Institute of Formal Methods in Computer Science

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis Nr. 28

Unified Routing and Map Rendering

Niklas Schnelle

Course of Study:	Informatik
Examiner:	Prof. Dr. Stefan Funke
Supervisor:	Dr. Sabine Storandt
Commenced:	March 12, 2015
Completed:	September 11, 2015
CR-Classification:	I.7.2

Abstract

Two major areas worth improving in route planning and the maps accompanying it are customization and routing with limited connectivity. This thesis will tackle both while unifying the mapping and routing aspects in a single always consistent scheme. For this we created an extensible framework based on the Contraction Hierarchy scheme originally developed to speed up routing.

This scheme combined with data structures from computational geometry allows us to identify which road segments within a view are most important for routing with a given cost function. Additionally it provides us with a simple yet powerful way to refine roads for rendering at different resolutions and pixel densities. Leading to maps that automatically adapt to both the individual routing scenario and required level of detail.

To allow routing under limited connectivity the identified road segments are packaged as self-contained subgraphs. These subgraphs may then be encoded for transfer to the client where they can be combined into larger graphs that can be rendered as a map and routed on.

Contents

List of Figures

List of Tables

Listings

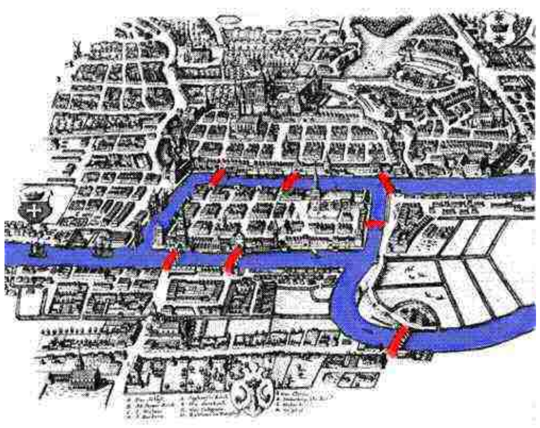
1 Introduction

Finding ones way and navigating through known and unknown territories is a challenge all moving animals have to face and which evolution has found many solutions for. These range from ant trails marked with pheromones to the navigation skills of migratory birds.

As humans we can communicate and describe an environment and a path through it to others. In fact it seems safe to assume that such descriptions of safe passages were among the first uses of languages and perhaps drawing simple maps was also one of the first practical applications of images. Today online map services such as Google Maps have billions of users and navigation systems have all but replaced traditional road maps for driving. These systems not only show us detailed maps of the entire globe but also completely automate the task of finding the best path.

As such creating maps and finding ways has a long history where a lot of progress has been made to reduce the effort required for this essential task. To understand how this progress was possible and where today's seemingly perfect online route planners still fall short, we have to take a step back and look at the beginnings of modern route planning that took maps as beautiful but complicated and underspecified works of art and opened them up for mathematical analysis.

When Leonhard Euler wondered whether there was a way to walk through the city of Königsberg using each of its seven bridges once and only once he needed a more abstract map description, one that would encompass all the information relevant to the problem while remaining abstract and clear enough for mathematical analysis. Finding such a description not only helped Euler to prove that no such path through Königsberg can exist, but also established the field of graph theory[Pao].


(a) Königsberg during Euler's time with the 7 bridges highlighted (Source [Pao])

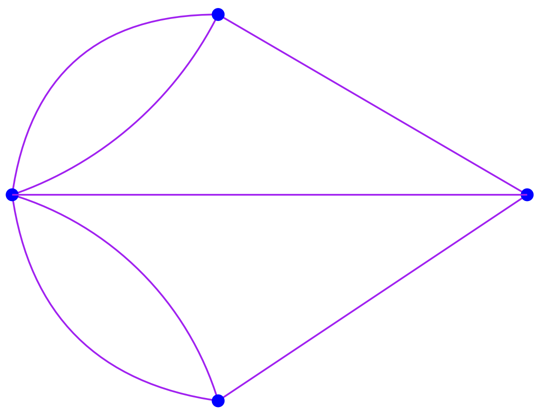

(b) Represented as an abstract graph

Figure 1.1: The Seven Bridges of Königsberg Problem which asks whether there exists a path visiting all quarters of Königsberg while using each of its seven bridges exactly once can be simplified by modeling Königsberg as a simple graph.
In this graph each vertex represents a quarter of the city and each edge represents all possible paths connecting two quarters while using a particular bridge. Note that unlike later graphs in this thesis this graph uses bidirectional edges as well as multiple edges between the same nodes.

Intuitively a graph is a collection of vertices which are arbitrary entities and their connections called edges. In the most abstract sense vertices and edges can be anything, like people and their social contacts, however in our setting of modeling street networks vertices will correspond to crossings and edges will correspond to street segments and in some cases entire paths. Similarly to Euler's original puzzle, graphs can be used to algorithmically compute shortest paths, model the flow of data or fluids in a network or analyze the structure of social networks.

In Euler's Königsberg graph (Figure 1.1) each edge represents all possible paths between two quarters that use a particular bridge. This is a significant simplification of the real road network of Königsberg, one that reduces it to the bare minimum representation that still encompasses all the information needed to answer the puzzle. As a consequence of this simplification the resulting graph shows almost no resemblance to the drawn map, in fact in this most abstract notion of a graph even the shape of the graph when drawn into a coordinate system is arbitrary as vertices do not specify their coordinates.

This disconnect between the graph used to analyze a network and the map that visualizes it is also common in modern routing systems as a more simplified graph also simplifies the analysis. This in turn allows better runtimes and less memory use enabling us to handle larger problems. It is also problematic however in that it may lead to inconsistencies between maps and the graphs they are used with and complicates the transition between solving a graph problem and displaying the solution. Additionally as we increase the scope of analysis less simplification is possible. For example while we can remove all nodes with only two adjacent edges and still compute routes we can no longer analyze whether such a route contains tight curves that would rule it out for use by an oversize vehicle.

It therefore makes sense to rethink the relation between the analysis of a network and its visualization, as shown in Figure 1.2 we can specify vertex positions in a suitable coordinate system and create increasingly fine representations of a road network where each refinement is more complex but also more detailed and closer to the classical map while still using the same well defined graph concept.

Interestingly even the coarser graph representations can be well suited as maps, for example Figure 1.2b could be used as part of a larger map, say an overview of all of Prussia and will still show the general shape of Königsberg with a minimal number of vertices and edges.

In this thesis we will show how we can come up with a continuous series of increasingly simplified versions of a road network graph algorithmically. To this end we extract the subgraph for a specific area from a source graph on demand and encode it in a way such that it can be used for rendering as well as to compute shortest paths.

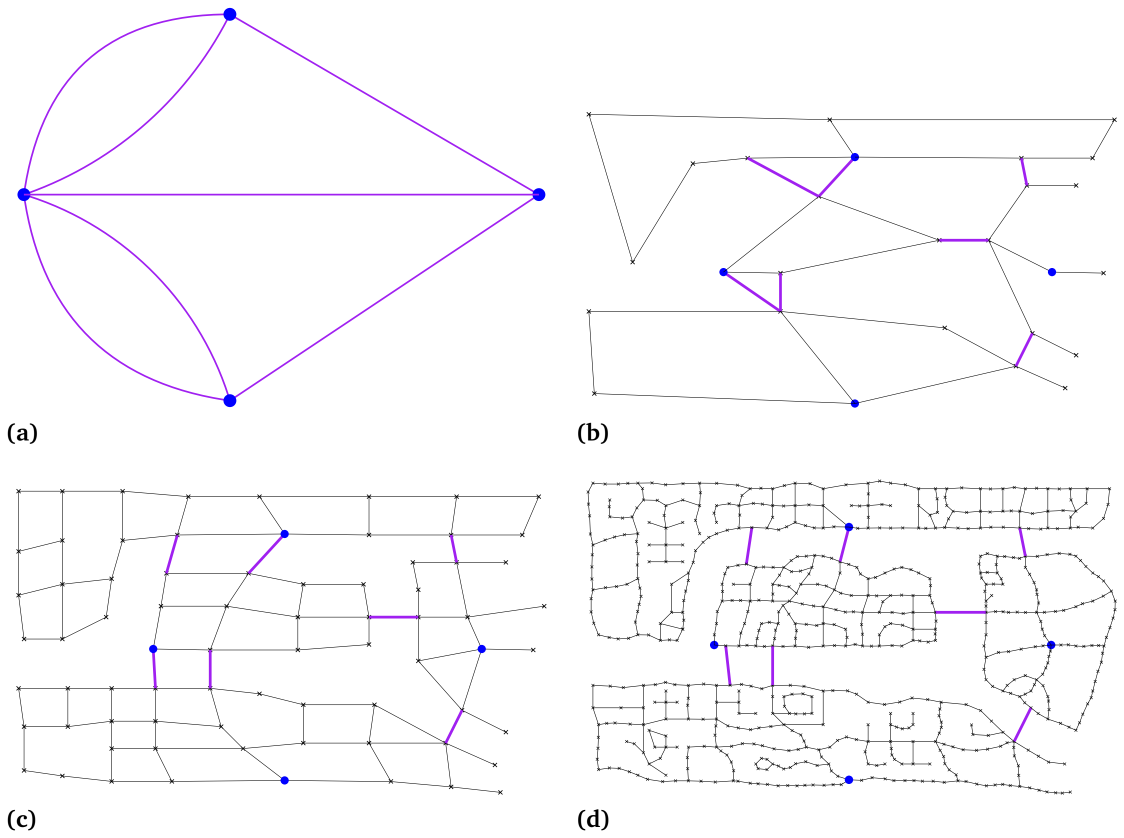


Figure 1.2: Instead of having a map of Königsberg as well as a completely separate highly abstract graph we may use graphs of different *levels of simplification*. Starting with Euler's Königsberg graph in Figure 1.2a we can built increasingly refined versions until we end up with a detailed road map in Figure 1.2d. We have highlited the edges representing the actual bridges as well as the nodes from the original graph to serve as points of reference.

1.1 Motivation

As we have outlined in the previous section there has traditionally been a disconnect between graphs and the visualization of the real world networks they represent. One consequence of this is that most theoretical research, with some notable exceptions such as [AS01], has focused on exploiting graphs purely as a tool to answer computational questions such as finding shortest paths or analyzing flows.

Especially since several break-throughs in shortest path computation (see Section 2.3), many systems both for the web[1] as well as for mobile devices[Büh13] have been developed to make use of this research in a practical way. They experiment with different architectures, server-side and client-side computation or even combinations thereof[SFS13], each innovating and improving on memory use, client-server traffic or processor time. To make their functionality accessible to users however all of them need to display a map and while they differ in the way they compute routes all of them simply use some external component for map rendering[2]. Most, if not all, of these rendering components use their own map data that in many cases comes down to pre-rendered images of some sort. So in order to do routing these systems effectively use two separate data sets one for routing and another one for rendering while conceptually and in the eyes of the user they are dealing with a single map.

This lack of integration between rendering and routing necessarily leads to a loss of efficiency, after all we could always compute the same route just by looking at the map without consulting a separate data set. Even more problematic separate data sets can always become inconsistent. Often times the data sets are even maintained by different people, updated on different schedules or even generated from different data sources. This is especially problematic since inconsistencies are hard or even impossible to track and every inconsistency can lead to routes that can be displayed but not computed or vice versa.

Even apart from missed synergistic effects current map rendering schemes are often too restrictive and static. Tile based systems are restricted to a small number of zoom levels and often lack customizability. While some routing systems[DGPW11] allow to compute routes for pedestrians, bikes, slow vehicles, farming machinery and so on, each variant would need to be pre-rendered in tile based schemes and would need its own customized set of rendering rules that again need to be kept consistent with the routing scheme.

With the scheme presented in this thesis we aim to solve these problems by unifying the rendering and routing aspects in a single system using a single data source which is both conceptually nicer, guarantees consistency and automatically customizes the map when the cost function used for routing changes, thereby closing the representative gap between the graph used for analysis and the visualization.

[1]See for example `http://www.openrouteservice.org/`

[2]See `http://wiki.openstreetmap.org/wiki/Rendering` for a list of map rendering libraries

2 Background and Related Work

This thesis will bring together the two fields of graph theory and its approaches to shortest path computation and map rendering. While these are certainly related fields it should be noted that they vary quite a bit in the maturity of their theoretical work. While a lot of research has gone into graph theory and shortest path computations most works on map rendering are practical in nature and often times little theoretical background can be found for the techniques used in practice while other times not even technical descriptions are available and some approaches to map rendering remain trade secrets.

This chapter will nevertheless try to provide background into both fields and we will add some definitions of our own to bring our discussion of the map rendering part of this thesis on a level of formalization comparable to that of the parts dealing with more classical graph theory.

2.1 Definitions

In the previous section, we discussed graphs as an intuitive while abstract concept. Now we will treat them as a formalized mathematical construct that allows us to formulate precise definitions for paths and the problem of finding the shortest path.

2.1.1 Formalized Graph

We define a directed graph as a tuple $G = (V, E)$ where V is an arbitrary set of vertices, in the scope of road networks more commonly called nodes, and $E \subseteq V \times V$ is the set of edges. In general both sets don't have to be finite though we will only be dealing with finite graphs in this work.

Since we are modeling street networks and will be interested in the length, or more generally cost, of paths we will also need to model a notion of the cost of an edge. Therefore, we define a cost function $\mathrm{cost} : E \to \mathbb{R}$ that assigns costs to edges. From here on we will also assume $\forall e \in E : \mathrm{cost}(e) \geq 0$, in most road network scenarios this assumption is inherent in the problem and it is a necessary condition for several of the algorithms including Dijkstra's algorithm. If negative edge costs are needed such as when modeling battery recuperation in electric vehicles where road segments with downwards slopes can actually add to the energy budget another possible criteria would allow negative edge costs but disallow cycles with a negative overall cost, in this case we can shift weights to make them non-negative[EFS11]

Paths

Having formalized graphs we can also formalize the concept of paths. Given a graph $G = (V, E)$ a path of length n from some source node s to a target node t is a sequence of nodes $\pi_{s,t} = v_0 v_1 \dots v_{n-1}$ where $\forall i : v_i \in V$ and $v_0 = s, v_{n-1} = t$ with $\forall v_i \in \pi_{s,t} : (v_i, v_{i+1}) \in E$. As we have defined edges as unique for each pair of nodes[1] we can alternatively see paths as a sequence of edges $\hat{\pi}_{s,t} = (s, v_1)e_1 \dots e_{n-3}(v_{n-2}, t)$. Our cost function now generalizes naturally to paths as $\mathrm{cost}(\hat{\pi}) = \sum_{e_i \in \hat{\pi}} \mathrm{cost}(e_i)$.

2.2 The Shortest Path Problem

Given a graph G and a cost function as defined in Section 2.1.1 a *shortest path* from source s to target t is a path π^* that minimizes $\mathrm{cost}(\pi_{s,t}) = \sum_{e \in \pi_{s,t}} \mathrm{cost}(e)$ among all paths from s to t. It always exists if there is a path from s to t because there are only a finite number of cycle free paths and cycles can always be removed without increasing cost as we have excluded cycles with negative costs. However it does not have to be unique as there can be several paths of the same minimal cost. We also introduce the *shortest path distance* between two nodes $d : V \times V \to \mathbb{R}^+$ as

$$d(u, v) = \begin{cases} cost(\pi^*_{u,v}) & \text{if there is a path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

2.2.1 Dijkstra's Algorithm

The classical algorithm to solve the shortest path problem in graphs with non-negative edge costs is Dijkstra's algorithm [Dij59]. It is a greedy search algorithm with the same basic structure as depth-first and breadth-first search, that is it explores the graph by expanding a set of discovered but unprocessed nodes Q by choosing and removing some node $v \in Q$ which is then considered *active* and used to discover new nodes by following its outgoing edges.
Where depth-first search uses a stack to track nodes in Q, with the active node removed from the top of the stack, and breadth-first search uses a FIFO queue, Dijkstra chooses the active node as a node v with minimal distance from the source $v = \arg\min_x \{d(s, v) | v \in Q\}$. It is important to note that once a node v is removed from Q it never reenters it and its distance value is final, we call such a node *settled*. This is because all nodes in Q have larger distance

[1]This is guaranteed because E is a set

values and therefore can not be part of a shorter path to v without negative edge costs. This also means we do not need to separately store which nodes have already been visited.

To be able to choose v the algorithm keeps track of known minimal distances while considering undiscovered nodes to have infinite distance. By also storing for each node its predecessor edge, that is the edge used to discover the node, we can finally recover the path by backtracking from the target to the source. To make the choice of v efficient Q is best organized as a priority queue data structure ordered by the minimal known distances.

When distances of nodes in Q change we would need to update them in a way that keeps the priority queue ordered, this is known as the $decreaseKey()$ operation as distances always decrease. However in practice this operation is hard to implement efficiently as we have to be able to find the position of each element in the queue given only its value. While we could use a continuously updated hash map to achieve this such an implementation destroys locality of reference. Alternatively we can always add nodes with updated minimal distances and just ignore stale entries which is the mechanism used in Listing Listing 2.1 as well as in our implementations of Dijkstra derived algorithms. The overhead of this is clearly a constant factor since we only ignore nodes which have previously been added.

```
def dijkstra(G, s, t):
   V, E, cost = G
   Q = PriorityQueue()
   d = initializeArrayWith(infinity, len(V))
   predecessor = initializeArrayWith(None, len(V))
   d[s] = 0
   Q.addWithPriority(value=s, priority=d[s])
   while not Q.empty():
      priority, active = Q.extractMin()
      # Are we there yet?
      if active == t:
         return backtrack(predecessor, s, t)

      # decreaseKey() is problematic, instead we just ignore stale entries
      if priority > d[active]:
         continue

      for (active, v) in outEdges(active, E):
         tempDist = d[active] + cost((active, v))
         if tempDist < d[v]:
            d[v] = tempDist
            predecessor[v] = (active, v)
            Q.addWithPriority(v, tempDist)
   return None

def backtrack(predecessor, s, t):
   path = [t]
   currentEdge = predecessor[t]
   while currentEdge != None:
      path.insert(0, currentEdge[0])
      currentEdge = predecessor[currentEdge[0]]

   return path
```

Listing 2.1: Dijkstra's algorithm

Time Complexity

In the worst case the target node has the largest distance from s and every edge decreases the distance to some node and thus leads to adding a node to the priority queue (or a $decreaseKey()$ operation). Since every node is active at one point in this case and every active node came from an $extractMinimum()$ operation this leads to a worst case runtime of $O(|E|T_{\text{add}} + |V|T_{\text{min}})$ where T_{add} and T_{min} are the time complexities of the $add()$ and $extractMin()$ operations of the priority queue. With standard binary heaps this leads to a runtime of $O((|V| + |E|)log|V|)$ while Fibonacci Heaps decrease this to $O(|E| + |V|log|V|)$.

2.3 Speedup Techniques

While Dijkstra's algorithm correctly solves the shortest path problem it takes on the order of seconds to compute a shortest path on a country wide road network. This may be prohibitively long especially for server based routing solutions such as (Microsoft, Yahoo!, Google) Maps where backend servers have to handle thousands of routing queries per second with users expecting immediate answers. The main reason running times increase so much is that Dijkstra's algorithm blindly searches the entire graph until it reaches the target which leads to a circular search space that easily encompasses large parts of the graph before reaching a target.
This problem is clearly not solved by better priority queues and for general graphs there seems to be little room for improvement after all before reaching the target there could always be a short edge directly to the target anywhere in the graph. However intuitively for many real world graphs there are no "wormhole" edges that would take one straight to the target from the other side of the network. Formally speaking additional constraints hold such as the edges being bounded by euclidean distance, intuitively on road networks the air-line distance is a lower bound for path lengths.
Additionally road networks tend to show significant hierarchical structure. For example most long shortest paths will use highways for most of the way. As we will see all these properties can be exploited to speed-up shortest path computations on real world road networks.
One simple technique to shrink the search space is the A^* algorithm[HNR68] which uses a heuristic to steer the search space of Dijkstra's algorithm towards the goal. In its most simple application one can use euclidean air-line distances as its heuristic, because they always underestimate path lengths they are an admissible heuristic and A^* remains correct. Using precomputed shortest path distances to landmark points for its heuristic [GH05] builds on A^* and achieves an order of magnitude better performance. In [Gut04] the reach metric is introduced, it tries to capture the importance of nodes in the graph with respect to shortest paths by looking at "the length of shortest paths on which it lies" and allows pruning of the search space leading to similar speedups. Both landmarks based A^* and reach based pruning can be combined as described in [GKW06]. Somewhat similarly the transit nodes algorithm[BFM09] tries to find important nodes and uses precomputed distances among them to speed up queries. Instead of searching for global landmarks however it looks for nodes that are important to non-local shortest paths leaving or entering a specific area.

Arguably one of the most important break-throughs in recent research on shortest path computation, however comes from the Contraction Hierarchies scheme originally proposed by [GSSD08]. As it enables a whole range of algorithms[DGWN10, SFS13, Büh13] and forms the basis of our proposed scheme we will examine it in more detail in the following section.

2.4 Contraction Hierarchies

Like most of the previously discussed speedup techniques Contraction Hierarchies (CH) is a prepossessing scheme that yields additional information and constraints that allows us to massively increase the efficiency of subsequent shortest path queries. Compared with the previous schemes however Contraction Hierarchies and especially its newer variants are quite simple and it has turned out that the information created during its preprocessing step not only allows us to speedup shortest path point-to-point queries but also enables a whole range of algorithms including one-to-many and even efficient all-pairs [DGWN10] computations.

2.4.1 Overview

As the name implies Contraction Hierarchies creates during preprocessing a hierarchy on the graph that is strongly correlated with the natural hierarchy in the original graph such as the differing importance of individual roads and their generalized types in road networks, the exact nature of which is subject to ongoing research. In [AFGW10] Abraham et al. try to formalize and model this relation with the introduction of the *highway dimensions* metric which they also use to examine the impact on search space size. The “Contraction” part of the name on the other hand hints at the basic operation of preprocessing which is based on the idea that a graph can be simplified by removing a node v while keeping all shortest paths, except of course those starting or ending at v, intact by introducing *shortcuts* between neighbouring nodes if and only if they are connected by a shortest path via v. A shortcut is an edge which skips over a removed node v by replacing subpaths of the form uvw with a single edge (u, w). This is called a node contraction and we will discuss it in more detail in Section 2.4.2

As its result Contraction Hierarchies’ preprocessing of a graph $G = (V, E)$ yields an overlay graph $G^+ = (V, E^+)$ with an augmented edge set $E^+ \supseteq E$ and an assignment of levels $\Phi : V \to \mathbb{N}_0$ to the nodes. This assignment imposes a partial ordering on the nodes with the additional property, that for every shortest s, t path $\pi_{s,t}$ in G there exists a representation $\pi'_{s,t} = v_0 v_1 \dots v_k$ in G^+ such that there exists an index $l \in [0, k]$ with $\forall i \in [0, l] : \Phi(v_i) \leq \Phi(v_{i+1})$ and $\forall i \in [l + 1, k] : \Phi(v_i) \geq \Phi(v_{i+1})$. More informally speaking the shortest path in G^+ breaks down into two subpaths an upwards part of increasing levels and disjunct downwards part with decreasing levels. For such a path we can recover the original shortest path π by unpacking shortcuts recursively until only edges from the original edge set E are used.

2.4.2 Preprocessing

We present a variation on the original Contraction Hierarchies preprocessing, instead of a total ordering on nodes defining a contraction order, we work in phases contracting a maximal independent set of nodes at a time assigning them them the phase number as level. This gets rid of many special cases and enables several additional applications including [DGWN10].

Contracting a Node v

Contracting a node v means removing it from the graph while leaving all shortest paths except those originating or ending at v intact, that is the same distance. We might want to allow keeping only one alternative for shortest paths of equal length.

It is clear that for paths that do not use v there is nothing to do, we are thus only interested in paths that use v. All such paths have edges (u, v) and (v, w) where u, w are neighbors of v, it is therefore sufficient to look at all pairs of incoming and outgoing edges of v and their respective neighbor nodes. Let (u, v) and (v, w) be such a pair, if there is a path from u to w that is at least as cheap as $\text{cost}((u, v)) + \text{cost}((v, w))$ we know that v is not on a singular shortest path using the subpath uvw and so no shortcut is needed, we call such a path a *witness* because it is witness to the existence of a shorter path. See Figure 2.1. We can easily check for the existence of a witness path by running a local Dijkstra search from u searching for w.

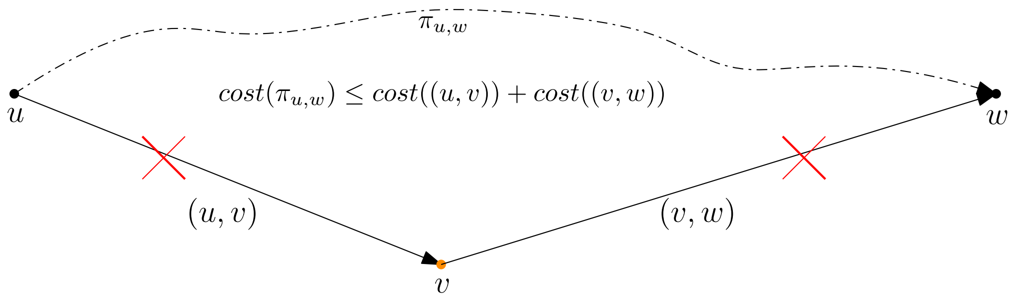


Figure 2.1: The node v with a pair of adjacent edges (u, v) and (v, w) is not on the shortest u, w path as proven by the witness path $\pi_{u,w}$. Therefore we can remove (u, v) and (v, w) without adding a shortcut and still leave all shortest paths intact.

If on the other hand no such path is found uvw is part of at least the shortest path uvw and we need to add a shortcut edge (u, w) with $\text{cost}((u, w)) = \text{cost}((u, v)) + \text{cost}((v, w))$ to preserve shortest path costs. Additionally we save the mapping $unpack((u, w)) \mapsto ((u, v), (v, w))$ which allows us to unpack shortcuts into their constituents later. This is illustrated in Figure 2.2. The whole contraction of a node v now yields the set of created shortcuts $E^{\text{shortcuts}}(v)$.

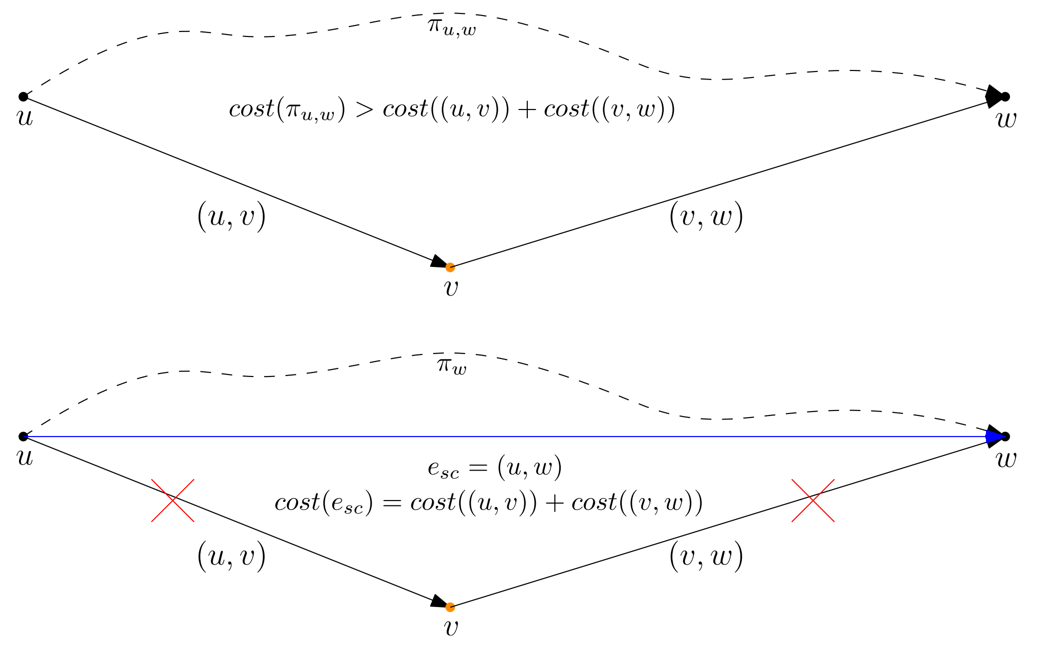

$\pi_{u,w}$
u
$cost(\pi_{u,w}) > cost((u,v)) + cost((v,w))$
w
(u,v)
(v,w)
v
π_w
u
$e_{sc} = (u,w)$
$cost(e_{sc}) = cost((u,v)) + cost((v,w))$
w
(u,v)
(v,w)
v

Figure 2.2: Node v with adjacent edges (u,v) and (v,w) lies on shortest path uvw. Therefore removing (u,v) and (v,w) would result in losing all shortest paths that have the subpath uvw (at least uvw itself) and we need to add a shortcut (u,w) (blue) with the same cost as uvw to keep these paths intact. This cost is exactly $cost((u,v)) + cost((v,w))$.

Contracting the Graph

Formally we start with $i = 0$ and a graph $G_0 = (V_0 = V, E_0 = E)$ and compute for each phase a maximal independent set $I_i \subset V$ where $\forall(u, w) \in I^2 : (u, w) \notin E_i$ by greedily choosing nodes in order of increasing importance as measured by their edge difference, that is the difference between the number of shortcuts added and their degree.

We then *contract* all nodes $v \in I$ adding the necessary shortcuts. Using an independent set ensures that node contractions in a single phase do not interact with each other while the ordering helps to reduce the number of added shortcuts.

This yields a new graph $G_{i+1} = (V_{i+1} = V_i \setminus I_i, E_{i+1})$ where $E_{i+1} = E_i \setminus E^{\text{removed}} \cup E^{\text{shortcuts}}$ where E^{removed} are the edges removed during the contractions that is all edges adjacent to the contracted nodes and $E^{\text{shortcuts}} = \bigcup_{v \in I} E^{\text{shortcuts}}(v)$ are the shortcuts added. After each fewer and fewer nodes remain though especially in later phases their degree increases. We can break this process after each phase though for best performance we keep contracting until only a single node remains.

2.4.3 Correctness

We will now proof by contradiction that this preprocessing leads to an assignment of levels, where for every shortest path in G there exists a representation in G^+ that consists of an upwards- and then a downwards subpath in G^+ as described in Section 2.4.1. Let us first define what we mean by a representation of a shortest path. Given a shortest path $\pi_{s,t}$ in G a *representation* of $\pi_{s,t}$ is a shortest path in G^+ that results in the path $\pi_{s,t}$ when all shortcuts are fully unpacked.

We also note that the upwards- and downwards subpaths can be empty. The claim is thus only violated if for some s, t a shortest path $\pi_{s,t}$ in G exists that has only representations that do not break up into an upwards- and then a downwards subpath. It follows that these representations contain some *violating* node v_l that is of lower level than both its predecessor and successor, formally $\Phi(v_{l-1}) > \Phi(v_l) < \Phi(v_{l+1})$. In Figure 2.3 the orange representation of the shortest path $sv_1v_2 \dots v_{12}t$ has two such violating nodes v_3 and v_9.

If such a node exists it would have been contracted before its predecessor and successor in the path as $\Phi(v_l)$ is simply the number of the phase that v_l was contracted in. However this also means that as v_l lies on a shortest path from its predecessor to its successor, as a representation is a shortest path in G^+, a shortcut between them would have been added connecting them without going down to v_l resulting in another representation that does not violate the claim at v_l. In the example this means that after a correct preprocessing the dashed edges (v_2, v_5) and (v_8, v_{10}) have to exist.

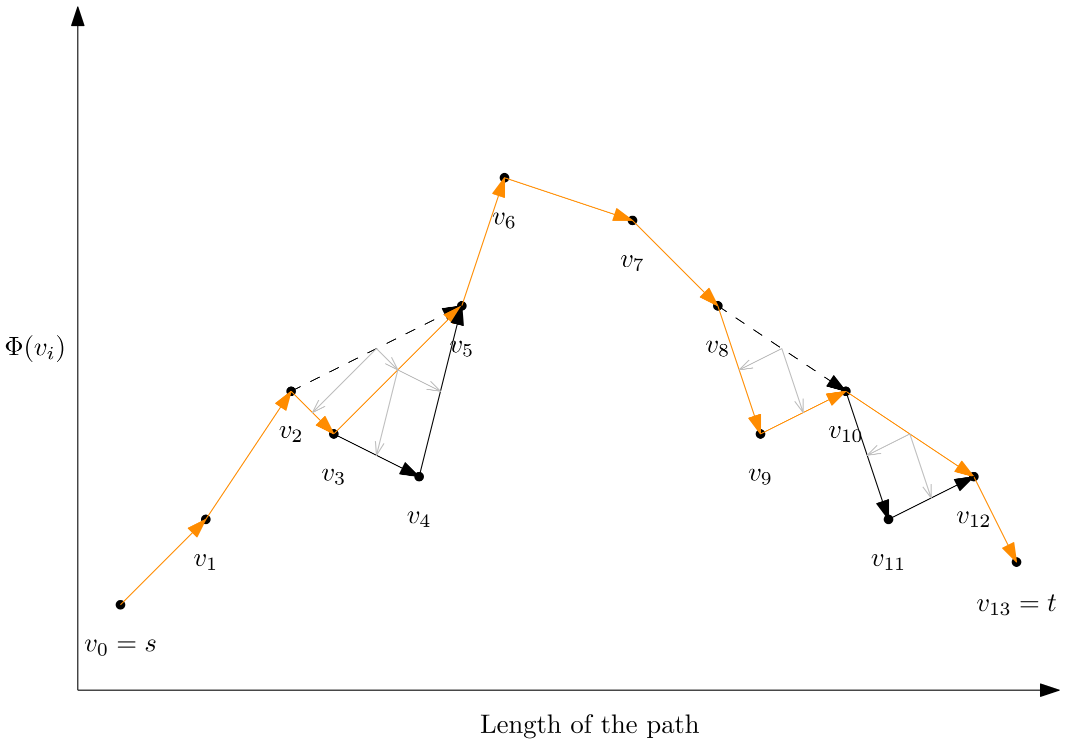


Figure 2.3: Shown are a shortest path $\pi_{s,t} = sv_1v_2 \ldots v_{12}t$ as well as one of its representations (in orange) which is a shortest path in G^+ that results in $\pi_{s,t}$ when all shortcuts are unpacked. The representation violates the upwards- then downwards subpath property at nodes v_3 and v_9. This is because both lie on the representation yet their predecessors v_2 respectively v_8 and their sucessors v_5 and v_{10} are of higher level. Nodes v_4 and v_{11} on the other hand do not lie on the representation in question. There has to exist another non-violating representation however as the contraction of v_3 and v_9 would have introduced the shortcuts (v_2, v_5) and (v_8, v_{10}) (dashed arrows) as the representation is a shortest path and therefore $v_2v_3v_5$ and $v_8v_9v_{10}$ are also shortest paths.

2.5 Applying Contraction Hierarchies

We will now look into applications of Contraction Hierarchies and how to exploit the the upwards- then downwards subpath constraint from Section 2.4.1. In our descriptions in this section as well as in the rest of the thesis we will use the following formal definitions of subgraphs of G^+

Upwards graph: $G^\uparrow = (V, E^\uparrow), E^\uparrow = \{(v,w) \in E^+ | \Phi(v) \leq \Phi(w)\}$

Downwards graph: $G^\downarrow = (V, E^\downarrow), E^\downarrow = \{(v,w) \in E^+ | \Phi(v) \geq \Phi(w)\}$

Upwards graph of v:

$$G^\uparrow(v) = (V^\uparrow(v), E^\uparrow(v)),$$
$$V^\uparrow(v) = \{v\} \bigcup_{u \in V^\uparrow(v)} \{w | (u,w) \in E^\uparrow\}$$
$$E^\uparrow(v) = \{(u,w) \in E^\uparrow | u, w \in V^\uparrow(v)\}$$

Downwards graph of v:

$$G^\downarrow(v) = (V^\downarrow(v), E^\downarrow(v)),$$
$$V^\downarrow(v) = \{v\} \bigcup_{u \in V^\downarrow(v)} \{w | (w,u) \in E^\downarrow\}$$
$$E^\downarrow(v) = \{(w,u) \in E^\downarrow | u, w \in V^\downarrow(v)\}$$

Intuitively upwards graphs and downwards graphs are the search spaces in G^+ after applying the upwards and then downwards constraint either on the entire graph or as seen from start or target node. Upwards graphs and downwards graphs of a node v can be computed by breadth-first searches from v that only consider upwards respectively downwards edges.

With these definitions in hand we can easily formulate how to modify a whole range of algorithms to take advantage of the overlay graph by restricting their search spaces. Let us first note however that because G^+ is an augmentation we can still use normal Dijkstra on it by ignoring all shortcuts.

2.5.1 Marking Dijkstra

One simple yet versatile way to speedup Dijkstra's algorithm without the complexity of a bidirectional search is the marking based Dijkstra approach. Let s be the source node and let $T = \{t_0, t_1, \dots, t_l | t_i \in V\}$ be a set of target nodes. We restrict Dijkstra's algorithm to search in $G^\uparrow(s) \bigcup_{t \in T} G^\downarrow(t)$ by first running a breadth-first search from every t on $G^\downarrow(t)$ and marking its edges. Then we run a Dijkstra search from s that only looks at edges from $E^\uparrow$ as well as marked edges. This results in a quite efficient algorithm for one-to-many queries.

2.5.2 Bidirectional Dijkstra

There is a slightly more complicated variant of Dijkstra's algorithm that is constrained to one-to-one queries but shows an additional increase in efficiency and search space reduction. For a query from start node s to target node t instead of restricting a forward search to $E^{\uparrow}(s) \cup E^{\downarrow}(t)$ we do bidirectional search with a separate forward search restricted to $E^{\uparrow}(s)$ and a backward search restricted to $E^{\downarrow}(t)$ that follows edges backwards from target to source. While these searches are separate they share a single priority queue where entries are marked as belonging to a particular search direction. While the forward search sets distance values $d(s,v)$ from the start node, the backward search stores distances to the target $d(v,t)$. We now track the node with the best sum of forward and backwards distances $d(s,v) + d(v,t)$ at the point of settling v in both searches which gives us a best cost candidate. Since some edges are no longer explored temporary forward and backward distances are only upper bounds and don't have to be correct with respect to the original graph. Only when $d(s,v)$ or $d(v,t)$ becomes larger than the current best cost candidate do we know that this candidates distance is the final shortest path distance and that the candidate is the top node separating the upwards and downwards subpaths.

During the search an active node with larger than optimal distance will only lead to further exploring nodes with sub optimal distances. To reduce such futile explorations we can apply an optimization that looks back at incoming (outgoing) edges and *stalls* the current node if an edge would lead to a smaller distance. Stalling here means that the node is not explored further. In practice this simple one step stalling leads to an order of magnitude in search space reduction.

2.5.3 The Core Graph

In addition to the previously defined subgraphs we now introduce the *core graph* of level l, that is defined as the portion of G^+ that only includes nodes of level at least l as well as the edges between them.

Intuitively the core graph represents a subgraph of the most important nodes and edges, however let it be noted that the exact meaning of importance in this context is not entirely understood. As we will see later, for road networks it seems to correlate quite well with a human understanding of *importance of a road segment*.

We also introduce *restricted* versions of *upwards graphs* and *downwards graphs* that only span G^+ up to the core graph which can easily computed by terminating the breadth-first searches used to explore normal upwards graphs and downwards graphs once they reach the core graph. For some applications we also include the edges entering the core into the restricted upwards graphs and downwards graphs. These subgraphs will later play a vital role in the core algorithms of this thesis.

$$
\begin{aligned}
&G^{core}(l) = (V^{core}(l), E^{core}(l))\\
&V^{core}(l) = \{v \in V | \Phi(v) \geq l\},\\
&E^{core}(l) = \{(u,v) \in E^{+} | u, v \in V^{core}(l)\}\\
&G^{\uparrow r_l}(v) = (V^{\uparrow r_l}(v), E^{\uparrow r_l}(v))\\
&V^{\uparrow r_l}(v) = V^{\uparrow}(v) \setminus V^{core}(l)\\
&E^{\uparrow r_l}(v) = \{(u,v) \in E^{\uparrow}(v) | u \in V^{\uparrow r_l}(v) \land v \in V^{core}(l) \cup V^{\uparrow r_l}\}\\
&G^{\downarrow r_l}(v) = (V^{\downarrow r_l}(v), E^{\downarrow r_l}(v))\\
&V^{\downarrow r_l}(v) = V^{\downarrow}(v) \setminus V^{core}(l)\\
&E^{\downarrow r_l}(v) = \{(u,v) \in E^{\downarrow}(v) | v \in V^{core}(l) \cup V^{\downarrow r_l} \land u \in V^{\downarrow r_l}(v)\}
\end{aligned}
$$

Size of the Core Graph

We will only briefly discuss the size of the core graph here as we will look into it in more detail later when discussing the core graph variant used for unified rendering and routing. In [SFS13] we showed that for our German road network graph of $16{,}271{,}859$ nodes and $62{,}062{,}727$ edges a core graph of level $l = 40$ has 1092 nodes and $39{,}019$ edges which is less than 0.007% of the nodes and 0.063% of the edges.

2.5.4 Client-Side Routing and the DORC Algorithm

The idea behind using core graphs and restricted upwards and downwards graphs is that for a query with source nodes $S = \{s_0, s_1 \ldots s_k\}$ and target nodes $T = \{t_0, t_1 \ldots t_r\}$ the search space is restricted to [2]

$$G(core, S, T) = G^{core}(l) \cup \bigcup_{s \in S} G^{\uparrow r_l}(s) \cup \bigcup_{t \in T} G^{\downarrow r_l}(t)$$

where $G^{core}(l)$ is independent of the query specific source and target nodes. So when answering a large number of queries we only need to change the restricted upwards and downwards graphs.

[2]We use union over graphs for brevity here, this can be understood as $G_a \cup G_b := (V_a \cup V_b, E_a \cup E_b)$ where G_a and G_b are assumed to be subgraphs of the same graph.

Client-Side Routing

In pure client side routing as described in [Büh13] the static nature of the core graph is exploited to reduce disk accesses. Instead of holding all graph data in RAM only the core graph is held there across queries while the rest of the graph is stored on disk in a format that allows for efficient lookup and extraction of upwards and downwards graphs for a node. Additionally this lookup can efficiently be done by geographical coordinates only.

The DORC Algorithm

In the Distributed Online Route Computation algorithm from [SFS13] the core graph is the only graph data persistently stored on the client and the restricted upwards and downwards graphs of a queries source and target nodes are computed on demand on a remote server and then transferred to the client over the network. The client application then overlays them onto the core graph with the upwards and downwards graphs and runs a simple Dijkstra algorithm on this combined graph. The advantage now comes from the fact that these subgraphs are much simpler and faster to compute than full shortest path queries while still using acceptable network bandwidth. One problem with this approach that we will solve in the algorithms of this thesis is that cutting subgraphs from some original graph while retaining the ids to allow relating the two often lead to non-contiguous node ids. These in turn complicate or even rule out many efficient graph data structures.

2.6 The Map Rendering Problem

While the shortest path problem can easily be described on formalized abstract graphs, map rendering is less clearly defined. In general we understand map rendering as the process of creating a graphical representation, the *map*, of a physical or even fictional environment for the purpose of navigation or geographical visualization. While maps may be of artistic value they are differentiated from other graphical depictions of the environment by their more or less practical use. In almost all cases maps also employ significant simplification and abstraction though the degree of simplifications varies widely, as an extreme example autonomous vehicles may employ very detailed 3D scans of their environment for navigation which thus may also be considered to be maps.

While this thesis only deals with rendering roads we anticipate its techniques to be complemented with similar schemes for rendering regions such as counties, bodies of water and land use or even individual buildings as well as labeling and will therefore describe map rendering in this context.

2.6.1 Modeling Road Networks

Until now we have treated graphs as completely abstract constructs that even when they encoded the connectedness structure and costs of a road network lived in a purely mathematical context with no way to interact with it. When developing applications that interact with users we need to also treat graphs as geometrical objects that correspond more closely to a classical map. For this a graph needs to be embedded into some coordinate system. Many geospatial systems work with coordinates on an ellipsoid like the WGS-84[Def00] used in GPS, others project these coordinates onto a plane, for example using the Mercator projection, we may use either or any other coordinate system that makes sense for the interface used to interact with the map. Such interaction ranges from setting sources and targets for a shortest path query to displaying a computed path or simply visually exploring the road network. Since nodes are the points that shortest paths start and end at it makes sense to give them coordinates so that they can be shown and selected in the interface.

For the purpose of simple and consistent notation we will refer to the coordinates of nodes as $(x, y) \in \mathbb{R}^2$ associated to nodes via a mapping $coords : V \to \mathbb{R}^2$. Though these names suggest euclidean coordinates in a plane, as would result when projecting geographic coordinates via some map projection, our concepts transfer to using latitude and longitude coordinates directly though some accounting for edge cases at the poles and the meridians will be necessary, this is especially useful when displaying maps in 3D.

We also need to think about the nature of edges, one could use one edge for every intersection free road segment so that all nodes have either degree > 2 or mark the end of a road. We would then separately store information on how an edge should be drawn, however this approach is not very flexible as our basic graph data structure is now only really useful for routing and we need to bridge a gap to the information needed to draw every time we want to display a computed path or show the graph data structure as a rendered map.

If we look at the data set of OpenStreetMap[3], the Wikipedia of mapping, we find that it keeps all information organized as geospatial relations which are used for everything from polygons for the outlines and to display streets. So when preprocessing this data the straight forward transformation leads to a graph where each edge in a road relation becomes an edge in the graph.

Each edge now corresponds to a straight line single direction road segment, in curves the segments are short enough so that the curve appears as round at normal mapping scales and vertices represent turning points instead of of only entire intersections. Since edges are always unidirectional this also means that even road segments for small roads that can be used in both directions are modeled as two edges, though these usually share the source and target nodes (i.e. (u, v) and (v, u) model a bidirectional road segment connecting u and v). On

[3] http://openstreetmap.org

the one hand this leaves a lot of degree 2 nodes in the data that are inefficient for routing purposes, however we may trust our speed up techniques to make their effect on performance negligible and in return we get trivial transitions from paths in the graph to their geometrical representation. For example displaying a shortest path is as simple as drawing a line for each of its edges in the order they appear in the path.

2.6.2 View Rectangles and Map Simplification

As we have seen in Section 2.6.1 our graph model defines a global coordinate system, when rendering however we often only want to see a small part of the entire road network. We call this visible part the view rectangle $R = (x, y, width, height)$ and it restricts what is displayed to those elements that intersect the axis aligned rectangle $[x, x + width] \times [y, y + height]$ where (x, y) marks either the upper left or lower left corner depending on whether we treat y-coordinates as increasing upwards as it is common in classical euclidean coordinate system or as increasing downwards as is common in computer graphics. The elements to be displayed here include both the nodes and edge of the road network as well as any other map entities such as outlines or labels.

Additionally if we looked at the entire graph at once without any simplification the level of detail would be overwhelming. Thus it makes sense to also consider a *simplification level S* that describes the degree of simplification relative to displaying everything as accurate as the available data allows. In general this simplification happens according to some *notion of importance*. In classical road maps this often means that overview maps that cover a larger viewing rectangle will display only highways and speedways while maps of small viewing rectangles such as covering a city will display even the smallest roads. Similarly other objects like bodies of water or building outlines are displayed only at certain zoom levels.

The notion of importance of edges as well as other displayed entities is thus based on a constant and often small set of categories. Simplification levels are then simply sets of displayed categories of entities which often need to be hand tuned. This is particularly problematic when maps are customized for different routing purposes such as bike maps or maps for oversize transports. Especially in the latter scenario selecting the correct roads to display can be tedious or even impossible with a fixed set of categories as we do not only want to exclude narrow passages but also the roads that are only used on paths using such passages.

In addition to not displaying some entities at all which is also known as pruning both roads as well as outlines may be geometrically simplified for example using the Douglas/Peucker *line simplification* algorithm [DP73]. In this algorithm the goal is to minimize the number of nodes used to display a particular polyline by pruning degree two nodes subject to certain quality constraints such as shape preservation and geometrical faithfulness while also considering view-dependent properties such as the lines displayed size and the pixel density of the display itself. Additionally there exist variants which also respect topological constraints so that for example no town will end up on the wrong side of a border[BKS98]. It should be noted, that even the original variant by Douglas/Peucker is NP-hard to approximate better than within a factor of $n^{\frac{1}{5-\delta}}$ if no self-intersections are to be introduced[EM01]. There also exist modern variants that utilize graphics hardware to assist map simplification[MKKV01] though these may impose constraints such as non-intersection that require signficant precomputation if road network data is to be simplified and even then, pure line simplification and thus constraining ourselves to pruning of degree two nodes is too restrictive if we want to do map simplification in a generalized framework that does away with fixed categories.

For interactive map displays the view rectangle will be transformed dynamically with *panning* and *zooming* operations where panning describes a simple lateral motion of the view rectangle $R = (x, y, width, height) \to R' = (x + \Delta x, y + \Delta y, width, height)$ while zooming changes the size of the viewing rectangle with an optional translational component to direct the zoom towards a target $R' = (x + \Delta x, y + \Delta y, width * z, height * z)$ (see also Section 3.5.1).

As these operations are most often applied in longer sequences progressively exploring a map some consistency between consecutive steps is needed to prevent distracting and confusing artifacts. In particular we have identified the following consistency properties.

Translation Property: If an edge e is displayed for the current view rectangle R and intersects the panned view rectangle R' it should be displayed for this as well, assuming the simplification level S stays constant. This prevents edges from disappearing when panning.

Zoom Property: If an edge e is displayed for the current view rectangle R with simplification level S it should be displayed for all subrectangles $R' \subseteq R$ containing e for all simplification levels $S' \leq S$. That is all zoomed in rectangles with less or equal simplification level, instead of displaying e directly we may also display a refined variant of it fitting the new simplification level S'.

Compression Property: For all $R' \supseteq R$ we should be able to find an $S' \geq S$ such that there are no more elements reported for R' than for R while maitaining a reasonable result for the potentially much larger R'. In particular the resulst should only be empty if there are no entities in R. This means the simplification actually decreases the complexity of the structure to render.

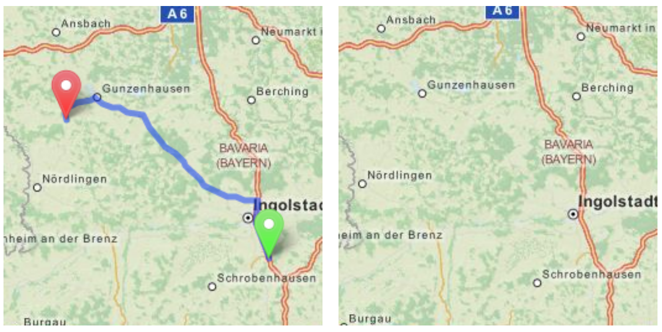
Importance Property: If an enitity with importance p in R is rendered all entitities of importance $p' \geq p$ in R should be rendered.

Connectivity Property: If for any two points s, t rendered for a given R, S the shortest path $\pi_{s,t}$ between them in G is inside R, they should be connected through a path $\pi'_{s,t}$ with the same length as $\pi_{s,t}$ in all view rectangles R' containing s and t

2.6.3 Tiling and Zoom Levels

To realize map rendering in an online setting most current map services utilize a tile based system where the entire map is rendered in a Mercator projection at a preset, typically small, number of simplification levels that directly correspond to a scaling of the map (also referred to as zoom factor), together they form the combined *zoom level*. In this terminology higher zoom levels correspond to higher levels of detail which is an important difference to the previously defined simplification levels where higher levels of simplification correspond to lower levels of detail.

Each zoom level is precomputed at least conceptually as a potentially large image split into square tiles of equal size. Later when users view the map the tiles needed to display the part of the map given by the view rectangle can easily be determined by the views coordinates, its size as well as the requested scaling of the map. After determining which tiles are needed they they can then be retrieved from backend storage. As the tiles are precomputed as rasterized images the zooming operations are non contiguous and the rendering can not account for different pixel densities or bandwidth requirements. This can be somewhat alleviated by interpolation but this still often leaves visible artifacts and is most often only used for transitions.



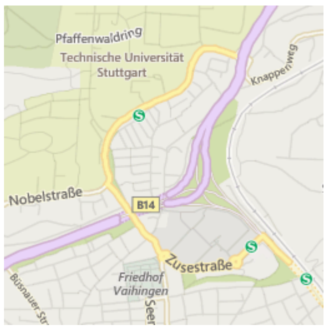
(a) A path in OpenStreetMaps **(b)** Same area without the path

Figure 2.4: In this screenshot taken from OSRM (OpenStreetMap Routing) we can see an inconsistency between the map and the computed routes. A path is drawn yet the roads it uses are not visible at that zoom level.

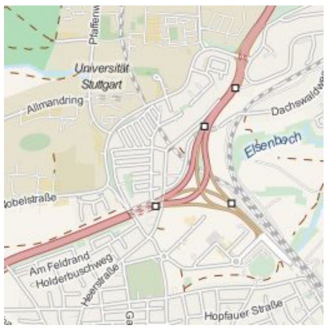
Looking at the properties from Section 2.6.2 we can see that the importance, translation and zoom properties are trivial to fulfill if tiles of each level display the same edge set and the edge sets of higher zoom levels are supersets of the edge sets at lower zoom levels. The compression property is also easy to fulfill as tiles for lower levels of detail can just cover larger areas with the same resolution. The connectivity property on the other hand is often not fulfilled as the edge sets are fixed and do not necessarily correspond to the edges needed for shortest paths between visible nodes. This leads to unexpected behavior like shortest paths being drawn in the dark, that is using streets that are not visible in the zoom level used to display them. This is illustrated in Figure 2.4

The Bing, Here and OSM Tiling Scheme

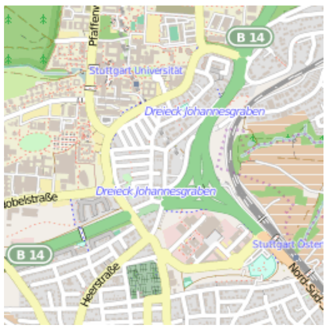
Figure 2.5 shows tiles for a map area to the south east of the University of Stuttgart in several popular map services. Interestingly the current version of Google Maps does not use a tile based approach though details on this are scarce. As we can see all obtained tiles are aligned at the top left corner, in fact apart from Nokia Here which uses larger tiles they all display the same area.



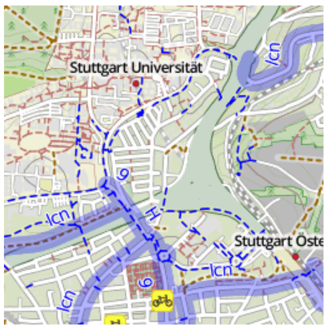
(a) Bing Maps



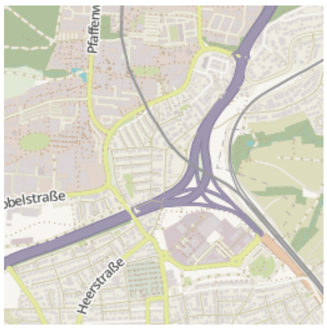
(b) OSM MapQuest



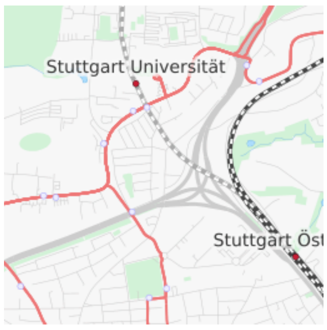
(c) OSM Standard



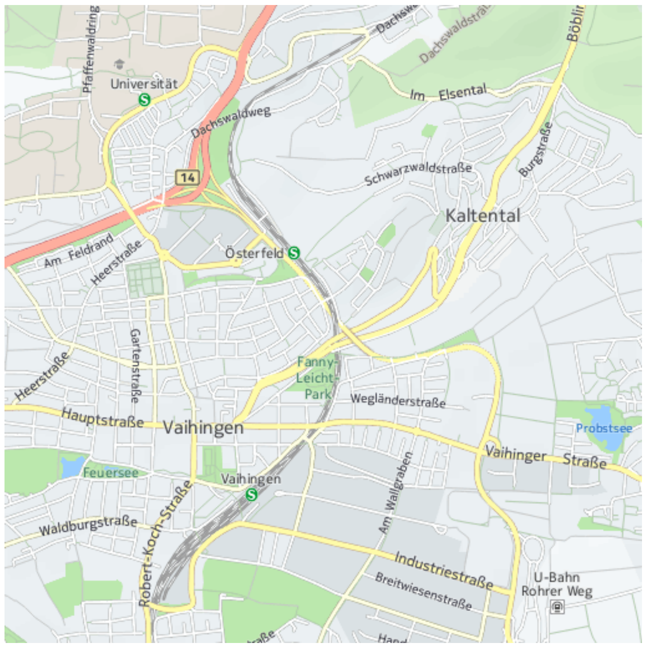
(d) OSM Cycle Map



(e) OSM Humanitarian



(f) OSM Transportation



(g) Nokia Here (uses twice the tile width)

Figure 2.5: Map tiles for different popular map services show different entities even if generated from the same data set. In particular the cycle map shows even the smallest roads while the transportation focused map shows fewer labels and highlights important roads whereas only the standard map shows building outlines. All these selection criteria are hand tuned and somewhat arbitrary.

We will now look into this tiling scheme in more detail as it uses several interesting techniques to simplify conversion between pixel and geographical coordinates as well as speed up the lookup of tiles. This overview is largely based on the Microsoft Developer Network resources on Bing Map tiles [Sch].

Map Projection

Geographical latitude and longitude coordinates are projected into the plane with the spherical *Mercator projection*. While this projection leads to significant distortion of scale and area particularly at the poles it has several advantages, it is a *conformal* projection meaning it preserves the shape of small objects such as building outlines and it is *cylindrical* which means that north and south are always straight up and down while west and east are always straight left and right. As the Mercator projection goes to infinity at the poles it can not be used as is, instead the map is cut to a square aspect ratio so that the maximum latitude is reduced to about 85.05 degrees.

Ground Resolution

As mentioned earlier this scheme uses a predefined set of simplification levels with fixed zoom factors that together form a zoom level. Thus each zoom level can be thought of as a (potentially very large) image of the entire map with a fixed simplification level. In Bing for example there are 23 such levels with level 1 being a 512×512 pixel overview and level 23 resulting in a $2{,}147{,}483{,}648 \times 2{,}147{,}483{,}648$ image covering the entire globe with $0.0187 \frac{\text{meters}}{\text{pixel}}$ In general the width/height of the map image at zoom level z is calculated as

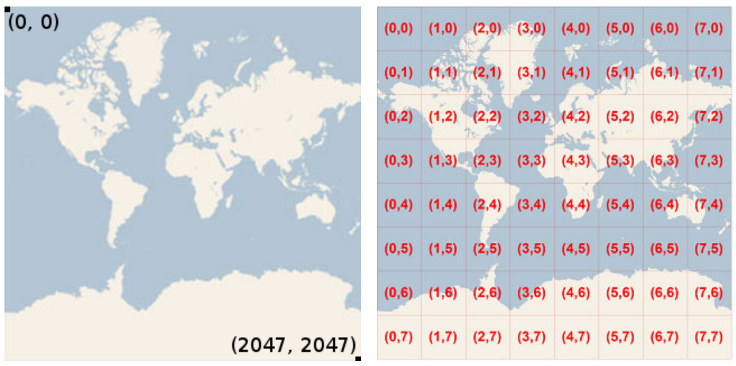
$$width = height = 256 * 2^l$$

while the ground resolution in meters per pixel can be computed using the radius of earth which is approximately $6{,}378{,}137$ meters

$$ground\_resolution = \frac{\cos(latitude\frac{\pi}{180})2\pi 6{,}378{,}137}{256 * 2^l} \frac{\text{meters}}{\text{pixel}}$$

Pixel Coordinates

Since each level has a different height and width the range of pixel coordinates within a level are similarly level dependent. In each level l coordinates range from $(0, 0)$ at the top left corner to $(width - 1, height - 1) = (256 * 2^l - 1, 256 * 2^l - 1)$ in the lower right corner. Figure 2.6a depicts these bounding coordinates. With the coordinate systems defined we can calculate



(a) Pixel Coordinates **(b)** Tile Coordinates

Figure 2.6: Pixel and tile coordinate systems at level $l = 3$ (Source [Sch])

the pixel coordinates in level l for geographical coordinates given as $latitude$ (clipped to the $[-85.05, 85.05]$ range) and $longitude$ as follows

$$\begin{aligned}
sinLatitude &= \sin(latitude\frac{\pi}{180}) \\
pixelX &= \frac{longitude + 180}{360} 256 * 2^l \\
pixelY &= (0.5 - \frac{\log(\frac{1+sinLatitude}{1-sinLatitude})}{4\pi}) 256 * 2^l
\end{aligned}$$

Tile Coordinates and Quadkeys

With a resolution of $256 * 2^l \times 256 * 2^l$ at level l map images for higher levels of detail would be several TB in size when considered as a single image. Such a high resolution would be prohibitively large for direct use especially when the data needs to be transferred over the Internet. It is therefore necessary to split images up into tiles, to facilitate this the factor 256 has been chosen so that we can easily split the map image at level l into 2^l tiles of size 256×256 (or 512×512 with Nokia Here). Additionally each tile is given XY coordinates within the levels partitioning into tiles, ranging from $(0, 0)$ in the upper left corner to $(2^l - 1, 2^l - 1)$ in the lower right, an example of which can be seen in Figure 2.6b.

For given pixel coordinates in the map image of level l we can now easily compute tile coordinates as

$$\begin{aligned}
tileX &= \lfloor \frac{pixelX}{256} \rfloor \\
tileY &= \lfloor \frac{pixelY}{256} \rfloor
\end{aligned}$$

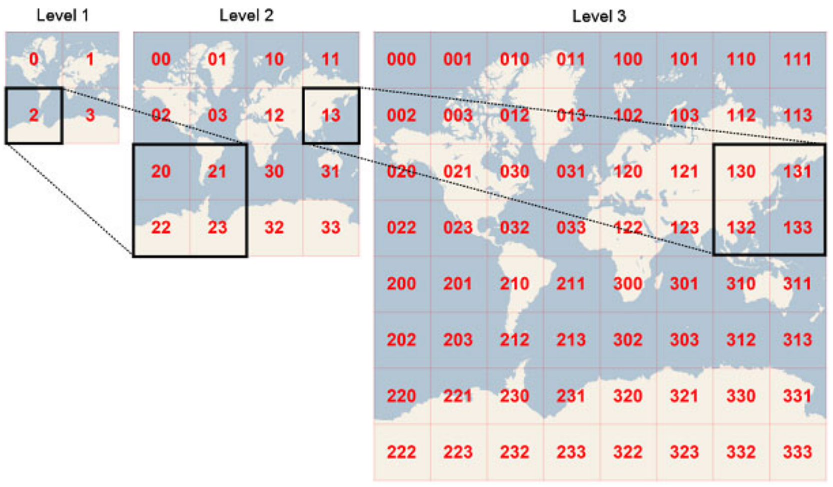


Figure 2.7: Quadkeys of subtiles at higher levels of detail have the quadkey of their parent as a prefix (Source [Sch])

To uniquely identify tiles even across levels in a way that can efficiently be stored in database indices while still being computable from just knowing the tile coordinates we resort to a trick. Each tile is identified with what we call a quadtree key known as *quadkey* for short. A quadkey is computed from a tiles x and y coordinates by interleaving their bits and interpreting the result as a 4-bit number retaining leading zeros and converting it to a string. For instance for the tile $(tileX, tileY) = (3, 5)$ at level $l = 3$ the quadkey is determined as follows

$$
\begin{aligned}
tileX &= 3 = 011_2 \\
tileY &= 5 = 101_2 \\
quadkey &= 100111_2 = 213_4 = \text{“213”}
\end{aligned}
$$

This scheme has several interesting and desirable properties, firstly a quadkeys length is equal to the zoom level of the corresponding tile, for example the key “213” in the example has length and level 3. Secondly the quadkeys of subtiles have their parent quadkey as a prefix. This is illustrated in Figure 2.7 where for example the lowest level tile “2” has the subtiles “20”,“21”,“22” and “23”. These properties also mean that quadkeys for neighboring tiles are often also close in lexical order which makes looking up neighboring tiles in a database or distributed key-value store much more performant.

3 URAR - Unified Routing and Rendering

In the previous chapter we introduced several existing techniques for speeding up the computation of shortest paths (Section 2.3) on road networks and looked into classic tile-based techniques to solve the map rendering problem for online services (Section 2.6). In the following we will present our contribution that tries to solve both the shortest path problem and online map rendering with a unified, parameterized and extendable approach based on Contraction Hierarchies. In particular it will allow us to use a single data set from which compact packets of data so called *bundles* are generated that are sufficient for rendering a view rectangle R at a particular parameterized level of simplification S (as described in Section 2.6.2), while also allowing to route between any two nodes visible in this or any other previously cached bundle.

Let us first note however that even though the scheme described in this thesis and especially the parts on edge selection are somewhat specific to the use of Contraction Hierarchy levels as measure for node importance similar schemes could be devised using another importance measure such as Reach[GKW06] or Highway Hierarchies[SS05].

At a high-level URAR uses Contraction Hierarchy levels as a measure of node importance by treating the CH level $\Phi(v)$ of a node v as its priority. When we want to render a view rectangle R we first decide upon a simplification level $S = (p, \sigma)$ where p specifies the minimum priority of nodes to render and σ specifies criteria determining which edges need to be drawn to achieve a pleasant rendering result. The edge selection criteria σ as well as p need to be chosen so that the level of detail matches the display capabilities such as pixel density, fits the size of the view rectangle R as appropriate for the application and needs to balance the size of the result with the drawing accuracy and possibly bandwidth requirements. More details of this can be found in Section 3.5.2.

We then construct a bundle $\beta(R, S, l)$ by first extracting the set of high importance nodes $H(R, p) \subseteq V$ by selecting those nodes from the underlying graph data structure that fall within R and that have priority at least $p \in \mathbb{N}_0$ and subsequently extracting their shared upwards and downwards graphs $\bigcup_{v \in H(R,p)} G^{\uparrow_{r_l}}(v)$ and $\bigcup_{v \in H(R,p)} G^{\downarrow_{r_l}}(v)$ where l is the level of the core graph already available bounding the upwards graphs. Finally we refine the edges of the upwards and downwards graphs according to the edge selection criteria σ. The result of which is a set of drawable *edge paths* that represent sequences of road segments of a length suitable to drawing as a single line at the current scale.

Once we have extracted a set of bundles $\mathrm{B} = \{\beta(R_i, S_i, l) | i \in 0 \ldots k\}$ of the same level l we can combine them with the matching core graph $G^{core}(l)$ to form a combined graph that contains all information necessary to route between all nodes in their respective view rectangles as well as the higher level nodes in the core graph while also rendering them as a combined road network.

3.1 Design Considerations

After this overview of the URAR algorithm we will now discuss a few design decisions we made for it, some of which still leave room for further research, refinements. As we gain a deeper understanding of the relation of CH levels and intuitive road importance.

3.1.1 Prioritization of Nodes vs Edges

When we think of importance of roads we may assign the measure of importance to the edges rather than, as we did, to the nodes of the graph and in fact for example for Reach[Gut04] such a definition is straight forward and useful. therefore instead of taking the apparent detour of first selecting important nodes lying within the view rectangle and then exploring edges we could just select important edges. For this selection process several standard data structures from computational geometry like range trees, segment or interval trees[BCKO08] can be used to report all edges intersecting a view in $O(\log^2 n + k)$ time and with a space consumption of $O(n \log n)$ where n is the total number of edges and k is the size of the query result. We can extend this easily to take importance values into account for an additional $\log n$ factor in query time. There are some caveats though: for one these data structures only work for non-intersecting segments which is not normally the case for road networks and much less so for CH's augmented graph G^+. Also our query time now depends on the total number of edges which for CH graphs is typically about four times the number of nodes. With our scheme of selecting nodes and then locally exploring upwards and downwards graphs the query time depends only on the total number of nodes and the number of edges in the *localized* upwards and downwards graphs.

3.1.2 Choosing Contraction Hierarchies vs Reach

We have chosen CH levels as our measure of importance not only because of a large body of prior experience but also because the freedom to contract nodes in any order gives us a lot flexibility in tuning the importance metric. For example to make sure some node is always visible it is sufficient to simply delay its contraction to the very end. The details of which adaptations of the contraction scheme yield the best results, which metrics to include and how to root their influences in sound theoretical models is still an open question of research however. While we know that such a scheme would likely try to delay the contraction of nodes defining the outline of a graph for example at country borders considerable further effort is necessary and goes beyond the scope of this work. Still while we see Contraction Hierarchies as the prime candidate for a sucessful unification of routing and rendering with a high level of flexibility trying other schemes as well should definitely be considered.

3.2 Bundles

The Unified Routing and Rendering concept hinges on the idea of a *bundle* as a sealed packet of graph data. Each bundle holds all the necessary information to render a map for its respective viewing rectangle and when combined with a core graph[1] allows routing between all nodes of this bundle as well as to and from nodes in other available bundles and the core graph.

Bundles may be cached and do not depend on each other meaning that they can be used to route with other bundles independent of the order in which they were acquired, their simplification levels or view rectangles and even their respective minimum node priority, the only requirements for compatibility being to be obtained from the same graph and for the same core level.

As a data structure bundles are remarkably simple and while we will discuss their practical encoding in Section 4.5 for now we can think of them as a static data structure consisting of the following substructures plus some metadata such as the original viewing rectangle and simplification level. Given a viewing rectangle R and a simplification level $S = (p, \sigma)$ with minimum priority p, edge selection criteria σ as well as a core level l a bundle $\beta(R, S, l)$ contains the following:

- Bundle nodes $V^{\beta}(R, p, l)$
- Upwards edges $E^{\uparrow\beta}(R, S, l)$
- Downwards edges $E^{\downarrow\beta}(R, S, l)$
- A set of *edge paths* associated with the upwards and downwards edges $\mathrm{P}(\beta(R, S)) = \{\rho(e, \sigma) | e \in E^{\uparrow\beta}(R, S, l) \cup E^{\downarrow\beta}(R, S, l)\}$ which directly or indirectly include drawing information such as coordinates, road segment types etc.

[1]The core graph can technically be empty but this leads to, prohibitively large bundles see Section 4.6.3

3.2.1 Node Selection

First we need to select the high importance nodes $H(R, p)$ that lie within the query rectangle $R = (x, y, width, height)$ and have priority at least p. A simple approach for this would be to overlay the graph with a grid and then storing the nodes within each cell in order of descending priority, then for a query we would simply enumerate all grid cells intersecting the query rectangle and output all nodes from the cells lists that fall within the query rectangle and have priority greater or equal p. The problem with this approach is that it is not clear how to choose a grid size that fits all query rectangle sizes. When grid cells are too small there will be many empty ones, when they are too large intersecting cells may contain many nodes that do not fall within the query rectangle that will still need to be scanned. To handle large maps for example using the OSM planet data set we would thus have to resort to more complicated multi grid variants.

The Bounding Box Priority Search Tree

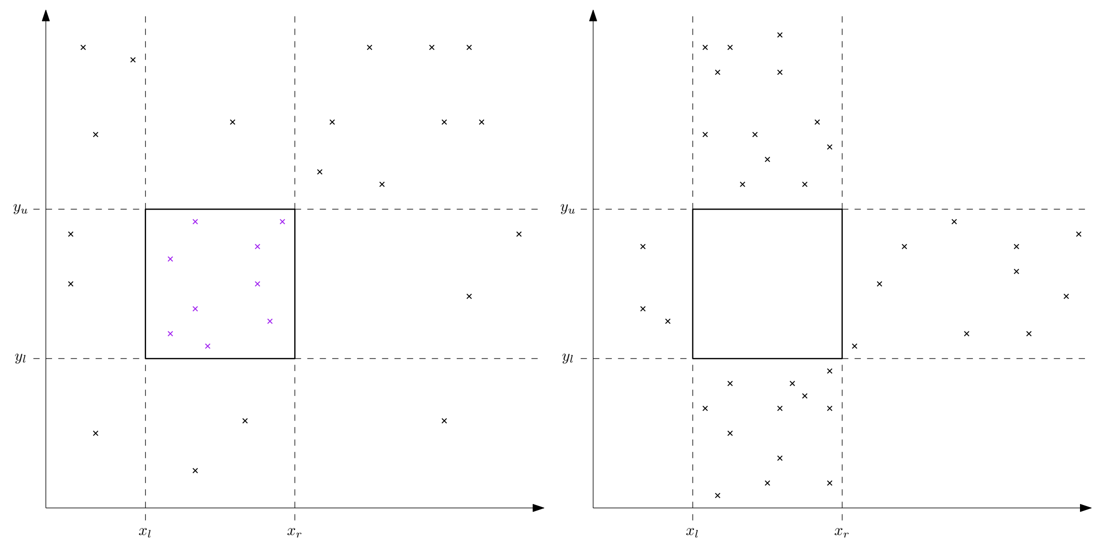
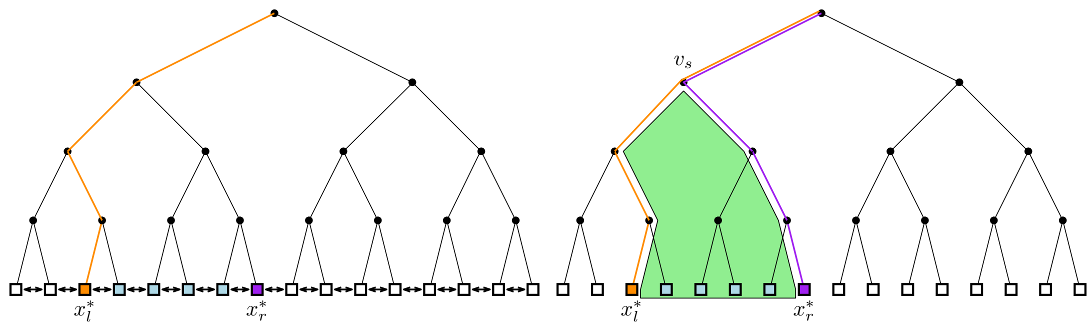


Figure 3.1: Intersecting the results of independent range tree queries on an x-coordinate and a y-coordinate range tree may fare poorly. While the example on the left is still fine we need to scan all data points in the right example resulting in $\Omega(n)$ query time only to end up with an empty result set.

Instead we look for help in computational geometry, by realizing that the query rectangle, also known as a bounding box, is just a generalization of a range in two dimensions, where instead of $R = (x, y, width, height)$ we often write $R = [x_l, x_r] \times [y_l, y_u]$ with $x_l = x, x_r = x + width$ and $y_l = y, y_u = y + height$. Sometimes we also see switched definitions of y_l and y_u depending on whether we see y coordinates as increasing upwards as in classical euclidean coordinate systems or downwards as is common in computer graphics. For range queries we find *range trees* as a simple yet powerful data structure. In one dimension such a range tree is simply a balanced binary search tree where to answer a query for a range $[x_l, x_r]$ we simply search for x_l and report all larger data points that are still smaller than x_r, this may for example be realized as a simple list traversal by storing all data at the leaves and keeping them organized in a linked list reporting all data points in the leaves starting at the node we found as next larger or equal to x_l and ending once we find a node larger than x_r. This results in an overall query time of $O(\log n + K)$ where K is the number of data points in the output. With this in hand a straight forward approach to answering a two dimensional range query for $[x_l, x_r] \times [y_l, y_u]$ would see us constructing two range trees, one on x-coordinates and one on y-coordinates, searching for both ranges $[x_l, x_r]$ and $[y_l, y_u]$ and then reporting their intersection. This naïve solution however is problematic in that the sub results may be very large while their intersection can be small or even empty. Figure 3.1 shows a scenario in which the subresults encompass all data points yet the final result is empty. The query time of this approach is thus $\Omega(n)$ for inspecting the subresults even though we only output an empty result. Instead what we really want is what is known as an *output sensitive* data structure with only a poly-logarithmic term like $\log n$ depending on the data set size and an ideally linear term depending on the output size.



(a) Range query traversing list of leaves **(b)** Range query reporting interior trees

Figure 3.2: We can compute a range query on a one dimensional range tree either by searching for x_l and then traversing the list of leaves until we find the last data point smaller than x_r or searching for x_l and x_r and reporting all trees hanging to inside of the searches and below the split point v_s, here shown as the light green area. We use x^* to denote the data point closest to the original query x.

To achieve this we need to combine the range trees for x and y coordinates instead of merely using them side by side. Looking at a range query in a one dimensional range tree we see that if we search for x_l and x_r together the searches split at some vertex v_s where the search for x_l continues left and the search for x_r continues right. We know that the result of the query is the disjoint union of the leaves inside the area $x_l^* \swarrow v_s \searrow x_r^*$ because these are exactly what a traversal of the leaves from x_l^* to x_r^* would report. See also Figure 3.2. This also means however that after the split all subtrees hanging to the right of the search path for x_l as well as to the left of the search path for x_r are wholly contained in the result while only the $O(\log n)$ data points directly on the path need to be explicitly checked. We can thus store at each internal node its entire subtree organized as another range tree organized on the y coordinates to form a combined two dimensional range tree (Figure 3.3). Now instead of directly reporting the result data points we first report the subtrees as $O(\log n)$ batches as we follow the searches for x_l and x_r and then query these subtrees for $[y_l, y_u]$ leading to an overall query time of $O(\log^2 n + K)$ and a space consumption of $O(n \log n)$. With an additional optimization called *Fractional Cascading* this can even be reduced to $O(\log n + K)$ time by taking advantage off the fact that the y-searches on subtrees are highly correlated as we search all of them for the same range while the subtrees are also hierarchical.

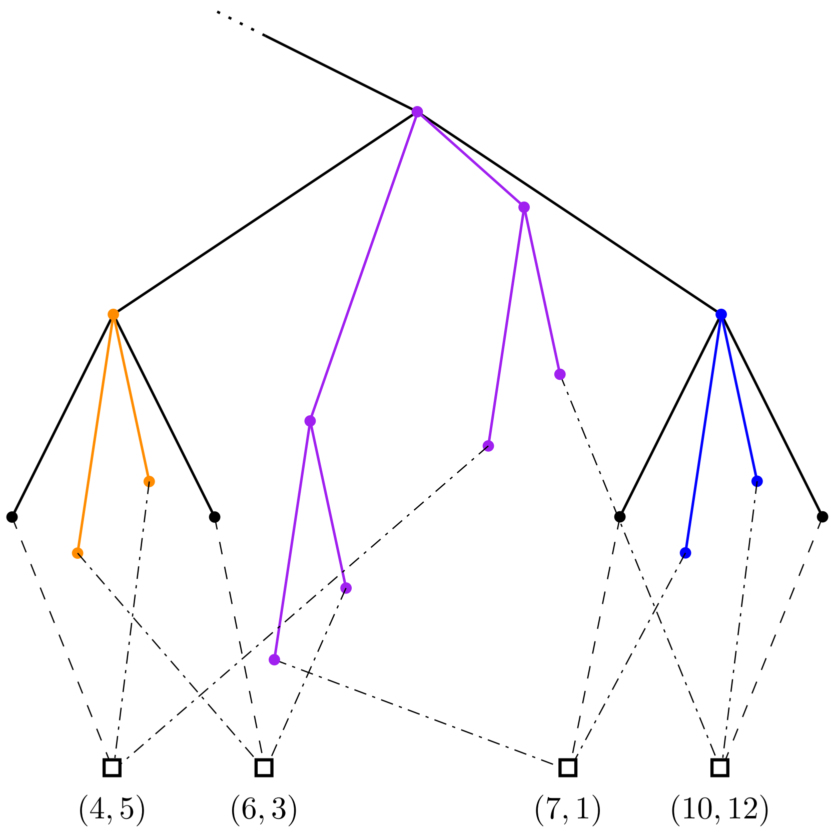


Figure 3.3: In a two dimensional range tree we use one primary range tree organized by x-coordinates in which we store at every node its entire subtree organized by y-coordinates. The range tree on x-coordinates is drawn in black while organized subtrees are drawn in orange, purple and blue.

Taking priorities into account we could simply regard them as a third dimension and use another cascaded range tree that we would search for the range $[p, maxPriority]$ in the priority dimension. This would result in a query time of $O(\log^2 n + K)$ (with Fractional Cascading) using $O(n * \log^2 n)$ space. Especially the space consumption could proof problematic though. Fortunately there exists a much better data structure called a *Priority Search Tree* that handles two dimensional range queries where one of the intervals is half-open, in our case combining the y-dimension with the priorities as $[y_l, y_u] \times [p, \infty]$, by blending a binary search tree with a heap. This is achieved by recursively constructing a binary tree where each node T holds a y-key and a priority-key as follows. Let P be the set of data points to store. First we find the point with maximum priority p_{max} and store it as the priority-key in T, then we find the y-median of the remaining nodes $P \setminus \{p_{\text{max}}\}$ and store the respective y value as y-key in T. Lastly we partition $P \setminus \{p_{\text{max}}\}$ into two sets with smaller/larger y-coordinates than the y-key that we recurse into to build the left and right subtrees of T. Clearly this data structure takes $O(n)$ space and has a height of $O(\log n)$. To query it for a range $[y_l, y_u] \times [p, \infty]$ we search for y_l and y_u and then treat the subtrees hanging between both searches which contain all nodes covering the correct y-range as max-heaps while only considering priority-keys.

If we combine this data structure with a range tree over the x coordinates by using it to organize the subtrees of internal nodes analogously to how we cascaded range trees we end up with what we call a *Bounding Box Priority Search Tree* that answers queries of the form R, p in $O(\log n + K)$ with a space consumption of $O(n \log n)$ fitting our requirements for node selection. One thing that may still pose problems is the space consumption, especially in mobile settings and with increasingly large data sets even using $\gamma n \log n$ space can be prohibitive. Ideally we want to be able to trade lower performance for lowered space use and indeed we can. Instead of storing its entire organized subtree at every node we may only store subtrees at nodes in every r-th level. During queries we would then have to (in the worst case) query the 2^{r-1} organized subtrees of nodes at most $r - 1$ levels below. Resulting in an increased query time of $O(2^{r-1} \log^2 n + K)$ and a reduced space use of $\gamma n \frac{\log n}{r}$.

Expanding to the Upwards and Downwards Graphs

We have now extracted the high priority nodes $H(R, p)$, however these nodes plus the edges among them alone are not sufficient for routing. Recall that in Contraction Hierarchies shortest paths always have a representation of upwards and then downwards edges and that the core graph we will combine the bundles with already covers all nodes and edges above the core level l. To cover all shortest paths leading to and from extracted nodes we thus need to expand them to cover the entire upwards and downwards graphs of the high priority nodes up to the core. With the definitions of Section 2.5.3 and a simplification level $S = (p, \sigma)$ the set of nodes we really need for a bundle is then:

$$V^{\beta}(R, p, l) = \bigcup_{v \in H(R,p)} \left(V^{\uparrow r_l}(v) \cup V^{\downarrow r_l}(v) \right) \setminus V^{core}$$

These can obtained by a simple depth-first search that is initialized with the high priority nodes $H(R, p)$ and expands this set by all nodes it discovers up to the core.

There is another important property of Contraction Hierarchies that we can take advantage off however. Because of their upwards then downwards property shortest paths $\pi_{s,t}$ in G^+ can be found [2] by relaxing edges in $G^\uparrow$ ($G^\downarrow$) in a topological ordering of their source (target) nodes s_i (t_i). Relaxing here means to check whether an edge leads to a smaller distance labeling and updating it if it does. Assuming we initialized the distance label of the source (target) node $d(s,s) = 0$ ($d(t,t) = 0$) and $d(s,v) = \infty \forall v \neq s$ ($d(v,t) = \infty \forall v \neq t$) the topological ordering ensures that the distance labels of nodes are correct once they are used to compute new distance labels. This is because edges in $G^\uparrow$ ($G^\downarrow$) are relaxed in order of increasing source (target) rank aligned with the order they appear in the shortest path $\pi_{s,t}$, we also know that such an ordering exists because the upwards and downwards graphs of a node v are trees and their unions being subsets of $G^\uparrow$ ($G^\downarrow$) are acyclic.

By topologically ordering the upwards and downwards graphs we can thus ensure that we only need to scan them in that order to relax edges instead of using a more complicated and less efficient Dijkstra search.

3.2.2 Edge Selection and Refinement

If we only had the nodes of a bundle $V^\beta(R,p,l)$ we could draw nothing but a bunch of points and routing would not make much sense, we clearly need edges too. For routing knowing which edges we need is easy, those are simply all edges in the union of the respective upwards and downwards graphs which we keep separated into upwards and downwards edge sets:

$$E^{\uparrow\beta}(R,p,l) = \bigcup_{v \in H(R,p)} E^{\uparrow r_l}(v)$$
$$E^{\downarrow\beta}(R,p,l) = \bigcup_{v \in H(R,p)} E^{\downarrow r_l}(v)$$

As these edge sets are optimized to be small during CH construction which is important for routing performance they include mostly shortcuts often of considerable length recursively shortcutting a large number of edges. therefore simply drawing these edges gives very unpleasant results and we need to unpack them. In principle unpacking means recursively replacing a shortcut with the two edges it originally shortcut during node contraction. Since we want to keep routing performance high and as we do not want to include even more nodes for which we would then also have to include their possibly still uncovered upwards and downwards graph we introduce the concept of an *edge path*. An edge path of an edge $e = (u,v) \in E^+$ is a path of edges from G^+ that is a partially or fully unpacked version of e consisting only of edges fulfilling the edge selection criteria σ. Formally we use the notation

$$\rho(e,\sigma) = e_0 e_1 \dots e_k, e_0 = (u,\cdot), e_k = (\cdot,v)$$

[2]To be exact we will not find all shortest paths in G^+ but a representation for every shortest path in the original G

As it is just a refinement of an edge it is a path of degree 2 nodes, shares the original edges length and starts and ends with that edges source and target. The refinement process of unpacking an edge path is illustrated in Figure 3.4.

Edge Selection Criteria

The selection criteria σ is a parameterized selection function that heuristically selects edges for inclusion in the final result by either deciding to continue unpacking an edge or including it as is. It is generally designed to select edges in a way that the overall edge path $\rho(e, \sigma)$ when rendered will represent the part of the road network represented by the edge faithfully and visually appealing. Faithfully in that it should retain a strong level of correspondence between the rendered path and the geometrical shape of the real road network. In particular edge selection should retain significant parts of the shape of roads such as characteristic twists of serpentines, rounded shapes of large highway curves and block structure of city streets while simplifying shapes in a visually pleasing way at lower levels of detail. Additionally it should balance these requirements with the need to keep bundles small especially when URAR is used in a client-server scheme as it is the case for our prototype implementation.

Since both the requirements for bundle sizes as well as rendered level of detail depend on properties of the display device for example its pixel density, display size, current map scaling and network bandwidth the selection parameters may change for every bundle construction. Some useful parameters are listed below and Section 4.4 describes their values as used in our prototype implementations.

- Checking whether an edge lies entirely or partially outside the requested view rectangle and is thus part of the upwards/downwards graph but not useful for drawing the content of the view rectangle. We may need it for drawing a path though so we may want to include a coarse version.
- Minimum and maximum euclidean edge lengths so that edges shorter than the minimum length are always included as is while edges larger than the maximum are always unpacked further. It is useful to scale these lengths in accordance with the map scaling on the displaying device. Also the minimum length should be chosen so that edges of this length can be used to render rounded structure at the current scaling.
- Given a shortcut edge $e = (u, w)$ and the two edges it shortcuts $s_1 = (u, v)$ and $s_2 = (v, w)$ we define its *shortcut bending ratio* as the ratio between the height of the triangle uvw and its baseline. It is a good indicator of the bending of a shortcut independent of its

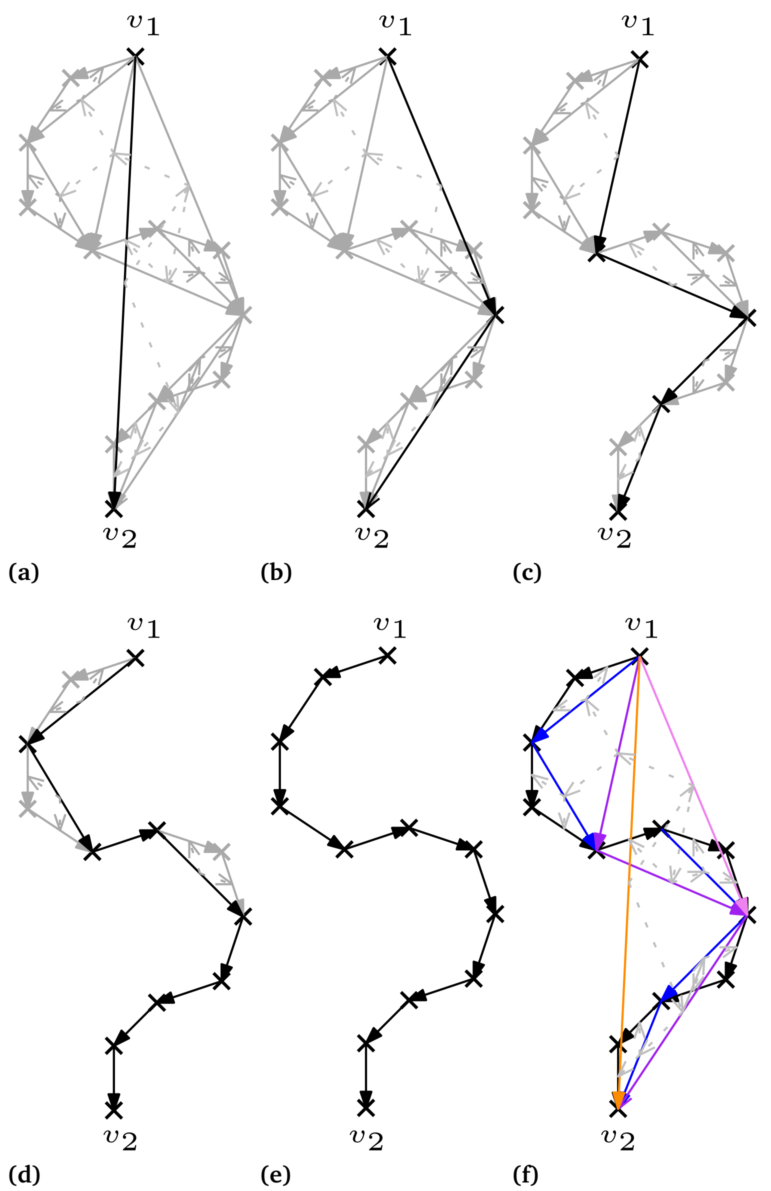


Figure 3.4: The edge (v_1, v_2) (orange in Figure 3.4f) is refined by recursive unpacking resulting in a sequence of increasingly detailed edge paths (Figure 3.4a through Figure 3.4e), drawn in black. The gray edges remain unpacked with dashed arrows indicating which edges a shortcut replaced. In Figure 3.4f we can see all edges as they are part of the Contraction Hierarchy. Edge colors encode the order of contraction black edges are the original edges in G while colored edges are shortcuts created in the order blue, purple, violet, orange.

length. It may be computed efficiently from coordinates via a determinant formula for the area of a triangle [3] and $A = \frac{1}{2} Baseline * Height$

$$A = \text{abs}\left(\frac{1}{2}\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}\right)$$

Edge Unification and Separation of Drawing Data

Apart from choosing the correct edge selection criteria σ and node priority p there is another issue that needs to be handled to achieve small bundle sizes and reduce unnecessary data to be included. The refinement of single edges as described in Section 3.2.2 always yields a single edge path with no intersections or cycles and is therefore free from duplication, when extracting multiple edges together to obtain all the edge paths of a subgraph however there is considerable overlap and duplication. Figure 3.5 illustrates this with an example and as nodes of higher degree are shortcut more shortcuts can share subpaths. Without special attention this leads to larger bundles as they contain the same paths repeatedly as part of different edge paths including a duplication of coordinates when all necessary information for drawing also needs to be stored in a bundle as it is the case for our client-server based prototype implementation. Specifics on the influence of duplicated edges and the effects of different ways to encode the edge paths can be found in Section 4.5.3 and we will only mention some constraints on these techniques in this section.

At first this seems like an easy to solve problem, after all during edge refinement we could simply remember which edges have already been refined and stored for later drawing. As we always unpack edges top down we could then simply ignore these edges when we see them again and would thus eliminate all duplication in edge paths.

Since we have logically separated edge paths from the true edges of the upwards and downwards graphs this also does not interfere with routing although we might end up with edges without any edge path when the edge itself is part of some edge path. The problem now is that once we are done routing and need to be able to draw a sequence of upwards and downwards edges in refined form to render the route appropriately for the current simplification we encounter edges with holes in their edge path and no way to find out which parts of other edge paths should be drawn to fill them.

One solution for this is to restrict the unification of edge paths to the individual path segments instead of also unifying edges that are unpacked further. These path segments are then stored as separate drawing data. With this change edge paths become simple lists of references to these segments, which still allows us to lookup which segments to draw for a particular

[3] See `https://people.richland.edu/james/lecture/m116/matrices/applications.html` for an explanation

upwards or downwards edge while preventing the duplication of other data such as road types, coordinates and direction. Since this data is only used for drawing purposes we may even unify edges of the same length but switched source and target which is very beneficial as we have modeled (see Section 2.6.1) road segments that can be used in both directions as two unidirectional edges while for rendering purposes it makes sense to draw them as a single road.

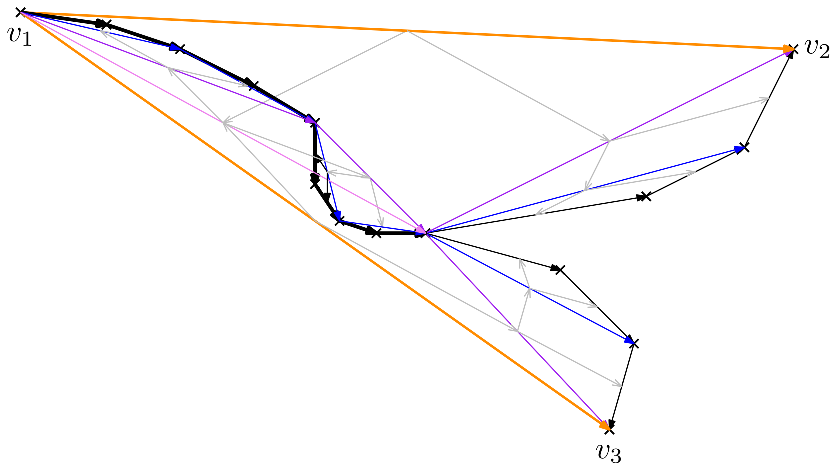


Figure 3.5: In this example a significant part of the original path (black and thicker lines) is shared between the two orange higher level shortcuts and is therefore part of both their edge paths. Thus if we stored all necessary drawing information directly in the edge paths of (v_1, v_2) and (v_1, v_3) we would end up with duplicated data. As in Figure 3.4 colors encode the order of contraction (blue, purple, violet, orange).

A Note on Edge Types and the Travel Time Metric

Until now we have assumed that all edges whether from the original G or the CH graph G^+ have associated edge types to distinguish between different types of roads. As the shortcut edges in G^+ are created from two edges during a node contraction it is not clear what edge type they should have as road types are often arbitrary and there is no notion of mixing them. One solution to this would be to tweak the contraction phase so that we only combine edges of the same type and this option should probably be considered during further work into tuning the contraction for rendering, for our prototype implementation however we chose a simpler solution. Instead of combining edge types we take advantage of the travel time metric we based on the maximum speed of a road type and distances to calculate the average max speed on an edge and use that as its type.

To calculate travel time costs we use the following formula where the maximum speed value (in km/h) may be hand tuned

$$cost(e) = \frac{1.3 * length(e)}{0.01 * maxspeed(e)}$$

which means after the CH preprocessing where we keep track of both the (euclidean) length of shortcuts and their cost as the sum of the respective values of the edges they shortcut we can calculate the maximum speed of any edge as.

$$maxspeed(e) = \frac{130 * length(e)}{cost(e)}$$

We can then use this value to define classes of edges for example classifying every edge with an average maximum speed over 120 km/h as a speedway. While this results in visually pleasing results and allows for a clear definition, the impact on usability is less clear as users may expect the color of edges on the map to always represent the exact type of the road. This is somewhat alleviated by the fact that these speed based road types get more fine grained as the level of simplification is decreased with negligible inconsistencies at the finest view. Potential inconsistencies thus mainly play a role when looking at overviews and one may argue that in these average max speed may even be more representative than classical exact road types that may produce small sections of differing road type which may overstate the importance of that section.

3.3 The Renderable Core Graph

While we could use only bundles for both routing and rendering by keeping the upwards and downwards graphs unrestricted and thus covering all necessary edges for shortest path queries within as well as across bundles, this would unnecessarily increase their size and keep some important properties of Contraction Hierarchies unexploited. As we discussed in previous work [SFS13, Sch13] the size of upwards and downwards graphs significantly decreases when we restrict them to below some mid-range CH level and combine everything above that level into a core graph that is independent of the query and still of manageable size. This is especially effective since the higher level parts of each upwards and downwards graph are less localized and largely shared between different parts of the graph, as an extreme example when contracting until only one node remains this node will be shared by all upwards and downwards graphs.

To make the core graph as defined in Section 2.5.3 compatible with rendering of both computed paths as well as general map display we augment its definition with a set of edge paths associated with the edges. Note that while each edge is associated with exactly one edge path, an edge path could be associated with several edges for example it could be shared by edges between the same nodes but going in opposite directions. Unlike bundles and in line with our previous definition our core graph is flattened, meaning we do not distinguish between upwards and downwards edges. We thus define the renderable core graph (from here on simply referred to as the core graph) as

$$\begin{aligned}
G^{core}(l) &= (V^{core}(l), E^{core}(l), \mathrm{P}^{core}),\\
V^{core}(l) &= \{v \in V | \Phi(v) \geq l\},\\
E^{core}(l) &= \{(u,v) \in E^{+} | u, v \in V^{core}(l)\}\\
\mathrm{P}^{core}(\sigma) &= \{\rho(e,\sigma) | e \in E^{core}(l)\}
\end{aligned}$$

As we may use the same core graph at every scaling and because it is defined globally and not only within some view rectangle R the edge selection function σ should be chosen conservatively to allow fine drawing at all expected scalings and we will look at some practical considerations in Section 4.4. Of course we also apply edge unification as described in Section 3.2.2 to the core where this is particular effective because high level nodes have higher degree and thus also share more and longer subpaths than bundles.

3.3.1 The Core Shortcut Optimization

One problem with this definition of the core graph is that for a shortcut that skips a node v that already has a CH level $\Phi(v) \geq l$ the core will include both the shortcut as well as both shortcutted edges. As we will later use a flattened version of the core where we ignore levels beyond the core we may drop these shortcuts and rely only on their constituent edges. The core graph edge set thus becomes

$$E^{core}(l) = \{(u,v) \in E^{+} | u, v \in V^{core}(l)\} \setminus \{e \in E^{+} | e \text{ shortcuts a node } v \text{ with } \Phi(v) \geq l\}$$

3.4 The Combined Graph

To be able to route between the available bundles as well as the core graph they need to be combined into one at least conceptually. From a theoretical point of view this just means using the union of the graphs, given a set of bundles $\mathrm{B} = \{\beta(R_i, S_i, l) | i \in 0 \ldots k\}$ and a core graph $G^{core}(l) = (V^{core}(l), E^{core}(l), \mathrm{P}^{core}(\sigma))$ the combined graph is then simply defined as follows, where analogously to the individual bundle and the core graph each edge is associated with exactly one edge path.

$$
\begin{aligned}
G^C &= (V^U, E^U, \mathrm{P}^U) \\
V^C &= \left(\bigcup_{\beta=(V^\beta, E^{\uparrow\beta}, E^{\downarrow\beta}, \mathrm{P}^\beta)\in\mathrm{B}} V^\beta \right) \cup V^{core} \\
E^C &= \left(\bigcup_{\beta=(V^\beta, E^{\uparrow\beta}, E^{\downarrow\beta}, \mathrm{P}^\beta)\in\mathrm{B}} E^{\uparrow\beta} \cup E^{\downarrow\beta} \right) \cup E^{core} \\
\mathrm{P}^U &= \left(\bigcup_{\beta=(V^\beta, E^{\uparrow\beta}, E^{\downarrow\beta}, \mathrm{P}^\beta)\in\mathrm{B}} \mathrm{P}^\beta \right) \cup \mathrm{P}^{core}
\end{aligned}
$$

This is very similar to the augmented core graph we constructed from an upwards, a downwards and the core graph in [SFS13] that was only used for routing. One major shortcoming with the augmented graph there was the need for resorting to hash maps to map nodes to their adjacent edges to deal with non-contiguous node ids that prevented us from using more traditional graph data structures. While hash maps generally work quite well bad interactions between their employed hash function and the hashed ids sometimes leads to erratic growth in lookup times which necessitates a close evaluation of the employed hash function thereby increasing implementation difficulty.

3.4.1 Dealing with Non-Contiguous Node IDs

For this work we have developed a different approach for dealing with non-contiguous node ids that requires no more sophisticated data structures than arrays to store and work with the combined graph. Let us first look into why non-contiguous node ids are problematic. In the high performance graph data structures used in our backend implementation node ids directly correspond to array indices into both the arrays associating information with nodes such as their coordinates and into offset arrays that allow us to lookup adjacent incoming and outgoing edges. With a straightforward use of non-contiguous node ids these array indices would have to be replaced with hash maps, now if we only extracted a single subgraph this could easily be avoided by remapping the node ids to the range of the subgraphs node set, in our case however we are combining several subgraphs originating from the same source graph without knowing beforehand which subgraphs will be combined or even how many there will be. To be able to combine the subgraphs we need to retain some correspondence between them particularly at the boundary edges that have their source in one subgraph and their target in another or the other way around.

Taking a look at how our routing scheme (see Section 3.6) relaxes edges across subgraph boundaries we realize that the only edges going across subgraph boundaries are the upwards or downwards edges at the top of each bundle crossing into the core graph. Between bundles on the other hand we do not need to relax edges as the upwards and downwards searches use naturally disjunct edge sets. Instead we only need to merge distance values as described in Section 3.6.3 which can be done in a single pass instead of intervened with relaxing edges within the bundles subgraph with the additional advantage that there are far fewer nodes and thus distance values than there are edges to relax.

We therefore only need node ids in the bundles to be compatible with the core instead of also with other bundles. It is also clear that as the bundles are restricted to nodes below the core level their node sets do not overlap with the core. We can exploit this to create an ordering of nodes such that all node ids in the core are contiguous by simply ordering the nodes of the graph in descending order of their levels thereby confining the core graph to the node id range $[0, |V^{core}| - 1]$. To make the node ids in a bundle β compatible we simply map them to $[|V^{core}|, |V^{core}| + |V^{\beta}| - 1]$ during its construction where we simply leave the source/target ids of edges going into or out of the core unchanged. To use these ids as array indices in the bundle we then only have to subtract $|V^{core}|$. While this scheme leaves node ids in different bundles as overlapping they are disjunct and compatible with the node ids in the core, additionally we can simply detect whether a node is part of the core by checking whether its id is below $|V^{core}|$.

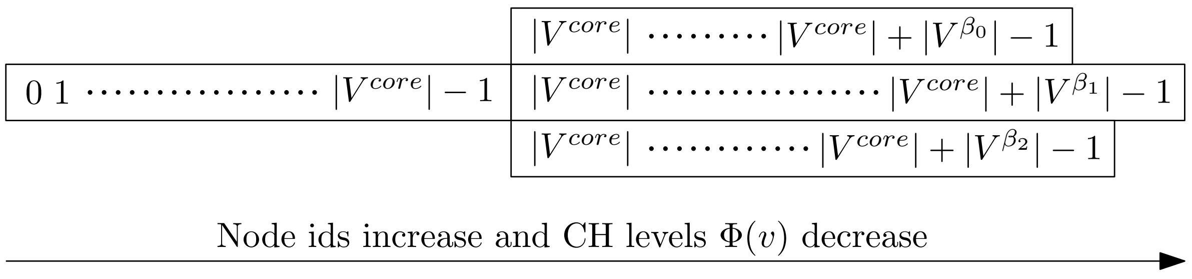

$|V^{core}| \cdots\cdots\cdots |V^{core}| + |V^{\beta_0}| - 1$
$0 \; 1 \cdots\cdots\cdots |V^{core}| - 1$
$|V^{core}| \cdots\cdots\cdots |V^{core}| + |V^{\beta_1}| - 1$
$|V^{core}| \cdots\cdots\cdots |V^{core}| + |V^{\beta_2}| - 1$
Node ids increase and CH levels $\Phi(v)$ decrease

Figure 3.6: By ordering nodes according to their level with high levels being assigned low node ids we confine the core to the id range $[0, |V^{core}| - 1]$ and bundles β_i to $[|V^{core}|, |V^{core}| + |V_i^{\beta}| - 1]$ we create a contiguous node id range between the core and each bundle.

Finally this allows us to drop the requirement of using everything above some level l as the core, instead we may simply use the k nodes with lowest ids and the edges among them to create a core of size k altering the definitions slightly by replacing checks for the correct level by checks for the node id value.

3.5 Map Rendering

We will now take a look at how we can use a set of bundles $\mathrm{B} = \{\beta(R_i, S_i, l) | i \in 0 \ldots k\}$ and a core graph $G^{core}(l, \sigma) = (V^{core}(l), E^{core}(l), \mathrm{P}^{core}(\sigma))$ to render a map. Here we will focus on the case of using projected x,y coordinates as using latitudes and longitudes directly for example by rendering onto a 3D sphere is conceptually very simple because coordinates can be used as is, though the implementation with a framework like OpenGL can be tricky in practice. Note however that our remarks on panning consistency and varying levels of simplification remain applicable.

Usually when rendering on some display we have to draw onto some rectangular axis aligned canvas and use input devices such as a mouse to zoom in and out and to pan the map for example by dragging it. While the size of the canvas is not fixed and can change for example when resizing the window it remains fixed during normal panning and zooming operations and therefore we may think of it as being fixed for the remainder of this section. We model this canvas very similarly to the view rectangle with the main difference being that the view rectangle is always expressed in the coordinates of the graph (either geographical or projected x,y coordinates) and its size and position always representing the visible region of the graph. The canvas rectangle $R^C = (x^C, y^C, width^C, height^C)$ on the other hand is expressed in screen or its own coordinates (in the latter case its x,y often remain fixed to the origin) and does not change during panning or zooming operations. Instead we independently move and resize the view rectangle $R^V = (x^V, y^V, width^V, height^V)$ as the current view of the graph. Keeping bundles, the core graph and the view rectangle in graph coordinates only projecting them into the canvas coordinate space on drawing. This is illustrated in Figure 3.7.

As the view rectangle represents the visible part of the graph we want to project a scaled down version of it onto the drawing canvas allowing for overlap only to match the aspect ratio. This can be achieved by using a scaling factor computed as

$$scale = \max\left(\frac{width^V}{width^C}, \frac{height^V}{height^C}\right)$$

with this factor we can now project coordinates from the graphs coordinate space to the canvas space and back and also convert lengths between them which will be particularly helpful for specifying the edge selection functions minimal and maximal length parameters in terms of canvas coordinates keeping them consistent across zooming operations. The formulas for this are as follows where $\cdot^*$ denotes the data point to project and $\cdot^C$ denotes coordinates in canvas space while $\cdot^V$ denotes the view or graph space.

$$
\begin{aligned}
x^{*C} &= \frac{x^{*V} - x^V}{scale + x^C} \\
y^{*C} &= \frac{y^{*V} - y^V}{scale + y^C} \\
d^{*C} &= \frac{d^{*V}}{scale}
\end{aligned}
$$

$$
\begin{aligned}
x^{*V} &= (x^{*C} - x^C) * scale + x^V \\
y^{*V} &= (y^{*C} - y^C) * scale + y^V \\
d^{*V} &= d^{*C} * scale
\end{aligned}
$$

Since the coordinates stored with the edge paths of the core P^{core} and the bundles P^{β_i} are expressed in graph coordinates we can now draw them by first projecting into canvas coordinates.

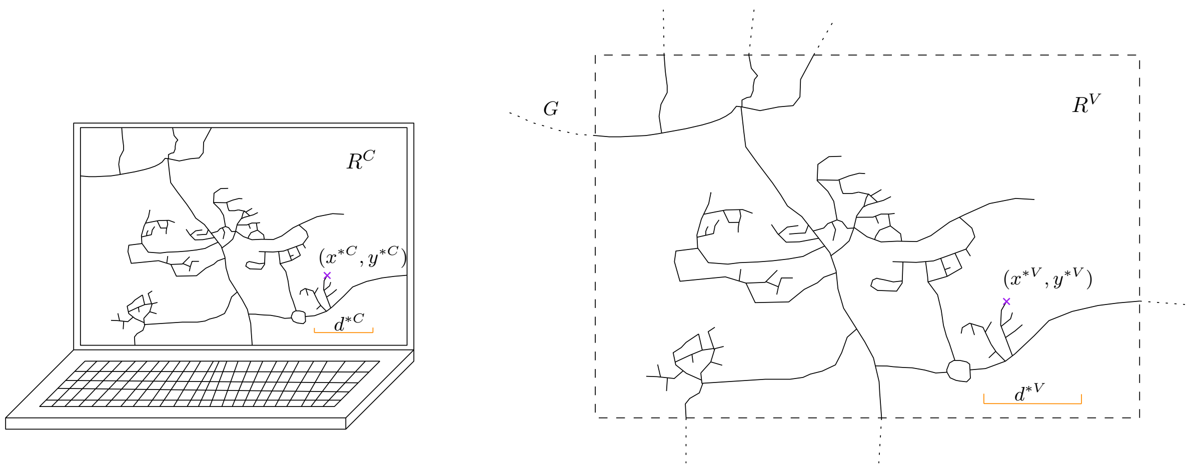


Figure 3.7: Here a view rectangle R^V showing a part of the graph G is projected onto a canvas R^C. The point x^* and the length d^* are shown in both the canvas and view coordinate systems.

3.5.1 Panning and Zooming

To pan the map by a change $(\Delta x, \Delta y)$ in canvas coordinates as we would get for example from dragging with the mouse we simply move the view rectangle by the change distance as projected into graph coordinates using the formula from the previous section. Panning now takes the form $R = (x, y, width, height) \to R' = (x + \Delta x * scale, y + \Delta y * scale, width, height)$.

After a panning operation the view rectangle has changed and we would need to request a new bundle to cover it unless we already have a cached bundle for it. This would then lead to new parts of the map suddenly appearing instead of being panned in smoothly which is less appealing. Instead we make use of the fact that dragging is restricted to move the view by less than or equal its diagonal and instead of extracting bundles for exactly the view rectangle we request bundles for an *extended view rectangle* of 9 times the size which ensures that no panning operation will get us outside of map data we can draw. We can thus draw existing data during panning and only request new data once panning has stopped which when using a mouse to drag the map coincides with the user repositioning making request times less noticeable.

To zoom by a factor z we simply scale the (extended) view rectangles $width$ and $height$ by z, as this alone would increase the size of the view without moving its center which looks unnatural we additionally move the view rectangle so as to bring the mouse pointer closer to its center. Thus a zooming operation can be described formally as $R = (x, y, width, height) \to R' = (x + \Delta x * scale, y + \Delta y * scale, width * z, height * z)$. As zooming may always bring new data into view either by increasing the visible area or increasing the level of detail required to render a smaller view rectangle on a canvas of unchanged size we always request a new bundle when zooming.

3.5.2 Varying Level of Simplification

We introduced the level of simplification of a bundle as a tuple $S = (p, \sigma)$ where p is the minimum priority of nodes in the bundle (CH level $\Phi(v)$) and σ is an edge selection function and listed some possible edge selection criteria in Section 3.2.2. Now we will take a look at how these criteria can be applied to achieve a consistent level of detail across zoom and panning operations that can be fine tuned to specific rendering parameters such as display/canvas size, pixel density and network bandwidth. Let us first note that as we have decoupled the simplification level of a bundle S from the view rectangle R our scheme supports both very detailed maps of both small and large areas as well as overview maps that only capture the most important parts of the road network simply by changing the simplification level parameters.

For typical interactive maps however it makes most sense to use coarser levels of detail for larger views while displaying fine details for smaller views aiming to fulfill the consistency properties described in Section 2.6.2 while providing visually pleasing results. Firstly we need to choose the edge selection function σ such that edge paths are always displayed smoothly while keeping their size in check, if the size of edge paths would not be of concern we could simply choose to always unpack edges until no shortcuts remains, however as this is not an option we need to be more clever. If we think about what it means that something is displayed smoothly we realize that smooth displaying entirely depends on the size and pixel density of the display canvas while we need a large number of segments to make a curve look smooth on a large high density display fewer are necessary for smaller displays. It therefore makes sense to define the edge selection criteria so that they depend on the canvas size and pixel density. Looking at the edge selection criteria for a minimum and maximum euclidean edge lengths as described in Section 3.2.2 we see that while they are defined in terms of graph coordinates they will in principle ensure that large edges that can still be unpacked will not simply be included in the edge paths while edges that are small enough will not be unpacked needlessly. To make these parameters dependent on the canvas size and consistent across zooming and panning operations all we need to do is expressing these lengths in terms of canvas coordinates and then transforming them into the respective lengths for the current view by using the coordinate transforms described in Section 3.5. The shortcut bending ratio on the other hand is independent of the view size and we choose it by experimentation, in our testing a value of 0.01 worked well.

While the edge selection function σ influences the size and detail of edge paths it does not change which nodes and consequently their upwards and downwards graphs are added to a particular bundle. This is determined by the minimum priority parameter p and somewhat obviously by the view rectangle. We explore the influence of the priority parameter on bundle size in Section 4.6.3 but it should already be clear that adding more nodes by decreasing p and thus including more lower level nodes and their upwards and downwards graphs can increase bundle size significantly. Additionally choosing higher values of p prunes more of the less important parts of the graph, where importance is measured by the CH level and is thus related to the number of shortest paths using a node. This pruning while not only decreasing bundle size is also decisive for the thinning of the graph creating easier to grasp overviews.

Choosing a good value of p is less obvious than choosing σ as it depends not only on the size of the viewing rectangle but may vary depending on its content, for example a city will need a fair amount of detail to make its overall shape recognizable while out in the countryside less detailed views will still show the recognizable roads while on the other hand from a size point of view the sparsity of the road network may allow to display less important streets without overwhelming the user. This influence of the view rectangle content also means that value of p that might be just right in an area next to a city may create prohibitively large bundles when panning towards the city center. Additionally the influence of the minimum priority p on the map display also depends on details of the CH construction where rendering specific tuning still warrants further research.

One option to deal with this is to choose a fixed target for the number of high nodes $H(R, p)$ such that bundle sizes remain manageable and then letting the backend choose p to best match this target for example by starting at the maximum level and halving it until the number of nodes surpasses the target. The problem with this approach is that the actually chosen value of p will jump when entering or leaving a dense area of the road network such as a city leading to roads suddenly (dis-)appearing. To counteract this effect we supply the p value of the previous bundle extraction when requesting a new bundle and let the backend choose a p value between matching the target number of high nodes and staying consistent with previous bundles we call this the *hinted* mode but also allow specifying arbitrary values of p in *exact* mode or relying completely on a target number of high nodes in *auto* mode.

3.5.3 Evaluation based on Consistency Properties

Looking back at the consistency properties defined in Section 2.6.2 we can conclude the following. Our map rendering scheme fulfills the *importance property* as well as the *connectivity property* by design. The first one is exactly what our node selection process does while the second is a direct consequence of the use of Contraction Hierarchies as nodes are never removed without adding shortcuts for the shortest paths that use them. Higher levels thus always preserve or create all necessary edges to route between all remaining nodes. Viewed differently the upwards then downwards property of shortest paths combined with the fact that bundles and the core graph contain all nodes and edges above any given node ensures that we always have the edges to display a shortest paths between visible nodes.

The *compression property* is also easy to fulfill as choosing a higher value for p and thus a higher threshold on node importance always selects a subset of the nodes selected for a lower value of p when keeping R fixed. Thus when using a larger $R' \supset R$ we can always increase p until the bundle is smaller. By choosing p to match a target number of high nodes $H(R,p)$ we similarly ensure that for larger views we will increase the level of simplification to match similar size though in this case the size remains approximately equal instead of decreasing which is however also allowed under our definition.

Similarly but even more strongly our scheme fulfills the *zoom property*, as we zoom in that is decrease the size of the view rectangle merely holding p constant ensures that edges do not vanish. By increasing p we additionally get more edges and thus show less important road segments.

Finally we can trivially fulfill the *translation property* by keeping p constant during panning, however this somewhat clashes with the scheme of using a target on the number of high nodes as described in the last paragraph of Section 3.5.2. We find that the proposed auto mode is a working compromise that keeps the amount of suddenly (dis-)appearing edges to a minimum though.

3.6 Routing

One of the primary advantages that sets our scheme apart is the ability to not only render a road network from a clearly defined data set but to also find the shortest path between any two nodes in that data set. In this section we will look at the necessary algorithms to make this possible. Let us first describe the type of query we wish to answer and how our answers will look like. Assume we have cached a set of bundles $\mathrm{B} = \{\beta(R_i, S_i, l) | i \in 0 \dots k\}$ and a core graph $G^{core}(l, \sigma) = (V^{core}(l), E^{core}(l), \mathrm{P}^{core}(\sigma))$, which together we may also think of as a single combined graph $G^C = (V^C, E^C, \mathrm{P}^C)$ as discussed in Section 3.4. Given two points in the shared coordinate system of the core and bundles, a start point $p^s = (x_0, y_0)$ and a target point $p_t = (x_1, y_1)$ we search for the shortest path $\pi_{s,t}$ between those two nodes $s, t \in V^C$ which have smallest euclidean distance to the source respectively target point. Finally we wish to display this path with the same facilities used for map rendering.

3.6.1 Bundle and Node Lookup

First however we need to find the nodes s and t and determine for each whether it lies in the core or a bundle and for the latter case also determine said bundles which we will call the source bundle β_s respectively target bundle β_t where either or both may be undefined indicating that its respective node lies in the core. For this we can employ a nearest neighbor lookup data structure for example reusing the earlier discussed range trees or we may also simply use linear search as both the core and the bundles are of manageable size and we can preselect bundles to search by their bounding boxes.

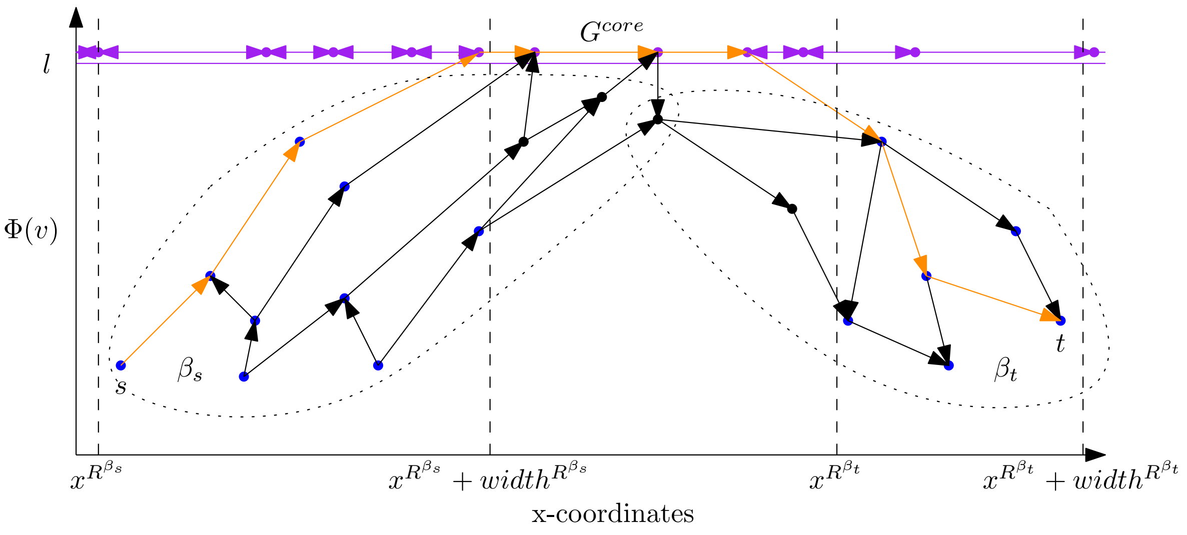


Figure 3.8: A shortest path $\pi_{s,t}$ (orange) spans through the source bundle β_s (only the upwards graph is shown) and a part of the core graph (purple) and the target bundle β_t (only the downwards graph is shown). Blue nodes are part of the respective high node sets and fall within each bundle's view rectangle while black nodes are only added as part of the upwards and downwards graphs. Also note that the core is flattened and can be thought of as lying entirely at level l.

3.6.2 3-Phase Routing

Routing now typically works in 3 phases using the topologically ordered upwards and downwards graphs of the source and target bundle as well as the core graph. Assume all distances except for the forward distance of s and the backward distance of t which are both 0 are set to infinity.

First we scan the upwards graph of β_s relaxing edges in topological order settling the core nodes where the restricted upwards graph enters the core with forward distances $d(s, v)$, then we analogously do a backward scan on the downwards graph of β_t settling those core nodes where the restricted downwards graph meets the core with backward distances $d(v, t)$. Finally we run a Dijkstra search on the core where we prefill the priority queue with the nodes settled by the upwards graph scan and get a candidate for the best cost whenever we take a node from the priority queue that is also settled in the backwards scan i.e. has a finite backward distance. During all of this we keep track of the edges we used by storing at every node which edge was used to compute its current distance value.

If the source node s or the target node t lies in the core graph, we simply skip scanning the respective upwards or downwards graph (or both if s and t lie in the core) and just treat that node as settled with forward or backward distance 0.

3.6.3 Lower Level Merge

The algorithm for 3-phase routing in the previous section assumes that shortest paths always go through the core graph however especially for shorter paths this is not always true. Instead the highest level node of a shortest path in G^+ may still lie below the core graph and so this shortest path would not be found as a candidate in the 3-phase routing algorithm. To also find such paths we need to add a special merge step between scanning the upwards and downwards graphs and doing a Dijkstra on the core. In principle this step simply finds the node v with the smallest sum of forward and backwards distances $d(s, v) + d(v, t)$ contained in both the source and target bundle. This is complicated by the fact however that while node ids are consistent between each bundle and the core graph they are not consistent between bundles (see Section 3.4.1) we therefore store the original node ids with a bundles nodes. One simple algorithm to then find the minimum sum of distances is by using a hash map to associate the distance values with these ids allowing us to compute the distance sums in $O(|V^{\beta_s}| + |V^{\beta_t}|)$ by scanning both bundles, while this has good asymptotic runtime hash maps incur allocation overhead and complicate the implementation.

Instead we opt for a trick, by not only sorting nodes topological but also by their node id we can do the merge as a simple intervened scan of V^{β_s} and V^{β_t} where we go backwards for the target bundle advancing the index into the bundle with lower node id and checking the sum for matching node ids. Interestingly as the graph is already sorted by descending CH level $\Phi(v)$ sorting the node sets by decreasing node ids results in a topological order as well.

If this lower level search finds a node with a finite sum of forward and backwards distance it therefore yields another best cost candidate path. Additionally if all forward settled nodes in the core already have larger forward distances than the entire cost of this candidate path we may skip the Dijkstra search on the core graph entirely as we have already found the overall best cost path.

3.6.4 Backtracking and Drawing the Path

To extract the final best cost path and to display it we now only need to backtrack along the edges stored as leading to a node. In the simplest case where the shortest path lies entirely below the core we simply start at the node with the best sum of forward and backwards distances and separately follow the edges of the downwards and upwards graphs that were used to reach each consecutive node. We gain the information necessary for drawing by collecting the edge path $\rho(e, \sigma)$ for each edge e we visit during backtracking, where we need to append them to the result for the backtracking towards the target and prepend them when backtracking towards the source.

If on the other hand the shortest path uses the core, the node with the minimal sum of forward and backwards distances is a core node where the backwards search on the downwards graph of the target bundle met the core or the target itself. We can thus backtrack the downwards edges towards the target. We then backtrack through the core to find the node where the forwards search on the upwards graph entered the core or the source itself, from which we can then follow the upwards edges towards the source.

3.7 Bundle Caching

Until now we simply treated the set of cached bundles B as an unorganized set of bundles available at the client, now we will look at how we can keep the size of this set bounded while achieving good coverage of the map which allows us to keep large parts available for offline routing and rendering should our connectivity suffer. Caching bundles also allows us to keep more detailed views of areas that have previously been explored available with a higher level of detail which is especially useful when these areas are also those that the user is most interested in.

As the user explores the map in a series of panning and zooming operations we request a new bundle for each operation to make sure that we have the best available data for the current view and the area reachable in a single panning operation (i.e. 9 times the visible area).

After n operation we have requested a series of bundles $\beta_0, \beta_1 \dots \beta_n$ where the view rectangle R_i of the bundle β_i with $i > 0$ is the result of transforming the view rectangle R_{i-1} of the bundle β_{i-1} starting with an initial view rectangle R_0 for β_0. As these operations often change the view only slightly consecutive bundles in the series are often quite similar. This also means that always evicting the oldest bundle from the cache when the new bundle β_{n+1} comes in is not very effective as it is the same as always caching the last k bundles in the series which almost always have considerable overlap.

Let us now consider what a good set of cached bundles B would look like. For one for each pair of bundles $\beta, \beta' \in \mathrm{B}$ their overlap would be minimized though we want to allow some overlap so that bundles that are small sub areas of already cached bundles can also be cached which allows us to deal with zoomed in views also some overlap makes rendering more seamless. Additionally the set of cached bundles would be evenly distributed within the series of requested bundles and therefore within the already explored area of the map.

A naïve caching strategy aiming for such a selection of cached bundles while keeping the fresh bundle β_{n+1} available would always evict the bundle with the largest overlap with the fresh bundle while simply caching every bundle during the startup phase where less than k bundles are in the cache. This largest overlap bundle however is almost always the last bundle β_n which is in the cache because until now it was the most detailed data for the current view and we therefore keep replacing just one bundle in the cache.

To fix this we separate out the latest bundle into a “hot seat” position that is used if no other bundle meets our criteria for eviction. This allows us to state stricter criteria for the bundles in the cache that do not always yield an eviction in favor of the new bundle. In our implementation we use two criteria, for one we do not cache very small bundles and impose a minimum edge count, secondly we only cache a bundle if it has more than some threshold percentage of pair-wise non-overlapping area with all of the bundles already in the cache with values in the $\leq 75\%$ range. To this end we calculate the shared area between bundles and compute its percentage of the area of the larger bundle or 100% if the bundles do not overlap. When a bundle meets the criteria for inclusion it evicts the oldest bundle. During startup the first bundle initializes the hot seat, after that we only add bundles that meet the criteria and start evicting once the cache contains k elements. This caching strategy is illustrated in Figure 3.9.

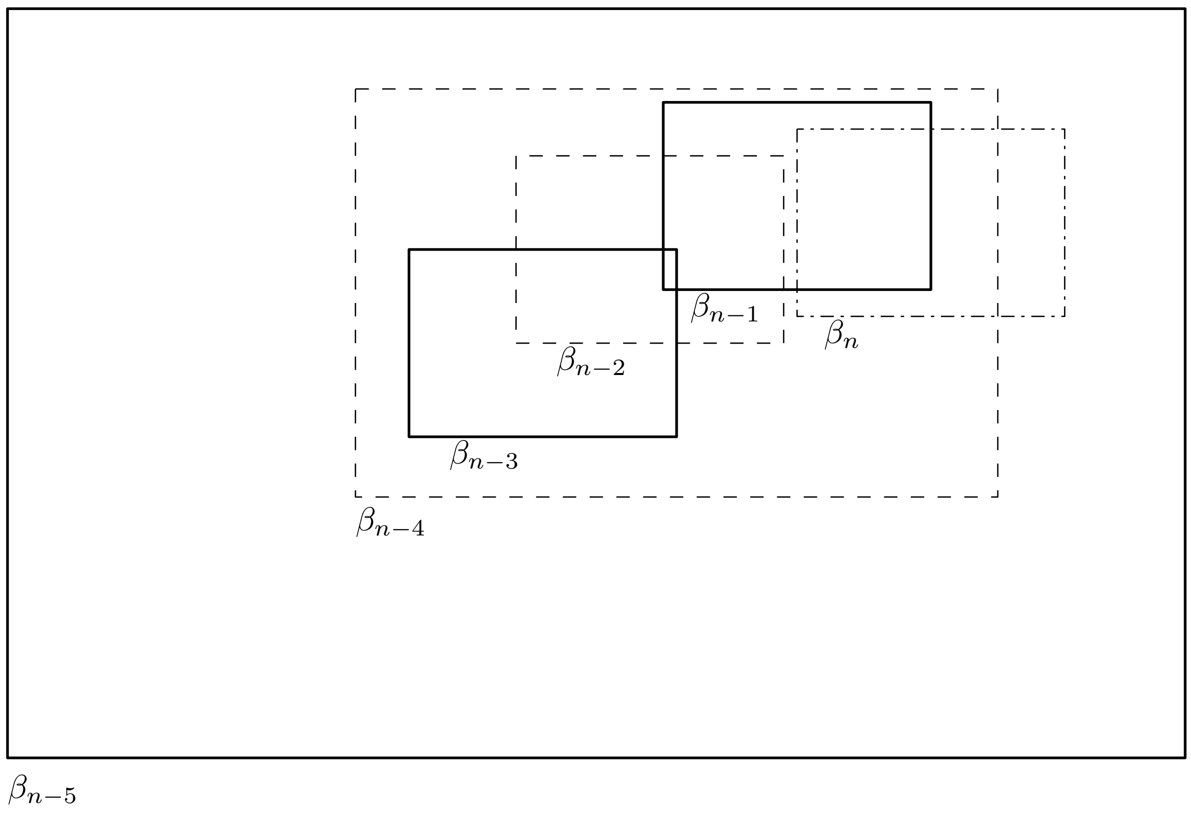


Figure 3.9: At the end of a series of bundles $\ldots\beta_{n-5}\beta_{n-4}\beta_{n-3}\beta_{n-2}\beta_{n-1}\beta_n$ and with a cache of size $k = 3$ and a non-overlap threshold of 75% the bundles with the view rectangles drawn with solid lines are in the cache and bundle β_n is in the "hot seat" as the latest bundle.
Assume that as we enter the shown series β_{n-5} is in the cache and so β_{n-4} is not added as the non overlapping area between it and β_{n-5} is smaller than 75% of the larger one. When β_{n-3} is added it can be cached as it encompasses less than 25% of β_{n-5} and so there is more than 75% of non overlapping area between them. Bundle β_{n-2} on the other hand conflicts with β_{n-3}. Then β_{n-1} again has enough pair-wise non overlapping area to be cached.
Looking at the resulting cache population we can see that β_{n-3} and β_{n-1} is an example for desirable tiling behavior while caching these sub areas of β_{n-5} is an example for the intended zoom behavior where we cache zoomed in views but only if their difference is large enough.

4 Prototype Implementation and Evaluation

To test and evaluate our approach we developed a proof of concept implementation with a focus on proofing the general idea, algorithms and data structures, thus it should be noted that the user interface, drawing performance and visual appearance of individual road segments are not finalized but only adequate for testing and should be interpreted accordingly. To mitigate this we are working on an OpenGL based implementation that uses modern drawing techniques and a 3D globe like view that will also show the applicability of our approach to the direct use of latitude and longitude coordinates.

4.1 Architecture

For our prototype implementation we use a client-server architecture where a powerful backend server holds all graph data in memory, allowing for flexible experimentation independent of efficient storage access strategies as would be needed for a pure client side approach[Büh13].

From this server clients request the initial core graph and subsequently bundles for individual view rectangles. Additionally clients can cache bundles locally, though currently only in RAM, to offer limited functionality during loss of connectivity. While a client that loses its connection to the server can not request further bundles it will still allow routing between and rendering cached bundles.

4.2 Informal Evaluation of Contraction Hierarchy Levels as Importance Metric for Map Rendering

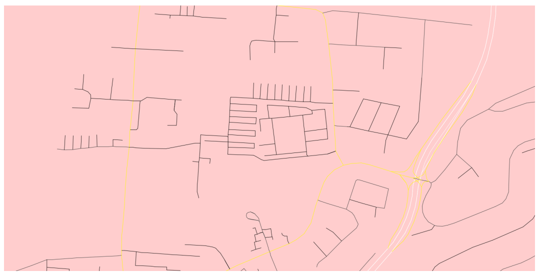
A thorough evaluation of using CH levels as an importance metric for map rendering can only be done after considering changes to the preprocessing step and is out of the scope of this thesis. We will therefore restrict ourselves to an explicitly informal evaluation based on observable anecdotal evidence.

Using a standard CH scheme geared towards simple travel time based routing of cars we can observe the map rendering examples shown in Figure 4.1 and Figure 4.2. These already show a promising level of “natural” simplification, observe how smaller streets get reliably pruned from overviews and how motorways are preserved at state size views. Additionally we can observe that even when only rendering streets the recognizable shapes in cities like Frankfurt and Hamburg are clearly visible in the views covering the city without overwhelming us with small streets.

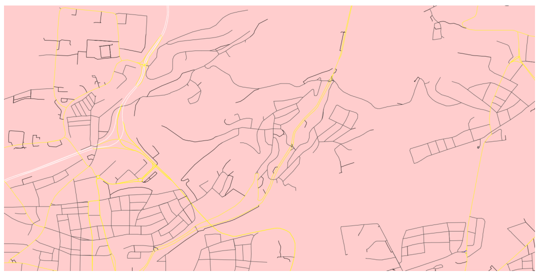
4.2.1 Customizing Maps with Cost Functions

In addition to an informal evaluation on a standard Contraction Hierarchy we tested the effects of some simple changes to the cost function used during CH preprocessing. For our first simple test we limited the maximum speed in travel time computations to 85 km/h to simulate routing geared towards a slow vehicle like the all-electric Renault Twizzy which is still allowed on all road types but does not gain as much from using a motorway as a faster vehicle would.

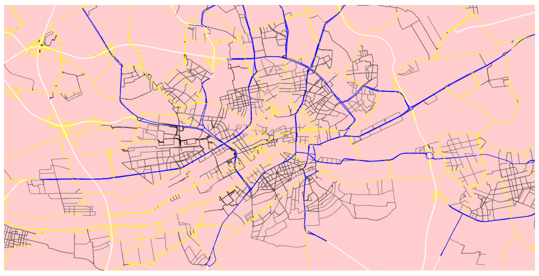
For the second test we used a very crude version of routing for a bike where we increased the cost for all roads allowing a speed above 100 km/h a thousand fold while limiting all max speeds to below 35 km/h. Note that a real routing metric customized for bikes will likely incorporate much finer criteria such as a special cost reduction where bike lanes exist and completely excluding motorways that bikes are not allowed on. The result of these two proof of concept test is shown in Figure 4.3.



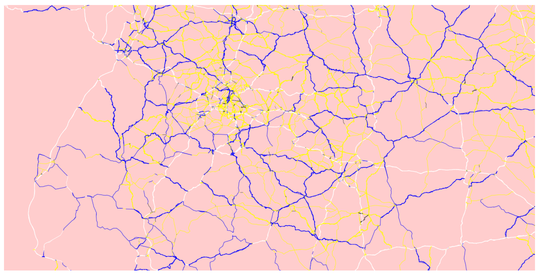
(a) University of Stuttgart



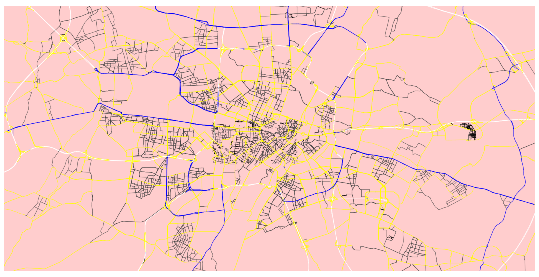
(b) Stuttgart Vaihingen and Kaltental



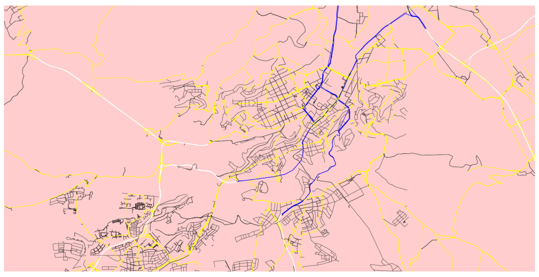
(c) Frankfurt (Main)



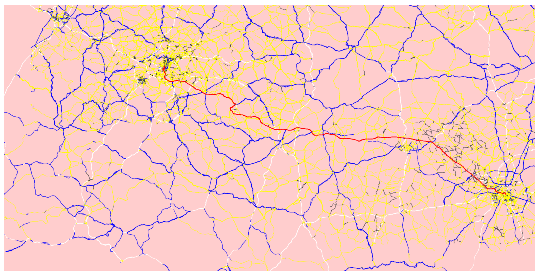
(d) Baden-Württemberg



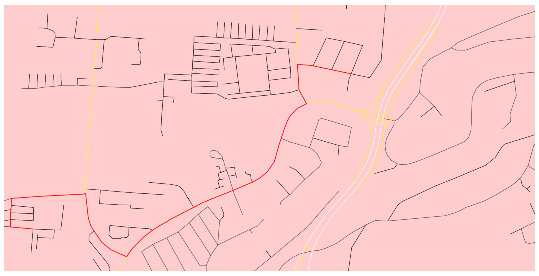
(e) Munich



(f) Stuttgart



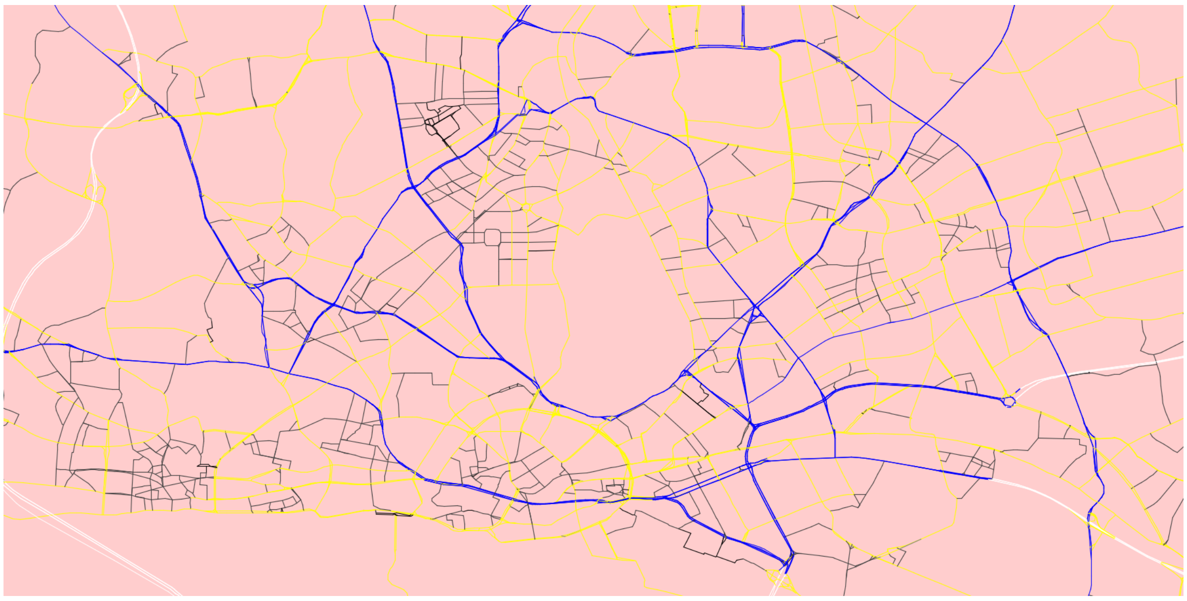
(g) Route from Stuttgart to Munich



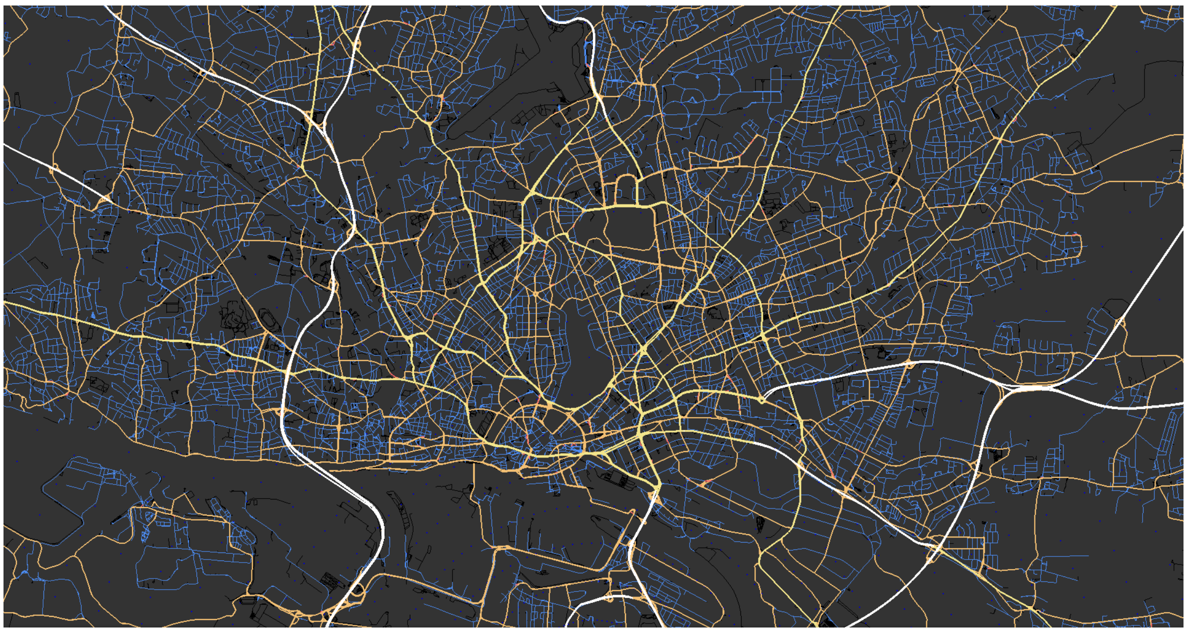
(h) Route across campus

Figure 4.1: A collection of example renderings created using our proposed scheme. Roads are colored according to their maximum average speed and colored from fast to slow in the order white, blue, yellow and black while shortest paths are drawn in red. Also the views of the University of Stuttgart is a zoom in of the upper left corner of the Stuttgart Vaihingen map. Additional impressions from using the prototype clients are available on YouTube[a].

[a] `https://youtu.be/gECqpq7N87E` for the original prototype client and `https://youtu.be/77f-TmyzJ3U` for an early version of OpenGL based rendering.

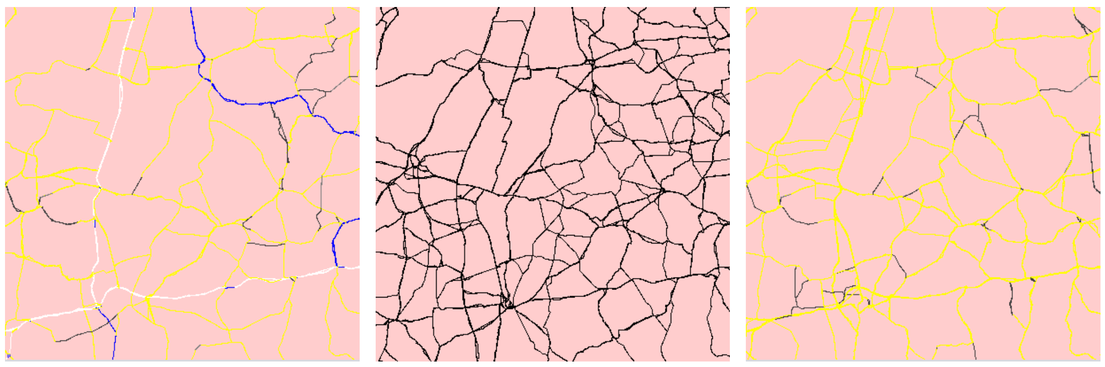


(a) Hamburg



(b) Hamburg Rendered with OpenGL

Figure 4.2: Hamburg with the Alster clearly visible is shown as rendered in the two prototype clients. The rendering in the standard client uses a 2.47 MiB bundle with 2688 nodes, 81,135 edges and 83,774 lines of which 7281 are actually drawn and is thus on the larger side. The discrepancy between lines in the bundle and lines actually drawn comes mainly from requesting 9 times the area to be able to handle panning in all directions.
In the OpenGL based client drawing is significantly faster and we show more of the graph, this is still subject to further experimentation though.



(a) Car **(b)** Bike **(c)** Slow Car

Figure 4.3: Here we see detailed views of an area north west of Munich rendered with different underlying Contraction Hierarchies but using the exact same client configuration. The first one shows the familiar standard map with the fastest roads drawn in white and blue and slower roads drawn in yellow and black. In the bike and slow car examples on the other hand the speed is limited to 35 respectively 85 kilometers per hour thus showing roads in black respectively yellow. This miscoloring could be prevented by deriving the speed of the road independent from the cost metric we currently use (see Section 3.2.2). We can see effects of the change in cost function on the structure of the graph in both examples, generally as the speed of the vehicle decreases so does the importance difference between motorways and slower streets.
In the bike example we can also observe the effect of increasing the weight of fast roads significantly. In this view for example the entire Autobahn (white in the car map) is pruned away.

4.3 Server-Side Components

Our backend server is based on the proven design of the ToureNPlaner Server as originally developed during the ToureNPlaner student project at the University of Stuttgart, an earlier version of which was also used in our previous work[SFS13]. This provided us with robust base functionality using the high performance networking framework Netty[1] together with the serialization framework Jackson[2] for communication and an optimized graph data structure designed to handle large graphs efficiently even in the face of some shortcomings of our implementation language Java such as the lack of structs by relying entirely on arrays of base types primarily ints.

Its concept of pluggable graph algorithms was used to implement two additional algorithms one to extract bundles and one to compute a renderable core graph as described in Section 3.3.

Additionally we augmented our graph data structure with an implementation of Bounding Box Priority Search Trees (BBPST) as described in Section 3.2.1 and precomputed x,y coordinates in a Mercator like projection. As we want to keep our design flexible we create one BBPST to do node selection based on x,y coordinates and another one for latitude and longitude based queries which we also reused to take over the role of doing nearest neighbor lookup for other algorithms freeing up the space used by our older grid based approach.

4.4 Edge Selection Criteria

For both of our prototype clients we use the criteria defined in Section 3.2.2 with values derived from the size of the visible area. In the first prototype client that uses a 2D projection we simply define a maximum and minimum edge length in terms of pixels and use the formula in Section 3.5 to derive edge lengths in the global coordinate system that match the current scaling. In our implementation we chose a minimum length of 10 pixels and a maximum length of 40 pixels and a constant maximum shortcut bending ratio of 0.01. For the core graph which is visible in large and small views we chose fixed edge selection criteria that give a moderate core size around 30 MiB for a core of 1000 nodes while providing fine enough edges even in zoomed in views.

In the OpenGL based client that renders geographical coordinates directly with scaling handled by the hardware we instead chose the minimum and maximum edge lengths as percentages of the diagonal of the view rectangle with a minimum edge length of 0.5% of the diagonal and a maximum length of 2%.

[1]http://netty.io/
[2]https://github.com/FasterXML/jackson

4.5 Core and Bundle Request

In line with the rest of the ToureNPlaner server we use the Jackson serialization framework to encode the core graph as well as bundles for transfer over the ubiquitous HTTP protocol. Using HTTP while incurring some overhead allows us to integrate with almost all programming languages as well as browser based platforms such as JavaScript or the upcoming WebAssembly platform for “almost native” code.

Using the Jackson framework on the other hand gives us a very flexible platform for experimenting with different approaches to encoding bundles and the core graph. While we use the text based JSON format[3] in the following examples and this always remains an option in the implementation Jackson allows us to seamlessly switch to a binary format called Smile[4] using the same code. Note however that even when using JSON we rely on low level serialization primitives to directly read and write into our own data structures instead of going through a document model. For the future we are also looking at more efficient binary serializations such as Protocol Buffers[5] especially since the more binary Smile format is not well supported in C++ while the Jackson framework does also support Protocol Buffers.

4.5.1 Core Graph Requests

As we discussed in Section 3.4.1 our renderable core graph definition is slightly modified to deal with non-contiguous node ids and a core with n nodes is simply defined as the nodes with ids $0 \ldots n-1$ which are the n highest level nodes and the edges between them. This n is the “nodeCount” parameter when requesting a core. Additionally we must define the edge selection criteria for σ, in our implementation these are the minimum and maximum euclidean edge lengths as well as the maximum ratio as defined in Section 3.2.2.

We can therefore extract a renderable core graph by sending a POST request with the parameters encoded as JSON as shown in Listing 4.1 to the URL suffix **/algdrawcore**.

```
{
  "nodeCount": 10,
  "minLen": 40,
  "maxLen": 400,
  "maxRatio": 0.01,
  "coords" : "xy"
}
```

Listing 4.1: Core Request

[3] https://en.wikipedia.org/wiki/JSON
[4] http://wiki.fasterxml.com/SmileFormat
[5] https://github.com/google/protobuf

The result of such a request and thus the serialized core graph is then returned in the format shown in Listing 4.2.

```
{
 "nodeCount": <nodeCount>,
 "edgeCount": <edgeCount>,
 "draw": {
   "vertices": [
     <x0>, <y0>,
     <x1>, <y1>,
     ...
   ],
   "lines": [
     <srcVId0>, <trgtVId0>, <type0>,
     <srcVId1>, <trgtVId1>, <type1>,
     ...
   ]
 },
 "edges" : [
   <srcId0>, <trgtId0>, <cost0>, [<pathEdge0>, <pathEdge1>,...],
   ...
 ]
}
```

Listing 4.2: Core Result

To understand this we may take a look back at Section 3.2.2 and note that the part under “draw” corresponds to the individual segments of edge paths, where “vertices” is a list of x,y points (or latitude, longitude if “coords” is set to “latlon”) and “lines” defines segments by indexing a source vertex, a target vertex and an edge type.

The “edges” field on the other hand is a list of core edges where the source and target ids are those of the combined graph and thus for the core graph directly correspond to their node ids in the original graph. Each edge e has an associated edge path $\rho(e)$ which is encoded as a JSON array directly in the list of edges and its “<pathEdgeX>” values are simply indices into the lines defined in the “draw” section.

4.5.2 Bundle Requests

Bundles are requested in a very similar fashion by sending a POST request of the form shown in Listing 4.3 to the URL suffix **/algbbbundle**. As new fields we have added the view rectangle also known as a bounding box or “bbox” for short and instead of setting a fixed “nodeCount” we now only hint at the number of nodes we want using a “nodeCountHint” field. When in “auto” or “hinted” mode (see Section 3.5.2) the server tries to either directly match our desired number of nodes or balances it with the minimum priority specified while in “exact” mode the “nodeCountHint” field is ignored and we can directly set a minimum priority P.

```
{
  "bbox" : {
    "x" : -431560,
    "y" : -119163,
    "width" : 1700611,
    "height" : 850305
  },
  "nodeCountHint" : 1000,
  "mode" : "hinted",
  "minPrio" : 30,
  "minLen" : 40,
  "maxLen" : 800,
  "maxRatio" : 0.01,
  "coreSize" : 1000,
  "coords" : "xy"
}
```

Listing 4.3: Bundle Request

The serialized bundle that is returned as an HTTP response is again very similar too what we used for the core graph. In fact the drawing data is in the same format and so are edges. However here we distinguish between upwards edges and downwards edges and use a "head" field to bundle their respective edge counts as well as the number of nodes and the priority "level" that was actually used to satisfy our request. We also find a field "oNodeIds" that maps bundle internal node ids (their combined graph id minus the size of the core) to the original node ids which we need for the lower level merge as described in Section 3.6.3.

```
{
  "head" : {
    "nodeCount": <nodeCount>,
    "upEdgeCount": <upEdgeCount>,
    "upEdgeCount": <downEdgeCount>,
    "level" : <level>
  },
  "draw": {
    "vertices": [
      <x0>, <y0>,
      <x1>, <y1>,
      ...
    ],
    "lines": [
      <srcVId0>, <trgtVId0>, <type0>,
      <srcVId1>,<trgtVId1>, <type1>,
      ...
    ]
  },
  "oNodeIds" : [
    <originalNodeId0>,
    <originalNodeId1>,
    ...
  ],
  "upEdges" : [
    <srcId0>, <trgtId0>, <cost0>, [<pathEdge0>, <pathEdge1>,...],
    ...
  ],
  "downEdges" : [
    <srcId0>, <trgtId0>, <cost0>, [<pathEdge0>, <pathEdge1>,...],
    ...
  ]

}
```
Listing 4.4: Bundle Result

4.5.3 Variations Explored

During development we tried several variations of these formats. We started out with all of the data used for drawing contained directly within the edge path as sequences of tuples $(x_i, y_i, x_{i+1}, y_{i+1}, type_i)$ which are easiest to compute during edge path unpacking. This however has the big disadvantage that each point is stored again not only for different edge paths but even in the same edge path. This is easily fixed by instead using a sequence of the form $[x_0, y_0, type_0, x_1, y_1, type_1 \dots x_{n-1}, y_{n-1}]$ but that still leaves points duplicated across edge paths. Thus we moved on to store the coordinate pairs outside the edge paths and using sequences of the form $[vId_0, type_0, vId_1, \dots vId_{n-1}]$ which still allows easy subsampling of the paths by skipping over points. However this was prohibitive as well as when edges are unified this leaves holes in the edge paths when trying to draw just a subset of edge paths such as a route instead of an entire bundle.

To render both single edge paths as well as an entire bundle we need edge paths to be contiguous sequences of road segments and therefore no simple edge unification that would just skip storing an already known edge can be applied. Edge paths will therefore have some overlap as discussed in Section 3.2.2 but by storing only indices into separately stored drawing data the impact of this can be reduced. This could be improved further from a size perspective by unifying runs of segments but we deemed this too computationally intensive considering that edge extraction is already by far the slowest step on the server (see also Section 4.6.4).

4.6 Evaluation

In the following we will look at some rendered maps from our prototype implementations and evaluate its performance data including an analysis of bundle and core size and the performance characteristics of the server side components. Note that these performance results are somewhat preliminary for the overall scheme as we expect additional improvements from changes to the Contraction Hierarchy scheme to make it better applicable for rendering.

4.6.1 Setup

We performed our tests on a server system with 256 GB of RAM and a E5-2630 Xeon CPU with 2.6 GHz and 12 cores plus hyper threading, while this is a relatively powerful system it may well be considered standard server hardware. The client on the other hand is a laptop system with an Intel i7-4750HQ with 2.0 GHz, 4 cores plus hyper threading and 16 GB of RAM.

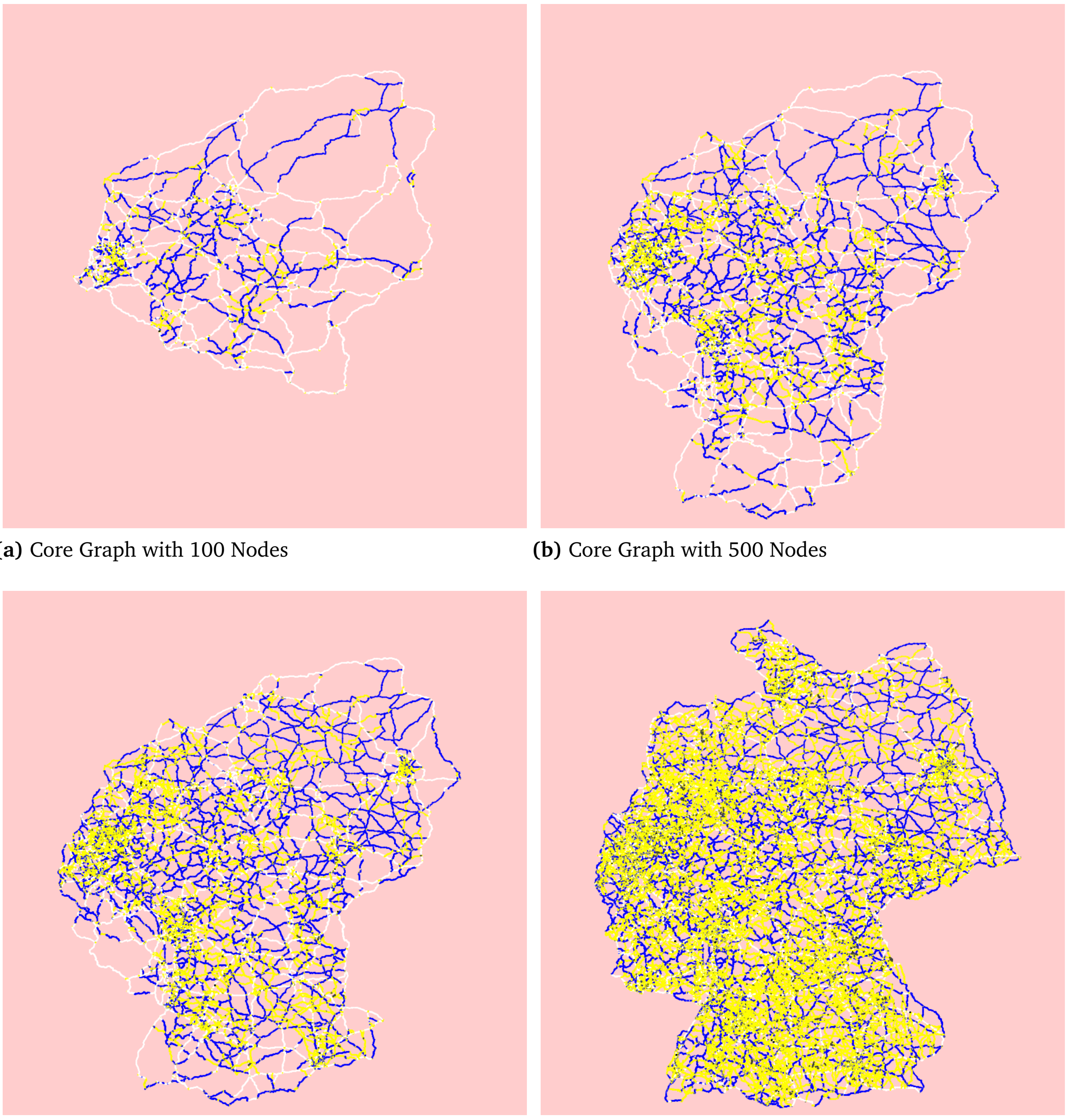
The graph data comprises the road network data of Germany extracted from Open Street Map and consists of 21,721,465 nodes and 44,108,723 edges. Based on this graph we constructed a CH graph which has 80,548,473 edges and of course the same number of nodes.

4.6.2 Core

For the renderable core graph we are mainly interested in two major aspects, we want the core graph to be a good representation of the road network for rendering an overview and we want the core graph to be small at least when serialized for transfer to and storage on the rendering client.

Figure 4.4 looks at some exemplary renderings of core graphs of different sizes and clearly shows that our CH preprocessing loses at least parts of the graphs outline at too low CH levels, something that should definitely be addressed when tuning a CH graph for rendering.

Figure 4.5 then shows the size of the core graph in terms of the size of its serialization and edge count for a varying number of core nodes and explains why both may even temporarily decrease when including more nodes.



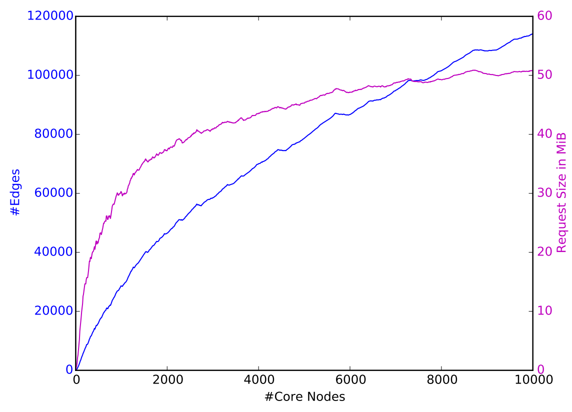
(a) Core Graph with 100 Nodes

(b) Core Graph with 500 Nodes

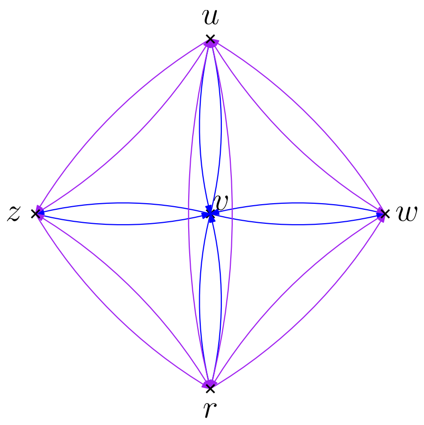
(c) Core Graph with 1000 Nodes

(d) Core Graph with 6000 Nodes

Figure 4.4: Core graphs of varying sizes exhibit the outline losing behavior at higher levels (smaller sizes) but otherwise show a good overview of the graph. With additional tuning of the CH preprocessing however we hope to get much better results at the smaller sizes. On the other hand as the core graph only needs to be updated when the underlying graph changes we may store even larger core graphs and at 6000 nodes and less than 50 MiB of storage we get a very clear outline of Germany even with this unmodified CH preprocessing.



(a) Core Size vs Number of Nodes



(b) Contracting a Node with Degree > 3

Figure 4.5: Looking at the number of edges and the size of the serialized core we can see two effects. Firstly as expected the the size of the core increases as we add more nodes but stays manageable even at $10{,}000$ nodes when compared to the 3.8 GiB for the whole graph. Secondly while the number of edges increases for the most part we see some bumps where the number of edges does not increase or even decreases when including more nodes in the core. This may seem counter intuitive but it is an effect of our core shortcut optimization explained in Section 3.3.1 and the way higher degree nodes are contracted. Figure 4.5b shows an example of contracting the node v of degree 4 where all shortest paths between the nodes u, z, w, r go via v. After its contraction we replace 8 blue edges with 10 purple shortcuts. Now when we include v in the core we do not need to include the purple edges and thus end up with 2 edges less.

4.6.3 Bundles

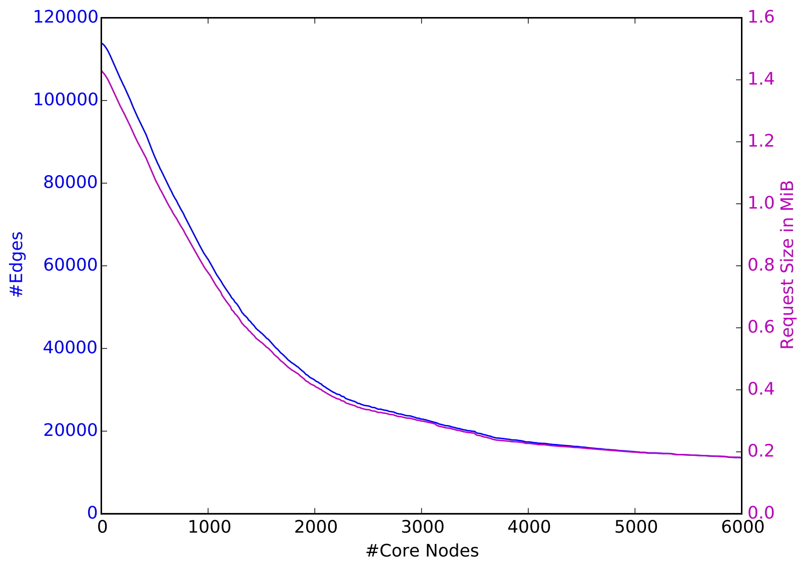
Since the size of bundles is highly dependent on the local density of the graph within a bundles view rectangle it is less clear how to evaluate their size objectively while still being representative of a realistic use case. When examining the performance of shortest path algorithms it is common to do queries between random points even though this often leads to long paths that may not be representative of the most common usage patterns. As measuring more long path queries then the actual usage pattern will for most speed up techniques overestimate runtimes this is often accepted in favor of easier comparability and implementation. With our scheme on the other hand a typical usage pattern will look especially often at cities and thus denser areas of the graph therefore leading to more hard cases. Randomly selected view rectangles would thus likely underestimate the runtimes compared to a typical usage pattern.

Instead we opted for a harder to repeat but more representative evaluation method where we look at the bundle sizes and the server performance during a manual exploration of the map. During this exploration 652 bundles were generated and we covered both overviews, detailed exploration of cities including Berlin, Hamburg and Stuttgart as well as medium zoom panning between them. The core was limited to 1000 nodes which is more on the lower end during this experiment as the drawing performance of the Java based prototype renderer would suffer when using larger core graphs early versions of our OpenGL based renderer on the other hand can easily handle core graphs with several thousand nodes. Similarly we the desired count of high nodes was fixed at 800, note that this is counted before expanding to the upwards/downwards graphs which explains the higher number of nodes in the result. We also turned off bundle caching, or rather used a cache of size 1. The results of this evaluation run can be seen in Figure 4.7

Firstly however Figure 4.6 takes a look at the relation between the size of bundles and the size of the core graph for a bundle with a detailed view of the campus of the University of Stuttgart. We chose a very detailed view because this allows the upwards and downwards graphs to have maximum size.

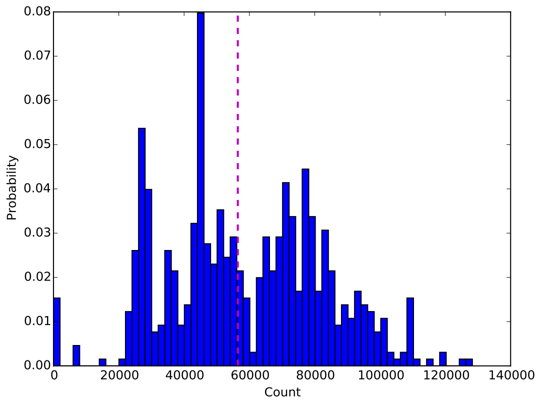
Figure 4.7 gives a detailed overview of the bundle sizes The median bundle for this run had 2883 nodes, $56{,}451$ edges, $54{,}081$ lines and a request size of 1.18 MiB. Note that the median number of edges is slightly above the median number of lines. This is due to the edge unification described in Section 3.2.2 which allows us to have a single line for an upwards and a downwards edge between the same nodes as well as due to keeping lines to a minimum by using edge selection criteria for σ that match the view rectangle.

While the median size of bundles is somewhat higher than desirable this is mainly due to a relatively small core graph. We also see large spikes at around 0.5 MiB which considering that each bundle covers 9 times the size of the area on screen is quite satisfactory. In the future we hope to reduce this size both by using better drawing primitives such as OpenGL that allow for larger core graphs as well as by tuning the CH preprocessing.

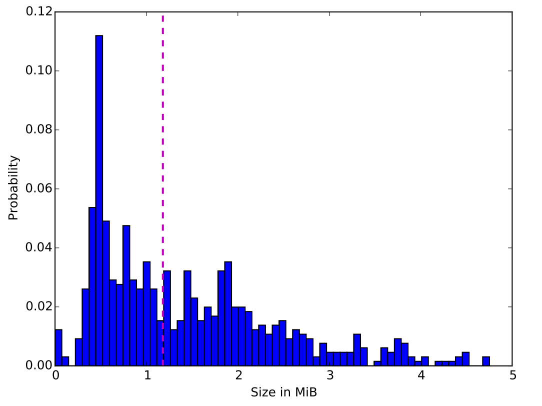


(a) Bundle Size vs Size of the Core Graph

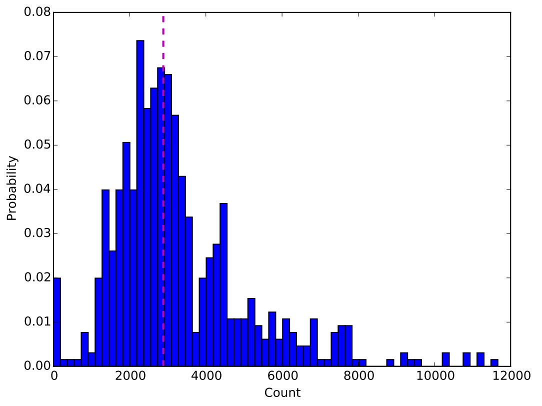
Figure 4.6: The size of bundles rapidly decreases when using a larger core graph as the upwards and downwards graphs get terminated earlier and more locally.



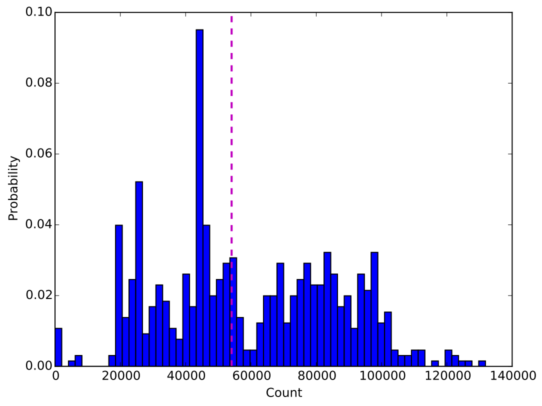
(a) Edges



(b) Request Size



(c) Nodes



(d) Draw Lines

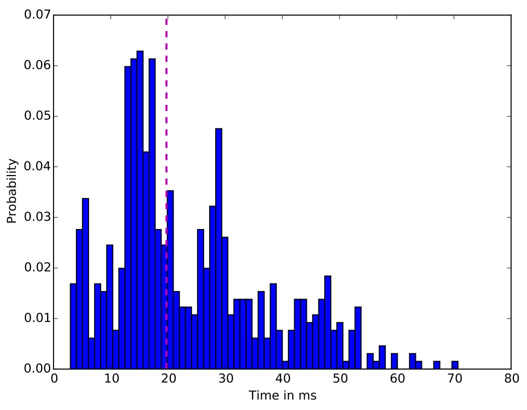
Figure 4.7: We show the bundle sizes as a distribution histogram over the 652 bundles produced during a manual exploration run with the median value indicated by the vertical dashed line.

4.6.4 Server Performance

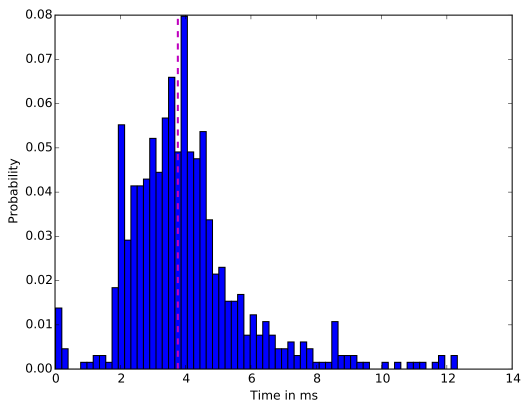
We also measured the performance characteristics of the server side components during the same manual exploration run used for the bundle data. The results as with the previous bundle statistics are again somewhat variable as they also depend on the local density of the graph Figure 4.8 therefore shows the distribution of runtimes as a histogram while Table 4.1 gives a summary of the most important statistics.

	Times in Milliseconds					
Name	Max	Min	Median	Mean	Std. Dev.	% of Total
Find BBox Nodes	70.714	2.944	19.742	23.580	13.615	23.691
Topological Sort	12.321	0.016	3.761	3.999	1.824	4.018
Extract Edges	308.995	0.401	58.621	71.952	51.983	72.292
Overall	346.497	4.347	89.812	99.530	51.786	100.000

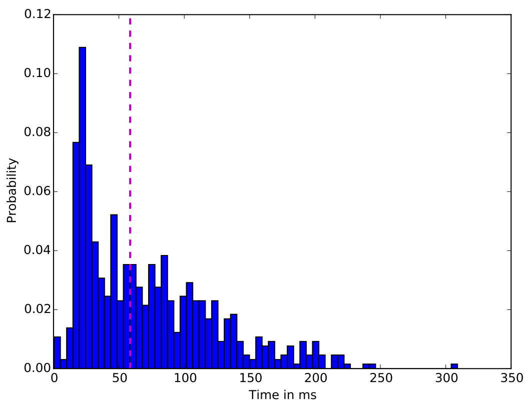
Table 4.1: The server side computations for bundle extraction are dominated by edge extraction that select edges for inclusion in the edge paths according to our edge extraction function σ.



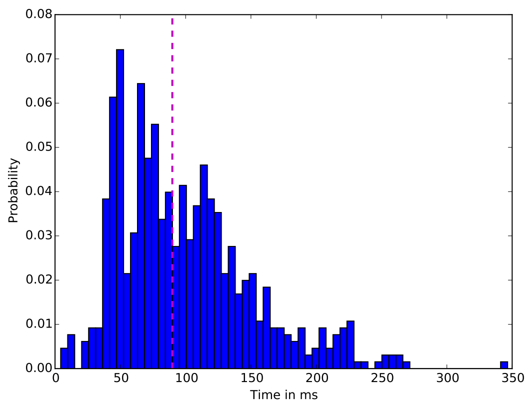
(a) Finding High Priority Nodes $H(R, p)$



(b) Topological Sort



(c) Extract Edges



(d) Overall

Figure 4.8: The performance of the server side algorithms is shown as a distribution of their runtimes with the median marked by the vertical dashed line. Evidently the performance relatively high variability which is mainly due to the fact that the size of bundles directly depends on the local density of the graph within the view rectangle R

4.6.5 Shortest Path Queries

To measure the performance of shortest path queries we use the aforementioned methodology of selecting random source points as well as random target points within the maps bounding box and computing a shortest path between them. Unlike in the previous tests we turned on bundle caching with a cache size of 10 bundles. This ensures that our shortest paths queries are computed also between different bundles instead of only within a single bundle. In total over $47{,}000$ shortest paths were computed and the results are shown in Table 4.2. For this test we can skip the representation as distributional histograms as the all our measurements are tightly centered around the mean which is also evident in the small standard deviations.

The shortest path query performance is dominated by the search for the nodes closest to the query points. This could largely be mitigated by using a nearest neighbor search data structure instead of searching all bundles that contain the query points linearly, however at the cost of more preprocessing and increased memory usage. As these times are still well below the threshold for feeling instantaneous we feel like this would not be worth it.

	Times in Milliseconds					
Name	Max	Min	Median	Mean	Std. Dev.	% of Total
Overall	40.741	11.368	28.147	28.199	0.710	100.000
Finding IDs	35.846	10.283	26.884	26.936	0.574	95.522
Scan Upwards Graph	5.083	0.001	0.438	0.392	0.249	1.392
Scan Downwards Graph	6.626	0.000	0.447	0.396	0.242	1.405
Merge Below Core	12.516	0.000	0.024	0.026	0.065	0.093
Core Dijkstra	9.805	0.000	0.498	0.448	0.225	1.589

Table 4.2: Shortest Path Query Performance

5 Conclusion and Outlook

In this chapter we will first take a look back at the overall contribution of this thesis and the strengths as well as weaknesses of the Unified Rendering and Routing scheme. Then we will take a look forward and explore which directions we deem most promising for further research, discuss the ideas we have for overcoming some remaining weaknesses and hint at some techniques that could be combined with our scheme to widen its scope from only rendering road networks to rendering full maps.

5.1 Conclusion

With Unified Rendering and Routing we have developed and prototyped a scheme that allows us to render and route on a road network from which we only extract simple, relatively small and localized subsets we call bundles as well as a static core graph augmented by rendering information. We also showed that this scheme exhibits some interesting properties such as a guarantee that all road segments that will be used by a shortest path at a particular simplification level are both available and drawable, continuous zooming and panning as well as very efficient routing queries in the sub 100 ms range.

We also developed a client-server based prototype implementation with two clients one using pure Java based 2D drawing with a Mercator like projection and the other utilizing a 3D globe rendered using OpenGL and a C++ code base that directly utilizes geographical coordinates thereby also demonstrating the language independence and flexibility of our backend server.

Finally we evaluated our approach and showed that while the size of bundles is somewhat larger than originally hoped even without optimizations especially in the Contraction Hierarchy preprocessing our scheme is already practical and that the resulting graph simplification yields overview graphs that only include important country roads and highways while finer details such as inner city roads emerge with increasing levels of detail. Furthermore our prototype implementations demonstrate the feasibility of using URAR across a network including wireless links with most of the sub 500 ms latency hidden between consecutive panning and zooming operations. With the use of bundle caching we can even provide useful functionality during network outages and even allow the user to continue with shortest path queries between all available nodes.

5.2 Outlook

Apart from minor missing features and possible improvements to our prototype implementations like binary bundle serialization in the OpenGL client or improvements to the drawing such as rendering roads as more than simple lines there remain some promising avenues for further improving the URAR scheme and the overall state of dynamic map rendering.

As we already hinted at one major area of future research will be the customization of the Contraction Hierarchy preprocessing step for rendering purposes. As Figure 4.4 shows the current untuned preprocessing does not sufficiently preserve the graphs outlines at higher levels. This could be mitigated by pinning specific nodes at the edge of the graph to only be contracted later, such nodes could be chosen heuristically for example by selecting only the nodes on the border that are also part of a high speed motorway. Similarly we may want to prioritize some subset of nodes close to city centers so that important cities are guaranteed to be visible in overviews. Here the flexibility of the CH preprocessing is definitely advantageous and allows for fine tuning which may even be customized for special routing schemes. For example we could generate a custom map for a transportation company that will keep all its storage facilities visible and thus routeable even at high levels.

In addition to these practical improvements further research into the preprocessing will also have to take a closer look at the relation between CH levels and the intuitive notion of road importance outside of the classical routing context. Furthermore it should be investigated whether a tuning of the preprocessing can also help to decrease the size of the upwards and downwards graphs for example by helping to keep them more localized.

Another area that we consider worthy of further consideration is the edge unpacking step. As outlined in Section 4.6.4 this remains the primary bottle neck of server performance while remaining duplication in the edges and edge paths also significantly contributes to the bundle size. In particular we need to look into improving on the need to store edge paths as sequences of single line segments instead of referencing entire subpaths for example by precomputing intersection free edge path segments or by reducing the amount of sharing during CH preprocessing. Yet another approach would try to combine precomputed upwards and downwards graphs of small sets of nodes into new bundles in a way that reduces duplication between them.

In addition to improving the URAR scheme and the CH preprocessing future work will be needed to integrate it with similarly dynamic rendering techniques for geographic outlines such as rivers and lakes, administrative districts as well as dynamic labeling schemes. Working in tandem these techniques will then allow for a completely dynamic, customizable map rendering with integrated routing both online and on cached or explicitly preloaded data sets.

Bibliography

[AFGW10] I. Abraham, A. Fiat, A. V. Goldberg, R. F. Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pp. 782–793. Society for Industrial and Applied Mathematics, 2010. (Cited on page 20)

[AS01] M. Agrawala, C. Stolte. Rendering effective route maps: improving usability through generalization. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pp. 241–249. ACM, 2001. (Cited on page 13)

[BCKO08] M. de Berg, O. Cheong, M. van Kreveld, M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 3rd rev. ed. 2008 edition, 2008. (Cited on page 40)

[BFM09] H. Bast, S. Funke, D. Matijevic. Ultrafast Shortest-Path Queries via Transit Nodes. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 74:175–192, 2009. (Cited on page 19)

[BKS98] M. de Berg, M. van Kreveld, S. Schirra. Topologically correct subdivision simplification using the bandwidth criterion. *Cartography and Geographic Information Systems*, 25(4):243–257, 1998. (Cited on page 31)

[Büh13] S. Bühler. Onboard-Routenplanung auf dem Smartphone. 2013. URL http://elib.uni-stuttgart.de/opus/volltexte/2013/8688/pdf/DIP_3457.pdf. (Cited on pages 13, 19, 28 and 65)

[Def00] D. of Defence. World Geodetic System 1984. 2000. (Cited on page 29)

[DGPW11] D. Delling, A. V. Goldberg, T. Pajor, R. F. Werneck. Customizable Route Planning. *Experimental Algorithms*, 6630:1–12, 2011. doi:10.1007/978-3-642-20662-7_32. URL http://www.springerlink.com/index/R927U847TNQ15476.pdf. (Cited on page 13)

[DGWN10] D. Delling, A. V. Goldberg, R. F. Werneck, A. Nowatzyk. PHAST : Hardware-Accelerated Shortest Path Trees. *2011 IEEE International Parallel Distributed Processing Symposium*, (MSR-TR-2010-125):921–931, 2010. doi:10.1109/IPDPS.2011.89. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6012901. (Cited on pages 19 and 20)

[Dij59] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(0029-599X):269–271, 1959. doi:10.1007/BF01386390. URL http://www.springerlink.com/index/10.1007/BF01386390http://www.springerlink.com/content/uu8608u0u27k7256/. (Cited on page 16)

[DP73] D. H. Douglas, T. K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10(2):112–122, 1973. (Cited on page 30)

[EFS11] J. Eisner, S. Funke, S. Storandt. Optimal Route Planning for Electric Vehicles in Large Networks. In *AAAI*. 2011. (Cited on page 16)

[EM01] R. Estkowski, J. S. Mitchell. Simplifying a polygonal subdivision while keeping it simple. In *Proceedings of the seventeenth annual symposium on Computational geometry*, pp. 40–49. ACM, 2001. (Cited on page 31)

[GH05] A. V. Goldberg, C. Harrelson. Computing the shortest path: A search meets graph theory. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pp. 156–165. Society for Industrial and Applied Mathematics, 2005. (Cited on page 19)

[GKW06] A. V. Goldberg, H. Kaplan, R. F. Werneck. Reach for A* : Shortest Path Algorithms with Preprocessing. 2006. (Cited on pages 19 and 39)

[GSSD08] R. Geisberger, P. Sanders, D. Schultes, D. Delling. Contraction Hierarchies : Faster and Simpler Hierarchical Routing in Road Networks. *Experimental Algorithms*, 2(July):319–333, 2008. URL http://www.springerlink.com/index/j062316602803057.pdf. (Cited on page 19)

[Gut04] R. Gutman. Reach-based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. 2004. (Cited on pages 19 and 40)

[HNR68] P. Hart, N. Nilsson, B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, 1968. doi:10.1109/TSSC.1968.300136. (Cited on page 19)

[MKKV01] N. Mustafa, E. Koutsofios, S. Krishnan, S. Venkatasubramanian. Hardware-assisted view-dependent map simplification. In *Proceedings of the seventeenth annual symposium on Computational geometry*, pp. 50–59. ACM, 2001. (Cited on page 31)

[Pao] T. Paoletti. Leonard Euler's Solution to the Königsberg Bridge Problem. URL http://www.maa.org/press/periodicals/convergence/leonard-eulers-solution-to-the-konigsberg-bridge-problem. (Cited on pages 9 and 10)

[Sch] J. Schwartz. Bing Maps Tile System. URL `https://msdn.microsoft.com/en-us/library/bb259689.aspx`. (Cited on pages 35, 36 and 37)

[Sch13] N. Schnelle. *Distributed Shortest-Path Computation*. Bachelor thesis, Universität Stuttgart, 2013. URL `http://elib.uni-stuttgart.de/opus/volltexte/2013/8297/`. (Cited on page 51)

[SFS13] N. Schnelle, S. Funke, S. Storandt. DORC: Distributed online route computation - Higher throughput, more privacy. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2013 IEEE International Conference on*, pp. 344–347. 2013. doi:10.1109/PerComW.2013.6529512. (Cited on pages 13, 19, 27, 28, 51, 53 and 70)

[SS05] P. Sanders, D. Schultes. Highway hierarchies hasten exact shortest path queries. In *Algorithms–Esa 2005*, pp. 568–579. Springer, 2005. (Cited on page 39)

All links were last followed on September 8, 2015.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature