Time-Dependent Contraction Hierarchies*

G. Veit Batz, Daniel Delling, Peter Sanders, and Christian Vetter
Universität Karlsruhe (TH), 76128 Karlsruhe, Germany
{batz,delling,sanders,vetter}@ira.uka.de

December 17, 2008

Abstract

Contraction hierarchies are a simple hierarchical routing technique that has proved extremely efficient for static road networks. We explain how to generalize them to networks with time-dependent edge weights. This is the first hierarchical speedup technique for time-dependent routing that allows bidirectional query algorithms. For large realistic networks with considerable time-dependence (Germany, weekdays) our method outperforms previous techniques with respect to query time using comparable or lower preprocessing time.

1 Introduction

In recent years, there has been considerable work on routing in static road networks (see [8] for an overview). It is now possible to compute shortest path distances in road networks several orders of magnitude faster than with Dijkstra's algorithm. However, this does not mean that routing is a solved problem since constant edge weights are only a rough approximation of reality. In this paper, we look at the more advanced model where edge weights depend on time. This model is important since it models congestion effects in road networks and time tables in public transportation. More precisely, we look at the earliest arrival problem in networks with piece-wise linear weight functions and the FIFO property (no overtaking). These restrictions seem natural since for this model we can use a simple generalization of Dijkstra's algorithm [4].

The most successful speedup techniques for static networks exploit the hierarchical nature of transportation networks – only few edges are used for long-distance travel. Hierarchical query algorithms are typically bidirectional. They search forward from the source node and backward from the target node. For time-dependent networks, this approach looks problematic since we do not know the arrival time which would be necessary to run Dijkstra's algorithm backward. We show that bidirectional search is possible and successful anyway. We explain how contraction hierarchies (CHs) [10, 9] can be generalized to allow time-dependent edge weights (TCHs).

1.1 Related Work. Due to the difficulty of bidirectional routing, the first promising approaches to fast time-dependent routing used goal directed rather than hierarchical routing and accepted suboptimal routes [15].

Schultes [17] gives a way to make hierarchical queries in static networks unidirectional but this approach does not directly yield a time-dependent approach. The basic ideas behind time-dependent CHs were released in a short technical report [2].

SHARC routing [3] was specifically developed to encode hierarchical information into a goal-directed framework based on *arc-flags* [14, 13] allowing unidirectional search and recently was generalized to exact time-dependent routing [6]. Similar to CHs, SHARC also uses node contraction but invests less work in keeping the graph sparse. The consequence is that contraction has to stop at a *core* to keep the graph sufficiently sparse.

Very recently, the simple contraction routing from SHARC was combined with the goal-directed technique *landmark A* (ALT)* [11] to obtain *T(ime)D(ependent)C(ore)ALT* [7].

1.2 Overview and Results. After introducing known basic concepts in Section 2, Section 3 explains how a TCH is constructed. While the basic definition is straightforward, some sophistication is needed to achieve acceptable preprocessing times. To this end, we use static searches to prune the search space of the required local time-dependent searches. Section 4 explains the query algorithm. Here, the main challenge is to implement the backward search without knowing the arrival time. What saves us is that contraction hierarchies define acyclic search graphs that imply limited search spaces even when all reachable nodes are

*Partially supported by DFG grants SA 933/5-1, WA 654/16-1, a Google Research Award, and by EU project ARRIVAL (contract no. FP6-021235-2)

considered. Again, this search space can be pruned using static searches. Several important refinements are explained in Section 5: Node ordering can take time-dependence into account by making the node ordering algorithm itself time-dependent. Again, this is straightforward in principle but a naive implementation would be prohibitively slow. We also explain how to output paths. The experiments in Section 6 use commercial data with realistic traffic patterns for Germany. It turns out that our approach is largely the best currently available technique with respect to query time and preprocessing time. Space consumption is high but affordable for current server technology. Section 7 summarizes the results and discusses future improvements.

2 Preliminaries

Contraction Hierarchies. We are dealing with a graph $G = (V, E)$ with n nodes and m edges. CHs first order the nodes by increasing importance. From now on, we assume that $V = 1..n$ where node 1 is the least important node.

A CH is constructed by *contracting* the nodes in the above order. Contracting a node v means removing v from the graph without changing shortest path distances between the remaining (more important) nodes. A trivial way to achieve this is to introduce a *shortcut* edge (u, w) for every path of the form $\langle u, v, w\rangle$. In order to keep the graph sparse, we can try to avoid a shortcut (u, w) by finding a *witness* – a path different from $P = \langle u, v, w\rangle$ that is no longer than P. This can be achieved by performing a local search from each *in-node* u (i.e, u has an edge into v) to find shortest paths to the *out-nodes* w (that are reached by edges leaving v). Note that we are free to break the local searches early without compromising correctness – we will just get some superfluous shortcuts. One useful stopping rule bounds the number of edges (hops) on a path explored by a local search. To compensate, *on-the-fly edge reduction* is used: After a local search from u, we check all its outgoing edges (u, w); if the search found a path $\langle u, \ldots, w\rangle$ shorter than the edge (u, w) then this edge can be removed.

The node ordering is computed by simulating the contraction process. To this end, the nodes which have not yet been contracted are kept in a priority queue. In each step, the "least cost" node is removed from the queue and contracted. The cost function takes various issues into account. Perhaps the most important measure for static routing is the *edge difference* between the number of shortcuts introduced by a contraction and the number of adjacent edges that would be removed. Other important terms ensure that the network is contracted everywhere in a reasonably uniform way, and estimate the cost for contraction and queries.

The CH *query* algorithm finds a shortest path in the CH from source s to the target t by performing a bidirectional Dijkstra search in the graph with the following modifications: (1) Only edges leading to more important nodes are relaxed. (2) The search can only be stopped in a direction D when the best s–t-path seen so far is shorter than the smallest priority queue entry for direction D. (3) Search need not be continued from nodes w for which the query does not find an optimal distance in the forward or backward search tree. This can happen if the shortest path to w goes via a more important node $v > w$. The *stall-on-demand* technique identifies such nodes w by checking whether (downward) edges coming into w from more important nodes give shorter paths than the (upward) Dijkstra search.

The path computed by a CH-query contains shortcuts that have to be unpacked to actually output a path in the original graph. This can be done efficiently by a recursive routine if we store the middle node v with each shortcut $\langle u, v, w\rangle$.

Modelling. We consider time-dependent networks where the objective function is travel time and edges have a weight function $f(t)$ that specifies the travel time at the endpoint of the edge when the edge is entered at time t. We assume that all edge cost functions have the *FIFO-property*: $\forall \tau < \tau' : \tau + f(\tau) \leq \tau' + f(\tau')$. We focus on this case and further assume that the travel time functions are representable by a piece-wise linear function. However, all our algorithms view travel-time functions (TTFs) as an abstract data type with a small number of operations: *Evaluation* returns $f(t)$. Using a sorted array of interpolation points, evaluation can be implemented to run in time $O(\log |f|)$ where $|f|$ specifies the number of line segments defining f. Actually, we use a bucket representation with $\Theta(|f|)$ buckets. Thus, we can find the appropriate bucket in constant time and then just scan one bucket. Since the average bucket size is constant, we get average case constant time for evaluating the function. *Chaining* computes a piece-wise linear representation of the travel time function $f_{uw}(t) = f_{vw}(t + f_{uv}(t))$ for a path $\langle u, v, w\rangle$. This takes time $O(|f_{uv}| + |f_{vw}|)$ and produces a function with $|f_{uw}| \leq |f_{uv}| + |f_{vw}| - 1$. For road networks it looks like the actual complexity is often close to this upper bound. Similarly, we sometimes need the *minimum* of two piece-wise linear functions. This operation has similar worst-case complexity as chaining.

It seems that any exact time-dependent preprocessing technique needs a basic ingredient that computes travel times not only for a point in time but for an entire *time-interval*. An easy way to implement this *profile* query is a generalization of Dijkstra's algorithm to

profiles [12]. Tentative distances then become TTFs. Adding edge weights is replaced by chaining TTFs and taking the minimum takes the minimum of TTFs. Unfortunately, the algorithm loses its label-setting property. However, the performance as a label-correcting algorithm seems to be good in important practical cases.

3 Construction

The most expensive preprocessing phase of static CHs orders the nodes by importance. For a first version we adopt the *static* algorithm for time-dependent CHs (TCHs).[1] This decision is based on the assumption that averaged over the planning period, the importance of a node is not heavily affected by its exact traffic pattern.

The second stage of CH-preprocessing – contraction – is in principle easy to adapt to time-dependence: When contracting node v, we are given a current (time-dependent) graph with node set $v..n$. For every combination of an incoming edge (u, v) and an outgoing edge (v, w) we have to decide whether the path $\langle u, v, w\rangle$ may be a unique shortest path *at any point in time*. If so, we have to insert the shortcut (u, w). The weight function of this shortcut can be computed by chaining the weight functions of its constituents.

The starting point for our implementation of contraction is to perform a separate *profile-Dijkstra* for every combination of in-node u and out-node w of v. This search can be stopped when the minimum of the current distance label exceeds the minimum of shortcut function. We also have a hop-limit similar to the static case. If the resulting travel-time function is dominated by the travel-time function of the shortcut, we have found a witness and need not insert the shortcut.[2] Introducing shortcuts may lead to parallel edges which are replaced by a single edge whose edge weight function is the minimum of the original edges.[3] Similar to the static case, we use information gained during profile searches for on-the-fly edge reduction. Additionally, we use *aggressive edge reduction*: From time to time we go through all remaining nodes, performing hop-limited local searches to identify superfluous edges.

The apparently wasteful pairwise approach to witness search is made efficient using preparatory static searches that only take the minimum and the maximum travel time over the considered edges into account. This search yields three kinds of information:

1. If the static search finds a witness whose maximal travel time is smaller than the minimal travel time of the shortcut, we do not need the shortcut for sure – no profile search is needed.

2. If the static search does not find a path whose lower bound on travel time is smaller than the minimal travel time of the shortcut, the shortcut is certainly needed. Again no profile search is needed.

3. Otherwise, we can derive a *corridor* of nodes that appear useful for a witness path from u to w bypassing v. Only the corridor is searched with a profile search. It turned out that a very restrictive heuristics for defining the corridor gives good results: We only consider nodes on paths from u to w that solely consist of edges which define the minimum or maximum distance label of a node in the search space of the static search. Note that with this heuristics we may miss some witnesses. But this does not affect the correctness of the computed TCH. Furthermore, superfluous edges will often be eliminated later using edge reduction techniques.

4 Query

The basic static query algorithm for CHs consists of a forward search in an upward graph $G_\uparrow = (V, E_\uparrow)$ and a backward search in a downward graph $G_\downarrow$. Wherever these searches meet, we have a candidate for a shortest path. The shortest such candidate is a shortest path.

Since the departure time is known, the forward search is easy to generalize. In particular, the only overhead compared to the static case is that we have to evaluate each relaxed edge for one point in time. In our experience with a plain time-dependent Dijkstra, this means only a small constant factor overhead in practice.

The most easy way to adapt the backward search is to explore *all* nodes that can *reach* t in $G_\downarrow$. During this exploration we mark all edges connecting nodes that can reach t. Let E_{marked} denote the set of marked edges.

Now, we can perform an s–t-query by a forward search from s in $(V, E_\uparrow \cup E_{\text{marked}})$. This procedure is guaranteed to find the shortest path for reasons analogous to the correctness of static CHs [9]. Roughly, the properties of TCHs imply that there must be a shortest path P in the TCH that consists of two segments: One using only edges in $G_\uparrow$ leading to a peak node v_p and one connecting v_p to t in $G_\downarrow$. Since all edges of P are in the search space of our forward search, this path or some other shortest path will be found.

[1] Robert Geisberger has provided us with a slightly modified version that takes a new estimate of query complexities into account. We use the minima of the weight-functions as static edge weights.

[2] Our initial implementation used unidirectional profile searches from the in-nodes, analogous to the static code. But somewhat astonishingly, even a sophisticated variant of this was much slower than our new solution.

[3] So far we do not exploit that a shortcut may only be needed during certain time intervals.

Our current query algorithm is somewhat more sophisticated. We perform the backward exploration using a static Dijkstra algorithm with edge weights equal to a lower bound on the travel time. Thus, we obtain lower bounds for the travel time to t. Furthermore, in the same run of Dijkstra, we compute upper bounds on travel time. Using these bounds we can prune the search at some nodes using a method similar to stall-on-demand in the static case. Our forward search initially only explores $G_\uparrow$ and it is interspersed with the backward search in a similar way as in static bidirectional search algorithms. When a node v is settled from both sides, we get two kinds of information: First, we obtain an upper bound for the travel time from s to t – we remember the previously best such bound U. Second, we get a lower bound L for the travel time from s to t via v. If $L \geq U$, we prune (i.e., discontinue) the forward search from v. Once this bidirectional search is finished, we continue the forward search on the edges in E_{marked}, still using the above pruning rule. The most important effect of pruning is that the downward part of the search is funnelled into the direction of t.

5 Refinements

5.1 Node Ordering. Since node ordering is essentially a simulated contraction process, it is in principle easy to adapt to the time-dependent setting. However, new terms in the cost function used in the priority queue for contraction make sense. Our current, preliminary cost function for the priority is quite simple and contains only three terms. The edge difference already described in Section 2, a similarly defined *segment difference* that measures the change in the number of line segments needed to define the time-dependent edge weight functions of the edges involved, and a term q that affects query time and uniformity of contraction. Initially, each node has $q = 0$. When a node v is contracted, a neighbor u of v computes $q(u) := \max(q(u), q(v) + \mathsf{segments}(u, v))$ where $\mathsf{segments}(u, v)$ denotes the number of segments in the time dependent edge weight of edge (u, v). Since node ordering may simulate the contraction of a node several times, we use a cache remembering the results (witness found? yes/no) of all witness searches that have already been performed.

5.2 Outputting Paths. Similar to static CHs, we can recursively unpack shortcuts. Since parallel shortcuts are reduced to single shortcuts during contraction, there is no longer a unique middle point for a shortcut. Rather, the middle point also depends on time. Hence, rather than storing a single middle point as in static CHs, we now store a middle-point profile specifying time intervals during which a middle-point is valid.

6 Experiments

The experimental evaluation was done on two machines with two Xeon 5345 processors clocked at 2.33 GHz with 16 GB of RAM and 2x2x4MB of Cache running SUSE Linux 10.3. We always used one core at a time.

The program was compiled with GCC 4.3.2, using optimization level 3. All figures refer to the scenario that only the lengths of the shortest paths have to be determined, without outputting a complete description of the paths. Note, that the static node ordering was compiled with GCC 4.2.1 and also optimization level 3.

The experiments with the European network have been done on another machine (because of memory shortage): a machine with four Dual-Core AMD Opteron Processors 2218 with 4x1MB Cache and 32 GB of RAM also running SUSE 10.3. We also used one core at a time. The compiler was a GCC 4.2.1, using optimization level 3.

Inputs. We use real-world time-dependent road networks of Germany as our main input. They reflect five different traffic scenarios, collected from historical data: Monday, midweek (Tuesday till Thursday), Friday, Saturday, and Sunday. As expected, road congestion is higher during the week than on the weekend: $\approx 8\%$ of the edges are time-dependent for Monday, midweek, and Friday. The corresponding figures for Saturday and Sunday are $\approx 5\%$ and $\approx 3\%$, respectively. All these inputs have approximately 4.7 million nodes and 10.8 million edges and has been provided by PTV AG for scientific use.

We have made selected additional experiments with a Western European network, provided by PTV AG for scientific use. The network has been augmented with synthetic time-dependent travel times using the methodology from [15, 7, 6]. We use a medium amount of time-dependence where motorways and national highways have time-dependent edge weights. Roughly, this amount of time-dependence is comparable to the Sunday traffic in the German network. The graph has approximately 18 million nodes and 42.6 million edges.

When not otherwise mentioned, we measure average query performance for 1 000randomly selected source-destination pairs.

6.1 Contraction. Figures 1, 2, and 3 show the progress of the contraction process (see also [10, 9]) for the German network (midweek and Sunday scenario, each for the time-dependent and the static node ordering). Figure 1 shows how the running time of a node contraction increases over time (espacialy for the nodes contracted later).

Figures 2 and 3 show the average node degree and the average complexity of the weight-functions of the

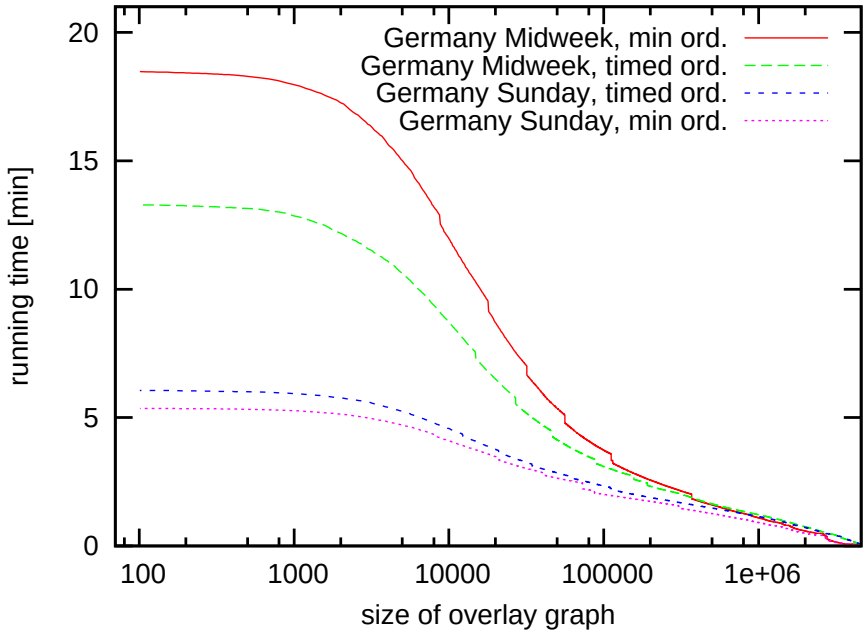

Figure 1: How the running time evolves over the contraction process for timed and static node ordering (the latter uses the minima of the weight-functions as static weights). The size of the overlay indicates the number of nodes not yet contracted.

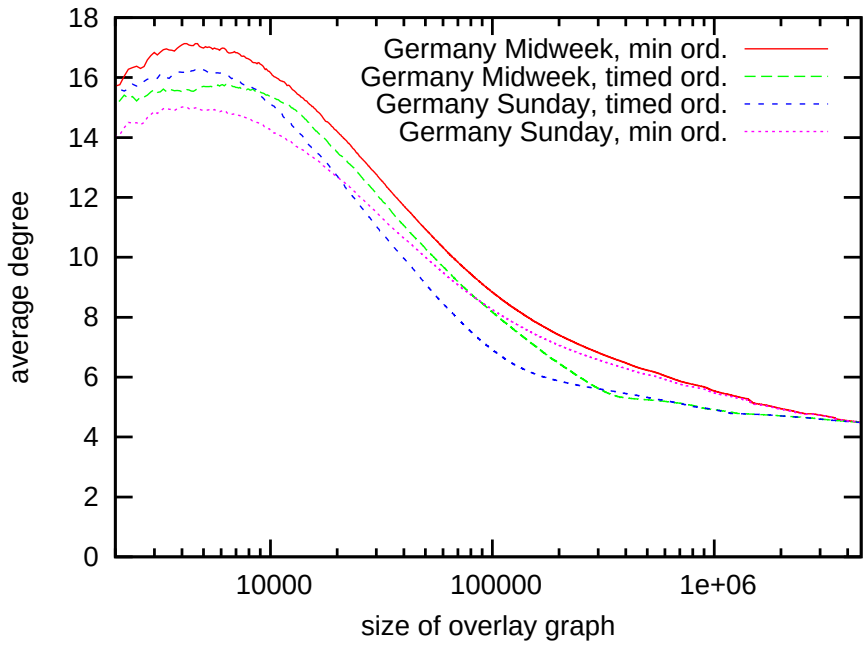

Figure 2: Like Figure 1 but showing the average degree of the nodes not yet contracted.

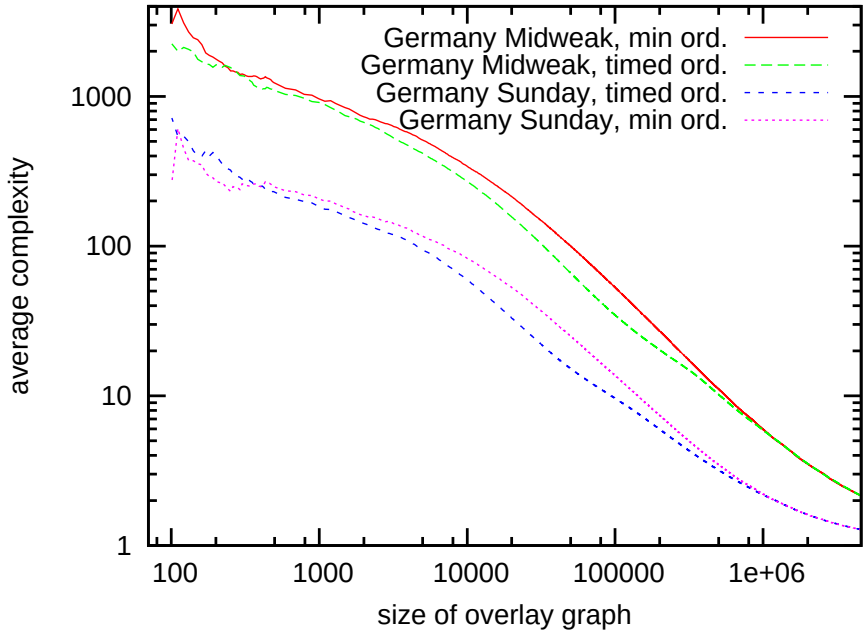

Figure 3: Like Figures 1 and 2 but showing the average complexity of the weight-functions of the part of the graph that is not contracted yet.

part of the graph that is not contracted yet. The average degree of the nodes remains remarkably small, indeed comparable to the value observed for static CHs. This comes as a positive surprise since time-dependence leads to a larger spectrum of relevant routes. However, the average complexity of the weight-functions increases dramatically towards the end of the contraction process. Regarding space consumption this is more expensive than we might hope but affordable on modern servers. Also the query time hardly depends on the complexity of the functions. Unfortunately, the effect on the running time of the contraction process is dramatic – contracting the last 18000 nodes (0.4% of the input) takes half of the time.

6.2 Average Performance. Table 1 summarizes the performance of several variants of TCHs on five different traffic patterns for Germany. We obtain query times around 1ms. Less for the low traffic on a Sunday and more for the high traffic on weekday. Our time-dependent node ordering heuristics is currently only optimized for the high traffic scenario. Here it yields small improvements for the query time and about 25 % reduction in the preprocessing time (not counting node ordering itself) and space. Of course, time-dependent node ordering is most attractive when we do not have to redo it every time the preprocessing is executed. To validate this assumption, we show in the lines with ordering type "Monday" what happens if a time-dependent node ordering for the Monday traffic pattern is applied to a different input. We see that this indeed has little effect on the performance. The rightmost columns show performance for simplified query algorithm where backward search only uses BFS exploration of the search space. No distance information is computed during backward search and no pruning of the search space is done where forward and backward search meet. The query time increases by more than a factor of four for weekdays but is still reasonably good.

6.3 Local Queries. In Figure 4 we take a closer look at the distribution of query times using the well-established methodology from [16]. This plot is for the European network. For $i \in 8..24$ we look at 1 000 queries (per rank) with the property that the time-dependent Dijkstra-Algorithm settles 2^i nodes. We see that the algorithm is well-behaved in the sense that local queries (which dominate in many applications) are considerably more efficient than the global averages from Table 1.

6.4 Comparison. Finally, in Table 2, we compare TCHs with the best other existing approaches. More precisely, we report the performance of various SHARC [6, 5] variants, TDCALT [7], and TCHs (see also Section 1.1).

For the more difficult (and perhaps more interest-

Table 1: Overview of performance of TCHs for different traffic patterns, node orderings, and query strategies.

		Contr.			Queries			bfs-Queries	
input	type of ordering	ordering [h:m]	const. [h:m]	space [B/n]	#delete mins	time [ms]	speed up	time [ms]	speed up
Monday	static min	0:05	0:20	1 035	518	1.19	1 240	5.04	294
	static max	0:05	0:19	1 133	529	1.22	1 215	4.70	315
	timed	1:47	0:14	750	546	1.19	1 244	4.40	337
midweek	static min	0:05	0:20	1 029	528	1.22	1 212	5.17	287
	static max	0:05	0:19	1 122	547	1.26	1 180	4.76	312
	timed	1:48	0:14	743	551	1.19	1 242	4.46	333
	Monday	–	0:15	771	561	1.24	1 198	4.81	308
Friday	static min	0:05	0:16	856	497	1.11	1 381	4.57	336
	static max	0:05	0:15	929	511	1.14	1 346	4.27	359
	timed	1:30	0:12	620	526	1.13	1 362	4.24	362
	Monday	–	0:12	640	532	1.14	1 350	4.26	360
Saturday	static min	0:05	0:08	391	428	0.81	1 763	2.97	484
	static max	0:05	0:08	428	441	0.85	1 680	3.15	456
	timed	0:52	0:08	282	529	1.09	1 313	4.44	324
	Monday	–	0:08	308	479	0.89	1 606	3.27	440
Sunday	static min	0:05	0:06	248	407	0.71	1 980	2.43	580
	static max	0:05	0:06	272	420	0.75	1 877	2.71	519
	timed	0:38	0:07	177	541	1.07	1 321	4.45	317
	Monday	–	0:06	203	463	0.80	1 756	2.87	491

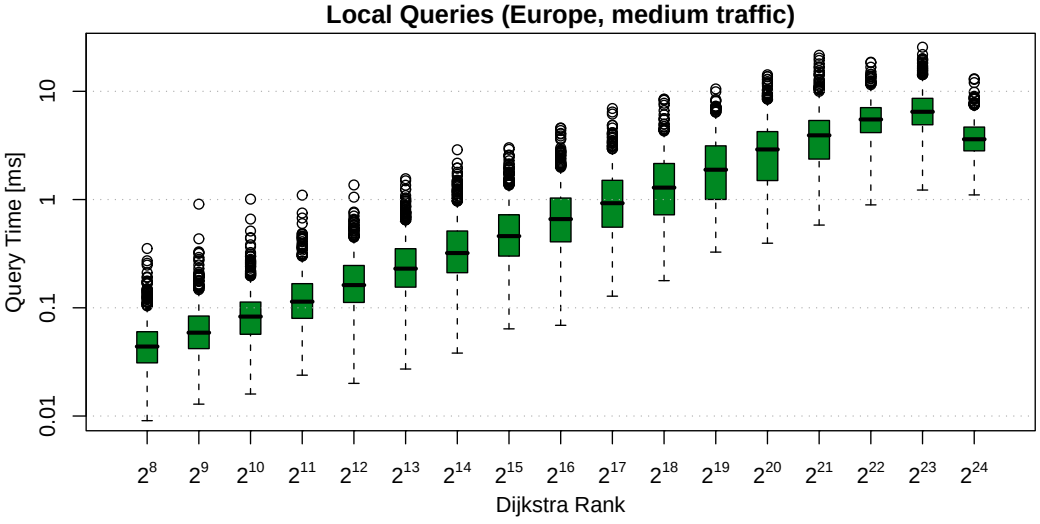

Figure 4: Query time distribution as a function of the Dijkstra-rank of the queries.

Table 2: Performance of Core-ALT and SHARC compared to approaches based on Contraction Hierarchies.

		PREPRO		QUERIES			
		time	space	#delete	speed	time	speed
input	algorithm	[h:m]	[B/n]	mins	up	[ms]	up
Germany midweek	TDCALT	0:09	50	3 190	723	5.36	280
	eco SHARC	1:16	155	19 425	119	25.06	60
	eco L-SHARC	1:18	219	2 776	831	6.31	238
	TCH timed ord.	1:48 + 0:14	743	551	4 356	1.19	1 242
	TCH min ord.	0:05 + 0:20	1 029	528	4 546	1.22	1 212
Germany Sunday	TDCALT	0:05	19	1 773	1 325	2.13	688
	eco SHARC	0:30	65	2 142	1 097	1.86	787
	eco L-SHARC	0:32	129	576	4 076	0.73	2 011
	agg SHARC	27:20	61	670	3 504	0.50	2 904
	agg L-SHARC	27:22	125	283	8 300	0.29	5 045
	TCH timed ord.	0:38 + 0:07	177	541	4 436	1.07	1 321
	TCH min ord.	0:05 + 0:06	248	407	5 896	0.71	1 980
Europe medium	eco SHARC	4:37	43	42 776	210	42.75	132
	TCH min ord.	0:21 + 3:11	309			3.45	1 592

ing) midweek scenario, we see that SHARC is dominated by TDCALT in every respect. This is astonishing, since TDCALT is much simpler than SHARC. TDCALT is about twice as fast as TCHs with respect to preprocessing time but more than four time slower with respect query time. TDCALT needs much less space than TCHs.

For the Sunday scenario, we can afford to run SHARC with exact arc-flag computation. This yields the best overall query time however at the cost of more than a day of preprocessing. TCHs have only slightly worse query times but two order of magnitude smaller preprocessing times.

For the European network, SHARC has somewhat better preprocessing time yet is seven time slower than TCHs. Unfortunately, we have data for TDCALT but it is to be expected that it has very good preprocessing time yet query times even higher than SHARC – the larger the network, the more important is a speedup technique that can exploit hierarchy also globally.

In [9], the product of preprocessing time and query time was used as a way to compare speedup techniques. In this respect TCHs considerably outperform the previous techniques albeit at the cost of much higher space consumption.

7 Conclusions and Future Work

We have demonstrated that the earliest arrival problem in large time-dependent road networks can be solved exactly and very efficiently with contraction hierarchies (TCHs) – a purely hierarchical speedup technique. In particular, it turns out that the backward search part is no significant problem because the hierarchical aspects of the networks even work when we perform a plain reachability search in the backward search graph. From a 'historical' point of view, it is interesting to note that the query times we achieve are faster than the best static routing techniques three years ago. This would not have been possible, if research on static techniques would have stopped in 2006 on the grounds that the methods existing at that point were already "more than fast enough".

Although we believe that TCHs are ready for time-dependent routing in server based systems, it is interesting to discuss how the three main performance aspects of a speedup technique could be further improved:

Space Consumption. Perhaps the main weakness of TCHs are their large space consumption (up to a kilobyte per node). We believe that this problem can be mitigated by working with approximate upper and lower bounds for the travel time functions. In [2] we explain how this information can be used to obain *exact* query results nevertheless. For a mobile implementation even more radical space reductions are needed. Such savings might be possible if we drop the requirement of exact queries which is an illusion anyway.

Preprocessing Time. There is an entire catalog of further optimizations for improved preprocessing times including limiting shortcuts to time ranges where they form shortest paths, more sophisticated node ordering heuristics, faster witness search, use of approximations within witness searches, and parallelization, e.g., by identifying sets of nodes that can be contracted independently.

Query Time. Better node ordering heuristics can have some impact on query time but otherwise, the main source of improvements may be the combination with other speedup techniques. We already have evidence that a combination with arc-flags [14, 13] can yield several times better query performance at a moderate increase of preprocessing time in particular in low traffic scenarios. We might also use TCHs to implement the preprocessing for a time-dependent variant of transit-node routing [1].

Further Issues. We want to apply TCHs to public transportation networks and we want to use them for profile-queries. Beyond that, we eventually want a further generalization of the model including more flexible objective functions and a fast and realistic integration of dynamic changes of edge weights, e.g., due to traffic jams. For realistic results, we would like to be able to estimate the reaction of other drivers on the traffic delays, e.g., using a game-theoretic approach.

Simplicity. Opposite to further sophistication, it is interesting for commercial applications of TCHs how we can simplify our results. Note that the basic idea behind TCHs is quite easy. Already a very basic version can yield significant speedup with small preprocessing times if the contraction process is aborted before things get really difficult. This way TCHs become closer to TDCALT albeit still using a more sophisticated contraction routing which makes the ALT component optional. Considering the results for TDCALT, this version might result in method with the fast preprocessing of TDCALT but with better query performance. Further simplifications are possible if we are content with approximate results.

Acknowledgements. We would like to thank Robert Geisberger for his help with some of his CH code we adapted and with his help in adapting the static node ordering routine to yield good orderings for TCHs. We also thank Giacomo Nannicini for providing us with his time-dependent edge-cost generator.

References

[1] H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast routing in road networks with transit nodes. *Science*, 316(5824):566, 2007.

[2] G. V. Batz, R. Geisberger, and P. Sanders. Time dependent contraction hierarchies – basic algorithmic ideas. Technical report, Universität Karlsruhe, 2008. `arXiv:0804.3947v1[cs.DS]`.

[3] R. Bauer and D. Delling. SHARC: Fast and Robust Unidirectional Routing. In *Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX'08)*, pages 13–26. SIAM, 2008.

[4] K. Cooke and E. Halsey. The shortest route through a network with time-dependent intermodal transit times. *Journal of Mathematical Analysis and Applications*, 14:493–498, 1966.

[5] D. Delling. *Engineering and Augmenting Route Planning Algorithms*. PhD thesis, 2008. submitted.

[6] D. Delling. Time-Dependent SHARC-Routing. In *Proceedings of the 16th Annual European Symposium on Algorithms (ESA'08)*, volume 5193 of *Lecture Notes in Computer Science*, pages 332–343. Springer, Sept. 2008.

[7] D. Delling and G. Nannicini. Bidirectional Core-Based Routing in Dynamic Time-Dependent Road Networks. In *Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC'08)*, volume 5369 of *Lecture Notes in Computer Science*, pages 813–824. Springer, Dec. 2008.

[8] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering Route Planning Algorithms. In J. Lerner, D. Wagner, and K. A. Zweig, editors, *Algorithmics of Large and Complex Networks*, Lecture Notes in Computer Science. Springer, 2009. to appear. Online available at `http://i11www.iti.uni-karlsruhe.de/members/delling/files/dssw-erpa-09.pdf`.

[9] R. Geisberger. Contraction hierarchies: Faster and simpler hierarchical routing in road networks, 2008. Diploma Thesis, Universität Karlsruhe.

[10] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, June 2008.

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

[12] D. E. Kaufman and R. L. Smith. Fastest paths in time-dependent networks for intelligent vehicle-highway systems application. *Journal of Intelligent Transportation Systems*, 1(1):1–11, 1993.

[13] E. Köhler, R. H. Möhring, and H. Schilling. Acceleration of Shortest Path and Constrained Shortest Path Computation. In *Proceedings of the 4th Workshop on Experimental Algorithms (WEA'05)*, Lecture Notes in Computer Science, pages 126–138. Springer, 2005.

[14] U. Lauther. An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. In *Geoinformation und Mobilität – von der Forschung zur praktischen Anwendung*, volume 22, pages 219–230. IfGI prints, Institut für Geoinformatik, Münster, 2004.

[15] G. Nannicini, D. Delling, L. Liberti, and D. Schultes. Bidirectional A* Search for Time-Dependent Fast Paths. In *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture*

Notes in Computer Science, pages 334–346. Springer, June 2008.

[16] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In *13th European Symposium on Algorithms (ESA)*, volume 3669 of *LNCS*, pages 568–579. Springer, 2005.

[17] D. Schultes. *Route Planning in Road Networks*. PhD thesis, 2008.