Universität Stuttgart

Contraction Hierarchies: Theory and Applications

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Tobias Rupp

aus Ludwigsburg

Hauptberichter:	Prof. Dr. Stefan Funke
Mitberichter:	Prof. Dr. Christian S. Jensen
Tag der mündlichen Prüfung:	16.11.2022

Institut für Formale Methoden der Informatik

2022

Contents

Zusammenfassung

Contraction Hierarchies beschreibt eine Technik, die die Berechnung kürzester Wege auf Straßennetzen in der Praxis enorm beschleunigt.

Im ersten Teil dieser Arbeit wird diese Beschleunigung theoretisch analysiert. Dazu führen wir die Graphenfamilie der *Lightheaded*-Grids ein. Auf der einen Seite haben Lightheaded-Grids realistische Eigenschaften wie Planarität und kommen in Straßennetzen der realen Welt vor. Andererseits sind sie aufgrund ihrer generischen Struktur für eine theoretische Analyse geeignet. Wir beweisen eine untere Schranke $\Omega(\sqrt{n})$ für die durchschnittliche Queryphase von Contraction Hierarchies auf Lightheaded-Grids. *Hub-Labels* sind eine weitere Technik, die eine noch schnellere Beantwortung von Kürzeste-Wege-Anfragen auf Kosten von mehr Speicherplatz ermöglicht als Contraction Hierarchies. Für Hub-Labels beweisen wir anhand der gleichen Idee wie aus dem vorigen Beweis, dass die durchschnittliche Hub-Label-Größe eines Knotens in $\Omega(\sqrt{n})$ auf Lightheaded-Grids.

Dann entwickeln wir diese Beweisidee noch weiter zu einem Schema, das instanzbasierte untere Schranken sowohl für Kontraktionshierarchien als auch für Hub-Labels auf jedem beliebigen gegebenen Graphen berechnen kann. Für einige Graphenfamilien wie balancierte Ternärbäume zeigen wir, dass dieses Schema scharfe Schranken liefert, d.h. die unteren Grenzen passen zu einer konkreten Contraction Hierarchy und einem Hub-Labeling.

In früheren Beiträgen zu oberen Schranken wurde angenommen, dass es ausreicht nur explorierte Knoten zu berücksichtigen, um die Abfragezeit nach oben zu beschränken. Zum Beispiel wurde für eine vorgeschlagene Kontraktionsstrategie basierend auf Separatoren bewiesen, dass während der Abfragezeit von Contraction Hierarchies im Durchschnitt nur sublinear viele Knoten für eine große Graphfamilie untersucht werden, zu denen auch Lightheads-Grids gehören. Für Lightheaded-Grids beweisen wir, dass diese Strategie im Durchschnitt linear viele zu untersuchende Kanten ergibt, wodurch gezeigt wird, dass es nicht ausreicht, nur die Anzahl der untersuchten Knoten zu berücksichtigen. Wir zeigen auch, dass nicht das Contraction Hierarchy Framework notwendigerweise fehlerhaft ist, sondern die vorgeschlagene Strategie, indem wir eine andere Strategie angeben für die im Durchschnitt nur sublinear viele Kanten untersucht werden müssen.

Im zweiten Teil dieser Arbeit werden Contraction Hierarchies angepasst, um sie für andere Zwecke als zur Beantwortung von Kürzeste-Wege-Anfragen zu verwenden. Zunächst zeigen wir, wie Trajektorien, d.h. Bewegungen von Personen auf dem Straßennetz, effizient gespeichert werden können falls die Datenstruktur der Contraction Hierarchy vorhanden ist. Bei der Beschreibung des Komprimierungsprozesses von rohen Trajektoriendaten beweisen wir auch, dass die komprimierte Darstellung einzigartig ist. Das Komprimieren der Daten ist auch vorteilhaft für die Beantwortung

von Abfragen nach allen Trajektorien welche einen gegebenen Raum-Zeit-Würfel schneiden. Wir beschreiben, wie die Datenstruktur der Kontraktionshierarchie erweitert werden kann, um solche Anfragen schneller zu beantworten als herkömmliche Ansätze.

Interessanterweise können Contraction Hierarchies auch verwendet werden, um Graphvereinfachungen zu visualisieren, wie es bereits im URAN-Framework demonstriert wurde. Das URAN-Framework wies jedoch einige Mängel auf, die wir behandeln. Wir stellen sicher, dass ein vereinfachter Graph seine ursprüngliche Silhouette besser beibehält und einige topologische Inkonsistenzen vermeidet. Darüber hinaus wenden wir ähnliche Techniken an um Trajektoriendatensätze innerhalb unseres Frameworks PATHFINDER$^{\text{VIS}}$ zu visualisieren, das auch in der Lage ist, verschiedene Arten der Visualisierung bereitzustellen, einschließlich Heatmap-basierter.

Abstract

Contraction Hierarchies describes a technique that enormously accelerates the calculation of shortest paths on road networks in practice.

In the first part of this thesis, this acceleration is analyzed theoretically. For this purpose, we introduce the graph family of *lightheaded* grids. On one hand, light-headed grids have realistic properties as planarity and occur in real world road networks. On the other hand, their generic structure makes them suitable for theoretical analysis. We prove a lower bound $\Omega(\sqrt{n})$ for the average query phase of contraction hierarchies on lightheaded grids. At the cost of more disk space, *hub labels* is another technique which enables even quicker answering of shortest path queries than contraction hierarchies. For hub labels, we prove that the average hub label size of a node is in $\Omega(\sqrt{n})$ on lightheaded grids with the same idea as in the previous proof.

Then we develop this proof idea even further into a schema which is able to compute instance-based lower bounds for both contraction hierarchies and hub labels on any arbitrary given graph. For some graph families such as balanced ternary trees, we show that this schema yields tight bounds, i.e. the lower bounds matches a concrete contraction hierarchy and hub labeling.

In previous contributions concerning upper bounds, it was believed that it is meaningful to consider only explored nodes to upper bound the query time. For example, it was proven for one proposed contraction strategy based on separators, that during query time of contraction hierarchies, only sublinear many nodes are explored on average for a wide family of graphs which includes lightheaded grids. For lightheaded grids, we prove that this strategy yields linear many edges to be explored on average, therefore showing that considering only the number of explored nodes is not sufficient. We also show that it is not the contraction hierarchy framework which is necessarily flawed but the proposed strategy by giving another strategy for which only sublinear many edges need to be explored on average.

In the second part of this thesis, contraction hierarchies are adapted to fit other purposes than answering short paths queries only. First, we show how trajectories, i.e. movement of individuals on the road network, can be stored efficiently if the contraction hierarchy data structure is present. When describing the compression process of raw trajectory data, we also prove that the compressed presentation is unique. Having the data compressed will also be beneficial for answering retrieval queries which ask for all trajectories intersecting a given space-time cube. We describe how to augment the contraction hierarchy data structure further to answer such queries quicker than state-of-the-art approaches.

Interestingly, contraction hierarchies can also be instrumentalized to visualize graph simplifications

as it was already demonstrated in the URAN framework. However, the URAN framework had some shortcomings which we address. We ensure that a simplified graph retains its original silhouette better and avoids some topological inconsistencies. Moreover, we apply similar techniques to visualize trajectory data sets within our framework PATHFINDER$^{\text{VIS}}$ which is also able to provide different flavors of visualization including heat-map based ones.

Acknowledgements

The work presented in this thesis was in part supported by the Deutsche Forschungsgemeinschaft (DFG) within the priority program 1894: Volunteered Geographic Information: Interpretation, Visualization and Social Computing.

This thesis could not have been written without the help of many people. First of all, I thank my advisor Prof. Dr. Stefan Funke. His guidance, advice, and encouragement have been invaluable throughout my master's and doctoral studies.

I am also grateful to my co-authors of the papers which are relevant for this thesis: André Nusser, Sabine Storandt, Lukas Baur and Claudius Proissl.

Moreover, I want to express my gratitude to all former and current members of the FMI for a great time and for many inspiring conversations. Special thanks to Armin Weiß and Felix Weitbrecht for proofreading.

Thanks also goes to the students Enis Spahiu, Patrick Lindemann and Shoma Kaiser for implementing a rudimental version of PATHFINDER$^{\text{VIS}}$.

Finally, I also like to thank my family Dietmar, Inge and Christian Rupp for their support.

Part I.

Prologue

Chapter 1

Preface

Navigation from one location A to another location B is a recurring task in logistics and everyday life. Naturally, one wants to travel from A to B on the shortest route possible. Note that the term *shortest* does not necessarily refer to geographical distance here but can also mean the shortest amount of time needed for traveling. Minimizing another measure like energy can also make sense when this resource becomes scarcer.

But in any case, determining the shortest path in for example, a road network, is not trivial since you have to exclude the possibility that there exists an even shorter path. Therefore, you basically have to know and consider every street in your travel network to confidently declare a path shortest. For the average human, this task may be possible when the navigation is restricted within their local village or small town but becomes infeasible for targets far away where they have never been before.

Fortunately, computers can store detailed maps of our road networks and there exist simple algorithms so they can compute our optimal travel route even on continental-sized networks. An exemplary result can be seen in Figure 1.1. Even though computers can answer such shortest path queries much faster and more reliable than humans by these simple algorithms, they may still need several minutes on the biggest networks.

To answer these queries instantly by human reaction time standards, i.e. within milliseconds, more sophisticated algorithms and frameworks were developed. One of these frameworks is the contraction hierarchies [GSSV12b] framework which stand at the center of this thesis and works by speeding up one of the simple algorithms. In particular, we will investigate the speed-up effect from a theoretical perspective.

Having electronic devices which can perform such navigation tasks are also capable of doing other related tasks. For example, standard mobile phones can identify their position by GPS, so traveling individuals can track their own movement. There are big commercial and non-commercial platforms which collect these trajectory data from their customers. Having a large set of trajectory data allows to discover movement patterns for different purposes. For example, if the data shows that some road is frequently part of trajectories at Monday mornings, individual drivers can then avoid this road at that time slot to avoid getting stuck in a likely traffic jam. For other actors such as urban planners, the knowledge about these movement patterns can help them to design road networks which are less susceptible to traffic jams from the get-go.

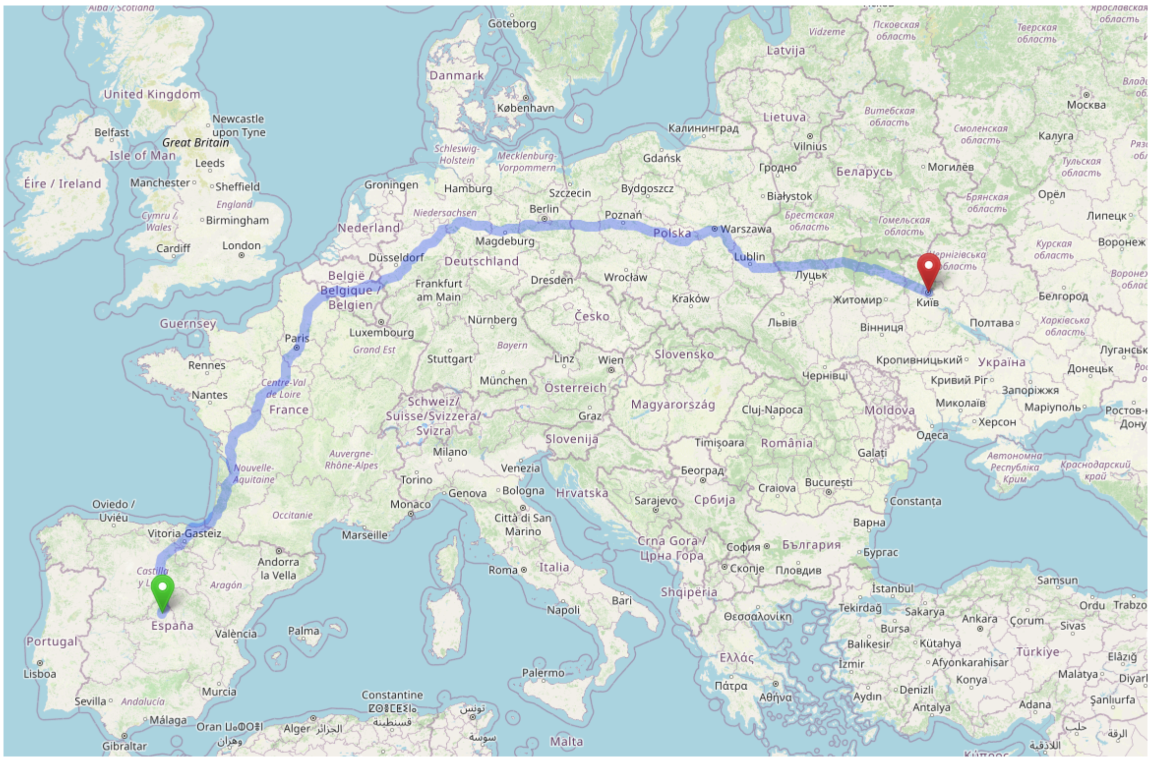


Figure 1.1.: Car route from Madrid to Kiev according to the navigation service of https://www.openstreetmap.org.

Moreover, mobile phones can also display these trajectories on a map. While showing a route on a map is already very handy when navigating, this is even more important when human analysts are tasked to find these patterns in large trajectory data sets. For that purpose, a visual representation of the trajectory data set has to be rich in information but should not be too cluttered. But removing clutter by simplification often comes at the cost of faithfulness with respect to the actual data. We will show how the concept of contraction hierarchies can be extended to boost these visualization capabilities too.

The rest of the thesis is structured as follows:

- Preliminaries: Firstly, we introduce standard graph theory notation. Then we explain a simple algorithm to compute shortest paths, Dijkstra's algorithm. We proceed by presenting advanced speed-up techniques as the bidirectional variant of Dijkstra's algorithm and the contraction hierarchies for which we also prove correctness. Moreover, we present hub labels, a different but related technique for computing shortest distances. As a basis for the theoretical analysis, we also introduce the notion of search spaces and explain the concept of separators.
- Theory: We first introduce the graph family of lightheaded grids which has many properties in common with real street networks. For these lightheaded grids, we prove strong lower bounds for the time and space complexity of contraction hierarchy and hub labels respectively. Both the result and the proof are the most significant contributions of this thesis. Moreover, we use the proof idea to develop a schema to construct instance-based lower bounds for arbitrary graphs. For this schema, we are able to prove that it is tight for certain graph families like balanced

ternary trees.

Another chapter picks up a proposed strategy for building a contraction hierarchy from the literature for which a upper bound on the number of nodes during the contraction hierarchy query has been proven. We show that their results are not as meaningful as they might appear by pointing out significant differences between the number of nodes and edges which have to be inspected during query time. In particular, we even show that their seemingly reasonably strategy based on separators yields no asymptotical speed-up over the simple Dijkstra's algorithm. We also show that the contraction hierarchy framework is not necessarily to blame by providing a strategy for sublinear edge search spaces.

- Practice: We show how trajectory data can be represented with the contraction hierarchy framework efficiently. We describe the algorithm for compression and show the uniqueness of the representation. Furthermore, having trajectory data in compressed representation will also help us to answer retrieval queries which ask for all trajectories intersecting a given space-time cube. For this purpose, we augment the contraction hierarchies data structure so that we can traverse and use it in a similar way as an R-Tree.

  In another chapter, we deal with visualization of trajectory data and simplifications of the road network. URAN, an already existing graph visualization framework based on contraction hierarchies, has some shortcomings concerning silhouette preservation and topological inconsistencies for which we make improvements. Furthermore, we present our framework PATHFINDER$^{\text{VIS}}$ which provides several options to derive compact visual representations for the result of a retrieval query.

- Finally, we give a summary and an outlook on potential future work.

Chapter 2 Preliminaries

2.1. Definitions

We have a set of nodes V where we also write n for the cardinality of V, i.e. $n := |V|$. In this work, the nodes are always associated with coordinates in the plane or on the surface of a sphere. Then there is a set of edges $E \subseteq V \times V$ with $m := |E|$. We use directed edges when direction matters conceptually and undirected edges otherwise. Undirected edges can be thought of a pair of two directed edges for technical purposes. Nodes and edges together define a graph $G(V, E)$.

For modeling different travel time on road networks, we assign each edge a weight. Formally, we define a cost function on the edges: $c : E \to \mathbb{R}^+$. If not explicitly defined, we assume a cost function which maps every edge to weight 1. We denote the shortest path distance between two nodes as $d : V \times V \to \mathbb{R}^+$.

An example of a graph with directed edges and explicit weight function can be see in Figure 2.1.

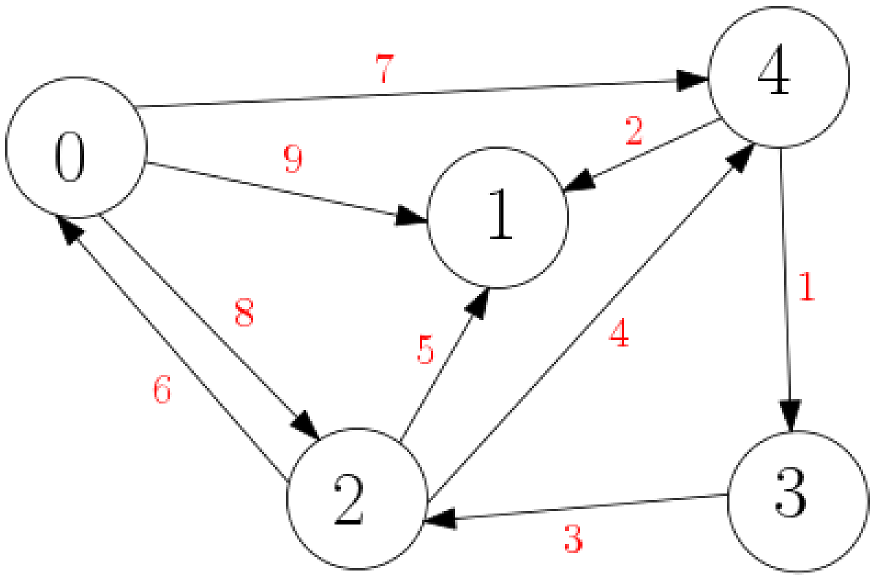


Figure 2.1.: A simple graph with directed edges and weights.

2.2. Dijkstra's Algorithm

Algorithm 2.1 Dijkstra's algorithm

```

dijkstra(V, E, cost, s, t) {
    M = MinHeap();
    d = initializeArrayWith(infinity, V.size());
    predecessor = initializeArrayWith(None, V.size());
    d[s] = 0;
    M.push (s, d[s]);

    while (!M.empty()) {
        active, priority = M.pop();
        // Target reached?
        if (active == t)
            return backtrack(predecessor, t);

        //ignore already settled nodes
        if (priority > d[active])
            continue;

        for (Edge (active, v): outEdges(active, E)) {
            tempDist = d[active] + cost(active, v);
                if (tempDist < d[v]) {
                d[v] = tempDist;
                predecessor[v] = active;
                M.push(v, tempDist);
            }
        }
    }
    return NULL;
}

backtrack (predecessor, t) {
    path  = {t};
    current_predecessor = predecessor[t];
    while (current_predecessor != None) {
        path.push_front(current_predecessor);
        current_predecessor = predecessor[current_predecessor];
    }
    return path;
}
```

For arbitrary graphs in general, Dijkstra's algorithm, also called Dijkstra for short, is a simple and efficient way to compute a shortest path between a given source s and target t. Dijkstra is basically a flooding algorithm which consecutively determines the cheapest remaining node reachable from the start node out of a set of candidate nodes. See algorithm 2.1 for pseudocode.

Different variants exist which handle the set of candidate nodes with different data structures. In most cases an implementation with a simple min-heap is the easiest to think about and is not much worse than the others performance-wise.

If we focus on special graph families however, like planar graphs or almost planar graphs which we encounter in real-world route planning, there exists a multitude of improvements over plain Dijkstra which are designed to take advantage of the planarity property and are in fact much faster in practice.

2.3. Bidirectional Dijkstra

As plain Dijkstra can terminate when the cost of the target node is settled, the search space of a successful run on a planar graph can be imagined as a ball of radius $d(s,t)$. Since all nodes and edges in the ball have to be inspected during runtime, one can think of the ball's surface area as computational cost. The idea of bidirectional Dijkstra is to reduce the search space using two balls with half the radius each, which results in half the surface area in total and therefore half the runtime. Although the termination condition is not as simple as stopping when both balls touch, it roughly works like that on street networks in practice.

As the Dijkstra from target t has to be executed in the reverse direction, implementations have to store the graph data in a way that not only outcoming but also incoming edges from a node can be retrieved quickly. Usually this is done by storing all edges twice, once sorted with respect to source and once with respect to target node.

2.4. Contraction Hierarchies

The contraction hierarchies (CH) approach [GSSV12a] computes an overlay graph in which so-called *shortcut edges* span large sections of shortest paths. This reduces the hop length of shortest paths and therefore allows a variant of Dijkstra to answer queries more efficiently.

The preprocessing is based on the so-called *node contraction* operation. Here, a node v as well as its adjacent edges are removed from the graph. In order not to affect shortest path distances between the remaining nodes, shortcut edges are inserted between two neighbors u and w of v, if and only if uvw was a shortest path (which can cheaply be checked via a plain Dijkstra run). The cost of the new shortcut edge (u,w) is the sum of costs of (u,v) and (v,w). In the preprocessing phase, all nodes are contracted conceptually one-by-one in some order which defines the rank of the nodes. To speed up the preprocessing phase in practice, a set of independent nodes can be contracted simultaneously in one round where the round number in which a node is contracted is the *level* of the node. Therefore, the notions of level and rank can be almost used interchangeably. We use both notions in different contexts to cover subtle differences but these are of rather technical nature. To refer to the level or rank of a node v, we write $l(v)$ and $\text{rank}(v)$, respectively. Moreover, for $l(v_1) < l(v_2)$ or $\text{rank}(v_1) < \text{rank}(v_2)$, we write $v_1 < v_2$ for short.

Having contracted all nodes, the new graph $G^+(V,E^+)$ contains all original edges of G as well as all shortcuts that were inserted in the contraction process. An edge $e = (v,w)$ – original or shortcut – is called upwards, if the level of v is smaller than the level of w, and downwards otherwise. We will prove in the next section the following property: For every pair of nodes $s,t \in V$, there exists a shortest path in G^+ that consists of a sequence of upward edges followed by a sequence of downward edges. This property allows us to search for the optimal path with a bidirectional Dijkstra only considering upward edges in the search starting at s, and only downward [1] edges in the reverse search starting at t. For the correct termination criterion, the length of the shortest path found so far has to be kept and updated as a tentative distance. The algorithm can then stop if the heap only contains nodes for which the distance to either source or target is more than the tentative distance. This reduces the search space significantly and allows for answering of shortest path queries within the *milliseconds* range compared to *seconds* on a country-sized road network. We call this algorithm the CH-Dijkstra.

[1]These downward edges could arguably also be seen as upwards from the perspective of target t in the reverse search space.

2.4.1. Search Spaces

For our theoretical results about performance later on, the notion of *direct search space (DSS)* (as defined in [FS15]) is useful. A node w is in $DSS(v)$, if on the shortest path from v to w all nodes have level at most level(w). Hence, w will be settled with the correct distance $d(v, w)$ in the CH-Dijkstra run. Moreover, the set of nodes which are actually settled is called *search space* $SS(v)$. Usually, $DSS(v) \subsetneq SS(v)$, as also nodes on monotonously increasing (w.r.t. level) but non-shortest paths are considered in $SS(v)$.

2.4.2. Query Complexity

With the term *query complexity* we refer to the number of processing steps in a full upward search. While sometimes the bidirectional search can be aborted earlier, random source-target queries tend to be overwhelmingly long-range and hence require a full upward search. As our results will be about average query complexity, occasional premature abortions are therefore negligible.

2.4.3. CH Correctness

It remains to prove that the CH-Dijkstra always finds a path which is shortest.

Proof. This is the case when there is a shortest path from source to target on which the node levels first increase and then decrease. An example of such a path can be seen in Figure 2.2. We prove the existence of such a path by contradiction: We assume such a path does not exist but we know that other shortest paths exist but do not fulfill the criteria. From those, take one path p which minimizes the number of edges. By the assumption, p must have a node v where both neighbors in the path have higher levels. As p is a shortest path, every subpath must also be a shortest path and hence the path between the neighbors over v. So when v is contracted in the construction process, a shortcut edge between the neighbors must have been added. We now can substitute the two edges between v to its neighbors in p by the shortcut edge to get a path with fewer edges which is a contradiction. □

2.4.4. Contraction Order

It is worth noting that the order in which nodes are contracted can be arbitrary but not surprisingly, the performance of the CH-Dijkstra is dependent on the contraction order. The theoretical part of this thesis is fully dedicated to explore that connection. In practice, heuristics are used to obtain decent contractions. A simple standard heuristic is to contract nodes according to a low *edge-difference* which is the number of new edges introduced minus the number of removed edges when contracting a node.

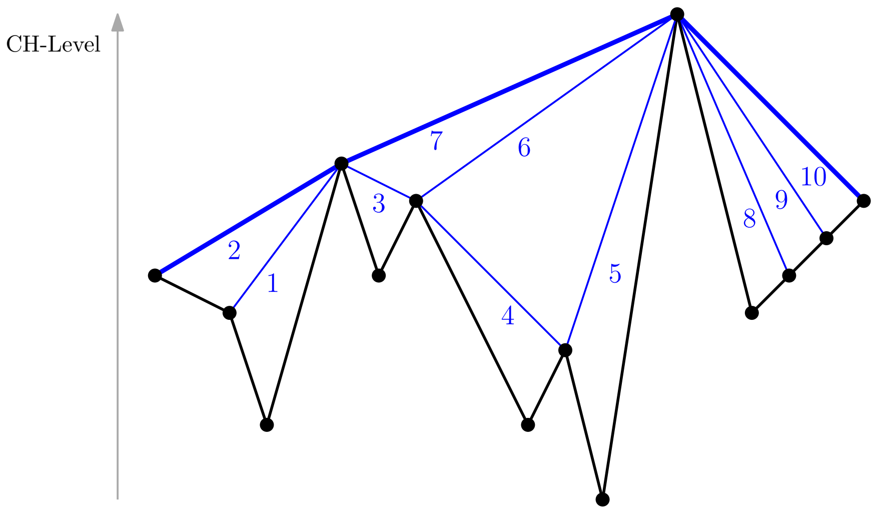


Figure 2.2.: The 3 bold blue edges form a shortest path which is first monotonously increasing, then decreasing respective to CH-level.

2.5. Hub Labels

Hub labelling (HL) [GPPR04] is a schema to answer shortest path distance queries which differs fundamentally from graph search based methods. Here, the idea is to compute for every $v \in V$ a *label* $L(v)$ such that for given $s, t \in V$ the distance between s and t can be determined by just inspecting the labels $L(s)$ and $L(t)$. All the labels are determined in a preprocessing step (based on the graph G), later on, the graph G can even be discarded. There have been different approaches to compute such labels (even in theory); we will be concerned with hub labels that work well for road networks, following the ideas in [ADGW12]. To be more concrete, the labels we are interested in have the following form:

$$L(v) = \{(w, d(v, w)) : w \in H(v)\}$$

Here we call $H(v)$ a set of *hubs* – important nodes – for v. The hubs must be chosen such that for any s, t, the shortest path from s to t intersects $H(s) \cap H(t)$. This is also called the *cover* property since all shortest paths need to be covered by the hubs.

If such label sets are computed, the computation of the shortest path distance between s and t boils down to determining the node $w \in H(s) \cap H(t)$ minimizing the summed distance. If the labels $L(.)$ are stored lexicographically sorted, this can be done in a cache-efficient manner in time $O(|L(s)| + |L(t)|)$.

Knowing about CH, there is a natural way of computing such labels [ADGW12]: simply run an upward Dijkstra from each node v and let the label $L(v)$ be the settled nodes with their respective distances. Clearly, this yields valid labels since CH answers queries exactly. The drawback is that the space requirement is quite large; depending on the metric and the CH construction, one can expect labels consisting of several hundreds to thousands node-distance pairs. It turns out, though, that many of the labels created in such a manner are useless as they do not represent shortest-path distance (as we restricted ourselves to a search in the upgraph only); pruning out those (which means reducing them to the direct search space of the respective node) typically reduces the label size by a factor of around 4. A distance query from some source to some target can then be answered in the *microseconds* range on country-sized networks.

2.6. Separators

Separators are a generally useful concept in graph theory. A separator is a set of nodes $S \subset V$, such that the remaining graph is split in two separate components $V \setminus S = A \dot{\cup} B$ where $\nexists (a, b) \in E$ for $a \in A \wedge b \in B$. Many divide-and-conquer algorithms on graphs make use of separators. Notably, small separators of $O(\sqrt{n})$ are guaranteed [LT79] for planar graphs and real road networks are almost planar [ZC09].

2.7. R-Tree

The R-tree [Gut84] is a basic query structure from computational geometry. Given a set of rectangles $\mathcal{R}$, an R-tree allows to quickly retrieve all $R \in \mathcal{R}$ having non-empty intersection with a query rectangle Q. In its tree structure, each node is associated with a bounding box of the bounding boxes of its descendants, the rectangles from $\mathcal{R}$ are stored at the leaves. A query with rectangle Q is processed recursively, always exploring nodes whose bounding boxes have a non-empty intersection with Q.

Part II.

Theory

Chapter 3
Motivation

In practice it has been experimentally verified countless times that contraction hierarchies provide a massive speedup for routing on real-world street networks [BDG+16], often around 3 orders of magnitude. In contrast, the speedup is much less analyzed from a theoretical perspective. The few theoretical contributions are almost all about upper bounds.

While the problem of computing shortest paths in general graphs with non-negative edge weights seems to have been well understood already decades ago, the last 10–15 years have seen tremendous progress when it comes to the specific problem of efficiently computing shortest paths in *real-world road networks*. Here the idea is to spend some time for preprocessing where auxiliary information about the network is computed and stored, such that subsequent queries can be answered much faster than standard Dijkstra's algorithm. One might classify most of the employed techniques into two classes: ones that are based on *pruned graph search* and others that are based on *distance lookups*. Most approaches fall into the former class, e.g., reach-based methods [Gut04], [GKW06], highway hierarchies [SS12], arc-flags-based methods [BD10], or contraction hierarchies (CH) [GSSV12a]. Here, Dijkstra's algorithm is given a hand to ignore some vertices or edges during the graph search. The speed-up for road networks compared to plain Dijkstra's algorithm ranges from one up to three orders of magnitude ([GSSV12a; Gut04]). In practice, this means that a query on a country-sized network like that of Germany (around 20 million nodes) can be answered in less than a *millisecond* compared to the multiple seconds of Dijkstra's algorithm. While these methods directly yield the actual shortest path, the latter class is primarily concerned with the computation of the (exact) distance between given source and target – recovering the actual path often requires some additional effort. Examples of such distance-lookup-based methods are transit nodes [BFSS07], [BFM+07] and hub labels (HL) [ADGW12]. They allow for the answering of *distance queries* another one or two orders of magnitude faster.

There have also been attempts at theoretically explaining the impressive practical performance of these speed-up schemas. These approaches first identify certain properties of a graph, which supposedly characterize 'typical' inputs in the real world and then show that for graphs satisfying these properties, certain speed-up schemas have guaranteed query/construction time or space consumption. Examples of such graph characterizations are given via *highway dimension* [AFGW10], *skeleton dimension* [KV17], or *bounded growth* [BFS18] which we look at in more detail in the next section. It is important to note that almost all contributions in this field are concerned with *upper bounds*.

Chapter 4 addresses lower bounds, not only for CH but also for Hub labels. The insights of our theoretical lower bound analysis also allows us to devise a schema to compute *instance-based* lower bounds, that is, for a given concrete road network, we algorithmically compute a lower bound on the average query time. Note that such an instance-specific lower bound is typically much stronger than an analytical lower bound.

Many of the existing contributions give upper bounds for the number of nodes which are explored in the CH-query and indicate that it is meaningful to upper bound the computation cost for a CH-query by assuming that the number of explored edges is closely related to the number of explored nodes. In the following, we also speak of these sets of explored nodes and edges as nodes search space and edge search space, respectively.

In Chapter 5, we will show that for contraction heuristics proposed by some of these papers, the difference between the sizes of those search spaces are indeed significant up to the point where their heuristics fail to improve plain Dijkstra. Note that for deriving lower bounds, the distinction between those search spaces is not important as a lower bound for either of these search spaces implies a lower bound for the query time. Speaking of search spaces in the context of hub labels is also unproblematic since the meaning is unambiguous there.

3.0.1. Related Work

In [Mil12], APX-hardness of finding a contraction order with minimal number of shortcuts was proven by a reduction from the feedback arc set problem. Although not proven, it's therefore almost certain to assume that optimizing with respect to other criteria like search space sizes is also NP-hard and that is why in practice heuristics are applied.

One of the first theoretically sound concepts for speed-up analysis was *reach* by [Gut04]. The fundamental idea of *reach* is that some edges are more important than others for shortest paths. Formally, the reach of a node v is defined as $\text{reach}(v) = \max_{(s,t)|v \in SP(s,t)} \min\{\text{dist}(s,v), \text{dist}(v,t)\}$. So the reach of a node is high if it is in the middle of a long shortest path. During a Dijkstra, nodes can sometimes then be pruned depending on their reach value leading to a speed-up.

In [AFGW10], a graph property called *highway dimension* is proposed to analyze shortest path speed-up schemas. Intuitively, the highway dimension h of a graph is small if there exist sparse local hitting sets for shortest paths of a certain length. More formally, let $H_{v,r}$ be the smallest hitting set for all shortest paths of length at least r within a given ball around a node v of radius $4r$. Then the highway dimension [AFGW10] h is defined as $h = \max_{v \in V, r \in \mathbb{R}} |H_{v,r}|$. For contraction hierarchies and hub labels, a query time of $O(h \log n)$ was proven (using an NP-hard preprocessing phase; polynomial time preprocessing increases this by a $\log h$ factor). While one might hope that real road networks exhibit a 'small' highway dimension, e.g., constant or polylogarithmic, it is known that $h \in \Omega(\sqrt{n})$ holds for grids. Moreover, it is not feasible to compute highway dimension for large graphs in practice as this is known to be NP-hard [FFKP18].

In one of the few contributions which is concerned with lower bounds [Whi15], White constructs for any given highway dimension h a family of graphs $G_{t,k,q}$ (as introduced in [Mil12]) of highway dimension h, such that hub labelling requires a label size of $\Omega(h \log n)$ and CH a query time of $O((h \log n)^2)$. Unfortunately, the graphs $G_{t,k,q}$ according to the author himself "are not representative of real-world graphs. For instance, the graphs do not have small separators and are not planar". In fact this could be more an indication of the unsuitability of the notion of highway dimension to characterize real-world road networks rather than a weakness of [Whi15].

Motivated by the shortcomings of highway dimension, another graph characteristic called skeleton dimension [KV17] was introduced. For a formal definition according to [BFS18], let $\tilde{T}_u$ be the geometric realization of the shortest path tree T_u of some vertex u. Intuitively, $\tilde{T}_u$ consists of infinitely many infinitely short edges such that for every edge vw of T_u and any $\alpha \in [0, 1]$ there is a vertex in $\tilde{T}_u$ at distance α from v and distance $1-\alpha$ from w. The skeleton T_u^* of T_u is now defined as the subtree of $\tilde{T}_u$ induced by all vertices v that have a descendant w satisfying $d_v(w) \geq \frac{1}{2} d_u(v)$. Then the skeleton dimension k of a graph G is defined as $k = \max_{u \in V, r>0} x_{u,r}^*$ where $x_{u,r}^*$ denotes the number of vertices in T_u^* that are at distance r from u. Rather informally, a skeleton dimension of k means that each shortest-path tree is built around a core skeleton with at most k branches at a given distance range while the rest of the branches are relatively short. For hub labels, the skeleton dimension k [KV17] has been instrumented to prove a search space size of $O(k \log n)$. Still, for grids, we have $k \in \Omega(\sqrt{n})$.

For the more established graph characteristic treewidth, CH were analyzed for graphs with treewidth t and a node search space size of $O(t \log n)$ was shown [BCRW13]. Yet, for grids we again have $t \in \Omega(\sqrt{n})$.

Finally, the bounded growth model was introduced in [FS15]. A graph family $G(V, E)$ has bounded growth if there is a constant c such that for any radius r and any node u

$$|\{v \in V : d_{u,v} = r\}| \leq cr.$$

This implies $|\{v \in V : d_{u,v} \leq r\}| \leq cr(r+1)/2$ which mimics the area growth in $\mathbb{R}^2$ when increasing the radius of a circle. Limiting this area growth of the plane while not having to enforce strict planarity itself is the basic idea of capturing the essential characteristics of real world networks here. This model allowed to bound the node search space size of CH queries to expected $O(\sqrt{n} \log n)$ for realistic graphs including grids.

Chapter 4

A Lower Bound for the Query Phase of Contraction Hierarchies and Hub Labels and a Provably Optimal Instance-based Schema

4.1. Introduction

This chapter is based on joint work with Stefan Funke. A conference version [RF20] was published in the proceedings of the 15th International Computer Science Symposium in Russia (CSR 2020) and an extended one [RF21] in the Special Issue on “Selected Algorithmic Papers From CSR 2020” of the *Algorithms* journal.

Here, we are concerned with two specific speed-up techniques, namely contraction hierarchies [GSSV12a] and hub labels [ADGW12], and provide *lower bounds*.

As grid-like substructures are quite common in real-world road networks, see Figure 4.1, one might ask whether better upper bounds than $O(\sqrt{n}\log n)$ for such networks are even possible or a polylogarithmic upper bound could be shown via more refined proof or CH construction techniques. Our work settles this question for contraction hierarchies as well as hub labels up to a logarithmic factor. We show that for CH, no matter what contraction order is chosen, and for HL, no matter how the hub labels are generated, there are grid networks for which not only the worst-case but even the *average* number of nodes to be considered during a query (which we call *search space*) is $\Omega(\sqrt{n})$.

The insights of our theoretical lower bound analysis also allow us to devise a schema to compute *instance-based* lower bounds, that is, for a given concrete road network, we algorithmically compute a lower bound on the average search space size. Note that such an instance-specific lower bound is typically much stronger than an analytical lower bound.

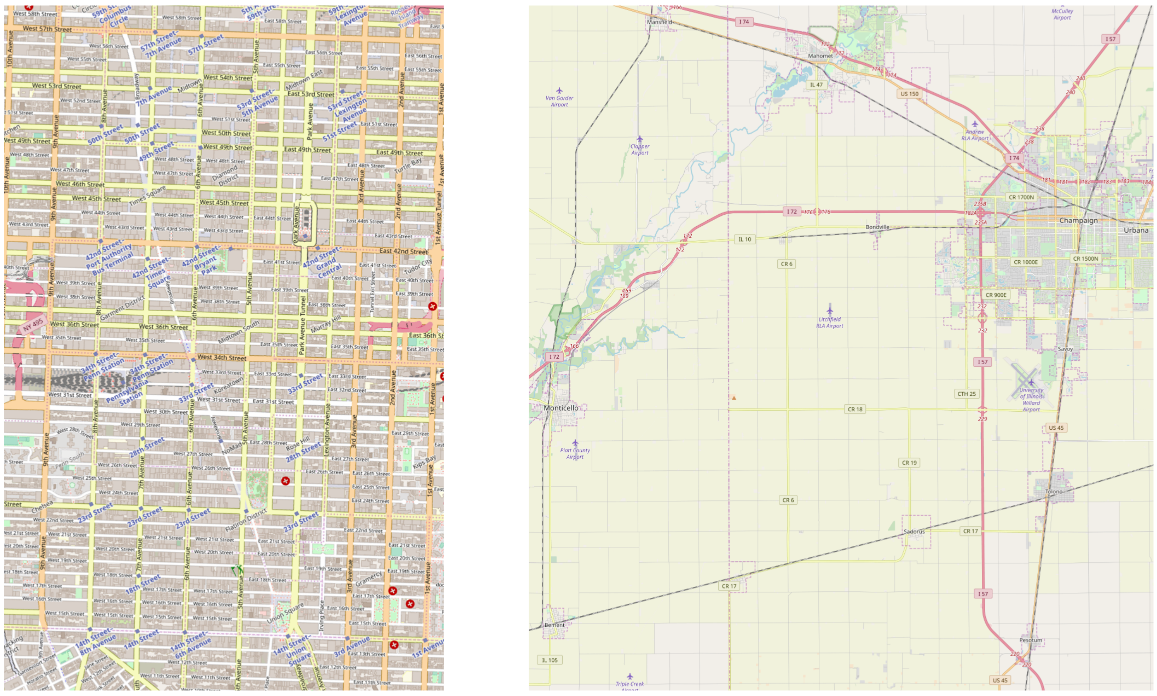


Figure 4.1.: Grid-like substructures in real networks (Manhattan on the left, Urbana-Champaign on the right) (by OpenStreetMap).

4.1.1. Related Work

Specifically for planar graphs, the search space is $O(\sqrt{n})$ by combining the planar separator Theorem [LT79] with nested dissection [BCRW13]. Therefore, our lower bound for the presented grid graph will be tight.

For transit nodes [BFM+07], instance-based lower bounds based on an LP formulation and its dual were derived in [EF12]. We are not aware of results regarding instance-based lower bound constructions for HL or CH.

Our Contribution and Outline

In this chapter we prove a lower bound on the search space (and hence the processing time) of the query phase of contraction hierarchies as well as hub labels. More concretely, we define so-called *lightheaded grids* for which we show that the average direct search space size is $\Omega(\sqrt{n})$, irrespective of what contraction order or whatever hub labelling schema was employed. This obviously lower bounds the number of nodes that have to be considered during CH-Dijkstra. Based on a $\sqrt{n} \times \sqrt{n}$ grid, our graph is planar and has bounded degree. Our lower bound applies to both CH [GSSV12a] and HL [ADGW12] schemas. Based on our insights from the lower bound proof, we also show how to construct *instance-based* lower bounds algorithmically. Our experiments indicate that current CH constructions yield search space sizes close to optimal.

We first present and prove our main theoretical result. Then we also show how to algorithmically construct instance-based lower bounds and present some experimental results. We conclude by proving that an instance-based schema of ours is optimal for ternary trees in the sense that it can yield tight lower bounds.

4.2. Theory: A Lower Bound Construction

In this section we first provide a simple graph construction which is essentially a slightly modified $\sqrt{n} \times \sqrt{n}$ grid graph with some desirable properties. Then we provide a lower bound on the direct search space size of *any* contraction order via an amortized analysis. A slight variation of the analysis also yields a lower bound for any hub labelling schema. For the sake of simplicity, we assume without loss of generality that n is always a square number and a multiple of 4 for our analysis. Furthermore, our construction assumes an *undirected* graph but generalization to the directed case is quite straightforward.

4.2.1. The Lightheaded $\sqrt{n} \times \sqrt{n}$-grid G_{lh}

The basis of our construction is a regular $\sqrt{n} \times \sqrt{n}$ grid with uniform edge costs. We then modify only costs of the *horizontal* edges such that they become 'lighter towards the head', hence the name *lightheaded grid*. More precisely, the horizontal edges in row i ($i = 0, 1, \ldots, \sqrt{n} - 1$, counted from top to bottom) have cost $1 + i\epsilon$ for some small enough $\epsilon < 1$. See Figure 4.2, for an example.

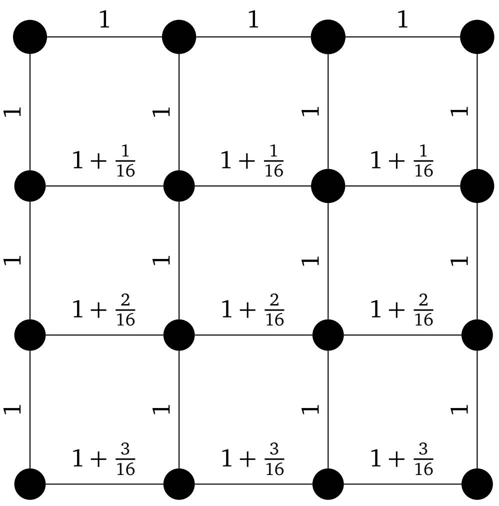


Figure 4.2.: Lightheaded grid for $n = 16, \epsilon = 1/16$.

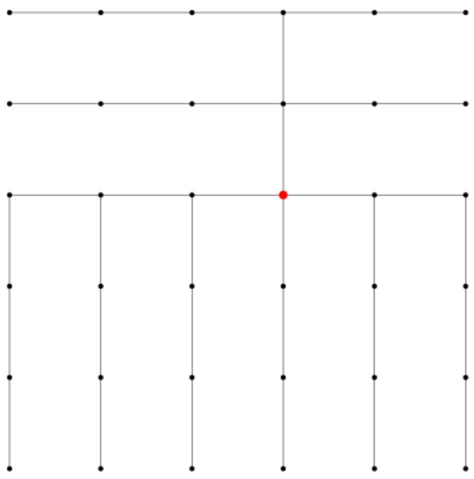


Figure 4.3.: Shortest path tree from red source.

It is interesting to note that the lightheaded grid structure can arise quite naturally in a real-world application: to route over large distances on the globe, a common approach is to discretize the planet along constant latitudes and longitudes, which can be seen in Figure 4.4, to build a graph on which the mentioned routing algorithms can operate If source and target are both placed in the northern Atlantic Ocean as indicated with the red axis-aligned rectangle, routing takes place on a lightheaded grid.

4.2.2. Shortest Path Trees in Lightheaded Grids

For small enough choice of ϵ, the following Lemma holds:

Lemma 4.2.1
For $\epsilon < 1/n$, the shortest path between some s and t in a lightheaded grid G_{lh} is unique and always consists of a horizontal and a vertical part, where the horizontal part is on the height of the higher of the nodes s and t.

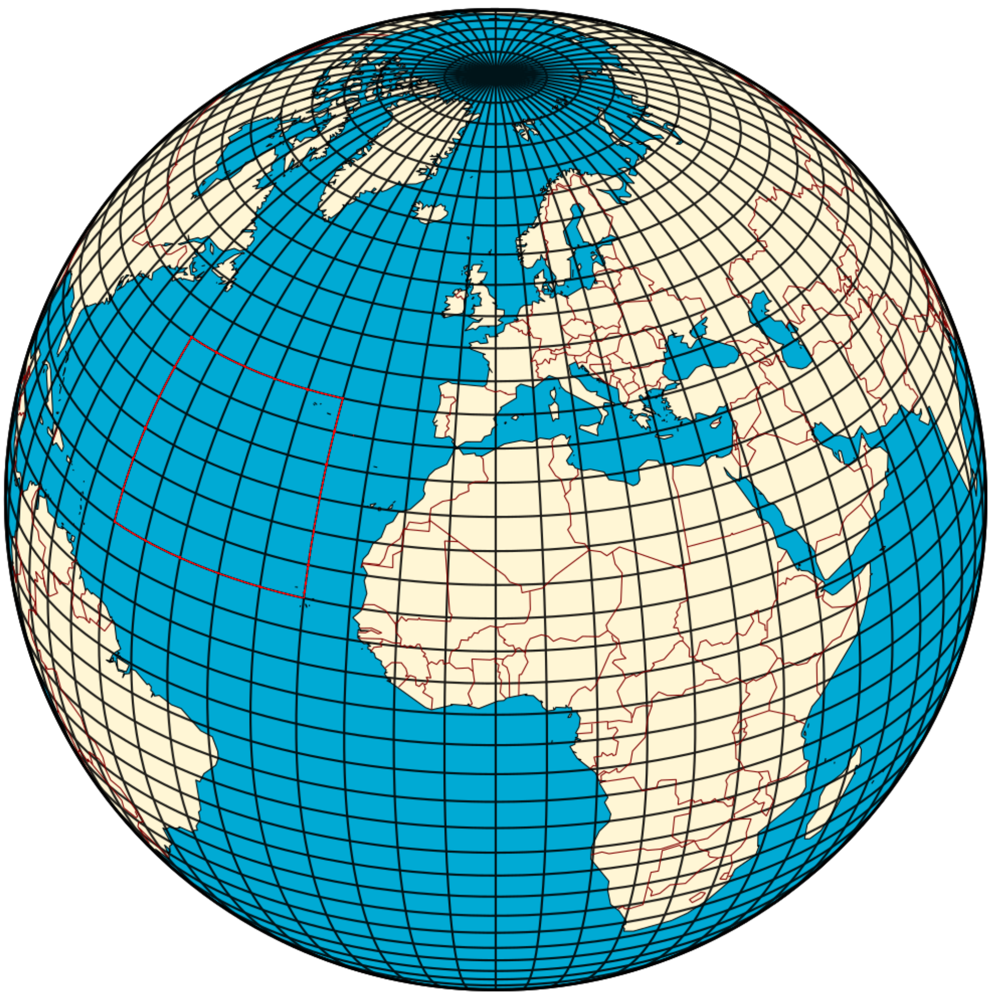


Figure 4.4.: Light-headed grid appearing in discretization of the globe along constant latitudes and longitudes) (Based on a image by Hellerick - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=26737079).

Proof. If a shortest path between s and t in the *unweighted* grid has cost d, the modified edge costs add a cost of less than 1 for the same path, hence all shortest paths in G_{lh} have (unweighted) Manhattan distance d. Let $d = d_v + d_h$ where d_v is the vertical, d_h the horizontal distance. Any shortest Manhattan path must be composed of d_h horizontal and d_v vertical segments. In G_{lh}, horizontal edges towards the top have lower cost, hence the shortest path must have all its horizontal edges on the height of the higher of the nodes s and t. □

See Figure 4.3 for an illustration of a shortest path tree in the lightheaded grid. Observe that for all targets in the lower left and the upper right parts, the shortest path has an upper left corner, whereas for all targets in the upper left and lower right part, it has an upper right corner.

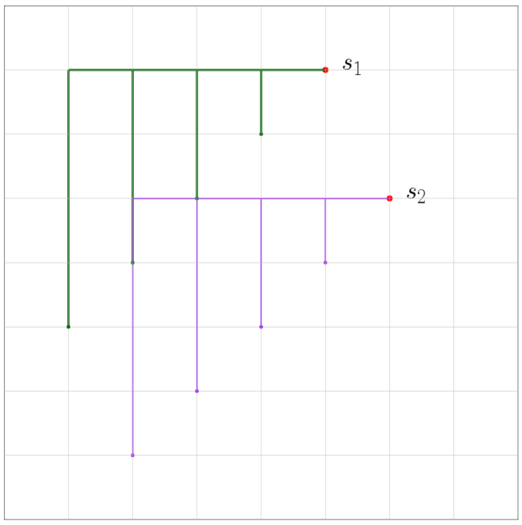


Figure 4.5.: Relevant lower left parts of shortest path trees rooted in top right quarter.

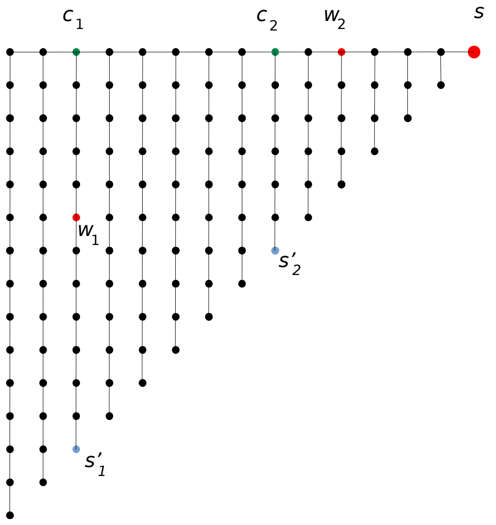


Figure 4.6.: Subtree in the lower left of a shortest path tree and charging argument.

4.2.3. Lower Bounding the Direct Search Space

Let us now assume that the contraction hierarchy has been created with an *arbitrary* contraction order. We will show that no matter what this contraction order is, the average size of the direct search space is $\Omega(\sqrt{n})$.

In our analysis we only consider shortest path trees rooted in the top right quarter of the grid (there are $n/4$ of them). For these shortest path trees, their lower left part always contains a subtree of size $\Theta(n)$ as depicted in Figure 4.5.

The idea of the analysis is to identify pairs (x, v) such that $v \in DSS(x)$. We will consider each of the shortest path trees rooted at some node s in the top right quarter and for each identify $\Theta(\sqrt{n})$ such pairs (not necessarily with $x = s$). The main challenge will be to make sure that no double counting of pairs occurs when considering all these shortest path trees.

Let us focus on one shortest path tree rooted at s and the subtree of the lower left part as shown in Figure 4.6. By construction, we have $\Theta(\sqrt{n})$ vertical branches in this subtree. Consider one branch, spawning at its corner node c. Furthermore, let s' be the node in the branch which has the same hop distance to c as c to s. One can think of s being mirrored at c, see Figure 4.6. Let w be the highest-CH-level node on the shortest path from s to s'. There are two cases: (a) w lies on the vertical branch including c (this is depicted as w_1 in Figure 4.6). (b) w lies on the horizontal part of the shortest path from s to s' excluding c (this is depicted as w_2 in the Figure). In case (a) we generate (s, w_1) since obviously $w_1 \in DSS(s)$. In case (b), we cannot simply generate (s, w_2) since the same pair might be identified when considering other branches in the shortest path tree from s, leading to double counting. But we also know that w_2 lies in the direct search space of s'_2. Intuitively, we charge s'_2 by generating the pair (s'_2, w_2).

So when considering the shortest path tree of a node s we generate exactly one pair (x, v) for each of the $\Theta(\sqrt{n})$ vertical branches to the lower left. Let us first show that we did not double count in this process.

Lemma 4.2.2
No pair (x, v) is generated more than once.

Proof. Consider a pair (s, v) that was generated according to case (a), i.e., s lies to the upper right or the right of v. Clearly, the same pair cannot be generated according to case (a) again, since the vertical branch in which v resides is only considered once from source s. But it also cannot be generated according to case (b), since these pairs always have s to the lower left of v.

A pair (s, v) generated according to case (b) has s to the lower left of v, hence cannot be generated by case (a) as these pairs have s to the upper right or right of v. As (s, v) was generated according to case (b), it was generated when inspecting the shortest path tree from a source s' which is the node s mirrored at the corner vertex of the shortest path from s to v. But this source s' is uniquely determined, so (s, v) can only be generated when the shortest path tree rooted at s' with the vertical branch containing v was considered. □

Now we are ready to prove the first main result of this chapter.

Theorem 4.2.3
The average direct search space of G_{lh} is $\Omega(\sqrt{n})$.

Proof. In our process we considered $n/4$ shortest path trees, in each of which we identified $\Omega(\sqrt{n})$ pairs (x, v) where $v \in DSS(x)$ and no pair appears twice. Hence, we have identified $\Omega(n\sqrt{n})$ such pairs, which on average yields $\Omega(\sqrt{n})$. □

4.2.4. Lower Bounding of Hub Label Sizes

Note that the above argument and proof can be modified to also cover label sizes in a hub labelling schema. Assume hub labels according to an arbitrary hub labelling schema have been constructed. Then, when considering the shortest path tree rooted at s and the node s' in a vertical branch, we define w to be a node in $H(s) \cap H(s')$. A pair (x, v) corresponds to $v \in H(x)$. Exactly the same arguments as above apply and we obtain the following second main result:

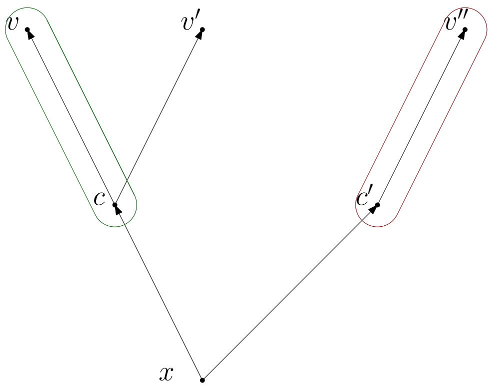
Theorem 4.2.4
The average hub label size of G_{lh} is $\Omega(\sqrt{n})$.

4.3. Practice: Instance-based Lower Bounds

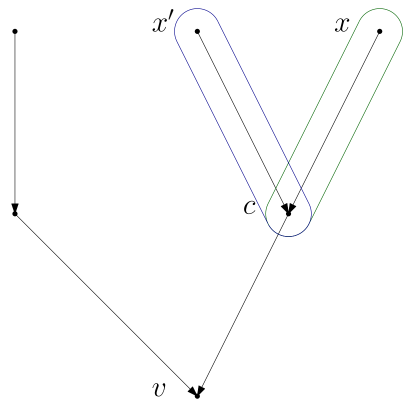
Our theoretical lower bound as just proven only applies to the lightheaded grid G_{lh} as defined before. Yet, even though similar substructures appear in real-world road networks, see Figure 4.1, the typical road network is certainly not a lightheaded grid and hence our lower bound proof does not apply.

Still, we can use the insights from the lower bound proof to construct *instance-based* lower bounds algorithmically. Concretely, for a given road network instance, we aim at computing a certificate which *for this instance* proves that the average search space size of a CH query, or the average hub label size cannot be below some lower bound, no matter what CH or HL construction was used.

Note that while for the previous lower bound proof we assumed an undirected graph for sake of a simpler exposition, we will now also include the more general case of a directed graph. To address the bidirectional nature of the CH-Dijkstra here we now also have to differentiate between forward and backward search space. In the same vein we refer to the forward and backward shortest path



(a) Forward shortest path tree $SPT_{\rightarrow}(x)$: The witness triple (x,c,v) where the relevant nodes for the conflict check are in the green bubble. There is no conflict with (x,c,v'') and its relevant nodes in the red bubble.



(b) $SPT_{\leftarrow}(v)$: The green witness triple (x,c,v) of the left Figure shown in the backward tree of v. It is in conflict with the blue triple (x',c,v) because they share the node c.

Figure 4.7.: Examples for witness triples in shortest path trees.

tree as $SPT_{\rightarrow}(v)$ and $SPT_{\leftarrow}(v)$, respectively. Both the CH as well as the HL schema can be easily generalized to directed graphs; in case of HL, compute for each node two labels, an *outlabel* $L^{out}(v)$ storing distances *from* v to hubs and an *inlabel* $L^{in}(v)$ storing distances from hubs *to* v. A query from s to t is then answered by scanning the outlabel $L^{out}(s)$ and the inlabel $L^{in}(t)$. CH also generalizes in a straightforward manner to the directed case, see [GSSV12a].

4.3.1. Witness Triples

In our lower bound proof for the lightheaded grid we identified pairs (x,v) such that $v \in DSS(x)$, making use of a concrete (but arbitrary) CH (or HL) for the lightheaded grid to actually identify those pairs (x,v). We cannot do this for a given instance of a road network, since we would have to consider all possible CH/HL constructions. So instead of pairs (x,v) let us now try to identify *witness triples* (x,c,v) where c is again a node on the (directed) shortest path from x to v. The intuition for (x,c,v) is the following: On the shortest path $\pi(x,v) = x \dots c \dots v$, some node of the suffix $c \dots v$ of π must be in the forward search space of x, or some node on the prefix $x \dots c$ must be in the backward search space of v. So it must be emphaccounted for at least once in total. This intuition mimics the proof idea in the previous section but also allowing for *directed* graphs and leaving the choice of c open.

In the following, we sometimes treat paths just as sets of nodes to simplify presentation. Let us first define the notion of a conflict between two triples.

Definition 4.1 (conflict between triples)
We say two triples (x,c,v) and (x',c',v') are in conflict *if at least one of the following conditions holds true:*

1. $x = x'$ *and* $\pi(c,v) \cap \pi(c',v') \neq \emptyset$
2. $v = v'$ *and* $\pi(x,c) \cap \pi(x',c') \neq \emptyset$

See Figure 4.7a for a non-conflicting and 4.7b for a conflicting example. Our goal will be to identify a set W of *conflict-free* witness triples. Moreover, W should be as large as possible. The following Lemma proves that the size of such a set W lower bounds the average search space and label size.

Lemma 4.3.1
If W is a set of conflict-free witness triples, then $|W|$ lower bounds the sum of (backward and forward) search spaces of all nodes in the network.

Proof. Consider a triple (x, c, v) and the following two cases:

1. it accounts for a node in $\pi(c, v)$ in the forward search space of x. Nodes in the forward search space of x can only be accounted for by triples (x, c', v'). But since $\pi(c, v) \cap \pi(c', v') = \emptyset$ due to W being conflict-free, we have not doubly accounted for it.
2. it accounts for a node in $\pi(x, c)$ in the backward search space of v. Nodes in the backwards search space of v can only be accounted for by triples (x', c', v). But since $\pi(x, c) \cap \pi(x', c') = \emptyset$ due to W being conflict-free, we have not doubly accounted for it. □

For any witness set W, $|W|/(2n)$ yields a lower bound on the average size of an in- or outlabel in case of the HL schema, and on the average size of the direct search space from a single node in case of the CH schema.

So it remains to compute a large conflict-free witness set W. Theoretically, one could enumerate all potential triples, build the conflict graph and use Integer Linear Programming (ILP) techniques to identify a conflict-free subset amongst the triples. A similar approach was conducted in [EF12] to compute lower bounds for transit nodes constructions. However, enumerating all potential triples in $V \times V \times V$ here seems somewhat inefficient (having to inspect $\Theta(n^3)$ such triples) on its own and employing the ILP techniques on top of it even more so. Hence, in the following we propose a simple greedy heuristic. Again, let us emphasize that the resulting lower bound holds for *any* CH/HL construction for the given road network.

4.3.2. Generation of Witness Triples

In the following we will describe two strategies for generating witness triples. The Fixed-Lengths method is based on an explicit representation of all shortest path trees with the advantage that the witness triple candidates are easily checked for conflict. The Peak-Node method builds upon precomputed hub labels and requires little working space, yet requires extensive and non-trivial conflict checks of the triples. Hence we also devise a simple method to check for conflicts which does not require explicit path representations.

4.3.2.1. Method A: Fixed Lengths

The high-level idea is as follows: We first compute and store all $2n$ shortest path trees within the network (both forward and reverse). We then enumerate candidate shortest paths in increasing hop length [1] $\ell = 1, 3, 7, 15, \ldots$ (always doubling and adding 1) from the shortest path trees, greedily adding corresponding triples to W in case no conflict is created. As center node c, we always choose the center node of a candidate path (in terms of hops), see, e.g., Figure 4.7a.

This specific choice for c as well as the decision to always roughly double the lengths of the considered candidate paths was motivated by the observation that in practice when considering

[1]We define hop length as the number of nodes here.

Algorithm 4.1 The algorithm to find a set of non-conflicting witness triples.

```
1:  procedure findWitnesses($G$)
2:      $SPTs \leftarrow$ ComputeShortestPathTrees($G$)
3:      $W \leftarrow \emptyset$
4:      $\ell \leftarrow 1$
5:      while $\ell \leq Diameter(G)$ do
6:          $WC \leftarrow collectCandidatesOfLength(SPTs, \ell)$
7:          while $WC \neq \emptyset$ do
8:              $(x, c, v) \leftarrow WC[0]$
9:              $W \leftarrow W \cup (x, c, v)$
10:             $WC \leftarrow WC \setminus (x, c, v)$
11:             pruneConflictingTriples($WC, (x, c, v)$)
12:         end while
13:         $\ell \leftarrow 2\ell + 1$
14:     end while
15:     return $W$
16: end procedure
```

a candidate triple (x, c, v), checking for conflict all triples $(.,.,v)$ and $(x,.,.)$ already present in W becomes quite expensive for large W. Intuitively, those decisions ensures that triples with different hop lengths can never be in conflict at all.

Our actual greedy algorithm is stated in Algorithm 4.1. Since we made sure that conflicts can only occur between triples with identical hop length, we can restrict the check for conflicts to the candidate set of triples of identical hop length. For an example of our greedy heuristic execution, look at the shortest path trees in Figure 4.7: Let us assume we collected the set of candidates for length 3 which contains (x, c, v), (x, c, v'), (x, c', v'') and (x', c, v) beside other triples. We now pick the candidate (x, c, v) and add it to W. Clearly, we remove this from our candidate set but we also have to remove (x, c, v') because it would lead to a conflict in the forward shortest path tree $SPT_{\rightarrow}(x)$ and also remove (x', c, v) because of the conflict in the backward tree $SPT_{\leftarrow}(v)$.

Note that after computation of the shortest path trees, the collection of candidates can be organized in such a manner that each tree only needs to be traversed once. Moreover, the candidates can be arranged in buckets with respect to the shortest path trees and the center nodes. Note though, that each candidate is represented in two buckets. As only candidates within one bucket need to be checked for conflicts, this leads to a reduced number of comparisons.

Storing and computing the shortest path trees requires $\Theta(n^2)$ space and $\Omega(n(n \log n + m))$ time when using Dijkstra's algorithm (here, $m = |E|$). Generation and pruning of candidate triples can be bounded by $O(n^3)$, yet in practice the computation of the shortest path trees clearly dominates the running time.

4.3.2.2. Method B: Peak nodes

Another strategy to generate witness triples uses CH-based hub labels. Here, the idea is to enumerate triples (x, c, v) with their center c being the highest-CH-level node on the shortest path from x to v. This approach is most efficiently implemented by enumerating via the the center c, i.e., determining all nodes x which have c in their outlabel and all nodes v which have c in their inlabel. If c is the highest-CH-level node on the shortest path $\pi(x, v)$, we have a valid triple candidate (x, c, v). Intuitively, high-CH-level nodes are 'important' junctions in the road network and are hence a natural choice for center points occurring in many non-conflicting witness triples. The great advantage of this method is the modest space consumption, which is essentially only the storage required for the hub labels. In contrast to the previous method, though, conflict checks of triples with coinciding source or target always have to be performed.

We want to emphasize that even though this construction method relies on precomputed hub labels, the resulting lower bound holds for *any* hub labelling schema.

4.3.2.3. Efficient Conflict Check

Depending on the method for generating witness triple candidates, an efficient check for conflict is necessary. Our method A can already exclude many conflict tests, yet method B essentially has to check every candidate pair (x, c, v) and (x, c', v'). The following consideration allows for a fast evaluation of the test which does not require explicit path representations:

$$\begin{align} & (x, c, v) \text{ and } (x, c', v') \text{ are in conflict} \tag{4.1}\\ \iff & \pi(c, v) \cap \pi(c', v') \neq \emptyset \tag{4.2}\\ \iff & \exists k : k \in \pi(x, v) \wedge k \in \pi(x, v') \wedge c, c' \in \pi(x, k) \tag{4.3}\\ \iff & c \in \pi(x, v') \wedge c' \in \pi(x, v) \tag{4.4}\\ \iff & d(x, c) + d(c, v') = d(x, v') \wedge d(x, c') + d(c', v) = d(x, v) \tag{4.5} \end{align}$$

From (1) to (2) is basically Definition 4.1 for the case $x = x'$. If additionally $v = v'$, this is also implicitly covered. Also keep in mind that shortest paths are unambiguous. (2) is true if and only if there exists a common prefix $\pi(x, k)$ of $\pi(x, v)$ and $\pi(x, v')$ where c and c' are part of this prefix (3). In turn, this is equivalent to that the c's are part of the shortest paths of each other's triple (4) which is only the case if the distance equations of (5) are fulfilled. The check for triples with coinciding targets proceed analogously. In practice, the distance lookups can be performed very quickly using a distance oracle like hub labels.

4.3.3. Experimental Results

We implemented our witness search heuristics in C++ and evaluated it on several graphs. Besides a very small *test graph* for sanity checking, we extracted real-world networks graphs based on Open Street Map (OSM) data for our main experiment. We picked *lower Manhattan* to get a grid-like graph. For the street network of the German federal city-state *Bremen*, we created a large version *car* which includes all streets passable by car, as well as the versions *fast car* and *highway* only containing streets allowing at least a speed of 50 km/h and 130 km/h respectively. The code was run on a 24-core Xeon(R) CPU E5-2650v4, 768 GB RAM machine.

To assess the quality of our lower bounds, we constructed a CH where the contraction order was defined by the *edge-difference* heuristic. From this CH, we calculated the average $|DSS|$ (forward + backward) and compared it with the lower bounds computed by our two methods.

In Table 4.1 we list our results. We updated the experiment results accordingly. Our biggest graph has over a quarter of a million edges and almost $120k$ nodes. As expected, the space consumption is quadratic which makes our current algorithm infeasible for continental-sized networks. For the large Bremen graph, our 24-core machine took 32.5 hours to complete the lower bound construction via our greedy heuristic and used 354GB of RAM. Most important is the quotient $\frac{LB}{|DSS|}$, which gives us a

Table 4.1.: Experimental results.

	test graph	lower Manhattan	Bremen (fast car)	Bremen (car)	Bremen (highway)
# nodes	22	2,828	40,426	119,989	1,781
# edges (org)	52	4,020	64,663	227,567	1,766
# edges (CH)	77	7,752	126,055	400,038	3,340
LB-construction space (A)	5.5 MB	233 MB	40 GB	354 GB	84 MB
LB-construction space (B)	4.9 MB	8.8 MB	91 MB	328 MB	6 MB
LB-construction time (A)	$< 1s$	$36s$	$100m$	$32.5h$	$< 1s$
LB-construction time (B)	$< 1s$	$10s$	$80m$	$22.9h$	$< 1s$
LB (A)	7.18	13.43	20.11	23.75	7.31
LB (B)	2.40	6.65	10.35	13.61	2.19
$\|DSS\|(avg.)$	10.27	29.85	61.99	78.34	7.58
$\frac{LB}{\|DSS\|}$ (A)	0.699	0.449	0.324	0.303	0.964
$\frac{LB}{\|DSS\|}$ (B)	0.233	0.222	0.166	0.173	0.288

Table 4.2.: Experimental results on lightheaded grid.

side length	1	5	10	15	20	25
LB	2.00	8.48	12.28	14.84	16.27	17.75
$\|DSS\|(avg.)$	2.00	12.48	32.94	53.21	77.2	102.88
$\frac{LB}{\|DSS\|}$	1.000	0.679	0.372	0.278	0.210	0.172

hint about the quality of our computed lower bound: For the *highway* version of Bremen, we achieve even 96%, that is, the average search spaces in the computed CH are guaranteed to be less than a factor of 1.05 above the optimum. Note however, that this graph is atypical for road networks in the sense that it is far from being connected. The value of $\frac{LB}{|DSS|}$ decreases for our bigger graphs down to around 30%. The decrease can have several reasons: On one hand, our CH is based on a heuristic and is not necessarily optimal (no optimal algorithm is known), so even if we had brute-forced a maximum sized triple-set we would most certainly not achieve 100%. Indeed, our contribution is to show that the gap between the heuristic and the perfect CH is at most $1 - \frac{LB}{|DSS|}$. On the other hand, the results strongly indicate that the missing of some long triples which do not have a length of $2^i - 1$ becomes more relevant in bigger graphs. Note that it could also mean that the edge-difference heuristic performs worse on bigger graphs. It has been shown that constructing the optimal CH (even though according to a different optimality criterion) is NP-hard, see [Mil12], so we actually conjecture that finding the largest conflict-free set of triples would also turn out to be NP-hard if investigated further. Our alternative method B for generating witness triples consistently fares worse in terms of the constructed lower bound, but requires very little space. It remains to see whether a more elaborate order of considering witness candidates leads to better results, or if a different underlying contraction hierarchy (which determines in which order to enumerate the center points) makes a big difference.

In another experiment, we generated lightheaded grid graphs and tested them with our method A. The results can be seen in Table 4.2. According to expectations, the average search space is growing linearly with the side length of the grid. Fulfilling the intentions of its construction, the average search space of a lightheaded grid is quite high in comparison to real world graphs: for a side length of 20, i.e. only 400 nodes in total, the search space size is close to the one of the much bigger real world graph *Bremen (car)* with almost $120k$ nodes. Also note that our approach for a trivial graph with only 1 node yields the tight bound. However, we also clearly see that the quality of the lower

Table 4.3.: Experimental results on grid with almost uniform weights.

side length	1	5	10	15	20	25
LB	2.00	8.80	14.68	19.0978	23.82	26.83
$\lvert DSS\rvert(avg.)$	2.00	13.00	27.13	41.6178	56.85	69.59
$\frac{LB}{\lvert DSS\rvert}$	1.000	0.676	0.541	0.458	0.419	0.385

bound deteriorates with bigger grids. The reason is probably that the average search space size grows linearly with the side length of the grid but Method A is quite inflexible concerning triple length and therefore seems to collect only roughly $\log(\sqrt{n})$ triples.

As real street networks are not structured exactly like lightheaded grids but exhibit grid-like structures as seen in the introduction, we conducted a final experiment on grids with almost uniform weights, see Table 4.3. On one hand, slightly perturbed weights seem more realistic and on the other hand, we need unique shortest paths for the computation anyway. Interestingly, we observe that both the size of the computed lower bounds as well as the average search space develop in less extreme manners than for the experiments on the lightheaded grid. Consequently, the term $\frac{LB}{|DSS|}$ develops in similar way as for the real-world graphs. However, even though these perturbed grid graphs are not as awkward for the CH-heuristic to produce small average search spaces as their lightheaded counterparts, they still exhibit relatively big search spaces for a small number of nodes compared to our measured real-world networks.

Moreover, we coincidentally observed that sea-routing implementations based on a globe discretization as mentioned in Section 4.2.1 exhibited significantly underwhelming performance of the CH-technique when compared to its speedup on street networks of similar size. This confirms the relevance of Theorem 4.2.3.

4.4. Tightness of Instance-based Lower Bounding Technique

In this last section we want to investigate whether there are graph classes where our instance-based lower bounding techniques can yield lower bounds matching the upper bound [1]. To that end we relax the conflict-free condition for the set W and show that this still yields a valid lower bound. This allows us to prove *tight* lower bounds for a some graph classes, the most interesting being balanced ternary trees. We also present other graph classes where our approach cannot deliver tight bounds.

4.4.1. Center-conflict

Definition 4.2 (center-conflict)
For two conflicting triples (x, c, v) and (x', c', v') we say that they are

- forward center-conflicting *if $x = x'$ and $\pi(c, v) \cap \pi(c', v') = \{c\} = \{c'\}$*
- backward center-conflicting *if $v = v'$ and $\pi(x, c) \cap \pi(x', c') = \{c\} = \{c'\}$*

Intuitively, we say that triples are *center-conflicting* if the only reason for their conflict is the common center node.

[1]At the time of publication, CH search spaces were the primary focus compared to HL and therefore the upper bounds were given in the context of CH instead of HL. With the insights of the next chapter, the CH upper bounds on the node search spaces are still valid but are less meaningful in hindsight while the upper bound stays meaningful for HL. Nevertheless, contraction orders can be given in a more concise form than hub labellings, so we stick to the presentation with CH search spaces for the purpose of brevity.

Definition 4.3 (only-center-conflict)
We call a set $|W|$ only-center-conflicting *if each triple has up to one forward and up to one backward center-conflict and is otherwise conflict-free.*

Lemma 4.4.1
A only-center-conflicting *set $|W|$ still lower bounds the sum of search spaces of all nodes in the network.*

Proof. The argument builds up on Lemma 4.3.1. The difference is that if a node c is the highest node in both of two center-conflicting triples (x, c, v) and (x, c, v'), c seems to be doubly accounted in the forward search space of x. However, we can let account w.l.o.g. the triple (x, c, v) for the node c in the forward search space of x and at the same time we do not have to let (x, c, v') go away empty handed: We let (x, c, v') account for the node c in the backward search space of v'. Now it could happen that in the backward search space of v' there is a center-conflict of (x, c, v') with another triple (x', c, v'). Then we let (x', c, v') account for the node c in the forward search space of x and so on. Either this process terminates because there are no center-conflicts anymore or we come back to (x, c, v) from where we started, but then we are done because (x, c, v) already contributes to our search space. Therefore, each center-conflict can be resolved such that no double counting occurs.□

As a side note, we experimented with other triple definitions where e.g. $c \in E$ which can deliver as big sets W, but we found the presented only-center-conflicting definition the easiest for the following analysis.

4.4.2. Special Triples Representing Nodes in their own Search Space

In the literature of analyzing search spaces, it is common that a node contributes to its own search space. Taking account of this with our triple definition forces us to do a minor technical adjustment: we allow triples of the form (x, x, x) twice in our set, essentially making it a multiset. One can imagine there is one of these triples representing the forward search space and one for the backward search space. Other possibilities to handle that technical detail would be to not count nodes for their own search spaces or to assign a direction to triples, so that e.g. $(x, x, x)_\rightarrow$ and $(x, x, x)_\leftarrow$ would be two distinguishable elements.

4.4.3. Balanced Ternary Trees

Lemma 4.4.2
For a balanced ternary tree, a CH and an only-center-conflicting *multiset W can be found where $|W|$ and the sum of the search spaces SS_Σ of the CH matches exactly.*

Proof. For an illustration of a balanced ternary tree see Figure 4.8, each undirected edge represents two directed edges, each for one direction. We construct the CH in a natural way: Nodes closer to the root get higher levels and the root has the highest level. For the example in Figure 4.8, we choose the level equal to the node index. Then, the search space (forward or backward) of a node in the layer i, where $i = 1$ is the layer of the root node, has size i. Let d be the depth of the tree, i.e. the number of layers, then the sum of the search spaces (forward + backward) can be calculated as follows:

$$SS_\Sigma = 2\sum_{1}^{d} i \cdot 3^{(i-1)}$$

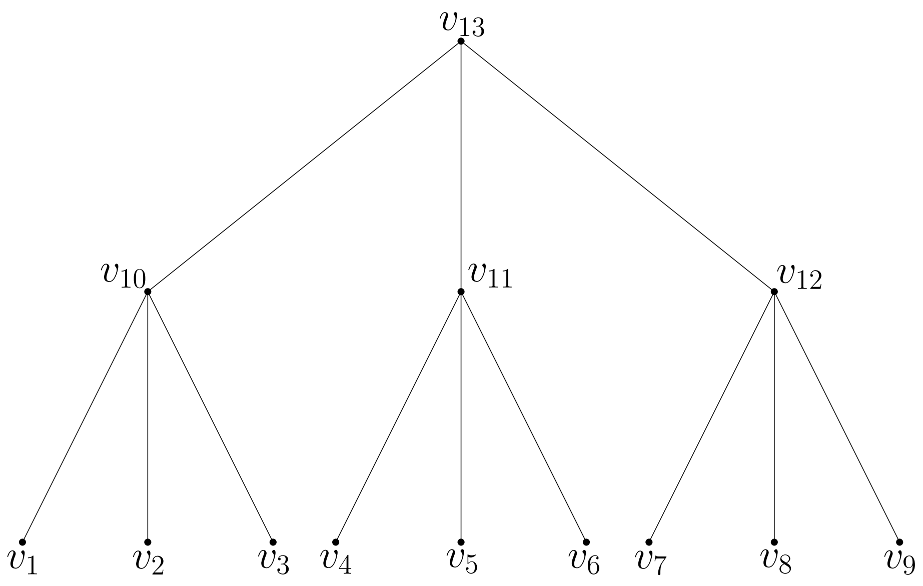


Figure 4.8.: Balanced ternary tree.

The factor 2 originates from putting forward and backward search space together, i nodes are in a search space of a node in depth i and we have $3^{(i-1)}$ nodes at depth i.

For the set $|W|$, we choose the triples as illustrated in Figure 4.9. In each shortest path tree, consider the path from the shortest path tree root s to the ternary tree root r. For each node c on such a path we can always find two triples of the form (s, c, x), where x has the same distance to c as s. Such triples can always be added without violating the conflict conditions. The two circles in the figure represent the special triples mentioned in Section 4.4.2. So for a node of layer i, $2i$ triples appear in both of its forward and backward shortest path trees. As each triple appears in a forward and a backward tree, we have to divide the number of appearances by 2 at the end:

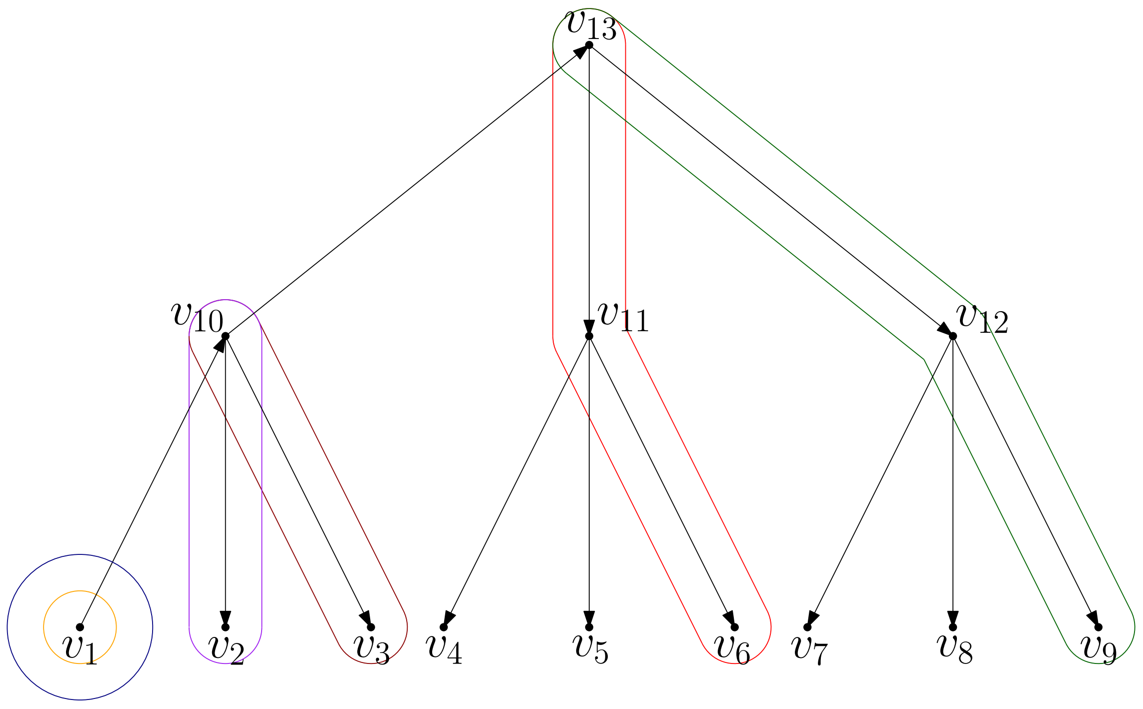
$$|W| = \frac{1}{2} \cdot 2 \cdot \sum_{1}^{d} 2i \cdot 3^{(i-1)}$$

Therefore, $SS_\Sigma = |W|$. □

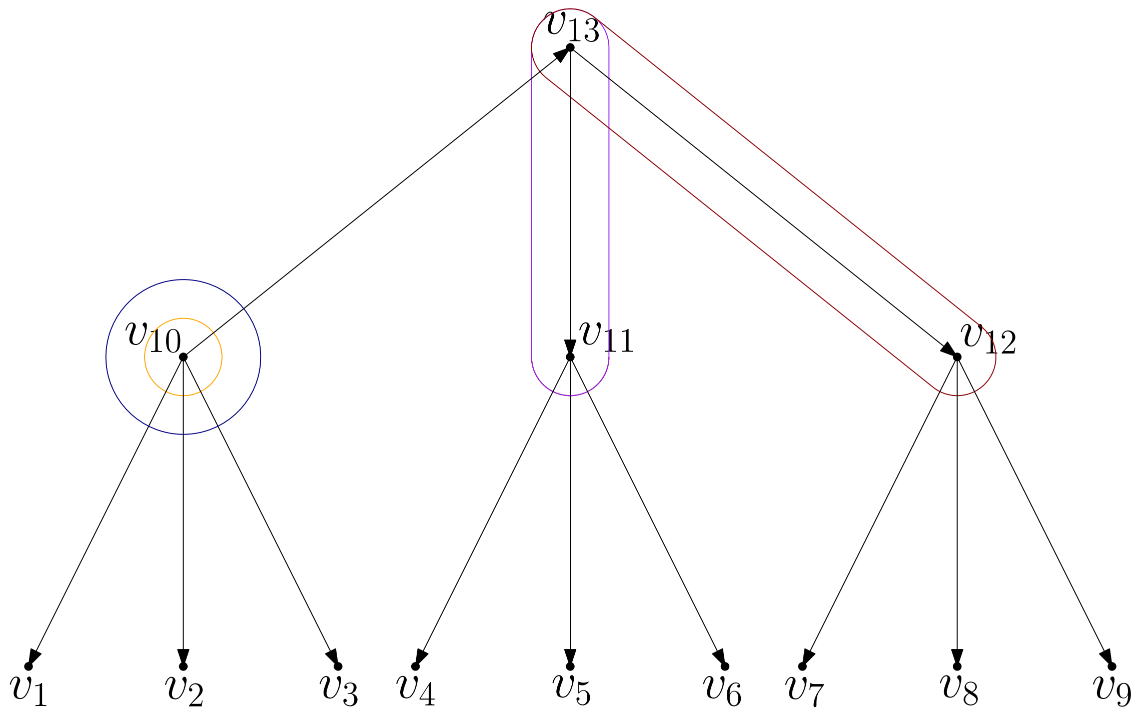
4.4.4. Balanced Binary Trees

Balanced binary trees seem to be the simplest graphs where our approach is unable to deliver tight lower bounds. For the balanced binary tree of depth 3 in Figure 4.10a, the straightforward (and almost certainly the optimal) contraction strategy from Section 4.4.3 yields

$$SS_\Sigma = 2 \sum_{1}^{3} i \cdot 2^{(i-1)} = 34$$

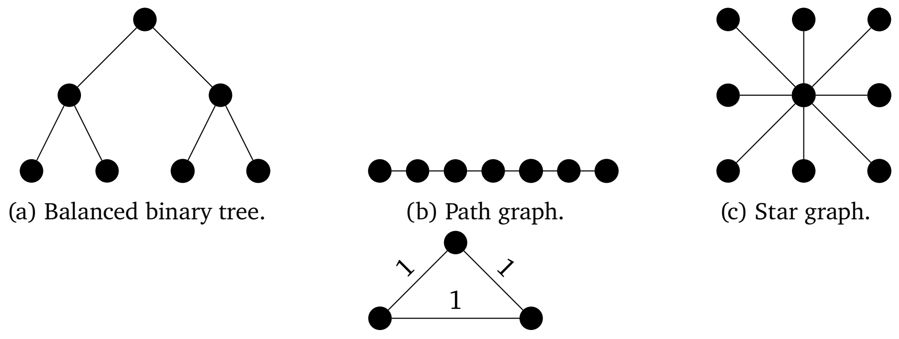


(a) $SPT_{\rightarrow}(v_1)$.



(b) $SPT_{\rightarrow}(v_{10})$.

Figure 4.9.: Shortest path trees.



(a) Balanced binary tree. (b) Path graph. (c) Star graph. (d) Triangle graph.

Figure 4.10.: Example for Graphs

But in contrast to finding $2i$ non-conflicting triples for each node in the ternary tree, here we rather get only $2+(i-1)$, because we only get the triples (s,s,s) twice. This yields

$$|W| = \frac{1}{2}\cdot 2\cdot \sum_{1}^{3}(2+(i-1))\cdot 2^{(i-1)} = 24$$

and therefore $\frac{|W|}{SS_\Sigma} = \frac{24}{34} < 1$. We can also easily see that this schema leads to even worse coefficients for deeper balanced binary trees, i.e. $\lim_{d\to\infty}\frac{|W|}{SS_\Sigma} = 0.5$.

4.4.5. Path Graph

For path graphs P_n as depicted in Figure 4.10b, only $n \le 2$ allows a tight lower bound construction. For $n = 2^k - 1, k \in \mathbb{N}$ the straightforward contraction hierarchy can be constructed if the middle node gets the highest level and then the left and right parts get the level assigned in a recursive manner. Then the analysis becomes very similar to that of the balanced binary trees.

4.4.6. Star Graph

Star graphs like in Figure 4.10c allow tight lower bounds. The central node has to be the highest level and the triples can be chosen in a similar manner as for the balanced ternary tree so that they are center-conflicting only in the central node. Compared to the balanced ternary trees, there are many more possibilities for the choice of the triple set.

4.4.7. Balanced N-ary Trees

For $n \ge 4$ tight bounds can be computed, basically exactly like in the ternary tree case. Similar to the star graph case, there are more possibilities for the choice of triples.

4.4.8. Non-tree Graphs

For non-tree graphs, the analysis becomes much more tedious as the SPT are not isomorphic to the original graph. Edge weights also become important as they define the SPTs, we focus on a simple triangle graph with uniform edge weights as in Figure 4.10d here. This triangle is symmetric in such a way that the contraction order can essentially be seen as unambiguous. It is not hard to see that $SS_\Sigma = 12$. Then 4 triples can be set in each SPT in a straightforward and symmetric manner which yields $|W| = 3 \cdot 4 = 12$, therefore the bound is tight here.

Of course, we expect more complicated non-trees to become non-tight quite easily since they are far more complex than tree graphs and even balanced binary tree graphs exhibit non-tightness. As a side note, in these examples the direct search space was equal to the actual search space which is in general not true for CHs on more complex non-tree graphs.

4.5. Materials and Methods

The code of method A for the experiments we described in Section 4.3.3 can be found under `https://theogit.fmi.uni-stuttgart.de/ruppts/compchlowerbounds`. Some of the smaller graphs are also included there for reproducibility.

4.6. Conclusions

In this chapter we have proven a strong theoretical lower bound on the number of nodes that have to be considered during a CH search or HL lookup. Our lower bound instance is not too far from road network structures that occur in the real world. Our theoretical results imply that existing CH or HL construction schemas are essentially optimal (up to a logarithmic factor) for such grid-like networks.

In general, most theoretical considerations (in particular concerned with upper bounds) are somewhat unsatisfying in that they talk about the search space as the number of nodes that are visited in the search. In practice, the number of edges to relax is considerably larger than the number of nodes, though, and this can also happen in theory which we will see in the next chapter. Often, one simply upper bounds the number of edges by the square of the number of nodes to be considered (e.g., [AFGW10]), which is not really meaningful, if the latter is already $\Omega(\sqrt{n})$. Also, bounds not only on the *direct search space* but on the 'real' *search space* during a CH query, i.e., including nodes that are considered with non-shortest-path distance, might be interesting to investigate.

More on the practical side, we instrumented the insights from our lower bound proof to come up with a construction schema for *instance-based lower bounds* for CH and HL, if a concrete road network instance is given. For moderately sized networks, we could show that current CH and HL construction schemas indeed yield average search spaces not too far away from the optimum (less than a factor of 4). For a variant of our instance-based schema we have also shown that it can yield tight lower bounds for some graph classes, most notably ternary tree graphs. In future work, we aim at making the respective algorithms constructing the instance-based lower bounds more scalable to allow the lower bound construction even for country- or continent-sized networks. Furthermore, it might be worth investigating for which other graph classes our lower bounding technique has the potential to compute tight lower bounds. Perhaps it is even possible to slightly tweak our triple definition a step further to be potentially tight for every graph.

Chapter 5 On the Difference between Search Space Size and Query Complexity in Contraction Hierarchies

This chapter is based on joint work [PR21] with Claudius Proissl. It was presented at the SIAM Conference on Applied and Computational Discrete Algorithms (ACDA21). I omitted some part of this work where I was not the primary contributor.

5.1. Related Work

Bauer et al. [BCRW13] show upper bounds on the query complexity of Contraction Hierarchies (CH) that solely depend on the structure of the graph G (not on the edge weights). Their considerations focus on the notion of the node *search space*. Bauer et al. [BCRW13] then show that the size of the search space of nodes in a graph G with treewidth k is in $O(k \log n)$.

Funke and Storandt [FS15] follow the idea of investigating node search space sizes in CH instead of the query complexity itself. They construct the overlay graph in a random fashion and propose that *bounded growth* is a possible characteristic to explain the success of CH. Based on these results, Blum, Funke, and Storandt [BFS18] show that if a graph G exhibits bounded growth then the random construction produces expected search space sizes in $O(\sqrt{n} \log n)$ and expected *direct* search space sizes in $O(\sqrt{n})$.

The size of the search space, as typically defined in [BDG+16] or Section 2.4.1, is not an upper bound on the query complexity, though, as it only depends on the number of nodes. The number of edges that need to be considered could be much larger and in the worst case quadratic in the number of nodes. However, the common assumption was that for network-like graphs this worst case upper bound would turn out to be far too pessimistic. In fact, there has been the tendency to neglect the difference between search space sizes and the query complexity of CH. In this paper we show that, unfortunately, the result of Bauer et al. [BCRW13] is a good example that the difference can be significant. We show this by constructing lower bounds of $\Omega(n)$ for the expected number of processed edges. As a side note, we showed a very similar result for Blum, Funke, and Storandt [BFS18] in the publication [PR21] this chapter is based on.

5.2. Contribution

In this chapter we give an example that the difference between the search space sizes with respect to nodes and edges can be significant. Firstly, we address a separator-based CH-construction schema described by [BCRW13] which yields node search spaces of $O(\sqrt{n})$ in the case of grids. We then show that this contraction schema can lead to edge search spaces of size $\Omega(n)$ for a specific family of grid graphs. Hence, we show that the gap between the average search space size with respect to nodes and the average query complexity can be surprisingly large, even for grid graphs. We achieve this result by counting edges that need to be processed during queries. This is in contrast to most of the previous work, where edges are neglected and only nodes are considered. Thus, with our example, we are able to emphasize how important it is to take edges into account when analyzing the query complexity in CH.

Moreover, we also show that for the same graphs significantly better CH-constructions exist which yield edge search spaces of size $O(\sqrt{n} \log n)$.

5.3. Edge Search Spaces

Search spaces, as typically defined, only consider nodes. We now define analogous search spaces that consider edges and explain their connection to query times of the actual algorithm.

Definition 5.1 (edge search space)
Given a graph $G = (V, E)$ with a node order. The Edge Search Space $SS_e(u)$ of a node $u \in V$ is defined as follows.

$$SS_e(u) := \{(v_1, v_2) \in E : \ v_1 \in SS(u), \ v_1 < v_2\}$$

Hence, $SS_e(u)$ contains all upward edges in the search space of node u. The size of $SS_e(u)$ is an upper bound of edges that need to be processed during queries from node u.

We introduce the Minimal Edge Search Space to obtain a lower bound for the query complexity.

Definition 5.2 (minimal edge search space)
Given a graph $G = (V, E)$ with a node order. The Minimal Edge Search Space $MSS_e(u)$ of a node $u \in V$ is defined as follows.

$$MSS_e(u) := \{(v_1, v_2) \in E : \ v_1 \in DSS(u), \ v_1 < v_2\}$$

Hence, $MSS_e(u)$ contains all upward edges whose source is in the Direct Search Space (DSS) of node u. We intentionally do not call this set *Direct* Edge Search Space because this name would imply that $MSS_e(u)$ only contained edges that are part of shortest paths from node u. This is in general not the case. Nonetheless, all elements in $MSS_e(u)$ must be considered in queries from node u. This remains true even if the common pruning schema *stall-on-demand* [GSSD08] is performed.

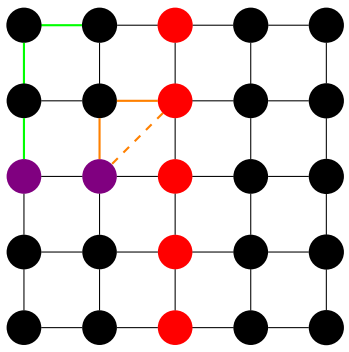


Figure 5.1.: Red and purple separators with example shortest paths.

5.4. Analysis of Separator-based Contraction Schemes

In this section we look at differences between the node and edge search space sizes when CHs are constructed based on separators.

5.4.1. Divide and Conquer Strategy with Separators

For planar graphs, there is a straightforward divide-and-conquer CH-construction strategy to upper bound the node search spaces: Compute a balanced separator, then assign the highest ranks to nodes in the separator. Recursively separating the remaining graph in parts A and B has the effect that for nodes $a \in A$ and $b \in B$ no shortcut edge (a, b) needs to be constructed. Having few shortcuts makes the analysis to show that the node search spaces have size $O(\sqrt{n})$ for a grid graph quite simple. For a more generic result and a formal proof, see Bauer et al. [BCRW13] (Theorem 5). While their result clearly upper bounds the number of *nodes* to be considered during a query, this might be considerably less than the number of edges to look at. We apply a balanced separator strategy upon lightheaded grids which have already proved useful for showing lower bounds on CH constructions in the previous chapter. For this construction, we show that the average minimal edge search space $MSS_{e,\text{avg}}$ is in $\Omega(n)$. We also show that it is not the CH framework to blame by providing an alternative construction schema for these specific graphs which yields sublinear $SS_{e,\text{avg}}$ of size $\mathrm{O}(\sqrt{n}\log n)$.

5.4.2. Separator Strategy

Now we show that for lightheaded grids, the seemingly reasonably separator strategy can yield linear edge search spaces.

Theorem 5.4.1
Applying the divide-and-conquer separator strategy as described in Section 5.4.1 on a lightheaded grid can yield $MSS_{e,avg}$ of size $\Omega(n)$.

Proof. We assign the highest ranks according to the separators as indicated in Figure 5.1: the red nodes form the separator of the whole grid and are assigned to the highest ranks. The purple nodes form the separator of the left part and therefore get smaller ranks than the red nodes but they are ranked higher than the black nodes. The rank order within the red, purple and black nodes can be arbitrary.

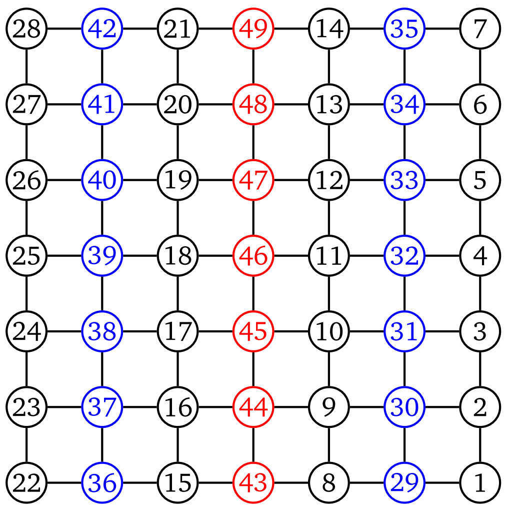


Figure 5.2.: Vertical Red and blue separators.

Let B be the set of all black nodes in the upper left quadrant. We show that for the individual edge search spaces it holds that: $\forall b \in B : |MSS_e(b)| \in \Omega(n)$. As $|B| \in \Theta(n)$ this will suffice for our claim.

Let now P be the set of all purple nodes and R all the red nodes in the upper half without the center node. Note that $|P| = |R| \in \Omega(\sqrt{n})$.

Due to shortest paths between nodes $p \in P$ and $r \in R$ as illustrated in orange, there have to be shortcuts (p, r) as indicated by the tilted line. So, for any $p \in P, \forall r \in R : (p, r) \in MSS_e(p)$.

Also, all black nodes reach all purple nodes as indicated with the green path, so $\forall b \in B, \forall p \in P :$ $p \in DSS(b)$. Note that shortcut edges (b, p) do not have to exist, though.

We now look at subsets of the minimal edge search spaces for our lower bound: $\forall b \in B : MSS_e(b) \supseteq$ $\dot{\bigcup}_{p \in DSS(b) \cap P} \{(p, r) | r \in R\}$. Putting our last observations together, we notice that we have $|P| \cdot |R| \in \theta(n)$ shortcuts which are contained in each minimal edge search space of the nodes $b \in B$, so we have $\forall b \in B : |MSS_e(b)| \in \Omega(n)$. □

5.4.3. Improved Construction

Having just seen linear average edge search spaces for a seemingly reasonably constructed CH may invoke doubts about the theoretical efficiency of the CH framework itself. However, we show that sublinear edge search spaces are still possible here.

Theorem 5.4.2
Lightheaded grids admit CH constructions for which $|SS_{e,avg}| \in O(\sqrt{n} \log n)$.

Proof. For the construction we still use separators. The schema is best understood by looking at the example in Figure 5.2, where each individual node is labeled with its rank. For simplicity, we refer to a node by its rank label in the following.

To make the analysis easy, the ranks were assigned in such a way that no vertical or diagonal shortcuts have to be added during the contraction process. Only horizontal shortcuts may appear, e.g. when node 8 is contracted, there will be the shortcut (29, 43).

The black nodes have the lowest ranks and the nodes at the bottom have lower ranks compared to other nodes in the same column. Therefore, the lowest black nodes have the largest edge search spaces and w.l.o.g. we can focus on node 1. For $SS_e(1)$, all the vertical edges from 1 to 7 are included, which are $O(\sqrt{n})$ in the general case. Then all the horizontal edges to the next vertical blue separator 29-35 are included, which are $(1, 29), (2, 30), \cdots, (7, 35)$, also $O(\sqrt{n})$ many. The edges within the blue separator 29-35 are also in the search space, but those are only $O(\sqrt{n})$ again.

From the separator 29-35, only 7 (horizontal) upward shortcuts go directly to the next highest separator and are therefore part of $SS_e(1)$ while the columns of nodes in-between are irrelevant. For the general case of a bigger grid we continue this schema of analyzing the search space by proceeding with the next highest separator successively.

As we encounter $O(\log n)$ separators in general and each is contributing $O(\sqrt{n})$ edges, we get $|SS_{e,\text{avg}}| \in O(\sqrt{n} \log n)$. □

5.5. Conclusion

Bauer et al. [BCRW13] showed that for certain graph families, (expected) node search spaces were rather small, i.e. sublinear. It is clear that upper bounds on search space sizes are not necessarily upper bounds on the query complexity as well (even though the name *search space* suggests it). However, the common assumption was that for transportation network-like graphs the difference between these two quantities would be minor.

On the basis of lightheaded grids from Rupp and Funke [RF20] we showed that the separator-based construction schema of Bauer et al. [BCRW13] can indeed lead to linear edge search spaces. We also showed that the CH framework was not the problem as these specific graphs admit significantly better CH-constructions with respect to the edge search space.

Note that linear edge search spaces mean that the number of processed edges in CH queries are asymptotically equal to the number of edges processed by Dijkstra's algorithm. A speed-up compared to Dijkstra's algorithm is therefore uncertain in these cases.

Our results motivate two things. First, the query complexity of CH mainly depends on the number of processed edges, not on the nodes. Second, our example shows that the difference between the query complexity and the node search space sizes can be surprisingly large.

In summary, there is still no really satisfying theoretical explanation for the tremendous efficiency of speedup techniques like CH in practice. We hope that this chapter stimulates further research in that direction.

Part III.

Contraction Hierarchy as Search Structure

Chapter 6
Motivation

With the ubiquity of mobile devices capable of tracking positions (be it via GPS or via Wi-Fi/mobile network localization) there is a continuous stream of *location data* being generated every second. Not all of this data is stored permanently, but platforms like Strava, GPSies, or OpenStreetMap allow users to collect and share their location data with the community. Mobile network providers or companies like Google or Apple also have access to the location data of their customers and use it to improve their services, e.g., to measure traffic flows or detect special events.

In all of these cases, location measurements are typically not considered individually but rather as sequences, each of which reflects the movement of one person or vehicle. In this work we assume that such sequences of location measurements have already been mapped to paths in an underlying transportation network using appropriate methods. For our contributions, we use the term *trajectories* to refer to such already map-matched movement sequences.

In Chapter 7, we will look at algorithmic challenges of storing and retrieving such trajectory data.

However, just being able to retrieve trajectory data in a raw form is usually not sufficient for most use cases like traffic planning where a human must be able to interpret the data. Typically, the trajectory data is then visualized on top of a map. This task is very related to displaying street networks for natural user interaction which we also tackle in the second chapter of this part.

Displaying graphs poses quite a few algorithmic challenges since simply rendering all elements of the network yields a completely cluttered screen when considering, for example, a complete view of a country like Germany (see Figure 6.1, left). Hence, depending on the desired zoom level and rendering capability of the device that displays the network, we seek to compute a simplified graph representation that still allows the user to navigate the map while not overloading it at the same time (see Figure 6.1, right).

For an offline setting, where all the network data is known beforehand, pre-rendered tiles at a fixed number of zoom levels might be a viable solution. However, those are still non-trivial to design, they need a significant amount of memory to be stored or transmitted to a mobile client, and with this approach it is not possible to adapt the level-of-detail to user or device demands. Furthermore, if network details change or if additional graph data shall be displayed online as an overlay, this is difficult to integrate with fixed tiles, and a graph representation of the different zoom levels is needed on top.

Therefore, in [FSS17], a graph data structure was proposed that allows for continuous level-of-

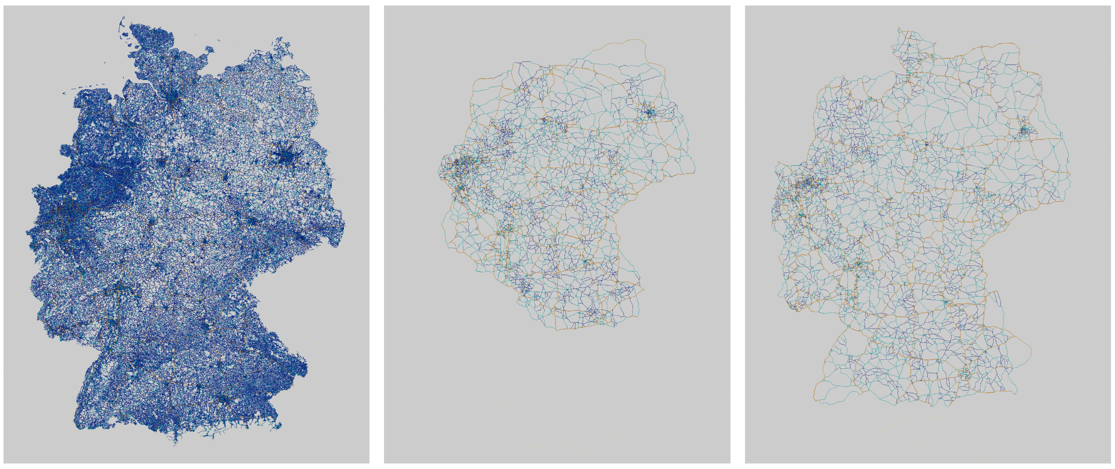


Figure 6.1.: Road Network Simplification. Left: Germany, all roads (about 50 million segments). Center: Germany simplified (about 130k segments) URAN [FSS17] Rendering of Germany with standard CH ordering; around 130k road segments. Right: URAN rendering with about 130k segments with silhouette preservation order

detail variation and hence for rendering of the respective graph simplifications on demand. The data structure was inspired by the CH technique. An augmentation of the CH data structure allows to use it for both, road network rendering and efficient path planning. The resulting URAN framework (Unified data structure for Rendering and Navigation) was shown to perform well even on large networks, typically allowing to extract the set of edges to be rendered within a few milliseconds. However, we will see in this part, that the visual quality of the road network simplifications computed with URAN is lacking, especially for complex network parts such as freeway interchanges, where topological inconsistencies occur. Also, the original URAN makes no effort to retain the silhouette of the original graph which can be seen in Figure 6.1, middle. Those issues can be quite confusing for the user. In Chapter 8, we will propose efficient methods to overcome some of these shortcomings, leading to a drastically improved simplification quality with only a mild increase in preprocessing effort.

6.1. Related Work

6.1.1. Storage and Retrieval of Trajectory data

Problem Description An underlying road network is given as a directed weighted graph $G(V, E, c)$ with V embedded in $\mathbb{R}^2$. The trajectory data is provided as a collection $\mathcal{T}$ of paths, where each path $t \in \mathcal{T}$ is a sequence of nodes $\pi = v_0 v_1 \dots v_k$ in G annotated with time stamps $\tau_0, \tau_1, \dots, \tau_k$.

Our goal is to construct an index for $\mathcal{T}$ which allows to efficiently answer queries of the form

$$[x_l, x_u] \times [y_l, y_u] \times [\tau_l, \tau_u]$$

that aim to identify all trajectories which in the time interval $[\tau_l, \tau_u]$ traverse the rectangular region $[x_l, x_u] \times [y_l, y_u]$. In the literature this kind of query is often named *window*-query [YWY+18] or *range*-query [KJT16], where the formal definitions may differ in detail, also see [Zhe15]. In addition, we want to answer queries which specify periodic time events, e.g., a query which asks for all trajectories on weekends that intersect a query rectangle $[x_l, x_u] \times [y_l, y_u]$.

Two straightforward approaches for answering window-queries are the *linear scan* where every edge of every trajectory $t \in \mathcal{T}$ is explicitly checked for intersection with the query rectangle/space-time cube. There is also the idea of an *inverted index* where a network edge is associated with all trajectories using that edge. Then at query time, one determines the set of network edges intersecting the query rectangle and checks the time constraints for all respective associated trajectories. However, both approaches are too slow and space consuming to handle large networks and huge sets of trajectories in practice.

In the literature, storage and retrieval of trajectory data is an established field of research. There are basically two different flavors of the problem. The more geometric and continuous variant considers the trajectory data as mere sequences of points in $\mathbb{R}^2$ (or even $\mathbb{R}^3$) that are freely located in ambient space, see, e.g., [WZX+14]. In the other, more discrete variant, as, e.g., proposed in [GAD06], trajectories are paths in an underlying network, which can be exploited for storage as well as indexing. In our work we focus on the latter variant and assume that the 'raw' trajectories, which are typically given as a sequence of GPS coordinates, have been matched to respective paths in G using some *map matching* technique, e.g. as described in [BPSW05; LHK+13; QON07; Sey17; Zhe15].

Many other approaches also focus on already map-matched trajectories. Within this group of approaches there are still some different flavors of the problem, e.g. when it comes to the acceptance of data loss when compressing. In the following we give an overview of the most important contributions and highlight the different flavors.

There is a distinction between whether the index is designed to work in a disk or in a RAM context. The state-of-the-art disk based systems are PARINET [SZO+11] and NETTRA [KPTT14]. PARINET stands for "PARtitionned Index for in-NEtwork Trajectories". It is based on a combination of graph partitioning and a set of composite B^+-tree local indexes. It is designed to be combined with a database management system. NETTRA stands for "NETwork-constrained TRAjectory index". NETTRA is implemented in standard SQL and focuses on retrieving trajectories which hit all edges of a given query. Both PARINET and NETTRA can also be adapted to work in RAM but were then shown to be inferior to SPNET [KJT16] which was explicitly designed to work in RAM. As our approach is also designed to work in RAM, we consider SPNET as our strongest competitor and focus on comparing PATHFINDER to SPNET in our empirical studies. Nevertheless we also report on some experiments externalizing our PATHFINDER structure.

Another storage and retrieval framework is PRESS [SSZZ]. PRESS stands for "Paralleled Road-Network-Based Trajectory Compression". It has the major drawback that it has to store a table for all shortest paths in the scale of $|V|^2$. Therefore it doesn't scale well and was only evaluated on city-sized graphs.

Similar to PRESS is the framework TED [YWY+18] "where a trajectory is represented by a spatial entry path (E), distances (D) that locations appear in the spatial entry path, and a time flag sequence (T)". Similar to other frameworks, the authors of TED used grids to partition the road network. Moreover, they focus on relatively fine-grained bit-optimization. Even though TED should scale better than PRESS, they also restrict their experiments to city-sized networks. Moreover, SPNET seems to be superior to TED in their own experiments.

As SPNET [KJT16] generally outperforms the other approaches and is most similar to our own approach, it's worth describing a bit more in detail. Its name is based on "Shortest-Path compressed trajectories in NETworks". In regard of compression, SPNET keeps only the start and end times of trajectories as the authors argue that these are by far the most significant for practical purposes. In comparison, our approach also has some loss for time compression but strictly less than SPNET. For the purpose of querying, both approaches act as conservative filters, i.e. the correct result set is always a sub-set of what is returned. One of the basic ideas of SPNET is that for compression, it is sufficient to store only the first and last edge in the path of a trajectory that follows a (unique) shortest path. They implement this idea by extending the HL concept which in turn is related to the CH concept our approach utilizes. More concretely, they extend HL, such that HL can determine both the shortest path and whether there are multiple shortest paths between two vertices. The general drawback of HL is the space consumption which scales badly with the size of the graph. Whereas they restricted themselves to the street network of Denmark with 1.8 million edges, our approach handled more detailed continent-sized networks of almost a billion edges. Moreover, they used a non-public data set of trajectories while we use only openly available data to make our results fully reproducible. They also recognized and made use of the fact that in their framework, it is (usually) not necessary to fully decode a candidate trajectory to determine whether it spatially intersects a given set of edges. This will also be the case for our approach. They also partition the road network and try balancing it so that trajectories should not be in many partitions. They note that “A future line of research is to use a hierarchical network partitioning scheme to better support both small and large query regions, and to explore new techniques for filtering out shortest paths.” Even though we were not aware of SPNET when we created our approach, this hierarchical partitioning is inherently incorporated by our approach.

Our compression is lossless concerning spatial information while keeping a moderate amount of temporal information. Since we do not drop spatial information in the compression, we are able to answer purely spatial queries with a 100% accuracy. For queries including a time interval, our approach works conservatively like SPNET, always containing the exact result as a subset.

6.1.2. Visualization

6.1.2.1. Graph Simplification

Graph simplification is well-studied for several special cases, most prominently path or polyline simplification or the simplification of planar subdivisions.

Simplifying general graphs is significantly more challenging. In [HST+20], a road network simplification technique (called COMA) was introduced that prunes certain nodes and edges and replaces them with so called *bridge edges* if necessary. Their envisioned application is not road network rendering, though, but to preserve high accuracy for map-matching queries while significantly reducing the space consumption for the underlying road network. In [SJP+07], road network compression was achieved by clustering roads of similar shape, computing a representative for each cluster, and then replacing the original parts with this unified pattern. A short survey over further road network compression and simplification techniques was conducted in [KHN+14], and a longer one concerning different graph types in [LSDK18]. Although there is a multitude of different approaches out there, none of them except for URAN allows to extract simplified graph views at any desired level-of-detail or to enable fast shortest path planning. We hence focus in this part on improving URAN while maintaining its full feature set.

6.1.2.2. Trajectories

In general, trajectory visualization can be seen as special case of graph visualization and hence graph simplification techniques are also applicable here. Nevertheless, visualizing trajectories is a well researched topic on its own across several domains [AA13]. Typical use cases are traffic/movement analysis [LWY15; WLY+13] and urban planning (e.g. [AAC+17; AWK+16]). Trajectories from different objects other than cars, e.g. ships [ACM+21], may also favor different specialized approaches. Also, general visualization techniques as e.g. heat maps are used [ACM+21] which we also incorporate in one of our approaches.

Chapter 7
Pathfinder

7.1. Introduction

This chapter is based on joint work [FRNS19] with Stefan Funke, André Nusser and Sabine Storandt. It was published in the Proceedings of the 16th International Symposium on Spatial and Temporal Databases (SSTD 19). Theorem 7.2.2 and its proof are secondary contributions of mine.

The main contribution of this chapter is the development of a data structure allowing for efficient compression and storage of as well as access to a huge number of such trajectories. Here, by 'huge' we mean tens of millions of trajectories in a country- or continent-sized network, or – in the long run – even billions of trajectories. Taking the trajectories of vehicles into consideration, this enables a plethora of important use cases such as: (1) traffic anomaly detection by monitoring the travel time of vehicles traversing a certain region at a certain time interval, (2) determining most frequently used paths within a region to facilitate historical trajectory based route planning, or (3) quick access to relevant trajectories while mapping the world (the OpenStreetMap project currently hosts more than a million such trajectories but without an efficient retrieval system).

7.1.1. Contribution Summary and Outline

In this chapter we present a novel index structure that allows to answer window-queries on network-constrained trajectory sets on an unprecedented scale. While current indexing schemes work on small network sizes (e.g., PRESS [SSZZ]: network of Singapore, PARINET [SZO+11]: cities of Stockton and Oldenburg, TED [YWY+18]: cities of Singapore and Beijing) or moderate sizes (e.g., SPNET [KJT16]: network of Denmark with 800k vertices), our approach efficiently deals with country- (Germany with 57 million vertices) or even continent-sized networks (Europe with 437 million vertices). For example, for the network of Europe and 10 million trajectories, we can answer a window-query within few *microseconds* per reported trajectory in the output. The scalability of our approach is mostly due to the fact that our index structure is a very lean augmentation of a constructed *contraction hierarchy* [GSSV12b], which might be available anyway if routing queries are to be answered for the network. It inherits the hierarchical structure from the contraction hierarchies and is hence equally suitable for small and large query windows.

In the following, we develop our spatial indexing scheme in Section 7.2. The extension to cater for temporal information is described in Section 7.3, followed by an extensive experimental evaluation in Section 7.4.

7.2. Spatial Pathfinder

This section contains the main contribution of this chapter: our compression scheme as well as the spatial part of the PATHFINDER data structure and query algorithm.

7.2.1. Compression

As a very first step, we compute a CH for $G(V, E, c)$. Then, for each trajectory $t \in \mathcal{T}$, we construct its CH-representation, that is, we represent its path $\pi = v_0 v_1 v_2 \dots v_k = e_0 e_1 \dots e_{k-1}$ with $e_i \in E$ as a path $\pi' = e'_0 e'_1 e'_2 \dots e'_{k'}$ where $e'_i \in E^+$. The CH-representation has to fulfill that for $\pi' = e'_0 e'_1 e'_2 \dots e'_{k'}$, the recursive substitution of every $e'_i \in E^+ \setminus E$, i.e. shortcut, with its two bridged edges results in the original path π with respect to the edges. By that requirement, the complete spatial information of t can be reconstructed from π'. Moreover, there must not exist a shortcut bridging two neighboring edges e'_i and e'_{i+1} for $i \in \{1 \dots k' - 1\}$. It turns out that the CH-representation is unique, even when shortest paths are not unique. Let us first prove the following lemma, which we then use to prove uniqueness of CH-representation.

7.2.1.1. Uniqueness

Lemma 7.2.1
For every shortcut $e = (v_s, v_t)$ which represents a shortest path $v_0 v_1 \dots v_l$ where $v_0 = v_s$ and $v_t = v_l$, all bridged vertices have a lower level than the shortcuts endpoints i.e. $\forall i \in \{1, \dots, l-1\} : l(v_i) < l(v_s) \wedge l(v_i) < l(v_t)$.

Proof. We prove this lemma by induction. Firstly, a shortcut $e = (v_s, w_t)$ always results of the contraction of a node v_{contr} which fulfills $l(v_{\text{contr}}) < l(v_s) \wedge l(v_{\text{contr}}) < l(v_t)$ by construction of the CH. As the represented path of e is the concatenation of the paths represented by the shorter bridged edges (v_s, v_{contr}) and (v_{contr}, v_t), we look at them more closely. If a bridged edge is no shortcut, there are no nodes which could violate the level constraint. If a bridged edge is a shortcut we know by induction that all of its bridged nodes have lower level than v_{contr} and v_s (respectively v_t). □

We can now prove the uniqueness of the CH-representation.

Theorem 7.2.2
The CH-representation is unique.

Proof. We prove this by contradiction. Let us assume that there are two different CH-representations of $\pi = e_0 e_1 e_2 \dots e_k$: $\pi' = e'_0 e'_1 e'_2 \dots e'_{k'}$ and $\pi'' = e''_0 e''_1 e''_2 \dots e''_{k''}$.

An important observation is that edges e' and e'' never overlap if one does not completely contain the other. More formally, if there are edges e' and e'' which represent the paths $v_a \dots v_c$ and $v_b \dots v_d$ respectively, then $a > b > c > d$ does not hold. That is because applying Lemma 7.2.1 twice would yield $l(v_c) > l(v_b)$ and $l(v_b) > l(v_c)$, a contradiction.

With that observation and since π' and π'' are different, there must be w.l.o.g. an edge e' which represents the same path as edges $e''_1 e''_2 \dots e''_o$ with $o \geq 1$.

Algorithm 7.1 Converting π into its CH-representation. Note that it directly works on π so the indices change while replacing edges.

```
1: procedure toCHPath($\pi = e_0 \dots e_{k-1}$)
2:     $i \leftarrow 0$
3:     while $i + 1 < \text{length}(\pi)$ do
4:         if $e_i$ and $e_{i+1}$ form a shortcut then
5:             $s \leftarrow \text{getShortcut}(e_i, e_{i+1})$
6:             replace $e_i$ and $e_{i+1}$ by $s$ in $\pi$
7:             $i \leftarrow \max\{i-1, 0\}$
8:         else
9:             $i \leftarrow i + 1$
10:        end if
11:    end while
12:    return $\pi$
13: end procedure
```

We now look at the two edges e_{child_1} and e_{child_2} which are bridged by e' and consider 2 cases:

The first case is $o = 2$. Then $e''_1 = e_{\text{child}_1}$ and $e''_2 = e_{\text{child}_2}$. But then e''_1 and e''_2 could have been bridged with the shortcut e'_i, which contradicts that π'' is a valid CH-representation.

The other case is $o > 2$. Then at least either e_{child_1} or e_{child_2} represents a path which is represented by at least two consecutive elements of $e''_1 e''_2 \dots e''_o$. That is again because non-containing overlap can not happen. We inspect this child edge and proceed recursively. The recursive inspection must end after a finite number of steps since the represented path shortens by at least one edge in every step and therefore we always come to a contradiction. □

7.2.1.2. Computation

Given a precomputed CH graph, we construct a CH-representation for each trajectory $t \in \mathcal{T}$, that is, we transform the path $\pi = e_0 e_1 \dots e_{k-1}$ with $e_i \in E$ in the original graph into a path $\pi' = e'_0 e'_1 e'_2 \dots e'_{k'-1}$ with $e'_i \in E^+$ in the CH graph.

Our algorithm to compute a CH-representation is quite simple: We repeatedly check if there is a shortcut bridging two neighboring edges e_i and e_{i+1}. If so, we substitute them with the shortcut. We do this until there are no more such shortcuts. See Algorithm 7.1 for the pseudocode and Figure 7.1 for an example. Note that uniqueness of the CH-representation can be proven and therefore it does not matter in which order neighboring edges are replaced by shortcuts. The running time of that algorithm is linear in the number of edges:

Theorem 7.2.3
The CH-representation of a trajectory $\pi = e_0 \dots e_{k-1}$ can be computed in $O(k)$.

Proof. During CH construction we can store all neighbors of a node v in a hash map, allowing for constant access time, see [FKS84]. Given two edges $e_1 = (v_1, v_2)$ and $e_2 = (v_2, v_3)$, we can then check for and retrieve a shortcut (v_1, v_3) in constant time. Thus, the body of the while-loop in Algorithm 7.1 takes $O(1)$ time. However, the while-loop can be entered at most $2k$ times since in every round either i is increased or the length of π decreases by one. □

For the resulting CH-representation $\pi' = e'_0 e'_1 e'_2 \dots e'_{k'-1}$, the repeated substitution of an $e'_i \in E^+ \setminus E$ with the two edges it represents (also called *child edges*) yields the original path π. Thus, we have a lossless compression scheme with respect to the spatial information of t. Note that by switching

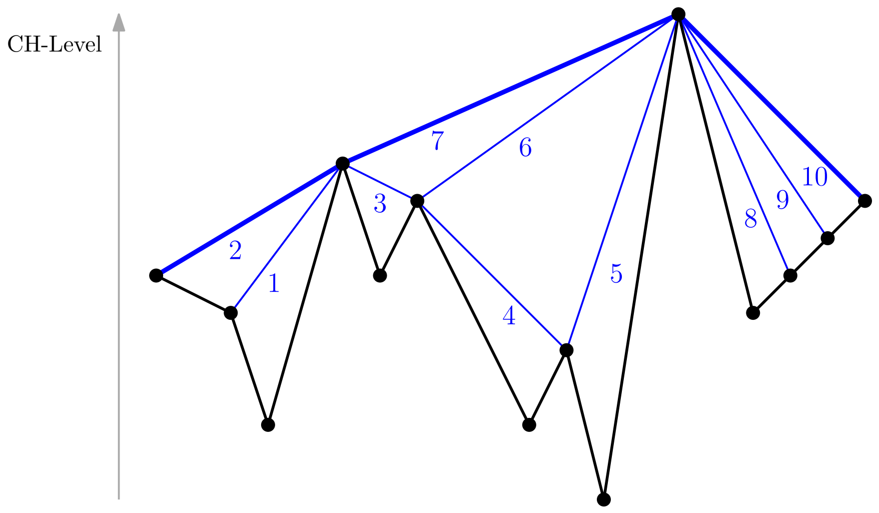


Figure 7.1.: Original path (black, 13 edges) and derivation of its CH-representation (bold blue, 3 edges) via repeated shortcut substitution (in order according to the numbers).

to the CH-representation, we can achieve a considerable compression rate in case the trajectory is composed of few shortest paths (as shortest paths usually have a very concise CH-representation, see e.g. [FS15]).

7.2.2. Retrieval Overview

At a high level, the idea of our retrieval process is to associate a trajectory with all edges of its *compressed representation* in E^+. Storing a huge number of trajectories within the index is only feasible with that compression. Answering a spatial query then boils down to finding all edges of the CH for which a corresponding path in the original graph intersects the query rectangle. Typically, an additional query data structure would be used for that purpose. Yet, we show how to utilize the CH itself as a geometric query data structure. Details are given in Section 7.2.3 and improvements in Section 7.2.4. After this step, however, some of the edges we retrieve represent edges or paths which do not actually intersect the query rectangle. This overestimation is due to checking the intersection with the bounding box and not the actual underlying path. Therefore, these 'pseudo-intersecting' edges need to be filtered out by closer inspection. The details are explained in Section 7.2.5. Finally, we only have to return those trajectories which are associated with any of the edges remaining after the filtering step. High-level pseudocode is provided in Algorithm 7.2.

Algorithm 7.2 Spatial PATHFINDER Algorithm

```
1: procedure PATHFINDERQUERY($Q$)
2:     $E_O \leftarrow$ FINDEDGECANDIDATES($Q$)
3:     $E_r \leftarrow$ REFINEEDGECANDIDATES($Q, E_O$)
4:     return GETASSOCIATEDTRAJECTORIES($E_r$)
5: end procedure
```

Algorithm 7.3 The algorithm to find edge candidates given a query rectangle Q.

```
 1: procedure FindEdgeCandidates($Q$)
 2:     $V_T \leftarrow$ FetchTopNodes($Q$)
 3:     $E_O \leftarrow \emptyset$
 4:     for $v \in V_T$ do
 5:         $E_O \leftarrow E_O \cup$ FindCandidatesForNode($v, Q$)
 6:     end for
 7:     return $E_O$
 8: end procedure
 9: procedure FindCandidatesForNode($v, Q$)
10:     $C \leftarrow \emptyset$
11:     for $e \in$ down edges of $v$ do
12:         if $PB(e) \cap Q \neq \emptyset$ then
13:             $C \leftarrow C \cup \{e\}$
14:         end if
15:         $v_l \leftarrow$ lower node of $e$
16:         if $DB(v_l) \cap Q \neq \emptyset$ then
17:             $C \leftarrow C \cup$ FindCandidatesForNode($v_l, Q$)
18:         end if
19:     end for
20:     return $C$
21: end procedure
```

7.2.3. Finding Edge Candidates

Let us now explain the details of the function called in Line 2 of Algorithm 7.2. This requires two central definitions:

- With $PB(e)$ we denote the *path box* of an edge e. It is defined as the bounding box for the path that e represents in the original graph G in case $e \in E^+$ is a shortcut, or simply the bounding box for the edge e if $e \in E$.
- We define the *downgraph box* $DB(v)$ of a node v as the bounding box of all nodes that are reachable from v on a down-path (only visiting nodes of decreasing CH-level), ignoring the orientation of the edges. In Figure 7.2, the downgraph boxes of the green/blue/red nodes are depicted in light green/blue/red.

Both $PB(e)$ and $DB(v)$ can be computed for all nodes and edges in linear time via a bottom-up traversal of the CH in a preprocessing step and *independently* of the trajectory set to be indexed.

For a spatial-only window-query with query rectangle Q, we start traversing the CH level-by-level in a top-down fashion, first inspecting all nodes which do not have a higher-level neighbor (note that there can be several of them in case the graph is not a single connected component). We can check in constant time for intersection of the query rectangle and the downgraph box of a node, only continuing with children of nodes with non-empty intersection. We call the set of nodes with non-empty intersection V_Q. The set of candidate edges E_O are then all edges adjacent to a node in V_Q. See Algorithm 7.3 for the pseudocode.

Let us first prove the following lemma, which we then use to prove correctness of our query routine.

Lemma 7.2.4
For every edge $e = (v, w)$ or $e = (w, v)$ with $l(v) < l(w)$, $DB(w)$ contains the path represented by e.

Proof. We prove the lemma by induction over the nesting depth of e. Additionally, assume that $e = (v, w)$ as the proof for $e = (w, v)$ is equivalent. Clearly, if $nd(e) = 0$, then e is an original (non-shortcut) edge, the lemma follows trivially since $DB(w)$ by definition contains v and w. Now consider the case when $nd(e) > 0$. Then e is a shortcut bridging edges $e_1 = (v, u), e_2 = (u, w)$ with smaller nesting depths. By the induction hypothesis, $DB(v)$ contains the path represented by e_1 since $l(v) > l(u)$, and $DB(w)$ contains the path represented by e_2 by the same argument. As $l(v) < l(w)$ and thus v is reachable on a down-path from w, $DB(w)$ contains the path represented by e. □

A simple application of this lemma shows that every edge that has to be reported is found by our query routine.

Theorem 7.2.5
Every edge e which represents a path π intersecting Q is adjacent to a node in V_Q.

Proof. Consider an edge $e = (v, w)$ whose represented path intersects Q, w.l.o.g. $l(v) < l(w)$. By the previous lemma, we know that $DB(w)$ contains the path represented by e. As all the ancestors of w also contain $DB(w)$, the search from the root will indeed reach w and thus $w \in V_Q$. Finally, note that e is adjacent to w. □

7.2.4. Improvements

In the following, we explain several significant improvements to the just described edge retrieval data structure. The first improvement addresses how to quickly retrieve the top nodes. The following three improvements are related to pruning the search on the CH graph. Finally, we explain how to parallelize the search.

R-Tree for Top Nodes

In Line 2 of Algorithm 7.3 we fetch the relevant top nodes. Recall that top nodes are all the nodes $v \in V$ which have no edge to a higher level node. Continental road networks are often not connected and hence several top nodes might exist. By organizing the top nodes with their downgraph boxes in an *R-tree* [Gut84] we can quickly identify the top nodes $v \in V$ for which $DB(v) \cap Q \neq \emptyset$ and continue with them as in Algorithm 7.3.

Downgraph Box Contained in Query Rectangle

If we notice during the CH-traversal that a *downgraph box* is completely contained in the query rectangle, we do not need to check the spatial constraint for all of its child *downgraph boxes* anymore, see Figure 7.2.

Obsolete Edges

Edges not associated with any trajectory and whose lower-leveled end node can be reached via other non-obsolete edges are marked obsolete and can therefore be omitted in the search. Note that the marking of obsolete edges is not uniquely determined. In this work, we greedily mark the obsolete edges. After having marked the obsolete edges, we sort our edge lists accordingly, such that we can directly access the non-obsolete edges without having to check for the obsoleteness status at query time.

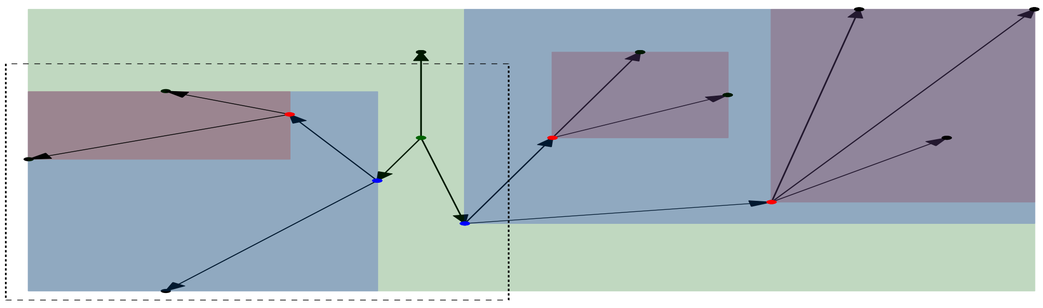


Figure 7.2.: Dashed query rectangle: The left blue *downgraph box* is fully contained in the query rectangle, therefore we know that the inner red one is contained too.

Tree Edges

During the CH-traversal, it suffices to visit a node once. To speed up the CH-traversal, we reduce the DAG to a tree in the preprocessing by greedily 'deleting' edges. We store the tree by marking the tree edges, which is very similar to marking obsolete edges. The tree edges are a subset of the non-obsolete edges.

Parallelization

By imposing a tree structure on the CH graph, parallelization of the CH graph also becomes straightforward via a breadth-first search where in each round we have a set of nodes (on the same CH level) which needs to be processed. This set is split up among the threads which process their part and then return a set of nodes to be processed in the following round. All the returned sets of the threads are merged and become the set of nodes which needs to be processed in the next round.

7.2.5. Refining Edge Candidates and Retrieving Associated Trajectories

Having retrieved the candidate edges, we now have to filter out edges e for which only the path box $PB(e)$ intersects but not the path represented by e. For simplicity, let us first focus on a single edge $e \in E_O$. If e is a non-shortcut edge, we can easily decide whether e intersects Q. However, if $e = (v, w)$ is a shortcut, more effort might be necessary. Clearly, if $PB(e) \cap Q = \emptyset$, then e must not be reported, but if $v \in Q$ or $w \in Q$, e must be reported. The interesting case is thus when $PB(e) \cap Q \neq \emptyset$ but $v, w \notin Q$. This setting is shown in Figure 7.3. In this case we need to recursively unpack e to decide whether e (and the trajectories associated with e) has to be reported. As soon as one child edge reports a non-empty intersection in the recursion, the search can stop and e must be reported. We call the set of edges that results from this step E_r. Note that in practice, it is almost never necessary to completely unpack an edge for a definitive decision.

Our final query result is all the trajectories which are referenced at least once by the retrieved edges, i.e. those $t \in \bigcup_{e \in E_r} \mathcal{T}_e$.

7.2.6. Discussion

In some sense our index structure could be interpreted as a much improved inverted index, where (a) we do not only use original edges but also CH shortcut edges to represent a trajectory if possible and

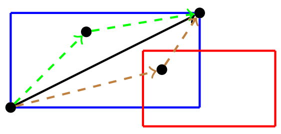


Figure 7.3.: Unclear intersection of edge and query rectangle (red): unpacking could result in a path which does intersect the red query rectangle (brown) or not (green).

(b) instead of scanning all edges we instrument the constructed contraction hierarchy as a spatial index. Improvement (a) not only dramatically decreases the space consumption of the index structure compared to a naive inverted index but also limits the number of edge-trajectory associations to collect. Additionally, (b) drastically cuts down on the edges whose associations one needs to consider at all. It is also noteworthy that a considerable part of the index construction does *not depend on the actual trajectory set to be indexed*. In particular, the construction of the CH itself as well as the path and downgraph boxes are just based on the structure of the network itself. Only the tagging of obsolete and tree edges actually depends on the trajectory set. This is in stark contrast to, e.g., SPNET [KJT16] where the spatial index is a partition guided by the trajectory set.

7.3. Adding Temporal Information

7.3.1. Time Stamps

Time stamps of a trajectory t are annotations to its nodes. In the CH-representation of t, we omit nodes via shortcuts hence losing some temporal information. Yet, PATHFINDER will always answer queries conservatively, i.e., returning a superset of the exact result set. It has been observed in [KJT16] that "fine-grained temporal information on trajectories has limited value for spatio-temporal filtering", so we do not expect this to be a real issue in practice, but will verify this conjecture experimentally as well.

7.3.2. Time Intervals

Like the spatial bounding boxes $PB(e)$, we store time intervals to keep track of the earliest and latest trajectory passing over an edge. Similar to $DB(v)$ we compute minimal time intervals containing all time intervals associated with edges on a down-path from v. This allows us to efficiently answer queries which specify a time interval $[\tau_l, \tau_u]$. Like the spatial bounding boxes, we use these time intervals to prune tree branches when they do not intersect the time interval of the query.

An edge is associated with a set of trajectories, each of which we could check for the time when the respective trajectory traverses the edge. It is more efficient to store for all trajectories traversing an edge their time intervals in a so-called *interval tree* [BCKO08]. By that we can efficiently retrieve the associated trajectories matching the time interval constraint of the query for a given edge. An interval tree storing ℓ intervals has space complexity $O(\ell)$, can be constructed in $O(\ell \log \ell)$, and can retrieve all intervals intersecting a given query interval in time $O(\log \ell + o)$ where o is the output size.

7.3.3. Time Slices

We first define *time slices* more formally. Let p be the period length and k the number of slices, we define $S_{all} = \{0, \dots, k-1\}$ to be the set of slices and $S_q \subseteq S_{all}$ a query set of slices. For example, when the period is a week and the slices are the days of the week, we have $k = 7$ and $S_q = \{5, 6\}$ for Saturday and Sunday. Formally, the set of times of a time slice query is given by

$$\bigcup_{i \in \mathbb{N}, j \in S_q} \left[\left(i + \frac{j}{k} \right) \cdot p, \left(i + \frac{j+1}{k} \right) \cdot p \right],$$

where we assume that the zero time stamp marks the beginning of a new period due to simplicity. Otherwise, we have to adjust by a constant offset.

This enables queries for periodic time intervals. In our implementation, we set p to be one week and split it into $k = 64$ time slices. To get the corresponding time slices of a time interval, we need to compute the time slices of its borders. For a given border τ, we first compute its time stamp within the period $\tau_p = (\tau \mod p)$. Then we calculate the slice with $j = \lfloor \frac{\tau_p}{k} \rfloor$. As we know start and end slice, we can easily determine the set S_{slices} of slices $[\tau_l, \tau_u]$ falls into. We store S_{slices} as bitset with size k in which all bits for $j \in S_{\text{slices}}$ are set, whereas all $j \in S_{\text{all}} \setminus S_{\text{slices}}$ are not set. Unification and intersection of time slice sets are simple bitwise 'AND' and 'OR' operations in this implementation. With unification and intersection, we build analogues of $PB(e)$ and $DB(v)$ to speed up queries which specify S_q.

At this point our PATHFINDER structure can also answer queries like "return all trajectories intersecting $[x_l, x_u] \times [y_l, y_u]$ on Monday and Friday afternoons in March 2017".

7.4. Experiments

We implemented the described algorithms in C++. Experiments were conducted on two machines:

1. AMD Ryzen Threadripper 1950X (16-Core), 128 GB RAM and a 512GB Toshiba OCZ RD400 NVMe SSD (2.6 GB/s)
2. Intel(R) Xeon(R) CPU E5-2650 v4 (24-Core), 768 GB RAM

Note that the 768GB of RAM of the second machine are humble in comparison to [KJT16] which used 2TB of RAM for far smaller graphs (800k nodes for the Danish road network) and fewer trajectories.

7.4.1. Graph Data

We build our graphs from OpenStreetMap (OSM) data.[1] From the available networks, we chose to use the German and the European graph and constructed the respective CH-graphs G_{ger} and G_{eu}. In both cases, all types of path segments – from hiking trails to motorways – are included, and the construction of the CH roughly doubles the number of edges. G_{ger} has over 57 million nodes and almost 250 million edges, G_{eu} 437 million nodes and 1.7 billion edges; see Table 7.1 for the details. The maximum level of the CH never exceeded 560, which also bounds the depth of our search procedure. For our experiments with real trajectory data on G_{ger}, we used our medium sized Threadripper machine. The Xeon machine was only necessary for G_{eu} and very large synthesized data sets.

Table 7.1.: Characteristics of considered networks ($M = 10^6$).

	Germany	Europe
# nodes	$57.4M$	$437.4M$
# edges (original)	$121.7M$	$902.1M$
# edges (CH)	$248.4M$	$1694.2M$
CH construction time (min)	16	125

Table 7.2.: Characteristics of trajectories in $\mathcal{T}_{\text{ger,real}}$.

	$\varnothing$	σ	max.
# shortest paths	11	16.6	1118
length (km)	14.78	34.6	1928
original length (#edges)	347	654	46112
compressed length (#edges)	37	56	3726

7.4.2. Real-World Trajectory Data

For real world data, we considered all traces within Germany in the bundled public collection of GPS traces from OSM[1], only dropping low quality traces (e.g., due to extreme outliers, non-monotonous time stamps, ...). As a result, we obtain 350 million GPS measurements that are matched to G_{ger} with the map matcher from [Sey17] to get a dataset with 372,534 trajectories which we call $\mathcal{T}_{\text{ger,real}}$.

We consider the number of shortest paths a trajectory consists of, its length in kilometers, and its length in edges given the original (non-CH) representation as insightful quantities, which can be found in Table 7.2 with average, standard deviation and maximum. Note that on average, a trajectory can be represented by 11 shortest paths. Since the OSM data set is highly heterogeneous, as users can upload all sorts of trajectories from short hiking trips to long road trips, the maximum values are far from the average.

7.4.3. Compression

The original edge representation of $\mathcal{T}_{\text{ger,real}}$ consists of 121.8 million edges (992 MB on disk), whereas the CH-representation only requires 13.8 million edges (112MB on disk). The actual compression for the 372k trajectories took 42 seconds, that is, around 0.1ms per trajectory. No further compression technique was employed.

As the relationship between the number of edges in the original representation and the CH-representation is of great interest for the quality of the latter as a compression scheme, we compiled various characteristic aspects in Table 7.2. To no surprise, the CH-representation is significantly more compact.

7.4.4. Synthesized Trajectory Data

To demonstrate that Pathfinder scales well, we additionally generate trajectory data ourselves by randomly choosing a source node v_s and a target node v_t. If the great-circle distance between v_s and

[1] https://download.geofabrik.de/

[1] https://planet.openstreetmap.org/gps/gpx-planet-2013-04-09.tar.xz

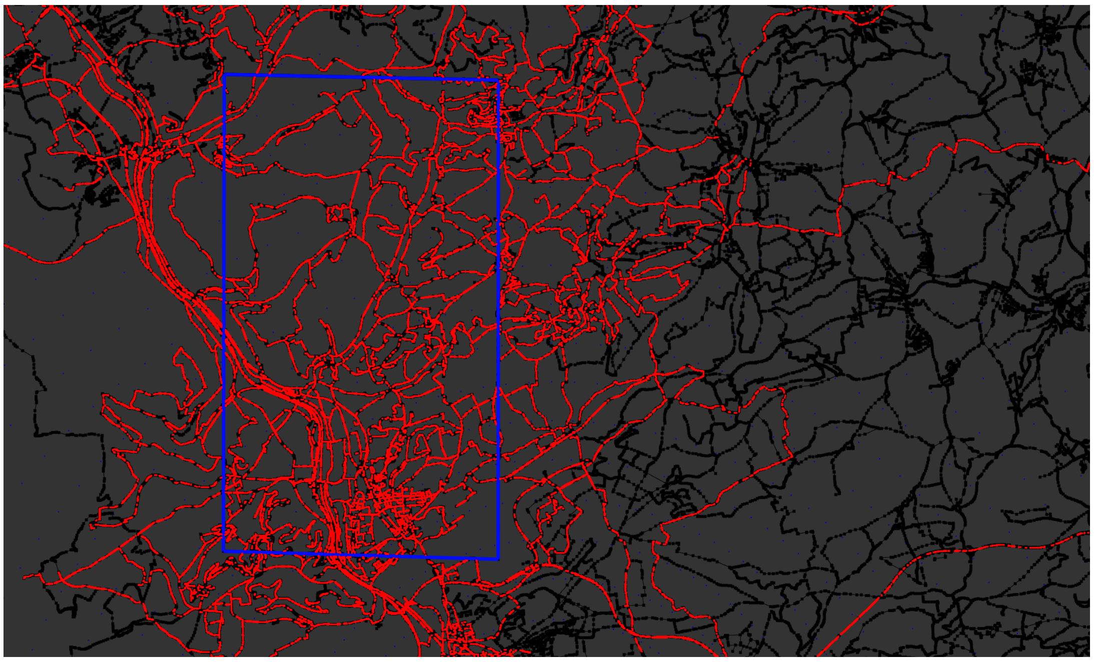


Figure 7.4.: Visualization in the German region Saarland. The blue rectangle is the query, the red lines are the returned trajectories of $\mathcal{T}_{\text{ger,real}}$. Edges which are not contained in any retrieved trajectory are drawn black.

v_t is below a given parameter d, we include the shortest path in our test data $\mathcal{T}_{\text{synth}}$. By varying d we can investigate the influence of the length of the trajectories on our data structure.

Compared to a real world trajectory which consisted on average of 11 shortest paths, t is only one shortest path by construction and has therefore a far conciser CH-representation. To make our experiments more realistic, we did not compute the CH-representation for the whole trajectory t but we represent it by the concatenated CH-representations of a number of shortest path segments instead. We sampled the number of such segments uniformly between 8 and 14 to get an expected average of 11. This may look like a negligible tweak, but it worsened the query time in our experiments by up to a factor of 3. To generate time data for t, we set τ_0 to a random date from 2008 onwards. For $i \in \{1, \dots k\}$ (with k being the uncompressed path length) we set $\tau_{i+1} = \tau_i + \tau_{\text{step}}$, where τ_{step} is sampled uniformly between 1 s and 9 s to get an expected average of 5 s which is the median edge time of $\mathcal{T}_{\text{ger,real}}$.

7.4.5. Index Structure

In Table 7.3 we state the setup time and space requirement of our index structure for the real-world trajectory set on the network of Germany and the synthetic trajectory set on the network of Europe. Note that currently the construction does not make use of parallelism at all when building the index structure. A considerable speed-up by parallelization can be expected there. For additional insights into the two data sets, we give the histograms of how many trajectories are associated with each edge, see Figure 7.5.

Table 7.3.: Setting up the auxiliary data structures (CH construction and compression excluded). $|\mathcal{T}_{\text{ger,real}}| = 372,534$ and $|\mathcal{T}_{\text{eu,synth}}| = 10^7$, $d = 10^5$ km.

	$\mathcal{T}_{\text{ger,real}}$	$\mathcal{T}_{\text{eu,synth}}$
setup time	349s	5040s
total size	126GB	485GB

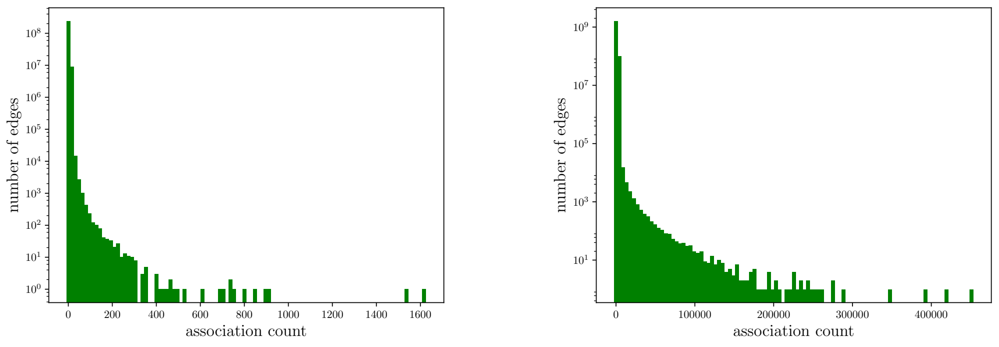


Figure 7.5.: The histograms of edge-trajectory associations for $|\mathcal{T}_{\text{ger,real}}| = 372,534$ (left) and $|\mathcal{T}_{\text{eu,synth}}| = 10^7$, $d = 10^5$ km (right).

7.4.6. Query Answering

Since we expect practical applications to query for different sized rectangles, we tested our algorithms in all experiments with rectangle sizes of different orders of magnitude. For the magnitude parameter r within the range $\{1, \dots, 5\}$, we created a query rectangle $R_r = [x_l, x_u] \times [y_l, y_u]$ as follows: First we compute the bounding box R_G of the graph. Then, a random node of our graph is sampled and taken as the lower left corner of the query rectangle. Finally, we set the width and height of R_r to 2^{-r} of the width or height of R_G respectively. A visual depiction of an answered query can be seen in Figure 7.4.

To evaluate performance with time interval constraints we also generate intervals of the form $[\tau_l, \tau_u]$, where τ_l lies in the time frame of the given trajectory data and the difference between τ_l, τ_u is chosen to be one week. For the time slices we constructed queries which ask for all trajectories in a region which happened on a certain day of the week.

The measurements are obtained by averaging the time for 100 queries with the same configuration. For each cell in one column, we used the same 100 queries.

Comparison to Baselines

In order to demonstrate the efficiency of Pathfinder, we first compare it to the *linear scan* algorithm and to the *inverted index* approach described in Section 6.1.1. However, we do not use the most naive variant of the inverted index here but already the tuned variant where we index the trajectories in the CH graph instead of the original graph. Note that the naive variant does not allow for any compression and exhibits worse query times. Thus, using the inverted CH index already provides some improvements. However, the full power of Pathfinder is only achieved by augmenting the CH with a spatial index structure.

Table 7.4 shows the measured query times for all three approaches for different sizes of the query rectangle. We notice that Pathfinder is faster than the naive approaches by several orders of

Table 7.4.: Timings in seconds for $|\mathcal{T}_{\text{ger,real}}| = 372,534$ for different variants with spatial only queries, single thread.

	1/2	1/4	1/8	1/16	1/32
linear scan	74.53	79.65	82.37	83.33	83.90
inverted index CH	87.26	69.78	60.05	56.45	55.37
PATHFINDER	0.89	0.39	0.14	0.04	0.01

Table 7.5.: Timings in seconds for different sized sets $\mathcal{T}_{\text{ger,synth}}$, spatial only queries, single thread.

set size	100,000	1M	10M
linear scan	156.237	1568.022	15884.508
inverted index CH	69.577	68.403	71.550
PATHFINDER	0.029	0.107	0.415

Table 7.6.: Timings *in milliseconds* for $|\mathcal{T}_{\text{ger,real}}| = 372,534$ with 16 threads for different constraints.

	1/2	1/4	1/8	1/16	1/32
pure spatial	108.7	48.5	15.9	7.2	3.2
intervals	12.1	6.2	3.6	2.3	1.9
slices	44.0	20.5	8.5	3.6	2.4
intervals + slices	10.0	5.8	3.2	2.0	1.5

magnitude, especially for small rectangles. At first glance, it is surprising that the approach using the inverted index is sometimes slower than the linear scan. This can be explained by our relatively sparse trajectory dataset. To confirm the asymptotic benefit of the inverted index, we ran tests with larger synthesized data sets for which the results can be found in Table 7.5. There we can see that the time complexity for the linear scan is unsurprisingly linear in $|\mathcal{T}_{\text{synth}}|$. As the time complexity for the inverted index approach is not, we can also see that it outperforms the linear scan as the number of trajectories increases. PATHFINDER is several orders of magnitude faster than the inverted index approach.

In Table 7.4, the 12 ms taken by PATHFINDER for the 1/32 sized query allows a good comparison to TED: In [YWY+18], it is reported in Figure 17f that TED requires more than 40 ms for its *window* queries in a similar setting (500k trajectories) with the exception of geographical graph size. Their graph is only Singapore (20k vertices compared to 57.4 million in our case). As time complexity with respect to the number of trajectories is linear for TED while it is not for PATHFINDER, this difference only grows when going to larger scales.

Parallelization

In Table 7.6 we show the times for different variants using parallelization. We can see that for smaller rectangles, a query can be answered within a few milliseconds. Adding a time slice constraint significantly reduces the required time because the output set $\mathcal{T}_{\text{out}}$ becomes smaller and parts of the traversed tree can be pruned. Specifying a time interval constraint leads to even lower response times because, on one hand $\mathcal{T}_{\text{out}}$ is even smaller, and on the other hand we used interval trees to speed up such queries.

As already mentioned, we synthesized big datasets to test the scaling behaviour of PATHFINDER

Table 7.7.: Measurements for $|\mathcal{T}_{\text{eu,synth}}| = 10^7$ with 24 threads for spatial only queries.

d	1/2	1/4	1/8	1/16	1/32
25km	0.817	0.628	0.270	0.093	0.041
100km	5.630	4.381	1.866	0.609	0.201
400km	6.177	4.959	2.124	0.685	0.222
100,000km	6.889	4.773	1.882	0.608	0.211

(a) Overall query time in *seconds*

d	1/2	1/4	1/8	1/16	1/32
25km	0.254	0.191	0.088	0.037	0.021
100km	1.219	0.886	0.372	0.140	0.050
400km	1.191	0.893	0.377	0.140	0.052
100,000km	1.194	0.815	0.315	0.106	0.043

(b) FindEdgeCandidates in *seconds*

d	1/2	1/4	1/8	1/16	1/32
25km	0.280	0.207	0.086	0.025	0.007
100km	1.576	1.158	0.474	0.144	0.042
400km	1.504	1.152	0.458	0.135	0.043
100,000km	1.430	0.931	0.344	0.104	0.028

(c) RefineEdgeCandidates in *seconds*

d	1/2	1/4	1/8	1/16	1/32
25km	0.269	0.221	0.090	0.028	0.009
100km	2.754	2.269	0.992	0.314	0.103
400km	3.379	2.827	1.246	0.391	0.118
100,000km	4.090	2.879	1.139	0.360	0.119

(d) GetAssociatedTrajectories in *seconds*

d	1/2	1/4	1/8	1/16	1/32
25km	168,570	134,451	57,107	18,468	6,595
100km	2,288,640	1,948,294	903,384	314,285	122,174
400km	2,824,558	2,511,392	1,359,735	568,822	256,962
100,000km	4,781,852	4,174,645	2,577,375	1,334,826	745,257

(e) Result size $|\mathcal{T}_{out}|$

on the large graph G_{eu}. The resulting times for pure spatial queries using 24-fold parallelization can be seen in Table 7.7a. Additionally, timings for the most essential steps are shown in 7.7b, 7.7c, and 7.7d. For example, having generated 10^7 random trajectories where source and target have an Euclidean distance of at most 400km, querying with a randomly placed rectangle of 1/16th the width and 1/16th the height of the bounding box of Europe takes 0.685 seconds, reporting around 568k trajectories in the output (see Table 7.7e). Note that such a query rectangle still has width and height of several hundred kilometers. In case of such a rather large result set, most of the time is spent collecting the associated trajectories (0.391 seconds), while the other steps take only 0.140 (FindEdgeCandidates) and 0.135 (RefineEdgeCandidates) seconds. In general, though, the times for these steps are in the same order of magnitude, meaning there is no single bottleneck, which implies that all of our various optimizations are necessary as otherwise one of the steps would dominate the runtime and thereby become the bottleneck. Notably, the measured times make a big jump between $d = 25\,\text{km}$ and $d = 100\,\text{km}$. The reason seems to be the size of the result set $\mathcal{T}_{\text{out}}$, which is larger for

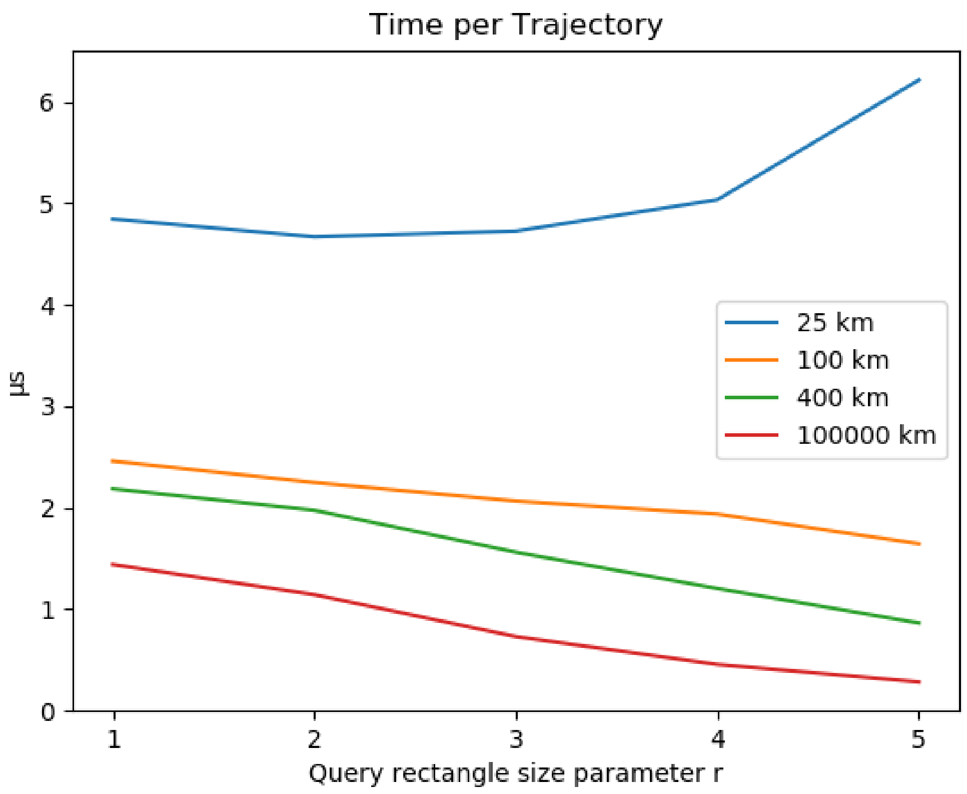


Figure 7.6.: Time per trajectory in the output for $|\mathcal{T}_{\text{eu,synth}}| = 10^7$ with 24 threads for spatial only query.

$d = 100\,\text{km}$ than for $d = 25\,\text{km}$ by at least a factor of 15 for all parametrizations of the query size, as we can see in 7.7e. Additionally, on average a trajectory in $\mathcal{T}_{\text{out}}$ has 4 times more original graph edges for $d = 100\,\text{km}$ than for $d = 25\,\text{km}$ by construction. Luckily, our CH-representation cushions this data growth. Of highest significance is that we can handle the smaller sized rectangle queries with times far below one second. We want to emphasize again that even the smaller rectangles still have a significant size as their size is chosen relative to the bounding box of Europe. Obviously, if the set of trajectories $\mathcal{T}_{\text{out}}$ to be reported is large – e.g., querying with 1/4th of the bounding box width and height of Europe, we have $|\mathcal{T}_{\text{out}}| > 4$ million – the query times must be higher. Therefore, it is of interest to consider the query time *per reported trajectory* as we have done in Figure 7.6 – essentially dividing the query times of Table 7.7a by the result set sizes of Table 7.7e. We see that the time per reported trajectory is always in the *microseconds* range.

In Table 7.9 we have similar measurements as in Table 7.6 but on larger scale. In comparison, the additional time constraints do not speed up the query times by the same amount, but the general trend stays the same and proves the scalability of PATHFINDER.

In Table 7.10 we ran only spatial queries with different numbers of threads. In Figure 7.7 these times are normalized and plotted. We can see that our algorithm has only a small speedup from 1 to 2 threads because $|\mathcal{T}_{out}|$ does not need to be merged in the single-threaded case. For larger rectangles in particular we see an almost linear speedup when using more threads.

Table 7.9.: Times in seconds for $|\mathcal{T}_{\text{eu,synth}}| = 10^7$ with $d = 100\,\text{km}$ and 32 threads for different constraints.

	1/2	1/4	1/8	1/16	1/32
pure spatial	5.630	4.381	1.866	0.609	0.201
intervals	1.439	1.095	0.532	0.174	0.076
slices	4.273	3.412	1.449	0.461	0.157
intervals + slices	1.447	1.059	0.520	0.168	0.068

Table 7.10.: Times in seconds for $|\mathcal{T}_{\text{eu,synth}}| = 10^7$ with $d = 25\,\text{km}$ and different number of threads for spatial only query.

threads	1/2	1/4	1/8	1/16	1/32
1	8.334	4.987	2.029	0.864	0.150
2	4.811	3.788	1.537	0.703	0.122
4	2.722	1.999	0.830	0.348	0.072
8	1.351	1.067	0.447	0.210	0.056
16	0.801	0.624	0.273	0.130	0.037
24	0.632	0.495	0.196	0.099	0.032

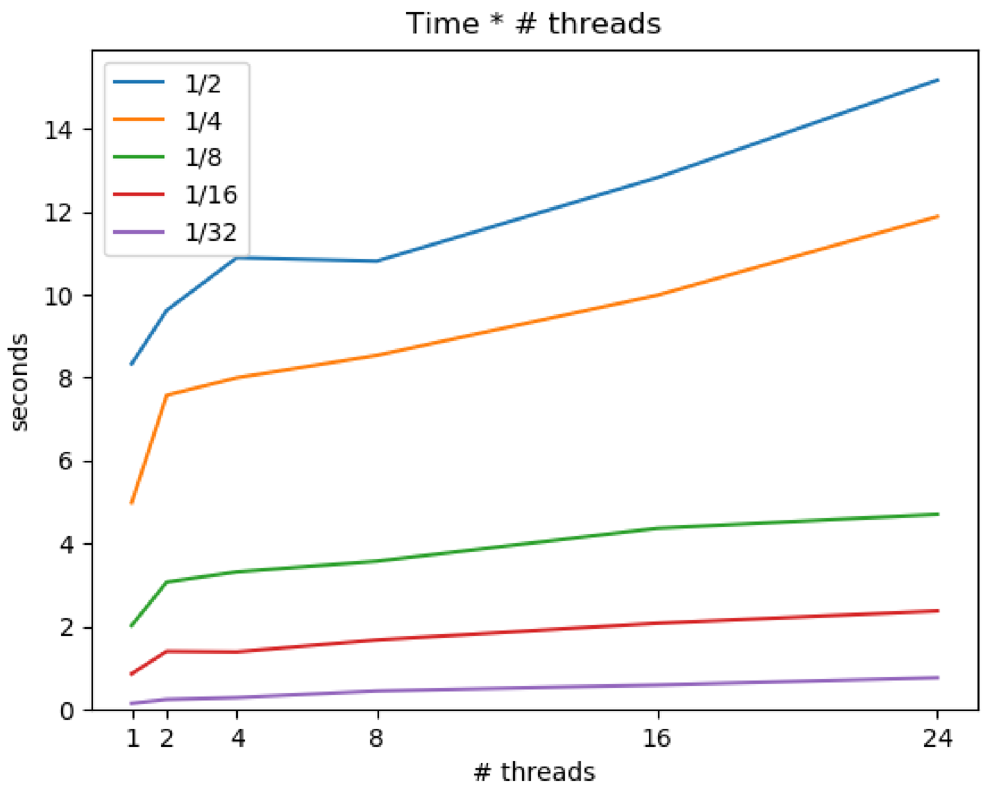


Figure 7.7.: Table 7.10 as a plot with reported times normalized by multiplication with the number of used threads.

Table 7.11.: On disk and RAM results for step GetAssociatedTrajectories and $|\mathcal{T}_{out}|$, single thread.

	1/2	1/4	1/8	1/16	1/32
On disk (s)	36.33	22.60	9.35	3.86	1.53
RAM (s)	19.20	10.67	3.56	1.11	0.26
$|\mathcal{T}_{out}|$ (M)	16.49	12.46	6.80	3.46	1.42

Table 7.12.: Size of result set $\mathcal{T}_{\text{out}}$: quality loss due to decrease of temporal resolution (averaged over 2000 queries).

	1/2	1/4	1/8	1/16	1/32
uncompressed	959.8600	313.8005	98.2340	28.5760	9.6105
compressed	960.3985	314.2435	98.4980	28.7365	9.6890
$\frac{\text{uncompressed}}{\text{compressed}}$	0.999	0.998	0.997	0.994	0.991

Index on Disk

Although we designed Pathfinder as an in-memory index we also considered the case that RAM is limited and therefore implemented a variant which stores the trajectory data on disk by using the STXXL-library [DKS08]. We generated a huge dataset of $|\mathcal{T}_{\text{ger,synth}}| = 75 \cdot 10^6$ with $d = 100\,\text{km}$ which uses 358 GB on the SSD of our Threadripper machine and ran spatial queries there and also completely in the RAM of our Xeon machine for comparison. The results with respect to the step GetAssociatedTrajectories are shown in Table 7.11. We see that fetching from a (very fast) NVMe SSD only incurs an overhead of at most a factor of 6. Note that in both cases we assume the search structure (essentially the augmented CH) to be resident in RAM, yet the number of trajectories can be increased almost arbitrarily according to the capacity of the SSD.

Precision With Respect to Temporal Queries

In [KJT16] it was argued (but not experimentally verified) that fine-grained temporal information has only limited value for spatio-temporal filtering. We verify this conjecture by analyzing the *interval* queries from Table 7.6, in particular how many more trajectories are reported due to non-exact temporal information stored in the CH representation, see Table 7.12. We see for example that for random queries of size $\frac{1}{8} \times \frac{1}{8}$ with a random slot out of 64 slots within a week, the average number of trajectories reported is 98.498 compared to a ground truth of 98.234. This is due to the fact that if a trajectory is not fully contained in the spatial query rectangle, our temporal compression might report trajectories which should not be reported (this is the only possible reason). We see, though, that this happens very rarely. We can also see that bigger query rectangles have a higher precision because their boundary-to-area ratio is smaller.

Comparison to SPNET

To make a comparison with SPNET, we have a closer look at the result for one thread and rectangle size 1/32 in Table 7.10 which is 150 ms. For SPNET, such queries were benchmarked in [KJT16] with better hardware, on a smaller graph and with fewer trajectories which were also shorter on average. Note that the input for SPNET's range query is not a rectangle but the set of edges in a rectangle. Therefore, their query is strictly easier than ours since we have to compute the edges in the rectangle first, which is exactly what FindEdgeCandidates does. For the rectangle used in our query with size 1/32, we counted the edges within the rectangle which were on average 5.18×10^6. In Figure 5 (d) of the SPNET paper [KJT16], we compare with the result for $\mathbf{R}_{19}$ which corresponds to rectangles containing up to 2^{19+1} edges. Consequently, we consider more edges since $2^{19+1} < 1.05 \cdot 10^6 < 5.18 \cdot 10^6$. Figure 5 (d) states that for $\mathbf{R}_{19}$, SPNET requires on average more than 10^3 ms, whereas Pathfinder requires 150 ms under harder conditions.

7.5. Conclusion and Future Work

We built a novel framework which delivers high compression rates for heterogeneous network constrained trajectory data. However, the main achievement of our framework is to speed up spatio-temporal range queries tremendously. An advantage of Pathfinder is that it is built upon a data structure which can also be used to speed up routing related queries. We demonstrated that Pathfinder is highly parallelizable with almost linear speedup and able to deal with much larger data sets than previous work.

We also investigated storage of trajectories in external memory on an SSD. This is possible since only the CH has to reside in RAM, and the memory consumption of the CH is *independent* of the size of the trajectory set. Since the latter can then be stored on an SSD, we are only limited by the SSD capacity for the trajectory set and the RAM for the CH representation of the network. The runtime penalty for external storage of the trajectory set is moderate in case of a fast NVMe PCIe SSD.

Pathfinder drops part of the temporal information. Even though we experimentally validated the conjecture by [KJT16] that this does not significantly affect the query results, one can simply replace some long shortcuts by shorter shortcuts or even the original edges they represent if higher precision is required at certain places. This leads to more accurate temporal resolution at the cost of higher space consumption and query time. It might be interesting to see whether more sophisticated approaches can increase the temporal resolution without affecting query times and space consumption too much.

Also, we have not discussed how to deal with dynamic updates (in particular insertion) of trajectories. Fortunately, the main components of our index structure are trajectory oblivious in a sense that they do not depend on the trajectory set to be indexed. Only the marking of obsolete and tree edges actually depends on the trajectory set. So with little effort, we are be able to cater for insertions and deletions of trajectories. Changes to the underlying road network are unfortunately more difficult to handle, and we leave this as possible future work.

Chapter 8 Visualization

This chapter is based on a selection of joint work [BFR22; BFRS22] with Lukas Baur and Stefan Funke. These papers were published in the Proceedings of the 30th International Conference on Advances in Geographic Information Systems (SIGSPATIAL 22). Some content of [BFRS22] is based on the work of Sabine Storandt to which I did not make meaningful contributions and therefore omit it here. In a master's thesis [Bau21] under my supervision, efficient visualization of trajectory data was studied and this chapter recaps some of the content and presents enhancements. I made secondary contributions in Sections 8.3.3.3 and 8.3.3.4.

8.0.1. Contribution

In this chapter, we study the problem of graph and trajectory set visualization.

- We present the framework PATHFINDER$^{\text{VIS}}$ and its various approaches to visualize trajectory sets.
- Finally, we dig deeper into one of the approaches which utilizes a similar unfolding approach as URAN but reduces topological inconsistencies. We prove theoretical bounds for the unpacking scheme and evaluate the approach experimentally.

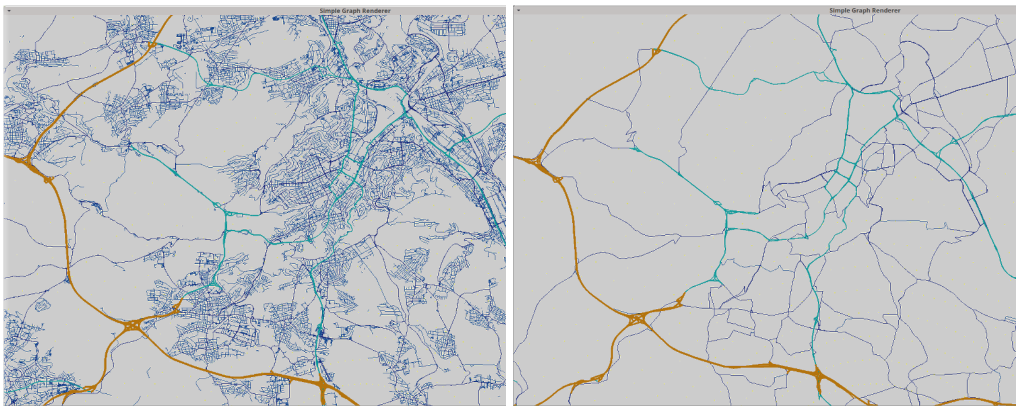


Figure 8.1.: Metropolitan area of Stuttgart, full detail (left) vs simplified (right).

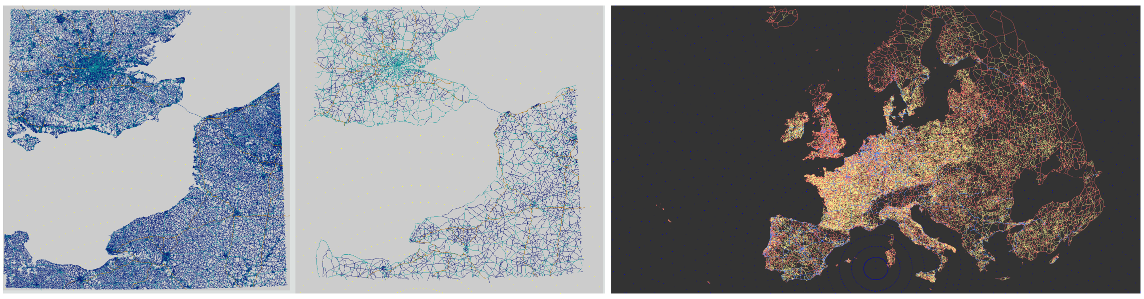


Figure 8.2.: Exemplary queries: English channel with London metropolitan area and Normandy; $L = 0$; 4,539,820 edges reported in 16ms (left); $L = 100$, 166,294 edges in 9ms (center); view on Europe $L = 74$; 4,397,822 edges in 39ms (right).

- A critical shortcoming of the URAN framework is that the silhouette of the network is not preserved for simplifications. We suggest a simple strategy for **silhouette preservation** of the network even in coarse simplifications.

8.1. PATHFINDER$^{\text{VIS}}$

8.2. Architecture

PATHFINDER$^{\text{VIS}}$ is a client-server environment written in C++ which allows for requests over the network using a Pistache [22d] server. The frontend is powered by the Bootstrap framework [22a] and the JavaScript library Leaflet [22b] for map support. The architecture is depicted in Figure 8.3 where the original PATHFINDER framework is integrated into the server as a backend which computes data for the client. The client deals with rendering and user interaction.

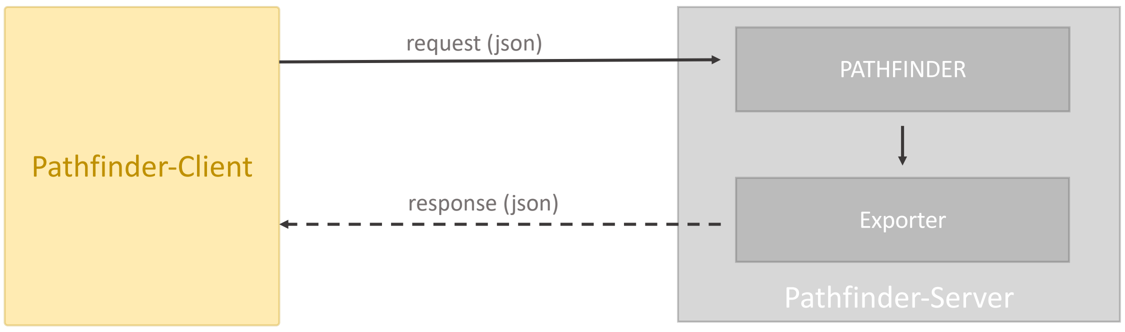


Figure 8.3.: Architecture of PATHFINDER$^{\text{VIS}}$

8.3. Features

We now present our different visualization approaches from micro to macro context. A demonstration video by Baur featuring the various approaches is also available [22c].

Note that some of the approaches, in particular the one of Section 8.3.4, are also kind of orthogonal to the others, i.e. they could be combined.

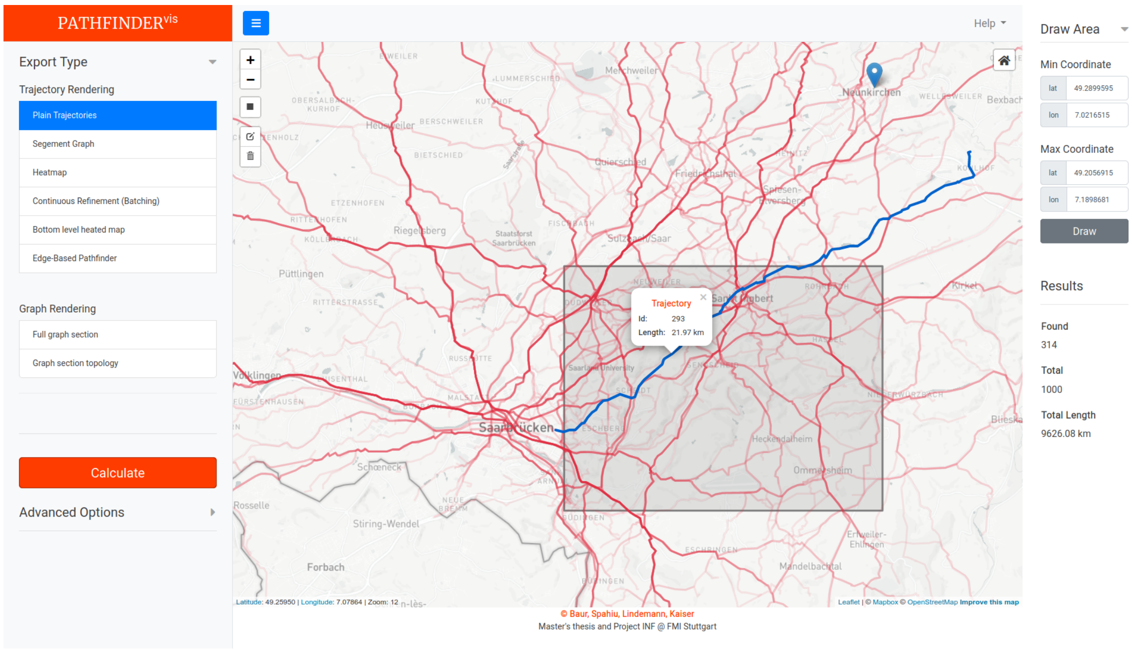
8.3.1. Plain Trajectories

The most straightforward approach is to transmit and display the original PATHFINDER backend's output, which is a list of trajectories intersecting the query rectangle. Each trajectory is represented by a list of CH-edges it uses. To distinguish between trajectories, hovering can be used to highlight a single trajectory (Figure 8.4).

Note that this representation is of compressed form, i.e. the edges may be shortcut edges. To enhance the visual quality, we replace some of these shortcuts with the edges they bridge. We call this replacement *unfolding* and continue until a certain depth is reached. This depth depends dynamically on the client's zoom level. In principle, this is very similar to URAN's simplification approach for street network visualization. The difference is that we focus on a dynamic subgraph induced by the trajectories rather than the whole static street network graph. Another general difference is that we want to highlight high-traffic edges by drawing them thicker.

8.3.2. Segment Graph

When trajectories sets grow massive in size, even transmitting and presenting the trajectories in compressed form becomes impractical. At the same time, when trajectory sets become large, many edges are used multiple times by different trajectories. As the frontend turned out to be the bottleneck



Min Coordinate: lat 49.2899595, lon 7.0216515
Max Coordinate: lat 49.2056915, lon 7.1898681

Trajectory
Id: 293
Length: 21.97 km

Found 314
Total 1000
Total Length 9626.08 km

Latitude: 49.25950 | Longitude: 7.07864 | Zoom: 12

Figure 8.4.: Plain trajectories in Saarland (Germany) with one highlighted in blue

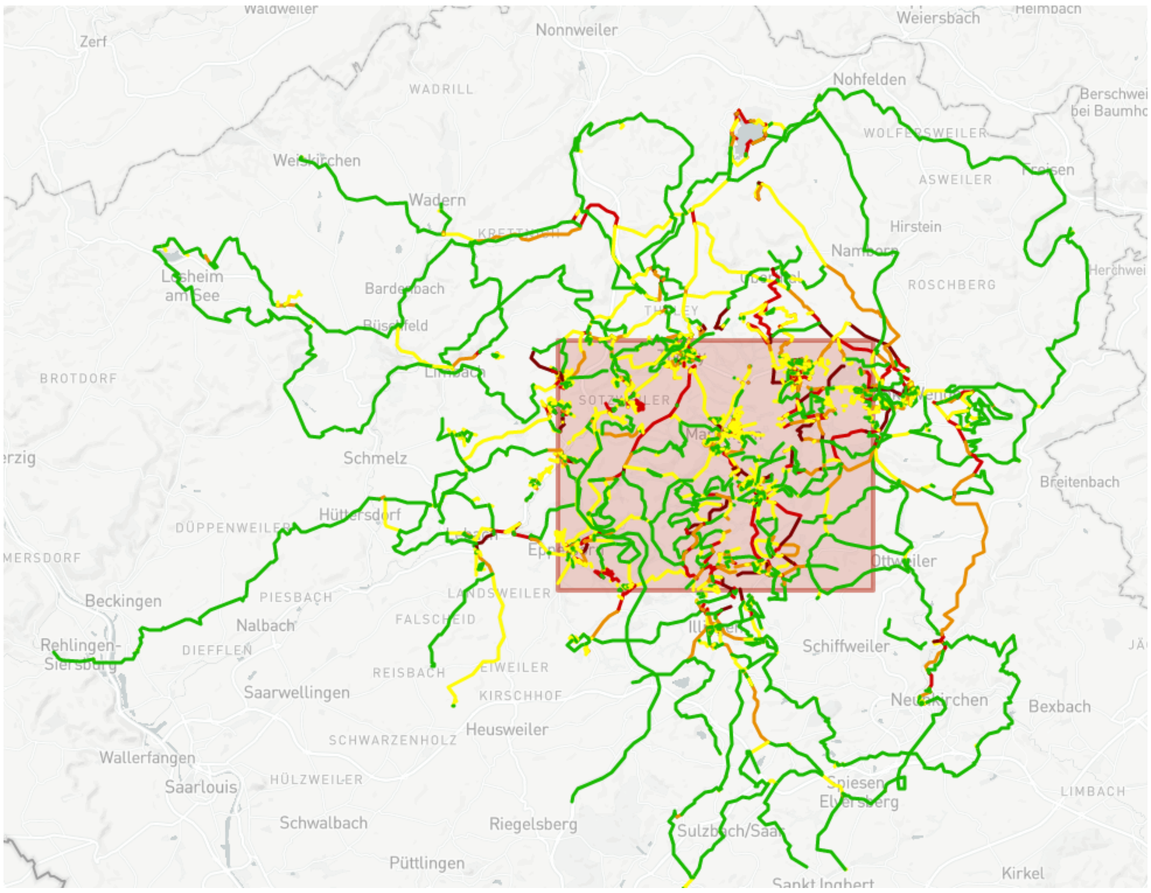


Figure 8.5.: Segment graph visualization of Saarland (Germany)

compared to the backend in our experiments, identifying multiple use of edges in the backend increased performance by only sending the edge once. This also allows to identify longest common subpaths between trajectories (Figure 8.5).

8.3.3. Edge-Based PATHFINDER

When trajectory data grows further, it even takes too much time to just iterate over the trajectory ids.

The chunk of trajectory edges which lies outside the box can be comparably large or even larger as shown in Figure 8.6 and is arguably not so much in the focus of the user as those inside. If we constrain ourselves to the edges within the query rectangle we can simply omit the final step GetAssociatedTrajectories from Section 7.2.2 and focus on the edges which are used at least once from RefineEdgeCandidates.

An example of a visualization can be seen in Figure 8.7. Note that there are edges drawn outside the rectangle which result from long shortcuts which intersected the rectangle and had to be unpacked.

Due to the way the original PATHFINDER implemented its inverted index data structure, we can easily detect edges which are not used often and prune them to get less cluttered views on a continental scale in Figures 8.8 and 8.9.

Note that unfolding without getting visual artifacts is not trivial and this is what the following sections are about.

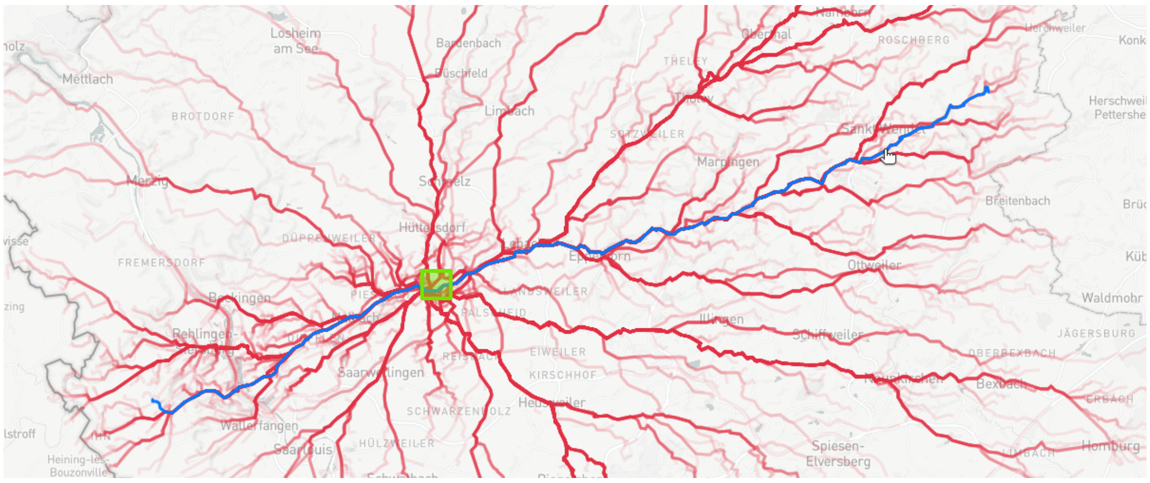


Figure 8.6.: Most of the trajectory data may lie outside of the query rectangle.

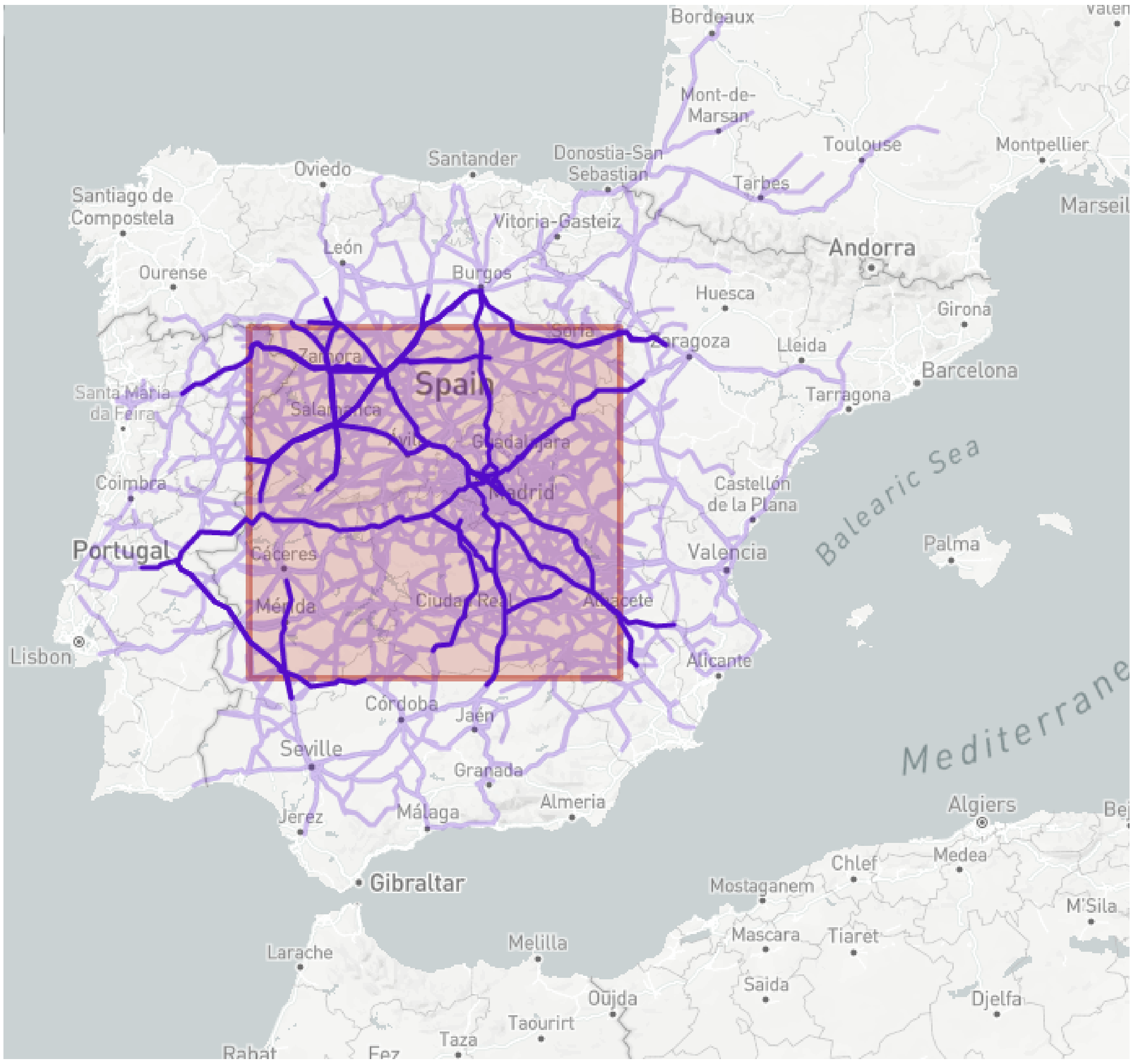


Figure 8.7.: Edge-Based PATHFINDER with query rectangle in Spain.

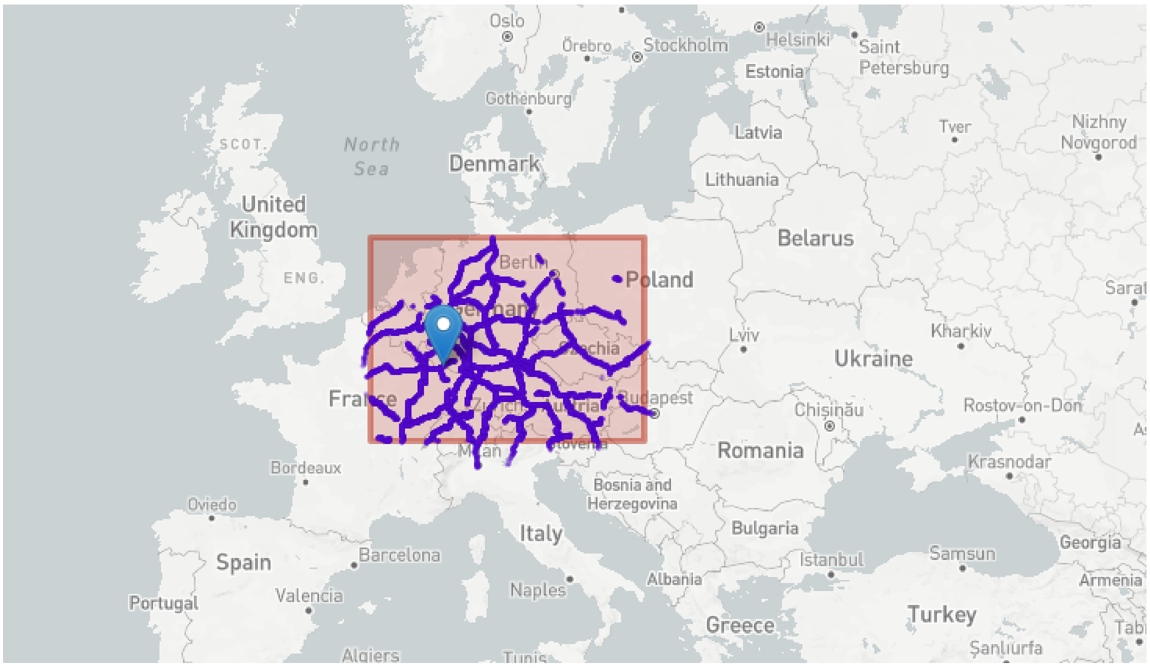


Figure 8.8.: Edge-Based PATHFINDER with query rectangle in central Europe.

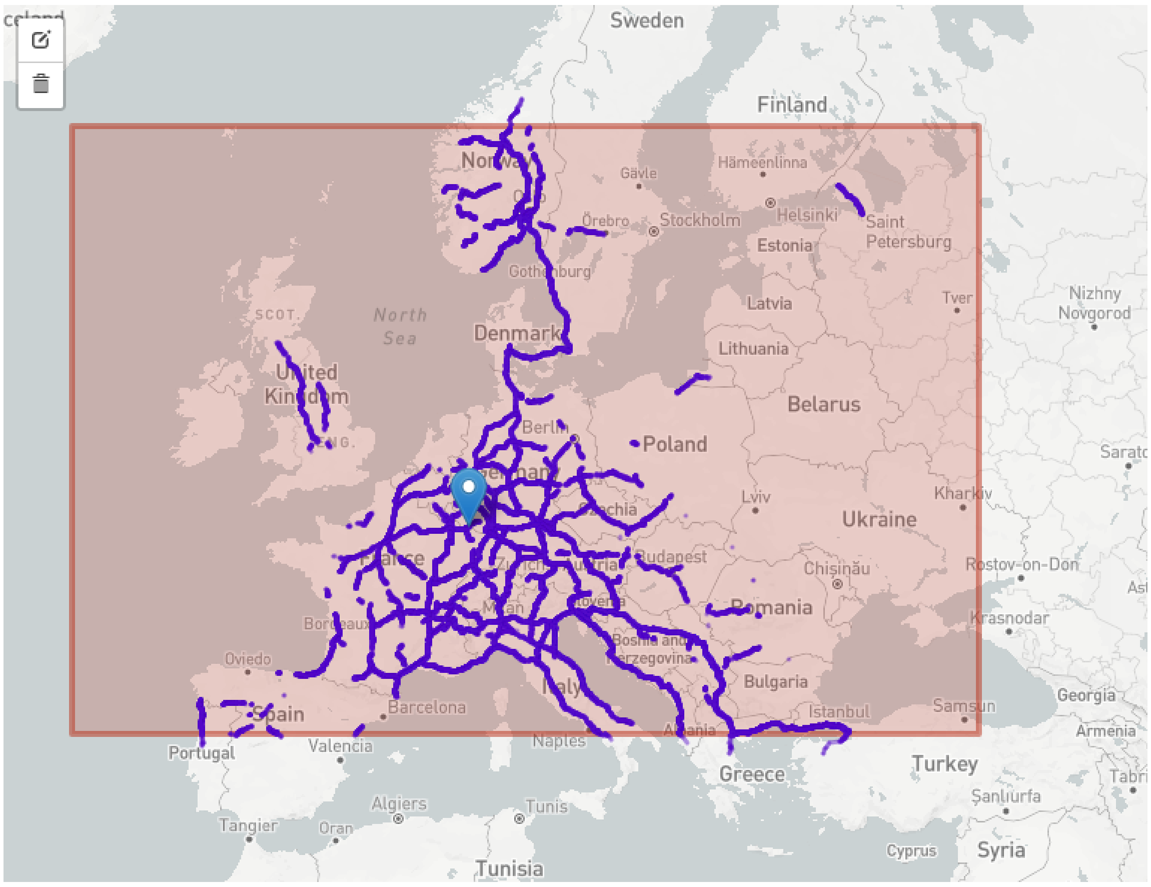


Figure 8.9.: Edge-Based PATHFINDER with big query rectangle over Europe.

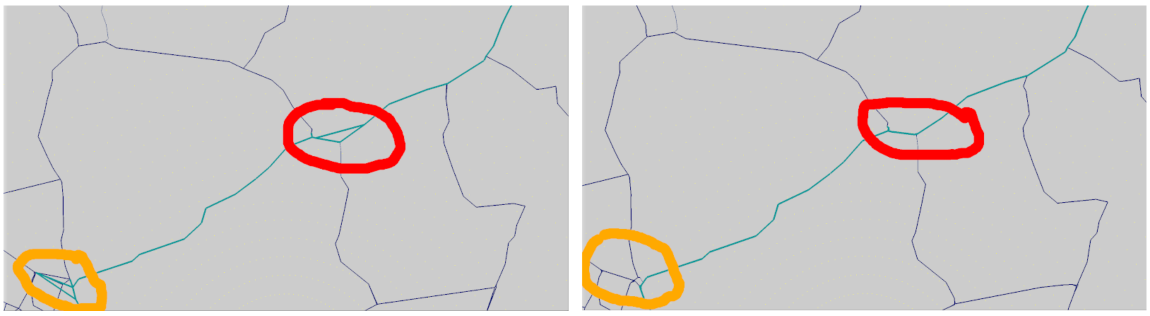


Figure 8.10.: Topological inconsistencies of complex network parts (left) vs topologically correct representation (right).

8.3.3.1. Topological Inconsistencies

One problem with the naive recursive unfolding strategy of shortcut edges as used in URAN or described in Section 8.3.1, is that it is oblivious to topological changes introduced in the contraction process. Contraction of a degree-3 node during the CH preprocessing often leads to triangle-shaped subgraphs which is a change in topology also unpleasantly visible in renderings, see Figure 8.10. This could be solved by always unfolding down to the original edges, yet this will also lead to unnecessary unfoldings and significantly increased visual complexity.

8.3.3.2. Topology Preservation via Unfolding and Refolding

One might be tempted to fix the problem of topology disturbances by modifying the contraction order accordingly. In essence, though, this means that one can only contract nodes of degree 2 or less, which would leave large parts of the network uncontracted. We propose a different approach here: while allowing an arbitrary contraction order and threshold level, we algorithmically determine which shortcuts have to be unfolded for topological correctness.

- mark all edges returned by RefineEdgeCandidates
- *completely* unfold and mark all marked edges recursively
- refold by undoing the unfolding for degree-2 nodes (degree with respect to the marked edges)

Full unfolding, of course, contradicts the purpose of approximation, which aims at reducing the number of road segments to be drawn. Note, though, that the unfolding and refolding could take place on the server in case of a client-server scenario, where the limiting factor is bandwidth for transmission, not the processing power on the server side.

8.3.3.3. Focus on Edge is problematic

In his master's thesis [Bau21], Baur tried to prevent topological inconsistencies by counting how often an edge was used. Basically, two edges were refolded to a common shortcut edge if their counters were the same. This approach gave a reasonable visualization but contained minor technical problems. For example, as we are working on directed graphs, we have forward and backward edges which we do not want to be unpacked independently. He presented some ideas for remedy but they turned out to produce other inconsistencies or technical problems.

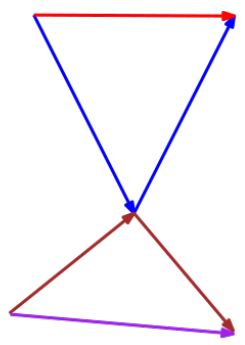


Figure 8.11.: Touching paths. The blue and brown paths touch at a common center node but their shortcut edges (red and purple, respectively) don't touch anymore.

Finally we noticed that there are some situations as in Figure 8.11 which couldn't be covered by edge counters. There, two paths are touching each other while their shortcut representations do not. As we wanted to retain such touching nodes and the edge counters can't keep track of such situations, we instead focused on storing relevant information for consistent unpacking at the nodes.

8.3.3.4. Bounds on Unpacking

Algorithm 8.1 Full unfolding based on [Bau21]

```
 1: procedure fullUnfolding($E$)
 2:     $E_{out} \leftarrow []$
 3:     $closed \leftarrow Set()$
 4:     while $|E| > 0$ do
 5:         $e \leftarrow E$.pop()
 6:         if $e \in closed$ then
 7:             continue
 8:         end if
 9:         $closed \leftarrow closed \cup \{e\}$
10:         if $e$.is_shortcut then
11:             $e_1, e_2 \leftarrow e$.children
12:             $E \leftarrow E \cup [e_2, e_1]$
13:         else
14:             $E_{out} \leftarrow E_{out} \cup [e]$
15:         end if
16:     end while
17:     return $E_{out}$
18: end procedure
```

Pseudocode for a simple unpacking algorithm without refolding can be seen in Algorithm 8.1. When the input is only a single edge the output tends to be much larger after unpacking. Moreover, the number of inspected edges during the algorithm is linear in the output in this case.

However, in [FSS17], it was observed that for a very similar unpacking algorithm, there was one instance where the output was smaller than the input, 327,569 to 451,100 edges. This can happen due to the fact that a single unpacked output edge might be part of many input edges.

Baur [Bau21] believed that the algorithm runs in linear time with respect to the output. We disprove this by showing an input and output sensitive lower bound of $\Omega(n\sqrt{n})$. We also give an upper bound for graphs with reasonable diameter.

Theorem 8.3.1
There are graphs with contraction orders where unpacking a set of input edges E_{in} into a set of output edges E_{out} requires the algorithm to inspect $\Omega(n\sqrt{n})$ where $n := \max(|E_{in}|, |E_{out}|)$ edges.

Proof. W.l.o.g. let n be of the form $n = 1 + k^2$. Our star-like graph has one central node which is connected to $\sqrt{n}$ arms which are $\sqrt{n}$ nodes long.

The central node is contracted first. Then the nodes of the first arm are contracted consecutively from inner to outer nodes except for the outermost node. Then we proceed to contract the other arms the same way until only the most outermost nodes are left. Finally we contract these remaining nodes in some order.

Contracting a node on an arm which isn't an outermost node yields $\sqrt{n} - 1$ new shortcuts. As we are contracting $\Omega(n)$ such nodes, we get $\Omega(n\sqrt{n})$ shortcuts overall.

Now we only need a set of input edges which when unpacked touches all those shortcuts. Let the set be all the shortcuts edges between the outermost nodes. This set has size $O(n)$ which concludes the proof. □

Theorem 8.3.2
For graphs with a diameter of $O(\sqrt{n})$, the number of inspected edges is not more than a factor of $O(\sqrt{n})$ larger than the input and output, i.e. Theorem 8.3.1 is tight for these graphs.

Proof. It suffices to look at the input sensitivity: If an input edge is no shortcut, i.e. an original edge, no further edges need to be inspected because of it, so it barely contributes anything to the gap. If it is a shortcut, it has to represent a shortest path in the original graph. As the diameter is in $O(\sqrt{n})$, the represented path also can't be longer than that. Incrementally compressing the original path towards our given edge shortens the size of the CH-representation in the number of edges by 1 each time we are substituting two edges with a shortcut. So this edge can contribute at most $O(\sqrt{n})$ inspected shortcuts in the reverse unpacking process. □

The constraint for the diameter $O(\sqrt{n})$ is somewhat unpleasant but note that real street networks fulfill this requirement easily.

8.3.4. Bottom Level Heat Map

For even larger zoom scales, transmitting all requested edges even in compressed form may be impractical and sometimes an overly detailed map as in Figure 8.12 (left) can even be perceived as cluttered. A different idea is to plot only the most important edges and to conflate the rest in form of a heat map. This can be seen in Figure 8.12 (right). Normally, there is the problem that one lacks a notion of importance for this distinction but in our setup the underlying CH framework inherently gives us such a notion by its eponymous nature. For contraction heuristics used in practice, nodes which are central for traffic get assigned high levels which is therefore a very natural notion of importance.

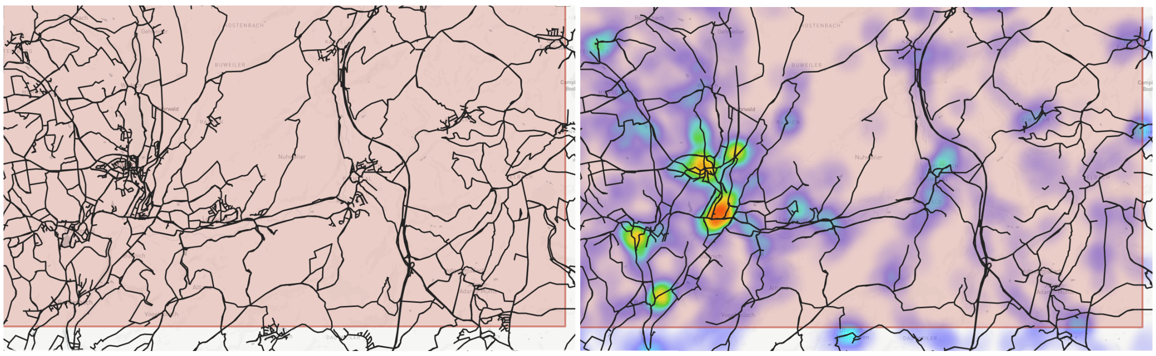


Figure 8.12.: Different level of details. Left: Maximum level of detail. Right: Lower levels are abstracted by a heat map.

8.4. Graph visualization

8.4.1. Unified Rendering And Navigation (URAN)

In the following, we briefly recap the idea of the URAN framework [FSS17] for integrated network routing and rendering.

The key insight of the URAN framework is the instrumentation of a precomputed Contraction Hierarchy as a data structure to support efficient network rendering. The basic idea is as follows: If a simplified graph view shall be rendered, a minimum rank value R is selected and then only nodes v with a rank of $r(v) \geq R$ are displayed. This automatically restricts the edge set to be rendered to the ones where both end nodes have a sufficiently high rank (most of which are typically shortcuts). However, we do not need to render a shortcut uw with $r(u), r(w) \geq R$ if v is the skip node of the shortcut and v has a rank of at least R itself. Because then the path from u to w is already represented in the rendering by the shortcuts uv and vw (or refinements thereof).

To efficiently identify the elements to be rendered, a so called REAPER data structure was designed. It augments the CH data structure with R-tree (to extract nodes within the current view rectangle) and max-heap functionality (to efficiently extract nodes of sufficiently high rank).

To get a more visually pleasing rendering, the idea of further unfolding of the shortcuts was proposed. In particular, it was suggested to store an error value with each shortcut and then to recursively replace a shortcut to be rendered by the two edges to its skip node until a sufficiently small error is achieved. Or in a simpler variant, one can unfold shortcuts until their cost drops below a certain threshold to get a visual pleasing outcome.

8.4.2. Silhouette of Graph Simplification

8.4.2.1. Silhouette Shrinking

An unpleasant side effect of the standard CH-contraction order is that nodes at the boundaries of the road network will typically be contracted quite early even if they are on major highways or interstates. While this makes sense for accelerating shortest path queries (as such nodes tend to appear in fewer optimal paths), this is quite a drawback for the URAN framework. If the threshold rank R is chosen relatively large, because a coarse representation of the network is desired, many of the peripheral parts of the road network will have been contracted and not be rendered at all. See Figure 6.1, center for an example of this phenomenon.

8.4.2.2. Silhouette Preservation

As shown in the previous section, the standard CH contraction order as described in Section 2.4.4 has the negative side-effect of not being able to preserve the overall silhouette of the network when choosing coarser representations. Intuitively, the standard CH contraction order considers non-importance of a node as criterion for contraction. But importance is not only dependent on the type of the road a node is part of, but also whether it is (geographically) central to the network or more in the periphery. For drawing purposes, though, we expect major highways and interstates be completely rendered even in very coarse representations. By simply incorporating the street type into the priority function during the contraction process, this is easy to achieve. The result is a faithful preservation of the overall silhouette even in coarse representations, see Figure 6.1, right. This modifications of the contraction process barely affects preprocessing or query time for CH .

8.5. Experimental Evaluation

We implemented all described algorithms in C++. Experiments were conducted on an AMD Ryzen Threadripper 3970X 16-Core Processor clocked at 2.061GHz with 256 GB RAM. The benchmark data was G_{ger} and $\mathcal{T}_{\text{ger,real}}$ from section 7.4.1.

8.5.1. Evaluation of Edge-Based Pathfinder Variants

We evaluated the running time of computing the Edge-Based Pathfinder on the backend for different variants of the Edge-Based Pathfinder. One was designed to perform well in local environments where the size of the bounding box rectangle was not too big, so a simple stack was best to keep track

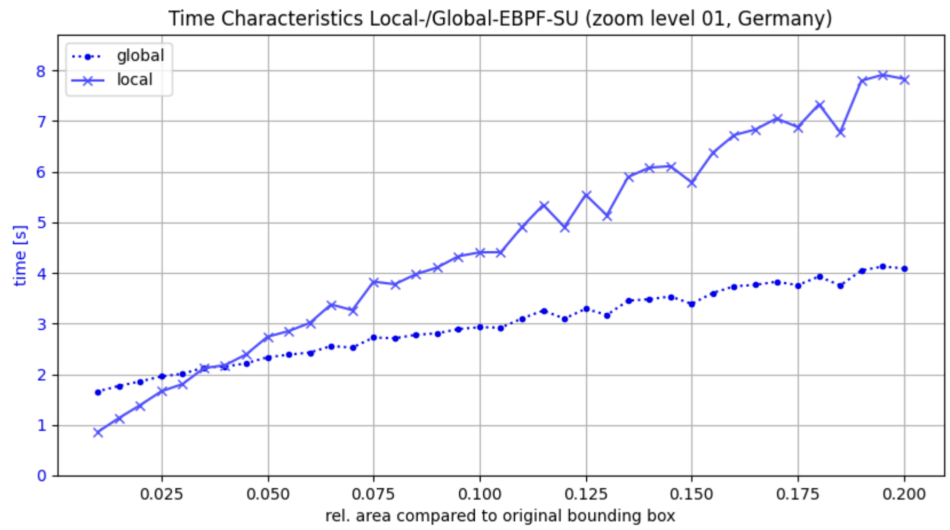


Figure 8.13.: Running times for Edge-Based Pathfinder variants.

of the marked edges. In contrast, the marking and unfolding can alternatively be done in sweeps on arrays where the nodes were ordered with respect to the levels.

The results can be seen in Figure 8.13. As expected, the stack variant was faster for small rectangles while the sweep-based variant performed better for larger ones.

8.6. Conclusions and Future Work

We described improvements to the URAN framework which allow to simplify road networks consistently while ensuring shape preservation. This allows to render road networks at any desired level-of-detail. However, to use URAN as a full replacement for pre-generated tiles at different zoom levels, other map structures such as rivers or buildings would need to be integrated as well.

While the original PATHFINDER utilized the structure of CH quite naturally for compression and retrieval of trajectory data on backend side, the presented frontend shows several approaches how the CH can be exploited elegantly for the visualization of trajectory data. We implemented these approaches which are able to visualize massive trajectory data sets on maps of continental size. If omission of details is necessary for conciseness and performance, our best approaches can give fine-grained and continuous simplifications for which certain qualities like topological consistency can be guaranteed.

Part IV.

Epilogue

Chapter 9

Summary and Future Work

We now give a summary of our contributions.

- In Chapter 4, we introduced lightheaded grids which mimic properties of real street networks. For those, we showed a surprisingly high lower bound of $\Omega(\sqrt{n})$ for the average direct search space which is in turn a lower bound for the query time of CH-Dijkstra. Very similarly, we proved that the average hub label size for lightheaded grids is $\Omega(\sqrt{n})$.

  Moreover, we developed a schema to construct instance-specific lower bounds for arbitrary graphs algorithmically. We evaluated this schema experimentally and also analyzed graph families for which it yields tight bounds.

- In Chapter 5, we looked at an existing strategy from the literature based on separators which was designed to guarantee a good contraction order for speed-up. It had even been proven that this strategy yields sublinear average node search space. However, we were able to prove that this strategy can yield linear average edge search spaces which lower bound the query time. However, we were able to show that the average edge search space can be linear on lightheaded grids which renders this seemingly reasonably strategy pointless for reducing the query time of CH-Dijkstra asymptotically. We also show that the CH framework as a speed-up technique is not necessarily flawed by providing contraction orders for lightheaded grids which provide average edge search spaces of $O(\sqrt{n}\log n)$.

- In Chapter 7, we studied storage and retrieval of large trajectory data in our Pathfinder framework. For that purpose, we use the CH framework in ways other than those it was originally designed for. First, we represent trajectories with fewer edges to achieve a high compression rate. With lean augmentations, we also use the CH data structure as an R-Tree. These modifications allow us to retrieve trajectories intersecting a give space-time cube in few microseconds per reported trajectory.

- In Chapter 8, we looked at the visualization of street network graphs in general and of trajectory data sets in particular. First we showed how the existing URAN framework can be improved to preserve silhouettes of the graph. Then we presented PATHFINDER$^{\text{VIS}}$, the extension of the Pathfinder backend with a frontend and its various possibilities to display queried trajectory sets for different use cases. For the Edge-Based Pathfinder option in particular, we used a similar unfolding technique as URAN but improved it to reduce topological inconsistencies in the simplification. We proved bounds on the unfolding technique and evaluated it in experiments.

9.1. Future Work

Finally, we hope to inspire future work where lightheaded grids could be used to lower bound other speed-up techniques. On the practical side, lightheaded grids could also be used in benchmark experiments for evaluating existing or future speed-up techniques. More on the theoretical side, it is more unclear than before whether sublinear CH query times on planar graphs can be guaranteed on average.

Regarding our implementation PATHFINDER$^{\text{VIS}}$, it could be unified with the URAN framework and possibly developed further towards a full-fledged geo-service platform.

We also hope to inspire the adaption and extension of different algorithmic frameworks which could be extended or adapted in a similarly elegant way to cover problems beyond their original scope.

Bibliography

[22a] *Bootstrap*. https://getbootstrap.com/. Last accessed 30 May 2022. 2022 (cit. on p. 82).

[22b] *Leaflet*. https://leafletjs.com/. Last accessed 30 May 2022. 2022 (cit. on p. 82).

[22c] *PATHFINDER VIS Demonstration Video*. https://fmi.uni-stuttgart.de/files/alg/research/pathfinder_vis.mp4. Last accessed 13 June 2022. 2022 (cit. on p. 83).

[22d] *Pistache*. https://github.com/pistacheio/pistache. Last accessed 30 May 2022. 2022 (cit. on p. 82).

[AA13] N. Andrienko, G. Andrienko. 'Visual Analytics of Movement: An Overview of Methods, Tools and Procedures'. In: *Information visualization* 12.1 (2013), pp. 3–24 (cit. on p. 61).

[AAC+17] G. Andrienko, N. Andrienko, W. Chen, R. Maciejewski, Y. Zhao. 'Visual Analytics of Mobility and Transportation: State of the Art and Further Research Directions'. In: *IEEE Transactions on Intelligent Transportation Systems (T-ITS)* 18.8 (2017), pp. 2232–2249 (cit. on p. 61).

[ACM+21] A. S. Andersen, A. D. Christensen, P. Michaelsen, S. Gjela, K. Torp. 'AIS Data as Trajectories and Heat Maps'. In: *Proceedings of the 29th International Conference on Advances in Geographic Information Systems (SIGSPATIAL '21)*. 2021, pp. 431–434 (cit. on p. 61).

[ADGW12] I. Abraham, D. Delling, A. V. Goldberg, R. F. Werneck. 'Hierarchical Hub Labelings for Shortest Paths'. In: *European Symposium on Algorithms (ESA)*. Vol. 7501. Lecture Notes in Computer Science. Springer, 2012, pp. 24–35 (cit. on pp. 23, 27, 31, 32).

[AFGW10] I. Abraham, A. Fiat, A. V. Goldberg, R. F. Werneck. 'Highway Dimension, Shortest Paths, and Provably Efficient Algorithms'. In: *Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM. 2010, pp. 782–793 (cit. on pp. 27, 28, 47).

[AWK+16] S. Al-Dohuki, Y. Wu, F. Kamw, J. Yang, X. Li, Y. Zhao, X. Ye, W. Chen, C. Ma, F. Wang. 'Semantictraj: A New Approach to Interacting with Massive Taxi Trajectories'. In: *IEEE transactions on visualization and computer graphics (TVCG)* 23.1 (2016), pp. 11–20 (cit. on p. 61).

[Bau21] L. Baur. 'Over-the-Web Retrieval and Visualization of Massive Trajectory Sets'. MA thesis. University of Stuttgart, Dec. 2021 (cit. on pp. 81, 87, 88).

[BCKO08] M. d. Berg, O. Cheong, M. v. Kreveld, M. Overmars. *Computational Geometry: Algorithms and Applications*. 3rd ed. Springer-Verlag TELOS, 2008 (cit. on p. 70).

[BCRW13] R. Bauer, T. Columbus, I. Rutter, D. Wagner. 'Search-Space Size in Contraction Hierarchies'. In: *Proc. 40th Int. Colloq. on Automata, Languages, and Programming (ICALP)*. Vol. 6124. LNCS. Springer. 2013, pp. 93–104 (cit. on pp. 29, 32, 49–51, 53).

[BD10] R. Bauer, D. Delling. 'SHARC: Fast and Robust Unidirectional Routing'. In: *Journal of Experimental Algorithmics (JEA)* 14 (2010), pp. 2–4 (cit. on p. 27).

[BDG+16] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, R. F. Werneck. 'Route Planning in Transportation Networks'. In: *Algorithm engineering*. Springer, 2016, pp. 19–80 (cit. on pp. 27, 49).

[BFM+07] H. Bast, S. Funke, D. Matijevic, P. Sanders, D. Schultes. 'In Transit to Constant Time Shortest-path Queries in Road Networks'. In: *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2007, pp. 46–59 (cit. on pp. 27, 32).

[BFR22] L. Baur, S. Funke, T. Rupp. 'PATHFINDERVIS'. In: *Proceedings of the 30th International Conference on Advances in Geographic Information Systems*. SIGSPATIAL '22. Seattle, Washington: Association for Computing Machinery, 2022. URL: https://doi.org/10.1145/3557915.3560990 (cit. on p. 81).

[BFRS22] L. Baur, S. Funke, T. Rupp, S. Storandt. 'Gradual Road Network Simplification with Shape and Topology Preservation'. In: *Proceedings of the 30th International Conference on Advances in Geographic Information Systems*. SIGSPATIAL '22. Seattle, Washington: Association for Computing Machinery, 2022. URL: https://doi.org/10.1145/3557915.3560987 (cit. on p. 81).

[BFS18] J. Blum, S. Funke, S. Storandt. 'Sublinear Search Spaces for Shortest Path Planning in Grid and Road Networks'. In: *Proc. 32nd AAAI Conference on Artificial Intelligence (AAAI)*. AAAI Press, 2018, pp. 6119–6126 (cit. on pp. 27, 29, 49).

[BFSS07] H. Bast, S. Funke, P. Sanders, D. Schultes. 'Fast Routing in Road Networks with Transit Nodes'. In: *Science* 316.5824 (2007), pp. 566–566 (cit. on p. 27).

[BPSW05] S. Brakatsoulas, D. Pfoser, R. Salas, C. Wenk. 'On Map-matching Vehicle Tracking Data'. In: *Proc. 31st Int. Conf. on Very Large Data Bases*. Trondheim, Norway: VLDB Endowment, 2005, pp. 853–864. URL: http://dl.acm.org/citation.cfm?id=1083592.1083691 (cit. on p. 59).

[DKS08] R. Dementiev, L. Kettner, P. Sanders. 'STXXL: Standard Template Library for XXL Data Sets'. In: *Software: Practice and Experience* 38.6 (2008), pp. 589–637 (cit. on p. 79).

[EF12] J. Eisner, S. Funke. 'Transit Nodes - Lower Bounds and Refined Construction'. In: *Proc. 14th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM / Omnipress, 2012, pp. 141–149 (cit. on pp. 32, 38).

[FFKP18] A. E. Feldmann, W. S. Fung, J. Konemann, I. Post. 'A $(1+\varepsilon)$-Embedding of Low Highway Dimension Graphs into Bounded Treewidth Graphs'. In: *SIAM Journal on Computing* 47.4 (2018), pp. 1667–1704 (cit. on p. 28).

[FKS84] M. L. Fredman, J. Komlós, E. Szemerédi. 'Storing a Sparse Table with 0 (1) Worst Case Access Time'. In: *Journal of the ACM (JACM)* 31.3 (1984), pp. 538–544 (cit. on p. 65).

[FRNS19] S. Funke, T. Rupp, A. Nusser, S. Storandt. 'PATHFINDER: Storage and Indexing of Massive Trajectory Sets'. In: *Proceedings of the 16th International Symposium on Spatial and Temporal Databases (SSTD'19)*. 2019, pp. 90–99 (cit. on p. 63).

[FS15] S. Funke, S. Storandt. 'Provable Efficiency of Contraction Hierarchies with Randomized Preprocessing'. In: *Proc. 26th Int. Symp. on Algorithms and Computation (ISAAC)*. Vol. 9472. LNCS. Springer, 2015, pp. 479–490 (cit. on pp. 22, 29, 49, 66).

[FSS17] S. Funke, N. Schnelle, S. Storandt. 'URAN: A Unified Data Structure for Rendering and Navigation'. In: *International Symposium on Web and Wireless Geographical Information Systems (W2GIS)*. Springer. 2017, pp. 66–82 (cit. on pp. 57, 58, 88, 90).

[GAD06] R. H. Güting, V. T. de Almeida, Z. Ding. 'Modeling and Querying Moving Objects in Networks'. In: *The VLDB Journal* 15.2 (June 2006), pp. 165–190. URL: https://doi.org/10.1007/s00778-005-0152-x (cit. on p. 59).

[GKW06] A. V. Goldberg, H. Kaplan, R. F. Werneck. 'Reach for A*: Efficient Point-to-Point Shortest Path Algorithms'. In: *2006 Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2006, pp. 129–143 (cit. on p. 27).

[GPPR04] C. Gavoille, D. Peleg, S. Pérennes, R. Raz. 'Distance Labeling in Graphs'. In: *Journal of Algorithms* 53.1 (2004), pp. 85–112 (cit. on p. 23).

[GSSD08] R. Geisberger, P. Sanders, D. Schultes, D. Delling. 'Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks'. In: *International Workshop on Experimental and Efficient Algorithms (WEA 2008)*. Springer, 2008, pp. 319–333 (cit. on p. 50).

[GSSV12a] R. Geisberger, P. Sanders, D. Schultes, C. Vetter. 'Exact Routing in Large Road Networks Using Contraction Hierarchies'. In: *Transportation Science* 46.3 (2012), pp. 388–404 (cit. on pp. 21, 27, 31, 32, 37).

[GSSV12b] R. Geisberger, P. Sanders, D. Schultes, C. Vetter. 'Exact Routing in Large Road Networks using Contraction Hierarchies'. In: *Transport. Science* 46.3 (2012), pp. 388–404 (cit. on pp. 15, 63).

[Gut04] R. J. Gutman. 'Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks'. In: *6th Workshop on Algorithm Engineering and Experiments ALENEX*. SIAM, 2004, pp. 100–111 (cit. on pp. 27, 28).

[Gut84] A. Guttman. 'R-trees: A Dynamic Index Structure for Spatial Searching'. In: *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*. SIGMOD '84. Boston, Massachusetts: ACM, 1984, pp. 47–57. URL: http://doi.acm.org/10.1145/602259.602266 (cit. on pp. 24, 68).

[HST+20] A. Hendawi, J. A. Stankovic, A. Taha, S. El-Sappagh, A. A. Ahmadain, M. Ali. 'Road Network Simplification for Location-based Services'. In: *GeoInformatica* 24.4 (2020), pp. 801–826 (cit. on p. 60).

[KHN+14] A. Khot, A. Hendawi, A. Nascimento, R. Katti, A. Teredesai, M. Ali. 'Road Network Compression Techniques in Spatiotemporal Embedded Systems: A Survey'. In: *Proceedings of the 5th ACM SIGSPATIAL International Workshop on GeoStreaming (IWGS 2014)*. 2014, pp. 33–36 (cit. on p. 60).

[KJT16] B. Krogh, C. S. Jensen, K. Torp. 'Efficient In-memory Indexing of Network-constrained Trajectories'. In: *Proc. 24th ACM SIGSPATIAL Int. Conf. on Advances in Geographic Information Systems (SIGSPATIAL '16)*. Burlingame, California: ACM, 2016, 17:1–17:10. URL: http://doi.acm.org/10.1145/2996913.2996972 (cit. on pp. 58, 59, 63, 70, 71, 79, 80).

[KPTT14] B. Krogh, N. Pelekis, Y. Theodoridis, K. Torp. 'Path-based Queries on Trajectory Data'. In: *Proc. 22nd ACM SIGSPATIAL Int. Conf. on Advances in Geographic Information Systems (SIGSPATIAL '14)*. ACM. 2014, pp. 341–350 (cit. on p. 59).

[KV17] A. Kosowski, L. Viennot. 'Beyond Highway Dimension: Small Distance Labels Using Tree Skeletons'. In: *Proc. 28th Ann. ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Barcelona, Spain: SIAM, 2017, pp. 1462–1478 (cit. on pp. 27, 29).

[LHK+13] Y. Li, Q. Huang, M. Kerber, L. Zhang, L. Guibas. 'Large-scale Joint Map Matching of GPS Traces'. In: *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL'13)*. 2013, pp. 214–223 (cit. on p. 59).

[LSDK18] Y. Liu, T. Safavi, A. Dighe, D. Koutra. 'Graph Summarization Methods and Applications: A Survey'. In: *ACM computing surveys (CSUR)* 51.3 (2018), pp. 1–34 (cit. on p. 60).

[LT79] R. J. Lipton, R. E. Tarjan. 'A Separator Theorem for Planar Graphs'. In: *SIAM Journal on Applied Mathematics* 36.2 (1979), pp. 177–189 (cit. on pp. 24, 32).

[LWY15] M. Lu, Z. Wang, X. Yuan. 'Trajrank: Exploring Travel Behaviour on a Route by Trajectory Ranking'. In: *2015 IEEE Pacific Visualization Symposium (PacificVis)*. IEEE. 2015, pp. 311–318 (cit. on p. 61).

[Mil12] N. Milosavljević. ‘On Optimal Preprocessing for Contraction Hierarchies’. In: *Proceedings of the 5th ACM SIGSPATIAL International Workshop on Computational Transportation Science (IWCTS)*. ACM. 2012, pp. 33–38 (cit. on pp. 28, 41).

[PR21] C. Proissl, T. Rupp. ‘On the Difference between Search Space Size and Query Complexity in Contraction Hierarchies’. In: *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA21)*. SIAM. 2021, pp. 77–87 (cit. on pp. 49, 50).

[QON07] M. A. Quddus, W. Y. Ochieng, R. B. Noland. ‘Current Map-matching Algorithms for Transport Applications: State-of-the Art and Future Research Directions’. In: *Transportation Research Part C: Emerging Technologies* 15.5 (2007), pp. 312–328. URL: http://www.sciencedirect.com/science/article/pii/S0968090X07000265 (cit. on p. 59).

[RF20] T. Rupp, S. Funke. ‘A Lower bound for the Query Phase of Contraction Hierarchies and Hub labels’. In: *15th International Computer Science Symposium in Russia (CSR 2020)*. Springer. 2020, pp. 354–366 (cit. on pp. 31, 53).

[RF21] T. Rupp, S. Funke. ‘A Lower Bound for the Query Phase of Contraction Hierarchies and Hub Labels and a Provably Optimal Instance-Based Schema’. In: *Algorithms* 14.6 (2021), p. 164 (cit. on p. 31).

[Sey17] M. P. Seybold. ‘Robust Map Matching for Heterogeneous Data via Dominance Decompositions’. In: *Proc. SIAM International Conference on Data Mining*. 2017, pp. 813–821. URL: https://doi.org/10.1137/1.9781611974973.91 (cit. on pp. 59, 72).

[SJP+07] J. Suh, S. Jung, M. Pfeifle, K. T. Vo, M. Oswald, G. Reinelt. ‘Compression of Digital Road Networks’. In: *10th International Symposium on Spatial and Temporal Databases (SSTD 2007)*. Springer. 2007, pp. 423–440 (cit. on p. 60).

[SS12] P. Sanders, D. Schultes. ‘Engineering Highway Hierarchies’. In: *ACM Journal of Experimental Algorithmics* 17.1 (2012) (cit. on p. 27).

[SSZZ] R. Song, W. Sun, B. Zheng, Y. Zheng. ‘PRESS: A Novel Framework of Trajectory Compression in Road Networks.(2014)’. In: *Proceedings of the VLDB Endowment: 40th VLDB 2014, September 1-5, Hangzhou* 7 (), pp. 661–672 (cit. on pp. 59, 63).

[SZO+11] I. Sandu Popa, K. Zeitouni, V. Oria, D. Barth, S. Vial. ‘Indexing in-network Trajectory Flows’. In: *The VLDB Journal—The International Journal on Very Large Data Bases* 20.5 (2011), pp. 643–669 (cit. on pp. 59, 63).

[Whi15] C. White. ‘Lower Bounds in the Preprocessing and Query Phases of Routing Algorithms’. In: *Proc. 23rd Ann. Europ. Symp. on Algorithms (ESA)*. Vol. 9294. LNCS. Springer, 2015, pp. 1013–1024 (cit. on p. 28).

[WLY+13] Z. Wang, M. Lu, X. Yuan, J. Zhang, H. Van De Wetering. ‘Visual Traffic Jam Analysis Based on Trajectory Data’. In: *IEEE transactions on visualization and computer graphics* 19.12 (2013), pp. 2159–2168 (cit. on p. 61).

[WZX+14] H. Wang, K. Zheng, J. Xu, B. Zheng, X. Zhou, S. Sadiq. ‘Sharkdb: An In-memory Column-oriented Trajectory Storage’. In: *Proc. 23rd ACM Int. Conf. on Information and Knowledge Management (CIKM ’14)*. ACM. 2014, pp. 1409–1418 (cit. on p. 59).

[YWY+18] X. Yang, B. Wang, K. Yang, C. Liu, B. Zheng. ‘A Novel Representation and Compression for Queries on Trajectories in Road Networks’. In: *IEEE Transactions on Knowledge and Data Engineering* 30.4 (2018), pp. 613–629 (cit. on pp. 58, 59, 63, 75).

[ZC09] W. Zeng, R. L. Church. ‘Finding Shortest Paths on Real Road Networks: The Case for A*’. In: *International journal of geographical information science* 23.4 (2009), pp. 531–543 (cit. on p. 24).

[Zhe15] Y. Zheng. ‘Trajectory Data Mining: An Overview’. In: *ACM Transactions on Intelligent Systems and Technology (TIST)* 6.3 (May 2015), 29:1–29:41. URL: http://doi.acm.org/10.1145/2743025 (cit. on pp. 58, 59).

List of Figures

List of Tables

List of Definitions