Institute of Formal Methods in Computer Science
University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelor's Thesis Nr. 20

Distributed Shortest-Path Computation

Niklas Schnelle

Course of Study:	Software Engineering
Examiner:	Prof. Dr. Stefan Funke
Supervisor:	Prof. Dr. Stefan Funke
Commenced:	June 14, 2012
Completed:	December 14, 2012
CR-Classification:	G.2.2

Abstract

We are proposing a technique to distribute the computational load of online route planners like Google/Bing/Nokia Maps between backend servers and clients. We are presenting an implementation integrated in the ToureNPlaner online routing service and its Android client that increases throughput by more than a factor of 8.
Based on Contraction Hierarchies we are decreasing the amount of per request computation on backend systems enabling a leaner and less energy hungry infrastructure to be used that scales with IO instead of CPU power.

Contents

List of Figures

List of Tables

1. Introduction

One of the most ubiquitous applications of computer science in the last decade has been the advent of navigation systems in cars and with the rise of smartphones computer-aided navigation becomes even more ubiquitous.
From a computer science perspective on the other hand, route planning is one of the most direct applications of graph theory and Dijkstra's Algorithm [Dij59], developed in 1959, has provided us with a provably optimal algorithm for general graphs with non negative edge weights. So for a long time speeding up shortest path computations on real road networks was mostly about engineering and hacks, that, in many cases, would not even even preserve correctness but increased performance to a level where it became possible to do routing calculations on small embedded devices directly in the car.
With such devices becoming quite common and the availability of real road network data, research into routing algorithms intensified. It became apparent that the nature of real world road networks with their fast high capacity highways and small low capacity rural roads posed a huge potential for speedup using algorithms that take these special properties into account. This research leads to several different algorithms that all use preprocessing on the road network to make it's aforementioned special structure exploitable during the actual queries that compute a shortest path.
Among them arguably the most important is the development of Contraction Hierarchy based variants of Dijkstra's Algorithm by Geisberger et al [GSSD08].
Based on this work the time needed to compute the distance between two vertices in a graph has been pushed down to the sub-microsecond range and preprocessing can be done in the order of a few minutes [ADGW10].

1.1. Motivation

During the Studienprojekt ToureNPlaner (see http://tourenplaner.github.com) at the Institute of Formal Methods in Computer Science we took some of this research, in particular the use of Contraction Hierarchies, to build a navigation system with a web and an Android based interface. This system is based on a typical client/server architecture as is the case for many modern navigation systems like Google Maps and Android's built in navigation software.
There are backend servers that hold the graph data and run the the actual queries, while clients handle visualization and user interaction. With the backend software developed during this project a single modern computer can handle about 100 queries per second, however even with fast algorithms it requires modern CPUs and creates a lot of computational load which increases energy consumption. Performance is limited by the number of available CPU

cores and the memory bandwidth which are expensive to scale up, especially since using slow, distributed and very power efficient computers like the upcoming ARM or Atom based systems designed for IO-bound web applications query times would times would be increased a lot and thus is not an option.
Another problem encountered with this design with central, powerful computing servers is that it does not fit well with the distributed nature of the Internet. While IO-bound web applications can benefit from Content Delivery Networks that move data closer to the clients, reducing bandwidth on the busy interconnects, this system can not.
These problems - combined with the clients becoming more and more powerful, smartphone CPUs reaching the gigahertz range and web browsers being more of a platfom than merely a view port for HTML - lead to the idea of shifting the computational load from the backend towards the frontend.
However, unlike the navigation systems in cars, modern clients lack cheap storage capacity and are always online. So while we want to keep the client/server architecture so as to be able to store the graph data in the backend, where storage is cheap and updates are easy, we really want to reduce the *computational* load on the backend.
This thesis will explore algorithms and their implementation that try to transform the backend away from centralized computation towards distributed, cacheable storage.

2. Background

2.1. The Problem

Finding the shortest/fastest route between two or more points in a road network can formally be described and solved as a classical optimization problem on a graph. Let $G = (V, E)$ be the graph with V being a finite set of vertices and E being a finite set of edges $E \subset V^2$, then we can define the length/cost of an edge as a function $l : E \mapsto \mathbb{R}$. In road networks we can reasonably assume $\forall e \in E : l(e) \geq 0$ because any time or distance traveled has an associated non-negative cost.
So the problem of finding the shortest route between two nodes can be formulated as finding the path π from s to t which minimizes the sum $\Sigma_{e \in \pi} l(e)$.
The complexity of this algorithm depends to a great extend on the efficiency of the priority queue, with Fibonacci Heaps the overall complexity is $O(n\log n + m)$

2.2. Dijkstra's Algorithm

The classical algorithm for solving the Shortest Path Problem with positive edge costs is Dijkstra's Algorithm [Dij59]. It's a greedy search algorithm sharing its basic structure with Depth-First-Search and Breadth-First-Search, however instead of keeping a simple stack or list of frontier vertices it maintains a priority queue U, ordered by the best known distance to the vertex. Thus, at each step the algorithm selects the nearest unexplored vertex for further exploration.
To achieve this we maintain for each vertex v the minimal distance $d(v)$ from s to v as well as the predecessor edge $p(v)$ that leads to v.
Initially $\forall v \neq s \in V : d(v) = \infty$, $d(s) = 0$, $p(v) = null$ and $U = \{s\}$, then in each step we remove from U the vertex v with minimum $d(v)$ value and for each outgoing edge $e = (v, w) \in E$ we check if $d(v) + l(e) < d(w)$ and if it is, the edge is said to be relaxed and we set $d(w) := d(v) + l(e)$ and $p(w) := e$ also w is added to U. The algorithm terminates when the goal vertex t is removed from U.
When a vertex v is removed from U it is said to be *settled* and its $d(v)$ is final.

2.3. Contraction Hierarchies

2.3.1. Basic Principle

Let us first revisit the basic approach of Contraction Hierarchies as proposed by [GSSD08]. Note that the scheme described here differs slightly from the original approach that uses a total ordering of the vertices instead of merely a partitioning into levels; the principles remain the same however and the use of a less strict ordering enables several additional algorithms including PHAST [DGWN10] as well as the new algorithms described later on in this thesis. At its heart Contraction Hierarchies are a preprocessing scheme that given a graph $G = (V, E)$ produces an overlay graph $G^+ = (V, E^+)$ with an augmented edge set $E^+ \supseteq E$ and an assignment (ordering) $\phi : V \mapsto \mathbb{N}$ of *levels* to nodes with the property that for every shortest path π from s to t in G there exists a representation $\pi' = v_0 v_1 \dots v_k$ with $v_0 = s, v_k = t$ in G^+ such that there exists an index $l \in [0, k]$ with $\phi(v_i) \leq \phi(v_{i+1})$ for all $i \in [0, l-1]$ and $\phi(v_i) \geq \phi(v_{i+1})$ for $i \in [l, k-1]$. Thus the shortest path on the overlay graph breaks down into an upward (increasing level) and a downward (decreasing level) sub path, this is a very important new constraint on shortest paths that (for graphs with low highway dimensions [AFGW]) massively decreases the size of the search space when doing shortest path queries. It's important to note that every shortest path in G^+ maps to exactly one shortest path in G when unpacking shortcut edges.

2.3.2. Preprocessing

First note that the preprocessing step of Contraction Hierarchies in principle works with any ordering of vertices but only a well chosen one prevents the overlay graph from converging towards a complete graph.
The most important and basic operation of Contraction Hierarchy preprocessing is the *contraction of a node*. Given a graph $G = (V, E)$ a node $v \in V$ the goal of the contraction is to remove v and all it's adjacent edges from G without affecting shortest path distances between the remaining nodes of G. This is achieved by adding a so-called *shortcut* edge between each pair of neighbors (u, w) of v iff the only shortest path $u \rightsquigarrow w$ is uvw. If we set the length of the newly-created shortcut edge $e = (u, w)$ to the length of $len(u, v) + len(v, w)$ all shortest paths of which uvw was a subpath keep their length, thus our goal of removing v and it's adjacent edges without affecting shortest path distances is achieved.
To determine whether uvw is in fact the shortest path $u \rightsquigarrow w$ we simply run a Dijkstra starting at u and check whether $dist(w) = len(u, v) + len(v, w)$. Our preprocessing implementation now works in phases to contract all nodes of the graphs until only one node remains. Beginning with $i = 0$ for each phase i, our algorithm computes a maximal independent set I_i, that is a subset $I_i \subset V$ of nodes such that $\forall (u, w) \in I^2$ there is no edge $(u, w) \in E$. All nodes in I_i are then contracted, removed from G and assigned level i in order of increasing importance (measured by their edge-difference and degree) keeping the number of shortcut edges to be added as low as possible. Using an independent set instead of using an ordering over all nodes has the advantage that the contraction step has fewer special cases when compared

to contracting nodes based on a global ordering of nodes as was the case in the original CH scheme.

2.3.3. Algorithms using Contraction Hierarchies

Unlike other speed-up techniques that produce prepossessed data for a specific algorithm, Contraction Hierarchies are simply an augmentation of the original graph with additional, exploitable constraints on shortest paths and can be used by a whole family of algorithms.

Normal Dijkstra

We first note that by ignoring shortcuts a nearly unmodified Dijkstra can be run on the Contraction Hierarchy; though this doesn't lead to any speed-up it can be quite useful when we want to use one and the same graph for Contraction Hierarchy based queries as well as, for example, constrained shortest path queries that use a standard Dijkstra with varying edge weights.

Bidirectional Dijkstra

The classic algorithm, and one of the fastest, to exploit Contraction Hierarchies is a special bidirectional Dijkstra. To exploit Contraction Hierarchies, the forward search from s is restricted to the *upward graph* $G = (V, E^{\uparrow})$ with $E^{\uparrow} = \{(v, w) \in E \cup E^{+} | \phi(v) \leq \phi(w)\}$. The backward search is restricted to the *downward graph* $G = (V, E^{\downarrow})$ with $E^{\downarrow} = \{(v, w) \in E \cup E^{+} | \phi(v) \geq \phi(w)\}$.
As with other variants of bidirectional Dijkstra, the algorithm tracks independent distance labels $d_s(u)$ and $d_t(u)$ for the distances to the source/target. It also keeps track of the vertex v minimizing $\mu = d_s(v) + d_t(v)$; each search can stop when the minimum distance value in its priority queue is at least μ. This is because - as shown in [GSSD08], the maximum level vertex on the $\langle s, t \rangle$ path minimizes $d_s(v) + d_t(v)$ and both searches only look in the direction of increasing level.

Marking Dijkstra

While a bidirectional Dijkstra optimized for Contraction Hierarchies provides very good query times, it is not the only way to speed up shortest path computations CHs.The marking Dijkstra variant, though slower than bidirectional Dijkstra, is still fast and very easy to implement. It can be used for 1-to-1 and 1-to-many shortest path queries simply by only changing the stopping criterion and making sure that the correct edges have been marked.
A lot of the algorithms presented later in this thesis are based on this simple approach and the up-/downgraphs defined here. One very flexible approach that is also very easy to implement and still allows query times in the single digit millisecond range is an even simpler modification of the classical Dijkstra algorithm.

Based on the properties of Contraction Hierarchies it is easy to see that the search space for shortest paths from a source s to some targets $t_1 \dots t_n$ is limited to $G^{\uparrow}(s) \cup G^{\downarrow}(t_1) \cup \dots \cup G^{\downarrow}(t_n)$. Where $G^{\uparrow}(s)$, the *upgraph of* s is the subset of $G^{\uparrow}$ that is reachable from s, and analogous $G^{\downarrow}(t)$, the *downgraph of* t is the subset of $G^{\downarrow}$ reachable from t.
To determine reachability we run a breadth-first-search from s in $G^{\uparrow}$ (t_x in $G^{\downarrow}$), this BFS marks the edges (the search space), so that we can run a standard Dijkstra that ignores all non marked edges from s that will correctly set the labels for $t_1 \dots t_n$ and create a shortest path tree in G^{+} that after unpacking the edges yields the correct shortest path tree in G. Instead of marking both the upgraph of s and the downgraphs of all t we can also only mark the latter and restrict the Dijkstra so that it looks at upward edges and marked edges only.
This algorithm, though slower than bidirectional Dijkstra, is still fast and very easy to implement. It can be used for 1-to-1 and 1-to-many shortest path queries simply by changing the stopping criterion and making sure that the correct edges have been marked.
A lot of the algorithms presented later in this thesis are heavily based on this simple approach and the up-/downgraphs defined here.

3. DORC - Distributed Online Route Computation

3.1. Idea and Motivation

During the Studienprojekt ToureNPlaner we developed a routing server for web- and mobile based clients that among other algorithms can compute shortest $s \rightsquigarrow t$ paths based on a Contraction Hierarchy derived from OpenStreetMap data. For our implementation we used the marking Dijkstra approach described in 2.3.3 because it allowed us to reuse the same code for example as a many-to-many Dijkstra to compute the distance matrix needed to solve (or approximate for larger instances) the traveling salesmen problem. Although this is less efficient than a more optimized bidirectional Dijkstra our server can still perform about 100 shortest path queries per second.
When looking for ways to improve how our algorithms scale, we found that memory bandwidth soon becomes the bottleneck so adding more processor cores does not increase performance (see Figure 5.1) Since adding more servers for computation is expensive and server side computation favors machines with very fast CPUs to keep latencies low, it seems desirable to reduce the burden on servers by moving the work load towards the clients.
Based on the Contraction Hierarchy scheme it is clear that for any given source node s and target node t, even without computing a full Dijkstra, we can find a subgraph, that is a graph $G' = (V', E')$ with $V' \subseteq V, E' \subseteq E$ that captures all the information necessary to compute the shortest $s \rightsquigarrow t$ path. This subgraph can then be transferred to the client which can use it to compute the actual path. We can even restrict the information transferred to abstract graph data and let the client request coordinate data only after it has found the actual path, lookup of this data can then be performed very efficiently on the server as we only need to look up data for known nodes and unpack shortcut edges.
The efficiency of this now depends on three factors.

- How fast the server is able to compute subgraphs, especially how much faster than a full shortest path computation
- How much data needs to be transferred
- How fast the client is able to compute the actual path

3.2. Basic Algorithm

The simplest approach to distributing part of the workload to the clients is by simply transferring the complete upwards and downwards graphs $G^{\uparrow}(s) \cup G^{\downarrow}(t)$ for a given pair of source node s and target node t.

3.2.1. On the Server

To determine the nodes and edges in $G^{\uparrow}(s)$ we use the same technique as in the marking Dijkstra and simply run a breadth-first-search on upwards out-edges starting in s and another breadth-first-search for $G^{\downarrow}(t)$ on downwards in-edges starting at t. For each edge we only need to sent the client a triple of the form $(src, trgt, len)$ with the node ids of the source and target and the length of the edge. Note that we also send shortcuts this way because the client doesn't need to know whether an edge is actually a shortcut, it leaves the unpacking to the server.
Then, after the client has finished computing the shortest path, which thanks to the structure of the Contraction Hierarchy usually consists of only a few dozen shortcut edges, it sends this path as a series of node ids to the server where we can simply go over it and unpack the shortcut edges connecting the nodes, this can (for a limited node degree) be done in linear time and doesn't use complex data structures such as the heap needed for a full Dijkstra.

3.2.2. On the Client

After the client receives the combined graph data of $G^{\uparrow}(s) \cup G^{\downarrow}(t)$ it needs to determine the actual shortest path. Since the client doesn't know about the CH properties of the graph or any other additional data like node coordinates our only option here is a classical Dijkstra. However the combined graph is quite small, about 1823 KB on the Germany graph (see 5.1) and thus even on a mobile phone a normal Dijkstra will run in less than a second. This is despite the fact that we can't use a very efficient graph representation as loading the graph already dominates the computation and the node ids are from a large non continuous range.

3.2.3. How does this Approach Fare

When running the naive algorithm on our Germany graph (see 5.1) the upwards and downwards graphs have on average about 2000 nodes and $40,000$ edges. While directly storing both upwards and downwards graphs for each node would be far less than storing all possible shortest path combinations it is clear that completely static storage would still be unfeasible at least when we want to store all data in memory keeping latencies low.
So instead of storing these graphs, we decided to compute them on demand on the server. At first glance this might seem like a disadvantage even when compared to performing a Dijkstra on the server, however as mentioned earlier, computing the combined upwards and downwards graph $G^{\uparrow}(s) \cup G^{\downarrow}(t)$ for a given s and t only involves running two breadth-first-searches which

only need a queue with $O(1)$ insertion/removal complexity compared to the heap used in Dijkstra which only has $O(log\ n)$ complexity for these operations. This advantage however is offset by the fact that the data we need to transfer is a lot larger than for the resulting path of a full shortest path computation, on our graph about 1823 KB on average compared with 85 KB for a shortest path.
Still, as can be seen in Figure 5.1, we get a slight improvement in performance when running with fast networking.

3.3. Possible Improvements

3.3.1. Pruning

One property of $G^{\uparrow}(s)$ we can exploit to decrease it's size is the fact that any node v that lies on an upwards sub path of any shortest path starting at s will be assigned the same $d(v)$ by a Dijkstra constrained to $G^{\uparrow}(s)$ as by a Dijkstra running on the whole graph G, that is $d(v)$ is set correctly by a Dijkstra constrained to $G^{\uparrow}(s)$.
On the other hand, a v' that lies upwards of s but only on downwards sub paths of shortest paths starting at s can still be part of $G^{\uparrow}(s)$ because $\phi(s) \leq \phi(v')$ but a constrained Dijkstra may assign a higher than optimal $d(v')$ to it. This is because it does not contain the downward edges leading down to v' from the highest level node u (we know it exists, see 2.3.1) on a potentially shorter $s \rightsquigarrow u \rightsquigarrow v'$ path.
Let us first understand why this holds true. It is clear that as v lies in $G^{\uparrow}(s)$ it follows that $\phi(s) \leq \phi(v)$, so for a shortest path $s \rightsquigarrow v \rightsquigarrow z$ where v lies on the upwards sub path all edges on this sub path are upward and reachable from s and thus are part of $G^{\uparrow}(s)$. This means the restricted Dijkstra finds this path and sets $d(v)$ correctly.
Analogously for $G^{\downarrow}(t)$ any node v on a downwards sub path leading to t will be assigned the same $d(v)$ by a Dijkstra running backwards from t both when restricted or running on G.
Thus we can eliminate all nodes v and their adjacent edges from $G^{\uparrow}(s)$ that don't get assigned the same $d(v)$ from a restricted and an unrestricted Dijkstra starting in s; they will never be part of an upwards sub path of any shortest path starting at s. The same of course is true for $G^{\downarrow}(t)$ with Dijkstras running in reverse from t.

3.3.2. Exploiting Commonality between Subgraphs

Tables 3.1 and 3.2 show the distribution of nodes to levels and we can see that over 99% of the nodes fall into the 10 lowest levels. With higher levels the number of nodes decreases rapidly, for example there are only 3160 nodes in all levels >40 combined. This is quite intuitive, as the Contraction Hierarchy is constructed, we remove all newly contracted nodes at the end of each phase. So, as there are fewer nodes overall, we also find fewer independent nodes to contract. Another contributing factor is that our heuristic tries to contract important nodes, that is nodes that lie on many shortest paths, late in the process. This means that these fewer nodes introduce comparatively many shortcuts making the upper levels highly connected even

Levels	# Nodes	# Higher nodes (including)	Percent of total
[0-10)	16117470	16271859	99.051
[10-20)	134981	154389	0.829
[20-30)	12877	19408	0.079
[30-40)	3371	6531	0.020
[40-50)	1371	3160	0.008
[50-60)	697	1789	0.004
[60-70)	383	1092	0.002
[70-80)	233	709	0.001
[80-90)	150	476	9.218E-4
[90-100)	100	326	6.145E-4
[100-110)	68	226	4.178E-4
[110-120)	55	158	3.380E-4
[120-130)	36	103	2.212E-4
[130-140)	26	67	1.597E-4
[140-150)	20	41	1.229E-4
[150-160)	12	21	7.374E-5
[160-169)	9	9	5.531E-5

Table 3.1.: Distribution of nodes to levels for Germany (see 5.1)

Level	# Nodes	# Higher nodes (including)	Percent of total
0	7280318	16271859	44.741
1	3870914	8991541	23.789
2	2039810	5120627	12.535
3	1367127	3080817	8.401
4	703234	1713690	4.321
5	380902	1010456	2.340
6	215451	629554	1.324
7	127676	414103	0.784
8	79257	286427	0.487
9	52781	207170	0.324
10	37287	154389	0.229
11	26756	117102	0.164
12	19505	90346	0.119
13	14394	70841	0.088
14	10711	56447	0.065
15	8138	45736	0.050

Table 3.2.: Detailed distribution of nodes to $levels \leq 15$ for Germany (see 5.1)

approaching a complete graph.
For the upwards and downwards graphs this density and the overall low number of nodes on higher levels of the CH lead to considerable overlap between subgraphs of different nodes. This is clear as there are n (n being the number of nodes in the graph) different upwards/downwards graphs and considerably fewer nodes on higher levels, thus these nodes have to be shared between subgraphs.
One approach to exploit this commonality would be to store for each node a reference to all nodes in its upwards/downwards graph that fall into some range of levels and then combine several such level range based subgraphs into the response. In fact because these subgraphs are disjoint, we could even store them compressed and use the property of most compression schemes to allow a simple concatenating of their output to send a completely compressed response without unpacking the stored subgraphs.
The problem with this approach is that we would still send the complete up-/downgraph for each request which means we need a lot of bandwidth and this would then become our bottleneck.

3.4. DORC + CORE Less Bandwidth More Throughput

As we have seen earlier (see section 3.3.2) there is considerable overlap between the upwards and downwards graphs and we could exploit this commonality to make it easier and faster to construct a complete upwards/downwards graph for a given node.
However it turns out that we can exploit this commonality even more easily and also save a lot of bandwidth by not having to transfer complete up-/downgraphs for each request. Instead of storing a band of a node specific up-/downgraph we can simply store all nodes and edges above some level bound as its own graph so that we only need to transfer ir to the client exactly *once*. We call this graph the CORE and it is defined for a given overlay graph G^+ and level bound l as $G^{\text{CORE}}(l) = (V^{\text{CORE}}(l), E^{\text{CORE}}(l))$ with nodes $V^{\text{CORE}}(l) = \{v \in V | \phi(v) \geq l\}$ and the edges defined as $E^{\text{CORE}}(l) = \{(v, w) \in E^+ | v \in V^{\text{CORE}}(l) \wedge w \in V^{\text{CORE}}(l)\}$.
Additionally we can prune all shortcut edges that contract a high level node $v \in V^{\text{CORE}}$ because the edges it shortcuts are already present in the CORE. With this CORE graph present on the client we only need to transfer the node specific portion of the upwards/downwards graphs *up to* the level bound l. Table 3.3 shows the size of this CORE graph for different level bounds, which even for bounds as low as level 40 is small enough to be practical even in a mobile setting. Since the CORE graph is completely static and quite compressible when stored in text form, we can further cut down on the cost of transfer by keeping a compressed copy of the CORE available on the server which clients can use to update their locally stored copy when necessary.

Level Bound	# Nodes	# Edges (pruned)	# Edges	Size (KB)	Compressed (KB)
20	19408	216900	562404	5620	1348
30	6531	128083	277978	3356	764
40	3160	82514	161153	2180	480
50	1789	54630	101591	1452	312
60	1092	39017	66394	1044	220
70	709	28695	44775	772	160
80	476	18698	30036	504	104
90	326	13800	20445	376	76
100	226	10443	13455	284	60
110	158	7100	8667	196	40
120	103	4542	4858	124	24
130	67	2491	2605	68	16
140	41	1135	1145	32	8

Table 3.3.: CORE graph sizes for Germany (see 5.1). We use the JSON based format used by the Android client for size comparison and bzip2 as compressor.

4. Our Implementation

4.1. Preprocessing the CORE

Preprocessing of the CORE is as easy as doing a sweep over all nodes and edges and writing out all nodes with a higher level than our *level bound*, as well as all edges that connect two of these high level nodes. To make the CORE even smaller, we prune all shortcut edges where the shortcut node is also above the level bound. We can do this because both skipped edges are already included in the core, thus we will not lose any of the shortest paths.
As can be seen in table 3.3 this roughly halves our space consumption.

4.2. On the Server

The server side portion of DORC is integrated directly with the existing ToureNPlaner server application. Because of its flexible design we could easily add two additional algorithms to support DORC.
The first algorithm we need to support DORC is a breadth-first-search based algorithm that takes the source node s and target node t for a DORC request and computes $G^{\uparrow}(s) \cup G^{\downarrow}(t)$ with the additional constraint that the requesting client can supply a maximum level for exploration so that we can stop the searches at nodes of this level and the client can then use the static CORE graph to construct the whole union graph as described in section 3.3.2. The only dynamic data structures we need for this is a bit vector to store which node was already visited by the search and a queue that holds edge ids of all edges in the combined upwards and downwards graph.
The performance of this algorithm is the main deciding factor in the performance of DORC as a whole and it is evaluated in detail in chapter 5.

4.2.1. Encoding Subgraphs in JSON/Smile

As mentioned in Section 3.2.1 for every edge in the upwards and downwards graphs we only need to transfer its source, target and length. However because the client is normally not aware of the node ids of the queried source and target, so for the client side shortest path computation to work we need to transfer these as well.
To integrate as closely as possible with the existing infrastructure we use the same data format as the rest of the ToureNPlaner system. In fact the requests for DORC look exactly the same as the requests for a normal shortest path with the maximum exploration level supplied as

a constraint in the same way as the altitude difference for our Constrained Shortest Path implementation.
This means all requests and responses need to be encoded as JSON which in turn might be transferred in the binary but equivalent Smile format supported by the Jackson JSON library[1].
In the end the format for the response containing the graph data looks something like this $\{graph : [src1, trgt1, dist1, src2, trgt2, dist2, \ldots], source : srcId, target : trgtId\}$ so instead of encoding each edge as a triple we save the space for the delimiters and use the fact that each edge is encoded by 3 integer values.

4.2.2. Unfolding the complete path

The second algorithm implemented in the server is used to convert a list of node ids into a complete path. As mentioned earlier this is very fast because the unpacked path calculated by the client is very short (on the order of a dozen nodes) and finding edges and extracting shortcuts is extremely fast.
The response of this algorithm in turn is of exactly the same format as used by normal shortest path requests which allows us to reuse the path drawing infrastructure on the client. It's important to note here, that this step may be combined with further information about the route such as street names and turn by turn directions which could again keep DORC in line with the classical server side algorithms when these features get added there.

4.3. On the Client

To show that the DORC scheme actually pays off, we have incorporated DORC into our existing ToureNPlaner Android client that was developed as a student project at the University of Stuttgart. We were able to implement our scheme with only modest changes to the apps core architecture and kept all of its previous functionality intact. The enhanced version of our ToureNPlaner app is available on Google Play.

4.3.1. The Graph Representation

The most important additional software component on the client is a light weight graph data structure. Unlike typical high performance graph representations as used in our server (e.g. offset array based), this one has to cope with node/edge id's that are distributed over a large non-contiguous range, which makes it impossible to use them as index into an array. This problem is amplified by the fact that we need to keep the exact id values, at least for the nodes, to be able to link them back to the corresponding items on the server.
Therefore we used a hash based data structure so we can address nodes by the same id as on the server, this also enables us to transfer subsets of the server's graph as simple lists of

[1] see http://jackson.codehaus.org/

the aforementioned triples. Building on this principle, we developed a graph representation that can be initialized with the CORE and stored in the same JSON based format as used for the graph data supplied by the server. When running a request the client then receives the additional graph data from the server and augments its own graph with it. After a request, the client can easily discard parts or all of the augmentation. This augmentation is implemented by the concept of a parent graph. When creating a new graph data structure we supply a parent graph (the empty Null Graph for the CORE) and when a node/edge is not found in a graph we try looking it up in its parent. Thus we can overlay the CORE with the upwards and downwards graphs from the request and could even add additional graph data for example to be able to compute a 1-to-many shortest path.
This also allows us to keep the CORE graph loaded between requests without any program logic involved in removing the request specific graph data. Even though hash table based approaches have $O(1)$ lookup from a theoretical point of view, there is of course an additional overhead involved. However, since DORC keeps the size of the client graph quite low and because we use hash tables from the HPPC project (http://labs.carrotsearch.com/hppc.html)that use ints directly instead of the normal object based hash maps Java offers, performance is still fast enough to provide a good user experience.

4.3.2. Integration with the Existing Client

To integrate closely with the existing Android client and to be able to keep all of its functionality intact, we integrated the DORC code with the infrastructure used to perform server side queries. Instead of sending one shortest path request to the server and using its result to display the route, we used the asynchronous request handler to first send a request for the combined up- and downgraph up to the level for which we have a static CORE graph, combine the result with it and compute a shortest path, then to use the path lookup on the server, which results in a response containing the exact path which is now compatible with the rest of the client.
So the only point where our additional code interfaces with the original functionality is where we need to create a client side compute handler instead of a normal request handler when DORC is chosen as the algorithm to be run.

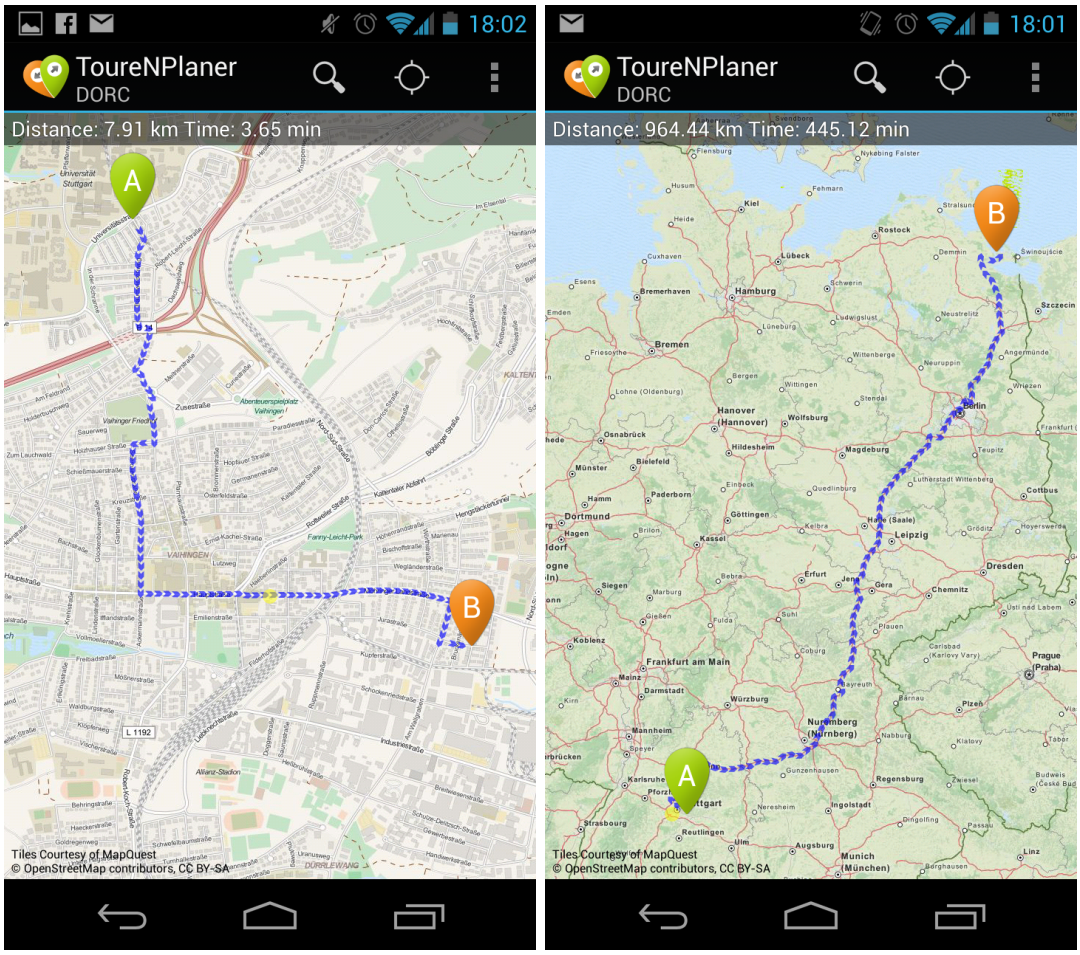

Figure 4.1.: Android client in DORC mode

5. Experimental Results

5.1. Road Network Data

Our experiments are based on the road network of Germany extracted from publicly available OSM data. The respective graph has $16,271,859$ and $62,062727$ edges (when augmented with CH shortcuts); as edge costs we used travel times based on the respective road categories.

5.2. Throughput Analysis

5.2.1. Testing Environment

All measurements have been performed on typical server hardware with relatively low single core performance. We used a dual socket motherboard with 16 GB RAM and two AMD Opteron 6128 with a total of 16 cores clocked at 2.0 Ghz. On the software side we used Arch Linux with Kernel version ≥ 3.6 and 64 bit Java 7 (OpenJDK 64-Bit Server VM build 23.2-b09).

5.2.2. Methodology

Unless otherwise noted, all measurements have been performed with a special testing client over *loopback* networking on the same system. This client can also be found on our Github page. It uses the same HTTP based interface as our Android prototype but adds the ability to issue requests in parallel, creating requests for coordinates randomly distributed over the map data and acquiring timing and throughput information.

5.2.3. Measurements and Analysis

Figure 5.1 shows the performance in requests per second of DORC while sending $G_s^{up} \cup G_t^{down}$ for each request and how it scales in the number of threads; however due to the nature of the Java Runtime Environment other cores may be used for garbage collection etc. To put the data in perspective, we also show the current server side shortest path implementation as a baseline.

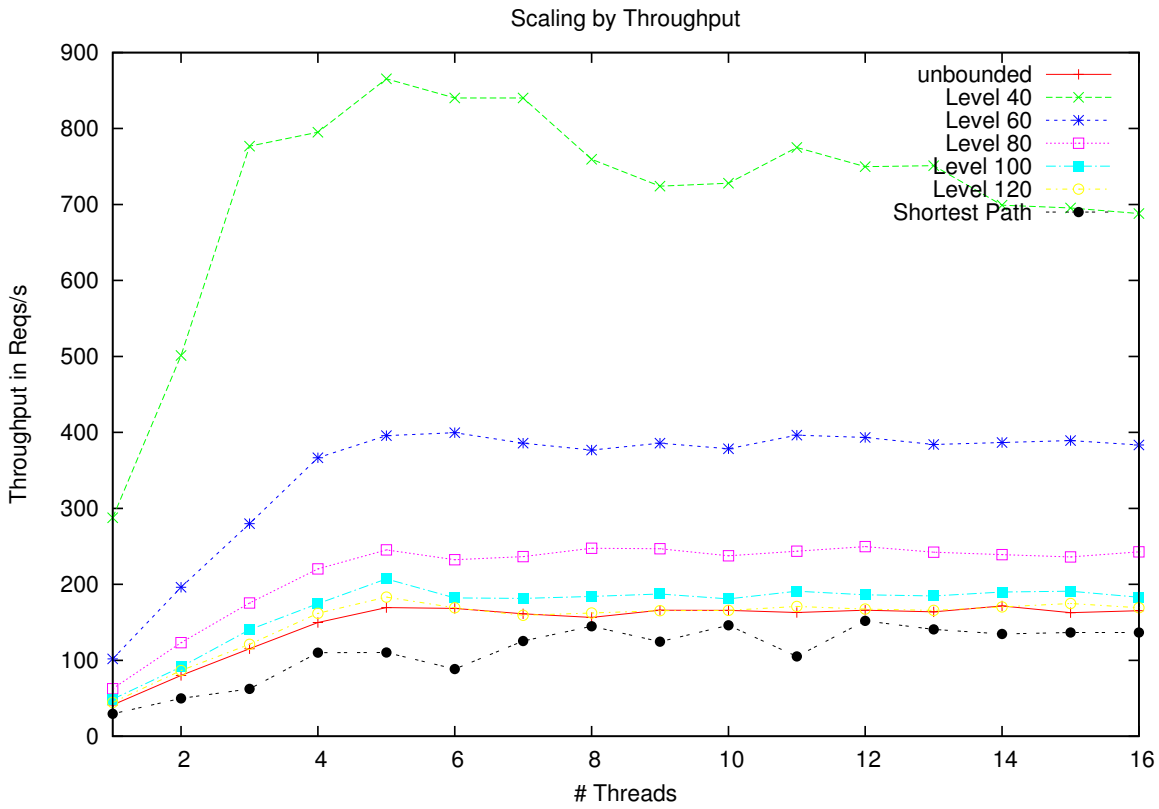

Figure 5.1.: DORC throughput dependent on different CORE parameters. See A.1 for the raw data. We can see higher throughput than classical Dijkstra even without a bound on the level

We see that even without making use of a CORE graph, throughput is improved compared to pure server side computation. For few cores, we also observe linear speed up, using more than 4-5 cores does not pay off probably as the memory bandwidth forms a bottleneck. It must also be noted that the system used has a NUMA architecture and performance suffers when accessing the graph data from cores further away from the memory bank holding it. Employing a CORE graph yields considerably higher throughput, e.g. when using a CORE graph above level 40, a single server can handle close to 900 requests/second.

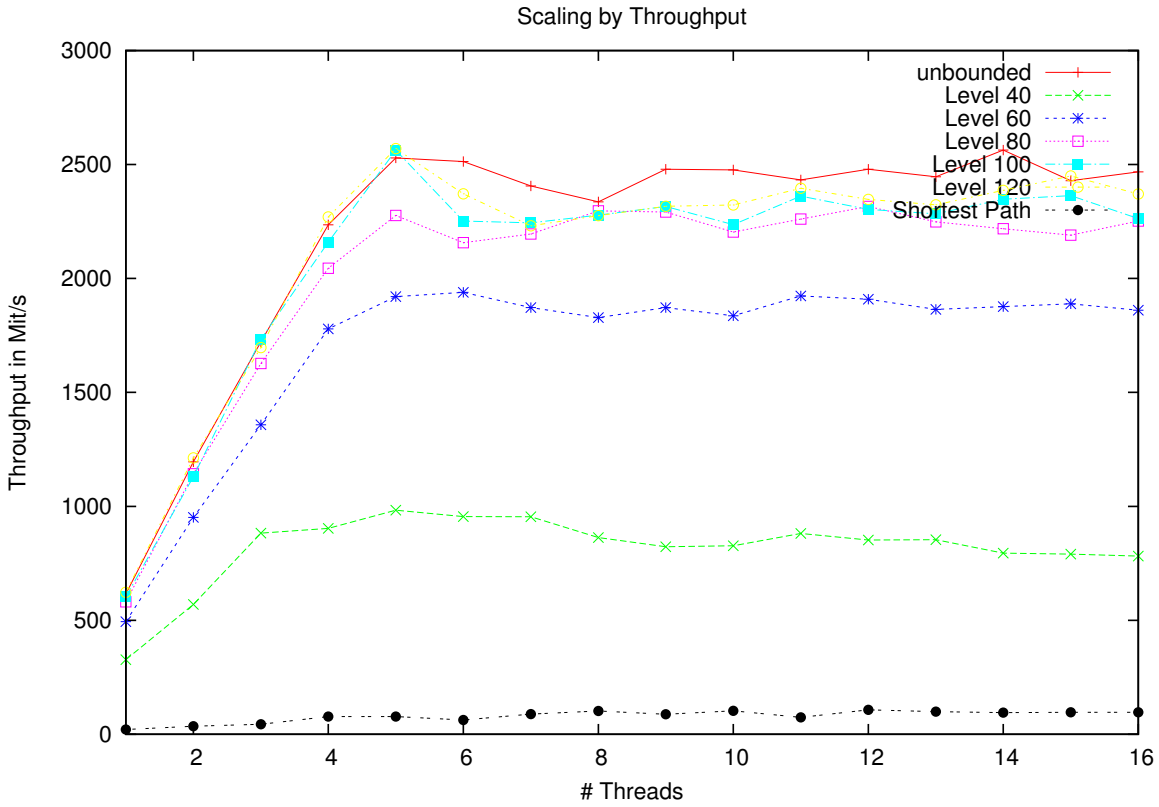

Figure 5.2.: DORC network throughput dependent on different CORE parameters. See A.1 for the raw data. We can see higher throughput than classical Dijkstra even without a bound on the level and throughput considerably higher than gigabit Ethernet

Figure 5.2 additionally shows the corresponding network throughput in Mbit/s. As the numbers clearly show DORC can saturate a full gigabit Ethernet link even for mid-range levels, for example 40, where there are many distinct requests and each response is small (see Figure 5.3).

Figure 5.3 depicts (for *one single thread* the correlation of the size of the graphs transmitted and the throughput versus the choice of the level parameter for the CORE graph. As expected, graph sizes increase dramatically with growing level parameter whereas throughput improves drastically.

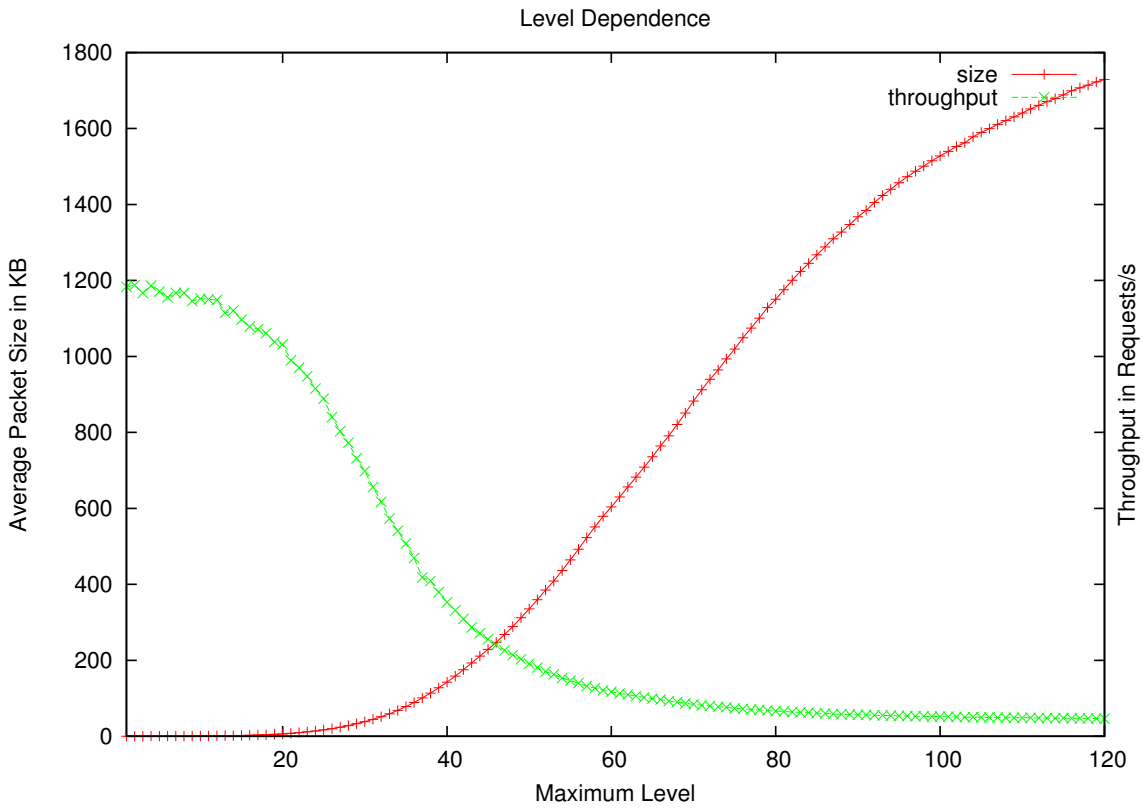

Figure 5.3.: Average result sizes and throughput with different level parameters for the CORE graph; *one single* thread. We can see that throughput increases dramatically for lower level parameters

5.3. Client Side Performance

Finally we provide some performance data on the prototype Android client. As the capabilities of mobile devices vary broadly, we can focus only on one device which we consider 'standard' by today's standards. Our measurements were taken on a Galaxy Nexus with a 1.2 GHz ARM Cortex-A9 processor running Android 4.0. On the other hand, the performance on the client does not have an impact on the overall throughput, number of serviceable clients or cost of the entire system; its performance matters mostly in terms of usability. If a mobile app can compute and display a route fast enough to make it feel right for the user there is not really any point in putting a lot of effort into scraping of another millisecond. The transfer of the CORE graph with level parameter 40 takes about 1.4 seconds (this has to be done only *once*!). Adding the upwards and downwards graphs for source and target takes another 115 milliseconds. The time for actual Dijkstra computation on the augmented graph depends on the distance of the path to be computed, but was measured at about 500 milliseconds for a cross-country route in Germany.

6. Conclusion and Outlook

6.1. Conclusion

Looking back at the development of DORC we can be quite proud of our increase in throughput and the way it integrates seamlessly with the existing infrastructure. We have developed a scheme that builds on existing technology and concepts, namely Contraction Hierarchies, and puts their properties to use at distributing work away from the easily overloaded backend.
While we have not reached the goal of making routing completely IO bound and ready for the cloud, yet we have developed a scheme that pushes forward in exactly this direction and offers further opportunities to achieve this goal (see 6.2.3).
One interesting byproduct of our development is the advanced exploration of more flexible data structures for graphs that enable us to overlay different subgraphs and generally deal with arbitrary parts of a larger graph in a more flexible way. While our current implementation is quite simple and does not perform well for large graphs, it offers an extensible API that enables us to combine high performance graph data structures with very flexible smaller graphs yo provide a single overlay.

6.2. Outlook

6.2.1. Developing a Web based DORC Client

One feature of our online route service that is not currently able to make use of DORC is the JavaScript based browser client. In principle it should be easy to adapt DORC to work in this environment; however it is not yet clear how the need to transfer the CORE to the client would fare in this setting where sessions are frequently reloaded. One idea needing investigation here is the use of browser local storage to store the CORE graph beyond the lifetime of a session. However this mechanism has only been available in recent versions of modern browsers as its part of the upcoming HTML5 standard.

6.2.2. Extending DORC to other Algorithms

As described in this thesis DORC only speeds up $s \rightsquigarrow t$ single source, single target shortest path computations; however as mentioned earlier, the current ToureNPlaner online routing service already offers several other algorithms like an approximation for the Traveling Salesperson Problem.

So it seems only logical that we need to extend the principles used in DORC to distribute at least part of the load of these algorithms between the backend servers and clients. This is particularly interesting as these algorithms create a much higher load than simple shortest path queries, thus the potential gain of offloading this is quite high.
In fact, to some extent DORC already implements most features to realize at least part of this, so for example one could let the client request upwards/downwards graphs for several nodes and then compute the distance matrix between them by running a Dijkstra for each node on their union with the CORE graph. However this approach has the problem that it would be far slower to compute a TSP approximation for this distance matrix on the slow mobile clients than on the current hardware of our backend servers.
Another problem we want to tackle is the need to update CORE graphs with every change of the graph data used on the backend. While currently this happens less frequently, some of the algorithms already being planned - like special routing for other kinds of transportation or routing data that incorporates traffic data - will induce more frequent graph changes. So we are investigating update strategies that would allow only parts of the CORE graph to be updated or a more general CORE to be used that allows the same data structure to be reused for different schemes.

6.2.3. Towards Completely Static Routing

Looking further in the future, it is clear that we want to make DORC even less computationally demanding on the backend aiming for a completely static solution where a request for an up-/downwgraph leads to a simple lookup of precompressed data instead of a breadth-first search on a graph in memory. Since our current approach already fills up > 1 GBit/s of network bandwidth this compression is the key to increase throughput.
Additionally we would like to be able to use much less powerful backend servers like the upcoming ARM based server systems or cheap cloud instances with far less than 8 GB of RAM.
One possible direction worth investigating here is to store precompressed upwards/downwards graphs directly in the filesystem of backend servers and to use a central lookup server to redirect clients to the server holding the up-/downgraph of a particular node.
An interesting property of this approach is that - thanks to the built in redirect capability of the HTTP protocol that underlies the current ToureNPlaner implementation - this redirecting could be done without any (or only modest) changes to the client. This approach would also allow us to use standard web servers like nginx (http://nginx.org/) that have been well tuned towards the delivery of static data.

A. Appendix

A.1. Raw Performance Data

This is the raw performance data used to generate figure 5.1. Each individual line was computed from 320 requests with a concurrency of 16 parallel connections, where duration and size values are averaged.

Table A.1.: Raw data of Figure 5.1

Algorithm	# Threads	Duration (ms)	Size (kb)	Req/s	Mbit/s	Level
	2	199.83	1823.04	80.06	1195.75	unbounded
	3	138.79	1823.04	115.27	1721.55	unbounded
	4	106.87	1823.04	149.70	2235.77	unbounded
	5	94.49	1823.04	169.31	2528.65	unbounded
	6	95.08	1823.04	168.27	2513.01	unbounded
	7	99.33	1823.04	161.07	2405.59	unbounded
	8	102.27	1823.04	156.43	2336.32	unbounded
	9	96.36	1823.04	166.03	2479.58	unbounded
	10	96.50	1823.04	165.79	2476.11	unbounded
	11	98.24	1823.04	162.86	2432.29	unbounded
	12	96.38	1823.04	166.00	2479.17	unbounded
	13	97.67	1823.04	163.80	2446.32	unbounded
	14	93.21	1823.04	171.64	2563.35	unbounded
	15	98.37	1823.04	162.63	2428.91	unbounded
	16	96.83	1823.04	165.23	2467.64	unbounded
updowng-40	1	55.61	138.72	287.67	326.92	40
	2	31.92	138.72	501.13	569.50	40
	3	20.59	138.72	776.88	882.87	40
	4	20.12	138.72	795.14	903.62	40
	5	18.48	138.72	865.38	983.45	40
	6	19.04	138.72	840.30	954.94	40
	7	19.04	138.72	840.16	954.78	40
	8	21.06	138.72	759.49	863.10	40
	9	22.09	138.72	724.07	822.86	40
	10	21.97	138.72	727.93	827.25	40

Continued on next page

Table A.1 – continued from previous page

Algorithm	# Threads	Duration (ms)	Size (kb)	Req/s	Mbit/s	Level
	11	20.64	138.72	775.04	880.78	40
	12	21.33	138.72	749.92	852.23	40
	13	21.29	138.72	751.20	853.68	40
	14	22.88	138.72	699.07	794.44	40
	15	23.01	138.72	695.34	790.20	40
	16	23.25	138.72	688.15	782.04	40
updowng-60	1	157.09	592.35	101.85	494.23	60
	2	81.58	592.35	196.12	951.69	60
	3	57.16	592.35	279.88	1358.14	60
	4	43.64	592.35	366.58	1778.87	60
	5	40.43	592.35	395.71	1920.23	60
	6	40.04	592.35	399.59	1939.07	60
	7	41.47	592.35	385.75	1871.89	60
	8	42.46	592.35	376.75	1828.24	60
	9	41.47	592.35	385.75	1871.91	60
	10	42.27	592.35	378.45	1836.46	60
	11	40.36	592.35	396.40	1923.56	60
	12	40.66	592.35	393.42	1909.13	60
	13	41.64	592.35	384.15	1864.14	60
	14	41.38	592.35	386.64	1876.23	60
	15	41.11	592.35	389.18	1888.52	60
	16	41.72	592.35	383.42	1860.58	60
updowng-80	1	255.55	1132.71	62.60	580.97	80
	2	129.90	1132.71	123.16	1142.88	80
	3	91.25	1132.71	175.32	1626.91	80
	4	72.59	1132.71	220.40	2045.16	80
	5	65.22	1132.71	245.31	2276.34	80
	6	68.82	1132.71	232.48	2157.31	80
	7	67.63	1132.71	236.56	2195.14	80
	8	64.65	1132.71	247.48	2296.44	80
	9	64.79	1132.71	246.94	2291.44	80
	10	67.36	1132.71	237.51	2203.97	80
	11	65.66	1132.71	243.65	2260.89	80
	12	64.10	1132.71	249.57	2315.83	80
	13	66.02	1132.71	242.31	2248.51	80
	14	66.94	1132.71	239.01	2217.87	80
	15	67.80	1132.71	235.98	2189.76	80
	16	65.92	1132.71	242.70	2252.14	80

Continued on next page

Table A.1 – continued from previous page

Algorithm	# Threads	Duration (ms)	Size (kb)	Req/s	Mbit/s	Level
updowng-100	1	328.44	1509.75	48.71	602.49	100
	2	175.11	1509.75	91.36	1130.05	100
	3	114.12	1509.75	140.19	1733.91	100
	4	91.63	1509.75	174.60	2159.54	100
	5	77.21	1509.75	207.20	2562.71	100
	6	87.88	1509.75	182.05	2251.68	100
	7	88.19	1509.75	181.42	2243.80	100
	8	86.88	1509.75	184.15	2277.59	100
	9	85.43	1509.75	187.26	2316.12	100
	10	88.49	1509.75	180.80	2236.18	100
	11	83.79	1509.75	190.94	2361.58	100
	12	85.93	1509.75	186.19	2302.87	100
	13	86.64	1509.75	184.65	2283.74	100
	14	84.30	1509.75	189.79	2347.39	100
	15	83.73	1509.75	191.06	2363.10	100
	16	87.45	1509.75	182.95	2262.74	100
updowng-120	1	360.25	1711.25	44.41	622.60	120
	2	184.99	1711.25	86.48	1212.44	120
	3	132.21	1711.25	121.01	1696.47	120
	4	98.81	1711.25	161.92	2269.93	120
	5	87.28	1711.25	183.30	2569.63	120
	6	94.59	1711.25	169.14	2371.21	120
	7	100.51	1711.25	159.17	2231.38	120
	8	98.57	1711.25	162.31	2275.43	120
	9	96.82	1711.25	165.25	2316.58	120
	10	96.56	1711.25	165.69	2322.75	120
	11	93.61	1711.25	170.92	2396.08	120
	12	95.55	1711.25	167.43	2347.23	120
	13	96.53	1711.25	165.73	2323.43	120
	14	93.99	1711.25	170.22	2386.32	120
	15	91.57	1711.25	174.72	2449.42	120
	16	94.60	1711.25	169.12	2370.84	120
sp	1	541.99	85.86	29.52	20.76	n.a.
	2	320.66	85.86	49.89	35.09	n.a.
	3	256.71	85.86	62.32	43.84	n.a.
	4	145.49	85.86	109.97	77.35	n.a.
	5	145.06	85.86	110.29	77.58	n.a.
	6	180.71	85.86	88.53	62.28	n.a.

Continued on next page

Table A.1 – continued from previous page

Algorithm	# Threads	Duration (ms)	Size (kb)	Req/s	Mbit/s	Level
	7	127.53	85.86	125.45	88.25	n.a.
	8	110.46	85.86	144.83	101.88	n.a.
	9	128.60	85.86	124.41	87.51	n.a.
	10	109.60	85.86	145.97	102.68	n.a.
	11	152.15	85.86	105.15	73.97	n.a.
	12	105.27	85.86	151.98	106.91	n.a.
	13	113.75	85.86	140.65	98.94	n.a.
	14	118.88	85.86	134.58	94.67	n.a.
	15	117.07	85.86	136.65	96.13	n.a.
	16	117.10	85.86	136.63	96.11	n.a.

Bibliography

[ADGW10] Abraham, Ittai ; Delling, Daniel ; Goldberg, Andrew V. ; Werneck, Renato F.: A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks. In: *Proceedings of the 10th International Symposium on Experimental Algorithms SEA11* (2010), Nr. v, 230–241. http://research.microsoft.com/pubs/142356/HL-TR.pdf. ISBN 9783642206610 (Cited on page 9)

[AFGW] Abraham, Ittai ; Fiat, Amos ; Goldberg, Andrew V. ; Werneck, Renato F.: Highway Dimension , Shortest Paths , and Provably Efficient Algorithms. (Cited on page 12)

[DGWN10] Delling, Daniel ; Goldberg, Andrew V. ; Werneck, Renato F. ; Nowatzyk, Andreas: PHAST : Hardware-Accelerated Shortest Path Trees. In: *2011 IEEE International Parallel Distributed Processing Symposium* (2010), Nr. MSR-TR-2010-125, 921–931. http://dx.doi.org/10.1109/IPDPS.2011.89. – DOI 10.1109/IPDPS.2011.89. – ISBN 9780769543857 (Cited on page 12)

[Dij59] Dijkstra, Edsger W.: A Note on Two Problems in Connexion with Graphs. In: *Numerische Mathematik* 1 (1959), Nr. 0029-599X, 269–271. http://dx.doi.org/10.1007/BF01386390. – DOI 10.1007/BF01386390. – ISSN 0029599X (Cited on pages 9 and 11)

[GSSD08] Geisberger, Robert ; Sanders, Peter ; Schultes, Dominik ; Delling, Daniel: Contraction Hierarchies : Faster and Simpler Hierarchical Routing in Road Networks. In: *Experimental Algorithms* 2 (2008), Nr. July, 319–333. http://www.springerlink.com/index/j062316602803057.pdf. ISBN 3540685480, (Cited on pages 9, 12 and 13)

All links were last followed on December 10, 2012.

Declaration

I declare herewith that I am the author of this bachelor's thesis. I did not use any other sources than those named and all those passages quoted from other works either literally or in the general sense are marked as such. Neither the complete thesis nor essential parts of it have been used up to now in any other examination procedure. I have not published this thesis up to now either in part or as a whole. The electronically submitted version is identical to all the other versions which have been submitted.

(Niklas Schnelle)