neighbourhood size H iteratively. However, we obtained good results for a highway hierarchy consisting of seven levels, where we doubled the neighbourhood size in each iteration of the construction procedure, starting with 100. This led to an average query time of 32 ms; the speedup in terms of the number of settled nodes compared to DIJKSTRA's algorithm was 122.



(a) linear scale



(b) logarithmic scale

Figure 6.7: Histogram of unidirectional queries in Europe. The minimum and maximum costs are given explicitly as x-axis labels. Note that in (a) the extreme outliers are not visible because their frequency is very small. Queries from nodes that do not belong to the largest connected component of the road network have been omitted since they cause only very small costs, which cannot be compared directly to the other values.

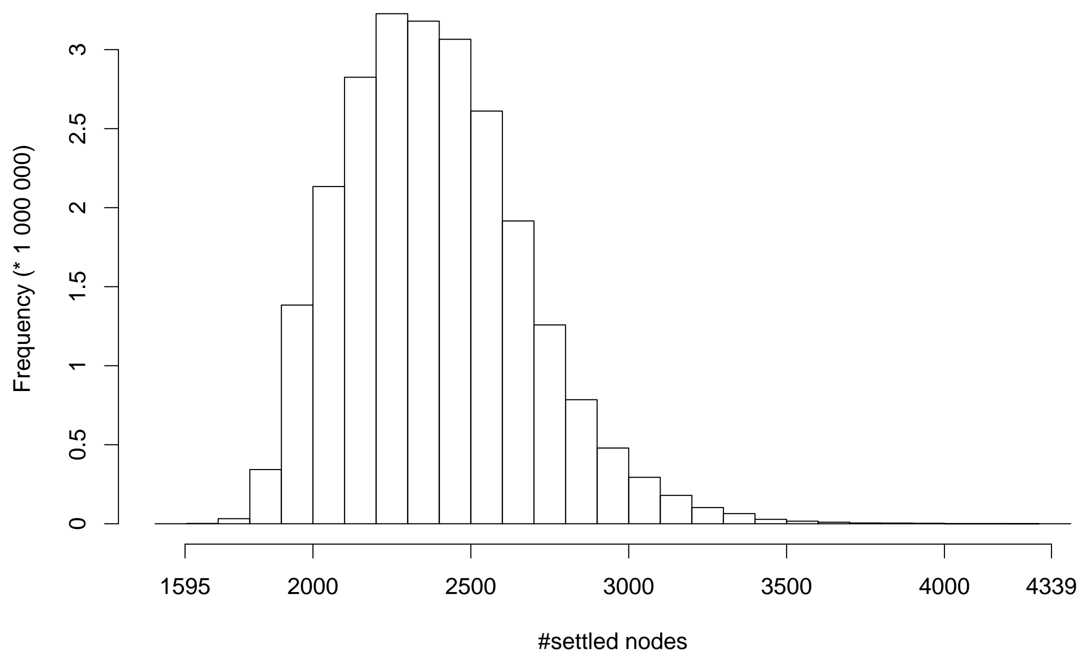


Figure 6.8: Histogram of unidirectional queries in the USA, analogous to Fig. 6.7 (a).

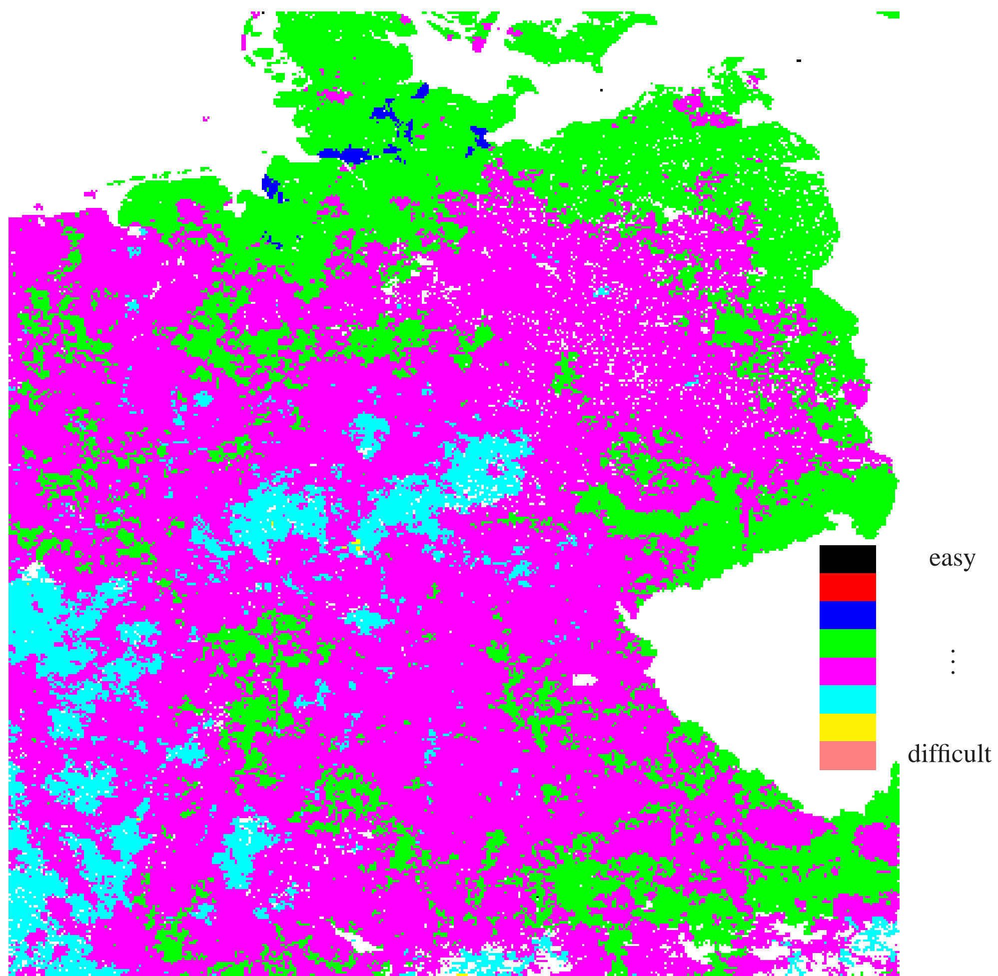


Figure 6.9: Unidirectional queries in Europe, clipped by a bounding box around Germany. Each node is coloured by the costs of a search started from it.

Chapter 7

Discussion

Conclusion

Starting from a simple definition of local search, we have developed nontrivial algorithms for constructing and querying highway hierarchies. We have presented strong evidence that highway hierarchies of the largest road networks currently used can be constructed in a few hours, i.e., fast enough to allow daily updates. The space consumption is only a small constant factor of the input size. The query times around 8 ms are more than fast enough for interactive use. In particular, overhead for the user interface (and possible internet communication) will probably dominate the interactive delays. The only previous speedup techniques that would achieve comparable speedup (bit vectors, geometric containers) have prohibitive preprocessing times for very large graphs.

Future Work

The current implementation supports only undirected graphs. At some points in this thesis, we have indicated how the implementation can be generalised in the future so that it can deal with directed graphs. Even faster preprocessing is a major issue for future work. We see many small (and not so small) opportunities for improvement. Obviously, parallelisation will yield a significant speedup. Adaptive neighbourhood sizes could benefit both the construction and the query. The local nature of preprocessing makes it likely that highway hierarchies can be quickly updated dynamically when only a few edges (e.g., for taking traffic jams into account) or a region of the network changes. Furthermore, multiple objective functions can be handled by a single highway hierarchy that is the union of the highway hierarchies for the individual objective functions. It seems likely that highway hierarchies for multiple *reasonable* objective functions have a very big overlap so that their union will still be useful.

Even faster queries are also interesting. For example, for some traffic simulations, millions of shortest paths queries are needed and there is no overhead for a user interface. Besides many small improvements (e.g. faster priority queues) a combination with other speedup techniques seems interesting. In particular, bit vectors, geometric containers, or landmarks give the search a strong sense of direction (Fig. 7.1) that highway hierarchies lack (Fig. 7.2). Thus, these two basic approaches may complement one another, i.e., we could achieve a significant improvement if we restricted the search space of our approach to

the intersection with the search space of a goal directed approach (Fig. 7.3). Moreover, the higher levels of the hierarchy are so small that superlinear time or space may be tolerable as long as the contributions of the lower levels can be incorporated efficiently.

Highway hierarchies are also promising for handling graphs that are too large for fast internal memory and only fit on hard disks of PCs or into the slow flash memory of mobile devices: The higher levels fit into fast memory and the lower levels are only searched locally. Hence, by packing local patches of the graph into the same external memory block, local searches should only need a small number of block accesses.

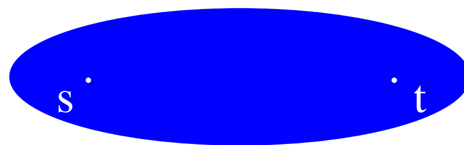


Figure 7.1: Schematic representation of the search space of a goal directed approach. The search from s and t does not spread uniformly into all directions, but tends to the respective goal.

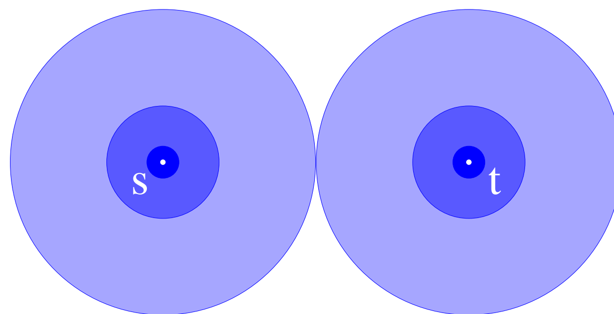


Figure 7.2: Schematic representation of the search space of our approach based on highway hierarchies. The search from s and t spreads uniformly into all directions. However, in contrast to Fig. 7.1, the search space gets thinner and thinner.

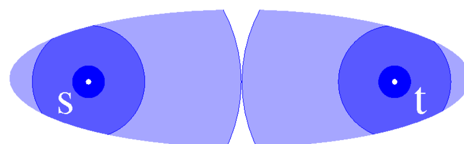


Figure 7.3: Intersection of the search spaces represented in Fig. 7.1 and 7.2.

Bibliography

[1] R. K. Ahuja, K. Mehlhorn, J. B. Orlin, and R. E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 37(2):213–223, 1990.

[2] V. Batagelj and M. Zaversnik. An $O(m)$ algorithm for cores decomposition of networks. *CoRR*, cs.DS/0310049, 2003.

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[4] U. Brandes, F. Schulz, D. Wagner, and T. Willhalm. Travel planning with self-made maps. In *3rd Workshop on Algorithm Engineering and Experiments*, volume 2153 of *LNCS*, pages 132–144. Springer, 2001.

[5] U. Brandes, F. Schulz, D. Wagner, and T. Willhalm. Generating node coordinates for shortest-path computations in transportation networks. *ACM Journal of Experimental Algorithmics*, 9(1.1), 2004.

[6] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest path algorithms: Theory and experimental evaluation. *Math. Programming*, 73:129–174, 1996.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

[8] R. B. Dial. Algorithm 360: Shortest-path forest with topological ordering. *Communications of the ACM*, 12(11):632–633, 1969.

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

[10] J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. In *42nd IEEE Symposium on Foundations of Computer Science*, pages 232–241, 2001.

[11] I. C. M. Flinsenberg. *Route planning algorithms for car navigation*. PhD thesis, Technische Universiteit Eindhoven, 2004.

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

[13] A. V. Goldberg and C. Harrelson. Computing the shortest path: A^* meets graph theory. Technical Report MSR-TR-2004-24, Microsoft Research, 2004.

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

[15] R. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *6th Workshop on Algorithm Engineering and Experiments*, 2004.

[16] 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.

[17] M. Holzer, F. Schulz, and T. Willhalm. Combining speed-up techniques for shortest-path computations. In *3rd International Workshop on Experimental and Efficient Algorithms*, volume 3059 of *LNCS*, pages 269–284. Springer, 2004.

[18] P. Klein, S. Rao, M. Rauch, and S. Subramanian. Faster shortest-path algorithms for planar graphs. In *26th ACM Symposium on Theory of Computing*, pages 27–37, 1994.

[19] E. Köhler, R. H. Möhring, and H. Schilling. Acceleration of shortest path and constrained shortest path computation. In *4th International Workshop on Efficient and Experimental Algorithms*, 2005.

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

[21] U. Meyer. Single-source shortest-paths on arbitrary directed graphs in linear average-case time. In *12th Symposium on Discrete Algorithms*, pages 797–806, 2001.

[22] 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.

[23] I. Pohl. Bi-directional search. *Machine Intelligence*, 6:124–140, 1971.

[24] R Development Core Team. R: A Language and Environment for Statistical Computing. http://www.r-project.org, 2004.

[25] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In *13th European Symposium on Algorithms (ESA)*, 2005. To appear.

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

[27] F. Schulz, D. Wagner, and K. Weihe. Dijkstra's algorithm on-line: an empirical case study from public railroad transport. *ACM Journal of Experimental Algorithmics*, 5:12, 2000.

[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] S. S. Skiena. *The Algorithm Design Manual*. Springer, 1998.

[30] M. Thorup. On RAM priority queues. In *7th ACM-SIAM Symposium on Discrete Algorithms*, pages 59–67, 1996.

[31] M. Thorup. Undirected single source shortest paths in linear time. In *Foundations of Computer Science*, 1997.

[32] M. Thorup. Undirected single source shortest paths in linear time. *Journal of the ACM*, 46(3):362–394, 1999.

[33] M. Thorup. On RAM priority queues. *SIAM Journal on Computing*, 30:86–109, 2000.

[34] M. Thorup. Compact oracles for reachability and approximate distances in planar digraphs. In *42nd IEEE Symposium on Foundations of Computer Science*, pages 242–251, 2001.

[35] M. Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. In *35th ACM Symposium on Theory of Computing*, pages 149–158, 2003.

[36] M. Thorup. Compact oracles for reachability and approximate distances in planar digraphs. *Journal of the ACM*, 51(6):993–1024, 2004.

[37] M. Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. *Journal of Computer and System Sciences*, 69(3):330–353, 2004.

[38] M. Thorup and U. Zwick. Approximate distance oracles. In *33th ACM Symposium on the Theory of Computing*, pages 183–192, 2001.

[39] M. Thorup and U. Zwick. Approximate distance oracles. *Journal of the ACM*, 51(1):1–24, January 2005.

[40] U.S. Census Bureau, Washington, DC. UA Census 2000 TIGER/Line Files. http://www.census.gov/geo/www/tiger/tigerua/ua_tgr2k.html, 2002.

[41] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Syst. Theory*, 10:99–127, 1977.

[42] 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.

[43] D. Wagner and T. Willhalm. Drawing graphs to speed up shortest-path computations. In *7th Workshop on Algorithm Engineering and Experiments*, 2005.

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

[45] J. W. J. Williams. Heapsort. *Communications of the ACM*, 7:347–348, June 1964.

Appendix A

Canonical Shortest Paths

A.1 Modifications of DIJKSTRA's Algorithm

We can modify DIJKSTRA's algorithm so that only canonical shortest paths are found. In order to do so, three conditions must be fulfilled:

1. An element in the priority queue is only updated if a shorter path to the corresponding node is found, and *not* if another path of the same length is found.

2. The adjacency list of a node is always processed in the same order.

3. The priority queue has the *FIFO property*, i.e., if there is more than one minimum element, then the older element is removed first. The age of an element refers to the last *decreaseKey* operation or to the *insert* operation if no *decreaseKey* operation has taken place.

Theorem 9 *On these conditions, we claim that if* DIJKSTRA *finds the shortest path* $P = \langle s, \dots, s', \dots, t', \dots, t \rangle$ *from* s *to* t*, then started from* s'*, it will find the corresponding subpath* $P|_{s' \to t}$ *of* P *as the shortest path to* t*. Obviously, this implies that* $P|_{s' \to t'}$ *is found as the shortest path from* s' *to* t'*.*

Proof: In order to obtain a contradiction, we assume that DIJKSTRA finds another shortest path $Q \neq P|_{s' \to t}$ from s' to t. We can write

$$\begin{array}{rllll} P = & \langle s, \dots, & s', \dots, s'', u_j, u_{j-1}, \dots, & u_2, u_1, & t', \dots, t \rangle \quad \text{and} \\ Q = & \langle & s', \dots, s'', v_k, v_{k-1}, \dots, & v_2, v_1, & t', \dots, t \rangle \end{array}$$

such that $u_j \neq v_k$ and $u_1 \neq v_1$.

When the search is started from s', the node v_1 is settled before u_1. (Otherwise, t' would be settled from u_1 because of Condition 1 and the fact that the distance from s' via v_1 to t' is equal to the distance from s' via u_1 to t'.) Hence, $d(s', v_1) \leq d(s', u_1)$. Furthermore, we have $d(s', u_1) \leq d(s', v_1)$ because during the search started from s, u_1 is settled before v_1. Thus, $d(s', u_1) = d(s', v_1)$.

Therefore, the search from s' settles v_2 before u_2. (Otherwise, u_1 would be settled before v_1 because of Condition 3.) We can conclude that $d(s', u_2) = d(s', v_2)$. We can use this

argument inductively to obtain $d(s', u_\ell) = d(s', v_\ell)$, where $\ell = \min\{j, k\}$. Furthermore, we can show that v_ℓ is settled before u_ℓ. We now distinguish three cases.

The case that $j > k = \ell$ cannot occur: When the search is started from s and when s'' is settled, then u_k has not been settled yet. (Otherwise, the shortest path P from s to t would be different.) Obviously, the node u_k cannot be in the priority queue with a smaller tentative distance than $d(s, u_k)$. Furthermore, it cannot be in the queue with the same distance $d(s, u_k)$ because in this case the shortest path from s to u_k would not pass by s'' due to Condition 3. For similar reasons, there cannot be a direct link from s'' to u_k of the same distance $d(s'', u_k)$. Therefore, after s'' has been settled, v_k is in the priority queue with the tentative distance $d(s, v_k) = d(s, u_k)$, while u_k is not yet in the queue with this tentative distance. Hence, due to Condition 3, v_k would be settled before u_k so that the shortest path from s to t would be different from P.

The case that $\ell = j < k$ cannot occur either. It is symmetric to the first case.

The case that $\ell = j = k$ remains. When the search is started from s' and when s'' is settled, neither u_j nor v_k is in the priority queue with the tentative distance $d(s', u_j) = d(s', v_k)$. (Otherwise, the shortest path from s' to t would not pass through s'' due to Condition 3.) Since v_k is settled before u_j, we know that the edge (s'', v_k) appears in the adjacency list of s'' before (s'', u_j). Analogously, when the search is started from s and when s'' is settled, neither u_j nor v_k is in the priority queue with the tentative distance $d(s, u_j) = d(s, v_k)$. According to Condition 2, the adjacency list of s'' is always processed in the same order. Therefore, v_k is added to the queue before u_j (or a *decreaseKey* operation on v_k is performed first). Hence, v_k is settled before u_j so that the resulting shortest path from s to t is different from P, which is a contradiction. □

A.2 FIFO Priority Queues

While Condition 1 and 2 are usually fulfilled automatically by any implementation of DIJKSTRA's algorithm, Condition 3 is in general *not* guaranteed by a usual implementation of a priority queue. However, for any given implementation of any priority queue, we can ensure Condition 3 by adding a counter that is initially set to 0 and that counts all *insert* and *decreaseKey* operations. When an element is inserted or a *decreaseKey* operation is performed on an element, the current value of the counter is stored as a timestamp in addition to the key of the element. When a comparison between two elements takes place and the keys are equal, then the counts are compared, which leads to an unambiguous result.

The asymptotic complexity of the operations is not affected. However, the constants can rise if it is not possible to store both the key and the count in one machine word.

Appendix B

Examples

B.1 Construction

The Road Network of Karlsruhe. Figures B.1–B.6 visualise the first two iterations of the construction process of the European highway hierarchy, clipped by a bounding box around the German city of Karlsruhe. The used parameters comply with Table 6.1.

Figure B.1: Input. Level 0.

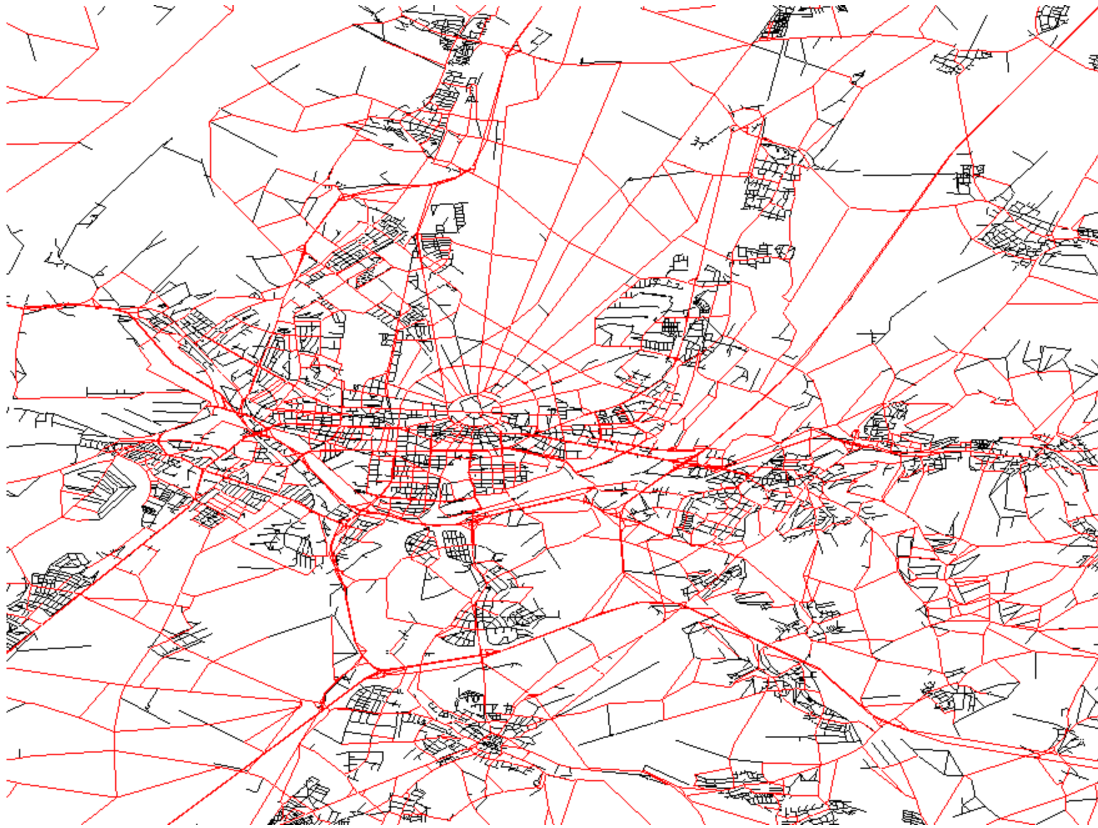


Figure B.2: First iteration of the construction. Level 0 and level 1.

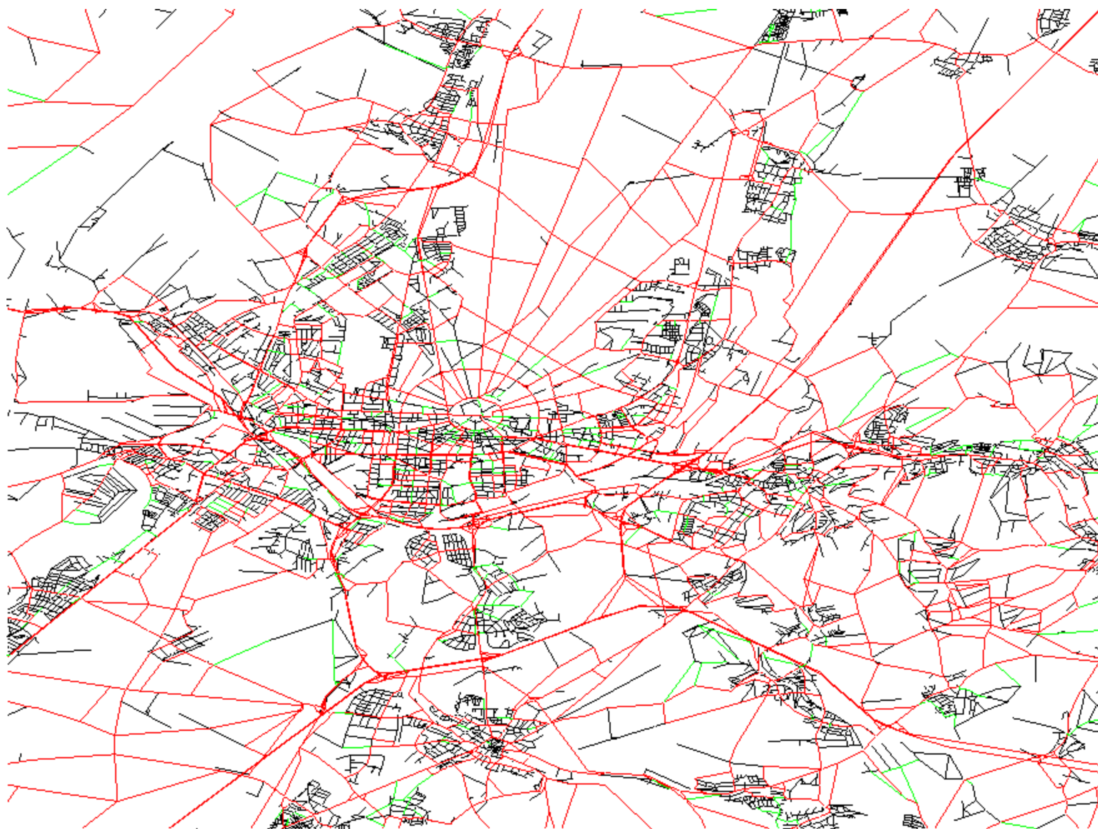


Figure B.3: Determine trees. Level 0 and level 1, trees are highlighted.

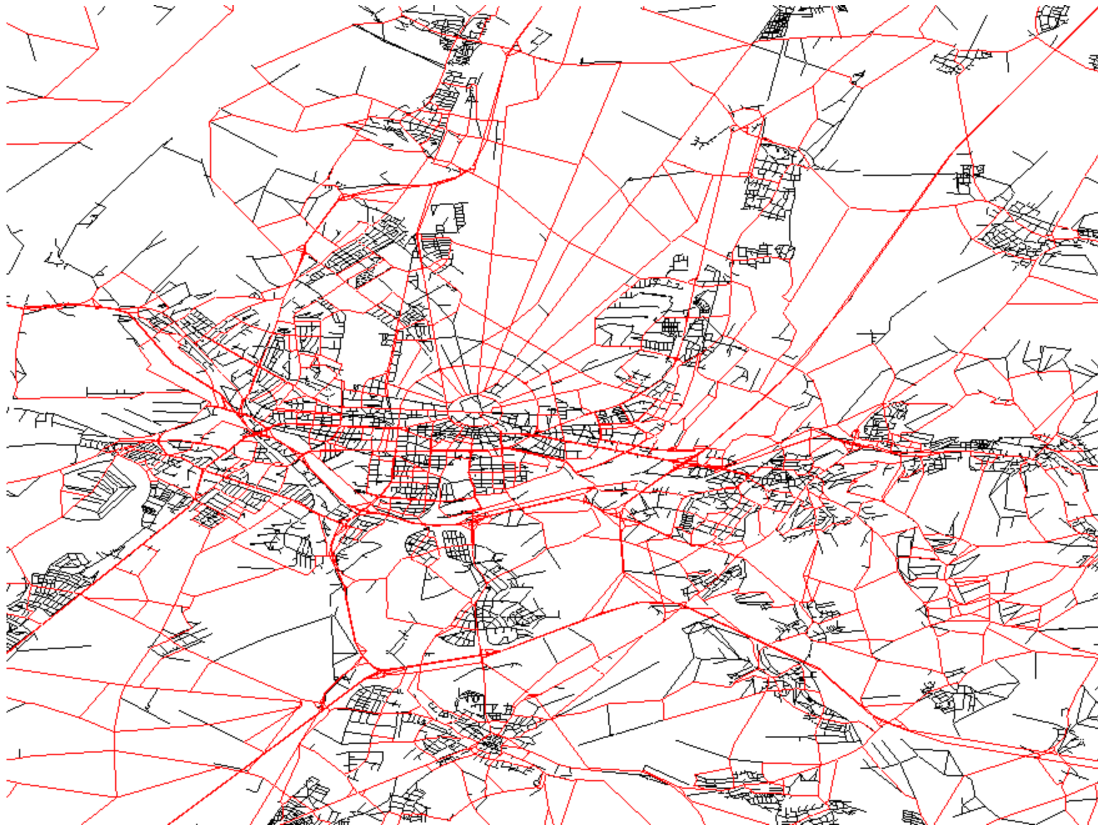


Figure B.4: Remove trees. Level 0 and 2-core of level 1.

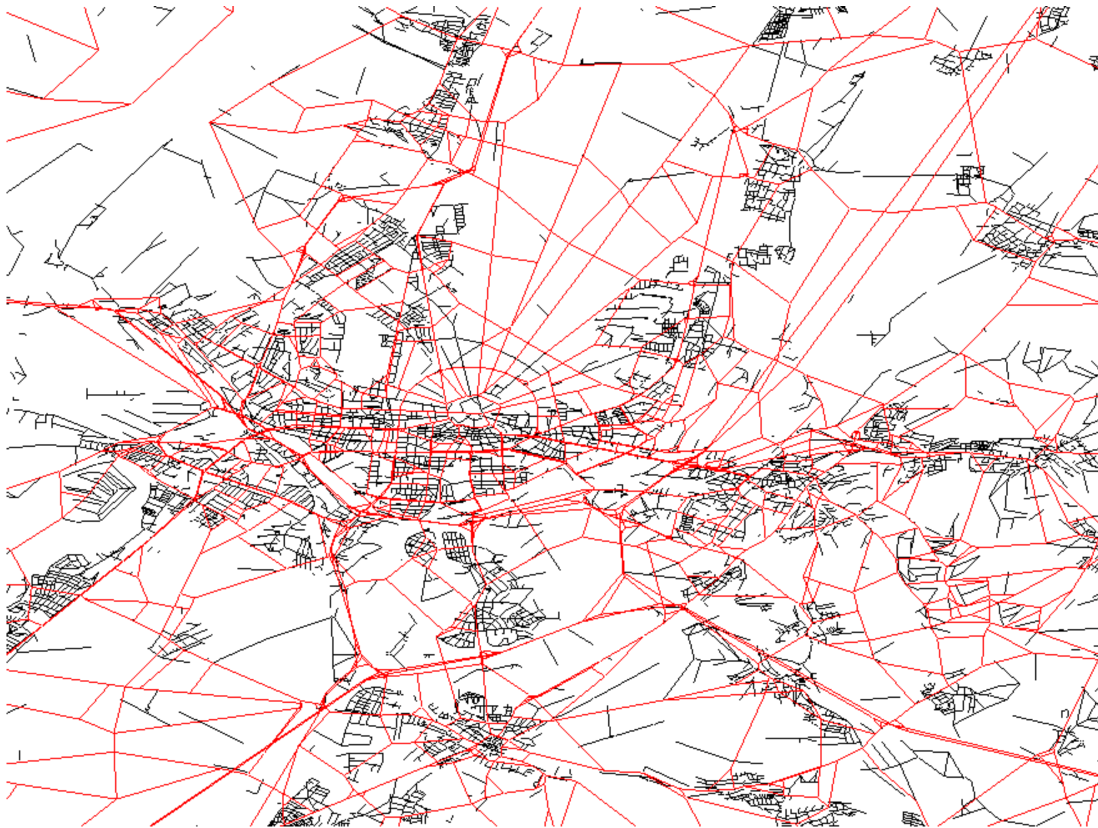


Figure B.5: Contract Lines. Level 0 and core of level 1.

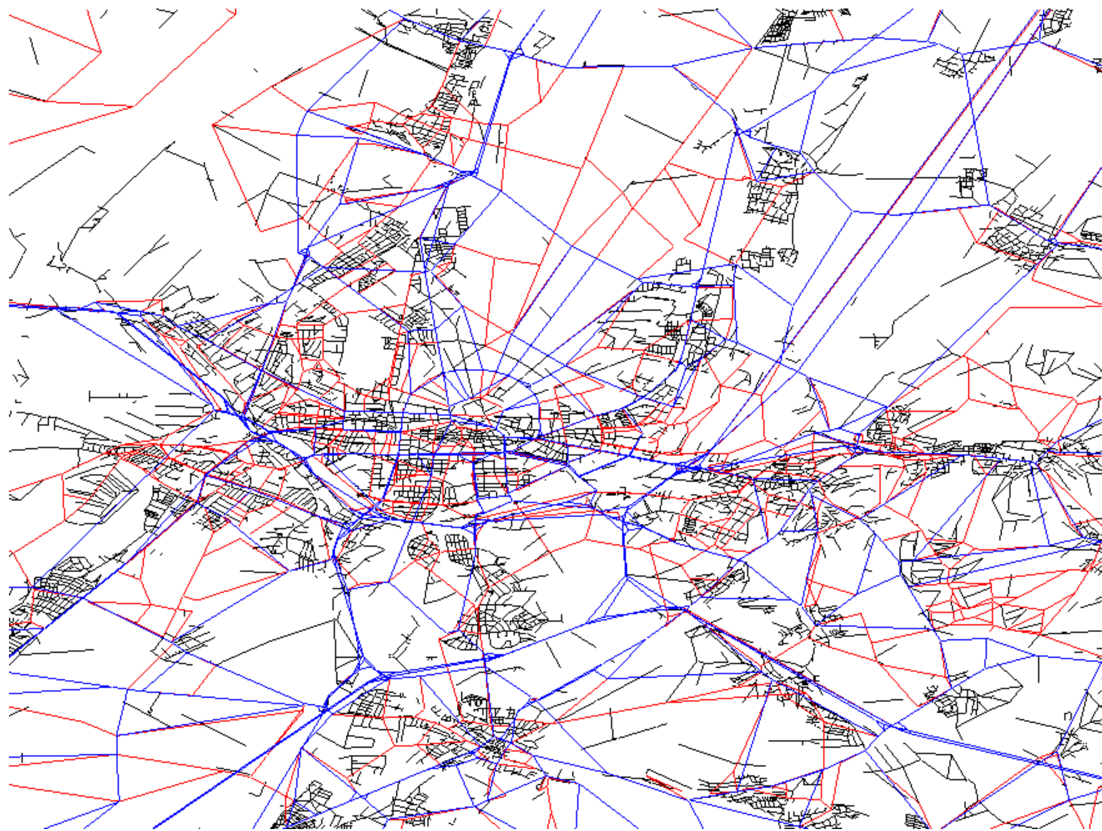


Figure B.6: Second iteration of the construction and contraction. Level 0, core of level 1, and core of level 2.

B.2 Highway Hierarchy

Western Europe – Colours by Level. Figures B.7–B.9 show the European highway hierarchy. Note that edges representing long subpaths ('*lines*') are not drawn as direct shortcuts, but by showing the actual geographic route that they represent.

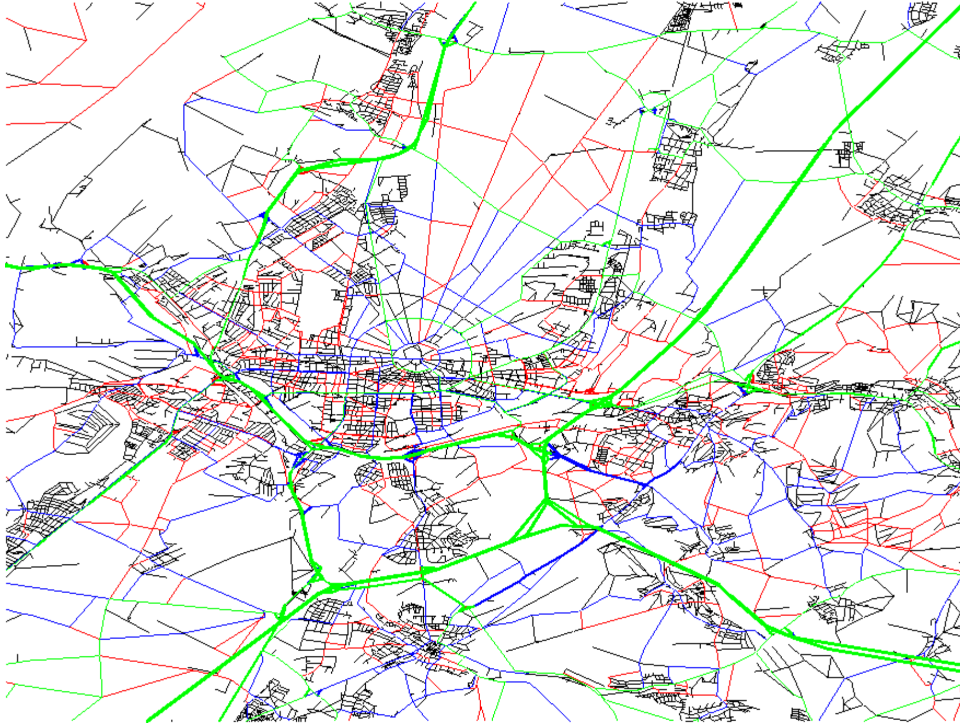


Figure B.7: Level 0, level 1, level 2, and level 3, clipped by a bounding box around Karlsruhe (20 km in north-south direction).

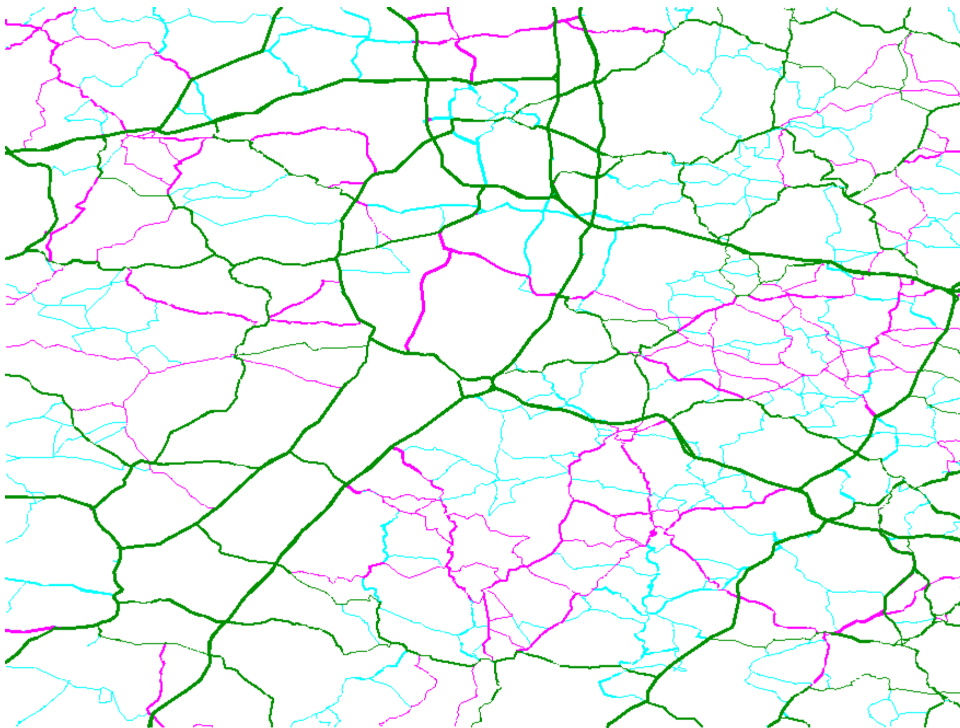


Figure B.8: Level 4, level 5, and level 6, clipped by a bounding box around Karlsruhe (150 km).

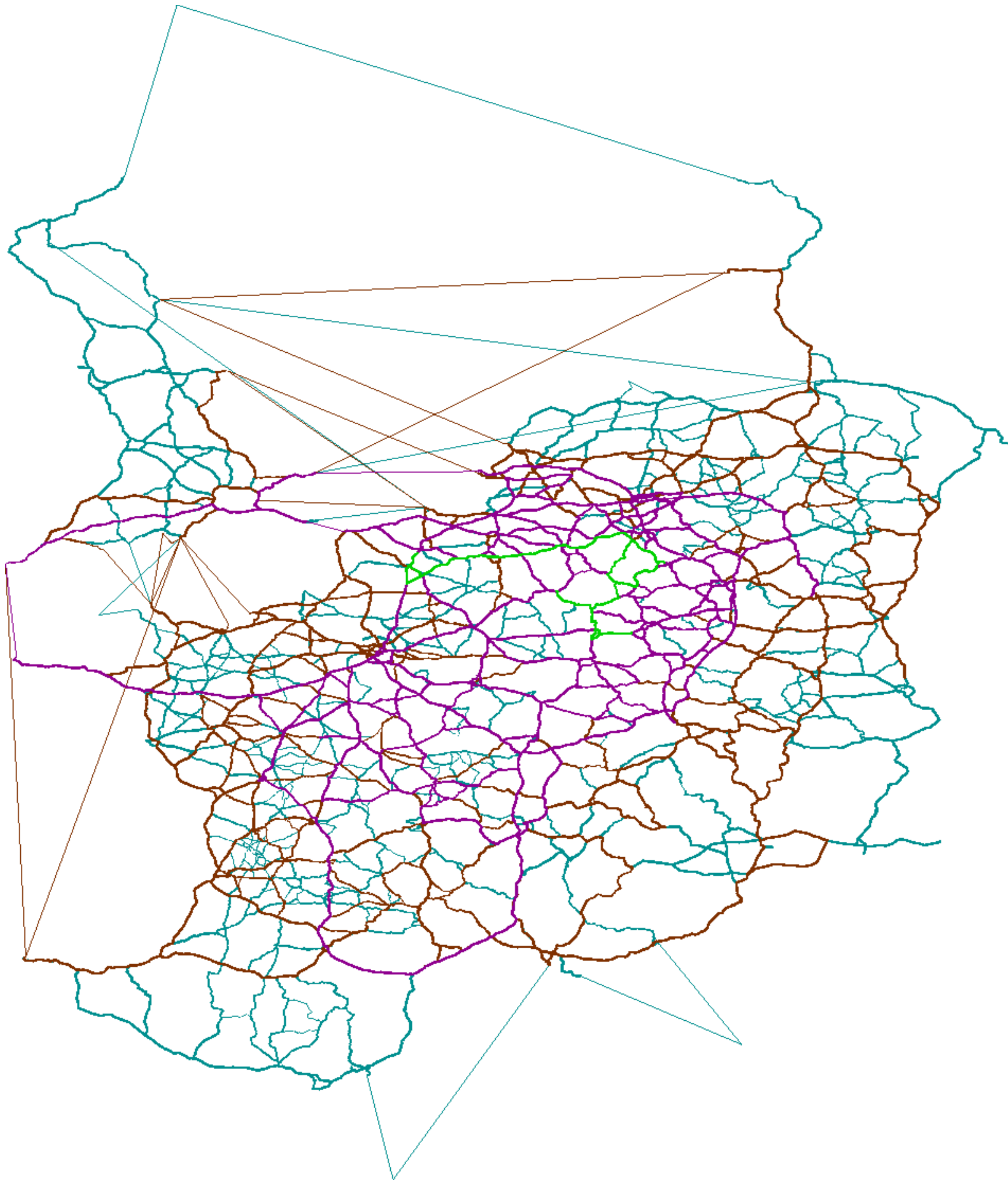


Figure B.9: Level 7, level 8, level 9, and level 10.

Western Europe – Colours by Road Category. Figures B.10–B.12 show the European highway hierarchy.

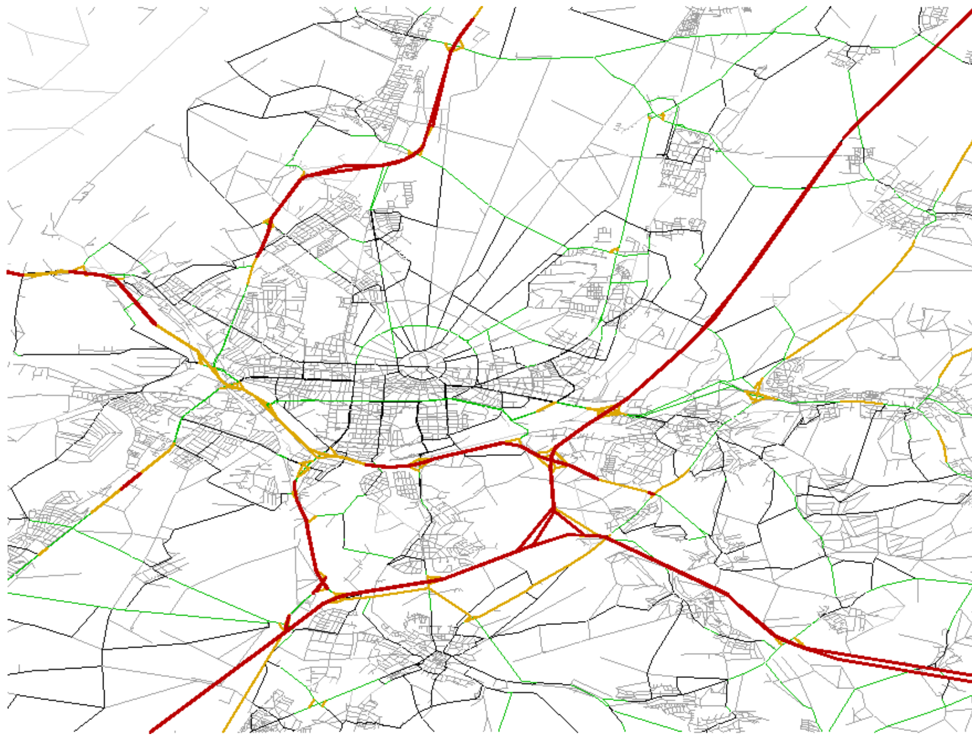


Figure B.10: Level 2. The roads are distinguished by their supercategory: motorway, national road, regional road, or urban street. Clipped by a bounding box around Karlsruhe (20 km in north-south direction). Levels 0 and 1 are plotted in grey.

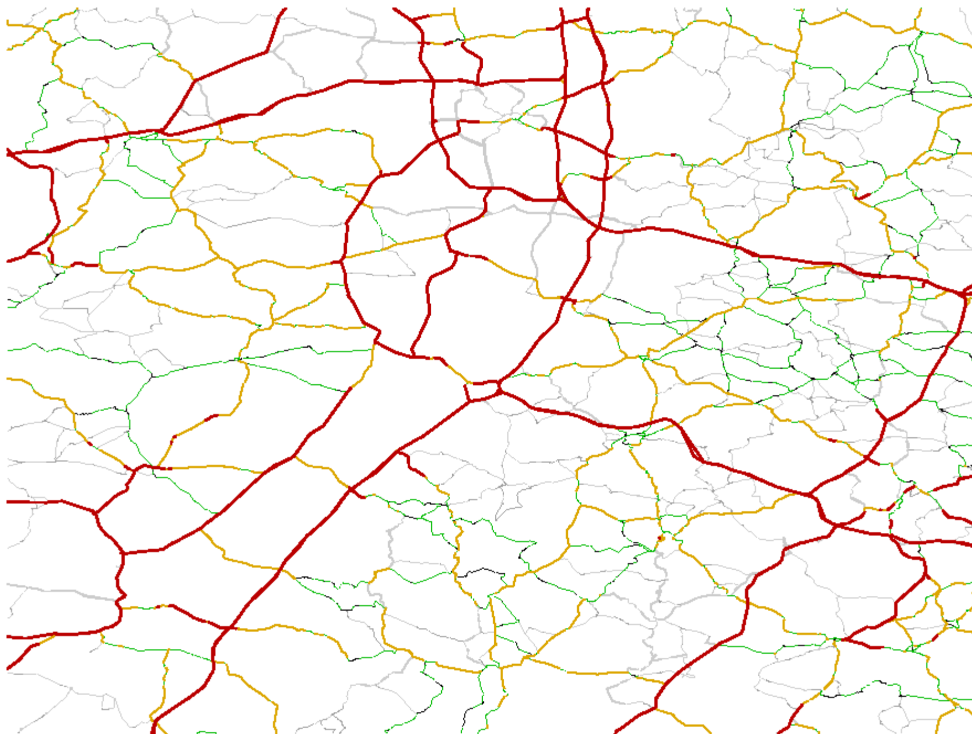


Figure B.11: Level 5. Clipped by a bounding box around Karlsruhe (150 km in north-south direction). Level 4 is plotted in grey.

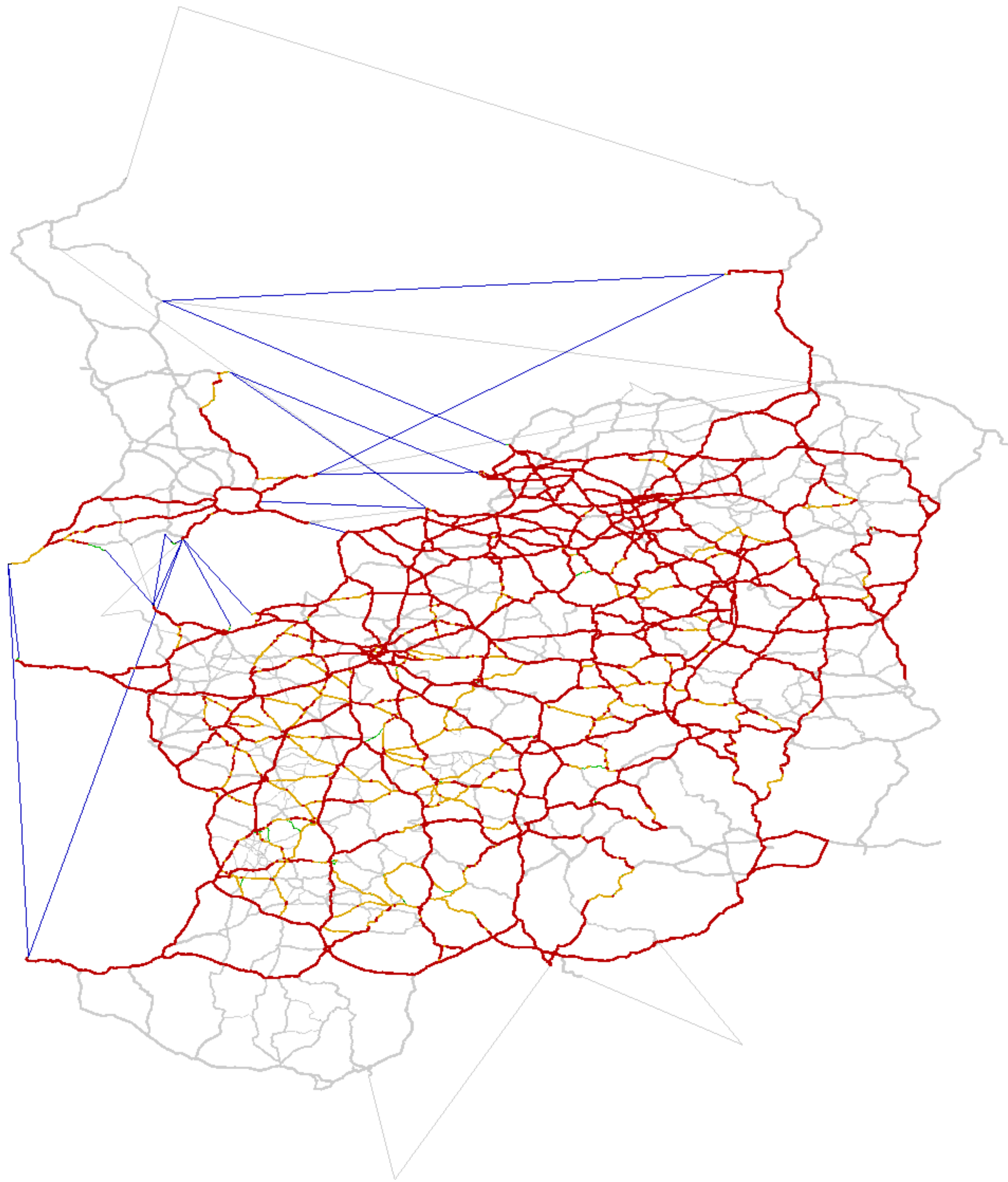


Figure B.12: Level 8. Ferries are represented by blue lines. Level 7 is plotted in grey.

B.3 Query

From Karlsruhe To Palma de Mallorca. Figures B.13–B.16 illustrate a query from Karlsruhe, Am Fasanengarten 5 to Palma de Mallorca. All edges that are relaxed during the search are coloured – the colour reflects the search level. For the sake of clarity, the progress is demonstrated search level by search level. However, the actual order of events does not necessarily correspond to this representation: the nodes are settled according to their distance from the source node, irrespectively of the search level; thus, it is possible that a node in search level ℓ is settled earlier than a node in search level $\ell' < \ell$.

In addition to the search space, the unused edges of the corresponding level of the highway hierarchy are plotted in grey. Thick lines represent (a part of) the shortest path. Note that edges representing long subpaths (‘*lines*’) are not drawn as direct shortcuts, but by showing the actual geographic route taken. For the first levels, we omit the search space of the backward search (from Palma) and concentrate on the forward search (from Karlsruhe).

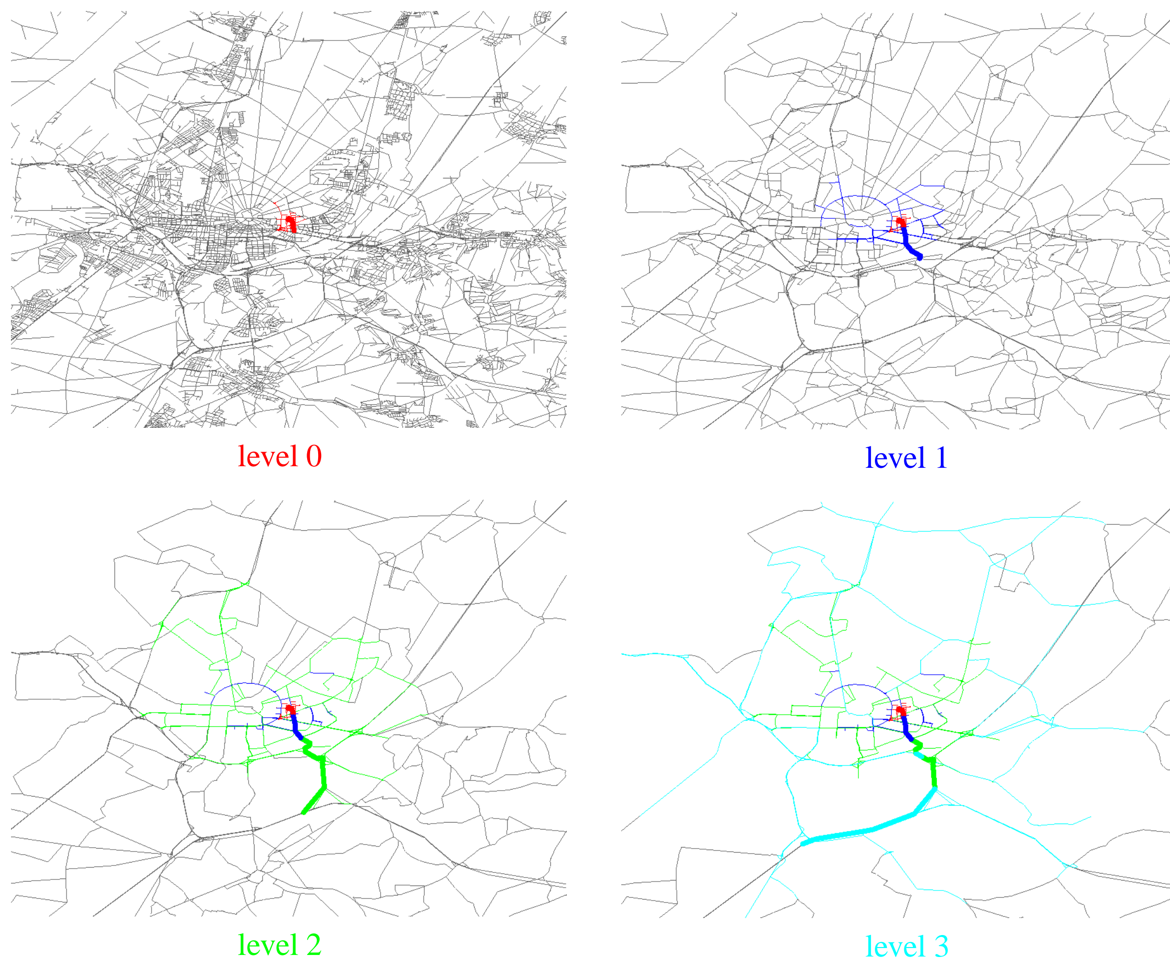
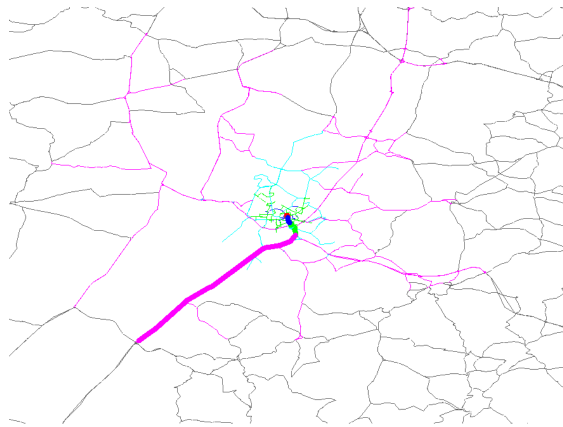
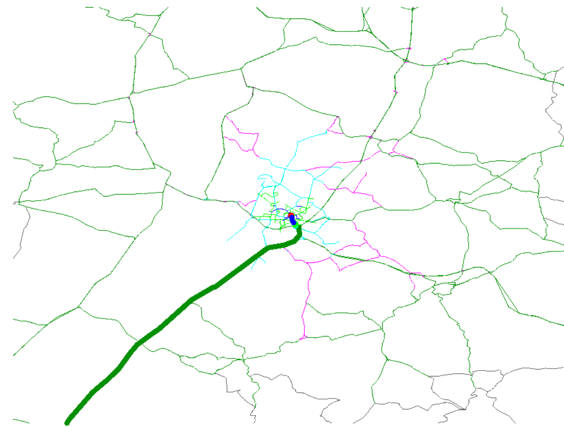


Figure B.13: Search from Karlsruhe. Bounding box size in north-south direction: 20 km.

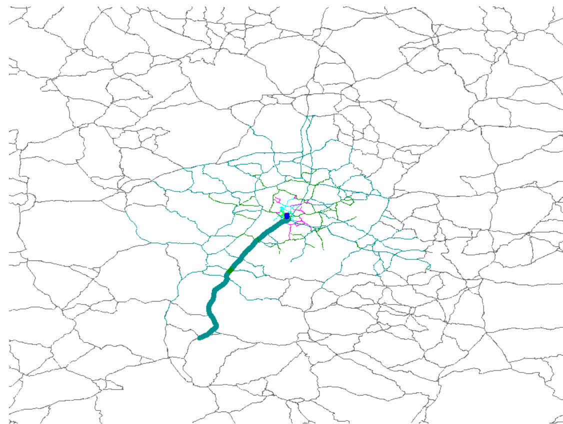


level 4

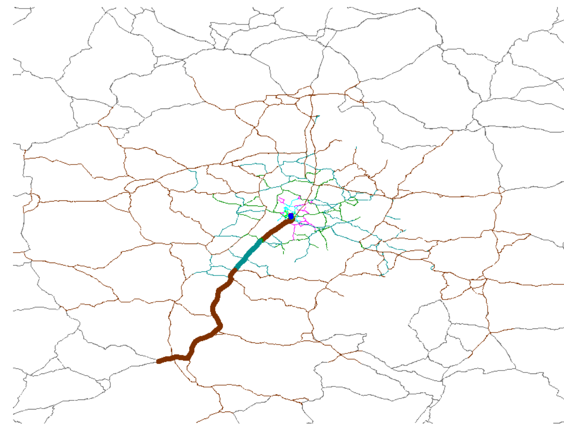


level 5

Figure B.14: Search from Karlsruhe. Bounding box size in north-south direction: 80 km.



level 6



level 7

Figure B.15: Search from Karlsruhe. Bounding box size in north-south direction: 400 km.

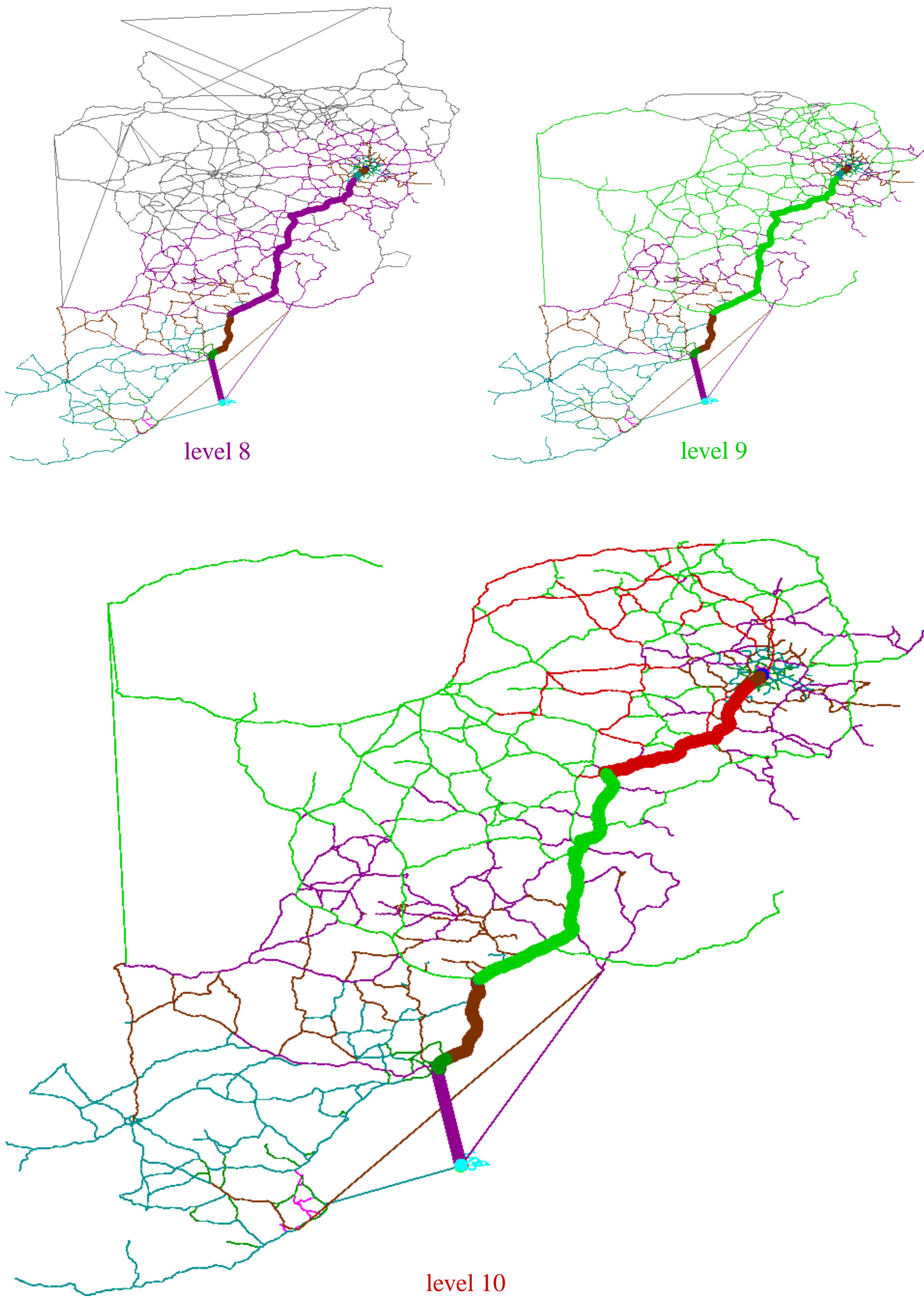


Figure B.16: Bidirectional search.